software into the machine. This lead (sic) to the project of having students write debugging programs which fit into this top page. Since a locally written binary loader was written in 74 words, that meant 54 words were available in which the student had to write an interesting program.

The results of that assignment clearly demonstrated that significant programs could be written in a minimum of space. Some of the programs which resulted were: a dump program which dumped core onto the teletype at 8 words per line, a modify program which displayed the contents of a memory location onto the teletype and enabled the user to change the contents of a location via the teletype, and a punch program which punched a paper tape of specified core segments in a format which enabled it to be reloaded by the binary loader at a later time. These programs, in a very crude manner, approximated the facilities available in the standard DEC debugging program ODT (1), however, each one was significantly shorter, and only took 10 seconds to load from a teletype - thus it was useful for debugging programs since no high speed I/O device was available.

This example has some important consequences. While core is becoming less expensive, there are still applications where size is important. Instructors of programming courses should not completely lose sight of that fact. While one shouldn't stress size over all other considerations, the above exercise shows that small programs can do relatively powerful things, even on a relatively simple PDP 8 computer.

(1) Digital Equipment Corporation, Introduction to Programming, 1970 pp7.51-7.54

A PEDAGOGICAL MODEL FOR TOP-DOWN SYNTAX ANALYSIS

G. E. Hedrick
Department of Computing and Information Sciences
Oklahoma State University
Stillwater, Oklahoma 74074

This paper describes a technique that has been used to teach syntactic analysis to computer science students at Oklahoma State University. This technique is used in a course corresponding to course 2 as described by Wegner (5). The particular methods taught are modifications of those developed by Irons, and Cheatham and Sattley (2,3). The programming techniques used to implement the model are found in Barron (1). A concise summary of the theory involved appears in Keller and Wright (4).

A set of rules which describes the syntax of a language is called a grammar. The grammar specifies the valid strings in a language. If a string is valid it is called a sentence of the language. An example of a grammar is given in table 1.

Table 1. A simple grammar.

$$
\begin{aligned}
P &\rightarrow V\$P \\
P &\rightarrow V \\
V &\rightarrow a \\
V &\rightarrow b \\
V &\rightarrow c \\
&\vdots \\
V &\rightarrow z
\end{aligned}
$$

This grammar is equivalently stated in the familiar Backus Naur Form (BNF) in table 2.

Table 2. The BNF equivalent of the grammar given in table 1.

<P>::=V$P|V
<V>::=a|b|c|...|z

The symbol → is read "consists of" or "is defined to be." P is (arbitrarily) called the distinguished symbol: any complete entity is required to be a P if it is to be valid in the language whose syntax this grammar describes. This language is said to be the language generated by the grammar. In the sample grammar above, each item of the form Q → R is called a production. A production is a rule of the grammar. In the grammar there are terminal and non-terminal symbols. A terminal symbol could be called an atomic symbol -- it needs no further definition and stands for itself. A non-terminal symbol is sometimes called a metavariable; it is defined in terms of other non-terminal and terminal symbols. In the example grammar the set of non-terminal symbols is {P,V} and the set of

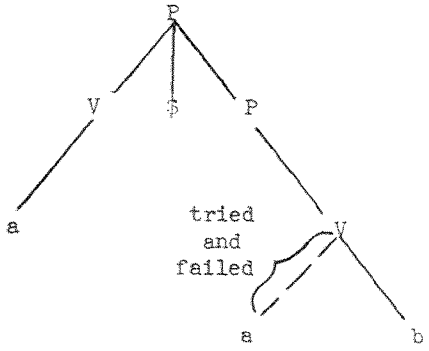
terminal symbols is {$,a,b,c,...,z}.

One common method of determining whether a string belongs to a given language is called top-down syntax analysis. In a top-down syntax analysis a tree search takes place. The root node of the tree is the first item to be searched for and the last item to be found in a successful search. The root node is always the distinguished symbol. The item that is being searched for is called a goal. In the preceding grammar the intitial goal is the distinguished symbol, P. From the first definition of P, V must be set as a goal and recognized before P can be recognized. This process continues until the goal is a terminal symbol, at which time a symbol from the input string can be examined. If it does not match the goal, the search backs up and the next alternative definition at that level is tested. If it does match, the program continues scanning the current definition.

The string a$b is valid in the language generated by the sample grammar. In order to show this, it is necessary to show that the string is a P in the grammar. The search would have the form shown in figure 1.

A top-down syntax analysis program such as the one described in this paper can be of significant value when teaching about production grammars. By using this program students can gain insight into the nature of production grammars; they can observe how the same program can be used with more than one grammar; and they can see how syntax trees can be grown in a top-down sense.

Figure 1. The form of the seach. Successive goals in the search are shown as nodes of the tree (read first top to bottom; then left to right.)



The program is used to demonstrate the concepts and structure of top-down syntax analysis. This top-down recognition scheme has been programed on the IBM 1130 at Oklahoma State University. This program, written in 1130 basic FORTRAN, reads a grammar by which it will analyze succeeding input strings. Each student is required to create and supply his own grammar. The restrictions on the nature of the grammar that the student supplies are described in detail later in this paper.

After the program reads the grammar, it reads one or more input strings, testing each string to determine whether it is valid in the language generated by the student's grammar. That is, the original grammar describes the syntax of some language and the syntax analysis program checks to determine if the input strings are valid in that language. If the input string is found to be valid, then a syntax tree (a graphical representation of the structure of the string, similar in nature to the grammatical diagram of a sentence in high school) for that string is produced on the console printer. If the input string is not valid, then a message to that effect is typed on the console printer.

The input to the syntax analysis program is divided into two distinct sections plus control cards, the distinct sections comprising the input of some grammar, G, and the input of some set of input strings, S_1, S_2, ..., S_n. The first of the control cards contains the number of productions in the grammar. (e.g., there are twenty-eight productions in table 1.) There is another input card which terminates the reading of the grammar. The final card with control information is the last card in the deck; it serves to terminate all input to the syntax analysis program.

A production can be considered to be a definition in the input grammar. The item being defined is called the left part of the production; the definition of that item is called the right part of the production. After the card which indicates the number of productions is read, each production of G is read in turn. It is possible for the same left part to have more than one right part; when this is the case, the successive right parts are called alternatives. All alternatives for a left part must be input as successive productions. The last input production is used only to flag the end of the grammar; in that role, it must have a left part which appears in the right part of no other production.

The student who makes use of this program is required to manipulate his grammar to a certain extent. Several things which could be done automatically by the program are relegated to the student as an exercise. The exercises permit him to observe the structure of the grammar and the organization of the top-down search. Internally the program uses unique positive integer values to represent

each non-terminal symbol. The distinguished symbol of the grammar always has the value of unity assigned to it. The integers may be chosen arbitrarily except for the fact that no one of these values can exceed the dimension of the array which is used to hold the grammar. To enable the students to make similar assignments to the non-terminal symbols, the following scheme was suggested: assign unity to the distinguished symbol; scan the list of productions counting lines during the scan; when a new left part is encountered, assign the line number to the non-terminal symbol which is the left part. Finally all terminal symbols in the grammar are assigned a value of zero. The positive integer assignments to the non-terminal symbols allow definitions for new goals to be found quickly during the search. A simple test for zero can be used to determine whether or not a symbol is a terminal symbol.

The last section of input that the student must supply comprises the strings to be analyzed. Each string consists of one to eighty characters. Any valid EBCDIC character, except a slash (/) in column one, is permitted in the input strings. The input characters must be terminal symbols in the grammar if an input string is to be valid. A card with a slash in column one signifies the end of input data; the program will terminate normally upon reading such a card.

After each input string S_i is read, a top-down analysis is begun. Pointers are set to the first production (the one which gives the initial definition of the distinguished symbol) and the first symbol on the right hand side of the production. These pointers are called G and C respectively. Another pointer is set so that it points to the first character in S_i.

The pointer, G, which was set to point to the distinguished symbol is said to point to the current goal -- the distinguished symbol is the initial goal. As the right side of a production is scanned, some non-terminal symbols may be encountered. Whenever a non-terminal symbol is found, the environment (of the program) at that time is placed in a stack and the non-terminal symbol just found is set up as a new goal. If a goal other than the original goal is recognized, the environment is restored so that the next symbol in the right part can be examined. After all of the symbols in a right part have been recognized, the symbol which is the left part is said to be recognized. This requires at least one production to contain only terminal symbols in its right part.

When a student uses this program he may obtain the output which was described previously or he may run the program in the trace mode. When the program is run in the trace mode, the search can be followed explicitly; it is possible to look at each step of the search by looking at each successive goal and examining the output for indications of "failures" and retries.

The model for top-down syntax analysis described in this paper has been used with success at Oklahoma State University. It has been used to demonstrate the general technique and to show how the structure of the grammar can affect the time required to perform analysis.

References

1. Barron, D. W., Recursive techniques in programming, American Elsevier Publishing Co., New York, New York, 1968.

2. Cheatham, T. E. and Sattley, Kirk, Syntax Directed compiling, Proc. American Federation of Information Processing Societies Spring Joint Computer Conference, 1964.

3. Irons, Edgar T., A syntax directed compiler for ALGOL60, Association for Computing Machinery Communications, 4(1961) 51-55.

4. Keller, R. F., and Wright, C. T., Algebraic languages and their translators, Holt, Rinehart and Winston, New York, New York, to be published.

5. Wegner, Peter, A view of computer science education, American Mathematical Monthly 79(1972) 168-179.