CHAPTER 8: MACRO PROCESSING

Macros are used to extend some underlying language — to perform a translation from one language to another. For example, many of our programs contain lines like

```
while (getc(c) <> ENDFILE) do
```

where ENDFILE is some unspecified constant value that indicates end of file. "Symbolic constants" like ENDFILE tell you what a number signifies in a way that the number itself could never do: if we had written some magic value like -1 you would not know what it meant without understanding the surrounding context. Besides, the value of ENDFILE may well differ from machine to machine, and it is much easier and safer to redefine the value of a constant in a single place than it is to go through an entire program finding all the -1's that really mean end of file.

In Pascal, the const declaration lets us define the value of ENDFILE, so long as it is a number (or a boolean or a quoted string). For most of the programs that we have written here, that is perfectly adequate. But there are other situations where this notion of a constant is too limited.

What we want is a program that lets us define symbolic constants like ENDFILE so that subsequent occurrences of the name are replaced by the defining string of characters, regardless of the contents of the definition or its context. Such a definition is called a *macro*, the replacement process is called *macro expansion*, and the program for doing it is called a *macro processor*. A macro processor copies its input to its output with the macro definitions deleted and the macro references expanded. This lets us use parameters even in places where a compiler would insist on numbers, and it permits more ambitious language extensions such as our error handler.

Our first step in this chapter is a program define for replacement of one string of text by another — the most elementary form of macro processing. This lets us say, for instance,

```
define(ENDFILE, -1)
```

and thereafter have all occurrences of ENDFILE replaced by -1. Although this

is not much of a "language translation," it does make programs easier to read and change. This is about all the macro processing we would have needed for the programs in this book if `const` had not been available.

The second stage, a much bigger job, is to construct a processor that allows macros to have arguments, so we can say, for example,

```
define(putc(c), putcf(c, STDOUT))
```

to have all occurrences of `putc(x)` replaced by `putcf(x, STDOUT)` for whatever value the argument `x` might take.

The third stage is to add to the macro processor a handful of other built-in operations that materially assist in writing complicated macro operations. The most important of these are facilities for conditional testing and for evaluating arithmetic expressions. These give the macro processor the full capabilities of a programming language, at least in a formal sense.

The second and third stages are really luxuries: they are convenient to have, and it is instructive to see how to build them, but you can accomplish a great deal without them, as is evident from the utility of `const`.

We should also emphasize that this is not the only way to specify macros. Our notation is *functional*, i.e., it resembles the way function references are written in most programming languages, so macro calls mesh well with such languages. We could have borrowed syntax from Pascal:

```
define ENDFILE = -1
```

but that does not extend as well to multi-line definitions or to macros with arguments or to some of the other built-in operations we want to add. In Chapter 7 we suggested the form

```
define name
    body
endmarker
```

which is suitable for a language where input is handled a line at a time. Another possibility is a *template* macro processor, in which the macros correspond to operators (like the + and - in arithmetic expressions), and the arguments are the operands. Processors for template macros are sometimes easier to use, but are harder to write. The bibliographic notes at the end of the chapter suggest additional reading.

8.1 Simple Text Replacement

Let us begin with the easiest case. What we want is to copy input to output, except that when certain input strings appear they are to be replaced by previously defined replacement text. In a programming language like Pascal, the natural unit of replacement is the identifier, that is, a string of alphanumeric characters beginning with a letter and surrounded by non-alphanumerics. In the text

```
while (getc(c) <> ENDFILE) do
```

ENDFILE is surrounded by non-alphanumerics and is thus a candidate for replacement. Of course so are while, getc, c, and do, but since they are presumably not defined to the macro processor, they should be copied unaltered.

The unit of replacement is called a *token*. In other situations, a token might be anything between "white space" (blanks, tabs, newlines) as it was in format, or anything between a pair of specified left and right markers. In any case, one part of the processor is a routine that reads input and divides it up into tokens according to some rule.

How are definitions provided? The syntax suggested above is convenient:

```
define(name, replacement text)
```

defines *name* to be whatever text follows, up to a balancing right parenthesis; this allows the replacement text to be longer than one line. We will need modules to collect the name and replacement text, and to record new names and definitions as they are encountered.

Some of the implementation details are critical, because the order in which operations are done can make a big difference in the power and convenience of a macro processor. One significant decision is what should happen when one name is defined in terms of another one. For example, after the definitions

```
define(x, 1)
define(y, x)
```

does the input y produce x or 1? If the definitions are in reverse order,

```
define(y, x)
define(x, 1)
```

then what is y?

We don't want users of define to have to worry too much about the order in which their definitions appear. Accordingly, define is built so examples like this one will work in the more useful way — after a macro has been evaluated, its replacement text is *rescanned*. If it contains any further macros they in turn go through the same expansion process. (This introduces the chance of an infinite loop, of course, so we must be prepared for that eventuality.)

There are also several possibilities for *when* macro calls are evaluated. If we have already defined x with

```
define(x, 1)
```

then when

```
define(y, x)
```

is encountered, we can either replace x by 1 immediately, or we can ignore the

fact that x is a macro and replace it later when y is invoked. In the example above these two methods produce the same result, but if x should subsequently be redefined, there would be a difference. Different choices here lead to somewhat different but equally useful processors. In our `define` processor, definitions are *not* scanned for macro calls while they are being copied into the table of definitions; the interpretation of macros is done as late as possible.

But first here is the outline of the no-argument macro processor.

```
while (gettok(token) <> ENDFILE)
    look up token
    if (token = 'define')
        install new token and value
    else if (token was found in table)
        switch input to definition of token
    else
        copy token to output
```

Since there are nested sources of input, in principle this is a recursive process. In `define` we will deal with recursion in a different way from the explicit recursion that we have used in other programs. We will get back to this shortly.

`gettok` is analogous to the `getword` routine we wrote in Chapter 3, but it must be made somewhat more complicated to handle non-alphabetic characters properly. For example, blanks are now significant and can't be ignored. The call

```
c := gettok(token, maxtok)
```

copies the next token from the standard input into `token`. A token is either a string of letters and digits, or a single non-alphanumeric character. The function value returned by `gettok` is the first character of the token; this determines whether or not the token is alphabetic.

```
{ gettok -- get token for define }
function gettok (var token : string; toksize : integer)
        : character;
var
    i : integer;
    done : boolean;
begin
    i := 1;
    done := false;
    while (not done) and (i < toksize) do
        if (isalphanum(getpbc(token[i]))) then
            i := i + 1
        else
            done := true;
    if (i >= toksize) then
        error('define: token too long');
    if (i > 1) then begin   { some alpha was seen }
        putback(token[i]);
        i := i - 1
    end;
    { else single non-alphanumeric }
    token[i+1] := ENDSTR;
    gettok := token[1]
end;
```

Looking for tokens one character at a time, we don't know that we have seen the end of the token until we have gone one character too far. This is a classic example of an undesirable side effect, one that can tremendously complicate a program if we let it. Each time we need another character, we must check whether to read a new character or use the one we already have. Tangling this up with the logic of what to *do* with each character would make an unreadable mess.

Instead we hide the complication by introducing a pair of cooperating routines. `getpbc` delivers the next input character to be considered, both in its argument and as its function value. `putback` puts a character back on the input, so that the next call to `getpbc` will return it again. Now, every time `gettok` reads one character too many, it promptly pushes it back, so the rest of the code does not have to know about the problem.

One possibility is to make `putback` a primitive operation, so `getpbc` can be simply `getc`. We have separated them here to illustrate how the pushback can be done, since in general you will have to provide your own. `putback` puts the pushed-back characters into a buffer. `getpbc` reads from the buffer if there is anything there; it calls `getc` if the buffer is empty.

```
{ putback -- push character back onto input }
procedure putback (c : character);
begin
    if (bp >= BUFSIZE) then
        error('too many characters pushed back');
    bp := bp + 1;
    buf[bp] := c
end;

{ getpbc -- get a (possibly pushed back) character }
function getpbc (var c : character) : character;
begin
    if (bp > 0) then
        c := buf[bp]
    else begin
        bp := 1;
        buf[bp] := getc(c)
    end;
    if (c <> ENDFILE) then
        bp := bp - 1;
    getpbc := c
end;
```

bp is the index of the next character to be returned from buf; if bp is zero a fresh character is fetched by a call to getc. (bp must be initialized to zero.) The buffer and pointer used by getpbc and putback are global variables in define, and initialized by initdef.

```
buf : array [1..BUFSIZE] of character;  { for pushback }
bp : 0..BUFSIZE;    { next available character; init=0 }
```

As written, gettok never pushes back more than one character between calls to getpbc, so buf could have been an ordinary scalar variable instead of an array. But pushback is a useful mechanism, well worth generalizing. We can even write pbstr, which pushes back an entire string by repeated calls to putback.

```
{ pbstr -- push string back onto input }
procedure pbstr (var s : string);
var
    i : integer;
begin
    for i := length(s) downto 1 do
        putback(s[i])
end;
```

It is of course necessary to push a string back in reverse order.

Only getpbc and putback know about the data structure of buf and bp. pbstr could be faster if it also knew about them and could avoid the overhead of calling putback for each character, but as much as possible we try to

minimize data connections between routines. This is one of the most effective ways we know of to write code that can be easily changed. Certainly if it later proves true that the overhead in `pbstr` is a bottleneck, then we can improve it. The important thing is to start with a good design. It is much easier to relax the standards for something written well than it is to tighten them for something done badly.

Since we can push back something different from what was read, it has probably occurred to you that `putback` provides an elegant way to implement the rescanning of macro replacement text. Suppose that after a defined name is found, we push its *replacement text* back onto the input. When that is read, if it in turn contains a defined name, the name will be looked up and translated just as if it had been in the input originally. This pushback is how we handle the recursion implicit in nested sources of input.

Now we can write the main program, `define`:

```
{ define -- simple string replacement macro processor }
procedure define;
#include "defcons.p"
#include "deftype.p"
#include "defvar.p"
    defn : string;
    token : string;
    toktype : sttype;    { type returned by lookup }
    defname : string;    { value is 'define' }
    null : string;       { value is '' }
#include "defproc.p"
begin
    null[1] := ENDSTR;
    initdef;
    install(defname, null, DEFTYPE);
    while (gettok(token, MAXTOK) <> ENDFILE) do
        if (not isletter(token[1])) then
            putstr(token, STDOUT)
        else if (not lookup(token, defn, toktype)) then
            putstr(token, STDOUT)    { undefined }
        else if (toktype = DEFTYPE) then begin  { defn }
            getdef(token, MAXTOK, defn, MAXDEF);
            install(token, defn, MACTYPE)
        end
        else
            pbstr(defn) { push replacement onto input }
end;
```

If the token returned by `gettok` is not a letter (as determined by `isletter`) it cannot be a defined symbol. We test for that right away, to avoid looking up every non-alphanumeric character. Here is one possible implementation of `isletter` for ASCII machines (but recall the discussion of character sets and `isupper` in Chapter 2):

```
{ isletter -- true if c is a letter of either case }
function isletter (c : character) : boolean;
begin
    isletter :=
        c in [ord('a')..ord('z')] + [ord('A')..ord('Z')]
end;
```

The token is looked up with `lookup`, which also returns the defining text and type if the token was found. If the token wasn't found by `lookup`, it has no special significance, and can be output immediately. If it was *define*, the name and replacement text are isolated with `getdef` and installed in the table by `install`. If the token was found and was not a *define*, the replacement text is pushed back onto the input. The type `sttype` is an enumeration of the possible symbol table entry types, which for `define` are just `DEFTYPE` for the built-in "`define`" and `MACTYPE` for a macro name:

```
type
    sttype = (DEFTYPE, MACTYPE);     { symbol table types }
```

`install` is used to place the keyword *define* in the table in the first place, along with a null string as its replacement and `DEFTYPE` as its type. This is better than entering it with a set of assignment statements, because the program doesn't need to know anything about the format of table entries. `lookup` returns the type `DEFTYPE` when it finds *define*, so we can quickly check whether a token is a *define*. Besides the elements of `sttype`, `lookup` and `install` are the only visible parts of the table-handling mechanism; they are the subject of the next section.

Here is `getdef`:

```
{ getdef -- get name and definition }
procedure getdef (var token : string; toksize : integer;
        var defn : string; defsize : integer);
var
    i, nlpar : integer;
    c : character;
begin
    token[1] := ENDSTR; { in case of bad input }
    defn[1] := ENDSTR;
    if (getpbc(c) <> LPAREN) then
        message('define: missing left paren')
    else if (not isletter(gettok(token, toksize))) then
        message('define: non-alphanumeric name')
    else if (getpbc(c) <> COMMA) then
        message('define: missing comma in define')
    else begin  { got '(name,' so far }
        while (getpbc(c) = BLANK) do
            ;   { skip leading blanks }
        putback(c); { went one too far }
        nlpar := 0;
        i := 1;
        while (nlpar >= 0) do begin
            if (i >= defsize) then
                error('define: definition too long')
            else if (getpbc(defn[i]) = ENDFILE) then
                error('define: missing right paren')
            else if (defn[i] = LPAREN) then
                nlpar := nlpar + 1
            else if (defn[i] = RPAREN) then
                nlpar := nlpar - 1;
            { else normal character in defn[i] }
            i := i + 1
        end;
        defn[i-1] := ENDSTR
    end
end;
```

Most of the task here is coping with balanced parentheses and invalid input. Where possible, `getdef` simply notes the error and continues, in an attempt to give the user as much information as possible per run.

For completeness, here is `initdef`; we will show `inithash` after we discuss table lookup in the next section.

```
{ initdef -- initialize variables for define }
procedure initdef;
begin
    { setstring(defname, 'define'); }
        defname[1] := ord('d');
        defname[2] := ord('e');
        defname[3] := ord('f');
        defname[4] := ord('i');
        defname[5] := ord('n');
        defname[6] := ord('e');
        defname[7] := ENDSTR;
    bp := 0;     { pushback buffer pointer }
    inithash
end;
```

Exercise 8-1. What happens if you say

```
define(d, define)
d(a, b)
a
```

What happens with

```
define(define, x)
define(a, b)
```

□

Exercise 8-2. What happens if you say

```
define(x, x)
```

or

```
define(x, y)
define(y, x)
```

and then ask for `x`? What would you like to have happen? □

Exercise 8-3. If you write

```
define(x, x x)
```

sooner or later the pushback buffer will overflow and `define` will exit. This is generally regarded as better behavior than causing an infinite loop. Can you devise a scheme for turning all potential infinite loops into stack overflows? □

Exercise 8-4. `getdef` deletes any blanks that might appear at the front of the replacement text. Why is this desirable? Would any harm result if they were retained? What happens to blanks around the macro name? □

Exercise 8-5. If a line contains nothing but a definition, any trailing blanks and the newline are copied to the output, even though it might seem more natural to eliminate them completely. Modify `getdef` or some other part of the program so no output is produced from a line containing only definitions. Is this an appropriate action if the output is fed to a compiler that uses line numbers for diagnostics? □

Exercise 8-6. As an alternate and more general solution to the previous problem, implement a built-in operation `dnl` (for "delete newline") which deletes all characters from its occurrence up to and including the next newline. Thus in the input

```
define(x, 1)dnl
```

the `dnl` deletes all text after the definition, and the line produces no output. □

8.2 Table Lookup

Let us now design `lookup` and `install`, the routines for handling tables of names and definitions. We have already made one important design decision — all information about table format, search strategy, and the like is private, known only by `lookup` and `install`. All other routines must access the table through them. Information hiding is critical to proper program design: routines that don't need to know about the internal representation of a data structure should not know about it. Not only does this ensure that data is not inadvertently changed, but more important, it breaks the program into independent pieces, where each can be changed without affecting the others. Each piece is a black box, presenting only a well-defined interface to the world. In our case, if we change some aspect of the table — to sort the names for binary search, for instance — we can do so with impunity, because no other routine knows what the tables look like. Of course the "need to know" has to be genuine. It's all too easy to design routines whose users "need" to know about the data, when with more care the structure could be concealed.

Inside the lookup code, the lookup strategy determines the table structures needed. The simplest table management is a linear table: add new entries to the end of a table as they arrive, and search the table from one end to the other each time a token must be looked up.

In the early stages of a program, fancy search techniques are not worth the extra complexity. Linear search is not always the best thing, but it is an excellent first choice. It is easy to implement and likely to work right the first time. If it later proves to be a bottleneck, it can be replaced with a faster algorithm without affecting the rest of the program. The expected time to find a token in the table (or determine that it is not present) is proportional to the length of the table, however, so the bottleneck can appear early.

The next step up in complexity is probably to sort the entries and use binary search to locate tokens. But if definitions arrive at unpredictable times rather than all at once, it is necessary to sort each time a definition arrives. The search time for such a table is proportional to the logarithm of the table size if the sorting time can be ignored, which is true so long as there are many more calls to `lookup` than to `install`. It runs faster, but it's more complicated.

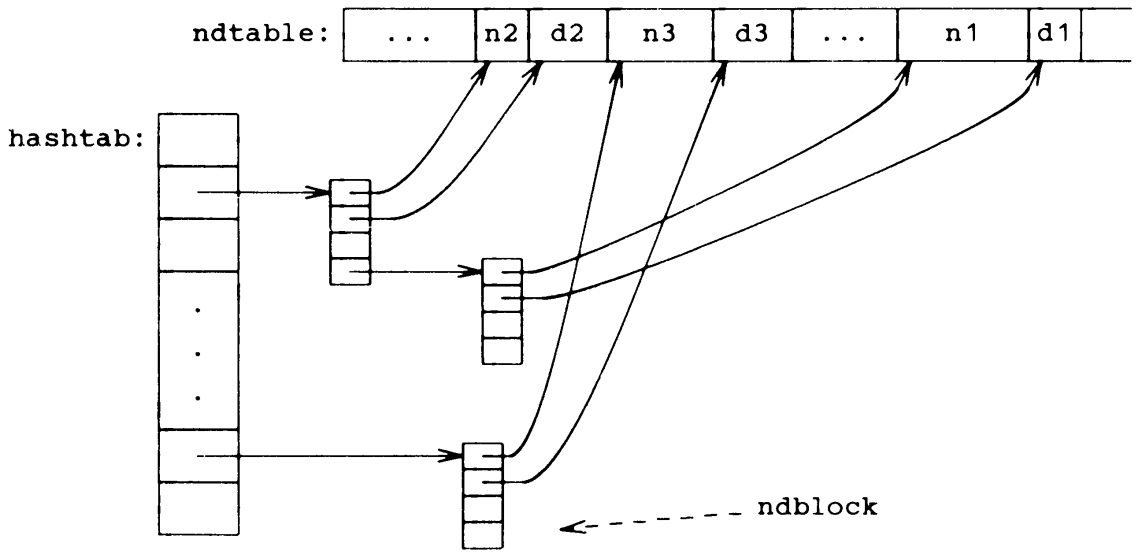
A third solution, and in real life, usually the best, is to use a hash search. A token is "hashed" to create an index into a table of lists. Scanning the list at the hash index determines whether the token is present or not. If the hashing function does a good job, tokens are spread uniformly through the table and the

search time is constant, that is, independent of the number of entries present, until the table becomes nearly full. Even then, the search time increases only linearly as more elements are added.

Clearly efficiency and simplicity are at conflict in the design of table searching strategies, but as it turns out, an efficient solution is not greatly more complicated than a simple one. The hash search is organized like this: One large table called `ndtable` contains the names and replacement texts, stored one after another as

name ENDSTR *definition* ENDSTR *name* ENDSTR *defn* ENDSTR ...

This is the same idea that we used in `sort` in Chapter 4, except that since there is a name and a definition for each entry, we use a record to hold the `ndtable` indices. A second array, called `hashtab`, contains pointers to name-definition records. An element of `hashtab` points to the beginning of a linked list of records describing names that have that hash value; it is set to `nil` if no names have hashed to that value. (`nil` is a Pascal keyword. A pointer which does not point to an object should have the value `nil`.) Each record in the chain contains the `ndtable` indices of the name and definition, the type of the object (`DEFTYPE` or `MACTYPE`), and a pointer to the next name-definition record. This too is `nil` for the last record in the chain. A picture is worth about a thousand words:



The declarations are as follows:

```
{ deftype -- type definitions for define }
type
    charpos = 1..MAXCHARS;
    charbuf = array [1..MAXCHARS] of character;
    sttype = (DEFTYPE, MACTYPE);    { symbol table types }
    ndptr = ^ndblock;    { pointer to a name-defn block }
    ndblock =
        record      { name-defn block }
            name : charpos;
            defn : charpos;
            kind : sttype;
            nextptr : ndptr
        end;
```

The notation "`ndptr = ^ndblock`" is read as "`ndptr` is a pointer to an `ndblock`." Some character sets use an up-arrow '↑' instead of the circumflex '^'.

The corresponding variables are

```
{ defvar -- var declarations for define }
var
    hashtab : array [1..HASHSIZE] of ndptr;
    ndtable : charbuf;
    nexttab : charpos;  { first free position in ndtable }
    buf : array [1..BUFSIZE] of character;  { for pushback }
    bp : 0..BUFSIZE;    { next available character; init=0 }
```

We have to put all these variables into global variables because `lookup`, `install` and any subordinates all have to know about them and there is no other way to pass the information.

`nexttab` is the next free position in `ndtable`; it must be initialized to 1 and all entries in `hashtab` made `nil`. This is done in `inithash`:

```
{ inithash -- initialize hash table to nil }
procedure inithash;
var
    i : 1..HASHSIZE;
begin
    nexttab := 1;   { first free slot in table }
    for i := 1 to HASHSIZE do
        hashtab[i] := nil
end;
```

`lookup` returns `true` and extracts the definition and type if the token was found; otherwise it returns `false`.

```
{ lookup -- locate name, get defn and type from table }
function lookup (var name, defn : string; var t : sttype)
        : boolean;
var
    p : ndptr;
begin
    p := hashfind(name);
    if (p = nil) then
        lookup := false
    else begin
        lookup := true;
        cscopy(ndtable, p^.defn, defn);
        t := p^.kind
    end
end;
```

A construct like `p^.defn` accesses the `defn` component of the record that `p` points to. `cscopy` is the string-copying procedure we wrote in Chapter 4 to copy a string from a big array of type `charbuf` to a `string`.

The real work of finding an occurrence of `name` is done by `hashfind`:

```
{ hashfind -- find name in hash table }
function hashfind (var name : string) : ndptr;
var
    p : ndptr;
    tempname : string;
    found : boolean;
begin
    found := false;
    p := hashtab[hash(name)];
    while (not found) and (p <> nil) do begin
        cscopy(ndtable, p^.name, tempname);
        if (equal(name, tempname)) then
            found := true
        else
            p := p^.nextptr
    end;
    hashfind := p
end;
```

The actual hashing done by `hash`. Ours is dead simple, though not especially good: it adds up the characters of the string, each time multiplying the previous sum by three, then takes the remainder modulo `HASHSIZE`. `HASHSIZE` ought to be prime to get a reasonably uniform distribution of hash values.

```
{ hash -- compute hash function of a name }
function hash (var name : string) : integer;
var
    i, h : integer;
begin
    h := 0;
    for i := 1 to length(name) do
        h := (3 * h + name[i]) mod HASHSIZE;
    hash := h + 1
end;
```

`install` adds a new name, definition and type to the head of the chain of records which have that hash value; it is called when a *define* is encountered. `install` does not check whether the name is already in the table, so names may be redefined just by giving a new definition: since `lookup` scans the chain from the front, the new definition supersedes the old.

Space for the pointer blocks is obtained by calling the Pascal storage management function `new`. `new(p)` creates an object of the type that `p` points to, then sets `p` to point to it. The contents of a dynamic object are accessed by `p^`; if the object is actually a record, as it is here, then individual components are selected with `p^.`*name*.

```
{ install -- add name, definition and type to table }
procedure install (var name, defn : string; t : sttype);
var
    h, dlen, nlen : integer;
    p : ndptr;
begin
    nlen := length(name) + 1;    { 1 for ENDSTR }
    dlen := length(defn) + 1;
    if (nexttab + nlen + dlen > MAXCHARS) then begin
        putstr(name, STDERR);
        error(': too many definitions')
    end
    else begin  { put it at front of chain }
        h := hash(name);
        new(p);
        p^.nextptr := hashtab[h];
        hashtab[h] := p;
        p^.name := nexttab;
        sccopy(name, ndtable, nexttab);
        nexttab := nexttab + nlen;
        p^.defn := nexttab;
        sccopy(defn, ndtable, nexttab);
        nexttab := nexttab + dlen;
        p^.kind := t
    end
end;
```

The constants and procedures needed by `define` are as follows:

```
{ defcons -- const declarations for define }
const
    BUFSIZE = 500;        { size of pushback buffer }
    MAXCHARS = 5000;      { size of name-defn table }
    MAXDEF = MAXSTR;      { max chars in a defn }
    MAXTOK = MAXSTR;      { max chars in a token }
    HASHSIZE = 53;        { size of hash table }

{ defproc -- procedures needed by define }
#include "cscopy.p"
#include "sccopy.p"
#include "putback.p"
#include "getpbc.p"
#include "pbstr.p"
#include "gettok.p"
#include "getdef.p"
#include "inithash.p"
#include "hash.p"
#include "hashfind.p"
#include "install.p"
#include "lookup.p"
#include "initdef.p"
```

PROGRAM

`define` expand string definitions

USAGE

`define`

FUNCTION

`define` reads its input, looking for macro definitions of the form

```
define(ident, string)
```

and writes its output with each subsequent instance of the identifier `ident` replaced by the sequence of characters `string`. `string` must be balanced in parentheses. The text of each definition proper results in no output text. Each replacement string is rescanned for further possible replacements, permitting multi-level definitions.

EXAMPLE

```
define
define(ENDFILE, (-1))
define(DONE, ENDFILE)
    if (getit(line) = DONE) then
        putit(sumline);
<ENDFILE>


    if (getit(line) = (-1)) then
        putit(sumline);
```

BUGS

A recursive definition such as `define(x, x)` will cause an infinite loop when `x` is invoked.

Exercise 8-7. Verify that `getdef` and `install` work correctly if the definition is empty, so that

```
define(nothing,)
```

defines a string with no replacement text. Why would you want to define such a thing? What is the effect of the macro call

```
nothing(this is a line of text)
```

□

Exercise 8-8. Redefining names without salvaging the old space is obviously profligate if done often. Modify `install` to make better use of the space in `ndtable`. □

Exercise 8-9. Add an `undefine` command

```
undefine(name)
```

that removes the most recent definition of *name*. What should happen if you `undefine` a name that wasn't defined? □

Exercise 8-10. Experiment with different hashing algorithms, to see how randomly they distribute the hash value between 1 and `HASHSIZE`, and what effect they have on `define`'s performance as a function of number of entries in the table. □

Exercise 8-11. Implement a version of `define` that does not use pushback in the sense that we have, but instead maintains a stack of current input sources, and switches those appropriately. Try another version that uses explicit recursion to select different inputs. Which version is easiest? Which version is fastest? □

Exercise 8-12. How does `define` deal with the comment conventions of common programming languages? Should `define` know about quoted strings? That is, should defined names appearing within quotes be replaced? □

Exercise 8-13. It is often useful to have at least a rudimentary conditional test. Suppose we say that a line like

```
ifdef(name, text)
```

means "if *name* is defined, put *text* in the input, otherwise skip over it." You could parameterize a program for different machines by writing definitions like

```
ifdef(pdp11, define(wordsize, 16) define(charsize, 8))
ifdef(pdp10, define(wordsize, 36) define(charsize, 7))
ifdef(ibm370, define(wordsize, 32) define(charsize, 8))
```

Then defining `pdp11` with the (empty) definition

```
define(pdp11,)
```

sets parameters like `wordsize` correctly for the PDP-11 when the `ifdef` lines are encountered. Changing this single definition and reprocessing resets the program for some other machine. Implement this conditional facility. □

8.3 Some Measurements

We timed `define` by replacing all `const` symbols from the code for `edit` and its supporting routines with `define`'s and running it through `define`. Here are some data from one timing study, with `HASHSIZE` set to 53. Total

time was 38 seconds on a DEC VAX 11/780.

	#calls	CPU time(%)
`isalphanum`	46997	21.2
`putcf`	36884	17.4
`isletter`	21830	14.8
`getc`	39917	10.2
`gettok`	21831	7.7
`getpbc`	47763	6.6
`putstr`	21182	4.3
`cscopy`	9915	3.1
`hash`	5890	2.5
`hashfind`	5781	2.4
`define`	1	2.1
`equal`	9375	1.6
`length`	6540	1.5
`putback`	7847	0.9
`lookup`	5781	0.7
`getdef`	108	0.1
`install`	109	0.1
`pbstr`	432	0.0

Our `putstr` calls `putcf`.

Before we consider the run time percentages, it's worth remarking that there is a lot of information in a measurement as simple as the number of times each procedure is called, especially if the program is carefully modularized so each routine does only one thing. For example, this data tells us that the input contains 39916 characters (one call of `getc` finds an `ENDFILE`), 36884 output characters, 108 defined tokens, 432 occurrences of defined tokens out of 5781 that were looked up (which means that most of the time `lookup` reports failure), and so on. Some of this data provides consistency checking on the operation of the program. For example, we expect one more `install` than `getdef` (installing the keyword `define` in the first place); if that is not so, something is badly amiss.

The CPU time data tells us checking the types of characters with `isalphanum` and `isletter` is surprisingly expensive, 36 percent, which suggests that a different implementation is called for. I/O time is next, somewhat over 27 percent. The table lookup mechanism is cheap; all of its components add up to only about 7 percent. The pushback mechanism is a modest cost on each character, well worth it for the clarity it brings to the program. Our decision to write `pbstr` in terms of `putback` is also vindicated.

We substituted range-test versions of `isalphanum` and `isletter` and repeated the test. Total time went down to 29 seconds; run-time percentages for `isalphanum` and `isletter` went to 8.6 and 4.3 respectively.

It is important to justify decisions made in the name of efficiency for one very good reason. Most of the time programmers have no real idea where time is being consumed by a program. Consequently nearly all the effort expended

(and the clarity sacrificed) for "efficiency" is wasted. We have found that the best way to avoid too-early optimization is to make a regular practice of instrumenting code. Only from such first-hand experience can one learn a proper sense of priorities.

8.4 Macros with Arguments

Macros with arguments add to the power of the macro processor. For example, we said that `getc` and `putc` are equivalent to `getcf(c, STDIN)` and `putcf(c, STDOUT)` respectively. By defining `getc` and `putc` as macros that expand into references to `getcf` and `putcf`, we guarantee equivalence, and we eliminate a level of procedure call, which may improve efficiency. The replacement is not possible without an argument capability.

As another example with immediate relevance, we could replace `isalphanum` and `isletter` by macros. The entire body of `isletter` is just

```
isletter :=
    c in [ord('a')..ord('z')] + [ord('A')..ord('Z')]
```

and `isalphanum` is not much bigger. These are such short routines that they could readily be macros instead of functions, expanding into in-line code rather than calling another routine. We could define a macro

```
isletter(c)
```

that would expand into the lines above (or into appropriate range tests), with occurrences of the formal parameter `c` replaced by the actual argument used when the macro is invoked. For the user, there would be no difference.

The syntax for specifying macros with arguments is an extension of what we used before:

`define(`*name*`,` *replacement text*`)`

defines *name*. This time, however, any occurrence in the replacement text of `$`*n*, where *n* is between 1 and 9, will be replaced by the *n*th argument when the macro is actually called. Thus

```
define(isletter,
    ($1 in [ord('a')..ord('z')] + [ord('A')..ord('Z')]))
```

defines the `isletter` macro.

Specifying arguments with `$`*n* is not as pleasant as being able to use dummy names for the parameters, as in

```
define(isletter(c),
    (c in [ord('a')..ord('z')] + [ord('A')..ord('Z')]))
```

but it is easier to build. Write something clean and acceptable that works, then polish it later if necessary.

The restriction to nine arguments is another example of the same

philosophy. It is silly to get sidetracked worrying about macros with lots of arguments until the rest of the processor is working. You will find that in practice there is rarely any call for more anyway. Hard cases can wait until the easy ones are well under control.

It's harder to write a macro processor that allows arguments than one that doesn't. Furthermore, we intend to add a small set of "built-in" operations in addition to *define*: a conditional statement, an arithmetic capability, and a couple of string functions, and we want these to go in without much effort. The main thing is to ensure that any operation — macro call, definition, other built-in — can occur in the middle of any other one. If this is possible, then in principle the macro processor is capable of doing any computation, although it may well be hard to express.

As long as no macro calls are encountered (or built-ins, since they are treated identically), the input is copied directly to the output. When a macro is called, however, its name, its definition, and its arguments (if any) are all collected. Once the argument collection is finished, the macro is evaluated as follows. If it is a built-in like *define*, an appropriate routine is called that does whatever it has to with the arguments. If the macro is not a built-in, the definition text is pushed back onto the input. As it is being pushed back, any $n's in it are replaced by the corresponding argument that was just collected.

The fun starts when one of the arguments includes a call of another macro or built-in. Although there are various ways to deal with this situation, one of the easiest is to interpret the arguments as they are being collected, then push them back onto the input.

When a macro invocation is seen, the name and definition are placed in an evaluation area organized as a stack. Any arguments that follow are copied into this area as well, except that when an argument contains another macro invocation (a nested one) a new stack frame is created, and that inner macro is evaluated *completely* and its translation pushed back onto the input before the stack is popped and we resume working on the outer macro. The outer macro never sees the inner one, just its translation. (Of course the inner macro may in turn call upon other macros; the process is recursive.)

The thing to keep firmly in mind at all times is that arguments are evaluated completely as they are being collected. This is different behavior from the string replacement process we showed earlier in the chapter, but for common uses like replacing symbolic parameters in programs, the two methods produce the same result. (We will also provide for deferred evaluation so we can have the benefits of the earlier method when we need them.) Here are some examples, before we start on the actual code.

Suppose we have

```
define(ENDSTR, 0)
```

When the `define` is seen, `define` and its replacement text (which is null) are

put in the evaluation stack. Now we collect the arguments. `ENDSTR` at this point is nothing special, nor is `0`, so they are put on the stack at positions 3 and 4 respectively. At the end of the `define` (when the definition is finished) we can evaluate, which in this case involves calling a routine to install the name and definition, which are the arguments at positions 3 and 4. Then the top four items on the stack are popped.

If we subsequently see `ENDSTR` in the input, it will be put on the stack with its definition, and no arguments. The definition `0` is pushed back onto the input and the stack popped.

More complicated, here is an example with arguments.

```
define(incr, $1:=$1+1)
```

defines `incr` to be a macro that generates code to increment its argument by 1. The input

```
incr(x)
```

causes `incr` and `$1:=$1+1` to be copied onto the evaluation stack. `x` is collected as the argument; if it is not a defined name, we reach the end of the invocation of `incr` without incident. The definition is pushed back onto the input, with each occurrence of `$1` replaced by `x`, to yield `x:=x+1`.

But imagine for a moment that `x` had earlier been defined to be something else, say `a[i]`. Then `x` is a macro call, so when it is encountered, a new stack frame is formed, and `x` and its definition are copied into that frame. Then the definition `a[i]` is pushed back onto the input, and the frame popped. When argument collection resumes for the previous level (`incr`), the input that used to be `x` has become `a[i]`, and this becomes the actual argument to `incr`. As far as `incr` is concerned, it *was* called with `a[i]` as its argument, and the result is `a[i]:=a[i]+1`.

Exercise 8-14. Assuming that `getc` and `putc` have been defined in terms of `getcf` and `putcf`, go through the expansion process by hand for the input

```
putc(getc(c))
```

including the processing of values for `STDIN` and `STDOUT`. □

8.5 Implementation

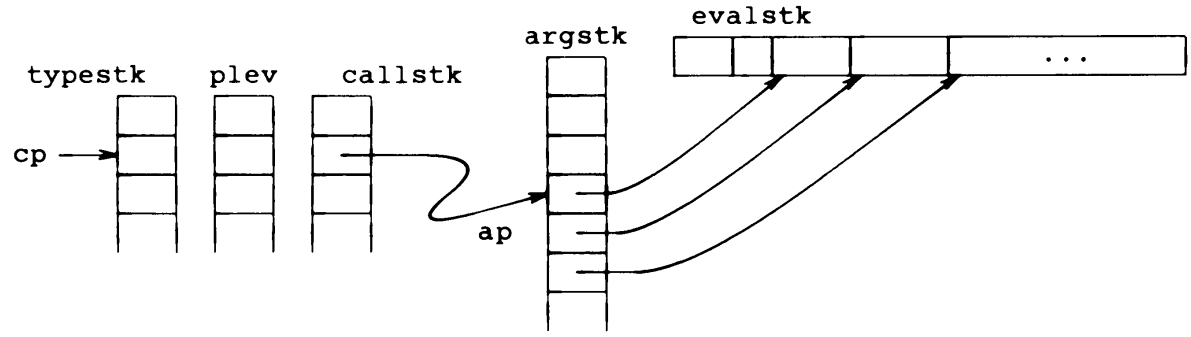
The processing can now be spelled out in more detail.

```
while (gettok(token, maxtok) <> ENDFILE)
    if (type is letter)
        if (not lookup(token))
            copy token to current evaluation stack frame
               or directly to output
        else
            make new stack frame
            copy name and definition to current stack frame
    else if (stack empty)   { not saving arguments }
        copy directly to output
    else if (at end of an argument list)
        if (built-in)
            do appropriate function
        else
            push definition back onto input,
               replacing $n's by arguments
        pop stack frame
    else                    { saving arguments }
        copy token to current stack frame
```

Of course this skips over a few details like precisely how we know when we're at the end of an argument list, and what the stack looks like. We will get to them in due course.

First the evaluation stack. This is just a long array `evalstk`. The first free position in `evalstk` is kept in `ep`, which is initially 1. Whenever we are processing a macro or built-in, `evalstk` contains the strings for the name, definition and arguments. The array `argstk` contains the locations in `evalstk` where these strings begin: `argstk[i]` is the beginning of the `i`th string in `evalstk`. `ap` is the first free position in `argstk`; it is also initially 1.

Since macros and built-ins may be nested, the strings in `evalstk` that `argstk` points to are in general associated with different levels. The array `callstk` keeps track of each stack frame: `callstk[i]` points to the position in `argstk` that in turn points to the defining text of the `i`th macro. A parallel array `typestk` records the type of the corresponding macro or built-in. `cp` is the *current* call stack position. If `cp` is zero, we are not in any macro. Inside a single level of invocation, `cp` is one, and so on. Thus to find the first argument of the third level of invocation, we first set `i:=callstk[3]`. Then `argstk[i]` is the defining text of this macro, `argstk[i+1]` is the name, and `argstk[i+2]` is the first argument. Thus, `argstk` effectively becomes an array of variable length records, implemented of necessity as a less structured array of indices.



Argument collection requires keeping track of balanced parentheses independently for each level of macro, so we add another array `plev`, also parallel to `callstk`, to count parentheses for the corresponding stack frame.

Several routines need to know about `cp` and the output buffer; `callstk`, `typestk`, `argstk`, `ap`, and `plev` are used only in the main routine.

```
callstk : posbuf;    { call stack }
cp : pos;            { current call stack position }
typestk : array[1..CALLSIZE] of sttype; { type }
plev : array [1..CALLSIZE] of integer;  { paren level }
argstk : posbuf;     { argument stack for this call }
ap : pos;            { current argument position }
evalstk : charbuf;   { evaluation stack }
ep : charpos;        { first character unused in evalstk }
```

There is one last complication. Any macros encountered during argument collection are expanded immediately. But there are times when we must defer the evaluation until later. For example, consider this attempt to make a new macro `d` synonymous with `define`:

```
define(d, define($1,$2))
```

On cursory inspection it should work, because the replacement text of `d` appears to be `define($1,$2)`. But macros and built-ins are evaluated *as soon as they are encountered.* The inner `define` is evaluated before the outer one. Because a `define` has no replacement text, the net effect is to define `d` to be empty, which is hardly what was wanted. To get around the problem of premature evaluation, there must be a quoting convention, so input can be treated as literal text when necessary. In our convention, any input surrounded by ` and ' is left absolutely alone, except that one level of ` and ' is stripped off. With this facility we can write the macro `d` as

```
define(d, `define($1,$2)')
```

The replacement text for `d`, protected by the quotes, is literally `define($1,$2)`. Now when we say

```
d(a, bc)
```

everything works and `a` is defined to be `bc`.

Quotes must also be used when it is desired to redefine an identifier:

```
define(x,y)
define(x,z)
```

would define `y` in the second line, instead of redefining `x`. (The first definition is still active, however, so `x` ultimately becomes `z`.) If you do not want to redefine `y`, the operation must be expressed as

```
define(x,y)
define('x',z)
```

which will have the desired effect.

Because the right quote character ' means something in Pascal, sometimes it is a nuisance to have it pre-empted, so we will also add a built-in to permit the quote characters to be changed at will.

Putting all of these considerations together creates a long main program, but it is not really complicated. It follows the outline we gave earlier, except for the addition of quotes. It is a seven-way branch, with the code for each case in-line instead of in a separate routine. We have called it `macro` rather than `define`, because that better reflects what it does.

```
{ macro -- expand macros with arguments }
procedure macro;
#include "maccons.p"
#include "mactype.p"
#include "macvar.p"
    defn : string;
    token : string;
    toktype : sttype;
    t : character;
    nlpar : integer;
#include "macproc.p"
begin
    initmacro;
    install(defname, null, DEFTYPE);
    install(exprname, null, EXPRTYPE);
    install(subname, null, SUBTYPE);
    install(ifname, null, IFTYPE);
    install(lenname, null, LENTYPE);
    install(chqname, null, CHQTYPE);

    cp := 0;
    ap := 1;
    ep := 1;
    while (gettok(token, MAXTOK) <> ENDFILE) do
        if (isletter(token[1])) then begin
            if (not lookup(token, defn, toktype)) then
                puttok(token)
            else begin  { defined; put it in eval stack }
                cp := cp + 1;
                if (cp > CALLSIZE) then
                    error('macro: call stack overflow');
                callstk[cp] := ap;
                typestk[cp] := toktype;
                ap := push(ep, argstk, ap);
                puttok(defn);   { push definition }
                putchr(ENDSTR);
                ap := push(ep, argstk, ap);
                puttok(token);  { stack name }
                putchr(ENDSTR);
                ap := push(ep, argstk, ap);
                t := gettok(token, MAXTOK); { peek at next }
                pbstr(token);
                if (t <> LPAREN) then begin { add () }
                    putback(RPAREN);
                    putback(LPAREN)
                end;
                plev[cp] := 0
            end
        end
```

```
        else if (token[1] = lquote) then begin  { strip quotes }
            nlpar := 1;
            repeat
                t := gettok(token, MAXTOK);
                if (t = rquote) then
                    nlpar := nlpar - 1
                else if (t = lquote) then
                    nlpar := nlpar + 1
                else if (t = ENDFILE) then
                    error('macro: missing right quote');
                if (nlpar > 0) then
                    puttok(token)
            until (nlpar = 0)
        end
        else if (cp = 0) then   { not in a macro at all }
            puttok(token)
        else if (token[1] = LPAREN) then begin
            if (plev[cp] > 0) then
                puttok(token);
            plev[cp] := plev[cp] + 1
        end
        else if (token[1] = RPAREN) then begin
            plev[cp] := plev[cp] - 1;
            if (plev[cp] > 0) then
                puttok(token)
            else begin  { end of argument list }
                putchr(ENDSTR);
                eval(argstk, typestk[cp], callstk[cp], ap-1);
                ap := callstk[cp];  { pop eval stack }
                ep := argstk[ap];
                cp := cp - 1
            end
        end
        else if (token[1]=COMMA) and (plev[cp]=1) then begin
            putchr(ENDSTR); { new argument }
            ap := push(ep, argstk, ap)
        end
        else
            puttok(token);  { just stack it }
    if (cp <> 0) then
        error('macro: unexpected end of input')
end;
```

We want to retain the property of `define` that a macro call without arguments (like `ENDFILE` or `ENDSTR`) does not require parentheses. Thus if a token is a defined name, and it is not followed immediately by a left parenthesis, we push back an empty set of balanced parentheses, so that macro calls without arguments are not a special case for the rest of the program. This is another example of altering some data representation in a minor way to avoid much greater

complexity in the code.

You may have noticed that quotes are removed even outside macro definitions. Although this may look like unnecessary meddling on the part of the macro processor, there are good reasons for doing it that way. As the simplest example, if you really want a literal occurrence of the word `define` in your text, you have to protect it with a layer of quotes or it will be interpreted as a call to the built-in `define`. We will see some more substantial instances of this shortly.

`puttok` and `putchr` put strings and characters respectively either into `evalstk` (if we are in the middle of a macro), or directly onto the output with `putc` (if we are not). The test for what destination to use occurs in a single place in `putchr`, not scattered throughout the code.

```
{ puttok -- put token on output or evaluation stack }
procedure puttok (var s : string);
var
    i : integer;
begin
    i := 1;
    while (s[i] <> ENDSTR) do begin
        putchr(s[i]);
        i := i + 1
    end
end;

{ putchr -- put single char on output or evaluation stack }
procedure putchr (c : character);
begin
    if (cp <= 0) then
        putc(c)
    else begin
        if (ep > EVALSIZE) then
            error('macro: evaluation stack overflow');
        evalstk[ep] := c;
        ep := ep + 1
    end
end;
```

When a new argument is to be put into `evalstk` we have to record the current value of the position `ep` and increment `ap`; this is done by `push`:

```
{ push -- push ep onto argstk, return new position ap }
function push (ep : integer; var argstk : posbuf;
        ap : integer) : integer;
begin
    if (ap > ARGSIZE) then
        error('macro: argument stack overflow');
    argstk[ap] := ep;
    push := ap + 1
end;
```

Once a macro has been identified and all its arguments collected in `evalstk` (signaled by the parenthesis level becoming zero), `eval` is called to process a built-in or to push back a definition with the appropriate arguments.

`macro` pushes the definition onto `evalstk` before the name, so when `eval` is called, `args[i]` points to the defining text for the macro and `args[i+1]` points to the name. `args[i+2]` through `args[j]` are the arguments, of which there are `j-i-1`. This organization means that `$0` is the name of the macro itself. Although this will probably be little used, the regularity is nice to have.

```
{ eval -- expand args i..j: do built-in or push back defn }
procedure eval (var argstk : posbuf; td : sttype;
        i, j : integer);
var
    argno, k, t : integer;
    temp : string;
begin
    t := argstk[i];
    if (td = DEFTYPE) then
        dodef(argstk, i, j)
    else begin
        k := t;
        while (evalstk[k] <> ENDSTR) do
            k := k + 1;
        k := k - 1; { last character of defn }
        while (k > t) do begin
            if (evalstk[k-1] <> ARGFLAG) then
                putback(evalstk[k])
            else begin
                argno := ord(evalstk[k]) - ord('0');
                if (argno >= 0) and (argno < j-i) then begin
                    cscopy(evalstk, argstk[i+argno+1], temp);
                    pbstr(temp)
                end;
                k := k - 1  { skip over $ }
            end;
            k := k - 1
        end;
        if (k = t) then     { do last character }
            putback(evalstk[k])
    end
end;
```

If the type is `define`, `dodef` is called; otherwise the definition is pushed back onto the input, with each `$`*n* replaced by the corresponding argument. The symbolic constant `ARGFLAG` is a `$`.

We haven't said what the macro processor should do when a macro definition asks for an argument that wasn't supplied. The most harmless thing is to ignore it — in effect to replace the `$`*n* by an empty string — and this is what `eval` does. This is true even if no arguments are present; that way if `x` is defined by

```
define(x, a$1b)
```

the inputs

```
x(+)
x(-,+)
x()
x
```

all produce something sensible: `a+b`, `a-b`, `ab` and `ab` respectively.

`dodef` is easy: most of the work has already been done for it.

```
{ dodef -- install definition in table }
procedure dodef (var argstk : posbuf; i, j : integer);
var
    temp1, temp2 : string;
begin
    if (j - i > 2) then begin
        cscopy(evalstk, argstk[i+2], temp1);
        cscopy(evalstk, argstk[i+3], temp2);
        install(temp1, temp2, MACTYPE)
    end
end;
```

One of the first things to try with the macro processor is extending the syntax of our programming language. Suppose we try to use macros to create the `error` primitive that we have been using throughout the book. If we say

```
define(error, `begin writeln($1); halt end')
```

then a "statement" like

```
error('argument stack overflow');
```

will be converted into

```
begin writeln('argument stack overflow'); halt end;
```

thus avoiding the problem of fixed-length strings.

The `halt` statement is not standard Pascal either, by the way, though it is widely available. If your Pascal has no such mechanism, you can go on to say

```
define(halt, goto 9999)
```

where `9999` is a label in the outermost block of your program, just after the call of your main procedure.

Exercise 8-15. Modify the definition of `error` given above so that the error message includes the name of the program being run. You might also arrange that output appears on a different file than normal, so it does not disappear down a pipeline. Write a macro for `message` as well. □

Exercise 8-16. The definition

```
define(sqr, $1 * $1)
```

defines a macro to square an expression. Or does it? What is `sqr(x+1)`? What can

you do about it? How much should a macro processor know about the language(s) it is used with? □

Exercise 8-17. Invent a syntax that allows macros to have more than nine arguments. Make it compatible with the `$n` syntax if $n<10$. How difficult is it to implement? □

Exercise 8-18. Improve `define` to allow the parameters in a macro definition to be specified by dummy names instead of by `$n`. That is, if `m` is defined by

`define(m(x,y),` *replacement text containing tokens* `x` *and* `y)`

then the invocation `m(a,b)` should replace all occurrences of the tokens `x` and `y` in the replacement text by `a` and `b` respectively. How much existing machinery can you use? □

8.6 Conditionals, Arithmetic, and Other Built-ins

`macro` has been designed to make it easy to add new built-in functions as the need arises. The next step in the evolution is the addition of a conditional test, with a built-in function `ifelse`. The input

`ifelse(`*a*`,`*b*`,`*c*`,`*d*`)`

compares *a* and *b* as character strings. If they are the same, *c* is pushed back onto the input; if they differ, *d* is pushed back. As a rudimentary example,

```
define(compare, `ifelse($1,$2,yes,no)')
```

defines `compare` as a two-argument macro returning `yes` if its arguments are the same, and `no` if they're not. As usual, the quotes prevent the `ifelse` from being evaluated too soon.

While we are adding built-in functions, we will do four more.

`expr(`*expression*`)`

evaluates the string *expression* as an arithmetic expression and returns that as its replacement text (as a string of characters). *expression* had better be a valid expression, or the results may be undesirable. "Arithmetic expression" includes `+`, `-`, `*`, `/` (integer division only, like `div`), `%` for remainder (Pascal's `mod`), and parentheses.

Suppose that you need parameters such as

```
const
    SCREENWID = 80;
    MAXLINE = SCREENWID + 1;
```

But the second definition is illegal: a constant cannot be an expression, however appealing it might seem. The problem with writing two definitions, as in

```
SCREENWID = 80;
MAXLINE = 81;
```

is that you must remember to update both if one changes (and recognize that `81` is actually `80+1` in disguise). Clearly it is better to define one in terms of the other. With `expr`, the job is easy:

```
define(SCREENWID, 80)
define(MAXLINE, `expr(SCREENWID+1)')
```

`expr` has much more capability than this example would indicate; we'll show some more soon.

The next built-in is a function to take substrings of strings.

```
substr(s,m,n)
```

produces the substring of *s* which starts at position *m* (with origin one), of length *n*. If *n* is omitted or too big, the rest of the string is used, while if *m* is out of range the result is a null string.

```
substr(abc, 2, 1)
```

is `b`,

```
substr(abc, 2)
```

is `bc`, and

```
substr(abc, 4)
```

is empty.

The built-in `len` returns the length of its argument. Interestingly, `len` can be defined in terms of the other built-ins. What is the length of a string *s*? If *s* is empty, its length is zero. Otherwise it is one more than the length of the substring of *s* obtained by chopping off one character. This is a recursive definition, which is a natural form of expression if you happen to have a recursive language at hand — and we do. Let's say it with macros:

```
define(len,`ifelse($1,,0,`expr(1+len(substr($1,2)))')')
```

This is certainly a mouthful, but not hard to understand in the light of the recursive definition above. It is permissible, and often necessary, to define macros in terms of themselves. It works because conditional testing can be used to prevent an infinite loop. In this case the test is whether all the characters of the string have been chopped away.

The outer layer of quotes prevents all evaluation as the definition is being copied into the table. The inner layer prevents the `expr` construction from being done as the arguments of the `ifelse` are collected.

As you can imagine, computing a length this way is expensive, which is our main reason for including `len` as a built-in.

The final built-in is `changeq`, which permits the default quote characters ` and ' to be set to something else. `changeq(xy)` sets the quotes to `x` and `y`; `changeq` without an argument resets them to the default.

The changes needed to add `ifelse`, `expr`, `substr`, `len` and `changeq` are minor. We modify `macro` to install the new keywords and their types (`IFTYPE`, `EXPRTYPE`, `SUBTYPE`, `LENTYPE`, `CHQTYPE`), and change `eval` to

look for them as well as for DEFTYPE. In eval we only have to add the extra tests and procedure calls.

```
...
else if (td = EXPRTYPE) then
    doexpr(argstk, i, j)
else if (td = SUBTYPE) then
    dosub(argstk, i, j)
else if (td = IFTYPE) then
    doif(argstk, i, j)
else if (td = LENTYPE) then
    dolen(argstk, i, j)
else if (td = CHQTYPE) then
    dochq(argstk, i, j)
else begin
    { process normal macro as before }
    ...
```

doif compares the first two arguments, and pushes back the appropriate one onto the input. If there is no else argument, a null string is returned.

```
{ doif -- select one of two arguments }
procedure doif (var argstk : posbuf; i, j : integer);
var
    temp1, temp2, temp3 : string;
begin
    if (j - i >= 4) then begin
        cscopy(evalstk, argstk[i+2], temp1);
        cscopy(evalstk, argstk[i+3], temp2);
        if (equal(temp1, temp2)) then
            cscopy(evalstk, argstk[i+4], temp3)
        else if (j - i >= 5) then
            cscopy(evalstk, argstk[i+5], temp3)
        else
            temp3[1] := ENDSTR;
        pbstr(temp3)
    end
end;
```

doexpr does the arithmetic, converts the number, and pushes the result back as a character string with pbnum.

```
{ doexpr -- evaluate arithmetic expressions }
procedure doexpr (var argstk : posbuf; i, j : integer);
var
    temp : string;
    junk : integer;
begin
    cscopy(evalstk, argstk[i+2], temp);
    junk := 1;
    pbnum(expr(temp, junk))
end;

{ pbnum -- convert number to string, push back on input }
procedure pbnum (n : integer);
var
    temp : string;
    junk : integer;
begin
    junk := itoc(n, temp, 1);
    pbstr(temp)
end;
```

Most of the work in evaluating expressions is pushed off into `expr`, which uses recursive calls on several sub-functions to evaluate the components of the expression. The method is quite conventional: the grammar for an arithmetic expression can be written as

```
expr   : term + term
       ¦ term - term
term   : factor * factor
       ¦ factor / factor
       ¦ factor % factor
factor : number
       ¦ ( expr )
```

The recursion is in the process of finding the pieces. Each syntactic type (*expr*, *term*, *factor*) has a corresponding function. `expr` searches for a pair of *term*'s separated by `+` or `-`, `term` searches for *factor*'s, and `factor` looks for either a plain number or a parenthesized *expr*.

```
{ expr -- recursive expression evaluation }
function expr (var s : string; var i : integer) : integer;
var
    v : integer;
    t : character;
#include "gnbchar.p"
#include "term.p"
begin
    v := term(s, i);
    t := gnbchar(s, i);
    while (t in [PLUS, MINUS]) do begin
        i := i + 1;
        if (t = PLUS) then
            v := v + term(s, i)
        else
            v := v - term(s, i);
        t := gnbchar(s, i)
    end;
    expr := v
end;

{ term -- evaluate term of arithmetic expression }
function term (var s : string; var i : integer) : integer;
var
    v : integer;
    t : character;
#include "factor.p"
begin
    v := factor(s, i);
    t := gnbchar(s, i);
    while (t in [STAR, SLASH, PERCENT]) do begin
        i := i + 1;
        case t of
        STAR:
            v := v * factor(s, i);
        SLASH:
            v := v div factor(s, i);
        PERCENT:
            v := v mod factor(s, i)
        end;
        t := gnbchar(s, i)
    end;
    term := v
end;
```

```
{ factor -- evaluate factor of arithmetic expression }
function factor (var s : string; var i : integer)
         : integer;
begin
    if (gnbchar(s, i) = LPAREN) then begin
        i := i + 1;
        factor := expr(s, i);
        if (gnbchar(s, i) = RPAREN) then
            i := i + 1
        else
            message('macro: missing paren in expr')
    end
    else
        factor := ctoi(s, i)
end;
```

The function gnbchar finds the beginning of the next token; it differs from skipbl only in that it also skips newlines, and returns a character for testing.

```
{ gnbchar -- get next non-blank character }
function gnbchar (var s : string; var i : integer)
         : character;
begin
    while (s[i] in [BLANK, TAB, NEWLINE]) do
        i := i + 1;
    gnbchar := s[i]
end;
```

dolen is straightforward:

```
{ dolen -- return length of argument }
procedure dolen(var argstk : posbuf; i, j : integer);
var
    temp : string;
begin
    if (j - i > 1) then begin
        cscopy(evalstk, argstk[i+2], temp);
        pbnum(length(temp))
    end
    else
        pbnum(0)
end;
```

dosub does the substr function; it is entirely concerned with getting indices right, particularly in boundary cases where the substring requested is in some way outside the string. By calling expr instead of ctoi, dosub causes the second and third arguments to be evaluated as expressions without a surrounding call to expr.

```
{ dosub -- select substring }
procedure dosub (var argstk : posbuf; i, j : integer);
var
    ap, fc, k, nc : integer;
    temp1, temp2 : string;
begin
    if (j - i >= 3) then begin
        if (j - i < 4) then
            nc := MAXTOK
        else begin
            cscopy(evalstk, argstk[i+4], temp1);
            k := 1;
            nc := expr(temp1, k)
        end;
        cscopy(evalstk, argstk[i+3], temp1);    { origin }
        ap := argstk[i+2];  { target string }
        k := 1;
        fc := ap + expr(temp1, k) - 1;  { first char }
        cscopy(evalstk, ap, temp2);
        if (fc >= ap) and (fc < ap+length(temp2)) then begin
            cscopy(evalstk, fc, temp1);
            for k := fc+min(nc,length(temp1))-1 downto fc do
                putback(evalstk[k])
        end
    end
end;
```

dochq is easy:

```
{ dochq -- change quote characters }
procedure dochq (var argstk : posbuf; i, j : integer);
var
    temp : string;
    n : integer;
begin
    cscopy(evalstk, argstk[i+2], temp);
    n := length(temp);
    if (n <= 0) then begin
        lquote := ord(GRAVE);
        rquote := ord(ACUTE)
    end
    else if (n = 1) then begin
        lquote := temp[1];
        rquote := lquote
    end
    else begin
        lquote := temp[1];
        rquote := temp[2]
    end
end;
```

Finally, here are the other components that make up the program:

```
{ mactype -- type declarations for macro }
type
    charpos = 1..MAXCHARS;
    charbuf = array [1..MAXCHARS] of character;
    posbuf = array [1..MAXPOS] of charpos;
    pos = 0..MAXPOS;
    sttype = (DEFTYPE, MACTYPE, IFTYPE, SUBTYPE,
        EXPRTYPE, LENTYPE, CHQTYPE); { symbol table types }
    ndptr = ^ndblock;
    ndblock =
        record
            name : charpos;
            defn : charpos;
            kind : sttype;
            nextptr : ndptr
        end;
```

```
{ maccons -- const declarations for macro }
const
    BUFSIZE = 1000;      { size of pushback buffer }
    MAXCHARS = 5000;     { size of name-defn table }
    MAXPOS = 500;        { size of position arrays }
    CALLSIZE = MAXPOS;
    ARGSIZE = MAXPOS;
    EVALSIZE = MAXCHARS;
    MAXDEF = MAXSTR;     { max chars in a defn }
    MAXTOK = MAXSTR;     { max chars in a token }
    HASHSIZE = 53;       { size of hash table }
    ARGFLAG = DOLLAR;    { macro invocation character }

{ macvar -- var declarations for macro }
var
    buf : array [1..BUFSIZE] of character;  { for pushback }
    bp : 0..BUFSIZE;     { next available character; init=0 }

    hashtab : array [1..HASHSIZE] of ndptr;
    ndtable : charbuf;
    nexttab : charpos;   { first free position in ndtable }

    callstk : posbuf;    { call stack }
    cp : pos;            { current call stack position }
    typestk : array[1..CALLSIZE] of sttype; { type }
    plev : array [1..CALLSIZE] of integer;  { paren level }
    argstk : posbuf;     { argument stack for this call }
    ap : pos;            { current argument position }
    evalstk : charbuf;   { evaluation stack }
    ep : charpos;        { first character unused in evalstk }

    { built-ins: }
    defname : string;    { value is 'define' }
    exprname : string;   { value is 'expr' }
    subname : string;    { value is 'substr' }
    ifname : string;     { value is 'ifelse' }
    lenname : string;    { value is 'len' }
    chqname : string;    { value is 'changeq' }

    null : string;       { value is '' }
    lquote : character; { left quote character }
    rquote : character; { right quote character }
```

```
{ initmacro -- initialize variables for macro }
procedure initmacro;
begin
    null[1] := ENDSTR;
    { setstring(defname, 'define'); }
        defname[1] := ord('d');
        defname[2] := ord('e');
        defname[3] := ord('f');
        defname[4] := ord('i');
        defname[5] := ord('n');
        defname[6] := ord('e');
        defname[7] := ENDSTR;
    { setstring(subname, 'substr'); }
        subname[1] := ord('s');
        subname[2] := ord('u');
        subname[3] := ord('b');
        subname[4] := ord('s');
        subname[5] := ord('t');
        subname[6] := ord('r');
        subname[7] := ENDSTR;
    { setstring(exprname, 'expr'); }
        exprname[1] := ord('e');
        exprname[2] := ord('x');
        exprname[3] := ord('p');
        exprname[4] := ord('r');
        exprname[5] := ENDSTR;
    { setstring(ifname, 'ifelse'); }
        ifname[1] := ord('i');
        ifname[2] := ord('f');
        ifname[3] := ord('e');
        ifname[4] := ord('l');
        ifname[5] := ord('s');
        ifname[6] := ord('e');
        ifname[7] := ENDSTR;
    { setstring(lenname, 'len'); }
        lenname[1] := ord('l');
        lenname[2] := ord('e');
        lenname[3] := ord('n');
        lenname[4] := ENDSTR;
    { setstring(chqname, 'changeq'); }
        chqname[1] := ord('c');
        chqname[2] := ord('h');
        chqname[3] := ord('a');
        chqname[4] := ord('n');
        chqname[5] := ord('g');
        chqname[6] := ord('e');
        chqname[7] := ord('q');
        chqname[8] := ENDSTR;
    bp := 0; { pushback buffer pointer }
    inithash;
    lquote := ord(GRAVE);
    rquote := ord(ACUTE)
end;
```

PROGRAM

`macro` expand string definitions, with arguments

USAGE

`macro`

FUNCTION

`macro` reads its input, looking for macro definitions of the form

```
define(ident,string)
```

and writes its output with each subsequent instance of the identifier `ident` replaced by the arbitrary sequence of characters `string`.

Within a replacement string, any dollar sign `$` followed by a digit is replaced by an argument corresponding to that digit. Arguments are written as a parenthesized list of strings following an instance of the identifier, e.g.,

```
ident(arg1,arg2,...)
```

So `$1` is replaced in the replacement string by `arg1`, `$2` by `arg2`, and so on; `$0` is replaced by `ident`. Missing arguments are taken as null strings; extra arguments are ignored.

The replacement string in a definition is expanded before the definition occurs, except that any sequence of characters between a grave ` and a balancing apostrophe ' is taken literally, with the grave and apostrophe removed. Thus, it is possible to make an alias for define by writing

```
define(def,`define($1,$2)')
```

Additional predefined built-ins are:

`ifelse(a,b,c,d)` is replaced by the string `c` if the string `a` exactly matches the string `b`; otherwise it is replaced by the string `d`.

`expr(expression)` is replaced by the decimal string representation of the numeric value of `expression`. For correct operation, the expression must consist of parentheses, integer operands written as decimal digit strings, and the operators `+`, `-`, `*`, `/` (integer division), and `%` (remainder). Multiplication and division bind tighter than addition and subtraction, but parentheses may be used to alter this order.

`substr(s,m,n)` is replaced by the substring of `s` starting at location `m` (counting from one) and continuing at most `n` characters. If `n` is omitted, it is taken as a very large number; if `m` is outside the string, the replacement string is null. `m` and `n` may be expressions suitable for `expr`.

`len(s)` is replaced by the string representing the length of its argument in characters.

`changeq(xy)` changes the quote characters to `x` and `y`. `changeq()` changes them back to ` and '.

Each replacement string is rescanned for further possible replacements, permitting multi-level definitions to be expanded to final form.

EXAMPLE

The macro `len` could be written in terms of the other built-ins as:

```
define(`len',`ifelse($1,,0,`expr(1+len(substr($1,2)))')')
```

BUGS

A recursive definition of the form `define(x,x)` will cause an infinite loop.

Expression evaluation is fragile. There is no unary minus.

It is unwise to use parentheses as quote characters.

Exercise 8-19. (Frivolous) Define a macro `reverse` that will reverse a string. □

Exercise 8-20. The function `expr` is particularly vulnerable to syntactically invalid expressions. How would you make it more robust? Is it worth it? □

Exercise 8-21. Define an `assert` macro that will cause conditional compilation of assertions in a program: if assertions are turned on,

```
assert(i < j)
```

should expand into something like

```
if (not (i < j)) then
    error('false assertion: i < j')
```

Can you invent a way to include either the name of the procedure or the assertion number as part of the message? You will probably also want to define macros that turn assertion checking off and on at desired places. □

Exercise 8-22. Modify `doexpr` to do arbitrary precision arithmetic. Can you handle floating point operations? Add an exponentiation operator and a base 2 logarithm operation. Add relational operators. □

Exercise 8-23. Modify the storage management facilities in `macro` so that arrays are managed by calls to Pascal's `new` function. Add command-line arguments to permit the sizes of arrays to be specified when the program is run, with sensible defaults. □

Exercise 8-24. What changes would you make to `macro` to adapt it to providing a macro capability for the `format` program of Chapter 7? □

8.7 Applications

Let us write macros to handle the string declaration that we have been using in our programs. Suppose that

```
setstring(name,'text')
```

is a shorthand for

```
{ setstring(name,'text'); }
    name[1] := ord('t');
    name[2] := ord('e');
    name[3] := ord('x');
    name[4] := ord('t');
    name[5] := ENDSTR;
```

The task is to convert the `setstring` declaration into this expanded form.

The solution comes in three parts. First we replicate the call with braces:

```
{ setstring(name,'text'); }
```

Then we loop over the characters between quotes, producing lines of the form

```
name[i] := ord('c');
```

where *c* is the *i*th character of `text`. Finally we end with

```
name[n+1] := ENDSTR;
```

where *n* is the length of `text`.

`setstring` itself is

```
changeq(<>)
define(setstring,<{ <setstring>($1,$2); }
str($1,substr($2,2),0)  $1[len(substr($2,2))] := ENDSTR;
>)
```

where we have changed the quote characters to < and > to avoid difficulties with the ' used in Pascal. The call `len(substr($2,2))` computes the effective string length (excluding the quotes but including the `ENDSTR`). `str` creates the intervening `ord` statements:

```
define(str,<ifelse($2,',,  $1[expr($3+1)] := ord('substr($2,1,1)');
<str($1,substr($2,2),expr($3+1))>)>)
```

It isolates one character, increments the index, generates the line, and calls itself recursively until it sees the terminating quote.

This is obviously not the most transparent programming language in the world. It takes some getting used to before you can think of looping in terms of recursion, although with practice you get the hang of it. But beware of becoming too clever with macros. In principle, `macro` is capable of performing any computing task, but it is all too easy to write incomprehensible macros.

It is also the case that complicated recursive macro operations like `setstring` can be painfully slow. For example, here are some statistics for processing two short strings, of three and nine characters in length:

	#calls	CPU time(%)
putchr	4114	18.8
putback	3010	12.9
gettok	1694	11.9
isalphanum	3278	9.9
getpbc	3278	6.9
macro	1	5.9
puttok	1357	5.9
cscopy	534	5.9
length	520	4.0
putc	366	4.0
isletter	1201	2.0
getc	268	1.0
eval	91	1.0
equal	168	1.0
pbstr	243	1.0
push	391	1.0

Total time was 1.6 seconds.

This is a lot of procedure calls for such a small input; if you did nothing but process `setstring` macros, it would be intolerable. Fortunately the use of `macro` as a front end for a language processor tends to involve primarily substituting one string for another, as in `define`. This is much less demanding, so processing an occasional `setstring` is quite practical. The added complexity

of `macro` costs very little extra for this kind of application; `macro` takes about ten percent longer than `define` on the same input.

The measurements above do indicate where attention can be most profitably directed if it is necessary to speed `macro` up. One possibility is to observe that some of the calls to `gettok` could be replaced by calls to `getpbc`, since only a single character is involved (for example, while processing bracketed text). More generally, there are a number of rather small routines which we wrote to modularize the program properly. Part of the cost of `macro` is the overhead of the procedure calling mechanism, which can be very inefficient on some machines. We can avoid much of this by replacing procedure calls by in-line code in these places (although we would do it by defining macros to replace the procedure bodies, not by writing out the code!). Specifically, since virtually all of the calls to `putchr` originate in `puttok`, `putchr` can be moved into `puttok` with only minor rearrangements. If characters are small positive integers, `isalphanum` and `isletter` can be replaced by in-line references to an array which contains the type of the corresponding character; this will essentially eliminate the cost of finding character types. And if the variables for the pushback buffer are used more widely, `getpbc` and `putback` can also be made in-line operations.

Although care is necessary to keep the program relatively clean, the payoff can be substantial. The original version of `macro`, written in the language C, was speeded up by a factor of about four by such transformations. Similar results could be expected in Pascal on many machines. The process should be as we have described several times here, however: write a clean program that implements an appropriate algorithm; measure it to identify the hot spots; refine those as cleanly as possible. Starting from the other end is a sure way to an unworkable mess.

One thing that can be done to make macros faster and more comprehensible is to increase the set of built-ins, so computations don't have to be spelled out in excruciating detail. Here are some suggestions.

Exercise 8-25. Add a tracing facility to dump the name and definition of each macro as it is encountered. □

Exercise 8-26. Add a built-in analogous to the `index` function defined in Chapter 2: `index(s1,s2)` should return the position of the string `s2` in the string `s1`, or zero if `s2` does not occur in `s1`. Can you do `index` with the existing facilities? Should you? □

Exercise 8-27. Add a `translit` built-in that performs transliterations similar to the `translit` program of Chapter 2. Add a built-in `match(s,r)` which returns the position where the regular expression `r` begins in the string `s`, or zero if it doesn't. □

Exercise 8-28. Write a macro `rpt(s,n)` that evaluates `s(1)`, `s(2)`, ..., `s(n)`. Add a built-in that does the same job. □

Exercise 8-29. Add a built-in that will cause the contents of a file to be copied in at the point where it is encountered, as with the `include` processor of Chapter 3. Make sure that the included text is also scanned for macros! □

Exercise 8-30. Add a built-in `divert(f)` to cause the output of `macro` to be appended to file `f` instead of the standard output. Add another built-in `incl(f)` that will copy file `f` to the standard output without macro scanning. These two built-ins make it possible to do significant rearrangements of the input. For example in a Pascal program you can collect all `const`'s, all `type`'s, etc., in an arbitrary order and output them in the proper order. □

Bibliographic Notes

There is a lot more to macro processing than we have room for here. *An Introduction to Macros* by M. Campbell-Kelly (American-Elsevier, 1973) provides a brief discussion of several different forms of macro processors. *Macro Processors and Techniques for Portable Software* by P. J. Brown (Wiley, 1974) goes into more detail on the subtle aspects of macro processing.

The PL/I macro preprocessor is an attempt to make a macro language that is essentially the same as a compiler language. This is discussed in various PL/I texts and in reference manuals for particular implementations. For example, see *IBM System/360 PL/I Language Specification,* IBM Form Y33-6003, or *Student Text: An Introduction to the Compile-Time Facilities of PL/I,* IBM Form C20-1689.

Macros have been valuable in making "portable" software — programs that move from one machine to another with much less effort than complete rewriting. The program is written in terms of a modest number of macros; nothing but the macros must be written for a particular environment. Snobol is probably the best known example of a major language so implemented. See R. E. Griswold, J. F. Poage and I. P. Polonsky, *The Snobol4 Programming Language,* Prentice-Hall, 1969, or R. E. Griswold, *The Macro Implementation of Snobol4,* Freeman, 1972. The book by Brown discusses other work in this area.

Any number of books on data structures deal with the problems of maintaining tables of information. As usual, one standard reference is D. E. Knuth's *The Art of Computer Programming* (Addison-Wesley). Volume 1 (1968) is concerned with data structures; Volume 3 (1973) discusses searching techniques in great detail, including the selection of hashing functions.

It is possible to augment Pascal with a macro processor, even more than we have done, to overcome some of its drawbacks. For a description of one such effort, see "MAP: A Pascal macro preprocessor for large program development," by D. E. Comer, *Software Practice and Experience,* March, 1979.

The `macro` processor described in this chapter was originally designed and implemented in C by D. M. Ritchie; we are grateful to him for letting us steal it.