

# Exercise 5

Compilation 0368-3133

Due 16/3/2025 (Moed B) before 23:59

## 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 fifth exercise is brought here again.

## 2 Programming Assignment

The fifth (and last) exercise implements the code generation phase for **L** programs. The chosen destination language is MIPS assembly, favored for its straightforward syntax, complete toolchain and available tutorials. The exercise can be roughly divided into three parts as follows:

- Recursively traverse the AST to create an intermediate representation (IR) of the program.
- Translate IR to MIPS instructions, but use an unbounded number of temporaries instead of registers.
- Perform liveness analysis, build the interference graph, and allocate those hundreds (or so) temporaries into 10 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** 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 ( $*_{[L]}$ ,  ${}_{[L]}$ , 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  $s_1s_2$  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.

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

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

**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 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.

### 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 code in Figure 1 should print 32766.<sup>1</sup>

---

<sup>1</sup>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.

**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.

### 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 “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 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
<code>PrintInt(17)</code>	<code>li \$a0,17 li \$v0,1 syscall</code>	
<code>string s:="abc" PrintString(s)</code>	<code>.data myLovelyStr: .asciiz "abc" .text main: la \$a0,myLovelyStr li \$v0,4 syscall</code>	Printed string must be null terminated. It can be allocated inside the text section, or in the heap.
<code>Malloc(17)</code>	<code>li \$a0,17 li \$v0,9 syscall</code>	allocated address is returned in <code>\$v0</code>
<code>Exit()</code>	<code>li \$v0,10 syscall</code>	Make sure every MIPS program ends with exit.

Table 2: 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).

## 4 Additional Guidelines

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

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

**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.

**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.

## 5 Input

The input for this exercise is a single text file, the input **L** program. In this exercise, you should handle the entire **L** programming language. (Recall that in exercise 4 you were allowed to assume that the input program is written in a simple subset of **L**.) You can assume that the input program has no lexical, syntax, or semantic errors and is valid.

## 6 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: “Register Allocation Failed”.

## 7 SPIM

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

```
$ sudo apt-get install spim
```

Note that SPIM is already pre-installed in the course’s Docker container on the school server.

## 8 Submission Guidelines

The skeleton for this exercise can be found here:

<https://github.com/noa-schiller/compilation-tau/tree/main/skeletons/ex5>.

The makefile must be located in the following path:

- ex5/Makefile

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

- ex5/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. Some files of interest in the provided skeleton:

- *src/MIPS/MIPSGenerator.java* (contains translations of IR commands to MIPS commands)
- *examples* contain two simple **L** programs and their corresponding MIPS translations.
- *input* and *expected\_output* contain input tests and their corresponding expected results. Note that the expected output files contain the results generated by SPIM with the compiled MIPS code as input.

As in previous exercises, you need to use *your own* files from previous exercises.

### 8.2.1 Compiling

To build the skeleton, run the following command (in the *ex5* directory):

```
$ make compile
```

This performs the following steps:

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

To also run *COMPILER* on input/Input.txt run the following command (in the *ex5* directory):

```
$ make everything
```

This performs the following steps:

- Generates the relevant files using jflex/cup
- Compiles the modules into *COMPILER*
- Runs *COMPILER* on input/Input.txt and creates a file with the compiled MIPS code (MIPS.txt)
- Runs the MIPS program using SPIM and creates a file with the program output (MIPS\_OUTPUT.txt)