EB7

• Exercise 1: Assume pass by REF

```
let b=3
in let p=proc(x)proc(y)
     begin
       set x=4;
       y
     end
   in ((p b) b)
```

What is the result? (num-val 4)

ENV:

b	0
p	1
x	0
y	0

DRAW TRACE:

0	(num-val 3) (num-val 4)
1	(Proc-val (closure 'x (proc-exp 'y (begin-exp (set-exp ((var-exp 'x) (const-exp 4)) (var-exp 'y)

b 0

• Exercise 2.

```
letrec inf(x)=(inf x)
 in let f=proc(z)11
  in (f (inf 0))
```

VALUE	REF	NAME	NEED
Inf loop NO OUTPUT	Inf loop NO OUTPUT	11	11

Results of execution

CallBy	NAME		NEED	
Input	var	func	var	func
What is passed	REF	THUNK	REF	THUNK

NAME - Thunk runs everytime called.
NEED - thunk runs once, value stored for later use.

A function passed by REF and by VALUE runs immediately after pass.

• Exercise 4. Difference between a thunck and closure: A thunk has no formal parameters.

• Exercise 5 Trace by ref and by need.

```
let a=3
in let g=proc(x)
         Proc(y)
           if zero?(x)
           then y
           else -(x,y)
   in((g a)a)
```

→ Input is variable

So trace of ref and need are the same.

ENV.

a	0
g	1
x	0
y	0

STORE

0	(num-val 3)
1	@Proc-val (closure 'x (proc-exp (if-exp (zero?-exp var-exp 'x) (var-exp 'y) (diff-exp (var-exp 'x) (var-exp 'y)))))

RETURNS 0

Both ref and Need.

• Exercise 3 Write a program that test call by name and need - the same program as Ex5 but instead with else -(y, -(x, y)).

• Exercise 6 Trace REF and NEED

```
let a=3
in let g=proc(x)
         proc(y)
           if zero?(x)
           then y
           else -(x,y)
   in((g a) -(a,1))
```

→ Input is func so will be thunked in NEED.

REF

ENV:

g	0
x	0
y	2

STORE:

0	(n-v 3)
1	Proc...
2	(n-v 2) func runs immediately

NEED

ENV:

a	0
x	0
y	2

STORE

0	(n-v 3)
1	Proc
2	(A-thunk (diff-exp ...

• Exercise 8 Extend IMPLICIT-REFS by adding expression <Expression> ::= ref <identifier>

Note we also need deref and setref.

```
let a=3
in let b=4
   in let swap=proc(x)proc(y)
          let temp=deref(x)
          in begin
               setref(x,deref(y));
               setref(y,temp) end
   in begin ((swap ref a) ref b);
            -(a,b) end
```

Expressed
Expval = int + bool + proc
Denoted
Denval = Ref(Expval)

LANG (expression ("setref" "(" "expression" ")") Setref-exp) deref is the same only deref.

LANG (expression ("ref" identifier) ref-exp)

EB8

- Exercise 5 What is the result of evaluating the following expressions in CHECKED

((check "letrec int double (x:int) = if zero?(x) then 0 else -((double -(x,1)),-2) in (double 5)") Returns int modify:in double") Returns int -> int
modify: letrec bool double Returns type error.

- Exercise 2 Argue that the expression x x (a variable applied to itself) is not typable.

$$\frac{tenv \vdash x :: t1 \to t2 \quad tenv \vdash x :: t1}{tenv \vdash (x\ x) :: t2}\ \text{TProcApp}$$

* x can not have both types t1→t2 and t1.

- Exercise 6
Extend the checker to include pair and unpair

CHECKER

```
(pair-exp (e1 e2)
  (let ((t1 (type-of e1 tenv))
        (t2 (type-of e2 tenv)))
    (pair-type t1 t2)))

(unpair-exp (id1 id2 exp1 body)
  (let ((t-pair (type-of exp1 tenv)))
    (cases type t-pair
      (pair-type (t1 t2) (type-of body (extend-tenv
                                         id1 t1 (extend-tenv
                                                  id2 t2 tenv))))
      (else (write "error")))))
```

TOP

```
unpair(id1, id2) =
  exp1 in body
  ↑
  must be a pair
```

LANG

```
(expression
 ("unpair" "(" identifier "," identifier ")" "=" expression
  "in" expression)
 unpair-exp)

(type
 ("<" type "*" type ">")
  pair-type)
```

} Adding Pair types.

DATA-STRUCTURES

```
(pair-val
  (1st exp-val?)
  snd exp-val?))
```

- Exercise 3 typable term

(bool→bool) (proc (x:bool) 2)

((bool→int)→int) (proc (x:(bool→int)) 2)

((s→t)→(s→t)) ∀s,t
(proc (x:s→t) x)

Expression that is untypable

(4 5) - TprocApp OR
if 5 then 1 else 2

Exercise 1.6

tenv1 = {x::int}∅
tenv2 = {two?:int→int}∅

tenv1+x:int+tenv1+2:int — Tdiff
tenv1+-(x,2)::int — Tzero? — X — Tconst
tenv1+zero?(-(x,2))::bool — X — Tconst
tenv1+if zero?(-(x,2)) then 0 else 1::int — Tif
[x=int]∅+if zero?(-(x,2)) then 0 else 1::int — Tproc
∅+Proc(x:int) if zero?(-(x,2)) then 0 else 1::int→int — [two?=int→int]∅ + (two? 3)::int — TLet
∅+let two?=proc(x:int) if zero?(-(x,2)) then 0 else 1 in (two? 3)::int

tenv(x)=int — Tvar — X — Tconst
tenv1+x:int+tenv1+2:int — Tdiff
tenv1+-(x,2)::int — Tzero? — X — Tconst
tenv1+zero?(-(x,2))::bool tenv1+0::int tenv1+1::int — Tif
[x=int]∅+if zero?(-(x,2)) then 0 else 1::int — Tproc
∅+Proc(x:int) if zero?(-(x,2)) then 0 else 1::int→int — [two?=int→int]∅ + (two? 3)::int — TLet
∅+let two?=proc(x:int) if zero?(-(x,2)) then 0 else 1 in (two? 3)::int

tenv2+two?::int→int tenv2+3::int — Tvar — X — Tconst — TProcApp