2. Lisp/Scheme

CS 3030 Lecture Notes
Yan Shi
UW-Platteville
**PROLOG**

- A database of **facts** and **rules**.

**Facts**
- tasty(cheese).
- made_from(cheese, milk).
- contains(milk, calcium).

**Rules**
- contains(X, Y) :-
  - made_from(X, Z), contains(Z, Y)

**Question:** ?- contains(cheese, calcium).
**Answer:** yes

**LISP: handling lists of data**

- Programs can be treated as data
- Allow writing “self-modifying” programs ➔ **EVOLVE!**
**Racket**

- Dialect of LISP
  - descendent of Scheme
  - Roughly equivalent to the difference between C++ and Java
- Online textbook: *How to Design Programs*
  - required reading: section 2-6, 9, 10
Read-Eval-Print Loop

- A.K.A interactive toplevel or language shell
  - a simple, interactive computer programming environment that takes single user inputs, evaluates them, and returns the result

- DrRacket basics:
  - Top part of screen: editor; where one would write complete programs
  - Bottom part: read-eval-print-loop (interpreter)
  - An interesting part of DrRacket: support for multiple versions of the language
    - Language | Choose Language... to change the language
Numbers and Expressions

- 5, -5, 2/3, #i1.41421356, true, false
  - #i: inexact representation of a real number
    - e.g.: (- #i1.0 #i0.9) = #i0.09999999999999998

- Expression format:
  (operation  A ... B )
  - If A ... B are numbers, evaluate the expression
  - otherwise, evaluate A ... B first.
  - allow arbitrary number of operands for certain operations

  (sqrt 2)
  (expt 2 3)
  (remainder 8 3)
  (* 2 (+ 3 4.5))
  (* 1 2 3 4)
Operations and Expressions

- + - * /
- remainder sqr sqrt log sin expt...
- comparisons: < > = <= >=
  - no !=: (not (= 5 3))

- Boolean expressions:
  (not true)
  (and (< 3 5) (not (= 4 5)))
  (or false false false true)

- Conditional expressions:
  (if (< -3 0)
   (- -3)
   -3)
Functions and Variables

- Functions are named expressions.
- Create a function with
  
  \[
  \text{(define (name params) expression)}
  \]

- Example:

  \[
  \text{(define (area-of-disk r)}
  \text{ (* 3.14 (* r r)))}
  \]

- Variables: names can contain almost any character!

  \[
  \text{(define PI 3.14)}
  \text{ (* PI 2 2)}
  \]

- Documentation: start with `;;` describe prototype and purpose

  \[
  ;; \text{area-of-disk: number} \rightarrow \text{number}
  ;; \text{given the radius of a disk, return the area}
  \]
- Compare a and b, return 1 if a>b, -1 if a<b, 0 if a=b
- cond: [] or () for conditions.

```lisp
(define (compare a b)
  (if (> a b)
      1
      (if (= a b)
          0
          1)))
```

OR

```lisp
(define (compare a b)
  (cond ((> a b) 1)
        ((= a b) 0)
        (else -1)))
```

can be confusing!
Local Binding: let

- **Parallel binding**: bind a set of identifiers, each to the result of some expression, for use in the let body.

- **Syntax:**
  
  ```
  (let ([id1 expr1] ...) body)
  ```

  - evaluates the `expr` and bind the result to `ids`, left-to-right
  - then evaluates the `body`, in which the `ids` are bound.

- **Example:**

  ```scheme
  (define [pow4 x]
    (let ((x (sqr x)))
      (sqr x)))

  (define (f x y)
    (let ([x 5] [y x])
      (+ x y)))
  ```
Exercise

- Calculate salary by weekly working hours and hourly rate. If there is any over-time (above 40 hours), the over-time rate becomes 1.5*rate.
Symbols

- Symbol: a sequence of keyboard characters preceded by a single forward quotation mark:
  - Atomic data
  - No inherit meaning

- Example: ‘the ‘dog ‘ate ‘a ‘cat! ‘and%so%on

- comparison operation (the only basic symbol operation):
  \[(symbol=? \symbol1 \symbol2)\]

- example:
  - (symbol=? 'Hello 'Hello) = true
  - (symbol=? 'Hello 'Howdy) = false
  - (symbol=? 'Hello x) = true if x stands for 'Hello
Characters

- A racket character corresponds to a **Unicode** value.
  - Printable: `\A`
  - Unprintable: `\u0011`
  - Special ones: `\space`, `\newline`

- Some operations:

  ```scheme
  > (integer->char 65)  ➞   #\A
  > (char->integer #\A) ➞   65
  > (char-alphabetic? #\A) ➞   true
  > (char-numeric? #\0) ➞   false
  > (char-whitespace? #\newline) ➞   true
  > (char-downcase #\A) ➞   #\a
  > (char-upcase #\ß) ➞   #\ß
  > (char=? #\a #\A) ➞   false
  ```
String is a second form of symbolic data.
- Compound data (not atomic): a fixed-length array of characters
- For now, treat strings as fancy symbols

Some string operations:

```scheme
> (string? "Apple") ➞ true
> (make-string 5 #\z) ➞ "zzzzz"
> (string #\A #\d #\a #\m) ➞ "Adam"
> (string-length "apple") ➞ 5
> (string-ref "apple" 3) ➞ #\l
> (substring "apple" 1 4) ➞ "ppl"
> (string-append "apple" "ton") ➞ "appleton"
> (string>? "apple" "app") ➞ true
> (string->list "apple") ➞ (list #\a #\p #\p #\l #\e)
```
List

- A collection of objects (compound data)
- Empty list: empty
- Forming a list:
  - (list 5 6 20)
  - (cons 5 (cons 6 empty)) ;; “cons” is short for “construct”
  - (list "John" 'Smith #'m 21)
- Embedded list:
  - (list (list 'a 'b) (list #'c #'d))
  - (list empty empty empty empty)
- `(quote) operator:
  - Stops all evaluations
  - it makes all function calls into lists and
  - all operators/function names into symbols
Getting elements in a list

- **first**: return the first element of the list:
  - (first '(4 5 6)) ➞ 4
  - (first (list '(1) 2 3)) ➞ (list 1)

- **rest**: return the remainder of the list without the first item:
  - It always returns a list:
    - (rest '(4 5 6)) ➞ (list 5 6)
    - (rest (list '(1) 2 3)) ➞ (list 2 3)
    - (rest '(1)) ➞ empty

- Alternative names for **first** and **rest**:
  - car (contents of address register)
  - cdr (contents of data register)

- **second, third, ..., eighth**: get the xth element in a list
  - (second (list 1 3 9))
List operations

- **append**: joins multiple lists together
  - (append '(1 5) '#\a '#\b))
- **member**: check if an item is in a list
  - (member 'a '(a b c d))
  - (member 'a '(1 3 5 7))
- **remove**: remove first occurrence of an item is in a list
  - (remove 'a '(b a n a n a))
  - (remove 1 '(2 3))
- **length**: length of a list
  - (length '(a b c))
- **reverse**: return the list with the elements in reverse order
  - (reverse '(a b c))
Exercise (tricky)

What does the following do?

- `(remove 'c '(a c c e p t))`
- `(remove '(2) '( 2 3 5))`
- `(member 1 (cons 3 (cons 5 1)))`
- `(append '(1) 2)`
- `(cons '(1 3) '(3 4))`
- `(rest (first (rest '(6 (a b) () ((5))) ))))`
- `(append (list 1 2) (cons '(c) (list 4 5)))`
- `(first '(rest (a b)))`
- `(first (rest '(a b)))`
- `(reverse '(a (1 2) b () [c d])))`
Example: representing a tree

- How to represent a tree in racket?

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

What if there is only one child?

```
(define (node value left right)
  (list 'node value left right))
```

See example code: binary-tree.rkt
Predicates

- Functions that return true/false;
  - usually ends with ?

- Examples:
  - `empty?`: true if an empty list
  - `cons?`: returns true if the argument is a non-empty list (a construction),
  - `number?`
  - `integer?`

- Example:

```lisp
(define (atom? x)
  (if (not (or (empty? x) (cons? x)))
      true
      false))
```
Advantage of recursion: direct translation from math definitions to code! (e.g., factorial)

```
(define (! n)
  (if (zero? n)
      1
      (* n (! (- n 1)))))
```

Use recursion to implement looping: (e.g. length of a list)

```
(define (len ls)
  (if (empty? ls)
      0
      (+ 1 (len (rest ls)))))
```

In this course, you should only use recursion for looping!

Exercise: recursion

- Sum from 1 to n
- Implement append
- Construct a list of squares values of 1 to n
- Count all atoms in a nested list
Structures

- **Syntax:**
  
  `(define-struct struct-name (var1 ... ))`

- This by default defines several operations:
  
  - `make-date` ; create a date object
  - `date-month`, `date-day`, `date-year` ; return fields
  - `date?` ; true if it is the date structure

- **Example**

  `(define-struct date (month day year))`
  `(define today (make-date 'Feb 6 2015))`
  `(date-month today)`

*Can you rewrite the BST using structure?*
Question: Can we apply the same operation to a collection of inputs one by one?

— Use the operation(function) as an argument!

Simple example:

```
(define my-add +)
(my-add 1 2 3 4)
```

```
(define (merge a b)
  (let ([op (if (and (number? a) (number? b))
              + list)])
    (op a b)))
```

Similar as function pointers in C++ (STL notes)
High-order functions

- Functions that take other functions as arguments
  - apply: apply a function to a list of values
    - apply : function X list -> function-result
    - (define (average lst) (/ (apply + lst) (length lst)))
      (average '(1 4 6))
  - map: map the function on each element in the list
    - map : function X list -> list
    - (map sqr '(1 2 3 4 5))
    - (map + '(1 2) '(3 4))
  - filter: return a list with elements of the input list "passing" the predicate
    - filter : predicate X list -> list
    - (filter positive? '(1 2 -4 0 -19 5))
Sometimes, you want to use a high-order function, but the operation will be used only once, so it is not worthwhile to define a function.

Introduce a "temporary", nameless function that will go away when we are done.

- `(lambda (paras) expressions)`

Examples:

- `((lambda (x) (/ x 2)) 15)`
- `(define (cube-list nums)
  (map (lambda (x) (* x x x)) nums))`
eval

- the opposite of ‘
- executing a list
- need “pretty big”
- **eval : list → value**

```scheme
(define (sum lst)
  (if (empty? lst)
      0
      (+ (first lst)
          (sum (rest lst)))))
```

- This allows creating code and then executing it on the fly!
Summary

- numbers, characters, symbols, strings
- list: very important!
  - empty, list, cons,
  - first, rest, append, member, remove, reverse, length
  - ', eval
- structure
- basic operations:
  - arithmetic, Boolean, predicates, if, cond, let, ...
- recursion: very important!
  - will only use recursion to achieve iterations.
- high-order functions
  - apply, map, filter