

HPCC Systems®

Introduction to ECL Mini Course Training Manual - (Part 1) Concepts and Queries

HPCC Training Team



Introduction to ECL Mini Course Training Manual - (Part 1) Concepts and Queries

HPCC Training Team

Copyright © 2023 HPCC Systems. All rights reserved

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

LexisNexis and the Knowledge Burst logo are registered trademarks of Reed Elsevier Properties Inc., used under license. Other products, logos, and services may be trademarks or registered trademarks of their respective companies. All names and example data used in this manual are fictitious. Any similarity to actual persons, living or dead, is purely coincidental.

May 1, 2023 Version 8.0.0

Course Overview	7
Introduction	7
Documentation Conventions	8
The Big Picture: System Overview	9
Client Interfaces	10
ECL Middleware Layer	11
Cluster Layer	12
Auxiliary Components	13
Spraying Data to the HPCC	14
Introduction	14
Exercise 1: Spray <i>Persons</i> to THOR (Fixed)	15
Exercise 2: Spray <i>Accounts</i> to THOR (Delimited)	20
ECL Language Overview - Basics	26
Overview	26
Constants	27
Definitions	30
Basic Definition Types	32
Recordset Filtering	35
Function Definitions (Parameter Passing)	36
Definition Visibility	41
Field and Definition Qualification	43
Actions and Definitions	45
Expressions and Operators	46
Expressions and Operators	46
Logical Operators	48
Record Set Operators	49
Set Operators	51
String Operators	52
IN Operator	53
BETWEEN Operator	54
Value Types	55
BOOLEAN	55
INTEGER	56
REAL	57
DECIMAL	58
STRING	59
QSTRING	60
UNICODE	61
DATA	62
VARSTRING	63
VARUNICODE	64
SET OF	65
TYPEOF	66
RECORDOF	67
ENUM	68
Type Casting	69
Record Structures and Files	72
RECORD Structure	72
DATASET	81
Scope and Logical Filenames	96
Reserved Keywords and the MODULE Structure	99
IMPORT	100
EXPORT	102
MODULE Structure	103

SHARED	107
Getting Started with the ECL IDE	108
Exercise 3 - Repository Folders	108
Exercise 4 - Define <i>Persons</i>	112
Exercise 5 - Define <i>Accounts</i>	113
Basic Actions	114
OUTPUT	115
Aggregate Functions	124
COUNT	125
MAX	127
MIN	128
SUM	129
Basic Queries	130
Exercise 6 - Basic Queries	130
Filtering Your Data	131
Exercise 7a - Filters (Persons)	131
Exercise 7b - Filters (Accounts)	132
ECL Definitions	133
Definition Creation	133
Boolean Definitions	135
Exercise 8a - Boolean Definitions	135
Exercise 8b - More Boolean Definitions	136
SET Definitions	137
SET	138
Exercise 9 - Creating SET Definitions	140
RecordSet Definitions	141
Exercise 10a - Recordset Definition (Persons) - Part 1 of 2	141
Exercise 10b: Recordset Definition (Persons) - Part 2 of 2	141
Exercise 10c: Recordset Definitions (Accounts) - Part 1 of 2	142
Exercise 10d: Recordset Definitions (Accounts) - Part 2 of 2	142
Conditional Functions	143
IF	144
MAP	145
CASE	146
CHOOSE	147
REJECTED	148
WHICH	149
Mathematical Functions	150
ABS	151
ROUND	152
The FUNCTION Structure	153
FUNCTION Structure	154
Exercise 11: Function Definition without FUNCTION	157
Exercise 12: Function Definition using FUNCTION	158
Recordset Functions	159
EXISTS	160
SORT	161
Functional SORT Example	165
DEDUP	166
Functional DEDUP Example	169
Value Definitions	171
Exercises 13a - 13c: Using Value Definitions	171
Exercise 13a	171
Exercise 13b	172

Exercise 13c	173
More on DEDUP	174
DEDUP ALL	175
Lab Exercise Solutions	177
Exercise 4 - File_Persons.ECL	177
Exercise 5 - File_Accounts.ECL	178
Exercise 6 - BWR_BasicQueries.ECL	179
Exercise 7a: BWR_BasicPersonsFilters.ECL	180
Exercise 7b: BWR_BasicAccountsFilters.ECL	181
Exercise 8a: IsYoungMaleFloridian.ECL	182
Exercise 8b: IsOldInvoice.ECL	183
Exercise 9: Set Definitions	184
Exercise 10: Recordset Definitions	185
Exercise 11: Function Definitions	186
Exercise 12: Using a FUNCTION Structure	187
Exercise 13: Value Definitions	188
Exercise 13: Value Definitions	189
Exercise 13: Value Definitions	190

Course Overview

Introduction

This class introduces the Enterprise Control Language (ECL).

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 by using an ECL **definition**. Each definition can then be used in succeeding ECL Attribute definitions---*the language extends itself as you use it*.

The ECL IDE program used in this class is an ECL programmer's tool. It's main use is to create ECL definitions and is designed to make ECL coding as easy as possible. As such, it has many features that an end user tool would not normally have, such as the ability to "drill down" and look at raw data values. Therefore, its query functionality is meant mainly to assist in debugging and not as an end user tool for "real" queries into the database.

Definitions

ECL Definitions are the basic building blocks from which you create queries into your data and create processes for ETL (Extract, Transform, and Load) functions. You create a definition using an expression---some value calculation, logical expression, set of scalar values, or a set of data records.

Once a definition is created you can use that definition in succeeding definitions, making each succeeding definition more and more highly leveraged upon the work you have done before. This results in extremely efficient query construction.

Data and Functions

The Training Environment for this class uses two datasets (see the *Training Data* section of this document) that you will spray in Lab Exercise 1 and then define in subsequent exercises. Some pre-defined Definitions and Functions may be found in the Training Repository, but you will not need to use them in the Lab Exercises in this course.

How are these used to build Queries?

You will use the datasets that you will define with any ECL Definitions to create queries. Because definitions build upon each other, the resulting queries can be as complex as needed to obtain the result.

Once the Query is built, you send it to the High Performance Computing Cluster, or HPCC (a data-centric supercomputer designed to process massive amounts of data), which processes the query and returns the result---extremely quickly. Complex queries that may have taken weeks to format, program, and execute using old-style mainframe data-mining tools can literally execute and return the result in seconds.

This class introduces the basic syntax of ECL and its fundamental building blocks---the four basic ECL definition types and recordset filtering. These basic tools represent the underlying concepts from which any ECL query can be built...

Documentation Conventions

ECL language

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.

Names

Definitions 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.

Example Code

All example code in this document appears in the following font:

```
MyDefinitionName := COUNT(People);  
// MyDefinitionName is a user-defined Definition  
// COUNT is a built-in ECL function  
// People is the name of a dataset
```

Actions

In tutorial sections, there will be explicit actions to perform. These are all shown with a bullet to differentiate action steps from explanatory text, as shown here:

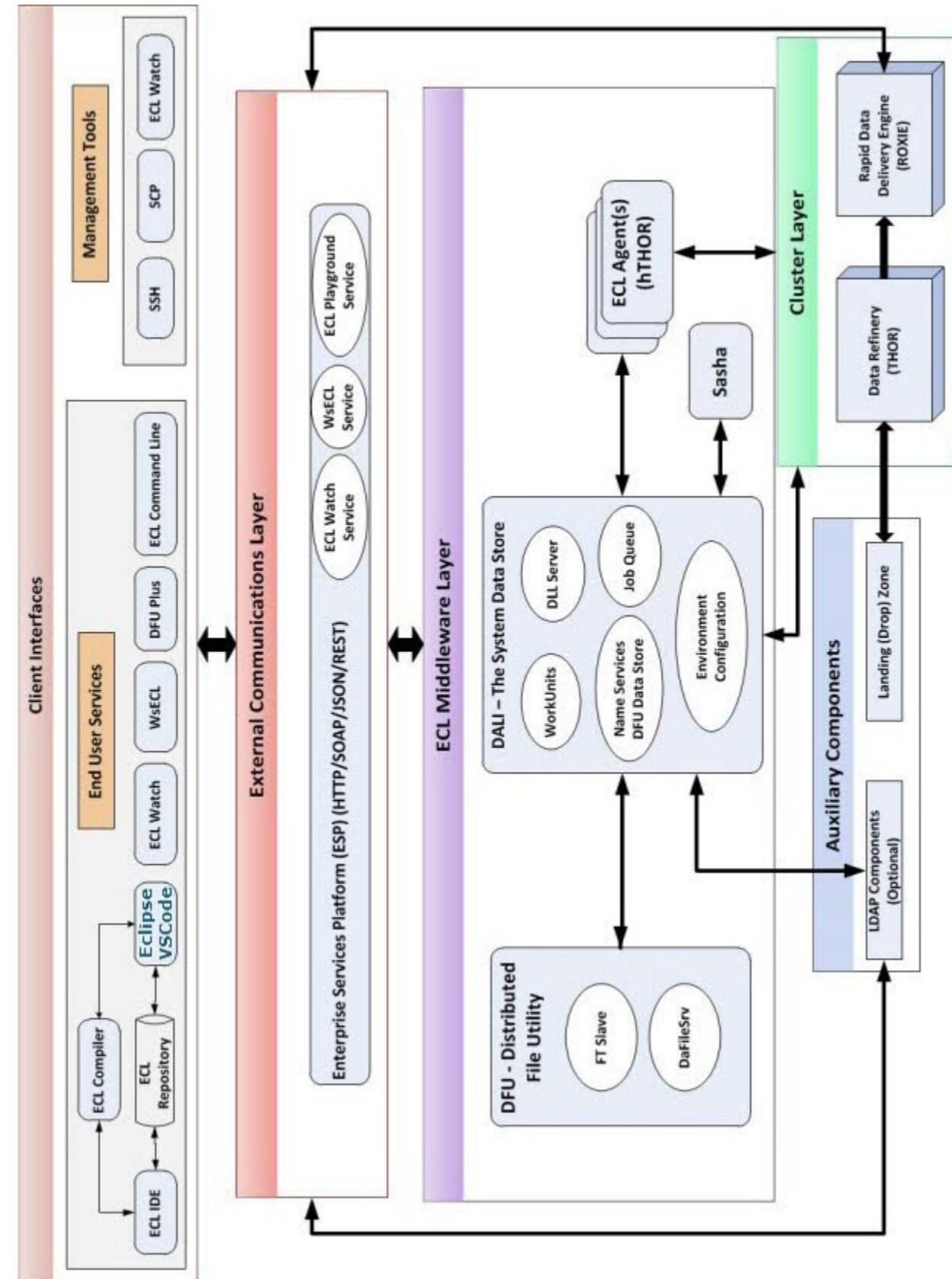
- Keyboard and mouse actions are shown in small caps, such as: **DOUBLE-CLICK**, or press the **ENTER** key.
- Onscreen items to select are shown in boldface, such as: press the **OK** button to return

ECL Language Excerpts

This manual contains discussions of a number of specific ECL features that are used in the exercises. This information has been excerpted from the *ECL Language Reference*. However, not all the information contained in that document has been placed in this one. This means that you still need to read the *ECL Language Reference* for the complete discussion of any ECL feature.

In the case of any appearance of conflict between this document and the ECL Language Reference, now or in any future release, the ruling authority is the ECL Language Reference.

The Big Picture: System Overview



Client Interfaces



These client applications interface with the High Performance Computer Cluster (HPCC) to build and send queries, and manage the HPCC systems. Our collection of code libraries allow rapid development of Client Interface Applications that interact with the HPCC components using industry-standard HTTP or SOAP interfaces, as some of the Management Tools do. Examples of these Client Interfaces include:

ECL IDE

The ECL IDE is the ECL Programmer's Development Environment, and includes capabilities for ECL Definition File Editing, Syntax Checking, Repository Access, Version Control Management, Interactive Query Execution and Result Viewing.

ECL Watch

The ECL Watch is a Web-based Query Execution, Monitoring, and File Management Tool, and includes an interface for file sprays and desprays.

WSECL

WSECL is an ESP Service that allows you to visually test published queries to THOR, ROXIE, and hTHOR targets.

DFUPlus

Command line tool for all file operations on the HPCC target cluster

ECL Command Line

Command line tool for all ECL operations.

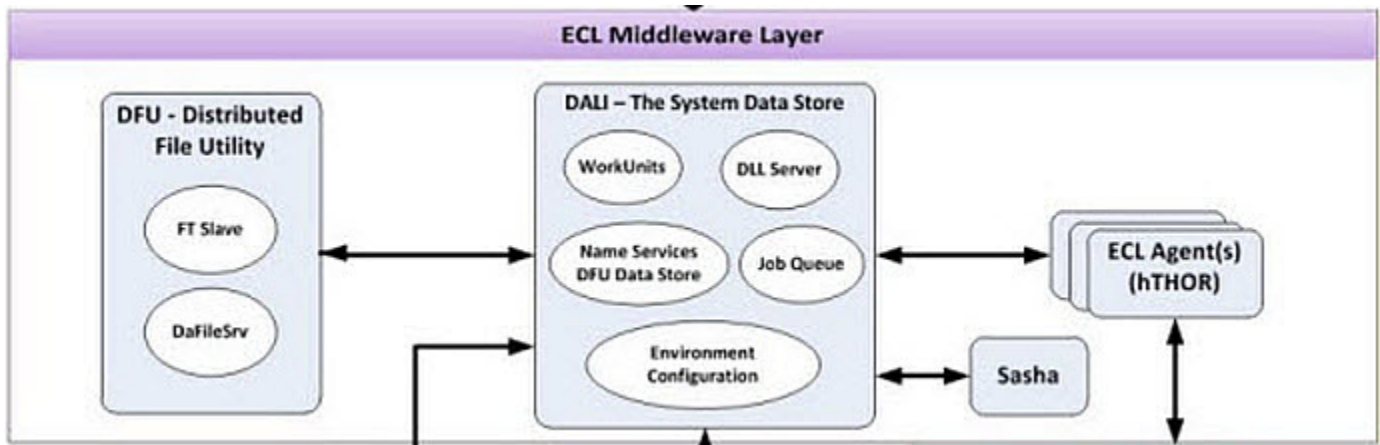
SSH

SSH stands for Secure Shell, It's a way of exchanging data over a secure connection between two remote computers.

SCP

Clusters may be accessed using any Secure Copy tool.

ECL Middleware Layer



These applications form the HPCC's metadata management layer. They act as the gateway between the actual cluster layer and the outside world. The ECL compiler, executable code generator, and job server, along with the system data store and others are all here.

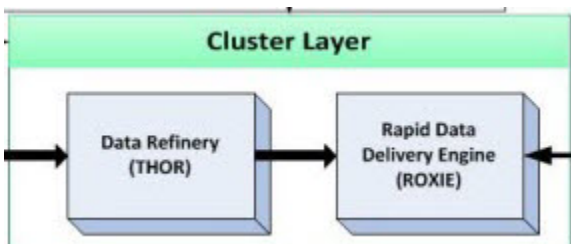
The Enterprise Control Language (ECL) is an advanced programming language supporting both query and ETL (Extract, Transform, and Load) operations. Simpler to write than other programming languages, it compiles to code that executes with maximum speed on our massively parallel processing clusters. ECL's design allows you to leverage the code you create for re-use in subsequent queries---the language extends itself as you use it.

The ECL programmer thinks in terms of writing code that specifies what result to return rather than how to get that result. This is an important concept in that, the programmer is telling the supercomputer what they want and not directing how it must be accomplished. This frees the code generator to optimize the actual execution in any way it needs to produce the desired result.

The process flow for any query is:

- The Client Interface builds the query in ECL and sends it to the Middleware via the External Communications Layer.
- The Middleware compiles the ECL, producing native executable code, then sends that code to the appropriate cluster for execution.
- The cluster executes the query and returns the result to the Middleware.
- The Middleware returns the result to the Client Interface via the External Communications Layer.

Cluster Layer



This layer encompasses the real working "guts" of the High Performance Computer Cluster (HPCC) and is comprised of two major HPCC types:

Data Refinery (THOR)

The Data Refinery, THOR, is a massively parallel computer cluster optimized for sorting, manipulating, and transforming data. It is built to handle massive ETL workloads.

Rapid Data Delivery Engine (ROXIE)

The Rapid Data Delivery Engine, ROXIE, is a massively parallel Query Processing platform, using indexed datasets and pre-compiled repeatable queries. The Rapid Data Delivery Engine delivers thousands of standard query results per second.

Auxiliary Components

Landing Zone

The Landing Zone is your data gateway to the HPCC. Importing Data from the Landing Zone to Thor is called a Spray operation, and exporting Data from Thor to Landing Zone is called a despray operation. A Landing Zone (AKA "Drop Zone") is a physical storage location defined in your system's environment. There can be one or more of these locations defined. A daemon (DaFileSrv) must be running on that server to enable file sprays and desprays. In other words, any computer (Windows, Linux, or Unix) that is IP addressable to the Thor can be configured as a Landing Zone.

Spraying Data to the HPCC

Introduction

A *spray* operation copies a data file from a Landing Zone to a Data Refinery (THOR) cluster. The term "spray" refers to the nature of the file movement -- the data in the file being sprayed is evenly partitioned across all nodes within a cluster, resulting in distributed data. All data files in the HPCC are distributed, with multiple physical parts (files) comprising a single logical entity (a dataset).

There are three *ways* to spray in the HPCC:

- The DFU interface in ECL Watch
- The DFU Plus command line utility (see the *Client Tools* manual)
- Using File Standard library functions in ECL Code (see the *Standard Library Reference*)

There are six *types* of spray operations:

1. Spraying **Fixed** length records
2. Spraying variable length records (or **Delimited**)
3. Spraying XML (Extensible Markup Language) data records
4. Spraying **Variable**, based on an input source file.
5. Spraying **BLOB** data.
6. Spraying **JSON** data.

In this course we will examine only the first two techniques, XML and the other types are covered in another advanced course.

Exercise 1: Spray *Persons* to THOR (Fixed)

Exercise Spec:

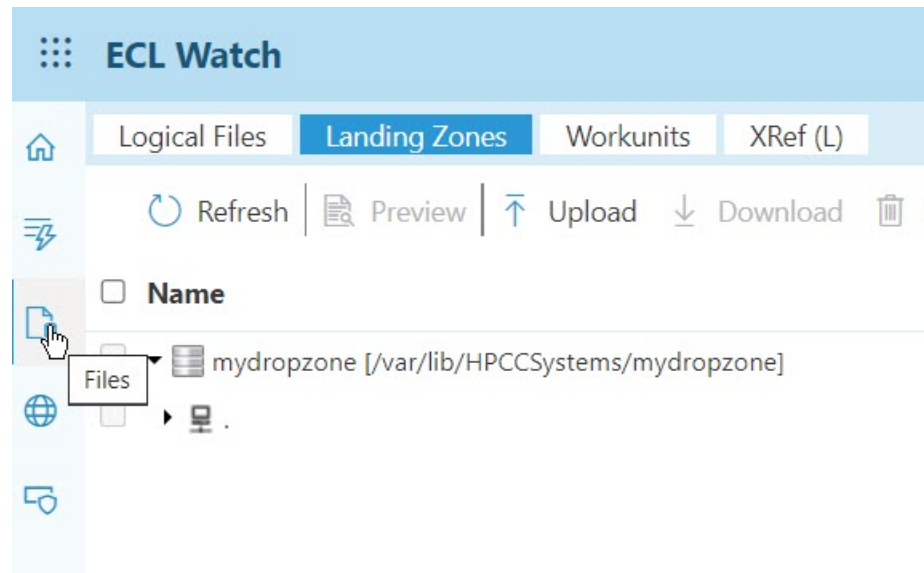
In this first Lab Exercise, we'll spray the Persons data file that we will be using in this class. Before beginning any spray operation, the following two items should be verified.

1. Verify your Landing Zone (AKA "Drop Zone")

Files sprayed to a THOR cluster are first placed on a Landing Zone. ECL Watch can locate and verify your Landing Zone location.

To access the ECL Watch, open any Internet browser and go to <http://nnn.nnn.nnn.nnn:8010> (where *nnn.nnn.nnn.nnn* is the IP address of the ESP Server for your HPCC). In GitPods, the ECL Watch is accessed via the 8010 port (click on the Ports view window):

Click on the **Files** page link in the top menu area, and verify your Landing Zone location, shown as follows:



2. Copy the file to spray to the Landing Zone

Once you know the location of your Landing Zone, you need to copy your data to that location. The only issue at this time will be to confirm that you have a connection between your data source and the landing zone. The **Landing Zone** sub menu has an *Upload* selection that can accomplish this for you, or you can use any number of utilities to do this (for example, WINSXP is a good and free tool).

File Uploader ✕

Landing Zone *

mydropzone

Machines *

.

Folder *

/

#	Type	File Name	Size
1	TXT	accounts.txt	5018.30 kb
2	TXT	persons.txt	1513.67 kb


☐ Overwrite

Upload

Steps:

1. Open ECL Watch

To access ECL Watch in GitPods, open the Ports View and click on the 8010 link shown here:

PROBLEMS	OUTPUT	TERMINAL	PORTS	COMMENTS	DEBUG CONSOLE
Port		Address			
	8010	https://8010-hpccsystemssol-introecl-tzmotwkbmo.ws-us98.gitpod.io			

2. Access the Spray: Fixed page in the Landing Zone section

Click (select) the file to Spray (**Persons**) and then click on the **Fixed** link in the top menu area, and follow the instructions described in Step 3:

3. Select your options to spray the Persons Data File

Enter or verify the following settings as directed:

Import

Group *

mythor (Thor)

Queue *

dfuserver_queue

Target Scope

MINI::XXX::Intro

Target Name	Record Length
persons	155

☐ Overwrite

☐ Replicate

☐ No Split

☐ No Common

☐ Compress

☐ Fail If No Source File

Expire in (days)

☒ Delayed replication

Import

Cancel

Target

Group:

Accept the default here (*mythor*), but you can use the drop list to select an alternate location if you have configured multiple clusters.

Target Scope:

Specify the logical target scope of the sprayed file for the DFU (Distributed File Utility). The name must start with *CLASS::*, followed by *your initials*, followed by *Intro::Persons* as in this example:

```
MINI::XXX::Intro
```

Target Name:

In this exercise, our target name is **Persons**.

Record Length:

The size of each record. The *Persons* file record length is **155**.

Options

Overwrite:

Check this box to overwrite files of the same name.

Compress:

Checking this box will compress the sprayed data (please leave this unchecked for this exercise).

Replicate:

Checking this box will replicate (back up) the sprayed data.

4. Spray the file and verify a successful operation

After entering and verifying all prompts in Step Three (3.), press the **Spray** button to begin the spray operation. You will be directed to the **DFU Workunit** page as shown below:

Result Comparison

Examine the **Percent Done** bar to watch the spray progress. When completed, you should see the following information displayed:

D20230601-193351 XML Target

 Refresh  Copy WUID |  Save  Delete | Abort

ID D20230601-193351

Cluster Name thor

Job Name

DFU Server Name mydfuserver

Queue dfuserver_queue

User

Protected ☐

Command Spray (Import)

State **finished**

Time Started 2023-06-01 19:33:51

Time Stopped 2023-06-01 19:33:51

Percent Done

Progress Message **100% Done, 0 secs left (1550/1550KB @5871KB/sec) current rate=5871KB/sec [1/1]**

Summary Message Total time taken 0 secs, Average transfer 5871KB/sec

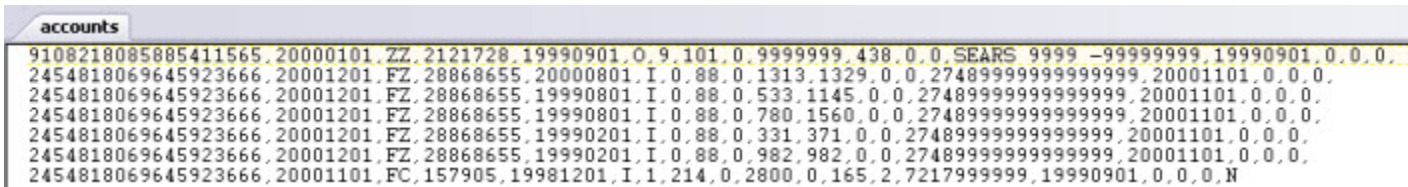
If you receive any error during this process, please see your instructor for assistance. This completes Exercise 1!

Exercise 2: Spray Accounts to THOR (Delimited)

Exercise Spec:

In this Lab Exercise, we'll complete our spray tasks with the *Accounts* file that we will be using in this class.

In Lab Exercise 1, we sprayed a file where each data record was fixed in length. In this exercise, when we examine the input file to spray, we see the following:



```
accounts
9108218085885411565,20000101,ZZ,2121728,19990901,0,9,101,0,9999999,438,0,0,SEARS 9999 -99999999,19990901,0,0,0,
2454818069645923666,20001201,FZ,28868655,20000801,I,0,88,0,1313,1329,0,0,2748999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990801,I,0,88,0,533,1145,0,0,2748999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990801,I,0,88,0,780,1560,0,0,2748999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990201,I,0,88,0,331,371,0,0,2748999999999999,20001101,0,0,0,
2454818069645923666,20001201,FZ,28868655,19990201,I,0,88,0,982,982,0,0,2748999999999999,20001101,0,0,0,
2454818069645923666,20001101,FC,157905,19981201,I,1,214,0,2800,0,165,2,7217999999,19990901,0,0,0,N
```

After examination we can conclude that we need a Delimited (comma) spray type, which also implies *variable* length records.

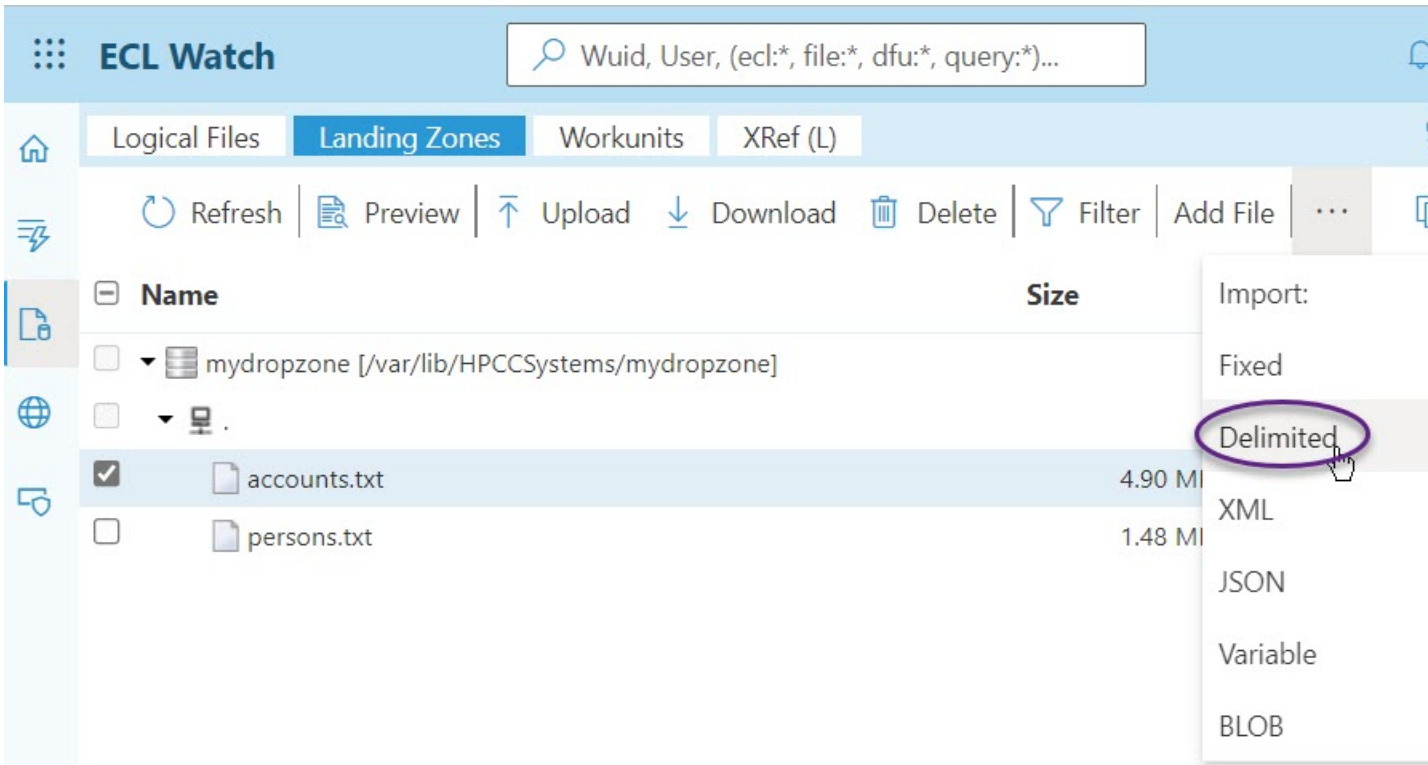
Use **Spray Delimited** to spray *any* variable-length record file that has a record delimiter.

Steps:

1. Open ECL Watch

To access ECL Watch in our training HPCC, open any Internet browser and go to <http://nnn.nnn.nnn.nnn:8010> (where *nnn.nnn.nnn.nnn* is the IP address of the ESP Server for the HPCC system. You may be asked to enter your user name and password at this time.

2. Access the Spray Delimited page in ECL Watch



Select the **Accounts** file located on your Landing Zone, and then click on the **Spray: Delimited** link in the top menu area of the **Files** page, and follow the instructions described in Step 3:

3. Select options to spray the *Accounts* Data File:

Import

Group *

mythor (Thor)

Queue *

dfuserver_queue

Target Scope

MINI::XXX::Intro

Target Name

accounts

Format

ASCII

Max Record Length

8192

Quote

"

Escape

Separators

,

Line Terminators

\n,\r\n

☐ Overwrite

☐ Replicate

☐ No Split

☒ No Common

☐ Compress

☐ Fail If No Source File

☒ Record Structure Present

☐ Quoted Terminator

Expire in (days)

☒ Delayed replication

Import

Cancel

Enter or verify the following settings as directed:

Target

Group:

Accept the default here (*mythor*), but you can use the drop list to select an alternate location if you have configured multiple THOR clusters.

Target Scope:

Specify the logical name of the sprayed file for the DFU (Distributed File Utility):

The **Target Scope** to Spray to must start with *CLASS::*, followed by *your initials*, followed by *Intro::* as in this example:

```
MINI::XXX::Intro
```

Target Name:

Verify that your target name is **Accounts**. This filename will be appended to your Name Prefix that you entered above.

Options:

Format:

Select the file format from the list. In this exercise accept the default *ASCII* setting.

Max Record Length:

Specify the length of the longest record in the file. The default is 8192. This file we are spraying has a maximum record length of 120, so you can just accept the default.

Separator:

The character, or characters used as field separators in the source file. Since a comma is used to separate multiple possible field delimiters, it must be escaped (using the leading \) to designate that the comma is the actual value to use as the separator. Accept the default as shown.

Separator:

Line Terminators:

The character(s) used as a record delimiter in the source file. Accept the default as shown.

Quote:

The character used to quote data in the source file that may contain **Separator** or **Line Terminator** characters as part of the data. Again, accept the default as shown.

Overwrite:

Check this box to overwrite files of the same name.

Compress:





Check this box to compress the sprayed data. In this exercise leave this box unchecked.

4. Spray the file and verify a successful operation

After entering and verifying all prompts in Step Three (3.), press the **Spray** button to begin the spray operation. A DFU Workunit page will automatically open.

Result Comparison

When completed, you should see the following window displayed:

D20230601-194113		XML	Target
<hr/>			
 Refresh  Copy WUID  Save  Delete Abort			
ID	D20230601-194113		
Cluster Name	thor		
Job Name	<input type="text" value="accounts"/>		
DFU Server Name	mydfuserver		
Queue	dfuserver_queue		
User			
Protected	<input type="checkbox"/>		
Command	Spray (Import)		
State	finished		
Time Started	2023-06-01 19:41:13		
Time Stopped	2023-06-01 19:41:13		
Percent Done	<hr/>		
Progress Message	100% Done, 0 secs left (5138/5138KB @65047KB/sec) current rate=65047KB/sec [
Summary Message	Total time taken 0 secs, Average transfer 65047KB/sec		

If you receive any error during this process, please see your instructor for assistance.

This completes Exercise 2!

ECL Language Overview - Basics

Overview

Enterprise Control Language (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'«'
```

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

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

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

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

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

```
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
```

Compile Time Constants

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

<code>__ECL_VERSION__</code>	A STRING containing the value of the platform version. For example, '6.4.0'
<code>__ECL_VERSION_MAJOR__</code>	An INTEGER containing the value of the major portion of the platform version. For example, '6'
<code>__ECL_VERSION_MINOR__</code>	An INTEGER containing the value of the minor portion of the platform version. For example, '4'
<code>__ECL_LEGACY_MODE__</code>	A BOOLEAN value indicating if it is being compiled with legacy IMPORT semantics.
<code>__OS__</code>	A STRING indicating the operating system to which it is being compiled. Possible values are: 'windows', 'macos', or 'linux'.
<code>__STAND_ALONE__</code>	A BOOLEAN value indicating if it is being compiled to a stand-alone executable.
<code>__TARGET_PLATFORM__</code>	A STRING containing the value of the target platform (the type of cluster the query was submitted to). Possible values are: 'roxie', 'hthor', 'thor', or 'thorlcr'.
<code>__PLATFORM__</code>	A STRING containing the value of the platform where the query will execute. Possible values are: 'roxie', 'hthor', 'thor', or 'thorlcr'.

Example:

```
IMPORT STD;
STRING14 fGetDateTimeString() :=
#IF(__ECL_VERSION_MAJOR__ > 5) or ((__ECL_VERSION_MAJOR__ = 5) AND (__ECL_VERSION_MINOR__ >= 2))
    STD.Date.SecondsToString(STD.Date.CurrentSeconds(true), '%Y%m%d%H%M%S');
#ELSE
    FUNCTION
        string14 fGetDimeTime():= // 14 characters returned
        BEGINC++
        #option action
        struct tm localtime;          // localtime in "tm" structure
        time_t timeinsecs;           // variable to store time in secs
        time(&timeinsecs);
        localtime_r(&timeinsecs,&localt);
        char temp[15];
        strftime(temp, 15, "%Y%m%d%H%M%S", &localt); // Formats the localtime to YYYYMMDDhhmmss
        strncpy(__result, temp, 14);
        ENDC++;
    RETURN fGetDimeTime();
END;
#END;
```

Definitions

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

All definitions take the form:

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

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

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

Definition Name Rules

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

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

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

Reserved Words

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

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

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

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

Definition Naming

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

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 datasets 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 PHYSICALENGTH(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;

<i>ValueType</i>	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.
<i>AliasName</i>	Names the parameter for use in the expression.
<i>DefaultValue</i>	Optional. Provides the value to use in the expression if the parameter is omitted. The <i>DefaultValue</i> may be the keyword ALL if the <i>ValueType</i> 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.
<i>expression</i>	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

<i>FunctionName</i>	The name of a function, the type of which may be passed as a parameter.
<i>parameters</i>	The parameter definitions for the <i>FunctionName</i> parameter.
<i>PrototypeName</i>	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 omissible), 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 <i>AliasName</i> clashes with a reserved word.
<i>AliasName</i>	The names of the parameter in the definition's function definition. This must be a valid label (See Definition Name Rules)
<i>value</i>	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 same folder as Definition2) you have:
IMPORT $;
//makes definitions from the current module available to this code, as needed

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

Local

A definition without either the **EXPORT** or **SHARED** keywords is available only to subsequent definitions, until the end of the next **EXPORT** or **SHARED** definition. This makes them private definitions used only within the scope of that one **EXPORT** or **SHARED** definition, which allows you to keep private any definitions that are only needed to implement internal functionality. Local definitions 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.

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

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

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

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

The `divideByZero` option can have the following values:

<code>'zero'</code>	Evaluate to 0 - the default behaviour.
<code>'fail'</code>	Stop and report a division by zero error.
<code>'nan'</code>	This is only currently supported for real numbers. Division by zero creates a quiet NaN, which will propagate through any real expressions it is used in. You can use <code>NOT ISVALID(x)</code> 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	<code>&</code>
Bitwise OR	<code> </code>
Bitwise Exclusive OR	<code>^</code>
Bitwise NOT	<code>BNOT</code>

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. When STRINGS are compared for equivalence, trailing spaces are ignored.

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 can be bypassed. However, this is not guaranteed. This is also true of the order of MAP function conditions.

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

```
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 operators are supported (all 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
-	Subtract records from a file

Example:

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

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

ExcludeThese := SecondRecSet (Num > 6);

WholeRecSet := FirstRecSet + SecondRecSet;
ResultSet := WholeRecSet - ExcludeThese;

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

Prefix Append Operator

(+) (*ds_list*) [*options*]

(+)	The prefix append operator.
<i>ds_list</i>	A comma-delimited list of record sets to append (two or more). All the record sets must have identical RECORD structures.
<i>options</i>	Optional. A comma-delimited list of options from the list below.

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

The following *options* may be used:

[, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)]

UNORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
<i>bool</i>	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.

<i>numthreads</i>	Optional. Try to evaluate this activity using <i>numthreads</i> threads.
ALGORITHM	Optional. Override the algorithm used for this activity.
<i>name</i>	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.

Example:

```
ds_1 := (+)(ds1, ds2, UNORDERED);  
    //equivalent to: ds := ds1 + ds2;  
  
ds_2 := (+)(ds1, ds2);  
    //equivalent to: ds := ds1 & ds2;  
  
ds_3 := (+)(ds1, ds2, ds3);  
    //multiple file appends are supported
```

Set Operators

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

+	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*

<i>value</i>	The value to find in the <i>value_set</i> . This is usually a single value, but if the <i>value_set</i> is a DICTIONARY with a multiple-component key, this may also be a ROW.
<i>value_set</i>	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); //false

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*

<i>SeekVal</i>	The value to find in the inclusive range.
<i>LoVal</i>	The low value in the inclusive range.
<i>HiVal</i>	The high value in the inclusive range.

The **BETWEEN** operator is shorthand for an inclusive range check using standard comparison operators (*SeekVal* \geq *LoVal* AND *SeekVal* \leq *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

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

TypeSignificant DigitsLargest ValueSmallest 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] DECIMAL n [_ y]

UDECIMAL n [_ 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.

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

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.

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

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.

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

Example:

```
UNICODE16 MyUNIStrIng := U'1234567890ABCDEF';  
    // utf-16-encoded string  
UNICODE4 MyUnicodeString := U'abcd';  
    // same as: (UNICODE)'abcd'  
UNICODE_de5 MyUnicodeString := U'abcd\353';  
    // becomes 'abcdë' with a German locale  
UNICODE_de5 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.

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

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.

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

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.

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

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*]

<i>type</i>	The value type of the data in the set. Valid value types are: INTEGER, REAL, BOOLEAN, STRING, UNICODE, DATA, or DATASET(<i>reconstruct</i>). If omitted, the <i>type</i> 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*)

<i>expression</i>	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* ,[**LOOKUP**])

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

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*] ...])

<i>type</i>	The numeric value type of the <i>values</i> . If omitted, defaults to UNSIGNED4.
<i>name</i>	The label of the enumerated <i>value</i> .
<i>value</i>	The numeric value to associate with the <i>name</i> . If omitted, the <i>value</i> is the previous <i>value</i> plus one (1). If all <i>values</i> 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', 'O', GenderEnum.Male, Pflg.Foreign+Pflg.Terrorist+Pflg.Wanted}
                        ], namesRecord);

OUTPUT(namesTable2);

myModule(UNSIGNED4 baseError, STRING x) := MODULE
  EXPORT ErrorCode := ENUM( ErrorBase = baseError,
                           ErrNoActiveTable,
                           ErrNoActiveSystem,
                           ErrFatal,
                           ErrLast);
  EXPORT reportX := FAIL(ErrorCode.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 to cast to within parentheses. That creates a casting operator. Then place that casting operator immediately to the left of the expression to cast.

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

```
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: $(\text{REAL})(\text{INTEGER})1+(\text{INTEGER})2+(\text{INTEGER})3+(\text{INTEGER})4*(1.0*(\text{REAL})5)$

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

while: $'5' + 4$ evaluates as: $5 + (\text{STRING})4$ //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
------	----	------------

Introduction to ECL Mini Course Training Manual - (Part 1) Concepts and Queries
Value Types

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 code-point (0xFFFD)
UTF8	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 code-point (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	UTF8	UTF8 representation of the value
DECIMAL	UTF8	UTF8 representation of the value, including decimal and sign
REAL	UTF8	UTF8 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
DATA	UTF8	Value is cast with no translation
UTF8	DATA	Value is cast with no translation
UTF8	UNICODE	Value is cast with no translation
UNICODE	UTF8	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;

<i>attr</i>	The name of the RECORD structure for later use in other definitions.
<i>baserec</i>	Optional. The name of a RECORD structure from which to inherit all fields. Any RECORD structure that inherits the <i>baserec</i> fields in this manner becomes compatible with any TRANSFORM function defined to take a parameter of <i>baserec</i> type (the extra <i>fields</i> will, of course, be lost).
MAXLENGTH	Optional. This option is used to create indexes that are backward compatible for platform versions prior to 3.0. Specifies the maximum number of characters allowed in the RECORD structure or field. MAXLENGTH on the RECORD structure overrides any MAXLENGTH on a field definition, which overrides any MAXLENGTH specified in the TYPE structure if the <i>datatype</i> names an alien data type. This option defines the maximum size of variable-length records. If omitted, fixed size records use the minimum size required and variable length records produce a warning. The default maximum size of a record containing variable-length fields is 4096 bytes (this may be overridden by using <i>#OPTION(maxLength,####)</i> 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).
<i>length</i>	An integer constant specifying the maximum number of characters allowed.
LOCALE	Optional. Specifies the Unicode <i>locale</i> for any UNICODE fields.
<i>locale</i>	A string constant containing a valid locale code, as specified in ISO standards 639 and 3166.
PACKED	Optional. Specifies the order of the <i>fields</i> may be changed to improve efficiency (such as moving variable-length fields after the fixed-length fields)..
<i>fields</i>	Field declarations. See below for the appropriate syntaxes.
IFBLOCK	Optional. A block of <i>fields</i> that receive "live" data only if the <i>condition</i> is met. The IFBLOCK must be terminated by an END . This is used to define variable-length records. If the <i>condition</i> expression references <i>fields</i> in the RECORD preceding the IFBLOCK, those references must use SELF. prepended to the fieldname to disambiguate the reference.
<i>condition</i>	A logical expression that defines when the <i>fields</i> within the IFBLOCK receive "live" data. If the expression is not true, the <i>fields</i> receive their declared default values. If there's no default value, the <i>fields</i> receive blanks or zeros.

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

	<i>datatype identifier</i> [{ <i>modifier</i> }] [:= <i>defaultvalue</i>];
	<i>identifier</i> := <i>defaultvalue</i> ;
	<i>defaultvalue</i> ;
	<i>sourcefield</i> ;
	<i>recstruct</i> [<i>identifier</i>] ;
	<i>sourcedataset</i> ;
	<i>childdatasetidentifier</i> [{ <i>modifier</i> }];

<i>datatype</i>	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 <i>defaultvalue</i> expression.
<i>identifier</i>	The name of the field. If omitted, the <i>defaultvalue</i> expression defines a column with no name that may not be referenced in subsequent ECL.
<i>defaultvalue</i>	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.
<i>modifier</i>	Optional. One of the keywords listed in the Field Modifiers section below.
<i>sourcefield</i>	A previously defined data field, which implicitly provides the <i>datatype</i> , <i>identifier</i> , and <i>defaultvalue</i> for the new field--inherited from the <i>sourcefield</i> .
<i>recstruct</i>	A previously defined RECORD structure. See the Field Inheritance section below.
<i>sourcedataset</i>	A previously defined DATASET or derived recordset definition. See the Field Inheritance section below.
<i>childdataset</i>	A child dataset declaration (see DATASET and DICTIONARY discussions), which implicitly defines all the fields of the child at their already defined <i>datatype</i> , <i>identifier</i> , and <i>defaultvalue</i> (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 *restruct* (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 *restruct*. 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:

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

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 (<i>length</i>) }
	{ MAXCOUNT (<i>records</i>) }
	{ 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 <i>records</i> allowed in a child DATASET field (similar to MAXLENGTH above).
{ XPATH ('tag') }	Specifies the XML or JSON <i>tag</i> that contains the data, in a RECORD structure that defines XML or JSON data. This overrides the default <i>tag</i> name (the lowercase field <i>identifier</i>). See the XPATH Support section below for details.
{ XMLDEFAULT ('value') }	Specifies a default XML <i>value</i> for the field. The <i>value</i> must be constant.
{ DEFAULT (value) }	Specifies a default <i>value</i> for the field. The <i>value</i> must be constant. This <i>value</i> will be used: <ol style="list-style-type: none"> 1. When a DICTIONARY lookup returns no match. 2. When an out-of-range record is fetched using ds[n] (as in ds[5] when ds contains only 4 records). 3. In the default records passed to TRANSFORM functions in non-INNER JOINS where there is no corresponding row. 4. When defaulting field values in a TRANSFORM using SELF = [].
{ 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] ...

<i>node</i>	Can contain wildcards.
<i>qualifier</i>	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 or JSON DATASETs reading and processing results of the SOAPCALL function, the following XPATH syntax is specifically supported:

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

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

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

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

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

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

Example:

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

```
R1 := RECORD  
  UNSIGNED1 F1; //only value type and name required  
  UNSIGNED4 F2;  
  STRING100 F3;  
END;  
  
D1 := DATASET('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>  
*/
```

XPATH can also be used to define a JSON file

```
/* a JSON file called "MyBooks.json" contains this data:  
[  
  {  
    "id" : "978-0641723445",  
    "name" : "The Lightning Thief",  
    "author" : "Rick Riordan"  
  },  
  {  
    "id" : "978-1423103349",  
    "name" : "The Sea of Monsters",  
    "author" : "Rick Riordan"  
  }  
]  
*/  
  
BookRec := RECORD  
  STRING ID {XPATH('id')}; //data from id tag -- renames field to uppercase  
  STRING title {XPATH('name')}; //data from name tag, renaming the field  
  STRING author; //data from author tag, tag name is lowercase and matches field name  
END;  
  
books := DATASET('~jd:mybooks.json',BookRec,JSON('/'));  
OUTPUT(books);
```

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

DATASET

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

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

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

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

DATASET(*row*)

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

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

DATASET(*dict*)

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

<i>attr</i>	The name of the DATASET for later use in other definitions.
<i>file</i>	A string constant containing the logical file name. See the <i>Scope & Logical Filenames</i> section for more on logical filenames.
<i>struct</i>	The RECORD structure defining the layout of the fields. This may use RECORDOF.
<i>filetype</i>	One of the following keywords, optionally followed by relevant options for that specific type of file: THOR /FLAT, CSV, XML, JSON, PIPE. Each of these is discussed in its own section, below.
<i>dataset</i>	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: <div style="background-color: #f0f0f0; padding: 5px; border: 1px solid #ccc;"> <code>ds := DATASET('filename',RECORDOF(anotherdataset), ...)</code> </div>
LOOKUP	Optional. Specifies that the file layout should be looked up at compile time. See <i>File Layout Resolution at Compile Time</i> in the <i>Programmer's Guide</i> for more details.
WORKUNIT	Specifies the DATASET is the result of an OUTPUT with the NAMED option within the same or another workunit.
<i>wuid</i>	Optional. A string expression that specifies the workunit identifier of the job that produced the NAMED OUTPUT.
<i>namedoutput</i>	A string expression that specifies the name given in the NAMED option.
<i>recordset</i>	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.
<i>recstruct</i>	Optional. The RECORD structure of the <i>recordset</i> . Omissible <u>only</u> if the <i>recordset</i> parameter is just one record or a list of in-line transform functions.
<i>row</i>	A single data record. This may be a single-record passed parameter, or the ROW or PROJECT function that defines a 1-row dataset.

<i>childstruct</i>	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).
<i>count</i>	An expression defining the number of child records. This may be a constant or a field in the enclosing RECORD structure (addressed as <i>SELF.fieldname</i>).
LENGTH	Optional. Specifies the <i>size</i> of the child records attached to the parent (for use when interfacing to external file formats).
<i>size</i>	An expression defining the size of child records. This may be a constant or a field in the enclosing RECORD structure (addressed as <i>SELF.fieldname</i>).
CHOOSEN	Optional. Limits the number of child records attached to the parent. This implicitly uses the CHOOSEN function wherever the child dataset is read.
<i>maxrecs</i>	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.
<i>struct</i>	The RECORD structure of the dataset field or parameter. This may use the RECORDOF function.
<i>dict</i>	The name of a DICTIONARY definition.
<i>count</i>	An integer expression specifying the number of records to create.
<i>transform</i>	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, JSON, PIPE**).

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

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

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

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

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

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

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

THOR/FLAT Files

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

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

THOR	Specifies the <i>file</i> 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 <i>file</i> is compressed because it is a result of the PERSIST Workflow Service or was OUTPUT with the COMPRESSED option.
__GROUPED__	Specifies the DATASET has been grouped using the GROUP function.
OPT	Optional. Specifies that using dataset when the THOR <i>file</i> doesn't exist results in an empty recordset instead of an error condition.
UNSORTED	Optional. Specifies the THOR <i>file</i> is not sorted, as a hint to the optimizer.
PRELOAD	Optional. Specifies the <i>file</i> is left in memory after loading (valid only for Rapid Data Delivery Engine use).
<i>nbr</i>	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 <i>file</i> was created by OUTPUT with the ENCRYPT option.
<i>key</i>	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,,'RTEMP::TestFile');
//write a THOR file

Ptbl := DATASET('~Thor400::RTEMP::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)] [, **__COMPRESSED__**]);**

CSV	Specifies the <i>file</i> is a "comma separated values" ASCII file.
HEADING(n)	Optional. The number of header records in the <i>file</i> . 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.
<i>f_delimiters</i>	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.
<i>r_delimiters</i>	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.
<i>characters</i>	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).
<i>esc</i>	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 <i>file</i> . 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 <i>file</i> was created by OUTPUT with the ENCRYPT option.
<i>key</i>	A string constant containing the encryption key used to create the file.
__COMPRESSED__	Optional. Specifies that the <i>file</i> is compressed because it was OUTPUT with the COMPRESSED option.

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

Example:

```
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 <i>file</i> is an XML file.
<i>xpath</i>	A string constant containing the full XPATH to the tag that delimits the records in the <i>file</i> .
NOROOT	Specifies the <i>file</i> is an XML file with no file tags, only row tags.
ENCRYPT	Optional. Specifies the <i>file</i> was created by OUTPUT with the ENCRYPT option.
<i>key</i>	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(xpath tag)}* appended to the field name (where the *xpath tag* 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'));

```

JSON Files

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

JSON	Specifies the <i>file</i> is a JSON file.
<i>xpath</i>	A string constant containing the full XPATH to the tag that delimits the records in the <i>file</i> .
NOROOT	Specifies the <i>file</i> is a JSON file with no root level markup, only a collection of objects.
ENCRYPT	Optional. Specifies the <i>file</i> was created by OUTPUT with the ENCRYPT option.
<i>key</i>	A string constant containing the encryption key used to create the file.

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

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

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

Example:

```

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

BookRec := RECORD

```

```

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

books := DATASET('~jd:mybooks.json',BookRec,JSON('/'));
OUTPUT(books);

```

PIPE Files

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

PIPE	Specifies the <i>file</i> comes from the <i>command</i> program. This is a "read" pipe.
<i>command</i>	The name of the program to execute, which must output records in the <i>struct</i> 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;
htable := DATASET([ {rms5008,72}, {rms5009,7}, {rms5010,65} ], myrec);

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

// Inline DATASET containing a Child DATASET
childPersonRecord := {STRING fname,UNSIGNED1 age};
personRecord := RECORD
    STRING20 fname;
    STRING20 lname;
    UNSIGNED2 numChildren;
    DATASET(childPersonRecord) children;
END;
1234567890123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
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),
```

```
summary.fl;                                f3 := MAX(GROUP,Y)});
```

- 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 := 1;
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

Scope and Logical Filenames

File Scope

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

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

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

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

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

```
RecStruct := {STRING line};  
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.

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

Foreign Files

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

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

The syntax looks like this:

'~foreign::<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.

Reserved Keywords and the MODULE Structure

IMPORT

IMPORT *module-selector-list*;

IMPORT *folder* **AS** *alias*;

IMPORT *symbol-list* **FROM** *folder*;

IMPORT *language*;

<i>module-selector-list</i>	A comma-delimited list of folder or file names in the repository. The dollar sign (\$) makes all definitions in the current folder available. The caret symbol (^) can be used as shorthand for the container of the current folder. Using a caret within the module specifier (such as, myModule.^) selects the container of that folder. A leading caret specifies the logical root of the file tree.
<i>folder</i>	A folder (or file name containing an EXPORTed MODULE structure) in the repository.
AS	Defines a local <i>alias</i> name for the <i>folder</i> , typically used to create shorter local names for easier typing.
<i>alias</i>	The short name to use instead of the <i>folder</i> name.
<i>symbol-list</i>	A comma-delimited list of definitions from the <i>folder</i> to make available without qualification. A single asterisk (*) may be used to make all definitions from the <i>folder</i> available without qualification.
FROM	Specifies the <i>folder</i> name in which the <i>symbol-list</i> resides.
<i>language</i>	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 <i>language</i> 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.

Examples:

```
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,
                                         //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 available a specific file
                                         // containing an EXPORTed MODULE

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

IMPORT ^ as root;                        //allows access to non-modules defined
                                         //in the root of the repository

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

EXPORT

EXPORT[**VIRTUAL**]*definition*

VIRTUAL	Optional. Specifies the <i>definition</i> is VIRTUAL. Valid only inside a MODULE Structure.
<i>definition</i>	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

MODULE Structure

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

members;

END;

<i>modulename</i>	The ECL definition name of the module.
<i>parameters</i>	Optional. The parameters to make available to all the <i>definitions</i> .
<i>inherit</i>	A comma-delimited list of INTERFACE or abstract MODULE structures on which to base this instance. The current instance inherits all the <i>members</i> from the base structures. This may not be a passed parameter.
<i>members</i>	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 <i>definitions</i> must exactly implement the EXPORTed members of the <i>interface</i> .
VIRTUAL	Optional. Specifies the MODULE defines an abstract interface whose <i>definitions</i> do not require values to be defined for them.
LIBRARY	Optional. Specifies the MODULE implements a query library <i>interface</i> definition.
<i>interface</i>	Specifies the INTERFACE that defines the <i>parameters</i> passed to the query library. The <i>parameters</i> passed to the MODULE must exactly match the parameters passed to the specified <i>interface</i> .
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 concrete member, overriding default value
    //while func remains abstract
END;

Mod3 := MODULE(Mod1) //a fully concrete 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](#)

SHARED

SHARED[**VIRTUAL**]*definition*

VIRTUAL	Optional. Specifies the <i>definition</i> is VIRTUAL. Valid only inside a MODULE Structure.
<i>definition</i>	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

Getting Started with the ECL IDE

Exercise 3 - Repository Folders

Exercise Spec:

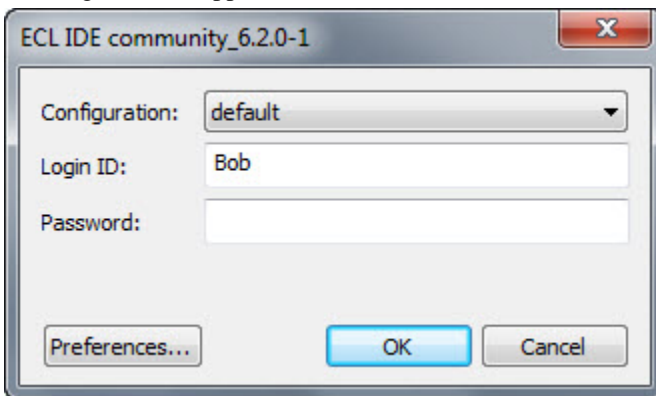
Get familiar with the ECL IDE. Examine the IDE preferences and then control your viewing display of the IDE support windows. Finally, create a folder in the IDE Repository to store all subsequent ECL code that you will write in this class.

Steps:

Start the ECL IDE, and Login

1. From the Windows Program Menu, locate and start the ECL IDE.

The Login Screen appears:



2. Click on the **Preferences** button and review the following options on each tab:

Server:

3. Verify that the correct IP address is entered (as provided by your instructor)

Editor:

4. Check the **Line Numbers** and **Open MDI Children Maximized** check boxes.

Colors and Results:

5. Accept all defaults on these tabs.

Compiler:

If you are using the Community Edition, you can designate additional **ECL Folders** as needed. In the Enterprise Edition, this tab is currently not applicable.

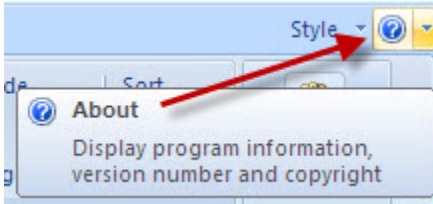
6. On your local training machine, verify that a *Training* folder exists on your root *C* drive. If it does not exist, use the Windows Explorer to create one. Next, in the **ECL folders** text box, press the **Add** button, and add the *C:\Training* folder to the **ECL folder** text box.

Other:

7. Accept all defaults on this tab.

8. After verifying all settings, press the **OK** button to save your options and close the **Preferences** dialog and then complete the Login process.

After a successful Login, this might be a good time to review the ECL IDE documentation provided with your Community or Enterprise Edition. To access this PDF, locate the Help button in the upper right corner of the ECL IDE:



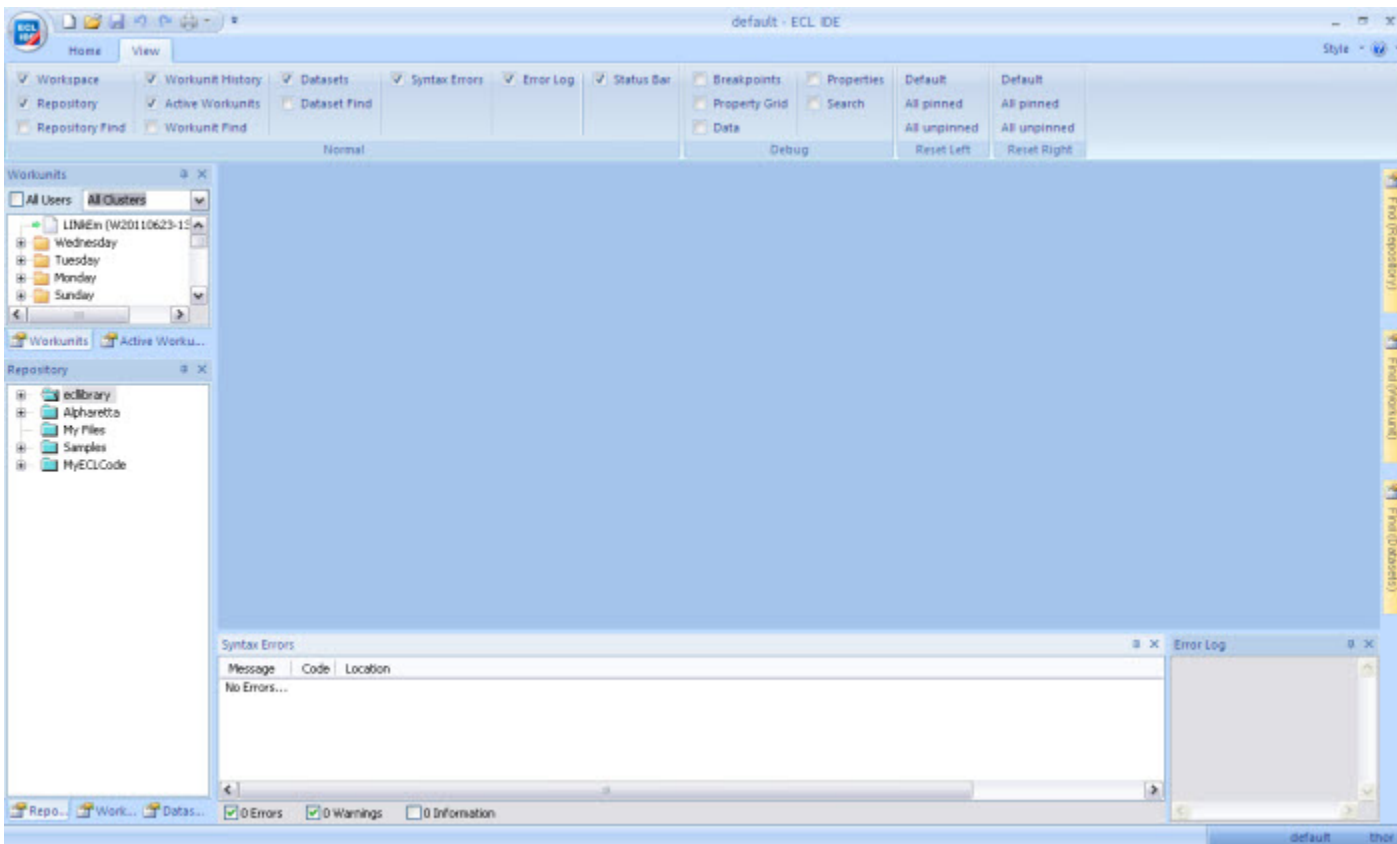
From here, you can access the *Client Tools* PDF, where you will find a chapter on the ECL IDE and the Preferences window.

IDE Display Options

At this point, you established a connection to the HPCC Training environment. Let's adjust our visual settings to allow for better consistency throughout the remaining lab exercises.

1. In the Ribbon Bar Menu, select the **View** tab, and then select **Default** at the top of the **Reset Left** ribbon tab.

Your screen should now look like this:



All the toolboxes are now docked in their default left positions. The toolboxes may be re-sized, re-positioned, docked, floating, or autohide. The exact configuration you create is saved between sessions.

2. Close the **Error Log** window in the lower right corner, and then verify in the ribbon bar that no **Debug** windows are opened (they should all be closed by default).

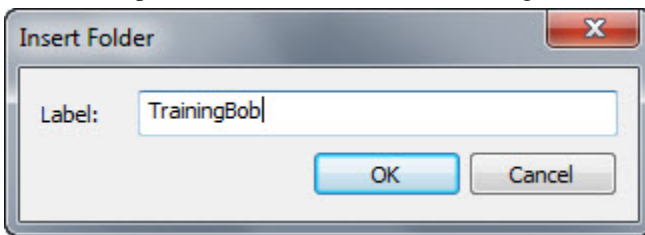
Now we are ready to create our Repository folder!

Create a Repository Folder

To create a new folder for your ECL definition files, you must first name it. Folder and file names in ECL cannot contain spaces, so it is important that you do not use any. Valid ECL folder and file names must begin with a letter and may only contain letters, numbers, underscores (_), and dollar sign characters.

1. RIGHT-CLICK anywhere in the **Repository** tree, and select **Insert Folder** from the popup menu.
2. In the **Label** entry control, type in *Training* followed by your *full name* --- there should be no spaces between the words; spaces are not allowed in folder names (i.e., *TrainingBillJones*).

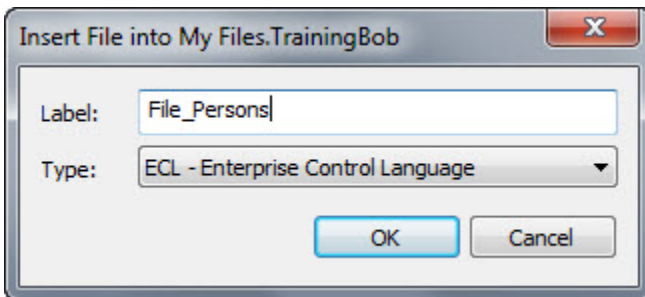
The following screen shots and action steps will use *Training<YourName>* for this (Example: *TrainingBob*). Your screens will, of course, display your own name just as you have typed it in here. In addition, all subsequent Lab Exercise Steps and Solutions will use this naming convention where appropriate.



3. Press the **OK** button.
4. Scroll through the **Repository** window. You should now see your new folder in the **Repository**.
5. RIGHT-CLICK on *TrainingYourName* and select **Insert File** from the popup menu.
6. In the **Label** entry control, type:

File_Persons

You are naming the ECL Definition File here for use in later definitions.



7. Verify that the file **Type** is *ECL – Enterprise Control Language*, and then press the **OK** button.

Result Comparison

A new window appears (or a new tab, if all windows are opening maximized) with *TrainingYourName.Persons* in the title bar (or tab). This is the ECL Editor, and the main window for creating and editing ECL definitions. The Repository tree under *TrainingYourName* is also expanded, so you can see the new file in the tree.

Notice that the text control already contains a default ECL definition. It is extremely important that the definition name in the ECL code exactly match the name that you gave it in the **Insert File** window. That's why the ECL IDE does the initial typing for you.

This completes this Lab exercise!

Leave this window open and proceed directly to Lab Exercise 4.

Exercise 4 - Define *Persons*

Exercise Spec:

In this exercise we will define the MODULE, RECORD structure, and DATASET definition for the Persons table sprayed in Exercise 1.

Steps:

1. Use the definition file that you created in **Lab Exercise 3** as a starting point. This file should be in the *TrainingYourName* folder and named *File_Persons*.
2. Create the MODULE structure for *File_Persons* and make sure it is EXPORTed (the default).
3. Inside the MODULE, create the RECORD definition using the following layouts of the fields. The recommended name of the RECORD definition is **Layout**. EXPORT this definition.
4. The layout of the fields is:

Description	Type	Field Name
Individual Identifier	UNSIGNED 8-byte integer	ID
First Name	15-character string	FirstName
Last Name	25-character string	LastName
Middle Name	15-character string	MiddleName
Name Suffix (SR, JR, 1-9)	2-character string	NameSuffix
Date Added in YYYYMMDD format	8-character string	FileDate
3-digit numeric code	UNSIGNED 2-byte integer	BureauCode
Marital Status (blank)	1-character string	MaritalStatus
Sex (M, F, N, U)	1-character string	Gender
Number of dependents	UNSIGNED 1-byte integer	DependentCount
Date of Birth (YYYYMMDD format)	8-character string	BirthDate
Address	42-character string	StreetAddress
City	20-character string	City
State	2-character string	State
5-digit zip code	5-character string	ZipCode

4. Create the DATASET definition, using the name of the file that you sprayed in Lab Exercise 1, the RECORD definition that you created above, and determine the type of file defined (THOR, CSV or XML). Make sure to EXPORT this definition. Name this DATASET definition **File**.

Best Practices Hint

It is always recommended to include your RECORD definition and corresponding DATASET in the same definition file. Using a MODULE , both DATASET definition and the RECORD structure will be EXPORTed,.

Result Comparison

Verify that your definitions and corresponding results look reasonable by opening a New Builder Window and enter the following ECL Code:

```
IMPORT TrainingYourName;  
TrainingYourName.File_Persons.File;
```

In the Workunit's Result tab, you should see names and address fields formatted properly and numeric fields displayed with only numbers.

This completes this Lab Exercise!

Exercise 5 - Define Accounts

Exercise Spec:

In this exercise we will define the MODULE, RECORD structure and DATASET definition for the *Accounts* file sprayed in *Exercise 2*.

Steps:

1. Create a new ECL definition file as a starting point. This file should be in the *TrainingYourName* folder and named **File_Accounts**. This should be an EXPORTed MODULE structure, similar to the type we created in the last exercise.
2. Inside the MODULE, create the RECORD definition using the following layouts of the fields. The recommended name of the RECORD definition is **Layout**. EXPORT this definition.
3. The layout of the fields is:

Description:	Type:	Field Name:
Foreign Key to Persons record	UNSIGNED 8-byte integer	PersonID
Date added in YYYYMMDD format	8-character string	ReportDate
2-letter industry code	2-character string	IndustryCode
Member ID	UNSIGNED 4-byte integer	Member
Account Open Date in YYYYMMDD format	8-character string	OpenDate
Open, Invoice, Receivable	1-character string	TradeType
0-9, Z, *	1-character string	TradeRate
0-255	UNSIGNED 1-byte integer	Narr1
0-255	UNSIGNED 1-byte integer	Narr2
Credit Limit in Dollars	UNSIGNED 4-byte integer	HighCredit
Account Balance	UNSIGNED 4-byte integer	Balance
Payment Terms (days)	UNSIGNED 2-byte integer	Terms
Receivables code (0,1,2)	UNSIGNED 1-byte integer	TermTypeR
Account Number	20-character string	AccountNumber
YYYYMMDD last activity	8-character string	LastActivityDate
30 late Boolean	UNSIGNED 1-byte integer	Late30Day
60 late flag	UNSIGNED 1-byte integer	Late60Day
90 late flag	UNSIGNED 1-byte integer	Late90Day
N or M	1-character string	TermType

4. Create the DATASET definition inside of the MODULE, using the name of the file that you sprayed in *Lab Exercise 2*, the RECORD definition that you created above, and determine the type of file defined (THOR, CSV or XML). **Make sure to EXPORT this definition and name it File.**

Best Practices Hint

It is always recommended to include your RECORD definition and corresponding DATASET in the same definition file. Using a MODULE, both DATASET definition and the RECORD structure will be EXPORTed,

Result Comparison

Verify your definitions and ensure reasonable results by opening a New Window and enter the following ECL Code:

```
IMPORT TrainingYourName;  
TrainingYourName.File_Accounts.File;
```

Press **Submit** and view your data in the ECL IDE Results window.

This completes this Lab Exercise!

Basic Actions

OUTPUT

```
[attr := ] OUTPUT(recordset [, [format] [, file [thorfileoptions] ] [, NOXPATH] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] ] [, ALGORITHM( name ) ] );
```

```
[attr := ] OUTPUT(recordset, [format] , file , CSV [ (csvoptions) ] [csvfileoptions] [, NOXPATH] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] ] [, ALGORITHM( name ) ] );
```

```
[attr := ] OUTPUT(recordset, [format] , file , XML [ (xmloptions) ] [xmlfileoptions] [, NOXPATH] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] ] [, ALGORITHM( name ) ] );
```

```
[attr := ] OUTPUT(recordset, [format] , file , JSON [ (jsonoptions) ] [jsonfileoptions] [, NOXPATH] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] ] [, ALGORITHM( name ) ] );
```

```
[attr := ] OUTPUT(recordset, [format] , PIPE( pipeoptions [, NOXPATH] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] ] [, ALGORITHM( name ) ] );
```

```
[attr := ] OUTPUT(recordset [, format] , NAMED( name ) [, EXTEND] [, ALL] [, NOXPATH] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] ] [, ALGORITHM( name ) ] );
```

```
[attr := ] OUTPUT( expression [, NAMED( name ) ] [, NOXPATH] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] ] [, ALGORITHM( name ) ] );
```

```
[attr := ] OUTPUT( recordset , THOR [, NOXPATH] [, UNORDERED | ORDERED( bool ) ] [, STABLE | UNSTABLE ] [, PARALLEL [ ( numthreads ) ] ] [, ALGORITHM( name ) ] );
```

<i>attr</i>	Optional. The action name, which turns the action into a definition, therefore not executed until the <i>attr</i> is used as an action.
<i>recordset</i>	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.
<i>format</i>	Optional. The format of the output records. If omitted, all fields in the <i>recordset</i> 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.
<i>file</i>	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.
<i>thorfileoptions</i>	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 <i>format</i> or the RECORD structure of the <i>recordset</i> are ignored and field names are used instead. This allows control of whether XPATHs are used for output, so that XPATHs that were meant only for xml or json input can be ignored for output.
UNORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
<i>bool</i>	When False, specifies the output record order is not significant. When True, specifies the default output record order.

STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
<i>numthreads</i>	Optional. Try to evaluate this activity using <i>numthreads</i> threads.
ALGORITHM	Optional. Override the algorithm used for this activity.
<i>name</i>	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
CSV	Specifies the file is a field-delimited (usually comma separated values) ASCII file.
<i>csvoptions</i>	Optional. A comma-delimited list of options defining how the file is delimited.
<i>csvfileoptions</i>	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.
<i>xmloptions</i>	Optional. A comma separated list of options that define how the output XML file is delimited.
<i>xmlfileoptions</i>	Optional. A comma-delimited list of options valid for an XML file (see the section below for details).
JSON	Specifies the file is output as JSON data with the name of each field in the format becoming the JSON tag for that field's data.
<i>jsonoptions</i>	Optional. A comma separated list of options that define how the output JSON file is delimited.
<i>jsonfileoptions</i>	Optional. A comma-delimited list of options valid for an JSON file (see the section below for details).
PIPE	Indicates the specified command executes with the <i>recordset</i> provided as standard input to the command. This is a "write" pipe.
<i>pipeoptions</i>	The name of a program to execute, which takes the <i>file</i> 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.
<i>name</i>	A string constant containing the result label. This must be a compile-time constant and meet the attribute naming requirements. This must be a valid label (See Definition Name Rules)
EXTEND	Optional. Specifies appending to the existing NAMED result <i>name</i> 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 <i>recordset</i> are output to the ECL IDE.
<i>expression</i>	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).
<i>target</i>	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.
<i>key</i>	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.
<i>days</i>	Optional. The number of days from last file read after which the file may be automatically deleted. If EXPIRE is specified without number of days, it defaults to use the ExpiryDefault setting in Sasha.

This form writes the *recordset* to the specified *file* in the specified *format*. If the *format* is omitted, all fields in the *recordset* are output. If the *file* is omitted, then the result is sent back to the requesting program (usually the ECL IDE or the program that sent the SOAP query to a Roxie).

Example:

```
OutputFormat1 := RECORD
  People.firstname;
  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 People 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)] [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).
<i>target</i>	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.
<i>key</i>	A string constant containing the encryption key to use to encrypt the data.
COM-PRESSED	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.
<i>days</i>	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][, FORMAT(stringfunction)])

SEPARATOR(delimiters)

TERMINATOR(delimiters)

QUOTE([delimiters])

ASCII | EBCDIC | UNICODE

HEADING	Specifies file headers and footers.
<i>headertext</i>	Optional. The text of the header record to place in the file. If omitted, the field names are used.
<i>footertext</i>	Optional. The text of the footer record to place in the file. If omitted, no <i>footertext</i> is output.

SINGLE	Optional. Specifies the <i>headertext</i> is written only to the beginning of part 1 and the <i>footertext</i> is written only at the end of part n (producing a "standard" CSV file). If omitted, the <i>headertext</i> and <i>footertext</i> are placed at the beginning and end of each file part (useful for producing complex XML output).
FORMAT	Optional. Specifies the headertext should be formatted using the <i>stringfunction</i> .
<i>stringfunction</i>	Optional. The function to use to format the column headers. This can be any function that takes a single string parameter and returns a string result
SEPARATOR	Specifies the field delimiters.
<i>delimiters</i>	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 <i>delimiters</i> for string values that may contain SEPARATOR or TERMINATOR <i>delimiters</i> 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>')));

//FORMAT option writes the header using the specified formatting function:
IMPORT STD;
OUTPUT(ds, '~thor::outdata.csv', CSV(HEADING(FORMAT(STD.Str.ToUpperCase))));
```

OUTPUT XML Files

[*attr* :=] **OUTPUT**(*recordset*, [*format*] *file*, **XML** [(*xmloptions*)] [, **ENCRYPT**(*key*)] [, **CLUSTER**(*target*)] [, **OVERWRITE**] [, **UPDATE**] [, **EXPIRE**([*days*])])

CLUSTER	Optional. Specifies writing the file to the specified list of target clusters. If omitted, the file is written to the cluster on which the workunit executes. The number of physical file parts written to disk is always determined by the number of nodes in the cluster on which the workunit executes, regardless of the number of nodes on the target cluster(s).
<i>target</i>	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.

<i>key</i>	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.
<i>days</i>	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

<i>rowtag</i>	The text to place in record delimiting tag.
HEADING	Specifies placing header and footer records in the file.
<i>headertext</i>	The text of the header record to place in the file.
<i>footertext</i>	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 JSON Files

[*attr* :=] **OUTPUT**(*recordset*, [*format*] ,*file* ,**JSON** [(*jsonoptions*)] [, **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).
<i>target</i>	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.
<i>key</i>	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.
<i>days</i>	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 JSON data with the name of each field in the specified *format* becoming the JSON tag for that field's data. The valid set of *jsonoptions* are:

'rowtag'

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

TRIM

OPT

<i>rowtag</i>	The text to place in record delimiting tag.
HEADING	Specifies placing header and footer records in the file.
<i>headertext</i>	The text of the header record to place in the file.
<i>footertext</i>	The text of the footer record to place in the file.
TRIM	Specifies removing trailing blanks from string fields before output.
OPT	Specifies omitting tags for any empty string field from the output.

If no *jsonoptions* are specified, the defaults are:

```
JSON( 'Row' ,HEADING( ' [ ' , ' ] ' ) )
```

Example:

```
R := {STRING10 fname,STRING12 lname};
B := DATASET([{'Fred','Bell'},{'George','Blanda'},{'Sam',''}],R);

OUTPUT(B, 'fred1.json', JSON); // writes B to the fred1.json file
/* the Fred1.json file looks like this:
{"Row": [
{"fname": "Fred", "lname": "Bell"},
{"fname": "George", "lname": "Blanda"},
{"fname": "Sam", "lname": ""}
]}
*/
OUTPUT(B, 'fred2.json', JSON('MyResult', HEADING(['', ''])));
/* the Fred2.json file looks like this:
["MyResult": [
{"fname": "Fred", "lname": "Bell"},
{"fname": "George", "lname": "Blanda"},
{"fname": "Sam", "lname": ""}
]]
```

OUTPUT PIPE Files

[*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.
<i>command</i>	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*. This must be a valid label (See Definition Name Rules)

The **EXTEND** option allows multiple **OUTPUT** actions to the same *named* result. The **ALL** option is used to override the implicit **CHOSEN** applied to interactive queries in the Query Builder program. This specifies returning all records.

Example:

```
OUTPUT(CHOSEN(people(firstname[1]='A'),10));
// writes the A People to the query builder
OUTPUT(CHOSEN(people(firstname[1]='A'),10),ALL);
// writes all the A People to the query builder
OUTPUT(CHOSEN(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, CHOOSE

Aggregate Functions

COUNT

COUNT(*recordset*[, *expression*] [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

COUNT(*valuelist*)

<i>recordset</i>	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.
<i>expression</i>	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.
UN-ORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
<i>bool</i>	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
<i>numthreads</i>	Optional. Try to evaluate this activity using <i>numthreads</i> threads.
ALGO-RITHM	Optional. Override the algorithm used for this activity.
<i>name</i>	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
<i>valuelist</i>	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

MAX

MAX(*recordset*, *value* [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

MAX(*valuelist*)

<i>recordset</i>	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.
<i>value</i>	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.
<i>valuelist</i>	A comma-delimited list of expressions to find the maximum value of. This may also be a SET of values.
UN-ORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
<i>bool</i>	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
<i>numthreads</i>	Optional. Try to evaluate this activity using <i>numthreads</i> threads.
ALGO-RITHM	Optional. Override the algorithm used for this activity.
<i>name</i>	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	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**

MIN

MIN(*recordset*, *value* [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

MIN(*valuelist*)

<i>recordset</i>	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.
<i>value</i>	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.
<i>valuelist</i>	A comma-delimited list of expressions to find the minimum value of. This may also be a SET of values.
UN-ORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
<i>bool</i>	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
<i>numthreads</i>	Optional. Try to evaluate this activity using <i>numthreads</i> threads.
ALGORITHM	Optional. Override the algorithm used for this activity.
<i>name</i>	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	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**

SUM

SUM(*recordset*, *value*,[, *expression*] [, **KEYED**])

SUM(*valuelist*[, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	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.
<i>value</i>	The expression to sum.
<i>expression</i>	Optional. A logical expression indicating which records to include in the sum. Valid only when the <i>recordset</i> parameter is the keyword GROUP to indicate summing the elements in a group.
KEYED	Optional. Specifies the activity is part of an index read operation, which allows the optimizer to generate optimal code for the operation.
<i>valuelist</i>	A comma-delimited list of expressions to find the sum of. This may also be a SET of values.
UN-ORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
<i>bool</i>	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
<i>numthreads</i>	Optional. Try to evaluate this activity using <i>numthreads</i> threads.
ALGO-RITHM	Optional. Override the algorithm used for this activity.
<i>name</i>	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	SUM returns a single value.

The **SUM** function returns the additive sum of the *value* in each record of the *recordset* or *valuelist*.

Example:

```
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**

Basic Queries

Exercise 6 - Basic Queries

Exercise Spec:

Explore some very basic ways to query your data. Normally these queries are performed just after a spray and RECORD and DATASET defines, in order to verify that they are correct and the data looks reasonable. The first rule of ECL is to *know your data*!

Steps:

1. Open a new ECL definition file in your repository, naming it **BWR_BasicQueries**.
2. Comment out or delete the starting line in the **BWR_BasicQueries** file that you just created:

```
// export BWR_BasicQueries := 'todo';
```
3. IMPORT all definitions from your Training folder to access other EXPORTed definitions in your active Training folder.
4. Generate an output for the Persons and Accounts tables by simply using the name of the definition in your ECL file. (Hint: This was done in the last two lab exercises).
5. Generate a count of all records in the Persons and Accounts tables.
6. Generate an output for the Persons table, limiting the output to the ID, Last and First Name.
7. Generate an output for the Accounts table, limiting the output to the ReportDate, HighCredit, and Balance fields.
8. Generate an output for the Persons table, limiting the output to the ID, StreetAddress, City, State and ZipCode, and name the output tab in the ECL IDE "Address_Info" (Hint: review the *Named OUTPUT* section in this book).
9. Generate an output for the Accounts table, limiting the output to the AccountNumber, LastActivityDate, and Balance fields, and name the output tab in the ECL IDE "Acct_Activity".

Result Comparison

Verify in the ECL IDE Results tab that all outputs look reasonable.

Verify that the NAMED outputs show correctly in the ECL IDE Results.

The counts you receive should be 841,400 for the Persons table and 8,335,660 for the Accounts table.

This completes this Lab Exercise!

Filtering Your Data

Exercise 7a - Filters (Persons)

Exercise Spec:

Create some simple filters on the *persons* dataset that provides meaningful information and analysis. Except for the file specified below, do not use any new definitions in this exercise; use the existing **Persons** ECL definition file to generate meaningful output in the ECL IDE.

Steps:

1. Create a new definition file and name it **BWR_BasicPersonsFilters**. Use this file to create and generate all of your queries in this exercise.

2. Comment out or delete the starting line in the **BWR_BasicPersonsFilters** file that you just created:

```
// export BWR_BasicPersonsFilters := 'todo';
```

3. **IMPORT** all definitions from your Training Module to eliminate the need to fully qualify your definitions.

4. Filter and Count all persons who live in the state of Florida. Your expected count is 40854.

5. Filter and Count all persons who live in the state of Florida and the city of Miami. Your expected count is now 2438.

6. Filter and Count all persons who live in the state of Florida, the city of Miami, and have a zip code of 33102. Your expected count is now 40.

7. Filter and Count all persons whose First Name begins with the letter 'B'. Use multiple filter conditions in this query. Your expected count is 31193.

8. Filter and Count all persons whose file date year is after 2000. Your expected count is 687.

Best Practices Hint

Refer to the *Recordset Filtering* section found earlier in this book. Also, review the sections that discuss *Set Ordering and Indexing* and *Logical Operators*.

Result Comparison

The results for each step reflected by the expected COUNTs are shown in the steps above.

Exercise 7b - Filters (Accounts)

Exercise Spec:

Create some simple filters on the *accounts* dataset that provides meaningful information and analysis. Except for the file specified below, do not use any new definitions in this exercise; use the existing **Accounts** definition to generate meaningful output in the ECL IDE.

Steps:

1. Create a new definition file and name it **BWR_BasicAccountsFilters**. Use this file to create and generate all of your queries in this exercise (and comment out or delete the export line, just as you did in the previous two exercises).
2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
3. Filter and Count all Accounts whose balance is greater than or equal to 100000. The expected count is 255131.
4. Filter and Count all whose balance is greater than or equal to 100000 and has any Late flag (30, 60 or 90 days). The expected count is 20684.
5. Filter and Count all Accounts who were opened after the year 1999. The expected count is 756676.
6. Filter and Count all Accounts who do not have a term type designated. The expected count is 4288443.

Best Practices Hint

As in Exercise 7a, the use of String Indexing and Logical Operators is the key in this exercise.

Result Comparison

The results for each step, reflected by the expected COUNTs, are shown above.

ECL Definitions

Definition Creation

Similarities and Differences

The similarities come from the fundamental requirement of solving any data processing problem: First, understand the problem.

After that, in many programming environments, you use a "top-down" approach to map out the most direct line of logic to get your input transformed into the desired output. This is where the process diverges in ECL, because the tool itself requires a different way of thinking about forming the solution, and the direct route is not always the fastest. ECL requires a "bottom-up" approach to problem solving.

"Atomic" Programming

In ECL, once you understand what the end result needs to be, you ignore the direct line from problem input to end result and instead start by breaking the issue into as many pieces as possible--the smaller the better. By creating "atomic" bits of ECL code, you've done all the known and easy bits of the problem. This usually gets you 80% of the way to the solution without having to do anything terribly difficult.

Once you've taken all the bits and made them as atomic as possible, you can then start combining them to go the other 20% of the way to produce your final solution. In other words, you start by doing the little bits that you know you can easily do, then use those bits in combination to produce increasingly complex logic that builds the solution organically.

Growing Solutions

The basic Definition building blocks in ECL are the Set, Boolean, Recordset, and Value Definition types. Each of these can be made as "atomic" as needed so that they may be used in combination to produce "molecules" of more complex logic that may then be further combined to produce a complete "organism" that produces the final result.

For example, assume you have a problem that requires you to produce a set of records wherein a particular field in your output must contain one of several specified values (say, 1, 3, 4, or 7). In many programming languages, the pseudo-code to produce that output would look something like this:

```
Start at top of MyFile
Loop through MyFile records
  If MyField = 1 or MyField = 3 or MyField = 4 or MyField = 7
    Include record in output set
  Else
    Throw out record and go back to top of loop
end if and loop
```

While in ECL, the actual code would be:

```
SetValidValues := [1,3,4,7];           //Set Definition
IsValidRec := MyFile.MyField IN SetValidValues; //Boolean
ValRecsMyFile := MyFile(IsValidRec);    //filtered Recordset
OUTPUT(ValRecsMyFile);
```

The thought process behind writing this code is:

"I know I have a set of constant values in the spec, so I can start by creating a Set attribute of the valid values..."

"And now that I have a Set defined, I can use that Set to create a Boolean Attribute to test whether the field I'm interested in contains one of the valid values...

"And now that I have a Boolean defined, I can use that Boolean as the filter condition to produce the Recordset I need to output."

The process starts with creating the Set Attribute "atom," then using it to build the Boolean "molecule," then using the Boolean to grow the "organism"--the final solution.

"Ugly" ECL is Possible, Too

Of course, that particular set of ECL could have been written like this (following a more top down thinking process):

```
OUTPUT(MyFile(MyField IN [1,3,4,7]));
```

The end result, in this case, would be the same.

However, the overall usefulness of this code is drastically reduced because, in the first form, all the "atomic" bits are available for re-use elsewhere when similar problems come along. In this second form, they are not. And in all programming styles, code re-use is considered to be a good thing.

Easy Optimization

Most importantly, by breaking your ECL code into its smallest possible components, you allow ECL's optimizing compiler to do the best job possible of determining how to accomplish your desired result. This leads to another dichotomy between ECL and other programming languages: usually, the less code you write, the more "elegant" the solution; but in ECL, the more code you write, the better and more elegant the solution is generated for you. Remember, your Attributes are just **definitions** telling the compiler what to do, not how to do it. The more you break down the problem into its component pieces, the more leeway you give the optimizer to produce the fastest, most efficient executable code.

Boolean Definitions

Exercise 8a - Boolean Definitions

Exercise Spec:

Create a Boolean Definition that will be TRUE for all male Persons living in Florida who were born after the year 1979. This will be used in subsequent exercises to filter the Persons dataset.

Requirements:

1. Create a new EXPORT Boolean definition file called **IsYoungMaleFloridian**.
2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
3. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsYoungMaleFloridian** text control, neither EXPORT nor SHARED) called **IsFloridian** that will test if a person lives in Florida.
 - Check your Persons record definition to determine the name of the state field.
 - The abbreviation for Florida is 'FL'
4. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsYoungMaleFloridian** text control, neither EXPORT nor SHARED) called **IsMale** that will test if a person's gender is marked as male ('M').
 - Check your Persons record definition to determine the name of the gender field.
5. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsYoungMaleFloridian** text control, neither EXPORT nor SHARED) called **IsBorn80** that will test if a person was born after the year 1979.
 - Check your Persons record definition to determine the name of the birth date field.
 - Use string indexing to check for birth dates after the year 1979. Make sure to also eliminate records with no birth dates.
6. Create the **IsYoungMaleFloridian** Boolean definition so that it results in TRUE for any male living in Florida and born after the year 1979.
 - Use the three local Boolean definitions (**IsFloridian**, **IsMale**, **IsBorn80**) you just created.

Best Practices Hint

Review the sections on *Boolean Attributes* (e.g., definitions) and *Attribute Visibility* discussed earlier in this book.

Result Comparison

At this time you only need to make sure that your ECL syntax is correct. We will use this definition in a subsequent exercise to verify its accuracy.

Exercise 8b - More Boolean Definitions

Exercise Spec:

Create a Boolean definition that will be TRUE if a particular Account is an Invoice type reported before the year 1995 and having any existing balance due.

Requirements:

1. Create an EXPORT Boolean definition called **IsOldInvoice**.
2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
3. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsOldInvoice** text control, neither EXPORT nor SHARED) called **IsInvoice** that will test if an account record is an actual invoice.
 - Use the Accounts *TradeType* field and look for any record marked as 'I' (upper case I).
4. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsOldInvoice** text control, neither EXPORT nor SHARED) called **IsBefore1995** that will test if an account record report was prior to 1995.
 - Make sure that you use the report date of the account record instead of the date that the account was opened.
5. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained in the **IsOldInvoice** text control, neither EXPORT nor SHARED) called **IsActiveBalance** that will test if an account record invoice has any existing balance.
 - An active balance is any account record with a balance greater than zero.
6. Create the **IsOldInvoice** Boolean definition so that it results in TRUE for any account record before 1995 that is marked as an Invoice and has an existing balance due.
 - Use the three local Boolean attributes you just created.

Best Practices Hint

As in Exercise 8a, review the sections on *Boolean Attributes* (e.g., definitions) and *Attribute Visibility* discussed earlier in this book.

Result Comparison

At this time, we only need to make sure that your ECL syntax is correct. We will use this definition in a subsequent exercise to verify its accuracy.

SET Definitions

SET

SET(*recordset*, *field* [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	The set of records from which to derive the SET of values.
<i>field</i>	The field in the recordset from which to obtain the values.
UN-ORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
<i>bool</i>	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
<i>numthreads</i>	Optional. Try to evaluate this activity using <i>numthreads</i> threads.
ALGO-RITHM	Optional. Override the algorithm used for this activity.
<i>name</i>	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	SET returns a SET of values of the same type as the field.

The **SET** function returns a SET for use in any set operation (such as the IN operator), similar to a sub-select in SQL when used with the IN operator. It does not remove duplicate elements and does not order the set.

One common problem is the use of the SET function in a filter condition, like this:

```
MyDS := myDataset(myField IN SET(anotherDataset, someField));
```

The code generated for this is inefficient if "anotherDataset" contains a large number of elements, and may also cause a "Dataset too large to output to workunit" error. A better way to recode the expression would be this:

```
MyDS := JOIN(myDataset, anotherDataset, LEFT.myField = RIGHT.someField, TRANSFORM(LEFT), LOOKUP) ;
```

The end result is the same, the set of "myDataset" records where the "myField" value is one of the "someField" values from "anotherDataset," but the code is much more efficient in execution.

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

Exercise 9 - Creating SET Definitions

Exercise Spec:

A *Set definition* is any whose expression is a set of values, defined within square brackets and separated by commas. Values must all be of the same type (STRING, INTEGER, etc.). Create a static set definition for the Persons dataset and a static set definition for the Accounts dataset. In addition, create a dynamic set definition using the ECL SET function. We will wrap all three definitions inside of a MODULE structure.

Steps:

1. Create a new EXPORT MODULE structure and name it **Sets**.
2. Inside of the **Sets** MODULE, create an EXPORT ECL set definition called **MidwestStates**.
3. Use the following set of strings to create the **MidwestStates** set definition:

```
'ND', 'SD', 'NE', 'KS', 'MN', 'IA', 'MO', 'WI', 'IL', 'IN', 'MI', 'OH'
```

4. Inside of the **Sets** MODULE, create an EXPORT ECL set definition called **AllStates**. Use the SET function to create a dynamic set definition of all states in the Persons dataset.
5. Inside of the **Sets** MODULE, create an EXPORT ECL definition called **AcctTradeTypes**. Use the following set of strings to create the **AcctTradeTypes** definition: I, O, and R

Best Practices Hint

Review the sections on *SET Attributes* (e.g., definitions) and the built-in *SET function* just introduced prior to this exercise. In addition, remember that you are creating three new definitions that will reside inside of your newly created **Sets** MODULE.

Result Comparison

Make sure that the syntax check is correct for all ECL definitions that you created. **DO NOT SUBMIT this MODULE!** We will verify if they are logically correct in subsequent exercises.

RecordSet Definitions

Exercise 10a - Recordset Definition (Persons) - Part 1 of 2

Exercise Spec:

Create a RecordSet definition for the set of male persons living in Florida who were born after 1979.

Steps:

1. Create an EXPORT RecordSet definition called **YoungMaleFloridaPersons**.
2. IMPORT all definitions from your Training folder for easy reference.
3. Use the **IsYoungMaleFloridian** Boolean attribute that you created in *Exercise 8A*.

Result Comparison

Use a new Builder window and verify that the **YoungMaleFloridaPersons** output data looks correct.

Also, perform a COUNT and verify that the result is 462.

Exercise 10b: Recordset Definition (Persons) - Part 2 of 2

Exercise Spec:

Create a RecordSet definition for the set of male persons living in Midwest states.

Steps:

1. Create an EXPORT RecordSet definition called **MenInMidwestStatesPersons**.
2. IMPORT all definitions from your Training folder for easy reference.
3. Use the **MidwestStates** Set definition that you created in Exercise 9.

Best Practices Hint

Review the sections on *Recordset filtering* and the *IN operator* in this book.

Result Comparison

Open a new Builder window and verify that the **MenInMidwestStatesPersons** output data looks correct. Also, perform a COUNT and verify that the result is **92235**.

Exercise 10c: Recordset Definitions (Accounts) - Part 1 of 2

Exercise Spec:

Create a RecordSet definition for the set of all invoice account records opened before the year 1995 and having any existing balance due.

Steps:

1. Create an EXPORT RecordSet definition called **OldActiveInvoiceAccounts**.
2. IMPORT all definitions from your Training folder for easy reference.
3. Use the **IsOldInvoice** Boolean attribute that you created in *Exercise 8B*.

Best Practices Hint

No hints are needed for this simple exercise.

Result Comparison

Verify that the **OldActiveInvoiceAccounts** output data looks correct. Also, perform a COUNT and verify that the result is **5981**.

Exercise 10d: Recordset Definitions (Accounts) - Part 2 of 2

Exercise Spec:

Create a RecordSet definition for the set of all account records that are not in a valid set of trade type codes.

Steps:

1. Create an EXPORT RecordSet definition called **NoTradeTypeAccounts**.
2. IMPORT all definitions from your Training folder for easy reference.
3. Use the **AcctTradeTypes** Set definition that you created in the MODULE of Exercise 9.

Best Practices Hint

The key to this exercise is not what exists in the SET definition, but what does *not*.

Result Comparison

Verify that the **NoTradeTypeAccounts** output data looks correct. Also, perform a COUNT and verify that the result is **280823**.

Conditional Functions

IF

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

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

MAP

MAP(*expression*=>*value*, [*expression*=>*value*, ...] [,*elsevalue*])

<i>expression</i>	A conditional expression.
=>	The "results in" operator--valid only in MAP, CASE, and CHOOSESETS.
<i>value</i>	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.
<i>elsevalue</i>	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 <i>value</i> .

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 are 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

CASE

CASE(*expression*, *caseval* => *value*, [... , *caseval* => *value*][, *elsevalue*])

<i>expression</i>	An expression that results in a single value.
<i>caseval</i>	A value to compare against the result of the expression.
=>	The "results in" operator--valid only in CASE, MAP and CHOOSESETS.
<i>value</i>	The value to return. This may be any expression or action.
<i>elsevalue</i>	Optional. The value to return when the result of the expression does not match any of the <i>caseval</i> 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

CHOOSE

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

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

REJECTED

REJECTED(*condition*,...,*condition*)

<i>condition</i>	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**

WHICH

WHICH(*condition*,...,*condition*)

<i>condition</i>	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

Mathematical Functions

ABS

ABS(*expression*)

<i>expression</i>	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
```

ROUND

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

<i>realvalue</i>	The floating-point value to round.
<i>decimals</i>	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

The FUNCTION Structure

FUNCTION Structure

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

code

RETURN *retval*;

END;

<i>resulttype</i>	The return value type of the function. If omitted, the type is implicit from the <i>retval</i> expression.
<i>funcname</i>	The ECL attribute name of the function.
<i>parameterlist</i>	A comma separated list of the parameters to pass to the <i>function</i> . These are available to all attributes defined in the FUNCTION's <i>code</i> .
<i>code</i>	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 <i>retval</i> .
<i>retval</i>	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,0);
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

Exercise 11: Function Definition without FUNCTION

Exercise Spec:

Create a Value function to return a value limited to a maximum amount. This will be used in subsequent exercises to limit a result value to a specified maximum.

Requirements:

1. Create an EXPORT value function called **Limit_Value**(*n*, *maxval*)
 - Use the IF function to determine if the passed *n* value is greater than the passed *maxval*.
 - If true, return the *maxval*, otherwise return *n*.

Best Practices Hint

It's important to remember that any ECL definition that receives parameters is a function. The FUNCTION structure is simply a language organization tool that allows you to organize multiple definitions that work together to comprise a single logical function return value more efficiently. You do not need to use a FUNCTION structure in this exercise.

Result Comparison

Make sure your syntax is correct, and you can test this function using static values.

Exercise 12: Function Definition using FUNCTION

Exercise Spec:

Create a STRING function that returns detail information regarding an integer value passed to it within an inclusive range of high and low values. **Write this function using a FUNCTION structure.**

Steps:

1. Create an EXPORT String FUNCTION structure named **ValidInRange**(*PassedVal*, *LoVal*, *HiVal*)
2. Inside the FUNCTION, create a local Boolean definition named **IsNegative** to determine If the *LoVal* or *HiVal* is a negative value.
3. Also inside the FUNCTION, create a second local Boolean definition named **IsBackwards** to determine if the *HiVal* is less than or equal to the *LoVal*.
4. Create a third local Boolean definition inside the FUNCTION named **IsInRange** that validates that the *PassedVal* is between the inclusive range of *LoVal* and *HiVal*. .
5. If the *PassedVal* is within the inclusive range of the converted values, return "In Range".
6. If the *PassedVal* is NOT within the inclusive range of the converted values, return "Out of Range".
7. If the *LoVal* value is greater than or equal to the *HiVal* value, return "Invalid Inputs - Parameters are reversed".
8. If either the *LoVal* or the *HiVal* is negative, return "Invalid Inputs - Negative value"
9. The FUNCTION will RETURN the result of a conditional **MAP** function to test the conditions above and return the correct values.

Best Practices Hint

It's important to remember that any ECL definition that receives parameters is a function. The FUNCTION structure is simply a language organization tool that allows you to organize multiple definitions that work together to comprise a single logical function return value more efficiently.

Result Comparison

Make sure your syntax is correct, and you can test this function in a Builder window using a variety of static values, including negative values.

Recordset Functions

EXISTS

EXISTS(*recordset* [, **KEYED**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL**] [, (*numthreads*)]] [, **ALGORITHM**(*name*)])

EXISTS(*valuelist*)

<i>recordset</i>	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.
<i>valuelist</i>	A comma-delimited list of expressions. This may also be a SET of values.
UN-ORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
<i>bool</i>	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
<i>numthreads</i>	Optional. Try to evaluate this activity using <i>numthreads</i> threads.
ALGO-RITHM	Optional. Override the algorithm used for this activity.
<i>name</i>	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	EXISTS returns a single BOOLEAN value.

The **EXISTS** function returns true if the number of records in the specified *recordset* is > 0, or the *valuelist* is populated. This is most commonly used to detect whether a filter has filtered out all the records.

When checking for an empty recordset, use the **EXISTS**(*recordset*) function instead of the expression: **COUNT**(*recordset*) > 0. Using EXISTS results in more efficient processing and better performance under those circumstances.

Example:

```
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

SORT

SORT(*recordset*,*value*[[**JOINED**(*joinedset*)][**SKEW**(*limit*[[*target*]]),**THRESHOLD**(*size*)][**LOCAL**] [**FEW**][**STABLE**] [*algorithm*)] | **UNSTABLE** [*algorithm*)]][, **UNORDERED** | **ORDERED**(*bool*)] [, **PARALLEL** [*numthreads*]]][, **ALGORITHM**(*name*)])

<i>recordset</i>	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.
<i>value</i>	A comma-delimited list of expressions or key fields in the recordset on which to sort, with the left-most 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 <i>joinedset</i> so that matching records between the recordset and <i>joinedset</i> end up on the same supercomputer nodes. Used to optimize supercomputer joins where the <i>joinedset</i> is very large and the recordset is small.
<i>joinedset</i>	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.
<i>limit</i>	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%).
<i>target</i>	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.
<i>size</i>	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.
<i>algorithm</i>	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.
UNORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
<i>bool</i>	When False, specifies the output record order is not significant. When True, specifies the default output record order.
PARALLEL	Optional. Try to evaluate this activity in parallel.
<i>numthreads</i>	Optional. Try to evaluate this activity using <i>numthreads</i> threads.

ALGORITHM	Optional. Override the algorithm used for this activity.
<i>name</i>	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	SORT returns a set of records.

The **SORT** function orders the *recordset* according to the *values* specified, and (if LOCAL is not specified) partitions the result such that all records with the same *values* are on the same node. SORT is usually used to produce the record sets operated on by the DEDUP, GROUP, and ROLLUP functions, so that those functions may operate optimally. Sorting final output is, of course, another common use.

Sorting Algorithms

There are three sort algorithms available: quicksort, 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 CHOSEN, 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

Functional SORT Example

SORT

Open BWR_Training_Examples.SORT_Example in an ECL window and Submit this query:

```
MyRec := RECORD
  STRING1 Value1;
  STRING1 Value2;
END;

SomeFile := DATASET([{'C','G'},
                    {'C','C'},
                    {'A','X'},
                    {'B','G'},
                    {'A','B'}],MyRec);

SortedRecs1 := SORT(SomeFile,Value1,Value2);
SortedRecs2 := SORT(SomeFile,-Value1,Value2);
SortedRecs3 := SORT(SomeFile,Value1,-Value2);
SortedRecs4 := SORT(SomeFile,-Value1,-Value2);

OUTPUT(SortedRecs1);
OUTPUT(SortedRecs2);
OUTPUT(SortedRecs3);
OUTPUT(SortedRecs4);

/*
SortedRecs1 results in:
  Rec#    Value1    Value2
  1       A         B
  2       A         X
  3       B         G
  4       C         C
  5       C         G

SortedRecs2 results in:
  Rec#    Value1    Value2
  1       C         C
  2       C         G
  3       B         G
  4       A         B
  5       A         X

SortedRecs3 results in:
  Rec#    Value1    Value2
  1       A         X
  2       A         B
  3       B         G
  4       C         G
  5       C         C

SortedRecs4 results in:
  Rec#    Value1    Value2
  1       C         G
  2       C         C
  3       B         G
  4       A         X
  5       A         B
*/
```

DEDUP

DEDUP(*recordset* [, *condition* [[**MANY**], **ALL**[, **HASH**]] [, **BEST** (*sort-list*)] [, **KEEP** *n*] [, *keeper*]] [, **LOCAL**] [, **UNORDERED** | **ORDERED**(*bool*)] [, **STABLE** | **UNSTABLE**] [, **PARALLEL** [(*numthreads*)]] [, **ALGORITHM**(*name*)])

<i>recordset</i>	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.
<i>condition</i>	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.
MANY	Optional. Specifies or perform a local sort/dedup before finding duplicates globally. This is most useful when many duplicates are expected.
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.
BEST	Optional. Provides additional control over which records are retained from a set of "duplicate" records. The first in the <sort-list> order of records are retained. BEST cannot be used with a KEEP parameter greater than 1.
<i>sort-list</i>	A comma delimited list of fields defining the duplicate records to keep.. The fields may be prefixed with a minus sign to require a reverse sort on that field.
KEEP	Optional. Specifies keeping <i>n</i> number of duplicate records. If omitted, the default behavior is to KEEP 1. Not valid with the ALL option present.
<i>n</i>	The number of duplicate records to keep.
<i>keeper</i>	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.
UN-ORDERED	Optional. Specifies the output record order is not significant.
ORDERED	Specifies the significance of the output record order.
<i>bool</i>	When False, specifies the output record order is not significant. When True, specifies the default output record order.
STABLE	Optional. Specifies the input record order is significant.
UNSTABLE	Optional. Specifies the input record order is not significant.
PARALLEL	Optional. Try to evaluate this activity in parallel.
<i>numthreads</i>	Optional. Try to evaluate this activity using <i>numthreads</i> threads.
ALGO-RITHM	Optional. Override the algorithm used for this activity.
<i>name</i>	The algorithm to use for this activity. Must be from the list of supported algorithms for the SORT function's STABLE and UNSTABLE options.
Return:	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.

The **BEST** option provides additional control over which records are retained from a set of "duplicate" records. The first in the *sort-list* order of records are retained. The *sort-list* is comma delimited list of fields. The fields may be prefixed with a minus sign to require a reverse sort on that field.

`DEDUP(recordset, field1, BEST(field2))` means that from set of duplicate records, the first record from the set duplicates sorted by field2 is retained. `DEDUP(recordset, field1, BEST(-field2))` produces the last record sorted by field2 from the set of duplicates.

The **BEST** option cannot be used with a **KEEP** parameter greater than 1.

Example:

```
SomeFile := DATASET([{'001','KC','G'},
                    {'002','KC','Z'},
                    {'003','KC','Z'},
                    {'004','KC','C'},
                    {'005','KA','X'},
                    {'006','KB','A'},
                    {'007','KB','G'},
                    {'008','KA','B'}],{STRING3 Id, String2 Value1, String1 Value2});

SomeFile1 := SORT(SomeFile, Value1);

DEDUP(SomeFile1, Value1, BEST(Value2));
// Output:
// id value1 value2
// 008 KA B
// 006 KB A
// 004 KC C

DEDUP(SomeFile1, Value1, BEST(-Value2));
// Output:
// id value1 value2
// 005 KA X
// 007 KB G
// 002 KC Z

DEDUP(SomeFile1, Value1, HASH, BEST(Value2));
// Output:
```

```
// id value1 value2
// 008 KA B
// 006 KB A
// 004 KC C
```

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>NamesTbl2,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

Functional DEDUP Example

Simple DEDUP

Open BWR_Training_Examples.DEDUP_Example in an ECL window and Submit this query:

```
MyRec := RECORD
  STRING1 Value1;
  STRING1 Value2;
END;

SomeFile := DATASET([{'C','G'},
                     {'C','C'},
                     {'A','X'},
                     {'B','G'},
                     {'A','B'}],MyRec);

Val1Sort := SORT(SomeFile,Value1);
Val2Sort := SORT(SomeFile,Value2);

Dedup1    := DEDUP(Val1Sort,LEFT.Value1 = RIGHT.Value1);

/* Result set is:
  Rec#    Value1    Value2
  1        A         X
  2        B         G
  3        C         G
*/

Dedup2 := DEDUP(Val2Sort,LEFT.Value2 = RIGHT.Value2);

/* Result set is:
  Rec#    Value1    Value2
  1        A         B
  2        C         C
  3        C         G
  4        A         X
*/

Dedup3 := DEDUP(Val1Sort,LEFT.Value1 = RIGHT.Value1,RIGHT);

/* Result set is:
  Rec#    Value1    Value2
  1        A         B
  2        B         G
  3        C         C
*/

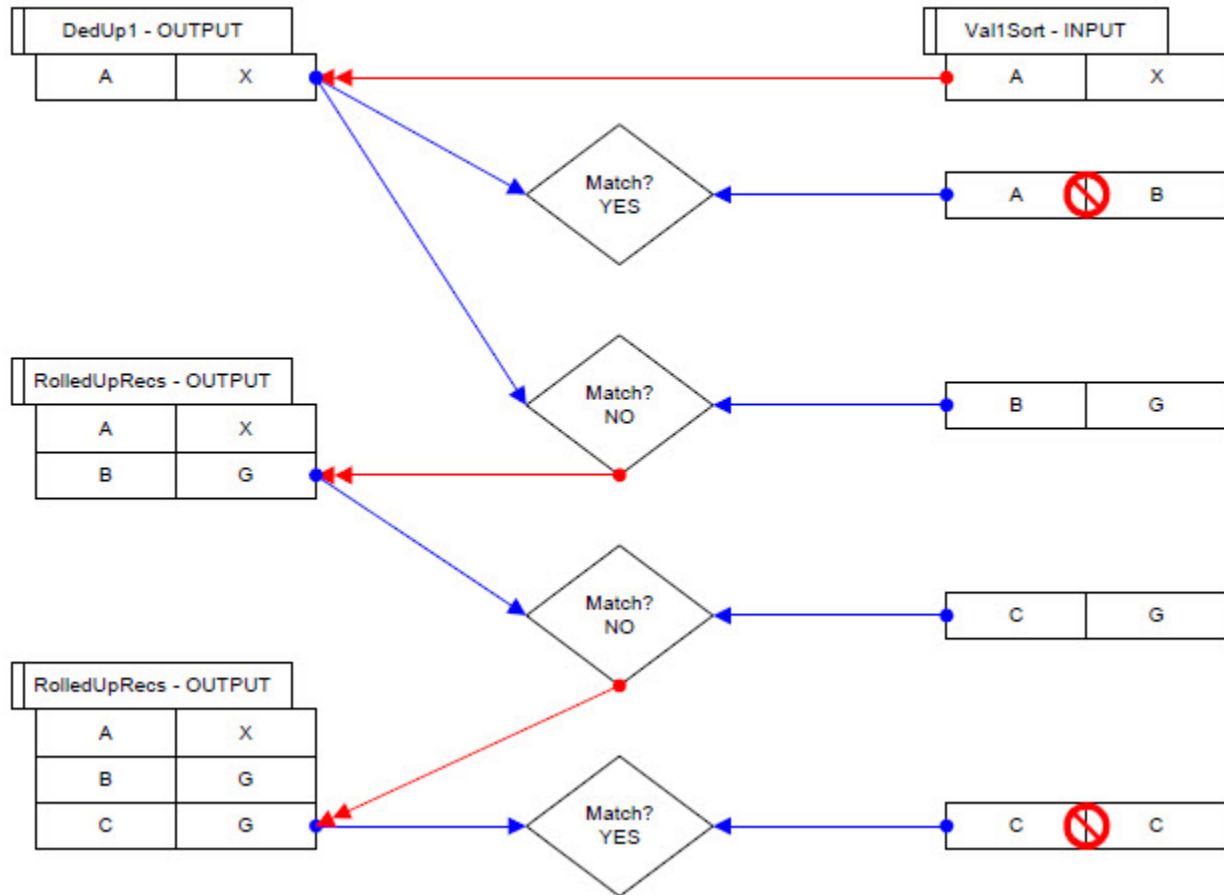
Dedup4 := DEDUP(Val2Sort,LEFT.Value2 = RIGHT.Value2,RIGHT);

/* Result set is:
  Rec#    Value1    Value2
  1        A         B
  2        C         C
  3        B         G
  4        A         X
*/

output(Dedup1);
output(Dedup2);
output(Dedup3);
output(Dedup4);
```

DEDUP Functional Example Diagram

Simple DEDUP Functional Example Diagram



Value Definitions

Exercises 13a - 13c: Using Value Definitions

The following exercises are used to showcase some powerful ECL language statements introduced in the last section and used with Value definitions.

Exercise 13a

Exercise Spec:

Calculate the total number of Invoice Accounts that have a zero (0) balance.

Steps:

1. Create a new EXPORT Value Definition named **Val001**.
2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
 - Use the COUNT function to implement the definition.
 - Use the *Balance* and *TradeType* fields from the *Accounts* dataset.

Best Practices Hint

Create local Boolean definitions and then incorporate them into your filter expression.

Result Comparison

Submit your ECL file to the THOR target and verify that the expected count is **1559626**.

Exercise 13b

Exercise Spec:

Calculate the ratio of High Credit values to the Balance owed for all accounts, rounding to the nearest integer.

Steps:

1. Create a new EXPORT Value Definition called **Val002**.
2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
 - Use the ROUND function to implement the definition.
 - Ratio is the cumulative sum of the *HighCredit* field divided by the cumulative sum of the *Balance* field in the *Accounts* dataset.
3. Use the SUM function to total the *HighCredit* and *Balance* fields as needed.

Best Practices Hint

Create two (2) local attributes to store the intermediate sums that will be used in the ratio.

Result Comparison

Submit your ECL code to the THOR target and verify that the expected ratio integer result is **3**.

Exercise 13c

Exercise Spec:

Given the dynamic set of states that you created in Exercise 9, use an inline DATASET with SORT and DEDUP to determine the unique COUNT of states in the Persons dataset. Refer to the DATASET *Inline Dataset* section therein, and also review the SORT and DEDUP sections in this book.

Steps:

1. Create a new EXPORT Value Definition called **Val003**.
2. IMPORT all definitions from your Training folder for easy reference.
3. Use the Sets.AllStates dynamic set that you created in Exercise 9 as the input to your inline dataset.
4. SORT and DEDUP the inline dataset.
5. The output of **Val003** will be the COUNT of the deduped inline dataset.

Result Comparison

Submit the **Val003** definition in a new Builder window, and verify that the result is **59**.

More on DEDUP

DEDUP ALL

DEDUP with the ALL Option

Open BWR_Training_Examples.DEDUP_ALL_Example in an ECL window and Submit this query:

```
MyRec := RECORD
  STRING1 Value1;
  STRING1 Value2;
END;

SomeFile := GROUP(DATASET([{'C','G'},
                           {'C','C'},
                           {'A','X'},
                           {'B','G'},
                           {'A','B'}],MyRec),TRUE);

Dedup1 := DEDUP(SomeFile,
  LEFT.Value2 IN ['G','C','X'] AND
  RIGHT.Value2 IN ['X','B','C'],ALL);

/*
Processes as: LEFT    vs.   RIGHT
              1 (G)      2 (C) - lose 2 (RIGHT rec)
              1 (G)      3 (X) - lose 3 (RIGHT rec)
              1 (G)      4 (G) - keep RIGHT rec 4
              1 (G)      5 (B) - lose 5 (RIGHT rec)

              4 (G)      1 (G) - keep RIGHT rec 1

Result set is:
  Rec#   Value1   Value2
  1      C        G
  4      B        G
*/

Dedup2 := DEDUP(SomeFile,
  LEFT.Value2 IN ['G','C'] AND
  RIGHT.Value2 IN ['X','B'],ALL);
/* Result set is:
  Rec#   Value1   Value2
  1      C        G
  2      C        C
  4      B        G */

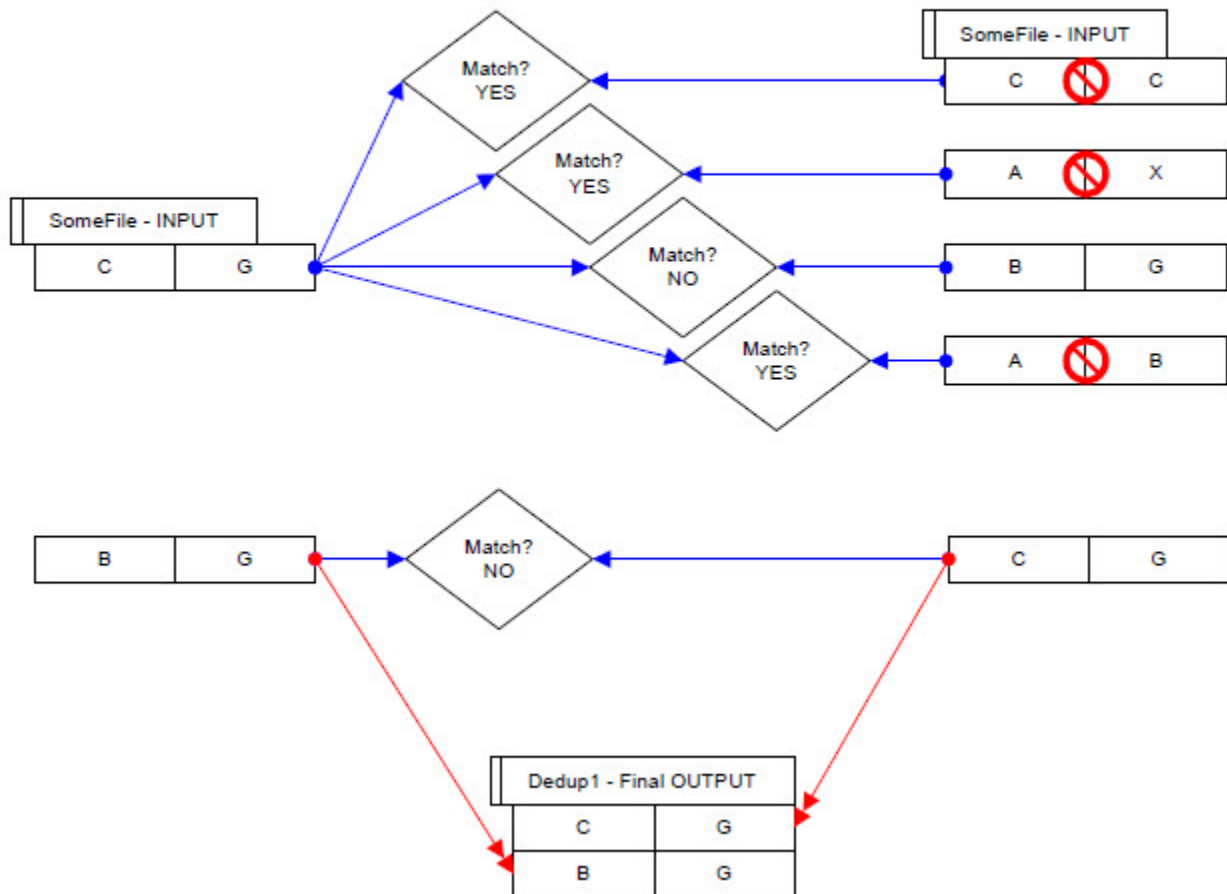
Dedup3 := DEDUP(SomeFile,
  LEFT.Value2 IN ['X','B'] AND
  RIGHT.Value2 IN ['G','C'],ALL);

/* Result set is:
  Rec#   Value1   Value2
  3      A        X
  5      A        B */

output(Dedup1);
output(Dedup2);
output(Dedup3);
```

DEDUP ALL Functional Example Diagram

DEDUP ALL Functional Example Diagram
LEFT.Value2 IN ['G','C','X'] AND RIGHT.Value2 IN ['X','B','C']



Lab Exercise Solutions

NOTE: There are no solutions for Exercises 1 - 3, and 16. Those exercises involved simple spray/despray operations and an introduction to the ECL IDE.

Exercise 4 - File_Persons.ECL

Solution:

```
EXPORT File_Persons := MODULE
  EXPORT Layout := RECORD
    UNSIGNED8 ID;
    STRING15  FirstName;
    STRING25  LastName;
    STRING15  MiddleName;
    STRING2   NameSuffix;
    STRING8   FileDate;
    UNSIGNED2 BureauCode;
    STRING1   MaritalStatus;
    STRING1   Gender;
    UNSIGNED1 DependentCount;
    STRING8   BirthDate;
    STRING42  StreetAddress;
    STRING20  City;
    STRING2   State;
    STRING5   ZipCode;
  END;

  //YOUR initials between CLASS and Intro:
  EXPORT File := DATASET('~CLASS::XX::Intro::Persons', Layout, FLAT);
END;
```

Exercise 5 - File_Accounts.ECL

Solution:

```
EXPORT File_Accounts := MODULE
  EXPORT Layout := RECORD
    UNSIGNED8 PersonID;
    STRING8 ReportDate;
    STRING2 IndustryCode;
    UNSIGNED4 Member;
    STRING8 OpenDate;
    STRING1 TradeType;
    STRING1 TradeRate;
    UNSIGNED1 Narr1;
    UNSIGNED1 Narr2;
    UNSIGNED4 HighCredit;
    UNSIGNED4 Balance;
    UNSIGNED2 Terms;
    UNSIGNED1 TermTypeR;
    STRING20 AccountNumber;
    STRING8 LastActivityDate;
    UNSIGNED1 Late30Day;
    UNSIGNED1 Late60Day;
    UNSIGNED1 Late90Day;
    STRING1 TermType;
  END;

  //YOUR initials between CLASS:: and ::INTRO
  EXPORT File := DATASET('~CLASS::XX::Intro::Accounts',Layout,CSV);
END;
```

Exercise 6 - BWR_BasicQueries.ECL

Solution:

```
//export BasicQueries := 'todo'; delete this line
IMPORT $;
Persons := $.File_Persons.File;
Accounts := $.File_Accounts.File;

Persons;
Accounts;

COUNT(Persons);
COUNT(Accounts);

OUTPUT(Persons, {ID, LastName, FirstName});
OUTPUT(Accounts, {ReportDate, HighCredit, Balance});

OUTPUT(Persons, {ID, StreetAddress, City, State, ZipCode}, NAMED('Address_Info'));
OUTPUT(Accounts, {AccountNumber, LastActivityDate, Balance}, NAMED('Acct_Activity'));
```

Exercise 7a:

BWR_BasicPersonsFilters.ECL

Solution:

```
IMPORT $;
Persons := $.File_Persons.File;

Persons(State = 'FL');
COUNT(Persons(State = 'FL')); //40854

Persons(State = 'FL',City = 'MIAMI');
COUNT(Persons(State = 'FL',City = 'MIAMI')); //2438

Persons(State = 'FL',City = 'MIAMI',ZipCode='33102');
COUNT(Persons(State = 'FL',City = 'MIAMI',ZipCode='33102')); //40

Persons(FirstName >= 'B', FirstName < 'C');
COUNT(Persons(FirstName >= 'B', FirstName < 'C')); //31193

Persons(FileDate[..4] > '2000');
COUNT(Persons(FileDate[..4] > '2000')); //687
```

Exercise 7b:

BWR_BasicAccountsFilters.ECL

Solution:

```
IMPORT $;
Accounts := $.File_Accounts.File;

Accounts(Balance >= 100000);
COUNT(Accounts(Balance >= 100000)); //255131

Accounts(Balance >= 100000, Late30Day >0 OR Late60Day >0 OR Late90Day >0);
COUNT(Accounts(Balance >= 100000, (Late30Day>0 OR Late60Day>0 OR Late90Day>0)));

Accounts(OpenDate[..4] >= '2000');
COUNT(Accounts(OpenDate[..4] >= '2000'));

Accounts(TermType = '');
COUNT(Accounts(TermType = ''));
```

Exercise 8a:

IsYoungMaleFloridian.ECL

Solution:

```
IMPORT $;
Persons      := $.File_Persons.File;

IsFloridian  := Persons.State = 'FL';

IsMale       := Persons.Gender = 'M';

IsBorn80     := Persons.Birthdate <> '' AND Persons.Birthdate[..4] >= '1980' ;

EXPORT IsYoungMaleFloridian := IsFloridian AND
                                IsMale AND
                                IsBorn80;
```

Exercise 8b: IsOldInvoice.ECL

Solution:

```
IMPORT $;
Accounts      := $.File_Accounts.File;
IsInvoice     := Accounts.TradeType = 'I';
IsBefore1995  := Accounts.ReportDate <> '' AND Accounts.ReportDate[..4] < '1995';
IsActiveBalance := Accounts.Balance > 0;

EXPORT IsOldInvoice := IsInvoice AND
                       IsBefore1995 AND
                       IsActiveBalance;
```

Exercise 9: Set Definitions

Solution: Sets.ECL

```
IMPORT $;
Persons := $.File_Persons.File;

EXPORT Sets := MODULE
  EXPORT MidwestStates := ['ND','SD','NE','KS','MN','IA','MO','WI','IL','IN','MI','OH'];
  EXPORT AcctTradeTypes := ['O','I','R'];
  EXPORT AllStates := SET(Persons,State);
END;
```


Exercise 10: Recordset Definitions

Solution 10a: YoungMaleFloridaPersons.ECL:

```
IMPORT $;  
Persons := $.File_Persons.File;  
  
EXPORT YoungMaleFloridaPersons := Persons($.IsYoungMaleFloridian);
```

Solution 10b: MenInMidwestStatesPersons.ECL:

```
IMPORT $;  
Persons := $.File_Persons.File;  
EXPORT MenInMidwestStatesPersons := Persons(State IN $.Sets.MidwestStates, Gender = 'M');
```

Solution 10c: OldActiveInvoiceAccounts.ECL:

```
IMPORT $;  
Accounts := $.File_Accounts.File;  
EXPORT OldActiveInvoiceAccounts := Accounts($.IsOldInvoice);
```

Solution 10d: NoTradeTypeAccounts.ECL:

```
IMPORT $;  
Accounts := $.File_Accounts.File;  
EXPORT NoTradeTypeAccounts := Accounts(TradeType NOT IN $.Sets.AcctTradeTypes);
```

Exercise 11: Function Definitions

Solution 11: Limit_Value.ECL

```
EXPORT Limit_Value(n,maxval) := IF(n > maxval, maxval, n);
```

Exercise 12: Using a FUNCTION Structure

Solution: ValidInRange.ECL

```
EXPORT
ValidInRange(PassedVal, LoVal, HiVal) := FUNCTION
  IsNegative := LoVal < 0 OR HiVal < 0;
  IsBackwards := HiVal < LoVal;
  IsInRange := PassedVal BETWEEN LoVal AND HiVal;
  RETURN MAP(IsNegative => 'Invalid Input - Negative Value',
             IsBackwards => 'Invalid Input, parameters are reversed',
             IsInRange => 'In Range',
             'Out of range');
END;
```

Exercise 13: Value Definitions

Solution 13a: Val001.ECL

```
IMPORT $;  
Accounts      := $.File_Accounts.File;  
IsInvoice     := Accounts.TradeType = 'I';  
IsZeroBalance := Accounts.Balance = 0;  
  
EXPORT Val001 := COUNT(Accounts(IsInvoice AND IsZeroBalance));
```

Exercise 13: Value Definitions

Solution 13b: Val002.ECL

```
IMPORT $;  
Accounts      := $.File_Accounts.File;  
  
SumHighCredit := SUM(Accounts,HighCredit);  
SumBalance    := SUM(Accounts,Balance);  
  
EXPORT Val002 := ROUND(SumHighCredit/SumBalance);
```

Exercise 13: Value Definitions

Solution 13c: Val003.ECL

```
IMPORT $;  
  
SetDS      := DATASET($.Sets.AllStates,{STRING2 State});  
SortedSet  := SORT(SetDS,State);  
DedupedSet := DEDUP(SortedSet,State);  
EXPORT Val003 := COUNT(DedupedSet);
```