ECL Language Reference

Introduction ................................................................................................................. 9
  Documentation Structure ...................................................................................... 9
  Documentation Conventions ................................................................................. 10

ECL Basics .................................................................................................................. 11
  Overview ................................................................................................................. 11
  Constants .................................................................................................................. 12
  Environment Variables ......................................................................................... 16
  Definitions .............................................................................................................. 17
  Basic Definition Types ......................................................................................... 19
  Recordset Filtering ............................................................................................... 22
  Function Definitions (Parameter Passing) .............................................................. 23
  Definition Visibility .............................................................................................. 28
  Field and Definition Qualification ...................................................................... 30
  Actions and Definitions ....................................................................................... 32

Expressions and Operators ....................................................................................... 33
  Expressions and Operators ................................................................................. 33
  Logical Operators ............................................................................................... 35
  Record Set Operators ......................................................................................... 36
  Set Operators ....................................................................................................... 38
  String Operators ................................................................................................. 39
  IN Operator .......................................................................................................... 40
  BETWEEN Operator ............................................................................................. 41

Value Types .............................................................................................................. 42
  BOOLEAN ............................................................................................................... 42
  INTEGER ............................................................................................................... 43
  REAL ..................................................................................................................... 44
  DECIMAL ............................................................................................................... 45
  STRING .................................................................................................................. 46
  QSTRING .............................................................................................................. 47
  UNICODE .............................................................................................................. 48
  UTF8 ....................................................................................................................... 49
  DATA ....................................................................................................................... 50
  VARSTRING .......................................................................................................... 51
  VARUNICODE ........................................................................................................ 52
  SET OF ................................................................................................................... 53
  TYPEOF ................................................................................................................ 54
  RECORDOF .......................................................................................................... 55
  ENUM ...................................................................................................................... 56
  Type Casting ......................................................................................................... 57

Record Structures and Files ...................................................................................... 60
  RECORD Structure .............................................................................................. 60
  DATASET ................................................................................................................ 70
  DICTIONARY ......................................................................................................... 85
  INDEX ..................................................................................................................... 87
  Scope and Logical Filenames .............................................................................. 92
  Implicit Dataset Relationality ............................................................................ 95

Alien Data Types .................................................................................................... 96
  TYPE Structure .................................................................................................... 96
  TYPE Structure Special Functions .................................................................... 97

Parsing Support ....................................................................................................... 99
  Parsing Support .................................................................................................... 99
  PARSE Pattern Value Types .............................................................................. 100
  NLP RECORD and TRANSFORM Functions ...................................................... 104
  XML Parsing RECORD and TRANSFORM Functions ....................................... 106

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
### Reserved Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>108</td>
</tr>
<tr>
<td>EXCEPT</td>
<td>109</td>
</tr>
<tr>
<td>EXPORT</td>
<td>110</td>
</tr>
<tr>
<td>GROUP keyword</td>
<td>111</td>
</tr>
<tr>
<td>IMPORT</td>
<td>112</td>
</tr>
<tr>
<td>KEYED and WILD</td>
<td>114</td>
</tr>
<tr>
<td>LEFT and RIGHT</td>
<td>116</td>
</tr>
<tr>
<td>LIKELY and UNLIKELY</td>
<td>117</td>
</tr>
<tr>
<td>ROWS(LEFT) and ROWS(RIGHT)</td>
<td>118</td>
</tr>
<tr>
<td>SELF</td>
<td>119</td>
</tr>
<tr>
<td>SHARED</td>
<td>120</td>
</tr>
<tr>
<td>SKIP</td>
<td>121</td>
</tr>
<tr>
<td>TRUE and FALSE</td>
<td>122</td>
</tr>
</tbody>
</table>

### Special Structures

<table>
<thead>
<tr>
<th>Structure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGINC++ Structure</td>
<td>124</td>
</tr>
<tr>
<td>EMBED Structure</td>
<td>131</td>
</tr>
<tr>
<td>FUNCTION Structure</td>
<td>133</td>
</tr>
<tr>
<td>FUNCTIONMACRO Structure</td>
<td>136</td>
</tr>
<tr>
<td>INTERFACE Structure</td>
<td>138</td>
</tr>
<tr>
<td>MACRO Structure</td>
<td>141</td>
</tr>
<tr>
<td>MODULE Structure</td>
<td>143</td>
</tr>
<tr>
<td>TRANSFORM Structure</td>
<td>146</td>
</tr>
</tbody>
</table>

### Built-in Functions and Actions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>150</td>
</tr>
<tr>
<td>ACOS</td>
<td>151</td>
</tr>
<tr>
<td>AGGREGATE</td>
<td>152</td>
</tr>
<tr>
<td>ALLNODES</td>
<td>153</td>
</tr>
<tr>
<td>APPLY</td>
<td>156</td>
</tr>
<tr>
<td>ASCII</td>
<td>157</td>
</tr>
<tr>
<td>ASIN</td>
<td>158</td>
</tr>
<tr>
<td>ASSERT</td>
<td>159</td>
</tr>
<tr>
<td>ASSTRING</td>
<td>160</td>
</tr>
<tr>
<td>ATAN</td>
<td>162</td>
</tr>
<tr>
<td>ATAN2</td>
<td>163</td>
</tr>
<tr>
<td>AVE</td>
<td>164</td>
</tr>
<tr>
<td>BUILD</td>
<td>165</td>
</tr>
<tr>
<td>CASE</td>
<td>166</td>
</tr>
<tr>
<td>CATCH</td>
<td>174</td>
</tr>
<tr>
<td>CHOOSE</td>
<td>175</td>
</tr>
<tr>
<td>CHOOSELEN</td>
<td>177</td>
</tr>
<tr>
<td>CHOOSESETS</td>
<td>178</td>
</tr>
<tr>
<td>CLUSTERSIZE</td>
<td>179</td>
</tr>
<tr>
<td>COMBINE</td>
<td>180</td>
</tr>
<tr>
<td>CORRELATION</td>
<td>181</td>
</tr>
<tr>
<td>COS</td>
<td>184</td>
</tr>
<tr>
<td>COSH</td>
<td>186</td>
</tr>
<tr>
<td>COUNT</td>
<td>187</td>
</tr>
<tr>
<td>COVARIANCE</td>
<td>188</td>
</tr>
<tr>
<td>CRON</td>
<td>190</td>
</tr>
<tr>
<td>DEDUP</td>
<td>192</td>
</tr>
<tr>
<td>DEFINE</td>
<td>193</td>
</tr>
<tr>
<td>DENORMALIZE</td>
<td>196</td>
</tr>
<tr>
<td>DISTRIBUTE</td>
<td>197</td>
</tr>
<tr>
<td>EXP</td>
<td>200</td>
</tr>
<tr>
<td>ECL Language Reference</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td></td>
</tr>
<tr>
<td>DISTRIBUTED</td>
<td>203</td>
</tr>
<tr>
<td>DISTRIBUTION</td>
<td>204</td>
</tr>
<tr>
<td>EBCDIC</td>
<td>206</td>
</tr>
<tr>
<td>ENTH</td>
<td>207</td>
</tr>
<tr>
<td>ERROR</td>
<td>208</td>
</tr>
<tr>
<td>EVALUATE</td>
<td>209</td>
</tr>
<tr>
<td>EVENT</td>
<td>211</td>
</tr>
<tr>
<td>EVENTNAME</td>
<td>212</td>
</tr>
<tr>
<td>EVENTEXTRA</td>
<td>213</td>
</tr>
<tr>
<td>EXISTS</td>
<td>214</td>
</tr>
<tr>
<td>EXP</td>
<td>216</td>
</tr>
<tr>
<td>FAIL</td>
<td>217</td>
</tr>
<tr>
<td>FAILCODE</td>
<td>218</td>
</tr>
<tr>
<td>FAILMESSAGE</td>
<td>219</td>
</tr>
<tr>
<td>FETCH</td>
<td>220</td>
</tr>
<tr>
<td>FROMJSON</td>
<td>222</td>
</tr>
<tr>
<td>FROMUNICODE</td>
<td>223</td>
</tr>
<tr>
<td>FROMXML</td>
<td>224</td>
</tr>
<tr>
<td>GETENV</td>
<td>225</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>226</td>
</tr>
<tr>
<td>GRAPH</td>
<td>227</td>
</tr>
<tr>
<td>GROUP</td>
<td>229</td>
</tr>
<tr>
<td>HASH</td>
<td>231</td>
</tr>
<tr>
<td>HASH32</td>
<td>232</td>
</tr>
<tr>
<td>HASH64</td>
<td>233</td>
</tr>
<tr>
<td>HASHCRC</td>
<td>234</td>
</tr>
<tr>
<td>HASHMD5</td>
<td>235</td>
</tr>
<tr>
<td>HAVING</td>
<td>236</td>
</tr>
<tr>
<td>HTTPCALL</td>
<td>237</td>
</tr>
<tr>
<td>IF</td>
<td>239</td>
</tr>
<tr>
<td>IFF</td>
<td>240</td>
</tr>
<tr>
<td>IMPORT</td>
<td>241</td>
</tr>
<tr>
<td>INTFORMAT</td>
<td>242</td>
</tr>
<tr>
<td>ISVALID</td>
<td>243</td>
</tr>
<tr>
<td>ITERATE</td>
<td>244</td>
</tr>
<tr>
<td>JOIN</td>
<td>246</td>
</tr>
<tr>
<td>KEYDIFF</td>
<td>255</td>
</tr>
<tr>
<td>KEYPATCH</td>
<td>256</td>
</tr>
<tr>
<td>KEYUNICODE</td>
<td>258</td>
</tr>
<tr>
<td>LENGTH</td>
<td>259</td>
</tr>
<tr>
<td>LIBRARY</td>
<td>260</td>
</tr>
<tr>
<td>LIMIT</td>
<td>262</td>
</tr>
<tr>
<td>LN</td>
<td>264</td>
</tr>
<tr>
<td>LOADXML</td>
<td>265</td>
</tr>
<tr>
<td>LOCAL</td>
<td>267</td>
</tr>
<tr>
<td>LOG</td>
<td>268</td>
</tr>
<tr>
<td>LOOP</td>
<td>269</td>
</tr>
<tr>
<td>MAP</td>
<td>272</td>
</tr>
<tr>
<td>MAX</td>
<td>273</td>
</tr>
<tr>
<td>MERGE</td>
<td>275</td>
</tr>
<tr>
<td>MERGEJOIN</td>
<td>277</td>
</tr>
<tr>
<td>MIN</td>
<td>279</td>
</tr>
<tr>
<td>NOLOCAL</td>
<td>281</td>
</tr>
<tr>
<td>NONEMPTY</td>
<td>282</td>
</tr>
</tbody>
</table>
Workflow Services

Workflow Overview
CHECKPOINT
DEPRECATED
FAILURE
GLOBAL - Service
INDEPENDENT
ONCE
ONWARNING
PERSIST
PRIORITY
RECOVERY
STORED - Workflow Service
SUCCESS
WHEN

Template Language
Template Language Overview

#APPEND
#CONSTANT
#DECLARE
#DEMANGLE
#ERROR
#EXPAND
#EXPORT
#EXPORTXML
#FOR
#GETDATATYPE
#IF
#IFDEF
#ISDEFINED
#INMODULE
#LOOP / #BREAK
#MANGLE
#ONWARNING
#OPTION
#SET
#STORED
#TEXT
#UNIQUENAME
#WARNING
#WEBSERVICE
#WORKUNIT

External Services

SERVICE Structure
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST</td>
<td>447</td>
</tr>
<tr>
<td>External Service Implementation</td>
<td>448</td>
</tr>
<tr>
<td>Index</td>
<td>455</td>
</tr>
</tbody>
</table>
Introduction

Documentation Structure

This manual documents the Enterprise Control Language (ECL). ECL has been designed specifically for working with huge sets of data. This book is designed to be both a learning tool and a reference work and is divided into the following sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECL Basics</td>
<td>Addresses the fundamental concepts of ECL.</td>
</tr>
<tr>
<td>Expressions and Operators</td>
<td>Defines available operators and their expression evaluation precedence.</td>
</tr>
<tr>
<td>Value Types</td>
<td>Introduces data types and type casting.</td>
</tr>
<tr>
<td>Record Structures and Files</td>
<td>Introduces the RECORD structure, DATASET, and INDEX.</td>
</tr>
<tr>
<td>Alien Data Types</td>
<td>Defines the TYPE structure and the functions it may use.</td>
</tr>
<tr>
<td>Natural Language Parsing Support</td>
<td>Defines the patterns and functions the PARSE function may use.</td>
</tr>
<tr>
<td>Reserved Keywords</td>
<td>Defines special-use ECL keywords not elsewhere defined.</td>
</tr>
<tr>
<td>Special Structures</td>
<td>Defines the TRANSFORM, MACRO, and other structures and their use.</td>
</tr>
<tr>
<td>Built-In Functions and Actions</td>
<td>Defines the functions and actions available as part of the language.</td>
</tr>
<tr>
<td>Workflow Services</td>
<td>Defines the job execution/process control aspects of ECL.</td>
</tr>
<tr>
<td>Templates</td>
<td>Defines the ECL Template commands.</td>
</tr>
<tr>
<td>External Services</td>
<td>Defines the SERVICE structure and its use.</td>
</tr>
</tbody>
</table>
Documentation Conventions

ECL Syntax Case

Although ECL is not case-sensitive, ECL reserved keywords and built-in functions in this document are always shown in ALL CAPS to make them stand out for easy identification. Definition and record set names are always shown in example code as mixed-case. Run-on words may be used to explicitly identify purpose in examples.

Optional Items

Optional-use keywords and parameters are enclosed in square brackets in syntax diagrams with either/or options separated by a vertical bar (|), like this:

```
EXAMPLEFUNC( parameter [,optionalparameter] [,OPTIONAL | WORD])
```

Example Code

All example code in this document appears as in the following listing:

```
TotalTrades := COUNT(Trades); // TotalTrades is the Definition name
   // COUNT is a built-in function, Trades is the name of a record set
```
Overview

Enterprise Control Language (ECL) has been designed specifically for huge data projects using the Lexis-Nexis High Performance Computer Cluster (HPCC). ECL’s extreme scalability comes from a design that allows you to leverage every query you create for re-use in subsequent queries as needed. To do this, ECL takes a Dictionary approach to building queries wherein each ECL definition defines an expression. Each previous Definition can then be used in succeeding ECL definitions—the language extends itself as you use it.

Definitions versus Actions

Functionally, there are two types of ECL code: Definitions (AKA Attribute definitions) and executable Actions. Actions are not valid for use in expressions because they do not return values. Most ECL code is composed of definitions.

Definitions only define what is to be done, they do not actually execute. This means that the ECL programmer should think in terms of writing code that specifies what to do rather than how to do it. This is an important concept in that, the programmer is telling the supercomputer what needs to happen and not directing how it must be accomplished. This frees the super-computer to optimize the actual execution in any way it needs to produce the desired result.

A second consideration is: the order that Definitions appear in source code does not define their execution order—ECL is a non-procedural language. When an Action (such as OUTPUT) executes, all the Definitions it needs to use (drilling down to the lowest level Definitions upon which others are built) are compiled and optimized—in other words, unlike other programming languages, there is no inherent execution order implicit in the order that definitions appear in source code (although there is a necessary order for compilation to occur without error—forward references are not allowed). This concept of "orderless execution" requires a different mindset from standard, order-dependent programming languages because it makes the code appear to execute "all at once."

Syntax Issues

ECL is not case-sensitive. White space is ignored, allowing formatting for readability as needed.

Comments in ECL code are supported. Block comments must be delimited with /* and */.

/* this is a block comment - the terminator can be on the same line
or any succeeding line -- everything in between is ignored */

Single-line comments must begin with //.

// this is a one-line comment

ECL uses the standard object.property syntax used by many other programming languages (however, ECL is not an object-oriented language) to qualify Definition scope and disambiguate field references within tables:

ModuleName.Definition //reference an definition from another module/folder

Dataset.Field //reference a field in a dataset or recordset
Constants

String

All string literals must be contained within single quotation marks ( ' ' ). All ECL code is UTF-8 encoded, which means that all strings are also UTF-8 encoded, whether Unicode or non-Unicode strings. Therefore, you must use a UTF-8 editor (such as the ECL IDE program).

To include the single quote character (apostrophe) in a constant string, prepend a backslash (\). To include the backslash character (\) in a constant string, use two backslashes (\\) together.

```
STRING20 MyString2 := 'Fred''s Place';  // evaluated as: "Fred's Place"
STRING20 MyString3 := 'Fred\Ginger''s Place';  // evaluated as: "Fred\Ginger's Place"
```

Other available escape characters are:

<table>
<thead>
<tr>
<th>Escape</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\n</td>
<td>new line</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
<tr>
<td>\nnn</td>
<td>3 octal digits (for any other character)</td>
</tr>
<tr>
<td>\uhhhh</td>
<td>lowercase &quot;u&quot; followed by 4 hexadecimal digits (for any other UNICODE-only character)</td>
</tr>
</tbody>
</table>

```
MyString1 := 'abcd';
MyString2 := U'abcd\353';  // becomes 'abcdë'
```

**Hexadecimal string constants** must begin with a leading "x" character. Only valid hexadecimal values (0-9, A-F) may be in the character string and there must be an even number of characters.

```
DATA2 MyHexString := x'0D0A';  // a 2-byte hexadecimal string
```

**Data string constants** must begin with a leading "D" character. This is directly equivalent to casting the string constant to DATA.

```
MyDataString := D'abcd';  // same as: (DATA)'abcd'
```

**Unicode string constants** must begin with a leading "U" character. Characters between the quotes are utf16-encoded and the type of the constant is UNICODE.

```
MyUnicodeString1 := U'abcd';  // same as: (UNICODE)'abcd'
MyUnicodeString2 := U'abcd\353';  // becomes 'abcdë'
MyUnicodeString3 := U'abcd\u00EB';  // becomes 'abcdë»'
```

**UTF8 string constants** must begin with leading "U8" characters. Characters between the quotes are utf8-encoded and the type of the constant is UTF8.

```
MyUTF8String := U8'abcd\353';
```

**VARSTRING string constants** must begin with a leading "V" character. The terminating null byte is implied and type of the constant is VARSTRING.

```
MyVarString := V'abcd';  // same as: (VARSTRING)'abcd'
```

**QSTRING string constants** must begin with a leading "Q" character. The terminating null byte is implied and type of the constant is VARSTRING.
Multiline Strings

A multiline string begins and ends with three single quotes (""").

Any quotes, tabs, or newlines in between the triple quotes are part of the string. While you can use the \ escape character inside a multiline string, escaping is not necessary (except for the \ character). The \ character at the end of a line in a multiline string removes the end of line and joins the two lines together.

Examples:

'Single
 quotes';
u'Can't be multiline and must escape embedded single quotes';
u8'€';
v'C an use various prefixes';
d'7172737475';
Q'ABCDE';

'''Triple quotes can have embedded newlines, but also \ support \
 escape sequence''';
'''Single quotes inside a multiline string don't need escaping''';
u'''Unicode triple quotes
should be the same, and also \ support \
 escape sequence''';
u'''Don't have to be multiline and need not escape embedded quotes (but \
'can' if they want)''';
u8'''€''';
v'''Can use same prefixes as single''';
d'''7172737475''';
Q'''ABCDE''';

Numeric

Numeric constants containing a decimal portion are treated as REAL values (scientific notation is allowed) and those without are treated as INTEGER (see Value Types). Integer constants may be decimal, unsigned, hexadecimal, or binary values. Hexadecimal values are specified with either a leading "0x" or a trailing "x" character. Binary values are specified with either a leading "0b" or a trailing "b" character. Decimal values are specified with trailing "d" character. Unsigned values are specified with a trailing "u" character.

MyInt1 := 10; // value of MyInt1 is the INTEGER value 10
MyInt2 := 0x0A; // value of MyInt2 is the INTEGER value 10
MyInt3 := 0Ax; // value of MyInt3 is the INTEGER value 10
MyInt4 := 0b1010; // value of MyInt4 is the INTEGER value 10
MyInt5 := 1010b; // value of MyInt5 is the INTEGER value 10
MyUint := 10u // value of MyUint is the UNSIGNED value 10
MyReal1 := 10.0; // value of MyReal1 is the REAL value 10.0
MyReal2 := 1.0e1; // value of MyReal2 is the REAL value 10.0
MyDec1 := 10d; // value of MyDec1 is the DECIMAL value 10
MyDec2 := 3.14159265358979323846d // value of MyDec2 is the DECIMAL
// value 3.14159265358979323846
// a REAL type would lose precision
Compile Time Constants

The following system constants are available at compile time. These can be useful in creating conditional code.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ECL_VERSION</strong></td>
<td>A STRING containing the value of the platform version. For example, '6.4.0'</td>
</tr>
<tr>
<td><strong>ECL_VERSION_MAJOR</strong></td>
<td>An INTEGER containing the value of the major portion of the platform version. For example, '6'</td>
</tr>
<tr>
<td><strong>ECL_VERSION_MINOR</strong></td>
<td>An INTEGER containing the value of the minor portion of the platform version. For example, '4'</td>
</tr>
<tr>
<td><strong>ECL_LEGACY_MODE</strong></td>
<td>A BOOLEAN value indicating if it is being compiled with legacy IMPORT semantics.</td>
</tr>
<tr>
<td><strong>OS</strong></td>
<td>A STRING indicating the operating system to which it is being compiled. Possible values are: 'windows', 'macos', or 'linux'.</td>
</tr>
<tr>
<td><strong>STAND_ALONE</strong></td>
<td>A BOOLEAN value indicating if it is being compiled to a stand-alone executable.</td>
</tr>
<tr>
<td><strong>TARGET_PLATFORM</strong></td>
<td>A STRING containing the value of the target platform (the type of cluster the query was submitted to). Possible values are: 'roxie', 'hthor', or 'thorlcr'.</td>
</tr>
<tr>
<td><strong>CONTAINERIZED</strong></td>
<td>A BOOLEAN value indicating if the platform is a containerized version.</td>
</tr>
</tbody>
</table>

Examples:

```ecl
// Any modules referenced inside the condition must be declared outside of the condition
// This will avoid syntax errors
prod_thor_dali := _Control.IPAddress.prod_thor_dali;
#IF(_TARGET_PLATFORM_ in ['thorlcr'])
   OUTPUT('thor');
   prod_thor_dali;
#ELSE
   OUTPUT('not thor');
#END

// Second example
IMPORT STD;
STRING14 fGetDateTimeString() :=
#IF(__ECL_VERSION_MAJOR__ > 5) or ((__ECL_VERSION_MAJOR__ = 5) AND (__ECL_VERSION_MINOR__ >= 2))
   STD.Date.SecondsToString(STD.Date.CurrentSeconds(true), '%Y%m%d%H%M%S');
#ELSE
   FUNCTION string14 fGetDimeTime():= // 14 characters returned
      BEGINC++
         #option action
         struct tm localtime; // localtime in "tm" structure
         time_t timeinsecs; // variable to store time in secs
         time(&timeinsecs);
         localtime_r(&timeinsecs,&localtime);
         char temp[15];
         strftime(temp, 15, "%Y%m%d%H%M%S", &localtime); // Formats the localtime to YYYYMMDDhhmms
         strncpy(__result, temp, 14);
         ENDC++;
         RETURN fGetDimeTime();
      END;
#END;
```
// Example using __CONTAINERIZED__
over := 'overwrite=1 ';
repl := 'replicate=1 ';
action := 'action=spray ';
srcplane := 'srcplane=mydropzone ';
srcfile := 'srcfile=originalperson ';
dstname := 'dstname=JD::originalperson ';
//dstcluster:= 'dstcluster=data '; // for containerized
//dstcluster:= 'dstcluster=mythor '; // for bare-metal
dstcluster := IF(__CONTAINERIZED__, 'dstcluster=data ','dstcluster=mythor ');
fmt := 'format=fixed ';
recsize := 'recordsize=124 ';
cmd := over + repl + action + srcplane + srcfile + dstname
  + dstcluster + fmt + recsize;
STD.File.DfuPlusExec(cmd);

Runtime Expressions
The following system constants are evaluated at runtime. Technically, these are runtime expressions, not constants. Therefore, they cannot be used in conditional code that requires a constant.

<table>
<thead>
<tr>
<th><strong>PLATFORM</strong></th>
<th>A STRING that represents the type of engine where the query is executing on. Possible values are: 'roxie', 'hthor', or 'thorlcr'.</th>
</tr>
</thead>
</table>
Environment Variables

Environment variables store data about the current platform environment. You can retrieve the value of an environment variable using the built-in function GETENV().

**HPCC_DEPLOYMENT**

In a Kubernetes deployment, this variable is auto-populated with the name of the Helm chart. For example, if deployed using:

```sh
helm install myenv1 hpcc/hpcc
```

then the HPCC_DEPLOYMENT variable will contain the value: myenv1.

For a bare-metal environment, this value can be set in environment.conf, by adding:

```conf
deploymentName=myenv1
```

You can retrieve this value using:

```ecl
OUTPUT(GETENV('HPCC_DEPLOYMENT', 'unknown'))
```

See Also: GETENV
Definitions

Each ECL definition is the basic building block of ECL. A definition specifies what is done but not how it is to be done. Definitions can be thought of as a highly developed form of macro-substitution, making each succeeding definition more and more highly leveraged upon the work that has gone before. This results in extremely efficient query construction.

All definitions take the form:

```
[Scope] [ValueType] Name [ (parms) ] := Expression [ :WorkflowService] ;
```

The Definition Operator ( := read as "is defined as") defines an expression. On the left side of the operator is an optional Scope (see Attribute Visibility), ValueType (see Value Types), and any parameters (parms) it may take (see Functions (Parameter Passing)). On the right side is the expression that produces the result and optionally a colon (:) and a comma-delimited list of WorkflowServices (see Workflow Services). A definition must be explicitly terminated with a semi-colon (;). The Definition name can be used in subsequent definitions:

```
MyFirstDefinition := 5; //defined as 5
MySecondDefinition := MyFirstDefinition + 5; //this is 10
```

Definition Name Rules

Definition names begin with a letter and may contain only letters, numbers, or underscores (_).

```
My_First_Definition1 := 5; // valid name
My First Definition := 5;  // INVALID name, spaces not allowed
```

You may name a Definition with the name of a previously created module in the ECL Repository, if the attribute is defined with an explicit ValueType.

Reserved Words

ECL keywords, built-in functions and their options are reserved words, but they are generally reserved only in the context within which they are valid for use. Even in that context, you may use reserved words as field or definition names, provided you explicitly disambiguate them, as in this example:

```
ds2 := DEDUP(ds, ds.all, ALL); //ds.all is the 'all' field in the
//ds dataset - not DEDUP's ALL option
```

However, it is still a good idea to avoid using ECL keywords as definition or field names.

Definition or field names cannot begin with UNICODE_, UTF8_, or VARUNICODE_. Labels beginning with those prefixes are treated as type names, and should be regarded as reserved.

Definition Naming

Use descriptive names for all EXPORTed and SHARED Definitions. This will make your code more readable. The naming convention adopted throughout the ECL documentation and training courses is as follows:

<table>
<thead>
<tr>
<th>Definition Type</th>
<th>Are Named</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>Is...</td>
</tr>
<tr>
<td>Set Definition</td>
<td>Set...</td>
</tr>
<tr>
<td>Record Set</td>
<td>...DatasetName</td>
</tr>
</tbody>
</table>

For example:
IsTrue := TRUE;                         // a BOOLEAN Definition
SetNumbers := [1,2,3,4,5];              // a Set Definition
R_People := People(firstname[1] = 'R'); // a Record Set Definition
Basic Definition Types

The basic types of Definitions used most commonly throughout ECL coding are: **Boolean**, **Value**, **Set**, **Record Set**, and **TypeDef**.

**Boolean Definitions**

A Boolean Definition is defined as any Definition whose definition is a logical expression resulting in a TRUE/FALSE result. For example, the following are all Boolean Definitions:

```
IsBoolTrue := TRUE;
IsFloridian := Person.per_st = 'FL';
IsOldPerson := Person.Age >= 65;
```

**Value Definitions**

A Value Definition is defined as any Definition whose expression is an arithmetic or string expression with a single-valued result. For example, the following are all Value Definitions:

```
ValueTrue := 1;
FloridianCount := COUNT(Person(Person.per_st = 'FL'));
OldAgeSum := SUM(Person(Person.Age >= 65), Person.Age);
```

**Set Definitions**

A Set Definition is defined as any Definition whose expression is a set of values, defined within square brackets. Constant sets are created as a set of explicitly declared constant values that must be declared within square brackets, whether that set is defined as a separate definition or simply included in-line in another expression. All the constants must be of the same type.

```
SetInts := [1,2,3,4,5]; // an INTEGER set with 5 elements
SetReals := [1.5,2.0,3.3,4.2,5.0];
    // a REAL set with 5 elements
SetStatusCodes := ['A','B','C','D','E'];
    // a STRING set with 5 elements
```

The elements in any explicitly declared set can also be composed of arbitrary expressions. All the expressions must result in the same type and must be constant expressions.

```
SetExp := [1,2+3,45,SomeIntegerDefinition,7*3];
    // an INTEGER set with 5 elements
```

Declared Sets can contain definitions and expressions as well as constants as long as all the elements are of the same result type. For example:

```
StateCapital(String2 state) :=
    CASE(state, 'FL' => 'Tallahassee', 'Unknown');
SetFloridaCities := ['Orlando', StateCapital('FL'), 'Boca '+'Raton',
    person[1].per_full_city];
```

Set Definitions can also be defined using the SET function (which see). Sets defined this way may be used like any other set.

```
SetSomeField := SET(SomeFile, SomeField);
    // a set of SomeField values
```

Sets can also contain datasets for use with those functions (such as: MERGE, JOIN, MERGEJOIN, or GRAPH) that require sets of datasets as input parameters.
SetDS := [ds1, ds2, ds3]; // a set of datasets

You can construct a DATASET from a SET.

SET OF STRING s := ['Jim','Bob','Richard','Tom'];
DATASET(s,{STRING txt});

**Set Ordering and Indexing**

Sets are implicitly ordered and you may index into them to access individual elements. Square brackets are used to specify the element number to access. The first element is number one (1).

MySet := [5,4,3,2,1];
ReverseNum := MySet[2]; //indexing to MySet's element number 2,
//so ReverseNum contains the value 4

Strings (Character Sets) may also be indexed to access individual or multiple contiguous elements within the set of characters (a string is treated as though it were a set of 1-character strings). An element number within square brackets specifies an individual character to extract.

MyString := 'ABCDE';
MySubString := MyString[2]; // MySubString is 'B'

Substrings may be extracted by using two periods to separate the beginning and ending element numbers within the square brackets to specify the substring (string slice) to extract. Either the beginning or ending element number may be omitted to indicate a substring from the beginning to the specified element, or from the specified element through to the end.

MyString := 'ABCDE';
MySubString1 := MyString[2..4]; // MySubString1 is 'BCD'
MySubString2 := MyString[..4]; // MySubString2 is 'ABCD'
MySubString3 := MyString[2.. ]; // MySubString3 is 'BCDE'

**Record Set Definitions**

The term "Dataset" in ECL explicitly means a "physical" data file in the supercomputer (on disk or in memory), while the term "Record Set" indicates any set of records derived from a Dataset (or another Record Set), usually based on some filter condition to limit the result set to a subset of records. Record sets are also created as the return result from one of the built-in functions that return result sets.

A Record Set Definition is defined as any Definition whose expression is a filtered dataset or record set, or any function that returns a record set. For example, the following are all Record Set Definitions:

FloridaPersons := Person(Person.per_st = 'FL');
OldFloridaPersons := FloridaPersons(Person.Age >= 65);

**Record Set Ordering and Indexing**

All Datasets and Record Sets are implicitly ordered and may be indexed to access individual records within the set. Square brackets are used to specify the element number to access, and the first element in any set is number one (1).

Datasets (including child datasets) and Record Sets may use the same method as described above for strings to access individual or multiple contiguous records.

MyRec1 := Person[1]; // first rec in dataset
MyRec2 := Person[1..10]; // first ten recs in dataset
MyRec4 := Person[2..]; // all recs except the first
Note: ds[1] and ds[1..1] are not the same thing--ds[1..1] is a recordset (may be used in recordset context) while ds[1] is a single row (may be used to reference single fields).

And you can also access individual fields in a specified record with a single index:

MyField := Person[1].per_last_name; // last name in first rec

Indexing a record set with a value that is out of bounds is defined to return a row where all the fields contain blank/zero values. It is often more efficient to index an out of bound value rather than writing code that handles the special case of an out of bounds index value.

For example, the expression:

IF(COUNT(ds) > 0, ds[1].x, 0);

is simpler as:

ds[1].x    //note that this returns 0 if ds contains no records.

TypeDef Definitions

A TypeDef Definition is defined as any Definition whose definition is a value type, whether built-in or user-defined. For example, the following are all TypeDef Definitions (exceptGetXLen):

GetXLen(DATA x, UNSIGNED len) := TRANSFER(((DATA4)(x[1..len])),UNSIGNED4);

EXPORT xstring(UNSIGNED len) := TYPE
    EXPORT INTEGER PHYSICALLENGTH(DATA x) := GetXLen(x, len) + len;
    EXPORT STRING LOAD(DATA x) := (STRING)x[(len+1)..GetXLen(x,len) + len];
    EXPORT DATA STORE(STRING x):= TRANSFER(LENGTH(x),DATA4)[1..len] + (DATA)x;
END;

pstr := xstring(1); // typedef for user defined type
pppstr := xstring(3); // typedef of a system type
nameStr := STRING20; // typedef of a system type

namesRecord := RECORD
    pstr surname;
    nameStr forename;
    pppStr addr;
END;
//A RECORD structure is also a typedef definition (user-defined)
Recordset Filtering

Filters are conditional expressions contained within the parentheses following the Dataset or Record Set name. Multiple filter conditions may be specified by separating each filter expression with a comma (,). All filter conditions separated by commas must be TRUE for a record to be included, which makes the comma an implicit AND operator (see Logical Operators) in this context only.

```
MyRecordSet := Person(per_last_name >= 'T', per_last_name < 'U');  // MyRecordSet contains people whose last name begins with "T"  // the comma is an implicit AND while also functioning as  // an expression separator (implicit parentheses)
```

```
MyRecordSet := Person(per_last_name >= 'T' AND per_last_name < 'U');  // exactly the same logical expression as above
RateGE7trds := Trades(trd_rate >= '7');
ValidTrades := Trades(NOT rmsTrade.Mortgage AND NOT rmsTrade.HasNarrative(rmsTrade.snClosed));
```

Boolean definitions should be used as recordset filters for maximum flexibility, readability and re-usability instead of hard-coding in a Record Set definition. For example, use:

```
IsRevolv := trades.trd_type = 'R'  
   | (~ValidType(trades.trd_type)  
   |   AND trades.trd_acct[1] IN ['4','5','6']);

isBank := trades.trd_ind_code IN SetBankIndCodes;
IsBankCard := IsBank AND IsRevolv;
WithinDate(INTEGER1 months) := ValidDate(trades.trd_drpt) AND 
   trades.trd_drpt_mos <= months;
BankCardTrades := trades(isBankCard AND WithinDate(6));
```

instead of:

```
BankCardTrades := trades(trades.trd_ind_code IN SetBankIndCodes,  
   (trades.trd_type = 'R' OR  
   (~ValidType(trades.trd_type) AND  
   trades.trd_acct[1] IN ['4', '5', '6'])),  
   ValidDate(trades.trd_drpt),  
   trades.trd_drpt_mos <= 6);
```

Commas used to separate filter conditions in a recordset filter definition act as both an implicit AND operation and a set of parentheses around the individual filters being separated. This results in a tighter binding than if AND is used instead of a comma without parentheses. For example, the filter expression in this definition::

```
BankMortTrades := trades(isBankCard OR isMortgage, isOpen);
```

is evaluated as if it were written:

```
(isBankCard OR isMortgage) AND isOpen
```

and not as:

```
isBankCard OR isMortgage AND isOpen
```
Function Definitions (Parameter Passing)

All of the basic Definition types can also become functions by defining them to accept passed parameters (arguments). The fact that it receives parameters doesn't change the essential nature of the Definition's type, it simply makes it more flexible.

Parameter definitions always appear in parentheses attached to the Definition's name. You may define the function to receive as many parameters as needed to create the desired functionality by simply separating each succeeding parameter definition with a comma.

The format of parameter definitions is as follows:

DefinitionName( [ ValueType ] AliasName [ = DefaultValue ] ) := expression;

<table>
<thead>
<tr>
<th>ValueType</th>
<th>Optional. Specifies the type of data being passed. If omitted, the default is INTEGER (see Value Types). This also may include the CONST keyword (see CONST) to indicate that the passed value will always be treated as a constant.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AliasName</td>
<td>Names the parameter for use in the expression.</td>
</tr>
<tr>
<td>DefaultValue</td>
<td>Optional. Provides the value to use in the expression if the parameter is omitted. The DefaultValue may be the keyword ALL if the ValueType is SET (see the SET keyword) to indicate all possible values for that type of set, or empty square brackets ([ ]) to indicate no possible value for that type of set.</td>
</tr>
<tr>
<td>expression</td>
<td>The function's operation for which the parameters are used.</td>
</tr>
</tbody>
</table>

Simple Value Type Parameters

If the optional ValueType is any of the simple types (BOOLEAN, INTEGER, REAL, DECIMAL, STRING, QSTRING, UNICODE, DATA, VARSTRING, VARUNICODE), the ValueType may include the CONST keyword (see CONST) to indicate that the passed value will always be treated as a constant (typically used only in ECL prototypes of external functions).

```ecl
ValueDefinition := 15;
FirstFunction(INTEGER x=5) := x + 5;
// takes an integer parameter named "x" and "x" is used in the
// arithmetic expression to indicate the usage of the parameter
SecondDefinition := FirstFunction(ValueDefinition);
// The value of SecondDefinition is 20
ThirdDefinition := FirstFunction();
// The value of ThirdDefinition is 10, omitting the parameter
```

SET Parameters

The DefaultValue for SET parameters may be a default set of values, the keyword ALL to indicate all possible values for that type of set, or empty square brackets ([ ]) to indicate no possible value for that type of set (and empty set).

```ecl
SET OF INTEGER1 SetValues := [5,10,15,20];
IsInSetFunction(SET OF INTEGER1 x=SetValues,y) := y IN x;
```
OUTPUT(IsInSetFunction([1,2,3,4],5)); //false
OUTPUT(IsInSetFunction((),5)); // true

### Passing DATASET Parameters

Passing a DATASET or a derived recordset as a parameter may be accomplished using the following syntax:

\[
\text{DefinitionName( DATASET( recstruct ) AliasName ) := expression;}
\]

The required `recstruct` names the RECORD structure that defines the layout of fields in the passed DATASET parameter. The `recstruct` may alternatively use the RECORDOF function. The required `AliasName` names the dataset for use in the function and is used in the Definition's `expression` to indicate where in the operation the passed parameter is to be used. See the DATASET as a Value Type discussion in the DATASET documentation for further examples.

```ecl
MyRec := {STRING1 Letter};
SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'}],MyRec);
FilteredDS(DATASET(MyRec) ds) := ds(Letter NOT IN ['A','C','E']);
//passed dataset referenced as "ds" in expression
OUTPUT(FilteredDS(SomeFile));
```

### Passing DICTIONARY Parameters

Passing a DICTIONARY as a parameter may be accomplished using the following syntax:

\[
\text{DefinitionName( DICTIONARY( structure ) AliasName ) := expression;}
\]

The required `structure` parameter is the RECORD structure that defines the layout of fields in the passed DICTIONARY parameter (usually defined inline). The required `AliasName` names the DICTIONARY for use in the function and is used in the Definition's `expression` to indicate where in the operation the passed parameter is to be used. See the DICTIONARY as a Value Type discussion in the DICTIONARY documentation.

```ecl
rec := RECORD
    STRING10 color;
    UNSIGNED1 code;
    STRING10 name;
END;
Ds := DATASET([{'Black',0,'Fred'},
               {'Brown',1,'Seth'},
               {'Red',2,'Sue'},
               {'White',3,'Jo'}], rec);
DsDCT := DICTIONARY(DS,{color => DS});
DCTrec := RECORD
    STRING10 color =>
    UNSIGNED1 code,
    STRING10 name,
END;
InlineDCT := DICTIONARY([{'Black' => 0,'Fred'},
                          {'Brown' => 1,'Sam'},
                          {'Red' => 2,'Sue'},
                          {'White' => 3,'Jo'}], DCTrec);
MyDCTfunc(DICTIONARY(DCTrec) DCT,STRING10 key) := DCT[key].name;
MyDCTfunc(InlineDCT,'White');  //Jo
```
Passing Typeless Parameters

Passing parameters of any type may be accomplished using the keyword ANY as the passed value type:

\[
\text{DefinitionName ( \text{ANY} \ \text{AliasName} ) := expression;}
\]

```
a := 10;
b := 20;
c := '1';
d := '2';
e := '3';
f := '4';
s1 := [c,d];
s2 := [e,f];
ds1 := DATASET(s1,{STRING1 ltr});
ds2 := DATASET(s2,{STRING1 ltr});

MyFunc(ANY l, ANY r) := l + r;
MyFunc(a,b);      //returns 30
MyFunc(a,c);      //returns '101'
MyFunc(c,d);      //returns '12'
MyFunc(s1,s2);    //returns a set: ['1','2','3','4']
MyFunc(ds1,ds2);  //returns 4 records: '1', '2', '3', and '4'
```

Passing Function Parameters

Passing a Function as a parameter may be accomplished using either of the following syntax options as the \textit{ValueType} for the parameter:

\[
\text{FunctionName(parameters)}
\]

\[
\text{PrototypeName}
\]

<table>
<thead>
<tr>
<th>FunctionName</th>
<th>The name of a function, the type of which may be passed as a parameter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>The parameter definitions for the FunctionName parameter.</td>
</tr>
<tr>
<td>PrototypeName</td>
<td>The name of a previously defined function to use as the type of function that may be passed as a parameter.</td>
</tr>
</tbody>
</table>

The following code provides examples of both methods:

```
//a Function prototype:
INTEGER actionPrototype(INTEGER v1, INTEGER v2) := 0;

INTEGER aveValues(INTEGER v1, INTEGER v2) := (v1 + v2) DIV 2;
INTEGER addValues(INTEGER v1, INTEGER v2) := v1 + v2;
INTEGER multiValues(INTEGER v1, INTEGER v2) := v1 * v2;

//a Function prototype using a function prototype:
INTEGER applyPrototype(INTEGER v1, actionPrototype actionFunc) := 0;

//using the Function prototype and a default value:
INTEGER applyValue2(INTEGER v1, actionPrototype actionFunc = aveValues) :=
    actionFunc(v1, v1+1)*2;

//Defining the Function parameter inline, with a default value:
```
**ECL Basics**

```ecl
INTEGER applyValue4(INTEGER v1,  
    INTEGER actionFunc(INTEGER v1,INTEGER v2) = aveValues)  
:= actionFunc(v1, v1+1)*4;

INTEGER doApplyValue(INTEGER v1,  
    INTEGER actionFunc(INTEGER v1, INTEGER v2))  
:= applyValue2(v1+1, actionFunc);

//producing simple results:
OUTPUT(applyValue2(1));                           // 2
OUTPUT(applyValue2(2));                           // 4
OUTPUT(applyValue2(1, addValues));                // 6
OUTPUT(applyValue2(2, addValues));                // 10
OUTPUT(applyValue2(1, multiValues));              // 4
OUTPUT(applyValue2(2, multiValues));              // 12
OUTPUT(doApplyValue(1, multiValues));             // 12
OUTPUT(doApplyValue(2, multiValues));             // 24

//A definition taking function parameters which themselves  
//have parameters that are functions...

STRING doMany(INTEGER v1,  
    INTEGER firstAction(INTEGER v1,  
        INTEGER actionFunc(INTEGER v1,INTEGER v2)),  
    INTEGER secondAction(INTEGER v1,  
        INTEGER actionFunc(INTEGER v1,INTEGER v2)),  
    INTEGER actionFunc(INTEGER v1,INTEGER v2))  
:= (STRING)firstAction(v1, actionFunc) + ':' + (STRING)secondAction(v1, actionFunc);

OUTPUT(doMany(1, applyValue2, applyValue4, addValues));  
// produces "6:12"

OUTPUT(doMany(2, applyValue4, applyValue2,multiValues));  
// produces "24:12"
```

---

**Passing NAMED Parameters**

Passing values to a function defined to receive multiple parameters, many of which have default values (and are therefore omittable), is usually accomplished by "counting commas" to ensure that the values you choose to pass are passed to the correct parameter by the parameter's position in the list. This method becomes untenable when there are many optional parameters.

The easier method is to use the following NAMED parameter syntax, which eliminates the need to include extraneous commas as place holders to put the passed values in the proper parameters:

```
Attr := FunctionName([ NAMED ] AliasName := value);
```

<table>
<thead>
<tr>
<th><strong>NAMED</strong></th>
<th>Optional. Required only when the <em>AliasName</em> clashes with a reserved word.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AliasName</strong></td>
<td>The names of the parameter in the definition's function definition. This must be a valid label (See Definition Name Rules)</td>
</tr>
<tr>
<td><strong>value</strong></td>
<td>The value to pass to the parameter.</td>
</tr>
</tbody>
</table>

This syntax is used in the call to the function and allows you to pass values to specific parameters by their *AliasName*, without regard for their position in the list. All unnamed parameters passed must precede any NAMED parameters.

```
outputRow(BOOLEAN showA = FALSE, BOOLEAN showB = FALSE,  
    BOOLEAN showC = FALSE, STRING aValue = 'abc',  
    INTEGER bValue = 10, BOOLEAN cValue = TRUE) :=
```
OUTPUT(IF(showA,' a='+aValue,'')+
     IF(showB,' b=('+(STRING)bValue,')'+
     IF(showc,' c=('+(STRING)cValue,')));

outputRow(); //produce blanks
outputRow(TRUE); //produce "a=abc"
outputRow(,,TRUE); //produce "c=TRUE"
outputRow(NAMED showB := TRUE); //produce "b=10"

outputRow(TRUE, NAMED aValue := 'Changed value');
    //produce "a=Changed value"

outputRow(,,, 'Changed value2',NAMED showA := TRUE);
     //produce "a=Changed value2"

outputRow(showB := TRUE); //produce "b=10"

outputRow(TRUE, aValue := 'Changed value');
outputRow(,,, 'Changed value2',showA := TRUE);
Definition Visibility

ECL code, definitions, are stored in .ECL files in your code repository, which are organized into modules (directories or folders on disk). Each .ECL file may only contain a single EXPORT or SHARED definition (see below) along with any supporting local definitions required to fully define the definition's result. The name of the file and the name of its EXPORT or SHARED definition must exactly match.

Within a module (directory or folder on disk), you may have as many EXPORT and/or SHARED definitions as needed. An IMPORT statement (see the IMPORT keyword) identifies any other modules whose visible definitions will be available for use in the current definition.

The following fundamental definition visibility scopes are available in ECL: "Global," Module, and Local.

"Global"

Definitions defined as EXPORT (see the EXPORT keyword) are available throughout the module in which they are defined, and throughout any other module that IMPORTs that module (see the IMPORT keyword).

---

//inside the Definition1.ecl file (in AnotherModule folder) you have:
EXPORT Definition1 := 5;
   //EXPORT makes Definition1 available to other modules and
   //also available throughout its own module

Module

The scope of the definitions defined as SHARED (see the SHARED keyword) is limited to that one module, and are available throughout the module (unlike local definitions). This allows you to keep private any definitions that are only needed to implement internal functionality. SHARED definitions are used to support EXPORT definitions.

---

//inside the Definition2.ecl file you have:
IMPORT AnotherModule;
   //makes definitions from AnotherModule available to this code, as needed

SHARED Definition2 := AnotherModule.Definition1 + 5;
   //Definition2 available throughout its own module, only

//then inside the Definition3.ecl file (in the same folder as Definition2) you have:
IMPORT $;
   //makes definitions from the current module available to this code, as needed

EXPORT Definition3 := $.Definition2 + 5;
   //make Definition3 available to other modules and
   //also available throughout its own module

Local

A definition without either the EXPORT or SHARED keywords is available only to subsequent definitions, until the end of the next EXPORT or SHARED definition. This makes them private definitions used only within the scope of that one EXPORT or SHARED definition, which allows you to keep private any definitions that are only needed to implement internal functionality. Local definitions definitions are used to support the EXPORT or SHARED definition in whose file they reside. Local definitions are referenced by their definition name alone; no qualification is needed.

---

//then inside the Definition4.ecl file (in the same folder as Definition2) you have:
IMPORT $;
   //makes definitions from the current module available to this code, as needed
LocalDef := 5;
   //local -- available through the end of Definition4's definition, only
EXPORT Definition4 := LocalDef + 5;
   //EXPORT terminates scope for LocalDef
LocalDef2 := Definition4 + LocalDef;
   //INVALID SYNTAX -- LocalDef is out of scope here
   //and any local definitions following the EXPORT
   //or SHARED definition in the file are meaningless
   //since they can never be used by anything
The **LOCAL** keyword is valid for use within any nested structure, but most useful within a FUNCTION-MACRO structure to clearly identify that the scope of a definition is limited to the code generated within the FUNCTIONMACRO.

AddOne(num) := FUNCTIONMACRO
   LOCAL numPlus := num + 1;
   RETURN numPlus;
ENDMACRO;
numPlus := 'this is a syntax error without LOCAL in the FUNCTIONMACRO';
numPlus;
AddOne(5);

See Also: IMPORT, EXPORT, SHARED, MODULE, FUNCTIONMACRO
Field and Definition Qualification

Imported Definitions

EXPORTed definitions defined within another module and IMPORTed (see the EXPORT and IMPORT keywords) are available for use in the definition that contains the IMPORT. Imported Definitions must be fully qualified by their Module name and Definition name, using dot syntax (module.definition).

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORT abc;</td>
<td>//make all exported definitions in the abc module available</td>
</tr>
<tr>
<td>EXPORT Definition1 := 5;</td>
<td>//make Definition1 available to other modules</td>
</tr>
<tr>
<td>Definition2 := abc.Definition2 + Definition1;</td>
<td>// object qualification needed for Definitions from abc module</td>
</tr>
</tbody>
</table>

Fields in Datasets

Each Dataset counts as a qualified scope and the fields within them are fully qualified by their Dataset (or record set) name and Field name, using dot syntax (dataset.field). Similarly, the result set of the TABLE built-in function (see the **TABLE** keyword) also acts as a qualified scope. The name of the record set to which a field belongs is the object name:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Young := YearOf(Person.per_dbrth) &lt; 1950;</td>
<td>// make the field 'YearOf(Person.per_dbrth)' part of the current scope</td>
</tr>
<tr>
<td>MySet := Person(Young);</td>
<td>// make the object 'Person' part of the current scope</td>
</tr>
</tbody>
</table>

When naming a Dataset as part of a definition, the fields of that Definition (or record set) come into scope. If Parameterized Definitions (functions) are nested, only the innermost scope is available. That is, all the fields of a Dataset (or derived record set) are in scope in the filter expression. This is also true for expressions parameters of any built-in function that names a Dataset or derived record set as a parameter.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySet1 := Person(YearOf(dbrth) &lt; 1950);</td>
<td>// MySet1 is the set of Person records who were born before 1950</td>
</tr>
<tr>
<td>MySet2 := Person(EXISTS(OpenTrades(AgeOf(trd_dla) &lt; AgeOf(Person.per_dbrth))));</td>
<td>// OpenTrades is a pre-defined record set.</td>
</tr>
<tr>
<td>//All Trades fields are in scope in the OpenTrades record set filter</td>
<td>//expression, but Person is required here to bring Person.per_dbrth into scope</td>
</tr>
<tr>
<td>//This example compares each trades' Date of Last Activity to the</td>
<td>//related person's Date Of Birth</td>
</tr>
<tr>
<td>related person's Date Of Birth</td>
<td></td>
</tr>
</tbody>
</table>

Any field in a Record Set can be qualified with either the Dataset name the Record Set is based on, or any other Record Set name based on the same base dataset. For example:

```
memtrade.trd_drpt
nondup_trades.trd_drpt
trades.trd_drpt
```

all refer to the same field in the memtrade dataset.

For consistency, you should typically use the base dataset name for qualification. You can also use the current Record Set's name in any context where the base dataset name would be confusing.

Scope Resolution Operator

Identifiers are looked up in the following order:

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
1. The currently active dataset, if any

2. The current definition being defined, and any parameters it is based on

3. Any definitions or parameters of any MODULE or FUNCTION structure that contains the current definition

This might mean that the definition or parameter you want to access isn't picked because it is hidden as in a parameter or private definition name clashing with the name of a dataset field.

It would be better to rename the parameter or private definition so the name clash cannot occur, but sometimes this is not possible.

You may direct access to a different match by qualifying the field name with the scope resolution operator (the carat (^) character), using it once for each step in the order listed above that you need to skip.

This example shows the qualification order necessary to reach a specific definition/parameter:

ds := DATASET([1], { INTEGER SomeValue });

INTEGER SomeValue := 10; //local definition

myModule(INTEGER SomeValue) := MODULE

   EXPORT anotherFunction(INTEGER SomeValue) := FUNCTION
   
   tbl := TABLE(ds,(SUM(GROUP, someValue), // 1 - DATASET field
                     SUM(GROUP, ^.someValue), // 84 - FUNCTION parameter
                     SUM(GROUP, ^^.someValue), // 42 - MODULE parameter
                     SUM(GROUP, ^^^.someValue), // 10 - local definition
                     0));

   RETURN tbl;
END;

EXPORT result := anotherFunction(84);
END;

OUTPUT(myModule(42).result);

In this example there are four instances of the name “SomeValue”:

a field in a DATASET.

a local definition

a parameter to a MODULE structure

a parameter to a FUNCTION structure

The code in the TABLE function shows how to reference each separate instance.

While this syntax allows exceptions where you need it, creating another definition with a different name is the preferred solution.
Actions and Definitions

While Definitions define expressions that may be evaluated, Actions trigger execution of a workunit that produces results that may be viewed. An Action may evaluate Definitions to produce its result. There are a number of built-in Actions in ECL (such as OUTPUT), and any expression (without a Definition name) is implicitly treated as an Action to produce the result of the expression.

Expressions as Actions

Fundamentally, any expression in ECL can be treated as an Action. For example,

```plaintext
Attr1 := COUNT(Trades);
Attr2 := MAX(Trades,trd_bal);
Attr3 := IF (1 = 0, 'A', 'B');
```

are all definitions, but without a definition name, they are simply expressions

```plaintext
COUNT(Trades);       //execute these expressions as Actions
MAX(Trades,trd_bal);
IF (1 = 0, 'A', 'B');
```

that are treated as actions, and as such, can directly generate result values by simply submitting them as queries to the supercomputer. Basically, any ECL expression can be used as an Action to instigate a workunit.

Definitions as Actions

These same expression definitions can be executed by submitting the names of the Definitions as queries, like this:

```plaintext
Attr1; //These all generate the same result values
Attr2; // as the previous examples
Attr3;
```

Actions as Definitions

Conversely, by simply giving any Action a Definition name it becomes a definition, therefore no longer a directly executable action. For example,

```plaintext
OUTPUT(Person);
```

is an action, but

```plaintext
Attr4 := OUTPUT(Person);
```

is a definition and does not immediately execute when submitted as part of a query. To execute the action inherent in the definition, you must execute the Definition name you've given to the Action, like this:

```plaintext
Attr4;    // run the previously defined OUTPUT(Person) action
```

Debugging Uses

This technique of directly executing a Definition as an Action is useful when debugging complex ECL code. You can send the Definition as a query to determine if intermediate values are correctly calculated before continuing on with more complex code.
Expressions and Operators

Expressions are evaluated left-to-right and from the inside out (in nested functions). Parentheses may be used to alter the default evaluation order of precedence for all operators.

Arithmetic Operators

Standard arithmetic operators are supported for use in expressions, listed here in their evaluation precedence.

Note: * , /, %, and DIV all have the same precedence and are left associative. + and - have the same precedence and are left associative.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division</td>
<td>/</td>
</tr>
<tr>
<td>Integer Division</td>
<td>DIV</td>
</tr>
<tr>
<td>Modulus Division</td>
<td>%</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
</tr>
</tbody>
</table>

Division by zero defaults to generating a zero result (0), rather than reporting a "divide by zero" error. This avoids invalid or unexpected data aborting a long job. The default behaviour can be changed using

```
#OPTION ('divideByZero', 'zero'); //evaluate to zero
```

The divideByZero option can have the following values:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'zero'</td>
<td>Evaluate to 0 - the default behaviour.</td>
</tr>
<tr>
<td>'fail'</td>
<td>Stop and report a division by zero error.</td>
</tr>
<tr>
<td>'nan'</td>
<td>This is only currently supported for real numbers. Division by zero creates a quiet NaN, which will propagate through any real expressions it is used in. You can use NOT ISVALID(x) to test if the value is a NaN. Integer and decimal division by zero continue to return 0.</td>
</tr>
</tbody>
</table>

Bitwise Operators

Bitwise operators are supported for use in expressions, listed here in their evaluation precedence:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
</tr>
<tr>
<td>Bitwise OR</td>
<td></td>
</tr>
<tr>
<td>Bitwise Exclusive OR</td>
<td>^</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td>BNOT</td>
</tr>
</tbody>
</table>
Bitshift Operators

Bitshift operators are supported for use in integer expressions:

<table>
<thead>
<tr>
<th>Bitshift Right</th>
<th>&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitshift Left</td>
<td>&lt;&lt;</td>
</tr>
</tbody>
</table>

Comparison Operators

The following comparison operators are supported:

<table>
<thead>
<tr>
<th>Equivalence</th>
<th>=</th>
<th>returns TRUE or FALSE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Equal</td>
<td>&lt;&gt;</td>
<td>returns TRUE or FALSE</td>
</tr>
<tr>
<td>Not Equal</td>
<td>!=</td>
<td>returns TRUE or FALSE</td>
</tr>
<tr>
<td>Less Than</td>
<td>&lt;</td>
<td>returns TRUE or FALSE</td>
</tr>
<tr>
<td>Greater Than</td>
<td>&gt;</td>
<td>returns TRUE or FALSE</td>
</tr>
<tr>
<td>Less Than or Equal</td>
<td>&lt;=</td>
<td>returns TRUE or FALSE</td>
</tr>
<tr>
<td>Greater Than or Equal</td>
<td>&gt;=</td>
<td>returns TRUE or FALSE</td>
</tr>
<tr>
<td>Equivalence Comparison</td>
<td>&lt;= &gt;</td>
<td>returns -1, 0, or 1</td>
</tr>
</tbody>
</table>

The Greater Than or Equal operator must have the Greater Than (>) sign first. For the expression `a <=> b`, the Equivalence Comparison operator returns -1 if `a<b`, 0 if `a=b`, and 1 if `a>b`. When STRINGs are compared, trailing spaces are generally ignored. Standard library functions, such as Std.Str.Find(), may consider trailing spaces. See the Standard Library Reference for specific details.
Logical Operators

The following logical operators are supported, listed here in their evaluation precedence:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>Boolean NOT operation</td>
</tr>
<tr>
<td>~</td>
<td>Boolean NOT operation</td>
</tr>
<tr>
<td>AND</td>
<td>Boolean AND operation</td>
</tr>
<tr>
<td>OR</td>
<td>Boolean OR operation</td>
</tr>
</tbody>
</table>

Logical Expression Grouping

When a complex logical expression has multiple OR conditions, you should group the OR conditions and order them from least complex to most complex to result in the most efficient processing.

If the probability of occurrence is known, you should order them from the most likely to occur to the least likely to occur, because once any part of a compound OR condition evaluates to TRUE, the remainder of the expression can be bypassed. However, this is not guaranteed. This is also true of the order of MAP function conditions.

Whenever AND and OR logical operations are mixed in the same expression, you should use parentheses to group within the expression to ensure correct evaluation and to clarify the intent of the expression. For example consider the following:

```plaintext
isCurrentRevolv := trades.trd_type = 'R' AND
trades.trd_rate = '0' OR
trades.trd_rate = '1';
```

This does not produce the intended result. Use of parentheses ensures correct evaluation, as shown below:

```plaintext
isCurrentRevolv := trades.trd_type = 'R' AND
(trades.trd_rate = '0' OR trades.trd_rate = '1');
```

An XOR Operator

The following function can be used to perform an XOR operation on 2 Boolean values:

```plaintext
BOOLEAN XOR(BOOLEAN cond1, BOOLEAN cond2) :=
(cond1 OR cond2) AND NOT (cond1 AND cond2);
```
Record Set Operators

The following record set operators are supported (all require that the files were created using identical RECORD structures):

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Append all records from both files, independent of any order</td>
</tr>
<tr>
<td>&amp;</td>
<td>Append all records from both files, maintaining record order on each node</td>
</tr>
<tr>
<td>-</td>
<td>Subtract records from a file</td>
</tr>
</tbody>
</table>

Example:

```ecl
MyLayout := RECORD
    UNSIGNED Num;
    STRING Number;
END;

FirstRecSet := DATASET([{1, 'ONE'}, {2, 'Two'}, {3, 'Three'}, {4, 'Four'}], MyLayout);
SecondRecSet := DATASET([{5, 'FIVE'}, {6, 'SIX'}, {7, 'SEVEN'}, {8, 'EIGHT'}], MyLayout);

ExcludeThese := SecondRecSet(Num > 6);
WholeRecSet := FirstRecSet + SecondRecSet;
ResultSet := WholeRecSet-ExcludeThese;

OUTPUT (WholeRecSet);
OUTPUT(ResultSet);
```

Prefix Append Operator

\((+)(\text{ds\_list})[,\ \text{options}])\)

<table>
<thead>
<tr>
<th>(+)</th>
<th>The prefix append operator.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{ds_list}</td>
<td>A comma-delimited list of record sets to append (two or more). All the record sets must have identical RECORD structures.</td>
</tr>
<tr>
<td>\text{options}</td>
<td>Optional. A comma-delimited list of options from the list below.</td>
</tr>
</tbody>
</table>

The prefix append operator \((+)\) provides more flexibility than the simple infix operators described above. It allows hints and other options to be associated with the operator. Similar syntax will be added in a future change for other infix operators.

The following \text{options} may be used:

\[
[,\ \text{UNORDERED} | \text{ORDERED}(\ bool)\ [,\ \text{STABLE} | \text{UNSTABLE}] [,\ \text{PARALLEL} [\ (\ \text{numthreads}\ )]] [,\ \text{ALGORITHM}(\ name)]
\]

<p>| UNORDERED | Optional. Specifies the output record order is not significant. |
| ORDERED | Specifies the significance of the output record order. |
| \text{bool} | When False, specifies the output record order is not significant. When True, specifies the default output record order. |
| STABLE | Optional. Specifies the input record order is significant. |
| UNSTABLE | Optional. Specifies the input record order is not significant. |
| PARALLEL | Optional. Try to evaluate this activity in parallel. |</p>
<table>
<thead>
<tr>
<th><strong>numthreads</strong></th>
<th>Optional. Try to evaluate this activity using <em>numthreads</em> threads.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALGORITHM</strong></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

**Example:**

```ecl
ds_1 := (+)(ds1, ds2, UNORDERED);
//equivalent to: ds := ds1 + ds2;

ds_2 := (+)(ds1, ds2);
//equivalent to: ds := ds1 & ds2;

ds_3 := (+)(ds1, ds2, ds3);
//multiple file appends are supported
```
## Set Operators

The following set operators are supported, listed here in their evaluation precedence:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Append (all elements from both sets, without re-ordering or duplicate element removal)</td>
</tr>
</tbody>
</table>
String Operators

The following string operator is supported:

| +   | Concatenation |
IN Operator

value $\textbf{IN}$ value_set

<table>
<thead>
<tr>
<th>value</th>
<th>The value to find in the value_set. This is usually a single value, but if the value_set is a DICTIONARY with a multiple-component key, this may also be a ROW.</th>
</tr>
</thead>
<tbody>
<tr>
<td>value_set</td>
<td>A set of values. This may be a set expression, the SET function, or a DICTIONARY.</td>
</tr>
</tbody>
</table>

The IN operator is shorthand for a collection of OR conditions. It is an operator that will search a set to find an inclusion, resulting in a Boolean return. Using IN is much more efficient than the equivalent OR expression.

Examples:

```ecl
ABCset := ['A', 'B', 'C'];
IsABCStatus := Person.Status IN ABCset;
  //This code is directly equivalent to:
  // IsABCStatus := Person.Status = 'A' OR
  //                Person.Status = 'B' OR
  //                Person.Status = 'C';

IsABC(STRING1 char) := char IN ABCset;
Trades_ABCStat := Trades(IsABC(rate));
  // Trades_ABCstat is a record set definition of all those
  // trades with a trade status of A, B, or C

//SET function examples
r := {STRING1 Letter};
SomeFile := DATASET( [{'A'}, {'B'}, {'C'}, {'D'}, {'E'},
                     {'F'}, {'G'}, {'H'}, {'I'}, {'J'}], r);
x := SET(SomeFile(Letter > 'C'), Letter);
y := 'A' IN x;  //results in FALSE
z := 'D' IN x; //results in TRUE
```

//DICTIONARY examples:

```ecl
rec := {STRING color, UNSIGNED1 code};
ColorCodes := DATASET( [{'Black', 0},
                      {'Brown', 1},
                      {'Red', 2},
                      {'White', 3}], rec);
ColorCodesDCT := DICTIONARY(ColorCodes,{Color => Code});
OUTPUT(ROW({'Red',2},rec) IN ColorCodesDCT);
```

See Also: Basic Definition Types, Definition Types (Set Definitions), Logical Operators, PATTERN, DICTIONARY, ROW, SET, Sets and Filters, SET OF, Set Operators
**BETWEEN Operator**

*SeekVal BETWEEN LoVal AND HiVal*

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SeekVal</td>
<td>The value to find in the inclusive range.</td>
</tr>
<tr>
<td>LoVal</td>
<td>The low value in the inclusive range.</td>
</tr>
<tr>
<td>HiVal</td>
<td>The high value in the inclusive range.</td>
</tr>
</tbody>
</table>

The **BETWEEN** operator is shorthand for an inclusive range check using standard comparison operators (*SeekVal >= LoVal AND SeekVal <= HiVal*). It may be combined with NOT to reverse the logic.

**Example:**

```
X := 10;
Y := 20;
Z := 15;

IsInRange := Z BETWEEN X AND Y;
    //This code is directly equivalent to:
    // IsInRange := Z >= X AND Z <= Y;

IsNotInRange := Z NOT BETWEEN X AND Y;
    //This code is directly equivalent to:
    // IsNotInRange := NOT (Z >= X AND Z <= Y);

OUTPUT(IsInRange);
OUTPUT(IsNotInRange);
```

See Also: Logical Operators, Comparison Operators
Value Types

Value types declare an Attribute’s type when placed left of the Attribute name in the definition. They also declare a passed parameter’s type when placed left of the parameter name in the definition. Value types also explicitly cast from type to another when placed in parentheses left of the expression to cast.

BOOLEAN

A Boolean true/false value. TRUE and FALSE are reserved ECL keywords; they are Boolean constants that may be used to compare against a BOOLEAN type. When BOOLEAN is used in a RECORD structure, a single-byte integer containing one (1) or zero (0) is output.

Example:

```ecl
BOOLEAN MyBoolean := SomeAttribute > 10;
  // declares MyBoolean a BOOLEAN Attribute
BOOLEAN MyBoolean(INTEGER p) := p > 10;
  // MyBoolean takes an INTEGER parameter
BOOLEAN Typtrd := trades.trd_type = 'R';
  // Typtrd is a Boolean attribute, likely to be used as a filter
```

See Also: TRUE/FALSE
**INTEGER**

\[[\text{IntType}]\] [\text{UNSIGNED}] \text{INTEGER}[n]

\[[\text{IntType}]\] \text{UNSIGNED}n

An \(n\)-byte integer value. Valid values for \(n\) are: 1, 2, 3, 4, 5, 6, 7, or 8. If \(n\) is not specified for the INTEGER, the default is 8-bytes.

The optional \text{IntType} may specify either the \text{BIG_ENDIAN} (Sun/UNIX-type, valid only inside a RECORD structure) or \text{LITTLE_ENDIAN} (Intel-type) style of integers. These two \text{IntTypes} have opposite internal byte orders. If the \text{IntType} is missing, the integer is \text{LITTLE_ENDIAN}.

If the optional \text{UNSIGNED} keyword is missing, the integer is signed. Unsigned integer declarations may be contracted to \text{UNSIGNED}n instead of \text{UNSIGNED INTEGER}n.

**INTEGER Value Ranges**

<table>
<thead>
<tr>
<th>Size</th>
<th>Signed Values</th>
<th>Unsigned Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-byte</td>
<td>-128 to 127</td>
<td>0 to 255</td>
</tr>
<tr>
<td>2-byte</td>
<td>-32,768 to 32,767</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>3-byte</td>
<td>-8,388,608 to 8,388,607</td>
<td>0 to 16,777,215</td>
</tr>
<tr>
<td>4-byte</td>
<td>-2,147,483,648 to 2,147,483,647</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>5-byte</td>
<td>-549,755,813,888 to 549,755,813,887</td>
<td>to 0 to 1,099,511,627,775</td>
</tr>
<tr>
<td>6-byte</td>
<td>-140,737,488,355,328 to 140,737,488,355,327</td>
<td>to 0 to 281,474,976,710,655</td>
</tr>
<tr>
<td>7-byte</td>
<td>-36,028,797,018,963,968 to 36,028,797,018,963,967</td>
<td>to 0 to 72,057,594,037,927,935</td>
</tr>
<tr>
<td>8-byte</td>
<td>-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
<td>to 0 to 18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>

Example:

```
INTEGER1 MyValue := MAP(MyString = '1' => MyString, '0');
    //MyValue is 1 or 0, changing type from string to integer
UNSIGNED INTEGER1 MyValue := 255; //max value possible in 1 byte
UNSIGNED1 MyValue := 255;
    //MyValue contains the max value possible in a single byte
MyRec := RECORD
    LITTLE_ENDIAN INTEGER2 MyLittleEndianValue := 1;
    BIG_ENDIAN INTEGER2 MyBigEndianValue := 1;
    //the physical byte-order is opposite in these two
END
```
REAL

REAL[n]

An n-byte standard IEEE floating point value. Valid values for n are: 4 (values to 7 significant digits) or 8 (values to 15 significant digits). If n is omitted, REAL is a double-precision floating-point value (8-bytes).

REAL Value Ranges

<table>
<thead>
<tr>
<th>Type</th>
<th>Significant Digits</th>
<th>Largest Value</th>
<th>Smallest Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL4</td>
<td>7 (9999999)</td>
<td>3.402823e+038</td>
<td>1.175494e-038</td>
</tr>
<tr>
<td>REAL8</td>
<td>15 (999999999999999)</td>
<td>1.797693e+308</td>
<td>2.225074e-308</td>
</tr>
</tbody>
</table>

Example:

REAL4 MyValue := MAP(MyString = '1.0' => MyString, '0');
// MyValue becomes either 1.0 or 0
DECIMAL

[UNSIGNED] DECIMALn [ _y ]

UDECIMALn [ _y ]

A packed decimal value of \( n \) total digits. If the \(_y\) value is present, \( y \) defines the number of decimal places in the value. There can be at most 32 leading digits and 32 fractional digits.

If the UNSIGNED keyword is omitted, the rightmost nibble holds the sign. Unsigned decimal declarations may be contracted to use the optional UDECIMALn syntax instead of UNSIGNED DECIMALn.

Using exclusively DECIMAL values in computations invokes the Binary Coded Decimal (BCD) math libraries (base-10 math), allowing up to 32-digits of precision (which may be on either side of the decimal point).

Example:

```ecl
DECIMAL5_2 MyDecimal := 123.45;
    //five total digits with two decimal places

OutputFormat199 := RECORD
    UNSIGNED DECIMAL9 Person.SSN;
        //unsigned packed decimal containing 9 digits,
        // occupying 5 bytes in a flat file

    UDECIMAL10 Person.phone;
        //unsigned packed decimal containing 10 digits,
        // occupying 5 bytes in a flat file

END;
```
STRING

[StringType] STRING[n]

A character string of \( n \) bytes, space padded (not null-terminated). If \( n \) is omitted, the string is variable length to the size needed to contain the result of the cast or passed parameter. You may use set indexing into any string to parse out a substring.

The optional \( StringType \) may specify ASCII or EBCDIC. If the \( StringType \) is missing, the data is in ASCII format. Defining an EBCDIC STRING Attribute as a string constant value implies an ASCII to EBCDIC conversion.

The upper size limit for any STRING value is 4GB.

Example:

```ecl
STRING1 MyString := IF(SomeAttribute > 10,'1','0');
    // declares MyString a 1-byte ASCII string

EBCDIC STRING3 MyString1 := 'ABC';
    // implicit ASCII to EBCDIC conversion
OUTPUT(MyString);
OUTPUT(MyString1);
```

See Also: LENGTH, TRIM, Set Ordering and Indexing, Hexadecimal String
QSTRING

QSTRING[n]

A data-compressed variation of STRING that uses only 6-bits per character to reduce storage requirements for large strings. The character set is limited to capital letters A-Z, the numbers 0-9, the blank space, and the following set of special characters:

! * # $ % & ' ( ) + , - . / ; < = > ? @ \ ^ _

If n is omitted, the QSTRING is variable length to the size needed to contain the result of a cast or passed parameter. You may use set indexing into any QSTRING to parse out a substring.

The upper size limit for any QSTRING value is 4GB.

Example:

QSTRING12 CompanyName := 'LEXISNEXIS';
// uses only 9 bytes of storage instead of 12
OUTPUT(CompanyName)

See Also: STRING, LENGTH, TRIM, Set Ordering and Indexing.
### UNICODE

**UNICODE[._locale]\[n\]**

A UTF-16 encoded unicode character string of \( n \) characters, space-padded just as STRING is. If \( n \) is omitted, the string is variable length to the size needed to contain the result of the cast or passed parameter. The optional locale specifies a valid unicode locale code, as specified in ISO standards 639 and 3166 (not needed if LOCALE is specified on the RECORD structure containing the field definition).

Type casting UNICODE to VARUNICODE, STRING, or DATA is allowed, while casting to any other type will first implicitly cast to STRING and then cast to the target value type.

The upper size limit for any UNICODE value is 4GB.

Example:

```ecl
UNICODE16 MyUNIString := U'1234567890ABCDEF';
   // utf-16-encoded string
UNICODE4 MyUnicodeString := U'abcd';
   // same as: (UNICODE)'abcd'
UNICODE_de5 MyUnicodeString_de := U'abcd\353';
   // becomes 'abcdë' with a German locale
UNICODE_de5 MyUnicodeString_de5 := U'abcdé';
   // same as previous example
OUTPUT(MyUNIString);
OUTPUT(MyUnicodeString);
OUTPUT(MyUnicodeString_de);
OUTPUT(MyUnicodeString_de5);
```
UTF8

UTF8[locale][n]

A UTF-8 encoded unicode character string of \(n\) characters, space-padded just as STRING is. If \(n\) is omitted, the string is variable length to the size needed to contain the result of the cast or passed parameter. The optional \(locale\) specifies a valid unicode locale code, as specified in ISO standards 639 and 3166 (not needed if LOCALE is specified on the RECORD structure containing the field definition).

Type casting UTF8 to UNICODE, VARUNICODE, STRING, or DATA is allowed, while casting to any other type will first implicitly cast to STRING and then cast to the target value type.

The upper size limit for any UTF8 value is 4GB.

Example:

```ecl
UTF8 FirstName := U8'Noël';                 // utf-8-encoded string
UTF8_de MyUnicodeString := U8'abcd\353';    // becomes 'abcdë' with a German locale
UTF8_4 FirstName4 := U8'Noël';              // 4-character utf-8-encoded string
UTF8_de_5 MyUnicodeString_de5 := U8'abcd\353'; // becomes 'abcdë' with a German locale

OUTPUT(FirstName);
OUTPUT(MyUnicodeString);
OUTPUT(FirstName4);
OUTPUT(MyUnicodeString_de5);
```
**DATA**

**DATA[n]**

A "packed hexadecimal" data block of \( n \) bytes, zero padded (not space-padded). If \( n \) is omitted, the DATA is variable length to the size needed to contain the result of the cast or passed parameter. Type casting is allowed but only to a STRING or UNICODE of the same number of bytes.

This type is particularly useful for containing BLOB (Binary Large OBject) data. See the Programmer's Guide article **Working with BLOBs** for more information on this subject.

The upper size limit for any DATA value is 4GB.

Example:

```ecl
DATA8 MyHexString := x'1234567890ABCDEF';
// an 8-byte data block - hex values 12 34 56 78 90 AB CD EF
OUTPUT(MyHexString);
```
VARSTRING

VARSTRING[n]

A null-terminated character string containing $n$ bytes of data. If $n$ is omitted, the string is variable length to the size needed to contain the result of the cast or passed parameter. You may use set indexing into any string to parse out a substring.

The upper size limit for any VARSTRING value is 4GB.

Example:

```ecl
VARSTRING3 MyString := 'ABC';
   // declares MyString a 3-byte null-terminated string
OUTPUT(MyString);
```

See Also: LENGTH, TRIM, Set Ordering and Indexing
VARUNICODE

VARUNICODE[locale][n]

A UTF-16 encoded Unicode character string of \( n \) characters, null terminated (not space-padded). The \( n \) may be omitted only when used as a parameter type. The optional \( locale \) specifies a valid Unicode locale code, as specified in ISO standards 639 and 3166 (not needed if LOCALE is specified on the RECORD structure containing the field definition).

Type casting VARUNICODE to UNICODE, STRING, or DATA is allowed, while casting to any other type will first implicitly cast to STRING and then cast to the target value type.

The upper size limit for any VARUNICODE value is 4GB.

Example:
**SET OF**

**SET [ OF type ]**

<table>
<thead>
<tr>
<th>type</th>
<th>The value type of the data in the set. Valid value types are: INTEGER, REAL, BOOLEAN, STRING, UNICODE, DATA, or DATASET(recstruct). If omitted, the type is INTEGER.</th>
</tr>
</thead>
</table>

The **SET OF** value type defines Attributes that are a set of data elements. All elements of the set must be of the same value type. The default value for SET OF when used to define a passed parameter may be a defined set, the keyword ALL to indicate all possible values for that type of set, or empty square brackets ([ ]) to indicate no possible value for that type of set.

Example:

```ecl
SET OF INTEGER1 SetIntOnes := [1,2,3,4,5];
SET OF STRING1 SetStrOnes := ['1','2','3','4','5'];
SET OF STRING1 SetStrOne1 := (SET OF STRING1)SetIntOnes;
    //type casting sets is allowed
r := (STRING F1, STRING2 F2);
SET OF DATASET(r) SetDS := [ds1, ds2, ds3];
```

```ecl
StringSetFunc(SET OF STRING passedset) := AstringValue IN passedset;
    //a set of string constants will be passed to this function
HasNarCode(SET s) := Trades.trd_narr1 IN s OR Trades.trd_narr2 IN s;
    // HasNarCode takes a parameter that specifies the set of valid
    // Narrative Code values (all INTEGERS)
SET OF INTEGER1 SetClsdNar := [65,66,90,114,115,123];
NarCodeTrades := Trades(HasNarCode(SetClsdNar));
    // Using HasNarCode(SetClsdNar) is equivalent to:
    // Trades.trd_narr1 IN [65,66,90,114,115,123] OR
    // Trades.trd_narr2 IN [65,66,90,114,115,123]
```

See Also: Functions (Parameter Passing), Set Ordering and Indexing
**TYPEOF**

**TYPEOF( expression )**

| expression | An expression defining the value type. This may be the name of a data field, passed parameter, function, or Attribute providing the value type (including RECORD structures). This must be a legal expression for the current scope but is not evaluated for its value. |

The **TYPEOF** declaration allows you to define an Attribute or parameter whose value type is "just like" the **expression**. It is valid for use anywhere an explicit value type is valid.

Its most typical use would be to specify the return type of a TRANSFORM function as "just like" a dataset or recordset structure.

Example:

```ecl
STRING3 Fred := 'ABC'; //declare Fred as a 3-byte string
TYPEOF(Fred) Sue := Fred; //declare Sue as "just like" Fred
OUTPUT(Fred);
OUTPUT(Sue);
```

See Also: TRANSFORM Structure
RECORDOF

RECORDOF( recordset, [LOOKUP])

<table>
<thead>
<tr>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>The set of data records whose RECORD structure to use. This may be a DATASET or any derived recordset. If the LOOKUP attribute is used, this may be a filename.</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>Optional. Specifies that the file layout should be looked up at compile time. See File Layout Resolution at Compile Time in the Programmer’s Guide for more details.</td>
</tr>
</tbody>
</table>

The RECORDOF declaration specifies use of just the record layout of the recordset in those situations where you need to inherit the structure of the fields but not their default values, such as child DATASET declarations inside RECORD structures.

This function allows you to keep RECORD structures local to the DATASET whose layout they define and still be able to reference the structure (only, without default values) where needed.

Example:

```ecl
Layout_People_Slim := RECORD
  STD_People.RecID;
  STD_People.ID;
  STD_People.FirstName;
  STD_People.LastName;
  STD_People.MiddleName;
  STD_People.NameSuffix;
  STD_People.FileDate;
  STD_People.BureauCode;
  STD_People.Gender;
  STD_People.BirthDate;
  STD_People.StreetAddress;
  UNSIGNED8 CSZ_ID;
END;

STD_Accounts := TABLE(UID_Accounts, Layout_STD_AcctsFile);

CombinedRec := RECORD, MAXLENGTH(100000)
  Layout_People_Slim;
  UNSIGNED1 ChildCount;
  DATASET (RECORDOF (STD_Accounts)) ChildAccts;
END;
```

//This ChildAccts definition is equivalent to:
// DATASET(Layout_STD_AcctsFile) ChildAccts;
//but doesn't require Layout_STD_AcctsFile to be visible (SHARED or EXPORT)

See Also: DATASET, RECORD Structure
**Enumerations (ENUM)**

```
ENUM( [ type .] name [=value] [ , name [=value] ... ] )
```

<table>
<thead>
<tr>
<th><strong>type</strong></th>
<th>The numeric value type of the values. If omitted, defaults to UNSIGNED4.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
<td>The label of the enumerated value.</td>
</tr>
<tr>
<td><strong>value</strong></td>
<td>The numeric value to associate with the name. If omitted, the value is the previous value plus one (1). If all values are omitted, the enumeration starts with one (1).</td>
</tr>
</tbody>
</table>

The `ENUM` declaration specifies constant values to make code more readable.

**Example:**

```ecl
GenderEnum := ENUM(UNSIGNED1,Male,Female,NonBinary,Unknown);
   //values are 1, 2, 3, 4
personFlag := ENUM(None=0,Dead=1,Foreign=2,Terrorist=4, Wanted=Terrorist*2);
   //values are 0, 1, 2, 4, 8
namesRecord := RECORD
   STRING20 surname;
   STRING10 forename;
   GenderEnum gender;
   UNSIGNED1 personFlag := 0;
END;

namesTable2 := DATASET([ {'Boreman','George',GenderEnum.Male,personFlag.Foreign},
                         {'Bin','O',GenderEnum.Male,personFlag.Foreign+personFlag.Terrorist+personFlag.Wanted} ], namesRecord);
OUTPUT(namesTable2);
```
Type Casting

Explicit Casting

The most common use of value types is to explicitly cast from one type to another in expressions. To do this, you simply place the value type to cast to within parentheses. That creates a casting operator. Then place that casting operator immediately to the left of the expression to cast.

This converts the data from its original form to the new form (to keep the same bit-pattern, see the TRANSFER built-in function).

```ecl
SomeAttribute := 11;
MyBoolean := (BOOLEAN) IF(SomeAttribute > 10, 1, 0);
  // casts the INTEGER values 1 and 0 to a BOOLEAN TRUE or FALSE
MyString := (STRING1) IF(SomeAttribute > 10, 1, 0);
  // casts the INTEGER values 1 and 0 to a 1-character string
  // containing '1' or '0'
MyValue := (INTEGER) MAP(MyString = '1' => MyString, '0');
  // casts the STRING values '1' and '0' to an INTEGER 1 or 0
MySet := (SET OF INTEGER1) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  // casts from a SET OF INTEGER8 (the default) to SET OF INTEGER1
OUTPUT(MyBoolean);
OUTPUT(MyString);
OUTPUT(MyValue);
OUTPUT(MySet);
```

Implicit Casting

During expression evaluation, different value types may be implicitly cast in order to properly evaluate the expression. Implicit casting always means promoting one value type to another: INTEGER to STRING or INTEGER to REAL. BOOLEAN types may not be involved in mixed mode expressions. For example, when evaluating an expression using both INTEGER and REAL values, the INTEGER is promoted to REAL at the point where the two mix, and the result is a REAL value.

INTEGER and REAL may be freely mixed in expressions. At the point of contact between them the expression is treated as REAL. Until that point of contact the expression may be evaluated at INTEGER width. Division on INTEGER values implicitly promotes both operands to REAL before performing the division.

The following expression: \((1+2+3+4) \cdot (1.0 \cdot 5)\)

evaluates as: \((\text{REAL})(\text{INTEGER}1+\text{INTEGER}2+\text{INTEGER}3+\text{INTEGER}4) \cdot (1.0 \cdot \text{REAL}5)\)

and: \(5/2+4+5\) evaluates as: \((\text{REAL}5/\text{REAL}2+\text{REAL}4+\text{REAL}5)\)

while: \('5' + 4\) evaluates as: \(5 + \text{STRING4} \ //\text{concatenation}\)

Comparison operators are treated as any other mixed mode expression. Built-in Functions that take multiple values, any of which may be returned (such as MAP or IF), are treated as mixed mode expressions and will return the common base type. This common type must be reachable by standard implicit conversions.

Type Transfer

Type casting converts data from its original form to the new form. To keep the same bit-pattern you must use either the TRANSFER built-in function or the type transfer syntax, which is similar to type casting syntax with the addition of angle brackets (<valuetype>).
### Casting Rules

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Results in</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>STRING</td>
<td>ASCII or EBCDIC representation of the value</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>STRING</td>
<td>ASCII or EBCDIC representation of the value, including decimal and sign</td>
</tr>
<tr>
<td>REAL</td>
<td>STRING</td>
<td>ASCII or EBCDIC representation of the value, including decimal and sign--may be expressed in scientific notation</td>
</tr>
<tr>
<td>UNICODE</td>
<td>STRING</td>
<td>ASCII or EBCDIC representation with any non-existent characters appearing as the SUBstitute control code (0x1A in ASCII or 0x3F in EBCDIC) and any non-valid ASCII or EBCDIC characters appearing as the substitution codepoint (0xFFFD)</td>
</tr>
<tr>
<td>UTF8</td>
<td>STRING</td>
<td>ASCII or EBCDIC representation with any non-existent characters appearing as the SUBstitute control code (0x1A in ASCII or 0x3F in EBCDIC) and any non-valid ASCII or EBCDIC characters appearing as the substitution codepoint (0xFFFD)</td>
</tr>
<tr>
<td>STRING</td>
<td>QSTRING</td>
<td>Uppercase ASCII representation</td>
</tr>
<tr>
<td>INTEGER</td>
<td>UNICODE</td>
<td>UNICODE representation of the value</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>UNICODE</td>
<td>UNICODE representation of the value, including decimal and sign</td>
</tr>
<tr>
<td>REAL</td>
<td>UNICODE</td>
<td>UNICODE representation of the value, including decimal and sign--may be expressed in scientific notation</td>
</tr>
<tr>
<td>INTEGER</td>
<td>UTF8</td>
<td>UTF8 representation of the value</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>UTF8</td>
<td>UTF8 representation of the value, including decimal and sign</td>
</tr>
<tr>
<td>REAL</td>
<td>UTF8</td>
<td>UTF8 representation of the value, including decimal and sign--may be expressed in scientific notation</td>
</tr>
<tr>
<td>INTEGER</td>
<td>REAL</td>
<td>Value is cast with loss of precision when the value is greater than 15 significant digits</td>
</tr>
<tr>
<td>INTEGER</td>
<td>REAL4</td>
<td>Value is cast with loss of precision when the value is greater than 7 significant digits</td>
</tr>
<tr>
<td>STRING</td>
<td>REAL</td>
<td>Sign, integer, and decimal portion of the string value</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>REAL</td>
<td>Value is cast with loss of precision when the value is greater than 15 significant digits</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>REAL4</td>
<td>Value is cast with loss of precision when the value is greater than 7 significant digits</td>
</tr>
<tr>
<td>INTEGER</td>
<td>DECIMAL</td>
<td>Loss of precision if the DECIMAL is too small</td>
</tr>
<tr>
<td>REAL</td>
<td>DECIMAL</td>
<td>Loss of precision if the DECIMAL is too small</td>
</tr>
<tr>
<td>STRING</td>
<td>DECIMAL</td>
<td>Sign, integer, and decimal portion of the string value</td>
</tr>
<tr>
<td>STRING</td>
<td>INTEGER</td>
<td>Sign and integer portions of the string value</td>
</tr>
<tr>
<td>REAL</td>
<td>INTEGER</td>
<td>Integer value, only--decimal portion is truncated</td>
</tr>
</tbody>
</table>
### Value Types

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>INTEGER</th>
<th>Value is cast with no translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>BOOLEAN</td>
<td>0 = FALSE, anything else = TRUE</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>INTEGER</td>
<td>FALSE = 0, TRUE = 1</td>
</tr>
<tr>
<td>STRING</td>
<td>BOOLEAN</td>
<td>&quot; = FALSE, anything else = TRUE</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>STRING</td>
<td>FALSE = &quot;, TRUE = '1'</td>
</tr>
<tr>
<td>DATA</td>
<td>STRING</td>
<td>Value is cast with no translation</td>
</tr>
<tr>
<td>STRING</td>
<td>DATA</td>
<td>Value is cast with no translation</td>
</tr>
<tr>
<td>DATA</td>
<td>UNICODE</td>
<td>Value is cast with no translation</td>
</tr>
<tr>
<td>UNICODE</td>
<td>DATA</td>
<td>Value is cast with no translation</td>
</tr>
<tr>
<td>DATA</td>
<td>UTF8</td>
<td>Value is cast with no translation</td>
</tr>
<tr>
<td>UTF8</td>
<td>DATA</td>
<td>Value is cast with no translation</td>
</tr>
<tr>
<td>UTF8</td>
<td>UNICODE</td>
<td>Value is cast with no translation</td>
</tr>
<tr>
<td>UNICODE</td>
<td>UTF8</td>
<td>Value is cast with no translation</td>
</tr>
</tbody>
</table>

The casting rules for STRING to and from any numeric type apply equally to all string types, also. All casting rules apply equally to sets (using the SET OF type syntax).
Record Structures and Files

**RECORD Structure**

\[ attr := \text{RECORD} \left[ \left( \text{baserec} \right) \right], \text{MAXLENGTH}(length)], \text{LOCALE}(locale) \\left[ \left( \text{PACKED} \right) \right] \]

\[ \text{fields} ; \]

\[ \left[ \text{IFBLOCK}(\text{condition}) \right] \]

\[ \text{fields} ; \]

\[ \text{END;} \]

\[ \left[ \Rightarrow \text{payload} \right] \]

\[ \text{END;} \]

<table>
<thead>
<tr>
<th>attr</th>
<th>The name of the RECORD structure for later use in other definitions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>baserec</td>
<td>Optional. The name of a RECORD structure from which to inherit all fields. Any RECORD structure that inherits the baserec fields in this manner becomes compatible with any TRANSFORM function defined to take a parameter of baserec type (the extra fields will, of course, be lost).</td>
</tr>
<tr>
<td>MAXLENGTH</td>
<td>Optional. This option is used to create indexes that are backward compatible for platform versions prior to 3.0. Specifies the maximum number of characters allowed in the RECORD structure or field. MAXLENGTH on the RECORD structure overrides any MAXLENGTH on a field definition, which overrides any MAXLENGTH specified in the TYPE structure if the datatype names an alien data type. This option defines the maximum size of variable-length records. If omitted, fixed size records use the minimum size required and variable length records produce a warning. The default maximum size of a record containing variable-length fields is 4096 bytes (this may be overridden by using #OPTION(maxLength,####) to change the default). The maximum record size should be set as conservatively as possible, and is better set on a per-field basis (see the Field Modifiers section below).</td>
</tr>
<tr>
<td>length</td>
<td>An integer constant specifying the maximum number of characters allowed.</td>
</tr>
<tr>
<td>LOCALE</td>
<td>Optional. Specifies the Unicode locale for any UNICODE fields.</td>
</tr>
<tr>
<td>locale</td>
<td>A string constant containing a valid locale code, as specified in ISO standards 639 and 3166.</td>
</tr>
<tr>
<td>PACKED</td>
<td>Optional. Specifies the order of the fields may be changed to improve efficiency (such as moving variable-length fields after the fixed-length fields).</td>
</tr>
<tr>
<td>IFBLOCK</td>
<td>Optional. A block of fields that receive “live” data only if the condition is met. The IFBLOCK must be terminated by an END. This is used to define variable-length records. If the condition expression references fields in the RECORD preceding the IFBLOCK, those references must use SELF. prepend- ed to the fieldname to disambiguate the reference. This is only for use in RECORD structures that define files on disk.</td>
</tr>
<tr>
<td>fields</td>
<td>Field declarations. See below for the appropriate syntaxes.</td>
</tr>
</tbody>
</table>
Record Structures and Files

**condition**
A logical expression that defines when the *fields* within the IFBLOCK receive "live" data. If the expression is not true, the *fields* receive their declared default values. If there's no default value, the *fields* receive blanks or zeros.

**=>**
Optional. The delimiter between the list of key *fields* and the *payload* when the RECORD structure is used by the DICTIONARY declaration. Typically, this is an inline structure using curly braces ({ }) instead of RECORD and END.

**payload**
The list of non-keyed *fields* in the DICTIONARY.

Record layouts are definitions whose expression is a RECORD structure terminated by the END keyword. The *attr* name creates a user-defined value type that can be used in built-in functions and TRANSFORM function definitions. The delimiter between field definitions in a RECORD structure can be either the semi-colon (;) or a comma (,).

IFBLOCK and alien data types (TYPE) should only be used when accessing external data files. It is much more efficient to use the native types for general processing. In particular, some optimizations to project and filter files remotely are not supported on IFBLOCKs and alien datatypes.

### In-line Record Definitions

Curly braces ({}) are lexical equivalents to the keywords RECORD and END that can be used anywhere RECORD and END are appropriate. Either form (RECORD/END or {}) can be used to create "on-the-fly" record formats within those functions that require record structures (OUTPUT, TABLE, DATASET etc.), instead of defining the record as a separate definition.

If the RECORD structure is defining an INDEX, you can also use the "results in" operator (=>) before payload fields as shown below.

```plaintext
VehicleKey2 := INDEX(Vehicles,SearchTerms,Payload,'vkey::st.city2');
BUILD(VehicleKey2);

// Using "results in" operator (=>) for payload fields
VehicleKey3 := INDEX(Vehicles,{st,city => lname},'vkey::st.city3');
BUILD(VehicleKey3);
```

### Field Definitions

All field declarations in a RECORD Structure must use one of the following syntaxes:

```plaintext
datatype identifier [ {modifier} ][ := defaultvalue] ;
identifier := defaultvalue ;
defaultvalue ;
sourcelfield ;
recrecstruct [ identifier ] ;
sourcedataset ;
childdataset identifier [ { modifier } ];
```

- **datatype**
The value type of the data field. This may be a child dataset (see DATASET). If omitted, the value type is the result type of the *defaultvalue* expression.

- **identifier**
The name of the field. If omitted, the *defaultvalue* expression defines a column with no name that may not be referenced in subsequent ECL.
Field definitions must always define the datatype and identifier of each field, either implicitly or explicitly. If the RECORD structure will be used by TABLE, PARSE, ROW, or any other function that creates an output recordset, then the defaultvalue must also be implicitly or explicitly defined for each field. In the case where a field is defined in terms of a field in a dataset already in scope, you may name the identifier with a name already in use in the dataset already in scope as long as you explicitly define the datatype.

Field Inheritance

Field definitions may be inherited from a previously defined RECORD structure or DATASET. When a recstruct (a RECORD Structure) is specified from which to inherit the fields, the new fields are implicitly defined using the datatype and identifier of all the existing field definitions in the recstruct. When a sourcedataset (a previously defined DATASET or recordset definition) is specified to inherit the fields, the new fields are implicitly defined using the datatype, identifier, and defaultvalue of all the fields (making it usable by operations that require a data source, such as TABLE and PARSE). Either of these forms may optionally have its own identifier to allow reference to the entire set of inherited fields as a single entity.

You may also use logical operators (AND, OR, and NOT) to include/exclude certain fields from the inheritance, as described here:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 AND R2</td>
<td>Intersection</td>
<td>All fields declared in both R1 and R2</td>
</tr>
<tr>
<td>R1 OR R2</td>
<td>Union</td>
<td>All fields declared in either R1 or R2</td>
</tr>
<tr>
<td>R1 AND NOT R2</td>
<td>Difference</td>
<td>All fields in R1 that are not in R2</td>
</tr>
<tr>
<td>R1 AND NOT F1</td>
<td>Exception</td>
<td>All fields in R1 except the specified field (F1)</td>
</tr>
<tr>
<td>R1 AND NOT [F1, F2]</td>
<td>Exception</td>
<td>All fields in R1 except those in listed in the brackets (F1 and F2)</td>
</tr>
</tbody>
</table>

The minus sign (-) is a synonym for AND NOT, so R1-R2 is equivalent to R1 AND NOT R2.

It is an error if the records contain the same field names whose value types don't match, or if you end up with no fields (such as: A-A). You must ensure that any MAXLENGTH/MAXCOUNT is specified correctly on each field in both RECORD Structures.

Example:

R1 := (STRING1 F1,STRING1 F2,STRING1 F3,STRING1 F4,STRING1 F5);
R2 := (STRING1 F4,STRING1 F5,STRING1 F6);
R3 := (R1 AND R2); //Intersection - fields F4 and F5 only
Record Structures and Files

R4 := \{R1 OR R2\}; //Union - all fields F1 - F6
R5 := \{R1 AND NOT R2\}; //Difference - fields F1 - F3
R6 := \{R1 AND NOT F1\}; //Exception - fields F2 - F5
R7 := \{R1 AND NOT \{F1,F2\}\}; //Exception - fields F3 - F5

//the following two RECORD structures are equivalent:
C := RECORD,MAXLENGTH(x)
    R1 OR R2;
END;

D := RECORD, MAXLENGTH(x)
    R1;
    R2 AND NOT R1;
END;

Field Modifiers

The following list of field modifiers are available for use on field definitions:

- \{ MAXLENGTH( length ) \} Specifies the maximum number of characters allowed in the field (see MAXLENGTH option above).
- \{ MAXCOUNT( records ) \} Specifies the maximum number of records allowed in a child DATASET field (similar to MAXLENGTH above).
- \{ XPATH( 'tag' ) \} Specifies the XML or JSON tag that contains the data, in a RECORD structure that defines XML or JSON data. This overrides the default tag name (the lowercase field identifier). See the XPATH Support section below for details.
- \{ XMLDEFAULT( 'value' ) \} Specifies a default XML value for the field. The value must be constant.
- \{ DEFAULT( value ) \} Specifies a default value for the field. The value must be constant:
  1. When a DICTIONARY lookup returns no match.
  2. When an out-of-range record is fetched using ds[n] (as in ds[5] when ds contains only 4 records).
  3. In the default records passed to TRANSFORM functions in non-INNER JOINS where there is no corresponding row.
  4. When defaulting field values in a TRANSFORM using SELF = [].
- \{ BLOB \}
Record Structures and Files

{ VIRTUAL( fileposition ) }

Specifies the field is a VIRTUAL field containing the relative byte position of the record within the entire file (the record pointer). This must be an UNSIGNED8 field and must be the last field, because it only truly exists when the file is loaded into memory from disk (hence, the "virtual").

{ VIRTUAL( localfileposition ) }

Specifies the local byte position within a part of the distributed file on a single node: the first bit is set, the next 15 bits specify the part number, and the last 48 bits specify the relative byte position within the part. This must be an UNSIGNED8 field and must be the last field, because it only truly exists when the file is loaded into memory from disk (hence, the "virtual").

{ VIRTUAL( logicalfilename ) }

Specifies the logical file name of the distributed file. This must be a STRING field. If reading from a superfile, the value is the current logical file within the superfile.

{ BLOB }

Specifies the field is stored separately from the leaf node entry in the INDEX. This is applicable specifically to fields in the payload of an INDEX to allow more than 32K of data per index entry. The BLOB data is stored within the index file, but not with the rest of the record. Accessing the BLOB data requires an additional seek.

XPATH Support

XPATH support is a limited subset of the full XPATH specification, basically expressed as:

node[qualifier] / node[qualifier] ...

<table>
<thead>
<tr>
<th>node</th>
<th>Can contain wildcards.</th>
</tr>
</thead>
<tbody>
<tr>
<td>qualifier</td>
<td>Can be a node or attribute, or a simple single expression of equality, inequality, or numeric or alphanumeric comparisons, or node index values. No functions or inline arithmetic, etc. are supported. String comparison is indicated when the right hand side of the expression is quoted.</td>
</tr>
</tbody>
</table>

These operators are valid for comparisons:

<, <=, >, >=, =, !=

An example of a supported xpath:

/a/*/c/*d/e[@attr]/f[child]/g[@attr="x"]/h[child="5"]/i[@x="2"]/j

You can emulate AND conditions like this:

/a/b[@x="1"][@y="2"]

Also, there is a non-standard XPATH convention for extracting the text of a match using empty angle brackets (<>):

R := RECORD
STRING blah(xpath('a/b<>'));
//contains all of b, including any child definitions and values
END;

An XPATH for a value cannot be ambiguous. If the element occurs multiple times, you must use the ordinal operation (for example, /foo[1]/bar) to explicit select the first occurrence.
For XML or JSON DATASETs reading and processing results of the SOAPCALL function, the following XPATH syntax is specifically supported:

1) For simple scalar value fields, if there is an XPATH specified then it is used, otherwise the lower case identifier of the field is used.

```
STRING name;                  //matches: <name>Kevin</name>
STRING FName{xpath('Fname')}; //matches: <Fname>Kevin</Fname>
```

2) For a field whose type is a RECORD structure, the specified XPATH is prefixed to all the fields it contains, otherwise the lower case identifier of the field followed by '/' is prefixed onto the fields it contains. Note that an XPATH of '' (empty single quotes) will prefix nothing.

```
NameRec := RECORD
    STRING FName{xpath('Fname')}; //matches: <Fname>Kevin</Fname>
    STRING Mname{xpath('Mname')}; //matches: <Mname>Alfonso</Mname>
    STRING Lname{xpath('Lname')}; //matches: <Lname>Jones</Lname>
END;
```

3) For a child DATASET field, the specified XPATH can have one of two formats: "Container/Repeated" or "/Repeated." Each "/Repeated" tag within the optional Container is iterated to provide the values. If no XPATH is specified, then the default value for the Container is the lower case field name, and the default value for Repeated is "Row." For example, this demonstrates "Container/Repeated":

```
DATASET(PeopleNames) People{xpath('people/name')};
    /*matches: <people>
        <name>Gavin</name>
        <name>Ricardo</name>
    </people> */
```

4) For a SET OF type field, an xpath on a set field can have one of three formats: "Repeated", "Container/Repeated" or "Container/Repeated/@attr". They are processed in a similar way to datasets, except for the following. If Container is specified, then the XML reading checks for a tag "Container/All" and if present the set contains all possible values. The third form allows you to read XML attribute values.

```
SET OF STRING people;
    //matches: <people><All/></people>
    //or: <people><Item>Kevin</Item><Item>Richard</Item></people>
```
SET OF STRING Npeople{xpath('Name')};
//matches: <Name>Kevin</Name><Name>Richard</Name>

SET OF STRING Xpeople{xpath('/Name/@id')};
//matches: <Name id='Kevin'/><Name id='Richard'/>

For writing XML or JSON files using OUTPUT, the rules are similar with the following exceptions:

- For scalar fields, simple tag names and XML/JSON attributes are supported.
- For SET fields, <All> will only be generated if the container name is specified.
- xpath filters are not supported.
- The "Container/Repeated/@attr" form for a SET is not supported.

Example:

For DATASET or the result type of a TRANSFORM function, you need only specify the value type and name of each field in the layout:

R1 := RECORD
  UNSIGNED1 F1; //only value type and name required
  UNSIGNED4 F2;
  STRING100 F3;
END;

D1 := DATASET('~LR::SomeFile',R1,THOR);

For "vertical slice" TABLE, you need to specify the value type, name, and data source for each field in the layout:

R2 := RECORD
  UNSIGNED1 F1 := D1.F1; //value type, name, data source all explicit
  D1.F2; //value type, name, data source all implicit
END;

T1 := TABLE(D1,R2);

For "crosstab report" TABLE:

R3 := RECORD
  D1.F1; //"group by" fields must come first
  UNSIGNED4 GrpCount := COUNT(GROUP);
    //value type, column name, and aggregate
  GrpSum := SUM(GROUP,D1.F2); //no value type -- defaults to INTEGER
    //no column name in output
  MAX(GROUP,D1.F2);
END;

T2 := TABLE(D1,R3,F1);

Form1 := RECORD
  Person.per_last_name; //field name is per_last_name - size
    //is as declared in the person dataset
  STRING25 LocalID := Person.per_first_name;
    //the name of this field is LocalID and it
    //gets its data from Person.per_first_name
  INTEGER8 COUNT(Trades); //this field is unnamed in the output file
  BOOLEAN HasBogey := FALSE;
    //HasBogey defaults to false
  REAL4 Valu8024;
    //value from the Valu8024 definition
END;
Form2 := RECORD
  Trades; //include all fields from the Trades dataset at their
  // already-defined names, types and sizes
  UNSIGNED8 fpos {VIRTUAL(fileposition)};
  //contains the relative byte position within the file
END;

Form3 := {Trades, UNSIGNED8 local_fpos {VIRTUAL(localfileposition)}};
//use of {} instead of RECORD/END
//"Trades" includes all fields from the dataset at their
// already-defined names, types and sizes
//local_fpos is the relative byte position in each part

Form4 := RECORD, MAXLENGTH(10000)
  STRING VarStringName1{MAXLENGTH(5000)};
  //this field is variable size to a 5000 byte maximum
  STRING VarStringName2{MAXLENGTH(4000)};
  //this field is variable size to a 4000 byte maximum
  IFBLOCK(MyCondition = TRUE) //following fields receive values
  //only if MyCondition = TRUE
    BOOLEAN HasLife := TRUE;
    //defaults to true unless MyCondition = FALSE
    INTEGER8 COUNT(Inquiries);
    //this field is zero if MyCondition = FALSE, even
    //if there are inquiries to count
END;

in-line record structures, demonstrating same field name use

ds := DATASET('d', { STRING s; }, THOR);
t := TABLE(ds, { STRING60 s := ds.s; });
// new "s" field is OK with value type explicitly defined

"Child dataset" RECORD structures

ChildRec := RECORD
  UNSIGNED4 person_id;
  STRING20 per_surname;
  STRING20 per_forename;
END;

ParentRecord := RECORD
  UNSIGNED8 id;
  STRING20 address;
  STRING20 CSZ;
  STRING10 postcode;
  UNSIGNED2 numKids;
  DATASET(ChildRec) children{MAXCOUNT(100)};
END;

an example using {XPATH('tag')}

R := RECORD
  STRING10 fname;
  STRING12 lname;
  SET OF STRING1 MySet{XPATH('Set/Element')} //define set tags
END;
B := DATASET([{'Fred','Bell',['A','B']},
  {'George','Blanda',['C','D']},
  {'Sam', '', ['E', 'F'] } ], R);
OUTPUT(B,,~LR::test.xml', XML,OVERWRITE);
/* this example produces XML output that looks like this:
<Dataset>
<Row><fname>Fred</fname><lname>Bell</lname>
<Set><Element>A</Element><Element>B</Element></Set></Row>
<Row><fname>George</fname><lname>Blanda</lname>
<Set><Element>C</Element><Element>D</Element></Set></Row>
<Row><fname>Sam</fname> </Row>
<Set><Element>E</Element><Element>F</Element></Set></Row>
</Dataset>
*/

another XML example with a 1-field child dataset

cr := RECORD,MAXLENGTH(1024)
    STRING phoneEx{XPATH('')};
END;
r := RECORD,MAXLENGTH(4096)
    STRING id{XPATH('COMP-ID')};
cr := RECORD,MAXLENGTH(1024)
    STRING phoneEx{XPATH('')};
END;
DS := DATASET([{'1002','1352,9493', ['1352','9493']},
{'1003','4846,4582,0779', ['4846','4582','0779']}],r);

XPATH can also be used to define a JSON file

/* a JSON file called "mybooks.json" contains this data: */
{
    "id" : "978-0641723445",
    "name" : "The Lightning Thief",
    "author" : "Rick Riordan"
}
BookRec := RECORD
  STRING ID {XPATH('id')}; //data from id tag -- renames field to uppercase
  STRING title {XPATH('name')}; //data from name tag, renaming the field
  STRING author; //data from author tag, tag name is lowercase and matches field name
END;
books := DATASET('~LR::mybooks.json',BookRec,JSON('/'));
OUTPUT(books);

See Also: DATASET, DICTIONARY, INDEX, OUTPUT, TABLE, TRANSFORM Structure, TYPE Structure, SOAPCALL
DATASET

\[\text{attr} := \text{DATASET}(\text{file, struct, filetype \{LOOKUP\}});\]

\[\text{attr} := \text{DATASET}(\text{dataset, file, filetype \{LOOKUP\}});\]

\[\text{attr} := \text{DATASET}(\text{WORKUNIT([wuid, ] namedoutput), struct});\]

\[\text{DATASET(row)}\]

\[\text{DATASET(childstruct \{\text{COUNT(count) \| LENGTH(size) \{CHOOSEN(maxrecs)\}}\}}\]

\[\text{[GROUPED] [LINKCOUNTED] [STREAMED] DATASET(struct)}\]

\[\text{DATASET(dict)}\]

\[\text{DATASET(count, transform \{\text{DISTRIBUTED | LOCAL}\}})\]

<table>
<thead>
<tr>
<th>attr</th>
<th>The name of the DATASET for later use in other definitions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>A string constant containing the logical file name. See the Scope &amp; Logical Filenames section for more on logical filenames.</td>
</tr>
<tr>
<td>struct</td>
<td>The RECORD structure defining the layout of the fields. This may use RECORDOF.</td>
</tr>
<tr>
<td>filetype</td>
<td>One of the following keywords, optionally followed by relevant options for that specific type of file: THOR /FLAT, CSV, XML, JSON, PIPE. Each of these is discussed in its own section, below.</td>
</tr>
<tr>
<td>dataset</td>
<td>A previously-defined DATASET or recordset from which the record layout is derived. This form is primarily used by the BUILD action and is equivalent to: [ds := \text{DATASET('filename',RECORDOF(anotherdataset), ... )}]</td>
</tr>
<tr>
<td>LOOKUP</td>
<td>Optional. Specifies that the file layout should be looked up at compile time. See File Layout Resolution at Compile Time in the Programmer’s Guide for more details.</td>
</tr>
<tr>
<td>WORKUNIT</td>
<td>Specifies the DATASET is the result of an OUTPUT with the NAMED option within the same or another workunit.</td>
</tr>
<tr>
<td>wuid</td>
<td>Optional. A string expression that specifies the workunit identifier of the job that produced the NAMED OUTPUT.</td>
</tr>
<tr>
<td>namedoutput</td>
<td>A string expression that specifies the name given in the NAMED option.</td>
</tr>
</tbody>
</table>
| recordset          | A set of in-line data records. This can simply name a previously-defined set definition or explicitly use square brackets to indicate an in-line set definition. Within the square brackets records are separated by commas. The records are specified by either:  

  1) Using curly braces (\{\}) to surround the field values for each record. The field values within each record are comma-delimited.  

  2) A comma-delimited list of in-line transform functions that produce the data rows. All the transform functions in the list must produce records in the same result format. |
**Record Structures and Files**

<table>
<thead>
<tr>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>recstruct</td>
<td>Optional. The RECORD structure of the recordset. Omittable only if the recordset parameter is just one record or a list of in-line transform functions.</td>
</tr>
<tr>
<td>row</td>
<td>A single data record. This may be a single-record passed parameter, or the ROW or PROJECT function that defines a 1-row dataset.</td>
</tr>
<tr>
<td>childstruct</td>
<td>The RECORD structure of the child records being defined. This may use the RECORDOF function.</td>
</tr>
<tr>
<td>COUNT</td>
<td>Optional. Specifies the number of child records attached to the parent (for use when interfacing to external file formats).</td>
</tr>
<tr>
<td>count</td>
<td>An expression defining the number of child records. This may be a constant or a field in the enclosing RECORD structure (addressed as SELF.fieldname).</td>
</tr>
<tr>
<td>LENGTH</td>
<td>Optional. Specifies the size of the child records attached to the parent (for use when interfacing to external file formats).</td>
</tr>
<tr>
<td>size</td>
<td>An expression defining the size of child records. This may be a constant or a field in the enclosing RECORD structure (addressed as SELF.fieldname).</td>
</tr>
<tr>
<td>CHOSEN</td>
<td>Optional. Limits the number of child records attached to the parent. This implicitly uses the CHOOSEN function wherever the child dataset is read.</td>
</tr>
<tr>
<td>maxrecs</td>
<td>An expression defining the maximum number of child records for a single parent.</td>
</tr>
<tr>
<td>GROUPED</td>
<td>Specifies the DATASET being passed has been grouped using the GROUP function.</td>
</tr>
<tr>
<td>LINKCOUNTED</td>
<td>Specifies the DATASET being passed or returned uses the link counted format (each row is stored as a separate memory allocation) instead of the default (embedded) format where the rows of a dataset are all stored in a single block of memory. This is primarily for use in BEGINC++ functions or external C++ library functions.</td>
</tr>
<tr>
<td>STREAMED</td>
<td>Specifies the DATASET being returned is returned as a pointer to an IRowStream interface (see the echelper.hpp include file for the definition). Valid only as a return type. This is primarily for use in BEGINC++ functions or external C++ library functions.</td>
</tr>
<tr>
<td>struct</td>
<td>The RECORD structure of the dataset field or parameter. This may use the RECORDOF function.</td>
</tr>
<tr>
<td>dict</td>
<td>The name of a DICTIONARY definition.</td>
</tr>
<tr>
<td>count</td>
<td>An integer expression specifying the number of records to create.</td>
</tr>
<tr>
<td>transform</td>
<td>The TRANSFORM function that will create the records. This may take an integer COUNTER parameter.</td>
</tr>
<tr>
<td>DISTRIBUTED</td>
<td>Optional. Specifies distributing the created records across all nodes of the cluster. If omitted, all records are created on node 1.</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Optional. Specifies records are created on every node.</td>
</tr>
</tbody>
</table>

The **DATASET** declaration defines a file of records, on disk or in memory. The layout of the records is specified by a RECORD structure (the struct or recstruct parameters described above). The distribution of records across execution nodes is undefined in general, as it depends on how the DATASET came to be (sprayed in from a landing zone or written to disk by an OUTPUT action), the size of the cluster on which it resides, and the size of the cluster on which it is used (to specify distribution requirements for a particular operation, see the DISTRIBUTE function).
The first two forms are alternatives to each other and either may be used with any of the filetypes described below (THOR/FLAT, CSV, XML, JSON, PIPE).

The third form defines the result of an OUTPUT with the NAMED option within the same workunit or the workunit specified by the wuid (see Named Output DATASETs below).

The fourth form defines an in-line dataset (see In-line DATASETs below).

The fifth form is only used in an expression context to allow you to in-line a single record dataset (see Single-row DATASET Expressions below).

The sixth form is only used as a value type in a RECORD structure to define a child dataset (see Child DATASETs below).

The seventh form is only used as a value type to pass DATASET parameters (see DATASET as a Parameter Type below).

The eighth form is used to define a DICTIONARY as a DATASET (see DATASET from DICTIONARY below).

The ninth form is used to create a DATASET using a TRANSFORM function (see DATASET from TRANSFORM below).

### THOR/FLAT Files

```
attr := DATASET( file, struct, THOR [__COMPRESSED__][,OPT ] [,UNSORTED][,PRELOAD([nbr])] [,ENCRYPT(key) ]);  
attr := DATASET( file, struct, FLAT [__COMPRESSED__] [,OPT ] [,UNSORTED] [,PRELOAD([nbr])] [,ENCRYPT(key) ]);  
```

<table>
<thead>
<tr>
<th>THOR</th>
<th>Specifies the file is in the Data Refinery (may optionally be specified as FLAT, which is synonymous with THOR in this context).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COMPRESSED</strong></td>
<td>Optional. Specifies that the THOR file is compressed because it is a result of the PERSIST Workflow Service or was OUTPUT with the COMPRESSED option.</td>
</tr>
<tr>
<td><strong>GROUPED</strong></td>
<td>Specifies the DATASET has been grouped using the GROUP function.</td>
</tr>
<tr>
<td>OPT</td>
<td>Optional. Specifies that using dataset when the THOR file doesn't exist results in an empty recordset instead of an error condition.</td>
</tr>
<tr>
<td>UNSORTED</td>
<td>Optional. Specifies the THOR file is not sorted, as a hint to the optimizer.</td>
</tr>
<tr>
<td>PRELOAD</td>
<td>Optional. Specifies the file is left in memory after loading (valid only for Rapid Data Delivery Engine use).</td>
</tr>
<tr>
<td>nbr</td>
<td>Optional. An integer constant specifying how many indexes to create &quot;on the fly&quot; for speedier access to the dataset. If &gt; 1000, specifies the amount of memory set aside for these indexes.</td>
</tr>
<tr>
<td>ENCRYPT</td>
<td>Optional. Specifies the file was created by OUTPUT with the ENCRYPT option.</td>
</tr>
<tr>
<td>key</td>
<td>A string constant containing the encryption key used to create the file.</td>
</tr>
</tbody>
</table>

This form defines a THOR file that exists in the Data Refinery. This could contain either fixed-length or variable-length records, depending on the layout specified in the RECORD struct.
The struct may contain an UNSIGNED8 field with either \{VIRTUAL(fileposition)\} or \{VIRTUAL(localfileposition)\} appended to the field name. This indicates the field contains the record's position within the file (or part), and is used for those instances where a usable pointer to the record is needed, such as the BUILD function.

Example:

```ecl
PtblRec := RECORD
    STRING2 State := Person.per_st;
    STRING20 City := Person.per_full_city;
    STRING25 Lname := Person.per_last_name;
    STRING15 Fname := Person.per_first_name;
END;

Tbl := TABLE(Person,PtblRec);

PtblOut := OUTPUT(Tbl,"RTTEMP::TestFile");
   //write a THOR file

Ptbl := DATASET('~Thor400::RTTEMP::TestFile',
    {PtblRec, UNSIGNED8 __fpos {VIRTUAL(fileposition)}},
    THOR,OPT);
   // __fpos contains the "pointer" to each record
   // Thor400 is the scope name and RTTEMP is the
   // directory in which TestFile is located
   //using ENCRYPT

OUTPUT(Tbl,"~Thor400::RTTEMP::TestFileEncrypted",ENCRYPT('mykey'));

PtblE := DATASET('~LR::TestFileEncrypted',
    PtblRec,
    THOR,OPT,ENCRYPT('mykey'));
```

### CSV Files

attr := DATASET( file, struct, CSV [ [ [ HEADING( n ) ] ][ , SEPARATOR( f_delimiters ) ] ]
[ , TERMINATOR( r_delimiters ) ] [ [ , QUOTE( characters ) ] ][ , ESCAPE( esc ) ] [ , MAXLENGTH( size ) ]
[ ASCII | EBCDIC | UNICODE ] [ , NOTRIM ] ) [ , ENCRYPT( key ) ] [ , __COMPRESSED__ ]);

<table>
<thead>
<tr>
<th>CSV</th>
<th>Specifies the file is a &quot;comma separated values&quot; ASCII file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEADING(n)</td>
<td>Optional. The number of header records in the file. If omitted, the default is zero (0).</td>
</tr>
<tr>
<td>SEPARATOR</td>
<td>Optional. The field delimiter. If omitted, the default is a comma (',') or the delimiter specified in the spray operation that put the file on disk.</td>
</tr>
<tr>
<td>f_delimiters</td>
<td>A single string constant, or set of string constants, that define the character(s) used as the field delimiter. If Unicode constants are used, then the UTF8 representation of the character(s) will be used.</td>
</tr>
<tr>
<td>TERMINATOR</td>
<td>Optional. The record delimiter. If omitted, the default is a line feed ('\n') or the delimiter specified in the spray operation that put the file on disk.</td>
</tr>
<tr>
<td>r_delimiters</td>
<td>A single string constant, or set of string constants, that define the character(s) used as the record delimiter.</td>
</tr>
<tr>
<td>QUOTE</td>
<td>Optional. The string quote character used. If omitted, the default is a single quote (&quot;&quot;) or the delimiter specified in the spray operation that put the file on disk.</td>
</tr>
<tr>
<td>characters</td>
<td>A single string constant, or set of string constants, that define the character(s) used as the string value delimiter.</td>
</tr>
</tbody>
</table>
ESCAPE | Optional. The string escape character used to indicate the next character (usually a control character) is part of the data and not to be interpreted as a field or row delimiter. If omitted, the default is the escape character specified in the spray operation that put the file on disk (if any).

esc | A single string constant, or set of string constants, that define the character(s) used to escape control characters.

MAXLENGTH(size) | Optional. Maximum record length in the file in bytes. If omitted, the default is 4096. There is a hard limit of 10MB but that can be overridden using #OPTION(maxCSVRowSizeMb,nn) where nn is the maximum size in MB. The maximum record size should be set as conservatively as possible.

ASCII | Specifies all input is in ASCII format, including any EBCDIC or UNICODE fields.

EBCDIC | Specifies all input is in EBCDIC format except the SEPARATOR and TERMINATOR (which are expressed as ASCII values).

UNICODE | Specifies all input is in Unicode UTF8 format.

NOTRIM | Specifies preserving all whitespace in the input data (the default is to trim leading blanks).

ENCRIPT | Optional. Specifies the file was created by OUTPUT with the ENCRYPT option.

key | A string constant containing the encryption key used to create the file.

__COMPRESSED__ | Optional. Specifies that the file is compressed because it was OUTPUT with the COMPRESSED option.

This form is used to read an ASCII CSV file. This can also be used to read any variable-length record file that has a defined record delimiter. If none of the ASCII, EBCDIC, or UNICODE options are specified, the default input is in ASCII format with any UNICODE fields in UTF8 format.

Example:

```ecl
CSVRecord := RECORD
    UNSIGNED4 person_id;
    STRING20 per_surname;
    STRING20 per_forename;
END;

file1 := DATASET('MyFile.CSV',CSVrecord,CSV);            //all defaults
file2 := DATASET('MyFile.CSV',CSVrecord,CSV(HEADING(1))); //1 header
file3 := DATASET('MyFile.CSV',
    CSVrecord,
    CSV(HEADING(1),
    SEPARATOR(['','\t']),
    TERMINATOR(['\n','\r\n','\n\r']));
    //1 header record, either comma or tab field delimiters,
    // either LF or CR/LF or LF/CR record delimiters
```

**XML Files**

attr := DATASET( file, struct, XML( xpath [, NOROOT ] ) [,ENCRYPT(key) ]); 

XML | Specifies the file is an XML file.

xpath | A string constant containing the full XPATH to the tag that delimits the records in the file.

NOROOT | Specifies the file is an XML file with no file tags, only row tags.
This form is used to read an XML file into the Data Refinery. The xpath parameter defines the record delimiter tag using a subset of standard XPATH (www.w3.org/TR/xpath) syntax (see the XPATH Support section under the RECORD structure discussion for a description of the supported subset).

The key to getting individual field values from the XML lies in the RECORD structure field definitions. If the field name exactly matches a lower case XML tag containing the data, then nothing special is required. Otherwise, {xpath(xpathtag)} appended to the field name (where the xpathtag is a string constant containing standard XPATH syntax) is required to extract the data. An XPATH consisting of empty angle brackets (<> indicates the field receives the entire record. An absolute XPATH is used to access properties of parent elements. Because XML is case sensitive, and ECL identifiers are case insensitive, xpaths need to be specified if the tag contains any upper case characters.

NOTE: XML reading and parsing can consume a large amount of memory, depending on the usage. In particular, if the specified xpath matches a very large amount of data, then a large data structure will be provided to the transform. Therefore, the more you match, the more resources you consume per match. For example, if you have a very large document and you match an element near the root that virtually encompasses the whole thing, then the whole thing will be constructed as a referenceable structure that the ECL can get at.

Example:

```ecl
/* an XML file called "MyFile" contains this XML data: 
<library>
  <book isbn="123456789X">
    <author>Bayliss</author>
    <title>A Way Too Far</title>
  </book>
  <book isbn="1234567801">
    <author>Smith</author>
    <title>A Way Too Short</title>
  </book>
</library> */

rform := RECORD
  STRING author; //data from author tag -- tag name is lowercase and matches field name
  STRING name {XPATH('title')}; //data from title tag, renaming the field
  STRING isbn {XPATH('@isbn')}; //isbn definition data from book tag
END;
books := DATASET('MyFile',rform,XML('library/book'));
```

**JSON Files**

```
attr := DATASET( file, struct, JSON( xpath [, NOROOT ] [, ENCRYPT(key) ] ));
```
**key**

A string constant containing the encryption key used to create the file.

This form is used to read a JSON file. The *xpath* parameter defines the path used to locate records within the JSON content using a subset of standard XPATH ([www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)) syntax (see the *xpath* Support section under the RECORD structure discussion for a description of the supported subset).

The key to getting individual field values from the JSON lies in the RECORD structure field definitions. If the field name exactly matches a lower case JSON tag containing the data, then nothing special is required. Otherwise, `{xpath(xpathtag)}` appended to the field name (where the *xpathtag* is a string constant containing standard XPATH syntax) is required to extract the data. An XPATH consisting of empty quotes (""") indicates the field receives the entire record. An absolute XPATH is used to access properties of child elements. Because JSON is case sensitive, and ECL identifiers are case insensitive, xpaths need to be specified if the tag contains any upper case characters.

**NOTE:** JSON reading and parsing can consume a large amount of memory, depending on the usage. In particular, if the specified xpath matches a very large amount of data, then a large data structure will be provided to the transform. Therefore, the more you match, the more resources you consume per match. For example, if you have a very large document and you match an element near the root that virtually encompasses the whole thing, then the whole thing will be constructed as a referenceable structure that the ECL can get at.

**Example:**

```ecl
/* a JSON file called "mybooks.json" contains this data: 
[ 
  { 
    "id": "978-0641723445", 
    "name": "The Lightning Thief", 
    "author": "Rick Riordan"
  }, 
  { 
    "id": "978-1423103349", 
    "name": "The Sea of Monsters", 
    "author": "Rick Riordan"
  } 
] */

BookRec := RECORD
  STRING ID {XPATH('id')}; //data from id tag -- renames field to uppercase
  STRING title {XPATH('name')}; //data from name tag, renaming the field
  STRING author; //data from author tag -- tag name is lowercase and matches field name
END;

books := DATASET('-LR::mybooks.json',BookRec,JSON('/'));
OUTPUT(books);
```

### PIPE Files

**attr := DATASET( file, struct, PIPE( command [ , CSV | XML ] ));**

<table>
<thead>
<tr>
<th>PIPE</th>
<th>Specifies the <em>files</em> comes from the <em>command</em> program. This is a “read” pipe.</th>
</tr>
</thead>
<tbody>
<tr>
<td>command</td>
<td>The name of the program to execute, which must output records in the <em>struct</em> format to standard output.</td>
</tr>
<tr>
<td>CSV</td>
<td>Optional. Specifies the output data format is CSV. If omitted, the format is raw.</td>
</tr>
<tr>
<td>XML</td>
<td>Optional. Specifies the output data format is XML. If omitted, the format is raw.</td>
</tr>
</tbody>
</table>
This form uses `PIPE(command)` to send the file to the command program, which then returns the records to standard output in the `struct` format. This is also known as an input PIPE (analogous to the PIPE function and PIPE option on OUTPUT).

Example:

```ecl
PtblRec := RECORD
    STRING2 State;
    STRING20 City;
    STRING25 Lname;
    STRING15 Fname;
END;

Ptbl := DATASET('~LR::TestFile',
    PtblRec,
    PIPE('ProcessFile'));
// ProcessFile is the input pipe
```

**Named Output DATASETs**

```
attr := DATASET( WORKUNIT([wuid,] namedoutput).struct);
```

This form allows you to use as a DATASET the result of an OUTPUT with the NAMED option within the same workunit, or the workunit specified by the `wuid` (workunit ID). This is a feature most useful in the Rapid Data Delivery Engine.

Example:

```
//Named Output DATASET in the same workunit:
a := OUTPUT(Person(per_st='FL'), NAMED('FloridaFolk'));
x := DATASET(WORKUNIT('FloridaFolk'),
    RECORDOF(Person));
b := OUTPUT(x(per_first_name[1..4]= 'RICH'));

SEQUENTIAL(a,b);

//Named Output DATASET in separate workunits:
//First workunit (wuid=W20051202-155102) contains this code:
MyRec := [STRING1 Value1, STRING1 Value2, INTEGER1 Value3];
SomeFile := DATASET([{'C','G',1},{'C','C',2},{'A','X',3},
    {'B','G',4},{'A','B',5}], MyRec);
OUTPUT(SomeFile, NAMED('Fred'));

// Second workunit contains this code, producing the same result:
ds := DATASET(WORKUNIT('W20051202-155102','Fred'), MyRec);
OUTPUT(ds);
```

**In-line DATASETs**

```
[ attr := ] DATASET( recordset , recstruct );
```

This form allows you to in-line a set of data and have it treated as a file. This is useful in situations where file operations are needed on dynamically generated data (such as the runtime values of a set of pre-defined expressions). It is also useful to test any boundary conditions for definitions by creating a small well-defined set of records with constant values that specifically exercise those boundaries. This form may be used in an expression context.

Nested RECORD structures may be represented by nesting records within records. Nested child datasets may also be initialized inside TRANSFORM functions using inline datasets (see the Child DATASET discussion).
Example:

//Inline DATASET using definition values
myrec := {REAL diff, INTEGER1 reason};
rms5008 := 10.0;
rms5009 := 11.0;
rms5010 := 12.0;
htable := DATASET([{rms5008,72},{rms5009,7},{rms5010,65}], myrec);

//Inline DATASET with nested RECORD structures
nameRecord := {STRING20 lname,STRING10 fname,STRING1 initial := ''};
personRecord := RECORD
  nameRecord primary;
  nameRecord mother;
  nameRecord father;
END;
personDataset := DATASET([{{'James','Walters','C'},
                           {'Jessie','Blenger'},
                           {'Horatio','Walters'}},
                           {{'Anne','Winston'},
                           {'Sant','Aclause'},
                           {'Elfin','And'}}], personRecord);

// Inline DATASET containing a Child DATASET
childPersonRecord := {STRING fname, UNSIGNED1 age};
personRecord := RECORD
  STRING20 fname;
  STRING20 lname;
  UNSIGNED2 numChildren;
  DATASET(childPersonRecord) children;
END;
personDataset := DATASET([{'Kevin','Hall',2,[['Abby',2],['Nat',2]]},
                           {'Jon','Simms',3,[['Jen',18],['Ali',16],['Andy',13]]}],
                           personRecord);

// Inline DATASET derived from a dynamic SET function
SetIDs(STRING fname) := SET(People(firstname=fname),id);
ds := DATASET(SetIDs('RICHARD'),{People.id});

// Inline DATASET derived from a list of transforms
IDtype := UNSIGNED8;
FMtype := STRING15;
Ltype := STRING25;
resultRec := RECORD
  IDtype id;
  FMtype firstname;
  Ltype lastname;
  FMtype middlename;
END;
T1(IDtype idval,FMtype fname,Ltype lname ) :=
  TRANSFORM(resultRec,
    SELF.id := idval,
    SELF.firstname := fname,
    SELF.lastname := lname,
    SELF := []);
T2(IDtype idval,FMtype fname,FMtype mname, Ltype lname ) :=
  TRANSFORM(resultRec,
    SELF.id := idval,
    SELF.firstname := fname,
SELF.middlename := mname,
SELF.lastname := lname);

ds := DATASET([T1(123,'Fred','Jones'),
               T2(456,'John','Q','Public'),
               T1(789,'Susie','Smith')]);

// You can construct a DATASET from a SET.
SET OF STRING s := ['Jim','Bob','Richard','Tom'];
DATASET(s,{STRING txt});

**Single-row DATASET Expressions**

**DATASET( row )**

This form is only used in an expression context. It allows you to in-line a single record dataset.

**Example:**

//the following examples demonstrate 4 ways to do the same thing:

```ecl
personRecord := RECORD
  STRING20 surname;
  STRING10 forename;
  INTEGER2 age := 25;
END;

namesRecord := RECORD
  UNSIGNED     id;
  personRecord;
END;

namesTable := DATASET('~LR::TestRow',namesRecord,THOR);
//simple dataset file declaration form

addressRecord := RECORD
  UNSIGNED         id;
  DATASET(personRecord) people;   //child dataset form
  STRING40       street;
  STRING40       town;
  STRING2        st;
END;

personRecord tc0(namesRecord L) := TRANSFORM
  SELF := L;
END;

//** 1st way - using in-line dataset form in an expression  context
addressRecord t0(namesRecord L) := TRANSFORM
  SELF.people := PROJECT(DATASET([{L.id,L.surname,L.forename,L.age}],
                                   namesRecord),
                         tc0(LEFT));
  SELF.id := L.id;
  SELF := [];
END;

p0 := PROJECT(namesTable, t0(LEFT));
OUTPUT(p0);

//** 2nd way - using single-row dataset form
addressRecord t1(namesRecord L) := TRANSFORM
  SELF.people := PROJECT(DATASET(L), tc0(LEFT));
  SELF.id := L.id;
  SELF := [];
END;
```
Child DATASETs

**DATASET( childstruct [ COUNT( count ) | LENGTH( size ) ] [, CHOOSE( maxrecs ) ] )**

This form is used as a value type inside a RECORD structure to define child dataset records in a non-normalized flat file. The form without COUNT or LENGTH is the simplest to use, and just means that the dataset the length and data are stored within myfield. The COUNT form limits the number of elements to the count expression. The LENGTH form specifies the size in another field instead of the count. This can only be used for dataset input.

The following alternative syntaxes are also supported:

- **childstructfieldname [ SELF.count]**
- **DATASETnewname := fieldname**
- **DATASET fieldname (deprecated form -- will go away post-SR9)**

Any operation may be performed on child datasets in hthor and the Rapid Data Delivery Engine (Roxie), but only the following operations are supported in the Data Refinery (Thor):

1) PROJECT, CHOOSE, TABLE (non-grouped), and filters on child tables.

2) Aggregate operations are allowed on any of the above

3) Several aggregates can be calculated at once by using

   ```
   summary := TABLE(x.children,{ f1 := COUNT(GROUP),
                                  f2 := SUM(GROUP,x),
                                  f3 := MAX(GROUP,y)});
   summary.f1;
   ```

4) **DATASET[n]** is supported to index the child elements

5) **SORT(dataset, a, b)[1]** is also supported to retrieve the best match.

6) Concatenation of datasets is supported.
7) Temporary TABLEs can be used in conjunction.

8) Initialization of child datasets in temp TABLE definitions allows [] to be used to initialize 0 elements.

Note that,

```
TABLE(ds, { ds.id, ds.children(age != 10) });
```

is not supported, because a dataset in a record definition means "expand all the fields from the dataset in the output." However adding an identifier creates a form that is supported:

```
TABLE(ds, { ds.id, newChildren := ds.children(age != 10); });
```

**Example:**

```ecl
ParentRec := {INTEGER1 NameID, STRING20 Name};
ParentTable := DATASET([[1,'Kevin'],[2,'Liz'],
                      [3,'Mr Nobody'],[4,'Anywhere']], ParentRec);
ChildRec := {INTEGER1 NameID, STRING20 Addr};
ChildTable := DATASET([[1,'10 Malt Lane'],[2,'10 Malt Lane'],
                      [2,'3 The cottages'],[4,'Here'],[4,'There'],
                      [4,'Near'],[4,'Far']],ChildRec);

DenormedRec := RECORD
  INTEGER1 NameID;
  STRING20 Name;
  UNSIGNED1 NumRows;
  DATASET(ChildRec) Children;
// ChildRec Children; //alternative syntax
END;

DenormedRec ParentMove(ParentRec L) := TRANSFORM
  SELF.NumRows := 0;
  SELF.Children := [];
  SELF := L;
END;

ParentOnly := PROJECT(ParentTable, ParentMove(LEFT));

DenormedRec ChildMove(DenormedRec L,ChildRec R,INTEGER C):=TRANSFORM
  SELF.NumRows := C;
  SELF.Children := L.Children + R;
  SELF := L;
END;

DeNormedRecs := DENORMALIZE(ParentOnly, ChildTable,
  LEFT.NameID = RIGHT.NameID,
  ChildMove(LEFT,RIGHT,COUNTER));

OUTPUT(DeNormedRecs,,,RTTEMP::TestChildDatasets);
```

// Using inline DATASET in a TRANSFORM to initialize child records
```
AkaRec := {STRING20 forename,STRING20 surname};
outputRec := RECORD
  UNSIGNED id;
  DATASET(AkaRec) children;
END;

inputRec := RECORD
  UNSIGNED id;
  STRING20 forename;
  STRING20 surname;
END;

inPeople := DATASET([{
                      [1,'Kevin','Halliday'],[1,'Kevin','Hall'],[1,'Gawain',''],
                      [2,'Liz','Halliday'],[2,'Elizabeth','Halliday'],
                      [2,'Elizabeth','MaidenName'],[3,'Lorraine','Chapman']},
                      {1,'Kevin','Halliday'],[1,'Kevin','Hall'],[1,'Gawain','']},
                      [2,'Liz','Halliday'],[2,'Elizabeth','Halliday'],
                      [2,'Elizabeth','MaidenName'],[3,'Lorraine','Chapman']},
```
Record Structures and Files

```ecl
{4, 'Richard', 'Chapman'}, {4, 'John', 'Doe'}], inputRec);
outputRec makeFatRecord(inputRec l) := TRANSFORM
  SELF.id := l.id;
  SELF.children := DATASET([{ l.forename, l.surname }], AkaRec);
END;

fatIn := PROJECT(inPeople, makeFatRecord(LEFT));
outputRec makeChildren(outputRec l, outputRec r) := TRANSFORM
  SELF.id := l.id;
  SELF.children := l.children + ROW({ r.children[1].forename,
    r.children[1].surname },
    AkaRec);
END;

r := ROLLUP(fatIn, id, makeChildren(LEFT, RIGHT));
```

### DATASET as a Parameter Type

#### [GROUPED] [LINKCOUNTED] [STREAMED] DATASET( struct )

This form is only used as a Value Type for passing parameters, specifying function return types, or defining a SET OF datasets. If GROUPED is present, the passed parameter must have been grouped using the GROUP function. The LINKCOUNTED and STREAMED keywords are primarily for use in BEGINC++ functions or external C++ library functions.

**Example:**

```ecl
MyRec := {STRING1 Letter};
SomeFile := DATASET([{'A'}, {'B'}, {'C'}, {'D'}, {'E'}], MyRec);

//Passing a DATASET parameter
FilteredDS(DATASET(MyRec) ds) := ds(Letter NOT IN ['A', 'C', 'E']);
//passed dataset referenced as "ds" in expression
OUTPUT(FilteredDS(SomeFile));

//*****************************************************************
// The following example demonstrates using DATASET as both a
// parameter type and a return type
rec_Person := RECORD
  STRING20 FirstName;
  STRING20 LastName;
END;
rec_Person_exp := RECORD(rec_Person)
  STRING20 NameOption;
END;
rec_Person_exp xfm_DisplayNames(rec_Person l, INTEGER w) :=
  TRANSFORM
    SELF.NameOption :=
      CHOOSE(w,
        TRIM(l.FirstName) + ' ' + l.LastName,
        TRIM(l.LastName) + ', ' + l.FirstName,
        l.FirstName[1] + l.LastName[1],
        l.LastName);
    SELF := l;
END;

DATASET(rec_Person_exp) prototype(DATASET(rec_Person) ds) :=
  DATASET([ ], rec_Person_exp);
DATASET(rec_Person_exp) DisplayFullName(DATASET(rec_Person) ds) :=
```
PROJECT(ds, xfm_DisplayNames(LEFT,1));

DATASET(rec_Person_exp) DisplayRevName(DATASET(rec_Person) ds) :=
  PROJECT(ds, xfm_DisplayNames(LEFT,2));

DATASET(rec_Person_exp) DisplayFirstName(DATASET(rec_Person) ds) :=
  PROJECT(ds, xfm_DisplayNames(LEFT,3));

DATASET(rec_Person_exp) DisplayLastName(DATASET(rec_Person) ds) :=
  PROJECT(ds, xfm_DisplayNames(LEFT,4));

DATASET(rec_Person_exp) PlayWithName(DATASET(rec_Person) ds_in, 
  prototype PassedFunc, 
  STRING1 SortOrder='A', 
  UNSIGNED1 FieldToSort=1, 
  UNSIGNED1 PrePostFlag=1) := FUNCTION

  FieldPre := CHOOSE(FieldToSort,ds_in.FirstName,ds_in.LastName);
  SortedDSPre(DATASET(rec_Person) ds) :=
    IF(SortOrder='A', 
      SORT(ds,FieldPre),
      SORT(ds,-FieldPre));

  InDS := IF(PrePostFlag=1,SortedDSPre(ds_in),ds_in);

  PDS := PassedFunc(InDS); //call the passed function parameter

  FieldPost := CHOOSE(FieldToSort, 
    PDS.FirstName, 
    PDS.LastName, 
    PDS.NameOption);

  SortedDSPost(DATASET(rec_Person_exp) ds) :=
    IF(SortOrder = 'A', 
      SORT(ds,FieldPost),
      SORT(ds,-FieldPost));

  OutDS := IF(PrePostFlag=1,PDS,SortedDSPost(PDS));
  RETURN OutDS;
END;

//define inline datasets to use.
ds_names1 := DATASET([{'John','Smith'},{'Henry','Jackson'},
  {'Harry','Potter'}], rec_Person);
ds_names2 := DATASET([ {'George','Foreman'},
  {'Sugar Ray','Robinson'},
  {'Joe','Louis'}], rec_Person);

//get name you want by passing the appropriate function parameter:
s_Name1 := PlayWithName(ds_names1, DisplayFullName, 'A',1,1);
s_Name2 := PlayWithName(ds_names2, DisplayRevName, 'D',3,2);
a_Name := PlayWithName(ds_names1, DisplayFirstName, 'A',1,1);
b_Name := PlayWithName(ds_names2, DisplayLastName, 'D',1,1);
OUTPUT(s_Name1);
OUTPUT(s_Name2);
OUTPUT(a_Name);
OUTPUT(b_Name);

DATASET from DICTIONARY

DATASET( dict )

This form re-defines the dict as a DATASET.

Example:
Record Structures and Files

```
rec := {STRING color, UNSIGNED1 code, STRING name};
ColorCodes := DATASET([{'Black', 0 , 'Fred'},
                      {'Brown', 1 , 'Sam'},
                      {'Red' , 2 , 'Sue'},
                      {'White', 3 , 'Jo'}], rec);
ColorCodesDCT := DICTIONARY(ColorCodes,{Color,Code});
ds := DATASET(ColorCodesDCT);
OUTPUT(ds);
```

See Also: OUTPUT, RECORD Structure, TABLE, ROW, RECORDOF, TRANSFORM Structure, DICTIONARY

**DATASET from TRANSFORM**

```
DATASET( count, transform [, DISTRIBUTED | LOCAL ] )
```

This form uses the `transform` to create the records. The result type of the `transform` function determines the structure. The integer COUNTER can be used to number each iteration of the `transform` function.

**LOCAL** executes separately and independently on each node.

**Example:**

```
IMPORT STD;
msg(UNSIGNED c) := 'Rec ' + (STRING)c + ' on node ' + (STRING)(STD.system.Thorlib.Node()+1);

// DISTRIBUTED example
DS := DATASET(CLUSTERSIZE * 2,
               TRANSFORM({STRING line},
                          SELF.line := msg(COUNTER),
                          DISTRIBUTED));
OUTPUT(DS);
/* creates a result like this:
   Rec 1 on node 1
   Rec 2 on node 1
   Rec 3 on node 2
   Rec 4 on node 2
   Rec 5 on node 3
   Rec 6 on node 3
*/

// LOCAL example
DS2 := DATASET(2,
               TRANSFORM({STRING line},
                          SELF.line := msg(COUNTER),
                          LOCAL));
OUTPUT(DS2);
/* An alternative (and clearer) way
creates a result like this:
   Rec 1 on node 1
   Rec 2 on node 1
   Rec 1 on node 2
   Rec 1 on node 2
   Rec 2 on node 3
   Rec 2 on node 3
*/
```

See Also: RECORD Structure, TRANSFORM Structure
DICTIONARY

attr := DICTIONARY( dataset, structure );

DICTIONARY( structure )

<table>
<thead>
<tr>
<th>attr</th>
<th>The name of the DICTIONARY for later use in other definitions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataset</td>
<td>The name of a DATASET or recordset from which to derive the DICTIONARY. This may be defined inline (similar to an inline DATASET).</td>
</tr>
<tr>
<td>structure</td>
<td>The RECORD structure (often defined inline) specifying the layout of the fields. The first field(s) are key fields, optionally followed the &quot;results in&quot; operator (=&gt;) and additional payload fields. This is similar to the payload version of an INDEX. The payload may specify individual fields or may use the name of the dataset to payload all the non-key fields.</td>
</tr>
</tbody>
</table>

A DICTIONARY allows you to efficiently check whether a particular data value is in a list (using the IN operator), or to simply map data. It is similar to a LOOKUP JOIN that can be used in any context.

DICTIONARY Definition

The DICTIONARY declaration defines a set of unique records derived from the dataset parameter and indexed by the first field(s) named in the structure parameter. The DICTIONARY will contain one record for each unique value(s) in the key field(s). You can access an individual record by appending square brackets ([ ] ) to the attr name of the DICTIONARY, which contain the key field value(s) that identify the specific record to access.

DICTIONARY as a Value Type

The second form of DICTIONARY is a value type with the structure parameter specifying the RECORD structure of the data. This data type usage allows you to specify a DICTIONARY as a child dataset, similar to the way DATASET may be used to define a child dataset. This may also be used to pass a DICTIONARY as a parameter.

Example:

```
ColorCodes := DATASET([[{'Black' ,0 },
{'Brown' ,1 },
{'Red' ,2 },
{'Orange' ,3 },
{'Yellow' ,4 },
{'Green' ,5 },
{'Blue' ,6 },
{'Violet' ,7 },
{'Grey' ,8 },
{'White' ,9 }], {STRING color,UNSIGNED1 code});
ColorCodesDCT := DICTIONARY(ColorCodes,{Color,Code});    //multi-field key
ColorCodeDCT := DICTIONARY(ColorCodes,{Color => Code}); //payload field
CodeColorDCT := DICTIONARY(ColorCodes,{Code => Color});

//mapping examples
MapCode2Color(UNSIGNED1 code) := CodeColorDCT[code].color;
MapColor2Code(STRING color)   := ColorCodeDCT[color].code;
OUTPUT(MapColor2Code('Red'));  //2
```
OUTPUT(MapCode2Color(4));   //'Yellow'

//Search term examples
OUTPUT('Green' IN ColorCodeDCT);  //true
OUTPUT(6 IN CodeColorDCT);       //true
OUTPUT(ROW(\{'Red',2\},RECORDOF(ColorCodes)) IN ColorCodesDCT); //multi-field key, true

//multi-field payload examples
rec := RECORD
  STRING10 color;
  UNSIGNED1 code;
  STRING10 name;
END;
Ds := DATASET([\{'Black' ,0 , 'Fred'\},
               \{'Brown' ,1 , 'Seth'\},
               \{'Red'   ,2 , 'Sue'\},
               \{'White' ,3 , 'Jo'\}], rec);
DsDCT := DICTIONARY(DS,{color => DS});
OUTPUT('Red' IN DsDCT); //true
DsDCT['Red'].code;      //2
DsDCT['Red'].name;      //Sue

//inline DCT examples
InlineDCT := DICTIONARY([\{'Black' => 0 , 'Fred'\},
                          \{'Brown' => 1 , 'Sam'\},
                          \{'Red'   => 2 , 'Sue'\},
                          \{'White' => 3 , 'Jo'\}],
                          {STRING10 color => UNSIGNED1 code,STRING10 name});
OUTPUT('Red' IN InlineDCT); //true
InlineDCT['Red'].code;    //2
InlineDCT['Red'].name;    //Sue
InlineDCT['Red'];         //Red   2   Sue

//Form 2 examples -- parameter passing
MyDCTfunc(DICTIONARY({STRING10 color => UNSIGNED1 code,STRING10 name}) DCT,
            STRING10 key) := DCT[key].name;
MyDCTfunc(InlineDCT,'White'); //Jo
MyDCTfunc(DsDCT,'Brown');     //Seth

See Also: DATASET, RECORD Structure, INDEX, IN Operator
INDEX

attr := INDEX([ baserecset, ] keys, indexfile [ ,SORTED ] [ ,OPT ] [ ,COMPRESSED( LZW | ROW | FIRST ) ] [ ,DISTRIBUTED ] [ ,FILEPOSITION( [ flag ] ) ] [ , MAXLENGTH([ ""value"" ] ) ] );

attr := INDEX([ baserecset, ] keys, payload, indexfile [ ,SORTED ] [ ,OPT ] [ ,COMPRESSED(option) ] [ ,DISTRIBUTED ] [ ,FILEPOSITION( [ flag ] ) ] [ , MAXLENGTH([ value ] ) ] );

attr := INDEX(index,newindexfile [ , MAXLENGTH([value]) ] );

<table>
<thead>
<tr>
<th>attr</th>
<th>The name of the INDEX for later use in other attributes.</th>
</tr>
</thead>
<tbody>
<tr>
<td>baserecset</td>
<td>Optional. The set of data records for which the index file has been created. If omitted, all fields in the keys and payload parameters must be fully qualified.</td>
</tr>
<tr>
<td>keys</td>
<td>The RECORD structure of key fields that reference into the baserecset (the &quot;search terms&quot; for the INDEX). Key fields may be baserecset fields or computed fields. REAL and DECIMAL types are not supported as &quot;search term&quot; fields. If omitted, all fields in the baserecset are used. This RECORD structure is typically defined inline within the INDEX using curly braces ({}), but may also be a separately defined RECORD structure. If the RECORD structure is separately defined it must meet the same requirements as used by the TABLE() function (the RECORD structure must define the type, name, and source of the data for each field), otherwise the BUILD action will not syntax check.</td>
</tr>
<tr>
<td>payload</td>
<td>The RECORD structure of the indexfile that contains additional fields not used as &quot;search term&quot; keys. This may contain fields from the baserecset and/or computed fields. If the name of the baserecset is in this structure, it specifies &quot;all other fields not already named in the keys parameter&quot; are added. The payload fields do not take up space in the non-leaf nodes of the index and cannot be referenced in a KEYED() filter clause. Any field with the (BLOB) modifier (to allow more than 32K of data per index entry) is stored within the indexfile, but not with the rest of the record; accessing the BLOB data requires an additional seek. This RECORD structure is typically defined inline within the INDEX using curly braces ({}), but may also be a separately defined RECORD structure. If the RECORD structure is separately defined it must meet the same requirements as used by the TABLE() function (the RECORD structure must define the type, name, and source of the data for each field), otherwise the BUILD action will not syntax check.</td>
</tr>
<tr>
<td>indexfile</td>
<td>A string constant containing the logical filename of the index. See the Scope &amp; Logical Filenames section for more on logical filenames.</td>
</tr>
<tr>
<td>SORTED</td>
<td>Optional. Specifies that when the index is accessed the records come out in the order of the keys. If omitted, the returned record order is undefined.</td>
</tr>
<tr>
<td>OPT</td>
<td>Optional. Specifies that using the index when the indexfile doesn't exist results in an empty recordset instead of an error condition.</td>
</tr>
<tr>
<td>COMPRESSED</td>
<td>Optional. Specifies the index should be compressed using the type of compression specified. If omitted, the default is LZW, a variant of the Lempel-Ziv-Welch algorithm.</td>
</tr>
<tr>
<td>option</td>
<td>See Indexes and Compression for options.</td>
</tr>
</tbody>
</table>
| DISTRIBUTED | Optional. Specifies that the index was created with the DISTRIBUTED option on the BUILD action or the BUILD action simply referenced the INDEX declaration with the DISTRIBUTED option. The INDEX is therefore accessed locally on each node (similar to the LOCAL function, which is preferred), is not glob-
ally sorted, and there is no root index to indicate which part of the index will contain a particular entry. This may be useful in Roxie queries in conjunction with ALLNODES use.

FILEPOSITION

Optional. If flag is FALSE, prevents the normal behavior of implicit fileposition field being created and will not treat a trailing integer field any differently from the rest of the payload.

flag

Optional. TRUE or FALSE, indicating whether or not to create the implicit fileposition field.

index

The name of a previously defined INDEX attribute to duplicate.

newindexfile

A string constant containing the logical filename of the new index. See the Scope & Logical Filenames section for more on logical filenames.

MAXLENGTH

Optional. This option is used to create indexes that are backward compatible for platform versions prior to 3.0. Specifies the maximum length of a variable-length index record. Fixed length records always use the minimum size required. If the default maximum length causes inefficiency problems, it can be explicitly overridden.

value

Optional. An integer value indicating the maximum length. If omitted, the maximum size is calculated from the record structure. Variable-length records that do not specify MAXLENGTH may be slightly inefficient.

INDEX declares a previously created index for use. INDEX is related to BUILD (or BUILDINDEX) in the same manner that DATASET is to OUTPUT--BUILD creates an index file that INDEX then defines for use in ECL code. Index files are compressed. A single index record must be defined as less than 32K and result in a less than 8K page after compression.

The Binary-tree metakey portion of the INDEX is a separate 32K file part on the first node of the Thor cluster on which it was built, but deployed to every node of a Roxie cluster. There are as many leaf-node file parts as there are nodes to the Thor cluster on which it was built. The specific distribution of the leaf-node records across execution nodes is undefined in general, as it depends on the size of the cluster on which it was built and the size of the cluster on which it is used.

These data types are supported in the keyed portion of an INDEX:

- BOOLEAN
- INTEGER
- UNSIGNED
- STRING
- DATA
- QSTRING

All STRINGs must be fixed length.

Keyed Access INDEX

This form defines an index file to allow keyed access to the baserecset. The index is used primarily by the FETCH and JOIN (with the KEYED option) operations.

Example:
Payload INDEX

This form defines an index file containing extra payload fields in addition to the keys. The payload may contain fields with the \{BLOB\} modifier to allow more than 32K of data per index entry. These BLOB fields are stored within the indexfile, but not with the rest of the record; accessing the BLOB data requires an additional seek.

This form is used primarily by “half-key” JOIN operations to eliminate the need to directly access the baserect-set, thus increasing performance over the “full-keyed” version of the same operation (done with the KEYED option on the JOIN). By default, payload fields are not sorted during the BUILD action to minimize space on the leaf nodes of the key. This sorting behavior can be controlled by using sortIndexPayload in a #OPTION statement.

You can also use the "results in" operator (=>) before payload fields as shown below.

Example:

VehicleKey3 := INDEX(Vehicles,{st,city => lname},'vkey::st.city3');
BUILD(VehicleKey3);

Duplicate INDEX

This form defines a newindexfile that is identical to the previously defined index.

Example:
NewVehicleKey := INDEX(VehicleKey, 'NEW::vkey::st.city');
//define NewVehicleKey like VehicleKey
Indexes and Compression

Compression Options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LZW</td>
<td>The default compression. It is a variant of the Lempel-Ziv-Welch algorithm. It remains the default for backward compatibility.</td>
</tr>
<tr>
<td>ROW</td>
<td>Compresses index entries based on differences between rows (for use with fixed-length records only). It typically does not compress as well as LZW, but takes up less space in memory because the rows are expanded on demand.</td>
</tr>
<tr>
<td>'inplace'</td>
<td>Causes the index to be built using the inplace compression format. The payload defaults to using lz4 compression.</td>
</tr>
<tr>
<td>'inplace: lz4hc'</td>
<td>Causes inplace compression on the key fields and lz4hc compression on the payload. The resulting index can be smaller than using lz4.</td>
</tr>
</tbody>
</table>

The inplace index compression format (introduced in version 9.2.0) improves compression of keyed fields and allows them to be searched without decompression. The original index compression implementation decompresses the rows when they are read from disk.

Because the branch nodes can be searched without decompression more branch nodes fit into memory which can improve search performance. The lz4 compression used for the payload is significantly faster at decompressing leaf pages than the previous LZW compression.

Whether performance is better with lz4hc (a high-compression variant of lz4) on the payload fields depends on the access characteristics of the data and how much of the index is cached in memory.

If you attempt to read an index with the inplace compression format on a system that does not support them, you will receive an error message.

See Also: DATASET, BUILDINDEX, JOIN, FETCH, KEYED/WILD

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
Scope and Logical Filenames

File Scope

The logical filenames used in DATASET and INDEX attribute definitions and the OUTPUT and BUILD (or BUILDINDEX) actions can optionally begin with a ~ meaning it is absolute, otherwise it is relative (the platform configured scope prefix is prepended). It may contain scopes delimited by double colons (::) with the final portion being the filename. It cannot have a trailing double colons (::). A cluster qualifier can be specified. For example, ~myfile@mythor2 points to one file where the file is on multiple clusters in the same scope. Valid characters of a scope or filename are ASCII >32 < 127 except * / : < > ? and |.

To reference uppercase characters in physical file paths and filenames, use the caret character (^). For example, '~file::10.150.254.6::var::lib::^h^p^c^c^systems::mydropzone::^people.txt'.

The presence of a scope in the filename allows you to override the default scope name for the cluster. For example, assuming you are operating on a cluster whose default scope name is "Training" then the following two OUTPUT actions result in the same scope:

```
OUTPUT(SomeFile,'SomeDir::SomeFileOut1');
OUTPUT(SomeFile,'~Training::SomeDir::SomeFileOut2');
```

The presence of the leading tilde in the filename only defines the scope name and does not change the set of disks to which the data is written (files are always written to the disks of the cluster on which the code executes). The DATASET declarations for these files might look like this:

```
RecStruct := {STRING line};
d1 := DATASET('SomeDir::SomeFileOut1',RecStruct,THOR);
d2 := DATASET('~Training::SomeDir::SomeFileOut2',RecStruct,THOR);
```

These two files are in the same scope, so that when you use the DATASETs in a workunit the Distributed File Utility (DFU) will look for both files in the Training scope.

However, once you know the scope name you can reference files from any other cluster within the same environment. For example, assuming you are operating on a cluster whose default scope name is "Production" and you want to use the data in the above two files. Then the following two DATASET definitions allow you to access that data:

```
FileX := DATASET('~Training::SomeDir::SomeFileOut1',RecStruct,THOR);
FileY := DATASET('~Training::SomeDir::SomeFileOut2',RecStruct,THOR);
```

Notice the presence of the scope name in both of these definitions. This is required because the files are in another scope.

You should be frugal with file scope usage. The depth of file scopes can have a performance cost in systems with File Scope Security enabled. This cost is higher still when File Scope Scans are enabled because the system must make an external LDAP call to check every level in the scope, from the top to the bottom.

Foreign Files

Similar to the scoping rules described above, you can also reference files in separate environments serviced by a different Dali. This allows a read-only reference to remote files (both logical files and superfies).

**NOTE:** If LDAP authentication is enabled on the foreign Dali, the user's credentials are verified before processing the file access request. If LDAP file scope security is enabled on the foreign Dali, the user's file access permissions are also verified.
The syntax looks like this:

'~foreign::<ip>::<scope>::<tail>'

For example,

MyFile :=DATASET('~foreign::10.150.50.11::training::thor::myfile', RecStruct, FLAT);

gives read-only access to the remote training::thor::myfile file in the 10.150.50.11 environment.

**Landing Zone Files**

You can also directly read and write files on a landing zone (or any other IP-addressable box) that have not been sprayed to Thor. The landing zone must be running the dafileserv utility program. If the box is a Windows box, dafileserv must be installed as a service.

The syntax looks like this:

'~file::<LZ-ip>::<path>::<filename>'

For example,

MyFile :=DATASET('~file::10.150.50.12::c$::training::import::myfile', RecStruct, FLAT);

gives access to the remote c$/training/import/myfile file on the linux-based 10.150.50.12 landing zone.

ECL logical filenames are case insensitive and physical names default to lower case, which can cause problems when the landing zone is a Linux box (Linux is case sensitive). The case of characters can be explicitly uppercased by escaping them with a leading caret (^), as in this example:

MyFile :=DATASET('~file::10.150.50.12::c$::^Advanced^E^C^L::myfile', RecStruct, FLAT);

gives access to the remote c$/AdvancedECL/myfile file on the linux-based 10.150.50.12 landing zone.

**Dynamic Files**

In Roxie queries (only) you can also read files that may not exist at query deployment time, but that will exist at query runtime by making the filename DYNAMIC.

The syntax looks like this:

DYNAMIC('<filename>'

For example,

MyFile :=DATASET(DYNAMIC('~training::import::myfile'), RecStruct, FLAT);

This causes the file to be resolved when the query is executed instead of when it is deployed.

**Temporary SuperFiles**

A SuperFile is a collection of logical files treated as a single entity (see the SuperFile Overview article in the Programmer's Guide). You can specify a temporary SuperFile by naming the set of sub-files within curly braces in the string that names the logical file for the DATASET declaration. The syntax looks like this:

DATASET( '{ listoffiles }', recstruct, THOR);
`listoffiles` A comma-delimited list of the set of logical files to treat as a single SuperFile. The logical filenames must follow the rules listed above for logical filenames with the one exception that the tilde indicating scope name override may be specified either on each appropriate file in the list, or outside the curly braces.

For example, assuming the default scope name is "thor," the following examples both define the same SuperFile:

```ecl
MyFile := DATASET('in::file1,
in::file2,
~train::in::file3'),
RecStruct, THOR);

MyFile := DATASET('~{thor::in::file1,
thor::in::file2,
train::in::file3}),
RecStruct, THOR);
```

You cannot use this form of logical filename to do an OUTPUT or PERSIST; this form is read-only.
Implicit Dataset Relationality

Nested child datasets in a Data Refinery (Thor) or Rapid Data Delivery Engine (Roxie) cluster are inherently relational, since all the parent-child data is contained within a single physical record. The following rules apply to all inherent relationships.

The scope level of a particular query is defined by the primary dataset for the query. During the query, the assumption is that you are working with a single record from that primary dataset.

Assuming that you have the following relational structure in your database:

<table>
<thead>
<tr>
<th>Household</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person</td>
<td>Child of Household</td>
</tr>
<tr>
<td>Accounts</td>
<td>Child of Person, Grandchild of Household</td>
</tr>
</tbody>
</table>

This means that, at the primary scope level:

a) All fields from any file that has a 1:M relationship with the primary file are available. That is, all fields in any parent (or grandparent, etc.) record are available to the child. For example, if the Person dataset is the primary scope, then all the fields in the Household dataset are available.

b) All child datasets (or grandchildren, etc.) can be used in sub-queries to filter the parent, as long as the sub-query uses an aggregate function or operates at the level of the existence of a set of child records that meet the filter criteria (see EXISTS). You can use specific fields from within a child record at the scope level of the parent record by the use of EVALUATE or subscripting ([[]]) to a specific child record. For example, if the Person dataset is the primary scope, then you may filter the set of related Accounts records and check to see if you've filtered out all the related Accounts records.

c) If a dataset is used in a scope where it is not a child of the primary dataset, it is evaluated in the enclosing scope. For example, the expression:

```
Household(Person(personage > AVE(Person,personage))
```

means "households containing people whose age is above the average age for the household." It does not mean "households containing people whose age is above the average for all the households." This is because the primary dataset (Household) encloses the child dataset (Person), making the evaluation of the AVE function operate at the level of the persons within the household.

d) An attribute defined with the STORED() workflow service is evaluated at the global level. It is an error if it cannot be evaluated independently of other datasets. This can lead to some slightly strange behaviour:

```
AveAge := AVE(Person,personage);
MyHouses := Household(Person(personage > aveAge));
```

means "households containing people whose age is above the average age for the household." However,

```
AveAge := AVE(Person,personage) : STORED('AveAge');
MyHouses := Household(Person(personage > aveAge));
```

Means "households containing people whose age is above the average for all the households." This is because the AveAge attribute is now evaluated outside the enclosing Household scope.
### Alien Data Types

#### TYPE Structure

```plaintext
TypeName := TYPE

functions;

END;
```

<table>
<thead>
<tr>
<th>TypeName</th>
<th>The name of the TYPE structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>functions</td>
<td>Function Attribute definitions. There are usually multiple functions.</td>
</tr>
</tbody>
</table>

The **TYPE** structure defines a series of **functions** that are implicitly invoked when the **TypeName** is subsequently used in a RECORD structure as a value type. Parameters may be passed to the TYPE structure Attribute which may then be used in any of the **function** definitions. To pass the parameters, simply append them to the **TypeName** used in the RECORD structure to define the value type for the field.

Alien data types (TYPE) should only be used when accessing external data files. It is much more efficient to use the native types for general processing. In particular, some optimizations to project and filter files remotely are not supported on alien datatypes.

A **TYPE** structure may only contain function definitions from the the list of available Special Functions (see **TYPE Structure Special Functions**).

**Example:**

```plaintext
EXPORT ReverseString4 := TYPE
  EXPORT STRING LOAD(STRING4 S) := Rev(S);
  EXPORT STRING4 STORE(STRING4 S) := Rev(S);
END;
需C(INTEGER len) := TYPE
  EXPORT STRING LOAD(STRING S) := 'C' + S[1..len];
  EXPORT STRING STORE(STRING S) := S[2..len+1];
  EXPORT INTEGER PHYSICALLENGTH(STRING S) := len;
END;
ScaleInt := TYPE
  EXPORT REAL LOAD(INTEGER4 I ) := I / 100;
  EXPORT INTEGER4 STORE(REAL R) := ROUND(R * 100);
END;
R := RECORD
  ReverseString4 F1;
  // Defines a field size of 4 bytes. When R.F1 is used,
  // the ReverseString4.Load function is called passing
  // in those four bytes and returning a string result.
  NeedC(5) F2;
  // Defines a field size of 5 bytes. When R.F2 is used,
  // those 5 bytes are passed in to NeedC.Load (along with
  // the length 5) and a 6 byte string is returned.
  ScaleInt F3;
  // Defines a field size of 4. When R.F3 is used, the
  // ScaleInt.Load function returns the number / 100.
END;
```

See Also: RECORD Structure, TYPE Structure Special Functions

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
**TYPE Structure Special Functions**

**LOAD**

`EXPORT LogicalType LOAD(PhysicalType alias) := expression;`

<table>
<thead>
<tr>
<th>LogicalType</th>
<th>The value type of the resulting output of the function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PhysicalType</td>
<td>The value type of the input parameter to the function.</td>
</tr>
<tr>
<td>alias</td>
<td>The name of the input to use in the <code>expression</code>.</td>
</tr>
<tr>
<td>expression</td>
<td>The operation to perform on the input.</td>
</tr>
</tbody>
</table>

`LOAD` defines the callback function to be applied to the bytes of the record to create the data value to be used in the computation. This function defines how the system reads the data from disk.

**STORE**

`EXPORT PhysicalType STORE(LogicalType alias) := expression;`

<table>
<thead>
<tr>
<th>PhysicalType</th>
<th>The value type of the resulting output of the function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogicalType</td>
<td>The value type of the input parameter to the function.</td>
</tr>
<tr>
<td>alias</td>
<td>The name of the input to use in the <code>expression</code>.</td>
</tr>
<tr>
<td>expression</td>
<td>The operation to perform on the input.</td>
</tr>
</tbody>
</table>

`STORE` defines the callback function to be applied to the computed value to store it within the record. This function defines how the system writes the data to disk.

**PHYSICALLENGTH**

`EXPORT INTEGER PHYSICALLENGTH(type alias) := expression;`

<table>
<thead>
<tr>
<th>type</th>
<th>The value type of the input parameter to the function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>The name of the input to use in the <code>expression</code>.</td>
</tr>
<tr>
<td>expression</td>
<td>The operation to perform on the input.</td>
</tr>
</tbody>
</table>

`PHYSICALLENGTH` defines the callback function to determine the storage requirements of the logical format in the specified physical format. This function defines how many bytes the data occupies on disk.

**MAXLENGTH**

`EXPORT INTEGER MAXLENGTH := expression;`

| expression        | An integer constant defining the maximum physical length of the data. |

`MAXLENGTH` defines the callback function to determine the maximum physical length of variable-length data.

**GETISVALID**

`EXPORT BOOLEAN GETISVALID(PhysicalType alias) := expression;`
**PhysicalType**  The value type of the input parameter to the function.

**alias**  The name of the input to use in the *expression*.

**expression**  The operation to perform on the input.

**GETISVALID** defines the callback function to determine that data values are in the specified physical format.

Example:

```ecl
EXPORT NeedC(INTEGER len) := TYPE
    EXPORT STRING LOAD(STRING S) := 'C' + S[1..len];
    EXPORT STRING STORE(STRING S) := S[2..len+1];
    EXPORT INTEGER PHYSICALLENGTH(STRING S) := len;
    EXPORT INTEGER MAXLENGTH(STRING S) := len;
    EXPORT BOOLEAN GETISVALID(STRING S) := S[1] <> 'C';
END;

// delimited string data type
EXPORT dstring(STRING del) := TYPE
    EXPORT INTEGER PHYSICALLENGTH(STRING s) :=
        Std.Str.Find(s,del)+length(del)-1;
    EXPORT STRING LOAD(STRING s) :=
        s[1..Std.Str.Find(s,del)-1];
    EXPORT STRING STORE(STRING s) := s + del;
END;
```

See Also: TYPE Structure
Parsing Support

Natural Language Parsing is accomplished in ECL by combining pattern definitions with an output RECORD structure (or TRANSFORM function) specifically designed to receive the parsed values, then using the PARSE function to perform the operation.

Pattern definitions are used to detect "interesting" text within the data. Just as with all other attribute definitions, these patterns typically define specific parsing elements and may be combined to form more complex patterns, tokens, and rules.

The output RECORD structure (or TRANSFORM function) defines the format of the resulting recordset. It typically contains specific pattern matching functions that return the "interesting" text, its length or position.

The PARSE function implements the parsing operation. It returns a recordset that may then be post-processed as needed using standard ECL syntax, or simply output.
PARSE Pattern Value Types

There are three value types specifically designed and required to define parsing pattern attributes:

**PATTERN**

```
patternid := parsepattern;
```

- **patternid**: The attribute name of the pattern.
- **parsepattern**: The pattern, very similar to regular expressions. This may contain other previously defined PATTERN attributes. See ParsePattern Definitions below.

The **PATTERN** value type defines a parsing expression very similar to regular expression patterns.

**TOKEN**

```
tokenid := parsepattern;
```

- **tokenid**: The attribute name of the token.
- **parsepattern**: The token pattern, very similar to regular expressions. This may contain PATTERN attributes but no TOKEN or RULE attributes. See ParsePattern Definitions below.

The **TOKEN** value type defines a parsing expression very similar to a PATTERN, but once matched, the parser doesn't backtrack to find alternative matches as it would with PATTERN.

**RULE**

```
[ ( recstruct ) ] ruleid := rulePattern;
```

- **recstruct**: Optional. The attribute name of a RECORD structure attribute (valid only when the PARSE option is used on the PARSE function).
- **ruleid**: The attribute name of the rule.
- **rulePattern**: The rule pattern, very similar to regular expressions. This may contain PATTERN attributes, TOKEN attributes, or RULE attributes. See ParsePattern Definitions below.

The **RULE** value type defines a parsing expression containing combinations of TOKENs. If a RULE definition contains a PATTERN it is implicitly converted to a TOKEN. Like PATTERN, once matched, the parser backtracks to find alternative RULE matches.

If the PARSE option is present on the PARSE function (thereby implementing tomita parsing for the operation), each alternative RULE **rulePattern** may have an associated TRANSFORM function. The different input patterns can be referred to using $1, $2 etc. If the pattern has an associated recstruct then $1 is a row, otherwise it is a string. Default TRANSFORM functions are created in two circumstances:

1. If there are no patterns, the default transform clears the row. For example:
   ```
   RULE(myRecord) := ; //empty expression = cleared row
   ```

2. If there is only a single pattern with an associated record, and that record matches the type of the rule being defined. For example:
   ```
   RULE(myRecord) e0 := '(' USE(myRecord, 'expression') ')';
   ```

**ParsePattern Definitions**

A **parsepattern** may contain any combination of the following elements:

- **pattern-name**: The name of any previously defined PATTERN attribute.
- **(pattern)**: Parentheses may be used for grouping.
<table>
<thead>
<tr>
<th>pattern1 pattern2</th>
<th>Pattern1 followed by pattern2.</th>
</tr>
</thead>
<tbody>
<tr>
<td>'string'</td>
<td>A fixed text string, which may contain escaped octal string control characters (for example, CtrlZ is '\032').</td>
</tr>
<tr>
<td>FIRST</td>
<td>Matches the start of the string to search. This is similar to the regular expression ^ token, which is not supported.</td>
</tr>
<tr>
<td>LAST</td>
<td>Matches the end of the string to search. This is similar to the regular expression $ token, which is not supported.</td>
</tr>
<tr>
<td>ANY</td>
<td>Matches any character.</td>
</tr>
<tr>
<td>REPEAT(pattern)</td>
<td>Repeat the pattern any number of times. The regular expression syntax pattern* is supported as a shorthand for REPEAT(pattern).</td>
</tr>
<tr>
<td>REPEAT(pattern, expression)</td>
<td>Repeat the pattern expression times. The regular expression syntax pattern^&lt;count&gt; is supported as a shorthand for REPEAT(pattern,expression), but the regular expression bounded repeats syntax pattern{expression} is not.</td>
</tr>
<tr>
<td>REPEAT(pattern, low, ANY [,MIN])</td>
<td>Repeat the pattern low or more times (with the MIN option making it a minimal match). The regular expression syntax pattern+ is supported as a shorthand for REPEAT(pattern,low,ANY), but the regular expression bounded repeats syntax pattern{expression} is not.</td>
</tr>
<tr>
<td>REPEAT(pattern, low, high)</td>
<td>Repeat the pattern from low to high times. The regular expression bounded repeats syntax pattern{low,high} is not supported.</td>
</tr>
<tr>
<td>OPT(pattern)</td>
<td>An optional pattern. The regular expression syntax pattern? is supported as a shorthand for OPT(pattern).</td>
</tr>
<tr>
<td>pattern1 OR pattern2</td>
<td>Either pattern1 or pattern2. The regular expression syntax pattern1</td>
</tr>
<tr>
<td>[list-of-patterns]</td>
<td>A comma-delimited list of alternative patterns, useful for string sets. This is the same as OR.</td>
</tr>
<tr>
<td>pattern1 [NOT] IN pattern2</td>
<td>Does the text matched with pattern1 also match pattern2? Pattern1 [NOT] = pattern2 and pattern1 ! = pattern2 are the same as using IN, but may make more sense in some situations.</td>
</tr>
<tr>
<td>pattern1 [NOT] BEFORE pattern2</td>
<td>Check if the given pattern2 does [not] follow pattern1. Pattern2 is not consumed from the input.</td>
</tr>
<tr>
<td>pattern1 [NOT] AFTER pattern2</td>
<td>Check if the given pattern2 does [not] precede pattern1. Pattern2 does not consume any input. It must also be a fixed length.</td>
</tr>
<tr>
<td>pattern LENGTH(range)</td>
<td>Check whether the length of a pattern is in the range. Range can have the form &lt;value&gt;,&lt;min&gt;..&lt;max&gt;,&lt;min&gt;.. or ..&lt;max&gt;. So &quot;digit<em>3 NOT BEFORE digit&quot; could be represented as &quot;digit</em> LENGTH(3).&quot; This is more efficient, and digit* can be defined as a token. &quot;digit* LENGTH(4..6)&quot; matches 4, 5, and 6 digit sequences.</td>
</tr>
<tr>
<td>VALIDATE(pattern, is-ValidExpression)</td>
<td>Evaluate isValidExpression to check if the pattern is valid or not. isValidExpression should use MATCHTEXT or MATCHUNICODE to refer to the text that matched the pattern. For example, VALIDATE(alpha*, MATCHTEXT[4]=&quot;Q&quot;) is equivalent to alpha* = ANY<em>3 'Q' ANY</em> or more usefully: VALIDATE(alpha*, isSurnameService(MATCHTEXT));</td>
</tr>
<tr>
<td>VALIDATE(pattern, is-ValidAsciiExpression, is-ValidUnicodeExpression)</td>
<td>A two parameter variant. Use the first isValidAsciiExpression if the string being searched is ASCII; use the second if it is Unicode.</td>
</tr>
</tbody>
</table>
### ECL Language Reference

#### Parsing Support

| NOCASE(pattern) | Matches the *pattern* case insensitively, overriding the CASE option on the PARSE function. This may be nested within a CASE pattern. |
| CASE(pattern)   | Matches the *pattern* case sensitively, overriding the NOCASE option on the PARSE function. This may be nested within a NOCASE pattern. |
| *pattern* PENALTY(*cost*) | Associate a penalty *cost* with this match of the *pattern*. This can be used to recover from grammars with unknown words. This requires use of the BEST option on the PARSE operation. |
| TOKEN(pattern)  | Treat the *pattern* as a token. |

**PATTERN(‘regular expression’)**

Define a pattern using a *regular expression* built from the following supported syntax elements:

- `(x)` Grouping (not used for matching)
- `x|y` Alternatives *x* or *y*
- `xy` Concatenation of *x* and *y*
- `x*` Zero or more. Greedy and minimal versions.
- `x+` One or more. Greedy and minimal versions.
- `x?` Zero or one. Greedy and minimal versions.
- `x{m}` `x{m,n}` Bounded repeats, also minimal versions
- `[0-9abcdef]` A set of characters
  (may use `^` for exclusion list)
- `(?=...)` `(!...)` Look ahead assertion
- `(?<=...)` `(?<!...)` Look behind assertion

Escape sequences can be used to define UNICODE Character ranges. The encoding is UTF-16 Big Endian.

For example:

```
PATTERN AnyChar := PATTERN(U'[u0001-u7fff]');
```

The following character class expressions are supported (inside sets):

- `[:alnum:]` `[:cntrl:]` `[:lower:]` `[:upper:]` `[:space:]`
- `[:alpha:]` `[:digit:]` `[:print:]` `[:blank:]` `[:graph:]`
- `[:punct:]` `[:xdigit:]`

**Regular expressions do not support:**

- `$` to mark the beginning/end of the string
- Collating symbols `[:ch:]`
- Equivalence class `[:e=]`

**USE([ recstruct , ]’symbolname’)**

Specifies using a pattern defined later with the DEFINE(’symbolname’) function. This creates a forward reference, practical only on RULE patterns for tomita parsing (the PARSE option is present on the PARSE function).

**SELF**

References the pattern being defined (recursive). This is practical only in RULE patterns for tomita parsing (the PARSE option is present on the PARSE function).

**Examples:**

```ecl
rs := RECORD
STRING100 line;
END;
ds := DATASET([{'the fox; and the hen'}], rs);
PATTERN ws := PATTERN('[ \t\r\n]');
PATTERN Alpha := PATTERN('[A-Za-z]');
PATTERN Word := Alpha+;
```
PATTERN Article := ['the', 'A'];
PATTERN JustAWord := Word PENALTY(1);
PATTERN notHen := VALIDATE(Word, MATCHTEXT != 'hen');
PATTERN NoHenWord := notHen PENALTY(1);
RULE NounPhraseComponent1 := JustAWord | Article ws Word;
RULE NounPhraseComponent2 := NoHenWord | Article ws Word;
ps1 := RECORD
   out1 := MATCHTEXT(NounPhraseComponent1);
END;

ps2 := RECORD
   out2 := MATCHTEXT(NounPhraseComponent2);
END;

p1 := PARSE(ds, line, NounPhraseComponent1, ps1, BEST, MANY, NOCASE);
p2 := PARSE(ds, line, NounPhraseComponent2, ps2, BEST, MANY, NOCASE);
OUTPUT(p1);
OUTPUT(p2);

See Also: PARSE, RECORD Structure, TRANSFORM Structure, DATASET
NLP RECORD and TRANSFORM Functions

The following functions are used in field definition expressions within the RECORD structure or TRANSFORM function that defines the result set from the PARSE function:

MATCHED( [ patternreference ] )

MATCHED returns true or false as to whether the patternreference found a match. If the patternreference is omitted, it indicates whether the entire pattern matched or not (for use with the NOT MATCHED option).

MATCHTEXT [ (patternreference) ]

MATCHTEXT returns the matching ASCII text the patternreference found, or blank if not found. If the patternreference is omitted, MATCHTEXT returns all matching text.

MATCHUNICODE(patternreference)

MATCHUNICODE returns the matching Unicode text the patternreference found, or blank if not found.

MATCHLENGTH(patternreference)

MATCHLENGTH returns the number of characters in the matching text the patternreference found, or 0 if not found.

MATCHPOSITION(patternreference)

MATCHPOSITION returns the position within the text of the first character in the matching text the patternreference found, or 0 if not found.

MATCHROW(patternreference)

MATCHROW returns the entire row of the matching text the patternreference found for a RULE (valid only when the PARSE option is used on the PARSE function). This may be used to fully qualify a field in the RECORD structure of the row.

Pattern References

The patternreference parameter to these functions is a slash-delimited (/) list of previously defined PATTERN, TOKEN, or RULE attributes with or without an instance number appended in square brackets.

If an instance number is supplied, the patternreference matches a particular occurrence, otherwise it matches any. The patternreference provides a path through the regular expression grammar to a particular result. The path to a particular attribute can either be fully or partially specified.

Examples:

```ecl
PATTERN ws := PATTERN("[ \t\r\n\"]");
PATTERN arb := PATTERN("[-!.\t a-zA-Z0-9]+";)
PATTERN number := PATTERN("[0-9]+")
PATTERN age := '(' number OPT('/I') ')'  
PATTERN role := '[' arb ']';
PATTERN m_rank := '<' number '>';  
PATTERN actor := arb OPT(ws '(I)' ws);
```
NLP_layout_actor_movie := RECORD
  STRING30 actor_name := MATCHTEXT(actor);
  STRING50 movie_name := MATCHTEXT(arb[2]);  //2nd instance of arb
  UNSIGNED2 movie_year := (UNSIGNED)MATCHTEXT(age/number);
    //number within age
  UNSIGNED1 cast_rank := (UNSIGNED)MATCHTEXT(m_rank/number);
END;

Tomita Example:

// This example demonstrates the use of productions in PARSE code
//(only supported in the tomita version of PARSE).

PATTERN ws := [' ','	'];
TOKEN number := PATTERN('[0-9]+');
TOKEN plus := '+';
TOKEN minus := '-';

attrRec := RECORD
  INTEGER val;
END;

RULE(attrRec) e0 :=
  '('# USE(attrRec,expr)? ')'
  number TRANSFORM(attrRec, SELF.val := (INTEGER)$1;)
  '-' SELF TRANSFORM(attrRec, SELF.val := -$2.val;);
RULE(attrRec) e1 :=
  e0
  SELF '*' e0 TRANSFORM(attrRec, SELF.val := $1.val * $3.val;)
  USE(attrRec, e1) '/' e0
  TRANSFORM(attrRec, SELF.val := $1.val / $3.val;);
RULE(attrRec) e2 :=
  e1
  SELF plus e1 TRANSFORM(attrRec, SELF.val := $1.val + $3.val;)
  SELF minus e1 TRANSFORM(attrRec, SELF.val := $1.val - $3.val;);
RULE(attrRec) expr := e2;

infile := DATASET([{'1+2*3'},{'1+2*z'},{'1+2+(3+4)*4/2'}],
{ STRING line });
resultsRec := RECORD
  RECORDOF(infile);
  attrRec;
  STRING exprText;
  INTEGER value3;
END;

resultsRec extractResults(infile l, attrRec attr) := TRANSFORM
  SELF := l;
  SELF := attr;
  SELF.exprText := MATCHTEXT;
  SELF.value3 := MATCHROW(e0[3]).val;
END;

OUTPUT(PARSE(infile,line,expr,extractResults(LEFT, $1),
FIRST,WHOLE,PARSE,SKIP(ws)));

See Also: PARSE, RECORD Structure, TRANSFORM Structure
xmltag | A string constant naming the XPATH to the tag containing the data (see the XPATH Support section under the RECORD structure discussion). This may contain an instance number (such as tagname[1]).

transform | The TRANSFORM function that produces the child dataset.

Example:

d := DATASET({'<library><book isbn="123456789X">' + 
    '<author>Bayliss</author><title>A Way Too Far</title></book>' + 
    '<book isbn="1234567801">' + 
    '<author>Smith</author><title>A Way Too Short</title></book>' + 
    '</library>'}, 
{STRING line });

rform := RECORD
  STRING author := XMLTEXT('author');
  STRING title := XMLTEXT('title');
END;

books := PARSE(d,line,rform,XML('library/book'));

OUTPUT(books);

//*******************************************
/* The following XML can be parsed using XMLPROJECT
<XML>
<Field name='surname' distinct=2>
  <Value count=3>Halliday</Value>
  <Value count=2>Chapman</Value>
</Field>
<XML>
*/

extractedValueRec := RECORD
  STRING value;
  UNSIGNED cnt;
END;

extractedRec := RECORD
  STRING name;
UNSIGNED cnt;
DATASET(extractedValueRec) values;
END;

x := DATASET({'<XML>' +
    '<Field name="surname" distinct="2">' +
    '<Value count="3">Halliday</Value>' +
    '<Value count="2">Chapman</Value>' +
    '</Field>' +
    '</XML>'}, {STRING line});

extractedRec t1 := TRANSFORM
    SELF.name := XMLTEXT('@name');
    SELF.cnt := (UNSIGNED)XMLTEXT('@distinct');
    SELF.values := XMLPROJECT('Value',
        TRANSFORM(extractedValueRec,
            SELF.value := XMLTEXT(''),
            SELF.cnt :=
                (UNSIGNED)XMLTEXT('@count'))(cnt > 1);
    )
END;
p := PARSE(x, line, t1, XML('XML/Field'));
OUTPUT(p);

See Also: PARSE, RECORD Structure, TRANSFORM Structure, DATASET
Reserved Keywords

**ALL**

The **ALL** keyword specifies the set of all possible values when used as the default value for a passed SET parameter or as a substitute for a SET in operations that expect a defined SET of values.

Example:

```ecl
MyFunc(STRING1 val, SET OF STRING1 S=ALL) := val IN S;
    //check for presence in passed set, if passed

SET OF INTEGER4 MySet := IF(SomeCondition=TRUE,
    [88888,99999,66666,33333,55555],ALL);
MyRecs := MyFile(Zip IN MySet);
```

See Also: SET OF, Attribute Functions (Parameter Passing)
EXCEPT

EXCEPT fieldlist

fields A comma-delimited list of data fields in a RECORD structure.

The EXCEPT keyword specifies a list of fields not to use in a SORT, GROUP, DEDUP, or ROLLUP operation. This allows you to perform the operation on all fields in the RECORD EXCEPT those fields you name, making the code more readable and maintainable.

Example:

```ecl
x := DATASET([{'Taylor','Richard','Jackson','M'},
{'Taylor','David','Boca','M'},
{'Taylor','Rita','Boca','F'},
{'Smith','Richard','Mansfield','M'},
{'Smith','Oscar','Boca','M'},
{'Smith','Rita','Boca','F'}],
{STRING10 lname, STRING10 fname,
STRING10 city, STRING1 sex});
y := SORT(x,EXCEPT sex); //sort on all fields but sex

OUTPUT(y)
```

See Also: SORT, GROUP, DEDUP, ROLLUP
**EXPORT**

**EXPORT [ VIRTUAL ] definition**

<table>
<thead>
<tr>
<th>VIRTUAL</th>
<th>Optional. Specifies the <em>definition</em> is VIRTUAL. Valid only inside a MODULE Structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>definition</em></td>
<td>A valid definition.</td>
</tr>
</tbody>
</table>

The **EXPORT** keyword explicitly allows other definitions to import the specified *definition* for use. It may be IMPORTed from code in any folder, therefore its visibility scope is global.

ECL code is stored in .ecl text files which may only contain a single EXPORT or SHARED definition. This definition may be a structure that allows EXPORT or SHARED definitions within their boundaries (such as MODULE, INTERFACE, TYPE, etc.). The name of the .ecl file containing the code must exactly match the name of the single EXPORT (or SHARED) definition that it contains.

Definitions without the EXPORT or SHARED keywords are local to the file within which they reside (see Definition Visibility). A local *definition*'s scope is limited to the next SHARED or EXPORT definition, therefore they must precede that file's EXPORT or SHARED definition.

Example:

```ecl
EXPORT MyDefinition := 5;
// allows other definitions to use MyModule.MyDefinition if they import MyModule
// the filename must be MyDefinition.ecl

// and in AnotherDef.ecl we have this code:
EXPORT AnotherDef := MODULE(x)
  EXPORT INTEGER a := c * 3;
  EXPORT INTEGER b := 2;
  EXPORT VIRTUAL INTEGER c := 3; // this def is VIRTUAL
END;
```

See Also: IMPORT, SHARED, Definition Visibility, MODULE Structure
GROUP keyword

GROUP

The **GROUP** keyword is used within output *format* parameter (RECORD Structure) of a TABLE definition where optional group by *expressions* are also present. GROUP replaces the *recordset* parameter of any aggregate built-in function used in the output to indicate the operation is performed for each group of the *expression*. This is similar to an SQL "GROUP BY" clause. The most common usage is to output a table as a crosstab report.

There is also a GROUP built-in function which provides a similar functionality.

Example:

```plaintext
A := TABLE(Person,{per_st,per_sex,COUNT(GROUP)},per_st,per_sex);
// create a crosstab report of each sex in each state
```

See Also: TABLE, COUNT, AVE, MAX, MIN, SUM, VARIANCE, COVARIANCE, CORRELATION, COMBINE
IMPORT

IMPORT module-selector-list;

IMPORT folder AS alias;

IMPORT symbol-list FROM folder;

IMPORT language;

<table>
<thead>
<tr>
<th>module-selector-list</th>
<th>A comma-delimited list of folder or file names in the repository. The dollar sign ($) makes all definitions in the current folder available. The caret symbol (^) can be used as shorthand for the container of the current folder. Using a caret within the module specifier (such as, myModule.^) selects the container of that folder. A leading caret specifies the logical root of the file tree.</th>
</tr>
</thead>
<tbody>
<tr>
<td>folder</td>
<td>A folder (or file name containing an EXPORTed MODULE structure) in the repository.</td>
</tr>
<tr>
<td>AS</td>
<td>Defines a local alias name for the folder, typically used to create shorter local names for easier typing.</td>
</tr>
<tr>
<td>alias</td>
<td>The short name to use instead of the folder name.</td>
</tr>
<tr>
<td>symbol-list</td>
<td>A comma-delimited list of definitions from the folder to make available without qualification. A single asterisk (*) may be used to make all definitions from the folder available without qualification.</td>
</tr>
<tr>
<td>FROM</td>
<td>Specifies the folder name in which the symbol-list resides.</td>
</tr>
<tr>
<td>language</td>
<td>Specifies the name of an external programming language whose code you wish to embed in your ECL. A language support module for that language must have been installed in your plugins directory. This makes the language available for use by the EMBED structure and/or the IMPORT function.</td>
</tr>
</tbody>
</table>

The IMPORT keyword makes EXPORT definitions (and SHARED definitions from the same folder) available for use in the current ECL code.
### Examples:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORT $;</td>
<td>makes all definitions from the same folder available</td>
</tr>
<tr>
<td>IMPORT $, Std;</td>
<td>makes the standard library functions available, also</td>
</tr>
<tr>
<td>IMPORT MyModule;</td>
<td>makes available the definitions from MyModule folder</td>
</tr>
<tr>
<td>IMPORT $.^.MyOtherModule</td>
<td>makes available the definitions from MyOtherModule folder,</td>
</tr>
<tr>
<td></td>
<td>located in the same container as the current folder</td>
</tr>
<tr>
<td>IMPORT $.^.^.SomeOtherModule</td>
<td>makes available the definitions from SomeOtherModule folder,</td>
</tr>
<tr>
<td></td>
<td>which is located in the grandparent folder of current folder</td>
</tr>
<tr>
<td>IMPORT SomeFolder.SomeFile;</td>
<td>make available a specific file</td>
</tr>
<tr>
<td></td>
<td>containing an EXPORTed MODULE</td>
</tr>
<tr>
<td>IMPORT SomeReallyLongFolderName AS SN;</td>
<td>alias the long name as &quot;SN&quot;</td>
</tr>
<tr>
<td>IMPORT ^ AS root;</td>
<td>allows access to non-modules defined</td>
</tr>
<tr>
<td></td>
<td>in the root of the repository</td>
</tr>
<tr>
<td>IMPORT Def1,Def2 FROM Fred;</td>
<td>makes Def1 and Def2 from Fred folder available, unqualified</td>
</tr>
<tr>
<td>IMPORT * FROM Fred;</td>
<td>makes everything from Fred available, unqualified</td>
</tr>
<tr>
<td>IMPORT Dev.Me.Project1;</td>
<td>makes the Dev/Me/Project1 folder available</td>
</tr>
<tr>
<td>IMPORT Python;</td>
<td>makes Python language code embeddable</td>
</tr>
</tbody>
</table>

See Also: EXPORT, SHARED, EMBED Structure, IMPORT function
KEYED and WILD

**KEYED** *( expression [, OPT ] )*  

**WILD** *( field )*  

<table>
<thead>
<tr>
<th>expression</th>
<th>An INDEX filter condition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT</td>
<td>Only generate An INDEX filter condition.</td>
</tr>
<tr>
<td>field</td>
<td>A single field in an INDEX.</td>
</tr>
</tbody>
</table>

The **KEYED** and **WILD** keywords are valid only for filters on INDEX attributes (which also qualifies as part of the **joincondition** for a "half-keyed" JOIN). They indicate to the compiler which of the leading index fields are used as filters (KEYED) or wildcarded (WILD) so that the compiler can warn you if you've gotten it wrong. Trailing fields not used in the filter are ignored (always treated as wildcards).

The rules for their use are as follows (the term "segmonitor" refers to an internal object created to represent the possible match conditions for a single keyable field):

1. **KEYED** generates a segmonitor. The segmonitor may be a wild one if the *expression* can never be false, such as:

```ecl
KEYED(inputval = '' OR field = inputval)
```

2. **WILD** generates a wild segmonitor, unless there is also a **KEYED()** filter on the same field.

3. **KEYED**, OPT generates a non-wild segmonitor only if the preceding field did.

4. Any field that is both **KEYED** and **KEYED OPT** creates a compile time error.

5. If **WILD** or **KEYED** are not specified for any fields, segmonitors are generated for all keyable conditions.

6. An INDEX filter condition with no **KEYED** specified generates a wild segmonitor (except as specified by 5).

7. **KEYED** limits are based upon all non-wild segmonitors.

8. Conditions that do not generate segmonitors are post-filtered.

Example:

```ecl
ds := DATASET('~LR::person',  
             ( STRING15 f1, STRING15 f2, STRING15 f3, STRING15 f4,  
             UNSIGNED8 filepos{VIRTUAL(fileposition)}) }, FLAT);  
ix := INDEX(ds, { ds },'\lexis\person.name_first.key');  
//*** Valid examples ****/
COUNT(ix(KEYED(f1='Kevin1')));  
   // legal because only f1 is used.  
COUNT(ix(KEYED(f1='Kevin2' and f2='Halliday')));  
   // legal because both f1 and f2 are used  
COUNT(ix(KEYED(f2='Kevin3') and WILD(f1)));  
   // keyed f2, but ok because f1 is marked as wild.  
COUNT(ix(f2='Halliday'));  
   // ok - if keyed isn't used then it doesn't have to have  
   // a wild on f1
```
COUNT(ix(KEYED(f1='Kevin3') and KEYED(f2='Kevin4') and WILD(f1)));  
  // it is ok to mark as wild and keyed otherwise you can get  
  // in a mess with compound queries.

COUNT(ix(f1='Kevin3' and KEYED(f2='Kevin4') and WILD(f1)));  
  // can also be wild and a general expression.

/***Error examples ***/

COUNT(ix(KEYED(f3='Kevin3' and f2='Halliday')));  
  // missing WILD(f1) before keyed

COUNT(ix(KEYED(f3='Kevin3') and f2='Halliday'));  
  // missing WILD(f1) before keyed after valid field

COUNT(ix(KEYED(f3='Kevin3') and WILD(f2)));  
  // missing WILD(f1) before a wild

COUNT(ix(WILD(f3) and f2='Halliday'));  
  // missing WILD(f1) before wild after valid field

COUNT(ds(KEYED(f1='Kevin')));  
  //KEYED not valid in DATASET filters

See Also: INDEX, JOIN, FETCH
LEFT and RIGHT

LEFT

RIGHT

The LEFT and RIGHT keywords indicate the left and right records of a record set. These may be used to substitute as parameters passed to TRANSFORM functions or in expressions in functions where a left and right record are implicit, such as DEDUP and JOIN.

Example:

dup_flags := JOIN(person,person,
    LEFT.current_address_key=RIGHT.current_address_key
    AND fuzzy_equal,req_output(LEFT,RIGHT));

See Also: TRANSFORM Structure, DEDUP
LIKEY and UNLIKELY

LIKEY(filtercondition, [ likelihood ]);  
UNLIKELY(filtercondition);

<table>
<thead>
<tr>
<th>filtercondition</th>
<th>A filter condition for the hint.</th>
</tr>
</thead>
<tbody>
<tr>
<td>likelihood</td>
<td>The probability value expressed in a decimal value between 0 and 1.</td>
</tr>
</tbody>
</table>

The LIKEY/UNLIKELY hint can be wrapped around a filter condition to indicate to the code generator the likelihood that the filter condition will filter the record.

LIKEY specifies that the filter condition is likely to match most records. UNLIKELY specifies that very few records are likely to be matched.

Specific probability value may be provided for LIKEY. The probability value is decimal value greater than 0 and less than 1. The closer this value is to 1.0 the more likely that the filter condition is likely to match a record. The closer the value is to 0.0 the less likely the filter condition is to match records. The code generator makes use of the likelihood information to produce better code.

The code generator uses the LIKEY/UNLIKELY hint together with the count of usage, to determine the cost of spilling and the cost of re-filtering the dataset every time it is used. Spills are only be generated when the cost of spilling is lower than the cost of re-filtering the dataset every time.

For example, say there is a dataset of people with millions of records. A filter is created to retain all records where the age is less than 100. The filter is expected to retain 99.9% of records. This filter result is used by 3 different activities. The cost of spilling the results of the filter is likely to be significantly higher than the simply re-filtering the input dataset every time it used. LIKEY can be used to share this likelihood information with the code generator so that it may make sensible decisions regarding when to spill.

Example:

```
PeopleYoungerThan100 := AllPeople( LIKEY(age < 100, 0.999) );  
// Probably not worth spilling PeopleYoungerThan100
PeopleOlderThan100 := AllPeople( UNLIKELY(age>100) );  
// Probably worth spilling even if PeopleOlderThan100 is used by only a couple of activities
```
Reserved Keywords

ROWS(LEFT) and ROWS(RIGHT)

ROWS(LEFT)

ROWS(RIGHT)

The **ROWS(LEFT)** and **ROWS(RIGHT)** keywords indicate the parameter being passed to the TRANSFORM function is a record set. These are used in functions where a dataset is being passed, such as COMBINE, ROLLUP, JOIN, DENORMALIZE, and LOOP.

Example:

```ecl
NormRec := RECORD
  STRING20 thaname;
  STRING20 addr;
END;
NamesRec := RECORD
  UNSIGNED1 numRows;
  STRING20 thaname;
  DATASET(NormRec) addresses;
END;
NamesTable := DATASET([ {0,'Kevin',[]},{0,'Liz',[]},
  {0,'Mr Nobody',[]},{0,'Anywhere',[]}],NamesRec);
NormAddrs := DATASET([{'Kevin','10 Malt Lane'},
  {'Liz','10 Malt Lane'},
  {'Liz','3 The cottages'},
  {'Anywhere','Here'},
  {'Anywhere','There'},
  {'Anywhere','Near'},
  {'Anywhere','Far'}],NormRec);
NamesRec DeNormThem(NamesRec L, DATASET(NormRec) R) := TRANSFORM
  SELF.NumRows := COUNT(R);
  SELF.addresses := R;
  SELF := L;
END;
DeNormedRecs := DENORMALIZE(NamesTable, NormAddrs,
  LEFT.thename = RIGHT.thename,
  GROUP,
  DeNormThem(LEFT,ROWS(RIGHT)));
OUTPUT(DeNormedRecs);
```

See Also: TRANSFORM Structure, COMBINE, ROLLUP, JOIN, DENORMALIZE, LOOP
SELF

SELF.element

element The name of a field in the result type RECORD structure of a TRANSFORM structure.

The SELF keyword is used in TRANSFORM structures to indicate a field in the output structure. It should not be used on the right hand side of any attribute definition.

Example:

```ecl
Ages := RECORD
  INTEGER8 Age; //a field named "Age"
END;

TodaysYear := 2001;
Ages req_output(person l) := TRANSFORM
  SELF.Age := TodaysYear - l.birthdate[1..4];
END;
```

See Also: TRANSFORM Structure
**SHARE**

**SHARE [ VIRTUAL ] definition**

<table>
<thead>
<tr>
<th>VIRTUAL</th>
<th>Optional. Specifies the definition is VIRTUAL. Valid only inside a MODULE Structure.</th>
</tr>
</thead>
<tbody>
<tr>
<td>definition</td>
<td>A valid definition.</td>
</tr>
</tbody>
</table>

The `SHARE` keyword explicitly allows other definitions within the same folder to import the specified definition for use throughout the module/folder/directory (i.e. module scope), but not outside that scope.

ECL code is stored in `.ecl` text files which may only contain a single EXPORT or SHARE definition. This definition may be a structure that allows EXPORT or SHARE definitions within their boundaries (such as MODULE, INTERFACE, TYPE, etc.). The name of the `.ecl` file containing the code must exactly match the name of the single EXPORT (or SHARE) definition that it contains.

Definitions without the EXPORT or SHARE keywords are local to the file within which they reside (see Definition Visibility). A local definition's scope is limited to the next SHARE or EXPORT definition, therefore they must precede that file's EXPORT or SHARE definition.

Example:

```ecl
//this code is contained in the GoodHouses.ecl file
BadPeople := Person(EXISTS(trades(EXISTS(phr(phr_rate > '4')))));  
            //local only to the GoodHouses definition
SHARE GoodHouses := Household(~EXISTS(BadPeople));  
            //available all thru the module

//and in AnotherDef.ecl we have this code:
EXPORT AnotherDef := MODULE(x)  
    EXPORT INTEGER a := c * 3;  
    EXPORT INTEGER b := 2;  
    SHARE VIRTUAL INTEGER c := 3; //this def is VIRTUAL
    EXPORT VIRTUAL INTEGER d := c + 3; //this def is VIRTUAL
    EXPORT VIRTUAL INTEGER e := c + 3; //this def is VIRTUAL
END;
```

See Also: IMPORT, EXPORT, Definition Visibility, MODULE Structure
**SKIP**

**SKIP** is valid for use only within a TRANSFORM structure and may be used anywhere an expression can be used to indicate the current output record should not be generated into the result set. COUNTER values are incremented even when SKIP eliminates generating the current record.

Example:

```ecl
SequencedAges := RECORD
  Ages;
  INTEGER8 Sequence := 0;
END;

SequencedAges AddSequence(Ages l, INTEGER c) := TRANSFORM
  SELF.Sequence := IF(c % 2 = 0, SKIP, c); //skip the even recs
  SELF := l;
END;

SequencedAgedRecs := PROJECT(AgedRecs, AddSequence(LEFT,COUNTER));
```

See Also: TRANSFORM Structure
TRUE and FALSE

TRUE

FALSE

The **TRUE** and **FALSE** keywords are Boolean constants.

Example:

```plaintext
BooleanTrue := TRUE;
BooleanFalse := FALSE;

OUTPUT(BooleanTrue);
OUTPUT(BooleanFalse)
```

See Also: BOOLEAN
Special Structures
BEGINC++ Structure

resulttype funcname ( parameterlist ) := BEGINC++

code

ENDC++;

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>resulttype</td>
<td>The ECL return value type of the C++ function.</td>
</tr>
<tr>
<td>funcname</td>
<td>The ECL definition name of the function.</td>
</tr>
<tr>
<td>parameterlist</td>
<td>A comma separated list of the parameters to pass to the function.</td>
</tr>
<tr>
<td>code</td>
<td>The C++ function source code.</td>
</tr>
</tbody>
</table>

The BEGINC++ structure makes it possible to add in-line C++ code to your ECL. This is useful where string or bit processing would be complicated in ECL, and would be more easily done in C++, typically for a one-off use. For more commonly used C++ code, writing a plugin would be a better solution (see the External Service Implementation discussion).

The implementation must be written to be thread safe and any calls to external libraries must be made to thread safe versions of those libraries.

You can use EMBED instead of BEGINC++ to embed C++ code and specify additional options (for example, DISTRIBUTED) using this form:

```ecl
myFunction(string name) := EMBED(C++ [: options])
    ... text
ENDEMBED
```

WARNING: This feature could create memory corruption and/or security issues, so great care and forethought are advised--consult with Technical Support before using.

ECL to C++ Mapping

Types are passed as follows:

```c
// The following typedefs are used below:
typedef unsigned size32_t;
typedef wchar_t UChar; [ unsigned short in linux ]
```

The following list describes the mappings from ECL to C++. For embedded C++ the parameters are always converted to lower case, and capitalized in conjunctions (see below).

<table>
<thead>
<tr>
<th>ECL</th>
<th>C++ [Linux in brackets]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOOLEAN xyz</td>
<td>bool xyz</td>
</tr>
<tr>
<td>INTEGER1 xyz</td>
<td>signed char xyz</td>
</tr>
<tr>
<td>INTEGER2 xyz</td>
<td>int16_t xyz</td>
</tr>
<tr>
<td>INTEGER4 xyz</td>
<td>int32_t xyz</td>
</tr>
<tr>
<td>INTEGER8 xyz</td>
<td>signed __int64 xyz [ long long ]</td>
</tr>
<tr>
<td>UNSIGNED1 xyz</td>
<td>unsigned char xyz</td>
</tr>
<tr>
<td>UNSIGNED2 xyz</td>
<td>uint16_t xyz</td>
</tr>
<tr>
<td>UNSIGNED4 xyz</td>
<td>uint32_t xyz</td>
</tr>
<tr>
<td>UNSIGNED8 xyz</td>
<td>unsigned __int64 xyz [ unsigned long long xyz ]</td>
</tr>
<tr>
<td>REAL4 xyz</td>
<td>float xyz</td>
</tr>
<tr>
<td>REAL/REAL8 xyz</td>
<td>double xyz</td>
</tr>
<tr>
<td>DATA xyz</td>
<td>size32_t lenXyz, void * xyz</td>
</tr>
<tr>
<td>STRING xyz</td>
<td>size32_t lenXyz, char * xyz</td>
</tr>
<tr>
<td>VARSTRING xyz</td>
<td>char * xyz;</td>
</tr>
<tr>
<td>QSTRING xyz</td>
<td>size32_t lenXyz, char * xyz</td>
</tr>
</tbody>
</table>
UNICODE xyz          size32_t lenXyz, UChar * xyz
VARUNICODE xyz       UChar * xyz
DATA<nn> xyz         void * xyz
STRING<nn> xyz       char * xyz
QSTRING<nn> xyz      char * xyz
UNICODE<nn> xyz      UChar * xyz
SET OF ... xyz      bool isAllXyz, size32_t lenXyz, void * xyz

Note that strings of unknown length are passed differently from those with a known length. A variable length input string is passed as a number of characters, not the size (i.e. qstring/unicode), followed by a pointer to the data, like this (size32_t is an UNSIGNED4):

STRING ABC -> size32_t lenAbc, const char * abc;
UNICODE ABC -> size32_t lenABC, const UChar * abc;

A dataset is passed as a size/pointer pair. The length gives the size of the following dataset in bytes. The same naming convention is used:

DATASET(r) ABC -> size32_t lenAbc, const void * abc
    The rows are accessed as x+0, x + length(row1), x + length(row1) + length(row2)
LINKCOUNTED DATASET(r) ABC -> size32_t countAbc, const byte * * abc
    The rows are accessed as x[0], x[1], x[2]

NOTE: variable length strings within a record are stored as a 4 byte number of characters, followed by the string data.

Sets are passed as a set of parameters (all, size, pointer):

SET OF UNSIGNED4 ABC -> bool isAllAbc, size32_t lenAbc, const void * abc

Return types are handled as C++ functions returning the same types with some exceptions. The exceptions have some extra initial parameters to return the results in:

ECL                      C++ [Linux in brackets]
DATA xyz                 size32_t & __lenResult, void * & __result
STRING xyz               size32_t & __lenResult, char * & __result
CONST STRING xyz         size32_t lenXyz, const char * xyz
QSTRING xyz              size32_t & __lenResult, char * & __result
UNICODE xyz              size32_t & __lenResult, UChar * & __result
CONST UNICODE xyz        size32_t & __lenResult, const UChar * & __result
DATA<nn> xyz             void * __result
STRING<nn> xyz           char * __result
QSTRING<nn> xyz          char * __result
UNICODE<nn> xyz          UChar * __result
SET OF ... xyz          bool __isAllResult, size32_t & __lenResult, void * & __result
DATASET(r)               size32_t & __lenResult, void * & __result
LINKCOUNTED DATASET(r)   size32_t & __countResult, byte * * & __result
STREAMED DATASET(r)      returns a pointer to an IRowStream interface
                          (see the eclhelper.hpp include file for the definition)

For example,

STRING process(STRING value, INTEGER4 len)

has the prototype:

void process(size32_t & __lenResult, char * & __result,
             size32_t lenValue, char * value, int len);
A function that takes a string parameter should also have the type prefixed by `const` in the ECL code so that modern compilers don't report errors when constant strings are passed to the function.

```c++
BOOLEAN isUpper(const string mystring) := BEGINC++
  size_t i=0;
  while (i < lenMystring)
  {
    if (!isupper((byte)mystring[i]))
      return false;
    i++;
  }
  return true;
ENDC++;
isUpper('JIM');
```

Parameters can also include streamed datasets.

If stream is specified on the dataset then the parameter is passed as an IRowStream. The next row from the dataset is obtained by calling:

```c++
dataset->nextRow();
```

After it has been processed the row must be freed by calling

```c++
rtlReleaseRow(next);
```

For example:

```c++
traceDataset(STREAMED DATASET(r) ds, BOOLEAN isLocal = FALSE) := EMBED(C++)
#include <stdio.h>
#body
for(;;)
{
  const byte * next = (const byte *)ds->nextRow();
  if (!next)
    return;
  unsigned __int64 id = *(__uint64 *)(next);
  size32_t lenName = *(size32_t *)(next + sizeof(__uint64));
  const char * name = (char *)(next + sizeof(__uint64) + sizeof(size32_t));
  printf("id(%lu) name(%.*s)\n", (unsigned)id, lenName, name);
  rtlReleaseRow(next);
}
ENDEMBED;
```

If the result of a C++ function is a streamed dataset, then it needs to return an instance of an IRowStream interface. The function will also be passed an extra implicit parameter:

```c++
IEngineRowAllocator * _resultAllocator
```

which is used to allocate the rows that are returned from the function.

For example:

```c++
// This function takes two streamed inputs and outputs the result of two values
// from the left multiplied together and added to a row from the right.
STREAMED DATASET(r) myDataset(STREAMED DATASET(r) ds1, STREAMED DATASET(r) ds2)
  := EMBED(C++ : activity)
#include <stdio.h>
#body
class MyStreamInlineDataset : public RtlCInterface, implements IRowStream
{
```
public:

    MyStreamInlineDataset(IEngineRowAllocator * _resultAllocator, IRowStream * _ds1,
                           IRowStream * _ds2)
        : resultAllocator(_resultAllocator), ds1(_ds1), ds2(_ds2)
    {
    
    
    
void * nextRow() override
    {
        const byte * next1a = (const byte *)ds1->nextRow();
        if (!next1a)
            return nullptr;
        const byte * next1b = (const byte *)ds1->nextRow();
        const byte * next2 = (const byte *)ds2->nextRow();
        if (!next1b || !next2)
            rtlFailUnexpected();
        unsigned __int64 value1a = *(const unsigned __int64 *)next1a;
        unsigned __int64 value1b = *(const unsigned __int64 *)next1b;
        unsigned __int64 value2 = *(const unsigned __int64 *)next2;
        rtlReleaseRow(next1a);
        rtlReleaseRow(next1b);
        rtlReleaseRow(next2);
        unsigned __int64 result = value1a * value1b + value2;
        RtlDynamicRowBuilder rowBuilder(resultAllocator);
        byte * row = rowBuilder.getSelf();
        *(__uint64 *)(row) = result;
        return rowBuilder.finalizeRowClear(sizeof(unsigned __int64));
    }

virtual void stop() override
    {
        ds1->stop();
        ds2->stop();
    }

protected:

    Linked<IEngineRowAllocator> resultAllocator;
    IRowStream * ds1;
    IRowStream * ds2;
};

    return new MyStreamInlineDataset(_resultAllocator, ds1, ds2);

Note: If the resulting row does not have a fixed size, you should call:

    byte * row = rowBuilder.ensureCapacity(<totalSize>, nullptr);

instead of:

    byte * row = rowBuilder.getSelf();

This code uses a RtlDynamicRowBuilder which is a class used by the code generator. Instead of using the
RtlDynamicRowBuilder class, you could directly call resultAllocator->createRow().

When a data type is included in an input row, rather than being passed as a parameter, the format is the
same as the parameters, except that instead of having a pointer to the string etc., the string follows the 4-
byte length. The data in the row is not aligned; that is, it has packing of 1.

Available Options

| #option pure | By default, embedded C++ functions are assumed to have side-effects, which | means the generated code won't be as efficient as it might be since the calls | can't be shared. Adding #option pure inside the embedded C++ code causes | it to be treated as a pure function without side effects. |
#option once
Indicates the function has no side effects and is evaluated at query execution time, even if the parameters are constant, allowing the optimizer to make more efficient calls to the function in some cases.

#option action
Indicates side effects, requiring the optimizer to keep all calls to the function.

#body
Delimits the beginning of executable code. All code that precedes #body (such as #include) is generated outside the function definition; all code that follows it is generated inside the function definition.

Example:

```c
//static int add(int x, int y) {
INTEGER4 add(INTEGER4 x, INTEGER4 y) := BEGINC++
   #option pure
   return x + y;
ENDC++;
OUTPUT(add(10, 20));

//static void reverseString(size32_t & __lenResult, char * & __result,
// size32_t lenValue, char * value) {
STRING reverseString(STRING value) := BEGINC++
   size32_t len = lenValue;
   char * out = (char *)rtlMalloc(len);
   for (unsigned i = 0; i < len; i++)
      out[i] = value[len-1-i];
   __lenResult = len;
   __result = out;
ENDC++;
OUTPUT(reverseString('Kevin'));

// This is a function returning an unknown length string via the
// special reference parameters __lenResult and __result

//this function demonstrates #body, allowing #include to be used
BOOLEAN nocaseInList(STRING search,
               SET OF STRING values) := BEGINC++
#include <string.h>
#body
   if (isAllValues)
      return true;
   const byte * cur = (const byte *)values;
   const byte * end = cur + lenValues;
   while (cur != end)
      {
         unsigned len = *(unsigned *)cur;
         cur += sizeof(unsigned);
         if (lenSearch == len && memicmp(search, cur, len) == 0)
            return true;
         cur += len;
      }
   return false;
ENDC++;

//and another example, generating a variable number of Xes
STRING buildString(INTEGER4 value) := BEGINC++
   char * out = (char *)rtlMalloc(value);
   for (unsigned i = 0; i < value; i++)
      out[i] = 'X';
   __lenResult = value;
   __result = out;
ENDC++;

//examples of embedded, LINKCOUNTED, and STREAMED DATASETs
```
inRec := { unsigned id; };
doneRec := { unsigned4 execid; };
out1rec := { unsigned id; };
out2rec := { real id; };

DATASET(doneRec) doSomethingNasty(DATASET(inRec) input) := BEGINC++
  __lenResult = 4;
  __result = rtlMalloc(8);
  *(unsigned *)__result = 91823;
ENDC++;

DATASET(out1Rec) extractResult1(doneRec done) := BEGINC++
  const unsigned id = *(unsigned *)done;
  const unsigned cnt = 10;
  __lenResult = cnt * sizeof(unsigned __int64);
  __result = rtlMalloc(__lenResult);
  for (unsigned i=0; i < cnt; i++)
    ((__int64 *)__result)[i] = id + i + 1;
ENDC++;

LINKCOUNTED DATASET(out2Rec) extractResult2(doneRec done) := BEGINC++
  const unsigned id = *(unsigned *)done;
  const unsigned cnt = 10;
  __countResult = cnt;
  __result = _resultAllocator->createRowset(cnt);
  for (unsigned i=0; i < cnt; i++)
  {
    size32_t allocSize;
    void * row = _resultAllocator->createRow(allocSize);
    *(double *)row = id + i + 1;
    __result[i] = (byte *)_resultAllocator->finalizeRow(allocSize, row, allocSize);
  }
ENDC++;

STREAMED DATASET(out1Rec) extractResult3(doneRec done) := BEGINC++
  class myStream : public IRowStream, public RtlCInterface
  {
    public:
      myStream(IEngineRowAllocator * _allocator, unsigned _id) : allocator(_allocator), id(_id), idx(0) {}
      RTLIMPLEMENT_IINTERFACE
      virtual const void *nextRow()
      {
        if (idx >= 10)
          return NULL;
        size32_t allocSize;
        void * row = allocator->createRow(allocSize);
        *(unsigned __int64 *)row = id + ++idx;
        return allocator->finalizeRow(allocSize, row, allocSize);
      }
    virtual void stop() {}
  }
private:
  Linked<IEngineRowAllocator> allocator;
  unsigned id;
  unsigned idx;
  
  #body
  const unsigned id = *(unsigned *)done;
  return new myStream(_resultAllocator, id);
ENDC++;

ds := DATASET([1,2,3,4], inRec);
processed := doSomethingNasty(ds);
out1 := NORMALIZE(processed, extractResult1(LEFT), TRANSFORM(RIGHT));
out2 := NORMALIZE(processed, extractResult2(LEFT), TRANSFORM(RIGHT));
out3 := NORMALIZE(processed, extractResult3(LEFT), TRANSFORM(RIGHT));

SEQUENTIAL(OUTPUT(out1), OUTPUT(out2), OUTPUT(out3));

See Also: External Service Implementation, EMBED Structure
EMBED Structure

\[
\text{resulttype } \text{funcname} ( \text{parameterlist} ) := \text{EMBED} ( \text{language} : \text{TIME} \{ (\text{label}) \} ]
\]

\[\text{code} \]
\[\text{ENDEMBED;}\]

\[
\text{resulttype } \text{funcname} ( \text{parameterlist} ) := \text{EMBED} ( \text{language}, \text{code} : \text{TIME} \{ (\text{label}) \} ]);
\]

<table>
<thead>
<tr>
<th>resulttype</th>
<th>The ECL return value type of the function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcname</td>
<td>The ECL definition name of the function.</td>
</tr>
<tr>
<td>parameterlist</td>
<td>A comma separated list of the parameters to pass to the function.</td>
</tr>
<tr>
<td>language</td>
<td>The name of the programming language being embedded. A language support module for that language must have been installed in your plugins directory. Modules are provided for languages such as Java, R, Javascript, and Python. You can write your own pluggable language support module for any language not already supported by using the supplied ones as examples or starting points.</td>
</tr>
<tr>
<td>code</td>
<td>The source code to embed.</td>
</tr>
<tr>
<td>TIME</td>
<td>Tracks timing of an external function call or embedded source code and reports them back as metrics to the user.</td>
</tr>
<tr>
<td>label</td>
<td>Optional. A string constant containing the name to associate with the timer. If omitted, the default is used.</td>
</tr>
</tbody>
</table>

The EMBED structure makes it possible to add in-line language code to your ECL. This is similar to the BEGINC++ structure, but available for any language with a pluggable language support module installed, such as R, Javascript, and Python. Others may follow or people can write their own using the supplied ones as templates/examples/starting points. This may be used to write Javascript, R, or Python code, but is not usable with Java code (use the IMPORT function for Java code).

The parameter types that can be passed and returned will vary by language, but in general the simple scalar types (INTEGER, REAL, STRING, UNICODE, BOOLEAN, and DATA) and SETs of those scalar types are supported, so long as there is an appropriate data type in the language to map them to.

The first form of EMBED is the structure that must terminate with ENDEMBED. This may contain any code in the supported language.

The second form of EMBED is a self-contained function. The code parameter contains all the code to execute, making this useful only for very simple expressions.

You can use EMBED instead of BEGINC++ to embed C++ code and specify additional options (for example, DISTRIBUTED) using this form:

\[
\text{myFunction(string name)} := \text{EMBED(C++ }[: \text{options}]\right) \\
\quad \ldots \text{ text} \\
\text{ENDEMBED}
\]

WARNING: This feature could create memory corruption and/or security issues, so great care and forethought are advised--consult with Technical Support before using.
Examples:

//First form: a structure
IMPORT Python3 AS Python; //make Python language available

INTEGER addone(INTEGER p) := EMBED(Python : TIME('MyTime'))
# Python code that returns one more than the value passed to it
if p < 10:
    return p+1
else:
    return 0
ENDEMBED;
addone(3);
addone(11);

//Second form: a function
IMPORT Python3 as Python;
INTEGER addtwo(INTEGER p) := EMBED(Python, 'p+2' : TIME('MyTime'));
addtwo(27);

See Also: BEGINC++ Structure, IMPORT, IMPORT function
FUNCTION Structure

[resulttype] funcname ( parameterlist ) := FUNCTION

code
RETURN retval;
END;

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>resulttype</td>
<td>The return value type of the function. If omitted, the type is implicit from the retval expression.</td>
</tr>
<tr>
<td>funcname</td>
<td>The ECL attribute name of the function.</td>
</tr>
<tr>
<td>parameterlist</td>
<td>A comma separated list of the parameters to pass to the function. These are available to all attributes defined in the FUNCTION's code.</td>
</tr>
<tr>
<td>code</td>
<td>The local attribute definitions that comprise the function. These may not be EXPORT or SHARED attributes, but may include actions (like OUTPUT).</td>
</tr>
<tr>
<td>RETURN</td>
<td>Specifies the function's return value expression--the retval.</td>
</tr>
<tr>
<td>retval</td>
<td>The value, expression, recordset, row (record), or action to return.</td>
</tr>
</tbody>
</table>

The FUNCTION structure allows you to pass parameters to a set of related attribute definitions. This makes it possible to pass parameters to an attribute that is defined in terms of other non-exported attributes without the need to parameterise all of those as well.

Side-effect actions contained in the code of the FUNCTION must have definition names that must be referenced by the WHEN function to execute.

Example:

```
//a coordinated set of 3 examples
IMPORT Std;
NameRec := RECORD
    STRING5 title := '';
    STRING20 fname := '';
    STRING20 mname := '';
    STRING20 lname := '';
    STRING5 name_suffix := '';
    STRING3 name_score := '';
END;
MyRecord := RECORD
    UNSIGNED id;
    STRING uncleanedName;
    NameRec Name;
END;
ds := DATASET([{'Mr. John Smith JR'},
    {'Mrs. Susie Samantha Jones 3'},
    {'Dr. Fred Taylor SR'}],MyRecord);
STRING73 CleanPerson73(STRING inputName) := FUNCTION
    suffix :=[' 0',' 1',' 2',' 3',' 4',' 5',' 6',' 7',' 8',' 9',
    ' J',' JR',' S',' SR'];
    InWords := Std.Str.CleanSpaces(inputName);
    HasSuffix := InWords[LENGTH(TRIM(InWords))-1 ..] IN suffix;
    WordCount := LENGTH(TRIM(InWords,LEFT,RIGHT)) -
                LENGTH(TRIM(InWords,ALL)) + 1;
    HasMiddle := WordCount = 5 OR (WordCount = 4 AND NOT HasSuffix);
    Sp1 := Std.Str.Find(InWords,' ',1);
    Sp2 := Std.Str.Find(InWords,' ',2);```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
Sp3 := Std.Str.Find(InWords,' ',3);
Sp4 := Std.Str.Find(InWords,' ',4);
STRING5 title := InWords[1..Sp1-1];
STRING20 fname := InWords[Sp1+1..Sp2-1];
STRING20 mname := IF(HasMiddle,InWords[Sp2+1..Sp3-1],'');
STRING20 lname := MAP(HasMiddle AND NOT HasSuffix => InWords[Sp3+1..],
NOT HasMiddle AND NOT HasSuffix => InWords[Sp2+1..],
NOT HasMiddle AND HasSuffix => InWords[Sp3+1..Sp4-1],
'');
STRING5 name_suffix := IF(HasSuffix,InWords[LENGTH(TRIM(InWords))-1..],'');
STRING3 name_score := '';
RETURN title + fname + mname + lname + name_suffix + name_score;
END;

//Example 1 - a transform to create a row from an uncleaned name
NameRec createRow(STRING inputName) := TRANSFORM
  cleanedText := CleanPerson73(inputName);
  SELF.title := cleanedText[1..5];
  SELF.fname := cleanedText[6..25];
  SELF.mname := cleanedText[26..45];
  SELF.lname := cleanedText[46..65];
  SELF.name_suffix := cleanedText[66..70];
  SELF.name_score := cleanedText[71..73];
END;
myRecord t(myRecord l) := TRANSFORM
  SELF.Name := ROW(createRow(l.uncleanedName));
  SELF := l;
END;
y := PROJECT(ds, t(LEFT));
OUTPUT(y);

//Example 2 - an attribute using that transform to generate the row.
NameRec cleanedName(STRING inputName) := ROW(createRow(inputName));
myRecord t2(myRecord l) := TRANSFORM
  SELF.Name := cleanedName(l.uncleanedName);
  SELF := l;
END;
y2 := PROJECT(ds, t2(LEFT));
OUTPUT(y2);

//Example 3 - Encapsulate the transform inside the attribute by
// defining a FUNCTION.
NameRec cleanedName2(STRING inputName) := FUNCTION
  NameRec createRow := TRANSFORM
    cleanedText := CleanPerson73(inputName);
    SELF.title := cleanedText[1..5];
    SELF.fname := cleanedText[6..25];
    SELF.mname := cleanedText[26..45];
    SELF.lname := cleanedText[46..65];
    SELF.name_suffix := cleanedText[66..70];
    SELF.name_score := cleanedText[71..73];
  END;
  RETURN ROW(createRow);
END;
myRecord t3(myRecord l) := TRANSFORM
  SELF.Name := cleanedName2(l.uncleanedName);
  SELF := l;
END;
y3 := PROJECT(ds, t3(LEFT));
OUTPUT(y3);
//Example using MODULE structure to return multiple values from a FUNCTION
OperateOnNumbers(Number1, Number2) := FUNCTION
    result := MODULE
        EXPORT Multiplied := Number1 * Number2;
        EXPORT Differenced := Number1 - Number2;
        EXPORT Summed := Number1 + Number2;
    END;
    RETURN result;
END;

OUTPUT(OperateOnNumbers(23,22).Multiplied);  //506  
OUTPUT(OperateOnNumbers(23,22).Differenced); //1  
OUTPUT(OperateOnNumbers(23,22).Summed);      //45

//a FUNCTION with side-effect Action
namesTable := FUNCTION
    namesRecord := RECORD
        STRING20 surname;
        STRING10 forename;
        INTEGER2 age := 25;
    END;
    o := OUTPUT('namesTable used by user <x>');
    ds1 := DATASET([{'x','y',22}],namesRecord);
    RETURN WHEN(ds1,o);
END;

z := namesTable : PERSIST('z');  
//the PERSIST causes the side-effect action to execute only when the PERSIST is re-built
OUTPUT(z);

See Also: MODULE Structure, TRANSFORM Structure, WHEN
**FUNCTIONMACRO Structure**

\[ \text{[resulttype]} \text{funcname ( parameterlist )} := \text{FUNCTIONMACRO} \]

**code**

```
RETURN retval;
ENDMACRO;
```

<table>
<thead>
<tr>
<th><strong>resulttype</strong></th>
<th>The return value type of the function. If omitted, the type is implicit from the retval expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>funcname</strong></td>
<td>The ECL definition name of the function/macro.</td>
</tr>
<tr>
<td><strong>parameterlist</strong></td>
<td>A comma separated list of names (tokens) of the parameters that will be passed to the function/macro. These names are used in the code and retval to indicate where the passed parameter values are substituted when the function/macro is used. Value types for these parameters are not allowed, but default values may be specified as string constants.</td>
</tr>
<tr>
<td><strong>code</strong></td>
<td>The local definitions that comprise the function. These may not be EXPORT or SHARED, but may include actions (like OUTPUT).</td>
</tr>
<tr>
<td><strong>RETURN</strong></td>
<td>Specifies the return value expression--the retval.</td>
</tr>
<tr>
<td><strong>retval</strong></td>
<td>The value, expression, recordset, row (record), or action to return.</td>
</tr>
</tbody>
</table>

The **FUNCTIONMACRO** structure is a code generation tool, like the MACRO structure, coupled with the code encapsulation benefits of the FUNCTION structure. One advantage the FUNCTIONMACRO has over the MACRO structure is that it may be called in an expression context, just like a FUNCTION would be.

Unlike the MACRO structure, #UNIQUENAME is not necessary to prevent internal definition name clashes when the FUNCTIONMACRO is used multiple times within the same visibility scope. However, the **LOCAL** keyword must be explicitly used within the FUNCTIONMACRO if a definition name in its code may also have been defined outside the FUNCTIONMACRO and within the same visibility scope -- LOCAL clearly identifies that the definition is limited to the code within the FUNCTIONMACRO.

**Example:**

This example demonstrates the **FUNCTIONMACRO** used in an expression context. It also shows how the FUNCTIONMACRO may be called multiple times without name clashes from its internal definitions:

```ecl
EXPORT Field_Population(infile,infield,compareval) := FUNCTIONMACRO
  c1 := COUNT(infile(infile=compareval));
  c2 := COUNT(infile);
  RETURN DATASET([{'Total Records',c2},
                    {'Recs=' + #TEXT(compareval),c1},
                    {'Population Pct', (INTEGER)(((c2-c1)/c2)* 100.0)}],
                    {STRING15 valuetype,INTEGER val});
ENDMACRO;

d1 := dataset([{'M'},{'M'},{'M'},{''},{''},{''},{''},{''},{''},{''}],{STRING1 Gender});
d2 := dataset([{'''},{''},{''},{''},{''},{''},{''},{''},{''},{''}],{STRING1 Gender});
OUTPUT(Field_Population(d1,Gender,''));
OUTPUT(Field_Population(d2,Gender,''));
```

This example demonstrates use of the **LOCAL** keyword to prevent name clashes with external definitions within the same visibility scope as the FUNCTIONMACRO.
numPlus := 'this creates a syntax error without LOCAL in the FUNCTIONMACRO';
AddOne(num) := FUNCTIONMACRO
   LOCAL numPlus := num + 1;  //LOCAL required here
   RETURN numPlus;
ENDMACRO;

AddTwo(num) := FUNCTIONMACRO
   LOCAL numPlus := num + 2;  //LOCAL required here
   RETURN numPlus;
ENDMACRO;

See Also: FUNCTION Structure, MACRO Structure
INTERFACE Structure

\[
\text{interfacename } [ ( \text{parameters} ) ] := \text{INTERFACE } [ ( \text{inherit} ) ]
\]

\[
\text{members;}
\]

\[
\text{END;}
\]

<table>
<thead>
<tr>
<th>interfacename</th>
<th>The ECL definition name of the interface.</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameters</td>
<td>Optional. The input parameters to the interface.</td>
</tr>
<tr>
<td>inherit</td>
<td>Optional. A comma-delimited list of INTERFACE structures whose members to inherit. This may not be a passed parameter. Multiple inherited interfaces may contain attributes with the same name if they are the same type and receive the same parameters, but if those inherited members have different values defined for them, the conflict must be resolved by overriding that member in the current instance.</td>
</tr>
<tr>
<td>members</td>
<td>Definitions, which may be EXPORTed or SHARED. These may be similar to fields defined in a RECORD structure where only the type and name are defined--the expression that defines the value may be left off (except in some cases where the expression itself defines the type of definition, like TRANSFORM structures). If no default value is defined for a member, any MODULE derived from the INTERFACE must define a value for that member before that MODULE can be used. These may not include other INTERFACE or abstract MODULE structures.</td>
</tr>
</tbody>
</table>

The INTERFACE structure defines a structured block of related members that may be passed as a single parameter to complex queries, instead of passing each attribute individually. It is similar to a MODULE structure with the VIRTUAL option, except errors are given for private (not SHARED or EXPORTed) member definitions.

An INTERFACE is an abstract structure--a concrete instance must be defined before it can be used in a query. A MODULE structure that inherits the INTERFACE and defines the values for the members creates the concrete instance for use by the query.

Example:

```ecl
HeaderRec := RECORD
  UNSIGNED4 RecID;
  STRING20 company;
  STRING25 address;
  STRING25 city;
  STRING2 state;
  STRING5 zip;
END;
HeaderFile := DATASET([{1,'ABC Co','123 Main','Boca Raton','FL','33487'},
  {2,'XYZ Co','456 High','Jackson','MI','49202'},
  {3,'ABC Co','619 Eaton','Jackson','MI','49202'},
  {4,'XYZ Co','999 Yamato','Boca Raton','FL','33487'},
  {5,'Joes Eats','666 Slippery Lane','Nether','SC','12345'}],HeaderRec);

//define an interface
IHeaderFileSearch := INTERFACE
  EXPORT STRING20 company_val;
  EXPORT STRING2 state_val;
  EXPORT STRING25 city_val := '';
END;
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
//define a function that uses that interface  
FetchAddress(IHeaderFileSearch opts) := FUNCTION

    //define passed values tests
    CompanyPassed := opts.company_val <> '';  
    StatePassed := opts.state_val <> '';  
    CityPassed := opts.city_val <> '';  

    //define passed value filters
    NFilter := HeaderFile.Company = opts.company_val;  
    SFilter := HeaderFile.State = opts.state_val;  
    CFilter := HeaderFile.City = opts.city_val;  

    //define the actual filter to use based on the passed values
    filter := MAP(CompanyPassed AND StatePassed AND CityPassed => NFilter AND SFilter AND CFilter, 
                  CompanyPassed AND StatePassed => NFilter AND SFilter, 
                  CompanyPassed AND CityPassed => NFilter AND CFilter, 
                  StatePassed AND CityPassed => SFilter AND CFilter, 
                  CompanyPassed => NFilter, 
                  StatePassed => SFilter, 
                  CityPassed => CFilter, 
                  TRUE);  
    RETURN HeaderFile(filter);  
END;  

//*****************************************************************  
//then you can use the interface

InRec := {HeaderRec AND NOT [RecID,Address,Zip]};  

//this MODULE creates a concrete instance  
BatchHeaderSearch(InRec l) := MODULE(IHeaderFileSearch)  
    EXPORT STRING20 company_val := l.company;  
    EXPORT STRING2 state_val := l.state;  
    EXPORT STRING25 city_val := l.city;  
END;  

//that can be used like this  
FetchAddress(BatchHeaderSearch(ROW({'ABC Co','',''},InRec)));  

//or we can define an input dataset

InFile := DATASET([{'ABC Co','Boca Raton','FL'},  
                    {'XYZ Co','Jackson','MI'},  
                    {'ABC Co','',''},  
                    {'XYZ Co','',''},  
                    {'Joes Eats','',''}  
                ],InRec);  

//and an output nested child structure  
HeaderRecs := RECORD  
    UNSIGNED4 Pass;  
    DATASET(HeaderRec) Headers;  
END;  

//and allow PROJECT to run the query once for each record in InFile

HeaderRecs XF(InRec L, INTEGER C) := TRANSFORM  
    SELF.Pass := C;  
    SELF.Headers := FetchAddress(BatchHeaderSearch(L));  
END;  
batchHeaderLookup := PROJECT(InFile,XF(LEFT,COUNTER));
batchHeaderLookup;

See Also: MODULE Structure, LIBRARY
**MACRO Structure**

\[ \text{resulttype} \] macroname ( parameterlist ) := MACRO

tokenstream;

ENDMACRO;

<table>
<thead>
<tr>
<th>resulttype</th>
<th>Optional. The result type of the macro. The only valid type is DATASET. If omitted and the tokenstream contains no Attribute definitions, then the macro is treated as returning a value (typically INTEGER or STRING).</th>
</tr>
</thead>
<tbody>
<tr>
<td>macroname</td>
<td>The name of the function the MACRO structure defines.</td>
</tr>
<tr>
<td>parameterlist</td>
<td>A comma separated list of names (tokens) of the parameters that will be passed to the macro. These names are used in the tokenstream to indicate where the passed parameters are substituted when the macro is used. Value types for these parameters are not allowed, but default values may be specified as string constants.</td>
</tr>
<tr>
<td>tokenstream</td>
<td>The Attribute definitions or Actions that the macro will perform.</td>
</tr>
</tbody>
</table>

The MACRO structure makes it possible to create a function without knowing the value types of the parameters that will eventually be passed to it. The most common use would be performing functions upon arbitrary datasets.

A macro behaves as if you had typed the tokenstream into the exact position you use it, using lexical substitution—the tokens defined in the parameterlist are substituted everywhere they appear in the tokenstream by the text passed to the macro. This makes it entirely possible to write a valid MACRO definition that could be called with a set of parameters that result in obscure compile time errors.

There are two basic type of macros: Value or Attribute. A Value macro does not contain any Attribute definitions, and may therefore be used wherever the value type it will generate would be appropriate to use. An Attribute macro does contain Attribute definitions (detected by the presence of the := in the tokenstream) and may therefore only be used where an Attribute definition is valid (a line by itself) and one item in the parameterlist should generally name the Attribute to be used to contain the result of the macro (so any code following the macro call can make use of the result).

Example:

```ecl
// This is a DATASET Value macro that results in a crosstab
DATASET CrossTab(File,X,Y) := MACRO
  TABLE(File,{X, Y, COUNT(GROUP)}),X,Y)
ENDMACRO;
// and would be used something like this:
OUTPUT(CrossTab(Person,person.per_st,Person.per_sex))
// this macro usage is the equivalent of:
// OUTPUT(TABLE(Person,{person.per_st,Person.per_sex,COUNT(GROUP)}),
//   person.per_st,Person.per_sex)
//The advantage of using this macro is that it can be re-used to
// produce another cross-tab without recoding
// The following macro takes a LeftFile and looks up a field of it in
// the RightFile and then sets a field in the LeftFile indicating if
// the lookup worked.
IsThere(OutFile ,RecType,LeftFile,RightFile,LinkId ,SetField ) := MACRO
  RecType Trans(RecType L, RecType R) := TRANSFORM
    SELF.SetField := IF(NOT R.LinkId,0,1);
    SELF := L;
END;
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
OutFile := JOIN(LeftFile,
    RightFile,
    LEFT.LinkId=RIGHT.LinkId,
    Trans(LEFT,RIGHT), LEFT OUTER);
ENDMACRO;

// and would be used something like this:
MyRec := RECORD
    Person.per_cid;
    Person.per_st;
    Person.per_sex;
    Flag:=FALSE;
END;
MyTable1 := TABLE(Person(per_first_name[1]='R'),MyRec);
MyTable2 := TABLE(Person(per_first_name[1]='R',per_sex='F'),MyRec);

IsThere(MyOutTable,MyRec,MyTable1,MyTable2,per_cid,Flag)
    // This macro call generates the following code:
    // MyRec Trans(MyRec L, MyRec R) := TRANSFORM
    // SELF.Flag := IF(NOT R.per_cid ,0,1);
    // SELF := L;
    // END;
    // MyOutTable := JOIN(MyTable1,
    // MyTable2,
    // LEFT.per_cid=RIGHT.per_cid,
    // Trans(LEFT,RIGHT),
    // LEFT OUTER );

OUTPUT(MyOutTable);

// This macro has defaults for its second and third parameters
MyMac(FirstParm,yParm='22',zParm='42') := MACRO
    FirstParm := yParm + zParm;
ENDMACRO;

// and would be used something like this:
MyMac(Fred)
    // This macro call generates the following code:
    // Fred := 22 + 42;
    //*****************************************************************************
    //This macro uses #EXPAND
MAC_join(attrname, leftDS, rightDS, linkflags) := MACRO
    attrname := JOIN(leftDS,rightDS,#EXPAND(linkflags));
ENDMACRO;
MAC_join(J1,People,Property,'LEFT.ID=RIGHT.PeopleID,LEFT OUTER')
    //expands out to:
    // J1 := JOIN(People,Property,LEFT.ID=RIGHT.PeopleID,LEFT OUTER);

See Also: TRANSFORM Structure, RECORD Structure, #UNIQUENAME, #EXPAND
MODULE Structure


members;

END;

<table>
<thead>
<tr>
<th><strong>modulename</strong></th>
<th>The ECL definition name of the module.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>parameters</strong></td>
<td>Optional. The parameters to make available to all the definitions.</td>
</tr>
<tr>
<td><strong>inherit</strong></td>
<td>A comma-delimited list of INTERFACE or abstract MODULE structures on which to base this instance. The current instance inherits all the members from the base structures. This may not be a passed parameter.</td>
</tr>
<tr>
<td><strong>members</strong></td>
<td>The definitions that comprise the module. These definitions may receive parameters, may include actions (such as OUTPUT), and may use the EXPORT or SHARED scope types. These may not include INTERFACE or abstract MODULEs (see below). If the LIBRARY option is specified, the definitions must exactly implement the EXPORTed members of the interface.</td>
</tr>
<tr>
<td><strong>VIRTUAL</strong></td>
<td>Optional. Specifies the MODULE defines an abstract interface whose definitions do not require values to be defined for them.</td>
</tr>
<tr>
<td><strong>LIBRARY</strong></td>
<td>Optional. Specifies the MODULE implements a query library interface definition.</td>
</tr>
<tr>
<td><strong>interface</strong></td>
<td>Specifies the INTERFACE that defines the parameters passed to the query library. The parameters passed to the MODULE must exactly match the parameters passed to the specified interface.</td>
</tr>
<tr>
<td><strong>FORWARD</strong></td>
<td>Optional. Delays processing of definitions until they are used. Adding ,FORWARD to a MODULE delays processing of definitions within the module until they are used. This has two main effects: It prevents pulling in dependencies for definitions that are never used and it allows earlier definitions to refer to later definitions. Note: Circular references are still illegal.</td>
</tr>
</tbody>
</table>

The MODULE structure is a container that allows you to group related definitions. The parameters passed to the MODULE are shared by all the related members definitions. This is similar to the FUNCTION structure except that there is no RETURN.

**Definition Visibility Rules**

The scoping rules for the members are the same as those previously described in the Definition Visibility discussion:

- Local definitions are visible only through the next EXPORT or SHARED definition (including members of the nested MODULE structure, if the next EXPORT or SHARED definition is a MODULE).
- SHARED definitions are visible to all subsequent definitions in the structure (including members of any nested MODULE structures) but not outside of it.
- EXPORT definitions are visible within the MODULE structure (including members of any subsequent nested MODULE structures) and outside of it.

Any EXPORT members may be referenced using an additional level of standard object.property syntax. For example, assuming the EXPORT MyModuleStructure MODULE structure is contained in an ECL Reposi-
tory module named MyModule and that it contains an EXPORT member named MyDefinition, you would reference that definition as MyModule.MyModuleStructure.MyDefinition:

```ecl
MyMod := MODULE
  SHARED x := 88;
  y := 42;
  EXPORT InMod := MODULE //nested MODULE
    EXPORT Val1 := x + 10;
    EXPORT Val2 := y + 10;
  END;
END;
```

```ecl
OUTPUT(MyMod.InMod.Val1);
OUTPUT(MyMod.InMod.Val2);
```

### MODULE Side-Effect Actions

Side-effect Actions are allowed in the MODULE only by using the WHEN function, as in this example:

```ecl
//An Example with a side-effect action
EXPORT customerNames := MODULE
  EXPORT Layout := RECORD
    STRING20 surname;
    STRING10 forename;
    INTEGER2 age := 25;
  END;
  Act := OUTPUT('customer file used by user <x>');</n
  EXPORT File := WHEN(DATASET([{'x','y',22}]),Layout),Act);
END;
```

```ecl
BOOLEAN doIt := TRUE : STORED('doIt');
IF (doIt, OUTPUT(customerNames.File));
//This code produces two results: the dataset, and the string
```

### Concrete vs. Abstract (VIRTUAL) Modules

A MODULE may contain a mixture of VIRTUAL and non-VIRTUAL members. The rules are:

- **ALL members** are VIRTUAL if the MODULE has the VIRTUAL option or is an INTERFACE
- A **member** is VIRTUAL if it is declared using the EXPORT VIRTUAL or SHARED VIRTUAL keywords
- A **member** is VIRTUAL if the definition of the same name in the **inherited** module is VIRTUAL.
- Some **members** can never be virtual -- RECORD structures.

All EXPORTed and SHARED members of an **inherited** abstract module can be overridden by re-defining them in the current instance, whether that current instance is abstract or concrete. Overridden definitions must exactly match the type and parameters of the **inherited members**. Multiple **inherited** interfaces may contain definitions with the same name if they are the same type and receive the same parameters, but if those **inherited members** have different values defined for them, the conflict must be resolved by overriding that **member** in the current instance.

### LIBRARY Modules

A MODULE with the LIBRARY option defines a related set of functions meant to be used as a query library (see the LIBRARY function and BUILD action discussions). There are several restrictions on what may be included in a query library. They are:

- It may not contain side-effect actions (like OUTPUT or BUILD)
It may not contain definitions with workflow services attached to them (such as PERSIST, STORED, SUCCESS, etc.)

It may only EXPORT:

- Dataset/recordset definitions
- Dataset/recordset definitions
- Dataset/recordset definitions
- Dataset/recordset definitions
- Dataset/recordset definitions
- Dataset/recordset definitions
- Dataset/recordset definitions
- Dataset/recordset definitions
- Dataset/recordset definitions
- Dataset/recordset definitions
- Single-valued and Boolean definitions

And may NOT export:

- Actions (like OUTPUT or BUILD)
- TRANSFORM functions
- Other MODULE structures
- MACRO definitions

Example:

```ecl
namesRecord := RECORD
  STRING20 surname;
  STRING10 forename;
  INTEGER2 age := 25;
END;
namesTable := DATASET([{'Smith','Sue',72},
                        {'Jones','Joe',32},
                        {'Jones','Fred',82}],namesRecord);

filterDataset1(STRING search, BOOLEAN onlyOldies) := MODULE
  f := namesTable; //local to the "g" definition
  SHARED g := IF (onlyOldies, f(age >= 65), f);
  //SHARED = visible only within the structure
  EXPORT included := g(surname = search);
  EXPORT excluded := g(surname <> search);
  //EXPORT = visible outside the structure
END;
filtered1 := filterDataset1('Smith', TRUE);
OUTPUT(filtered1.included,,NAMED('Included1'));
OUTPUT(filtered1.excluded,,NAMED('Excluded1'));

//same result, different coding style:
filterDataset2(BOOLEAN onlyOldies) := MODULE
  f := namesTable;
  SHARED g := IF (onlyOldies, f(age >= 65), f);
  EXPORT included(STRING search) := g(surname = search);
  EXPORT excluded(STRING search) := g(surname <> search);
END;
filtered2 := filterDataset2(TRUE);
OUTPUT(filtered2.included('Smith'),,NAMED('Included2'));
OUTPUT(filtered2.excluded('Smith'),,NAMED('Excluded2'));

//VIRTUAL examples
Mod1 := MODULE,VIRTUAL //a fully abstract module
  EXPORT val := 1;
  EXPORT func(INTEGER sc) := val * sc;
END;

Mod2 := MODULE(Mod1) //instance
  EXPORT val := 3; //a concrete member, overriding default value
```
//while func remains abstract
END;

Mod3 := MODULE(Mod1) //a fully concete instance
   EXPORT func(INTEGER sc) := val + sc; //overrides inherited func
END;
OUTPUT(Mod2.func(5)); //result is 15
OUTPUT(Mod3.func(5)); //result is 6

//FORWARD example
MyModule := MODULE, FORWARD
   EXPORT INTEGER foo := bar; //forward reference
   EXPORT INTEGER bar := 42;
END;
OUTPUT(MyModule.foo);

See Also: FUNCTION Structure, Definition Visibility, INTERFACE Structure, LIBRARY, BUILD

**TRANSFORM Structure**

```
resulttype funcname ( parameterlist ) := TRANSFORM [ , SKIP( condition )]
[ locals ]
SELF.outfield := transformation;
END;

TRANSFORM( resulttype, assignments )

TRANSFORM( datarow )
```

<table>
<thead>
<tr>
<th>resulttype</th>
<th>The name of a RECORD structure Attribute that specifies the output format of the function. You may use TYPEOF here to specify a dataset. Any implicit relationality of the input dataset is not inherited.</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcname</td>
<td>The name of the function the TRANSFORM structure defines.</td>
</tr>
<tr>
<td>parameterlist</td>
<td>A comma separated list of the value types and labels of the parameters that will be passed to the TRANSFORM function. These are usually the dataset records or COUNTER parameters but are not limited to those.</td>
</tr>
<tr>
<td>SKIP</td>
<td>Optional. Specifies the condition under which the TRANSFORM function operation is skipped.</td>
</tr>
<tr>
<td>condition</td>
<td>A logical expression defining under what circumstances the TRANSFORM operation does not occur. This may use data from the parameterlist in the same manner as a transformation expression.</td>
</tr>
<tr>
<td>locals</td>
<td>Optional. Definitions of local Attributes useful within the TRANSFORM function. These may be defined to receive parameters and may use any parameters passed to the TRANSFORM.</td>
</tr>
<tr>
<td>SELF</td>
<td>Specifies the resulting output recordset from the TRANSFORM.</td>
</tr>
<tr>
<td>outfield</td>
<td>The name of a field in the resulttype structure.</td>
</tr>
<tr>
<td>transformation</td>
<td>An expression specifying how to produce the value for the outfield. This may include other TRANSFORM function operations (nested transforms).</td>
</tr>
</tbody>
</table>
The **TRANSFORM** structure makes operations that must be performed on entire datasets (such as a JOIN) and any iterative type of record processing (PROJECT, ITERATE, etc.), possible. A TRANSFORM defines the specific operations that must occur on a record-by-record basis. It defines the function that is called each time the operation that uses the TRANSFORM needs to process record(s). One TRANSFORM function may be defined in terms of another, and they may be nested.

The TRANSFORM structure specifies exactly how each field in the output record set is to receive its value. That result value may simply be the value of a field in an input record set, or it may be the result of some complex calculation or conditional expression evaluation.

The TRANSFORM structure itself is a generic tool; each operation that uses a TRANSFORM function defines what its TRANSFORM needs to receive and what basic functionality it should provide. Therefore, the real key to understanding TRANSFORM structures is in understanding how it is used by the calling function -- each function that uses a TRANSFORM documents the type of TRANSFORM required to accomplish the goal, although the TRANSFORM itself may also provide extra functionality and receive extra parameters beyond those required by the operation itself.

The SKIP option specifies the condition that results in no output from that iteration of the TRANSFORM. However, COUNTER values are incremented even when SKIP eliminates generating the current record.

### Transformation Attribute Definitions

The attribute definitions inside the TRANSFORM structure are used to convert the data passed in as parameters to the output format. Every field in the resulttype record layout must be fully defined in the TRANSFORM. You can explicitly define each field, using the `SELF.outfield := transformation;` expression, or you can use one of these shortcuts:

```plaintext
SELF : [ ];
```

Clears all fields in the resulttype output that have not previously been defined in the transform function, while this form:

```plaintext
SELF.outfield := [] ; //the outfield names a child DATASET in
                     // the resulttype RECORD Structure
```

Clears only the child fields in the `outfield`, and this form:

```plaintext
SELF := label; //the label names a RECORD structure parameter
             // in the parameterlist
```

Defines the output for each field in the resulttype output format that has not previously been defined as coming from the `label` parameter's matching named field.

You may also define local attributes inside the TRANSFORM structure to better organize the code. These local attributes may receive parameters.

### TRANSFORM Functions

This form of TRANSFORM must be terminated by the `END` keyword. The resulttype must be specified, and the function itself takes parameters in the parameterlist. These parameters are typically RECORD structures, but may be any type of parameter depending upon the type of TRANSFORM function the using function expects to call. The exact form a TRANSFORM function must take is always directly associated with the operation that uses it.
Example:

Ages := RECORD
  AgedRecs.id;
  AgedRecs.id1;
  AgedRecs.id2;
END;
SequencedAges := RECORD
  Ages;
  INTEGER4 Sequence := 0;
END;
SequencedAges AddSequence(AgedRecs L, INTEGER C) :=
  TRANSFORM, SKIP(C % 2 = 0) //skip even recs
  INTEGER1 rangex(UNSIGNED4 divisor) := (l.id DIV divisor) % 100;
  SELF.id1 := rangex(10000);
  SELF.id2 := rangex(100);
  SELF.Sequence := C;
  SELF := L;
END;
SequencedAgedRecs := PROJECT(AgedRecs, AddSequence(LEFT, COUNTER));

//Example of defining a TRANSFORM function in terms of another
namesIdRecord assignId(namesRecord l, UNSIGNED value) := TRANSFORM
  SELF.id := value;
  SELF := l;
END;
assignId1(namesRecord l) := assignId(l, 1);
  //creates an assignId1 TRANSFORM that uses assignId
assignId2(namesRecord l) := assignId(l, 2);
  //creates an assignId2 TRANSFORM that uses assignId

**Inline TRANSFORMs**

This form of TRANSFORM is used in-line within the operation that uses it. The *resulttype* must be specified along with all the *assignments*. This form is mainly for use where the transform *assignments* are trivial (such as SELF := LEFT;).

Example:

namesIdRecord assignId(namesRecord L) := TRANSFORM
  SELF := L; //more like-named fields across
  SELF := [] ; //clear all other fields
END;

projected1 := PROJECT(namesTable, assignId(LEFT));
projected2 := PROJECT(namesTable, TRANSFORM(namesIdRecord, SELF := LEFT; SELF := []));
//projected1 and projected2 do the same thing

**Shorthand Inline TRANSFORMs**

This form of TRANSFORM is a shorthand version of Inline TRANSFORMs. In this form,

TRANSFORM(LEFT)

is directly equivalent to

TRANSFORM(RECORDOF(LEFT), SELF := LEFT)

Example:
namesIdRecord assignId(namesRecord L) := TRANSFORM
  SELF := L; //move like-named fields across
END;
projected1 := PROJECT(namesTable, assignId(LEFT));
projected2 := PROJECT(namesTable, TRANSFORM(namesIdRecord,
  SELF := LEFT));
projected3 := PROJECT(namesTable, TRANSFORM(LEFT));
//projected1, projected2, and projected3 all do the same thing

See Also: RECORD Structure, RECORDOF, TYPEOF, JOIN, PROJECT, ITERATE, ROLLUP, NORMALIZE, DENORMALIZE, FETCH, PARSE, ROW
Built-in Functions and Actions
ABS

ABS(expression)

expression  The value (REAL or INTEGER) for which to return the absolute value.

Return:  ABS returns a single value of the same type as the expression.

The ABS function returns the absolute value of the expression (always a non-negative number).

Example:

AbsVal1 := ABS(1); // returns 1
AbsVal2 := ABS(-1); // returns 1
OUTPUT(AbsVal1);
OUTPUT(AbsVal2);
ACOS

ACOS(cosine)

<table>
<thead>
<tr>
<th>cosine</th>
<th>The REAL cosine value for which to find the arccosine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return</td>
<td>ACOS returns a single REAL value.</td>
</tr>
</tbody>
</table>

The ACOS function returns the arccosine (inverse) of the cosine, in radians.

Example:

```plaintext
Deg2Rad := 0.0174532925199; //number of radians in a degree
cosineAngle := 0.75;
acos(cosineAngle) * Deg2Rad;
```

See Also: COS, SIN, TAN, ASIN, ATAN, COSH, SINH, TANH
AGGREGATE

AGGREGATE( recordset, resultrec, maintransform [ , mergetransform (RIGHT1,RIGHT2) | groupingfields ] [ , LOCAL | FEW | MANY ] [ , UNORDERED | ORDERED( bool ) ] [ , STABLE | UNSTABLE ] [ , PARALLEL [ ( numthreads ) ] ] [ , ALGORITHM( name ) ] )

recordset The set of records to process.

resultrec The RECORD structure of the result record set.

maintransform The TRANSFORM function to call for each matching pair of records in the recordset. This is implicitly a local operation on each node.

mergetransform Optional. The TRANSFORM function to call to globally merge the result records from the maintransform. If omitted, the compiler will attempt to deduce the merge from the maintransform.

groupingfields Optional. A comma-delimited list of fields in the recordset to group by. Each field must be prefaced with the keyword LEFT. If omitted, then all records match.

LOCAL Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE. Valid only if the mergetransform is omitted.

FEW Optional. Indicates that the expression will result in fewer than 10,000 records. This allows optimization to produce a significantly faster result.

MANY Optional. Indicates that the expression will result in more than 10,000 records.

UNORDERED Optional. Specifies the output record order is not significant.

ORDERED Specifies the significance of the output record order.

bool When False, specifies the output record order is not significant. When True, specifies the default output record order.

STABLE Optional. Specifies the input record order is significant.

UNSTABLE Optional. Specifies the input record order is not significant.

PARALLEL Optional. Try to evaluate this activity in parallel.

numthreads Optional. Try to evaluate this activity using numthreads threads.

ALGORITHM Optional. Override the algorithm used for this activity.

name The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

Return: AGGREGATE returns a record set.

The AGGREGATE function is similar to ROLLUP except its output format does not need to match the input format. It also has similarity to TABLE in that the groupingfields (if present) determine the matching records such that you will get one result for each unique value of the groupingfields. The input recordset does not need to have been sorted by the groupingfields.

The operation is implicitly local, in that the maintransform is called to process records locally on each node, and the result records on each node are then merged to produce the global result.
TRANSFORM Function Requirements - AGGREGATE

The *maintransform* must take at least two parameters: a LEFT record of the same format as the input *recordset* and a RIGHT record of the same format as the *resultrec*. The format of the resulting record set must be the *resultrec*. LEFT refers to the next input record and RIGHT the result of the previous transform.

The *mergetransform* must take at least two parameters: RIGHT1 and RIGHT2 records of the same format as the *resultrec*. The format of the resulting record set must be the *resultrec*. RIGHT1 refers to the result of the *maintransform* on one node and RIGHT2 the result of the *maintransform* on another.

The *mergetransform* is generated for expressions of the form:

```
SELF.x := <RIGHT.x <op> f(LEFT)
SELF.x := f(LEFT)  <op> RIGHT.x
```

where the `<op>` is: MAX, MIN, SUM, +, &, |, ^, *

How AGGREGATE Works

In the *maintransform*, LEFT refers to the next input record and RIGHT the result of the previous transform.

There are 4 interesting cases:

(a) If no records match (and the operation isn't grouped), the output is a single record with all the fields set to blank values.

(b) If a single record matches, the first record that matches calls the *maintransform* as you would expect.

(c) If multiple records match on a single node, subsequent records that match call the *maintransform* but any field expression in the *maintransform* that does not reference the RIGHT record is not processed. Therefore the value for that field is set by the first matching record matched instead of the last.

(d) If multiple records match on multiple nodes, then step (c) performs on each node, and then the summary records are merged. This requires a *mergetransform* that takes two records of type RIGHT. Whenever possible the code generator tries to deduce the *mergetransform* from the *maintransform*. If it can't, then the user will need to specify one.

//Example 1: Produce a list of box contents by concatenating a string:
IMPORT Std;
inRec := RECORD
  UNSIGNED box;
  STRING text{MAXLENGTH(100)};
END;
inds := DATASET([
  {1,'Fred1'}, {1,'Freddy1'}, {1,'FredJon1'},
  {3,'Fred3'}, {3,'Freddy3'}, {3,'FredJon3'},
  {4,'Fred4'}, {4,'Freddy4'}, {4,'FredJon4'},
  {2,'Freddi'}, {2,'Fredrik'}], inRec,DISTRIBUTED);
outRec := RECORD
  UNSIGNED box;
  STRING contents{MAXLENGTH(200)};
END;
outRec t1(inds l, outRec r) := TRANSFORM
  SELF.box := l.box;
  SELF.contents:= r.contents +IF(r.contents <> '', ',', '') +l.text +'-' +(Std.System.ThorLib.Node()+1);
END;
outRec t2(outRec r1, outRec r2) := TRANSFORM
  SELF.box := r1.box;
  SELF.contents := r1.contents +'::' + r2.contents;

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
END;
OUTPUT(AGGREGATE(inds, outRec, t1(LEFT, RIGHT), t2(RIGHT1, RIGHT2), LEFT.box));
//because there is a "group by" field, this will never call the second TRANSFORM
//because "group by" puts all grouped recs on a single node
//and it produces one result rec for each unique "group by" value

OUTPUT(AGGREGATE(inds, outRec, t1(LEFT, RIGHT), t2(RIGHT1, RIGHT2)));
//without the "group by" field, this calls the second TRANSFORM on a multi-node cluster
//and the second TRANSFORM produces a single result record after merging the results from
//each node

//Example 2: A PIGMIX style grouping operation:
inRecord := RECORD
  UNSIGNED box;
  STRING text{MAXLENGTH(10)};
END;
inTable := DATASET(
{1,'Fred'},{1,'Freddy'},
{2,'Freddi'},{3,'Fredrik'},{1,'FredJon'}), inRecord);

outRecord2 := RECORD
  UNSIGNED box;
  DATASET(inRecord) items;
END;
outRecord2 t3(inRecord l, outRecord2 r) := TRANSFORM
  SELF.box := l.box;
  SELF.items := r.items + l;
END;
OUTPUT(AGGREGATE(inTable, outRecord2, t3(LEFT, RIGHT), LEFT.box));

See Also: TRANSFORM Structure, RECORD Structure, ROLLUP, TABLE
ALLNODES

**ALLNODES**(operation)

<table>
<thead>
<tr>
<th>operation</th>
<th>The name of an attribute or in-line code that results in a DATASET or INDEX.</th>
</tr>
</thead>
</table>

Return: ALLNODES returns a record set or index.

The ALLNODES function specifies that the *operation* is performed on all nodes in parallel. **Available for use only in Roxie.**

Example:

```ecl
ds := ALLNODES(JOIN(SomeData,LOCAL(SomeIndex), LEFT.ID = RIGHT.ID));
```

See Also: THISNODE, LOCAL, NOLOCAL
**APPLY**

\[
\text{APPLY} \left( \text{dataset, actionlist} [\ , \ \text{BEFORE} (\text{actionlist})] [\ , \ \text{AFTER} (\text{actionlist} [\ , \ \text{UNORDERED} | \ \text{ORDERED} (\text{bool})] [\ , \ \text{STABLE} | \ \text{UNSTABLE}] [\ , \ \text{PARALLEL} [\ (\text{numthreads})]] [\ , \ \text{ALGORITHM} (\text{name})])] \right)
\]

| attrname | Optional. The action name, which turns the action into an attribute definition, therefore not executed until the attrname is used as an action. |
| dataset | The set of records to apply the action to. This must be the name of a physical dataset of a type that supports this operation. |
| actionlist | A comma-delimited list of the operations to perform on the dataset. Typically, this is an external service (see SERVICE Structure). This may not be an OUTPUT or any function that triggers a child query. |
| BEFORE | Specifies executing the enclosed actionlist before the first dataset row is processed. Not yet implemented in Thor, valid only in hthor and Roxie. |
| AFTER | Specifies executing the enclosed actionlist after the last dataset row is processed. Not yet implemented in Thor, valid only in hthor and Roxie. |
| UNORDERED | Optional. Specifies the output record order is not significant. |
| ORDERED | Specifies the significance of the output record order. |
| bool | When False, specifies the output record order is not significant. When True, specifies the default output record order. |
| STABLE | Optional. Specifies the input record order is significant. |
| UNSTABLE | Optional. Specifies the input record order is not significant. |
| PARALLEL | Optional. Try to evaluate this activity in parallel. |
| numthreads | Optional. Try to evaluate this activity using numthreads threads. |
| ALGORITHM | Optional. Override the algorithm used for this activity. |
| name | The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options. |

The APPLY action performs all the specified actions in the actionlist on each record of the nominated dataset. The actions execute in the order they appear in the actionlist.

Example:

```ecl
EXPORT x := SERVICE
  \text{echo(const string src):library='myfuncs',entrypoint='rtlEcho';}
END;
APPLY(person,x.echo(last_name + ':' + first_name));
// concatenate each person's lastname and firstname and echo it
```

See Also: SERVICE Structure, DATASET
# ASCII

\[
\text{ASCII}(\text{recordset}, \text{UNORDERED} | \text{ORDERED}(\text{bool}) |, \text{STABLE} | \text{UNSTABLE} |, \text{PARALLEL} [(\text{numthreads}) |, \text{ALGORITHM}(\text{name})])
\]

- **recordset**: The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.

- **UNORDERED**: Optional. Specifies the output record order is not significant.

- **ORDERED**: Specifies the significance of the output record order.

- **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.

- **STABLE**: Optional. Specifies the input record order is significant.

- **UNSTABLE**: Optional. Specifies the input record order is not significant.

- **PARALLEL**: Optional. Try to evaluate this activity in parallel.

- **numthreads**: Optional. Try to evaluate this activity using \textit{numthreads} threads.

- **ALGORITHM**: Optional. Override the algorithm used for this activity.

- **name**: The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

**Return:** ASCII returns a set of records.

The **ASCII** function returns the \textit{recordset} with all STRING fields translated from EBCDIC to ASCII.

**Example:**

\[
\text{AsciiRecs := ASCII(SomeEBCDICInput);}\]

**See Also:** EBCDIC
ASIN

**ASIN**

<table>
<thead>
<tr>
<th><strong>sine</strong></th>
<th>The REAL sine value for which to find the arcsine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>ASIN returns a single REAL value.</td>
</tr>
</tbody>
</table>

The **ASIN** function returns the arcsine (inverse) of the *sine*, in radians.

Example:

```plaintext
Rad2Deg := 57.295779513082; //number of degrees in a radian
SineAngle := .25;
ArcSine := ASIN(SineAngle)*Rad2Deg;
OUTPUT(ArcSine); // degrees
```

See Also: ACOS, COS, SIN, TAN, ATAN, COSH, SINH, TANH
# ASSERT

**ASSERT**

<table>
<thead>
<tr>
<th>condition</th>
<th>The logical expression that should be always be true.</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>Optional. The error to display in the workunit. If omitted, a message is generated from the approximate location in the code and the condition being checked.</td>
</tr>
<tr>
<td>FAIL</td>
<td>Optional. Specifies an exception is generated, immediately terminating the workunit.</td>
</tr>
<tr>
<td>CONST</td>
<td>Optional. Specifies the condition is evaluated during code generation.</td>
</tr>
<tr>
<td>recset</td>
<td>The set of records for which to check the condition against each record.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
<tr>
<td>assertlist</td>
<td>A comma-delimited list of ASSERTs of the first form, used to check multiple conditions against each record in the recset.</td>
</tr>
</tbody>
</table>

The **ASSERT** action evaluates the **condition**, and if false, posts the **message** in the workunit. The workunit terminates immediately if the **FAIL** option is present.

Form one is the scalar form, evaluating the **condition** once. Form two evaluates the **condition** once for each record in the **recset**. Form three is a variant of form two that nests multiple form one ASSERTs so that each condition is checked against each record in the **recset**.

Example:

```ecl
val1 := 1;
val2 := 1;
val3 := 2;
val4 := 2 : STORED('val4');
ASSERT(val1 = val2);
ASSERT(val1 = val2, 'Abc1');
ASSERT(val1 = val3);
ASSERT(val1 = val3, 'Abc2');
```
ECL Language Reference
Built-in Functions and Actions

ASSERT(val1 = val4);
ASSERT(val1 = val4, 'Abc3');
ds := DATASET([1,2], {INTEGER val1}) : GLOBAL;
// global stops advanced constant folding (if ever done)
ds1 := ASSERT(ds, val1 = val2);
ds2 := ASSERT(ds1, val1 = val2, 'Abc4');
ds3 := ASSERT(ds2, val1 = val3);
ds4 := ASSERT(ds3, val1 = val3, 'Abc5');
ds5 := ASSERT(ds4, val1 = val4);
ds6 := ASSERT(ds5, val1 = val4, 'Abc6');
OUTPUT(ds6);
ds7 := ASSERT(ds(val1 != 99),
    ASSERT(val1 = val2),
    ASSERT(val1 = val2, 'Abc7'),
    ASSERT(val1 = val3),
    ASSERT(val1 = val3, 'Abc8'),
    ASSERT(val1 = val4),
    ASSERT(val1 = val4, 'Abc9'));
OUTPUT(ds7);
rec := RECORD
    INTEGER val1;
    STRING text;
END;
rec t(ds l) := TRANSFORM
    ASSERT(l.val1 <= 3);
    SELF.text := CASE(l.val1, 1=>'One', 2=>'Two', 3=>'Three', 'Zero');
    SELF := l;
END;
OUTPUT(PROJECT(ds, t(LEFT)));

See Also: FAIL, ERROR
ASSTRING

**ASSTRING**(bitmap)

<table>
<thead>
<tr>
<th>bitmap</th>
<th>The value to treat as a string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>ASSTRING returns a single STRING value.</td>
</tr>
</tbody>
</table>

The **ASSTRING** function returns the *bitmap* as a string. This is equivalent to **TRANSFER**(bitmap,STRINGn) where *n* is the same number of bytes as the data in the *bitmap*.

**Example:**

```
INTEGER1 MyInt := 65; //MyInt is an integer whose value is 65
MyVal1 := ASSTRING(MyInt); //MyVal1 is "A" (ASCII 65)
OUTPUT(MyVal1);
// this is directly equivalent to:
// STRING1 MyVal1 := TRANSFER(MyInt,STRING1);INTEGER1 MyVal3 := (INTEGER)MyVal1;
//MyVal3 is 0 (zero) because "A" is not a numeric character
```

See Also: TRANSFER, Type Casting
ATAN

ATAN(tangent)

<table>
<thead>
<tr>
<th>tangent</th>
<th>The REAL tangent value for which to find the arctangent.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>ATAN returns a single REAL value.</td>
</tr>
</tbody>
</table>

The ATAN function returns the arctangent (inverse) of the tangent, in radians.

Example:

```plaintext
Rad2Deg := 57.29579513082; //number of degrees in a radian
TangentAngle := 57.74;
ArcTangent := ATAN(TangentAngle) * Rad2Deg;
OUTPUT(ArcTangent);
```

See Also: ATAN2, ACOS, COS, ASIN, SIN, TAN, COSH, SINH, TANH
ATAN2

ATAN2(y, x)

<table>
<thead>
<tr>
<th>y</th>
<th>The REAL numerator value for the tangent.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>The REAL denominator value for the tangent.</td>
</tr>
</tbody>
</table>

Return: ATAN2 returns a single REAL value.

The ATAN2 function returns the arctangent (inverse) of the calculated tangent, in radians. This is similar to the ATAN function but more accurate and handles the situations where x or y is zero.

Example:

```plaintext
// get the value of tan^-1(5.0 / 2.0)
TangentNumerator := 5.0;
TangentDenominator:=2.0;
ArcTangent := ATAN2(TangentNumerator, TangentDenominator);
OUTPUT(ArcTangent); //1.190289949682532
```

See Also: ATAN, ACOS, COS, ASIN, SIN, TAN, COSH, SINH, TANH
AVE

AVE(recordset, value [ , expression ] [ , KEYED ] [ , UNORDERED | ORDERED(bool) ] [ , STABLE | UNSTABLE ] [ , PARALLEL [ (numthreads) ] [ , ALGORITHM(name) ] ]

AVE(valuelist)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set. This also may be the keyword GROUP to indicate averaging the field values in a group.</td>
</tr>
<tr>
<td>value</td>
<td>The expression to find the average value of.</td>
</tr>
<tr>
<td>expression</td>
<td>Optional. A logical expression indicating which records to include in the average. Valid only when the recordset parameter is the keyword GROUP to indicate averaging the elements in a group.</td>
</tr>
<tr>
<td>KEYED</td>
<td>Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
<tr>
<td>valuelist</td>
<td>A comma-delimited list of expressions to find the average value of. This may also be a SET of values.</td>
</tr>
</tbody>
</table>

Return: AVE returns a single value.

The AVE function either returns the average value (arithmetic mean) from the specified recordset or the valuelist. It is defined to return zero if the recordset is empty.

Example:

```
AvgVal1 := AVE(4,8,16,2,1); //returns 6.2
SetVals := [4,8,16,2,1];
AvgVal2 := AVE(SetVals);    //returns 6.2
OUTPUT(AvgVal1);
OUTPUT(AvgVal2);
```

See Also: MIN, MAX
**BUILD**

```
[attrname := ] BUILD( baserecset, [ indexrec ], indexfile [, options ] );
[attrname := ] BUILD( baserecset, keys, payload, indexfile [, options ] );
[attrname := ] BUILD( indexdef [,indexfile] [, options ] );
[attrname := ] BUILD( indexdef, dataset [, options ] );
BUILD( library );
```

<table>
<thead>
<tr>
<th>attrname</th>
<th>Optional. The action name, which turns the action into an attribute definition, therefore not executed until the <code>attrname</code> is used as an action.</th>
</tr>
</thead>
<tbody>
<tr>
<td>baserecset</td>
<td>The set of data records for which the index file will be created. This may be a record set derived from the base data with the key fields and file position.</td>
</tr>
<tr>
<td>indexrec</td>
<td>Optional. The RECORD structure of the fields in the <code>indexfile</code> that contains key and file position information for referencing into the <code>baserecset</code>. Field names and types must match the <code>baserecset</code> fields (REAL and DECIMAL value type fields are not supported). This may also contain additional fields not present in the <code>baserecset</code> (computed fields). If omitted, all fields in the <code>baserecset</code> are used. The last field must be the name of an UNSIGNED8 field defined using the (VIRTUAL(filepposition)) field modifier in the DATASET declaration of the <code>baserecset</code>.</td>
</tr>
<tr>
<td>keys</td>
<td>The RECORD structure of key fields that reference into the <code>baserecset</code> (the &quot;search terms&quot; for the INDEX). Key fields may be <code>baserecset</code> fields or computed fields. REAL and DECIMAL types are not supported as &quot;search term&quot; fields. If omitted, all fields in the <code>baserecset</code> are used. This RECORD structure is typically defined inline within the BUILD using curly braces ({}), but may also be a separately defined RECORD structure. If the RECORD structure is separately defined it must meet the same requirements as used by the TABLE() function (the RECORD structure must define the type, name, and source of the data for each field), otherwise the BUILD action will not syntax check.</td>
</tr>
<tr>
<td>payload</td>
<td>The RECORD structure of the <code>indexfile</code> that contains additional fields not used as &quot;search term&quot; keys. This may contain fields from the <code>baserecset</code> and/or computed fields. If the name of the <code>baserecset</code> is in this structure, it specifies &quot;all other fields not already named in the keys parameter&quot; are added. The payload fields do not take up space in the non-leaf nodes of the index and cannot be referenced in a KEYED() filter clause. Any field with the {BLOB} modifier (to allow more than 32K of data per index entry) is stored within the <code>indexfile</code>, but not with the rest of the record; accessing the BLOB data requires an additional seek. This RECORD structure is typically defined inline within the INDEX using curly braces ({}), but may also be a separately defined RECORD structure. If the RECORD structure is separately defined it must meet the same requirements as used by the TABLE() function (the RECORD structure must define the type, name, and source of the data for each field), otherwise the BUILD action will not syntax check.</td>
</tr>
<tr>
<td>indexfile</td>
<td>A string constant containing the logical filename of the index to produce. See the Scope &amp; Logical Filenames article for more on logical filenames.</td>
</tr>
<tr>
<td>options</td>
<td>Optional. One or more of the options listed below.</td>
</tr>
<tr>
<td>indexdef</td>
<td>The name of the INDEX attribute to build.</td>
</tr>
<tr>
<td>dataset</td>
<td>The name of the DATASET to use when you omit the base dataset parameter from the INDEX definition.</td>
</tr>
<tr>
<td>library</td>
<td>The name of a MODULE attribute with the LIBRARY option.</td>
</tr>
</tbody>
</table>
The first four forms of the **BUILD** action create index files. Indexes are automatically compressed, minimizing overhead associated with using indexed record access. The keyword BUILDINDEX may be used in place of BUILD in these forms.

The fifth form creates an external query library—a workunit that implements the specified library. This is similar to creating a .DLL in Windows programming, or a .SO in Linux.

## Index BUILD Options

The following options are available on all three INDEX forms of BUILD (only):

```
```

### CLUSTER

Specifies writing the *indexfile* to the specified list of target clusters. If omitted, the *indexfile* is written to the cluster on which the workunit executes. The number of physical file parts written to disk is always determined by the number of nodes in the cluster on which the workunit executes, regardless of the number of nodes on the target cluster(s) unless the WIDTH option is also specified.

**target**

A comma-delimited list of string constants containing the names of the clusters to write the *indexfile* to. The names must be listed as they appear on the ECL Watch Activity page or returned by theStd.System.Thorlib.Group() function, optionally with square brackets containing a comma-delimited list of node-numbers (1-based) and/or ranges (specified with a dash, as in n-m) to indicate the specific set of nodes to write to.

### SORTED

Specifies that the baserecsset is already sorted, implying that the automatic sort based on all the indexrec fields is not required before the index is created.

### DISTRIBUTE

Specifies building the *indexfile* based on the distribution of the key.

**key**

The name of an existing INDEX attribute definition.

### MERGE

Optional. Specifies merging the resulting index into the specified key.

### DATASET

This is only needed when the baserecsset is the result of an operation (such as a JOIN) whose result makes it ambiguous as to which physical dataset is being indexed (in other words, use this option only when you receive an error that it cannot be deduced). Naming the basedataset ensures that the proper record links are used in the index.

**basedataset**

The name of the DATASET attribute from which the baserecsset is derived.

### OVERWRITE

Specifies overwriting the *indexfile* if it already exists.

### UPDATE

Specifies that the file should be rewritten only if the code or input data has changed.

### EXPIRE

Optional. Specifies the file is a temporary file that may be automatically deleted after the specified number of days since the file was read.

### FILEPOSITION

Optional. If flag is FALSE, prevents the implicit fileposition field from being created and will not treat a trailing integer field any differently from the rest of the payload.

**flag**

Optional. TRUE or FALSE, indicating whether or not to create the implicit fileposition field.
<table>
<thead>
<tr>
<th><strong>days</strong></th>
<th>Optional. The number of days from last file read after which the file may be automatically deleted. If omitted, the default is seven (7).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FEW</strong></td>
<td>Specifies the <em>indexfile</em> is created as a single one-part file. Used only for small datasets (typically lookup-type files, such as 2-character state codes). This option is now deprecated in favor of using the WIDTH(1).</td>
</tr>
<tr>
<td><strong>indexdef</strong></td>
<td>The name of an existing INDEX attribute definition that provides the baserecset, indexrec, and indexfile parameters to use.</td>
</tr>
<tr>
<td><strong>LOCAL</strong></td>
<td>Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE function.</td>
</tr>
<tr>
<td><strong>NOROOT</strong></td>
<td>Specifies that the index is not globally sorted, and there is no root index to indicate which part of the index will contain a particular entry. This may be useful in Roxie queries in conjunction with ALLNODES use.</td>
</tr>
<tr>
<td><strong>DISTRIBUTED</strong></td>
<td>Specifies both the LOCAL and NOROOT options (congruent with the DISTRIBUTED option on an INDEX declaration, which specifies the index was built with the LOCAL and NOROOT options).</td>
</tr>
<tr>
<td><strong>COMPRESSED</strong></td>
<td>Optional. Specifies the index should be compressed using the type of compression specified. If omitted, the default is <strong>LZW</strong>, a variant of the Lempel-Ziv-Welch algorithm.</td>
</tr>
<tr>
<td><strong>option</strong></td>
<td>See Indexes and Compression for options.</td>
</tr>
<tr>
<td><strong>WIDTH</strong></td>
<td>Specifies writing the <em>indexfile</em> to a different number of physical file parts than the number of nodes in the cluster on which the workunit executes. If omitted, the default is the number of nodes in the cluster on which the workunit executes. This option is primarily to create indexes on a large Thor that are destined to be deployed to a smaller Roxie (making the Roxie queries more efficient).</td>
</tr>
<tr>
<td><strong>nodes</strong></td>
<td>The number of physical file parts to write. If set to one (1), this operates exactly the same as the FEW option, above.</td>
</tr>
<tr>
<td><strong>DEDUP</strong></td>
<td>Specifies that duplicate entries are eliminated from the INDEX.</td>
</tr>
<tr>
<td><strong>SKEW</strong></td>
<td>Indicates that you know the data will not be spread evenly across nodes (will be skewed and you choose to override the default by specifying your own limit value to allow the job to continue despite the skewing.)</td>
</tr>
<tr>
<td><strong>limit</strong></td>
<td>A value between zero (0) and one (1.0 = 100%) indicating the maximum percentage of skew to allow before the job fails (the default skew is 1.0 / &lt;number of worker nodes on cluster&gt;).</td>
</tr>
<tr>
<td><strong>target</strong></td>
<td>Optional. A value between zero (0) and one (1.0 = 100%) indicating the desired maximum percentage of skew to allow (the default skew is 1.0 / &lt;number of worker nodes on cluster&gt;).</td>
</tr>
<tr>
<td><strong>THRESHOLD</strong></td>
<td>Indicates the minimum size for a single part before the SKEW limit is enforced.</td>
</tr>
<tr>
<td><strong>size</strong></td>
<td>An integer value indicating the minimum number of bytes for a single part. Default is 1GB.</td>
</tr>
<tr>
<td><strong>MAXLENGTH</strong></td>
<td>Optional. This option is used to create indexes that are backward compatible for platform versions prior to 3.0. Specifies the maximum length of a variable-length index record. Fixed length records always use the minimum size required. If the default maximum length causes inefficiency problems, it can be explicitly overridden.</td>
</tr>
<tr>
<td><strong>value</strong></td>
<td>Optional. An integer value indicating the maximum length. If omitted, the maximum size is calculated from the record structure. Variable-length records that do not specify MAXLENGTH may be slightly inefficient.</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>UNORDERED</strong></td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td><strong>ORDERED</strong></td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td><strong>bool</strong></td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><strong>STABLE</strong></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td><strong>UNSTABLE</strong></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><strong>PARALLEL</strong></td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td><strong>numthreads</strong></td>
<td>Optional. Try to evaluate this activity using <code>numthreads</code> threads.</td>
</tr>
<tr>
<td><strong>ALGORITHM</strong></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
<tr>
<td><strong>SET</strong></td>
<td>Optional. SET is used to set a value to a named metadata option. This allows you to set user metadata whose use and purpose is up to the developer. Currently <code>_nodeSize</code> is the only system-defined metadata, though other names starting with an underscore (_) should be considered reserved for system use. You may want to use SET(&quot;_nodeSize&quot;, '32768') if your hardware and usage pattern work better with larger page sizes. The default (8192) may not be optimal for all scenarios on modern hardware. We recommend using a power of 2 and not smaller than 8k.</td>
</tr>
<tr>
<td><strong>option</strong></td>
<td>A case sensitive string constant containing the name of the option to set.</td>
</tr>
<tr>
<td><strong>value</strong></td>
<td>The value to set the option to. This may be any type of value, dependent on what the option expects to be.</td>
</tr>
</tbody>
</table>

**BUILD an Access Index**

```ecl
[attribname := ] BUILD(baserecset, [ indexrec ], indexfile [, options ]); 
```

Form 1 creates an index file to allow keyed access to the `baserecset`. The index is used primarily by the FETCH and JOIN (with the KEYED option) operations.

Example:

```ecl
Vehicles := DATASET('vehicles',
  {STRING2 st,
    STRING20 city,
    STRING20 lname,
    UNSIGNED8 filepos{VIRTUAL(fileposition)}},
  FLAT);
BUILD(Vehicles,{lname,filepos},'vkey::lname');
//build key into Vehicles dataset on last name
```

**BUILD a Payload Index**

```ecl
[attribname := ] BUILD(baserecset, keys, payload, indexfile [, options ]); 
```

Form 2 creates an index file containing extra `payload` fields in addition to the `keys`. This form is used primarily to create indexes used by "half-key" JOIN operations to eliminate the need to directly access the `baserecset`,
thus increasing performance over the "full-keyed" version of the same operation (done with the KEYED option on the JOIN).

By default, the **payload** fields are sorted during the BUILDINDEX operation to minimize space on the leaf nodes of the key. This sorting can be controlled by using `sortIndexPayload` in a #OPTION statement.

Example:

```ecl
Vehicles := DATASET('vehicles',
    {STRING2 st,
     STRING20 city,
     STRING20 lname,
     UNSIGNED8 filepos{VIRTUAL{fileposition}}},
    FLAT);
BUILD(Vehicles,(st,city),(lname),'vkey::st.city1');
//build key into Vehicles dataset on state and city
//payload the last name

//same index build using non-inline RECORD structures
SearchTerms := RECORD
    Vehicles.st;
    Vehicles.city;
END;
Payload := RECORD
    Vehicles.lname;
END;
BUILD(Vehicles,SearchTerms,Payload,'vkey::st.city2');
```

**BUILD from an INDEX Definition**

```
[attname := ] BUILD( indexdef [,indexfile] [, options ]);
```

Form 3 creates an index file by using a previously defined INDEX definition.

You can also use this form to build an index based upon two MERGED indexes. To write a MERGED index to disk, you must use the BUILD action and include the optional `indexfile` parameter.

Example:

```ecl
nameKey := INDEX(mainTable,{surname,forename,filepos},'name.idx');
BUILD(nameKey); //gets all info from the INDEX definition

// BUILDing a MERGED index
BUILD(MERGE(idx1,idx2,'~idx::MergedIndex');
```

See Also: MERGE
Separating INDEX from DATASET

\[ \text{attrname} := \text{BUILD( indexdef, dataset, [, options ] )}; \]

Form 4 allows you to separate the index from the dataset, making it easy to use the INDEX form that does not require a dataset first parameter.

The usual way of defining and building an INDEX is in terms of the dataset that is used to create it, using an INDEX declaration where the dataset is named as the first parameter of the INDEX. This is fine when the dataset is relatively simple, but there are downsides:

- The INDEX can’t be logically separated from the dataset it was created from.
- If the dataset is very complicated (Mb of source), when the index is subsequently used in a query all the code to create it is also parsed.

This form of BUILD allows the two to be separated, so you can omit the base dataset parameter from the INDEX definition and just specify the dataset to use in the BUILD action. The fields are automatically mapped (by field name) from the dataset to the index.

Examples:

//usual way to BUILD an INDEX:
ds1 = DATASET(100, TRANSFORM({ UNSIGNED id }, SELF.id := COUNTER));
i1 := INDEX(ds1, { id }, 'myIndex'); //specifies the dataset to always use
BUILD(i1);

//Separated way to BUILD an INDEX:
ds2 = DATASET(100, TRANSFORM({ UNSIGNED id }, SELF.id := COUNTER));
i2 := INDEX({ UNSIGNED id }, 'myIndex');
BUILD(i2, ds2);       //builds the i2 INDEX from the ds2 dataset

BUILD a Query Library

BUILD( library );

Form 5 creates an external query library for use in hthor or Roxie, only.

A query library allows a set of related attributes to be packaged as a self contained unit so the code can be shared between different workunits. This reduces the time required to deploy a set of attributes, and also reduces the memory footprint for the set of queries within Roxie that use the library. Also, functionality in the library can be updated without having to re-deploy all the queries that use that functionality.

Query libraries are suitable for packaging together sets of functions that are closely related. They aren’t suited for including attributes defined as MACROs--the meaning of a macro isn’t known until its parameters are substituted.

The name form of $\#\text{WORKUNIT}$ names the workunit that BUILD creates as the external library. That name is the external library name used by the LIBRARY function (which provides access to the library from within the query that uses the library). Since the workunit itself is the external query library, BUILD(library) must be the only action in the workunit.

Example:

NamesRec := RECORD
  INTEGER1 NameID;
  STRING20 FName;

STRING20  LName;
END;

FilterLibIface1(DATASET(namesRec) ds, STRING search) := INTERFACE
  EXPORT DATASET(namesRec) matches;
  EXPORT DATASET(namesRec) others;
END;

FilterDsLib1(DATASET(namesRec) ds, STRING search) :=
  MODULE,LIBRARY(FilterLibIface1)
  EXPORT matches := ds(Lname  = search);
  EXPORT others  := ds(Lname != search);
END;
#WORKUNIT('name','Ppass.FilterDsLib')
BUILD(FilterDsLib1);
## Indexes and Compression

### Compression Options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LZW</strong></td>
<td>The default compression. It is a variant of the Lempel-Ziv-Welch algorithm. It remains the default for backward compatibility.</td>
</tr>
<tr>
<td><strong>ROW</strong></td>
<td>Compresses index entries based on differences between rows (for use with fixed-length records only). It typically does not compress as well as LZW, but takes up less space in memory because the rows are expanded on demand.</td>
</tr>
<tr>
<td><strong>'inplace'</strong></td>
<td>Causes the index to be built using the inplace compression format. The payload defaults to using lz4 compression.</td>
</tr>
<tr>
<td><strong>'inplace: lz4hc'</strong></td>
<td>Causes inplace compression on the key fields and lz4hc compression on the payload. The resulting index can be smaller than using lz4.</td>
</tr>
</tbody>
</table>

The inplace index compression format (introduced in version 9.2.0) improves compression of keyed fields and allows them to be searched without decompression. The original index compression implementation decompresses the rows when they are read from disk.

Because the branch nodes can be searched without decompression more branch nodes fit into memory which can improve search performance. The lz4 compression used for the payload is significantly faster at decompressing leaf pages than the previous LZW compression.

Whether performance is better with lz4hc (a high-compression variant of lz4) on the payload fields depends on the access characteristics of the data and how much of the index is cached in memory.

If you attempt to read an index with the inplace compression format on a system that does not support them, you will receive an error message.

See Also: INDEX, JOIN, FETCH, MODULE, INTERFACE, LIBRARY, DISTRIBUTE, #WORKUNIT
### CASE

**CASE(expression, caseval => value, [... , caseval => value][, elsevalue])**

- **expression**: An expression that results in a single value.
- **caseval**: A value to compare against the result of the expression.
- **=>**: The "results in" operator--valid only in CASE, MAP and CHOOSESETS.
- **value**: The value to return. This may be any expression or action.
- **elsevalue**: Optional. The value to return when the result of the expression does not match any of the *caseval* values. May be omitted if all return values are actions (the default would then be no action), or all return values are record sets (the default would then be an empty record set).

**Return:** CASE returns a single value, a set of values, a record set, or an action.

The **CASE** function evaluates the *expression* and returns the *value* whose *caseval* matches the *expression* result. If none match, it returns the *elsevalue*.

There may be as many *caseval => value* parameters as necessary to specify all the expected values of the *expression* (there must be at least one). All return *value* parameters must be of the same type.

**Example:**

```plaintext
//simple example
MyExp := 1+2;
MyChoice := CASE(MyExp, 1 => 9, 2 => 8, 3 => 7, 4 => 6, 5);
// returns a value of 7 for the value of MyExp=3
OUTPUT(MyChoice);

//example using a DATASET
personRecord := RECORD
  STRING UID;
  STRING first_name;
  STRING last_name;
  STRING address;
  STRING city;
  STRING state;
  STRING zip;
END;
person := DATASET([{'923','James','Jones','123 Elm Street','Hollywood','FL','33022'},
  {'924','Sally','Jones','22 Main Street','Tampa','FL','33604'},
  {'925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101'},
  {'926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108'},
  {'927','Evelyn','Murray','740 SW 10th Street','Boston ','MA','02116'},
  {'928','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131'}], personRecord);
MyRecSet := CASE(MyExp, 1 => Person(state = 'FL'),
  2 => Person(state = 'GA'),
  3 => Person(state = 'MA'),
  4 => Person(state = 'IL'),
  person);
// returns set of MA Persons for the value of MyExp=3
// set MyExp to a number > 4 to get all records
OUTPUT(MyRecSet);
```

**See Also:** MAP, CHOOSE, IF, REJECTED, WHICH
# CATCH

```plaintext
result := CATCH( recset, action [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] ] [, ALGORITHM( name ) ] );
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>result</code></td>
<td>The definition name for the resulting recordset.</td>
</tr>
<tr>
<td><code>recset</code></td>
<td>The recordset expression that, if it fails, causes the <code>action</code> to launch.</td>
</tr>
<tr>
<td><code>action</code></td>
<td>One of the three valid actions below.</td>
</tr>
<tr>
<td><code>UNORDERED</code></td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td><code>ORDERED</code></td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><code>STABLE</code></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td><code>UNSTABLE</code></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><code>PARALLEL</code></td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td><code>numthreads</code></td>
<td>Optional. Try to evaluate this activity using <code>numthreads</code> threads.</td>
</tr>
<tr>
<td><code>ALGORITHM</code></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: CATCH returns a set of records (which may be empty).

The **CATCH** function executes the `action` if the `recset` expression fails for any reason.

Valid **actions** are:

- **SKIP**
  - Specifies ignoring the error and continuing, returning an empty dataset.

- **ONFAIL**(transform)
  - Specifies returning a single record from the `transform` function. The TRANSFORM function may use FAILCODE and/or FAILMESSAGE to provide details of the failure and must result in a RECORD structure the same format as the `recset`.

- **FAIL**
  - The FAIL action, which specifies the error message to produce. This is meant to provide more useful information to the end user about why the job failed.

**Example:**

```plaintext
MyRec := RECORD
  STRING50 Value1;
  UNSIGNED Value2;
END;

d := DATASET([{'C',1},{'C',2},{'C',3},
  {'C',4},{'C',5},{'X',1},{'A',1}],MyRec);

MyRec FailTransform := TRANSFORM
  self.value1 := FAILMESSAGE[1..17];
  self.value2 := FAILCODE
END;

limited1 := LIMIT(d, 2);
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
limited2 := LIMIT(ds, 3);
limited3 := LIMIT(ds, 4);

recovered1 := CATCH(limited1, SKIP);
recovered2 := CATCH(limited2, ONFAIL(FailTransform));
recovered3 := CATCH(CATCH(limited3, FAIL(1, 'Failed, sorry')), ONFAIL(FailTransform));

OUTPUT(recovered1);  //empty recordset
OUTPUT(recovered2);  //
OUTPUT(recovered3);  //

See Also: TRANSFORM Structure, FAIL, FAILCODE, FAILMESSAGE
**CHOSE**

**CHOSE**(*expression, value,... , value, elsevalue*)

<table>
<thead>
<tr>
<th><strong>expression</strong></th>
<th>An arithmetic expression that results in a positive integer and determines which value parameter to return.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>value</strong></td>
<td>The values to return. There may be as many value parameters as necessary to specify all the expected values of the expression. This may be any expression or action.</td>
</tr>
<tr>
<td><strong>elsevalue</strong></td>
<td>The value to return when the expression returns an out-of-range value. The last parameter is always the elsevalue.</td>
</tr>
</tbody>
</table>

Return: CHOOSE returns a single value.

The CHOOSE function evaluates the expression and returns the value parameter whose ordinal position in the list of parameters corresponds to the result of the expression. If none match, it returns the elsevalue. All values and the elsevalue must be of the same type.

Example:

```ecl
define MyExp := 1+2;
define MyChoice := CHOOSE(MyExp,9,8,7,6,5); // returns 7  
define MyChoice2 := CHOOSE(MyExp,1,2,3,4,5); // returns 3  
define MyChoice3 := CHOOSE(MyExp,15,14,13,12,11); // returns 13  
OUTPUT(MyChoice);  
OUTPUT(MyChoice2);  
OUTPUT(MyChoice3);  
define IntRate:= 6.5;  
define RateRating := CHOOSE(IntRate,'Great','Good','Pretty Good','Fair','Fair','High','High','High','High','Invalid');  
// RateRating receives 'High' if the IntRate is between 6 and 9  
OUTPUT(RateRating);```

See Also: CASE, IF, MAP
CHOSEN

CHOSEN(recordset, n [, startpos] [, FEW] [, UNORDERED | ORDERED(bool)] [, STABLE | UNSTABLE] [, PARALLEL(numthreads)] [, ALGORITHM(name)])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.</td>
</tr>
<tr>
<td>n</td>
<td>The number of records to return. If zero (0), no records are returned, and if ALL or CHOSEN:ALL, all records are returned. The CHOSEN:ALL option is a constant that may be used in any expression.</td>
</tr>
<tr>
<td>startpos</td>
<td>Optional. The ordinal position in the recordset of the first record to return. If omitted, the default is one (1).</td>
</tr>
<tr>
<td>FEW</td>
<td>Optional. Specifies internally converting to a TOPN operation if n is a variable number (an attribute or passed parameter) and the input recordset comes from a SORT.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function’s STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: CHOSEN returns a set of records.

The CHOSEN function (choose-n) returns the first n number of records, beginning with the record at the startpos, from the specified recordset.

Example:

```
AllRecs := CHOSEN(Person,ALL); // returns all recs from Person
FirstFive := CHOSEN(Person,5); // returns first 5 recs from Person
NextFive := CHOSEN(Person,5,6); // returns next 5 recs from Person
LimitRecs := CHOSEN(Person,IF(MyLimit<>0,MyLimit,CHOSEN:ALL));
```

See Also: SAMPLE, CHOOSESETS
CHOOSESETS

CHOOSESETS( recset, condition => n[, o][, EXCLUSIVE [LAST | ENTH [, UNORDERED | ORDERED( bool )][, STABLE | UNSTABLE ][, PARALLEL [( numthreads )]]][, ALGORITHM( name )]])

recset The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.

condition The logical expression that defines which records to include in the result set.

=> The "results in" operator--valid only in CHOOSESETS, CASE, and MAP.

n The maximum number of records to return. If zero (0), no records that meet the condition are returned.

o Optional. The maximum number of records to return that meet none of the conditions specified.

EXCLUSIVE Optional. Specifies the condition parameters are mutually exclusive.

LAST Optional. Specifies choosing the last n records that meet the condition instead of the first n. This option is implicitly EXCLUSIVE.

ENTH Optional. Specifies choosing a sample of records that meet the condition instead of the first n. This option is implicitly EXCLUSIVE.

UNORDERED Optional. Specifies the output record order is not significant.

ORDERED Specifies the significance of the output record order.

bool When False, specifies the output record order is not significant. When True, specifies the default output record order.

STABLE Optional. Specifies the input record order is significant.

UNSTABLE Optional. Specifies the input record order is not significant.

PARALLEL Optional. Try to evaluate this activity in parallel.

numthreads Optional. Try to evaluate this activity using numthreads threads.

ALGORITHM Optional. Override the algorithm used for this activity.

name The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

Return: CHOOSESETS returns a set of records.

The CHOOSESETS function returns a set of records from the recset. The result set is limited to n number of records that meet each condition listed. CHOOSESETS may take as many condition => n parameters as needed to exactly specify the desired set of records. This is a shorthand way of concatenating the result sets of multiple CHOOSE function calls to the same recset with different filter conditions, but CHOOSESETS executes significantly faster. This technique is also know as a "cutback."

Example:

MyResultSet := CHOOSESETS(Person,
  per_first_name = 'RICHARD' => 100,
  per_first_name = 'GWENDOLYN' => 200, 100)
// returns a set containing 100 Richards, 200 Gwendolyns, 100 others

See Also: CHOOSE, SAMPLE
CLUSTERSIZE

CLUSTERSIZE

<table>
<thead>
<tr>
<th>Return:</th>
<th>CLUSTERSIZE returns a single INTEGER value.</th>
</tr>
</thead>
</table>

The **CLUSTERSIZE** compile time constant returns the number of nodes in the cluster. This is the same value as returned by the `Std.System.ThorLib.Nodes()` function.

Example:

```ecl
OUTPUT (CLUSTERSIZE); 
```
COMBINE

COMBINE( leftrecset, rightrecset [, transform ][,LOCAL])

COMBINE( leftrecset, rightrecset, GROUP , transform [,LOCAL] [, UNORDERED | ORDERED( bool) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ]] [, ALGORITHM( name ) ] )

leftrecset The LEFT record set.
rightrecset The RIGHT record set.
transform The TRANSFORM function call. If omitted, COMBINE returns all fields from both the leftrecset and rightrecset, with the second of any duplicate named fields removed.
LOCAL The LOCAL option is required when COMBINE is used on Thor (and implicit in hThor/Roxie).
GROUP Specifies the rightrecset has been GROUPed. If this is not the case, an error occurs.
UNORDERED Optional. Specifies the output record order is not significant.
ORDERED Specifies the significance of the output record order.
bool When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE Optional. Specifies the input record order is significant.
UNSTABLE Optional. Specifies the input record order is not significant.
PARALLEL Optional. Try to evaluate this activity in parallel.
numthreads Optional. Try to evaluate this activity using numthreads threads.
ALGORITHM Optional. Override the algorithm used for this activity.
name The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

Return: COMBINE returns a record set.

The COMBINE function combines leftrecset and rightrecset on a record-by-record basis in the order in which they appear in each.

COMBINE TRANSFORM Function Requirements

For form 1, the transform function must take at least two parameters: a LEFT record which must be in the same format as the leftrecset and a RIGHT record which must be in the same format as the rightrecset. The format of the resulting record set may be different from the inputs.

For form 2, the transform function must take at least three parameters: a LEFT record which must be in the same format as the leftrecset, a RIGHT record which must be in the same format as the rightrecset, and a ROWS(RIGHT) whose format must be a DATASET(RECORDOF(rightrecset)) parameter. The format of the resulting record set may be different from the inputs.

COMBINE Form 1

Form 1 of COMBINE produces its result by passing each record from leftrecset along with the record in the same ordinal position within rightrecset to the transform to produce a single output record. Grouping (if any) on the leftrecset is preserved. An error occurs if leftrecset and rightrecset contain a different number of records.
**Example:**

```sql
inrec := RECORD
  UNSIGNED6 uid;
END;
outrec := RECORD(inrec)
  STRING20 name;
  STRING11 ssn;
  UNSIGNED8 dob;
END;
ds := DATASET([1,2,3,4,5,6], inrec);
i1 := DATASET([ {1, 'Kevin' },
  {2, 'Richard'},
  {5, 'Nigel' } ],
  {UNSIGNED6 uid, STRING10 name });
i2 := DATASET([ {3, '000-12-3462'},
  {5, '000-12-8723'},
  {6, '000-10-1002' } ],
  {UNSIGNED6 uid, STRING11 ssn });
i3 := DATASET([ {1, 19700117},
  {4, 19831212},
  {6, 20010101} ],
  {UNSIGNED6 uid, UNSIGNED8 dob });
j1 := JOIN(ds, i1, LEFT.uid = RIGHT.uid, LEFT OUTER, LOOKUP);
j2 := JOIN(ds, i2, LEFT.uid = RIGHT.uid, LEFT OUTER, LOOKUP);
j3 := JOIN(ds, i3, LEFT.uid = RIGHT.uid, LEFT OUTER, LOOKUP);
combined1 := COMBINE(j1, j2,
  TRANSFORM(outRec,
    SELF := LEFT;
    SELF := RIGHT;
    SELF := []),
  LOCAL);
combined2 := COMBINE(combined1, j3,
  TRANSFORM(outRec,
    SELF.dob := RIGHT.dob;
    SELF := LEFT),
  LOCAL);
OUTPUT(combined1);
OUTPUT(combined2);
```

**COMBINE Form 2**

Form 2 of COMBINE produces its result by passing each record from *leftrecset*, the group in the same ordinal position within *rightrecset* (along with the first record in the group) to the *transform* to produce a single output record. Grouping (if any) on the *leftrecset* is preserved. An error occurs if the number of records in the *leftrecset* differs from the number of groups in the *rightrecset*.

**Example:**

```sql
inrec := {UNSIGNED6 uid};
outrec := RECORD(inrec)
  STRING20 name;
  UNSIGNED score;
END;
nameRec := RECORD
  STRING20 name;
END;
resultRec := RECORD(inrec)
  DATASET(nameRec) names;
END;
resultRec := RECORD(inrec)
  DATASET(nameRec) names;
END;
ds := GROUP(ds, ROW);
```
i1 := DATASET([[1, 'Kevin', 10],
                [2, 'Richard', 5],
                [5, 'Nigel', 2],
                [0, '', 0]], outrec);

i2 := DATASET([[1, 'Kevin Hall', 12],
                [2, 'Richard Chapman', 15],
                [3, 'Jake Smith', 20],
                [5, 'Nigel Hicks', 100],
                [0, '', 0]], outrec);

i3 := DATASET([[1, 'Halligan', 8],
                [2, 'Richard', 8],
                [6, 'Pete', 4],
                [6, 'Peter', 8],
                [6, 'Petie', 1],
                [0, '', 0]], outrec);

j1 := JOIN(dsg, i1,
            LEFT.uid = RIGHT.uid,
            TRANSFORM(outrec,
                       SELF := LEFT;
                       SELF := RIGHT),
            LEFT OUTER, MANY LOOKUP);

j2 := JOIN(dsg, i2,
            LEFT.uid = RIGHT.uid,
            TRANSFORM(outrec,
                       SELF := LEFT;
                       SELF := RIGHT),
            LEFT OUTER, MANY LOOKUP);

j3 := JOIN(dsg, i3,
            LEFT.uid = RIGHT.uid,
            TRANSFORM(outrec,
                       SELF := LEFT;
                       SELF := RIGHT),
            LEFT OUTER, MANY LOOKUP);

combined := REGROUP(j1, j2, j3);

resultRec t(inrec l, DATASET(RECORDOF(combined)) r) := TRANSFORM
     SELF.names := PROJECT(r, TRANSFORM(nameRec, SELF := LEFT));
     SELF := l;
END;

res1 := COMBINE(dsg, combined, GROUP, t(LEFT, ROWS(RIGHT)(score != 0)), LOCAL);
OUTPUT(res1);

// A variation using rows in a child query.
resultRec t2(inrec l, DATASET(RECORDOF(combined)) r) := TRANSFORM
     SELF.names := PROJECT(SORT(r, -score),
                            TRANSFORM(nameRec,
                                       SELF := LEFT));
     SELF := l;
END;

res2 := COMBINE(dsg, combined, GROUP, t2(LEFT, ROWS(RIGHT)(score != 0)), LOCAL);
OUTPUT(res2);

See Also: GROUP, REGROUP
CORRELATION


<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recset</td>
<td>The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set. This also may be the GROUP keyword to indicate operating on the elements in each group, when used in a RECORD structure to generate crosstab statistics.</td>
</tr>
<tr>
<td>valuex</td>
<td>A numeric field or expression.</td>
</tr>
<tr>
<td>valuey</td>
<td>A numeric field or expression.</td>
</tr>
<tr>
<td>expression</td>
<td>Optional. A logical expression indicating which records to include in the calculation. Valid only when the recset parameter is the keyword GROUP.</td>
</tr>
<tr>
<td>KEYED</td>
<td>Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: CORRELATION returns a single REAL value.

The **CORRELATION** function returns the Pearson's Product Moment Correlation Coefficient between valuex and valuey.

Example:

```ecl
pointRec := { REAL x, REAL y };
analyze( ds ) := MACRO
  #uniquename(stats)
  %stats% := TABLE( ds, [ c := COUNT(GROUP),
    sx := SUM(GROUP, x),
    sy := SUM(GROUP, y),
    sxx := SUM(GROUP, x * x),
    sxy := SUM(GROUP, x * y),
    syy := SUM(GROUP, y * y),
    varx := VARIANCE(GROUP, x);
    vary := VARIANCE(GROUP, y);
    varxy := COVARIANCE(GROUP, x, y);
    rc := CORRELATION(GROUP, x, y) ] );
OUTPUT( %stats% );
  // Following should be zero
OUTPUT( %stats%, { varx = (sxx-sx*sx/c)/c,
  vary = (syy-sy*sy/c)/c,
  varxy = (sxy-sx*sy/c)/c, );
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
rc - (varxy/SQRT(varx*vary)) });
OUTPUT(%stats%, { 'bestFit: y=' +
  (STRING)((sy-sx*varxy/varx)/c) +
' + ' +
  (STRING)(varxy/varx)+'x' }));
ENDMACRO;
ds1 := DATASET([{1,1},{2,2},{3,3},{4,4},{5,5},{6,6}], pointRec);
ds2 := DATASET([ {1.93896e+009, 2.04482e+009},
  {2.96181e+009, 1.24848e+009},
  {2.7744e+009, 1.26357e+009},
  {1.14416e+009, 4.3429e+008},
  {3.38728e+009, 1.30238e+009},
  {3.19538e+009, 1.71177e+009} ], pointRec);
ds3 := DATASET([ {1, 1.00039},
  {2, 2.07702},
  {3, 2.86158},
  {4, 3.87114},
  {5, 5.12417},
  {6, 6.20283} ], pointRec);
analyze(ds1);
analyze(ds2);
analyze(ds3);

See Also: VARIANCE, COVARIANCE
**COS**

**COS**(*angle*)

<table>
<thead>
<tr>
<th><strong>angle</strong></th>
<th>The REAL radian value for which to find the cosine.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Return:</strong></td>
<td>COS returns a single REAL value.</td>
</tr>
</tbody>
</table>

The **COS** function returns the cosine of the *angle*.

Example:

```plaintext
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad;    //translate 45 degrees into radians
Cosine45 := COS(Angle45);   //get cosine of the 45 degree angle
OUTPUT(Cosine45);
```

See Also: ACOS, SIN, TAN, ASIN, ATAN, COSH, SINH, TANH
**COSH**

**COSH(angle)**

<table>
<thead>
<tr>
<th>angle</th>
<th>The REAL radian value for which to find the hyperbolic cosine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>COSH returns a single REAL value.</td>
</tr>
</tbody>
</table>

The **COSH** function returns the hyperbolic cosine of the *angle*.

Example:

```plaintext
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad;   //translate 45 degrees into radians
HyperbolicCosine45 := COSH(Angle45); //get hyperbolic cosine of the 45 degree angle
OUTPUT(HyperbolicCosine45);
```

See Also: ACOS, SIN, TAN, ASIN, ATAN, COS, SINH, TANH
COUNT

**COUNT(recordset [ , expression ] [ , KEYED ] [ , UNORDERED | ORDERED( bool ) ] [ , STABLE | UNSTABLE ] [ , PARALLEL [ ( numthreads ) ] ] [ , ALGORITHM( name ) ] )**

**COUNT(valuelist)**

- **recordset**: The set of records to process. This may be the name of a DATASET or a record set derived from some filter condition, or any expression that results in a derived record set, or a the name of a DICTIONARY declaration. This also may be the GROUP keyword to indicate counting the number of elements in a group, when used in a RECORD structure to generate crosstab statistics.

- **expression**: Optional. A logical expression indicating which records to include in the count. Valid only when the recordset parameter is the keyword GROUP to indicate counting the number of elements in a group.

- **KEYED**: Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.

- **UNORDERED**: Optional. Specifies the output record order is not significant.

- **ORDERED**: Specifies the significance of the output record order.

- **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.

- **STABLE**: Optional. Specifies the input record order is significant.

- **UNSTABLE**: Optional. Specifies the input record order is not significant.

- **PARALLEL**: Optional. Try to evaluate this activity in parallel.

- **numthreads**: Optional. Try to evaluate this activity using `numthreads` threads.

- **ALGORITHM**: Optional. Override the algorithm used for this activity.

- **name**: The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

- **valuelist**: A comma-delimited list of expressions to count. This may also be a SET of values.

**Return:** COUNT returns a single value.

The **COUNT** function returns the number of records in the specified **recordset** or **valuelist**.

**Example:**

```ecl
MyCount := COUNT(Trades(Trades.trd_rate IN ['3', '4', '5']));
// count the number of records in the Trades record
// set whose trd_rate field contains 3, 4, or 5
R1 := RECORD
  person.per_st;
  person.per_sex;
  Number := COUNT(GROUP);
  //total in each state/sex category
  Hanks := COUNT(GROUP,person.per_first_name = 'HANK');
  //total of "Hanks" in each state/sex category
  NonHanks := COUNT(GROUP,person.per_first_name <> 'HANK');
  //total of "Non-Hanks" in each state/sex category
END;
T1 := TABLE(person, R1, per_st, per_sex);
Cnt1 := COUNT(4,8,16,2,1); //returns 5
SetVals := [4,8,16,2,1];
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
Cnt2 := COUNT(SetVals); //returns 5

See Also: SUM, AVE, MIN, MAX, GROUP, TABLE
COVARIANCE


- **recset**: The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set. This also may be the GROUP keyword to indicate operating on the elements in each group, when used in a RECORD structure to generate crosstab statistics.

- **valuex**: A numeric field or expression.

- **valuey**: A numeric field or expression.

- **expression**: Optional. A logical expression indicating which records to include in the calculation. Valid only when the recset parameter is the keyword GROUP.

- **KEYED**: Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.

- **UNORDERED**: Optional. Specifies the output record order is not significant.

- **ORDERED**: Specifies the significance of the output record order.

- **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.

- **STABLE**: Optional. Specifies the input record order is significant.

- **UNSTABLE**: Optional. Specifies the input record order is not significant.

- **PARALLEL**: Optional. Try to evaluate this activity in parallel.

- **numthreads**: Optional. Try to evaluate this activity using numthreads threads.

- **ALGORITHM**: Optional. Override the algorithm used for this activity.

- **name**: The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

**Return:** COVARIANCE returns a single REAL value.

The **COVARIANCE** function returns the extent to which valuex and valuey co-vary.

Example:

```ecl
pointRec := { REAL x, REAL y }; analyze( ds ) := MACRO
  #unique(name(stats))
  %stats% := TABLE( ds, { c := COUNT( GROUP),
    sx := SUM( GROUP, x),
    sy := SUM( GROUP, y),
    sxx := SUM( GROUP, x * x),
    sxy := SUM( GROUP, x * y),
    syy := SUM( GROUP, y * y),
    varx := VARIANCE( GROUP, x);,
    vary := VARIANCE( GROUP, y);,
    varxy := COVARIANCE( GROUP, x, y);,
    rc := CORRELATION( GROUP, x, y ) });
OUTPUT(%stats%);
// Following should be zero
OUTPUT(%stats%, { varx - (sxx-sx*sx/c)/c,
                  vary - (syy-sy*sy/c)/c,
                  varxy - (sxy-sx*sy/c)/c,
                  rc/;.. })
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
rc = (varxy/SQRT(varx*vary)) }});

OUTPUT(%stats%, 'bestFit: y=' +
   (STRING)((sy-sx*varxy/varx)/c) +
   ' + ' +
   (STRING)(varxy/varx)+'x' ));
ENDMACRO;

ds1 := DATASET([ {1,1},{2,2},{3,3},{4,4},{5,5},{6,6} ], pointRec);
ds2 := DATASET([ {1.93896e+009, 2.04482e+009},
   {1.77971e+009, 8.54858e+008},
   {2.96181e+009, 1.24848e+009},
   {2.7744e+009, 1.26357e+009},
   {1.4416e+009, 4.3429e+008},
   {3.38728e+009, 1.30238e+009},
   {3.19538e+009, 1.71177e+009} ], pointRec);
ds3 := DATASET([ {1, 1.00039},
   {2, 2.07702},
   {3, 2.86158},
   {4, 3.87114},
   {5, 5.12417},
   {6, 6.20283} ], pointRec);
analyze(ds1);
analyze(ds2);
analyze(ds3);

See Also: VARIANCE, CORRELATION
CRON

<table>
<thead>
<tr>
<th>CRON( time )</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
</tr>
<tr>
<td>Return:</td>
</tr>
</tbody>
</table>

The CRON function defines a timer event for use within the WHEN workflow service or WAIT function. This is synonymous with EVENT('CRON', time).

The time parameter is unix-standard cron time, expressed in UTC (aka Greenwich Mean Time) as a string containing the following, space-delimited components:

minute hour dom month dow

- **minute** An integer value for the minute of the hour. Valid values are from 0 to 59.
- **hour** An integer value for the hour. Valid values are from 0 to 23 (using the 24 hour clock).
- **dom** An integer value for the day of the month. Valid values are from 1 to 31.
- **month** An integer value for the month. Valid values are from 1 to 12.
- **dow** An integer value for the day of the week. Valid values are from 0 to 6 (where 0 represents Sunday).

Any time component that you do not want to pass is replaced by an asterisk (*). You may define ranges of times using a dash (-), lists using a comma (,), and ‘once every n’ using a slash (/). For example, 6-18/3 in the hour field will fire the timer every three hours between 6am and 6pm, and 18-21/3,0-6/3 will fire the timer every three hours between 6pm and 6am.

Example:

```ecl
EXPORT events := MODULE
  EXPORT dailyAtMidnight := CRON('0 0 * * *');
  EXPORT dailyAt( INTEGER hour,
    INTEGER minute=0 ) :=
    EVENT('CRON',
      (STRING)minute + ' ' + (STRING)hour + ' * * *');
  EXPORT dailyAtMidday := dailyAt(12, 0);
  EXPORT EveryThreeHours := CRON('0 0-23/3 * * *');
END;

BUILD(teenagers) : WHEN(events.dailyAtMidnight);
BUILD(oldies)    : WHEN(events.dailyAt(6));
BUILD(NewStuff)  : WHEN(events.EveryThreeHours);
```

See Also: EVENT, WHEN, WAIT, NOTIFY
**DEDUP**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDUP(recordset [, condition [[MANY], ALL[, HASH]] [,BEST (sort-list)][, KEEP n] [, keeper ]][, LOCAL] [, UNORDERED</td>
<td>ORDERED( bool )] [, STABLE</td>
</tr>
<tr>
<td>recordset</td>
<td>The set of records to process, typically sorted in the same order that the expression will test. This may be the name of a dataset or derived record set, or any expression that results in a derived record set.</td>
</tr>
<tr>
<td>condition</td>
<td>Optional. A comma-delimited list of expressions or key fields in the recordset that defines &quot;duplicate&quot; records. The keywords LEFT and RIGHT may be used as dataset qualifiers for fields in the recordset. If the condition is omitted, every recordset field becomes the match condition. You may use the keyword RECORD (or WHOLE RECORD) to indicate all fields in that structure, and/or you may use the keyword EXCEPT to list non-dedup fields in the structure.</td>
</tr>
<tr>
<td>MANY</td>
<td>Optional. Specifies or perform a local sort/dedup before finding duplicates globally. This is most useful when many duplicates are expected.</td>
</tr>
<tr>
<td>ALL</td>
<td>Optional. Matches the condition against all records, not just adjacent records. This option may change the output order of the resulting records.</td>
</tr>
<tr>
<td>HASH</td>
<td>Optional. Specifies the ALL operation is performed using hash tables.</td>
</tr>
<tr>
<td>BEST</td>
<td>Optional. Provides additional control over which records are retained from a set of &quot;duplicate&quot; records. The first in the &lt;sort-list&gt; order of records are retained. BEST cannot be used with a KEEP parameter greater than 1.</td>
</tr>
<tr>
<td>sort-list</td>
<td>A comma delimited list of fields defining the duplicate records to keep. The fields may be prefixed with a minus sign to require a reverse sort on that field.</td>
</tr>
<tr>
<td>KEEP</td>
<td>Optional. Specifies keeping n number of duplicate records. If omitted, the default behavior is to KEEP 1. Not valid with the ALL option present.</td>
</tr>
<tr>
<td>n</td>
<td>The number of duplicate records to keep. If keeper is set to RIGHT, the only valid number of duplicate records to keep is 1.</td>
</tr>
<tr>
<td>keeper</td>
<td>Optional. The keywords LEFT or RIGHT. LEFT (the default, if omitted) keeps the first record encountered and RIGHT keeps the last.</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>
Return:

DEDUP returns a set of records.

The **DEDUP** function evaluates the *recordset* for duplicate records, as defined by the *condition* parameter, and returns a unique return set. This is similar to the DISTINCT statement in SQL. The *recordset* should be sorted, unless ALL is specified.

If a *condition* parameter is a single value (*field*), DEDUP does a simple field-level de-dupe equivalent to *LEFT.* The *condition* is evaluated for each pair of adjacent records in the record set. If the *condition* returns TRUE, the *keeper* record is kept and the other removed.

The **ALL** option means that every record pair is evaluated rather than only those pairs adjacent to each other, irrespective of sort order. The evaluation is such that, for records 1, 2, 3, 4, the record pairs that are compared to each other are:

(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,1),(4,2),(4,3)

This means two compares happen for each pair, allowing the *condition* to be non-commutative.

**KEEP** *n* effectively means leaving *n* records of each duplicate type. This is useful for sampling. The **LEFT** *keeper* value (implicit if neither LEFT nor RIGHT are specified) means that if the left and right records meet the de-dupe criteria (that is, they "match"), the left record is kept. If the **RIGHT** *keeper* appears instead, the right is kept. In both cases, the next comparison involves the de-dupe survivor; in this way, many duplicate records can collapse into one.

The **BEST** option provides additional control over which records are retained from a set of "duplicate" records. The first in the *sort-list* order of records are retained. The *sort-list* is comma delimited list of fields. The fields may be prefixed with a minus sign to require a reverse sort on that field.

**DEDUP**(recordset, field1, **BEST**(field2) ) means that from set of duplicate records, the first record from the set duplicates sorted by field2 is retained. **DEDUP**(recordset, field1, **BEST**(-field2) ) produces the last record sorted by field2 from the set of duplicates.

The **BEST** option cannot be used with a KEEP parameter greater than 1.

Example:

```ecl
SomeFile := DATASET([{'001','KC','G'},
{'002','KC','Z'},
{'003','KC','Z'},
{'004','KC','C'},
{'005','KA','X'},
{'006','KB','A'},
{'007','KB','G'},
{'008','KA','B'}],[STRING3 Id, String2 Value1, String1 Value2]);
SomeFile1 := SORT(SomeFile, Value1);
DEDUP(SomeFile1, Value1, **BEST**(Value2));
// Output:
// id value1 value2
// 008 KA B
// 006 KB A
// 004 KC C
DEDUP(SomeFile1, Value1, **BEST**(-Value2));
// Output:
// id value1 value2
// 005 KA X
// 007 KB G
// 002 KC Z
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
Complex Record Set Conditions

The DEDUP function with the ALL option is useful in determining complex recordset conditions between records in the same recordset. Although DEDUP is traditionally used to eliminate duplicate records next to each other in the recordset, the conditional expression combined with the ALL option extends this capability. The ALL option causes each record to be compared according to the conditional expression to every other record in the recordset. This capability is most effective with small recordsets; larger recordsets should also use the HASH option.

Example:

```
LastTbl := TABLE(Person,(per_last_name));
Lasts := SORT(LastTbl,per_last_name);
MySet := DEDUP(Lasts,per_last_name);
    // unique last names -- this is exactly equivalent to:
    //MySet := DEDUP(Lasts,LEFT.per_last_name=RIGHT.per_last_name);
    // also exactly equivalent to:
    //MySet := DEDUP(Lasts);
NamesTbl1 := TABLE(Person,(per_last_name,per_first_name));
Names1 := SORT(NamesTbl1,per_last_name,per_first_name);
MyNames1 := DEDUP(Names1,RECORD);
    // dedup by all fields -- this is exactly equivalent to:
    //MyNames1 := DEDUP(Names1,per_last_name,per_first_name);
    // also exactly equivalent to:
    //MyNames1 := DEDUP(Names1);
NamesTbl2 := TABLE(Person,(per_last_name,per_first_name,per_sex));
Names2 := SORT(NamesTbl,per_last_name,per_first_name);
MyNames2 := DEDUP(Names,RECORD, EXCEPT per_sex);
    // dedup by all fields except per_sex
    // this is exactly equivalent to:
    //MyNames2 := DEDUP(Names, EXCEPT per_sex);
/* In the following example, we want to determine how many 'AN' or 'AU' type inquiries
have occurred within 3 days of a 'BB' type inquiry. The COUNT of inquiries in the deduped recordset is subtracted from the COUNT of the inquiries in the original recordset to provide the result. */
INTEGER abs(INTEGER i) := IF ( i < 0, -i, i );
WithinDays(ldrpt,lday,rdrpt,rday,days) :=
    abs(DaysAgo(ldrpt,lday)-DaysAgo(rdrpt,rday)) <= days;
DedupedInqs := DEDUP(inquiry, LEFT.inq_ind_code='BB' AND
    RIGHT.inq_ind_code IN ['AN','AU'] AND
    WithinDays(LEFT.inq_drpt,
        LEFT.inq_drpt_day,
        RIGHT.inq_drpt,
        RIGHT.inq_drpt_day,3),
    ALL);
InqCount := COUNT(Inquiry) - COUNT(DedupedInqs);
OUTPUT(person(InqCount >0),{InqCount});
```

See Also: SORT, ROLLUP, TABLE, FUNCTION Structure
**DEFINE**

**DEFINE (pattern, symbol)**

<table>
<thead>
<tr>
<th>pattern</th>
<th>The name of a RULE parsing pattern.</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol</td>
<td>A string constant specifying the name to use in the USE option on a PARSE function or the USE function in a RULE parsing pattern.</td>
</tr>
</tbody>
</table>

**Return:** DEFINE creates a RULE pattern.

The **DEFINE** function defines a symbol for the specified pattern that may be forward referenced in previously defined parsing pattern attributes. This is the only type of forward reference allowed in ECL.

Example:

```ecl
RULE a := USE('symbol');
  // uses the 'symbol' pattern defined later - b
RULE b := 'pattern';
  // defines a rule pattern
RULE s := DEFINE(b,'symbol');
  // associate the "b" rule with the
  // 'symbol' for forward reference by rule "a"
```

See Also: PARSE, PARSE Pattern Value Types
**DENORMALIZE**

The **DENORMALIZE** function is used to form a combined record out of a parent and any number of children. It acts very similar to a JOIN except that where JOIN with one parent and three children would call the `transform` three times and produce three outputs, DENORMALIZE calls the `transform` three times where the input to the first `transform` is the parent and one child, the input to the second `transform` is the output of the first `transform` and another child, and the input to the third `transform` is the output from the second `transform` and the remaining child. Also like JOIN, the order in which the `childrecset` records are sent to the `transform` is undefined.

Because **DENORMALIZE** is basically a specialized form of JOIN, the various join types (LEFT OUTER, RIGHT OUTER, FULL OUTER, LEFT ONLY, RIGHT ONLY, FULL ONLY) are also available for use on **DENORMALIZE** and act just as they do with JOIN.

---

**parentrecset**  The set of parent records to process, already in the format that will contain the denormalized parent and child records.

**childrecset**  The set of child records to process.

**condition**  An expression that specifies how to match records between the `parentrecset` and `childrecset`.

**transform**  The TRANSFORM function to call.

**LOCAL**  Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.

**NOSORT**  Optional. Specifies the operation is performed without sorting the `parentrecset` or `childrecset` --both must already be sorted so matching records in both are in order. This allows programmer control of the order of the child records.

**GROUP**  Specifies grouping the `childrecset` records based on the join condition so all the related child records are passed as a dataset parameter to the `transform`.

**UNORDERED**  Optional. Specifies the output record order is not significant.

**ORDERED**  Specifies the significance of the output record order.

**bool**  When False, specifies the output record order is not significant. When True, specifies the default output record order.

**STABLE**  Optional. Specifies the input record order is significant.

**UNSTABLE**  Optional. Specifies the input record order is not significant.

**PARALLEL**  Optional. Try to evaluate this activity in parallel.

**numthreads**  Optional. Try to evaluate this activity using `numthreads` threads.

**ALGORITHM**  Optional. Override the algorithm used for this activity.

**name**  The algorithm to use for this activity. Must be from the list of supported algorithms for the **SORT** function's **STABLE** and **UNSTABLE** options.

---

Return:  **DENORMALIZE** returns a record set.
All JOIN options are available for DENORMALIZE. See Join Options for details.

**DENORMALIZE TRANSFORM Function Requirements**

For form one, the *transform* function must take at least two parameters: a LEFT record of the same format as the combined *parentrecset* and *childrecset* (the resulting de-normalized record structure), and a RIGHT record of the same format as the *childrecset*. An optional third parameter may be specified: an integer COUNTER specifying the number of times the *transform* has been called for the current set of parent/child pairs (defined by the condition values). The result of the *transform* function must be a record set of the same format as the LEFT record.

For form two, the *transform* function must take at least two parameters: a LEFT record of the same format as the combined *parentrecset* and *childrecset* (the resulting de-normalized record structure), and ROWS(RIGHT) dataset of the same format as the *childrecset*. The result of the *transform* function must be a record set of the same format as the LEFT record.

**Example:**

**Form 1 example:**

```ecl
NormRec := RECORD
  STRING20  thename;
  STRING20  addr;
END;
NamesRec := RECORD
  UNSIGNED1  numRows;
  STRING20  thename;
  STRING20  addr1 := '';
  STRING20  addr2 := '';
  STRING20  addr3 := '';
  STRING20  addr4 := '';
END;
NamesTable := DATASET([ {0,'Kevin'},{0,'Liz'},{0,'Mr Nobody'},
  {0,'Anywhere'}], NamesRec);
NormAddrs := DATASET([{'Kevin','10 Malt Lane'},
  {'Liz','10 Malt Lane'},
  {'Liz','3 The cottages'},
  {'Anywhere','Here'},
  {'Anywhere','There'},
  {'Anywhere','Near'},
  {'Anywhere','Far'}],NormRec);
NamesRec DeNormThem(NamesRec L, NormRec R, INTEGER C) := TRANSFORM
  SELF.NumRows := C;
  SELF.addr1 := IF (C=1, R.addr, L.addr1);
  SELF.addr2 := IF (C=2, R.addr, L.addr2);
  SELF.addr3 := IF (C=3, R.addr, L.addr3);
  SELF.addr4 := IF (C=4, R.addr, L.addr4);
  SELF := L;
END;
DeNormedRecs := DENORMALIZE(NamesTable, NormAddrs,
  LEFT.thename = RIGHT.thename,
  DeNormThem(LEFT,RIGHT,COUNTER));
OUTPUT(DeNormedRecs);
```

**Form 2 example:**

```ecl
NormRec := RECORD
  STRING20  thename;
  STRING20  addr;
END;
NamesRec := RECORD
END;
NamesTable := DATASET([ {0,'Kevin'},{0,'Liz'},{0,'Mr Nobody'},
  {0,'Anywhere'}], NamesRec);
NormAddrs := DATASET([{'Kevin','10 Malt Lane'},
  {'Liz','10 Malt Lane'},
  {'Liz','3 The cottages'},
  {'Anywhere','Here'},
  {'Anywhere','There'},
  {'Anywhere','Near'},
  {'Anywhere','Far'}],NormRec);
NamesRec DeNormThem(NamesRec L, NormRec R, INTEGER C) := TRANSFORM
  SELF.NumRows := C;
  SELF.addr1 := IF (C=1, R.addr, L.addr1);
  SELF.addr2 := IF (C=2, R.addr, L.addr2);
  SELF.addr3 := IF (C=3, R.addr, L.addr3);
  SELF.addr4 := IF (C=4, R.addr, L.addr4);
  SELF := L;
END;
DeNormedRecs := DENORMALIZE(NamesTable, NormAddrs,
  LEFT.thename = RIGHT.thename,
  DeNormThem(LEFT,RIGHT,COUNTER));
OUTPUT(DeNormedRecs);
```
UNSIGNED1 numRows;
STRING20 thename;
DATASET(NormRec) addresses;
END;
NamesTable := DATASET([ {0,'Kevin',[]},{0,'Liz',[]},
    {0,'Mr Nobody',[]},{0,'Anywhere',[]}],
    NamesRec);
NormAddrs := DATASET([{'Kevin','10 Malt Lane'},
    {'Liz','10 Malt Lane'},
    {'Liz','3 The cottages'},
    {'Anywhere','Here'},
    {'Anywhere','There'},
    {'Anywhere','Near'},
    {'Anywhere','Far'}],NormRec);
NamesRec DeNormThem(NamesRec L, DATASET(NormRec) R) := TRANSFORM
    SELF.NumRows := COUNT(R);  
    SELF.addresses := R;  
    SELF := L;  
END;
DeNormedRecs := DENORMALIZE(NamesTable, NormAddrs,
    LEFT.thename = RIGHT.thename,
    GROUP,
    DeNormThem(LEFT,ROWS(RIGHT)));
OUTPUT(DeNormedRecs);

MyRec := RECORD
    STRING1 Value1;
    STRING1 Value2;
END;
ParentFile := DATASET([{'A','C'},{'B','B'},{'C','A'}],MyRec);
ChildFile  := DATASET([{'A','Z'},{'A','T'},{'B','S'},{'B','Y'},
    {'C','X'},{'C','W'}],MyRec);
MyOutRec := RECORD
    ParentFile.Value;
    ParentFile.Value2;
    STRING1 CVal2_1 := '';  
    STRING1 CVal2_2 := '';  
END;
P_Recs := TABLE(ParentFile, MyOutRec);
MyOutRec DeNormThem(MyOutRec L, MyRec R, INTEGER C) := TRANSFORM
    SELF.CVal2_1 := IF(C = 1, R.Value2, L.CVal2_1);
    SELF.CVal2_2 := IF(C = 2, R.Value2, L.CVal2_2);
    SELF := L;
END;
DeNormedRecs := DENORMALIZE(P_Recs, ChildFile,
    LEFT.Value1 = RIGHT.Value1,
    DeNormThem(LEFT,RIGHT,COUNTER),NOSORT);
OUTPUT(DeNormedRecs);
/* DeNormedRecs result set is:
Rec# Value1 PVal2 CVal2_1 CVal2_2
1   A   C   Z   T
2   B   B   S   Y
3   C   A   X   W
*/

See Also: JOIN, TRANSFORM Structure, RECORD Structure, NORMALIZE
DISTRIBUTE

DISTRIBUTE(recordset [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [( numthreads ) ] [, ALGORITHM( name ) ] )

DISTRIBUTE(recordset, expression [, MERGE( sorts ) ] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [( numthreads ) ] [, ALGORITHM( name ) ] )

DISTRIBUTE(recordset, index [, joincondition ] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [( numthreads ) ] [, ALGORITHM( name ) ] )

DISTRIBUTE(recordset, SKEW( maxskew [, skewlimit ] ) [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [( numthreads ) ] [, ALGORITHM( name ) ] )

| recordset   | The set of records to distribute. |
| expression  | An integer expression that specifies how to distribute the recordset, usually using one of the HASH functions for efficiency. |
| MERGE       | Specifies the data is redistributed maintaining the local sort order on each node. |
| sorts       | The sort expressions by which the data has been locally sorted. |
| index       | The name of an INDEX attribute definition, which provides the appropriate distribution. |
| joincondition | Optional. A logical expression that specifies how to link the records in the recordset and the index. The keywords LEFT and RIGHT may be used as dataset qualifiers for fields in the recordset and index. |
| SKEW        | Specifies the allowable data skew values. |
| maxskew     | A floating point number in the range of zero (0.0) to one (1.0) specifying the minimum skew to allow (0.1=10%). |
| skewlimit   | Optional. A floating point number in the range of zero (0.0) to one (1.0) specifying the maximum skew to allow (0.1=10%). |
| UNORDERED   | Optional. Specifies the output record order is not significant. |
| ORDERED     | Specifies the significance of the output record order. |
| bool        | When False, specifies the output record order is not significant. When True, specifies the default output record order. |
| STABLE      | Optional. Specifies the input record order is significant. |
| UNSTABLE    | Optional. Specifies the input record order is not significant. |
| PARALLEL    | Optional. Try to evaluate this activity in parallel. |
| numthreads  | Optional. Try to evaluate this activity using numthreads threads. |
| ALGORITHM   | Optional. Override the algorithm used for this activity. |
| name        | The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options. |

Return: DISTRIBUTE returns a set of records.

The DISTRIBUTE function re-distributes records from the recordset across all the nodes of the cluster.

"Random" DISTRIBUTE

DISTRIBUTE(recordset)
This form redistributes the *recordset* “randomly” so there is no data skew across nodes, but without the disadvantages the RANDOM() function could introduce. This is functionally equivalent to distributing by a hash of the entire record.

**Expression DISTRIBUTE**

\[
\text{DISTRIBUTE}(\text{recordset}, \text{expression})
\]

This form redistributes the *recordset* based on the specified *expression*, typically one of the HASH functions. Only the bottom 32-bits of the *expression* value are used, so either HASH or HASH32 are the optimal choices. Records for which the *expression* evaluates the same will end up on the same node. DISTRIBUTE implicitly performs a modulus operation if an *expression* value is not in the range of the number of nodes available.

If the MERGE option is specified, the *recordset* must have been locally sorted by the sorts expressions. This avoids resorting.

**Index-based DISTRIBUTE**

\[
\text{DISTRIBUTE}(\text{recordset}, \text{index} [, \text{joincondition}])
\]

This form redistributes the *recordset* based on the existing distribution of the specified *index*, where the linkage between the two is determined by the *joincondition*. Records for which the *joincondition* is true will end up on the same node.

**Skew-based DISTRIBUTE**

\[
\text{DISTRIBUTE}(\text{recordset}, \text{SKEW( maxskew [, skewlimit] )})
\]

This form redistributes the *recordset*, but only if necessary. The purpose of this form is to replace the use of DISTRIBUTE(*recordset*, RANDOM()) to simply obtain a relatively even distribution of data across the nodes. This form will always try to minimize the amount of data redistributed between the nodes.

The skew of a dataset is calculated as:

\[
\text{MAX}(\text{ABS}(\text{AvgPartSize-PartSize[node]})/\text{AvgPartSize})
\]

If the *recordset* is skewed less than *maxskew* then the DISTRIBUTE is a no-op. If *skewlimit* is specified and the skew on any node exceeds this, the job fails with an error message (specifying the first node number exceeding the limit), otherwise the data is redistributed to ensure that the data is distributed with less skew than *maxskew*.

Example:

```ecl
MySet1 := DISTRIBUTE(Person); //"random" distribution - no skew
MySet2 := DISTRIBUTE(Person,HASH32(Person.per_ssn));
//all people with the same SSN end up on the same node
//INDEX example:
mainRecord := RECORD
    INTEGER8 sequence;
    STRING20 forename;
    STRING20 surname;
    UNSIGNED8 filepos{VIRTUAL(fileposition)};
END;
mainTable := DATASET('~keyed.d00',mainRecord,THOR);
nameKey := INDEX(mainTable, {surname,forename,filepos}, 'name.idx');
incTable := DATASET('~inc.d00',mainRecord,THOR);
```
x := DISTRIBUTE(incTable, nameKey,
  LEFT.surname = RIGHT.surname AND
  LEFT.forename = RIGHT.forename);
OUTPUT(x);

//SKEW example:
Jds := JOIN(somedata,otherdata,LEFT.sysid=RIGHT.sysid);
Jds_dist1 := DISTRIBUTE(Jds,SKEW(0.1));
  //ensures skew is less than 10%
Jds_dist2 := DISTRIBUTE(Jds,SKEW(0.1,0.5));
  //ensures skew is less than 10%
  //and fails if skew exceeds 50% on any node

See Also: HASH32, DISTRIBUTED, INDEX
The DISTRIBUTED function is a compiler directive indicating that the records from the `recordset` are already distributed across the nodes of the Data Refinery based on the specified `expression`. Records for which the `expression` evaluates the same are on the same node.

If the `expression` is omitted, the function just suppresses a warning that is sometimes generated that the `recordset` hasn't been distributed.

Example:

```ecl
MySet := DISTRIBUTED(Person,HASH32(Person.per_ssn));
// all people with the same SSN are already on the same node
```

See Also: HASH32, DISTRIBUTE
DISTRIBUTION action produces a crosstab report in XML format indicating how many unique records there are in the `recordset` for each value in each field in the `recordset`.

When there is an excessively large number of distinct values, it returns an estimate in this form:

```xml
<XML>
  <Field name="seqnum" estimate="4000000"/>
</XML>
```

The DECIMAL data type is not supported by this action. You can use a REAL data type instead.

Example:

```ecl
SomeFile := DATASET([{'C','G'},{'C','C'},{'A','X'},{'B','G'}],
  {STRING1 Value1,STRING1 Value2});
DISTRIBUTION(SomeFile);
/* The result comes back looking like this: */
<XML>
  <Field name="Value1" distinct="3">
    <Value count="1">A</Value>
    <Value count="1">B</Value>
    <Value count="2">C</Value>
  </Field>
  <Field name="Value2" distinct="3">
    <Value count="1">C</Value>
    <Value count="2">G</Value>
    <Value count="1">X</Value>
  </Field>
</XML>
```
/*
//**************************************************
namesRecord := RECORD
    STRING20 surname;
    STRING10 forename;
    INTEGER2 age;
END;

namesTable := DATASET([
    {'Halligan','Kevin',31},
    {'Halligan','Liz',30},
    {'Salter','Abi',10},
    {'X','Z',5}], namesRecord);

doFirst := DISTRIBUTION(namesTable, surname, forename, NAMED('Stats'));
/* The result comes back looking like this:
<XML>
<Field name="surname" distinct="3">
    <Value count="2">Halligan</Value>
    <Value count="1">X</Value>
    <Value count="1">Salter</Value>
</Field>
<Field name="forename" distinct="4">
    <Value count="1">Abi</Value>
    <Value count="1">Kevin</Value>
    <Value count="1">Liz</Value>
</Field>
</XML>
*/

//Post-processing the result with PARSE:
x := DATASET(ROW(TRANSFORM({STRING line},
    SELF.line := WORKUNIT('Stats', STRING))));
res := RECORD
    STRING Fieldname := XMLTEXT('@name');
    STRING Cnt := XMLTEXT('@distinct');
END;

DoNext := PARSE(x, line, res, XML('XML/Field'));
SEQUENTIAL(DoFirst,OUTPUT(DoNext));
### EBCDIC

**EBCDIC**

```
EBCDIC(recordset[, UNORDERED | ORDERED(bool) [, STABLE | UNSTABLE [, PARALLEL [(numthreads)]] [, ALGORITHM(name)]])
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>recordset</code></td>
<td>The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.</td>
</tr>
<tr>
<td><code>UNORDERED</code></td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td><code>ORDERED</code></td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><code>STABLE</code></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td><code>UNSTABLE</code></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><code>PARALLEL</code></td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td><code>numthreads</code></td>
<td>Optional. Try to evaluate this activity using <code>numthreads</code> threads.</td>
</tr>
<tr>
<td><code>ALGORITHM</code></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

**Return:**

EBCDIC returns a set of records

---

The **EBCDIC** function returns the `recordset` with all STRING fields translated from ASCII to EBCDIC.

**Example:**

```ecl
EBCDICRecs := EBCDIC(SomeASCIIInput);
```

See Also: ASCII
ENTH

**ENTH**(*recordset*, *numerator* [*, *denominator* [*, *which* ]] [*, *LOCAL*] [*, *UNORDERED | ORDERED*(*bool*)] [*, *STABLE | UNSTABLE*] [*, *PARALLEL*(*numthreads*)] [*, *ALGORITHM*(*name*)])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>recordset</em></td>
<td>The set of records to sample. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.</td>
</tr>
<tr>
<td><em>numerator</em></td>
<td>The number of records to return. The chosen records are evenly spaced from throughout the recordset.</td>
</tr>
<tr>
<td><em>denominator</em></td>
<td>Optional. The size of each set from which to return numerator number of records. If omitted, the denominator value is the total number of records in the recordset.</td>
</tr>
<tr>
<td><em>which</em></td>
<td>Optional. An integer specifying the ordinal number of the sample set to return. This is used to obtain multiple non-overlapping samples from the same recordset. If the numerator is not 1, then some records may overlap.</td>
</tr>
<tr>
<td><em>LOCAL</em></td>
<td>Optional. Specifies that the sample is extracted on each supercomputer node without regard to the number of records on other nodes, significantly improving performance if exact results are not required.</td>
</tr>
<tr>
<td><em>UNORDERED</em></td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td><em>ORDERED</em></td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td><em>bool</em></td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><em>STABLE</em></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td><em>UNSTABLE</em></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><em>PARALLEL</em></td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td><em>numthreads</em></td>
<td>Optional. Try to evaluate this activity using <em>numthreads</em> threads.</td>
</tr>
<tr>
<td><em>ALGORITHM</em></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><em>name</em></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

**Return:** ENTH returns a set of records.

The **ENTH** function returns a sample set of records from the nominated *recordset*. ENTH returns *numerator* number of records out of each *denominator* set of records in the *recordset*. Unless LOCAL is specified, records are picked at exact intervals across all nodes of the supercomputer.

**Example:**

```ecl
SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'},
{ 'F'},{'G'},{'H'},{'I'},{'J'},
{ 'K'},{'L'},{'M'},{'N'},{'O'},
{ 'P'},{'Q'},{'R'},{'S'},{'T'},
{ 'U'},{'V'},{'W'},{'X'},{'Y'}],
{STRING1 Letter});
Set1 := ENTH(SomeFile,2,10,1); // returns E, J, O, T, Y

// other examples
// MySample1 := ENTH(Person,1,10,1); // 10% (1 out of every 10)
// MySample2 := ENTH(Person,15,100,1); // 15% (15 out of every 100)
// MySample3 := ENTH(Person,3,4,1); // 75% (3 out of every 4)
```

See Also: CHOOSE, SAMPLE
ERROR

ERROR [ ( errormessage | errorcode ) ];

ERROR ( errorcode , errormessage );

ERROR( datatype [, [ errorcode ][, errormessage ] ] );

<table>
<thead>
<tr>
<th>errormessage</th>
<th>Optional. A string constant containing the message to display.</th>
</tr>
</thead>
<tbody>
<tr>
<td>errorcode</td>
<td>Optional. An integer constant containing the error number to display.</td>
</tr>
<tr>
<td>datatype</td>
<td>The value type or name of a RECORD structure. This may use the TYPEOF function.</td>
</tr>
</tbody>
</table>

The `ERROR` function immediately halts processing on the workunit and displays the `errorcode` and/or `errormessage`. The third form is available for use in contexts where a value type or dataset is required. This function does the same thing as the `FAIL` action, but may be used in an expression context, such as within a `TRANSFORM` function.

Example:

```ecl
outrec Xform(inrec L, inrec R) := TRANSFORM
   SELF.key := IF(L.key <= R.key, R.key,ERROR('Recs not in order'));
END;
```

See Also: `FAILURE`, `FAIL`
EVALUATE

EVALUATE action

\[ \text{attrname} := \] EVALUATE(\text{expression}) ;

\[ \text{attrname} := \] EVALUATE(\text{module}[, \text{defname} ]) ;

<table>
<thead>
<tr>
<th>attrname</th>
<th>Optional. The action name, which turns the action into a definition, therefore not executed until the attrname is used as an action.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The function to call in an action context.</td>
</tr>
<tr>
<td>module</td>
<td>The module to evaluate.</td>
</tr>
<tr>
<td>defname</td>
<td>Optional. The name of a specific definition within the module to evaluate. If omitted, all definitions in the module are evaluated.</td>
</tr>
</tbody>
</table>

The first form of the EVALUATE action names an expression (typically a function call) to execute in an action context. This is mainly useful for calling functions that have side-effects, where you don’t care about the return value.

The second form of the EVALUATE action recursively expands the exported definitions of the module and evaluates them. If a defname is specified, then only that definition is evaluated.

Example:

Form 1 example:

```ecl
myService := SERVICE
   UNSIGNED4 doSomething(STRING text);
END;

ds := DATASET('MyFile', {STRING20 text}, THOR);
APPLY(ds, EVALUATE(doSomething(ds.text)));
// calls the doSomething function once for each record in the ds dataset, ignoring the returned values from the function
```

Form 2 example:

```ecl
M := MODULE
   EXPORT a := OUTPUT(10);
   EXPORT b := OUTPUT('Hello');
END;

M2 := MODULE
   EXPORT mx := M;
   EXPORT d := OUTPUT('Richard');
END;

EVALUATE(M2);
// produces three results:
// Result_1: 10
// Result_2: Hello
// Result_3: Richard
```

See Also: APPLY, SERVICE Structure,
**EVALUATE function**

**EVALUATE**(onerecord, value)

| onerecord | A record set consisting of a single record. |
| value     | The value to return. This may be any expression yielding a value. |

Return: EVALUATE returns a single value.

The `EVALUATE` function returns the `value` evaluated in the context of the `onerecord` set (which must be a single record, only). This function typically uses indexing to select a single record for the `onerecord` record-set. The usage is to return a value from a specific child record when operating at the parent record's scope level. The advantage that `EVALUATE` has over using recordset indexing into a single field is that the `value` returned can be any expression and not just a single field from the child dataset.

### Accessing Field-level Data in a Specific Record

To access field level data in a specific record, the recordset indexing capability must be used to select a single record. The `SORT` function and recordset filters are useful in selecting and ordering the recordset so that the appropriate record can be selected.

**Example:**

```ecl
TrdRec := RECORD
  UNSIGNED2 Trd_ID;
  INTEGER4 Trd_bal;
  INTEGER4 Trd_hc;
END;
PersonRec := RECORD
  STRING20 FirstName;
  STRING20 LastName;
  DATASET(TrdRec) Trd;
END;
Person := DATASET([{'Fred','Jones',
  [{1,2,3},{2,4,5}]}],PersonRec);
Trades := Person.Trd;
ValidMoney(n) := n > 0;
ValidBalTrades := trades(ValidMoney(trades.trd_bal));
HighestBals := SORT(ValidBalTrades,-trades.trd_bal);
Highest_HC := EVALUATE(HighestBals[1],trades.trd_hc);
//return trd_hc field of the trade with the highest balance
// could also be coded as (using indexing):
// Highest_HC := HighestBals[1].trades.trd_hc;
OUTPUT(Person,{lastname,firstname,Highest_HC});
//output that Highest_HC for each person
//This output operates at the scope of the Person record
// EVALUATE is needed to get the value from a Trades record
// because Trades is a Child of Person
```

See Also: `SORT`
**EVENT**

**EVENT** (*event*, *subtype*)

<table>
<thead>
<tr>
<th><strong>event</strong></th>
<th>A case-insensitive string constant naming the event to trap.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>subtype</strong></td>
<td>A case-insensitive string constant naming the specific type of event to trap. This may contain * and ? to wildcard-match the event's sub-type.</td>
</tr>
</tbody>
</table>

**Return:** EVENT returns a single event.

The EVENT function returns a trigger event, which may be used within the WHEN workflow service or the WAIT and NOTIFY actions.

**Example:**

```
IMPORT STD;
MyEventName := 'MyFileEvent';
MyFileName := 'test::myfile';

IF (STD.File.FileExists(MyFileName),
   STD.File.DeleteLogicalFile(MyFileName));
   //deletes the file if it already exists

STD.File.MonitorLogicalFileName(MyEventName,MyFileName);
   //sets up monitoring and the event name
   //to fire when the file is found

OUTPUT('File Created') : WHEN(EVENT(MyEventName,'*'),COUNT(1));
   //this OUTPUT occurs only after the event has fired

afile := DATASET([{'A', '0'}], {STRING10 key, STRING10 val});
OUTPUT(afile,,MyFileName);
   //this creates a file that the DFU file monitor will find
   //when it periodically polls

//*****************************************************
EXPORT events := MODULE
   EXPORT dailyAtMidnight := CRON('0 0 * * *');
   EXPORT dailyAt( INTEGER hour, INTEGER minute=0) :=
      EVENT('CRON',
      (STRING)minute + ' ' + (STRING)hour + ' * * *');
   EXPORT dailyAtMidday := dailyAt(12, 0);
END;
BUILD(teenagers): WHEN(events.dailyAtMidnight);
BUILD(oldies)  : WHEN(events.dailyAtMid(6));
```

**See Also:** EVENTNAME, EVENTEXTRA, CRON, WHEN, WAIT, NOTIFY
EVENTNAME

EVENTNAME

Return: EVENTNAME returns a single string value.

EVENTNAME returns the name of the trigger event.

Example:

```ecl
doMyService := FUNCTION
    O := OUTPUT('Did a Service for: ' + 'EVENTNAME=' + EVENTNAME);
    N := NOTIFY(EVENT('MyServiceComplete',
                      '<Event><returnTo>' + WORKUNIT + '</returnTo></Event>'),
                   EVENTEXTRA('returnTo'));
    RETURN WHEN(EVENTEXTRA('returnTo'), ORDERED(O, N));
END;
OUTPUT(doMyService) : WHEN('MyService');
```

// and a call (in a separate workunit): 
NOTIFY('MyService',
       '<Event><returnTo'>+ WORKUNIT + '</returnTo></Event>');
WAIT('MyServiceComplete');
OUTPUT('WORKUNIT DONE')

See Also: EVENT, EVENTEXTRA, CRON, WHEN, WAIT, NOTIFY
EVENTEXTRA

EVENTEXTRA( tag )

Return: EVENTEXTRA returns a single string value.

The EVENTEXTRA function returns the contents of the tag from the XML text in the EVENT function's second parameter.

Example:

```ecl
doMyService := FUNCTION
  O := OUTPUT('Did a Service for: ' + 'EVENTNAME=' + EVENTNAME);
  N := NOTIFY(EVENT('MyServiceComplete',
    '<Event><returnTo>FRED</returnTo></Event>' ),
    EVENTEXTRA('returnTo'));
  RETURN WHEN(EVENTEXTRA('returnTo'),ORDERED(O,N));
END;
OUTPUT(doMyService) : WHEN('MyService');

// and a call (in a separate workunit):
NOTIFY('MyService',
  '<Event><returnTo>'+ WORKUNIT + '</returnTo></Event>');
WAIT('MyServiceComplete');
OUTPUT('WORKUNIT DONE')
```

See Also: EVENT, EVENTNAME, CRON, WHEN, WAIT, NOTIFY
EXISTS


EXISTS( valuelist )

| recordset  | The set of records to process. This may be the name of an index, a dataset, or a record set derived from some filter condition, or any expression that results in a derived record set. |
| KEYED      | Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation. |
| valuelist  | A comma-delimited list of expressions. This may also be a SET of values. |
| UNORDERED  | Optional. Specifies the output record order is not significant. |
| ORDERED    | Specifies the significance of the output record order. |
| bool       | When False, specifies the output record order is not significant. When True, specifies the default output record order. |
| STABLE     | Optional. Specifies the input record order is significant. |
| UNSTABLE   | Optional. Specifies the input record order is not significant. |
| PARALLEL   | Optional. Try to evaluate this activity in parallel. |
| numthreads | Optional. Try to evaluate this activity using numthreads threads. |
| ALGORITHM  | Optional. Override the algorithm used for this activity. |
| name       | The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options. |

Return: EXISTS returns a single BOOLEAN value.

The EXISTS function returns true if the number of records in the specified recordset is > 0, or the valuelist is populated. This is most commonly used to detect whether a filter has filtered out all the records.

When checking for an empty recordset, use the EXISTS(recordset) function instead of the expression: COUNT(recordset) > 0. Using EXISTS results in more efficient processing and better performance under those circumstances.

Example:

```
TrdRec := RECORD
  UNSIGNED2 Trd_ID;
  INTEGER4 Trd_bal;
  INTEGER4 Trd_hc;
END;

PersonRec := RECORD
  STRING20 FirstName;
  STRING20 LastName;
  DATASET(TrdRec) Trd;
END;

Person := DATASET([[{'Noah','Brody',[]},
                   {'Fred','Jones',[[1,2,3],[2,4,5]]},
                   {'Sue','Smith',[[10,-2,60],[12,14,50]]},
                   {'Joe','Johnson',[[11,200,3000],[22,140,350],[25,100,850]]},
                   {'Susan','Stone',[[102,2,30],[125,14,50],[225,14000,50000]]}],PersonRec);
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
Trades := Person.Trd;

MyBoolean := EXISTS(Trades(Trd_ID = 15));
OUTPUT(MyBoolean); //returns FALSE

TradesExistPersons := Person(EXISTS(Trades));
OUTPUT(TradesExistPersons); //returns 4 records

NoTradesPerson := Person(NOT EXISTS(Trades));
OUTPUT(NoTradesPerson); //returns 1 record

SetExists1 := EXISTS(4,8,16,2,1);
OUTPUT(SetExists1); //returns TRUE

SetVals := [4,8,16,2,1];
SetExists2 := EXISTS(SetVals);
OUTPUT(SetExists2); //returns TRUE

EmptySet := [];
SetExists3 := EXISTS(EmptySet);
OUTPUT(SetExists3); //returns FALSE

See Also: DEDUP, Record Filters
EXP

EXP (n)

<table>
<thead>
<tr>
<th>n</th>
<th>The real number to evaluate.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>EXP returns a single real value.</td>
</tr>
</tbody>
</table>

The **EXP** function returns the natural exponential value of the parameter (en). This is the opposite of the LN function.

Example:

```ecl
MyPI := EXP(3.14159);
Interim := ROUND(1000 * (EXP(MyPI)/(1 + EXP(MyPI))));
OUTPUT(Interim);
```

See Also: LN, SQRT, POWER
FAIL

[attrname := ] FAIL [ ( errormessage | errorcode ) ] ;

[attrname := ] FAIL( errorcode , errormessage ) ;


<table>
<thead>
<tr>
<th>attrname</th>
<th>Optional. The action name, which turns the action into an attribute definition, therefore not executed until the attrname is used as an action.</th>
</tr>
</thead>
<tbody>
<tr>
<td>errormessage</td>
<td>Optional. A string constant containing the message to display.</td>
</tr>
<tr>
<td>errorcode</td>
<td>Optional. An integer constant containing the error number to display.</td>
</tr>
<tr>
<td>datatype</td>
<td>The value type, name of a RECORD structure, DATASET, or DICTIONARY to emulate.</td>
</tr>
</tbody>
</table>

The FAIL action immediately halts processing on the workunit and displays the errorcode and/or errormessage. The third form is available for use in contexts where a value type or dataset is required. FAIL may not be used in an expression context (such as within a TRANSFORM)--use the ERROR function for those situations.

Example:

```ecl
MyRec := RECORD
  STRING50 Value1;
  UNSIGNED Value2;
END;

ds := DATASET([{'C',1},{'C',2},{'C',3},
                {'C',4},{'C',5},{'X',1},{'A',1} ],MyRec);

MyRec FailTransform := TRANSFORM
  self.value1 := FAILMESSAGE[1..17];
  self.value2 := FAILCODE
END;

limited1 := LIMIT(ds, 2);
limited2 := LIMIT(ds, 3);
limited3 := LIMIT(ds, 4);

recovered1 := CATCH(limited1, SKIP);
recovered2 := CATCH(limited2, ONFAIL(FailTransform));
recovered3 := CATCH(CATCH(limited3, FAIL(1, 'Failed, sorry')), ONFAIL(FailTransform));

OUTPUT(recovered1);  //empty recordset
OUTPUT(recovered2);  //
OUTPUT(recovered3);  //
```

See Also: FAILURE, ERROR
FAILCODE

The **FAILCODE** function returns the last failure code, for use in the FAILURE workflow service or in the TRANSFORM structure referenced in the ONFAIL option of SOAPCALL.

Example:

```ecl
MyRec := RECORD
  STRING50 Value1;
  UNSIGNED Value2;
END;

ds := DATASET([{'C',1},{'C',2},{'C',3},
                  {'C',4},{'C',5},{'X',1},{'A',1}],MyRec);

MyRec FailTransform := TRANSFORM
  self.value1 := FAILMESSAGE[1..17];
  self.value2 := FAILCODE
END;

limited1 := LIMIT(ds, 2);
limited2 := LIMIT(ds, 3);
limited3 := LIMIT(ds, 4);

recovered1 := CATCH(limited1, SKIP);
recovered2 := CATCH(limited2, ONFAIL(FailTransform));
recovered3 := CATCH(CATCH(limited3, FAIL(1, 'Failed, sorry')), ONFAIL(FailTransform));

OUTPUT(recovered1); //empty recordset
OUTPUT(recovered2); //
OUTPUT(recovered3); //
```

See Also: FAILURE, FAILMESSAGE, SOAPCALL
FAILMESSAGE

**FAILMESSAGE** [(tag)]

| tag     | A string constant defining the name of XML tag containing the text to return, typically extra information returned by SOAPCALL. If omitted, the default is 'text.' |

The `FAILMESSAGE` function returns the last failure message for use in the FAILURE workflow service or the TRANSFORM structure referenced in the ONFAIL option of SOAPCALL.

Example:

```ecl
MyRec ::= RECORD
    STRING50 Value1;
    UNSIGNED Value2;
END;

ds ::= DATASET([{'C',1},{'C',2},{'C',3},
    {'C',4},{'C',5},{'X',1},{'A',1}],MyRec);

MyRec FailTransform ::= TRANSFORM
    self.value1 := FAILMESSAGE[1..17];
    self.value2 := FAILCODE
END;

limited1 := LIMIT(ds, 2);
limited2 := LIMIT(ds, 3);
limited3 := LIMIT(ds, 4);

recovered1 := CATCH(limited1, SKIP);
recovered2 := CATCH(limited2, ONFAIL(FailTransform));
recovered3 := CATCH(CATCH(limited3, FAIL(1, 'Failed, sorry')), ONFAIL(FailTransform));

OUTPUT(recovered1); //empty recordset
OUTPUT(recovered2); //
OUTPUT(recovered3); //
```

See Also: RECOVERY, FAILCODE, SOAPCALL
The **FETCH** function processes through all records in the `index` in the order specified by the `index`, fetching each related record from the `basedataset` and performing the `transform` function.

The `index` will typically have a filter expression to specify the exact set of records to return from the `basedataset`. If the filter expression defines a single record in the `basedataset`, FETCH will return just that one record. See KEYED/WILD for a discussion of INDEX filtering.

### FETCH TRANSFORM Function Requirements

The `transform` function must take up to two parameters: a LEFT record that must be of the same format as the `basedataset`, and an optional RIGHT record that that must be of the same format as the `index`. The optional second parameter is useful in those instances where the index contains information not present in the recordset.

Example:

```ecl
PtblRec := RECORD
  STRING2  State := Person.per_st;
  STRING20 City  := Person.per_full_city;
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>basedataset</code></td>
<td>The base DATASET attribute to process. Filtering is not allowed.</td>
</tr>
<tr>
<td><code>index</code></td>
<td>The INDEX attribute that provides keyed access into the <code>basedataset</code>. This will typically have a filter expression.</td>
</tr>
<tr>
<td><code>position</code></td>
<td>An expression that provides the means of locating the correct record in the <code>basedataset</code> (usually the field within the index containing the fileposition value).</td>
</tr>
<tr>
<td><code>transform</code></td>
<td>The TRANSFORM function to call for each record fetched from the <code>basedataset</code>. If omitted, FETCH returns a set containing all fields from both the <code>basedataset</code> and index, with the second of any duplicate named fields removed.</td>
</tr>
<tr>
<td><code>LOCAL</code></td>
<td>Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.</td>
</tr>
<tr>
<td><code>UNORDERED</code></td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td><code>ORDERED</code></td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><code>STABLE</code></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td><code>UNSTABLE</code></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><code>PARALLEL</code></td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td><code>numthreads</code></td>
<td>Optional. Try to evaluate this activity using <code>numthreads</code> threads.</td>
</tr>
<tr>
<td><code>ALGORITHM</code></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: FETCH returns a record set.
STRING25 Lname := Person.per_last_name;
STRING15 Fname := Person.per_first_name;
END;

PtblOut := OUTPUT(TABLE(Person,PtblRec),'RTTEMP::TestFetch');
Ptbl := DATASET('RTTEMP::TestFetch',
    {PtblRec,UNSIGNED8 __fpos {VIRTUAL(fileposition)}},
    FLAT);

Bld := BUILD(Ptbl,
    {state,city,lname,fname,__fpos},
    'RTTEMPkey::TestFetch');

AlphaInStateCity := INDEX(Ptbl,
    {state,city,lname,fname,__fpos},
    'RTTEMPkey::TestFetch');

TYPEOF(Ptbl) copy(Ptbl l) := TRANSFORM
    SELF := l;
END;

AlphaPeople := FETCH(Ptbl,
    AlphaInStateCity(state='FL',
        city = 'BOCA RATON',
        Lname = 'WIK',
        Fname = 'PICHA'),
    RIGHT.__fpos,
    copy(LEFT));

OutFile := OUTPUT(CHOOSEN(AlphaPeople,10));
SEQUENTIAL(PtblOut,Bld,OutFile)

// NOTE the use of a filter on the index file. This is an important
// use of standard filtering technique in conjunction with indexing
// to achieve optimal "random" access into the base record set

See Also: TRANSFORM Structure, RECORD Structure, BUILDINDEX, INDEX, KEYED/WILD
FROMJSON

FROMJSON( record, jsonstring,[ONFAIL(transform)])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>record</td>
<td>The RECORD structure to produce. Each field should specify the XPATH to the data in the jsonstring that it should hold. If omitted, the lower-cased field names are used.</td>
</tr>
<tr>
<td>jsonstring</td>
<td>A string containing the JSON to convert.</td>
</tr>
<tr>
<td>ONFAIL</td>
<td>Optional. Specifies a transform to handle errors in the JSON.</td>
</tr>
<tr>
<td>transform</td>
<td>A TRANSFORM structure matching the record structure of the first parameter.</td>
</tr>
</tbody>
</table>

Return: FROMJSON returns a single row (record).

The **FROMJSON** function returns a single row (record) in the *record* format from the specified *jsonstring*. This may be used anywhere a single row can be used (similar to the ROW function).

Example:

```ecl
namesRec := RECORD
    UNSIGNED2 EmployeeID{xpath('EmpID')};
    STRING10 Firstname{xpath('FName')};
    STRING10 Lastname{xpath('LName')};
END;

x := '{"FName": "George" , "LName": "Jetson", "EmpID": 42}';
rec := FROMJSON(namesRec,x);
OUTPUT(rec);
```

Example with Error handling and bad JSON:

```ecl
namesRec := RECORD
    UNSIGNED2 EmployeeID{xpath('EmpID')};
    STRING20 Firstname{xpath('FName')};
    STRING20 Lastname{xpath('LName')};
END;

x := '{"FName": "malformedJSON"George" , "LName": "Jetson", "EmpID": 42}';

namesRec createFailure() :=

    TRANSFORM
        SELF.FirstName := FAILMESSAGE;
        SELF := [];
    END;

rec := FROMJSON(namesRec,x,ONFAIL(createFailure()));
OUTPUT(rec);
```

See Also: ROW, TOJSON
FROMUNICODE

FROMUNICODE( string, encoding )

| string  | The UNICODE string to translate. |
| encoding | The encoding codepage (supported by IBM's ICU) to use for the translation. |

Return: FROMUNICODE returns a single DATA value.

The FROMUNICODE function returns the string translated from the specified encoding to a DATA value.

Example:

```
DATA5 x := FROMUNICODE(u'ABCDE','UTF-8'); // results in 4142434445
OUTPUT(x);
```

See Also: TOUNICODE, UNICODEORDER
FROMXML

FROMXML( record, xmlstring ,[ONFAIL(transform )])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>record</td>
<td>The RECORD structure to produce. Each field must specify the XPATH to the data in the xmlstring that it should hold.</td>
</tr>
<tr>
<td>xmlstring</td>
<td>A string containing the XML to convert.</td>
</tr>
<tr>
<td>ONFAIL</td>
<td>Optional. Specifies a transform to handle errors in the XML.</td>
</tr>
<tr>
<td>transform</td>
<td>A TRANSFORM structure matching the record structure of the first parameter.</td>
</tr>
</tbody>
</table>

Return: FROMXML returns a single row (record).

The FROMXML function returns a single row (record) in the record format from the specified xmlstring. This may be used anywhere a single row can be used (similar to the ROW function).

Example:

```ecl
namesRec := RECORD
    UNSIGNED2 EmployeeID{xpath('EmpID')};
    STRING10  Firstname{xpath('FName')};
    STRING10  Lastname{xpath('LName')};
END;
x := '<Row><FName>George</FName><LName>Jetson</LName><EmpID>42</EmpID></Row>'';
rec := FROMXML(namesRec,x);
OUTPUT(rec);
```

Example with Error handling and bad XML:

```ecl
namesRec := RECORD
    UNSIGNED2 EmployeeID{xpath('EmpID')};
    STRING20  Firstname{xpath('FName')};
    STRING20  Lastname{xpath('LName')};
END;
x := '<Row><FName>George</FName><LName><unmatchedtag>Jetson</LName><EmpID>42</EmpID></Row>'';

namesRec createFailure() :=
    TRANSFORM
    SELF.FirstName := FAILMESSAGE;
    SELF := [];
END;
rec := FROMXML(namesRec,x,ONFAIL(createFailure()));
OUTPUT(rec);
```

See Also: ROW, TOXML
GETENV

GETENV( name [, default ] )

<table>
<thead>
<tr>
<th>name</th>
<th>A string constant containing the name of the environment variable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>Optional. A string constant containing the default value to use if the environment variable does not exist.</td>
</tr>
</tbody>
</table>

Return: GETENV returns a STRING value.

The GETENV function returns the value of the named environment variable. If the environment variable does not exist or contains no value, the default value is returned.

Example:

g1 := GETENV('namesTable');
g2 := GETENV('myPort','25');
OUTPUT(GETENV('HPCC_DEPLOYMENT', 'unknown'));
OUTPUT(g1);
OUTPUT(g2);

See Also: Environment Variables
GLOBAL

GLOBAL( expression [, FEW | MANY ] )

expression The expression to evaluate at a global scope.

FEW Optional. Indicates that the expression will result in fewer than 10,000 records. This allows optimization to produce a significantly faster result.

MANY Optional. Indicates that the expression will result in many records.

Return: GLOBAL may return scalar values or record sets.

The GLOBAL function evaluates the expression at a global scope, similar to what the GLOBAL workflow service does but without the need to define a separate attribute.

Example:

```ecl
IMPORT doxie;
besr := doxie.best_records;
ssnr := doxie.ssn_records;

//**** Individual record defs
recbesr := RECORDOF(besr);
recssnr := RECORDOF(ssnr);

//**** Monster record def
rec := RECORD, MAXLENGTH(doxie.maxlength_report)
    DATASET(recbesr) best_information_children;
    DATASET(recssnr) ssn_children;
END;
nada := DATASET([0], {INTEGER1 a});
rec tra(nada l) := TRANSFORM
    SELF.best_information_children := GLOBAL(besr);
    SELF.ssn_children := GLOBAL(ssnr);
END;
EXPORT central_records := PROJECT(nada, tra(left));
```

See Also: GLOBAL Workflow Service
**GRAPH**

**GRAPH( recordset, iterations, processor[, UNORDERED | ORDERED( bool )][, STABLE | UNSTABLE][, PARALLEL [ ( numthreads ) ][, ALGORITHM( name ) ]])**

- **recordset**: The initial set of records to process.
- **iterations**: The number of times to call the processor function.
- **processor**: The function attribute to process the input. This function may use the following as arguments:
  - **ROWSET(LEFT)**: Specifies the set of input datasets, which may be indexed to specify the result set from any specific iteration -- ROWSET(LEFT)[0] indicates the initial input recordset while ROWSET(LEFT)[1] indicates the result set from the first iteration. This may also be used as the first parameter to the RANGE function to specify a set of datasets (allowing the graph to efficiently process N-ary merge/join arguments).
  - **COUNTER**: Specifies an INTEGER parameter for the graph iteration number.
  - **UNORDERED**: Optional. Specifies the output record order is not significant.
  - **ORDERED**: Specifies the significance of the output record order.
  - **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.
  - **STABLE**: Optional. Specifies the input record order is significant.
  - **UNSTABLE**: Optional. Specifies the input record order is not significant.
  - **PARALLEL**: Optional. Try to evaluate this activity in parallel.
  - **numthreads**: Optional. Try to evaluate this activity using numthreads threads.
  - **ALGORITHM**: Optional. Override the algorithm used for this activity.
  - **name**: The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

**Return:** GRAPH returns the record set result of the last of the iterations.

The **GRAPH** function is similar to the **LOOP** function, but it executes as though all the **iterations** of the **processor** call were expanded out, removing any branches that can't be executed, and then joined together. The resulting graph is as efficient as if the graph had been expanded out by hand.

This function is named **GRAPH** because it creates an execution graph at runtime (typically used in ROXIE queries) from a collection of compile-time components. The ROWSET(LEFT) keyword indicates that each iteration adds its result to the SET OF DATASETS passed to the next iteration. That means the result of each iteration must be a dataset in the same format as the initial input recordset parameter. Typical use of the GRAPH() function would be in queries for which each subsequent iteration requires access to any/all previous iterations' results.

**Example:**

```ecl
//This simple example demonstrates text searching
//for a variable number of search terms.
//For simplicity's sake this example uses sets,
//whereas in a production query you would most likely
```
// use inverted indexes into your huge datasets.

IMPORT Std;

// layout of the dataset containing text to search:
Rec := {UNSIGNED1 RecID, STRING line};

// The text search function:
SearchText(STRING SearchFor, DATASET(Rec) SearchIn) := FUNCTION
  // uppercase and clean the data:
  Ltrs := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  Upper(STRING s) := Std.Str.toUpperCase(s);
  LosePunc(STRING s) := STD.Str.SubstituteExcluded(Upper(s),Ltrs,' ');

  // create a set of words:
  SetWords(STRING s) := Std.Str.SplitWords(s,' ');

  // the set of words to search for:
  SearchWords := SetWords(LosePunc(SearchFor));

  // split the text lines to search into sets of words:
  LineWords :=
    PROJECT(SearchIn,
      TRANSFORM({UNSIGNED1 RecID,SET OF STRING wordset},
        SELF.wordset := SetWords(LosePunc(LEFT.line)),
        SELF := LEFT));

  // the graph processor function:
  GraphWork(SET OF DATASET(RECORDOF(LineWords)) ds, UNSIGNED4 C) := FUNCTION
    // find all matches for the current iteration's word:
    NewMatches := LineWords(SearchWords[C] IN wordset);
    // return only those lines that were also in the previous iteration's result:
    RETURN JOIN(ds[C-1],NewMatches,LEFT.RecID = RIGHT.RecID);
  END;

  GraphRes := GRAPH(LineWords,
    COUNT(SearchWords),
    GraphWork(ROWSET(LEFT),COUNTER));

  // Return original records where all search terms were found:
  RETURN SearchIn(RecID IN SET(GraphRes,RecID));
END;

ds1 := DATASET([{{1,'It is a truth universally acknowledged, that a single man' +
    ' in possession of a good fortune, must be in want of a wife.'},
    {2,'"My dear Mr. Bennet," said his lady to him one day,' +
      '"have you heard that Netherfield Park is let at last?"'},
    {3,'Mr. Bennet replied that he had not.'},
    {4,'"But it is," returned she; "for Mrs. Long has just been here,' +
      ' and she told me all about it."'},
    {5,'Mr. Bennet made no answer.'},
    {6,'"Do you not want to know who has taken it?" cried his' +
      ' wife impatiently.'},
    {7,'"You want to tell me, and I have no objection to hearing it."'}],Rec);

SearchText('Mr. Bennet',ds1);    // returns 2, 3, 5
SearchText('Mr. Bennet said',ds1);  // returns 2

See Also: LOOP, RANGE
The `GROUP` function fragments a `recordset` into a set of sets. This allows aggregations and other operations (such as `ITERATE`, `DEDUP`, `ROLLUP`, `SORT` and others) to occur within defined subsets of the data—the operation executes on each subset, individually. This means that the boundary condition code written in the `TRANSFORM` function for those functions that use them will be different than it would be for a `recordset` that has simply been SORTed.

The `recordset` must be sorted by the same elements as the `breakcriteria` if the ALL option is not specified.

The `recordset` gets 'ungrouped' by use in a `TABLE` function, by the `JOIN` function in some circumstances (see `JOIN`), by `UNGROUP`, or by another `GROUP` function with the second parameter omitted.

Example:

```ecl
personRecord := RECORD
    STRING UID;
    STRING first_name;
    STRING last_name;
    STRING address;
    STRING city;
    STRING state;
```

---

**GROUP**

`GROUP( recordset [, breakcriteria [, ALL ]] [, LOCAL ] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] [, ALGORITHM( name ) ] )

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>recordset</code></td>
<td>The set of records to fragment.</td>
</tr>
<tr>
<td><code>breakcriteria</code></td>
<td>Optional. A comma-delimited list of expressions or key fields in the recordset that specifies how to fragment the recordset. You may use the keyword RECORD to indicate all fields in the recordset, and/or you may use the keyword EXCEPT to list non-group fields in the structure. You may also use the keyword ROW to indicate each record in the recordset is a separate group. If omitted, the recordset is ungrouped from any previous grouping.</td>
</tr>
<tr>
<td><code>ALL</code></td>
<td>Optional. Indicates the <code>breakcriteria</code> is applied without regard to any previous order. If omitted, <code>GROUP</code> assumes the recordset is already sorted in <code>breakcriteria</code> order.</td>
</tr>
<tr>
<td><code>LOCAL</code></td>
<td>Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous <code>DISTRIBUTE</code>.</td>
</tr>
<tr>
<td><code>UNORDERED</code></td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td><code>ORDERED</code></td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><code>STABLE</code></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td><code>UNSTABLE</code></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><code>PARALLEL</code></td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td><code>numthreads</code></td>
<td>Optional. Try to evaluate this activity using <code>numthreads</code> threads.</td>
</tr>
<tr>
<td><code>ALGORITHM</code></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the <code>SORT</code> function's <code>STABLE</code> and <code>UNSTABLE</code> options.</td>
</tr>
</tbody>
</table>

Return: `GROUP` returns a record set.
STRING zip;
END;
person := DATASET([{'923','James','Jones','123 Elm Street','Hollywood','FL','33022'},
{'924','Sally','Jones','22 Main Street','Tampa','FL','33604'},
{'925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101'},
{'926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108'},
{'927','Evelyn','Murray','740 SW 10th Street','Boston','MA','02116'},
{'928','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131'}], personRecord);

SortedSet := SORT(Person,last_name); //sort by last name
GroupedSet := GROUP(SortedSet,last_name); //then group them
OUTPUT(GroupedSet);
SecondSort := SORT(GroupedSet,first_name);
  //sorts by first name within each last name group
  // this is a "sort within group"
OUTPUT(SecondSort);
UnGroupedSet := GROUP(GroupedSet); //ungroup the dataset
OUTPUT(UnGroupedSet);

See Also: REGROUP, COMBINE, UNGROUP, EXCEPT
HASH

**HASH(expressionlist)**

<table>
<thead>
<tr>
<th>expressionlist</th>
<th>A comma-delimited list of values.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return</td>
<td>HASH returns a single value.</td>
</tr>
</tbody>
</table>

The **HASH** function returns a 32-bit hash value derived from all the values in the `expressionlist`. Trailing spaces are trimmed from string (or UNICODE) fields before the value is calculated (casting to DATA prevents this).

Example:

```ecl
personRecord := RECORD
  STRING UID;
  STRING first_name;
  STRING last_name;
  STRING address;
  STRING city;
  STRING state;
  STRING zip;
  STRING SSN
END;

person := DATASET([{'923','James','Jones','123 Elm Street','Hollywood','FL','33022','000-423-6567'},
  {'924','Sally','Jones','22 Main Street','Tampa','FL','33604','000-123-4567'},
  {'925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101','000-123-3383'},
  {'926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108','000-123-4464'},
  {'927','Evelyn','Murray','740 SW 10th Street','Boston ','MA','02116','000-123-5556'},
  {'928','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131','000-123-7890'}], personRecord);

OUTPUT(Person,{SSN,HASH(SSN)});
  //output SSN and its hash value
```

See Also: **DISTRIBUTE**, **HASH32**, **HASH64**, **HASHCRC**, **HASHMD5**
HASH32

HASH32(expressionlist)

<table>
<thead>
<tr>
<th>expressionlist</th>
<th>A comma-delimited list of values.</th>
</tr>
</thead>
</table>

Return: HASH32 returns a single value.

The **HASH32** function returns a 32-bit FNV (Fowler/Noll/Vo) hash value derived from all the values in the *expressionlist*. This uses a hashing algorithm that is faster and less likely than HASH to return the same values from different data. Trailing spaces are trimmed from string (or UNICODE) fields before the value is calculated (casting to DATA prevents this).

Example:

```ecl
personRecord := RECORD
    STRING UID;
    STRING first_name;
    STRING last_name;
    STRING address;
    STRING city;
    STRING state;
    STRING zip;
    STRING SSN
END;

person := DATASET([
    {'923','James','Jones','123 Elm Street','Hollywood','FL','33022','000-423-6567'},
    {'924','Sally','Jones','22 Main Street','Tampa','FL','33604','000-123-4567'},
    {'925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101','000-123-3383'},
    {'926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108','000-123-4464'},
    {'927','Evelyn','Murray','740 SW 10th Street','Boston ','MA','02116','000-123-5556'},
    {'928','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131','000-123-7890'}
], personRecord);

OUTPUT(Person, {SSN, HASH32(SSN)});
```

See Also: **DISTRIBUTE, HASH, HASH64, HASHCRC, HASHMD5**
HASH64

**HASH64**(expressionlist)

<table>
<thead>
<tr>
<th>expressionlist</th>
<th>A comma-delimited list of values.</th>
</tr>
</thead>
</table>

**Return:** HASH64 returns a single value.

The **HASH64** function returns a 64-bit FNV (Fowler/Noll/Vo) hash value derived from all the values in the **expressionlist**. Trailing spaces are trimmed from string (or UNICODE) fields before the value is calculated (casting to DATA prevents this).

**Example:**

```ecl
personRecord := RECORD
STRING UID;
STRING first_name;
STRING last_name;
STRING address;
STRING city;
STRING state;
STRING zip;
STRING SSN
END;
person := DATASET([{'923','James','Jones','123 Elm Street','Hollywood','FL','33022','000-423-6567'},
{'924','Sally','Jones','22 Main Street','Tampa','FL','33604','000-123-4567'},
{'925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101','000-123-3383'},
{'926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108','000-123-4464'},
{'927','Evelyn','Murray','740 SW 10th Street','Boston ','MA','02116','000-123-5556'},
{'928','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131','000-123-7890'}], personRecord);
OUTPUT(Person,{SSN,HASH64(SSN)});
//output SSN and its 64-bit hash value
```

See Also: DISTRIBUTE, HASH, HASH32, HASHCRC, HASHMD5
**HASHCRC**

**HASHCRC(expressionlist)**

<table>
<thead>
<tr>
<th>expressionlist</th>
<th>A comma-delimited list of values.</th>
</tr>
</thead>
</table>

**Return:** HASHCRC returns a single value.

The **HASHCRC** function returns a CRC (cyclical redundancy check) value derived from all the values in the `expressionlist`.

**Example:**

```ecl
personRecord := RECORD
  STRING UID;
  STRING first_name;
  STRING last_name;
  STRING address;
  STRING city;
  STRING state;
  STRING zip;
  STRING SSN
END;
person := DATASET(
  [{'923','James','Jones','123 Elm Street','Hollywood','FL','33022','000-423-6567'},
   {'924','Sally','Jones','22 Main Street','Tampa','FL','33604','000-123-4567'},
   {'925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101','000-123-3383'},
   {'926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108','000-123-4464'},
   {'927','Evelyn','Murray','740 SW 10th Street','Boston ','MA','02116','000-123-5556'},
   {'928','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131','000-123-7890'}],
  personRecord);
OUTPUT(Person,{SSN,HASHCRC(SSN)});
  //output SSN and its CRC hash value
```

See Also: DISTRIBUTE, HASH, HASH32, HASH64, HASHMD5
HASHMD5

**HASHMD5(expressionlist)**

<table>
<thead>
<tr>
<th><strong>expressionlist</strong></th>
<th>A comma-delimited list of values.</th>
</tr>
</thead>
</table>

Return: HASHMD5 returns a single DATA16 value.

The **HASHMD5** function returns a 128-bit hash value derived from all the values in the **expressionlist**, based on the MD5 algorithm developed by Professor Ronald L. Rivest of MIT. Unlike other hashing functions, trailing spaces are NOT trimmed before the value is calculated.

Example:

```ecl
personRecord := RECORD
    STRING UID;
    STRING first_name;
    STRING last_name;
    STRING address;
    STRING city;
    STRING state;
    STRING zip;
    STRING SSN
END;
person := DATASET([{'923','James','Jones','123 Elm Street','Hollywood','FL','33022','000-423-6567'},
    {'924','Sally','Jones','22 Main Street','Tampa','FL','33604','000-123-4567'},
    {'925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101','000-123-3383'},
    {'926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108','000-123-4464'},
    {'927','Evelyn','Murray','740 SW 10th Street','Boston ','MA','02116','000-123-5556'},
    {'928','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131','000-123-7890'}], personRecord);
OUTPUT(Person,{SSN,HASHMD5(SSN)});
//output SSN and its MD5 hash value
```

See Also: **DISTRIBUTE**, **HASH**, **HASH32**, **HASH64**, **HASHCRC**
HAVING

HAVING( groupdataset, expression [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] [, ALGORITHM( name ) ] )

| groupdataset | The name of a GROUPed record set. |
| expression   | The logical expression by which to filter the groups. |
| UNORDERED    | Optional. Specifies the output record order is not significant. |
| ORDERED      | Specifies the significance of the output record order. |
| bool         | When False, specifies the output record order is not significant. When True, specifies the default output record order. |
| STABLE       | Optional. Specifies the input record order is significant. |
| UNSTABLE     | Optional. Specifies the input record order is not significant. |
| PARALLEL     | Optional. Try to evaluate this activity in parallel. |
| numthreads   | Optional. Try to evaluate this activity using numthreads threads. |
| ALGORITHM    | Optional. Override the algorithm used for this activity. |
| name         | The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options. |

Return: HAVING returns a GROUPed record set.

The HAVING function returns a GROUPed record set containing just those groups for which the expression is true. This is similar to the HAVING clause in SQL.

Example:

```
personRecord := RECORD
  STRING UID;
  STRING first_name;
  STRING last_name;
  STRING address;
  STRING city;
  STRING state;
  STRING zip;
  STRING SSN
END;

person := DATASET([{'923','James','Jones','123 Elm Street','Hollywood','FL','33022','000-423-6567'},
  {'924','Sally','Jones','22 Main Street','Tampa','FL','33604','000-123-4567'},
  {'925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101','000-123-3383'},
  {'926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108','000-123-4464'},
  {'927','Evelyn','Murray','740 SW 10th Street','Boston','MA','02116','000-123-5556'},
  {'928','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131','000-123-7890'}], personRecord);

MyGroups := GROUP(SORT(Person,state),state);
  //group by state
Filtered := HAVING(MyGroups,COUNT(ROWS(LEFT)) > 1);
  //filter out the small groups
OUTPUT(Filtered);
```

See Also: GROUP
HTTPCALL

result := HTTPCALL( url, httpmethod, responsemimetype, outstructure [, options ] );

<table>
<thead>
<tr>
<th>result</th>
<th>The definition name for the resulting recordset.</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>A string containing the URL that hosts the service to invoke. This may contain parameters to the service.</td>
</tr>
<tr>
<td>httpmethod</td>
<td>A string containing the HTTP Method to invoke. Valid methods are: &quot;GET&quot;</td>
</tr>
<tr>
<td>responsemimetype</td>
<td>A string containing the Response MIME type to use. Valid types are: &quot;text/xml&quot;</td>
</tr>
<tr>
<td>outstructure</td>
<td>A RECORD structure containing the output field definitions. For an XML-based responsemimetype these should use XPATH to specify the exact data path.</td>
</tr>
<tr>
<td>options</td>
<td>A comma-delimited list of optional specifications from the list below.</td>
</tr>
</tbody>
</table>

HTTPCALL is a function that calls a REST service.

Valid options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RETRY(count)</td>
<td>Specifies re-attempting the call count number of times if non-fatal errors occur. If omitted, the default is three (3).</td>
</tr>
<tr>
<td>TIMEOUT(period)</td>
<td>Specifies the amount of time to attempt the read before failing. The period is a real number where the integer portion specifies seconds. Setting to zero (0) indicates waiting forever. If omitted, the default is three hundred (300).</td>
</tr>
<tr>
<td>TIMELIMIT(period)</td>
<td>Specifies the total amount of time allowed for the HTTPCALL. The period is a real number where the integer portion specifies seconds. If omitted, the default is zero (0) indicating no limit.</td>
</tr>
<tr>
<td>XPATH(xpath)</td>
<td>Specifies the path used to access rows in the output. If omitted, the default is: ‘serviceResponse/Results/Result/Dataset/Row’.</td>
</tr>
<tr>
<td>ONFAIL(transform)</td>
<td>Specifies either the transform function to call if the service fails for a particular record, or the keyword SKIP. The TRANSFORM function must produce a resultype the same as the outstructure and may use FAILCODE and/or FAILMESSAGE to provide details of the failure.</td>
</tr>
<tr>
<td>TRIM</td>
<td>Specifies all trailing spaces are removed from strings before output.</td>
</tr>
<tr>
<td>HTTPHEADER</td>
<td>Specifies header information to be passed to the service. HTTPCALL supports multiple instances of the HTTPHEADER option if you need to specify multiple key/value header strings.</td>
</tr>
</tbody>
</table>

Example:

```
worldBankSource := RECORD
   STRING name {XPATH('name')}
END;

OutRec1 := RECORD
   DATASET(worldBankSource) Fred{XPATH('/source')};
END;

OUTPUT(raw);
```
OutRec1, HTTPHEADER('Authorization','Basic dXNlcm5hbWU6cGFzc3dvcmQ='),
HTTPHEADER('MyLiteral','FOO'));
OUTPUT(raw2);
**IF**

### Syntax

```
IF(expression, trueresult [, falseresult ])
```

<table>
<thead>
<tr>
<th><strong>expression</strong></th>
<th>A conditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>trueresult</strong></td>
<td>The result to return when the expression is true. This may be any expression or action.</td>
</tr>
<tr>
<td><strong>falseresult</strong></td>
<td>The result to return when the expression is false. This may be any expression or action. This may be omitted only if the result is an action.</td>
</tr>
</tbody>
</table>

**Return:** IF returns a single value, set, recordset, or action.

The **IF** function evaluates the expression (which must be a conditional expression with a Boolean result) and returns either the trueresult or falseresult based on the evaluation of the expression. Both the trueresult and falseresult must be the same type (i.e. both strings, or both recordsets, or ...). If the trueresult and falseresult are strings, then the size of the returned string will be the size of the resultant value. If subsequent code relies on the size of the two being the same, then a type cast to the required size may be required (typically to cast an empty string to the proper size so subsequent string indexing will not fail).

**Example:**

```ecl
MyDate := IF(ValidDate(Trades.trd_dopn),Trades.trd_dopn,0);
// in this example, 0 is the false value and
// Trades.trd_dopn is the True value returned

MyTrades := IF(person.per_sex = 'Male',
    Trades(trd_bal<100),
    Trades(trd_bal>1000));
// return low balance trades for men and high balance
// trades for women

MyAddress := IF(person.gender = 'M',
    cleanAddress182(person.address),
    (STRING182)'');
//cleanAddress182 returns a 182-byte string
// so casting the empty string false result to a
// STRING182 ensures a proper-length string return
```

See Also: IFF, MAP, EVALUATE, CASE, CHOOSE, SET
**IFF**

**IFF**(expression, trueresult [, falseresult ])

<table>
<thead>
<tr>
<th>expression</th>
<th>A conditional expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>trueresult</td>
<td>The result to return when the expression is true. This may be any expression or action.</td>
</tr>
<tr>
<td>falseresult</td>
<td>The result to return when the expression is false. This may be any expression or action. This may be omitted only if the result is an action.</td>
</tr>
</tbody>
</table>

**Return:** IF returns a single value, set, recordset, or action.

The **IFF** function performs the same functionality as IF, but ensures that an expression containing complex boolean logic is evaluated exactly as it appears.

The main difference between IF and IFF is with conditional scalar expressions. Generally the arguments to dataset IF() operators are always evaluated lazily, but there are situations, if the expression type is a row or a scalar, where they might not be.

For example:

```
foo := IF (age < 18, '', expensiveFunction());
```

Then there is no guarantee that expensiveFunction will not be executed. If you use IFF() the code generator internally converts it to a dataset expression, ensuring it is evaluated lazily.

See Also: IF, MAP, EVALUATE, CASE, CHOOSE, SET
**IMPORT**

\[
\text{resulttype funcname \ (parameterlist) := IMPORT(\ text{language}, \text{function} \ [::\text{TIME} \ [(\text{label})]]);}
\]

<table>
<thead>
<tr>
<th>resulttype</th>
<th>The ECL return value type of the \text{function}.</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcname</td>
<td>The ECL definition name of the \text{function}.</td>
</tr>
<tr>
<td>parameterlist</td>
<td>A comma separated list of the parameters to pass to the \text{function}.</td>
</tr>
<tr>
<td>language</td>
<td>Specifies the name of the external programming language whose code you wish to embed in your ECL. A language support module for that language must have been installed in your plugins directory. Modules are provided for languages such as Java, R, Javascript, and Python. You can write your own pluggable language support module for any language not already supported by using the supplied ones as examples or starting points.</td>
</tr>
<tr>
<td>function</td>
<td>A string constant containing the name of the function to include.</td>
</tr>
<tr>
<td>\text{TIME}</td>
<td>Tracks timing of an external function call or embedded source code and reports them back as metrics to the user.</td>
</tr>
<tr>
<td>label</td>
<td>Optional. A string constant containing the name to associate with the timer. If omitted, the default is used.</td>
</tr>
</tbody>
</table>

The **IMPORT** declaration allows you to call existing code written in the external \text{language}. This may be used to call Java or Python code, but is not usable with Javascript or R code (use the EMBED structure instead). Java code must be placed in a .java file and compiled using the javac compiler in the usual way. All Java classes used must be thread safe.

**WARNING:** This feature could create memory corruption and/or security issues, so great care and forethought are advised--consult with Technical Support before using.

Example:

```
IMPORT JAVA;
INTEGER jadd(INTEGER a, INTEGER b) := IMPORT(java, 'JavaCat.add:(II)I': TIME('MyTime'));
jadd (22,23);
```

See Also: IMPORT, EMBED Structure
INTFORMAT

**INTFORMAT**(expression, width, mode)

<table>
<thead>
<tr>
<th>expression</th>
<th>The expression that specifies the integer value to format.</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>The size of string in which to right-justify the value.</td>
</tr>
<tr>
<td>mode</td>
<td>The format type: 0 = leading blank fill, 1 = leading zero fill.</td>
</tr>
<tr>
<td>Return:</td>
<td>INTFORMAT returns a single value.</td>
</tr>
</tbody>
</table>

The **INTFORMAT** function returns the value of the `expression` formatted as a right-justified string of `width` characters.

Example:

```plaintext
val := 123456789;
OUTPUT(INTFORMAT(val,20,1)); //formats as '00000000000123456789'
OUTPUT(INTFORMAT(val,20,0)); //formats as '          123456789'
```

See Also: REALFORMAT
**ISVALID**

**ISVALID( field )**

<table>
<thead>
<tr>
<th>field</th>
<th>The name of a DECIMAL, REAL, or alien data TYPE field.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>ISVALID returns a single Boolean value.</td>
</tr>
</tbody>
</table>

The **ISVALID** function validates that the `field` contains a legal value. If the contents are not valid for the declared value type of the `field` (such as hexadecimal values greater than 9 in a DECIMAL), ISVALID returns FALSE, otherwise it returns TRUE.

Example:

```ecl
MyVal := IF(ISVALID(Infile.DecimalField),Infile.DecimalField,0);
//ISVALID returns TRUE if the value is legal
```

See Also: TYPE Structure, DECIMAL, REAL
# ITERATE

**ITERATE**(recordset, transform [, LOCAL] [, UNORDERED | ORDERED( bool)] [, STABLE | UNSTABLE] [, PARALLEL [ (numthreads) ]] [, ALGORITHM(name)])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>The set of records to process.</td>
</tr>
<tr>
<td>transform</td>
<td>The TRANSFORM function to call for each record in the recordset.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

**LOCAL** Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.

Return: ITERATE returns a record set.

The **ITERATE** function processes through all records in the recordset one pair of records at a time, performing the transform function on each pair in turn. The first record in the recordset is passed to the transform as the first right record, paired with a left record whose fields are all blank or zero. Each resulting record from the transform becomes the left record for the next pair.

## TRANSFORM Function Requirements - ITERATE

The transform function must take at least two parameters: LEFT and RIGHT records that must both be of the same format as the resulting recordset. An optional third parameter may be specified: an integer COUNTER specifying the number of times the transform has been called for the recordset or the current group in the recordset (see the GROUP function).

Example:

```plaintext
ResType := RECORD
  INTEGER1 Val;
  INTEGER1 Rtot;
END;

Records := DATASET([[1,0],[2,0],[3,0],[4,0]],ResType);

/* these are the recs going in: 
Val RTotal
-------
1 0
2 0
```
ResType T(ResType L, ResType R) := TRANSFORM
   SELF.Rtot := L.Rtot + R.Val;
   SELF := R;
END;

MySet1 := ITERATE(Records,T(LEFT,RIGHT));
OUTPUT(MySet1);

/* these are the recs coming out:
Val RTotal
----------
1   1
2   3
3   6
4   10 */

See Also: TRANSFORM Structure, RECORD Structure, ROLLUP
JOIN

JOIN(leftrecset, rightrecset, joincondition [, transform] [, jointype] [, joinflags] )

JOIN(setofdatasets, joincondition, transform, SORTED( fields) [, jointype] )

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>leftrecset</td>
<td>The left set of records to process.</td>
</tr>
<tr>
<td>rightrecset</td>
<td>The right set of records to process. This may be an INDEX.</td>
</tr>
<tr>
<td>joincondition</td>
<td>An expression specifying how to match records in the leftrecset and rightrecset or setofdatasets (see Matching Logic discussions below). In the expression, the keyword LEFT is the dataset qualifier for fields in the leftrecset and the keyword RIGHT is the dataset qualifier for fields in the rightrecset.</td>
</tr>
<tr>
<td>transform</td>
<td>Optional. The TRANSFORM function to call for each pair of records to process. If omitted, JOIN returns all fields from both the leftrecset and rightrecset, with the second of any duplicate named fields removed.</td>
</tr>
<tr>
<td>jointype</td>
<td>Optional. An inner join if omitted, else one of the listed types in the JOIN Types section below.</td>
</tr>
<tr>
<td>joinflags</td>
<td>Optional. Any option (see the JOIN Options section below) to specify exactly how the JOIN operation executes.</td>
</tr>
<tr>
<td>setofdatasets</td>
<td>The SET of recordsets to process ([idx1,idx2,idx3]), typically INDEXes, which all must have the same format.</td>
</tr>
<tr>
<td>SORTED</td>
<td>Specifies the sort order of records in the input setofdatasets and also the output sort order of the result set.</td>
</tr>
<tr>
<td>fields</td>
<td>A comma-delimited list of fields in the setofdatasets, which must be a subset of the input sort order. These fields must all be used in the joincondition as they define the order in which the fields are STEPPED.</td>
</tr>
</tbody>
</table>

Return: JOIN returns a record set.

The JOIN function produces a result set based on the intersection of two or more datasets or indexes (as determined by the joincondition).

JOIN Two Datasets

JOIN(leftrecset, rightrecset, joincondition [, transform] [, jointype] [, joinflags] )

The first form of JOIN processes through all pairs of records in the leftrecset and rightrecset and evaluates the condition to find matching records. If the condition and jointype specify the pair of records qualifies to be processed, the transform function executes, generating the result.

JOIN dynamically sorts/distributes the leftrecset and rightrecset as needed to perform its operation based on the condition specified, therefore the output record set is not guaranteed to be in the same order as the input record sets. If JOIN does do a dynamic sort of its input record sets, that new sort order cannot be relied upon to exist past the execution of the JOIN. This principle also applies to any GROUPing--the records are automatically “un-grouped” as needed except under the following circumstances:

* For LOOKUP and ALL joins, the GROUPing and sort order of the leftrecset are preserved.

* For KEYED joins the GROUPing (but not the sort order) of the leftrecset is preserved.

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
Matching Logic - JOIN

The record matching *joincondition* is processed internally as two parts:

| "equality" match | All the simple "LEFT.field = RIGHT.field" logic that defines matching records. For JOINs that use keys, all these must be fields in the key to qualify for inclusion in this part. If there is no "equality" part to the *joincondition* logic, then you get a "JOIN too complex" error. |
| "non-equality" match | All other matching criteria in the *joincondition* logic, such as "LEFT.field > RIGHT.field" expressions or any OR logic that may be involved with the final determination of which *leftrecset* and *rightrecset* records actually match. |

This internal logic split allows the JOIN code to be optimized for maximum efficiency--first the "equality" logic is evaluated to provide an interim result that is then evaluated against any "non-equality" in the matching *joincondition*.

Options

The following *joinflags* options may be specified to determine exactly how the JOIN executes.

```
[ PARTITION LEFT | PARTITION RIGHT | [MANY] LOOKUP [ FEW ] | GROUPED | ALL | NOSORT [ ( which ) ] | KEYED [ (index) ] | [ UNORDERED ] | LOCAL | HASH ]
```

**PARTITION LEFT | RIGHT**

Specifies which recordset provides the partition points that determine how the records are sorted and distributed amongst the supercomputer nodes. PARTITION RIGHT specifies the *rightrecset* while PARTITION LEFT specifies the *leftrecset*. If omitted, PARTITION LEFT is the default.

**[MANY] LOOKUP**

Specifies the *rightrecset* is a relatively small file of lookup records. If the LOCAL option is not present a hash table of the entire *rightrecset* is fully copied to every node, but if the LOCAL option is present the hash table is created locally on each node. If MANY is not present, the *rightrecset* records bear a Many to 0/1 relationship with the records in the *leftrecset* (for each record in the *leftrecset* there is at most 1 record in the *rightrecset*, enforced by an implicit dedup if there are duplicate records in the lookup table based on the key fields used in the *joincondition*). If MANY is present, the *rightrecset* records bear a Many to 0/Many relationship with the records in the *leftrecset*. This option allows the optimizer to avoid unnecessary sorting of the *leftrecset*. Valid only for inner, LEFT OUTER, or LEFT ONLY *jointypes*. The ATMOST, LIMIT, and KEEP options are supported in conjunction with MANY LOOKUP.

**SMART**

Specifies to use an in-memory lookup when possible, but use a distributed join if the right dataset is large.

**FEW**

Specifies the LOOKUP *rightrecset* has few records, so little memory is used, allowing multiple lookup joins to be included in the same Thor subgraph.

**GROUPED**

Specifies the same action as MANY LOOKUP but preserves grouping. Primarily used in the rapid Data Delivery Engine. Valid only for inner, LEFT OUTER, or LEFT ONLY *jointypes*. The ATMOST, LIMIT, and KEEP options are supported in conjunction with GROUPED.

**ALL**

Specifies the *rightrecset* is a small file that can be fully copied to every node, which allows the compiler to ignore the lack of any "equality" portion to the condition, eliminating the "join too complex" error that the condition would normally produce. If an
"equality" portion is present, the JOIN is internally executed as a MANY LOOKUP. The KEEP option is supported in conjunction with this option.

| NOSORT | Performs the JOIN without dynamically sorting the tables. This implies that the leftrecset and/or rightrecset must have been previously sorted and partitioned based on the fields specified in the joincondition so that records can be easily matched. |
| which | Optional. The keywords LEFT or RIGHT to indicate the leftrecset or rightrecset has been previously sorted. If omitted, NOSORT assumes both the leftrecset and rightrecset have been previously sorted. |
| KEYED | Specifies using indexed access into the rightrecset (see INDEX). |
| index | Optional. The name of an INDEX into the rightrecset for a full-keyed JOIN (see below). If omitted, indicates the rightrecset will always be an INDEX (useful when the rightrecset is passed in as a parameter to a function). |
| UNORDERED | Optional. Specifies the KEYED JOIN operation does not preserve the sort order of the leftrecset. |
| LOCAL | Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE. |
| HASH | Specifies an implicit DISTRIBUTE of the leftrecset and rightrecset across the supercomputer nodes based on the joincondition so each node can do its job with local data. |
| KEEP(n) | Specifies the maximum number of matching records (n) to generate into the result set from each leftrecset record. If omitted, all matches are kept. This is useful where there may be many matching pairs and you need to limit the number in the result set. KEEP is not supported for RIGHT OUTER, RIGHT ONLY, LEFT ONLY, or FULL ONLY jointypes. |
| ATMOST | Specifies a maximum number of matching records which, if exceeded, eliminates all those matches from the result set. This is useful for situations where you need to eliminate all "too many matches" record pairs from the result set. ATMOST is not supported on RIGHT ONLY or RIGHT OUTER jointypes. There are two forms: ATMOST(condition, n) -- maximum is computed only for the condition. ATMOST(n) -- maximum is computed for the entire joincondition, unless KEYED is used in the joincondition, in which case only the KEYED expressions are used. When ATMOST is specified (and the JOIN is not full or half-keyed), the joincondition and condition may include string field comparisons that use string indexing with an asterisk as the upper bound, as in this example: J1 := JOIN(dsL,dsR, LEFT.name[1..*]=RIGHT.name[3..*] AND LEFT.val < RIGHT.val, T(LEFT,RIGHT), ATMOST(LEFT.name[1..*]=RIGHT.name[3..*],3)); The asterisk indicates matching as many characters as necessary to reduce the number of candidate matches to below the ATMOST number (n). |
| condition | A portion of the joincondition expression. |
| n | Specifies the maximum number of matches allowed. |
| LIMIT | Specifies a maximum number of matching records which, if exceeded, either fails the job, or eliminates all those matches from the result set. This is useful for situations where you need to eliminate all "too many matches" record pairs from the result set. Typically used for KEYED and "half-keyed" joins (see below), LIMIT differs from ATMOST primarily by its affect on a LEFT OUTER join, in which a leftrecset record with too many matching records would be treated as a non-match by ATMOST (the leftrecset record would be in the output with no matching rightrecset records), whereas LIMIT would either fail the job entirely, or SKIP the record (eliminating the leftrecset record entirely from the output). The LIMIT is applied to the set of records...
that meet the the hard match ("equality") portion of the joincondition but before the soft match ("non-equality") portion of the joincondition is evaluated. If omitted, the default is LIMIT(10000). The implicit limit is only added if there is no explicit LIMIT, no ATMOST, and no KEEP, or has a non-keyed filter and isn't a left only JOIN.

<table>
<thead>
<tr>
<th>value</th>
<th>The maximum number of matches allowed; LIMIT(0) is unlimited.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIP</td>
<td>Optional. Specifies eliminating all the matching records whose total number exceeds the maximum value of the LIMIT result instead of failing the job.</td>
</tr>
<tr>
<td>transform</td>
<td>Optional. Specifies outputting a single record produced by the transform instead of failing the workunit (similar to the ONFAIL option of the LIMIT function).</td>
</tr>
<tr>
<td>FAIL</td>
<td>Optional. Specifies using the FAIL action to configure the error message when the job fails.</td>
</tr>
<tr>
<td>SKEW</td>
<td>Indicates that you know the data for this join will not be spread evenly across nodes (will be skewed after both files have been distributed based on the join condition) and you choose to override the default by specifying your own limit value to allow the job to continue despite the skewing. Only valid on non-keyed joins (the KEYED option is not present and the rightrecset is not an INDEX).</td>
</tr>
<tr>
<td>limit</td>
<td>A value between zero (0) and one (1.0 = 100%) indicating the maximum percentage of skew to allow before the job fails (the default is 0.1 = 10%).</td>
</tr>
<tr>
<td>target</td>
<td>Optional. A value between zero (0) and one (1.0 = 100%) indicating the desired maximum percentage of skew to allow (the default is 0.1 = 10%).</td>
</tr>
<tr>
<td>THRESHOLD</td>
<td>Indicates the minimum size for a single part of either the leftrecset or rightrecset before the SKEW limit is enforced. Only valid on non-keyed joins (the KEYED option is not present and the rightrecset is not an INDEX).</td>
</tr>
<tr>
<td>size</td>
<td>An integer value indicating the minimum number of bytes for a single part.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

The following options are mutually exclusive and may only be used to the exclusion of the others in this list: PARTITION LEFT | PARTITION RIGHT | [MANY] LOOKUP | GROUPED | ALL | NOSORT | HASH

In addition to this list, the KEYED and LOCAL options are also mutually exclusive with the options listed above, but not to each other. When both KEYED and LOCAL options are specified, only the INDEX part(s) on each node are accessed by that node.

Typically, the leftrecset should be larger than the rightrecset to prevent skewing problems (because PARTITION LEFT is the default behavior). If the LOOKUP or ALL options are specified, the rightrecset must be small enough to be loaded into memory on every node, and the operation is then implicitly LOCAL. The ALL option is impractical if the rightrecset is larger than a few thousand records (due to the number of
comparisons required). The size of the rightrecset is irrelevant in the case of "half-keyed" and "full-keyed" JOINs (see the Keyed Join discussion below).

Use SMART when the right side dataset is likely to be small enough to fit in memory, but is not guaranteed to fit.

If you get an error similar to this:

"error: 1301: Pool memory exhausted:..."

this means the rightrecset is too large and a LOOKUP JOIN should not be used. A SMART JOIN may be a good option in this case.

Example:

```ecl
//this example demos the implicit DEDUP of the lookup dataset
r1 := {INTEGER1 fred};
r2 := {INTEGER1 fred,INTEGER1 sue};
ds1 := DATASET([1,2,3,4],r1);
ds2 := DATASET([{1,2},{2,2},
                {1,1},{2,1},
                {3,1},{4,1}],r2);

r2 XF(ds1 L, ds2 R) := TRANSFORM
  SELF.fred := L.fred;
  SELF.sue := R.sue;
END;
JOIN(ds1,ds2,
  LEFT.Fred = RIGHT.fred,
  XF(LEFT,RIGHT),LOOKUP);
```

**Keyed Joins**

A "full-keyed" JOIN uses the KEYED option and the joincondition must be based on key fields in the index. The join is actually done between the leftrecset and the index into the rightrecset—the index needs the dataset's record pointer (VIRTUAL(fileposition)) field to properly fetch records from the rightrecset. The typical KEYED join passes only the rightrecset to the TRANSFORM.

If the rightrecset is an INDEX, the operation is a "half-keyed" JOIN. Usually, the INDEX in a "half-keyed" JOIN contains "payload" fields, which frequently eliminates the need to read the base dataset. If this is the case, the "payload" INDEX does not need to have the dataset's record pointer (VIRTUAL(fileposition)) field declared. For a "half-keyed" JOIN the joincondition may use the KEYED and WILD keywords that are available for use in INDEX filters, only.

For both types of keyed join, any GROUPing and/or distribution of the base record sets is left untouched. See KEYED and WILD for a discussion of INDEX filtering.

**Join Logic**

The JOIN operation follows this logic:

1. **Record distribution/sorting to get match candidates on the same nodes.**

The PARTITION LEFT, PARTITION RIGHT, LOOKUP, ALL, NOSORT, KEYED, HASH, and LOCAL options indicate how this happens. These options are mutually exclusive; only one may be specified, and PARTITION LEFT is the default. SKEW and THRESHOLD may modify the requested behaviour. LOOKUP also has the additional effect of deduping the rightrecset by the joincondition.

2. **Record matching.**

The joincondition, LIMIT, and ATMOST determine how this is done.
An implicit limit of 10000 is added when there is no LIMIT specified AND the following is true:

There is no ATMOST limit specified AND it is not a LEFT ONLY JOIN AND (there is either no KEEP limit specified OR the JOIN has a postfilter).

3. Determine what matches to pass to transform.

The jointype determines this.

4. Generate output records through the TRANSFORM function.

The implicit or explicit transform parameter determines this.

5. Filter output records with SKIP.

If the transform for a record pair results in a SKIP, then the output record is not counted towards any KEEP option totals.

6. Limit output records with KEEP.

Any output records for a given leftrecset record over and above the permitted KEEP value are discarded. In a FULL OUTER join, rightrecset records that match no record are treated as if they all matched different default leftrecset records (that is, the KEEP counter is reset for each one).

TRANSFORM Function Requirements - JOIN

The transform function must take at least one or two parameters: a LEFT record formatted like the leftrecset, and/or a RIGHT record formatted like the rightrecset (which may be of different formats). The format of the resulting record set need not be the same as either of the inputs.

Join Types: Two Datasets

The following jointypes produce the following types of results, based on the records matching produced by the joincondition:

<table>
<thead>
<tr>
<th>jointype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inner (default)</td>
<td>Only those records that exist in both the leftrecset and rightrecset.</td>
</tr>
<tr>
<td>LEFT OUTER</td>
<td>At least one record for every record in the leftrecset.</td>
</tr>
<tr>
<td>RIGHT OUTER</td>
<td>At least one record for every record in the rightrecset.</td>
</tr>
<tr>
<td>FULL OUTER</td>
<td>At least one record for every record in the leftrecset and rightrecset.</td>
</tr>
<tr>
<td>LEFT ONLY</td>
<td>One record for each leftrecset record with no match in the rightrecset.</td>
</tr>
<tr>
<td>RIGHT ONLY</td>
<td>One record for each rightrecset record with no match in the leftrecset.</td>
</tr>
<tr>
<td>FULL ONLY</td>
<td>One record for each leftrecset and rightrecset record with no match in the opposite record set.</td>
</tr>
</tbody>
</table>

Examples:

```ecl
outrec := RECORD
    people.id;
    people.firstname;
    people.lastname;
END;

RT_folk := JOIN(people(firstname[1] = 'R'),
    people(lastname[1] = 'T'),
    LEFT.id=RIGHT.id,
    TRANSFORM(outrec,SELF := LEFT));
OUTPUT(RT_folk);
```
/*********************** Half KEYED JOIN example:

peopleRecord := RECORD
  INTEGER8 id;
  STRING20 addr;
END;

peopleDataset := DATASET([{3000,'LONDON'},{3500,'SMITH'},
  {30,'TAYLOR'}], peopleRecord);

PtblRec doHalfJoin(peopleRecord l) := TRANSFORM
  SELF := l;
END;

FilledRecs3 := JOIN(peopleDataset, SequenceKey,
  LEFT.id=RIGHT.sequence,doHalfJoin(LEFT));

FilledRecs4 := JOIN(peopleDataset, AlphaKey,
  LEFT.addr=RIGHT.Lname,doHalfJoin(LEFT));

******************* Full KEYED JOIN example:

PtblRec := RECORD
  INTEGER8 seq;
  STRING2  State;
  STRING20 City;
  STRING25 Lname;
  STRING15 Fname;
END;

PtblRec Xform(person L, INTEGER C) := TRANSFORM
  SELF.seq      := C;
  SELF.State    := L.per_st;
  SELF.City     := L.per_full_city;
  SELF.Lname    := L.per_last_name;
  SELF.Fname    := L.per_first_name;
END;

Proj := PROJECT(Person(per_last_name[1]=per_first_name[1]),
  Xform(LEFT,COUNTER));

PtblOut := OUTPUT(Proj,'~RTTEMP::TestKeyedJoin',OVERWRITE);

Ptbl := DATASET('~RTTEMP::TestKeyedJoin',
  [PtblRec,UNSIGNED8 __fpos {VIRTUAL(fileposition)}],
  FLAT);

AlphaKey := INDEX(Ptbl,{lname,fname,__fpos},
  '~RTTEMP::TestKeyedJoin.lname.fname');

SeqKey := INDEX(Ptbl,{seq,__fpos}, '~RTTEMP::TestKeyedJoin.sequence');

Bld1 := BUILD(AlphaKey,OVERWRITE);
Bld2 := BUILD(SeqKey,OVERWRITE);

peopleRecord := RECORD
  INTEGER8 id;
  STRING20 addr;
END;

peopleDataset := DATASET([{3000,'LONDON'},{3500,'SMITH'},
  {30,'TAYLOR'}], peopleRecord);

joinedRecord := RECORD
  PtblRec;
  peopleRecord;
END;

joinedRecord doJoin(peopleRecord l, Ptbl r) := TRANSFORM
  SELF := l;
  SELF := r;
END;

FilledRecs1 := JOIN(peopleDataset, Ptbl,LEFT.id=RIGHT.seq,
  doJoin(LEFT,RIGHT), KEYED(SeqKey));

FilledRecs2 := JOIN(peopleDataset, Ptbl,LEFT.addr=RIGHT.Lname,
  doJoin(LEFT,RIGHT), KEYED(AlphaKey));

SEQUENTIAL(PtblOut,Bld1,Bld2,OUTPUT(FilledRecs1),OUTPUT(FilledRecs2))
JOIN Set of Datasets

```
```

The second form of JOIN is similar to the MERGEJOIN function in that it takes a SET OF DATASETS as its first parameter. This allows the possibility of joining more than two datasets in a single operation.

Record Matching Logic

The record matching _joincondition_ may contain two parts: a STEPPED condition that may optionally be ANDed with non-STEPPED conditions. The STEPPED expression contains leading equality expressions of the _fields_ from the SORTED option (trailing components may be range comparisons if the range values are independent of the LEFT and RIGHT rows), ANDed together, using LEFT and RIGHT as dataset qualifiers. If not present, the STEPPED condition is deduced from the _fields_ specified by the SORTED option.

The order of the datasets within the _setofdatasets_ can be significant to the way the _joincondition_ is evaluated. The _joincondition_ is duplicated between adjacent pairs of datasets, which means that this _joincondition_:

```
LEFT.field = RIGHT.field
```

when applied against a _setofdatasets_ containing three datasets, is logically equivalent to:

```
ds1.field = ds2.field AND ds2.field = ds3.field
```

TRANSFORM Function Requirements - JOIN _setofdatasets_

The _transform_ function must take at least one parameter which must take either of two forms:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFT</td>
<td>formatted like any of the <em>setofdatasets</em>. This indicates the first dataset in the <em>setofdatasets</em>.</td>
</tr>
<tr>
<td>ROWS(LEFT)</td>
<td>formatted like any of the <em>setofdatasets</em>. This indicates a record set made up of all records from any dataset in the <em>setofdatasets</em> that match the <em>joincondition</em>—this may not include all the datasets in the <em>setofdatasets</em>, depending on which <em>jointype</em> is specified.</td>
</tr>
</tbody>
</table>

The format of the resulting output record set must be the same as the input datasets.

Join Types: _setofdatasets_

The following _jointypes_ produce the following types of results, based on the records matching produced by the _joincondition_:

<table>
<thead>
<tr>
<th>Jointype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INNER</td>
<td>This is the default if no <em>jointype</em> is specified. Only those records that exist in all datasets in the <em>setofdatasets</em>.</td>
</tr>
<tr>
<td>LEFT OUTER</td>
<td>At least one record for every record in the first dataset in the <em>setofdatasets</em>.</td>
</tr>
<tr>
<td>LEFT ONLY</td>
<td>One record for every record in the first dataset in the <em>setofdatasets</em> for which there is no match in any of the subsequent datasets.</td>
</tr>
<tr>
<td>MOFN(min [,max))</td>
<td>One record for every record with matching records in min number of adjacent datasets within the <em>setofdatasets</em>. If max is specified, the record is not included if max number of dataset matches are exceeded.</td>
</tr>
</tbody>
</table>
Example:

```ecl
Rec := RECORD, MAXLENGTH(4096)
    STRING1 Letter;
    UNSIGNED1 DS;
    UNSIGNED1 Matches := 0;
    UNSIGNED1 LastMatch := 0;
    SET OF UNSIGNED1 MatchDSs := [];
END;

ds1 := DATASET([{'A',1},{'B',1},{'C',1},{'D',1},{'E',1}],Rec);
ds2 := DATASET([{'A',2},{'B',2},{'H',2},{'I',2},{'J',2}],Rec);
ds3 := DATASET([{'B',3},{'C',3},{'M',3},{'N',3},{'O',3}],Rec);
ds4 := DATASET([{'A',4},{'B',4},{'R',4},{'S',4},{'T',4}],Rec);
ds5 := DATASET([{'B',5},{'V',5},{'W',5},{'X',5},{'Y',5}],Rec);
SetDS := [ds1,ds2,ds3,ds4,ds5];

Rec XF(Rec L, DATASET(Rec) Matches) := TRANSFORM
    SELF.Matches := COUNT(Matches);
    SELF.LastMatch := MAX(Matches,DS);
    SELF.MatchDSs := SET(Matches,DS);
    SELF := L;
END;
j1 := JOIN(SetDS,
    STEPPED(LEFT.Letter=RIGHT.Letter),
    XF(LEFT,ROWS(LEFT)),SORTED(Letter));
j2 := JOIN(SetDS,
    STEPPED(LEFT.Letter=RIGHT.Letter),
    XF(LEFT,ROWS(LEFT)),LEFT OUTER);
j3 := JOIN(SetDS,
    STEPPED(LEFT.Letter=RIGHT.Letter),
    XF(LEFT,ROWS(LEFT)),LEFT ONLY);
j4 := JOIN(SetDS,
    STEPPED(LEFT.Letter=RIGHT.Letter),
    XF(LEFT,ROWS(LEFT)),MOFN(3));
j5 := JOIN(SetDS,
    STEPPED(LEFT.Letter=RIGHT.Letter),
    XF(LEFT,ROWS(LEFT)),MOFN(3,4));
OUTPUT(j1);
OUTPUT(j2);
OUTPUT(j3);
OUTPUT(j4);
OUTPUT(j5);
```

See Also: TRANSFORM Structure, RECORD Structure, SKIP, ROWDIFF, STEPPED, KEYED/WILD, MERGEJOIN
KEYDIFF

```ecl
[ attrname := ] KEYDIFF( index1, index2, file [, OVERWRITE ] [, EXPIRE( [ days ] ) ] [, UNORDERED | ORDERED( bool ) ][, STABLE | UNSTABLE ][, PARALLEL [ ( numthreads ) ][, ALGORITHM( name ) ]]);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attrname</td>
<td>Optional. The action name, which turns the action into an attribute definition, therefore not executed until the attrname is used as an action.</td>
</tr>
<tr>
<td>index1</td>
<td>An INDEX attribute.</td>
</tr>
<tr>
<td>index2</td>
<td>An INDEX attribute whose structure is identical to index1.</td>
</tr>
<tr>
<td>file</td>
<td>A string constant specifying the logical name of the file to write the differences to.</td>
</tr>
<tr>
<td>OVERWRITE</td>
<td>Optional. Specifies overwriting the filename if it already exists.</td>
</tr>
<tr>
<td>EXPIRE</td>
<td>Optional. Specifies the file is a temporary file that may be automatically deleted after the specified number of days.</td>
</tr>
<tr>
<td>days</td>
<td>Optional. The number of days after which the file may be automatically deleted. If omitted, the default is seven (7).</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

The KEYDIFF action compares `index1` to `index2` and writes the differences to the specified `file`. If `index1` to `index2` are not exactly the same structure, an error occurs. Once generated, the `file` may be used by the KEYPATCH action.

Example:

```ecl
Vehicles := DATASET('vehicles',
{STRING2 st,
 STRING20 city,
 STRING20 lname,
 UNSIGNED8 filepos{VIRTUAL{fileposition}}},
FLAT);

i1 := INDEX(Vehicles,
{st,city,lname,filepos},
'vkey::20041201::st.city.lname');
i2 := INDEX(Vehicles,
{st,city,lname,filepos},
'vkey::20050101::st.city.lname');
KEYDIFF(i1,i2,'KEY::DIFF::20050101::i1i2',OVERWRITE);
```

See Also: KEYPATCH, INDEX
**KEYPATCH**

```ecl
```

- **attrname**: Optional. The action name, which turns the action into an attribute definition, therefore not executed until the attrname is used as an action.
- **index**: The INDEX attribute to apply the changes to.
- **patchfile**: A string constant specifying the logical name of the file containing the changes to implement (created by KEYDIFF).
- **newfile**: A string constant specifying the logical name of the file to write the new index to.
- **OVERWRITE**: Optional. Specifies overwriting the `newfile` if it already exists.
- **EXPIRE**: Optional. Specifies the `newfile` is a temporary file that may be automatically deleted after the specified number of days.
- **days**: Optional. The number of days after which the file may be automatically deleted. If omitted, the default is seven (7).
- **UNORDERED**: Optional. Specifies the output record order is not significant.
- **ORDERED**: Specifies the significance of the output record order.
- **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.
- **STABLE**: Optional. Specifies the input record order is significant.
- **UNSTABLE**: Optional. Specifies the input record order is not significant.
- **PARALLEL**: Optional. Try to evaluate this activity in parallel.
- **numthreads**: Optional. Try to evaluate this activity using `numthreads` threads.
- **ALGORITHM**: Optional. Override the algorithm used for this activity.
- **name**: The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function’s STABLE and UNSTABLE options.

The **KEYPATCH** action uses the `index` and `patchfile` to write a new index to the specified `newfile` containing all the original index data updated by the information from the `patchfile`.

**Example:**

```ecl
Vehicles := DATASET('vehicles',
  {STRING2 st,
   STRING20 city,
   STRING20 lname,
   UNSIGNED8 filepos{VIRTUAL(fileposition)}},
  FLAT);
11 := INDEX(Vehicles,
  {st,city,lname,filepos},
  'vkey::20041201::st.city.lname');
12 := INDEX(Vehicles,
  {st,city,lname,filepos},
  'vkey::20050101::st.city.lname');
a := KEYDIFF(11,12,'KEY::DIFF::20050101::11i2',OVERWRITE);
b := KEYPATCH(11,
  'KEY::DIFF::20050101::11i2',
  'vkey::st.city.lname'OVERWRITE);
```
SEQUENTIAL(a, b);

See Also: KEYDIFF, INDEX
KEYUNICODE

KEYUNICODE(string)

<table>
<thead>
<tr>
<th>string</th>
<th>A UNICODE string.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>KEYUNICODE returns a single DATA value.</td>
</tr>
</tbody>
</table>

The **KEYUNICODE** function returns a DATA value derived from the *string* parameter, such that a comparison of these data values is equivalent to a locale sensitive comparison of the Unicode values that generated them—and, being a simple memcmp(), is significantly faster. The generating *string* values must be of the same locale or the results are unpredictable. This function is particularly useful if you're doing a lot of compares on a UNICODE field in a large dataset—it can be a good idea to generate a key field and do the compares on that instead.

Example:

```ecl
//where you might do this:
my_record := RECORD
  UNICODE_en_US str;
END;
my_dataset := DATASET('filename', my_record, FLAT);
my_sorted := SORT(my_dataset, str);
//you could instead do this:
my_record := RECORD
  UNICODE_en_US str;
  DATA strkey := KEYUNICODE(SELF.str);
END;
my_dataset := DATASET('filename', my_record, FLAT);
my_sorted := SORT(my_dataset, strkey);
```

See Also: UNICODE, LOCALE
LENGTH

**LENGTH(expression)**

<table>
<thead>
<tr>
<th>expression</th>
<th>A string expression.</th>
</tr>
</thead>
</table>

Return: LENGTH returns a single integer value.

The **LENGTH** function returns the length of the string resulting from the **expression** by treating the **expression** as a temporary STRING.

Example:

```
INTEGER MyLength := LENGTH('XYZ' + 'ABC');
OUTPUT(MyLength);  //MyLength is 6
```

See Also: String Operators, STRING
LIBRARY

LIBRARY( INTERNAL( module ), interface [ ( parameters) ] )

LIBRARY( module , interface [ ( parameters) ] )

| INTERNAL | Optional. Specifies the module is an attribute, not an external library (created by the BUILD action). |
| module   | The name of the query library. When INTERNAL, this is the name of the MODULE attribute that implements the query library. If not INTERNAL, this is a string expression containing the name of the workunit that compiled the query library (typically defined with #WORKUNIT). |
| interface| The name of the INTERFACE structure that defines the query library. |
| parameters| Optional. The values to pass to the INTERFACE, if defined to receive parameters. |

Return: LIBRARY results in a MODULE that can be used to reference the exported attributes from the specified module.

The LIBRARY function defines an instance of a query library--the interface as implemented by the module when passed the specified parameters. Query libraries are only used by hthor and Roxie.

INTERNAL libraries are typically used when developing queries, while external libraries are best for production queries. An INTERNAL library generates the library code as a separate unit, but then includes that unit within the query workunit. It doesn't have the advantage of reducing compile time or memory usage in Roxie that an external library would have, but it does retain the library structure, and means that changes to the code cannot affect anyone else using the system.

External libraries are created by the BUILD action and use the "name" form of #WORKUNIT to specify the external name of the library. An external library is pre-compiled and therefore reduces compile time for queries that use it. They also reduce memory usage in Roxie.

Example:

```ecl
NamesRec := RECORD
    INTEGER1 NameID;
    STRING20 FName;
    STRING20 LName;
END;
NamesTable := DATASET([ {1,'Doc','Holliday'},
                        {2,'Liz','Taylor'},
                        {3,'Mr','Nobody'},
                        {4,'Anywhere','but here'}],
    NamesRec);
FilterLibIface1(DATASET(namesRec) ds, STRING search) := INTERFACE
    EXPORT DATASET(namesRec) matches;
    EXPORT DATASET(namesRec) others;
END;
FilterDsLib1(DATASET(namesRec) ds, STRING search) :=
    MODULE, LIBRARY(FilterLibIface1)
    EXPORT matches := ds(Lname = search);
    EXPORT others := ds(Lname != search);
END;

// Run this to create the 'Ppass.FilterDsLib' external library
// #WORKUNIT('name','Ppass.FilterDsLib')
// BUILD(FilterDsLib1);
lib1 := LIBRARY(INTERNAL(FilterDsLib1),
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license. 260
FilterLibIface1(NamesTable, 'Holliday'));

lib2 := LIBRARY('Ppass.FilterDsLib',
    FilterLibIface1(NamesTable, 'Holliday'));

IFilterArgs := INTERFACE
    EXPORT DATASET(namesRec) ds;
    EXPORT STRING search;
END;

FilterLibIface2(IFilterArgs args) := INTERFACE
    EXPORT DATASET(namesRec) matches;
    EXPORT DATASET(namesRec) others;
END;

FilterDsLib2(IFilterArgs args) := MODULE,LIBRARY(FilterLibIface2)
    EXPORT matches := args.ds(Lname = args.search);
    EXPORT others := args.ds(Lname != args.search);
END;

// Run this to create the 'Ipass.FilterDsLib' external library
// #WORKUNIT('name','Ipass.FilterDsLib')
// BUILD(FilterDsLib2);
SearchArgs := MODULE(IFilterArgs)
    EXPORT DATASET(namesRec) ds := NamesTable;
    EXPORT STRING search := 'Holliday';
END;

lib3 := LIBRARY(INTERNAL(FilterDsLib2),
    FilterLibIface2(SearchArgs));
lib4 := LIBRARY('Ipass.FilterDsLib',
    FilterLibIface2(SearchArgs));

OUTPUT(lib1.matches,NAMED('INTERNAL_matches_straight_parms'));
OUTPUT(lib1.others, NAMED('INTERNAL_nonmatches_straight_parms'));
OUTPUT(lib2.matches,NAMED('EXTERNAL_matches_straight_parms'));
OUTPUT(lib2.others, NAMED('EXTERNAL_nonmatches_straight_parms'));
OUTPUT(lib3.matches,NAMED('INTERNAL_matches_interface_parms'));
OUTPUT(lib3.others, NAMED('INTERNAL_nonmatches_interface_parms'));
OUTPUT(lib4.matches,NAMED('EXTERNAL_matches_interface_parms'));
OUTPUT(lib4.others, NAMED('EXTERNAL_nonmatches_interface_parms'));

See Also: MODULE, INTERFACE, BUILD
The LIMIT function causes the attribute to fail with an exception if the recset contains more records than maxrecs on any single node of the supercomputer (unless the SKIP option is used for an index read or the ONFAIL option is present). If the failclause is present, it specifies the exception number and message. This is typically used to control "runaway" queries in the Rapid Data Delivery Engine supercomputer.

Example:

```ecl
RecStruct := RECORD
  INTEGER1 Number;
  STRING1 Letter;
END;
SomeFile := DATASET(
  [{1,'A'}, {1,'B'}, {1,'C'}, {1,'D'}, {1,'E'},
   {1,'F'}, {1,'G'}, {1,'H'}, {1,'I'}, {1,'J'},
   {2,'K'}, {2,'L'}, {2,'M'}, {2,'N'}, {2,'O'},
   {2,'P'}, {2,'Q'}, {2,'R'}, {2,'S'}, {2,'T'},
   {2,'U'}, {2,'V'}, {2,'W'}, {2,'X'}, {2,'Y'}],
  RecStruct);
//throw an exception
```
X := LIMIT(SomeFile, 10, FAIL(99, 'error!'));
//single record output
Y := LIMIT(SomeFile, 10,
    ONFAIL(TRANSFORM(RecStruct,
        SELF := ROW({0, ''}, RecStruct))));
//no exception, just no record
Z := LIMIT(SomeFile, 10, SKIP);
// OUTPUT(X); //This one will throw an exception
OUTPUT(Y);
OUTPUT(Z);

See Also: FAIL, TRANSFORM
**LN**

**LN(n)**

<table>
<thead>
<tr>
<th>n</th>
<th>The real number to evaluate.</th>
</tr>
</thead>
</table>

**Return:** LN returns a single real value.

The LN function returns the natural logarithm of the parameter. This is the opposite of the EXP function.

Example:

```
MyLogPI := LN(3.14159); //1.144729041185178
OUTPUT(MyLogPI);
```

See Also: EXP, SQRT, POWER, LOG
**LOADXML**

```
[attributename := ] LOADXML( xmlstring | symbol [, branch ])
```

<table>
<thead>
<tr>
<th><strong>attributename</strong></th>
<th>Optional. The action name, which turns the action into an attribute definition, therefore not executed until the <strong>attributename</strong> is used as an action.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>xmlstring</strong></td>
<td>A string expression containing the XML text to process inline (no carriage returns or line feeds).</td>
</tr>
<tr>
<td><strong>symbol</strong></td>
<td>The template symbol containing the XML text to process (typically loaded by #EXPORT or #EXPORTXML).</td>
</tr>
<tr>
<td><strong>branch</strong></td>
<td>A user-defined string naming the XML text, allowing #FOR to operate.</td>
</tr>
</tbody>
</table>

**LOADXML** opens an active XML scope for Template language statements or symbols to act on. **LOADXML** must be the first line of code to function correctly.

**LOADXML** is also used in "drilldown" MACRO code.

**Example:**

```ecl
LOADXML('&lt;section&gt;&lt;item type"count">&lt;set&gt;person&lt;/set&gt;&lt;/item&gt;&lt;/section&gt;');
//this macro receives in-line XML as its parameter
//and demonstrates the code for multiple row drilldown
EXPORT id(xmlRow) :=   MACRO
    STRING myxmlText := xmlRow;
    LOADXML(myxmlText);
    #DECLARE(OutStr)
    #SET(OutStr, '' )
    #FOR(row)
        #APPEND(OutStr,
            'OUTPUT(FETCH(Files.People,Files.PeopleIDX(id=
              + %'id'% + '+'),RIGHT.RecPos));\n' )
        #APPEND(OutStr,
            'ds' + %'id'%
            + ' := FETCH(Files.Property,Files.PropertyIDX(personid= ' + %'id'% + '+'),RIGHT.RecPos);\n' )
        #APPEND(OutStr,
            'OUTPUT(ds' + %'id'%
            + ' := FETCH(Files.Vehicle,Files.VehicleIDX(personid= ' + %'id'% + '+'),RIGHT.RecPos));\n' )
    #END
    %OutStr%
ENDMACRO;
//this is an example of code for a drilldown (1 per row)
EXPORT CountTaxdata(xmlRow) := MACRO
    LOADXML(xmlRow);
    OUTPUT(FETCH(Files.TaxData,
        Files.TaxdataIDX(propertyid=%propertyid%),
        RIGHT.RecPos));
ENDMACRO;
//This example uses #EXPORT to generate the XML
```

NamesRecord := RECORD
    STRING10 first;
    STRING20 last;
```
END;

r := RECORD
    UNSIGNED4 dg_parentid;
    STRING10 dg_firstname;
    STRING  dg_lastname;
    UNSIGNED1 dg_prange;
    IFBLOCK(SELF.dg_prange % 2 = 0)
        STRING20 extrafield;
    END;
    NamesRecord namerec;
    DATASET(NamesRecord) childNames;
END;

ds := DATASET('~RTTEST::OUT::ds', r, thor);

//Walk a record and do some processing on it.
#DECLARE(out)
#EXPORT(out, r);
LOADXML(%'out'%, 'FileStruct');

#FOR (FileStruct)
    #FOR (Field)
        #IF (%'{@isEnd}'% <> '')
            OUTPUT('END');
        #ELSE
            OUTPUT(%'{@type}'%)
            #IF (%'{@size}'% <> '-15' AND
                %'{@isRecord}'%=' ' AND
                %'{@isDataset}'%=' ')
                + %'{@size}'%
            #ENDIF
            + ' ' + %'{@label}'% + ';');
        #END
    #END
#END
OUTPUT('Done');

See Also: Templates, #EXPORT, #EXPORTXML
LOCAL

**LOCAL**(data)

<table>
<thead>
<tr>
<th>data</th>
<th>The name of a DATASET or INDEX attribute.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return</td>
<td>LOCAL returns a record set or index.</td>
</tr>
</tbody>
</table>

The `LOCAL` function specifies that all subsequent operations on the `data` are performed locally on each node (similar to use of the `LOCAL` option on a function). This is typically used within an `ALLNODES` operation. **Available for use only in Roxie.**

Example:

```ecl
ds := JOIN(SomeData,LOCAL(SomeIndex), LEFT.ID = RIGHT.ID);
```

See Also: `ALLNODES`, `THISNODE`, `NOLOCAL`
LOG

LOG(n)

<table>
<thead>
<tr>
<th>n</th>
<th>The real number to evaluate.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>LOG returns a single real value.</td>
</tr>
</tbody>
</table>

The LOG function returns the base-10 logarithm of the parameter.

Example:

```
MyLogPI := LOG(3.14159); // 0.4971495058611233
OUTPUT(MyLogPI);
```

See Also: EXP, SQRT, POWER, LN
LOOP

**LOOP**

```plaintext
LOOP( dataset [ ,loopcount ][ ,loopfilter ][ ,loopcondition ] , loopbody [, UNORDERED | ORDERED( bool ) ][ , STABLE | UNSTABLE ][ , ALGORITHM( name ) ][ , FEW ] )
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>dataset</strong></td>
<td>The record set to process.</td>
</tr>
<tr>
<td><strong>loopcount</strong></td>
<td>Optional. An integer expression specifying the number of times to iterate.</td>
</tr>
<tr>
<td><strong>loopfilter</strong></td>
<td>Optional. A Boolean expression that must be a record filter for the dataset identifying records whose processing is not yet complete. Records filtered out are complete, therefore immediately placed into the final result set. This evaluation occurs prior to each iteration of the loopbody.</td>
</tr>
<tr>
<td><strong>loopcondition</strong></td>
<td>Optional. A Boolean expression that continues loopbody iteration while TRUE. This can be any logical expression.</td>
</tr>
<tr>
<td><strong>loopbody</strong></td>
<td>The operation to iteratively perform. This might be a PROJECT, JOIN, a function, or any other such operation. ROWS(LEFT) is always used as a parameter to the loopbody operation, passing the current form of the dataset as the input parameter for each iteration.</td>
</tr>
<tr>
<td><strong>UNORDERED</strong></td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td><strong>ORDERED</strong></td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td><strong>bool</strong></td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><strong>STABLE</strong></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td><strong>UNSTABLE</strong></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><strong>ALGORITHM</strong></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
<tr>
<td><strong>FEW</strong></td>
<td>Optional. Indicates that activities will not require a large amount of memory. This may reduce the number of subgraphs generated within a LOOP which reduces overhead. Use only on Thor queries.</td>
</tr>
</tbody>
</table>

Return: LOOP returns a record set.

The **LOOP** function iteratively performs the loopbody operation. The COUNTER keyword is implicitly available for use to return the current iteration.

The loopcount, loopfilter, and loopcondition parameters are all optional, but at least one of the three must be present.

For each successive iteration, the input dataset (expressed as ROWS(LEFT) as the parameter to the loopbody) is the result set of the previous iteration after application of any loopfilter. The final result of the LOOP returns all records that completed processing, no matter which iteration that completion occurred (not just the result set from the final iteration).

Example:

```plaintext
namesRec := RECORD
  STRING20 lname;
  STRING10 fname;
  UNSIGNED2 age := 25;
  UNSIGNED2 ctr := 0;
END;
```
namesTable := DATASET(
[{
'Flintstone','Fred',35},
{'Flintstone','Wilma',43},
{'Jetson','Georgie',10},
{'Mr. T','Z-man'}],
namesRec);

BodyFunc(DATASET(namesRec) ds, UNSIGNED4 c) :=
  PROJECT(ds,
    TRANSFORM(namesRec,
      SELF.age := LEFT.age*c;
      SELF.ctr := COUNTER*c;
      SELF := LEFT));

/* Form 1 -- LOOP(ds, loopcount, loopbody)
   Processes loopcount times, basically a "for loop" construct.
   This example also demonstrates the two possible scopes of the COUNTER
   keyword within a LOOP:
   * The COUNTER in the LOOP function (passed to BodyFunc) is the number
     of iterations the LOOP has done.
   * The COUNTER in the TRANSFORM for the PROJECT in the BodyFunc counts
     the number of records processed by the current iteration of PROJECT.
   */
Form1 := LOOP(namesTable,
  2,  //iterate 2 times
  ROWS(LEFT) & BodyFunc(ROWS(LEFT),COUNTER)); //16 rows
OUTPUT(Form1,NAMED('Form1_example'));

/* Form 2 -- LOOP(ds, loopfilter, loopbody)
   Continues processing while the loopfilter expression is TRUE for any
   records in ROWS(LEFT). This is basically a "while loop" construct. The
   loopfilter expression is evaluated on the entire set of ROWS(LEFT)
   records prior to each iteration.
   */
Form2 := LOOP(namesTable,
  LEFT.age < 100, //process only recs where TRUE
  PROJECT(ROWS(LEFT),
    TRANSFORM(namesRec,
      SELF.age := LEFT.age*2;
      SELF := LEFT)));
OUTPUT(Form2,NAMED('Form2_example'));

/* Form 3 -- LOOP(ds, loopcondition, loopbody)
   Continues processing while the loopcondition expression is TRUE.
   This is basically a "while loop" construct. The loopcondition expression
   is evaluated on the entire set of ROWS(LEFT) records prior to each
   iteration.
   */
Form3 := LOOP(namesTable,
  SUM(ROWS(LEFT), age) < 1000 * COUNTER,
  PROJECT(ROWS(LEFT),
    TRANSFORM(namesRec,
      SELF.age := LEFT.age*2;
      SELF := LEFT)));
OUTPUT(Form3,NAMED('Form3_example'));

/* Form 4 -- LOOP(ds, loopcount, loopfilter, loopbody)
   Processes loopcount times, with the loopfilter expression
   defining when each record continues to process through the loopbody
   expression. This is basically a "for loop" construct with a filter
   specifying which records are processed each iteration.
   */
Form4 := LOOP(namesTable,
  10,
  LEFT.age < 100, //process only recs where TRUE
  BodyFunc(ROWS(LEFT), COUNTER));
OUTPUT(Form4,NAMED('Form4_example'));

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
/* Form 5 -- LOOP(ds, loopcount, loopcondition, loopbody)
Processes loopcount times, with the loopcondition expression defining the set of records that continue to process through the loopbody expression. This is basically a "for loop" construct with a filter specifying the record set processed for each iteration.

This example also demonstrates the two possible scopes of the COUNTER keyword within a LOOP:
* The COUNTER in the LOOP function's loopfilter expression is the number of recursive iterations the LOOP has done.
* The COUNTER in the TRANSFORM for the PROJECT counts the number of records processed by the current iteration of PROJECT.
*/
Form5 := LOOP(namesTable,
10,                        //iterate 10 times
LEFT.age * COUNTER <= 200, //process only recs where TRUE
PROJECT(ROWS(LEFT),
    TRANSFORM(namesRec,
        SELF.age := LEFT.age*2,
        SELF.ctr := COUNTER,
        SELF := LEFT)),
OUTPUT(Form5,NAMED('Form5_example')));

/* Form 6 -- LOOP(ds, loopfilter, loopcondition, loopbody)
Continues processing while the loopcondition expression is TRUE. Records where the loopfilter expression is TRUE continue processing. This is basically a "while loop" construct with individual record processing continuation logic.
*/
Form6 := LOOP(namesTable,
LEFT.age < 100,
    EXISTS(ROWS(LEFT)) and SUM(ROWS(LEFT), age) < 1000,
BodyFunc(ROWS(LEFT), COUNTER)),
OUTPUT(Form6,NAMED('Form6_example')));

/* Form 7 -- LOOP(ds, loopcount, loopfilter, loopcondition, loopbody)
Continues processing while the loopcondition expression is TRUE. Records where the loopfilter expression is TRUE continue processing. This is basically a "while loop" construct with individual record processing continuation logic.
*/
Form7 := LOOP(namesTable,
10,
    LEFT.age < 100,
    EXISTS(ROWS(LEFT)) and SUM(ROWS(LEFT), age) < 1000,
BodyFunc(ROWS(LEFT), COUNTER)),
OUTPUT(Form7,NAMED('Form7_example')));

See Also: GRAPH
MAP

MAP(expression => value, [ expression => value, ... ] [, elsevalue ])

expression  A conditional expression.

=>         The "results in" operator--valid only in MAP, CASE, and CHOOSESETS.

value      The value to return if the expression is true. This may be a single value expression, a
            set of values, a DATASET, a DICTIONARY, a record set, or an action.

elsevalue  Optional. The value to return if all expressions are false. This may be a single value
            expression, a set of values, a record set, or an action. May be omitted if all return
            values are actions (the default would then be no action), or all return values are record
            sets (the default would then be an empty record set).

Return:    MAP returns a single value.

The MAP function evaluates the list of expressions and returns the value associated with the first true
expression. If none of them match, the elsevalue is returned. MAP may be thought of as an "IF ... ELSIF ...
ELSIF ... ELSE" type of structure.

All return value and elsevalue values must be of exactly the same type or a "type mismatch" error will occur.
All expressions must reference the same level of dataset scoping, else an "invalid scope" error will occur.
Therefore, all expressions must either reference fields in the same dataset or the existence of a set of
related child records (see EXISTS).

The expressions are typically evaluated in the order in which they appear, but if all the return values are
scalar, the code optimizer may change that order.

Example:

Attr01 := MAP(EXISTS(Person(Person.EyeColor = 'Blue')) => 1,
               EXISTS(Person(Person.Haircolor = 'Brown')) => 2,
               3);
  //If there are any blue-eyed people, Attr01 gets 1
  //elsif there any brown-haired people, Attr01 gets 2
  //else, Attr01 gets 3
Valu6012 := MAP(NoTrades => 99,
                 NoValidTrades => 98,
                 NoValidDates => 96,
                 Count6012);
  //If there are no trades, Valu6012 gets 99
  //elsif there are no valid trades, Valu6012 gets 98
  //elsif there are no valid dates, Valu6012 gets 96
  //else, Valu6012 gets Count6012
MyTrades := MAP(rms.rms14 >= 93 => trades(trd_bal >= 10000),
                rms.rms14 >=  2 => trades(trd_bal >= 2000),
                rms.rms14 >=  1 => trades(trd_bal >= 1000),
                Trades);
  //this example takes the value of rms.rms14 and returns a
  //set of trades based on that value. If the value is <= 0,
  //then all trades are returned.

See Also: EVALUATE, IF, CASE, CHOOSE, CHOOSESETS, REJECTED, WHICH
MAX

MAX(recordset, value [, KEYED ] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL ( numthreads ) ], ALGORITHM( name ) )

MAX(valuelist)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set. This also may be the keyword GROUP to indicate finding the maximum value of the field in a group, when used in a RECORD structure to generate crosstab statistics.</td>
</tr>
<tr>
<td>value</td>
<td>The expression to find the maximum value of.</td>
</tr>
<tr>
<td>KEYED</td>
<td>Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.</td>
</tr>
<tr>
<td>valuelist</td>
<td>A comma-delimited list of expressions to find the maximum value of. This may also be a SET of values.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: MAX returns a single value.

The MAX function either returns the maximum value from the specified recordset or the valuelist. It is defined to return zero if the recordset is empty.

Example:

```
MaxVal2 := MAX(4,8,16,2,1);
SetVals := [4,8,16,2,1];
MaxVal3 := MAX(SetVals);

OUTPUT(MaxVal2); //returns 16
OUTPUT(MaxVal3); //returns 16

//example using a DATASET
SalesRecord := RECORD
    INTEGER OrderNumber;
    INTEGER SaleAmount;
END;
Sales := DATASET(
    {923,1001},
    {924,23},
    {925,3000},
    {926,3423},
    {927,9999},
    {928,1234},
);```
{931,113]}, SalesRecord);

OUTPUT(MAX(Sales,Sales.SaleAmount)); //returns 9999

See Also: MIN, AVE
# MERGE

The MERGE function returns a single dataset or index containing all the records from the datasets or indexes named in the `recordsetlist` or `recordsetset`. This is particularly useful for incremental data updates as it allows you to merge a smaller set of new records into an existing large dataset or index without having to re-process all the source data again.

The `recordsetset` form makes merging a variable number of datasets possible when used inside a GRAPH function.

To write a MERGEEd index to disk, you must use the BUILD action.

### Syntax

```mergetable
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>recordsetlist</code></td>
<td>A comma-delimited list of the datasets or indexes to merge, which must all be in exactly the same format and sort order.</td>
</tr>
<tr>
<td><code>fieldlist</code></td>
<td>A comma-delimited list of the fields that define the sort order.</td>
</tr>
<tr>
<td><code>recordsetset</code></td>
<td>A SET ([ds1,ds2,ds3]) of the datasets or indexes to merge, which must all be in exactly the same format.</td>
</tr>
<tr>
<td><code>SORTED</code></td>
<td>Specifies the sort order of the <code>recordsetlist</code>.</td>
</tr>
<tr>
<td><code>DEDUP</code></td>
<td>Optional. Specifies the result contains only records with unique values in the fields that specify the sort order <code>fieldlist</code>.</td>
</tr>
<tr>
<td><code>LOCAL</code></td>
<td>Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUT.</td>
</tr>
<tr>
<td><code>STABLE</code></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td><code>UNSTABLE</code></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><code>PARALLEL</code></td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td><code>numthreads</code></td>
<td>Optional. Try to evaluate this activity using <code>numthreads</code> threads.</td>
</tr>
<tr>
<td><code>ALGORITHM</code></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><code>ORDERED</code></td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td><code>ALGORITHM(name)</code></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

### Parameters

- **recordsetlist**: A comma-delimited list of the datasets or indexes to merge, which must all be in exactly the same format and sort order.
- **fieldlist**: A comma-delimited list of the fields that define the sort order.
- **recordsetset**: A SET ([ds1,ds2,ds3]) of the datasets or indexes to merge, which must all be in exactly the same format.
- **SORTED**: Specifies the sort order of the `recordsetlist`.
- **DEDUP**: Optional. Specifies the result contains only records with unique values in the fields that specify the sort order `fieldlist`.
- **LOCAL**: Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUT.
- **STABLE**: Optional. Specifies the input record order is significant.
- **UNSTABLE**: Optional. Specifies the input record order is not significant.
- **PARALLEL**: Optional. Try to evaluate this activity in parallel.
- **numthreads**: Optional. Try to evaluate this activity using `numthreads` threads.
- **ALGORITHM(name)**: Optional. Override the algorithm used for this activity.
- **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.
- **ORDERED**: Specifies the significance of the output record order.

### Return

MERGE returns a record set.
Example:

```ecl
ds1 := SORTED(DATASET([1,'A'],[1,'B'],[1,'C'],[1,'D'],[1,'E'],
    [1,'F'],[1,'G'],[1,'H'],[1,'I'],[1,'J'],
    (INTEGER1 number, STRING1 Letter)),
    letter, number);

ds2 := SORTED(DATASET([2,'A'],[2,'B'],[2,'C'],[2,'D'],[2,'E'],
    [2,'F'],[2,'G'],[2,'H'],[2,'I'],[2,'J'],
    (INTEGER1 number, STRING1 Letter)),
    letter, number);

ds3 := MERGE(ds1, ds2, SORTED(letter, number));

SetDS := [ds1, ds2];

ds4 := MERGE(SetDS, SORTED(letter, number));

OUTPUT(ds3);
OUTPUT(ds4);
```

See Also: BUILD
MERGEJOIN


<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setofdatasets</td>
<td>The SET of recordsets to process ([idx1,idx2,idx3]), typically INDEXes, which all must have the same format.</td>
</tr>
<tr>
<td>joincondition</td>
<td>An expression specifying how to match records in the setofdatasets.</td>
</tr>
<tr>
<td>SORTED</td>
<td>Specifies the sort order of records in the input setofdatasets and also the output sort order of the result set.</td>
</tr>
<tr>
<td>fields</td>
<td>A comma-delimited list of fields in the setofdatasets, which must be a subset of the input sort order. These fields must all be used in the joincondition as they define the order in which the fields are STEPPED.</td>
</tr>
<tr>
<td>jointype</td>
<td>Optional. An inner join if omitted, else one of the listed types below.</td>
</tr>
<tr>
<td>DEDUP</td>
<td>Optional. Specifies the output result set contains only unique records.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function’s STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

The MERGEJOIN function is a variation of the SET OF DATASETs forms of the MERGE and JOIN functions. Like MERGE, it merges records from the setofdatasets into a single result set, but like JOIN, it uses the joincondition and jointype to determine which records to include in the result set. It does not, however, use a TRANSFORM function to produce the result; it includes all records, unchanged, from the setofdatasets that match the joincondition.

Matching Logic

The record matching joincondition may contain two parts: a STEPPED condition that may optionally be AND-ed with non-STEPPED conditions. The STEPPED expression contains equality expressions of the fields from the SORTED option, ANDed together, using LEFT and RIGHT as dataset qualifiers. If not present, the STEPPED condition is deduced from the fields specified by the SORTED option.

The order of the datasets within the setofdatasets can be significant to the way the joincondition is evaluated. The joincondition is duplicated between adjacent pairs of datasets, which means that this joincondition:

LEFT.field = RIGHT.field

when applied against a setofdatasets containing three datasets, is logically equivalent to:

ds1.field = ds2.field AND ds2.field = ds3.field
Join Types:

The following `jointypes` produce the following types of results, based on the records matching produced by the `joincondition`:

<table>
<thead>
<tr>
<th>Jointype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INNER</td>
<td>Only those records that exist in all datasets in the <code>setofdatasets</code>.</td>
</tr>
<tr>
<td>LEFT OUTER</td>
<td>At least one record for every record in the first dataset in the <code>setofdatasets</code>.</td>
</tr>
<tr>
<td>LEFT ONLY</td>
<td>One record for every record in the first dataset in the <code>setofdatasets</code> for which there is no match in any of the subsequent datasets.</td>
</tr>
<tr>
<td>MOFN(min [,max])</td>
<td>One record for every record with matching records in min number of adjacent datasets within the <code>setofdatasets</code>. If max is specified, the record is not included if max number of dataset matches are exceeded.</td>
</tr>
</tbody>
</table>

Example:

```ecl
Rec := RECORD,MAXLENGTH(4096)
   STRING1 Letter;
   UNSIGNED1 DS;
END;
ds1 := DATASET([{'A',1},{'B',1},{'C',1},{'D',1},{'E',1}],Rec);
ds2 := DATASET([{'A',2},{'B',2},{'H',2},{'I',2},{'J',2}],Rec);
ds3 := DATASET([{'B',3},{'C',3},{'M',3},{'N',3},{'O',3}],Rec);
ds4 := DATASET([{'A',4},{'B',4},{'R',4},{'S',4},{'T',4}],Rec);
ds5 := DATASET([{'B',5},{'V',5},{'W',5},{'X',5},{'Y',5}],Rec);
SetDS := [ds1,ds2,ds3,ds4,ds5];
j1 := MERGEJOIN(SetDS,
   STEPPED(LEFT.Letter=RIGHT.Letter),
   SORTED(Letter));
j2 := MERGEJOIN(SetDS,
   STEPPED(LEFT.Letter=RIGHT.Letter),
   SORTED(Letter),LEFT OUTER);
j3 := MERGEJOIN(SetDS,
   STEPPED(LEFT.Letter=RIGHT.Letter),
   SORTED(Letter),LEFT ONLY);
j4 := MERGEJOIN(SetDS,
   STEPPED(LEFT.Letter=RIGHT.Letter),
   SORTED(Letter),MOFN(3));
j5 := MERGEJOIN(SetDS,
   STEPPED(LEFT.Letter=RIGHT.Letter),
   SORTED(Letter),MOFN(3,4));
OUTPUT(j1);
OUTPUT(j2);
OUTPUT(j3);
OUTPUT(j4);
OUTPUT(j5);
```

See Also: MERGE, JOIN, STEPPED
MIN(recordset, value [, KEYED ] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [( numthreads ) ] [, ALGORITHM( name ) ] )

MIN(valuelist)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set. This also may be the keyword GROUP to indicate finding the minimum value of the field in a group, when used in a RECORD structure to generate crosstab statistics.</td>
</tr>
<tr>
<td>value</td>
<td>The expression to find the minimum value of.</td>
</tr>
<tr>
<td>KEYED</td>
<td>Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.</td>
</tr>
<tr>
<td>valuelist</td>
<td>A comma-delimited list of expressions to find the minimum value of. This may also be a SET of values.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function’s STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: MIN returns a single value.

The MIN function either returns the minimum value from the specified recordset or the valuelist. It is defined to return zero if the recordset is empty.

Example:

```
MinVal2 := MIN(4,8,16,2,1);
SetVals := [4,8,16,2,1];
MinVal3 := MIN(SetVals);

OUTPUT(MinVal2); //returns 1
OUTPUT(MinVal3); //returns 1

//example using a DATASET
SalesRecord := RECORD
    INTEGER OrderNumber;
    INTEGER SaleAmount;
END;
Sales := DATASET([923,1001],
    (924,23),
    (925,3000),
    (926,3423),
    (927,9999),
```
See Also: MAX, AVE
NOLOCAL

**NOLOCAL***(data)*

<table>
<thead>
<tr>
<th>data</th>
<th>The name of a DATASET or INDEX attribute.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>NOLOCAL returns a record set or index.</td>
</tr>
</tbody>
</table>

The **NOLOCAL** function specifies that all subsequent operations on the `data` are performed on all nodes. This is typically used within a THISNODE operation. **Available for use only in Roxie.**

Example:

```ecl
ds := JOIN(SomeData, NOLOCAL(SomeIndex), LEFT.ID = RIGHT.ID);
```

See Also: ALLNODES, THISNODE, LOCAL
NONEMPTY

NONEMPTY(recordsetlist)

<table>
<thead>
<tr>
<th>recordsetlist</th>
<th>A comma-delimited list of record sets.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>NONEMPTY returns a record set.</td>
</tr>
</tbody>
</table>

The **NONEMPTY** function returns the first record set from the `recordsetlist` that contains any records. This is similar to using the EXISTS function in an IF expression to return one of two possible record sets.

Example:

```ecl
SalesRecord := RECORD
    INTEGER OrderNumber;
    INTEGER SaleAmount;
END;
Sales := DATASET(
    [{923,1001},
    {924,23},
    {925,3000},
    {926,3423},
    {927,9999},
    {931,113}], SalesRecord);
ds := NONEMPTY(Sales(SaleAmount>20000),
    Sales(SaleAmount>10000),
    Sales(SaleAmount>3000));
OUTPUT(ds);
```

See Also: EXISTS
NORMALIZE

NORMALIZE(recordset, expression, transform [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ]] [, ALGORITHM( name ) ] )

NORMALIZE(recordset, LEFT.childdataset, transform [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ]] [, ALGORITHM( name ) ] )

- **recordset**: The set of records to process.
- **expression**: A numeric expression specifying the total number of times to call the transform for that record.
- **transform**: The TRANSFORM function to call for each record in the recordset.
- **childdataset**: The field name of a child DATASET in the recordset. This must use the keyword LEFT as its qualifier.
- **UNORDERED**: Optional. Specifies the output record order is not significant.
- **ORDERED**: Specifies the significance of the output record order.
- **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.
- **STABLE**: Optional. Specifies the input record order is significant.
- **UNSTABLE**: Optional. Specifies the input record order is not significant.
- **PARALLEL**: Optional. Try to evaluate this activity in parallel.
- **numthreads**: Optional. Try to evaluate this activity using numthreads threads.
- **ALGORITHM**: Optional. Override the algorithm used for this activity.
- **name**: The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

Return: NORMALIZE returns a record set.

The **NORMALIZE** function normalizes child records out of a recordset where the child records are appended to the end of the parent data records. The purpose is to take variable-length flat-file records and split out the child information. The parent information can easily be extracted using either TABLE or PROJECT.

NORMALIZE Form 1

Form 1 processes through all records in the recordset performing the transform function the expression number of times on each record in turn.

TRANSFORM Function Requirements for Form 1

The **transform** function must take at least two parameters: a LEFT record of the same format as the recordset, and an integer COUNTER specifying the number of times the **transform** has been called for that record. The resulting record set format does not need to be the same as the input.

NORMALIZE Form 2

Form 2 processes through all records in the recordset iterating the transform function through all the childdataset records in each record in turn.
TRANSFORM Function Requirements for Form 2

The `transform` function must take at least one parameter: a RIGHT record of the same format as the child-dataset. The resulting record set format does not need to be the same as the input.

Example:

```ecl
//Form 1 example
NamesRec := RECORD
  UNSIGNED1 numRows;
  STRING20 thename;
  STRING20 addr1 := '';
  STRING20 addr2 := '';
  STRING20 addr3 := '';
  STRING20 addr4 := '';
END;
NamesTable := DATASET([ {1,'Kevin','10 Malt Lane'},
  {2,'Liz','10 Malt Lane','3 The cottages'},
  {0,'Mr Nobody'},
  {4,'Anywhere','Here','There','Near','Far'}],
  NamesRec);
OutRec := RECORD
  UNSIGNED1 numRows;
  STRING20 thename;
  STRING20 addr;
END;
OutRec NormIt(NamesRec L, INTEGER C) := TRANSFORM
  SELF := L;
  SELF.addr := CHOOSE(C, L.addr1, L.addr2, L.addr3,
    L.addr4);
END;
NormAddrs :=
  NORMALIZE(namesTable,LEFT.numRows,NormIt(LEFT,COUNTER));
OUTPUT(NormAddrs);
/* the result is:
numRows thename  addr
1       Kevin    10 Malt Lane
2       Liz      10 Malt Lane
2       Liz      3 The cottages
4       Anywhere Here
4       Anywhere There
4       Anywhere Near
4       Anywhere Far
*/
```
{3,'Mr Nobody', []},
{4,'Anywhere',[ {4,'Far'},
{4,'Here'},
{4,'There'},
{4,'Near'} ]},
DenormedRec;
ChildRec NewChildren(ChildRec R) := TRANSFORM
SELF := R;
END;
NewChilds := NORMALIZE(ds, LEFT.Children, NewChildren(RIGHT));
OUTPUT(NewChilds);

See Also: TRANSFORM Structure, RECORD Structure, DENORMALIZE
NOFOLD

[name := ] NOFOLD( expression )

<table>
<thead>
<tr>
<th>name</th>
<th>Optional. The identifier for this function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The expression to evaluate.</td>
</tr>
</tbody>
</table>

The NOFOLD function creates a barrier that prevents optimizations occurring between the expression and the context it is used in. This is used to prevent constant-folding in the context so that it may be evaluated as-is. Note that this does not prevent constant-folding within the expression itself. It is normally only used to prevent test cases being optimized into something completely different, or to temporarily work around bugs in the compiler.

Example:

```ecl
OUTPUT(2 * 2);  // is normally constant folded to:
OUTPUT(4);      // at compile time.

//However adding NOFOLD() around one argument prevents that
OUTPUT(NOFOLD(2) * 2);

//Adding NOFOLD() around the entire expression does NOT
//prevent folding within the argument:
OUTPUT(NOFOLD(2 * 2));
  //is the same as
OUTPUT(NOFOLD(4));
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
NOTHOR

\[\text{name} := \text{NOTHOR}(\text{action})\]

<table>
<thead>
<tr>
<th>name</th>
<th>Optional. The identifier for this action.</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>The action to execute.</td>
</tr>
</tbody>
</table>

The NOTHOR compiler directive indicates the action should not execute on thor, but inline instead, in a global context. You can only do very simple dataset operations within a NOTHOR, like filtering records or a simple PROJECT.

NOTHOR needs to be used around operations that use the superfile transactions, (such as the example below) where the compiler does not spot the appropriate context.

Example:

```
IMPORT STD;
rec := RECORD
  STRING10 S;
END;

srcnode := '10.239.219.2';
srcdir := '/var/lib/HPCCSystems/mydropzone/';
dir := STD.File.RemoteDirectory(srcnode,srcdir,'*.txt',TRUE);

//without NOTHOR this code gets this error:
// "Cannot call function AddSuperFile in a non-global context"
NOTHOR(SEQUENTIAL(
  STD.File.DeleteSuperFile('MultiSuper1'),
  STD.File.CreateSuperFile('MultiSuper1'),
  STD.File.StartSuperFileTransaction(),
  APPLY(dir,STD.File.AddSuperFile('MultiSuper1',
    STD.File.ExternalLogicalFileName(srcnode,srcdir+name))),
  STD.File.FinishSuperFileTransaction()));

F1 := DATASET('MultiSuper1', rec, THOR);
OUTPUT(F1,'testmulti1',OVERWRITE);
```

See Also: SEQUENTIAL
NOTIFY

```
```

<table>
<thead>
<tr>
<th>attributename</th>
<th>Optional. The identifier for this action.</th>
</tr>
</thead>
<tbody>
<tr>
<td>event</td>
<td>The EVENT function, or a case-insensitive string constant naming the event to generate.</td>
</tr>
<tr>
<td>parm</td>
<td>A case-insensitive string constant containing the event's parameter as either a single asterisk ('*') or an XML string beginning and ending with &quot;Event&quot; tags and user-defined tags within those to contain the specific extra information to pass along with the event.</td>
</tr>
<tr>
<td>expression</td>
<td>Optional. A case-insensitive string constant allowing simple message passing, to restrict the event to a specific workunit.</td>
</tr>
</tbody>
</table>

The NOTIFY action fires the event so that the WAIT function or WHEN workflow service can proceed with operations they are defined to perform.

The expression parameter allows you to define a service in ECL that is initiated by an event, and only responds to the workunit that initiated it.

Example:

```ecl
//run this first
FUNCTION
  doMyService := |
    O := OUTPUT('Did a Service for: ' + 'EVENTNAME=' + EVENTNAME);
    N := NOTIFY(EVENT('MyServiceComplete',
      '<Event><returnTo>FRED</returnTo></Event>'),
      EVENTEXTRA('returnTo'));
  RETURN WHEN(EVENTEXTRA('returnTo'), ORDERED(O, N));
OUTPUT(doMyService) : WHEN('MyService');
END;

// run this in a separate workunit after the first part above completes:
NOTIFY('MyService',
  '<Event><returnTo>'+ WORKUNIT + '</returnTo></Event>');
WAIT('MyServiceComplete');
OUTPUT('WORKUNIT DONE')
```

See Also: EVENT, EVENTNAME, EVENTEXTRA, CRON, WHEN, WAIT
ORDERED

**[attributename := ] ORDERED( actionlist )**

<table>
<thead>
<tr>
<th>attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>Optional. The action name, which turns the action into an attribute definition, therefore not executed until the attribute name is used as an action.</td>
</tr>
<tr>
<td>actionlist</td>
<td>A comma-delimited list of the actions to execute in order. These may be ECL actions or external actions.</td>
</tr>
</tbody>
</table>

The **ORDERED** action executes the items in the **actionlist** in the order in which they appear in the **actionlist**. This is useful when a subsequent action requires the output of a precedent action.

It has the ordering requirements of **SEQUENTIAL**. This is most useful for ordering actions which do not have anything in common, for example, generating files and then sending email. If there is any chance of a shared value which may change meaning, you should use **SEQUENTIAL**.

**ORDERED** has no effect on **PERSISTed** attributes.

**Example:**

```ecl
Action1 := OUTPUT(A_People,OutputFormat1,'//hold01/fred.out');
Action2 := OUTPUT(Person,{Person.per_first_name,Person.per_last_name})
Action2 := OUTPUT(Person,{Person.per_last_name}));
  //by naming these actions, they become inactive attributes
  //that only execute when the attribute names are called as actions
ORDERED(Action1,PARALLEL(Action2,Action3));
  //executes Action1 alone, and then executes Action2 and Action3 together
```

**See Also:** PARALLEL, PERSIST, SEQUENTIAL
OUTPUT

[\texttt{attr := \{} \texttt{OUTPUT( recordset [, [ \texttt{format} [, \texttt{file [\texttt{thorfileoptions} [, \texttt{NOXPATH} [, \texttt{UNORDERED | ORDERED( bool) [, \texttt{STABLE | UNSTABLE} [, \texttt{PARALLEL [ (numthreads) ]} [, \texttt{ALGORITHM( name) ]}]})};

[\texttt{attr := \{} \texttt{OUTPUT( recordset, [ \texttt{format} [, \texttt{file [, \texttt{CSV [ (csvoptions) [, \texttt{csvfileoptions} [, \texttt{NOXPATH} [, \texttt{UNORDERED | ORDERED( bool) [, \texttt{STABLE | UNSTABLE} [, \texttt{PARALLEL [ (numthreads) ]} [, \texttt{ALGORITHM( name) ]}]})];

[\texttt{attr := \{} \texttt{OUTPUT( recordset, [ \texttt{format} [, \texttt{file [, \texttt{XML [ (xmloptions) [, \texttt{xmlfileoptions} [, \texttt{NOXPATH} [, \texttt{UNORDERED | ORDERED( bool) [, \texttt{STABLE | UNSTABLE} [, \texttt{PARALLEL [ (numthreads) ]} [, \texttt{ALGORITHM( name) ]}]})];

[\texttt{attr := \{} \texttt{OUTPUT( recordset, [ \texttt{format} [, \texttt{file [, \texttt{JSON [ (jsonoptions) [, \texttt{jsonfileoptions} [, \texttt{NOXPATH} [, \texttt{UNORDERED | ORDERED( bool) [, \texttt{STABLE | UNSTABLE} [, \texttt{PARALLEL [ (numthreads) ]} [, \texttt{ALGORITHM( name) ]}]})];

[\texttt{attr := \}} \texttt{OUTPUT( recordset, [ \texttt{format} [, \texttt{file [, \texttt{PIPE [ (pipeoptions) [, \texttt{NOXPATH} [, \texttt{UNORDERED | ORDERED( bool) [, \texttt{STABLE | UNSTABLE} [, \texttt{PARALLEL [ (numthreads) ]} [, \texttt{ALGORITHM( name) ]}]})];

[\texttt{attr := \}} \texttt{OUTPUT( recordset, [ \texttt{format} [, \texttt{NAMED [ (name) [,\texttt{EXTEND [ (ALL) [, \texttt{NOXPATH} [, \texttt{UNORDERED | ORDERED( bool) [, \texttt{STABLE | UNSTABLE} [, \texttt{PARALLEL [ (numthreads) ]} [, \texttt{ALGORITHM( name) ]}]})];

[\texttt{attr := \}} \texttt{OUTPUT( expression [ ,\texttt{NAMED [ (name) ]} [, \texttt{NOXPATH} [, \texttt{UNORDERED | ORDERED( bool) [, \texttt{STABLE | UNSTABLE} [, \texttt{PARALLEL [ (numthreads) ]} [, \texttt{ALGORITHM( name) ]}]})];

[\texttt{attr := \}} \texttt{OUTPUT( recordset [ , \texttt{THOR [ \texttt{NOXPATH} [, \texttt{UNORDERED | ORDERED( bool) [, \texttt{STABLE | UNSTABLE} [, \texttt{PARALLEL [ (numthreads) ]} [, \texttt{ALGORITHM( name) ]}]})];

\begin{tabular}{|l|l|}
\hline
\texttt{attr} & Optional. The action name, which turns the action into a definition, therefore not executed until the \texttt{attr} is used as an action. \\
\hline
\texttt{recordset} & The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set. \\
\hline
\texttt{format} & Optional. The format of the output records. If omitted, all fields in the \texttt{recordset} are output. If not omitted, this must be either the name of a previously defined RECORD structure definition or an "on-the-fly" record layout enclosed within curly braces ({}), and must meet the same requirements as a RECORD structure for the TABLE function (the "vertical slice" form) by defining the type, name, and source of the data for each field. \\
\hline
\texttt{file} & Optional. The logical name of the file to write the records to. See the Scope & Logical Filenames section of the Language Reference for more on logical filenames. If omitted, the formatted data stream only returns to the command issuer (command line or IDE) and is not written to a disk file. \\
\hline
\texttt{thorfileoptions} & Optional. A comma-delimited list of options valid for a THOR/FLAT file (see the section below for details). \\
\hline
\texttt{NOXPATH} & Specifies any XPATHS defined in the \texttt{format} or the RECORD structure of the \texttt{recordset} are ignored and field names are used instead. This allows control of whether XPATHS are used for output, so that XPATHS that were meant only for xml or json input can be ignored for output. \\
\hline
\end{tabular}
| **UNORDERED** | Optional. Specifies the output record order is not significant. |
| **ORDERED** | Specifies the significance of the output record order. |
| **bool** | When False, specifies the output record order is not significant. When True, specifies the default output record order. |
| **STABLE** | Optional. Specifies the input record order is significant. |
| **UNSTABLE** | Optional. Specifies the input record order is not significant. |
| **PARALLEL** | Optional. Try to evaluate this activity in parallel. |
| **numthreads** | Optional. Try to evaluate this activity using numthreads threads. |
| **ALGORITHM** | Optional. Override the algorithm used for this activity. |
| **name** | The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options. |
| **CSV** | Specifies the file is a field-delimited (usually comma separated values) ASCII file. |
| **csvoptions** | Optional. A comma-delimited list of options defining how the file is delimited. |
| **csvfileoptions** | Optional. A comma-delimited list of options valid for a CSV file (see the section below for details). |
| **XML** | Specifies the file is output as XML data with the name of each field in the format becoming the XML tag for that field's data. |
| **xmloptions** | Optional. A comma separated list of options that define how the output XML file is delimited. |
| **xmlfileoptions** | Optional. A comma-delimited list of options valid for an XML file (see the section below for details). |
| **JSON** | Specifies the file is output as JSON data with the name of each field in the format becoming the JSON tag for that field's data. |
| **jsonoptions** | Optional. A comma separated list of options that define how the output JSON file is delimited. |
| **jsonfileoptions** | Optional. A comma-delimited list of options valid for a JSON file (see the section below for details). |
| **PIPE** | Indicates the specified command executes with the recordset provided as standard input to the command. This is a "write" pipe. |
| **pipeoptions** | The name of a program to execute, which takes the file as its input stream, along with the options valid for an output PIPE. |
| **NAMED** | Specifies the result name that appears in the workunit. Not valid if the file parameter is present. |
| **name** | A string constant containing the result label. This must be a compile-time constant and meet the attribute naming requirements. This must be a valid label (See Definition Name Rules) |
| **EXTEND** | Optional. Specifies appending to the existing NAMED result name in the workunit. Using this feature requires that all NAMED OUTPUTs to the same name have the EXTEND option present, including the first instance. |
| **ALL** | Optional. Specifies all records in the recordset are output to the ECL IDE. |
| **expression** | Any valid ECL expression that results in a single scalar value. |
| **THOR** | Specifies the resulting recordset is stored as a file on disk, "owned" by the workunit, instead of storing it directly within the workunit. The name of the file in the DFU is scope::RESULT::workunitid. |
The **OUTPUT** action produces a recordset result from the supercomputer, based on which form and options you choose. If no *file* to write to is specified, the result is stored in the workunit and returned to the calling program as a data stream.

### OUTPUT Field Names

Field names in an "on the fly" record format {...} must be unique or a syntax error results. For example:

```
OUTPUT(person(), {module1.attr1, module2.attr1});
```

will result in a syntax error. Output Field Names are assumed from the definition names.

To get around this situation, you can specify a unique name for the output field in the on-the-fly record format, like this:

```
OUTPUT(person(), {module1.attr1, name := module2.attr1});
```

### OUTPUT Thor/Flat Files

```
```

- **CLUSTER**
  Optional. Specifies writing the file to the specified list of target clusters. If omitted, the file is written to the cluster on which the workunit executes. The number of physical file parts written to disk is always determined by the number of nodes in the cluster on which the workunit executes, regardless of the number of nodes on the target cluster(s).

- **target**
  A comma-delimited list of string constants containing the names of the clusters to write the file to. The names must be listed as they appear on the ECL Watch Activity page or returned by the Std.System.Thorlib.Group() function, optionally with square brackets containing a comma-delimited list of node-numbers (1-based) and/or ranges (specified with a dash, as in n-m) to indicate the specific set of nodes to write to.

- **ENCRYPT**
  Optional. Specifies writing the file to disk using both 256-bit AES encryption and LZW compression.

- **key**
  A string constant containing the encryption key to use to encrypt the data.

- **COMPRESSED**
  Optional. Specifies writing the file using LZW compression.

- **OVERWRITE**
  Optional. Specifies overwriting the file if it already exists.

- **UPDATE**
  Specifies that the file should be rewritten only if the code or input data has changed.

- **EXPIRE**
  Optional. Specifies the file is a temporary file that may be automatically deleted after the specified number of days since the file was read.

- **days**
  Optional. The number of days from last file read after which the file may be automatically deleted. If EXPIRE is specified without number of days, it defaults to use the **ExpiryDefault** setting in Sasha.

This form writes the *recordset* to the specified *file* in the specified *format*. If the *format* is omitted, all fields in the *recordset* are output. If the *file* is omitted, then the result is sent back to the requesting program (usually the ECL IDE or the program that sent the SOAP query to a Roxie).

Example:

```
OutputFormat1 := RECORD
  People.firstname;
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
### OUTPUT CSV Files

```
[attr := ] OUTPUT(recordset, [ format ], file , CSV( (csvoptions) ) [, CLUSTER( target )] [,ENCRYPT(key) ] [,COMPRESSED]
[ ,OVERWRITE],[ , UPDATE] [, EXPIRE( [ days ] ) ] )
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLUSTER</strong></td>
<td>Optional. Specifies writing the file to the specified list of target clusters. If omitted, the file is written to the cluster on which the workunit executes. The number of physical file parts written to disk is always determined by the number of nodes in the cluster on which the workunit executes, regardless of the number of nodes on the target cluster(s).</td>
</tr>
<tr>
<td><strong>target</strong></td>
<td>A comma-delimited list of string constants containing the names of the clusters to write the file to. The names must be listed as they appear on the ECL Watch Activity page or returned by the Std.System.Thorlib.Group() function, optionally with square brackets containing a comma-delimited list of node-numbers (1-based) and/or ranges (specified with a dash, as in n-m) to indicate the specific set of nodes to write to.</td>
</tr>
<tr>
<td><strong>ENCRYPT</strong></td>
<td>Optional. Specifies writing the file to disk using both 256-bit AES encryption and LZW compression.</td>
</tr>
<tr>
<td><strong>key</strong></td>
<td>A string constant containing the encryption key to use to encrypt the data.</td>
</tr>
<tr>
<td><strong>COMPRESSED</strong></td>
<td>Optional. Specifies writing the file using LZW compression.</td>
</tr>
<tr>
<td><strong>OVERWRITE</strong></td>
<td>Optional. Specifies overwriting the file if it already exists.</td>
</tr>
<tr>
<td><strong>UPDATE</strong></td>
<td>Specifies that the file should be rewritten only if the code or input data has changed.</td>
</tr>
<tr>
<td><strong>EXPRESS</strong></td>
<td>Optional. Specifies the file is a temporary file that may be automatically deleted after the specified number of days.</td>
</tr>
<tr>
<td><strong>days</strong></td>
<td>Optional. The number of days after which the file may be automatically deleted. If omitted, the default is seven (7).</td>
</tr>
</tbody>
</table>

This form writes the `recordset` to the specified `file` in the specified `format` as a comma separated values ASCII file. The valid set of `csvoptions` are:

- **HEADING**( [ headerText [, footertext ] ] [, SINGLE ][, FORMAT(stringfunction) ] )
- **SEPARATOR**( delimiters )
TERMINATOR( delimiters )

QUOTE( [ delimiters ] )

ASCII | EBCDIC | UNICODE

| HEADING | Specifies file headers and footers. |
| headertext | Optional. The text of the header record to place in the file. If omitted, the field names are used. |
| footertext | Optional. The text of the footer record to place in the file. If omitted, no footertext is output. |

SINGLE | Optional. Specifies the headertext is written only to the beginning of part 1 and the footertext is written only at the end of part n (producing a "standard" CSV file). If omitted, the headertext and footertext are placed at the beginning and end of each file part (useful for producing complex XML output). |

FORMAT | Optional. Specifies the headertext should be formatted using the stringfunction. |
| stringfunction | Optional. The function to use to format the column headers. This can be any function that takes a single string parameter and returns a string result |

SEPARATOR | Specifies the field delimiters. |
| delimiters | A single string constant (or comma-delimited list of string constants) that define the character(s) used to delimit the data in the CSV file. |

TERMINATOR | Specifies the record delimiters. |

QUOTE | Specifies the quotation delimiters for string values that may contain SEPARATOR or TERMINATOR delimiters as part of their data. |

ASCII | Specifies all output is in ASCII format, including any EBCDIC or UNICODE fields. |

EBCDIC | Specifies all output is in EBCDIC format except the SEPARATOR and TERMINATOR (which are expressed as ASCII values). |

UNICODE | Specifies all output is in Unicode UTF8 format |

If none of the ASCII, EBCDIC, or UNICODE options are specified, the default output is in ASCII format with any UNICODE fields in UTF8 format. The other default csvoptions are:

```
CSV(HEADING('',''), SEPARATOR(','), TERMINATOR('
'), QUOTE())
```

Example:

```ecl
//SINGLE option writes the header only to the first file part:
OUTPUT(ds,,'~thor::outdata.csv',CSV(HEADING(SINGLE)));

//This example writes the header and footer to every file part:
OUTPUT(XMLds,,'~thor::outdata.xml',CSV(HEADING('<XML>','</XML>')));

//FORMAT option writes the header using the specified formatting function:
IMPORT STD;
OUTPUT(ds,,'~thor::outdata.csv',CSV(HEADING(FORMAT(STD.Str.ToUpperCase))));
```

## OUTPUT XML Files

```
[attr := ] OUTPUT(recordset, [ format ], file . XML [ (xmloptions) ][, ENCRYPT( key ) ][, CLUSTER( target ) ][, OVERWRITE ][, UPDATE ][, EXPIRE([ days ])])
```

CLUSTER | Optional. Specifies writing the file to the specified list of target clusters. If omitted, the file is written to the cluster on which the workunit executes. The number of physical
file parts written to disk is always determined by the number of nodes in the cluster on which the workunit executes, regardless of the number of nodes on the target cluster(s).

| target | A comma-delimited list of string constants containing the names of the clusters to write the file to. The names must be listed as they appear on the ECL Watch Activity page or returned by the Std.System.Thorlib.Group() function, optionally with square brackets containing a comma-delimited list of node-numbers (1-based) and/or ranges (specified with a dash, as in n-m) to indicate the specific set of nodes to write to. |
| ENCRYPT | Optional. Specifies writing the file to disk using both 256-bit AES encryption and LZW compression. |
| key | A string constant containing the encryption key to use to encrypt the data. |
| OVERWRITE | Optional. Specifies overwriting the file if it already exists. |
| UPDATE | Specifies that the file should be rewritten only if the code or input data has changed. |
| EXPIRE | Optional. Specifies the file is a temporary file that may be automatically deleted after the specified number of days. |
| days | Optional. The number of days after which the file may be automatically deleted. If omitted, the default is seven (7). |

This form writes the recordset to the specified file as XML data with the name of each field in the specified format becoming the XML tag for that field's data. The valid set of xmloptions are:

- **'rowtag'**
- **HEADING( headertext [, footertext ] )**
- **TRIM**
- **OPT**

| rowtag | The text to place in record delimiting tag. |
| HEADING | Specifies placing header and footer records in the file. |
| headertext | The text of the header record to place in the file. |
| footertext | The text of the footer record to place in the file. |
| TRIM | Specifies removing trailing blanks from string fields before output. |
| OPT | Specifies omitting tags for any empty string field from the output. |

If no xmloptions are specified, the defaults are:

```xml
XML('Row',HEADING('<Dataset>
','</Dataset>
'))
```

Example:

```ecl
R := {STRING10 fname,STRING12 lname};
B := DATASET([{'Fred','Bell'},{'George','Blanda'},{'Sam',''}],R);
OUTPUT(B,,'fred1.xml', XML); // writes B to the fred1.xml file
/* the Fred1.XML file looks like this:
   <Dataset>
      <Row><fname>Fred </fname><lname>Bell</lname></Row>
      <Row><fname>George</fname><lname>Blanda </lname></Row>
      <Row><fname>Sam </fname><lname></lname></Row>
   </Dataset> */
```
### OUTPUT JSON Files

```
[attr := ] OUTPUT(recordset, [ format ], file, JSON [(jsonoptions) ][, ENCRYPT( key ) ][, CLUSTER( target ) ][, OVERWRITE ][, UPDATE ][, EXPIRE([ days ])] )
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CLUSTER</strong></td>
<td>Optional. Specifies writing the file to the specified list of target clusters. If omitted, the file is written to the cluster on which the workunit executes. The number of physical file parts written to disk is always determined by the number of nodes in the cluster on which the workunit executes, regardless of the number of nodes on the target cluster(s).</td>
</tr>
<tr>
<td><strong>target</strong></td>
<td>A comma-delimited list of string constants containing the names of the clusters to write the file to. The names must be listed as they appear on the ECL Watch Activity page or returned by the Std.System.Thorlib.Group() function, optionally with square brackets containing a comma-delimited list of node-numbers (1-based) and/or ranges (specified with a dash, as in n-m) to indicate the specific set of nodes to write to.</td>
</tr>
<tr>
<td><strong>ENCRYPT</strong></td>
<td>Optional. Specifies writing the file to disk using both 256-bit AES encryption and LZW compression.</td>
</tr>
<tr>
<td><strong>key</strong></td>
<td>A string constant containing the encryption key to use to encrypt the data.</td>
</tr>
<tr>
<td><strong>OVERWRITE</strong></td>
<td>Optional. Specifies overwriting the file if it already exists.</td>
</tr>
<tr>
<td><strong>UPDATE</strong></td>
<td>Specifies that the file should be rewritten only if the code or input data has changed.</td>
</tr>
<tr>
<td><strong>EXPIRE</strong></td>
<td>Optional. Specifies the file is a temporary file that may be automatically deleted after the specified number of days.</td>
</tr>
<tr>
<td><strong>days</strong></td>
<td>Optional. The number of days after which the file may be automatically deleted. If omitted, the default is seven (7).</td>
</tr>
</tbody>
</table>

This form writes the `recordset` to the specified `file` as JSON data with the name of each field in the specified `format` becoming the JSON tag for that field's data. The valid set of `jsonoptions` are:

'rowtag'

**HEADING**( headertext [, footertext ])

**TRIM**

**OPT**

| **rowtag** | The text to place in record delimiting tag. |
ECL Language Reference
Built-in Functions and Actions

**HEADING**
Specifies placing header and footer records in the file.

**headertext**
The text of the header record to place in the file.

**footertext**
The text of the footer record to place in the file.

**TRIM**
Specifies removing trailing blanks from string fields before output.

**OPT**
Specifies omitting tags for any empty string field from the output.

If no `jsonoptions` are specified, the defaults are:

```json
JSON('Row', HEADING('[,]'))
```

Example:

```ecl
R := {STRING10 fname, STRING12 lname};
B := DATASET([{'Fred', 'Bell'}, {'George', 'Blanda'}, {'Sam', ' '}], R);

OUTPUT(B, 'fred1.json', JSON); // writes B to the fred1.json file
/* the Fred1.json file looks like this:
{
"Row": [
{"fname": "Fred      ", "lname": "Bell        "},
{"fname": "George    ", "lname": "Blanda      "},
{"fname": "Sam       ", "lname": "            "}
]
*/

OUTPUT(B, 'fred2.json', JSON('MyResult', HEADING('[,]')));
/* the Fred2.json file looks like this:
["MyResult": [
{"fname": "Fred      ", "lname": "Bell        "},
{"fname": "George    ", "lname": "Blanda      "},
{"fname": "Sam       ", "lname": "            "}
]]
```

## OUTPUT PIPE Files

```ecl
[ attr := ] OUTPUT(recordset, [ format ], PIPE( command [, CSV | XML] [, REPEAT] )
```

**PIPE**
Indicates the specified command executes with the recordset provided as standard input to the command. This is a "write" pipe.

**command**
The name of a program to execute, which takes the file as its input stream.

**CSV**
Optional. Specifies the output data format is CSV. If omitted, the format is raw.

**XML**
Optional. Specifies the output data format is XML. If omitted, the format is raw.

**REPEAT**
Optional. Indicates a new instance of the specified command executes for each row in the recordset.

This form sends the `recordset` in the specified `format` as standard input to the `command`. This is commonly known as an "output pipe."

Example:

```ecl
OUTPUT(A_People, PIPE('MyCommandLineProgram'), OVERWRITE);
// sends the A_People to MyCommandLineProgram as standard in
```

## Named OUTPUT

```ecl
[ attr := ] OUTPUT(recordset [, format ], NAMED( name ) [, EXTEND] [, ALL])
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
This form writes the recordset to the workunit with the specified name. This must be a valid label (See Definition Name Rules)

The EXTEND option allows multiple OUTPUT actions to the same named result. The ALL option is used to override the implicit CHOOSEN applied to interactive queries in the Query Builder program. This specifies returning all records.

Example:

```ecl
OUTPUT(CHOOSEN(people(firstname[1]='A'),10));
   // writes the A People to the workunit
OUTPUT(CHOOSEN(people(firstname[1]='A'),10),ALL);
   // writes all the A People to the workunit
OUTPUT(CHOOSEN(people(firstname[1]='A'),10),NAMED('fred'));
   // writes the A People to the fred named output
```

//a NAMED, EXTEND example:

```ecl
errMsgRec := RECORD
   UNSIGNED4 code;
   STRING text;
END;
makeErrMsg(UNSIGNED4 _code,STRING _text) := DATASET({[_code, _text]}, errMsgRec);
rptErrMsg(UNSIGNED4 _code,STRING _text) := OUTPUT(makeErrMsg(_code,_text),
   NAMED('ErrorResult'),EXTEND);
OUTPUT(DATASET({[100, 'Failed']},errMsgRec),NAMED('ErrorResult'),EXTEND);
   //Explicit syntax.
//Something else creates the dataset
OUTPUT(makeErrMsg(101, 'Failed again'),NAMED('ErrorResult'),EXTEND);
   //output and dataset handled elsewhere.
rptErrMsg(102, 'And again');
```

## OUTPUT Scalar Values

\[ \text{attr := } \text{OUTPUT( expression [ NAMED( name )] )} \]

This form is used to allow scalar expression output, particularly within SEQUENTIAL and PARALLEL actions.

Example:

```ecl
OUTPUT(10) // scalar value output
OUTPUT('Fred') // scalar value output
```

## OUTPUT Workunit Files

\[ \text{attr := } \text{OUTPUT( recordset , THOR )} \]

This form is used to store the resulting recordset as a file on disk "owned" by the workunit. The name of the file in the DFU is scope::RESULT::workunitid. This is useful when you want to view a large result recordset in the Query Builder program but do not want that much data to take up memory in the system data store.

Example:

```ecl
OUTPUT(Person(per_st='FL'), THOR)
   // output records to screen, but store the
   // result on disk instead of in the workunit
```

See Also: TABLE, DATASET, PIPE, CHOOSEN
PARALLEL

\[
[\text{definitionname} := ] \text{PARALLEL( actionlist )}
\]

- **definitionname**: Optional. The action name, which turns the action into a definition, therefore not executed until the `definitionname` is used as an action.
- **actionlist**: A comma-delimited list of the actions to execute simultaneously. These may be ECL actions or external actions.

The `PARALLEL` action allows the items in the action list to execute simultaneously. It does not force parallel execution, only allows it -- the compiler determines the actual execution order. This is already the default operative mode, so `PARALLEL` is only useful within the action list of a `SEQUENTIAL` set of actions.

Example:

```ecl
Act1 :=
OUTPUT(A_People,OutputFormat1,'//hold01/fred.out');
Act2 :=
OUTPUT(Person,{Person.per_first_name,Person.per_last_name});
Act2 := OUTPUT(Person,{Person.per_last_name});
```

//by naming these actions, they become inactive definitions
//that only execute when the definition names are called as actions

```ecl
SEQUENTIAL(Act1,PARALLEL(Act2,Act3));
```

//executes Act1 alone, and only when it's finished,
//executes Act2 and Act3 together

See Also: ORDERED, SEQUENTIAL
The `PARSE` function performs a text or XML parsing operation.

### PARSE Text Data

The first form operates on the `dataset`, finding records whose `data` contains a match for the `pattern`, producing a result set of those matches in the `result` format. If the `pattern` finds multiple matches in the `data`, then a result record is generated for each match. Each match for a PARSE is effectively a single path through the `pattern`. If there is more than one path that matches, then the `result` transform is either called once for each path, or if the BEST option is used, the path with the lowest penalty is selected.

If the `result` names a RECORD structure, then this form of PARSE operates like the TABLE function to generate the result set, but may also operate on variable length text. If the `result` names a TRANSFORM function, then the transform generates the result set. The TRANSFORM function must take at least one
parameter: a LEFT record of the same format as the dataset. The format of the resulting record set does not need to be the same as the input.

Flags can have the following values:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>Only return a row for the first match starting at a particular position.</td>
</tr>
<tr>
<td>ALL</td>
<td>Return a row for every possible match of the string at a particular position.</td>
</tr>
<tr>
<td>WHOLE</td>
<td>Only match the whole string.</td>
</tr>
<tr>
<td>NOSCANN</td>
<td>If a position matches, don’t continue searching for other matches.</td>
</tr>
<tr>
<td>SCAN</td>
<td>If a position matches, continue searching from the end of the match, otherwise continue from the next position.</td>
</tr>
<tr>
<td>SCAN ALL</td>
<td>Return matches for every possible start position. Use the TRIM function to eliminate parsing extraneous trailing blanks.</td>
</tr>
<tr>
<td>NOCASE</td>
<td>Perform a case insensitive comparison.</td>
</tr>
<tr>
<td>CASE</td>
<td>Perform a case sensitive comparison (this is the default).</td>
</tr>
<tr>
<td>SKIP(sep-pat)</td>
<td>Specify a pattern that can be inserted after each token in a search pattern. For example, SKIP (&quot;[^,&quot;\t]&quot; ) skips spaces and tabs between tokens.</td>
</tr>
<tr>
<td>KEEP(max)</td>
<td>Only keep the first max matches.</td>
</tr>
<tr>
<td>ATMOST(max)</td>
<td>Don’t produce any matches if there are more than max matches.</td>
</tr>
<tr>
<td>MAX</td>
<td>Return a row for the result that matches the longest sequence of the input.</td>
</tr>
<tr>
<td>MIN</td>
<td>Return a row for the result that matches the shortest sequence of the input.</td>
</tr>
<tr>
<td>MATCHED(rule)</td>
<td>Used when rule-reference is used in a user-matching function. If a rule-reference is not specified, the matching information may not be preserved.</td>
</tr>
<tr>
<td>MATCHED(ALL)</td>
<td>Retain all rule-names -- if they are used by user match functions.</td>
</tr>
<tr>
<td>NOT MATCHED</td>
<td>Generate a row if there were no matches on the input row. All calls to the MATCHED() function return false inside the resultstructure.</td>
</tr>
<tr>
<td>NOT MATCHED ONLY</td>
<td>Only generate a row if no matches were found.</td>
</tr>
<tr>
<td>BEST</td>
<td>Pick the match with the highest score (lowest penalty). If the MAX or MIN flags are also present, they are applied first. Only one match is returned unless the MANY option is also specified.</td>
</tr>
<tr>
<td>MANY</td>
<td>Return multiple matches for BEST, MAX, or MIN options.</td>
</tr>
<tr>
<td>PARSE</td>
<td>Implements Tomita parsing instead of regular expression parsing technology.</td>
</tr>
<tr>
<td>USE([ struct, ] x)</td>
<td>Specifies using a RULE pattern attribute defined further on in the code with the DEFINE(x) function, introducing a recursive grammar (the only recursion allowed in ECL). If the optional struct RECORD structure is specified, USE specifies using a RULE pattern attribute defined further on in the code with the DEFINE(x) function that produces a row result in the struct RECORD structure format (valid only with the PARSE option also present). USE is required on PARSE when any patterns cannot be found by walking the rules from the root down without following any USEs.</td>
</tr>
</tbody>
</table>

Example:

```ecl
rec := {STRING10000 line};
```
**ECL Language Reference**

**Built-in Functions and Actions**

```ecl
datafile := DATASET(
    {'Ge 34:2 And when Shechem the son of Hamor the Hivite, prince of the country, saw her,'+
     ' he took her, and lay with her, and defiled her.'},
    {'Ge 36:10 These are the names of Esau's sons; Eliphaz the son of Adah the wife of Esau,'+
     ' Reuel the son of Bashemath the wife of Esau.'},rec);

PATTERN ws1 := [',',',','];
PATTERN ws := ws1 ws1?;
PATTERN patStart := FIRST | ws;
PATTERN patEnd := LAST | ws;
PATTERN article := ['A','The','Thou','a','the','thou'];

TOKEN patWord := PATTERN('^[a-zA-Z]+$');
TOKEN Name := PATTERN('^[A-Z][a-zA-Z]+$');

RULE Namet := name OPT(ws ['the','king of','prince of'] ws name);
PATTERN produced := OPT(article ws) ['begat','father of','mother of'];
PATTERN produced_by := OPT(article ws) ['son of','daughter of'];
PATTERN produces_with := OPT(article ws) ['wife of'];

RULE relationtype ::= ( produced | produced_by | produces_with);
RULE progeny ::= namet ws relationtype ws namet;

results := RECORD
    STRING60 Le := MATCHTEXT(Name[1]);
    STRING60 Ri := MATCHTEXT(Name[2]);
    STRING30 RelationPhrase := MatchText(relationtype);
END;
outfile1 := PARSE(datafile,line,progeny,results,SCAN ALL);
OUTPUT(outfile1);
```

**PARSE XML Data**

The second form operates on an XML dataset, parsing the XML data and creating a result set using the result parameter, one output record per input. The expectation is that each row of data contains a complete block of XML. If the result names a RECORD structure, then this form of PARSE operates like the TABLE function to generate the result set.

If the result names a TRANSFORM function, then the transform generates the result set. The TRANSFORM function must take at least one parameter: a LEFT record of the same format as the dataset. The format of the resulting record set does not need to be the same as the input.

**NOTE:** XML reading and parsing can consume a large amount of memory, depending on the usage. In particular, if the specified xpath matches a very large amount of data, then a large data structure will be provided to the transform. Therefore, the more you match, the more resources you consume per match. For example, if you have a very large document and you match an element near the root that virtually encompasses the whole thing, then the whole thing will be constructed as a referenceable structure that the ECL can get at.

**Example:**

```ecl
linerec := { STRING line };
in1 := DATASET({
    '<ENTITY eid="P101" type="PERSON" subtype="MILITARY">"
      ' <ATTRIBUTE name="fullname">JOHN SMITH</ATTRIBUTE> ' +
      ' <ATTRIBUTE name="honorific">Mr.</ATTRIBUTE> ' +
      ' <ATTRIBUTE name="idNumber">W12468</ATTRIBUTE> ' +
      ' <ATTRIBUTE name="idType">pp</ATTRIBUTE> ' +
      ' <ATTRIBUTE name="issuingAuthority">JAPAN PASSPORT AUTHORITY</ATTRIBUTE> ' +
      ' <ATTRIBUTE name="country" value="L202"/> ' +
      ' <ATTRIBUTE name="age" value="19"/> ' +
  })
```
See Also: DATASET, OUTPUT, XMLENCODE, XMLDECODE, REGEXFIND, REGEXREPLACE, DEFINE

Extended PARSE Examples

This example parses raw phone numbers from a specific field in an input dataset into a single standard output containing just the numbers. A missing area code in the raw input results in three leading zeroes in the output.

infile := DATASET([{'5619994581'},{'15619994581'}, {'(561) 999-4581'},{'(561)999-4581'}, {'561-999-4581'},{'561 999 4581'}, {'561.999.4581'},{'561/999/4581'}, {'561 999-4581'},{'9994581'}, {'999-4581'}],{STRING20 rawnumber});
PATTERN areacode := OPT(FrontDigit) OPT(OpenParen) numbers length(3) OPT(CloseParen);

// Last Seven digits
PATTERN exchange := numbers length(3);
PATTERN lastfour := numbers length(4);
PATTERN seven := exchange OPT(Seperator) lastfour;

// Extension
PATTERN extension := ws alpha ws numbers;

// Phone Number
PATTERN phonenumber := OPT(areacode) OPT(Seperator) seven
opt(extension) ws;

layout_phone_append := RECORD
  infile;
  STRING10 clean_phone := MAP(NOT MATCHED(phonenumber) => '',
    NOT MATCHED(areacode) => '000' + MATCHTEXT(exchange) + MATCHTEXT(lastfour),
    MATCHTEXT(areacode/numbers) + MATCHTEXT(exchange) + MATCHTEXT(lastfour));
END;

outfile :=
  PARSE(infile, rawnumber, phonenumber, layout_phone_append,FIRST, NOT MATCHED, WHOLE);
OUTPUT(outfile);

This example parses a small subset of raw movie data into standard database fields:

IMPORT Std;
Layout_Actors_Raw := RECORD
  STRING120 IMDB_Actor_Desc;
END;

File_Actors := DATASET(["A.V., Subba Rao Chenchu Lakshmi (1958/I) <10>'
',
' Jayabheri (1959) <17>',
' Madalasa (1948) <3>'
',
' Mangalya Balam (1958) <12>'
',
' Mohini Bhasmasura (1938) <3>'
',
' Palletoori Pilla (1950) [Kamppana Dora] <4>'
',
' Peddamanushulu (1954) <6>'
',
' Sarangadhara (1957) <12>'
',
' Sri Seetha Rama Kalyanam (1961) <12>'
',
' Sri Venkateswara Mahatmyam (1960) [Akasa Raju] <5>
',
' Vara Vikrayam (1939) [Judge] <12>'
',
' Vinhyarani (1948) <7>
',
''],
'AA, Brynjar Adjo solidaritet (1985) [Ponker] <40>'
',
''],
'Aabel, Andreas Bor Borson Jr. (1938) [O.G. Hansen] <9>',
' Jeppe pa bjerget (1933) [En skomakerlaerling]',
' Kampen om tungtvannet (1948) <8>',
' Prinsessen som ingen kunne maqlbinde (1932) [Espen Askeland] <3>',
' Spokelse foreisker seg, Et (1946) [Et spokelse] <6>',
''],
'Aabel, Hauk (I) Alexander den store (1917) [Alexander Nyberg]',
' Du har lovet mig en kone! (1935) [Professoren] <6>',
' Glad gutt, En (1932) [Ola Nordistua] <1>',
' Jeppe pa bjerget (1933) [Jeppe] <1>',
' Morderen uten ansikt (1936)',
' Store barnedapen, Den (1931) [Evensen, kirketjener] <5>',
' Troll-Elgen (1927) [Piper, direktor] <9>',
' Ungen (1938) [Krestoffer] <8>',
' Valfangare (1939) [Jensen Sr.] <4>"]
© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
ds := DATASET([[1],['222+33*(1+2+(1))']], r1);

TOKEN val := PATTERN('[0-9]+');
RULE expr := SELF '*' SELF | SELF '+' SELF | SELF '(' SELF ')' | val;

parsed_record := RECORD
  STRING result := MATCHTEXT(expr);
END;

PARSE(ds, value, expr, parsed_record,PARSE);

Another example of Tomita parsing:

// This example demonstrates the use of productions in PARSE code
//(only supported in the Tomita version of PARSE)

PATTERN ws := ['', '	'];
TOKEN number := PATTERN('[0-9]+');
TOKEN plus := '+';
TOKEN minus := '-';
attrRec := RECORD  //record structure for Tomita parsing
  INTEGER val;
END;
RULE(attrRec) e0 := '(' USE(attrRec,expr)? ')'    //USE = forward reference to "expr" definition
  | number                   TRANSFORM(attrRec, SELF.val := (INTEGER)$1);
  | minus SELF               TRANSFORM(attrRec, SELF.val := -$2.val);
RULE(attrRec) e1 :=  e0
  | SELF '*' e0              TRANSFORM(attrRec, SELF.val := $1.val * $3.val);
  | SELF '/' e0              TRANSFORM(attrRec, SELF.val := $1.val / $3.val);
RULE(attrRec) e2 :=  e1
  | SELF plus e1             TRANSFORM(attrRec, SELF.val := $1.val + $3.val);
  | SELF minus e1            TRANSFORM(attrRec, SELF.val := $1.val - $3.val);
RULE(attrRec) expr := e2;

infile := DATASET([{'1+2*3'},{'1+2*100'},{'1+2+(3+4)*4/2'},{'-4*5'}], { STRING line });
resultsRec := RECORD
  RECORDOF(infile);
  attrRec;
  STRING exprText;
  INTEGER value3;
END;
resultsRec extractResults(infile L, attrRec attr) := TRANSFORM
  SELF := L;
  SELF := attr;
  SELF.exprText := MATCHTEXT;
  SELF.value3 := MATCHROW(e0[3]).val;
END;

OUTPUT(PARSE(infile,line,expr,extractResults(LEFT, $1),FIRST,WHOLE,PARSE,SKIP(ws+));
PIPE

\[ \text{PIPE}( \text{command, recorddef [, CSV | XML ]}) \]

\[ \text{PIPE}( \text{recordset, command[, recorddef]}, \text{REPEAT}[, CSV | XML ][, OUTPUT(CSV | XML )][, GROUP][, UNORDERED | ORDERED( bool )][, STABLE | UNSTABLE ][, PARALLEL [ ( numthreads )] ][, ALGORITHM( name )]) \]

command
- The name of a program to execute, which must take any input data through stdin and produce its output through stdout. This program must have already been deployed on the HPCC Systems cluster in the Thor instance directory (such as: /var/lib/HPCCSystems/mythor/) but that can be overridden by the externalProgDir environment setting for the Thor cluster).

recorddef
- The RECORD structure format for output. If omitted, output is the same as the input format.

CSV
- Optional. In form 1 (and as the parameter to the OUTPUT option), specifies the output data format is CSV. In form 2, specifies the input data format is CSV. If omitted, the format is raw.

XML
- Optional. In form 1 (and as the parameter to the OUTPUT option), specifies the output data format is XML. In form 2, specifies the input data format is XML. If omitted, the format is raw.

recordset
- The input dataset.

REPEAT
- Optional. Specifies a new instance of the command program is created for each row in the recordset.

OUTPUT
- Optional. Specifies CSV or XML result data format.

GROUP
- Optional. Specifies each result record is generated in a separate GROUP (only if REPEAT is specified).

UNORDERED
- Optional. Specifies the output record order is not significant.

ORDERED
- Specifies the significance of the output record order.

bool
- When False, specifies the output record order is not significant. When True, specifies the default output record order.

STABLE
- Optional. Specifies the input record order is significant.

UNSTABLE
- Optional. Specifies the input record order is not significant.

PARALLEL
- Optional. Try to evaluate this activity in parallel.

numthreads
- Optional. Try to evaluate this activity using numthreads threads.

ALGORITHM
- Optional. Override the algorithm used for this activity.

name
- The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

Return:
- PIPE returns a record set.

The PIPE function allows ECL code to launch an external command program on each node, effectively parallelizing a non-parallel processing program. PIPE has two forms:

Form 1 takes no input, executes the command, and produces its output in the recorddef format. This is an "input" pipe (like the PIPE option on a DATASET definition).
Form 2 takes the input *recordset*, executes the *command*, producing output in the *recorddef* format. This is a "through" pipe.

**Example:**

```ecl
namesRecord := RECORD
  STRING10 forename;
  STRING10 surname;
  STRING2 nl := '\r\n';
END;

d := PIPE('pipeRead 200', namesRecord); //form 1 - input pipe

t := PIPE(d, 'pipeThrough'); //form 2 - through pipe

OUTPUT(t,,PIPE('pipeWrite \thordata\names.all')); //output pipe

//Form 2 with XML input:
namesRecord := RECORD
  STRING10 Firstname{xpath('/Name/FName')};
  STRING10 Lastname{xpath('/Name/LName')};
END;

p := PIPE('echo <Name><FName>George</FName><LName>Jetson</LName></Name>', namesRecord, XML);
OUTPUT(p);
```

See Also: OUTPUT, DATASET
**POWER**

**POWER(base, exponent)**

<table>
<thead>
<tr>
<th>base</th>
<th>The real number to raise.</th>
</tr>
</thead>
<tbody>
<tr>
<td>exponent</td>
<td>The real power to raise x to.</td>
</tr>
</tbody>
</table>

**Return:** POWER returns a single real value.

The **POWER** function returns the result of the **base** raised to the **exponent** power.

**Example:**

```plaintext
MyCube := POWER(2.0,3.0); // = 8.0
MySquare := POWER(3.0,2.0); // = 9.0
OUTPUT(MyCube);
OUTPUT(MySquare);
```

**See Also:** SQRT, EXP, LN
**PRELOAD**

**PRELOAD**(*file[, nbr]*)

- **file**: The name of a DATASET definition.
- **nbr**: Optional. An integer constant specifying how many indexes to create "on the fly" for speedier access to the specified DATASET file (only). If > 1000, specifies the amount of memory set aside for these indexes.

**Return:** PRELOAD returns a record set.

The **PRELOAD** function leaves the *file* in memory after loading (valid only for Data Delivery Engine (Roxie) use). This is exactly equivalent to using the PRELOAD option on the DATASET definition.

**Example:**

```ecl
MyFile := DATASET('MyFile',{STRING20 F1, STRING20 F2},THOR);
COUNT(PRELOAD(MyFile))
```

See Also: DATASET
PROCESS

**PROCESS**(*recordset*, *datarow*, *datasettransform*, *rowtransform* [, **LOCAL**] [, **UNORDERED** | **ORDERED**(bool) ] [, **STABLE** | **UNSTABLE** ] [, **PARALLEL** [ (numthreads) ]] [, **ALGORITHM**(name) ])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>recordset</em></td>
<td>The set of records to process.</td>
</tr>
<tr>
<td><em>datarow</em></td>
<td>The initial RIGHT record to process, typically expressed by the ROW function.</td>
</tr>
<tr>
<td><em>datasettransform</em></td>
<td>The TRANSFORM function to call for each record in the recordset.</td>
</tr>
<tr>
<td><em>rowtransform</em></td>
<td>The TRANSFORM function to call to produce the next RIGHT record for the datasettransform.</td>
</tr>
<tr>
<td><strong>LOCAL</strong></td>
<td>Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.</td>
</tr>
<tr>
<td><strong>UNORDERED</strong></td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td><strong>ORDERED</strong></td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td><em>bool</em></td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><strong>STABLE</strong></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td><strong>UNSTABLE</strong></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><strong>PARALLEL</strong></td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td><em>numthreads</em></td>
<td>Optional. Try to evaluate this activity using <em>numthreads</em> threads.</td>
</tr>
<tr>
<td><strong>ALGORITHM</strong></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><em>name</em></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

**Return:** PROCESS returns a record set.

The **PROCESS** function operates in a similar manner to **ITERATE** in that it processes through all records in the *recordset* one pair of records at a time, performing the *datasettransform* function on each pair of records in turn. The first record is passed to the *datasettransform* as the first left record, paired with the *datarow* as the right record. The *rowtransform* is used to construct the right record for the next pair. If either the *datasettransform* or the *rowtransform* contains a SKIP, then no record is produced by the *datasettransform* for the skipped record.

**TRANSFORM Function Requirements - PROCESS**

The *datasettransform* and *rowtransform* functions both must take at least two parameters: a LEFT record of the same format as the *recordset* and a RIGHT record of the same format as the *datarow*. The format of the resulting record set for the *datasettransform* both must be the same as the input *recordset*. The format of the resulting record set for the *rowtransform* both must be the same as the initial *datarow*. Optionally, the *datasettransform* may take a third parameter: an integer COUNTER specifying the number of times the transform has been called for the *recordset* or the current group in the *recordset* (see the GROUP function).

**Examples:**

```ecl
DSrec := RECORD
  STRING4 Letter;
  STRING4 LeftRecIn := '';
  STRING4 RightRecIn := '';
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
StateRec := RECORD
  STRING2 Letter;
END;

ds := DATASET([{'AA'},{'BB'},{'CC'},{'DD'},{'EE'}], DSrec);

DSrec DSxform(DSrec L, StateRec R) := TRANSFORM
  SELF.LeftRecIn := L.Letter;
  SELF.RightRecIn := R.Letter;
END;
StateRec ROWxform(DSrec L, StateRec R) := TRANSFORM
END;
p := PROCESS(ds,
  ROW({'ZZ'}, StateRec),
  DSxform(LEFT,RIGHT),
  ROWxform(LEFT,RIGHT));
OUTPUT(p);
/* Result:
  AAAZ AA ZZ
  BBAZ BB AZ
  CCBA CC BA
  DDCB DD CB
  EEDC EE DC */

/* *************************************************************************/
/* This example uses different information for state tracking            */
/* (the point of the PROCESS function) through the input record set.    */
/* *************************************************************************/

w1 := RECORD
  STRING v{MAXLENGTH(100)};
END;

s1 := RECORD
  BOOLEAN priorA;
END;

ds := DATASET([{'B'},{'A'},{'C'},{'D'}], w1);
s1 doState(w1 l, s1 r) := TRANSFORM
  SELF.priorA := l.v = 'A';
END;
w1 doRecords(w1 l, s1 r) := TRANSFORM
  SELF.v := l.v + IF(r.priorA, '***', '');  
END;

initState := ROW({TRUE}, s1);
rs := PROCESS(ds,
  initState,
  doRecords(LEFT,RIGHT),
  doState(LEFT,RIGHT));
OUTPUT(rs);
/* Result:
  B***
  A
  C***
  D */
See Also: TRANSFORM Structure, RECORD Structure, ROW, ITERATE
The **PROJECT** function processes through all records in the `recordset` performing the `transform` function on each record in turn.

The **PROJECT** form is simply a shorthand synonym for:

```ecl
PROJECT(recordset,TRANSFORM(record,SELF := LEFT)).
```

making it simple to move data from one structure to another without a TRANSFORM as long as all the fields in the output `record` structure are present in the input `recordset`.  

```
recordset   The set of records to process. This may be a single-record in-line DATASET.
transform   The TRANSFORM function to call for each record in the recordset.
PREFETCH   Optional. Allows index reads within the transform to be as efficient as keyed JOINs. Valid for use only in Roxie queries.
lookahead   Optional. Specifies the number of look-ahead reads. If omitted, the default is the value of the _PrefetchProjectPreload tag in the submitted query. If that is omitted, then it is taken from the value of defaultPrefetchProjectPreload specified in the RoxieTopology file when the Roxie was deployed. If that is omitted, it defaults to 10.
PARALLEL   Optional. Specifies the lookahead is done on a separate thread, in parallel with query execution.
KEYED     Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
LOCAL     Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.
record    The output RECORD structure to use for each record in the recordset.
UNORDERED Optional. Specifies the output record order is not significant.
ORDERED   Specifies the significance of the output record order.
bool    When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE    Optional. Specifies the input record order is significant.
UNSTABLE  Optional. Specifies the input record order is not significant.
PARALLEL  Optional. Try to evaluate this activity in parallel.
numthreads Optional. Try to evaluate this activity using numthreads threads.
ALGORITHM Optional. Override the algorithm used for this activity.
name     The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
```

Return:  PROJECT returns a record set.
TRANSFORM Function Requirements - PROJECT

The `transform` function must take at least one parameter: a LEFT record of the same format as the `recordset`. Optionally, it may take a second parameter: an integer COUNTER specifying the number of times the `transform` has been called for the `recordset` or the current group in the `recordset` (see the GROUP function). The second parameter form is useful for adding sequence numbers. The format of the resulting record set does not need to be the same as the input.

Example:

```ecl
//form one example ******************************************************
Ages := RECORD
  STRING15 per_first_name;
  STRING25 per_last_name;
  INTEGER8 Age;
END;
TodaysYear := 2001;
Ages CalcAges(person l) := TRANSFORM
  SELF.Age := TodaysYear - l.birthdate[1..4];
  SELF := l;
END;
AgedRecs := PROJECT(person, CalcAges(LEFT));

//COUNTER example ******************************************************
SequencedAges := RECORD
  Ages;
  INTEGER8 Sequence := 0;
END;
SequencedAges AddSequence(Ages l, INTEGER c) :=
  TRANSFORM
  SELF.Sequence := c;
  SELF := l;
END;
SequencedAgedRecs := PROJECT(AgedRecs,
  AddSequence(LEFT,COUNTER));

//form two example ******************************************************
NewRec := RECORD
  STRING15 firstname;
  STRING25 lastname;
  STRING15 middlename;
END;
NewRecs := PROJECT(People,NewRec);
//equivalent to:
//NewRecs := PROJECT(People,TRANSFORM(NewRec,SELF :=
  LEFT));

//LOCAL example ******************************************************
MyRec := RECORD
  STRING1 Value1;
  STRING1 Value2;
END;
SomeFile := DATASET([{'C','G'},{'C','C'},{'A','X'},
  {'B','G'},{'A','B'}],MyRec);
MyOutRec := RECORD
  SomeFile.Value1;
  SomeFile.Value2;
```
STRING6 CatValues;
END;

DistFile := DISTRIBUT(SomeFile,HASH32(Value1,Value2));

MyOutRec CatThem(SomeFile L, INTEGER C) := TRANSFORM
    SELF.CatValues := L.Value1 + L.Value2 + '-' +
    (Std.System.Thorlib.Node()+1) + '-' + (STRING)C;
    SELF := L;
END;

CatRecs := PROJECT(DistFile,CatThem(LEFT,COUNTER),LOCAL);

OUTPUT(CatRecs);

/* CatRecs result set is:
Rec# Value1 Value2 CatValues
1 C C CC-1-1
2 B G BG-2-1
3 A X AX-2-2
4 A B AB-3-1
5 C G CG-3-2
*/

See Also: TRANSFORM Structure, RECORD Structure, ROW, DATASET

PROJECT - Module

PROJECT( module, interface [, OPT | attributelist ] )

<table>
<thead>
<tr>
<th>module</th>
<th>The MODULE structure containing the attribute definitions whose values to pass as the interface.</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface</td>
<td>The INTERFACE structure to pass.</td>
</tr>
<tr>
<td>OPT</td>
<td>Optional. Suppresses the error message that is generated when an attribute defined in the interface is not also defined in the module.</td>
</tr>
<tr>
<td>attributelist</td>
<td>Optional. A comma-delimited list of the specific attributes in the module to supply to the interface. This allows a specified list of attributes to be implemented, which is useful if you want closer control, or if the types of the parameters don't match.</td>
</tr>
</tbody>
</table>

Return: PROJECT returns a MODULE compatible with the interface.

The PROJECT function passes a module's attributes in the form of the interface to a function defined to accept parameters structured like the specified interface. This allows you to create a module for one interface with the values being provided by another interface. The attributes in the module must be compatible with the attributes in the interface (same type and same parameters, if any take parameters).

Example:

PROJECT(x,y)
/*is broadly equivalent to
MODULE(y)
    SomeAttributeInY := x.someAttributeInY
    //... repeated for all attributes in Y ...
END;
*/

myService(myInterface myArgs) := FUNCTION
    childArgs := MODULE(PROJECT(myArgs,Iface,isDead,did,ssn,address))
    BOOLEAN isFCRA := myArgs.isFCRA OR myArgs.fakeFCRA
END;
RETURN childService(childArgs);
END;

// you could directly pass PROJECT as a module parameter
// to an attribute:
myService(myInterface myArgs) := childService(PROJECT(myArgs, childInterface));

See Also: MODULE Structure, INTERFACE Structure, FUNCTION Structure, STORED
**PULL**

**PULL**(*dataset*)

<table>
<thead>
<tr>
<th><em>dataset</em></th>
<th>The set of records to fully load into the Data Refinery.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>PULL returns a recordset.</td>
</tr>
</tbody>
</table>

The **PULL** function is a meta-operation intended only to hint that the *dataset* should be fully loaded into the Data Refinery before continuing the operation in Data Refinery.

Example:

```ecl
MySet ::= PULL(Person);
//load Person into Data Refinery before continuing
```

See Also:
RANDOM

RANDOM()

Return:  RANDOM returns a single value.

The **RANDOM** function returns a pseudo-random non-negative integer value between 0 and 4,294,967,295.

Example:

```ecl
INTEGER1 Random1 := (RANDOM() % 25) + 1;
OUTPUT(Random1); // results are random
OUTPUT(Random1);
OUTPUT(Random1);
```

See Also: DISTRIBUTE
RANGE


<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setofdatasets</td>
<td>A set of datasets.</td>
</tr>
<tr>
<td>setofintegers</td>
<td>A set of integers.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: RANGE returns a set of datasets.

The RANGE function extracts a subset of the setofdatasets as a SET. The setofintegers specifies which elements of the setofdatasets comprise the resulting SET of datasets. This is typically used in the GRAPH function.

Example:

```
r := {STRING1 Letter};
ds1 := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'}],r);
ds2 := DATASET([{'F'},{'G'},{'H'},{'I'},{'J'}],r);
ds3 := DATASET([{'K'},{'L'},{'M'},{'N'},{'O'}],r);
ds4 := DATASET([{'P'},{'Q'},{'R'},{'S'},{'T'}],r);
ds5 := DATASET([{'U'},{'V'},{'W'},{'X'},{'Y'}],r);

SetDS := [ds1,ds2,ds3,ds4,ds5];
outDS := RANGE(setDS,[1,3]);
//use only 1st and 3rd elements

OUTPUT(outDS[1]); //results in A,B,C,D,E
OUTPUT(outDS[2]); //results in K,L,M,N,O
```

See Also: GRAPH
RANK

RANK(position, set [ ,DESCEND ])

| position | An integer indicating the element in the sorted set to return. |
| set      | The set of values. |
| DESCEND  | Optional. Indicates descending order sort. |

Return: RANK returns a single value.

The RANK function sorts the set in ascending (or descending, if DESCEND is present) order, then returns the ordinal position (its index value) of the unsorted set's position element after the set has been sorted. This is the opposite of RANKED.

Example:

```
Ranking1 := RANK(1,[20,30,10,40]);  // returns 2 - 1st element (20) in unsorted set is 2nd element after sorting to [10,20,30,40]
Ranking2 := RANK(1,[20,30,10,40],DESCEND);  // returns 3 - 1st element (20) in unsorted set is 3rd element after sorting to [40,30,20,10]
```

OUTPUT(Ranking1);
OUTPUT(Ranking2);

See Also: RANKED, SORT, SORTED, Sets and Filters
RANKED

RANKED(position, set [ ,DESCEND ])

<table>
<thead>
<tr>
<th>position</th>
<th>An integer indicating the element in the unsorted set to return.</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>The set of values.</td>
</tr>
<tr>
<td>DESCEND</td>
<td>Optional. Indicates descending order sort.</td>
</tr>
</tbody>
</table>

Return: RANKED returns a single value.

The RANKED function sorts the set in ascending (or descending, if DESCEND is present) order, then returns the ordinal position (its index value) of the sorted set's position element in the unsorted set. This is the opposite of RANK.

Example:

```
Ranking1 := RANKED(1,[20,30,10,40]);
// returns 3 - 1st element (10) in sorted set [10,20,30,40]
// was 3rd element in unsorted set

Ranking2 := RANKED(1,[20,30,10,40],DESCEND);
// returns 4 - 1st element (40) in sorted set [40,30,20,10]
// was 4th element in unsorted set
```

OUTPUT(Ranking1);
OUTPUT(Ranking2);

See Also: RANK, SORT, SORTED, Sets and Filters
REALFORMAT

REALFORMAT(expression, width, decimals)

| expression | The expression that specifies the REAL value to format. |
| width      | The size of string in which to right-justify the value.  |
| decimals   | An integer specifying the number of decimal places.     |

Return: REALFORMAT returns a single value.

The REALFORMAT function returns the value of the expression formatted as a right-justified string of width characters with the number of decimals specified.

Example:

```
REAL8 MyFloat := 1000.0063;
STRING12 FloatStr12 := REALFORMAT(MyFloat,12,6);
OUTPUT(FloatStr12); //results in ' 1000.006300'
```

See Also: INTFORMAT
REGEXFIND

REGEXFIND(regex, text [, flag ] [, NOCASE])

regex  A standard Perl regular expression.
text  The text to parse.
flag  Optional. Specifies the text to return. If omitted, REGEXFIND returns TRUE or FALSE as to whether the regex was found within the text. If 0, the portion of the text the regex was matched is returned. If >= 1, the text matched by the nth group in the regex is returned.

NOCASE  Optional. Specifies a case insensitive search.

Return:  REGEXFIND returns a single value.

The REGEXFIND function uses the regex to parse through the text and find matches. The regex must be a standard Perl regular expression. We use third-party libraries to support this, so for non-unicode text, see boost docs at http://www.boost.org/doc/libs/1_58_0/libs/regex/doc/html/index.html. Note that the version of Boost library may vary depending on your distro. For unicode text, see the ICU docs, the sections ‘Regular Expression Metacharacters’ and ‘Regular Expression Operators’ at http://userguide.icu-project.org/strings/regexp and the links from there, in particular the section ‘UnicodeSet patterns’ at http://userguide.icu-project.org/strings/unicodeset. We use version 2.6 which should support all listed features.

Example:

```ecl
namesRecord := RECORD
 STRING20 surname;
 STRING10 forename;
 STRING10 userdate;
 END;
namesTbl := DATASET([{'Halligan', 'Kevin', '10/14/1998'},
                      {'Halligan', 'Liz', '12/01/1998'},
                      {'Halligan', 'Jason', '01/01/2000'},
                      {'MacPherson', 'Jimmy', '03/14/2003'}],namesRecord);
searchpattern := '^(.*)/(.*)/(.*)$';
search := '10/14/1998';
filtered := namesTbl(REGEXFIND('^(Mc|Mac)', surname));
OUTPUT(filtered); //returns 1 record -- MacPherson, Jimmy
OUTPUT(namesTbl, { (string30) REGEXFIND (searchpattern, userdate, 0),
                   (string30) REGEXFIND (searchpattern, userdate, 1),
                   (string30) REGEXFIND (searchpattern, userdate, 2),
                   (string30) REGEXFIND (searchpattern, userdate, 3) });

REGEXFIND(searchpattern, search, 0); //returns '10/14/1998'
REGEXFIND(searchpattern, search, 1); //returns '10'
REGEXFIND(searchpattern, search, 2); //returns '14'
REGEXFIND(searchpattern, search, 3); //returns '1998'
```

See Also: PARSE, REGEXFINDSET, REGEXREPLACE
REGEXFINDSET

REGEXFINDSET(regex, text [, NOCASE])

regex  A standard Perl regular expression.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>The text to parse.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOCASE</td>
<td>Optional. Specifies a case insensitive search.</td>
</tr>
</tbody>
</table>

Return: REGEXFINDSET returns a set of strings.

The REGEXFINDSET function uses the regex to parse through the text and find matches. The regex must be a standard Perl regular expression. We use third-party libraries to support this, so for non-unicode text, see boost docs at [http://www.boost.org/doc/libs/1_58_0/libs/regex/doc/html/index.html](http://www.boost.org/doc/libs/1_58_0/libs/regex/doc/html/index.html). Note that the version of Boost library may vary depending on your distro. For unicode text, see the ICU docs, the sections 'Regular Expression Metacharacters' and 'Regular Expression Operators' at [http://userguide.icu-project.org/strings/regexp](http://userguide.icu-project.org/strings/regexp) and the links from there, in particular the section 'UnicodeSet patterns' at [http://userguide.icu-project.org/strings/unicodeset](http://userguide.icu-project.org/strings/unicodeset). We use version 2.6 which should support all listed features.

Example:

```ecl
sampleStr :=
  'To: jane@example.com From: john@example.com This is the winter of our discontent.';
eMails:=REGEXFINDSET('\\w+[a-zA-Z]+@[a-zA-Z_]+\.[a-zA-Z]{2,3}' , sampleStr);
OUTPUT(eMails);

UNICODE sampleStr2:=
  'U'To: janë@example.com From: john@example.com This is the winter of our discontent.';
eMails2:= REGEXFINDSET(U'\\w+[a-zA-Z]+@[a-zA-Z_]+\.[a-zA-Z]{2,3}' , sampleStr2);
OUTPUT(eMails2);
```

See Also: PARSE, REGEXFIND, REGEXREPLACE
REXEXREPLACE

REXEXREPLACE(regex, text, replacement [, NOCASE])

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>regex</td>
<td>A standard Perl regular expression.</td>
</tr>
<tr>
<td>text</td>
<td>The text to parse.</td>
</tr>
<tr>
<td>replacement</td>
<td>The replacement text. In this string, $0 refers to the substring that matched the regex pattern, and $1, $2, $3... match the first, second, third... groups in the pattern.</td>
</tr>
<tr>
<td>NOCASE</td>
<td>Optional. Specifies a case insensitive search.</td>
</tr>
</tbody>
</table>

**Return:**
REXEXREPLACE returns a single value.

The REXEXREPLACE function uses the regex to parse through the text and find matches, then replace them with the replacement string. The regex must be a standard Perl regular expression. We use third-party libraries to support this, so for non-unicode text, see boost docs at [http://www.boost.org/doc/libs/1_58_0/libs/regex/doc/html/index.html](http://www.boost.org/doc/libs/1_58_0/libs/regex/doc/html/index.html). Note that the version of Boost library may vary depending on your distro. For unicode text, see the ICU docs, the sections 'Regular Expression Metacharacters' and 'Regular Expression Operators' at [http://userguide.icu-project.org/strings/regexp](http://userguide.icu-project.org/strings/regexp) and the links from there, in particular the section 'UnicodeSet patterns' at [http://userguide.icu-project.org/strings/unicodeset](http://userguide.icu-project.org/strings/unicodeset). We use version 2.6 which should support all listed features.

**Example:**

```
REXEXREPLACE('(.a)t', 'the cat sat on the mat', '$1p');
//ASCII
REXEXREPLACE(u'(.a)t', u'the cat sat on the mat', u'$1p');
//UNICODE
// both of these examples return 'the cap sap on the map'
```

```
inrec := {STRING10 str, UNICODE10 ustr};
inset := DATASET({{'She', u'Eins'}, {'Sells', u'Zwei'}, {'Sea', u'Drei'}, {'Shells', u'Vier'}}, inrec);
outrec := {STRING10 orig, STRING10 withcase, STRING10 wocase, UNICODE10 uorig, UNICODE10 uwithcase, UNICODE10 uwocase};
outrec trans(inrec l) := TRANSFORM
  SELF.orig := l.str;
  SELF.withcase := REXEXREPLACE('s', l.str, 'f');
  SELF.wocase := REXEXREPLACE('s', l.str, 'f', NOCASE);
  SELF.uorig := l.ustr;
  SELF.uwithcase := REXEXREPLACE(u'e', l.ustr, u'ë');
  SELF.uwocase := REXEXREPLACE(u'e', l.ustr, u'ë', NOCASE);
END;
OUTPUT(PROJECT(inset, trans(LEFT)));
```

// the result set is:
orig withcase wocase uorig uwithcase uwocase
She She fhe Eins Eins \xc3\xb1ins
Sells Sellf fellf fellf Zwei Zw\xc3\xb1i Zw\xc3\xb1i
Sea Sea fea Drei Dr\xc3\xb1i D\xc3\xb1i
Shells Shellef thellf Vier Vi\xc3\xb1i xabi Vi\xc3\xb1i xabi */

See Also: PARSE, REGEXFIND
**REGROUP**

REGROUP(recset, ..., recset[, UNORDERED | ORDERED(bool)][, STABLE | UNSTABLE][, PARALLEL [(numthreads)]][, ALGORITHM(name)])

<table>
<thead>
<tr>
<th>recset</th>
<th>A grouped set of records. Each recset must be of exactly the same type and must contain the same number of groups.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: REGROUP returns a record set.

The **REGROUP** function combines the grouped recsets into a single grouped record set. This is accomplished by combining each group in the first recset with the groups in the same ordinal position within each subsequent recset.

Example:

```ecl
inrec := {UNSIGNED6 did};
outrec := RECORD(inrec
  STRING20 name;
  UNSIGNED score;
END;

ds := DATASET([1,2,3,4,5,6], inrec);
dsg := GROUP(ds, ROW);

i1 := DATASET([{1, 'Kevin', 10},
            {2, 'Richard', 5},
            {5, 'Nigel', 2},
            {0, '', 0}], outrec);
i2 := DATASET([{1, 'Kevin Halligan', 12},
            {2, 'Ricardo Chapman', 15},
            {3, 'Jake Smith', 20},
            {5, 'David Hicks', 100},
            {0, '', 0}], outrec);
i3 := DATASET([{1, 'Halligan', 8},
            {2, 'Ricardo', 8},
            {6, 'Pete', 4},
            {6, 'Peter', 8},
            {6, 'Petie', 1},
            {0, '', 0}], outrec);

j1 := JOIN(dsg, i1, LEFT.did = RIGHT.did, LEFT OUTER, MANY LOOKUP);
j2 := JOIN(dsg, i2, LEFT.did = RIGHT.did, LEFT OUTER, MANY LOOKUP);
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
j3 := JOIN(dsg, i3, LEFT.did = RIGHT.did, LEFT OUTER, MANY LOOKUP);

combined := REGROUP(j1, j2, j3);
OUTPUT(j1);
OUTPUT(j2);
OUTPUT(j3);
OUTPUT(combined);

See Also: GROUP, COMBINE
### REJECTED

**REJECTED**(condition,...,condition)

- **condition**: A conditional expression to evaluate.
- **Return**: REJECTED returns a single value.

The REJECTED function evaluates which of the list of conditions returned false and returns its ordinal position in the list of conditions. Zero (0) returns if none return false. This is the opposite of the WHICH function.

Example:

```ecl
Rejects := REJECTED(Person.first_name <> 'Fred', Person.first_name <> 'Sue');
// Rejects receives 0 for everyone except those named Fred or Sue
```

See Also: WHICH, MAP, CHOOSE, IF, CASE
**ROLLUP**


- **recordset**
  > The set of records to process, typically sorted in the same order that the condition or fieldlist will test.

- **condition**
  > An expression that defines "duplicate" records. The keywords LEFT and RIGHT may be used as dataset qualifiers for fields in the recordset.

- **transform**
  > The TRANSFORM function to call for each pair of duplicate records found.

- **LOCAL**
  > Optional. Specifies the operation is performed on each node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.

- **fieldlist**
  > A comma-delimited list of expressions or fields in the recordset that defines "duplicate" records. You may use the keywords WHOLE RECORD (or just RECORD) to indicate all fields in that structure, and/or you may use the keyword EXCEPT to list fields to exclude.

- **GROUP**
  > Specifies the recordset is GROUPed and the ROLLUP operation will produce a single output record for each group. If this is not the case, an error occurs.

- **UNORDERED**
  > Optional. Specifies the output record order is not significant.

- **ORDERED**
  > Specifies the significance of the output record order.

- **bool**
  > When False, specifies the output record order is not significant. When True, specifies the default output record order.

- **STABLE**
  > Optional. Specifies the input record order is significant.

- **UNSTABLE**
  > Optional. Specifies the input record order is not significant.

- **PARALLEL**
  > Optional. Try to evaluate this activity in parallel.

- **numthreads**
  > Optional. Try to evaluate this activity using numthreads threads.

- **ALGORITHM**
  > Optional. Override the algorithm used for this activity.

- **name**
  > The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

Return: ROLLUP returns a record set.

The **ROLLUP** function is similar to the DEDUP function with the addition of a call to the **transform** function to process each duplicate record pair. This allows you to retrieve valuable information from the "duplicate" record before it's thrown away. Depending on how you code the **transform** function, ROLLUP can keep the LEFT or RIGHT record, or any mixture of data from both.

The first form of ROLLUP tests a condition using values from the records that would be passed as LEFT and RIGHT to the **transform**. The records are combined if the condition is true. The second form of ROLLUP compares values from adjacent records in the input **recordset**, and combines them if they are the same. These two forms will behave differently if the **transform** modifies some of the fields used in the matching condition (see example below).
For the first pair of candidate records, the LEFT record passed to the transform is the first record of the pair, and the RIGHT record is the second. For subsequent matches of the same values, the LEFT record passed is the result record from the previous call to the transform and the RIGHT record is the next record in the recordset, as in this example:

```
ds := DATASET([[(1,10),(1,20),(1,30),(3,40),(4,50)],[UNSIGNED r, UNSIGNED n]]);
d t(ds L, ds R) := TRANSFORM
  SELF.r := L.r + R.r;
  SELF.n := L.n + R.n;
END;
ROLLUP(ds, t(LEFT, RIGHT), r);
/* results in:
   3 60
   3 40
   4 50
*/
ROLLUP(ds, LEFT.r = RIGHT.r, t(LEFT, RIGHT));
/* results in:
   2 30
   1 30
   3 40
   4 50
   the third record is not combined because the transform modified the value.
*/
```

**TRANSFORM Function Requirements - ROLLUP**

For forms 1 and 2 of ROLLUP, the transform function must take at least two parameters: a LEFT record and a RIGHT record, which must both be in the same format as the recordset. The format of the resulting record set must also be the same as the inputs.

For form 3 of ROLLUP, the transform function must take at least two parameters: a LEFT record which must be in the same format as the recordset, and a ROWS(LEFT) whose format must be a DATASET(RECORD-OF(recordset)) parameter. The format of the resulting record set may be different from the inputs.

**ROLLUP Form 1**

Form 1 processes through all records in the recordset performing the transform function only on those pairs of adjacent records where the match condition is met (indicating duplicate records) and passing through all other records directly to the output.

Example:

```
//a crosstab table of last names and the number of times they occur
MyRec := RECORD
  Person.per_last_name;
  INTEGER4 PersonCount := 1;
END;
LnameTable := TABLE(Person,MyRec); //create dataset to work with
SortedTable := SORT(LnameTable,per_last_name); //sort it first
MyRec Xform(MyRec L,MyRec R) := TRANSFORM
  SELF.PersonCount := L.PersonCount + 1;
  SELF := L; //keeping the L rec makes it KEEP(1),LEFT
  // SELF := R; //keeping the R rec would make it KEEP(1),RIGHT
END;
XtabOut := ROLLUP(SortedTable,
  LEFT.per_last_name=RIGHT.per_last_name,
  Xform(LEFT,RIGHT));
```
ROLLUP Form 2

Form 2 processes through all records in the recordset performing the transform function only on those pairs of adjacent records where all the expressions in the fieldlist match (indicating duplicate records) and passing through all other records to the output. This form allows you to use the same kind of EXCEPT field exclusion logic available to DEDUP.

Example:

```ecl
crc := {STRING1 str1, STRING1 str2, STRING1 str3};
d := DATASET([{'a', 'b', 'c'}, {'a', 'b', 'c'},
                {'a', 'c', 'c'}, {'a', 'c', 'd'}], rec);
rec tr(rec L, rec R) := TRANSFORM
  SELF := L;
  Cat(STRING1 L, STRING1 R) := L + R;
end;
```

```ecl
r1 := ROLLUP(ds, tr(LEFT, RIGHT), str1, str2);
// equivalent to LEFT.str1 = RIGHT.str1 AND
// LEFT.str2 = RIGHT.str2
r2 := ROLLUP(ds, tr(LEFT, RIGHT), WHOLE RECORD, EXCEPT str3);
// equivalent to LEFT.str1 = RIGHT.str1 AND
// LEFT.str2 = RIGHT.str2
r3 := ROLLUP(ds, tr(LEFT, RIGHT), RECORD, EXCEPT str3);
// equivalent to LEFT.str1 = RIGHT.str1 AND
// LEFT.str2 = RIGHT.str2
r4 := ROLLUP(ds, tr(LEFT, RIGHT), RECORD, EXCEPT str2, str3);
// equivalent to LEFT.str1 = RIGHT.str1 AND
r5 := ROLLUP(ds, tr(LEFT, RIGHT), RECORD);
// equivalent to LEFT.str1 = RIGHT.str1 AND
// LEFT.str2 = RIGHT.str2 AND
// LEFT.str3 = RIGHT.str3
r6 := ROLLUP(ds, tr(LEFT, RIGHT), str1 + str2);
// equivalent to LEFT.str1 + LEFT.str2 = RIGHT.str1 + RIGHT.str2
r7 := ROLLUP(ds, tr(LEFT, RIGHT), Cat(str1, str2));
// equivalent to Cat(LEFT.str1, LEFT.str2) =
// Cat(RIGHT.str1, RIGHT.str2)
```

ROLLUP Form 3

Form 3 is a special form of ROLLUP where the second parameter passed to the transform is a GROUP and the first parameter is the first record in that GROUP. It processes through all groups in the recordset, producing one result record for each group. Aggregate functions can be used inside the transform (such as TOPN or CHOOSEN) on the second parameter. The result record set is not grouped. This form is implicitly LOCAL in nature, due to the grouping.

Example:

```ecl
inrec := RECORD
  UNSIGNED6 did;
END;
```

```ecl
outrec := RECORD(inrec)
  STRING20 name;
  UNSIGNED score;
END;
```
nameRec := RECORD
  STRING20 name;
END;

finalRec := RECORD(inrec)
  DATASET(nameRec) names;
  STRING20 secondName;
END;

d := DATASET([1,2,3,4,5,6], inrec);
dg := GROUP(d, ROW);

i1 := DATASET([1, 'Kevin', 10],
  [2, 'Richard', 5],
  [5, 'Nigel', 2],
  [0, '', 0]), outrec);

i2 := DATASET([1, 'Kevin Halligan', 12],
  [2, 'Richard Charles', 15],
  [3, 'Blake Smith', 20],
  [5, 'Nigel Hicks', 100],
  [0, '', 0]), outrec);

i3 := DATASET([1, 'Halligan', 8],
  [2, 'Richard', 8],
  [6, 'Pete', 4],
  [6, 'Peter', 8],
  [6, 'Petie', 1],
  [0, '', 0]), outrec);

j1 := JOIN( dg,
  [1, 'Kevin', 10],
  [2, 'Richard', 5],
  [5, 'Nigel', 2],
  [0, '', 0]), outrec);

j2 := JOIN( dg,
  [1, 'Kevin Halligan', 12],
  [2, 'Richard Charles', 15],
  [3, 'Blake Smith', 20],
  [5, 'Nigel Hicks', 100],
  [0, '', 0]), outrec);

j3 := JOIN( dg,
  [1, 'Halligan', 8],
  [2, 'Richard', 8],
  [6, 'Pete', 4],
  [6, 'Peter', 8],
  [6, 'Petie', 1],
  [0, '', 0]), outrec);

combined := REGROUP(j1, j2, j3);

finalRec doRollup(outRec l, DATASET(outRec) allRows) :=
  TRANSFORM
  SELF.did := l.did;
  SELF.names := PROJECT(allRows(score != 0),
    TRANSFORM(nameRec, SELF := LEFT));
  SELF.secondName := allRows(score != 0)[2].name;
END;

results := ROLLUP(combined, GROUP, doRollup(LEFT, ROWS(LEFT)));
OUTPUT(results);

See Also: TRANSFORM Structure, RECORD Structure, DEDUP, EXCEPT, GROUP
ROUND

ROUND(realvalue[, decimals ])

<table>
<thead>
<tr>
<th>realvalue</th>
<th>The floating-point value to round.</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimals</td>
<td>Optional. An integer specifying the number of decimal places to round to. If omitted, the default is zero (integer result).</td>
</tr>
</tbody>
</table>

Return: ROUND returns a single numeric value.

The ROUND function returns the rounded realvalue by using standard arithmetic rounding (decimal portions less than .5 round down and decimal portions greater than or equal to .5 round up).

Example:

```plaintext
SomeRealValue1 := 3.14159;
INTEGER4 MyVal1 := ROUND(SomeRealValue1);  // MyVal1 is 3
REAL MyVal2 := ROUND(SomeRealValue1,2);    // MyVal2 is 3.14

SomeRealValue2 := 3.5;
INTEGER4 MyVal3 := ROUND(SomeRealValue2);  // MyVal is 4

SomeRealValue3 := -1.3;
INTEGER4 MyVal4 := ROUND(SomeRealValue3);  // MyVal is -1

SomeRealValue4 := -1.8;
INTEGER4 MyVal5 := ROUND(SomeRealValue4);  // MyVal is -2
OUTPUT(MyVal1);
OUTPUT(MyVal2);
OUTPUT(MyVal3);
OUTPUT(MyVal4);
OUTPUT(MyVal5);
```

See Also: ROUNDUP, TRUNCATE
ROUNDUP

ROUNDUP(realvalue)

<table>
<thead>
<tr>
<th>realvalue</th>
<th>The floating-point value to round.</th>
</tr>
</thead>
</table>

Return: ROUNDUP returns a single integer value.

The **ROUNDUP** function returns the rounded integer of the *realvalue* by rounding any decimal portion to the next larger integer value, regardless of sign.

Example:

```plaintext
SomeRealValue := 3.14159;
INTEGER4 MyVal1 := ROUNDUP(SomeRealValue); // MyVal is 4

SomeRealValue2 := -3.9;
INTEGER4 MyVal2 := ROUNDUP(SomeRealValue2); // MyVal is -4

OUTPUT(MyVal1);
OUTPUT(MyVal2);
```

See Also: ROUND, TRUNCATE
ROW

ROW( { fields }, recstruct )
ROW( row, resultrec )
ROW( [ row ], transform )

<table>
<thead>
<tr>
<th><strong>fields</strong></th>
<th>A comma-delimited list of data values for each field in the recstruct, contained in curly braces ( {} ).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>recstruct</strong></td>
<td>The name of the RECORD structure defining the field layout.</td>
</tr>
<tr>
<td><strong>row</strong></td>
<td>A single row of data. This may be an existing record, or formatted in-line data values like the fields parameter description above, or an empty set ([ ] ) to add a cleared record in the format of the resultrec. If omitted, the record is produced by the transform function.</td>
</tr>
<tr>
<td><strong>resultrec</strong></td>
<td>A RECORD structure that defines how to construct the row of data, similar to the type used by TABLE.</td>
</tr>
<tr>
<td><strong>transform</strong></td>
<td>A TRANSFORM function that defines how to construct the row of data.</td>
</tr>
<tr>
<td><strong>Return:</strong></td>
<td>ROW returns a single record.</td>
</tr>
</tbody>
</table>

The ROW function creates a single data record and is valid for use in any expression where a single record is valid.

**ROW Form 1**

The first form constructs a record from the in-line data in the fields, structured as defined by the recstruct. This is typically used within a TRANSFORM structure as the expression defining the output for a child dataset field.

Example:

```ecl
AkaRec := {STRING20 forename,STRING20 surname};
outputRec := RECORD
  UNSIGNED id;
  DATASET(AkaRec) kids;
END;
inputRec := {UNSIGNED id,STRING20 forename,STRING20 surname};
inPeople := DATASET([ {1,'Kevin','Halligan'}, {1,'Kevin','Hall'},{2,'Eliza','Hall'},{2,'Beth','Took'} ],inputRec);
outputRec makeFatRecord(inputRec L) := TRANSFORM
  SELF.id := l.id;
  SELF.kids := DATASET([ { L.forename, L.surname } ],AkaRec);
END;
fatIn := PROJECT(inPeople, makeFatRecord(LEFT));
outputRec makeChildren(outputRec L, outputRec R) := TRANSFORM
  SELF.id := L.id;
  SELF.kids := L.kids + ROW({ R.kids[1].forename, R.kids[1].surname },AkaRec);
END;
r := ROLLUP(fatIn, id, makeChildren(LEFT, RIGHT));
OUTPUT(r);
```

**ROW Form 2**

The second form constructs a record from the row passed to it using the resultrec the same way the TABLE function operates. This is typically used within a TRANSFORM structure as the expression defining the output for a child dataset field.
Example:

```ecl
AkaRec := {STRING20 forename, STRING20 surname};
outputRec := RECORD
UNSIGNED id;
DATASET(AkaRec) children;
END;
inputRec := {UNSIGNED id, STRING20 forename, STRING20 surname};
inPeople := DATASET(
[{1, 'Kevin', 'Halligan'}, {1, 'Kevin', 'Hall'},
 {1, 'Gawain', ''}, {2, 'Liz', 'Hall'},
 {2, 'Eliza', 'Hall'}, {2, 'Beth', 'Took'}], inputRec);
outputRec makeFatRecord(inputRec L) := TRANSFORM
  SELF.id := l.id;
  SELF.children := ROW(L, AkaRec); // using Form 2 here
END;
fatIn := PROJECT(inPeople, makeFatRecord(LEFT));
outputRec makeChildren(outputRec L, outputRec R) := TRANSFORM
  SELF.id := L.id;
  SELF.children := L.children +
    ROW({R.children[1].forename, R.children[1].surname}, AkaRec);
END;
r := ROLLUP(fatIn, id, makeChildren(LEFT, RIGHT));
OUTPUT(r);
```

**ROW Form 3**

The third form uses a TRANSFORM function to produce its single record result. The `transform` function must take at least one parameter: a LEFT record, which must be in the same format as the input record. The format of the resulting record may be different from the input.

Example:

```ecl
IMPORT Std;
NameRec := RECORD
  STRING5 title;
  STRING20 fname;
  STRING20 mname;
  STRING20 lname;
  STRING5 name_suffix;
  STRING3 name_score;
END;
MyRecord := RECORD
  UNSIGNED id;
  STRING uncleanedName;
  NameRec Name;
END;
x := DATASET('RTTEST::RowFunctionData', MyRecord, THOR);
STRING73 CleanPerson73(STRING inputName) := FUNCTION
  suffix:=[ ' 0', ' 1', ' 2', ' 3', ' 4', ' 5', ' 6', ' 7', ' 8', ' 9',
    ' J', ' JR', ' S', ' SR'];
  InWords := Std.Str.CleanSpaces(inputName);
  HasSuffix := InWords[LENGTH(TRIM(InWords))-1 ..] IN suffix;
  WordCount := LENGTH(TRIM(InWords, LEFT, RIGHT)) - LENGTH(TRIM(InWords, ALL)) + 1;
  HasMiddle := WordCount = 5 OR (WordCount = 4 AND NOT HasSuffix);
  Space1 := Std.Str.Find(InWords, ' ', 1);
  Space2 := Std.Str.Find(InWords, ' ', 2);
  Space3 := Std.Str.Find(InWords, ' ', 3);
  Space4 := Std.Str.Find(InWords, ' ', 4);
  STRING5 title := InWords[1 .. Space1 - 1];
```
STRING20 fname := InWords[Space1+1..Space2-1];
STRING20 mname := IF(HasMiddle,InWords[Space2+1..Space3-1],'');
STRING20 lname := MAP(HasMiddle AND NOT HasSuffix =>
    InWords[Space3+1..],
    HasMiddle AND HasSuffix =>
    InWords[Space3+1..Space4-1],
    NOT HasMiddle AND NOT HasSuffix =>
    InWords[Space2+1..],
    NOT HasMiddle AND HasSuffix =>
    InWords[Space2+1..Space3-1],
    '');
STRING5 name_suffix := IF(HasSuffix,InWords[LENGTH(TRIM(InWords))-1 ..'],'');
STRING3 name_score := '';
RETURN title + fname + mname + lname + name_suffix + name_score;
END;

//Example 1 - a transform to create a row from an uncleaned name
NameRec createRow(string inputName) := TRANSFORM
    cleanedText := CleanPerson73(inputName);
    SELF.title := cleanedText[1..5];
    SELF.fname := cleanedText[6..25];
    SELF.mname := cleanedText[26..45];
    SELF.lname := cleanedText[46..65];
    SELF.name_suffix := cleanedText[66..70];
    SELF.name_score := cleanedText[71..73];
END;
myRecord t(myRecord L) := TRANSFORM
    SELF.Name := ROW(createRow(L.uncleanedName));
    SELF := L;
END;
y := PROJECT(x, t(LEFT));
OUTPUT(y);

//Example 2 - an attribute using that transform to generate the row.
NameRec cleanedName(STRING inputName) := ROW(createRow(inputName));
myRecord t2(myRecord L) := TRANSFORM
    SELF.Name := cleanedName(L.uncleanedName);
    SELF := L;
END;
y2 := PROJECT(x, t2(LEFT));
OUTPUT(y2);

//Example 3 - Encapsulate the transform inside the attribute by
// defining a FUNCTION structure.
NameRec cleanedName2(STRING inputName) := FUNCTION
    createRow := TRANSFORM
        cleanedText := CleanPerson73(inputName);
        SELF.title := cleanedText[1..5];
        SELF.fname := cleanedText[6..25];
        SELF.mname := cleanedText[26..45];
        SELF.lname := cleanedText[46..65];
        SELF.name_suffix := cleanedText[66..70];
        SELF.name_score := cleanedText[71..73];
    END;
    RETURN ROW(createRow);
END;
myRecord t3(myRecord L) := TRANSFORM
    SELF.Name := cleanedName2(L.uncleanedName);
    SELF := L;
END;
y3 := PROJECT(x, t3(LEFT));
OUTPUT(y3);

See Also: TRANSFORM Structure, DATASET, RECORD Structure, FUNCTION Structure
**ROWDIFF**

**ROWDIFF**(left, right [, COUNT])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>The left record, or a nested record structure.</td>
</tr>
<tr>
<td>right</td>
<td>The right record, or a nested record structure.</td>
</tr>
<tr>
<td>COUNT</td>
<td>Optional. Specifies returning a comma delimited set of zeros and ones (0,1) indicating which fields matched (0) and which did not (1). If omitted, a comma delimited set of the non-matching field names.</td>
</tr>
</tbody>
</table>

Return: ROWDIFF returns a single value.

The **ROWDIFF** function is valid for use only within a TRANSFORM structure for a JOIN operation and is used as the expression defining the output for a string field. Fields are matched by name and only like-named fields are included in the output.

Example:

```ecl
FullName := RECORD
  STRING30 forename;
  STRING20 surname;
  IFBLOCK(SELF.surname <> 'Windsor')
    STRING20 middle;
  END;
END;

in1rec := {UNSIGNED1 id,FullName name,UNSIGNED1 age,STRING5 title};
in2rec := {UNSIGNED1 id,FullName name,REAL4 age,BOOLEAN dead};
in1 := DATASET([{'Kevin','Halligan','','',33,'Mr'},
               {'Liz','Halligan','','',33,'Dr'},
               {'Elizabeth','Windsor',99,'Queen'}], in1rec);
in2 := DATASET([{'Kevin','Halligan','','',33,false},
               {'Liz','','Jean',33,false},
               {'Elizabeth','Windsor',99.1,false}], in2rec);
outrec := RECORD
  UNSIGNED1 id;
  STRING35 diff1;
  STRING35 diff2;
  STRING35 diff3;
  STRING35 diff4;
END;
outrec t1(in1 L, in2 R) := TRANSFORM
  SELF.id := L.id;
  SELF.diff1 := ROWDIFF(L,R);
  SELF.diff2 := ROWDIFF(L.name, R.name);
  SELF.diff3 := ROWDIFF(L, R, COUNT);
  SELF.diff4 := ROWDIFF(L.name, R.name, COUNT);
END;
OUTPUT(JOIN(in1, in2, LEFT.id = RIGHT.id, t1(LEFT,RIGHT)));
```

The result set from this code is:

```
///id diff1 diff2 diff3 diff4
//1 0,0,0,0,0 0,0,0
//2 name.surname, name.middle surname, middle 0,0,1,0,0 0,1,1
//3 age 0,0,0,0,0 0,0,0
```

See Also: TRANSFORM Structure, JOIN
SAMPLE

SAMPLE(recordset, interval [, which ] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [( numthreads ) ]] [, ALGORITHM( name ) ] )

<table>
<thead>
<tr>
<th>recordset</th>
<th>The set of records to sample. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval</td>
<td>The interval between records to return.</td>
</tr>
<tr>
<td>which</td>
<td>Optional. An integer specifying the ordinal number of the sample set to return. This is used to obtain multiple non-overlapping samples from the same recordset.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function’s STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: SAMPLE returns a set of records.

The SAMPLE function returns a sample set of records from the nominated recordset.

Example:

```ecl
define personRecord := RECORD
  STRING UID;
  STRING first_name;
  STRING last_name;
  STRING address;
  STRING city;
  STRING state;
  STRING zip;
END;
define person := DATASET(
  ['923','James','Jones','123 Elm Street','Hollywood','FL','33022'],
  ['924','Sally','Jones','22 Main Street','Tampa','FL','33604'],
  ['925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101'],
  ['926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108'],
  ['927','Evelyn','Murray','740 SW 10th Street','Boston ','MA','02116'],
  ['928','Tom','Murray','740 SW 10th Street','Boston ','MA','02116'],
  ['929','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131']
), personRecord);
define MySample1 := SAMPLE(Person,3,1); // returns every 3rd record

define SomeFile := DATASET(
  ['A','B','C','D','E'],
  ['F','G','H','I','J'],
  ['K','L','M','N','O'],
  ['P','Q','R','S','T'],
  ['U','V','W','X','Y'],
  [STRING1 Letter]
);
define MySample2 := SAMPLE(SomeFile,5,1); // returns A, F, K, P, U
```
OUTPUT(MySample1);
OUTPUT(MySample2);

See Also: CHOSEN, ENTH
SEQUENTIAL

\[ definitionname := ] \textbf{SEQUENTIAL( actionlist )} \]

<table>
<thead>
<tr>
<th><strong>definitionname</strong></th>
<th>Optional. The action name, which turns the action into a definition, therefore not executed until the definitionname is used as an action.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>actionlist</strong></td>
<td>A comma-delimited list of the actions to execute in order. These may be ECL actions or external actions.</td>
</tr>
</tbody>
</table>

The \textbf{SEQUENTIAL} action executes the items in the \textit{actionlist} in the order in which they appear in the \textit{actionlist}.

Example:

```
Act1 := OUTPUT(A_People,OutputFormat1,'//hold01/fred.out');
Act2 := OUTPUT(Person,{Person.per_first_name,Person.per_last_name});
Act3 := OUTPUT(Person,{Person.per_last_name});
//by naming these actions, they become inactive definitions
//that only execute when the definition names are called as definitions
SEQUENTIAL(Act1,PARALLEL(Act2,Act3));
//executes Act1 alone, and only when it's finished,
//executes Act2 and Act3 together
```

See Also: ORDERED, PARALLEL, PERSIST
SET(recordset, expression[, UNORDERED | ORDERED(bool)][, STABLE | UNSTABLE][, PARALLEL([numthreads])][, ALGORITHM(name)])

- **recordset**: The set of records from which to derive the SET of values.
- **expression**: The expression from which to obtain the values, typically just a field in the recordset.
- **UNORDERED**: Optional. Specifies the output record order is not significant.
- **ORDERED**: Specifies the significance of the output record order.
- **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.
- **STABLE**: Optional. Specifies the input record order is significant.
- **UNSTABLE**: Optional. Specifies the input record order is not significant.
- **PARALLEL**: Optional. Try to evaluate this activity in parallel.
- **numthreads**: Optional. Try to evaluate this activity using numthreads threads.
- **ALGORITHM**: Optional. Override the algorithm used for this activity.
- **name**: The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

Return: SET returns a SET of values of the same type as the field.

The SET function returns a SET for use in any set operation (such as the IN operator), similar to a sub-select in SQL when used with the IN operator. It does not remove duplicate elements and does not order the set.

One common problem is the use of the SET function in a filter condition, like this:

```
MyDS := myDataset(myField IN SET(anotherDataset, someField));
```

The code generated for this is inefficient if "anotherDataset" contains a large number of elements, and may also cause a "Dataset too large to output to workunit" error. A better way to recode the expression would be this:

```
MyDS := JOIN(myDataset, anotherDataset, LEFT.myField = RIGHT.someField, TRANSFORM(LEFT), LOOKUP);
```

The end result is the same, the set of "myDataset" records where the "myField" value is one of the "someField" values from "anotherDataset," but the code is much more efficient in execution.

You can construct a DATASET from a SET.

```
ds := DATASET([[{'X',1},{'B',3},{'C',2},{'B',5}],
{{'C',4},{'D',6},{'E',2}]],
{STRING1 Ltr, INTEGER1 Val});
s1 := SET(ds,Ltr); //a SET of just the Ltr field values:
DATASET(s1,{STRING1 Ltr}); //a DATASET from the SET
```

Example:

```
ds := DATASET([[{'X',1},{'B',3},{'C',2},{'B',5}],
{{'C',4},{'D',6},{'E',2}]],
{STRING1 Ltr, INTEGER1 Val});
//a SET of just the Ltr field values:
s1 := SET(ds,Ltr);
COUNT(s1); //results in 7
```
OUTPUT(s1); //results in ['X', 'B', 'C', 'B', 'C', 'D', 'E']

//a simple way to get just the unique elements
//is to use a crosstab TABLE:
t := TABLE(ds, {Ltr}, Ltr); //order indeterminant

s2 := SET(t, Ltr);
COUNT(s2); //results in 5
OUTPUT(s2); //results in ['D', 'X', 'C', 'E', 'B']

//sorted unique elements
s3 := SET(SORT(t, Ltr), Ltr);
COUNT(s3); //results in 5
OUTPUT(s3); //results in ['B', 'C', 'D', 'E', 'X']

//a SET of expression values:
s4 := SET(ds, Ltr+Val);
OUTPUT(s4); // results in ['X1', 'B3', 'C2', 'B5', 'C4', 'D6', 'E2']

See Also: Sets and Filters, SET OF, Set Operators, IN Operator
**SIN**

**SIN(angle)**

<table>
<thead>
<tr>
<th>angle</th>
<th>The REAL radian value for which to find the sine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>SIN returns a single REAL value.</td>
</tr>
</tbody>
</table>

The **SIN** function returns the sine of the *angle*.

Example:

```
Rad2Deg := 57.295779513082; //number of degrees in a radian
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad;    //translate 45 degrees into radians
Sine45  := SIN(Angle45);    //get sine of the 45 degree angle
OUTPUT(Sine45);
```

See Also: ACOS, COS, ASIN, TAN, ATAN, COSH, SINH, TANH
**SINH**

**SINH**(angle)

<table>
<thead>
<tr>
<th>angle</th>
<th>The REAL radian value for which to find the hyperbolic sine.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>SINH returns a single REAL value.</td>
</tr>
</tbody>
</table>

The **SINH** function returns the hyperbolic sine of the *angle*.

**Example:**

```plaintext
Rad2Deg := 57.295779513082; //number of degrees in a radian
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad; //translate 45 degrees into radians
HyperbolicSine45 := SINH(Angle45); //get hyperbolic sine of the angle
OUTPUT(HyperbolicSine45);
```

See Also: ACOS, COS, ASIN, TAN, ATAN, COSH, SIN, TANH
SIZEOF

SIZEOF(data [, MAX ])

<table>
<thead>
<tr>
<th>data</th>
<th>The name of a dataset, RECORD structure, a fully-qualified field name, or a constant string expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX</td>
<td>Specifies the data is variable-length (such as containing child datasets) and the value to return is the maximum size.</td>
</tr>
</tbody>
</table>

Return: SIZEOF returns a single integer value.

The `SIZEOF` function returns the total number of bytes defined for storage of the specified `data` structure or field.

Example:

```ecl
MyRec := RECORD
  INTEGER1 F1;
  INTEGER5 F2;
  STRING1 F3;
  STRING10 F4;
  QSTRING12 F5;
  VARSTRING12 F6;
END;
MyData := DATASET(
  [{1,33333333333,'A','A','A','V'A'}], MyRec);
size0 := SIZEOF(MyRec); //result is 39
size1 := SIZEOF(MyData.F1); //result is 1
size2 := SIZEOF(MyData.F2); //result is 5
size3 := SIZEOF(MyData.F3); //result is 1
size4 := SIZEOF(MyData.F4); //result is 10
size5 := SIZEOF(MyData.F5); //result is 9 -12 chars stored in 9 bytes
size6 := SIZEOF(MyData.F6); //result is 13 -12 chars plus null terminator
size7 := SIZEOF('abc' + '123'); //result is 6
OUTPUT(size0);
OUTPUT(size1);
OUTPUT(size2);
OUTPUT(size3);
OUTPUT(size4);
OUTPUT(size5);
OUTPUT(size6);
OUTPUT(size7);
```

See Also: LENGTH
SOAPCALL

result := SOAPCALL([ recset, ] url, service, instructure, [ transform, ] DATASET(outstructure) | outstructure [, options ] [ , UNORDERED | ORDERED( bool ) ] [ , STABLE | UNSTABLE ] [ , PARALLEL [ ( numthreads ) ] ] [ , ALGORITHM( name ) ] ];

SOAPCALL([ recset, ] url, service, instructure, [ transform, ] [ options ] [ , UNORDERED | ORDERED( bool ) ] [ , STABLE | UNSTABLE ] [ , PARALLEL [ ( numthreads ) ] ] [ , ALGORITHM( name ) ] );

result The attribute name for the resulting recordset or single record.

recset Optional. The input recordset. If omitted, the single input record must be defined by default values for each field in the instructure parameter.

url A string containing a pipe-delimited ( | ) list of URLs that host the service to invoke (may append repository module names). This is intended to provide a means for the client to conduct a Federated search where the request is sent to each of the target systems in the list. These URLs may contain standard form usernames and passwords, if required. If calling an ESP Web service, you can append the ver_=n.nn parameter to specify the version of the service. For example:

SOAPCALL('https://eclwatch.example.com:8010/Wsdfu/?ver_=1.22', 'DFUSearchData', instructure,DATASET(outstructure));

service A string expression containing the name of the service to invoke. This may be in the form module.attribute if the service is on a Roxie platform.

instructure A RECORD structure containing the input field definitions from which the XML input to the SOAP service is constructed. The name of the tags in the XML are derived from the lowercase names of the fields in the input record; this can be overridden by placing an xpath on the field ( {xpath('tagname')} -- see the XPATH Support section of the RECORD Structure discussion). If the recset parameter is not present, each field definition must contain a default value that will constitute the single input record. If the recset parameter is present, each field definition must contain a default value unless a transform is also specified to supply that data values.

transform Optional. The TRANSFORM function to call to process the instructure data. This eliminates the need to define default values for all fields in the instructure RECORD structure. The transform function must take at least one parameter: a LEFT record of the same format as the input recset. The resulting record set format must be the same as the input instructure.

DATASET (outstructure) Specifies recordset result in the outstructure format.

outstructure A RECORD structure containing the output field definitions. If not used as a parameter to the DATASET keyword, this specifies a single record result. Each field definition in the RECORD structure must use an xpath attribute ( {xpath('tagname')}) to eliminate case sensitivity issues.

options A comma-delimited list of optional specifications from the list below.

UNORDERED Optional. Specifies the output record order is not significant.

ORDERED Specifies the significance of the output record order.

bool When False, specifies the output record order is not significant. When True, specifies the default output record order.
### STABLE
Optional. Specifies the input record order is significant.

### UNSTABLE
Optional. Specifies the input record order is not significant.

### PARALLEL
Optional. Try to evaluate this activity in parallel.

### numthreads
Optional. Try to evaluate this activity using `numthreads` threads per node.

### ALGORITHM
Optional. Override the algorithm used for this activity.

### name
The algorithm to use for this activity. Must be from the list of supported algorithms for the `SORT` function's STABLE and UNSTABLE options.

### Return:
SOAPCALL returns either a set of records, a single record, or nothing.

**SOAPCALL** is a function or action that calls a SOAP (Simple Object Access Protocol) service.

Valid options are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RETRY</strong></td>
<td>Specifies re-attempting the call count number of times if non-fatal errors occur. If omitted, the default is three (3).</td>
</tr>
<tr>
<td><strong>TIMEOUT</strong></td>
<td>Specifies the amount of time to attempt the read before failing. The <em>period</em> is a real number where the integer portion specifies seconds. Setting to zero (0) indicates waiting forever. If omitted, the default is three hundred (300).</td>
</tr>
<tr>
<td><strong>TIMELIMIT</strong></td>
<td>Specifies the total amount of time allowed for the SOAPCALL. The <em>period</em> is a real number where the integer portion specifies seconds. If omitted, the default is zero (0) indicating no limit.</td>
</tr>
<tr>
<td><strong>HEADING</strong></td>
<td>Specifies tags to wrap around the XML input fields. If omitted, the default is: <code>HEADING(&quot;&quot;,&quot;</code>).</td>
</tr>
<tr>
<td><strong>XPATH</strong></td>
<td>Specifies the path used to access rows in the output. If omitted, the default is: ‘serviceResponse/Results/Result/Dataset/Row’.</td>
</tr>
<tr>
<td><strong>MERGE</strong></td>
<td>Specifies processing <code>n</code> records per batch (the blocking). If omitted, the default is 1 (values other than 1 may be incompatible with non-Roxie services). Valid for use only if the <code>recset</code> parameter is also present.</td>
</tr>
<tr>
<td><strong>PARALLEL</strong></td>
<td>Specifies the number of concurrent threads per node to process queries, to a maximum of 50 (the default is 2). This is intended to limit the number of concurrent sessions.</td>
</tr>
<tr>
<td><strong>ONFAIL</strong></td>
<td>Specifies either the transform function to call if the service fails for a particular record, or the keyword SKIP. The TRANSFORM function must produce a <code>resultype</code> the same as the <code>outstructure</code> and may use <code>FAILCODE</code> and/or <code>FAILMESSAGE</code> to provide details of the failure.</td>
</tr>
<tr>
<td><strong>TRIM</strong></td>
<td>Specifies all trailing spaces are removed from strings before output.</td>
</tr>
<tr>
<td><strong>RESPONSE</strong></td>
<td>Sets flag to prevent space stripping on the response.</td>
</tr>
<tr>
<td><strong>NAMESPACE</strong></td>
<td>Specifies the top level <code>namespace</code> for the SOAP request.</td>
</tr>
<tr>
<td><strong>LITERAL</strong></td>
<td>Specifies the service is not necessarily implemented in ESP.</td>
</tr>
<tr>
<td><strong>SOAPACTION</strong></td>
<td>Specifies a <code>value</code> where that <code>value</code> is a string expression typically containing a URN or URL that is required by the web <code>service</code> for proper interoperability.</td>
</tr>
<tr>
<td><strong>LOG</strong></td>
<td>If specified, writes details to the log file of the engine (hThor, Thor, or Roxie) to which the SOAPCALL is submitted.</td>
</tr>
<tr>
<td><strong>LOG</strong></td>
<td>Specifies to write minimal details of the SOAPCALL to a log file.</td>
</tr>
<tr>
<td><strong>LOG</strong></td>
<td>Specifies to add the expression to the log when performing a SOAPCALL.</td>
</tr>
</tbody>
</table>
**LOG** \((\text{start expression}, \text{end expression})\)

Specifies to add the start_expression to the log when a SOAPCALL begins and add the end_expression to the log when a SOAPCALL ends.

**ENCODING**

Specifies that the Web service being called requires a different message format, where type information is embedded in the XML.

**HTTPHEADER**

Specifies header information to be passed to the service. SOAPCALL supports multiple instances of the HTTPHEADER option if you need to specify multiple key/value header strings.

---

### SOAPCALL Function

This form of SOAPCALL, the function, may take as input either a single record or a recordset, and both types of input can result in either a single record or a recordset.

The **outstructure** output record definition may contain an integer field with an XPATH of 
"_call_latency" to receive the time, in seconds, for the call which generated the row (from creating the socket to receiving the response). The latency is placed in every row the call returned, so if a call took 90 seconds and returned 11 rows then you will see 11 rows with 90 in the _call_latency field.

**Example:**

```ecl
OutRec1 := RECORD
    STRING500 OutData{XPATH('OutData')};
    UNSIGNED4 Latency{XPATH('_call_latency')};
END;
ip := 'http://service.example.com:8022/';
ips := 'https://service.example.com:8022/';
svc := 'MyModule.SomeService';
ips_secret := 'secret:myConnectSecret';
/* assumes a secret named http-connect-myConnectSecret exists & contains: */
    {"url": "https://service.example.com:8022/",
     "username": "username",
     "password": "password"
    }
*/
//1 rec in, 1 rec out
OneRec1 := SOAPCALL(ips,svc,{STRING500 InData := 'Some Input Data'},OutRec1);

//1 rec in, recordset out
ManyRec1 := SOAPCALL(ip,svc,{STRING500 InData := 'Some Input Data'},DATASET(OutRec1));

//recordset in, 1 rec out
OneRec2 := SOAPCALL(InputDataset,ip,svc,{STRING500 InData},OutRec1);

//recordset in, recordset out
ManyRec2 := SOAPCALL(InputDataset,ips_secret,svc,{STRING500 InData := 'Some In Data'},DATASET(OutRec1));

//TRANSFORM function usage example
namesRecord := RECORD
    STRING20 surname;
    STRING10 forename;
    INTEGER2 age := 25;
END;
ds := DATASET('x',namesRecord,FLAT);
inRecord := RECORD
    STRING name{xpath('Name')};
```
**UNSIGNED6 id{XPATH('ADL')};**

END;

outRecord := RECORD
  STRING name{xpath('Name')};
  UNSIGNED6 id{XPATH('ADL')};
  REAL8 score;
END;

inRecord t(namesRecord l) := TRANSFORM
  SELF.name := l.surname;
  SELF.id := l.age;
END;

outRecord genDefault1() := TRANSFORM
  SELF.name := FAILMESSAGE;
  SELF.id := FAILCODE;
  SELF.score := (REAL8)FAILMESSAGE('ip');
END;

outRecord genDefault2(namesRecord l) := TRANSFORM
  SELF.name := l.surname;
  SELF.id := l.age;
  SELF.score := 0;
END;

OUTPUT(SOAPCALL(ip, svc,{ STRING20 surname := 'Halligan',STRING20 forename := 'Kevin';},
DATASET(outRecord), ONFAIL(genDefault1())));

OUTPUT(SOAPCALL(ds, ip, svc, inRecord, t(LEFT),DATASET(outRecord), ONFAIL(genDefault2(LEFT))));

OUTPUT(SOAPCALL(ds, ip, svc, inRecord, t(LEFT),DATASET(outRecord), ONFAIL(SKIP)));

//Using HTTPHEADER to pass Authorization info
OUTPUT(SOAPCALL(ds, ip, svc, inRecord, t(LEFT),DATASET(outRecord), ONFAIL(SKIP),
  HTTPHEADER('Authorization','Basic dXNlcm5hbWU6cGFzc3dvcmQ='),
  HTTPHEADER('MyLiteral','FOO')));

---

**SOAPCALL Action**

The second form of SOAPCALL, the action, may take as input either a single record or a recordset. Neither type of input produces any returned result—it simply launches the specified SOAP service, providing it input data.

Example:

```ecl
//1 rec in, no result
SOAPCALL( 'https://service.example.com:8022/','MyModule.SomeService',
  {STRING500 InData := 'Some Input Data'});

//recordset in, no result
SOAPCALL( InputDataset,'https://service.example.com:8022/','MyModule.SomeService',{STRING500 InData});
```

See Also: RECORD Structure, TRANSFORM Structure
### SORT

**SORT** *(recordset, value [, JOINED( joinedset )][, SKEW( limit [, target ] )][, THRESHOLD( size )][, LOCAL][, FEW][, STABLE [ ( algorithm )] | UNSTABLE [ ( algorithm )] ][, UNORDERED | ORDERED( bool )][, PARALLEL [ ( numthreads )]][, ALGORITHM( name )])*

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.</td>
</tr>
<tr>
<td>value</td>
<td>A comma-delimited list of expressions or key fields in the recordset on which to sort, with the leftmost being the most significant sort criteria. A leading minus sign (-) indicates a descending-order sort on that element. You may have multiple value parameters to indicate sorts within sorts. You may use the keyword RECORD (or WHOLE RECORD) to indicate an ascending sort on all fields, and/or you may use the keyword EXCEPT to list non-sort fields in the recordset.</td>
</tr>
<tr>
<td>JOINED</td>
<td>Optional. Indicates this sort will use the same radix-points as already used by the joinedset so that matching records between the recordset and joinedset end up on the same supercomputer nodes. Used to optimize supercomputer joins where the joinedset is very large and the recordset is small.</td>
</tr>
<tr>
<td>joinedset</td>
<td>A set of records that has been previously sorted by the same value parameters as the recordset.</td>
</tr>
<tr>
<td>SKEW</td>
<td>Optional. Indicates that you know the data is not spread evenly across nodes (is skewed) and you choose to override the default by specifying your own limit value to allow the job to continue despite the skewing.</td>
</tr>
<tr>
<td>limit</td>
<td>A value between zero (0) and one (1.0 = 100%) indicating the maximum percentage of skew to allow before the job fails (the default is 0.1 = 10%).</td>
</tr>
<tr>
<td>target</td>
<td>Optional. A value between zero (0) and one (1.0 = 100%) indicating the desired maximum percentage of skew to allow (the default is 0.1 = 10%).</td>
</tr>
<tr>
<td>THRESHOLD</td>
<td>Optional. Indicates the minimum size for a single part of the recordset before the SKEW limit is enforced.</td>
</tr>
<tr>
<td>size</td>
<td>An integer value indicating the minimum number of bytes for a single part.</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Optional. Specifies the operation is performed on each node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE. An error occurs if the recordset has been GROUPed.</td>
</tr>
<tr>
<td>FEW</td>
<td>Optional. Specifies that few records will be sorted. This prevents spilling the SORT to disk if another resource-intensive activity is executing concurrently.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies a stable sort—duplicates output in the same order they were in the input. This is the default if neither STABLE nor UNSTABLE sorting is specified. Ignored if not supported by the target platform.</td>
</tr>
<tr>
<td>algorithm</td>
<td>Optional. A string constant that specifies the sorting algorithm to use (see the list of valid values below). If omitted, the default algorithm depends on which platform is targeted by the query.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies an unstable sort—duplicates may output in any order. Ignored if not supported by the target platform.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
</tbody>
</table>
bool

When False, specifies the output record order is not significant. When True, specifies the default output record order.

PARALLEL

Optional. Try to evaluate this activity in parallel.

numthreads

Optional. Try to evaluate this activity using numthreads threads.

ALGORITHM

Optional. Override the algorithm used for this activity.

name

The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

Return:

SORT returns a set of records.

The SORT function orders the recordset according to the values specified, and (if LOCAL is not specified) partitions the result such that all records with the same values are on the same node. SORT is usually used to produce the record sets operated on by the DEDUP, GROUP, and ROLLUP functions, so that those functions may operate optimally. Sorting final output is, of course, another common use.

Sorting Algorithms

There are three sort algorithms available: quicksort, insertion sort, and heapsort. They are not all available on all platforms. Specifying an invalid algorithm for the targeted platform will generate a warning and the default algorithm for that platform will be implemented.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thor</td>
<td>Supports stable and unstable quicksort--the sort will spill to disk, if necessary. Parallel sorting happens automatically on clusters with multiple-CPU or multi-CPU-core nodes.</td>
</tr>
<tr>
<td>hthor</td>
<td>Supports stable and unstable quicksort, stable and unstable insertion sort, and stable heapsort--the sort will spill to disk, if necessary. Stable heapsort is the default if both STABLE and UNSTABLE are omitted or if STABLE is present without an algorithm parameter. Unstable quicksort is the default if UNSTABLE is present without an algorithm parameter.</td>
</tr>
<tr>
<td>Roxie</td>
<td>Supports unstable quicksort, stable insertion sort, and stable heapsort--the sort does not spill to disk. Stable heapsort is the default if both STABLE and UNSTABLE are omitted or if STABLE is present without an algorithm parameter. The insertion sort implements blocking and heapmerging when there are more than 1024 rows.</td>
</tr>
</tbody>
</table>

Quick Sort

A quick sort does nothing until it receives the last row of its input, and it produces no output until the sort is complete, so the time required to perform the sort cannot overlap with either the time to process its input or to produce its output. Under normal circumstances, this type of sort is expected to take the least CPU time. There are rare exceptional cases where it can perform badly (the famous "median-of-three killer" is an example) but you are very unlikely to hit these by chance.

On a Thor cluster where each node has multiple CPUs or CPU cores, it is possible to split up the quick sort problem and run sections of the work in parallel. This happens automatically if the hardware supports it. Doing this does not improve the amount of actual CPU time used (in fact, it fractionally increases it because of the overhead of splitting the task) but the overall time required to perform the sort operation is significantly
reduced. On a cluster with dual CPU/core nodes it should only take about half the time, only about a quarter of the time on a cluster with quad-processor nodes, etc.

**Insertion Sort**

An insertion sort does all its work while it is receiving its input. Note that the algorithm used performs a binary search for insertion (unlike the classic insertion sort). Under normal circumstances, this sort is expected to produce the worst CPU time. In the case where the input source is slow but not CPU-bound (for example, a slow remote data read or input from a slow SOAPCALL), the time required to perform the sort is entirely overlapped with the input.

**Heap Sort**

A heap sort does about half its work while receiving input, and the other half while producing output. Under normal circumstances, it is expected to take more CPU time than a quick sort, but less than an insertion sort. Therefore, in queries where the input source is slow but not CPU-bound, half of the time taken to perform the sort is overlapped with the input. Similarly, in queries where the output processing is slow but not CPU-bound, the other half of the time taken to perform the sort is overlapped with the output. Also, if the sort processing terminates without consuming all of its input, then some of the work can be avoided entirely (about half in the limiting case where no output is consumed), saving both CPU and total time.

In some cases, such as when a SORT is quickly followed by a CHOOSE, the compiler will be able to spot that only a part of the sort's output will be required and replace it with a more efficient implementation. This will not be true in the general case.

**Stable vs. Unstable**

A stable sort is required when the input might contain duplicates (that is, records that have the same values for all the sort fields) and you need the duplicates to appear in the result in the same order as they appeared in the input. When the input contains no duplicates, or when you do not mind what order the duplicates appear in the result, an unstable sort will do.

An unstable sort will normally be slightly faster than the stable version of the same algorithm. However, where the ideal sort algorithm is only available in a stable version, it may often be better than the unstable version of a different algorithm.

**Performance Considerations**

The following discussion applies principally to local sorts, since Thor is the only platform that performs global sorts, and Thor does not provide a choice of algorithms.

**CPU time vs. Total time**

In some situations a query might take the least CPU time using a quick sort, but it might take the most total time because the sort time cannot be overlapped with the time taken by an I/O-heavy task before or after it. On a system where only one subgraph or query is being run at once (Thor or nthor), this might make quick sort a poor choice since the extra time is simply wasted. On a system where many subgraphs or queries are running concurrently (such as a busy Roxie) there is a trade-off, because minimizing total time will minimize the latency for the particular query, but minimizing CPU time will maximize the throughput of the whole system.

When considering the parallel quick sort, we can see that it should significantly reduce the latency for this query; but that if the other CPUs/cores were in use for other jobs (such as when dual Thors are running
on the same dual CPU/core machines) it will not increase (and will slightly decrease) the throughput for the machines.

**Spilling to disk**

Normally, records are sorted in memory. When there is not enough memory, spilling to disk may occur. This means that blocks of records are sorted in memory and written to disk, and the sorted blocks are then merged from disk on completion. This significantly slows the sort. It also means that the processing time for the heap sort will be longer, as it is no longer able to overlap with its output.

**When there is not enough memory to hold all the records and spilling to disk is not available (like on the Roxie platform), the query will fail.**

**How sorting affects JOINs**

A normal JOIN operation requires that both its inputs be sorted by the fields used in the equality portion of the match condition. The supercomputer automatically performs these sorts “under the covers” unless it knows that an input is already sorted correctly. Therefore, some of the considerations that apply to the consideration of the algorithm for a SORT can also apply to a JOIN. To take advantage of these alternate sorting algorithms in a JOIN context you need to SORT the input dataset(s) the way you want, then specify the NOSORT option on the JOIN.

Note well that no sorting is required for JOIN operations using the KEYED (or half-keyed), LOOKUP, or ALL options. Under some circumstances (usually in Roxie queries or in those cases where the optimizer thinks there are few records in the right input dataset) the supercomputer's optimizer will automatically perform a LOOKUP or ALL join instead of a regular join. This means that, if you have done your own SORT and specified the NOSORT option on the JOIN, that you will be defeating this possible optimization.

**Example:**

```ecl
MySet1 := SORT(Person,-last_name, first_name);
// descending last name, ascending first name
MySet2 := SORT(Person,RECORD,EXCEPT per_sex,per_marital_status);
// sort by all fields except sex and marital status
MySet3 := SORT(Person,last_name, first_name,STABLE('quicksort'));
// stable quick sort, not supported by Roxie
MySet4 := SORT(Person,last_name, first_name,UNSTABLE('heapsort'));
// unstable heap sort, not supported by any platform,
// therefore ignored
MySet5 := SORT(Person,last_name,first_name,STABLE('insertionsort'));
// stable insertion sort, not supported by Thor
```

See Also: SORTED, RANK, RANKED, EXCEPT
SORTED

SORTED(recordset, value)

### recordset
The set of sorted records. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.

### value
A comma-delimited list of expressions or key fields in the recordset on which the recordset has been sorted, with the leftmost being the most significant sort criteria. A leading minus sign (-) indicates a descending-order sort on that element. You may have multiple value parameters to indicate sorts within sorts. You may use the keyword RECORD to indicate an ascending sort on all fields, and/or you may use the keyword EXCEPT to list non-sort fields in the recordset.

### index
The attribute name of an INDEX definition. This is equivalent to adding the SORTED option to the INDEX definition.

### Return:
SORTED is a compiler directive that returns nothing.

The **SORTED** function indicates to the ECL compiler that the **recordset** is already sorted according to the **values** specified. Any number of **value** parameters may be supplied, with the leftmost being the most significant sort criteria. A leading minus sign (-) on any **value** parameter indicates a descending sort for that one parameter. **SORTED** typically refers to a DATASET to indicate the order in which the data is already sorted.

**Example:**

```ecl
InputRec := RECORD
  INTEGER4 Attr1;
  STRING20 Attr2;
  INTEGER8 Cid;
END;
MyFile := DATASET('filename', InputRec, FLAT)
MySortedFile := SORTED(MyFile, MyFile.Cid)
// Input file already sorted by Cid
```

**See Also:** SORT, DATASET, RANK, RANKED, INDEX
**SQRT**

**SQRT(n)**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>The real number to evaluate.</td>
</tr>
</tbody>
</table>

**Return:** SQRT returns a single real value.

The **SQRT** function returns the square root of the parameter.

Example:

```
MyRoot := SQRT(16.0);
OUTPUT(MyRoot); // result is 4.0
```

See Also: POWER, EXP, LN, LOG
### STEPPED

**STEPPED(index, fields [, UNORDERED | ORDERED(bool)] [, STABLE | UNSTABLE] [, PARALLEL [(numthreads)] ] [, ALGORITHM(name)])**

- **index**: The INDEX to sort. This can be filtered or the result of a PROJECT on an INDEX.
- **fields**: A comma-delimited list of fields by which to sort the result, typically trailing elements in the key.
- **UNORDERED**: Optional. Specifies the output record order is not significant.
- **ORDERED**: Specifies the significance of the output record order.
- **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.
- **STABLE**: Optional. Specifies the input record order is significant.
- **UNSTABLE**: Optional. Specifies the input record order is not significant.
- **PARALLEL**: Optional. Try to evaluate this activity in parallel.
- **numthreads**: Optional. Try to evaluate this activity using numthreads threads.
- **ALGORITHM**: Optional. Override the algorithm used for this activity.
- **name**: The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

The **STEPPED** function sorts the **index** by the specified **fields**. This function is used in those cases where the SORTED(index) function will not suffice.

There are some restrictions in its use:

The key fields before ordered **fields** should be reasonably well filtered, otherwise the sorting could become very memory intensive.

Roxie only supports sorting by trailing components on indexes that are read locally (single part indexes or superkeys containing single part indexes), or NOROOT indexes read within ALLNODES.

Thor does not support STEPPED.

Example:

```ecl
DataFile := '~RTTEST::TestStepped';
KeyFile := '~RTTEST::TestSteppedKey';
Rec := RECORD
  STRING2 state;
  STRING20 city;
  STRING25 lname;
  STRING15 fname;
END;
ds := DATASET(DataFile,s
{Rec,UNSIGNED8 RecPos {VIRTUAL(fileposition)}},
THOR);
IDX := INDEX(ds,{state,city,lname,fname,RecPos},KeyFile);
OUTPUT(IDX(state IN ['FL','PA']));
/* where this OUTPUT produces this result:
FL BOCA RATON WIK PICHA
FL DELAND WIKER OKE
FL GAINESVILLE WIK MACHOUSTON
```
OUTPUT(STEPPED(IDX(state IN ['FL','PA'])),fname));
/* this STEPPED OUTPUT produces this result:
PA NEW STANTON WIKER DESSIE
FL GAINESVILLE WIK MACHOUSTON
FL DELAND WIKER OKE
FL BOCA RATON WIK PICA */

See Also: INDEX, SORTED, ALLNODES
STORED

STORED( interface )

interface The name of an INTERFACE structure attribute.

The STORED function is a shorthand method of defining attributes for use in a SOAP interface. It is equivalent to defining a MODULE structure that inherits all the attributes from the interface and adds the STORED workflow service to each, using the attribute name as the STORED name.

Example:

```
Iname := INTERFACE
EXPORT STRING20 Name;
EXPORT BOOLEAN KeepName := TRUE;
END;

StoredName := STORED(Iname);
// is equivalent to:
// StoredName := MODULE(Iname)
// EXPORT STRING20 Name := '' : STORED('name');
// EXPORT BOOLEAN KeepName := TRUE : STORED('keepname');
// END;
```

See Also: STORED Workflow Service, INTERFACE Structure, MODULE Structure
### SUM

**SUM(recordset, value, [ , expression ] [, KEYED ])**

**SUM( valuelist [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] ] [, ALGORITHM( name ) ])**

- **recordset**: The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set. This also may be the keyword GROUP to indicate finding the sum of values of the field in a group, when used in a RECORD structure to generate crosstab statistics.

- **value**: The expression to sum.

- **expression**: Optional. A logical expression indicating which records to include in the sum. Valid only when the `recordset` parameter is the keyword GROUP to indicate summing the elements in a group.

- **KEYED**: Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.

- **valuelist**: A comma-delimited list of expressions to find the sum of. This may also be a SET of values.

- **UNORDERED**: Optional. Specifies the output record order is not significant.

- **ORDERED**: Specifies the significance of the output record order.

- **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.

- **STABLE**: Optional. Specifies the input record order is significant.

- **UNSTABLE**: Optional. Specifies the input record order is not significant.

- **PARALLEL**: Optional. Try to evaluate this activity in parallel.

- **numthreads**: Optional. Try to evaluate this activity using `numthreads` threads.

- **ALGORITHM**: Optional. Override the algorithm used for this activity.

- **name**: The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

**Return:** SUM returns a single value.

The **SUM** function returns the additive sum of the `value` in each record of the `recordset` or `valuelist`.

Example:

```ecl
personRecord := RECORD
    STRING UID;
    STRING first_name;
    STRING last_name;
    INTEGER hourly_wage;
END;

person := DATASET([{'923','James','Jones',15},
    {'924','Sally','Jones',15},
    {'925','Jose','Gomez',17},
    {'926','Adam','Wesson',77},
    {'927','Evelyn','Murray',74},
    {'928','Zom','Murray',74},
    {'929','Joe','Yung',75}], personRecord);

SumOfHourlyWage := SUM(person,person.hourly_wage); // total all hourly wage values
OUTPUT(SumOfHourlyWage);
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
SumVal1 := SUM(4,8,16,2,1); //returns 31
SetVals := [4,8,16,2,1];
SumVal2 := SUM(SetVals); //returns 31
OUTPUT(SumVal1);
OUTPUT(SumVal2);

See Also: COUNT, AVE, MIN, MAX
## TABLE

```ecl
TABLE(recordset, format [ , expression [ , FEW | MANY] [ , UNSORTED] [ , LOCAL] [ , KEYED] [ , MERGE] [ , SKEW(limit, target)] [ , THRESHOLD(size)] [ , UNORDERED] [ , ORDERED(bool)] [ , STABLE | UNSTABLE] [ , PARALLEL([numthreads])]) [ , ALGORITHM(name)])
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordset</td>
<td>The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.</td>
</tr>
<tr>
<td>format</td>
<td>An output RECORD structure definition that defines the type, name, and source of the data for each field.</td>
</tr>
<tr>
<td>expression</td>
<td>Optional. Specifies a &quot;group by&quot; clause. You may have multiple expressions separated by commas to create a single logical &quot;group by&quot; clause. If expression is a field of the recordset, then there is a single group record in the resulting table for every distinct value of the expression. Otherwise expression is a LEFT/RIGHT type expression in the DEDUP manner.</td>
</tr>
<tr>
<td>FEW</td>
<td>Optional. Indicates that the expression will result in fewer than 10,000 distinct groups. This allows optimization to produce a significantly faster result.</td>
</tr>
<tr>
<td>MANY</td>
<td>Optional. Indicates that the expression will result in many distinct groups.</td>
</tr>
<tr>
<td>UNSORTED</td>
<td>Optional. Specifies that you don't care about the order of the groups. This allows optimization to produce a significantly faster result.</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.</td>
</tr>
<tr>
<td>KEYED</td>
<td>Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.</td>
</tr>
<tr>
<td>MERGE</td>
<td>Optional. Specifies that results are aggregated on each node and then the aggregated intermediaries are aggregated globally. This is a safe method of aggregation that shines particularly well if the underlying data was skewed. If it is known that the number of groups will be low then FEW will be even faster; avoiding the local sort of the underlying data.</td>
</tr>
<tr>
<td>SKEW</td>
<td>Indicates that you know the data will not be spread evenly across nodes (will be skewed and you choose to override the default by specifying your own limit value to allow the job to continue despite the skewing.)</td>
</tr>
<tr>
<td>limit</td>
<td>A value between zero (0) and one (1.0 = 100%) indicating the maximum percentage of skew to allow before the job fails (the default skew is 1.0 / &lt;number of worker nodes on cluster&gt;).</td>
</tr>
<tr>
<td>target</td>
<td>Optional. A value between zero (0) and one (1.0 = 100%) indicating the desired maximum percentage of skew to allow (the default skew is 1.0 / &lt;number of worker nodes on cluster&gt;).</td>
</tr>
<tr>
<td>THRESHOLD</td>
<td>Indicates the minimum size for a single part before the SKEW limit is enforced.</td>
</tr>
<tr>
<td>size</td>
<td>An integer value indicating the minimum number of bytes for a single part. Default is 1GB.</td>
</tr>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><strong>STABLE</strong></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>UNSTABLE</strong></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><strong>PARALLEL</strong></td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td><strong>numthreads</strong></td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td><strong>ALGORITHM</strong></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><strong>name</strong></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

**Return:** TABLE returns a new table.

The **TABLE** function is similar to OUTPUT, but instead of writing records to a file, it outputs those records in a new table (a new dataset in the supercomputer), in memory. The new table is temporary and exists only while the specific query that invoked it is running.

The new table inherits the implicit relationality the recordset has (if any), unless the optional expression is used to perform aggregation. This means the parent record is available when processing table records, and you can also access the set of child records related to each table record. There are two forms of TABLE usage: the "Vertical Slice" form, and the "CrossTab Report" form.

For the "Vertical Slice" form, there is no expression parameter specified. The number of records in the input recordset is equal to the number of records produced.

For the "CrossTab Report" form there is usually an expression parameter and, more importantly, the output format RECORD structure contains at least one field using an aggregate function with the keyword GROUP as its first parameter. The number of records produced is equal to the number of distinct values of the expression.

**Example:**

```ecl
// "vertical slice" form:
MyFormat := RECORD
  STRING25 lname := Person.per_last_name;
  Person.per_first_name;
  STRING5 NewField := '';
END;
PersonTable := TABLE(Person,MyFormat);
// adding a new field is one use of this form of TABLE

// "CrossTab Report" form:
rec := RECORD
  Person.per_st;
  StCnt := COUNT(GROUP);
END
Mytable := TABLE(Person,rec,per_st,Few);
// group persons by state in Mytable to produce a crosstab
```

See Also: OUTPUT, GROUP, DATASET, RECORD Structure
TAN

TAN(angle)

<table>
<thead>
<tr>
<th>angle</th>
<th>The REAL radian value for which to find the tangent.</th>
</tr>
</thead>
</table>

Return: TAN returns a single REAL value.

The TAN function returns the tangent of the angle.

Example:

```plaintext
Rad2Deg := 57.295779513082; //number of degrees in a radian
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad;   //translate 45 degrees into radians
Tangent45 := TAN(Angle45); //get tangent of the 45 degree angle
OUTPUT(Tangent45); // 0.9999999999961035
```

See Also: ACOS, COS, ASIN, SIN, ATAN, COSH, SINH, TANH
TANH

TANH(angle)

<table>
<thead>
<tr>
<th>angle</th>
<th>The REAL radian value for which to find the hyperbolic tangent.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>TANH returns a single REAL value.</td>
</tr>
</tbody>
</table>

The TANH function returns the hyperbolic tangent of the angle.

Example:
```plaintext
Rad2Deg := 57.295779513082; //number of degrees in a radian
Deg2Rad := 0.0174532925199; //number of radians in a degree
Angle45 := 45 * Deg2Rad; //translate 45 degrees into radians
HyperbolicTangent45 := TANH(Angle45); //get hyperbolic tangent of the angle
OUTPUT(HyperbolicTangent45); // 0.655794202631562
```

See Also: ACOS, COS, ASIN, SIN, ATAN, COSH, SINH, TAN
**THISNODE**

**THISNODE**(operation)

<table>
<thead>
<tr>
<th>operation</th>
<th>The name of an attribute or in-line code that results in a DATASET or INDEX.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>THISNODE returns a record set or index.</td>
</tr>
</tbody>
</table>

The **THISNODE** function specifies that the *operation* is performed on each node, independently. This is typically used within an ALLNODES operation. *Available for use only in Roxie.*

Example:

```ecl
ds := ALLNODES(JOIN(THISNODE(GetData(SomeData)),
  THISNODE(GetIDX(SomeIndex)),
  LEFT.ID = RIGHT.ID));
```

See Also: ALLNODES, LOCAL, NOLOCAL
TOJSON

TOJSON( record )

<table>
<thead>
<tr>
<th>record</th>
<th>The row (record) of data to convert to JSON format.</th>
</tr>
</thead>
</table>

Return: TOJSON returns a UTF8.

The TOJSON function returns a single UTF-8 string with the data in the record re-formatted as JSON. If the RECORD structure of the record has XPATHs defined, then they will be used, otherwise the lower-cased field names are used as the JSON tag names.

Example:

```ecl
namesRec1 := RECORD
  UNSIGNED2 EmployeeID{xpath('EmpID')};
  STRING10 Firstname{xpath('FName')};
  STRING10 Lastname{xpath('LName')};
END;
str1 := TOJSON(ROW({42,'Fred','Flintstone'},namesRec1));
OUTPUT(str1);
//returns this string:
//"EmpID": 42, "FName": "Fred", "LName": "Flintstone"
namesRec2 := RECORD
  UNSIGNED2 EmployeeID;
  STRING10 Firstname;
  STRING10 Lastname;
END;
str2 := TOJSON(ROW({42,'Fred','Flintstone'},namesRec2));
OUTPUT(str2);
//returns this string:
//"employeeid": 42, "firstname": "Fred", "lastname": "Flintstone"
```

See Also: ROW, FROMJSON
### TOPN

**TOPN( recordset, count, sorts [, BEST( bestvalues ) ] [ , LOCAL ] [ , UNORDERED | ORDERED( bool ) ] [ , STABLE | UNSTABLE ] [ , PARALLEL [ ( numthreads ) ] ] [ , ALGORITHM( name ) ] )**

- **recordset**: The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set.

- **count**: An integer expression defining the number of records to return.

- **sorts**: A comma-delimited list of expressions or key fields in the recordset on which to sort, with the leftmost being the most significant sort criteria. A leading minus sign (-) indicates a descending-order sort on that element. You may use the keyword RECORD to indicate an ascending sort on all fields, and/or you may use the keyword EXCEPT to list non-sort fields in the recordset.

- **BEST**: Optional. Allows early termination of the operation if there are count number of records and the values contained in the last record match the bestvalues.

- **bestvalues**: A comma-delimited list, matching the list of sorts, of maximum (or minimum if the corresponding sort is descending) values.

- **LOCAL**: Optional. Specifies the operation is performed on each supercomputer node independently, without requiring interaction with all other nodes to acquire data; the operation maintains the distribution of any previous DISTRIBUTE.

- **UNORDERED**: Optional. Specifies the output record order is not significant.

- **ORDERED**: Specifies the significance of the output record order.

- **bool**: When False, specifies the output record order is not significant. When True, specifies the default output record order.

- **STABLE**: Optional. Specifies the input record order is significant.

- **UNSTABLE**: Optional. Specifies the input record order is not significant.

- **PARALLEL**: Optional. Try to evaluate this activity in parallel.

- **numthreads**: Optional. Try to evaluate this activity using numthreads threads.

- **ALGORITHM**: Optional. Override the algorithm used for this activity.

- **name**: The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function’s STABLE and UNSTABLE options.

**Return:** TOPN returns a set of records.

The TOPN function returns the first count number of records in the sorts order from the recordset. This is roughly equivalent to CHOOSE(SORT(recordset, sorts), count) but with simpler syntax. This also returns the top number of rows in each group of GROUPed recordsets and local operations.

**Example:**

```plaintext
y := TOPN(Person,1000,state,sex); //first 1000 recs in state, sex order
z := TOPN(Person,1000,sex,BEST('F')); //first 1000 females
```

See Also: CHOOSE, SORT, GROUP
TOUNICODE

TOUNICODE( string, encoding )

| string     | The DATA string to translate. |
| encoding   | The encoding codepage (supported by IBM’s ICU) to use for the translation. |
| Return:    | TOUNICODE returns a single UNICODE value. |

The TOUNICODE function returns the string translated from the DATA value to the specified unicode encoding.

Example:

```
DATA5 x := FROMUNICODE(u'ABCDE','UTF-8');
y := TOUNICODE(x,'US-ASCII');
OUTPUT(x); //results in 4142434445
OUTPUT(y); //results in 'ABCDE'
```

See Also: FROMUNICODE, UNICODEORDER
## TOXML

**TOXML** *(record)*

<table>
<thead>
<tr>
<th><strong>record</strong></th>
<th>The row (record) of data to convert to an XML format.</th>
</tr>
</thead>
</table>

**Return:**

TOXML returns a UTF8.

The **TOXML** function returns a single UTF-8 string with the data in the *record* re-formatted as XML. If the RECORD structure of the *record* has XPATHs defined, then they will be used, otherwise the lower-cased field names are used as the XML tag names.

**Example:**

```ecl
namesRec1 := RECORD
  UNSIGNED2 EmployeeID{xpath('EmpID')};
  STRING10  Firstname{xpath('FName')};
  STRING10  Lastname{xpath('LName')};
END;
rec1 := TOXML(ROW({42,'Fred','Flintstone'},namesRec1));
OUTPUT(rec1);
//returns this string:
//'<EmpID>42</EmpID><FName>Fred</FName><LName>Flintstone</LName>'

namesRec2 := RECORD
  UNSIGNED2 EmployeeID;
  STRING10  Firstname;
  STRING10  Lastname;
END;
rec2 := TOXML(ROW({42,'Fred','Flintstone'},namesRec2));
OUTPUT(rec2);
//returns this string:
//'<employeeid>42</employeeid><firstname>Fred</firstname><lastname>Flintstone</lastname>'
```

See Also: **ROW**, **FROMXML**
TRACE

\[ \text{attrname} := \] \text{TRACE}(\text{baserecset}, [\text{options}]);

<table>
<thead>
<tr>
<th>attrname</th>
<th>Optional. The name for the expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>baserecset</td>
<td>The set of data records for which the TRACE is defined.</td>
</tr>
<tr>
<td>options</td>
<td>Optional. One or more of the options listed below.</td>
</tr>
</tbody>
</table>

The TRACE expression defines tracing to log files (Thor worker logs, hThor logs, or Roxie logs).

You can add TRACE to your code at interesting junctures without any impact on performance. Later, if you need to investigate behavior, you can enable them without modifying the code by setting an option or stored BOOLEAN.

Tracing is written to log files, in the form:

```
TRACE: <name><fieldname>value</fieldname>...</name>
```

Tracing is not output by default even if TRACE statements are present; tracing is only output when the workunit debug value traceEnabled is set or if the default platform settings are changed to always output tracing. In Roxie you can also request tracing on a deployed query by specifying traceEnabled=1 in the query XML.

It is therefore possible to leave TRACE statements in the ECL without any detectable overhead until tracing is enabled. To enable tracing:

```
#OPTION ('traceEnabled', 1) // trace statements enabled
```

It is also possible to override the default value for KEEP at a global, per-workunit, or per-query level.

```
#OPTION ('traceLimit', 100) // overrides the default KEEP value (10)
```

You can use a stored BOOLEAN as the filter expression for a trace activity to allow you to turn individual trace activities on and off.
TRACE Options

The following options are available for TRACE:

\[
\text{[filterExpression,] [KEEP(n),] [SKIP(n),] [SAMPLE(n),][NAMED(string)]}
\]

<table>
<thead>
<tr>
<th>filterExpression</th>
<th>Optional. A valid expression which acts as a filter. Only rows matching the filter condition are included in the tracing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEEP(n)</td>
<td>Optional. Specifies the number of rows to trace.</td>
</tr>
<tr>
<td>SKIP(n)</td>
<td>Optional. Specifies the number of rows to skip before trace begins.</td>
</tr>
<tr>
<td>SAMPLE(n)</td>
<td>Optional. Specifies that only every nth row is traced.</td>
</tr>
<tr>
<td>NAMED(string)</td>
<td>Optional. Specifies the name for rows in tracing.</td>
</tr>
</tbody>
</table>

Example:

```ecl
#OPTION ('traceEnabled', TRUE); //TRACE writes to log only if TRUE
FilterValue := 4;
myRec := { STRING Name, REAL x, REAL y };
ds := DATASET([ {'Jim' , 1, 1.00039},
                  {'Jane', 2, 2.07702},
                  {'Emil', 3, 2.86158},
                  {'John', 4, 3.87114},
                  {'Jean', 5, 5.12417},
                  {'Gene', 6, 6.20283} ], myRec);
myds := TRACE(ds,x>FilterValue,NAMED('person')); //trace only if x > FilterValue
myds;
```
TRANSFER

TRANSFER(value,type)

<table>
<thead>
<tr>
<th>value</th>
<th>An expression containing the bitmap to return.</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>The value type to return.</td>
</tr>
<tr>
<td>Return:</td>
<td>TRANSFER returns a single value.</td>
</tr>
</tbody>
</table>

The TRANSFER function returns the value in the requested type. This is not a type cast because the bit-pattern stays the same.

Example:

```
INTEGER1 MyInt := 65; // MyInt is an integer whose value is 65
STRING1 MyVal1 := TRANSFER(MyInt, STRING1); // MyVal1 is "A" (ASCII 65)
INTEGER1 MyVal2 := (INTEGER)MyVal1; // MyVal2 is 0 (zero) because "A" is not a numeric character
```

OUTPUT(MyVal1); // returns A
OUTPUT(MyVal2); // returns 0 (zero)

See Also: Type Casting
**TRIM**

**TRIM(string_value [ ,flag ])**

<table>
<thead>
<tr>
<th>string_value</th>
<th>The string from which to remove spaces.</th>
</tr>
</thead>
<tbody>
<tr>
<td>flag</td>
<td>Optional. Specify which spaces to remove. Valid flag values are:</td>
</tr>
<tr>
<td>RIGHT</td>
<td>remove trailing spaces--this is the default</td>
</tr>
<tr>
<td>LEFT</td>
<td>remove leading spaces</td>
</tr>
<tr>
<td>LEFT, RIGHT</td>
<td>remove leading and trailing spaces</td>
</tr>
<tr>
<td>ALL</td>
<td>remove all spaces, even those within the string_value</td>
</tr>
<tr>
<td>WHITESPACE</td>
<td>Used in conjunction with any of the other flags, this removes ALL white space characters from the specified area. If omitted, only the space character (0x20) is removed.</td>
</tr>
</tbody>
</table>

Return: TRIM returns a single value.

The TRIM function returns the string_value with all trailing and/or leading spaces (0x20) removed.

The WHITESPACE option removes all white space characters. In STRING, this is space (0x20), horizontal tab, vertical tab, line feed, form feed, carriage return (0x09 to 0x0D), and non-breaking space (0xA0). For UNICODE, it removes all characters with the white space property.

Example:

```
STRING20 MyString1 := 'ABC';
  //contains 17 trailing spaces
VARSTRING MyTrimmedVarString1 := TRIM(MyString1);
  // MyVal is "ABC" with no trailing spaces
STRING20 MyString2 := ' ABC DEF';
  //contains 2 leading and 11 trailing spaces
VARSTRING MyTrimmedVarString2 := TRIM(MyString2,LEFT,RIGHT);
    // MyVal is "ABC DEF" with no trailing spaces
OUTPUT(MyString1);
OUTPUT(MyTrimmedVarString1);
OUTPUT(MyString2);
OUTPUT(MyTrimmedVarString2);
```

See Also: STRING, VARSTRING
TRUNCATE

TRUNCATE(real_value)

**real_value**  The floating-point value to truncate.

Return: TRUNCATE returns a single integer value.

The **TRUNCATE** function returns the integer portion of the *real_value*.

Example:

```ecl
MyRealValue := 3.75;
INTEGER4 MyValue := TRUNCATE(MyRealValue);
OUTPUT(MyValue); // MyValue is 3
```

See Also: ROUND, ROUNDUP
UNGROUP


<table>
<thead>
<tr>
<th>recordset</th>
<th>The set of previously GROUPed records.</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNORDERED</td>
<td>Optional. Specifies the output record order is not significant.</td>
</tr>
<tr>
<td>ORDERED</td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td>bool</td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td>STABLE</td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td>UNSTABLE</td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td>PARALLEL</td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td>numthreads</td>
<td>Optional. Try to evaluate this activity using numthreads threads.</td>
</tr>
<tr>
<td>ALGORITHM</td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td>name</td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: UNGROUP returns a record set.

The UNGROUP function removes previous grouping. This is equivalent to using the GROUP function without a second parameter.

Example:

```ecl
personRecord := RECORD
STRING UID;
STRING first_name;
STRING last_name;
STRING address;
STRING city;
STRING state;
STRING zip;
END;
person := DATASET([{'923','James','Jones','123 Elm Street','Hollywood','FL','33022'},
{'924','Sally','Jones','22 Main Street','Tampa','FL','33604'},
{'925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101'},
{'926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108'},
{'927','Evelyn','Gomez','740 SW 10th Street','Boston ','MA','02116'},
{'928','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131'}], personRecord);
SortedSet := SORT(person,last_name);       //sort by last name
GroupedSet := GROUP(SortedSet,last_name);  //then group them
SecondSort := SORT(GroupedSet,first_name); //sorts by first name within each last name group
// this is a "sort within group"
UnGroupedSet := UNGROUP(GroupedSet);       //ungroup the dataset
```

See Also: GROUP
**UNICODEORDER**

**UNICODEORDER( left, right [, locale ] )**

- **left**: The left Unicode expression to evaluate.
- **right**: The right Unicode expression to evaluate.
- **locale**: Optional. A string constant containing a valid locale code, as specified in ISO standards 639 and 3166.

**Return:** UNICODEORDER returns a single value.

The **UNICODEORDER** function returns either -1, 0, or 1 depending on the evaluation of the **left** and **right** expressions. This is equivalent to the <=> equivalence comparison operator but taking the unicode **locale** as the basis of determination. If **left** < **right** then -1 is returned, if **left** = **right** then 0 is returned, if **left** > **right** then 1 is returned.

**Example:**

```ecl
UNICODE1 x := u'a';
UNICODE1 y := u'b';
UNICODE1 z := u'a';
a := UNICODEORDER(x, y, 'es'); // returns -1
b := UNICODEORDER(x, z, 'es'); // returns 0
c := UNICODEORDER(y, z, 'es'); // returns 1
OUTPUT(a);
OUTPUT(b);
OUTPUT(c);
```

See Also: FROMUNICODE, TOUNICODE
The **UNORDERED** function is used to indicate that the order of the records in the *dataset* is not significant. This will allow the code generator in future versions to apply extra optimizations.

Example:

```ecl
Def1 := UNORDERED(MyDataset);
//indicates that the order of MyDataset is not significant,
//so the code generator can perform optimizations based on that.
```
**VARIANCE**

VARIANCE( recset, valuex [, expression] [, KEYED] [, UNORDERED | ORDERED( bool )] [, STABLE | UNSTABLE] [, PARALLEL [ (numthreads) ]] [, ALGORITHM( name )])

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>recset</code></td>
<td>The set of records to process. This may be the name of a dataset or a record set derived from some filter condition, or any expression that results in a derived record set. This also may be the GROUP keyword to indicate operating on the elements in each group, when used in a RECORD structure to generate crosstab statistics.</td>
</tr>
<tr>
<td><code>valuex</code></td>
<td>A numeric field or expression.</td>
</tr>
<tr>
<td><code>expression</code></td>
<td>Optional. A logical expression indicating which records to include in the calculation. Valid only when the <code>recset</code> parameter is the keyword GROUP.</td>
</tr>
<tr>
<td><code>KEYED</code></td>
<td>Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.</td>
</tr>
<tr>
<td><code>UNORDERED</code></td>
<td>Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.</td>
</tr>
<tr>
<td><code>ORDERED</code></td>
<td>Specifies the significance of the output record order.</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>When False, specifies the output record order is not significant. When True, specifies the default output record order.</td>
</tr>
<tr>
<td><code>STABLE</code></td>
<td>Optional. Specifies the input record order is significant.</td>
</tr>
<tr>
<td><code>UNSTABLE</code></td>
<td>Optional. Specifies the input record order is not significant.</td>
</tr>
<tr>
<td><code>PARALLEL</code></td>
<td>Optional. Try to evaluate this activity in parallel.</td>
</tr>
<tr>
<td><code>numthreads</code></td>
<td>Optional. Try to evaluate this activity using <code>numthreads</code> threads.</td>
</tr>
<tr>
<td><code>ALGORITHM</code></td>
<td>Optional. Override the algorithm used for this activity.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.</td>
</tr>
</tbody>
</table>

Return: VARIANCE returns a single REAL value.

The **VARIANCE** function returns the (population) variance of `valuex`.

Example:

```ecl
pointRec := {REAL x, REAL y};
analyse( ds ) := MACRO
  #uniquename(stats)
  %stats% := TABLE(ds, { c := COUNT(GROUP),
    sx := SUM(GROUP, x),
    sy := SUM(GROUP, y),
    sxx := SUM(GROUP, x * x),
    sxy := SUM(GROUP, x * y),
    syy := SUM(GROUP, y * y),
    varx := VARIANCE(GROUP, x);
    vary := VARIANCE(GROUP, y);
    varxy := COVARIANCE(GROUP, x, y);
    rc := CORRELATION(GROUP, x, y) });
OUTPUT(%stats%);

  // Following should be zero
  OUTPUT(%stats%, (varx - (sxx-sx*sx/c)/c, vary - (syy-sy*sy/c)/c, varxy - (sxy-sx*sy/c)/c, rc - (varxy/SQRT(varx*vary)) ));
  OUTPUT(%stats%, {'bestFit: y' + (STRING)((sy-sx*varxy/varx)/c) + ' + ' + (STRING)(varxy/varx)+'x' });
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
ds1 := DATASET([ {1,1},{2,2},{3,3},{4,4},{5,5},{6,6} ], pointRec);

ds2 := DATASET([ {1.93896e+009, 2.04482e+009},
                 {1.77971e+009, 8.54858e+008},
                 {2.96181e+009, 1.24848e+009},
                 {2.7744e+009, 1.26357e+009},
                 {1.14416e+009, 4.3429e+008},
                 {3.38728e+009, 1.30238e+009},
                 {3.19538e+009, 1.71177e+009} ], pointRec);

ds3 := DATASET([ {1, 1.00039},
                 {2, 2.07702},
                 {3, 2.86158},
                 {4, 3.87114},
                 {5, 5.12417},
                 {6, 6.20283} ], pointRec);

analyse(ds1);
analyse(ds2);
analyse(ds3);

See Also: CORRELATION, COVARIANCE
WAIT

WAIT(event)

<table>
<thead>
<tr>
<th>event</th>
<th>A string constant containing the name of the event to wait for.</th>
</tr>
</thead>
</table>

The **WAIT** action is similar to the WHEN workflow service, but may be used within conditional code.

Example:

```ecl
//You can either do this:
action1;
action2 : WHEN('expectedEvent');
//can also be written as:
SEQUENTIAL(action1,WAIT('expectedEvent'),action2);
```

See Also: EVENT, NOTIFY, WHEN, CRON
WHEN

WHEN(trigger, action [, BEFORE | SUCCESS | FAILURE])

| trigger | A dataset or action that launches the action. |
| action  | The action to execute. |
| BEFORE  | Optional. Specifies an action that should be executed before the input is read. |
| SUCCESS | Optional. Specifies an action that should only be executed on SUCCESS of the trigger (e.g., no LIMITs exceeded). |
| FAILURE | Optional. Specifies an action that should only be executed on FAILURE of the trigger (e.g., a LIMIT was exceeded). |

The WHEN function associates an action with a trigger (dataset or action) so that when the trigger is executed the action is also executed. This allows job scheduling based upon triggers.

Example:

```ecl
//a FUNCTION with side-effect Action
namesTable := FUNCTION
  namesRecord := RECORD
    STRING20 surname;
    STRING10 forename;
    INTEGER2 age := 25;
  END;
o := OUTPUT('namesTable used by user <x>);
ds := DATASET([('x','y',22)],namesRecord);
RETURN WHEN(ds,O);
END;

z := namesTable : PERSIST('z');
//the PERSIST causes the side-effect action to execute only when the PERSIST is re-built
OUTPUT(z);
```

See Also: FUNCTION Structure, WHEN, WAIT
WHICH

WHICH(condition, ..., condition)

<table>
<thead>
<tr>
<th>condition</th>
<th>A conditional expression to evaluate.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Return:</td>
<td>WHICH returns a single value.</td>
</tr>
</tbody>
</table>

The **WHICH** function evaluates which of the list of *conditions* returned true and returns its ordinal position in the list of *conditions*. Returns zero (0) if none return true. This is the opposite of the **REJECTED** function.

Example:

```ecl
Accept := WHICH(Person.per_first_name = 'Fred', Person.per_first_name = 'Sue');
//Accept is 0 for everyone but those named Fred or Sue
```

See Also: REJECTED, MAP, CHOOSE, IF, CASE
WORKUNIT

WORKUNIT

WORKUNIT( named [, type ] )

<table>
<thead>
<tr>
<th>named</th>
<th>A string constant containing the NAMED option scalar value to return.</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Optional. The value type of the named scalar value result to return.</td>
</tr>
</tbody>
</table>

Return: WORKUNIT returns a single value.

The WORKUNIT function returns values stored in the workunit. Given no parameters, it returns the unique workunit identifier (WUID) for the currently executing workunit, otherwise it returns the NAMED option result from the OUTPUT or DISTRIBUTION action.

Example:

```ecl
wuid := WORKUNIT; //get WUID
namesRecord := RECORD
  STRING20 surname;
  STRING10 forename;
  INTEGER2 age;
END;
namesTable := DATASET([{'Halligan','Kevin',31},
                        {'Halligan','Liz',30},
                        {'Salter','Abi',10},
                        {'X','Z',42}], namesRecord);
OUTPUT(wuid);
DISTRIBUTION(namesTable, surname, forename,NAMED('Stats'));
WORKUNIT('Stats',STRING);
```

See Also: #WORKUNIT, OUTPUT, DISTRIBUTION
XMLDECODE

XMLDECODE( unicode )

| unicode   | The unicode text to decode. |
| Return:   | XMLDECODE returns a single value. |

The XMLDECODE function decodes special characters into an XML string (for example, &lt is converted to <) allowing you to use the CSV option on OUTPUT to produce more complex XML files than are possible by using the XML option.

Example:

```ecl
encoded := XMLENCODE('<xml version 1><tag>data</tag>');
decoded := XMLDECODE(encoded);
OUTPUT(encoded); // results in '&lt;xml version 1&gt;&lt;tag&gt;data&lt;/tag&gt;' 
OUTPUT(decoded); // results in '<xml version 1><tag>data</tag>'
```

See Also: XMLENCODE
### XMLENCODE

**XMLENCODE( xml [, ALL ] )**

<table>
<thead>
<tr>
<th>xml</th>
<th>The XML to encode.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>Optional. Specifies including new line characters in the encoding so the text can be used in attribute definitions.</td>
</tr>
</tbody>
</table>

**Return:** XMLENCODE returns a single value.

The **XMLENCODE** function encodes special characters in an XML string (for example, `<` is converted to `&lt`) allowing you to use the CSV option on OUTPUT to produce more complex XML files than are possible by using the XML option.

**Example:**

```plaintext
coded := XMLENCODE('<xml version 1><tag>data</tag>');
decoded := XMLDECODE(encoded);

OUTPUT(encoded); // results in '&lt;xml version 1&gt;&lt;tag&gt;data&lt;/tag&gt;
OUTPUT(decoded); // results in '<xml version 1><tag>data</tag>'
```

**See Also:** XMLDECODE
Workflow Services
Workflow Overview

Workflow control within ECL is generally handled automatically by the system. It spots which processes can happen in parallel, when synchronization is required, and when processes must happen in series. These workflow services allow exceptions to the normal flow of execution to be specified by the programmer to give extra control (such as the FAILURE clause).

Workflow operations are implicitly evaluated in a separate global scope from the code to which it is attached. Therefore, any values from the code to which it is attached (such as loop counters) are unavailable to the workflow service.

It should also be noted that when a workflow operation is present within multiple SEQUENTIAL statements only the first instance will be evaluated.

Example:

```ecl
Chesney := OUTPUT("I am the one and only!" said Chesney')
  : SUCCESS(OUTPUT("Oh yeah, prove it?")));
SEQUENTIAL(
  OUTPUT("I am Spartacus" said one from the mob'),
  Chesney
);
SEQUENTIAL(
  OUTPUT("No, I am Spartacus" confessed another'),
  Chesney,
  OUTPUT("Ok, so you are!"
)
);
```

yields:

"I am Spartacus" said one from the mob
"I am the one and only!" said Chesney
"Oh yeah, prove it?"
"No, I am Spartacus" confessed another
"Ok, so you are!"

See Also: SEQUENTIAL
CHECKPOINT

attribute := expression : CHECKPOINT( name ) ;

<table>
<thead>
<tr>
<th>attribute</th>
<th>The name of the Attribute.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The definition of the attribute.</td>
</tr>
<tr>
<td>name</td>
<td>A string constant specifying the storage name of the value.</td>
</tr>
</tbody>
</table>

The CHECKPOINT service stores the result of the expression in the workunit so it remains available if the workunit fails to complete, and is automatically deleted when the job completes successfully. This is particularly useful for attributes based on large, expensive data manipulation sequences. This service implicitly causes the attribute to be evaluated at global scope instead of any enclosing scope.

However, CHECKPOINT is only useful when the unsuccessful workunit is resubmitted through ECL Watch; if a new workunit is instantiated, CHECKPOINT has no effect. The PERSIST service is more generally useful.

Example:

CountPeople := COUNT(Person) : CHECKPOINT('PeopleCount');
://Makes CountPeople available for reuse if the job does not complete

See Also: PERSIST
attribute := expression : DEPRECATED [ ( message ) ] ;

| attribute | The name of the Attribute. |
| expression | The definition of the attribute. |
| message | Optional. The text to append to the warning if the attribute is used. |

The **DEPRECATED** service displays a warning when the `attribute` is used in code that instantiates a workunit or during a syntax check. This is meant to be used on attribute definitions that have been superseded.

When used on a structure attribute (RECORD, TRANSFORM, FUNCTION, etc.), this must be placed between the keyword END and its terminating semi-colon.

Example:

```ecl
personRecord := RECORD
STRING UID;
STRING first_name;
STRING last_name;
STRING address;
STRING city;
STRING state;
STRING zip;
END;
person := DATASET([{'923','James','Jones','123 Elm Street','Hollywood','FL',33022},
{'924','Sally','Jones','22 Main Street','Tampa','FL',33604},
{'925','Jose','Gomez','111 Biscaya Lane','Miami','FL',33101},
{'926','Adam','Wesson','77 Sunset Blvd','Boston','MA',02108},
{'927','Evelyn','Murray','740 SW 10th Street','Boston ','MA',02116},
{'928','Joe','Yung','7511 Simson Avenue','Chicago','IL',60131}], personRecord);

OldSort := SORT(person,first_name) : DEPRECATED('Use NewSort instead.);
NewSort := SORT(person,-first_name);

OUTPUT(OldSort);
//produces this warning:
// Definition OldSort is marked as deprecated. Use NewSort instead.
```
**FAILURE**

\[\text{attribute} := \text{expression} : \text{FAILURE}(\text{handler}) [,\text{LABEL}(\text{text})];\]

<table>
<thead>
<tr>
<th>attribute</th>
<th>The name of the Attribute.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The definition of the attribute.</td>
</tr>
<tr>
<td>handler</td>
<td>The action to run if the expression fails.</td>
</tr>
<tr>
<td>LABEL</td>
<td>Optional. Defines the Text to display for the workflow item in the Graph for the workunit. If omitted, the code generator will deduce a label from the identifier being defined.</td>
</tr>
<tr>
<td>text</td>
<td>A string constant containing the text to display.</td>
</tr>
</tbody>
</table>

The **FAILURE** service executes the **handler** Attribute when the **expression** fails. **FAILURE** notionally executes in parallel with the failed return of the result. This service implicitly causes the **attribute** to be evaluated at global scope instead of the enclosing scope. Only available if workflow services are turned on (see #OPTION(workflow)).

Example:

```ecl
sPeople := SORT(Person,Person.per_first_name);
uniques := COUNT(DEDUP(sPeople,Person.per_first_name AND Person.address)): FAILURE(Email.simpleSend(SystemsPersonel, SystemsPersonel.email,'ouch.htm'));
```

See Also: SUCCESS, RECOVERY
## GLOBAL - Service

**attribute** := **expression** : **GLOBAL** [ ( **cluster** [, **FEW** ] ) ];

<table>
<thead>
<tr>
<th>attribute</th>
<th>The name of the Attribute.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>expression</strong></td>
<td>Optional. A string constant specifying the name of the supercomputer cluster on which to build the attribute. This makes it possible to use the attribute on a smaller cluster when it must be built on a larger cluster, allowing for more efficient resource utilization. If omitted, the attribute is built on the currently executing cluster.</td>
</tr>
<tr>
<td><strong>cluster</strong></td>
<td>Optional. When the expression is a dataset or recordset, <strong>FEW</strong> specifies that the resulting dataset is stored completely within the workunit. If not specified, then the dataset is stored as a THOR file and the workunit contains only the name of the file.</td>
</tr>
</tbody>
</table>

The **GLOBAL** service causes the **attribute** to be evaluated at global scope instead of the enclosing scope, similar to the **GLOBAL()** function -- that is, not inside a filter/transform etc. It may be evaluated multiple times in the same workunit if it is used from multiple workflow items, but it will share code with the context it is used.

**GLOBAL** is different from **INDEPENDENT** operates in that **INDEPENDENT** is only ever executed once, while **GLOBAL** is executed once in each workflow item that uses it.

### Example:

```ecl
I := RANDOM() : INDEPENDENT;  //calculated once, period
G := RANDOM() : GLOBAL;       //calculated once in each graph

ds :=
    DATASET([1,0,0,0],[2,0,0,0]),(UNSIGNED1 rec,UNSIGNED Ival, UNSIGNED Gval, UNSIGNED Aval });

RECORDOF(ds) XF(ds L) := TRANSFORM
    SELF.Ival := I;
    SELF.Gval := G;
    SELF.Aval := RANDOM();  //calculated each time used
    SELF := L;
END;

P1 := PROJECT(ds,XF(left)) : PERSIST('~TEMP::PERSIST::IndependentVsGlobal1');
P2 := PROJECT(ds,XF(left)) : PERSIST('~TEMP::PERSIST::IndependentVsGlobal2');

OUTPUT(P1);
OUTPUT(P2);  //this gets the same Ival values as P1, but the Gval value is different than P1
```

See Also: **GLOBAL** function, **INDEPENDENT**
INDEPENDENT

attribute := expression : INDEPENDENT [(cluster [,LABEL(text)])];

<table>
<thead>
<tr>
<th>attribute</th>
<th>The name of the Attribute.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The definition of the attribute.</td>
</tr>
<tr>
<td>cluster</td>
<td>Optional. A string constant specifying the name of the Thor cluster on which execute. If omitted, the attribute is run on the currently executing cluster.</td>
</tr>
<tr>
<td>LABEL</td>
<td>Optional. Defines the Text to display for the workflow item in the Graph for the workunit. If omitted, the code generator will deduce a label from the identifier being defined.</td>
</tr>
<tr>
<td>text</td>
<td>A string constant containing the text to display.</td>
</tr>
</tbody>
</table>

The INDEPENDENT service causes the attribute to be evaluated at a global scope and forces the attribute evaluation into a separate workflow item. The new workflow item is evaluated before the first workflow item that uses that attribute. It executes independently from other workflow items, and is only executed once (including inside SEQUENTIAL where it should be executed the first time it is used). It will not share any code with any other workflow items.

One use would be to provide a mechanism to common up code that is shared between different arguments to a SEQUENTIAL action--normally they are evaluated completely independently.

Example:

```
I := RANDOM() : INDEPENDENT(LABEL('CalcRandom')); //calculated once, period
G := RANDOM() : GLOBAL; //calculated once in each graph

ds :=
  DATASET([{1,0,0,0},{2,0,0,0}]){UNSIGNED1 rec, UNSIGNED Ival, UNSIGNED Gval , UNSIGNED Aval });
RECORDOF(ds) XF(ds L) := TRANSFORM
  SELF.Ival := I;
  SELF.Gval := G;
  SELF.Aval := RANDOM(); //calculated each time used
  SELF := L;
END;
P1 := PROJECT(ds,XF(left)) : PERSIST('~TEMP::PERSIST::IndependentVsGlobal1');
P2 := PROJECT(ds,XF(left)) : PERSIST('~TEMP::PERSIST::IndependentVsGlobal2');
OUTPUT(P1);
OUTPUT(P2); //this gets the same Ival values as P1, but the Gval value is different than P1
```

See Also: GLOBAL
The **ONCE** service allows you to specify that the code is to be executed only once at query load time.

This means a Roxie query will execute it only once even if the query is executed multiple times. For a Thor or hThor query, **ONCE** behaves the same as **INDEPENDENT**.

Example:

```plaintext
InlineDCT := DICTIONARY([
{0 => 'Black', 'Greys'},
{1 => 'Brown', 'Earth'},
{2 => 'Red', 'Reds'},
{3 => 'White', 'Greys'}],
(UNSIGNED code => STRING10 color, STRING10 Tone)) : ONCE;
```

.smart

See Also: **INDEPENDENT**

attribute := expression : **ONCE** ;

<table>
<thead>
<tr>
<th>attribute</th>
<th>The name of the attribute.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The definition of the attribute.</td>
</tr>
</tbody>
</table>
ONWARNING

attribute := expression : ONWARNING(code, action) ;

<table>
<thead>
<tr>
<th>attribute</th>
<th>The name of the Attribute.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The definition of the attribute.</td>
</tr>
<tr>
<td>code</td>
<td>The number displayed in the &quot;Code&quot; column of the ECL IDE's Syntax Errors toolbox.</td>
</tr>
<tr>
<td>action</td>
<td>One of these actions: ignore, error, or warning.</td>
</tr>
</tbody>
</table>

The ONWARNING service allows you to specify how to handle specific warnings for a given attribute. You may have it treated as a warning, promote it to an error, or ignore it. Useful warnings can get lost in a sea of less-useful ones. This feature allows you to get rid of the "clutter."

This service overrides any global warning handling specified by #ONWARNING.

Example:

```ecl
rec := { STRING x } : ONWARNING(1041, ignore);
//ignore "Record doesn't have an explicit maximum record size" warning
```

See Also: #ONWARNING
**PERSIST**

```
attribute ::= expression : PERSIST( filename [, cluster ] [, CLUSTER(target)] [, EXPIRE(days)] [, REFRESH(flag)] [, SINGLE | MULTIPLE[count]] [,LABEL(text)]);
```

- **attribute**: The name of the Attribute.
- **expression**: The definition of the attribute. This typically defines a recordset (but it may be any expression).
- **filename**: A string constant specifying the storage name of the expression result. See *Scope and Logical Filenames*.
- **cluster**: Optional. A string constant specifying the name of the Thor cluster on which to re-build the attribute if/when necessary. This makes it possible to use persisted attributes on smaller clusters but have them rebuilt on larger, making for more efficient resource utilization. If omitted, the attribute is re-built on the currently executing cluster.
- **CLUSTER**: Optional. Specifies writing the filename to the specified list of target clusters. If omitted, the filename is written to the cluster on which the PERSIST executes (as specified by the cluster parameter). The number of physical file parts written to disk is always determined by the number of nodes in the cluster on which the PERSIST executes, regardless of the number of nodes on the target(s).
- **target**: A comma-delimited list of string constants containing the names of the clusters to write the filename to. The names must be listed as they appear on the ECL Watch Activity page or returned by the Std.System.Thorlib.Group() function, optionally with square brackets containing a comma-delimited list of node-numbers (1-based) and/or ranges (specified with a dash, as in n-m) to indicate the specific set of nodes to write to.
- **EXPIRE**: Optional. Specifies the filename is a temporary file that may be automatically deleted after the specified number of days.
- **days**: Optional. The number of days after which the file may be automatically deleted. If omitted, it defaults to use the PersistExpiryDefault setting in Sasha.
- **REFRESH**: Optional. Option to control when the PERSIST rebuilds. If omitted, the PERSIST rebuilds if 1) the underlying file does not exist, or 2) the data has changed, or 3) the code has changed.
- **flag**: A boolean value indicating whether to rebuild the PERSIST. When set to FALSE, the PERSIST rebuilds ONLY if the underlying file does not exist. If your PERSIST layout has changed and you specify REFRESH(FALSE) the mismatch could cause your job to fail.
- **SINGLE**: Optional. Specifies to keep a single PERSIST. The name of the persist file is the same as the name of the persist. The default is MULTIPLE(-1) which retains all.
- **MULTIPLE**: Optional. Specifies to keep different versions of the PERSIST. The name of the persist file generated is a combination of the name supplied suffixed with a 32-bit value derived from the ECL.
- **count**: Optional. The number of versions of a PERSIST to keep. If omitted, the system default is used. If set to -1, then an unlimited number are kept.
- **LABEL**: Optional. Defines the Text to display for the workflow item in the Graph for the workunit. If omitted, the code generator will deduce a label from the identifier being defined.
- **text**: A string constant containing the text to display.

The PERSIST service stores the result of the expression globally so it remains permanently available for use (including the result of any DISTRIBUTE or GROUP operation in the expression). This is particularly...
useful for attributes based on large, expensive data manipulation sequences. The attribute is re-calculated only when the ECL code or underlying data that was used to create it have changed, otherwise the attribute data is simply returned from the stored name file on disk when referenced. This service implicitly causes the attribute to be evaluated at global scope instead of the enclosing scope.

PERSIST may be combined with the WHEN clause so that even though the attribute may be used more than once, its execution is based upon the WHEN clause (or the first use of the attribute) and not upon the number of times the attribute is used in the computation. This gives a kind of "compute in anticipation" capability.

Persisted attributes can still be subject to the ordering requirements of SEQUENTIAL. However, since PERSISTs are shared between workunits, there is no guarantee that the attribute will be evaluated within any given evaluation order.

You can use #OPTION to override the default settings, as shown in the example.

Example:

```
// #OPTION ('multiplePersistInstances', true|false); // if true retains MULTIPLE, if false SINGLE
// #OPTION ('defaultNumPersistInstances', <n>);       // the number to retain if MULTIPLE allowed.
// Defaults to -1 (retain all)

CountPeople := COUNT(Person) : PERSIST('PeopleCount');
  //Makes CountPeople available for use in all subsequent work units

sPeople := SORT(Person,first_name): PERSIST('SortPerson'),WHEN(Daily);
  //Makes sPeople available for use in all subsequent work units

s1 := SORT(Person,first_name): PERSIST('SortPerson1','OtherThor');
  //run the code on the OtherThor cluster
s2 := SORT(Person,first_name): PERSIST('SortPerson2','OtherThor',CLUSTER('AnotherThor'));
  //run the code on the OtherThor cluster
  // and write the file to the AnotherThor cluster
```

See Also: STORED, WHEN, GLOBAL, CHECKPOINT, #OPTION
# ECL Language Reference

## Workflow Services

## PRIORITY

**action**: `PRIORITY(value)`;

<table>
<thead>
<tr>
<th>action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>An integer in the range 0-100 indicating the relative importance of the action.</td>
</tr>
</tbody>
</table>

The **PRIORITY** service establishes the relative importance of multiple actions in the workunit. The higher value an action has, the greater its priority. The highest priority action executes first, if possible. PRIORITY is not allowed on attribute definitions, it must only be associated with an action. Only available if workflow services are turned on (see `#OPTION(workflow)`).

**Example:**

```ecl
personRecord := RECORD
  STRING UID;
  STRING first_name;
  STRING last_name;
  STRING address;
  STRING city;
  STRING state;
  STRING zip;
END;

person := DATASET(
  [{'923','James','Jones','123 Elm Street','Hollywood','FL','33022'},
   {'924','Sally','Jones','22 Main Street','Tampa','FL','33604'},
   {'925','Jose','Gomez','111 Biscaya Lane','Miami','FL','33101'},
   {'926','Adam','Wesson','77 Sunset Blvd','Boston','MA','02108'},
   {'927','Evelyn','Murray','740 SW 10th Street','Boston ','MA','02116'},
   {'928','Joe','Yung','7511 Simson Avenue','Chicago','IL','60131'}],
  personRecord);

OUTPUT(Person(state='MA')) : PRIORITY(30);
OUTPUT(Person(state='IL')) : PRIORITY(60);
OUTPUT(Person(state='FL')) : PRIORITY(90);
```

See Also: OUTPUT, #OPTION
**RECOVERY**

\[
\text{attribute} := \text{expression} : \text{RECOVERY}(\text{handler} [, \text{attempts}]) ;
\]

| attribute | The name of the Attribute. |
| expression | The definition of the attribute. |
| handler | The action to run if the expression fails. |
| attempts | Optional. The number of times to try before giving up. |

The **RECOVERY** service executes the *handler* Attribute when the *expression* fails then re-runs the *attribute*. If the *attribute* still fails after the specified number of *attempts*, any present FAILURE clause will execute. RECOVERY notionally executes in parallel with the failed return result. This service implicitly causes the *attribute* to be evaluated at global scope instead of the enclosing scope. Only available if workflow services are turned on (see #OPTION(workflow)).

Example:

```ecl
DoSomethingToFixIt := TRUE; // some action to repair the input
SPeople := SORT(Person,first_name);
nUniques := DEDUP(sPeople,first_name)
: RECOVERY(DoSomethingToFixIt,2),
    FAILURE(Email.simpleSend(SystemsPersonel,SystemsPersonel.email,'ouch.htm'));
```

See Also: SUCCESS, FAILURE
STORED - Workflow Service

```
STORED( storedname [, FEW ][, FORMAT( [SELECT(valuestring)] [FIELDWIDTH(widthvalue)] [FIELDHEIGHT(heightvalue)] [SEQUENCE(sequencevalue)] [NOINPUT]] [PASSWORD]) )
```

<table>
<thead>
<tr>
<th>attribute</th>
<th>Optional. The name of the Attribute.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The definition of the attribute.</td>
</tr>
<tr>
<td>storedname</td>
<td>A string constant containing the name of the stored attribute result.</td>
</tr>
<tr>
<td>FEW</td>
<td>Optional. When the expression is a dataset or recordset, FEW specifies that the dataset is stored completely within the workunit. If not specified, then the dataset is stored as a THOR file and the workunit contains only the name of the file. The FEW option is required when using STORED in a SOAP-enabled MACRO and the expected input is a dataset (such as tns:xmlDataset).</td>
</tr>
<tr>
<td>FORMAT</td>
<td>Optional. FORMAT specifies options for formatting the field on a Web form in WsECL.</td>
</tr>
<tr>
<td>SELECT</td>
<td>Optional. SELECT specifies a droplist input control on a Web form in WsECL.</td>
</tr>
<tr>
<td>valuestring</td>
<td>An string containing the possible values for the droplist. An asterisk (*) denotes the default value. A expression in the form of 'apple=1' within the string allows text to display and a different value to be stored. In that example, apple would display but a value of 1 is stored if the user selects apple.</td>
</tr>
<tr>
<td>FIELDWIDTH</td>
<td>Optional. FIELDWIDTH specifies the width of the input box on a Web form in WsECL.</td>
</tr>
<tr>
<td>widthvalue</td>
<td>An integer expression defining the width (number of characters) of the input box</td>
</tr>
<tr>
<td>FIELHEIGHT</td>
<td>Optional. FIELHEIGHT specifies the height of the input box on a Web form in WsECL.</td>
</tr>
<tr>
<td>heightvalue</td>
<td>An integer expression defining the height (number of rows) of the input box</td>
</tr>
<tr>
<td>SEQUENCE</td>
<td>Optional. SEQUENCE specifies field ordering on a Web form in WsECL.</td>
</tr>
<tr>
<td>sequencevalue</td>
<td>An integer expression defining the sequential location of the input box. These can be sparse values (e.g., 100, 200, 300) to allow insertion of new inputs in the future.</td>
</tr>
<tr>
<td>NOINPUT</td>
<td>Optional. If NOINPUT is specified, the field is not displayed on the Web form in WsECL.</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>Optional. If PASSWORD is specified, a password entry box is used on the Web form in WsECL and the field's supplied value is not displayed while entering it. The value is also hidden when viewing stored values in the workunit through EclWatch or from the command line when extracting the WU XML.</td>
</tr>
</tbody>
</table>

The `STORED` service stores the result of the `expression` with the work unit that uses the `attribute` so that it remains available for use throughout the work unit. If the `attribute` name is omitted, then the stored value can only be accessed afterwards from outside of the ECL execution. If an `attribute` name is provided then the value of that `attribute` will be pulled from storage, if it has not yet been set it will be computed, stored and then used from storage. This service implicitly causes the `attribute` to be evaluated at a global scope instead of the enclosing scope.

STORED creates a storage space in the workunit where the interface can place the values to pass to a published query. See Working with Roxie in the Programmer's Guide.

Example:

```
COUNT(person) : STORED('myCount');
  // Name in workunit is myCount,
  // stored value accessible only outside ECL
fred := COUNT(person) : STORED('fred');
```
// Name in workunit is fred
fred := COUNT(person) : STORED('mindy');
   // Name in workunit is mindy

//FORMAT options for WsECL form
Password :='' := STORED('Password',FORMAT(SEQUENCE(1),PASSWORD));
   //password entry box on form
Field1 := 1 : STORED('Field1',FORMAT(SEQUENCE(10)));
Field2 := 2 : STORED('Field2',FORMAT(SEQUENCE(20)));
AddThem := TRUE :STORED ('AddThem',FORMAT(SEQUENCE(15)));
   // places field in between Field1 and Field2
HiddenValue := 12 :STORED ('HiddenValue',FORMAT(NOINPUT)); // not on form
TextField1 :='Fill in description' :Stored('Description',
       FORMAT(FIELDWIDTH(25),FIELDHEIGHT(2),
             SEQUENCE(5)));
       //Creates 25 char wide, 2 row high input box

//SELECT options
UNSIGNED8 u8 := 0   : STORED('u8', FORMAT(fieldwidth(8),
             SEQUENCE(18),
             SELECT('one=1,two=2,three=3,*four=4')));
STRING ch1 := 'ban' : STORED('ch1', FORMAT(SELECT('apple=app,pear,*banana=ban')));
   //banana is default
STRING ch2 := ''    : STORED('ch2', FORMAT(SELECT(',apple=app,pear,banana=ban')));
   //starts empty, no specified default
STRING ch3 := ''    : STORED('ch3', FORMAT(SELECT('apple=app,pear,*banana=ban')));
   //empty in middle, empty is default

See Also: STORED function, #WEBSERVICE
SUCCESS

attribute := expression : SUCCESS(handler) [,LABEL(text)];

| attribute | The name of the Attribute. |
| handler   | The action to run if the expression succeeds. |
| LABEL     | Optional. Defines the Text to display for the workflow item in the Graph for the workunit. If omitted, the code generator will deduce a label from the identifier being defined. |
| text      | A string constant containing the text to display. |

The SUCCESS service executes the handler Attribute when the expression succeeds. SUCCESS notionally executes in parallel with the successful return of the result. This service implicitly causes the attribute to be evaluated at global scope instead of the enclosing scope. Only available if workflow services are turned on (see #OPTION(workflow)).

Example:

```ecl
$People := SORT(Person,Person.first_name);
nUniques := COUNT(DEDUP($People,Person.per_first_name AND Person.address))
          : SUCCESS(Email.simpleSend(SystemsPersonel,
                                     SystemsPersonel.email,'yeah.htm'));
```

See Also: FAILURE, RECOVERY
WHEN

\[ \text{WHEN}( \text{event} [ , \text{COUNT}( \text{repeat} ) ] ) ; \]

<table>
<thead>
<tr>
<th><strong>action</strong></th>
<th>Any valid ECL Action to execute.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>event</strong></td>
<td>The event that triggers action execution. This may be either the EVENT or CRON functions, EVENTNAME or the name of an EVENT (as a shorthand for EVENT(event,'*')), or any attribute defined with those functions.</td>
</tr>
<tr>
<td><strong>COUNT</strong></td>
<td>Optional. Specifies the number of events to trigger instances of the action. If omitted, the default is unlimited (continuously waiting for another event to trigger another instance of the action), until the workunit is manually removed from the list of workunits being monitored by the scheduler.</td>
</tr>
<tr>
<td><strong>repeat</strong></td>
<td>An integer expression.</td>
</tr>
</tbody>
</table>

The WHEN service executes the **action** whenever the **event** occurs.

Example:

```
IMPORT STD;
IF (STD.File.FileExists('test::myfile'),
   STD.File.DeleteLogicalFile('test::myfile'));
   //deletes the file if it already exists
STD.File.MonitorLogicalFileName('MyFileEvent','test::myfile');
   //sets up monitoring and the event name
   //to fire when the file is found
OUTPUT('File Created') : WHEN(EVENT('MyFileEvent','*'));
   //this OUTPUT occurs only after the event has fired
   //may also be coded in this shorthand form:
   // OUTPUT('File Created') : WHEN('MyFileEvent');
afile := DATASET(
   [{'A', '0'}], {STRING10 key,STRING10 val});
OUTPUT(afile,'test::myfile');
   //this creates a file that the DFU file monitor will find
   //when it periodically polls
   //******************************************************************************
EXPORT events := MODULE
   EXPORT dailyAtMidnight := CRON('0 0 * * *');
   EXPORT dailyAt ( INTEGER hour,
      INTEGER minute=0 ) :=
      EVENT('CRON',
         (STRING)minute + ' ' + (STRING)hour + ' * * *');
   EXPORT dailyAtMidday := dailyAt(12, 0);
END;
BUILD(teenagers) : WHEN(events.dailyAtMidnight);
BUILD(oldies) : WHEN(events.dailyAt(6));
BUILD(oldies) : WHEN(EVENT('FileDropped', 'x'));
```

See Also: EVENT, CRON, NOTIFY, WAIT
Template Language Overview

ECL was created to be the programming language for all of our HPCC Systems technology. Therefore, it must be able to meet all the demands of a complete business solution: from data ingest, through querying and processing, and all the way to fulfillment and customer output.

In most every business solution that we create, the end-users will be using some kind of a custom Graphical User Interface (GUI) application specific to their business (typically created for them by us) to specify their queries into the data and set up processing jobs for the supercomputer. These custom GUI applications can generate for the user the ECL that will actually perform the query or process. The task of generating that ECL can be daunting if approached through a hard-coding perspective when you consider the exponential curve of all possible sets of choices the user could make in any moderately-complex system, and as the system grows more complex the problem becomes even worse. That means that a hard-coding solution is out of the question.

ECL's Template language provides the solution to this problem. The Template language is a Meta-language that takes standard XML input, typically generated from an end-user GUI application (thereby vastly simplifying the coding problem in the GUI) and in turn generating the appropriate ECL code to implement the user's choices.

Template Language Statements

Template Language statements all begin with # to clearly differentiate them from the ECL code that will be generated by the template. Most statements take parameters that determine their specific action in each instance.

The required statement terminator is the semi-colon (just as in ECL) and there are multi-line structures that terminate with the #END statement. These structures may be nested within each other.

Template Symbols

Template Language uses user-defined symbols as variables. These symbols must be explicitly declared before use (see #DECLARE). The tag names in the XML text being processed are also treated like user-defined symbols.

A user-defined symbol or XML tag is referenced by surrounding the name of the symbol or tag with percent signs. An XML tag used as a template symbol may be a simple tag name, or an xpath to the XML data to retrieve (see the RECORD structure documentation for a description of the supported xpath syntax). If an xpath is used, then the symbol used must be the full xpath to the data expressed inside curly braces ({}).

This syntax takes several forms:

| %symbol%                | returns the value of the symbol          |
| %'symbol'%.            | returns value of the symbol as a string  |
| %"%                   | (an empty string) returns the contents of the current XML tag |
| %{xpath}%             | returns the value of the data pointed to by the xpath |
| %'{xpath}'%           | returns value of the data pointed to by the xpath as a string |
#APPEND

`#APPEND( symbol, expression );`

<table>
<thead>
<tr>
<th>symbol</th>
<th>The name of a previously declared user-defined symbol.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The string expression specifying the string to concatenate to the existing symbol contents.</td>
</tr>
</tbody>
</table>

The **#APPEND** statement adds the value of the **expression** to the end of the existing string contents of the **symbol**.

Example:

```ecl
#DECLARE(MySymbol); //declare a symbol named "MySymbol"
#SET(MySymbol,'Hello'); //initialize MySymbol to "Hello"
#APPEND(MySymbol,' World'); //make MySymbol's value "Hello World"
OUTPUT('%'MySymbol'%);
```

See Also: **#DECLARE**, **#SET**
#CONSTANT

#CONSTANT(name, value);

<table>
<thead>
<tr>
<th>name</th>
<th>A string constant containing the name of the stored value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>An expression for the value to assign to the stored name.</td>
</tr>
</tbody>
</table>

The #CONSTANT statement is similar to #STORED in that it assigns the value to the name, but #CONSTANT specifies the value is not over-writable at runtime. This statement may be used outside an XML scope and does not require a previous LOADXML to instantiate an XML scope.

Example:

```plaintext
PersonCount := 0 : STORED('myCount');
#CONSTANT('myCount',100);
    //set the stored PersonCount attribute value to 100
OUTPUT(PersonCount);
```

See Also: #STORED
#DECLARE

#DECLARE( symbol);

<table>
<thead>
<tr>
<th>symbol</th>
<th>The name of the template variable.</th>
</tr>
</thead>
</table>

The #DECLARE statement declares a user-defined symbol for use in the template. The symbol is simply created and not initialized to any particular value, therefore it may be destined to contain either string or numeric data.

Example:

```ecl
#DECLARE(MySymbol); //declare a symbol named "MySymbol"
#SET(MySymbol,11); //initialize MySymbol to 11
OUTPUT('%'MySymbol'%)
```

See Also: #SET, #APPEND
#DEMANGLE

#DEMANGLE( identifier );

| identifier | A valid ECL identifier label containing only letters, numbers, dollar sign ($), and underscore (_) characters. |

The #DEMANGLE statement takes an identifier string and returns the string as it was before it was #MANGLEd.

Example:

```ecl
#DECLARE (mstg);
#DECLARE (dmstg);
#SET (mstg, #MANGLE('SECTION STATES/AREACODES'));
EXPORT res1 := %'mstg'%;
OUTPUT(res1); //res1 = 'SECTION_5fSTATES_2fAREACODES'

// Do some processing with ECL Valid Label name "mstg"
#SET (dmstg, #DEMANGLE(%'mstg'%));
EXPORT res2 := %'dmstg'%;
OUTPUT(res2); //res2 = 'SECTION STATES/AREACODES'
```

See Also: #MANGLE, Attribute Names
#ERROR

#ERROR(  errormessage  );

<table>
<thead>
<tr>
<th>errormessage</th>
<th>A string expression containing the message to display.</th>
</tr>
</thead>
</table>

The #ERROR statement immediately halts processing on the workunit and displays the errormessage. This statement may be used outside an XML scope and does not require a previous LOADXML to instantiate an XML scope.

Example:

```ecl
a := TRUE; // pick one of these
//a := FALSE;
#IF(a)
  #ERROR('broken');
  OUTPUT('broken');
#ELSE
  #WARNING('maybe broken');
  OUTPUT('maybe broken');
#END;
```

See Also: #WARNING
#EXPAND

#EXPAND( token );

| token | The name of the MACRO parameter whose passed string constant value to expand. |

The #EXPAND statement substitutes and parses the text of the passed token's string within the MACRO.

Example:

MAC_join(attrname, leftDS, rightDS, linkflags) := MACRO
   attrname := JOIN(leftDS,rightDS,#EXPAND(linkflags));
ENDMACRO;

MAC_join(J1,People,Property,'LEFT.ID=RIGHT.PeopleID,LEFT OUTER')
   // expands out to:
   // J1 := JOIN(People,Property,LEFT.ID=RIGHT.PeopleID,LEFT OUTER);

MAC_join(J2,People,Property,'LEFT.ID=RIGHT.PeopleID')
   // expands out to:
   // J2 := JOIN(People,Property,LEFT.ID=RIGHT.PeopleID);

See Also: MACRO
#EXPORT

The `#EXPORT(symbol, data);` statement produces XML text from the specified `data` and places it in the `symbol`. This allows the LOADXML(symbol,name) form to instantiate an XML scope on the information from the `data` to process.

The XML output is generated with the following format:

```
<Data>
    <Field label="<label-of-field>"
           name="<name-of-field>"
           position="<n>"
           rawtype="<n>"
           size="<n>"
           type="<ecl-type-without-size>" />
    ...
</Data>
```

IFBLOCKs are simply expanded out in the XML. Nested RECORD types have an isRecord attribute that is set to 1, and are followed by the fields they contain, and then a Field tag with no name and the isEnd attribute set to 1. This representation is used rather than nested objects so it can be processed by a #FOR statement. Child dataset types are also expanded out in a similar way, and have an isDataset attribute set to 1 on the field.

Example:

```ecl
NamesRecord := RECORD
    STRING10 first;
    STRING20 last;
END;
r := RECORD
    UNSIGNED4 dg_parentid;
    STRING10 dg_firstname;
    STRING dg_lastname;
    UNSIGNED1 dg_prange;
    IFBLOCK(SELF.dg_prange % 2 = 0)
        STRING20 extrafield;
    END;
    NamesRecord namerec;
    DATASET(NamesRecord) childNames;
END;

ds := DATASET('~RTTEST::OUT::ds', r, thor);

#DECLARE(out);
#EXPORT(out, r);
OUTPUT(%'out'%);
/* produces this result: */
<Data>
    <Field label="DG_ParentID"
           name="DG_ParentID"
           position="0"
           rawtype="262401"
           size="4"
           type="unsigned integer"/>
    <Field label="DG_firstname"
           name="DG_firstname"
```
```xml
<Data>
  <Field label="DG_lastname"
    name="DG_lastname"
    position="2"
    rawtype="-983036"
    size="-15"
    type="string"/>
  <Field label="DG_Prange"
    name="DG_Prange"
    position="3"
    rawtype="65793"
    size="1"
    type="unsigned integer"/>
  <Field label="ExtraField"
    name="ExtraField"
    position="4"
    rawtype="1310724"
    size="20"
    type="string"/>
  <Field isRecord="1"
    label="namerec"
    name="namerec"
    position="5"
    rawtype="13"
    size="30"
    type="namesRecord"/>
  <Field label="first"
    name="first"
    position="6"
    rawtype="655364"
    size="10"
    type="string"/>
  <Field label="last"
    name="last"
    position="7"
    rawtype="1310724"
    size="20"
    type="string"/>
  <Field isEnd="1" name="namerec"/>
  <Field isDataset="1"
    label="childNames"
    name="childNames"
    position="9"
    rawtype="-983020"
    size="30"
    type="table of &lt;unnamed&gt;"/>
  <Field label="first"
    name="first"
    position="9"
    rawtype="655364"
    size="10"
    type="string"/>
  <Field label="last"
    name="last"
    position="10"
    rawtype="1310724"
    size="20"
    type="string"/>
  <Field isEnd="1" name="childNames"/>
</Data>
*/
```
// which you can then process like this:
LOADXML('%'out%', 'Fred');
#FOR (Fred)
  #FOR (Field)
    #IF (%'{@isEnd}'% <> '')
      OUTPUT('END');
    #ELSE
      OUTPUT('%'{@type}'%
        #IF (%'{@size}'% <> '-15' AND
            %'{@isRecord}'%='' AND
            %'{@isDataset}'%='')
          + %'{@size}'%
        #END
        + ' ' + %'{@label}'% + ';');
    #END
  #END
#END
OUTPUT('Done');

See Also: LOADXML, #EXPORTXML, #DECLARE
#EXPORTXML

#EXPORTXML(symbol, data);

<table>
<thead>
<tr>
<th>symbol</th>
<th>The name of a template variable that has not been previously declared.</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>The name of a field, RECORD structure, or dataset.</td>
</tr>
</tbody>
</table>

The #EXPORTXML statement produces the same XML as #EXPORT from the specified data and places it in the symbol, then does a LOADXML(symbol, 'label') on the data.

Example:

```ecl
NamesRecord := RECORD
    STRING10 first;
    STRING20 last;
END;

r := RECORD
    UNSIGNED4 dg_parentid;
    STRING10 dg_firstname;
    STRING dg_lastname;
    UNSIGNED1 dg_prange;
    IFBLOCK(SELF.dg_prange % 2 = 0)
        STRING20 extrafield;
    END;
NamesRecord namerec;
DATASET(NamesRecord) childNames;
END;

ds := DATASET('~RTTEST::OUT::ds', r, THOR);

//This example produces the same result as the example for #EXPORT.
//Notice the lack of #DECLARE and LOADXML in this version:
#EXPORTXML(Fred,r);

#FOR (Fred)
#FOR (Field)
#IF (%'{@isEnd}'% <> '')
OUTPUT('END');
#ELSE
OUTPUT('%'{@type}'% + %'{@size}'% + ' ' + %'{@label}'% + ';');
#ENDIF
#END
#END
OUTPUT('Done');
```

//These examples show some other possible uses of #EXPORTXML:

```ecl
//This could be greatly simplified as
// {%'(IsAStringMetaInfo/Field[1]/@type)'%='string'}
isAString(inputField) := MACRO
#EXPORTXML(IsAStringMetaInfo, inputField);
#IF (%'IsAString'%='')
#DECLARE(IsAString);
#END;
```
#SET(IsAString, false);
#FOR (IsAStringMetaInfo)
  #FOR (Field)
    #IF (%'{@type}'% = 'string')
      #SET (IsAString, true);
    #END
    #BREAK
  #END
#END
%IsAString%
ENDMACRO;

getFieldName(inputField) := MACRO
  #EXPORTXML(GetFieldNameMetaInfo, inputField);
  %'{GetFieldNameMetaInfo/Field[1]/@name}'%
ENDMACRO;

displayIsAString(inputField) := MACRO
  OUTPUT(getFieldName(inputField)
    + TRIM(IF(isAString(inputField), ' is', ' is not'))
    + ' a string.'
ENDMACRO;

SIZEOF(r.dg_firstname);
isAString(r.dg_firstname);
getFieldName(r.dg_firstname);
OUTPUT('ds.dg_firstname isAString? ' 
  + (STRING)isAString(ds.dg_firstname));
isAString(ds.namerec);

displayIsAString(ds.namerec);
displayIsAString(r.dg_firstname);

See Also: LOADXML, #EXPORT
#FOR

#FOR(  tag [[ ( filter ) ]] )

statements

#END

tag  An XML tag.

filter  A logical expression indicating which specific tag instances to process.

statements  The Template statements to execute.

#END  The #FOR structure terminator.

The #FOR structure loops through the XML, searching for each instance of the tag that meets the filter expression and executes the statements on the data contained within that tag.

Example:

```ecl
// This script processes XML and generates ECL COUNT statements
// which run against the datasets and filters specified in the XML.
XMLstuff :=
  '"<section>','
    '"<item>' +
      '"<dataset>person</dataset>'+
      '"<filter>firstname = \"RICHARD\"</filter>'+
    '"</item>'+
    '"<item>' +
      '"<dataset>person</dataset>'+
      '"<filter>firstname = \"JOHN\"</filter>'+
    '"</item>'+
    '"<item>' +
      '"<dataset>person</dataset>'+
      '"<filter>firstname = \"HENRY\"</filter>'+
    '"</item>'+
  '"</section>'';
LOADXML(XMLstuff);
#DECLARE(CountStr); // Declare CountStr
#SET(CountStr,''); // Initialize it to an empty string
#FOR(item)
  #APPEND(CountStr,'COUNT(' + %'dataset'% + '(' + %'filter'% + ' ));
');
#END
OUTPUT(%'CountStr%); // output the string just built
%CountStr% // then execute the generated "COUNT" actions

// Note that the "CountStr" will have 3 COUNT actions in it:
// COUNT(person(person.firstname = 'RICHARD'));
// COUNT(person(person.firstname = 'JOHN'));
// COUNT(person(person.firstname = 'HENRY'));
```

See Also: #LOOP, #DECLARE

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
#GETDATATYPE

#GETDATATYPE( field );

| field | A previously defined user-defined symbol containing the name of a field in a dataset. |

The #GETDATATYPE function returns the value type of the field. The field argument must be a scalar value or a field from a dataset (rather than a field from a record definition.)

Example:

```ecl
person := DATASET([{'D'6789ABCDE6789ABCDE'}], {'DATA9 per_cid'});
#DECLARE(fieldtype);
#DECLARE(field);
#SET(field, 'person.per_cid');
#SET(fieldtype, #GETDATATYPE(%field%));
res := %fieldtype%;
OUTPUT(res); // 'data9'
```

See Also: Value Types
#IF

#IF( condition )
truestatements
[#ELSEIF ( condition )
truestatements ]
[#ELSE falsestatements ]
#END

<table>
<thead>
<tr>
<th>condition</th>
<th>A logical expression.</th>
</tr>
</thead>
<tbody>
<tr>
<td>truestatements</td>
<td>The Template statements to execute if the condition is true.</td>
</tr>
<tr>
<td>#ELSEIF</td>
<td>Optional. Provides structure for statements to execute if its condition is true.</td>
</tr>
<tr>
<td>#ELSE</td>
<td>Optional. Provides structure for statements to execute if the condition is false.</td>
</tr>
<tr>
<td>falsestatements</td>
<td>Optional. The Template statements to execute if the condition is false.</td>
</tr>
<tr>
<td>#END</td>
<td>The #IF structure terminator.</td>
</tr>
</tbody>
</table>

The #IF structure evaluates the condition and executes either the truestatements or falsestatements (if present). This statement may be used outside an XML scope and does not require a previous LOADXML to instantiate an XML scope.

Example:

```ecl
// This script creates a set attribute definition of the 1st 10
// natural numbers and defines an attribute named "Set10"
#DECLARE (SetString);
#DECLARE (Ndx);
#SET (SetString, '['); //initialize SetString to [ 
#SET (Ndx, 1); //initialize Ndx to 1
#LOOP
  #IF (%Ndx% > 9) //if we've iterated 9 times
    #BREAK // break out of the loop
  #ELSE //otherwise
    #APPEND (SetString, %'Ndx'% + ','); //append Ndx and comma to SetString
    #SET (Ndx, %Ndx% + 1); //and increment the value of Ndx
  #END
#END
#APPEND (SetString, %'Ndx'% + ']'); //add 10th element and closing ]
EXPORT Set10 := %'SetString'%; //generate the ECL code
  // This generates:  
  // EXPORT Set10 := [1,2,3,4,5,6,7,8,9,10];
OUTPUT(Set10); // [1,2,3,4,5,6,7,8,9,10]
```

See Also: #LOOP, #DECLARE

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
#IFDEFINED

#IFDEFINED( definition, defaultValue )

<table>
<thead>
<tr>
<th>definition</th>
<th>The name of an ECL definition.</th>
</tr>
</thead>
<tbody>
<tr>
<td>defaultValue</td>
<td>The default value to use if the definition does not exist.</td>
</tr>
</tbody>
</table>

#IFDEFINED determines whether or not the definition has been defined at the point it appears in the code. If it has been defined, it generates the definition. If it has not been defined, it generates the defaultValue.

Example:

```ecl
definitions := MODULE
   EXPORT val1 := 'hi';
END;
root := MODULE
   EXPORT val2 := 'defined';
END;

val1 := 'me';
#IFDEFINED(val1, 'val1 undefined'); //me
IF(#IFDEFINED(val1, 'val1 undefined') = 'me', 'FRED', 'JOEY'); //FRED
IF(#IFDEFINED(val1, 'val1 undefined') = 'me too', 'FRED', 'JOEY'); //JOEY

#IFDEFINED(val2, 'val2 undefined'); //val2 undefined
#IFDEFINED(definitions.val1, 'definitions.val1 undefined'); //hi
#IFDEFINED(root.val2, 'root.val2 undefined'); //defined
```

See Also: #ISDEFINED
#ISDEFINED

**#ISDEFINED( definition )**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>definition</strong></td>
<td>The name of an ECL definition.</td>
</tr>
<tr>
<td><strong>Returns</strong></td>
<td>A BOOLEAN value</td>
</tr>
</tbody>
</table>

#ISDEFINED returns whether or not the definition has been defined at the point it appears in the code.

Example:

```
getFlagsValue(defname) := MACRO
    defname :=
        #IF (#ISDEFINED(DoAlways))
          1 +
        #ELSEIF (#ISDEFINED(DoNever))
          2 +
        #END
          3;
ENDMACRO;

#ISDEFINED(DoNever); //false
getFlagsValue(x1);   //neither is defined yet = 3
OUTPUT(x1);          //3

DoNever := 'fred';
getFlagsValue(x2);   //only DoNever is defined = 5
OUTPUT(x2);          //5

DoAlways := 42;
getFlagsValue(x3);   //both are defined = 4 (so the #ELSEIF isn't evaluated)
OUTPUT(x3);          //4

#ISDEFINED(DoNever);  //true
```

See Also: #IFDEFINED
#INMODULE

#INMODULE( module, attribute );

| module | A previously defined user-defined symbol containing the name of an ECL source module. |
| attribute | A previously defined user-defined symbol containing the name of an Attribute that may or may not be in the module. |

The #INMODULE statement returns a Boolean TRUE or FALSE as to whether the attribute exists in the specified module.

Example:

```ecl
#DECLARE (mod)
#DECLARE (attr)
#DECLARE (stg)

#SET(mod, 'default')
#SET(attr, 'YearOf')

#IF(#INMODULE(%mod%, %attr%) )
   #SET(stg, %'attr'% + ' Exists In Module ' + %'mod'%);
#ELSE
   #SET(stg, %'attr'% + ' Does Not Exist In Module ' + %'mod'%);
#END

EXPORT res := %'stg'%;
OUTPUT(res);
```
#LOOP / #BREAK

#LOOP

[ statements ]

#BREAK

[ statements ]

#END

The #LOOP structure iterates, executing the statements each time through the loop until a #BREAK statement executes. If there is no #BREAK then #LOOP iterates infinitely.

Example:

// This script creates a set attribute definition of the 1st 10
// natural numbers and defines an attribute named "Set10"

#DECLARE (SetString)
#DECLARE (Ndx)
#SET (SetString, '[');   //initialize SetString to [
#SET (Ndx, 1);          //initialize Ndx to 1
#LOOP
  #IF (%Ndx% > 9)  //if we've iterated 9 times
    #BREAK         // break out of the loop
  #ELSE            //otherwise
    #APPEND (SetString, %'Ndx'% + ',');  //append Ndx and comma to SetString
    #SET (Ndx, %Ndx% + 1)  //and increment the value of Ndx
  #END
#END
#APPEND (SetString, %'Ndx'% + ']'); //add 10th element and closing ]
EXPORT Set10 := %'SetString'%; //generate the ECL code
// This generates:
// EXPORT Set10 := [1,2,3,4,5,6,7,8,9,10];
OUTPUT(Set10); // [1,2,3,4,5,6,7,8,9,10]

See Also: #FOR, #DECLARE, #IF
#MANGLE

#MANGLE( string);

| string | A string value. |
|-------------------|

The #MANGLE statement takes any string and returns a valid ECL identifier label containing only letters, numbers, and underscore (_) characters. #MANGLE replaces non-alphanumeric characters with an underscore (_) followed by the hex value of the character it's replacing.

Example:

```
#DECLARE (mstg);
#DECLARE (dmstg);
#SET (mstg, #MANGLE('SECTION_STATES/AREACODES'));
EXPORT res1 := '%mstg%';
OUTPUT(res1); //res1 = 'SECTION_5fSTATES_2fAREACODES'
```

// Do some processing with ECL Valid Label name "mstg"
```
#SET (dmstg, #DEMANGLE('%mstg%'));
EXPORT res2 := '%dmstg%';
OUTPUT(res2); //res2 = 'SECTION_STATES/AREACODES'
```

See Also: #DEMANGLE, Attribute Names
#ONWARNING

#ONWARNING(code, action);

<table>
<thead>
<tr>
<th>code</th>
<th>The number displayed in the &quot;Code&quot; column of the ECL IDE's Syntax Errors toolbox.</th>
</tr>
</thead>
<tbody>
<tr>
<td>action</td>
<td>One of these actions: ignore, error, or warning.</td>
</tr>
</tbody>
</table>

The #ONWARNING statement allows you to globally specify how to handle specific warnings. You may have it treated as a warning, promote it to an error, or ignore it. Useful warnings can get lost in a sea of less-useful ones. This feature allows you to get rid of the "clutter."

The ONWARNING workflow service overrides any global warning handling specified by #ONWARNING.

Example:

```ecl
#ONWARNING(1041, error);
//globally promote "Record doesn't have an explicit maximum record size" warnings to errors
rec := { STRING x } : ONWARNING(1041, ignore);
//ignore "Record doesn't have an explicit maximum record size" warning on this attribute, only
```

See Also: ONWARNING
#OPTION

#OPTION (option, value);

<table>
<thead>
<tr>
<th>option</th>
<th>A case sensitive string constant containing the name of the option to set.</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>The value to set the option to. This may be any type of value, dependent on what the option expects to be.</td>
</tr>
</tbody>
</table>

The #OPTION statement is typically a compiler directive giving hints to the code generator as to how best to generate the executable code for a workunit. This statement may be used outside an XML scope and does not require a previous call to the LOADXML function to instantiate an XML scope.

Definition of Terms

These definitions are "internal-only" terms used in the option definitions that follow.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFA</td>
<td>Deterministic Finite-state Automaton.</td>
</tr>
<tr>
<td>Fold</td>
<td>To turn a complex expression into a simpler equivalent one. For example, the expression &quot;1+1&quot; can be replaced with &quot;2&quot; without altering the result.</td>
</tr>
<tr>
<td>Spill</td>
<td>Writing intermediate result sets to disk so that memory is available for subsequent steps.</td>
</tr>
<tr>
<td>Funnel</td>
<td>The + (append file) operator between datasets can be visualized as pouring all the records into a funnel and getting a single stream of records out of the bottom; hence the term &quot;funnel.&quot;</td>
</tr>
<tr>
<td>TopN</td>
<td>An internally generated activity used in place of CHOOSEN(SORT(xx), n) where n is small, as it can be computed much more efficiently than sorting the entire record set then discarding all but the first n.</td>
</tr>
<tr>
<td>Activity</td>
<td>An ECL operator that takes one or more datasets as inputs.</td>
</tr>
<tr>
<td>Graph</td>
<td>All the Activities in a query.</td>
</tr>
<tr>
<td>Subgraph</td>
<td>A collection of Activities that can all be active at the same time in Thor.</td>
</tr>
<tr>
<td>Peephole</td>
<td>A method of code optimization that looks at a small amount of the unoptimized code at a time, in order to combine operations into more efficient ones.</td>
</tr>
</tbody>
</table>

Available options

The following options are generally useful:

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxRunTime</td>
<td>none</td>
<td>Sets the maximum number of seconds a job runs before it times out.</td>
</tr>
<tr>
<td>freezePersist</td>
<td>false</td>
<td>If true, does not calculate/recalculate PERSISTed.</td>
</tr>
<tr>
<td>expirePersist</td>
<td>true</td>
<td>If true, PERSISTs expire after the specified period. This is set in the Sasha configuration setting (PersistExpiryDefault) or using #option (‘defaultPersistExpiry’, n) where n is the number of days.</td>
</tr>
<tr>
<td>defaultPersistExpiry</td>
<td>none</td>
<td>If set, PERSISTs expire after the number of days specified (overriding the Sasha PersistExpiryDefault setting).</td>
</tr>
<tr>
<td>multiplePersistInstances</td>
<td>true</td>
<td>If true, multiple PERSISTs are the default.</td>
</tr>
<tr>
<td>Variable Name</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>defaultNumPersistInstances</code></td>
<td>none</td>
<td>Specifies the default number of PERSISTs. A value of -1 specifies that all copies should be kept until they expire or manually deleted.</td>
</tr>
<tr>
<td><code>check</code></td>
<td>true</td>
<td>If true, check for potential overflows of records.</td>
</tr>
<tr>
<td><code>expandRepeatAnyAsDfa</code></td>
<td>true</td>
<td>If true, expand ANY* in a DFA.</td>
</tr>
<tr>
<td><code>forceFakeThor</code></td>
<td>false</td>
<td>If true, force code to use hthor.</td>
</tr>
<tr>
<td><code>forceGenerate</code></td>
<td>false</td>
<td>If true, force .SO to be generated even if it's not worth it</td>
</tr>
<tr>
<td><code>globalFold</code></td>
<td>true</td>
<td>If true, perform a global constant fold before generating.</td>
</tr>
<tr>
<td><code>globalOptimize</code></td>
<td>false</td>
<td>If true, perform a global optimize.</td>
</tr>
<tr>
<td><code>groupAllDistribute</code></td>
<td>false</td>
<td>If true, GROUP,ALL generates a DISTRIBUTE instead of a global SORT.</td>
</tr>
<tr>
<td><code>maximizeLexer</code></td>
<td>false</td>
<td>If true, maximize the amount of work done in the lexer.</td>
</tr>
<tr>
<td><code>maxLength</code></td>
<td>4096</td>
<td>Specify maximum length of a record.</td>
</tr>
<tr>
<td><code>minimizeSpillSize</code></td>
<td>false</td>
<td>If true, if a spill is filtered/deduped etc when read, reduce spill file size by splitting, filtering and then writing.</td>
</tr>
<tr>
<td><code>optimizeGraph</code></td>
<td>true</td>
<td>If true, optimize expressions in a graph before generating</td>
</tr>
<tr>
<td><code>orderDiskFunnel</code></td>
<td>true</td>
<td>If true, if all inputs to a funnel are disk reads, pull in</td>
</tr>
<tr>
<td><code>parseDfaComplexity</code></td>
<td>2000</td>
<td>Maximum complexity of expression to convert to a DFA.</td>
</tr>
<tr>
<td><code>pickBestEngine</code></td>
<td>true</td>
<td>If true, use hthor if it is more efficient than Thor</td>
</tr>
<tr>
<td><code>diskReadsAreSimple</code></td>
<td>true</td>
<td>If true, modifies the behavior of the pickBestEngine option so disk read operations are regarded the same as index read operations when deciding whether Thor is needed. The benefit is that simple jobs can run on hthor reading/filtering data remotely using dafilesrv.</td>
</tr>
<tr>
<td><code>targetClusterType</code></td>
<td>hthor</td>
<td>Thor</td>
</tr>
<tr>
<td><code>topnLimit</code></td>
<td>10000</td>
<td>Maximum number of records to do topN on.</td>
</tr>
<tr>
<td><code>outputLimit</code></td>
<td>10</td>
<td>Sets maximum size (in Mb) of result stored in workunit.</td>
</tr>
<tr>
<td><code>sortIndexPayload</code></td>
<td>true</td>
<td>Specifies sorting (or not) payload fields</td>
</tr>
<tr>
<td><code>workflow</code></td>
<td>true</td>
<td>Specifies enabling/disabling workflow services.</td>
</tr>
<tr>
<td><code>foldStored</code></td>
<td>false</td>
<td>Specifies that all the stored variables are replaced with their default values, or values overridden by #stored. This can significantly reduce the size of the graph generated.</td>
</tr>
<tr>
<td><code>skipFileFormatCrcCheck</code></td>
<td>false</td>
<td>Specifies that the CRC check on indices produces a warning and not an error.</td>
</tr>
<tr>
<td>Variable</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>allowedClusters</td>
<td>none</td>
<td>Specifies the comma-delimited list of cluster names (as a string constant) where the workunit may execute. This allows the job to be switched between clusters, manually or automatically, if the workunit is blocked on its assigned cluster and another valid cluster is available for use.</td>
</tr>
<tr>
<td>AllowAutoQueueSwitch</td>
<td>false</td>
<td>If true, specifies that the workunit is automatically re-assigned to execute on another available cluster listed in allowedClusters when blocked on its assigned cluster.</td>
</tr>
<tr>
<td>performWorkflowCse</td>
<td>false</td>
<td>If true, specifies that the code generator automatically detects opportunities for Common Sub-expression Elimination that may be &quot;buried&quot; within multiple PERSISTed attributes. If false, notification of these opportunities are displayed to the programmer as suggestions for the use of the INDEPENDENT Workflow Service.</td>
</tr>
<tr>
<td>defaultSkewError</td>
<td>none</td>
<td>A value between 0.0 and 1.0 that determines the amount of skew needed to generate a skew error. This value is ignored if the ECL has provided a SKEW attribute.</td>
</tr>
<tr>
<td>defaultSkewWarning</td>
<td>none</td>
<td>A value between 0.0 and 1.0 that determines the amount of skew needed to generate a skew warning. If set higher than defaultSkewError, then the value is ignored.</td>
</tr>
<tr>
<td>overrideSkewError</td>
<td>none</td>
<td>If set to a value between 0.0 and 1.0, it overrides any ECL SKEW(nn) attribute values in the current job.</td>
</tr>
<tr>
<td>defaultSkewThreshold</td>
<td>1GB</td>
<td>The size of the dataset (in bytes) local to a single node needed before Skew errors/warnings are generated if no THRESHOLD(nn) was supplied in ECL.</td>
</tr>
<tr>
<td>overrideSkewThreshold</td>
<td>none</td>
<td>The size of the dataset (in bytes) local to a single node needed before Skew errors/warnings are generated. Overrides any ECL THRESHOLD(nn) attribute values in the current job.</td>
</tr>
<tr>
<td>applyInstantEclTransformations</td>
<td>false</td>
<td>Limit non-file outputs with a CHOOSEEN</td>
</tr>
<tr>
<td>applyInstantEclTransformationsLimit</td>
<td>100</td>
<td>Number of records to limit to</td>
</tr>
<tr>
<td>divideByZero</td>
<td>zero</td>
<td>'zero' evaluates to 0, the default behavior. 'fail' causes the job to fail and report a division by zero error. 'nan' (only currently supported for real numbers) creates a quiet NaN, which will propagate through any real expressions it is used in. You can use NOT ISVALID(x) to test if the value is a NaN. Integer and decimal division by zero continue to return 0.</td>
</tr>
<tr>
<td>outputLimitMb</td>
<td>10 [MB]</td>
<td>Limit of output to a workunit in MB.</td>
</tr>
<tr>
<td><strong>hthorMemoryLimit</strong></td>
<td>Default 300 [MB]</td>
<td>Override memory usage limit set in ECL Agent's defaultMemoryLimitMB configuration option (for hThor only).</td>
</tr>
<tr>
<td>----------------------</td>
<td>------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>maxCsvRowSizeMb</strong></td>
<td>Default 10 [MB]</td>
<td>Upper limit of a CSV line read in MB.</td>
</tr>
<tr>
<td><strong>validateFileType</strong></td>
<td>Default true</td>
<td>If false, the engines use the definition in the ECL workunit and ignore the file type from the logical file meta data. If true, this check is always ignored if the ECL is reading a CSV or a fixed record width flat file. Also when true, if the ECL is reading XML or JSON, and there is a mismatch, it issues a warning not an error.</td>
</tr>
<tr>
<td><strong>compressInternalSpills</strong></td>
<td>Default true</td>
<td>Compress internal spills. (e.g., spills created by lookahead or sort gathering).</td>
</tr>
<tr>
<td><strong>hdCompressorType</strong></td>
<td>Default 'FLZ'</td>
<td>Distribute compressor to use.</td>
</tr>
<tr>
<td><strong>hdCompressorOptions</strong></td>
<td>Default ''</td>
<td>Distribute compressor options (e.g., AES key)</td>
</tr>
<tr>
<td><strong>splitterSpill</strong></td>
<td>Default -1</td>
<td>Integer value to indicate whether to force splitters to spill or not. [1 = force spill</td>
</tr>
<tr>
<td><strong>loopMaxEmpty</strong></td>
<td>Default 1000</td>
<td>Max # of iterations that LOOP can cycle through without results before reporting an error</td>
</tr>
<tr>
<td><strong>smallSortThreshold</strong></td>
<td>Default 0 (disabled)</td>
<td>If estimated size is below this threshold in bytes, a minisort approach should be used.</td>
</tr>
<tr>
<td><strong>sort_max_deviance</strong></td>
<td>Default 10 [MB]</td>
<td>Max (byte) variance allowed during sort partitioning</td>
</tr>
<tr>
<td><strong>joinHelperThreads</strong></td>
<td>Default same as number of cores</td>
<td>Number of threads to use in threaded variety of join helper</td>
</tr>
<tr>
<td><strong>bindCores</strong></td>
<td>Default 0</td>
<td>For Roxie queries. If non-zero, binds the query to only use the specified number of cores. This overrides the value set for coresPerQuery in Roxie configuration.</td>
</tr>
<tr>
<td><strong>translateDFSLayouts</strong></td>
<td>Default 0</td>
<td>Specifies that file layouts should be looked up at compile time. See File Layout Resolution at Compile Time in the Programmer's Guide for more details.</td>
</tr>
<tr>
<td><strong>timeLimit</strong></td>
<td></td>
<td>For Roxie queries. Maximum run time (in ms) for a query.</td>
</tr>
<tr>
<td><strong>generateGlobalId</strong></td>
<td>Default false</td>
<td>For Roxie queries. When true, generates a unique GlobalId if one is not provided.</td>
</tr>
<tr>
<td><strong>analyzeWorkunit</strong></td>
<td></td>
<td>Overrides the setting in ECL Agent to analyze workunits after ECL queries are executed (Thor only). This allows a workunit to be further analyzed to identify and display any potential issues. These possible issues display in ECL Watch's &quot;Warnings &amp; Errors&quot; area. The global setting defaults to</td>
</tr>
</tbody>
</table>
### Template Language

**maxCost**

Default: none

Overrides the limit setting in Thor's configuration. If the maxCost threshold is reached, the job guillotine is enforced and the job is halted. This does not override the hardlimit setting. This is only valid for Thor jobs.

The following options are all about generating Logical graphs in a workunit.

Logical graphs are stored in the workunit and viewed in ECL Watch. They include information about which attribute/line number/column the symbols are defined in. Exported attributes are represented by `<module>.<attribute>` in the header of the activity. Non-exported (local) attributes are represented as `<module>.<exported-attribute>::<non-exported-name>`.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>generateLogicalGraph</strong></td>
<td>false</td>
<td>If true, generates a Logical graph in addition to all the workunit graphs.</td>
</tr>
<tr>
<td><strong>generateLogicalGraphOnly</strong></td>
<td>false</td>
<td>If true, generates only the Logical graph for the workunit.</td>
</tr>
<tr>
<td><strong>logicalGraphExpandPersist</strong></td>
<td>true</td>
<td>If true, generates expands PERSISTed attributes.</td>
</tr>
<tr>
<td><strong>logicalGraphExpandStored</strong></td>
<td>false</td>
<td>If true, generates expands STORED attributes.</td>
</tr>
<tr>
<td><strong>logicalGraphIncludeName</strong></td>
<td>true</td>
<td>If true, generates attribute names in the header of the activity boxes.</td>
</tr>
<tr>
<td><strong>logicalGraphIncludeModule</strong></td>
<td>true</td>
<td>If true, generates module.attribute names in the header of the activity boxes.</td>
</tr>
<tr>
<td><strong>logicalGraphDisplayJavadoc</strong></td>
<td>true</td>
<td>If true, generates the Javadoc-style comments embedded in the ECL in place of the standard text that would be generated (see <a href="http://java.sun.com/j2se/javadoc/writing-doccomments/">http://java.sun.com/j2se/javadoc/writing-doccomments/</a>). Javadoc-style comments on RECORD structures or scalar attributes will not generate, as they have no graph Activity box directly associated.</td>
</tr>
<tr>
<td><strong>logicalGraphDisplayJavadocParameters</strong></td>
<td>false</td>
<td>If true, generates information about parameters in any Javadoc-style comments.</td>
</tr>
<tr>
<td><strong>filteredReadSpillThreshold</strong></td>
<td>2</td>
<td>Filtered disk reads are spilled if will be duplicated more than N times.</td>
</tr>
<tr>
<td><strong>foldConstantCast</strong></td>
<td>true</td>
<td>If true, (cast)value is folded at generate time.</td>
</tr>
<tr>
<td><strong>foldFilter</strong></td>
<td>true</td>
<td>If true, filters are constant folded.</td>
</tr>
<tr>
<td><strong>foldAssign</strong></td>
<td>true</td>
<td>If true, TRANSFORMs are constant folded.</td>
</tr>
<tr>
<td><strong>foldSQL</strong></td>
<td>true</td>
<td>If true, SQL is constant folded.</td>
</tr>
<tr>
<td><strong>optimizeDiskRead</strong></td>
<td>true</td>
<td>If true, include project and filter in the transform for a disk read.</td>
</tr>
<tr>
<td><strong>optimizeSQL</strong></td>
<td>false</td>
<td>If true, optimize SQL.</td>
</tr>
<tr>
<td><strong>optimizeThorCounts</strong></td>
<td>true</td>
<td>If true, convert COUNT(diskfile) into optimized version.</td>
</tr>
<tr>
<td><strong>peephole</strong></td>
<td>true</td>
<td>If true, peephole optimize memcpy/memsets, etc.</td>
</tr>
<tr>
<td>Option</td>
<td>Default:</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>spotCSE</td>
<td>true</td>
<td>If true, look for common sub-expressions in TRANSFORMs/filters.</td>
</tr>
<tr>
<td>noteRecordSizeInGraph</td>
<td>true</td>
<td>Add estimates of record sizes to the graph</td>
</tr>
<tr>
<td>showActivitySizeInGraph</td>
<td>false</td>
<td>Show estimates of generated C++ size in the graph</td>
</tr>
<tr>
<td>showMetaInGraph</td>
<td>false</td>
<td>Add distribution/sort orders to the graph</td>
</tr>
<tr>
<td>showRecordCountInGraph</td>
<td>true</td>
<td>Show estimates of record counts in the graph</td>
</tr>
<tr>
<td>spotTopN</td>
<td>true</td>
<td>If true, convert CHOOSEN(SORT()) into a topN activity.</td>
</tr>
<tr>
<td>spotLocalMerge</td>
<td>false</td>
<td>If true, if local JOIN and both sides are sorted, generate a light-weight merge.</td>
</tr>
<tr>
<td>countIndex</td>
<td>false</td>
<td>If true, optimize COUNT(index) into optimized version (also requires optimizeThorCounts).</td>
</tr>
<tr>
<td>allowThroughSpill</td>
<td>true</td>
<td>If true, allow through spills.</td>
</tr>
<tr>
<td>optimizeBoolReturn</td>
<td>true</td>
<td>If true, improve code when returning BOOLEAN from a function.</td>
</tr>
<tr>
<td>optimizeSubString</td>
<td>true</td>
<td>If true, don't allocate memory when doing a substring.</td>
</tr>
<tr>
<td>thorKeys</td>
<td>true</td>
<td>If true, allow INDEX operations in Thor.</td>
</tr>
<tr>
<td>regexVersion</td>
<td>0</td>
<td>If set to 1, specifies use of the previous regular expression implementation, which may be faster but also may exceed stack limits.</td>
</tr>
<tr>
<td>compileOptions</td>
<td>none</td>
<td>Specify override compiler options (such as /Zm1000 to double the compiler heap size to workaround a heap overflow error).</td>
</tr>
<tr>
<td>linkOptions</td>
<td>none</td>
<td>Specify override linker options.</td>
</tr>
<tr>
<td>optimizeProjects</td>
<td>true</td>
<td>If false, disables automatic field projection/distribution optimization.</td>
</tr>
<tr>
<td>notifyOptimizedProjects</td>
<td>0</td>
<td>If set to 1, reports optimizations to named attributes. If set to 2, reports all optimizations.</td>
</tr>
<tr>
<td>optimizeProjectsPreservePersists</td>
<td>false</td>
<td>If true, disables automatic field projection/distribution optimization around reading PERSISTed files. If a PERSISTed file is read on a different size cluster than it was created on, optimizing the projected fields can mean that the distribution/sort order cannot be recreated.</td>
</tr>
<tr>
<td>aggressiveOptimizeProjects</td>
<td>false</td>
<td>If true, enables attempted minimization of network traffic for sorts/distributes. This option doesn't usually result in significant benefits, but may do so in some specific cases.</td>
</tr>
<tr>
<td>percolateConstants</td>
<td>true</td>
<td>If false, disables attempted aggressive constant value optimizations.</td>
</tr>
</tbody>
</table>

The following options are useful for debugging:

<table>
<thead>
<tr>
<th>Option</th>
<th>Default:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>debugNlp</td>
<td>false</td>
<td>If true, output debug information about the NLP processing to the .cpp file.</td>
</tr>
</tbody>
</table>
### Resource Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>resourceMaxMemory</code></td>
<td>400M</td>
<td>Maximum amount of memory a subgraph can use.</td>
</tr>
<tr>
<td><code>resourceMaxSockets</code></td>
<td>2000</td>
<td>Maximum number of sockets a subgraph can use.</td>
</tr>
<tr>
<td><code>resourceMaxActivities</code></td>
<td>200</td>
<td>Maximum number of activities a subgraph can contain.</td>
</tr>
<tr>
<td><code>unlimitedResources</code></td>
<td>false</td>
<td>If true, assume lots of resources when resourcing the graphs.</td>
</tr>
<tr>
<td><code>traceRowXML</code></td>
<td>false</td>
<td>If true, turns on tracing in ECL Watch graphs. This should only be used with small datasets for debugging purposes.</td>
</tr>
<tr>
<td><code>_Probe</code></td>
<td>false</td>
<td>If true, display all result rows from intermediate result sets in the graph in ECL Watch when used in conjunction with the traceRowXML option. This should only be used with small datasets for debugging purposes.</td>
</tr>
<tr>
<td><code>debugQuery</code></td>
<td>false</td>
<td>If true, compile query using debug settings.</td>
</tr>
<tr>
<td><code>optimizeLevel</code></td>
<td>3 (for roxie, else 0)</td>
<td>Set the C++ compiler optimization level (optimizations can cause the compiler to take a lot longer).</td>
</tr>
<tr>
<td><code>checkAsserts</code></td>
<td>true</td>
<td>If true, enables ASSERT checking.</td>
</tr>
<tr>
<td><code>soapTraceLevel</code></td>
<td>1</td>
<td>The level of detail in reporting SOAPCALL or HTTPCALL information (set to 0 for none, 1 for normal, 2 - 8 for more detail)</td>
</tr>
<tr>
<td><code>traceEnabled</code></td>
<td>FALSE</td>
<td>Enables tracing to log files when TRACE actions are present. See TRACE.</td>
</tr>
<tr>
<td><code>traceLimit</code></td>
<td>10</td>
<td>Overrides the the default KEEP setting for a TRACE statement to indicate how many TRACE statement to write to log file. See TRACE.</td>
</tr>
</tbody>
</table>

### Advanced Code Generation Options

These options should be left alone unless you REALLY know what you are doing. Typically they are used internally by our developers to enable/disable features that are still in development. Occasionally the technical support staff will suggest that you change one of these settings to work around a problem that you encounter, but otherwise the default settings are recommended in all cases.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filteredReadSpillThreshold</code></td>
<td>2</td>
<td>Filtered disk reads are spilled if will be duplicated more than N times.</td>
</tr>
<tr>
<td><code>foldConstantCast</code></td>
<td>true</td>
<td>If true, (cast)value is folded at generate time.</td>
</tr>
<tr>
<td><code>foldFilter</code></td>
<td>true</td>
<td>If true, filters are constant folded.</td>
</tr>
<tr>
<td><code>foldAssign</code></td>
<td>true</td>
<td>If true, TRANSFORMs are constant folded.</td>
</tr>
<tr>
<td><code>foldSQL</code></td>
<td>true</td>
<td>If true, SQL is constant folded.</td>
</tr>
<tr>
<td><code>optimizeDiskRead</code></td>
<td>true</td>
<td>If true, include project and filter in the transform for a disk read.</td>
</tr>
<tr>
<td><code>optimizeSQL</code></td>
<td>false</td>
<td>If true, optimize SQL.</td>
</tr>
<tr>
<td><code>optimizeThorCounts</code></td>
<td>true</td>
<td>If true, convert COUNT(diskfile) into optimized version.</td>
</tr>
<tr>
<td><code>peephole</code></td>
<td>true</td>
<td>If true, peephole optimize memcpy/memsets, etc.</td>
</tr>
</tbody>
</table>

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
### Template Language

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>spotCSE</strong></td>
<td>true</td>
<td>If true, look for common sub-expressions in TRANSFORMs/filters.</td>
</tr>
<tr>
<td><strong>spotTopN</strong></td>
<td>true</td>
<td>If true, convert CHOOSEN(SORT()) into a topN activity.</td>
</tr>
<tr>
<td><strong>spotLocalMerge</strong></td>
<td>false</td>
<td>If true, if local JOIN and both sides are sorted, generate a light-weight merge.</td>
</tr>
<tr>
<td><strong>countIndex</strong></td>
<td>false</td>
<td>If true, optimize COUNT(index) into optimized version (also requires optimizeThorCounts).</td>
</tr>
<tr>
<td><strong>allowThroughSpill</strong></td>
<td>true</td>
<td>If true, allow through spills.</td>
</tr>
<tr>
<td><strong>optimizeBoolReturn</strong></td>
<td>true</td>
<td>If true, improve code when returning BOOLEAN from a function.</td>
</tr>
<tr>
<td><strong>optimizeSubString</strong></td>
<td>true</td>
<td>If true, don't allocate memory when doing a substring.</td>
</tr>
<tr>
<td><strong>thorKeys</strong></td>
<td>true</td>
<td>If true, allow INDEX operations in thor.</td>
</tr>
<tr>
<td><strong>regexVersion</strong></td>
<td>0</td>
<td>If set to 1, specifies use of the previous regular expression implementation, which may be faster but also may exceed stack limits.</td>
</tr>
<tr>
<td><strong>compileOptions</strong></td>
<td>none</td>
<td>Specify override compiler options (such as /Zm1000 to double the compiler heap size to workaround a heap overflow error).</td>
</tr>
<tr>
<td><strong>linkOptions</strong></td>
<td>none</td>
<td>Specify override linker options.</td>
</tr>
<tr>
<td><strong>optimizeProjects</strong></td>
<td>true</td>
<td>If false, disables automatic field projection/distribution optimization.</td>
</tr>
<tr>
<td><strong>notifyOptimizedProjects</strong></td>
<td>0</td>
<td>If set to 1, reports optimizations to named attributes. If set to 2, reports all optimizations.</td>
</tr>
<tr>
<td><strong>optimizeProjectsPreservePersists</strong></td>
<td>false</td>
<td>If true, disables automatic field projection/distribution optimization around reading PERSISTed files. If a PERSISTed file is read on a different size cluster than it was created on, optimizing the project- ed fields can mean that the distribution/sort order cannot be recreated.</td>
</tr>
<tr>
<td><strong>aggressiveOptimizeProjects</strong></td>
<td>false</td>
<td>If true, enables attempted minimization of network traffic for sorts/distributes. This option doesn't usually result in significant benefits, but may do so in some specific cases.</td>
</tr>
<tr>
<td><strong>percolateConstants</strong></td>
<td>true</td>
<td>If false, disables attempted aggressive constant value optimizations.</td>
</tr>
<tr>
<td><strong>exportDependencies</strong></td>
<td>false</td>
<td>Generate information about inter-definition dependencies.</td>
</tr>
<tr>
<td><strong>maxCompileThreads</strong></td>
<td>4 for eclccserv-er and 1 for ecclc</td>
<td>Number of compiler instances to compile the c++</td>
</tr>
<tr>
<td><strong>reportCppWarnings</strong></td>
<td>false</td>
<td>Report warnings from c++ compilation.</td>
</tr>
<tr>
<td><strong>saveCppTempFiles</strong></td>
<td>false</td>
<td>Retain the generated c++ files.</td>
</tr>
<tr>
<td><strong>spanMultipleCpp</strong></td>
<td>true</td>
<td>Generate a work unit in multiple c++ files.</td>
</tr>
</tbody>
</table>
ECL Language Reference
Template Language

**activitiesPerCpp**
Default 500
Number of activities in each c++ file (requires spanMultipleCpp)

**obfuscateOutput**
Default false
If true, details are removed from the generated workunit, including ECL code, estimates of record size, and number of records.

**The following options are for the workunit analyzer:**

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>analyzeWorkunit</td>
<td>true</td>
<td>If set to FALSE, disables analysis of the workunit</td>
</tr>
<tr>
<td>analyzer_minInterestingTime</td>
<td>1000</td>
<td>Analyze activities that exceed this minimum time to execute (milliseconds)</td>
</tr>
<tr>
<td>analyzer_minInterestingCost</td>
<td>30000</td>
<td>Report issues where the time penalty exceeds this value (milliseconds)</td>
</tr>
<tr>
<td>analyzer_skewThreshold</td>
<td>20</td>
<td>Report skew related issues that exceed this threshold</td>
</tr>
<tr>
<td>analyzer_minRowsPerNode</td>
<td>1000</td>
<td>Ignore activities that have this average number of rows per node</td>
</tr>
</tbody>
</table>

**Examples:**

```ecl
#OPTION('traceRowXml', TRUE);
#OPTION('_Probe', TRUE);
my_rec := RECORD
  STRING20 lname;
  STRING20 fname;
  STRING20 age;
END;
d := DATASET([{'PORTLY', 'STUART', '39'},
               {'PORTLY', 'STACIE', '36'},
               {'PORTLY', 'DARA', '1'},
               {'PORTLY', 'GARRETT', '4'}], my_rec);
OUTPUT(d(d.age > '1'), {lname, fname, age});
```

```ecl
//************************************
//This example demonstrates Logical Graphs and
//Javadoc-style comment blocks
#OPTION('generateLogicalGraphOnly',TRUE);
#OPTION('logicalGraphDisplayJavadocParameters',TRUE);
/**
 * Defines a record that contains information about a person
 */
namesRecord := RECORD
  string20 surname;
  string10 forename;
  integer2  age := 25;
END;
/**
 * Defines a table that can be used to read the information from the file
 * and then do something with it.
 */
```
namesTable := DATASET('x',namesRecord,FLAT);

/**
 * Allows the name table to be filtered.
 * @param ages        The ages that are allowed to be processed.
 * @param badForename Forname to avoid.
 * @return the filtered dataset.
 */
namesTable filtered(SET OF INTEGER2 ages, STRING badForename) :=
    namesTable(age in ages, forename != badForename);
OUTPUT(filtered([10,20,33], ''));
#SET

`#SET( symbol, expression );`

<table>
<thead>
<tr>
<th>symbol</th>
<th>The name of a previously declared user-defined symbol.</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>The expression whose value to assign to the symbol.</td>
</tr>
</tbody>
</table>

The `#SET` statement assigns the value of the `expression` to the `symbol`, overwriting any previous value the symbol had contained.

Example:

```ecl
#DECLARE(MySymbol); //declare a symbol named "MySymbol"
#SET(MySymbol,11);  //initialize MySymbol to 11
OUTPUT(%'MySymbol'%)
```

See Also: `#DECLARE`, `#APPEND`
#STORED

#STORED( storedname , value );

<table>
<thead>
<tr>
<th>storedname</th>
<th>A string constant containing the name of the stored attribute result.</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>An expression for the new value to assign to the stored attribute.</td>
</tr>
</tbody>
</table>

The #STORED statement assigns the value to the storedname, overwriting any previous value the stored attribute had contained. This statement may be used outside an XML scope and does not require a previous LOADXML to instantiate an XML scope.

Example:

```
PersonCount := 0 : STORED('myCount');
#STORED('myCount',100);
//change stored PersonCount attribute value to 100
OUTPUT(PersonCount);
```

See Also: STORED, #CONSTANT
The `#TEXT` statement returns the text of the specified `argument` to the MACRO. This statement may be used outside an XML scope and does not require a previous LOADXML to instantiate an XML scope.

Example:

```ECL
extractFields(ds, outDs, f1, f2='?') := MACRO
  #UNIQUENAME(r);
  %r% := RECORD
    f1 := ds.f1;
    #IF (#TEXT(f2)<>'?')
    #TEXT(f2)+':';
    f2 := ds.f2;
    #END
  END;
  outDs := TABLE(ds, %r%);
ENDMACRO;
extractFields(people, justSurname, lastname);
OUTPUT(justSurname);
ejectFields(people, justName, lastname, firstname);
OUTPUT(justName);
```

See Also: MACRO
## #UNIQUENAME

### #UNIQUENAME(namevar [,pattern]);

<table>
<thead>
<tr>
<th>namevar</th>
<th>The label of the template variable (without the percent signs) to use in subsequent statements (with the percent signs) that need the generated unique name.</th>
</tr>
</thead>
</table>
| pattern          | Optional. A template for unique name construction. It should contain a dollar sign ($) to indicate the position at which a unique number is generated, and may contain a pound sign (#) to include the namevar. This is useful for situations where #UNIQUENAME is being used to generate field names and the result is meant to be viewed in the ECL IDE program, since by default #UNIQUENAME generates identifiers that begin with a double underscore (__), and the ECL IDE treats them as hidden fields. If omitted, the default pattern is __#__$__.

The #UNIQUENAME statement creates a valid unique ECL identifier within the context of the current scope limit. This is particularly useful in MACRO structures as it allows the macro to be used multiple times in the same scope without creating duplicate attribute name errors from the attribute definitions within the macro. This statement may be used outside an XML scope and does not require a previous LOADXML to instantiate an XML scope.

**Example:**

```ecl
IMPORT Training_Compare;
EXPORT MAC_Compare_Result(module_name, attribute_name) := MACRO
  #UNIQUENAME(compare_file);
  %compare_file% := Training_Compare.File_Compare_Master;
  #UNIQUENAME(layout_per_attr);
  #UNIQUENAME(compare_attr, _MyField_$_);
  //the compare_attr fieldname is generated like: _MyField_1_
  %layout_per_attr% := RECORD
    person.per_cid;
    %compare_attr% := module_name.attribute_name;
  END;
  #UNIQUENAME(person_attr_out);
  %person_attr_out% := TABLE(person, %layout_per_attr%);

  #UNIQUENAME(person_attr_out_dist);
  %person_attr_out_dist% := DISTRIBUTE(%person_attr_out%, HASH(per_cid));

  #UNIQUENAME(layout_match_out);
  %layout_match_out% := RECORD
    data9 per_cid;
    boolean ValuesMatchFlag;
    TYPEOF(module_name.attribute_name) MyValue;
    TYPEOF(%compare_file%.attribute_name) CompareValue;
  END;
  #UNIQUENAME(layout_compare);
  %layout_compare% := RECORD
    %compare_file%.per_cid;
    %compare_file%.attribute_name;
  END;
  #UNIQUENAME(compare_table);
  %compare_table% := TABLE(%compare_file%, %layout_compare%);
  #UNIQUENAME(compare_table_dist);
  %compare_table_dist% := DISTRIBUTE(%compare_table%, HASH(per_cid));
  #UNIQUENAME(compare_attr_to_field);
```

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
%layout_match_out% %compare_attr_to_field%(%person_attr_out% L,%compare_table% R) := TRANSFORM
    SELF.ValuesMatchFlag := (L.%compare_attr% = R.attribute_name);
    SELF.MyValue := L.%compare_attr%;
    SELF.CompareValue := R.attribute_name;
    SELF := L;
END;

#UNIQUENAME(compare_out);
%compare_out% := JOIN(%person_attr_out_dist%,
    %compare_table_dist%,
    LEFT.per_cid = RIGHT.per_cid,
    %compare_attr_to_field%(LEFT, RIGHT),LOCAL);

#UNIQUENAME(match_out);
#UNIQUENAME(nomatch_out);
%match_out% := %compare_out%(ValuesMatchFlag);
%nomatch_out% := %compare_out%(~ValuesMatchFlag);

COUNT(%match_out%);
OUTPUT(CHOOSE(%match_out%, 50));
COUNT(%nomatch_out%);
OUTPUT(CHOOSE(%nomatch_out%, 50));
ENDMACRO;

See Also: MACRO
#WARNING

**#WARNING**

```plaintext
#WARNING(message);
```

| message | A string expression containing the warning message to display. |

The **#WARNING** statement displays the `message` in the workunit and/or syntax check. This statement may be used outside an XML scope and does not require a previous LOADXML to instantiate an XML scope.

Example:

```plaintext
a := TRUE; // pick one of these
//a := FALSE;
#IF(a)
  #ERROR('broken');
  OUTPUT('broken');
#ELSE
  #WARNING('maybe broken');
  OUTPUT('maybe broken');
#END;
```

See Also: **#ERROR**
#WEBSERVICE

#WEBSERVICE( [FIELDS(fieldlist),][HELP(helptext),][DESCRIPTION(descriptiontext),] );

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELDS</td>
<td>The FIELDS parameter specifies field sequence in WsECL Web forms. This is an exclusive list. If the FIELDS attribute is present, only the fields in the fieldslist are displayed on the Web form in WsECL.</td>
</tr>
<tr>
<td>fieldlist</td>
<td>A comma-separated list of field names in the order in which they should appear on the form. Use an asterisk (*) to include unspecified fields.</td>
</tr>
<tr>
<td>HELP</td>
<td>The HELP Parameter specifies to add help text to the WsECL Web form.</td>
</tr>
<tr>
<td>helptext</td>
<td>The help text to display.</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>The DESCRIPTION Parameter specifies to add descriptive text to the WsECL Web form.</td>
</tr>
<tr>
<td>descriptiontext</td>
<td>The description text to display.</td>
</tr>
</tbody>
</table>

The #WEBSERVICE statement sets options for the input parameters on a WsECL Web form for a published query.

Example:

```ecl
#WEBSERVICE(FIELDS('Field1','AddThem','Field2'),
    HELP('Enter Integer Values'),
    DESCRIPTION('If AddThem is TRUE, this adds the two integers'));
Field1 := 1 : Stored('Field1');
Field2 := 2 : Stored('Field2');
AddThem := TRUE : STORED('AddThem');
HiddenValue := 12 : STORED('HiddenValue'); //not in fieldlist, won't display on WsECL form
IF(AddThem,OUTPUT(Field1+Field2),OUTPUT('Not Added'));
#WEBSERVICE(FIELDS('field1','field2','*'));//includes unspecified fields on the WsECL form
```

See Also: STORED
#WORKUNIT

#WORKUNIT( option, value );

<table>
<thead>
<tr>
<th>option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A string constant specifying the name of the option to set.</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>The value to set for the option.</td>
</tr>
</tbody>
</table>

The #WORKUNIT statement sets the option to the specified value for the current workunit. This statement may be used outside an XML scope and does not require a previous call to the LOADXML function to instantiate an XML scope.

Valid option settings are:

<table>
<thead>
<tr>
<th>option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster</td>
<td>The value parameter is a string constant containing the name of the target cluster on which the workunit executes.</td>
</tr>
<tr>
<td>protect</td>
<td>The value parameter specifies true to indicate the workunit is protected from deletion, or false if not.</td>
</tr>
<tr>
<td>name</td>
<td>The value parameter is a string constant specifying the workunit's jobname.</td>
</tr>
<tr>
<td>priority</td>
<td>The value parameter is a string constant containing low, normal, or high to indicate the workunit's execution priority level, or an integer constant value (not a string) to specify how far above high the priority should be (&quot;super-high&quot;).</td>
</tr>
<tr>
<td>scope</td>
<td>The value parameter is a string constant containing the scope value to use to override the workunit's default scope (the user ID of the submitting person). This is a Workunit Security feature and requires a system which is LDAP-enabled.</td>
</tr>
</tbody>
</table>

Example:

```c
#WORKUNIT('cluster','400way');  //run the job on the 400-way target cluster
#WORKUNIT('protect',true);      //disallow deletion or archiving by Sasha
#WORKUNIT('name','My Job');     //name it "My Job"
#WORKUNIT('priority','high');   //run before other lower-priority jobs
#WORKUNIT('priority',10);       //run before other high-priority jobs
#WORKUNIT('scope','NewVal');    //override the default scope (on an LDAP enabled system)
```
External Services

SERVICE Structure

servicename := SERVICE [ : defaultkeywords [:TIME[(label)]]]

prototype : keywordlist;
END;

<table>
<thead>
<tr>
<th>servicename</th>
<th>The name of the service the SERVICE structure provides.</th>
</tr>
</thead>
<tbody>
<tr>
<td>defaultkeywords</td>
<td>Optional. A comma-delimited list of default keywords and their values shared by all prototypes in the external service.</td>
</tr>
<tr>
<td>TIME</td>
<td>Tracks timing of all function calls in the SERVICE structure and reports them back as metrics to the user.</td>
</tr>
<tr>
<td>label</td>
<td>Optional. A string constant containing the name to associate with the timer. If omitted, the default is used.</td>
</tr>
<tr>
<td>prototype</td>
<td>The ECL name and prototype of a specific function.</td>
</tr>
<tr>
<td>keywordlist</td>
<td>A comma-delimited list of keywords and their values that tell the ECL compiler how to access the external service.</td>
</tr>
</tbody>
</table>

The SERVICE structure makes it possible to create external services to extend the capabilities of ECL to perform any desired functionality. These external system services are implemented as exported functions in a .SO (Shared Object). An ECL system service .SO can contain one or more services and (possibly) a single .SO initialization routine.

Example:

```ecl
email := SERVICE :TIME('MyTime')
simpleSend( STRING address,
             STRING template,
             STRING subject) : LIBRARY='ecl2cw',
             INITFUNCTION='initEcl2Cw';
END;
MyAttr := COUNT(Trades): FAILURE(email.simpleSend('help@ln_risk.com',
                                            'FailTemplate',
                                            'COUNT failure'));
//An example of a SERVICE function returning a structured record
NameRecord := RECORD
    STRING5 title;
    STRING20 fname;
    STRING20 mname;
    STRING20 lname;
    STRING5 name_suffix;
    STRING3 name_score;
END;

LocalAddrCleanLib := SERVICE
NameRecord dt (CONST STRING name, CONST STRING server = 'x')
    : c,entrypoint='aclCleanPerson73',pure;
END;

MyRecord := RECORD
    UNSIGNED id;
    STRING uncleanedName;
```
ECL Language Reference
External Services

NameRecord Name;
END;
x := DATASET('x', MyRecord, THOR);

myRecord t(myRecord L) := TRANSFORM
    SELF.Name := LocalAddrCleanLib.dt(L.uncleanedName);
    SELF := L;
END;
y := PROJECT(x, t(LEFT));
OUTPUT(y);

//The following two examples define the same functions:
TestServices1 := SERVICE
    member(CONST STRING src)
        : holertl,library='test',entrypoint='member',ctxmethod;
    takesContext1(CONST STRING src)
        : holertl,library='test',entrypoint='takesContext1',context;
    takesContext2()
        : holertl,library='test',entrypoint='takesContext2',context;
    STRING takesContext3()
        : holertl,library='test',entrypoint='takesContext3',context;
END;

//this form demonstrates the use of default keywords
TestServices2 := SERVICE : holert,library='test'
    member(CONST STRING src) : entrypoint='member',ctxmethod;
    takesContext1(CONST STRING src) : entrypoint='takesContext1',context;
    takesContext2() : entrypoint='takesContext2',context;
    STRING takesContext3() : entrypoint='takesContext3',context;
END;

See Also: External Service Implementation, CONST
CONST

The `CONST` keyword specifies that the value passed as a parameter will always be treated as a constant. This is essentially a flag that allows the compiler to properly optimize its code when declaring external functions.

Example:

```ecl
STRING CatStrings(CONST STRING S1, CONST STRING S2) := S1 + S2;
```

See Also: Functions (Parameters Passing), SERVICE Structure
External Service Implementation

ECL external system services are implemented as exported functions in a .SO (Shared Object). An ECL system service .SO can contain one or more services and (possibly) a single .SO initialization routine. All system service libraries must be thread safe.

All exported functions in the .SO (hereafter referred to as "entry points") must adhere to certain calling and naming conventions. First, entry points must use the "C" naming convention. That is, function name decoration (like that used by C++) is not allowed.

Second, the storage class of __declspec(dllexport) and declaration type _cdecl need to be declared for Windows/Microsoft C++ applications. Typically, SERVICE_CALL is defined as __declspec(dllexport) and SERVICE_API is defined as _cdecl for Windows, and left as nulls for Linux. For example:

```c
Extern "C" _declspec(dllexport) unsigned _cdecl Countchars(const unsigned len, const char *string)
```

**Note:** The use of an external SERVICE may be restricted to signed modules. See Code Signing in the ECL Programmer’s Guide.

.SO Initialization

The following is an example prototype for an ECL (.SO) system service initialization routine:

```c
extern "C" void stdcall <functionName> (IEclWorkUnit *w);
```

The IEclWorkUnit is transparent to the application, and can be declared as Struct IEclWorkUnit; or simply referred to as a void *.

In addition, an initialization routine should retain a reference to its "Work Unit." Typically, a global variable is used to retain this value. For example:

```c
IEclWorkUnit *workUnit;
// global variable to hold the Work Unit reference
extern "C" void SERVICE_API myInitFunction (IEclWorkUnit *w)
{
    workUnit = w; // retain reference to "Work Unit"
}
```

Entry Points

Entry points have the same definition requirements as initialization routines. However, unlike initialization routines, entry points can return a value. Valid return types are listed below. The following is an example of an entry point:

```c
extern "C" __int64 SERVICE_API PrnLog(unsigned long len, const char *val)
{
}
```

SERVICE Structure - external

For each system service defined, a corresponding ECL function prototype must be declared (see SERVICE Structure).

```c
servicename := SERVICE
    functionname(parameter list) [: keyword = value];
```
External Services

For example:

e-mail := SERVICE
  simpleSend(STRING address, STRING template, STRING subject)
  : LIBRARY='ecl2cw', INITFUNCTION='initEcl2Cw';

END;

Keywords

This is the list of valid keywords for use in service function prototypes:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBRARY</td>
<td>Indicates the name of the .SO module an entry point is defined in.</td>
</tr>
<tr>
<td>ENTRYPOINT</td>
<td>Specifies a name for the entry point. By default, the name of the entry point is the function name.</td>
</tr>
<tr>
<td>INITFUNCTION</td>
<td>Specifies the name of the initialization routine defined in the module containing the entry point. Currently, the initialization function is called once.</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>Indicates the function prototype is in the specified include file, so the generated CPP must #include that file. If INCLUDE is not specified, the C++ prototype is generated from the ECL function definition.</td>
</tr>
<tr>
<td>C</td>
<td>Indicates the generated C++ prototype is enclosed within an extern &quot;C&quot; rather than just extern.</td>
</tr>
<tr>
<td>PURE</td>
<td>Indicates the function returns the same result every time you call it with the same parameters and has no side effects. This allows the optimizer to make more efficient calls to the function in some cases.</td>
</tr>
<tr>
<td>ONCE</td>
<td>Indicates the function has no side effects and is evaluated at query execution time, even if the parameters are constant. This allows the optimizer to make more efficient calls to the function in some cases.</td>
</tr>
<tr>
<td>FOLD</td>
<td>Specifies that the function is evaluated at compile time if all parameters are constants. Specifying FOLD to the SERVICE applys it to all function definitions in the service - in such cases NOFOLD may be useful to override this default for individual functions that are not suitable for constant folding.</td>
</tr>
<tr>
<td>NOFOLD</td>
<td>Specifies that the service is not suitable for constant folding.</td>
</tr>
<tr>
<td>ACTION</td>
<td>Indicates the function has side effects and requires the optimizer to not remove calls to the function.</td>
</tr>
<tr>
<td>CONTEXT</td>
<td>Internal use, only. Indicates an extra internal context parameter (ICodeContext *) is passed to the function. This must be the first function parameter.</td>
</tr>
<tr>
<td>GLOBALCONTEXT</td>
<td>Internal use, only. Same as CONTEXT, but there are restrictions on where the function can be used (for example, not in a TRANSFORM).</td>
</tr>
<tr>
<td>CTXMETHOD</td>
<td>Internal use, only. Indicates the function is actually a method of the internal code context.</td>
</tr>
<tr>
<td>TIME</td>
<td>Tracks timing of the external function call and reports them back as metrics to the user.</td>
</tr>
<tr>
<td>label</td>
<td>Optional. A string constant containing the name to associate with the timer. If omitted, the default is used.</td>
</tr>
</tbody>
</table>

Data Types

Please see ECL to C++ Mapping documentation for data type mapping.
Passing Set Parameters to a Service

Three types of set parameters are supported: INTEGER, REAL, and STRINGn.

**INTEGER**

If you want to sum up all the elements in a set of integers with an external function, to declare the function in the SERVICE structure:

```ecl
SetFuncLib := SERVICE
    INTEGER SumInt(SET OF INTEGER ss) :
        holertl,library='dab',entrypoint='rtlSumInt';
    END;
    x := 3 + 4.5;
    SetFuncLib.SumInt([x, 11.79]); // passed two REAL numbers - it works
```

To define the external function, in the header (.h) file:

```c
__int64 rtlSumInt(unsigned len, __int64 * a);
```

In the source code (.cpp) file:

```c
__int64 rtlSumInt(unsigned len, __int64 * a) {
    __int64 sum = 0;
    for(unsigned i = 0; i < len; i++) {
        sum += a[i];
    }
    return sum;
}
```

The first parameter contains the length of the set, and the second parameter is an int array that holds the elements of the set. **Note:** In declaring the function in ECL, you can also have sets of INTEGER4, INTEGER2 and INTEGER1, but you need to change the type of the C function parameter, too. The relationship is:

<table>
<thead>
<tr>
<th>INTEGER8</th>
<th>__int64</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER4</td>
<td>int</td>
</tr>
<tr>
<td>INTEGER2</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER1</td>
<td>char</td>
</tr>
</tbody>
</table>

**REAL**

If you want to sum up all the elements in a set of real numbers:

To declare the function in the SERVICE structure:

```ecl
SetFuncLib := SERVICE
    REAL8 SumReal(SET OF REAL8 ss) :
        holertl,library='dab',entrypoint='rtlSumReal';
    END;
    INTEGER r1 := 10;
    r2 := 20.345;
    SetFuncLib.SumReal([r1, r2]); // intentionally passed an integer to the real set, it works too.
```

To define the external function, in the header (.h) file:

```c
double rtlSumReal(unsigned len, double * a);
```

In the source code (.cpp) file:

```c
double rtlSumReal(unsigned len, double * a) {
    double sum = 0;
    for(unsigned i = 0; i < len; i++) {
        sum += a[i];
    }
    return sum;
}
```
The first parameter contains the length of the set, and the second parameter is an array that holds the elements of the set.

**Note:** You can also declare the function in ECL as set of REAL4, but you need to change the parameter of the C function to float.

**STRINGn**

If you want to calculate the sum of the lengths of all the strings in a set, with the trailing blanks trimmed off:

To declare the function in the SERVICE structure:

```ecl
SetFuncLib := SERVICE
    INTEGER SumCharLen(SET OF STRING20 ss) :
        holertl,library='dab',entrypoint='rtlSumCharLen';
END;
str1 := '1234567890'+'xxxx ';
str2 := 'abc';
SetFuncLib.SumCharLen([str1, str2]);
```

To define the external function, in the header (.h) file:

```c
__int64 rtlSumCharLen(unsigned len, char a[][20]);
```

In the source code (.cpp) file:

```c
__int64 rtlSumCharLen(unsigned len, char a[][20]) {
    __int64 sumtrimedlen = 0;
    for(unsigned i = 0; i < len; i++) {
        for(int j = 20-1; j >= 0; j--) {
            if(a[i][j] != ' ') {
                break;
            }
            a[i][j] = 0;
        }
        sumtrimedlen += j + 1;
    }
    return sumtrimedlen;
}
```

**Note:** In declaring the C function, we have two parameters for the set. The first parameter is the length of the set, the second parameter is char[][n] where n is the SAME as that in stringn. Eg., if the service is declared as "integer SumCharLen(set of string20)", then in the C function the parameter type must be char a[][20].

**Plugin Requirements**

Plugins require an exported function with the following signature under Windows:

```c
Extern "C" _declspec(dllexport) bool getECLPluginDefinition(ECLPluginDefinitionBlock *pb)
```

The function must fill the passed structure with correct information for the features of the plugin. The structure is defined as follows:

**Warning:** This function may be called without the plugin being loaded fully. It should not make any library calls or assume that dependent modules have been loaded or that it has been initialised. Specifically: "The
system does not call DllMain for process and thread initialization and termination. Also, the system does not load additional executable modules that are referenced by the specified module.

```c
Struct ECLPluginDefinitionBlock
{
    Size_t size;
    // size of passed structure - filled in by the calling function
    Unsigned magicVersion;
    // Filled in by .SO - must be PLUGIN_VERSION (1)
    Const char *moduleName;
    // Name of the module
    Const char *ECL;
    // ECL Service definition for non-HOLE applications
    Unsigned flags;
    // Type of plugin - for user plugin use 1
    Const char *version;
    // Text describing version of plugin - used in debugging
    Const char *description;
    // Text describing plugin
}
```

To initialize information in a plugin, use a global variable or class and it will be appropriately constructed/destructed when the plugin is loaded and unloaded.

## Deployment

External .SOS must be deployed to the /opt/HPCCSystems/plugins directory on each node of the target environment. If external data files are required, they should be either manually deployed to each node, or referenced from a network node (the latter requires hard-coding the address in the code for the .SO). Note that manually deployed files are not backed up with the standard SDS backup utilities.

## Constraints

The full set of data types is supported on the Data Refinery and Data Delivery Engines (Thor/Roxie/Doxie).

## An Example Service

The following code example depicts an ECL system service (.SO) called examplelib that contains one entry point (`stringfind`). This is a slightly modified version of the Find function found in the Str standard library. This version is designed to work in the Data Refinery supercomputer.

### ECL definitions

```ecl
EXPORT ExampleLib := SERVICE
    UNSIGNED4 StringFind(CONST STRING src,
        CONST STRING tofind,
        UNSIGNED4 instance )
    : c, pure, entrypoint='elStringFind';
END;
```

### .SO code module:

```c
//**************************************************
// hqlplugins.hpp : Defines standard values included
// in
// the plugin header file.
//**************************************************
```
#ifndef __HQLPLUGIN_INCL
#define __HQLPLUGIN_INCL

#define PLUGIN_VERSION 1
#define PLUGIN_IMPLICIT_MODULE 1
#define PLUGIN_MODEL_MODULE 2
#define PLUGIN_.SO_MODULE 4

struct ECLPluginDefinitionBlock
{
    size_t size;
    unsigned magicVersion;
    const char *moduleName;
    const char *ECL;
    const char *Hole;
    unsigned flags;
    const char *version;
    const char *description;
};

typedef bool (*EclPluginDefinition) (ECLPluginDefinitionBlock *);
#endif //__HQLPLUGIN_INCL

//****************************************************
// examplelib.hpp : Defines standard values included in
// the plugin code file.
//****************************************************
#ifndef EXAMPLELIB_INCL
#define EXAMPLELIB_INCL
#ifdef _WIN32
#define EXAMPLELIB_CALL __cdecl
#ifdef EXAMPLELIB_EXPORTS
#define EXAMPLELIB_API __declspec(dllexport)
#else
#define EXAMPLELIB_API __declspec(dllimport)
#endif
#else
#define EXAMPLELIB_CALL
#define EXAMPLELIB_API
#endif

#include "hqlplugins.hpp"

extern "C" {
EXAMPLELIB_API bool getECLPluginDefinition(ECLPluginDefinitionBlock *pb);
EXAMPLELIB_API void setPluginContext(IPluginContext * _ctx);
EXAMPLELIB_API unsigned EXAMPLELIB_CALL elStringFind(unsigned srcLen,
    const char * src, unsigned hitLen, const char * hit,
    unsigned instance);
}
#endif //EXAMPLELIB_INCL

//****************************************************
// examplelib.cpp : Defines the plugin code.
//****************************************************
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
#include "examplelib.hpp"

#define EXAMPLELIB_VERSION "EXAMPLELIB 1.0.00"

static const char * HoleDefinition = NULL;
static const char * EclDefinition =
"export ExampleLib := SERVICE
  string EchoString(const string src) : c, pure, fold, entrypoint='elEchoString';
"END;"

EXAMPLELIB_API bool getECLPluginDefinition(ECLPluginDefinitionBlock *pb)
{
  // Warning: This function may be called without the plugin being loaded fully.
  // It should not make any library calls or assume that dependent modules
  // have been loaded or that it has been initialised.
  //
  // Specifically: "The system does not call DllMain for process and thread
  // initialization and termination. Also, the system does not load
  // additional executable modules that are referenced by the specified module."

  if (pb->size != sizeof(ECLPluginDefinitionBlock))
    return false;

  pb->magicVersion = PLUGIN_VERSION;
  pb->version = EXAMPLELIB_VERSION " $Revision: 62376 $";
  pb->moduleName = "lib_examplelib";
  pb->ECL = EclDefinition;
  pb->Hole = HoleDefinition;
  pb->flags = PLUGIN_IMPLICIT_MODULE;
  pb->description = "ExampleLib example services library";
  return true;
}

namespace nsExamplelib {

  IPluginContext * parentCtx = NULL;
}
using namespace nsExamplelib;

EXAMPLELIB_API void setPluginContext(IPluginContext * _ctx) { parentCtx = _ctx; }

EXAMPLELIB_API unsigned EXAMPLELIB_CALL elStringFind(unsigned srcLen,
  const char * src, unsigned hitLen, const char * hit,
  unsigned instance)
{
  tgt = (char *)CTXMALLOC(parentCtx, srcLen);
  memcpy(tgt, src, srcLen);
  tgtLen = srcLen;
}
Symbols

#APPEND, 407
#BREAK, 424
#CONSTANT, 408
#DECLARE, 409
#DEMANGLE, 410
#ELSE, 420
#ELSEIF, 420
#END, 424
#ERROR, 411
#EXPAND, 412
#EXPORT, 413
#EXPORTXML, 416
#for, 418
#getdatatypE, 419
#IF, 420
#ifdefined, 421
#INMODULE, 423
#ISDEFINED, 422
#LOOP, 424
#MANGLE, 425
#ONWARNING, 426
#OPTION, 427
#SET, 437
#STORED, 438
#TEXT, 439
#UNIQUENAME, 440
#WARNING, 442
#WEBSERVICE, 443
#WORKUNIT, 171, 444
.ECL files, 28
.SO, 448
    __COMPRESSED__, 72
    __CONTAINERIZED__, 14
    __ECL_LEGACY_MODE__, 14
    __ECL_VERSION_MAJOR__, 14
    __ECL_VERSION_MINOR__, 14
    __ECL_VERSION__, 14
    __OS__, 14
    __PLATFORM__, 15
    __STAND_ALONE__, 14
    __TARGET_PLATFORM__, 14

A

ABS, 151
ABS function, 151
ACOS, 152
ACOS function, 152
Actions as Definitions, 32
Addition, 33

AFTER, 157
AGGREGATE, 153
AGGREGATE function, 153
ALL, 108, 193, 229, 247, 290, 297, 301, 388
ALL keyword, 108
ALLNODES, 156
ALLNODES function, 156
AND, 41, 62
AND NOT, 62
ANY, 25
APPLY, 157
APPLY function, 157
arguments, 23
arithmetic operators, 33
AS, 112
ASCII, 73, 158, 294
ASCII function, 158
ASIN, 159
ASIN function, 159
ASSERT, 160
ASSERT function, 160
ASSSTRING, 162
ASSSTRING function, 162
ATAN, 163
ATAN function, 163, 164
ATAN2, 164
ATAN2 function, 164
ATMOST, 247, 301
AVE, 165
AVE function, 165

B

BEFORE, 157
BEGINC++, 124
BEGINC++ Structure, 124
BEST, 193, 301, 370
BETWEEN, 41
Between Operator, 41
BIG_ENDIAN, 43
Binary values, 13
Bitshift Left, 34
Bitshift operators, 34
Bitshift Right, 34
Bitwise AND, 33
Bitwise Exclusive OR, 33
Bitwise NOT, 33
Bitwise operators, 33
Bitwise OR, 33
BLOB in INDEX, 63
Boolean, 19
BOOLEAN, 42, 97
Boolean AND, 35
Boolean Definition, 19, 22

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
| C | Boolean NOT, 35, 35  
  | Boolean OR, 35  
  | BOOLEAN value type, 42  
  | BUILD, 166  
  | BUILD action, 87, 89  

| C | CASE, 174, 301  
  | CASE function, 174  
  | casting operator, 57  
  | Casting Rules, 58  
  | CATCH, 175  
  | CATCH Function, 175  
  | Character Sets, 20  
  | CHECKPOINT, 391  
  | CHECKPOINT workflow service, 391  
  | Child Dataset, 80  
  | child dataset records, 80  
  | CHOOSE, 177  
  | CHOOSE function, 177  
  | CHOOSEN, 70, 80, 178  
  | CHOOSEN function, 178  
  | CHOOSESETS, 179  
  | CHOOSESETS function, 179  
  | CLUSTER, 167, 292, 293, 294, 296, 398  
  | CLUSTERSIZE, 180  
  | COMBINE, 181  
  | COMBINE function, 181  
  | comparison operator, 34, 379  
  | COMPRESSED, 72, 87, 292  
  | COMPRESSION, 167  
  | Concatenation, 39, 80  
  | CONST, 23, 160, 447  
  | CONST Function, 447  
  | Constant set, 19  
  | constant values, 19, 77  
  | constants, 12, 14  
  | CORRELATION, 184  
  | CORRELATION function, 184  
  | COS, 186  
  | COS function, 186  
  | COSH, 187  
  | COSH function, 187  
  | COUNT, 70, 80, 188, 262, 262, 340, 405  
  | COUNT function, 188  
  | COUNTER, 81, 121, 148, 198, 227, 252, 269, 316  
  | COVARIANCE, 190  
  | COVARIANCE function, 190  
  | CPU pinning, 430  
  | CRON, 192  
  | CRON function, 192  
  | CSV, 73, 76, 290, 293, 297, 307  
  | CSV Files, 73, 293  

| D | DATA, 50  
  | Data string, 12  
  | DATA value type, 50  
  | Dataset, 20  
  | DATASET, 70, 70, 73, 82, 93, 167, 349  
  | DATASET declaration, 92, 93  
  | DATASET parameter, 24  
  | DATASET parameters, 72  
  | DECIMAL, 45  
  | DECIMAL value type, 45  
  | Decimal values, 13  
  | DEDUP, 167, 193, 275, 277  
  | DEDUP function, 193, 195, 330  
  | DEFAULT, 63  
  | DEFINE, 196  
  | DEFINE function, 196  
  | Definition Name, 17  
  | Definition Operator, 17  
  | Definition Types, 19  
  | Definition Visibility, 28, 143  
  | Definitions as Actions, 32  
  | DENORMALIZE, 197  
  | DENORMALIZE function, 197  
  | DEPRECATED, 392  
  | DEPRECATED workflow service, 392  
  | DESCEND, 321, 322  
  | DICTIONARY, 85  
  | DICTIONARY parameter, 24  
  | DISTINCT statement in SQL, 194  
  | DISTRIBUTE, 167, 200  
  | DISTRIBUTE function, 200  
  | DISTRIBUTED, 87, 167, 203  
  | DISTRIBUTED function, 203  
  | DISTRIBUTION, 204  
  | DISTRIBUTION action, 386  
  | DISTRIBUTION function, 204  
  | Division, 33  
  | Division by zero, 33  
  | dot syntax, 30  
  | Dynamic Files, 93  

| E | EBCDIC, 73, 206, 294  
  | EBCDIC function, 206  
  | ECL, 11  
  | ECL definition, 17  
  | ECL IDE, 12  
  | ECL keywords, 17  
  | EMBED, 131  
  | EMBED Structure, 131  
  | ENCODING, 351  
  | ENCRYPT, 72, 73, 74, 75, 292, 293, 294, 296  

© 2023 HPCC Systems®. All rights reserved. Except where otherwise noted, ECL Language Reference content licensed under Creative Commons public license.
ENDDC++, 124
ENDEMBED, 131
ENDMACRO, 141
ENTH, 179, 207
ENTH function, 207
ENUM, 56
ENUM datatype, 56
Equivalence, 34, 102
Equivalence Comparison, 34
ERROR, 208
ERROR function, 208, 217
ESCAPE, 73
EVALUATE, 209
EVALUATE action, 209
EVALUATE function, 210
EVENT, 211
EVENT function, 211
EVENTEXTRA, 213
EVENTEXTRA function, 213
EVENTNAME, 212
EXCEPT, 109
EXCEPT keyword, 109
EXCLUSIVE, 179
EXISTS, 214
EXISTS function, 214
EXP, 216
EXP function, 216, 264
EXPIRE, 167, 255, 256, 292, 293, 294, 296, 398
Explicit Casting, 57
EXPORT, 28, 97, 110
EXPORTed, 30
EXPORTed Definitions, 30
Expression, 17
Expressions, 33
Expressions and Operators, 33
Expressions as Actions, 32
EXTEND, 290, 297
Extended PARSE, 303
Extended PARSE Examples, 303
External Service, 448
external system services, 448
FETCH, 220
FETCH function, 220
FEW, 153, 167, 178, 226, 247, 353, 364, 394, 402
field sequence, 443
File Scope, 92
FILEPOSITION, 167
Filters, 22
FIRST, 87, 167, 301
FLAT, 72
Flat Files, 292
floating point, 44
FOLD, 449
Foreign files, 92
FORMAT, 293
FORWARD, 143
forward reference, 11, 102, 196
FROM, 112
FROMJSON, 222
FROMJSON function, 222
FROMUNICODE, 223
FROMUNICODE function, 223
FROMXML, 224
FROMXML function, 224
FULL ONLY, 251
FULL OUTER, 251
FUNCTION, 133
FUNCTION Structure, 133
FUNCTIONMACRO, 136
FUNCTIONMACRO Structure, 136
Functions, 23
GETENV, 225
GETENV function, 225
GETISVALID, 97
GLOBAL, 226, 394
GLOBAL function, 226
GLOBAL workflow service, 226, 394
GRAPH, 227
GRAPH function, 227
Greater or Equal, 34
Greater Than, 34
GROUP, 111, 181, 197, 307, 330
Group, 229
GROUP function, 229, 378
GROUP keyword, 111, 381
GROUPED, 70, 82, 247
HASH, 193, 231, 247
HASH function, 231
HASH32 function, 232
HASH64 function, 232, 233
<table>
<thead>
<tr>
<th>Index</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>HASH64 function, 233</td>
<td>ISVALID, 243</td>
<td></td>
</tr>
<tr>
<td>HASHCRC, 234</td>
<td>ISVALID function, 233</td>
<td></td>
</tr>
<tr>
<td>HASHCRC function, 234</td>
<td>ITERATE, 244</td>
<td></td>
</tr>
<tr>
<td>HASHMD5, 235</td>
<td>ITERATE function, 244</td>
<td></td>
</tr>
<tr>
<td>HASHMD5 function, 235</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAVING, 236</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HAVING function, 236</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HEADING, 73, 293, 295, 296, 350</td>
<td></td>
<td></td>
</tr>
<tr>
<td>heapsort, 354</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hexadecimal, 12, 13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HPCC, 11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hthor, 354</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPCALL, 237</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPCALL Function, 237</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPCALL Options, 237</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPHEADER, 237, 351</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF, 239</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IF function, 239</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFBLOCK, 60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFF, 240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFF function, 240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implicit Casting, 57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implicit Dataset, 95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implicit Dataset Relationality, 95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPORT, 112</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPORT AS, 112</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPORT FROM, 112</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPORTed, 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN, 40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In Line Dataset, 77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN Operator, 40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>in-line a set of data, 77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>In-Line Dataset, 77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INCLUDE, 449</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEPENDENT, 395</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEPENDENT workflow service, 395</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEX, 87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEX declaration, 87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Indexing, 20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inline TRANSFORMs, 148, 148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INNER, 278</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertion sort, 354</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER, 13, 43, 97, 450</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer Division, 33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER value type, 43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERFACE, 138</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interface, 361</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERFACE Structure, 138</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERNAL, 260</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTORMAT, 242</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTFORMAT function, 242</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOIN, 246, 253</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOIN function, 246, 253</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOIN Set, 253</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOIN setofdatasets, 253</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Join Types, 278</td>
<td></td>
<td></td>
</tr>
<tr>
<td>joincondition, 114, 247</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOINED, 353</td>
<td></td>
<td></td>
</tr>
<tr>
<td>joinflags, 247</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JOINS FULL OUTER, 251</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSON, 75, 290, 296</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSON Files, 296</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEEP, 193, 247, 301</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYDIFF, 255</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYDIFF function, 255</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYED, 114, 165, 184, 188, 190, 214, 247, 262, 262, 279, 314, 362, 364, 381</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keyed JOIN, 250</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYED Keyword, 114</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYPATCH, 256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYPATCH action, 255</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYPATCH function, 256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYUNICODE, 258</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEYUNICODE function, 258</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Landing Zone files, 93</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAST, 179</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEFT, 116, 198, 283</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEFT ONLY, 251, 278</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEFT OUTER, 251, 278</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LENGTH, 70, 80, 259</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LENGTH function, 259</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Less or Equal, 34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Less Than, 34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIBRARY, 143, 260</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIBRARY function, 171, 260</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIKELY, 117</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIMIT, 247, 262</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIMIT function, 262</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINKCOUNTED, 70, 82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LITERAL, 350</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LITTLE_ENDIAN, 43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LN, 264</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LN function, 216, 264</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD, 97</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LOADXML, 265
LOADXML function, 265
LOCAL, 29, 153, 167, 181, 193, 197, 207, 220, 229, 244, 247, 267, 275, 311, 314, 330, 353, 364, 370
LOCAL function, 87, 267
LOCALE, 60
LOG, 268, 350
LOG function, 268
LOGICAL Filenames, 92
Logical graphs, 431
logical operators, 35, 62
LOOKUP, 247
LOOP, 269
LOOP function, 227, 269
loopbody, 269
loopcondition, 269
loopcount, 269
loopfilter, 269
LZW, 87, 167
MACRO, 141
MACRO Structure, 141
MANY, 153, 226, 247, 301, 364
MAP, 272
MAP function, 272
MATCHED, 104, 301
MATCHED ALL, 301
MATCHLENGTH, 104
MATCHPOSITION, 104
MATCHROW, 104
MATCHTEXT, 104
MATCHUNICODE, 104
MAX, 301, 348
MAX function, 273
MAXCOUNT, 63
MAXLENGTH, 60, 63, 73, 87, 87, 87, 97, 167, 300
memory exhausted, 250
MERGE, 167, 275, 350, 364
MERGE function, 275
MERGEJOIN, 277
MERGEJOIN function, 253, 277
MIN, 279, 301
MIN function, 279
MODULE, 143
MODULE Structure, 143
Modulus Division, 33
MOFN, 278
MULTIPLE, 398
Multiplication, 33
N
N-ary merge/join, 227
Name, 17
NAMED, 26, 204, 290, 297, 298
NAMED OUTPUT, 297
Named Output Dataset, 77
NAMESPACE, 350
Natural Language Parsing, 99
Nested child datasets, 95
NOCASE, 301, 324, 325, 326
NOFOLD, 286, 449
NOFOLD function, 286
NOLOCAL, 281
NOLOCAL function, 281
non-procedural language, 11
NONEMPTY, 282
NONEMPTY function, 282
NORMALIZE, 283
NORMALIZE function, 283
NOROOT, 74, 75, 167
NOSCAN, 301
NOSORT, 197, 247
Not Equal, 34
NOT MATCHED, 301
NOT MATCHED ONLY, 301
NOTHOR, 287
NOTHOR action, 287
NOTIFY, 288
NOTIFY function, 288
NOTRIM, 73, 350
NOXPATH, 290
O
ONCE, 396, 449
ONCE workflow service, 396
ONFAIL, 175, 237, 262, 350
ONWARNING, 397
ONWARNING workflow service, 397
Operators, 33
OPT, 72, 87, 114, 295, 296, 316
OR, 62
ORDERED, 289
ORDERED action, 289
OUTPUT, 290, 292, 293, 294, 296, 297, 298, 307
OUTPUT - CSV Files, 293
OUTPUT - JSON Files, 296
OUTPUT - NAMED Files, 297
OUTPUT - XML Files, 294
OUTPUT action, 290
OUTPUT Pipe Files, 297
OUTPUT Scalar Values, 298
OUTPUT Thor/Flat Files, 292
OUTPUT Workunit Files, 298
OVERWRITE, 167, 255, 256, 292, 293, 294, 296
ECL Language Reference

Index

P
PACKED, 60
packed decimal, 45, 45, 45
packed hexadecimal, 50
PARALLEL, 299, 314, 350
PARALLEL action, 298
PARALLEL function, 299
Parameter Passing, 23
parameters, 17
PARSE, 300, 301
PARSE Examples, 303
PARSE function, 104, 300
PARSE Text, 300
PARSE XML, 302
PARTITION LEFT, 247
PARTITION RIGHT, 247
Passing a DATASET parameter, 82
Passing Set Parameters, 450
PATTERN, 100
Perl regular expression, 324, 325, 326
PERSIST, 398
PERSIST workflow service, 398
PHYSICALLENGTH, 97
Pipe, 76
PIPE, 290, 297, 307
PIPE Files, 76
PIPE function, 77, 307
Pool memory exhausted, 250
POWER, 309
POWER function, 309
PREFETCH, 314, 314
PRELOAD, 72, 310
PRELOAD function, 310
PRIORITY, 400
PRIORITY workflow service, 400
PROCESS, 311
PROCESS function, 311
processor affinity, 430
PROJECT, 314, 316
PROJECT function, 314
PULL, 318
PULL function, 318
PURE, 449

Q
QSTRING, 47
QSTRING string constants, 12
QSTRING value type, 47
query library, 143, 167
quicksort, 354
QUOTE, 73, 294

R
RANDOM, 319
RANDOM function, 319
RANGE, 320
RANGE function, 320
RANK, 321
RANK function, 321
RANKED, 322
RANKED function, 322
REAL, 44, 450
REAL data type, 44
REALFORMAT, 323
REALFORMAT Function, 323
RECORD, 60
record matching, 277
Record Matching Logic, 277
Record Set, 19, 20, 22, 36
Record Set Definition, 20
Record Set Operators, 36
RECORD structure, 36, 60, 75, 76, 77, 96, 104, 106,
119, 147, 301, 301, 314, 364, 365, 381
RECORD Structure, 60
RECORDOF, 55
RECORDOF datatype, 55
RECOVERY, 401
RECOVERY workflow service, 401
restruct, 336
REFRESH, 398
regex, 324, 325, 326
REGEXFIND, 324
REGEXFIND function, 324
REGEXFINDSET, 325
REGEXFINDSET function, 325
REGEXREPLACE, 326
REGEXREPLACE function, 326
REGROUP, 327
REGROUP function, 327
regular expression, 100
REJECTED, 329
REJECTED function, 329, 385
Relationality, 95
REPEAT, 297, 307
Reserved Words, 17
resultrec, 336
RETRY, 237, 350
RETURN, 31, 133, 136
RIGHT, 116, 198
RIGHT ONLY, 251
RIGHT OUTER, 251
RIGHT1, 153
RIGHT2, 153
ROLLUP, 330
ROLLUP function, 330, 354
| ROUND, 334 |
| ROUND function, 334 |
| ROUNDUP, 335 |
| ROUNDUP function, 335 |
| ROW, 87, 167, 336 |
| ROW function, 145, 336 |
| ROWDIFF, 340 |
| ROWDIFF function, 340 |
| ROWS(LEFT), 118 |
| ROWS(RIGHT), 118 |
| ROWSET, 227 |
| ROWSET(LEFT), 227 |
| Roxie, 267, 354 |
| Roxie CPU pinning, 430 |
| Roxie processor affinity, 430 |
| RULE, 100 |
| runtime constants, 15 |
| runtime expressions, 15 |

| SAMPLE, 341 |
| SAMPLE function, 341 |
| Scalar OUTPUT, 298 |
| SCAN, 301 |
| SCAN ALL, 301 |
| Scope, 17 |
| SCOPE, 92 |
| SELF, 119, 146 |
| SELF keyword, 119 |
| SEPARATOR, 73, 293 |
| SEQUENTIAL, 343 |
| SEQUENTIAL function, 343 |
| Service Function Keywords, 449 |
| SERVICE Structure, 157, 445, 448 |
| SERVICE structure, 450, 451 |
| Set, 19 |
| SET, 23, 344 |
| Set Definition, 19, 19 |
| SET function, 19, 344 |
| SET OF, 53 |
| SET OF value type, 53 |
| Set Operators, 38 |
| Set Ordering, 20 |
| SET parameters, 23 |
| Set Parameters, 450 |
| Sets can contain definitions and expressions, 19 |
| SHARED, 28, 120 |
| Shared Object, 448 |
| SIN, 346 |
| SIN function, 346 |
| SINGLE, 293, 398 |
| Single Row Dataset, 79 |
| Single-Row Dataset, 79 |
| SINH, 347 |
| SINH function, 347 |
| SIZEOF, 348 |
| SIZEOF function, 348 |
| SKEW, 167, 200, 247, 353, 364 |
| SKIP, 121, 146, 175, 247, 262, 301 |
| SMART, 247 |
| SOAPACTION, 350 |
| SOAPCALL, 65, 349, 351 |
| SOAPCALL Action, 352 |
| SOAPCALL Function, 349, 351 |
| SOAPCALL Options, 350 |
| SORT, 353 |
| sort algorithms, 354 |
| SORT function, 353 |
| SORTED, 87, 167, 246, 253, 275, 277, 357 |
| SORTED function, 357 |
| SQRT, 358 |
| SQRT function, 358 |
| square brackets, 10, 19, 23, 53 |
| STABLE, 353 |
| STEPPED, 359 |
| STEPPED function, 359 |
| STORE, 97 |
| STORED, 361, 402 |
| STORED function, 361 |
| STORED workflow service, 402 |
| STREAMED, 70, 82 |
| String, 20 |
| STRING, 46 |
| string constants, 12 |
| string operator, 39 |
| string slice, 20 |
| STRING value type, 46 |
| STRINGn, 451 |
| substring, 20 |
| Subtraction, 33 |
| SUCCESS, 404 |
| SUCCESS workflow service, 404 |
| SUM, 362 |
| SUM function, 362 |
| SuperFile, 93 |
| system constants, 14 |

| TABLE, 30, 364 |
| TABLE function, 31 |
| TABLE Function, 364 |
| TAN, 366 |
| TAN Function, 366 |
| TANH, 367 |
| TANH Function, 367 |
| Template Language, 406 |
WILD, 114
WILD index filter, 114
WILD Keyword, 114
WORKUNIT, 70, 386
Workunit, 77
WORKUNIT Function, 386
Workunit OUTPUT, 298
WUID, 386

X
XML, 74, 76, 290, 294, 297, 300, 307
XML Files, 294
XMLDECODE, 387
XMLDECODE Function, 387
XMLDEFAULT, 63
XMLENCODE, 388
XMLENCODE Function, 388
XMLPROJECT, 106
XMLTEXT, 106
XMLUNICODE, 106
XOR Operator, 35
XPATH, 63, 237, 350
XPATH support, 64