

# CS 280 Programming Language Concepts Spring 2022

**Programming Assignment 1** 

**Building a Lexical Analyzer** 



## Programming Assignment 1

#### Objectives

- □ building a lexical analyzer for a small programming language.
- □ Writing a C++ program to test the lexical analyzer.

#### Notes:

- □ Read the assignment carefully to understand it.
- ☐ Make a list of the lexical rules of the language and the assigned tokens.
- □ Understand the functionality of the testing program, the required information to be collected and printed out.



## Programming Assignment 1

- In this programming assignment, you will be building a lexical analyzer for small programming language and a program to test it. This assignment will be followed by two other assignments to build a parser and interpreter to the same language.
- The syntax definitions of a small programming language, called Pascal-Like Simple Language (PLSL), are given below using EBNF notations. The PLSL syntax has some features similar to the well-known Pascal Language. The details of the meanings (i.e. semantics) of the language constructs will be given later on.

## Pascal-Like Small Language (PLSL)

```
1. Prog ::= PROGRAM IDENT; DeclBlock ProgBody
2. DeclBlock ::= VAR {DeclStmt;}
3. DeclStmt ::= Ident {, Ident} : (Integer | Real | String)
4. ProgBody ::= BEGIN {Stmt;} END
5. Stmt ::= AssigStmt | IfStmt | WriteLnStmt | ForStmt
6. WriteLnStmt ::= WRITELN (ExprList)
7. IfStmt ::= IF ( LogicExpr ) THEN Stmt [ELSE Stmt]
8. ForStmt ::= FOR Var := ICONST (TO | DOWNTO) ICONST DO Stmt
9. AssignStmt ::= Var := Expr
10. ExprList ::= Expr {, Expr}
11. Expr ::= Term \{(+|-) \text{ Term}\}
12. Term ::= SFactor { ( * | / ) SFactor}
13. SFactor ::= [(+ | -)] Factor
14. LogicExpr ::= Expr (= | > | <) Expr
15. Var ::= IDENT
16. Factor ::= IDENT | ICONST | RCONST | SCONST | (Expr)
```



- Identifiers (IDENT)
  - □ IDENT := [Letter \_] {(Letter | Digit | \_)}
  - $\square$  Letter := [a-z A-Z]
  - $\Box$  Digit := [0-9]
- Integer constants (ICONST)
  - $\Box$  ICONST := [0-9]+
- Real constants (RCONST)
  - $\square$  RCONST := ([0-9]\*)\.([0-9]+)
  - □ For example: real number constants such as 12.0 and .2 are accepted, but 2. and 2.45.2 are not.

# 10

- String constants (SCONST)
  - □ String literals are defined as a sequence of characters delimited by single quotes, that should all appear on the same line.
  - ☐ For example:
    - 'Hello to CS 280.' is a string literals.
    - While, "Hello to CS 280." Or 'Hello to CS 280." are not.
- Operators of the language are: +, -, \*, /, :=, (, ), =, >, <.
  - □ These operators are for plus, subtract, multiply, divide, assignment, left parenthesis, right parenthesis, equality, greater than, and less than operations. They have the following tokens, respectively: PLUS, MINUS, MULT, DIV, ASSOP, LPAREN, RPAREN, EQUAL, GTHAN, LTHAN.



- Reserved words: *program*, *end*, begin, *writeln*, *if*, *integer*, *real*, *string*, *for*, *to*, *downto*, *else*, *var*.
  - □ These reserved words have the following tokens, respectively: PROGRAM, END, BEGIN, WRITELN, IF, INTEGER, REAL, STRING, FOR, TO, DOWNTO, ELSE and VAR.
- The semicolon, colon, and comma characters are terminals with the following tokens: SEMICOL, COLON, and COMMA.
- A comment is defined by all the characters following the sequence of characters "(\*" as starting delimiters to the closing delimiters "\*)". Comments may overlap one line, as multi-line comments. A recognized comment is skipped and does not have a token.



- An error will be denoted by the ERR token.
- End of file will be denoted by the DONE token.
- White spaces are skipped.



## Lexical Analyzer Implementation

■ You will write a lexical analyzer function, called getNextToken having the following signature:

```
LexItem getNextToken (istream& in, int& linenumber);
```

- □ First argument is a reference to an istream object that the function should read from (input file).
- □ Second reference is an integer that contains the current line number of the line read from the input file.
- □ getNextToken returns a LexItem object. A LexItem is a class that contains a token, a string for the lexeme, and the line number as data members.

# м

## Implementation Issues

- Questions to be considered:
  - □ What patterns need to be recognized?
  - □ Is there a different approach for multi-character tokens and single character tokens?
  - □ What are the states?
  - ☐ How do you need to represent states?
- The lexical rules represent the patterns your lexical analyzer must recognize
  - ☐ You should understand the patterns and build a DFA representing all of the patterns
    - This will tell you what states you need in your implementation
  - □ Assignment requires writing code to implement the DFAs
  - □ Write pseudocode for the function.
  - ☐ Implement one state at a time.



## Implementation Issues

- Token Types:
  - $\square$  Single character tokens, such as +, -, \*, /, =, <, >, :
    - These are easy to recognize.
  - ☐ Two characters tokens such as ":="
    - First character is the same as another one for a token: ':' (COLON)
    - Needs to lookahead (peek) to decide which token these belong to, a COLON or ASSOP.
  - ☐ Multi-character tokens such as IDENT, ICONST, RCONST, and SCONST
    - Create a state for each type of token, for example INID for recognizing identifiers (IDENT), ININT for recognizing integer constants (ICONST), and INSTRING for recognizing string literals (SCONST).



## Implementation Issues

- □ Keywords' tokens such as *program*, *end*, *begin*, *writeln*, *if*, *integer*, *real*, *string*, *for*, *to*, *downto*, *else*, *var* are treated as identifiers
  - Each identifier is checked against a directory of keywords mapped to tokens. If the keyword identifier is found, then its reserved word token is returned, otherwise the IDENT token is returned.

# м

- Possible States:
  - □ START, INID, INSTRING, ININT, INREAL, INCOMMENT
  - □ This means, starting from the SART state, I have seen a character to make a transition into a state where you will be collecting characters for an identifier, a string constant, an integer constant, a real number or a comment.
- Each state has different rules \*inside\* that state. For example,
  - □ Zero or more letters, digits, or underscores for identifier.
  - □ All characters following a single quote for a quoted string (note, newline in string is an error, and also note there are no escape characters inside the string). A string must be terminated by a single quote.
- Use an enumerated type to define the states:

```
enum TokState {START, INID, INSTRING, ININT, INREAL,
INCOMMENT} lexstate = START;
```



- getNextToken outline
  - □ Set initial state to Start.
  - □ Loop, reading a character at a time from the input file till EOF is reached or an ERR is found. For each read character,
    - Select a block of code to execute based on the current state
      - ☐ Might need to change state based on a character (ex: a letter in the START state indicates the beginning of an identifier, so change to INID).
    - Return when a token is recognized.
  - ☐ If the end of file is found, return a DONE token; else, return an ERR token with a meaningful message.

#### Pseudocode

```
if( lexstate == START ) {
// START code
else if( lexstate == INID ) {
 // INID code
else if (. . .)
//other states follow
```

```
switch( lexstate ) {
case START: // START code
      break;
case INID: // INIT code
      break;
//other states follow
```



- A header file, "lex.h", is provided for you. You MUST use the provided header file. You may NOT change it. Lex.h includes the following
  - □ Definitions of all the possible token types
  - □ Class definition of LexItem
  - □ Some function prototypes
- You should implement the lexical analyzer function "lex.cpp" in one source file.
- You should implement a test main program in another source file.



};

```
//Definitions of all the possible token types
enum Token {
       // keywords
       PROGRAM, WRITELN, INTEGER, BEGIN, END, IF, REAL, STRING,
       VAR, ELSE, FOR, THEN, DO, TO, DOWNTO,
       // an identifier
       IDENT,
       // an integer, real, and string constant
       ICONST, RCONST, SCONST,
       // the operators, and delimiters
       PLUS, MINUS, MULT, DIV, ASSOP, LPAREN, RPAREN, COMMA,
       EQUAL, GTHAN, LTHAN, SEMICOL, COLON,
       // any error returns this token
       ERR,
       // when completed (EOF), return this token
       DONE
```

```
м
```

```
class LexItem { //Class definition of LexItem
       Token token;
       string lexeme;
       int lnum;
public:
       LexItem() {
               token = ERR;
                                           constructors
               lnum = -1;
       LexItem(Token token, string lexeme, int line) {
               this->token = token;
               this->lexeme = lexeme;
                                                   overloaded
               this->lnum = line;
                                                   operators
bool operator == (const Token token) const { return this->token ==
token; }
bool operator!=(const Token token) const { return this->token !=
token; }
                                                        getter
       Token GetToken() const { return token; }
                                                        methods
       string GetLexeme() const { return lexeme; }
       int GetLinenum() const { return lnum; }
};
```

## 7

```
bool operator==(const Token token) const {
    return this->token == token; }
bool operator!=(const Token token) const {
    return this->token != token; }
```

- The "overloaded operators" methods defined in LexItem are used to compare a LexItem object to a Token in your testing program using the "==" or "!=" operators.
  - ☐ This allows you to write code like this:

```
LexItem t;
t = getNextToken(...);//LexItem object
If (t.operator==(DONE) || t.operator==(ERR))
{ ... }
//or more conveniently written as
if(t == DONE || t == ERR) { ... }
```

# М

## Implementation Issues (cont'd)

```
//functions to be implemented
extern ostream& operator<<(ostream& out, const LexItem& tok);
extern LexItem id_or_kw(const string& lexeme, int linenum);
extern LexItem getNextToken(istream& in, int& linenum);</pre>
```

#### External Definitions

- "extern" tells the compiler that someone will provide functions with these signatures. \*you\* are the someone. The functions can be added to "lex.cpp" and your test program as specified below in order to avoid linking problems on Linux.
  - The operator<< function is an overload operator that lets you print a LexItem object to an output stream (more about overloaded operators in the next week class.) Please, implement the function operator<< in the test program.
  - id\_or\_kw() is a function that searches reserved words directory, and returns its corresponding token, if it is found, or the IDENT token if it is not. Implement the function in the lex.cpp file.

```
//a segment of the getNextToken code
LexItem getNextToken(istream& in, int& linenum) {
  enum TokState { START, INID, INSTRING, ININT, INREAL, INCOMMENT}
  lexstate = START;
  string lexeme;
  char ch;
  while(in.get(ch)) {
         switch( lexstate ) {
         case START:
               if(ch == '\n')
                  linenum++;
               if( isspace(ch) )
               lexeme = ch;
               if( isalpha(ch) ) {
             else if (...) //more possibilities to consider
             break;
         Case ININT:
               Break;
                                                                    22
         //Other states will follow
```



#### Points to be considered

- ☐ Your getNextToken function might need to look at the next character from input to decide if the token is finished or not.
  - Method 1: use the peek() method, to examine the next character, and only read it if it belongs to the token.
  - Method 2: if you read a character that does not belong to the token, use the putback () method to put it back, so that you get() it next time
- □ Any error detected by the lexical analyzer should result in a LexItem object to be returned with the ERR token, and the lexeme value equal to the string recognized when the error was detected.
- □ Note also that both ERR and DONE are unrecoverable. Once the getNextToken function returns a LexItem object for either of these tokens, you shouldn't call getNextToken again.

- Testing Program Issues
  - □ main() function that takes several command line arguments, called flags:
    - -v, -iconsts, -rconsts, -sconsts, -ident, and
    - filename argument must be passed to main function. Your program should open the file and read from that filename. Only one file name is allowed.
    - Read the rules for the flags and understand what does each one of them mean.
  - ☐ You need to keep a record of all the required information by creating directories for each one of them:
    - All identifiers
    - All integer constants
    - All real constants
    - All string literals



- Outline of the testing program
  - ☐ Process arguments: flags and input file
  - □ Call getNextToken until it returns DONE or ERR
    - Keep a record of all lexemes returned for each token type to be printed out.
  - □ Print out the required information according to the provided flags
- Note: Do not include the "lex.cpp in the testing program.



```
int lineNumber = 0;
LexItem tok;
ifstream file;
....
while((tok = getNextToken(file, lineNumber)) != DONE && tok != ERR ) {
    // handle flags mode
    // keep required information
    ...
}
```

- getNextToken function will read one character at a time.
- The main program will process the input one token at a time.
- The counts and required information directories are kept in main.



- Required information
  - ☐ How many lines in the input?
  - ☐ How many tokens in the input?
  - □ What strings are in the input?
  - ☐ What integers are in the input?
  - □ What real numbers are in the input?
  - □ What identifiers are in the input?
- You are provided by a set of 17 testing files associated with Programming Assignment 1. Vocareum automatic grading will be based on these testing files. These are available in compressed archive "PA1 Test Cases.zip" on Canvas assignment.

## Examples

- Example 1:
  - ☐ Input file: "reals"

□ Output with −rconst flag:

```
Lines: 3
Tokens: 5
REALS:
0.47
0.75
0.8
15.25
23.5
```

## Examples

- Example 2: Invalid string
  - ☐ Input file: "invstr1"

```
'Please type the coordinates of three points"

'The center of the circle through these points is'

End of
File -----
```

□ Output with –sconst flag:

Error in line 1 ('Please type the coordinates of three points")



☐ Input file: "comments"

□ Output with –v flag:

```
PROGRAM
IDENT (PROG1)
SEMICOL
BEGIN
IDENT (R)
EQUAL
ICONST (50)
END
DOT
Lines: 10
Tokens: 8
```



## Examples

- Example 4: Identifiers
  - ☐ Input file: "idents"

```
PROGRAM prog1;
  (* Testing all flags *)
  VAR
  integer _x1, y_1, z-, w234;
  string str := 'Welcome';
  real z = 0.0, _35;

begin
  r = 50;
  for x =1 to 10 do
  for y = 10 down to 0 do
END
```

End of File

□ Output with −ident flag:

```
Lines: 12
Tokens: 47
IDENTIFIERS:
DOWN, PROG1, R, STR, W234, X, Y, Y_1, Z, _35, _X1
```



### **Submission**

- Deadline: Sunday March 6, 2022
  - □ Submit all your implementation files for the "lex.cpp" and testing program through Vocareum. The "lex.h" header file will be propagated to your Work Directory.
  - □ Submissions after the due date are accepted with a fixed penalty of 25%. No submission is accepted after Wednesday 11:59 pm, March 9, 2022.

