Principles for Writing Reusable Libraries

Glenn S. Fowler **David G. Korn** **Kiem-Phong Vo**

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974 USA
{gsf,dgk,kpv}@research.att.com

Abstract

Over the past 10 years, the Software Engineering Research Department in AT&T has been engaging in a research program to build a collection of highly portable advanced software tools known as Ast, Advanced Software Technology. A recent monograph, "Practical Reusable UNIX Software" (John Wiley & Sons, Inc., 1995), summarizes the philosophy and components of this research program. A major component of this program is a collection of portable, and reusable libraries servicing a wide range of functions, from a porting base to all known UNIX platforms, to efficient buffered I/O, memory allocation, data compression, and expression evaluation. The libraries currently stand at about 150,000 non-commented lines of C code. They are developed and maintained independently by different researchers. Yet they work together seamlessly – largely because of a collection of library design principles and conventions developed to help maintaining interface consistency and reducing needless or overlapped work.

1 Introduction

In the early years of C and UNIX programming, many general purpose libraries were produced and widely distributed. These libraries provided a wide variety of functions for mathematics, buffered I/O, dynamic memory allocation, etc. Their availability led to a tremendous growth in programmer productivity. By virtue of their widespread use, the libraries became de facto standards and were commonly called the standard C libraries. These libraries stood as some of the best examples of successful reusable software.

In the early 80s, much fewer reusable C libraries were constructed. A number of factors contributed to this decline. In AT&T as well as the industry at large, the main focus of most development organizations was aimed toward hardware and kernel development, not reusable libraries. This direction of work was driven partially by the belief that except for application-specific products, the main value of software was to help sell hardware. This was always a dubious assumption and it is no longer valid at current prices for high performance stock hardware.

From a language point of view, an important factor was the lack of direct support for modularization in C. Though conventions could be formed to alleviate the problem, such conventions were either ill-defined or more often ignored. This situation was worsened by the explosive growth of the UNIX system as a platform for building software applications. During this time, most effort was dedicated to building application-specific products, not reusable software. The latter was often viewed as an unnecessary luxury. As applications expanded and branched into families and demands increased for quick turn-around of new features, the need for standard reusable software components has become critical.

The introduction of the C++ programming language in the mid 80s put an additional damper on the development of new C libraries. C++ had better support for interface encapsulation than C. This simplified the creation of new libraries. Moreover, since C++ was in its infancy, there was no backward compatibility problems to contend with. The result was that much of the recent best library work in the C family of languages occurred in the C++ arena, including many reimplementations of C libraries in C++.

Despite the lack of support for modularization, it is possible to write reusable C libraries that also works with any C variant, including C++. Over the past ten years, we have been writing a collection of reusable C libraries as a part of a research program to build highly portable advanced software development tools known within AT&T as *Ast*, Advanced Software Technology. The overall philosophy and specific components of this research program are discussed in a recent monograph, "Practical Reusable UNIX Software" [2].

The *Ast* libraries cover a broad spectrum of functions ranging from those traditionally provided in *libc* (but more portable) to others for general network connection, I/O, memory allocation, data compression, and other sophisticated computing techniques. The libraries currently stand at about 150,000 lines of C code and has been ported to virtually every combination of UNIX software/hardware platform, Windows and Windows NT. They are widely used both in our own work and in other applications including commercial products. The libraries came out of diverse needs and requirements and were often written by one or two researchers. A number of design principles and conventions were developed and evolved along with the code through the years. They helped to maximize effectiveness in this distributive mode of work. The usefulness of these principles and conventions will be demonstrated via a small subset of the libraries: *libast*, the portability base, *libcmd*, enhanced UNIX commands, *sfio*, safe/fast buffered I/O, *stak*, stack-like memory allocation, *expr*, C-like expression evaluation, and *libpp*, C preprocessing.

2 Design considerations

The primary goals in building reusable components are applicability, efficiency, ease of use, and ease of maintenance. However, there is no simple set of rules that would guarantee the simultaneous achievement of these goals. Often, the goals conflict, and decisions have to be made to balance the trade-offs. Below are an eclectic set of design considerations used as guidelines in building the *Ast* software.

2.1 Necessity

A component is not reusable unless it is used. This means that a reusable component should be built out of real needs. A way to meet this condition is to first plan some applications, then to build the functions that make up the applications as one or more libraries. Because libraries are often used in different ways, this approach has the additional advantage of forcing the programmer to think in advance about different usages so code quality is enhanced. Section 5 gives examples of function versions of many standard UNIX commands. These functions can be used to build stand-alone commands or as efficient built-ins in applications such as the shell program.

2.2 Generality

Except for efficiency concerns, reusable components should be designed for their most general applications. Often this means unifying separate but related concepts into a single interface. This is important because applications often use similar mechanisms (e.g., various search structures) in different ways (e.g., for storing objects of different types). A unifying interface both simplifies application construction and increases their ease of maintenance. Good examples of this are the libraries *expr* in Section 7 for C expression evaluation and *libpp* in Section 8 for C preprocessing. These libraries have enabled the construction of sophisticated data processing programs and program analysis systems. Generality often opens up new uses. For example, *sfio* string streams (Section 4) enable manipulation of memory-resident streams as any disk-based streams. In turn, this simplifies the construction of the *stak* library (Section 6) for stack-like memory manipulations.

An aspect of generality related to portability is to provide common abstractions that hide the differences in the underlying platforms. Though our software is UNIX-based, it is no secret that no two versions of UNIX are the same. In the short term, the existence of standard bodies such as POSIX [12] actually worsens the situation as the standards tend to be some amalgam of existing systems but unlike any of them. Sometimes when the differences in extant implementations of a desired feature are wide enough, the standards may even shy away from defining one. Section 3 describes a set of functions and header files that combine features from various UNIX flavors. Our tools are written based on this interface to increase portability.

2.3 Variability

A library has two different types of interfaces. The first is what it provides for applications to use and that should be general as discussed above. The second is what it requires from the external environments for its functioning. For example, a buffered I/O library such as *sfio* on UNIX systems would need system calls such as `read()` and `write()`. Sometimes it is profitable to make abstract such dependencies so that applications can redefine them as necessary. In this way, variants of a library can be created without having to tamper with its internals. The paper [15] discusses disciplines as interfaces designed to capture external resource dependencies. Section 4 gives an example of the power of such abstractions.

2.4 Efficiency

Efficiency is a primary consideration in building a reusable component because the performance of such a component is amplified by its repeated use. Without high performance reusable components, programmers will be tempted to hand-code and create applications that are hard to maintain. There are two aspects of efficiency: internal and external.

Internal efficiency: This means first that library components are implemented using best known data structures and algorithms. Then, it is sometimes beneficial to optimize code based on most popular use or local hardware and platform features. An example of this type of optimization is the decimal to ASCII conversion algorithm in the `sfprintf()` function of *sfio*. Here, because base 10 is most commonly used, it is handled using a fast customized algorithm. Other bases are handled by a general but slower method.

External efficiency: This means that the library interface is designed so that critical resources managed by the library can be efficiently accessed by applications. An example of this is the *sfio* function `sfreserve()` that allows an application to directly and safely access the internal buffer of an I/O stream. For applications accessing large chunks of data, this can dramatically reduce the number of memory copying operations between stream and application buffers while still minimizing system calls. We have rewritten many system commands such as *pack* and *wc* (Section 5) based on `sfreserve()` with up to a factor of four in performance improvement over the BSD4.3 versions of the same commands.

2.5 Robustness

A successful reusable component should be robust with respect to stresses on critical resources. There are two aspects of robustness: internal and external.

Internal robustness: This means that the library components should be well tested in a variety of environments, their implementation does not impose any artificial constraints on resources, and they can respond well to unexpected events. The *Ast* components are continually tested and used on nearly every UNIX platform. Artificial constraints such as fixed size arrays, number of bits in an `int`, etc. are duly avoided. The code is written in a style compilable under the K&R C, ANSI C and C++ dialects so that it can be tested with the type checking mechanisms of many C compilers, each with its own strengths and weaknesses. In addition, the code can be used transparently by applications based on different C dialects.

External robustness: This means that the library should prevent applications from making inherently unsafe usage and provide them with ways to deal with exceptions. An example of unsafe usage is the *stdio* function `gets()` which takes as input a buffer with unspecified size and returns data of unspecified length. Since neither buffer nor data sizes are known in advance, there is no precaution that either the library or the application can make to prevent buffer overflow. By contrast, the *sfio* function `sfgetr()` returns a pointer to a record delineated by some application-defined record separator. The space for the record is internally managed by the library as only it can know how much space is required.

2.6 Modularity

Modularity means to insulate components and functions from one another so that the implementation and use of one will not affect the implementation and use of another. This helps to reduce complexity in component interrelationship. There are two aspects of modularity: internal and external.

Internal modularity: Functions in a library should be orthogonal to simplify usage both within and outside the library. An example is to set the buffer of a stream in *stdio* or *sfio*. While *stdio* disallows buffer changing after any I/O operation, *sfio* streams can change buffers any time. This may seem to be a trivial improvement but it is a crucial feature because *sfio* string streams may use multiple strings.

External modularity: Libraries should be usable in arbitrary order. Of course, using some of them may mean that others will be implicitly required, but such requirements must be transparent at the application level. For example, the *stak* library is based on the *sfio* library. But unless an application wants to use *sfio* output functions on *stak* structures, no knowledge of *sfio* is required.

2.7 Minimality

Having too much in the interface is as bad as having an awkward or inconsistent interface. An interface is needed only if it does something that cannot be done otherwise without significant loss of efficiency or convenience. Examples of gratuitous interfaces are the *stdio* convenience functions such as `getchar()` and `putchar()` that provide simple veneers on top of the general functions `getc()` and `putc()`.

The downside of minimizing the interface is awkward and redundant code at the application level when certain aggregate operations are commonly performed. In such a case, a compromise should be reached. An example is the *sfio* function `sfprints()` that creates a formatted string in some system provided area and returns a pointer to that string. Strictly speaking, an application can create the effect of `sfprints()` by opening a string stream and using `sfprintf()`. However, this is too awkward to repeat in many places.

2.8 Portability

Given the multitude of hardware and software platforms available today, portability is an absolute requirement for successful software. There are two dimensions to portability: code and data.

Code portability: The *Ast* tools are all based on high level libraries. The libraries are written to be compilable with any variant of C, including ANSI C and C++. They hide all platform-specific details from applications and are portable to nearly all known UNIX platforms, and Windows and Windows NT. This level of portability is aided by the *iffe* [8] language for defining feature probes that record porting knowledge and configure code without user intervention.

Data portability: It is desirable that persistent data (e.g., disk files) or data communicated among processes be portable. That is, the data should be independent of the hardware representations. This is a hard problem and a complete solution for aggregate data types would require much more cooperation from languages and compilers than currently possible. However, for primitive types, the problem is treatable. Based on the reasonable assumption that the order of bits in bytes are the same across hardware platforms, the *sfio* library provides function to transparently read and write strings, integers and floating point values.

2.9 Evolvability

A successful reusable library will undergo revisions as its design and implementation are stressed by usage or technology advances. When the interface is sufficiently general, certain types of revision can be kept hidden within the package and the interface can be maintained intact. However, weakness in the design is often not revealed until challenged by new needs; then the interface must change. Sometimes, this amounts to adding new functions to alter the states of the library. However, if new, clean, and well-designed interfaces provide much more benefit than previous ones, then compatibility must be broken. In such cases, it is important to help users ease the transition. An example is the *stdio* source and binary compatibility packages provided with *sfio*. These packages allow applications based on *stdio* to either recompile or simply link with *sfio* transparently. This means that a software project can take advantage of new technologies immediately without too much upheaval in their programming practice.

2.10 Naming conventions

Good interface conventions help to ease the learning curve of a software package and reduce name clashing when different packages are used together in a single application. As libraries are developed by different people at different time, it is hard to achieve a uniform set of conventions. But, by and large, the naming conventions followed in *Ast* are:

Standard prefixes: Constants, functions, and variables used in a package are always named using a small and unique set of prefixes that clearly identify the package. For example, the prefixes `SF`, `Sf` and `sf` identify *sfio* elements.

Standard argument ordering: Functions typically manipulate some structures that carry states across calls. Such state-carrying structures always come first in a argument list. For example, in all *sfio* calls, the stream argument is always the first. Sometimes arguments come in pairs (e.g., a buffer and its size). Then, the one containing data or used to store data comes first (e.g., the buffer comes before its size). Flag arguments for mode control are always last.

Object identification: A library typically defines and uses many different objects. It is helpful to use naming conventions that distinguish different object types. Preprocessor symbols or macros (e.g., SF_READ) are defined using upper case letters. Non-functional global symbols (e.g., Sfio_t) often start with an upper case letter. Sfio_t also shows that a library-defined type often has an affix _t. Function names (e.g., sfopen()) are always in lower case.

Reducing private global symbols: Global data private to a library is often placed in a single struct so that only one identifier is taken from the name space. For example, all private global data of the *sfio* library are kept in a structure _Sfextern. The leading underscore in _Sfextern further emphasizes that it is a private symbol.

2.11 Architecture conventions

Architecture conventions help to fit a library into other families of libraries, simplifies the library design and eases the learning process for new users. Below are some of the conventions used in the *Ast* libraries.

Reusing well-known architecture conventions: Inventing a new library does not necessarily mean inventing new architecture and conventions. It is often advantageous to follow already familiar conventions. For example, in many libraries, the *modus operandi* is to create some data structure, manipulate it, and finally destroy it. A good existing convention is practiced by the UNIX file manipulation system calls: open(), read(), write(), lseek(), and close(). Here, open() creates a file descriptor, a data structure that carry states across system calls, and close() destroys this data structure.

Saving and restoring states: C and its sibling languages are stack-like in their function call convention. Certain data structures in a library are shared across function calls. Functions should be designed so that state information can be saved and restored seamlessly. A good convention for a function that alters states is to always return the previous state. In this way, a function can call another to perform some work, then restore the states before returning. For example, the *sfio* function sfset(), used to set the flags controlling a stream, always returns the previous set of flags.

Information hiding: A public structure only needs to reveal enough of its members as required by other interface elements (e.g., fast macro functions). Other members should be hidden from view (in headers private to the library). This prevents applications from improper use of private library data and allows a library to grow without violating compatibility. A somewhat surprising nice effect of minimizing public interface exposure is that the public headers become clear to read and easy to maintain. This is in contrast to many standard headers from UNIX and C++ systems that are littered with private data and other #ifdefs.

Meaningful use of exceptional values: Separate operations can often be merged into one using certain exceptional values. For example, an *sfio* stream stack is build with the call sfstack(base,top) which specifies that the stream top is to be pushed on top of the stream base. I/O operations on the stream stack identified by base are performed on the top stream. A required operation for a stack is to pop the top element. Instead of providing a separate "pop" function, *sfio* does this with sfstack(base,NULL). Since NULL is an exceptional value, using it in a meaningful way like this also induces programmers to be more aware and check for it.

Exception handling: A library should categorize exceptions in its operations and provide ways for applications to handle them. For example, an application based on the *sfio* library in Section 4 can define discipline functions to handle events such as read or write errors in its own way. A library can and should also define default methods to handle such exceptions. However, it should avoid irrecoverable measures such as calling *exit()*.

3 *libast*: The *Ast* porting base

Portability is an essential requirement in any platform designed to support widely used software. Our tools are based on *libast* which provides a common header and function interface for many UNIX systems and C compilers. By confining all architecture-specific details in *libast*, higher level tools can be programmed largely without #ifdef's. This encourages clean tool design and provides a convenient framework for portability. Many interface issues are addressed by *libast*:

Header interface: Determining the necessary set of #include headers for a given system is one of the hardest portability challenges. Missing headers can be handled with feature testing [8]. More difficult are system headers that omit information or define constructs that conflict with other headers. The header `ast_std.h` provides a self-consistent union of many ANSI and POSIX headers including `stdarg.h`, `stddef.h`, `sys/types.h` and `unistd.h`. Consistency is attained by supplying omitted headers, providing defaults for missing definitions, and fixing up botched constructs in local headers. An example is the type `size_t` required in the ANSI C header `stddef.h` and often but not always defined in `sys/types.h` on UNIX systems. The header `ast_std.h` guarantees the definition of `size_t`. `ast_std.h` includes local headers whenever possible (so it may define non-standard symbols). Missing headers and data are generated as necessary.

Missing functions: libast provides implementations for common system calls not supported by the local system. Some calls, like `rename()`, are emulated using `link()` and `unlink()`. Others, like `symlink()`, cannot be emulated, so the library provides a stub that always fails with `errno` set to `ENOSYS`. In this way, applications can be written based on a single system call model.

Replacement functions: Many functions in *libc* have changed little since their introduction in the late 1970's. In many cases, better algorithms and optimizations are now available. *libast* provides replacements for these. For example, `getcwd()` uses the `PWD` environment variable maintained by *ksh* [3] and other modern shells to avoid the complex construction for a current directory path. A dark side of standard headers and function prototypes is illustrated by `getgroups()` whose POSIX and BSD function prototypes are `getgroups(int size, gid_t* groups)` and `getgroups(int size, int* groups)`. This is a serious problem when `sizeof(gid_t)` is different from `sizeof(int)`. *libast* solves this by providing a macro `getgroups()` that calls `_ast_getgroups()` with the proper prototype. Any inconsistency between `gid_t*` and `int*` is handled by `_ast_getgroups()`.

New functions: libast is a common repository for new functions that are shared among the *Ast* tools. There are over 200 public functions in *libast* including large packages like *sfio* and other convenient functions. Examples of the latter are the `str*` routines to convert `char*` strings to other C types. `struid()` converts a string to a `uid_t` and `strperm()` converts a *chmod* file mode expression to a `mode_t`. Each of these has an inverse conversion routine. `fmtuid()` converts a `uid_t` into a `char*` and `fmtperm()` converts a `mode_t` to a *chmod* expression string.

4 *sfio*: Safe/Fast I/O

A main contribution of the UNIX system is the notion of byte streams for I/O. The byte streams, be they disk files, terminals, or disk files are uniformly accessed via the system calls: `read()`, `write()`, and `lseek()`. Since such calls can incur large costs, it is advantageous to use buffering to reduce their number of calls. The *sfio* library [10] provides general buffered I/O. This is done in such a way that local optimizations can be used for efficiency. For example, memory mapping [1], when available, is often more efficient for I/O than `read()` or `write()`. *sfio* provides functions similar to that of the *stdio* package but it corrects a number of deficiencies in *stdio*'s design and implementation. Beyond *stdio*, *sfio* has many new features:

String streams: String streams allow applications to read and write to memory using the same operations normally reserved for file streams. Buffers of write string streams are extended as necessary to accommodate data.

Portable numerical data: Integral and floating point values can be encoded in minimal portable formats for I/O purposes. This allows applications to transport data across hardware platforms without resorting to ASCII which implies space wastage and/or loss of accuracy.

Safe and efficient buffer access: A typical text file operation is to read lines. This can be done with the call `sfgetr(sfstdin,'\n',1)` which reads a record delineated by the newline character and replaces this character with 0. The resulting string is kept in the stream buffer if possible; otherwise, it is built in some system-defined area. Thus, `sfgetr()` is similar to *stdio*'s `gets()` but without any possibility of buffer overflow.

The function `sfreserve()` provides more general access to stream buffers. For example, the call `sfreserve(sfstdin,n,1)` reserves a data segment of size `n` from the standard input stream. `sfreserve()` gives the same I/O power as `sfread()` and `sfwrite()` but more efficient because intermediate buffer to buffer copies are avoided. This works particularly well with memory mapping.

Stream stacks: The call sfstack(base,top) pushes the stream top onto the stream stack identified by base. Any I/O operation on base will be performed on top. This is useful for processing nested files such as #include files. Stream-specific data such as line numbers can be synchronized by installing disciplines (see below) to process end-of-file events.

I/O disciplines: Methods to obtain raw data vary between platforms. To deal with such variability, *sfio* generalizes the I/O system calls and package them in a structure that defines data acquisition methods. This structure is called a discipline. Applications can specialize disciplines on a per stream basis. A discipline is of type Sfdisc_t and has four member functions. The first three functions are for I/O operations: (*readf)(), (*writef)(), and (*seekf)(). A fourth function (*exceptf)() processes exceptions. For example, the call (*except)(f,SF_READ,disc) is raised whenever an end-of-file or error condition occurs on the stream f during a read operation. Other exceptions announce a wide range of events including stream opening or closing, and discipline stack manipulations.

Below is an example of using a discipline to translate input data from upper case to lower case. Lines 1 to 9 define the function lower() which is used as the (*readf)() discipline function on line 10. Note that raw data is read via the function sfrd() on line 4 so that other disciplines, if any, can be invoked. This allows several disciplines to cooperate and process data into the final required form. Line 11 inserts the discipline into the standard input stream. The sfmove() call on line 12 moves the processed input data to the standard output stream. Though simplistic, this example shows how disciplines greatly extend the range of data processing.

```
 1: lower(Sfio_t* f,void* b,int n,Sfdisc_t* d)
 2: {  int    c;
 3:    char* buf = (char*)b;
 4:    n = sfrd(f,b,n,d);
 5:    for(c = 0; c < n; ++c)
 6:        if(isupper(buf[c]))
 7:            buf[c] = tolower(buf[c]);
 8:    return n;
 9: }
10: Sfdisc_t Disc = { lower, 0, 0, 0 };
...
11: sfdisc(sfstdin,&Disc);
12: sfmove(sfstdin,sfstdout,SF_UNBOUND,-1);
```

5 *libcmd*: Enhanced UNIX commands

Two main principles in writing reusable components are necessity and generality. This means that a component should be built only if it is truly needed and then it should be built for general usage. These principles are easily satisfied in our effort to reimplement many common commands in the IEEE POSIX 1003.2 Standard for shell and utilities. The main reason for this effort is to take advantage of the efficiency in existing library components but, once started, each command is implemented first as a library function then an actual command is a simple main() that passes arguments to this function.

Each command function is named b_*name* where *name* is the name of the command. For example, b_cat() is the function corresponding to the command *cat*. These command functions are grouped together in *libcmd*. Recent versions of *ksh* support dynamic linking of built-in commands. Using *libcmd* as a shared library, any of these commands can be made a built-in to the shell as desired. Currently, *libcmd* contains different types of commands: (1) simple commands that take more time to invoke than to run such as *basename* or *dirname*, (2) commands that walk a file hierarchy such as *chmod* or *chgrp*, and (3) I/O-intensive commands such as *cut*, *pack*, *wc* or *paste*.

6 *stak*: Stack-like memory allocation

Interpreters often build parse trees and text strings by substitution of text patterns. Such an object is typically constructed using several allocations but no frees, and when done, all space is freed at once. The allocation overhead for doing this can be high. Interfaces such as alloca() [5] and *vmalloc* [14] are more suitable but function call overhead is still high when many characters or small strings are being glued together. alloca() is also unsuitable if a constructed object must live beyond the function that builds it.

The *stak* library provides a set of macros and functions to build stack-like objects. A stack is represented by the type Stk_t which is derived from a Sfio_t structure so that *sfio* calls for output can also be used on Stk_t. Stacks are opened and closed with stkopen() and stkclose(). Objects on the stack, except the last or current one, are frozen. During its construction, the location of a current object may be moved. So until a current object is frozen with stkfreeze() locations within it can be referred to only by relative offsets and not pointers.

Below is an example of building a path name on the standard stack `stkstd` from a directory name and a base name before opening the corresponding file and returning the resulting file descriptor. Line 2 saves the current location on the standard stack so that it can be reset on line 7 for memory reuse in future calls. Line 8 calls `stkptr()` to convert the current `offset` into a memory address.

```
1: int myopen(const char *dir,const char *name)
2: {   long offset = stktell(stkstd);
3:     sfputr(stkstd,dir,-1);
4:     sfputc(stkstd,'/');
5:     sfputr(stkstd,name,-1);
6:     sfputc(stkstd,'\0');
7:     stkseek(stkstd,offset);
8:     return(open(stkptr(stkstd,offset),0));
9: }
```

7 libexpr: C expression

Runtime program control is a common feature of many UNIX tools. Much of this is done via so-called *little languages*, such as in *expr*, *find*, and *test*. Although they get the job done, the downside is that these commands often provide incompatible expression syntax for the same basic constructs or worse the same syntax with inconsistent usage. For example, *expr* numeric equality syntax is *num1=num2* while the same syntax is used for string matching in *test*. This leads to confusing expressions such as `0 = 00` which is true in *expr* but false in *test*.

libexpr provides a general approach for runtime expression evaluation based on simple C-style expressions which is familiar to most UNIX users. *libexpr* is the basis for popular *Ast* commands such as *tw* [9], a file tree walk command, and *cql* [7], a flat file database query program. Since this is for command level expression evaluation, there are a few diversions from C. String operands are accepted for `==` and `!=`, and the right operand is interpreted as a *ksh* file match pattern. Each expression context defines a set of expression procedures. For example, the below expression matches all names that end with "`.c`". The `action()` procedure defines what to do on each match; which, in this case, means to issue a message saying that a matched name is found.

```
name == "*.c"
void action()
{   printf("found %s\n", name);
}
```

Interface definitions are defined in `expr.h`. Expressions are interpreted against some parser context of type `Expr_t` which is opened and closed with `exopen()` and `exclose()`. Arguments to `exopen()` define application specific symbols and access functions for reference, and getting, setting, and converting values. Expressions are compiled with `excomp()` and evaluated with `exeval()`.

8 *libpp*: C preprocessor library

Certain major *Ast* tools and systems [6, 4, 13] require C preprocessing. This is hard to get right given the myriad of differences among C dialects, K&R, ANSI and C++, and platform variations. *libpp* provides a single and general interface to deal with all aspects of C preprocessing. A standalone program *cpp* is available which consists of a small `main()` with 30 lines of code to drive *libpp* functions.

There are two main functions, `ppop()` and `pplex()`. The call `pplex()` returns the token id for each fully expanded token in the input files. These ids are suitable for *yacc* grammars, and the library provides the *yacc* `%include` file `pp.yacc` for this purpose. The function `ppop()` sets preprocessor options and states. For example, the call `ppop(PP_PLUSPLUS,1)` enables recognition of `//` comments and the `.*`, `->*` and `::` tokens for C++.

There are over 100 option settings for `ppop()`. This may seem out of hand but it merely reflects the state of C compilation systems. Compiler vendors cannot resist the temptation to extend C. Some PC compilers have more than doubled the number of compiler reserved words (`near` and `far` are just the tip of the iceberg). GNU C and C++ are not far behind. Others add new directives: `#import` in Objective C, `#ident` in System V, `#eject` (to control program listings!) in Apollo C. *libpp* handles this complexity by probing each native compiler (at the first run) and posting the probe information for all users. The probe information includes predefined macros, dialect specific pragmas, non-standard directive and pragma maps, and other non K&R preprocessor reserved words. Probing at run-time to generate pragmas helps maintain a surprisingly stable user and programmer interface. *libpp* has weathered three lexical analyzer implementations, the last one, based on a lexical finite state machine from Dennis Ritchie, brought *libpp* speed within 10% of the K&R "Reiser" *cpp* which is still the most efficient preprocessor for K&R C. Below is an example of predefined macros probed by *libpp*:

```
#pragma pp:predefined
#define __unix 1
#pragma pp:nopredefined
```

From a programming perspective, *libpp* operates in either *standalone* or *compile* mode. The *standalone* mode constructs a text file to pass on to the compiler front end. Macros and include files are expanded. Special line synchronization directives identify included source files and line numbers. Since not all output tokens need to be delineated, the standalone mode skips some ANSI details to be picked up by the next compiler pass. The *compile* mode does full tokenization and hashes all identifiers into the symbol table `Hash_table_t* pp.symtab`. `pplex()` sets `struct ppsymbol* pp.symbol` to point to the symbol table entry for each identifier token. A place holder `void* pp.symbol->value` is available for use by *libpp* users. For example, compilers can use it to hold symbol type and scope information.

Below is an example code fragment to list each C source identifier once (after macro expansion). The `optjoin()` function on line 2 uses the function `ppargs` to subsume command line options required by C preprocessing. If there are any other compiler passes, their option parsers are added after `ppargs`.

```
 1. ppop(PP_DEFAULT, PPDEFAULT);
 2. optjoin(argv, ppargs, NULL);
 3. ppop(PP_COMPILE, ppkey);
 4. ppop(PP_INIT);
 5. while((n = pplex()) )
 6. if (n == T_ID && !pp.symbol->value)
 7. {   pp.symbol->value = (void*)"";
 8.     sfputr(sfstdout, pp.token, '\n');
 9. }
10. ppop(PP_DONE);
```

9 Discussion

The *Ast* libraries have been in use for about 10 years and proved to be a good base for building powerful, efficient and portable applications. Certain components such as *sfio* or *libdict* [11] have always been freely available and have been used widely beyond the scope of *Ast* applications. Other components are also available now. Reference [2] has directions to get them.

The libraries are written in a subset of C that is compatible with all variants of the C language including ANSI-C and C++. One may ask why not just use a language like C++ with better support for encapsulation so that the needs for certain naming conventions can be reduced. However, doing this would have decreased both portability and applicability of the libraries. It takes more work to ensure that the subset of C that we use is adequate for all C variants but this effort is well paid for by the wider applicability.

Basic parts of *Ast* such as the portability base have remained relatively stable throughout the years. However, the libraries continue to evolve as new needs arise and new solution techniques are found. New libraries such as *vmalloc* [14] for generalized memory allocations are occasionally added. The design principles and conventions outlined in Section 2 have been extremely useful in shaping the design and continuing examination of the libraries and to maintain consistency across them. However, these are only guidelines, not rules; and they do not provide all the answers. The main lesson that we have learned in this effort is that there is no simple road toward building reusable software. Useful libraries are built out of necessity. Care must be taken to make them fit into the existing framework. Then, continuing effort is required to chisel and refine them until their essence is revealed and their applicability fully realized.

REFERENCES

[1] AT&T. *Unix System V Release 4 Programmer's Reference Manual*, 1990.

[2] Editor B. Krisnamurthy. *Practical Reusable Unix Software*. John Wiley & Sons, Inc., 1995.

[3] Morris Bolsky and David G. Korn. *The KornShell Command and Programming Language*. Prentice-Hall Inc., 1989.

[4] Yih-Farn Chen. The C Program Database and Its Applications. In *Proceedings of the Summer 1989 USENIX Conference*, pages 157–171, June 1989.

[5] Computer Science Division, University of California, Berkeley. *UNIX Programmer's Manual, 4.3 Berkeley Software Distribution*, April 1986.

[6] Glenn S. Fowler. The Fourth Generation Make. In *Proceedings of the USENIX 1985 Summer Conference*, pages 159–174, June 1985.

[7] Glenn S. Fowler. cql – A Flat File Database Query Language. In *Proceedings of the USENIX Winter 1994 Conference*, pages 11–21, January 1994.

[8] Glenn S. Fowler, David G. Korn, J. J. Snyder, and Kiem-Phong Vo. Feature-Based Portability. In *VHLL Usenix Symposium on Very High Level Languages*, October 1994.

[9] Glenn S. Fowler, David G. Korn, and Kiem-Phong Vo. An Efficient File Hierarchy Walker. In *USENIX*

Summer 1989 Conference Proceedings, pages 173–188, Baltimore, MD USA, 1989. USENIX Association , Berkeley, CA , USA.

[10] David G. Korn and Kiem-Phong Vo. SFIO: Safe/Fast String/File IO. In *Proceedings of Summer USENIX Conference*, pages 235–256. USENIX, 1991.

[11] Stephen C. North and Kiem-Phong Vo. Dictionary and Graph Libraries. In *Proceeding of Winter USENIX Conference*, pages 1–11. USENIX, 1993.

[12] Posix - part 1: System application program interface, 1990.

[13] David S. Rosenblum. Towards a Method of Programming with Assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104. Association for Computing Machinery, May 1992.

[14] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. 1994. Available from the author.

[15] Kiem-Phong Vo. Writing reusable libraries with discipline and method. 1994. Available from the author.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery.To copy otherwise, or to republish, requires a fee and/or specific permission.
SSR '95, Seattle, WA,USA
© 1995 ACM 0-89791-739-1/95/0004...$3.50