Exercise 4 (Code Generation)

(for students in the *Iron Swords 28* group)

Due 25/5/2023 before 23:59

1 Introduction

In this exercise, you are going to finish the implementation of your compiler from **L** to MIPS. This exercise is designed especially for students in the *Iron Swords 28* group who wish to do a home exam and get a regular (0..100) grade. Thus, this exercise should be done individually by every student. (Obviously, you can use the code your group wrote in the previous exercises.) In your final grade, the (up to) 60 points you were supposed to get according to your grade in the class exam will be replaced by (up to) 30 points for your home exam and (up to) 30 points for this exercise.

2 Programming Assignment

In this exercise, you are asked to implement the code generation phase in the compiler 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. Perform liveness analysis, build the interference graph, and allocate those hundreds (or so) temporaries into 10 physical registers (t0-t9).
3. Translate the IR to MIPS instructions using physical registers instead of temporaries.

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 Input Programs

3.1 Syntax

In this exercise, you should handle the entire **L** programming language. (Recall that in exercise 4 (dataflow) you were allowed to assume that the input program is written in a simple subset of **L** .)

3.2 The L Semantics

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

3.2.1 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.)

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 1 when they are equal, or 0 otherwise.

Arrays Testing equality of arrays should be done by comparing the address values of the two arrays. The result is 1 if they are equal and otherwise 0.

Objects Testing equality of objects should be done by comparing the address values of the two objects. The result is 1 if they are equal and otherwise 0.

3.2.2 If and While Statements

While statements behave similarly 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 similarly 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.

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

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

3.2.3 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`:[1]

When initializing 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 to the evaluation of *all* binary operations.

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

[1] Recall Table 1 and that $2^{15} - 1 = 32767$.

```
class counter { int i := 32767; }
counter c := nil;
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()
{
    c := new counter;
    PrintInt(foo(inc(),dec()));
}
```

Figure 1: Evaluation order of a called function's parameters matters.

3.2.4 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 "Illegal 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];
```

3.2.5 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 2: Relevant MIPS system calls.

3.2.6 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 3.2.3).

4 Input

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

5 Output

The output is a single text file that contains the translation of the input program into MIPS assembly or an error message indicating the failure of register allocation.

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 for this exercise can be found here. The makefile must be located in the following path:

- ex4-mips/Makefile

This makefile will build the compiler (a runnable jar file) in the following path:

- ex4-mips/COMPILER

Feel free to reuse the makefile supplied in the skeleton, or write a new one if you want to.

8.1 Command-line usage

COMPILER receives 2 parameters (file paths):

- *input* (input file path)
- *output* (output file path containing the MIPS code)

8.2 Skeleton

You are encouraged to use the makefile provided by the skeleton.

8.2.1 Compiling

To build the skeleton, run the following command (in the *src/ex4-mips* directory):
`$ make compile`
This performs the following steps:

- Generates the relevant files using jflex/cup
- Compiles the modules into *COMPILER*

9 Additional Notes

9.1 Global Variables

You may assume that if a global variable is initialized, then the initial value is a constant (i.e., string, integer, nil).

9.2 PrintInt

When printing an integer, print an additional space after the integer.

9.3 Register Allocation

Note that you are performing register allocation on the IR. This is OK since the IR is rather low-level–all operations are performed between temporaries (virtual registers) and global and local variables are accessed via loads and stores. Thus, you should allocate registers only to temporaries. You should only allocate registers t0-t9.

You should implement only simplification-based register allocation. Specifically, you are *not* required to implement register spilling or MOV instruction coalescing. If you fail to allocate registers due to the need to perform an actual spill, the compiler should output an error message "Register Allocation Failed" and abort.

9.4 MIPS Code Generation

The translation from IR commands to MIPS commands is rather straightforward. If the translation requires the use of registers beyond those allocated to the temporaries in the IR command (as may happen in array accesses and method calls), you can use registers s0-s9.

Note that registers t0-t9 are caller-saved, and registers s0-s9 are callee-saved. Due to the simple usage of s registers in this exercise, you are not required to save and restore callee-saved registers.