Homework 1 — Warm-up exercise — assigned Wednesday 25 August, due Wednesday 1 September

Total points: 300.

Write an emulator for a hypothetical computer—a simple stack machine for integer list manipulation; then write a corresponding translator.

1.1 Interpreter

1.1.1 General description

Consider a simple stack machine for integer list manipulation (an exercise devised by P. Sestoft, IT Copenhagen). The stack machine has a program p, a program counter pc, a stack s that may hold integers and lists, and a stack top pointer sp. The instructions of the machine and their effect on the stack are described by the following table:

Instruction	Stack before		Stack after	Effect
`CST` i	s	$\Rightarrow$	$i\ s$	Push integer constant i
`ADD`	$i_2\ i_1\ s$	$\Rightarrow$	$(i_1 + i_2)\ s$	Add integers
`SUB`	$i_2\ i_1\ s$	$\Rightarrow$	$(i_1 - i_2)\ s$	Subtract integers
`DUP`	$v\ s$	$\Rightarrow$	$v\ v\ s$	Duplicate
`SWAP`	$v_2\ v_1\ s$	$\Rightarrow$	$v_1\ v_2\ s$	Swap
`POP`	$v\ s$	$\Rightarrow$	s	Pop
`GOTO` a	s	$\Rightarrow$	s	Jump to a
`IFNZRO` a	$i\ s$	$\Rightarrow$	s	Jump to a if $i \neq 0$
`CALL` a	$v\ s$	$\Rightarrow$	$v\ r\ s$	Call function at a, pushing return address r
`RET`	$v\ r\ s$	$\Rightarrow$	$v\ s$	Return: jump to r
`MKNIL`	s	$\Rightarrow$	`Nil` s	Push `Nil` (the empty list)
`MKCONS`	$t\ i\ s$	$\Rightarrow$	`Cons`(i,t) s	Push cons node
`LISTCASE` a	`Nil` s	$\Rightarrow$	s	If `Nil`, do not jump
`LISTCASE` a	`Cons`(i,t) s	$\Rightarrow$	$t\ i\ s$	If `Cons`, unpack components and jump to a
`PRINT`	$v\ s$	$\Rightarrow$	$v\ s$	Print v and keep it
`STOP`	s	$\Rightarrow$	_	Halt the machine

In the table, i stands for an integer, a for an address, within the program, r for an address within the program, t for a list, v for an arbitrary value, and s for the remainder of the stack (0 or more values).

An instruction with an argument (`CST`, `GOTO`, `IFNZRO`, `CALL`, `LISTCASE`) takes up two positions in the program.

Under any circumstances not covered by the table, the machine will crash. For instance, if the instruction to be executed is an `ADD` but the top stack element is a list rather than an integer, the machine will crash.

1.1.2 Code completion (40pts)

Given the skeleton of a list stack machine interpreter written in Java, provided below, complete the rest of the code.

Your code must be placed in a package named
`edu.unm.cs.cs591491aoop.fall2004.`*yourusername*`.liststackmachine`.

According to the provided code, you are to complete the method `execcode`. You must also write a `main` method, which must accept command-line arguments as follows: the first argument on the command line, a string, is the name of a file containing the "assembly" text of a program (i.e., machine instructions in human-readable form, as in the examples provided below); the second argument on the command line, an integer, is the initial value of the program counter; the remaining arguments (if any) on the command line, which are integers, are the initial contents of the stack, from top to bottom. Presumably you will write a separate `load` method, and, for debugging purposes, a `disassemble` method.

For example, if you put the assembly text:

```
    0: CST 3
    2: ADD
    3: PRINT
    4: POP
    5: STOP
```

in a file `add3.lsm`, then you should be able to invoke the interpreter as follows:

yourfavoritejvm `edu.unm.cs.cs591491aoop.fall2004.`*yourusername*`.liststack\`
`machine.ListStackMachine add3.lsm 0 5`

and expect to see 8 printed.

1.1.3 Code skeleton

(From P. Sestoft, IT Copenhagen.)

```
abstract class List
{
}
class Nil extends List
{
    public String toString ()
    {
        return "Nil";
    }
}
class ListStackMachine
{
    private static int execcode (int [] p, Object [] s, int pc, int sp)
    {
        for (;;)
```

```
            {
                switch (p [pc++])
                    {
                    case CST:
                        s [sp+1] = new Integer (p [pc++]);
                        sp++;
                        break;
                    case ADD:
                        s [sp-1] = new Integer (((Integer) s [sp-1]).intValue ()
                                                + ((Integer) s [sp]).intValue ());
                        sp--;
                        break;
                    case MKNIL:
                        s [sp+1] = new Nil ();
                        sp++;
                        break;
```

1.1.4 Code improvement (40pts)

The version of the interpreter you just completed is written essentially as corresponding C code might be written. Redesign and reimplement the interpreter, fully adhering to the principles and practices of object-oriented programming. Put in in a package `...betterliststackmachine`.

1.2 Analysis and discussion

Having implemented the interpreter, answer the following questions.

1.2.1 (5pts)

Manually execute the code below on the stack machine. Show the stack contents after every instruction and show what is printed on the console:

```
0: CST 100
2: CST 11
4: ADD
5: MKNIL
6: MKCONS
7: PRINT
8: STOP
```

1.2.2 (5pts)

Write stack machine code to create and print the list `Cons(111,Cons(222,Nil))`.

1.2.3 (5pts)

The instructions `CALL` and `RET` can be used to implement simple functions. What does the following stack machine program do, assuming that the integer 42 is on the stack top when execution is started at instruction address 0?

```
 0: CALL 4
 2: PRINT
 3: STOP
 4: DUP
 5: DUP
 6: MKNIL
 7: MKCONS
 8: MKCONS
 9: MKCONS
10: RET
```

Show the contents of the stack after each instruction, in the order in which the instructions are executed.

1.2.4 (10pts)

Write a stack machine program that, given that integer $n \geq 0$ is on the stack top, builds and prints an n-element list of the form `Cons(`n`,Cons(`$n-1$`,...,Cons(1,Nil),...))`. Provide both an iterative and a recursive solution.

1.2.5 (5pts)

The instruction `LISTCASE` can be used to test whether the list on the stack top is `Nil` or a `Cons`. If the stack top element is `Nil`, execution simply continues with the next instruction. If the stack top element is `Cons(`i`,`t`)`, then the integer i and the list tail t are unpacked on the stack and execution continues at address a.

Consider the following stack machine code fragment:

```
21: LISTCASE 27
23: CST 999
25: GOTO 28
27: POP
28: PRINT
```

Assume that the list `Cons(111,Cons(222,Nil))` is on the stack top when the function at address 21 is called (by `CALL 21`). What is on the stack top, and is therefore printed, when control reaches instruction 28? What is printed if the empty list is on the stack top when `CALL 21` is executed?

1.2.6 (10pts)

Write a stack machine function that, given that a list is on the stack top, computes the sum of the list elements. Provide both an iterative and a recursive solution.

1.2.7 (10pts)

For some programs and some initial states of the machine, the machine will crash. Write some programs that cause the machine to crash. What kinds of conditions cause a crash?

We would like to know for certain if a given stack machine program will crash when run. If we are given a program, an initial program counter, and an initial stack, can we prove that the program will crash, or, more usefully, that it will not crash? What do you think? If this cannot be done in general, discuss possible restrictions on admissible programs that may allow such proofs to go through.

1.2.8 (5pts)

Extend the machine with a new instruction `GETVAR` i that reads the contents of the ith stack element below the stack top and pushes that value onto the stack. Since the stack top is the 0th element below the stack top, `GETVAR` 0 should be equivalent to `DUP`.

1.2.9 (5pts)

Use the new `GETVAR` instruction to write a stack machine function that can create a list that contains n copies of v, like this:

$$\texttt{Cons(}v\texttt{,Cons(}v\texttt{,...,Cons(}v\texttt{,Nil)...))}$$

The function should be called with a stack of the form $n : v : s$, that is, with n on the stack top and v below it, and should return with a stack of the form $w : s$ where w is an n-element list all of whose elements are v. Provide both an iterative and a recursive solution.

1.2.10 (5pts)

Use the new `GETVAR` instruction to write a stack machine function that, given that integer $n \geq 0$ is on the stack top, builds and prints an n-element list of the form

$$\texttt{Cons(}2n\texttt{,Cons(}2n-2\texttt{,...,Cons(2,Nil),...))}.$$

Provide both an iterative and a recursive solution.

1.2.11 (5pts)

Without using the `GETVAR` instruction, write a stack machine function that, given that integer $n \geq 0$ is on the stack top, builds and prints an n-element list of the form

$$\texttt{Cons(}2n\texttt{,Cons(}2n-2\texttt{,...,Cons(2,Nil),...))}.$$

Provide both an iterative and a recursive solution.

1.3 Translator 1 (50pts)

Write (in Java) a translator from the machine code of the hypothetical list stack machine to C. The generated C code must be ANSI C.

Pay special attention to memory allocation and deallocation. The translated code, when C-compiled and run, must not leak memory.

1.4 Translator 2 (100pts)

Write (in Java) a translator from the machine code of the hypothetical list stack machine to Java bytecode.

For instance, the assembly program `add3.lsm` should be translated into a Java class file `add3.class`, such that executing

yourfavoritejvm `add3.class 0 5`

prints 8, i.e., has the same effect as with the interpreter.

It is acceptable and expected that the generated class files will invoke the services of some support classes, which you may write in Java, package into a Jar file and provide in the classpath in effect when the above command is executed.

To ease the nitty-gritty of the translation, especially the generation of well-formed class files, use the *Byte Code Engineering Library* (BCEL), available at `http://jakarta.apache.org/bcel/`.

Provide a detailed description of the translation strategy you used, its rationale, and an argument for its correctness. Is a straightforward translation always possible? Is translation always possible? Is the resulting code of similar size to the original?

How to turn in

For both the interpreters and the translators, turn in a top-level directory containing at least the following:

- a `README` file (a brief description of the structure and purpose of the code, with basic instructions for its use);
- a `COPYING` file (a copyright notice or something functionally equivalent);
- a `TODO` file (a list of known bugs and deficiencies with respect to the assignment, or identified but unimplemented extensions);
- a `src` directory containing the Java source hierarchy for your code;
- a `classes` directory containing the compiled Java class files hierarchy, exactly corresponding to the `src` files;

- a `doc` directory containing various detailed project documentation as needed (including a detailed description of code structure (classes, interfaces, and how they are related) in a file `structure.txt` or, preferably, `structure.ps`, a description of the design process that led to this code structure in a file `design.txt` or, preferably, `design.ps`, and a subdirectory `generated/javadoc` with Javadoc-generated HTML files, exactly corresponding to the `src` files);
- a `tests` directory with input files used for testing the correctness of the program, including both valid and invalid inputs;
- a `test-results` directory with output files generated by the program on test inputs; and
- a `RESULTS` file (a summary of the testing process, documenting the program's compliance with the specification).

Note that for the translator you have to provide, as tests, a number of assembly programs covering, to the best of your ability, all programming situations, and for each its translation into a Java class file (suitably disassembled), *together with tests that demonstrate that the translated code works correctly*.

Place everything in a compressed tar file and email to the instructor.