Functional Programming— Illustrated in Scheme

References:

- Dybvig, (available online and in the library)
- Sebesta 6th ed., chapter 15,

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 contex t in which it occurs
   - 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

   ⇒ all vars in function must be parameters

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

   ⇒ no iteration

   ⇒ 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)
```

```
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%
```


Formal Roots: λ-Calculus

- Defined by Alonzo Church, a logician, in 1930s as a computational theory of recursive functions
- λ-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 λ-calculus different?
  - λ-calculus emphasizes typed expressions and functions (which naturally leads to a functional style of programming).
  - No state transitions.

λ-Calculus (cont.)

λ-calculus is a formal system for defining recursive functions and their properties.

- Expressions are called λ-expressions.
- Every λ-expression denotes a function.
- A λ-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 λ-term.
  Applications: $(E1\ E2)$ where $E1$ and $E2$ are λ-terms. Applications are sometimes called combinations.

λ-Calculus (cont.)

Formal Syntax in BNF

```
<λ-term> ::=  <variable>
           |  λ<variable> . <λ-term>
           |  (<λ-term> <λ-term>)


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

Or more compactly

```
E ::=  V   |  λV.E   |  (E1   E2)
V ::=  x   |  y   |  z   |  ...
```

Where `V` is an arbitrary variable and `E` is an arbitrary λ-expression. We call λ`V` the **head** of the λ-expressions and `E` the **body**.

λ-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 ,   g(x) = 3 * x

then,

    h(x) = f(g(x)) 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, h(x) = 2 * x, 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)

λ-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 λ-calculus is tedious;

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

Both Turing machines and λ-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 λ-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 λ-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 *λ-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

Commonalities between LISP and SCHEME

- Expressions are written in prefix, parenthesized form
  - (function $\texttt{arg}_1$ $\texttt{arg}_2$ ...$\texttt{arg}_n$)
  - `(+ 4 5)`
  - `(+ (* 3 4 5) (- 5 3))`
- In order to evaluate an expression:
  1. evaluate `function` to a function value
  2. evaluate each $\texttt{arg}_i$ in order to obtain its value
  3. apply the function value to these values

S-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>
     | <macro use>
     | <macro block>
    #t (true)
    () (false)
    (a b c)
    (a (b c) d)
    ((a b c) (d e (f)))
    (1 (b) 2)
    (+ '1 2)
```

Lists have nested structure.

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))`` =

More about lists

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)`

`(a b c)` → `(a . (b . (c . ())))`

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 previous 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)`

Other (Predicate) Procedures

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

- `+ - * /` numeric operators, e.g.,
  `(+ 5 3)` = 8, `(- 5 3)` = 2
  `(* 5 3)` = 15, `(/ 5 3)` = 1.6666666
- `= < > <= >=` number comparison ops
- Run-time type checking procedures:
  - All return Boolean values: `#t` and `()`
  - `(number? 5)` is `#t`
  - `(zero? 0)` is `#t`
  - `(symbol? 'sam)` is `#t`
  - `(list? '(a b))` is `#t`
  - `(null? '())` is `#t`

Other Predicate Procedures

- `(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`

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)
  )
)
```


Recursion: Five Steps to a Recursive Function

1. **Strategy:** How to reduce the problem?
2. **Header:**
   - What info needed as input and output?
   - Write the function header. Use a noun phrase for the function name.
3. **Spec:** Write a method specification in terms of the parameters and return value. Include preconditions.
4. **Base Cases:**
   - When is the answer so simple that we know it without recursing?
   - What is the answer in these base case(s)?
   - Write code for the base case(s).
5. **Recursive Cases:**
   - Describe the answer in the other case(s) in terms of the answer on smaller inputs.
   - Simplify if possible.
   - Write code for the recursive case(s).

Recursive Scheme Procedures: Sum-N

Parameter: integer n ≥ 0.

Result: sum of integers from 0 to n.

```
(define (sum-n n)

   (cond (                                   )


         (else                               )

    )
 )
```


Recursive Scheme Procedures: Length

```
(define (length x)






   ))
```

This is called "cdr-recursion."

Note: There is a built-in `length` procedure.

Length (cont.)

```
1 ]=> (trace length)

;No value

1 ]=> (length '(a b c))

[Entering #[compound-procedure 5 length]
    Args: (a b c)]
[Entering #[compound-procedure 5 length]
    Args: (b c)]
[Entering #[compound-procedure 5 length]
    Args: (c)]
[Entering #[compound-procedure 5 length]
    Args: ()]
[0
      <== #[compound-procedure 5 length]
    Args: ()]
[1
      <== #[compound-procedure 5 length]
    Args: (c)]
[2
      <== #[compound-procedure 5 length]
    Args: (b c)]
[3
      <== #[compound-procedure 5 length]
    Args: (a b c)]
;Value: 3
```


Recursive Scheme Procedures: Abs-List

- `(abs-list '(1 -2 -3 4 0))` ⇒ `(1 2 3 4 0)`
- `(abs-list '())` ⇒ `()`

```
(define (abs-list lst)





)
```


Recursive Scheme Procedures: Append

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

(define (append x y)




)
```

Note: There is a built-in `append` procedure.