Exercise 4

Compilation 0368:3133

Due 20/1/2022 before 23:55

1 Introduction

Congratulations, you have made it to the final step of building an entire compiler for **L** programs. 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 **L** programs. The chosen destination language 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 **L** program, and the output is a (single) text file that contains the translation of the input program into MIPS assembly.

3 The L Semantics

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

3.1 Binary Operations

Integers in **L** are artificially bounded between -2^{15} and $2^{15}-1$. The semantics of integer binary operations in **L** is therefore somewhat different than that of standard programming languages. It is presented in Table 1, and to distinguish **L** operators from the usual arithmetic signs, we shall use a **L** subscript inside brackets: ($*_{[\mathbf{L}\ ]}$, $+_{[\mathbf{L}\ ]}$ etc.)

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

Table 1: Semantics of **L** binary operations between integers

Strings can be concatenated with binary operation $+$, and tested for (contents) equality with binary operator $=$. When concatenating two (null terminated) strings $\{s_i\}_{i=1}^{2}$, the resulting string $s_1 s_2$ is allocated on the heap, and should be null terminated. The result of testing contents equality is either 0 when they are equal, or 1 otherwise.

3.2 If and While Statements

While statements behave similar to (practically) all programming languages: before executing their body, their condition is evaluated. If it equals 0, the body is ignored, and control is transferred to the statement immediately after the body. Otherwise, the body is executed, then the condition is evaluated again, and so forth.

If statements behave similar to (practically) all programming languages: before executing their body, their condition is evaluated. If it equals 0, the body is ignored, and control is transferred to the statement immediately after the body. Otherwise, the body is executed exactly once, then control is transferred to the statement immediately after the body.

3.3 Function Calls and Order of Evaluation

When calling a function with more than one input parameter, the evaluation order matters. You should evaluate the sent parameters from left to right, so for example, the following code should print `32766`:

```
CLASS counter { int i := 32767; }
counter c := NEW counter;
int inc(){ c.i := c.i + 1; return 0;}
int dec(){ c.i := c.i - 1; return 9;}
int foo(int m, int n){ return c.i; }
void main()
{
    PrintInt(foo(inc(),dec()));
}
```

Table 2: Evaluation order of a called function's parameters matters.

When evaluating global variables order matters, and it should be the same as the order of appearance in the original program. Note that before entering main, all global variables with initialized values should be evaluated.

When evaluating assignments order matters, and the left hand side should be evaluated first. The same applies for evaluation of *all* binary operations.

Initializing class data members should occur during the construction of the object. The order in which you evaluate the data members initializations is irrelevant.

3.4 Runtime Checks

L enforces three kinds of runtime checks: division by zero, invalid pointer dereference and out of bound array access.

Division by zero should be handled by printing "Division By Zero", and then exit gracefully by using the exit system call. The following code will result in such behaviour:

```
int i:= 6; while (i+1) { int j := 8/i; i := i-1; }
```

Invalid pointer dereference can occur when trying to access data members or methods of uninitialized class variable. For example, here:

```
CLASS Father { int i; int j; } Father f; int i := f.i;
```

Similarly, assigning `NIL` to `f` should clearly trigger the same behaviour:

```
CLASS Father { int i; int j; } Father f := NIL; int i := f.i;
```

When an invalid pointer dereference occurs, the program should print "Invalid Pointer Dereference" and then then exit gracefully by using the exit system call.

Out of bound array access should be handled by printing "Access Violation" and then exit gracefully by using the exit system call. The following code demonstrates an illegal array access:

```
ARRAY IntArray = int[] IntArray A := NEW int[6]; int i := A[18];
```

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.

system call example	MIPS code	Remarks

`PrintInt(17)`	`li $a0,17`
	`li $v0,1`
	`syscall`	

`string s:="abc"`
`PrintString(s)`
	`.data`
	`myLovelyStr: .asciiz "abc"`
	`.text`
	`main:`
	`la $a0,myLovelyStr`
	`li $v0,4`
	`syscall`
		Printed string must be
		null terminated.
		It can be allocated
		inside the text section,
		or in the heap.

`Malloc(17)`	`li $a0,17`
	`li $v0,9`
	`syscall`
		allocated address
		is returned in `$v0`

`Exit()`	`li $v0,10` `syscall`	Make sure every MIPS program ends with exit.

Table 3: Relevant MIPS system calls.

3.6 Program entry point (main)

Every (valid) **L** program has a main function with signature: `void main()`. This function is the entry point of execution (see also 3.3).

Program	::=	dec$^+$
dec	::=	varDec | funcDec | classDec | arrayDec
varDec	::=	ID ID [ASSIGN exp] ';'
	::=	ID ID ASSIGN newExp ';'
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
var	::=	ID
	::=	var '.' ID
	::=	var '[' exp ']'
stmt	::=	varDec
	::=	var ASSIGN exp ';'
	::=	var ASSIGN newExp ';'
	::=	RETURN [exp] ';'
	::=	IF '(' exp ')' '{' stmt [stmt]* '}'
	::=	WHILE '(' exp ')' '{' stmt [stmt]* '}'
	::=	[var '.'] ID '(' [exp [',' exp]*] ')' ';'
newExp	::=	NEW ID | NEW ID '[' exp ']'
cField	::=	varDec | funcDec
BINOP	::=	+ | − | ∗ | / | < | > | =
INT	::=	$[1-9][0-9]^*$ | 0

Table 4: Context free grammar for the **L** programming language.

4 Input

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

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 5: Binary operators of **L** along with their associativity and precedence. 1 stands for the lowest precedence, and 9 for the highest.

5 Output

The output is a single text file that contains the translation of the input program into MIPS assembly.

6 Skeleton

To run the skeleton use the following command:

- make everything

This will create a file with the compiled MIPS code (MIPS.txt), and a file with the output of SPIM (MIPS_OUTPUT.txt).

7 SPIM

We are going to use SPIM 8.0 in the project. Can be installed using:

- sudo apt-get install spim xspim

8 Submission Guidelines

The skeleton code for this exercise resides (as usual) in subdirectory EX4 of the course repository (`https://github.com/davidtr1037/COMPILATION_TAU_FOR_STUDENTS/tree/master/FOLDER_3_SOURCE_CODE/EX4`). 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:

InputProgram.txt
OutputMIPS.s

9 Additional Notes

9.1 PrintInt

When printing an integer, print an additional space at the end.