CS131, Spring19 – Discussion 1B

Week 7 (05/17/19)

Administrations

1. TA: Wenhao Zhang (wenhaoz@cs.ucla.edu)
2. Office hour:
   - Time: 9:30am ~ 11:30am, Monday
   - Location: 3rd floor, common area in Eng VI
3. Class Announcements
   - All TAs slides are under **resources tab** on Piazza
   - HW4 dued on this past Tuesday
   - HW5 dued on **2019-05-23** (This coming Tuesday)
   - Midterm is still being graded
   - Please dont copy & paste code from stackoverflow or other sources.
4. Today's agenda
   - Scheme/Racket basics
   - HW5 walkthrough

Scheme

Installing scheme

We'll be using racket for this class. Its a language in the Lisp-Scheme family. https://racket-lang.org/

References

Racket Guide - https://docs.racket-lang.org/guide/index.html Racket Reference - https://docs.racket-lang.org/reference/index.html Racket Cheat Sheet - https://docs.racket-lang.org/racket-cheat/index.html

What is scheme?

It is a LISP variant, (the second oldest high level programming language!)

Features:

- functional programming language
- prefix syntax
- dynamically typed : values are typed, not variables -> runtime type checking!
- the line between program and data is very very thin

1. (+ 3 5)
2. (- 1 1 1 1) -> -2

Basics

Identifiers - string with any chars except ( ) [ ] { } " , ' ; # |

ex: hello, string->int, number?

Numbers ex: 1, 1/2, 0.5

There are exact numbers and inexact numbers in scheme, and its kind of interesting to read about it if you're feeling curious - https://docs.racket-lang.org/guide/numbers.html

Booleans ex: #t, #f anything other than #f evaluates to true, so be careful.

Strings "hello world" (display "hello world")

Predicates Type checking during run time!

helpful predicates:

- Numbers: number?, integer? rational? real?
- Booleans: boolean?
- Strings: string?

e.g.

```
(boolean? #f) -> #t
(boolean? 1) -> #f
```

Arithmetic

```
(+ 3 5) -> 8
(- 1 1 1 1) -> -2
(* 3 5) -> 15
(/ 4 3) -> 4/3
(quotient 4 3) -> 1 # quotient floor the result
```

Lists

Lists in scheme are all linked lists

`(list 1 2 3) -> '(1 2 3)`

The empty list is '() and proper lists end with the empty list.

helpful operators: length, append, reverse, member, empty?, list?, pair?

e.g.

```
(reverse '(1 2 3)) -> '(3 2 1)
(member 2 (list 1 2 3 4)) -> '(2 3 4)
(member 9 '(1 2 3 4)) -> #f
```

`(member v lst)` locates the first element of last that is equal to v. If such an element exists, the tail of let starting with that element is returned. Otherwise, the result is #f.

```
(pair? 1) -> #f
(pair? (cons 1 2)) -> #t
(pair? '()) -> #f
```

A pair combines exactly two values. The first value is accessed with the car procedure, and the second value is accessed with the cdr procedure. A list is recursively defined: it is either the constant null(i.e. '()), or it is a pair whose second value is a list.

```
(pair? '(1 2 3)) -> #t
(pair? '()) -> #f
```

Cons

`(cons a d)` Returns a newly allocated pair whose first element is a and second element is d.

`(cons 1 2) -> '(1 . 2)` -> This is the improper list and its denoted by the . notation

improper list

- The last argument is used directly in the tail of the result
- The last argument need not be a list, in which case the result is an "improper list"

e.g.

```
(cdr (list 1 2)) -> (2)
(cdr (cons 1 2)) -> 2

(cons 1 '()) -> (1)
(cons 1 (cons 2 '())) -> '(1 2)

(Here we can draw some cons cells)

(cons 1 (list 2 3)) -> '(1 2 3)
```

All lists are by default pairs. (except for the empty list)

```
(pair? empty) -> #f
(pair? (list 1 2 3)) -> #t
(pair? (cons 1 2)) -> #t
```

car and cdr

- car -> take first
- cdr -> take rest

```
(car '(1 2 3)) -> 1
(cdr '(1 2 3)) -> '(2 3)
```

Exercise:

```
(car '(2 3 4)) -> ? #2
(cdr '(2 3 4)) -> ? #(3,4)
(cdr '(2 . 4)) -> ? #4
```

Equivalence

`equal?`

- checks if its two arguments have the same value.

```
(equal? '(1 2 3) '(1 2 3)) -> #t
(equal? '(a b c) '(d e f)) -> #f
```

`=`

- checks if two numbers are equal

```
(= 3 3) -> #t
(= 500 1/2) -> #f
(= 1 (* 3 1/3)) -> #t
```

`eq?`

- checks if its two arguments are the same. pointer equality!

```
(eq? '(1 2 3) '(1 2 3)) -> #f

(let ([x 5])
  (eq? x x)) -> #t
```

Other useful functions to note

`and`, `or`, `not`

- `and` & `or` use short circuit evaluation

e.g. if `and` finds something false, it immediately returns the false value. if `or` finds something true, it immediately returns the true value

e.g.

```
(and 1 2) -> 2
(and 1 #f) -> #f
(or #f #f) -> #f
(or 1 #f) -> 1
(/ 1 0) -> division by zero
```

question

```
(or 1 (/ 1 0)) -> ? #1
```

Define

- Define variables

```
(define <id> <expr>)
```

- This defines a variable. eg:

```
(define e 2.71828)
(define v (+ 1 2))
```

- Define a function syntax: `(define (<id> <id>*) <expr>+)`
  - define a function with 0 or more arguments
  - the final expr in the body is the return value

eg:

```
(define (timesTwo x) (* x 2))
(define (print_sth) (display "hello world"))
```

we can also define functions that take an arbitrary number of arguments

Exercise write the doTwice function.

```
(define (doTwice f x) (f (f x)))
```

Lambdas

we can write anonymous functions in scheme

syntax: `(lambda (<id>*) <expr>+)`

Exercise the syntax we used for defining functions is just syntatical sugar for lambdas. write doTwice using a lambda.

```
(define doTwice (lambda (f x) (f (f x))))
```

Lets call doTwice, `(doTwice (lambda (x) (+ x 2)) 3)`

Lambda expression for arbitrary num of args There are functions in scheme that can take an arbitrary number of args

ex: `(list 1 2 3) -> '(1 2 3)`

We can recreate this behavior!

syntax: `(lambda <id> <expr>+)`

```
((lambda x x) 1 2 3) -> '(1 2 3)
```

Alternate syntax syntax: `(define (<id> . <id>) <expr>)`

```
(define (id . x) x)
(id 1 2 3) -> '(1 2 3)
```

Conditionals

```
(if <expr> <expr> <expr>)
          and
(cond {[<expr> <expr>]}* )
```

example:

```
(define (evenorodd x)
  (cond
    [(equal? (modulo x 2) 0) "even"]
    [else "odd"]))
```

Let syntax: `(let ( {[<id> <expr>]}*) <expr>+)`

ex: `(let ([x 5] [y 6]) (+ x y)) -> 11`

`(letrec ( {[<id> <expr>]}*) <expr>+)`

letrec lets you define recursive functions as your ids.

Scope of Let: only within the exprs at the end are id's allowed to be referenced. Outside of the exprs, ids aren't allowed to be referenced.

Question:

`(let ([x 5]) (* x (let ([y 6]) y)))` is this okay?

`(let ([x 5]) (* y (let ([y 6]) y)))` is y in scope here?

Exercise 1 write a function that takes a proper list and then return the length of it

```
(define list-length (lambda (lst)
            (if (null? lst)
                0
                (+ 1 (list-length (cdr lst)))))))
```

Why do we need two functions here?

Is this tail recursive? - how could we make it tail recursive?

```
(define list-len (lambda (lst len)
            (if (null? lst)
                len
                (list-len (cdr lst) (+ 1 len)))))
```

Exercise 2 write a factorial function

```
(define factorial (lambda (n fact)
            (if (< n 2)
                fact
                (factorial (- n 1) (* fact n)))))
```

Exercise 3 write a function that checks for equality between two lists with the same length.

Quote Lists and programs are equivalent, we just decide whether or not to execute the code or treat it as data.

```
(quote (+ 1 2)) -> '(+ 1 2)

(list '+ 1 2) -> '(+ 1 2)
(list (+ 1 2)) -> '(3) (the list of just 3)
```

HW5

Goal: Write a program to detect similarities between two Scheme programs

`expr-compare` function takes two expressions and returns a new expression with similar parts combined

Variable `%` defines which program we want to execute

case 1

```
(expr-compare 12 12) -> 12
(expr-compare 12 20) -> (if % 12 20)
(expr-compare 'a '(cons a b)) -> (if % a (cons a b))
```

case 2

```
(expr-compare '(cons a b) '(cons a c)) -> (cons a (if % b c))
(expr-compare '(cons (cons a b) (cons b c)) '(cons (cons a c) (cons a c)))
-> (cons (cons a (if % b c)) (cons (if % b a) c))
```

case 3

```
(expr-compare '(list) '(list a))
-> (if % (list) (list a))
(expr-compare '(quote (a b)) '(quote (a c)))
-> (if % '(a b) '(a c))
```

```
(expr-compare '(if x y z) '(g x y z))
-> (if % (if x y z) (g x y z))
```

case 4

```
(expr-compare '((lambda (a) (f a)) 1) '((  (a) (g a)) 2))
-> ((  (a) ((if % f g) a)) (if % 1 2))
```

case 5

```
(expr-compare '((lambda (a) a) c) '((lambda (b) b) d))
-> ((lambda (a!b) a!b) (if % c d))
```