National University of Singapore

CS4212 Project Assignment 1

AY 2021/2022 Semester 1

Due Date: Sunday, 19th September 2021 (23:59 Hrs)

1 Introduction

In this assignment, you are required to construct a parser for a small object-based programming language called JLITE. The syntax description of JLITE and sample programs are provided in Appendix A and B.

You will write the lexer and the parser for JLITE in `Python 3`. You are required to write both the lexer and parser *manually*, without using any lexer/parser generator tools. This is slightly more laborious than using automated tools, but gives a much deeper understanding of the concepts.

For the lexical analyzer (scanner), you may use the standard Python 3 regular expression library `re` if you wish, but this neither required nor recommended - it is arguably easier to write the token matching code manually. For parsing, you should implement a **recursive descent parser**.

The grammar given in Appendix B is (intentionally) *not* suited for recursive descent parsing, so you will need to rewrite this grammar into a suitable form before you can implement the parser. There are multiple ways to do this, as you will discover, some of which lead to efficient parsing, with no backtracking, and some of which lead to inefficient parsing. Your parser's execution performance is not graded, so it is up to you to decide whether to implement backtracking or rewrite the grammar so that no backtracking is needed.

The parser must accept those *and only* those syntactically valid programs spelt out in the grammar specification, as shown in Appendix A and B. The provided grammar is ambiguous, i.e., the same expression can produce multiple derivations. You will need to resolve this ambiguity in a sensible way. Note that there are errors, especially those involving type agreement, that can only be checked later.

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

We *highly recommend* that you annotate your Python classes and methods with PEP 526 type annotations, as demonstrated in `https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html`.

You can type check your annotated Python code using Mypy or using an IDE such as PyCharm. This helps catch many stupid errors early, so you don't waste precious time debugging uninteresting issues.

- Python 3: `https://www.python.org/`
- Mypy (type checker for Python): `https://mypy.readthedocs.io/en/stable/`
- PyCharm (IDE, can get Professional for free with NUS email): `https://www.jetbrains.com/pycharm/`

3 Testing of Your Programs

You are required to create and submit six sample programs to test your parser.

4 Submission of your product

Please **submit a ZIP archive**, with your name and student ID as part of the ZIP's filename, containing the following files to `LumiNUS CS4212` website under the "`Submissions for Project Assignment 1`" folder.

1. The zip file should at the top level contain a single folder bearing your name and student ID. In that folder should be your complete project sources, including test files and expected output for these tests. Please ensure that at the highest level is a single folder. This will greatly facilitate the bulk unzipping of the entire submission archive. Leaving all files in the top level will waste our time cleaning it up and organizing the unzipped archive.

2. We will test your submission using Python 3.8.10, so you should ensure your code works with this version without requiring any changes, or additional package installations. Your sources should include the following files:

   - A file named `lex.py` for the lexer.
     Invoking `./lex.py program.j` should print the lexer output (sequence of tokens) to stdout.
   - A file named `parse.py` for the parser.
     Invoking `./parse.py program.j` should print the parse tree to stdout. The format in which to print the parse tree is up to you, but it should have sufficient brackets to disambiguate, e.g., `1 + (2 * 3)` from `(1 + 2) * 3` on input `1 + 2 * 3`.
   - Optional: if you construct the *abstract syntax tree* (AST) in memory[1] rather than just print it at parse time, the classes for the AST should be in a file named `ast.py`.

3. A subfolder called `test/` within the main folder. This should include at least 6 pairs of files - *created by you* - to test your code. Each pair consisting of `TESTNAME.j` and `TESTNAME.j.gold`, where `TESTNAME.j` is the program to be parsed and `TESTNAME.j.gold` is the expected (correct) output from your parser on that file.

4. The main folder should also include a document named `[YourFullName]_pa1_readme.md` describing the design of your lexer and parser, the file structure of your submission (if it includes other files besides those described here), and any important information you would like to share with us.

   Please make sure to fully identify yourself in the submission. In previous years, there have been submissions with no identifying information. Marks will be deducted if that happens.

5 Late Submission

We try to discourage you from submitting your assignment after the 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 folder. 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 ... %
19^{th}	100	80	80
20^{th} Sep	80	80	64
21^{th} Sep	60	80	48
22^{st} Sep	0	-	0

A Specification of JLITE Syntax

A.1 Lexical Issues

- *id* ∈ **Identifiers**:

  An *identifier* is a sequence of letters, 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 letters, 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.

[1] You will *need* to do this for Project Assignment 2, so you might *want* to do it now.

- **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 (e.g.: `"this is a string"`) where some constraints hold on the sequence of characters. Specifically, 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. Note that `""` is the empty string.

- **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.
  4. **String Operator.** There is only one: '`+`'. If `s1` and `s2` are two strings, then '`s1 + s2`' produces the concatenation of `s1` followed by `s2`.

  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. JLITE follows the precedence rules of Java. See `https://introcs.cs.princeton.edu/java/11precedence/` where applicable (i.e., when the equivalent operator is present.)

- **Unary Operators** :

  There are only two unary operators:

  1. `!`. This is a negation operator. If v is `true`, then `!`v is `false`, and vice versa. If v is an integer, then `!`v is `true` if $v = 0$, and `!`v is `false`, otherwise. If v is a string, then `!`v is `true` if v is the empty string, otherwise it is `false`.
  2. `-`. This is a negative operator, that returns the integer value multiplied by -1.

- **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 initialize all its attributes to zeroes. Thus, for the given example, after the creation of the object, the attributes are (automatically) 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. Note that nested comments, i.e., 'comments within comments', are allowed. However, while any number of '`/`' that is greater than two will be deemed comments till the end of the line, starting from the first two `//`, comments using the `/*`, `*/` pair must *not* have an improperly paired `/*` or `*/` nested within.

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*⟩ → ⟨*SExp*⟩ `+` ⟨*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;
  }

}
```