CS 330 Homework
Bleakr

1 Overview

Your responsibility in this homework is to learn about assembly programming, the mechanics of a CPU, and mixing languages. You will do this in the context of creating a virtual machine and its visual debugger. The assembly language understood by the virtual machine is called Bleakr[1]. The graphical interface will be implemented in JRuby—which provides bindings for Java's Swing library.

2 Bleakr ISA

Bleakr's *instruction set architecture* is bleaker than Intel's x86 or even ARM. In summary, it has the following:

1. An input list of integers.
2. An output list of integers.
3. A mapping from human-readable string labels to instruction numbers.
4. Ten general purpose registers, named `r0`, `r1`, ..., `r9`.
5. A register named `pc`—short for *program counter*—for recording which instruction is next to be executed.
6. A register named `ic`—short for *input counter*—for recording which integer input is next to be read.
7. Instructions `inc`, `dec`, `add`, and `sub` for performing arithmetic.
8. Instructions `input` and `output` for performing I/O.
9. Instructions `jmp`, `jpos`, `jneg`, and `jzilch` for jumping to labeled instructions.
10. Instruction `store` for assigning a value to a register.

Before reading on, try using your intuition to manually execute this example Bleakr program given the input 1 2 3 4:

```
0  input r0 <- read
1  input r1
2  add r0 r1
3  output r0
4  jmp read
```

[1]Bleakr is a Rubified contraction of the portmanteau "assem-bleak." Compared to a high-level language, the assembly landscape is bleak.

2.1 Execution

When the Bleakr VM starts up, all registers—general purpose or otherwise—have value 0. All instructions and input are gathered into an indexable collection for use during execution. It traverses the code to identify which labels map to which instructions.

On each step of execution, it executes the instruction currently pointed to by `pc`, triggering any side effects associated with the instruction. Then it updates `pc` to prepare for the next instruction. Often this means incrementing `pc` by 1, but jump instructions may change `pc` in less predictable ways.

The virtual machine halts when it encounters one of the following situations regarding the instruction about to be executed:

1. The instruction doesn't exist—that is, `pc` is out-of-bounds.
2. The instruction doesn't follow the grammar specified in Section 2.2.
3. The instruction is `input`, but no more input is available to be read—that is, `ic` is out-of-bounds.

2.2 Instructions

Following is the complete grammar specification for Bleakr's 11 instructions. The placeholders `lval` and `rval` are discussed in Section 2.3.

`add lval rval` Adds the values together and stores the result in `lval`.

`sub lval rval` Subtracts `rval` from `lval` and stores the result `lval`.

`inc lval` Adds one to `lval` and stores the result in `lval`.

`dec lval` Subtracts one from `lval` and stores the result in `lval`.

`input lval` Reads the next input from the input list (as indicated by `ic`) and stores the result in `lval`. Increments `ic`

`output rval` Emits `rval` to the output list.

`store lval rval` Copies `rval` into `lval`.

`jmp label` Alters `pc` so that the next instruction executed is the one with the given label.

`jpos rval label` If `rval` is positive, alters `pc` so that the next instruction executed is the one with the given label.

`jneg rval label` If `rval` is negative, alters `pc` so that the next instruction executed is the one with the given label.

`jzilch rval label` If `rval` is 0, alters `pc` so that the next instruction executed is the one with the given label.

Any instruction may be followed by `<- label`, where `label` is an alphanumeric string that can be used to assign a symbolic name to an instruction. Jump instructions reference these labels rather than an instruction's less meaningful line number. (Bleakr's not that bleak.) Consider this example program, which for each number `N` in the input, outputs all numbers [0, `N`]:

```
0  input r0 <- getN
1  store r1 0
2  jneg r0 getN <- checkN
3  output r1
4  inc r1
5  dec r0
6  jmp checkN
```

2.3 Operands

The operands of an instruction come in two flavors: *lvalues* and *rvalues*. Lvalues are values associated with an identifiable location in the virtual machine—in our case, a register. Any instructions that involve an assignment need a location to store the value and will therefore have an lvalue operand[2].

We can also have values that are more temporary in nature and not necessarily associated with a location. Literals, for instance, are part of an instruction and don't reside in any assignable location. Literals are rvalues—values that can appear in an expression but not as a destination for an assignment. Registers too may be treated as rvalues when we are merely referring to their value but aren't assigning anything to them.

2.4 Direct vs. Indirect Addressing

Registers can be accessed directly through their names `r0`...`r9`. However, they can also be accessed indirectly through another register. In the expression `[r0]`, `r0` is assumed to hold a value in the range 0 through 9. Suppose `r0` is 5. We say that `[r0]` refers to or points to `r5`.

This indirect addressing scheme can simplify certain algorithms. Suppose one wants to assign the value 100 to registers `r1` through `r9`. This code accomplishes the task:

```
0  store r1 100
1  store r2 100
2  store r3 100
3  store r4 100
4  store r5 100
5  store r6 100
6  store r7 100
7  store r8 100
8  store r9 100
```

But indirect addressing allows us to write this with less grunting. We set up `r0` as a counter and have it point to each of the registers. Then we indirectly store 100 through the `r0` counter:

```
0  store r0 9
1  store [r0] 100 <- fill
2  dec r0
3  jpos r0 fill
```

[2]In fact, in high-level languages, terms that can appear on the *left* side of assignment statement are *l*values. However, so are constants that reside in a particular place in memory. These don't appear on the left-hand side of any assignment.

3 Requirements

In order to complete this homework, you must satisfy these requirements:

1. Place all code in directory `<YOUR-REPOSITORY>/bleakr`.
2. All code must be written in JRuby and must run on `thing-0[456]`. You'll need a version of JRuby that supports recent changes to the MiniTest library used by the grader. Using RVM, one can install a recent version with `rvm install jruby-9.0`.
3. Write classes `BleakrVM` and `BleakrDebugger`, described below.
4. Write at least two Bleakr programs to implement algorithms of your choosing. Algorithms must have some definable purpose (calculates x for each input, finds the kth thing, outputs all numbers y such that y is z, and so on) and cannot duplicate previous students' submissions. (This means you must look at them.) At least one program must use indirect addressing.
5. Share your two programs and a screenshot of your `BleakrDebugger` executing one of them on Piazza under the folder `bleakr_share`. Provide a clear description of what each program accomplishes.

3.1 BleakrVM

Write class `BleakrVM` in file `bleakrvm.rb`. This class represents a CPU for the Bleakr assembly language. It has the following `public` interface:

- Method `initialize`, which accepts a path to a Bleakr source file and a path to an input file containing a list of integers, one per line. It initializes the state of the virtual machine to meet the specification described in Section 2.1.

  Suppose you have the following in file `numbers.in`:

  ```
  10
  3
  82
  ```

  And the following Bleakr source code in `echo.bleakr`:

  ```
  0  input r0 <- read
  1  output r0
  2  jmp read
  ```

  One can then instantiate a `BleakrVM` with the following code:

  ```
  BleakrVM.new('echo.bleakr', 'numbers.in')
  ```

  In this case, at the end of `initialize`, the state of the VM will be as follows, with arrays shown in `[]` and hashes shown in `{}`.

  ```
  input: [10, 3, 82]
  output: []
  registers: {r0: 0, r1: 0, ..., r9: 0, pc: 0, ic: 0}
  instructions: [
    'input r0 <- read',
  ```

```
  'output r0',
  'jmp read'
]
labels: {read: 0}
```

- Accessors `input`, `output`, `registers`, `instructions`, and `labels`. They return the VM state according to the types depicted in the `initialize` requirement. If you name your instance variables in accord with these accessors, you can let Ruby automatically generate these methods for you using the `attr_accessor` method.

- Method `resolve_lvalue`, which accepts a `String` expression and returns a location. If the expression is a register name (a *direct* address), just return the expression as is. If the expression is a register name in square brackets (an *indirect* address), look up the number n stored in the specified register, and return the name of register n. Otherwise, return `nil`. For example, suppose `r5` is 9. Then `resolve_lvalue('[r5]')` $\rightarrow$ `'r9'`.

- Method `resolve_rvalue`, which accepts a `String` expression and returns its value. If the expression is a register name, return the register's value. If the expression is a register name in square brackets, look up the number n stored in the specified register, and return the value of register n. For example, suppose `r5` is 9 and `r9` is 20. Then `resolve_rvalue('[r5]')` $\rightarrow$ `20`. If the expression is a literal integer, return it as an integer. Otherwise, return `nil`.

- Method `step`, which evaluates the instruction pointed to by `pc`. Your `resolve_[lr]value` methods will come in handy. If a halting situation is identified, raise an exception (a `RuntimeError` in particular) with an explanatory message of your choosing. `raise`-ing a string is sufficient. For example:

  ```
  raise "The well's run dry. There are no more numbers to input."
  ```

  Update `pc` accordingly. Only registers and the output list may be modified.

- Method `reset`, which restores the registers to their initial states and clears the output list.

3.2 BleakrDebugger

Write class `BleakrDebugger` in file `bleakrdebugger.rb`. This class represents a visual interface to a `BleakrVM` as it executes. It has the following `public` interface:

- Method `initialize` that accepts a `BleakrVM` as the virtual machine to debug. After this method finishes, the debugger window is visible and ready to be interacted with by the user.

- It provides a graphical user interface with at least the following features, implemented using standard Swing components:

  1. A list of the VM's instructions, with the instruction next to be executed highlighted in some way.
  2. A list of the VM's registers, with names clearly mapped to values.
  3. A list of the VM's input, with the number next to be read highlighted in some way.
  4. A list of the VM's output.

5. A widget for advancing the VM by one step.
6. A widget for resetting the VM to its initial state.

You have considerable freedom in how you design your interface. The SpecChecker does not actively test this class, but it will prompt you to manually assert that you are meeting the above requirements. Answer truthfully; your instructor will verify compliance when he inspects your code.

4 Later-week Qualifications

To qualify for later-week submission, you must complete `BleakrVM` in its entirety.

5 Submission

To submit your work for grading:

1. Run the grading script from your homework directory using `../specs/grade`.
2. Commit and push your work to your repository.
3. Verify that your solution is on Bitbucket by viewing your repository in a web browser.

A passing grading script does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The grading script checks some of them, but not all.
- You must successfully submit your code to your repository. Expect to have issues with Git.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

The grading script allows you to signal your instructor when requirements are met. You only need to send an email if you qualified for later week submission and are resubmitting after the original deadline.