Scheme

- Lisp Dialect.
- Expression Oriented.
- Functional style (not pure).
- Interpreted, but compilers exist.
- Simple but expressive.

Expressions

Also called S-Expressions (for symbolic).

Recursively Defined, either:

- An atom: a variable or value, or
- A list of S-Expressions.

Primitive Values: Atoms

Atoms can be:

- Numbers: Integers, Reals, Rationals.
- Symbols (e.g. `fred`, `g42`, `*`).
- Booleans (e.g. `#t` or `#f`).
- Strings (e.g. `"hello world"`)
- Characters (e.g. `#\c`, `#\C`, `#\newline`)

Combinations

(A list of S-Expressions)

A combination is a function application.

In general: `(<f-expr> <expr-1> <expr-2> ... )`

For example: `(+ 5 6)` or `(display "hello world")`

Read-Eval-Print Loop

Interactive interpreter's infinite loop:

1. **Read** an S-Expression.
2. **Evaluate** the S-Expression (in the global environment).
3. **Print** the expression's value.

Eval: Atoms

Variables: (if the expression is a symbol)

- Look up variables value in current environment.
- Return the value.

Values: (Any other primitive type)

- Return it.

Eval: Combinations

Given: `(<f-expr> <expr-1> <expr-2> ... )`

- Evaluate all expressions to get $f, e_1, e_2, \ldots$
- Apply f as a function to arguments $e_1, e_2, \ldots$
- Return the result of the function.

Evaluate?

- `1`
- `"hi there"`
- `(+ 5 4)`
- `(1 2 3)`
- `(+ (* 2 3) 4)`

Primitive Procedures

Builtin functions defined in global environment.

arithmetic: `+`, `-`, `*`, `/`, `sqrt`, and `abs`
For example: `(+ 1 2 3 4)` or `(*)`

Relational (numbers): `=`, `<`, `>`, `<=`, and `>=`
For example: `(< 1 2)` or `(< 1 2 3)` or `(<)`

Predicates: `number?`, `string?`, `procedure?`, etc.
For example: `(procedure? +)` or `(number? +)`

... and many more.

Defining Variables

Just like this: `(define x 5)`

Or even: `(define answer (* 1 2 3 4))`

But how does the evaluator handle this?

Special Forms

Not every list of S-Expressions is a combination.

If first element of list is symbol naming a special form. Then evaluate special form in a special way.

Why is this necessary?

Conditionals ...

conditionals: `(if <test> <consequent> <alternate>)`
For Example:

```
(if (> x 5)
    5
    x
)
```

boolean relations: `or` and `and`
For Example: `(or (< 4 3) #t (/ 1 0))`

Defining Functions

```
(define (<fname> <formal-1> <formal-2> ...) <body>)
```

For example:

```
(define (double x) (* 2 x))
```


Temporary Variables

Special form: `let`

```
(let ((<var-1> <expr-1>)
      (<var-2> <expr-2>)
      ...
      (<var-n> <expr-n>)
     )
     <body>
)
```

For Example:

```
(let ((x 50)
      (y 10)
     )
     (+ x y)
)
```


Quoting

How do we get symbols for values? E.g. `(define plus-sym +)`

Answer: Use `quote`, `(define plus-sym (quote +))`

Or use a simpler special notation: `(define plus-sym '+)`

Quote tells evaluator not to evaluate. Just to return the symbolic value of the atom.

What is `(number? (quote 5))`?

Lists

```
(list 1 "hi" 'bob)
```

Functions: `list`, `length`, `equal?`, `car`, `cdr`, `cons`, etc.
Predicates: `list?`, `pair?`, and `null?`.

Creating Lists:

```
(cons 1 (cons "hi" (cons 'bob '())))
'(1 "hi" bob)
(quote (1 "hi" bob))
```

cons, car, cdr

Who came up with these names?

`cons` stands for *construct*.

From original Lisp implementation on IBM 704 where "address" or "decrement" part of memory could be accessed.

`car` stands for "Contents of Address part of Register"

`cdr` stands for "Contents of Decrement part of Register"

Why do we still use these names? Because of contractions:

```
(define (cadr x) (car (cdr x)))      ;; second element of list
(define (cdar x) (cdr (car x)))      ;; ??
(define (cadaaar x) (car (cdr (car (car (car x))))))
```


List Dissecting

```
(define (sum lst)
   (if (null? lst)
       0                              ; base case
       (+ (car lst) (sum (cdr lst)))      ; recursive call
   )
)
```

and

```
(define (double-list lst)
   (if (null? lst)
       '()
       (cons (double (car lst)) (double-list (cdr lst)))
   )
)
```


Linear Recursive Process

```
guile> (sum '(1 2 3))

(sum '(1 2 3))
=> (+ 1 (sum '(2 3)))
=> (+ 1 (+ 2 (sum '(3))))
=> (+ 1 (+ 2 (+ 3 (sum '()))))
=> (+ 1 (+ 2 (+ 3 0)))
=> (+ 1 (+ 2 3))
=> (+ 1 5)
=> 6
```


Tail Recursion

Linear Recursion builds up call stack to depth of recursion.

If last thing a function does is make a recursive call, that recursive call's stack frame replaces the current functions stack frame.

```
;; return last element in list
(define (last lst)
   (if (null? (cdr lst))
       (car lst)
       (last (cdr lst))
   )
)
```

Iterative Process

```
guile> (last '(1 2 3))

(last '(1 2 3))
=> (last '(2 3))
=> (last '(3))
=> 3
```


Tail Recursive `sum`

Can we make `sum` tail recursive?

Trees

Lists can contain lists.

```
'(1 (2 3) ((4) 5))
```

Tree Recursion:

```
(define (sum-tree t)
   (cond ((null? t)   0)
         ((number? t) t)
         ((pair? t)   (+ (sum-tree (car t)) (sum-tree (cdr t))))
         (else (error "non-number in tree"))
   )
)
```


Lists and Memory

Appending two lists:

```
(define (append first second)
   (if (null? first)
       second
       (cons (car first) (append (cdr first) second))
   )
)
```

What happens below?

```
(define x '(1 2 3))
(define y '(a b c))
(define z (cdr x))
(define w (append x y))
```

Numbers

Three types of numbers *integers*, *reals*, and *rationals*.

There are different internal representations for the same number.

Integers can be any size.

Ever wonder what `(factorial 1000)` is?

Equality

Testing for equality:

`eq?` tests pointer equality. Equal numbers may not be `eq?`

`eqv?` tests pointer and number equality.

`equal?` tests *deep* equality (lists, trees, numbers, not procedures).

`=` tests number equality only. Non numbers are errors.

Equality Examples

How about?

```
(eq? 5 5)
(eq? 'the-little-red-hen 'the-little-red-hen)
(eq? "hi there" "hi there")
(eq? (list 1 2) (list 1 2))
(eq? (factorial 200) (factorial 200))
(eqv? 5 'a)
(eqv? (factorial 200) (factorial 200))
(equal? (list 1 2) '(1 2))
(= 5 'a)
```


First Class Functions

Like numbers, Scheme functions are first class citizens.

They

- can put stored in variables or lists,
- can be passed to other functions, and
- can be returned from other functions.

1st Class Func Examples

Consider:

```
guile> (define x 10)
guile> +
#<primitive-procedure +>
guile> x
10
```

What is special about +?

```
guile> (define plus +)
guile> (plus 1 2 3 4 5)
15
```


Data Abstractions

Providing an interface to a data type. Actual implementation of data type can change.

Binary Tree abstract data type, provide *constructor* and *selectors*:

- `(make-tree <data> <left-child> <right-child>)` ⟶ `<tree>`
- `(tree-data <tree>)` ⟶ `<data>`
- `(tree-left-child <tree>)` ⟶ `<left-child>`
- `(tree-right-child <tree>)` ⟶ `<right-child>`

Someone can provide binary search tree facilities `insert`, `lookup`, and `delete` using this abstract binary tree.

Procedural Abstractions

Recall `double-list`. Other functions, e.g. `square-list`, have same structure.

```
(define (double-list lst)
   (if (null? lst)
       '()
       (cons (double (car lst)) (double-list (cdr lst)))
   )
)
```

Abstract that Structure:

```
(define (my-map func lst)
   (if (null? lst)
       '()
       (cons (func (car lst)) (my-map func (cdr lst)))
   )
)
```


First-Order Funcs

First-Order Functions take other functions as arguments. E.g. `map`.

What do these do?

```
(map double '(1 2 3))
(map sqrt '(1 4 9))
(map symbol? '(3 a 6 b 9 c))
(map + '(1 2 3) '(4 5 6))
```

What if no existing function does exactly what you want?

Lambdas

Anonymous Functions: `lambda`

```
(map (lambda (x) (* x x)) '(1 2 3))
```

Compare

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

Rename this `mystery` function:

```
(define (mystery lst)
   (sum (map (lambda (x) 1) lst))
)
```


Abstracting Computations

Abstracting computational patterns gives, flexibility, power, and glue.

Can we abstract `sum` so that we can easily implement `product`?

```
(define (sum lst)
   (if (null? lst)
       0                              ; base case
       (+ (car lst) (sum (cdr lst)))      ; recursive call
   )
)
```


Fold

Fold is useful for many things.

```
(define (append first second)
   (foldr cons second first)
)

(define (length lst)
   (foldr (lambda (x y) (+ 1 y)) 0 lst)
)

(define (product lst)
   (foldr * 1 lst)
)
```

We defined `foldr` (fold right) which starts at the rear and works forwards. `foldl` (fold left) does the opposite and is also useful.

Funcs as Return Values

Recall constant function `(lambda (x) 1)`.

In general we might want a constant function to return any constant.

```
(define (const-func val)
   (lambda (x) val)
)
```

Now, rewrite `mystery` as:

```
(define (length lst)
   (sum (map (const-func 1) lst))
)
```

Benefits . . .

Compare:

```
(define (length lst)
   (sum (map (const-func 1) lst))
)

(define (length lst)
   (if (null? lst)
       0
       (+ 1 (length (cdr lst)))
   )
)
```


Not Convinced?

How about:

```
(define (sum lst)
   (foldr + 0 lst)
)

(define (sum lst)
   (if (null? lst)
       0
       (+ (car lst) (sum (cdr lst)))
   )
)
```


Meta-Circular Interpreter

A Scheme interpreter written in Scheme.
(See SICP: Ch 4 or 341 Winter 1999 Homework 3)

Why?

- Tweaking language, trying new constructs.
- Debugging.
- Because we can (and it is easy)!
- Educational value.

MCI: internals

Eval: `(eval <expr> <env>)` evaluates S-Expression `<expr>` in environment `<env>`.

Apply: `(apply <proc> <arg-list>)` does procedure applications.

Environment: List of environment frames.

Environment Frame: List of variable bindings.
E.g. `((x 10) (y "hi"))`

Procedures: Tuples of `(<param-list> <body> <env>)`.

Read-Eval-Print Loop

```
(define the-global-env (...))

(define (read-eval-print)
   (display "MCI> ")
   (display (eval (read) the-global-env))
   (newline)
   (read-eval-print)
)
```


Environments

Recall evaluation procedure for symbols:

- Look up symbol's value in the *current environment*.
- Return the value.

So, what is the current environment?

At the *read-eval-print* loop prompt it is the global environment.

What about inside a function or a let?

Environments (cont)

An *environment frame* is a set of variable bindings:

```
((x 10) (y 20) (+ #<primitive-procedure +>) ...)
```

The *environment* is a list of environment frames.

To look up `x` we first check each frame until we find it.

The Global Environment

Contains only the global environment frame:

```
((+ #<primitive-procedure +>)
 (- #<primitive-procedure ->)
 ...
 (car #<primitive-procedure car>))
```

After `(define x 10)` we have:

```
((x 10)
 (+ #<primitive-procedure +>)
 (- #<primitive-procedure ->)
 ...
 (car #<primitive-procedure car>))
```

let Environments

In what environment do we evaluate (+ x y) in:

```
(let ((x 10)
      (y 100)
     )
     (+ x y)
)
```


let Environments (cont)

```
(((x 10)
  (y 100))
 ((+ #<primitive-procedure +>)
  (- #<primitive-procedure ->)
  ...
  (car #<primitive-procedure car>)))
```


eval

Scheme code for evaluation procedure discussed in first Scheme lecture.

```
(define (eval expr env)
   (cond ((symbol? expr) (lookup env expr))
         ((literal? expr) expr)
         ((special-form? expr) (eval-special-form expr env))
         ((pair? expr)
             (let ((exp-list (map (lambda (arg) (eval arg env)) expr)))
                (apply (car exp-list) (cdr exp-list))
             )
         )
   )
)
```


Lexical Scoping

(also called static scoping)

Variables are bound when procedures are defined not when they are called. For example:

```
(define x 10)
(define (x-plus y) (+ x y))

(x-plus 2)

(define (x-plus-double x) (x-plus (* 2 x)))

(x-plus-double 2)
```

In a procedure application, the *free variables* in the body of a procedure are evaluated in the procedure's *defining environment*.

A procedure is . . .

Represented by tuple consisting of:

- A parameter list.
- A body.
- The procedure's defining environment.

MCI procedure selectors: `(proc-params proc)`, `(proc-body proc)`, and `(proc-env proc)`.

Applying Procedures

Scheme primitive `(apply <proc> <arg-list>)`
For Example: `(+ 1 2 3)` is same as `(apply + '(1 2 3))`

For Meta-Circular Interpreter:

```
(define (apply proc args)
   (let ((new-frame (map list (proc-params proc) args)))
        (eval (proc-body proc) (cons new-frame (proc-env proc)))
   )
)
```


Defining Environments

Consider:

```
guile> (define z 10)
guile> (define foo (lambda (x y) (+ x y z)))

guile> (define bar (let ((z 100)) (lambda (x y) (+ x y z))))
```


Eval (foo 2 3)

Add local environment frame,

```
((x 2) (y 3))
```

to defining environment to get:

```
(((x 2)
  (y 3))
 ((bar #<procedure bar (x y)>)
  (foo #<procedure foo (x y)>)
  (z 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure ->)
  ...
  (car #<primitive-procedure car>)))
```

Evaluate body `(+ x y z)` in this environment.

Eval `(bar 2 3)`

Add local environment frame,

```
((x 2) (y 3))
```

to defining environment to get:

```
(((x 2)
  (y 3))
 ((z 100))
 ((bar #<procedure bar (x y)>)
  (foo #<procedure foo (x y)>)
  (z 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure ->)
  ...
  (car #<primitive-procedure car>)))
```

Evaluate body `(+ x y z)` in this environment.

`lambda` and `let`

Recall how procedure application works:

```
(<func> <arg-1> <arg-2> ... )
```

Here `<func>` is just an S-expression that evaluates to a procedure. E.g.

```
((if #f car cdr) '(1 2 3 4))
```

```
(define z 10)
(define w -5)
((lambda (x) (* x x)) (+ z w))
```


`lambda` and `let` (cont)

Compare:

```
(define z 10)
(define w -5)
```

```
((lambda (x) (* x x)) (+ z w))
```

with

```
(let ((x (+ z w)))
     (* x x)
)
```

`let` is not so special. It is a source-to-source transformation to `lambda`.

Nested `let`s: `let*`

Derived form: `let*`

```
(let* ((<var-1> <expr-1>)        ⇒   (let ((<var-1> <expr-1>))
       (<var-2> <expr-2>)                  (let ((<var-2> <expr-2>))
       ...                                   ...
       (<var-n> <expr-n>)                      (let ((<var-n> <expr-n>))
      )                                             <body>
      <body>                                   )
)                                          )
                                     )
```

Extended Example

```
(define (my-map func lst)
   (if (null? lst)
       '()
       (cons (func (car lst))
             (my-map func (cdr lst)))))

(define (const-func val)
   (lambda (x) val))

(define const1 (const-func 1))
(define const5 (const-func 5))

(my-map const1 '(1 2 3))
(my-map const5 '(1 2 3))
```


Variable Argument Lengths

Scheme has primitive functions `+`, `list`, `map`, etc. that accept a variable number of arguments. E.g.

```
(list)          => ()
(list 1)        => (1)
(list 1 2)      => (1 2)
```

Can we define our own?

You Bet!

These are identical:

```
(define (my-list . args) args)
```

```
(define my-list (lambda args args))
```

Note this “dot” notation is the same thing you get when you `cons` something to a non-list.

What if we want to require some arguments?

You Bet! (cont)

```
(define (my-map func . lists) ...)
```

```
(define my-map (lambda (func . lists) ...))
```

Bootstrapping

Creating something from a simpler version. For Example:

```
(define (my-map1 func lst)
   (if (null? lst)
       '()
       (cons (func (car lst)) (my-map1 func (cdr lst)))
   )
)

(define (my-map func . lists)
   (if (null? (car lists))
       '()
       (cons (apply func (my-map1 car lists))
             (apply my-map (cons func (my-map1 cdr lists)))
       )
   )
)
```


The Power of Lambda

Look for powerful constructs in a language.

Lambda is powerful.

We have already seen that it can be used in place of `let`.

What else can it do?

`cons` cells

Scheme has primitive procedures `cons`, `car`, and `cdr`.

Can we get by without them? Yes! Define abstract `cons` cell using `lambda` for implementation:

```
(define (my-cons the-car the-cdr)
   (lambda (which)
      (cond ((eq? which 'car) the-car)
            ((eq? which 'cdr) the-cdr)
            (else (error "cons cell usage error"))
      )
   )
)

(define (my-car the-pair)
   (the-pair 'car)
)

(define (my-cdr the-pair)
   (the-pair 'cdr)
)
```


Manipulating Functions

We have seen:

- Function creation (static).
- Function application (static and dynamic).
- Functions in variables

We can also:

- Manipulate functions to dynamically create new functions.

Dynamic Function Creation

Facilitated by:

- `lambda` to create new functions.
- First-class-ness of functions.
- Persistent environments.

Currying

Named for Haskell B. Curry

- Given n argument function f
- Specify k arguments to get $n-k$ argument function g.
- If $g = \text{curry}(f, a_1, \ldots, a_k)$ then $g(a_{k+1}, \ldots, a_n) = f(a_1, \ldots, a_n)$

Currying (cont)

A `curry` for a two argument function, f:

```
(define (curry f x)
   (lambda (y) (f x y))
)
```

But we can use variable argument lengths to write a general `curry`:

```
(define (curry f . first-args)
   (lambda second-args (apply f (append first-args second-args)))
)
```


Uses for `curry`

Lambda definitions are hard to read, curried functions are more clear.

```
(define add-lists (curry map +))
(add-lists '(1 2 3) '(4 5 6))

(map (curry + 5) '(1 2 3))

(define positive-only (curry filter positive?))

(define sum (curry foldr + 0))
(define sum (curry apply +))
```

Composing

Define `compose` to take two single argument functions f and g and return a function h that is f composed with g. That is

$$h(x) = f(g(x))$$

For example:

```
(define my-cadr (compose car cdr))
(define my-cadaar (foldr compose identity (list car cdr car car)))
```


Reversing Arguments

Define `reverse-args` that takes a function f and returns a new function g that takes the arguments in reverse. That is:

$$h(x, y) = f(y, x)$$

More Examples

```
(define my-reverse (curry foldl (reverse-args cons) '()))

(define compose-list (curry foldr compose identity))
(define my-cadaar (compose-list (list car cdr car car)))

;; from eval definition
(map (lambda (arg) (eval arg env)) expr)
;; becomes
(map (curry (reverse-args eval) env) expr)
```


Building Blocks

We have seen some useful higher-order building blocks: `map`, `foldr`, `curry`, `compose`, and `reverse-args`.

Scheme could provide these.

Instead Scheme provides the equipment to make them and more.

Side-Effects!

Side-effects are:

- Doing I/O (read or write).
- Changing a variable value.
- Crashing: `(/ 1 0)`.

So far we have only seen the purely functional subset of Scheme.

Input and Output

write/display outputs a scheme object. E.g. `(display "hello world")`.

newline outputs a newline.

write-char outputs a character.

read reads in a scheme object.

read-char reads in a character.

readline (Guile only) reads in a line as a string.

Setting Variable Values

In general commands that change values end in `!`:

set! sets variable value: `(set! x 10)`

set-car! sets car of existing `cons` cell.

set-cdr! sets cdr of existing `cons` cell.

Use these sparingly!

Sequencing Operations

Use `begin` to sequence expressions:

```
(define x 0)
(define y 0)
(begin
   (display "enter two numbers: ")
   (set! x (read))
   (set! y (read))
   (display "the sum is ")
   (display (+ x y))
   (newline)
)
```

Return value is value of last expression.

With no side-effects sequencing expressions is not useful:

```
(begin
   (+ 1 2 3)
   (car '(1 2 3))
)
```

Sequencing Operations (cont)

Special forms `lambda`, `let`, and `cond` allow for sequencing operations:

```
(define (welcome)
   (display "hello world")
   (newline)
)

(cond (#f (display "never") 0)
      (#t (display "always") 1)
)

(let ((x 5))
     (display "inside let\n")
     (+ x 10)
)
```

Value of sequence is value of last expression.

Examples of `set!`

```
(define x 10)
(define (x-plus y) (+ x y)

(x-plus 2)

(set! x 100)

(x-plus 2)
```


`fib` Example

Ever wonder how many times recursive `(fib 30)` calls `+`?

```
(define (fib n)
    (if (<= n 1)
        n
        (+ (fib (- n 1)) (fib (- n 2)))
    )
)

(define plus +)
(define count 0)
(define (+ . args)
   (set! count (plus 1 count))
   (apply plus args)
)

(fib 30)                  => 832040
count                     => 1346268
```


List Surgery

Can set `car` and `cdr` of a `cons` cell with `set-car!` and `set-cdr!`.

```
(define x '(1 2 3))
(define y '(a b c))
(define z (cdr x))
(define w (append x y))

(set-cdr! x '(200 300))
(set-cdr! (cdr y) '(ccc))
(set-car! z 99)

x
y
z
w
```

Note: cannot `set-c[ad]r!` the empty list. Why?

Circular Lists

```
(define x '(1 2 3))
(set-cdr! (cddr x) x)

x
```


set! vs set-car!

`set!` sets binding in environment.

`set-car!` sets binding in heap.

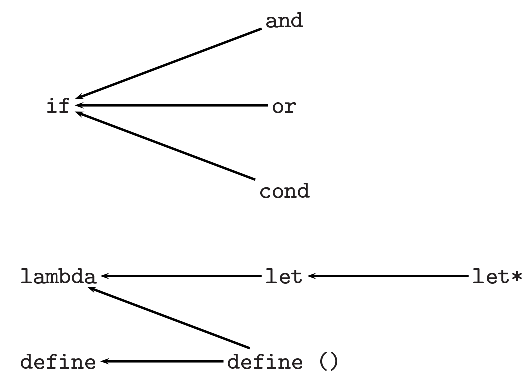
Compare:

```
(define (fix-car1 lst)
   (set! lst (cons 10 (cdr lst)))
)

(define (fix-car2 lst)
   (set-car! lst 10)
)

(define x '(1 2 3))
(define y '(1 2 3))

(fix-car1 x)
(fix-car2 y)

x
y
```


Derived Expressions

(aka Source-to-Source transformations)
These are special-forms that can be stated in terms of other special-forms.

Redundant in that Scheme is equally as expressive without them.



let->combination

Turn a symbolic `let` expression into a `lambda` and application.
For example:

```
guile> (let->combination '(let ((x (+ 5 6)) (y 10)) (+ x y)))
((lambda (x y) (+ x y)) (+ 5 6) 10)

(define (let->combination expr)
   (let ((bindings (cadr expr))
         (body     (cddr expr))
        )
        (let ((vars (map car bindings))
              (exps (map cadr bindings))
             )
             (cons (cons 'lambda (cons vars body)) exps)
        )
   )
)
```

Recall Nested lets: let*

Derived form: `let*`

```
(let* ((<var-1> <expr-1>)          ⇒   (let ((<var-1> <expr-1>))
       (<var-2> <expr-2>)                  (let ((<var-2> <expr-2>))
       ...                                    ...
       (<var-n> <expr-n>)                       (let ((<var-n> <expr-n>))
      )                                               <body>
      <body>                                    )
)                                          )
                                       )
```


and->if

Convert `(and <e1> <e2> <e3>)` to an `if` expression

Recall Lexical Scoping

Variables are bound when procedures are defined not when they are called. For example:

```
(define x 10)
(define (x-plus y) (+ x y))

(x-plus 2)                     => 12

(define (x-plus-double x) (x-plus (* 2 x)))

(x-plus-double 2)              => 14
```


Their Environments

The environment where `(* 2 x)` is evaluated is:

```
(((x 2))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 4.
The environment where `(+ x y)` is evaluated is:

```
(((y 4))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 14.

Dynamic Scoping

The *free variables* are bound dynamically at procedure call.
Local environment frame is added to calling procedure's environment.

```
(define x 10)
(define (x-plus y) (+ x y))

(x-plus 2)                    => 12

(define (x-plus-double x) (x-plus (* 2 x)))

(x-plus-double 2)             => 6
```

Environment frames in dynamic scoping correspond with call stack.

Their Environments

The environment where `(* 2 x)` is evaluated is:

```
(((x 2))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 4.
The environment where `(+ x y)` is evaluated is:

```
(((y 4))
 ((x 2))
 ((x 10)
  (+ #<primitive-procedure +>)
  (- #<primitive-procedure +>)
  ...
 ))
```

And we get 6.

Why Dynamic Scoping?

- It is a way to pass parameters.
- Unix environment variables use dynamic scoping.
- Exceptions handlers are dynamically scoped.

Scheme Programs

Want to execute your scheme code as a program? Write "shell" scripts in Scheme?

```
#!/usr/bin/guile -s
!#

;;; this shell script prints out its
;;; arguments in reverse order

(for-each            ;; like 'map', but ignores value.
   (lambda (x)
      (display x)
      (newline)
   )
   (reverse (command-line))
)
```

Where is Scheme?

- Education: used for introductory programming courses.
- Artificial Intelligence.
- Application Extension Language.
- Rapid Prototyping.

Scheme Summary

- Expression oriented.
- Uses lists extensively.
- Uses recursive extensively, has tail recursive optimization.
- Lexically scoped.
- Functional style (not pure).
- Interpreted, but compilers exist.
- Simple syntax but expressive.
- Dynamically typed (and type-safe).
- Garbage collected.