Programming Techniques

R. M. McCLURE, Editor

Regular Expression Search Algorithm

KEN THOMPSON
Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

A method for locating specific character strings embedded in character text is described and an implementation of this method in the form of a compiler is discussed. The compiler accepts a regular expression as source language and produces an IBM 7094 program as object language. The object program then accepts the text to be searched as input and produces a signal every time an embedded string in the text matches the given regular expression. Examples, problems, and solutions are also presented.

KEY WORDS AND PHRASES: search, match, regular expression
CR CATEGORIES: 3.74, 4.49, 5.32

The Algorithm

Previous search algorithms involve backtracking when a partially successful search path fails. This necessitates a lot of storage and bookkeeping, and executes slowly. In the regular expression recognition technique described in this paper, each character in the text to be searched is examined in sequence against a list of all possible current characters. During this examination a new list of all possible next characters is built. When the end of the current list is reached, the new list becomes the current list, the next character is obtained, and the process continues. In the terms of Brzozowski [1], this algorithm continually takes the left derivative of the given regular expression with respect to the text to be searched. The parallel nature of this algorithm makes it extremely fast.

The Implementation

The specific implementation of this algorithm is a compiler that translates a regular expression into IBM 7094 code. The compiled code, along with certain runtime routines, accepts the text to be searched as input and finds all substrings in the text that match the regular expression. The compiling phase of the implemention does not detract from the overall speed since any search routine must translate the input regular expression into some sort of machine accessible form.

In the compiled code, the lists mentioned in the algorithm are not characters, but transfer instructions into the compiled code. The execution is extremely fast since a transfer to the top of the current list automatically searches for all possible sequel characters in the regular expression.

This compile-search algorithm is incorporated as the context search in a time-sharing text editor. This is by no means the only use of such a search routine. For example, a variant of this algorithm is used as the symbol table search in an assembler.

It is assumed that the reader is familiar with regular expressions [2] and the machine language of the IBM 7094 computer [3].

The Compiler

The compiler consists of three concurrently running stages. The first stage is a syntax sieve that allows only syntactically correct regular expressions to pass. This stage also inserts the operator "$\cdot$" for juxtaposition of regular expressions. The second stage converts the regular expression to reverse Polish form. The third stage is the object code producer. The first two stages are straightforward and are not discussed. The third stage expects a syntactically correct, reverse Polish regular expression.

The regular expression $a(b \mid c)*d$ will be carried through as an example. This expression is translated into $abc \mid * \cdot d \cdot$ by the first two stages. A functional description of the third stage of the compiler follows:

The heart of the third stage is a pushdown stack. Each entry in the pushdown stack is a pointer to the compiled code of an operand. When a binary operator ("|" or "$\cdot$") is compiled, the top (most recent) two entries on the stack are combined and a resultant pointer for the operation replaces the two stack entries. The result of the binary operator is then available as an operand in another operation. Similarly, a unary operator ("*") operates on the top entry of the stack and creates an operand to replace that entry. When the entire regular expression is compiled, there is just one entry in the stack, and that is a pointer to the code for the regular expression.

The compiled code invokes one of two functional routines. The first is called NNODE. NNODE matches a single character and will be represented by an oval containing the character that is recognized. The second functional routine is called CNODE. CNODE will split the

current search path. It is represented by ⊕ with one input path and two output paths.

Figure 1 shows the functions of the third stage of the compiler in translating the example regular expression. The first three characters of the example *a*, *b*, *c*, each create a stack entry, *S*[*i*], and an NNODE box.

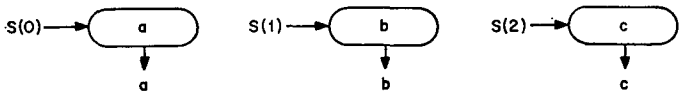

FIG. 1

The next character "*" combines the operands *b* and *c* with a CNODE to form *b*|*c* as an operand. (See Figure 2.)

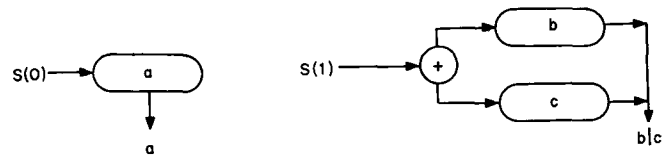

FIG. 2

The next character "*" operates on the top entry on the stack. The closure operator is realized with a CNODE by noting the identity $X* = \lambda|XX*$, where X is any regular expression (operand) and λ is the null regular expression. (See Figure 3.)

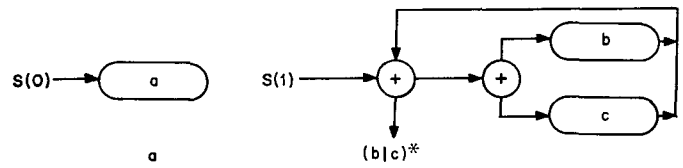

FIG. 3

The next character "·" compiles no code, but just combines the top two entries on the stack to be executed sequentially. The stack now points to the single operand $a \cdot (b|c)*$. (See Figure 4.)

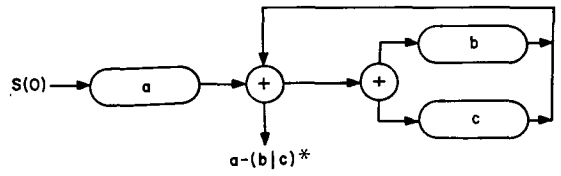

FIG. 4

The final two characters *d*· compile and connect an NNODE onto the existing code to produce the final regular expression in the only stack entry. (See Figure 5.)

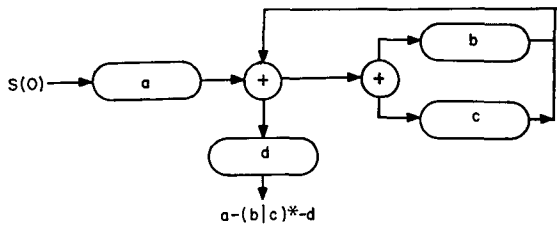

FIG. 5

A working example of the third stage of the compiler appears below. It is written in ALGOL-60 and produces object programs in IBM 7094 machine language.

```
begin
  integer procedure get character; code;
  integer procedure instruction(op, address, tag, decrement);
    code;
  integer procedure value(symbol); code;
  integer procedure index(character); code;
  integer char, lc, pc;
  integer array stack[0:10], code[0:300];
  switch switch := alpha, juxta, closure, or, eof;
  lc := pc := 0;
advance:
  char := get character;
  go to switch[index(char)];
alpha:
  code[pc] := instruction('tra', value('code')+pc+1, 0, 0);
  code[pc+1] := instruction('txl', value('fail'), 1, -char-1);
  code[pc+2] := instruction('txh', value('fail'), 1, -char);
  code[pc+3] := instruction('tsx', value('nnode'), 4, 0);
  stack[lc] := pc;
  pc := pc+4;
  lc := lc+1;
  go to advance;
juxta:
  lc := lc-1;
  go to advance;
closure:
  code[pc] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+1] := code[stack[lc-1]];
  code[stack[lc-1]] := instruction('tra', value('code')+pc, 0, 0);
  pc := pc+2;
  go to advance;
or:
  code[pc] := instruction('tra', value('code')+pc+4, 0, 0);
  code[pc+1] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+2] := code[stack[lc-1]];
  code[pc+3] := code[stack[lc-2]];
  code[stack[lc-2]] := instruction('tra', value('code')+pc+1, 0, 0);
  code[stack[lc-1]] := instruction('tra', value('code')+pc+4, 0, 0);
  pc := pc+4;
  lc := lc-1;
  go to advance;
eof:
  code[pc] := instruction('tra', value('found'), 0, 0);
  pc := pc+1
end
```

The integer procedure *get character* returns the next character from the second stage of the compiler. The

integer procedure *index* returns an integer index to classify the character. The integer procedure *value* returns the location of a named subroutine. It is an assembler symbol table routine. The integer procedure *instruction* returns an assembled 7094 instruction.

When the compiler receives the example regular expression, the following 7094 code is produced:

```
CODE    TRA     CODE+1          0       a
        TXL     FAIL,1,-'a'-1   1
        TXH     FAIL,1,-'a'     2
        TSX     NNODE,4         3
        TRA     CODE+16         4       b
        TXL     FAIL,1,-'b'-1   5
        TXH     FAIL,1,-'b'     6
        TSX     NNODE,4         7
        TRA     CODE+16         8       c
        TXL     FAIL,1,-'c'-1   9
        TXH     FAIL,1,-'c'     10
        TSX     NNODE,4         11
        TRA     CODE+16         12      |
        TSX     CNODE,4         13
        TRA     CODE+9          14
        TRA     CODE+5          15
        TSX     CNODE,4         16      *
        TRA     CODE+13         17
        TRA     CODE+19         18      ·d
        TXL     FAIL,1,-'d'-1   19
        TXH     FAIL,1,-'d'     20
        TSX     NNODE,4         21
        TRA     FOUND           22      ·eof
```

Runtime Routines

During execution of the code produced by the compiler, two lists (named CLIST and NLIST) are maintained by the subroutines CNODE and NNODE. CLIST contains a list of TSX **,2 instructions terminated by a TRA XCHG. Each TSX represents a partial match of the regular expression and the TRA XCHG represents the end of the list of possible matches. A call to CNODE from location x moves the TRA XCHG instruction down one location in CLIST and inserts in its place a TSX $x+1,2$ instruction. Control is then returned to $x+2$. This effectively branches the current search path. The path at $x+1$ is deferred until later while the branch at $x+2$ is searched immediately. The code for CNODE is as follows:

```
CNODE   AXC   **,7          CLIST COUNT
        CAL   CLIST,7
        SLW   CLIST+1,7     MOVE TRA XCHG
                              INSTRUCTION
        PCA   ,4
        ACL   TSXCMD
        SLW   CLIST,7       INSERT NEW TSX **,2
                              INSTRUCTION
        TXI   *+1,7,-1
        SCA   CNODE,7       INCREMENT CLIST
                              COUNT
        TRA   2,4           RETURN
*
TSXCMD  TSX   1,2           CONSTANT, NOT
                              EXECUTED
```

The subroutine NNODE is called after a successful match of the current character. This routine, when called from location x, places a TSX $x+1,2$ in NLIST. It then returns to the next instruction in CLIST. This sets up the place in CODE to be executed with the next character. The code for NNODE is as follows:

```
NNODE   AXC   **,7          NLIST COUNT
        PCA   ,4
        ACL   TSXCMD
        SLW   NLIST,7       PLACE NEW TSX **,2
                              INSTRUCTION
        TXI   *+1,7,-1
        SCA   NNODE,7       INCREMENT NLIST
                              COUNT
        TRA   1,2
```

The routine FAIL simply returns to the next entry in the current list CLIST.

```
FAIL    TRA   1,2
```

The routine XCHG is transferred to when the current list is exhausted. This routine copies NLIST onto CLIST, appends a TRA XCHG instruction, gets a new character in index register one, and transfers to CLIST. The instruction TSX CODE,2 is also executed to start a new search of the entire regular expression with each character. Thus the regular expression will be found anywhere in the text to be searched. Variations can be easily incorporated. The code for XCHG is:

```
XCHG    LAC   NNODE,7       PICK UP NLIST COUNT
        AXC   0,6           PICK UP CLIST COUNT
X1      TXL   X2,7,0
        TXI   *+1,7,1
        CAL   NLIST,7
        SLW   CLIST,6       COPY NLIST ONTO CLIST
        TXI   X1,6,-1
X2      CLA   TRACMD
        SLW   CLIST,6       PUT TRA XCHG AT
                              BOTTOM
        SCA   CNODE,6       INITIALIZE CNODE
                              COUNT
        SCA   NNODE,0       INITIALIZE NNODE
                              COUNT
        TSX   GETCHA,4
        PAC   ,1            GET NEXT CHARACTER
        TSX   CODE,2        START SEARCH
        TRA   CLIST         FINISH SEARCH
*
TRACMD  TRA   XCHG          CONSTANT, NOT
                              EXECUTED
```

Initialization is required to set up the initial lists and start the first character.

```
INIT    SCA   NNODE,0
        TRA   XCHG
```

The routine FOUND is transferred to for each successful match of the entire regular expression. There is a one character delay between the end of a successful match and the transfer to FOUND. The null regular expression is found on the first character while one character regular expressions are found on the second character. This means that an extra (end of file) character must be put through

the code in order to obtain complete results. FOUND depends upon the use of the search routine and is therefore not discussed in detail.

The integer procedure GETCHA (called from XCHG) obtains the next character from the text to be searched. This character is right adjusted in the accumulator. GETCHA must also recognize the end of the text and terminate the search.

Notes

Code compiled for $a**$ will go into a loop due to the closure operator on an operand containing the null regular expression, λ. There are two ways out of this problem. The first is to not allow such an expression to get through the syntax sieve. In most practical applications, this would not be a serious restriction. The second way out is to recognize lambda separately in operands and remember the CODE location of the recognition of lambda. This means that $a*$ is compiled as a search for $\lambda|aa*$. If the closure operation is performed on an operand containing lambda, the instruction TRA FAIL is overlaid on that portion of the operand that recognizes lambda. Thus $a**$ is compiled as $\lambda|aa*(aa*)*$.

The array *lambda* is added to the third stage of the previous compiler. It contains zero if the corresponding operand does not contain λ. It contains the *code* location of the recognition of λ if the operand does contain λ. (The *code* location of the recognition of λ can never be zero.)

```
begin
  integer procedure get character; code;
  integer procedure instruction(op, address, tag, decrement);
    code;
  integer procedure value(symbol); code;
  integer procedure index(character); code;
  integer char, lc, pc;
  integer array stack, lambda[0:10], code[0:300];
  switch switch := alpha, juxta, closure, or, eof;
  lc := pc := 0;
advance:
  char := get character;
  go to switch[index(char)];
alpha:
  code[pc] := instruction('tra', value('code')+pc+1, 0, 0);
  code[pc+1] := instruction('txl', value('fail'), 1, -char-1);
  code[pc+2] := instruction('txh', value('fail'), 1, -char);
  code[pc+3] := instruction('tsx', value('nnode'), 4, 0);
  stack[lc] := pc;
  lambda[lc] := 0;
  pc := pc+4;
  lc := lc+1;
 go to advance;
juxta:
  if lambda[lc-1] = 0 then
    lambda[lc-2] := 0;
  lc := lc-1;
  go to advance;
closure:
  code[pc] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+1] := code[stack[lc-1]];
  code[pc+2] := instruction('tra', value('code')+pc+6, 0, 0);
  code[pc+3] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+4] := code[stack[lc-1]];
  code[pc+5] := instruction('tra', value('code')+pc+6, 0, 0);
  code[stack[lc-1]] := instruction('tra', value('code')+pc+3, 0, 0);
  if lambda[lc-1] ≠ 0 then
    code[lambda[lc-1]] := instruction('tra', value('fail'), 0, 0);
  lambda[lc-1] := pc+5;
  pc := pc+6;
  go to advance;
or:
  code[pc] := instruction('tra', value('code')+pc+4, 0, 0);
  code[pc+1] := instruction('tsx', value('cnode'), 4, 0);
  code[pc+2] := code[stack[lc-1]];
  code[pc+3] := code[stack[lc-2]];
  code[stack[lc-2]] := instruction('tra', value('code')+pc+1, 0, 0);
  code[stack[lc-1]] := instruction('tra', value('code')+pc+4, 0, 0)
  if lambda[lc-2] = 0 then
    begin if lambda[lc-1] ≠ 0 then
      lambda[lc-2] = lambda[lc-1]
    end else
    if lambda[lc-1] ≠ 0 then
      code[lambda[lc-1]] :=
        instruction('tra', value('code')+lambda[lc-2], 0, 0);
  pc := pc+4;
  lc := lc-1;
  go to advance;
eof:
  code[pc] := instruction('tra', value('found'), 0, 0);
  pc := pc+1
end
```

The next note on the implementation is that the sizes of the two runtime lists can grow quite large. For example, the expression $a*a*a*a*a*a*a*$ explodes when it encounters a few concurrent a's. This expression is equivalent to $a*$ and therefore should not generate so many entries. Such redundant searches can be easily terminated by having NNODE (CNODE) search NLIST (CLIST) for a matching entry before it puts an entry in the list. This now gives a maximum size on the number of entries that can be in the lists. The maximum number of entries that can be in CLIST is the number of TSX CNODE,4 and TSX NNODE,4 instructions compiled. The maximum number of entries in NLIST is just the number of TSX NNODE,4 instructions compiled. In practice, these maxima are never met.

The execution is so fast, that any other recognition and deleting of redundant searches, such as described by Kuno and Oettinger [4], would probably waste time.

This compiling scheme is very amenable to the extension of the regular expressions recognized. Special characters can be introduced to match special situations or sequences. Examples include: beginning of line character, end of line character, any character, alphabetic character, any number of spaces character, lambda, etc. It is also easy to incorporate new operators in the regular expression routine. Examples include: not, exclusive or, intersection, etc.

REFERENCES

1. Brzozowski, J. A. Derivatives of regular expressions. *J. ACM 11*, 4 (Oct. 1964), 481–494.
2. Kleene, S. C. Representation of events in nerve nets and finite automata. In *Automata Studies*, Ann. Math. Stud. No. 34. Princeton U. Press, Princeton, N.J., 1956, pp. 3–41.
3. IBM Corp. IBM 7094 principles of operation. File No. 7094-01, Form A22-6703-1.
4. Kuno, S., and Oettinger, A. G. Multiple-path syntactic analyzer. Proc. IFIP Congress, Munich, 1962, North-Holland Pub. Co., Amsterdam.