1. Axiomatic Semantics

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

\[
\text{gif } E_1 \rightarrow S_1 \\
\quad \text{ [] } E_2 \rightarrow S_2 \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{...} \\
\quad \text{ [] } E_n \rightarrow S_n \\
\text{ end}
\]

where the $E_i$ are boolean expressions. This statement is executed by non-deterministically choosing \textit{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 $\text{max}$, e.g.

\[
\text{gif } x_1 \geq x_2 \rightarrow \text{max} := x_1 \\
\quad \text{ [] } x_2 \geq x_1 \rightarrow \text{max} := x_2 \\
\text{ end}
\]

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

\[
\text{gif } \text{true} \rightarrow \text{coin} := 0 \\
\quad \text{ [] } \text{true} \rightarrow \text{coin} := 1 \\
\text{ endcoin}
\]

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

This question has two parts. Type your answers to both parts into a file \texttt{sol3_1.txt}, and submit this file.

(a) Extend the axiomatic semantics given in lecture by writing down an appropriate rule of inference (GIF) for \textit{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 \rightarrow y := -x \\
\[ x >= 0 \rightarrow y := x \end \\
end; \\
z := z - y \\
end
\{ z < 0 \}
\]

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.

2. Functions, Globals, Pairs

The remaining questions concern a simple language with imperative expressions, functions, global variables, and pairs, which we’ll call “E3.” Its “concretized” abstract syntax is given by the following grammar:

\[
prog := '(', \{ def \}, ')', exp \\
def := \text{globaldef} \mid \text{fundef} \\
\text{globaldef} := '(', 'global', var, exp, ')' \\
\text{fundef} := '(', 'fun', fname, '(', \{ var \}, ')', exp, ')' \\
exp := var \\
| \int \\
| '(', ':=', var, exp, ')' \\
| '(', 'while', exp, exp, ')' \\
| '(', 'if', exp, exp, exp, ')' \\
| '(', 'write', exp, ')' \\
| '(', 'block', \{ exp \}, ')' \\
| '(', '0', fname, \{ exp \}, ')' \\
| '(', '+', exp, exp, ')' \\
| '(', '-', exp, exp, ')' \\
| '(', '*', exp, exp, ')' \\
| '(', '/', exp, exp, ')' \\
| '(', '<=', exp, exp, ')' \\
| '(', 'pair', exp, exp, ')' \\
| '(', 'fst', exp, ')' \\
| '(', 'snd', exp, ')' \\
| '(', 'ispair', exp, ')' \\
fname := \text{letter} \{ \text{letter} \mid \text{digit} \} \\
var := \text{letter} \{ \text{letter} \mid \text{digit} \}
\]

As before, comments may be included by enclosing them between ‘{’ and ‘}’ characters, and they may be nested. hw The informal semantics of E3 programs is as follows. Values include integers and pairs, each of which has a left and a right component value. A program (\(d_1 \ldots \)

$d_n$) $e$ is evaluated by elaborating each definition $d_1, \ldots d_n$ in that order and then evaluating the top-level expression $e$, whose value is the program result. A global definition (global $x$ $e$) is elaborated by evaluating its initializing expression $e$ to a value $v$ and extending the variable environment with a binding from $x$ to $v$. A function definition is elaborated by recording the function name in an environment of available functions.

Functions and variables live in separate name spaces, so their names may overlap. The language uses a combination of static and dynamic scope rules. Function names are handled dynamically; the most recently elaborated definition with a matching name is used. Global variable names are also handled dynamically using a similar rule. But functions may have formal parameters, whose scope is statically limited to the body of the function. If a formal parameter has the same name as a global, the parameter hides the global. It is a checked runtime error to use an undefined function or variable name.

The semantics of E3 expressions are similar to those of E2, with the following extensions:

- A variable $x$ can refer to either a formal parameter or a global.
- Evaluating the function application expression (@ $f$ $e_1 \ldots e_n$) evaluates $e_1, \ldots e_n$ in that order, binds the resulting values to the $n$ formal parameters of function $f$, evaluates the body of $f$ in the resulting environment, and yields the resulting value. It is a checked runtime error if $f$ doesn’t exist or has the wrong number of parameters.
- Evaluating (pair $e_1$ $e_2$) evaluates $e_1$ and $e_2$ (in that order) to values $v_1$ and $v_2$, and yields a new pair whose left element is $v_1$ and whose right element is $v_2$.
- Evaluating (fst $e$) evaluates $e$ to a pair value, and extracts and yields the left element value. It is a checked runtime error is $e$ evaluates to a non-pair value.
- Evaluating (snd $e$) evaluates $e$ to a pair value, and extracts and yields the right element value. It is a checked runtime error is $e$ evaluates to a non-pair value.
- Evaluating (ispair $e$) evaluates $e$ and yields 1 if the result is a pair, 0 otherwise.
- The value tested by if or while must be an integer; otherwise, a checked runtime error results.
- The value written by write can be either an integer or a pair.
- The arithmetic operators (+, -, *, /,\leq) work only on integers; it is a checked runtime error to apply them on a pair.

An E3 interpreter in Java (only) has been provided (hw3.java). As usual, it reads a file containing an E3 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.

Write the following list-manipulation functions in E3, without using the while expression. Put both your function definitions and a test expression that exercises them in a single file sol3_2.e3 and submit that file. Some useful list manipulation code is in hw3_2.e3.
- \((\texttt{reverse} \ 1)\) returns a list containing the elements of 1 in reverse order. For example, 
  \((\texttt{reverse} \ (\texttt{list} \ 1 \ 2 \ 3))\) yields \((3. \ (2. \ (1. \ 0)))\). Hint: use \texttt{append}.

- \((\texttt{count} \ x \ 1)\) counts the number of occurrences of integer \(x\) in list 1. For example, 
  \((\texttt{count} \ 1 \ (\texttt{list} \ 1 \ 2 \ 1))\) yields 2.

3. Interpreter Changes

Make two modifications to the E3 interpreter, as described in (a) and (b) below. Submit just one modified interpreter, combining your answers to both parts, called \texttt{sol3.java}.

Also submit an input file \texttt{sol3.e3} as required by part (c).

(a) Modify the interpreter to support local variables, by adding a new expression form:

\[
\begin{align*}
\text{exp} &:= \ldots \\
&| \ ('\ 'local' '(', \{ \text{var exp} \}, ')') \exp ')
\end{align*}
\]

where the parenthesized list specifies a set of local variable names and associated initializing expressions.

The informal semantics of \((\texttt{local} \ (x_1 \ e_1 \ \ldots \ x_n \ e_n) \ e)\) is as follows: evaluate \(e_1, \ldots, e_n\) in that order, bind the resulting values to newly created local variables \(x_1, \ldots, x_n\) respectively, then evaluate \(e\) in the resulting environment, and yield the resulting value. (Don’t worry about what happens if two of the variables have the same name.)

The scope of the local variables is just the expression \(e\). If a local variable has the same name as a parameter or global, it hides the parameter or global.

For example, the program

\[
() \\
(\texttt{local} \ (a \ 1 \ b \ 10) \\
(\texttt{block} \\
(\texttt{local} \ (a \ 100) \\
(\texttt{block} \\
\quad (\texttt{:=} \ b \ (+ \ a \ b)) \\
\quad (\texttt{:=} \ a \ 0))) \\
\quad (+ \ a \ b)))
\]

should evaluate to 111.

The necessary parsing support is already present in \texttt{hw3.java}. All you have to do is add the AST, printing, and evaluation code for \texttt{local}. Hint: You don’t need to introduce a fourth environment component for local variables; just use the existing \texttt{vars} environment which currently holds parameters. Remember that \texttt{local} expressions can be nested.
(b) Add two new expression forms to E3:

\[
\text{exp} \ := \ \ldots \\
\quad | \ (\ 'setfst' \ \text{exp} \ \text{exp} \ ')
\]

\[
\quad | \ (\ 'setsnd' \ \text{exp} \ \text{exp} \ ')
\]

The informal semantics of these expressions is as follows. To evaluate \((\text{setfst} \ e_1 \ e_2)\), first evaluate \(e_1\) to a value \(v_1\), which must be a pair, then evaluate \(e_2\) to a value \(v_2\), then update the left component of \(v_1\) with \(v_2\), and yield the (updated) pair \(v_1\) as result. \((\text{setsnd} \ e_1 \ e_2)\) is similar, except that the right component of \(v_1\) is updated. For either expression, it is a checked runtime error if \(v_1\) is not a pair.

The necessary parsing support is already present in \texttt{hw3.java}. All you have to do is add new AST classes for these expressions. Note that the value printing code can now go into an infinite loop; don’t worry about this.

(c) Test your solution to part (b) by writing an E3 function \texttt{nreverse} that reverses a list \textit{in place}, that is, without constructing any new pairs. Hints: You’ll need \texttt{setsnd} but not \texttt{setfst}. It is helpful to define a two-argument auxiliary function (e.g., \texttt{nrevaux}) to do the real work. Submit a test program defining and using \texttt{nreverse} called \texttt{sol3_3.e3}.