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 integer binary operations in RioMare is therefore somewhat different than that of standard programming languages. It is presented in Table 1, and to distinguish RioMare operators from the usual arithmetic signs, we shall use a RioMare subscript inside brackets: ($*_{[RioMare]}$, $+_{[RioMare]}$ etc.) Binary operations between strings include concatenation, and testing for equal-

$a \; *_{[RioMare]} \; b \; = \; \begin{cases} -2^{15} & \text{when } a * b \in (-\infty, -2^{15}] \\ a * b & \text{when } a * b \in (-2^{15}, 2^{15} - 1] \\ 2^{15} - 1 & \textit{otherwise} \end{cases}$
$a +_{[RioMare]} b \; = \; \begin{cases} -2^{15} & \text{when } a + b \in (-\infty, -2^{15}] \\ a + b & \text{when } a + b \in (-2^{15}, 2^{15} - 1] \\ 2^{15} - 1 & \textit{otherwise} \end{cases}$
$a -_{[RioMare]} b \; = \; \begin{cases} -2^{15} & \text{when } a - b \in (-\infty, -2^{15}] \\ a - b & \text{when } a - b \in (-2^{15}, 2^{15} - 1] \\ 2^{15} - 1 & \textit{otherwise} \end{cases}$
$a \;/_{[RioMare]}\; b \; = \; \begin{cases} -2^{15} & \text{when } \lfloor a/b \rfloor \in (-\infty, -2^{15}] \\ \lfloor a/b \rfloor & \text{when } \lfloor a/b \rfloor \in (-2^{15}, 2^{15} - 1] \\ 2^{15} - 1 & \textit{otherwise} \end{cases}$

Table 1: Semantics of RioMare binary operations between integers

ity. When concatenating two strings s_1 and s_2, we should allocate enough

3.2 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`.

3.3 Equality Testing

3.4 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 **??**, 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 2 summarizes these facts.

3.5 System Calls

MIPS supports a limited set of system calls, out of which we will need only four: printing an integer, printing a string, allocating heap memory and exit the program.

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

1	`int salary := 7800;`
	`void foo()`
	`{`
	`    string salary := “6950”;`
	`}`
		OK

2	`int salary := 7800;`
	`void foo(string salary)`
	`{`
	`    PrintString(salary)`
	`}`
		OK

3	`void foo(string salary)`
	`{`
	`    int salary := 7800;`
	`    PrintString(salary)`
	`}`
		ERROR

4	`string myvar := “80”;`
	`CLASS Father`
	`{`
	`    Father myvar := nil;`
	`    void foo()`
	`    {`
	`        int myvar := 100;`
	`        PrintInt(myvar);`
	`    }`
	`}`
		OK

5	`int foo(string s) { return 800;}`
	`CLASS Father`
	`{`
	`    string foo(string s)`
	`    {`
	`        return s;`
	`    }`
	`    void Print()`
	`    {`
	`        PrintString(foo(“Jerry”));`
	`    }`
	`}`
		OK

Table 2: 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	::=	$+ \mid - \mid * \mid / \mid < \mid > \mid =$
INT	::=	$[1-9][0-9]^* \mid 0$

Table 3: 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 4: Binary operators of RioMare along with their associativity and precedence. 1 stands for the lowest precedence, and 9 for the highest.