National University of Singapore

CS4212 Project Assignment 1

AY 2015/2015 Semester 1

Due Date: Thursday 13^{th} September 2015 (23:59 Hrs)

(Submission will be done electronically to IVLE Workbin)

1 Introduction

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

The syntax description of **mOOL** and a sample program are provided in Appendix A. Please examine them carefully.

A file directory (`READONLY`) is provided to you, containing necessary information and files for you to begin constructing the lexer and parser for **mOOL**. *Please duplicate a copy of this directory, and work on the duplicated copy to complete your construction of both the lexer and the parser.*

OCaml tools, `ocamllex` and `ocamlyacc`, will be used to construct your solutions. Specifically, in the `READONLY` directory you will find four special files:

1. `mOOL_main.ml` is the main program that links the lexer and parser together to perform the task of constructing parse trees for **mOOL** programs.
2. `mOOL_structs.ml` provides the **OCaml** data structures for parse trees of **mOOL** programs, and functions that perform pretty-printing of the parse tree (in the form of a program). **Please do not modify this file.**
3. `mOOL_lexer.mll` is the input file to `ocamllex` tool, and it contains *partial specification* of the tokens required by **mOOL**, as well as actions required to link to the parser.
4. `mOOL_parser.mly` is the input file to `ocamlyacc` tool, and it contains *partial specification* of the syntax required by **mOOL**, as well as actions required to build the final parse trees.

You can use the `make` utility to create the executable (named `mOOL_parser`) for the lexer and the parser as given in the directory (replicated by you). You can test this executable by running it against the sample **mOOL** program `e.txt` in the directory.

```
> mOOL_parser e.txt
```

In this assignment, you are required to complete the input files so that you can scan and parse **mOOL** programs written in its complete specification, through completing a number of tasks, as listed below:[1]

[1] You are encouraged to complete the tasks in this specific order. However, you may wish to complete them in whatever way you deem fit. But, you should not make changes to the `mOOL_structs.ml`, and the parse trees produced by your parser should be constructed from the data structures declared in `mOOL_structs.ml`.

Task 1: *Provide token specifications for string literals, and two kinds of* **mOOL** *comments.* This requires you to change the file `mOOL_lexer.mll`. The first kind of comment is marked by the two-character string: `//`, the second is marked by a pair of comment markers: `/*` and `*/`. In the case of providing comments using a pair of comment markers, you may wish to consider the possibility of allowing complex nested comments. such as the following:

```
/* ================================================
   This is the beginning of the program that evaluates the
   quality of responses as obtained from the social media.
   /* But who really care about the quality of these responses
      since when there isn't sufficient data, people will just
      listen to the responses and report their "finding as if
      they are the professionals
   */
   The evaluation will be done over multiple sets of data
   before making a summative statements about the quality.
================================================ */
```

Task 2: *Provide the grammar for* **mOOL** *expressions.* This requires you to change the file `mOOL_parser.mly`. Note that there are three kinds of expressions: the boolean, the relational and the arithmetic expressions. The grammar you provide in the file needs not follow exactly what's specified in the Specification as presented in the Appendix. But, you parser should be able to handle all of them. Also, try to get rid of as many conflicts as possible, especially the reduce/reduce conflicts.

Task 3: *Provide the token specifications and grammar for* **mOOL** *inheritance constructs..* This requires you to change both the files `mOOL_lexer.mll` and `mOOL_parser.mly`. The constructs to be considered are:

> the class inheritance construct `extends`, the modifier keyword `private`, the expression that refers to parent class `super`.

Note that, in the case when the modifier (or we call it access keyword in other part of the document) is not provided, the associated attribute is assumed to be *public*. If you read carefully the input file `mOOL_parser.mly`, you will notice the use of the term `public` in some of the actions provided.

Task 4: *Modify the class declaration to meet the* **mOOL** *specification.* This requires you to change the file `mOOL_parser.mly`. Notice that at this point, the class declaration only contains *one* variable declaration and *one* method declaration. In the actual specification, and can contains zero or more occurrences of variable and method declarations. Again, try to get rid of as many conflicts – especially the reduce/reduce conflicts – as possible.

At the end of the development, the lexer and the parser you develop must accept those *and only* those syntactically valid programs spelt out in the grammar specification, as shown in Appendix A.

At the end of this project assignment, you are required to submit a directory containing your code, and other relevant information. The details are as follows:

1. The directory name should be of the form `p1_xxxxxxx`, where `xxxxxxx` is your student id, such as `A1234567`.

2. The directory must include a `README` file, which includes details about what you have done to complete the tasks above, and how we should process your directory to assess your accomplishment.

3. The directory must include a sub-directory named `TESTCASES` containing the test cases which you have created for testing during development. It will be good to divide the test cases into sub-directories, each of which aims to test one of the development tasks above. Each task should have at least three interesting and distinctive test cases.

You may wish to access the folder named *Project Resources* in IVLE to jumpstart the development of your programs.

> **Please submit the directory in zipped format** to `IVLE CS4212` website under the "`Project Assignment 1 Submission`" folder.

2 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 IVLE 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 ... %
13th Sept (Sun)	100	80	80
14th Sept (Mon)	80	80	64
15h Sept (Tue)	60	80	48
After 15th (Thu)	0	-	0

A Specification of mOOL 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 a 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 components**: A class declaration comprises two (optional) components: *Attributes* and *Method declarations*. Both attributes and methods declared can have an optional access/modifier mode: *Private*. When the access mode *Private* is **not** present in a declaration for class, say C, we assume that the corresponding declared attribute or method in C can be access by the public. On the other hand, when the access mode *Private* is

present, the corresponding declared attribute or method *can only be access within the class C.* In another words, they **cannot be accessible by objects of other classes, not even the subclasses of the declared class C.**

- **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.

A.2 Some Sample Programs

```
// Program 1
class Factorial {
     void main (Int a) {
        println(new Fac().computeFac(a)) ;
        return ;
     }
}

class Fac {

/* This is the factorial function that is not written
   in tail-recursive manner. (A tail recursive factorial
   function is expected to take in two parameters.)
   Can your compiler optimize it to a tail-recursive function?

   Even if optimizing to tail-recursive form is beyond reach,
   could you attempt to optimize it so that it can run faster?
*/

     Int computeFac(Int num) {
        Int num_aux;

        if (num < 1)   { // shouldn't it be num <= 1?
           num_aux = 1 ;
```

```
        }
        else {
           num_aux = num * (this.computeFac(num-1)) ;
        }
        return num_aux ;
     }
}

// Program 2
class myDrawing {
    Rectangle rect;

    Void main () {
        Int theArea ;
        rect = new Rectangle() ;
        rect.setRect(5,7) ;
        theArea = rect.getArea() ;
        println (``The area is :'') ;
        println (theArea) ;
    }
}

class 2DShape {
    private Int width ;
    private Int height;

    Void setWidth(Int w) {
        width = w ;
   }
   Void setHeight(Int h) {
       height = h ;
   }
   Int getWidth() {
       return width ;
   }
   Int getHeight() {
       return height ;
   }
}

class Rectangle extends 2DShape {
    Int area ;

    Rectangle setRect(Int w, Int h) {
        setWidth(w) ;
        setHeight(h);
    }

    Int getArea() {
        return getWidth() * getHeight() ;
    }
}
```

A.3 Grammar

The grammar of **mOOL** in BNF notation is provided in the following page.

mOOL Syntax

$\langle Program \rangle \rightarrow \langle MainClass \rangle \ \langle ClassDecl \rangle^{\star}$

$\langle MainClass \rangle \rightarrow$ `class` $\langle cname \rangle$ { `Void main (` $\langle FmlList \rangle$ `)` $\langle MdBody \rangle$ }

$\langle ClassDecl \rangle \rightarrow$ `class` $\langle cname \rangle \ \langle Inherit \rangle$ { $\langle AttrDecl \rangle^{\star} \ \langle MdDecl \rangle^{\star}$ }

$\langle AttrDecl \rangle \rightarrow \langle Modifer \rangle \ \langle VarDecl \rangle$

$\langle VarDecl \rangle \rightarrow \langle Type \rangle \ \langle id \rangle$ `;`

$\langle MdDecl \rangle \rightarrow \langle Modifer \rangle \ \langle Type \rangle \ \langle id \rangle$ `(` $\langle FmlList \rangle$ `)` $\langle MdBody \rangle$

$\langle FmlList \rangle \rightarrow \langle Type \rangle \ \langle id \rangle \ \langle FmlRest \rangle^{\star} \mid \epsilon$

$\langle FmlRest \rangle \rightarrow$ `,` $\langle Type \rangle \ \langle id \rangle$

$\langle Inherit \rangle \rightarrow$ `extends` $\langle cname \rangle \mid \epsilon$

$\langle Modifier \rangle \rightarrow$ `private` $\mid \epsilon$

$\langle Type \rangle \rightarrow$ `Int` | `Bool` | `String` | `Void` | $\langle cname \rangle$

$\langle MdBody \rangle \rightarrow$ { $\langle VarDecl \rangle^{\star} \langle \ Stmt \rangle^{+}$ }

$\langle Stmt \rangle \rightarrow$ `if (` $\langle Exp \rangle$ `)` { $\langle Stmt \rangle^{+}$ } `else` { $\langle Stmt \rangle^{+}$ }

| `while (` $\langle Exp \rangle$ `)` {$\langle Stmt \rangle^{\star}$} | `readln (` $\langle id \rangle$ `) ;`

| `println (` $\langle Exp \rangle$ `) ;` | `return` $\langle Exp \rangle$ `;` | `return ;`

| $\langle id \rangle$ `=` $\langle Exp \rangle$ `;` | $\langle Atom \rangle$`.`$\langle id \rangle$ `=` $\langle Exp \rangle$ `;`

| $\langle Atom \rangle$`.`$\langle id \rangle$ `(` $\langle ExpList \rangle$ `) ;` | $\langle id \rangle$ `(` $\langle ExpList \rangle$ `) ;`

$\langle Exp \rangle \rightarrow \langle BExp \rangle \mid \langle AExp \rangle \mid \langle SExp \rangle$

$\langle BExp \rangle \rightarrow \langle BExp \rangle$ `||` $\langle Conj \rangle \mid \langle Conj \rangle$

$\langle Conj \rangle \rightarrow \langle Conj \rangle$ `&&` $\langle RExp \rangle \mid \langle RExp \rangle$

$\langle RExp \rangle \rightarrow \langle AExp \rangle \ \langle BOp \rangle \ \langle AExp \rangle \mid \langle BGrd \rangle$

$\langle BOp \rangle \rightarrow$ `<` | `>` | `<=` | `>=` | `==` | `!=`

$\langle BGrd \rangle \rightarrow$ `!`$\langle BGrd \rangle$ | `true` | `false` | $\langle Atom \rangle$

$\langle AExp \rangle \rightarrow \langle AExp \rangle$ `+` $\langle Term \rangle \mid \langle AExp \rangle$ `-` $\langle Term \rangle \mid \langle Term \rangle$

$\langle Term \rangle \rightarrow \langle Term \rangle$ `*` $\langle Ftr \rangle \mid \langle Term \rangle$ `/` $\langle Ftr \rangle \mid \langle Ftr \rangle$

$\langle Ftr \rangle \rightarrow$ `INTEGER_LITERAL` | `-`$\langle Ftr \rangle \mid \langle Atom \rangle$

$\langle SExp \rangle \rightarrow$ `STRING_LITERAL` | $\langle Atom \rangle$

$\langle Atom \rangle \rightarrow$ `(` $\langle cname \rangle$ `)` $\langle Atom \rangle \mid \langle Atom \rangle$`.`$\langle id \rangle \mid \langle id \rangle$

| $\langle Atom \rangle$`.`$\langle id \rangle$ `(` $\langle ExpList \rangle$ `)` | $\langle id \rangle$ `(` $\langle ExpList \rangle$ `)`

| `this` | `super` | `new` $\langle cname \rangle$`()` | `(` $\langle Exp \rangle$ `)` | `null`

$\langle ExpList \rangle \rightarrow \langle Exp \rangle \ \langle ExpRest \rangle^{\star} \mid \epsilon$

$\langle ExpRest \rangle \rightarrow$ `,` $\langle Exp \rangle$