## Expressions

Common structure for both procedures and data.
In Scheme, functions are called *procedures*.

When an expression is evaluated it creates a value or list of values that can be embedded into other expressions. Therefore programs can be written to manipulate other programs.

```
<expression> --> <variable>
     | <literal>
     | <procedure call>
     | <lambda expression>
     | <conditional>
     | <assignment>
     | <derived expression>
     | ...
```

See

`http://swiss.csail.mit.edu/~jaffer/r5rs_9.html#SEC72`

for the full syntax, if you're interested.



## Literals

Literals are *quoted* datum or anything that is *self-evaluating*, i.e., (quoted) booleans, numbers, characters, strings quoted lists, quoted vectors are all literals. E.g.,

`#t` evaluates to `#t` (true)

`()` evaluates to `()` (false)

`#f` evaluates to `()` (also false)

`5` evaluates to `5`

`'5` evaluates to `5`

`1/2` evaluates to `1/2`

`"Scheme Rocks"` evaluates to `"Scheme Rocks"`

`'(a b c d)` evaluates to `(a b c d)` (list)

`'(1 (2 3) 4)` evaluates to `(1 (2 3) 4)` (list)

Experiment with the Scheme interpreter!

More on lists soon....

## Procedure Application

The main form of a Scheme expression is the procedure application. (Terminology: in Scheme, the official name for what you would think of as a function is *procedure*.)

```
(procedure arg1 arg2 ... argn)
```

### Evaluation

- Each argument is evaluated.
- The procedure is applied to the results.

Exception: **syntactic forms**.

Syntactic forms violate the rule—they are built in to the language to handle cases the rule above can't handle. Examples: `define`, `if`, `cond`, `lambda`---more on this later.



### Examples

- `(- 1)` evaluates to -1
- `(* 5 7)` evaluates to 35
- `(+ 1 2 (* 2 3))` evaluates to 9
- `(+ (- 6 3) (/ 10 2) 2 (* 2 3))` evals to 16
- `(cos 0)` evaluates to 1

Exercise: run Scheme and try the arithmetic operators with 0, 1, 2 and 3 arguments, and figure out how the results make sense.

## Variables

Any identifier that is not a syntactic keyword is a variable.

To bind a name to a value:

```
(define var value)
```

```
1 ]=> (define a 2)
;Value: a

1 ]=> (define b 4)
;Value: b

1 ]=> (define c (+ a b))
;Value: c

1 ]=> c
;Value: 6

1 ]=> (define a 7)
;Value: a

1 ]=> c
;Value: 6
```

```
Hey...could define be a procedure?
```



## Built-In Procedures

- `eq?`: identity on atoms
- `null?`: is list empty?
- `car`: selects first element of list
- `cdr`: selects rest of list
- `(cons element list)`: constructs lists by adding `element` to front of `list`
- `quote` or `'`: produces constants

## Built-In Procedures

- ’() is the empty list

- (car ’(a b c)) =

- (car ’((a) b (c d))) =

- (cdr ’(a b c)) =

- (cdr ’((a) b (c d))) =



- car and cdr can break up any list:
  - (car (cdr (cdr ’((a) b (c d))))) =

  - (caddr ’((a) b (c d)))

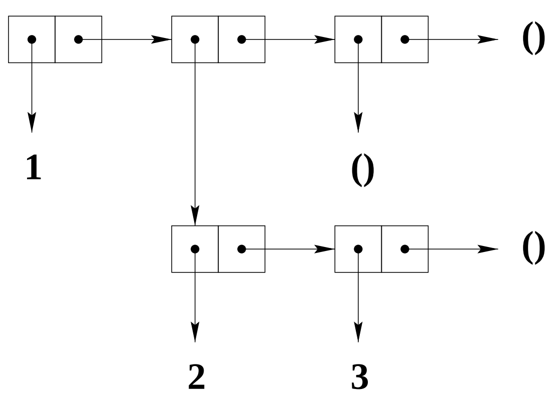
- cons can construct any list:
  - (cons ’a ’()) =
  - (cons ’d ’(e)) =
  - (cons ’(a b) ’(c d)) =
  - (cons ’(a b c) ‘((a) b)) =

# Lists

A simple but powerful general-purpose datatype. (How many datatypes have we seen so far?)

```
(1 #t 1)
()
(1 (2 3) ())
```

Building block: the cons cell.




Note: Sometimes you'll see NIL. This is LISP notation! In Scheme, we use `()`.



# Things you should know about cons, pairs and lists

The *pair* or *cons cell* is the most fundamental of Scheme's structured object types.

A **list** is a sequence of **pairs**; each pair's cdr is the next pair in the sequence.

The cdr of the last pair in a **proper list** is the empty list. Otherwise the sequence of pairs forms an **improper list**. I.e., an empty list is a proper list, and and any pair whose cdr is a proper list is a proper list.

An improper list is printed in **dotted-pair notation** with a period (dot) preceding the final element of the list. A pair whose cdr is not a list is often called a **dotted pair**

**`cons` vs. `list`:** The procedure `cons` actually builds *pairs*, and there is no reason that the cdr of a pair must be a list, as illustrated on the next page.

The procedure `list` is similar to `cons`, except that it takes an arbitrary number of arguments and always builds a proper list.

E.g., `(list 'a 'b 'c)` → `(a b c)`

# More about lists

A list in dotted-pair notation:

(a b c) $\rightarrow$ (a . (b . (c . ())))

```
1 ]=> (define foo '(a . (b . (c . ()))))
;Value: foo

1 ]=> (list? foo)
;Value: #t

1 ]=> (pair? foo)
;Value: #t
```

Proper lists:

(), (a (b (c) d) e)

(cons 'a '(b)) $\rightarrow$ (a b)

Dotted pairs (improper lists):

(cons 'a 'b) $\rightarrow$ (a . b)

(car '(a . b)) $\rightarrow$ a

(cdr '(a . b)) $\rightarrow$ b

(cons 'a '(b . c)) $\rightarrow$ (a b . c)



# Other (Predicate) Procedures

Predicate procedures return #t or () (i.e., false).

- = < > <= >= number comparison ops
- Run-time type checking procedures:
  - All return Boolean values: `#t` and ()
  - `(number? 5)` evaluates to `#t`
  - `(zero? 0)` evaluates to `#t`
  - `(symbol? 'sam)` evaluates to `#t`
  - `(list? '(a b))` evaluates to `#t`
  - `(pair? '(a b))` evalutates to `#t`
  - `(null? '())` evalutates to `#t`

## Other Predicate Procedures

A few more examples....

- `(number? 'sam)` evaluates to `()`
- `(null? '(a))` evaluates to `()`
- `(zero? (- 3 3))` evaluates to `#t`
- `(zero? '(- 3 3))` ⇒ type error
- `(list? (+ 3 4))` evaluates to `()`
- `(list? '(+ 3 4))` evaluates to `#t`
- `(pair? '(a . c))` evaluates to `#t`



## READ-EVAL-PRINT Loop

**READ:** Read input from user:
a procedure application

**EVAL:** Evaluate input:
(f $\texttt{arg}_1$ $\texttt{arg}_2$ ...$\texttt{arg}_n$)
1. evaluate `f` to obtain a procedure
2. evaluate each $\texttt{arg}_i$ to obtain a value
3. apply procedure to argument values

**PRINT:** Print resulting value:
the result of the procedure application

## READ-EVAL-PRINT Loop Example

```
1 ]=> (cons 'a (cons 'b '(c d)))
;Value 1: (a b c d)
```

1. Read the procedure application `(cons 'a (cons 'b '(c d)))`
2. Evaluate `cons` to obtain a procedure
3. Evaluate `'a` to obtain `a` itself
4. Evaluate `(cons 'b '(c d))`:
   (a) Evaluate `cons` to obtain a procedure
   (b) Evaluate `'b` to obtain `b` itself
   (c) Evaluate `'(c d)` to obtain `(c d)` itself
   (d) Apply the `cons` procedure to `b` and `(c d)` to obtain `(b c d)`
5. Apply the `cons` procedure to `a` and `(b c d)` to obtain `(a b c d)`
6. Print the result of the application: `(a b c d)`



## Quotes Inhibit Evaluation

```
;;Same as before:
1 ]=> (cons 'a (cons 'b '(c d)))
;Value 2: (a b c d)

;;Now quote the second argument:
1 ]=> (cons 'a '(cons 'b '(c d)))
;Value 3: (a cons (quote b) (quote (c d)))

;;Instead, un-quote the first argument:
1 ]=> (cons a (cons 'b '(c d)))
;Unbound variable:  a
;To continue, call RESTART...
2 error> ^C^C
1 ]=>
```

## Quotes vs. Eval

```
;;Some things evaluate to themselves:
1 ]=> (list 1 42 #t #f ())
;Value 4: (1 2 #t () ())

;;They can also be quoted:
1 ]=> (list '1 '42 '#t '#f '())
;Value 5: (1 2 #t () ())
```

Eval Activates Evaluation

```
1 ]=> '(+ 1 2)
;Value 6: (+ 1 2)

;;Eval can be used to evaluate an expression
1 ]=> (eval '(+ 1 2) '())
;Value 7: 3
```



## READ-EVAL-PRINT Loop

Can also be used to define procedures.

**READ:** Read input from user:
 a symbol definition

**EVAL:** Evaluate input:
 store function definition

**PRINT:** Print resulting value:
 the symbol defined

Example:

```
1 ]=> (define (square x) (* x x))

;Value: square
```

## Procedure Definition

Two syntaxes for definition:

1. (define (<fcn-name> <fcn-params>) <expression>)

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

(define (mean x y)
  (/ (+ x y) 2))
```

2. (define <fcn-name> <fcn-value>)

```
(define square
  (lambda (n) (* n n)))

(define mean
  (lambda (x y) (/ (+ x y) 2)))
```

Lambda procedure syntax enables the creation of anonymous procedures. More on this later!



## Conditional Execution: if

```
(if <condition> <result1> <result2>)
```

1. Evaluate <condition>
2. If the result is a “true value” (i.e., anything but () or #f), then evaluate and return <result1>
3. Otherwise, evaluate and return <result2>

```
(define (abs-val x)
  (if (>= x 0) x (- x)))

(define (rest-if-first e lst)
  (if (eq? e (car lst)) (cdr lst) '()))
```

## Conditional Execution: cond

```
(cond (<condition1> <result1>)
      (<condition2> <result2>)
      ...
      (<conditionN> <resultN>)
      (else <else-result>) ;optional else
 )                         ;clause
```

1. Evaluate conditions in order until obtaining one that returns a true value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return <else-result>



## Conditional Execution: cond

```
(define (abs-val x)
        (cond ((>= x 0) x)
              (else (- x))
         )
 )

(define (rest-if-first e lst)
        (cond ((null? lst) '())
              ((eq? e (car lst)) (cdr lst))
              (else '())
         )
 )
```

## Conditional vs. Boolean Expressions

Write a procedure that takes a parameter `x` and returns `#t` if `x` is an atom, and false otherwise. Using `cond`:

```
(define (atom? x)
  (cond ((symbol? x) '#t)
        ((number? x) '#t)
        ((char? x) '#t)
        ((string? x) '#t)
        ((null? x) '#t)
        (else ())
  )
)
```



## Conditional vs. Boolean Expressions

Now write `atom?` without using `cond`:

```
(define (atom? x)
  (if (symbol? x) '#t
    (if (number? x) '#t
      (if (char? x) '#t
        (if (string? x) '#t
          (if (null? x) '#t () )
        )
      )
    )
  )
)
```

# Better atom? procedure

Any list is a pair (dotted pair with CAR and CDR), except the empty list (which is both list and atom).

```
(define (atom? x)
  (if (pair? x) () '#t)
)

(define (atom? x)
  (cond ((pair? x) ())
        (else '#t)
  )
)
```