

A METHOD FOR DESIGNING A LEXICAL ANALYZER

BY

UDAY G. GUJAR

JOHN M. DEDOUREK

MARION E. MCINTYRE

TR79-015, MAY 1979

A METHOD FOR DESIGNING A LEXICAL ANALYZER

BY

Uday G. Gujar

School of Computer Science
University of New Brunswick
Fredericton, N.B.

John M. DeDourek

School of Computer Science
University of New Brunswick
Fredericton, N.B.

Marion E. McIntyre

New Brunswick Telephone Company
Saint John, N.B.

TR79-015, May 1979

This report has been submitted for outside publication and has been issued as a Research Report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely circulated until after the date of outside publication.

ABSTRACT

The scanner is a subroutine which is frequently called by an application program like a compiler. The primary function of a scanner is to combine characters from the input stream into recognizable units called tokens. A method has been presented in this paper for designing such a scanner, also frequently referred to as a lexical analyzer in the current literature. The major steps involved in this design process are: identification of tokens, construction of a state diagram, building driver tables and finally writing a scanning routine. The rules for generating the driver tables are described and an algorithm for the scanner, utilizing these driver tables, is included. The method has been successfully used to build the system scanner for a user oriented plotting language. It is concluded that the method is well defined, gives rise to a modular design and as such easily lends itself to language extensions.

1.0. Preliminary Remarks:

A scanner, or a lexical analyzer as it is referred to in the current literature, is a subroutine repetitively called by an application program, for example a compiler. Basically, the function of a scanner is to combine characters into recognizable units called tokens. Some typical examples of tokens in commonly occurring languages are strings enclosed in predefined delimiters, keywords, non-keywords, operators, etc.

The scanner examines characters from input records and extracts a token from them. The scanner then checks if this recognized token is of interest to the calling program and if so, it is returned to the calling routine along with an associated code. Examples of tokens which may not be of interest to any calling program would be comments, blanks, etc.

Although not all early compilers recognized the scanner as a separate module, the recent trend is to recommend this separation (see e.g. Aho and Ullman¹). This improves the maintainability and reliability of a compiler. In other areas involving the analysis of text, e.g. control languages for application packages, this separation is highly desirable and equally useful.

A scanner module must operate properly not only when presented with valid input but also generate an appropriate code when presented with invalid input. If a scanner is not well organized, it can be very difficult to guarantee that it will perform appropriately for all possible inputs.

Several scanner generators have appeared in the literature. Johnson et al², Delaney³, Lesser⁴, and Lesk⁵ have developed schemes based on

regular expressions. The user of such a system specifies the tokens to be recognized by the scanner in some modified form of regular expressions. A constructor program then generates either a scanner module or a set of tables to be interpreted by a supplied scanner module. This scanner module is then incorporated into the desired application. Other similar methods not based on regular expressions exist; for example, a method based on extracting tokens from a BNF grammar is described by Pierson and Lynch^{6,7}.

The difficulty with the above systems is that they are based on algorithms which are reasonably complex. Therefore, executing the algorithms by hand does not appear feasible. It is also difficult to justify the implementation of a complex scanner generator program unless the writing of scanners is to be a fairly frequent activity at a given location. The existing scanner generators do not seem to have been designed with portability in mind, thus limiting their widespread use.

The work reported in this paper attempts to extract the essentials of a scanner generator and incorporate them into a method which is simple to compute by hand. Though we have presented the method in this spirit, the table generator portion would be relatively easy to program. In order to keep the algorithm simple, this method utilizes a "one character look-ahead" scheme. Some features are incorporated into the scanner to accommodate special situations which are not economically handled by the basic algorithm (e.g. reserved words, a large number of one character operators). Certain modifications and extensions which allow wider use of the method

are also outlined.

2.0 Design and Implementation:

The proposed method of designing a scanner consists of the following four steps:

- a) Identify the tokens to be recognized.
- b) Draw a state diagram representation of the tokens to be recognized.
- c) Use this state diagram to build the driver tables which are used to determine if a token is complete.
- d) Write the scanning routine using these tables along with some additional auxiliary tables.

The above steps are discussed in the following subsections. A simple running example is included to illustrate the steps involved.

2.1 The Tokens:

Identifying the tokens to be recognized is a simple task if the input language is well defined. The tokens become apparent as a result of answering the types of questions that might be posed in a thorough analysis of the language. For example, if the language is keyword driven, the scanner has to be able to recognize keywords. What characters are valid in a keyword? The answer may be alphanumeric, only alphabetic or hyphenated alphanumeric. What forms of numbers does the language allow? The answer may be integers, real, exponential or all of these. This type of questioning continues until a satisfactory answer is found for all such questions; when this is achieved, a complete list of tokens is identified at this stage.

For our running example, consider the following token types:

1. An identifier is one or more characters long, is composed of letters and digits and contains at least one letter.
2. A number is composed of one or more digits.
3. A comment begins with /* and ends with */ and may contain any characters except the sequence */.
4. Blank space consists of one or more blanks.
5. The characters "+", "-", "*" and "/" are single character operators, while the character pair "/" is a double character operator.

These token types are summarized in Table 1.

2.2 The State Diagram:

The next step of the design is to represent the tokens to be recognized by a state diagram (also called transition diagram¹). A state diagram for a scanner consists of a set of nodes called states represented by circles, and a set of directed edges joining these states. These directed edges are labelled with character class names. The first step in arriving at the state diagram is the definition of these character classes.

The set of allowable characters (which may consist of all possible combinations in the input record) is partitioned into classes such that no character falls into more than one class. This partitioning is clearly a function of the tokens. For our running example, a suitable set of character classes is given in Table 2.

The state diagram is now drawn according to the following specifications:

- a) One state is distinguished as the initial state by the label START.
- b) Edges connecting the states are normally labelled by one or more character class names. An unlabelled edge represents an edge with all character classes which do not appear on any other edge coming out of that state.
- c) It is required that the state diagram be deterministic. A state diagram is deterministic if no more than one edge leaving a given state is labelled with the same character class. Because of this requirement, it follows that there can be at most one unlabelled edge leaving any given state.
- d) The directed edges indicate the flow. If one is currently in state s and there is an edge labelled with character class k joining it to state t , one moves from state s to t if the next character scanned is a member of the character class k .
- e) Some states are designated as "final states" by double circles. These represent possible identification of tokens and are labelled with the corresponding possible token types. A string of characters is a token of a given type if and only if there is a path from the START state to a final state for that token type.

A state diagram for the running example is given in Figure 1.

2.3 The Driver Tables:

The next stage is to encode the state diagram in two tables. The first one, called the NEXT STATE table, specifies the next state given the present state and the character class for the next character from the input record; the second one, called the OUTPUT table, specifies whether or not a token is complete (by a non-zero or zero entry). Each table has one row for each state in the state diagram and one column for each character class. The following mechanical steps are employed in constructing these tables. We will illustrate the application of these rules by generating the tables for the running example as we go along.

a) Step 1 - Basic Transitions:

If an edge labelled with character class c leads from state s to state t , enter t in row s column c in the NEXT STATE table. Note that every label on an edge creates one entry. An edge with no labels completes that entire row after labelled edges from a given state have been processed.

Set the corresponding entries in the OUTPUT table to zero indicating that all of these edges contain a character which is to be included in the token currently being formed. (See Table 3).

b) Step 2 - End of Input File Processing:

It is assumed that the procedure for obtaining the next character and translating it into a character class can detect the end of the input file. This procedure should be organized such that after the end of file has occurred, it will supply the artificial character class "end of file" to the scanner procedure on that and every subsequent call for a new

character. The scanner procedure is arranged to supply an "end of file" token type to whatever processor is making use of the scanner on every call subsequent to supplying the last actual token.

A new state is added to the scanner by adding a row to both tables. A column is also added to represent the new character class "end of file" (EOF). The "end of file" token is added to the list of token types. Fill the new row in the NEXT STATE table with its own row number (actually only the last column of this row is ever used). The corresponding row of the OUTPUT table is filled with the "end of file" token number. Finally into row 1 (the "START" state row) of the NEXT STATE table, place the new row number in the column corresponding to the "end of file" character class; the corresponding OUTPUT table entry is filled with the "end of file" token number. (This makes null inputs legitimate.) (See Table 4).

c) Step 3 - Error Detection Scheme:

A new row is added to both tables for detecting errors when the scanner is supplied with an input string which is not a sequence of valid token types. Adding this row corresponds to adding an error state. A new token type, the "error" token, is devised and represents a string of characters in the input which is skipped over before resuming processing of the characters into tokens. As a practical matter, the scanner could print an error message or simply supply the error token to the caller, allowing processing of the error to take place at a higher level.

A new column may be added at this step corresponding to a new class of "error chracters" (ERR). These are characters which are not valid in any

token but which could occur in an invalid input. Notice that in the particular example being considered, this added character class and column would not be necessary if all characters are allowed within comments. However, for the sake of completeness, it is assumed at this point that there are a few possible input characters which are not allowed in comments; these then form the "error" character class.

For each filled entry in row 1 ("START" row) of the NEXT STATE table, fill the corresponding entry in the new row of the NEXT STATE table with the same value; each corresponding entry of the OUTPUT table is filled with the number of the "error" token. (This ends error detection at the possible beginning of a new token.) The remaining unfilled entries of the new row in the NEXT STATE table are filled with the new row number itself; the corresponding entries in the OUTPUT table are set to zero. (This has the effect of continuing error detection until the possible start of a new token.) Finally in the "error" character class column of the row added in step 2 (the "end of file" row) enter the "end of file" row number in the NEXT STATE table and the "end of file" token number in the OUTPUT table. (This entry is never used and is merely filled for consistency.) The results of step 3 for the running example appear in Table 5.

d) Step 4 - Completion of "START" Row:

Complete row 1 (the "START" row) by filling any unfilled entries in the NEXT STATE table with the "error" row number and the corresponding entries in the OUTPUT table with zero. (This handles the case of an error at the beginning of the input stream.) (See Table 6.)

e) Step 5 - Completion of Rows Corresponding to "Final States":

In this step, any unfilled entries in those rows of the NEXT STATE and OUTPUT tables corresponding to states marked as "final states" in the state diagram are filled in. In each case, the entry in the NEXT STATE table is the same as the corresponding entry of row 1. The corresponding entry in the OUTPUT table is the token number of the token for which the row represents a final state. (This allows recognition and output of a completed token on occurrence of the first character of the next token and initiates processing of the next token.) (See Table 7.)

f) Step 6 - Error Detection for an Incomplete Token at the End of File:

Fill all unfilled entries in the "end of file" character class column of the NEXT STATE table with the row number of the "end of file" row. Fill the corresponding OUTPUT table entries with the "error" token number.

g) Step 7 - Error Detection for an Incomplete Token at Other than the End of File:

Fill all unfilled entries in the NEXT STATE table with the "error" row number; corresponding entries in the OUTPUT table are filled with zeros.

The final tables for the running example are given in Table 8.

2.4 Scanning Routines:

Now we are ready to write the scanner routines. However, we should consider a couple of practical cases which are commonly encountered in many languages for the sake of improving efficiency.

First, it is often the case that certain identifiers are reserved,

i.e. these identifiers represent special tokens and the scanner is expected to return these token codes. Let us call this table a KEYWORD table.

Second, there are often a large number of single character tokens and it is desirable to output individual token codes for each of these. Using separate final states, however, greatly enlarges the NEXT STATE and OUTPUT tables. As such, all these one character tokens are treated as a single type of token when designing the state diagram. A special additional table is added to specify the individual token codes; let us call this a ONECHAR table.

We can illustrate the use of these tables by modifying our running example to have the reserved key words DO and END as token types 8 and 9 respectively. Further we will associate the operators +, -, * and / with token codes 10, 11, 12 and 13 respectively. The associated tables are given in Table 9.

The major processes of the scanner algorithm, which is table driven, are outlined in the flowchart given in Figure 2. The following comments may be helpful while looking at this flowchart:

- a) The flowchart specifies a call to the GNC (Get Next Character) routine; this routine is assumed to supply the next character from the input stream (in CHAR) along with its character class (in CHRCLS). In the case that the input has been exhausted, GNC supplies the special "end of file" character class. Depending upon implementation considerations, GNC may be either a subroutine or in-line code.

- b) "maxlen" specifies the maximum number of characters allowed in a token. Note that the excess characters, if any, are ignored by the flowchart; a warning message could easily be produced by the scanner if desired.
- c) SCAN returns the token in TOKEN as a character string, token code in TOKCOD and the length of the token in TOKLEN.
- d) The variables STATE and CHAR are assumed to maintain their values from one call to the next.
- e) Note that the variable STATE has to be initialized to 1 (the starting state) prior to the first call to SCAN. Since this starting state is never reentered, STATE will never have this value again.

3.0 Extensions and Modifications:

It has been shown by an example how the general method works. However, sometimes the constraints imposed on the language by the available characters, the explicit error detection requirements, etc. require that these special conditions be accommodated.

As an example, internal character representations for an arithmetic minus sign and a hyphen are same in the EBCDIC character code used on most IBM computers. This poses a problem in accommodating hyphenated keywords in the language. The difficulty can be solved by defining the "minus" and "hyphen" as

```
Minus := {- followed by a digit or a decimal}  
Hyphen:= {- followed by anything but a digit or a decimal}
```

This is obviously a modification to the GNC routine which is required to look one character beyond the next character in the above case.

Another practical requirement is to detect error conditions in some specific instances. For example, it may be desirable, for certain applications, that an ambiguous case like 1.355.3 should generate an error message rather than the two numbers 1.355 and .3. Such error conditions can be easily incorporated by including explicit directed edges to the error state on the state diagram. Then the error state is no longer a detached state and has to be considered along with other states.

These extensions are included in the larger example given in the next section.

4.0 Larger Example:

The method described in this paper has been used to build the system scanner for generating a user oriented plotting language^{9,10}. The details of this language are beyond the scope of this paper. Basically, it is a keyword oriented language with flexible syntax and minimal rules. Though written primarily for non-programmers, this language is equally suitable for programmers as well.

The token for this language are:

1. Identifiers consisting of a sequence of letters and hyphens
2. Integers
3. Real numbers
4. Strings of characters in quotes

5. End of Command delimiter
6. Continuation delimiter
7. Sequence of characters to be ignored
8. End of record or comment delimiter
9. End of file
10. Error

The character classes are:

1. Letter:= {A B C D F ... Z}
2. Letter E:= {E}
3. Digit := {0 1 2 ... 9}
4. Plus := {+}
5. Minus := {- followed by a Digit or Decimal}
6. Hyphen := {- followed by anything but digit or decimal}
7. Decimal := {.
8. Quote := {'}
9. End of Command symbol
10. Continuation symbol
11. Ignore := {blank, all others not included in 1 to 10 above}
12. End of record, comment symbol (EOR)
13. End of file (EOF)
14. Error

Note that, though there are defaults, the characters in classes 9, 10, 11 and 12 are not fixed since the user may choose his own delimiters.

The state diagram for this example is given in Figure 3 while the corresponding NEXT STATE and OUTPUT tables appear in Table 10.

A scanner based on these tables has been written and is found to work very satisfactorily.

5.0 Conclusions:

A general method for building a scanner has been presented. It is believed that this method should help to produce a well documented, easily maintainable and reliable scanner program. The rules are explicit and can be mechanically followed. Provisions are made to incorporate some special features, like reserved words and a large number of one character operators, economically. Certain modifications and extensions which allow wider use of the method have been outlined. The method has been used to construct the scanner for a user oriented plotting language. No major problems were encountered in using the method. The results are satisfactory. Since the design of the scanner is highly modular, extensions to the language could be accommodated rather easily. More widespread use is necessary to identify the range of applicability and to suggest improvements which would extend this range.

References

1. A.V. Aho and J.D. Ullman, Principles of Compiler Design, Addison-Wesley Publishing Company, Reading, Massachusetts, 1977.
2. W.L. Johnson, J.H. Porter, S. Ackley and D.J. Ross, 'Automatic Generation of Efficient Lexical Processors Using Finite State Techniques', Comm A.C.M., 11, 805-813 (1968).
3. R.P. Delaney, Automatic Construction of a Lexical Analyzer from Regular Expressions, M.Sc. Thesis, University of New Brunswick, Fredericton, N.B., Canada, 1974.
4. B.C. Lesser, Implementation of a Scanner Generator, M.Sc. Report, University of New Brunswick, Fredericton, N.B., Canada, 1977.
5. M.E. Lesk, Lex - A Lexical Analyzer Generator, Computing Science Technical Report #39, Bell Laboratories, Murray Hill, N.J., 1975.
6. H.L. Pierson, A Finite Transducer Model of Compiler Scanners, Masters Thesis, Case Institute of Technology, Cleveland, Ohio, 1966.
7. H.L. Pierson and W.C. Lynch, A Finite Transducer Model for Compiler Lexical Scanners, Technical Report 1098, Case Western Reserve University, Cleveland, Ohio, 1968.
8. M.E. McIntyre, A User-Oriented Plotting Language, M.Sc. Thesis, University of New Brunswick, Fredericton, N.B., April 1979.
9. U.G. Gujar and M.E. McIntyre, A User-oriented Plotting Language, (in preparation).

TABLE 1: Tokens for the Running Example

<u>Token Number</u>	<u>Token Type</u>
1	Identifier
2	Number
3	Comment
4	Blanks
5	Operator

TABLE 2: Character Classes for Running Example

<u>Number</u>	<u>Character Class Name</u>	<u>Characters</u>
1	LET (Letter)	A B C Z
2	DIG (Digit)	0 1 2 9
3	OPC (Operator Characters)	+ -
4	STR (Star)	*
5	SLS (Slash)	/
6	BLK (Blank)	blank or space
7	OTH (Other)	all characters not listed above

