CSE 304 – Compiler Design
Fall, 2023
Assignment 01 – SSM Interpreter

Assigned: Wednesday, 08/30/2023
Due: **Friday, 09/15/2023, at 11:59 PM**

Problem: SSM Interpreter

For this problem, you are asked to implement, in Python, an interpreter for SSM, a simple stack machine-based assembly language.

Script name: `ssm_interpreter.py`

Input:
Your program should read the name of a file from the command-line. This file will contain the SSM instructions that your program must scan and then execute.

Output:
The result of the execution of the SSM instructions in the input file should be written to standard output.

Stack Machine:
The input file consists of a sequence of instructions for a stack machine.

- This machine has no general-purpose registers.
- It has two separate memory areas:
  - the first, called the **`store`**, which is directly addressed
  - and the second, called the **`stack`**, which holds operands for operations
- There are two kinds of machine instructions (all defined below):
  - load/store to move values between the two memory areas
  - and the rest which manipulate the top (few) entries on the stack.
- The machine manipulates only integer data.

SSM Assembly Language:
The SSM assembly language consists of the following instructions:

- **`ildc`** ***num***: push the given integer **num** on to the stack.
- **`iadd`**: pop the top two elements of the stack, add them, and push their sum on to the stack.
- **`isub`**: subtract the top-most element on stack from the second-to-top element; pop the top two elements of the stack and push the result of the subtraction on to the stack.
- **`imul`**: pop the top two elements of the stack, multiply them, and push their product on to the stack.
- **`idiv`**: divide the second-to-top element on the stack by the top-most element; pop the top two elements of the stack and push the result of the division (the quotient) on to the stack.

- **`imod`**: divide the second-to-top element on the stack by the top-most element; pop the top two elements of the stack and push the result of the division (the remainder) on to the stack.
- **`pop`**: pop the top-most element off the stack.
- **`dup`**: push the value on the top of stack on to the stack (i.e. duplicate the top-most entry in the stack).
- **`swap`**: swap the top two values on the stack. That is, if **n** is the top-most value on the stack, and **m** is immediately below it, make **m** the top-most value of the stack with **n** immediately below it.
- **`jz` *`label`***: pop the top-most value from the stack; if it is zero, jump to the instruction labelled with the given label; otherwise go to the next instruction.
- **`jnz` *`label`***: pop the top-most value from the stack; if it is not zero, jump to the instruction labelled with the given label; otherwise go to the next instruction.
- **`jmp` *`label`***: jump to the instruction labelled with the given label.
- **`load`**: the top-most element of the stack is the address in **`store`**, say **a**. This instruction pops the top-most element, and takes the value at address **a** in **`store`**, and pushes that value to the stack.
    - For instance, let **`store`** be such that integer 10 is at address 4. Then, when load is executed with 4 at top of stack, the 4 is replaced with 10.
    - load does not change or remove values from **`store`**
- **`store`**: Treat the second-from-top element on the stack as an address **a**, and the top-most element as an integer **i**. Pop the top two elements from stack. The cell at address **a** in the **`store`** is updated with integer **i**.
    - For instance, let **`store`** be such that integer 10 is at address 4. Let the stack have 4 as the second-from-top entry and 12 as the top entry. The store instruction will pop off 4 and 12 and update the value of cell 4 from 10 to 12.

SSM Programs:
An SSM assembly language program is a sequence of instructions:

- Each instruction may be preceded by an optional label and a semi-colon (":").
- A label is a sequence of alphabetic characters or numeric characters or underscore ("_"), beginning with an alphabetic character.
- An integer num is a sequence of numeric characters, optionally preceded by a minus sign ("-").
- The instructions in the assembly language should always be in lower-case.

Note that instructions **`ildc`**, **`jz`**, **`jnz`**, and **`jmp`** take one argument; the other instructions have no arguments.

The label may be separated from the following instruction by a sequence of zero or more whitespace (blank, tab, newline) characters. An instruction and its argument will be separated by a sequence of one or more whitespace characters.

IMPORTANT: In the examples below, each instruction is on a line of its own; in general, though, there may be more than one instruction in a line; or a single instruction may be split over multiple lines.

The input assembly program may also have comments. A comment begins with the "#" symbol and ends at the end of the line.

SSM Program Execution:
When a program is evaluated, it starts with an empty stack. Program evaluation continues until the last instruction in the program is evaluated. When an input program is completely evaluated, the interpreter should print the top-most value on the stack and exit.

Errors in SSM Programs:
If an input program has errors (e.g., improperly formed labels, invalid instruction, invalid format for numbers, labels that appear in destinations of jumps but not defined elsewhere, etc.), the interpreter should give an error message and exit without executing a single instruction. The details (i.e., the source of the error, line number, etc.) in the error message are up to you. All I expect is that your interpreter should at least say that there is some error and exit instead of trying to execute an illegitimate program.

Similarly, when executing a program, if the program tries to access values in an empty stack, the interpreter should signal an error and exit; and if the program tries to access a value in a cell in **`store`** without initializing the cell first, it should signal an error and exit. Again, you can choose the error message format.

Assignment Advice:

- Start by assuming that your input will always be only valid (error-free) programs that do not contain any labels. Example 1 below is one such program.
- Second, consider SSM programs without labels that may have errors. Now you have to first check for errors before you try to execute an input program. Modify your code to do this.
- Only after you have successfully completed the first two steps, should you modify your interpreter to handle programs with labels and comments.

Submission Instructions:

1. Submit all of your files (at least one) in a single zip archive named: `cse304_a01.zip`
2. Make sure you include your name, netid, and Student ID number as comments at the top of your source code files.
3. Make sure you use any specified file, function, and class names (and any other names specified in the instructions). Grading may be partially automated (by importing your code to run against our test cases) and the use of incorrect names will interfere.

This is an independent programming assignment. Any collaboration on coding, with anyone other than your recognized partner, will be considered a violation of academic honesty.

Examples:

Example 01:

Input File:

```
ildc 10
ildc 20
iadd
```

Output:

```
30
```

Example 02:

Input File:

```
          ildc 20
          ildc 5
here:     ildc 1
          isub
          dup
          jz there
          swap
          ildc 10
          iadd
          swap
          jmp here
there:
          pop
```

Output:

```
60
```