

Top Level Definitions

- top level definitions for variables are visible in every expression (unless shadowed by another definition).
- any variable referenced in a lambda (or let) expression that is not bound by any enclosing expression refer to a top level definition.

Adding Top-level definitions

- Procedure `define`

```
(define pi 3.141593)
```

```
(define double
```

```
  (lambda (x) (+ x x)))
```

```
(define grades '(90,80,22,65,88))
```

Top level definition and lexical scoping

- What is the difference between these 2 definitions?

```
(define incy (lambda (x) (+ x y)))
```

```
(define incy (let ((y 1))  
              (lambda (x) (+ x y))))
```

Top level procedures

```
(define cadr  
  (lambda (x) (car (cdr x))))
```

```
(define cddr  
  (lambda (x) (cdr (cdr x))))
```

What is:

```
(cadr '(a b c))
```

```
(cddr '(a b c))
```

Alternate syntax for top level procedure definitions

```
(define var0  
  (lambda (var1 var2 ...)  
    e1 e2 ...))
```

Can also be written:

```
(define (var0 var1 var2 ...)  
  e1 e2 ...)
```

Example: define a procedure that
doubles a number

```
(define (double x)  
  (+ x x))
```

-or-

```
(define double  
  (lambda (x) (+ x x)))
```

Better example: define a procedure that creates a new “doubling” procedure

```
(define (doubler f)
```

```
  (lambda (x) (f x x)))
```

```
(define double (doubler +))
```

```
(double 2) => 4
```

```
(define double-cons (doubler cons))
```

```
(double-cons 'a) => (a . a)
```

Conditional Expressions

(if test consequent alternative)

if *test* evaluates to true (*#t*), the value of *consequent* is returned, otherwise the value of *alternative* is returned.

if is not a procedure - it is a new syntactic form.

Why can't *if* be a procedure ?

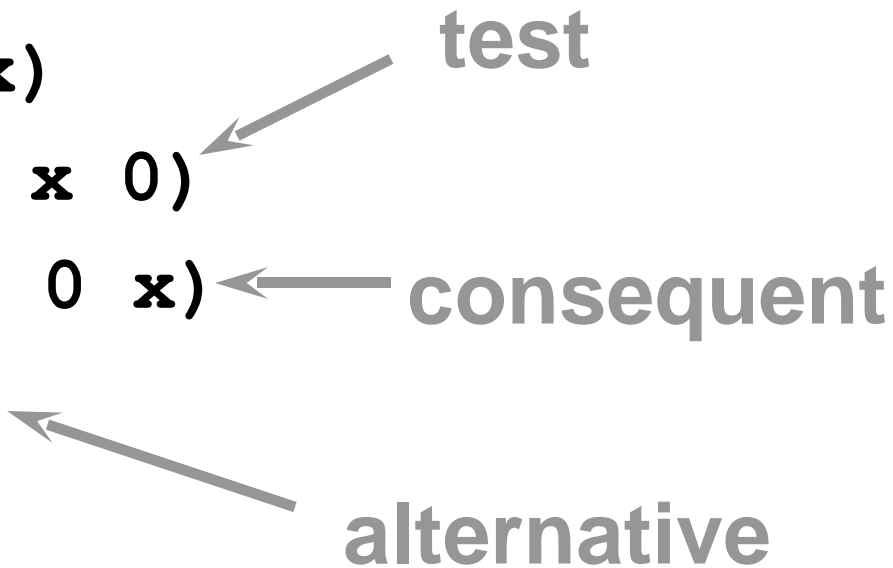
if example

```
(define abs  
  (lambda (x)  
    (if (< x 0)  
        (- 0 x)  
        x)  
  ))
```

test

consequent

alternative



Another way

```
(define abs  
  (lambda (x)  
    ((if (>= x 0)  
        +  
        -)  
     0 x)))
```

test

consequent

alternative

The **or** syntactic form

(or exp1 exp2 ...)

or evaluates expressions 1 at a time until it find one that is true. If no true expression is found **or** returns **#f**, otherwise **or** returns the value of the last expression evaluated.

(or) => #f

(or #f 'a #f) => 'a

The **and** syntactic form

(**and** *exp1* *exp2* . . .)

and evaluates expressions 1 at a time until it finds one that is false. If no false expression is found **and** returns **#t**. Otherwise **and** returns the value of the last expression evaluated.

The **not** procedure

(not exp)

not returns **#t** if **exp** evaluates to **#f** (and vice-versa).

not is a procedure (not a syntactic form).

Predicates

- a predicate answers a question about it's arguments and returns **#t** or **#f**.

= **<** **>** **<=** **>=**

null? returns **#t** if its argument is **()**.

eqv? returns **#t** if it's 2 arguments are equivalent (same number, symbol, boolean, string or pair). 2 pairs constructed by different calls to **cons** are not **eqv?**

Type predicates

- type predicates return true or false depending on the type of the argument.

`pair? symbol? number? string?`

```
(define safe-division
```

```
  (lambda (x y)
```

```
    (if (and
```


```
        (and (number? x) (number? y))
```

```
        (not (= y 0)))) (/ x y) (#f)))
```

The cond syntactic form

`(cond (t1 exp1a exp1b ...) (t2 exp2a exp2b ...) ... (else expa expb ...))`

clause



cond evaluates the test in each clause until it finds one that is true, and returns the result of evaluating the rest of the clause.

cond example

```
(define doubler
  (lambda (x)
    (cond
      ((number? x) (+ x x))
      ((string? x) (string-append x x))
      (else (cons x x))))))
```

(doubler 3) => 6

(doubler "hi dave") => "hi davehi dave"

Recursion

- A recursive procedure is a procedure that includes an application of itself.
- There must be a way to terminate the recursion - so there must be some condition that prevents application of the procedure within itself.

Typical Recursion

- Base case:
 - the condition(s) that result in a termination of recursion.
- Recursion step:
 - creates a return value in terms of the application of the procedure applied to a different argument.

Example - length of a list

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

base case

recursive step

Recursion Quiz

- Create a procedure named **list-copy** that returns a copy of a list.
- Create a procedure named **position** that returns the position of an element in a list.

For example:

```
(position '(a b c) 'b) => 1
```

```
(position '(a b c) 'a) => 0
```

Iteration

- Scheme doesn't need typical iteration constructs (*for* or *while* loops)!!!
- All iteration can be done with recursion.
- Some recursion (tail recursion) is effectively iteration...

mapping

- mapping is a specific type of iteration - the application of the same procedure to all elements in a list.

```
(define abs-all
  (lambda (x)
    (if (null? x)
        `()
        (cons (abs (car x))
              (abs-all (cdr x))))))
```

The **map** procedure

(map proc x)

The **map** procedure applies the procedure **proc** to each element of **x** and returns a list of the results.

(map abs '(3 -4 5 -6)) => (3 4 5 6)

More `map`

- `Map` can also be used with procedures that accept multiple arguments:

```
(map + '(2 4 6) '(1 3 5)) => (3 7 11)
```

- The lists must be of the same length and the procedure must accept as many arguments as there are lists.

Assignment

- **set!** can be used to assign a new value to a variable.
- **set!** works on let and lambda bound variables, and on top level variable definitions.
- Question: Is **set!** a syntactic form or a procedure ?

set! usage

- **set!** is not nearly as necessary as you might think.
- **set!** is commonly used to implement procedures that maintain state.

set! example

- Define a procedure named **count** that returns 0 the first time it is called, 1 the next time, 2 the next time, ...

```
(define count-val 0)
```

```
(define count
```

```
  (lambda ()
```

```
    (let ((x count-val))
```

```
      (set! count-val (+ 1 count-val))
```

```
      x) ) )
```

A better `count`

- `count` uses a “global” variable `count-val`, we can also create a procedure that uses a `let` bound variable.

```
(define count
  (let ((count-val 0))
    (lambda ()
      (let ((x count-val))
        (set! count-val (+ 1 count-val))
        x))))
```

Even better

- We can also create a procedure that creates a new counting procedure:

```
(define make-counter
  (lambda ()
    (let ((cval ))
      (lambda ()
        (let ((x cval))
          (set! cval (+ 1 cval))
          x) ))))
```

Stack Example

- Want to create a procedure that implements a stack data structure with the following operations:
 - push: add a new value on the top of the stack
 - pop: take the top value off the stack
 - empty?: boolean that indicates whether the stack is empty.

```
(define stack
  (let ((ls `()))
    (lambda (op . args)
      (cond
        ((eqv? op `empty?) (null? ls))
        ((eqv? op `push)
         (set! ls (cons (car args) ls)))
        ((eqv? op `pop)
         (let ((top (car ls)))
              (set! ls (cdr ls))
              top))
        (else "invalid operation")))))
```

Stack improvements

- Error checking - pop on an empty stack will not work since car of () is not defined.
- We can create a procedure make-stack that will create a new stack procedure. This allows us to use many stacks.
- Returning an error string is not good - perhaps we could print an error message.

I/O

- **load** reads and evaluates Scheme expressions from a file:

```
(load "mprog.scm")
```

- **read** reads an object (could be a scheme expression)

```
(let ((x (read))) (+ x 1))
```

- **read-char** reads a single character.

Output

- **write** will output a scheme object

```
(write `("hi" b c))  
("hi" b c)
```

- **display** will output strings and characters without quotes or #\ character formatting.

```
(display `("hi" b c))  
(hi b c)
```

I/O Example

```
(define ask_and_double
  (lambda ()
    (display "Enter a number\n")
    (let ((x (read)))
      (+ x x))))
```

```
(ask_and_double) Enter a number
```

```
10
```

```
=> 20
```

More I/O

- `write-char` outputs a single char
- `newline` outputs a ‘\n’
- `#\space` - character constant for ‘ ’
- `#\newline` - character constant for ‘\n’
- `transcript-on` recording of a session to a file.