Week 7 Discussion

CS 131 Section 1B

13 May 2022

Danning Yu

Announcements

- HW5 released, due 5/20
- Project released, due 5/31
- HW2 and midterm grades released
- Homeworks should be submitted on BruinLearn, under Assignments
- Before submitting
  - Make sure your code compiles on SEASnet server
  - Make sure your function signatures are correct
  - Follow all instructions and specifications
  - Do not submit files in a .zip unless told to do so
- Help and starter code from past TAs
  - https://github.com/CS131-TA-team

Scheme

Scheme

- Functional programming language, like OCaml
- Part of LISP programming language family
  - LISP developed in 1958 by John McCarthy
  - Pioneered many new concepts: GC, recursion, dynamic typing, etc.
  - Important feature: Program code as a (list-like) data structure
  - Pervasive use of parentheses
- Scheme: a dialect of LISP
  - Created in 1970s at MIT AI Lab (CSAIL)
  - Historically popular in academia
  - Designed to be minimal

Racket

- Language for HW5
- LISP dialect based on Scheme
- Can use DrRacket IDE or any text editor
  - A very minimal IDE, text editor + interactive environment
  - It can make your life a little bit easier
  - Simple debugger, matching parenthesis
- We suggest using Racket 8
  - Available on SEASnet as v8.4
  - Start with `racket` command
  - `.ss` file extension

Hello World in Racket

- Comments
  - Semi-colon `;` starts a single-line comment
  - `#| this is block comment |#`
- Numbers: `1, 1/2, 3.14, 6.02e+23`
- Strings: `"Hello, world!\n"`
- Booleans: `#t, #f`
- Hello world: `helloworld.ss`

```
#lang racket
(display "Hello, world!\n")
```

- To run: `racket helloworld.ss`

Identifiers in Racket

- Scheme allows nearly anything in identifiers
  - Much more lenient than other languages
- For example, any of the following can be used:
  - `+`
  - `%`
  - `integer?`
  - `pass/fail`
  - `a-very-long-identifier`
  - `a-b-c+1+2+3`
- Forbidden characters: `( ) [ ] { } " , ' ` ` ` `; # | \`

Function Calls

- All code is written as *s*-expressions
  - In layperson terms, parenthesized lists
  - (), [], {} are identical, () is typically used
- For function calls, the name of function is the first element, followed by arguments

```
> (display "Hello")
Hello
> (+ 1 2)
3
> (+ 1 2 (- 4 3))
4
```

```
> (+ 1 2 3 4)
10
; + and * can take more
; than 2 arguments

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

Function Calls: Infix to Prefix

- Convert the following expressions to Scheme
  - 3.2 * (1 - 0.3) + -8.7
  - (2/3 + 5/9) ÷ (3/11 - 7/4)
- Idea: convert infix expressions to prefix ones
- Pay attention to priority!
- Answers
  - (+ (* 3.2 (- 1 0.3)) -8.7)
  - (/ (+ 2/3 5/9) (- 3/11 7/4))

Definitions

- Variables and functions are defined using the `define` keyword
  - Not a function call, but rather a syntactic form, despite having the same syntax
  - Is a local binding

```
(define pi 3.14)
> pi
3.14
(define (print-name name)
      (display (string-append "Hello, " name)))
> (print-name "Steve")
Hello, Steve
```

Function Naming Conventions

- Functions that return boolean value usually end with ?
  - Examples: equal?, zero?
  - Except arithmetic comparisons (e.g. <, >)
- Functions that case side effects usually end with !
  - Examples: set!
  - **Do not use these functions in the homework**

Lambda Functions

- Use the `lambda` keyword to define an anonymous function

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

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

Local Bindings

- The `let` keyword defines new variables inside a smaller scope
  - Syntax: `(let (<var definitions>) expr)`

```
(define (sum-squares a b)
  (let ([a-squared (* a a)]
        [b-squared (* b b)])
   (+ a-squared b-squared)))
```

```
> (sum-squares 5 6)
61
```

Local Function Bindings: `let` and `lambda`

- Local function bindings can be created with `let` and `lambda`
  - When creating local recursive functions, use `letrec` instead of `let`, except when you are using named `let`

OCaml

```
let rev_list l =
  let rec rev_helper acc =
function
    [] -> acc
    | h::t -> rev_helper
(h::acc) t
  in rev_helper [] l;;
```

Scheme

```
(define (rev-list l)
  (letrec
    ([rev-helper
    (lambda (acc l)
         (if
         (null? l)
        acc
         (rev-helper
         (cons (car l) acc)
         (cdr l))))])
    (rev-helper (list) l)))
```

Named `let`

- Binds a function identifier that is only visible in the function's body
    - Syntax: `(let proc-id ([arg-id init-expr] ...) body ...+)`
- Equivalent letrec form:

```
(letrec ([proc-id (lambda (arg-id ...)
                     body ...+)])
  (proc-id init-expr ...))
```

```
(define (rev-list l)
  (letrec
    ([rev-helper
    (lambda (acc l)
          (if (null? l)
          acc
          (rev-helper
          (cons (car l) acc)
          (cdr l))))])
    (rev-helper (list) l)))
```

```
(define (rev-list-2 l)
  (let rev-helper-2 ([acc (list)] [l l])
          (if (null? l)
          acc
          (rev-helper-2 (cons (car l)
acc)
                           (cdr l)))))
```

Comparison Operators

- Three comparisons for equality
  - `(= 1 2)` checks if numbers are equal
  - `(equal? (list 1 2 3) (list 1 2 3))` checks if values are equal recursively
  - `(eq? a a)` checks if object references are equal (rarely needed, especially for programs without side effect)
- Comparison operators: `<, >, <=, >=`
  - They can take multiple arguments
  - `(< 1 2) => #t`
  - `(< 1 2 3) => #t`
  - `(< 1 3 2) => #f`

Checking Types

- Scheme provides functions to check types
  - For any basic type: <type name>?

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

Conditionals: `if` and `cond`

- Syntax: `(if <cond> <then-expr> <else-expr>)`

```
> (if (equal? 1 2) "Equal" "Not equal")
"Not equal"
> (if (< 1 2) #t #f)
#t
```

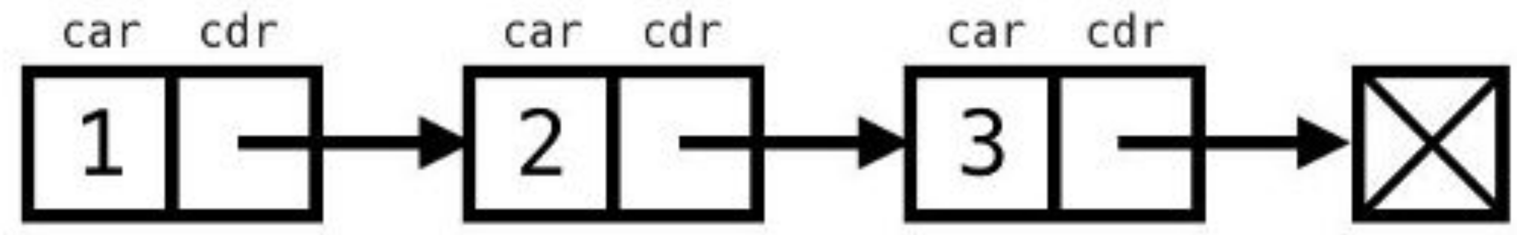
- Syntax: `(cond [<condition> <then>] [<2nd-condition> <then>] ...)`
- Return the then value of first true conditional

```
> (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: Unlike OCaml, compiler will not check whether conditions are exhaustive
- Returns nothing if none of the condition is true

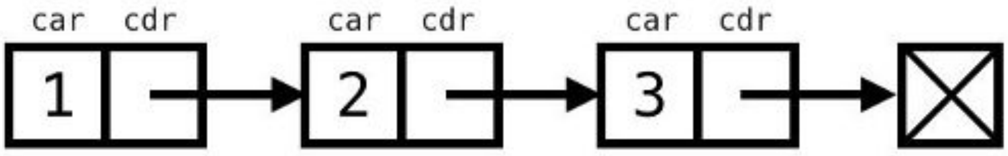
Lists (1 of 3)

- Similar to OCaml and Prolog, Scheme uses a singly-linked list
- Creating a list: `(list 1 2 3)` or `'(1 2 3)`
- Accessing head: `(car my-list)` or `(first my-list)`
- Accessing tail: `(cdr my-list)` or `(rest my-list)`
- Empty list: `'()` or `empty`
- Checking for an empty list: `(null? '())` returns `#t`



Lists (2 of 3)

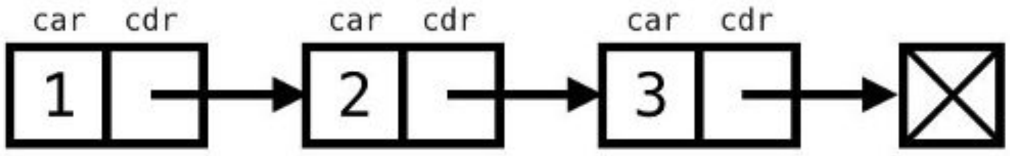
```
> (define my-list (list 1 2 3))
> (car my-list)
1
> (first my-list)
1
> (cdr my-list)
'(2 3)
> (rest my-list)
'(2 3)
> (car (cdr (cdr my-list)))
3
> (caddr my-list)
3
```



Lists (3 of 3)

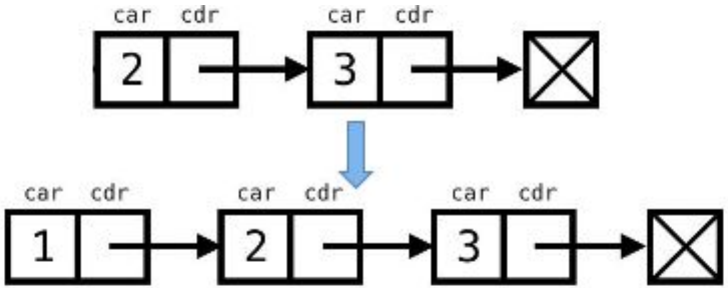
- Use the `cons` keyword in Scheme to construct a new list by adding an element to the head of another list
  - Similar to OCaml's `::` operator
- Example: constructing list from individual cells:

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

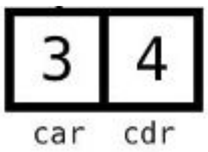


The `cons` Operator

- `cons` constructs a `cons` cell with two slots: `car` and `cdr`
  - `car` is a value and `cdr` points to a list, so `cons` gives us a normal list
- A `cons` cell can actually hold different data types, allowing for more flexible data structures



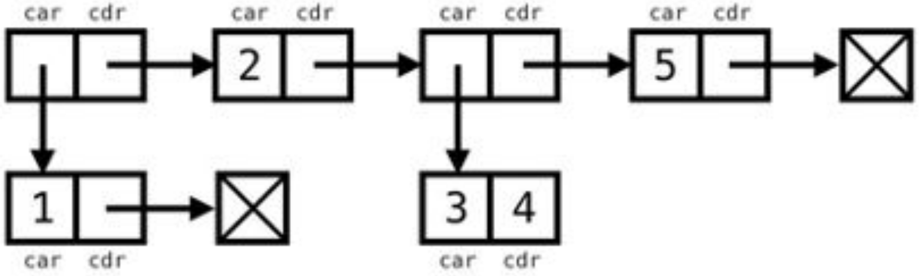
- A list as a pair:




```
> (cons 3 4)
'(3 . 4)
```

- A more complicated example

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



List Exercises

- What do the following expressions evaluate to?
  - `(car (cons 1 (list 2 3)))`
  - `(cons (list 1 2) (list 3 4))`
  - `(cons (car (list 1 2 3)) (cdr (list 4 5 6)))`
- Answers
  - `1`
  - `'((1 2) 3 4)`
  - `'(1 5 6)`

Hash Map

- Unordered key-value store
- Amortized constant time access/update

```
(define first-hashmap (hash-set (hash) "a" 5))
(define second-hashmap (hash-set first-hashmap "b" 7))

> (hash-ref second-hashmap "b" "Not found")
7

> (hash-ref first-hashmap "b" "Not found")
"Not found"
```

Programs as Data (1 of 3)

- Notice that the list structure looks like Scheme code

```
'(1 2 (3 (4 5) 6))
```

- `'` (single quote) makes the following expression a list of symbols
    - Contents are not evaluated

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


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

`eval` and Namespaces

- The `eval` operator evaluates the given expression
- 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 Data (2 of 3)

- Inspecting quoted program symbols with normal list operations
  - `'(<list contents>)` is a shorthand for `(quote (<list contents>))`

```
> (define my-program '(display (+ 1 2)))
> (eval my-program)
3


> (car my-program)
'display
> (cdr my-program)
'((+ 1 2))
> (car (car (cdr my-program)))
'+
```

Programs as Data (3 of 3)

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

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

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

Programs as Data: Exercise 1

- Problem: for debugging purposes, we'd like to print the values that are added together within a certain code
  - Want to replace all the + operations that appears in the code with our own print-and-add procedure

```
(define (print-and-add x y)
  (begin
    (display (string-append
    "+ "
    (number->string x)
    " "
    (number->string y)
    "\n"))
    (+ x y)))
```

```
(+ 1 (+ 2 (+ 3 (+ 4 5))))

Should print:
+ 4 5
+ 3 9
+ 2 12
+ 1 14
```

Programs as Data: Exercise 1

- How to solve this problem
  - Iterate through list of program symbols recursively
  - When we see a `'+`, replace with `'print-and-add`

```
(define (debug program)
  (cond [(list? program) (map debug program)]
        [(equal? program '+) 'print-and-add]
        [else program]))
```

Programs as Data: Exercise 2

- Create a function `transform-if` that does the following transformation on a program
  - For all `if` construct inside, do a negation of the guard and then swap the then and else branches
- Example: `(if (> x 3) (+ y 5) 6)` becomes `(if (not (> x 3)) 6 (+ y 5))`
- Solution

```
(define (transform-if p)
  (if (and (list? p) (not (null? p)))
    (let ([tp (map transform-if p)])
    (if (equal? (car p) 'if)
          `(if (not ,(cadr tp))
                ,(cadddr tp)
                ,(caddr tp))
          tp))
    p))
```

Homework #5

`expr-compare`

- Goal: Write a program to detect similarities between two Scheme programs
  - Think of it as a `diff` program
- `expr-compare` 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))
```

`expr-compare`: Combine Similar Parts

- If the differences are deeper inside the program, combine outer parts
  - Only diff the parts that are different

```
> (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))
```

`expr-compare`: Combine Similar Parts

- In some cases, similar parts cannot be combined

```
; empty list is different
> (expr-compare '(list) '(list a))
(if % (list) (list a))

; comparison stops at “quote”
> (expr-compare '(quote (a b)) '(quote (a c)))
(if % '(a b) '(a c))

; “if” is not a function
> (expr-compare '(if x y z) '(g x y z))
(if % (if x y z) (g x y z))
```

`expr-compare`: `lambda` and `λ`

- `lambda` and `λ` should be combined
  - (Yes, Racket support Unicode characters)
  - When both `lambda` and `λ` appear, use `λ` after combination
  - Two `lambda`s cannot be combined into `λ`
  - Typing the lambda symbol
    - `(define LAMBDA (string->symbol "\u03BB"))`
  - In DrRacket: Insert > Insert λ, or press Ctrl-\

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

`expr-compare`: Renamed Variables

- If we declare new variables with different names, combine them
- Need to replace all occurrences of these variables within the lambda expression
- Hint: use a hashmap to keep track of renamed variables

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

Resources

- Download Racket (Includes DrRacket IDE)
  - https://racket-lang.org/download/
- The Racket Guide
  - https://docs.racket-lang.org/guide/index.html
- The Scheme Programming Language (Dybvig)
  - https://www.scheme.com/tspl4/

Thank You