

# CSC 533: Programming Languages

Spring 2012

## Functional programming

- LISP & Scheme
- S-expressions: atoms, lists
- functional expressions, evaluation, define
- primitive functions: arithmetic, predicate, symbolic, equality, higher-order
- special forms: if, cond
- recursion: tail vs. full

1

## Functional programming

imperative languages are modeled on the von Neumann architecture

in mid 50's, AI researchers (Newell, Shaw & Simon) noted that could define a language closer to human reasoning

- symbolic
- (dynamic) list-oriented
- transparent memory management

in late 50's, McCarthy developed LISP (List Processing Language)

- instantly popular as the language for AI
- separation from the underlying architecture tended to make it less efficient (and usually interpreted)

2

## LISP

### LISP is very simple and orthogonal

- only 2 kinds of data objects
  1. atoms (identifiers, strings, numbers, ...)
  2. lists (of atoms and sublists)
    - unlike arrays, lists do not have to store items of same type/size*
    - do not have to be stored contiguously*
    - do not have to provide random access*
- all computation is performed by applying functions to arguments
  - in pure LISP: no variables, no assignments, no iteration*
- functions and function calls are also represented as lists
  - no distinction between program and data*

3

## Scheme

### Scheme was developed at MIT in the mid 70's

- clean, simple subset of LISP
- static scoping
- first-class functions
- efficient tail-recursion

function calls appear as lists: (FUNC ARG1 ARG2 ... ARGn)

```
> (+ 3 2)
5
> (+ 3 (* 2 5))
13
> (car '(foo bar biz baz))
foo
> (cdr '(foo bar biz baz))
(bar biz baz)
```

quote symbol denotes data  
- not evaluated by the interpreter  
- numbers are implicitly quoted

car : returns first element of list  
cdr : returns rest of list

4

## Scheme functions

Scheme functions are also defined as lists (MORE DETAILS LATER)

```
(define (FUNC ARG1 ARG2 . . . ARGn)
  RETURN_EXPRESSION)
```

```
(define (square x)
  (* x x))
```

```
> (square 3)
```

```
9
```

```
> (square 1.5)
```

```
2.25
```

```
(define (last arblist)
  (car (reverse arblist)))
```

```
> (last '(a b c))
```

```
c
```

```
> (last '(foo))
```

```
foo
```

5

## Obtaining a Scheme interpreter

many free Scheme interpreters/environments exist

- Dr. Scheme is an development environment developed at Rice University
- contains an integrated editor, syntax checker, debugger, interpreter
- Windows, Mac, and UNIX versions exist
- can download a personal copy from  
<http://download.plt-scheme.org/drscheme/>
- ignore *Racket* propaganda, select older 4.2.4 version (4.2.5 crashes for me)
- be sure to set Language to "Pretty Big"

PLAY TIME! Details to follow...

6

## S-expressions

in LISP/Scheme, data & programs are all of the same form:  
S-expressions (short for Symbolic-expressions)

- an S-expression is either an atom or a list

### atoms

- numbers 4 3.14 1/2 #xA2 #b1001
- characters #\a #\Q #\space #\tab
- strings "foo" "Dave Reed" "@%!?#"
- Booleans #t #f
- symbols Dave num123 miles->km !\_^\_!

*symbols are sequences of letters, digits, and "extended alphabetic characters"*

*+ - . \* / < > = ! ? : \$ % + & ~ ^*

*can't start with a digit, case insensitive by default (but can set preferences in Dr. Scheme)*

7

## S-expressions (cont.)

### lists

() is a list

(L1 L2 . . . Ln) is a list, where each  $L_i$  is either an atom or a list

for example:

```
() (a)
(a b c d) ((a b) c (d e))
((((a))))
```

note the recursive definition of a list – GET USED TO IT!

also, get used to parentheses (LISP = Lots of Inane, Silly Parentheses)

8

## Functional expressions

computation in a functional language is via function calls

```
(FUNC ARG1 ARG2 . . . ARGn)
```

*note: functional expressions are S-expressions*

```
(car '(a b c))          (+ 3 (* 4 2))
```

evaluating a functional expression:

- function/operator name & arguments are evaluated in unspecified order  
*note: if argument is a functional expression, evaluate recursively*
- the resulting function is applied to the resulting values

```
(car '(a b c))
```

evaluates to list (a b c) : ' terminates recursive evaluation

evaluates to primitive function

so, primitive car function is called with argument (a b c)

9

## Arithmetic primitives

predefined functions: + - \* /  
quotient remainder modulo  
max min abs gcd lcm  
floor ceiling truncate round  
= < > <= >=

- many of these take a variable number of inputs

```
(+ 3 6 8 4)           → 21  
(max 3 6 8 4)        → 8  
(= 1 (-3 2) (* 1 1)) → #t  
(< 1 2 3 4)          → #t
```

- functions that return a true/false value are called *predicate functions*  
zero? positive? negative? odd? even?

```
(odd? 5)              → #t  
(positive? (- 4 5))  → #f
```

10

## Data types in LISP/Scheme

similar to JavaScript, LISP/Scheme is dynamically typed

- types are associated with values rather than variables, bound dynamically

numbers can be described as a hierarchy of types



integers and rationals are *exact* values, others can be *inexact*

- arithmetic operators preserve exactness, can explicitly convert

```
(+ 3 1/2)      → 7/2
(+ 3 0.5)     → 3.5

(inexact->exact 4.5) → 9/2
(exact->inexact 9/2) → 4.5
```

11

## Symbolic primitives

predefined functions: car cdr cons  
list list-ref length member  
reverse append

```
(list 'a 'b 'c)      → (a b c)
(list-ref '(a b c) 1) → b
(member 'b '(a b c)) → (b c)
(member 'd '(a b c)) → #f
```

- car and cdr can be combined for brevity

```
(cadr '(a b c)) = (car (cdr '(a b c))) → b
```

```
cadr returns 2nd item in list
caddr returns 3rd item in list
caddr returns 4th item in list (can only go 4 levels deep)
```

12

## Equality primitives

`equal?` compares 2 inputs, returns `#t` if equivalent, else `#f`

```
(equal? 'a 'a)           → #t
(equal? '(a b) '(a b))  → #t
(equal? (cons 'a '(b)) '(a b)) → #t
```

other (more restrictive) equivalence functions exist

`eq?` compares 2 symbols (efficient, simply compares pointers)  
`eqv?` compares 2 atomics (symbols, numbers, chars, strings, bools)  
-- less efficient, strings & numbers can't be compared in constant time

```
(eq? 'a 'a) → #t           (eqv? 'a 'a) → #t
(eq? '(a b) '(a b)) → #f   (eqv? '(a b) '(a b)) → #f
(eq? 2 2) → unspecified    (eqv? 2 2) → #t
```

`equal?` uses `eqv?`, applied recursively to lists

13

## Defining functions

can define a new function using `define`

- a function is a mapping from some number of inputs to a single output

```
(define (NAME IN1 IN2 ... INn)
  OUTPUT_VALUE)
```

```
(define (square x)      (define (next-to-last arblast)
  (* x x))              (cadr (reverse arblast)))
```

basically, parameter passing is by-value since each argument is evaluated before calling the function – but no copying (instead, structure sharing)

14

## Conditional evaluation

can select alternative expressions to evaluate

```
(if TEST TRUE_EXPRESSION FALSE_EXPRESSION)

(define (my-abs num)
  (if (negative? num)
      (- 0 num)
      num))

(define (singleton? arblast)
  (if (and (list? arblast) (= (length arblast) 1))
      #t
      #f))
```

and, or, not are standard boolean connectives  
evaluated from left-to-right, short-circuit evaluation

15

## Conditional evaluation (cont.)

predicates exist for selecting various types

```
symbol?   char?   boolean?  string?  list?   null?
number?   complex? real?    rational? integer?
exact?    inexact?
```

note: an if-expression is a *special form*

- is *not* considered a functional expression, doesn't follow standard evaluation rules

```
(if (list? x)
    (car x)
    (list x))
```

test expression is evaluated

- if value is anything but #f, first expression is evaluated & returned
- if value is #f, second expression is evaluated & returned

- anything but #f is considered "true"

```
(if (member 'foo '(biz foo foo bar)) 'yes 'no)
```

16



## Multi-way conditional

when there are more than two alternatives, can

- nest if-expressions (i.e., cascading if's)
- use the `cond` special form (i.e., a switch)

```
(cond (TEST1 EXPRESSION1)
      (TEST2 EXPRESSION2)
      . . .
      (else EXPRESSIONn))
```

evaluate tests in order

- when reach one that evaluates to "true", evaluate corresponding expression & return

```
(define (compare num1 num2)
  (cond ((= num1 num2) 'equal)
        (> num1 num2) 'greater
        (else 'less)))

(define (my-member item lst)
  (cond ((null? lst) #f)
        (equal? item (car lst)) lst
        (else (my-member item (cdr lst)))))
```

17

## Repetition via recursion

pure LISP/Scheme does not have loops

- repetition is performed via recursive functions

```
(define (sum-1-to-N N)
  (if (< N 1)
      0
      (+ N (sum-1-to-N (- N 1)))))

(define (my-member item lst)
  (cond ((null? lst) #f)
        (equal? item (car lst)) lst
        (else (my-member item (cdr lst)))))

(define (my-length lst)
  IN-CLASS EXERCISE
  )
```

18

## Tail-recursion vs. full-recursion

a tail-recursive function is one in which the recursive call occurs last

```
(define (my-member item lst)
  (cond ((null? lst) #f)
        ((equal? item (car lst)) lst)
        (else (my-member item (cdr lst)))))
```

a full-recursive function is one in which further evaluation is required

```
(define (sum-1-to-N N)
  (if (< N 1)
      0
      (+ N (sum-1-to-N (- N 1)))))
```

each full-recursive call requires a new activation record on the run-time stack with tail-recursion, don't need to retain current activation record when make call

- can discard the current activation record, push record for new recursive call
- thus, no limit on recursion depth (each recursive call reuses the same memory)
- Scheme interpreters are required to perform this tail-recursion optimization

19

## Tail-recursion vs. full-recursion (cont.)

any full-recursive function can be rewritten using tail-recursion

- often accomplished using a help function with an accumulator

```
(define (factorial N)
  (if (zero? N)
      1
      (* N (factorial (- N 1)))))
```

value is computed "on the way up"

```
(factorial 2)
  ↑
(* 2 (factorial 1))
  ↑
(* 1 (factorial 0))
  ↑
1
```

```
(define (factorial N)
  (factorial-help N 1))

(define (factorial-help N value-so-far)
  (if (zero? N)
      value-so-far
      (factorial-help (- N 1) (* N value-so-far))))
```

value is computed "on the way down"

```
(factorial-help 2 1)
  ↓
(factorial-help 1 (* 2 1))
  ↓
(factorial-help 0 (* 1 2))
  ↓
2
```

20

## Scoping in Scheme

unlike early LISPs, Scheme is statically scoped

- can nest functions and hide details

```
(define (factorial N)

  (define (factorial-help N value-so-far)
    (if (zero? N)
        value-so-far
        (factorial-help (- N 1) (* N value-so-far))))

  (factorial-help N 1))
```

- since factorial-help is defined inside of factorial, hidden to outside
- since statically scoped, arguments in enclosing function are visible to enclosed functions (i.e., non-local variables)

21

## When tail-recursion?

```
(define (sum-1-to-N N)

  (define (sum-help N sum-so-far)
    (if (< N 1)
        sum-so-far
        (sum-help (- N 1) (+ N sum-so-far))))

  (sum-help N 0))
```

```
(define (my-length lst)
```

*IN-CLASS EXERCISE*

```
(length-help lst 0))
```

usually, a full-recursive solution is simpler, more natural

- for a small number of repetitions, full-recursion is sufficient
- for a potentially large number of repetitions, need tail-recursion

22

## Higher-order primitives

`(apply FUNCTION LIST)` applies the function with the list elements as inputs

```
(apply + '(1 2 3)) ≡ (+ 1 2 3) → 6  
(apply min '(5 2 8 6)) ≡ (min 5 2 8 6) → 2
```

`(map FUNCTION LIST)` applies the function to each list element

```
(map sqrt '(9 25 81)) ≡ (list (sqrt 9) (sqrt 25) (sqrt 81)) → (3 5 9)  
(map car '((a b) (c d) (e))) ≡  
  (list (car '(a b)) (car '(c d)) (car '(e))) → (a c e)
```