Functional Programming Languages
(LISP/ Scheme)

# Traditional programming languages (i.e. Imperative languages)

# Based on the Von Neumann Architecture.

# Make the most efficient language, given the Von Neumann hardware architecture.

# Functional programming languages

# Based on mathematical functions.

# Mathematical Functions:

# Maps members of the domain set into a range set.

# Evaluation order of expressions in a function is controlled by recursion and conditional expressions, instead of sequencing and iterative repetition (used in imperative languages)

# Simple Functions:

\[
\text{cube}(X) \equiv X \times X \times X, \text{ where } X \text{ is a real number.}
\]

(the / symbol is read "is defined as")

(the domain and range sets are the set of real numbers)

1.0 --------> 1.0
2.0 --------> 8.0
3.0 --------> 27.0
**# Lambda Expression** (by Alanzo Church 1941)

# Method for defining **nameless** functions.

# Separating the task of defining a function from that of naming a function.

# Lambda expression specifies the parameter and the mapping of a function. (the value of the lambda expression is the function itself.)

# \(8(x) X\times X\times X\)

# Application of Lambda expression:

# \((8(x) X\times X\times X)\ (2) \Rightarrow 8\)

# In LISP:

\(( (\text{lambda} \ (x) \ (\times \ X \ X)) \ 2 )\) will evaluate as 8

\((\text{lambda} \ (x) \ (\times \ X \ X))\) is the nameless function

2 is the parameter
Functional Forms

A higher-order function is a function that either takes a function as parameters or yields a function as its result or both.

# Examples of functional forms: (many exists)

1) Function Composition
2) Construction
3) Apply-to-all
4) ....
1) Function Composition

# Has two functional parameters and yield a function whose value is the first actual parameter function applied to the result of the second.

# Composition is syntactically denoted as "" "

\[ h \circ f \circ g \]

for example if:

\[ f(x) = x + 2 \]
\[ g(x) = 3 \times x \]

then h is defined as:

\[ h(x) = f(g(x)), \text{ or } h(x) = (3 \times x) + 2 \]
2) Construction

# Takes a list of functions as parameters. When a construction is applied to an argument, it applies each of its functional parameters to that argument and collects the results in a list or sequence.

# A construction is syntactically denoted as:

\[ [f, g] \]  \hspace{1cm} (a list of functions in \textbf{brackets})

# Example:

Given the following functions:

\[ g(x) / x \times x \]
\[ h(x) / 2 \times x \]
\[ i(x) / x / 2 \]

the Construction:

\[ [g, h, i] (4) \]

Yields:

\[ (16, 8, 2) \]
3) Apply-to-all

# Takes a single function as a parameter. When applied to a list of arguments, it will apply its functional parameter to each of the values in the list of arguments and collects the results in a list or sequence.

# Apply-to-all is syntactically denoted by "".

# Example:
Given the following function:

\[ h(x) = x^2 \]

the Construction:

"" (h, (2, 3, 4))

Yields:

(4, 9, 16)
Functional programming languages

# Used to mimic mathematical functions.

# A purely functional language does not use variables or assignment statements.

# A program is written as a number of function definitions and function applications.

# A functional language provides the following:

1) A set of primitive functions (car, cdr, cons, etc.)

2) A set of functional forms to form complex functions from primitive ones. (define, defun, etc.)

3) A function application operation. (eval)

4) Some structure or structures for storing data. (Define,setq, etc.)
LISP/Scheme

# Developed by John McCarthy (1956-1959)

# Traditionally an interpreted and interactive language however, compilers are available.

# Began as purely functional language (pure lisp). Provided list processing operations based on Lambda Calculus.

# Later added imperative features to increase its execution efficiency and completeness

- Variables
- Iterations & loops
- Assignment
- Property list
- Efficient numerical computation
- Free variables (global variables)
- Conditional branches (cond)
- Unconditional branches (go)

# For many years, it was the most popular non-imperative language for AI applications. (in U.S., Prolog for Japan)

# Excellent for expressing recursive algorithms which manipulate dynamic data structures.

# Different Implementations of LISP/Scheme

  # MacLisp (MIT)
  # InterLisp
  # Franz LISP (U.C. Berkeley)
  # Lisp Machines (Direct execution concept)
    - Texas Instrument's Explorer system
  # Common Lisp
# Scheme (MIT, IU, etc.)
# Newlisp (INTERDATA DEVELOPMENT GROUP, MENLO PARK, CA)
# No ANSI standards for LISP yet!

# Major applications of LISP

# Applications that deal with data in the form of symbols and structures of symbolic expressions.

# List processing

# Theorem proving

# Game playing

# Natural Language Understanding/Processing.

# Computer Vision

# Robotics

# Knowledge representation

# Expert Systems.
Writing LISP programs

# Interpreted and interactive

# Fully parenthesized expressions (parens. should match)

# Operators appear in PREFIX form:
   (+ 2 3)

# Data structures or data objects can be either:
   1) Atoms
   2) Lists (expressions)

# Comments in LISP begin with a ";;" (semicolon)

# Free variables: are generally undeclared and have global scope.
   Ex: (define Greeting '(Hi there)) ;defines Greeting to be a free variable.

# Bound variables: are local to a function and have dynamic scope.
   Ex: local variables defined within a function or arguments passed to a function.

# Loops in LISP/Scheme are often implemented as recursion rather than iteration.

# Elementary Data Types:

1- Numbers (integer and real)

2- Symbols (any string of characters which does not represent a decimal number)
# Note:

Variables in LISP/Scheme do not have the same notion of TYPE as they do in imperative languages.
## Atoms

- Can be either a number or a symbol.
- The basic building blocks for all data structures in LISP/Scheme.
- Predefined Atoms:
  
  - `t` = The logical value **TRUE**.
  - `nil/null` = The logical value **False** or the empty list `()`. Depending on the context.

## S-expressions

- Both **atoms** and **lists** are often called **S-expressions**. The "S" stands for Symbolic.

- S-expressions are the data objects manipulated by LISP.

---

### Relationship between various types of objects in LISP:

- **S-expression**
  - **Atom**
    - **Number**
      - Fixed Point
      - Floating Point
    - **Symbol**
      - Symbolic atoms are atoms that are not numbers.
# Primitive Functions

Predefined functions which can be used to build higher level functions.

- `abs` ; (abs -23) ==> 23
- `min` ; (min 3 44 2 3) ==> 2
- `max`
- `car` ; return the first element of the list
- `cdr` ; return rest of the list (excluding the first elem.)
- `quote , (')` ; to inhibit evaluation
- `eq, equal, eq?` ; scheme uses eq? (eq? 3 3) ==> #t
  ; (eq? 3 4) ==> () ; an empty list.
- `and`
- `or`
- `defun, define` ; assign an expression to a variable
- `setq` ; assignment a S-expr to a variable (Franz lisp)
- `list` ; list constructor
cons ; list constructor
cond ; case
print, display ; scheme uses display
terpri ; terminate print (new line)
append ; append two lists
not ; negation
reverse ; reverse a list
null
atom
read
eval ; evaluate an expression

# Note:
The above primitives are not reserved words therefore, the user may redefine them!!

# General form of a LISP Function:

(f-name arg1 arg2 arg3 - - - - - - - - - -)

# f-name: Identifies the function
# arg: Arguments to which the function must be applied.

# Example:
(+ 2 3) ==> 5
(list 2 3) ==> (2 3)
(cons '2 (3)) ==> (2 3)
(+ (* 2 3) 4) ==> 10
(car '(a b c)) ==> a
(cdr '(a b c)) ==> (b c)

# LISP Variables have the following properties: [pg. 322 Tucker86]

# Name (any symbol)

# Value (atom or list)
# Function names are also considered as variable names!!

# Are not declared

# Are allocated during execution of the program. (dynamically)

# Have dynamic types. (the type of the variable can vary during the execution)
# To Enter Lisp or Scheme:

$ lisp ; Lucid common lisp  
$ flisp ; Franz Lisp (UCB version)  
$ scheme ; IU or MIT versions

# To load a lisp program:

(load "prog1.l") ; in Common lisp  
(load 'prog1.l) ; in Franz lisp  
(load "prog1.l") ; in Scheme

# To Quit Lisp/Scheme:

(quit) ; in Common lisp  
(exit) or ctrl-D ; in Franz lisp  
(exit) ; in Scheme

# To run lisp or scheme interactively:

  # Enter the LISP or Scheme environment  
  # Enter LISP/Scheme statements
Sample Code

; C-311 Concepts of programming Languages
; Introduction to Lisp/Scheme

% scheme

> 'a
a
> '(a b c)
(a b c)
> ()
()

; Arithmetic Functions

> 37
37
> (+ 5 8)
13
> (+ 5 8 9)
22
> (+ 5 8 9 5)
27
> (- 5 9)
-4
> (* (+ 5 4) (- 3 9))
-54
; List processing functions

> (car '(a b c))
a
> (cdr '(a b c)) ; always returns a list
(b c)

> (cdr '(a)) ; always returns a list
()
> (car '((a b c d) e (f g) h))
(a b c d)
Lists or Pairs:
- A pair (sometimes called a dotted pair) is a data structure with two fields called the car and cdr fields.
- Pairs are created by the procedure cons.
- The car and cdr fields are accessed by the procedures car and cdr.
- The car and cdr fields are assigned by the procedures set-car! and set-cdr!.
- A list can be defined recursively as either the empty list or a pair whose cdr is a list.

; Constructing a list or Pair (one of the items must be a list)

> (cons 'a '(b c d)) ; construct a list
(a b c d)

> (cons '() '(a b c))
() a b c

> (cons '(a b) '(c))
((a b) c)

> (cons '(a b) 'c)
((a b) . c)

; list

> (list 'a 'b 'c)
(a b c)

> (list 'a (+ 3 4) 'c)
(a 7 c)
>; Append
- Returns a list consisting of the elements of the first list followed by
  the elements of the other lists.

- The resulting list is always newly allocated, except that it shares
  structure with the last list argument. The last argument may actually
  be any object; an improper list results if the last argument is not a
  proper list.

; Last item has to be a lists

(append '(a b c) '(d e f g h))
(a b c d e f g h)

> (append '(a b) '(c (dee) f gh))
(a b c (dee) f gh)

(append '(x) '(y)) => (x y)
(append '(a) '(b c d)) => (a b c d)
(append '(a (b)) '(c))) => (a (b) (c))
(append) => ()
; using variables & define

> (define A '(a b c)) ; Scheme
> A
(a b c)

> (car A)
a

> (cadr A) ; (car (cdr A))
b
Define & Set!

- Syntax:
  (define variable expression)

  > (define x 'a)

  > (define A '(a b c))

  > (define B '(1 2 3))
  > B
  (1 2 3)

  > (define C (list A B)) ; combine the two lists
  > C
  ((a b c) (1 2 3))

- Define has essentially the same effect as this assignment expression, if variable is bound:

  (set! variable expression)

  (set! y 'a) ; Gives you an error.
  (define y 'a) ; bind the variable to a location and sets the value to 'a
  (set! y 'b) ; now you can use set!

- If variable is not bound, however, define binds variable to a new location in the current environment before performing the assignment

- It is an error to perform a set! on an unbound variable

(define add3
  (lambda (x) (+ x 3))))
(add3 3) => 6

; Predicates (functions that return either TRUE or FALSE)

; atom and atom?

> (atom 'a) ; Franz lisp
Error: variable atom is not bound.
Type (debug) to enter the debugger.

> (atom? 'a) ; Scheme (no longer supported!!!)
#t

; list?
> (list? '(a b c)) ; Scheme
#t

; null and null?
> (null? 'a)
#f Or empty list ()
> (null? ())
#t
> (null? '())
#t

; eq and eq?
> (eq? 3 4)
#f or ()

> (eq? 'a 'b)
#f or ()
> (eq? 'a 'a)
#t
>
- Arguments of the "eq?" must be either atoms or the same list for the eq? To return true.

> (eq? '(a b c) '(a b c)) ; will return false
#f

> (eq? '(a) '(a))
#f or ()

> (define x '(a b c))
> (eq? x x) ; will return true
#t
Numbers, Symbols and Strings

; number?
(number? 5)
#t
> (define x 5)
> (number? x)
#t

; symbol?
> (symbol? x)
#f or ()

> (symbol? 'a)
#t

> (symbol? '(a b))
#f

; string?
> (define s "this is a test")

> s
"this is a test"

> (string? s)
#t

; string-length and string-append
> (string-length s)
14
> (define ss (string-append s " of appending strings"))
> ss
"this is a test of appending strings"

; lambda is used to create new procedures

> ((lambda (n) (+ n 2)) 4) ; 4 is passed as an argument
6

; define is used to assign an expression to a variable
> (define add2 ; define = defun in Franz lisp
   (lambda (n)
      (+ n 2)
   )
)

> (add2 9)
11
> (add2 6)
8

> (define f
   (lambda (x y)
      (+ x y)
   )
)

> (f 7 8)
15

; exit the scheme interpreter

> (exit)
FACTORIAL

Math. Notation:

\[
\begin{align*}
  f(n) &= 1, \quad \text{if } n = 0 \\
  &\quad ; \\
  &\quad < n \cdot f(n-1), \quad \text{if } n > 0
\end{align*}
\]

LISP:

```lisp
(define (factorial (lambda (n)
    (cond ; take one or more pairs of s-expr. in which the
    ; first is a predicate (returns T/F)
    ((eq? n 0) 1)
    (t (* n (factorial (- n 1)))) ; default case
    )))
```
This file contains a number of sample LISP/SHCEME functions. You may load and execute any of these function by performing the following.

To load this file in to SCHEME type:

1) Enter SCHEME
   $ scheme
2) Load the file
   > (load "file.name")

To run the following function type:

(heading)

Print heading

(define heading
  (lambda ()
    (display " C311")
    (newline)
    (display " Concepts of Programming languages")
    (newline)
  )
)
(define interaction
  (lambda ()
    (display "what is your name? ")
    (define name (read-string char-set:whitespace)) ; read until whitespace
    (newline)
    (display "Nice to meet you ")
    (display name)
    (newline)
  )
)

(our_member 'a '(b c a d))

(define our_member
  (lambda (element the_list)
    (cond
      ((null? the_list) 'No_Sir)               ; if the list is empty
      ((eq? element (car the_list)) 'Yes_Sir)  ; found it return true
      (else (our_member element (cdr the_list))) ; default case
    )
  )
)
To run the following function type:

(my_reverse '(a b c d e f g (h i)))

(define my_reverse
  (lambda (alist)
    (display "List : ")
    (display alist)
    (newline)
    (cond
      ((null? alist) alist) ;if the list is empty
      ((atom? alist) alist) ;if it is an atom
      ((null? (cdr alist)) alist) ;if it is an atom
      (else (append (my_reverse (cdr alist)) (list (car alist)) ))
    ))
  )
)

To run the following function type:

(f_to_c 25)

; convert Fahrenheit to Celcius

(define f_to_c
  (lambda (fahrenheit)
    (/ (- fahrenheit 32) 1.8)
  ))
(define f_2_c
   (lambda (fahrenheit)
     (define celcius (/ (- fahrenheit 32) 1.8))
     (display "Fahrenheit = ")
     (display fahrenheit)
     (newline)
     (display "Celcius = ")
     (display celcius)
     (newline)
   ))

(define exchange
   (lambda (pair)
     (list (cadr pair) (car pair)))
   )
(define factorial
  (lambda (number)
    (cond
      ((zero? number) 1) ; if the number = zero return 1
      (else (* number (factorial (- number 1)))))))
(define atom? (lambda (alist_or_atom)
    (cond
      ((null? alist_or_atom) '#f)       ; if the list is empty do nothing
      ((not (list? alist_or_atom))   #t)  
      (else '#f)
    )
  )
)

(define list_atoms (lambda (alist)
    (cond
      ((null? alist)  )       ; if the list is empty do nothing
      ((atom? alist)     (display alist) (newline))  
      ((not (list? alist)) (display alist) (newline))
      (else
        (list_atoms (car alist)) (list_atoms (cdr alist))
      )
    )
  )
)
(define sum
  (lambda (alist)
    (cond
      ((null? alist) 0 )
      ((atom? alist) (alist) )
      (else
        (+ (car alist) (sum (cdr alist)) )
      ))))

(define count
  (lambda (alist)
    (cond
      ((null? alist) 0 )
      ((atom? alist) 1 )
      (else
        (add1 (count (cdr alist) ) )
      ))))
; To run the following function type:
; -----------------------------------
; (Add1 '()) will return a zero
; (add1 '(non empty list or atom)) will return value sent +1
;
;
(define add1
  (lambda (alist)
    (cond
      ((null? alist) 0 )
      (else (+ alist 1) )
    )
  )
)