CS 558 Homework 2 – due 6:00pm, Wednesday, Oct. 15, 2003

Homework must be submitted by mail to cs558acc@cs.pdx.edu. 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 e1 e2) repeatedly performs the following steps: e1 is evaluated; if the result is non-zero e2 is evaluated; otherwise, the evaluation of the while is complete. A while expression always yields the value 0.

- Evaluating (if e1 e2 e3) evaluates e1; if the result is non-zero, then e2 is evaluated and resulting value is yielded as the value of the if expression; otherwise, e3 is evaluated and the resulting value is yielded as the value of the if expression.
Evaluating \((\text{write } e)\) evaluates \(e\), prints the resulting value (following by a newline) to standard output, and yields that value.

Evaluating \((\text{block } e_1 \ e_2 \ldots \ e_n)\) evaluates \(e_1, e_2, \ldots, e_n\) in that order, and yields the value of \(e_n\). If \(n = 0\), the \text{block} expression yields 0.

Evaluating \((+ \ e_1 \ e_2)\) evaluates \(e_1\) and \(e_2\) and yields the sum of their values.

The other arithmetic operations are similar.

Evaluating \((<= \ e_1 \ e_2)\) 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 \text{primes.e2} program written in this language is available on the web page.

Two E2 interpreters have been provided, one in Java (\text{hw2_1.java}) and the other in ML (\text{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 \text{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 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 \text{Machine.traceOn := true}; before executing \text{Interp.interp}.

(a) In what order do the interpreters evaluate the operands to \(+\), \(-\), \(*\), \(/\), and \(\leq\)? Is it left-to-right or right-to-left? To help find out, write an example E2 program, not involving \text{write} statements, that produces different answers depending on the order of evaluation. Put this program in a file \text{sol2_1.e2} and include it in your submission.

(b) Modify each interpreter so that it evaluates the operands listed in (a) in the opposite order from the current versions. (In other words, if the versions I handed out do things left-to-right, change them to do things right-to-left, and vice-versa.) When modifying the ML version, you’ll need to change the \text{Compile.compile} function; \textit{don’t} change the \text{Machine.exec} function. Hint: You may find the \text{SWAP} instruction useful.

(c) Modify each interpreter to support \text{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 \text{for} expression has the following syntax:

\[
\text{exp} := (' ' 'for' ' var exp exp exp ')
\]

Evaluating \((\text{for } x \ e_1 \ e_2 \ e_3)\) evaluates \(e_1\) to a value \(v_1\), then evaluates \(e_2\) to a value \(v_2\), and then stores \(v_1\) into \(x\). It then repeats the following steps: fetch \(x\); if \(x > v_2\) then terminate evaluation of the \text{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 2) 10) (block (write i) (::= i (+ i 2))))

prints out the numbers 0, 3, 6, 9.

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.

Combine your answers to parts (b) and (c) into a single revised Java interpreter, sol2_1.java, and a single revised ML interpreter, sol2_1.sml, and include both these files in your submission.

2. Consider the very simple language used to illustrate axiomatic semantics in Lecture 2. This problem asks you do to some “paper and pencil” work (no programming). Type your solution to all three parts of the problem into a single file sol2_2.txt and include this file in your submission.

(a) Suppose we add a break statement to this language. Execution of break causes control to transfer immediately out the bottom of the nearest enclosing while loop. (Any program containing a break that is not within a while loop is illegal; for this problem, assume such illegal programs don’t occur.)

The proof rule for while given in lecture, namely

\[
\begin{align*}
\{ P \land E \} & S \{ P \} \\
\{ P \} & \text{while } E \text{ do } S \{ P \land \neg E \}
\end{align*}
\]

is no longer valid for this revised language containing break. Illustrate this fact by giving an example program in the revised language, annotated with one or more assertions that are provable using the while proof rule, but are actually false.

(b) Now suppose that we modify the original language in a different way, by adding a statement form

\[
\text{stmt} := \text{'}loop\text{'} \text{stmt} \text{'exitif'} \text{exp} \text{stmt} \text{'end'}
\]

The semantics of loop s₁ exitif e s₂ end are equivalent to
\[ b = \text{true}; \]
\[ \text{while } (b) \text{ do} \]
\[ s_1; \]
\[ \text{if } e \text{ then} \]
\[ b := \text{false} \]
\[ \text{else} \]
\[ s_2 \]
\[ \text{endif} \]
\[ \text{end} \]

where \( b \) is a fresh variable that doesn’t appear anywhere else in the program. In other words, \text{loop} sets up an indefinite iteration with a conditional exit in the middle.

Consider the following program fragment in this modified language (which can be found in file hw2_2.txt).

\begin{verbatim}
loop
  x := x - 1;
  exitif (x <= 0);
  if (x <= 1)
    y := y + x
  else
    y := y + 1
  endif
end;
if (x <= 100)
  x := x + 1
else
  x := 0;
endif
\end{verbatim}

Construct an informal proof that this fragment maintains the invariant \( x + y = c \); in other words, assuming that \( x + y = c \) before the fragment executes, prove that \( x + y = c \) after it is executes.

You can record your proof by adding suitable assertions after each statement, in the style shown in lecture.

(c) Write down the strongest proof rule you can for \text{loop} \( S_1 \text{ exitif } E \text{ } S_2 \text{ end}. \) Hint: It should be strong enough to justify the informal proof you gave in part (b)!

4