Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @UTS 203, Labs: Tue 12:00-13:50 & Wed 12:00-13:50 @FAB 88-10

Lab 2: Interpreters

In this lab, you'll have a hands-on practice in implementating several simple interpreters. The experience you gain here will be useful for Assignment 1, which is to implement an interpreter for a slightly more complex language.

Download from D2L the file `lab2.zip` and unzip it.

Interpreters for LL0 & LL1

In this week's lecture, we've seen two simple low-level langauges , LL0 and LL1. The following is the instruction set of LL1:

Instruction	Meaning
`LOAD` *addr*	`ACC` ← `mem`[*addr*]
`STORE` *addr*	`mem`[*addr*] ← `ACC`
`MOVE` *i*	`ACC` ← *i*
`ADD` *addr*	`ACC` ← `ACC` + `mem`[*addr*]
`SUB` *addr*	`ACC` ← `ACC` − `mem`[*addr*]
`JUMP` *tgt*	`PC` ← *tgt*
`JUMPZ` *tgt*	`PC` ← *tgt*, if `ACC` = 0
`CALL` *tgt*	`save PC`; `PC` ← *tgt*
`RETURN`	`PC` ← `saved caller's PC`
`HALT`	stop execution

LL0 is a subset of LL1; it does not have the `CALL` and `RETURN` instructions.

Exercise 1 Take a look at the program `LL0Interp.java`. It contains an interpreter for LL0. See the close connection between the instructions' semantics and the actual interpretation code. Compile and run this interpreter.

Exercise 2 Take a look at the program `LL1Interp.java`. Complete this interpreter by providing execution code for the two new instructions.

Exercise 3 Write a new LL1 test program for adding 1, 2, and 3. In this new version, the `main()` function calls a function `f()`, which calls a function `g()` with argument 1, and returns the call's result as its own return value. The function `g(i)` adds 2 and 3 to the input argument `i` and returns the sum.

Interpreters for EL0 & EL1

The two expression languages shown in this week's lecture, EL0 and EL1, are functaional-style high-level languages. Their grammars are shown below:

1. Prog → Exp
2. Exp → Exp + Exp
3. Exp → `let` var = Exp `in` Exp `end`
4. Exp → var
5. Exp → num
6. Exp → `fn` var => Exp
7. Exp → Exp Exp

EL0's grammar is defined by the first five productions, while El1's grammar by all seven productions.

In the lecture, the interpretation actions for these two languages are defined abstractly by the semantic actions:

```
Prog -> {env = empty;} Exp {Prog.val = Exp.val;}
Exp  -> Exp1 + Exp2 {Exp.val = Exp1.val + Exp2.val;}
Exp  -> let var = Exp1 {env = extend(env, var.id, Exp1.val);}
        in Exp2 end {env = retract(env, var.id); Exp.val = Exp2.val;}
Exp  -> var {Exp.val = lookup(env, var.id);}
Exp  -> num {Exp.val = num.val;}
```

```
Exp -> fn var => Exp1 {Exp.val = closure(var.id, Exp1, env);}
Exp -> Exp1 {c = Exp1.val;}
       Exp2 {actual = Exp2.val; stack.push(env);
             env = extend(c.env, c.formal, actual);
             Exp.val = c.body.val; env = stack.pop();}
```

Exercise Take a look at the program `EL0Interp.java`. It contains an interpreter for EL0. See the connection between a production's semantic actions and the actual interpretation code. Compile and run it.

Now complete the interpreter for EL1 in `EL1Interp.java` by converting the semantic actions for the `CALL` node into actual interpretation code. Compile and test the finished program.

Interpreter for SC0

SC0 is a simple stack-machine IR. It is a subset of SC1, which we programmed in Lab 1.

SC0's Instructions

Instruction	Sematics	Stack Top (before *vs* after)
CONST n	load constant n to stack	→ n
LOAD n	load var[n] to stack	→ val
STORE n	store val to var[n]	val →
ALOAD	load array element	arrayref,idx → val
ASTORE	store val to array element	arrayref,idx,val →
NEWARRAY	allocate new array	count → arrayref
NEG	- val	val → result
ADD	val1 + val2	val1,val2 → result
SUB	val1 - val2	val1,val2 → result
MUL	val1 * val2	val1,val2 → result
DIV	val1 / val2	val1,val2 → result
AND	val1 & val2	val1,val2 → result
OR	val1 | val2	val1,val2 → result
GOTO n	pc = pc + n	
IFZ n	if (val == 0) pc = pc + n	val →
IFNZ n	if (val != 0) pc = pc + n	val →
IFEQ n	if (val1 == val2) pc = pc + n	val1,val2 →
IFNE n	if (val1 != val2) pc = pc + n	val1,val2 →
IFLT n	if (val1 < val2) pc = pc + n	val1,val2 →
IFLE n	if (val1 <= val2) pc = pc + n	val1,val2 →
IFGT n	if (val1 > val2) pc = pc + n	val1,val2 →
IFGE n	if (val1 >= val2) pc = pc + n	val1,val2 →
PRINT	print val	val →

Note: For the jump instructions, the operand n represents the *relative* displacement from the the current instruction position. n can be either positive or negative.

SC0 Interpreter Structure

There are three parts to this interpreter implementation:

1. A front-end to read in a SC0 program from a file.
2. An implementation of the three memory models used in SC0:
   - the operand stack,
   - the local variable array, and
   - the heap storage for array objects.
3. The actual interpretation of the SC0 instructions.

We represent SC0's instructions as strings ending with the EOL (end-of-line). With this representation, the front-end is quite easy to handle with Java's BufferedReader:

```
FileInputStream stream = new FileInputStream(args[0]);
BufferedReader reader = new BufferedReader(new InputStreamReader(stream));
ArrayList<String> insts = new ArrayList<String>();
String line;
while ((line = reader.readLine()) != null) {
  if (!(line.isEmpty() || line.trim().equals("")
        || line.startsWith("#")))
    insts.add(line);
}
```

To decode an individual SC0 instruction, we use Java's string `Scanner`:

```
sc = new Scanner(inst);
lnum = sc.next();                        // line number (ignored)
name = sc.next();                        // inst name
n = sc.hasNextInt() ? sc.nextInt() : 0; // inst operand
```

The three storage models can be implemented with Java's `HashMap`, `ArrayList`, and `Stack`, respectively:

```
static HashMap<Integer,Integer> vars = new HashMap<Integer,Integer>();
static List<Integer> heap = new ArrayList<Integer>();
static Stack<Integer> stack = new Stack<Integer>();
```

Exercise A starter version of an SC0 interpreter is provided in `SC0Interp0.java`. It contains the above-mentioned code portion, as well as the structure of instruction execution routine. Your task is to complete the interpreter by providing execution code for individual instructions.