CS 558 Homework 3 – due 2pm, Monday, February 4, 2013

The solutions to problems 1–3 must be submitted by mail to apt@cs.pdx.edu, with the subject line “CS558 HW3”. 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.

The solution to problem 4 must be submitted on paper in class. If you cannot attend class, make sure that the paper gets to the instructor’s mailbox by 2pm.

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:

```
prog := '(' { def } ')', exp
def := globaldef | fundef
globaldef := '(' 'global' var exp ')'
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 := letter { letter | digit }
var := letter { letter | digit }
```

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 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 \((\text{pair } e_1 e_2)\) 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 \((\text{fst } e)\) 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 \((\text{snd } e)\) 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 \((\text{ispair } e)\) evaluates \( e \) and yields 1 if the result is a pair, 0 otherwise.
- The value tested by \( \text{if} \) or \( \text{while} \) must be an integer; otherwise, a checked runtime error results.
- The value written by \( \text{write} \) can be either an integer or a pair.
- The arithmetic operators \((+, -, \ast, /, \leq, \leq\) work only on integers; it is a checked runtime error to apply them on a pair.

An E3 interpreter in Python (only) has been provided (\texttt{hw3.py}). 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 \texttt{write} expressions), and displays the evaluation result.

1. Pairs and Lists

Write the following list-manipulation functions in E3. Put both your function definitions and a test expression that exercises them in a single file \texttt{sol3_1.e3} and submit that file. Some useful list manipulation examples are in \texttt{lists.e3}. You may find it easier (or just more fun!) to write your solutions in recursive functional style (like the \texttt{append} example) rather than in imperative style (like \texttt{gen} and \texttt{length}).
• (@min 1) returns the minimum value in list 1. You can assume that 1 is a non-empty list of integers; i.e., any other argument may cause an unchecked runtime error. Hint: If you try a recursive functional solution, you’ll need to write an auxiliary function with two parameters.

• (@zip 11 12) returns a list obtained by pairing up the corresponding elements of 11 and 12. For example, (@zip (@list3 1 2 3) (@list3 4 5 6)) yields ((1.4).((2.5).((3.6).0))). Also, (@length (@zip 11 12)) = (@min (@list2 (@length 11) (@length 12))). Hint: The recursive functional style really works better here.

2. Local variables

Modify the E3 interpreter to support local variables, by adding a new expression form:

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

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 program

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

should evaluate to 111.

The necessary parsing support is already present in hw3.py. All you have to do is add the AST, printing, and evaluation code for local. Put your solution in a file sol3_2.py. 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.

3. Mutable pairs

(a) Further modify the E3 interpreter you produced for problem 2 by adding two new expression forms to the E3 language:

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

The informal semantics of these expressions is as follows. To evaluate (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. 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.

Again, the necessary parsing support is already present in hw3.py, but you will have to 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. Put your solution in a file sol3_3.py.

(b) Test your solution to part (b) by writing an E3 function nreverse that reverses a list in place, that is, without constructing any new pairs. Hint: You’ll need setsnd but not setfst. Again, your can use either a functional recursive style or an imperative style. Submit a test program defining and using nreverse called sol3_3.e3.

4. Formal Operational Semantics

Reminder: This problem must be submitted on paper, not emailed.

Consider the operational semantics rules presented in lecture.

(a) Write down a rule for the block expression form of E3. For simplicity, assume that each block has exactly two sub-expressions.

(b) Write down a rule for a new expression form (alias \(x_1\) \(x_2\) \(e\)), whose informal semantics is as follows: make the newly created local variable \(x_1\) an alias to the existing variable \(x_2\), evaluate \(e\) in the resulting environment, and yield the resulting value. Note that alias is not the same as local: when two variables are aliased, they refer to the same location. (Note: A simple variable aliasing form like this is not very useful, but many real languages introduce aliasing less directly — although this is often considered a bad feature, because it makes it easy to write confusing programs.)

(c) Write down the full derivation tree for the following judgment:

\[
\langle \text{local x 1 (block (alias y x (+ (:= x 3) (:= y 5))) x)),} \emptyset, \emptyset \rangle \Downarrow \langle 5, \emptyset \rangle
\]

Use symbolic names (e.g., \(l_1, l_2, \ldots\)) for locations. When writing concrete environments and stores, use set notation, e.g. \(\{x \mapsto l_x, y \mapsto l_y\}\) for the environment \(E\) where \(E(x) = l_x\) and \(E(y) = l_y\). Write \(\emptyset\) for the empty environment or store. (Hint: use a pencil and a broad sheet of paper. Your completed tree should have ten nodes.)