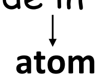
Functional programming

- LISP: John McCarthy 1958 MIT
  - List Processing => Symbolic Manipulation
- First functional programming language
  - Every version after the first has imperative features, but we will discuss the functional subset

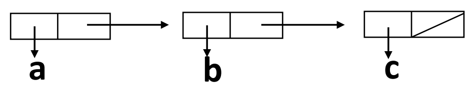
LISP Data Types

- There are only two types of data objects in the original LISP
  - Atoms: symbols, numbers, strings,…
    - E.g., a, 100, "foo"
  - Lists: specified by delimitating elements within parentheses
    - Simple lists: elements are only atoms
      - E.g., (A B C D)
    - Nested lists: elements can be lists
      - E.g., (A (B C) D (E (F G)))

LISP Data Types

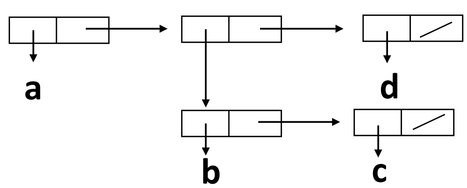
- Internally, lists are stored as **single-linked list** structures
  - Each node has two pointers: one to element, the other to next node in the list
  - Single atom: 

  - List of atoms: (a b c)



LISP Data Types

- List containing list (a (b c) d)



Scheme

- Scheme is a dialect of LISP, emerged from MIT in 1975
- Characteristics
  - simple syntax and semantics
  - small size
  - exclusive use of static scoping
  - treating functions as first-class entities
    - As first-class entities, Scheme functions can be the values of expressions, elements of lists, assigned to variables, and passed as parameters

Interpreter

- Most Scheme implementations employ an interpreter that runs a "read-eval-print" loop
  - The interpreter repeatedly reads an expression from a standard input, evaluates the expression, and prints the resulting value

Primitive Numeric Functions

- Primitive functions for the basic arithmetic operations: +, -, *, /
  - + and * can have zero or more parameters. If * is given no parameter, it returns 1; if + is given no parameter, it returns 0
  - - and / can have two or more parameters
  - Prefix notation

Expression	Value
42	42
(* 3 6)	18
(+ 1 2 3)	6
(sqrt 16)	4

Numeric Predicate Functions

- Predicate functions return Boolean values (#T or #F): =, <>, >, <, >=, <=, EVEN?, ODD?, ZERO?

Expression	Value
(= 16 16)	#T
(even? 29)	#F
(> 10 (* 2 4))	
(zero? (-10(* 2 5)))	

Type Checking

- Dynamic type checking
- Type predicate functions

  (boolean? x) ; Is x a Boolean?

  (char? x)

  (string? x)

  (symbol? x)

  (number? x)

  (pair? x)

  (list? x)

Lambda Expression

- E.g., lambda(x) (* x x) is a nameless function that returns the square of its given numeric parameter
- Such functions can be applied in the same ways as named functions
  - E.g., ((lambda(x) (* x x)) 7) = 49
- It allows us to pass function definitions as parameters

"define"

- To bind a name to the value of a variable:
  (define symbol expression)
  - E.g., (define pi 3.14159)
  - E.g., (define two_pi (* 2 pi))
- To bind a function name to an expression:
  (define (function_name parameters)
  (expression)
)
  - E.g., (define (square x) (* x x))

"define"

- To bind a function name to a lambda expression
  (define function_name
  (lambda_expression)
)
  - E.g., (define square (lambda (x) (* x x)))

Control Flow

- Simple conditional expressions can be written using if:
  - E.g. (if (< 2 3) 4 5) => 4
  - E.g., (if #f 2 3) => 3

Control Flow

- It is modeled based on the evaluation control used in mathematical functions:

```
(COND
    (predicate_1 expression)
    (predicate_2 expression)
    …
    (predicate_n expression)
    [ELSE expression]
)
```

An Example

$$f(x) = \begin{cases} 1 \; if \;\; x = 0 \\ x * f(x-1) \; if \;\; x > 0 \end{cases}$$

```
( define ( factorial x )
      ( cond
            (( < x 0 ) #f)
            (( = x 0 ) 1)
            ( #t (* x  (factorial (- x 1)))) ; or else (…)
      )
)
```