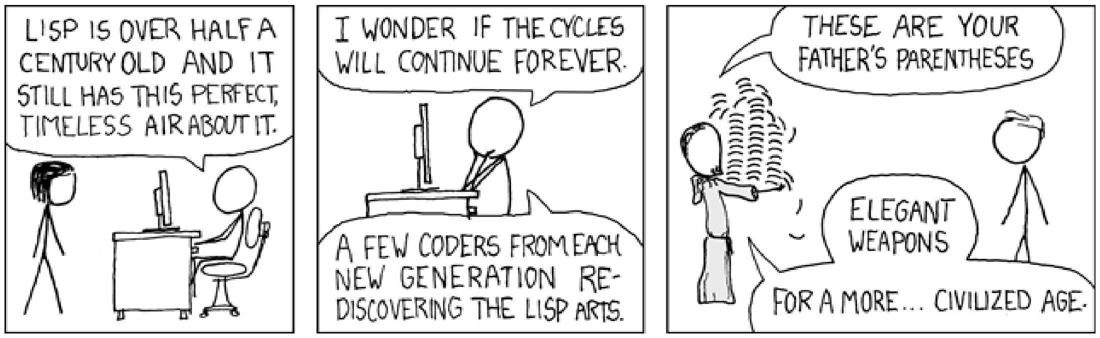
CSC172 LAB 13

THE SCHEME OF THINGS
(a bit of racket)


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

1 Introduction

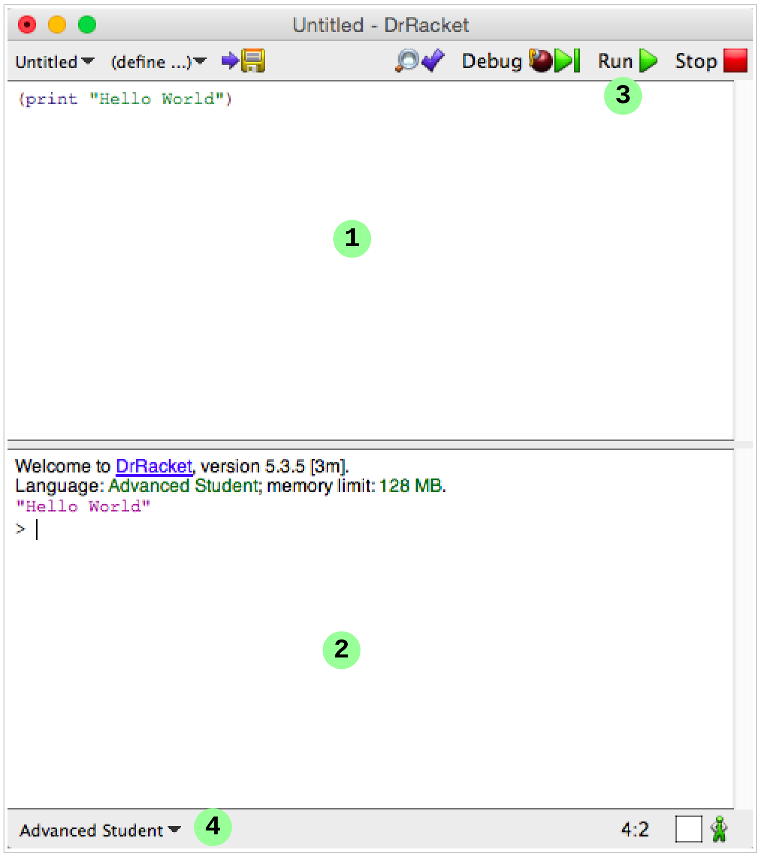
The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming .

Every student must hand in their own work, but every student must list the name of their lab partner if any on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

Throughout CSC172, we have examined the concept of a "list" in many forms. Primarily, we have looked at using lists as a way of storing data. However, it turns out that there are whole computer languages based on the concept of the list. The Scheme programming language is a dialect of the LISP programming language (according to unsubstantiated internet legend, "**LISP**" is an acronym which stands for "**L**ots of **I**nsipid **S**purious **P**arentheses"). We use the LISP language in CSC172 because, perhaps more than any other language, LISP teaches us to think recursively. Languages like C, Pascal, and Fortran are "procedural" languages. They teach us to think of processes in terms of sequences of operations on data. Languages like Java and C++ are "object oriented" languages and help us to think of processes in terms of messages passed between objects. Lisp is the archetype of the "functional" languages and introduces us to a different paradigm for the conceptualization of abstract process knowledge. Many programmers consider lisp a uniquely beautiful language because of it's conceptual purity and brevity of syntax.

In order to complete this lab, you will need to download and install a free Scheme interpreter on your computer. Lab computers should already have this installed, but if not, the same instructions apply. You can download the appropriate version of Scheme for your computer at http://download.racket-lang.org/. Install the program using its default settings. We'll be using the newly installed DrRacket program to write and run our Scheme code. A general overview of the interface is shown below.



```
(print "Hello World")
```

```
Welcome to DrRacket, version 5.3.5 [3m].
Language: Advanced Student; memory limit: 128 MB.
"Hello World"
> 
```

1. You should type your code in the definitions window
2. Output from your program appears in the interaction window. You can also type and run commands directly from this window for debugging and testing features.
3. The run button executes the code in the definitions window (1) and outputs the results in the interaction window (2). Any previous interactions will be erased when you click run.
4. This is where you select the language you're using. You should use Advanced Student.

2 A Bit of Racket

The purpose of this lab is to introduce you to the Advanced Student dialect of Scheme using DrRacket.

1. Scheme can be thought of as a "*prefix calculator on steroids with parentheses*". The best way to learn Scheme is to start interacting with it. So, try typing the following 21 lines, before you hit "Enter" at the end of each line, try to guess what the response will be based on the previous results.

```
486
( + 1 3 )
( - 7 4 )
( - 4 7 )
( * 3 5 )
( /  35  7)
( /  7 35)
; Starting to get the hang of this?
; You can insert comments with the semicolon
( + 1 2 3 4)
( * 3 5 7 )
( / 7 5 3)
( *   (  + 3 4) ( + 10  3))
; single line
(+ ( * 3 ( + ( * 2 4) (+ 3 5 ))) (+ ( - 10 7 ) 6 ))
; or multi line
(+ ( * 3
     ( +  ( * 2 4 )
          (+ 3 5)))
      (+  (- 10 7)
     6 ))
(< 3 0)
(> 3 0)
(and (> 3 0) (< 3 0))
(or (> 3 0) (< 3 0))
(print "Hello, world")
```

2. Ok, not bad. Perhaps a lot of work if we only were getting a prefix calculator. Have patience, Young Padawan. What we've got here is a whole language. In fact, we can define variables. Try the following.

```
(define length2 13)
length2
(* 3 length2)
(* 3 width) ; this will draw an error
; don't worry it's just an example
; you have to define variables before you use them
(define width 17)
(* length2 width)
(define area (* width length2))
area
(define my_pi 3.14159 ) ; pi is already defined in the language
(define radius 11)
(* my_pi (* radius radius ))
(define circumference (* 2 my_pi radius))
circumference
```

3. So, that's a little more power and flexibility, don't you think? Variables are good. We can also define functions so that we can save processes for reuse. The general form of function defintion is: ***(define** ((name) (parameters)) (body))*, but as is with many things in this language, it's often best learned by a few examples. Try the following:

```
(define (square x) (* x x))
(square 10)
(square length2)
(+ (square 5) (square 4))
(square (square 3))
(define (sum-of-squares x y) (+ (square x) (square y)))
(sum-of-squares 3 4)
```

4. A programming language wouldn't be much good with out conditionals. The general form of function definition is: ***(if** (predicate) (consequent) (alternative))*, but as is with many things in this language, it's often best learned by a few examples. Try the following:

```
(if (< width 3) (print "short") (print "long"))
(define width2 2)
(if (< width2 3) (print "short") (print "long"))
(if (> length2 2) 7 11)
; so, now we can combine conditionals with function definition
; to write some more complex functions
;
; absolute value
(define (absolute-value x)
     (if (< x 0)
          (- x)
          x
     )
)
(absolute-value -3)
(absolute-value (* 7 -3))
;
; maximum
(define (maximum x y) (if (> x y) x y))
(maximum 3 7)
;
; and of course, recursion is quite natural
(define (factorial n)
     (if (< n 1 ) 1 (* n (factorial (- n 1 )))))
(factorial 3)
(factorial 7)
```

5. Using the skills you've learned thus far, write your own recursive function that computes the greatest common divisor of two numbers. Call the function a few times to demonstrate that it works.
6. So, not a bad little language, so far. We will be doing more with Scheme in the coming labs. For now save your definitions to a file named Lab11.rkt and save your interactions to a file named OUTPUT.txt. These options are available on the File menu. Edit the saved text by adding your name and contact information in comments at the top of the files and then hand in the files in to complete the lab.

3 Hand In

Hand in the source code from this lab at the appropriate location on the blackboard system at my.rochester.edu. You should hand in a single compressed/archived (i.e. “zipped” file that contains the following.)

1. A plain text file named README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.). And one sentence explaining the contents of all the other files you hand in.
2. The Racket file containing the definitions you wrote in the lab with author information commented out at the top.
3. A plain text file named OUTPUT that includes your interactions from DrRacket from part 5.

4 Grading

15% each for parts 1-4

30% for part 5

10% README and OUTPUT files