

---

## Functional Programming— Illustrated in Scheme

---

### References:

- Dybvig, (available online and in the library)
- Sebesta Chapter 15.1-15.6, 15.9, 15.10.

Lisp slides © D. Horton 200. Scheme slides © S. Stevenson, D. Inkpen 2001. Adapted for Scheme © E. Joanis 2000, 2002. Modified, updated and extended © S. McIlraith 2004, 2005, 2007. Additional slides use material taken from © G. Baumgartner 2001.

---

## Scheme on CDF

---

Invoking: `scheme`

Exiting: `(exit)` or `Ctrl-D`

Loading filename.scm: `(load ‘‘filename’’)`  
or  
`(load ‘‘filename.scm’’)`

Tracing: `(trace proc_name)`

Transcript:  
`(transcript-on <my_trans>)`  
`(transcript-off)`  
saves a transcript of a session to `<my_trans>`.

Debugger:  
-start: `(debug)`  
-help: `?`  
-go back (read-eval-print level): `(restart 1)`  
or  
`Ctrl-C Ctrl-C`  
-quit: `q`

---

## Jumping right in

---

### A Scheme procedure

```
(define increment
  (lambda (n)
    (+ n 1)
  )
)
```

or

```
(define (increment n)
  (+ n 1)
)
```

### A call to the procedure

```
(increment 21)
```

---

## The Spirit of Lisp-like Languages

---

We shall first define a class of **symbolic expressions** in terms of ordered pairs and lists. Then we shall define five elementary **functions and predicates**, and build from them by **composition, conditional expressions** and **recursive definitions** an extensive class of functions of which we shall give a number of examples. We shall then show how these **functions can themselves be expressed as symbolic expressions**, and we shall give a **universal function** *apply* that allows us to compute from the expressions for a given function its value for given arguments. Finally, we shall define some **functions with functions as arguments** and give some useful examples.

McCarthy, J, [1960]. Recursive functions of symbolic expressions and their computation by machine, Part I. *Comm. ACM* 3:4; quoted in Sethi.

---

## Pure Functional Languages

---

Fundamental concept: **application** of (mathematical) **functions** to **values**

1. **Referential transparency:** The value of a function application is independent of the context  $t$  in which it occurs (i.e., given the same parameters, it always returns the same results). Or alternatively, a language is referentially transparent if we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program. This is achieved by not having side effects in programs, e.g.,
  - value of  $f(a,b,c)$  depends only on the values of  $f$ ,  $a$ ,  $b$  and  $c$
  - It does not depend on the global state of computation $\Rightarrow$  all vars in function must be parameters

Main advantage: facilitates reasoning about programs and applying program transformations.

See [http://en.wikipedia.org/wiki/Referential\\_transparency](http://en.wikipedia.org/wiki/Referential_transparency)

---

## Pure Functional Languages (cont.)

---

2. The concept of assignment is **not** part of functional programming
  - no explicit assignment statements
  - variables bound to values only through the association of actual parameters to formal parameters in function calls
  - function calls have no side effects
  - thus no need to consider global state
3. Control flow is governed by function calls and conditional expressions
  - $\Rightarrow$  no iteration
  - $\Rightarrow$  recursion is widely used

---

## Pure Functional Languages (cont.)

---

4. All storage management is implicit
  - needs garbage collection
5. Functions are *First Class Values*
  - Can be returned as the value of an expression
  - Can be passed as an argument
  - Can be put in a data structure as a value
- Unnamed functions exist as values

---

## A Functional Program

---

A program includes:

1. A set of function definitions
2. An expression to be evaluated

E.g. in Scheme:

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

```
;Value: abs-val
```

```
1 ]=> (abs-val (- 3 5))
```

```
;Value: 2
```

---

## Jumping Back In

---

### The MIT Scheme Interface

```
werewolf 1% scheme
Scheme Microcode Version ...
```

```
1 ]=> (+ 8 3 5 16 9)
;Value: 41
```

```
1 ]=> (define increment (lambda (n) (+ n 1)))
;Value: increment
```

```
1 ]=> (increment 21)
;Value: 22
```

```
1 ]=> (load "incr")
;Loading "incr.scm" -- done
;Value: increment-list
```

```
1 ]=> (increment-list (1 32 7))
;The object 1 is not applicable.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify a procedure to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.
```

```
2 error> (restart 1)
;Abort!
```

```
1 ]=> (increment-list '(1 32 7))
;Value 1: (2 33 8)
```

9

```
1 ]=> (trace increment-list)
;Unspecified return value
```

```
1 ]=> (increment-list '(1 32 7))
```

```
[Entering #[compound-procedure 2 increment-list]
  Args: (1 32 7)]
```

```
[Entering #[compound-procedure 2 increment-list]
  Args: (32 7)]
```

```
[Entering #[compound-procedure 2 increment-list]
  Args: (7)]
```

```
[Entering #[compound-procedure 2 increment-list]
  Args: ()]
```

```
[()]
```

```
<== #[compound-procedure 2 increment-list]
  Args: ()]
```

```
[(8)
```

```
<== #[compound-procedure 2 increment-list]
  Args: (7)]
```

```
[(33 8)
```

```
<== #[compound-procedure 2 increment-list]
  Args: (32 7)]
```

```
[(2 33 8)
```

```
<== #[compound-procedure 2 increment-list]
  Args: (1 32 7)]
```

```
;Value 3: (2 33 8)
```

```
1 ]=> (exit)
```

```
Kill Scheme (y or n)? Yes
Happy Happy Joy Joy.
werewolf 2%
```

10

---

## Formal Roots: $\lambda$ -Calculus

---

- Defined by Alonzo Church, a logician, in 1930s as a computational theory of recursive functions
- $\lambda$ -calculus is equivalent in computational power to Turing machines
- Recall: what's a Turing machine?  
Turing machines are abstract machines that emphasize computation as a series of state transitions driven by symbols on an input tape (which leads naturally to an imperative style of programming based on assignment)
- How is  $\lambda$ -calculus different?
  - $\lambda$ -calculus emphasizes typed expressions and functions (which naturally leads to a functional style of programming).
  - No state transitions.

---

## $\lambda$ -Calculus (cont.)

---

$\lambda$ -calculus is a formal system for defining recursive functions and their properties.

- Expressions are called  $\lambda$ -expressions.
- Every  $\lambda$ -expression denotes a function.
- A  $\lambda$ -expression consists of 3 kinds of terms:
  - Variables:**  $x, y, z$  etc  
 $V$  denotes arbitrary variables
  - Abstractions:**  $\lambda V.E$   
where  $V$  is some variable and  $E$  is another  $\lambda$ -term.
  - Applications:**  $(E1 E2)$  where  $E1$  and  $E2$  are  $\lambda$ -terms. Applications are sometimes called combinations.

---

## $\lambda$ -Calculus (cont.)

---

Formal Syntax in BNF

```
< $\lambda$ -term> ::= <variable>
            |  $\lambda$ <variable> . < $\lambda$ -term>
            | (< $\lambda$ -term> < $\lambda$ -term>)
```

```
<variable> ::= x | y | z | ...
```

Or more compactly

```
E ::= V |  $\lambda$ V.E | (E1 E2)
V ::= x | y | z | ...
```

Where  $V$  is an arbitrary variable and  $E$  is an arbitrary  $\lambda$ -expression. We call  $\lambda V$  the **head** of the  $\lambda$ -expressions and  $E$  the **body**.

---

## $\lambda$ -Calculus: Functional Forms

---

A higher-order function (functional form):

- Takes functions as parameters
- Yields a function as a result

E.g.: Given

$$f(x) = x + 2, \quad g(x) = 3 * x$$

then,

$$h(x) = f(g(x)) \text{ and} \\ h(x) = (3 * x) + 2$$

$h(x)$  is called a **higher-order function**.

**Types of Functional Forms:**

Construction form: E.g.,

$$g(x) = x * x, \quad h(x) = 2 * x, \quad i(x) = x / 2 \\ [g,h,i] (4) = (16,8,2)$$

Apply-to-all form: E.g.,

$$h(x) = x * x \\ y(h, (2,3,4)) = (4,9,16)$$

---

## $\lambda$ -Calculus Is it really Turing Complete?

---

Can we represent the class of Turing computable functions?

Yes, we can represent:

- Boolean and conditional functions
- Numerical and arithmetic functions
- Data structures: ordered pairs, lists, etc.
- Recursion

But, doing so in  $\lambda$ -calculus is tedious;

- Need syntactic sugar to simplify task,
- $\lambda$ -calculus more suitable as an abstract model of a programming language rather than a practical programming language.

*Both Turing machines and  $\lambda$ -calculus are idealized, mathematical models of computation.*

---

## Scheme: A Functional Programming Language

---

1958: Lisp

1975: Scheme (revised over the years)

1980: Common Lisp ("CL")

1980s: Lisp Machines (e.g, Symbolics, TI Explorer, etc.)

Lisp, Scheme and CL contrasted on following pages.

Some features of Scheme:

- denotational semantics based on the  $\lambda$ -calculus.  
i.e., the meaning of programming constructs in the language is defined in terms of mathematical functions.

- lexical scoping

i.e., all free variables in a  $\lambda$ -expression are assigned values at the time that the  $\lambda$  is defined (i.e., evaluated and returned).

- arbitrary ctrl structures w/ *continuations*.

- functions as first-class values

- automatic garbage collection.



---

## LISP

---

- Functional language developed by John McCarthy in 1958.
- Semantics based on  $\lambda$ -Calculus
- All functions operate on lists or atomic symbols: (called "S-expressions")
- Only five basic functions: list functions `cons`, `car`, `cdr`, `equal`, `atom` and one conditional construct: `cond`
- Uses dynamic scoping
- Useful for list-processing applications
- Programs and data have the same syntactic form: S-expressions
- Used in Artificial Intelligence

---

## SCHEME

---

- Developed in 1975 by G. Sussman and G. Steele
- A version of LISP
- Consistent syntax, small language
- Closer to initial semantics of LISP
- Provides basic list processing tools
- Allows functions to be first class objects
- Provides support for *lazy evaluation*
- lexical scoping of variables

---

## COMMON LISP (CL)

---

- Implementations of LISP did not completely adhere to semantics
- Semantics redefined to match implementations
- COMMON LISP has become the standard
- Committee-designed language (1980s) to unify LISP variants
- Many defined functions
- Simple syntax, large language

---

## 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](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*.)

(*p*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)) =`

---

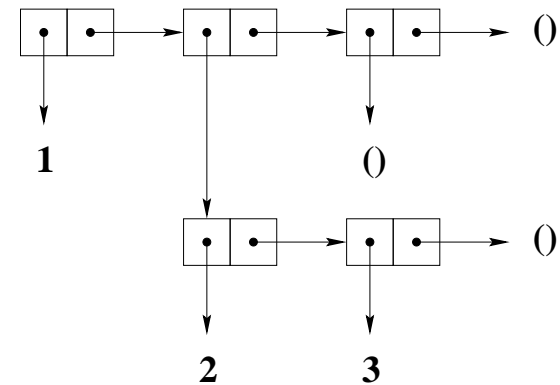
## Lists

---

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 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) → (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)) → (a b)`

Dotted pairs (improper lists):

`(cons 'a 'b) → (a . b)`

`(car '(a . b)) → a`

`(cdr '(a . b)) → b`

`(cons 'a '(b . c)) → (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))` evaluates to `#t`
  - `(null? '())` evaluates 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 arg<sub>1</sub> arg<sub>2</sub> ... arg<sub>n</sub>)  
1. evaluate f to obtain a procedure  
2. evaluate each arg<sub>i</sub> 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)
  )
)
```