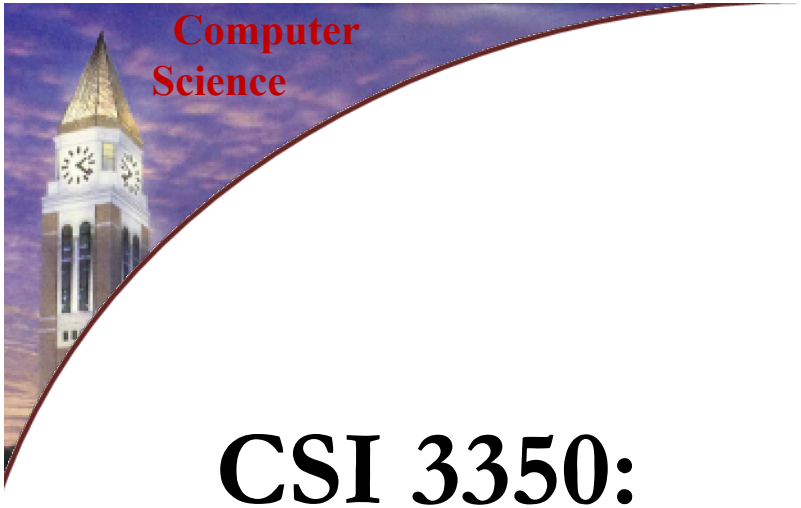


CSI 3350: PROGRAMMING LANGUAGES

Department of Computer Science & Engineering

Oakland University

What have we covered so far ?

- **`Described main quality criteria for high level programming languages such as readability, writability etc.`** (for example you can add new syntax to the language to facilitate the readability of your program using *define-syntax-rule, define-syntax etc.*)
- **`Described syntax of fundamental program components`** (hw06 *loop, block structure, scoping mechanism, etc.*,)
- **`Discussed fundamental concepts of operational semantics`** (hw06, coding the **`value-of`** function)
- Describe parameter passing and access to non-locals (hw07 – soon!)
- **`Described data types and type systems`** (*hw06, grammar, hw05, define-datatype*)
- **`Apply major features of functional programming languages`** (hw01~hw04, map, foldl, high order functions, lambda etc.)
- **`Described activation records`** (Sep 30 lecture notes, slides 43 ~ 66)

yet to cover

Road Ahead -

HW05 Due: Nov 12

↓

HW06 Due: Nov 24

↓

Exam02 on Nov 27

↓

HW07 Due: Dec 7 @ 10pm

↓

Final Exam : 7pm ~10pm : Dec 09, 2019 (same classroom)

The LET language

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero? (`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Figure 3.2 Syntax for the LET language

How to extend **LET** with **procedure** handling capabilities?

The Extended LET language

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero?` `(`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Figure 3.2 Syntax for the LET language

How to extend **LET** with **procedure** handling capabilities?

Expression ::= `proc` `(`*Identifier*`)` *Expression*
`proc-exp (var body)`

Expression ::= `(`*Expression* *Expression*`)`
`call-exp (rator rand)`

Suggested reading:

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

Fold Functions

- Two variants: `foldl` & `foldr`
  - applies a procedure to the elements of one or more lists.
  - (`foldl proc default` $\texttt{list}_1$ ... $\texttt{list}_n$)
    - the # of parameters that proc have must be **n+1**
    - following left to right order
    - the result of (`proc` x_1 ... x_n `default`) is stored and was used as the `default` for the next round

Fold Functions

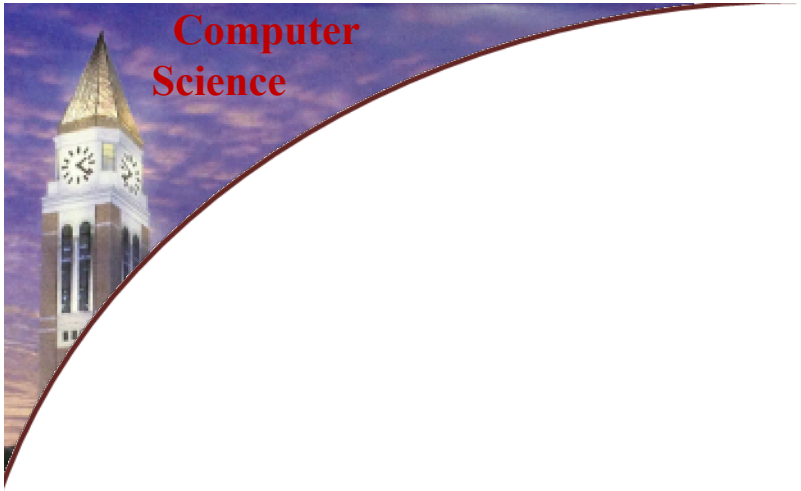
- Two variants: `foldl` & `foldr`
  - applies a procedure to the elements of one or more lists.
  - (`foldl proc default` $list_1$ ... $list_n$)
    - the # of parameters that proc have must be **n+1**
    - following left to right order
    - the result of (`proc` x_1 ... x_n `default`) is stored and was used as the `default` for the next round

Oct 9 lecture notes

Fold Functions

- Two variants: `foldl` & `foldr`
  - applies a procedure to the elements of one or more lists.
  - (`foldl proc default` $list_1$ ... $list_n$)
    - the # of parameters that proc have must be **n+1**
    - following left to right order
    - the result of (`proc` x_1 ... x_n `default`) is stored and was used as the `default` for the next round

- `( foldl proc` **`default`** `list`$_1$ `... list`$_n$`)`
  - the # of parameters that proc have must be **n+1**
  - following left to right order
  - the result of `(proc x`$_1$ `... x`$_n$ **`default`**`)` is stored and was used as the `default` for the next round




```
;(expr ("move" "(" expr (arbno expr)")") move-expr)
(move-expr
 (point-expr first-move rest-of-moves)
 (letrec
     ([start-p (point-val->p (value-of point-expr env))]
      [all-moves-as-expr (map (lambda (ex) (value-of ex env)) (flat-list
first-move rest-of-moves))]
      [all-moves-step (map step-val->st all-moves-as-expr)]
      [final-p (foldl move start-p all-moves-step)])
   (point-val final-p)
   )
 )
```

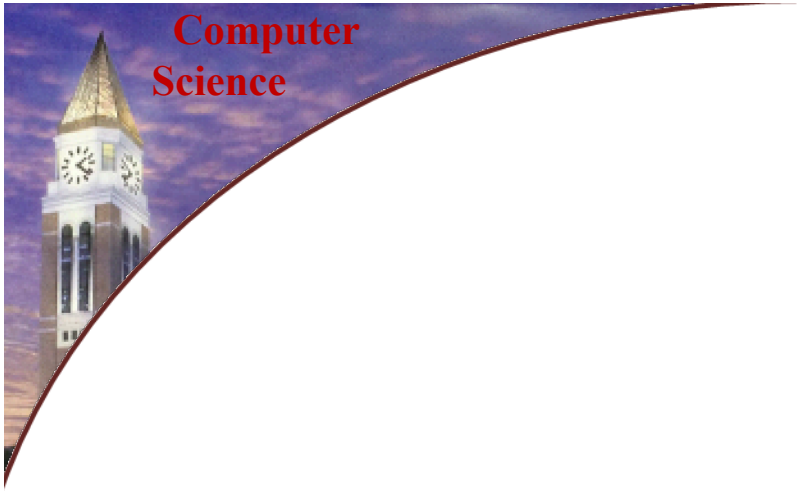
```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
;(expr ("move" "(" expr (arbno expr)")") move-expr)
(move-expr
 (point-expr first-move rest-of-moves)
 (letrec
     ([start-p (point-val->p (value-of point-expr env))]
      [all-moves-as-expr (map (lambda (ex) (value-of ex env)) (flat-list
first-move rest-of-moves))]
      [all-moves-step (map step-val->st all-moves-as-expr)]
      [final-p (foldl move start-p all-moves-step)])
   (point-val final-p)
   )
 )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
;(expr ("move" "(" expr (arbno expr)")") move-expr)
(move-expr
 (point-expr first-move rest-of-moves)
 (letrec
     ([start-p (point-val->p (value-of point-expr env))]
      [all-moves-as-expr (map (lambda (ex) (value-of ex env)) (flat-list
first-move rest-of-moves))]
      [all-moves-step (map step-val->st all-moves-as-expr)]
      [final-p (foldl move start-p all-moves-step)])
   (point-val final-p)
   )
 )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```




```
;(expr ("move" "(" expr (arbno expr)")") move-expr)
(move-expr
 (point-expr first-move rest-of-moves)
 (letrec
     ([start-p (point-val->p (value-of point-expr env))]
      [all-moves-as-expr (map (lambda (ex) (value-of ex env)) (flat-list
first-move rest-of-moves))]
      [all-moves-step (map step-val->st all-moves-as-expr)]
      [final-p (foldl move start-p all-moves-step)])
   (point-val final-p)
   )
 )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

swap **st** and **start-p** causes a problem ?

```
;(expr ("move" "(" expr (arbno expr)")") move-expr)
(move-expr
 (point-expr first-move rest-of-moves)
 (letrec
     ([start-p (point-val->p (value-of point-expr env))]
      [all-moves-as-expr (map (lambda (ex) (value-of ex env)) (flat-list
first-move rest-of-moves))]
      [all-moves-step (map step-val->st all-moves-as-expr)]
      [final-p (foldl move start-p all-moves-step)])
   (point-val final-p)
   )
 )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

swap **st** and **start-p** causes a problem ? **YES !!**

Oct 9 lecture notes

Fold Functions

- Two variants: `foldl` & `foldr`
  - applies a procedure to the elements of one or more lists.
  - (`foldl proc default` $\texttt{list}_1$ ... $\texttt{list}_n$)
    - the # of parameters that proc have must be **n+1**
    - following left to right order
    - the result of (`proc` x_1 ... x_n `default`) is stored and was used as the `default` for the next round

Fold Functions

- Two variants: `foldl` & `foldr`
  - applies a procedure to the elements of one or more lists.
  - `( foldl proc` **`default`** `list`$_1$ `... list`$_n$`)`
    - the # of parameters that proc have must be **n+1**
    - following left to right order
    - the result of `(proc` x_1 `...` x_n **`default`**`)` is stored and was used as the `default` for the next round

- (foldl proc **default** $list_1$ ... $list_n$)
  - the # of parameters that proc have must be **n+1**
  - following left to right order
  - the result of (proc x_1 ... x_n **default**) is stored and was used as the default for the next round

```
;(expr ("move" "(" expr (arbno expr)")") move-expr)
(move-expr
 (point-expr first-move rest-of-moves)
 (letrec
     ([start-p (point-val->p (value-of point-expr env))]
      [all-moves-as-expr (map (lambda (ex) (value-of ex env)) (flat-list
first-move rest-of-moves))]
      [all-moves-step (map step-val->st all-moves-as-expr)]
      [final-p (foldl move start-p all-moves-step)])
   (point-val final-p)
   )
 )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

replace `proc` with `move`

- (foldl proc **default** $list_1$ ... $list_n$)
  - the # of parameters that proc have must be **n+1**
  - following left to right order
  - the result of (proc x_1 ... x_n **default**) is stored and was used as the default for the next round

```
;(expr ("move" "(" expr (arbno expr)")") move-expr)
(move-expr
 (point-expr first-move rest-of-moves)
 (letrec
     ([start-p (point-val->p (value-of point-expr env))]
      [all-moves-as-expr (map (lambda (ex) (value-of ex env)) (flat-list
first-move rest-of-moves))]
      [all-moves-step (map step-val->st all-moves-as-expr)]
      [final-p (foldl move start-p all-moves-step)])
   (point-val final-p)
   )
 )
```

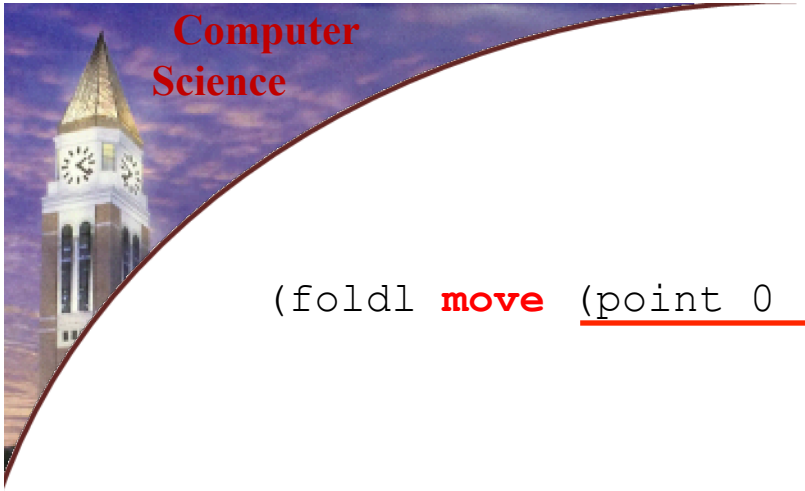
```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

for example if `start-p` is (point 0 0)

```
(foldl move start-p '( (left-step 4)  (up-step 2)  (right-step 3)  )  )
```

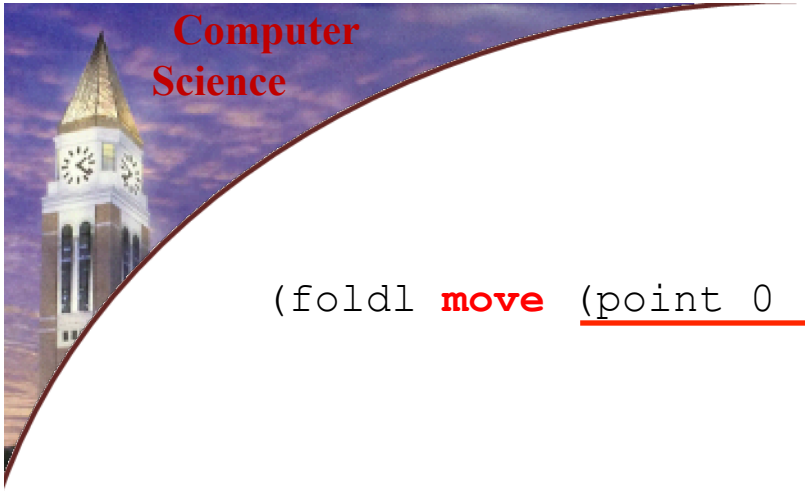
for example if `start-p` is (point 0 0)

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```




```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

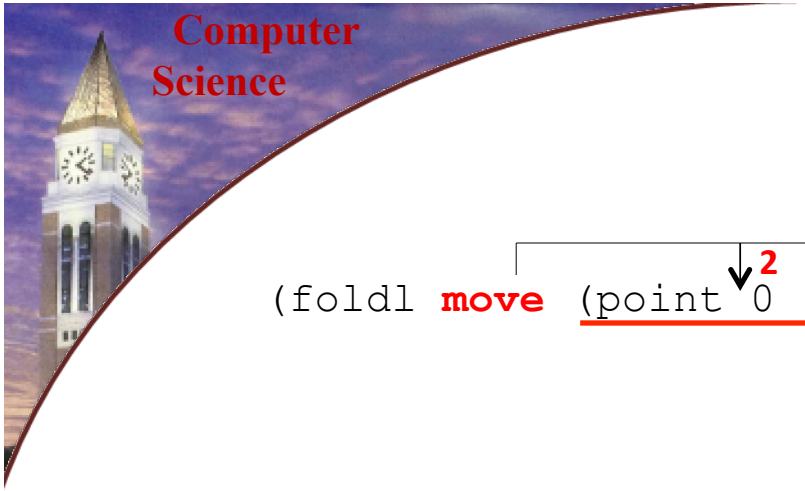
```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```




```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

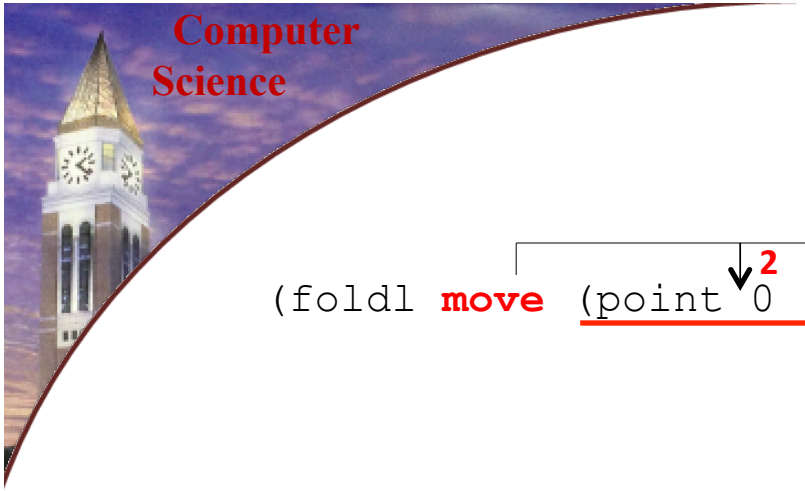
1 2

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```



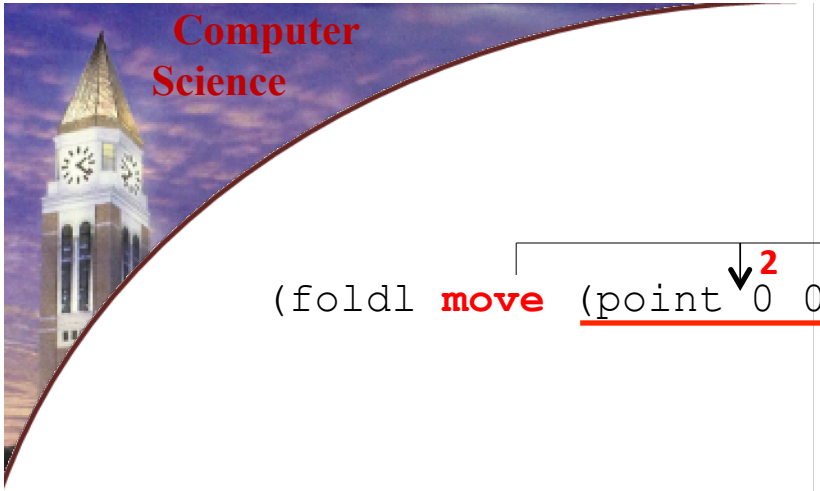

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```




```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

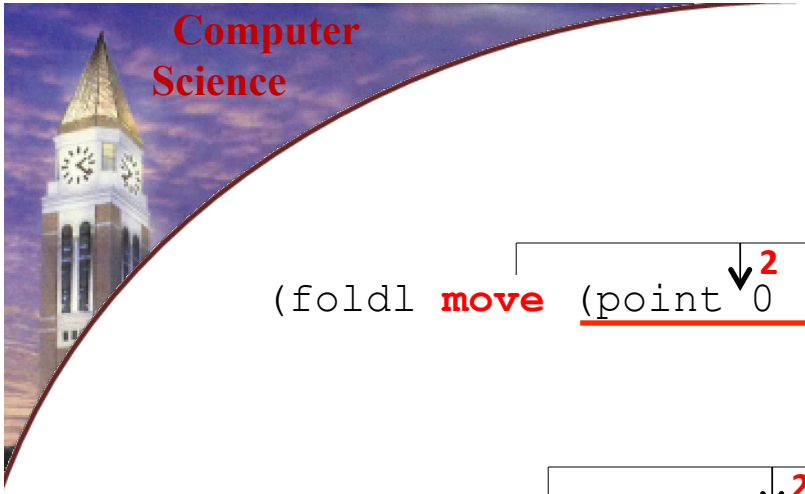
```
(foldl move (point 0 0) `( (left-step 4) (up-step 2) (right-step 3) ) )
```




```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 0) '( (up-step 2) (right-step 3) ) )
```




```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

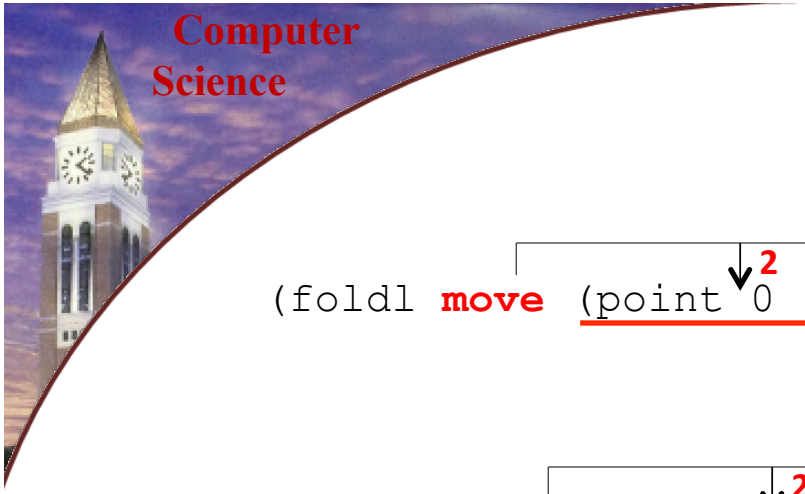
```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 0) '( (up-step 2) (right-step 3) ) )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 0) '( (up-step 2) (right-step 3) ) )
```




```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 0) '( (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 2) '( (right-step 3) ) )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 0) '( (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 2) '( (right-step 3) ) )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 0) '( (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 2) '( (right-step 3) ) )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 0) '( (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 2) '( (right-step 3) ) )
```

```
(foldl move (point -1 2) '( ) )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 0) '( (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 2) '( (right-step 3) ) )
```

```
(foldl move (point -1 2) '( ) )
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 0) '( (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 2) '( (right-step 3) ) )
```

```
(foldl move (point -1 2) '( ) )
```

```
(point -1 2)
```

```
(define (move st start-p)
        ; to save space we omit the function body here
        ; which you can find on page 4 of the code brochure.
)
```

```
(foldl move (point 0 0) '( (left-step 4) (up-step 2) (right-step 3) ) )
```

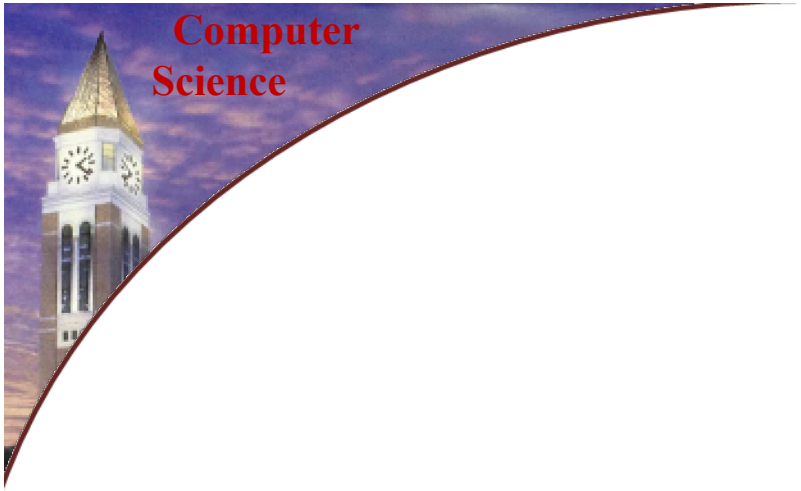
```
(foldl move (point -4 0) '( (up-step 2) (right-step 3) ) )
```

```
(foldl move (point -4 2) '( (right-step 3) ) )
```

```
(foldl move (point -1 2) '( ) )
```

```
(point -1 2)
```

each intermediate point is decided by **move** function !



EXTEND LET LANGUAGE WITH PROCEDURES

The LET language

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero?` `(`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Figure 3.2 Syntax for the LET language

The LET language

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero?` `(`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Figure 3.2 Syntax for the LET language

How to extend **LET** with **procedure** handling capabilities?

Suggested reading:

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

Extend the language with Procedures

- Concrete Syntax rules -

Procedure Definition

Expression ::= `proc` (Identifier) Expression

| (Expression Expression)

Procedure Call

The Extended LET language

Program ::= *Expression*
`a-program (exp1)`

Expression ::= *Number*
`const-exp (num)`

Expression ::= `-(`*Expression* `,` *Expression*`)`
`diff-exp (exp1 exp2)`

Expression ::= `zero?` `(`*Expression*`)`
`zero?-exp (exp1)`

Expression ::= `if` *Expression* `then` *Expression* `else` *Expression*
`if-exp (exp1 exp2 exp3)`

Expression ::= *Identifier*
`var-exp (var)`

Expression ::= `let` *Identifier* `=` *Expression* `in` *Expression*
`let-exp (var exp1 body)`

Expression ::= `proc` `(`*Identifier*`)` *Expression*
`proc-exp (var body)`

Expression ::= `(`*Expression* *Expression*`)`
`call-exp (rator rand)`

Figure 3.2 Syntax for the LET language

Examples

```
let f = proc (x) -(x,11)
       in (f (f 77))
```

Examples

```
let f = proc (x) -(x,11)
        in (f (f 77))
```

Examples

```
let f = proc (x) -(x,11)
      in (f (f 77))
```

← (num-val 55)

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

To Define a Procedure in `Scheme` –

```
(define a
      (lambda (x) (+ x 1) ) )
```

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

To Define a Procedure in `Scheme` –

```
(define a
      (lambda (x) (+ x 1) ) )
```

whole thing is one value, given to **a**

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

To Define a Procedure in `Scheme` –

```
(define a
      (lambda (x) (+ x 1) ) )
```

whole thing is one value, given to **a**

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

To Define a Procedure in `Scheme` –

```
(define a
        (lambda (x) (+ x 1) ) )
```

whole thing is one value, given to **a**

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

To Define a Procedure in `Scheme` –

```
(define a
      (lambda (x) (+ x 1) ) )
```

whole thing is one value, given to **a**

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```

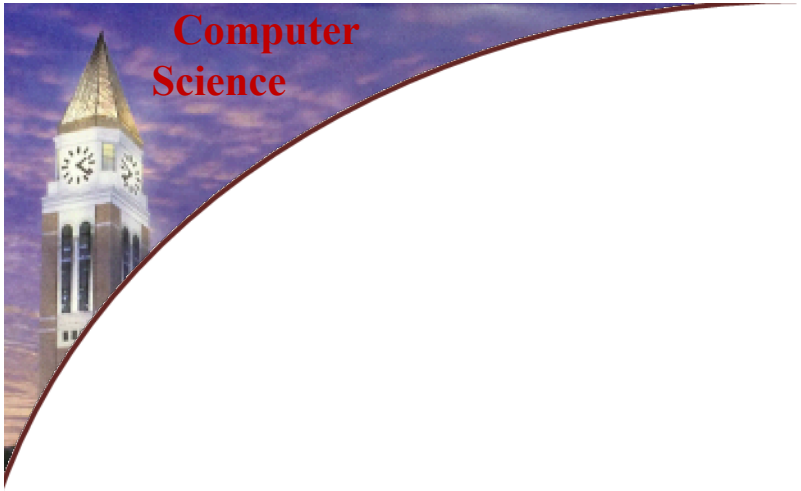
```
(define-datatype proc proc?
   (procedure
       (var identifier?)
       (body expression?)
       (saved-env environment?)))
```

Found at procedure creation-time, **not** procedure call-time !!

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```

```
(define-datatype proc proc?
   (procedure
       (var identifier?)
       (body expression?)
       (saved-env environment?)))
```



Closure (in hw07)

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```

```
(define-datatype proc proc?
   (procedure
       (var identifier?)
       (body expression?)
       (saved-env environment?)))
```

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```

Procedures have to be associated with an environment !

```
(define-datatype proc proc?
   (procedure
       (var identifier?)
       (body expression?)
       (saved-env environment?)))
```

- EOPL: 3.1-3.2 (implementation of LET language)
- EOPL: 3.3 (Extend the LET language with Procedures p74- p82)

```
(define-datatype expval expval?
  (num-val
      (num number?) )
  (bool-val
      (bool boolean?))
  (proc-val
      (proc proc?))
)
```

```
(define-datatype proc proc?
   (procedure
        (var identifier?)
        (body expression?)
        (saved-env environment?)))
```

Procedures have to be associated with an environment !

Together they are called a **closure** .

SEMANTICS

SEMANTICS
(operational semantics)

Extend (value-of ...)

For procedure definition!

```
(define (value-of ex env)
  (cases expression ex
    ...
     (proc-exp (var body)
       (proc-val (procedure var body env) ))
  )
)
```

AST returned by SLLGEN parser

Expression ::= proc (Identifier) Expression

proc-exp (var body)

Concrete Syntax

Abstract Syntax
(SLLGEN ...)

Extend (value-of ...)

For procedure call!

```
(define (value-of ex env)
  (cases expression ex
    ...
     (call-exp (rator rand)
       (let ((proc (expval->proc (value-of rator env)))
             (arg (value-of rand env)))
            (apply-procedure proc arg)) )))
```

Expression ::= (Expression Expression)
call-exp (rator rand)

```
( proc (x) -(x, 11)  12)
```

(Expression Expression)

SLLGEN Parser

Expression ::= proc (Identifier) Expression
proc-exp (var body)

Extend (value-of ...)

For procedure call!

```
(define (value-of ex env)
  (cases expression ex
    ...
     (call-exp (rator rand)
       (let ((proc (expval->proc (value-of rator env)))
             (arg (value-of rand env)))
            (apply-procedure proc arg)) )))
```

Expression ::= (Expression Expression)
call-exp (rator rand)

```
( proc (x) -(x, 11)  12)
```

rator → (value-of rator env) → (proc-val (procedure (var-exp 'x) (diff-exp (var-exp 'x) (const-exp 11)) env))

rand → (value-of rand env) → (num-val 12)

(expval->proc (value-of rator env))

Extend (value-of ...)

For procedure call!

```
(define (value-of ex env)
  (cases expression ex
    ...
     (call-exp (rator rand)
       (let ((proc (expval->proc (value-of rator env)))
             (arg (value-of rand env)))
            (apply-procedure proc arg)) )))
```

Expression ::= (Expression Expression)
call-exp (rator rand)

```
( proc (x) -(x, 11)  12)
```

rator

rand → (value-of rand env) → (num-val 12) → arg

(value-of rator env)

(proc-val

(procedure (var-exp 'x) (diff-exp (var-exp 'x) (const-exp 11)) env)

) → proc

Extend (value-of ...)

For procedure call!

```
(define (value-of ex env)
  (cases expression ex
    ...
     (call-exp (rator rand)
       (let ((proc (expval->proc (value-of rator env)))
             (arg (value-of rand env)))
            (apply-procedure proc arg)) )))
```

Expression ::= (Expression Expression)
call-exp (rator rand)

```
( proc (x) -(x, 11)  12)
```

rator

rand → (value-of rand env) → (num-val 12) ← arg

(value-of rator env)

(proc-val

(procedure (var-exp 'x) (diff-exp (var-exp 'x) (const-exp 11)) env)

) ← proc

Extend (value-of ...)

For procedure call!

```
(define (apply-procedure proc1 val)
  (cases proc proc1
     (procedure (var body saved-env)
       (value-of body (extend-env var val saved-env))
  )
)
```

```
( proc (x) -(x, 11)  12)
```

```
(apply-procedure
proc  (procedure (var-exp 'x)  (diff-exp  (var-exp  'x)  (const-exp 11)  )  (empty-env)  )
arg   (num-val 12)
)
  (value-of
      (diff-exp (var-exp 'x) (const-exp 11) )
      (extend-env 'x (num-val 12) (empty-env))
  )
```