Introduction to Computer Science (CMSC151)
Lab 1: Introduction to DrScheme - Simple Programs
Due: Friday, October 2nd 2009, 10:00pm

0. Introduction

In this lab, you will learn how to use DrScheme and design a few simple programs.

0.1. Setup

Course website and mailing list

The course website is available at http://www.classes.cs.uchicago.edu/archive/2009/fall/15100-1/index.html
If you have not yet had a chance to, please subscribe to the class mailing list at
https://mailman.cs.uchicago.edu/mailman/listinfo/cmsc15100

Install DrScheme

You can install DrScheme on your machine (recommended) by following the instructions from the PLT Scheme website or use the installed version available on cs machines in the lab.

Setup your handin account

You can set up your submission account by following the steps given on the course website (here).

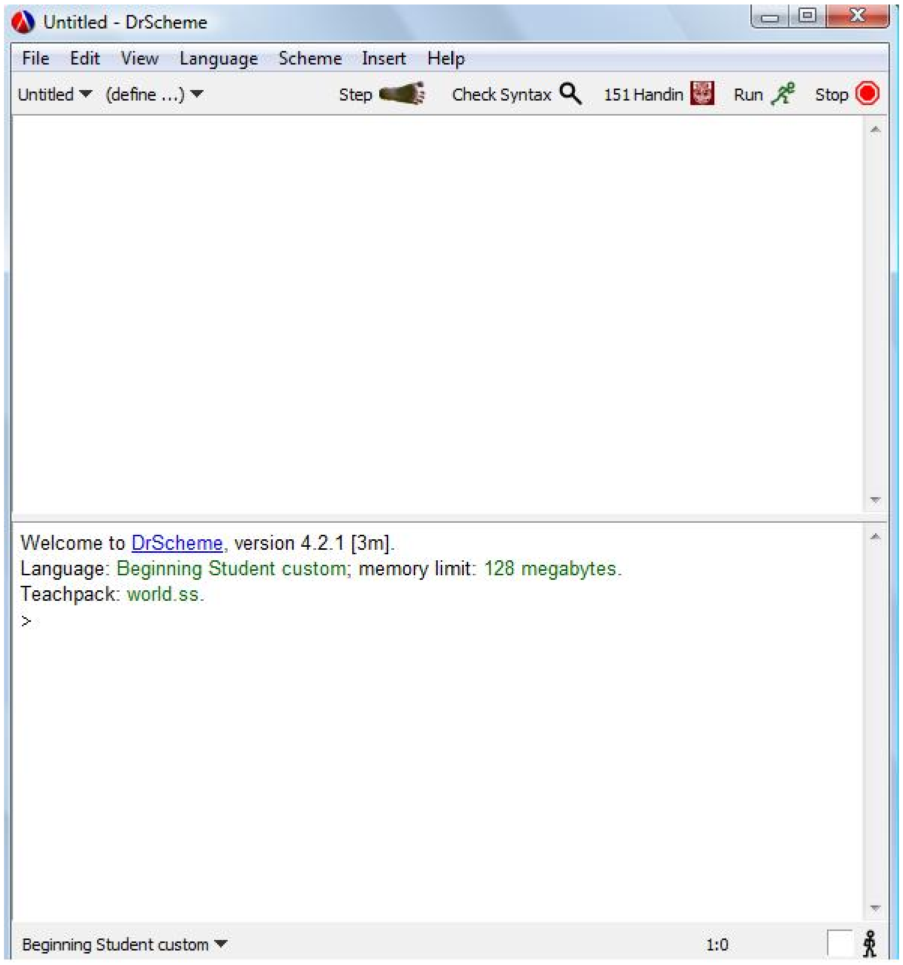
1. Choose the File → Install .plt file menu item in DrScheme and paste in this url:
   http://www.classes.cs.uchicago.edu/archive/2009/fall/15100-1/files/uc-cmsc15100.plt
2. Restart DrScheme. A handin button should now appear in the top right hand corner of the DrScheme window.
3. Create an account by using the File → Manage CMSC15100 Handin Account menu item.
4. Use the Manage CMSC 15100 Handin Account... button to hand in your work. You can handin updated work as you keep working, but please make sure your final submission is in by Friday 10pm. You may also retrieve your current submission by selecting the ”Retrieve” checkbox from this same interface.

Documentation/Help

The course textbook contents, teachpacks references, a language reference are all available from the documentation section of the PLT Scheme website or from the Help → Help Desk menu in DrScheme. For help outside the lab sessions or office hours, please send an email to the class mailing list or email me at `gabri@cs..`

1. Getting started with DrScheme

DrScheme is a programming environment that allows the exploration of the different variants of Scheme that PLT Scheme provides. DrScheme looks like this:



```
Welcome to DrScheme, version 4.2.1 [3m].
Language: Beginning Student custom; memory limit: 128 megabytes.
Teachpack: world.ss.
>
```

The upper area is the *definitions window* which is used for defining new programs. The lower area (the *interactions window*) allows direct interaction.

Before we begin, we must first select the language to work with: from the ”Languages → Choose Language” menu item, choose ”Beginning Student”. We then click the ”Execute” button and the selected language should show up at the bottom left corner of the DrScheme window.

2. Numbers, Expressions, Simple Programs

Numbers in Scheme may be integers (4, -5), rationals ($\frac{7}{3}$), reals ($\sqrt{2}$) and complex ($3 + 5i$) with exact or inexact representations (preceded by *# i*, e.g. $\sqrt{(2)} = \#i1.4142135623730951$).

A **numeric expression** is of the form:

```
(operation $E_1 ... E_n$)
```

where E_i are either **atomic** (numbers, variables - see below) or **compound** expressions of the form given above. e.g. (+ 4 5), (sqrt 4), (+ 3 (/ 8 2))

Try evaluating a few expressions in the DrScheme interaction window and check the help desk to see if there are predefined operations for finding the cube of a number.

To see how a more complex expression is evaluated, we can use DrScheme's **step tool**. We will write our expression in the definitions window this time. Clicking "Run" will yield the final result 7. To follow the evaluation step by step, we will open a stepper window by clicking the "Step" button at the top of the DrScheme window. The left-hand side highlights in green the subexpression evaluated in the current step, while the right-hand side highlights in purple the result of this evaluation. By clicking "Next" repeatedly, we see the evaluation step by step.

A simple **program** can be thought of as a rule for producing data when given input data. This rule specification may include placeholders for the data being processed: **variables**. For example, a simple program that calculates the amount resulting from doubling a bet:

```
(define (double-bet b)
  (* b 2))
```

Once we have defined this program (in the DrScheme definitions window) we can call it as if it were a primitive operation (in the interactive window) thus applying it to concrete input data:

```
> (double-bet 10)
20
```

Even in designing such a simple program example we are actually following a **design recipe**:

1. We have given our program a *meaningful name* and decided what information it *consumes* (a number representing the initial betting amount) and what it *produces* (another number representing the amount after doubling). This is our program's contract **CONTRACT**. We should state this contract as follows (double ;; represents a comment) :

   ;;Contract: double-bet : number - $>$ number

   Our program's **HEADER** restates the program name and names the inputs:

   (define (double-bet b) ...)

2. Knowing the program's contract and parameters we are ready to formulate a short **PURPOSE STATEMENT** for the program.

   ;;Purpose: Calculate the result of doubling an initial bet of value b.

3. *Before we write the program body* we should add representative **EXAMPLES** of the program functionality.

   ;;Examples: (double-bet 5) should produce 10

4. Writing the **PROGRAM BODY**, which amounts to refining the header:

   (define (double-bet b) (* b 2))

5. Lastly, we must **TEST** our program on representative examples, at least on the ones we provided in the previous steps.

The final resulting program would be:

```
;;Contract: double-bet : number - > number
;;Purpose: Calculate the result of doubling an initial bet of value b.
;;Examples: (double-bet 5) should produce 10

(define (double-bet b) (* b 2))

(double-bet 5)
```

Problem 1

Define a program `height-conv` that takes a person's height in feet and inches and calculates the person's height in centimeters. (1 inch=2.54cm). Follow the design recipe above and remember to test your program.

Assume our task is to determine the total amount that was bet by two players after first doubling their bets. We now have a contract and header of the form:

```
;;Contract: add-double-bets : number number -> number
(define (add-double-bets bet-player1 bet-player2) ...)
```

Instead of *repeating* specifying the rule for one player doubling his/her bet, we can reuse that code:

```
(define (add-double-bets bet-player1 bet-player2)
  (+ (* 2 bet-player1) (* 2 bet-player2)))


(define (add-double-bets bet-player1 bet-player2)
  (+ (double-bet bet-player1) (double-bet bet-player2)))
```

Notice how this design choice makes increases our program's **readability** and **modularity**.

Use the stepper to observe how the evaluation of (add-double-bets 3 4) happens.

Problem 2

Define a program `cube-diff` that calculates the volume difference between 2 cubes for each of which the length of its side is given.

3. Conditional expressions

Conditional expressions in Scheme can use *relational operators* (<, >, =, <=, >=)

```
(= 5 7)
```

They will evaluate to *truth values (boolean values)* `true` or `false`.

Compound conditional expressions using operators *and, or, not* work on boolean values.

```
(not (or (= 5 7) (< 2 5)))
```

A *cond-expression* has the form:

```
(cond                                        (cond
   [question answer]                            [question answer]
   ...                              or          ...
   [question answer])                           [else answer])
```

Each [question answer] line is called a *cond-clause* and the *question* must be a valid conditional expression. Scheme will evaluate a cond-expression by evaluating these conditions one by one. For the first one that evaluates to true, Scheme evaluates the corresponding answer which becomes the value of the entire cond-expression.

```
> (cond
    [(> 4 5) 0]
    [(= 4 4) (+ 1 1)])
2
```

An answer may be another cond-expression, for example:

```
> (cond
    [(> 4 5) 0]
    [(= 4 4) (cond
                [(> 2 5) 4]
                [else 5])])
5
```

Problem 3

Define a program `is-traffic-low?` that determines if the traffic is low based on if the weather is sunny or not, windy or not and the day of the week (weekend or not) knowing that:
1. The traffic is always high on sunny weekends.
2. The traffic is always low on cloudy weekdays.
3. In all other cases the trafic is low if it's windy.
For example (is-traffic-low? false false false) should evaluate to true. Assume the first argument indicates if the weather is sunny, the second if it is windy and the third if the day is a weekend.

4. Working with images

From the "Language → Add teachpack" menu item select *image.ss*.

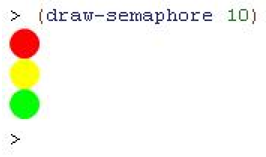
This teachpack offers primitives for manipulating images. Please read carefully through the teachpack documentation available from the help desk or online here.

Practice drawing a few basic shapes and working with their pinholes. Recall that *pinholes* are not necessarily the geometric centers of the objects drawn. We can retrieve the pinhole coordinates by using the *pinhole-x* and *pinhole-y*, move a pinhole using *move-pinhole*. We can compose basic images by overlaying them on their pinholes (*overlay* and *overlay/xy*).

We can include pictures directly in the definitions or interactions window in DrScheme by just "Copying" the image and then Pasting it in DrScheme (e.g. using the Edit - > Paste menu item).

Problem 4

Using the overlay functions create a semaphore with circles of given radius



```
> (draw-semaphore 10)
```

```
>
```

Problem 5

Align 2 copies of the following picture (download here) as shown below, where the picture on the left-hand side is the original picture, and the one on the right-hand side is the middle part of the former (25% of the original picture has been removed on each side):