UCI Summer 2020 1X1: PROGRAMMING LANGUAGES

Assignment 2 - SIMPLESEM INTERPRETER

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 implements 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 of 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.

SIMPLESEM 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.

PROGRAM INPUTS

Script file (*.S)

Program script file will be specified in the first (and the only) command line argument. An example is provided below,

Program0.S

```
set 0, read

jumpt 5, D[0] == 0

set write, D[D[0]]+2

set 0, D[0] - 1

jump 1

halt
```

Input file

Input file `input.txt` is the place where you specify input data for

```
set <Expr>, read
```

statements. The input file is meant to be read sequentially from line 1 to the end of file. The maximum length of input.txt is 40 lines. The first "set <Expr>, read" will read from the first line of `input.txt`, the second "set <Expr>, read" will read from the first line of `input.txt`, and so on.

To access `input.txt`, ***hardcode*** a file stream reader to read from `input.txt` under program working directory.

input.txt example (may have up to 40 lines)

```
10
35
3
4
5
6
7
8
9
```

OUTPUT FORMATTING REQUIREMENTS

Using the provided `void printDataSeg()` function, to print the contents of memory once the program has terminated. Please see the sample `.out` for example output.

An example of program output is listed below.

Program#.S.out example

```
2
3
...
6
5
7
Data Segment Contents
0: 3
```

1: 4

2: 6

3: 6

4: 7

5: 9

. . .

99: 5

Each output file contains two sections, 1) the

`set write <Expr>`

outputs, and 2) data segment contents.

The first section should list all `set write <Expr>` outputs in individual lines, separated by newline characters (\n).

The second section begins with a line identical to “Data Segment Contents” (note that this line is case-sensitive), followed by 100 lines of data register (0-99) values denoted in the format of

`<data_register_id>:␣<register_value>`

The symbol “␣” stands for a space.

Output file location

The output file should be written to the same directory as the input file (*.S). For example (assume C++)

`./interpreter /some/directory/Program#.S`

should lead to the creation of

`/some/directory/Program#.S.out`

SUBMISSION

Submit the following items to the Assignment #2 on Grade scope.

- Submission must be a single program file named **INTERPRETER.{java, cpp, py}**

- **File name must be named exactly one of the three "INTERPRETER.java" "INTERPRETER.cpp" "INTERPRETER.py"**
- **No ZIP submission supported**
- **No late submissions will be accepted**
- Submission languages:
    - You can choose Java, C++, and Python to complete this assignment.
    - **Java**: All projects **must** be able to execute via command lines using the following format:

      ```
      javac INTERPRETER.java

      java INTERPRETER Program#.S
      ```

    - **C++**: All projects **must** be able to execute via command lines using the following format:

      ```
      g++ INTERPRETER.cpp -o interpreter

      ./interpreter Program#.S
      ```

    - **Python**: All projects **must** be able to execute via one command line using the following format:

      ```
      python INTERPRETER.py Program#.S
      ```