Exercise 5 – Code Generation

Compilation 0368-3133

Due 15/3/2026 before 23:59

1 Assignment Overview

Congratulations! You have reached the final step in building a complete compiler for **L** programs. The fifth and final 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.
- Perform liveness analysis, build the interference graph, and allocate those hundreds (or so) temporaries into 10 physical registers.
- Translate the IR into MIPS instructions.

The input for this final exercise is a single text file containing a **L** program. The output is a single text file containing the corresponding MIPS assembly translation.

2 The L Runtime Behavior

This section describes how **L** programs execute.

2.1 Binary Operations

Integers: 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. **L** uses *saturation arithmetic*: results exceeding the range are "clamped" to the nearest boundary. For any operator $\odot \in \{+,-,*,/\}$ (where / denotes integer floor division $\lfloor a/b \rfloor$), let $\odot_{[\mathbf{L}]}$ denote the corresponding operator in **L**. The operation is defined as follows:

$$a \odot_{[\mathbf{L}]} b = \begin{cases} -2^{15} & \text{if } (a \odot b) \leq -2^{15} \\ 2^{15}-1 & \text{if } (a \odot b) \geq 2^{15}-1 \\ a \odot b & \text{otherwise} \end{cases}$$

Strings: 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.

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

2.2 If and While Statements

If and while statements in **L** behave similarly to (practically) all programming languages.

Control Flow: Both constructs evaluate an integer condition, where 0 represents *false* and any non-zero value represents *true*.

While statements: The condition is evaluated before every iteration. If the condition is non-zero, the body is executed, and the process repeats. If the condition is 0, the loop terminates, and control passes to the statement immediately following the loop body.

If statements: The condition is evaluated first. If the result is non-zero, the body of the if is executed. If the result is 0, the body of the else is executed (if one is present). Upon completion of either path, control continues to the statement immediately following the entire structure.

2.3 Program Execution and Evaluation Order

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

Function argument 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`.[1]

```
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.

Global variables initialization: When initializing global variables, the order matters and should match their order of appearance in the original program. As in the fourth exercise, before entering main, *all* global variables with initialized values should be evaluated. Note that global variables may be initialized using *arbitrary non-constant expressions*.

Binary operations and assignments: For all binary operations (including assignment), the left-hand side should be evaluated first.

[1]Recall that integer operations use saturation arithmetic, and that the maximum value is $2^{15} - 1 = 32767$.

Class data members initialization: Occurs during object construction. The specific order in which data members are initialized is irrelevant. You may assume that data members may be initialized *only with constant literals* (specifically integers, strings, and `nil`).

2.4 Library Functions

You can assume that the names `PrintInt` and `PrintString` are never redefined. Consequently, calling these functions always invokes the corresponding **L** library functions. Both functions must be implemented using the corresponding MIPS system calls. Specifically:

- `PrintInt` prints the integer argument followed by a single space.
- `PrintString` prints the string argument.

(Note: The implementation of `PrintInt`, including the trailing space, is provided in the starter code).

2.5 Runtime Checks

L enforces three kinds of runtime checks: division by zero, invalid pointer dereference, and out-of-bounds array access. In all cases, the program must print a specific error message and exit gracefully (using the exit system call).

Division by zero: Must be detected when a denominator evaluates to 0.

- **Error message:** `Illegal Division By Zero`
- **Example:**

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

Invalid pointer dereference: Occurs when attempting to access a field, method, or array element of a *class* or *array* variable that is uninitialized or holding the value `nil`.

- **Error message:** `Invalid Pointer Dereference`
- **Examples:**

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

Out of bound array access: Occurs when the accessed index falls outside the valid boundaries of the array. That is, when the index is negative or greater than or equal to the length of the array.

- **Error message:** `Access Violation`
- **Example:**

```
array IntArray = int[]; IntArray A := new int[6]; A[18];
```

2.6 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. The specific codes and arguments are detailed in Table 1.

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 $v0,9`
	`syscall`
		The allocated address is returned in `$v0`. Note that the allocation size is specified in bytes.

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

Table 1: Relevant MIPS system calls.

3 Additional Guidelines

Uninitialized variables: You are not required to perform the uninitialized variable analysis from the previous exercise. Accessing an uninitialized variable results in undefined behavior. Hence, we will not check scenarios where the behavior of the program depends on uninitialized variables.

Register allocation: The register allocation is performed on the IR. Since the IR is low-level, performing operations on virtual registers (temporaries) and accessing variables via loads/stores, register allocation applies strictly 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 the allocator fails to color the graph (meaning a spill would be necessary), the compiler must print `Register Allocation Failed` and terminate.

Important: We will check specific cases where the allocation should fail. Since the number of physical registers required depends on the specific translation to IR and the subsequent translation from IR to MIPS, you must follow the translation scheme provided in the tutorial to guarantee predictable behavior. You are provided with one specific test case where allocation fails because the number of function arguments exceeds the limit of available physical registers.

MIPS code generation: The translation from IR commands to MIPS instructions is generally straightforward. However, 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`.

4 Input and Output

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**.)

Output: The output is a single text file containing exactly one of the following:

- `ERROR`
  If a lexical error is detected.
- `ERROR(`*location*`)`
  If a syntax or semantic error is detected. The *location* is the line number of the *first* error encountered.
- `Register Allocation Failed`
  If the register allocation fails.
- **The MIPS assembly translation of the input program**
  If no errors occurred.

5 SPIM

The project uses the SPIM 8.0 simulator. To use it outside the school server, you may need to install it:

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

Note that your compiler must output a text file containing the translated MIPS instructions. We will grade the assignment by running this output file directly in SPIM.

6 Submission Guidelines

Each group should have **only one** student submit the solution via the course Moodle. Submit all your code in a single zip file named `<ID>.zip`, where `<ID>` is the ID of the submitting student. The zip file must have the following structure at the top level:

1. A text file named `ids.txt` containing the IDs of all team members (one ID per line).
2. A folder named `ex5/` containing all your source code.

Inside the `ex5` folder, include a makefile at `ex5/Makefile`. This makefile must build your source files into the compiler, which should be a runnable jar file located at `ex5/COMPILER` (note the lack of .jar suffix). You may reuse the makefile provided in the skeleton, or create a new one if you prefer.

Command-line usage: `COMPILER` receives 2 parameters (file paths):

- `input` (input file path)
- `output` (output file path containing the expected output)

Self-check script: Before submitting, you *must* run the self-check script provided with this exercise. This script verifies that your source code compiles successfully using your makefile and passes several provided tests. You must execute the self-check on your submission zip file within the school server (`nova.cs.tau.ac.il`) and ensure that your submission passes all checks. Follow these steps:

1. Download `self-check-ex5.zip` from the course Moodle and place it in the same directory as your submission zip file. Make sure no other files are present in that directory.
2. Run the following commands:
   ```
   unzip self-check-ex5.zip
   python self-check.py
   ```

Make sure that your `ids.txt` file follows the required format, and that your makefile does *not* contain hard-coded paths. The self-check script does **not** verify these aspects automatically and you are responsible for ensuring that your code compiles correctly in any directory.

Note: The script fails if any provided test fails. If the failure is not due to the output format and you cannot pass all tests by the submission date, it is still acceptable to submit.

Submissions that fail to compile or do not produce the required runnable will receive a grade of 0, and resubmissions will not be allowed.

7 Starter Code

The exercise skeleton can be found in the repository for this exercise. The skeleton provides a basic translation of the IR to MIPS instructions, which you may modify as needed. Some files of interest in the provided skeleton:

- `src/ir/*.java` (classes for the IR commands with an additional `mipsMe` method that contains the translation to MIPS)
- `src/mips/MipsGenerator.java` (contains methods to print MIPS instructions to the output)
- `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.

Building and running the skeleton: From the `ex5` directory, run:

```
$ make
```

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` and execute the program using SPIM, run:

```
$ make everything
```

This performs the following additional steps:

- Runs `COMPILER` on `input/Input.txt` to generate the MIPS assembly file (`MIPS.txt`).
  (Note: The provided skeleton code is currently hardcoded to write the MIPS output to `MIPS.txt`, regardless of the command-line arguments).
- Runs the MIPS program using SPIM and creates a file with the program output (`MIPS_OUTPUT.txt`).

Note: the steps performed in everything mode should not be performed when running `make` using your own makefile.