Declarative Programming

Functional Programming
- LISP
  - Scheme
  - Common Lisp
  - ML
  - Haskell
  based on the $\lambda$-calculus

Logic Programming
- PROLOG
  based on the first-order predicate calculus
Functional Programming

- based on the $\lambda$-calculus
- functions as “first-class” values
- recursion is the main paradigm for repeated execution
- *List* is the basic data structure
- garbage collection
Mathematical Functions

Function: a mapping from a set (the *domain*)
to another set (the *co-domain*)

\[ f : D \rightarrow C \]

Composition: \( h = f \circ g \)

\[ h(x) = f(g(x)) \]

used to define functions in terms of other functions
\[ f(x) = x^2 \text{ (mathematical notation)} \]

\[ f = \lambda x . x^2 \text{ (\(\lambda\)-calculus notation)} \]

useful in creating unnamed functions

\[ \lambda x . E \]

formal parameter

body
\( \lambda \text{-calculus} \)

“pure” \( \lambda \text{-calculus}: \)

\[
\begin{align*}
\text{<expression>} & \ ::= \text{<identifier>} \\
& \ ::= \lambda \text{<identifier>} \ . \text{<expression>} \\
& \ ::= (\text{<expression>} \ \text{<expression>})
\end{align*}
\]

\( \lambda x. x \) stands for the identity function
λ-expressions in Scheme

built-in functions +, *, ...

(lambda <list of formal parameters>
   <expression>
)

Examples:

(lambda (x) (* x x))

(lambda (x) (+ x 2))

(lambda (x y) x)
(lambda (x y) (+ x y))

is different from

(lambda (x) (lambda (y) (+ x y)))
Creating functions

\[(\lambda (x_1 \ x_2 \ \ldots \ x_k) \ E)\]

When this functional object is invoked,

the parameters are call-by-value

E is evaluated and

the resulting value is returned
Lexical scoping

(define f
  (lambda (x₁ x₂ ... xₖ) E)
)

The value of a λ-expression is a procedure (closure) that consists of

the list of parameters

the body E

the environment in which the free variables in the body are bound at the time the λ-expression is evaluated

lexical scoping: values of free variables are looked up in the environment in which the procedure was defined
Defining functions

Suppose the value of x is 2

```
1 ]=> (define foo (lambda (y) (+ x y)))

;Value: foo
```

```
1 ]=> (foo 1000)

;Value: 1002
```

```
1 ]=> (let ((x 1)) (foo 1000))

;Value: 1002
```
letrec

the proper way to define recursive functions

(letrec
  
  (\n    (<var_1>  <expr_1>)
    
    ...
    
    (<var_k>  <expr_k>)
  
  )

  <expression>

)
(define factorial
    (letrec
        ((f (lambda (n)
            (if (= n 0) 1
                (* n (f (- n 1)))
            )))
    )
)
)

f
)
)
(define remove
  (letrec
      ((rhelp
         (lambda (x ls)
           (cond ((null? ls) '())
                 ((equal? x (car ls))
                  (rhelp x (cdr ls)))
                 (else (cons (car ls)
                              (rhelp x (cdr ls)))))

     )
     )
     )
     )
     )
     )
rhelp)
)
)
define again

\[(\text{define} \ (<\text{Name}> \ <f_{p_1}> \ \ldots \ <f_{p_n}> ) \ E)\]

is equivalent to

\[(\text{define} \ <\text{Name}> \ (\lambda (\ <f_{p_1}> \ \ldots \ <f_{p_n}> ) \ E) )\]
Nested definitions

internal definitions

(define (remove x ls)
  (define (loop L M)
    (cond ((null? L) M)
                ((equal? x (car L)) (loop (cdr L) M))
                  (else
                     (loop (cdr L) (cons (car L) M)))
      )
    )
  )
)

(reverse (loop ls '()))
)

loop is not visible outside remove
Higher-order functions

- functions as arguments
- functions as (part of) the return value

```
(define (compose f g)
  (lambda (x) (f (g x))))
```  

```
((compose 1+ 1+) 100) → 102
```
map

(map \( f (a_1 \ldots a_n) \))

\[ \equiv (f(a_1) \ldots f(a_n)) \]

(map 1+ '(10 20 30 40))

\[ \Rightarrow (11 21 31 41) \]

(map (lambda (x) (* x x)) '(10 20 30 40))

\[ \Rightarrow (100 400 900 1600) \]
(define (mapcan f ls)
    (if (null? ls)
        ()
        (append (f (car ls)) (mapcan f (cdr ls)))
    )
)

(mapcan cdr '((1 2 3) (4 5 6) (7)))
⇒ (2 3 5 6)
fold-left

\[
\begin{array}{c}
\text{f} \\
\text{f} \\
x \quad a_1 \quad a_{n-1} \quad a_n
\end{array}
\]
fold-left

(define (fold-left f x ls)
  (define (lrhelp y L)
    (if (null? L) y
     (lrhelp (f y (car L)) (cdr L)))
  )
  (lrhelp x ls)
)

(fold-left + 0 '(1 3 5 7 9 11))
⇒ 36
(fold-left
  (lambda (x y) (cons y x))
  '()
  '(1 3 5 7)
)
⇒ (7 5 3 1)

(fold-left (lambda (x y) (cons (1+ y) x))
  '()
  '(10 20 30)
)
⇒ (31 21 11)
fold-right

\[
\begin{array}{c}
\text{f} \\
\text{a}_1 \text{ f} \\
\text{a}_2 \text{ f} \\
\text{a}_n \text{ x}
\end{array}
\]
fold-right

(define (fold-right f x ls)
  (define (frhelp ls)
    (if (null? ls)
        x
        (f (car ls) (frhelp (cdr ls))))
  )
  )
  (frhelp ls)
)

(fold-right - 0 '(7 11 13))
⇒ 9

(fold-right + 0 '(1 3 5 7 9 11))
⇒ 36
(fold-right cons '() '(1 3 5 7))
⇒ (1 3 5 7)

(define (I x) x)

(define (myf x f) (lambda (y) (f (cons x y))))

((fold-right myf I '(1 3 5 7)) '())
⇒ (7 5 3 1)
\( \lambda x. (\text{cons } 7 \ (\text{cons } 5 \ (\text{cons } 3 \ (\text{cons } 1 \ x))) \)