Languages

LET: diff-exp: -(x, y)
zero?-exp: zero?(x)
if-exp: if — then — else —
let-exp: let x = 3 in —

PROC: LET +
Proc-exp: let f = proc(x) -(x,1) in (f 3)

REC: PROC +
letrec-exp: letrec f(x) = proc(x) if zero?(x) then 1 else * (x,(f -(x,1))) in (f 4)

IMPLICIT-REFS:
REC +
- env holds memory addresses
- store holds values @ Memory address

```
let f= proc(x) x
in begin
     set f =proc(x) if zero? (x)
                    then 1
                    else *(x,(f -(x, 1)))
   (f 4)
   end
```

EXPLICIT-REFS:
REC + env & store
- newref(n), deref(x), setref(x,n)
  n is a number, x is a name.

```
let counter = newref(0)
in —
```

Quiz 2

Question 1:

```
1. let x=2
2. in let f= proc(x) x
3.    in let g = proc(y) x
4.       in 3
```

Draw environment: (before line 4)

x	(num-val 2)
f	(proc-val (closure 'x (var-exp x) [x | (num-val 2)]))
g	(proc-val (closure 'y (var-exp x) [x | (num-val 2) ; f | (proc-val (...))]))

Question 2:

```
let x=2
in let f = proc(x) proc(y) -(x, y)
   in f
```

result:

```
(proc-val
   (closure
      'x
      (proc-exp 'y
                (diff-exp (var-exp x) (var-exp y)))
      [x | (num-val 2)] ))
```

Inductive Sets & Derivations

Ex 1:

<NAT> ::= 0 | S(<NAT>)

S(S(0)) in <NAT>? YES

<NAT> → S(<NAT>)
→ S(S(<NAT>))
→ S(S(0))

Ex 2:

<BTree> ::= leaf(<NAT>)
| node(<BTree>, <BTree>)

node(leaf(s(0)), leaf(s(s(0)))) in <BTree>? YES

<BTree> → node(<BTree>, <BTree>)
→ node(leaf(<NAT>), <BTree>)
→ node(leaf(S(0)), leaf(<NAT>))
→ node(leaf(S(0)), leaf(S(S(0))))

Back patching

Factorial

```
1. let F = proc(x) x
   in begin
2.    set F = proc(x) if zero?(x)
                      then 1
                      else *(x,(f -(x,1)))
3.  (f 4)
    end
```

env: [F | 0]

store:

After 1.

0	(proc-val (closure 'x (var-exp x) [ ]))

After 2.

0	(proc-val (closure 'x (if-exp (zero?-exp (var-exp x)) (num-val 1) (mult-exp ...) [f | 0])))

Good Luck!

Make this test your BITCH!

Implicit vs Explicit References

IMPLICIT

```
let g = let count = 0
        in proc(d)
            begin set count = -(count,-1);
                  count
            end
in -((g 11), (g 22))
```

EXPLICIT

```
let g = let counter = newref(0)
        in proc (d)
            begin
               setref(counter, -(deref(counter)
               deref(counter)                  -1));
            end
in -((g 11),(g 22))
```

a=3, b=4, c=12 } example

a	0
b	1
c	2

IMPLICIT ENV

STORE

0	num-val 3
1	num-val 4
2	num-val 12

a=3, b=4, c=12 } ex.

EXPLICIT ENV

a	ref-val 0
b	ref-val 1
c	ref-val 2

1. let a=2
2. in let b=3
3. in begin
4. set a=b;
5. a
6. end

"What does the env. and store look like after executing line 4?"

ENV:

a	0
b	1

STORE:

0	(num-val 3)
1	(num-val 3)

SHARING A MEMORY LOCATION

```
let x=2
in
  let f= proc(d) set x = -(x,-1)
  in
    let g= proc(d) x
    in
    begin
       (f 2);
       (g 2)
    end
```

ENV:

x	0
f	1
g	2

STORE:

0	(num-val 2)
1	(proc-val ... [x | 0]))
2	(proc-val (closure 'd ... [x | 0; f | 1])

these are dummy values that let proc run.

¡ÉXITOS!

(g 11) → counter=0

proc(11):
  counter=1
  return 1

ENV:

g	(ref-val 0)
counter	(ref-val 1)

Implicit Refs is the same as E/O + "ref-val"

(g 22) → counter=1

proc(22):
  counter=2
  return 2

STORE:

0	(num-val 1)
1	(num-val 2)

-((g 11),(g 22)) = -(1,2) = -1

EXPLICIT-REFS EXAMPLE

"What is the result of executing the following?"

let r = newref(10) in r

STORE:

0	(num-val 10)

ENV:

r	(ref-val 0)

GARBAGE: "Reference in store that you can never get to from the env."

```
let newref(10)
 in 2
```

NONEXISTENT

ENV:

STORE:

0	num-val 10

IMPLICIT BACK PATCHING

```
let F = proc(x) x
 in begin
      set f = proc(x) if zero?(x)
                      then 0
                      else + (x, (f, -(x,1)));
      (f 5)
    end
```

"Partial Sum of n numbers"

→ THIS PRODUCES: 5+4+3+2+1 = 15

Runtime Syntax

```
Proc (x) ...

(Proc-val
   (closure
     'x
     (...)))
```

Abstract syntax

```
(define-datatype exp-val...
 (...
 (proc-val
   (proc proc?))
(define-datatype proc proc?
 (procedure
   (bvar symbol?)
   (body expression?)
   (env environment?)))
```

(RUN/MAIN) Expanding Language

```
let p = proc(x) if =(x,3)
                 then 1
                 else 2
in (P2)
```

HEADER FILE Abstract syntax

```
(define-datatype exp exp?
  (eq-exp (exp1 exp?)
          (exp2 exp?)
```

CLASS FILE

```
(define (value-of exp env)
  (cases exp expression?
   (...
   (eq-exp (e1 e2)
     (let ((f (value-of e1 env))
           (x (value-of e2 env))
           (f1 (expval->num f))
           (x1 (expval->num x)))
     (bool-val (equal? f1 x1))
```

TAKE OFF TAG! expval->num / list / bool

equal? ← scheme

f1 x1 ← Have to be nums not num-vals.

INDUCTIVE SETS

<Num> ::= n, w/ n ∈ ℕ

<E> ::= <Num>
 | <E> + <E>
 | <E> − <E>
 | (<E>)

(Productions)

<E> ↑ Nonterminals

Terminals: n, +, −, (,)

CONT. UP-Right

INDUCTIVE SETS CONT. ABSTRACT SYNTAX

```
(define-datatype nat nat?
  (num (n positive?)))
```

```
(num (or (n positive?)
         (n zero?)))
```

```
(define-datatype exp exp?
  (num-exp (exp1 nat?))
  (add-exp (exp1 exp?)
           (exp2 exp?))
  (diff-exp (exp1 exp?)
            (exp2 exp?))
  (parent-exp (exp1 exp?)))
```

COMPLETE THE CODE

```
(define stack
  (let ((stk '()))
   (lambda (message)
    (case message
     ((empty?) (lambda ()
                 (null? stk)))
     ((push!) (lambda (x)
 Stack   (set! stk (append
                    (list x)
                    stk))
 Queue   (set! stk (append
                    stk
                    (list x)))
     ((pop!) (lambda ()
       (if (null? stk) ("error")
           (set! stk (cdr stk)))
     ((peek) (lambda ()
       (if (null? stk) ("error")
           (car stk))
     ))
     (else (error "stack error:"
            message))))))
```

ENV & EXECUTION

```
(run "let a=4 in 0")
  (num-val 0)          ↓
  (  ...  )            a
  (num-val 4 )         ↓
  (  ...  )        -(3,1)
  (num-val 2 )         ↓
    ( ... )        -(a,1)
      (numval 3)
```

DEAR DIARY...

LOVE ALWAYS!
PS. "okay, me too"

(ex1 V1 V2) returns '(v2 v1) ↓

COMPLETE THE CODE

```
(define (ex1 v1 v2)
  (let ((f
    (let ((mylist '()))
     (lambda (x)
      (let ((dummy
        (set! mylist
THIS {  (append mylist (list x)))))
     {  (reverse mylist))))
OR
THIS {  (append (list x) mylist)))))
     {  mylist)))
    (begin
      (f v1)
      (f v2))))
```

RESULT: (ex1 4 5) → '(5 4)

LET SYNTAX (Let ((x 3)) x+7)

LET = Local variable
 * Returns something

Define = Global variable

LANGUAGES	SCHEME
Let a=3 in...	((let ((x 3))

If not initial env is given, assume the env is empty. Otherwise include initial env.

i	num-val 1
v	num-val 5
x	num-val 10
a	num-val 4

GOOD LUCK!