Prof. Jingke Li (lij@pdx.edu), Lec: T 1000-1150; Labs: WR 1400-1550, R 1000-1150; Office Hour: TR 1200-1300; ZID: 87054734177

Lab 6: Imperative Languages

(with thanks to Andrew Tolmach)

Learning Objectives

Upon successful completion, students will be able to:

- Understand side-effects in expressions.
- Program with small languages of different levels.
- Implement some compiler functions for a simple language.

Instructions

Download the file `lab6.zip` from D2L into any convenient directory and type

```
$ unzip lab6.zip
```

You'll see a new directory `lab6` with a few files and a sub-directory containing sample solutions.

1. A Simple Imperative Expression Language

Here we introduce a simple language, Imp. It's syntax is defined by the following context-free grammar (except for `Expr` and `Op`, all symbols are terminals):

```
Expr -> num                   # integer
     |  var                   # string
     |  (Op Expr Expr)        # binop
     |  (:= var Expr)         # assign
     |  (print Expr)          # output
     |  (seq Expr Expr)       # sequence
     |  (if Expr Expr Expr)   # if-stmt
     |  (while Expr Expr)     # while-loop
Op   -> + | - | * | / | < | =
```

Note that Imp is an expression language because every program is a value-producing expression. Yet it is also an imperative language, since it has structured control constructs, `seq`, `if`, and `while`, and its expressions may have side-effect.

The evaluation of each expression yields a single integer result, specifically

- A number `i` yields itself.
- A variable `x` yields its current value. Every variable is implicitly initialized to 0 at the beginning of program execution.
- Evaluating `(+ e1 e2)` evaluates `e1` and then `e2`, and yields the sum of their values.

  The other arithmetic operations are similar, with one exception. For a `/` expression, if `e2`'s value is 0, the expression's value is undefined.
- Evaluating `(< e1 e2)` evaluates `e1` and then `e2`, and compares their values. If the first is less than the second, the expression yields 1; otherwise it yields 0.

- Evaluating `(= e1 e2)` evaluates `e1` and then `e2`, and compares their values. If the two values are the same, the expression yields 1; otherwise it yields 0.
- Evaluating `(:= x e)` evaluates `e`, assigns the resulting value into variable `x`, and yields that value.
- Evaluating `(print e)` evaluates `e`, prints the resulting value (followed by a `newline`) to standard output, and yields that value.
- Evaluating `(seq e1 e2)` evaluates `e1` and then `e2`, and yields the value of `e2`.
- Evaluating `(if c t f)` evaluates `c`; if the result is non-zero, then expression `t` is evaluated, otherwise expression `f` is evaluated. The `if` expression yields the value of the selected expression, either `t` or `f`.
- Evaluating `(while c b)` evaluates expression `c`; if the result is non-zero, expression `b` is evaluated and the entire `while` expression is evaluated again; otherwise the evaluation of the `while` is complete. A while expression always yields the value 0.

These evaluation rules form an informal semantics for Imp.

Exercises

1. Of all the Imp expression forms,
   (a) Which ones directly introduce side-effects? List them all.
   (b) Which ones never have side-effects? List them all.
   (c) Based on the above answers, the remaining expression forms *may* have side-effects. Choose one such form, and give two concrete expression examples of the form, one with side-effects, one without.
2. According to the binop expression's evaluation rule listed above, a binop's operands are evaluated from left to right. If we relax this rule, do you think a binop expression may produce different values? Give a concrete example.

2. A New Stack Machine

In Lab 4, you worked with a stack machine (SM0) for evaluating boolean expressions. Here, we define a new stack machine, SM:

Instruction	Semantics
PUSH n	push constant n to stack
LOAD x	load `vars[`x`]` to stack
STORE x	store val to `vars[`x`]`
ADD	$val_1 + val_2$, push result to stack
MUL	$val_1 * val_2$, push result to stack
DIVREM	$val_1 \,/\, val_2$, push div and rem results to stack
LE	$val_1 < val_2$, push 1 (if true) or 0 (if false) to stack
EQ	$val_1 = val_2$, push 1 (if true) or 0 (if false) to stack
POP	pop off val
DUP	replicate val
SWAP	swap val_1 and val_2
PRINT	print val
LABEL i	nop, marking a label position in program
JUMP i	branch to Label i
JUMPZ i	if $val = 0$ branch to Label i

- n represents an integer value.
- x represents a variable name (a string).
- i represents a label number.
- `vars[]` is an auxiliary array mapping variables to values. All variables are assumed to have an initial value of 0.
- val and val_2 represent the top element of the stack, while val_1 represents the second-to-top element.

A SM program is a list of instructions, with two possible forms. In the single-line form, the instructions are separated by semicolons; in the multi-line form, each instruction occupies a separate line. Here are some examples, expressed in the single-line form:

```
PUSH 1; PUSH 2; ADD
LOAD x; DUP; STORE y
PUSH 5; DUP; PRINT
PUSH 0; JUMPZ 101; PUSH 1; LABEL 101; PUSH 2
```

A valid SM program leaves one and only one item on the operand stack at the end of its execution. That item represents the program's result. The above four programs' results are 3, 0, 5, and 2, respectively. (*Note:* the `DUP` instructions in the second and third programs are needed; without them, there would be no item on the operand stack at the end of those two programs.)

Exercises Manually translate the following Imp expressions into SM programs:

1. `(- 5 3)`
2. `(/ 5 3)`
3. `(:= x 1)`
4. `(seq 1 2)`
5. `(seq x y)`
6. `(if 1 2 3)`
7. `(if x y z)`

Note: There is no subtraction instruction in SM. You need to find a combination of other instructions to implement subtraction. Also, you need to pay attention to the SM's requirement that there is one only one item on the stack at the end of program execution.

A SM interpreter is provided in `sm.py`. You should use it to verify your SM programs:

```
linux> ./python3 sm.py
PUSH 1; PUSH 2; ADD
^D
```

Note that `sm.py` takes input from `stdin`. This allows it to read input from a file:

```
linux> cat test.sm | ./python3 sm.py
```

3. A Simple Compiler

The file `impcomp.py` contains an incomplete Imp to SM compiler. The incomplete parts correspond to the hand-translated expressions you did above. Your task is to complete the compiler implementation. The hand-traslation examples can serve as blue-prints for the general cases.

Here are some hints and suggestions:

- The SM programs must precisely match their corresponding Imp expressions' semantics. In particular, each operand should be evaluated only once and the order of operand evaluation should be left to right.
- As you've already encountered above, since a valid SM program leaves one and only one item on the operand stack at the end of its execution, you need to pay attention to intermediate results of an expression, and pop-off those unneeded items from the stack.
- For implementing the `if` expression, you'll need to introduce labels. There is a `newlab` iterator; a unique label number is returned every time `next(newlab)` is called.

Once you've completed the implementation, you should try compiling some simple Imp programs, and then test the results with the SM interpreter.

Reminder: You are required to submit your work by the end of the lab session (with a one-hour grace period) on D2L. It does not need to be complete; just submit what you have by the end of the session.