FP Foundations, Scheme (2)

In Text: Chapter 15

N. Meng, F. Poursardar

Functional programming

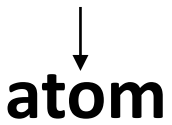
- LISP: John McCarthy 1958 MIT
  - List Processing => Symbolic Manipulation
- First functional programming language
  - Every version after the first has imperative features, but we will discuss the functional subset

LISP Data Types

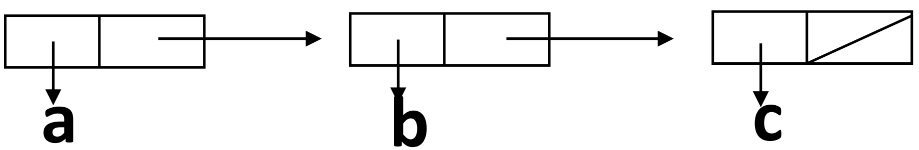
- There are only two types of data objects in the original LISP
  - Atoms: symbols, numbers, strings,…
    - E.g., a, 100, "foo"
  - Lists: specified by delimitating elements within parentheses
    - Simple lists: elements are only atoms
      - E.g., (A B C D)
    - Nested lists: elements can be lists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

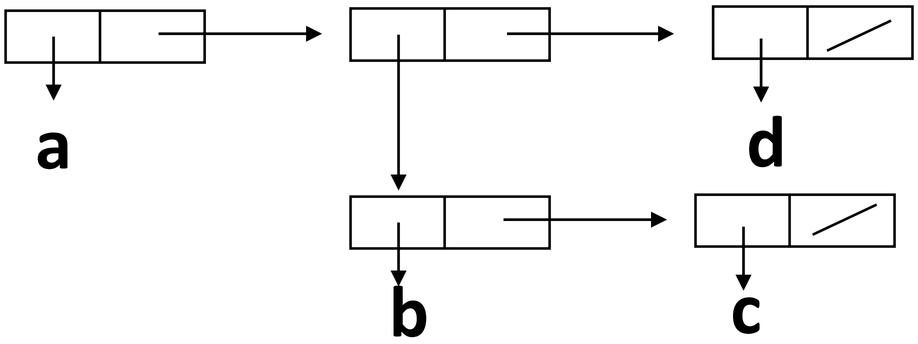


  - List of atoms: (a b c)



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

Lambda Expression

- Lambda expressions can have any number of parameters.
- E.g.,

```
(LAMBDA (a b c x) (+ (* a x x) (* b x) c))
```

“define”

- Scheme special form function
- 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)))

Another Example

Factorial function using "define":

```
(define (factorial n)
   (if (<= n 1)
    1
    (* n (factorial (− n 1)))
))
```

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

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 (cont'd)

- 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 (…)
        )
)
```

Bindings & Scopes

- `let` is a function that **creates a local scope** in which names are temporarily bound to the values of expressions
- Names can be bound to values by introducing a nested scope

- `let` takes two or more arguments:
  - The first argument is a list of pairs
    - In each pair, the first element is the name, while the second is the value/expression
  - Remaining arguments are evaluated in order
  - The value of the construct as a whole is the value of the final argument
  - E.g. (let ((a 3)) a)

`let` Examples

- computes the roots of a given quadratic equation, ax2 + bx + c:
  root1 = (-b + sqrt(b2 - 4ac))/2a and
  root2 = (-b - sqrt(b2 - 4ac))/2a

```
(define (quadratic_roots a b c)
   (let    (
      (root_part_over_2a
                    (/ (SQRT (− (* b b) (* 4 a c))) (* 2 a)))
      (minus_b_over_2a (/ (− 0 b) (* 2 a)))
             )
   (LIST (+ minus_b_over_2a root_part_over_2a)
                    (− minus_b_over_2a root_part_over_2a))
))
```

`let` Examples

- E.g., (let ((a 3)

```
              (b 4)
              (square (lambda (x) (* x x)))
              (plus +))
          (sqrt (plus (square a) (square b))))
```

- The scope of the bindings produced by `let` is its second and following arguments

`let` Examples

- E.g., (let ((a 3))
```
(let ((a 4)
      (b a))
  (+ a b))) => ?
```

- b takes the value of the outer a, because the defined names are visible "all at once" at the end of the declaration list

`let*` Example

- `let*` makes sure that names become available "one at a time"
- E.g., (let*((x 1) (y (+ x 1)))
  (+ x y)) => ?

Functions

- quote: identity function
  - When the function is given a parameter, it simply returns the parameter
  - E.g., (quote A) => A
    (quote (A B C)) => (A B C)
- The common abbreviation of quote is apostrophe (‘)
  - E.g., ‘ a => a
    ‘ (A B C) => (A B C)

List Functions

- car: returns the first element of a given list
  - E.g., (car ‘(A B C)) => A
    (car ‘((A B) C D)) => (A B)
    (car ‘A) => ?
    (car ‘(A)) => ?
    (car ‘()) => ?

List Functions

- cdr: returns the remainder of a given list after its car has been removed
  - E.g., (cdr '(A B C)) => (B C)
    (cdr '((A B) C D)) => (C D)
    (cdr 'A) => ?
    (cdr '(A)) => ?
    (cdr '()) => ?

List Functions

- cons: concatenates an element with a list
- cons builds a list from its two arguments
  - The first can be either an atom or a list
  - The second is usually a list
  - E.g., (cons ‘A ‘()) => (A)
    (cons ‘A ‘(B C)) => (A B C)
    (cons ‘() ‘(A B)) => ?
    (cons ‘(A B) ‘(C D)) => ?
  - How to compose a list (A B C) from A, B, and C?

List Functions

- Note that cons can take two atoms as parameters, and return a dotted pair
  - E.g., (cons ‘A ‘B) => (A . B)
  - The dotted pair indicates that this cell contains two atoms, instead of

    an atom + a pointer

    or

    a pointer + a pointer

More Predicate Functions

- The following returns #t if the symbolic atom is of the indicated type, and #f otherwise
  - E.g., (symbol? ‘a) => #t
    (symbol? ‘()) => #f
  - E.g., (number? ‘55) => #t
    (number? 55) => #t
    (number? ‘(a)) => #f
  - E.g., (list? ‘(a)) => #t
  - E.g., (null? ‘()) => #t

More Predicate Functions

- eq? returns true if two objects are equal through pointer comparison
  - Guaranteed to work on symbols
  - E.g., (eq? ‘A ‘A) => #T
    (eq? ‘A ‘(A B)) => #F
- equal? recursively compares two objects to determine if they are equal
  - The objects can be atoms or lists

How do we implement equal?

```
(define (atom? atm)
  (cond
    ((list? atm) (null? atm))
    (else #T)
  )
)
```

```
(define (equal? lis1 lis2)
  (cond
    ((atom? lis1) (eq? lis1 lis2))
    ((atom? lis2) #F)
    ((equal? (car lis1) (car lis2))
       ((equal? (cdr lis1) (cdr lis2))
    (else #F)
  )
)
```

More Examples

```
(define (member? atm lis)
  (cond
    ((null? lis) #F)
    ((eq? atm (car lis)) #T)
    (else (member? atm (cdr lis)))
  )
)
```

What is returned for the following function?
(member? 'b '(a (b c)))

```
(define (append lis1 lis2)
  (cond
    ((null? lis1) lis2)
    (else (cons (car lis1)
                (append(cdr lis1) lis2)))
  )
)
```

Is lis2 appended to lis1, or lis1 prepended to lis2?

An example: apply-to-all function

```
(define (mapcar fctn lis)
    (cond
        ((null? lis) '())
        (else (cons (fctn (car lis))
                         (mapcar fctn (cdr lis)) ))
)
```