CS 558 Homework 3 – due 6:00pm, Wednesday, Oct. 22, 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.

All the 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:

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

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

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 \((\text{global} \ x \ e)\) 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 the list 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 $(\mathit{f} e_1 \ldots e_n)$ 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 $(\mathit{pair} e_1 e_2)$ 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 $(\mathit{fst} e)$ evaluates $e$ to a pair value, and extracts and yields the left element value. It is a checked runtime error if $e$ evaluates to a non-pair value.
- Evaluating $(\mathit{snd} e)$ evaluates $e$ to a pair value, and extracts and yields the right element value. It is a checked runtime error if $e$ evaluates to a non-pair value.
- Evaluating $(\mathit{ispair} e)$ evaluates $e$ and yields 1 if the result is a pair, 0 otherwise.
- The value tested by $\mathit{if}$ or $\mathit{while}$ must be an integer; otherwise, a checked runtime error results.
- The value written by $\mathit{write}$ can be either an integer or a pair.
- The arithmetic operators ($+,-,*,$ and $/,$ $<=$) 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 $\mathit{write}$ expressions), and displays the evaluation result.

1. Write the following list-manipulation functions in E3, without using the $\mathit{while}$ expression. Put both your function definitions and a test expression that exercises them in a single file sol3_1.e3 and submit that file. Some useful list manipulation code is in hw3_1.e3.

- $(\mathit{reverse} 1)$ returns a list containing the elements of 1 in reverse order. For example, $(\mathit{reverse} (\mathit{list3} 1\ 2\ 3))$ yields $(3. (2. (1. 0)))$. Hint: use $\mathit{append}$.
- $(\mathit{count} x 1)$ counts the number of occurrences of integer $x$ in list 1. For example, $(\mathit{count} 1 (\mathit{list3} 1\ 2\ 1))$ yields 2.
2. Answer the following questions about E3 and its interpreter. There’s no need to write any code, just English. Put your solutions in a file `sol3.2.txt`.

(a) Under the current Java interpreter for E3, what happens if we give two functions the same name? Two globals? Two parameters of the same function? If we wanted to prohibit programmers from doing this (for any or all of these categories) should we treat it as a static or runtime error? Is there any difference between static and runtime errors in an interpreter?

(b) Explain how any E3 program that uses functions with multiple parameters can be systematically rewritten to use a combination of pairs and functions with exactly one parameter. (Thus, multiple parameters, although syntactically convenient, don’t add any fundamental power to the language.)

3. 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 `sol3.java`. Also submit an input file as required by part (c).

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

\[
\text{exp} := \ldots \\
| \text{local (} \{ \text{var exp} \} \text{) exp}
\]

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

The informal semantics of `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 expression

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

should evaluate to 111.

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

\[
\text{exp ::= } \ldots \\
| \ '(. \text{setfst}\ exp \ exp ')' \\
| \ '(. \text{setsnd}\ exp \ exp ')'\\n\]

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. \(\text{setsnd}\) 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}. 