Conditional Evaluation

If statement

(if *bool-expr then-expr else-expr*)

Examples:

```
(if (< 2 3) 5 7)
(if #t #t #f)
(* 2 (if (zero? x)
   (if (zero? (- x 1)) 1 2) 4))
```

Multiway branching -

`(cond (`T_1 E_1`) ... (`T_n E_n`) (else` E`))`

Example:

```
(cond ((equal? x 0) 'zero)
  ((< x 0) 'less)
  (else 'greater))
```


Lambda expressions

functions are represented as
(lambda (<*formal-parameters*>) <*expression*>)

Examples:

`((lambda (x) (* x x)) 5)` ⇒ 25

`(lambda () x)` ⇒ whatever `x` was

```
(define f (lambda () x))
(f)
(define square (lambda (x) (* x x)))
(define (square x) (*x x))
```

• note that `(lambda (`x_1 `...` x_n`)` E`)` binds x_1 ... x_n locally (in E) to the values of the actual arguments when the procedure is applied:

`((lambda (x y) (+ x y)) 2 3)`

• function values can appear within expressions, either as operators or as arguments

• but cannot do recursion until we name functions

Why `Lambda`s?

- Allows defining functions, sending them into functions and returning them from functions without having to name them.
- Can't name them all the time:

`(define (next-par x) (lambda (y) (list x y)))`

Why not `Lambda`s?

- Whenever the same function is more than once.

Global Definitions

`(define` x E`)`	binds x to E
`(define (`x y_1 `...` y_n`)` E`)`	binds x to proc. E with params. y_1, ..., y

Examples:

```
(define my-number 5)
my-number ⇒ 5
(define x 4)
(define x 7) ;Value of x is 7
(define (square x) (* x x)) ;Function square
(square 5) ⇒ 25
```

What if in `(define` x E`)`, x is a free variable in E? Then get *recursive definition*:

```
(define (fact n) (if zero? n) 1 (* n (fact
(- n 1))))
(fact 3) = (* 3 (fact (-3 1)))
  = (* 3 (* 2 (fact (-2 1))))
  = (* 3 (* 2 (* 1 (fact (- 1 1)))))
```

Local Bindings

- Local variables exist in Scheme.
- Lexical (or static) scope, i.e., local variables can be renamed without changing the result of the expression.

Syntax:`(let` $((x_1\ E_1)\ \ldots\ (x_n\ E_n))\ E)$

- Evaluate E_1, E_2, ..., E_k
- Evaluate E with x_i representing the value of E_i
- Return result.

Definitions: `(let` $((x_1\ E_1))\ E)$

- x is called *binding occurrence* (or just a *binding*).
- All occurrences of x in E are said to be within the *scope* of this binding.
- The occurrences of x in E are said to be *bound*.
- Occurrences of x in E_1 are not in the scope of this binding.

Bindings, Cont'd

Why use `lets`?

- Allow subexpressions to be named:

```
(let ((square-3 (square 3))
 (square-4 (square 4)))
 (+ square-3 square-4))
```

- Used to avoid duplicate evaluation

```
(let ((square-3 (square 3)))
   (+ square-3 square-3))

(let ((x 3)) (* x x))
```

- `let` is like passing parameters. Names of formal parameters can be changed and the result remains the same.
- so, "variables can be renamed"

```
(let ((z 3)) (* z z))
(let ((foo 3)) (* foo foo))
```


Bindings (Cont'd)

What is the scope of `x` in

```
(let ((x 2))
   (let ((y 5)) (+ x (* 2 y))))
```

What happens if we have nested bindings of the same variable:

```
(let ((x 2))
   (let ((x (+1 x))) (* x x)))
```

Rename to avoid confusion!

```
(let ((x 2))
   (let ((y (+1 x))) (* y y)))
```

Local bindings "shadow" global, which are then restored

```
(define x 4)
(define (x 3) (y (+ x 2)) (+ x y)) ⇒
x ⇒ 4
```


Combining Bindings

- `let` evaluates all expressions before binding them
- `let*` is sequential `let`:

`(let*` $((x_1\ E_1)\ \ldots\ (x_n\ E_n))\ E)$

- Evaluate E_1 and bind to x_1; evaluate E_2 and bind to x_2...
- Evaluate E with x_i representing the value of E_i
- Return result

```
(define x 0)
(let ((x 2) (y x)) y) ⇒
(let* ((x 2) (y x)) y) ⇒
```

Bindings, Cont'd

```
(define x 4)                (let*
(define f (lambda () x))      (( x 4)
(define x 5)                  (f (lambda () x))
(f) ⇒ 5                       (x 5))
                            (f)) ⇒ 4
```

- local definitions with `let` are non-recursive
- for local recursive definition, `letrec` form
- allows mutually recursive local definitions:

```
(letrec (
  (odd? (lambda (m) (if (zero? m)
    #f (even? (- m 1)))))
  (even? (lambda (n) (if (zero? n)
    #t (odd? (- n 1))))))
  (even? 3))
```


Structure of Lists

- Lisp has one data structure - list
- program and data are represented as lists
- a list is zero or more elements in parentheses
- elements are boolean, numbers, symbols, other lists, strings, characters, functions

```
()
(a b c)

(a (b c))

(((a) b) c)

(a ())
```

- (+ 2 3) - expression. '(+ 2 3) - list
- quote stops evaluation of a list
  `quote (+ 2 (* 3 4)) ⇒ (+ 2 (* 3 4))`

Operations on Lists

`(null? x)`	is list `x` empty
`(car x)`	first element of (non-empty) list `x`
`(cdr x)`	all elements of x except the first
`(cons x y)`	make a list with `x` as the first element of list `y`
`(list a b c)`	make a list with elements `a`, `b`, `c`

```
(list 'a 'b '(c d)))   ⇒
(list '() 'a))         ⇒
(list 'a '(a)))        ⇒
(car '(a b c))         ⇒
(car '()))             ⇒
(car '((a) b c)))      ⇒
(cdr '(b c))           ⇒
(cdr '(c))             ⇒
(cdr '((c)))           ⇒
(cons 'a '(b c)))      ⇒
(cons 'a 'b))          ⇒
(cons '(a) '(b c)))    ⇒
(caar '((a) b c)))     ⇒
(cadr '((a)(b c)))     ⇒
(cdar '((a)(b c)))     ⇒
```


Other Operations on Lists

```
(reverse '(a b c)) ⇒ (c b a)
(append '(a b c) '(d e)) ⇒ (a b c d e)
(length '(a b c)) ⇒ 3
(min '(1 2 3)) ⇒ 1
(max '(1 2 3)) ⇒ 3
(apply + '(1 2 3)) ⇒ 6

(define (plus-2 x) (+ 2 x))
(map plus-2 '(1 2 3)) ⇒ (3 4 5)
```

Structure of Lists (Cont'd)

• `cons` is a list constructor in graphical representation!

Recursive Programming

• Very powerful tool for processing *recursive types*
• In Scheme - mostly lists
• Can reason inductively about recursive types.

Example:

1. `0 is a number`
2. if `n` is a number then `(+1 n)` is a number
3. nothing else is a number.

`0` and $+1$ are *constructors* for the datatype pos. integer:

```
3 = (+1 (+1 (+1 0)))
```

Also need *accessors* - procedures to take datatype apart: `zero?`, `pred`:

```
(zero?  0) = #t
(zero?  (+1 n)) = n
(pred (+1 n)) = n
```


Recursion (Cont'd)

Now addition of numbers (binary + from unary `+1`):

```
n + 0 = n
n + m = (+1 n) + (pred m)
```

Convert into Scheme:

```
define (+ m n)
  (cond ((zero?  m) n)
  (else (+1 (+ (pred m) n)))))
```


Recursive Programming using Lists

1. '() is a list (the empty list)
2. if `E` is an expression, `L` is a list, then `(cons E L)` is a list
3. nothing else is a list.

```
'(a b c) = (cons 'a (cons 'b (cons 'c '())))
```

Accessors: `null?`, `car`, `cdr`

```
(null?  '() ) = #t
(null?  (cons E L)) = #f
(car (cons E L)) = E
(cdr (cons E L)) = L
```

`Length` function

```
(length '()) = 0
(length (cons E L)) = (+1 (length L))
```

And now in Scheme:

```
(define (length L)
  (cond ((null?  L) 0)
  (else (+1 (length (cdr L))))))
```

Tracing:

```
(length '(1 2 3))
     = (+1 (length '(2 3)))
     = (+1 (+1 (length '(3))))
     = (+1 (+1 (+1 (length '()))))
     = (+1 (+1 (+1 0))) = 3
```


Function `sum`

`sum '(1 2 3)` ⇒ `6`

Definition:

```
(sum '()) = 0
(sum (car E L)) = E + (sum L)
```

In Scheme:

```
(define (sum L)
  (cond ((null?  L) 0)
  (else (+ (car L) (sum (cdr L))))))
```

So, most of functions for lists are:

`(define (f x)`
 if list `x` is empty then...
  otherwise do something involving `(car x)`, `(cdr x)` and `f`.

`Member` Function

```
(member x '()) = false
(member x (car E L)) =
  if (x = E) then true else (member x L)
```

And in Scheme:

```
(define (member x L)
  (cond ((null?  L) #f)
  ((equal?  x (car L)) #t)
  (else (member (x (cdr L))))))
```

Tracing:

```
(member 'c '(a b c))
   = (if (equal 'c 'a) then #t else (member 'c '(b c)))
   = (if (equal 'c 'b) then #t else (member 'c '(c)))
   = (if (equal 'c 'c) then #t else (member 'c '()))

(member 'd '(a b c))
   = (if (equal 'd 'a) then #t else (member 'd '(b c)))
   = (if (equal 'd 'b) then #t else (member 'd '(c)))
   = (if (equal 'd 'c) then #t else (member 'd '()))
   = #f
```


`Append` Function

Append second list at the end of the first:

`(append '(a b c) '(d e f))` ⇒ `(a b c d e f)`

Definition:

```
(append '() B) = B
(append (cons E L) B) = (cons E (append L B))
```

In Scheme: `(define (append La Lb)`

```
  (cond ((null?  La) Lb)
  (else (cons (car La) (append (cdr La) Lb)))))
```

Tracing:

```
(append '(a b) '(c d))
   = (cons 'a (append '(b) '(c d)))
   = (cons 'a (cons 'b (append '() '(c d))))
   = (cons 'a (cons 'b '(c d)))
   = (cons 'a '(b c d))
   = '(a b c d)
```

`Reverse` Function

(reverse '(1 2 3)) ⇒ (3 2 1)

Definition:

```
(reverse '()) = '()
(reverse (car E L)) = (append (reverse L) E)
```

In Scheme:

```
(define (reverse L)
  (cond ((null?  L) '())
  (else (append (reverse (cdr L)) (list (car
L))))))
```

What is complexity of `reverse`?

Improving Performance: Accumulators and Auxiliary Functions

Motivation:

(reverse (E_1 E_2 ... E_n) =
  (cons E_n (reverse (E_1 ... E_{n-1}))))

Have a function that has current list and result so far - accumulator:

```
(rev L Result) = if L = '() then Result
    else (rev cdr(L) (cons (car L) Result))
```

But we want our `reverse` to have only one parameter! Result:

```
(define (rev L Result)
  (cond ((null?  L) Result)
  (else (rev (cdr L) (cons (car L) Result)))))
(define (reverse-2 L) (rev L '()))
```

What is complexity of `reverse-2`?

Tracing `Reverse-2`

```
(define (rev L Result)
  (cond ((null?  L) Result)
  (else (rev (cdr L) (cons (car L) Result)))))
(define (reverse-2 L) (rev L '()))
```

```
(reverse-2 '(1 2 3)) = (rev '(1 2 3) '())
   = (rev '(2 3) '(1))
   = (rev '(3) '(2 1))
   = (rev '() '(3 2 1))
   = '(3 2 1)
```


Other Types of Recursion

Function `leftmost`

```
(leftmost '((((2 3)) (4 (5 6))) 7))
  ⇒ 2
```

```
(define (leftmost l)
  (cond ((null? l) '())
        ((not (list? (car l))) (car l))
        (else (leftmost (car l)))))
```

Trace:

```
(leftmost '((((2 3)) (4 (5 6))) 7))
  = (leftmost '(((2 3)) (4 (5 6))))
  = (leftmost '((2 3)))
  = (leftmost '(2 3))
  = 2
```

This is `car` recursion.

Other Types of Recursion

Function `flatten`

```
(flatten '(((2 3) (4 (5 6))) 7))
  ⇒ (2 3 4 5 6 7)
```

```
(define (flatten l)
 (cond
  ((null? l) '())
  ((not (list? (car l)))
    (cons (car l) (flatten (cdr l))))
  (else
    (append (flatten (car l))
            (flatten (cdr l))))))
```

Tracing:

```
(flatten '((2 3) (4 (5 6))))
 = (append (flatten '(2 3)) (flatten '(4 (5 6))))
 = (append (cons 2 (flatten (3))) (flatten '(4 (5 6))))
 = (append (cons 2 (cons (3 (flatten '())))
           (flatten '(4 (5 6))))
 = (append (cons 2 (cons (3 '()))) (flatten '(4 (5 6))))
 = (append (cons 2 (3)) (flatten '(4 (5 6))))
 = (append (2 3) (flatten '(4 (5 6))))
 = ...
 = (append (2 3) (4 5 6))
 = (2 3 4 5 6)
```

This is `car` and `cdr` recursion.

Expression Evaluation

Innermost Evaluation(eager)

Suppose we want to evaluate <*name*><*actual parameter*>.

- Evaluate the expression represented by <*actual-parameter*>
- Substitute the result for the formal in the function body
- Evaluate the body
- Return value as result

This is *inside-out* evaluation of expressions, or, *call by value* (e.g., C, Pascal, Turing).

Example: `(successor (+ 2 3))`

Expression Evaluation(cont'd)

Selective evaluation - the ability to evaluate some parts of an expression and ignore others.
If statement:
(`if` *condition* $expr_1$ $expr_2$)

- *condition* evaluates to `#t` or `#f`.
- evaluate $expr_1$ only if *condition* is `#t`.
- otherwise, evaluate $expr_2$.
- so, either $expr_1$ or $expr_2$ is evaluated, but not both.

To evaluate recursive functions - just evaluate actual parameters and substitute them into the function body:

```
( if (null?  '(a b)) 0 (+1 (length (cdr '(a b)))))
```

becomes

```
(+1 (length (cdr '(a b))))
```


Alternatives to Innermost Evaluation

Many functional languages use *Outmost Evaluation* from Left to Right ("lazy"):

- Substitute the actuals for the formals in function body
- Evaluate the body
- Return its value as the result

This is basically "call by name" - evaluate parameters only when needed.

Scheme does not do that (although some dialects of Lisp do).

Expression Evaluation - Comparison

"Innermost" (I) and "outermost" (O) evaluations, when converge, yield the same result. But in many cases I does a lot of redundant work, and sometimes O terminates when I does not.

Short-Circuit Evaluation - functions `and` and `or` perform short-circuit evaluation of boolean expressions in which the right operand is evaluated only if it has to be:

- `( and E F )`
  - if E is false, no need to check F

```
(cond
  ((and (list? a) (equal? (car a) 'b))) ...
```

- `(or E F)`
  - if E is true, no need to check F

```
(cond
  ((or (not (list? a)) (equal? (car a) 'b))))
```


Expression Evaluation - Cont'd

Thus, `(and #f F)` terminates even if F is a non-terminating expression, and similarly, `(or #t F)`.

However, the following function does not do short-circuit evaluation:

```
(define (or x y)
  (cond ((equal? x #t) #t)
        ((equal? y #t) #t)
        (else #f)))
```

Under innermost evaluation, both subexpressions `E` and `F` in `(or E F)` are evaluated before they are substituted into the function body. So, if `E=true` and `F` is a non-terminating computation, then `(or E F)` will not terminate.

Expression Evaluation - Cont'd

Under outermost evaluation, actuals are substituted for formals without being evaluated:

```
( or #t F ) =
  (if #t #t F)
```

The right side immediately simplifies to `true`, and the nonterminating computation `F` is never reached.

Scheme uses innermost evaluation, so the functions `or` and `and` have to be provided by the language. They cannot be user-defined.

Proofs of Correctness

- Can use induction to prove properties of recursively defined procedures
- (Pure) functional languages are *referentially transparent*, so can substitute equals for equals in the proof. No side-effects.

Prove the following theorem:
`(length (append X Y)) = (length X)+(length Y)`:

1. Base case: `X = '()`

```
(length (append '() Y)) = (length Y)
   ; definition of append
(length '()) = 0 ; definition of length
(length Y) = 0 + (length Y)
  = (length '()) + (length Y)
```

2. Inductive hypotheses:
`(length (append X Y)) = (length X)+(length Y)`

Proofs of Correctness, Cont'd

3. Inductive step: `X = (cons E L)`

```
(length (append (cons E L) Y))
  = (length (cons E (append L Y)))
      ; definition of append
  = 1 + (length (append L Y))
      ; definition of length
  = 1 + (length L) + (length Y)
      ; inductive hypotheses
(length (cons E L)) = 1 + (length L)
      ; definition of length
1 + (length L) + (length Y) (1)
  = (length (cons E L)) + (length Y) ; by (1)
  (length X) + (length Y)
```

So,
`(length (append X Y)) = (length X)+(length Y)`

Another Proof

Prove: `(member A X)` → `(member A (append X Y))`

1. Base case: `X = '()`

```
(member A '()) = false, so
(member A '()) → (member A (append '() Y))
```

2. Inductive hypotheses:

`(member A X)` → `(member A (append X Y))`

3. Inductive step: `X = (cons E L)`

```
(member A (append (cons E L) Y))
  = (member A (cons E (append L Y)))
      ;definition of append
  = (A = E) or (member A (append L Y)) (1)
      ;definition of member
(member A (cons E L))
  = (A = E) or (member A L) ;definition of member
  → (A = E) or (member A (append L Y))
      ;logic and inductive hypothesis
  = (member A (append (cons E L) Y))
      ;from (2)
```


`Map` function

Scheme provides a function `map` to apply a function `f` to each element of a list x.

Examples:

```
(map square '(1 4 5 2)) ⇒ (1 16 25 4)
(map first '((1 2) (3 4) (5 6))) ⇒ (1 3 5)
(map second '((1 2) (3 4) (5 6))) ⇒ (2 4 6)
(map + '(1 2 3) '(4 5 6)) ⇒ (5 7 9)
```


Association Lists

- *a-list* is a list of pairs.
- traditionally used to implement dictionaries and environments
- a binding with key and value:
  `((a 1) (b 2) (c 3) ...)`

Operations on association lists:

- `bind` returns an a-list with new binding for a key
- `bind-all` binds keys in list `key` to values in list `values`
- `assoc` returns most recent binding for a key.

Examples:

```
(bind-all '(a b c) '(1 2 3))
   ⇒ ((a 1) (b 2) (c 3))
(bind 'a 1 '((b 2) (c 3)))
   ⇒ ((a 1) (b 2) (c 3))
(assoc 'a '((a 1) (b 2) (c 3)))
   ⇒ (a 1)
```

Association Lists (Cont'd)

Implementation:

```
(define (bind key value env)
   (cons (list key value) env))

(define (bind-all keys values env)
   (map list keys valus))

(assoc key env)
   ;; supported directly by Scheme
```


Higher-Order Programming

A function is called *higher order* if either of its arguments or its results are themselves functions. `map` and `apply`.

Example (first-order):

```
(define (remove-even l)
  (cond ((null? l) '())
    ((even? (car l)) (remove-even (cdr l)))
    (else (cons (car l) (remove-even (cdr l))))))
```

Can rewrite function `remove-even` to function `remove-if`:

```
(define (remove-if f l)
  (cond ((null? l) '())
    ((f (car l)) (remove-if f (cdr l)))
    (else (cons (car l) (remove-if f (cdr l))))))
```

Here `f` is a predicate (a function).

Mapping Functions Across List Elements

A *filter* is a function that copies a list, making (useful) changes to its elements as they are copied.

`copy` makes no changes:

```
(define (copy l)
  (cond ((null? l) '())
        (else (cons (car l) (copy (cdr l))))))
```

`copysq` squares each element:

```
(define (copysq l)
  (cond ((null? l) '())
        (else (cons (square (car l))
        (copysq (cdr l))))))
```

So, do we need to write similar functions every time we need to perform a similar task?

`Map` function

Function `map` is used to apply a function `f` to each element of a list `x`. Example:

`(map square '(1 4 5 2))` ⇒ `(1 16 25 4)`

Definition of `map`:

```
(define (map f l)
  (cond
    ((null?  l) '())
    (else (cons (f (car l)) (map f (cdr l))))))
```

- `map` can use anonymous functions as its argument:
  `(map (lambda (x) (* x 2)) '(1 2 3 4))`
- `map` basically implements a loop:
  for each element `a` of a list
    do something with `a`
  return a list of the results

Selective Copying

`remove-if` copies list element `a` unless predicate `f` is true on `a`.

```
(remove-if even '(1 2 3)) ⇒ (1 3)
(remove-if symbol? '(a b 2)) ⇒ (2)
```

First argument can be an anonymous (lambda) function:

```
(remove-if (lambda (x) (equal? (mod x 2) 0))
'(1 2 3)) ⇒ (1 3)
```


Function `reduce`

How do we write a function to add all elements of the list?

```
(define (sum-all l)
  (cond ((null? l) 0)
    (else (+ (car l) (sum-all (cdr y))))))
```

And to multiply all elements of the list?

```
(define (product-all l)
  (cond ((null? l) 1)
    (else (* (car l) (product-all (cdr y))))))
```

We want to generalize such functions (REUSE!).

```
(define (reduce f l e)
  (cond ((null? l) e)
    (else (f (car l) (reduce f (cdr l) e)))))
```

Now,

```
(define (add x y) (+ x y))
(define (multiply x y) (* x y))
(reduce add '(1 2 3 4 5) 0) ⇒ 15
(reduce multiply '(1 2 3 4 5) 1) ⇒ 120
```


How does `reduce` work?

```
(reduce add '(1 2 3) 0
  = (add 1 (reduce add '(2 3) 0))
  = (add 1 (add 2 (reduce add '(3) 0)))
  = (add 1 (add 2 (add 3 (reduce add '() 0))))
  = (add 1 (add 2 (add 3 0)))
  = (add 1 (add 2 3))
  = (add 1 5)
  = (add 6)
```

In general case, `(reduce f` $(l_1, l_2, ...l_n)$ `v) =`
`(f` l_1 `(f` l_2 `( ... (f` l_n `v) ...)))`

Creative use of `reduce`

Can define a lot of functions using `reduce`, like `append`, `length`, etc.

How to do that:

1. write the original function, e.g., `length`.
2. `e` is whatever the function returns on base case (i.e., on the empty list).
3. The problem is to write `f`
   (a) Use definition of `reduce` and example to figure out what `f` is supposed to produce
   (b) Write a (named or unnamed) function `f`.

Writing `length` in terms of `reduce`

```
(define (length l)
  (cond ((null?  l) 0)
    (else (+1 (length (cdr l))))))
```

So, `e` is 0.

Now, try `(length '(1 2 3)` using `reduce`:

`(f 1 (f 2 (f 3 0)))`:

We want `(f 3 0)` to return `1`,

`(f 2 (f 3 0))` to return `2`, and

`(f 3 (f 2 (f 3 0)))` to return `3`.

So, function `f` should take 2 parameters and return second + 1:

`(lambda (x y) (+1 y))`

Therefore, `length` in terms of `reduce` is:

```
(length L) =
  (reduce (lambda (x y) (+1 y)) L 0)
```


`reduce_if even` in terms of `reduce`

Function `(reduce-if even L)` should return `'()` when `L` is an empty list. So, `e` is `'()`.

Now, try `reduce-if even '(2 5 3 4)` using `reduce`:

`(f 2 (f 5 (f 3 (f 4 '() ))))`:

We want `(f 4 '())` to return '()
 (since 4 is even)

`(f 3 (f 4 '()))` to return `(3)`,

`(f 5 (f 3 (f 4 '())))` to return `(5 3)`, and

`(f 2 (f 5 (f 3 (f 4 '()))))` to return `(5 3)`.

So, what is `f`? `f` takes an element and and a list, checks if the element is even, and if not, cons it to the list:

```
(define (temp a y)
  (if (even a) y (cons (a y))))
```

`reduce-if even` in terms of `reduce`:

`(reduce-if even L) = (reduce temp L '())`

Unrestricted Lambda Functions

Goal: want to define functions to take arbitrary number of arguments, e.g.

`(add 1 3 5 7 9)` ⇒ 25

`(add 1 3 5 7 9 11)` ⇒ 36

`(add 1 3 5 7 9 11 13)` ⇒ 39

We do this using *unrestricted* Lambda functions.

Usual form of lambda functions:

(lambda (var$_1$ var$_2$ ...) expr...)

Unrestricted form:

`(lambda var expr...)`

Invocation:

((lambda var expr...) operand$_1$ operand$_2$...)

- If the operands operand$_1$ operand$_2$... have the values arg$_1$ arg$_2$ ..., then the variable `var` is a *list* of arguments (arg$_1$ arg$_2$...).

- The body `expr ...` is evaluated with this binding in effect.

Examples of Unrestricted Lambda Functions

- Function `add`

```
(define add
  (letrec (list-add
    (lambda (ls)
      (if (null? ls) 0)
         (+ (car ls) (list-add (cdr ls))))))
  (lambda args (list-add args)))
```

`(add 1 2 3 4 5)` ⇒ 15

Remember that `args` is a list!

- Function `list`

```
(define list (lambda args args))
```

- Another version of `add`

```
(define add
  (lambda args
    (if (null? args) 0
      (+ (car args) (apply add (cdr args))))))
```

Unrestricted Lambda Functions (Cont'd)

In Scheme, a number of functions are defined to take an arbitrary number of arguments: `+`, `*`, `max`, `min`:

`(+ 1 3 5 7 9)` $\Rightarrow$ 25
`(+ 5)` $\Rightarrow$ 5
`(+)` $\Rightarrow$ 0
`(* 2 4 6)` $\Rightarrow$ 1
`(* 5)` $\Rightarrow$ 5
`(*)` $\Rightarrow$ 1

Memory Management Overview

- How is memory allocated?
- Notion of equality
- How is memory deallocated?
  - When to deallocate?
  - Algorithms to deallocate

This is section 10.6 of Sethi.

Memory Allocation

• Each list can be represented as a sequence of applications of `cons`:

`'(1 2) = (cons 1 '(2)) = (cons 1 (cons 2 '()))`

• Operator `cons` ends up creating a cons pair, with a pointer to head and a pointer to tail:

So, what happens here?

`(define x '(1 2 3)`

`(define y (cons (car y) (cdr y)))`

Notion of Equality

Three operators: `eqv` (will not cover), `equal?` and `equ?`.

- all three have the same results when applied to symbols:

`(eqv?/eq?/equal? 'hello 'hello)` $\Rightarrow$ `#t`

- `eq?` tests that the two operators have identical pointers
- `equal?` recursively checks that the two arguments are lists with equal elements. May fail on circular data structures.

Example:

`(equal? 'hi 'hi)` $\Rightarrow$ #t
`(eq? 'hi 'hi)` $\Rightarrow$ #t
`(equal? '(hi bye) '(hi bye))` $\Rightarrow$ #t
`(eq? '(hi bye) '(hi, bye))` $\Rightarrow$ #f

Why?

Allocation and Deallocation

- Cells not in use have to be recovered, otherwise program runs out of memory.
- A cell is no longer in use if nothing points to it
- *garbage collection* - reallocates memory without explicit instructions

Technique - have a free list - stack of cells
  - A pop returns a freshly allocated cell
  - A push returns a cell back onto the stack

Examples

```
'( 1 )


(let ((x '( 1 ))) (x))


(define x '( 1 ))
x
```


Approaches to cell deallocation

Lazy approach:
- Wait until memory runs out and only then collect dead cells.
- Advantage - if enough memory is available, the need for collecting cells may never arise
- Disadvantage - once garbage collecting starts, all other work comes to a halt. Example: emacs.

Eager approach:
- Each time a cell is reached, check whether the cell will be needed after the operation; if not, deallocate the cell by placing it on the free list.
- Advantage - work does not come to a halt
- Disadvantage - to implement it, need space.

Implementing Eager Approach

- For each cell, store *reference count* - the number of pointers to the cell. - When a new pointer appears, increment count
- When count becomes zero:
  - Decrement reference count of cell pointed to by `cdr` and continue recursively
  - Free cell

Example:

Implementation of Garbage Collection

Mark-Sweep approach
1. *Mark phase*. Start from all global variables. Mark all the cells that can be reached by following pointers.
2. *Sweep phase*. Sweep (scan) through memory, starting at "beginning", looking for unmarked cells. Unmarked cells are returned to the free list. Unlike mark phase, this starts at one end of memory and looks at every cell.

Example:

Optimization of Mark-Sweep Algorithm

This is called *copying collector*.
- Divides memory into two halves, the *working* and the *free*
- Allocates cells from working half
- When no more cells in working, copies all reachable cells into consecutive locations in the free half.
- Switches roles of working and free halves

Advantage: No need to to do sweep part of the mark-sweep algorithm. Just copy reachable cells (those marked by marking phase of the algorithm).

Very effective in a virtual memory system

Summary

Programming in untyped functional language Scheme.

- History and Introduction
- Primitive and compound expressions
- Local and Global Definitions
- Lists and their manipulation
- Recursive reasoning
- Expression evaluation
- Proofs of correctness
- Higher-order programming
- Miscellaneous stuff:
  - Association lists
  - Functions with arbitrary number of parameters
- Memory management

Functional Programming - Review

- Do away with variables and reassignments (referential transparency)
- No notion of state (in pure functional programming)
- Declarative rather than imperative approach
- Allows to analyze programs more easily

Software engineering issues:
- want to construct programs more reliably.
- OO programming: better code faster. That's good but we want BEST, i.e., no bugs.
- want to do away with testing. Testing can prove presence of bugs rather than their absence. Want to replace testing with reasoning

Functional Programming - Review (Cont'd)

Functional programming allows to develop programs which submit to logical analysis relatively easily.

- Can make assertions about programs and prove them correct.
- All datatypes have constructors and base cases, so can reason by induction
- Allows to write "better" programs
- Can write good code in imperative language (hard) or bad code in functional languages

Problems:
- Slow ("efficiency does not matter")
- Primitive user interface

Functional vs Imperative programming

The development cycle:

Imperative:
- Using informal language a specification may be open to interpretation.
- Using appropriate testing strategies we can improve confidence, but not in any measurable way. - Mistakes/bugs are common and difficult to spot and correct.

Functional:
- Using logic, we can state the specification exactly.
- Using mathematics we may be able to prove useful properties of our programs.

Functional vs Imperative Programming (Cont'd)

The Development Language:

Imperative:
- Using structured programming or object oriented techniques we can reuse code.
- Can partition the program into more manageable chinks.

Functional:
- Using structured programming or OO techniques we can reuse code.
- Can partition the problem into easy to use chunks. These are often "higher-level" abstractions. Can write things in Scheme which are difficult or impossible to write in a traditional language

Functional vs Imperative Programming (Cont'd)

The Run-time System:

Imperative:
- The compiler can produce fast compact code taking a fixed amount of memory
- Parallel processing is not possible, in general
- Fancy GUI's may be added

Functional:
- Interpretors are available
- The memory requirements are large and unpredictable
- Parallel processing is possible
- Fancy GUI's may be added, although with great difficulty