UNIVERSITY AT ALBANY
State University of New York

COLLEGE OF ENGINEERING AND APPLIED SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

ICSI311 Principles of Programming Languages

Assignment 02 Created by Qi Wang

Table of Contents

Part I: General Information

- All assignments are individual assignments unless it is notified otherwise.
- All assignments must be submitted via Blackboard. No late submissions or e-mail submissions or hard copies will be accepted.
- Unlimited submission attempts will be allowed on Blackboard. **Only the last attempt will be graded.**
- <u>Work will be rejected with no credit if</u>
  - The work is late.
  - The work is not submitted properly (Blurry, wrong files, crashed files, files that can't open, etc.).
  - The work is a copy or partial copy of others' work (such as work from another person or the Internet).
- Students must turn in their original work. Any cheating violation will be reported to the college. Students can help others by sharing ideas and should not allow others to copy their work.
- Documents to be submitted:
  - **Java source file(s) with Javadoc style inline comments**
  - **Supporting files if any** (For example, files containing all testing data.)

  Note: Only the above-mentioned files are needed. Copy them into a folder, zip the folder, and submit the **zipped** file. We don't need other files from the project.
- Students are required to submit a design, all the error-free source files with Javadoc style inline comments and supporting files. Lack of any of the required items or programs with errors will result in a low credit or no credit.
- **Grades and feedback**: TAs will grade. Feedback and grades for properly submitted work will be posted on Blackboard. For questions regarding the feedback or the grade, students should reach out to their TAs first. Students have limited time/days from when a grade is posted to dispute the grade. Check email daily for the grade review notifications sent from the TAs. **Any grade dispute request after the dispute period will not be considered.**

Part II: Grading Rubric

The following includes, but not limited to, a list of performance indicators used for grading.

		Levels of Performance			
		UNSATISFACTORY	**DEVELOPING**	**SATISFACTORY**	**EXEMPLARY**
	Performance Indicator #1: Comments	None/Excessive	Some tags are correct. “What” not “Why”, few	Most tags are correct. Some “what” comments or missing some	All tags are correct. Anything not obvious has reasoning
		0		1	2
	Performance Indicator #2: Five methods for lexical analyzer	None		All methods present with minor issues. Most of the requirements are met.	All methods present without issues. All requirements are met.
		0		9	11
	Performance Indicator #3: Three methods for recursive-decent parser	None		All methods present with minor issues. Most of the requirements are met.	All methods present without issues. All requirements are met.
		0			15
	Performance Indicator #4: The driver	None		At least in one method and there are minor issues.	At least in one method and it is correct.
		0		1	2
Total Points Awarded		**Total points**: 30			
Feedback to the student					

Part III: Examples

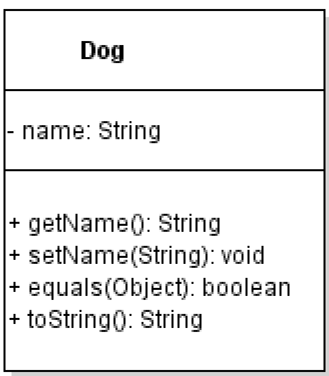
To complete a project, the following steps of a software development cycle should be followed. These steps are not pure linear but overlapped.

Analysis-design-code-test/debug-documentation.

1) Read project description to understand all specifications(**Analysis**)
2) Create a design (an algorithm or a UML class diagram) (**Design**)
3) Create programs that are translations of the design (**Code/Implementation**)
4) Test and debug(**test/debug)**
5) Complete all required documentation. (**Documentation**)

The following shows a sample design and the conventions.

- *Constructors* and *constants* should not be included in a class diagram.
- For each *field* (instance variable), include *visibility*, *name*, and *type* in the design.
- For each *method*, include *visibility*, *name*, *parameter type(s)* and *return type* in the design.
    - DON'T include *parameter names*, only *parameter types* are needed.
- Show class relationships such as *dependency*, *inheritance*, *aggregation*, etc. in the design. Don't include the *driver* program or any other testing classes since they are for testing purpose only.
    - Aggregation: For example, if Class A has an instance variable of type Class B, then, A is an aggregate of B.



The corresponding source codes with inline Javadoc comments are included on next page.

```java
import java.util.Random;

/**
 * Representing a dog with a name.
 * @author Qi Wang
 * @version 1.0
 */
public class Dog{
    /**
     * The name of this dog
     */
    private String name;

    /**
     * Constructs a newly created Dog object that represents a dog with an empty name.
     */
    public Dog(){
        this("");
    }
    /**
     * Constructs a newly created Dog object wit
     * @param name The name of this dog
     */
    public Dog(String name){
        this.name = name;
    }

    /**
     * Returns the name of this dog.
     * @return The name of this dog
     */
    public String getName(){
        return this.name;
    }

    /**
     * Changes the name of this dog.
     * @param name The name of this dog
     */
    public void setName(String name){
        this.name = name;
    }

    /**
     * Returns a string representation of this dog. The returned string contains the type of
     * this dog and the name of this dog.
     * @return A string representation of this dog
     */
    public String toString(){
        return this.getClass().getSimpleName() + ": " + this.name;
    }

    /**
     * Indicates if this dog is "equal to" some other object. If the other object is a dog,
     * this dog is equal to the other dog if they have the same names. If the other object is
     * not a dog, this dog is not equal to the other object.
     * @param obj A reference to some other object
     * @return A boolean value specifying if this dog is equal to some other object
     */
    public boolean equals(Object obj){
        //The specific object isn't a dog.
        if(!(obj instanceof Dog)){
            return false;
        }
        //The specific object is a dog.
        Dog other = (Dog)obj;
        return this.name.equalsIgnoreCase(other.name);
    }
}
```

TAB

TAB

TAB

TAB

open {

open {

Class comments must be written in Javadoc format before the class header. A **description** of the class, author information, and version information are required.

Comments for fields are required.

Method comments must be written in Javadoc format before the method header. The first word must be a verb in title case and in the **third** person. Use punctuation marks properly.

A description of the method, comments on parameters if any, and comments on the return type if any are required.

A Javadoc comment for a formal parameter consists of three parts:

- parameter tag,
- a name of the formal parameter in the design ,
  (The name must be consistent in the comments and the header.)
- and a phrase explaining what this parameter specifies.

A Javadoc comment for return type consists of two parts:

- return tag,
- and a phrase explaining what this returned value specifies

More inline comments can be included in single line or block comments format in a method.

Part IV: Description
A lexical analyzer and a syntax analyzer
Goals:

- Be familiar with techniques for lexical analysis and parsing

For this assignment, you will build a lexical analyzer and a syntax analyzer for simple arithmetic expressions and implement it in Java.

<u>Lexical Analyzer:</u>

A lexical analyzer is essentially a pattern matcher. A pattern matcher attempts to find a substring of a given string of characters that matches a given character pattern. Pattern matching is a traditional part of computing.

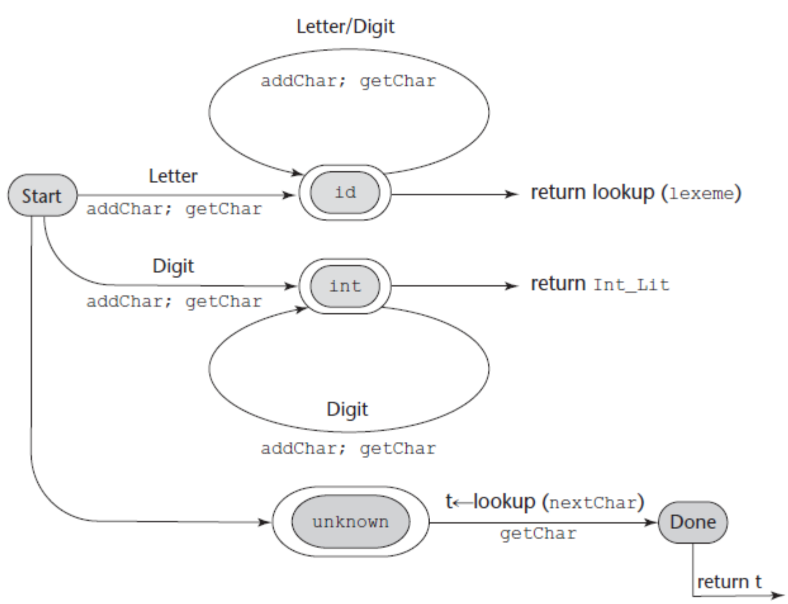
A lexical analyzer serves as the front end of a syntax analyzer. Technically, lexical analysis is a part of syntax analysis. A lexical analyzer performs syntax analysis at the lowest level of program structure.

An input program appears to a compiler as a single string of characters. The lexical analyzer collects characters into logical groupings(lexemes) and assigns internal codes to the groupings according to their structure(tokens). For example, the following shows the tokens and lexemes for expression `(sum + 47) / total`.

Token	Lexeme
LEFT_PAREN	(
IDENT	sum
ADD_OP	+
INT_LIT	47
LEFT_PAREN	)
DIV_OP	/
IDENT	total

Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens. The lexical-analysis process includes skipping comments and white space outside lexemes, as they are not relevant to the meaning of the program. Also, the lexical analyzer inserts lexemes for user-defined names into the symbol table, which is used by later phases of the compiler. Finally, lexical analyzers detect syntactic errors in tokens, such as ill-formed floating-point literals, and report such errors to the user.

Assume that a character can be a letter, a digit or something unknown such as parentheses, operators, etc. An identifier must begin with a letter and can have both letters and digits. A literal is an integer literal and can only have digits. The following state diagram shows the three patterns of accepted tokens by the lexical analyzer.



Here is one sample run:
Input program:

```
(sum + 47) / total
```

Output:

```
Next token is: 25 Next lexeme is (
Next token is: 11 Next lexeme is sum
Next token is: 21 Next lexeme is +
Next token is: 10 Next lexeme is 47
Next token is: 26 Next lexeme is )
Next token is: 24 Next lexeme is /
Next token is: 11 Next lexeme is total
Next token is: -1 Next lexeme is EOF
```

Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word). Use a table lookup to determine whether a possible identifier is in fact a reserved word. Modify the lexical analyzer to recognize the following list of reserved words and return their respective token codes:

- **for** (FOR_CODE, 30),
- **if** (IF_CODE, 31),
- **else** (ELSE_CODE, 32),
- **while** (WHILE_CODE, 33),
- **do** (DO_CODE, 34),
- **int** (INT_CODE, 35),
- **float** (FLOAT_CODE, 36),
- **switch** (SWITCH_CODE, 37).

You must test the lexical analyzer with at least 20 inputs. **Notice that it is required to build a lexical analyzer as a lexical analyzer/pattern matcher. It is not allowed to use any tokenizing class to generate tokens. Otherwise, the assignment will be returned with no credit.**

Syntax Analyzer(parser):

Use the same lexical analyzer, build a recursive-decent parser that constructs a parse tree by providing information required to build the parse tree. The following grammar for simple expressions is used.

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | ( <expr> )
```

You should write a subprogram for each nonterminal in the grammar. When given an input string, it traces out the parse tree that can be rooted at that nonterminal and whose leaves match the input string. A recursive-descent parsing subprogram is a parser for the language (set of strings) that is generated by its associated nonterminal. Notice that the parse begins by calling the lexical analyzer routine, `lex` , and the start symbol routine, in this case, `expr`. When `lex` is called, the next token code is returned. Here is one sample run:
Input program:

```
(sum + 47) / total
```

Output:

```
Next token is: 25, Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11, Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21, Next lexeme is +
Exit <factor>
```

```
Exit <term>
Next token is: 10, Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26, Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24, Next lexeme is /
Exit <factor>
Next token is: 11, Next lexeme is total
Enter <factor>
Next token is: -1, Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

You are required to test the parser with at least 20 inputs.

The following is the C program of the lexical analyzer from the textbook.

```
/* A lexical analyzer system for simple arithmetic expressions */
#include<stdio.h>
#include<ctype.h>

/* Global declarations */
/* Variables */
int charClass;
char lexeme[100];
char nextChar;
int lexLen;
int token;
int nextToken;
FILE *in_fp, *fopen ();

/* Function declarations */
void addChar();
void getChar();
void getNonBlank();
int lex();

/* Character classes */
#define LETTER 0
#define DIGIT 1
#define UNKNOWN 99

/* Token codes */
#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define SUB_OP 22
#define MULT_OP 23
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26

/******************************************************/
/* lookup - a function to lookup operators and parentheses and return the token */
int lookup(char ch){
  switch(ch){
```

```
    case '(':
      addChar();
      nextToken = LEFT_PAREN;
      break;
    case ')':
      addChar();
      nextToken = RIGHT_PAREN;
      break;
    case '+':
      addChar();
      nextToken = ADD_OP;
      break;
    case '-':
      addChar();
      nextToken = SUB_OP;
      break;
    case '*':
      addChar();
      nextToken = MULT_OP;
      break;
    case '/':
      addChar();
      nextToken = DIV_OP;
      break;
    default:
      addChar();
      nextToken = EOF;
      break;
    }
  return nextToken;
}

/*****************************************************/
/* addChar - a function to add nextChar to lexeme */
void addChar(){
  if(lexLen <= 98){
      lexeme[lexLen++] = nextChar;
      lexeme[lexLen] = 0;
  }
  else
    printf ("Error - lexeme is too long \n");
}

/*****************************************************/
/* getChar - a function to get the next character of input and determine its character
             class */
void getChar(){
  if ((nextChar = getc (in_fp)) != EOF){
      if (isalpha (nextChar))
            charClass = LETTER;
      else if (isdigit (nextChar))
            charClass = DIGIT;
      else
            charClass = UNKNOWN;
  }
  else
    charClass = EOF;
}


/*****************************************************/
/* getNonBlank - a function to call getChar until it returns a non- whitespace character
```

```
 */
void getNonBlank(){
  while (isspace (nextChar))
    getChar();
}

/*****************************************************/
/* lex - a simple lexical analyzer for arithmetic expressions */
int lex(){
  lexLen = 0;
  getNonBlank();
  switch (charClass){
    /* Parse identifiers */
    case LETTER:
      addChar();
      getChar();
      while(charClass == LETTER || charClass == DIGIT){
        addChar ();
        getChar ();
      }
      nextToken = IDENT;
      break;
    /* Parse integer literals */
    case DIGIT:
      addChar();
      getChar();
      while(charClass == DIGIT){
        addChar();
        getChar();
      }
      nextToken = INT_LIT;
      break;
    /* Parentheses and operators */
    case UNKNOWN:
      lookup(nextChar);
      getChar();
      break;
      /* EOF */
    case EOF:
      nextToken = EOF;
      lexeme[0] = 'E';
      lexeme[1] = 'O';
      lexeme[2] = 'F';
      lexeme[3] = 0;
      break;
    }

  printf ("Next token is: %d, Next lexeme is %s\n", nextToken, lexeme);
  return nextToken;
}

int main(){
  /* Open the input data file and process its contents */
  if((in_fp = fopen ("front.in", "r")) == NULL)
     printf ("ERROR - cannot open front.in \n");
  else{
      getChar ();
      do{
        lex ();
      }while (nextToken != EOF);
  }
```

```
  return 0;
}
```