

# Assignment 6: Languages with contexts

## Sample Answers

1. Let  $K_0 = \text{begin var } A; \dots \text{end}$   
 $C_1 = A := 2; K_1$   
 $K_1 = \text{begin var } B; \dots \text{end}$   
 $C_2 = B := A+1$

$$\begin{aligned} & \mathbf{P}[\![K_0]\!] \\ &= \lambda l. \mathbf{K}[\![K_0]\!](\text{emptyenv } l) \\ &= \lambda l. \mathbf{K}[\![\text{begin var } A; C_1 \text{end}]\!](\text{emptyenv } l) \\ &= \lambda l. (\lambda e. \mathbf{C}[\![C_1]\!](\mathbf{D}[\![\text{var } A]\!]e))(\text{emptyenv } l) \\ &= \lambda l. \mathbf{C}[\![C_1]\!](\mathbf{D}[\![\text{var } A]\!](\text{emptyenv } l)) \end{aligned}$$

Taking the declaration first,

$$\begin{aligned} & \mathbf{D}[\![\text{var } A]\!](\text{emptyenv } l) \\ &= \mathbf{D}[\![\text{var } A]\!][\text{map}_0, l] \\ &= \text{let } [l', e'] = (\text{reserve-locn } [\text{map}_0, l]) \text{ in } \text{updateenv } [\![A]\!] \text{ inLocation}(l') e' \\ &= \text{updateenv } [\![A]\!] \text{ inLocation}(l) [\text{map}_0, l'], \text{ where } l' = \text{next-locn } l \\ &= [\text{map}_1, l'], \text{ where } \text{map}_1 = [[\![A]\!] \mapsto \text{inLocation}(l)] \text{map}_0 \end{aligned}$$

Executing  $C_1$  gives:

$$\begin{aligned} & \mathbf{C}[\![C_1]\!][\text{map}_1, l'] \\ &= \mathbf{C}[\![A := 2; K_1]\!][\text{map}_1, l'] \\ &= ((\text{check } \mathbf{C}[\![K_1]\!][\text{map}_1, l']) \circ (\mathbf{C}[\![A := 2]\!][\text{map}_1, l'])) \end{aligned}$$

If we execute  $A := 2$  first, we get

$$\begin{aligned} & \mathbf{C}[\![A := 2]\!][\text{map}_1, l'] \\ &= \lambda s. \text{cases } \text{accessenv } [\![A]\!][\text{map}_1, l'] \text{ of} \\ & \quad \text{isLocation}(l_1) \rightarrow (\text{cases } \mathbf{E}[\![2]\!][\text{map}_1, l'] s \text{ of} \\ & \quad \quad \text{isNat}(n) \rightarrow (\text{return}(\text{update } l_1 \ n \ s)) \\ & \quad \quad \square \text{isErrorvalue}() \rightarrow (\text{signalerr } s)) \\ & \quad \text{end}) \\ & \quad \text{isNat}(n) \rightarrow (\text{signalerr } s) \\ & \quad \square \text{isErrorvalue}() \rightarrow (\text{signalerr } s) \\ & \quad \text{end} \\ &= \lambda s. \text{return}(\text{update } l_1 \ \text{inNat}(\text{two}) \ s) \end{aligned}$$

This function is composed with the check of the execution of the block  $K_1$ :

$$\begin{aligned}
& \mathbf{C}[\mathbf{K}_1][map_1, l'] \\
&= \mathbf{K}[\mathbf{K}_1][map_1, l'] \\
&= \mathbf{C}[\mathbf{C}_2](\mathbf{D}[\mathbf{var} \ B][map_1, l']) \\
&= \mathbf{C}[\mathbf{C}_2](\text{let } [l', e'] = (\text{reserve} - \text{locn } [map_1, l']) \text{ in } \text{updateenv } [\mathbf{B}] \text{ inLocation } (l') e')^{\text{Execut}} \\
&= \mathbf{C}[\mathbf{C}_2](\text{updateenv } [\mathbf{B}] \text{ inLocation } (l') [map_1, l'']), \text{ where } l'' = \text{next} - \text{locn } l' \\
&= \mathbf{C}[\mathbf{C}_2][map_2, l''], \text{ where } map_2 = [[\mathbf{B}] \mapsto l'] map_1
\end{aligned}$$

Executing  $C_2$  gives

$$\begin{aligned}
& \mathbf{C}[\mathbf{B} := \mathbf{A} + 1][map_2, l''] \\
&= \lambda s. \text{cases } \text{accessenv } [\mathbf{B}][map_2, l''] \text{ of} \\
&\quad \text{isLocation}(l') \rightarrow (\text{cases } \mathbf{E}[\mathbf{A} + 1][map_2, l''] s \text{ of} \\
&\quad\quad \text{isNat}(n) \rightarrow (\text{return}(\text{update } l' n s)) \\
&\quad\quad \square \text{isErrorvalue}() \rightarrow (\text{signalerr } s) \\
&\quad\quad \text{end}) \\
&\quad \text{isNat}(n) \rightarrow (\text{signalerr } s) \\
&\quad \square \text{isErrorvalue}() \rightarrow (\text{signalerr } s) \\
&\quad \text{end}
\end{aligned}$$

Evaluating the expression  $A + 1$  gives

$$\begin{aligned}
& \mathbf{E}[\mathbf{A} + 1][map_2, l''] s \\
&= \text{cases } \mathbf{E}[\mathbf{A}][map_2, l''] s \text{ of} \\
&\quad \text{isNat}(n_1) \rightarrow (\text{cases } \mathbf{E}[1][map_2, l''] s \text{ of} \\
&\quad\quad \text{isNat}(n_2) \rightarrow \text{inNat}(n_1 \text{ plus } n_2) \\
&\quad\quad \square \text{isErrorvalue}() \rightarrow \text{inErrorvalue}() \\
&\quad\quad \text{end}) \\
&\quad \square \text{isErrorvalue}() \rightarrow \text{inErrorvalue}() \\
&\quad \text{end}
\end{aligned}$$

$$= \text{inNat}((\text{access } l s) \text{ plus one})$$

Thus the assignment  $B := A + 1$  gives

$$\begin{aligned}
& \mathbf{C}[\mathbf{B} := \mathbf{A} + 1][map_2, l''] \\
&= (\text{return}(\text{update } (l')((\text{access } l s) \text{ plus one}) s))
\end{aligned}$$

If we reassemble the composition and reattach the lambda  $l$ , we get

$$\lambda l. (\text{check}(\lambda s. (\text{return}(\text{update } (l')((\text{access } l s) \text{ plus one}) s))) \circ (\lambda s. \text{return}(\text{update } l \text{ two } s)))$$

which is the final denotation. If we apply this function to *zero*, and an empty store  $s_0$ , we can arrive at a final simplification

$\text{inOK}(\text{update } l_1 \text{ three}(\text{update } l_0 \text{ } s_0))$

2. The new definition of *Denotable-value* is

Domain *Denotable-value* = *Location* + *Nat* + *Errvalue* + *Proc*  
 where *Errvalue* = *Unit*, *Proc* = *Store* → *Poststore*<sub>⊥</sub>

The valuation function for declaring a procedure binds the name of the procedure to the execution of the procedure body in the same environment that is passed to *updateenv*. When the procedure is called, this is the environment that will be used, not the environment at the point of call. This is static scoping. The valuation function for calling a procedure is:

$$\begin{aligned} \mathbf{C}[\mathbf{call\ I}] &= \lambda e.\lambda s.\text{cases } (\text{accessenv}[\mathbf{I}]e) \text{ of} \\ &\quad \text{isLocation}(l) \rightarrow (\text{signalerr } s) \\ &\quad \square \text{isNat}(n) \rightarrow (\text{signalerr } s) \\ &\quad \square \text{isProc}(p) \rightarrow (\text{return}(p\ s)) \\ &\quad \square \text{isErrorvalue}() \rightarrow (\text{signalerr } s) \\ &\text{end} \end{aligned}$$

[As an example of its use, consider the program

```
begin
  var X;
  proc P = X := 1;
  begin
    var X;
    call P
  end
end.
```

The question is, which X gets assigned 1 by the body of P? The derivation makes it clear. Both the first X and the procedure P are added to the initial environment. The environment for the inner block is thus  $e_2$ , where

$$\begin{aligned} e_1 &= \text{updateenv}[\mathbf{X}] \text{inLocation}(l) [\text{map}_0, (\text{next-locn } l)] \\ e_2 &= \text{updateenv}[\mathbf{P}] \text{inProc}(\mathbf{C}[\mathbf{X := 1}]e_1)e_1 \end{aligned}$$

The declaration of X in the inner block creates a new environment

$$e_3 = \text{updateenv}[\mathbf{X}] \text{inlocation}(\text{next-locn } l)e_2$$

which is the environment for the procedure call. Clearly the new declaration overrides the old one. This is the stack discipline of block-structured languages. However, when P is accessed in this environment, it is bound to the original definition, which includes the environment  $e_1$ . The call is thus:

$$\lambda s.\text{return}(\mathbf{C}[\mathbf{X := 1}]e_1\ s)$$

When the assignment is executed the X that gets the value *one* is the X declared in the outer block.]

3.

a.

$$\begin{aligned} & \mathbf{E}[\text{LET } G = \text{LAMBDA } (X) X \text{ IN LET } G = \text{LAMBDA } (Y) (G Y) \text{ IN } (G a_0)] \\ &= \lambda e. \mathbf{E}[\text{LET } G = \text{LAMBDA } (Y) (G Y) \text{ IN } (G a_0)](\text{updateenv}[\mathbf{G}](\mathbf{E}[\text{LAMBDA } (X) X]e)e) \end{aligned}$$

The binding of G in the outer LET is

$$\begin{aligned} & \mathbf{E}[\text{LAMBDA } (X) X] \\ &= \lambda e. \text{inFunction}(\lambda d. \mathbf{E}[X](\text{updateenv}[X]d e)) \end{aligned}$$

giving an environment

$$e_1 = \text{updateenv}[\mathbf{G}](\text{inFunction}(\lambda d. \mathbf{E}[X](\text{updateenv}[X]d e)))e$$

The inner LET is then given this environment:

$$\begin{aligned} & \mathbf{E}[\text{LET } G = \text{LAMBDA } (Y) (G Y) \text{ IN } (G a_0)]e_1 \\ &= \mathbf{E}[(G a_0)](\text{updateenv}[\mathbf{G}](\mathbf{E}[\text{LAMBDA } (Y) (G Y)]e_1)e_1) \end{aligned}$$

The binding of G in  $e_1$  is overridden with the new one to produce

$$e_2 = \text{updateenv}[\mathbf{G}](\text{inFunction}(\lambda d. \mathbf{E}[G Y](\text{updateenv}[Y]d e_1)))e_1$$

Applying G to  $a_0$  in this environment produces

$$\begin{aligned} & \mathbf{E}[G a_0]e_2 = \text{cases } \mathbf{E}[G]e_2 \text{ of} \\ & \quad \text{isFunction}(f) \rightarrow f(\mathbf{E}[a_0]e_2) \\ & \quad \dots \\ & \quad \text{end} \end{aligned}$$

$$= (\lambda d. \mathbf{E}[G Y](\text{updateenv}[Y]d e_1))a_0$$

$$\begin{aligned} &= \text{cases } \mathbf{E}[G]e_3 \text{ of} \\ & \quad \text{isFunction}(f) \rightarrow f(\mathbf{E}[Y]e_3) \\ & \quad \dots \\ & \quad \text{end} \end{aligned}$$

where  $e_3 = \text{updateenv}[Y]a_0 e_1$

Accessing G in  $e_3$  gives  $\text{inFunction}(\lambda d. \mathbf{E}[X](\text{updateenv}[X]d e))$ , and accessing Y in  $e_3$  gives  $a_0$ , so we get

$$\begin{aligned} & (\lambda d. \mathbf{E}[X](\text{updateenv}[X]d e))a_0 \\ &= a_0, \text{ since we simply retrieve the binding for } X \end{aligned}$$

The final result is simply  $\lambda e. a_0$

$\llbracket$  LETREC LISTIFY = LAMBDA (L)  
 IFNULL L THEN NIL  
 b. ELSE (((HEAD L) CONS NIL) CONS (LISTIFY (TAIL L)))  
 IN LISTIFY (a<sub>0</sub> CONS (a<sub>1</sub> CONS NIL))  $\rrbracket$

We shall write the program as

LETREC LISTIFY = E<sub>1</sub> IN

LISTIFY (a<sub>0</sub> CONS (a<sub>1</sub> CONS NIL))

$\mathbf{E} \llbracket$  LETREC LISTIFY = E<sub>1</sub> IN LISTIFY (a<sub>0</sub> CONS (a<sub>1</sub> CONS NIL))  $\rrbracket$

=  $\lambda e. \mathbf{E} \llbracket$  LISTIFY (a<sub>0</sub> CONS (a<sub>1</sub> CONS NIL))  $\rrbracket$  (*fix* ( $\lambda e'. \text{updateenv} \llbracket$  LISTIFY  $\rrbracket$  ( $\mathbf{E} \llbracket$  E<sub>1</sub>  $\rrbracket$  e') e))

=  $\lambda e. \mathbf{E} \llbracket$  LISTIFY (a<sub>0</sub> CONS (a<sub>1</sub> CONS NIL))  $\rrbracket$  e<sub>1</sub>

where e<sub>1</sub> = *fix* ( $\lambda e'. \text{updateenv} \llbracket$  LISTIFY  $\rrbracket$  ( $\mathbf{E} \llbracket$  E<sub>1</sub>  $\rrbracket$  e') e)

The LAMBDA expression gives

$\mathbf{E} \llbracket$  E<sub>1</sub>  $\rrbracket$

=  $\lambda e. \text{inFunction}(\lambda d. \mathbf{E} \llbracket$  IFNULL...  $\rrbracket$  (*updateenv*  $\llbracket$  L  $\rrbracket$  d e))

Expanding the application, we get

$\mathbf{E} \llbracket$  LISTIFY (a<sub>0</sub> CONS (a<sub>1</sub> CONS NIL))  $\rrbracket$  e<sub>1</sub>

= cases  $\mathbf{E} \llbracket$  LISTIFY  $\rrbracket$  e<sub>1</sub> of

*isFunction*(f) → f( $\mathbf{E} \llbracket$  a<sub>0</sub> CONS (a<sub>1</sub> CONS NIL)  $\rrbracket$  e<sub>1</sub>)

...

end

To retrieve the binding of LISTIFY in e<sub>1</sub>, we need to use the fixed-point property

$\mathbf{E} \llbracket$  LISTIFY  $\rrbracket$  e<sub>1</sub>

=  $\mathbf{E} \llbracket$  LISTIFY  $\rrbracket$  (*fix*  $\lambda e'. \dots$ )

=  $\mathbf{E} \llbracket$  LISTIFY  $\rrbracket$  (( $\lambda e'. \dots$ ) (*fix*  $\lambda e'. \dots$ ))

=  $\mathbf{E} \llbracket$  LISTIFY  $\rrbracket$  (( $\lambda e'. \dots$ ) e<sub>1</sub>)

= *accessenv*  $\llbracket$  LISTIFY  $\rrbracket$  (*updateenv*  $\llbracket$  LISTIFY  $\rrbracket$  ( $\mathbf{E} \llbracket$  E<sub>1</sub>  $\rrbracket$  e<sub>1</sub>) e)

=  $\mathbf{E} \llbracket$  E<sub>1</sub>  $\rrbracket$  e<sub>1</sub>

= *inFunction* ( $\lambda d. \mathbf{E} \llbracket$  IFNULL...  $\rrbracket$  (*updateenv*  $\llbracket$  L  $\rrbracket$  d e<sub>1</sub>))

We can thus apply this function to the value of the constant list

$\mathbf{E} \llbracket$  a<sub>0</sub> CONS (a<sub>1</sub> CONS NIL)  $\rrbracket$  e<sub>1</sub>

= a<sub>0</sub> cons (a<sub>1</sub> cons nil)

to give

$$\mathbf{E}[\text{IFNULL...}] (\text{updateenv}[\mathbf{L}] (a_0 \text{ cons } (a_1 \text{ cons nil})) e_1)$$

Call the environment for evaluating the body  $e_2$ . Then

$$\begin{aligned} \mathbf{E}[\text{IFNULL L THEN NIL ELSE } E_2] e_2 = \\ \lambda e. \text{let } x = (\mathbf{E}[\mathbf{L}] e_2) \text{ in} \\ \text{cases } x \text{ of} \\ \text{isFunction}(f) \rightarrow \text{inError}() \\ \square \text{isList}(t) \rightarrow ((\text{null } t)) \rightarrow (\mathbf{E}[\text{NIL}] e_2 \square \mathbf{E}[E_2] e_2) \\ \square \text{isAtom}(a) \rightarrow \text{inError}() \\ \square \text{isError}() \rightarrow \text{inError}() \\ \text{end} \end{aligned}$$

where  $E_2 = (((\text{HEAD L}) \text{ CONS NIL}) \text{ CONS (LISTIFY (TAIL L))))$

Since L is bound to a non-null list in  $e_2$ , we evaluate  $E_2$ , which is the CONS of two expressions.

$$\begin{aligned} \mathbf{E}[\dots \text{ CONS } \dots] e_2 \\ = \text{let } x = \mathbf{E}[\text{LISTIFY (TAIL L)}] e_2 \text{ in} \\ \text{cases } x \text{ of} \\ \text{isList}(t) \rightarrow \text{inList}((\mathbf{E}[(\text{HEAD L}) \text{ CONS NIL}] e_2) \text{ cons } t) \\ \dots \\ \text{end} \end{aligned}$$

The evaluation of LISTIFY (TAIL L) is just like the previous evaluation except that the constant argument is different. TAIL L gives  $a_1 \text{ cons nil}$ . Then

$$\begin{aligned} \mathbf{E}[\text{LISTIFY (TAIL L)}] e_2 \\ = \mathbf{E}[\text{IFNULL...}] (\text{updateenv}[\mathbf{L}] (a_1 \text{ cons nil}) e_1) \end{aligned}$$

Again this results in the evaluation of  $E_2$  in the valuation function for IFNULL. This time the list passed to the recursive call is just  $nil$ , so the evaluation of  $E_1$  takes place, which is just NIL, resulting in the value  $\text{inList}(nil)$ . This takes care of the tails of the list being formed in the  $t$  in  $\text{inList}((\mathbf{E}[(\text{HEAD L}) \text{ CONS NIL}] e) \text{ cons } t)$

The heads of this list are

$\mathbf{E}[(\text{HEAD L}) \text{ CONS NIL}] e$ , where  $e$  is the environment containing the successive bindings for L. The first environment has L bound to  $\text{inList}(\text{inAtom}(a_0) \text{ cons } (\text{inAtom}(a_1) \text{ cons nil}))$ , so the expression evaluates to  $\text{inList}(\text{inAtom}(a_0) \text{ cons nil})$ . The second environment binds L to  $\text{inList}(\text{inAtom}(a_1) \text{ cons nil})$ , so the evaluation is  $\text{inList}(\text{inAtom}(a_1) \text{ cons nil})$ . The third time L is bound to  $\text{inList}(nil)$ , so the evaluation ends. The final result, collecting all the conses together is  $\text{inList}((\text{inAtom}(a_0) \text{ cons nil}) \text{ cons } ((\text{inAtom}(a_0) \text{ cons nil}) \text{ cons nil}))$

c.

$$\begin{aligned} & \mathbf{E}[\text{LETREC } L = \text{HEAD } L \text{ IN } L] \\ &= \lambda e. \mathbf{E}[L] \left( \text{fix} \left( \lambda e'. \text{updateenv}[L] \left( \mathbf{E}[\text{HEAD } L] e' \right) e \right) \right) \end{aligned}$$

$$\text{Let } e_1 = \text{fix} \left( \lambda e'. \text{updateenv}[L] \left( \mathbf{E}[\text{HEAD } L] e' \right) e \right)$$

Then, using the fixed-point property

$$\begin{aligned} & \mathbf{E}[L] e_1 \\ &= \mathbf{E}[L] \left( \text{fix}(\lambda e'. \dots) \right) \\ &= \mathbf{E}[L] \left( (\lambda e'. \dots) \left( \text{fix}(\lambda e'. \dots) \right) \right) \\ &= \mathbf{E}[L] \left( (\lambda e'. \dots) e_1 \right) \\ &= \mathbf{E}[L] \left( \text{updateenv}[L] \left( \mathbf{E}[\text{HEAD } L] e_1 \right) \right) \\ &= \mathbf{E}[\text{HEAD } L] e_1 \end{aligned}$$

This evaluation is

$$\mathbf{E}[\text{HEAD } L] e_1 =$$

$$\text{let } x = \mathbf{E}[L] e_1 \text{ in}$$

cases  $x$  of

$$\text{isFunction}(f) \rightarrow \text{inError}()$$

$$\square \text{isList}(t) \rightarrow (\text{null } t \rightarrow \text{nError}() \square (\text{hd } x))$$

$$\square \text{isAtom}(a) \rightarrow \text{inError}()$$

$$\square \text{inError}() \rightarrow \text{inError}()$$

end

Clearly this expansion will go on for ever. We will never get to test the type of the value of  $L$  because its computation never terminates. We have created an infinite recursion in the derivation, just as we could create an infinite derivation for the while loop.

## Language 2

### Abstract Syntax

$E \in \text{Expression}$

$A \in \text{Atomic-symbol}$

$I \in \text{Identifier}$

$E ::= \text{LET } I = E_1 \text{ IN } E_2 \mid \text{LAMBDA } (I) E \mid E_1 E_2 \mid$   
 $\text{IFNULL } E_1 \text{ THEN } E_2 \text{ ELSE } E_3 \mid \text{LETREC } I = E_1 \text{ IN } E_2 \mid$   
 $I \mid A \mid (E) \mid$   
 $E_1 \text{ CONS } E_2 \mid \text{HEAD } E \mid \text{TAIL } E \mid \text{NIL}$

### Semantic algebras

Domain  $Atom = (\text{infinite set of distinguishable objects})$

Domain  $Denotable\text{-value} = (Function + List + Atom + Error)_{\perp}$

where  $Function = Denotable\text{-value} \rightarrow Denotable\text{-value}$ ,

$List = Denotable\text{-value}^*$ ,

$Error = Unit$

Domain  $Environment = Id \rightarrow Denotable\text{-value}$

$accessenv : Id \rightarrow Environment \rightarrow Denotable\text{-value}$

$accessenv = \lambda i. \lambda e. e(i)$

$updateenv : Id \rightarrow Denotable\text{-value} \rightarrow Environment \rightarrow Environment$

$updateenv = \lambda i. \lambda d. \lambda e. [i \mapsto d]e$

Domain  $Denotable\text{-value}^*$

Operations

$nil : Denotable\text{-value}$

$nil = \text{inUnit}()$

$hd : Denotable\text{-value}^* \rightarrow Denotable\text{-value}$

$hd[x, [\dots]] = x$

$tl : Denotable\text{-value}^* \rightarrow Denotable\text{-value}^*$

$tl[x, y] = y$

$cons : Denotable\text{-value} \times Denotable\text{-value}^* \rightarrow Denotable\text{-value}^*$

$x \text{ cons } y = [x, y]$

## Valuation functions

$E : \text{Expression} \rightarrow \text{Environment} \rightarrow \text{Expressible-value}$

where *Expressible-value* = *Denotable-value*

$E[\text{LET } I = E_1 \text{ IN } E_2] = \lambda e. E[E_2](\text{updateenv}[I](E[E_1]e)e)$

$E[\text{LAMBDA } (I) E] = \lambda e. \text{inFunction}(\lambda d. E[E](\text{updateenv}[I]d e))$

$E[E_1 E_2] = \lambda e. \text{cases } E[E_1]e \text{ of}$

$\text{isFunction}(f) \rightarrow f(E[E_2]e)$

$\square \text{isList}(t) \rightarrow \text{inError}()$

$\square \text{isAtom}(a) \rightarrow \text{inError}()$

$\square \text{isError}() \rightarrow \text{inError}()$

end

$E[\text{IFNULL } E_1 \text{ THEN } E_2 \text{ ELSE } E_3] =$

$\lambda e. \text{let } x = (E[E_1]e) \text{ in}$

cases  $x$  of

$\text{isFunction}(f) \rightarrow \text{inError}()$

$\square \text{isList}(t) \rightarrow ((\text{null } t)) \rightarrow (E[E_2]e \square E[E_3]e)$

$\square \text{isAtom}(a) \rightarrow \text{inError}()$

$\square \text{isError}() \rightarrow \text{inError}()$

end

$E[\text{LETREC } I = E_1 \text{ IN } E_2] = \lambda e. E[E_2](\text{fix } (\lambda e'. \text{updateenv}[I](E[E_1]e')e))$

$E[E_1 \text{ CONS } E_2] = \lambda e. \text{let } x = E[E_2]e \text{ in}$

cases  $x$  of

$\text{isFunction}(f) \rightarrow \text{inError}()$

$\square \text{isList}(t) \rightarrow \text{inList}((E[E_1]e) \text{ cons } t)$

$\square \text{isAtom}(a) \rightarrow \text{inError}()$

$\square \text{inError}() \rightarrow \text{inError}()$

end

$$\mathbf{E}[\text{HEAD } E] = \lambda e. \text{let } x = \mathbf{E}[E] e \text{ in}$$

cases  $x$  of

- $\text{isFunction}(f) \rightarrow \text{inError}()$
- $\square \text{isList}(t) \rightarrow (\text{null } t \rightarrow \text{nError}() \square (\text{hd } x))$
- $\square \text{isAtom}(a) \rightarrow \text{inError}()$
- $\square \text{inError}() \rightarrow \text{inError}()$

end

$$\mathbf{E}[\text{NIL}] = \lambda e. \text{inList}(\text{nil})$$

$$\mathbf{E}[I] = \text{accessenv}[I]$$

$$\mathbf{E}[A] = \lambda e. \text{inAtom}(\mathbf{A}[A])$$

$$\mathbf{E}[(E)] = \mathbf{E}[E]$$