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

NISTIR 4822

National PDES Testbed Report Series

Fed-X: The NIST Express Translator

Revised April, 1992

Stephen Nowland Clark
Don Libes

NATIONAL PDES TESTBED™

April 3, 1992

NIST

NISTIR 4822

National PDES Testbed
Report Series

Sponsored by:

U.S. Department of Defense
CALS Evaluation and
Integration Office
The Pentagon
Washington, DC 20301-8000

NATIONAL PDES TESTBED™

Fed-X: The NIST Express Translator

Revised April, 1992

Stephen Nowland Clark
Don Libes

U.S. Department of Commerce
Barbara Hackman Franklin,
Secretary

Technology Administration
Robert M. White,
Undersecretary for Technology

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

April 3, 1992

DEPARTMENT OF COMMERCE
UNITED STATES OF AMERICA

Table Of Contents

Disclaimer

Fed-X: The NIST Express Translator

Stephen Nowland Clark
Don Libes[1]

1 Introduction

The NIST (Federal) Express Translator (Fed-X), and the associated Express Working Form, are Public Domain software tools for manipulating information models[2] written in the Express language [Part11]. The Express Working Form is part of the NIST PDES Toolkit [Clark90a]. It is intended to be used to provide the input to various conceptual-schema-driven applications in a STEP implementation. For example, tools such as Data Probe, a prototype STEP and Express schema browser and editor developed at NIST [Morris91], and the STEP Working Form with its associated STEP physical file parser, STEPparse [Clark90b], have been written independently of any particular information model. Fed-X-based translators are used to provide the information model definitions to drive these applications. This approach results in smaller applications (which need not have entire information models embedded within them), as well as insulating these applications against changes in the conceptual schema and, to a certain extent, in Express itself. Indeed, an application such as STEPparse can be used with different conceptual schemas, or different versions of the same schema, without modification. The Data Probe has been used to edit STEP product models in the context of several different Express information models.

A primary goal in the development of Fed-X was to provide a clean back-end interface, in order to allow various output modules to be easily plugged into a basic front-end parser. To accomplish this, the Fed-X parser populates a set of data structures (the Express Working Form, or WF) containing all of the information in an Express specification. A user-supplied back-end[3] can then walk through the data structure, extracting relevant portions of the available data and producing an appropriately formatted output file.

1. Don Libes is responsible for the minor changes made to this document to track the actual implementation of the software described. However, credit for the bulk of the document, its style, and the implementation of the NIST Express Working Form remains with Stephen Nowland Clark. Recent changes are denoted by a change bar to the left of the text.
2. The terms *information model*, *data model*, and *conceptual schema* are used interchangeably throughout this document.
3. Two Fed-X output modules have been provided with the NIST PDES Toolkit in the past; they are not currently distributed with the toolkit. One of these produces Smalltalk-80™ class definitions [Clark90e] for use with QDES. The other forms the back end of Fed-X-SQL, a translator which produces relational database table definitions in SQL from an Express information model [Morris90] [Metz89].

1.1 Context

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

The reader is presumed to have a working knowledge of Express N14 and the C programming language.

2 Implementation Environment

Fed-X was developed on Sun Microsystems Sun-3™ and Sun-4™ series workstations running the Unix™ operating system. The Working Form is implemented in ANSI Standard C [ANSI89]. The Fed-X parser itself is implemented in Yacc and Lex, the Unix languages for specifying parsers and lexical analyzers. In the NIST development environment, the grammar can also be processed by Bison, the Free Software Foundation's[1] implementation of Yacc. Similarly, the lexical analyzer can be produced by Flex[2], a fast, Public Domain implementation of Lex. The C compiler used is GCC, also a product of the Free Software Foundation. The implementation currently depends on certain features of Standard C but presumably, any conformant compiler could be used.

3 Running Fed-X

A default main procedure is available for applications which choose not to supply their own top-level control. The following section describes invocation of applications built this way.

Fed-X takes several optional command-line arguments:

```
fedex [-d <number>]
      [-e <express>]
```

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 privileges, but grants free distribution rights under certain terms. At this writing, further information is available by electronic mail on the Internet from gnu@prep.ai.mit.edu.
2. Vern Paxson's Fast Lex is usually distributed with GNU software. It is, however, in the Public Domain, and is not an FSF product. Thus, it does not come under the FSF licensing restrictions.

`{-w|-i all|none|<warning>}`

The `-d` option controls the debugging level; the argument can range from 0 (the default) to 10. The Express source file is specified with `-e`; if no `-e` option is given, Fed-X reads from standard input. The last two options control which warning messages Fed-X will produce. `-w` is used to turn on warning classes and `-i` (ignore) to turn them off. A parameter of `all` behaves in a predictable fashion, instructing Fed-X to enable/disable all of the warning classes initially; similarly, `none` instructs Fed-X to begin with no warning classes enabled/disabled. Allowable values for `<warning>`, with their interpretation and default values, are:

- `subtypes` - Warnings about subtypes: Fed-X only traverses the class hierarchy by way of superclass information, so problems in subclass lists can "safely" be ignored. Default: on.
- `code` - Warnings about problems in algorithms and `where` clauses. Fed-X does not yet handle all of Express' scoping rules properly, nor does it attempt to compute the return types of expressions, so some of these warnings may be extraneous. Default: off.
- `comment` - Nested comment warning. Default: off.
- `shadows` - Warnings about overloaded names. The scoping rules of Express can disambiguate these shadowed definitions, but cannot be invoked outside of Express, e.g. in STEP files. Default: on.

Fed-X can be built in two different ways, resulting in different interaction patterns. For many applications, a single output module is bound into Fed-X at build time. In this statically linked case, after the first two passes are completed, the user is normally prompted for a single file name. This is the name of the file to which Fed-X's output will be written. In the other (dynamically linked) version, no specific output module is loaded at build time. In this case, when the first two passes are complete, the program asks for an output module. If the file named is an appropriate object file, it is loaded and an output file name requested. This is the name of the file to which the output will be written. Another output module is then requested, and this sequence continues until an empty line is entered as the name of the output module, which signals Fed-X to exit. This dynamic loading facility is available only under BSD4.2 Unix and its derivates.

4 Design Overview

Fed-X is a three-pass translator. The first two passes are the standard parsing and symbol-table resolution passes of a traditional compiler. The third is a flexible output generation pass. The Working Form which is produced by the first two passes consists of data structures which directly reflect the structure and contents of the Express source. The third pass, which can be tailored to various specific applications, traverses these data structures and produces output in a specified format.

4.1 Fed-X Control Flow

4.1.1 First Pass: Parsing

The first pass of Fed-X builds a set of data structures which completely represent the information in the Express input. This pass makes no attempt at resolving most name references; thus, the resulting data structures are linked only indirectly by names: in order to resolve a function call, the name of the function must be looked up in the symbol table for the appropriate scope. The entire structure of the file is represented at this point, however. If any syntax errors are encountered, the parser attempts to print meaningful error messages and to continue parsing.

The N14 specification [Spiby 91] provides a suggested grammar, however it suffers from various defects, such as unreferenced rules. This is probably because it was edited from previous drafts rather than being constructed anew. The grammar also sacrifices pedagogy for efficiency in many places.

The grammar used by Fed-X resembles the N14 grammar in spirit and language acceptance, but differs widely in some places. Since the N14 specification provides no mapping of rules to pages, an index was built for our own requirements. It is given in appendix A.

4.1.2 Second Pass: Reference Resolution

In the second pass, an attempt is made to resolve all names. An error message is generated for any reference to an undefined name and for any use of a name in an inappropriate context (e.g., an algorithm name as the type of a variable). Some checks are made on the consistency of the model during this pass. For example, one check ensures that every supertype of a given entity also lists the entity as a subtype, and vice versa. Also during this pass, warnings may be issued about names which are multiply defined in different scopes. Express has a hierarchical scoping mechanism to disambiguate these names, so that such overloading is allowed. In practice, however, Express models are mapped onto STEP physical files, which have no notion of a hierarchically scoped information model. When this "flattening out" of the model takes place, overloaded names may conflict; hence the need for these warnings about shadowed definitions.

Here is a cursory overview of the sequence of code that builds the Working Form. Occasional annotations are provided for clarity. The following steps all occur in the function:

```
EXPRESSpass_2                                                express.c
  SCHEMAresolve(model->schema);                              pass2.c
    SCOPEresolve_pass1(schema);                              pass2.c
      for each imported schema name (SCOPEget_imports)       scope.c
        replace name with actual schema
        SCHEMAresolve_pass1                                  pass2.c
          SCOPEresolve_pass1
      for each non-imported schema (SCOPEget_schemata)       scope.c
        SCHEMAresolve_pass1
      for each entity (SCOPEget_entities)
```

```
    ENTITYresolve_pass1(e)                                pass2.c
      convert supertype list from list of idents to entities
        EXPresolve_qualification
      convert subtype list from expression to list of entities
        (EXPresolve_subtype_expression(expr, entity, list);)
      resolve explicit, derived attributes (via VARresolve)
        VARresolve                                        variable.c
          attempt to resolve type by calling
          TYPEresolve(type_reference)                     type.c
            if type is an entity, generate an entity ref
            and attach to var by...
            OBJbecome(type,OBJcreate(Class_Entity_Type...
            attach entity to entity type by...
            ENT_TYPEput_entity(type,def)
      resolve unique

SCOPEresolve_pass2(schema);                               pass2.c
  for every symbol in scope
    report on shadowed decl's
  for every schema in scope
    SCHEMAresolve_pass2                                   pass2.c
      SCOPEresolve_pass2 (see above)
  for every type in scope
    resolve all references (TYPEresolve)                  pass2.c
  for every entity in scope
    ENTITYresolve_pass2                                   pass2.c
      for every supertype of this entity
        recursively call ENTITYresolve_pass2 (see above)
        tag each entity with count of attributes inc. inherited
        verify one of its subtypes is the entity
        add copy of supertype to entity's list of scopes
      for each subtype of this entity
        verify one of its supertypes is entity
      for each attribute
        for each supertype, if we can find a variable with
                          same name, report "overloaded"
        resolve inverses (VARresolve)
  for every algorithm in scope
    resolve references (ALGresolve)
      skip rules
      if function
        TYPEresolve
         if
      for each parameter
        VARresolve
          TYPEresolve
```

```
        if initializer, EXPresolve it
    SCOPEdefine_symbol
  for statement in body
    STMTresolve
      big case here on various statement types
```

EXPRESSpass_2 and SCHEMAresolve are very simple, just calling the routines indicated above.

4.1.3 Third Pass: Output Generation

After the first two passes have built and linked the in-memory Working Form, a third pass may be invoked to write the output. This pass can load several output modules in succession, so that several file representations of the Express input can be produced from a single parse. Alternatively, a specific module can be built into the translator, and this dynamic loading phase bypassed.

4.2 Working Form Data Structures

The Express Working Form is designed in object-oriented fashion, with one data abstraction corresponding to each concept in Express. Thus, there are abstractions which represent types, entities, variables (which include entity attributes and formal parameters, as well as local variables), expressions, statements, algorithms, and schemas. An additional concept which recurs in Express, and which is represented by a corresponding data abstraction, is that of a scope, which is, in effect, a symbol table. Algorithms, schemas, and entities all introduce their own local scopes.

In the following sections, we examine each abstraction in turn. Although each abstraction parallels the corresponding construct in Express quite closely, so that the descriptions below often seem to be echos [Spiby91], bear in mind that the objects described are actually the abstract data types of the Express Working Form.

4.2.1 Constant

The Constant abstraction represents symbolic constants. In the current implementation of the Working Form, constants appear only as elements of an enumerated type. A constant is named, and is marked with a type. The type of an enumeration constant simply points back at the enumeration of which it is an element. Each constant has a value, which can be of any C type (although it should be compatible with the type of the constant); in the case of enumeration constants, this value is always an integer.

Data private to constant objects is:

```
struct Constant {
    Type        type;
    Generic     value;
};
```

4.2.2 Type

The Type abstraction is used to represent Express types. Every type has a name, which is empty in many cases. When it is not, the type represents a type declaration, as in the `TYPE <id> = <type> END_TYPE` construct of Express. When the name is empty, the type represented appears within some other context - perhaps as the type of a function parameter or the base type of an aggregate. A type may have a list of constraints (WHERE rule) associated with it; these constraints restrict the legal values of the type.

Several classes of types are represented, including simple types (numeric, logical, string), enumerations, various aggregates, entity types, and select types. Several type classes are implicitly or explicitly subclasses of other type classes. Thus, boolean is a subtype of logical, and the various classes of aggregation types are subclasses of the general aggregate type. The attributes of a type depend on its class. Thus, integer, floating point, and string types may have a precision specification: an expression which constrains the number of significant digits or characters allowed in a value of the type. An enumeration type includes a list of the enumeration constants which are the allowable values for the type.

Every aggregate type (which may be an array, bag, list, set, or general aggregate) includes a base type, which indicates the type of objects which can be inserted into an instance of the aggregate type. In addition, an aggregate type may have lower and upper bounds. In the case of an array, these expressions indicate the first and last allowable index into the array. For other aggregate types, these expressions constrain the total number of objects which can (must) appear in an instantiation. If the bounds are not specified, they are assumed to be 0 and infinity, respectively. Two flags are also associated with each aggregate type, corresponding to the UNIQUE and OPTIONAL keywords in an Express aggregate definition. The 'unique' flag, if set, indicates that all elements of an aggregate must be unique among themselves. As this requirement already applies to a set, the flag is not valid for a set type. The 'optional' flag, which applies only to an array type, indicates that all positions in the array need not be filled in a valid instantiation of the type - the array may contain null entries.

An entity type is simply one or more entities packaged as a type. No further information is added beyond the entity definitions themselves. Entity types exist to allow entity instantiations to be represented (c.f. STEP Working Form [Clark90b]), and to provide a clean mechanism for recognizing entity names in type contexts.

A select type consists of a list of selectable types. An instantiation of any of these selections is a valid instantiation of the select type. In this sense, the select is similar to the C language `union` construct and the Pascal variant `record`. In Express, the list of selections may only include references to named types.

There are two type classes, generic and number, which are distinguished by the fact that the corresponding Express types (GENERIC and NUMBER, respectively) cannot be instantiated. These can only be used as types of formal parameters to algorithms, where an actual parameter will provide an instantiation of a more specific type at run time.

A special type class is used to represent type references. These are (possibly qualified) references which appear in type contexts, but which are not yet resolved to a particular type. In normal operation under the control of Fed-X, they are replaced during the second pass by appropriate type constructs. A type reference uses an expression (see section 4.2.5) to record the qualified type name it represents. The components of this expression are identifiers, and they are combined into binary expressions with the dot operator.

There are several type constants available. These constants can be used to avoid creating multiple copies of some common types, including generic, integer, unbounded generic set, logical, etc.

Data private to type objects is:

```
struct Type {
    Linked_List where;
    Type original_type;
};

struct Aggregate_Type {
    Type        base_type;
    Expression  lower;
    Expression  upper;
    short       flags;
};

 struct Composed_Type {
    Linked_List list;
};

struct Sized_Type {
    Expression  size;
    Boolean     fixed;
};

struct Type_Reference {
    Expression  name;
};
```

4.2.3 Entity

The Entity abstraction represents Express entity declarations. Every entity consists of a name, and (possibly empty) lists of attributes, subtypes, and supertypes. In addition, an entity includes a boolean expression which describes the relationships among its various subtypes. The attributes are represented as variables which are defined in the local scope of the entity. The sub- and supertypes are themselves entities.

In order to give a hierarchical structure to an Express model, entities are arranged in a class hierarchy, as in the Object-Oriented world. This hierarchy is defined by the subclass and superclass lists of its component entities. As specified by Express, the class hierarchy provides for conjunctive as well as disjunctive subclassing: `foo SUPERTYPE OF (bar AND blat)` means that any instance of `foo` is also an instance both of `bar` and of `blat`, while `foo SUPERTYPE OF ONEOF(bar, blat, blit)` represents standard inheritance, in which an instance of `foo` is also either an instance of `bar` or an instance of `blat` or an instance of `blit`.

An entity may also include a list of uniqueness sets (from the Express UNIQUE rule) and a list of constraints (from the Express WHERE clause). Each uniqueness set is a list of attributes whose values, when taken together, must uniquely identify a particular instance of the entity. The constraints, if any, are expressions which compute logical results. Each must evaluate to `true` in a valid product model. These constraints can apply to individual instantiations of the entity as well as to the collection of all instances of the entity.

Since one possible way of looking at an entity class is as the collection of its instances, provision is made in this abstraction for maintaining this collection. Thus, it is possible to add instances to an entity, or to retrieve a list of all of the instances of an entity. This mechanism is used by the STEP Working Form.

Data private to entity objects is:

```
struct Entity {
    Linked_List supertypes; /* list of supertypes */
    Linked_List subtypes;   /* simple list of */
           /* subtypes, useful for simple lookup */
    Expression  subtype_expression; /* DAG of */
          /* subtypes, with complete information */
           /* including, OR, AND, and ONEOF */
    Linked_List attributes; /* explicit attributes */
    int         inheritance;
    int         attribute_count;
    Linked_List unique;
    Linked_List constraints;
    Linked_List instances;
    int         mark;
    Boolean     abstract; /* is this an abstract */
           /* supertype? */
};
```

4.2.4 Variable

The Variable abstraction is used to represent entity attributes and formal parameters to algorithms as well as local variables in a scope. A variable consists of a name, a type, a reference, an offset, and some flags. A variable may optionally have an initializer, which is an expression used to specify an initial value for the variable.

The reference of a variable is the original name of an entity object in the schema from which it has been used or referenced.

A variable's offset indicates its position in a storage block. Thus, the offset of a local variable is its offset into the data space of the scope in which it is defined, while the offset of an entity attribute is its position relative to the first attribute of the entity. It is important to realize that, in the latter case, this offset is not sufficient to locate the attribute in an instantiation of the entity, since this total offset cannot be determined from the entity definition alone. To see this, consider entities `A` and `B`, each with a single attribute (call these `aa` and `bb`, respectively) The offset to `bb` in an instantiation of `B` is 0. But now suppose there is a third entity class, `C`, which inherits from both `A` and `B`, in that order. Then the offset to `bb` in an instance of `C` must be 1, even though `bb` is inherited from `B`, where its offset was 0. Thus, a variable's offset may not be a useful piece of information by itself.

The 'optional' flag is used with entity attributes, and indicates that the attribute need not have a value in a valid instantiation of the entity. A variable representing an entity attribute can also be marked 'derived,' indicating that the attribute value is always derived from the values of other attributes, and can never be specified by a user. The 'variable' flag, meaningful for formal parameters, indicates that the parameter is to be passed by reference, i.e., it can be modified by the receiver.

Data private to variable objects is:

```
struct Variable {
    Type                type;
    Expression          initializer;
    Expression          reference;  /* true name */
       /* in use'd or reference'd object */
    int                 offset;
    short               flags;
   Symbol           inverse;  /* remote entity ref */
        /* entity related by this inverse */
        /* relationship */
};
```

4.2.5 Expression

Expression is one of the more complex abstractions, simply because of the wide variety of expressions found in Express. There are five basic classes of Expressions, some of which are further divided into conceptual subclasses: literals (including integer, logical, real, set, and string literals), identifiers, operations (including unary operations and binary operations), function calls, and queries. Every expression includes a type, which is the type of the value it computes. Although this type is intended to be computed automatically, it currently is neither computed nor used by the Working Form code, except in the case of a literal. In this case, the type is an implied part of the definition of the literal's class.

Literal classes exist for most of the concrete simple types (as opposed to the abstract simple types, NUMBER and GENERIC). Boolean literals do not exist in Express; they are interpreted as logical literals instead. There may also be set literals (notably, the empty set). There are several literal expression constants representing, for example, zero, infinity, and the empty set.

An identifier expression represents a reference to a variable. It consists simply of the variable referenced. (Simple) identifier expressions can be composed using (binary) field reference expressions to form the complex qualified identifiers which Express provides.

An operation expression includes one (unary operation) or two (binary operation) operands, which are themselves expressions, and an operator, such as addition, negation, array indexing, or attribute extraction. All of the operations of Express are supported.

A function call is composed of an algorithm (which may not be a procedure) and a list of actual parameters to the algorithm. The actual parameters to the function call are themselves expressions. Entity subtype expressions (see section 4.2.3) make use of a closely related expression class, the oneof expression, which consists of a list of entity references.

A query expression represents the set-theoretic "set of all *x* in *X* such that …" construct. It consists of a domain set (*X*), a temporary identifier which represents each element of the domain successively (*x*), and a list of conditions to apply to each *x*. The result computed is a set containing all of the values of *x* which satisfy the constraints.

Data private to expression objects is:

```
struct Ary_Expression {
    Op_Code     op_code;
    Expression  op1;
};

struct Ternary_Expression {
    Expression op2;
    Expression op3;
};

struct Query {
    Variable    identifier;
    Expression  fromSet;
    Expression  discriminant;
    Scope       local_scope;
};
```

4.2.6 Statement

The Statement abstraction is used to represent the wide variety of statements which occur in Express. There are many classes of statements, including assignments, case statements, conditionals, loops, procedure calls, returns, and with statements. A series of statements may be combined into a single compound statement.

An assignment statement consists of a left-hand-side expression, which must be assignable (this limits the expression to a possibly qualified identifier, although the restriction currently is not enforced by the Working Form), and a right-hand-side expression, computing the value to be assigned.

A CASE statement is, as in Pascal, a multi-branch conditional. It contains an expression (the case selector) and a list of branches. Each branch is a case item, represented by the Case Item abstraction. A case item consists of a list of one or more values against which the selector will be compared and a statement to be executed if the selector matches one of these values.

The looping construct in Express is quite general, combining the functionality of the `repeat .. until`, `while .. do`, and `for` loops of modern programming languages. An Express loop consists of a controlled statement (the body of the loop) and a list of loop controls. There are three classes of loop control: increment (corresponding to a `FOR` loop), until, and while. The first consists of a controlling identifier expression, initial and terminal expressions, and an optional increment expression, which defaults to 1 if not present. The controlling identifier takes on successive values from the initial to the terminal expressions, and is incremented by the increment expression on each iteration. An until control consists of a single expression (which must compute a boolean result); it causes the loop to terminate when this expression evaluates to `true`. Similarly, a while control causes the loop to terminate as soon as its single expression evaluates to `false`.

A procedure call is very much like a function call, with the exception that the algorithm is expected to be a procedure, rather than a function or rule. The procedure call statement includes a list of expressions, representing the actual parameters to the call.

A RETURN statement is the mechanism by which a function reports a value to its caller. It contains a single expression, which computes the value to be returned.

A simple statement is one which consists of a single keyword. There are two such statements in Express: ESCAPE and SKIP. No statement class is provided for simple statements; rather, they are represented by statement constants, unique instances of the Statement abstraction itself.

Finally, Express includes the WITH statement, which resembles Pascal's construct of the same name. It includes a controlled statement and a controlling expression which provides (optional) partial qualification to any expression in this statement. If a name in the controlled statement cannot be resolved, an attempt is made to resolve the name as if it were prepended with the controlling expression. The Working Form currently does not attempt to acknowledge WITH statements when resolving identifiers.

Data private to statement objects is:

```
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 {
    Linked_List controls;
    Statement   statement;
};

struct Procedure_Call {
    Procedure   procedure;
    Linked_List parameters;
};

struct Return_Statement {
    Expression  value;
};
```

4.2.7 Algorithm

Express functions, procedures, and rules are each represented by a subclass of the Algorithm abstraction. A procedure is simply a sequence of statements. A function is a sequence of statements which computes a result and returns it to the caller. A rule is a special kind of function whose result is always a boolean (logical). A rule also has slightly different scoping rules than other algorithms, to allow it to manipulate entity classes as well as instances.

Any algorithm consists of a name, a list of formal parameters (which are represented by variables), and a list of statements forming the body of the algorithm. In addition, a function has a return type. A rule implicitly returns a logical value. This value is computed by a list of constraints (WHERE clause), which is evaluated after the statements which form the rule body.

Data private to algorithm objects is:

```
struct Algorithm {
    Linked_List parameters;
    Linked_List body;
};

struct Function {
    Type        return_type;
};

struct Rule {
    Linked_List where;
};
```

4.2.8 Scope

All scoping and symbol table functionality are managed by the Scope abstraction. A local scope is established by each algorithm, schema, and entity. For this reason, each of these abstractions is considered to be a subclass of scope, thereby inheriting all of its functionality. Pascal-like hierarchical scoping and inheritance are implemented by having each scope point to its immediate containing scope(s), if any. For example, an algorithm's local scope points to the scope in which the algorithm is defined; an entity's scope may have several parents: the scope in which the entity is defined, and all of the supertype entity scopes. In its role as a symbol table, a scope includes definitions of various names as entities, types, variables, algorithms, constants, and schemas.

A scope can be queried for its definition of a particular symbol. If the scope does not itself define the symbol, its superscopes are in turn queried, and so forth. If no definition can be found, the query fails.

Data private to scope objects is:

```
typedef struct Express {
    FILE*       file;
    Dictionary  schemas;
} *Express;

struct Scope {
    Linked_List parents;
    Dictionary  symbol_table;
    Dictionary  references;
```

```
        Linked_List use;
        int         last_search;
        Boolean     resolved;
    };
```

4.2.9 Schema, Schemas

The Schema abstraction represents the Express construct of the same name, which is, in effect, a named scope. Most operations of interest are performed on the scope. There is no data private to schema objects.

The Schemas abstraction represents a set of Schemas. The object produced by the first two passes of Fed-X is such a set, which ultimately contains all of the definitions found in the source file. There is no data private to schemas.

4.3 Class Hierarchy

In order to get a better idea of how the objects and classes fit together, this section presents a class hierarchy. The left column defines the class names and the hierarchy – the data local to each class object is defined in the right column. The hierarchy is presented so that each class is defined to be a superclass of the first class above it that is exdented to a different position.

```
Typename                                      sizeof
(all prefixed by "Class_")

Null
    Construct                                 struct Construct
        Case_Item                             struct Case_item
        Expression                            Type
            Ary_Expression                    struct Ary_Expression
                Unary_Expression
                Binary_Expression             Expression
                Ternary_Expression            struct Ternary_Expression
            Function_Call                     Algorithm
            Identifier                        Variable
            Literal
                Aggregate_Literal             Linked_List
                Binary_Literal                Binary
                Integer_Literal               Integer
                Logical_Literal               Logical
                Real_Literal                  Real
                String_Literal                String
            One_Of_Expression                 Linked_List
            Query                             struct Query
        Loop_Control                          struct Loop_Control
            Increment_Control                 struct Increment_Control
            Conditional_Control
            Until_Control
```

```
        While_Control
    Statement
        Assignment                      struct Assignment
        Case_Statement                  struct Case_Statement
        Compound_Statement              struct Compound_Statement
        Conditional                     struct Conditional
        Loop                            struct Loop
        Procedure_Call                  struct Procedure_Call
        Return_Statement                struct Return_Statement
        With_Statement                  struct With_Statement
Dictionary                              struct Dictionary
Linked_List                             struct Linked_List
    Stack
Symbol                                  struct Symbol
    Constant                            struct Constant
    Instance (step only)                struct Instance
    Scope                               struct Scope
        Algorithm                       struct Algorithm
            Function                    struct Function
            Rule                        struct Rule
            Procedure
        Entity                          struct Entity
        Schema
    Type                                struct Type
        Aggregate_Type                  struct Aggregate_Type
            Array_Type
            Bag_Type
                Set_Type
            List_Type
        Type_Reference                  struct Type_Reference
        Sized_Type                      struct Sized_Type
            Binary_Type
            Integer_Type
            Real_Type
            String_Type
        Number_Type
        Logical_Type
            Boolean_Type
        Generic_Type
        Composed_Type                   struct Composed_Type
            Entity_Type
            Enumeration_Type
            Select_Type
    Variable                            struct Variable
```

You will note that there is no multiple inheritance. This is a serious drawback and prevents certain N14 constructs such as a type and enumeration with the same name. In order to provide this, you would need to be able to have multiple symbols in a single scope with the same name (or multiple scopes at the same level). The cleanest solution would be to add a scope to enumeration types, but this is currently impossible since types and scopes cannot inherit from one another due to the lack of multiple inheritance.

It is our suspicion that adding multiple inheritance to the current object implementation would greatly decrease the operating speed even for Express files that do not make use of this capability.

4.4 Object Processing

4.4.1 Use, Reference

The USE and REFERENCE constructs in the N14 version of Express permits a schema to access definitions in other schemas. The two constructs differ in the type of access. USE treats definitions in the other schema as local. REFERENCE permits entities in the local schema to reference items in the other schema but the definitions are not considered local.

The processing of the USE construct is as follows. The parser returns a list containing the schema name as the first element followed by an optional list of expressions. The expressions contain the name of the foreign entity and an optional local name. If the foreign schema is not yet resolved, then SCHEMAresolve is called on it. The foreign schema must be resolved since it may have a USE statement which brings in additional entity definitions and so on. Schema resolution will expand any USE constructs into their equivalent entities. After any USE'd schemas are resolved, the list of foreign entities is traversed and they are copied into the local schema, renaming as necessary.

The processing of the REFERENCE construct is somewhat different. The structure returned from the parser is the same as with USE. However, instead of being added to the schema's symbol table, a separate dictionary is used to contain references to foreign items. The parser adds references to the dictionary. During REFERENCEresolve a copy of the dictionary is traversed and pointers to the objects being referenced are added. If an object is renamed with a local name, a copy of the object is made and that pointer is added to the reference dictionary. This is because the referenced object does not know it has been referenced and only knows its original name. Thus, referenced objects are not local to a schema, i.e., they would not appear on the list of entities for a schema. The reference dictionary is searched by SCOPElookup so that a referenced item is known to the schema but it is not considered local.

4.5 Missing Features

While Fed-X accepts almost all of the syntactic constructs of Express, the Working Form does not yet represent as many of them; nor does it observe all of those which it represents. In particular, `constant` is syntactically observed but semantically ignored.

Although the full type system of Express is represented in the Working Form, type derivations are not performed. It is theoretically possible to assign a type to any expression on the basis of the operator and operands (or by looking up a function in the symbol table), but this functionality is not yet implemented. Thus, erroneous messages about type mismatches are sometimes produced simply because type information about certain expressions is not available.

Express implicitly suggests an evaluation environment yet it does not define one. Fed-X makes occasional attempts to mitigate this deficiency, however a rigorous treatment is impossible without further specificiation.

Due to problems with the Express language definition, qualified identifiers may not always be interpreted properly. Problems are particularly common when dealing with enumeration identifiers. Similarly, Express allows a subtype entity to redefine an attribute which it inherits from a supertype. The effect of this redefinition on scoping remains an open issue, and so Fed-X currently does not allow it.

Fed-X responds robustly to semantic errors. Syntax error recovery is somewhat more haphazard.

Comments are discarded during lexical analysis and so have no chance of being recorded by the parser.

5 Conclusion

Although the Express Working Form in its current state is sufficient for current applications, it is only a matter of time before some of the missing features are required. In addition, Express is still evolving, and the software must continue to change with the language.

Fed-X has proven to be an effective tool for the creation of schema-independent applications based on STEP. Translators using each of the output modules distributed with the Express Working Form are in common use at NIST. Fed-X is also part of the toolkit distributed by PDES, Inc.

A Cross-Reference to N14 Rules

Rules below 117 are omitted, since they are trivial mappings.

Rule #	Page #(s)	Rule name
117	72	add_like_op
118	24	binary_literal
119	24	bit
120	25	character
121	18	digit
122	24	digits
123	24	integer_literal
124	19	letter
125	25	logical_literal
126	20	lparen_not_star
127	72	multiplication_like_op
128	20	not_lparen_star
129	20	not_paren_star
130	19-20	not_paren_star_special
131	20	not_rparen
132	20	not_star
133	24	real_literal
134	72	rel_op
135	24	sign
136		simple_id
137	19	special
138	20	star_not_rparen
139	25	string_literal
140	89,97	actual_parameter_list
141	90	aggregate_initializer
142	86	aggregate_source
143	60	aggregate_type
144	31-34,40,45-46,58-60,62-64	aggregation_types
145	63-64	algorithm_head
146	93	alias_id
148	93	alias_stmt
147		alias_ref
149	31	array_type
150	94	assignment_stmt
151	45-46	attribute_decl
152	45-46,48	attribute_id
153	45-46,49,93-94	attribute_qualifier
154		attribute_ref
155	32	bag_types
156	31-34,45-46,58-60,62-64	base_type
157	28	binary_type
158	28	boolean_type
159	31-34,48,62,77,98	bound_1
160	31-34,48,62,77,98	bound_2
161	31-34,48,62	bound_spec
163	89	built_in_function
164	97	built_in_procedure
165	95	case_action

166	95	case_label
167	95	case_stmt
168	96	compound_stmt
169	62	conformant_aggregate
170	62	conformat_type
171	58	constant_decl
172	58	constant_body
173	72	constant_factor
174	58	constant_id
175		constant_ref
176	58,63-64	declaration
177	46	derived_attr
179	90	element
180	20	embedded_remark
181	41	entity_block
182	41	entity_body
183	41	entity_head
184	34,41,66-67	entity_id
185	91	entity_init
186	66	entity_or_rename
187	34	entity_ref
188	36,72	enumeration_id
189	72	enumeration_ref
190	36	enumeration_type
191	96	escape_stmt
192	45	explicit_attr
193	72	expression
194	72	factor
195	59,63-64	formal_parameter
196	63	function_block
197	89	function_call
198	63	function_head
199	63,67	function_id
200		function_ref
201	72,93-94	general_ref
202	61	generic_type
203	45-46,49,93-94	group_qualifier
204	97	if_stmt
205	98	increment
206	98	increment_control
207	79,81,83,93-94	index
208	79,81,83,93-94	index_qualifier
209	72	initializer
210	27	integer_type
211	58,66	interface_specification
212	77	interval
213	77	interval_item
214	77	interval_op
215	48	inverse_attr
216	(example on p.48 hints at syntax)	inverse_clause
217	49-50	label
218	49	labelled_attrib_list
219	50	labelled_expression
220	33	list_type

221	23	literal
222	62	local_decl
223	62	local_variable
224	86	logical_expression
225	28	logical_type
226	31-34,37,45-46,58-60,62-64	named_types
227	93	null_stmt
228	26	number_type
229	52	one_of
230	89,97	parameter
231	59,63-64	parameter_id
232		parameter_ref
233	59-60,62-64	parameter_type
234	27	precision_spec
235	64	procedure_block
236	97	procedure_call_stmt
237	64	procedure_head
238	64,67	procedure_id
239		procedure_ref
240	72,93-94	qualifier
241	72	qualifiable_factor
242	45-46,49	qualified_attribute
243	86	query_expression
244	27	real_type
245	67	reference_clause
246	45-46,49	referenced_attribute
247	72	rel_op_extended
248		remark
249	67	rename_id
250	97	repeat_control
251	97	repeat_stmt
252	90	repetition
253	67	resource_or_rename
254	67	resource_ref
255	100	return_stmt
256	64	rule_block
257	64	rule_head
258	64	rule_id
259	58	schema_block
260	58	schema_body
261	58	schema_id
262		schema_ref
263	37	select_type
264	95	selector
265	34	set_type
266	72	simple_expression
267	72	simple_factor
268	31-34,40,45-46,58-60,62-64	simple_types
269	100	skip_stmt
270	93	stmt
271	29	string_type
272	79,81,93-94	subcomponent_qualifier
273	52	subsuper
274	91	subsuper_init

275	52	subtype_declaration
276	52	supertype_declaration
277	52	supertype_expression
278	52	supertype_factor
279	20	tail_remark
280	72	term
281	40	type_decl
282	35,40,67	type_id
283	60-61	type_label
284	35	type_ref
285	72	unary_op
286	40	underlying_type
287	49	unique_clause
288	99	until_control
289	66	use_clause
290	62,86,98	variable_id
291		variable_ref
292	50	where_clause
293	99	while_control
294	28-29	width

B References

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

[Clark90c] Clark, S.N., Libes, D., NIST Express Working Form Programmer's Reference, NISTIR 4814, National Institute of Standards and Technology, Gaithersburg, MD, September 1990

[Clark90d] Clark, S.N., QDES User's Guide, NISTIR 4361, National Institute of Standards and Technology, Gaithersburg, MD, June 1990

[Clark90e] Clark, S.N., QDES Administrative Guide, NISTIR 4334, National Institute of Standards and Technology, Gaithersburg, MD, May 1990

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

[Metz89] Metz, W.P., and K.C. Morris, Translation of an Express Schema into SQL, PDES Inc. internal document, November 1989

[Morris90] Morris, K.C., Translating Express to SQL: A User's Guide, NISTIR 4341, National Institute of Standards and Technology, Gaithersburg, MD, May 1990

[Morris91] Morris, K.C., McLay M. Carr, P. J., Validation Testing System Requirements, NISTIR 4636, National Institute of Standards and Technology, Gaithersburg, MD, September 1991.

[Perlotto89] Perlotto, K. L., The Use of GMAP Software as a PDES Environment in the National PDES Testbed Project, NISTIR 89-4117, National Institute of Standards and Technology, Gaithersburg, MD, June 1989

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