Exercise 3

Compilation 0368:3133

Due 27/12/2017

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.

3.1.1 Classes

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, a method `M1` can *not* refer to a method `M2`, whenever `M2` is defined after `M1` in the class. In contrast to all that, a method `M` *can* refer to a data member `d`, even if `d` is defined *after* `M` in the class. Table 1 summarizes these facts.

<table>
<tr><td>1</td><td><pre>CLASS Son EXTENDS Father
{
    int bar;
}
CLASS Father
{
    void foo() { PrintInt(8); }
}</pre></td><td>ERROR</td></tr>
<tr><td>2</td><td><pre>CLASS Edge
{
    Vertex u;
    Vertex v;
}
CLASS Vertex
{
    int weight;
}</pre></td><td>ERROR</td></tr>
<tr><td>3</td><td><pre>CLASS UseBeforeDef
{
    void foo() { bar(8); }
    void bar(int i) { PrintInt(i); }
}</pre></td><td>ERROR</td></tr>
<tr><td>4</td><td><pre>CLASS UseBeforeDef
{
    void foo() { PrintInt(i); }
    int i;
}</pre></td><td>OK</td></tr>
</table>

Table 1: Referring to classes, methods and data members

Methods overloading is *illegal* in RioMare, with the obvious exception of overriding a method in a derived class. Similarly, it is illegal to define a variable with the same name of a previously defined variable (shadowing), or a previously defined method. Table 2 summarizes these facts.

<table>
<tr><td>1</td><td><pre>CLASS Father
{
    int foo() { return 8; }
}
CLASS Son EXTENDS Father
{
    void foo() { PrintInt(8); }
}</pre></td><td>ERROR</td></tr>
<tr><td>2</td><td><pre>CLASS Father
{
    int foo(int i) { return 8; }
}
CLASS Son EXTENDS Father
{
    int foo(int j) { return j; }
}</pre></td><td>OK</td></tr>
<tr><td>3</td><td><pre>CLASS IllegalSameName
{
    void foo() { PrintInt(8); }
    void foo(int i) { PrintInt(i); }
}</pre></td><td>ERROR</td></tr>
<tr><td>4</td><td><pre>CLASS Father
{
    int foo;
}
CLASS Son EXTENDS Father
{
    string foo;
}</pre></td><td>ERROR</td></tr>
</table>

Table 2: Method overloading and variable shadowing are both illegal in RioMare.

Inheritance if class `Son` is derived from class `Father`, then any place in the program that semantically allows an expression of type `Father`, should semantically allow an expression of type `Son`. For example,

`CLASS Father { int i; }`
`CLASS Son EXTENDS Father { int j; }`
`void foo(Father f) { PrintInt(f.i); }`
`void main(){ foo(new Son); }`
	OK

Table 3: new Son is a semantically valid input for foo.

Null expressions any place in the program that semantically allows an expression of type class, should semantically allow `NULL` instead. For instance,

`CLASS Father { int i; }`
`void foo(Father f){ PrintInt(f.i); }`
`void main(){ foo(NULL); }`
	OK

Table 4: NULL sent instead of a (Father) class is semantically allowed.

3.1.2 Arrays

Arrays can only be defined in the uppermost (global) scope. They are defined with respect to some previously defined 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[];
```

In addition, any place in the program that semantically allows an expression of type array, should semantically allow `NULL` instead. For instance,

`ARRAY IntArray EQ int[]`
`void F(IntArray A){ PrintInt(A[8]); }`
`void main(){ F(NULL); }`
	OK

Table 5: NULL sent instead of an integer array is semantically allowed.

Note that if two arrays of type `T` are defined, they are *not* interchangeable:

<table>
<tr><td><pre>ARRAY gradesArray = int[];
ARRAY IDsArray   = int[];
void F(IDsArray ids){ PrintInt(ids[6]); }
void main()
{
    IDsArray ids := new int[8];
    gradesArray grades := new int[8];
    F(grades);
}</pre></td><td>ERROR</td></tr>
</table>

Table 6: Non interchangeable array types.

3.2 Asignments

Assigning an expression to a variable is clearly legal whenever the two have the same type. In addition, following the concept in 3.1.1, if class `Son` is derived from class `Father`, then a variable of type `Father` can be assigned an expression of type `Son`. Furthermore, following the concept in 3.1.1 and 3.1.2, assigning `NULL` to array and class variables is legal. In contrast to that, assigning `NULL` to string variables is *illegal*. Table 9 summarizes these facts.

<table>
<tr><td>1</td><td><pre>CLASS Father
{
    int foo() { return 8; }
}
Father f := NULL;</pre></td><td>OK</td></tr>
<tr><td>2</td><td><pre>CLASS Father { ... }
CLASS Son EXTENDS Father { ... }
void foo(Father f) { ... }
Father f := new Son;
foo(new Son);</pre></td><td>OK</td></tr>
<tr><td>3</td><td><pre>ARRAY gradesArray = int[];
ARRAY IDsArray   = int[];
IDsArray := new int[8];
gradesArray := new int[8];
gradesArray := IDsArray;</pre></td><td>ERROR</td></tr>
<tr><td>4</td><td><pre>string s := NULL;</pre></td><td>ERROR</td></tr>
</table>

Table 7: Assignments in RioMare.

3.3 Equality Testing

Testing equality between two expressions is legal whenever the two have the same type. In addition, following the same reason in 3.2, if class `Son` is derived from class `Father`, then an expression of type `Father` can be tested for equality with an expression of type `Son`. Furthermore, any class variable, string variable or array variable can be tested for equality with `NULL`. The resulting type of a semantically valid comparison is the primitive type int. Table 9 summarizes these facts.

1	`CLASS Father`
	`{`
	`    int foo() { return 8; }`
	`}`
	`Father f := NULL;`
		OK

2	`CLASS Father { ... }`
	`CLASS Son EXTENDS Father { ... }`
	`void foo(Father f) { ... }`
	`Father f := new Son;`
	`foo(new Son);`
		OK

3	`ARRAY gradesArray = int[];`
	`ARRAY IDsArray    = int[];`
	`IDsArray := new int[8];`
	`gradesArray := new int[8];`
	`gradesArray := IDsArray;`
		ERROR

4	`string s := NULL;`	ERROR

Table 8: Assignments in RioMare.

3.4 Binary Operations

Most binary operations $(-, *, /, <, >)$ are performed only between integers. The single exception to that is the + binary operation, that can be performed between integers *and strings*. Note that equality testing (=) is treated differently, as discussed in 3.3. The resulting type of a semantically valid binary operation is the primitive type int. Table 9 summarizes these facts.

<table>
<tr><td>1</td><td><pre>CLASS Father
{
    int foo() { return 8; }
}
Father f := NULL;</pre></td><td>OK</td></tr>
<tr><td>2</td><td><pre>CLASS Father { ... }
CLASS Son EXTENDS Father { ... }
void foo(Father f) { ... }
Father f := new Son;
foo(new Son);</pre></td><td>OK</td></tr>
<tr><td>3</td><td><pre>ARRAY gradesArray = int[];
ARRAY IDsArray   = int[];
IDsArray := new int[8];
gradesArray := new int[8];
gradesArray := IDsArray;</pre></td><td>ERROR</td></tr>
<tr><td>4</td><td><pre>string s := NULL;</pre></td><td>ERROR</td></tr>
</table>

Table 9: Assignments in RioMare.

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

Program	::=	dec^+
dec	::=	funcDec | varDec | classDec | arrayDec
varDec	::=	ID ID [ASSIGN exp] ';'
funcDec	::=	ID ID '(' [ID ID [',' ID ID]*] ')' '{' stmt [stmt]* '}'
classDec	::=	CLASS ID [EXTENDS ID] '{' cField [cField]* '}'
arrayDec	::=	ARRAY ID = ID '[' ']'
exp	::=	var
	::=	'(' exp ')'
	::=	exp BINOP exp
	::=	[var '.'] ID '(' [exp [',' exp]*] ')'
	::=	['−'] INT | NIL | STRING | NEW ID | NEW ID '[' exp ']'
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
BINOP	::=	$+ \mid - \mid * \mid / \mid < \mid > \mid =$
INT	::=	$[1-9][0-9]^* \mid 0$

Table 10: Context free grammar for the RioMare programming language.

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:

InputRioMareProgram.txt
OutputStatus.txt