The INDIA Lexic Generator

MICHAEL ALBINUS WERNER AßMANN*

1 Introduction

Because lexical analysis takes a conciderable amount of compilation time it is necessary to build fast scanners. Generated scanners were brought into discredit because their lack of efficiency, although finite automata are an appreciated method for generating scanners. Some effort was made to improve the speed of generated scanners.

This paper describes the lexic generator of the INDIA system.

2 Lexical analysis in the INDIA system

The INDIA compiler generator, described in [Albinus86], is the basis for compiler construction in the INDIA system. It generates tables for all compiler components, which contain the language specific informations. As presented in [Aßmann86], the compiler can be viewed as a set of abstract machines associated by tables which contain the abstract instruction codes to control the work. In this way we have a *lexical machine* (scanner) that reads a sequence of input characters and fits them into a sequence of lexical items, the smallest symbols known by the *syntactical machine* (parser). The *syntactical machine* (based on LR(1) respectively LALR(1) mechanism) transforms this sequence of lexical items into a sequence of meta symbols again, and so on. Therefore, we have the following model:

*Academy of Sciences of the G.D.R.
Institute of Informatics and Computing Technique
Rudower Chaussee 5, Berlin, G.D.R.

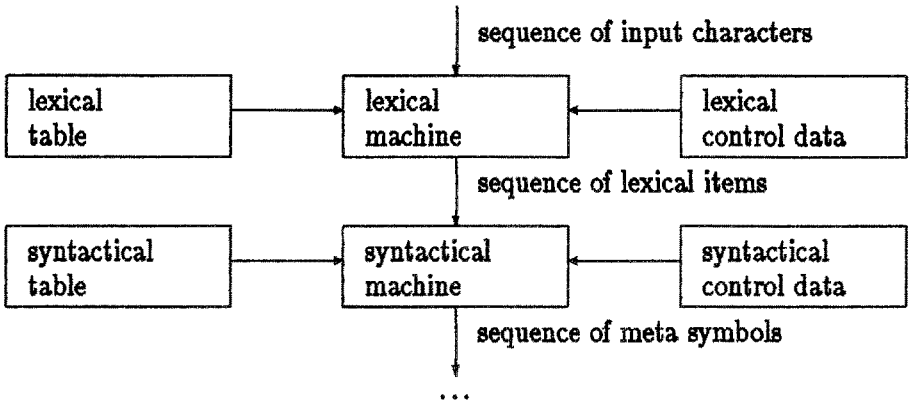

Figure 1: Principle of the lexical and syntactical machines

The other abstract machines of the compiler (tree constructor, table constructor, reparser, and code generator) are working in the same way. Every abstract machine is controlled by control data including options for list regime, production of test results and so on.

3 Generation of lexical table

The scanner has to construct the lexical items. There exists two kinds of lexical items, namely lexical items without "semantic" like '<', ':=' or 'BEGIN', and lexical items with a determined value like identifiers or numbers. We call them *terminal symbols* and *pseudoterminal symbols*, respectively. Keywords (like 'BEGIN') are ordinary *terminal symbols* from the viewpoint of syntactical analysis. Comments are special *pseudoterminal symbols*. Every lexical item is represented by an item number.

For generating the lexical table it is necessary to describe

- the text (string) of every *terminal symbol* and the item number belonging to,
- the syntactical structure of every *pseudoterminal symbol* and the item number belonging to,
- special features for handling of keywords, and
- special features for handling of comments.

The lexic generator transforms the description of lexical items into an deterministic finite automaton as described in [Aho77] and stores an abstract program, representing the automaton, into the lexical table. Some special features introduced in the next subsections are represented in the automaton.

3.1 Defining lexical items

The *terminal* and *pseudoterminal symbols* are defined during the generation of the syntactical table. Using extended BNF notation (described in [Aßmann85]), the syntax production

```
<Procedure Head>      ::= 'PROCEDURE' "Identifier" <Parameters> ';'  ||
```

defines the *terminal symbols* `'PROCEDURE'` and `';'` as well as the *pseudoterminal symbol* `"Identifier"`. The item numbers belonging to are generated automatically. In a special part of the compiler generator it is possible to declare the item numbers explicitly. Nevertheless, in the most cases it is unnecessary.

Keywords are recognized by the property of being terminal symbols containing letters only. All these facts are available for the generation of the lexical table.

3.2 Syntactical structure of pseudoterminal symbols

The syntax of *pseudoterminal symbols* is described by using productions of regular grammars (instead of regular expressions as proposed in [Aho77]). The possibilities of description are derived from Alexis ([Mössenbeck86]) using the INDIA notation.

A typical production is

```
"Real_Number"         ::= [<Decimal Digit>]+ '.' [<Decimal Digit>]*
                          ['E' [('+'|'-')] [<Decimal Digit>]+ ]      ||
```

It defines the syntactical nature of real numbers (in MODULA-2). Elements of a production are simple character literals (like `'E'`) or previously declared character sets (`<Decimal Digit>`). Character sets are introduced in section 3.5. A non printable character literal can be written in its octal notation. For example, the character literal `36C` stands for the EOL character.

Expressions are built using

- alternatives: `('+'|'-')`
- options: `[('+'|'-')]`
  (repeat factor 0 or 1)
- optional iterations: `[<Decimal Digit>]*`
  (repeat factor 0, 1 or any more)
- iterations: `[<Decimal Digit>]+`
  (repeat factor 1 or any more)

It is possible to mark "redundant" character literals. These characters will be removed by the scanner from the string containing the *pseudoterminal symbol*. $\{\alpha\}$ describes removing a character.

3.3 Comments

Comments are treated as a special *pseudoterminal symbol*. The description contains the leading and ending characters of an comment. It is possible to describe the nested structure of comments with the keyword NESTED. This breaks the regularity of the expressions and will be handled in a special way.

The next productions describe nested MODULA-2 comments and unnested Ada comments.

```
"Comment"        ::= {'(*'} [ANY FOLLOWED BY '*)']* {'*)'} NESTED ||

"Comment"        ::= {'--'} [ANY FOLLOWED BY 36C]*  {36C}        ||
```

ANY stands for the character set containing all characters. FOLLOWED BY α is a construction that terminates the optional iteration [ANY]*, which never ends otherwise. It can be used in other lexical declarations too for avoiding ambiguities, but requires multiple access to characters and decreases the efficiency of the scanner.

3.4 Keywords

Keywords are sampled from the set of *terminal symbols* defined during the generation of the syntactical table. Using the phrase EXCEPT KEYWORDS in the identifier production, keywords and identifiers are distinguished.

```
"Identifier"     ::= <letter> [<extended letter>]* EXCEPT KEYWORDS ||
```

The lexic generator produces a perfect hash function h over the keywords. It uses the (heuristic) approach from [Sager85]. This function is defined as

$$h : K \rightarrow [0 \quad .. \quad N-1]$$

whereby K stands for the set of keywords and N is a cardinal number with $N \geq \|K\|$. h is called a perfect hash function if h proves as an injection (h is unique).

Perfect hash functions have the advantage that the decision for being a keyword (or not) is very fast, because after computing the hash code of an identifier it needs only one comparison with the keyword represented by this hash code.

A perfect hash function h is called minimal perfect hash function if $N = \|K\|$.

The lexic generator realizes the hash function h as

$$h(k) = (h_0(k) + h_1(k) + h_2(k)) \quad \text{MOD} \quad N$$

$$h_0(k) = \text{ORD}(k[i_{01}]) + \text{ORD}(k[i_{02}])$$

$$h_1(k) = g_1[(\text{ORD}(k[i_{11}]) + \text{ORD}(k[i_{12}])) \quad \text{MOD} \quad r]$$

$$h_2(k) = g_2[(\text{ORD}(k[i_{21}]) + \text{ORD}(k[i_{22}])) \quad \text{MOD} \quad r]$$

whereas

- $k \in K$ is a keyword interpreted as a string (ARRAY OF CHAR)
- r is the smallest power of 2 with $r > \frac{\|K\|}{3}$;
- i_{xy} are array indices describing access to keyword characters;
- ORD is a function converting a character into its binary representation; and
- g_1, g_2 are arrays for parametrizing the hash function.

For details, see [Ernst87].

All these N, r, i_{xy}, g_1, g_2 are computed by the lexic generator and stored into the lexical table as parameters for the hash function used during lexical analysis. Appendix B contains the values computed for the keyword set of MODULA–2. By the way, all hash functions computed with this algorithm by the authors were minimal perfect hash functions. This holds for the compiler generator INDIA itself (22 keywords), MODULA–2 (40 keywords), PALM (our MODULA–2 extension, 51 keywords) and CHILL (86 keywords).

3.5 Character sets

Character sets are used in the productions of the lexic generator to allow the choice of one character from a set. It is in principle a simplified notation for a choice only. For example,

```
<octal digit>      ::= '01234567'                          ||
"Octal_Number"     ::= [<octal digit>]+ 'B'                ||
```

is equivalent to

```
"Octal_Number"     ::= [('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7')]+ 'B'    ||
```

Definition of new character sets can use unions or differences of strings or already defined character sets, respectively.

```
<hexadecimal digit> ::= <decimal digit> + 'ABCDEFabcdef'   ||
```

3.6 The generated automaton and its abstract program

The lexic generator samples all *terminal* and *pseudoterminal symbols* delivered by the syntax generator and builds an deterministic automaton from it. *Terminal symbols* (except keywords) are included into the automaton as chain, *pseudoterminal symbols* as partial automaton, derived from the affiliated production. Accepting any lexical item is done by using the longest chain. This technique is well known and described in [Aho77], for example.

The partial automaton for scanning the '<' symbol would be

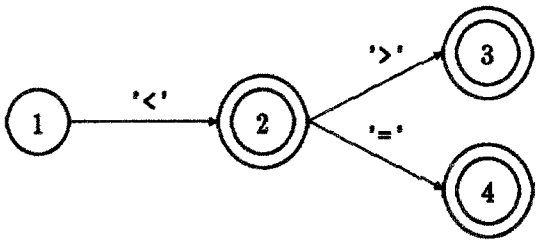

Figure 2: Abstract automaton

The lexic generator produces an abstract program using basic operations from this automaton. This abstract program is stored into the lexical table and interpreted by the scanner during the lexical analysis. The set of basic operations is described in appendix C. The resulting abstract program for state 2 above is

```
                    (* scan '<' *)
lex_accept
                    (* scan '>' and return item number of '<>' *)
lex_accept_and_return_if_next '>', 85
                    (* scan '=' and return item number of '<=' *)
lex_accept_and_return_if_next '=', 86
                    (* return item number of '<' *)
lex_return_code 72
```

4 Remarks on efficiency

4.1 Arrangements for increasing efficiency

- The approach of converting the deterministic finite automaton into program text of the scanner, favoured in the most lexic generators ([Horspool87], [Eulenstein88], [Grosch88], [Heuring86], [Mössenbeck86]), was not suitable for us, because the INDIA system is multilingual. Currently, it supports PALM (our MODULA-2 extension, see [Baum88]) and CHILL. Therefore, the scanner has to interprete the lexical table very efficiently. It tries to avoid multiple access to a character if possible.

  The main loop interpreting the lexical operations is a closed program part without any procedure calls (except keyword handling). The data structures inside the lexical table are optimized for this task and allow fast access to all parts of the lexical table.

- The length of a *terminal symbol* (except keywords) is restricted up to 2 characters. It results simple automata with short chains. *Terminal symbols* with the same start

character decreases efficiency. For example, scanning '<' needs in MODULA-2 four operations instead of the simple `lex_accept_and_return` operation.

- Using character sets for transitions between states simplifies the automaton and allows shorter and faster operation sequences interpreting the automaton.
- The identifier automaton is optimized depending on the keywords:

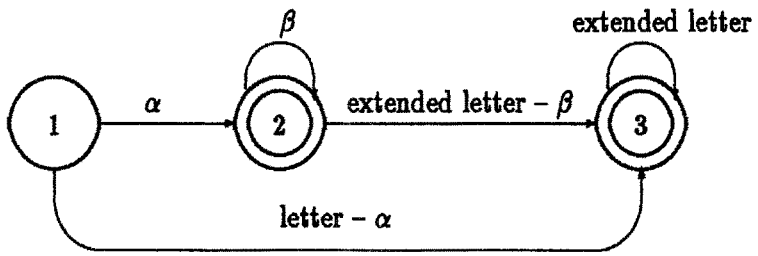

Figure 3: Abstract automaton for identifiers

α is the character set containing all start characters of the keywords. The character set β contains all characters occuring in keywords at any position but not the first. The check for being a keyword occurs only in state 2. Achieving state 3 an identifier cannot be a keyword, and the check would be absurd (and time consuming).

Therefore, if an identifier contains at least one character not included in the character set α or β, the check for being a keyword doesn't appear. In many programming languages it holds for all identifiers containing at least a small letter or a digit.

The abstract program for this automaton in respect of MODULA-2 keywords is

```
state 1:
 lex_goto_state_if_next_in_set
     ABCDEFILMNOPQRSTUVW, 2
 lex_goto_state_if_next_in_set
     GHJKXYZabcdefghijklmnopqrstuvwxyz, 3

state 2:
 lex_accept
 lex_accept_while_in_set
     ACDEFGHILMNOPRSTUVXY
 lex_goto_state_if_next_in_set
     0123456789BJKQWZabcdefghijklmnopqrstuvwxyz, 3
 lex_return_code_if_keyword
 lex_return_code 1
```

```
state 3:
 lex_accept
 lex_accept_while_in_set
     0123456789
     ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
 lex_return_code 1
```

4.2 Results (for MODULA–2 lexic)[1]

```
lexic table: 2 KByte
  finite automaton: 54 states, 90 transitions
  abstract program: 159 operations

MODULA-2 mix:  2*25 modules
  1 116 525 characters            28 921 lines
    362 894 blanks/EOL            302 854 characters in comments

     99 456 lexical items          4 662 comments (4.7%)
     10 131 keywords (10.2%)      33 794 identifiers (33.98%)

runtime: 230 sec
      7 545 lines/min              4 854 characters/sec
    447 393 operations
      4 662 characters handled more than once (0.42%)
      5 308 identifiers were asked to be a keyword (15.71%)
```

[1] time measured on an 8 MHz IBM/AT

A Lexic definition part for MODULA–2

```
                                        (* Character set definitions *)
                                        (* ~~~~~~~~~~~~~~~~~~~~~~~~~ *)
<octal digit>       ::= '01234567'                                    ||
<decimal digit>     ::= <octal digit> + '89'                          ||
<hexadecimal digit> ::= <decimal digit> + 'ABCDEFabcdef'              ||
<letter>            ::= 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' +
                        'abcdefghijklmnopqrstuvwxyz'                  ||
<extended letter>   ::= <letter> + <decimal digit>                    ||
<string1 character> ::= ANY - 36C - '"'                               ||
<string2 character> ::= ANY - 36C - 47C                               ||
                                    (* Pseudoterminal declarations *)
                                    (* ~~~~~~~~~~~~~~~~~~~~~~~~~~~ *)
"IDENTIFIER"        ::= <letter> [<extended letter>]* EXCEPT KEYWORDS ||
"OCTAL_NUMBER"      ::= [<octal digit>]+ 'B'                          ||
"CARD_NUMBER"       ::= [<decimal digit>]+ [FOLLOWED BY '..']         ||
"HEX_NUMBER"        ::= <decimal digit> [<hexadecimal digit>]* 'H'    ||
"REAL_NUMBER"       ::= [<decimal digit>]+ '.' [<decimal digit>]*
                        ['E' [('+'|'-')] [<decimal digit>]+ ]         ||
"CHARACTER"         ::= {'"'} <string1 character> {'"'}               |
                        {47C} <string2 character> {47C}               |
                        [<octal digit>]+ 'C'                          ||
"STRING"            ::= {'"'} [<string1 character>
                        [<string1 character>]+ ] {'"'}                |
                        {47C} [<string2 character>
                        [<string2 character>]+ ] {47C}                ||
"COMMENT"           ::= {'(*'} [ANY FOLLOWED BY '*)']* {'*)'} NESTED  ||
```

B Minimal perfect hash function for MODULA–2

The hash value must be computed by the following formula:

```
h := ( ORD(c1) + ORD(c0 )

       + g1[ (ORD(c3) + ORD(c )) MOD  16 ]

       + g2[ (ORD(c2) + ORD(c )) MOD  16 ] ) MOD   40
```

Note: c0 means length of terminal
c means blank character (Code = 32)
(can be eliminated by redefinition of g)

*** RESULTS ***

h	keyword	h	keyword
0	MOD	20	LOOP
1	NOT	21	FOR
2	FROM	22	VAR
3	MODULE	23	RECORD
4	OF	24	THEN
5	DIV	25	CASE
6	ARRAY	26	QUALIFIED
7	TO	27	IMPLEMENTATION
8	POINTER	28	AND
9	TYPE	29	BY
10	UNTIL	30	OR
11	WITH	31	DO
12	SET	32	END
13	BEGIN	33	ELSE
14	RETURN	34	ELSIF
15	REPEAT	35	CONST
16	WHILE	36	IN
17	PROCEDURE	37	EXIT
18	DEFINITION	38	IF
19	IMPORT	39	EXPORT

Nr	g1	g2
0	0	0
1	9	26
2	27	29
3	8	0
4	39	0
5	12	7
6	13	3
7	15	0
8	0	4
9	0	1
10	0	0
11	0	0
12	0	32
13	0	20
14	2	1
15	19	1

C Basic operations for a deterministic finite automaton interpreter

```
LexEndOfLine                (*                                  *)
LexEndOfFile                (*                                  *)
LexOverreadBlanks           (*                                  *)
LexOverreadUntil            (* <character>                       *)
LexOverreadUntils           (* <character>,   <character>       *)
LexOverreadIfCondUntil      (* <character>                       *)
LexOverreadIfCondUntils     (* <character>,   <character>       *)
LexReadComment              (* <character>,   <character>,      *)
                            (* <character>,   <character>,      *)
                            (* <nested>                         *)
LexAccept                   (*                                  *)
LexReturnCode               (* <lexical code>                    *)
LexAcceptAndReturn          (* <lexical code>                    *)
LexAcceptAndReturnIfNext    (* <character>,   <lexical code>    *)
LexReturnCodeIfNextNot      (* <character>,   <lexical code>    *)
LexAcceptWhileInSet         (* <class>                          *)
LexAcceptWhileNotInSet      (* <class>                          *)
LexReturnCodeIfKeyword      (*                                  *)
LexInsertCharacter          (* <character>                      *)
LexSkipCharacter            (*                                  *)
LexReturnCharacter          (*                                  *)
LexGotoInitialState         (*                                  *)
LexGotoStateIfNext          (* <character>,   <address>         *)
LexGotoStateIfNextNot       (* <character>,   <address>         *)
LexGotoStateIfNextInSet     (* <class>,       <address>         *)
LexGotoStateIfNextNotInSet  (* <class>,       <address>         *)
LexGotoState                (* <address>                        *)
LexError                    (* <error code>                     *)
```

References

[Aho77] Aho, A.V.; Ullman, J.D.:
Principles of Compiler Design
Addison Wesley 1977

[Albinus86] Albinus, M.; Aßmann, W.; Enskonatus, P.:
The INDIA Compiler Generator
Workshop on Compiler Compiler and Incremental Compilation,
iir-reporte 2(1986)12, Berlin 1986, pp. 58–84

[Aßmann85] Aßmann, W.:
Die Metasprache des Compiler-Rahmensystems INDIA
iir-reporte 1(1985)7, Berlin 1985, pp. 8–13

[Aßmann86] Aßmann, W.:
The INDIA System for Incremental Dialog Programming
Workshop on Compiler Compiler and Incremental Compilation,
iir-reporte 2(1986)12, Berlin 1986, pp. 12–34

[Baum88] Baum, M.:
PALM
internal material, IIR, Berlin 1988

[Ernst87] Ernst, Th.:
Eine Implementation des Minimalzyklen-Algorithmus zum Bestimmen perfekter Hashfunktionen
internal material, IIR, Berlin 1987

[Eulenstein88] Eulenstein, M.:
The POCO Compiler Generating System
Workshop on Compiler Compiler and High Speed Compilation, Berlin 1988

[Grosch88] Grosch, J.:
Generators for High-Speed Front-Ends
Workshop on Compiler Compiler and High Speed Compilation, Berlin 1988

[Heuring86] Heuring, V.P.:
The Automatic Generation of Fast Lexical Analysers
Software — Practice and Experience 16(1986)9, pp. 801–808

[Horspool87] Horspool, R.N.; Levy, M.R.:
Mkscan — An Interactive Scanner Generator
Software — Practice and Experience 17(1987)6, pp. 369–378

[Mössenbeck86] Mössenbeck, H.:
Alex — A Simple and Efficient Scanner Generator
SIGPLAN Notices 21(1986)5, pp. 69–78

[Sager85] Sager, T.J.:
Polynomial Time Generator for Minimal Perfect Hash Functions
Commun. ACM 28(1985)5, pp. 523–532

[Szwillus86] Szwillus, Gerd.; Hemmer, W.:
Die automatische Erzeugung effizienter Scanner
University of Dortmund, Report Nr. 217, 1986

[Waite86] Waite, W. M.:
The Cost of Lexical Analysis
Software — Practice and Experience 16(1986)5, pp. 473–488