# CS370 – ASSIGNMENT #7

## NAME: Kostiantyn Makrasnov

**GRADE:**

|  |  |  |
| --- | --- | --- |
| **CATEGORY** | **POINTS** |  |
| EX07\_01 |  | 5 |
| EX07\_02 |  | 15 |
| EX07\_03 |  | 5 |
| EX07\_04 |  | 15 |
| EX07\_05 |  | 5 |
| EX07\_06 |  | 15 |
| EX07\_07 |  | 5 |
| EX07\_08 |  | 15 |
| EX07\_09 |  | 20 |
|  |  |  |
| **TOTAL** |  | 100 |

## EXERCISES:

**Note: There are two programming assignments for each language. The assignments are purposefully the same and simple. I want you to be somewhat comfortable with the interfaces for each language, and not worry too much about the paradigms each language represents. We’ll get to those in the next weeks.**

**EX07\_01 –** Complete EX07\_01 (attached)

**EX07\_02 –** Complete EX07\_02 (attached)

**EX07\_03 –** Complete EX07\_03 (attached)

**EX07\_04 –** Complete EX07\_04 (attached)

**EX07\_05 –** Complete EX07\_05 (attached)

**EX07\_06 –** Complete EX07\_06 (attached)

**EX07\_07 –** Complete EX07\_07 (attached)

**EX07\_08 –** Complete EX07\_08 (attached)

**EX07\_09 –** Complete EX07\_09 (attached)

Compilers:

Imperative: We'll use Perl, which is already in cygwin, so no install necessary here

Object-oriented: We'll use SmallTalk, and Squeak is the IDE: <http://www.squeak.org/>

Note: Students in the past have hated Squeak. You can use Pharo as an alternative: <http://pharo.org/>

Functional: We'll use Haskell: <http://www.haskell.org/platform/>

Logic: We'll use Prolog, and SWIProlog is the IDE: <http://www.swi-prolog.org/Download.html>

Statistical: We’ll use R: <https://www.r-project.org/>

Perl Primer

**The Interface**

Well, it’s time for old-school programming. Notepad and the console window.

In C++ we learn about classes and objects. Some of you know C# and Java, where everything is an object. In Perl we don’t get to say object. Perl was originally an imperative language, though with version 5 it added some object-oriented features. We won’t be using those. Our focus is on the code statements and what they do, not on how objects are manipulated. The difference is very important! Instead of the mindset of “sending a message to an object” to do some work, we’re going to take advantage of functions to perform tasks for us. Really, it’s the difference between giving control to something else (an object in this case) and trusting it will do the right thing, and grabbing hold of a machine and getting it to do what I want it to do. In object-oriented programming, encapsulation means in part that functionality is hidden from me. In imperative programming, I know exactly how everything gets done.

Perl is a dynamic, interpreted language. By interpreted, I mean that Perl scripts are run through an interpreter rather than compiled into some kind of machine-level code. By dynamic, I mean that the language uses dynamic typing. Largely, it will only be used to develop small, quick scripts rather than robust software systems. Perl has been around quite a while, and it’s kind of a “kitchen-sink” language. If someone needed a task done, it was added to the Perl language. Further, the creator of the language favored having many ways to complete the same task. Therefore, you’ll see some inconsistencies in the language. It was designed for building small batch scripts and server scripts quickly and easily.

Perl is easy to install. Go to <https://www.perl.org/>, I chose to download [Strawberry Perl](http://strawberryperl.com/) on my Windows machine.

Probably Perl’s greatest strength is in parsing text, and pattern matching using regular expressions. If you don’t know regular expressions (RegEx) yet, at least to some extent, I recommend learning the basics. It is a powerful pattern matching language. Our text shows us a Perl version of the standard utility grep, which searches files for text that matches patterns. The code is 12 lines long.

**Syntax**

Perl steals much of its syntax from C/C++ languages. For example, semi-colons end statements, curly braces ({ and }) start and end scope, looping constructs are denoted with while and for. Perl also has a number of interesting new keywords, including die, my, and shift. We’ll see how those play out during this tutorial. Variables in Perl use one of three prefixes, depending on the kind of variable. Scalar variables (such as int, bool, and char) are prefixed with a $. Array variables are prefixed with a @, and associative arrays (dictionaries in STL) are prefixed with a %.

Though Perl is a kind of kitchen-sink language, it is still a valuable language to learn. Many Internet scripts are written in Perl, as are many scripts in Linux. It has been a top ten language on Tiobe’s programming language index for a long time now, which suggests its use is still strong.

For compatibility with Linux, Perl programs start off with:

#! /usr/bin/perl

In this line, you are informing the Linux OS that the text file contains a script, and the interpreter for that script is the program perl, found in /usr/bin. If you’re running your script in Windows, you can associate .pl files with the perl interpreter, and then you can just type the script file in, and it will be executed for you. Everything that follows that top line is script. We don’t need to have any kind of “main” function, or any other prologue to our code – it’s just code!

print "Hello World!"

Type the above code into notepad, then save the program wherever you want. Then start a console window, navigate to that directory, then type “perl hello.pl”, where hello.pl is the name of your script file. It would be wise right now to put some syntax errors into the code to see how interpreter failures look.

Alternative (Cygwin): Navigate to the folder that contains your hello world script. Type “chmod a+x hello.pl” to tell Cygwin that this script can be executed. Now type “./hello.pl” to execute your script with the Perl interpreter.

**Types**

Perl is a dynamic language, meaning that variable types are determined at runtime. You can make up variables whenever you want, and assign them to whatever you want. They can change types at any time in the program. Further, Perl will perform implicit type conversions automatically. As discussed in our text, one result of this is that Perl has a lot of operators, often specific to a type. Assignment is accomplished with = operator. That is $age = 21. Note, as a consequence of dynamic typing and implicit conversions, there are type-specific operators. We have the following operators:

**Operator Operation Operands Result**

+ Addition or unary positive numeric numeric

- Subtraction or unary negative numeric numeric

\* Multiplication numeric numeric

/ Division numeric numeric

% Modulus (remainder division) numeric numeric

. Concatentation string string

< Less than comparison numeric numeric

> Greater than comparison numeric numeric

lt Less than comparison string string

gt greater than comparison string string

For example, check out the following code:

$x = 5;

$y = 10;

$sa = "Hello ";

$sb = "World.";

$s1 = "5";

$s2 = "10";

print "s1 + y = " . ($s1+$y) . "\n";

print "s1 - y = " . ($s1-$y) . "\n";

print "s1 \* y = " . ($s1\*$y) . "\n";

print "s1 / y = " . ($s1/$y) . "\n";

print "s1 % y = " . ($s1%$y) . "\n";

The results below show that $s1, though a string, is converted to a number before performing the operations. Cool, huh?

|  |
| --- |
| C:\Perl64\eg>perl test.pl  s1 + y = 15  s1 - y = -5  s1 \* y = 50  s1 / y = 0.5  s1 % y = 5 |

Looping is accomplished in a similar fashion as most other languages. Perl supports for loops through the following syntax:

#! /usr/bin/perl

$total = 1;

for ($x = 1; $x <= 4; $x++) {

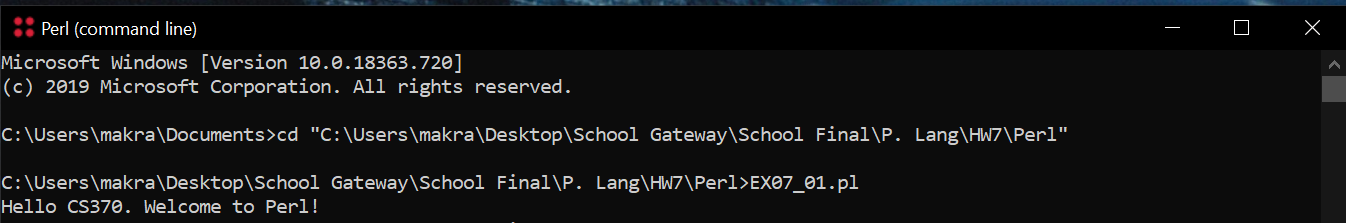
$total = $total \* $x;

}

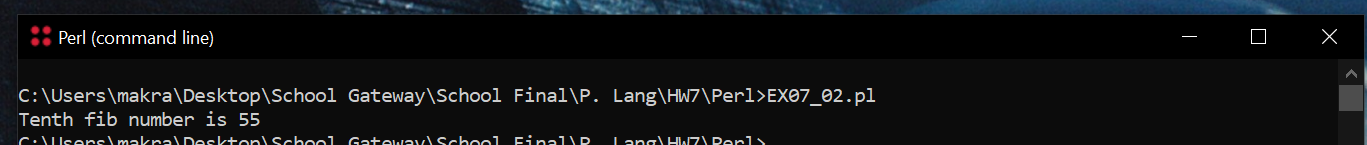
print "4! = " . $total . "\n";

Notice that the curly braces are required.

**EX07\_01**: Output the string “Hello CS370. Welcome to Perl!”

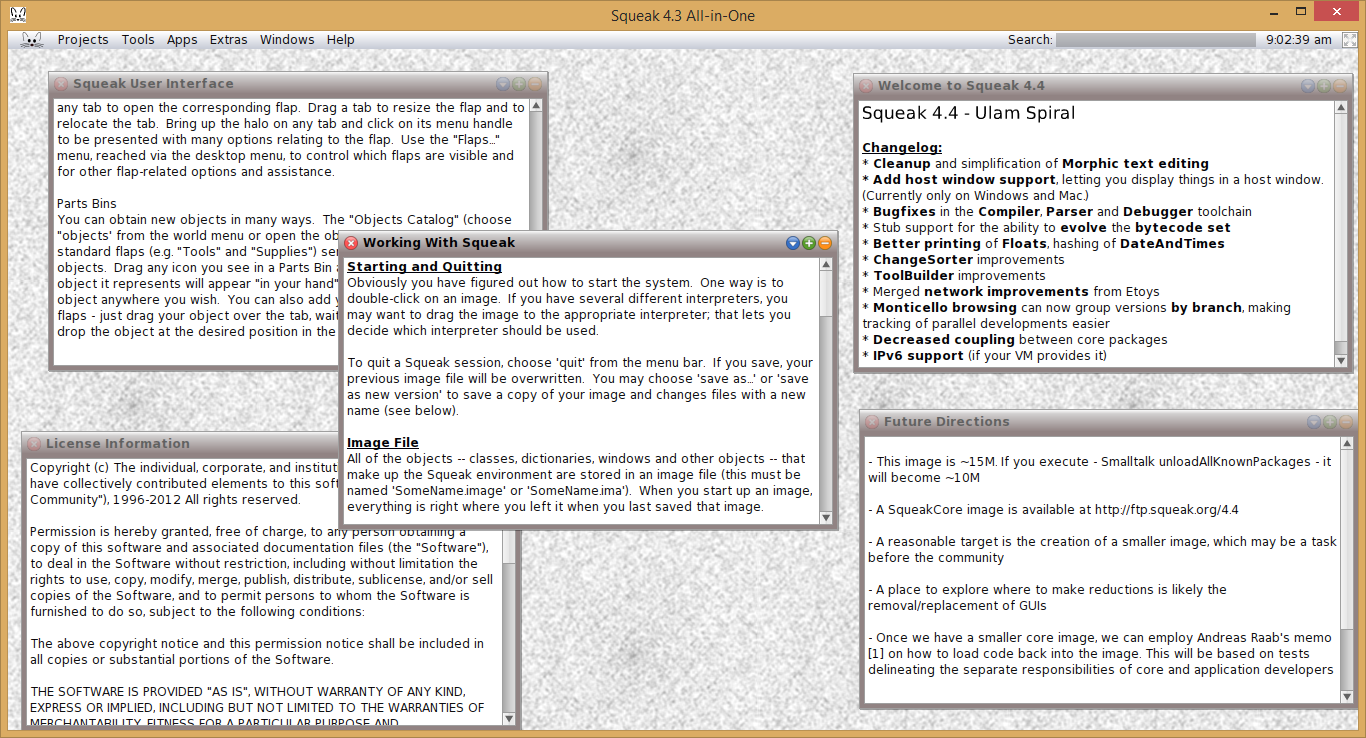


**EX07\_02**: Write a Perl script to output the 10th value in the Fibonacci sequence.



SmallTalk and Squeak IDE Primer

**The Interface**



First and foremost, remember this: In Smalltalk, everything is an object. And I mean everything. Strings, points, and mooses are objects. Integers, floats, and bools are objects (not values as in C++). Constant values are objects. Even the windows shown in the user interface are objects (and you can send messages to those windows from those windows). To be successful in Smalltalk and in using Squeak, you must remember the mantra: Everything is an object.

All objects allow you to *send messages* to those objects when you want them to do something. That is, you don’t control an object – you call on it to do a task, and get results from it when that task completes. It doesn’t matter how that task is completed, just that it completed. You don’t get to know how the job got done; you only get the results from that job.

You can download Squeak and Smalltalk [here](https://squeak.org/). Squeak is an IDE, but all of the code is stored with the IDE rather than in separate text files, so it makes sense to extract the zip file in a place where you keep most of your coding projects rather than where you keep your applications. Now, take a look at the screenshot above of the Squeak interface. When you start Squeak, you will see a number of different windows, each with different information. If you click on the background of the interface, you will get the menu shown above called “World”. For this tutorial, we’ll use a clean workspace window and the transcript window. So, first close some of those workspace windows (‘x’ is in the left-hand corner), then click on the background of the Squeak interface to open the World menu. Choose “open…” and then choose “Workspace”. This will give you a workspace window in which to type code. Now, re-open the World menu, choose “open…” and choose “Transcript”. This window will be where we will display the results of your programs (soon). You should have a Squeak window that resembles the figure above.

**Syntax**

OK, now what’s the Smalltalk mantra? Say it with me: “everything is an object”. No, really, say that out loud. “Everything is an object.” To get something to happen in Smalltalk, you “send a message” to the object that will do what you want. For example, if I wanted the Transcript window to display “hello world”, I’ll send a message to the Transcript window to show my text. Notice the slightly different mindset here from the imperative language. I don’t print “hello world” to the Transcript window. I tell the Transcript window to display “hello world” in whatever manner it sees fit. All messages are sent to objects using the syntax: *obj message*: *parameter.* Certain messages will require more than one parameter, so we’ll just add to the message: *obj message1*: *parameter1 message2: parameter2.* Note, statements are separated by a ‘.’ character. Note, the last statement in any block doesn’t need a ‘.’ character. OK, let’s try it out. In the Workspace window (or Playground window in Pharo), let’s tell the Transcript window to show “hello world”. Type the following into the Workspace window:

Transcript show: 'hello world'.

Now, highlight that line and wheel-click on it. Choose “do it” from the menu, and notice that the text is displayed. Yeah! You’ve written some Smalltalk code. Notice that you can make code execute using Ctrl-d rather than having to right-click every time.

Variable assignment is accomplished with the ‘\_’ message. I know, it’s weird, but comparison is the ‘=’ message and we don’t want to confuse those: myVar \_ 'hello world'.

**Types**

Again, everything is an object. When you start Squeak, you use an image file which stores all code for Smalltalk. In fact, 95% of Smalltalk is written in Smalltalk. The other 5% could have been written in Smalltalk also, but for efficiency it was written in a lower-level language. Since the entirety of Smalltalk is in the image file, it is modifiable. If you want to change how strings behave, you can go into the code for the string class and make whatever changes you want. They can be saved in the image file, and will be part of your version of Smalltalk forever.

Click again on the background for Squeak, and choose “open…” and “class browser”. All types available to you to program with are listed there, in a hierarchy. For example, if you want to know what kinds of messages you can send numbers, you’ll find them in Kernel-numbers/Number. The messages offered by a Number object are grouped by functionality, and those lists are listed in the third window. Choose “intervals”. Now in the far right window you’ll see the messages that are interval-type messages. In fact, these messages make up your loops. Type this into the workspace window and “do it”:

1 to: 9 do:

[:index |

Transcript show: (index printString), ' times... '.

Transcript cr.

].

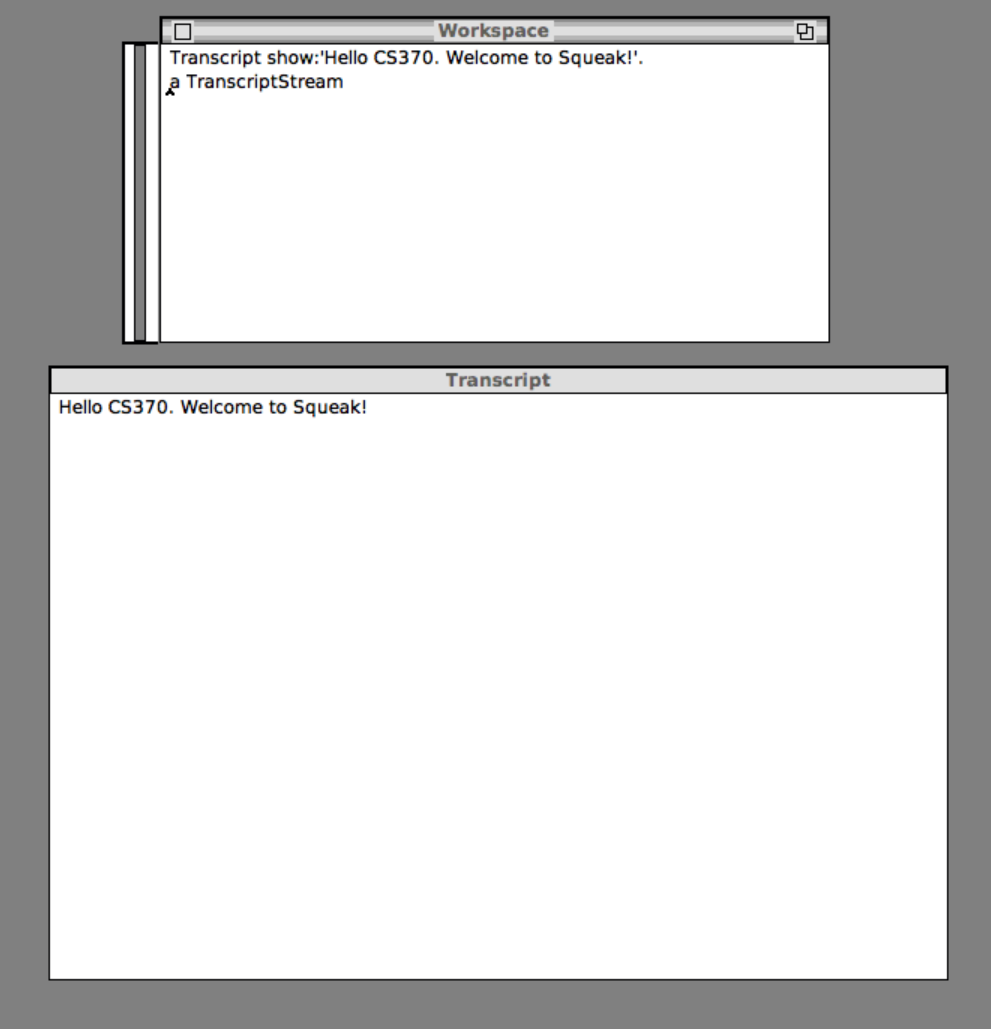
You’ll see that the transcript window has 9 strings printed. What happened? We used the “to:do:” message for numbers. Notice that even a constant value (e.g. 1) is an object that we can send messages to. This is slightly different than most languages you’re familiar with. You can’t say “Hello World”.size() in C++, can you? The to:do: message for numbers takes an integer (9) and a block of code. In this case, the block of code is denoted in the square brackets. The to:do: message above also shows how blocks can have local objects. The to:do: message has local object for the current index value. We can access that object using the :obj | syntax. The | character allows programmers to define local variables. Note that variables do not need to be declared (as in other languages), but the option is there. The :obj names the variable, and by prefixing it with :, it is assigned to the block’s local scope. Notice also that the ‘,’ character is string concatenation. The second line in the code block sends a “carriage return” message to the Transcript.

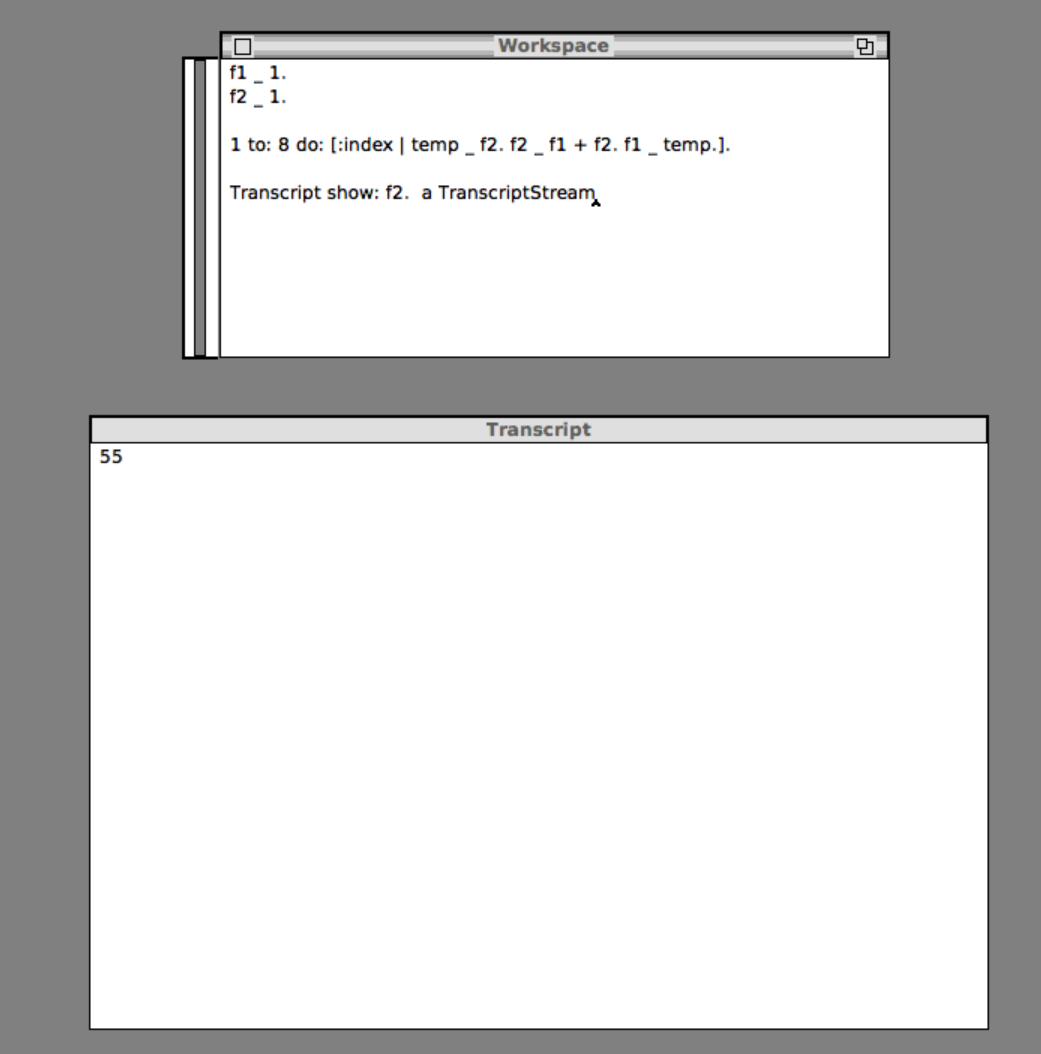
Multiple lines of code are separated with the ‘.’ character. For example, below is code to write 10! to the Transcript.

total \_ 1.

1 to: 9 do: [:index | total \_ total \* index].

Transcript show: total.

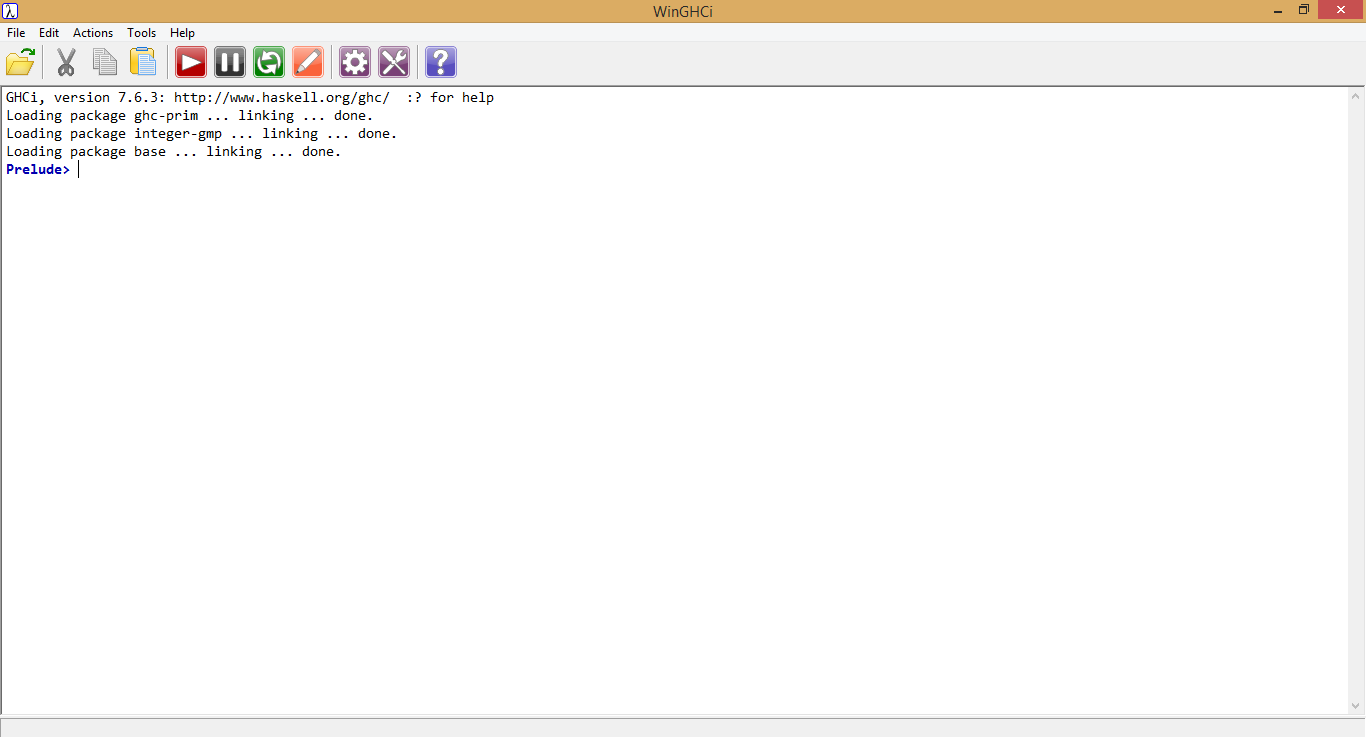
**EX07\_03**: Write the string “Hello CS370. Welcome to Squeak!” to the Transcript object.

**EX07\_04**: Write a Smalltalk script to write the 10th value in the Fibonacci sequence to the Transcript object.

Haskell and WinGHCi IDE Primer

**The Interface**

Haskell is a functional programming language, which means that functions are first-class object. That is, functions can be manipulated in meaningful ways, similar to atomic variables. You can install Haskell from [here](https://www.haskell.org/downloads/). I went ahead and installed the Haskell Platform. You can run winghci.exe to get the WinGHCi interpreter for Haskell.

  
In Haskell, programs are function evaluations. The results of evaluating those functions are output to the interpreter. The code is not “step-by-step”, but instead the code calls functions, which may call functions, which may call functions, and in the end, the functions evaluate to a result. It takes some getting used to, but once you do, it’s very powerful.

**Syntax**

Functions are defined using the equals sign (=). That is, to have a function that evaluates to “hello world”, I simply assign “hello world” to a function that take 0 arguments. To start, create a new text file called “test.hs”. Open that file in Notepad (or your favorite editor). Add the following code at the end of the file:

display = "hello world"

Save the file, then load the file into WinGHCi. Type “display” at the prompt to get the interpreter to evaluate the function you created.

**Types**

Types are key to Haskell, but we won’t go into that here. That’s much too deep for this particular primer. Many basic types are defined in a file called prelude.hs (C:\Program Files (x86)\WinHugs\packages\base\prelude.hs on my machine), which is automatically loaded when you start the interpreter. You will find basic types such as Int, Integer, Float, Double, and Rational. You can also find the basic functions that operate on those types.

Loops are set up in Haskell using recursion and pattern matching. By pattern matching, I mean that Haskell matches function argument values with function parameters. Most recursive functions need some kind of exit clause. We can use pattern matching to get those exit clauses. Consider the factorial example we’ve used already (and will use again). Factorial can be defined recursively to say that 0! = 1, and n! = n \* (n-1)!. In this case, the exit clause is 0, and we work our way toward that value. We can use this definition to express factorial in Haskell, as follows:

fact 0 = 1

fact n = (fact (n-1)) \* n

Note that we have the exit clause (0) as a pattern. If the parameter is 0, then we know the answer. Otherwise, we recursively work toward that exit clause and return the correct answer. Try it out. In the interpreter, type “fact 0”. Notice that the 0 matched the pattern, and returned 1. Now call fact with some other integer. That call matches the other pattern (fact n), and starts the recursive process to get your answer.

Again, types are important in programming in Haskell. It is as strongly-typed a language as you may ever use. In the interpreter, let’s find out what the types are of various things. First, type “:t 5”. You should see:

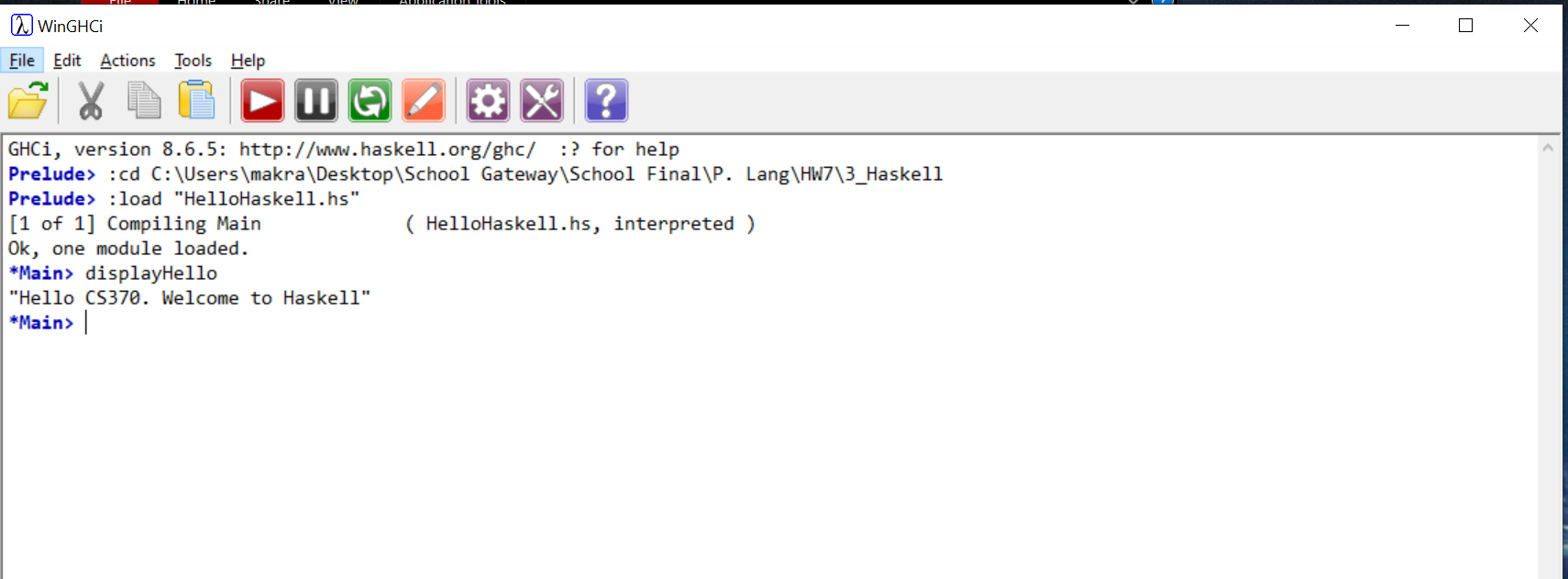
5 :: Num a => a

That is, 5 is a type that supports Num functionality. Now check the type of your display function, using “:t display”. You’ll see display :: [Char]. That is, display is a list of Char things. Finally, check out the type for fact. You should see:

fact :: (Eq a, Num a) => a -> a

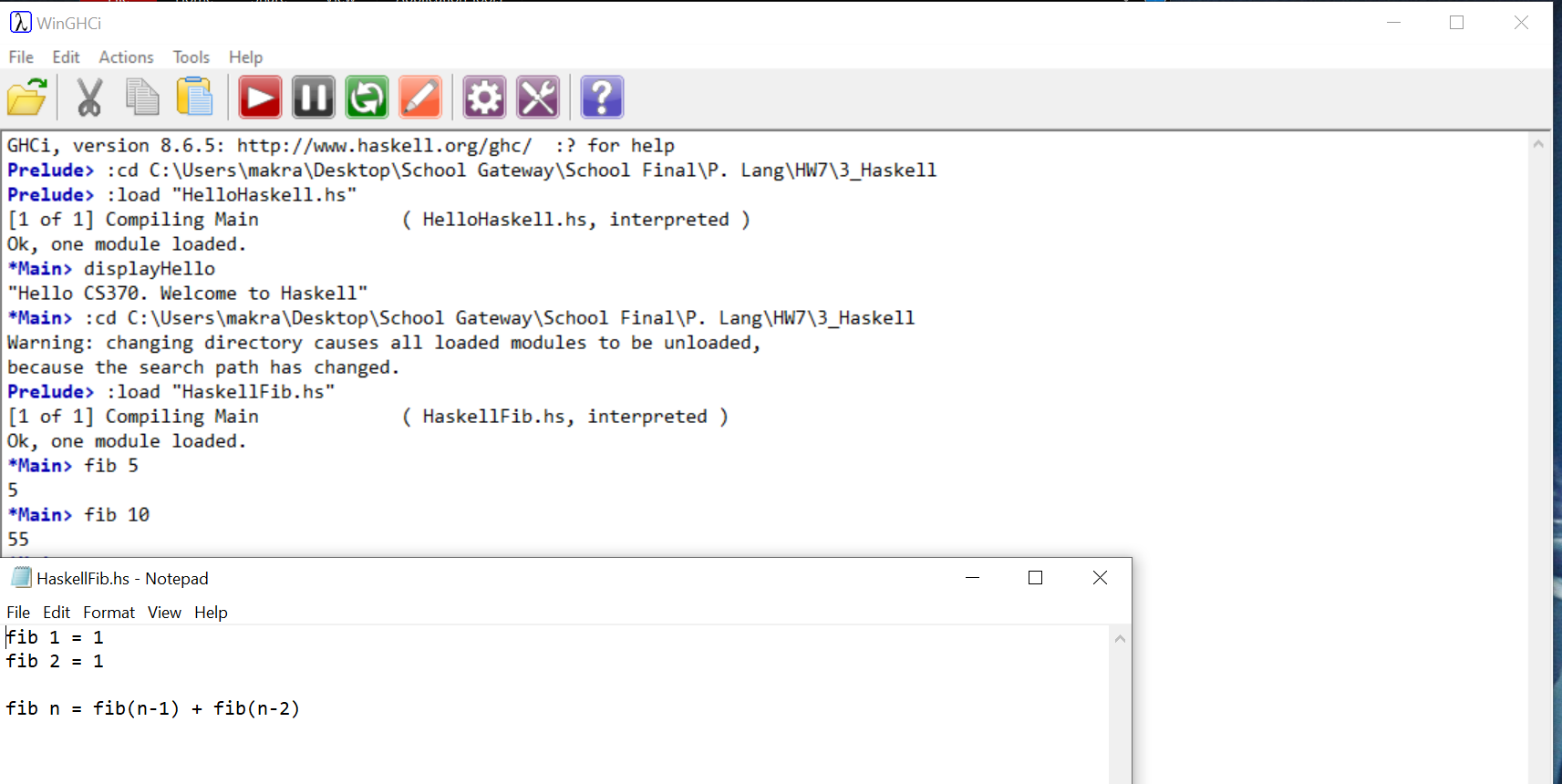
In this case, fact takes an a thing and returns something of that same a type. Also, a must support comparing for equality (Eq), and must be a number (Num). Notice that it doesn’t say that a is an integer, or a double, or whatever. It just says that it’ll return something of the same type as the input parameter.

**EX07\_05**: Write a function (that takes no arguments) that returns “Hello CS370. Welcome to Haskell”



**EX07\_06**: Write a Haskell function to output the 10th value in the Fibonacci sequence.

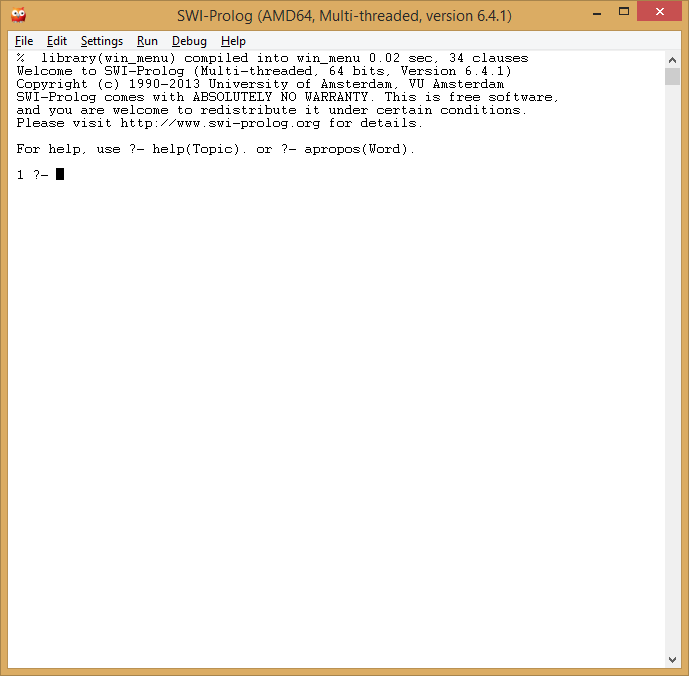
This is very cool!



Prolog and SWI-Prolog IDE Primer

**The Interface**

I downloaded Prolog from [here](https://www.swi-prolog.org/). It must be said that it would be inappropriate to have a “hello world” program written in Prolog. We’ll do it for completeness, but as we work through Prolog and logic programming, you’ll see that the power is very different.



The Prolog language is largely about *facts*, *rules*, and *logical inference* based on those rules. For example, I can say that Linda is the mother of Peter, Steven is the father of Peter, Peter is the father of Catherine, and Peter is the father of Andrew. Those are facts. Then we can specify rules that state that the father of the father of a person is the grandfather of that person. Finally, we can ask queries that use logical inference using the rules and return results.

**Syntax**

Facts are listed in a Prolog program database, which is loaded into the Prolog interpreter. Choose File/New to create a new Prolog database script. Type in the following *facts*:

father('Peter','Steven').

father('Geneva','Steven').

father('Andrew','Peter').

father('Catherine','Peter').

mother('Peter','Linda').

And follow those facts with the following *rules*, using the :- syntax:

grandfather(X,Y) :- father(X,Z), father(Z,Y).

Note that variables are given in upper case. With just that information, I can use the Prolog syntax to determine grandfathers. I first load the Prolog database using File/Consult, or with the following syntax: ['test.pl']. Then at the query prompt, I can get the grandfather by using grandfather('Andrew',X). The result is:

5 ?- grandfather('Andrew',G).

G = 'Steven'

**Types**

Prolog clearly supports strings. In fact, Prolog will support whatever you can make a fact or rule for. Numbers can be operated on with the standard operators (+, -, \*, /).

Loops are supported through recursion, just like Haskell earlier. Prolog rules also use pattern matching, and for recursive functions we use patterns to give the exit clauses. For example, the following rules compute factorials:

fact(0,1).

fact(M,N) :-

M > 0,

C is M-1,

fact(C,N1),

N is M \* N1.

We can get results from queries on these rules:

3 ?- fact(0,N).

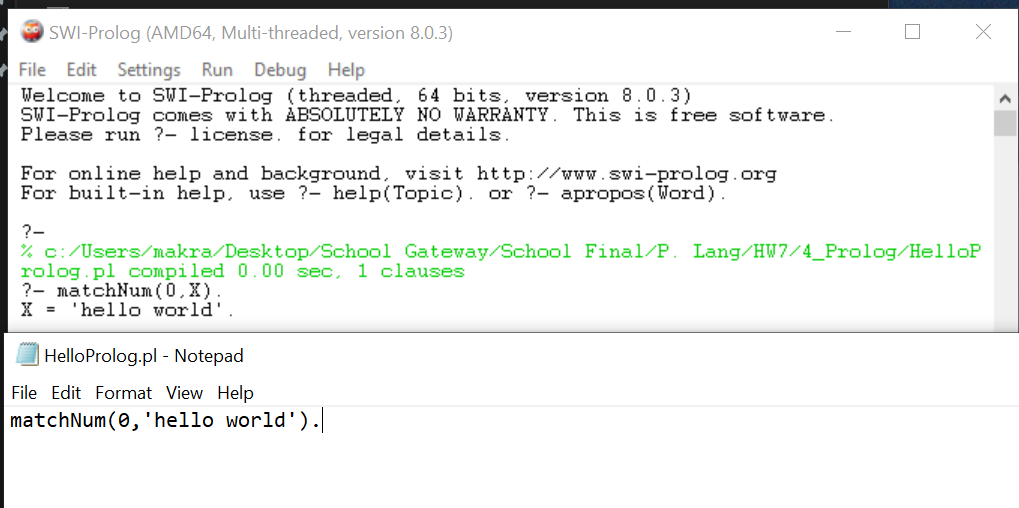
N = 1 .

4 ?- fact(5,N).

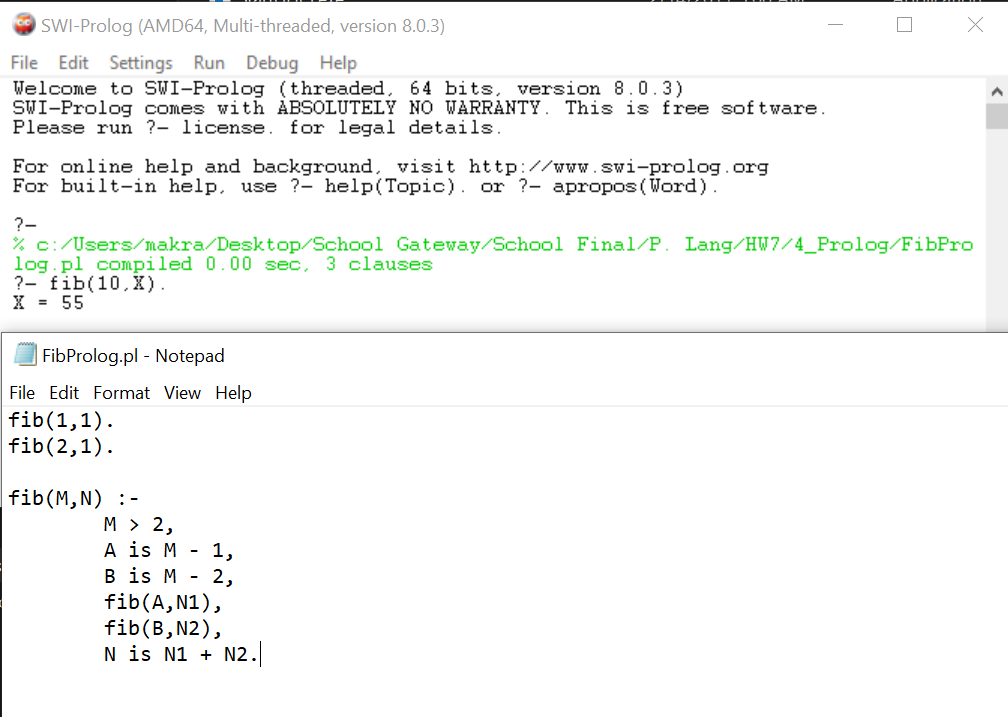
N = 120 .

5 ?-

Note the first rule for fact states that 0! is 1. That is, pattern matching sees that the first argument is a 0, and so sets the variable given in the second parameter to 1. For anything other values, Prolog infers results from the second rule. The second rule states first the first parameter is larger than 0. It then assigns M-1 to the local variable C, and uses C to recursively call fact. That recursion continues to decrement the first parameter until it matches the first parameter (0).

**EX07\_07**: Write a rule that matches the value 0 with the value “hello world”

**EX07\_08**: Write a set of Prolog rules to output the 10th value in the Fibonacci sequence.

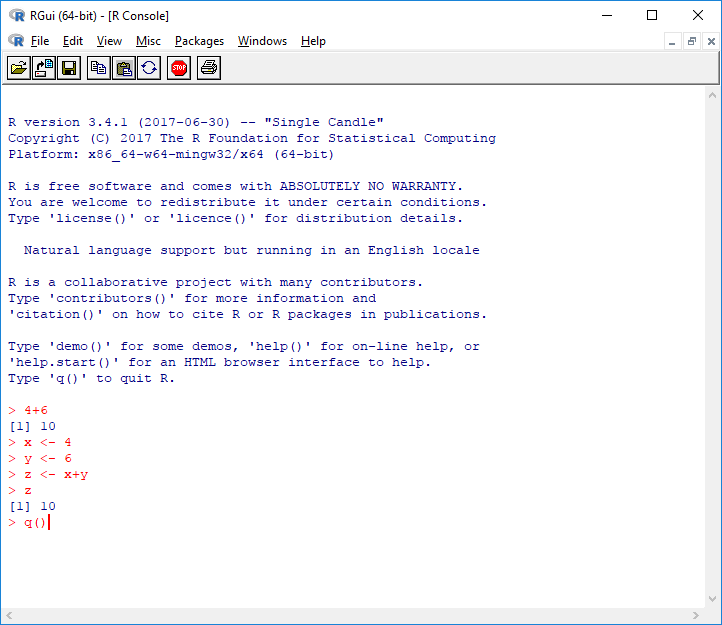


The **R** Project for Statistical Computing

**The Interface**

I think we all know by now that statistical modelling and big data analysis is kind of a big deal. The problems that need to be solved in this area are growing very rapidly, and many organizations in a wide variety of domains are reaching out for people with these skills. We clearly won’t be teaching statistics in this course – it takes two semesters of work led by real PhDs in Mathematics to really do this well. But we can at least try some things out, and you’ll be able to apply your own math skills toward more interesting problems.

You can download **R** [here](https://www.r-project.org/). In **R**, data are stored in various kinds of objects. And as we’ll see when we get to types, the various operations in **R** will be behave appropriately for the given types (as you would expect from a good type system). You will create objects at the console as you work in **R**, and those objects will be stored in your workspace (also called image, similar to SmallTalk) throughout your session. You can save your image for later use through the interface. To end a session, type q() in the console.



**EX07\_09**: Do the exercises in <https://www.nceas.ucsb.edu/files/scicomp/Dloads/RProgramming/BestFirstRTutorial.pdf> for chapters 2, 3, and 4.

**Note:** while *RGui* uses spaces to separate individual entries I will be also using commas for better visibility of the delimiters

**Chapter 2:**

**1.**

**a.** 3

**b.** 12

**c.** 56

**d.** 5, 2, 5

**e.** 4, 0, -6

**f.** 2, 0, 4

**g.** 16, 4, 36

**2.**

**a.** 7, 8, 9, 10, 11

**b.** 2, 3, 4, 5, 6, 7, 8, 9

**c.** 4, 6, 8, 10

**d.** 3, 13, 23

**e.** 6, 4, 2, 0, -2, -4

**3.**

**a.** 2, 2, 2, 2

**b.** 1, 2, 1, 2, 1, 2, 1, 2

**c.** 1, 1, 1, 1, 2, 2, 2, 2

**d.** 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4

**e.** 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4

**4.**

**a.** rep(6,6)

**b.** rep(c(5,8), 4)

**c.** rep(c(5,8), c(4,4))

**Chapter 3:**

**1.**

**a.** 9

**b.** 9, 2, 3

**c.** 9, 2, 6

**d.** 5, 9, 2, 3, 4, 12, 2, 9

**e.** 5, 9, 2, 3, 4, 6, 7, 0, 8

**2.**

**a. Each store:**

**Store 1:**

summary(y[seq(1,15,by=3)])

Min. 1st Qu. Median Mean 3rd Qu. Max.

11 16 22 23 33 33

**Store 2:**

summary(y[seq(2,15,by=3)])

Min. 1st Qu. Median Mean 3rd Qu. Max.

19.0 21.0 24.0 26.6 25.0 44.0

**Store 3:**

summary(y[seq(3,15,by=3)])

Min. 1st Qu. Median Mean 3rd Qu. Max.

29.0 45.0 49.0 46.6 54.0 56.0

**Each Day:**

**Monday:**

summary(y[1:3])

Min. 1st Qu. Median Mean 3rd Qu. Max.

29.00 31.00 33.00 35.33 38.50 44.00

**Tuesday:**

summary(y[4:6])

Min. 1st Qu. Median Mean 3rd Qu. Max.

16.00 20.50 25.00 28.67 35.00 45.00

**Wednesday:**

summary(y[7:9])

Min. 1st Qu. Median Mean 3rd Qu. Max.

19.00 26.00 33.00 35.33 43.50 54.00

**Thursday:**

summary(y[10:12])

Min. 1st Qu. Median Mean 3rd Qu. Max.

21.00 21.50 22.00 30.67 35.50 49.00

**Friday:**

summary(y[13:15])

Min. 1st Qu. Median Mean 3rd Qu. Max.

11.00 17.50 24.00 30.33 40.00 56.00

**Chapter 4:**

**1.**

**Create Matrixes:**

x<-matrix(c(3,2,-1,1),nrow=2,byrow=T)

y<-matrix(c(1,4,0,0,1,-1),nrow=2,byrow=T)

**Exercises:**

**a.** 6, 4

-2, 2

1. 9, 4

1, 1

1. 7, 8

4, -1

1. 3, 14, -2

-1, -3, -1

1. 1, 0

4, 1

0, -1

1. 0.2, -0.4

0.2, 0.6

**2.**

**a.** 3, 2

**b.** -1, 1

**c.** 2, 1

**d.** 4

**e.** 4, 0

1, -1