The Automatic Generation of Fast Lexical Analysers

V. P. HEURING
Department of Electrical and Computer Engineering, University of Colorado, Boulder, Colorado 80309 U.S.A.

SUMMARY

This paper describes lexical analyser generators that accept specifications for the basic symbols of a programming language, and produce directly executing lexical analyser in Pascal or C as output. The resulting analyser are up to five times as fast as the typical interpreted lexical analyser generated from arbitrary regular expressions and actions. An example is given showing how these tools may be combined with others in a compiler construction environment.

KEY WORDS  Lexical analysis  Programming environments  Compiler construction

INTRODUCTION

Nearly all of the lexical analysers built for production compilers are written by hand.\(^1\) The primary reason for this appears to be the wish to achieve the highest possible speed, since up to 50 per cent of total compilation time may be spent in converting input text to tokens.\(^2\) The only important characteristic of the lexical analyser is speed, which the designer trades against construction time. This is unlike the situation in the design of the semantic analyser and the code generator, where one must weigh a number of conflicting objectives, such as quality of emitted code against compiler speed, for example. Thus, in lexical analyser design and construction the issue of hand construction against automatic generation is clearly drawn—'shall I spend the time to construct a fast lexical analyser by hand, or shall I sacrifice speed and size for the convenience of using a lexical analyser generator?'\(^3\)

Certainly there is no lack of lexical analyser generators. The RWORD system,\(^4\) LAWS\(^5\) and LEX\(^6\) are lexical analyser generating systems that accept regular expressions as input, and build lexical analysers consisting of tables and procedures. DeRemer\(^7\) describes a lexical analyser generator based on a modified LR technique. Rohlfing\(^7\) also discusses a similar lexical analyser generator. The lexical analyser generators described above are 'general purpose' in that they are designed to accept arbitrary regular expressions and actions, and to produce arbitrary actions to be taken upon expression recognition. These approaches generally represent the character-to-character transition

0038-0644/86/090801-08$05.00  
© 1986 by John Wiley & Sons, Ltd.  
Received 27 September 1985  
Revised 13 February 1986
function as a matrix, table[state, character], which is interpreted during lexical analysis:

    while basic symbol not complete do
      begin
        state := table [state, next_character];
        case state of
          {code for actions to be performed in each state}
        end
      end;

The overhead due to interpretation can be greater than a factor of five, when compared to a carefully hand-built, directly executable, lexical analyser scanning typical Pascal text.8

The purpose of the present research was to construct tools that build fast, directly executable lexical analysers from specifications of the basic symbols of the language to be recognized. The directly executable nature of the analyser imposes some restrictions on the basic symbol set that can be recognized, but the restrictions encompass the basic symbols found in most current languages.

The 'bottom up' approach taken to building the lexical analyser generators was to start with fast, hand-built lexical analysers, which were specifically designed as the front ends for modern language compilers. These lexical analysers were decomposed by partitioning the basic symbols that were recognized into classes based on how those symbols were processed by the recognizers. The behaviour of the analysers in recognizing instances of each symbol class was then generalized, and abstracted into the lexical analyser generators described below.

Lexical analyser generators were implemented in Standard Pascal for portability, and also in C. These implementations are compatible with semantic analysers generated using the GAG system5 and parsers generated using the PGS LALR(1) parser generation system.10

STRUCTURE OF A FAST LEXICAL ANALYSER

The basic symbols of the language being processed are represented internally by tokens that the parser obtains by calls to the routine BasicSymbol, which forms the heart of the lexical analyser module. Under this schema, tokens are classified as delimiters, identifiers and denotations.

Delimiters represent operators or program structuring symbols, and carry no information beyond their terminal symbol number (token number). Identifiers are freely chosen by the programmer, and the compiler must be able to determine the unique identity of each identifier. Denotations represent constants in the universe of values of the programming language, and the compiler must likewise be able to determine the value represented. Each token also contains information about its co-ordinates (line number and column number) in the program text for error reporting. Therefore, each token must contain the terminal code corresponding to it in the context-free grammar for the source language, and its position in the source text. Tokens representing identifiers and denotations must also specify which unique identifier or denotation they represent. These specifiers reflect intrinsic attributes of the token in the attributed structure tree.
The lexical analyser uses the services of an identifier table module, which embodies the set of distinct identifiers used in the program, a constant table module, which represents the constant values used in the program (strings, integers and floating point numbers, for example), and an error-reporting module. The services of these three modules are available to the rest of the compiler as well. Details of the interfaces to these modules can be found in Reference 8.

Figure 1 describes the interface for the lexical analysis module. TERMNODE is a variant record representing the token. The field, position, is a record with two fields that represent the co-ordinates of the token. The variant fields specify the intrinsic attributes of the token, representing objects stored in the symbol or constant table module.

```
const
(* codes for terminal symbols *)
NULL=8;
EOP=1;
IDNT=2;
INTG=3;
FLTPI=4;
STRG=5;
LSGR=6
LSEQ=7
...
LSTN=15
.
.
type

TERMINALS=0...maxterminalcode;
CODETERMS=IDNT..STRG;

TERMNODE=record
  pos: position;
syntaxcode: terminalsympol;
case codeterms of
  2: (idntfield: symbol);
  3: (intgfield: string);
  .
end;
.

procedure BasicSymbol (var t: termnode);
(* obtain the next basic symbol from the source file."
  On exit: t describes the basic symbol *)

Figure 1. Lexical analyser interface
```

The fastest analysers use the design principles of 'touching' each character as few times as possible, and avoiding procedure calls and indirect references whenever possible. The analyser described herein adheres to these principles.

Figure 2 shows the skeleton of the procedure BasicSymbol, which recognizes basic
repeat
  while current character = ' ' do advance one character;
  set position field of token;
  set the syntaxcode field of token to chtbl[current character];
  advance one character
  case syntaxcode field of
    .
    IDNT:
    begin
      while idn in scantab[current character] do advance one character;
      symboltable(start pos'n, length, syntaxcode field, symbol ID field);
    end;
    INTG:
    begin
      while int in scantab[current character] do advance one character;
      constanttable(start pos'n, length, syntaxcode field, integer ID field);
    end;
    .
    LSTN:
    begin
      if current character = '>' then
        begin syntaxcode field := isgr; advance one character end
      else if current character = '=' then
        begin syntaxcode field := iseq; advance one character end;
    end;
    .
    one additional case element for each two- or three- character symbol;
    .
    null case elements for each single character symbol;
  end; {case}
until terminal field of token <> NULL;

Figure 2. Lexical analyser algorithm

symbols. The case statement selects a sub-automaton on the basis of the class of the first character as defined in chtbl[char].

These classes are characters that begin identifiers, characters that begin integers, characters that begin floating point numbers, character(s) that begin strings, character(s) that begin comments, characters that begin three or two-character non-alphanumeric symbols, and single character symbols.

Once in subautomata of the first four classes, further automaton action is controlled by scantbl, which is an array of flags, indexed by characters:

```plaintext
type flags = (idn, int, comment, strg);
var scantbl: array [char] of set of flags;
```

Thus, scantbl represents the set of characters that can continue each of the first four classes of symbols. (The construct idn in scantbl[char] leads to more efficient code than char in idset.) Symbols in the fifth class are recognized by specific tests on the second or second and third characters. Symbols in the sixth class require no additional action.

The analyser assumes no lexical distinction between identifiers and keywords: both
are processed by the identifier table, which is pre-loaded with the keywords, each of which is given its proper code. When a keyword is recognized, the identifier table resets its syntax code to the terminal code of the keyword recognized.

In a similar fashion, once the character string representing a valid integer, floating point number or string has been extracted, a call to the proper routine in the constant table module stores the denotation in some internal representation, and fills in the appropriate variant of the terminal symbol.

PRESENT WORK

The objective of the present work was to generate these ‘fastest’ analysers automatically from a restricted set of basic symbols and from specifications defining valid identifiers, denotations and comments.

The analysers, written in Pascal or C, were partitioned into their variant and invariant parts. The variant part of an analyser is the part that changes with a change of token set, identifier, denotation or comment specification. The major part of this research was concerned with designing procedures that would scan a set of basic symbols for a language, read high-level specifications describing valid identifiers, denotations and comments, and reconstruct the analyser to recognize the basic symbols specified.

In order of their appearance in the completed analyser, the variant parts are the constants describing the codes for terminal symbols, the type definitions defining the codes for the continuation flags for comments and symbols with intrinsic attributes, the variant parts of the record TERMNODE, which define the storage of the intrinsic attributes, the table chtbl[Char], which specifies a terminal code for each character, and thus specifies the proper case selector, and the table scantbl[Char], which specifies continuation characters for valid identifiers, denotations and comments, and the case statement, which actually embodies the algorithm for recognizing basic symbols.

The specifications describing the language to be recognized were partitioned into two sets: the basic symbol specification defines the set of basic symbols of the language, treating identifiers and denotations as ‘generic’—i.e. it describes them only through their terminal symbol codes. The second set of specifications permits the language designer to describe valid identifiers, and to specify the characters that delimit comments and strings. For the Pascal analyser, special ‘stop symbols’ were inserted into the invariant skeleton of the analyser at every place where invariant information would need to be filled in. The process of building the complete analyser was one of alternately copying from the skeleton up to a given stop symbol, and building and filling in the variant part as constructed from the specifications. For the C analyser, the include facility was used.

The constants describing the codes for terminal symbols can be read directly from the set of basic symbols, since they are defined by the compiler writer. However, since some of these constants are used as case selectors, the generator rearranges the code assignments so that those constants that will be used as case selectors form a compact set. This makes it more likely that the case statement will be compiled as a jump table rather than as a search tree.

Type definitions defining continuation codes for symbols with intrinsic attributes and for comments, are made by scanning the generic specification, assigning a unique continuation code for each generic specified. The present implementation limits the generics that may be specified to identifier, integer, floating point number, string,
character and comment. Chtbl is build by examining the first character of each basic symbol requiring special processing, and from the generic specifications. Scantbl, the table of continuation flags, is build from the generic specifications. The case statement is constructed by repeatedly passing over both specification files, partitioning all symbols to be recognized into one of the six classes, followed by construction of each case selector in order of terminal symbol code.

AN EXAMPLE: INTEGRATION INTO A COMPILER CONSTRUCTION ENVIRONMENT

The following example shows how the lexical analyser generator can be integrated into a compiler construction environment. The example environment consisted of GAG, a semantic analyser generator, PGS, a parser generator, and CGSS, a code selector generator. In practice, GAG processes an attribute grammar for the desired language, written in ALADIN, yielding a semantic analyser, and files suitable for input to PGS, the parser generator. Among these files is PGSSX.G, which contains the set of basic symbols and terminal symbol codes for the language, and SCANTAB.C, which contains the initial contents of the symbol table. GAG also emits a file of definitions, DEFOUT.PRO, which contains, among other things, the definition of the record TERMNODE, which embodies the definition of a token.

Thus, most of the information needed for generation of the lexical analyser is already available after GAG has processed the attribute grammar and extracted the context-free part. The only additional information needed by the generator is the identities of the symbols corresponding to tokens with intrinsic attributes, the specification of valid identifiers, denotations and comment and string delimiters. The compiler writer specifies these details in a small control file. For identifiers one specifies the set of initial characters, continuation characters and embedded characters. For denotations one specifies the initial and continuation characters, and in the case of reals, embedded characters. One also specifies the string and character delimiters, comment types and delimiters, and whether comments are terminated by termination characters or by the end of line marker. A typical Pascal specification would run as follows:

```
IDENTIFIER name [a-zA-Z] [a-zA-Z0-9] [ -]
INTEGER integer_number [0-9] [0-9]
REAL real_number [0-9] [0-9] [eE]
CHARACTER character [''] ['']
STRING string [''] ['']
BEGINCOMMENT 1 { (*
ENDCOMMENT 1 ) *}
```

The analyser generator rearranges the code assignments in files PGSSX.G and SCANTAB.C, as discussed above, and then generates the analyser module as described. This module is combined with an identifier table module, a constant table module, and an error handling module, using either the PROPP Pascal preprocessor or the C include facility.

The output of the lexical analyser generator is a complete lexical analyser in standard Pascal or C, a rearranged table of basic symbols and terminal symbol codes, and the initial contents of the identifier table.
UTILITY OF THE GENERATED ANALYSER

The generators make a number of assumptions about the language being specified: the set of characters that begin identifiers, denotations and delimiters must be disjoint; keywords are reserved words (i.e. there is no lexical distinction between keywords and identifiers); starting and ending string delimiters must be the same single character; lexical elements may not continue across program text lines; and language elements must be determinable by the principle of longest match—i.e. the automaton always proceeds until it encounters an invalid continuation character, at which time it returns to 'state zero' without backtracking. These restrictions do not pose serious problems in analysing most modern languages.

In any event, considerable care was taken to make the generated Pascal or C analysers understandable to the compiler writer, so that minor modifications can be made by hand.

There are several advantages to using this system compared to hand coding. The user spends just minutes to generate a highly optimized analyser instead of several days hand coding and optimizing one. (A typical generated analyser and tables may be nearly 1000 lines long.) When designing or modifying a language there are even greater advantages, since each change to the terminal symbols of the grammar necessitates recoding a portion of the analyser.

EXPERIENCE

The system has been used to successfully generate analysers for Pascal, LAX,¹ an experimental teaching language, and several other model languages designed to test the capabilities of the system. The generated analysers behaved like hand build analysers in all respects. They were faster than LEX⁵ by factors ranging from 5·7 to 6·5 when scanning typical Pascal text.

ACKNOWLEDGEMENT

The author thanks Professor W. M. Waite for several helpful discussions and suggestions concerning this work.

REFERENCES


