Find Patterns in Text Files

Kiyoshi Akima
http://kiyoshiakima.tripod.com/funprogs

2006.07.27

Contents

1 Find

`find FROM/A,TO/K,PAT/K,C/S,N/S`

The `find` command copies the file given by the `FROM` argument to the file given by the `TO` argument. In the process it scans the text for occurences of a search pattern. Only lines containing or not containing the search pattern are output; that is, `find` selects from a file just those lines containing a search pattern or just those lines not containing it.

The search pattern is given by the `PAT` argument and is formed according to the rules given below. If the pattern is prefixed by a tilde symbol (`~`), then lines not containing the pattern are output, otherwise lines containing the pattern are output.

If the `C` switch is given then case distinctions are ignored when matching.

If the `N` switch is given then the lines are numbered on output.

1.1 Patterns

The simplest form of a search pattern is a character string identical to the one sought.

Preceding the pattern with a grave accent (`‘`) specifies that the string must appear at the beginning of a line. The grave accent in any other position has no special meaning.

Terminating a pattern with an apostrophe (`’`) specifies that the string must occur at the end of a line. An apostrophe in any other position has no special meaning.

The following patterns illustrate their use:

pattern	**meaning**
`‘abcd`	the string “abcd” at the beginning of a line
`xyz’`	the string “xyz” at the end of a line
`‘xxx’`	a line consisting only of “xxx”
`ab’cd‘e`	the string “ab’cd‘e” occuring anywhere in a line

A question mark (`?`) in a pattern matches any character in that position of a string. Thus the pattern `f??t` matches “foot”, “feet”, “f it”, and so on.

An asterisk (`*`) causes a match on zero or more occurences of the preceding character. An asterisk at the beginning of a pattern has no special meaning. The following patterns illustrate the use of the asterisk:

pattern	**matching strings**
`*abc`	“*abc”
`a*bc`	“bc”, “abc”, “aabc”, “aaabc”, ...
`aa*bc`	“abc”, “aabc”, “aaabc”, “aaaabc”, ...
`s?*p`	“sp”, “sxp”, “sleep”, “s12 xp”, ...

1.2 Escape sequences

Sometimes it is necessary to enter nonprintable characters or characters that ordinarily have special meaning. You can enter such characters from the keyboard by using the colon as an escape character. An escape character changes the meaning of the character that follows it.

Together the escape character and the character following it are seen as a single character by the command. The escape sequences are:

`:b`	backspace
`:n`	newline
`:s`	space
`:t`	tab
`:<other character>`	the actual character given

Some special characters known as *metacharacters* have special meaning when they appear in the pattern. You may use the `<other character>` escape sequence to force them to be seen as themselves in these contexts.

The colon, having a special use (escape character), must be escaped to be accepted as itself; thus `::` is taken for a single colon.

Provision is made for the space character, since if an actual space were included in the pattern it would delimit the pattern.

Despite the provision of the newline character as an escape, this command will not match patterns spanning multiple lines. Perhaps a future enhancement...

1.3 Metacharacters

Certain characters assume special meanings when they appearn in the pattern. As a group these characters are designated *metacharacters* (as opposed to ordinary characters). Since these metacharacters occasionally need to appear as ordinary characters in a search pattern, they may, in such cases, be entered as escape sequences. The metacharacters are given below:

symbol	*name*	*use*
`:`	colon	escape character
`‘`	grave accent	matches the beginning of the line
`’`	apostrophe	matches the end of the line
`?`	question mark	matches any character
`*`	asterisk	matches zero or more occurences of the preceding character
`[`	left bracket	begins a character class definition
`]`	right bracket	ends a character class definition
`-`	hyphen	indicates a range of characters in a character class definition
`~`	tilde	complements a character class definition

The metacharacters are defined in Section 2.2. You may change these metacharacter assignments to suit your fancy by changing that section before tangling and compiling.

1.4 Character classes

Since the set of decimal digits, lowercase letters, and uppercase letters are used frequently, and since they are such long lists, a shorthand method of specifying `[012\dots9]`, `[abc\dots z]`, and `[ABC\dots Z]` exists. You may place a hyphen between the first and last characters. Thus, the pattern `a[0-9]` matches "a0", "a1", and so on.

You need not specify the entire set of decimal digits, nor all of the letters when the shorthand notation is used. You may give `[5-7]`, `[a-g]`, and so on. The only restrictions are that the lower-valued character must be listed in front of of the hyphen. You may use the shorthand notation in a list of characters specifying a character class. Thus, `[s12g5-7a-zA-Z$(]` is a valid character class.

The hyphen (`-`) has special meaning only when it falls between characters in a character class definition. If it appears at either end of the definition or outside such a definition, it has no special meaning.

If the first character inside the left bracket is a tilde `~`, it causes a match on any character except those listed.

It is important to think of the character class as a single character position.

If you need a literal `[` or `]` in a pattern, then escape it as `:[` or `:]`, respectively.

1.5 Rationale

Admittedly, this is a very limited program when compared to grep and its kin. So why bother writing and presenting such a command when every programmer already has the familiar and much more capable grep in his toolbox? While this command certainly won't replace grep and its kin in anyone's toolbox, it does have the advantage of working from within the BCPL interpreter cinterp. This makes it convenient for answering those quick "How did I spell that variable?" type of questions without having to leave the interpreter.

And, the program is big enough to be a nontrivial test of using literate programming techniques (Section 2) with BCPL while being small enough to be completed in a reasonable amount of time.

2 Literate Programs

This document not only describes the implementation of `find`, it *is* the implementation. The noweb system for "literate programming" generates both the document and the code from a single source. This source consists of interleaved prose and labelled code *fragments*. The fragments are written in the order that best suits describing the program, namely the order you see in this document, not the order dictated by the BCPL programming language. The program noweave accepts the source and produces the document's typescript, which includes all of the code and all of the text. The program notangle extracts all of the code, in the proper order for compilation.

Fragments contain source code and references to other fragments. Fragment definitions are preceded by their labels in angle brackets. For example, the code

4a ⟨*a fragment label* 4a⟩≡ 4c▷
```
sum := 0
FOR i = 1 TO 10 DO ⟨increment sum 4b⟩
```

4b ⟨*increment sum* 4b⟩≡ (4a)
```
sum := sum + x!i
```

sums the elements of `x`. Several fragments may have the same name; notangle concatenates their definitions to produce a single fragment. noweave identifies this concatenation by using + ≡ instead of ≡ in continued definitions:

4c ⟨*a fragment label* 4a⟩+≡ ◁4a
```
writef("%i*n", sum)
```

Fragment definitions are like macro definitions; notangle extracts a program by expanding one fragment. If its definition refers to other fragments, they themselves are expanded, and so on.

Fragment definitions include aids to help readers navigate among them. Each fragment name ends with the number of the page on which the fragment's definition begins and a letter giving its sequence within that page. If there is only one fragment on a page then there is no letter. This is also shown in the left margin. Each continued definition also shows the previous definition, and the next continued definition, if there is one. ◁ 7b is an example of a previous definition that appears on page 7, and 11 ▷ says the definition is continued on page 11. These annotations form a double linked list of definitions; the left arrow points to the previous definition in the list and the right arrow points to the next one. The previous link on the first definition is omitted, and the next link on the last definition is omitted. These lists are complete: If some of a fragment's definition appears on the same page with each other, the links refer to the page on which they appear.

Fragments also show a list of pages on which the fragment is used, as illustrated by the (4a) to the right of the definition for ⟨*increment sum*⟩, above.

2.1 The program

This program is translated from the Small-C program of the same name appearing in the Small-Tools package by J. E. Hendrix.

Translated into BCPL the program has the usual structure. The fragment name consisting of an asterisk indicates to noweb that this is the *root* fragment, which is expanded to generate the program.

5a ⟨* 5a⟩≡

```
GET "libhdr"

⟨manifests 5b⟩

⟨statics 6c⟩

⟨debug stuff 20a⟩

⟨procedure start 6a⟩
```

2.2 The metacharacters

All of the metacharacters are defined here. You may change them to suit your fancy before tangling and compiling. (And don't forget to change the documentation in Section 1.3.)

5b ⟨manifests 5b⟩≡ (5a) 10c▷

```
MANIFEST {
 Char    = 'c'      // identifies a character
 BoL     = '`'      // beginning of line
 EoL     = '*''     // end of line
 Any     = '?'      // any character
 CCl     = '['      // begin character class
 NCCl    = '^'      // negation of chracter class
 CClEnd  = ']'      // end of character class
 Closure = '**'     // zero or more occurences
 Escape  = ':'      // escape character
 NotC    = '^'      // negation character
}
```

3 The start procedure

By convention execution of a BCPL program begins with a call to start. In this program start processes its command line to get the pattern, compiles the pattern into an internal format, opens the input stream, searches for matching lines and prints them, and finally cleans up.

6a ⟨*procedure* start 6a⟩≡ (5a)

```
LET start() = VALOF {
 ⟨error procedures 19b⟩
 ⟨i/o procedures 11a⟩
 ⟨case conversion 7e⟩
 ⟨pattern creation procedures 13a⟩
 ⟨pattern matching procedures 17a⟩

 ⟨procedure start's variables 7a⟩

 ⟨prepare for bail out 6b⟩
 ⟨process command line 7b⟩
 ⟨compile pattern 8e⟩
 ⟨find input stream 9b⟩
 ⟨search for matching lines 9c⟩

fin:
 ⟨deallocate memory 8g⟩
 ⟨close streams 10b⟩

 RESULTIS 0
}
```

3.1 Bailing out in case of trouble

We hope we don't have to, but just in case...

6b ⟨*prepare for bail out* 6b⟩≡ (6a)

```
fin_p, fin_l := level(), fin
```

6c ⟨*statics* 6c⟩≡ (5a) 7d▷

```
STATIC {
 fin_p; fin_l
}
```

Now we can get to the cleanup code from anywhere within the program.

6d ⟨*bail out* 6d⟩≡ (19b)

```
longjump(fin_p, fin_l)
```

3.2 Processing the command line arguments

The first thing we have to do is to extract the arguments from the command line. The input stream and the pattern are required arguments, while an output stream is optional.

7a ⟨*procedure* `start`*'s variables* 7a⟩≡ (6a)
```
LET argv = VEC 10
```

We let `rdargs` process the command line for us.

7b ⟨*process command line* 7b⟩≡ (6a) 7c▷
```
IF 0 = rdargs("⟨rdargs argument 8a⟩", argv, 10)  DO ⟨error: usage 8b⟩
```

Ignoring case distinctions

We have to be a little careful in ignoring case distinctions. We can't simply map both the pattern and the input line to a single case, as then we wouldn't be able to print the original input line. Instead, we set a flag accordingly.

7c ⟨*process command line* 7b⟩+≡ (6a) ◁7b 7f▷
```
ignore_case := argv!3
```

And, of course, the flag needs to be defined before it can be used.

7d ⟨*statics* 6c⟩+≡ (5a) ◁6c 7g▷
```
STATIC {
 ignore_case
}
```

If the user specified case distinctions to be ignored, we convert letters to lower case for comparison purposes.

7e ⟨*case conversion* 7e⟩≡ (6a)
```
LET case(c) = ignore_case & ('A'<= c <= 'Z') ->
              c + 'a' - 'A',
              c
```

Numbering output lines

We're going to get a little tricky with the line counter. If line numbering is not specified then we initialize the counter to -1. If line numbering is specified then we initialize the counter to the number of lines read so far (which is zero). For this we rely on the fact that BCPL defines `TRUE` as -1. From this point on, the line count will get incremented upon newline only if the count is nonnegative.

7f ⟨*process command line* 7b⟩+≡ (6a) ◁7c
```
lcount := -1 - argv!4
```

We need to define the line counter.

7g ⟨*statics* 6c⟩+≡ (5a) ◁7d 8c▷
```
STATIC {
 lcount
}
```

We define the `rdargs` argument as a separate fragment because it will be passed on to `error` to tell the user what we expect on the command line. This way, if new options are added in the future, the error message will also be updated automatically.

8a ⟨*rdargs argument* 8a⟩≡ (7b 8b)
```
FROM/A,TO/K,PAT/A/K,C/S,N/S
```

Rather than merely telling the user his arguments are wrong, let's tell him what we expect.

8b ⟨*error: usage* 8b⟩≡ (7b)
```
error("Invalid args: FIND ⟨rdargs argument 8a⟩")
```

3.3 Compiling the pattern

The pattern must have been specified on the command line.

8c ⟨*statics* 6c⟩+≡ (5a) ◁7g 8d▷
```
STATIC {
 pbuf = 0
}
```

The user may have specified lines *not* matching the pattern by putting a tilde (~)as the first character of the pattern.

8d ⟨*statics* 6c⟩+≡ (5a) ◁8c 9a▷
```
STATIC {
 invert = 0
}
```

Now it's a matter of allocating the pattern buffer and compiling the pattern from the string given us on the command line.

8e ⟨*compile pattern* 8e⟩≡ (6a)
```
pbuf := getvec(MaxPat)
UNLESS pbuf DO ⟨error: no memory 8f⟩
⟨initialize pattern buffer 20b⟩
IF '~' = argv!2%1 DO invert := -1
UNLESS makpat(argv!2) error("Pattern too long")
```

In the unlikely event we run out of memory we want to let the user know why we terminated without doing any real work.

8f ⟨*error: no memory* 8f⟩≡ (8e 10a)
```
error("Insufficient memory")
```

The pattern needs to be deallocated when we're done.

8g ⟨*deallocate memory* 8g⟩≡ (6a) 9g▷
```
IF freevec DO freevec(pbuf)
```

3.4 Opening the input stream

The input stream must have been specified on the command line.

9a ⟨*statics* 6c⟩+≡ (5a) ◁8d 9d▷
```
STATIC {
 instream = 0
}
```

We attempt to find the file and if successful, select it for input.

9b ⟨*find input stream* 9b⟩≡ (6a)
```
instream := findinput(argv!0)
UNLESS instream DO error("Can't open input")
selectinput(instream)
```

3.5 Searching for matching lines

We open an output stream if one was specified, then print matching lines from the input stream.

9c ⟨*search for matching lines* 9c⟩≡ (6a)
```
⟨open output stream 9e⟩
⟨print matching lines 10a⟩
```

The user may have specified an output stream.

9d ⟨*statics* 6c⟩+≡ (5a) ◁9a 9f▷
```
STATIC {
 outstream = 0
}
```

If the user had specified an output stream then we find it and select it for output.

9e ⟨*open output stream* 9e⟩≡ (9c)
```
IF argv!1 DO {
 outstream := findoutput(argv!1)
 UNLESS outstream DO error("Can't open output")
 selectoutput(outstream)
}
```

We need a buffer to hold the input line. This buffer will be allocated when the command executes.

9f ⟨*statics* 6c⟩+≡ (5a) ◁9d
```
STATIC {
 lbuf = 0
}
```

And we must not forget to deallocate it when we're done with it.

9g ⟨*deallocate memory* 8g⟩+≡ (6a) ◁8g
```
IF lbuf DO freevec(lbuf)
```

Now we allocate the line buffer then read each line of input and see whether it matches.

10a ⟨*print matching lines* 10a⟩≡ (9c)
```
lbuf := getvec(MaxLine + 1)
UNLESS lbuf DO ⟨error: no memory 8f⟩
WHILE 0 <= readline() DO
 IF match() NEQV invert DO writeline()
```

3.6 Closing the streams

We close both the input stream and the output stream before terminating.

10b ⟨*close streams* 10b⟩≡ (6a)
```
IF instream  DO endread()
IF outstream DO endwrite()
```

4 Input/Output

This command would be rather useless if it couldn't perform any input or output.

This command works with individual lines. Even though patterns spanning multiple lines *could* be given on the command line, such patterns will **not** work.

The input line is stored unpacked (one character per word) in the variable `lbuf`. There is no length word; the line is terminated with a zero word. This means that the standard library routines such as `writes` cannot be used to write it out. It does mean that lines longer than 255 characters can be handled... as long as they're less than `MaxLine` characters.

4.1 Input

We will fold input lines longer than `MaxLine` characters. Note that this will throw off the line numbering, as long lines get counted as multiple lines.

10c ⟨*manifests* 5b⟩+≡ (5a) ◁5b 12b▷
```
MANIFEST {
 MaxLine = 1024
}
```

Since we're dealing with lines, we need a function to read a line from the input stream. The characters are placed unpacked into `lbuf`. A line is terminated by a newline or an end of stream, or if we reach `MaxLine` characters. Carriage returns are ignored, thus making the command virtually worthless for Mac-formatted files. But then, this is already rather worthless for true binary files.

This function returns the number of characters read, or -1 if end of stream.

11a ⟨*i/o procedures* 11a⟩≡ (6a) 12a▷

```
LET readline() = VALOF {
 LET i, ch = 0, ?

 ⟨increment line number 11b⟩
 lbuf!2 := 0
 WHILE i < MaxLine DO {
  ⟨read and store character 11c⟩
  ⟨handle possible line termination 11d⟩
 }
 ⟨store terminator 11e⟩
 RESULTIS i - 1
}
```

Remember that `lcount` serves double duty as both the line number and the line-numbering flag. We increment the humberonly if it is ≥ 0.

Heaven help anyone who uses this command on a text file with more than two billion lines.

11b ⟨*increment line number* 11b⟩≡ (11a)

```
UNLESS lcount < 0 DO lcount := lcount + 1
```

It would make life easier if we could map letters to a single case at this time to avoid case distinctions, but then we wouldn't be able to display the original line. Thus we must leave the input line alone and complicate the matching process somewhat instead.

11c ⟨*read and store character* 11c⟩≡ (11a)

```
i, ch, lbuf!i := i + 1, rdch(), ch
```

The character just read may terminate the line.

11d ⟨*handle possible line termination* 11d⟩≡ (11a)

```
SWITCHON ch INTO {
CASE endstreamch:   IF 1 = i    RESULTIS -1
CASE '*n':                      i := i - 1; BREAK
CASE '*c':                      i := i - 1
default:                        ENDCASE
}
```

We want to make sure the line buffer is terminated properly.

11e ⟨*store terminator* 11e⟩≡ (11a)

```
i, lbuf!i := i + 1, 0
```

4.2 Output

Since we're dealing with unpacked lines, we need a procedure to print one out. This is a lot easier than reading one since the only interpretation we have to apply to the line is searching for its end. And, since we stripped off the newline when we read it in, we have to tack one on at the end after writing it out.

12a ⟨*i/o procedures* 11a⟩+≡ (6a) ◁11a

```
AND writeline() BE {
 IF 0 <= lcount DO writef("%i5: ", lcount)
 FOR i = 1 TO MaxLine DO {
  UNLESS lbuf!i BREAK
  wrch(lbuf!i)
 }
 newline()
}
```

5 Pattern creation

We want to compile the pattern into an internal format to make the matching easier. The compiled pattern is placed unpacked into the vector `pbuf`.

We have to put an upper limit on the size of the pattern.

12b ⟨*manifests* 5b⟩+≡ (5a) ◁10c 12c▷

```
MANIFEST {
 MaxPat = 257
}
```

Some symbolic names will help make it easier to deal with patterns.

12c ⟨*manifests* 5b⟩+≡ (5a) ◁12b

```
MANIFEST {
 Count = 1; PrevCl; StartCl; CloSize
}
```

Compile pattern specified by `arg` into pattern buffer `pbuf`.

13a ⟨*pattern creation procedures* 13a⟩≡ (6a)

```
LET makpat(arg) = VALOF {
 ⟨add set 13d⟩
 ⟨map escape character 14a⟩
 ⟨get character class 14b⟩
 ⟨insert closure 15b⟩

 LET i, j, lastcl, lastj, lj, from = ?, 1, -1, 1, ?, ?

 i, from := 1 - invert, i
 WHILE i <= arg%0 DO {
  lj := j
  TEST      Any     = arg%i                 THEN addset(Any, @j)
  ELSE TEST BoL     = arg%i & i = from      THEN addset(BoL, @j)
  ELSE TEST EoL     = arg%i & 0 = arg%(i+1) THEN addset(EoL, @j)
  ELSE TEST CCl     = arg%i                 THEN
   UNLESS getccl(arg, @i, @j)                   BREAK
  ELSE TEST Closure = arg%i & from < i      THEN {
   ⟨add closure 13c⟩
  } ELSE {
   ⟨add literal character 13b⟩
  }
  lastj, i := lj, i + 1
 }
 IF FALSE = addset(0, @j) | i < arg%0 RESULTIS FALSE
 RESULTIS TRUE
}
```

A literal character is added to the pattern by flagging it as such and then adding the character. Remember that the character *could* be an escaped character.

13b ⟨*add literal character* 13b⟩≡ (13a)

```
addset(Char, @j)
addset(case(esc(arg, @i)), @j)
```

A closure is zero or more occurrences of a character or class.

13c ⟨*add closure* 13c⟩≡ (13a)

```
lj := lastj
IF BoL = pbuf!lj | EoL = pbuf!lj | Closure = pbuf!lj BREAK
lastcl := stclos(@j, @lastj, lastcl)
```

This function puts character `c` into pattern buffer `pbuf` and increments index `!j`. It returns `FALSE` if the pattern buffer is full, `TRUE` otherwise.

13d ⟨*add set* 13d⟩≡ (13a)

```
LET addset(c, j) = VALOF {
 IF MaxPat <= !j RESULTIS FALSE
 pbuf!!j, !j := c, !j + 1
 RESULTIS TRUE
}
```

This function maps `array%i` into escaped character if appropriate. If the character `array%i` isn't `Escape` then it's easy — it's simply that character. If it is `Escape`, then we have to look at the next character. If the `Escape` is the last character in the pattern, then we simply have a literal asterisk. Otherwise, if the next character is one of the special escape sequences then we do the appropriate translation. Else it's simply the character following the `Escape`.

14a ⟨*map escape character* 14a⟩≡ (13a)

```
AND esc(array, i) = VALOF {
 TEST Escape ~= array%!i     RESULTIS array%!i
 ELSE TEST 0 =  array%(!i+1) RESULTIS Escape
 ELSE {
  !i := !i + 1
  SWITCHON array%!i INTO {
  CASE 't':                  RESULTIS '*t'
  CASE 'b':                  RESULTIS '*b'
  CASE 's':                  RESULTIS ' '
  DEFAULT:                   RESULTIS array%!i
  }
 }
}
```

This routine puts the character class at `arg%i` into `pbuf!j`.

14b ⟨*get character class* 14b⟩≡ (13a)

```
AND getccl(arg, i, j) = VALOF {
 ⟨expand hyphen 16a⟩

 LET jstart = ?
 LET digit  = "0123456789"
 LET loalf  = "abcdefghijklmnopqrstuvwxyz"
 LET upalf  = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

 !i := !i + 1
 TEST NotC = arg%!i THEN {
  addset(NCCl, j)
  !i := !i + 1
 } ELSE addset(CCl, j)
 ⟨expand class 15a⟩
 RESULTIS CClEnd = arg%!i
}
```

Expand character class in `arg` into `pbuf`.

15a ⟨*expand class* 15a⟩≡ (14b)

```
jstart := !j
addset(0, j)
WHILE arg%!i & CClEnd ~= arg%!i DO {
 TEST      Escape = arg%!i           THEN addset(esc(arg, i), j)
 ELSE TEST '-'    ~= arg%!i          THEN addset(arg%!i, j)
 ELSE TEST j <= 1 | 0 = arg%!i       THEN addset('-', j)
 ELSE TEST '0' <= pbuf!(!j-1) <= '9' THEN dodash(digit, arg, i, j)
 ELSE TEST 'a' <= pbuf!(!j-1) <= 'z' THEN dodash(loalf, arg, i, j)
 ELSE TEST 'A' <= pbuf!(!j-1) <= 'Z' THEN dodash(upalf, arg, i, j)
 ELSE                                addset('-', j)
 !i := !i + 1
}
pbuf!jstart := !j - jstart - 1
```

This function inserts a closure entry at `pbuf!j`.

15b ⟨*insert closure* 15b⟩≡ (13a)

```
AND stclos(j, lastj, lastcl) = VALOF {
 LET jp, jt = ?, ?

 jp := !j - 1
 WHILE !lastj <= jp DO {
  jt := jp + CloSize
  addset(pbuf!jp, @jt)
  jp := jp - 1
 }
 ⟨put closure 15c⟩
 RESULTIS jp
}
```

We have to ensure we leave appropriate space in the pattern buffer when we place the closure there.

15c ⟨*put closure* 15c⟩≡ (15b)

```
!j, jp := !j + CloSize, !lastj
addset(Closure, lastj)
addset(0,       lastj)
addset(lastcl,  lastj)
addset(0,       lastj)
```

This routine is used to expand the character range `arg%(i-1) - arg%(i+1)` into `pbuf!j` ...

16a ⟨*expand hyphen* 16a⟩≡ (14b)

```
LET dodash(set, arg, i, j) BE {
 ⟨find character 16d⟩

 LET lower, upper = ?, ?

 ⟨determine range limits 16b⟩
 ⟨put range 16c⟩
}
```

16b ⟨*determine range limits* 16b⟩≡ (16a)

```
!i, !j := !i + 1, !j - 1
upper, lower := index(set, esc(arg, i)), index(set, pbuf!!j)
```

16c ⟨*put range* 16c⟩≡ (16a)

```
WHILE lower <= upper DO {
 addset(case(set%lower), j)
 lower := lower + 1
}
```

This function attempts to find the character `c` in string `s`. It returns the index if found, -1 otherwise.

16d ⟨*find character* 16d⟩≡ (16a)

```
LET index(s, c) = VALOF {
 LET i = 1

 WHILE s%i DO {
  IF s%i = c RESULTIS i
  i := i + 1
 }
 RESULTIS -1
}
```

6 Pattern matching

These procedures try to match the compiled pattern in `pbuf` against the input line in `lbuf`.

This function tries to match a pattern anywhere in `lbuf`.

17a ⟨*pattern matching procedures* 17a⟩≡ (6a)

```
LET match() = VALOF {
 ⟨look for match 17b⟩

 LET i = 1

 WHILE TRUE DO {
  IF 0 <= amatch(i) RESULTIS TRUE
  i := i + 1
  UNLESS lbuf!i RESULTIS FALSE
 }
}
```

This function looks for a match starting at `lbuf!from`.

17b ⟨*look for match* 17b⟩≡ (17a)

```
LET amatch(from) = VALOF {
 ⟨match single pattern 18b⟩

 LET i, j, offset, stack = ?, 1, ?, -1

 offset := from
 WHILE pbuf!j DO {
  TEST Closure = pbuf!j THEN {
   ⟨match closure 17c⟩
  } ELSE UNLESS omatch(@offset, j) DO {
   ⟨match non-closure 18a⟩
  }
  ⟨increment by pattern size 19a⟩
 }
 RESULTIS offset
}
```

Try to match a closure.

17c ⟨*match closure* 17c⟩≡ (17b)

```
stack := j
j := j + CloSize
i := offset
WHILE lbuf!i UNLESS omatch(@i, j) BREAK
pbuf!(stack+Count) := i - offset
pbuf!(stack+StartCl) := offset
offset := i
```

Try to match something other than a closure.

18a ⟨*match non-closure* 18a⟩≡ (17b)

```
WHILE 0 <= stack DO {
 IF 0 < pbuf!(stack+Count) BREAK
 stack := pbuf!(stack+PrevCl)
}
IF stack < 0 RESULTIS -1
pbuf!(stack+Count) := pbuf!(stack+Count) - 1
j := stack + CloSize
offset := pbuf!(stack+StartCl) + pbuf!(stack+Count)
```

This function attempts to match a single pattern at `pbuf!j`. If we've been toldto ignore case distinctions then we map any upper case input input characters to lowercase before attempting a match.

18b ⟨*match single pattern* 18b⟩≡ (17b)

```
LET omatch(i, j) = VALOF {
 ⟨locate character in class 18c⟩

 LET bump, c = -1, case(lbuf!!i)

 TEST      BoL  = pbuf!j  IF     1 = !i     bump := 0
 ELSE TEST EoL  = pbuf!j  UNLESS lbuf!!i    bump := 0
 ELSE TEST 0    = lbuf!!i                   RESULTIS FALSE
 ELSE TEST Char = pbuf!j  IF     case(lbuf!!i) = pbuf!(j+1)
                                             bump := 1
 ELSE TEST Any  = pbuf!j                     bump := 1
 ELSE TEST CCl  = pbuf!j  IF     locate(case(lbuf!!i), j + 1)
                                             bump := 1
 ELSE TEST NCCl = pbuf!j  UNLESS locate(case(lbuf!!i), j + 1)
                                             bump := 1
 ELSE error("In omatch: can't happen")
 IF 0 <= bump THEN {
  !i := !i + bump
  RESULTIS TRUE
 }
 RESULTIS FALSE
}
```

This function tries to locate the character `c` in the character class beginning at `offset`

18c ⟨*locate character in class* 18c⟩≡ (18b)

```
LET locate(c, offset) = VALOF {
 LET i = offset + pbuf!offset

 WHILE offset < i DO {
  IF c = pbuf!i RESULTIS TRUE
  i := i - 1
 }
 RESULTIS FALSE
}
```

Determine the size of the entry at `pbuf!j` and increment `j` accordingly.

19a ⟨*increment by pattern size* 19a⟩≡ (17b)

```
TEST      Char    = pbuf!j   THEN j := j + 2
ELSE TEST BoL     = pbuf!j | EoL = pbuf!j | Any = pbuf!j
                             THEN j := j + 1
ELSE TEST CCl     = pbuf!j | NCCl = pbuf!j
                             THEN j := j + 2 + pbuf!(j+1)
ELSE TEST Closure = pbuf!j   THEN j := j + CloSize
ELSE error("In amatch: can't happen")
```

7 Error reporting

Before we can print the error message we have to ensure that we're writing to the console where it can be seen by the user. Then we can print the message and tack on a newline. Rather than terminating the program here, we bail out to the cleanup code at the end of `start`.

19b ⟨*error procedures* 19b⟩≡ (6a)

```
LET error(msg) BE {
 IF outstream DO {
  endwrite()
  outstream := 0
 }
 selectoutput(findoutput("**"))
 writes(msg)
 newline()
 ⟨bail out 6d⟩
}
```

8 Debugging

The program is functional enough for the author's purposes. However some debugging code is still left in the program. Once written, this code looked too good to just throw away. Conditional compilation keeps the code from being compiled into the final command but still available if needed in the future. To activate, remove the two slashes at the beginning of the first line of this code fragment.

20a ⟨*debug stuff* 20a⟩≡ (5a)

```
//$$Debug
$<Debug
⟨debug procedures 20c⟩
$>Debug
```

8.1 The pattern buffer

When debugging it's useful to have the pattern buffer initialized to a known state. In this case the known state is all zeros, which is also the terminator.

20b ⟨*initialize pattern buffer* 20b⟩≡ (8e)

```
$<Debug
 FOR c = 0 TO MaxPat DO pbuf!c := 0
$>Debug
```

This function dumps out the entire pattern buffer in hexadecimal. The buffer shouldn't come close to being filled up in normal use, and it should be properly terminated (we initialized it to be filled with zeros). However if some errant code should inadvertently plant a zero into it we want to be able to see it.

20c ⟨*debug procedures* 20c⟩≡ (20a) 21a▷

```
LET DbgDumpPattBuf() BE {
 FOR i = 1 TO MaxPat DO {
  writef(" %x3", pbuf!i)
  UNLESS i REM 16 DO newline()
 }
 newline()
}
```

8.2 The input line

Just in case you ever suspect the weirdness is happening in the input, this procedure dumps out the input buffer in hexadecimal. This one does terminate on hitting a zero.

21a ⟨*debug procedures* 20c⟩+≡ (20a) ◁20c

```
LET DbgDumpLineBuf() BE {
 LET i = 0

 {
  i := i + 1
  writef(" %x3", lbuf!i)
  UNLESS i REM 16 DO newline()
 } REPEATWHILE lbuf!i
 newline()
}
```

8.3 Calling the debug procedures

And we'll provide conditional calls to these dump procedures. These fragments may be sprinkled in wherever needed.

21b ⟨*dump pattern buffer* 21b⟩≡

```
$<Debug
DbgDumpPattBuf()
$>Debug
```

21c ⟨*dump line buffer* 21c⟩≡

```
$<Debug
DbgDumpLineBuf()
$>Debug
```

9 Change Log

2004.06.28	Began work
	Translation into BCPL from the original C by J. E. Hendrix
	Conversion into the literate programming system noweb
2004.07.13	Initial semi-public release (to M. Richards)
2004.08.25	Added `C` switch to ignore case
2005.06.27	Cosmetic documentation changes

A Index of Code Fragments

Underlined entries are to the definition of the Code Fragment. In many cases, the definition of a fragment can be continued from one piece to another.

B Index of Identifiers

Underlined entries are their definitions. Standard library definitions are not listed here. Nor are FOR control variables and most other variables local to a procedure.