Lab 1: Brainf—k

Due: May 30, 2011

1 Objective

You will write both an interpreter and a compiler for Brainf—k, a simple, but Turing-complete language.[1] You will use the *Visitor* design pattern to complete the lab.[2]

2 Brainf—k Semantics

The detailed description for the language is available at `http://en.wikipedia.org/wiki/Brainfuck`.

The runtime environment consists of:

- A 30,000 byte `array` with all elements initialized to zero.
- A movable `pointer` initialized to index 0 (the leftmost element) of the array.
- Input and output streams in ASCII encoding.

The commands consist of:

Character	Type	Meaning
`>`	`Right`	Increment `pointer` (move pointer right)
`<`	`Left`	Decrement `pointer` (move pointer left)
`+`	`Increment`	Increment byte at `pointer`
`-`	`Decrement`	Decrement byte at `pointer`
`.`	`Output`	Output `array[pointer]`
`,`	`Input`	Input `array[pointer]`
`[`	`Loop`	If `array[pointer]` is zero, jump past matching `]`
`]`		jump back to matching `[`

3 Lab Instructions

Follow the instructions and ask for help if you get stuck.

1. Create a class called `BF`.

[1] Technically, the language is only *Turing-complete* if the array is unlimited in size.

[2] `http://en.wikipedia.org/wiki/Visitor_pattern`

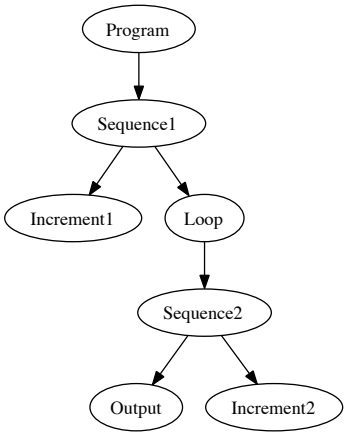
2. Inside `BF`, add the following code (the brainf—k program in `main` should be on one line):[3]

```
public interface Visitor {
    void visit (Left left);
    void visit (Right right);
    void visit (Increment increment);
    void visit (Decrement decrement);
    void visit (Input input);
    void visit (Output output);
    void visit (Loop loop);
    void visit (Program program);
}
public interface Node {
    void accept (Visitor v);
}
private int i = 0;
public static Program parse (String str) {
    return new Program (new BF().doParse(str));
}
public static class PrintVisitor implements Visitor {
    public void visit (Left n) { System.out.print('<'); }
    public void visit (Right n) { System.out.print('>'); }
    public void visit (Increment n) { System.out.print('+'); }
    public void visit (Decrement n) { System.out.print('-'); }
    public void visit (Input n) { System.out.print(','); }
    public void visit (Output n) { System.out.print('.'); }
    public void visit (Loop n) {
        System.out.print('[');
        n.body.accept(this);
        System.out.print(']');
    }
    public void visit (Program n) { n.body.accept(this); }
}
public static void main (String[] args) {
    Node hello = BF.parse("++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+
    ++++++..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.>.");
    hello.accept(new BF.PrintVisitor());
}
```

3. Create the following classes (that implement `Node`) within `BF`: `Left`, `Right`, `Increment`, `Decrement`, `Input`, `Output`, `Loop`, `Program`, and `Sequence`. The implementation of most classes will be identical, with the exception of `Sequence`, `Loop`, and `Program`. `Sequence` has no corresponding `visit` method; in its `accept` method, it instructs all its children to `accept` the visitor in order. `Loop` and `Program` have a `Node`, called `body`. `Program` is a `static` class.

[3] `http://pastebin.com/zdac5EQj`

4. Write the following recursive method inside `BF`: `private Sequence doParse (String str)`. Use `BF` member variable `i` to track the location in `str`. `doParse` should return `Sequence1` for brainf—k program `+[.+]`:



5. If the `PrintVisitor` prints the original Brainf—k program given to `BF.parse`, then you correctly parsed the code.
6. Write an `InterpreterVisitor` inside `BF`. If it works correctly, you should see `Hello World!`
7. Test `InterpreterVisitor` using source code from:
   `http://esoteric.sange.fi/brainfuck/bf-source/prog/`
8. Write a `CompilerVisitor` inside `BF`. Generate equivalent code in your choice of programming language (e.g., Java, Python, C++). Be sure to test that this works, too!

4 Next steps

Complete any two of the following tasks.[4]

1. `DebugVisitor`. Write an interactive debugger for this language.[5]
2. `AnimationVisitor`. Render the contents of the 30,000 byte `array` as a 300×100 animation.[6]
3. `AssemblyVisitor`. Output assembly for the GNU assembler (`as`), the Netwide Assembler (`nasm`), or the Microsoft Assembler (`masm`).
4. `OptimizationVisitor`. Write an optimizer that improves execution performance by replacing runs of the same instruction with a single instruction.[7]
5. Write a program in Brainf—k that prints your name.

[4] To earn an additional 1 day extension or 25% extra credit, implement any four of the tasks.
[5] Support stepping through program execution and printing out the pointer and its content.
[6] The GUI should show the location of the pointer and the contents of the entire array.
[7] E.g., ++++ becomes a single Increment node with a count of 4.