A Preprocessor for Icon*

Kelvin Nilsen

TR 88-1

ABSTRACT

Programming language preprocessors are important source-code development tools because they simplify source-code configuration and contribute to a programmer's expressiveness in describing portable and efficient data abstractions. To satisfy the growing numbers of Icon users who are applying Icon to real programming problems, a preprocessor with a knowledge of Icon syntax is now provided with the Icon system. The Icon preprocessor has been fashioned after the well-known C preprocessor [1]. This document describes the Icon preprocessor and discusses suggestions for how it might be used to facilitate development of Icon programs.

December 29, 1987

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8502015.

A Preprocessor for Icon

1. Tutorial Introduction

The Icon [2] preprocessor is simply a special-purpose editor that performs editing on source files before those files are translated by the standard Icon translator. To minimize the cost of preprocessing, the set of editing instructions that the preprocessor is capable of performing is limited to instructions that require only a single forward pass through the file of source code. One of the editing capabilities of the preprocessor, for example, is to replace all occurrences of a particular word with some string of text. Another capability is to selectively include sections of a source file that are appropriate only for a particular configuration of the code.

The Icon preprocessor executes as a filter, reading Icon source files from its standard input and writing the results of editing to its standard output. Unlike most interactive editors, the preprocessor does not read the entire source file into memory or buffers before processing editing instructions. Instead, the preprocessor maintains in memory only a list of the editing instructions that apply to a particular region of the source file and a small buffer that holds the portion of the file that is being preprocessed. As lines are read in, they are modified according to this list of editing instructions and written out.

Editing instructions to the preprocessor take the form of preprocessor directives. All preprocessor directives begin with a '$' character in the first column of a new line. Immediately following the '$' is a keyword specifying the type of editing instruction. For example, the $include directive denotes that its file argument should be inserted into the input stream at the point of this directive. This is frequently used to include existing libraries of Icon source into the current file or to include global declarations or more editing instructions for the preprocessor. The line below causes the preprocessor to insert the file snobol.lib into the current file.

```
$include "snobol.lib"
```

Another editing instruction offered by the preprocessor globally replaces a particular identifier with some string. This editing instruction is specified using the $define directive, as shown below.

```
$define LIB_PATH "/usr/lib"
```

After reading the above statement, the preprocessor replaces all subsequent occurrences of LIB_PATH in which LIB_PATH appears as a complete token with the string "/usr/lib". Note that the quotes surrounding /usr/lib are included in the definition of LIB_PATH. The preprocessor treats runs of alphabetic, numeric, and underscore characters starting with either an alphabetic or underscore character as identifier tokens. In the line below, the preprocessor would not replace LIB_PATH because it does not appear as a complete token.

```
library := open(USER_LIB_PATH || "/lib.1")
```

The editing capability described above is called macro expansion. The $define directive can also be used to define parameterized macros. The Max macro defined below, for example, has parameters a and b.

```
$define Max(a,b)            (((b) < (a), (a)) | (b))
```

Following execution of the above directive, all subsequent occurrences of the identifier Max appearing in the file are expanded according to the definition above. The preprocessor expects all occurrences of Max to be accompanied by a two-item parameter list. The following statement,

```
x := Max(3.5, y+6)
```

for example, expands into:

```
x := (((y+6) < (3.5), (3.5)) | (y+6))
```

If Max occurs without exactly two parameters, the preprocessor treats this as a fatal error.

Parameterized macros are often preferable to procedure calls because they are potentially faster to execute and because they offer different abstraction capabilities. For example, Icon parameters are passed by value. But macros can simulate some aspects of pass-by-reference semantics. Consider the following usage of the Max macro:

```
Max(x, y) := -1
```

Another common use of the preprocessor is to selectively include certain sections of code from a source file. For example, the following might be found within a program designed to run both on UNIX[1] and smaller single-tasking computers.

```
$ifdef UNIX
  file_list := open("ls -l", "pr")
$else
  file_list := &null
  write("Unable to list files on this computer")
$endif
```

In the above section of code, if a macro named UNIX is defined, the first line is included and the others are omitted by the preprocessor. Otherwise, the first line is omitted and the other two are included.

2. Invocation of the Icon Preprocessor

The standard Icon translator icont does not automatically invoke the preprocessor. To invoke the preprocessor, specify the -p command-line option to the icont program. To translate, for example, program.icn using the preprocessor option of icont, issue the following command:

```
icont -p program.icn
```

Alternatively, the preprocessor can be invoked as a stand-alone program by executing the command iconpp. iconpp processes its single file argument if a file name is specified on the command line. Otherwise, iconpp preprocesses its standard input. In either case, the results of preprocessing are written to standard output.

With either the icont or iconpp invocation, all -D and -I command-line flags are interpreted by the preprocessor. The -D flag is used to define macros on the command line. The argument to the -D flag is a single string. Spaces within this string must be enclosed within quotes[2]. If the string includes an equal sign, the remaining portion of the string is treated as the definition for the variable. Otherwise, the definition is set to 1. This might be handy if, for example, a single set of source files contains code for multiple versions or configurations of a program. Depending on the combination of command-line definitions specified at translation time, the translator can produce many different versions of a program without the programmer ever having to modify any source code. The invocation below might be used to compile a demonstration version of a program that is intended to run for 30 seconds and then quit.

```
icont -p -DDemo -DTimeLimit=30 program.icn
```

The command-line definitions shown above produce the same effect as the two $define directives below:

```
$define Demo        1
$define TimeLimit   30
```

The -I command-line flag gives users control over the paths searched to locate the file arguments of the $include directive. The file name arguments to the $include directive may be specified as either absolute path names or as relative file names[3]. Wherever a relative path name is supplied as the argument to a $define directive, the

[1] UNIX is a trademark of AT&T Bell Laboratories.

[2] This description assumes that command-line parameters are processed as they are under the UNIX operating system. Some of the computers on which the preprocessor runs do not use these quoting conventions for command-line arguments. For such computers, quoting behavior depends on the C implementation that compiled iconpp.

[3] The syntax for relative paths and host file names depends on the host operating system and is specified for several implementations in the appendix.

preprocessor searches for the file relative to a list of possible directories. By default, this list includes a small number of directories that are specified when Icon is installed on a particular machine. The initial value of this list can be overwritten by setting the environment variable PPATH to represent a new list of search directories. The value of PPATH should be a blank-separated string of the form $p_1\ p_2 \ldots p_n$ where each of the p_i names a directory. The directory name arguments that accompany each -I flag are appended to the end of the search list. The example that follows assumes that some of the files included by the specified Icon source files are found in the directory /usr/icon/include.

```
icont -p -I/usr/icon/include main.icn util.icn lib.icn
```

3. Command Reference

The editing commands supported by the preprocessor can be grouped according to functionality. There are four general functions provided by the preprocessor:

Global replacement of an identifier with a possibly parameterized replacement string is called macro replacement.

Conditional compilation refers to the preprocessor's ability to determine that certain blocks of code should be compiled only if certain conditions are satisfied.

The preprocessor's ability to insert files into the current input stream is referred to as file inclusion.

Because the preprocessor may insert or delete lines from the original Icon source, and because the translator needs to know the original number of each line of source to provide helpful error messages, a method of communicating original line numbers to the translator has been devised. This is referred to as line numbering.

The last topic discussed in this section is the existence of several implementation-dependent predefined macros. Programmers can use these predefined macros to write programs that automatically yield different configurations when compiled in different environments.

3.1 Macro Replacement

Macros are defined using the $define directive, which has two forms. The first form is used to define an unparameterized macro.

$define *name sequence-of-tokens*

The example above defines a macro of the specified *name*. Subsequent occurrences of the identifier *name* in the input stream are replaced by the specified sequence of tokens. In the above, *sequence-of-tokens* may be empty.

Other macro definitions by the same name in existence at the time this directive is executed are hidden on a stack beneath this definition until this particular macro definition is erased. The second form of the $define directive is used to define parameterized macros:

$define *name*($name_1$, $name_2$, ..., $name_n$) *sequence-of-tokens*

In the line above, there must be no white space[4] between *name* and the left parenthesis that follows it. It is an error for any subsequent occurrence of *name* in the input stream to not be parameterized with exactly the number of parameters (n) as are specified in the macro definition. Each occurrence of *name(parameters)* in the input stream is replaced with *sequence-of-tokens* after substituting within that sequence the actual parameter values for each parameter name. Each actual parameter, which is separated from other parameters by commas, may be a sequence of tokens itself. Commas enclosed in nested parentheses are not interpreted as boundaries between parameters. For example, the following invocation of the Max macro, which was defined above, is expanded using 100 as the value of the first parameter and acker(5, 8) as the value of the second. Note that the second parameter of the expansion includes a comma enclosed in parentheses.

[4] White space refers to tab, space, newline and carriage return characters.

```
y := Max(100, acker(5, 8))
```

As with the first form of this directive, other macro definitions by the same name in existence at the time this directive is executed are hidden on a stack beneath this definition until this particular macro definition is erased. The convention of hiding other macro definitions on a stack facilitates the creation of macro definitions that are restricted in both scope and side effects[5].

In both the parameterized and unparameterized versions of this directive, the name of the macro must be separated from the $define keyword by at least one tab or space character. The macro and parameter names must consist only of alphabetic, numeric, and underscore characters. No restrictions are placed on the length of macro and parameter names. However, very long names (above 100 characters) may result in fatal errors because of buffer overflows when the macros are expanded. Tabs and spaces separating parameter names from each other and from commas and parentheses are allowed.

Following specification of the macro pattern (the name by itself for unparameterized macros, or the name and parameter list for parameterized macros), spaces and tabs up to the first non-white character are ignored. *Sequence-of-tokens* is parsed using the following rules:

1) If the '#' character is seen, the remainder of the line is treated as a comment. The macro definition is terminated at this point, and the comment is ignored by the preprocessor.
2) If the '\' character is seen and the following character is a newline[6], the macro definition is extended to the line that follows. If the following character is not a newline, the '\' character is simply added to the verbatim replacement string for the macro.
3) Runs of alphabetic, numeric, and underscore characters are treated as identifiers. Each identifier within the sequence of tokens that is the same as one of the macro's parameter names is flagged for replacement at macro expansion time. If an identifier is not recognized as the name of a parameter, then the identifier is added to the verbatim replacement string for the macro. All identifiers are case sensitive so, for example, the preprocessor treats Max and max as distinct tokens.

The rules above have the effect of separating the replacement string into identifiers to be replaced at macro expansion time, verbatim text to be inserted when the macro is expanded, and comments.

Notice that the replacement string for a macro is not modified by the preprocessor. This means, for example, that the definition of y after execution of the two $define directives below is x:

```
$define x z
$define y x
```

The $erase directive has the effect of removing a macro that was already defined. Invocation of this directive with a name that is not defined is treated as a fatal error. The following line erases the most recent definition for the macro Compare. If the most recent definition of that name hid an earlier definition, the older definition is revealed by the $erase directive.

```
$erase Compare
```

There is a special preprocessor directive, $undef, that has the effect of removing all macro definitions of a particular name. This is especially useful when macro names are used to control conditional compilation. A configuration file, for example, might contain the following:

```
$undef      VAX_VMS      # define to obtain VAX VMS configuration
$define     IBM_PC       # define to obtain IBM PC configuration
```

[5] Contrast this with many C preprocessors in which the scope of a defined macro may be restricted through use of the #undef directive, but the side effects of the #undef directive persist to the end of the preprocessed file.

[6] For computer systems that do not use the Unix convention of separating consecutive lines of text with a single newline character, this discussion assumes that the preprocessor opens files of source in a translate mode, such that the preprocessor sees only a newline character to mark the boundary between two lines.

3.2 Conditional Compilation

By compiling certain code only if a certain named macro has or has not been defined, programmers can, among other things, combine several versions of a program into a single source file. Within the source file, text that is to be conditionally compiled is enclosed between $ifdef or $ifndef directives and $endif directives. $ifdef is an abbreviation for "if defined". $ifndef stands for "if not defined". The discussion that follows focuses on the behavior of the $ifdef directive. The $ifndef directive behaves identically, except the effects of the conditional tests are reversed. The general template for the $ifdef directive is shown below:

$ifdef *name*

Each $ifdef directive must be followed by an $endif directive. Any code appearing between the $ifdef and the $endif directives is included in the output of the preprocessor only if the *name* argument of the $ifdef directive is the name of a defined macro. An optional $else directive may appear between the $ifdef and $endif directives. If the $else directive is present, all code between the $ifdef and the $else directives is included in the preprocessor output only if the name argument of the $ifdef directive is a defined macro. If, however, the name argument is not a defined macro, the code up to the $else directive is omitted from the preprocessor output and whatever code appears between the $else and the $endif directives is included. Any text appearing on the same line as the $else or $endif directives is ignored by the preprocessor. It is customary to place the same identifier following the $else and $endif keywords as appears on the $ifdef or $ifndef that opens the conditionally compiled block of code. This information, which is treated as a comment, aids readers of the source code.

Blocks of conditionally compiled code may be nested within each other as shown below:

```
$ifdef VMS
 helpfile := "UA_COMMON:[UAHELP.BITNET]"
$ifdef Demo
 timeout := TimeLimit;
 banner := "Demonstration Version for VMS"
$endif Demo
$endif VMS
```

3.3 File Inclusion

Files may be included in other files using the $include directive. There are two forms of this preprocessor directive:

$include "*file*"
$include <*file*>

The search for a file specified by a relative path name enclosed in quotes begins with the directory of the file that issued the $include directive. If the file is not found relative to that directory, the search proceeds to consider, in order, each of the directories on the list of search directories until it either finds the desired file, or exhausts the list. The search for a relative path name enclosed in angle brackets executes similarly except the search begins with the search list, skipping the directory of the file that issued the $include request. In either case, if the preprocessor is unable to find the specified file, it aborts with a fatal error message.

The initial value of the search list is specified when the Icon system is installed. This initial value can be overwritten by setting the environment variable PPATH to represent a new list of search directories. The value of PPATH should be a blank-separated string of the form $p_1\ p_2 \ldots p_n$ where each of the p_i names a directory. Additional directories can be added to the search list by specifying them on the command line of the icont invocation as arguments to the -I flag.

3.4 Line Numbering

In order for the Icon translator and the run-time system to associate error conditions with the original source line number of the offending code, it is necessary for the preprocessor to communicate to the translator the location of certain code in terms of file name and line number. Note that the preprocessor may combine source code from many different files into a single stream to be processed by the translator. The $line directive is designed to serve this purpose. The preprocessor inserts this directive into its output stream to inform the translator that the code that

follows appears in the original source at a particular line number of a particular file. The general form of this command is shown below:

```
$line num "file"
```

If, for example, the line below appears on line 4 of the file main.icn:

```
$include "snobol.lib"
```

the output of the preprocessor is given by:

```
$line 0 "snobol.lib"
  the contents of the file "snobol.lib"
$line 4 "main.icn"
```

Note that the newline character appearing at the end of this preprocessor directive is significant in that, upon encountering it, the translator increments its internal representation of the current line number. In the example above, the line immediatedly following the last line shown corresponds to line 5 of main.icn. This is communicated to the translator by first setting the line number and file name to 4 and main.icn respectively, and allowing the newline character that terminates the preprocessor directive to increment the line number from 4 to 5.

Besides inserting the $line directive into its output, the preprocessor also understands and acts appropriately whenever this directive occurs in its input stream. This permits, for example, the preprocessor to filter a single source file more than once as the file is prepared for translation.

It is also conceivable that tools might become available to generate Icon source according to user specifications. Frequently, the code from this sort of tool is combined automatically with user-supplied code, possibly even within the same file of Icon source code. An example of this sort of application is the UNIX YACC[3] program, which takes as input a context-free grammar and fragments of C source code and produces a C program that combines the user-supplied C source code with an automatically generated parser for the context-free grammar written in C. In this application, it is helpful for the C compiler to know the origin of certain blocks of code in terms of the original specifications that were supplied to the YACC program. YACC communicates this information to the translator by inserting the equivalent of $line directives into its output, which becomes the input to the preprocessor. Developers of similar tools for Icon are free to use this same approach, placing $line directives in the Icon source they generate.

3.5 Predefined Macros

Some features of Icon are not available in all implementations of Icon. Other capabilities behave differently, depending on the implementation. For example, the "t" mode of the open function is not meaningful in UNIX implementations of Icon. The code below might appear in a source file intended to run on either UNIX or MS-DOS computers.

```
$ifdef UNIX
  f := open(fname, "r")
$else
  f := open(fname, "rt")
$endif
```

Certain environment-specific macros are automatically defined to facilitate the writing of programs that, at preprocessing time, configure themselves for their environment. For each implementation of Icon, exactly one of the following is automatically defined within the preprocessor:

```
$define VMS          # on a VAX computer running VMS
$define MSDOS        # on an MS-DOS computer
$define AMIGA        # on a Commodore Amiga
$define ATARI_ST     # on an Atari ST
$define MACINTOSH    # on an Apple Macintosh
$define UNIX         # on any computer running the UNIX operating system
```

4. Style Conventions and Recommendations

Accompanying the increased expressiveness of programming with a preprocessor come increased complexity to programs and the accompanying increased probability of programmer error. Certain style conventions and concerns are discussed here which help programmers avoid some of the common pitfalls of programming with a preprocessor.

Because parameterized macro expansions appearing in the source code use the same syntax as procedure and function invocations, the readability of preprocessed source code occasionally suffers. Consider, for example, the definition of a macro named zap that might appear in some included file:

```
$define zap()        (tab(many(' \t')))
```

Invocations of zap found throughout the source might lead a reader of the code to believe that a procedure named zap has been defined. However, the reader would not be able to find the definition of any procedure by that name. Also, because certain types of expressions, which are mentioned below, are not recommended for use as macro parameters, it is important for the programmer to easily recognize the distinction between a procedure call and a macro invocation.

To reduce both of the above-mentioned sources of error or confusion, it is recommended that the designer of an Icon program adopt some consistent convention that distinguishes procedure invocations from macro expansions. One convention, for example, is to give macros names that at least begin with an uppercase character and to use only lowercase characters for procedure names.

For similar reasons, defining macros on the icont or iconpp command line is not recommended unless documentation of some sort is provided. One method of documenting this usage is to place a batch script or UNIX-style makefile in the same directory as the source code for a particular application.

When defining parameterized macros, care must be taken to ensure that expansion of the macro using actual parameters not violate the macro's original intent. One characteristic of macro expansion that programmers should take care to avoid is that each of the macro parameters may be replicated in the result of macro expansion. If the parameters are generating expressions or expressions that otherwise may produce side effects, unwanted replication of the side effects may occur when the macro is expanded. The following example demonstrates this problem. The intention of the RecEq macro defined below is to test for record equality. In this example, record equality is defined by the programmer as equality by reference of the id fields in each record.

```
$define RecEq(e1, e2)        e1.id === e2.id & e2
```

If the RecEq macro is used in the following context, its original intent is violated.

```
if RecEq(read_rec(), !rec_list) then
  write("duplicate record")
```

The problem is that the preprocessor expands the above source into:

```
if read_rec().id === !rec_list.id & !rec_list then
  write("duplicate record")
```

which is clearly not what was desired. In general, programmers should avoid the use of generators and other expressions producing side effects as macro parameters. Alternatively, it is sometimes possible to define macros that avoid this problem:

```
$define RecEq(e1, e2)          e1.id === 2(tmp := e2, tmp.id) & tmp
```

Another precaution to take in designing reliable macro definitions is to enclose within parentheses all occurrences of parameters in the macro expansion. This is necessary because the parameters may be expressions within which the relative precedence of operations might be affected by the context of the macro expansion. Consider a macro written to perform unit conversions from inches to feet.

```
$define Feet(inches)    inches / 12.0
```

Because division has higher precedence than addition in Icon, this expression produces undesired results in the following:

```
write("5 + 7 inches makes: ", Feet(5+7), " feet")
```

Instead of writing 1, this line outputs the number: 5.5833333! To fix this, enclose inches in parentheses:

```
$define Feet(inches)    ((inches) / 12.0)
```

Similar problems may occur when operators surrounding the macro expansion interact in an unexpected way with the macro expansion itself. For example, since not has a very high precedence in Icon, the following use of the RecEq macro is corrupted by its context:

```
not RecEq(r1, r2) & r1 := r2
```

The Icon translator groups the result of macro expansion as parenthesized below.

```
(not r1.id) === 2(tmp := r2, r2.id) & tmp & r1 := r2
```

To avoid this problem, the entire macro expansion should be enclosed in parentheses. Below is a revised definition of RecEq.

```
$define RecEq(e1, e2)          ((e1).id === 2(tmp := (e2), tmp.id) & tmp)
```

Yet another source of possible preprocessor-induced programmer error results from the preprocessor's limited knowledge of Icon syntax. The preprocessor's concept of an identifier is any string of alphabetic, numeric and underscore characters that begins with an alphabetic or underscore character. This somewhat naive definition treats Icon keywords such as &cset or &subject each as a pair of tokens: an & followed by an identifier. Because of this, it is possible to define a macro by the name of, for example, subject that would replace occurrences of &subject with & followed by the result of macro expansion. Since an Icon programmer sees &subject as a complete token, macro expansion of only subject would probably be unexpected. For this reason, it is best to avoid keyword names when selecting names for macros. Note that a similar problem may occur with floating-point constants. A macro named e4, for example, might produce surprising results if applied to real literals such as 5e4.

5. Theory of Operation

The preprocessor is organized internally as two major components, a lexical analyzer and a parser[4]. Most of the details of preprocessing are dealt with by the lexical analyzer. The parser's principal responsibility is to recognize occurrences of macro names and expand their definitions.

5.1 Lexical Analysis

The first phase of preprocessing consists of lexical analysis. The lexical analyzer searches for certain patterns in the input stream and acts on the patterns seen. For example, the lexical analyzer recognizes Icon comments and removes them. The lexical analyzer also recognizes preprocessor directives and takes responsibility for dealing with them.

Any time the '$' character is seen as the first character of a line, the remainder of the line is assumed to be a preprocessor directive. If the identifier that immediately follows the '$' is not the name of a valid preprocessor directive, the preprocessor aborts with a fatal error message.

The lexical analyzer removes from the input file preprocessor directives after interpreting the editing instructions they contain. The lexical analyzer also removes blocks of code that are enclosed in conditional compilation directives whose conditions are not satisfied. The preprocessor keeps track of the current file name and line number, setting these values whenever a $include or $line directive is processed, and incrementing the line number each time a newline character is encountered. This information is made available to the parser so that error messages generated by the parser can refer to the file name and line number of the offending code. Rules for the parsing of macro definitions are provided in § 3.1, where usage of the $define directive is described in detail.

Other than preprocessor directives, the lexical analyzer has only a few special tokens to recognize. The preprocessor's concept of a token is considerably more relaxed than that of the Icon translator. It makes no attempt to verify proper Icon syntax and may produce output that the Icon translator rejects.

Comments are treated as complete tokens by the preprocessor. Wherever a '#' character that is not a part of some other token is seen, the text that immediately follows up to a newline character is treated as a comment. Comments are removed from the input file by the preprocessor.

When double or single quotes are found, the preprocessor recognizes these as introducing Icon strings and csets respectively. The preprocessor treats the entire string or cset as a single token. It does this by searching for the terminating double or single quote respectively. The preprocessor understands Icon escape mechanisms for special characters such as '\'', '\"', '\n' and '\033', and permits the escaping of a newline to extend a string or cset definition to the following line. If, while processing a string or cset, the last character on a line is an underscore, the preprocessor likewise extends the string or cset definition to the line that follows.

The preprocessor's concept of an identifier is the same as that of Icon; runs of underscore, alphabetic, and numeric characters that begin with either an underscore or alphabetic character are treated as identifiers.

White space is ignored by the preprocessor. Anything not fitting any of the patterns specified above is treated as a token consisting of a single character. As mentioned above, preprocessor tokens are different from Icon tokens. Icon's lexical analyzer would, for example, treat the string := as a single token representing the assignment operator. The preprocessor represents this as two tokens, colon and equals. This is permitted because the preprocessor's parser does not need complete information about the source file. It searches only for macro invocations.

The parser communicates with the lexical analyzer in several ways. The main function of the lexical analyzer is to divide the input file into tokens. The input file is revealed to the parser by the lexical analyzer one complete token at a time. The parser receives each token upon its request to the lexical analyzer.

The lexical analyzer also handles most of the details of editing the input file. This is necessary principally because the parser sees only the list of tokens comprising the input file. The parser does not know, for example, where line breaks or other white space occurs. This is especially significant because the preprocessor's tokens are different from the Icon translator's tokens. Because of this, white space, even though it is ignored by the parser, must be preserved by the preprocessor. Consider, for example, the consequences of inserting a space between the colon and equals tokens of assignment. Consider also what might occur if separating spaces are removed between two identifiers. A second reason it is necessary that the lexical analyzer perform the editing is because the results of macro expansion must be rescanned for further macro replacements by the preprocessor.

The lexical analyzer maintains a buffer which represents the limited region of current interest within the preprocessed file. File data that precedes this window is completely preprocessed and is either written to the preprocessor's standard output, or buffered internally for the purpose of writing to standard output. The parser can edit and view only text that appears in the window. The parser anchors the beginning point of the window by placing markers in the input file. The end of the window is simply the point at which the parser is currently looking. As place markers are shifted or removed, the beginning of the window shifts forward to the next marked point in the file. As the parser requests tokens, the end of the window advances. The lexical analyzer automatically writes to standard output the text that is revealed when the window's starting point advances.

There are only two types of place markers, and only one marker of each type required by the parser. A delete marker is placed when the parser recognizes that a certain section of the original file must be deleted. A request by the parser to place a delete marker positions the marker immediately before the most recently returned token. A save marker is placed when the parser anticipates the need to obtain an exact copy of a contiguous region of text from the

input file. A request to place a save marker positions that marker immediately following the most recently returned token. Having placed the appropriate marker, the parser can issue to the lexical analyzer commands to delete from the delete marker to the current focus, or to save from the save marker to the current focus. In response to the save command, the lexical analyzer returns a pointer to a copy of the saved text.

In expanding macros, the parser removes occurrences of the macro with its parameters, if there are any, and replaces this with the macro's definition. An insert instruction is provided by the lexical analyzer for use by the parser in accomplishing this task. The inserted text is placed immediately following the current focus of the lexical analyzer so that inserted text is scanned as analysis proceeds.

5.2 Parsing

As discussed above, the parser's only responsibility is to expand macro definitions. The parser repeatedly asks the lexical analyzer for a token until it receives an identifier that represents a macro name. The parser places a delete marker at the beginning of that identifier token, and looks up the definition of the macro.

If the definition is unparameterized, the parser simply tells the lexical analyzer to delete from the delete marker to the current focus (which is the character immediately following the identifier). Then the preprocessor inserts the verbatim replacement text for the macro.

If the macro definition is parameterized, the preprocessor parses the parameter list. It does this by first looking for the opening parenthesis, then placing a save marker and scanning forward for either a comma or a closing parenthesis. While searching for the end of a particular parameter, the parser maintains a count of the number of parentheses opened but not closed. This count must be zero in order for a comma or closing parenthesis to be recognized as the end of a parameter. When the end of a particular parameter is found, the parser tells the lexical analyzer to save from the save marker to its current focus. The string that is returned by the save instruction is stored by the parser on a list of actual parameters. If the token following the parameter is a comma, the parser repeats the above process, looking for the next parameter. Otherwise, the next token is the closing parenthesis that terminates the list of parameters. After all parameters have been parsed, the parser checks to be sure that the number of parameters found matches the macro's definition. Then it tells the preprocessor to delete from the delete marker to the current focus and to insert the result of macro expansion.

Note that, in both the unparameterized and parameterized forms of macro expansion, the result of macro expansion is rescanned by the preprocessor for further substitution. Suppose the following is found in an Icon source file.

```
# status codes
$define OK                  1
$define FIRST_ERROR         2
$define END_OF_FILE         (FIRST_ERROR + 0)
$define BUFFER_OVERFLOW (FIRST_ERROR + 1)

$define Eof(f)              ((f).status = END_OF_FILE)

  if Eof(file1) then
    write("can't read data")
```

The lexical analyzer strips the comments and stores the macro definitions. The first token seen by the parser is the identifier if. The next token is the identifier Eof. The parser recognizes Eof as the name of an existing macro, and expands it. Since file1 is the parameter to Eof, the result of expanding the macro is:

```
  if ((file1).status >= END_OF_FILE) then
    write("can't read data")
```

The replacement is then scanned by the preprocessor. When the lexical analyzer reaches the identifier END_OF_FILE, the parser looks this up and finds that it is an unparameterized macro. Upon replacing this identifier with its expansion, the code becomes:

```
  if ((file1).status >= (FIRST_ERROR + 0)) then
    write("can't read data")
```

Scanning proceeds with the left parenthesis immediately to the left of FIRST_ERROR. When FIRST_ERROR is

encountered by the parser, it is replaced by 2, yielding:

```
if ((file1).status >= (2 + 0)) then
  write("can't read data")
```

6. Acknowledgements

The design of the Icon preprocessor strongly resembles the C preprocessor. As such, I am indebted to those who designed it. All members of the Icon group at the University of Arizona have discussed the design and implementation of this preprocessor and have read and commented on at least one version of this document. The group consists of Ralph Griswold, Dave Gudeman, Bill Mitchell, Janalee O'Bagy, Gregg Townsend, and Ken Walker. Gregg Townsend has been especially helpful in providing a very detailed commentary on an earlier draft of the iconpp documentation. Ralph Griswold has helped by reviewing several drafts of this documentation and suggesting a variety of improvements, especially regarding this paper's scope and focus.

References

1. S. P. Harbison and G. L. Steele Jr., *C: Reference Manual*, Prentice Hall, 1984.
2. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
3. S. C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, New Jersey, 1978.
4. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley Publication, 1976.

Appendix
System Dependent File Naming Conventions

The preprocessor, in dealing with an $include directive, must determine whether the actual file name argument is a relative or absolute file name. If the argument is a relative file name, the preprocessor must combine that name with each of the directories on its search list as it attempts to locate the file for inclusion.

The rules for distinguishing between relative and absolute file names are system dependent, as are the rules for combining directory names from the search list with relative file names to obtain absolute file names. In some cases, the conventions supported by the preprocessor do not allow all of the generality available from the host operating system. However, the preprocessor attempts to provide at least one way to name each possible file.

MS-DOS

The MS-DOS version of the preprocessor treats both forward and backward slashes as delimiters between directory levels. If the second character of a file name is a colon, the first character is treated as a drive name. Drive names may be specified for either relative or absolute file names.

If the character immediately following the optional drive name and colon in a file name is either a forward or backward slash, the file is treated as an absolute path name. If the drive name is omitted from an absolute file name, the current drive serves as the drive for the absolute path.

Under the MS-DOS operating system, the concept of a current working directory represents a different directory for each available drive. Relative file names specified with a drive name must be found relative to the current directory for that drive. Such file names are simply handed to the open function of the C run-time system. No attempt is made to combine relative file names of this form with any of the directories named on the search list.

Relative file names without a drive name are combined with directory names by appending a forward slash and the relative file name to the end of the directory name. When combining a relative file name of this form with the current directory, the relative file name is only combined with the current directory of the current drive.

UNIX

Under the UNIX operating system, any file name beginning with a '/' is treated as an absolute file name. All other files are assumed to be relative path names.

Directory names are combined with relative path names by appending a '/' to the end of the directory name and catenating the result with the relative path name.

VMS

File names under the VMS operating system are considerably more complex than the UNIX file names that the preprocessor was originally designed to handle. Further, the C programmer's interface to the VMS operating system provides slightly different capabilities than are available to UNIX programmers. Instead of attempting to define rules that allow the preprocessor to distinguish between absolute and relative file names, all file names are assumed to be relative file names. In processing $include directives, the preprocessor conceptually sets the current working directory to each of the directories in its search list, and for each directory attempts to open the specified file, proceeding to the next directory on the list if the open fails and there are more directories on the search list. The operating system ignores the current directory's state when opening files specified with absolute path names. If the file exists, the first attempt to open that file succeeds, regardless of what directory is current. If the file does not exist, all attempts to open the file fail, and the preprocessor aborts with a fatal error message.