CS 131 - Week 7

TA : Tanmay Sardesai

How to find these slides

Piazza -> CS 131 -> Resources -> Discussion 1B

Announcements

- Email tanmays@cs.ucla.edu
- Office Hours Thursday 1:30 pm - 3:30 pm. Bolter Hall 3256S-A
- HW4 will be due Tonight at 11:55 pm
- HW5 will be due March 1st (next Friday) at 11:55 pm

Topics covered today

- Scheme
- HW 5
- Hands on if we have time

Scheme Basics

- Functional Language
- Part of LISP family
- Scheme is ‘dialect’ of LISP
- Minimalistic design
  - Compact language design
  - Simple syntax
- Dynamic Typing
  - Types attached to value not variable.
  - Type checking happens at runtime
- Parameters evaluated before passing - no laziness

Installing scheme

- http://download.racket-lang.org
- Choose OS and bit version

What is Racket

- Racket is a programing language
  - Dialect of Lisp
  - Descendant of Scheme
- Also a family of programming languages
  - It includes all the variants of Racket
- Main tools are:
  - racket: compiler, interpreter, run-time system.
  - DrRacket: programming environment - IDE
  - raco: command line tool to install racket packages, build binaries

Scheme Arithmetic

Binary Operations

```
> (+ 1 2)
3

>(- 5 3)
2

>(+ (- 2 3) 4)
3
```

More examples

(1+39) * (53-45)

(1020 / 39) + (45 * 2)

Sum of 39, 48, 72, 23, and 91

Average of 39, 48, 72, 23, and 91 as a floating point.

Other arithmetic operators

- quotient
- modulo
- sqrt
- sin
- atan
- ……..

```
> (quotient 10 3)
3

>(modulo 5 3)
2
```

Booleans and Comparisons

```
> #t
#t

>(not #f)
#t

>(and #t (or #f #t #f))
#t
```

```
> (equals? 1 2)
#f

>(< 1 2)
#t
```

Lists

- Similar to Ocaml. Lists are stored as linked list in memory
- Unlike Ocaml we can have different types in the same list
- Not to confuse with a pair (improper list)
- Improper lists are those that don’t end with ‘() i.e the empty cell

Lists

Cons creates a pair.

The 1st part is called car - Contents of the Address part of the Register

The 2nd part is called cdr - Contents of the Decrement part of the Register

‘(1 2 . 3) is actually ‘(1 . (2 . 3))

How is this stored in memory?

> (cons 1 2)
‘(1 . 2)

> (cons 1 (cons 2 3))
‘(1 2 . 3)

> (cons (cons 0 “hello) (cons 1 “world”))
‘((0 . “hello”) 1 . “world”)

Lists

Basically same as before but make sure last cell is ‘() - empty list

How is this stored in memory?

```
> ‘()
‘()

> ‘(cons 1 (cons 2 (cons 3 ‘())))
‘(1 2 3)

> (list 1 2 3)
‘(1 2 3)
```

Quote

Quote is used to protect token from being evaluated.

```
> (1 2)
ERROR

> (quote (1 2)
‘(1 2)

> (+ 1 2)
3

> ‘(+ 1 2)
‘(+ 1 2)
```

car and cdr

Functions that return car and cdr part of the list/pair

```
> (car (cons 1 2))
1

> (car (cons 1 (cons 2 '())))
1

> (cdr (cons 1 2))
2

> (cdr (cons 1 (cons 2 '())))
'(2)
```

Defining

- Similar syntax for variables and functions
- define takes 2 arguments
  - Declares the first argument as a global parameter and binds it to the 2nd
- lambda is used to define functions
  - First parameter is list of input arguments

> (define hello “hello world”)

> (define hi (lambda (x)
(string-append “hello “ x)))

>hello
“hello world”

>(hi “bob”)
“hello bob”

More examples

- Think of lambda as anonymous functions
- So we can use shorthand notation

```
> (define (sum3 a b c) (+ a b c))

> (define (hi x) (string-append “hello “ x))
```

```
> (define sum3 (lambda (a b c) (+ a b c)))

> (sum3 4 5 6)
15

> (lambda (a b c) (+ a b c))
#procedure

> ((lambda (a b c) (+ a b c)) 4 5 6)
15
```

If

- It follows the syntax (if predicate then_value else_value)
- predicate should always return a boolean

```
> (if (< 1 2) 1 2)
1
```

Cond

- Group multiple conditionals
- Think if else if else if ...

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

eq?, equal?, =

eq? checks if two items refer to the same thing in memory

= checks equality for 2 numbers

equal? checks structural equivalence

Useful functions for boolean checks

- pair?
- list?
- symbol?
- null?
- string?
- number?
- integer?
- >,<, <=,>=
- odd?
- positive?
- ……..

Local variables - let

- Let helps bind local variables.
- Bindings are only available in the body of let
- Scope is easy to understand based on how code is written. Lexically scoped
- Use let* for the last statement. Syntactic sugar for nested let

```
> (let ((x 1) (y 2)) (+ x y))
3

> (let ((i 1)) (let ((j (+ i 1))) (+ i j)))
3

> (let ((i 1) (j (+ i 1))) (+ i j))
error
```

List operations

- (length (list 1 2 3)) -> 3 ; count number of elements
- (list-ref (list 1 2 3) 1) -> 2 ; extract by index
- (append (list 1 2) (list 3) -> ‘(1 2 3) ; append two lists
- (reverse (list 1 2 3)) -> ‘(3 2 1) ; reverse the list
- (member 4 (list 1 2 3)) -> #f ; check if element is in list

We also have predefined list loops:

- Map, filter, andmap, ormap
  - (map sqrt (list 1 4 9 16)) -> ‘(1 2 3 4)
  - (andmap string? (list “a” “b” 1)) -> #f

Iterating a list

```
> (define (my-length lst)
      (cond
            ((empty? lst) 0)
            (else (+ 1 (my-length (rest lst))))))

> (my-length empty)
0

> (my-length (list 1 2 3))
3
```

Removing consecutive duplicates

We want to write a function that removes consecutive duplicates in a list

```
> (remove-cons-dups (list 1 1 2 2 2 2 3 3 3))
‘(1 2 3)
```

Removing consecutive duplicates

```
> (define (remove-dups lst)
      (cond
            ((empty? lst) empty)
            ((empty? (rest lst)) lst)
            (else
                  (let ((h (first lst)) (t (rest lst)))
                        (if (equals? h (first t))
                              (remove-dups t)
                              (cons h (remove-dups t)))))))
```

Program as lists

```
> ‘(define (my-length lst)
      (cond
            ((empty? lst) 0)
            (else (+ 1 (my-length (rest lst))))))
 ‘(define (my-length lst)
      (cond ((empty? lst) 0) (else (+ 1 (my-length (rest lst))))))
```

Recall that quote gives us a list of symbols

Eval

- Think of this as opposite of quote.
- Takes a list and treats it like a program
- Explore more when you do homework

```
> (eval ‘(+ 1 2))
3
```

Homework 5

http://web.cs.ucla.edu/classes/winter19/cs131/hw/hw5.html

- Task: Write scheme code difference analyzer.
  - Plagiarism detector
  - For given input check if both are similar. I.e different variable names or small changes

Homework 5

- Write a scheme procedure (expr-compare x y)
  - Returns an expression that combines similar parts based on some criteria
  - Use % so that when we run with % as true we get get 1st expression else we get 2nd
  - You also want the summary expression to use the same identifiers as the two input expressions where they agree, and that if x declares the bound variable X in the same place where y declares the bound variable Y, the summary expression should declare a bound variable X!Y and use it consistently thereafter wherever the input expressions use X and Y respectively
  - (expr-compare 12 12) -> 12
  - (expr-compare 12 20) -> (if % 12 20)
  - (expr-compare ‘a ‘(cons a b)) -> (if % a (cons a b))
  - (expr-compare ‘(cons a b) ‘(cons a c)) -> (cons a (if % b c))
  - More examples on the website

Homework 5

Understanding example using both % and !

```
> (expr-compare
      '((λ (a b) (f a b)) 1 2)
      '((λ (a c) (f c a)) 1 2))

((λ (a b!c) (f (if % a b!c) (if % b!c a))) 1 2)
```

Homework 5

- (test-expr-compare x y) that tests your implementation of expr-compare using eval
  - (eval x) and (expr-compare x y) with % bound to #t should be same.
  - (eval y) and (expr-compare x y) with % bound to #f should be same.
- Write test-expr-x and test-expr-y which also tests our expr-compare but on a specific example.
  - Use something that covers all the test cases in the specs.
- Make sure code compiles and runs on racket. Available on Seasnet