# CSC 550: Introduction to Artificial Intelligence

## Spring 2004

### Scheme programming

- S-expressions: atoms, lists
- functional expressions, evaluation
- primitive functions: arithmetic, predicate, symbolic, equality, high-level
- defining functions: define
- special forms: if, cond
- recursion: tail vs. full
- let expressions, I/O

---

# Functional programming

### 1957: FORTRAN was first high-level programming language
- mathematical in nature, efficient due to connection with low-level machine
- not well suited to AI research, which dealt with symbols & dynamic knowledge

### 1959: McCarthy at MIT developed LISP (List Processing Language)
- symbolic, list-oriented, transparent memory management
- instantly popular as the language for AI
- separation from the underlying architecture tended to make it less efficient (and usually interpreted)

### 1975: Scheme was developed at MIT
- clean, simple subset of LISP
- static scoping, first-class functions, efficient tail-recursion, …

# 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/`

  be sure to set Language to "Textual (MzScheme, includes R5RS)"

---

# LISP/Scheme

LISP/Scheme is very simple
- only 2 kinds of data objects

  1. atoms (identifiers/constants)   `robot    green    12.5`
  2. lists (of atoms and sublists)   `(1 2 3.14)`
     `(robot (color green) (weight 100))`

  *Note: lists can store different types, not contiguous, not random access*

- functions and function calls are represented as lists (i.e., program = data)

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

- all computation is performed by applying functions to arguments, also as lists

  | | |
  |---|---|
  | `(+ 2 3)` | **evaluates to** 5 |
  | `(square 5)` | **evaluates to** 25 |
  | `(car (reverse '(a b c)))` | **evaluates to** c |

# S-expressions

in LISP/Scheme, data & programs are all of the same form:
S-expressions (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*

---

# S-expressions (cont.)

## Lists

`()`                    is a list
`(L1 L2 . . . Ln)`  is a list, where each `Li` 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)

# Functional expressions

computation in a functional language is via function calls (also S-exprs)

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

(+ 3 (* 4 2))

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

> quote specifies data, not to be evaluated further
> (numbers are implicitly quoted)

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)

---

# Arithmetic primitives

predefined functions:
```
+  -  *  /
quotient  remainder  modulo
max  min  abs  gcd  lcm  expt
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
```

# Data types in LISP/Scheme

## LISP/Scheme is loosely typed

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

## numbers can be described as a hierarchy of types

```
number
complex
real            ↑       MORE GENERAL
rational
integer
```

## 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
```

---

# Symbolic primitives

## predefined functions:
```
car  cdr  cons
list  list-ref  length  member
reverse  append  equal?
```

```
(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

(equal? 'a (car '(a b c))   →   #t
```

- `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
`cadddr` returns 4th item in list   (can only go 4 levels deep)

# 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 INPUTS) OUTPUT_VALUE)
```

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

(define (next-to-last arblist)
   (cadr (reverse arblist)))

(define (add-at-end1 item arblist)
   (reverse (cons item (reverse arblist))))

(define (add-at-end2 item arblist)
   (append arblist (list item)))
```

```
(square 5) ➔ 25

(next-to-last '(a b c d))
          ➔ c

(add-at-end1 'x '(a b c))
          ➔ '(a b c x)

(add-at-end2 'x '(a b c))
          ➔ '(a b c x)
```

---

# Examples

```
(define (miles->feet mi)
  IN-CLASS EXERCISE
  )
```

```
(miles->feet 1) ➔ 5280

(miles->feet 1.5) ➔ 7920.0
```

```
(define (replace-front new-item old-list)
  IN-CLASS EXERCISE
  )
```

```
(replace-front 'x '(a b c))
 ➔ (x b c)

(replace-front 12 '(foo))
 ➔ (12)
```

# 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 (wind-chill temp wind)
  (if (<= wind 3)
      (exact->inexact temp)
      (+ 35.74 (* 0.6215 temp)
              (* (- (* 0.4275 temp) 35.75) (expt wind 0.16)))))
```

---

# Conditional evaluation (cont.)

## logical connectives `and, or, not` can be used

## 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 expr evaluated & returned
- if value is #f, second expr evaluated & returned

```
(if (and (list? x) (= (length x) 1))
    'singleton
    'not)
```

Boolean expressions are evaluated left-to-right, short-circuited

# 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 (wind-chill temp wind)
  (cond ((> temp 50) 'UNDEFINED)
        ((<= wind 3) (exact->inexact temp))
        (else (+ 35.74 (* 0.6215 temp)
                 (* (- (* 0.4275 temp) 35.75)
                    (expt wind 0.16))))))
```

---

# Examples

```
(define (palindrome? lst)
  IN-CLASS EXERCISE
  )
```

```
(palindrome? '(a b b a))
  ➔ #t

(palindrome? '(a b c a))
  ➔ #f
```

```
(define (safe-replace-front new-item old-list)
  IN-CLASS EXERCISE
  )
```

```
(safe-replace-front 'x '(a b c))
  ➔ (x b c)

(safe-replace-front 'x '())
  ➔ 'ERROR
```

# 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)))))
```

# Examples

```
(define (sum-list numlist)
  IN-CLASS EXERCISE
  )
```

```
(sum-list '())  ➔ 0

(sum-list '(10 4 19 8))  ➔ 41
```

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

```
(my-length '())  ➔ 0

(my-length '(10 4 19 8))  ➔ 4
```

# 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)))))
```

full-recursive call requires memory proportional to number of calls
     → limit to recursion depth
tail-recursive function can reuse same memory for each recursive call
     → no limit on recursion

---

# 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
- since Scheme is statically scoped, can hide help function by nesting

```
(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)

  (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)))
```

value is computed "on the way down"

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

# Finally, variables!

Scheme does provide for variables and destructive assignments

➤ `(define x 4)`            `define` creates and initializes a variable

➤ `x`
`4`

➤ `(set! x (+ x 1))`      `set!` updates a variable

➤ `x`
`5`

since Scheme is statically scoped, can have global variables
- destructive assignments destroy the functional model
- for efficiency, Scheme utilizes structure sharing – messed up by set!

---

# Let expression

fortunately, Scheme provides a "clean" mechanism for creating variables to store (immutable) values

```
(let ((VAR1 VALUE1)
      (VAR2 VALUE2)
       . . .
      (VARn VALUEn))
  EXPRESSION)
```

*let expression* introduces a new environment with variables (i.e., a block)

good for naming a value (don't need `set!`)

same effect could be obtained via help function

game of craps:
- if first roll is 7, then WINNER
- if first roll is 2 or 12, then LOSER
- if neither, then first roll is "point"
  – keep rolling until get 7 (LOSER) or point (WINNER)

```
(define (craps)

  (define (roll-until point)
    (let ((next-roll (+ (random 6) (random 6) 2)))
      (cond ((= next-roll 7) 'LOSER)
            ((= next-roll point) 'WINNER)
            (else (roll-until point)))))

  (let ((roll (+ (random 6) (random 6) 2)))
    (cond ((or (= roll 2) (= roll 12)) 'LOSER)
          ((= roll 7) 'WINNER)
          (else (roll-until roll)))))
```

# Scheme I/O

to see the results of the rolls, could append rolls in a list and return

or, bite the bullet and use non-functional features
- `display` displays S-expr (`newline` yields carriage return)
- `read` reads S-expr from input
- `begin` provides sequencing (for side effects), evaluates to last value

```
(define (craps)

  (define (roll-until point)
    (let ((next-roll (+ (random 6) (random 6) 2)))
      (begin (display "Roll: ")(display next-roll) (newline)
             (cond ((= next-roll 7) 'LOSER)
                   ((= next-roll point) 'WINNER)
                   (else (roll-until point))))))

  (let ((roll (+ (random 6) (random 6) 2)))
    (begin (display "Point: ") (display roll) (newline)
           (cond ((or (= roll 2) (= roll 12)) 'LOSER)
                 ((= roll 7) 'WINNER)
                 (else (roll-until roll))))))
```

# Next week…

AI programming and logic
- classic AI programs in Scheme
- logic, predicate calculus
- automated deduction

Read Chapter 2

Be prepared for a quiz on
- this week's lecture (moderately thorough)
- the reading (superficial)