Issac Zheng CS 496-1A Bonelli I pledge my honor that I have abided by the Stevens Honor System

Evaluation Rules

$$\frac{}{\texttt{Proc(id,e)},\rho \Downarrow (\texttt{id},\texttt{e},\rho)}\ \text{EProc}$$

$$\frac{\texttt{e1},\rho \Downarrow (\texttt{id},\texttt{e},\sigma) \quad \texttt{e2},\rho \Downarrow w \quad \texttt{e},\sigma \oplus \{\texttt{id} := w\} \Downarrow v}{\texttt{App(e1,e2)},\rho \Downarrow v}\ \text{EApp}$$

$$\frac{\texttt{e1},\rho \Downarrow v \quad v \notin \mathbb{CL}}{\texttt{App(e1,e2)},\rho \Downarrow \mathit{error}}\ \text{EAppErr}$$

$$\frac{\texttt{e2},\rho \oplus \{\texttt{id} := (\texttt{par},\texttt{e1},\rho)^r\} \Downarrow v}{\texttt{Letrec([(id,par,\_,\_,e1)],e2)},\rho \Downarrow v}\ \text{ELetRec}$$

$$\frac{}{\texttt{Debug(e)},\rho \Downarrow \mathit{error}}\ \text{EDebug}$$

$$\frac{\rho(\texttt{id}) = (\texttt{par},\texttt{e},\sigma)^r}{\texttt{Var(id)},\rho \Downarrow (\texttt{par},\texttt{e},\sigma \oplus \{\texttt{id} := (\texttt{par},\texttt{e},\sigma)^r\})}\ \text{EVarLetRec}$$

itional evaluation rules for REC

$\texttt{e},\rho \Downarrow r$ Evaluation judgement

$\Gamma \vdash \texttt{e} : t$ Typing judgement

Additional Evaluation rules for PROC (error propagation rules o

$$\frac{}{\texttt{Int(n)},\rho \Downarrow n}\ \text{EInt} \qquad \frac{\rho(\texttt{id}) = v}{\texttt{Var(id)},\rho \Downarrow v}\ \text{EVar}$$

$$\frac{\texttt{e1},\rho \Downarrow m \quad \texttt{e2},\rho \Downarrow n \quad n \neq 0 \quad p = m/n}{\texttt{Div(e1,e2)},\rho \Downarrow p}\ \text{EDiv}$$

$$\frac{\texttt{e},\rho \Downarrow 0}{\texttt{IsZero(e)},\rho \Downarrow \mathit{true}}\ \text{EIZTrue} \qquad \frac{\texttt{e},\rho \Downarrow m \quad m \neq 0}{\texttt{IsZero(e)},\rho \Downarrow \mathit{false}}\ \text{EIZFalse}$$

$$\frac{\texttt{e1},\rho \Downarrow \mathit{true} \quad \texttt{e2},\rho \Downarrow v}{\texttt{ITE(e1,e2,e3)},\rho \Downarrow v}\ \text{EITETrue} \qquad \frac{\texttt{e1},\rho \Downarrow \mathit{false} \quad \texttt{e3},\rho \Downarrow v}{\texttt{ITE(e1,e2,e3)},\rho \Downarrow v}\ \text{EITEFalse}$$

$$\frac{\texttt{e1},\rho \Downarrow w \quad \texttt{e2},\rho \oplus \{\texttt{id} := w\} \Downarrow v}{\texttt{Let(id,e1,e2)},\rho \Downarrow v}\ \text{ELet}$$

$$\frac{\texttt{id} \notin \mathrm{dom}(\rho)}{\texttt{Var(id)},\rho \Downarrow \mathit{error}}\ \text{EVarErr} \qquad \frac{\texttt{e1},\rho \Downarrow m \quad \texttt{e2},\rho \Downarrow 0}{\texttt{Div(e1,e2)},\rho \Downarrow \mathit{error}}\ \text{EDivErr}$$

$$\frac{\texttt{e},\rho \Downarrow v \quad v \notin \mathbb{Z}}{\texttt{IsZero(e)},\rho \Downarrow \mathit{error}}\ \text{EIZErr} \qquad \frac{\texttt{e1},\rho \Downarrow v \quad v \notin \mathbb{B}}{\texttt{ITE(e1,e2,e3)},\rho \Downarrow \mathit{error}}\ \text{EITEErr}$$

Evaluation Semantics for LET (error propagation rules omitted)

$$\frac{}{\texttt{Int(n)} \Downarrow n}\ \text{EInt}$$

$$\frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow n \quad p = m - n}{\texttt{Sub(e1,e2)} \Downarrow p}\ \text{ESub} \qquad \frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow n \quad n \neq 0 \quad p = m/n}{\texttt{Div(e1,e2)} \Downarrow p}\ \text{EDiv}$$

$$\frac{\texttt{e1} \Downarrow \mathit{error}}{\texttt{Sub(e1,e2)} \Downarrow \mathit{error}}\ \text{ESubErr1} \qquad \frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow \mathit{error}}{\texttt{Sub(e1,e2)} \Downarrow \mathit{error}}\ \text{ESubErr2}$$

$$\frac{\texttt{e1} \Downarrow \mathit{error}}{\texttt{Div(e1,e2)} \Downarrow \mathit{error}}\ \text{EDivErr1} \qquad \frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow \mathit{error}}{\texttt{Div(e1,e2)} \Downarrow \mathit{error}}\ \text{EDivErr2} \qquad \frac{\texttt{e1} \Downarrow m \quad \texttt{e2} \Downarrow 0}{\texttt{Div(e1,e2)} \Downarrow \mathit{error}}\ \text{EDivErr3}$$

Figure 2.1: Evaluation rules for ARITH

Derivation

$$\dfrac{\dfrac{\dfrac{}{\mathit{Int\ 4} \Downarrow 4}\ \text{EInt} \quad \dfrac{}{\mathit{Int\ 2} \Downarrow 2}\ \text{EInt}}{\mathit{Div(Int\ 4,Int\ 2)} \Downarrow 2}\ \text{EInt} \quad \dfrac{}{\mathit{Int(1)} \Downarrow 1}\ \text{EInt} \quad 1 = 2 - 1}{\mathit{Sub(Div(Int\ 4,Int\ 2),Int\ 1)} \Downarrow 1}\ \text{ESub}$$

An example of an evaluation judgement that is not derivable is $\mathit{Sub(Int\ 3,Int\ 1)} \Downarrow 1$.

Derivation

let x = 1 in if zero?(x) then 1 else 2

{x:=1}(x)=1 ---------- EVar

Var("x"), {x:=1} \||/ 1 1!=0 ---------- EIZFalse ---------- EInt

IsZero(Var "x"), {x:=1} \||/ false Int 2, {x:=1} \||/ 2

---------- EInt ---------- EITEFalse

Int 1, {} \||/ 1 ITE(IsZero(Var "x", Int 1, Int 2)), {x:=1} \||/ 2

---------- ELet

Let("x", Int 1, ITE(IsZero(Var "x"), Int 1, Int 2)), {} \||/ 2

EInt and EVar are axioms.

Evaluating Code

Rec

```
letrec add(n) = proc (m) { if zero?(n)  then m else 1 + ((add (n-1)) m) } in ((add 2) 3) #=> Numval 5
interp: Letrec ([("add", "n", None, None,
          Proc ("m", None,
           ITE (IsZero (Var "n"), Var "m",
           Add (Int 1, App (App (Var "add", Sub (Var "n", Int 1)), Var "m"))))))],
          App (App (Var "add", Int 2), Int 3))
```

Debug (REC)

let x = 5 in let true = zero?(0) in letrec fact(x) = if zero?(x) then 1 else x*(fact (x-1)) in debug(5);;

#=> Environment:

[x:=NumVal 5,

true:=BoolVal true,

fact:=Rec(x,ITE(Zero?(Var x),Int 1,Mul(Var x,App(Var fact,Sub(Var x,Int 1))))))]

Evaluating Code

Proc

```
- let f=proc (x) { proc (y) { x + y } }
  in ((f 2) 3);; #=> Ok (NumVal 5)
- let f=proc (x) { proc (y) { x + y } }
  in (f 2);; #=> ProcVal ("y", Add (Var "x", Var "y"),
                    ExtendEnv ("x", NumVal 2, EmptyEnv))
- let f=proc (x) { x + 1 }
  in let g=proc (y) { y + 2 }
  in g;; #=> ProcVal ("y", Add (Var "y", Int 2),
                ExtendEnv ("f", ProcVal ("x",
                Add (Var "x", Int 1), EmptyEnv), EmptyEnv))
- let f = proc (x) { x -11 }
  in ( f (f 77)) #=> Ok (NumVal 55)
  ( proc (f) { (f ( f 77)) } proc (x) { x -11 }) #=> Ok (NumVal 55)
- let pred = proc(x) { x-1 } in (pred 5) #=> Ok (NumVal 4)
  parse: Let ("pred", Proc ("x", None, Sub (Var "x", Int 1)),
                    App (Var "pred", Int 5)))
- let f=(let b=2 in proc (x) { x }) in f #=> ProcVal ("x", Var "x",
                    ExtendEnv ("b", NumVal 2, EmptyEnv))
  parse: Let ("f", Let ("b", Int 2, Proc ("x", None, Var "x")), Var "f")
          (*middle param of Proc() will always be None rn*)
- let x = 2
  in let f = proc (z ) { z - x }
  in let x = 1
  in let g = proc (z ) { z - x }
  in ( f 1) - (g 1)
   - x in f = 2, while x in g = 1. the result = -1 (statically scoped)
   - If we let x override the value of x in f, it's dynamically scoped.
```

Notes

- In an environment, the lookup of the operation starts from THE END. Bottom-most var in env is most recent.
- Parser turns flat syntax into a tree.
- Debug() dumps environment at point of call.

```
let rec eval_expr : expr -> exp_val ea_result = fun e ->
  match e with
  (* sequence: ('a ea_result) list -> ('a list) ea_result *)
  (* extend_env_list: string list -> exp_val list -> env ea_result *)
  | Record(fs) -> (* there's a bool before the value that we can disregard. *)
    sequence (List.map (fun (_,(_,e)) -> eval_expr e) fs) >>= fun evs ->
    let ids = List.map (fun (id,(_,_)) -> id) fs in (* need id to evaluate to string *)
    extend_env_list ids evs >>+
    return @@ RecordVal (List.combine ids evs)
  | Proj(e,id) ->
    eval_expr e >>=
    fields_of_recordVal >>= fun f ->
    (match List.assoc_opt id f with
    | None -> error "Proj: field does not exist"
    | Some v -> return v
    )
  | IsEmpty(e1)  ->
    eval_expr e1 >>=
    tree_of_treeVal >>= fun t ->
    return @@ BoolVal (t = Empty)
  | EmptyTree(_) ->
    return @@ TreeVal(Empty)
  | Node(e1,lte,rte) ->
    eval_expr e1 >>= fun v ->
    eval_expr lte >>=
    tree_of_treeVal >>= fun lnode ->
    eval_expr rte >>=
    tree_of_treeVal >>= fun rnode ->
    return @@ TreeVal(Node(v,lnode,rnode))
  | CaseT(target,emptycase,id1,id2,id3,nodecase) ->
    eval_expr target >>=
    tree_of_treeVal >>= fun t ->
    (match t with
    | Empty -> eval_expr emptycase
    | Node(v,l,r) ->
      extend_env id1 v >>+
      extend_env id2 (TreeVal(l)) >>+
      extend_env id3 (TreeVal(r)) >>+
      eval_expr nodecase
    ) (*match for trees: target match empty/node*)
      (*extend env with the node and return nodecase*)
and
  eval_exprs : expr list -> (exp_val list) ea_result  =
  fun es ->
  match es with
  | [] -> return []
  | h::t -> eval_expr h >>= fun i ->
    eval_exprs t >>= fun l ->
    return (i::l)
```

```
match e with
  | Int(n) -> return (NumVal n)
  | Var(id) -> apply_env id
  | Add(e1,e2) -> ...
  | Sub(e1,e2) -> ...
  | Mul(e1,e2) -> ...
  | Div(e1,e2) ->...
  | Let(v,def,body) ->
    eval_expr def >>=
    extend_env v >>+
    eval_expr body
  | ITE(e1,e2,e3) ->
    eval_expr e1 >>=
    bool_of_boolVal >>= fun b ->
    if b then eval_expr e2
    else eval_expr e3
  | IsZero(e) -> ...
  | Pair(e1,e2) -> ...
  | Fst(e) -> ...
  | Snd(e) -> ...
  | Proc(id,_,e)  -> ...
    lookup_env >>= fun en ->
    return (ProcVal(id,e,en))
  | App(e1,e2)  ->
    eval_expr e1 >>=
    clos_of_procVal >>= fun (id,e,en) ->
    eval_expr e2 >>= fun ev ->
    return en >>+
    extend_env id ev >>+
    eval_expr e
  | Letrec([(id,par,_,_,e1)],e2) ->
    extend_env_rec id par e1 >>+
    eval_expr e2
  | Debug(_e) ->
    string_of_env >>= fun str ->
    print_endline str;
    error "Debug called"
  | _ -> failwith ("Not implemented yet!"^string_of_expr e)

type exp_val =
  | NumVal of int
  | BoolVal of bool
  | UnitVal
  | PairVal of exp_val*exp_val
  | ProcVal of string*expr*env
and
  env =
  | EmptyEnv
  | ExtendEnv of string*exp_val*env
  | ExtendEnvRec of string*string*expr*env

let int_of_numVal : exp_val -> int ea_result =  function
  |  NumVal n -> return n
  | _ -> error "Expected a number!"
```

Extra Notes

```
let (>>=) (c:'a ea_result) (f: 'a -> 'b ea_result) : 'b ea_result =
  fun env ->
  match c env with
  | Error err -> Error err
  | Ok v -> f v env
(*>>+ produces  value from env, >>+ produces env from value*)
let (>>+) (c:env ea_result) (d:'a ea_result): 'a ea_result =
  fun env ->
  match c env with
  | Error err -> Error err
  | Ok newenv -> d newenv
```

DO NOT PRINT THIS PAGE

- Derivation
- Result of evaluating code in PROC
- Result of evaluating code in REC
- debug question (typically in REC).
- Simple extension to LET.
  a. The homework pertains to this question
  b. code the interpreter
  c. example: extend with record, trees, etc
  d. we are not asked to write evaluation rules, if asked to write a derivation
     we will always be given the evaluation rules
  e. won't be asked to write an evaluation rule for a new program/feature

Write:

- which expression each evaluation rule is associated with
- EmptyEnv under evaluating code is for extendenv
- Add notes for App().