Defining 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
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
  (
    (<var_1>  <expr_1>)
    
    
    (<var_k>  <expr_k>)
    
    
    (<expression>)
  )
)

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

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

is equivalent to

\[(\text{define} \ <\text{fun\_name}> \n\ (\text{lambda} \ (<f_{p_1}> \ \ldots \ <f_{p_n}>) \ E)\n)\]
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
Tail recursion

result of a recursive call (within the body) is not further modified

it is the result of the function

```
(define (fact n)
  (define (loop m p)
    (if (= m 0) p
      (loop (- m 1) (* m p))
    )
  )
  (loop n 1)
)
```
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)  \rightarrow  102)
\( \text{map} \)

\[
(\text{map } f \ (a_1 \ \ldots \ a_n))
\]

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

\[
(\text{map} \ 1+ \ ' (10 \ 20 \ 30 \ 40))
\]

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

\[
(\text{map} \ (\lambda \ (x) \ (* \ x \ x)) \ ' (10 \ 20 \ 30 \ 40))
\]

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

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

\[
\begin{array}{c}
\text{fold-right} \\
\begin{array}{c}
\text{f} \\
\text{a_1} & \text{f} \\
\text{a_2} \\
\text{f} \\
\text{a_n} & \text{x}
\end{array}
\end{array}
\]
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)
\[ \lambda x. (\text{cons } 7 \ (\text{cons } 5 \ (\text{cons } 3 \ (\text{cons } 1 \ x)))) \]