# CSI2120 Programming Paradigms

Jochen Lang



jlang@uottawa.ca





**Jochen Lang, EECS**
**jlang@uOttawa.ca**

# Scheme: Functional Programming

- Local Binding, le … ariables
- Named let-boun
- Characters
- Strings



**Jochen Lang, EECS**
**jlang@uOttawa.ca**

# Local Binding, let-bound Variables: `let`

- let
  - to define a list [illegible]ables for a list of expressions
  - each variable n[illegible]d with a value
  - let returns the result of the last expression
    - but evaluates all expressions from left to right



```
(let ((a 2) (b 3)) ; local variables a and b
   (+ a b))        ; expression where the
                   ; variables are bound
=> 5
a
=> Unbound variable: a
b
=> Unbound variable: b
```

# Local Function Definitions

- let can be used … local functions

```
(let ((a 3)
      (b 4)
      (square (lambda (x) (* x x)))
      (plus +)) ; end of definitions
  ; applied to
  (sqrt (plus (square a) (square b))))
=> 5
```

# Sequential Definitions with `let*`

```
(let ((x 1) (y (+ x 1)))
  (list x y))
=> Error: variable x is not bound.
```

- In order to define y in terms of x
  - the function let* exists

```
(let* ((x 1) (y (+ x 1)))
  (list x y))
=> (1 2)
```

- let* is similar to let but allows for sequential definitions.

# Example using `let` vs. `let*`

```
(let ((x 2) (y 3))
  (let ((x 7)          ; x = 7
        (z (+ x y))) ; z = 2 + 3
    (* z x)))          ; 5 * 7
=> 35

(let ((x 2) (y 3))
  (let* ((x 7)         ; x = 7
         (z (+ x y))) ; z = 7 + 3
    (* z x)))          ; 10 * 7
=> 70
```

# Setting let-bound Variables

- Let-bound variables can be changed with `set!`

```
(define seconds-set
    (lambda (h m s)
        (let ((sh 0) (sm 0) (total 0))
            (set! sh (* 60 (* 60 h)))
            (set! sm (* 60 m))
            (set! total (+ s (+ sh sm)))
            total)))
=> seconds-set
(seconds-set 1 5 3)
=> 3903
```

# Same Example in Functional Style

```
(define secon
  (lambda (h
    (let ((sh (* 60 (* 60 h)))
      (sm (* 60 m)))
      (+ s (+ sh sm))))))
=> seconds
(seconds 1 5 3)
=> 3903
```



**Jochen Lang, EECS**
**jlang@uOttawa.ca**

# Recursive Definitions with `letrec`

- `letrec`
  - permits the r[illegible]efinitions of functions
  - letrec is similar `let*` but all the bindings are within the scope of the corresponding variable



- Example: Local definition of factorial

```
(letrec ((fact (lambda (n)
                 (if (= n 1)
                     1
                     (* n (fact (- n 1)))))))
  (fact 5))
=> 120
```

# Recursive Application of a Function to a List

- Example:
  - The function fct [illegible] to all elements in a list

```
(define (apply-f fc[illegible]
(letrec ((app
          (lambda (L)
            (if (null? L)
                ()
                (cons (fct (car L)) (app (cdr L)))))))
  (app L)))
=> apply-f
(define double-ele (lambda (x) (+ x x)))
=> double-ele
(apply-f double-ele '(1 2 3 4))
=> (2 4 6 8)
```

# Named let-bound Variables

- Use of a name in the let expression
  *(let name ((var val) ...) exp$_1$ exp$_2$ ...)*
  - Factorial example

```
(let ft ((k 5))
  (if (<= k 0)
      1
      (* k (ft (- k 1))))) ; call with k=k-1
```

- is the same as:
  *(letrec ((name (lambda (var ...) exp1 exp2 ...)) (name val) ...)*

```
(letrec ((ft (lambda (k)
               (if (<= k 0)
                   1
                   (* k (ft (- k 1))))))) (ft 5))
```

# Examples: Named let-Bound

- Used for recursion loops

```
(define divisors
  (lambda (n)
    (let f ((i 2))
      (cond
       ((>= i n) '())
       ((integer? (/ n i))
      (cons i (f (+ i 1)))) ; call body with i=i+1
       (else (f (+ i 1)))))))) ; call body with i=i+1
=> divisors
(divisors 32)
=> (2 4 8 16)
```

# A Further Example

```
(let loop ((numbers[illegible] 6 -5))
           (nonneg '[illegible]
           (neg '())[illegible]
  (cond ((null? numbers) (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers) ; 3 arg. for loop
               (cons (car numbers) nonneg)
               neg))
        ((< (car numbers) 0) ; 3 other arg. for loop
         (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg))))))
=> ((6 1 3) (-5 -2))
```

# Store State in a Global with `set!`

```
(define num-c
=> num-calls
(define kons
    (lambda (x y)
        (set! num-calls (+ num-calls 1))
        (cons x y)))
=> kons
(kons 3 5)
=> (3 . 5)
(display num-calls)
1
```

# Types Characters

- Character consta

  `#\a` `#\A` `#\space` `#\newline`

- Predicats:



  - Mostly obvious

  `(char? obj)` tests whether obj is a character.



```
(char-alphabetic? char)
(char-numeric? char)
(char-whitespace? char)
(char-upper-case? char)
(char-lower-case? char)
```

# Character Comparisons

- Boolean functio[illegible]acters:

```
(char=? char_[illegible]
(char<? char_1 char_2)
(char>? char_1 char_2)
(char<=? char_1 char_2)
(char>=? char_1 char_2)
```

- Corresponding case insensitive functions with the ending `-ci` exist.

```
(char=? #\a #\A)
=> #f
(char-ci=? #\a #\A)
=> #t
```

# Character Conversions

- Character to asc

```
(char->intege
97
```

- Character to ascii and back

```
(integer->char (1+ (char->integer #\a)))
#\b
```



Jochen Lang, EECS
jlang@uOttawa.ca

# Strings

- String constants are written in double quotation marks

```
"Hello"
```

- Boolean comparison functions for strings

```
(string=? string_1 string_2)
(string<? string_1 string_2)
(string>? string_1 string_2)
(string<=? string_1 string_2)
(string>=? string_1 string_2)
```

- Examples

```
(string=? "Foo" "foo")
#f
(string-ci=? "Foo" "foo")
#t
```

# More String Functions

```
(string-length "Hello")
=> 5
(string->list "Hello")
=> (#\H #\e #\l #\l #\o)
(substring "computer" 3 6)
=> "put"
```

# ABC

```
(define (abc-co          k)
  (if (char-alp          char)
      (let ((ba          har-upper-case? char)
                         ->integer #\A)
                 (char->integer #\a))))
        (integer->char
         (+ base
            (modulo
             (+ k
              (- (char->integer char) base))
             26))))
      char)) ; apply let to char
=> abc-count
(abc-count #\b 5)
#\g
```

# Summary

- Local Binding, le[illegible]ariables
  - let for local v[illegible]nding
  - let* for sequential local varible binding
  - letrec for local variable binding allowing recursions
- Named let-bounds
- Characters
- Strings



Jochen Lang, EECS
jlang@uOttawa.ca