Regular Expression Search Algorithm

KEY WORDS: search, match, regular expression.

The algorithm

Previous search algorithms involve backtracking when a partially successful search path fails. This necessitates a lot of storage and bookkeeping, and is very slow. 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 in 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.

In the compiled code, the lists mentioned in the algorithm are not characters, but transfer instructions into the compiled code. This execution is extremely fast since a transfer to the top of the current list automatically searches for all possible sequent 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 "*" 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 is translated into the third stage of the compiler.

The heart of the third stage is a pushdown stack. Each entry in the pushdown stack is a pointer to the compiled code if an operand. When a binary operator ("*" or "+") is compiled, the top (most recent) two entries on the stack are combined and a resultant pointer for the operations replaces the two stack entries. The result of the binary operator is then available as an operand in another operation. Similarly, a unary operator ("*" or "+") operates as 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 involves 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. 

Fig. 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(a), and an XNODE box. 

The next character "c" combines the operands b and c 
with a "CNODE to form b-c=a" an operand. (See Fig. 2.)

The next character "-" operates on the top entry on 
the stack. The closure operator is realized by a CNODE by 
noting the identity X = λX, where X is any regular 
expression (operand) and λ is the null regular expression. 
(See Fig. 3.)

The next character "-" completes 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-λb+λc. (See Fig. 4.)

The final two characters d - compile and connect an 

**COMMUNICATIONS OF THE ACM**

Vol. 11 / Number 6 / June, 1968

420
Writing an Online Debugging Program for the Experienced User

PATT T. BARTY
Bell Telephone Laboratories, Inc., Holmdel, N. J.

Presently available online debugging routines are often unsatisfactory for the experienced user because they require unnecessarily rigid and complicated typing formats, make it difficult to recognize and correct typing errors, and consume excessive memory with intricate features. In a debugging program it is of prime importance that the program be simple, flexible, and comprehensible to the user and the debugging program can be improved by using certain techniques applicable to most online debugging programs.

These techniques are presented and are illustrated by their use in OPAK (special package), a debugging program coded for the PDP-5, a digital computer. The simplicity, ease of correction of user-detected typing errors result in computer action bordering on the positive—skilful time consuming messages are typed, and the user often must retypc whole sequences of commands to correct the error. Each of these defects can be illustrated with examples paraphrased from existing debugging programs.

1. Introduction

The past decade has witnessed a proliferation of small, versatile digital computers. It is difficult to find a special example that is not an experimental tool. They are generally programmed by experienced users; any novice expose computer becomes experienced in a matter of days. The most important single utility program is probably the online debugging program. Debugging programs are machine-supplied by computer manufacturers, such as DEC's DDT (digital debugging tape) and SDB's AID, and many more are written by users who desire special features. Examples of the latter may be found in computer user society bulletins and in the literature [1, 2, 3]. Unfortunately, existing utility programs are often difficult to use because, ironically, they try to appeal to the novice user by requiring an elaborate verbal intercommunication. The need for writing a debugging program for the expert is noted by Lampson [3]: "An interactive debugging system should not be designed for the occasional user. Its emphasis must be on completeness, convenience, and commonness, not on highly mmnimal commands and self-explanatory output." Lampson streamlines his program by using single-letter mnemonics and special control char-
acters, but does not address himself to the problem of correcting typing errors. The present paper contains further typing shortcuts useful in debugging, and stresses ease of error correction. The techniques described here can be applied to many existing programs. They are illustrated by specific reference to OPAK, a recently written debugging program. The concept of OPAK is limited to a discussion of its user-program communication, plus a brief outline of its features.

In Section 3 certain communication defects found in many recently written debugging programs are briefly illustrated, and in Section 3 examples of how to overcome these defects are given.

In Section 4 the balance between economy of core allocation and inclusion of elaborate features in the debugging program is discussed.

2. Difficulties with Present Programs

Commonly found defects in the user-program communication structure of debugging programs are: (1) too much user typing is required; (2) it is too difficult to correct user-detected typing errors; and (3) program-detected typing errors result in computer action bordering on the positive—unnecessary time consuming messages are typed, and the user often must retypc whole sequences of commands to correct the error. Each of these defects can be illustrated with examples paraphrased from existing debugging programs.

1. Too much typing. To aid the novice user, whole words must be typed as program directives (PUNCH, STORE, LOAD, etc.). For example, the output of the echo dump of lines 4314 to 4320, type OCTAL/(CAR RET); 4314*FP+2000/80/ARC RET. (The symbol ^ denotes escape.) All word typing may be any alteration of the indicated sequence results in an error return.

2. Corruption of user-detected errors. If the user detects his own typing error, he is generally required to hit a special key (perhaps EN) and then possibly a transmission code. This requires the user to key out an "error command" mode, possibly at the expense of wasting a whole line of good typing. The user may now take a separate sequence to avoid this command mode.

3. Unfairness in diagnosis. It can be infuriating to an experienced programmer to have to sit helplessly while typing error messages like "CONTROL WORD, PLEASE TRY AGAIN." In another example, one existing program waits for an entire line of text to be typed before doing any error detection, and then rejects the whole line if an error appears anywhere.

3. Streamlining the Debugging Program

In early 1964, A. D. House of Bell Telephone Laboratories wrote a program called "Oeteral Package for the PDP-5 Computer." It was an extremely compact program (less than 400 words), which had certain elegant algorithms for ease of typing and error diagnosing. The author, while at New York University in 1965, spent