HPCC Systems®

ECL Language Reference

Boca Raton Documentation Team

LexisNexis® Risk Solutions

ECL Language Reference

Boca Raton Documentation Team

We welcome your comments and feedback about this document via email to <docfeedback@hpccsystems.com> subject to the HPCC Contribution Agreement at: hpccsystems.com/contribution. Please include **Documentation Feedback** in the subject line and reference the document name, page numbers, and current Revision Number in the text of the message.

2014 Version 5.0.4-1

Introduction 8
Documentation Structure 8
Documentation Conventions 9
ECL Basics 10
Overview 10
Constants 11
Definitions 13
Basic Definition Types 14
Recordset Filtering 17
Function Definitions (Parameter Passing) 18
Definition Visibility 23
Field and Definition Qualification 25
Actions and Definitions 27
Expressions and Operators 28
Expressions and Operators 28
Logical Operators 30
Record Set Operators 31
Set Operators 32
String Operators 33
IN Operator 34
BETWEEN Operator 35
Value Types 36
BOOLEAN 36
INTEGER 37
REAL 38
DECIMAL 39
STRING 40
QSTRING 41
UNICODE 42
DATA 43
VARSTRING 44
VARUNICODE 45
SET OF 46
TYPEOF 47
RECORDOF 48
ENUM 49
Type Casting 50
Record Structures and Files 52
RECORD Structure 52
DATASET 61
DICTIONARY 75
INDEX 77
Scope and Logical Filenames 80
Implicit Dataset Relationality 82
Alien Data Types 83
TYPE Structure 83
TYPE Structure Special Functions 84
Parsing Support 86
Parsing Support 86
PARSE Pattern Value Types 87
NLP RECORD and TRANSFORM Functions 91
XML Parsing RECORD and TRANSFORM Functions 93
Reserved Keywords 95
ALL 95

EXCEPT ... 96
EXPORT ... 97
GROUP keyword ... 98
IMPORT ... 99
KEYED and WILD ... 100
LEFT and RIGHT ... 102
ROWS(LEFT) and ROWS(RIGHT) ... 103
SELF ... 104
SHARED ... 105
SKIP ... 106
TRUE and FALSE ... 107
Special Structures ... 108
BEGINC++ Structure ... 109
EMBED Structure ... 114
FUNCTION Structure ... 116
FUNCTIONMACRO Structure ... 119
INTERFACE Structure ... 121
MACRO Structure ... 123
MODULE Structure ... 125
TRANSFORM Structure ... 128
Built-in Functions and Actions ... 131
ABS ... 132
ACOS ... 133
AGGREGATE ... 134
ALLNODES ... 136
APPLY ... 137
ASCII ... 138
ASIN ... 139
ASSERT ... 140
ASSTRING ... 142
ATAN ... 143
ATAN2 ... 144
AVE ... 145
BUILD ... 146
CASE ... 151
CATCH ... 152
CHOOSE ... 153
CHOOSEN ... 154
CHOOSESETS ... 155
CLUSTERSIZE ... 156
COMBINE ... 157
CORRELATION ... 160
COS ... 162
COSH ... 163
COUNT ... 164
COVARIANCE ... 165
CRON ... 167
DEDUP ... 168
DEFINE ... 170
DENORMALIZE ... 171
DISTRIBUTE ... 174
DISTRIBUTED ... 176
DISTRIBUTION ... 177
EBCDIC ... 179

ENTH 180
ERROR 181
EVALUATE 182
EVENT 184
EVENTNAME 185
EVENTEXTRA 186
EXISTS 187
EXP 188
FAIL 189
FAILCODE 190
FAILMESSAGE 191
FETCH 192
FROMUNICODE 194
FROMXML 195
GETENV 196
GLOBAL 197
GRAPH 198
GROUP 199
HASH 200
HASH32 201
HASH64 202
HASHCRC 203
HASHMD5 204
HAVING 205
HTTPCALL 206
IF 207
IFF 208
IMPORT 209
INTFORMAT 210
ISVALID 211
ITERATE 212
JOIN 214
KEYDIFF 222
KEYPATCH 223
KEYUNICODE 224
LENGTH 225
LIBRARY 226
LIMIT 228
LN 229
LOADXML 230
LOCAL 232
LOG 233
LOOP 234
MAP 236
MAX 237
MERGE 238
MERGEJOIN 239
MIN 241
NOLOCAL 242
NONEMPTY 243
NORMALIZE 244
NOFOLD 246
NOTHOR 247
NOTIFY 248

OUTPUT 249
PARALLEL 256
PARSE 257
PIPE 263
POWER 265
PRELOAD 266
PROCESS 267
PROJECT 269
PULL 272
RANDOM 273
RANGE 274
RANK 275
RANKED 276
REALFORMAT 277
REGEXFIND 278
REGEXREPLACE 279
REGROUP 280
REJECTED 281
ROLLUP 282
ROUND 286
ROUNDUP 287
ROW 288
ROWDIFF 292
SAMPLE 293
SEQUENTIAL 294
SET 295
SIN 296
SINH 297
SIZEOF 298
SOAPCALL 299
SORT 303
SORTED 307
SQRT 308
STEPPED 309
STORED 310
SUM 311
TABLE 312
TAN 314
TANH 315
THISNODE 316
TOPN 317
TOUNICODE 318
TOXML 319
TRANSFER 320
TRIM 321
TRUNCATE 322
UNGROUP 323
UNICODEORDER 324
VARIANCE 325
WAIT 327
WHEN 328
WHICH 329
WORKUNIT 330
XMLDECODE 331

XMLENCODE 332
Workflow Services 333
Workflow Overview 334
CHECKPOINT 335
DEPRECATED 336
FAILURE 337
GLOBAL - Service 338
INDEPENDENT 339
ONWARNING 340
PERSIST 341
PRIORITY 343
RECOVERY 344
STORED - Workflow Service 345
SUCCESS 346
WHEN 347
Template Language 348
Template Language Overview 348
#APPEND 349
#CONSTANT 350
#DECLARE 351
#DEMANGLE 352
#ERROR 353
#EXPAND 354
#EXPORT 355
#EXPORTXML 358
#FOR 360
#GETDATATYPE 361
#IF 362
#INMODULE 363
#LOOP / #BREAK 364
#MANGLE 365
#ONWARNING 366
#OPTION 367
#SET 374
#STORED 375
#TEXT 376
#UNIQUENAME 377
#WARNING 379
#WORKUNIT 380
External Services 381
SERVICE Structure 381
CONST 383
External Service Implementation 384
A. Creative Commons License 391
Index 395

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:

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

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
```

ECL Basics

Overview

Enterprise **C**ontrol **L**anguage (ECL) has been designed specifically for huge data projects using the LexisNexis 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:

\t	tab
\n	new line
\r	carriage return
\nnn	3 octal digits (for any other character)
\uhhhh	lowercase "u" followed by 4 hexadecimal digits (for any other UNICODE-only character)

```
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 utf8-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'
```

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.

```
MyQString := Q'ABCD'; // same as: (QSTRING)'ABCD'
```

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, 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.

```
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
MyReal1 := 10.0;   // value of MyReal1 is the REAL value 10.0
MyReal2 := 1.0e1;  // value of MyReal2 is the REAL value 10.0
```

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 attribute 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 attribute or field names.

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:

```
Definition Type     Are Named
Boolean              Is...
Set Definition       Set...
Record Set           ...DatasetName
```

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:

```
StateCapitol(STRING2 state) :=
          CASE(state, 'FL' => 'Tallahassee', 'Unknown');
SetFloridaCities := ['Orlando', StateCapitol('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 datsets as input parameters.

```
SetDS := [ds1, ds2, ds3]; // a set of datasets
```

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 (except GetXLen):

```
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);
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'
                OR (~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**;**

ValueType	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.
AliasName	Names the parameter for use in the expression.
DefaultValue	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.
expression	The function's operation for which the parameters are used.

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).

```
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).

```
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:

*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.

```
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:

*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.

```
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
MyDCTfunc(DsDCT,'Brown');      //Seth
```

Passing Typeless Parameters

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

DefinitionName **(ANY** *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 *ValueType* for the parameter:

FunctionName(*parameters*)

PrototypeName

FunctionName	The name of a function, the type of which may be passed as a parameter.
parameters	The parameter definitions for the *FunctionName* parameter.
PrototypeName	The name of a previously defined function to use as the type of function that may be passed as a parameter.

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, witha default value:
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* **);**

NAMED	Optional. Required only when the *AliasName* clashes with a reserved word.
AliasName	The names of the parameter in the definition's function definition.
value	The value to pass to the parameter.

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 sam efolder 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 FUNCTIONMACRO 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).

```
IMPORT abc;                //make all exported definitions in the abc module available
EXPORT Definition1 := 5;    //make Definition1 available to other modules
Definition2 := abc.Definition2 + Definition1;  // object qualification needed for Definitions from abc module
```

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:

```
Young := YearOf(Person.per_dbrth) < 1950;
MySet := Person(Young);
```

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.

```
MySet1 := Person(YearOf(dbrth) < 1950);
// MySet1 is the set of Person records who were born before 1950
```

```
MySet2 := Person(EXISTS(OpenTrades(AgeOf(trd_dla) < AgeOf(Person.per_dbrth))));
```

```
// OpenTrades is a pre-defined record set.
//All Trades fields are in scope in the OpenTrades record set filter
//expression, but Person is required here to bring Person.per_dbrth
// into scope
//This example compares each trades' Date of Last Activity to the
// related person’s Date Of Birth
```

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:

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 can be treated as an Action. For example,

```
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

```
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:

```
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,

```
OUTPUT(Person);
```

is an action, but

```
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:

```
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 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:

Division	/
Integer Division	DIV
Modulus Division	%
Multiplication	*
Addition	+
Subtraction	-

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', ...);
```

The #option can take the following values:

'zero'	Evaluate to 0 - the default behaviour.
'fail'	Stop and report a division by zero error.
'nan'	This is only currently supported for real numbers. Division by zero creates a quiet NaN, which will propogate 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.

Bitwise Operators

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

Bitwise AND	&
Bitwise OR	|
Bitwise Exclusive OR	^
Bitwise NOT	BNOT

Bitshift Operators

Bitshift operators are supported for use in integer expressions:

Bitshift Right	>>
Bitshift Left	<<

Comparison Operators

The following comparison operators are supported:

Equivalence = returns TRUE or FALSE

Not Equal	<>	returns TRUE or FALSE
Not Equal	!=	returns TRUE or FALSE
Less Than	<	returns TRUE or FALSE
Greater Than	>	returns TRUE or FALSE
Less Than or Equal	<=	returns TRUE or FALSE
Greater Than or Equal	>=	returns TRUE or FALSE
Equivalence Comparison	<=>	returns -1, 0, or 1

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.

Logical Operators

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

NOT	Boolean NOT operation
~	Boolean NOT operation
AND	Boolean AND operation
OR	Boolean OR operation

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 is bypassed. 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:

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

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

```
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:

```
BOOLEAN XOR(BOOLEAN cond1, BOOLEAN cond2) :=
          (cond1 OR cond2) AND NOT (cond1 AND cond2);
```

Record Set Operators

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

+	Append all records from both files, independent of any order
&	Append all records from both files, maintaining record order on each node

Set Operators

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

+	Append (all elements from both sets, without re-ordering or duplicate element removal)

String Operators

The following string operator is supported:

+	Concatenation

IN Operator

value **IN** *value_set*

value	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.
value_set	A set of values. This may be a set expression, the SET function, or a DICTIONARY.

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.

Example:

```
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:
rec := {STRING color,UNSIGNED1 code};
ColorCodes := DATASET([{'Black' ,0 },
                       {'Brown' ,1 },
                       {'Red'   ,2 },
                       {'White' ,3 }], rec);

CodeColorDCT  := DICTIONARY(ColorCodes,{Code => Color});
OUTPUT(6 IN CodeColorDCT);     //true

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*

SeekVal	The value to find in the inclusive range.
LoVal	The low value in the inclusive range.
HiVal	The high value in the inclusive range.

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:
   // IsInNotRange := NOT (Z >= X AND Z <= Y);
```

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

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:

```
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

[*IntType***] [UNSIGNED] INTEGER[***n***]**

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

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

INTEGER Value Ranges

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

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

Type Significant Digits Largest Value Smallest Value

```
Type    Significant Digits     Largest Value   Smallest Value
REAL4    7 (9999999)           3.402823e+038   1.175494e-038
REAL8   15 (999999999999999)   1.797693e+308   2.225074e-308
```

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 (to a maximum of 32). If the *_y* value is present, the *y* defines the number of decimal places in the value.

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

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:

```
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. However, defining an EBCDIC STRING Attribute as a hexadecimal string constant value implies no conversion, as the programmer is assumed to have supplied the correct hexadecimal EBCDIC value.

Example:

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

EBCDIC STRING3 MyString1 := 'ABC';
        //implicit ASCII to EBCDIC conversion

EBCDIC STRING3 MyString2 := x'616263';
        //NO conversion here
```

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.

Example:

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

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.

Example:

```
UNICODE16 MyUNIString := U'1234567890ABCDEF';
        // utf-16-encoded string
UNICODE4 MyUnicodeString := U'abcd';
        // same as: (UNICODE)'abcd'
UNICODEde5 MyUnicodeString := U'abcd\353';
        // becomes 'abcdë' with a German locale
UNICODEde5 MyUnicodeString := U'abcdë';
        // same as previous example
```

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.

Example:

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

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.

Example:

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

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.

Example:

```
VARUNICODE16 MyUNIString := U'1234567890ABCDEF';
        // utf-16-encoded string
VARUNICODE4 MyUnicodeString := U'abcd';
        // same as: (UNICODE)'abcd'
VARUNICODE5 MyUnicodeString := U'abcd\353';
        // becomes 'abcdë'
VARUNICODE5 MyUnicodeString := U'abcdë';
        // same as previous example
```

SET OF

SET [OF *type* **]**

type	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.

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:

```
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];

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:

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

See Also: TRANSFORM Structure

RECORDOF

RECORDOF(*recordset* **)**

recordset	The set of data records whose RECORD structure to use. This may be a DATASET or any derived recordset.

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:

```
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

ENUM

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

type	The numeric value type of the *values*. If omitted, defaults to UNSIGNED4.
name	The label of the enumerated *value*.
value	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).

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

Example:

```
GenderEnum := ENUM(UNSIGNED1,Male,Female,Either,Unknown);
        //values are 1, 2, 3, 4

Pflg := 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;
  INTEGER2 age := 25;
END;

namesTable2 := DATASET([{'Foreman','George',GenderEnum.Male,Pflg.Foreign},
                        {'Bin Laden','Osama',GenderEnum.Male,Pflg.Foreign+Pflg.Terrorist+Pflg.Wanted}
                       ], namesRecord);
OUTPUT(namesTable2);

myModule(UNSIGNED4 baseError, STRING x) := MODULE
  EXPORT ErrCode := ENUM( ErrorBase = baseError,
                          ErrNoActiveTable,
                          ErrNoActiveSystem,
                          ErrFatal,
                          ErrLast);
  EXPORT reportX := FAIL(ErrCode.ErrNoActiveTable,'No ActiveTable in ' + x);
END;

myModule(100, 'Call1').reportX;
myModule(300, 'Call2').reportX;
```

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 in parentheses in the expression immediately preceding the element 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).

```
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
```

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)*(1.0*5)

evaluates as: (REAL)((INTEGER)1+(INTEGER)2+(INTEGER)3+(INTEGER)4)*(1.0*(REAL)5)

and: 5/2+4+5 evaluates as: (REAL)5/(REAL)2+(REAL)4+(REAL)5

while: '5' + 4 evaluates as: 5 + (STRING)4 //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*<).

```
INTEGER1 MyInt := 65; //MyInt is an integer value 65
STRING1 MyVal := (>STRING1<) MyInt; //MyVal is "A" (ASCII 65)
```

Casting Rules

From	To	Results in

INTEGER	STRING	ASCII or EBCDIC representation of the value
DECIMAL	STRING	ASCII or EBCDIC representation of the value, including decimal and sign
REAL	STRING	ASCII or EBCDIC representation of the value, including decimal and sign—may be expressed in scientific notation
UNICODE	STRING	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)
STRING	QSTRING	Uppercase ASCII representation
INTEGER	UNICODE	UNICODE representation of the value
DECIMAL	UNICODE	UNICODE representation of the value, including decimal and sign
REAL	UNICODE	UNICODE representation of the value, including decimal and sign—may be expressed in scientific notation
INTEGER	REAL	Value is cast with loss of precision when the value is greater than 15 significant digits
INTEGER	REAL4	Value is cast with loss of precision when the value is greater than 7 significant digits
STRING	REAL	Sign, integer, and decimal portion of the string value
DECIMAL	REAL	Value is cast with loss of precision when the value is greater than 15 significant digits
DECIMAL	REAL4	Value is cast with loss of precision when the value is greater than 7 significant digits
INTEGER	DECIMAL	Loss of precision if the DECIMAL is too small
REAL	DECIMAL	Loss of precision if the DECIMAL is too small
STRING	DECIMAL	Sign, integer, and decimal portion of the string value
STRING	INTEGER	Sign and integer portions of the string value
REAL	INTEGER	Integer value, only—decimal portion is truncated
DECIMAL	INTEGER	Integer value, only—decimal portion is truncated
INTEGER	BOOLEAN	0 = FALSE, anything else = TRUE
BOOLEAN	INTEGER	FALSE = 0, TRUE = 1
STRING	BOOLEAN	'' = FALSE, anything else = TRUE
BOOLEAN	STRING	FALSE = '', TRUE = '1'
DATA	STRING	Value is cast with no translation
STRING	DATA	Value is cast with no translation
DATA	UNICODE	Value is cast with no translation
UNICODE	DATA	Value is cast with no translation

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 **:= RECORD [(** *baserec* **)] [, MAXLENGTH(** *length* **)] [, LOCALE(** *locale* **)] [, PACKED]**

fields ;

[IFBLOCK(*condition* **)**

fields ;

END;]

[=> *payload* **]**

END;

attr	The name of the RECORD structure for later use in other definitions.
baserec	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).
MAXLENGTH	Optional. 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, a warning is generated. 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).
length	An integer constant specifying the maximum number of characters allowed.
LOCALE	Optional. Specifies the Unicode *locale* for any UNICODE fields.
locale	A string constant containing a valid locale code, as specified in ISO standards 639 and 3166.
PACKED	Optional. Specifies the order of the *fields* may be changed to improve efficiency (such as moving variable-length fields after the fixed-length fields)..
fields	Field declarations. See below for the appropriate syntaxes.
IFBLOCK	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. prepended to the fieldname to disambiguate the reference.
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 (,).

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.

Field Definitions

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

	datatype identifier ***[{***modifier***}]*** **[:=** *defaultvalue***] ;**
	identifier **:=** *defaultvalue* **;**
	defaultvalue **;**
	sourcefield **;**
	recstruct **[** *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.
defaultvalue	Optional. An expression defining the source of the data (for operations that require a data source, such as TABLE and PARSE). This may be a constant, expression, or definition providing the value.
modifier	Optional. One of the keywords listed in the **Field Modifiers**section below.
sourcefield	A previously defined data field, which implicitly provides the *datatype*, *identifier*, and *defaultvalue* for the new field—inherited from the *sourcefield*.
recstruct	A previously defined RECORD structure. See the **Field Inheritance**section below.
sourcedataset	A previously defined DATASET or derived recordset definition. See the **Field Inheritance**section below.
childdataset	A child dataset declaration (see DATASET and DICTIONARY discussions), which implicitly defines all the fields of the child at their already defined *datatype*, *identifier*, and *defaultvalue* (if present in the child dataset's RECORD structure).

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:

R1 **AND** *R2*	Intersection	All fields declared in both *R1* and *R2*
R1 **OR** *R2*	Union	All fields declared in either *R1* or *R2*
R1 **AND NOT** *R2*	Difference	All fields in *R1* that are not in *R2*
R1 **AND NOT** *F1*	Exception	All fields in *R1* except the specified field (*F1*)
R1 **AND NOT [***F1*, *F2***]**	Exception	All fields in *R1* except those in listed in the brackets (*F1*and*F2*)

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
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* **) }**
	{ MAXCOUNT(*records* **) }**

	{ XPATH('*tag*' **) }**
	{ XMLDEFAULT('*value*' **) }**
	{ DEFAULT(*value* **) }**
	{ VIRTUAL(fileposition) }
	{ VIRTUAL(localfileposition) }
	{ VIRTUAL(logicalfilename) }
	{ BLOB }

{ 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 *tag* that contains the data, in a RECORD structure that defines XML 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. This *value* will be used: 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 record passed to TRANSFORM functions in LEFT ONLY or RIGHT ONLY JOINs where there is no corresponding row. 4. When defaulting field values in a TRANSFORM using SELF = [ ].
{ 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] ...

node	Can contain wildcards.
qualifier	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.

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 DATASET 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;

PersonRec := RECORD
  STRING Uid{xpath('Person[@UID]')};
  NameRec Name{xpath('Name')};
```

```
    /*matches: <Name>
               <Fname>Kevin</Fname>
               <Mname>Alfonso</Mname>
               <Lname>Jones</Lname>
               </Name> */
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> */
```

This demonstrates "/Repeated":

```
DATASET(Names) Names{xpath('/name])};
          /*matches: <name>Gavin</name>
                     <name>Ricardo</name> */
```

"Container" and "Repeated" may also contain xpath filters, like this:

```
DATASET(doctorRec) doctors{xpath('person[@job=\'doctor\'])};
          /*matches: <person job='doctor'>
                       <FName>Kevin</FName>
                       <LName>Richards</LName>
                     </person> */
```

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 files using OUTPUT, the rules are similar with the following exceptions:

- For scalar fields, simple tag names and XML 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('RTTEMP::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
  MAX(GROUP,D1.F2); //no column name in output
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;
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,,'~RTTEST::test.xml', XML);

/* 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><lname> </lname>
<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')};
  STRING phone{XPATH('PHONE-NUMBER')};
```

```
  DATASET(cr) Fred{XPATH('PHONE-NUMBER-EXP')};
END;

DS := DATASET([{'1002','1352,9493',['1352','9493']},
            {'1003','4846,4582,0779',['4846','4582','0779']}],r);

OUTPUT(ds,,'~RTTEST::XMLtest2',
      XML('RECORD',
          HEADING('<?xml version="1.0" encoding="UTF-8"?><RECORDS>',
                  '</RECORDS>')));

/* this example produces XML output that looks like  this:
<?xml version="1.0" encoding="UTF-8"?>
   <RECORDS>
      <RECORD>
         <COMP-ID>1002</COMP-ID>
          <PHONE-NUMBER>1352,9493</PHONE-NUMBER>
          <PHONE-NUMBER-EXP>1352</PHONE-NUMBER-EXP>
          <PHONE-NUMBER-EXP>9493</PHONE-NUMBER-EXP>
       </RECORD>
       <RECORD>
         <COMP-ID>1003</COMP-ID>
          <PHONE-NUMBER>4846,4582,0779</PHONE-NUMBER>
          <PHONE-NUMBER-EXP>4846</PHONE-NUMBER-EXP>
          <PHONE-NUMBER-EXP>4582</PHONE-NUMBER-EXP>
          <PHONE-NUMBER-EXP>0779</PHONE-NUMBER-EXP>
       </RECORD>
     </RECORDS>
 */
```

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

DATASET

attr **:= DATASET(** *file, struct, filetype* **);**

attr **:= DATASET(** *dataset, file, filetype* **);**

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

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

DATASET(*row* **)**

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

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

DATASET(*dict* **)**

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

attr	The name of the DATASET for later use in other definitions.
file	A string constant containing the logical file name. See the *Scope & Logical Filenames* section for more on logical filenames.
struct	The RECORD structure defining the layout of the fields. This may use RECORDOF.
filetype	One of the following keywords, optionally followed by relevant options for that specific type of file: THOR /FLAT, CSV, XML, PIPE. Each of these is discussed in its own section, below.
dataset	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 := DATASET('filename',RECORDOF(anotherdataset), ... )`
WORKUNIT	Specifies the DATASET is the result of an OUTPUT with the NAMED option within the same or another workunit.
wuid	Optional. A string expression that specifies the workunit identifier of the job that produced the NAMED OUTPUT.
namedoutput	A string expression that specifies the name given in the NAMED option.
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.
recstruct	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.
row	A single data record. This may be a single-record passed parameter, or the ROW or PROJECT function that defines a 1-row dataset.
childstruct	The RECORD structure of the child records being defined. This may use the RECORDOF function.

COUNT	Optional. Specifies the number of child records attached to the parent (for use when interfacing to external file formats).
count	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*).
LENGTH	Optional. Specifies the *size* of the child records attached to the parent (for use when interfacing to external file formats).
size	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*).
CHOOSEN	Optional. Limits the number of child records attached to the parent. This implicitly uses the CHOOSEN function wherever the child dataset is read.
maxrecs	An expression defining the maximum number of child records for a single parent.
GROUPED	Specifies the DATASET being passed has been grouped using the GROUP function.
LINKCOUNTED	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.
STREAMED	Specifies the DATASET being returned is returned as a pointer to an IRowStream interface (see the eclhelper.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.
struct	The RECORD structure of the dataset field or parameter. This may use the RECORDOF function.
dict	The name of a DICTIONARY definition.
count	An integer expression specifying the number of records to create.
transform	The TRANSFORM function that will create the records. This may take an integer COUNTER parameter.
DISTRIBUTED	Optional. Specifies distributing the created records across all nodes of the cluster. If omitted, all records are created on node 1.
LOCAL	Optional. Specifies records are created on every node.

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, 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***)]);**

THOR	Specifies the *file* is in the Data Refinery (may optionally be specified as **FLAT**, which is synonymous with THOR in this context).
__COMPRESSED__	Optional. Specifies that the THOR *file* on another supercomputer cluster is compressed because it is a result of the PERSIST Workflow Service.
__GROUPED__	Specifies the DATASET has been grouped using the GROUP function.
OPT	Optional. Specifies that using dataset when the THOR *file* doesn't exist results in an empty recordset instead of an error condition.
UNSORTED	Optional. Specifies the THOR *file* is not sorted, as a hint to the optimizer.
PRELOAD	Optional. Specifies the *file* is left in memory after loading (valid only for Rapid Data Delivery Engine use).
nbr	Optional. An integer constant specifying how many indexes to create “on the fly” for speedier access to the dataset. If > 1000, specifies the amount of memory set aside for these indexes.
ENCRYPT	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.

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:

```
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('~Thor400::RTTEMP::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***)]);**

CSV	Specifies the *file* is a “comma separated values” ASCII file.
HEADING(*n*)	Optional. The number of header records in the *file*. If omitted, the default is zero (0).
SEPARATOR	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.
f_delimiters	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.
TERMINATOR	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.
r_delimiters	A single string constant, or set of string constants, that define the character(s) used as the record delimiter.
QUOTE	Optional. The string quote character used. If omitted, the default is a single quote ('\'') or the delimiter specified in the spray operation that put the file on disk.
characters	A single string constant, or set of string constants, that define the character(s) used as the string value delimiter.
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*. If omitted, the default is 4096.
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).
ENCRYPT	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.

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:

```
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.
ENCRYPT	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.

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:

```
/* 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
tag
END;
books := DATASET('MyFile',rform,XML('library/book'));
```

PIPE Files

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

PIPE	Specifies the *file*comes from the *command*program. This is a “read” pipe.
command	The name of the program to execute, which must output records in the *struct* format to standard output.
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.

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:

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

Ptbl := DATASET('~Thor50::RTTEMP::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 DATASETs** discussion).

Example:

```
//Inline DATASET using definition values
myrec := {REAL diff, INTEGER1 reason};
rms5008 := 10.0;
rms5009 := 11.0;
rms5010 := 12.0;
btable := 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;
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
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')]);
```

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

namesRecord := RECORD
  UNSIGNED     id;
  personRecord;
END;

namesTable := DATASET('RTTEST::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;

p1 := PROJECT(namesTable, t1(LEFT));
OUTPUT(p1);

//** 3rd way - using single-row dataset form and ROW function
addressRecord t2(namesRecord L) := TRANSFORM
  SELF.people := DATASET(ROW(L,personRecord));
  SELF.id := L.id;
  SELF := [];
END;

p2 := PROJECT(namesTable, t2(LEFT));
OUTPUT(p2);

//** 4th way - using in-line dataset form in an expression context
addressRecord t4(namesRecord l) := TRANSFORM
  SELF.people := PROJECT(DATASET([L], namesRecord), tc0(LEFT));
  SELF.id := L.id;
  SELF := [];
END;
p3 := PROJECT(namesTable, t4(LEFT));
OUTPUT(p3);
```

Child DATASETs

DATASET(*childstruct* **[, COUNT(** *count* **) | LENGTH(** *size* **)] [, CHOOSEN(** *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:

childstruct **fieldname [** SELF.*count* **]**

DATASET newname := 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, CHOOSEN, 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:

```
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'},
         {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:

```
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:

```
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);
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);
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 2 on node 2
   Rec 1 on node 3
   Rec 2 on node 3
*/
```

See Also: RECORD Structure, TRANSFORM Structure

DICTIONARY

attr **:= DICTIONARY(** *dataset*, *structure* **);**

DICTIONARY(*structure* **)**

attr	The name of the DICTIONARY for later use in other definitions.
dataset	The name of a DATASET or recordset from which to derive the DICTIONARY. This may be defined inline (similar to an inline DATASET).
structure	The RECORD structure (often defined inline) specifying the layout of the fields. The first field(s) are key fields, optionally followed the "results in" operator (=>) 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.

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] [,PRELOAD] [,OPT] [,COMPRESSED(LZW | ROW | FIRST)] [,DISTRIBUTED]) [,FILEPOSITION(*false*);**

attr **:= INDEX([** *baserecset,* **]** *keys, payload, indexfile* **[,SORTED] [,PRELOAD] [,OPT] [,COMPRESSED(LZW | ROW | FIRST)] [,DISTRIBUTED]) [,FILEPOSITION(false);**

attr **:= INDEX(***index,newindexfile***);**

attr	The name of the INDEX for later use in other attributes.
baserecset	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.
keys	The RECORD structure of the fields in the *indexfile* that contains key and file position information for referencing into the *baserecset*. Field names and types must match the *baserecset* fields (REAL and DECIMAL type fields are not supported). This may also contain additional fields not present in the *baserecset* (computed fields). If omitted, all fields in the *baserecset* are used.
payload	The RECORD structure of the *indexfile* that contains additional fields not used as keys. If the name of the *baserecset* is in the structure, it specifies "all other fields not already named in the *keys* parameter." This may contain fields not present in the *baserecordset* (computed fields). 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.
indexfile	A string constant containing the logical filename of the index. See the *Scope & Logical Filenames* section for more on logical filenames.
SORTED	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.
PRELOAD	Optional. Specifies that the *indexfile* is left in memory after loading (valid only for Data Delivery Engine use).
OPT	Optional. Specifies that using the index when the *indexfile* doesn't exist results in an empty recordset instead of an error condition.
COMPRESSED	Optional. Specifies the type of compression used. If omitted, the default is **LZW**, a variant of the Lempel-Ziv-Welch algorithm. Specifying **ROW** compresses index entries based on differences between contiguous rows (for use with fixed-length records, only), and is recommended for use in circumstances where speedier decompression time is more important than the amount of compression achieved. **FIRST** compresses common leading elements of the key (recommended only for timing comparison use).
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 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.
FILEPOSITION(false)	Optional. Prevents the implicit fileposition field from being created and will not treat a trailing integer field any differently from the rest of the payload.
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.

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.

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:

```
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;

PtblOut := OUTPUT(TABLE(Person,PtblRec),,'RTTEMP::TestFetch');

Ptbl := DATASET('RTTEMP::TestFetch',
          {PtblRec,UNSIGNED8 RecPtr {virtual(fileposition)}},
          FLAT);

AlphaInStateCity := INDEX(Ptbl,
                   {state,city,lname,fname,RecPtr},
                   'RTTEMPkey::TestFetch');
Bld := BUILDINDEX(AlphaInStateCity);
```

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 *baserecset*, 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.

Example:

```
Vehicles := DATASET('vehicles',
          {STRING2 st,STRING20 city,STRING20 lname,
          UNSIGNED8 fpos{virtual(fileposition)}},FLAT);

VehicleKey := INDEX(Vehicles,{st,city},{lname,fpos},'vkey::st.city');
BUILDINDEX(VehicleKey);
```

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
```

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

Scope and Logical Filenames

File Scope

The logical filenames used in DATASET and INDEX attribute definitions and the OUTPUT and BUILD (or BUILDINDEX) actions may begin with a scope name, indicated by a leading tilde (~), and may contain directories. The scope and directories are delimited by double colons (::). 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};
ds1 := DATASET('SomeDir::SomeFileOut1',RecStruct,THOR);
ds2 := 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.

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 superfiles).

The syntax looks like this:

'~foreign::<dali-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:

```
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:

```
    Household          Parent
       Person          Child of Household
         Accounts      Child of Person, Grandchild of Household
```

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

TypeName **:= TYPE**

functions;

END;

TypeName	The name of the TYPE structure.
functions	Function Attribute definitions. There are usually multiple *functions*.

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.

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

Example:

```
STRING4 Rev(STRING4 S) := S[4] + S[3] + S[2] + S[1];
EXPORT ReverseString4 := TYPE
        EXPORT STRING4 LOAD(STRING4 S) := Rev(S);
        EXPORT STRING4 STORE(STRING4 S) := Rev(S);
END;
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;
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

TYPE Structure Special Functions

LOAD

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

LogicalType	The value type of the resulting output of the function.
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.

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;*

PhysicalType	The value type of the resulting output of the function.
LogicalType	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.

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;*

type	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.

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:

```
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

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* **:=** *parsepattern***;**

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.
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 **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 *parsepattern* 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.
pattern1 pattern2	*Pattern1* followed by *pattern2*.

'*string*'	A fixed text *string*, which may contain escaped octal string control characters (for example, CtrlZ is '\032').
FIRST	Matches the start of the string to search. This is similar to the regular expression ^ token, which is not supported.
LAST	Matches the end of the string to search. This is similar to the regular expression $ token, which is not supported.
ANY	Matches any character.
REPEAT(*pattern*)	Repeat the *pattern* any number of times. The regular expression syntax *pattern** is supported as a shorthand for REPEAT(*pattern*).
REPEAT(*pattern*, *expression*)	Repeat the *pattern expression* times. The regular expression syntax *pattern**<count> is supported as a shorthand for REPEAT(*pattern,expression*), but the regular expression bounded repeats syntax *pattern*{*expression*} is not.
REPEAT(*pattern*, *low*, ANY [,MIN])	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.
REPEAT(*pattern*, *low*, *high*)	Repeat the *pattern* from *low* to *high* times. The regular expression bounded repeats syntax *pattern*{*low*,*high*} is not supported.
OPT(*pattern*)	An optional *pattern*. The regular expression syntax *pattern*? is supported as a shorthand for OPT(*pattern*).
pattern1 **OR** *pattern2*	Either *pattern1* or *pattern2*. The regular expression syntax *pattern1* | *pattern2* is supported as a shorthand for OR.
[*list-of-patterns*]	A comma-delimited list of alternative *patterns*, useful for string sets. This is the same as OR.
pattern1 **[NOT] IN** *pattern2*	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.
pattern1 **[NOT] BEFORE** *pattern2*	Check if the given *pattern2* does [not] follow *pattern1*. *Pattern2* is not consumed from the input.
pattern1 **[NOT] AFTER** *pattern2*	Check if the given *pattern2* does [not] precede *pattern1*. *Pattern2* does not consume any input. It must also be a fixed length.
pattern **LENGTH**(*range*)	Check whether the length of a *pattern* is in the *range*. *Range* can have the form <value>,<min>..<max>,<min>.. or ..<max> So "digit*3 NOT BEFORE digit" could be represented as "digit* LENGTH(3)." This is more efficient, and digit* can be defined as a token. "digit* LENGTH(4..6)" matches 4,5 and 6 digit sequences.
VALIDATE(*pattern*, *isValidExpression*)	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]='Q') is equivalent to alpha* = ANY*3 'Q' ANY* or more usefully: VALIDATE(alpha*,isSurnameService(MATCHTEXT));
VALIDATE(*pattern*, *isValidAsciiExpression*, *isValidUnicodeExpression*)	A two parameter variant. Use the first *isValidAsciiExpression* if the string being searched is ASCII; use the second if it is Unicode.
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 Alteratives x or y xy Concatenation of x and y. x* x*? Zero or more. Greedy and minimal versions. x+ x+? One or more. Greedy and minimal versions. x? x?? Zero or one. Greedy and minimal versions. x{m} 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 End 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:

```
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.

Example:

```
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
  STRING20 movie_role := MATCHTEXT(role/arb); //arb within role
  UNSIGNED1 cast_rank := (UNSIGNED)MATCHTEXT(m_rank/number);
END;

// This example demonstrates the use of productions in PARSE code
//(only supported in the tomita version of PARSE).
PATTERN ws := [' ','\t'];
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

XML Parsing RECORD and TRANSFORM Functions

The following functions are valid for use only in field definition expressions within a RECORD structure or TRANSFORM function that is used to define the result set from the PARSE function, or the input RECORD structure for a DATASET containing XML data.

XMLTEXT(*xmltag***)**

XMLTEXT returns the ASCII text from the *xmltag*.

XMLUNICODE(*xmltag***)**

XMLUNICODE returns the Unicode text from the *xmltag*.

XMLPROJECT(*xmltag*, *transform***)**

XMLPROJECT returns the text from the *xmltag* as a child dataset.

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

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:

```
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:

```
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*

VIRTUAL	Optional. Specifies the *definition* is VIRTUAL. Valid only inside a MODULE Structure.
definition	A valid definition.

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:

```
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:

```
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*;

module-selector-list	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.
folder	A folder or file name in the repository.
AS	Defines a local *alias* name for the *folder*, typically used to create shorter local names for easier typing.
alias	The short name to use instead of the *folder* name.
symbol-list	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.
FROM	Specifies the *folder* name in which the *symbol-list* resides.
language	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.

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

Example:

```
IMPORT $;                               //makes all definitions from the same folder available

IMPORT $, Std;                          //makes the standard library functions available, also

IMPORT MyModule;                        //makes available the definitions from MyModule folder

IMPORT $.^.MyOtherModule                //makes available the definitions from MyOtherModule folder,
                                        //which is located in the same container as the current folder

IMPORT $.^.^.SomeOtherModule            //makes available the definitions from SomeOtherModule folder,
                                        //which is located in the grandparent folder of current folder

IMPORT SomeFolder.SomeFile;             //make the specific file available

IMPORT SomeReallyLongFolderName AS SN;  //alias the long name as "SN"

IMPORT Def1,Def2 FROM Fred;             //makes Def1 and Def2 from Fred folder available, unqualified

IMPORT * FROM Fred;                     //makes everything from Fred available, unqualified

IMPORT Dev.Me.Project1;                 //makes the Dev/Me/Project1 folder available

IMPORT Python;                          //makes Python language code embeddable
```

See Also: EXPORT, SHARED, EMBED Structure, IMPORT function

KEYED and WILD

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

WILD(*field* **)**

expression	An INDEX filter condition.
OPT	Only generate An INDEX filter condition.
field	A single field in an INDEX.

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:

```
   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:

```
ds := DATASET('~local::rkc::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

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:

```
NormRec := RECORD
  STRING20 thename;
  STRING20 addr;
END;
NamesRec := RECORD
  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);
```

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:

```
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

SHARED

SHARED [VIRTUAL] *definition*

VIRTUAL	Optional. Specifies the *definition* is VIRTUAL. Valid only inside a MODULE Structure.
definition	A valid definition.

The **SHARED** 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 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:

```
//this code is contained in the GoodHouses.ecl file
BadPeople := Person(EXISTS(trades(EXISTS(phr(phr_rate > '4'))));
        //local only to the GoodHouses definition
SHARED 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;
  SHARED 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

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:

```
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:

```
BooleanTrue := TRUE;
Booleanfalse := FALSE;
```

See Also: BOOLEAN

Special Structures

BEGINC++ Structure

resulttype funcname **(** *parameterlist*) **:= BEGINC++**

code

ENDC++;

resulttype	The ECL return value type of the C++ function.
funcname	The ECL definition name of the function.
parameterlist	The parameters to pass to the C++ function.
code	The C++ function source code.

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).

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:

```
//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).

```
ECL                 C++ [Linux in brackets]
BOOOLEAN xyz        bool xyz
INTEGER1 xyz        signed char xyz
INTEGER2 xyz        signed short xyz
INTEGER4 xyz        signed int xyz
INTEGER8 xyz        signed __int64 xyz [ long long ]
UNSIGNED1 xyz       unsigned char xyz
UNSIGNED2 xyz       unsigned short xyz
UNSIGNED4 xyz       unsigned int xyz
UNSIGNED8 xyz       unsigned __int64 xyz [ unsigned long long xyz ]
REAL4 xyz           float xyz
REAL/REAL8 xyz      double xyz
DATA xyz            size32_t lenXyz, void * xyz
STRING xyz          size32_t lenXyz, char * xyz
VARSTRING xyz       char * xyz;
QSTRING xyz         size32_t lenXyz, char * xyz
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.

```
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');
```

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:

```
//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++)
       ((unsigned __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:
        unsigned id;
        unsigned idx;
        Linked<IEngineRowAllocator> allocator;
```

```
    };
    #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

resulttype funcname **(** *parameterlist* **) := EMBED(** *language* **)**

code

ENDEMBED;

resulttype funcname **(** *parameterlist* **) := EMBED(** *language, code* **);**

resulttype	The ECL return value type of the function.
funcname	The ECL definition name of the function.
parameterlist	The parameters to pass to the function.
language	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.
code	The source code to embed.

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.

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:

```
//First form: a structure
IMPORT Python;  //make Python language available

INTEGER addone(INTEGER p) := EMBED(Python)
# Python code that returns one more than the value passed to it
if p < 10:
  return p+1
else:
  return 0
ENDEMBED;

//Second form: a function
INTEGER addtwo(INTEGER p) := EMBED(Python, 'p+2');
```

See Also: BEGINC++ Structure, IMPORT, IMPORT function

FUNCTION Structure

[resulttype] *funcname* **(** *parameterlist*) **:= FUNCTION**

code

RETURN *retval*;

END;

resulttype	The return value type of the function. If omitted, the type is implicit from the *retval* expression.
funcname	The ECL attribute name of the function.
parameterlist	The parameters to pass to the *code*. These are available to all attributes defined in the FUNCTION's *code*.
code	The local attribute definitions that comprise the function. These may not be EXPORT or SHARED attributes, but may include actions (like OUTPUT).
RETURN	Specifies the function's return value expression—the *retval*.
retval	The value, expression, recordset, row (record), or action to return.

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:

```
EXPORT doProjectChild(parentRecord l,UNSIGNED idAdjust2) := FUNCTION
  newChildRecord copyChild(childRecord l) := TRANSFORM
     SELF.person_id := l.person_id + idAdjust2;
     SELF := l;
   END;

   RETURN PROJECT(CHOOSEN(l.children, numChildren),copyChild(LEFT));
END;
     //And called from
SELF.children := doProjectChild(l, 99);

//*********************************
EXPORT isAnyRateGE(STRING1 rate) := FUNCTION
   SetValidRates := ['0','1','2','3','4','5','6','7','8','9'];
   IsValidTradeRate := ValidDate(Trades.trd_drpt) AND
                       Trades.trd_rate >= rate AND
                       Trades.trd_rate IN SetValidRates;
   ValidPHR := Prev_rate(phr_grid_flag = TRUE,
                         phr_rate IN SetValidRates,
                         ValidDate(phr_date));
   IsPHRGridRate := EXISTS(ValidPHR(phr_rate >= rate,
                                    AgeOf(phr_date)<=24));
   IsMaxPHRRate := MAX(ValidPHR(AgeOf(phr_date) > 24),
                       Prev_rate.phr_rate) >= rate;
   RETURN IsValidTradeRate OR IsPHRGridRate OR IsMaxPHRRate;
END;

//*************************************************************
//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);

//*************************************************************
//a coordinated set of 3 examples

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('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) ;
  Sp1 := Std.Str.Find(InWords,' ',1);
  Sp2 := Std.Str.Find(InWords,' ',2);
  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..],
                        HasMiddle AND HasSuffix => InWords[Sp3+1..Sp4-1],
                        NOT HasMiddle AND NOT HasSuffix => InWords[Sp2+1..],
                        NOT HasMiddle AND HasSuffix => InWords[Sp2+1..Sp3-1],
                        '');
  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 := LocalAddrCleanLib.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 := LocalAddrCleanLib.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;

OperateOnNumbers(23,22).Multiplied;
OperateOnNumbers(23,22).Differenced;
OperateOnNumbers(23,22).Summed;
```

See Also: MODULE Structure, TRANSFORM Structure, WHEN

FUNCTIONMACRO Structure

[resulttype] *funcname* **(** *parameterlist*) **:= FUNCTIONMACRO**

code

RETURN *retval*;

ENDMACRO;

resulttype	The return value type of the function. If omitted, the type is implicit from the *retval* expression.
funcname	The ECL definition name of the function/macro.
parameterlist	A 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.
code	The local definitions that comprise the function. These may not be EXPORT or SHARED, but may include actions (like OUTPUT).
RETURN	Specifies the return value expression—the *retval*.
retval	The value, expression, recordset, row (record), or action to return.

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:

```
EXPORT Field_Population(infile,infield,compareval) := FUNCTIONMACRO
  c1 := COUNT(infile(infield=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;

ds1 := dataset([{'M'},{'M'},{'M'},{''},{''},{'M'},{''},{'M'},{'M'},{''}],{STRING1 Gender});
ds2 := dataset([{''},{'M'},{'M'},{''},{''},{'M'},{''},{''},{'M'},{''}],{STRING1 Gender});

OUTPUT(Field_Population(ds1,Gender,''));
OUTPUT(Field_Population(ds2,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;

numPlus;
AddOne(5);
AddTwo(8);
```

See Also: FUNCTION Structure, MACRO Structure

INTERFACE Structure

interfacename **[(** *parameters* **)] := INTERFACE [(** *inherit* **)]**

members;

END;

interfacename	The ECL definition name of the interface.
parameters	Optional. The input parameters to the interface.
inherit	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.
members	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.

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:

```
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;

//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 STRING120 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

[resulttype] *macroname* **(** *parameterlist* **) := MACRO**

tokenstream;

ENDMACRO;

resulttype	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).
macroname	The name of the function the MACRO structure defines.
parameterlist	A 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.
tokenstream	The Attribute definitions or Actions that the macro will perform.

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:

```
// 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;
    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

modulename **[(** *parameters* **)] := MODULE [(** *inherit* **)] [, VIRTUAL] [, LIBRARY(***interface***)] [, FORWARD]**

members;

END;

modulename	The ECL definition name of the module.
parameters	Optional. The parameters to make available to all the *definitions*.
inherit	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.
members	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*.
VIRTUAL	Optional. Specifies the MODULE defines an abstract interface whose *definitions* do not require values to be defined for them.
LIBRARY	Optional. Specifies the MODULE implements a query library *interface* definition.
interface	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*.
FORWARD	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.**

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 Repository module named MyModule and that it contains an EXPORT *member* named MyDefinition, you would reference that *definition* as:

```
MyModule.MyModuleStructure.MyDefinition
```

```
MyMod := MODULE
  SHARED x := 88;
  y := 42;
  EXPORT InMod := MODULE //nested MODULE
    EXPORT Val1 := x + 10;
    EXPORT Val2 := y + 10;
  END;
END;

MyMod.InMod.Val1;
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:

```
//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>');
  EXPORT File := WHEN(DATASET([{'x','y',22}],Layout),Act);
END;
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
- Datarow definitions (such as the ROW function)
- Single-valued and Boolean definitions

And may NOT export:

- Actions (like OUTPUT or BUILD)
- TRANSFORM functions
- Other MODULE structures
- MACRO definitions

Example:

```
EXPORT filterDataset(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;
filtered := filterDataset('Halliday', TRUE);
OUTPUT(filtered.included,,NAMED('Included'));
OUTPUT(filtered.excluded,,NAMED('Excluded'));

//same result, different coding style:
EXPORT filterDataset(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;
filtered := filterDataset(TRUE);
OUTPUT(filtered.included('Halliday'),,NAMED('Included'));
OUTPUT(filterDataset(true).excluded('Halliday'),,NAMED('Excluded'));


//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 concete 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
EXPORT MyModule := MODULE, FORWARD
```

```
  EXPORT INTEGER foo := bar;
  EXPORT INTEGER bar := 42;
END;

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* **)**

resulttype	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.
funcname	The name of the function the TRANSFORM structure defines.
parameterlist	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.
SKIP	Optional. Specifies the *condition* under which the TRANSFORM function operation is skipped.
condition	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.
locals	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.
SELF	Specifies the resulting output recordset from the TRANSFORM.
outfield	The name of a field in the *resulttype* structure.
transformation	An expression specifying how to produce the value for the *outfield*. This may include other TRANSFORM function operations (nested transforms).
assignments	A semi-colon delimited list of SELF.*outfield*:= *transformation* definitions.
datarow	A single record to transform, typically the keyword LEFT.

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 *resulttype* 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:

```
SELF := [ ];
```

clears all fields in the *resulttype* output that have not previously been defined in the transform function, while this form:

```
SELF.outfield := [];   //the outfield names a child DATASET in
                       // the resulttype RECORD Structure
```

clears only the child fields in the *outfield*, and this form:

```
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
```

ACOS

ACOS(*cosine***)**

cosine	The REAL cosine value for which to find the arccosine.
Return:	ACOS returns a single REAL value.

The **ACOS** function returns the arccosine (inverse) of the *cosine*, in radians.

Example:

```
ArcCosine := ACOS(CosineAngle);
```

See Also: COS, SIN, TAN, ASIN, ATAN, COSH, SINH, TANH

AGGREGATE

AGGREGATE(*recordset, resultrec,maintransform* **[** , *mergetransform* **(RIGHT1,RIGHT2)] [,** *groupingfields* **] [, LOCAL | FEW | MANY])**

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.
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.

```
inRecord := RECORD
  UNSIGNED box;
  STRING text{MAXLENGTH(10)};
END;
inTable := DATASET([{1,'Fred'},{1,'Freddy'},
                    {2,'Freddi'},{3,'Fredrik'},{1,'FredJon'}], inRecord);

//Example 1: Produce a list of box contents by concatenating a string:

outRecord1 := RECORD
  UNSIGNED box;
  STRING contents{MAXLENGTH(200)};
END;
outRecord1 t1(inRecord l, outRecord1 r) := TRANSFORM
  SELF.box := l.box;
  SELF.contents := r.contents + IF(r.contents <> '', ',', '') + l.text;
END;

outRecord1 t2(outRecord1 r1, outRecord1 r2) := TRANSFORM
  SELF.box := r1.box;
  SELF.contents := r1.contents + ',' + r2.contents;
END;
OUTPUT(AGGREGATE(inTable, outRecord1, t1(LEFT, RIGHT), t2(RIGHT1, RIGHT2), LEFT.box));

//This example could eliminate the merge transform if the SELF.contents expression in
//the t1 TRANSFORM were simpler, like this:
//     SELF.contents := r.contents + ',' + l.text;
//which would make the AGGREGATE function like this:
//   OUTPUT(AGGREGATE(inTable, outRecord1, t1(LEFT, RIGHT), LEFT.box));


//Example 2: A PIGMIX style grouping operation:
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***)**

operation	The name of an attribute or in-line code that results in a DATASET or INDEX.
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:

```
ds := ALLNODES(JOIN(SomeData,LOCAL(SomeIndex), LEFT.ID = RIGHT.ID));
```

See Also: THISNODE, LOCAL, NOLOCAL

APPLY

[*attrname* := **] APPLY(***dataset, actionlist* **[, BEFORE(** *actionlist* **)] [, AFTER(** *actionlist* **)])**

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.

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:

```
EXPORT x :=  SERVICE
  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

ASCII(*recordset***)**

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.
Return:	ASCII returns a set of records.

The **ASCII** function returns the *recordset* with all STRING fields translated from EBCDIC to ASCII.

Example:

```
AsciiRecs := ASCII(SomeEBCDICInput);
```

See Also: EBCDIC

ASIN

ASIN(*sine***)**

sine	The REAL sine value for which to find the arcsine.
Return:	ASIN returns a single REAL value.

The **ASIN** function returns the arcsine (inverse) of the *sine*, in radians.

Example:

```
ArcSine := ASIN(SineAngle);
```

See Also: ACOS, COS, SIN, TAN, ATAN, COSH, SINH, TANH

ASSERT

ASSERT(*condition* **[** , *message* **] [** , **FAIL] [** , **CONST])**

ASSERT(*recset*, *condition* **[** , *message* **] [** , **FAIL] [** , **CONST])**

ASSERT(*recset*, *assertlist* **)**

condition	The logical expression that should be always be true.
message	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.
FAIL	Optional. Specifies an exception is generated, immediately terminating the workunit.
CONST	Optional. Specifies the condition is evaluated during code generation.
recset	The set of records for which to check the condition against each record.
assertlist	A comma-delimited list of ASSERTs of the first form, used to check multiple conditions against each record in the *recset*.

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:

```
val1 := 1;
val2 := 1;
val3 := 2;
val4 := 2 : STORED('val4');
ASSERT(val1 = val2);
ASSERT(val1 = val2, 'Abc1');
ASSERT(val1 = val3);
ASSERT(val1 = val3, 'Abc2');
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***)**

bitmap	The value to treat as a string.
Return:	ASSTRING returns a single STRING value.

The **ASSTRING** function returns the *bitmap* as a string. This is equivalent to TRANSFER(*bitmap*,STRING*n*) 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)
 // 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***)**

tangent	The REAL tangent value for which to find the arctangent.
Return:	ATAN returns a single REAL value.

The **ATAN** function returns the arctangent (inverse) of the *tangent*, in radians.

Example:

```
ArcTangent := ATAN(TangentAngle);
```

See Also: ATAN2, ACOS, COS, ASIN, SIN, TAN, COSH, SINH, TANH

ATAN2

ATAN2(*y*, *x***)**

y	The REAL numerator value for the tangent.
x	The REAL denominator value for the tangent.
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:

```
ArcTangent := ATAN2(TangentNumerator, TangentDenominator);
```

See Also: ATAN, ACOS, COS, ASIN, SIN, TAN, COSH, SINH, TANH

AVE

AVE(*recordset, value* **[, KEYED])**

AVE(*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. This also may be the keyword GROUP to indicate averaging the field values in a group.
value	The expression to find the average value of.
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 average value of. This may also be a SET of values.

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:

```
AvgBal1 := AVE(Trades,Trades.trd_bal);
AvgVal2 := AVE(4,8,16,2,1); //returns 6.2
SetVals := [4,8,16,2,1];
AvgVal3 := AVE(SetVals);    //returns 6.2
```

See Also: MIN, MAX

BUILD

[attrname := **] BUILD(***baserecset***, [** *indexrec* **] ,** *indexfile* **[,** *options* **]);**

[*attrname* := **]BUILD(***baserecset***,** *keys***,** *payload***,** *indexfile* **[,** *options* **]);**

[*attrname* := **] BUILD(** *indexdef* **[,** *options* **]);**

BUILD(*library* **);**

attrname	Optional. The action name, which turns the action into an attribute definition, therefore not executed until the *attrname* is used as an action.
baserecset	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.
indexrec	Optional. The RECORD structure of the fields in the indexfile that contains key and file position information for referencing into the *baserecset*. Field names and types must match the *baserecset* fields (REAL and DECIMAL value type fields are not supported). This may also contain additional fields not present in the *baserecset* (computed fields). If omitted, all fields in the *baserecset* are used. The last field must be the name of an UNSIGNED8 field defined using the {virtual(filepposition)} function in the DATASET declaration of the *baserecset*.
keys	The RECORD structure of fields in the *indexfile* that contains key and file position information for referencing into the *baserecset*. Field names and types must match the *baserecset* fields (REAL and DECIMAL value type fields are not supported). This may also contain additional fields not present in the *baserecset*. If omitted, all fields in the *baserecset* are used.
payload	The RECORD structure of the *indexfile* that contains additional fields not used as keys . If the name of the *baserecset* is in the structure, it specifies "all other fields not already named in the keys parameter." This may contain fields not present in the *baserecset* (computed fields). These fields do not take up space in the non-leaf nodes of the index and cannot be referenced in a KEYED() filter clause
indexfile	A string constant containing the logical filename of the index to produce. See the Scope & Logical Filenames article for more on logical filenames.
options	Optional. One or more of the options listed below.
indexdef	The name of the INDEX attribute to build.
library	The name of a MODULE attribute with the LIBRARY option.

The first three 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 fourth 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(*target* **)] [, SORTED] [, DISTRIBUTE(** *key* **) [, MERGE]][, DATASET(** *basedataset* **)] [, OVERWRITE] [, UPDATE][,EXPIRE([days])][, FEW] [, FILEPOSITION(false)] [, LOCAL] [, NOROOT] [, DISTRIBUTED][, COMPRESSED(LZW | ROW | FIRST)] [, WIDTH(** *nodes* **)] [, DEDUP][,SKEW(limit[, target]) [, THRESHOLD(size)]]**

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 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.
SORTED	Specifies that the *baserecset* 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 *baserecset* 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 *baserecset* 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(false)	Optional. Prevents the implicit fileposition field from being created and will not treat a trailing integer field any differently from the rest of the payload.
days	Optional. The number of days from last file read after which the file may be automatically deleted. If omitted, the default is seven (7).
FEW	Specifies the *indexfile* 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).
indexdef	The name of an existing INDEX attribute definition that provides the *baserecset*, *indexrec*, and *indexfile* parameters to use.
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 function.
NOROOT	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.
DISTRIBUTED	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).
COMPRESSED	Specifies the type of compression used. If omitted, the default is LZW, a variant of the Lempel-Ziv-Welch algorithm. Specifying ROW compresses index entries based on differences between contiguous rows (for use with fixed-length records, only), and is recommended for use in circumstances where speedier decompression time is

	more important than the amount of compression achieved. FIRST compresses common leading elements of the key (recommended only for timing comparison use).
WIDTH	Specifies writing the *indexfile* 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).
nodes	The number of physical file parts to write. If set to one (1), this operates exactly the same as the FEW option, above.
DEDUP	Specifies that duplicate entries are eliminated from the INDEX.
SKEW	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.)
limit	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 / <number of slaves on cluster>).
target	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 / <number of slaves on cluster>).
THRESHOLD	Indicates the minimum size for a single part before the SKEW limit is enforced.
size	An integer value indicating the minimum number of bytes for a single part. Default is 1GB.

BUILD an Access Index

[*attrname* := **] 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:

```
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

[*attrname* := **] 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:

```
Vehicles := DATASET('vehicles',
     {STRING2 st,
      STRING20 city,
      STRING20 lname,
      UNSIGNED8 filepos{virtual(fileposition)}},
      FLAT);
BUILD(Vehicles,{st,city},{lname},'vkey::st.city');
 //build key into Vehicles dataset on state and city
 //payload the last name
```

BUILD from an INDEX Definition

[*attrname* := **] BUILD(** *indexdef* **[,** *options* **]);**

Form 3 creates an index file by using a previously defined INDEX definition.

Example:

```
nameKey := INDEX(mainTable,{surname,forename,filepos},'name.idx');
BUILD(nameKey); //gets all info from the INDEX definition
```

BUILD a Query Library

BUILD(*library* **);**

Form 4 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 #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);
```

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:

```
MyExp := 1+2;
MyChoice := CASE(MyExp, 1 => 9, 2 => 8, 3 => 7, 4 => 6, 5);
  // returns a value of 7 for the caseval of 3
MyRecSet := CASE(MyExp, 1 => Person(per_st = 'FL'),
    2 => Person(per_st = 'GA'),
    3 => Person(per_st = 'AL'),
    4 => Person(per_st = 'SC'),
    Person);
  // returns set of Alabama Persons for the caseval of 3
MyAction := CASE(MyExp, 1 => FAIL('Failed for reason 1'),
    2 => FAIL('Failed for reason 2'),
    3 => FAIL('Failed for reason 3'),
    4 => FAIL('Failed for reason 4'),    FAIL('Failed for unknown reason'));
  // for the caseval of 3, Fails for reason 3
```

See Also: MAP, CHOOSE, IF, REJECTED, WHICH

CATCH

result **:= CATCH(** *recset, action* **);**

result	The definition name for the resulting recordset.
recset	The recordset expression that, if it fails, causes the *action* to launch.
action	One of the three valid actions below.
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:

```
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, dude')), ONFAIL(FailTransform));

OUTPUT(recovered1);  //empty recordset
OUTPUT(recovered2);  //
OUTPUT(recovered3);  //
```

See Also: TRANSFORM Structure, FAIL, FAILCODE, FAILMESSAGE

CHOOSE

CHOOSE(*expression, value,… , value, elsevalue***)**

expression	An arithmetic expression that results in a positive integer and determines which value parameter to return.
value	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.
elsevalue	The value to return when the expression returns an out-of-range value. The last parameter is always the *elsevalue*.
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:

```
MyExp := 1+2;
MyChoice := CHOOSE(MyExp,9,8,7,6,5); // returns 7
MyChoice := CHOOSE(MyExp,1,2,3,4,5);  // returns 3
MyChoice := CHOOSE(MyExp,15,14,13,12,11);  // returns 13
WorstRate := CHOOSE(IntRate,1,2,3,4,5,6,6,6,6,0);
 // WorstRate receives 6 if the IntRate is 7, 8, or 9
```

See Also: CASE, IF, MAP

CHOOSEN

CHOOSEN(*recordset, n* **[**, *startpos* **] [, FEW])**

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.
n	The number of records to return. If zero (0), no records are returned, and if ALL or CHOOSEN:ALL, all records are returned. The CHOOSEN:ALL option is a constant that may be used in any expression.
startpos	Optional. The ordinal position in the recordset of the first record to return. If omitted, the default is one (1).
FEW	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.
Return:	CHOOSEN returns a set of records.

The **CHOOSEN** function (choose-*n*) returns the first *n* number of records, beginning with the record at the *startpos*, from the specified *recordset*.

Example:

```
AllRecs   := CHOOSEN(Person,ALL); // returns all recs from Person
FirstFive := CHOOSEN(Person,5);   // returns first 5 recs from Person
NextFive  := CHOOSEN(Person,5,6); // returns next 5 recs from Person
LimitRecs := CHOOSEN(Person,IF(MyLimit<>0,MyLimit,CHOOSEN:ALL));
```

See Also: SAMPLE, CHOOSESETS

CHOOSESETS

CHOOSESETS(*recset, condition => n* **[**, *o* **][** , **EXCLUSIVE | LAST | ENTH])**

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.
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 CHOOSEN 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: CHOOSEN, SAMPLE

CLUSTERSIZE

CLUSTERSIZE

Return:	CLUSTERSIZE returns a single INTEGER value.

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:

```
OUTPUT(CLUSTERSIZE)
```

COMBINE

COMBINE(*leftrecset, rightrecset* **[**, *transform* **][,LOCAL])**

COMBINE(*leftrecset, rightrecset,* **GROUP** , *transform* **[,LOCAL])**

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.
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:

```
inrec := RECORD
  UNSIGNED6 did;
END;
outrec := RECORD(inrec)
  STRING20  name;
  STRING10  ssn;
  UNSIGNED8  dob;
END;
ds := DATASET([1,2,3,4,5,6], inrec);
i1 := DATASET([{1, 'Kevin'}, {2, 'Richard'}, {5,'Nigel'}],
    { UNSIGNED6 did, STRING10 name });
i2 := DATASET([{3, '123462'}, {5, '1287234'}, {6,'007001002'}],
    { UNSIGNED6 did, STRING10 ssn });
i3 := DATASET([{1, 19700117}, {4, 19831212}, {6,20000101}],
    { UNSIGNED6 did, UNSIGNED8 dob});
j1 := JOIN(ds, i1, LEFT.did = RIGHT.did, LEFT OUTER, LOOKUP);
j2 := JOIN(ds, i2, LEFT.did = RIGHT.did, LEFT OUTER, LOOKUP);
j3 := JOIN(ds, i3, LEFT.did = RIGHT.did, LEFT OUTER, LOOKUP);
```

```
combined1 := COMBINE(j1,
                     j2,
              TRANSFORM(outRec,
      SELF := LEFT;
      SELF := RIGHT;
      SELF := []));
combined2 := COMBINE(combined1,
                     j3,
              TRANSFORM(outRec,
      SELF.dob := RIGHT.dob;
      SELF := LEFT));
```

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:

```
inrec := {UNSIGNED6 did};
outrec := RECORD(inrec)
  STRING20 name;
  UNSIGNED score;
END;
nameRec := RECORD
  STRING20 name;
END;

resultRec := RECORD(inrec)
  DATASET(nameRec) names;
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, '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.did = RIGHT.did,
  TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
  LEFT OUTER, MANY LOOKUP);
j2 := JOIN( dsg,
  i2,
  LEFT.did = RIGHT.did,
  TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
  LEFT OUTER,
  MANY LOOKUP);
j3 := JOIN( dsg,
  i3,
```

```
  LEFT.did = RIGHT.did,
  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)));
//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)));
```

See Also: GROUP, REGROUP

CORRELATION

CORRELATION(*recset, valuex, valuey* **[** *, expresssion***] [, KEYED])**

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.
Return:	CORRELATION returns a single REAL value.

The **CORRELATION** function returns the Pearson's Product Moment Correlation Coefficient between *valuex* and *valuey*.

Example:

```
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' });
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},
   {1.14416e+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: VARIANCE, COVARIANCE

COS

COS(*angle***)**

angle	The REAL radian value for which to find the cosine.
Return:	COS returns a single REAL value.

The **COS** function returns the cosine 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

Cosine45 := COS(Angle45);   //get cosine of the 45 degree angle
```

See Also: ACOS, SIN, TAN, ASIN, ATAN, COSH, SINH, TANH

COSH

COSH(*angle***)**

angle	The REAL radian value for which to find the hyperbolic cosine.
Return:	COSH returns a single REAL value.

The **COSH** function returns the hyperbolic cosine 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

HyperbolicCosine45 := COSH(Angle45);
                            //get hyperbolic cosine of the 45 degree angle
```

See Also: ACOS, SIN, TAN, ASIN, ATAN, COS, SINH, TANH

COUNT

COUNT(*recordset* **[** *, expression* **] [, KEYED])**

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.
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:

```
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];
Cnt2    := COUNT(SetVals); //returns 5
```

See Also: SUM, AVE, MIN, MAX, GROUP, TABLE

COVARIANCE

COVARIANCE(*recset, valuex, valuey* **[** *, expresssion***] [, KEYED])**

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.
Return:	COVARIANCE returns a single REAL value.

The **COVARIANCE** function returns the extent to which *valuex* and *valuey* co-vary.

Example:

```
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' });
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},
   {1.14416e+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: VARIANCE, CORRELATION

CRON

CRON(*time* **)**

time	A string expression containing a unix-standard cron time.
Return:	CRON defines a single timer event.

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 0 to 31.
month	An integer value for the month. Valid values are from 0 to 12.
dow	An integer value for the day of the week. Valid values are from 0 to 7 (where both 0 and 7 represent 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 0-6/3,18-23/3 will fire the timer every three hours between 6pm and 6am.

Example:

```
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

DEDUP(*recordset* **[,** *condition* **[, ALL[, HASH]] [, KEEP** *n* **] [**, *keeper* **]] [**, **LOCAL])**

recordset	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.
condition	Optional. A comma-delimited list of expressions or key fields in the recordset that defines "duplicate" 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.
ALL	Optional. Matches the condition against all records, not just adjacent records. This option may change the output order of the resulting records.
HASH	Optional. Specifies the ALL operation is performed using hash tables.
KEEP	Optional. Specifies keeping n number of duplicate records. If omitted, the default behavior is to KEEP 1. Not valid with the ALL option present.
n	The number of duplicate records to keep.
keeper	Optional. The keywords LEFT or RIGHT. LEFT (the default, if omitted) keeps the first record encountered and RIGHT keeps the last.
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:	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.*field*=RIGHT.*field*. 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.

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(Names,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* **)**

pattern	The name of a RULE parsing pattern.
symbol	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.
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:

```
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

DENORMALIZE(*parentrecset, childrecset, condition, transform* **[,LOCAL] [,NOSORT])**

DENORMALIZE(*parentrecset, childrecset, condition,* **GROUP**, *transform* **[,LOCAL] [,NOSORT])**

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.
Return:	DENORMALIZE returns a record set.

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.

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:

```
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'},
                      {'Anywhee','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:

```
NormRec := RECORD
  STRING20  thename;
  STRING20  addr;
END;
NamesRec := RECORD
  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);
```

NOSORT example:

```
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.Value1;
  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      Y         S
 3       C      A      X         W
 */
```

See Also: JOIN, TRANSFORM Structure, RECORD Structure, NORMALIZE

DISTRIBUTE

DISTRIBUTE(*recordset* **)**

DISTRIBUTE(*recordset, expression* **[, MERGE(** *sorts* **)])**

DISTRIBUTE(*recordset, index* **[**, *joincondition* **])**

DISTRIBUTE(*recordset,* **SKEW(** *maxskew* **[**, *skewlimit* **]))**

recordset	The set of records to distribute.
expression	An integer expression that specifies how to distribute the recordset, usually using one 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%).
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

DISTRIBUTE(*recordset, 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

DISTRIBUTE(*recordset, index* **[**, *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 *joncondition* is true will end up on the same node.

Skew-based DISTRIBUTE

DISTRIBUTE(*recordset*, **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:

MAX(ABS(AvgPartSize-PartSize[node])/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:

```
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

DISTRIBUTED

DISTRIBUTED(*recordset* **[**, *expression* **])**

recordset	The set of distributed records.
expression	Optional. An expression that specifies how the recordset is distributed.
Return:	DISTRIBUTED returns a set of records.

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:

```
MySet := DISTRIBUTED(Person,HASH32(Person.per_ssn));
       //all people with the same SSN are already on the same node
```

See Also: HASH32, DISTRIBUTE

DISTRIBUTION

DISTRIBUTION(*recordset* **[**, *fields* **] [**, **NAMED(** *name* **)])**

recordset	The set of records on which to run statistics.
fields	Optional. A comma-delimited list of fields in the recordset to which to limit the action. If omitted, all fields are included.
NAMED	Optional. Specifies the result name that appears in the workunit.
name	A string constant containing the result label.

The **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*.

The DECIMAL data type is not supported by this action. You can use a REAL data type instead.

Example:

```
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);

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>
 <Value count="1">Z</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;

out := PARSE(x, line, res, XML('XML/Field'));
out;
```

EBCDIC

EBCDIC(*recordset***)**

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.
Return:	EBCDIC returns a set of records

..

The **EBCDIC** function returns the *recordset* with all STRING fields translated from ASCII to EBCDIC.

Example:

```
EBCDICRecs := EBCDIC(SomeASCIIInput);
```

See Also: ASCII

ENTH

ENTH(*recordset, numerator* **[**, *denominator* **[**, *which* **]] [**, **LOCAL])**

recordset	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.
numerator	The number of records to return. The chosen records are evenly spaced from throughout the recordset.
denominator	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.
which	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.
LOCAL	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.
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:

```
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)

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
```

See Also: CHOOSEN, SAMPLE

ERROR

ERROR [(*errormessage* | *errorcode* **)] ;**

ERROR (*errorcode* , *errormessage* **) ;**

ERROR(*datatype* **[**, **[** *errorcode* **] [**, *errormessage* **]]) ;**

errormessage	Optional. A string constant containing the message to display.
errorcode	Optional. An integer constant containing the error number to display.
datatype	The value type or name of a RECORD structure. This may use the TYPEOF function.

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:

```
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

[attrname := **] EVALUATE(***expression***) ;**

[attrname := **] EVALUATE(***module* **[**, *defname* **]) ;**

attrname	Optional. The action name, which turns the action into a definition, therefore not executed until the *attrname* is used as an action.
expression	The function to call in an action context.
module	The module to evaluate.
defname	Optional. The name of a specific definition within the *module* to evaluate. If omitted, all definitions in the *module* are evaluated.

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:

```
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:

```
M := MODULE
  EXPORT a := 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* recordset. 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:

```
WorstCard := SORT(Cards,Std.Scoring);
MyValue   := EVALUATE(WorstCard[1],Std.Utilization);
 // WorstCard[1] uses indexing to get the first record
 // in the sort order, then evaluates that record
 // returning the Std.Utilization value

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,{per_last_name,per_first_name,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

IsValidInd := trades.trd_ind_code IN ['FM','RE'];
IsMortgage := IsValidInd OR trades.trd_rate = 'G';
SortedTrades := SORT(trades(ValidDate(trades.trd_dopn),isMortgage),
      trades.trd_dopn_mos);
CurrentRate := MAP(~EXISTS(SortedTrades) => ' ',
                   EVALUATE(SortedTrades[1], trades.trd_rate));

OUTPUT(person,{CurrentRate});
```

See Also: SORT

EVENT

EVENT(*event* , *subtype* **)**

event	A case-insensitive string constant naming the event to trap.
subtype	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.
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:

```
EventName := 'MyFileEvent';
FileName  := 'test::myfile';

IF (FileServices.FileExists(FileName),
 FileServices.DeleteLogicalFile(FileName));
 //deletes the file if it already exists

FileServices.MonitorLogicalFileName(EventName,FileName);
 //sets up monitoring and the event name
 //to fire when the file is found

OUTPUT('File Created') : WHEN(EVENT(EventName,'*'),COUNT(1));
 //this OUTPUT occurs only after the event has fired

afile := DATASET([{ 'A', '0'}], {STRING10 key,STRING10 val});
OUTPUT(afile,,FileName);
 //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));
```

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:

```
doMyService := FUNCTION
  OUTPUT('Did a Service for: ' + 'EVENTNAME=' + EVENTNAME);
  NOTIFY(EVENT('MyServiceComplete',
               '<Event><returnTo>FRED</returnTo></Event>'),
    EVENTEXTRA('returnTo'));
  RETURN EVENTEXTRA('returnTo');
END;

doMyService : WHEN('MyService');

// and a call
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:

```
doMyService := FUNCTION
  OUTPUT('Did a Service for: ' + 'EVENTNAME=' + EVENTNAME);
  NOTIFY(EVENT('MyServiceComplete',
               '<Event><returnTo>FRED</returnTo></Event>'),
    EVENTEXTRA('returnTo'));
  RETURN EVENTEXTRA('returnTo');
END;

doMyService : WHEN('MyService');

// and a call
NOTIFY('MyService',
       '<Event><returnTo>'+WORKUNIT+'</returnTo></Event>');
WAIT('MyServiceComplete');
OUTPUT('WORKUNIT DONE')
```

See Also: EVENT, EVENTNAME, CRON, WHEN, WAIT, NOTIFY

EXISTS

EXISTS(*recordset* **[, KEYED])**

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.
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:

```
MyBoolean := EXISTS(Publics(pub_type = 'B'));
TradesExistPersons := Person(EXISTS(Trades));
NoTradesPerson := Person(NOT EXISTS(Trades));

MinVal2 := EXISTS(4,8,16,2,1); //returns TRUE
SetVals := [4,8,16,2,1];
MinVal3 := EXISTS(SetVals);  //returns TRUE
NullSet := [];
MinVal3 := EXISTS(NullSet);  //returns FALSE
```

See Also: DEDUP, Record Filters

EXP

EXP(*n*)

n	The real number to evaluate.
Return:	EXP returns a single real value.

The **EXP** function returns the natural exponential value of the parameter (en). This is the opposite of the LN function.

Example:

```
MyPI := EXP(3.14159);
Interim := ROUND(1000 * (EXP(MyPI)/(1 + EXP(MyPI))));
```

See Also: LN, SQRT, POWER

FAIL

[attrname := **] FAIL [(** *errormessage* | *errorcode* **)] ;**

[*attrname* := **] FAIL(** *errorcode* , *errormessage* **) ;**

[*attrname* := **] FAIL(** *datatype* **[**, **[** *errorcode* **] [**, *errormessage* **]])** ;

attrname	Optional. The action name, which turns the action into an attribute definition, therefore not executed until the *attrname* is used as an action.
errormessage	Optional. A string constant containing the message to display.
errorcode	Optional. An integer constant containing the error number to display.
datatype	The value type, name of a RECORD structure, DATASET, or DICTIONARY to emulate.

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:

```
IF(header.version <> doxie.header_version_new,
   FAIL('Mismatch -- header.version vs. doxie.header_version_new.'));

FailedJob := FAIL('ouch, it broke');
sPeople   := SORT(Person,Person.per_first_name);
nUniques  := COUNT(DEDUP(sPeople,Person.per_first_name AND
                  Person.address))
          : FAILURE(FailedJob);
MyRecSet  := IF(EXISTS(Person),Person,
             FAIL(Person,99,'Person does not exist!!'));
```

See Also: FAILURE, ERROR

FAILCODE

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:

```
SPeople := SORT(Person,Person.per_first_name);
nUniques := COUNT(DEDUP(sPeople,Person.per_first_name AND
                Person.address))
:FAILURE(Email.simpleSend(SystemsPersonnel,
SystemsPersonel.email,FAILCODE));
```

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:

```
SPeople := SORT(Person,Person.per_first_name);
nUniques := COUNT(DEDUP(sPeople,Person.per_first_name ANDPerson.address))
:FAILURE(Email.simpleSend(SystemsPersonnel,
        SystemsPersonel.email,FAILMESSAGE));
```

See Also: RECOVERY, FAILCODE, SOAPCALL

FETCH

FETCH(*basedataset, index, position* **[**, *transform* **] [, LOCAL])**

basedataset	The base DATASET attribute to process. Filtering is not allowed.
index	The INDEX attribute that provides keyed access into the *basedataset*. This will typically have a filter expression.
position	An expression that provides the means of locating the correct record in the *basedataset* (usually the field within the index containing the fileposition value).
transform	The TRANSFORM function to call for each record fetched from the *basedataset*. If omitted, FETCH returns a set containing all fields from both the *basedataset* and index, with the second of any duplicate named fields removed.
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:	FETCH returns a record set.

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:

```
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;

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

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
```

See Also: TOUNICODE, UNICODEORDER

FROMXML

FROMXML(*record, xmlstring* **)**

record	The RECORD structure to produce. Each field must specify the XPATH to the data in the *xmlstring* that it should hold.
xmlstring	A string containing the XML to convert.
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:

```
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);
```

See Also: ROW, TOXML

GETENV

GETENV(*name* **[,** default **])**

name	A string constant containing the name of the environment variable.
default	Optional. A string constant containing the default value to use if the environment variable does not exist.
Return:	GETENV returns a STRING value.

The **GETENV** function returns the value of the *name*d 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(g1);
```

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:

```
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* **)**

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.
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.

Example:

```
namesRec := RECORD
  STRING20 lname;
  STRING10 fname;
  UNSIGNED2 age := 25;
  UNSIGNED2 ctr := 0;
END;
namesTable2 := DATASET([{'Flintstone','Fred',35},
    {'Flintstone','Wilma',33},
    {'Jetson','Georgie',10},
    {'Mr. T','Z-man'}], namesRec);

loopBody(SET OF DATASET(namesRec) ds, UNSIGNED4 c) :=
      PROJECT(ds[c-1],    //ds[0]=original input
     TRANSFORM(namesRec,
    SELF.age := LEFT.age+c; //c is graph COUNTER
    SELF.ctr := COUNTER;    //PROJECT's COUNTER
    SELF := LEFT));

g1 := GRAPH(namesTable2,10,loopBody(ROWSET(LEFT),COUNTER));

OUTPUT(g1);
```

See Also: LOOP, RANGE

GROUP

GROUP(*recordset* **[**, *breakcriteria* **[**, **ALL]] [**, **LOCAL])**

recordset	The set of records to fragment.
breakcriteria	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.
ALL	Optional. Indicates the *breakcriteria* is applied without regard to any previous order. If omitted, GROUP assumes the recordset is already sorted in *breakcriteria* order.
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:	GROUP returns a record set.

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 maximum size allowed for any one subgroup is 64 Mb and subgroups never span nodes; if the *breakcriteria* results in a subgroup larger than 64 Mb, an error occurs.

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:

```
MyRec := RECORD
 STRING20 Last;
 STRING20 First;
END;
SortedSet := SORT(Person,Person.last_name); //sort by last name
GroupedSet := GROUP(SortedSet,last_name);  //then group them

SecondSort := SORT(GroupedSet,Person.first_name);
   //sorts by first name within each last name group
   // this is a "sort within group"

UnGroupedSet := GROUP(GroupedSet); //ungroup the dataset
MyTable := TABLE(SecondSort,MyRec); //create table of sorted names
```

See Also: REGROUP, COMBINE, UNGROUP, EXCEPT

HASH

HASH(*expressionlist***)**

expressionlist	A comma-delimited list of values.
Return:	HASH returns a single value.

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:

```
MySet := DISTRIBUTE(Person,HASH(Person.per_ssn));
    //people with the same SSN go to same Data Refinery node
```

See Also: DISTRIBUTE, HASH32, HASH64, HASHCRC, HASHMD5

HASH32

HASH32(*expressionlist***)**

expressionlist	A comma-delimited list of values.
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:

```
MySet := DISTRIBUTE(Person,HASH32(Person.per_ssn));
    //people with the same SSN go to same Data Refinery node
```

See Also: DISTRIBUTE, HASH, HASH64, HASHCRC, HASHMD5

HASH64

HASH64(*expressionlist***)**

expressionlist	A comma-delimited list of values.
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:

```
OUTPUT(Person,{per_ssn,HASH64(per_ssn)});
    //output SSN and its 64-bit hash value
```

See Also: DISTRIBUTE, HASH, HASH32, HASHCRC, HASHMD5

HASHCRC

HASHCRC(*expressionlist***)**

expressionlist	A comma-delimited list of values.
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:

```
OUTPUT(Person,{per_ssn,HASHCRC(per_ssn)});
    //output SSN and its CRC hash value
```

See Also: DISTRIBUTE, HASH, HASH32, HASH64, HASHMD5

HASHMD5

HASHMD5(*expressionlist***)**

expressionlist	A comma-delimited list of values.
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:

```
OUTPUT(Person,{per_ssn,HASHMD5(per_ssn)});
    //output SSN and its 128-bit hash value
```

See Also: DISTRIBUTE, HASH, HASH32, HASH64, HASHCRC

HAVING

HAVING(*groupdataset, expression* **)**

groupdataset	The name of a GROUPed record set.
expression	The logical expression by which to filter the groups.
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:

```
MyGroups := GROUP(SORT(Person,lastname),lastname);
     //group by last name
Filtered := HAVING(MyGroups,COUNT(ROWS(LEFT)) > 10);
     //filter out the small groups
```

See Also: GROUP

HTTPCALL

result **:= HTTPCALL(** *url, httpmethod, responsemimetype, outstructure* **[**, *options* **]);**

result	The definition name for the resulting recordset.
url	A string containing the URL that hosts the service to invoke. This may contain parameters to the service.
httpmethod	A string containing the HTTP Method to invoke. Valid methods are: "GET"
responsemimetype	A string containing the Response MIME type to use. Valid types are: "text/xml"
outstructure	A RECORD structure containing the output field definitions. For an XML-based *responsemimetype* these should use XPATH to specify the exact data path.
options	A comma-delimited list of optional specifications from the list below.

HTTPCALL is a function that calls a REST service.

Valid *options* are:

RETRY(*count*)	Specifies re-attempting the call count number of times if non-fatal errors occur. If omitted, the default is three (3).
TIMEOUT(*period*)	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).
TIMELIMIT(*period*)	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.
XPATH(*xpath*)	Specifies the path used to access rows in the output. If omitted, the default is: 'serviceResponse/Results/Result/Dataset/Row'.
ONFAIL(*transform*)	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.
TRIM	Specifies all trailing spaces are removed from strings before output.

Example:

```
worldBankSource := RECORD
  STRING name {XPATH('name')}
END;

OutRec1 := RECORD
  DATASET(worldBankSource) Fred{XPATH('/source')};
END;

raw := HTTPCALL('http://api.worldbank.org/sources', 'GET', 'text/xml', OutRec1);

OUTPUT(raw);
```

IF

IF(*expression, trueresult* **[**, *falseresult* **])**

expression	A conditional expression.
trueresult	The result to return when the expression is true. This may be any expression or action.
falseresult	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.
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:

```
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* **])**

expression	A conditional expression.
trueresult	The result to return when the expression is true. This may be any expression or action.
falseresult	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.
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.

See Also: IF, MAP, EVALUATE, CASE, CHOOSE, SET

IMPORT

resulttype funcname **(** *parameterlist* **) := IMPORT(** *language, function* **);**

resulttype	The ECL return value type of the *function*.
funcname	The ECL definition name of the *function*.
parameterlist	The parameters to pass to the *function*.
language	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.
function	A string constant containing the name of the function to include.

The **IMPORT** declaration allows you to call existing code written in the external *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.

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 Python;

INTEGER addthree(INTEGER p) := IMPORT(Python, 'python_mod_name.addThree');

//Java Example setting the classpath
IMPORT java;
STRING jcat(STRING a, STRING b) := IMPORT(java, 'JavaCat.cat:(Ljava/lang/String;Ljava/lang/String;)Ljava/lang
                                        : classpath('/opt/HPCCSystems/classes/'));
jcat('I',' concatenate');
```

See Also: IMPORT, EMBED Structure

INTFORMAT

INTFORMAT(*expression, width, mode***)**

expression	The expression that specifies the integer value to format.
width	The size of string in which to right-justify the value.
mode	The format type: 0 = leading blank fill, 1 = leading zero fill.
Return:	INTFORMAT returns a single value.

The **INTFORMAT** function returns the value of the *expression* formatted as a right-justified string of *width* characters.

Example:

```
val := 123456789;
OUTPUT(INTFORMAT(val,20,1));
  //formats as '00000000000123456789'
OUTPUT(INTFORMAT(val,20,0));
  //formats as '          123456789'
```

See Also: REALFORMAT

ISVALID

ISVALID(*field* **)**

field	The name of a DECIMAL, REAL, or alien data TYPE field.
Return:	ISVALID returns a single Boolean value.

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:

```
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])**

recordset	The set of records to process.
transform	The TRANSFORM function to call for each record in the *recordset*.

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:

```
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 Rtot
 1   0
 2   0
 3   0
 4   0 */

ResType T(ResType L, ResType R) := TRANSFORM
  SELF.Rtot := L.Rtot + R.Val;
  SELF := R;
END;

MySet1 := ITERATE(Records,T(LEFT,RIGHT));

/* these are the recs coming out:
Val Rtot
 1   1
 2   3
 3   6
 4   10 */

//The following code outputs a running balance:
Run_bal := RECORD
  Trades.trd_bal;
  INTEGER8 Balance := 0;
END;
```

```
TradesBal := TABLE(Trades,Run_Bal);

Run_Bal DoRoll(Run_bal L, Run_bal R) := TRANSFORM
  SELF.Balance := L.Balance + IF(validmoney(R.trd_bal),R.trd_bal,0);
  SELF := R;
END;

MySet2 := ITERATE(TradesBal,DoRoll(LEFT,RIGHT));
```

See Also: TRANSFORM Structure, RECORD Structure, ROLLUP

JOIN

JOIN(*leftrecset, rightrecset, joincondition* **[**, *transform***] [**, *jointype***] [**, *joinflags***])**

JOIN(*setofdatasets, joincondition, transform*, **SORTED(** *fields***) [**, *jointype***])**

leftrecset	The left set of records to process.
rightrecset	The right set of records to process. This may be an INDEX.
joincondition	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*.
transform	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.
jointype	Optional. An inner join if omitted, else one of the listed types in the JOIN Types section below.
joinflags	Optional. Any option (see the JOIN Options section below) to specify exactly how the JOIN operation executes.
setofdatasets	The SET of recordsets to process ([idx1,idx2,idx3]), typically INDEXes, which all must have the same format.
SORTED	Specifies the sort order of records in the input *setofdatasets* and also the output sort order of the result set.
fields	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.
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.

Matching Logic - JOIN

The record matching *joincondition* is processed internally as two parts:

"equality" (hard 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" (soft 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][, KEEP(*n*)] [**, **ATMOST([** *condition,* **] *n*)] [**, **LIMIT(** *value* **[, SKIP** | *transform* | **FAIL])] [**, **SKEW(***limit* **[**, *target***]) [, THRESHOLD(** *size* **)]] [, PARALLEL][, SMART]**

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 that can be fully copied to every 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*). 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 *left-recset*.
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. 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). If omitted, the default is LIMIT(10000). 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.
value	The maximum number of matches allowed; LIMIT(0) is unlimited.
SKIP	Optional. Specifies eliminating the matching records that exceed the maximum value of the LIMIT result instead of failing the job.
transform	Optional. Specifies outputting a single record produced by the *transform* instead of failing the workunit (similar to the ONFAIL option of the LIMIT function).
FAIL	Optional. Specifies using the FAIL action to configure the error message when the job fails.
SKEW	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).
limit	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%).
target	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%).
THRESHOLD	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).
size	An integer value indicating the minimum number of bytes for a single part.
PARALLEL	Specifies the *leftrecset* and *rightrecset* should be read on separate threads to minimize latency.

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.

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 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.

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*:

inner (default)	Only those records that exist in both the *leftrecset* and *rightrecset*.
LEFT OUTER	At least one record for every record in the *leftrecset*.
RIGHT OUTER	At least one record for every record in the *rightrecset*.
FULL OUTER	At least one record for every record in the *leftrecset* and *rightrecset*.
LEFT ONLY	One record for each *leftrecset* record with no match in the *rightrecset*.
RIGHT ONLY	One record for each *rightrecset* record with no match in the *leftrecset*.
FULL ONLY	One record for each *leftrecset* and *rightrecset* record with no match in the opposite record set.

Example:

```
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},
                  '~RTTEMPkey::lname.fname');
SeqKey := INDEX(Ptbl,{seq,__fpos},'~RTTEMPkey::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

JOIN(*setofdatasets, joincondition, transform,* **SORTED(** *fields***) [**, *jointype***])**

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 setof-datasets

The *transform* function must take at least one parameter which must take either of two forms:

LEFT	formatted like any of the *setofdatasets*. This indicates the first dataset in the *setofdatasets*.
ROWS(LEFT)	formatted like any of the *setofdatasets*. This indicates a record set made up of all records from any dataset in the *setofdatasets* that match the *joincondition*—this may not include all the datasets in the *setofdatasets*, depending on which *jointype* is specified.

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*:

INNER	This is the default if no *jointype* is specified. Only those records that exist in all datasets in the *setofdatasets*.
LEFT OUTER	At least one record for every record in the first dataset in the *setofdatasets*.
LEFT ONLY	One record for every record in the first dataset in the *setofdatasets* for which there is no match in any of the subsequent datasets.
MOFN(min [,max])	One record for every record with matching records in min number of adjacent datasets within the *setofdatasets*. If max is specified, the record is not included if max number of dataset matches are exceeded.

Example:

```
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)),SORTED(Letter),LEFT OUTER);
j3 := JOIN(SetDS,
           STEPPED(LEFT.Letter=RIGHT.Letter),
           XF(LEFT,ROWS(LEFT)),SORTED(Letter),LEFT ONLY);
j4 := JOIN(SetDS,
           STEPPED(LEFT.Letter=RIGHT.Letter),
           XF(LEFT,ROWS(LEFT)),SORTED(Letter),MOFN(3));
j5 := JOIN(SetDS,
           STEPPED(LEFT.Letter=RIGHT.Letter),
           XF(LEFT,ROWS(LEFT)),SORTED(Letter),MOFN(3,4));

OUTPUT(j1);
OUTPUT(j2);
OUTPUT(j3);
OUTPUT(j4);
OUTPUT(j5);
```

See Also: TRANSFORM Structure, RECORD Structure, SKIP, STEPPED, KEYED/WILD, MERGEJOIN

KEYDIFF

[*attrname* **:=] KEYDIFF(** *index1, index2, file* **[, OVERWRITE] [, EXPIRE([** *days* **])]);**

attrname	Optional. The action name, which turns the action into an attribute definition, therefore not executed until the attrname is used as an action.
index1	An INDEX attribute.
index2	An INDEX attribute whose structure is identical to index1.
file	A string constant specifying the logical name of the file to write the differences to.
OVERWRITE	Optional. Specifies overwriting the filename if it already exists.
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).

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:

```
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

[attrname **:=] KEYPATCH(** *index, patchfile, newfile* **[, OVERWRITE] [, EXPIRE([** *days* **])]);**

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).

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:

```
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');
a := KEYDIFF(i1,i2,'KEY::DIFF::20050101::i1i2',OVERWRITE);
b := KEYPATCH(i1,
    'KEY::DIFF::20050101::i1i2',
    'vkey::st.city.lname'OVERWRITE);
SEQUENTIAL(a,b);
```

See Also: KEYDIFF, INDEX

KEYUNICODE

KEYUNICODE(*string***)**

string	A UNICODE string.
Return:	KEYUNICODE returns a single DATA value.

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:

```
//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***)**

expression	A string expression.
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');
      //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:

```
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),
      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'));
```

LIMIT

LIMIT(*recset*, *maxrecs* **[**, *failclause* **] [**, **KEYED [**, **COUNT]] [**, **SKIP])**

LIMIT(*recset*, *maxrecs* **[**, **ONFAIL(***transform***)] [**, **KEYED [**, **COUNT]])**

recset	The set of records to limit. This may be an INDEX or any expression that produces a recordset result.
maxrecs	The maximum number of records allowed on a single supercomputer node.
failclause	Optional. A standard FAIL workflow service call.
KEYED	Optional. Specifies limiting the keyed portion of an INDEX read.
COUNT	Optional. Specifies the KEYED limit is pre-checked using keyspan.
SKIP	Optional. Specifies that when the limit is exceeded it is simply eliminated from any result instead of failing the workunit.
ONFAIL	Optional. Specifies outputting a single record produced by the transform instead of failing the workunit.
transform	The TRANSFORM function to call to produce the single output record.

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:

```
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);
```

See Also: FAIL, TRANSFORM

LN

LN(*n*)

n	The real number to evaluate.
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); //1.14473
```

See Also: EXP, SQRT, POWER, LOG

LOADXML

[attributename := **] LOADXML(** *xmlstring* | *symbol* **[**, *branch* **])**

attributename	Optional. The action name, which turns the action into an attribute definition, therefore not executed until the *attributename* is used as an action.
xmlstring	A string expression containing the XML text to process inline (no carriage returns or line feeds).
symbol	The template symbol containing the XML text to process (typically loaded by #EXPORT or #EXPORTXML).
branch	A user-defined string naming the XML text, allowing #FOR to operate.

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:

```
LOADXML('<section><item type="count"><set>person</set></item></section>')
//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'%
     + ',{countTaxdata := COUNT(Taxrecs), ds'
     + %'id'% + '});\n' )
  #APPEND(OutStr,
     'OUTPUT(FETCH(Files.Vehicle,Files.VehicleIDX(personid= '
         + %'id'% + '),RIGHT.RecPos));\n' )
#END
%OutputStr%
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}'%
   #END
+ ' ' + %'{@label}'% + ';');
  #END
 #END
#END
OUTPUT('Done');
```

See Also: Templates, #EXPORT, #EXPORTXML

LOCAL

LOCAL(*data***)**

data	The name of a DATASET or INDEX attribute.
Return:	LOCAL returns a record set or index.

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:

```
ds := JOIN(SomeData,LOCAL(SomeIndex), LEFT.ID = RIGHT.ID);
```

See Also: ALLNODES, THISNODE, NOLOCAL

LOG

LOG(*n*)

n	The real number to evaluate.
Return:	LOG returns a single real value.

The **LOG** function returns the base-10 logarithm of the parameter.

Example:

```
MyLogPI := LOG(3.14159); //0.49715
```

See Also: EXP, SQRT, POWER, LN

LOOP

LOOP(*dataset, loopcount, loopbody* **[, PARALLEL(** *iterations* | *iterationlist* **[**, *default* **])])**

LOOP(*dataset, loopcount, loopfilter, loopbody* **[, PARALLEL(** *iterations* | *iterationlist* **[**, *default* **])])**

LOOP(*dataset, loopfilter, loopbody* **)**

LOOP(*dataset, loopcondition, loopbody* **)**

LOOP(*dataset, loopcondition, rowfilter, loopbody* **)**

dataset	The record set to process.
loopcount	An integer expression specifying the number of times to iterate .
loopbody	The operation to iteratively perform. This may be a PROJECT, JOIN, or other such operation. ROWS(LEFT) is always used as the operation's first parameter, indicating the specified dataset is the input parameter.
PARALLEL	Optional. Specifies parallel execution of loop iterations. This option is available only on Roxie.
iterations	The number of parallel iterations.
iterationlist	A set of integers (contained in square brackets) specifying the number of parallel iterations for each loop. The first set element specifies the parallel iterations for the first loop, the second for the second, ...
default	Optional. The number of parallel iterations to execute once all elements in the *iterationlist* have been used.
loopfilter	A logical expression that specifies the set of records whose processing is not yet complete. The set of records not meeting the condition are no longer iteratively processed and are placed into the final result set. This evaluation occurs before each iteration of the *loopbody*.
loopcondition	A logical expression specifying continuing *loopbody* iteration while TRUE.
rowfilter	A logical expression that specifies a single record whose processing is complete. The record meeting the condition is no longer iteratively processed and is placed into the final result set. This evaluation occurs during the iteration of the *loopbody*.
Return:	LOOP returns a record set.

The **LOOP** function iteratively performs the *loopbody* operation. The COUNTER is implicit and available for use to return the current iteration.

The PARALLEL Option

The PARALLEL option is offered to solve the following type of problem: When implementing a text search (A and B and C) or (D and E), where each element in the search is evaluated on an iteration of a LOOP(), you want to ensure that the execution is broken in the correct places. If it were split every 2 iterations, the iterations would produce:

(A and B)

(A and B and C), (D)

(A and B and C) or (D and E)

The second iteration would potentially generate a very large number of temporary records. To prevent this, the number of iterations at each step can be controlled. For this specific case you would probably use PARALLEL([3,3]). For more complicated search criteria the numbers would be different.

If a very large number is provided as the *iterations* or *default* value, then the all the iterations will execute in parallel. Doing this will likely significantly reduce the number of temporary rows stored in the system, but may potentially use a large amount of resources.

There is a restriction: ROWS(LEFT) cannot be directly used in a sub-query of the *loopbody*.

Example:

```
namesRec := RECORD
STRING20 lname;
STRING10 fname;
  UNSIGNED2 age := 25;
  UNSIGNED2 ctr := 0;
END;
namesTable2 := DATASET([{'Flintstone','Fred',35},
    {'Flintstone','Wilma',33},
    {'Jetson','Georgie',10},
    {'Mr. T','Z-man'}], namesRec);
loopBody(DATASET(namesRec) ds, unsigned4 c) :=
 PROJECT(ds,
    TRANSFORM(namesRec,
    SELF.age := LEFT.age*c;
    SELF.ctr := COUNTER ;
    SELF := LEFT));
//Form 1:
OUTPUT(LOOP(namesTable2,
  COUNTER <= 10,
            PROJECT(ROWS(LEFT),
     TRANSFORM(namesRec,
     SELF.age := LEFT.age*2;
     SELF.ctr := LEFT.ctr + COUNTER ;
     SELF := LEFT))));
OUTPUT(LOOP(namesTable2, 4, ROWS(LEFT) & ROWS(LEFT)));
//Form 2:
OUTPUT(LOOP(namesTable2,
  10,
  LEFT.age * COUNTER <= 200,
            PROJECT(ROWS(LEFT),
     TRANSFORM(namesRec,
     SELF.age := LEFT.age*2;
     SELF := LEFT))));
//Form 3:
OUTPUT(LOOP(namesTable2,
  LEFT.age < 100,
  loopBody(ROWS(LEFT), COUNTER)));
//Form 4:
OUTPUT(LOOP(namesTable2,
  SUM(ROWS(LEFT), age) < 1000 * COUNTER,
            PROJECT(ROWS(LEFT),
     TRANSFORM(namesRec,
     SELF.age := LEFT.age*2;
     SELF := LEFT))));
//Form 5:
OUTPUT(LOOP(namesTable2,
  LEFT.age < 100,
  EXISTS(ROWS(LEFT)) and SUM(ROWS(LEFT), age) < 1000,
  loopBody(ROWS(LEFT), COUNTER)));
```

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])**

MAX(*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. 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.
value	The expression to find the maximum value of.
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 maximum value of. This may also be a SET of values.
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:

```
MaxVal1 := MAX(Trades,Trades.trd_rate);
MaxVal2 := MAX(4,8,16,2,1); //returns 16
SetVals := [4,8,16,2,1];
MaxVal3 := MAX(SetVals); //returns 16
```

See Also: MIN, AVE

MERGE

MERGE(*recordsetlist* , **SORTED(** *fieldlist* **) [**, **DEDUP] [**, **LOCAL])**

MERGE(*recordsetset* , *fieldlist* , **SORTED(** *fieldlist* **) [**, **DEDUP] [**, **LOCAL])**

recordsetlist	A comma-delimited list of the datasets or indexes to merge, which must all be in exactly the same format and sort order.
SORTED	Specifies the sort order of the *recordsetlist*.
fieldlist	A comma-delimited list of the fields that define the sort order.
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 DISTRIBUTE.
recordsetset	A SET ([ds1,ds2,ds3]) of the datasets or indexes to merge, which must all be in exactly the same format.
Return:	MERGE returns a record set.

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.

Example:

```
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,letter,number);
```

MERGEJOIN

MERGEJOIN(*setofdatasets, joincondition,* **SORTED(** *fields***) [**, *jointype***] [, DEDUP])**

setofdatasets	The SET of recordsets to process ([idx1,idx2,idx3]), typically INDEXes, which all must have the same format.
joincondition	An expression specifying how to match records in the *setofdatasets*.
SORTED	Specifies the sort order of records in the input *setofdatasets* and also the output sort order of the result set.
fields	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.
jointype	Optional. An inner join if omitted, else one of the listed types below.
DEDUP	Optional. Specifies the output result set contains only unique records.

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 ANDed 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*:

INNER	Only those records that exist in all datasets in the *setofdatasets*.
LEFT OUTER	At least one record for every record in the first dataset in the *setofdatasets*.
LEFT ONLY	One record for every record in the first dataset in the *setofdatasets* for which there is no match in any of the subsequent datasets.
MOFN(min **[**,max**])**	One record for every record with matching records in min number of adjacent datasets within the *setofdatasets*. If max is specified, the record is not included if max number of dataset matches are exceeded.

Example:

```
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

MIN(*recordset, value* **[, KEYED])**

MIN(*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. 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.
value	The expression to find the minimum value of.
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 minimum value of. This may also be a SET of values.
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:

```
MinVal1 := MIN(Trades,Trades.trd_rate);
MinVal2 := MIN(4,8,16,2,1); //returns 1
SetVals := [4,8,16,2,1];
MinVal3 := MIN(SetVals); //returns 1
```

See Also: MAX, AVE

NOLOCAL

NOLOCAL(*data***)**

data	The name of a DATASET or INDEX attribute.
Return:	NOLOCAL returns a record set or index.

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:

```
ds := JOIN(SomeData,NOLOCAL(SomeIndex), LEFT.ID = RIGHT.ID);
```

See Also: ALLNODES, THISNODE, LOCAL

NONEMPTY

NONEMPTY(*recordsetlist***)**

recordsetlist	A comma-delimited list of record sets.
Return:	NONEMPTY returns a record set.

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:

```
ds := NONEMPTY(SomeData(SomeFilter),
SomeData(SomeOtherFilter),
SomeOtherData(YetAnotherFilter));
```

See Also: EXISTS

NORMALIZE

NORMALIZE(*recordset, expression, transform* **)**

NORMALIZE(*recordset,* **LEFT.***childdataset, transform* **)**

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.
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 *childdataset*. The resulting record set format does not need to be the same as the input.

Example:

```
//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));
/* 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 */
//***********************
//Form 2 example
ChildRec := RECORD
INTEGER1 NameID;
STRING20 Addr;
END;
DenormedRec := RECORD
INTEGER1 NameID;
STRING20 Name;
DATASET(ChildRec) Children;
END;

ds := DATASET([ {1,'Kevin',[ {1,'10 Malt Lane'}]},
{2,'Liz', [ {2,'10 Malt Lane'},
{2,'3 The cottages'}]},
{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));
```

See Also: TRANSFORM Structure, RECORD Structure, DENORMALIZE

NOFOLD

[*name* := **] NOFOLD(** *expression* **)**

name	Optional. The identifier for this function.
expression	The expression to evaluate.

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:

```
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));
```

NOTHOR

[*name* := **] NOTHOR(** *action* **)**

name	Optional. The identifier for this action.
action	The action to execute.

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:

```
rec := RECORD
  STRING10 S;
END;

srcnode := '10.150.199.2';
srcdir := '/c$/test/';

dir := FileServices.RemoteDirectory(srcnode,srcdir,'*.txt',TRUE);

  //without NOTHOR this code gets this error:
  // "Cannot call function AddSuperFile in a non-global context"
NOTHOR(SEQUENTIAL(
  FileServices.DeleteSuperFile('MultiSuper1'),
  FileServices.CreateSuperFile('MultiSuper1'),
  FileServices.StartSuperFileTransaction(),
  APPLY(dir,FileServices.AddSuperFile('MultiSuper1',
  FileServices.ExternalLogicalFileName(srcnode,srcdir+name))),
  FileServices.FinishSuperFileTransaction()));

F1 := DATASET('MultiSuper1', rec, THOR);
OUTPUT(F1,,'testmulti1',overwrite);
```

See Also: SEQUENTIAL

NOTIFY

[*attributename* := **] NOTIFY(** *event* **[**, *parm* **] [**, *expression* **])**

attributename	Optional. The identifier for this action.
event	The EVENT function, or a case-insensitive string constant naming the event to generate.
parm	Optional. A case-insensitive string constant containing the event's parameter.
expression	Optional. A case-insensitive string constant allowing simple message passing, to restrict the event to a specific workunit.

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:

```
NOTIFY('testevent', 'foobar');

receivedFileEvent(STRING name) := EVENT('ReceivedFile', name);
NOTIFY(receivedFileEvent('myfile'));

//as a service
doMyService := FUNCTION
OUTPUT('Did a Service for: ' + 'EVENTNAME=' + EVENTNAME);
NOTIFY(EVENT('MyServiceComplete',
'<Event><returnTo>FRED</returnTo></Event>'),
EVENTEXTRA('returnTo'));
RETURN EVENTEXTRA('returnTo');
END;

doMyService : WHEN('MyService');
// and a call to the service
NOTIFY('MyService',
'<Event><returnTo>'+WORKUNIT+'</returnTo>....</Event>');
WAIT('MyServiceComplete');
OUTPUT('WORKUNIT DONE')
```

See Also: EVENT, EVENTNAME, EVENTEXTRA, CRON, WHEN, WAIT

OUTPUT

[*attr* := **] OUTPUT(***recordset* **[, [** *format* **] [,***file* **[***thorfileoptions* **]] [**, **NOXPATH]);**

[*attr* := **] OUTPUT(***recordset***, [** *format* **] ,***file* , **CSV [(***csvoptions***)] [***csvfileoptions* **] [**, **NOXPATH]);**

[*attr* := **] OUTPUT(***recordset***, [** *format* **] ,** *file* , **XML [(***xmloptions***)] [***xmlfileoptions* **] [**, **NOXPATH]);**

[*attr* := **] OUTPUT(***recordset***, [** *format* **] ,PIPE(** *pipeoptions* **[**, **NOXPATH]);**

[*attr* := **] OUTPUT(***recordset* **[,** *format* **] ,** **NAMED(** *name* **) [,EXTEND] [,ALL] [**, **NOXPATH]);**

[*attr* := **] OUTPUT(** *expression* **[**, **NAMED(** *name* **)] [**, **NOXPATH]);**

[*attr* := **] OUTPUT(** *recordset* , **THOR [**, **NOXPATH]);**

attr	Optional. The action name, which turns the action into a definition, therefore not executed until the *attr* is used as an action.
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.
format	Optional. The format of the output records. If omitted, all fields in the *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.
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.
thorfileoptions	Optional. A comma-delimited list of options valid for a THOR/FLAT file (see the section below for details).
NOXPATH	Specifies any XPATHs defined in the *format* or the RECORD structure of the *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 input can be ignored for output.
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).
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.
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

[*attr* := **] OUTPUT(***recordset* **[, [** *format* **] [,***file* **[, CLUSTER(** *target* **)] [,ENCRYPT(** *key* **)]**

[,COMPRESSED] [,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.
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 omitted, the default is seven (7).

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;
  People.lastname;
END;

A_People := People(lastname[1]='A');
Score1 := HASHCRC(People.firstname);
Attr1 := People.firstname[1] = 'A';

OUTPUT(SORT(A_People,Score1),OutputFormat1,'hold01::fred.out');
  // writes the sorted A_People set to the fred.out file in
  // the format declared in the OutputFormat1 definition

OUTPUT(People,{firstname,lastname});
  // writes just First and Last Names to the command issuer
  // full qualification of the fields is unnecessary, since
  // the "on-the-fly" records structure is within the
  // scope of the OUTPUT -- People is assumed

OUTPUT(People(Attr1=FALSE));
  // writes all Peeople fields from records where Attr1 is
  // false to the command issuer
```

OUTPUT CSV Files

[*attr* := **] OUTPUT(***recordset***, [** *format* **] ,***file* **, CSV [(***csvoptions***)] [, CLUSTER(** *target* **)] [,ENCRYPT(key)]**

[, **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* in the specified *format* as a comma separated values ASCII file. The valid set of *csvoptions* are:

HEADING([*headertext* **[** , *footertext* **]] [, SINGLE])**

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).
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('\n'), QUOTE())
```

Example:

```
//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>')));
```

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('Row',HEADING('<Dataset>\n','</Dataset>\n'))
```

Example:

```
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(B,,'fred2.xml',XML('MyRow', HEADING('<?xml version=1.0 ...?>\n<filetag>\n','</filetag>\n')));
/* the Fred2.XML file looks like this:
<?xml version=1.0 ...?>
<filetag>
```

```
  <MyRow><fname>Fred </fname><lname>Bell</lname></MyRow>
  <MyRow><fname>George</fname><lname>Blanda</lname></MyRow>
  <MyRow><fname>Sam </fname><lname></lname></MyRow>
</filetag> */

OUTPUT(B,,'fred3.xml',XML('MyRow',TRIM,OPT));
/* the Fred3.XML file looks like this:
<Dataset>
  <MyRow><fname>Fred</fname><lname>Bell</lname></MyRow>
  <MyRow><fname>George</fname><lname>Blanda</lname></MyRow>
  <MyRow><fname>Sam</fname></MyRow>
</Dataset> */
```

OUTPUT PIPE Files

[*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:

```
OUTPUT(A_People,,PIPE('MyCommandLIneProgram'),OVERWRITE);
   // sends the A_People to MyCommandLIneProgram as
   // standard in
```

Named OUTPUT

[*attr* := **] OUTPUT(***recordset* **[,** *format* **]** ,**NAMED(** *name* **) [,EXTEND] [**,**ALL])**

This form writes the *recordset* to the workunit with the specified *name*. 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:

```
OUTPUT(CHOOSEN(people(firstname[1]='A'),10));
  // writes the A People to the query builder
OUTPUT(CHOOSEN(people(firstname[1]='A'),10),ALL);
  // writes all the A People to the query builder
OUTPUT(CHOOSEN(people(firstname[1]='A'),10),NAMED('fred'));
  // writes the A People to the fred named output

//a NAMED, EXTEND example:
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

[*attr* := **] OUTPUT(** *expression* **[**, **NAMED(** *name* **)])**

This form is used to allow scalar *expression* output, particularly within SEQUENTIAL and PARALLEL actions.

Example:

```
OUTPUT(10) // scalar value output
OUTPUT('Fred') // scalar value output
```

OUTPUT Workunit Files

[*attr* := **] 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:

```
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

[*attributename* := **] PARALLEL(** *actionlist* **)**

attributename	Optional. The action name, which turns the action into an attribute definition, therefore not executed until the *attributename* 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 executes the items in the *actionlist* simultaneously. This is already the default operative mode, so PARALLEL is only useful within the action list of a SEQUENTIAL set of actions.

Example:

```
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 attributes
//that only execute when the attribute names are called as actions

SEQUENTIAL(Act1,PARALLEL(Act2,Act3));

//executes Act1 alone, and only when it's finished,
// executes Act2 and Act3 together
```

See Also: SEQUENTIAL

PARSE

PARSE(*dataset, data, pattern, result , flags* **[, MAXLENGTH(** *length* **)])**

PARSE(*dataset, data, result ,* **XML(** *path* **))**

dataset	The set of records to process.
data	An expression specifying the text to parse, typically the name of a field in the dataset.
pattern	The parsing pattern to match.
result	The name of either the RECORD structure attribute that specifies the format of the output record set (like the TABLE function), or the TRANSFORM function that produces the output record set (like PROJECT).
flags	One or more parsing options, listed below.
MAXLENGTH	Specifies the the maximum length the pattern can match. If omitted, the default length is 4096.
length	An integer constant specifying the maximum number of matching characters.
XML	Specifies the dataset contains XML data.
path	A string constant containing the XPATH to the tag that delimits the XML data in the dataset.
Return:	PARSE returns a record set.

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:

FIRST	Only return a row for the first match starting at a particular position.
ALL	Return a row for every possible match of the string at a particular position.
WHOLE	Only match the whole string.
NOSCAN	If a position matches, don't continue searching for other matches.
SCAN	If a position matches, continue searching from the end of the match, otherwise continue from the next position.
SCAN ALL	Return matches for every possible start position. Use the TRIM function to eliminate parsing extraneous trailing blanks.
NOCASE	Perform a case insensitive comparison.
CASE	Perform a case sensitive comparison (this is the default).

SKIP(*separator-pattern*)	Specify a pattern that can be inserted after each token in a search pattern. For example, SKIP ([' ','\t']*) skips spaces and tabs between tokens.
KEEP(*max*)	Only keep the first *max* matches.
ATMOST(*max*)	Don't produce any matches if there are more than *max* matches.
MAX	Return a row for the result that matches the longest sequence of the input. Only one match is returned unless the MANY option is also specified.
MIN	Return a row for the result that matches the shortest sequence of the input. Only one match is returned unless the MANY option is also specified.
MATCHED([*rule-reference*])	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.
MATCHED(ALL)	Retain all rule-names – if they are used by user match functions.
NOT MATCHED	Generate a row if there were no matches on the input row. All calls to the MATCHED() function return false inside the *resultstructure*.
NOT MATCHED ONLY	Only generate a row if no matches were found.
BEST	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.
MANY	Return multiple matches for BEST, MAX, or MIN options.
PARSE	Implements Tomita parsing instead of regular expression parsing technology.
USE([*struct*,] x)	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.

Example:

```
rec := {STRING10000 line};
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 Esaus sons; Eliphaz the son of Adah the wife of Esau,'+
     ' Reuel the son of Bashemath the wife of Esau.'}],rec);
PATTERN ws1 := [' ','\t',','];
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(Namet[1]);
  STRING60 Ri := MATCHTEXT(Namet[2]);
  STRING30 RelationPhrase := MatchText(relationtype);
END;
outfile1 := PARSE(datafile,line,progeny,results,SCAN ALL);
```

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:

```
linerec := { STRING line };
in1 := DATASET([{
        '<ENTITY eid="P101" type="PERSON" subtype="MILITARY">' +
        '  <ATTRIBUTE name="fullname">JOHN SMITH</ATTRIBUTE>' +
        '  <ATTRIBUTE name="honorific">Mr.</ATTRIBUTE>' +
        '  <ATTRIBUTEGRP descriptor="passport">' +
        '     <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"/>' +
        '  </ATTRIBUTEGRP>' +
        '</ENTITY>'}],
     linerec);
passportRec := RECORD
  STRING id;
  STRING idType;
  STRING issuer;
  STRING country;
  INTEGER age;
END;
outrec := RECORD
  STRING id;
  UNICODE fullname;
  UNICODE title;
  passportRec passport;
  STRING line;
END;
outrec t(lineRec L) := TRANSFORM
  SELF.id := XMLTEXT('@eid');
  SELF.fullname := XMLUNICODE('ATTRIBUTE[@name="fullname"]');
  SELF.title := XMLUNICODE('ATTRIBUTE[@name="honorific"]');
  SELF.passport.id := XMLTEXT('ATTRIBUTEGRP[@descriptor="passport"]'
                              + '/ATTRIBUTE[@name="idNumber"]');
  SELF.passport.idType := XMLTEXT('ATTRIBUTEGRP[@descriptor="passport"]'
                                  + '/ATTRIBUTE[@name="idType"]');
  SELF.passport.issuer := XMLTEXT('ATTRIBUTEGRP[@descriptor="passport"]'
```

```
                                + '/ATTRIBUTE[@name="issuingAuthority"]');
  SELF.passport.country := XMLTEXT('ATTRIBUTEGRP[@descriptor="passport"]'
                                 + '/ATTRIBUTE[@name="country"]/@value');
  SELF.passport.age := (INTEGER)XMLTEXT('ATTRIBUTEGRP[@descriptor="passport"]'
                                      + '/ATTRIBUTE[@name="age"]/@value');
  SELF := L;
END;

textout := PARSE(in1, line, t(LEFT), XML('/ENTITY[@type="PERSON"]'));
```

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 numbers := PATTERN('[0-9]')+;
PATTERN alpha := PATTERN('[A-Za-z]')+;
PATTERN ws := [' ','\t']*;
PATTERN sepchar := PATTERN('[-./ ]');
PATTERN Seperator := ws sepchar ws;

// Area Code
PATTERN OpenParen := ['[','(','{','<'];
PATTERN CloseParen := [']',')','}','>'];
PATTERN FrontDigit := ['1', '0'] OPT(Seperator);
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 (freely available at IMDB.com) into standard database fields:

```
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) [Kampanna 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>'},
{' Vindhyarani (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
          Askeladd] <3>'},
{' Spokelse forelsker 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>'},
{''},
{'Aabel, Per (I) Brudebuketten (1953) [Hoyland jr.] <3>'},
{' Cafajestes, Os (1962)'},
{' Farlige leken, Den (1942) [Fredrik Holm, doktor]'},
{' Herre med bart, En (1942) [Ole Grong, advokat] <1>'},
{' Kjaere Maren (1976) [Doktor]'},
{' Kjaerlighet og vennskap (1941) [Anton Schack] <3>'},
{' Ombyte fornojer (1939) [Gregor Ivanow] <2>'},
{' Portrettet (1954) [Per Haug, provisor] <1>'}],
Layout_Actors_Raw);

//Basic patterns:
PATTERN arb := PATTERN('[-!.,\t a-zA-Z0-9]')+;

//all alphanumeric & certain special characters
PATTERN ws := [' ','\t']+; //word separators (space & tab)
PATTERN number := PATTERN('[0-9]')+; //numbers

//extended patterns:
PATTERN age := '(' number OPT('/I') ')';

//movie year -- OPT('/I') required for first rec
PATTERN role := '[' arb ']'; //character played
PATTERN m_rank := '<' number '>'; //credit appearance number
PATTERN actor := arb OPT(ws '(I)' ws);
//actor's name -- OPT(ws '(I)' ws)
// required for last two actors

//extended pattern to parse the actual text:
PATTERN line := actor '\t' arb ws OPT(age) ws OPT(role) ws OPT(m_rank) ws;
```

```
//output record structure:
NLP_layout_actor_movie := RECORD
  STRING30 actor_name := Std.Str.filterout(MATCHTEXT(actor),'\t');
  STRING50 movie_name := MATCHTEXT(arb[2]);
  UNSIGNED2 movie_year := (UNSIGNED)MATCHTEXT(age/number);
  STRING20 movie_role := MATCHTEXT(role/arb);
  UNSIGNED1 cast_rank := (UNSIGNED)MATCHTEXT(m_rank/number);
END;

//and the actual parsing operation
Actor_Movie_Init := PARSE(File_Actors,
                          IMDB_Actor_Desc,
                          line,
                          NLP_layout_actor_movie,WHOLE,FIRST);

// then iterate to propagate actor name in each record
NLP_layout_actor_movie IterNames(NLP_layout_actor_movie L,
                                 NLP_layout_actor_movie R) := TRANSFORM
  SELF.actor_name := IF(R.actor_Name='',L.actor_Name,R.actor_name);
  SELF:= R;
END;

NLP_Actor_Movie := ITERATE(Actor_Movie_Init,IterNames(LEFT,RIGHT));

// and output the result set
OUTPUT(NLP_Actor_Movie);
```

PIPE

PIPE(*command, recorddef* **[, CSV | XML])**

PIPE(*recordset, command* **[**, *recorddef* **] [**, **REPEAT] [**, **CSV | XML] [**, **OUTPUT(CSV | XML)] [**, **GROUP])**

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 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 record-set.
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).
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:

```
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***)**

base	The real number to raise.
exponent	The real power to raise x to.
Return:	POWER returns a single real value.

The **POWER** function returns the result of the *base* raised to the *exponent* power.

Example:

```
MyCube := POWER(2.0,3.0); // = 8
MySquare := POWER(3.0,2.0); // = 9
```

See Also: SQRT, EXP, LN

PRELOAD

PRELOAD(*file* **[**, *nbr* **])**

file	The name of a DATASET or INDEX 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 use). This is exactly equivalent to using the PRELOAD option on the DATASET or INDEX definition.

Example:

```
MyFile := DATASET('MyFile',{STRING20 F1, STRING20 F2},THOR);
  COUNT(PRELOAD(MyFile))
```

See Also: DATASET, INDEX

PROCESS

PROCESS(*recordset, datarow, datasettransform, rowtransform* **[, LOCAL])**

recordset	The set of records to process.
datarow	The initial RIGHT record to process, typically expressed by the ROW function.
datasettransform	The TRANSFORM function to call for each record in the recordset.
rowtransform	The TRANSFORM function to call to produce the next RIGHT record for the *datasettransform*.
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:	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 in the recordset 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 *datsettransform* 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).

Example:

```
DSrec := RECORD
  STRING4 Letter;
  STRING4 LeftRecIn := '';
  STRING4 RightRecIn := '';
END;
StateRec := RECORD
  STRING2 Letter;
END;
ds := DATASET([{'AA'},{'BB'},{'CC'},{'DD'},{'EE'}],DSrec);

DSrec DSxform(DSrec L,StateRec R) := TRANSFORM
  SELF.Letter := L.Letter[1..2] + R.Letter;
  SELF.LeftRecIn := L.Letter;
  SELF.RightRecIn := R.Letter;
END;
StateRec ROWxform(DSrec L,StateRec R) := TRANSFORM
  SELF.Letter := L.Letter[1] + R.Letter[1];
END;

p := PROCESS(ds,
             ROW({'ZZ'},StateRec),
             DSxform(LEFT,RIGHT),
             ROWxform(LEFT,RIGHT));
OUTPUT(p);
/* Result:
```

```
AAZZ AA ZZ
BBAZ BB AZ
CCBA CC BA
DDCB DD CB
EEDC EE DC */

//*****************************************************************
// This examples 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

PROJECT

PROJECT(*recordset, transform* **[, PREFETCH [(***lookahead* **[, PARALLEL])]] [, KEYED] [, LOCAL])**

PROJECT(*recordset, record* **[, PREFETCH [(***lookahead* **[, PARALLEL])]] [, KEYED] [, LOCAL])**

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.
Return:	PROJECT returns a record set.

The **PROJECT** function processes through all records in the *recordset* performing the *transform* function on each record in turn.

The PROJECT(*recordset*,*record*) form is simply a shorthand synonym for:

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*.

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:

```
//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 := DISTRIBUTE(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* **])**

module	The MODULE structure containing the attribute definitions whose values to pass as the interface.
interface	The INTERFACE structure to pass.
OPT	Optional. Suppresses the error message that is generated when an attribute defined in the interface is not also defined in the module.
attributelist	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.
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***)**

dataset	The set of records to fully load into the Data Refinery.
Return:	PULL returns a recordset.

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:

```
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 positive integer value.

Example:

```
MySet := DISTRIBUTE(Person,RANDOM()); //random distribution
```

See Also: DISTRIBUTE

RANGE

RANGE(*setofdatasets, setofintegers* **)**

setofdatasets	A set of datasets.
setofintegers	A set of integers.
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:

```
Ranking := RANK(1,[20,30,10,40]);
// returns 2 - 1st element (20) in unsorted set is
// 2nd element after sorting to [10,20,30,40]
Ranking := 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]
```

See Also: RANKED, SORT, SORTED, Sets and Filters

RANKED

RANKED(*position, set* **[,DESCEND])**

position	An integer indicating the element in the unsorted set to return.
set	The set of values.
DESCEND	Optional. Indicates descending order sort.
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:

```
Ranking := RANKED(1,[20,30,10,40]);
// returns 3 - 1st element (10) in sorted set [10,20,30,40]
// was 3rd element in unsorted set

Ranking := 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
```

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* specifed.

Example:

```
REAL8 Float := 1000.0063;
STRING12 FloatStr12 := REALFORMAT(float,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_39_0/libs/regex/doc/html/index.html. 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:

```
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); //1 record -- MacPherson
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, REGEXREPLACE

REGEXREPLACE

REGEXREPLACE(*regex, text, replacement* **[, NOCASE])**

regex	A standard Perl regular expression.
text	The text to parse.
replacement	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.
NOCASE	Optional. Specifies a case insensitive search.
Return:	REGEXREPLACE returns a single value.

The **REGEXREPLACE** 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_39_0/libs/regex/doc/html/index.html. 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:

```
REGEXREPLACE('(.a)t', 'the cat sat on the mat', '$1p');
        //ASCII
REGEXREPLACE(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 := REGEXREPLACE('s', l.str, 'f');
SELF.wocase := REGEXREPLACE('s', l.str, 'f', NOCASE);
SELF.uorig := l.ustr;
SELF.uwithcase := REGEXREPLACE(u'e', l.ustr, u'\u00EB');
SELF.uwocase := REGEXREPLACE(u'e', l.ustr, u'\u00EB',
        NOCASE);
END;
OUTPUT(PROJECT(inset, trans(LEFT)));

/* the result set is:
orig withcase wocase uorig uwithcase uwocase
She She fhe Eins Eins \xc3\xabins
Sells Sellf fellf Zwei Zw\xc3\xabi Zw\xc3\xabi
Sea Sea fea Drei Dr\xc3\xabi Dr\xc3\xabi
Shells Shellf fhellf Vier Vi\xc3\xabr Vi\xc3\xabr */
```

See Also: PARSE, REGEXFIND

REGROUP

REGROUP(*recset,…,recset***)**

recset	A grouped set of records. Each recset must be of exactly the same type and must contain the same number of groups.
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:

```
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);
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:

```
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

ROLLUP(*recordset, condition, transform* **[, LOCAL])**

ROLLUP(*recordset, transform, fieldlist* **[, LOCAL])**

ROLLUP(*recordset*, **GROUP**, *transform* **)**

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.
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(RECORDOF(*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_las_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:

```
rec := {STRING1 str1,STRING1 str2,STRING1 str3};
ds := DATASET([{'a', 'b', 'c'},{'a', 'b', 'c'},
               {'a', 'c', 'c'},{'a', 'c', 'd'}], rec);
rec tr(rec L, rec R) := TRANSFORM
  SELF := L;
END;
Cat(STRING1 L, STRING1 R) := L + R;
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
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:

```
inrec := RECORD
  UNSIGNED6 did;
END;

outrec := RECORD(inrec)
  STRING20 name;
  UNSIGNED score;
END;

nameRec := RECORD
  STRING20 name;
END;

finalRec := RECORD(inrec)
  DATASET(nameRec) names;
  STRING20 secondName;
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, '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( dsg,
            i1,
            LEFT.did = RIGHT.did,
            TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
            LEFT OUTER, MANY LOOKUP);
j2 := JOIN( dsg,
            i2,
            LEFT.did = RIGHT.did,
            TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
            LEFT OUTER,
            MANY LOOKUP);

j3 := JOIN( dsg,
            i3,
            LEFT.did = RIGHT.did,
            TRANSFORM(outrec, SELF := LEFT; SELF := RIGHT),
            LEFT OUTER,
            MANY LOOKUP);

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)));
```

See Also: TRANSFORM Structure, RECORD Structure, DEDUP, EXCEPT, GROUP

ROUND

ROUND(*realvalue***[**, *decimals* **])**

realvalue	The floating-point value to round.
decimals	Optional. An integer specifying the number of decimal places to round to. If omitted, the default is zero (integer result).
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:

```
SomeRealValue1 := 3.14159;
INTEGER4 MyVal1 := ROUND(SomeRealValue1);   // MyVal1 is 3
INTEGER4 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
```

See Also: ROUNDUP, TRUNCATE

ROUNDUP

ROUNDUP(*realvalue***)**

realvalue	The floating-point value to round.
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:

```
SomeRealValue := 3.14159;
INTEGER4 MyVal := ROUNDUP(SomeRealValue); // MyVal is 4

SomeRealValue := -3.9;
INTEGER4 MyVal := ROUNDUP(SomeRealValue); // MyVal is -4
```

See Also: ROUND, TRUNCATE

ROW

ROW({ *fields* } , *recstruct*)

ROW(*row* , *resultrec* **)**

ROW([*row* , **]** *transform* **)**

fields	A comma-delimited list of data values for each field in the *recstruct*, contained in curly braces ({}).
recstruct	The name of the RECORD structure defining the field layout.
row	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.
resultrec	A RECORD structure that defines how to construct the row of data, similar to the type used by TABLE.
transform	A TRANSFORM function that defines how to construct the row of data.
Return:	ROW returns a single record.

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:

```
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));
```

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:

```
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));
```

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:

```
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

  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); //omitted row parameter
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])**

left	The left record, or a nested record structure.
right	The right record, or a nested record structure.
COUNT	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.
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:

```
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([{1,'Kevin','Halligan','',33,'Mr'},
                {2,'Liz','Halligan','',33,'Dr'},
                {3,'Elizabeth','Windsor',99,'Queen'}], in1rec);
in2 := DATASET([{1,'Kevin','Halligan','',33,false},
                {2,'Liz','','Jean',33,false},
                {3,'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 0,0,1,1,0  0,1,1
//3  age                                     0,0,0,0,1  0,0,0
```

See Also: TRANSFORM Structure, JOIN

SAMPLE

SAMPLE(*recordset, interval* **[**, *which* **])**

recordset	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.
interval	The interval between records to return.
which	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.
Return:	SAMPLE returns a set of records.

The **SAMPLE** function returns a sample set of records from the nominated *recordset*.

Example:

```
MySample := SAMPLE(Person,10,1) // get every 10th record

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 := SAMPLE(SomeFile,5,1); // returns A, F, K, P, U
```

See Also: CHOOSEN, ENTH

SEQUENTIAL

[*attributename* := **] SEQUENTIAL(** *actionlist* **)**

attributename	Optional. The action name, which turns the action into an attribute definition, therefore not executed until the attributename is used as an action.
actionlist	A comma-delimited list of the actions to execute in order. These may be ECL actions or external actions.

The **SEQUENTIAL** 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. By definition, PERSIST on an attribute means the attribute is evaluated outside of any given evaluation order. Therefore, SEQUENTIAL has no effect on PERSISTed attributes.

Example:

```
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
        attributes
//that only execute when the attribute names are called as
        actions
SEQUENTIAL(Act1,PARALLEL(Act2,Act3));
//executes Act1 alone, and only when it's finished, // executes
          Act2 and Act3 together
```

See Also: PARALLEL, PERSIST

SET

SET(*recordset,field***)**

recordset	The set of records from which to derive the SET of values.
field	The field in the recordset from which to obtain the values.
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.

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
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
s2;         //results in   ['D','X','C','E','B']

//sorted unique elements
s3 := SET(SORT(t,Ltr),Ltr);
COUNT(s3);  //results in 5
s3;         //results in ['B','C','D','E','X']
```

See Also: Sets and Filters, SET OF, Set Operators, IN Operator

SIN

SIN(*angle***)**

angle	The REAL radian value for which to find the sine.
Return:	SIN returns a single REAL value.

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
```

See Also: ACOS, COS, ASIN, TAN, ATAN, COSH, SINH, TANH

SINH

SINH(*angle***)**

angle	The REAL radian value for which to find the hyperbolic sine.
Return:	SINH returns a single REAL value.

The **SINH** function returns the hyperbolic 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

HyperbolicSine45 := SINH(Angle45); //get hyperbolic sine of the angle
```

See Also: ACOS, COS, ASIN, TAN, ATAN, COSH, SIN, TANH

SIZEOF

SIZEOF(*data* **[, MAX])**

data	The name of a dataset, RECORD structure, a fully-qualified field name, or a constant string expression.
MAX	Specifies the data is variable-length (such as containing child datasets) and the value to return is the maximum size..
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:

```
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);
SIZEOF(MyRec); //result is 39
SIZEOF(MyData.F1); //result is 1
SIZEOF(MyData.F2); //result is 5
SIZEOF(MyData.F3); //result is 1
SIZEOF(MyData.F4); //result is 10
SIZEOF(MyData.F5); //result is 9 -12 chars stored in 9
        bytes
SIZEOF(MyData.F6); //result is 13 -12 chars plus null
          terminator

Layout_People := RECORD
STRING15 first_name;
STRING15 middle_name;
STRING25 last_name;
STRING2 suffix;
STRING42 street;
STRING20 city;
STRING2 st;
STRING5 zip;
STRING1 sex;
STRING3 age;
STRING8 dob;
BOOLEAN age_flag;
UNSIGNED8 __filepos { virtual(fileposition)};
END;
File_People := DATASET('ecl_training::People', Layout_People,
          FLAT);
SIZEOF(File_People); //result is 147
SIZEOF(File_People.street); //result is 42
SIZEOF('abc' + '123'); //result is 6
SIZEOF(person.per_cid); //result is 9 - Person.per_cid is
        DATA9
```

See Also: LENGTH

SOAPCALL

result **:= SOAPCALL([** *recset,* **]** *url, service, instructure,* **[** *transform,* **] DATASET(***outstructure***)** | *outstructure* **[**, *options* **]);**

SOAPCALL([*recset,* **]** *url, service, instructure,* **[** *transform,* **] [** *options* **]);**

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. The default username/password are those contained in the workunit.
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 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.
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:

RETRY(*count*)	Specifies re-attempting the call count number of times if non-fatal errors occur. If omitted, the default is three (3).
TIMEOUT(*period*)	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).

TIMELIMIT(*period*)	Specifies the total amount of time allowed for the SOAPCALL. The *period* is a real number where the integer portion specifies seconds. If omitted, the default is zero (0) indicating no limit.
HEADING(*prefix,suffix*)	Specifies tags to wrap around the XML input fields. If omitted, the default is: HEADING(",",).
XPATH(*xpath*)	Specifies the path used to access rows in the output. If omitted, the default is: 'serviceResponse/Results/Result/Dataset/Row'.
MERGE(*n*)	Specifies processing n 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 *recset* parameter is also present.
PARALLEL(*n*)	Specifies the number of concurrent threads to have processing Data Delivery Engine queries, to a maximum of 50 (the default is 2). This is intended to limit the number of concurrent sessions.
ONFAIL(*transform*)	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.
TRIM	Specifies all trailing spaces are removed from strings before output.
RESPONSE (*NOTRIM*)	Sets flag to prevent space stripping on the response.
NAMESPACE (*namespace*)	Specifies the top level *namespace* for the SOAP request.
LITERAL	Specifies the service is not necessarily implemented in ESP.
SOAPACTION (*value*)	Specifies a *value* where that *value* is a string expression typically containing a URN or URL that is required by the web *service* for proper interoperability.
LOG	If specified, writes details to the log file of the engine (hThor, Thor, or Roxie) to which the SOAPCALL is submitted.
LOG (*MIN*)	Specifies to write minimal details of the SOAPCALL to a log file.
LOG (*expression*)	Specifies to add the expression to the log when performing a SOAPCALL.
ENCODING	Specifies that the Web service being called requires a different message format, where type information is embedded in the XML.

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:

```
OutRec1 := RECORD
  STRING500 OutData{XPATH('OutData')};
  UNSIGNED4 Latency{XPATH('_call_latency')};
END;
ip := 'http://127.0.0.1:8022/';
ips := 'https://127.0.0.1:8022/';
ipspw := 'https://username:password@127.0.0.1:8022/';
svc := 'MyModule.SomeService';
```

```
//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,ipspw,svc,{STRING500 InData := 'Some Input 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;

ip := 'http://127.0.0.1:8022/';
svc:= 'MyModule.SomeService';
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)));
```

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:

```
//1 rec in, no result
SOAPCALL( 'https://127.0.0.1:8022/','MyModule.SomeService',{STRING500 InData := 'Some Input Data'});
```

```
//recordset in, no result
SOAPCALL( InputDataset,'https://127.0.0.1:8022/','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* **)]])**

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.
value	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.
JOINED	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.
joinedset	A set of records that has been previously sorted by the same value parameters as the recordset.
SKEW	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.
limit	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%).
target	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%).
THRESHOLD	Optional. Indicates the minimum size for a single part of the recordset before the SKEW limit is enforced.
size	An integer value indicating the minimum number of bytes for a single part.
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. An error occurs if the recordset has been GROUPed.
FEW	Optional. Specifies that few records will be sorted. This prevents spilling the SORT to disk if another resource-intensive activity is executing concurrently.
STABLE	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.
algorithm	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.
UNSTABLE	Optional. Specifies an unstable sort—duplicates may output in any order. Ignored if not supported by the target platform.
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, insertionsort, 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.

Thor	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.
hthor	Supports stable and unstable quicksort, stable and unstable insertionsort, 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.
Roxie	Supports unstable quicksort, stable insertionsort, 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 insertionsort implements blocking and heapmerging when there are more than 1024 rows.

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 CHOOSEN, 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 hthor), 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:

```
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***)**

SORTED(*index***)**

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:

```
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*)

n	The real number to evaluate.
Return:	SQRT returns a single real value.

The **SQRT** function returns the square root of the parameter.

Example:

```
MyRoot := SQRT(16.0);
```

See Also: POWER, EXP, LN, LOG

STEPPED

STEPPED(*index, fields* **)**

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.

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:

```
DataFile := '~RTTEST::TestStepped';
KeyFile := '~RTTEST::TestSteppedKey';
Rec := RECORD
STRING2 state;
STRING20 city;
STRING25 lname;
STRING15 fname;
END;
ds := DATASET(DataFile,
{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
PA NEW STANTON WIKER DESSIE */

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 PICHA */
```

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* **[, KEYED])**

SUM(*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. 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.
valuelist	A comma-delimited list of expressions to find the sum of. This may also be a SET of values.
KEYED	Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
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:

```
MySum := SUM(Person,Person.Salary); // total all salaries

SumVal2 := SUM(4,8,16,2,1); //returns 31
SetVals := [4,8,16,2,1];
SumVal3 := SUM(SetVals); //returns 31
```

See Also: COUNT, AVE, MIN, MAX

TABLE

TABLE(*recordset, format* **[,***expression* **[,FEW | MANY] [, UNSORTED]] [,LOCAL] [, KEYED] [, MERGE] [,SKEW(limit[, target]) [, THRESHOLD(size)]])**

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.
format	An output RECORD structure definition that defines the type, name, and source of the data for each field.
expression	Optional. Specifies a "group by" clause. You may have multiple expressions separated by commas to create a single logical "group by" 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.
FEW	Optional. Indicates that the expression will result in fewer than 10,000 distinct groups. This allows optimization to produce a significantly faster result.
MANY	Optional. Indicates that the expression will result in many distinct groups.
UNSORTED	Optional. Specifies that you don't care about the order of the groups. This allows optimization to produce a significantly faster result.
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.
KEYED	Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
MERGE	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.
SKEW	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.)
limit	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 / <number of slaves on cluster>).
target	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 / <number of slaves on cluster>).
THRESHOLD	Indicates the minimum size for a single part before the SKEW limit is enforced.
size	An integer value indicating the minimum number of bytes for a single part. Default is 1GB.
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:

```
//"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***)**

angle	The REAL radian value for which to find the tangent.
Return:	TAN returns a single REAL value.

The **TAN** function returns the tangent 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

Tangent45 := TAN(Angle45);  //get tangent of the 45 degree angle
```

See Also: ACOS, COS, ASIN, SIN, ATAN, COSH, SINH, TANH

TANH

TANH(*angle***)**

angle	The REAL radian value for which to find the hyperbolic tangent.
Return:	TANH returns a single REAL value.

The **TANH** function returns the hyperbolic tangent 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

HyperbolicTangent45 := TANH(Angle45);
                           //get hyperbolic tangent of the angle
```

See Also: ACOS, COS, ASIN, SIN, ATAN, COSH, SINH, TAN

THISNODE

THISNODE(*operation***)**

operation	The name of an attribute or in-line code that results in a DATASET or INDEX.
Return:	THISNODE returns a record set or index.

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:

```
ds := ALLNODES(JOIN(THISNODE(GetData(SomeData)),
      THISNODE(GetIDX(SomeIndex)),
      LEFT.ID = RIGHT.ID));
```

See Also: ALLNODES, LOCAL, NOLOCAL

TOPN

TOPN(*recordset, count, sorts* **[, BEST(** *bestvalues* **)] [,LOCAL])**

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.
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 CHOOSEN(SORT(*recordset*,*sorts*),*count*) but with simpler syntax that will also work for grouped *recordsets* and local operations.

Example:

```
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: CHOOSEN, SORT

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');
  //results in 4142434445
y := TOUNICODE(x,'US-ASCII');
```

See Also: FROMUNICODE, UNICODEORDER

TOXML

TOXML(*record* **)**

record	The row (record) of data to convert to an XML format.
Return:	TOXML returns a STRING.

The **TOXML** function returns a single 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:

```
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

TRANSFER

TRANSFER(*value*,*type***)**

value	An expression containing the bitmap to return.
type	The value type to return.
Return:	TRANSFER returns a single value.

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 MyVal := TRANSFER(MyInt,STRING1); //MyVal is "A" (ASCII 65)

INTEGER1 MyVal2 := (INTEGER)MyVal; //MyVal2 is 0 (zero) because
          "A" is not a numeric character
```

See Also: Type Casting

TRIM

TRIM(*string_value* **[** ,*flag* **])**

string_value	The string from which to remove spaces.
flag	Optional. Specify which spaces to remove. Valid flag values are: RIGHT (remove trailing spaces—this is the default), LEFT (remove leading spaces), LEFT, RIGHT (remove leading and trailing spaces), and ALL (remove all spaces, even those within the string_value).
Return:	TRIM returns a single value.

The **TRIM** function returns the *string_value* with all trailing and/or leading spaces removed.

Example:

```
STRING20 SomeStringValue := 'ABC';
//contains 17 trailing spaces

VARSTRING MyVal := TRIM(SomeStringValue);
// MyVal is "ABC" with no trailing spaces

STRING20 SomeStringValue := ' ABC DEF';
//contains 2 leading and 11 trailing spaces

VARSTRING MyVal := TRIM(SomeStringValue,LEFT,RIGHT);
// MyVal is "ABC DEF" with no trailing spaces
```

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:

```
SomeRealValue := 3.75;
INTEGER4 MyVal := TRUNCATE(SomeRealValue); // MyVal is 3
```

See Also: ROUND, ROUNDUP

UNGROUP

UNGROUP(*recordset* **)**

recordset	The set of previously GROUPed records.
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:

```
MyRec := RECORD
STRING20 Last;
STRING20 First;
END;

SortedSet := SORT(Person,Person.last_name); //sort by last
        name
GroupedSet := GROUP(SortedSet,last_name); //then group
        them

SecondSort := SORT(GroupedSet,Person.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:

```
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
```

See Also: FROMUNICODE, TOUNICODE

VARIANCE

VARIANCE(*recset*, *valuex* **[** , *expresssion***] [, KEYED])**

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.
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.
Return:	VARIANCE returns a single REAL value.

The **VARIANCE** function returns the (population) variance of *valuex*.

Example:

```
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' });
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},
{1.14416e+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***)**

event	A string constant containing the name of the event to wait for.

The **WAIT** action is similar to the WHEN workflow service, but may be used within conditional code.

Example:

```
//You can either do this:
action1;
action2 : WHEN('expectedEvent');

//can also be written as:
SEQUENTIAL(action1,WAIT('expectedEvent'),action2);
```

See Also: EVENT, NOTIFY, WHEN

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

Example:

```
//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***)**

condition	A conditional expression to evaluate.
Return:	WHICH returns a single value.

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:

```
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* **])**

named	A string constant containing the NAMED option scalar value to return.
type	Optional. The value type of the named scalar value result to return.
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:

```
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'}], namesRecord);

DISTRIBUTION(namesTable, surname, forename,
        NAMED('Stats'));
x := DATASET(ROW(TRANSFORM({STRING line},
SELF.line := 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:

```
d := XMLENCODE('<xml version 1><tag>data</tag>');
e := XMLDECODE(d);
```

See Also: XMLENCODE

XMLENCODE

XMLENCODE(*xml* **[**, **ALL])**

xml	The XML to encode.
ALL	Optional. Specifies including new line characters in the encoding so the text can be used in attribute definitions.
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:

```
d := XMLENCODE('<xml version 1><tag>data</tag>');
e := XMLDECODE(d);
```

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.

CHECKPOINT

attribute := *expression* **: CHECKPOINT(** *name* **)** ;

attribute	The name of the Attribute.
expression	The definition of the attribute.
name	A string constant specifying the storage name of the value.

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

DEPRECATED

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:

```
OldSort := SORT(Person,Person.per_first_name) : DEPRECATED('Use NewSort instead.');
NewSort := SORT(Person,-Person.per_first_name);

OUTPUT(OldSort);
//produces this warning:
// Attribute OldSort is marked as deprecated. Use NewSort instead.

//*****************************************
ds := DATASET(['A','B','C'],{STRING1 letter});

R1 := RECORD
  STRING1 letter;
END : DEPRECATED('Use R2 now.');

R2 := RECORD
  STRING1 letter;
  INTEGER number;
END;

R1 Xform1(ds L) := TRANSFORM
  SELF.letter := Std.Str.ToLowerCase(L.letter);
END : DEPRECATED('Use Xform2 now.');

R2 Xform2(ds L, integer C) := TRANSFORM
  SELF.letter := Std.Str.ToLowerCase(L.letter);
  SELF.number := C;
END;

OUTPUT(PROJECT(ds,Xform1(LEFT))); //produces these warnings:
// Attribute r1 is marked as deprecated. Use R2 now.
// Attribute Xform1 is marked as deprecated. Use Xform2 now.
```

FAILURE

attribute := *expression* **: FAILURE(***handler***)** ;

attribute	The name of the Attribute.
expression	The definition of the attribute.
handler	The action to run if the expression fails.

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:

```
sPeople  := SORT(Person,Person.per_first_name);
nUniques := 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])]**;

attribute	The name of the Attribute.
expression	The definition of the attribute.
cluster	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.
FEW	Optional. When the expression is a dataset or recordset, FEW 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.

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:

```
I := RANDOM() : INDEPENDENT;  //calculated once, period
G := RANDOM() : GLOBAL;       //calculated once in each graph

ds := DATASET([{1,0,0},{2,0,0}],{UNSIGNED1 rec,UNSIGNED Ival, UNSIGNED Gval });

RECORDOF(ds) XF(ds L) := TRANSFORM
  SELF.Ival := I;
  SELF.Gval := G;
  SELF := L;
END;

P1 := PROJECT(ds,XF(left)) : PERSIST('~RTTEST::PERSIST::IndependentVsGlobal1');
P2 := PROJECT(ds,XF(left)) : PERSIST('~RTTEST::PERSIST::IndependentVsGlobal2');

OUTPUT(P1);
OUTPUT(P2);    //this gets the same Ival values as P1, but different Gval values
```

See Also: GLOBAL function, INDEPENDENT

INDEPENDENT

attribute := *expression* **: INDEPENDENT**;

attribute	The name of the Attribute.
expression	The definition of the attribute.

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 for commoning up code that is shared between different arguments to a SEQUENTIAL action—normally they are evaluated completely independently.

Example:

```
File1 := 'Filename1';
File2 := 'Filename2';

SrcIP := '10.150.50.14';
SrcPath := 'c:\\InputData\\';
DestPath := '~THOR::IN::';
ESPportIP := 'http://10.150.50.12:8010/FileSpray';

DeleteOldFiles :=
   PARALLEL(FileServices.DeleteLogicalFile(DestPath+File1),
            FileServices.DeleteLogicalFile(DestPath+File2))
                           : INDEPENDENT;
SprayNewFiles :=
   PARALLEL(FileServices.SprayFixed(SrcIP,SrcPath+File1,255,
                                    '400way',DestPath+File1,
                                    -1,ESPportIP),
            FileServices.SprayFixed(SrcIP,SrcPath+File2,255,
                                   '400way',DestPath+File2,
                                   -1,ESPportIP))
                           : INDEPENDENT;
SEQUENTIAL(DeleteOldFiles,SprayNewFiles);
```

See Also: GLOBAL

ONWARNING

attribute := *expression* **: ONWARNING(***code, action***)** ;

attribute	The name of the Attribute.
expression	The definition of the attribute.
code	The number displayed in the "Code" column of the ECL IDE's Syntax Errors toolbox.
action	One of these actions: ignore, error, or warning.

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:

```
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***)] [, SINGLE] [, MULTIPLE[(***count***)]])** ;

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, the default is seven (7).
SINGLE	Optional. Specifies to keep a single PERSIST. The name of the persist file is the same as the name of the persist.
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.

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.

By definition, PERSIST on an attribute means the attribute is evaluated outside of any given evaluation order. Therefore, SEQUENTIAL has no effect on PERSISTed attributes.

Example:

```
CountPeople := COUNT(Person) : PERSIST('PeopleCount');
```

```
//Makes CountPeople available for use in all subsequent work units

sPeople := SORT(Person,Person.per_first_name) :
        PERSIST('SortPerson'),WHEN(Daily);
//Makes sPeople available for use in all subsequent work units

s1 := SORT(Person,Person.per_first_name) :
        PERSIST('SortPerson1','OtherThor');
    //run the code on the OtherThor cluster
s2 := SORT(Person,Person.per_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

PRIORITY

action **: PRIORITY(** *value* **)** ;

action	An action (typically OUTPUT) that will produce a result.
value	An integer in the range 0-100 indicating the relative importance of the action.

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:

```
OUTPUT(Person(per_st='NY')) : PRIORITY(30);
OUTPUT(Person(per_st='CA')) : PRIORITY(60);
OUTPUT(Person(per_st='FL')) : PRIORITY(90);
  //The Florida
```

See Also: OUTPUT, #OPTION

RECOVERY

attribute := *expression* **: RECOVERY(***handler* **[**, *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:

```
DoSomethingToFixIt := TRUE; //some action to repair the input

SPeople := SORT(Person,Person.per_first_name);

nUniques := DEDUP(sPeople,Person.per_first_name AND Person.address)
         :RECOVERY(DoSomethingToFixIt,2),
          FAILURE(Email.simpleSend(SystemsPersonel,
                  SystemsPersonel.email,
                  'ouch.htm'));
```

See Also: SUCCESS, FAILURE

STORED - Workflow Service

[attribute := **]** *expression* **: STORED(** *storedname* **[, FEW])** ;

attribute	Optional. The name of the Attribute.
expression	The definition of the attribute.
storedname	A string constant containing the name of the stored attribute result.
FEW	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).

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.

Example:

```
COUNT(person) : STORED('myname');
   // Name in work unit is myname,
   // stored value accessible only outside ECL
fred := COUNT(person) : STORED('fred');
   // Name in work unit is fred
fred := COUNT(person) : STORED('mindy');
   // Name in work unit is mindy
```

See Also: STORED function

SUCCESS

attribute := *expression* **: SUCCESS(***handler***)** ;

attribute	The name of the Attribute.
expression	The definition of the attribute.
handler	The action to run if the expression succeeds.

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:

```
SPeople  := SORT(Person,Person.first_name);
nUniques := COUNT(DEDUP(sPeople,Person.per_first_name AND
                 Person.address))
         : SUCCESS(Email.simpleSend(SystemsPersonel,
                  SystemsPersonel.email,'yeah.htm'));
```

See Also: FAILURE, RECOVERY

WHEN

action **: WHEN(** *event* **[,COUNT(** *repeat* **)])** ;

action	Any valid ECL Action to execute.
event	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.
COUNT	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.
repeat	An integer expression.

The **WHEN** service executes the *action* whenever the *event* occurs.

Example:

```
IF (FileServices.FileExists('test::myfile'),
   FileServices.DeleteLogicalFile('test::myfile'));
   //deletes the file if it already exists
FileServices.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

Template Language Overview

ECL was created to be the programming language for all of our HPCC 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* **);**

symbol	The name of a previously declared user-defined symbol.
expression	The string expression specifying the string to concatenate to the existing symbol contents.

The **#APPEND** statement adds the value of the *expression* to the end of the existing string contents of the *symbol*.

Example:

```
#DECLARE(MySymbol);          //declare a symbol named "MySymbol"
#SET(MySymbol,'Hello');      //initialize MySymbol to "Hello"
#APPEND(MySymbol,' World'); //make MySymbol's value "Hello World"
```

See Also: #DECLARE, #SET

#CONSTANT

#CONSTANT(*name, value* **);**

name	A string constant containing the name of the stored value.
value	An expression for the value to assign to the stored name.

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:

```
PersonCount := 0 : STORED('myname');
#CONSTANT('myname',100);
    //make stored PersonCount attribute value to 100
```

See Also: #STORED

#DECLARE

#DECLARE(*symbol* **);**

symbol	The name of the template variable.

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:

```
#DECLARE(MySymbol); //declare a symbol named "MySymbol"
#SET(MySymbol,1); //initialize MySymbol to 1
```

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:

```
#DECLARE (mstg);
#DECLARE (dmstg);
#SET (mstg, #MANGLE('SECTION_STATES/AREACODES'));

export res1 := %'mstg'%;
res1;      //res1 = 'SECTION_5fSTATES_2fAREACODES'

// Do some processing with ECL Valid Label name "mstg"

#SET (dmstg, #DEMANGLE(%'mstg'%));
export res2 := %'dmstg'%;
res2; //res2 = 'SECTION_STATES/AREACODES'
```

See Also: #MANGLE, Attribute Names

#ERROR

#ERROR(*errormessage* **);**

errormessage	A string expression containing the message to display.

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:

```
#IF(TRUE)
  #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

#EXPORT(*symbol*, *data* **);**

symbol	The name of a previously declared template variable.
data	The name of a field, RECORD structure, or dataset.

The **#EXPORT** 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:

```
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"
      position="1"
```

```
       rawtype="655364"
       size="10"
       type="string"/>
   <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="8"
       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 ;ike 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* **);**

symbol	The name of a template variable that has not been previously declared.
data	The name of a field, RECORD structure, or dataset.

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:

```
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}'%
   #IF (%'{@size}'% <> '-15' AND
        %'{@isRecord}'%='' AND
        %'{@isDataset}'%='')
+ %'{@size}'%
   #END
+ ' ' + %'{@label}'% + ';');
  #END
 #END
#END
OUTPUT('Done');
//*********************************************************
//These examples show some other possible uses of #EXPORTXML:

//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:

```
// 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'% + ' ));\n');
#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

#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*.

Example:

```
#DECLARE(fieldtype);
#DECLARE(field);

#SET(field, 'person.per_cid');

#SET(fieldtype, #GETDATATYPE(%field%));

export res := %'fieldtype'%;
res; // Output: res = 'data9'
```

See Also: Value Types

#IF

#IF(*condition* **)**

truestatements

[#ELSEIF(*condition* **)**

truestatements **]**

[#ELSE *falsestatements* **]**

#END

condition	A logical expression.
truestatements	The Template statements to execute if the condition is true.
#ELSEIF	Optional. Provides structure for statements to execute if its condition is true.
#ELSE	Optional. Provides structure for statements to execute if the condition is false.
falsestatements	Optional. The Template statements to execute if the condition is false.
#END	The #IF structure terminator.

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:

```
// 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];
```

See Also: #LOOP, #DECLARE

#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:

```
#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'%;
res;

// Output: (For 'default.YearOf')
// stg = 'YearOf Exists In Module default'
//
// Output: (For 'default.Fred')
// stg = 'Fred Does Not Exist In Module default'
```

#LOOP / #BREAK

#LOOP

[*statements* **]**

#BREAK

[*statements* **]**

#END

statements	The Template statements to execute each time.
#BREAK	Terminates the loop.
#END	The #LOOP structure terminator.

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];
```

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'%;
res1;          //res1 = 'SECTION_5fSTATES_2fAREACODES'

// Do some processing with ECL Valid Label name "mstg"

#SET (dmstg, #DEMANGLE(%'mstg'%));
export res2 := %'dmstg'%;
res2;          //res2 = 'SECTION_STATES/AREACODES'
```

See Also: #DEMANGLE, Attribute Names

#ONWARNING

#ONWARNING(*code, action***);**

code	The number displayed in the "Code" column of the ECL IDE's Syntax Errors toolbox.
action	One of these actions: ignore, error, or warning.

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:

```
#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* **);**

option	A case sensitive string constant containing the name of the option to set.
value	The value to set the option to. This may be any type of value, dependent on what the option expects to be.

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.

DFA	Deterministic Finite-state Automaton.
Fold	To turn a complex expression into a simpler equivalent one. For example, the expression "1+1" can be replaced with "2" without altering the result.
Spill	Writing intermediate result sets to disk so that memory is available for subsequent steps.
Funnel	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 "funnel."
TopN	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.
Activity	An ECL operator that takes one or more datasets as inputs.
Graph	All the Activities in a query.
Subgraph	A collection of Activities that can all be active at the same time in Thor.
Peephole	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.

Available options

The following options are generally useful:

maxRunTime	Default: none	Sets the maximum number of seconds a job runs before it times out
freezePersists	Default: false	If true, does not calculate/recalculate PERSISTed
expirePersists	Default: true	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.
defaultPersistExpiry	Default: none	If set, PERSISTs expire after the number of days specified (overriding the Sasha PersistExpiryDefault setting).
multiplePersistInstances	Default: true	If true, multiple PERSISTs are the default.

defaultNumPersistInstances	Default: none	Specifies the default number of PERSISTs. A value of -1 specifies that all copies should be kept until they expire or manually deleted.
check	Default: true	If true, check for potential overflows of records.
expandRepeatAnyAsDfa	Default: true	If true, expand ANY* in a DFA.
forceFakeThor	Default: false	If true, force code to use hthor.
forceGenerate	Default: false	If true, force .SO to be generated even if it's not worth it
globalFold	Default: true	If true, perform a global constant fold before generating.
globalOptimize	Default: false	If true, perform a global optimize.
groupAllDistribute	Default: false	If true, GROUP,ALL generates a DISTRIBUTE instead of a global SORT.
maximizeLexer	Default: false	If true, maximize the amount of work done in the lexer.
maxLength	Default: 4096	Specify maximum length of a record.
minimizeSpillSize	Default: false	If true, if a spill is filtered/deduped etc when read, reduce spill file size by splitting, filtering and then writing.
optimizeGraph	Default: true	If true, optimize expressions in a graph before generation
orderDiskFunnel	Default: true	If true, if all inputs to a funnel are disk reads, pull in
parseDfaComplexity	Default: 2000	Maximum complexity of expression to convert to a DFA.
pickBestEngine	Default: true	If true, use hthor if it is more efficient than Thor
targetClusterType	hthor|Thor|roxie	What supercomputer type are we generating code for?
topnLimit	Default: 10000	Maximum number of records to do topN on.
outputLimit	Default: 10	Sets maximum size (in Mb) of result stored in workunit.
sortIndexPayload	Default: true	Specifies sorting (or not) payload fields
workflow	Default: true	Specifies enabling/disabling workflow services.
foldStored	Default: false	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.
skipFileFormatCrcCheck	Default: false	Specifies that the CRC check on indices produces a warning and not an error.
allowedClusters	Default: none	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.
AllowAutoSwitchQueue	Default: false	If true, specifies that the workunit is automatically reassigned to execute on another available cluster listed in allowedClusters when blocked on its assigned cluster.

performWorkflowCse	Default: false	If true, specifies that the code generator automatically detects opportunities for Common Sub-expression Elimination that may be "buried" 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.
defaultSkewError	Default: none	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.
defaultSkewWarning	Default: none	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.
overrideSkewError	Default: none	If set to a value between 0.0 and 1.0, it overrides any ECL SKEW(nn) attribute values in the current job.
defaultSkewThreshold	Default: 1GB	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.
overrideSkewThreshold	Default: none	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.

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>

generateLogicalGraph	Default: false	If true, generates a Logical graph in addition to all the workunit graphs.
generateLogicalGraphOnly	Default: false	If true, generates only the Logical graph for the workunit.
logicalGraphExpandPersist	Default: true	If true, generates expands PERSISTed attributes.
logicalGraphExpandStored	Default: false	If true, generates expands STORED attributes.
logicalGraphIncludeName	Default: true	If true, generates attribute names in the header of the activity boxes.
logicalGraphIncludeModule	Default: true	If true, generates module.attribute names in the header of the activity boxes.
logicalGraphDisplayJavadoc	Default: true	If true, generates the Javadoc-style comments embedded in the ECL in place of the standard text that would be generated (see http://java.sun.com/j2se/javadoc/writingdoccomments/). Javadoc-style comments on RECORD structures or scalar attributes will not generate, as they have no graph Activity box directly associated.
logicalGraphDisplayJavadocParameters	Default: false	If true, generates information about parameters in any Javadoc-style comments.

filteredReadSpillThreshold	Default: 2	Filtered disk reads are spilled if will be duplicated more than N times.
foldConstantCast	Default: true	If true, (cast)value is folded at generate time.
foldFilter	Default: true	If true, filters are constant folded.
foldAssign	Default: true	If true, TRANSFORMs are constant folded.
foldSQL	Default: true	If true, SQL is constant folded.
optimizeDiskRead	Default: true	If true, include project and filter in the transform for a disk read.
optimizeSQL	Default: false	If true, optimize SQL.
optimizeThorCounts	Default: true	If true, convert COUNT(diskfile) into optimized version.
peephole	Default: true	If true, peephole optimize memcpy/memsets, etc.
spotCSE	Default: true	If true, look for common sub-expressions in TRANSFORMs/filters.
spotTopN	Default: true	If true, convert CHOOSEN(SORT()) into a topN activity.
spotLocalMerge	Default: false	If true, if local JOIN and both sides are sorted, generate a light-weight merge.
countIndex	Default: false	If true, optimize COUNT(index) into optimized version (also requires optimizeThorCounts).
allowThroughSpill	Default: true	If true, allow through spills.
optimizeBoolReturn	Default: true	If true, improve code when returning BOOLEAN from a function.
optimizeSubString	Default: true	If true, don't allocate memory when doing a substring.
thorKeys	Default: true	If true, allow INDEX operations in Thor.
regexVersion	Default: 0	If set to 1, specifies use of the previous regular expression implementation, which may be faster but also may exceed stack limits.
compileOptions	Default: none	Specify override compiler options (such as /Zm1000 to double the compiler heap size to workaround a heap overflow error).
linkOptions	Default: none	Specify override linker options.
optimizeProjects	Default: true	If false, disables automatic field projection/distribution optimization.
notifyOptimizedProjects	Default: 0	If set to 1, reports optimizations to named attributes. If set to 2, reports all optimizations.
optimizeProjectsPreservePersists	Default: false	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.
aggressiveOptimizeProjects	Default: false	If true, enables attempted minimization of network traffic for sorts/distributes. This option doesn't usually re-

		sult in significant benefits, but may do so in some specific cases.
percolateConstants	Default: true	If false, disables attempted aggressive constant value optimizations.

The following options are useful for debugging:

clusterSize	Default: none	Override the number of nodes in the cluster (for testing)
debugNlp	Default: false	If true, output debug information about the NLP processing to the .cpp file.
resourceMaxMemory	Default: 400M	Maximum amount of memory a subgraph can use.
resourceMaxSockets	Default: 2000	Maximum number of sockets a subgraph can use.
resourceMaxActivities	Default: 200	Maximum number of activities a subgraph can contain.
unlimitedResources	Default: false	If true, assume lots of resources when resourcing the graphs.
traceRowXML	Default: false	If true, turns on tracing in ECL Watch graphs. This should only be used with small datasets for debugging purposes.
_Probe	Default: false	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.
debugQuery	Default: false	If true, compile query using debug settings.
optimizeLevel	Default: 3 for roxie, else -1	Set the optimization level (optimizing compiler can be a lot slower...).
checkAsserts	Default: true	If true, enables ASSERT checking.

The following options are for advanced code generation use:

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.

filteredReadSpillThreshold	Default: 2	Filtered disk reads are spilled if will be duplicated more than N times.
foldConstantCast	Default: true	If true, (cast)value is folded at generate time.
foldFilter	Default: true	If true, filters are constant folded.
foldAssign	Default: true	If true, TRANSFORMs are constant folded.
foldSQL	Default: true	If true, SQL is constant folded.
optimizeDiskRead	Default: true	If true, include project and filter in the transform for a disk read.
optimizeSQL	Default: false	If true, optimize SQL.
optimizeThorCounts	Default: true	If true, convert COUNT(diskfile) into optimized version.
peephole	Default: true	If true, peephole optimize memcpy/memsets, etc.

spotCSE	Default: true	If true, look for common sub-expressions in TRANSFORMs/filters.
spotTopN	Default: true	If true, convert CHOOSEN(SORT()) into a topN activity.
spotLocalMerge	Default: false	If true, if local JOIN and both sides are sorted, generate a light-weight merge.
countIndex	Default: false	If true, optimize COUNT(index) into optimized version (also requires optimizeThorCounts).
allowThroughSpill	Default: true	If true, allow through spills.
optimizeBoolReturn	Default: true	If true, improve code when returning BOOLEAN from a function.
optimizeSubString	Default: true	If true, don't allocate memory when doing a substring.
thorKeys	Default: true	If true, allow INDEX operations in thor.
regexVersion	Default: 0	If set to 1, specifies use of the previous regular expression implementation, which may be faster but also may exceed stack limits.
compileOptions	Default: none	Specify override compiler options (such as /Zm1000 to double the compiler heap size to workaround a heap overflow error).
linkOptions	Default: none	Specify override linker options.
optimizeProjects	Default: true	If false, disables automatic field projection/distribution optimization.
notifyOptimizedProjects	Default: 0	If set to 1, reports optimizations to named attributes. If set to 2, reports all optimizations.
optimizeProjectsPreservePersists	Default: false	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.
aggressiveOptimizeProjects	Default: false	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.
percolateConstants	Default: true	If false, disables attempted aggressive constant value optimizations.

Example:

```
#OPTION('traceRowXml', TRUE);
#OPTION('_Probe', TRUE);

my_rec := RECORD
  STRING20 lname;
  STRING20 fname;
  STRING2 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} );

//**********************************
//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.
          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* **);**

symbol	The name of a previously declared user-defined symbol.
expression	The expression whose value to assign to the symbol.

The **#SET** statement assigns the value of the *expression* to the *symbol*, overwriting any previous value the symbol had contained.

Example:

```
#DECLARE(MySymbol); //declare a symbol named "MySymbol"
#SET(MySymbol,1);   //initialize MySymbol to 1
```

See Also: #DECLARE, #APPEND

#STORED

#STORED(*storedname* , *value* **);**

storedname	A string constant containing the name of the stored attribute result.
value	An expression for the new value to assign to the stored attribute.

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 := COUNT(person) : STORED('myname');
#STORED('myname',100);
     //change stored PersonCount attribute value to 100
```

See Also: STORED, #CONSTANT

#TEXT

#TEXT(*argument* **);**

argument	The MACRO parameter whose text to supply.

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:

```
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);

extractFields(people, justName, lastname, firstname);
OUTPUT(justName);
```

See Also: MACRO

#UNIQUENAME

#UNIQUENAME(*namevar* **[** ,*pattern***]);**

namevar	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.
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:

```
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);
```

```
%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(CHOOSEN(%match_out%, 50));
COUNT(%nomatch_out%);
OUTPUT(CHOOSEN(%nomatch_out%, 50));

ENDMACRO;
```

See Also: MACRO

#WARNING

#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:

```
#IF(TRUE)
  #ERROR('broken');
  OUTPUT('broken');
#ELSE
  #WARNING('maybe broken');
  OUTPUT('maybe broken');
#END;
```

See Also: #ERROR

#WORKUNIT

#WORKUNIT(*option, value* **);**

option	A string constant specifying the name of the option to set.
value	The value to set for the option.

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:

cluster	The value parameter specifies the name of the cluster on which the workunit executes.
protect	The value parameter specifies true to indicate the workunit is protected from deletion, or false if not.
name	The value parameter is a string constant specifying the workunit's jobname.
priority	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 ("super-high").
scope	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.

Example:

```
#WORKUNIT('cluster',400way);   //run the job on the 400-way cluster
#WORKUNIT('protect',true);     //disallow deletion
#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
```

External Services

SERVICE Structure

servicename **:= SERVICE [:** *defaultkeywords* **]**

prototype : *keywordlist*;

END;

servicename	The name of the service the SERVICE structure provides.
defaultkeywords	Optional. A comma-delimited list of default keywords and their values shared by all prototypes in the external service.
prototype	The ECL name and prototype of a specific function.
keywordlist	A comma-delimited list of keywords and their values that tell the ECL compiler how to access the external service.

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:

```
email := SERVICE
  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;
  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

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:

```
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 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:

```
Extern "C" _declspec(dllexport) unsigned _cdecl Countchars(const unsigned len, const char *string)
```

.SO Initialization

The following is an example prototype for an ECL (.SO) system service initialization routine:

```
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:

```
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:

```
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**).

```
  servicename := SERVICE
    functionname(parameter list) [: keyword = value];
    END;

  For example:
  email := 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:

LIBRARY	Indicates the name of the .SO module an entry point is defined in.
ENTRYPOINT	Specifies a name for the entry point. By default, the name of the entry point is the function name.
INITFUNCTION	Specifies the name of the initialization routine defined in the module containing the entry point. Currently, the initialization function is called once.
INCLUDE	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.
C	Indicates the generated C++ prototype is enclosed within an extern "C" rather than just extern.
PURE	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.
ONCE	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.
ACTION	Indicates the function has side effects and requires the optimizer to not remove calls to the function.
CONTEXT	Internal use, only. Indicates an extra internal context parameter is passed to the function.
GLOBALCONTEXT	Internal use, only. Same as CONTEXT, but there are restrictions on where the function can be used (for example, not in a TRANSFORM).
CTXMETHOD	Internal use, only. Indicates the function is actually a method of the internal code context.

Data Types

Please see the BEGINC++ documentation for data type mapping.

Passing Set Parameters to a Service

Three types of set parameters are supported: INTEGER, REAL, and STRING*n*.

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:

```
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:

```
__int64 rtlSumInt(unsigned len, __int64 * a);
```

In the source code (.cpp) file:

```
__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:

```
INTEGER8 -- __int64
INTEGER4 -- int
INTEGER2 -- short
INTEGER1 -- char
```

REAL

If you want to sum up all the elements in a set of real numbers:

To declare the function in the SERVICE structure:

```
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:

double rtlSumReal(unsigned len, double * a);

In the source code (.cpp) file:

```
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:

```
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:

```
__int64 rtlSumCharLen(unsigned len, char a[ ][20]);
```

In the source code (.cpp) file:

```
__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].

Plug-In Requirements

Plug-ins require an exported function with the following signature under Windows:

Extern "C" _declspec(dllexport) bool getECLPluginDefinition(ECLPluginDefinitionBlock *pb)

The function must fill the passed structure with correct information for the features of the plug-in. 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."

```
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 plug-in - 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 plug-in, 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

```
EXPORT ExampleLib := SERVICE
  UNSIGNED4 StringFind(CONST STRING src,
        CONST STRING tofind,
        UNSIGNED4 instance )
     : c, pure,entrypoint='elStringFind';
END;
```

.SO code module:

```
//ifdef __HQLPLUGIN_INCL
// hqlplugins.hpp : Defines standard values included
            in
// the plugin header file.
//ifdef __HQLPLUGIN_INCL
#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

//ifdef __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 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 <memory.h>
#include "examplelib.hpp"

static char buildVersion[] = "$Name$ $Id$";

#define EXAMPLELIB_VERSION "EXAMPLELIB 1.0.00"

static const char * const HoleDefinition =
  "SYSTEM\n"
  "MODULE (SYSTEM)\n"
  " FUNCTION StringFind(string src, string search,
      unsigned4 instance),unsigned4,c,name('elStringFind')\n"
  "END\n";

static const char * const EclDefinition =
  "export ExampleLib := SERVICE\n"
  " unsigned integer4 StringFind(const string src,
      const string tofind, unsigned4 instance )
          : c, pure,entrypoint='elStringFind'; \n"
  "END;";

EXAMPLELIB_API bool getECLPluginDefinition(ECLPluginDefinitionBlock *pb)
{
  if (pb->size != sizeof(ECLPluginDefinitionBlock))
     return false;
  pb->magicVersion = PLUGIN_VERSION;
  pb->version = EXAMPLELIB_VERSION " $Name$ $Id$";
  pb->moduleName = "lib_examplelib";
  pb->ECL = EclDefinition;
  pb->Hole = HoleDefinition;
  pb->flags = PLUGIN_IMPLICIT_MODULE;
  pb->description = "ExampleLib example services library";
  return true;
}
```

```
//---------------------------------------------------------------
EXAMPLELIB_API unsigned EXAMPLELIB_CALL elStringFind(unsigned srcLen,
  const char * src, unsigned hitLen, const char * hit,
  unsigned instance)
{
  if ( srcLen < hitLen )
     return 0;
  unsigned steps = srcLen-hitLen+1;
  for ( unsigned i = 0; i < steps; i++ )
     if ( !memcmp((char *)src+i,hit,hitLen) )
          if ( !--instance )
               return i+1;
  return 0;
}
```

Appendix A. Creative Commons License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

a. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c. **"Distribute"** means to make available to the public the original and copies of the Work through sale or other transfer of ownership.

d. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

e. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

f. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected

as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections; and,

b. to Distribute and Publicly Perform the Work including as incorporated in Collections.

c. For the avoidance of doubt:

   1. **Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

   2. **Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

   3. **Voluntary License Schemes**. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Adaptations. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of

the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(b), as requested.

b. If You Distribute, or Publicly Perform the Work or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. The credit required by this Section 4(b) may be implemented in any reasonable manner; provided, however, that in the case of a Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

c. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different

license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

c. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

d. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

e. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

Index

Symbols

#APPEND, 349
#BREAK, 364
#CONSTANT, 350
#DECLARE, 351
#DEMANGLE, 352
#ELSE, 362
#ELSEIF, 362
#END, 364
#ERROR, 353
#EXPAND, 354
#EXPORT, 355
#EXPORTXML, 358
#FOR, 360
#GETDATATYPE, 361
#IF, 362
#INMODULE, 363
#LOOP, 364
#MANGLE, 365
#ONWARNING, 366
#OPTION, 367
#SET, 374
#STORED, 375
#TEXT, 376
#UNIQUENAME, 377
#WARNING, 379
#WORKUNIT, 149, 380
.ECL files, 23
.SO, 384
__COMPRESSED__, 63

A

ABS, 132
ABS function, 132
ACOS, 133
ACOS function, 133
Actions as Definitions, 27
Addition, 28
AFTER, 137
AGGREGATE, 134
AGGREGATE function, 134
ALL, 95, 168, 199, 215, 249, 254, 257, 332
ALL keyword, 95
ALLNODES, 136
ALLNODES function, 136
AND, 35, 54
AND NOT, 54
ANY, 20
APPLY, 137
APPLY function, 137
arguments, 18
arithmetic operators, 28
AS, 99
ASCII, 64, 138, 252
ASCII function, 138
ASIN, 139
ASIN function, 139
ASSERT, 140
ASSERT function, 140
ASSTRING, 142
ASSTRING function, 142
ATAN, 143
ATAN function, 143, 144
ATAN2, 144
ATAN2 function, 144
ATMOST, 215, 258
AVE, 145
AVE function, 145

B

BEFORE, 137
BEGINC++, 109
BEGINC++ Structure, 109
BEST, 258, 317
BETWEEN, 35
Between Operator, 35
BIG_ENDIAN, 37
Binary values, 12
Bitshift Left, 29
Bitshift operators, 28
Bitshift Right, 29
Bitwise AND, 28
Bitwise Exclusive OR, 28
Bitwise NOT, 28
Bitwise operators, 28
Bitwise OR, 28
BLOB in INDEX, 55
Boolean, 14
BOOLEAN, 36, 84
Boolean AND, 30
Boolean Definition, 14, 17
Boolean NOT, 30, 30
Boolean OR, 30
BOOLEAN value type, 36
BUILD, 146
BUILD action, 77, 78

C

CASE, 151, 257
CASE function, 151
Casting Rules, 50
CATCH, 152
CATCH Function, 152

Character Sets, 15
CHECKPOINT, 335
CHECKPOINT workflow service, 335
Child Dataset, 69
child dataset records, 69
CHOOSE, 153
CHOOSE function, 153
CHOOSEN, 61, 69, 154
CHOOSEN function, 154
CHOOSESETS, 155
CHOOSESETS function, 155
CLUSTER, 146, 250, 251, 252, 341
CLUSTERSIZE, 156
COMBINE, 157
COMBINE function, 157
comparison operator, 29, 324
COMPRESSED, 63, 77, 250
COMPRESSION, 146
Concatenation, 33, 70
CONST, 18, 140, 383
CONST Function, 383
Constant set, 14
constant values, 14, 67
constants, 11
CORRELATION, 160
CORRELATION function, 160
COS, 162
COS function, 162
COSH, 163
COSH function, 163
COUNT, 61, 69, 164, 228, 228, 292, 347
COUNT function, 164
COUNTER, 70, 106, 130, 171, 198, 219, 234, 245, 270
COVARIANCE, 165
COVARIANCE function, 165
CRON, 167
CRON function, 167
CSV, 64, 66, 249, 251, 254, 263
CSV Files, 64, 251

D

DATA, 43
Data string, 11
DATA value type, 43
Dataset, 15
DATASET, 61, 61, 64, 71, 81, 146, 299
DATASET declaration, 80, 81
DATASET parameter, 19
DATASET parameters, 63
DECIMAL, 39
DECIMAL value type, 39
DEDUP, 146, 168, 238, 239
DEDUP function, 168, 168, 282
DEFAULT, 55
DEFINE, 170
DEFINE function, 170
Definition Name, 13
Definition Operator, 13
Definition Types, 14
Definition Visibility, 23, 125
Definitions as Actions, 27
DENORMALIZE, 171
DENORMALIZE function, 171
DEPRECATED, 336
DEPRECATED workflow service, 336
DESCEND, 275, 276
DICTIONARY, 75
DICTIONARY parameter, 19
DISTINCT statement in SQL, 168
DISTRIBUTE, 146, 174
DISTRIBUTE function, 174
DISTRIBUTED, 77, 146, 176
DISTRIBUTED function, 176
DISTRIBUTION, 177
DISTRIBUTION action, 330
DISTRIBUTION function, 177
Division, 28
Division by zero, 28
dot syntax, 25
Dynamic Files, 81

E

EBCDIC, 64, 179, 252
EBCDIC function, 179
ECL, 10
ECL definition, 13
ECL IDE, 11
ECL keywords, 13
EMBED, 114
EMBED Structure, 114
ENCODING, 300
ENCRYPT, 63, 64, 65, 250, 251, 252
ENDC++, 109
ENDEMBED, 114
ENDMACRO, 123
ENTH, 155, 180
ENTH function, 180
ENUM, 49
ENUM datatype, 49
Equivalence, 29, 89
Equivalence Comparison, 29
ERROR, 181
ERROR function, 181, 189
ESCAPE, 64
EVALUATE, 182
EVALUATE action, 182

EVALUATE function, 182
EVENT, 184
EVENT function, 184
EVENTEXTRA, 186
EVENTEXTRA function, 186
EVENTNAME, 185
EXCEPT, 96
EXCEPT keyword, 96
EXCLUSIVE, 155
EXISTS, 187
EXISTS function, 187
EXP, 188
EXP function, 188, 229
EXPIRE, 146, 222, 223, 250, 251, 252, 341
Explicit Casting, 50
EXPORT, 23, 84, 97
EXPORTed, 25
EXPORTed Definitions, 25
Expression, 13
Expressions, 28
Expressions and Operators, 28
Expressions as Actions, 27
EXTEND, 249, 254
Extended PARSE, 260
Extended PARSE Examples, 260
External Service, 384
external system services, 384

F

FAIL, 140, 152, 189
FAIL action, 181, 189
FAILCODE, 190
FAILCODE function, 190
FAILMESSAGE, 152, 191, 206, 300
FAILMESSAGE function, 191
FAILURE, 337
FAILURE workflow service, 190, 337
FALSE, 36, 107
FALSE keyword, 107
FETCH, 192
FETCH function, 192
FEW, 134, 146, 154, 197, 215, 303, 312, 338, 345
File Scope, 80
FILEPOSITION, 146
Filters, 17
FIRST, 77, 146, 257
FLAT, 63
Flat Files, 250
floating point, 38
Foreign files, 80
FORWARD, 125
forward reference, 10, 89, 170
FROM, 99
FROMUNICODE, 194
FROMUNICODE function, 194
FROMXML, 195
FROMXML function, 195
FULL ONLY, 218
FULL OUTER, 218
FUNCTION, 116
FUNCTION Structure, 116
FUNCTIONMACRO, 119
FUNCTIONMACRO Structure, 119
Functions, 18

G

GETENV, 196
GETENV function, 196
GETISVALID, 84
GLOBAL, 197, 338
GLOBAL function, 197
GLOBAL workflow service, 197, 338
GRAPH, 198
GRAPH function, 198
Greater or Equal, 29
Greater Than, 29
GROUP, 98, 157, 171, 263, 282
Group, 199
GROUP function, 199, 323
GROUP keyword, 98, 325
GROUPED, 61, 71, 215

H

HASH, 168, 200, 215
HASH function, 200
HASH32 function, 201
HASH64, 201, 202
HASH64 function, 202
HASHCRC function, 203
HASHMD5, 204
HASHMD5 function, 204
HASHRC, 203
HAVING, 205
HAVING function, 205
HEADING, 64, 252, 253, 300
heapsort, 304
Hexadecimal, 11, 12
HPCC, 10
hthor, 304
HTTPCALL, 206
HTTPCALL Function, 206
HTTPCALL Options, 206

I

IF, 207
IF function, 207

IFBLOCK, 52
IFF, 208
IFF function, 208
Implicit Casting, 50
Implicit Dataset, 82
Implicit Dataset Relationality, 82
IMPORT, 99
IMPORT AS, 99
IMPORT FROM, 99
IMPORT function, 209
IMPORTed, 25
IN, 34
In Line Dataset, 67
IN Operator, 34
in-line a set of data, 67
In-Line Dataset, 67
INDEPENDENT, 339
INDEPENDENT workflow service, 339
INDEX, 77
INDEX declaration, 77
Indexing, 15
Inline TRANSFORMs, 130, 130
INNER, 239
insertionsort, 304
INTEGER, 11, 37, 84, 385
Integer Division, 28
INTEGER value type, 37
INTERFACE, 121
interface, 310
INTERFACE Structure, 121
INTERNAL, 226
INTFORMAT, 210
INTFORMAT function, 210
ISVALID, 211
ISVALID function, 211
ITERATE, 212
ITERATE function, 212

J

JOIN, 214, 220
JOIN function, 214, 220
JOIN Set, 220
JOIN setofdatasets, 220
Join Types, 239
joincondition, 100, 215
JOINED, 303
joinflags, 215
JOINS FULL OUTER, 218

K

KEEP, 168, 215, 258
KEYDIFF, 222
KEYDIFF function, 222
KEYED, 100, 145, 160, 164, 165, 187, 215, 228, 228, 241, 269, 311, 312, 325
Keyed JOIN, 217
KEYED Keyword, 100
KEYPATCH, 223
KEYPATCH action, 222
KEYPATCH function, 223
KEYUNICODE, 224
KEYUNICODE function, 224

L

Landing Zone files, 80
LAST, 155
LEFT, 102, 171, 244
LEFT ONLY, 218, 239
LEFT OUTER, 218, 239
LENGTH, 61, 69, 225
LENGTH function, 225
Less or Equal, 29
Less Than, 29
LIBRARY, 125, 226
LIBRARY function, 149, 226
LIMIT, 215, 228
LIMIT function, 228
LINKCOUNTED, 61, 71
LITERAL, 300
LITTLE_ENDIAN, 37
LN, 229
LN function, 188, 229
LOAD, 84
LOADXML, 230
LOADXML function, 230
LOCAL, 24, 134, 146, 157, 168, 171, 180, 192, 199, 212, 215, 232, 238, 267, 269, 282, 303, 312, 317
LOCAL function, 77, 232
LOCALE, 52
LOG, 233, 300
LOG function, 233
LOGICAL Filenames, 80
Logical graphs, 369
logical operators, 30, 54
LOOKUP, 215
LOOP, 234
LOOP function, 198, 234
loopbody, 234
loopcondition, 234
loopcount, 234
loopfilter, 234
LZW, 77, 146

M

MACRO, 123
MACRO Structure, 123

MANY, 134, 197, 215, 258, 312
MAP, 236
MAP function, 236
MATCHED, 258
MATCHED ALL, 258
MAX, 258, 298
MAX function, 237
MAXCOUNT, 54
MAXLENGTH, 52, 54, 64, 84, 257
MERGE, 146, 238, 300, 312
MERGE function, 238
MERGEJOIN, 239
MERGEJOIN function, 220, 239
MIN, 241, 258
MIN function, 241
MODULE, 125
MODULE Structure, 125
Modulus Division, 28
MOFN, 239
MULTIPLE, 341
Multiplication, 28

N

N-ary merge/join, 198
Name, 13
NAMED, 21, 177, 249, 254, 255
NAMED OUTPUT, 254
Named Output Dataset, 66
NAMESPACE, 300
Natural Language Parsing, 86
Nested child datasets, 82
NOCASE, 257, 278, 279
NOFOLD, 246
NOFOLD function, 246
NOLOCAL, 242
NOLOCAL function, 242
non-procedural language, 10
NONEMPTY, 243
NONEMPTY function, 243
NORMALIZE, 244
NORMALIZE function, 244
NOROOT, 65, 146
NOSCAN, 257
NOSORT, 171, 215
Not Equal, 29
NOT MATCHED, 258
NOT MATCHED ONLY, 258
NOTHOR, 247
NOTHOR action, 247
NOTIFY, 248
NOTIFY function, 248
NOTRIM, 64, 300
NOXPATH, 249

O

ONFAIL, 152, 206, 228, 300
ONWARNING, 340
ONWARNING workflow service, 340
Operators, 28
OPT, 63, 77, 100, 253, 271
OR, 54
OUTPUT, 249, 250, 251, 252, 254, 255, 263
OUTPUT - CSV Files, 251
OUTPUT - NAMED Files, 254
OUTPUT - XML Files, 252
OUTPUT action, 249
OUTPUT Pipe Files, 254
OUTPUT Scalar Values, 255
OUTPUT Thor/Flat Files, 250
OUTPUT Workunit Files, 255
OVERRIGHT, 222
OVERWRITE, 146, 223, 250, 251, 252

P

PACKED, 52
packed decimal, 39, 39, 39
packed hexadecimal, 43
PARALLEL, 215, 234, 256, 269, 300
PARALLEL action, 255
PARALLEL function, 256
Parameter Passing, 18
parameters, 13
PARSE, 257, 258
PARSE Examples, 260
PARSE function, 91, 257
PARSE Text, 257
PARSE XML, 259
PARTITION LEFT, 215
PARTITION RIGHT, 215
Passing a DATASET parameter, 71
Passing Set Parameters, 385
PATTERN, 87
Perl regular expression, 278, 279
PERSIST, 341
PERSIST workflow service, 341
PHYSICALLENGTH, 84
Pipe, 66
PIPE, 249, 254, 263
PIPE Files, 66
PIPE function, 66, 263
POWER, 265
POWER function, 265
PREFETCH, 269, 269
PRELOAD, 63, 77, 266
PRELOAD function, 266
PRIORITY, 343
PRIORITY workflow service, 343

PROCESS, 267
PROCESS function, 267
PROJECT, 269, 271
PROJECT function, 269
PULL, 272
PULL function, 272

Q

QSTRING, 41
QSTRING string constants, 11
QSTRING value type, 41
query library, 125, 146
quicksort, 304
QUOTE, 64, 252

R

RANDOM, 273
RANDOM function, 273
RANGE, 274
RANGE function, 274
RANK, 275
RANK function, 275
RANKED, 276
RANKED function, 276
REAL, 38, 386
REAL data type, 38
REALFORMAT, 277
REALFORMAT Function, 277
realvalue, 287
RECORD, 52
record matching, 239
Record Matching Logic, 239
Record Set, 14, 15, 17, 31
Record Set Definition, 15
Record Set Operators, 31
RECORD structure, 31, 52, 65, 67, 83, 91, 93, 104, 129, 258, 258, 269, 312, 313, 325
RECORD Structure, 52
RECORDOF, 48
RECORDOF datatype, 48
RECOVERY, 344
RECOVERY workflow service, 344
recstruct, 288
regex, 278, 279
REGEXFIND, 278
REGEXFIND function, 278
REGEXREPLACE, 279
REGEXREPLACE function, 279
REGROUP, 280
REGROUP function, 280
regular expression, 87
REJECTED, 281
REJECTED function, 281, 329
Relationality, 82
REPEAT, 254, 263
Reserved Words, 13
resultrec, 288
RETRY, 206, 299
RETURN, 26, 116, 119
RIGHT, 102, 171
RIGHT ONLY, 218
RIGHT OUTER, 218
RIGHT1, 134
RIGHT2, 134
ROLLUP, 282
ROLLUP function, 282, 303
ROUND, 286
ROUND function, 286
ROUNDUP, 287
ROUNDUP function, 287
ROW, 77, 146, 288
ROW function, 127, 288
ROWDIFF, 292
ROWDIFF function, 292
ROWS(LEFT), 103
ROWS(RIGHT), 103
ROWSET, 198
ROWSET(LEFT), 198
Roxie, 232, 304
RULE, 87

S

SAMPLE, 293
SAMPLE function, 293
Scalar OUTPUT, 255
SCAN, 257
SCAN ALL, 257
Scope, 13
SCOPE, 80
SELF, 104, 128
SELF keyword, 104
SEPARATOR, 64, 252
SEQUENTIAL, 294
SEQUENTIAL function, 294
Service Function Keywords, 385
SERVICE Structure, 137, 381, 384
SERVICE structure, 386, 386
Set, 14
SET, 18, 295
Set Definition, 14, 14
SET function, 14, 295
SET OF, 46
SET OF value type, 46
Set Operators, 32
Set Ordering, 15
SET parameters, 18

Set Parameters, 385
Sets can contain definitions and expressions, 14
SHARED, 23, 105
Shared Object, 384
SIN, 296
SIN function, 296
SINGLE, 252, 341
Single Row Dataset, 68
Single-Row Dataset, 68
SINH, 297
SINH function, 297
SIZEOF, 298
SIZEOF function, 298
SKEW, 146, 174, 215, 303, 312
SKIP, 106, 128, 152, 215, 228, 258
SMART, 215
SOAPACTION, 300
SOAPCALL, 56, 299, 300
SOAPCALL Action, 301
SOAPCALL Function, 299, 300
SOAPCALL Options, 299
SORT, 303
sort algorithms, 304
SORT function, 303
SORTED, 77, 146, 214, 220, 238, 239, 307
SORTED function, 307
SQRT, 308
SQRT function, 308
square brackets, 9, 14, 18, 46, 234
STABLE, 303
STEPPED, 309
STEPPED function, 309
STORE, 84
STORED, 310, 345
STORED function, 310
STORED workflow service, 345
STREAMED, 61, 71
String, 15
STRING, 40
string constants, 11
string operator, 33
string slice, 15
STRING value type, 40
STRINGn, 386
substring, 15
Subtraction, 28
SUCCESS, 346
SUCCESS workflow service, 346
SUM, 311
SUM function, 311
SuperFile, 81

T

TABLE, 25, 312
TABLE function, 26
TABLE Function, 312
TAN, 314
TAN Function, 314
TANH, 315
TANH Function, 315
Template Language, 348
Temporary SuperFile, 81
TERMINATOR, 64, 252
THISNODE, 316
THISNODE Function, 316
THOR, 63, 249, 255
Thor, 304
THRESHOLD, 146, 215, 303, 312
TIMELIMIT, 206, 300
TIMEOUT, 206, 299
TOKEN, 87
Tomita parsing, 258
TOPN, 317
TOPN Function, 317
TOUNICODE, 318
TOUNICODE Function, 318
TOXML, 319
TOXML function, 319
TRANSFER, 320
TRANSFER Function, 320
TRANSFORM, 128
transform function, 192, 212, 218
TRANSFORM Function, 267, 269
TRANSFORM Function Requirements, 267, 269
Transform Requirement Process, 267
Transform Requirement Project, 269
Transform Requirements, 267, 269
TRANSFORM structure, 104, 106, 128, 190, 191, 288
Treating DICTIONARY as a DATASET, 73
TRIM, 206, 253, 300, 321
TRIM Function, 321
TRIM OPT, 253
TRUE, 36, 107
TRUE keyword, 107
TRUNCATE, 322
TRUNCATE Function, 322
TYPE, 83
Type Casting, 50
TYPE structure, 52, 83
Type Transfer, 50
TypeDef, 14
TypeDef Definition, 16
TYPEOF, 47
TYPEOF data type, 47

U

UDECIMALn, 39
UNGROUP, 323
UNGROUP Function, 323
UNICODE, 42, 64, 252
Unicode string, 11
UNICODE value type, 42
UNICODEORDER, 324
UNICODEORDER function, 324
UNORDERED, 215
UNSIGNED, 37, 39
UNSIGNED value type, 37
UNSORTED, 63, 312
UNSTABLE, 303
UPDATE, 146, 250, 251, 252
USE, 258
UTF-8, 11

V

Value, 14
Value Definition, 14
Value Type, 13
Value Types, 18, 36
VARIANCE, 325
VARSTRING, 44
VARSTRING string constants, 11
VARSTRING value type, 44
VARUNICODE, 45
VARUNICODE value type, 45
Virtual, 55
VIRTUAL, 125
VIRTUAL EXPORT, 97
Virtual fileposition, 55
Virtual localfileposition, 55
Virtual logicalfilename, 55
VIRTUAL SHARED, 105

W

WAIT, 327
WAIT Function, 327
WHEN, 328, 347
WHEN Function, 328
WHEN workflow service, 167, 184, 347
WHICH, 329
WHICH function, 281
WHICH Function, 329
WHOLE, 257
WIDTH, 146
WILD, 100
WILD index filter, 100
WILD Keyword, 100
WORKUNIT, 61, 330
Workunit, 66
WORKUNIT Function, 330
Workunit OUTPUT, 255
WUID, 330

X

XML, 65, 66, 249, 252, 254, 257, 263
XML Files, 252
XMLDECODE, 331
XMLDECODE Function, 331
XMLDEFAULT, 55
XMLENCODE, 332
XMLENCODE Function, 332
XOR Operator, 30
XPATH, 55, 206, 300
XPATH support, 56