# Exercise 3

Compilation 0368:3133

Due 6/12/2017

## 1 Introduction

We continue our journey of building a compiler for the invented object oriented language RioMare. Remember that the entire specification of RioMare appears inside the relevant folder of the course website. In order to make this document self contained, all the information needed to complete the third exercise is brought here again.

## 2 Programming Assignment

The third exercise implements a semantic analyzer that recursively scans the AST produced by CUP, and checks if it contains any semantic errors. The input for the semantic analyzer is a (single) text file containing a RioMare program, and the output is a (single) text file indicating whether the input program is semantically valid or not. In addition to that, whenever the input program is valid semantically, the semantic analyzer will add meta data to the abstract syntax tree, which is needed for later phases (code generation and optimization). The added meta data content will not be checked in exercises 3, but the best time to design and implement this addition is exercise 3.

## 3 The RioMare Semantics

This section describes the semantics of RioMare, and provides a multitude of legal and illegal example programs.

### 3.1 Types

The RioMare programming language defines two native types: integers and strings. In addition, it is possible to define a class by specifying its data members and methods. Also, given an existing type T, one can define an array of T's. Note, that defining classes and arrays is only possible in the uppermost (global) scope. The exact details follow.

**Arrays** can only be defined in the uppermost (global) scope. They are defined with respect to some existing type, as in the following example:

```
ARRAY IntArray EQ int[]
```

Defining an integer matrix, for example, is possible as follows:

```
ARRAY IntArray EQ int[]; ARRAY IntMat EQ IntArray[];
```

**Classes** contain data members and methods, and can only be defined in the uppermost (global) scope. They can refer to/extend only previously defined classes, to ensure that the class hierarchy has a tree structure. Following the same concept, method M1 can *not* refer to method M2, whenever M2 is defined after M1 in the class. In contrast to all that, method M *can* refer to a data member d, even if d is defined *after* M in the class. Table 1 summarizes these facts.

| 1 | `CLASS Son EXTENDS Father`<br>`{`<br>`    int bar;`<br>`}`<br>`CLASS Father`<br>`{`<br>`    void foo() { PrintInt(8); }`<br>`}` | ERROR |
|---|---|---|
| 3 | `CLASS Edge`<br>`{`<br>`    Vertex u;`<br>`    Vertex v;`<br>`}`<br>`CLASS Vertex`<br>`{`<br>`    int weight;`<br>`}` | ERROR |
| 2 | `CLASS UseBeforeDef`<br>`{`<br>`    void foo() { bar(8); }`<br>`    void bar(int i) { PrintInt(i); }`<br>`}` | ERROR |
| 1 | `CLASS UseBeforeDef`<br>`{`<br>`    void foo() { PrintInt(i); }`<br>`    int i;`<br>`}` | OK |

Table 1:

2

**Methods overloading** is *illegal* in RioMare, with the obvious exception of overriding a method in a derived class. So, example 1 in Table 2 is legal, and all other examples there are not. On a similar note, it is illegal for a class to have

| 1 | ``` CLASS Father { int foo(int i) { return 8; } } CLASS Son EXTENDS Father { int foo(int i) { return i; } } ``` | OK |
|---|---|---|
| 2 | ``` CLASS Father { int foo() { return 8; } } CLASS Son EXTENDS Father { void foo() { PrintInt(8); } } ``` | ERROR |
| 3 | ``` CLASS OverloadOverride { void foo() { PrintInt(8); } void foo(int i) { PrintInt(i); } } ``` | ERROR |
| 4 | ``` CLASS OverloadOverride { void foo() { PrintInt(8); } int foo() { return 8; } } ``` | ERROR |

Table 2: Method overloading is illegal in RioMare.

a variable with the same name as an existing method, or an existing variable. So, all examples in Table 3 are illegal.

## 3.2   Asignments

Binary operations are performed only between integers and strings. A variable is a recursive entity, represented as a sub tree in the AST. Assigning a value to a variable, Table 4 summarizes the context free grammar of RioMare. You will need to feed this grammar to CUP, and make sure there are no shift-reduce conflicts.

To create a graph visualization of the AST, please install graphviz and run

```
$ dot -Tjpeg -o ./AST_Graph.jpeg ./AST_Graph.txt
```

| | | |
|---|---|---|
| 3 | ```CLASS Father
{
    void foo() { PrintInt(8); }
}
CLASS Son EXTENDS Father
{
    int foo;
}``` | ERROR |
| 3 | ```CLASS Father
{
    string foo;
}
CLASS Son EXTENDS Father
{
    string foo;
}``` | ERROR |

Table 3: Variables can not have the same name as existing methods, or other variables.

from `EX5/LINUX_GCC_MAKE`

# 4  Input

The input for this exercise is a single text file, the input RioMare program.

# 5  Output

The output is a *single* text file that contains a *single* word. Either OK when the input program is correct semantically, or otherwise ERROR(*location*), where *location* is the line number of the *first* error that was encountered.

# 6  Submission Guidelines

The skeleton code for this exercise resides (as usual) in subdirectory EX3 of the course repository. COMPILATION/EX3 should contain a makefile building your source files to a runnable jar file called COMPILER (note the lack of the .jar suffix). Feel free to use the makefile supplied in the course repository, or write a new one if you want to. Before you submit, make sure that your exercise compiles and runs on the school server: *nova.cs.tau.ac.il*. This is the formal running environment of the course.

**Execution parameters**   compiler receives 2 input file names:

|          |      |                                                                                        |
|----------|------|----------------------------------------------------------------------------------------|
| Program  | ::=  | dec$^+$                                                                                 |

| dec | ::= | funcDec | varDec | classDec |
|-----|-----|--------------------------|

| varDec   | ::=  | ID ID [ ASSIGN exp ] ';'                                                               |
|----------|------|----------------------------------------------------------------------------------------|
| funcDec  | ::=  | ID ID ′(′ [ ID ID [ ',' ID ID ]* ] ′)′ ′{′ stmt [ stmt ]* ′}′                          |
| classDec | ::=  | CLASS ID [ EXTENDS ID ] ′{′ cField [ cField ]* ′}′                                      |

| exp | ::= | ′(′ exp ′)′ |
|-----|-----|-------------|
|     | ::= | exp BINOP exp |
|     | ::= | [ var '.' ] ID ′(′ [ exp [ ',' exp ]* ] ′)′ |
|     | ::= | INT | NIL | STRING | NEW ID | var |

| var | ::= | ID |
|-----|-----|-----|
|     | ::= | var '.' ID |
|     | ::= | var ′[′ exp ′]′ |

| stmt | ::= | varDec |
|------|-----|--------|
|      | ::= | var ASSIGN exp ';' |
|      | ::= | RETURN [ exp ] ';' |
|      | ::= | IF ′(′ exp ′)′ ′{′ stmt [ stmt ]* ′}′ |
|      | ::= | WHILE ′(′ exp ′)′ ′{′ stmt [ stmt ]* ′}′ |
|      | ::= | [ var '.' ] ID ′(′ [ exp [ ',' exp ]* ] ′)′ ';' |

| cField | ::= | varDec | funcDec |
|--------|-----|-------------------|

Table 4: Context free grammar for the RioMare programming language.

InputRioMareProgram.txt
OutputStatus.txt