Programming Paradigms CSI2120 – Winter 2018

Jochen Lang

EECS, University of Ottawa

Canada

uOttawa

L'Université canadienne
Canada's university

Université d'Ottawa | University of Ottawa

uOttawa.ca

Scheme: Functional Programming

- Local Binding, let-bound Variables
- Named let-bounds
- Characters
- Strings

Local Binding, let-bound Variables: `let`

- **let**
  - to define a list of local variables for a list of expressions
  - each variable name is bound 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
```

Example Use: Polynomial

$$f(x,y) = x*(1+x*y)^2 + y*(1-y) + (1+x*y)*(1-y)$$

$$a = 1+x*y$$

$$b = 1-y$$

$$f(x,y) = x*a^2 + y*b + a*b$$

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
    (b (- 1 y)))
    (+ (* x a a) (* y b) (* a b))))
=> f
(f 1 2)
=> 4
```

Local Function Definitions

- **let can be used to define 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
```

Local and Global Definitions

- **let can be used with top-level defines**

```
 (define x 'a)
=> x
 (define y 'b)
=> y
 (list x y)       b        a
=> (a b)
 (let ((x y) (y x)) (list x y))
=> (b a)
```

First the variables are let-bound and then they are applied to the expressions following the list of definitions

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

Recursive Definitions with `letrec`

- **`letrec`**
  - permits the recursive definitions 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 is applied to all elements in a list

```
(define (apply-f fct L)
(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 recursions and 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 '(3 -2 1 6 -5))
          (nonneg '())
          (neg '()))
  (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-calls 0)
=> 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 constants:**

```
#\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 functions for characters:**

```
(char=? char_1 char_2)
(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 ascii**

```
(char->integer #\a)
97
```

- **Character to ascii and back**

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

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-count char k)
  (if (char-alphabetic? char)
      (let ((base (if (char-upper-case? char)
                   (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, let-bound Variables**
  - let for local variable binding
  - let* for sequential local varible binding
  - letrec for local variable binding allowing recursions
- **Named let-bounds**
- **Characters**
- **Strings**