CS202 Assignment 1:
Building a Language Processor in C^{++}

Benjamin Cisneros

1 Description

The purpose of this assignment is to help refresh your memory of C^{++} programming and get your mind back into thinking logically and critically. The objective of this assignment is to build an assembler and interpreter (a virtual machine) that simulates the assembly language for a hypothetical processor using everything you learned in CS135. Take into account that there are better ways to do this, but you will not learn them until about a month into CS202. Finally, in order to get high marks, your program must compile and run on code grade; that is, your program should work correctly and follow all instructions.

2 Background

Low-level machine programs (like the ones you will write in CS218) are rarely written by humans as they are typically generated by compilers or interpreters. However, by looking at the translated code, we can gain valuable insight on how to improve high-level programs by avoiding low-level problems and better utilizing the underlying hardware. (Therefore, do not be alarmed if you ever see assembly code in any of the C^{++} programs you bump into.) For a language like C^{++}, for instance, one of the key players in this translation process is the compiler − a program designed to translate code written in a high-level language into code written in binary machine language (this is your object code or the famous `a.out` file). Here you will build an assembler for translation and an array-based interpreter for execution.

An **assembler** is a program that reads an assembly language and translates the code into binary machine language. The translation procedure includes the removal of comments and the conversion of labels into proper addresses as required by the processor for execution. An **interpreter**, on the other hand, is a program that translates a program written in one language (often a high-level language) into an *intermediate representation*, sometimes a form of *instruction set* for efficient execution, that itself is run on the hardware. For a typical interpreter, we can distinguish four main phases (such as scanning, parsing, semantic analysis and intermediate representation, and evaluation), but we will focus mainly on just one − **evaluation**.

Although there are typically numerous phases (or passes) involved in translating code, we will keep things simple for this assignment. You will revisit all of this in a far more elegant way in CS460 compilers course by Dr. Matt Pedersen.

3 Task

The assembler's job is simply to do a translation in two passes. In the **first pass**, lexical and syntactic analysis is carried out using field splitting. This involves reading the assembly language, removing comments, allocating a memory location for each label, and writing an intermediate representation of operations and operands into a temporary file. The **second pass** reads the temporary file, translates the symbolic operands to the memory locations calculated in the first pass, encodes the operations and operands, and then stores the resulting machine language program in a stack (we will simulate one using an array of integers). The other component of the task is to construct an interpreter for a processor that mimics the operations of a computer while running programs written in assembly language. For example, an instruction is retrieved from the stack (our array) by the interpreter, which then breaks it down into an operator and an operand before simulating the instruction. The program counter is kept in the variable *ip* (the instruction pointer). It should be said that the associative array stores label positions in memory. The value at index *i* is set to nothing if an input line has no labels. Thus, when debugging your program, it is OK to see random values but make sure to bypass their addresses in order to avoid undefined behaviors.

4 Behavior

There are three ways to describe the desire behavior of your assembler: (1) when loaded into your assembler, an "assembly" file containing a valid assembly language should be parse appropriately before writing the translated code to a file. (2) The output produced by the assembler must be identical to the output produced by the supplied solution in code grade. (3) Your assembler must implement the translation specification given below. Remember that the job of your interpreter is to execute a number of instructions (which are specified in Table 1). If the generated program from the second pass is wrong, your interpreter will − and ultimately *should* − generate an error.

Table 1. Assembly language instructions

Opcode	Instruction	Meaning
00	`const`	defines either a constant value or a location in memory
01	`get`	reads a numbers from the input
02	`put`	writes the contents to the output
03	`ld X`	loads register with contents of memory location X
04	`st X`	stores contents of register in memory location X
05	`add X`	adds contents of location X to register
06	`sub X`	subtract contents of location X from register
07	`mul X`	multiplies contents of location X with register
08	`div X`	divides contents of location X by register
09	`cmp X`	compares cotents of location X with register
10	`jpos X`	jumps to location X if register is positive
11	`jz X`	jumps to location X if register is zero
12	`j X`	jumps to location X
13	`jl X`	jumps to location X if register less than zero
14	`jle X`	jumps to location X if register is less than or equal to zero
15	`jg X`	jumps to location X if register is greater than zero
16	`jge X`	jumps to location X if register is greater than or equal to zero
17	`halt`	stops execution

4.1 Translation Specification

We will assume that every computer in the world has just one register called *r* (which is depressing but also makes this assignment simpler), 18 instructions, and a 1000-word word-addressable memory which happens to be the size of the stack. As a result, we can only stored 1000 things at a time in our array. We will also assume that a *word* of computer memory has five decimal digits, with the first two encoding the operation and the last three serving as the address if the word is an instruction. For example, consider the following snippet of assembly code generated by the assembler during the second pass (note that this sample output is for debugging purposes only)

```
03015  loopInit  ld  one
04017            st  i
12006            j   loopTest
03017  ...
```

If we look at the first command `ld one`, we can see that its address begins with the operation or the instruction `ld` (in this case, and in accordance with Table 1, the opcode for `ld` should be 03) and terminates with the address of label `one`. The address of this label (among other things) can be seen in the listing below

```
Symbols
> [loopInit] = 0
> ...
> [out] = 14
> [one] = 15
> [ten] = 16
> [i] = 17
```

This is a table of symbols (also an array) that was created during the first pass of the assembler. It should be clear that we can also determine the location of label `i` using the information listed above, and the opcode for the instruction `st` using Table 1. Consequently, the address for the command `st one` should be 04017. Now, how do we compute or calculate these addresses? Typically, we would use object-oriented traits in C^{++} to create a stack that holds “different” values (independent of what each value actually represents in memory). Then, whenever we need to put something onto the stack or remove something from it, we would increment or decrement *ip* by one without considering the type of values we are inserting or removing. For example, the command `ld X` involves the following operations

input	stack	
`ld one`	10	← top value
`ld ten`	1	

Therefore, entering the expression `ld <num>` will simply push *num* onto the stack. Note that the input is read from top to bottom (↓) while the stack is build from the bottom up (↑). This is the reason why 1 appears at bottom, and 10 at the top of the stack. Unfortunately, we are unable to apply any object-oriented traits at this time, so we must find a simple but effective solution. What we will do instead is to use the address of the labels along with the opcodes' mnemonic values as follows

```
memory[nextMemory++] = STACK_SIZE * opcodes[<instruction>] + labels[<label>]
```

where *instruction* is any opcode value from Table 1, and *label* is any label that appears in the assembly file. For example, for the command `ld one`, the address is computed as follows

$$\begin{aligned}\text{memory location} &= 1000 \times 3 + 15\\ &= 3015\end{aligned}$$

where 1000 is the size of our array, 3 represents the `ld` instruction, and 15 is the index where the value `one` can be found. Now, in order to obtain the address of label `one`, we would mod 3015 (the memory location) by the stack size as follows

$$\begin{aligned}\text{label address} &= 3015 \ \% \ 1000\\ &= 15\end{aligned}$$

and in order to obtain the opcode value, we would simple divide 3015 by the stack size

$$\begin{aligned}\text{opcode value} &= 3015 \ / \ 1000\\ &= 3\end{aligned}$$

Clearly, we can get the first one by combining the last two

memory location	
3015	
opcode value	label address
03	015

It is important to keep in mind that each memory location *must* be computed during the second pass of the assembler. The interpreter is responsible for computing the last two since they are only needed during the interpretation of the code. Let us take a look at the above assembly program example once more but in more detailed.

What you see is an illustration of the assembly program as a table with the three columns: the

Table 2. Assembly language illustration

index	memory location	encode instruction		
0:	03015	`loopInit`	`ld`	`one`
1:	04017		`st`	`i`
2:	12006		`j`	`loopTest`
3:	03017	`...`		
...:	`...`	`...`		
15:	00001	`one`	`const`	`1`
16:	00010	`ten`	`const`	`10`
17:	00000	`i`	`const`	

index of each *memory address*, the *memory* location, and finally, the *encoded instruction*. Memory location at index 0, for example, contains the translation of the first instruction of the assembly language program – that is, `ld one`. (We are going to ignore the `loopInit` label for this short example.) As you can see from the previous explanation, label `one`'s address is 015 (or just 15). Therefore, every time we execute a load instruction, we need to grab the address of the label that follows the instruction in order to store the value that the label points to in memory in register r. To put it simply, we must transfer values back and forth between memory and register. Of course, this operation moves relatively slowly since we only have one register, namely r, available. To make up for this penalty, we will permit direct memory access for all types of arithmetic instructions (such as `add`, `sub`, `div`, etc.) and the comparison instruction `cmp` as long as r has a value.

Take the `add` instruction as an example. Adding two operands requires loading r with the value of the first operand (from some address in memory) and accessing the value of the second operand directly from a different address in memory.

```
ld   i
add  i
```

Note that while we are using `i` twice, only a single copy of `i`'s value is assigned to r, that is, the command `ld i` is equivalent to the assignment $r = i$. This is best depicted as follows, where *addr* is the label address of `i`.

input		equivalent to		interpretation
`ld   i`	→	$r = i$	→	$r = memory[addr]$
`add  i`		$r = r + i$		$r = r + memory[addr]$

Additionally, note that the value of `i` is never updated. This means that moving the value from register r to memory is the only way to accomplish this. Therefore, if we wanted to update the value of `i`, we would need to execute the following operations

input		equivalent to		interpretation
`ld   i`		$r = i$		$r = memory[addr]$
`add  i`	→	$r = r + i$	→	$r = r + memory[addr]$
`st   i`		$i = r$		$memory[addr] = r$

So far we have no considered code that branches. Let us consider the following C^{++} snippet of code:

```
if (x > 10) {
   /* statement here..  */
}
```

The above **if** statement code could be interpreted as

```
      ld   x
      cmp  ten
      jl   done   ; if x <= 10
      ...         ; if x > 10
done  ...
```

Note that one way to skip the statement "guarded" by the conditional expression $x > 10$ – that is, `/* statement here.. */` – is to force a jump if the expression is false or simply fall through if it is not. But how should the above assembly code be interpreted? The important thing to pay attention to here is the `cmp` instruction. This instruction will *always* appear before any jump instruction (refer to Table 1 as we have a number of them). Since we are already familiar with the `ld` instruction, we know that by the time the `cmp` instruction is executed, r would already have the value of `x`; therefore, all we need to do is deduct `ten` from r (10 is the actual value in memory). This is required because we need to leave some value in the register so that, when the instruction `jl` is executed, we can decide whether to jump to `done` or not. Here is an illustration to help you better understand this:

input		equivalent to		interpretation
`ld   x`		$r = x$		$r = memory[addr]$
`cmp  ten`	→	$r = r - 10$	→	$r = r - memory[addr]$
`jl   done`		$ip = done$ `if` $r < 0$		$ip = addr$ `if` $r < 0$

It is important to note that *addr* indicates the position of the value stored in `x`, then it represents the value of `ten`, and finally, it represents the place to jump to if $r < 0$. This makes sense since we execute instructions in a sequential order, meaning that *addr* can only ever be one thing at a time. This distinction from the previous example with the `add` instruction is crucial! Remember that when we looked at compering values, we must subtract something from somewhere in order to arrive at a result that is either less than or equal to zero, equal to zero, or greater than or equal to zero. Do **not** forget that `cmp` should return the value

$$\texttt{cmp} \overset{\text{def}}{=} \begin{cases} 0 & \textbf{if } x = y, \text{ where } x \text{ is } \boldsymbol{r} \text{ and } y \text{ is } \textbf{any} \text{ label} \\ \leq 0 & \textbf{if } x < y \\ \geq 0 & \textbf{if } x > y \end{cases}$$

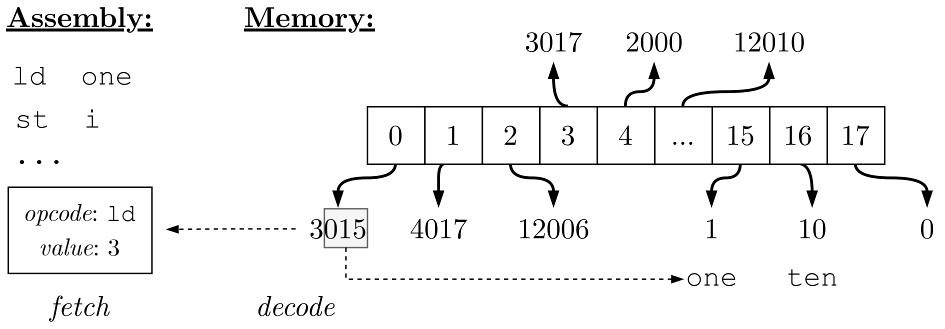
Also, do **not** forget to leave the resulting value in r.

4.2 Basic Computation

Your interpreter should be able to handle any number of commands and generate errors when necessary. We will use a technique of evaluation that is divided into three parts: *fetch*, *decode*, and *execute*. Fetch refers to getting the next instruction to execute from the code currently being executed. Decode means determining which instruction we have just fetched, and finally, executing the instruction means performing the action that the semantics of the instructions prescribes. Therefore, your interpreter has a simple job, it should loop around a fetch-decode-execute mechanism. The pseudo-code provided below will give you a general idea of what you need to implement and ultimately complete.

```
Input: A vector of command-line arguments
Output: 0 if the program terminates successfully
Function main
    /* code for first and second pass goes here */
    set register $r$ and instruction pointer $ip$ to 0
    while $ip \geq 0$ do
        address := compute label address at memory index $ip$
        code := compute opcode value at memory index $ip$
        /* jump to the next instruction */
        if code is get then
            read input from the user
            display input entered by the user
        else if code is put then
            display value stored in register $r$
        else if code is ld then
            execute instruction
        else if code is ... then
            ...
        else if code is halt then
            set instruction pointer $ip$ to -1
        else
            set instruction pointer $ip$ to -1
    return 0
```

This is basically a large case of **if-else** statements with a conditional expression for each if-statement that checks each instruction (as given in Table 1) inside a loop that repeats until there is no more code to interpret. Basically, your interpreter should stop when it hits a `halt` instruction or runs out of instructions to execute at the end of the code memory. (Please, refer to the sample out below to see examples of output messages.) Here is an example of a fetch-decode-execute operation:



1. We fetch the command `ld one`.
2. We decode the command: this involves computing the address of label `one`, as well as computing the mnemonic value of `ld`.
3. We execute the instruction: store the value of `one` in r and then assign it to `i`.

5 Input

An input file can have single-line comments, each starting with a semicolon (`;`), which you must remove. Keep in mind that undefined behavior during the execution of your interpreter may result from any left comments, so remove them! Additionally, labels begin in column one and can also be found in column three. Operators or instructions are *only* found in column two. Therefore, it is probably a good idea to set some sort of field separator variable in your assembler's first pass that can be used to identify every possible combination of blanks and tabs in the current input line. For example, leading white spaces may be treated as a field separator as well as any white spaces in between. Again, since Labels *always* appear in the first column (and occasionally in the third), while operators or instruction *always* appear in the second column, to find the column in which each appears, use blank and tab characters.

Consider the following snippets of assembly code, where the first code has two labels, one in column one and the other in column three

```
begin  ld  tmp
```

and the second code has a label only in column three

```
       ld  tmp
```

Clearly, both `ld` instructions appear only in column two, for example

input file 1:

columns		
one	two	three
`begin`	`ld`	`tmp`

input file 2:

columns		
one	two	three
	`ld`	`tmp`

therefore, when processing and parsing an input file that has content similar to the second snippet of assembly code, make sure to remember that opcodes *always* appear in column two.

6 Output

Let us consider the following Fibonacci sequence written in assembly.

```
; compute the Fibonacci sequence
; Approach:
; fibonacci($n) = $n if $n=0 or $n=1
; fibonacci($n-2) +
; fibonacci($n-1) if $n>=2

  ld  zero  ; $first = 0
```

```
        st      first
        ld      one      ; $second = 1
        st      second

        get              ; get $n from the user
        st      n

        ld      one      ; $i = 1
        st      i
loop    ld      first    ; display $first
        put
        add     second   ; $temp = $first + $second
        st      tmp
        ld      second   ; $first = $second
        st      first
        ld      tmp      ; $second = $tmp
        st      second
        ld      i        ; $i++
        add     one
        st      i
        cmp     n        ; if $i <= $n
        jle     loop     ; go back and compute the next Fibonacci number

        halt             ; terminate the program
zero    const  0
one     const  1
first   const
second  const
tmp     const
n       const
i       const
```

You must remove comments, keep track of labels, store instructions/opcodes and addresses, and then write the resultant code to an output file that will be processed a second time during the second pass of your assembler. The generated output file should have all comments removed and labels that initially appeared in column one also removed. Although the format is not important, make sure to leave a space between instructions and labels for each line. Here is an example of the resulting output file:

```
ld zero
st first
ld one
st second
get
st n
ld one
st i
ld first
put
```

```
add second
st tmp
ld second
st first
ld tmp
st second
ld i
add one
st i
cmp n
jle loop
halt
const 0
const 1
const
const
const
const
const
```

This is the output that you will use for the second pass of your assembler. Now, the output that your assembler generates during the second pass should be stored in an array of integers. This array represents the "memory", that is, this is the location where all commands − instructions and labels − are stored.

7 Specifications

- From a design standpoint, and to keep things simple, we will use a `struct` to store a collection of related or unrelated items − besides a `struct` is one of the logical choices. The information you will store includes *symbols* and *opcodes*.
    1. You have to write a forward-declaration of a `struct` called **Symbol** that consists of the *name* of a label, and a *memory* that represents the location where an instruction should jump to, or the location of a value in memory. Remember that *memory* should be, for all intended purpose, an integer.
    2. You have to write a forward-declaration of a `struct` called **Opcode** that consists of the *name* of an opcode and a *mnemonic* representation of such instruction, which is an integer.
- You are **only** allowed to use the following (constant) global variables:
    - **`const int`** `LABEL_COUNT = 100`
    - **`const int`** `OPCODE_COUNT = 18`
    - **`const int`** `MAX_CHARS = 5`
    - **`const int`** `STACK_SIZE = 1000`
    - **`const std::string`** `OPCODE_LIST = ...`

- Your program must use the following functions:
  - *splitOpcodes:* a function that breaks a string into words, each of which represents a different instruction, and then stores the words in an array of opcodes. The string's value is a collection of all the instructions. (*Hint*: You must make use of the global variable `OPCODE_LIST`.)

    ```
    void splitOpcodes(std::string inst, Opcode* opcodes)
    ```

  - *isOpcode:* a function that returns *true* if the value of a string matches one of the instructions in the array of opcodes.

    ```
    bool isOpcode(std::string& word, Opcode* opcodes)
    ```

  - *getOpcode:* a function that returns an *integer* that corresponds to the value of the string of one the opcodes in the array of opcodes. Otherwise, it returns −1 if the string is not found in the array of opcodes.

    ```
    int getOpcode(std::string& word, Opcode* opcodes)
    ```

  - *getLocation:* a function that returns a label's location in memory or −1 if such label does not exists. It is important to point out that since this function is used during the second pass of the assembler, it takes the name of a label, and array of "current" labels, and the position of the next label in memory to check if the label that goes by some name can be found in the array of labels.

    ```
    int getLocation(std::string& name, Symbol* labels, int
       nextLabel
    ```

  - *isNumber:* a function that returns *true* if the string is a valid number. (*Hint*: Some labels may represent numbers, such as 1, 10, etc; thus, you must correctly transform a string into an integer. You can use the technique you learned in 135 to convert a value from a *string* to *int*.)

    ```
    bool isNumber(std::string& str)
    ```

  - You should use the four additional utility functions that I created for you to test and debug your program's output: *padding*, *dumpOpcodes*, *dumpSymbols*, and *dumpMemory*.
- If you use the UNLV remote server to compile your program, please use `bobby.cs.unlv.edu` or `sally.cs.unlv.edu`.

8 Sample Run

```
$> g++ main.cpp
$> ./a.out test

Running program...
45
read: 45
67
read: 67
```

```
0
read: 0
result: 112
** Program terminated **

$> g++ main.cpp
$> ./a.out test2

Running program...
45
read: 45
67
read: 67
result: 3015
** Program terminated **

$> g++ main.cpp
$> ./a.out test5

Running program...
result: 1
result: 2
result: 3
result: 4
result: 5
result: 6
result: 7
result: 8
result: 9
result: 10
** Program terminated **

$> g++ main.cpp
$> ./a.out test4

Running program...
10
read: 10
result: 0
result: 1
result: 1
result: 2
result: 3
result: 5
result: 8
result: 13
result: 21
result: 34
```

```
** Program terminated **
```

Here is an example with an error message. This is the same summation program that expects the user to enter two correct integer values.

```
$> g++ main.cpp
$> ./a.out test

Running program...
break
ERROR: invalid input!
** Program terminated with an error code **
```

Use this example as a guide to identify any potential invalid input.

9 Assignment Notes

- To get the highest marks, you will have to prevent errors. Good programming practice means that your program should prevent errors whenever possible − that is, it should detect them when possible or necessary.
- All declared variables must be used!
  - Do **not** use global variables unless specified in the assignment.
  - Use local variables whenever possible.
  - Use meaningful names for your variables. I understand that some of my variable names, such as *ip* for instruction pointer and *r* for register, were terrible, but it is OK because I only used them to explain the assignment.
- Use indentation and comments. Comment your source code appropriately according to the stipulations on the rubric. Please format your program using a text editor (or an IDE, if available) so that we can read it. Otherwise, we will not be able to assist you. We will tell you to come back after you are done fixing the format of your program!
  - Each line of code that is at the same block level needs to have the same indentation.
  - Inner blocks must be indented by at least two spaces compared to the outer block.
- Do **not** try to do everything at once. I repeat, do **not** try to do everything at once. Instead, write one execution routine (or function) at a time, then test it thoroughly before going on the next one. In fact, write a few lines and then test that your program compiles and runs properly.
- Do **not** forget to make your program visually appealing by neatly printing the correct output and error messages. Make sure your output is **exactly** formatted like the given sample output or code grade will eat your grade!
- We will run your code with different content, so we will catch you if you just hard-code output! You will receive a score of **zero** for this assignment.
- Do **not** look for solutions online as you will not find any. The assembly language is something that I created for a unique/never-seen/never-used/never-run virtual machine, so the set of

instructions are therefore different.

- Of course, feel free to reach out to me if you have any questions. (For the other sections of 202, please contact your instructor and/or TA.)
- May the C^{++} compiler be with you.

10 Submission

Submit the source files to code grade by the deadline.

11 References

Supplemental video https://youtu.be/YReFdzr6cVQ