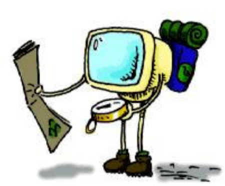
Programming Languages

Tevfik Koşar

Lecture - XIII
March 2nd, 2006

Roadmap

- Functional Languages
  - Lambda Calculus
- Intro to Scheme
  - Basics
  - Functions
  - Bindings
  - Equality Testing
  - Searching



2

Functional Languages

- Functional languages make heavy use of subroutines (more than Van Neumann languages or any other language class)
- Examples:
  - Scheme
  - Lisp
  - Miranda
  - Haskell
  - Sisal
  - pH
  - ML

3

Lambda Calculus

- Lambda Calculus: inspiration for functional programming
  - Based on parameterized expressions (each parameter introduced by the occurrence of the letter λ)
  - Used to compute by substituting parameters into expressions
  - Output of a program is defined as a mathematical function of the inputs, with no notion of internal state and, and thus no side effects
    - Unless explicit use of assignment (set! function)

4

Functional Programming Concepts

Features of functional programming languages which are often missing in other languages;

- first class function values: values that can be passed as parameters to subroutines or returned from a subroutine.
- high-order functions: functions which take other functions as arguments or return a function as a result.
- extensive polymorphism: allow a function to be used on a class of arguments
- list types and operators:
- recursion
- structured function returns
- constructors for structures objects
- garbage collection

5

Intro to Scheme

- Scheme uses Cambridge Polish notation for expressions (pre-order).
  - Eg. (+ 3 4)
  - First argument inside the left parenthesis is the function, the remaining are its arguments

- Scheme interpreter runs a read-eval-print loop:

```
>> (+3 4)
7
>> 8
8
>> ((+ 3 4))
Error!
```

6

Intro to Scheme

- Load function:

>> (load “my_scheme_program”)

- Quote: (instead of evaluating)

>> (quote (+ 3 4))
(+ 3 4)

>> ‘(+ 3 4)
(+ 3 4)

Conditional statements

- (if <comp> <expr1> <expr2>)

  >> (if (> a 0) (+ a 2) (- a 2))
  a+2 or a-2

  >> (if (> a 0) (+ a 2) (- a “foo”))
  Error!

>> (cond
  ((< 3 2) 1)
  ((< 4 3) 2)
  (else 3))
 3

Function definitions

- User defined functions:

```
>> (define min (lambda (a b) (if (< a b) a b) ) )
>> (min x y)
>> (min 24 45)
```

Predefined functions:

```
(boolean? x)
(char? x)
(string? x)
(symbol? x)
(number? x)
(pair? x)
(list? x)
```

➔ #t or #f

Bindings

```
>> (let ( (a 3)
          (b 4)
          (square (lambda (x) (* x x)))
          (plus +))
     (sqrt (plus (square a) (square b))) )
```

➔5

```
>> (let ((a 3))
      (let ((a 4)
            (b a))
        (+ a b)))
```

➔ 7

Bindings

- let:
  - does not allow recursive calls
  - “all at once” visibility at the end of the declaration
- letrec:
  - allows recursive calls
- let*:
  - “one at a time visibility”

Bindings

```
>> (letrec ((fact
                (lambda (n)
                  if (= n 1) 1
                      ( * n (fact (- n 1) ))))))
                fact 5))
➔ 120
```

Lists and Numbers

- car: returns the head of a list
- cdr: returns the rest of the list (everything after the head)
- cons: joins a head to the rest of the list

- Take the list (2 3 4)

```
>> (car `(2 3 4))
2
>> (cdr `(2 3 4))
(3 4)
>> (cons 1 `(2 3 4))
(1 2 3 4)
```

13

Lists and Numbers

- null? predicate determines whether its argument is the empty list

```
>>(null? (cdr `(2)))
#t
```

14

Equality Testing and Searching

- eq?
  - tests whether its arguments refer to the same object
- eqv?
  - tests whether its arguments are semantically equivalent
- equal?
  - tests whether its arguments have the same recursive structure with semantically equivalent leaves

15

Equality Testing

eq? :

➔ Return `#t' if X and Y are the same object, except for numbers and characters.
For example,

```
>>(define x (vector 1 2 3))
>>(define y (vector 1 2 3))

>>(eq? x x) => #t
>>(eq? x y) => #f
```

16

Equality Testing

eqv? :

- Return `#t' if X and Y are the same object, or for characters and numbers the same value.
- On objects except characters and numbers, `eqv?' is the same as `eq?' above, it's true if X and Y are the same object.
- If X and Y are numbers or characters, `eqv?' compares their type and value.
- An exact number is not `eqv?' to an inexact number (even if their value is the same).

```
>> (eqv? 3 (+ 1 2)) => #t
>> (eqv? 1 1.0) => #f
```


Equality Testing

equal? :

- Return `#t' if X and Y are the same type, and their contents or value are equal.
- For a pair, string, vector or array, `equal?' compares the contents, and does so using the same `equal?' recursively, so a deep structure can be traversed.

```
>> (equal? (list 1 2 3) (list 1 2 3)) => #t
>> (equal? (list 1 2 3) (vector 1 2 3)) => #f
```

- For other objects, `equal?' compares as per `eqv?' above, which means characters and numbers are compared by type and value (and like `eqv?', exact and inexact numbers are not `equal?', even if their value is the same).

```
>> (equal? 3 (+ 1 2)) => #t
>> (equal? 1 1.0) => #f
```

Searching

- memq -> uses eq?
- memv -> uses eqv?
- member -> uses equal?

➔ Take an element and a list as an argument, return the longest suffix of the list beginning with the element

```
>> (memq `z `(x y z w))
(z w)
>> (memq `(z) `(x y (z) w))
#f
>> (member `(z) `(x y (z) w))
((z) w)
```


Useful Links

- An Introduction to Scheme and its Implementation
  - http://www.federated.com/~jim/schintro-v14/schintro_toc.html
- The Scheme Homepage
  - http://www.swiss.ai.mit.edu/projects/scheme/
- Scheme Interpreter (MIT/GNU)
  - http://www.gnu.org/software/mit-scheme/

Midterm will cover:

- Ch 1.1-1.6
- Ch 2.1-2.3
- Ch 3.1-3.3
- Ch 4.1-4.6
- Ch 6.1-6.7
- Ch 10.1-10.5
- Scheme