THE NIST EXPRESS TOOLKIT – DESIGN AND IMPLEMENTATION

Don Libes

Factory Automation Systems Division
National Institute of Standards and Technology
Gaithersburg, Maryland

ABSTRACT

The NIST EXPRESS toolkit is a software library for building EXPRESS-related tools. EXPRESS is an ISO language for describing information models. EXPRESS descriptions are neutral to different data storage paradigms and systems on different hardware platforms and networks.

This paper describes the design and implementation of the toolkit including its important interfaces, data structures, and algorithms. This paper is recommended for anyone wishing to modify the toolkit or anyone wishing to build their own EXPRESS implementation. The reader is assumed to be familiar with the EXPRESS language, the basics of traditional language implementations, and C – the language with which the toolkit is implemented.

As a testbed against which to benchmark the evolving EXPRESS language, conformance to the standard (currently Draft International Standard) is the highest priority in the toolkit. Nonetheless, time/space efficiency, accurate and helpful diagnostics, and ease-of-use are also critical to the success of the toolkit. The paper describes how these concerns are addressed even though EXPRESS is a complex and sophisticated language.

The toolkit is available from the National Institute of Standards and Technology. The toolkit is just one of a number of tools for data management in STEP, a family of ISO standards currently in development. All of the NIST tools, including the NIST EXPRESS toolkit, are in the public domain.

Keywords: compiler, EXPRESS; implementation; National PDES Testbed; PDES; STEP

CONTEXT

The PDES (Product Data Exchange using STEP) activity (Furlani, 1990) is the United States' effort in support of the Standard for the Exchange of Product Model Data (STEP) (ISO, 1992a), an emerging international standard for the interchange of product data between various vendors' CAD/CAM systems and other manufacturing-related software. A National PDES Testbed has been established at the National Institute of Standards and Technology to provide testing and validation facilities for the emerging standard. The Testbed is funded by the Computer-aided Acquisition and Logistic Support (CALS) program of the Office of the Secretary of Defense.

As part of the testing effort, NIST is charged with providing a software toolkit for manipulating STEP data. The NIST EXPRESS Toolkit is a part of this. The toolkit is an evolving, research-oriented set of software tools. This document is one of a set of reports (documented by Libes (1993a)) which describe various aspects of the Toolkit.

INTRODUCTION

The NIST EXPRESS toolkit is a software library for building EXPRESS-related tools. This paper describes the philosophy and design of the toolkit implementation including its important interfaces, data structures, and algorithms. Also, the conventions found within the toolkit will be described. This paper is recommended for anyone wishing to modify the NIST toolkit or anyone wishing to build their own EXPRESS implementation. The reader is assumed to be familiar with the EXPRESS language (ISO, 1990b), the basics of traditional language implementations, and C – the language with which the toolkit is implemented.

The implementation is subject to change. Because of this, no guarantee is made that the descriptions in this paper are still accurate with regard to the current implementation.

OVERVIEW

The toolkit is not a traditional translator in the sense of a compiler or interpreter which translates source into executable code. Rather, the toolkit is a library with which to build translators. For instance, a subroutine is provided to tokenize a stream of ASCII characters into a stream of EXPRESS tokens while another subroutine is provided to parse a stream of EXPRESS tokens.

The library stops short of providing subroutines to perform code generation since EXPRESS schemas are not intended to be translated directly into executable code. In lieu of this, the toolkit provides functions with which to query the EXPRESS schemas in a high-level way. These functions collectively can be considered a specialized database that describes the EXPRESS schemas.

The goal of the EXPRESS toolkit then is to populate the database and provide access to the database to application programs. Application-specific modules typically use these functions to translate the original EXPRESS to another language (producing application-dependent programs) or to immediately manipulate some other data (application-independent).

The toolkit Programmer's Reference (Libes, 1993b) defines bindings between function names and information in the database. In contrast, this document concentrates on the underlying data structures and algorithms used in the database. A comparison of this implementation to earlier implementations is presented by Libes and Clark (1992).

Lest the reader be concerned by the apparent complexity of the toolkit description herein, please note that it is not exposed to the application programmer. The internals are covered by suitable macros and functions; all the programmer sees is a very straightforward EXPRESS database.

BACKGROUND

It would be misleading to think that the work described here was created out of whole cloth. In this paper, references made to earlier work is sparse, yet much of what is described here depends on experience gained during that earlier work. The toolkit as it stands now has been redesigned and rewritten from top to bottom three times and in two different languages over a period of five years while the EXPRESS specification changed at least a dozen times.

As the EXPRESS specification matures (it is now a Draft International Standard), and with our implementation and usage experience, we have gradually settled upon the data structures and algorithms defined here. Libes and Clark (1992) describe experiences and rationale for these choices.

There is no such thing as a perfect implementation. An implementation is full of choices, such as speed vs. space. In that particular respect, the toolkit is strongly speed conscious. While space is well-managed, it is sacrificed immediately for speed. We believe this is a correct choice in the context of information modeling tasks today and in the future, with respect to the direction of computing resources.

Another trade off is data structures vs. algorithms. We have clearly tried to push as much of the complexity of the system as possible into the data structures to reduce the complexity of the code. This will be evident. While some of the data structures strongly parallel their EXPRESS counterparts, some are completely alien, existing only for supporting the algorithms.

OBJECTS

This section describes the types of objects used by the implementation, and how they map to EXPRESS objects. In theory, each EXPRESS object is represented as a C structure with pointers to other C structures representing other EXPRESS objects. For example, the EXPRESS notion of an Entity is represented by a C structure called "**`Entity`**".[1] EXPRESS entities include information about subtypes, supertypes, attributes, types, etc. Similarly, so does the C structure representing entities.[2]

```
struct Entity {
      ... subtypes
      ... supertypes;
      ... attributes;
};
```

In EXPRESS, virtually all objects are referenced multiple times. For example, Entity definitions are referenced by other Entity definitions. In order to efficiently make use of such references, the toolkit creates a C pointer to the object being referenced. In essence, a pointer provides the definition behind a reference.

For instance, examine the EXPRESS attribute declaration:

A: B;

This declares A as an attribute reference and B is the corresponding type. In order to effectively use A, the actual definition of B must be known. But B is defined elsewhere in the schema. In the toolkit, B is represented by another C structure. To record that A is of type B, we record a pointer to B within A.

The act of locating a referent involves searching through masses of definitions. This operation is called *dereferencing* or colloquially, *mapping back*. In order to efficiently respond to user queries, it is useful to search for each object no more than once. It is also useful to search for each object at least once – to verify that all ref-

[1] For each structure exposed to the user, a C typedef is defined which can point to it, i.e., **`typedef struct Entity *Entity;`** Structures used only internally to the implementation use the raw declaration form to avoid cluttering the user's namespace.

[2] C source is set in **`Courier bold`**. EXPRESS source is set in Times Roman (as is the text of this paper).

erences are meaningful. The toolkit is designed to address both concerns. It dereferences all references in a set of schemas, leaving a network of C structures connected by pointers. If any references could not be located, diagnostics are issued.

As the references in each object are being dereferenced, the object is said to be in *resolution*. After resolution is complete, the object is *resolved*. During resolution, it is convenient to check semantic consistency. For example, if an attribute reference is used where a type reference is expected, a diagnostic will be issued. Such checks are not a necessary part of resolution and can be disabled, but if desired, they are efficiently handled at this time.

Symbol

Every object in the original EXPRESS file is associated with a **`Symbol`** structure. This structure provides a place to record the object name. The structure also records the file name and line number corresponding to where the object was defined. The file name and line number are necessary only for generating diagnostics and are not otherwise used by the toolkit.

```
struct Symbol {
     char *name;
     char *filename;
     short line;
     char resolved;
};
```

A bitfield, **`resolved`**, describes whether the object has been resolved. The values of the bitfield are:

```
#define UNRESOLVED[1]
#define RESOLVED
#define RESOLVE_FAILED
#define RESOLVE_IN_PROGRESS
```

Dictionary, Object

A dictionary is used to store object pointers[2] which can later be retrieved by name. The dictionary is implemented as a traditional open hash table that expands when it is a certain percentage full. It is augmented with extra functionality specifically for the toolkit.

Unlike the typical hash table, it is possible to step through a dictionary sequentially, for example, to resolve each object in the dictionary. In addition, each entry is augmented with an object's symbol and type.

Since the dictionary knows the type of each object, it is possible to search only for specific types. This solves several problems. For instance, attribute types can be declared in the entity scope itself. However, EXPRESS allows attributes to be named the same name as the type:

A: A;

Since the dictionary knows the types of each object, it can skip attribute objects in this case where it is only interested in types. To select sets of different object types, the selectors are bit strings which can be ORed together. Some of the type selectors are:

```
#define OBJ_TYPE_BITS
#define OBJ_ENTITY_BITS
#define OBJ_FUNCTION_BITS
#define OBJ_EXPRESS_BITS
```

The dictionary's knowledge of types also allows an efficient solution to the problem of multiple enumerations with the same name in the same scope (in which the enumeration type is declared). EXPRESS allows this name clash but only for enumerations. Such ambiguous enumerations must be qualified in use while non-ambiguous enumerations need not be. Name clashes of other types are errors. Since the dictionary knows all objects' symbols, it can detect an existing declaration when the object is stored and report the problem accurately rather than having each caller handle it.

If two objects with the same name are indeed enumerations but in different enumeration types, the type in the common scope is changed from **`OBJ_ENUM`** to **`OBJ_AMBIG_ENUM`**. (If they are in the same enumeration type scope, than a diagnostic is issued reporting this.) This technique sacrifices extra storage space (for duplicate entries in two scopes), but greatly reduces processing time since scoping exceptions are only handled during dictionary collisions, rather than having a complex scope lookup algorithm.

Primarily for the benefit of diagnostics, a table (**`OBJ`**) is provided that maps object types to type selectors or an English description of the type. In addition, the table contains functions which can be applied to any type object that will yield its Symbol.

```
struct Object {
     struct Symbol *(*get_symbol)();
     char* type;
     int bits;
} OBJ[...];
```

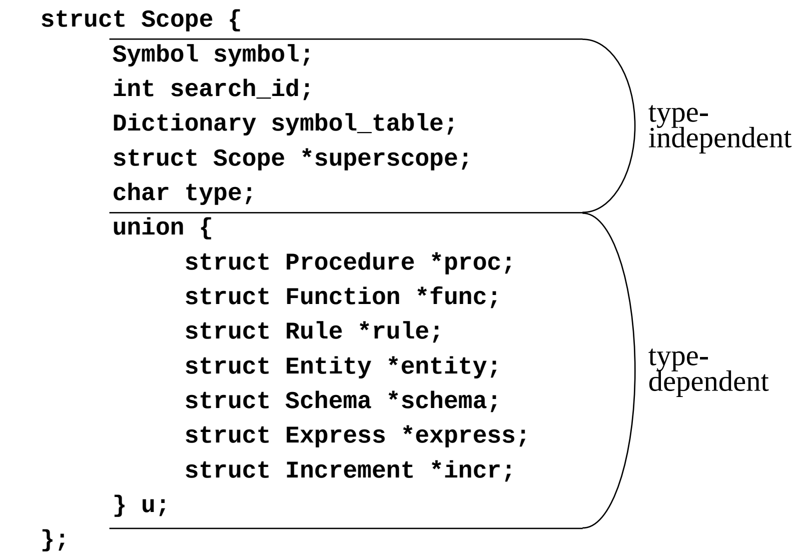
This is particularly useful in generating error messages of the form: "*Expected entity but encountered <some object name> of type <some other type name> on line <#> in file <file name>*." Additional types can be added to this table. For example, the NIST Part 21 Exchange File Toolkit (Libes, 1993c) defines "instances" as **`OBJ_INSTANCE`**.

Linked List

A **`Linked_List`** is the omnipresent linked list, used to store various lists of information. It is implemented (Libes, 1993d) as a doubly-linked list. The implementation will not be further described here.

[1] When irrelevant to the discussion, macro values are omitted.

[2] Object pointers are simply "**`void *`**" meaning that they can point to anything. While object-oriented techniques are used, this is more due to the requirements of the task being performed than what the environment supports.



```
struct Scope {
     Symbol symbol;
     int search_id;
     Dictionary symbol_table;
     struct Scope *superscope;
     char type;
     union {
          struct Procedure *proc;
          struct Function *func;
          struct Rule *rule;
          struct Entity *entity;
          struct Schema *schema;
          struct Express *express;
          struct Increment *incr;
     } u;
};
```

Figure 2: C representation of Scope

Scope

Many objects in EXPRESS define scopes, such as entities, types, and schemas. Some objects do not define scopes, but are more easily implemented by imagining that they do define scopes. The **`Scope`** structure, shown in Figure 2, defines elements common to all scopes, real or imaginary.

`search_id` prevents looping in the network of scopes created by partially-recursive USE statements. Each time a search is initiated, a new value for **`search_id`** is generated. As each scope is searched, its **`search_id`** is compared with the new **`search_id`**. If unequal, the **`search_id`** is set to the new **`search_id`**, and the scope is searched. If equal, the scope has already been searched.

`symbol_table` is a collection of pointers to all the objects directly defined by the scope. **`superscope`** is a pointer to the scope that lexically encloses the scope. For instance, if the current scope is an entity, the superscope might be the scope of the schema.

The lines in the listing are not present in the actual source. Here they simply emphasize that scopes contain type-dependent and -independent information. (Other object definitions are similar and will omit the lines from hereon.) The type of any particular scope is selected by the element **`type`** which effectively selects a member of the union **`u`**. For instance, an entity scope defines **`u.entity`** with the value of **`type == OBJ_ENTITY`**. **`Entity`** and other definitions are described later. Other scope types are:

```
#define OBJ_EXPRESS
#define OBJ_PROCEDURE
#define OBJ_RULE
#define OBJ_FUNCTION
#define OBJ_SCHEMA
#define OBJ_INCREMENT
```

Several scope types do not have type-dependent information. They are as follows:

```
#define OBJ_RULE
#define OBJ_ALIAS
#define OBJ_QUERY
```

A Note about Unions

As in the previous section on scope, many other structures in the system use unions, a C construct that permits storage of different types in a single place at different times. This is not so much for storage reasons (although this is helpful) but for type flexibility. Since unions are not particularly pleasant to use and they are pervasive in the toolkit, a brief note is appropriate.

EXPRESS objects can naturally be viewed as creating a class hierarchy. For example, scope would be inherited by procedures, functions and other objects. Indeed, an earlier implementation of the toolkit was written on top of an object-oriented (OO) engine and appropriate EXPRESS classes.

Pleasant as it was to write, the OO implementation sacrificed a great deal of speed. While the current toolkit code lays more of the typing burden on the implementor, there is actually 30% less code and the toolkit runs in less than 1% of the time than did the object-oriented implementation. Libes and Clark (1992) present a complete and fascinating discussion of the problems encountered and rationales in developing both systems.

Type

The EXPRESS type system is complex. This naturally shows up in the implementation. Consider the statements:

```
TYPE A = SET OF INTEGER;
TYPE B = A;
```

In this statement, A is the name of a type being defined while SET OF INTEGER is the type body. Two different structures fully represent a type. Type names (e.g., A) are associated to type bodies (e.g., SET OF INTEGER) by the structure **TypeHead**, while the type bodies themselves are defined by **TypeBody**.

```
struct TypeHead {
      Symbol symbol;
      struct TypeHead *head;
      struct TypeBody *body;
      Linked_List where;
};
```

If type references are defined in terms of other type references, **head** points to the other type reference. **body** points to the body definition. As a shorthand, **body** is always defined. Even if a type reference is defined in terms of another type reference, **body** points to the true type definition. Declarations of type references can have an associated where clause.

Type bodies encode notions such as set and bounds. Primitive types (e.g., INTEGER) are defined by the **type** element. There are over thirty type values, nine of which are primitive types. Several others denote types of expressions. A few others are noteworthy. Aggregate types (e.g., ARRAY) require that **base** point to the type of the aggregate value. Enumeration types require a dictionary in which enumeration identifiers can be found. Entity types include a pointer to the entity scope. The type **self** implies a reference to the entity or type defining the current type. Operations such as indexing from an array of SELECT types are not well-defined by the resolution process described here, but require run-time evaluation. In cases like these, the type **runtime** is used.

Several boolean flags record additional information such as whether a type is unique or optional. **shared** denotes that the type body is shared by multiple **TypeHead**s. This is used by various predefined types. For example, if multiple attributes are declared as INTEGER, they can all share the same **TypeBody**. **repeat** indicates that the object is a repeat count.

```
struct TypeBody {
      enum type_enum type;
      struct {
            unsigned unique :1;
            unsigned optional :1;
            unsigned fixed :1;
            unsigned shared :1;
            unsigned repeat :1;
      } flags;
      struct TypeHead *base;
      Type tag;
      Expression precision;
      Linked_List list;
      Dictionary enumeration;
      Expression upper;
      Expression lower;
      struct Scope *entity;
};
```

The remaining elements are generally optional and depend on the specific value of type. Several of these should be **union**ized for space efficiency, but this has not been a problem in our work.

An optional type tag may be recorded in **tag**, **precision** points to an optional precision expression, and **upper** and **lower** are optional expressions for aggregate bounds. Finally, **list** is a catch-all for types that need to keep object lists such as select types and composed types.

Variable

EXPRESS attributes and constants are almost identical as far as processing required. For this reason, the definition **Variable** is used for both. The choice of name is meant to be synonymous with “variables” in typical programming languages.

```
struct Variable {
      Expression name;
      Type type;
      Expression initializer;
      int offset;
      struct {
            int optional:1;
            int var:1;
            int constant:1;
      } flags;
      Symbol *inverse_symbol;
      Variable inverse_attribute;
};
```

Because attribute names can be compound, they are stored as **Expression**s. **type** is the variable type. **initializer** is set if the attribute is derived or the constant is initialized. If the variable is indeed an EXPRESS constant, the **constant** flag is set. **var** and **optional** correspond to the keywords of the same name.

inverse_attribute contains the attribute related by the optional inverse relationship. During parsing, the name of the inverse is known but not the attribute itself. Rather than allocating a dummy attribute, the code saves the name in **inverse_symbol**. After parsing, the symbol is mapped back to the true attribute which is then stored in **inverse_attribute**.

offset is one of several hooks for run-time support. In each entity, the size required to store each attribute value is computed.

Offsets are computed by adding up all the sizes prior to the current attribute.

Expression

EXPRESS expressions are complicated because of the many operators and forms these expressions take.

```
struct Expression {
        Symbol symbol;
        Type type;
        Type return_type;
        struct Op_Subexpression e;
        union expr_union u;
};
```

Each expression has a `Symbol`. Sometimes there is no reason to have a name (e.g., A+B) but sometimes there is. For example, enumeration identifiers, strings, and function calls all have names.

Syntactic and semantic types are recorded in `type` and `return_type`. For example, the expression A+B has the syntactic type of `op` (meaning "has opcode and operands") while the return type is `integer`, `real`, etc.

`Op_Subexpression` is a straightforward structure for recording an `op_code` and up to three operands (since EXPRESS supports operators with up to three operands). The next element is a `union` of other possible expression values. (The `Op_Subexpression` has not been folded into this union for historical reasons.)

```
union expr_union {
        int integer;
        float real;
        char *attribute;
        char *binary;
        int logical;
        Boolean boolean;
        struct Query *query;
        Linked_List list;
        Expression expression;
        struct Scope *entity;
        Variable variable;
};
```

Most of these fields can be intuited from their names. `list` is a catch-all for any expression that contains a list such as an aggregate definition or the parameters in a function. `variable` is for attribute references. A special structure is defined for query expressions.

```
struct Query {
        Variable local;
        Expression aggregate;
        Expression expression;
        struct Scope *scope;
};
```

`local` is the variable generated by the query expression. `aggregate` and `expression` are the set from which to test and the logical expression, respectively. Finally, `scope` is a scope created specifically created for the query expression.

Entity

EXPRESS entities are represented by a `Scope` structure that points to an `Entity` structure. The entity structure is defined as:

```
struct Entity {
        Linked_List supertype_symbols;
        Linked_List supertypes;
        Linked_List subtypes;
        Expression subtype_expression;
        Linked_List attributes;
        int inheritance;
        int attribute_count;
        Linked_List unique;
        Linked_List instances;
        int mark;
        Boolean abstract;
        Linked_List where;
        Type type;
};
```

`subtypes` and `supertypes` are linked lists of other entities. `subtype_expression` is an expression representing the directed-acyclic-graph form as originally specified by the SUPERTYPE clause. `supertype_symbols` is a temporary convenience. The parser stores the supertype names in a list of `Symbol`s as they are encountered. After parsing, they are mapped back to the true entities and stored in `supertype`s as a list of `Entity`s. This procedure is not necessary for `subtypes`, since expressions are self-describing while lists are not. The same expression can be re-used whether it points to a symbol or an entity.

`attributes` is a list of the entity's attributes. `unique` is a list of the unique attributes. `where` is a list of Expressions denoting WHERE clauses.

`attribute_count` is the number of attributes directly defined by the entity while `inheritance` is the number of attributes inherited by the entity. `instances` is a pointer not set by the implementation. Rather, it is provided to run-time packages so that they can point to entity instances. All of these elements are not required by the toolkit but are provided as a convenience.

Two other elements are not required but exist as internal conveniences. `mark` is used to prevent repeated traversal of the subtype/supertype network while searching for entities. The use of `mark` is analogous to `search_id` in `Scope`s. `type` is a pointer to an entity type with the value of `entity` equal to this entity. This is particularly useful to have easily available when evaluating expressions involving entities.

Warning, Error

For consistent diagnostic generation and handling, a diagnostic package is provided. Each error/warning can be enabled or disabled. This is extremely useful for EXPRESS, a language that has not been standardized and is continually changing. Constructions acceptable in this release may not be accepted in the next release and vice versa. Errors are formatted appropriately so that they are acceptable to "GNU emacs compile-mode". Using this ability of the GNU emacs editor (Stallman, 1992), the user can move the cursor from one diagnostic to the next in one window, while in a second window, the schema is automatically repositioned to the line that the diagnostic describes.

The diagnostic package supports a number of features that may no longer be particularly useful with the current toolkit implementation. For example, diagnostics can be buffered and sorted. This was particularly useful with previous toolkits that generated diagnostics in a non-intuitive order. Although the toolkit no longer needs these features, it is possible that applications may find them of use.

Currently, diagnostics are generated in the order of greatest importance to least importance. This is a consequence of the multiple-pass nature of the resolution process and will be mentioned further later. The diagnostic package is further described by Clark (1990a).

Statement

EXPRESS statements have almost nothing in common with one another except that they usually contain expressions and other statements. Thus, a common structure is defined. But it is more a grouping artifice so that all statements are easily processed at the same time in the toolkit.

```
struct Statement {
     Symbol symbol;
     int type;
     union u_statement {
          struct Alias *alias;
          struct Assignment *assign;
          struct Case_Statement *Case;
          struct Compound_Statement
*compound;
          struct Conditional *cond;
          struct Loop *loop;
          struct Procedure_Call *proc;
          struct Return_Statement
*ret;
     } u;
};
```

Inside the **`Statement`** structure is a **`union`** of structures for each different kind of EXPRESS statements. They are as follows:

```
struct Alias {
     struct Scope *scope;
     struct Variable *variable;
     Statement statement;
};

struct Assignment {
     Expression lhs;
     Expression rhs;
};

struct Case_Statement {
     Expression selector;
     Linked_List cases;
};

struct Compound_Statement {
     Linked_List statements;
};

struct Conditional {
     Expression test;
     Statement code;
     Statement otherwise;
};

struct Loop {
     struct Scope *scope;
     Expression while_expr;
     Expression until_expr;
     Statement statement;
};

struct Increment {
     Expression init;
     Expression end;
     Expression increment;
};

struct Procedure_Call {
     struct Scope *procedure;
     Linked_List parameters;
};
```

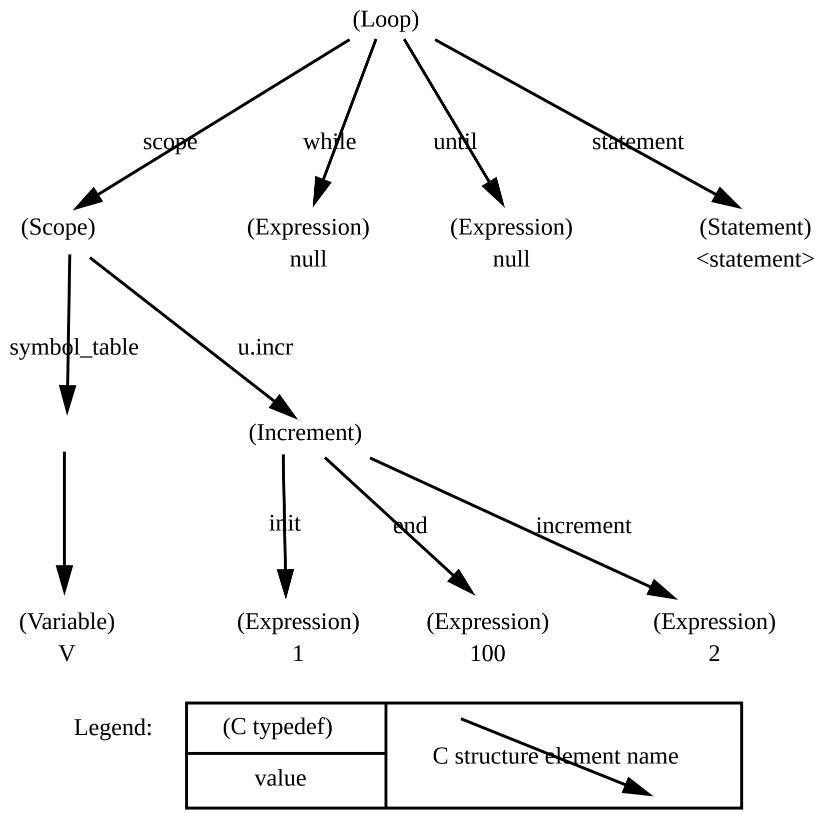


Figure 2: Representation of EXPRESS REPEAT statement

```
struct Return_Statement {
      Expression value;
};
```

Some of these statements have interesting behaviors. In particular, **Procedure_Call** requires a scope to deal with tagged parameters. Similarly, **Alias** introduces a very tiny scope – of only one variable! It is convenient to use the **Scope** mechanism even though EXPRESS proper does not define these statements in terms of having scopes.

REPEAT statements require a scope if an increment control is specified, in which case it is provided by a **Scope** of type **Increment**. Figure 2 shows a symbolic representation of the following REPEAT statement in memory:

```
REPEAT V:=1 TO 100 BY 2;
  <statement>
END_REPEAT;.
```

The SKIP and ESCAPE statements do not have defined structures. Since every SKIP statement is identical, the **type** field of **Statement** suffices to describe it completely. The ESCAPE statement is analogous.

While statements are not named by the user, it is still useful to tag each one with a **Symbol**, since this information can be used to identify the source (line number and file name) should the user need to be informed about a conflict arising involving the statement. Several other structures in the toolkit have no name, but are usefully tagged in this same way.

Case Item

The CASE statement is defined by a list of actions. Each action may be labelled by a number of expressions. The following structure is used to hold the association between a single action and a list of expressions (labels):

```
struct Case_Item {
      Symbol symbol;
      Linked_List labels;
      struct Statement *action;
};
```

Express

An artificial construct, **Express**, is used to refer to all the schemas and their contents as a single model. This allows the possibility of reading two EXPRESS definitions and comparing them, generating one from another, or other applications using multiple models.

The actual definition of an **`Express`** structure contains the file descriptors and other miscellaneous detail that is not relevant here. The important aspect of **`Express`** is that it is owned by a **`Scope`** structure which supplies a dictionary containing all the top-level information that would normally be found in a schema file such as SCHEMA and CONSTANT information.

Schema, Rename

Each schema is maintained by a **`Scope`** structure which points to a **`Schema`** structure.

```
struct Schema {
     Linked_List rules;
     Dictionary refdict;
     Dictionary usedict;
     Linked_List uselist;
     Linked_List reflist;
};
```

`rules` is a list of WHERE clauses represented as **`Expression`**s. Everything else in the structure supports intra-schema references. References to entire schemas are placed on **`uselist`** (for USE) or **`reflist`** (for REFERENCE). During parsing, references to specific objects are inserted into **`refdict`** or **`usedict`**. Objects are entered under their new name if they are renamed, such as in:

REFERENCE FROM SCHEMA1 (OBJ1 AS OBJ2)

All dictionary references are made using the following structure:

```
typedef struct Rename {
     struct Symbol *schema_sym;
     Schema schema;
     struct Symbol *old;
     struct Symbol *new;
     Generic object;
     char type;
     enum rename_type rename_type;
};
```

During parsing, schemas names are saved in **`schema_sym`**. **`old`** and **`new`** are the original and new name of the object. If the object is not being renamed, **`new == old`**.

After parsing, each reference is tracked through any number of referencing schemas to the schema in which the object is actually defined. A pointer to the defining schema is stored in **`schema`**. An object pointer to the object definition is stored in **`object`** and its type in **`type`**. **`rename_type`** is set to either **`use`** or **`ref`**, allowing later searching to behave differently for the two.

[2] contrasts the current implementation with a prior implementation that literally copied referenced objects from one schema to another.

Procedure, Function, Rule

The following definitions support procedures, functions, and rules.

```
struct Procedure {
     int pcount;
     int tag_count;
     Linked_List parameters;
     Linked_List body;
};

struct Function {
     int pcount;
     int tag_count;
     Linked_List parameters;
     Linked_List body;
     Type return_type;
};

struct Rule {
     Linked_List parameters;
     Linked_List body;
     Linked_List where;
};
```

`parameters` is a list of lists of type expressions. For convenience to the toolkit, **`pcount`** is a count of parameters. In contrast, **`tag_count`** is just the number of parameters which are tagged (with type labels). The tagged parameters may first require resolution of other tagged parameters, for example if one is marked INTEGER while another is GENERIC. A more likely scenario is that the algorithm gives no type information which instead must be garnered from the algorithm callers. This can change at each call and even mid-call. For example, imagine a function with the following header:

```
FUNC FOO(
                A:     GENERIC;TAG1;
                B:     GENERIC:TAG2;
          ):           GENERIC:TAG2;
```

which is called as:

FOO(FOO(T1,T2),FOO(T2,T3))

In order to determine the types of this function's arguments, it is necessary to determine the type of the function in two other calls at the same time! Hence the type labels cannot be associated with the functions (which is why they don't show up in the **`Function`** structure definition) but are associated with each individual function call.

For each type label and call, a **tag** structure holds the mapping between name and type. This is discarded after resolution.

```
struct tag {
	char *name;
	Type type;
};
```

Memory

A special-purpose memory allocation system (Libes, 1993d) is provided that understands the predominant number of requests for memory that come in a small number of sizes corresponding to the different objects in the system. It speeds up the entire toolkit by approximately 10-15% over the more general-purpose memory allocators (i.e., malloc and friends) provided with C compilers.

The memory allocation system will not be further described here.

MULTIPLE PASSES – OVERVIEW

The toolkit provides a number of functions that combine to implement a multiple-pass translation. Normally, these passes are called consecutively however, this is not necessary. Nonetheless, we will describe this as the 'usual' use of the toolkit.

An earlier release of this software used a three pass total, single pass resolution system. An attribute of single pass resolution is that objects have to be resolved when they are first encountered. There are a number of drawbacks to this including: heavily recursive processing can be complex especially when handling errors; detection of recursive references takes a lot of overhead; and diagnostics are generated in a very non-intuitive way as processing follows depth-first paths of references.

In contrast to that system, the multiple pass architecture of the current system guarantees that all objects are resolved before being used. This makes error handling simpler – errors are handled at very shallow levels of processing. Lastly, since objects are resolved only after all their dependencies are, diagnostics are naturally generated in a most-meaningful-first order. Libes and Clark (1992) present a complete discussion of the differences between single and multiple passes in EXPRESS translation.

The specific passes are briefly described below. They will be described further later in the paper.

Pass 1: **Parse**

Tokens are scanned and a parse tree is built representing a collection of schemas. A dictionary is created for each scope, and names are entered into the dictionaries during parsing.

If any referenced schema is unknown, a search is made to find an appropriate file in the file system and the Parse pass is repeated.

Pass 2: **Resolve Use and References**

Use and reference lists and dictionaries are traversed and connections are created to objects in the defining schemas.

Pass 3: **Resolve Subtypes and Supertypes**

Entity subtypes and supertypes are resolved.

Pass 4: **Resolve Other Static Types**

All static type definitions, attribute types, return types, etc., are resolved.

Pass 5: **Resolve Expressions, Statements, and Dynamic Types**

Expressions and statements are resolved. Types for ALIAS and other dynamically-typed variables are resolved here since they depend upon expression return types, and the types are not necessary for earlier results.

Pass 6: **Application-Specific Backend**

While not restricted to following all the other passes, application-specific code is usually performed at this point.

MULTIPLE PASSES – DETAIL

The multiple passes of the toolkit will now be described in detail.

Scanner

Individual characters are read and grouped into tokens by the *scanner*. The current implementation of the scanner can be built either with Lex or Flex (Lesk and Schmidt, 1978).

The only salient difference between Lex and Flex is that Flex does not count lines (required for tagging diagnostics and objects). Some other difference exist but they are minor syntactic-level elements handled by **#ifdef**s. Flex provides support for scanning by state. Using this would simplify the scanner, but we have avoided it for portability.

As the scanner runs, comments are discarded. While comments have semantic value, they could conceivably be maintained by the implementation. Unfortunately, there is no obvious way to associate comments with objects, since comments can appear between arbitrary tokens.

EXPRESS allows comments to be nested, so the scanner keeps a stack for this purpose. A termination routine reports all non-terminated comments. EXPRESS also allows non-nested comments, which the scanner carefully handles even in the presence of nested comments. (In other words, they are both handled as described in the EXPRESS specification.)

A token is returned for each EXPRESS reserved word, symbol, and identifier. Numbers, literals, etc. are returned as single tokens. In the case of tokens such as **`TOK_INTEGER`** and **`TOK_IDENTIFIER`**, the **`union exp_yylval`** is used to store additional data. For example, when an integer is scanned, the value of the integer is stored in ex**`p_yylval.iVal`** and **`TOK_INTEGER`** is passed to the parser.

EXPRESS identifiers are case-insensitive. Thus, all identifiers are translated to uppercase. If they are keywords or built-in functions or procedures), the matching token is returned. If they are not keywords, they are presumed to be user-defined identifiers. If identifiers, the name, line number, and file name are stored in **`exp_yylval.symbol`** and **`TOK_IDENTIFIER`** is returned to the parser.

Parser

The parser is a traditional LR parser. The current implementation runs either with Yacc (Johnson, 1978) or Bison (Stallman, 1992). The following discussion assumes a rudimentary familiarity with Yacc parsers.

The grammar is fairly close to that found in the current EXPRESS specification, however there are several reasons why it diverges significantly in many places:

- The grammar evolved over time and multiple EXPRESS specifications, rather than being rewritten from scratch with each release of the specification. For the most part, the minimum changes necessary were made to keep the language accepted in conformance with the specification.
- Little semantic analysis occurs during parsing. For this reason, the parser itself accepts some constructs that are illegal, the idea being that they will be caught during semantic processing later. While this requires some additional complexity later, the parser is greatly simplified. For example, a number of specification rules can be handled by a single Yacc rule.
- The BNF used by the specification is a superset of what is accepted by Yacc. Some of the rules had to be simplified.
- Parts of the grammar were rewritten to be more efficient to the LR parser built by Yacc. The EXPRESS grammar provides no assistance in building efficient implementations.

As the parser runs, it builds a tree representing its input. (After parsing is completed, the tree will then be manipulated into a network.) At the root of the tree is a dictionary of schemas. Each schema in turn is a tree to entities, functions, etc.

The only semantic processing that occurs during parsing are object definitions. For example, when the string “ENTITY FOO” is encountered, an entity is created. This entity is entered into the previous scope. Similarly, a scope is created for future attributes that will be encountered while parsing the rest of the entity definition. Other objects are created similarly.

A stack called **`scopes`** is used to retain lexical knowledge of scopes as they are encountered. This is primarily used to simplify access to the current scope while adding objects to it, however it also simplifies other actions such as scope creation and generation of more descriptive contexts in diagnostics.

As each schema reference is encountered, it is added to a fifo called **`PARSEnew_schemas`**. This is explored immediately after parsing. If any unknown schema references are found, matching files are searched for and the parser is restarted.

The **`exp_yylval`** union is used extensively during bottom-up parsing to pass low-level information back up to higher-level rules. A few rules require information to be passed down, but these are rare. Implemented as static variables, they can be found at the beginning of the parser source.

One of the shortcomings in Yacc/Bison has been its crude error messages. We have improved that by incorporating a mechanism (Schreiner and Friedman, 1985) to report what tokens were encountered and what tokens were expected. Due to the design of Yacc/Bison, this modification requires one-time hand editing of the Yacc/Bison template. This must be done when the toolkit is ported to a new computer (or the parser is “upgraded”).

The Yacc parser is composed of machine-generated C, and as such is not particularly efficient (although we use GNU’s Bison (Stallman, 1992) , a more efficient version of Yacc). Nonetheless, we have not tried to rewrite an EXPRESS parser by hand. While such an effort would produce more efficient code, we expect that it would be a lengthy task, and further changes would be extremely painful.

The scanner is similarly constructed by Lex (or Flex, a faster Lex). Currently, the parser and scanner together take 75% of the time used by parsing and resolution of an EXPRESS schema. We speculate that a hand-built pair could reduce the total run-time by as much as 50%.

After Parsing But Before Resolution

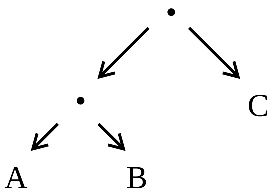
As mentioned earlier, the parser creates a tree representing all the information (except comments) found in the original schema file. At this point, most of it has little semantic information. For example, in the attribute declaration ATTRX = TYPEX, TYPEX becomes represented as a Type but with no information except its name. Later resolution will determine whether it is a Type, Entity, or erroneous in some way.

In some parts of the grammar, it is impossible to tell anything about a token other than it is a name for some object. While EX-

PRESS has a well-defined (BNF) syntax, without semantic help it is possible to parse valid schemas in multiple ways. Unfortunately, EXPRESS allows objects to be referenced before being declared, so semantic assistance is impossible in all cases. Thus, may tokens are forced to be represented simply as **`Symbol`**s – that is, just a name and a record of where it was found. During later resolution, it will be replaced or augmented.

Expressions are represented in the form suggested by the precedence of their operators. For example, the expression “X + (Y + Z)” is represented by two **`Expression`**s. Both have + as operators. The first has Y and Z as its operands. The second has X as its first operand and a pointer to the first **`Expression`** as its second operand.

`Expression`s are used to represent several other things. For instance, each entity has an expression representing its subtype relationships. Entity-attribute dereferencing is another example of an expression. For example, the EXPRESS expression “A.B.C” is represented in memory as:



Recalling the Parser

If schemas have been referenced but not defined, a schema file is searched for so that references can be resolved. This is performed by **`EXPRESSfind_schema`**. This function examines each directory defined by the environment variable **`EXPRESS_PATH`** (or the current directory if **`EXPRESS_PATH`** is not defined) for a file by a name similar to that of the schema itself. The only difference is that the schema name is converted to lowercase and “**`.exp`**” is appended.

If such a file is found, it is read and parsed (as described earlier). This procedure of resolved schema references is repeated as long as such references exist.

As schemas are located, USE and REFERENCE lists are resolved as described earlier. After parsing, they are initially simply lists of schema names and local (and possibly remote) names. A separate pass is made to establish connections to remote schemas. A second pass connects the actual objects (**`Entity`**s, **`Type`**s, etc.) together. This separation of passes avoids problems related to partial or total recursion between schemas that reference each other.

Resolving Subtypes, Supertypes, and Type Definitions

Once all the schemas have been resolved, object types must be resolved. Unlike schemas which occur only at the top-level of an EXPRESS model, types can be nested within other objects including other types. Because of this, type resolution recurses if necessary to resolve all types.

Type resolution is broken into several steps, partly for simplicity and partly of necessity.

During the first type resolution pass, subtypes, supertypes, and type definitions are resolved. **`Symbol`**s representing types are sought out and replaced by pointers to the true type. A recursive search to do this is fairly simple, because the only object that can contain other objects are scopes. Pseudocode for this is as follows:

```
SCOPEresolve_subsupers(Scope s)
    for each object x in scope dictionary s
        if object type is
        OBJ_ENTITY:
            ENTITYresolve_supertypes(x);
            ENTITYresolve_subtypes(x);
        OBJ_FUNCTION:
        OBJ_PROCEDURE:
        OBJ_RULE:
            SCOPEresolve_subsupers(x);
        case OBJ_TYPE:
            TYPEresolve(x,scope);
        default:
            /* ignored everything else */
}
```

Notice that resolution of algorithms (**`OBJ_FUNCTION`**, etc.) does nothing but recurse looking for entities and types. When entities are found, the sub and supertypes are traversed and resolved. When types are found in a scope, the scope must also be passed to the resolver since our representation of a type does not include a scope.

In each of these routines, scopes are used to search for objects. For instance, if an entity has declared that it is a supertype of another entity, the name of that other entity is known but nothing else. The search strategy to locate the entity is as follows:

Step 1: If the object is in the local Scope, return it.
Step 2: Else if the current Scope is not a Schema, repeat step 1 with the superscope.
Step 3: When the schema is reached:
Step 3.1: Search the scope of every full USEed schema.
Step 3.2: Search the rename dictionary of all partially USEed schemas.
Step 3.3: Search the scope of every fully REFERENCEd schema.

Step 3.4: Search the rename dictionary of all partially REFERENCEd schemas.

Since the subtypes and supertypes form a network rather than a simple tree, each type is tagged as it is seen. Much like a garbage collection algorithm, this prevents the possibility of looping over types already seen.

Resolving Static Types

After all type definitions are known, they can be referred to by other objects. For example, attributes can redefine other attributes in inherited supertypes. Naturally, the supertype must be resolved in order to extract its attributes. Hence, supertype resolution must precede attribute resolution. Other miscellaneous static type resolutions can also be performed at this time such as the resolution of function return types.

Searching for inherited entities is a common operation. The search strategy to locate such an entity is as follows:

Step 1: For each supertype of the entity:
Step 1.1: If the supertype name matches, return the supertype.
Step 2: For each supertype of the entity:
Step 2.1: Recursively apply this search.
Step 2.1: If the supertype is found, return it.

As with the earlier search strategy, each entity is tagged as it is encountered to prevent looping.

Resolving Expressions, Statements, and Dynamic Types

All expressions have a semantic type, such as INTEGER or ARRAY OF REAL. These types must be resolved by visiting each expression and examining its operands and operations. For instance, in the expression "A+B", if A and B are both integers, the resulting expression is an integer. The expression is tagged with the type which can then be propagated upwards.

Expression types are examples of dynamic types. Dynamic types are those that depend upon context. Not all dynamic types are expressions although they almost always depend upon expressions at some level. For example, the control variable of a query expression is not explicitly typed. Rather the type is intuited from the query expression. Fortunately, it is always possible to determine the type of a query expression before having to determine the type of the control variable.

EXPRESS statements must also be resolved. Statements do not have return types, but the types of expressions used within statements must be resolved. For example, the assignment statement has a left-hand-side and a right-hand-side, both of which are expressions. These must be type-compatible.

Expression-type resolution is quite complicated because there are many EXPRESS expressions and types and they can interact in a variety of ways and have a large number of exceptions. Here are some examples:

Functions need not have parameters. There is no way for the parser to distinguish parameter-less functions from other identifiers such as entity or variable names. This can occur only later when all the identifiers have been defined. Similarly, entities can also behave like functions or types depending on context. The expression resolver must therefore have the flexibility to treat identifiers in expressions in vastly different ways.

SELF matches the nearest enclosing type or entity. It is not sufficient simply to use the nearest enclosing **`Scope`** since this can also be a **`Query`**, **`Alias`**, etc. If an entity is used as an implicit constructor function (such as in a derived initialization of an entity), SELF can potentially refer to the function (originally an entity).

A large number of functions and operators exist. They can be grouped into different classes depending upon what type of processing is required to derive the return type from their arguments. To simplify processing, the toolkit resolves functions and operators by table-lookup. An operator-resolution table produces a function based on the opcode which can be called with the user-supplied arguments to resolve the operator expression. A function-resolution dictionary produces similar information based upon the name of the function. A similar dictionary is provided for procedure-resolution.

Both function and procedure resolution require typechecking of the supplied parameters against the formals. Formals can declare dependencies against other formals or even parts of formals via the tag mechanism. GENERIC formal declarations with matching tags require that parameters be typechecked against each other, but through the function or procedure – a circuitous path.

A very few types cannot be resolved. For example, evaluation of a group operation requires a specific entity, however EXPRESS allows constructs such as entity aggregates or selects. It is conceivable that two entities are provided, one of which has a referred to attribute and one which does not. Should the former be presumed? Should a diagnostic be issued. What if three entities are supplied, two of which match? What if a set of entities is supplied, indexed by a run-time value. There is no general solution to this other than run-time evaluation. Expressions such as these are marked with a type of **`runtime`**.

Reporting Suspicious Constructions

In each pass, the toolkit issues diagnostics when an EXPRESS error prevents complete resolution. A small number of other common user-mistakes are checked which are not actually necessary to performance of the resolver.

There are an infinite number of "suspicious constructions" which could be checked. For example, shadowed types are not illegal but are probably a mistake nonetheless. (Or they might not be.) Classical compiler optimization such as code-hoisting can discover likely errors in semantics, yet this does not seem appropriate to the realm of the translator. Since semantic checks can potentially take an unbounded length of time, we encourage people to tell us what kinds of things are feasibly worth checking.

Additional checking could be added to each pass and/or could be a separate pass.

Application-Specific Backend Processing

While not restricted to following all the other passes, application-specific code is usually performed at this point. There are no restrictions whatsoever on what can be done.

Not only can the model be read, but it can be modified and augmented. Additional schema files can be read. Because each structure includes a resolution status, the resolution functions will not attempt to resolve objects that are already resolved (or have failed to resolve).

A variety of tools have been built using the toolkit including a graphical schema editor (Clark, 1990b), an EXPRESS-to-SQL translator (Morris, 1990), a Part 21 Exchange File parser (Libes, 1993e), and an EXPRESS-to-C++ translator (Morris et al, 1993).

SUMMARY

This paper has sketched out the important data structures and algorithms for an EXPRESS toolkit. The toolkit and its predecessors serve as a testbed for the continued evolution of the EXPRESS language, a baseline for commercial systems, and a source of experience from which to draw on when designing new implementations or modifying existing ones.

TO OBTAIN THIS SOFTWARE

The software described in this paper is in the public domain. Contact the Factory Automation Systems Division (301 975-3508 or npt-info@cme.nist.gov) to obtain the software, or for any information related to NIST work on the National PDES Testbed.

REFERENCES

Clark, S. N., “Fed-X: The NIST Express Translator”, NISTIR 4371, National Institute of Standards and Technology, Gaithersburg, MD, July 1990a.

Clark, S. N., “QDES User’s Guide”, NISTIR 4361, National Institute of Standards and Technology, Gaithersburg, MD, June 1990b.

Furlani, C. “ Status of PDES-Related Activities (Standards & Testing)”, NISTIR 4432, National Institute of Standards and Technology, Gaithersburg, MD, October 1990.

International Organization for Standardization, “ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Overview and Fundamental Principles”, Draft International Standard, ISO TC184/SC4, 1992a.

International Organization for Standardization, “ISO 10303 Industrial Automation Systems and Integration — Product Data Representation and Exchange — Description Methods: The EXPRESS Language Reference Manual, Draft International Standard”, ISO TC184/SC4, 1992b.

Johnson, S.C., “Yacc: Yet Another Compiler compiler”, *UNIX Programmer’s Manual*, Seventh Edition, Bell Laboratories, Murray Hill, NJ, 1978.

Lesk, M.E. and Schmidt, E., “Lex: A Lexical Analyzer Generator”, *UNIX Programmer’s Manual, Seventh Edition*, Bell Laboratories, Murray Hill, NJ, 1978.

Libes, D., “The NIST EXPRESS Toolkit – Introduction and Overview”, National Institute of Standards and Technology, Gaithersburg, MD, 1993a.

Libes, D., “The NIST EXPRESS Toolkit - Programmer’s Reference”, National Institute of Standards and Technology, Gaithersburg, MD, 1993b.

Libes, D., and Clark, S., “The NIST EXPRESS Toolkit – Lessons Learned”, *Proceedings of the 1992 EXPRESS Users’ Group (EUG ‘92) Conference*, Dallas, TX, October 17-18, 1992.

Libes, D., “The NIST STEP Part 21 Exchange File Toolkit: An Update”, National Institute of Standards and Technology, Gaithersburg, MD, 1993c.

Libes, D., *Obfuscated C and Other Mysteries*, John Wiley and Sons, pp. 71-77, New York, NY, January 1993d.

Libes, D., “The NIST STEP Part 21 Exchange File Toolkit: An Update”, National Institute of Standards and Technology, Gaithersburg, MD, 1993e.

Morris, K.C., “Translating EXPRESS to SQL: A User’s Guide”, NISTIR 4341, National Institute of Standards and Technology, Gaithersburg, MD, May 1990.

Morris, K.C., Sauder, David, and Ressler, Sandy, “Validation Testing System: Reusable Software Component Design”, National Institute of Standards and Technology, Gaithersburg, MD, 1993.

Schreiner, A. T. and Friedman., Jr., H. G., *Introduction to Compiler Construction with UNIX*, Prentice Hall, New York, NY, 1985.

Stallman, R. M., et al, *GNU’s Bulletin*, Free Software Foundation, Inc., Cambridge, MA, June 1992.