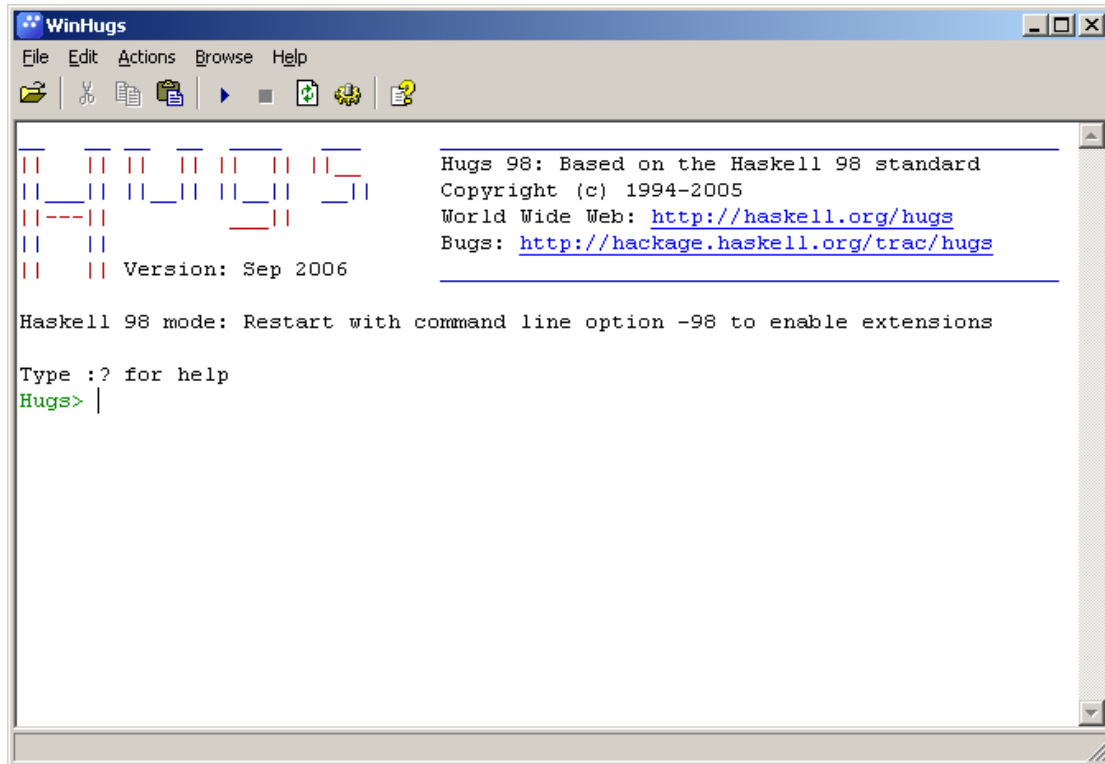


1. Getting Started on Hugs (a Haskell Interpreter)

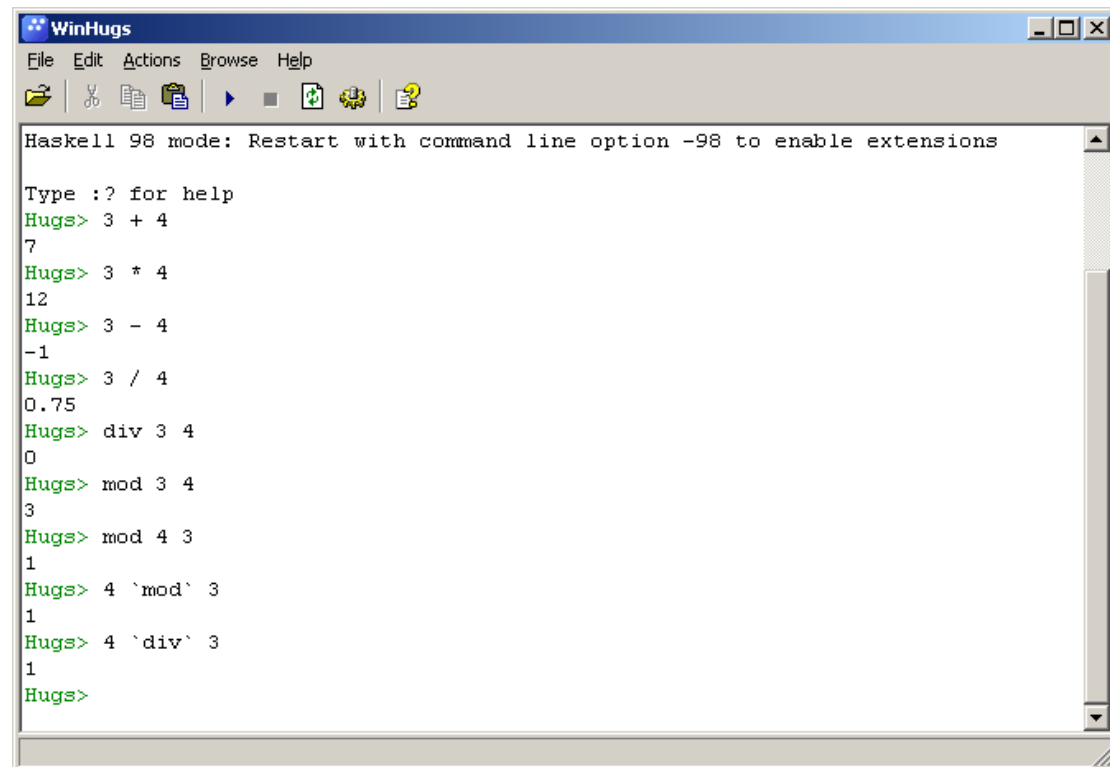
There should be an icon on the desktop for [Hugs](#) or [WinHugs](#) , double click on it to start Hugs.

Hugs should then run and look like:



Evaluating Arithmetic Expressions

When an arithmetic expression, e.g. $3 + 4$, is typed in, Hugs evaluates it and returns the result on the next line as is illustrated below.



```
WinHugs
File Edit Actions Browse Help
Haskell 98 mode: Restart with command line option -98 to enable extensions
Type :? for help
Hugs> 3 + 4
7
Hugs> 3 * 4
12
Hugs> 3 - 4
-1
Hugs> 3 / 4
0.75
Hugs> div 3 4
0
Hugs> mod 3 4
3
Hugs> mod 4 3
1
Hugs> 4 `mod` 3
1
Hugs> 4 `div` 3
1
Hugs>
```

Remark on div and mod, prefix and infix

The expression

div x y

divides x by y and ignores the remainder, whereas

mod x y

divides x by y and returns the remainder.

In Hugs one can write these operators in **prefix** or **infix**.

Prefix:	<code>div x y</code>	<code>mod x y</code>	<code>(+) 3 5</code>	<code>(*) x y</code>
Infix:	<code>x `div` y</code>	<code>x `mod` y</code>	<code>3 + 5</code>	<code>x * y</code>

Infix is more natural for arithmetic operators like $+$ $-$ $*$ $/$ and prefix for functions like `mod` and `div`. E.g. $3 + 5$ is more intelligible for most people than $(+) 3 5$.

Exercise 1

Start Hugs and use it to evaluate the arithmetic expressions in the above diagram. Next rewrite them in prefix and evaluate them Hugs again.

Operator Precedence and Associativity

What does $3 + 4 * 5$ evaluate to?

$*$ and $/$ have a higher precedence than $+$ or $-$ and in the absence of parentheses are evaluated first. So $3 + 4 * 5$ gives 23 whereas $(3 + 4) * 5$ gives 35.

How is $3 + 4 - 5 + 1$ evaluated?

Operators of equal precedence are evaluated from left to right, so the expression gives 3. Right to left evaluation would yield 1. We say that $+$, $-$, $*$ and $/$ have left to right associativity.

This can be contrasted with the power operator $^$.

$3 ^ 2$ gives 9, i.e. 3^2 .

How does $3 ^ 2 ^ 3$ evaluate? Left to right (729) or right to left (6561)? This operator has right to left associativity so the expression evaluates to 6561.

Exercise 2

Use Hugs to evaluate:

```
3 + 4 * 5
3 + (4 * 5)
(3 + 4) * 5
```

```
3 + 4 - 5 + 1
((3 + 4) - 5) + 1
3 + (4 - (5 + 1))
```

```
3 ^ 2
3 ^ 2 ^ 3
(3 ^ 2) ^ 3
3 ^ (2 ^ 3)
```

Which operator has higher precedence, $*$ or $^$? Evaluate :

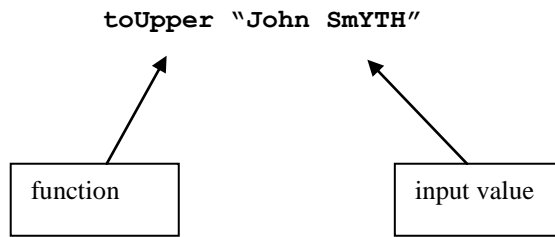
```
3 * 2 ^ 2
(3 * 2) ^ 2
3 * (2^2)
```

Functions and Functional Programming

Haskell is a *functional programming* language and as such derives its power by the extensive use of functions.

First of all, what is meant by a function? One can think of a function as representing some computation or calculation that can be performed on some value. The result of the computation is then ‘given back’ or ‘returned’ to whoever/whatever initiated the computation.

For example, suppose there is a computation or program that can convert any string into upper case, e.g. “John SmYTH” to “JOHN SMYTH”. We can think of this computation as a function. We don’t need to know how it works in order to apply it. Suppose further that this computation has been given the name **toUpper** say. When this program or computation is run on “John SmYTH”, we can think of this as applying the function **toUpper** to the value “John SmYTH”. In Hugs such a computation could be performed by typing:



As a further example, suppose there is a computation or function called **square** which squares every number it is applied to. So in Hugs

```
square 4
```

would yield or return 16.

Functions can be applied to single value, sometimes called an argument, or two values or in fact as many values or arguments as you want. Even better, functions can be applied to complex data structures (will be seen later on) and to other functions. This makes Hugs very flexible and thus expressive. It is a higher level language than C, Java or C++ in the same way that C is higher than assembly language.

Something else about functions in all programming languages is that when a function finishes doing some computation, it returns or gives back a single value.

The fact that functions are so to speak the primary citizens in Haskell is the reason why it is called a **functional** language. Every program is constructed entirely from functions.

Comment on side effects

Functions are also widely used in C and most programming languages, so what makes functions in Haskell so special? The difference is very important but may be hard to grasp at the moment. Put tersely: Haskell is side-effect free or pure. What this means is that if a function is applied to the same value at different points in a program, it always returns the same result or it always does the same thing. The result depends only on the input value and the function itself. By contrast, in C, a function could return one value here and a different value at another point when applied to the same input value. It's behaviour may depend on things outside of the function itself. One cannot depend on it to do the same thing always.

A significant benefit of a function always doing the same thing or being free of side effects as is the case in Haskell, is that it is easier to prove that a piece of software written works correctly. This can be especially important in safety critical applications.

Functions included in Haskell

Haskell comes with many useful functions including the trigonometry functions and dozens of others.. A function in Haskell has a similar meaning to a function in mathematics.

For example, the **sin** function:

sin 30° means you apply the sin function to the angle 30 degrees (30°). The result is 0.5. In Haskell we refer to sin as the function and 30° as the argument or input value. One often see an expression like **sin x** or **sin(x)** where **x** is a variable. In programming, **x** is referred to as a parameter of the function **sin**. When the **sin** function is to be used or applied, a value is substituted for **x** and then the sin of the value is computed. The value substituted for **x** is called the argument.

Also in Haskell, the sin function expects the angle in radians rather than degrees. So instead one would write:

```
sin (pi/6)
```

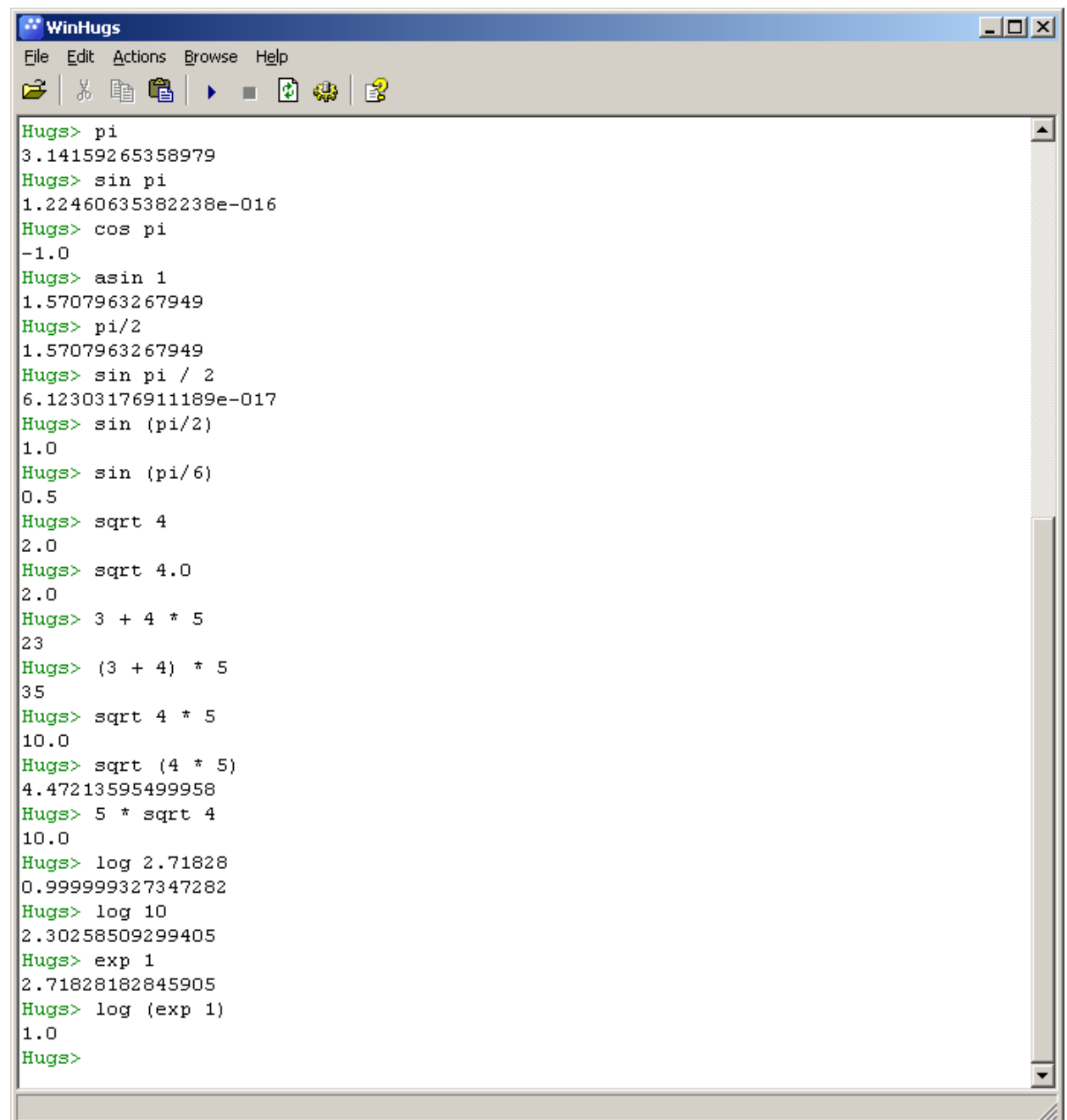
pi is a predefined constant in Hugs with a value of 3.14159, i.e. an approximation to π .

Why use parentheses? Why not `sin pi/6` ?

The reason is that functions have a higher precedence than arithmetic operators. So `sin pi/6` would give `(sin pi)/6` which is 0.

So functions are evaluated before `*` `/` `+` and `^`.

Some common built in Haskell functions are shown in the following diagram.



```
WinHugs
File Edit Actions Browse Help
[Icons]

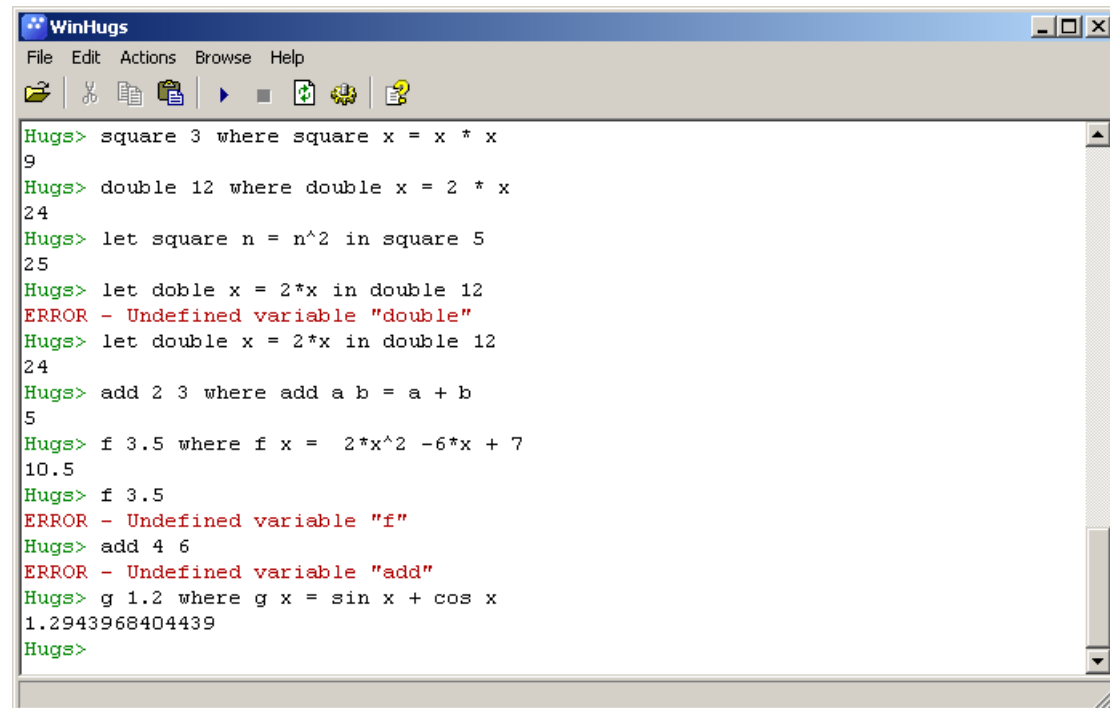
Hugs> pi
3.14159265358979
Hugs> sin pi
1.22460635382238e-016
Hugs> cos pi
-1.0
Hugs> asin 1
1.5707963267949
Hugs> pi/2
1.5707963267949
Hugs> sin pi / 2
6.12303176911189e-017
Hugs> sin (pi/2)
1.0
Hugs> sin (pi/6)
0.5
Hugs> sqrt 4
2.0
Hugs> sqrt 4.0
2.0
Hugs> 3 + 4 * 5
23
Hugs> (3 + 4) * 5
35
Hugs> sqrt 4 * 5
10.0
Hugs> sqrt (4 * 5)
4.47213595499958
Hugs> 5 * sqrt 4
10.0
Hugs> log 2.71828
0.999999327347282
Hugs> log 10
2.30258509299405
Hugs> exp 1
2.71828182845905
Hugs> log (exp 1)
1.0
Hugs>
```

Exercise 3

Try out all the expressions in the above diagram. Note that `log` refers to the natural log which is sometimes in maths written as *ln* or *log_e*.

Writing one's own functions in Hugs

In Hugs one can temporarily define one's own functions and constants and use them as the examples in the following diagram show. Try them!



```
WinHugs
File Edit Actions Browse Help
Hugs> square 3 where square x = x * x
9
Hugs> double 12 where double x = 2 * x
24
Hugs> let square n = n^2 in square 5
25
Hugs> let doble x = 2*x in double 12
ERROR - Undefined variable "double"
Hugs> let double x = 2*x in double 12
24
Hugs> add 2 3 where add a b = a + b
5
Hugs> f 3.5 where f x = 2*x^2 - 6*x + 7
10.5
Hugs> f 3.5
ERROR - Undefined variable "f"
Hugs> add 4 6
ERROR - Undefined variable "add"
Hugs> g 1.2 where g x = sin x + cos x
1.2943968404439
Hugs>
```

It can be seen from this that a function can be defined and applied in two ways, e.g.

double 12 where double x = 2 * x

or

let double x = 2 * x in double 12 .

Also, it is clear from the attempts to apply **f** and **add** in the bottom of the diagram, that the function definitions are temporary and only hold in the lines in which they are defined. So an **ERROR** occurs as shown.

Exercise 4

Using Hugs, try out the expressions in the previous diagram.

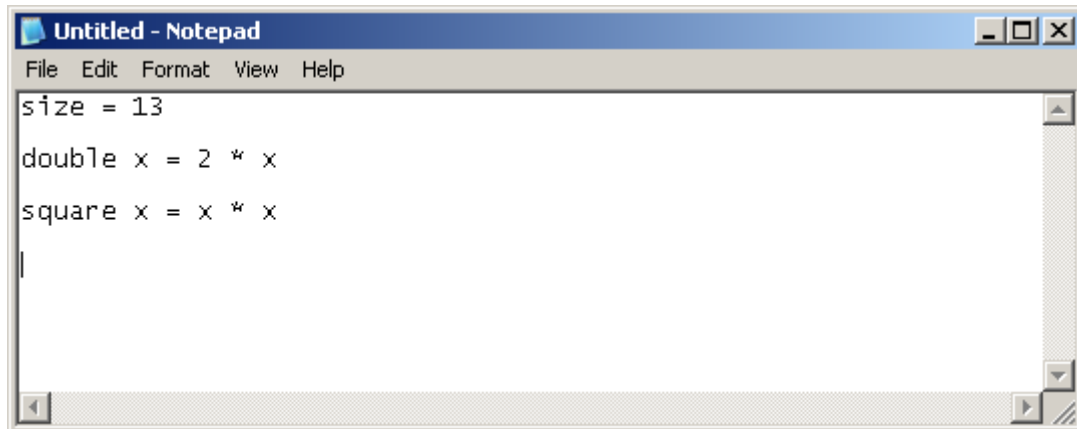
Using a Notepad to write and save a Haskell function

When a function is defined one may wish to use it several times. The earlier approach where the function has to be defined on each usage is not suitable for this. For repeated use, a function should be defined in a text file using **Notepad** or some text editor such as **Textpad** and then loaded into Hugs whenever needed.

A file storing Haskell code should have **.hs** at the end of its name. Such a file with a function definition can be thought of as a Haskell program.

Exercise 5

Create a folder using Windows Explorer called **myHugs** on your network drive.
Using Notepad, write the Haskell code as shown below:



```
size = 13
double x = 2 * x
square x = x * x
|
```

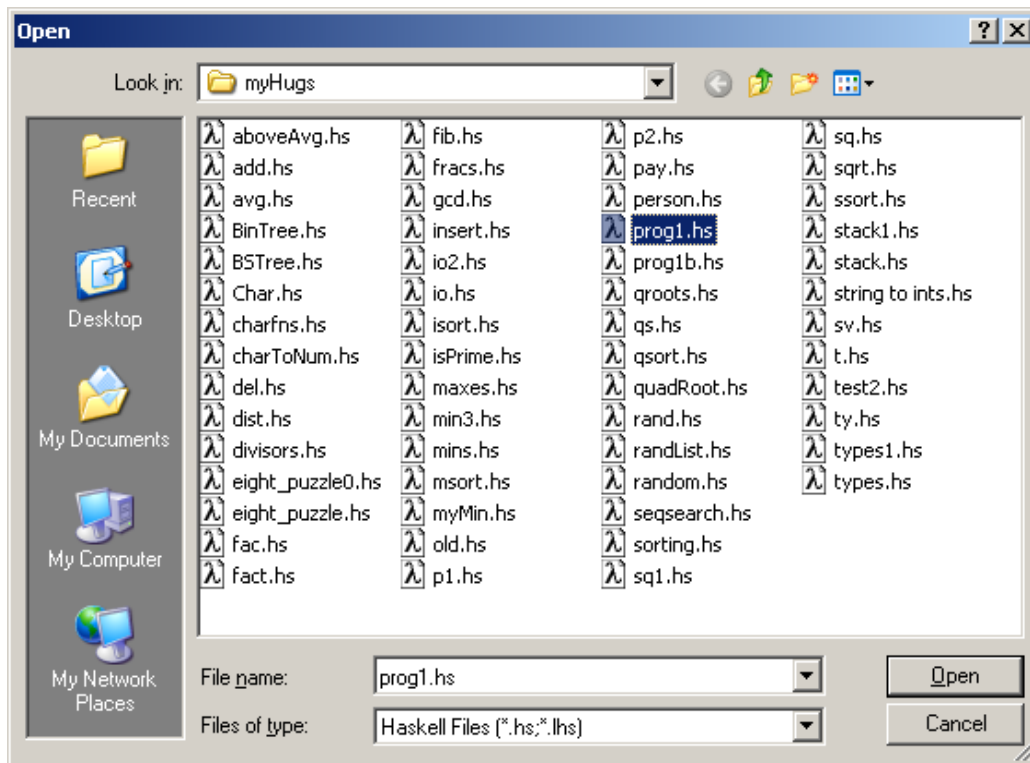
Then save the file with the name **"prog1.hs"** using the menu **File/Save as** to save it into the folder **myHugs**. Make sure to use quotation marks " " in the file name, otherwise notepad may save the file as *prog1.hs.txt*.

Loading the function definition into Hugs

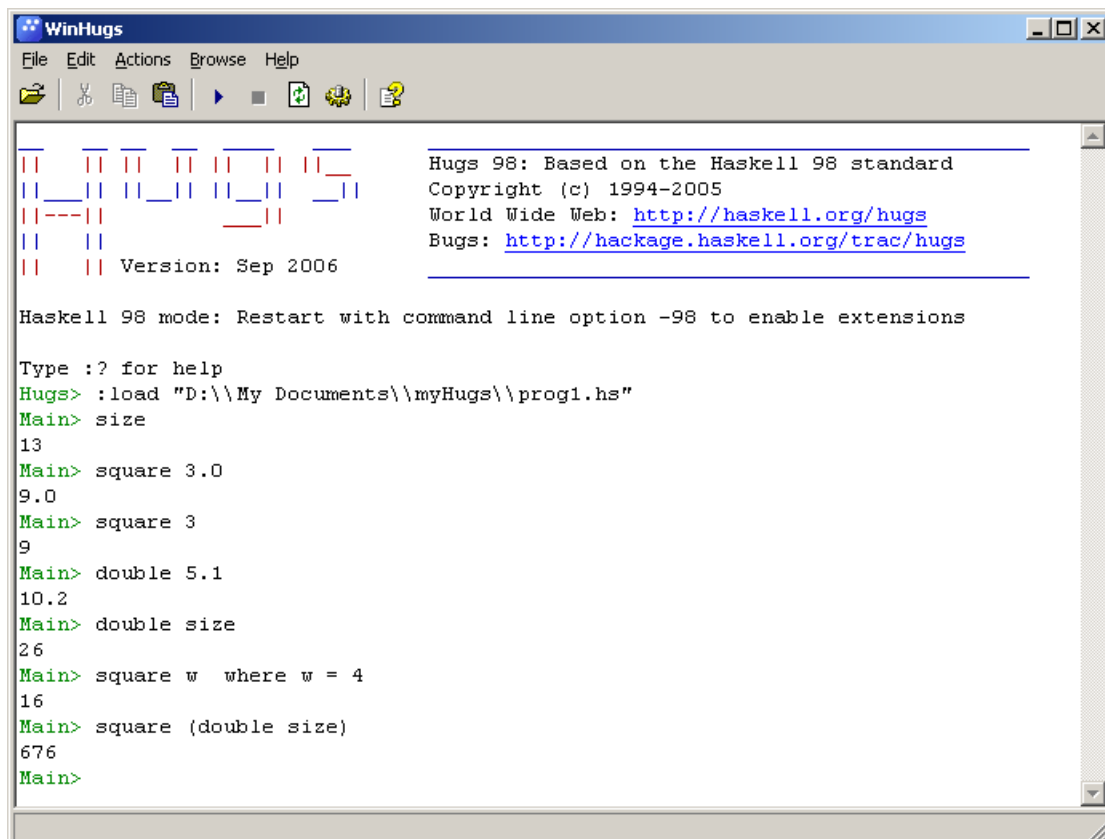
Now the function definitions have been saved to a text file. The next stage is to load them into Hugs.

Exercise 6

Use the Hugs menu **File/Open** or the icon on the toolbar as shown below to load the file that you saved a couple of minutes ago.

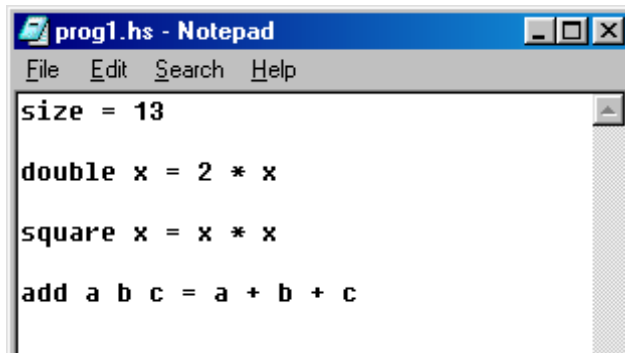


Hugs will then “knows” these definitions and you can use them as follows:



Exercise 7

Add another function definition to [prog1.hs](#) as shown below, save the file and reload it into Hugs.



```
prog1.hs - Notepad
File Edit Search Help
size = 13
double x = 2 * x
square x = x * x
add a b c = a + b + c
```

Make sure to save the file after modifying it. However, Hugs will not be aware of the change. You can use **File/Open** to load the program file again or press the reload button on the toolbar.

Then use Hugs to evaluate:

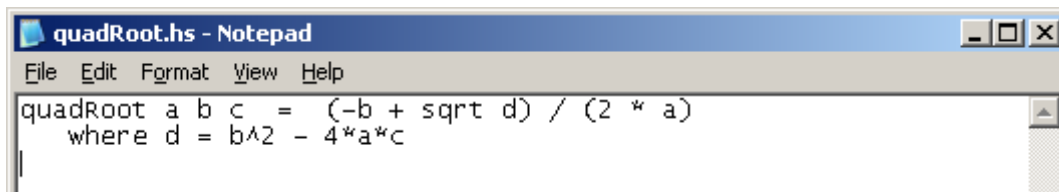
```
add 6 7
add 6 7 10
```

What happens with **add 6 7** ?

Exercise 8

Use notepad to write a program called "[quadRoot.hs](#)". Then load it into Hugs and run it. [quadRoot.hs](#) has a function **quadRoot** to find one root of a quadratic equation according to the formula

$(-b + \sqrt{b^2 - 4ac}) / 2a$. The function is defined with 3 parameters a b and c.



```
quadRoot.hs - Notepad
File Edit Format View Help
quadRoot a b c = (-b + sqrt d) / (2 * a)
  where d = b^2 - 4*a*c
```

An argument is a value that the function is applied to. For example

```
quadRoot 1.1 9.2 (-0.4)
```

has 3 arguments 1.1, 9.2 and -0.4. The parameters a, b and c are set to these values when the function is computed.

Remark on negative numbers

When entering a negative number for use in a Haskell function, you should enclose it in parentheses as in (-0.4) above. Otherwise Haskell think that the minus sign - means subtraction.

Exercise 9

Write a program with a function called **dist** for calculating the distance between two points (x1,y1) and (x2,y2). This function has four inputs and should be like:

```
dist x1 y1 x2 y2 =
```