Comp200--Structure and Interpretation of Computer Programs

Project 0

- **Due To: 09.10.2015**
- Submission: Please name your homework file to project0.scm and upload it to DriveF@VOL/COURSES/UGRADS/COMP200/HOMEWORK/[USERNAME]/project0.scm
  - Projects that are uploaded incorrectly will **not be graded**.
  - Ex:
    - DriveF@VOL/COURSES/UGRADS/COMP200/HOMEWORK/**myatbaz/project0.scm**
- Reading: Go through the syllabus and the steps outlined in the "DO NOT PANIC!" handout. This should familiarize you with the labs and the scheme environment.

The purpose of Project 0 is to familiarize you with the Scheme programming environment and the resources available to you. The format of this project is different from the more substantive ones you will see later in the term. Those projects focus on extended experience in creating and testing code, and integrating such code with existing computational frameworks. As a consequence, those projects will take a significant amount of time to complete. This project is really just a warm up, and is intended to get you up to speed on the mechanics of using Scheme and the course resources. We expect that this project should not take much time, and while we understand that some of the things we ask you to do seem fairly obvious, we want to make sure you get the mechanics down right. Don't worry; the later projects and problem sets will involve much more extensive thinking and coding!

1. Getting Started

The purpose of this section is to get you started using Scheme as quickly as possible. Start by looking at the handouts available on the course website. If you have a PC capable of running Scheme, we suggest that you install it there, since it will be convenient for you to work at home. ENG-B19 is the best places to work. You can also contact TA using the comp200help@ku.edu.tr email.

Remember, whether you're working with your machine or the lab's, the source of all useful information is the Comp200 web page.

Starting Scheme

The first thing to do is to get an implementation of Scheme and start using it:

- *In ENG-B19*: A good way to get started learning Scheme is to do this project in the Lab. You can login to the system (Windows) with your Novell username and password and you can immediately start DrRacket from “All Programs” .
- If you have problems logging in contact to CIT. After logging in, you can prefer opening the command-line Racket interpreter or you can directly use the core compiler

of DrRacket.

- *On your home computer...* Follow the instructions in the "Working from Home" handout on the course web page for downloading and installing Scheme (either Linux or Windows versions), and/or connecting to the linuxlab machine using ssh.

Getting Started with Rocket

On Windows, you can start DrRacket from the **Racket** entry in the Start menu. In Windows Vista or newer, you can just type DrRacket. You can also run it from its folder, which you can find in **Program Files → Racket → DrRacket**.

On Mac OS X, double click on the **DrRacket** icon. It is probably in a **Racket** folder that you dragged into your **Applications** folder. If you want to use command-line tools, instead, Racket executables are in the "bin" directory of the **Racket** folder (and if you want to set your PATH environment variable, you'll need to do that manually).

On Unix (including Linux), the drracket executable can be run directly from the command-line if it is in your path, which is probably the case if you chose a Unix-style distribution when installing. Otherwise, navigate to the directory where the Racket distribution is installed, and the drracket executable will be in the "bin" subdirectory.

Depending on how you look at it, **Racket** is

- a *programming language*—a dialect of Lisp and a descendant of Scheme;
- a family of programming languages—variants of Racket, and more; or
- a set of tools—for using a family of programming languages.

Where there is no room for confusion, we use simply Racket.

- Racket's main tools are
- racket, the core compiler, interpreter, and run-time system;
- DrRacket, the programming environment; and
- raco, a command-line tool for executing Racket commands that install packages, build libraries, and more.

Most likely, you'll want to explore the Racket language using DrRacket, especially at the beginning. If you prefer, you can also work with the command-line racket interpreter and your favorite text editor; see also Command-Line Tools and Your Editor of Choice. The rest of this guide presents the language mostly independent of your choice of editor.

If you're using DrRacket, you'll need to choose the proper language, because DrRacket accommodates many different variants of Racket, as well as other languages. Assuming that you've never used DrRacket before, start it up, type the line

```
#lang racket
```

in DrRacket's top text area, and then click the Run button that's above the text area. DrRacket then understands that you mean to work in the normal variant of Racket (as opposed to the

smaller racket/base or many other possibilities).

If you've used DrRacket before with something other than a program that starts #lang, DrRacket will remember the last language that you used, instead of inferring the language from the #lang line. In that case, use the Language|Choose Language... menu item. In the dialog that appears, select the first item, which tells DrRacket to use the language that is declared in a source program via #lang. Put the #lang line above in the top text area, still.

Scheme Expression Evaluation

Scheme programming consists of writing and evaluating *expressions*. The simplest expressions are things like numbers. More complex arithmetic expressions consist of the name of an arithmetic operator (things like +, -, *) followed by one or more other expressions, all of this enclosed in parentheses. So the Scheme expression for adding 3 and 4 is `(+ 3 4)` rather than 3 + 4.

An expression may be typed on a single line or on several lines; the Scheme interpreter ignores redundant spaces and line breaks. It is to your advantage to format your work so that you (and others) can read it easily. It is also helpful in detecting errors introduced by incorrectly placed parentheses. For example, the two expressions

```
(* 5 (+ 2 (/ 4 2) (/ 8 3)))

(* 5 (+ 2 (/ 4 2)) (/ 8 3))
```

look deceptively similar but have different values. Properly indented, however, the difference is obvious.

```
(* 5
   (+ 2
      (/ 4 2)
      (/ 8 3)))

(* 5
   (+ 2
      (/ 4 2))
   (/ 8 3))
```

DrRocket's editor provides a good environment to "pretty-print" your code, indenting lines to reflect the inherent structure of the Scheme expressions (see Section B.2.1 of the *Introduction to the Scheme Programming Environment* manual).

While the Scheme interpreter ignores redundant spaces and carriage returns, it does not ignore redundant parentheses! Try evaluating

```
((+ 3 4))
```

Scheme should respond with a message akin to the following:

; application: not a procedure;
; expected a procedure that can be applied to arguments
; given: 7
; arguments...: [none]

Indenting Code

Indentation is very important when you are programming with Scheme due the fact that Scheme is a paranthesis based language and we highly recommend you to indent your codes during all the projects.

DrRacket automatically indents according to the standard style when you type Enter in a program or REPL expression. For example, if you hit Enter after typing *(define (greet name)*, then DrRacket automatically inserts two spaces for the next line. If you change a region of code, you can select it in DrRacket and hit Tab, and DrRacket will re-indent the code (without inserting any line breaks). Editors like Emacs offer a Racket or Scheme mode with similar indentation support.

Re-indenting not only makes the code easier to read, it gives you extra feedback that your parentheses match in the way that you intended. For example, if you leave out a closing parenthesis after the last argument to a function, automatic indentation starts the next line under the first argument, instead of under the define keyword:

```
(define (halfbake flavor
          (string-append flavor " creme brulee")))
```

In this case, indentation helps highlight the mistake. In other cases, where the indentation may be normal while an open parenthesis has no matching close parenthesis, both racket and DrRacket use the source's indentation to suggest where a parenthesis might be missing.

Debugging with DrRocket

Racket's built-in debugging support is limited to context (i.e., "stack trace") information that is printed with an exception. In some cases, disabling the JIT compiler can affect context information. The errortrace library supports more consistent (independent of the JIT compiler) and precise context information. The racket/trace library provides simple tracing support. Finally, the DrRacket programming environment provides much more debugging support.

Tracing

The bindings documented in this section are provided by the racket/trace library, not racket/base or racket. So, in order to be able to use the trace library you need to type:
(require racket/trace)
Syntax for using trace function: *(trace id ...)*

Each *id* must be bound to a procedure in the environment of the trace expression. Each *id*is set!ed to a new procedure that traces procedure calls and returns by printing the arguments and results of the call via current-trace-notify. If multiple values are returned, each value is displayed starting on a separate line.

When traced procedures invoke each other, nested invocations are shown by printing a nesting prefix. If the nesting depth grows to ten and beyond, a number is printed to show the actual nesting depth.

The `trace` form can be used on an identifier that is already traced. In this case, assuming that the variable's value has not been changed, `trace` has no effect. If the variable has been changed to a different procedure, then a new trace is installed.

Tracing respects tail calls to preserve loops, but its effect may be visible through continuation marks. When a call to a traced procedure occurs in tail position with respect to a previous traced call, then the tailness of the call is preserved (and the result of the call is not printed for the tail call, because the same result will be printed for an enclosing call). Otherwise, however, the body of a traced procedure is not evaluated in tail position with respect to a call to the procedure.

2. Your Turn

There are several things that you need to do for this part: evaluate some simple Scheme expressions, use the provided tools to manipulate these expressions, and answer some documentation and administrative questions.

2.1 Preparing your project material for submission

- Please follow the instructions in the "Project Submission Instructions" handout to get your project files.
- Start DrRocket
- Download and open the file project0.scm in Rocket.
- Insert your answers for the following parts.
- Do not forget to save your file project0.scm periodically .

2.2 Expressions to Evaluate

Below is a sequence of Scheme expressions. Can you predict what the value of each expression would be when evaluated? Go ahead and type in and evaluate each expression in the order it is presented. **Copy these expressions into 2.2. part in your "project0.scm" file that you downloaded from course website.**

In your submission, include these expressions and the resulting values of their evaluation. Also, include some comments that document your work. Perform the same procedure as in the example below in sections 2.3 and 2.4. This is a good habit to get into NOW!
For example:

```
;;;
;;; The following test cases explore the evaluation of simple expressions.
;;;

(* 8 9)
;Value: 72
```

```
-37

(* 8 9)

(> 10 9.7)

(- (if (> 3 4)
       7
       10)
   (/ 16 10))

(* (- 25 10)
   (+ 6 3))

+

(define double (lambda (x) (* 2 x)))

double

(define c 4)

c

(double c)

c

(double (double (+ c 5)))

(define times-2 double)

(times-2 c)

(define d c)

(= c d)

(cond ((>= c 2) d)
      ((= c (- d 5)) (+ c d))
      (else (abs (- c d))))
```

2.3 Pretty printing

As you begin to write more complicated code, being able to read it becomes very important. To start building good habits, type the following simple expression into Scheme :

```
(define abs C-j (lambda (a) C-j (if (> a 0) C-j a C-j (- a))))
```

Show a copy of how this actually appears if it is written in an indented manner by simply typing enter in the points where C-j is written in the expression. When do we use tab in indentation, what is the difference.

2.4. Tracing

Have a look at the trace property of Racket and the example given. Try this yourself:

```
(define (fun x)
  (if zero? x)
    1
    (* x (fun (- x 1)))))
(trace fun)
(fun 4)
```

Show the result of trace example above. Do not forget to include tracing library that is explained in Getting Started->Tracing part above.

2.5 Documentation and Administrative Questions

Explore the Comp200 and DrRocket webpages to find the answers to the following questions:

1. **Explain for what purposes can you use debug and trace options in DrRacket? By trying the debug in Racket, explain how does the stepping in debug mode works.**

2. **According to the *MIT Scheme Reference*, which of the words, in the scheme expressions you evaluated as part of section 2.2 above, are ``special forms''? (Check the "WWW Links" section of the course website)**

3. **What does the MIT Scheme Reference Manual say about treatment of upper and lower case in expressions?**

Debugging can be frustrating. The debugging tools are your friends. Call on them regularly!

Congratulations! You have reached the end of Project 0!