
NJIT
THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Fall 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. Although, we are not concerned about the syntax definitions of the language in this assignment, we intend to introduce it ahead of Programming Assignment 2 in order to show the language reserved words, constants, and operators.
- The syntax definitions of the small programming language are given below using EBNF notations. However, the details of the meanings (i.e. semantics) of the language constructs will be given later on.

CS 280 Projects' Language Definition

1. Prog ::= PROGRAM IDENT StmtList END PROGRAM
2. StmtList ::= Stmt; { Stmt; }
3. Stmt ::= DeclStmt | ControlStmt
4. DeclStmt ::= (INT | FLOAT | BOOL) VarList
5. VarList ::= Var { ,Var }
6. ControlStmt ::= AssigStmt | IfStmt | PrintStmt
7. PrintStmt ::= PRINT (ExprList)
8. IfStmt ::= IF (Expr) THEN StmtList { ELSE StmtList } END IF
9. AssignStmt ::= Var = Expr
10. Var ::= IDENT
11. ExprList ::= Expr { , Expr }
12. Expr ::= LogORExpr ::= LogANDExpr { || LogANDRxpr }
13. LogANDExpr ::= EqualExpr { && EqualExpr }
14. EqualExpr ::= RelExpr [== RelExpr]
15. RelExpr ::= AddExpr [(< | >) AddExpr]
16. AddExpr :: MultExpr { (+ | -) MultExpr }
17. MultExpr ::= UnaryExpr { (* | /) UnaryExpr }
18. UnaryExpr ::= (- | + | !) PrimaryExpr | PrimaryExpr
19. PrimaryExpr ::= IDENT | ICONST | RCONST | SCONST | BCONST | (Expr)

Lexical Rules for Tokens to be Recognized

- Identifiers (IDENT)
  - IDENT := [Letter _] {(Letter | Digit | _ | @)}
  - Letter := [a-z A-Z]
  - Digit := [0-9]
- Integer constants (ICONST)
  - ICONST := [0-9]+
- Real constants (RCONST)
  - RCONST := ([0-9]+)\.([0-9]+)
  - For example: real number constants such as 12.0 and 0.2 are accepted, but 2., .2 and 2.45.2 are not.
- Boolean constants (BCONST)
  - BCONST := (true | false)

Lexical Rules for Tokens to be Recognized

- String constants (SCONST)
  - String literals are defined as a sequence of characters delimited by double 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, less than, logical AND, logical OR, and NOT operations. They have the following tokens, respectively: PLUS, MINUS, MULT, DIV, ASSOP, LPAREN, RPAREN, EQUAL, GTHAN, LTHAN, AND, OR, and NOT.

Lexical Rules for Tokens to be Recognized

- The reserved words of the language are: *program*, *end*, *print*, *if*, *int*, *float*, *bool*, *else*, *true*, *false*.
  - These reserved words have the following tokens, respectively: PROGRAM, END, PRINT, IF, INT, FLOAT, BOOL, THEN, ELSE, TRUE, FALSE.
- 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.

Lexical Rules for Tokens to be Recognized

- 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.

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:
  - Single character tokens, such as +, -, *, /, =, <, >, :
    - These are easy to recognize.
  - Two characters tokens such as "==", "&&", and "||"
    - First character in "==" is the same as another one for a token: '=' (ASSOP)
    - Needs to lookahead (peek) to decide which token these belong to, a ASSOP or EQUAL.
  - 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*, *print*, *if*, *int*, *float*, *bool*, *else*, *true*, *and false* 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. However, if the string value of a keyword corresponds to either the TRUE or FALSE tokens, the function should return a LexItem object containing the BCONST token instead.

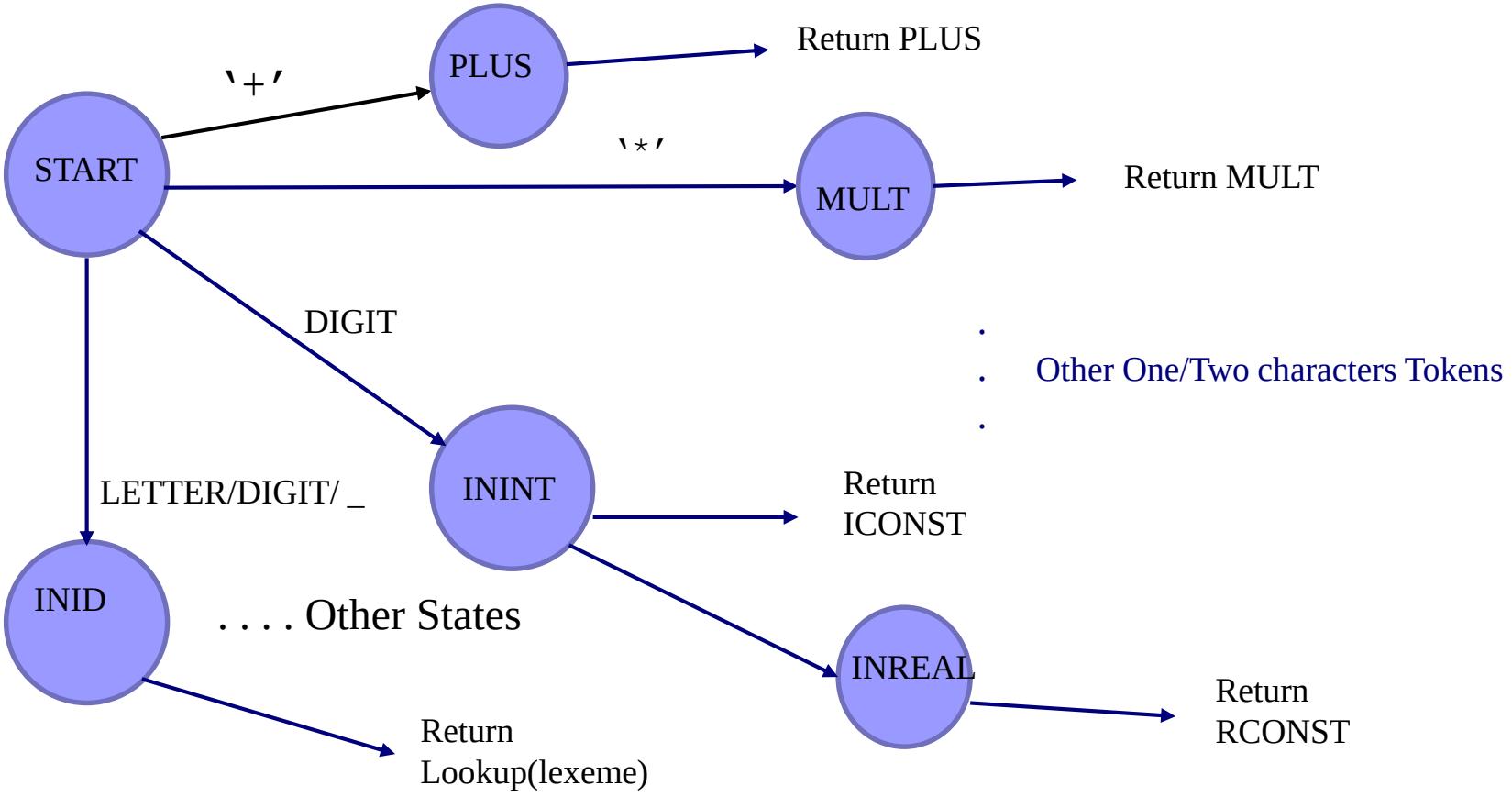
Implementation Issues (cont'd)

- Possible States:
  - START, INID, INSTRING, ININT, INREAL, INCOMMENT
  - This means, starting from the START 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, underscores, and at symbol (called asperand) character for identifier.
  - All characters following a double 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 double quote.
- Use an enumerated type to define the states:

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

Implementation Issues (cont'd)

- DFAs Diagrams



Implementation Issues (cont'd)

- `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.

Implementation Issues (cont'd)

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
. . .
}
```

Implementation Issues (cont'd)

- 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.

Implementation Issues (cont'd)

```
//Definition of all the possible token types
enum Token {
        // keywords
        PROGRAM, PRINT, INT, END, IF, FLOAT, BOOL,
        ELSE, THEN, TRUE, FALSE,

        // an identifier
        IDENT,

        // an integer, real, and string constant
        ICONST, RCONST, SCONST, BCONST,

        // the operators, parens, semicolon
        PLUS, MINUS, MULT, DIV, ASSOP, LPAREN, RPAREN, COMMA,
        EQUAL, GTHAN, LTHAN, SEMICOL, AND, OR, NOT,
        // any error returns this token
        ERR,

        // when completed (EOF), return this token
        DONE
};
```

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

        Token   GetToken() const { return token; }
        string  GetLexeme() const { return lexeme; }
        int     GetLinenum() const { return lnum; }
};
```

constructors

overloaded operators

getter methods

Implementation Issues (cont'd)

```
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 ) { … }
```

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);
```

- External Definitions
  - "extern" tells the compiler that someone will provide functions with these signatures. *you* are the someone. Include the implemented functions in the "lex.cpp" file. The definitions of the functions are:
    - 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.)
    - 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.

```
//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;
        //Other states will follow
```

Implementation Issues (cont'd)

- 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.

Implementation Issues (cont'd)

- 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

Implementation Issues (cont'd)

- 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.**

Implementation Issues (cont'd)

```
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.

Implementation Issues (cont'd)

- 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 18 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: “realerr1”

End of File →

```
23.5  15.25
0.75
.8   .47
```

  - Output with –v –rconst flags:

```
RCONST(23.5)
RCONST(15.25)
RCONST(0.75)
Error in line 3 (.)
```

Examples

- Example 2: Invalid string
  - Input file: “invstr1”

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


        "The center of the circle through these points is'

        "Its radius is 25 cm"
```

End of File →

  - Output with –sconst flag:

```
Error in line 4 ("The center of the circle through these points is' )
```

- Example 3: Multi-line comments
  - Input file: “comments”

```
PROGRAM prog1
        /* Testing all flags

        int x1, y1;
        string str = "Welcome";
        float z = 0.0 */

        r = 50;
END
```

End of File →

  - Output with –v flag:

```
PROGRAM
IDENT(PROG1)
IDENT(R)
ASSOP
ICONST(50)
SEMICOL
END
Lines: 9
Tokens: 7
```

Examples

- Example 4: Identifiers
  - Input file: “idents”

```
PROGRAM prog1
        /* Testing all flags

        int x1, y1;
        string str = "Welcome";
        float z = 0.0 */

        r = 50;
END
```

End of File →

  - Output with –v –ident flags:

```
PROGRAM
IDENT(PROG1)
IDENT(R)
ASSOP
ICONST(50)
SEMICOL
END
Lines: 9
Tokens: 7
```

Submission

- Deadline: Sunday October 23, 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, October 26, 2022.**


NJIT
THE EDGE IN KNOWLEDGE