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

NISTIR 4407

National PDES Testbed
Report Series

NIST Express Working Form Programmer's Reference

NATIONAL PDES TESTBED

NIST

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

NISTIR 4407

National PDES Testbed
Report Series

NIST Express Working Form Programmer's Reference

Stephen Nowland Clark

U.S. DEPARTMENT OF
COMMERCE
Robert A. Mosbacher,
Secretary of Commerce

National Institute of
Standards and Technology
John W. Lyons, Director

September 5, 1990

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.

Table Of Contents

NIST Express Working Form Programmer's Reference

Stephen Nowland Clark

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 [Schenck89]. 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 [Smith88]. 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 CALS (Computer-aided Acquisition and Logistic Support) 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 PDES 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.

For further information on the Express Working Form or other components of the Toolkit, or to obtain a copy of the software, use the attached order form.

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. This Working Form can be traversed by an output

module in the third. 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 large enough that UNIX Yacc's statically allocated tables cannot represent it. Bison, a Yacc clone available from the Free Software Foundation[1], has no such static limits, and so is used to build the parser. The lexical analyzer is processed by Flex, a fast, Public Domain implementation of Lex[2]. The analyzer makes use of one feature of Flex which us not present in Lex: it uses an exclusive start condition to scan comments properly. The scanner can easily be rewritten to use only standard start conditions if it is necessary to use Lex. Other differences between Lex and Flex are handled properly by conditional compilation (`#ifdef .. #endif` pairs).

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 interpreted as either, the parser always guesses 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`.

---

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.

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. In the dynamically linked version of Fed-X, this pass loads successive output modules, calling each one to traverse the Working Form. The dynamic linking mechanism is discussed briefly in [Clark90c]. It is also possible to build a statically linked translator, with a particular output module loaded in at build time; this is, at present, the only mechanism available in an environment which is not derived from BSD 4.2 UNIX.

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). In a dynamically linked translator, this object module is linked into the running parser, and its entry point (by convention a function called `print_file()`) is called. 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 ANSI Standard C [ANSI89]. Each abstraction except Schema is implemented as a `Symbol` or `Construct` header block (see section 3.2, below) with a pointer to a private `struct`. This C structure contains the real definition of the abstraction, but is never manipulated directly outside of the abstraction's module. For example:

```
/* the actual contents of a Foo */
struct Foo {
     int i;
     double d;
};
/* type Foo is a Construct whose definition */
/* field will point at a struct Foo */
typedef Construct Foo;
```

Outside of `Foo`'s module, we will never see a `struct Foo`. We will only see a `Foo`, which is actually a `Construct` which points at a `struct Foo`. This indirection makes bookkeeping and symbolic reference resolution easier to do. A `Schema`, being a very simple object, has a `Symbol` header block which points directly at a `Scope`, which is itself implemented as a `Construct`.

3.1 Primitive Types

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

3.2 Symbol and Construct

The types `Symbol` and `Construct` are conceptually, in Object-Oriented terminology, abstract supertypes for the various types in the Working Form. The two are quite similar, both in concept and in implementation: each is implemented as a header block with a generic pointer to a "definition." When a concrete subtype (`Type`, `Statement`, etc.) is instantiated, this pointer points at a `struct` of the appropriate type. In addition to this definition field, these two abstract types share three other attributes: a class indicator (which takes on values `SYMBOL_REFERENCE`, `SYMBOL_ENTITY`, `CONSTRUCT_EXPRESSION`, `...`), a reference count, and a line number (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 namable objects are represented as `Symbols`. These include `Algorithm`, `Constant`, `Entity`, `Schema`, `Type`, and `Variable`. Other abstractions (`Case_Item`, `Expression`, `Loop_Control`, `Scope`, and `Statement`) are represented as `Constructs`. Each of these abstractions then defines a `struct <name>`, which contains the components of that abstraction. Instances of these `structs` are pointed at by the definition fields of the `Symbol` and `Construct` headers.

Although the specifications for the `Symbol` and `Construct` modules are included in this document for completeness, these calls should not normally be needed by application programmers. In particular, the structures which are returned by `SYMBOLget_definition()` are not public, so that this call is not of use outside of the various Working Form module definitions.

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: All of the Working Form types are implemented as either `Symbol` or `Construct`, which *are* defined when the `struct` is compiled; the choice of this supertype can actually be viewed as a part of the specification of the abstraction.

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. `foo.c` resides in `~pdes/src/express/`; `foo.h` in `~pdes/include/`.

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).

Every abstraction defines a constant `FOO_NULL`, which represents an empty or missing value of the type. In addition, there are several operations which are defined for every type; these are primarily general management operations. Each abstraction defines at least one creation function, e.g. `FOOcreate()`. The parameters to this creation function vary, depending on the abstraction. A permanent copy of an object (as opposed to a temporary copy which will immediately be read and discarded) can be obtained by calling `FOOcopy(foo)`. This helps the system keep track of references to an object, ensuring that it is not prematurely garbage-collected. Similarly, when an object or a copy is no longer needed, it should be released by calling `FOOfree(foo)`, allowing it to be garbage-collected if appropriate.

For each abstraction, there is a function `FOOis_foo(obj)` which returns `true` if and only if its argument is a `Foo`. This is useful when dealing with a heterogeneous list, for example. 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` contains 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 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 this memory often is not released until an application exits. Hooks for doing memory management do exist (e.g., `XXXfree()` and reference counts), but currently are largely ignored.

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. The Fed-X-QDES output module (which can be found in `~pdes/src/fedex_qdes/output_smalltalk.c`) has been annotated for this purpose, although the traversal logic has become somewhat convoluted, due to peculiarities of Smalltalk-80™.

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 layed 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: 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 previous authors' coding style(s), feel free to muck around!

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 Output Module Linkage Mechanisms

One of the powers of Fed-X is the flexibility which it gives a user with regard to generating output. An important component of this flexibility on BSD UNIX systems is the dynamic loading of output modules. Both static and dynamic binding of output modules are supported by Fed-X. This is implemented by physically breaking the object code from the Working Form manager (`express.c`) into three separate `.o` files: the initialization code and the first two passes of Fed-X are compiled into `express.o`, which is stored in `libexpress.a`. The static linking version of the third pass (without any output module) is compiled into `express_static.o`; and the dynamic loading version into `express_dynamic.o`. Sources for all of these components reside in `express.c`; the various sections are extracted via conditional compilation: This file is compiled with the preprocessor symbols `reports` and `static_reports` defined to produce `express_static.o`. To produce `express_dynamic.o`, it is compiled with `reports` and `dynamic_reports` defined; and these symbols are all left undefined to produce `express.o`.

Since `express_static.o` and `express_dynamic.o` both define the function `EXPRESSpass_3()`, only one can be linked into any given executable. This selection is what determines whether a Fed-X translator links in output modules statically or dynamically. Note that a suitable output module (`.o` file) must appear *after* `express_static.o` in the linker's argument list when a statically linked translator is being built. For more information on how to build a report generator into a Fed-X translator, see [Clark90c].

5 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. The error codes are manipulated by the Error module [Clark90d]. 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.

5.1 Working Form Manager

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). When this function is compiled with `-Ddynamic_reports`, it will repeatedly prompt for report generators and output files, dynamically loading and executing them. In this case, the file parameter is ignored. When it is compiled with `-Dstatic_reports`, a report generator must also be included at link time, with the entry point `print_file(Express, FILE*)`.

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

5.2 Algorithm

Type: Algorithm_Class
Description: This type is an enumeration of `ALG_FUNCTION`, `ALG_PROCEDURE`, and `ALG_RULE`.

Procedure: ALGcreate
Parameters: Algorithm_Class class - class of algorithm to create
Returns: Algorithm - the algorithm created
Description: Create an algorithm of the indicated class. The return type of the algorithm (if applicable) is given a default value of `TY_LOGICAL`; all other attributes of the algorithm are initially undefined (appropriate NULL values).

Procedure: ALGcreate_from
Parameters: Symbol algorithm - template symbol to create from
Algorithm_Class class - class of algorithm to create
Returns: Algorithm - the algorithm created
Description: Create an algorithm of the indicated class, using an existing symbol as a template. The return type of the algorithm (if applicable) is given a default value of `TY_LOGICAL`, and the symbol's name is retained. All other attributes of the algorithm are initially undefined (appropriate NULL values). This call is used in Fed-X's parser to fill out generic symbols returned by the lexical analyzer. The template `Symbol` is modified by this call.

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

Procedure: ALGget_class
Parameters: Algorithm algorithm - algorithm to examine
Returns: Algorithm_Class - the class of the algorithm

Procedure: ALGget_code
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

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: ALGget_resolved
Parameters: Algorithm algorithm - algorithm to examine
Returns: Boolean - has algorithm been resolved?
Description: Checks whether symbol references within an algorithm have been resolved (see `ALGresolve()`)

Procedure: ALGget_return_type
Parameters: Algorithm algorithm - algorithm to examine
Returns: Type - algorithm's return type
Requires: ALGget_class(algorithm) != ALG_PROCEDURE

Procedure: ALGget_scope
Parameters: Algorithm algorithm - algorithm to examine
Returns: Scope - algorithm's local scope

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: ALGput_code
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: ALGput_resolved
Parameters: Algorithm algorithm - algorithm to modify
Returns: void
Description: Set the 'resolved' flag for an algorithm. This normally should only be called by `ALGresolve()`, which actually resolves the algorithm.

Procedure: ALGput_return_type
Parameters: Algorithm algorithm - algorithm to modify
Type type - the algorithm's return type
Returns: void
Requires: ALGget_class(algorithm) == ALG_FUNCTION
Description: Set the return type of a function. Note that procedures have no return type, and that the return type of a rule must be `TY_LOGICAL`, which is the default.

Procedure: ALGput_scope
Parameters: Algorithm algorithm - algorithm to modify
Scope scope - new local scope for algorithm
Returns: void
Description: Set the local scope of an algorithm. This scope will include declarations of the algorithm's formal parameters as well as any local 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()`.

5.3 Case Item

Procedure: CASE_ITcreate
Parameters: Linked_List of Expression labels - list of case labels
Statement statement - statement associated with this branch
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_ITfree
Parameters: Case_Item item - case item to free
Returns: void
Description: Release a case item. Indicates that the item is no longer used by the caller; if there are no other references to the item, all storage associated with it may be released.

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_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()`.

5.4 Constant

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: CSTcreate_from
Parameters: Symbol constant - template symbol to create from
Type type - type of new constant
Generic value - value for new constant
Returns: Constant - the constant created
Description: Create a new constant, using an existing symbol as a template. The name of the template symbol is retained. This call is used in Fed-X's parser to fill out generic symbols returned by the lexical analyzer. The template `Symbol` is modified by this call.

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

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: CSTget_name
Parameters: Constant constant - constant to examine
Returns: String - the name of the constant

Procedure: CSTget_type
Parameters: Constant constant - constant to examine
Returns: Type - the type of the constant

Procedure: CSTget_value
Parameters: Constant constant - constant to examine
Returns: Generic - the value of the constant

5.5 Construct

Type: Construct_Class
Description: This type is an enumeration of `CONSTR_ANY`, `CONSTR_CASE_ITEM`, `CONSTR_EXPRESSION`, `CONSTR_LOOP_CONTROL`, `CONSTR_SCOPE`, or `CONSTR_STATEMENT`.

Procedure: CONSTRcopy
Parameters: Construct construct - construct to copy
Returns: Construct - copy of construct
Description: Create a copy of a construct. This copy is a shallow copy, meaning that future changes to the original will be reflected in the copy.

Procedure: CONSTRcreate
Parameters: Construct_Class class - class of construct to create
Returns: Construct - newly created construct
Description: Create a new construct. The new construct's definition field is NULL. `CONSTRcreate()` is normally called by one of the client create functions, e.g. `EXPcreate()`, which then fills in the definition field.

Procedure: CONSTRdestroy
Parameters: Construct construct - construct to destroy
Returns: void
Description: Release a construct. Indicates that the construct is no longer used by the caller; if there are no other references to the construct, all storage associated with it may be released.

Procedure: CONSTRget_class
Parameters: Construct construct - construct to examine
Returns: Construct_Class - class of construct

Procedure: CONSTRget_definition
Parameters: Construct construct - construct to examine
Returns: Generic - definition of construct

Procedure: CONSTRis_kind_of
Parameters: Construct construct - construct to test
Construct_Class kind - kind of construct to test for
Returns: Boolean - is this construct of the given class?

Procedure: CONSTRput_definition
Parameters: Construct construct - construct to define
Generic definition - definition of construct
Returns: void
Description: Store into the definition of a construct.

5.6 Entity

Procedure: ENTITYadd_attribute
Parameters: Entity entity - entity to modify
Variable attribute - attribute to add
Returns: void

Procedure: ENTITYadd_instance
Parameters: Entity entity - entity to modify
Generic instance - new instance
Returns: void

Procedure: ENTITYcreate
Parameters: String name - name of new entity
Returns: Entity - the entity created
Description: Create a new entity. The entity has a name, and is otherwise empty.

Procedure: ENTITYcreate_from
Parameters: Symbol entity - symbol to create from
Returns: Entity - the entity created
Description: Create a new entity, using an existing symbol as a template. The name of the template symbol is retained. This call is used in Fed-X's parser to fill out generic symbols returned by the lexical analyzer. The template `Symbol` is modified by this call.

Procedure: ENTITYdelete_instance
Parameters: Entity entity - entity to modify
Generic instance - instance to delete
Returns: void

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

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 [Altemeuller88]. 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

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_resolved
Parameters: Entity entity - entity to examine
Returns: Boolean - has entity been resolved?
Description: Checks whether symbol references within an entity definition have been resolved.

Procedure: ENTITYget_scope
Parameters: Entity entity - entity to examine
Returns: Scope - local scope of this entity

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_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. This list should not be `LISTfree`'d. The issue, which arises in Express, of a boolean expression specifying the subtypes, currently is not dealt with.

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_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: 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_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 children of
Entity child - child to check for
Returns: Boolean - is child a direct subtype of 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: 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_name
Parameters: Entity entity - entity to modify
String name - entity's name
Returns: void
Description: Set the name of an entity.

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_resolved
Parameters: Entity entity - entity to modify
Returns: void
Description: Set the 'resolved' flag for an entity. This normally should only be called by `ENTITYresolve()`, which actually resolves the entity definition.

Procedure: ENTITYput_scope
Parameters: Entity entity - entity to modify
Scope scope - entity's local scope
Returns: void
Description: Set the local scope of an entity. This will contain definitions of the entity's locally-defined attributes.

Procedure: ENTITYput_subtypes
Parameters: Entity entity - entity to modify
Linked_List list - subclasses
Returns: void
Description: Set the (immediate) subtype list of an entity. The elements of the list should be `Entitys` or (unresolved) `Symbols`. The issue, which arises in Express, of a boolean expression specifying the subtypes, is not dealt with here.

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()`.

5.7 Expression

Constant: LITERAL_EMPTY_SET - a generic set literal representing the empty set
Constant: LITERAL_INFINITY - a numeric literal representing infinity
Constant: LITERAL_PI - a real literal with the value 3.1415...
Constant: LITERAL_ZERO - an integer literal with the value 0

Type: Expression_Class
Description: This type is an enumeration of `EXP_IDENT`, `EXP_LITERAL`, `EXP_OPERATION`, `EXP_FUNCTION`, and `EXP_FIELD`.

Procedure: EXPcreate
Parameters: Expression_Class class - class of expression to create
Returns: Expression - the expression created
Description: Create and return a new expression of the indicated class. The type of the new expression is initially `TY_INTEGER`. Other attributes are initially undefined.

Procedure: EXPcreate_binary
Parameters: Op_Code op - operation
Expression op1 - first operand
Expression op2 - second operand
Error* errc - buffer for error code
Returns: Expression - the expression created
Description: Create a binary operation expression.
Errors: `ERROR_wrong_operand_count` - requested operation is not binary

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

Procedure: EXPget_algorithm
Parameters: Expression expression - expression to examine
Returns: Algorithm - the algorithm called in the expression
Requires: EXPget_class(expression) == EXP_FUNCTION

Procedure: EXPget_algorithm_parameters
Parameters: Expression expression - expression to examine
Returns: Linked_List of Expression - list of actual parameters
Requires: EXPget_class(expression) == EXP_FUNCTION
Description: Retrieve the actual parameter `Expressions` from a function call expression. This list should not be `LISTfree`'d.

Procedure: EXPget_class
Parameters: Expression expression - expression to examine
Returns: Expression_Class - the class of the expression

Procedure: EXPget_field
Parameters: Expression expression - expression to examine
Returns: Symbol - field extracted by expression
Requires: EXPget_class(expression) == EXP_FIELD
Description: Retrieve the name of the field from a field (attribute) extraction expression. The value returned ought to be a `Variable`, but scoping for attribute references is not yet handled, and so the reference cannot be resolved to a variable.

Procedure: EXPget_first_operand
Parameters: Expression expression - expression to examine
Returns: Expression - the first (left-hand) operand
Requires: EXPget_class(expression) == EXP_OPERATION

Procedure: EXPget_identifier
Parameters: Expression expression - expression to examine
Returns: Symbol - the identifier referenced in the expression
Requires: EXPget_class(expresion) == EXP_IDENT

Procedure: EXPget_integer_literal
Parameters: Expression expression - integer literal to examine
Error* errc - buffer for error code
Returns: Integer - the literal's value
Requires: EXPget_class(expression) == EXP_LITERAL
Errors: `ERROR_integer_literal_expected`

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_logical_literal
Parameters: Expression expression - logical literal to examine
Error* errc - buffer for error code
Returns: Boolean - the literal's value
Requires: EXPget_class(expression) == EXP_LITERAL
Errors: `ERROR_logical_literal_expected`

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

Procedure: EXPget_operator
Parameters: Expression expression - expression to examine
Returns: Op_Code - the operator invoked by the expression
Requires: EXPget_class(expression) == EXP_OPERATION

Procedure: EXPget_real_literal
Parameters: Expression expression - real literal to examine
Error* errc - buffer for error code
Returns: Real - the literal's value
Requires: EXPget_class(expression) == EXP_LITERAL
Errors: `ERROR_real_literal_expected`

Procedure: EXPget_second_operand
Parameters: Expression expression - expression to examine
Error* errc - buffer for error code
Returns: Expression - the expression's second operand
Requires: EXPget_class(expression) == EXP_OPERATION
Errors: `ERROR_wrong_operand_count` - expression is not a binary operation

Procedure: EXPget_set_literal
Parameters: Expression expression - set literal to examine
Error* errc - buffer for error code
Returns: Linked_List of Generic - the literal's contents
Requires: EXPget_class(expression) == EXP_LITERAL
Description: Retrieve the value of a set literal, as a list.
Errors: `ERROR_set_literal_expected`

Procedure: EXPget_string_literal
Parameters: Expression expression - string literal to examine
Error* errc - buffer for error code
Returns: String - the literal's value
Requires: EXPget_class(expression) == EXP_LITERAL
Errors: `ERROR_string_literal_expected`

Procedure: EXPget_structure
Parameters: Expression expression - expression to examine
Returns: Expression - structure referenced by expression
Requires: EXPget_class(expression) == EXP_FIELD
Description: Retrieves the structure examined by a field extraction expression. This is the expression which computes the entity instance from which a field is to be extracted.

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: EXPput_algorithm
Parameters: Expression expression - expression to modify
Algorithm algorithm - function called by expression
Returns: void
Requires: EXPget_class(expression) == EXP_FUNCTION
ALGget_class(algorithm) == ALG_FUNCTION | ALG_RULE
Description: Set the algorithm called by a function call expression.

Procedure: EXPput_algorithm_parameters
Parameters: Expression expression - expression to modify
Linked_List parameters - list of actual parameters
Returns: void
Requires: EXPget_class(expression) == EXP_FUNCTION
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: EXPput_field
Parameters: Expression expression - expression to modify
Symbol field - field extracted by expression
Returns: void
Requires: EXPget_class(expression) == EXP_FIELD
Description: Set the field in a field extraction expression.

Procedure: EXPput_identifier
Parameters: Expression expression - expression to modify
Symbol identifier - the referent of the identifier
Returns: void
Requires: EXPget_class(expression) == EXP_IDENT
Description: Set the referent of an identifier expression.

Procedure: EXPput_integer_literal
Parameters: Expression expression - literal to modify
Integer value - the value for the literal
Returns: void
Requires: EXPget_class(expression) == EXP_LITERAL
Description: Set the type and value of an integer literal.

Procedure: EXPput_logical_literal
Parameters: Expression expression - literal to modify
Boolean value - the value for the literal
Returns: void
Requires: EXPget_class(expression) == EXP_LITERAL
Description: Set the type and value of a logical literal.

Procedure: EXPput_operand
Parameters: Expression expression - expression to modify
Expression operand - the single operand to the expression
Error* errc - buffer for error code
Returns: void
Requires: EXPget_class(expression) == EXP_OPERATION
Description: Set the single operand of a unary operation expression.
Errors: `ERROR_wrong_operand_count` - expression is not a unary operation

Procedure: EXPput_operands
Parameters: Expression expression - expression to modify
Expression operand1 - the first operand to the expression
Expression operand2 - the second operand to the expression
Error* errc - buffer for error code
Returns: void
Requires: EXPget_class(expression) == EXP_OPERATION
Description: Set the two operands to a binary operation expression.
Errors: `ERROR_wrong_operand_count` - expression is not a binary operation

Procedure: EXPput_operator
Parameters: Expression expression - expression to modify
Op_Code operation - the operation invoked by the expression
Returns: void
Requires: EXPget_class(expression) == EXP_OPERATION
Description: Set the operator of an operation expression.

Procedure: EXPput_real_literal
Parameters: Expression expression - literal to modify
Real value - the value for the literal
Returns: void
Requires: EXPget_class(expression) == EXP_LITERAL
Description: Set the type and value of a real literal.

Procedure: EXPput_set_literal
Parameters: Expression expression - literal to modify
Linked_List value - contents of the set literal
Returns: void
Requires: EXPget_class(expression) == EXP_LITERAL
Description: Set the type and value of a set literal (from a list of `Generic` elements).

Procedure: EXPput_string_literal
Parameters: Expression expression - literal to modify
String value - the value for the literal
Returns: void
Requires: EXPget_class(expression) == EXP_LITERAL
Description: Set the type and value of a string literal.

Procedure: EXPput_structure
Parameters: Expression expression - expression to modify
Expression structure - structure referenced by expression
Returns: void
Requires: EXPget_class(expression) == EXP_FIELD
Description: Set the structure examined by a field extraction expression. This is the expression which computes the entity instance from which a field is to be extracted.

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 and 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()`.

5.8 Loop Control

Type: Loop_Control_Class
Description: This type is an enumeration of `LOOP_INCREMENT`, `LOOP_SET_SCAN`, `LOOP_UNTIL`, and `LOOP_WHILE`.

Procedure: LOOP_CTLcreate_increment
Parameters: Expression control - controlling expression
Expression start - initial value
Expression end - terminal value
Expression increment - amount by which to increment
Returns: Loop_Control - loop control created

Procedure: LOOP_CTLcreate_set_scan
Parameters: Expression control - controlling expression
Expression set - set to scan over
Error* errc - buffer for error code
Returns: Loop_Control - the loop control created
Requires: TYPEget_class(EXPget_type(set)) == TYPE_SET
Description: Create a set scan control over the indicated set. Set scan controls are eliminated by Tokyo Express, but still appear in the Tokyo IPIM. This call may disappear at any time.
Errors: `ERROR_set_scan_set_expected` - scan control is not a set

Procedure: LOOP_CTLcreate_until
Parameters: Expression control - termination condition
Error* errc - buffer for error code
Returns: Loop_Control - the loop control created
Requires: EXPget_type(control) == TY_LOGICAL
Errors: `ERROR_control_boolean_expected` - controlling expression is not boolean

Procedure: LOOP_CTLcreate_while
Parameters: Expression control - continuation condition
Error* errc - buffer for error code
Returns: Loop_Control - the loop control created
Requires: EXPget_type(control) == TY_LOGICAL
Errors: `ERROR_control_boolean_expected` - controlling expression is not boolean

Procedure: LOOP_CTLfree
Parameters: Loop_Control control - control to free
Returns: void
Description: Release a loop control. Indicates that the control is no longer used by the caller; if there are no other references to the control, all storage associated with it may be released.

Procedure: LOOP_CTLget_control_class
Parameters: Loop_Control control - loop control to examine
Returns: Loop_Control_Class - the loop control's class

Procedure: LOOP_CTLget_control_set
Parameters: Loop_Control control - loop control to examine
Returns: Expression - set scanned over by the control
Requires: LOOP_CTLget_control_class(control) == LOOP_SET_SCAN

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_CTLget_final
Parameters: Loop_Control control - loop control to examine
Returns: Expression - terminal value for controlling expression
Requires: LOOP_CTLget_control_class(control) == LOOP_INCREMENT
Description: Retrieve the final value from an increment control.

Procedure: LOOP_CTLget_increment
Parameters: Loop_Control control - loop control to examine
Returns: Expression - amount to increment by on each iteration
Requires: LOOP_CTLget_control_class(control) == LOOP_INCREMENT
Description: Retrieve the increment expression from an increment control.

Procedure: LOOP_CTLget_start
Parameters: Loop_Control control - loop control to examine
Returns: Expression - initial expression for controlling expression
Requires: LOOP_CTLget_control_class(control) == LOOP_INCREMENT
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()`.

5.9 Schema

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

Procedure: SCHEMAcreate_from
Parameters: Symbol schema - symbol to build from
Scope scope - local scope for schema
Returns: Schema - the schema created
Description: Create a new schema, using an existing symbol as a template. The template symbol's name is retained. This call is used in Fed-X's parser to fill out generic symbols returned by the lexical analyzer. The template `Symbol` is modified by this call.

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: SCHEMAget_scope
Parameters: Schema schema - schema to examine
Returns: Scope - schema's local scope

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

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
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()`.

5.10 Scope

Procedure: SCOPEadd_import
Parameters: Scope scope - scope to modify
Symbol schema - schema to import (assume)
Returns: void
Description: Add a schema to the import list of a scope. If the symbol given has not been resolved to a schema, `SCOPEresolve()` will see to it that it is.

Procedure: SCOPEadd_private
Parameters: Scope scope - scope to modify
Symbol name - item to add to private list
Returns: void
Description: Add an item to a scope's list of private declarations. Note that after `SCOPEput_everything_private()` is called, the items added to the private list are actually the ones which are public.

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. There are several aliases for this procedure, which can be used when the class of the symbol being defined is known: `SCOPEdefine_algorithm()`, `SCOPEdefine_constant()`, `SCOPEdefine_entity()`, `SCOPEdefine_schema()`, `SCOPEdefine_type()`, and `SCOPEdefine_type()`
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: SCOPEfree
Parameters: Scope scope - scope to free
Returns: void
Description: Release a scope. Indicates that the scope is no longer used by the caller; if there are no other references to the scope, all storage associated with it may be released.

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 not be `LISTfree`'d.

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

Procedure: SCOPEget_superscope
Parameters: Scope scope - scope to examine
Returns: Scope - next outer (containing) scope
Description: Retrieve 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_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
Symbol_Class sections - section(s) in which to look
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. This is the basic lookup function for scopes, and normally is not called from outside the scope module. It is the heart of the six lookup functions which follow. Two or more `Symbol_Classes` can be or'ed together to form the sections parameter. Note that `SYMBOL_ANY` is the result of or'ing together all of the known symbol classes. If the scope does not define the name, the parent scopes are successively queried. If no definition is found, `SYMBOL_NULL` is returned. If an inappropriate definition is found first, it is returned.
Errors: `ERROR_inappropriate_use` - the first definition found is not of the requested class
`ERROR_undefined_identifier` - no definition was found

Procedure: SCOPElookup_algorithm
Parameters: Scope scope - scope in which to look up name
String name - name to look up
Error* errc - buffer for error code
Returns: Algorithm - definition of name as an algorithm in the scope

Procedure: SCOPElookup_constant
Parameters: Scope scope - scope in which to look up name
String name - name to look up
Error* errc - buffer for error code
Returns: Constant - definition of name as a constant in the scope

Procedure: SCOPElookup_entity
Parameters: Scope scope - scope in which to look up name
String name - name to look up
Error* errc - buffer for error code
Returns: Entity - definition of name as an entity in the scope

Procedure: SCOPElookup_schema
Parameters: Scope scope - scope in which to look up name
String name - name to look up
Error* errc - buffer for error code
Returns: Schema - definition of name as a schema in the scope

Procedure: SCOPElookup_type
Parameters: Scope scope - scope in which to look up name
String name - name to look up
Error* errc - buffer for error code
Returns: Type - definition of name as a type in the scope

Procedure: SCOPElookup_variable
Parameters: Scope scope - scope in which to look up name
String name - name to look up
Error* errc - buffer for error code
Returns: Variable - definition of name as a variable in the scope

Procedure: SCOPEput_everything_private
Parameters: Scope scope - scope to modify
Boolean flag - are declarations private by default?
Returns: void
Description: Indicate whether declarations are private or exported by default. In Express, any declaration is available to any scope which imports the scope in which it appears, unless the declaration is explicitly marked 'private'. This is the default behavior for the `Scope` abstraction. If this flag is set, however, a declaration is kept private by default, unless it appears on the 'private' list. The meaning of the 'private' list is thus reversed. This is to allow the Express `PRIVATE EVERYTHING [EXCEPT ...]` directives to be handled conveniently.

Procedure: SCOPEput_imports
Parameters: Scope scope - scope to modify
Linked_List imports - list of schemata to assume
Returns: void
Description: Set the entire list of assumed schemata in one fell swoop.

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
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()`.

5.11 Statement

Type: Statement_Class
Description: This type is an enumeration of `STMT_ASSIGNMENT`, `STMT_CASE`, `STMT_COMPOUND`, `STMT_IF`, `STMT_PROCEDURE`, `STMT_REPEAT`, `STMT_RETURN`, `STMT_SIMPLE`, and `STMT_WITH`.

Type: Statement_Simple
Description: This type is an enumeration of `STATEMENT_ESCAPE` and `STATEMENT_SKIP`.

Procedure: STMTcreate_assignment
Parameters: Expression lhs - the left-hand-side of the assignment
Expression rhs - the right-hand-side of the assignment
Returns: Statement - the assignment statement created
Description: Create an assignment statement.

Procedure: STMTcreate_case
Parameters: Expression selector - expression to case on
Linked_List case - list of case branches
Returns: Statement - the case statement created
Description: Create a case statement. The elements of the case branch list should be `Case_Items`.

Procedure: STMTcreate_compound
Parameters: Linked_List statements - list of compound statement elements
Returns: 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: STMTcreate_if
Parameters: Expression test - the condition for the if
Statement then - code executed when test == true
Statement otherwise - code executed when test == false
Returns: Statement - 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: STMTcreate_procedure_call
Parameters: Algorithm algorithm - procedure called by statement
Linked_List parameters - list of actual parameters
Returns: Statement - the procedure call created
Requires: ALGget_algorithm_class(Algorithm) == ALG_PROCEDURE
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: STMTcreate_repeat
Parameters: Linked_List controls - list of controls for the loop
Statement body - statement to be repeated
Returns: Statement - the repeat statement created
Description: Create a repeat statement. The elements of the controls list should be `Loop_Controls`.

Procedure: STMTcreate_return
Parameters: Expression expression - expression to compute return value
Returns: Statement - the return statement created
Description: Create a return statement.

Procedure: STMTcreate_simple
Parameters: Statement_Simple simple - type of simple statement
Returns: Statement - the simple statement created
Description: Create a simple statement. A simple statement is a statement which consists of a single keyword. In Express, the two examples are 'escape' and 'skip'.

Procedure: STMTcreate_with
Parameters: Expression expression - controlling expression for the with
Statement body - controlled statement for the with
Returns: Statement - the with statement created
Description: Create a with statement.

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

Procedure: STMTget_assignment_lhs
Parameters: Statement statement - statement to examine
Returns: Expression - left-hand-side of assignment statement
Requires: STMTget_class(statement) == STMT_ASSIGNMENT

Procedure: STMTget_assignment_rhs
Parameters: Statement statement - statement to examine
Returns: Expression - right-hand-side of assignment statement
Requires: STMTget_class(statement) == STMT_ASSIGNMENT

Procedure: STMTget_case_items
Parameters: Statement statement - statement to examine
Returns: Linked_List - case branches
Requires: STMTget_class(statement) == STMT_CASE
Description: Retrieve a list of the branches in a case statement. The elements of this list are `Case_Items`.

Procedure: STMTget_case_selector
Parameters: Statement statement - statement to examine
Returns: Expression - the selector for the case statment
Requires: STMTget_class(statement) == STMT_CASE
Description: Retrieve the selector from a case statement. This is the expression whose value is compared to each case label in turn.

Procedure: STMTget_class
Parameters: Statement statement - statement to examine
Returns: Statement_Class - the class of the statement

Procedure: STMTget_compound_items
Parameters: Statement statement - statement to examine
Returns: Linked_List - list of statements in compound
Requires: STMTget_class(statement) == STMT_COMPOUND
Description: Retrieve a list of the `Statements` comprising a compound statement.

Procedure: STMTget_else_clause
Parameters: Statement statement - statement to examine
Returns: Statement - code for 'else' branch
Requires: STMTget_class(statement) == STMT_IF

Procedure: STMTget_if_condition
Parameters: Statement statement - statement to examine
Returns: Expression - the test condition
Requires: STMTget_class(statement) == STMT_IF

Procedure: STMTget_procedure
Parameters: Statement statement - statement to examine
Returns: Algorithm - algorithm called by this statement
Requires: STMTget_class(statement) == STMT_PROCEDURE
Description: Retrieve the algorithm called by a procedure call statement.

Procedure: STMTget_procedure_parameters
Parameters: Statement statement - statement to examine
Returns: Linked_List - actual parameters to this call
Requires: STMTget_class(statement) == STMT_PROCEDURE
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: STMTget_repeat_body
Parameters: Statement statement - statement to examine
Returns: Statement - the body of the loop
Requires: STMTget_class(statement) == STMT_REPEAT
Description: Retrieve the body (repeated portion) of a repeat statement.

Procedure: STMTget_repeat_controls
Parameters: Statement statement - statement to examine
Returns: Linked_List - list of loop controls
Requires: STMTget_class(statement) == STMT_REPEAT
Description: Retrieve a list of a repeat statement's controls. The elements of this list are `Loop_Controls`.

Procedure: STMTget_return_expression
Parameters: Statement statement - statement to examine
Returns: Expression - expression returned by this statement
Requires: STMTget_class(statement) == STMT_RETURN
Description: Retrieve the expression whose value is computed and returned by a return statement.

Procedure: STMTget_simple_name
Parameters: Statement statement - statement to examine
Returns: Statement_Simple - the name of this simple statement
Requires: STMTget_class(statement) == STMT_SIMPLE

Procedure: STMTget_then_clause
Parameters: Statement statement - statement to examine
Returns: Statement - code for 'then' branch
Requires: STMTget_class(statement) == STMT_IF

Procedure: STMTget_with_body
Parameters: Statement statement - statement to examine
Returns: Statement - statement forming the body of the with statement
Requires: STMTget_class(statement) == STMT_WITH

Procedure: STMTget_with_control
Parameters: Statement statement - statement to examine
Returns: Expression - the controlling expression
Requires: STMTget_class(statement) == STMT_WITH
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.

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: STMTput_procedure
Parameters: Statement statement - statement to modify
Algorithm procedure - definition of called algorithm
Returns: void
Requires: STMTget_class(statement) == STMT_PROCEDURE
Description: Set the actual algorithm 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: 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()`.

5.12 Symbol

Type: Symbol_Class
Description: This type is an enumeration of `SYMBOL_ANY`, `SYMBOL_REFERENCE`, `SYMBOL_ALGORITHM`, `SYMBOL_CONSTANT`, `SYMBOL_ENTITY`, `SYMBOL_SCHEMA`, `SYMBOL_TYPE`, `SYMBOL_VARIABLE`, and `SYMBOL_OBJECT`. `SYMBOL_ANY` is the bitwise-or of all other values of `Symbol_Class`, and is useful in `SCOPElookup()`. `SYMBOL_REFERENCE` indicates a symbol reference which has not yet been resolved. `SYMBOL_OBJECT` is used by the STEP Working Form.

Procedure: SYMBOLbecome
Parameters: Symbol old - symbol to replace definition of
Symbol new - symbol to replace with
Returns: void
Requires: old != SYMBOL_NULL
new != SYMBOL_NULL
Description: Replace a symbol with a new symbol. All references to the old symbol will now refer to the new symbol. This call is used by the various `XXXresolve()` routines when an initial interpretation of some symbol turns out to be wrong.

Procedure: SYMBOLcopy
Parameters: Symbol symbol - symbol to copy
Returns: Symbol - copy of symbol
Description: Create a copy of a symbol. This copy is a shallow copy, meaning that future changes to the original will be reflected in the copy.

Procedure: SYMBOLcreate
Parameters: Symbol_Class class - class of symbol to create
Returns: Symbol - newly created symbol
Description: Create a new symbol. The new symbol's definition field is NULL. `SYMBOLcreate()` is normally called by one of the client create functions, e.g. `ALGcreate()`, which then fills in the definition field.

Procedure: SYMBOLdeep_copy
Parameters: Symbol symbol - symbol to copy
Returns: Symbol - copy of symbol
Description: Create a deep copy of a symbol. This call copies the symbol header, so that multiple headers (thus with different names) can point to the same definition. This clearly causes problems with memory management, but is needed in order to deal with declarations like `TYPE foo = bar`.

Procedure: SYMBOLequal
Parameters: Symbol sym1 - first symbol to test
Symbol sym2 - second symbol to test
Returns: Boolean - are the symbols equal?
Description: Test two symbols for equality. Two symbols are equal if they are the same symbol or if they share the same definition (in Lisp terminology, if the headers are eq or the definitions are eq).

Procedure: SYMBOLfree
Parameters: Symbol symbol - symbol to free
void (*func)(Generic) - function to destroy symbol definition
Returns: void
Description: Free a reference to a symbol. If there are no more references to the symbol, its definition is passed to the given destructor function before the symbol header is `free`'d. The usual destruction paradigm for a symbol client is to have a function `FOOfree(Foo foo)` which calls `SYMBOLfree(foo, FOOdestroy)`, where module `Foo` includes a static function `FOOdestroy(struct Foo*)`. Of course, in a truly object-oriented environment, this garbage would be unnecessary!

Procedure: SYMBOLget_class
Parameters: Symbol symbol - symbol to examine
Returns: Symbol_Class - class of symbol

Procedure: SYMBOLget_definition
Parameters: Symbol symbol - symbol to examine
Returns: Generic - definition of symbol
Description: Retrieve a symbol's definition field. This will need to be cast to the appropriate pointer type, according to the class of the symbol.

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.

LProcedure: SYMBOLis_kind_of
Parameters: Symbol symbol - symbol to test
Symbol_Class kind - kind of symbol to test for
Returns: Boolean - is this symbol of the given class?

Procedure: SYMBOLput_class
Parameters: Symbol symbol - symbol to modify
Symbol_Class class - class for symbol
Returns: void
Description: Set a symbol's class.

Procedure: SYMBOLput_definition
Parameters: Symbol symbol - symbol to define
Generic definition - definition of symbol
Returns: void
Description: Store into the definition field of a symbol.

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.

5.13 Type

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

Constant: TY_GENERIC
Description: The simple type 'generic.'

Constant: TY_INTEGER
Description: Integer type with default precision.

Constant: TY_LOGICAL
Description: Logical type.

Constant: TY_NUMBER
Description: Number type.

Constant: TY_REAL
Description: Real type with default precision.

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

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

Type: Type_Class
Description: This type is an enumeration of `TYPE_AGGREGATE`, `TYPE_ARRAY`, `TYPE_BAG`, `TYPE_ENTITY`, `TYPE_ENUM`, `TYPE_GENERIC`, `TYPE_INTEGER`, `TYPE_LIST`, `TYPE_LOGICAL`, `TYPE_NUMBER`, `TYPE_REAL`, `TYPE_SELECT`, `TYPE_SET`, and `TYPE_STRING`.

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: TYPEcreate
Parameters: Type_Class class - the class of type to create
Returns: Type - the type created
Description: Create a new type. The type's class is as specified; all other fields have appropriate NULL values.

Procedure: TYPEcreate_from
Parameters: Symbol type - template symbol to fill in for type
Type_Class class - the class of type to create
Returns: Type - the type created
Description: Create a new type of the indicated class, using an existing symbol as a template. The template symbol's name is retained. All other attributes of the type have appropriate NULL values. This call is used in Fed-X's parser to fill out generic symbols returned by the lexical analyzer. The template `Symbol` is modified by this call.

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

Procedure: TYPEget_aggregate_optional
Parameters: Type type - type to examine
Returns: Boolean - are elements of this aggregate optional?
Requires: TYPEget_class(type) == TYPE_ARRAY
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: TYPEget_aggregate_unique
Parameters: Type type - type to examine
Returns: Boolean - must elements of this aggregate be unique?
Requires: TYPEget_class(type) == TYPE_ARRAY | TYPE_LIST
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: TYPEget_base_type
Parameters: Type type - type to examine
Returns: Type - the base type of the aggregate type
Requires: TYPEget_class(type) == TYPE_AGGREGATE | TYPE_ARRAY | TYPE_BAG | TYPE_LIST | TYPE_SET
Description: Retrieve the base type of an aggregate. This is the type of each element of an instantiation of the type.

Procedure: TYPEget_class
Parameters: Type type - type to examine
Returns: Type_Class - the class of the type

Procedure: TYPEget_entity
Parameters: Type type - type to examine
Returns: Entity - definition of entity type
Requires: TYPEget_class(type) == TYPE_ENTITY
Description: Retrieve the entity referenced by an entity type.

Procedure: TYPEget_fields
Parameters: Type type - type to examine
Returns: Linked_List - list of selectable types
Requires: TYPEget_class(type) == TYPE_SELECT
Description: Retrieve a list of the selectable types from a select type.

Procedure: TYPEget_items
Parameters: Type type - type to examine
Returns: Linked_List - list of enumeration items
Requires: TYPEget_class(type) == TYPE_ENUM
Description: Retrieve an enumerated type's list of identifiers. Each element of this list is a `Constant`.

Procedure: TYPEget_lower_limit
Parameters: Type type - type to examine
Returns: Expression - lower limit of the aggregate type
Requires: TYPEget_class(type) == TYPE_AGGREGATE | TYPE_ARRAY | TYPE_BAG | TYPE_LIST | TYPE_SET
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: TYPEget_name
Parameters: Type type - type to examine
Returns: String - the name of the type

Procedure: TYPEget_precision
Parameters: Type type - type to examine
Returns: Expression - the precision specification of the type
Requires: TYPEget_class(type) == TYPE_INTEGER | TYPE_REAL | TYPE_STRING
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: TYPEget_resolved
Parameters: Type type - type to examine
Returns: Boolean - has type been resolved?
Description: Checks whether symbol references within a type have been resolved.

Procedure: TYPEget_size
Parameters: Type type - type to examine
Returns: Boolean - 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_upper_limit
Parameters: Type type - type to examine
Returns: Expression - upper limit of the aggregate type
Requires: TYPEget_class(type) == TYPE_AGGREGATE | TYPE_ARRAY | TYPE_BAG | TYPE_LIST | TYPE_SET
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: TYPEget_varying
Parameters: Type type - type to examine
Returns: Boolean - is the string type of varying length?
Requires: TYPEget_class(type) == TYPE_STRING
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: TYPEinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Type module. This is called by `EXPRESSinitialize()`, and so normally need not be called individually.

Procedure: TYPEput_aggregate_optional
Parameters: Type type - type to modify
Boolean optional - are array elements optional?
Returns: void
Requires: TYPEget_class(type) == TYPE_ARRAY
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: TYPEput_aggregate_unique
Parameters: Type type - type to modify
Boolean unique - are aggregate elements required to be unique?
Returns: void
Requires: TYPEget_class(type) == TYPE_ARRAY | TYPE_LIST
Description: Set the 'unique' flag for an aggregate type. This flag indicates that an instantiation of the type may not contain duplicate items.

Procedure: TYPEput_base_type
Parameters: Type type - type to modify
Type base - the base type for this aggregate
Returns: void
Requires: TYPEget_class(type) == TYPE_AGGREGATE | TYPE_ARRAY | TYPE_BAG | TYPE_LIST | TYPE_SET
Description: Set the base type of an aggregate type. This is the type of every element.

Procedure: TYPEput_entity
Parameters: Type type - type to modify
Entity entity - definition of type
Returns: void
Requires: TYPEget_class(type) == TYPE_ENTITY
Description: Set the entity referred to by an entity type.

Procedure: TYPEput_fields
Parameters: Type type - type to modify
Linked_List list - list of selectable types
Returns: void
Requires: TYPEget_class(type) == TYPE_SELECT
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: TYPEput_items
Parameters: Type type - type to modify
Linked_List list - list of enumeration items
Returns: void
Requires: TYPEget_class(type) == TYPE_ENUM
Description: Set the list of identifiers for an enumerated type. Each element of this list should be a `Constant`.

Procedure: TYPEput_limits
Parameters: Type type - type to modify
Expression lower - lower bound for aggregate
Expression upper - upper bound for aggregate
Returns: void
Requires: TYPEget_class(type) == TYPE_AGGREGATE | TYPE_ARRAY | TYPE_BAG | TYPE_LIST | TYPE_SET
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, the specify the minimum and maximum number of elements which an instance may contain.

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_precision
Parameters: Type type - type to modify
Expression prec - the precision of the type
Returns: void
Requires: TYPEget_class(type) == TYPE_INTEGER | TYPE_REAL | TYPE_STRING
Description: Set the precision of certain types. This is the maximum number of significant digits or characters in an instance.

Procedure: TYPEput_resolved
Parameters: Type type - type to modify
Returns: void
Description: Set the 'resolved' flag for a type. This normally should only be called by `TYPEresolve()`, which actually resolves the type.

Procedure: TYPEput_varying
Parameters: Type type - type to modify
Boolean varying - is string type of varying length?
Returns: void
Requires: TYPEget_class(type) == TYPE_STRING
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: TYPEresolve
Parameters: Type type - type to resolve
Scope scope - scope in which to resolve
Returns: void
Description: Resolve all references in a type definition. This is called, in due course, by `EXPRESSpass_2()`.

5.14 Variable

Type: Reference_Class
Description: This type is an enumeration of `REF_INTERNAL`, `REF_EXTERNAL`, and `REF_DYNAMIC`.

Procedure: VARcreate
Parameters: String name - name of variable to create
Type type - type of variable to create
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: VARcreate_from
Parameters: Symbol variable - symbol to create from
Type type - type of variable to create
Returns: Variable - the Variable created
Description: Create a new variable, using an existing symbol as a template. The reference class of the variable is, by default, dynamic. All special flags associated with the variable (e.g., optional) are initially false. The template symbol's name is retained. This call is used in Fed-X's parser to fill out generic symbols returned by the lexical analyzer. The `Symbol` provided is used as a template, and is modified and returned as the function value.

Procedure: VARfree
Parameters: Variable var - variable to destroy
Returns: void
Description: Release a variable. Indicates that the variable is no longer used by the caller; if there are no other references to the variable, all storage associated with it may be released.

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_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_reference_class
Parameters: Variable var - variable to examine
Returns: Reference_Class - the variable's reference class

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: 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_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_reference_class
Parameters: Variable var - variable to modify
Reference_Class ref - the variable's reference class
Returns: void
Description: Set the reference class of a variable. The reference class defaults to `REF_DYNAMIC`.

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()`.

6 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 INCLUDEd 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

A References

[Altemeuller88] Altemeuller, J., Mapping from Express to Physical File Structure, ISO TC184/SC4/WG1 Document N280, Septembner, 1988

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

[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., The NIST PDES Toolkit: Technical Fundamentals, NISTIR 4335, National Institute of Standards and Technology, Gaithersburg, MD, May 1990

[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

[Schenck89] Schenck, D., ed., Information Modeling Language Express: Language Reference Manual, ISO TC184/SC4/WG1 Document N362, May 1989

[Smith88] Smith, B., and G. Rinaudot, eds., Product Data Exchange Specification First Working Draft, NISTIR 88-4004, National Institute of Standards and Technology, Gaithersburg, MD, December 1988

NIST-114A
(REV. 3-89)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 4407

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
SEPTEMBER 1990

4. TITLE AND SUBTITLE

NIST Express Working Form Programmer's Reference

5. AUTHOR(S)

Stephen Nowland Clark

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

10. SUPPLEMENTARY NOTES

[ ] DOCUMENT DESCRIBES A COMPUTER PROGRAM; SF-185, FIPS SOFTWARE SUMMARY, IS ATTACHED.

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

The Product Data Exchange Specification (PDES) is an emerging standard for the exchange of product information among various manufacturing applications. PDES includes an information model written in the Express language; other PDES-related information models are also written in Express. The National PDES Testbed at NIST has developed software to manipulate and translate Express models. This software consists of an in-memory working form and an associated Express language parser, Fed-X. The internal operation of the Fed-X parser is described. The implementation of the data abstractions which make up the Express Working Form is discussed, and specifications are given for the Working Form access functions. The creation of Express language translators using Fed-X is discussed.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

data modeling; Express; PDES; schema translation; STEP

13. AVAILABILITY

[X] UNLIMITED

[ ] FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).

[ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.

[X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
56

15. PRICE
A04

ELECTRONIC FORM