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

NISTIR 4815

National PDES Testbed
Report Series

The NIST PDES Toolkit: Technical Fundamentals

Revised April, 1992

Stephen Nowland Clark
Don Libes

NATIONAL PDES TESTBED™

April 3, 1992

NIST

NISTIR 4815

National PDES Testbed
Report Series

Sponsored by:

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

The NIST PDES Toolkit: Technical Fundamentals

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

Table of Contents

The NIST PDES Toolkit: Technical Fundamentals

Stephen Nowland Clark
Don Libes[1]

1 Introduction

The NIST PDES Toolkit [Clark90a] provides a set of software tools for manipulating Express [Part11] information models and STEP [Part21] product models. It is a research-oriented toolkit, intended for use in a research and testing environment. This document gives a technical introduction to the Toolkit, providing a programmer with basic knowledge of its structure. Also covered are the mechanics of building Toolkit-based applications.

In addition to describing the structure and usage of the Toolkit, we describe three fundamental code libraries which it includes. The most significant of these, `libmisc.a`, contains various modules of general utility, including such abstractions as linked lists and hash tables. `libbison.a` is a small library containing support routines and global variables for the Toolkit's parsers. The third library, `libdyna.a`, provides a dynamic (run-time) loading facility for `a.out` format object files under BSD 4.2 Unix and its derivates.

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 that describe various aspects of the Toolkit. An overview of the Toolkit is provided in [Clark90a], along with references to the other documents in the set.

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

For further information on the Toolkit, or to obtain a copy of the software, use the attached order form.

1.2 Development Environment and Tools

The NIST PDES Toolkit is implemented in ANSI Standard C [ANSI89]. All software has been developed on Sun Microsystems Sun-3™ and Sun-4™ workstations running the Unix™ operating system. The parsers are written in Yacc and Lex, the standard Unix™ languages for generating parsers and lexical analyzers. The development compiler for the Toolkit is GCC, the GNU Project's[1] C compiler, and the parsers are compiled by Bison, the GNU Project's implementation of Yacc. The lexical analyzers are compiled by Flex[2], a Public Domain implementation of Lex. Rules for building the Toolkit are specified using the Unix Make utility.

2 Structure of the Toolkit

The NIST PDES Toolkit consists of the Express [Clark90b] and STEP [Clark90c] Working Forms, and several applications which make use of these Working Forms. The Working Forms reside in object libraries, which can be linked into applications which use them.

In addition to the various design and usage documents referenced elsewhere, technical reference manuals on the Express [Clark90d] and STEP [Clark90e] Working Forms are also available.

The Toolkit distribution may be installed anywhere on a particular filesystem. For simplicity, the root directory of the distribution is referred to as `~pdes/` throughout the technical documentation. This root directory contains a number of subdirectories of interest, which we now describe. Brief descriptions also appear in `~pdes/README`.

The directories `~pdes/bin/`, `~pdes/include/`, and `~pdes/lib/` contain, respectively, Toolkit application binaries, C header (`.h`) files for the Toolkit libary modules, and the Toolkit object libraries (`.a` files) themselves. PostScript® and/or ASCII versions of the Toolkit documentation can be found in `~pdes/docs/`. Various utility tools which are needed to build or run pieces of the Toolkit are in `~pdes/etc/`. Finally, the directory `~pdes/src/` contains the source code for the Toolkit libraries. There is a separate subdirectory for each library. This `src/` directory also includes a subdirectory called `Template/`, which includes a `Makefile` template which can be used as a model when building new applications which use the Toolkit.

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

2.1 Conventions

Each Working Form is composed of a number of data abstractions. Each of these abstractions is implemented as a separate module. Modules share only their interface specifications with other modules. For example, consider a module called Foo, composed of two C source files, `foo.c` and `foo.h`. The former contains the body of the module, including all non-inlined functions. The latter contains function prototypes for the module, as well as all type and macro definitions. In addition, global variables are defined in `foo.h`. These declarations are made using the following macros:

```
#ifdef FOO_C
# define GLOBAL
# define INITIALLY(v) = v/* historical */
# define INITIALLY1(v1) = {v1}
# define INITIALLY2(v1, v2) = {v1, v2}
# define INITIALLY3(v1, v2, v3) = {v1, v2, v3}
/* the rest (up to 10) omitted */
#else
# define GLOBAL extern
# define INITIALLY(v)     /* historical */
# define INITIALLY1(v1)
# define INITIALLY2(v1, v2)
# define INITIALLY3(v1, v2, v3)
/* the rest omitted */
#endif FOO_C
GLOBAL int FOO_GLOBAL_INT INITIALLY1(4);
```

This allows the same declarations to be used both in `foo.c` and in other modules which use it: when `foo.h` is included in `foo.c`, `FOO_GLOBAL` has storage declared and is initialized. When `foo.h` is included elsewhere, an uninitialized `extern` declaration is produced.

Finally, `foo.h` contains inline function definitions. If the C compiler supports inline functions (as GCC does), these are declared `static inline` in every module which includes `foo.h`, including `foo.c` itself. Otherwise, they are undefined except when included in `foo.c`, when they are compiled as ordinary functions.

The type defined by module Foo is named `Foo`. Access functions are named as `FOOfunction()`; this function prefix is abbreviated for longer abstraction names, so that access functions for type `Foolhardy_Bartender` might be of the form `FOO_BARfunction()`. Some functions may be implemented as macros; they are not distinguished typographically from other functions, and are guaranteed not to have unpleasant side effects like evaluating arguments more than once. These macros are thus virtually indistinguishable from functions. Functions which are intended for internal use only are named `FOO_function()`, and are usually `static` as well, unless this is not possible. Global variables are often named `FOO_variable`; most enumeration identifiers and constants are named `FOO_CONSTANT`. For example, every abstraction defines a constant `FOO_NULL`, which represents an empty or missing value of the type.

There are, of course, exceptions to all of these rules. The global variable and enumeration identifier rules are the most frequently broken. Library modules which were developed before all of the rules solidified, as well as components which were not developed locally by the Toolkit project, tend to stretch the rules more than the actual Working Form modules, which have tended to be more dynamic later in the project.

2.2 Object-Oriented Framework Modules: `Class` and `Object`

Most of the Working Form abstractions are implemented on top of the Class and Object modules defined in `libmisc`. Together, these modules provide a simple object-oriented framework on which various abstractions can be built. The Class module manipulates representations of classes in the object-oriented sense, defining management operations for classes of values and representing sub- and supertype relationships between these classes. The Object module supports instantiation of these classes. It actually performs the management operations specified by an Object's class, and interprets the class hierarchy defined by a set of sub- and supertype relationships between classes.

2.3 A Note on Memory Management and Garbage Collection

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

3 Compiling With the Toolkit: The `Makefile` Template

The file `~pdes/src/Template/Makefile` (reproduced in Appendix B) in the Toolkit distribution is a skeletal `Makefile` which can be configured to build a wide variety of applications which use one or both of the Working Forms. This `Makefile` uses a number of macros and rules which are defined in `~pdes/include/make_rules`. It assumes that the source code for the application to be built resides in `~pdes/src/<appl>/`.

The following sections discuss the various classes of applications which can be built, and the appropriate configuration for the `Makefile`. There are several macros defined in the `Makefile` which are used to configure an application. The most important, in that it determines how the application will use the Working Form(s), is called `LIBS`. This macro is defined in the section of the `Makefile` entitled "Library Selection," which contains a number of possible definitions. Each option is preceded by a comment describing the situation in which it is appropriate; exactly one definition should be uncommented. Next, two options are given for the `CFLAGS` macro: one for STEP applications and one for applications which use only Express. This is not a necessary distinction, since things will always build correctly with the former definition; it is provided for the benefit of those who (like the author) prefer possibly inordinate neatness.

There are two macros which can be used to specify the auxiliary object (`.o`) files and libraries (`.a` files) required by the application. Object files should be named in the `OFILES` macro; several sample definitions are included for the applications provided with the toolkit. The macro `MYLIBS` can contain any additional libraries which are required by the application.

In addition to the fundamental configuration options discussed above, there are several more macros which can be used to make "cosmetic" changes to an application. At the top of the `Makefile` is a macro called `CC`, which selects the C compiler to be used. Common options are `/bin/cc` (the vendor-supplied compiler under Unix) and `$(GCC)`, which should point to the Gnu Project's C Compiler. The contents of `MY_CFLAGS` are passed to every invocation of `$(CC)`; this is the place to add debugging and/or optimization flags, for example. The default rule for compiling `.c` files (from `~pdes/include/make_rules`) probably should not be changed, but it appears in the template to provide a hook for unforeseen requirements. Finally, toward the end of the `Makefile` is a macro called `PROG`. This macro holds the name of the executable which will be built.

The `Makefile` provides three targets: `$(PROG)` rebuilds the application from scratch, as necessary. The `relink` target assumes that all `.o` files and libraries are up-to-date, and simply relinks the application. This is useful, for example, when one of the Toolkit libraries has been rebuilt, but the application source itself has not been changed. The last target, `clean`, removes `$(PROG)` and `$(OFILES)`. This rule may be modified for a particular application. Any additional rules which are required to build the application can be added at the end of the `Makefile`.

3.1 STEPparse STEP Translators

The first class of applications which we examine are the STEP translators. These programs parse a STEP Physical File into the STEP Working Form and then invoke one or more report generators which traverse these data structures and produce output files containing some or all of the product model represented in a different format.

The Make macro `$(STEP_LIBS)` expands to list all of the libraries needed to create a STEP translator. These include: `libstep.a` and `libexpress.a`, the STEP and Express Working Form libraries; `libmisc.a` and `libbison.a`; and `libl.a`, which provides support for lexical analyzers produced by Lex. The first four are located in `~pdes/lib/`, while `libl.a` is normally found in `/usr/lib/`. The order in which these libraries are listed is significant: `libstep` and `libexpress` both include definitions of `main()`, the standard entry point to a C program. To build a STEP translator, the first definition of `main()` which the linker finds must be the STEPparse driver, which is in `libstep.a`.

In addition to these libraries, two more pieces of code are needed to build a complete translator: a report generator and a linkage mechanism for this report generator. The latter is needed because the translator can load its report generator(s) in either of two ways: it can load a specific one at compile time, or it can dynamically load one or more at run time. The dynamic approach has at least two major advantages: It allows multiple output formats to be produced by a single executable; and it allows several reports to be created by a single run of the translator, so that the parsing phase need only be executed once. This approach has the unfortunate disadvantage that it is (currently) only available under BSD 4.2 Unix and its derivates; it is therefore considered optional in the current incarnation of the Toolkit.

In the library selection section of the `Makefile`, the first two options are alternate definitions of `LIBS` for building a STEP translator. The first is for a translator with a single, statically bound report generator. Since the static linkage facility is included in `libstep.a`, the linkage mechanism is not explicitly listed. The second alternative, for a translator with dynamically bound report generators, selects `~pdes/lib/step_dynamic.o` to provide the run-time linking mechanism. In addition, it adds `libdyna.a` to the link.

If a dynamically loading translator is being built, then no report generator object file should be listed in the `OFILES` macro, since the report generator will be selected at run time. The first sample definition of `OFILES` is appropriate here. If a report generator is being loaded at build time, then any object files which are needed to implement it should be listed in this macro.

3.2 Fed-X Express Translators

The process of configuring the `Makefile` to build an Express translator is similar to that described for STEP translators. The `$(EXP_LIBS)` macro expands to the list of libraries needed to build a Fed-X translator; these include the same libraries listed in `$(STEP_LIBS)`, with the exception of `libstep.a`. Again, there are two possible definitions of `LIBS`. The first selects a build-time (static) linkage (which is included in `libexpress.a`); the second adds `~pdes/lib/express_dynamic.o` and `-ldyna` for run-time (dynamic) linkage.

As in the case of a STEP translator, a dynamically bound Express translator requires no object files in `$(OFILES)`, while a statically bound translator expects to find the report generator in this macro. The first sample definition of `OFILES` can again be used in the former case.

3.3 Other Applications

We now turn to the more free-form applications which might make use of the Express and/or STEP working forms. A notable difference between these applications and the translators is that the programmer must define the flow of control, by providing `main()`. As mentioned above, both the STEP library and the Express library include definitions of `main()` which are used to drive the respective translators; source code for these can be found in `~pdes/src/step/step.c` and `~pdes/src/express/fedex.c`, respectively. These might serve as useful starting points for other applications. In general, the first two passes of the Express parser (`EXPRESSpass_1()` and `EXPRESSpass_2()`) will have to be run in any application, unless a conceptual schema is to be built by hand. `EXPRESSpass_3()` invokes a report generator via the selected linkage mechanism. The call which invokes the STEP parser is `STEPparse()`; this is the simplest way of building an instantiated STEP model. A STEP report generator is invoked by calling `STEPreport()`; unfortunately, Express and STEP report generators and associated linkages currently cannot coexist in a single executable. This restriction is not due to anything fundamental, and so may disappear should there be sufficient demand.

Assume for the moment that no STEP or Express report generators are needed. In this case, it is quite simple to configure the `Makefile` to use one or both of the Working Form(s): First, set `LIBS` to either `$(STEP_LIBS)` or `$(EXP_LIBS)`, depending on which Working Form is needed (remember that the former includes the latter, so that it is never necessary to use both macros at once). These are the last two sample definitions in the library selection section. Next, in `OFILES` and `MY_LIBS` list the object files and libraries which the application uses. Bear in mind that the application's `main()` must appear in `$(OFILES)` in order to override the default one which will otherwise be found in one of the Working Form libraries. Finally, be sure to set `PROG` to the name of the application which will be built.

We now return to the problem of an application which will use a STEP or Express report generator without being just a translator. A `main()` must be provided for this application and included in the `OFILES` macro, just as in the previous case. What gets messy is the library selection. To use a Fed-X report generator in an application which uses only the Express working form, or to use a STEPparse report generator in a STEP application, just select the appropriate `LIBS` macro for a translator with the same report generator linkage, one of the first four sample definitions. To build an application which produces Fed-X reports while using the STEP working form, choose either the static or the dynamic binding option from the section "STEP applications with Express report generators" in the `Makefile` template. This will select the full set of STEP libraries, and pull in the specified Fed-X output linkage.

4 Basic Libraries

This section discusses the three basic libraries in the Toolkit. Portions of the libraries are discussed in varying levels of detail, according to the level of code reuse from other sources (who may or may not provide additional documentation).

4.1 The Library of Miscellany: `libmisc.a`

This library contains various modules which are used throughout the Toolkit. The abstractions in most common use are String, Linked_List, Dictionary, and Error. Other modules in this library are Stack, Dynamic, and Hash. The object library is

`~pdes/lib/libmisc.a`, and the sources can be found in `~pdes/src/libmisc/` (`.h` files in `~pdes/include/`).

The file `~pdes/include/basic.h` includes various simple definitions: a `typedef Boolean`, as an enumeration of `{false, true}`; a `Generic` pointer type; `MAX` and `MIN` macros, etc. It is included by every source file in the Toolkit.

Only error codes unique to each routine, are listed after each description.

4.1.1 Boolean

In almost all cases, booleans are manipulated as primitives by the C runtimes. One exception exists – printing.

Procedure: BOOLprint
Parameters: Boolean
Returns: String
Description: despite its name, this function returns a string describing the boolean.

4.1.2 Class

A Class encapsulates meta-data about a class of similar data objects. It includes various generic manipulation functions and information about how instances of the class are arranged in memory.

Type: Constructor
Definition: void (*)(Generic)
Description: The constructor function for a class initializes the block of class-specific data for an instance of the class. It does not allocate storage for the block itself.

Type: Copier
Definition: void (*)(Generic, Generic)
Description: The copier function for a class copies a block of class-specific data for an instance of the class into a second such block.

Type: Comparator
Definition: Boolean (*)(Generic, Generic)
Description: The comparator function for a class compares the blocks of class-specific data from two instances of the class; it returns false if the blocks are considered unequal and true otherwise.

Type: Destructor
Definition: void (*)(Generic)
Description: The destructor function for a class releases the various class-specific components of an instance of the class. It does not release the data block itself.

Type: Printer
Definition: void (*)(Generic)
Description: The printer function for a class prints an object instance in a format specific to the class. Its primary use is for debugging.

Procedure: CLASScreate
Parameters: String name - name of new class
Class super - parent of new class
Constructor create - constructor for instance data
Copier copy - copy method for instance data
Comparator compare - comparison method for instance data
Destructor delete - destructor for instance data
Printer print - printer for instance data
Error *errc - buffer for error code
Returns: Class - the newly created class
Description: Creates and returns a new class.

Procedure: CLASScreate_dataless
Parameters: String name - name of new class
Class super - parent of new class
Error *errc - buffer for error code
Returns: Class - the newly created class
Description: Creates and returns a new dataless class. A dataless class has no instance data slot of its own, and so does not require a constructor, destructor, copier, or comparator.

Procedure: CLASSget_comparator
Parameters: Class class - class to examine
Returns: Comparator - the comparator for the class
Description: Retrieves a class' instance data comparison method.

Procedure: CLASSget_constructor
Parameters: Class class - class to examine
Returns: Constructor - the constructor for the class
Description: Retrieves a class' instance data constructor.

Procedure: CLASSget_copier
Parameters: Class class - class to examine
Returns: Copier - the copier for the class
Description: Retrieves a class' instance data copy method.

Procedure: CLASSget_destructor
Parameters: Class class - class to examine
Returns: Destructor - the destructor for the class
Description: Retrieves a class' instance data destructor.

Procedure: CLASSget_name
Parameters: Class class - class to examine
Returns: String - the name of the class
Description: Retrieves a class' name.

Procedure: CLASSget_size
Parameters: Class class - class to examine
Returns: int - size of class' instance data slot
Description: Retrieves the size (in bytes) of a class' instance data.

Procedure: CLASSget_slot
Parameters: Class class - class to examine
Returns: int - slot number of class
Description: Retrieves the slot number in which a class' instance data is stored. Note that this is a constant for a given class.

Procedure: CLASSget_superclass
Parameters: Class class - class to examine
Returns: Class - the class' immediate superclass

Procedure: CLASSinherits_from
Parameters: Class child - the class whose ancestry is to be tested
Class parent - the hypothetical parent class to search for
Returns: Boolean - Is the parent class in the child's superclass chain?
Description: Determine whether a class (the child) is a descendant of a particular class (the parent). This function reports true in the degenerate case where parent == child.

4.1.3 Dictionary

A Dictionary consists of a naming function and a homogeneous collection. The collection is ordered alphabetically according to the items' names, as reported by the naming function. The current implementation of this module makes no claim to efficiency: it is simply a wrapper around the Linked List module. Entries are added by insertion sort, and retrieval is by linear search.

Type: Naming_Function
Definition: String (*)(Generic)
Description: This is the type of the function which a Dictionary expects to use to retrieve the name of one of its entries.

Procedure: DICTadd_entry
Parameters: Dictionary dictionary - dictionary to modify
String name - string to be used as key
Generic entry - entry to be added
Error* errc - buffer for error code
Returns: Generic - the added entry, or `NULL` on failure
Requires: Entry is of an appropriate type for the dictionary's naming function.
Description: Adds an entry to a dictionary, provided that the dictionary does not yet contain a definition for the entry's name (as given by the dictionary's naming function).
Errors: `ERROR_duplicate_entry` - An entry with the given name already appears in the dictionary. In this case, entry is set to the original entry.

Procedure: DICTcreate
Parameters: Naming_Function func - the naming function to be used by the new dictionary
Error* errc - buffer for error code
Returns: Dictionary - the newly created dictionary
Description: Creates an empty dictionary. Entries will be sorted according to the strings they produce when passed to the naming function given in this call. Thus, `item1` will precede `item2` exactly when `strcmp(func(item1), func(item2)) < 0`.

Procedure: DICTinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Dictionary module.

Procedure: DICTlookup
Parameters: Dictionary dictionary - the dictionary to look in
String name - the name to look for
Returns: Generic - the entry whose name matches that given
Description: Looks up a name in a dictionary. If no matching entry can be found, `NULL` is returned.

Procedure: DICTdo
Parameters: -- none --
Returns: Generic (whatever kind of object was stored previously)
Description: Successive calls of this function return each element of the dictionary, named in the previous call to DICTdo_init(). When no more objects remain, OBJECT_NULL is returned.

Procedure: DICTdo_init
Parameters: Dictionary dictionary
Returns: void
Description: This function names the dictionary to be traversed by following calls of DICTdo (see that function for more infomation).

Procedure: DICTprint
Parameters: Dictionary
Returns: void
Description: prints the contents of a dictionary. Exactly what is printed can be controlled by setting various elements of the variable dict_print.

Procedure: DICTremove_entry
Parameters: Dictionary dictionary - the dictionary to modify
String name - the name of the entry to remove
Returns: Generic - the entry removed
Description: Removes the named entry from a dictionary, and returns this entry to the caller. If no entry with the given name can be found, `NULL` is returned.

4.1.4 Dynamic

This module puts a clean wrapper on the routines in `libdyna.a` (see section 4.3). Only two calls are provided.

Procedure: DYNAinit
Parameters: -- none --
Returns: void
Description: Initializes the dynamic loading module. This must be called with `argv` in scope, as it is actually a macro which examines `argv[0]`. Alternatively, call `DYNA_init(String me)`, whose single parameter should be `argv[0]`. This method is not recommended, but will work in situations where, for some reason, the value of `argv[0]` is available while `argv` itself is not.

Procedure: DYNAload
Parameters: String filename - the name of the object file to load
Returns: void (*)() - the loaded file's entry point
Description: Loads the named object file into the currently running image, and performs symbol relocation as necessary. The entry point to the file is returned as a pointer to a function of no arguments which returns `void`. If an error occurs during the loading process, it is reported to `stderr` and `NULL` is returned as the entry point.

4.1.5 Error

Error reporting throughout the Toolkit is managed by the Error abstraction. This module was not present in the initial Toolkit design; rather, it has grown in response to needs which have appeared over the course of the Toolkit's development. Some of the specifications and behavior thus seem contrived. The Error module allows subordinate routines to report error conditions to their callers, and allows the callers to strongly influence the form of the message reported to the user. In order to do this, the caller is trusted to test for and report error conditions. A caller who breaches this trust is asking for trouble, since it is the act of actually reporting the error which gives control of the program to the Error module, allowing it to take appropriate steps (such as halting the program on a fatal error).

Modules which may wish to report error conditions create instances of type `Error` at initialization time. Routines which may report errors then expect a pointer to an error buffer as a parameter, declared by convention as the last parameter, `Error* errc`. On exit, this buffer will contain either `ERROR_none`, indicating successful completion, or some error code. The caller may then report the error, filling in the necessary blanks in the format specification (see below), attempt to recover, or simply ignore it (realizing that ignoring any but the most innocuous errors will most likely lead to trouble later on).

An Error has two main components. The severity of an error indicates how serious the error is. A warning may be reported to the user, but is not really considered an error. Continuing past a warning, or even 100 warnings, should cause no serious problems. An error, on the other hand, must be noted by the program: The program need not halt immediately, but at some point in the future, it will become impossible to proceed. An error of "exit" severity causes the program to exit immediately, as gracefully as possible. An error of "dump" severity causes the program to dump core and exit immediately. All of these actions are taken only when the error is reported (with `ERRORreport()`), rather than when the error is discovered.

The other component of an error is its text. This is a `printf`-style format string, whose arguments will be filled in when the error is reported. For example, the text for `ERROR_memory_exhausted` is "Out of memory allocating %d for %s." When this error is reported, the amount of memory requested and its intended purpose should be provided by the programmer:

```
ERRORreport(ERROR_memory_exhausted,
     block_size, "file buffer block");
```

For specifications of the Errors defined in `libmisc.a`, see section 4.1.11.

For greater flexibility in error reporting, Errors can be enabled and disabled individually. Disabled errors which are given to `ERRORreport()` will be ignored, just as `ERROR_none` is.

An alternate routine for reporting errors is `ERRORreport_with_line()`, which inserts a line number indication at the beginning of a message. Particularly when line numbers are included, it may be useful to sort error messages before printing them. This can be done by asking that error messages be buffered. When this message buffer is flushed, its contents are sorted according to the third column, which is where `ERRORreport_with_line()` puts the line number. This feature is used in the second pass of Fed-X, for example, where the Express Working Form data structures are walked in the most convenient order, which bears little resemblance to the order in which the original constructs appeared in the source file. Any error messages encountered are buffered, and all are sorted and flushed after the entire pass is complete, resulting in sensibly ordered output.

Type: Severity
Description: This type is an enumeration of `SEVERITY_WARNING`, `SEVERITY_ERROR`, `SEVERITY_EXIT`, `SEVERITY_DUMP`, and `SEVERITY_MAX` (which is guaranteed to be the highest possible severity of any error).

Procedure: ERRORabort
Parameters: -- none --
Returns: does not return
Description: provides a way of aborting the program when an unusual error occurs such as an internal Fed-X error that should be investigated. If ERRORdebugging is true, control is returned to the debugger, else an image of the program is dumped (core) and the program is aborted.
In all cases, pending messages are flushed.

Procedure: ERRORbuffer_messages
Parameters: Boolean flag - to buffer or not to buffer
Returns: void
Description: Selects buffering of error messages. Buffering is useful when error messages are produced by `ERRORreport_with_line()`, as it allows the messages to be sorted according to line number before being displayed.
Note that this should be called with parameter false at program termination.

Procedure: ERRORclear_occurred_flag
Parameters: -- none --
Returns: void
Description: Clear the flag which is used to remember whether any errors have occurred.

Procedure: ERRORcreate
Parameters: String message - message to print for error
Severity severity - severity of error
Error* errc - buffer for error message
Returns: void
Description: Create a new error. The meanings of the various severity levels are as follows: `SEVERITY_WARNING` indicates that a warning message should be generated. This will not interfere with later operation of the program. `SEVERITY_ERROR` produces an error message, and the fact that an error has occurred will be remembered (e.g., so that no reports will be generated). `SEVERITY_EXIT` indicates that the error is fatal, and should cause the program to exit immediately. `SEVERITY_DUMP` causes the program to exit immediately and produce a core dump. `SEVERITY_MAX` is guaranteed to be the highest severity level available. The message string may contain `printf`-style formatting codes, which will be filled when the message is printed.

Variable: ERRORdebugging
Type: Integer
Description: If true, serious errors trap back to the debugger. If false, the program is aborted with a core dump.

Procedure: ERRORdisable
Parameters: Error error - the error to disable
Returns: void
Description: Disable an error, so that the `ERRORreport` calls will ignore it.

Procedure: ERRORenable
Parameters: Error error - the error to enable
Returns: void
Description: Enable an error, ensuring that the `ERRORreport` calls will report it.

Procedure: ERRORflush_messages
Parameters: -- none --
Returns: void
Description: Flushes the error message buffer to the standard output, sorted by line number (the third column).
Despite the name, ERRORbuffer_messages(false) should be called at program termination rather than this function, since it has the unfortunate side-effect of creating a new message buffer. (This should be changed.)

Variable: ERROR_from_file
Type: char *
Description: Defines the name of the file used by the error printing routines.

Procedure: ERRORhas_error_occurred
Parameters: -- none --
Returns: Boolean - has an error occurred?
Description: Check whether any errors (`severity >= SEVERITY_ERROR`) have occurred since the flag was last cleared.

Procedure: ERRORinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Error module. If not explicitly called, this is nonetheless called when necessary. Thus, it can safely be ignored, but is included for completeness.

Procedure:	ERRORis_enabled
Parameters:	Error error - the error to test
Returns:	Boolean - is reporting of this error enabled?

Procedure:	ERRORreport
Parameters:	Error what - the error to report ... - arguments for error string
Returns:	void
Description:	Report an error, taking action appropriate for its severity. The remaining arguments should match the format codes in the message string for the error.

Procedure:	ERRORreport_with_line
Parameters:	Error what - the error to report int line - line number of error ... - arguments for error string
Returns:	void
Description:	Report an error, including a line number. Otherwise identical to `ERRORreport()`.

4.1.6 Hash

The Hash module emulates Unix's `hsearch(3)` package with dynamic hashing. The module header reads, in part:

```
Dynamic hashing, after CACM April 1988 pp 446-457,
     by Per-Ake Larson.
Coded into C, with minor code improvements, and with
     hsearch(3) interface,
by ejp@ausmelb.oz, Jul 26, 1988: 13:16;
```

The code was downloaded from the Internet, and modified significantly in order to support hash table traversal, hash table copying, entry deletion, detecting duplicate entries or removal of nonexistent entries.

Note that all entries in the hash table are shallow copies.

Type:	Action
Description:	This type is an enumeration of `HASH_FIND`, `HASH_INSERT`.

Type:	Element
Description:	The entries in a hash table are stored as Elements. An Element has a `char*` (string) key, a `char*` data field, and a next pointer.

Procedure:	HASHcopy
Parameters:	Hash_Table
Returns:	Hash_Table
Description:	A new table is return that is a duplicate of the original table. The objects in the table are shallow copied.

Procedure:	HASHcreate
Parameters:	unsigned count - estimated maximum number of table elements
Returns:	Hash_Table - the new hash table
Description:	Creates a new, empty hash table.

Procedure: HASHdestroy
Parameters: Hash_Table table - the table to be destroyed
Returns: void
Description: Destroys a hash table, releasing all associated storage.

Procedure: HASHlist
Parameters: -- none --
Returns: Element
Description: Successive calls of this function return each element of the hash table, named in the previous call to HASHlist_init(). When no more objects remain, NULL is returned.

Procedure: HASHlist_init
Parameters: Hash_Table
Returns: void
Description: This function names the hash table to be traversed by following calls of HASHlist (see that function for more infomation).

Procedure: HASHsearch
Parameters: Hash_Table table - the table to search
Element item - the item to search for/insert
Action action - the action to take on the search item
Returns: Element - the result of the action
Description: If action is HASH_INSERT, element is inserted if new. If duplicate, NULL is returned.
If action is HASH_FIND, element is returned, or NULL if no such element exists.
If action is HASH_DELETE, element is returned, or NULL if no such element exists.

4.1.7 Linked List

The Linked List abstraction represents heterogeneous linked lists. Each element of a list is treated as an object of type `Generic`; any object which can be cast to this type can be stored in a list. Note that the programmer must provide a mechanism for determining the type of an object retrieved from a list: this module maintains no such type information.

Type: Link
Description: Each element of a linked list is stored as a `Link`, which has `next` and `prev` pointers and a Generic data field.

Procedure: LISTadd_all
Parameters: Linked_List list - list to modify
Linked_List items - list of items to add
Returns: void
Description: Add the contents of `items` to the end of `list`.

Procedure: LISTadd_first
Parameters: Linked_List list - list to modify
Generic item - item to add
Returns: Generic - the item added
Description: Add an item to the front of a list.

Procedure: LISTadd_last
Parameters: Linked_List list - list to modify

	Generic item - item to add
Returns:	Generic - the item added
Description:	Add an item to the end of a list.

Iterator:	LISTdo ... LISTod
Usage:	Linked_List list; LISTdo(list, <variable_name>, <type>) process_value(<variable_name>); LISTod;
Description:	The macro pair `LISTdo()...LISTod;` are used to iterate over a list. `type` is a C language type; `variable` is declared to be of this type within the block bracketed by these two macros. `variable` is successively assigned each value on the list, in turn.

Procedure:	LISTempty
Parameters:	Linked_List list - the list to be tested
Returns:	Boolean - true if and only if the list contains no elements

Procedure:	LISTget_first
Parameters:	Linked_List
Returns:	Generic
Description:	returns first element of list or NULL if no such element

Procedure:	LISTget_second
Parameters:	Linked_List
Returns:	Generic
Description:	returns second element of list or NULL if no such element

Procedure:	LISTinitialize
Parameters:	-- none --
Returns:	void
Description:	Initialize the Linked_List module.

Procedure:	LISTpeek_first
Parameters:	Linked_List list - list to examine
Returns:	Generic - the first item on the list
Requires:	!LISTempty(list)

Procedure:	LISTprint
Parameters:	Linked_List
Returns:	void
Description:	prints the contents of a list. Exactly what is printed can be controlled by setting various elements of the variable list_print.

Procedure:	LISTremove_first
Parameters:	Linked_List list - list to modify
Returns:	Generic - the item removed
Description:	Remove the first item from a list and return it.
Requires:	!LISTempty(list)

4.1.8 Object

Together with the Class module, this module provides an object-oriented framework on which class hierarcies with data inheritance can be built. One aspect of the Class/Object representation deserves comment. An Object is represented as a header block and a set of instance data slots. Each slot contains the instance data specific to a particular class in the ancestry of the Object's class. For example, if `Cartesian_Point` is a subclass of `Point`, and `Point` is a subclass of `Geometry`, which has no superclass, then an instance of `Cartesian_Point` will contain three slots. The first will contain instance data for a generic `Geometry` object; the next will contain the data for a `Point` object; and the last will contain the instance data which is specific to a `Cartesian_Point`. The instance data required by a particular class, then, is always found in the same slot: In the example above, `Geometry` data will always be found in slot 0, and `Cartesian_Point` data in slot 2. This slot number is recorded in the definition of a Class. A call is provided to retrieve the instance data from a particular Class' slot in an Object (see `OBJget_data()`).

Procedure:	OBJbase_class_assertion
Parameters:	Object object
	Class class
	Boolean error_type - if error should be considered an internal error or a application error
Returns:	Boolean - true if assertion is true, else false
Description:	A pointer to a function supplied by the application. The function may be called by the user or Fed-X internals when testing whether an object is of a given class. Presumably, the function may issue diagnostics describing what class of object was encountered and what was expected.
	If the error type (ERROR_fedex, ERROR_user) indicates it is an internal Fed-X error, ERRORabort() is called.

Procedure:	OBJbecome
Parameters:	Object old - object to replace definition of
	Object new - object to replace with
	Error* errc - buffer for error code
Returns:	void
Requires:	old != OBJECT_NULL
	new != OBJECT_NULL
Description:	Replace an object with a new object. All references to the old object will refer to the new object. This operation is not commutative: the old object is destroyed in the process.

Procedure:	OBJcopy
Parameters:	Object object - the object to be duplicated
	Error* errc - buffer for error code
Returns:	Object - copy of object
Description:	Creates a duplicate (deep copy) of an object. The contents of each instance data slot are copied using the corresponding class' copy method.

Procedure:	OBJcreate
Parameters:	Class class - class of object to create
	Error* errc - buffer for error code

Returns: Object - the newly created object
Description: Create a new object of a particular class. The contents of each instance data slot are initialized using the corresponding class' constructor.

Procedure: OBJcreate_constant
Parameters: Class class - class of object to create
Error* errc - buffer for error code
Returns: Object - the newly created constant object
Description: Create a new constant object of a particular class. A constant object cannot be modified. The contents of each instance data slot are initialized using the corresponding class' constructor.

Procedure: OBJequal
Parameters: Object object1 - one object to compare
Object object2 - one object to compare
Error* errc - buffer for error code
Returns: Boolean - are the objects equal?
Description: Compares two objects and determines whether they are equal. The contents of corresponding instance data slots are compared using the appropriate class' comparison method.

Procedure: OBJfree
Parameters: Object object - the object to be freed
Error* errc - buffer for error code
Returns: void
Description: Releases (a reference to) an object. If possible (i.e., if there are no other references to this object), all storage associated with the object may be released. The contents of each instance data slot are freed using the corresponding class' destructor.
Errors: `ERROR_manipulate_constant` - the object to be freed is a constant

Procedure: OBJget_class
Parameters: Object object - the object to examine
Returns: Class - the object's class
Description: Retrieves the object's class.

Procedure: OBJget_data
Parameters: Object object - the object to examine
Class class - the class for which instance data is requested
Error* errc - buffer for error code
Returns: Generic - instance data for object from the appropriate class
Description: Retrieves a pointer to the instance data for an object, viewing the object as an instance of a particular class.

Procedure: OBJinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Object module.

Procedure: OBJis_constant
Parameters: Object object - the object to test
Returns: Boolean - is this object a constant?
Description: Determine whether an object is a constant.

Procedure: OBJis_kind_of
Parameters: Object object - the object to examine
Class class - the class to test for
Returns: Boolean - is this object a member of the class?
Description: Determines whether a particular object is an instance of a particular class or of any of its subclasses.

Procedure: OBJprint
Parameters: Object
Returns: void
Description: Prints an object. Output is sent to stdout, unless redirected by calls to OBJprint_file.

Procedure: OBJprint_file
Parameters: String filename
Returns: void
Description: Names file to send further output from OBJprint. `(char *)0` signifies stdout.

The struct Print provides additional control. Attributes are as follows:

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

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

Procedure: OBJreference
Parameters: Object object - the object to be referenced
Returns: Object - reference to input object
Description: Creates a reference (shallow copy) to an object

Procedure: OBJspecialize
Parameters: Object object - the object to be specialized
Class class - new class for object
Error* errc - buffer for error code
Returns: Object - the specialized object
Description: Specializes an object to be an instance of some subclass of its class. All references to the old object will refer to the new object.
Errors: `ERROR_subclass_required` - the new class is not a subclass of the object's current class. This error is reported locally, and `ERROR_subordinate_failed` is propagated.

`ERROR_manipulate_constant` - the object to be specialized is a constant.

4.1.9 Stack

This module implements the classic LIFO Stack. It is implemented as a set of macros wrapped around the Linked List abstraction. Stacks may be heterogeneous.

Procedure: STACKempty
Parameters: Stack stack - the stack to be tested
Returns: Boolean - is the stack empty?
Description: Returns true if stack is empty, else false.

Procedure: STACKinitialize
Parameters: -- none --
Returns: void
Description: Initialize the Stack module.

Procedure: STACKpeek
Parameters: Stack stack - the stack to peek at
Returns: Generic - the top item on the stack
Requires: !STACKempty(stack)
Description: Peeks at the top of a stack, returning it without removing it from the stack.

Procedure: STACKpop
Parameters: Stack stack - the stack to pop
Returns: Generic - the top item on the stack
Requires: !STACKempty(stack)
Description: Removes the top item from a stack and returns it to the caller.

Procedure: STACKprint
Parameters: Stack
Returns: void
Description: prints the contents of a stack. Exactly what is printing can be controlled by setting various elements of the variable list_print (since the current implementation of a stack is via a list.

Procedure: STACKpush
Parameters: Stack stack - the stack to push onto
Generic item - the item to push
Returns: void
Description: Pushes an item onto the top of a stack.

4.1.10 String

This module defines macros and functions for manipulating C strings. Some routines provide special functionality, while others simply rename standard calls from the C library to fit the naming scheme of the Toolkit. The `String` type is a synonym for `char*`.

Procedure: STRINGcompare
Parameters: String s1 - first comparison string
String s2 - second comparison string
Returns: int - measure of equality of strings
Description: This is an alias for the standard C call `strcmp()`. The result is `0` when the two arguments are equal, negative when s1 precedes s2 in lexicographical order, and positive when s1 follows s2.

Procedure: STRINGcopy
Parameters: String string - the string to copy
Returns: String - a deep copy of the argument
Description: Allocates a String large enough to hold the (`NUL`-terminated) argument, copies the argument into this String, and returns it to the caller.

Procedure: STRINGcopy_into
Parameters: String dest - the destination string
String src- the string to be copied
Returns: dest
Requires: STRINGlength(dest) >= STRINGlength(src)
Description: This is an alias for the C library call `strcpy()`. The source string is copied into the destination string, which must be of equal or greater length.

Procedure: STRINGcreate
Parameters: int length - length of string to create
Returns: String - a new, empty string of at least the given length
Description: Creates a new string.

Procedure: STRINGdowncase_char
Parameters: char c - the character to convert
Returns: char - the argument character, as lower case if it is a letter
Description: Converts an uppercase character to lowercase.

Procedure: STRINGequal
Parameters: String s1 - first string for comparison
String s2 - second string for comparison
Returns: Boolean - are the two strings equal?
Description: Compares two strings for value equality. This call is equivalent to `strcmp(s1, s2) == 0`.

Procedure: STRINGfree
Parameters: String string - the string to be released
Returns: void
Description: Allows all storage associated with a string to be reclaimed. References to the string may no longer be valid.

Procedure: STRINGlength
Parameters: String string - the string to measure
Returns: int - the actual length of the string, excluding the `NUL` terminator
Description: This call is equivalent to `strlen(string)`.

Procedure: STRINGlowercase
Parameters: String string - the string to convert
Returns: String - lowercased version of the argument
Description: A new string is created and returned which contains the same value as the argument, but with all letters replaced with their lowercase counterparts.

Procedure: STRINGsubstring
Parameters: String str - string to extract a substring from
int from - beginning index for substring
int to - ending index for substring
Returns: String - the specified substring
Description: A new string is created and returned whose value is a particular substring of some string. The index of the first character of a string is 0.

Procedure: STRINGupcase_char
Parameters: char c - the character to convert
Returns: char - the argument character, as upper case if it is a letter
Description: Converts a lowercase character to uppercase.

Procedure: STRINGuppercase
Parameters: String string - the string to convert
Returns: String - uppercased version of the argument
Description: A new string is created and returned which contains the same value as the argument, but with all letters replaced with their uppercase counterparts.

4.1.11 Error Codes

This section specifies all of the Errors which are defined in `libmisc.a`. Note that each is a global variable; storage is allocated for each by the module named.

Error: ERROR_duplicate_entry
Defined In: Dictionary
Severity: SEVERITY_ERROR
Meaning: A name was duplicated in a dictionary
Format: %s - the duplicated name

Error: ERROR_empty_list
Defined In: Linked_List
Severity: SEVERITY_ERROR
Meaning: Illegal operation on an empty list
Format: %s - the context (function) in which the error occurred

Error: ERROR_free_null_pointer
Defined In: Error
Severity: SEVERITY_DUMP
Meaning: A NULL pointer was freed
Format: %s - the name of the offending function

Error: ERROR_memory_exhausted
Defined In: Error
Severity: SEVERITY_EXIT
Meaning: A `malloc(2)` request could not be satisfied
Format: %d - number of bytes requested
%s - intended use for memory

Error: ERROR_none
Defined In: Error
Severity: N/A
Meaning: No error occurred. In another life, this might have been called `ERROR_NULL`. But then, who knows?!
Format: -- none --

Error:	ERROR_not_implemented
Defined In:	Error
Severity:	SEVERITY_EXIT
Meaning:	An unimplemented function was called.
Format:	%s - the name of the function

Error:	ERROR_obsolete
Defined In:	Error
Severity:	SEVERITY_WARNING
Meaning:	An obsolete function was called.
Format:	%s - the obsolete function name %s - new name to use OR reference to replacement code OR "<No Replacement>"

Error:	ERROR_subordinate_failed
Defined In:	Error
Severity:	SEVERITY_ERROR
Meaning:	A subordinate function has failed and reported an error to the user. Useful when the caller only needs to know that a problem has occurred. This error is not reported.
Format:	-- none --

4.2 The Bison Support Library: `libbison.a`

The Bison support library is based on the standard Unix Yacc support library `libyacc.a`., with modifications to support better error handling/reporting, implementation differences between Yacc and Bison (and also between Lex and Flex), and more careful use of global variables, this latter to allow more than one Bison parser to be linked into a single executable. The library is in `~pdes/lib/libbison.a`, and sources can be found in `~pdes/src/libbison/`.

The definitions of `yyerror()` in `yyerror.c` and `yywhere()` in `yywhere.c` are from [Schreiner85].

Several variable declarations in these two files had to be modifed for Bison/Flex parsers. A documented difference between Lex and Flex is that the token buffer, `yytext`, is declared as a `char*` in Flex and as a `char[]` in Lex. Also, Flex does not provide Lex's `yyleng` variable. Other variables which need to be declared `extern` in Bison parsers so as not to collide when multiple parsers are linked together have storage allocated in `yyvars.c`. This file also defines a function `yynewparse()`, which can be used to restart a Bison parser.

A word on the `~pdes/etc/uniquify_*` scripts. These `csh` scripts modify the code produced by Yacc/Bison/Lex/Flex so that multiple scanners and parsers can coexist in a single executable. For the most part, it is sufficient to change some global variable declarations to be `static`. Each script strips any of several suffixes off of the filename it is given to determine the actual name of the parser/scanner and then prepends this name to type and function declarations which are externally visible. Thus, a parser called `expyacc.y` ends up with the entry point `exp_yyparse()`, expects tokens of type `exp_YYSTYPE`, and calls `exp_yylex()` to get these tokens. Similarly, a scanner called `stepscan.l` would provide `step_yylex()` as an entry point, and would produce tokens of type `step_YYSTYPE`.

4.3 BSD Unix Dynamic Loading: `libdyna.a`

This package was retrieved from the Internet. Authorship information seems to have been lost. The routines provided are at the level of reading `a.out` headers and walking through symbol tables. We will not attempt to document this library; there are `.doc` files in the source directory, `~pdes/src/libdyna/`, which include examples of the package's use.

A 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., Libes, D., Fed-X: The NIST Express Translator, NISTIR 4822, National Institute of Standards and Technology, Gaithersburg, MD, August 1990.

[Clark90c] Clark, S.N., The NIST Working Form for STEP, NISTIR 4351, National Institute of Standards and Technology, Gaithersburg, MD, June 1990.

[Clark90d] Clark, S.N., Libes, D., NIST Express Working Form Programmer's Reference, NISTIR 4814, National Institute of Standards and Technology, Gaithersburg, MD, March 1992.

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

[Goldberg85] Goldberg, A. and D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, MA, July, 1985.

[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.

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

[Nickerson90] Nickerson, D., The NIST SQL Database Loader: STEP Working Form to SQL, NISTIR 4337, National Institute of Standards and Technology, Gaithersburg, MD, May 1990.

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

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

[Schreiner85] Schreiner, A.T., and H.G. Friedman, Jr., Introduction to Compiler Construction with Unix, Prentice-Hall, Englewood Cliffs, NJ, 1985.

B The Makefile Template

```
#
# This is a Makefile template for translators and other applications
# which use the Express and/or STEP Working Forms from the NIST PDES
# Toolkit.
#
# This software was developed by U.S. Government employees as part of
# their official duties and is not subject to copyright.
#

# Pick up default macros and rules
include ../../include/make_rules

################################
# Pick a C compiler ... any C compiler!
################################

#CC = $(Unix_CC)
CC = $(GCC)

################################
# User-definable flags to CC:
# Put whatever you want in here!
################################

#MY_CFLAGS = -g -O
MY_CFLAGS = -g

################################
# CC flags for Express and STEP
#
# Use the first form for STEP applications.
# Use the second if only Express is required.
################################

CFLAGS = $(STEP_CFLAGS) $(MY_CFLAGS)
#CFLAGS = $(EXPRESS_CFLAGS) $(MY_CFLAGS)

################################
# Default rule to compile C source files
#
# You probably shouldn't need to change this ...
################################

#.c.o:
#        $(CC) $(CFLAGS) -c $*.c

################################################################
#
# Library Selection
#
# Select the first one of the following forms which describes your
# application.  For further discussion, see "The NIST PDES Toolkit:
# Technical Fundamentals."
#
################################################################

################################
# STEPparse translators/applications:

# with statically bound report generators
#LIBS =  $(STEP_LIBS)
```

```
# with dynamically bound report generators
#LIBS = $(PDESLIBDIR)step_dynamic.o $(STEP_LIBS) -ldyna

################################
# Fed-X Express translators/applications:

# with statically bound report generators
#LIBS = $(EXPRESS_LIBS)

# with dynamically bound report generators
#LIBS = $(PDESLIBDIR)express_dynamic.o $(EXPRESS_LIBS) -ldyna

################################
# STEP applications with Express report generators

# statically bound
#LIBS = $(STEP_LIBS)

# dynamically bound
#LIBS = $(PDESLIBDIR)express_dynamic.o $(STEP_LIBS) -ldyna

################################
# STEP application with no report generators

#LIBS = $(STEP_LIBS)

################################
# Pure Express application with no report generators

#LIBS = $(EXPRESS_LIBS)

################################################################
# List all of your object files here.  If you are building a
# translator which will dynamically load its report generators,
# do not list any output modules here.
################################################################

# Object files for Fed-X or STEPparse translator with dynamically
# loaded report generators
#OFILES =

# Object files for STEPparse translator with STEP report
# generator statically loaded
#OFILES = step_output_step.o

# Object files for Fed-X translator with Smalltalk-80 report
# generator statically loaded
#OFILES = output_smalltalk.o

################################################################
# List all of your libraries here
################################################################

MYLIBS =

################################################################
# The name of the executable to build
################################################################

PROG =

################################################################
# Here's the rule that builds the executable.
```

```
##################################################################

$(PROG): $(OFILES)
	$(CC) $(CFLAGS) -o $(PROG) $(OFILES) $(LIBS) $(MYLIBS)

relink:
	$(CC) $(CFLAGS) -o $(PROG) $(OFILES) $(LIBS) $(MYLIBS)

install:
	cp $(PROG) $(PDESBINDIR)

clean:
	rm -f $(OFILES) $(PROG) *.~*~ #*

##################################################################
# Put any rules for building your object files here.
##################################################################
```