Outline

- (*) Homework 4 plain_tower
- Homework 5
- Scheme (Racket)
- Scheme and Prolog in Exams
(*) HW4: plain_tower efficiency
Leftover from Prolog: faster plain_tower

- We have hints on building grids at
  - Note that these are **hints** only
  - It is **not** even an efficient implementation, **only** better than enumerating all the lengths of each row.

- Other hints?
  - Order of execution matters, Prolog check the rules **in order**
  - Cuts: https://en.wikibooks.org/wiki/Prolog/Cuts_and_Negation
HW 5
Overview on HW5

- **Due date:** May 20th
- **Goal:** Write a program to detect similarities between two Scheme programs
- We use [Racket](https://racket-lang.org), not the standard MIT-scheme
- **Resources** on Scheme and Racket (course page)
- **Documentation** of Racket, and Racket [guide](https://docs.racket-lang.org/guide.html)
- This part is not included in your textbook, but you need Chapter 1~5 [here](#).
- **Submission:**
  - `expr-compare.ss`
    - `expr-compare`
    - `test-expr-compare`
    - `test-expr-x`
    - `test-expr-y`
Hint Code & Other Helpers

- Sample grading script on Github: [https://github.com/CS131-TA-team/hw5-grading-script](https://github.com/CS131-TA-team/hw5-grading-script)

Note: since 2 quarters ago, past-year answers online are almost useless.
**HW5 - Your Goal**

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

How: By implementing the `expr-compare` function that takes two expressions and returns a new expression with similar parts combined.

`%` is a **boolean variable** that defines our choice on which program we want to execute. `#t` for the first one, `#f` for the second one.
Detailed requirements

- **test-expr-x** is an example of \( x \) expression to compare with \( y \) expression
- **test-expr-y** is an example of \( y \) expressions to compare with \( x \) expressions
- **expr-compare** is a procedure taking \( x \) and \( y \) as its arguments, and return a combined expression, as a list of symbols
- **test-expr-compare** when taken in \( x, y \) arguments, is supposed to judge if the result value of (1) expr-compare with \% set as \#t is exactly the same as directly calling \( x \), and (2) result from \( y \) is the same as setting \% as \#f.
HW5 - Understanding of combining

> (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 #t #f)
'%
Homework 5 with lambda and λ

- Either λ or `lambda` work equally in Scheme
- Lambda function in Python and anonymous function in OCaml
- This is the typical **anonymous** function.
  - `(lambda (args*) expr)`

```scheme
> (λ (a b c) (+ a b c))
#<procedure>
> ((λ (a b c) (+ a b c)) 1 2 3)
6
> ((lambda (a b c) (+ a b c)) 1 2 3)
6
> (lambda (a b c) (+ a b c))
#<procedure>
> (random-name (a b c) (+ a b c))
... random_name: undefined;
cannot reference an identifier before its definition
```
Clarifying on lambda-related combining rules:

You prefer to have a shorter summary expression, so you decide that the summary should use a λ expression in places where either one input expression used a lambda expression and the other used a λ expression (however, the summary should use lambda in places where both input expressions used lambda).

Examples (from @Xinyu):

(expr-compare '(cons a lambda) '(cons a λ))
; => '(cons a (if % lambda λ))

(expr-compare '(lambda (a) a) '(lambda (b) b))
; => '(lambda (a!b) a!b)

> (expr-compare '(lambda (a) a) '(λ (b) b))
; => '(λ (a!b) a!b)
When observing differences deeper (inner) parts 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))

> (expr-compare '(((lambda (a) (f a)) 1) '(((lambda (a) (g a)) 2))
'(((lambda (a) ((if % f g) a)) (if % 1 2))
HW5 - when not to combine

> (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))
HW5 - combine the alternative names

> (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! And to consider if there’s another function who could replace **lambda** (\(\lambda\)).

How to create new symbol automatically? Hint:

- `string->symbol`            (string->symbol "a")
- `symbol->string`            (symbol->string 'a)
- `string-append`             (string-append "abc" "def")
HW5 - an example case

> (expr-compare '(((lambda (lambda) (+ lambda if (f lambda))) 3)
  '(((lambda (if) (+ if if (f λ))) 3))
'(((lambda (lambda!if) (+ lambda!if (if % if lambda!if) (f (if % lambda!if λ)))) 3)

explanations:

> (expr-compare '(((lambda (lambda) (+ lambda if (f lambda))) 3)
  '(((lambda (if) (+ if if (f λ))) 3))
'(((lambda (lambda!if) (+ lambda!if (if % if lambda!if) (f (if % lambda!if λ)))) 3)
Scheme (Racket)
Why Racket

- Another **functional** programming language, but in many ways different from OCaml
- LISP (**LIS**t **P**rocessor): The second oldest high-level programming language. Pioneered many new concepts including:
  - Garbage Collection; Recursion; Dynamic Typing; **Program code as a data structure**
  - An “expression-oriented” language
- **Scheme**: **very minimal** dialect of LISP
  - Once popular in academia
  - CS161 AI
- **Racket**: standard Scheme + some additional features, very popular recently
Program as List could be dynamic!

> (define my-program ' (display "Hello, World!\n"))
> my-program
'(display "Hello, World!\n")
> (car my-program)
;display
> (cdr my-program)
'("Hello, World!\n")
> (car (cdr my-program))
"Hello, World!\n"
Racket: Installation

- **Installing Racket**
- Add to path if you cannot start the interactive environment from your command line
  - On mac, adding it to your path requires: *(filling in YOUR path instead, this is only an example)*
    
    ```
    echo $PATH
    PATH=$PATH:/Applications/Racket\ v7.4/bin
    ```

- Any text editor is fine, but DrRacket is specifically **recommended**.
  - To get started see this [tutorial](#)
Hello World in Racket without DrRacket

Option 1: from command line interactive terminal (enter by `racket`, exit by `(exit)`)

(display "Hello, world!\n")

Option 2: in a file named `hello.ss`:

(display "Hello, world!\n")

And then call it from command-line as:

`racket hello.ss`
Hello World in Racket with DrRacket

#lang racket
(display "Hello, world\n")
Basic Syntax in Racket: Procedure

- The Scheme system knows when it has an entire expression by matching double quotes and parentheses.

```
> + 1 2
#<procedure:+>
1
2

> (+ 1 2)
3
```
Basic Syntax in Racket: Variable Type

- **Comments**
  - ; comment a line
  - #| comment a block |

- **Numbers:** 1, 1/2, 0.5, 5e-1

- **Strings:** "Hello World\n"

- **Booleans:** #t, #f

- **Pair (restricted to 2 elements):**
  - (cons 1 2)
  - (cons 1 "hello")

- **List:**
  - (list 1 2 3)
  - (list 1 "hi")
  - '(1 2 3)
  - '(1 "hi")
  - (quote (1 2 3))

- **Type-checking:** `<type-name>?`
  - pair? is equivalent with cons?

- **Symbol:** but lists are not symbols!
  - 'a

```racket
> (number? 5)
#t
> (string? "My string")
#t
> (boolean? #t)
#t
> (pair? (cons 1 2))
#t
> (cons? (cons 1 2))
#t
> (list? (list 1 2 3 4))
#t
> 'a
'a
> (symbol? 'a)
#t
> (symbol? (list 1 2 3 4 5))
#f
> (symbol? (first '(hello world)))
#t
> (list hello world)
.error: undefined
  cannot reference an identifier before its definition
> (list 'hello 'world)
'(hello world)
```
Basic Syntax in Racket: If Statement

- Syntax for if statements:
  - `(if <condition> <then> <else>)`
  - Where `<condition>` is a statement with boolean value, `<then>` and `<else>` are operations to do when condition value is true or false respectively.

```racket
> (define (true_or_false_string bool)
  (if bool "true" "false"))
> true_or_false_string #t
#<procedure:true_or_false_string>
#t
> (true_or_false_string #t)
"true"
```
Basic Syntax in Racket: Conditions (cond)

- It checks the listed conditions one by one, until find the first satisfied condition (with #t value), then the following statements becomes the value we get from here.
- Similar with switch in C/C++, pattern-matching in OCaml.
- Not checking if it is exhaustive:
  - If all false, then it returns nothing

```racket
> (cond [#f "false 1"]
  [#t "true 2"]
  [#t "true 3"])
"true 2"

> (cond [#f "false 1"]
  [#f "false 2"]
  [#f "false 3"])

> (display (cond [#f "false 1"]
  [#f "false 2"]
  [#f "false 3"]))
#<void>
```
Basic Syntax in Racket: Function Calls

- In Scheme, function name **always** comes first in function calls
  - Even arithmetic operations !!!
- Similar with OCaml but
  - More strict in terms of the order
    - In OCaml we still have the option of putting some operations in the middle, here we don’t
  - Less strict on types (automatically convert)
- The parenthesis!
  - For procedures

> (= 5e-1 0.5)
#t

> (/ 1 2)
1/2

> (/ 1.0 3.0)
0.3333333333333333

> (* (/ 6 2) 5.0)
15.0

> (display "Hello World\n")
Hello World

> (+ 1 2)
3

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

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

>
Scheme expression is Preorder Traversal of AST

- What it means in terms of traversals on the abstract syntax tree?
  - Inorder (Left, Root, Right)
  - Preorder (Root, Left, Right)
  - Postorder (Left, Right, Root)
Basic Syntax in Racket: Definitions

- Defining variables and functions have a similar syntax.
- This is also similar with what you’ve experienced in OCaml
  - `let ... = ... ;`

```racket
> (define pi 3.1415926)
> pi
3.1415926
> (define (print_name name_to_be_printed)
   (display (string-append "Hello, " name_to_be_printed)))
> (print_name "Patricia")
Hello, Patricia
> 
```
What about `let` in Racket?

- It is similar with the use of the OCaml syntax: `let ... in ... ;;`

```racket
> (let ([x 5]) (* x 2))
10
> (let ([x 5] [y 6]) (+ x y))
11
```
Basic Syntax in Racket: Local bindings and Identifier

- **Very** liberal in naming rules
- Only forbidden a few characters in the name: ( ) [ ] { } " , ' ` ; # | \ 

```racket
> (define (Crazy-nAme+yes/no? bool)
  (if bool (display "yes") (display "no")))
> (Crazy-nAme+yes/no? #t)
yes
```
Example of Variable-Binding Usage

> (define (print_name)
  (let ([greeting "Hello, "]
       [name "Patricia"])
    (display (string-append greeting name))))

> (print_name)
Hello, Patricia

(* equivalent with the following OCaml code *)

# let print_name =
  let greeting = "Hello, " and name = "Patricia" in
  greeting ^ name;;
val print_name : string = "Hello, Patricia"
Basic Syntax in Racket: Comparison of Numbers

- `=`, `<`, `>`, `<=`, `>=`
- They could be applied to **multiple** values, instead of only binary operation

```racket
> (< 1 2)
#t
> (< 1 2 3)
#t
> (< 1 3 2)
#f
```
Basic Syntax in Racket: Comparison

- Equality
  - (= 1 2) for numbers
  - (equal? (list 1 2 3) (list 1 2 3)) for other values (checked recursively)
  - (eq? a a) for object references

> (= 1 2)
#f
> (= "hello" "hi")
Type error expected here
> (equal? "hello" "hi")
#f
> (equal? (list 1 2 3) (list 1 2 3))
#t
> (define a (list 1 2 3))
> (define b (list 1 2 3))
> (eq? a a)
#t
> (eq? a b)
#f
Basic Syntax in Racket: Short-circuit evaluation

- Almost **exactly** the same with Python behavior, **except** the definition of false
  - in Python, empty / 0 are all false
  - In Scheme, only false itself is false
- Rule 1: **EVERYTHING** that is not #f are regarded as #t
- Rule 2: **and/or** execute instructions until the expression has been evaluated
  - Holds the last evaluated value as the expression value.

```racket
> (or #f 2 3)
2
> (and 2 3)
3
> (and 2 3 #f 6)
#f
```
An Exercise: How to implement do-twice?

Test case and expected output:

```plaintext
> (do-twice (lambda (x) (* x 2)) 2)
8
```
List - from C/C++ to OCaml to Racket

- Why always “head + tail (the rest of the list)” in OCaml?
  - They are linked lists
  - Each element is a node with reference to its own value and reference to the rest of the list

- CAR = Content of Address Register
  - In other words, head

- CDR = Content of Decrement Register
  - In other words, tail

- Why list is important?
  - Racket → Scheme → LIST Processor

- Documentation
Linked-List and Pairs in Racket

- In Racket, **ALL** lists are pairs
  - For the obvious reason: head and tail
- Pairs are **NOT** necessarily lists

```racket
> (pair? (list 1 2))
#t
> (pair? (list 1 2 3))
#t
> (list? (list* 1 2))
#f
> (list? (cons 1 2))
#f
> (list? (cons 1 (list 2)))
#t
```
List - Construction

- Constructing an ordinary list
  - '(<v1>  <v2>  …  <vn>)
  - (quote (<v1>  <v2>  …  <vn>))
  - (list <v1> <v2> … <vn>)
  - (list* <v1> <v2> … <tail>)
    - It is a list only when <tail> is a list
  - (cons <head> <tail>)
    - It is a list only when <tail> is a list
- Empty list
  - '()
  - empty
  - (list)
- Build a list based on the list 0, 1, ... n - 1
  - build-list
- For more see the Racket documentation

---

```scheme
> '()
'(())
> empty
'()
> (list)
'(())
> (cons 1 empty)
'(1)
> (list 1 2 3 4)
'(2 3 4 4)
> (quote (1 2 3))
'(1 2 3)
> (pair? (list* 1 2))
#t
> (list? (list* 1 2))
#f
> (list* 1 2 3 4 5 empty)
'(1 2 3 4 5)
> (values 1)
1
> (values (list 1 2 3))
'(1 2 3)
> (values 1 2 3)
1
2
3
> (build-list 5 values)
'(0 1 2 3 4)
> (build-list 5 (lambda (x) (* x x)))
'(0 1 4 9 16)
```
Pair Operations (also apply to Lists)

- You can apply these operations to both lists and pairs
- Basically the same with hd (head) and tl (tail) in OCaml

> (define a-list (list 1 2 3))
> (define a-pair (cons 1 2))
> (car a-list)
  1
> (car a-pair)
  1
> (cdr a-list)
  '(2 3)
> (cdr a-pair)
  2
Pair Accessor Shorthands

Generally looks like (describing by hw2-style grammar):

Starting symbol: Shorthand
Shorthand → [[c, X, r]]
  X → [[x, X], [x]]
  x → [[a], [d]]

> (caar '((1 2) 3 4))
1
> (cadr '((1 2) 3 4))
3
> (car (cdr '((1 2) 3 4)))
3
List Functions

- The function `first` and `rest` are exactly the same with `car` and `cdr`, but restricted to be applied on lists only!
- Besides, we even have second, third, …
  - See the documentation
- Check if a list is empty by calling `empty?`
  - It works syntactically okay on pairs, but semantically meaningless to do so.
- Check the length of the list by
  - `(length lst)`
- Get the element at position `pos` by
  - `(list-ref lst pos)`

```scheme
> (define a-list (list 1 2 3))
> (define a-pair (cons 1 2))
> (first a-list)
1
> (rest a-list)
'(2 3)
> (second a-list)
2
> (third a-list)
3
> (empty? (cons 1 2))
#f
> (empty? (cons empty empty))
#f
> (length (list 1 2 3))
3
> (list-ref (list 1 2 3 4 5) 1)
2
```
HOF of list: **iterations** and **filter** and **search**

- Most frequently used members of them include:
  - map, filter,
- For more please refer to the documentation (the links are attached above)

```lisp
> (map (lambda (x) (+ x 1)) '(1 2 3))
'(2 3 4)
> (filter (lambda (x) (> x 2)) '(1 2 3 4))
'(3 4)
> (foldl (lambda (a b) (+ a b)) 0 '(1 2 3 4))
10
> (foldr (lambda (a b) (+ a b)) 0 '(1 2 3 4))
10
> (foldr (lambda (v l) (cons (add1 v) l)) '() '(1 2 3 4))
'(2 3 4 5)
> (foldl (lambda (v l) (cons (add1 v) l)) '() '(1 2 3 4))
'(5 4 3 2)
> (sort '(5 4 3 2 1) <)
'(1 2 3 4 5)
> (length '(1 2 3 4 5))
5
> (reverse '(1 2 3 4 5))
'(5 4 3 2 1)
```
Eval and Program as List

- 1. Every **procedure** is within parentheses
- 2. List could be constructed as: '(......) or (quote (......))
- 3. List content could be **undefined** symbols
- We can easily **FORCE** any expression to be interpreted as a list of symbols, not a function call.
- Quite flexible this way.

```
> '(a b c)
'(a b c)
> (define symbol_list '(a b c))
> (eval symbol_list)
. . ./Applications/Racket
cannot reference an identifier **before its definition**
> (symbol_list)
. . application: not a procedure;
expected a procedure that can be applied to arguments
given: '(a b c)
arguments...: [none]
> (define a +)
> (define b 1)
> (define c 2)
> (eval symbol_list)
3

> (define my-program '(display "Hello, World!\n"))
> my-program
'(display "Hello, World!\n")
> (eval my-program)
Hello, World!
```
 Eval and Program as List: Namespace

- In your **interpreter**, eval works *with* / *without* defining a namespace
- In file we **must specify namespace**.
  - Otherwise throws the error () like: **display**: unbound identifier; also, no #app syntax transformer is bound at: display

```racket
; (eval my-program)
(define my-program '(display "Hello, World!\n"))
; namespace
; run the file as racket hello.ss
; or click run in DrRacket
(define ns (make-base-namespace))
(eval my-program ns)
```
Quasiquote and Unquote (guide) (reference)

- **Unquote has to be within quasiquote.**
  - Quasiquote could be expressed as:
    - (quasiquote ( ... ))
    - `( ... )`
  - Unquote is either way of these:
    - (unquote ( ... ))
    - ,( ... )

- Their meaning is: within the whole quasiquote contents list, the unquote parts’ symbols should be combined and evaluated.

```scheme
> (quasiquote (1 2 (unquote (+ 1 2))
                (unquote (- 5 1))))
'(1 2 3 4)
> `(1 2 ,(+ 1 2) ,(- 5 1))
'(1 2 3 4)
> `(my-function ,(+ 1 2))
'(my-function 3)
> `,(,+ 1 2))
'(3)
```
In addition: macros

- quote, unquote, quasiquote, define, etc. they are often referred to as “special forms”
- What makes them so different is that, they are built-in macros, instead of functions.
- You can also define your own macros
  - [https://docs.racket-lang.org/reference/define.html](https://docs.racket-lang.org/reference/define.html)
  - [https://docs.racket-lang.org/guide/pattern-macros.html](https://docs.racket-lang.org/guide/pattern-macros.html)
  - etc.
In addition: **dictionary**

- Dictionary is probably useful for recording the variable names.
- Using `dict-set` instead of `dict-set`!
  - Why? (Hint: mutable...)
- **Not really mandatory** to use the dictionary. Just telling you you can use it. ()

```lisp
> (dict-set #hash() 'a "apple")
'#hash((a . "apple"))
> (dict-set #hash((a . "apple")
  (b . "beer")) 'b "banana")
'#hash((a . "apple") (b . "banana"))
> (dict-set '() 'a "apple")
'((a . "apple"))
> (dict-set '((a . "apple") (b . "beer")) 'b "banana")
'((a . "apple") (b . "banana"))
```
Scheme & Prolog in Exams
Covered in your final

- Prolog is in Webber, Scheme is not
- Please go through Scheme textbook Chapter 1~5
- Scheme will be in your final (almost for sure)
- Prolog / Scheme questions in your final exam are most likely testing you on:
  - Finding significant problems in the code (in prolog, typically caused by cut/negation, please check Negation as failure for more details)
  - Fixing the bugs in some simple code
  - Rewriting / Simplifying existing code
  - Implementing some simple functions
  - Combining with other concepts, e.g. grammar, scope, cost, parameters (call by reference, name, etc.), memory, possibility of writing language-B interpreter in language A, comparing the differences among the languages, etc.

  E.g. Given an extension form of the standard Prolog / Scheme, then ask: please write down the grammar of this extended expression in EBNF form; rewrite standard Prolog / Scheme code; looking for ambiguity, etc.
Simple Prolog (not real exam question)

Let \( z \) represent zero. If \( x \) represents an integer \( n \), then \( \text{twice}(x) \) represents \( n \times 2 \), \( \text{inc}(x) \) represents \( n + 1 \). Assume we already defined \( \text{sum}(x, y, z) \) which is true when \( z \) is the sum of \( x \) and \( y \).

Can you fix the following function?

\[
\text{square}(x, y) \text{ should be true if } y \text{ is the square of } x
\]

\[
\text{square}(z, z).
\]

\[
\text{square}(\text{inc}(M), \text{inc}(N)) :\text{-} \text{square}(M, \text{Msquare}),
\]

\[
\text{sum}(\text{inc}(\text{Msquare}), \text{twice}(M), N).
\]
Simple Prolog Answer Hint

● Need fixation in logic
● Need fixation in base cases (*fail*)

Extension:

Can \(+ (\text{square}(X, Y))\) express two numbers X, Y where Y is not the square of X?

● Consider what it is denying.
A Tricky Part of Prolog: \textbf{cut \& negation} ($\backslash+$ for not)

- Prolog enumerate all possible values satisfying a rule in DFS, and then move on to the next rule, etc. When it
- Try using the \texttt{visualizer} to help you understand backtracking
- Cut stops the backtracking
- Removing cut, we could easily get into trouble with efficiency, even run into an infinite loop.
- Adding cut, we could also have problem, like fail to find the answers when there should be some.
- Naive examples of cut:
  
  \begin{verbatim}
  \end{verbatim}
A Tricky Part of Scheme: Function v.s. Special Form

- Function, or special form (macro)?
  - `lambda`
  - `define`
  - `quote`
  - `if`
  - `+`
  - `length`

- If we introduce an extended-Scheme, who has a different version (e.g. we change the order of the whole expression) of the special forms, and now we ask you to rewrite standard-Scheme code into extended-Scheme version:
  - **Should you apply the changes to functions?**