http://pl.skuniv.ac.kr

컴파일러 입문

제 4 장

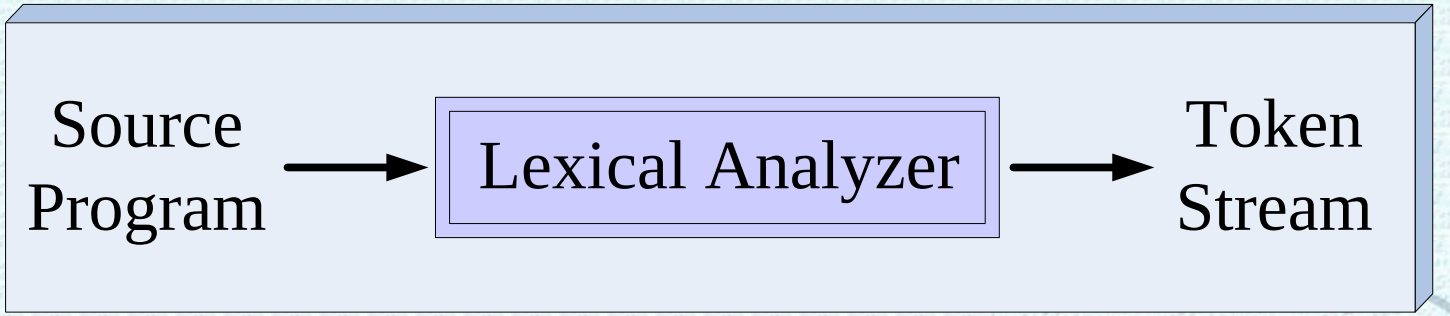
어휘 분석

목 차

4.1 서 론

Text p.130

- **Lexical Analysis**
  - the process by which the compiler **groups** certain strings of characters **into** individual tokens.



- **Lexical Analyzer ⏩ Scanner ⏩ Lexer**

- Token
  - 문법적으로 의미 있는 최소 단위

Token - a single syntactic entity(terminal symbol).
Token Number - string 처리의 효율성 위한 integer number.
Token Value - numeric value or string value.

ex)

	if	(	a	>	10	)	...
Token Number :	32	7	4	25	5	8	
Token Value :	0	0	'a'	0	10	0	

- **Token classes**
  - **Special form** - ***language designer***

    1. **Keyword** --- const, else, if, int, ...
    2. **Operator symbols** --- +, -, *, /, ++, -- etc.
    3. **Delimiters** --- ;, ,, (,), [,] etc.

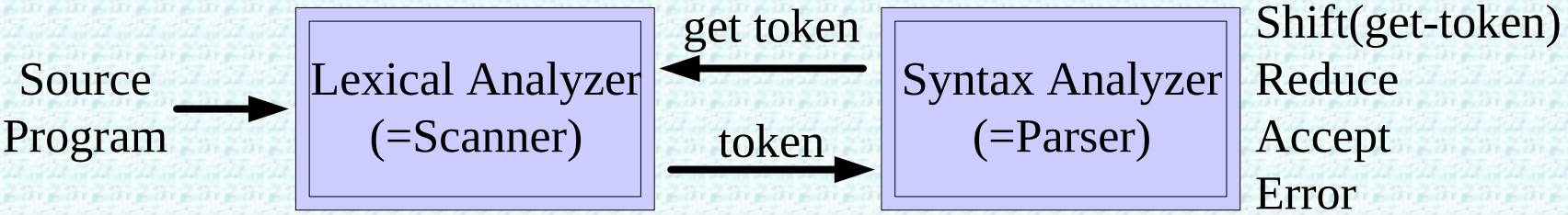
  - **General form** - ***programmer***

    4. **identifier** --- stk, ptr, sum, ...
    5. **constant** --- 526, 3.0, 0.1234e-10, 'c', "string" etc.

- **Token Structure** - represented by **regular expression**.

  ex) id = $(l + \_)(l + d + \_)^*$

- Interaction of Lexical Analyzer with Parser
  - Lexical Analyzer is the procedure of Syntax Analyzer.
    - L.A. → *Finite Automata.*
    - S.A. → *Pushdown Automata.*



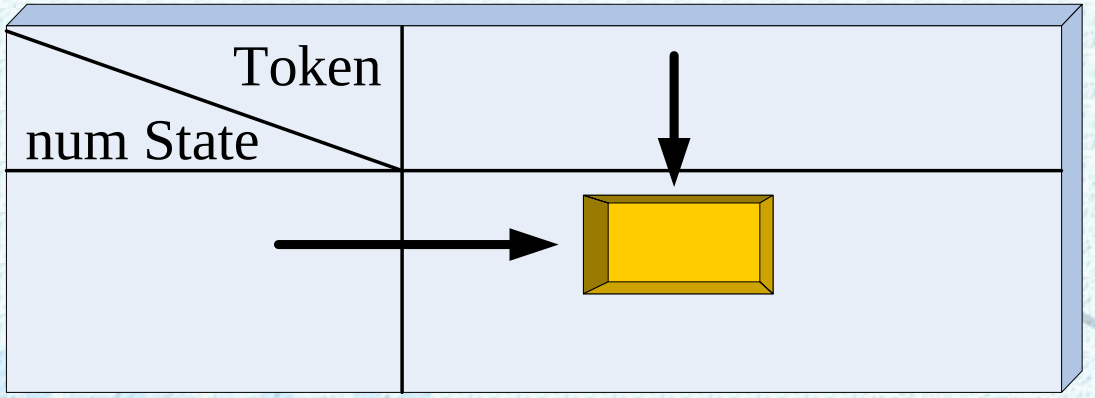
  - Token type
    - scanner가 parser에게 넘겨주는 토큰 형태.
      (token number, token value)

ex) if (x > y) x = 10 ;

(32,0) (7,0) (4,x) (25,0) (4,y) (8,0) (4,x) (23,0) (5,10) (20,0)

- The **reasons** for separating the analysis phase of compiling into lexical analysis(***scanning***) and syntax analysis(***parsing***).
  1. modular construction - simpler design.
  2. compiler efficiency is improved.
  3. compiler portability is enhanced.

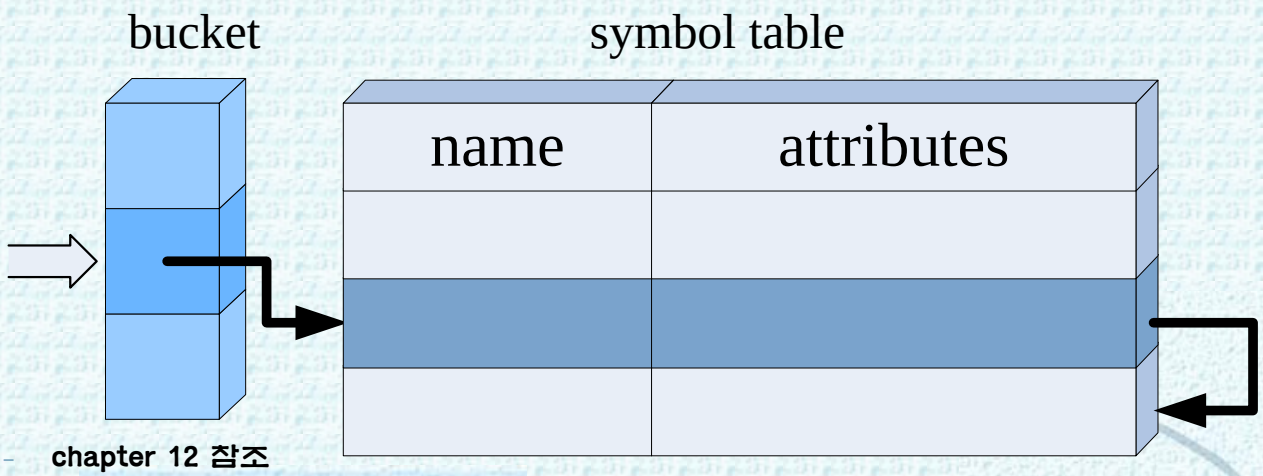
- **Parsing table**
  - Parser의 행동(*Shift, Reduce, Accept, Error*)을 결정.



  - Token number는 Parsing table의 index.

- **Symbol table의 용도**
  - L.A와 S.A시 identifier에 관한 정보를 수집하여 저장.
  - Semantic analysis와 Code generation시에 사용.
  - name + attributes

ex) Hashed symbol table



- chapter 12 참조

4.2 토큰 인식

Text p.134

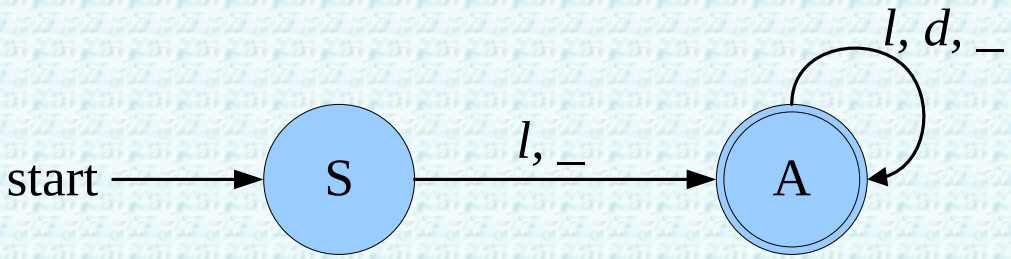
- Specification of token structure - RE
- Specification of PL - CFG

- **Scanner design steps**
  1. describe the structure of tokens in **re**.
  2. or, directly design a **transition diagram** for the tokens.
  3. and **program** a scanner according to the diagram.
  4. moreover, we **verify** the scanner action through *regular language theory*.

- **Character classification**
  - letter : a ¦ b ¦ c... ¦ z ¦ A ¦ B ¦ C ¦...¦ Z → *l*
  - digit : 0 ¦ 1 ¦ 2... ¦ 9 → *d*
  - special character : + ¦ - ¦ * ¦ / ¦ . ¦ , ¦ ...

4.2.1 Identifier Recognition

- Transition diagram



- Regular grammar

$S \rightarrow lA \mid \_A \qquad A \rightarrow lA \mid dA \mid \_A \mid \varepsilon$

- Regular expression

$S = lA + \_A = (l + \_)A$

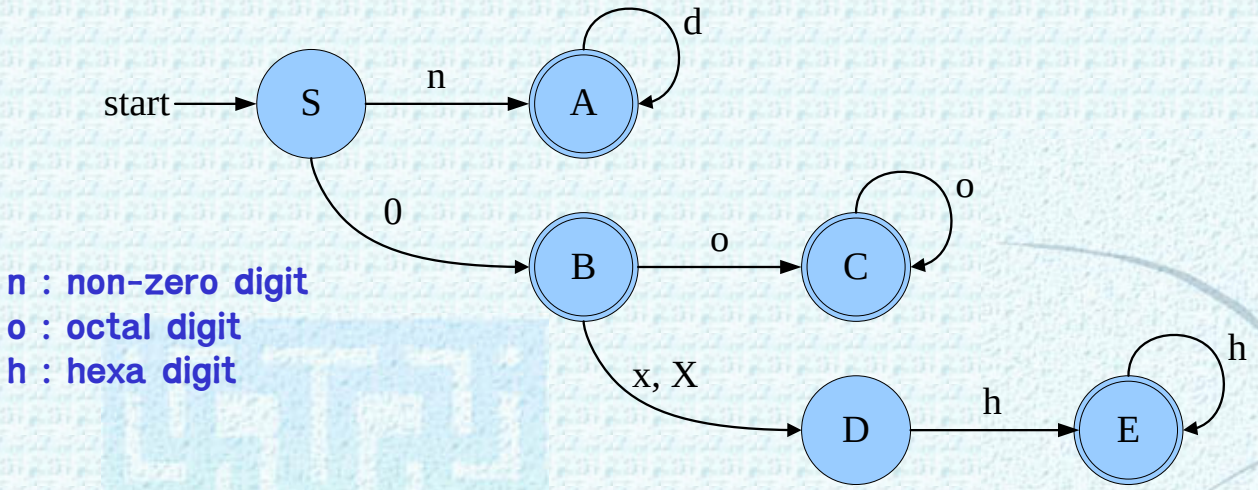
$A = lA + dA + \_A + \varepsilon = (l + d + \_)A + \varepsilon = (l + d + \_)^*$

$\therefore S = (l + \_)(l + d + \_)^*$

4.2.2 Integer number Recognition

- Form : 10진수, 8진수, 16진수로 구분되어진다.
  10진수 : 0이 아닌 수 시작
  8진수 : 0으로 시작, 16진수 : 0x, 0X로 시작

- Transition diagram



Regular grammar

$S \rightarrow nA \mid 0B \qquad A \rightarrow dA \mid \varepsilon \qquad B \rightarrow oC \mid xD \mid XD \mid \varepsilon$

$C \rightarrow oC \mid \varepsilon \qquad D \rightarrow hE \qquad E \rightarrow hE \mid \varepsilon$

Regular expression

$E = hE + \varepsilon = h^*\varepsilon = h^* \qquad D = hE = hh^* = h^+$

$C = oC + \varepsilon = o^*$

$B = oC + xD + XD + \varepsilon = o^+ + (x + X)D = o^+ + (x + X)h^+ + \varepsilon$

$A = dA + \varepsilon = d^*$

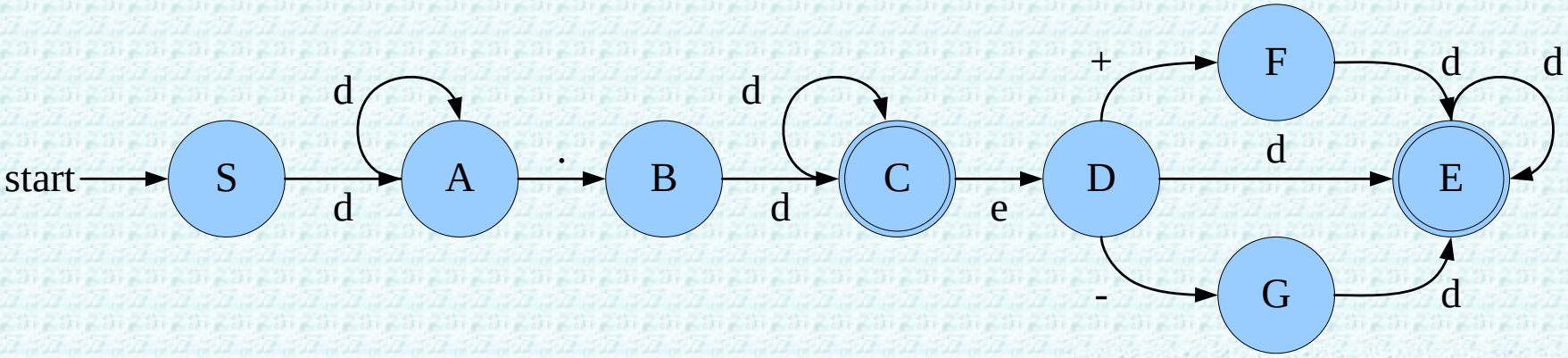
$S = nA + 0B = nd^* + 0(o^+ + (x + X)h^+ + \varepsilon)$

$= nd^* + 0 + 0o^+ + 0(x + X)h^+$

$\therefore \; S = \mathbf{nd^* + 0 + 0o^+ + 0(x + X)h^+}$

4.2.3 Real number Recognition

- Form : Fixed-point number & Floating-point number
- Transition diagram



- Regular grammar

$S \rightarrow dA$
$A \rightarrow dA \mid .B$
$B \rightarrow dC$
$C \rightarrow dC \mid eD \mid \varepsilon$
$D \rightarrow dE \mid +F \mid -G$
$E \rightarrow dE \mid \varepsilon$
$F \rightarrow dE$
$G \rightarrow dE$

Text p.138

Regular expression

$E = dE + \varepsilon = d^*$ $F = dE = dd^* = d^+$ $G = dE = dd^* = d^+$

$D = dE + '+'F + -G = dd^* + '+'d^+ + -d^+$

$= d^+ + '+'d^+ + -d^+ = (\varepsilon + '+' + -)d^+$

$C = dC + eD + \varepsilon = dC + e(\varepsilon + '+' + -)d^+ + e$

$= d^*(e(\varepsilon + '+' + -)\, d^+ + \varepsilon)$

$B = dC = dd^*(e(\varepsilon + '+' + -)d^+ + \varepsilon)$

$= d^+ + (e(\varepsilon + '+' + -)\, d^+ + \varepsilon)$

$A = dA + .B$

$= d^*.d + (e(\varepsilon + '+' + -)d^+ + \varepsilon)$

$S = dA$

$= dd^*.\, d^+(e(\varepsilon + '+' + -)\, d^+ + \varepsilon)$

$= d^+.d^+(e(\varepsilon + '+' + -)\, d^+ + \varepsilon)$

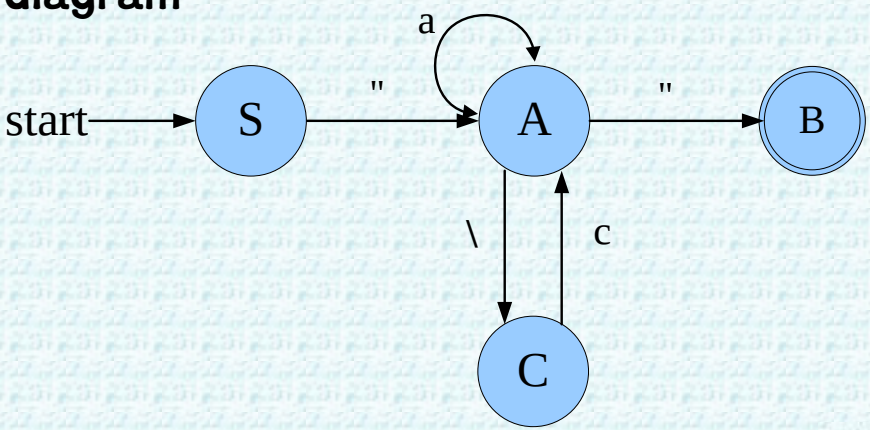
$= d^+.d^+ + d^+.d^+e(\varepsilon + '+' + -)\, d^+$

참고 Terminal +를 '+'로 표기.

4.2.4 String Constant Recognition

- Form : a sequence of characters between a pair of double quotes.
- Transition diagram

Text p.139



where, a = char_set - {", \} and c = char_set

- Regular grammar

$S \rightarrow "A$

$A \rightarrow aA \mid "B \mid \backslash C$

$B \rightarrow \varepsilon$

$C \rightarrow cA$

Regular expression

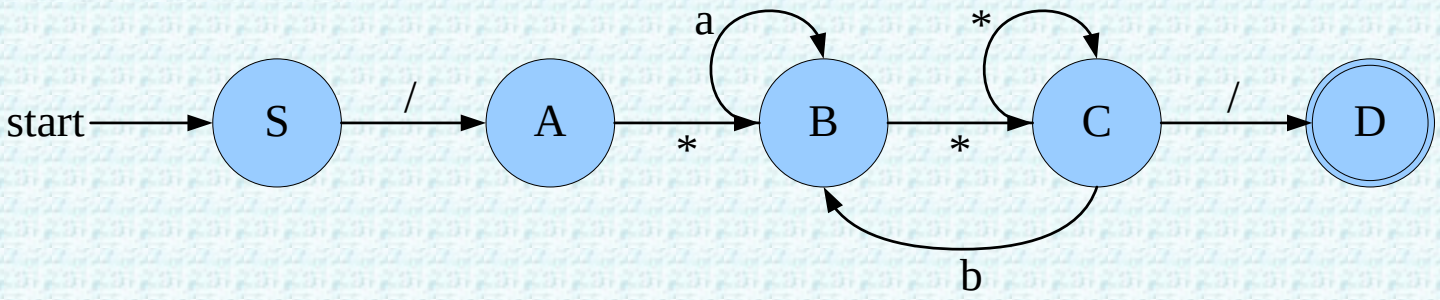
A = aA + " B + \C
= aA + " + \cA
= (a + \c)A + "
= (a + \c)* "

S = " A
= "(a + \c)*"

∴ S = **"(a + \c)* "**

4.2.5 Comment Recognition

- **Transition diagram**



where, **a** = char_set - {*} and **b** = char_set - {*, /}.

- **Regular grammar**

$S \rightarrow /A$

$A \rightarrow *B$

$B \rightarrow aB \mid *C$

$C \rightarrow *C \mid bB \mid /D$

$D \rightarrow \varepsilon$

- **Regular expression**

  C = *C + bB + /D = $*^{*}$(bB + /)

  B = aB + $**^{*}$(bB + /)

  = aB + $**^{*}$bB + $**^{*}$/

  = (a + $**^{*}$ b)B + $**^{*}$/= (a + $**^{*}$b)$^{*}**^{*}$/

  A = *B = *(a + $**^{*}$b)$^{*}**^{*}$/

  ∴ S = /A = **/* (a + $**^{*}$b)$^{*}**^{*}$/**

- **A program which recognizes a comment statement.**

```
do {
    while (ch != '*') ch = getchar();
    ch = getchar();
} while (ch != '/');
```

4.3 어휘분석기의 구현

Text p.142

- Design methods of a Lexical Analyzer
  - **Programming** the lexical analyzer using conventional programming language.
  - **Generating** the lexical analyzer using ***compiler generating tools*** such as **LEX**.
- Programming vs. Constructing

- The Tokens of Mini C (p.142, 문법: pp.619-622)
  - Special symbols (30개-연산자, 구분자)

!	!=	%	%=	&&
(	)	*	*=	+
++	+=	,	-	--
-=	/	/=	;	<
<=	=	==	>	>=
[	]	{	||	}

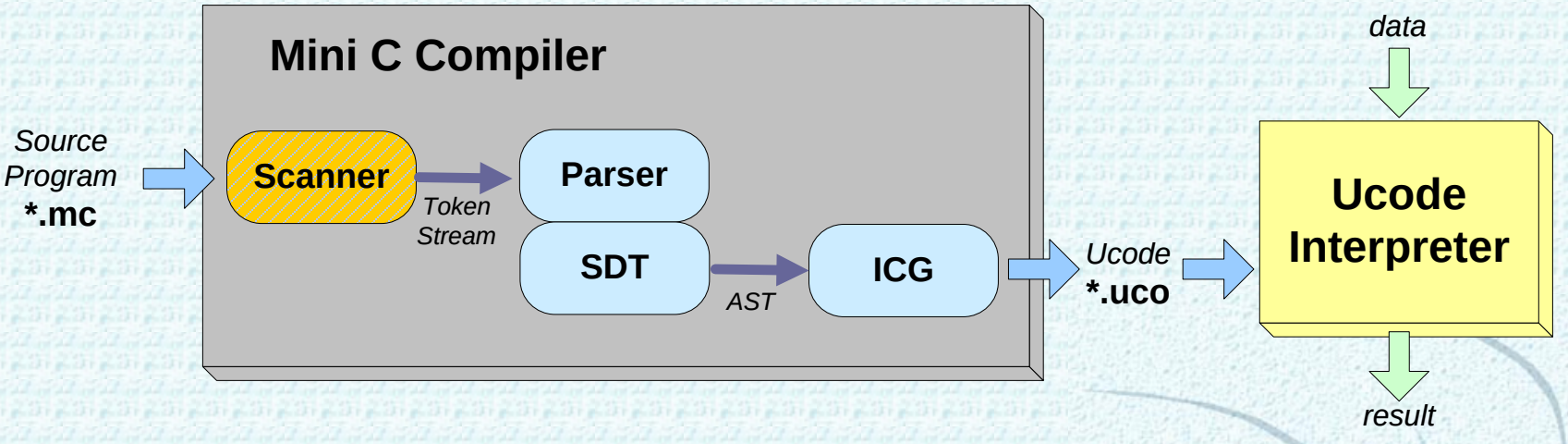
  - Reserved symbols (7개)

const else if int return void while

- State diagram for Mini C -- pp.143-144
- Mini C Scanner Source -- pp.145-150

형식언어 입문 숙제 #2

- **연습문제 4.11(교과서 167쪽)**
- **Implementation Model for an Experimental Compiler:**



Lexical Analysis

Lexical Analyzer(Scanner) for Expression Grammar

- Expression Grammar

E -> E + T | E – T | T

T -> T * F | T / F | T mod F | F

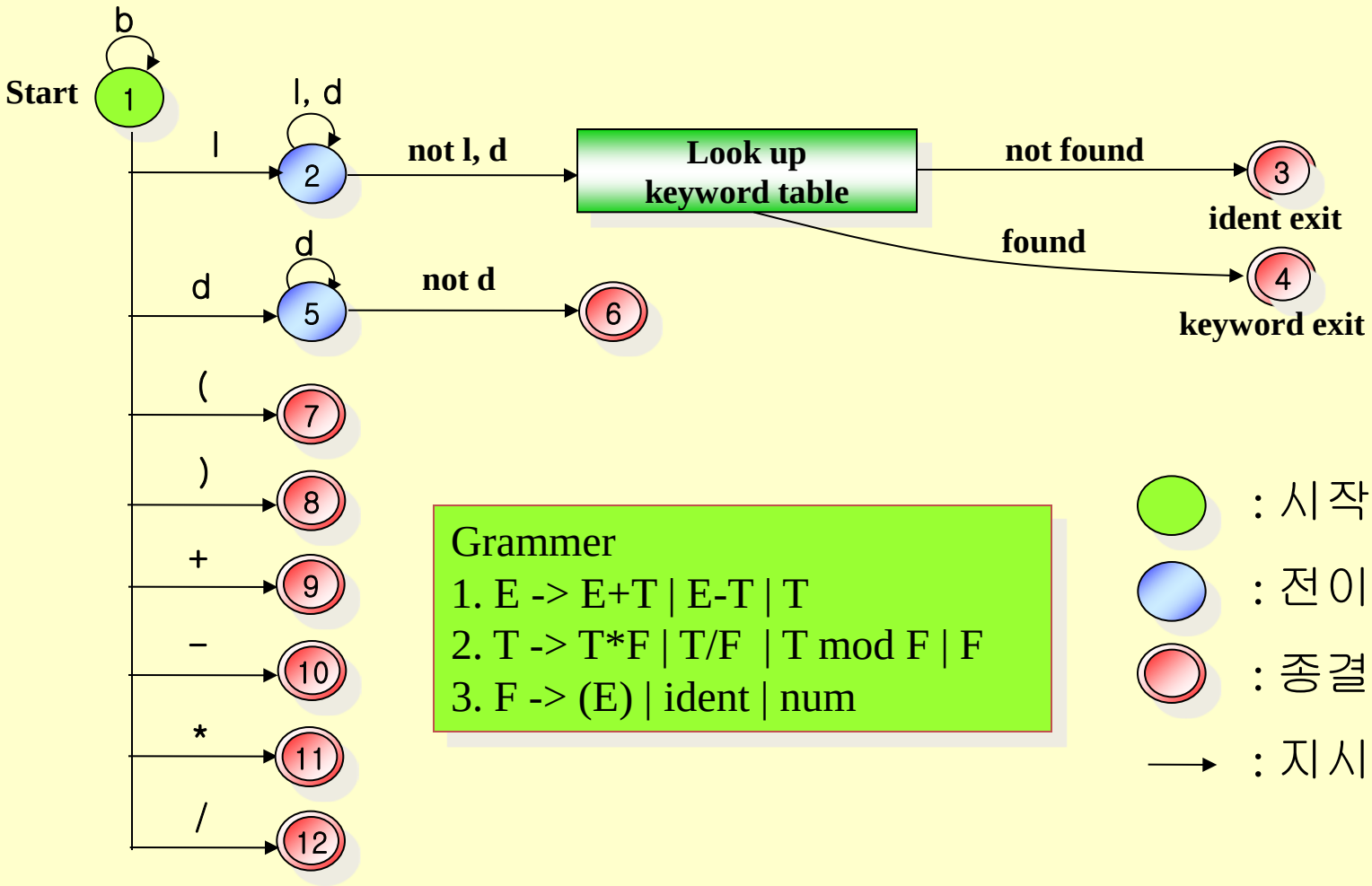
F -> (E) | num

- The Tokens of Expression Grammar
  - Special symbol(6개)

    ( ) * / + -
  - Reserved symbol(1개)

    mod
- State diagram for Expression Grammar
- Scanner Source for Expression Grammar

State Diagram for Expression Grammar



• Expression Grammar - Scanner Source

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#define ID_LENGTH 12
#define NUMKEYWORD 1

enum tsymbol {
      tnull=0,     tlparen,     trparen,     tmul,          tdiv,
      tadd,          tsub,          tident,      tnumber,     teof,
      kw_mod
};
```

```
struct tokentype {
        int number;        // token number
        union{
                char id[ID_LENGTH];
                 int num;
        }value;              // token value
};

 char*keyword[] = { “mod" };

 enum tsymbol tnum[] = { modsym };
```

```
/*************************************************************************
                '('      ')'      '*'     ‘/‘      'mod'     '+'      '-'     'NUM'     ‘$'
 *************************************************************************/
```

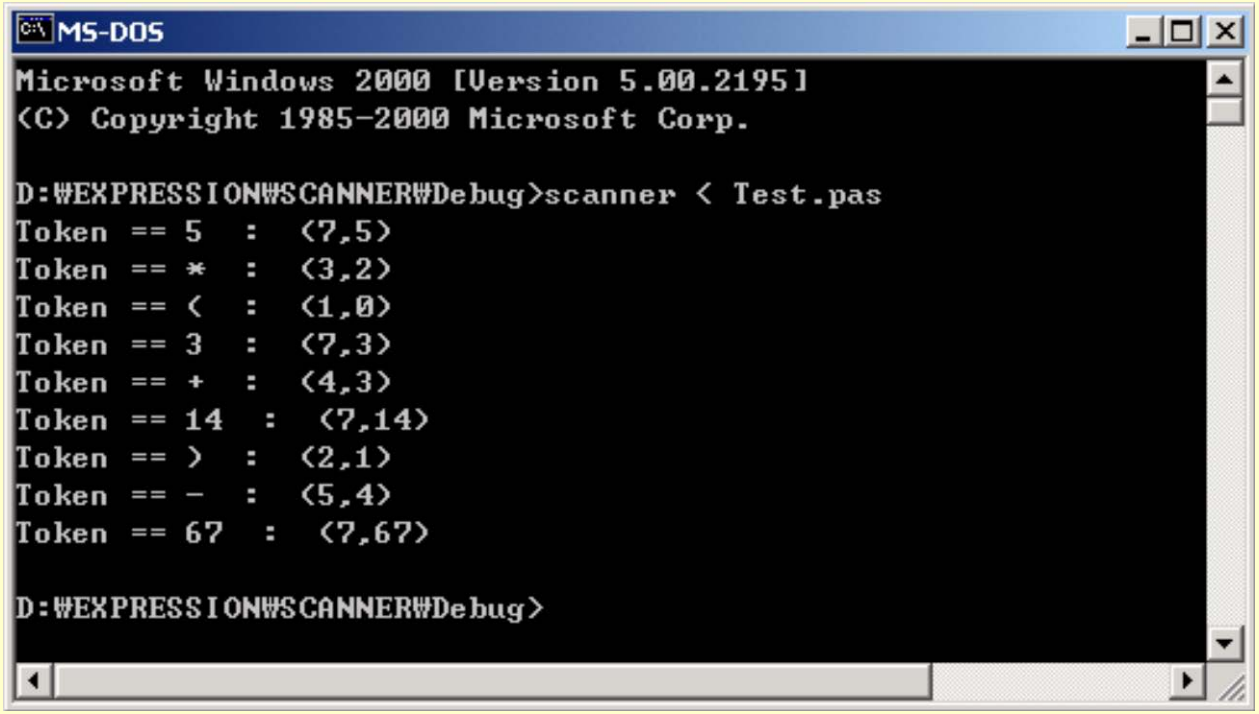
```
struct tokentype scanner(){
    struct tokentype token;
    int i, j, k, num;
    char ch, id[ID_LENGTH];

    token.number = tnull;
    do{
        while (isspace(ch = getchar())) ;          // state 1 : skip blanks
        if (isalpha(ch)) {                         // state 2 : identifier or keyword
            i = 0;
            do {
                if  (i < ID_LENGTH) id[i++] = ch;
                ch = fgetc(fp);
            }while (isalnum(ch));
            id[i] = '\0';
            ungetc(ch, stdin);      // retract
```

```
    /* find the identifier in the keyword table */
    i=0;
    j=NUMKEYWORD-1;

    do {                        // binary search
        k=(i+j) / 2;
        if (strcmp(id, keyword[k]) >= 0)        i = k + 1;
        if (strcmp(id, keyword[k]) <= 0)        j = k - 1;
    } while (i <= j);

    if ((i-1) > j ) {           // found, keyword exit
        token.number = tnum[k];
        strcpy(token.value.id, id);
    } else{                     // not found, identifier exit
        token.number = tident;
        strcpy(token.value.id, id);
    }
}
```

```
else if (isdigit(ch)){                    // state 5 : number
        num=0;
        do{
            num = 10*num + (int)(ch-'0');
            ch = fgetc(fp);
         } while (isdigit(ch));
         ungetc(ch, fp);                  // retract
         token.number = tnumber;
         token.value.num = num;
}// number

else { switch (ch) {          // special characters
      case '(' : strcpy( token.value.id, "(“ );
                 token.number=tlparen; break;
      case ')' : strcpy(token.value.id, ")");
                 token.number=trparen; break;
      case '+' : strcpy(token.value.id, "+");
                  token.number=tadd; break;
      case '-' : strcpy(token.value.id, "-");
                 token.number=tsub; break;
```

```
                    case '*' : strcpy(token.value.id, "*");
                               token.number=tmul; break;
                    case EOF: token.number=teof; break;
                  }
                }
        }while(token.number==tnull);
        return token;
    }

    void main()
    { struct tokentype token;

      token = scanner();
      while (token.number != teof) {
          if (token.number==tnumber)
              printf("Token -----> %-12d : ( %2d, %7d ) \n",
                        token.value.num, token.number, token.value.num);
            else if (token.number==teof) exit(0);
            else printf("Token -----> %-12s : ( %2d, %7d ) \n", token.value.id, token.number, 0);
           token=scanner();
    }
}
```

Data : 5*(3+14)-67


MS-DOS
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.
D:₩EXPRESSION₩SCANNER₩Debug>scanner < Test.pas
Token == 5 : (7,5)
Token == * : (3,2)
Token == (: (1,0)
Token == 3 : (7,3)
Token == + : (4,3)
Token == 14 : (7,14)
Token ==) : (2,1)
Token == - : (5,4)
Token == 67 : (7,67)
D:₩EXPRESSION₩SCANNER₩Debug>

Lexical Analysis

Lexical Analyzer(Scanner) for MiniC Grammar

- **Tokens of MiniC (p.142)**

  - **Special symbol(30개)**

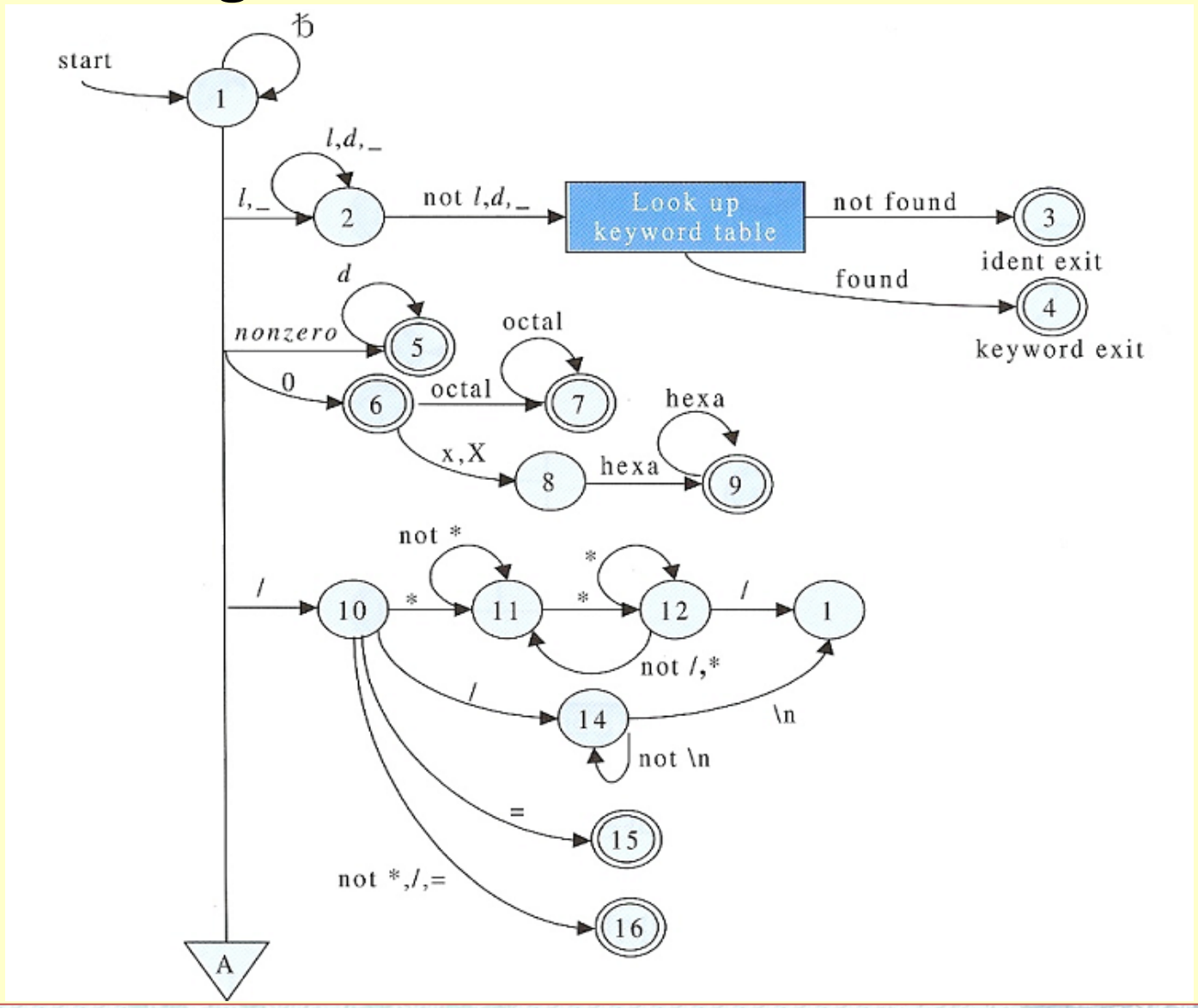
!	!=	%	%=	&&	(	)	*
*=	+	++	+=	,	-	--	-=
/	/=	;	<	<=	=	==	>
>=	[	]	{	||	}		

  - **Reserved symbol(7개)**

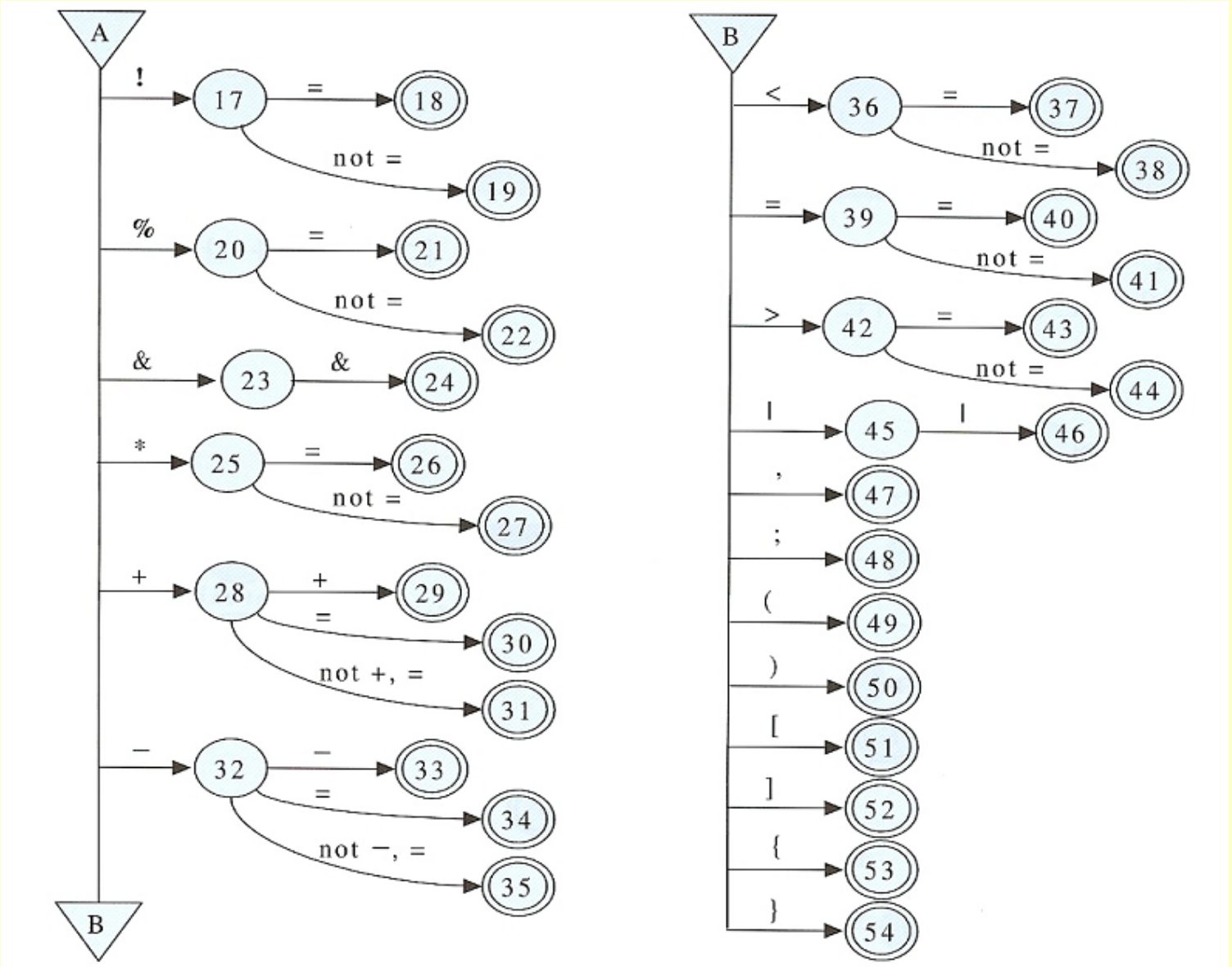
const	else	if	int	return	void	while

- **State diagram for MiniC (pp.143-144)**
- **MiniC Scanner Source (pp.145-150)**

- **State Diagram-1 for MiniC**



- **State Diagram-2 for MiniC**



• MiniC Scanner Source

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

#define ID_LENGTH  12
#define NO_KEYWORDS  7

enum tsymbol{  tnull=-1,
      tnot,    tnotequ,    tmod,    tmodAssign,    tident,
      tnumber,  tand,    tlparen,    trparen,    tmul,
      tmulAssign,    tplus,    tinc,    taddAssign,
      tcomma,    tminus,    tdec,    tsubAssign,

      tdiv,    tdivAssign,  tsemicolon,  tless,    tlesse,
      tassign, tequal,    tgreat,    tgreate,    tlbracket,
      trbracket,    teof,    tconst,    telse,
      tif,    tint,    treturn,    tvoid,    twhile,
      tlbrace,    tor,    trbrace

};
```

```
struct tokentype{
        int number;          // token number
        union{
            char id[ID_LENGTH];
            int num;
        }value;              // token value
};

char*keyword[] = { “const",  “else",    “if",
                   “int",    “return",  “void",
                   “while"
};

enum tsymbol tnum[] = {
        tconst,    telse,    tif,    tint
        treturn,   tvoid,    twhile
};
```

```
struct tokentype scanner(){
    struct tokentype token;
    int  i, j, k, num;
    char  ch, id[ID_LENGTH];

    token.number = tnull;
    do{
        while (isspace(ch = getchar())) ;          // state 1 : skip blanks
        if (isalpha(ch)) {                         // state 2 : identifier or keyword
          i = 0;
          do {
              if  (i < ID_LENGTH) id[i++] = ch;
              ch = getchar();
          }while (isalnum(ch));
          id[i] = '\0';
          ungetc(ch, stdin);      // retract
```

```
/* find the identifier in the keyword table */
i=0;
j=NUMKEYWORD-1;

do {                        // binary search
      k=(i+j) / 2;
      if (strcmp(id, keyword[k]) >= 0)          i = k + 1;
      if (strcmp(id, keyword[k]) <= 0)          j = k - 1;
} while (i <= j);

if ((i-1) > j ) {                   // found, keyword exit          // state 4 : keyword
    token.number = tnum[k];
    strcpy(token.value.id, id);
} else{                             // not found, identifier exit   // state 3 : identifier
    token.number = tident;
    strcpy(token.value.id, id);
}
}
```

```
else if (isdigit(ch)){                                  // state 5 : number
        num=0;
        do{
            num = 10*num + (int)(ch-'0');
            ch = getchar();
         } while (isdigit(ch));
         ungetc(ch, stdin);                             // retract
         token.number = tnumber;
         token.value.num = num;
}// number

else { switch (ch) {        // special characters
      case '/' : ch = getchar();                        // state 10
                 if (ch == '*') {                       // comment
                              do {
                     while (ch!='*') ch = getchar();
                     ch = getchar();
                   } while (ch != '/');
                 } else{ token.number = tlparen;
                         ungetc(ch, stdin);             // retract
     } break;
```

```
else { switch (ch) {        // special characters
    case '/' : ch = getchar();                          // state 10
        if (ch == '*') {                                // text comment
          do {
            while (ch!='*') ch = getchar();
            ch = getchar();
          } while (ch != '/');
        } else { if (ch == '/') {                       // line comment
                   while (getchar() != '\n')  ;
        } else { if (ch == '=') token.number = tdivAssign;
        } else { token.number = tdiv;
                 ungetc(ch, stdin);                     // retract
        } break;
    case '!':       …                                   // state 17
    case '%':       …                                   // state 20
    case '&':       …                                   // state 23
    case '*':       …                                   // state 25
    case '+':       …                                   // state 28
    case '-':       …                                   // state 32
```

```
                case '<':    …                          // state 36
                             ch = getchar();
                             if (ch == '=') token.number = tlesse;
                             else { token.number = tless;
                                    ungetc(ch, stdin); // retract
                             } break;
                case '=':    …                          // state 39
                case '>' :   …                          // state 42
                case '|':    …                          // state 45
                case '(':    token.number = tlparen;    // state 49
                             break;
                case ')':    …                          // state 50
                case '[':    …                          // state 51
                case ']':    …                          // state 52
                case '{':    …                          // state 53
                case '}':    …                          // state 54
        } // switch end
    } while (token.number == tnull);
    return token;
} // end of scanner
```

```
void main()
{ struct tokentype token;

  token = scanner();
  while (token.number != teof) {
          …                                          //Token 출력 루틴작성
  }
}
```

```
Token -----> program      : ( 34,       0 )
Token -----> perfect      : (  1, perfect )
Token -----> ;            : (  7,       0 )
Token -----> const        : ( 25,       0 )
Token -----> max          : (  1,     max )
Token -----> =            : ( 16,       0 )
Token -----> 500          : (  2,     500 )
Token -----> ;            : (  7,       0 )
Token -----> var          : ( 36,       0 )
Token -----> i            : (  1,       i )
Token -----> ,            : (  6,       0 )
Token -----> j            : (  1,       j )
Token -----> ,            : (  6,       0 )
Token -----> k            : (  1,       k )
Token -----> ,            : (  6,       0 )
Token -----> r            : (  1,       r )
Token -----> ,            : (  6,       0 )
Token -----> sum          : (  1,     sum )
Token -----> :            : (  8,       0 )
Token -----> integer      : ( 30,       0 )
Token -----> ;            : (  7,       0 )
Token -----> begin        : ( 24,       0 )
Token -----> i            : (  1,       i )
Token -----> :=           : (  9,       0 )
Token -----> 2            : (  2,       2 )
```

```
Token -----> ;            : (  7,       0 )
Token -----> while        : ( 37,       0 )
Token -----> i            : (  1,       i )
Token -----> <=           : ( 19,       0 )
Token -----> max          : (  1,     max )
Token -----> do           : ( 27,       0 )
Token -----> begin        : ( 24,       0 )
Token -----> sum          : (  1,     sum )
Token -----> :=           : (  9,       0 )
Token -----> 0            : (  2,       0 )
Token -----> ;            : (  7,       0 )
Token -----> k            : (  1,       k )
Token -----> :=           : (  9,       0 )
Token -----> i            : (  1,       i )
Token -----> div          : ( 26,       0 )
Token -----> 2            : (  2,       2 )
Token -----> ;            : (  7,       0 )
Token -----> j            : (  1,       j )
Token -----> :=           : (  9,       0 )
Token -----> 1            : (  2,       1 )
Token -----> ;            : (  7,       0 )
Token -----> while        : ( 37,       0 )
Token -----> j            : (  1,       j )
Token -----> <=           : ( 19,       0 )
Token -----> k            : (  1,       k )
Token -----> do           : ( 27,       0 )
```

4.4 LEX: A Lexical Analyzer Generator

M.E. Lesk

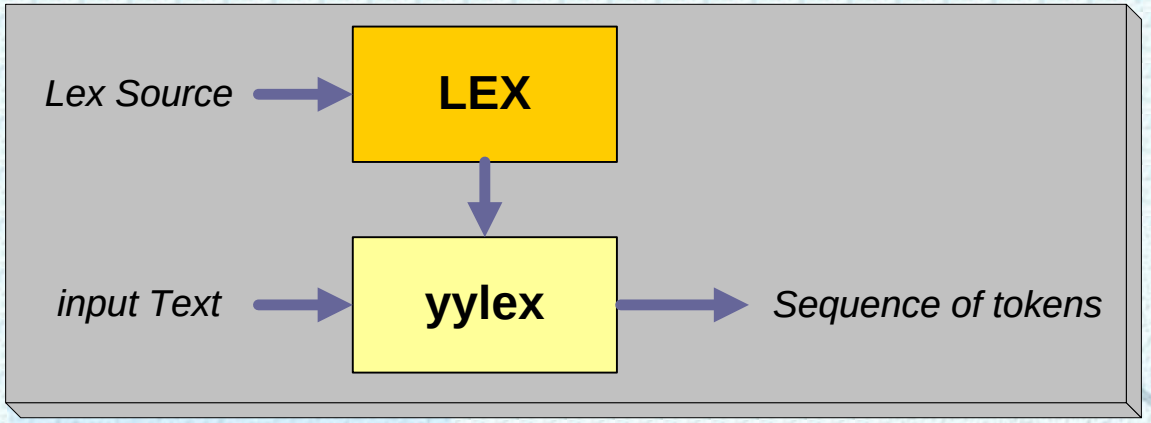
Bell laboratories,

Murry Hill, N.J. 07974

October, 1975

4.4.1 Introduction

- Lex helps write programs whose control flow is directed by instances of *regular expressions* in the input stream.
- Roles of Lex



(1) Lex translates the user's expressions and actions into the host general-purpose language; the generated program is named **lex.yy.c**.

(2) The **yylex** function will recognize expressions in a stream and perform the specified actions for each expression as it is detected.

4.4.2 Lex Source

- **format**:

```
{ definitions }
%%
{  rules  }
%%
{ user subroutines }
```

  - The second %% is optional, but the first is required to mark the beginning of the rules.
  - Any source not interpreted by Lex is copied into the generated program.

- **Rules** ::= ***regular expressions*** + ***actions***

```
ex) integer    printf("found keyword INT");
    color      { nc++; printf("color"); }
    [0-9]+     printf("found unsigned integer : %s\n", yytext);
```

4.4.3 Lex regular expressions

- Lex regular expressions
  ::= *text characters* + *operator characters*

- **Text characters** match the corresponding characters in the strings being compared. The letters of alphabet and the digits are always text characters.

- **Operator characters** --- " [] ^ - ? . * + () $ / {} % <>

  (1) " (double quote) --- whatever is contained between a pair of quotes is to be taken as text characters.

  ex) XYZ"++" <=> XYZ++

  (2) \ (backslash) --- single character escape.

  ex) XYZ\+\+ <=> XYZ++

(3) [ ] --- classes of characters.

(가) - (dash) --- specify ranges.

ex) [a-z0-9] indicates the character class containing all the lower case letters and the digits.

[-+0-9] matches all the digits and the two signs.

(나) ^ (hat) --- negate or complement.

ex) [^a-zA-Z] is any character which is not a letter.

(다) \ (backslash) --- escape character, escaping into octal.

ex) [\40-\176] matches all printable characters in the ASCII character set, from octal 40(blank) to octal 176(tilde).

(4) . --- the class of all characters except new line.
arbitrary character.
ex) "--".* <==> from "--" to end line

(5) ? --- an optional element of an expression.
ex) ab?c <=> ac or abc

(6) * , + --- repeated expressions
a* is any number of consecutive a characters, including zero.
a+ is one or more instances of a.
ex) [a-z]+
[0-9]+
[A-Za-z_] [A-Za-z0-9_]* --- Identifier

(7) | --- alternation

ex) (ab | cd) matches ab or cd.

(ab | cd+)?(ef)*

("+" | "–")? [0-9]+

(8) ^ --- new line context sensitivity.
matches only at the beginning of a line.

(9) $ --- end line context sensitivity.
matches only at the end of a line.

(10) / --- trailing context

ex) ab/cd matches the string ab, but only if followed by cd.

ex) ab$ <=> ab/\n

(11) < > --- start conditions.

(12) { } --- definition(macro) expansion.

4.4.4 Lex actions

- when an expression is matched, the corresponding action is executed.
- **default action**
  - copy the input to the output.
    this is performed on all strings not otherwise matched.
  - One may consider that actions are what is done instead of copying the input to the output.
- **null action** - ignore the input.

  ex) [\t\n] ;

  causes the three spacing characters (blank, tab, and newline) to be ignored.

- | (alternation)
  - the action for this rule is the action for the next rule.

    ex) [\t\n] ; <=> " " |
    "\t" |
    "\n" ;

- Global *variables* and *functions*

  (1) **yytext** : the actual context that matched the expression.

  ex) [a-z]+ printf("%s",yytext);

  (2) **yyleng** : the number of characters matched.

  ex) yytext[yyleng-1] : the last character in the string matched.

  (3) **ECHO** : prints the matched context on the output.

  ex) ECHO <===> printf("%s",yytext);

(4) yymore can be called to indicate that the next input expression recognized is to be tacked on to the end of this input

(5) yyless(n) : n개의 character만을 yytext에 남겨두고 나머지는 reprocess를 위하여 input으로 되돌려 보낸다.

(6) I/O routines

1) *input()* returns the next input character.

2) *output(c)* writes the characters c on the output.

3) *unput(c)* pushes the character c back onto the input stream to be read later by input().

(7) yywrap() is called whenever Lex reaches an end-of-file.

4.4.5 Ambiguous source rules

- **Rules**
  1) The longest match is preferred.
  2) Among rules which matched the same number of characters, the rule given **first** is preferred.

```
ex) integer     Keyword  action;
    [a-z]+      identifier action;
```

- **Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for <u>once and only once</u>.**

=====> REJECT : "go do the next alternative."

4.4.6 Lex Source definitions

- **Form:**

```
definitions
%%
 rules
 %%
 user routines
```

  - Any source **not interpreted by Lex** is copied into the generated program.
  - %{ %} is copied.
  - user routines is copied out **after** the Lex output.

- **Definitions**

  ::= ***dcl part*** + ***macro definition part***

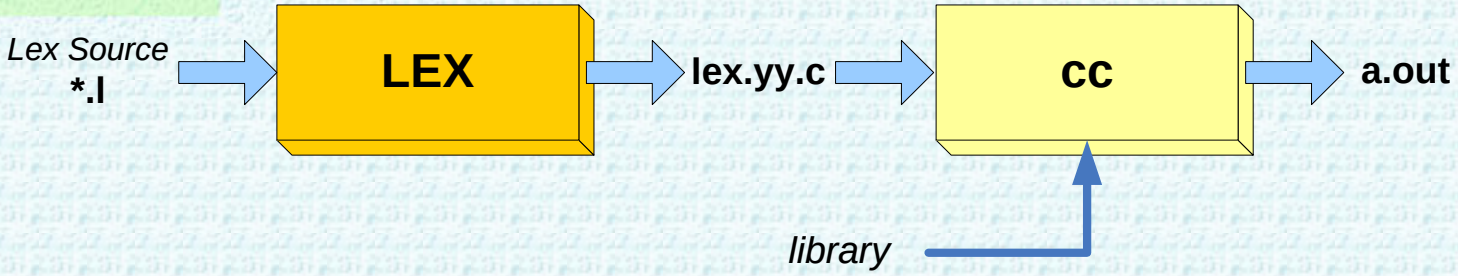
  - **Dcl part --- %{ ... %}**
  - **The format of macro definitions :**

    name translation
  - **The use of definition : {name}**

    ex)

```
D               [0-9]
L               [a-zA-Z]
%%
{L}({L}|{D})*   return IDENT;
```

4.4.7 Usage



- UNIX : **lex source**
  cc lex.yy.c -ll -lp

  where, libl.a : lex library
  libp.a : portable library.

4.4.8 Lex and Yacc

- **Yacc** will call yylex(). In this case, each **Lex** rule should end with return(token); where the appropriate token value is returned.
- Place #include "lex.yy.c" in the last section of Yacc input.

```
ex)   lex    better
      yacc   good
      cc     y.tab.c -ly -ll -lp

      where, liby.a : Yacc library
             libl.a : Lex  library
             libp.a : portable library
```

  - The Yacc library(-ly) should be loaded before the Lex library, to obtain a main program which invokes the Yacc parser.

4.4.9 Summary

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y
[x-z]	the characters x, y, or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
x$	an x at the end of a line.
<y>x	an x when Lex is in start condition y.
x?	an optional x.

x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x | y	an x or y.
(x)	an x
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.

Lexical Analysis

LEX(test.l, mc.l) for Test, MiniC Grammar

test.l (pp.160-161)

```
%{
#include <stdio.h>
#include <stdlib.h>
enum tnumber  { TEOF, TIDEN, TNUM, TASSIGN, TADD, TSEMI, TDOT, TBEGIN,
                TEND, TERROR};
%}
letter [a-zA-Z_]
digit  [0-9]
%%
begin                   return(TBEGIN);
end                     return(TEND);
[A-Za-z][A-Za-z0-9]*    return(TIDEN);      // {letter}({letter}|{digit})*
[0-9]+                  return(TNUM);       // {digit}+
":="                    return(TASSIGN);
"+"                     return(TADD);
";"                     return(TSEMI);
\.                      return(TDOT);
[ \t\n]                 ;
.                       return(TERROR);
%%
```

```
void main()
{
        enum tnumber token;                 /*token number*/
        printf("Start of Lex\n");
        while ((token=yylex()) != TEOF) {
           switch(token) {
                    case TBEGIN         :  printf("Begin\n"); break;
                    case TEND           :  printf("End\n"); break;
                    case TIDEN          :  printf("Identifier : %s\n", yytext); break;
                    case TASSIGN        :  printf("Assignment_op\n"); break;
                    case TADD           :  printf("Add_op\n"); break;
                    case TNUM           :  printf("Number : %d\n", atoi(yytext)); break;
                    case TSEMI          :  printf("Semicolon\n"); break;
                    case TDOT           :  printf("Dot\n"); break;
                    case TERROR         :  printf("Error : %c\n", yytext[0]); break;
             }
        }
}
int yywrap()
{   printf("End of Lex\n");
    return 1;
}
```

데이터 파일 : **test.dat**

```
begin
num = 0;
num = num + 526;
end.
```

```
[carotple@coe carotple]$ ./test < tt.dat
Start of Lex
Begin
Identifier : num
Assignment_op
Number : 0
Semicolon
Identifier : num
Assignment_op
Identifier : num
Add_op
Number : 526
Semicolon
End
Dot
End of Lex
```

mc.l (pp.162-163)

```
%{
/*Lex Source for Mini C */
%}
%%
“const"      return(TCONST);
“else"       return(TELSE);
“if"         return(TIF);
“return"     return(TRETURN);
“void"       return(TVOID);
“while"      return(TWHILE);
“=="         return(TEQUAL);
“!="         return(TNOTEQUAL);
“<="         return(TLESSE);
“>="         return(TGREATE);
“&&"         return(TAND);
“||"         return(TOR);
“++"         return(TINC);
“--"         return(TDEC);
“+="         return(TADDASSIGN);
“-="         return(TSUBASSIGN);
   :         /* 중간생략 */
“{"          return(TLBRACKET);
“]"          return(TRBRACKET);
“{"          return(TLBRACE);
“}"          return(TRBRACE);
[A-Za-z][A-Za-z0-9]*   return(TIDENT);
[0-9]+       return(TNUM);
"(*"([^*]|"*"+[^*)])*"*"+")“   ;
[ \t\n]      ;
.            return(yytext[0]);
%%
int  yywrap()
{
     return 1;
}
```

```
Token -----> program      : ( 34,       0 )
Token -----> perfect      : (  1, perfect )
Token -----> ;            : (  7,       0 )
Token -----> const        : ( 25,       0 )
Token -----> max          : (  1,     max )
Token -----> =            : ( 16,       0 )
Token -----> 500          : (  2,     500 )
Token -----> ;            : (  7,       0 )
Token -----> var          : ( 36,       0 )
Token -----> i            : (  1,       i )
Token -----> ,            : (  6,       0 )
Token -----> j            : (  1,       j )
Token -----> ,            : (  6,       0 )
Token -----> k            : (  1,       k )
Token -----> ,            : (  6,       0 )
Token -----> r            : (  1,       r )
Token -----> ,            : (  6,       0 )
Token -----> sum          : (  1,     sum )
Token -----> :            : (  8,       0 )
Token -----> integer      : ( 30,       0 )
Token -----> ;            : (  7,       0 )
Token -----> begin        : ( 24,       0 )
Token -----> i            : (  1,       i )
Token -----> :=           : (  9,       0 )
Token -----> 2            : (  2,       2 )
```

```
Token -----> ;            : (  7,       0 )
Token -----> while        : ( 37,       0 )
Token -----> i            : (  1,       i )
Token -----> <=           : ( 19,       0 )
Token -----> max          : (  1,     max )
Token -----> do           : ( 27,       0 )
Token -----> begin        : ( 24,       0 )
Token -----> sum          : (  1,     sum )
Token -----> :=           : (  9,       0 )
Token -----> 0            : (  2,       0 )
Token -----> ;            : (  7,       0 )
Token -----> k            : (  1,       k )
Token -----> :=           : (  9,       0 )
Token -----> i            : (  1,       i )
Token -----> div          : ( 26,       0 )
Token -----> 2            : (  2,       2 )
Token -----> ;            : (  7,       0 )
Token -----> j            : (  1,       j )
Token -----> :=           : (  9,       0 )
Token -----> 1            : (  2,       1 )
Token -----> ;            : (  7,       0 )
Token -----> while        : ( 37,       0 )
Token -----> j            : (  1,       j )
Token -----> <=           : ( 19,       0 )
Token -----> k            : (  1,       k )
Token -----> do           : ( 27,       0 )
```