Exercise 4

Compilation 0368:3133

Due 24/1/2018

1 Introduction

Congratulations, you have made it to the final step of building an entire compiler for RioMare programs. 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 fourth exercise is brought here again.

2 Programming Assignment

The fourth (and last) exercise implements the code generation phase for RioMare programs. The chosen destination language this year is MIPS assembly, favoured for it straightforward syntax, complete toolchain and available tutorials. The exercise can be roughly divided into three parts as follows: (1) recursively traverse the AST to create an intermediate representation (IR) of the program. (2) Translate IR to MIPS instructions, but use an unbounded number of temporaries instead of registers. (3) Perform liveness analysis, build the interference graph, and allocate those hundreds (or so) temporaries into 8 physical registers. The input for this last exercise is a (single) text file, containing a RioMare program, and the output is a (single) text file that contains the translation of the input program into MIPS assembly.

3 The RioMare Semantics

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

3.1 Binary Operations

Recall that integers in RioMare are artificially bounded between -2^{15} and 2^{15}. The semantics of binary operations in RioMare is therefore somewhat different than that of standard programming languages. Table 1 uses a RioMare subscript to distinguish its operators from theie usual mathematical meaning.

		-2^{15}			$a * b$	$\leq$	-2^{15}
$a\ [*_{RioMare}]\ b$	$=$	$a * b$	-2^{15}	$\leq$	$a * b$	$\leq$	$2^{15}-1$
		$2^{15}-1$	$2^{15}-1$	$\leq$	$a * b$		
		-32768	$a * b \leq -32768$				
$a\ [*_{RioMare}]\ b$	$=$	$a * b$	$-32768 \leq a * b \leq 32767$				
		-32768	$a * b \leq -32768$				
		-32768	$a * b \leq -32768$				
$a\ [*_{RioMare}]\ b$	$=$	$a * b$	$-32768 \leq a * b \leq 32767$				
		-32768	$a * b \leq -32768$				
		-32768	$a * b \leq -32768$				
$a\ [*_{RioMare}]\ b$	$=$	$a * b$	$-32768 \leq a * b \leq 32767$				
		-32768	$a * b \leq -32768$				

Table 1: Binary Operations in RioMare

$$a\ [*_{RioMare}]\ b = \begin{array}{ll} a * b & a * b \in [-32768, 32767] \\ a * b & a * b \in [-32768, 32767] \end{array}$$

ioMare 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 2 summarizes these facts.

1	`CLASS Son EXTENDS Father`
	`{`
	`    int bar;`
	`}`
	`CLASS Father`
	`{`
	`    void foo() { PrintInt(8); }`
	`}`
		ERROR

2	`CLASS Edge`
	`{`
	`    Vertex u;`
	`    Vertex v;`
	`}`
	`CLASS Vertex`
	`{`
	`    int weight;`
	`}`
		ERROR

3	`CLASS UseBeforeDef`
	`{`
	`    void foo() { bar(8); }`
	`    void bar(int i) { PrintInt(i); }`
	`}`
		ERROR

4	`CLASS UseBeforeDef`
	`{`
	`    void foo() { PrintInt(i); }`
	`    int i;`
	`}`
		OK

Table 2: 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 3 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 3: 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,

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

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 = int[]
```

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

Table 4: new Son is a semantically valid input for foo.

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

Table 5: nil sent instead of a (Father) class is semantically allowed.

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

```
ARRAY IntArray = int[] ARRAY IntMat = IntArray[]
```

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

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

Table 6: nil sent instead of an integer array is semantically allowed.

Note that allocating arrays with the new operator must be done with an *integral size*. Similarly, accessing an array entry is semantically valid only when the *subscript expression has an integer type*. Note further that if two arrays of type `T` are defined, they are *not* interchangeable:

3.2 Assignments

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 `nil` to array and class variables is legal. In contrast to that, assigning `nil` to int and string variables is *illegal*. To avoid an overly complex semantics, we will enforce a strict policy of initializing data members inside classes: a declared data member inside a class can be initialized *only with a constant value* (that matches its type). Specifically, only constant integers, strings and `nil` can be used, and even a simple expression like $5 + 6$ is forbidden. Table 8 summarizes these facts.

`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);`
`}`
	ERROR

Table 7: Non interchangeable array types.

3.3 If and While Statements

The type of the condition inside if and while statements is the primitive type int.

3.4 Return Statements

According to the syntax of RioMare, return statements can only be found inside functions. Since functions can *not* be nested, it follows that a return statement belongs to *exactly one* function. when a function `foo` is declared to have a `void` return type, then all of its return statements must be *empty* (`return;`). In contrast, when a function `bar` has a non void return type `T`, then a return statement inside `bar` must be *non empty*, and the type of the returned expression must match `T`.

1	`CLASS Father { int i; }` `Father f := nil;`	OK

2	`CLASS Father { int i; }`
	`CLASS Son EXTENDS Father { int j; }`
	`Father f := NEW Son;`
		OK

3	`CLASS Father { int i; }` `CLASS Son EXTENDS Father { int j := 8; }`	OK
4	`CLASS Father { int i := 9; }` `CLASS Son EXTENDS Father { int j := i; }`	ERROR
5	`CLASS Father { int foo() { return 90; } }` `CLASS Son EXTENDS Father { int j := foo(); }`	ERROR

6	`CLASS IntList`
	`{`
	`    int head := -1;`
	`    IntList tail := NEW IntList;`
	`}`
		ERROR

7	`CLASS IntList`
	`{`
	`    void Init() { tail := NEW List; }`
	`    int head;`
	`    IntList tail;`
	`}`
		OK

8	`ARRAY gradesArray  = int[]`
	`ARRAY IDsArray     = int[]`
	`IDsArray    i := NEW int[8];`
	`gradesArray g := NEW int[8];`
	`i := g;`
		ERROR

9	`string s := nil;`	ERROR

Table 8: Assignments.

3.5 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 or array variable can be tested for equality with `nil`. But, in contrast, it is *illegal* to compare a string variable to `nil`. The resulting type of a semantically valid comparison is the primitive type int. Table 9 summarizes these facts.

1	`CLASS Father { int i; int j; }`
	`int Check(Father f)`
	`{`
	`    if (f = nil)`
	`    {`
	`        return 800;`
	`    }`
	`    return 774;`
	`}`
		OK

2	`int Check(string s)`
	`{`
	`    return s = “LosPollosHermanos”;`
	`}`
		OK

3	`ARRAY gradesArray = int[]`
	`ARRAY IDsArray    = int[]`
	`IDsArray   i:= NEW int[8];`
	`gradesArray g:=NEW int[8];`
	`int j := i = g;`
		ERROR

4	`string s1;`
	`string s2 := “HankSchrader”;`
	`int i := s1 = s2;`
		OK

Table 9: Equality testing.

3.6 Binary Operations

Most binary operations $(-, *, /, <, >)$ are performed only between integers. The single exception to that is the + binary operation, that can be performed between two integers or between two *strings*. The resulting type of a semantically valid binary operation is the primitive type int, with the single exception of adding two strings, where the resulting type is a string. Table 10 summarizes these facts.

1	`CLASS Father`
	`{`
	`    int foo() { return 8/0; }`
	`}`
		OK

2	`CLASS Father { string s1; string s2; }`
	`void foo(Father f)`
	`{`
	`    f.s1 := f.s1 + f.s2;`
	`}`
		OK

3	`CLASS Father { string s1; string s2; }`
	`void foo(Father f)`
	`{`
	`    int i := f.s1 < f.s2;`
	`}`
		ERROR

4	`CLASS Father { int j; int k; }`
	`int foo(Father f)`
	`{`
	`    int i := 620;`
	`    return i < f.j;`
	`}`
		OK

Table 10: Binary Operations.

3.7 Scope Rules

RioMare defines four kinds of scopes: block scopes of if and while statements, function scopes, class scopes and the outermost global scope. When an identifier is being used at some point in the program, its declaration is searched for in all of its enclosing scopes. The search starts from the innermost scope, and ends at the outermost (global) scope.

Note that array type declarations and class type declarations can only be defined in the outermost (global) scope. Class type names and array type names must be different than any previously defined variable names, function names, class type names and array type names.

Functions can be defined only in the class scopes, and the global scope. Following the same reason in 2, functions may only refer to previously defined types, variables and functions. When a function is being called inside a class scope, the declaration of a function with that name is searched first in its class scope. If no such function is found, the search moves to the global scope, and if the declaration is missing there too, a semantic error is issued. Following the same reason, when a function is being called inside the global scope, only the global scope is searched for its declaration.

Resolving a variable identifier follows the same principal, with the slight difference that variables can be declared in all four kinds of scopes. Table 11 summarizes these facts.

3.8 Library Functions

RioMare defines two library functions: PrintInt and PrintString.
The signatures of these functions are as follows:

```
void PrintInt(int i)       { ... }
void PrintString(string s){ ... }
```

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 the translation of the input program into MIPS assembly.

6 Submission Guidelines

The skeleton code for this exercise resides (as usual) in subdirectory EX4 of the course repository. COMPILATION/EX4 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:

InputRioMareProgram.txt
OutputMIPS.s

<table>
<tr><td>1</td><td><pre>int salary := 7800;
void foo()
{
    string salary := “6950”;
}</pre></td><td>OK</td></tr>
<tr><td>2</td><td><pre>int salary := 7800;
void foo(string salary)
{
    PrintString(salary)
}</pre></td><td>OK</td></tr>
<tr><td>3</td><td><pre>void foo(string salary)
{
    int salary := 7800;
    PrintString(salary)
}</pre></td><td>ERROR</td></tr>
<tr><td>4</td><td><pre>string myvar := “80”;
CLASS Father
{
    Father myvar := nil;
    void foo()
    {
        int myvar := 100;
        PrintInt(myvar);
    }
}</pre></td><td>OK</td></tr>
<tr><td>5</td><td><pre>int foo(string s) { return 800;}
CLASS Father
{
    string foo(string s)
    {
        return s;
    }
    void Print()
    {
        PrintString(foo(“Jerry”));
    }
}</pre></td><td>OK</td></tr>
</table>

Table 11: Scope Rules.

```
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     ::=  + | − | ∗ | / | < | > | =
INT       ::=  [1 − 9][0 − 9]* | 0
```

Table 12: Context free grammar for the RioMare programming language.

Precedence	Operator	Description	Associativity
1	$:=$	assign	
2	$=$	equals	left
3	$<, >$		left
4	$+, -$		left
5	$*, /$		left
6	[	array indexing	
7	(	function call	
8	.	field access	left

Table 13: Binary operators of RioMare along with their associativity and precedence. 1 stands for the lowest precedence, and 9 for the highest.