1X1: PROGRAMMING LANGUAGES Assignment 2

Due Aug 22nd in the Assignment 2 Dropbox on

E3

OVERVIEW

As mentioned in lecture, all of the languages we will study this quarter are interpreted.

An interpreter translates programs written in one language into computational actions, effectively running the program being interpreted. Code Analysis is the first phase in Interpretation, which ensures the source program is syntactically & semantically correct.

This assignment requires a SIMPLESEM parser (Assignment 1), and implement the semantics of the program instructions; in other words, the interpreter's computational engine.

SIMPLESEM MACHINE ARCHITECTURE & INSTRUCTION-CYCLE

A major difference between assignment #1 & #2 is how the program is parsed. In assignment #1, statements were read from the input file and parsed by the parser. In assignment #2, statements will be read from the input file and stored into the Code (C) segment of memory.

The SIMPLESEM machine is a vonNeumann machine. The interpreter requires:

- a Program Counter (PC)—that points to the current or next instruction
- an Instruction Register (IR)—that contains an instruction for the current cycle
- Memory: Code (C), and Data (D)—store the instructions & data values for use by the program
- A run-bit—indicates whether the processor should continue fetching-executing instructions.

The values of these registers & memory represent the state of the processor. The SIMPLESEM machine also follows the FETCH-INCREMENT-EXECUTE cycle. A description of the cycle follows:

- **FETCH**—The IR is updated with the instruction pointed at by PC; **IR=C[PC]**
- **INCREMENT**—The PC is incremented to *point* to the next instruction in C; **PC = PC+1**
- **EXECUTE**—The semantics of the instruction in the IR (see below) change the state øf the SIMPLESEM processor

SIMPLESEM INSTRUCTION FORMAT & SEMANTICS

The SIMPLESEM Interpreter executes SIMPLESEM instructions. The SIMPLESEM instruction set contains four op-codes: `set`, `jump`, `jumpt`, and `halt`. The instruction formats and semantics are as follows:

Instruction Format	**Instruction Semantics**
`set destination*, source**`	`D[destination] = source`
`jump destination`	`PC = destination`
`jumpt destination, Boolean-condition`	`if(boolean-condition)` `  PC = destination`
`Halt`	`run_bit = false`

*-- except when the destination is `write`, in which case the source is a parameter to the provided `void write()` function.

**--except when the source is `read`, in which case the source is a call to the provided `int read()` function.

To perform the interpretation of a SIMPLESEM program you will:

- Initialize the processor's state:
  - Open & Read a SIMPLESEM program into the code segment: `C`
  - Program Counter to the first instruction of the program: `PC = 0`
  - Data Segment: `D` entries to `0`
  - `run_bit = true`
- Enter the **fetch-increment-execute** cycle and repeat this cycle while `run_bit == true`

To properly implement the semantics of each SIMPLESEM instruction, use the structure and recursion of the parser built in assignment #1 to parse the parameters of each instruction and implement semantics. This will be covered in greater detail during lecture & discussion.

SI**MPLESEM GRAMMAR**

The grammar for SIMPLESEM is as follows:

<Program>→ <Statement> {<Statement>}

<Statement> → <Set> | <Jump> | <Jumpt> | **`halt`**

<Set>→ **`set`** (`write` | <Expr>)**`,`** (`read` |<Expr>)

<Jump>→ **`jump`** <Expr>

<Jumpt>→ **`jumpt`** <Expr>**`,`** <Expr> (**`!=`** | **`==`** | **`>`** | **`<`** | **`>=`** | **`<=`**) <Expr>

<Expr>→ <Term> {(**`+`** | **`-`**) <Term>}

<Term>→ <Factor> {(**`*`** | **`/`** | **`%`**) <Factor>}

<Factor>→ <Number> | **`D[`**<Expr>**`]`** | **`(`**<Expr>**`)`**

<Number>→ **`0`**| (`1..9`){`0..9`}

Guide to EBNF: |— separate alternate choices, ()—choose 1, {}—choose 0 or more, **`keyword/symbol`**—what is in bold **`CourierNew`** font.

OUTPUT

NOTE: Program output will be directed to an output file. Use the provided `void printDataSeg()` function, to print the contents of memory once the program has terminated. Please see the sample `.out` for example output.

SUBMISSION : SUBMIT CODE ONLY. DO NOT SUBMIT: EXECUTABLES, TEST CASE INPUT, OR OUTPUT

Submit the following items to the Assignment #2 Dropbox on E3:

- A single zip file labeled with your Student ID, containing ONLY .cpp/.h OR .py files needed to implement this project. Do not include files that end in: .exe, .o, .obj, etc...
- All projects **must** be able to execute via command line using the following format:

For Python: `python INTERPRETER.py Program#.S`
For C++:
To Compile: `g++ INTERPRETER.cpp -o INTERPRETER`
To Execute: `INTERPRETER Program#.S`