CS131 - Week 7

UCLA Spring 2019
TA: Kimmo Karkkainen

Today

- Scheme
- Homework #5

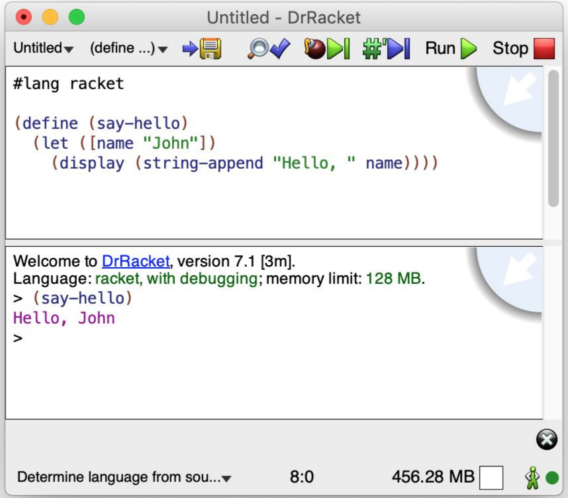
Scheme

Scheme introduction

- Functional programming language
- Part of LISP language family
  - LISP developed in 1958 by John McCarthy
  - Pioneered many new concepts, including:
    - Garbage collection
    - Recursion
    - Dynamic typing
    - Program code as a data structure
- Scheme is a *dialect* of LISP
  - Created 1970s at MIT AI Lab
  - Historically very popular language in academia
  - Designed to be very minimal
- You’ll encounter Scheme again in CS161 - Artificial Intelligence

Racket

- For the homework, we will use Racket which is a descendant of Scheme
  - Racket implements the Scheme standard plus some additional features
- https://racket-lang.org
- You can use DrRacket IDE or any text editor
  - DrRacket is a very minimal IDE, just a text editor and an interactive environment
  - DrRacket can make your life a lot easier
    - Simple debugger, matching parentheses, ...



Hello world

helloworld.ss file:

```
#lang racket

(display "Hello, world!\n")
```

Command line:

```
$ racket helloworld.ss
"Hello world"
```

Basic syntax

- Comments
  - ; (semi-colon) starts a line comment
  - #| Block comment |#
- Numbers
  - 1, 1/2, 3.14, 6.02e+23
- Strings
  - “Hello, World!”
- Booleans
  - #t, #f

Function calls

- In Scheme, function name always comes first in function calls
    - Even arithmetic operations are function calls!

```
> (display "Hello")
Hello

> (+ 1 2)
3

> (+ 1 2 (- 4 3))
4

> (/ (+ 1/3 1/6) 2)
1/4
```

Function calls - Exercises

Convert the following to Scheme expressions:

1. 1.2 × (2 - 1/3) + -8.7
2. (2/3 + 4/9) ÷ (5/11 - 4/3)
3. 1 + 1 ÷ (2 + 1 ÷ (1 + 1/2))

Definitions

- Defining variables and functions have a similar syntax:

```
(define pi 3.14)
> pi
3.14

(define (print-name name)
          (display (string-append “Hello, “ name)))
> (print-name "Steve")
Hello, Steve
```

Lambda functions

- Anonymous functions can be defined with *(lambda (args*) expr)*

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

> ((lambda (a b c) (+ a b c)) 1 2 3)
6
```

Local bindings

- *Let* keyword defines a new variable inside an expression

```
(define (say-hello)
  (let ([name "John"]
        [greeting "Hello, "])
    (display (string-append greeting name))))

> (say-hello)
Hello, John
```

Functions - Exercise

- How would you implement *dotwice* function?

```
> (dotwice (lambda (x) (* x 2)) 2)

8
```

Functions - Exercise

- How would you implement *dotwice* function?

```
> (dotwice (lambda (x) (* x 2)) 2)

8
```

```
(define (dotwice f x) (f (f x)))
```

Identifiers

- Scheme is very liberal with identifiers
- Any of the following could be used as identifiers:

```
+
Hfuhruhurr
integer?
pass/fail
john-jacob-jingleheimer-schmidt
a-b-c+1-2-3
```

- Forbidden characters: ( ) [ ] { } " , ' ` ; # | \

Comparison operators - Equality

Three comparisons for equality:

- **(= 1 2)** checks if numbers are equal
- **(equal? (list 1 2 3) (list 1 2 3))** checks if other values are equal (recursively)
- **(eq? a a)** checks if object references are equal (rarely needed)

Comparison operators - Inequality

- Basic comparison operators (=, <, >, <=, >=)

```
> (< 1 2)
#t

> (< 1 2 3)
#t

> (< 1 3 2)
#f
```

Checking types

- Scheme provides functions to check types, for example:

```
> (number? 5)
#t
> (string? "My string")
#t
> (list? (list 1 2 3 4))
#t
> (pair? (cons 1 2))
#t
```

Conditionals - If

- Syntax for if statements: *(if <condition> <then> <else>)*

```
> (if (equal? 1 2) "Equal" "Not equal")
"Not equal"

> (if (< 1 2) #t #f)
#t
```

Conditionals - Cond

- Syntax for cond statements: *(cond [<condition> <then>] [<2nd-condition> <then>])*

```
> (cond [(boolean? 1) "One is a boolean value"]
        [(boolean? #t) "#t is a boolean value"]
        [(boolean? #f) "#f is a boolean value"])
"#t is a boolean value"
```

- Note: Compiler will not check whether this is exhaustive
    - Returns nothing if none of the conditions are true

Short-circuit evaluation

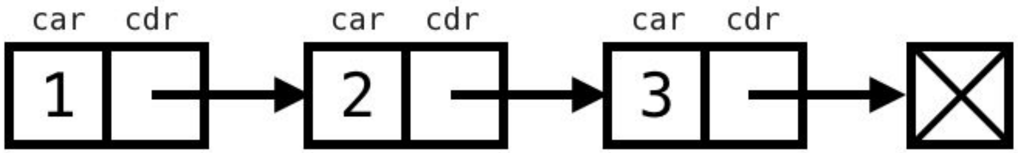
- *and/or* execute instructions until the expression has been evaluated
- Return the last evaluated value

```
> (or #f 2 3)
2

> (and 2 3)
3

> (and 2 3 #f 6)
#f
```

- Note: Everything that is not *#f* is true
    - *(if 5 1 2) -> 1*

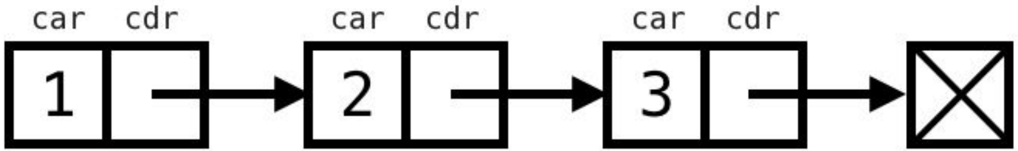
Lists

- Scheme uses linked lists, similar to OCaml and Prolog:



- To create a list, you can use *(list 1 2 3)* or *‘(1 2 3)*
- To access the head, you can use *(car my-list)* or *(first my-list)*
- To access the tail, you can use *(cdr my-list)* or *(rest my-list)*
- Empty list: *‘()* or *empty*
  - Checking for an empty list: *(empty? ‘()) -> #t*

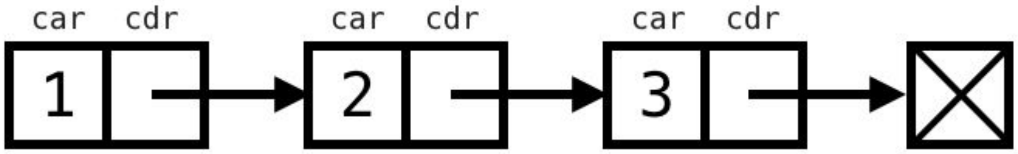
Lists




```
> (define my-list (list 1 2 3))
> (car my-list)
1
> (first my-list)
1
> (cdr my-list)
'(2 3)
> (rest my-list)
'(2 3)
```

```
> (car (cdr (cdr my-list)))
3
> (caddr my-list)
3
```

Lists

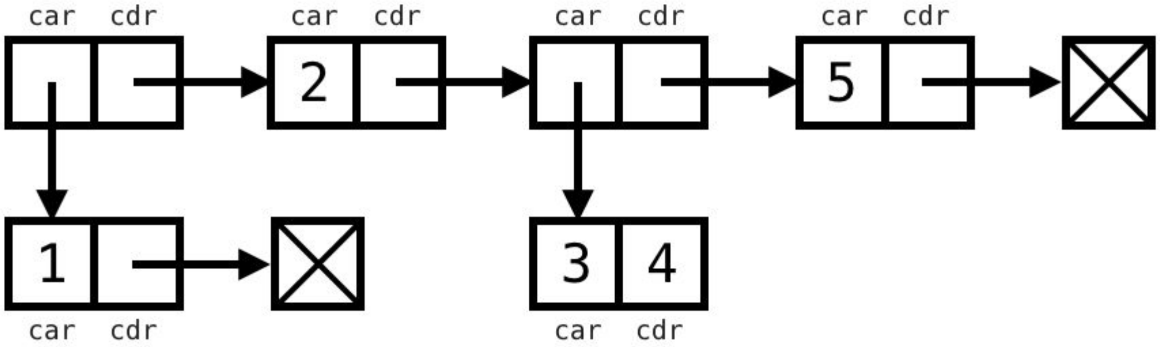


- List can also be constructed from *cells* with *cons* keyword:

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

Lists

- Cells can have different data types inside them:




```
> (cons (cons 1 '()) (cons 2 (cons (cons 3 4) (cons 5 '()))))
'((1) 2 (3 . 4) 5)
```

Lists - Exercises

What do the following expressions evaluate to:

1. (car (cons 1 (list 2 3)))
2. (cons (list 1 2) (list 3 4))
3. (cons (car (list 1 2 3)) (cdr (list 4 5 6)))

Lists - Exercises

- How would you write *mylength* function that returns the length of a list?

```
> (my-length '(1 2 3 4))
4
```

Lists - Exercises

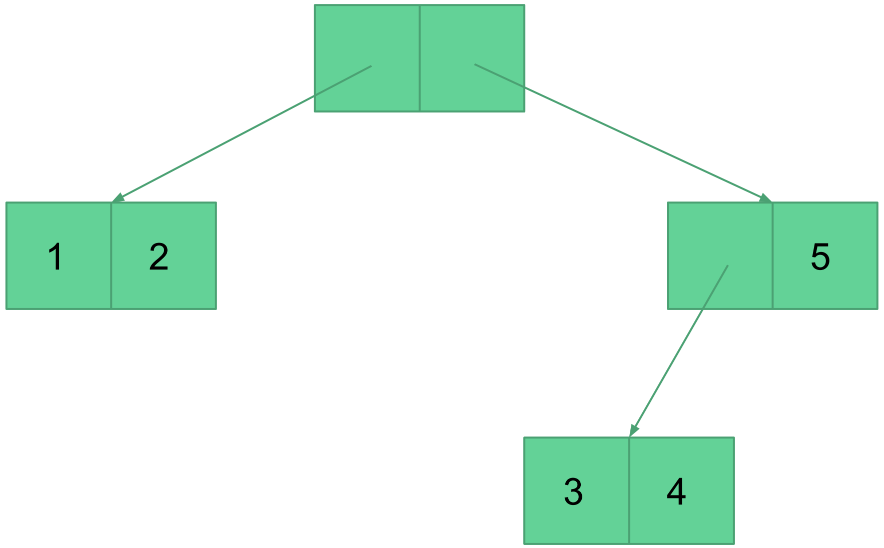
- How would you write *mylength* function that returns the length of a list?

```
> (my-length '(1 2 3 4))
4
```

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

Lists - Exercises

How would you write the following structure in Scheme?



List operations

- map
    - *(map (lambda (x) (+ x 1)) '(1 2 3)) -> ‘(2 3 4)*
- filter
    - *(filter (lambda (x) (> x 2)) '(1 2 3 4)) -> ‘(3 4)*
- foldl/foldr
    - *(foldl (lambda (a b) (+ a b)) 0 '(1 2 3 4)) -> 10*
- sort
    - *(sort '(5 4 3 2 1) <) -> '(1 2 3 4 5)*
- length
    - *(length ‘(1 2 3 4 5)) -> 5*
- reverse
    - (reverse ‘(1 2 3 4 5)) -> ‘(5 4 3 2 1)

Lists

- Notice the list structure looks very similar to Scheme code:

```
‘(1 2 (3 (4 5) 6))
```

- **‘** before any expression forces Scheme to interpret the following expression as *symbols* (= data), not as a function call
- Name **LISP** comes from ***List Processor***

```
> (define my-program '(display "Hello, World!\n"))
> my-program
'(display "Hello, World!\n")
> (eval my-program)
Hello, World!
```

- Note: *eval* does not include the current namespace in the call!

Eval - namespaces

- In the interpreter, *eval* works without defining a namespace
- In your code file, it has to be defined:

```
(define ns (make-base-namespace))
(eval my-program ns)
```

Programs as lists

```
> (define my-program '(display "Hello, World!\n"))

> (car my-program)
'display
> (cdr my-program)
'("Hello, World!\n")
> (car (cdr my-program))
"Hello, World!\n"
```

Note that *'(<list contents>)* is a shorthand for *(quote (<list contents>))*

Programs as lists

- *Quote (')* gives you a list of symbols
- *Quasiquote (`)* and *unquote (,)* allow you to combine symbols and evaluated code:

```
> (quasiquote (my-function (unquote (+ 1 2))))
'(my-function 3)

> `(my-function ,(+ 1 2))
'(my-function 3)
```

Homework #5

Homework #5

- DL Thursday May 23
- Goal: Write a program to detect similarities between two Scheme programs

Homework #5

- *expr-compare* function takes two expressions and returns a new expression with similar parts combined
- Variable **%** defines which program we want to execute

```
(expr-compare 12 12)
-> 12

(expr-compare 12 20)
-> (if % 12 20)

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

Homework #5

- If the differences are deeper inside the program, combine the outer parts

```
(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))
```

Homework #5

- In some cases, similar parts can't be combined:

```
(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))
```

Homework #5

- Lambda and λ should be combined:

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

Homework #5

- If we define new variables with different names, combine them:

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

- Need to replace all occurrences of these variables within the lambda expression
- Hint: Dictionary can be useful for keeping track of variable names
  - Provides an immutable implementation

Resources

- Download Racket (Includes DrRacket IDE)
- The Racket Guide

Questions?