
NJIT
THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Spring 2023

Programming Assignment 1

Building a Lexical Analyzer for the SPL Language

Programming Assignment 1

- Objectives
  - building a lexical analyzer for a small programming language, called Simple Perl-Like (SPL)
  - 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, called Simple Perl-Like (SPL), and a program to test it. This assignment will be followed by two other assignments to build a parser and interpreter to the SPL 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 determine the language terminals: reserved words, constants, identifier(s), and operators.
- The syntax definitions of the SPL 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.

Simple Perl-Like (SPL) Language Definition

1. Prog ::= StmtList
2. StmtList ::= Stmt ;{ Stmt; }
3. Stmt ::= AssignStme | WriteLnStmt | IfStmt
4. WriteLnStmt ::= WRITELN (ExprList)
5. IfStmt ::= IF (Expr) ‘{‘ StmtList ‘}’ [ELSE ‘{‘ StmtList ‘}’]
6. AssignStmt ::= Var = Expr
7. Var ::= NIDENT | SIDENT
8. ExprList ::= Expr { , Expr }
9. Expr ::= RelExpr [(-eq|==) RelExpr]
10. RelExpr ::= AddExpr [(-lt | -gt | < | >) AddExpr]
11. AddExpr :: MultExpr { (+ | - | .) MultExpr }
12. MultExpr ::= ExponExpr { (* | / | **) ExponExpr }
13. ExponExpr ::= UnaryExpr { ^ UnaryExpr }
14. UnaryExpr ::= [(- | +)] PrimaryExpr
15. PrimaryExpr ::= IDENT | SIDENT | NIDENT | ICONST | RCONST | SCONST | (Expr)

Lexical Rules for Tokens to be Recognized

- Identifiers (IDENT)
  - IDENT := [Letter _] {(Letter | Digit | _)}
  - Letter := [a-z A-Z]
  - Digit := [0-9]
  - Note that all identifiers are case sensitive.
- The language variables are either numeric scalar variables or string scalar variables. Numeric variables start by a "$" and followed by an IDENT. While a string variable starts by "@" and followed by an IDENT. Their definitions are as follows:
  - NIDENT := $ IDENT
  - SIDENT := @ IDENT
- Integer constants (ICONST)
  - ICONST := [0-9]+

Lexical Rules for Tokens to be Recognized

- Real constants (RCONST)
  - RCONST := ([0-9]+)\.([0-9]*)
  - For example, real number constants such as 12.0, and 0.2, 2. are accepted as real constants, but .2, and 2.45.2 are not. Note that ".2" is recognized as a dot (CAT operator) followed by the integer constant 2.
- 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.`

Lexical Rules for Tokens to be Recognized

- The reserved words of the language are: *writeln*, *if*, *and else*,.
  - These reserved words have the following tokens, respectively: WRITELN, IF, ELSE.
- The semicolon, comma, left parenthesis, right parenthesis, left braces, and right braces characters are terminals with the following tokens: SEMICOL, COMMA, LPAREN, RPAREN, LBRACES, and RBRACES, respectively.
- A comment is defined by all the characters following the character "#" to the end of line. A recognized comment is skipped and does not have a token.

Lexical Rules for Tokens to be Recognized

- The operators of the language are: +, -, *, /, ^, =, ==, >, <, . (dot), ** (repeat), -eq, -lt, and -gt. These operators are for add, subtract, multiply, divide, exponent, assignment, numeric equality, numeric greater than, numeric less than, string concatenation, string repetition, string equality, string less-than, and string greater-than operations, respectively. They have the following tokens, respectively: PLUS, MINUS, MULT, DIV, EXPONENT, ASSOP, NEQ, NGTHAN, NLTHAN, CAT, SREPEAT, SEQ, SLTHAN, and SGTHAN. Note that the string comparison operators -eq, -lt, and -gt are not case sensitive.
- 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 +, -, *, /, =, <, >, ^, . (dot), (,), {, }:
    - These are easy to recognize.
  - Two or more characters tokens such as “==“, “**”,:
    - First character in “==“ and “**” are the same as another one for the tokens ‘=’ (ASSOP), and ‘*’ multiplication.
    - Needs to lookahead (peek) to decide which token these belong to, a ASSOP or EQUAL.
  - Multi-character tokens such as IDENT, NIDENT, SIDENT, ICONST, RCONST, SCONST, string comparison operators (SEQ, SGT, and SLT)
    - 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) .
    - Transition from one state to another state is possible. For example from the state of recognizing ICONST to the state of recognizing RCONST.

Implementation Issues

- Keywords' tokens such as *writeln*, *if*, *and else* 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 an identifier token (IDENT, NIDENT, or SIDENT) is returned.

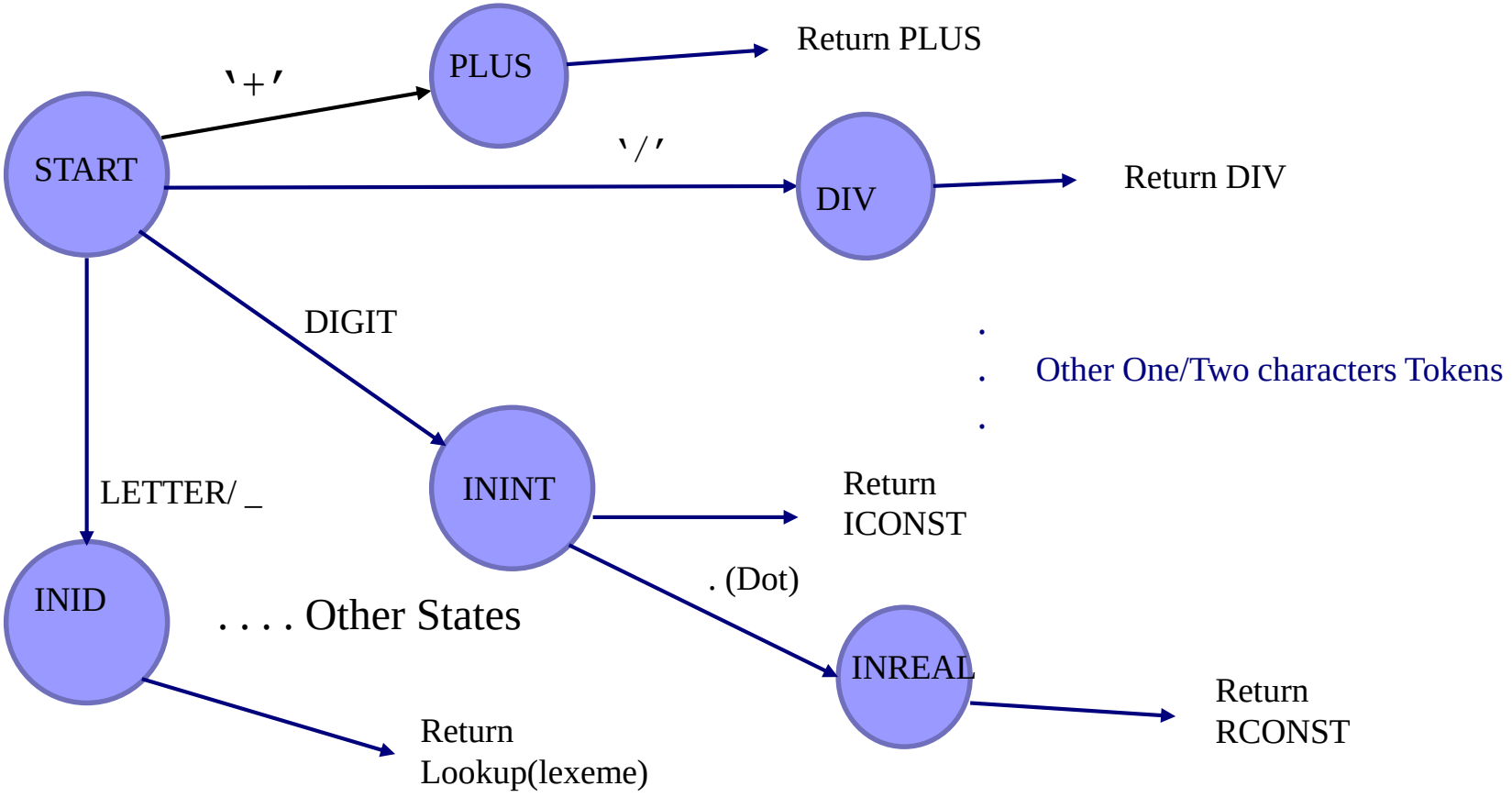
Implementation Issues (cont'd)

- Possible States:
  - START, INID, INSTRING, ININT . . .
  - 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, etc.
- Each state has different rules *inside* that state. For example,
  - Zero or more letters, digits, or underscores.
  - All characters following a single quote for a 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, . . .}
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
        WRITELN, IF, ELSE,
        // identifiers
        IDENT, NIDENT, SIDENT,
        // an integer, real, and string constant
        ICONST, RCONST, SCONST,
        // the numeric operators, assignment, numeric and
        // string comparison operators
        PLUS, MINUS, MULT, DIV, EXPONENT, ASSOP, NEQ,
        NGTHAN, NLTHAN, CAT, SREPEAT, SEQ, SLTHAN, SGTHAN,
        //Delimiters
        COMMA, SEMICOL, LPAREN, RPAREN, LBRACES, RBRACES,
        // 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 overloaded 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 one of the possible types of identifiers (i.e., IDENT, NIDENT, or SIDENT).

```
//a segment of the getNextToken code
LexItem getNextToken(istream& in, int& linenum){
  enum TokState { START, INID, ININT, . . .} lexstate = START;
  string lexeme;
  char ch;
  while(in.get(ch)) {
        switch( lexstate ) {
        case START:
              . . .
            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, -nconsts, -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 numeric 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 numeric constants are in the input?
  - What identifiers are in the input?

- You are provided by a set of 16 test cases associated with Programming Assignment 1. Vocareum automatic grading will be based on these testing files. These are available in compressed archive "PA 1 Test Cases.zip" on Canvas assignment.

Examples

- Example 1:
  - Input file: “realerr”

End of File →

```
23.5  15.  0.75
0.8  27.57.54 34.6
```

  - Output with –v –nconst flags:

```
RCONST(23.5)
RCONST(15.)
RCONST(0.75)
RCONST(0.8)
Error in line 2 (27.57.)
```

Examples

- Example 2: No set flags
  - Input file: “noflags”

```
$r = 50;
@flag = 'true';
if (flag -seq 'true') {
        $y_1 = 5;
}
else {
        $y_1 = (7.5);
}
writeln 'Value = ', $r, '$y_1 = ', $y_1;
```

End of File →

  - Output with no specified flags:

```
Lines: 10
Total Tokens: 39
Identifiers: 8
Numbers: 3
Strings: 3
```

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

```
23.5  15.25  4587
0.75  128  256
.8  0.47
```

End of File →

  - Output with –v –nconst flags:

```
RCONST(23.5)
RCONST(15.25)
ICONST(4587)
RCONST(0.75)
ICONST(128)
ICONST(256)
CAT
ICONST(8)
RCONST(0.47)

Lines: 4
Total Tokens: 9
Identifiers: 0
Numbers: 8
Strings: 0
NUMBERS:
0.47
0.75
8
15.25
23.5
128
256
4587
```

Examples

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

```
PROGRAM prog1
        # Testing all flags

        $x1 = .5; #numeric variable
        @_y1;   #string variable
        @str = 'Welcome' . @_y1;
        float z = 0.0 , _45, ___75

        r_25 = 50.;
END
```

  - Output with –ident flag:

```
Lines: 10
Total Tokens: 28
Identifiers: 11
Numbers: 3
Strings: 1
IDENTIFIERS:
$x1, @_y1, @str, END,
PROGRAM, _45, ___75,
float, prog1, r_25, z
```

Submission

- **Deadline: Sunday March 5, 2023**
  - 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 (March 5, 2023) are accepted with a fixed penalty of 25% from the student's score. No submission is accepted after Wednesday 11:59 pm, March 8, 2023.**


NJIT
THE EDGE IN KNOWLEDGE