U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

NISTIR 4814

National PDES Testbed Report Series

NIST Express Working Form Programmer's Reference

Revised April, 1992

NATIONAL PDES TESTBED™

Stephen Nowland Clark
Don Libes

April 3, 1992

NIST

NISTIR 4814

National PDES Testbed
Report Series

Sponsored by:

U.S. Department of Defense

CALS Evaluation and
Integration Office

The Pentagon

Washington, DC 20301-8000

NATIONAL PDES TESTBED™

NIST Express Working Form Programmer's Reference

Revised April, 1992

Stephen Nowland Clark
Don Libes

U.S. Department of Commerce

Barbara H. Franklin, Secretary

Technology Administration

Robert M. White,
Undersecretary for Technology

National Institute of Standards and Technology

John W. Lyons, Director

April 3, 1992

DEPARTMENT OF COMMERCE
UNITED STATES OF AMERICA

Table Of Contents

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied.

Unix is a trademark of AT&T Technologies, Inc.

Smalltalk-80 is a trademark of ParcPlace Systems, Inc.

NIST Express Working Form Programmer's Reference

Stephen Nowland Clark
Don Libes[1]

1 Introduction

The NIST Express Working Form [Clark90b], with its associated Express parser, Fed-X, is a Public Domain set of software tools for manipulating information models written in the Express language [Part11]. The Express Working Form (WF) is part of the NIST PDES Toolkit [Clark90a]. This reference manual discusses the internals of the Working Form, including the Fed-X parser. The information presented will be of use to programmers who wish to write applications based on the Working Form, including output modules for Fed-X, as well as those who will maintain or modify the Working form or Fed-X. The reader is assumed to be familiar with the design of the Working Form, as presented in [Clark90b].

1.1 Context

The PDES (Product Data Exchange using STEP) activity is the United States' effort in support of the Standard for the Exchange of Product Model Data (STEP), an emerging international standard for the interchange of product data between various vendors' CAD/CAM systems and other manufacturing-related software [Mason91]. A National PDES Testbed has been established at the National Institute of Standards and Technology to provide testing and validation facilities for the emerging standard. The Testbed is funded by the Computer-aided Acquisition and Logistic Support (CALS) program of the Office of the Secretary of Defense. As part of the testing effort, NIST is charged with providing a software toolkit for manipulating STEP data. This NIST PDES Toolkit is an evolving, research-oriented set of software tools. This document is one of a set of reports which describe various aspects of the Toolkit. An overview of the Toolkit is provided in [Clark90a], along with references to the other documents in the set.

1. Don Libes is responsible for the minor changes made to this document to track the actual implementation of the software described. However, credit for the bulk of the document, its style, and the implementation of the NIST Express Working Form remains with Stephen Nowland Clark. Recent changes are denoted by a change bar to the left of the text.

2 Fed-X Control Flow

A Fed-X translator consists of three separate passes: parsing, reference resolution, and output generation. The first two passes can be thought of as a single unit which produces an instantiated Working Form (WF). This Working Form can be traversed by an output module in the third pass. It is anticipated that users will need output formats other than those provided with the NIST Toolkit. The process of writing a report generator for a new output format is discussed in detail in section 4.

2.1 First Pass: Parsing

The first pass of Fed-X is a fairly straightforward parser, written using the Unix™ parser generation languages, Yacc and Lex. As each construct is parsed, it is added to the Working Form. No attempt is made to resolve symbol references: they are represented by instances of the type `Symbol` (see below), which are replaced in the second pass with the referenced objects.

The grammar used by Fed-X is processed by Yacc or Bison (a Yacc clone available from the Free Software Foundation[1]). The lexical analyzer is processed by Lex or Flex[2], a fast, public domain implementation of Lex. Generally, Flex and Bison are faster and provide more features. For portability, some of these features are avoided by Fed-X even though such use might make the result simpler and faster (such as the multiple start condition machinary offered by Flex). When easily handled (such as by conditional compilation (`#ifdef .. #endif` pairs)), certain features of Flex and Bison are taken advantage of. In general, Flex and Bison are prefered over Lex and Yacc. The choice is controlled by the Makefile (and make_rules) that directs the building of the system.

2.2 Second Pass: Reference Resolution

The reference resolution pass of Fed-X walks through the Working Form built by the parser and attempts to replace each `Symbol` with the object to which it refers. The name of each symbol is looked up in the scope which is in effect at the point of reference. If a definition for the name is found which makes sense in the current context, the definition replaces the symbol reference. Otherwise, Fed-X prints an error message and proceeds.

In some cases, the changes which must be made when a symbol is resolved are slightly more drastic. For example, the syntax of Express does not distinguish between an identifier and an invocation of a function of no arguments. When a token could be inter-

1. The Free Software Foundation (FSF) of Cambridge, Massachusetts is responsible for the GNU Project, whose ultimate goal is to provide a free implementation of the UNIX operating system and environment. These tools are not in the public domain: FSF retains ownership and copyright priviledges, but grants free distribution rights under certain terms. At this writing, further information is available via electronic mail on the Internet from gnu@prep.ai.mit.edu.
2. Vern Paxson's Flex is usually distributed with GNU software, although, being in the public domain, it does not come under the FSF licensing restrictions.

preted as either, the parser always assumes that it is a simple identifier. When the second pass determines that one of these objects actually refers to a function, the `Identifier` expression is replaced by an appropriate `Function_Call` expression.

Thus, the result of the second pass (in the absence of any errors) is a tightly linked set of structures in which, for example, `Function_Call` expressions reference the called `Algorithms` directly. At this point, it is possible to traverse the data structures without resorting to any further symbol table lookups. The scopes in the Working Form are only needed to resolve external references - e.g., from a STEP physical file.

2.3 Third Pass: Output Generation

The report or output generation pass manages the production of the various output files. Control is essentially handed over to the application-programmer-supplied output module loaded at build time.

In theory, the module could do anything, but more typically, the output module translates the Working Form into some other form such as a human-readable report, or input to an SQL database.

A report generator is an object module, most likely written in C, which has been compiled as a component module for a larger program (i.e., with the `-c` option to a UNIX C compiler). The code of this module consists of calls to Express Working Form access functions and to standard output routines. A detailed description of the creation of a new output module appears in section 4.

3 Working Form Implementation

The Express Working Form data abstractions are implemented in Standard C [ANSI89]. Standard C is not essential to Fed-X, and some effort has been taken to make the source Classic C compatible but this work is not complete. Application modules (i.e., output modules) can be written in either Standard C or Classic C.

Each abstraction is implemented as one or more classes, using the `Class`/`Object` modules in `libmisc` [Clark90c]. The data specific to a particular class is encapsulated in a private C `struct`. This structure is never manipulated directly outside of the abstraction's module. For example:

```
/* the actual contents of a Foo */
struct Foo {
     int i;
     double d;
};

typedef Object Foo;

/* Class_Foo is created in FOOinitialize() */
```

```
Class Class_Foo;
```

Outside of `Foo`'s module, we will never see a `struct Foo`. We will only see a `Foo`, which is actually an `Object` which ultimately points at a `struct Foo`.

3.1 Primitive Types

The Express Working Form makes use of several modules from the Toolkit general libraries, including the Class, Object, Error, Linked_List, and Dictionary modules. These are described in [Clark90c]. The underlying representation for all of the Working Form abstractions makes use of the Class and Object modules.

3.2 Symbol and Construct

All Working Form objects are subclassed from the types `Symbol` and `Construct`. After the working form has been built, these types become, in Object-Oriented terminology, abstract supertypes[1] for the various types in the Working Form. The two are quite similar, both in concept and in implementation. Both have an attribute containing the line number on which the represented construct appears in the source file (probably useful only within Fed-X). A `Symbol` also includes a name and a flag indicating whether the symbol has been resolved.

Abstractions which represent nameable objects are subclassed from `Symbol`. These include `Constant`, `Type`, `Variable`, `Algorithm`, `Entity`, and `Schema`. The latter three are actually subclasses of another `Symbol` subclass, `Scope`. Other abstractions (`Case_Item`, `Expression`, `Loop_Control`, and `Statement`) are subclassed from `Construct`.

3.3 Express Working Form Manager Module

In addition to the abstractions discussed in [Clark90b], `libexpress.a` contains one more module, the package manager. Defined in `express.c` and `express.h`, this module includes calls to intialize the entire Express Working Form package, and to run each of the passes of a Fed-X translator.

3.4 Code Organization and Conventions

Each abstraction is implemented as a separate module. Modules share only their interface specifications with other modules. There is one exception to this rule: In order to avoid logistical problems compiling circular type definitions across modules, an Express Working Form module includes any other Working Form modules it uses *after* defining its own private `struct`. Thus, the types defined by these other modules are not yet known at the time an abstraction's private `struct` is defined, and references to these other Working Form types must assume knowledge of their implementations. This is, in fact, not a serious limitation: Each Working Form types is implemented as an `Object`, which *is* defined when the `struct` is compiled.

1. During the generation of the Working Form, many `Symbols` are not abstract supertypes.

A module Foo is composed of two C source files, `foo.c` and `foo.h`. The former contains the body of the module, including all non-inlined functions. The latter contains function prototypes for the module, as well as all type and macro definitions. In addition, global variables are defined here, using a mechanism which allows the same declarations to be used both for `extern` declarations in other modules and the actual storage definition in the declaring module. These globals can also be given constant initializers. Finally, `foo.h` contains inline function definitions. In a compiler which supports inline functions, these are declared `static inline` in every module which `#includes foo.h`, including `foo.c` itself. In other compilers, they are undefined except when included in `foo.c`, when they are compiled as ordinary functions.

The type defined by module Foo is named `Foo`, and its private structure is `struct Foo`. Access functions are named as `FOOfunction()`; this function prefix is abbreviated for longer abstraction names, so that access functions for type `Foolhardy_Bartender` might be of the form `FOO_BARfunction()`. Some functions may be implemented as macros; these macros are not distinguished typographically from other functions, and are guaranteed not to have unpleasant side effects like evaluating arguments more than once. These macros are thus virtually indistinguishable from functions. Functions which are intended for internal use only are named `FOO_function()`, and are usually `static` as well, unless this is not possible. Global variables are often named `FOO_variable`; most enumeration identifiers and constants are named `FOO_CONSTANT` (although these latter two rules are by no means universal). For example, every abstraction defines a constant `FOO_NULL`, which represents an empty or missing value of the type.

If an instance of `Foo` might contain unresolved `Symbols`, then there is a function `FOOresolve(...)`, called during Fed-X's second pass, which attempts to resolve all such references and reports any errors found. This call may or may not require a `Scope` as a parameter, depending on the abstraction. For example, an `Algorithm` defines its own local `Scope`, from which the next outer `Scope` (in which the `Algorithm` is defined) can be determined; `ALGresolve()` thus requires no `Scope` parameter. A `Type`, on the other hand, has no way of getting at its `Scope`, so `TYPEresolve()` requires a second parameter indicating the `Scope` in which the `Type` is to be resolved.

3.5 Memory Management and Garbage Collection

In reading various portions of the Express Working Form documentation, one may get the impression that the Working Form does some reasonably intelligent memory management. This is not entirely true. The NIST PDES Toolkit is primarily a research tool. This is especially true of the Express and STEP Working Forms. The Working Forms allocate huge chunks of memory without batting an eye, and often this memory is not released until an application exits. Hooks for doing memory management do exist (e.g., `OBJfree()` and reference counts), and some attempt is made to observe them, but this is not given high priority in the current implementation.

3.6 Default Print Routines

The library provides default print routines. This is oriented towards producing human-readable text and can be overridden by defining a new subroutine by the same name. However, as is, it provides a reasonable means of interactively browsing through the Working Form, especially if the Working Form is 'broken', such as when Fed-X itself is being debugged.

The following discussion assumes you are printing a Fed-X object from within gdb, the GNU debugger.

Every class has a 'print' function

3.6.1 Printing Unknown Objects

Thus, to print out an object, say:

```
p OBJprint(obj)
```

This is useful if you have no idea what the object is.

3.6.2 Printing Known Objects or Specific Classes of Objects

If you know 'obj' is a scope (or is a subclass of scope), you can also just say:

```
p SCOPEprint(obj)
```

For example, you can print out just the scope of an entity as:

```
p SCOPEprint(entity)
```

Alternatively, if you already have a handle to the hidden structure, you can directly print it out as:

```
p SCOPE_print(scope)
```

(You can not print out the scope of an entity this way, since the hidden forms do not inherit anything by themselves.)

Dataless classes may not necessarily have a print function, but can use print functions defined for classes that have private data.

3.6.3 Printing Specific Object Attributes

Each class has a special variable called '`X_print`' (for example '`scope_print`') which determines which attributes of the scope are printed. For example, if you want scope references to be printed, do:

```
set scope_print.references = 1
set scope_print.self = 1
```

Element 'self' is 0 (no attributes), 1 (some), or 2 (all). By default, it is set to 1 for linked lists, dictionaries and symbols, and 0 for all other classes. By default, all other elements are set to 1 (which means print, 0 means don't print). If 'self' is 0, it is forced to 1 when printed by its high level print function. (In other words, `SCOPEprint(object)` will force the scope to be printed, while `OBJprint(object)` will print only if `scope_print` says to.)

Except for the 'self' element, element names are exactly the same names as the names used in the hidden types. Classes that have only one attribute use a common print structure type with only a 'self' element.)

For convenience, the prefix of the print structure (i.e., '`scope`' in '`scope_print`' is the same as the prefix used in the low-level functions (e.g., '`aggr_lit_print`' is used rather '`aggregate_literal_print`').

3.6.4 Global Printing Options

The structure 'Print' provides some additional control. Attributes are as follows:

'header' controls whether header information such as class names are printed. By default, header is 1 meaning only the most specific class is described. 0 disables class descriptions, while 2 forces all class descriptions to be printed. Class specific data is printed after each class header.

'depth_max' controls the depth of object recursion. By default, the depth is 2.

'debug' controls whether internal functioning of the print routines themselves are printed. This is only useful if you have some doubts about the correct functioning of the print routines. Incorrect function has always turned out to be the case of something else having sabotaged the environment, so this 'debug' element is more useful for reassuring yourself that the environment (stack, heap, whatever) has not been corrupted.

Other elements in 'Print' are of value only to the implementation.

3.6.5 Printing to a File

By default, output is printed to the standard output. To redirect this to a file, say:

```
p OBJprint_file("foo")
```

To redirect back to the standard output and close the current output file:

```
p OBJprint_file((char *)0)
```

4 Writing An Output Module

It is expected that a common use of the Express WF will be to build Express translators. The Fed-X control flow was designed with this application in mind. A programmer who wishes to build such a translator need only write an output module for the target language. We now turn to the topic of writing this output module. The end result of the process described will be an object module (under Unix, a `.o` file) which can be loaded into Fed-X. This module contains a single entry point which traverses a given `Schema` and writes its output to a particular file.

The stylistic convention taken in the existing output modules, and which meshes most cleanly with the design of the Working Form data structures, is to define a procedure `FOOprint(Foo foo, FILE* file)` corresponding to each Working Form abstraction. Thus, `SCHEMAprint(Schema schema, FILE* file)` is the conceptual entry point to the output module; an `Algorithm` is written by the call `ALGprint(Algorithm algorithm, FILE* file)`, etc. With this breakdown, most of the actual output is generated by the routines for `Type`, `Entity`, and other concrete Express constructs. The routines for `Schema` and `Scope`, on the other hand, control the traversal of the data structures, and produce little or no actual output. For this reason, it is probably useful to base new report generators on existing ones, copying the traversal logic wholesale and modifying only the routines for the concrete objects.

Note that the library has default definitions of object print routines, although they are primarily for the purpose of producing human-readable descriptions. These may be overridden by supplying new definitions as suggested above. Note, however, that overriding a built-in print routine may cause misbehavior of other built-in print routines which depend on it.

4.1 Layout of the C Source

The layout of the C source file for a report generator which will be dynamically loaded is of critical importance, due to the primitive level at which the load is carried out. The very first piece of C source in the file must be the `entry_point()` function, or the loader may find the wrong entry point to the file, resulting in mayhem. Only comments may precede this function; even an `#include` directive may throw off the loader. An output module is normally laid out as shown:

```
void
entry_point(void* schema, void* file)
{
	extern void print_file();
	print_file(schema, file);
}

#include "express.h"

... actual output routines . ..

void
print_file(void* schema, void* file)
{
	print_file_header((Schema)schema,
```

```
                (FILE*)file);
        SCHEMAprint((Schema)schema, (FILE*)file);
        print_file_trailer((Schema)schema,
                (FILE*)file);
}
```

The `print_file()` function will probably always be quite similar to the one shown, although in many cases, the file header and/or trailer may well be empty, eliminating the need for these calls. In this case, `SCHEMAprint()` and `print_file()` will probably become interchangeable.

Having said all of the above about templates, code layout, and so forth, we add the following note: In the final analysis, the output module really is a free-form piece of C code. There is one and only one rule which must be followed, and this only if the report generator will be dynamically loaded: The entry point (according to the `a.out` format) to the `.o` file which is produced when the report generator is compiled must be appropriate to be called with a `Schema` and a `FILE*`. The simplest (and safest) way of doing this is to adhere strictly to the layout given, and write an `entry_point()` routine which jumps to the real (conceptual) entry point. But any other mechanism which guarantees this property may be used. Similarly, the layout of the rest of the code is purely conventional. There is no *a priori* reason to write one output routine per data structure, or to use the `print_file()` routine suggested. This approach has simply proved to work nicely for current and past report generators, and seems to provide the shortest path to a new output module. In other words, if you don't like the authors' coding style(s), feel free to use your own techniques.

4.2 Traversing a Schema

Following the one-routine-per-abstraction rule, there are two general classes of output routines. Those corresponding to primitive Express constructs (`ENTITYprint()`, `TYPEprint()`, `VARprint()`) will produce most of the actual output, while `SCOPEprint()` (and, to a lesser extent `SCHEMAprint()`) will be responsible for traversing the instantiated working form. A typical definition for `SCOPEprint()` would be:

```
void
SCOPEprint(Scope scope, FILE* file)
{
        Linked_List list;

        list = SCOPEget_types(scope);
        LISTdo(list, type, Type)
                TYPEprint(type, file);
        LISTod;
        LISTfree(list);

        list = SCOPEget_entities(scope);
```

```
            LISTdo(list, ent, Entity)
                 ENTITYprint(ent, file);
            LISTod;
            LISTfree(list);

            list = SCOPEget_algorithms(scope);
            LISTdo(list, alg, Algorithm)
                 ALGprint(alg, file);
            LISTod;
            LISTfree(list);

            list = SCOPEget_variables(scope);
            LISTdo(list, var, Variable)
                 VARprint(var, file);
            LISTod;
            LISTfree(list);

            list = SCOPEget_schemata(scope);
            LISTdo(list, schema, Schema)
                 SCEMAprint(schema, file);
            LISTod;
            LISTfree(list);
       }
```

This function traverses the model from the outermost schema inward. All types, entities, algorithms, and variables in a schema are printed (in that order), followed by all definitions for any sub-schemas. The only traversal logic required in `SCHEMAprint()` is simply to call `SCOPEprint()`.

An approach which is taken in the Fed-X-QDES output module is to divide the logical functionality of `SCOPEprint()` into two separate passes, implemented by functions `SCOPEprint_pass1()` and `SCOPEprint_pass2()`. The first pass prints all of the entity definitions, in superclass order (i.e., subclasses are not printed until after their superclasses), without attributes. This is necessary because of some difficulties with forward references in Smalltalk-80. The second pass then looks much like the sample definition of `SCOPEprint()` given above. This multi-pass strategy could also be used to print, for example, all of the type and entity definitions in the entire model, followed by all variable and algorithm definitions.

4.3 Working Form Routines

The remainder of this manual consists of specifications and brief descriptions of the access routines and associated error codes for the Express Working Form. Each subsection below corresponds to a module in the Working Form library. The Working Form Manager module is listed first, followed by the remaining data abstractions in alphabetical order.

The error codes are manipulated by the Error module [Clark90d]. Only error codes unique to each routine, are listed after each description.

4.4 Working Form Manager

Type: Express

Procedure: EXPRESSdump_model
Parameters: Express model - Express model to dump
Returns: void
Description: Dump an Express model to `stderr`. This call is provided for debugging purposes.

Procedure: EXPRESSfree
Parameters: Express model - Express model to free
Returns: void
Description: Release an Express model. Indicates that the model is no longer used by the caller; if there are no other references to the model, all storage associated with it may be released.

Procedure: EXPRESSinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Express package. This call in turn initializes all components of the Working Form package. Normally, it is called instead of calling all of the individual `XXXinitialize()` routines. In a typical Express (or STEP) translator, this function is called by the default `main()` provided in the Working Form library. Other applications should call it at initialization time.

Procedure: EXPRESSpass_1
Parameters: FILE* file - Express source file to parse
Returns: Express - resulting Working Form model
Description: Parse an Express source file into the Working Form. No symbol resolution is performed

Procedure: EXPRESSpass_2
Parameters: Express model - Working Form model to resolve
Returns: void
Description: Perform symbol resolution on a loosely-coupled Working Form model (which was probably created by `EXPRESSpass_1()`).

Procedure: EXPRESSpass_3
Parameters: Express model - Working Form model to report
FILE* file - output file
Returns: void
Description: Invoke one (or more) report generator(s), according to the selected linkage mechanism.

Procedure: PASS2initialize
Parameters: -- none --
Returns: void
Description: Initialize the Fed-X second pass.

4.5 Algorithm

Type: Algorithm
Supertype: Scope
Subtypes: Function, Procedure, Rule

Procedure: ALGget_body
Parameters: Algorithm algorithm - algorithm to examine
Returns: Linked_List - body of algorithm
Description: Retrieve the code body of an algorithm. The elements of the list returned are `Statements`.

Procedure: ALGget_name
Parameters: Algorithm algorithm - algorithm to examine
Returns: String - the name of the algorithm
Description: Retrieve the name of an algorithm.

Procedure: ALGget_parameters
Parameters: Algorithm algorithm - algorithm to examine
Returns: Linked_List - formal parameter list
Description: Retrieve the formal parameter list for an algorithm. When `ALGget_class(algorithm) == ALG_RULE`, the returned list contains the `Entitys` to which the rule applies. Otherwise, it contains `Variables` specifying the formal parameters to the function or procedure.

Procedure: ALGinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Algorithm module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: ALGprint
Parameters: Algorithm
Returns: void
Description: Prints an algorithm. Exactly what is printed can be controlled by setting various elements of the variable alg_print.

Procedure: ALGput_body
Parameters: Algorithm algorithm - algorithm to modify
Linked_List statements - body of algorithm
Returns: void
Description: Set the code body of an algorithm. The second parameter should be a list of `Statements`.

Procedure: ALGput_name
Parameters: Algorithm algorithm - algorithm to modify
String name - new name for algorithm
Returns: void
Description: Set the name of an algorithm.

Procedure: ALGput_parameters
Parameters: Algorithm algorithm - algorithm to modify
Linked_List list - formal parameters for this algorithm
Returns: void
Description: Set the formal parameter list of an algorithm. When `ALGget_class(algorithm) == ALG_RULE`, the formal parameters should be the `Entitys` to which the rule applies. Otherwise, they should be `Variables`.

Procedure: ALGresolve
Parameters: Algorithm algorithm - algorithm to resolve
Scope scope - scope in which to resolve
Returns: void
Description: Resolve all references in an algorithm definition. This is called, in due course, by `EXPRESSpass_2()`.

Procedure: FUNCget_return_type
Parameters: Function function - function to examine
Returns: Type - function's return type
Description: Return the type of the function.

Procedure: FUNCprint
Parameters: Function
Returns: void
Description: Prints a function. Exactly what is printed can be controlled by setting various elements of the variable func_print.

Procedure: FUNCput_return_type
Parameters: Function function - function to modify
Type type - the function's return type
Returns: void
Description: Set the return type of a function.

Procedure: RULEget_where_clause
Parameters: Rule rule - rule to examine
Returns: Linked_List - list of rule's WHERE clause constraints
Description: Return the where clause of a rule.

Procedure: RULEprint
Parameters: Rule
Returns: void
Description: Prints a rule. Exactly what is printed can be controlled by setting various elements of the variable rule_print.

Procedure: RULEput_where_clause
Parameters: Rule rule - rule to modify
Linked_List where - list of WHERE clause constraints for rule
Returns: void
Description: Set the where clause of a rule

4.6 Case Item

Type: Case_Item
Supertype: Construct

Procedure: CASE_ITcreate
Parameters: Linked_List of Expression labels - list of case labels
Statement statement - statement associated with this branch
Error* errc - buffer for error code
Returns: Case_Item - the case item created
Description: Create a new case item. If the 'labels' parameter is `LIST_NULL`, a case item matching in the default case is created. Otherwise, the case item created will match when the case selector has the same value as any of the `Expressions` on the labels list.

Procedure: CASE_ITget_labels
Parameters: Case_Item item - case item to examine
Returns: Linked_List - list of case labels
Description: Retrieve the list of label `Expressions` for which a case item matches. For an item which matches in the default case, `LIST_NULL` is returned.

Procedure: CASE_ITget_statement
Parameters: Case_Item item - the case item to examine
Returns: Statement - statement associated with this branch
Description: Retrieve the statement to be executed when this case item is matched.

Procedure: CASE_ITinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Case Item module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: CASE_ITprint
Parameters: Case_Item
Returns: void
Description: Prints a Case_Item. Exactly what is printed can be controlled by setting various elements of the variable case_it_print.

Procedure: CASE_ITresolve
Parameters: Case_Item item - case item to resolve
Scope scope - scope in which to resolve
Returns: void
Description: Resolve all symbol references in a case item. This is called, in due course, by `EXPRESSpass_2()`.

4.7 Constant

Type: Constant
Supertype: Symbol

Procedure: CSTcreate
Parameters: String name - name of new constant
Type type - type of new constant
Generic value - value for new constant
Returns: Constant - the constant created
Description: Create a new constant.

Procedure: CSTget_name
Parameters: Constant constant - constant to examine
Returns: String - the name of the constant
Description: Return the name of a constant.

Procedure: CSTget_type
Parameters: Constant constant - constant to examine
Returns: Type - the type of the constant
Description: Return the type of a constant.

Procedure: CSTget_value
Parameters: Constant constant - constant to examine
Returns: Generic - the value of the constant
Description: Return the value of a constant.

Procedure: CSTinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Constant module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: CSTprint
Parameters: Constant
Returns: void
Description: Prints a Constant. Exactly what is printed can be controlled by setting various elements of the variable cst_print.

Procedure: CSTput_name
Parameters: Constant constant - constant to modify
String - name for constant
Returns: void
Description: Set the name of a constant

Procedure: CSTput_type
Parameters: Constant constant - constant to modify
Type - type for constant
Returns: void
Description: Set the type of a constant

Procedure: CSTput_value
Parameters: Constant constant - constant to modify
Generic - value of constant
Returns: void
Description: Set the value of a constant

4.8 Construct

Type: Construct
Supertype: -- none --

Procedure: CONSTRget_line_number
Parameters: Construct construct - construct to examine
Returns: int - line number of construct
Description: Return the line number of a construct.

Procedure: CONSTRinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Construct module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: CONSTRprint
Parameters: Construct
Returns: void
Description: Prints a construct. Exactly what is printed can be controlled by setting various elements of the variable constr_print.

Procedure: CONSTRput_line_number
Parameters: Construct construct - construct to modify
int number - line number for construct
Returns: void
Description: Set a construct's line number.

4.9 Entity

Type: Entity
Supertype: Scope

Procedure: ENTITYadd_attribute
Parameters: Entity entity - entity to modify
Variable attribute - attribute to add
Returns: void
Description: Adds an attribute to the entity.

Procedure: ENTITYadd_instance
Parameters: Entity entity - entity to modify
Generic instance - new instance
Returns: void
Description: Adds an instance of the entity.

Procedure: ENTITYdelete_instance
Parameters: Entity entity - entity to modify
Generic instance - instance to delete
Returns: void
Description: Deletes an instance of the entity.

Procedure: ENTITYget_abstract
Parameters: Entity
Returns: Boolean
Description: returns boolean defining when entity is abstract or not

Procedure: ENTITYget_all_attributes
Parameters: Entity entity - entity to examine
Returns: Linked_List of Variable - all attributes of this entity
Description: Retrieve the complete attribute list of an entity. The attributes are ordered as required by the STEP Physical File format [Part21]. This list should be `LISTfree`'d when no longer needed.

Procedure: ENTITYget_attribute_offset
Parameters: Entity entity - entity to examine
Variable attribute - attribute to retrieve offset for
Returns: int - offset to given attribute
Description: Retrieve offset to an entity attribute. This offset takes into account all superclass of the entity:. it is computed by `ENTITYget_initial_offset(entity) + VARget_offset(attribute)`. If the entity does not include the attribute, -1 is returned. This call should be preferred over `ENTITYget_named_attribute_offset()`.

Procedure: ENTITYget_attributes
Parameters: Entity entity - entity to examine
Returns: Linked_List of Variable - local attributes of this entity
Description: Retrieve the local attribute list of an entity. The local attributes of an entity are those which are defined by the entity itself (rather than being inherited from supertypes). This list should be `LISTfree`'d when no longer needed.

Procedure: ENTITYget_constraints
Parameters: Entity entity - entity to examine
Returns: Linked_List of Expression - this entity's constraints
Description: Retrieve the list of constraints from an entity's "where" clause. This list should not be `LISTfree`'d.

Procedure: ENTITYget_initial_offset
Parameters: Entity entity - entity to examine
Returns: int - number of inherited attributes
Description: Retrieve the initial offset to an entity's local frame. This is the total number of explicit attributes inherited from supertypes.

Procedure: ENTITYget_instances
Parameters: Entity entity - entity to examine
Returns: Linked_List - list of instances of the entity
Description: Retrieve an entity's instance list. This list should not be `LISTfree`'d.

Procedure: ENTITYget_mark
Parameters: Entity entity - entity to examine
Returns: int - entity's current mark
Description: Retrieve an entity's mark. See `ENTITYput_mark()`.

Procedure: ENTITYget_name
Parameters: Entity entity - entity to examine
Returns: String - entity name
Description: Return the name of an entity.

Procedure: ENTITYget_named_attribute
Parameters: Entity entity - entity to examine
String name - name of attribute to retrieve
Returns: Variable - the named attribute of this entity
Description: Retrieve the definition of an entity attribute by name. If the entity has no attribute with the given name, `VARIABLE_NULL` is returned.

Procedure: ENTITYget_named_attribute_offset
Parameters: Entity entity - entity to examine
String name - name of attribute for which to retrieve offset
Returns: int - offset to named attribute of this entity
Description: Retrieve the offset to an entity attribute by name. If the entity has no attribute with the given name, -1 is returned. This call is slower than `ENTITYget_attribute_offset()`, and so should be avoided when the actual attribute definition is already available.

Procedure: ENTITYget_size
Parameters: Entity entity - entity to examine
Returns: int - storage size of instantiated entity
Description: Compute the storage size of an instantiation of this entity. This is the total number of attributes which it contains.

Procedure: ENTITYget_subtype
Parameters: Entity
String
Returns: Entity
Description: Given name, returns subtype

Procedure: ENTITYget_subtype_expression
Parameters: Entity entity - entity to examine
Returns: Expression - immediate subtype expression
Description: Retrieve the controlling expression for an entity's immediate subtype list.

Procedure: ENTITYget_subtypes
Parameters: Entity entity - entity to examine
Returns: Linked_List of Entity - immediate subtypes of this entity
Description: Retrieve a list of an entity's immediate subtypes.

Procedure: ENTITYget_supertype
Parameters: Entity
String
Returns: Entity
Description: Given name, returns supertype

Procedure: ENTITYget_supertypes
Parameters: Entity entity - entity to examine
Returns: Linked_List of Entity - immediate supertypes of this entity
Description: Retrieve a list of an entity's immediate supertypes. This list should not be `LISTfree`'d.

Procedure: ENTITYget_uniqueness_list
Parameters: Entity entity - entity to examine
Returns: Linked_List of Linked_List - this entity's uniqueness sets
Description: Retrieve an entity's uniqueness list. Each element of this list is itself a list of `Variables`, specifying a uniqueness set for the entity. The uniqueness list should not be `LISTfree`'d, nor should any of the component lists.

Procedure: ENTITYhas_immediate_subtype
Parameters: Entity parent - entity to check children of
Entity child - child to check for
Returns: Boolean - is child a direct subtype of parent?

Procedure: ENTITYhas_immediate_supertype
Parameters: Entity child - entity to check parentage of
Entity parent - parent to check for
Returns: Boolean - is parent a direct supertype of child?

Procedure: ENTITYhas_subtype
Parameters: Entity parent - entity to check descendants of
Entity child - child to check for
Returns: Boolean - does parent's subclass tree include child?

Procedure: ENTITYhas_supertype
Parameters: Entity child - entity to check parentage of
Entity parent - parent to check for
Returns: Boolean - does child's superclass chain include parent?

Procedure: ENTITYinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Entity module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: ENTITYprint
Parameters: Entity
Returns: void
Description: Prints an Entity. Exactly what is printed can be controlled by setting various elements of the variable entity_print.

Procedure: ENTITYput_abstract
Parameters: Entity
Boolean
Returns: void
Description: Define an entity to be abstract or not.

Procedure: ENTITYput_constraints
Parameters: Entity entity - entity to modify
Linked_List constraints - list of constraints which entity must satisfy
Returns: void
Description: Set the constraints on an entity. The elements of the constraints list should be `Expressions` of type `TY_LOGICAL`.

Procedure: ENTITYput_inheritance_count
Parameters: Entity entity - entity to modify
int count - number of inherited attributes
Returns: void
Description: Set the number of attributes inherited by an entity. This should be computed automatically (perhaps only when needed), and this call removed. The count is currently computed by `ENTITYresolve()`.

Procedure: ENTITYput_mark
Parameters: Entity entity - entity to modify
int value - new mark for entity
Returns: void
Description: Set an entity's mark. This mark is used, for example, in `SCOPE_dfs()`, part of `SCOPEget_entities_superclass_order()`, to mark each entity as having been touched by the traversal.

Procedure: ENTITYput_name
Parameters: Entity entity - entity to modify
String name - entity's name
Returns: void
Description: Set the name of an entity.

Procedure: ENTITYput_subtypes
Parameters: Entity entity - entity to modify
Expression expression - controlling subtype expression
Returns: void
Description: Set the (immediate) subtypes list of an entity.

Procedure: ENTITYput_supertypes
Parameters: Entity entity - entity to modify
Linked_List list - superclass entities
Returns: void
Description: Set the (immediate) supertype list of an entity. The elements of the list should be `Entitys` or (unresolved) `Symbols`.

Procedure: ENTITYput_uniqueness_list
Parameters: Entity entity - entity to modify
Linked_List list - uniqueness list
Returns: void
Description: Set the uniqueness list of an entity. Each element of the uniqueness list should itself be a list of `Variables` and/or (unresolved) `Symbols` referencing entity attributes. Each of these sublists specifies a single uniqueness set for the entity.

Procedure: ENTITYresolve
Parameters: Entity entity - entity to resolve
Returns: void
Description: Resolve all symbol references in an entity definition. This function is called, in due course, by `EXPRESSpass_2()`.

4.10 Expression

Type: Expression
Supertype: Construct

Private Type: Ary_Expression
Supertype: Expression

Type: Binary_Expression
Supertype: Ary_Expression

Type: Ternary_Expression
Supertype: Ary_Expression

Type: Unary_Expression
Supertype: Ary_Expression

Type: One_Of_Expression
Supertype: Expression

Type: Function_Call
Supertype: One_Of_Expression

Type: Identifier
Supertype: Expression

Private Type: Literal
Supertype: Expression

Type: Aggregate_Literal
Supertype: Literal

Type: Binary_Literal
Supertype: Literal

Type: Integer_Literal
Supertype: Literal

Type: Logical_Literal
Supertype: Literal

Type: Real_Literal
Supertype: Literal

Type: String_Literal
Supertype: Literal

Type: Query
Supertype: Expression

Constant: LITERAL_E - a real literal with the value 2.18281...
Type: Real_Literal

Constant: LITERAL_EMPTY_SET - a generic set literal representing the empty set
Type: Aggregate_Literal

Constant: LITERAL_INFINITY - a numeric literal representing infinity
Type: Integer_Literal

Constant: LITERAL_PI - a real literal with the value 3.1415...
Type: Real_Literal

Constant: LITERAL_ZERO - an integer literal with the value 0
Type: Integer_Literal

Procedure: AGGR_LITcreate
Parameters: Type type - type of aggregate literal to be created
Linked_List value - value for literal
Error* errc - buffer for error code
Returns: Aggregate_Literal - the literal created
Description: Create an aggregate literal expression.

Procedure: AGGR_LITget_value
Parameters: Aggregate_Literal literal - aggregate literal to examine
Error* errc - buffer for error code
Returns: Linked_List of Generic - the literal's contents
Description: Retrieve the value of an aggregate literal, as a list.

Procedure: AGGR_LITprint
Parameters: Aggregate_Literal
Returns: void
Description: Prints an Aggregate_Literal. Exactly what is printed can be controlled by setting various elements of the variable aggr_lit_print.

Procedure: ARY_EXPget_operand
Parameters: Ary_Expression operand
Returns: Unary Expression - the expression created
Description: Create a unary operation expression

Procedure: ARY_EXPget_operator
Parameters: Ary_Expression
Returns: Op_Code
Description: Return operator of expression

Procedure: ARY_EXPprint
Parameters: Ary_Expression
Returns: void
Description: Prints an Ary_Expression. Exactly what is printed can be controlled by setting various elements of the variable ary_exp_print.

Procedure: ARY_EXPput_operand
Parameters: Ary_Expression - Unary expression to modify
Expression - Expression to become new operand
Returns: void
Description: Modifies the operand of a unary expression

Procedure: BIN_EXPcreate
Parameters: Op_Code op - operation
Expression operand1 - first operand
Expression operand2 - second operand
Error* errc - buffer for error code
Returns: Binary_Expression - the expression created
Description: Create a binary operation expression.

Procedure: BIN_EXPget_first_operand
Parameters: Binary_Expression expression - expression to examine
Returns: Expression - the first (left-hand) operand of the expression
Description: Return first operand of binary expression.

Procedure: BIN_EXPget_operator
Parameters: Binary_Expression expression - expression to examine
Returns: Op_Code - the operator invoked by the expression
Description: Return operator of binary expression.

Procedure: BIN_EXPget_second_operand
Parameters: Binary_Expression expression - expression to examine
Returns: Expression - the second (right-hand) operand of the expression
Description: Return second operand of binary expression.

Procedure: BIN_EXPprint
Parameters: Bin_Expression
Returns: void
Description: Prints an Bin_Expression. Exactly what is printed can be controlled by setting various elements of the variable bin_exp_print.

Procedure: BIN_LITcreate
Parameters: Binary
Error *
Returns: Binary_Literal
Description: Creates a binary literal

Procedure:	BIN_LITget_value
Parameters:	Binary_Literal Error *
Returns:	Binary
Description:	Returns the binary corresponding to the binary_literal

Procedure:	BIN_LITprint
Parameters:	Binary_Literal
Returns:	void
Description:	Prints an Binary_Literal. Exactly what is printed can be controlled by setting various elements of the variable bin_lit_print.

Procedure:	EXPas_string
Parameters:	Expression expression - expression to print as string
Returns:	String - string representation of expression
Description:	Generate the string representation of an expression. Only (qualified) identifiers are currently supported.

Procedure:	EXPget_integer_value
Parameters:	Expression expression - expression to evaluate Error* errc - buffer for error code
Returns:	int - value of expression
Description:	Compute the value of an integer expression. Currently, only integer literals can be evaluated; other classes of expressions evaluate to 0 and produce a warning message. `EXPRESSION_NULL` evaluates to 0, as well.
Errors:	`ERROR_integer_expression_expected`

Procedure:	EXPget_type
Parameters:	Expression expression - expression to examine
Returns:	Type - the type of the value computed by the expression

Procedure:	EXPinitialize
Parameters:	-- none --
Returns:	void
Description:	Initialize the Expression module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure:	EXPprint
Parameters:	Expression
Returns:	void
Description:	Prints an Expression. Exactly what is printed can be controlled by setting various elements of the variable exp_print.

Procedure:	EXPput_type
Parameters:	Expression expression - expression to modify Type type - the type of result computed by the expression
Returns:	void
Description:	Set the type of an expression. This call should actually be unnecessary: the type of an expression is derivable from its definition. While this is currently true in the case of literals, there are no rules in place for deriving the type from, for example, the return type of a function or an operator together with its operands.

Procedure: EXPresolve
Parameters: Expression expression - expression to resolve
Scope scope - scope in which to resolve
Returns: void
Description: Resolve all symbol references in an expression. This is called, in due course, by `EXPRESSpass_2()`.

Procedure: EXPresolve_qualification
Parameters: Expression expression - expression to resolve
Scope scope - scope in which to resolve
Error* errc - buffer for error code
Returns: Symbol - the symbol referenced by the expression
Description: Retrieves the symbol definition referenced by a (possibly qualified) identifier.

Procedure: FCALLcreate
Parameters: Algorithm algorithm - algorithm invoked by expression
Linked_List parameters - actual parameters to function call
Error* errc - buffer for error code
Returns: Function_Call - the function call created
Description: Create a function call expression.
Errors: -- none --

Procedure: FCALLget_algorithm
Parameters: Function_Call expression - function call expresion to examine
Returns: Algorithm - the algorithm invoked by the function call
Description: Retrieves the algorithm of the function call.

Procedure: FCALLget_parameters
Parameters: Function_Call expression - function call expression to examine
Returns: Linked_List of Expression - list of actual parameters
Description: Retrieve the actual parameter `Expressions` from a function call expression.

Procedure: FCALLprint
Parameters: Function_Call
Returns: void
Description: Prints a Function_Call. Exactly what is printed can be controlled by setting various elements of the variable fcall_print.

Procedure: FCALLput_algorithm
Parameters: Function_Call expression - function call expression to modify
Algorithm algorithm - algorithm invoked by expression
Returns: void
Description: Set the algorithm invoked by a function call expression.

Procedure: FCALLput_parameters
Parameters: Function_Call expression - function call expression to modify
Linked_List parameters - list of actual parameters
Returns: void
Description: Set the actual parameter list to a function call expression. The elements of the parameter list should be `Expressions`. The types of the actual parameters currently are not verified against the formal parameter list of the called algorithm.

Procedure: IDENTcreate
Parameters: Symbol ident - identifier referenced by expression
Error* errc - buffer for error code
Returns: Identifier - the identifier expression created
Description: Create a simple identifier expression.

Procedure: IDENTget_identifier
Parameters: Identifier expression - expression to examine
Returns: Symbol - the identifier referenced in the expression

Procedure: IDENTprint
Parameters: Identifier
Returns: void
Description: Prints an Identifier. Exactly what is printed can be controlled by setting various elements of the variable ident_print.

Procedure: IDENTput_identifier
Parameters: Identifier expression - identifier expression to modify
Symbol identifier - the referent of the identifier
Returns: void
Description: Set the referent of an identifier expression.

Procedure: INT_LITcreate
Parameters: Integer value - value for literal
Error* errc - buffer for error code
Returns: Integer_Literal - the literal created
Description: Create an integer literal expression.

Procedure: INT_LITget_value
Parameters: Integer_Literal literal - integer literal to examine
Error* errc - buffer for error code
Returns: Integer - the literal's value

Procedure: INT_LITprint
Parameters: Integer_Literal
Returns: void
Description: Prints an Integer_Literal. Exactly what is printed can be controlled by setting various elements of the variable int_lit_print.

Procedure: LOG_LITcreate
Parameters: Logical value - value for literal
Error* errc - buffer for error code
Returns: Logical_Literal - the literal created
Description: Create a logical literal expression.

Procedure: LOG_LITget_value
Parameters: Logical_Literal literal - logical literal to examine
Error* errc - buffer for error code
Returns: Logical - the literal's value

Procedure: LOG_LITprint
Parameters: Logical_Literal
Returns: void
Description: Prints a Logical_Literal. Exactly what is printed can be controlled by setting various elements of the variable log_lit_print.

Procedure: ONEOFcreate
Parameters: Linked_List selections - list of selections for oneof()
Error* errc - buffer for error code
Returns: One_Of_Expression - the oneof expression created
Description: Create a oneof() expression.

Procedure: ONEOFget_selections
Parameters: One_Of_Expression expression - expression to examine
Returns: Linked_List of Expression - list of selections for oneof()

Procedure: ONEOFprint
Parameters: One_Of_Expression
Returns: void
Description: Prints a One_Of_Expression. Exactly what is printed can be controlled by setting various elements of the variable oneof_print.

Procedure: ONEOFput_selections
Parameters: One_Of_Expression expression - expression to modify
Linked_List selections - list of selections for oneof()
Returns: void
Description: Set the list of selections for a oneof() expression.

Procedure: opcode_print
Parameters: Op_Code
Returns: void
Description: Despite the name, this function returns a string describing the opcode.

Procedure: OPget_number_of_operands
Parameters: Op_Code operation - the opcode to query
Returns: int - number of operands required by this operator.

Procedure: QUERYcreate
Parameters: String ident - local identifier for source elements
Expression source - source aggregate to query
Expression discriminant - discriminating expression for query
Error* errc - buffer for error code
Returns: Query - the query expression created
Description: Create a query expression.

Procedure: QUERYget_discriminant
Parameters: Query expression - query expression to examine
Returns: Expression - the discriminant expression
Description: Retrieves the discriminant expression from a query expression. The discriminant expresses the query criteria.

Procedure: QUERYget_source
Parameters: Query expression - query expression to examine
Returns: Expression - the source aggregation
Description: Retrieves the expression which computes the aggregation against which a query will be applied.

Procedure: QUERYget_variable
Parameters: Query expression - query expression to examine
Returns: Variable - the local iteration variable of the query

Procedure: QUERYprint
Parameters: Query Expression
Returns: void
Description: Prints a Query Expression. Exactly what is printed can be controlled by setting various elements of the variable query_print.

Procedure: REAL_LITcreate
Parameters: Real value - value for literal
Error* errc - buffer for error code
Returns: Real_Literal - the literal created
Description: Create a real literal expression.

Procedure: REAL_LITget_value
Parameters: Real_Literal literal - real literal to examine
Error* errc - buffer for error code
Returns: Real - the literal's value

Procedure: REAL_LITprint
Parameters: Real_Literal
Returns: void
Description: Prints a Real_Literal. Exactly what is printed can be controlled by setting various elements of the variable real_lit_print.

Procedure: STR_LITcreate
Parameters: String value - value for literal
Error* errc - buffer for error code
Returns: String_Literal - the literal created
Description: Create a string literal expression.

Procedure: STR_LITget_value
Parameters: String_Literal literal - string literal to examine
Error* errc - buffer for error code
Returns: String - the literal's value

Procedure: STR_LITprint
Parameters: String_Literal
Returns: void
Description: Prints a String_Literal. Exactly what is printed can be controlled by setting various elements of the variable str_lit_print.

Procedure: TERN_EXPcreate
Parameters: Op_Code
Expression
Expression
Expression
Error *
Returns: Ternary_Expression
Description: Creates and returns a ternary expression

Procedure: TERN_EXPget_second_operand
Parameters: Ternary_Expression
Returns: Expression
Description: Returns second operand of a ternary expression

Procedure: TERN_EXPget_third_operand
Parameters: Ternary_Expression
Returns: Expression
Description: Returns third operand of a ternary expression

Procedure: TERN_EXPprint
Parameters: Ternary_Expression
Returns: void
Description: Prints a Ternary_Expression. Exactly what is printed can be controlled by setting various elements of the variable tern_exp_print.

Procedure: UN_EXPcreate
Parameters: Op_Code op - operation
Expression operand - operand
Error* errc - buffer for error code
Returns: Unary_Expression - the expression created
Description: Create a unary operation expression.

Procedure: UN_EXPget_operand
Parameters: Unary_Expression expression - expression to examine
Returns: Expression - the operand of the expression

Procedure: UN_EXPget_operator
Parameters: Unary_Expression expression - expression to examine
Returns: Op_Code - the operator invoked by the expression

4.11 Loop Control

Type: Loop_Control
Supertype: Construct

Type: Increment_Control
Supertype: Loop_Control

Private Type: Conditional_Control
Supertype: Loop_Control

Type: Until_Control
Supertype: Conditional_Control

Type: While_Control
Supertype: Conditional_Control

Procedure: INCR_CTLcreate
Parameters: Expression control - controlling expression
Expression start - initial value
Expression end - terminal value
Expression increment - amount by which to increment
Error* errc - buffer for error code
Returns: Increment_Control - the loop control created

Procedure: INCR_CTLprint
Parameters: Increment_Control
Returns: void
Description: Prints an Increment_Control. Exactly what is printed can be controlled by setting various elements of the variable incr_ctl_print.

Procedure: UNTILcreate
Parameters: Expression control - termination condition
Error* errc - buffer for error code
Returns: Until - the loop control created
Requires: OBJis_kind_of(EXPget_type(control), Class_Logical_Type)
Errors: `ERROR_control_boolean_expected` - controlling expression is not logical

Procedure: WHILEcreate
Parameters: Expression control - continuation condition
Error* errc - buffer for error code
Returns: While - the loop control created
Requires: OBJis_kind_of(EXPget_type(control), Class_Logical_Type)
Errors: `ERROR_control_boolean_expected` - controlling expression is not logical

Procedure: LOOP_CTLget_controlling_expression
Parameters: Loop_Control control - loop control to examine
Returns: Expression - controlling expression
Description: Retrieve a loop control's controlling expression. For while and until controls, this is the termination or continuation condition, respectively. For iteration and set scan controls, this is the expression which receives successive values in the iteration.

Procedure: LOOP_CTLprint
Parameters: Loop_Control
Returns: void
Description: Prints a Loop_Control. Exactly what is printed can be controlled by setting various elements of the variable loop_ctl_print.

Procedure: INCR_CTLget_final
Parameters: Increment_Control control - increment control to examine
Returns: Expression - terminal value for controlling expression
Description: Retrieve the final value from an increment control.

Procedure: INCR_CTLget_increment
Parameters: Increment_Control control - increment control to examine
Returns: Expression - amount to increment by on each iteration
Description: Retrieve the increment expression from an increment control.

Procedure: INCR_CTLget_start
Parameters: Increment_Control control - increment control to examine
Returns: Expression - initial expression for controlling expression
Description: Retrieve the initial value from an increment control.

Procedure: LOOP_CTLinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Loop Control module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: LOOP_CTLresolve
Parameters: Loop_Control control - control to resolve
Scope scope - scope in which to resolve
Returns: void
Description: Resolve all symbol references in a loop control. This is called, in due course, by `EXPRESSpass_2()`.

4.12 Reference

Procedure: REFERENCEresolve
Parameters: Scope
Returns: void
Description: resolves all references in a scope.

4.13 Schema

Type: Schema
Supertype: Scope

Type: Schemas
Supertype: Dictionary

Procedure: SCHEMAcreate
Parameters: String name - name of schema to create
Scope scope - local scope for schema
Error* errc - buffer for error code
Returns: Schema - the schema created
Description: Create a new schema.

Procedure: SCHEMAdump
Parameters: Schema schema - schema to dump
FILE* file - file to dump to
Returns: void
Description: Dump a schema to a file. This function is provided for debugging purposes.

Procedure: SCHEMAget_name
Parameters: Schema schema - schema to examine
Returns: String - the schema's name

Procedure: SCHEMAinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Schema module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: SCHEMAresolve
Parameters: Schema schema - schema to resolve
Schemas schemas - all schemas in the Express file
Returns: void
Description: Resolve all symbol references within a schema. In order to avoid problems due to references to as-yet-unresolved symbols, schema resolution is broken into two passes, which are implemented by `SCHEMAresolve_pass1()` and `SCHEMAresolve_pass2()`. These two are called in turn by `SCHEMAresolve()`.

4.14 Scope

Type: Scope
Supertype: Symbol

Procedure: SCOPEadd_reference
Parameters: Scope
Linked_List
Returns: void
Description: Adds a list of references (from one REFERENCE statement) to an entity.

Procedure: SCOPEadd_use
Parameters: Scope
Linked_List
Returns: void
Description: Adds a list of references (from one USE statement) to an entity.

Procedure: SCOPEadd_superscope
Parameters: Scope scope - scope to modify
Scope parent - additional parent scope
Returns: void
Description: Adds an immediate parent to a scope.

Procedure: SCOPEcreate
Parameters: Scope scope - next higher scope
Returns: Scope - the scope created
Description: Create an empty scope. Note that the connection between this new scope and its parent (the sole parameter to this call) is uni-directional: the parent does not immediately know about the child.

Procedure: SCOPEdefine_symbol
Parameters: Scope scope - scope in which to define symbol
Symbol symdef - new symbol definition
Error* errc - buffer for error code
Returns: void
Description: Define a symbol in a scope.
Errors: Reports all errors directly, so only `ERROR_subordinate_failed` is propagated.

Procedure: SCOPEdump
Parameters: Scope scope - scope to dump
FILE* file - file stream to dump to
Returns: void
Description: Dump a schema to a file. This function is provided for debugging purposes.

Procedure: SCOPEget_algorithms
Parameters: Scope scope - scope to examine
Returns: Linked_List - list of locally defined algorithms
Description: Retrieve a list of the algorithms defined locally in a scope. The elements of this list are `Algorithms`. The list should be `LISTfree`'d when no longer needed.

Procedure: SCOPEget_constants
Parameters: Scope scope - scope to examine
Returns: Linked_List - list of locally defined constants
Description: Retrieve a list of the constants defined locally in a scope. The elements of this list are `Constants`. The list should be `LISTfree`'d when no longer needed.

Procedure: SCOPEget_entities
Parameters: Scope scope - scope to examine
Returns: Linked_List - list of locally defined entities
Description: Retrieve a list of the entities defined locally in a scope. The elements of this list are `Entitys`. The list should be `LISTfree`'d when no longer needed. This function is considerably faster than `SCOPEget_entities_superclass_order()`, and should be used whenever the order of the entities on the list is not important.

Procedure: SCOPEget_entities_superclass_order
Parameters: Scope scope - scope to examine
Returns: Linked_List - list of locally defined entities in superclass order
Description: Retrieve a list of the entities defined locally in a scope. The elements of this list are `Entitys`. The list should be `LISTfree`'d when no longer needed. The list returned is ordered such that each entity appears before all of its subtypes.

Procedure: SCOPEget_imports
Parameters: Scope scope - scope to examine
Returns: Linked_List - 'assumed' schemata
Description: Retrieve a list of the schemata assumed in a scope. The elements of this list are `Schemas`. The list should <u>not</u> be `LISTfree`'d.

Procedure: SCOPEget_references
Parameters: Scope
Returns: Dictionary
Description: All the references (from all the REFERENCE statements) of an entity.

Procedure: SCOPEget_resolved
Parameters: Scope scope - scope to examine
Returns: Boolean - has this scope been resolved?
Description: Check whether symbol references in a scope have been resolved.

Procedure: SCOPEget_superscopes
Parameters: Scope scope - scope to examine
Returns: Linked_List - list of next outer (containing) scopes
Description: Retrieve a list of a scope's parent scope.

Procedure: SCOPEget_types
Parameters: Scope scope - scope to examine
Returns: Linked_List - list of locally defined types
Description: Retrieve a list of the types defined locally in a scope. The elements of this list are `Types`. The list should be `LISTfree`'d when no longer needed.

Procedure: SCOPEget_uses
Parameters: Scope
Returns: Linked_List
Description: Returns a list of all references (from USE statements) from an entity.

Procedure: SCOPEget_variables
Parameters: Scope scope - scope to examine
Returns: Linked_List - list of locally defined variables
Description: Retrieve a list of the variables defined locally in a scope. The elements of this list are `Variables`. The list should be `LISTfree`'d when no longer needed.

Procedure: SCOPEinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Scope module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: SCOPElookup
Parameters: Scope scope - scope in which to look up name
String name - name to look up
Boolean walk - look in parent and imported scopes?
Error* errc - buffer for error code
Returns: Symbol - definition of name in scope
Description: Retrieve a name's definition in a scope. If the scope does not define the name, the parent scopes are successively queried. If no definition is found, `SYMBOL_NULL` is returned.
Errors: `ERROR_undefined_identifier` - no definition was found

Procedure: SCOPEprint
Parameters: Scope
Returns: void
Description: Prints a Scope. Exactly what is printed can be controlled by setting various elements of the variable scope_print.

Procedure: SCOPEput_resolved
Parameters: Scope scope - scope to modify
Returns: void
Description: Set the 'resolved' flag for a scope. This normally should only be called by `SCOPEresolve()`, which actually resolves the scope.

Procedure: SCOPEresolve
Parameters: Scope scope - scope to resolve
Schemas schemas - all conceptual schemas in the express file
Returns: void
Description: Resolve all symbol references in a scope. In order to avoid problems due to references to as-yet-unresolved symbols, scope resolution is broken into two passes, which are implemented by `SCOPEresolve_pass1()` and `SCOPEresolve_pass2()`. These two are called in turn by `SCOPEresolve()`.

4.15 Statement

Private Type: Statement
Supertype: Construct

Type: Assignment
Supertype: Statement

Type: Compound_Statement
Supertype: Statement

Type: Conditional
Supertype: Statement

Type: Loop
Supertype: Statement

Type: Procedure_Call
Supertype: Statement

Type: Return_Statement
Supertype: Statement

Type: With_Statement
Supertype: Statement

Procedure: ASSIGNcreate
Parameters: Expression lhs - the left-hand-side of the assignment
Expression rhs - the right-hand-side of the assignment
Error* errc - buffer for error code
Returns: Assignment - the assignment statement created
Description: Create an assignment statement.

Procedure: ASSIGNget_lhs
Parameters: Assignment statement - statement to examine
Returns: Expression - left-hand-side of assignment statement
Description: Return left-hand-side of the assignment statement.

Procedure: ASSIGNget_rhs
Parameters: Assignment statement - statement to examine
Returns: Expression - right-hand-side of assignment statement
Description: Return right-hand-side of the assignment statement.

Procedure: ASSIGNprint
Parameters: Assignment statement
Returns: void
Description: Prints an assignment statement. Exactly what is printed can be controlled by setting various elements of the variable assign_print.

Procedure: CASEcreate
Parameters: Expression selector - expression to case on
Linked_List case - list of case branches
Error* errc - buffer for error code
Returns: Case_Statement - the case statement created
Description: Create a case statement. The elements of the case branch list should be `Case_Items`.

Procedure: CASEget_items
Parameters: Case_Statement statement - statement to examine
Returns: Linked_List - case branches
Description: Retrieve a list of the branches in a case statement. The elements of this list are `Case_Items`.

Procedure: CASEget_selector
Parameters: Case_Statement statement - statement to examine
Returns: Expression - the selector for the case statment
Description: Retrieve the selector from a case statement. This is the expression whose value is compared to each case label in turn.

Procedure: CASEprint
Parameters: Case_Statement
Returns: void
Description: Prints a case statement. Exactly what is printed can be controlled by setting various elements of the variable case_print.

Procedure: COMP_STMTcreate
Parameters: Linked_List statements - list of compound statement elements
Error* errc - buffer for error code
Returns: Compound_Statement - the compound statement created
Description: Create a compound statement. The elements of the statements list should be `Statements`, in the order they appear in the compound statement to be represented.

Procedure: COMP_STMTget_items
Parameters: Compound_Statement statement - statement to examine
Returns: Linked_List - list of statements in compound
Description: Retrieve a list of the `Statements` comprising a compound statement.

Procedure: COMP_STMTprint
Parameters: Compound_Statement
Returns: void
Description: Prints a compound statement. Exactly what is printed can be controlled by setting various elements of the variable comp_stmt_print.

Procedure: CONDcreate
Parameters: Expression test - the condition for the if
Statement then - code executed when test == true
Statement otherwise - code executed when test == false
Error* errc - buffer for error code
Returns: Conditional - the if statement created
Description: Create an if statement. For a simple `if .. then ..` with no else clause, set the third parameter to `STATEMENT_NULL`.

Procedure: CONDget_else_clause
Parameters: Conditional statement - statement to examine
Returns: Statement - code for 'else' branch

Procedure: CONDget_condition
Parameters: Conditional statement - statement to examine
Returns: Expression - the test condition

Procedure: CONDget_then_clause
Parameters: Conditional statement - statement to examine
Returns: Statement - code for 'then' branch

Procedure: CONDprint
Parameters: Conditional statement
Returns: void
Description: Prints a conditional statement. Exactly what is printed can be controlled by setting various elements of the variable cond_print.

Procedure: LOOPcreate
Parameters: Linked_List controls - list of controls for the loop
Statement body - statement to be repeated
Error* errc - buffer for error code
Returns: Loop - the loop statement created
Description: Create a loop statement. The elements of the controls list should be `Loop_Controls`.

Procedure: LOOPget_body
Parameters: Loop statement - statement to examine
Returns: Statement - the body of the loop
Description: Retrieve the body (repeated portion) of a loop statement

Procedure: LOOPget_controls
Parameters: Loop statement - statement to examine
Returns: Linked_List - list of loop controls
Description: Retrieve a list of a loop statement's controls. The elements of this list are `Loop_Controls`.

Procedure: LOOPprint
Parameters: Loop statement
Returns: void
Description: Prints a loop statement. Exactly what is printed can be controlled by setting various elements of the variable loop_print.

Procedure: PCALLcreate
Parameters: Procedure procedure - procedure called by statement
Linked_List parameters - list of actual parameters
Error* errc - buffer for error code
Returns: Procedure_Call - the procedure call created
Description: Create a procedure call statement. The elements of the actual parameter list should be `Expressions` which compute the values to be passed to the procedure.

Procedure: PCALLget_procedure
Parameters: Procedure_Call statement - statement to examine
Returns: Procedure - procedure called by this statement
Description: Retrieve the procedure called by a procedure call statement.

Procedure: PCALLget_parameters
Parameters: Procedure_Call statement - statement to examine
Returns: Linked_List - actual parameters to this call
Description: Retrieve the actual parameters for a procedure call statement. The elements of this list are `Expressions` which compute the values to be passed to the called routine.

Procedure: PCALLprint
Parameters: Procedure_Call statement
Returns: void
Description: Prints a Procedure_Call statement. Exactly what is printed can be controlled by setting various elements of the variable pcall_print.

Procedure: PCALLput_procedure
Parameters: Procedure_Call statement - statement to modify
Procedure procedure - definition of called procedure
Returns: void
Description: Set the actual procedure called by a procedure call statement. If a procedure stub (unresolved `Symbol`) is present in the statement, it is replaced such that all references remain valid.

Procedure: RETcreate
Parameters: Expression expression - expression to compute return value
Error* errc - buffer for error code
Returns: Return_Statement - the return statement created
Description: Create a return statement.

Procedure: RETget_expression
Parameters: Return_Statement statement - statement to examine
Returns: Expression - expression returned by this statement
Description: Retrieve the expression whose value is computed and returned by a return statement.

Procedure: RETprint
Parameters: Return statement
Returns: void
Description: Prints a Return statement. Exactly what is printed can be controlled by setting various elements of the variable return_print.

Procedure: STMTinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Statement module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: STMTresolve
Parameters: Statement statement - statement to resolve
Scope scope - scope in which to resolve
Returns: void
Description: Resolve all symbol references in a statement. This is called, in due course, by `EXPRESSpass_2()`.

Procedure: WITHcreate
Parameters: Expression expression - controlling expression for the with
Statement body - controlled statement for the with
Error* errc - buffer for error code
Returns: With_Statement - the with statement created
Description: Create a with statement.

Procedure: WITHget_body
Parameters: With_Statement statement - statement to examine
Returns: Statement - statement forming the body of the with statement

Procedure: WITHget_control
Parameters: With_Statement statement - statement to examine
Returns: Expression - the controlling expression
Description: Retrieve the controlling expression from a with statement. This is the expression which will be prepended to any expression which cannot otherwise be evaluated in the current scope.

4.16 Symbol

Type: Symbol
Supertype: -- none --

Procedure: SYMBOLget_line_number
Parameters: Symbol symbol - symbol to examine
Returns: int - line number of symbol

Procedure: SYMBOLget_name
Parameters: Symbol symbol - symbol to examine
Returns: String - name of symbol

Procedure: SYMBOLget_resolved
Parameters: Symbol symbol - symbol to examine
Returns: Boolean - is the symbol resolved?
Description: Test whether a symbol has been resolved.

Procedure: SYMBOLinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Symbol module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: SYMBOLprint
Parameters: Symbol
Returns: void
Description: Prints a Symbol. Exactly what is printed can be controlled by setting various elements of the variable symbol_print.

Procedure: SYMBOLput_line_number
Parameters: Symbol symbol - symbol to modify
int number - line number for symbol
Returns: void
Description: Set a symbol's line number.

Procedure: SYMBOLput_name
Parameters: Symbol symbol - symbol to name
String name - name of symbol
Returns: void
Description: Set the name of a symbol.

Procedure: SYMBOLput_resolved
Parameters: Symbol symbol - symbol to mark resolved
Returns: void
Description: Mark a symbol as being resolved. This is normally called by the client `XXXput_resolved()` functions, since a symbol cannot itself be resolved.

4.17 Type

Private Type: Type
Supertype: Symbol

Type: Aggregate_Type
Supertype: Type

Type: Array_Type
Supertype: Aggregate_Type

Type: Bag_Type
Supertype: Aggregate_Type

Type: Binary_Type
Supertype: Type

Type: List_Type
Supertype: Aggregate_Type

Type: Set_Type
Supertype: Aggregate_Type

Private Type: Composed_Type
Supertype: Type

Type: Entity_Type
Supertype: Composed_Type

Type: Enumeration_Type
Supertype: Composed_Type

Type: Select_Type
Supertype: Composed_Type

Type: Generic_Type
Supertype: Type

Type: Logical_Type
Supertype: Type

Type: Boolean_Type
Supertype: Logical_Type

Type: Number_Type
Supertype: Type

Private Type: Sized_Type
Supertype: Type

Type: Integer_Type
Supertype: Sized_Type

Type: Real_Type
Supertype: Sized_Type

Type: String_Type
Supertype: Sized_Type

Type: Type_Reference
Supertype: Type

Constant: TYPE_AGGREGATE
Description: Type for general aggregate of generic.

Constant: TYPE_BINARY
Description: Binary type.

Constant: TYPE_BOOLEAN
Description: Boolean type.

Constant: TYPE_GENERIC
Description: The type 'generic.'

Constant: TYPE_INTEGER
Description: Integer type with default precision.

Constant: TYPE_LOGICAL
Description: Logical type.

Constant: TYPE_META
Description: Meta type (for TYPEOF expressions).

Constant: TYPE_NUMBER
Description: Number type.

Constant: TYPE_REAL
Description: Real type with default precision.

Constant: TYPE_SET_OF_GENERIC
Description: Type for unconstrained set of generic.

Constant: TYPE_STRING
Description: String type with default precision (length).

Procedure: AGGR_TYPEget_optional
Parameters: Aggregate_Type type - type to examine
Returns: Boolean - are elements of this aggregate optional?
Description: Retrieve the 'optional' flag from an aggregate type. This flag is true if and only if a legal instantiation of the type need not have all of its slots filled.

Procedure: AGGR_TYPEget_unique
Parameters: Aggregate_Type type - type to examine
Returns: Boolean - must elements of this aggregate be unique?
Description: Retrieve the 'unique' flag from an aggregate type. This flag is true if and only if a legal instantiation of the type may not contain duplicates.

Procedure: AGGR_TYPEget_base_type
Parameters: Aggregate_Type type - type to examine
Returns: Type - the base type of the aggregate type
Description: Retrieve the base type of an aggregate. This is the type of each element of an instantiation of the type.

Procedure: AGGR_TYPEget_lower_limit
Parameters: Aggregate_Type type - type to examine
Returns: Expression - lower limit of the aggregate type
Description: Retrieve an aggregate type's lower bound. For an array type, this is the lowest index; for other aggregate types, it specifies the minimum number of elements which the aggregate must contain.

Procedure: AGGR_TYPEget_upper_limit
Parameters: Aggregate_Type type - type to examine
Returns: Expression - upper limit of the aggregate type
Description: Retrieve an aggregate type's upper bound. For an array type, this is the high index; for other aggregate types, it specifies the maximum number of elements which the aggregate may contain.

Procedure: AGGR_TYPEprint
Parameters: Aggregate_Type
Returns: void
Description: Prints an Aggregate_Type. Exactly what is printed can be controlled by setting various elements of the variable aggr_type_print.

Procedure: AGGR_TYPEput_optional
Parameters: Aggregate_Type type - type to modify
Boolean optional - are array elements optional?
Returns: void
Description: Set the 'optional' flag for an array type. This flag indicates that all slots in an instance of the type need not be filled.

Procedure: AGGR_TYPEput_unique
Parameters: Aggregate_Type type - type to modify
Boolean unique - are aggregate elements required to be unique?
Returns: void
Description: Set the 'unique' flag for an aggregate type. This flag indicates that an instantiation of the type may not contain duplicate items.

Procedure: AGGR_TYPEput_base_type
Parameters: Aggregate_Type type - type to modify
Type base - the base type for this aggregate
Returns: void
Description: Set the base type of an aggregate type. This is the type of every element.

Procedure: AGGR_TYPEput_limits
Parameters: Aggregate_Type type - type to modify
Expression lower - lower bound for aggregate
Expression upper - upper bound for aggregate
Returns: void
Description: Set the lower and upper bounds for an aggregate type. For an array type, these are the low and high indices; for other aggregates, these specify the minimum and maximum number of elements which an instance may contain.

Procedure: COMP_TYPEadd_items
Parameters: Composed_Type
Linked_List
Returns: void
Description: Add to the list of items for a Composed_Type.

Procedure: COMP_TYPEget_items
Parameters: Composed_Type
Returns: Linked_List of Symbol
Description: Retrieve a composed types list of identifiers.

Procedure: COMP_TYPEprint
Parameters: Composed_Type
Returns: void
Description: Prints a Composed_Type. Exactly what is printed can be controlled by setting various elements of the variable comp_type_print.

Procedure: COMP_TYPEput_items
Parameters: Composed_Type
Linked_List
Returns: void
Description: Set the list of items for a Composed_Type.

Procedure: ENT_TYPEget_entity
Parameters: Entity_Type type - type to examine
Returns: Entity - definition of entity type
Description: Retrieve the (first) entity referenced by an entity type.

Procedure: ENT_TYPEget_entity_list
Parameters: Entity_Type type - type to examine
Returns: Linked_List - definition of entity type
Description: Retrieve a list of the entities referenced by an entity type.

Procedure: ENT_TYPEput_entity
Parameters: Entity_Type type - type to modify
Entity entity - definition of type
Returns: void
Description: Set the entity referred to by an entity type.

Procedure: ENT_TYPEput_entity_list
Parameters: Entity_Type type - type to modify
Linked_List - definition of type
Returns: void
Description: Set the list of entities referred to by an entity type.

Procedure: ENUM_TYPEget_items
Parameters: Enumeration_Type type - type to examine
Returns: Linked_List - list of enumeration items
Description: Retrieve an enumerated type’s list of identifiers. Each element of this list is a `Constant`.

Procedure: ENUM_TYPEput_items
Parameters: Enumeration_Type type - type to modify
Linked_List list - list of enumeration items
Returns: void
Description: Set the list of identifiers for an enumerated type. Each element of this list should be a `Constant`.

Procedure: SEL_TYPEget_items
Parameters: Select_Type type - type to examine
Returns: Linked_List - list of selectable types
Description: Retrieve a list of the selectable types from a select type.

Procedure: SEL_TYPEput_items
Parameters: Select_Type type - type to modify
Linked_List list - list of selectable types
Returns: void
Description: Set the list of selections for a select type. An instance of any these types is a legal instantiation of the select type. Each `Type` on the list should be of class `TYPE_ENTITY` or `TYPE_SELECT`.

Procedure: SZD_TYPEget_precision
Parameters: Sized_Type type - type to examine
Returns: Expression - the precision specification of the type
Description: Retrieve the precision specification from certain types. This specifies the maximum number of significant digits or characters in an instance of the type.

Procedure: SZD_TYPEget_varying
Parameters: Sized_Type type - type to examine
Returns: Boolean - is the string type of varying length?
Description: Retrieve the 'varying' flag from a string type. This flag is true if and only if the length of an instance may vary, up to the type's precision. It is true by default.

Procedure: SZD_TYPEprint
Parameters: Sized_Type
Returns: void
Description: Prints a Sized_Type. Exactly what is printed can be controlled by setting various elements of the variable szd_type_print.

Procedure: SZD_TYPEput_precision
Parameters: Sized_Type type - type to modify
Expression prec - the precision of the type
Returns: void
Description: Set the precision of certain types. This is the maximum number of significant digits or characters in an instance.

Procedure: SZD_TYPEput_varying
Parameters: Sized_Type type - type to modify
Boolean varying - is string type of varying length?
Returns: void
Description: Set the 'varying' flag of a string type. This flag indicates that the length of an instance may vary, up to the type's precision. The default behavior for a string type is to be varying, i.e., strings are initialized as if TYPEput_varying(string, true) were called.

Procedure: TYPEcompatible
Parameters: Type lhs_type - type for left-hand-side of assignment
Type rhs_type - type for right-hand-side of assignment
Returns: Boolean - are the types assignment compatible?
Description: Determine whether two types are assignment-compatible. It must be possible to assign a value of `rhs_type` into a slot of `lhs_type`.

Procedure: TYPEget_name
Parameters: Type type - type to examine
Returns: String - the name of the type
Description: Return the name of the type.

Procedure: TYPEget_original_type
Parameters: Type type
Returns: Type
Description: returns the original type, allowing a way to see through TYPE declarations.

Procedure: TYPEget_size
Parameters: Type type - type to examine
Returns: int - logical size of a type instance
Description: Compute the size of an instance of some type. Simple types all have size 1, as does a select type. The size of an aggregate type is the maximum number of elements an instance can contain; and the size of an entity type is its total attribute count. If an aggregate type is unbounded, the constant `TYPE_UNBOUNDED_SIZE` is returned. This value may be ambiguous; the upper bound of the type should be relied on to determined unboundedness. It is intended that the initial memory allocation for such an aggregate should give space for `TYPE_UNBOUNDED_SIZE` elements, and that this should grow as needed. By returning some reasonable initial size, this call allows its return value to be used immediately as a parameter to a memory allocator, without being checked for validity. This is the approach taken in the STEP Working Form [Clark90d], [Clark90e].

Procedure: TYPEget_where_clause
Parameters: Type type - type to examine
Returns: Linked_List - the type's WHERE clause
Description: Retrieve the WHERE clause associated with a type. Each element of the returned list will be an `Expression` which computes a `Logical` result.

Procedure: TYPEinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Type module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: TYPEprint
Parameters: Type
Returns: void
Description: Prints a Type. Exactly what is printed can be controlled by setting various elements of the variable type_print.

Procedure: TYPEput_name
Parameters: Type type - type to modify
String name - new name for type
Returns: void
Description: Set the name of a type.

Procedure: TYPEput_original_type
Parameters: TYPE new_type
TYPE original_type
Returns: void
Description: Sets original type. See TYPEget_original_type.

Procedure: TYPEput_where_clause
Parameters: Type type - type to modify
Linked_List - the type's WHERE clause
Returns: void
Description: Set the WHERE clause associated with a type. Each element of the list should be an `Expression` which computes a `Logical` result.

Procedure: TYPEresolve
Parameters: Type type - type to resolve
Scope scope - scope in which to resolve
Returns: void
Description: Resolve all references in a type definition, and transform a type reference into the appropriate Type or Entity construct. This is called, in due course, by `EXPRESSpass_2().`

Procedure: TYPE_REFget_full_name
Parameters: Type_Reference type - type reference to examine
Returns: Expression - [qualified] identifier expression for type reference
Description: Retrieve the identifier expression for a type reference. This expression consists of identifier components assembled into binary expressions with `OP_DOT`.

Procedure: TYPE_REFprint
Parameters: Type_Reference
Returns: void
Description: Prints a Type_Reference. Exactly what is printed can be controlled by setting various elements of the variable type_ref_print.

Procedure: TYPE_REFput_full_name
Parameters: Type_Reference type - type reference to modify
Expression name - [qualified] identifier expression for type reference
Returns: void
Description: Set the identifier expression for a type reference.

4.18 Use

Procedure: USEresolve
Parameters: Scope
Returns: void
Description: resolves all references (from USE statements) in a scope.

4.19 Variable

Type: Variable
Supertype: Symbol

Procedure: VARcreate
Parameters: String name - name of variable to create
Type type - type of variable to create
Error* errc - buffer for error code
Returns: Variable - the Variable created
Description: Create a new variable. The reference class of the variable is, by default, `REF_DYNAMIC`. All special flags associated with the variable (e.g., optional) are initially false.

Procedure: VARget_derived
Parameters: Variable var - variable to examine
Returns: Boolean - value of variable's derived flag
Description: Retrieve the value of a variable's 'derived' flag. This flag indicates that an entity attribute's value should always be computed by its initializer; no value will ever be specified for it.

Procedure: VARget_initializer
Parameters: Variable var - variable to modify
Returns: Expression - variable initializer
Description: Retrieve the expression used to initialize a variable.

Procedure: VARget_inverse
Parameters: Variable
Returns: Symbol
Description: Returns inverse relationship of a variable. Typically used after resolution, this will be either a Set_Type or an Identifier of the entity of the variable.

Procedure: VARget_name
Parameters: Variable var - variable to examine
Returns: String - the name of the variable

Procedure: VARget_offset
Parameters: Variable var - variable to examine
Returns: int - offset to variable in local frame
Description: Retrieve the offset to a variable in its local frame. This offset alone is not sufficient in the case of an entity attribute (see `ENTITYget_attribute_offset()`).

Procedure: VARget_optional
Parameters: Variable var - variable to examine
Returns: Boolean - value of variable's optional flag
Description: Retrieve the value of a variable's 'optional' flag. This flag indicates that a particular entity attribute need not have a value when the entity is instantiated.

Procedure: VARget_type
Parameters: Variable var - variable to examine
Returns: Type - the type of the variable

Procedure: VARget_variable
Parameters: Variable var - variable to examine
Returns: Boolean - value of variable's variable flag
Description: Retrieve the value of a variable's 'variable' flag. This flag indicates that an algorithm parameter is to be passed by reference, so that it can be modified by the callee.

Procedure: VARinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Variable module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: VARprint
Parameters: Variable
Returns: void
Description: Prints a Variable. Exactly what is printed can be controlled by setting various elements of the variable var_print.

Procedure: VARput_derived
Parameters: Variable var - variable to modify
Boolean val - new value for derived flag
Returns: void
Description: Set the value of the 'derived' flag for a variable. This flag is currently redundant, as a derived attribute can be identified by the fact that it has an initializing expression. This may not always be true, however.

Procedure: VARput_initializer
Parameters: Variable var - variable to modify
Expression init - initializer
Returns: void
Description: Set the initializing expression for a variable.

Procedure: VARput_inverse
Parameters: Variable
Symbol
Returns: void
Description: Set inverse relationship for a variable. See VARget_inverse.

Procedure: VARput_offset
Parameters: Variable var - variable to modify
int offset - offset to variable in local frame
Returns: void
Description: Set a variable's offset in its local frame. Note that in the case of an entity attribute, this offset is *from the first locally defined attribute*, and must be used in conjunction with entity's initial offset (see `ENTITYget_attribute_offset()`).

Procedure: VARput_optional
Parameters: Variable var - variable to modify
Boolean val - value for optional flag
Returns: void
Description: Set the value of the 'optional' flag for a variable. This flag indicates that a particular entity attribute need not have a value when the entity is instantiated. It is initially false.

Procedure: VARput_type
Parameters: Variable
Type
Returns: void
Description: Set the type of a variable.

Procedure: VARput_variable
Parameters: Variable var - variable to modify
Boolean val - new value for variable flag
Returns: void
Description: Set the value of the 'variable' flag for a variable. This flag indicates that an algorithm parameter is to be passed by reference, so that it can be modified by the callee.

Procedure: VARresolve
Parameters: Variable variable - variable to resolve
Scope scope - scope in which to resolve
Returns: void
Description: Resolve all symbol references in a variable definition. This is called, in due course, by `EXPRESSpass_2()`.

5 Express Working Form Error Codes

The Error module, which is used to manipulate these error codes, is described in [Clark90c].

Error: ERROR_bail_out
Defined In: Express
Severity: SEVERITY_DUMP
Meaning: Fed-X internal error
Format: -- none --

Error: ERROR_control_boolean_expected
Defined In: Loop_Control
Severity: SEVERITY_WARNING
Meaning: The controlling expression for a while or until does not seem to return boolean. In the current implementation, this message can be erroneously produced because proper types are not derived for complex expressions; thus, an expression which truly does compute a boolean result may not appear to do so according to the Working Form.
Format: -- none --

Error: ERROR_corrupted_expression
Defined In: Expression
Severity: SEVERITY_DUMP
Meaning: Fed-X internal error: an Expression structure was corrupted
Format: %s - function detecting error

Error: ERROR_corrupted_statement
Defined In: Statement
Severity: SEVERITY_DUMP
Meaning: Fed-X internal error: a Statement structure was corrupted
Format: %s - function detecting error

Error: ERROR_corrupted_type
Defined In: Type
Severity: SEVERITY_DUMP
Meaning: Fed-X internal error: a Type structure was corrupted
Format: %s - function detecting error

Error: ERROR_duplicate_declaration
Defined In: Scope
Severity: SEVERITY_ERROR
Meaning: A symbol was redeclared in the same scope
Format: %s - name of redeclared symbol
%d - line number of previous declaration

Error: ERROR_inappropriate_use
Defined In: Scope
Severity: SEVERITY_ERROR
Meaning: A symbol was used in a context which is inappropriate for its declaration.
Format: %s - the name of the symbol

Error: ERROR_include_file
Defined In: Scanner
Severity: SEVERITY_ERROR
Meaning: An `INCLUDE`d file could not be opened.
Format: %s - the name of the file

Error: ERROR_integer_expression_expected
Defined In: Expression
Severity: SEVERITY_WARNING
Meaning: A non-integer expression was encountered in an integer-only context
Format: -- none --

Error: ERROR_integer_literal_expected
Defined In: Expression
Severity: SEVERITY_WARNING
Meaning: A non-integer or non-literal was encountered in an integer-literal context
Format: -- none --

Error: ERROR_logical_literal_expected
Defined In: Expression
Severity: SEVERITY_WARNING
Meaning: A non-logical or non-literal was encountered in a logical-literal context
Format: -- none --

Error: ERROR_missing_subtype
Defined In: Pass2
Severity: SEVERITY_WARNING
Meaning: An entity which lists a particular supertype does not appear in that entity's subtype list.
Format: %s - the name of the subtype
%s - the name of the supertype

Error: ERROR_missing_supertype
Defined In: Pass2
Severity: SEVERITY_ERROR
Meaning: An entity which lists a particular subtype does not appear in that entity's supertype list.
Format: %s - the name of the supertype
%s - the name of the subtype

Error: ERROR_nested_comment
Defined In: Scanner
Severity: SEVERITY_WARNING
Meaning: A start comment symbol `(*` was encountered within a comment.
Format: -- none --

Error: ERROR_overloaded_attribute
Defined In: Pass2
Severity: SEVERITY_ERROR
Meaning: An attribute name was previously declared in a supertype
Format: %s - the attribute name
%s - the name of the supertype with the previous declaration

Error: ERROR_real_literal_expected
Defined In: Expression
Severity: SEVERITY_WARNING
Meaning: A non-real or non-literal was encountered in a real-literal context
Format: -- none --

Error: ERROR_set_literal_expected
Defined In: Expression
Severity: SEVERITY_WARNING
Meaning: A non-set or non-literal was encountered in a set-literal context
Format: -- none --

Error: ERROR_set_scan_set_expected
Defined In: Loop_Control
Severity: SEVERITY_WARNING
Meaning: The control set for a set scan control is not a set
Format: -- none --

Error: ERROR_shadowed_declaration
Defined In: Pass2
Severity: SEVERITY_WARNING
Meaning: A symbol declaration shadows a definition in an outer (or assumed) scope.
Format: %s - name of redeclared symbol
%d - line number of previous declaration

Error: ERROR_string_literal_expected
Defined In: Expression
Severity: SEVERITY_WARNING
Meaning: A non-string or non-literal was encountered in a string-literal context
Format: -- none --

Error: ERROR_syntax
Defined In: Express
Severity: SEVERITY_EXIT
Meaning: Unrecoverable syntax error
Format: %s - description of error
%s - name of scope in which error occurred

Error ERROR_undefined_identifier
Defined In: Pass2
Severity: SEVERITY_WARNING
Meaning: An identifer was referenced which has not been declared. This error only produces a warning because Fed-X does not deal with all of the scoping issues in algorithms.
Format: %s - the name of the identifier

Error: ERROR_undefined_type
Defined In: Pass2
Severity: SEVERITY_ERROR
Meaning: An undeclared identifier was used in a context which requires a type.
Format: %s - the name of the type

Error: ERROR_unknown_expression_class
Defined In: Expression
Severity: SEVERITY_DUMP
Meaning: Fed-X internal error
Format: %d - the offending expression class
%s - the context (function) in which the error occurred

Error: ERROR_unknown_schema
Defined In: Pass2
Severity: SEVERITY_WARNING
Meaning: An unknown schema was ASSUMEd
Format: %s - the assumed schema name

Error: ERROR_unknown_subtype
Defined In: Pass2
Severity: SEVERITY_WARNING
Meaning: An entity lists a subtype which is not itself declared as an entity.
Format: %s - the subtype name
%s - the supertype name

Error: ERROR_unknown_supertype
Defined In: Pass2
Severity: SEVERITY_EXIT
Meaning: An entity lists a supertype which is not itself declared as an entity. Fed-X is unable to proceed in this situation.
Format: %s - the supertype name
%s - the subtype name

Error:	ERROR_unknown_type_class
Defined In:	Type
Severity:	SEVERITY_DUMP
Meaning:	Fed-X internal error
Format:	%d - the offending type class %s - the context (function) in which the error occurred

Error:	ERROR_wrong_operand_count
Defined In:	Expression
Severity:	SEVERITY_WARNING
Meaning:	Mismatch between actual and expected (on the basis of code context) operand count
Format:	%s - the operator

6 Building Fed-X

The Fed-X toolkit is distributed in two ways. The usual form is the latest release of the software. An alternate form is the RCS archives [Bodarky91] which contain all prior releases.

If you only have the latest release of the software, simply visit each directory named src and type 'make install'. This will create the necessary libraries. You may skip the rest of this section.

The following discussion assumes you have the RCS archives. To build the toolkit, you must find out where the archives are and where you would like to build the toolkit. This discussion assumes that the toolkit archives are stored in `~pdes` and you would like to build it in `~/pdes`.

First create the directory in which you are going to keep all your files.

```
mkdir ~/pdes
```

Check out a copy of `make_rules`.

```
cd ~/pdes
mkdir include
cd include
co ~pdes/include/make_rules
```

`make_rules` contains definitions common to all other parts of Fed-X as well as applications. If you examine it, you will find ways to customize the toolkit. For example, you can choose whether to use yacc or bison by changing this file. Only one change will be described in detail here. Namely, you must tell `make_rules` the directory in which you are keeping all your Fed-X code.

In order to make this change, start by making it writeable:

```
chmod +w make_rules
```

Change the definition of `PDES` to reflect the root of the directories where you have your Fed-X code stored. Note that Make does not understand the ~ notation – thus, you must provide the hardcoded path, which for this example is assumed to be `/home/fred`:

```
PDES=/home/fred/pdes
```

Fed-X will ultimately be stored in several libraries. A directory must be created to contain the libraries. It is created as follows:

```
mkdir -p ~/pdes/arch/lib
```

If you are using bison, you should now create or link the bison library to this directory. For example, to create the library from scratch:

```
cd ~/pdes/src/libbison
co CheckOut
CheckOut
make install
```

In order to build the libraries, several programs must exist. These live in `~pdes/bin` and it is normally sufficient to create a symbolic link between this and your own bin directory as:

```
ln -s ~pdes/bin ~/pdes/bin
```

If you already have a directory by that name, you may link the individual files:

```
ln -s ~/pdes/bin/* ~/pdes/bin
```

Fed-X is composed of sources in two directories and include files in two other directories. The following example extracts the files from all four directories. After running each `CheckOut`, expect a page or so of output as each file composing the toolkit is checked out. The command `make install` compiles the toolkit and installs the library version in the `arch/lib` directory created previously.

```
cd ~/pdes/include/libmisc
co CheckOut
CheckOut
cd ~/pdes/src/libmisc
co CheckOut
CheckOut
make install
cd ~/pdes/include/express
co CheckOut
CheckOut
cd ~/pdes/src/express
co CheckOut
CheckOut
```

```
make install
```

You can now build applications with Fed-X

7 Building Applications with Fed-X

Assuming the Fed-X toolkit has been built (as described in the previous section), building an application requires compiling and linking with the toolkit.

The easiest way to do this is copy the `Makefile` and main.c from an extant Fed-X application and modify it as necessary. For example, `fedex` is a very simple program that calls the toolkit to create a working form and do nothing else. To get `fedex`, create a directory for it and check out the code:

```
mkdir ~/pdes/src/fedex
cd ~/pdes/src/fedex
co CheckOut
CheckOut
```

If you want to compile `fedex` itself, run make:.

```
cd ~/pdes/src/fedex
make
```

Now you may copy the `Makefile` and `main.c` as appropriate for you application.

A References

[ANSI89] American National Standards Institute, Programming Language C, Document ANSI X3.159-1989.

[Bodarky91] Bodarky, S., A Guide to Configuration Management and the Revision Control System for Testbed Users, NISTIR 4646, August 1991.

[Clark90a] Clark, S. N., An Introduction to The NIST PDES Toolkit, NISTIR 4336, National Institute of Standards and Technology, Gaithersburg, MD, May 1990.

[Clark90b] Clark, S.N., Fed-X: The NIST Express Translator, NISTIR 4371, National Institute of Standards and Technology, Gaithersburg, MD, August 1990.

[Clark90c] Clark, S.N., Libes., D., The NIST PDES Toolkit: Technical Fundamentals, NISTIR 4335, National Institute of Standards and Technology, Gaithersburg, MD, March 1992.

[Clark90d] Clark, S.N., The NIST Working Form for STEP, NISTIR 4351, National Institute of Standards and Technology, Gaithersburg, MD, June 1990.

[Clark90e] Clark, S.N., NIST STEP Working Form Programmer's Reference, NISTIR 4353, National Institute of Standards and Technology, Gaithersburg, MD, June 1990.

[Mason 91] Mason, H., ed., Industrial Automation Systems – Product Data Representation and Exchange – Part 1: Overview and Fundamental Principles, Version 9, ISO TC184/SC4/WG PMAG Document N50, December 1991.

[Part21] ISO CD 10303 – 21, Product Data Representation and Exchange – Part 21, Clear Text Encoding of the Exchange Structure, ISO TC184/SC4 Document N78, February, 1991.

[Part11] ISO 10303-11 Description Mehods: The EXPRESS Language Reference Manual, ISO TC184/SC4 Document N14, April 1991.