

CS 558 Homework 2 – due 5:00pm, Tuesday, Jan. 22, 2004

Homework must be submitted by mail to `apt@cs.pdx.edu`, with the subject line “CS558 HW2”. All submitted files (4 for this assignment) must be sent as plain-text *attachments* to the mail message (the contents of the message itself will be ignored). It is *your* responsibility to submit the homework in the proper format.

All programs mentioned can be downloaded from the course web page.

1. Imperative Language Interpreter.

Consider a simple language of imperative expressions, which we’ll call “E2.” Its “concretized” abstract syntax is given by the following grammar:

```
prog := exp
exp := var
    | int
    | '(' ':' var exp ')'
    | '(' 'while' exp exp ')'
    | '(' 'if' exp exp exp ')'
    | '(' 'write' exp ')'
    | '(' 'block' { exp } ')'
    | '(' '+' exp exp ')'
    | '(' '-' exp exp ')'
    | '(' '*' exp exp ')'
    | '(' '/' exp exp ')'
    | '(' '<=' exp exp ')'
var := string of letters
```

As before, comments may be included by enclosing them between ‘{’ and ‘}’ characters, and they may be nested.

An informal semantics for E2 is as follows. The evaluation of each expression (and hence of a program) yields a single integer result.

- A variable x yields its current value. Every variable is implicitly initialized to 0 at the beginning of program evaluation.
- An integer i yields itself.
- Evaluating the assignment expression (`:= x e`) evaluates e , assigns the resulting value into variable x , and yields that value.
- Evaluating (`while e_1 e_2`) repeatedly performs the following steps: e_1 is evaluated; if the result is non-zero e_2 is evaluated; otherwise, the evaluation of the `while` is complete. A `while` expression always yields the value 0.
- Evaluating (`if e_1 e_2 e_3`) evaluates e_1 ; if the result is non-zero, then e_2 is evaluated and resulting value is yielded as the value of the `if` expression; otherwise, e_3 is evaluated and the resulting value is yielded as the value of the `if` expression.

- Evaluating (`write e`) evaluates e , prints the resulting value (following by a newline) to standard output, and yields that value.
- Evaluating (`block e1 e2 ... en`) evaluates e_1, e_2, \dots, e_n in that order, and yields the value of e_n . If $n = 0$, the `block` expression yields 0.
- Evaluating (`+ e1 e2`) evaluates e_1 and e_2 and yields the sum of their values.
- The other arithmetic operations are similar.
- Evaluating (`<= e1 e2`) evaluates e_1 and e_2 , and compares their values. If the first is less than or equal to the second, the expression yields 1; otherwise it yields 0.

An example `primes.e2` program written in this language is available on the web page.

Two E2 interpreters have been provided, one in Java (`hw2_1.java`) and the other in ML (`hw2_1.sml`). Each interpreter reads a file containing an E2 program in the syntax described above, echoes the program (to confirm correct parsing), evaluates the program (possibly producing output from `write` expressions), and displays the evaluation result. The Java program interprets the abstract syntax directly. The ML program first translates it into stack machine code (for an extended and somewhat different version of the stack machine from Homework 1), prints out the stack code (for debugging purposes), and then executes it. For more debugging help, you can trace the behavior of the stack machine by executing the top-level command `Machine.traceOn := true;` before executing `Interp.interp`.

(a) One of the interpreters evaluates the operands to binary operators (`+`, `-`, `*`, `/`, and `<=`) from left-to-right; the other from right-to-left. Write an example E2 program, *not* involving `write` statements, that produces *different* answers depending on the order of evaluation of these operands, and hence different for the two interpreters. Put your program in file `sol2_1.e2` and submit this file. Your program must include a comment stating:

- which interpreter has which behavior; and
- based on this program, what (if anything) can be deduced about the order of evaluation of operands in the Java and ML languages themselves.

(b) Modify *each* interpreter to support `for` expressions, as specified below. The code for parsing these expressions is already present; what you need to do is extend the abstract syntax, add printing code, and add evaluation code (in Java) or compilation code (in ML). When modifying the ML version don't change or add to the stack machine.

The `for` expression has the following syntax:

```
exp := '( 'for' var exp exp exp )'
```

Evaluating (`for x e1 e2 e3`) first evaluates e_1 to a value v_1 and stores v_1 into x . It then repeats the following steps: evaluate e_2 to a value v_2 ; fetch x ; if $x > v_2$ then terminate evaluation of the `for`, yielding the value 0; otherwise, evaluate e_3 and discard the yielded result, fetch x , add one to it, store back into x , and repeat.

For example,

```
(for i 1 10 (write i))
```

prints the numbers from 1 to 10, and yields the value 0.

Make sure you get the order of evaluation right. For example, the bizarre program

```
(for i i (block (:= i (+ i 2)) 10) (block (write i) (:= i (+ i 3))))
```

prints out the numbers 2,8 and yields the value 0.

Hints: Model your code on the existing code for `while`. The ML version may require some tricky stack manipulation to get the order of evaluation right; once again, you may find the `SWAP` instruction useful.

Put your revised Java interpreter in `sol2_1.java`, and your revised ML interpreter in `sol2_1.sml`, and submit both these files.

2. Axiomatic Semantics

Consider the very simple language used to illustrate axiomatic semantics in lecture. Suppose we add a non-deterministic *guarded if* statement to our language. The syntax of this statement is

```
gif E1 -> S1
[] E2 -> S2
...
[] En -> Sn
end
```

where the E_i are boolean expressions. This statement is executed by non-deterministically choosing *any one* i such that E_i evaluates to true, and executing the corresponding S_i . Among other things, this statement gives an elegant, symmetric way to express functions like `max`, e.g.

```
gif x1 ≥ x2 -> max := x1
[] x2 ≥ x1 -> max := x2
end
```

Note that if $x_1 = x_2$ then either branch might be chosen non-deterministically. In this example, the resulting value of `max` will be the same either way, but we can also write guarded `gif`'s that are intentionally non-deterministic, e.g., to simulate a coin toss:

```
gif true -> coin := 0
[] true -> coin := 1
endcoin
```

For more about this statement, see Loudon, p. 270.

This question has two parts. Type your answers to both parts into a file `sol2.2.txt`, and submit this file.

(a) Extend the axiomatic semantics given in lecture by writing down an appropriate rule of inference (GIF) for `gif` statements.

(b) Derive the following triple, by presenting a formal proof tree for it (as shown in lecture).

```
{ z < 0 }
while (x > -5) do
  gif x <= 0 -> y := -x
  □ x >= 0 -> y := x
end;
z := z - y
end
{ z < 0 }
```

Remember to label each node in the tree with the appropriate axiom or rule of inference. You'll need one (WHILE), one (COMP), one (GIF), three (ASSIGN), and four (CONSEQ) steps. Don't worry about formatting your answer neatly, so long as it is clear.