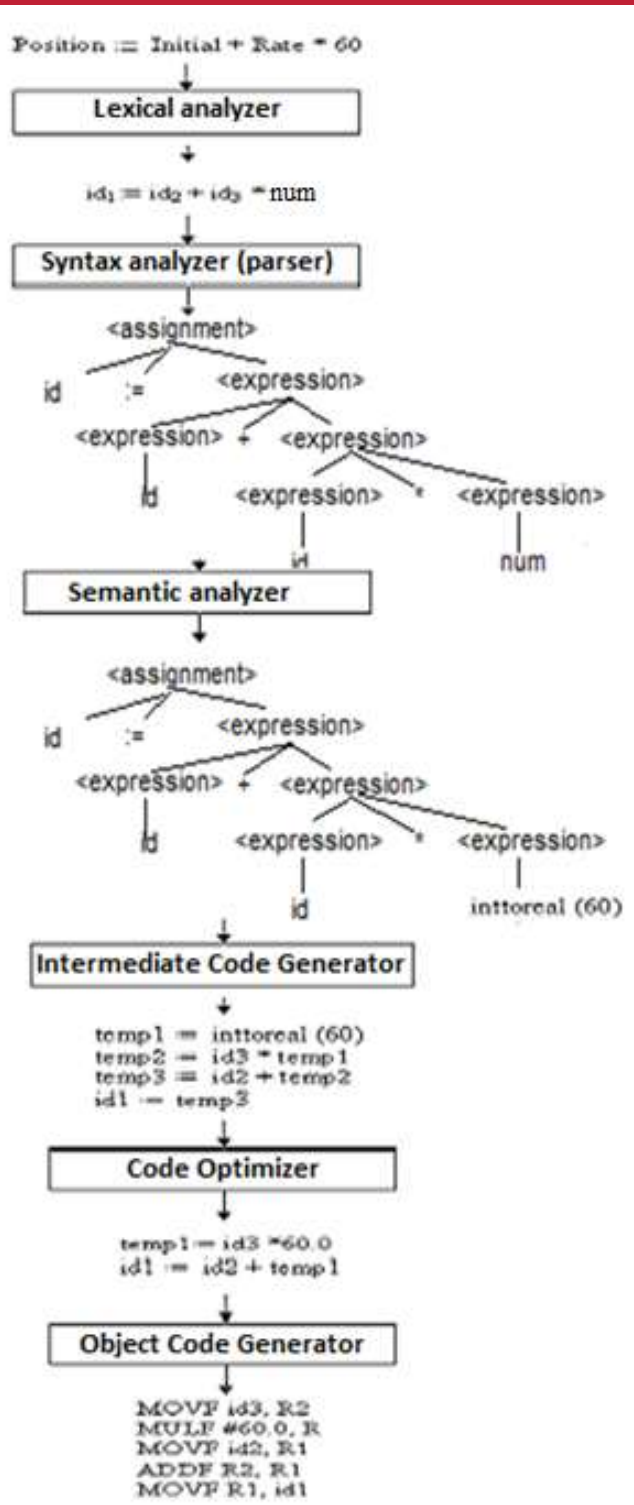
Experiment in
Compiler Construction

Scanner design

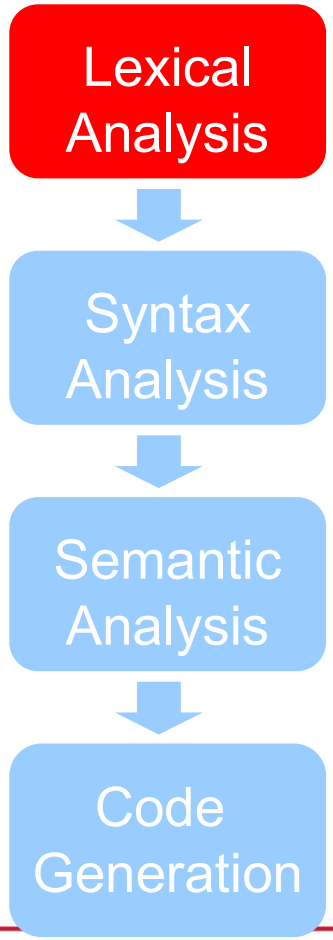
ONE LOVE. ONE FUTURE.

Translation of a statement

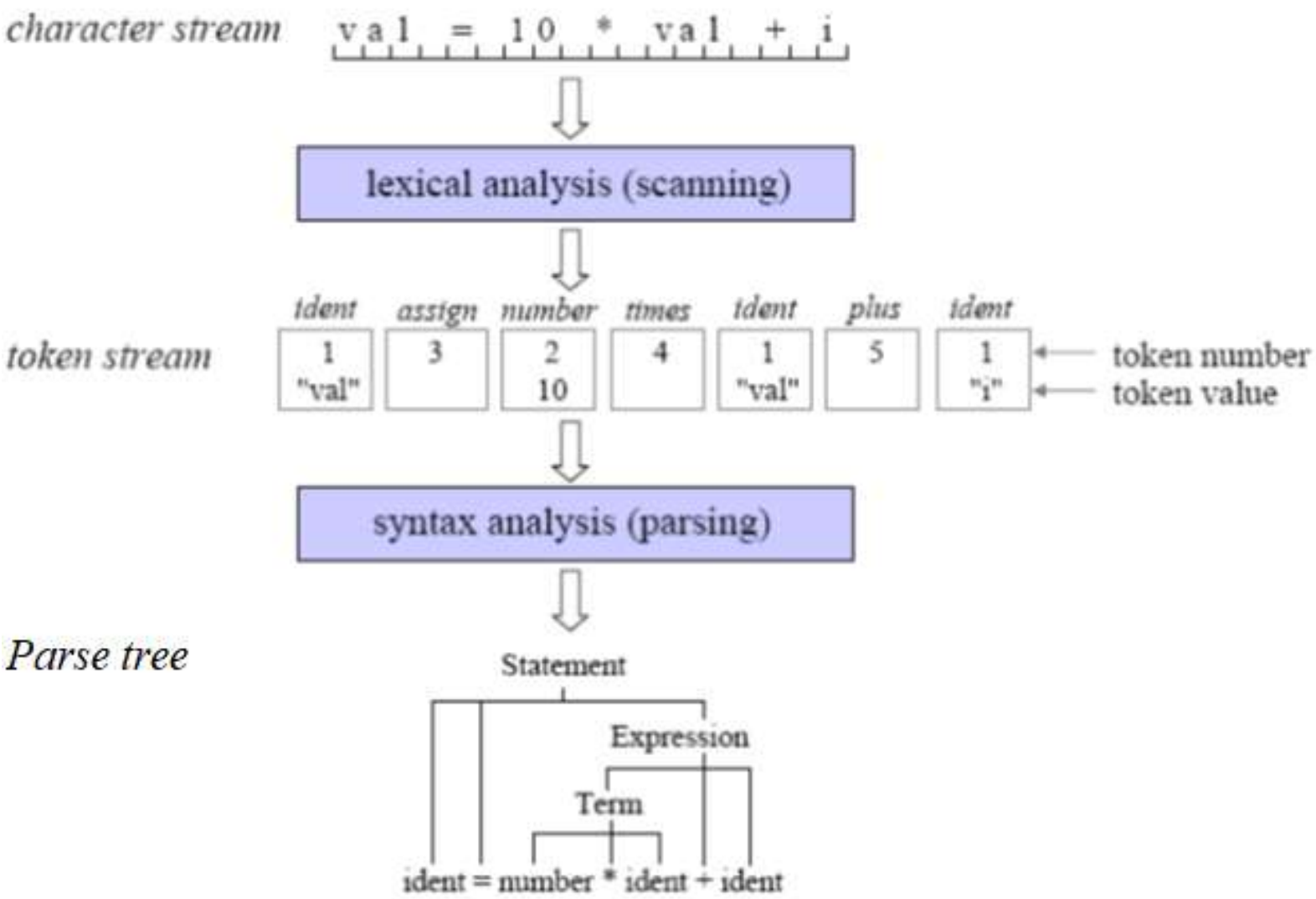


What is a scanner?

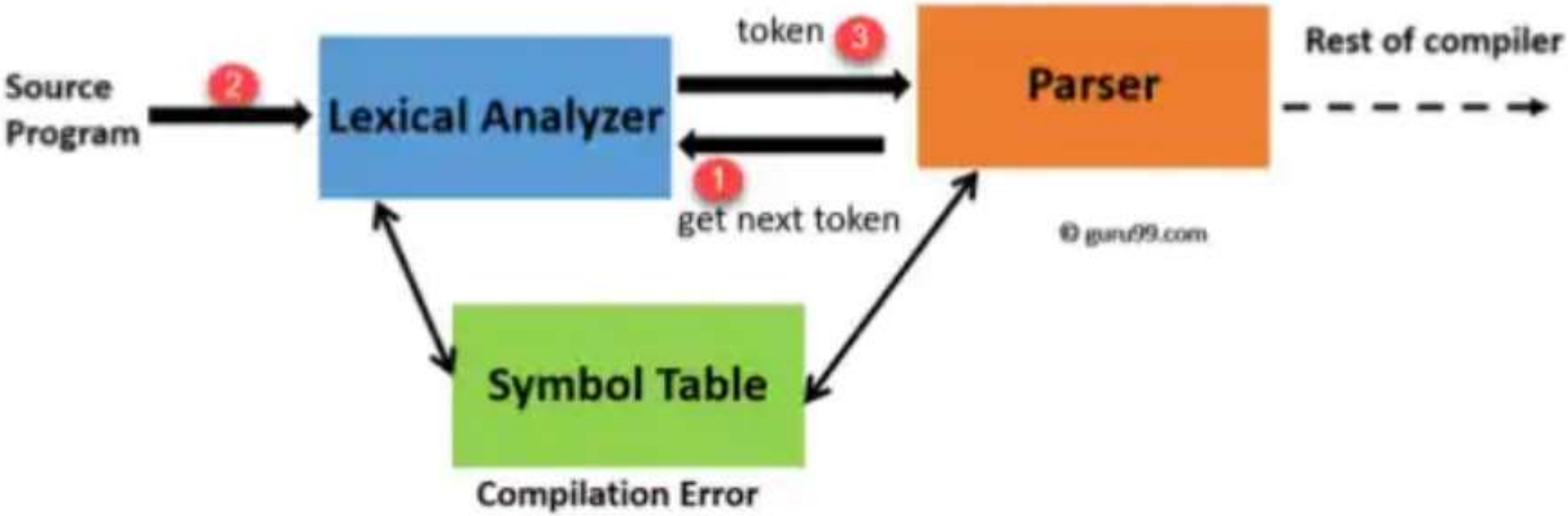
- The compiler's component/module that perform the job of lexical analysis (scanning) is called *scanner*.
- Compiler's first phase



What is a scanner?



Scanner – Parser interaction



Tasks of a scanner

- Skip meaningless characters: blank, tab, new line character, comment.
- Recognize illegal character
- Return error message
- Recognize different types of token
  - identifier
  - keyword
  - number
  - special symbols
  - ...
- Pass recognized tokens to the *parser* (the module that perform the job of syntatic analysis)

Lexical rules of KPL

- Only use unsigned integer
- The KPL identifier is made with a combination of lowercase or uppercase letters, digits. An identifier must start with a letter. The length <=15.
- Only allows character constants. A character constant is enclosed with a pair of single quote marks. ‘’
- The language do not use string constant.
- - is use for subtraction only. The language does not allow unary minus and negative numbers
- The relational operator “not equal to” is represented by !=

KPL's alphabet

- Letter: a b c … x y z
  A B C … X Y Z
- Digit: 0 1 2 … 8 9
- Special character:
  - + - * /
  - > < ! =
  - [space] ,(comma) . : ; ‘ _
  - ( )

KPL's tokens

- Keywords

  PROGRAM, CONST, TYPE, VAR, PROCEDURE, FUNCTION, BEGIN, END, ARRAY, OF, INTEGER, CHAR, CALL, IF, THEN, ELSE, WHILE, DO, FOR, TO

- Operators

  := (assign)

  + (addition), - (subtraction), * (multiplication), / (division)

  = (comparison of equality), != (comparison of difference), > (comparison of greaterness), < (comparison of lessness), >= (comparison of greaterness or equality), <= (comparison of lessness or equality)

KPL's tokens

- Special characters

  ; (semicolon), . (period), : (colon), , (comma), ((left parenthesis),) (right parenthesis), ‘ (singlequote)

- Also

  (. and .) to mark the index of an array element

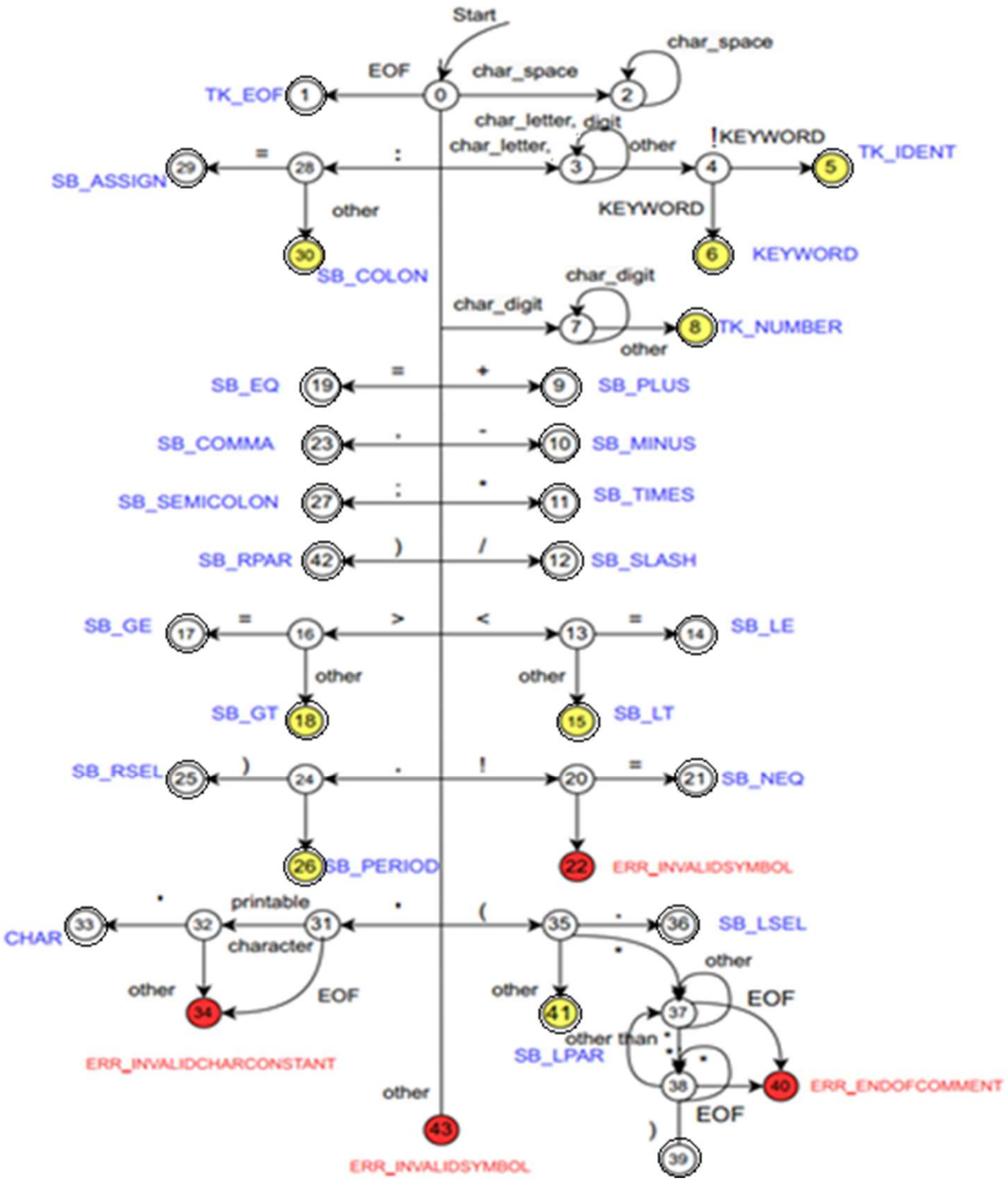
  (* and *) to mark the comment

- Others

  identifier, number, illegal charater

Recognizing KPL's tokens

- All KPL's tokens make up a regular language.
- They can be described with regular grammar, regular expression
- They can be recognized by a Deterministic Finite Automaton (DFA)
- The scanner is a big DFA



The scanner as a Deterministic Finite Automaton

After every recognized token, the scanner starts in state 0 again

If an illegal character is met, the scanner would change to the states 22 or 43 which tell the scanner to stop scanning and return error messages.

Notice the yellow states

Input, output of scanner

- Input

Program Example1; (* Example 1*)
Begin
End. (* Example1*)

- Output

1-1:KW_PROGRAM
1-9:TK_IDENT(Example1)
1-17:SB_SEMICOLON
2-1:KW_BEGIN
3-1:KW_END
3-4:SB_PERIOD

KPL scanner - organization

#	Filename	Task
1	Makefile	Project
2	scanner.c	Main
3	reader.h, reader.c	Read the source code
4	charcode.h, charcode.c	Classify character
5	token.h, token.c	Classify and recognize token, keywords
6	error.h, error.c	Manage error types and messages

KPL scanner – reader

```
// Read a character from input stream
int readChar(void);

// Open input stream
int openInputStream(char *fileName);

// Close input stream
void closeInputStream(void);

// Current line number and column number
int lineNo, colNo;

// Current character
int currentChar;
```

KPL scanner – charcode

- In *charcode.c*, we define *charCodes* array that associates every ASCII character with an unique predifined *CharCode*.
- *getc*() function may return EOF (or -1) which is not an ASCII character.

KPL scanner – charcode

```
typedef enum {
  CHAR_SPACE,             // space
  CHAR_LETTER,            // character
  CHAR_DIGIT,             // digit
  CHAR_PLUS,              // '+'
  CHAR_MINUS,             // '-'
  CHAR_TIMES,             // '*'
  CHAR_SLASH,             // '/'
  CHAR_LT,                // '<'
  CHAR_GT,                // '>'
  CHAR_EXCLAMATION,       // '!'
  CHAR_EQ,                // '='
  CHAR_COMMA,             // ','
  CHAR_PERIOD,            // '.'
  CHAR_COLON,             // ':'
  CHAR_SEMICOLON,         // ';'
  CHAR_SINGLEQUOTE,       // '\''
  CHAR_LPAR,              // '('
  CHAR_RPAR,              // ')'
  CHAR_UNKNOWN            // invalid character
} CharCode;
```

KPL scanner – token

```
typedef enum {
  TK_NONE,       // Invalid token - Error
  TK_IDENT,      // Identifier token
  TK_NUMBER,     // Number token
  TK_CHAR,       // Character constant token
  TK_EOF,        // End of program token
  // keywords
  KW_PROGRAM, KW_CONST, KW_TYPE, KW_VAR,
  KW_INTEGER, KW_CHAR, KW_ARRAY, KW_OF,
  KW_FUNCTION, KW_PROCEDURE,
  KW_BEGIN, KW_END, KW_CALL,
  KW_IF, KW_THEN, KW_ELSE,
  KW_WHILE, KW_DO, KW_FOR, KW_TO,
  // Special character
  SB_SEMICOLON, SB_COLON, SB_PERIOD, SB_COMMA,
  SB_ASSIGN, SB_EQ, SB_NEQ, SB_LT, SB_LE, SB_GT, SB_GE,
  SB_PLUS, SB_MINUS, SB_TIMES, SB_SLASH,
  SB_LPAR, SB_RPAR, SB_LSEL, SB_RSEL
} TokenType;
```

KPL scanner – token

```
// Structure of a token
typedef struct {
  char string[MAX_IDENT_LEN + 1];
  int lineNo, colNo;
  TokenType tokenType;
  int value;
} Token;

// Check whether a string is a keyword or not
TokenType checkKeyword(char *string);

// Create new token, provided type of token and location
Token* makeToken(TokenType tokenType, int lineNo, int
colNo);
```

KPL scanner – error management

```
// List of error may occur in lexical analysis
typedef enum {
  ERR_ENDOFCOMMENT,
  ERR_IDENTTOOLONG,
  ERR_NUMBERTTOOLONG,
  ERR_INVALIDCHARCONSTANT,
  ERR_INVALIDSYMBOL
} ErrorCode;
// Error message
#define ERM_ENDOFCOMMENT "End of comment expected!"
#define ERM_IDENTTOOLONG "Identification too long!"
#define ERM_INVALIDCHARCONSTANT "Invalid const char!"
#define ERM_INVALIDSYMBOL "Invalid symbol!“
#define ERM_NUMBERTOOLONG "Value of integer number
exceeds the range!"
```

KPL scanner – scanner.c

```
// Read the source code and show tokens
int scan(char *fileName) {
  Token *token;

  if (openInputStream(fileName) == IO_ERROR)
    return IO_ERROR;

  token = getToken();
  while (token->tokenType != TK_EOF) {
    printToken(token);
    free(token);
    token = getToken();
  }

  free(token);
  closeInputStream();
  return IO_SUCCESS;
}
```

KPL scanner – getToken function

```
Token* getToken(void)
{
  Token *token;
  switch(state)
  {
  case 0:
        if (currentChar == EOF) state =1;
        else
        switch (charCodes[currentChar])
        {
                case CHAR_SPACE:
                        state =2;break;
                case CHAR_LETTER:
                        ln=lineNo;
                        cn=colNo;
                        state =3;
                        break;
                case CHAR_DIGIT:
                        state =7;
                        break;
                case CHAR_PLUS:
                        state = 9;
                        break;
 case … // more cases
```

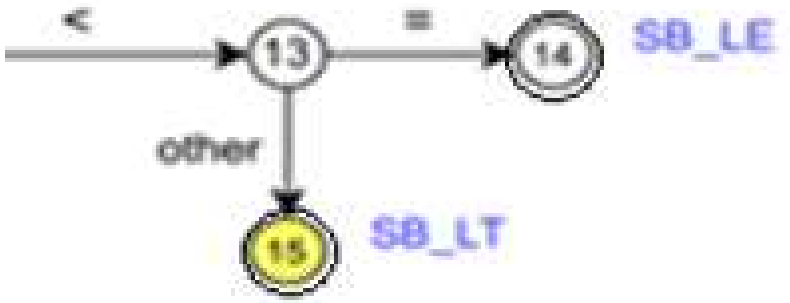
KPL scanner – getToken function

```
case 0:
...
case CHAR_PLUS:
                          state =9;
                          break;
...
```


```
case 9:
        readChar();
    return makeToken(SB_PLUS, lineNo,
colNo-1);
```

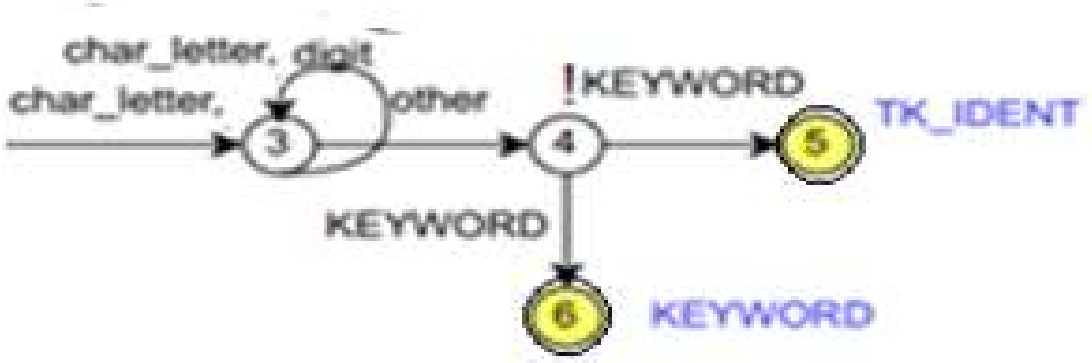
getToken: token composed of 2 characters

```
case 13:
    readChar();
        if (charCodes[currentChar] == CHAR_EQ) state = 14;
         else state =15;
        return getToken();
  case 14:
        readChar();
        return makeToken(SB_LE, lineNo, colNo-1);
  case 15:
        return makeToken(SB_LT, lineNo, colNo-1);
```



getToken: identifier and keyword recognition

- An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting character must be a letter.
- Length of identifiers not over than 15 (MAX_IDENT_LENGTH)
- Keywords are case insensitive?
- How about identifiers?
- Function checkKeyword is completed. Input, output?

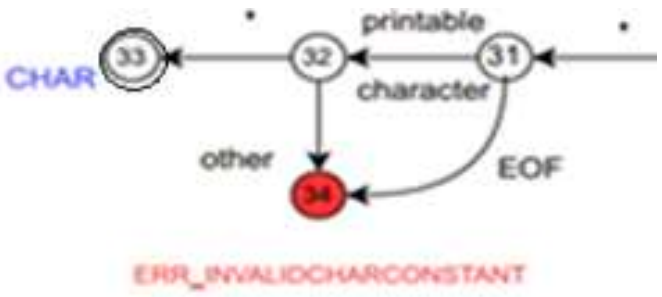


getToken: number recognition

```
typedef struct
{
  char string[];
  int lineNo, colNo;
  TokenType tokenType;
  int value;
} Token;
```

- Convert string to integer with function atoi (stdlib.h)
- Range 0÷ 2^{31}-1 (Not more than 10 digits)

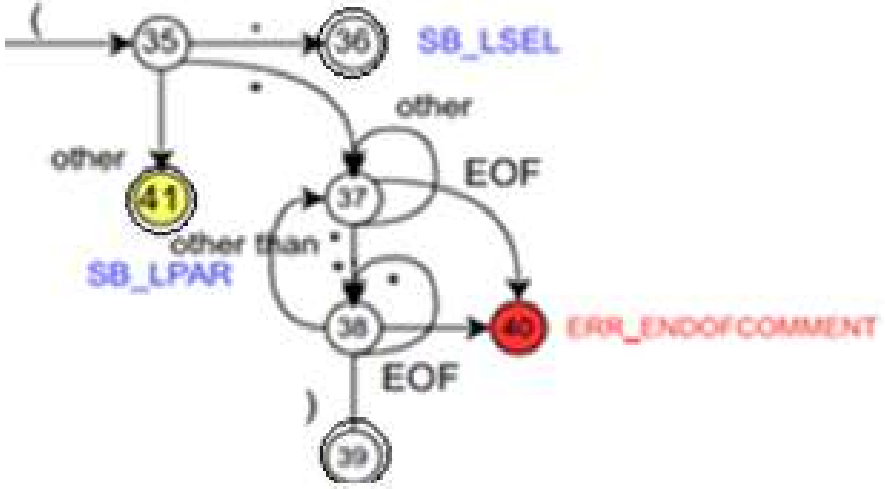
getToken: character constant recognition




```
case 31:
    readChar();
        if (currentChar == EOF)
                state=34;
        else
                        if(isprint(currentChar))
                                state =32;
                        else state =34;
                return getToken();
case 32:
        c= currentChar;
        readChar();
        if (charCodes[currentChar] == CHAR_SINGLEQUOTE)
                        state=33;
                else
                        state =34;
                return getToken();
case 33:
        token = makeToken(TK_CHAR, lineNo, colNo-1);
        token->string[0] =c;
        token->string[1] ='\0';
        readChar();
        return token;
case 34:
        error(ERR_INVALIDCHARCONSTANT, lineNo, colNo-2);
```

getToken: Skip comment

```
case 35: // tokens begin with lpar, skip comments
        ln = lineNo;
    cn = colNo;
    readChar();
    if (currentChar == EOF)
                state=41;
    else
        switch (charCodes[currentChar])
                {
                case CHAR_PERIOD:
                        state =36;
                                break;
                case CHAR_TIMES:
                        state =38;
                                break;
                default:state =41;
         }
    return getToken();
```



Assignment

- Complete following function in `scanner.c`
  - **`Token* getToken(void);`**