6. Reusable Incremental C Preprocessing, Part I

W.M. McKeeman & Shota Aki
Digital Equipment Corporation
110 Spitbrook Road
Nashua, NH 03062

Abstract
We present the design of a preprocessor for standard C that is independently reusable and incremental. The preprocessor itself uses reusable components for scanning, diagnostics, and string management. Reusability is achieved by separation of function, simple and consistent interfaces, and provision of component-level testing. The preprocessing algorithms presented here are constrained by the incremental processing requirement. Incremental capability is achieved by saving lists of post-pp tokens corresponding to blocks of source lines. When the source is unchanged and all macro definitions used in a block are unchanged, the saved post-pp tokens are reused instead of repeating the preprocessing. Changes in macro definitions are detected by journaling macro table activities, analogous to a common error recovery technique.

Introduction
We introduced reusable and incremental compiler components in earlier papers[8, 5]. The information in those papers is used here without further explanation. Here we apply the same approach used for an incremental scanner to preprocessing. As before, the design focuses on the desire for speed to enable incremental processing downstream.

Part I of this paper proposes a regular interface to a preprocessor for Standard C to achieve reusability. The -E flag on a typical C compiler suggests the reuse of preprocessor output; subsequent runs could reconstitute the post-preprocessor source text instead of merely starting over and feeding the back end.

The proposal here differs from traditional K&R Reiser-style preprocessors[2, Chapter 16][4][3] in that it is a token-stream-to-token-stream transformation

3Knowledge about C preprocessors is most effectively spread through source listings of existing preprocessors, not the published literature. An early version of the MetaWare Hi C documentation also contains useful information.
instead of a source-to-source transformation. The source text abstraction (used by the scanner) is invisible to the preprocessor. The required demonstration of adequacy (part of our reuse strategy) assumes the existence of a stand-alone test for the preprocessor which exercises all of its functions. By stand-alone we imply that not even the scanner is used—a scanner stub in the test jig supplies token input.

Our approach to non-incremental preprocessing is consistent with that reported in a recent case study[6], except that we do not combine scanning and preprocessing. This paper presumes the reader has access to a description of Standard C[1].

Part II of this paper (to be published in the next issue of The Journal) extends the interface to provide for incremental preprocessing, something that does not seem to have been reported in the literature. The principal innovation, attaching journals of interface activity to the incremental structures, has application to a broad class of algorithms, in particular to all compiler processes after preprocessing. There is a supporting innovation to avoid gratuitous changes in the interface values which would otherwise cause unnecessary re-preprocessing.

Preprocessor Tasks

The preprocessor treats input text in a few stylized ways. All text must consist of acceptable preprocessing-tokens. The \#include directive and non-directive source can cause tokens to be passed to the preprocessor output. The preprocessor interprets the other tokens to control its behavior.

The preprocessor operates as a separate pass. After it is called, it returns leaving the needed post-preprocessing tokens in its internal state, awaiting calls to its output interface.

The functions of our preprocessor are separated into three relatively independent subcomponents: the pp-directive module, the pp-macro module, and the pp-expression module. Figure 1 illustrates the flow of tokens between these and auxiliary components. Each of these major subcomponents has a grammar and parser.

The pp-directive module handles the preprocessor input and output (token sequences) and recognizes directives. When appropriate, the pp-directive module calls the pp-macro module for macro definition or expansion. Within the pp-macro module the tokens are kept as a linked list to facilitate elision and replacement for macro expansion. Macro-expanded output may be further interpreted by the pp-directive module. The pp-directive modules uses the pp-expression module to evaluate constant expressions for \#if or \#elif. There is no direct connection between the pp-macro module and the pp-expression module.

---

Because a compiler must behave as though phase 3 of translation is completed before phase 4, the preprocessor will never encounter text that cannot be scanned[1, section 2.1.1.2]. This turned out to be essential for our need to separate incremental scanning from incremental preprocessing.
If the text is on the false side of a conditional construct such as `#if`, the tokens of that text must be unexamined. This "blindness" does not apply to the first token on a line or the token following an initial `#` (called the preprocessor directive keyword,) because the nesting of the conditional directives must be tracked regardless of whether they are on the true or false side. Non-standard directives such as `#token` must be diagnosed, even on the false side of a conditional construct.

If an `#include` directive is encountered (except in unexamined lines,) the pp-directive module must recur to insert and process the header file.

Table 1 summarizes the regions of input and the kind of processing to be done in each. The two columns are for the true and false sides of conditional inclusion (`#if` and friends.) The entries are:

- **interpret** The preprocessor will use the information directly.
- **empty** No tokens are allowed (although comments and white space are.)
- **expand** The text must pass through the macro expander.
- **recur** The preprocessor must process the body of the include file before returning to the current task.
- **unexamined** Nothing, even ill-formed input, is to be examined on the rest of this logical line.
- **pass** Tokens will get through to the output. Our choice of expanding the body of a `#pragma` is arbitrary.
### Table 1: Actions in Preprocessing Regions

<table>
<thead>
<tr>
<th>key</th>
<th>if</th>
<th>ifdef</th>
<th>ifndef</th>
<th>elif</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define...</td>
<td>interpret</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#undef...</td>
<td>interpret</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#include...</td>
<td>expand, interpret, recur</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#if...</td>
<td>expand, value, interpret</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#elif...</td>
<td>expand, value, interpret</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#ifdef...</td>
<td>interpret</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#ifndef...</td>
<td>interpret</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#else</td>
<td>empty</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#endif</td>
<td>empty</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#error...</td>
<td>interpret</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#line...</td>
<td>expand, interpret</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#pragma...</td>
<td>expand, interpret</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>empty</td>
<td>empty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>source</td>
<td>expand, pass</td>
<td>unexamined</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Preprocessor Input and Output Interface

We present the preprocessor interfaces as a set of reusable abstractions, including the form of, relation between, and naming conventions for these abstractions. Each abstraction is based on a principal data structure. Each data structure's name also names the abstraction and prefixes functions defined in the abstraction. These conventions are used throughout without further mention (see pp. below, for example.) We also use a convention which we call struct-wrapped handles to enhance the reusability of our abstractions[5].

The C standard uses the term *preprocessing-token* in the description of translation phases 1–7 and *token* in the description of phases 7–8[1, section 3.1]. For Standard C, because of escape sequence conversion in strings, there may not be a source representation after the completion of phase 5—the traditional -E compiler flag therefore best presents the result of phase 4.

The scanner implements phases 1–3. The preprocessor implements phase 4. Phases 5 and 6 can be deferred until the string content must be interpreted. The transformation from *preprocessing-token* to *token* in phase 7 is most conveniently done as part of preprocessor output activity, and therefore perhaps prior to implementing phases 5 and 6.

The preprocessor assumes the existence of a scanner to provide it with its input. The scanner-output interfaces described in our earlier paper[5] are used for input here, specifically the interfaces for *token.h*, *string.h*, and *tokinc.h*. 

---

*The Journal of C Language Translation – September, 1992*
typedef struct {private layout} Pp

iterator functions

p = PpFirst()  start at head of the input
p = PpNext(p)  step to next token
p = PpPrev(p)  step back to previous token

attribute functions

c = PpClass(p)  small integer classifying token p
h = PpLoc(p)    source file location of token p
s = PpString(p) string representation of token p

Table 2: Preprocessor Output Functions

The preprocessor uses tokinc.h only if it needs to process its input incrementally. The preprocessor also assumes one new abstraction for diagnostics.

Preprocessor Token Abstraction

The preprocessor token abstraction is implemented in module pp.c with public interface pp.h. It is the principal data type produced by the preprocessor and is named Pp. All uses of Pp except for declarations, assignments, and parameter passing are private to the preprocessor. The values of type Pp are referred to as post-preprocessor token handles. These names are not consistent with the names in the C standard, because reusable modules need names that are not tied to the consumer (as is the preprocessing-token which describes input to the preprocessor.)

The form of this interface is given in Table 2. It is similar to that of our scanner's token abstraction.

The iterator functions provide the primitive navigation operations out of which a scanner jacket can be built. Traditional parser use (requiring a look-ahead of just one token) can be built without using PpPrev(p) at all by keeping one token ahead.

Function PpClass(p) returns the token class for Pp. The token class must distinguish C keywords, identifiers, integers, floats, character constants, and string literals. The values of PpClass(pp) need not be identical to the values of TokenClass(pp) produced by the scanner[5] since scanner tokens are not visible to any compiler process past the preprocessor.

Function PpLoc(p) returns a value of type Locator which is a further abstraction providing access to token starting and ending position, filename and line, and other information associating tokens and source text position. Function PpLoc(p) is used in compilers principally to place diagnostics relative to the source text. The analogous function TokenLoc(t) supplied by the scanner is simpler—the scanner sees only what is actually in the source text and therefore can locate items in it easily. Preprocessor output, on the other hand, can
have tokens from the source file, include files, macro actuals or macro bodics, stringizing or token pasting, or combinations of them. Every post-pp token is built out of whole source tokens, but more than one token may have been used and more than one point of construction may have been involved. All tokens used and all points of construction or insertion can be found via the locator. The preprocessor must go to some trouble to gather all this information.

Function \texttt{PpString(p)} has type \texttt{String[6]}. The value of this function provides the text of the token itself. The preceding white space, available from the scanner, need not be passed on by the preprocessor for use in a compiler. On the other hand, for other uses, such as pretty printers, the intervening comments must be passed along.

**Preprocessor Input Description**

The input to a preprocessor has two forms: preprocessor directives (starting with \#) and C program text. All program text and some text within directives is subject to macro expansion. The preprocessor is responsible for processing appropriate to the input forms, providing predefined information and diagnosing input errors.

Program text is a sequence of tokens which may contain macro invocations. One way to describe the preprocessor input is to ignore preprocessor directives, leaving them to be screened out by a preprocessor scanner (a jacket in the preprocessor, using the scanner \texttt{TokenNext(t)} routine.) Whenever the actual value of \texttt{TokenNext(t)} is a \# that begins a preprocessor directive, the preprocessor scanner does not return until it has carried out the directive, including all other directives coupled with it. For example, once it sees \#if, the preprocessor scanner retains control until it carries out the matching \#endif directive. Within the preprocessor scanner, more program text can be encountered (for example, between \#if and \#elif.) The preprocessor is then called recursively. The output from the preprocessor, on all levels of recursion, is combined in a single token stream. The definition of C has specific language enabling this approach—all scanning can be entirely done before any preprocessing, nothing nasty is allowed to break across include file boundaries, and nothing suspicious is allowed inside the parameter lists of macro invocations[1, Section 3.8].

Another way to describe preprocessor input combines the description of program text and preprocessor directives. The grammar for a \textit{preprocessing-file} in the C standard presents this view. So does this paper, because it also leads more naturally to line-granularity incremental processing.

**Preprocessor Input Grammar**

The grammar used here is an elaboration of the one in the C standard and follows its form. This grammar is strictly context-free so as to provide a definition for the derived shift/reduce sequence. Lines containing \texttt{e} represent empty
right-hand sides. The grammar exhibits not only the input structure of the text but also the state of the processing when conditional constructs are involved. This allows the easy separation of lines to be processed and lines to be ignored.

This grammar is ambiguous. One kind of ambiguity is isolated in the *if-success*, *if-failure*, *elif-success*, and *elif-failure* non-terminals. The resolution of the ambiguity depends on the evaluation of the constant expressions following #if and #elif. The other ambiguity is isolated in *body* when it begins with a left parenthesis. If the left parenthesis is preceded by white space, it starts *formals* instead. Once the ambiguities are resolved, the rest of the processing naturally follows the grammar.

The symbol *pp-token* represents input from the scanner. Some tokens are explicit (for example, *if* and *new-line*) because they are part of the structure of the preprocessor directives.

```
translation-unit:
  groups
groups:
  groups group
group:
  source-lines
  if-section
  control-line
source-lines:
  c
  source-lines expand new-line
if-section:
  if-success un-elif-groups un-else-group endif-line
  if-success un-elif-groups endif-line
  if-failure elif-succeeded un-else-group endif-line
  if-failure elif-succeeded endif-line
  if-failure elif-failures else-group endif-line
  if-failure elif-failures endif-line
if-success:
  if-test groups
if-failure:
  if-test un-groups
if-test:
  # if expand new-line
  # ifdef identifier new-line
  # ifndef identifier new-line
elif-succeeded:
  elif-failures elif-success un-elif-groups
elif-failures:
  c
  elif-failures elif-failure
```
elif-success:
    elif-test groups
elif-failure:
    elif-test un-groups
elif-test:
    # elif expand new-line
else-group:
    # else new-line groups
endif-line:
    # endif new-line
control-line:
    # include expand new-line
    # define identifier body new-line
    # define identifier formals body new-line
    # undef identifier new-line
    # line expand new-line
    # error inert-tokens new-line
    # pragma expand new-line
    # new-line
expand:
    ε
    not-leading-sharp pp-tokens
pp-tokens:
    ε
    pp-tokens pp-token
inert-tokens:
    ε
    inert-tokens pp-token
formals:
    ( identifier-list )
    ( )
identifier-list:
    identifier
    identifier-list identifier
body:
    ε
    body pp-token
un-groups:
    ε
    un-groups un-group
un-group:
    un-source-lines
    un-if-section
    un-control-line
un-if-section:
Preprocessor Input Parsing and Actions

Preprocessor input is implemented as a recursive descent parser with shift/reduce sequence output[8]. The parser is ordinary except for two kinds of ambiguities that it must resolve by extra-syntactic means. Enhancements to the parser for incremental preprocessing will be described in part II of this paper.

The interpretation of the shift/reduce sequence is implemented in a separate action module. During parsing the actions may cause new input to be inserted at the scan point (file include.) Thus the grammar and the parser are not actually parsing the text as presented. The actions are also responsible for redirecting tokens to the macro expander, expression evaluator, internal preprocessor tables, or preprocessor output.

The structure of our macro table is conventional. It is essentially a non-scoped stack of definitions. The attributes of a definition are type of macro (function versus object,) formal parameters (if a function macro,) and the body of the macro. The tokens in the macro table carry locators for possible diagnostic use. A #define causes the table to be searched for non-benign redefinition prior to entering the new macro. An #undef just makes an existing definition invisible to the lookup routine, but leaves it otherwise intact. This turns out to be important for incremental preprocessing. The C standard requires that if a function-like macro name is found but not followed by actual parameters, it
is treated as an ordinary identifier.

Because a macro lookup must be done for every identifier, the speed of lookup is an important design criterion. This is also true for incremental preprocessing, since all lookups are journalled and replayed for subsequent processing. Our choice is to use hashing. In fact the string abstraction\[5\] provides a perfect hash, provided that the hash root table can be grown as the number of unique strings increases.

An \#include directive is processed by one of the three filename rules\[1, section 3.8.2\] and then causes the behavior of the pp input scanner to be modified. Include directives hide a nasty complication: path names and search lists. There is no information in the standard on how to interpret the file name and each existing implementation seems to have some unique difference. For example, the character set in which file names are expressed can be different from the character set in which the program is written. For reusability we suggest isolating the interpretation of filenames in a single platform-dependent routine.

The granularity of the shift/reduce sequence is such that populating the switch in the preprocessor input action routine is straightforward.

Preprocessor Macros

Once the preprocessor input has passed a token sequence needing macro expansion to the macro expander, the macro expander must recognize and expand the macro invocations embedded in otherwise inert text. The process of removing a macro invocation and replacing it with its expansion is most convenient if the tokens are held in a linear list structure. As the tokens arrive, the macro expander builds the list. In fact, because of the recursive nature of macro expansion, the macro expander itself may be the source of its own input. All reexamination of expanded text is completed within the macro expander before any resulting tokens are reported out to the caller. Therefore the caller does not need to reexamine the output for more macro activity.

The main loop of the macro expander examines tokens, immediately passing them to its output list unless it recognizes a macro invocation. This leads to a grammar for macro expander input which describes invocations and a corresponding shift/reduce parser for the new grammar. The macro invocation parser is private to the pp-macro submodule.

Macro Invocation Parsing Grammar

\[
\begin{align*}
\text{control-expand} : & \\
& \text{expr-expand eol} \\
& \text{normal-expand eol} \\
\text{expr-expand} : & \\
& \epsilon \\
& \text{expr-expand expr-expand-action}
\end{align*}
\]
expr-expand-action:
  object-like-macro-name
  function-like-macro-name macro-actuals
  function-like-macro-name
defined ( identifier )
defined identifier
token

normal-expand:
  empty
  normal-expand normal-expand-action

normal-expand-action:
  object-like-macro-name
  function-like-macro-name macro-actuals
  function-like-macro-name
token
object-like-macro-name:
  identifier
function-like-macro-name:
  identifier
macro-actuals:
  ( macro-actual-list )
  ()
macro-actual-list:
  macro-actual
  macro-actual-list , macro-actual
macro-actual:
  no-free-parens
  not-paren-not-comma
  macro-actual no-free-parens
  macro-actual not-paren-not-comma
no-free-parens:
  empty
  no-free-parens ( no-free-parens )
  no-free-parens not-paren

The grammar sorts out the complexities of macro actual parameters where commas and parentheses can be part of the macro machinery or part of the input, depending on the exact context. The corresponding reduce actions provide sufficiently fine granularity to process expansions. The nonterminals not-paren-not-comma and not-paren stand for any token except the excluded ones.
Macro Expansion

By the time a macro is to be expanded, its effect is either predefined or must have been entered by a user directive interpreted in the preprocessor input phase (above.) Predefined macros have their own special interpretations. For user defined macros, copying the macro body, identifying the formals, stringizing, pasting, parameter expansion, and parameter substitution can be done in a single pass, provided that macro expansion can be called recursively. The existence of white space within actual macro parameters must be carried through into the output[1, Section 3.8.3.5].

Even though the defined operator is conceptually in the preprocessor expression module, it must be evaluated by the macro expander, suppressing macro expansion of its argument, and leaving a 0 or 1 behind. One implication is that the macro expander must react differently to input destined for expression evaluation and input destined for preprocessor interpretation or output. The macro expansion grammar is ambiguous in non-terminals expr-expand and normal-expand. The information on what the expanded output is going to be used for is available when the macro expander is called and is therefore passed in for disambiguation. It is also convenient to leave 0 behind for undefined identifiers in preprocessor expressions rather than pass them on to the expression evaluator.

To call a macro, its name and the actual arguments are stacked in a macro call frame. The macro invocation tokens are removed from the input list and the macro expander is called. The macro expander looks up the macro definition and copies the macro body into a local list. Then it proceeds through the list to find each formal parameter. Each parameter that is an argument to the # or ## operators is copied exactly into the body; otherwise the actual parameter is passed to the macro expander which recurs, fully expands the parameter, and copies the expanded result into the body. The # and ## operators are obeyed as soon as their operands are available. The substituted body is then inserted in place of the elided invocation in the input list.

Once the body has been properly substituted for its invocation, it must be rescanned for further macros, including ones constructed by ##, but excluding any with the name of the just-expanded macro. This recursion-avoiding requirement can be implemented by inserting begin-suppress and end-suppress markers for the name of the expanded macro just before the first token of its body and just after the last token of its body. When the first marker is encountered, the expander stacks the name of the macro on a suppress stack. When it encounters the second one, the macro expander checks that the name is still on top, and then removes it. If a macro name is found in the suppress stack, it is treated as an ordinary identifier.
typedef enum {
   PpLongSignedInt,
   PpUnsignedLongInt,
   PpDiagnostic
} PpType;

typedef struct {
   PpType t;
   union {
      signed long int sli;
      unsigned long int uli;
      char *err;
   } v;
} PpValue;

Figure 2: Discriminated Unions for Preprocessor Types and Values

Preprocessor Expression Evaluation

Preprocessor expressions are defined in terms of expressions in C. Some constructs are not meaningful (such as subscripts and pointers) and there is one new operator (defined.) Much of the difference can be captured as a subgrammar of the C expression grammar. As mentioned earlier, neither the defined operator nor identifier appears in this grammar since they were processed earlier during macro expansion. The resulting 0 or 1 is all that gets to the evaluator. Float and string values are rejected.

The preprocessor expression evaluator takes post-expansion tokens as input. The evaluation requires a running computation of both value and type. The rules for the type of constant expressions are different from those in the semantic processing in that all integral values are forced to one of the long types. There is no requirement that the limits of integral values or the values of character constants be the same in preprocessor evaluation and compiled code.

One solution for evaluation is to implement a recursive recognizer for preprocessor expressions, each function returning the type and value of the expression it has recognized. The value returned is of type long signed int or long unsigned int when no error occurred, or char * for a diagnostic message. This leads to a so-called discriminated union data structure where the first field of the struct is used to choose a field from the union, displayed in Figure 2.

Each recursive functions is responsible for recognizing the syntax of its non-terminal, checking the validity of the operands, and computing both type and value for the result. Diagnostics can be required for syntax errors or evaluation failures. Because the C standard requires that conditional evaluation of expressions (operators || && ? :) follow the usual rules, the recursion must be able to
static PpValue
ConditionalExpression(bool evaluating) { /* maybe skip all */
  PpValue tt, tf;
  bool et;  /* maybe skip tt */
  bool ef;  /* maybe skip tf */

  t = LogicalOrExpression(evaluating); /* eval 1st arg */
  if (TokenClass(NextToken) != questionTOKEN) return t;

  NextToken = Scan(); /* discard '?' */
  et = evaluating && PpValue2bool(t);
  tt = ConstantExpression(et); /* eval 2nd arg */
  if (TokenClass(NextToken) != colonTOKEN)
    return PpSyntaxError("expected ':'"); /* syntax error */

  NextToken = Scan(); /* discard ':' */
  ef = evaluating && !PpValue2bool(t);
  tf = ConditionalExpression(ef); /* eval 3rd arg */
  UsualConversions(&tt, et, &tf, ef); /* make same type */
  return et ? tt : tf; /* eval '?' */
}

Figure 3: Typical Preprocessor Evaluation Function

syntax and type check while either evaluating or ignoring evaluation.
The function in Figure 3 is typical[1, Section 3.3.15]. Parsing is intermixed
with interpretation. Once the expression parts are separately evaluated, the
C arithmetic rules combining types and values are applied. Error values act
like a type to which conversion is always possible and can be produced by any
of the routines called below. The final return value uses the same operator it
is processing (in this case ? :). Our decision to map preprocessor expression
semantics into the host machine architecture is convenient and allowed, but not
necessarily desirable, especially in cross-compiler situations.

It is possible to cause arithmetic faults in the preprocessor. For example,
one can cause an integer overflow.

    # if LONG_MAX + 1 == 0

or a spurious divide check

    # if 2 || 1/0

The result of an overflow is undefined[1, Sections 1.6, 3.3]. It is probably
best to issue a diagnostic. Division by zero is not an error. These kinds of
problems arise during recursive evaluation. It is possible to guard arithmetic
operators with fault-free fault-predicting code so as to avoid the faults occurring in the evaluator and also to provide an opportunity to issue a diagnostic. The details are beyond the scope of this paper.

In our implementation, about half of the code is in the recursive evaluator and half in the conversion of the various formats of constants.

Examples: Preprocessor Dark Corners

The examples of preprocessing in the C standard[1, Section 3.8.3.5] constitute a strenuous test of a preprocessor. The placement of white space is significant in the example for redefinition and reexamination. The quoting effect of the # and ## operators is shown in the example illustrating literals and token concatenation. Getting the white space right here, and distinguishing function-like macro definitions from object-type definitions are the reasons for the scanner keeping track of white space. A single bit, indicating the presence of preceding white space, is adequate for preprocessing. For other uses the actual white space is sometimes needed.

Although the standard does not provide pathological examples, the preprocessor must deal with them. For example, it is clear that

```c
#define __STDC__
```

is in violation of the rule forbidding redefinition of __STDC__[1, Section 3.8.8]. What is not so clear is the indirect, local redefinition below. We issue a diagnostic.

```c
#define P(__STDC__) !__STDC__
```

When the file name in an #include is not one of the two standard forms, the third option is macro expansion. But what should be done about the following dark corner? Our choice is to carry out the include.[1, section 3.8.2]

```c
#define HB h>
#include <cstdlib.HB
```

Can actual parameters to macros to disappear before they are substituted as shown below[1, section 3.8.3]?  

```c
#define N(a)  
#define T(a, b) a b  
T(Hello, N(Dolly))
```

Can a macro definition disappear before it is fully used[1, section 3.8.3]? In fact, # is just out of place in macro parameters.

```c
#define N(a) a  
N (  
#define M  
M (  
# undef M  
oops
```
Supposing one has a macro and a use as follows:

```c
#define M(a) a(a)

M(M)
```

In the expansion of the outer M, the substitution of M for a provides two opportunities to expand the actual parameter M, but in neither case is M expanded because M is a function-like macro and not followed by a left paren. After substitution and before rescanning the original macro has become M(M) again but this time M is on the suppress stack and no further macro evaluation is carried out, leaving the resulting text M(M)[1, Sections 3.8.3, 3.8.3.4].

One cannot successfully construct the ## or # operators. The reason is that these operators must be interpreted before the macro evaluation in the body is started. The created operators appear too late to work. Using these two operators anywhere except on formal parameters can be rejected, but supposing the preprocessor is more forgiving we might wonder what would happen below:

```c
#define M(a, b) a b
#define N(a) M(#, a)
#define P(a) a ## a
#define Q(a, b) a P(#) b
```

In the first case N(H1) → M(#, H1) → # H1, not "Hi".
In the second case Q(x, y) → x P(#) y → x ## y, not xy.

One cannot separate the beginning ( of a macro invocation from the closing ), or an #if from its matching #endif across the end of an include file[1, section 2.1.1.2].

The use of the preprocessor to define large tables takes it far beyond the minimal translation limit of 509 characters in a single source line[1, section 2.2.4.1]. A mature reusable preprocessor must be tested for all the minimal limits, and be implemented so that there are no hard limits short of exhausting memory.

**Summary**

The preprocessor described here consists of three main submodules implemented as relatively conventional parsers together with actions to carry out the preprocessor functions. The preprocessor can be tested apart from its input scanner and its consumer parser. Each of the three submodules can also be tested apart from the other two. The organization of this preprocessor is matched first to the requirements of the C standard, and second to the requirements of incremental processing, to be described in part II of this paper.
References


William McKeeman is a Senior Consulting Engineer for Digital. He has co-authored several books and has published papers in the areas of compilers, programming language design, and programming methodology. His current technical interests are studying and improving compile speed and responsiveness and the application of Software Engineering techniques to small programming projects. He can be reached at mckeeman@tle.dec.com.

Shot Aki is a Principal Software Engineer at Digital. He was a principal developer of the VAX APL interpreter. His current interests are in the areas of CASE, Analysis and Design Methods, Compilers, and Tools. He can be reached at aki@tle.dec.com.