CSE 101: Introduction to Computational and Algorithmic Thinking

Homework #2-1

Fall 2018

Assignment Due: Monday, December 10, 2018, by 11:59 PM

Getting Started

To help you get started on this assignment, we have given you a skeleton Python file named `interpreter.py`. This file contains several *function stubs* and a few tests you can try out to see if your code seems to be correct. Stubs are functions that have no bodies (or have very minimal bodies). You must fill in the bodies of these functions for the assignments. **Do not, under any circumstance, change the names of the functions or their parameter lists.** The automated grading system will be looking for exactly those functions provided in `2-1.py`. **Please note that the test cases we provide you in the bare bones files will not necessarily be the same ones we use during grading!**

Directions

This assignment is worth a total of 20 points. The automated grading system will execute your solution several times, using different input values each time. Each test that produces the correct/expected output will earn 4 points. You must earn at least 16 points in order to pass (receive credit for) this assignment.

- At the top of the `interpreter.py` file, include the following information in comments, with each item on a separate line:
  - your full name ***AS IT APPEARS IN BLACKBOARD***
  - your Stony Brook ID #
  - your Stony Brook NetID (your Blackboard username)
  - the course number (CSE 101)
  - the assignment name and number (Assignment #2-1)

⚠ Any functions that we tell you to complete (i.e., that we provide stubs for in the starter code) **MUST** use the names and parameter lists indicated in the starter code file. Submissions that have the wrong function names (or whose functions contain the wrong number of parameters, or that use different parameter names) can't be graded by our automated grading system, and may receive a grading penalty.

⚠ Be sure to submit your final work as directed by the indicated due date and time. Late work will not be accepted for grading. Work is late if it is submitted after the due date and time.

⚠ Programs that crash will likely receive a grade of zero, so make sure you thoroughly test your work before submitting it.

A Simplified Machine Language Interpreter

Earlier this semester, we (briefly) discussed machine language, which is ultimately what is processed by a computer (programs written using high-level languages like Python and Java must be compiled or interpreted into equivalent sequences of machine language instructions in order to be executed). Different types of CPUs "speak" different machine language "dialects". In this lab, we will work with part of a simple machine language called Pep/8 (Pep/8 is not a "real" machine language supported by actual CPUs; it is a simplified example used in some computer architecture classes to teach machine language concepts).

Pep/8 performs its work using a single memory register called the *accumulator*, which stores one value at a time. Various instructions can be used to modify or retrieve the value stored in the accumulator; when a program finishes, whatever is left in the accumulator is the final result.

Every Pep/8 instruction is defined as a sequence of 24 bits (or 6 hexadecimal digits):

- The first four bits indicate the *operation code* (opcode), which identifies the type of instruction
- The fifth bit indicates the register where the operation is to be applied; in our case, we will always use the same memory register (Register 0).
- The sixth through eighth bits indicate the *addressing mode*, which tells us how to interpret the operand:
    - If the addressing mode is 000 (*immediate mode*), then the operand is treated as a literal number.
    - If the addressing mode is 001 (*direct mode*), then the operand is treated as a memory address.
- The remaining 16 bits represent the *operand*: a numerical value (in binary).

The full Pep/8 language contains a large number of instructions. For this lab, we will only consider the following seven Pep/8 instructions (with the assumption that the operands will always be unsigned integer values):

1. LOAD (opcode 1100): This instruction assigns the value of the operand (or the value stored in the operand memory address) to the accumulator
2. STORE (opcode 1110): This instruction copies the current accumulator value into the memory address referred to by the operand
3. ADD (opcode 0111): This instruction adds the current accumulator value to the value of the operand (or the value stored in the operand memory address), and stores the result back in the accumulator
4. SUBTRACT (opcode 1000): This instruction subtracts the value of the operand (or the value stored in the operand memory address) from the current accumulator value, and stores the result back in the accumulator
5. MULTIPLY (opcode 1001): This instruction multiplies the current accumulator value by the value of the operand (or the value stored in the operand memory address), and stores the result back in the accumulator
6. DIVIDE (opcode 1010): This instruction divides the current value of the accumulator by the value of the operand (or the value stored in the operand memory address) using floor division (i.e., the `//` operator in Python), and stores the result back in the accumulator
7. HALT (opcode 0000): This instruction signals the end of a Pep/8 program; no further instructions will be executed.

For example, the Pep/8 instruction `0111000100001011` would be broken down into the following elements:

- opcode = 0111 = ADD
- addressing mode = 001 = direct mode
- operand = 00001011 = 11 (in base 10). Since we are using direct addressing mode as indicated above, this means that we will use the value stored in memory address 11, not the value 11 itself.

Thus, this instruction means, "Add the value contained in memory address 11 to the accumulator."

For this assignment, you will write part of a Python program that loads a text file containing a sequence of Pep/8 instructions (encoded as 6-character hexadecimal strings), translates and "executes" that sequence of instructions, and returns information regarding the final result of execution. The `interpreter.py` starter code includes some useful global variables and helper functions for you to use along the way. Follow the instructions below to complete the Python functions that will execute the main program logic.

Part I: Processing Single Instructions (14 points)

Complete the `processInstruction()` function, which takes a single string argument representing one Pep/8 instruction (in hexadecimal form). This function returns a string (see the steps below for details on what string to return in which situations). It uses two global variables: an integer that stores the current contents of the accumulator register, and a dictionary representing the computer's memory (keys are positive integers representing memory addresses, while values represent the data values stored at those memory addresses).

1. Start by calling the provided `hexToBin()` helper function to translate the parameter into a 24-bit binary string. If the result does not contain exactly 24 characters, immediately return the string `"ERROR"`.
2. Use string slicing to break the binary string into three parts: the opcode (the first 4 bits), the address mode (the 6th–8th bits), and the operand (the 9th–24th bits); skip the fifth bit, which will always be 0 for our purposes. Store each part in its own variable. Then convert the operand from binary into base 10.

   Easy Binary-to-Decimal (Base 10) Translation

   Python's `int()` command will translate a number in any base (represented as a string) into base 10 if you supply the original base as the second argument. For example, `int("11010", 2)` means that the operand is in base 2, and will return 26, its base 10 equivalent.

3. Now it is time to process the instruction:
   (a) If the opcode is `0000`, return the string `"HALT"` to indicate that the Pep/8 program has ended.
   (b) If the opcode is `1100`:
   - If the address mode is `000`, set the value of the accumulator to the value of the operand. Return the string `"LOAD X INTO ACCUMULATOR"`, where X is the operand's value.
   - If the address mode is `001`, use the operand value as a dictionary key. If that key is present in the dictionary, assign its value to the accumulator variable and return the string `"LOAD CONTENTS OF MEMORY ADDRESS X INTO ACCUMULATOR"`, where X is the operand's value. If the key is not present, return the string `"ERROR"` instead.
   - If the address mode is any value other than `000` or `001`, return the string `"ERROR"` and do not modify the value of the accumulator.

(c) If the opcode is `1110`:

- If the address mode is `001`, treat the operand as a dictionary key; assign it the accumulator variable's value. Return the string `"STORED ACCUMULATOR INTO MEMORY ADDRESS X"`, where X is the operand's value.
- If the address mode is any other value, return the string `"ERROR"` and do not modify the accumulator (`STORE` commands require the operand to be a memory address, not a literal value).

(d) If the opcode is `0111`:

- If the address mode is `000`, add the operand to the value of the accumulator and assign the result to the accumulator variable. Return the string `"ADDED X TO ACCUMULATOR"`, where X is the operand's value.
- If the address mode is `001`, use the operand value as a dictionary key. If that key is present in the dictionary, add its value to the value of the accumulator, assign the result to the accumulator variable, and return the string `"ADDED CONTENTS OF ADDRESS X TO ACCUMULATOR"` (where X is the operand's value). If the key is not present, return the string `"ERROR"` instead.
- If the address mode is any value other than `000` or `001`, return the string `"ERROR"` and do not modify the value of the accumulator.

(e) If the opcode is `1000`:

- If the address mode is `000`, subtract the operand's value from the value of the accumulator, and assign the result to the accumulator variable. Return the string `"SUBTRACTED X FROM ACCUMULATOR"`, where X is the operand's value.
- If the address mode is `001`, use the operand value as a dictionary key. If that key is present in the dictionary, subtract its value from the accumulator, assign the result to the accumulator variable, and return the string `"SUBTRACTED CONTENTS OF ADDRESS X FROM ACCUMULATOR"`, where X is the operand's value. If the key is not present, return the string `"ERROR"`.
- If the address mode is any value other than `000` or `001`, return the string `"ERROR"` and do not modify the value of the accumulator.

(f) If the opcode is `1001`:

- If the address mode is `000`, multiply the accumulator's value by the operand's value and assign the result to the accumulator variable. Return the string `"MULTIPLIED ACCUMULATOR BY X"`, where X is the operand's value.
- If the address mode is `001`, use the operand value as a key into your dictionary. If that key is present in the dictionary, multiply its value by the value of the accumulator, assign the result to the accumulator variable, and return the string `"MULTIPLIED ACCUMULATOR BY CONTENTS OF ADDRESS X"`, where X is the operand's value. If the key is not present, return the string `"ERROR"`.
- If the address mode is any value other than `000` or `001`, return the string `"ERROR"` and do not modify the value of the accumulator.

(g) If the opcode is `1010`:

- If the address mode is `000`, divide the accumulator's value by the operand's value, using floor division, and assign the result to the accumulator variable. Return the string `"DIVIDED ACCUMULATOR BY X"`, where X is the operand's value.
- If the address mode is `001`, use the operand value as a key into your dictionary. If that key is present in the dictionary, divide the accumulator's value by the dictionary value (using floor division), assign the result to the accumulator variable, and return the string `"DIVIDED ACCUMULATOR BY CONTENTS OF ADDRESS X"`, where X is the operand's value. If the key is not present, return the string `"ERROR"`.
- If the address mode is any value other than `000` or `001`, return the string `"ERROR"` and do not modify the value of the accumulator.

(h) Otherwise (for any other opcode value), return the string `"ERROR"`.

Examples:

Function Call	Return Value
`processInstruction("D1008D")`	`ERROR`
`processInstruction("8000CB")`	`SUBTRACTED 203 FROM ACCUMULATOR`
`processInstruction("310B4")`	`ERROR`
`processInstruction("00008D")`	`HALT`
`processInstruction("A10073")`	`DIVIDED ACCUMULATOR BY CONTENTS OF ADDRESS 115`
`processInstruction("9000D4")`	`MULTIPLIED ACCUMULATOR BY 212`
`processInstruction("7100D7")`	`ERROR`
`processInstruction("C0005F")`	`LOAD 95 INTO ACCUMULATOR`
`processInstruction("7200F1")`	`ERROR`
`processInstruction("E0004D")`	`ERROR`

Part II: Processing an Entire Program (6 points)

The `executeProgram()` function takes a single string argument representing the name of a text data file to process. It returns an integer value. Like `processInstruction()`, it uses the global variables `ram` and `accumulator`.

1. Start by setting `ram` to an empty dictionary and setting the value of `accumulator` to 0.
2. Create a variable to count the number of lines (instructions) that have been successfully processed. Initialize this variable to 0.
3. For each line in the data file:
   (a) Call `processInstruction()` with the current line. Save its result (the string returned by the function) in a new variable.
   (b) If the result is the string `"ERROR"`, use the `break` command to end the loop.
   (c) Otherwise, add 1 to the number of lines processed. If the result is the string `"HALT"`, use the `break` command to end the loop at this time.
4. After the loop ends, return the number of instructions that were successfully "executed" (this number includes the final `"HALT"` instruction, if present).

Examples:

Function Call	Return Value
`executeProgram("program1.txt")`	`4`
`executeProgram("program2.txt")`	`3`
`executeProgram("program3.txt")`	`10`