National University of Singapore

CS4212 Project Assignment 1

AY 2019/2020 Semester 1

Due Date: Sunday, 15^{th} September 2019 (23:59 Hrs)

1 Introduction

In this assignment, you are required to construct a parse tree for a small object-based programming language called JLITE.

The syntax description of JLITE and a sample program are provided in Appendix A and B.

You are required to build the lexer and the parser for JLITE using `Java` tools, specifically, `JFlex` and `CUP`. At the end of the assignment, you are required to produce at least the following (please refer to the latter part of this assignment for more information):

1. An input `.flex` file to `JFlex` encoding the regular definitions and actions for tokens to be recognized for programs written in JLITE.
2. An input `.cup` file to `CUP` encoding the grammar specification for JLITE syntax.
3. Other Java files to handle parse trees generated by `CUP` and print out processed JLITE programs, e.g. adding toString() methods.

The parser must accept those *and only* those syntactically valid programs spelt out in the grammar specification, as shown in Appendix A and B.

We also provide two simple source JLITE programs so that you can test run your code. These two source JLITE programs are named `e.j` and `e1.j` . Their parsing outputs are called `e.out` and `e1.out` respectively. These are given in Appendix C - which you can cut and paste as needed.

Additional credit will be given for the usefulness of the errors reported to the user.

2 Resources

- JFlex: Home page
- CUP: Home page

3 Testing of Your Programs

You are required to create some sample programs to test your product.

4 Submission of your product

Please **submit a zipped item**, with your name and student ID as part of the zipfile's filename, containing the following documents to `LumiNUS CS4212` website under the "`Programming Assignment 1 Submission`" folder.

1. Your complete project and an instruction on how to run it. For example, you can make it as a Makefile like in our provided example.
2. Six (or more) sample programs that you have tried on your product, including at least two that *failed* to parse, and the output of these for judging the usefulness of the error reporting capability.
3. A document, named `xxxxxxx_pja1_readme.txt`, describing your product, the content of your submission, and any important information which you would like to share with us. Here, `xxxxxxx` stands for your matriculation number.

5 Late Submission

We try to discourage you from submitting your assignment beyond deadline. This is to ensure that you have time to prepare for other modules, as well as time for other assignments handed out in this module.

Any submission after the deadline will have its mark automatically deducted by certain percentages. Your submission time is determined by the time recorded in LumiNUS submission folders. If you have multiple submissions, we will take the latest submission time. Any submission to places other than the appropriate submission folders (such as to the instructor's email account) will be discarded and ignored.

Here is the marking scheme:

Submit by 23:59HRS of	Maximum Mark	If your score is ... %	It becomes ... %
16^{th} Sep	100	80	80
17^{th} Sep	80	80	64
18^{th} Sep	60	80	48
19^{st} Sep	0	-	0

A Specification of JLITE Syntax

A.1 Lexical Issues

- *id* ∈ **Identifiers**:

  An *identifier* is a sequence of alphabets, digits, and underscore, **starting with a lower letter**. Except the first letter, uppercase letters are not distinguished from lowercase.

- *cname* ∈ **Class Names**:

  A *class name* is a sequence of alphabets, digits, and underscore, **starting with an uppercase letter**. Except for the first letter, uppercase letters are not distinguished from lowercase.

- **Integer Literals**:

  A sequence of decimal digits (from 0 to 9) is an integer constant that denotes the corresponding integer value. Here, we use the symbol `INTEGER_LITERAL` to stand for an integer constant.

- **String Literals**:

  A string literal is the representation of a string value in an JLITE program. It is defined as a quoted sequence of ascii characters (Eg: ``` ``this is a string'' ```) where some constraints hold on the sequence of characters. Specifially, some special characters such as double quotes have to be represented in the string literal by preceding them with an escape character, backslash ("\"). More formally, a string literal is defined as a quoted sequence of: escaped sequences representing either special characters ($\backslash\backslash, \backslash n, \backslash r, \backslash t, \backslash b$) or the ascii value of an ascii character in decimal or hexadecimal base (e.g. $\backslash 032, \backslash x08$); characters excluding double quote, backslash, new-line or carriage return. Here, we use the symbol `STRING_LITERAL` to stand for any string constant.

- **Boolean Literals**:

  Believe it or not, there are only two boolean literals: `true` and `false`.

- **Binary Operators** :

  Binary operators are classified into several categories:

  1. **Boolean Operators** include conjunction and disjunction.
  2. **Relational Operators** are comparative operators over two integers
  3. **Arithmetic Operators** are those that perform arithmetic calculations.

  In addition to this categorization, each binary operator is associated with its own associativity rule; two distinct binary operators are related by a precedence relation.

- **Unary Operators** :

  There are only two unary operators:

  1. `!`. This is a negation operator, which aims to negate a Boolean value.

2. `-`. This is a negative operator, which aims to negate an integer value.

- **Class constructor** : There is **no** class constructor. Given the following declaration of a class, say `Box`,

```
class Box {
   Int x ;
   Int y ;
   Int z ;
   Box b1 ;
}
```

  The call `new Box()` will create an object instance, and initalize all its attributes, through *shallow* initialization. Thus, for the given example, the attributes are initialized as follows:

```
x = 0 ; y = 0 ; z = 0 ; b1 = null
```

- **Comments**: A *comment* may appear between any two tokens. There are two forms of comments: One starts with `/*`, ends with `*/`, and may run across multiple lines; another begins with `//` and goes to the end of the line.

B Grammar of JLITE

The grammar in BNF notation is provided in the following page.

⟨*Program*⟩ → ⟨*MainClass*⟩ ⟨*ClassDecl*⟩ *

⟨*MainClass*⟩ → `class` ⟨*cname*⟩ { `Void main (` ⟨*FmlList*⟩ `)` ⟨*MdBody*⟩ }

⟨*ClassDecl*⟩ → `class` ⟨*cname*⟩ { ⟨*VarDecl*⟩ * ⟨*MdDecl*⟩ * }

⟨*VarDecl*⟩ → ⟨*Type*⟩ ⟨*id*⟩ `;`

⟨*MdDecl*⟩ → ⟨*Type*⟩ ⟨*id*⟩ `(` ⟨*FmlList*⟩ `)` ⟨*MdBody*⟩

⟨*FmlList*⟩ → ⟨*Type*⟩ ⟨*id*⟩ ⟨*FmlRest*⟩ * | ϵ

⟨*FmlRest*⟩ → `,` ⟨*Type*⟩ ⟨*id*⟩

⟨*Type*⟩ → `Int` | `Bool` | `String` | `Void` | ⟨*cname*⟩

⟨*MdBody*⟩ → { ⟨*VarDecl*⟩ * ⟨ *Stmt*⟩ + }

⟨*Stmt*⟩ → `if (` ⟨*Exp*⟩ `)` { ⟨*Stmt*⟩ + } `else` { ⟨*Stmt*⟩ + }

| `while (` ⟨*Exp*⟩ `)` {⟨*Stmt*⟩ *}

| `readln (` ⟨*id*⟩ `) ;` | `println (` ⟨*Exp*⟩ `) ;`

| ⟨*id*⟩ `=` ⟨*Exp*⟩ `;` | ⟨*Atom*⟩`.`⟨*id*⟩ `=` ⟨*Exp*⟩ `;`

| ⟨*Atom*⟩ `(` ⟨*ExpList*⟩ `) ;` | `return` ⟨*Exp*⟩ `;` | `return ;`

⟨*Exp*⟩ → ⟨*BExp*⟩ | ⟨*AExp*⟩ | ⟨*SExp*⟩

⟨*BExp*⟩ → ⟨*BExp*⟩ `||` ⟨*Conj*⟩ | ⟨*Conj*⟩

⟨*Conj*⟩ → ⟨*Conj*⟩ `&&` ⟨*RExp*⟩ | ⟨*RExp*⟩

⟨*RExp*⟩ → ⟨*AExp*⟩ ⟨*BOp*⟩ ⟨*AExp*⟩ | ⟨*BGrd*⟩

⟨*BOp*⟩ → `<` | `>` | `<=` | `>=` | `==` | `!=`

⟨*BGrd*⟩ → `!`⟨*BGrd*⟩ | `true` | `false` | ⟨*Atom*⟩

⟨*AExp*⟩ → ⟨*AExp*⟩ `+` ⟨*Term*⟩ | ⟨*AExp*⟩ `-` ⟨*Term*⟩ | ⟨*Term*⟩

⟨*Term*⟩ → ⟨*Term*⟩ `*` ⟨*Ftr*⟩ | ⟨*Term*⟩ `/` ⟨*Ftr*⟩ | ⟨*Ftr*⟩

⟨*Ftr*⟩ → `INTEGER_LITERAL` | `-`⟨*Ftr*⟩ | ⟨*Atom*⟩

⟨*SExp*⟩ → `STRING_LITERAL` | ⟨*Atom*⟩

⟨*Atom*⟩ → ⟨*Atom*⟩`.`⟨*id*⟩ | ⟨*Atom*⟩`(` ⟨*ExpList*⟩ `)`

| `this` | ⟨*id*⟩ | `new` ⟨*cname*⟩`()`

| `(` ⟨*Exp*⟩ `)` | `null`

⟨*ExpList*⟩ → ⟨*Exp*⟩ ⟨*ExpRest*⟩ * | ϵ

⟨*ExpRest*⟩ → `,` ⟨*Exp*⟩

C Some Sample Program Runs

C.1 First Program

Following is a sample and yet meaningless program `e.j` that can be parsed by your system.

```
class Main {

Void main(Int i, Int a, Int b,Int d){

   while(true){

      b = 340 ;

      t1 = t2 ;
   }
}

}


class Dummy {
   Dummy j;

   Int dummy() {

      Bool i;

      Bool j;

      return i ;
   }

}
```

Following is a likely output produced from your code after parsing the above sample program:

```
class Main{
  void main(Int i,Int a,Int b,Int d){
    While(true)
    {
      b=340;
      t1=t2;
    }
  }

}

class Dummy{
  Dummy j;

  Int dummy(){
    Bool i;
    Bool j;
    Return i;
  }

}
```

C.2 Second Program

Following is second program `e1.j` which is equally senseless but complicated.

```
/* Mainly test multiple class (defined later but referenced first),
   Variable shadowing in Dummy class,
   chained field access expressions,
   e.g. this.getCompute().square(-3);
   Test combination of "if .. else .." "return" and "while"

 */

class Main {

Void main(Int i, Int a, Int b,Int d){

 Int t1;
 Int t2;

 Compute help;

 /*

 help = new Compute();

 help.chachedValue = t1 * 3;
```

```
 t1 = help.addSquares(a,b) + help.square(i);

 t2 = help.square(d);

 if(t2>t1){

    println("Square of d larger than sum of squares");

 }

 else{

      println("Square of d larger than sum of squares");

 }

 */

while(true){

//  t1 = 1*2;

 t1 = t2 ;

}
}
}


class Dummy {

Compute c;
Int i;
Dummy j;

Int dummy() {

Bool i;
Bool j;
  if (i || j) {
return 1;
    }
    else {
      while(i) {
        i = !j;
      }
c = this.getCompute();

    }
    return this.getCompute().square(-3);
    return i ;

}

Compute getCompute() {

   // c = new Compute();
   return c;
}
}
```

Following is a likely output produced from your code after parsing the above sample program:

```
class Main{
  void main(Int i,Int a,Int b,Int d){
    Int t1;
    Int t2;
    Compute help;
    While(true)
    {
      t1=t2;
    }
  }

}

class Dummy{
  Compute c;
  Int i;
  Dummy j;

  Int dummy(){
    Bool i;
    Bool j;
    If([i,j](||))
    {
      Return 1;
    }
    else
```

```
    {
      While(i)
      {
        i=(!)[j];
      }
      c=[this.getCompute()];
    }
    Return [[this.getCompute()].square((-)[3])];
    Return i;
  }


  Compute getCompute(){
    Return c;
  }

}
```