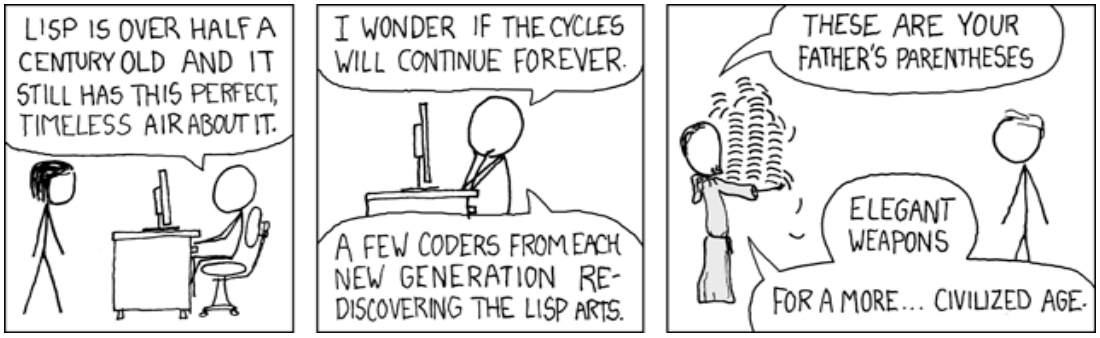
CSC172 LAB



THE SCHEME OF THINGS
(a bit of a racket)

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 his own work, but every student must list the name of the 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.

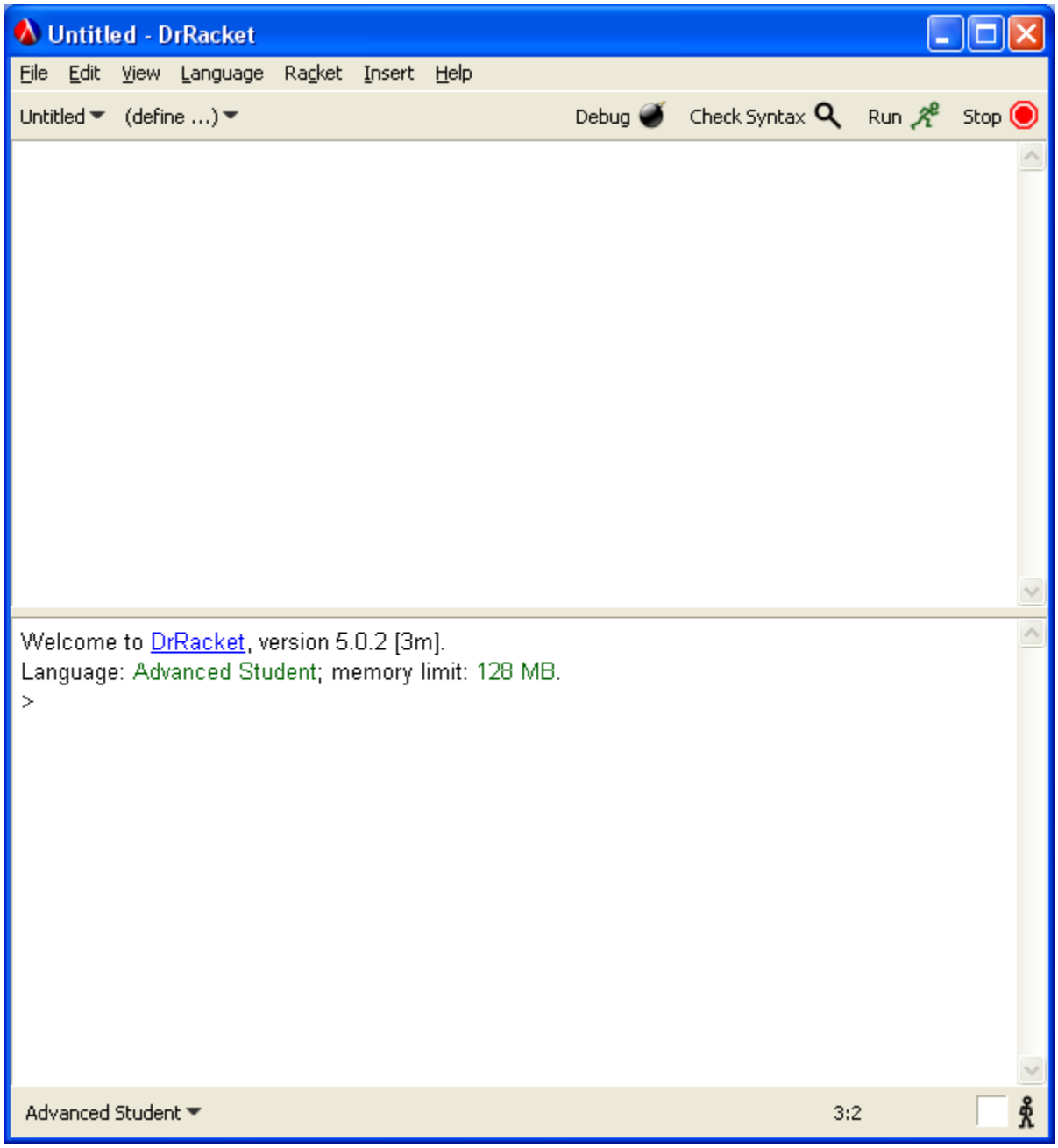
The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

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 its conceptual purity and brevity of syntax.

1. Begin this lab by implementing by downloading and running a free Scheme interpreter on your lab PC. This is a fairly simple operation. Follow the ten simple steps. They should take about 5 minutes.
    1. Type "drscheme" into google
    2. navagate to http://racket-lang.org/ (It should be the first item)
    3. Click on "Download DrRacket"
        1. Select Windows (95 and up) x86 assuming that you are in the Gavett lab.
        2. Click Download
        3. Pick any mirror site
    4. Save the .exe installer to your disk
    5. Click “open” to run the installer
        1. Click “Next” during the install procedure
        2. For the lab machines, use D:\temp\ as an install location rather than the C: drive
        3. Click “Install”
        4. Set the “run” checkbox
        5. Click “Finish”
    6. Once DrRacket starts use the menus : Language -> Choose Language
        1. Select “Teaching Languages -> Advanced Student”
    7. Click the "Run" button at the top to load your language selection
    8. Type stuff in. (The 5 remaining steps of this lab – see below)

9. When you have finished the lab use the menu File -> “Save Other” -> “Save Interactions As Text ” to save your work to a text file.
10. Hand in the text file (Edit it to include your standard identification information)

2. At this point, you should be looking at something like the following picture on your computer.



You interact with this by typing things into the lower pane. This is the scheme language interpreter. Try thinking of scheme as a kind of “*prefix calculator on steroids with parenthesis*”. This language may best be learned by doing. 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”)
```

3. OK, not bad. Perhaps a lot of work if we only were getting a prefix calculator. Patience, Padawan, patience. What we have 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 pi 3.14159 )
(define radius 11)
(* pi (* radius radius ))
(define circumference (* 2 pi radius))
circumference
```

4. So, that's a little more power and flexibility, don't you think? So variables are good. We can also define functions so that we can save processes for reuse. The general form of function definition 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 length )
(+ ( square 5 ) (square 4))
(square (square 3 ))
(define (sum-of-squares x y) (+ (square x) (square y))
(sum-of-squares 3 4)
```

5. 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 width 2 )
(if (< width 3) (print “short”) (print “long”))
(if (> length 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)
```

6. So, not a bad little language, so far. We will be doing more with scheme in the coming labs. For now save your work with the File -> “Save Other” -> “Save Interactions As Text ” menu sequence. Edit the saved text by adding your name and contact information and then hand in the file to complete the lab.

2 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 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.).
2. The source code files representing the work accomplished for this lab. All source code files should contain author and partner identification in the comments at the top of the file.

3. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line.

3 Grading

172/grading.html
Each section (1-6) accounts for 15% of the lab grade (total 90%)

(README file counts for 10%)