CS 241 — Spring 2020 — Assignment 3

Assignments for CS 241

← Assignment 2	Assignment 3	Assignment 4 ↓
~~Friday, May 29th at 5:00 pm~~ Sunday, May 31st at 5:00 pm	**Friday, June 5th at 5:00 pm**	Friday, June 12th at 5:00 pm

P1 • P2 • P3 • P4

Assignments 3 and 4 may be done in either Racket or C++14. See language-specific notes for each option at the end of this document.

In Assignments 3 and 4, you will incrementally write an assembler for MIPS assembly language (CS241 dialect).

Note carefully: In order to do Assignment 4, you must do Assignment 3 first. We will not be distributing a solution to Assignment 3 for you to use as a starting point for Assignment 4.

Reminder: For this and future assignments, be sure to run the command `source /u/cs241/setup` to gain access to the CS 241 tools.

Marmoset Notes

For each question, Marmoset has public and release tests, and often has secret tests as well. The weights assigned to each question are given on the assignment.

When you release test a submission, Marmoset will only report on public and release tests. No information about secret tests will be visible until after the assignment deadline.

This means the number of marks you see on Marmoset (prior to the assignment deadline) will not necessarily match the number of marks shown on the assignment page.

Part I. Tree Parsing

Problem 1 — 15 marks of 66 (filename: `traverse.rkt` or `traverse.cc`)

Write a Racket or C++ program that reads a pre-order traversal of a non-empty tree from standard input and prints the corresponding post-order traversal for that tree. Each line of both the input and output will consist of two non-negative integers:

```
<NODE-VALUE> <NUMBER-OF-CHILDREN>
```

for example the following input:

```
1 3
2 2
5 0
6 0
3 1
7 0
4 4
```

```
8 0
9 0
10 0
11 1
12 0
```

corresponds to this tree.

and the output of the program given the above input would be:

```
5 0
6 0
2 2
7 0
3 1
8 0
9 0
10 0
12 0
11 1
4 4
1 3
```

Test your program with different input.

You must solve this problem by constructing a tree from the given pre-order traversal, and then performing a post-order traversal on this tree. We reserve the right to hand-check that you have done this.

Submit a file called `traverse.rkt` or `traverse.cc` containing the Racket or C++ source code for your program.

Click here to return to the top of Assignment 3.

Part II. Writing an Assembler

For the remaining problems in assignment 3 and assignment 4 you will implement an assembler for progressively larger subsets of MIPS assembly language. Subject to the assumptions stated in the problems, your assembler must report all errors, and must correctly translate all correct assembly language programs to MIPS machine language.

We have provided a scanner (also called a tokenizer) for MIPS assembly language for each available language option (see language-specific notes). You should use this scanner as a starting point for your assembler.

Each problem in Part II requires you to submit a program that reads from standard input and writes to standard output as well as standard error. **The input and output specifications are identical regardless of which language you choose.** The only difference is that you must submit the appropriate `.rkt` or `.cc` file depending on your choice of language.

For each problem, we ask you to implement support for additional instructions. You may submit the same assembler for all the problems. We encourage you to submit to Marmoset early. As soon as you implement support for the instructions specified by a problem, submit the current version of your assembler to Marmoset. That way, if you do not complete all of the problems before the deadline, you will still get credit for those that you did complete.

Hint: Depending on the design decisions you make in your solutions to problems 2 and 3, you may have to restructure your code to get a working solution to problem 4. Therefore, you may want to read and understand all of the problems (especially up to and including problem 4) before beginning problem 2. However, if you find this overwhelming, you may find it easier to just focus on solving problems 2 and 3 first, and deal with problem 4 when you come to it. The decision is yours.

Problem 2 — 17 marks of 66 (filename: `asm.rkt` or `asm.cc`)

Begin by writing an assembler that correctly translates input containing no labels and no instructions other than `.word`. You may assume that the input to your assembler contains no labels and no instructions other than `.word`.

Your assembler should never crash, even if the input is not a valid assembly language program. Your assembler should not silently ignore errors in the input program. If the input contains a line that is not valid in MIPS assembly language, your assembler should print an appropriate error message containing the word ERROR in all capitals to **standard error** and stop. It is good practice, but not a requirement, to embed ERROR within a meaningful error message.

Hint: there are relatively few ways in which an assembly language program can be valid (and all the valid forms are spelled out here), but many ways in which it can be invalid. You will find it much easier to write code that looks for valid input and rejects everything unexpected, rather than code that explicitly looks for all the different ways in which the input could be invalid.

If the input contains a correct MIPS assembly language program, your assembler should output the equivalent MIPS machine language to standard output.

Click here to return to the top of Assignment 3.

Problem 3 — 17 marks of 66 (filename: `asm.rkt` or `asm.cc`)

Add support for label definitions to your assembler. Other than the inclusion of label definitions, the restrictions, assumptions and output requirements (including error-checking) stated in problem 2 apply to problem 3.

In addition, if the input is a correct MIPS assembly program, your assembler should output a symbol table: a listing of the names and values of all defined labels to standard error. The list should be printed on several lines, one line for each label in the input. Each line should consist of the label (without the trailing colon), followed by a space, followed by the value of the label (in decimal). The labels may appear in the symbol table in any order. For example, the following input:

```
begin: .word 2
middle: .word 0
.word 0
end:
```

Should print the following to stderr (but possibly with the lines reordered):

```
begin 0
middle 4
end 12
```

In handling labels, you may use any data structure or data structures you choose, but be sure to take efficiency into account.

Click here to return to the top of Assignment 3.

Problem 4 — 17 marks of 66 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to allow labels to be defined and also to be used as operands.

Other than the inclusion of label definitions and labels as operands, the restrictions, assumptions, and output requirements (including error-checking) stated in problem 2 apply to problem 4. (Note that you need not list the names and values of defined labels as in problem 3.)

Click here to return to the top of Assignment 3.

Assignment 4

Assignments for CS 241

↑ Assignment 3	Assignment 4	Assignment 5→
Friday, June 5th at 5:00 pm	**Friday, June 12th at 5:00 pm**	Monday, June 22nd at 5:00 pm

P1 • P2 • P3 • P4 • P5 • P6 • P7

Note: The restrictions, assumptions, and output requirements (including error-checking) as stated in Assignment 3 apply throughout Assignment 4 as well. In addition, your solution for each problem should continue to be a correct solution for each problem that came before it (for example, a correct solution of A4P3 will also meet the requirements of A4P1 and A4P2).

Problem 1 — 9 marks of 60 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle `jr` and `jalr` instructions.

Click here to return to the top of Assignment 4.

Problem 2 — 9 marks of 60 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle `add`, `sub`, `slt`, and `sltu` instructions.

Click here to return to the top of Assignment 4.

Problem 3 — 9 marks of 60 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle `beq` and `bne` instructions with an integer or hex constant as the branch offset.

Click here to return to the top of Assignment 4.

Problem 4 — 9 marks of 60 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle `beq` and `bne` instructions with a label as the branch target operand.

Click here to return to the top of Assignment 4.

Problem 5 — 8 marks of 60 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle the `lis`, `mflo`, and `mfhi` instructions.

Click here to return to the top of Assignment 4.

Problem 6 — 8 marks of 60 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle the `mult`, `multu`, `div`, and `divu` instructions.

Click here to return to the top of Assignment 4.

Problem 7 — 8 marks of 60 (filename: `asm.rkt` or `asm.cc`)

Modify your assembler to correctly handle the `sw` and `lw` instructions.

Your assembler should correctly translate any MIPS assembly language program, and write ERROR to standard error for any input that is not a valid MIPS assembly program.

Click here to return to the top of Assignment 4.

Language-Specific Details

Racket

The provided starter `asm_rkt.zip` has a function called `scan` that takes as input a string and returns a list of tokens.

The Using Racket in CS 241 document contains hints and techniques for using Racket to write the assembler. See also the comments in the provided scanner.

Run a Racket program using the command: `racket asm.rkt`

Click here to return to the top of the page.

C++

The provided starter `asm_cpp.zip` has a method called `assemble` that is called with a vector of lines containing tokens.

When submitting to Marmoset, if you have chosen C++, you will need to add all your files to a `.zip` file and submit that to Marmoset. The top level directory of your `.zip` file must contain the `asm.cc` file. For example, if your zip file contains a directory called `a3` and the `asm.cc` file is stored under this directory, Marmoset will not be able to find the file.

The STL Quick Reference for CS 241 document outlines the parts of the STL most likely to be of use in CS 241.

You are **strongly** advised to check for memory-related errors by vetting your programs with Valgrind. To do

this, run "valgrind *program optionsAndArguments*" instead of just "*program optionsAndArguments*". **Marmoset will run your submissions with Valgrind as well, and will reject any submission that contains memory-related errors.** Be aware that running Valgrind increases the execution time of your program by a factor of 5 to 20.

See the following page on Debugging Valgrind Errors for a discussion of common Valgrind errors and how to resolve them.

Compile a program in C++ using the command "`g++-6 -g -std=c++14 -lm -Wl,--warn-common,--fatal-warnings -o asm asm.cc scanner.cc`".

This command will create a file called `asm` containing the compiled code.

Run the program using the command: `./asm`

Click here to return to the top of the page.