C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 7: Pattern matching, custom types in Haskell

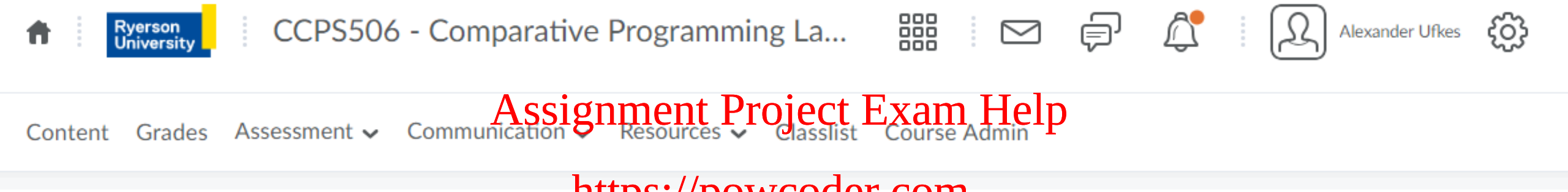
Ryerson University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

The copyright to this original work is held by Alex Ufkes. Students registered in course COPS 506 can use this material for the purposes of this course but no other use is permitted, and there can be no sale or transfer or use of the work for any other purpose without explicit permission of Alex Ufkes.

Course Administration



It's important to start thinking about the assignment if you haven't already.

Any Questions?

Let’s Get Started!

Types in Haskell

Statically Typed:

- Haskell uses static type checking.
- Every expression is assigned a type.
- If a function's arguments aren't the expected type, a compile error occurs.

Type Inference

- Like Python, and unlike Java, we need not specify type.
- It is inferred by the context: X = "Hello", X is a string.
- However, we can explicitly specify types.
- Good practice when we know what types we want; compiler will give errors upon type mismatch.

Types in Haskell

`:t` can be used to reveal type:

WinGHCi

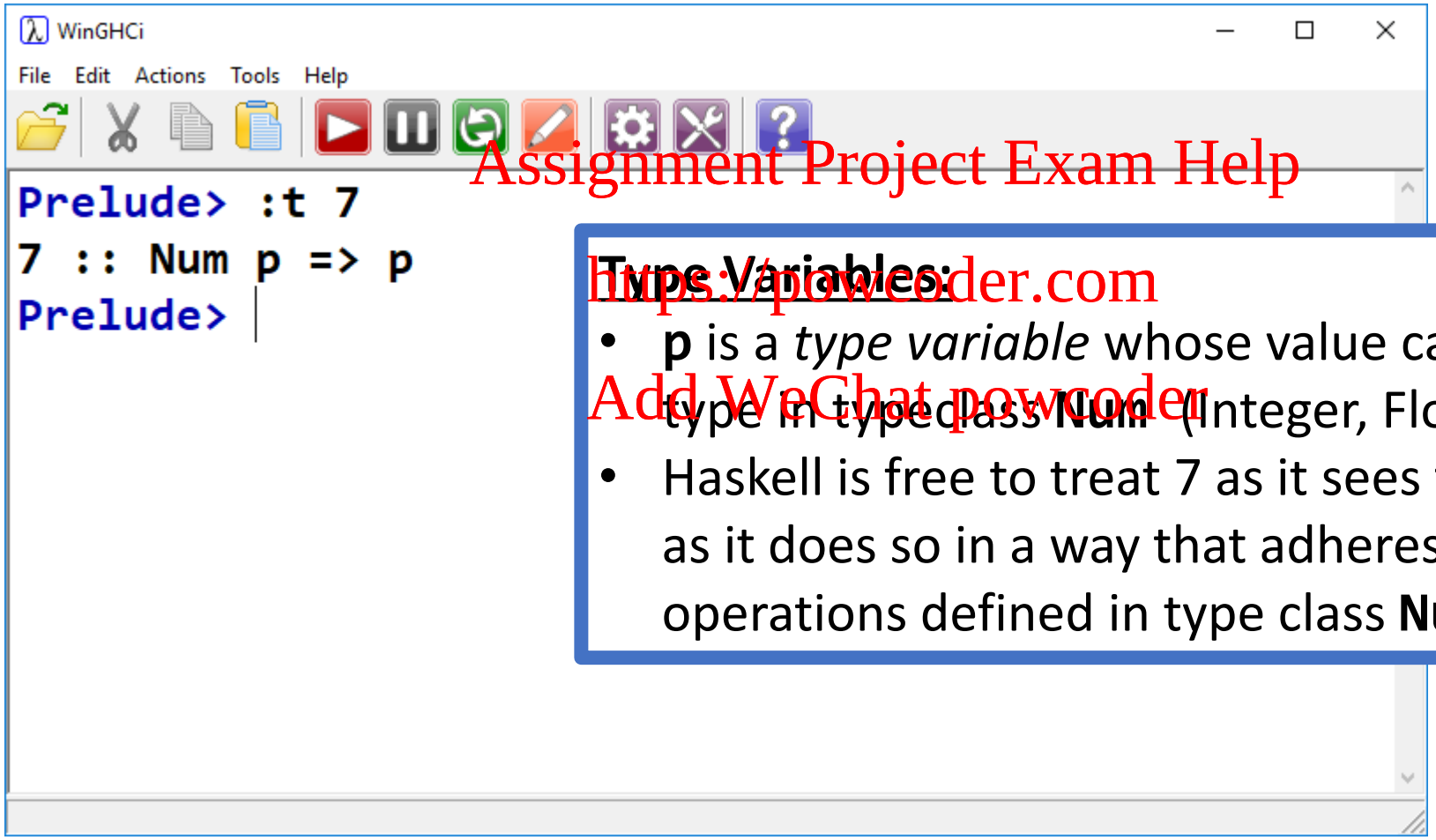
File Edit Actions Tools Help


```
Prelude> :t 1
1 :: Num p => p
Prelude> :t 1.0
1.0 :: Fractional p => p
Prelude> :t 'a'
'a' :: Char
Prelude> :t "Hello"
"Hello" :: [Char]
Prelude> :t 1 > 2
1 > 2 :: Bool
Prelude>
```


- 1 is instance of **Num** type class.
- 1.0 is instance of **Fractional** type class.

- ‘a’ is a **Char**
- “Hello” is a **[Char]**
- **[Char]** = **String**
- t is a **Bool**

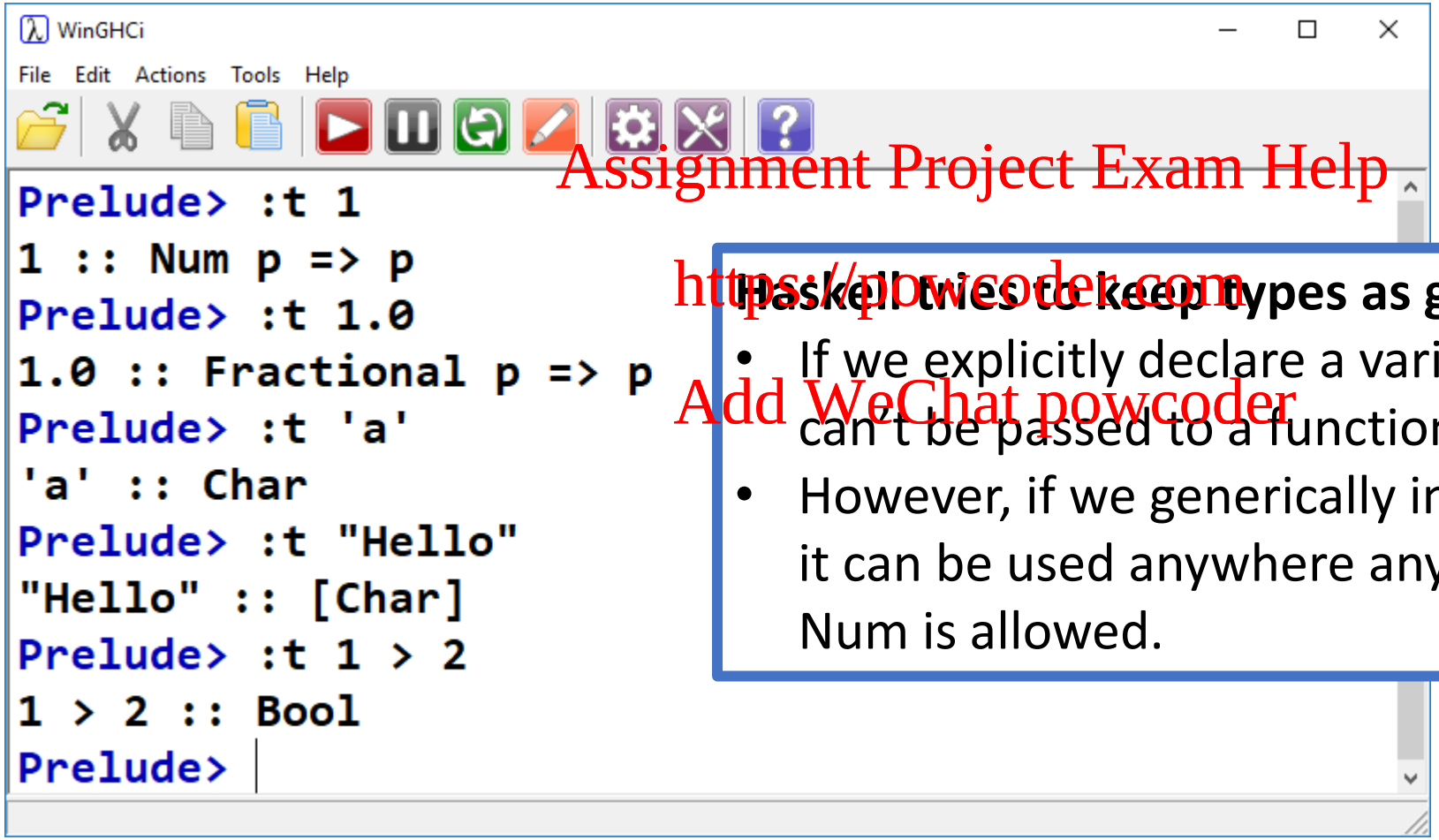
Num p => p ?



Type Variables:

- **p** is a *type variable* whose value can be any type in typeclass **Num** (Integer, Float, etc.)
- Haskell is free to treat 7 as it sees fit, so long as it does so in a way that adheres to the operations defined in type class **Num.**

Typeclasses?



Haskell tries to keep types as generic as possible

- If we explicitly declare a variable as integer, it can't be passed to a function requiring float.
- However, if we generically infer it to be a **Num**, it can be used anywhere any other member of Num is allowed.

Types in Haskell

We can explicitly indicate types:

WinGHCi

File Edit Actions Tools Help

```
Prelude> :t 1
1 :: Num p => p
Prelude> :t 1.0
1.0 :: Fractional p => p
Prelude> :t 5::Int
5::Int :: Int
Prelude> :t 5.0::Double
5.0::Double :: Double
Prelude>
```


- Use **::** to assign a type
- My advice for you is to start by letting the inference engine figure it out.
- **At this point, it knows better than you.**

Types in Haskell

We can explicitly indicate types:

WinGHCi

File Edit Actions Tools Help

```
Prelude> 5::Int
5
Prelude> 5::Double
5.0
Prelude> m = 5
Prelude> :t m
m :: Num p => p
Prelude> m = 5::Double
Prelude> :t m
m :: Double
Prelude>
```


When defining a name, can indicate type explicitly:

Type Classes

Type polymorphism and type variables:

Recall: Overloading

- In languages like C++, the **==** operator is overloaded to work with many different types.

- Numeric type equality and string equality are performed differently.

- In general, if we want to compare two values of type α, we use an ***α-compare***
- α is a *type variable*, because its value is a type.

Type Classes

Consider the equality (==) operator:

Takes two parameters, each of the same type (call it a), and returns a Boolean

This operator may not be defined for *all* types, just some.

Thus, we can associate == with a specific ***type class*** containing those types for which == is defined.

This type class is called **Eq** in Haskell.

Eq Type Class

(==) is defined for types in typeclass **Eq**

```
*Test> :t (==)
(==) :: Eq a => a -> a -> Bool
*Test>
```

- (==) takes two args of type **a**, where **a** is a member of type class **Eq**
- It returns **Bool**

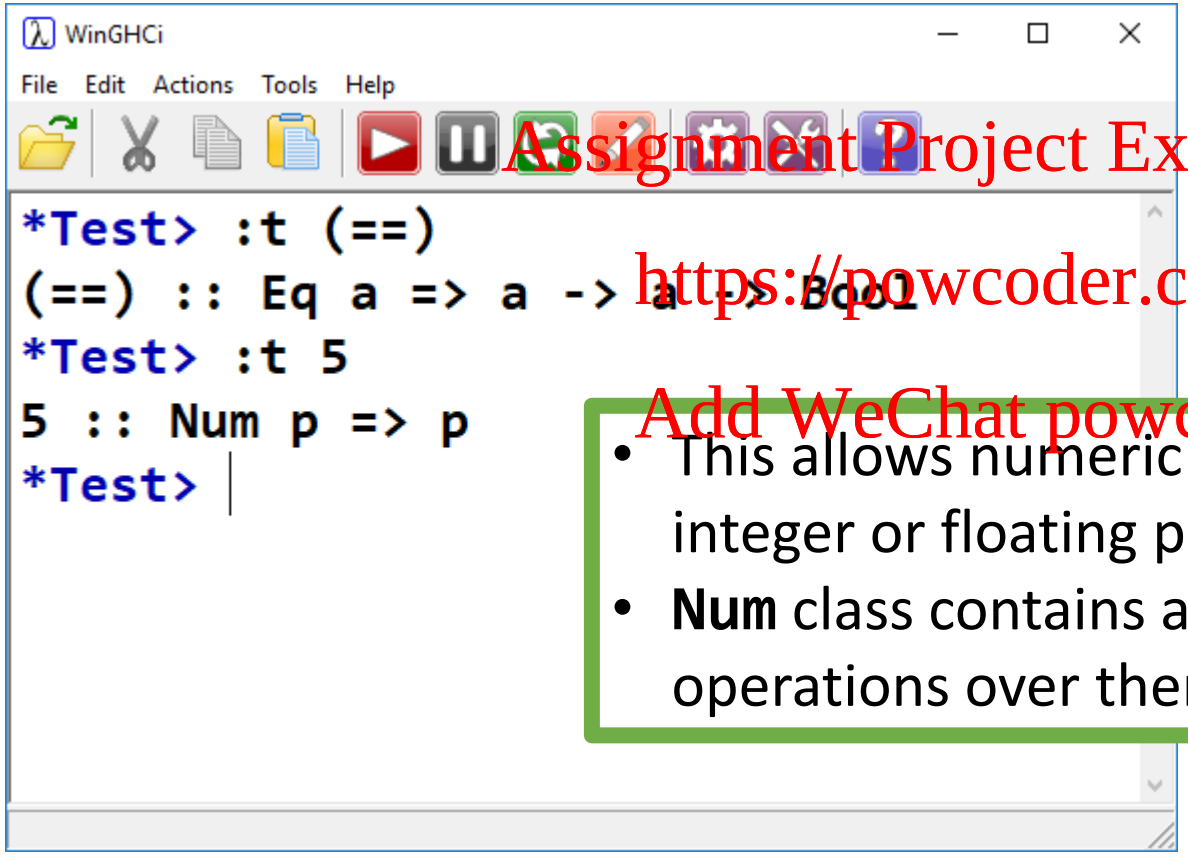
- If a concrete type, **a**, belongs to a certain type class, we say **a** is an *instance* of that type class.
- **Int** is an instance of **Eq**, for example.

WinGHCi

File Edit Actions Tools Help

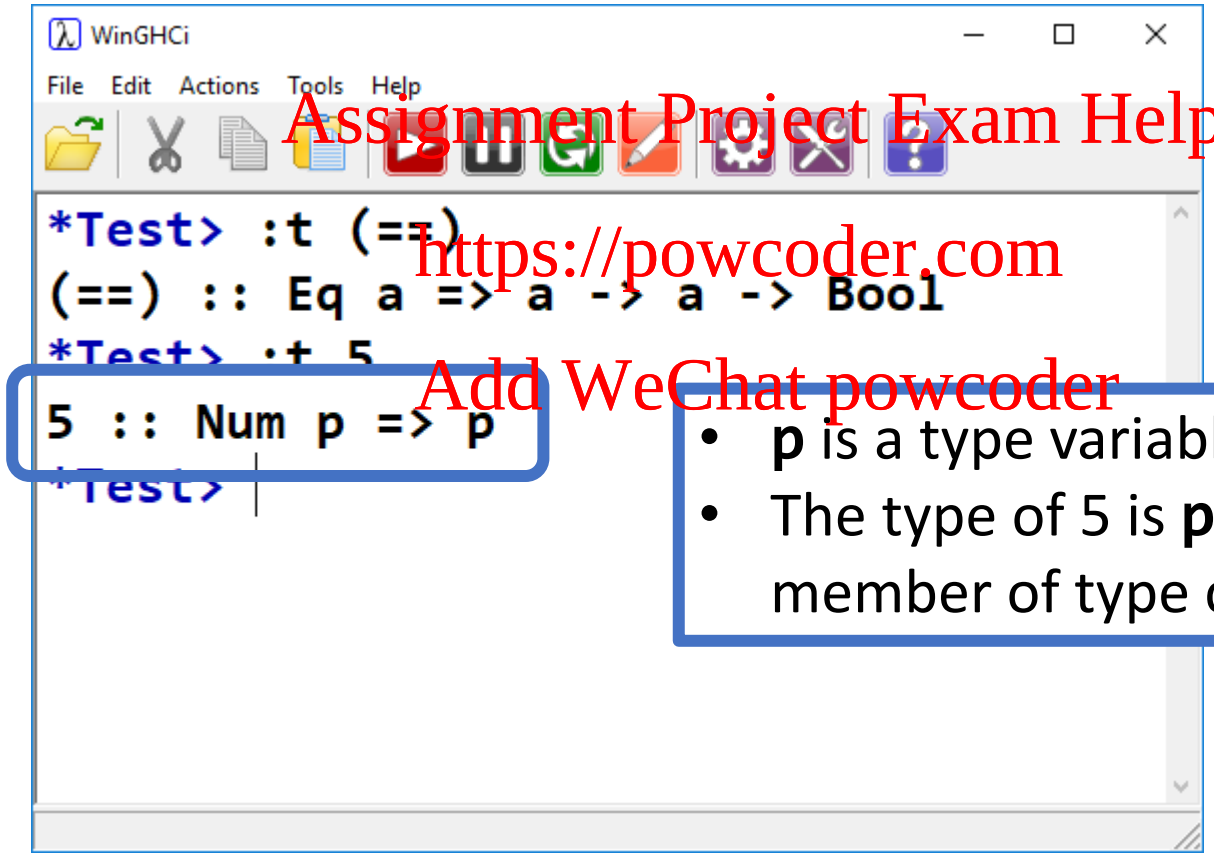
```
Prelude> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
   -- Defined in ‘GHC.Classes’
instance Eq a => Eq [a] -- Defined in ‘GHC.Classes’
instance Eq Word -- Defined in ‘GHC.Classes’
instance Eq Ordering -- Defined in ‘GHC.Classes’
instance Eq Int -- Defined in ‘GHC.Classes’
instance Eq Float -- Defined in ‘GHC.Classes’
instance Eq Double -- Defined in ‘GHC.Classes’
instance Eq Char -- Defined in ‘GHC.Classes’
instance Eq Bool -- Defined in ‘GHC.Classes’
```

Num Type Class



- This allows numeric values freedom to be an integer or floating point as the compiler sees fit.
- **Num** class contains all numbers, and certain operations over them such as addition.

Num Type Class



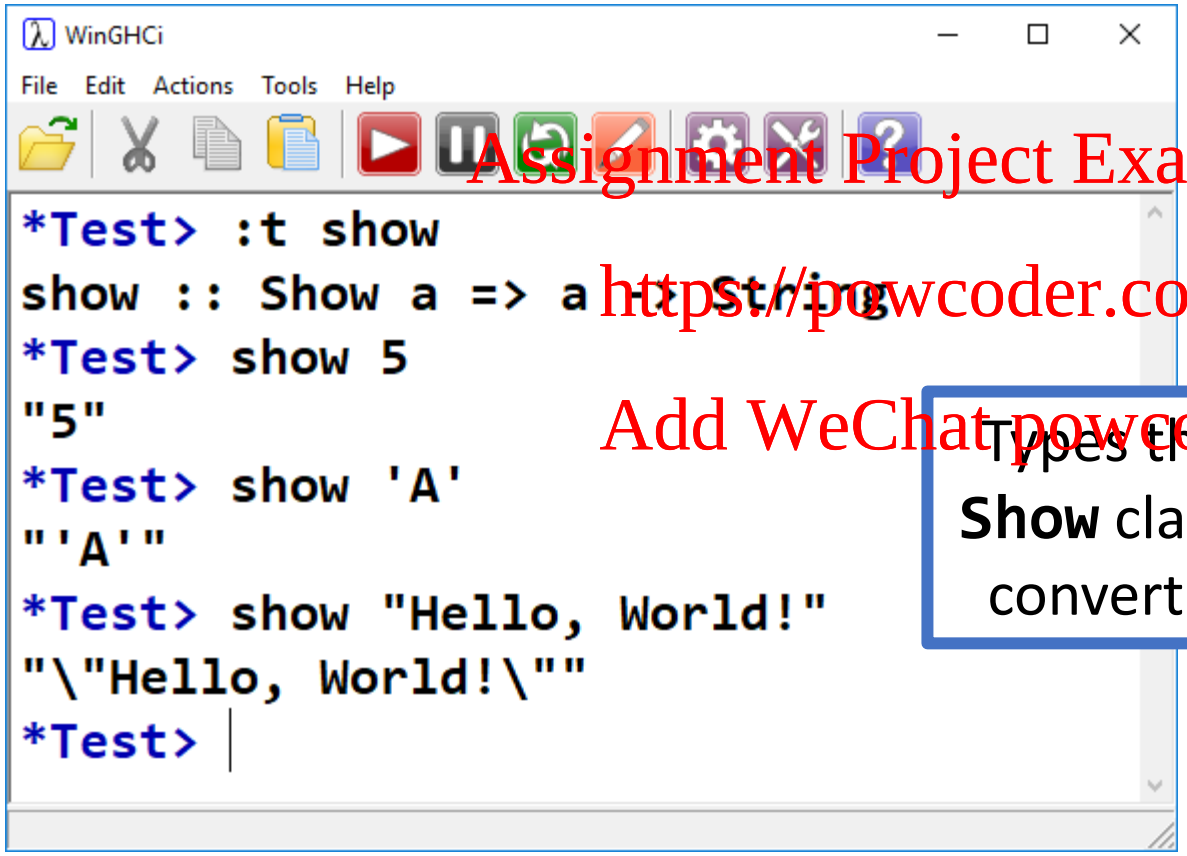
- **p** is a type variable
- The type of 5 is **p**, and **p** is a member of type class **Num**

WinGHCi

File Edit Actions Tools Help

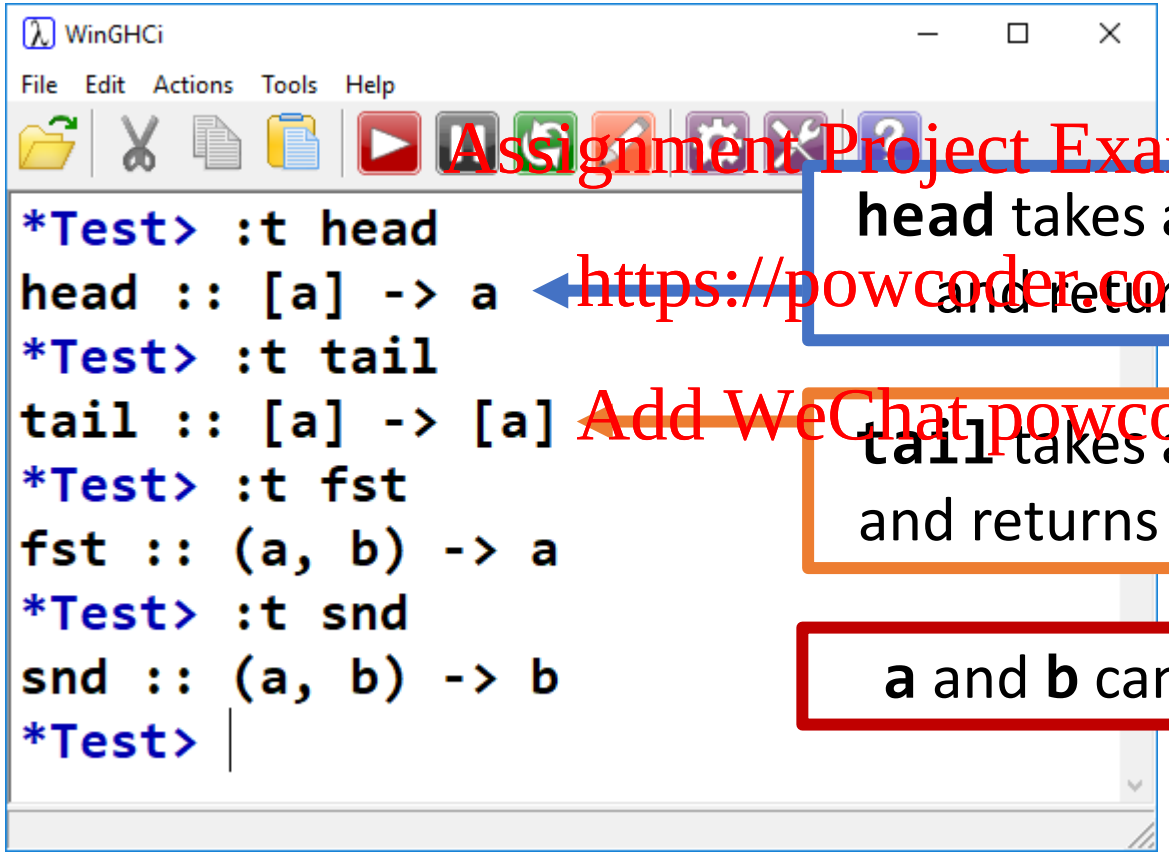
```
Prelude> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)
) #-}
   -- Defined in ‘GHC.Num’
instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
```

Show Type Class



Types that are members of the **Show** class have functions which convert their value to a String.

Function Types

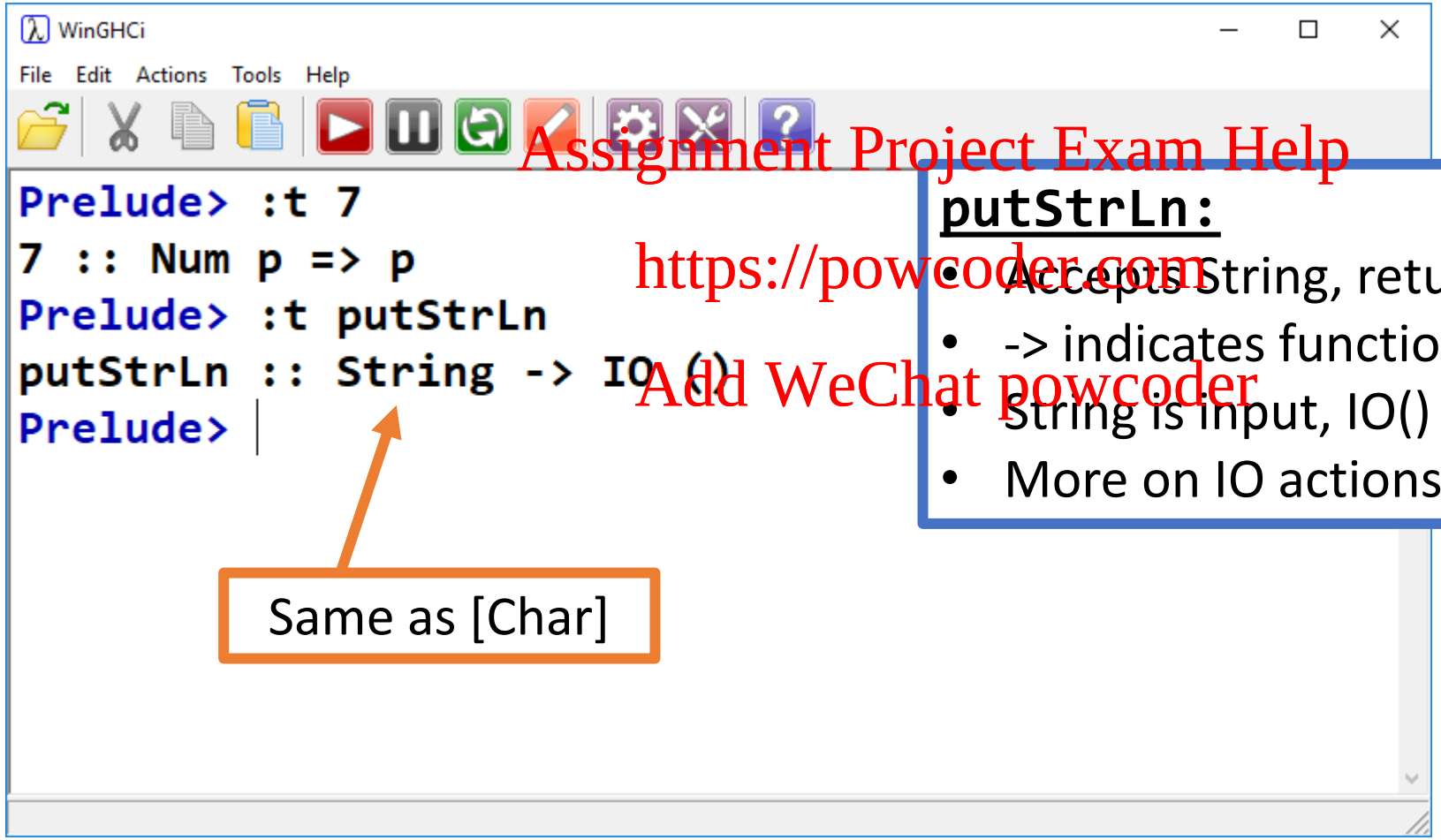


head takes a list containing type **a**, and returns a value of type **a**

tail takes a list containing type **a**, and returns a list containing type **a**

a and **b** can be *literally any type!*

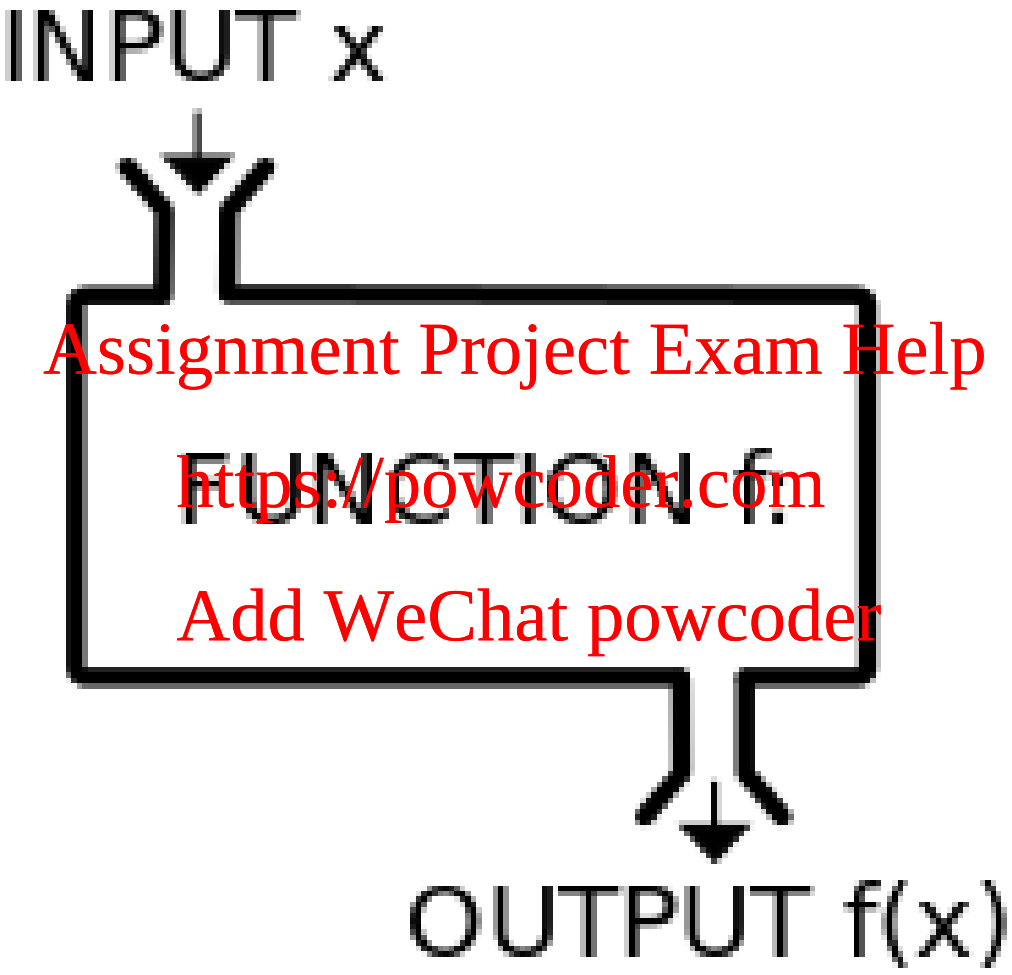
Function Types



putStrLn:

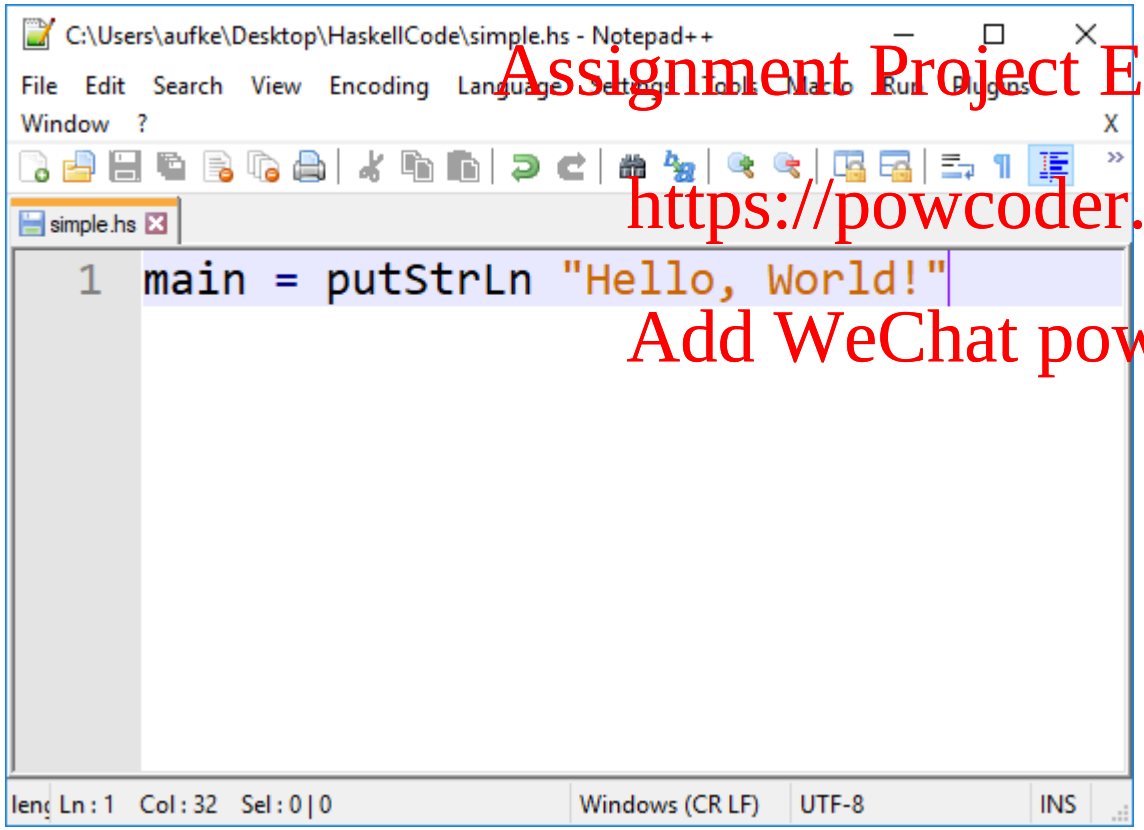
- Accepts String, returns ***IO action***.
- -> indicates function
- String is input, IO() is output.
- More on IO actions later.

We'll create our own types soon, and see how to add them to existing type classes.



Functions in Haskell

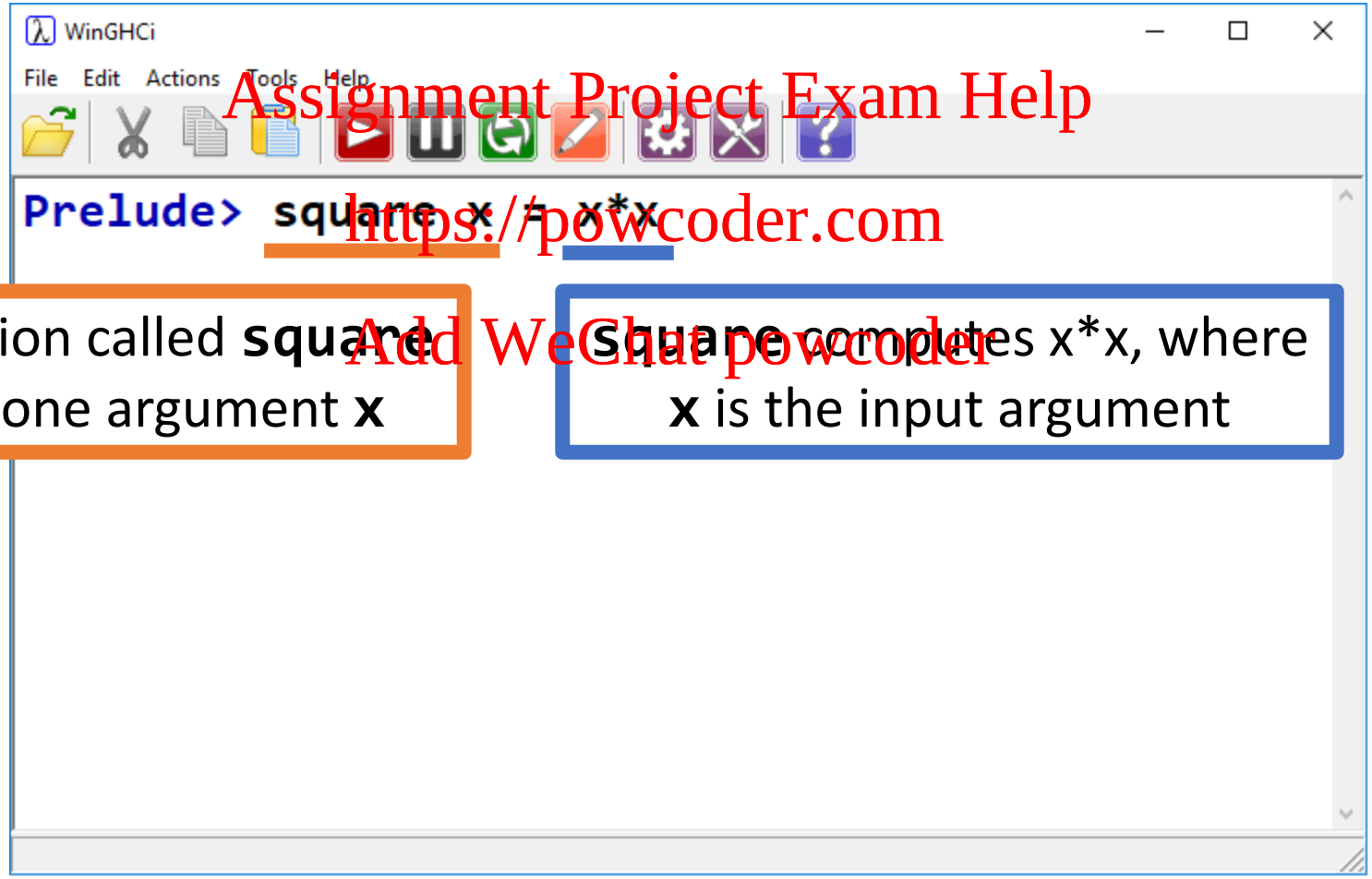
As expected of a pure functional language, functions are central in Haskell



- If we're compiling our code into an executable, we need a main.
- If we're using the GHCi shell, we don't.

Functions in Haskell

Let's start simple:

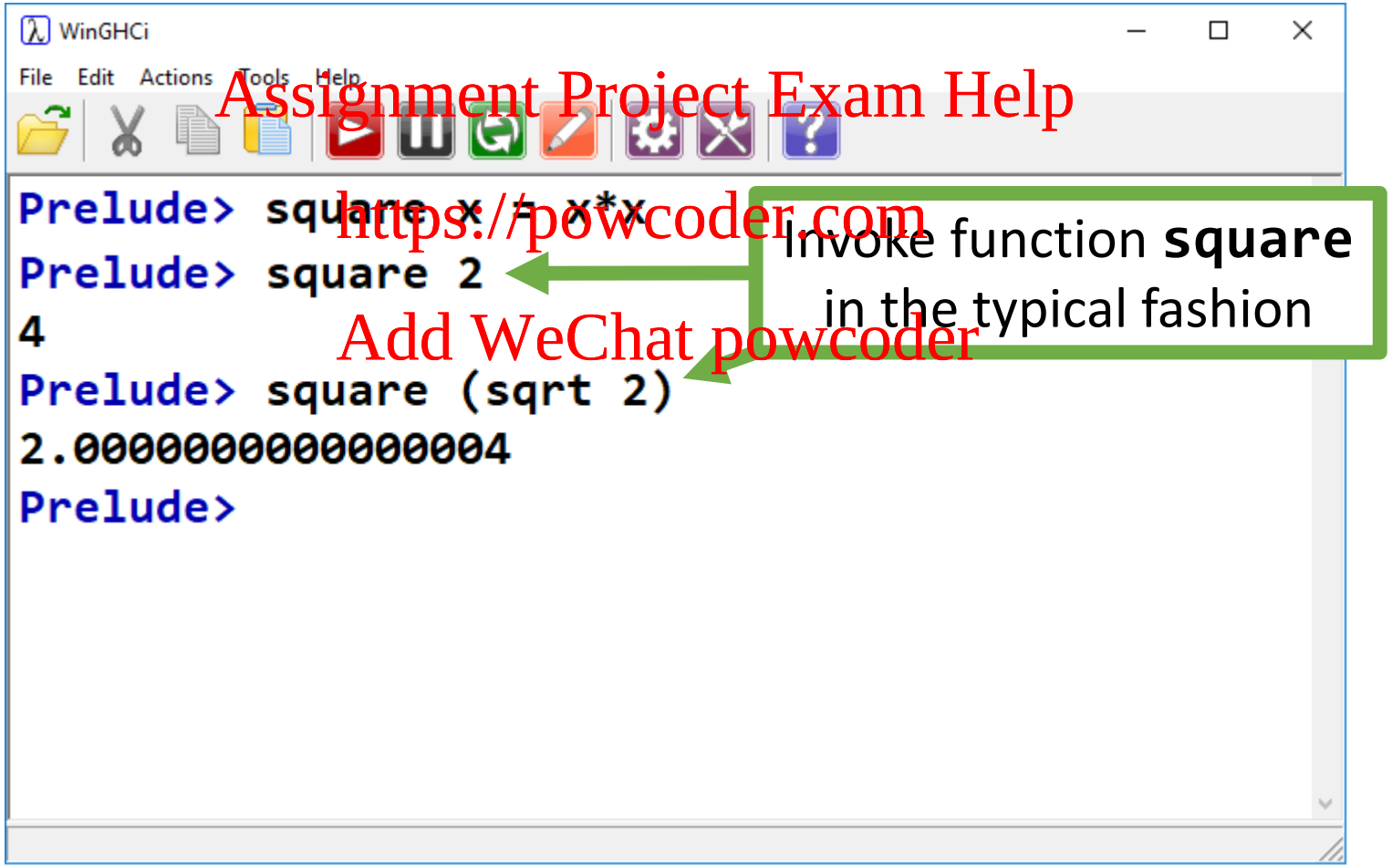


Define function called **square** that takes one argument **x**

square computes x*x, where **x** is the input argument

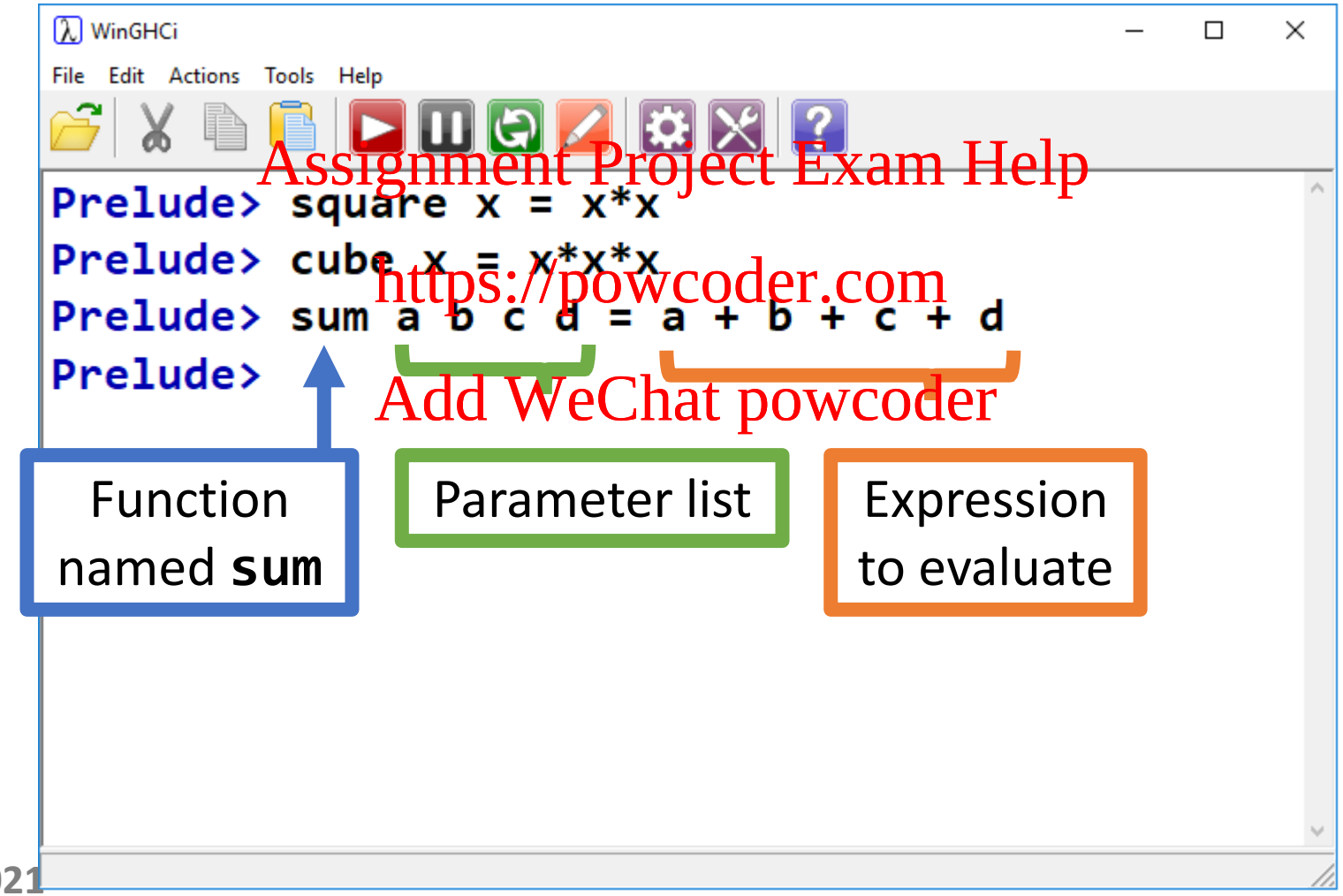
Functions in Haskell

Let's start simple:

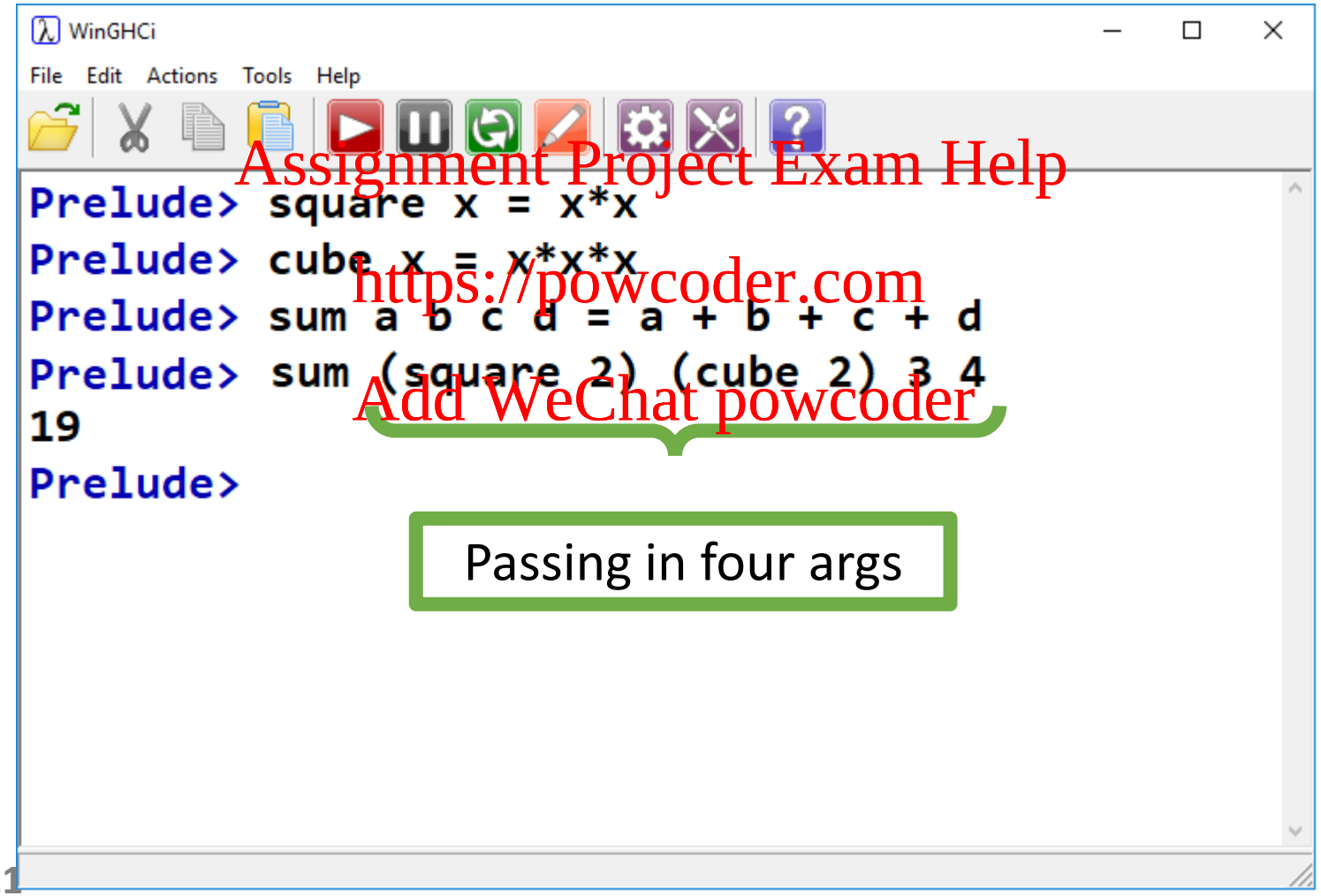


Invoke function **square** in the typical fashion

Functions in Haskell



Functions in Haskell



Functions in Haskell

WinGHCi

File Edit Actions Tools Help

```
GHCi, version 8.4.2: http://www.haskell.org/ghc/  :? for help
Prelude> square x = x*x
Prelude> sum a b c = a+b+c
Prelude> :t square
square :: Num a => a -> a
Prelude> :t sum
sum :: Num a => a -> a -> a -> a
Prelude>
```


- Based on what we're doing in **`square`** and **`sum`** (multiplying and adding)…
- Haskell determined that input and output type should be instances of typeclass **`Num`**.
- **`(+)`** and **`(*)`** are both defined for all types in typeclass **`Num`**.

Haskell Modules

This is getting tedious to type interactively.

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs HaskellList.hs

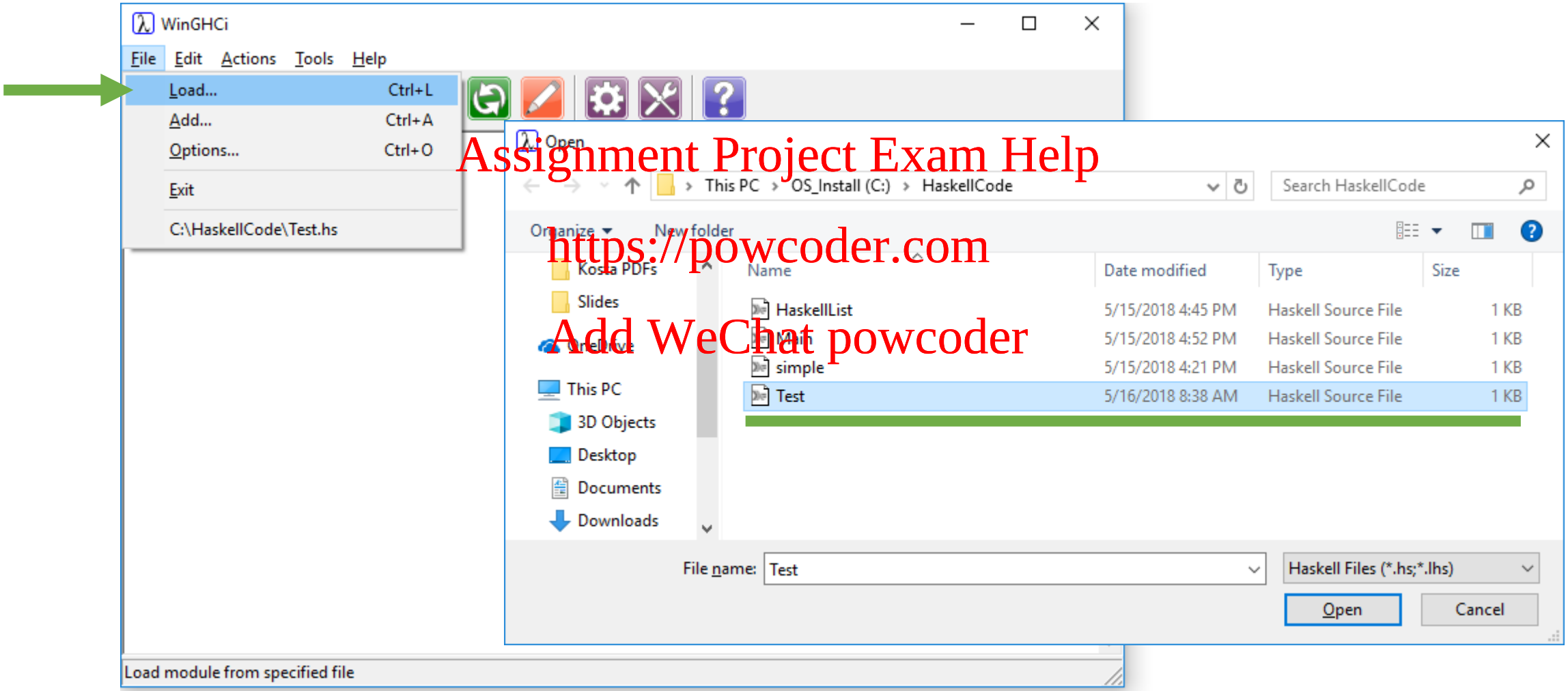
```haskell
module Test where

 square x = x*x
 cube x = x*x*x
 sum x y = x + y
```

Ln : 8 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Let's create a module!

- Similar to modules in Elixir
- We can load the module in GHCi
- Access its functions and expressions

Loading a Module



WinGHCi

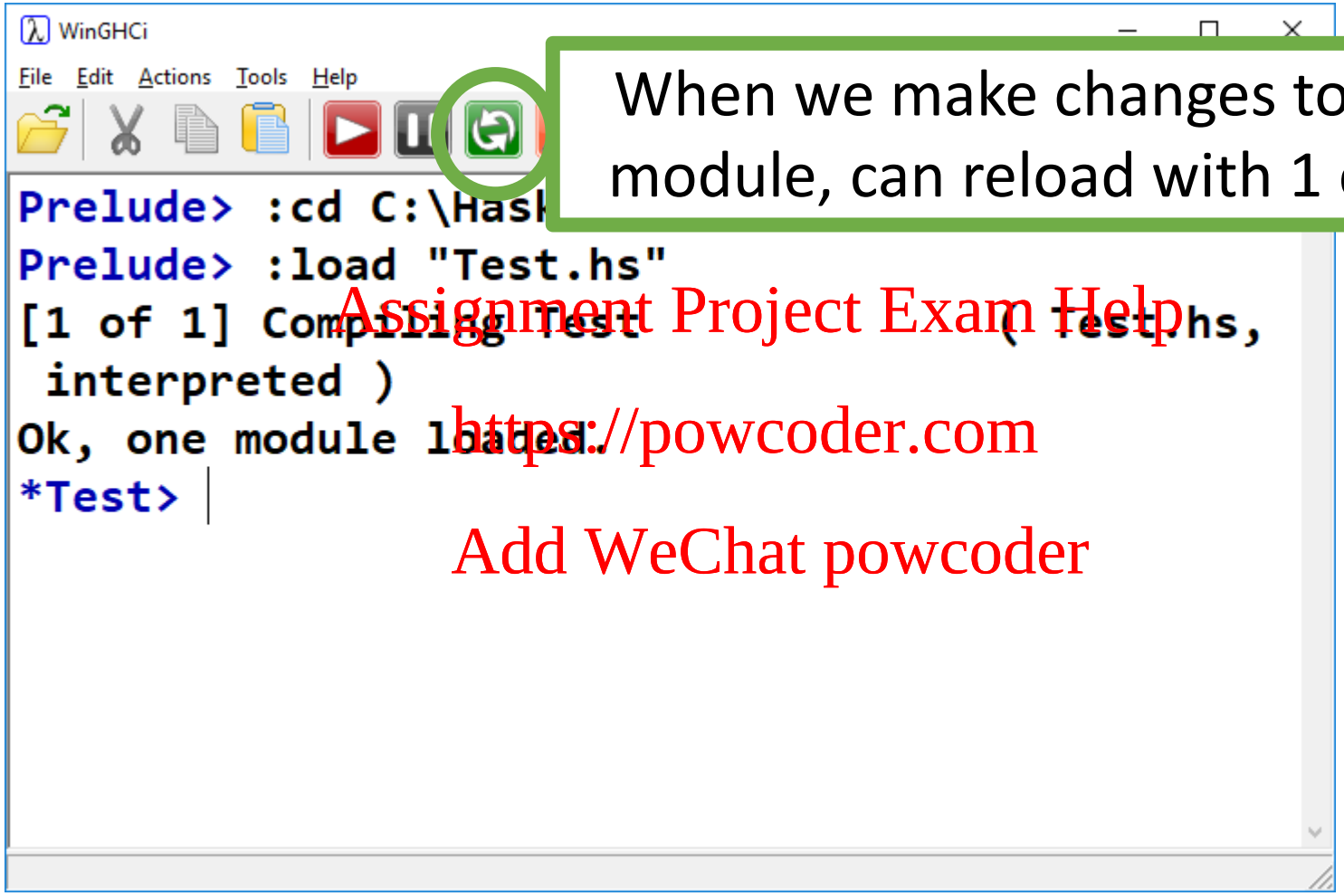
File Edit Actions Tools Help

```
Prelude> :cd C:\HaskellCode
Prelude> :load "Test.hs"
[1 of 1] Compiling Test             ( Test.hs,
 interpreted )
Ok, one module loaded.
*Test>
```


C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?


```
module Test where

square x = x*x
cube x = x*x*x
sum x y = x + y
```



When we make changes to Test module, can reload with 1 click!

Loading a Module

Use **`:load`** in terminal GHCi:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\aufke\Google Drive\Teaching\CCPS 506\Resources\Code\Haskell> ghci
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
Prelude> :load Test.hs
[1 of 1] Compiling Test             ( Test.hs, interpreted )
Ok, one module loaded.
*Test> cube 5
125
*Test> square 10
100
*Test> :t square
square :: Num a => a -> a
*Test>
```

Control Structures

`if then else`

WinGHCi

File Edit Actions Tools Help


```
Prelude> f x = if x < 0 then 0 else 1
Prelude> f 67
1
Prelude> f 0
1
Prelude> f (-6)
0
Prelude> f -6

<interactive>:203:1: error:
    • Non type-variable argument in the constr
aint: Num (a -> p)
```


- Brackets required around negative arguments
- Otherwise it thinks you're subtracting 6 from f

Control Structures

`if then else if then else`

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run

Test.hs | HaskellList.hs

```haskell
module Test where

 sign x =
  if x < 0 then -1
  else if x > 0 then 1
  else 0
```

Ln : 8 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

- Here we have a function named **`sign`**, that takes one argument x
- **It returns:**
  - -1 if x is negative
  - 1 if x is positive
  - 0 if x is 0

- If/else construct in Haskell is similar to most other languages.
- It must include a **then** and an **else**

Control Structures

`if then else if then else`

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 square x = x*x
 cube x = x*x*x
 sum x y = x + y

 sign x =
  if x < 0 then -1
  else if x > 0 then 1
  else 0
```

Ln : 13 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

We can now format across multiple lines.
HOWEVER: Indentation matters in Haskell!

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 sign x =
 if x < 0 then -1
 else if x > 0 then 1
 else 0
```

length : 92 lines : 9 Ln : 6 Col : 2 Sel : 0 | 0

WinGHCi

File Edit Actions Tools Help

```
Prelude> :reload

Test.hs:4:2: error:
    parse error (possibly incorrect indentatio
n or mismatched brackets)
  |
4 |  if x < 0 then -1   |  ^
[1 of 1] Compiling Test             ( Test.hs,
 interpreted )
Failed, no modules loaded.
Prelude>
```

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 sign x =
  if x < 0 then -1
  else if x > 0 then 1
  else 0
```

length : 95 lines : 9 Ln : 4 Col : 3 Sel : 0 | 0

WinGHCi

File Edit Actions Tools Help

```
n or mismatched brackets)
  |
4 |  if x < 0 then -1    |  ^
[1 of 1] Compiling Test              ( Test.hs,
 interpreted )
Failed, no modules loaded.
Prelude> :reload
[1 of 1] Compiling Test              ( Test.hs,
 interpreted )
Ok, one module loaded.
*Test>
```

Indenting in Haskell

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```
1  module Test where
2
3  sign x =
4   if x < 0 then -1
5   else if x > 0 then 1
6   else 0
7
8
9
```


Golden Rule:
Code which is part of some expression should be indented further than the beginning of that expression

https://en.wikibooks.org/wiki/Haskell/Indentation

length : 95 lines : 9 | Ln : 8 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

If all that weren’t enough, Tabs don’t work properly unless they’re 8 spaces exactly.

Local Names in Functions

*C:_cps506\haskell\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X

Test.hs

```
1  module Test where
2
3   sign x = do
4    let q = x
5    if q < 0 then -1
6    else if q > 0 then 1
7    else 0
8
9
10
```

length : 114 lin Ln : 4 Col : 12 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

When binding a name inside a function or module, we use the "`let`" keyword

Multiple Expressions

*C:_cps506\haskell\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X

Test.hs

```haskell
module Test where

 sign x = do
  let q = x
  if q < 0 then -1
  else if q > 0 then 1
  else 0
```

length : 114 lin Ln : 4 Col : 12 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

- We now have two expressions in this function!
  - `if/else` and `let`
- Must add the `do` keyword

*C:_cps506\haskell\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X

Test.hs

```haskell
module Test where

 sign x = do
  let q = x
  if q < 0 then -1
  else if q > 0 then 1
  else 0
```

length : 114 lin Ln : 4 Col : 12 Sel : 0 | 0 Window

WinGHCi

File Edit Actions Tools Help

```
*Test> :reload
Ok, one module loaded.
*Test> sign 0
0
*Test> sign 42
1
*Test> sign (-1)
-1
*Test>
```

Case Statement

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs


```haskell
module Test where

 isNum x =
  case x of
   0 -> 0
   1 -> 1
   2 -> 2
   _ -> -1

 sign x = do
  let q = x
  if x < 0 then -1
  else if x > 0 then 1
```


- When matching specific values, a case construct is easier to write.
- Just like Elixir, the _ is wild. It catches everything else.

Length : 184 lines : 17 | Ln : 14 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

Case Statement

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run

Test.hs | HaskellList.hs

```haskell
module Test where

 isNum x =
  case x of
   0 -> 0
   1 -> 1
   2 -> 2
   _ -> -1

 sign x = do
  let q = x
  if x < 0 then -1
  else if x > 0 then 1
```

length : 184 lines : 17 Ln : 14 Col : 2 Sel : 0 | 0

WinGHCi

File Edit Actions Tools Help

```
*Test> :reload
Ok, one module loaded.
*Test> isNum 0
0
*Test> isNum 1
1
*Test> isNum 2
2
*Test> isNum 999
-1
*Test>
```

Pattern Matching: Case

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins

Test.hs | HaskellList.hs

```haskell
module Test where

 chkClr rgb =
  case rgb of
   (255, _, _) -> "RED"
   (_, 255, _) -> "GREEN"
   (_, _, 255) -> "BLUE"
   (_, _, _)   -> "None"


 pos x = x >= 0
```

length : 874 li Ln : 11 Col : 4 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

WinGHCi

File Edit Actions Tools Help

```
*Test> chkClr (1, 2, 255)
"BLUE"
*Test> chkClr (155, 255, 55)
"GREEN"
*Test> chkClr (255, 255, 255)
"RED"
*Test> chkClr (55, 25, 2)
"None"
*Test>
```

Pattern Matching: Case

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plu

Test.hs | HaskellList.hs

```haskell
module Test where

 chkClr rgb =
  case rgb of
   (255, _, _)    -> "RED"
   (_, 255, _)    -> "GREEN"
   (_, _, 255)    -> "BLUE"
   (255, 255, _) -> "YELLOW"
   (255, _, 255) -> "MAGENTA"
   (_, 255, 255) -> "CYAN"
```

Will never match!

length : 971 lin Ln : 15 Col : 4 Sel : 0 | 0 Windows (CR LF) UTF-

WinGHCi

File Edit Actions Tools Help

```
*Test> :reload
[1 of 1] Compiling Test             (
Test.hs, interpreted )

Test.hs:8:4: warning: [-Woverlapping-p
atterns]
    Pattern match is redundant
    In a case alternative: (255, 255,
_) -> ...

  |
8 |    (255, 255, _) -> "YELLOW"    |
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins

Test.hs | HaskellList.hs

```haskell
module Test where

 chkClr rgb =
  case rgb of
   (255, 255, _) -> "YELLOW"
   (255, _, 255) -> "MAGENTA"
   (_, 255, 255) -> "CYAN"
   (255, _, _)   -> "RED"
   (_, 255, _)   -> "GREEN"
   (_, _, 255)   -> "BLUE"
   (_, _, _)     -> "None"
```

length : 971 lin Ln : 14 Col : 4 Sel : 0 | 0 Windows (CR LF) UTF-8

WinGHCi

File Edit Actions Tools Help

```
*Test> :reload
Ok, one module loaded.
*Test> chkClr (255, 5, 255)
"MAGENTA"
*Test> chkClr (255, 5, 55)
"RED"
*Test> chkClr (25, 255, 255)
"CYAN"
*Test> chkClr (25, 55, 5)
"None"
*Test>
```

Unlike Elixir…

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 chkClr rgb =
  case rgb of
   (255, _, _) -> "RED"
   (_, 255, _) -> "GREEN"
   (_, _, 255) -> "BLUE"
   x -> "None"
```

Try and be more general to catch anything that isn't a 3-tuple?

WinGHCi

File Edit Actions Tools Help

```
*Test> chkClr (3, 3, 3)
"None"
*Test> chkClr (3, 3)

<interactive>:406:8: error:
    • Couldn't match expected type
‘(Integer, Integer, Integer)’
                  with actual type
‘(Integer, Integer)’
    • In the first argument of ‘chk
Clr’, namely ‘(3, 3)’
      In the expression: chkClr (3,
 3)
```

Piecewise Functions

Just like Elixir's function signature pattern matching

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window

Test.hs | HaskellList.hs

```haskell
module Test where

fac 0 = 1
fac x = x*fac(x-1)

fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)


isNum x =
 case x of
```

WinGHCi

File Edit Actions Tools Help

```
*Test> fac 3
6
*Test> fac 5
120
*Test> fib 5
5
*Test> fib 9
34
*Test>
```


Makes recursion very easy!

Piecewise Functions

C:\HaskellCode\Test.hs - Notepad++

```haskell
module Test where

 chkAxis (0, _) = (0, 1)
 chkAxis (_, 0) = (1, 0)
 chkAxis (a, b) = (a, b)
```

- Return unit vector if point lies on axis
- Return input point otherwise

WinGHCi

```
*Test> chkAxis (1, 1)
(1,1)
*Test> chkAxis (8, 0)
(1,0)
*Test> chkAxis (42.3338, 0)
(1.0,0)
*Test>
```

Functions: Guards

C:\HaskellCode\Test.hs - Notepad++

```haskell
module Test where

cmp2 x y | x < y = "First is smaller"
         | x > y = "Second is smaller"
         | otherwise = "Equal"
```

WinGHCi

```
*Test> cmp2 1 2
"First is smaller"
*Test> cmp2 8 2
"Second is smaller"
*Test> cmp2 8 8
"Equal"
*Test>
```


- Matches input arguments to **x** and **y**
- Guards denoted with **|**
- **`otherwise`** is the same as saying True

Recursion

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Wi

Test.hs | HaskellList.hs

```haskell
module Test where

 llen [] = 0
 llen x  = 1 + llen(tail x)
```

Classic list length finder

WinGHCi

File Edit Actions Tools Help

```
*Test> :reload
Ok, one module loaded.
*Test> l = [1, 3, 2, 7, 5]
*Test> length l
5
*Test> llen l
5
*Test> llen []
0
*Test> llen [1]
1
*Test>
```

Built-in length function

Our own user function

Tail Recursion?

Less important in Haskell

- In Haskell, function call model is different
- Function calls don't necessarily create a new stack frame
- In practice, tail recursion not a big deal.

Recursion: cons

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins

Test.hs | HaskellList.hs

```haskell
module Test where

 llen [] = 0
 llen (xh:xt) = 1 + (llen xt)
```

Here we treat the input argument as a pair containing the head and tail of the list.

length : 623 lines : 47 Ln : 7 Col : 2 Sel : 0 | 0 Windows

WinGHCi

File Edit Actions Tools Help

```
*Test> :reload
Ok, one module loaded.
*Test> l = [1,6,8,2,4,6,7]
*Test> length l
7
*Test> llen l
7
*Test> llen []
0
*Test> llen [1]
1
*Test>
```

Recursion: `filter`

*C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings To

Test.hs HaskellList.hs

```
1  module Test where
2
3   pos x = x >= 0
4
5   filt p [] = []
6   filt p (xh:xt) =
7    if p xh then xh : filt p xt
8    else filt p xt
9
10
11
12
13
```

Length : 132 Lines : 13 Ln : 13 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

- Returns true if x >= 0
- False otherwise

- First argument is a Boolean function
- Second input is a list
- Base case is if the list is empty
- Otherwise, we call the function p with the head of the list.
- If true, append it to the running list
- If false, do not append
- In both cases, make the recursive call with the tail.

Recursion: `filter`

*C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window

Test.hs | HaskellList.hs

```haskell
module Test where

 pos x = x >= 0

 filt p [] = []
 filt p (xh:xt) =
  if p xh then xh : filt p xt
  else filt p xt
```

Length : 132 Lines : 53 Ln : 13 Col : 2 Sel : 0 | 0 Windows (CR LF)

WinGHCi

File Edit Actions Tools Help

```
*Test> pos 4
True
*Test> pos (-5)
False
*Test> l = [-1, 2, -3, 4, -5, 6]
*Test> filt pos l
[2,4,6]
*Test> filt pos [-1]
[]
*Test> filt pos []
[]
*Test>
```

Test **pos** function

Return Multiple *Things*?

Lists/tuples to the rescue!

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

roots a b c =
 ( (-b + sqrt(b*b - 4*a*c))/(2*a),
   (-b - sqrt(b*b - 4*a*c))/(2*a) )

fac 0 = 1
fac x = x*fac(x-1)

fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

Length : 270 Lines : 28 | Ln : 7 Col : 11 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

- Notice there is a lot of duplicate computation here.
- Add a local variable?

`let/in` Structure

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

roots a b c =
 let disc = sqrt(b*b - 4*a*c)
 in  ((-b + disc)/(2*a),
      (-b - disc)/(2*a))

fac 0 = 1
fac x = x*fac(x-1)

fib 0 = 0
fib 1 = 1
```

length : 387 lines : 29 | Ln : 10 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

- This is one expression (`let`/`in`)
- We don't need to add **do** keyword

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 roots a b c =
  let disc = sqrt(b*b - 4*a*c)
  in  ((-b + disc)/(2*a),
       (-b - disc)/(2*a))

 fac 0 = 1
 fac x = x*fac(x-1)

 fib 0 = 0
 fib 1 = 1
```

length : 387 lines : 29 Ln : 10 Col : 2 Sel : 0 | 0 Window

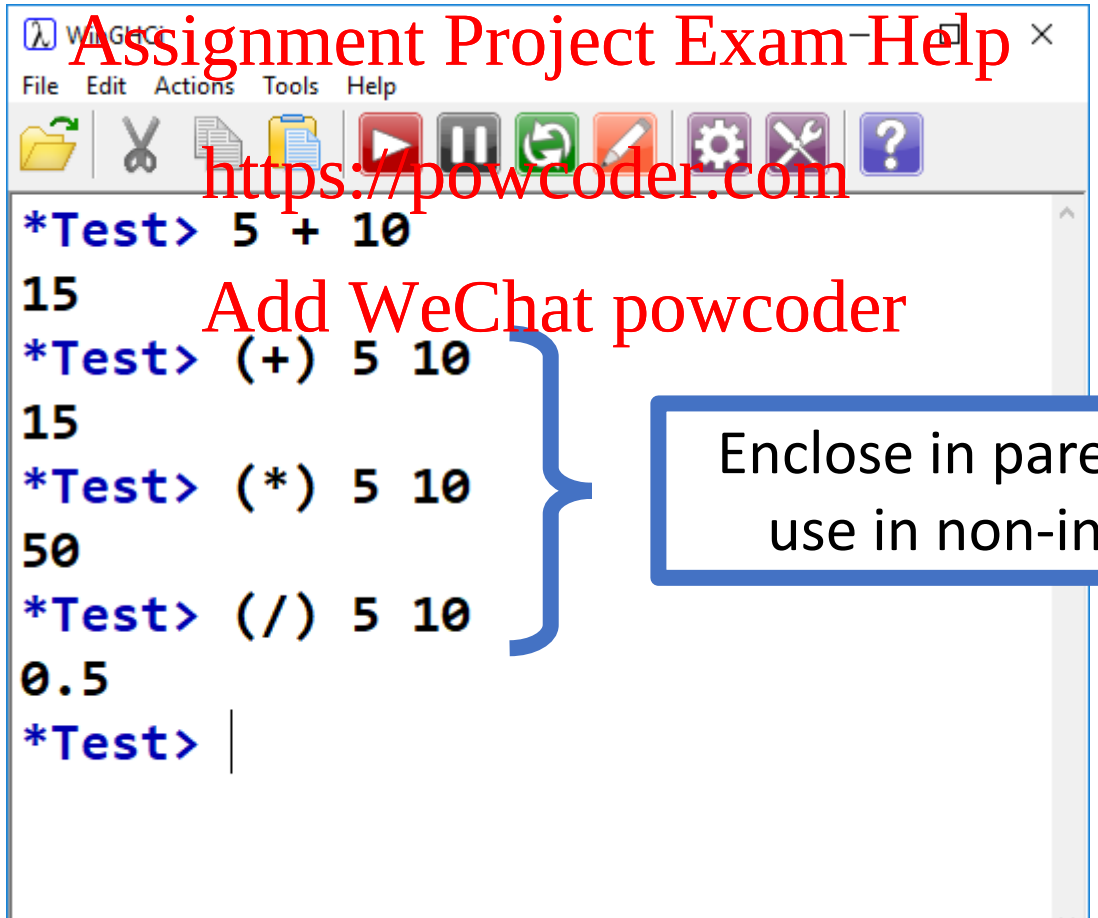
WinGHCi

File Edit Actions Tools Help

```
*Test> roots 1 2 (-6)
(1.6457513110645907,-3.6457513110645907)
*Test> roots 1 (-2) 4
(NaN,NaN)
*Test> roots (-1) 2 (-4)
(NaN,NaN)
*Test> roots (-1) 2 4
(-1.2360679774997898,3.2360679774997979)
*Test>
```

Infix Functions

Use symbolic operators as functions:



Enclose in parentheses to use in non-infix mode

Function Composition

WinGHCi

File Edit Actions Tools Help

```
*Test> fac(fib(4))
6
*Test> fac(fib(5))
120
*Test> fac(fib(6))
40320
*Test>
```

Can be written as:

WinGHCi

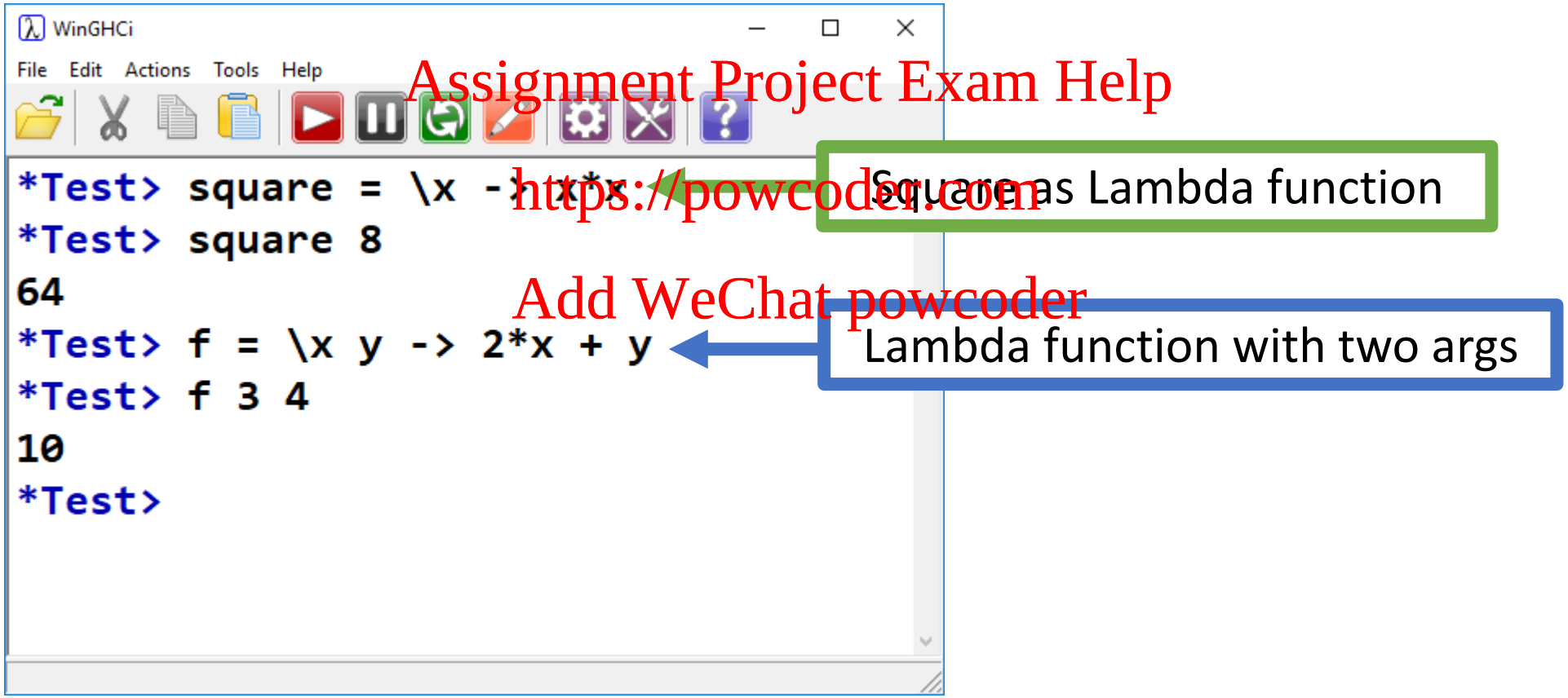
File Edit Actions Tools Help

```
*Test> (fac.fib) 4
6
*Test> (fac.fib) 5
120
*Test> (fac.fib) 6
40320
*Test>
```


In math, **f** ∘ **g** means "***f*** *following* ***g***". Same thing in Haskell.

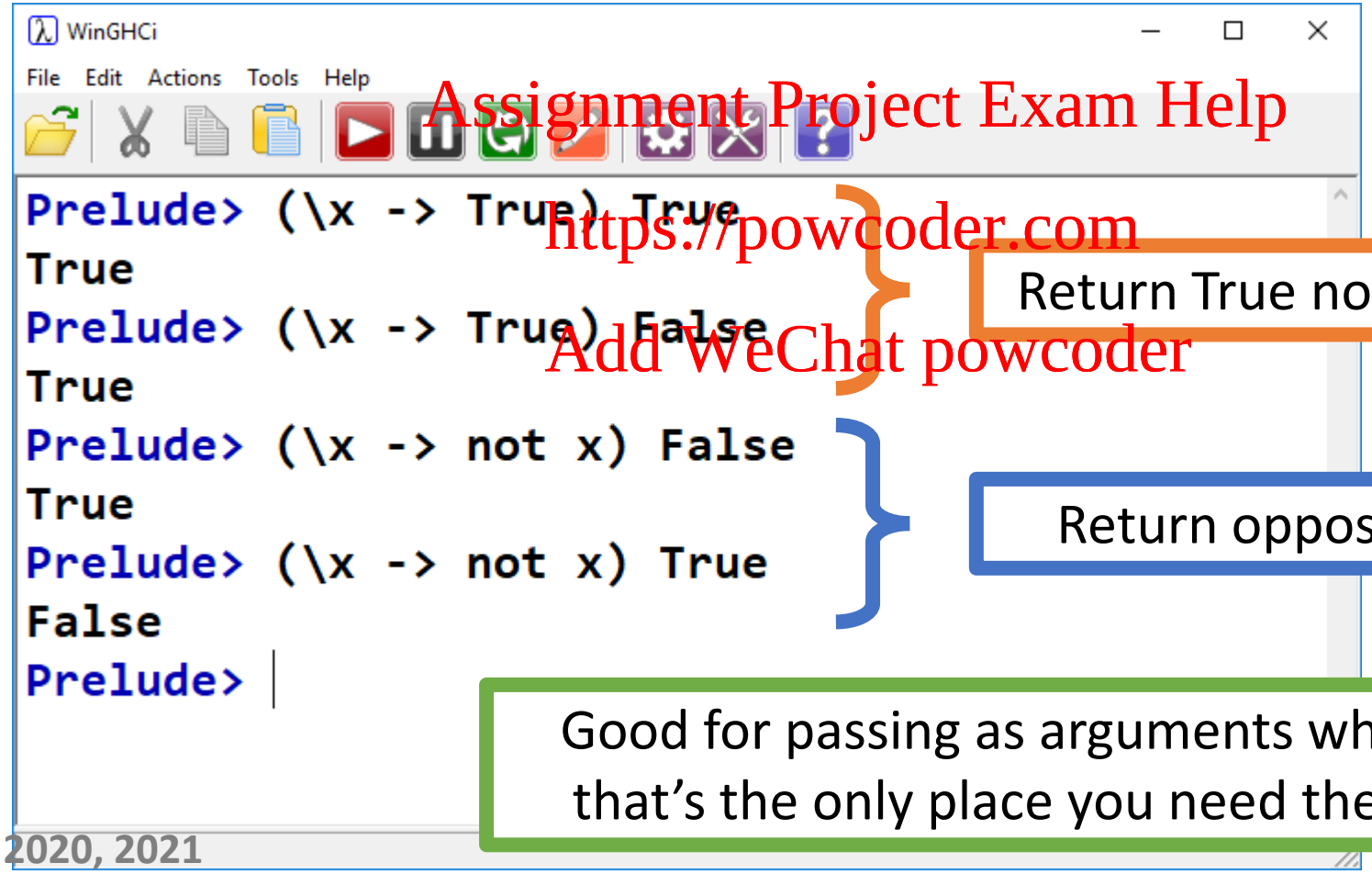
Lambda Functions

Like anonymous functions in Elixir:



Lambda Functions

They don't need names!

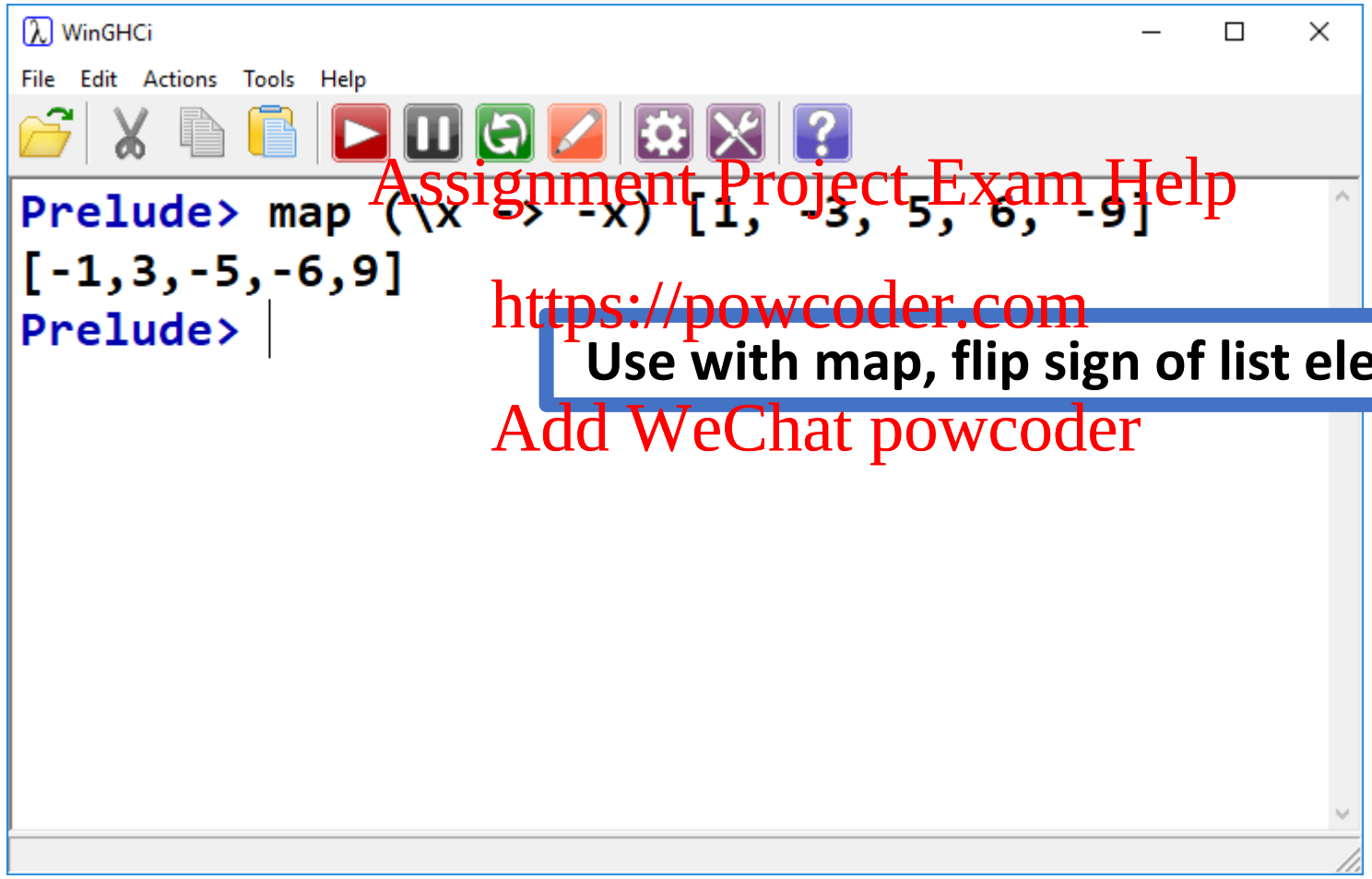


Return True no matter what

Return opposite of input

Good for passing as arguments when that's the only place you need them

Good for passing as arguments when that's the only place you need them



Use with map, flip sign of list elements

Comments

About time we mentioned them…

*C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 -- Single line comment, like // Java or C++
 -- Computes roots.
 roots a b c =
  let disc = sqrt(b*b - 4*a*c)
  in  ((-b + disc)/(2*a),
       (-b - disc)/(2*a))

 {- Block comment. Like /* */ in Java or C++ -}
 {- Compute factorial. Recursive. -}
 fac 0 = 1
 fac x = x*fac(x-1)
```

length : 542 lines : 33 | Ln : 14 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

Type Inference

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 chkClr rgb =
  case rgb of
   (255, _, _) -> "RED"
   (_, 255, _) -> "GREEN"
   (_, _, 255) -> "BLUE"
   x -> "None"
```

Remember this error?

length : 864 lin Ln : 11 Col : 4 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

WinGHCi

File Edit Actions Tools Help

```
*Test> chkClr (3, 3, 3)
"None"
*Test> chkClr (3, 3)

<interactive>:406:8: error:
    • Couldn't match expected type
‘(Integer, Integer, Integer)’
                  with actual type
‘(Integer, Integer)’
    • In the first argument of ‘chk
Clr’, namely ‘(3, 3)’
      In the expression: chkClr (3,
 3)
```

WinGHCi

File Edit Actions Tools Help

```
<interactive>:406:8: error:
    • Couldn't match expected type ‘(Integer, Integer, In
teger)’
                  with actual type ‘(Integer, Integer)’
    • In the first argument of ‘chkClr’, namely ‘(3, 3)’
      In the expression: chkClr (3, 3)
      In an equation for ‘it’: it = chkClr (3, 3)
```



```
*Test> :t chkClr
chkClr
  :: (Eq a1, Eq a2, Eq a3, Num a1, Num a2, Num a3) =>
     (a1, a2, a3) -> [Char]
*Test>
```

?

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X

Test.hs

```
1  modu
2
3   chk
4    ca
5     (
6     (
7     (
8     x
9
10
11
12  pos
```

length : 864 lin Ln :

WinGHCi

File Edit Actions Tools Help

```
<interactive>:406:8: err
    • Couldn't match exp
teger)'
                    with a
    • In the first argum
      In the expression:
      In an equation for
*Test> :t chkClr
chkClr
  :: (Eq a1, Eq a2, Eq a3, Num a1, Num a2, Num a3) =>
     (a1, a2, a3) -> [Char]
*Test>
```

Type of chkClr is inferred:

- Takes as input a tuple with three values
  - (a1, a2, a3)
- The type of each value in the tuple must be instances of type class **Eq** and **Num**.
- The output of chkClr is a character list.

Type Inference

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X

Test.hs | HaskellList.hs

```
1  module Test where
2
3   chkClr rgb =
4    case rgb of
5     (255, _, _) -> "RED"
6     (_, 255, _) -> "GREEN"
7     (_, _, 2
8     x -> "No
9
10
11
12  pos x = x
```

length : 864 lin Ln : 11 Col : 4 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Based on the contents of chkClr:

- Haskell determined that the output is [Char]
- The input is a 3-tuple whose elements must be instances of **Num** and **Eq**.

```
*Test> :t chkClr
chkClr
  :: (Eq a1, Eq a2, Eq a3, Num a1, Num a2, Num a3) =>
     (a1, a2, a3) -> [Char]
*Test>
```

Specify Function Type

C:\HaskellCode\Test.hs - Notepad++

```haskell
module Test where

 chkAxis :: (Float, Float) -> (Float, Float)
 chkAxis (0, _) = (0, 1)
 chkAxis (_, 0) = (1, 0)
 chkAxis (a, b) = (a, b)
```

WinGHCi

```
*Test> chkAxis (1, 0)
(1.0,0.0)
*Test> chkAxis (0, 4.5)
(0.0,1.0)
*Test> chkAxis (3, 4)
(3.0,4.0)
*Test> chkAxis (4.333, 0)
(1.0,0.0)
*Test> :t chkAxis
chkAxis :: (Float, Float) ->
(Float, Float)
*Test>
```

- **`chkAxis`** takes a pair-tuple of Floats as input, and returns the same as output.
- Instead of constants being of type Num or Fractional, they are treated as Floats

Specify Function Type

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window

Test.hs | HaskellList.hs

```haskell
module Test where

cmp2 :: Int -> Int -> [Char]
cmp2 x y | x < y = "First is smaller"
         | x > y = "Second is smaller"
         | otherwise = "Equal"
```

length : 1,265 lines Ln : 9 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

WinGHCi

File Edit Actions Tools Help

```
*Test> cmp2 1 2
"First is smaller"
*Test> cmp2 8 2
"Second is smaller"
*Test> cmp2 8 8
"Equal"
*Test>
```

Thoughts?

WinGHCi

File Edit Actions Tools Help

```
*Test> cmp2 1 2
"First is smaller"
*Test> cmp2 8 2
"Second is smaller"
*Test> cmp2 8 8
"Equal"
*Test> cmp2 1.1 1.2
```

WinGHCi

File Edit Actions Tools Help

```
*Test> cmp2 8 8
"Equal"
*Test> cmp2 1.1 1.2

<interactive>:46:6: error:
    • No instance for (Fractional Int) arising from
the literal ‘1.1’
    • In the first argument of ‘cmp2’, namely ‘1.1’
      In the expression: cmp2 1.1 1.2
      In an equation for ‘it’: it = cmp2 1.1 1.2
*Test>
```

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 --cmp2 :: Int -> Int -> [Char]
 cmp2 x y | x < y = "First is smaller"
          | x > y = "Second is smaller"
          | otherwise = "Equal"
```

length : 1,267 lines Ln : 9 Col : 2 Sel : 0 | 0

WinGHCi

File Edit Actions Tools Help

```
*Test> cmp2 1.1 1.2
"First is smaller"
*Test> :t cmp2
cmp2 :: Ord a => a -> a -> [Char]
*Test>
```


Ord is a type class:

- When we didn't explicitly define our types, Haskell inferred the type for us.
- Ord is a type class under which the operations used on our inputs are defined.
- I.e., comparison operators.

Type VS Type Class

C:\HaskellCode\Test.hs - Notepad++

```haskell
module Test where

cmp2 :: Int -> Int -> [Char]
cmp2 x y | x < y = "First is smaller"
         | x > y = "Second is smaller"
         | otherwise = "Equal"
```

- Int & Char are types, ***not*** type classes
- We can use the above notation

C:\HaskellCode\Test.hs - Notepad++

```haskell
module Test where

cmp2 :: Ord a => a -> a -> [Char]
cmp2 x y | x < y = "First is smaller"
         | x > y = "Second is smaller"
         | otherwise = "Equal"
```

- Ord is a type class, thus we specify that **a** is an instance of Ord
- **cmp2** accepts two instances of Ord as arguments.
- Ord contains many different types, **a** can be any of them

Ord Type Class

C:\HaskellCode\Test.hs - Notepad++

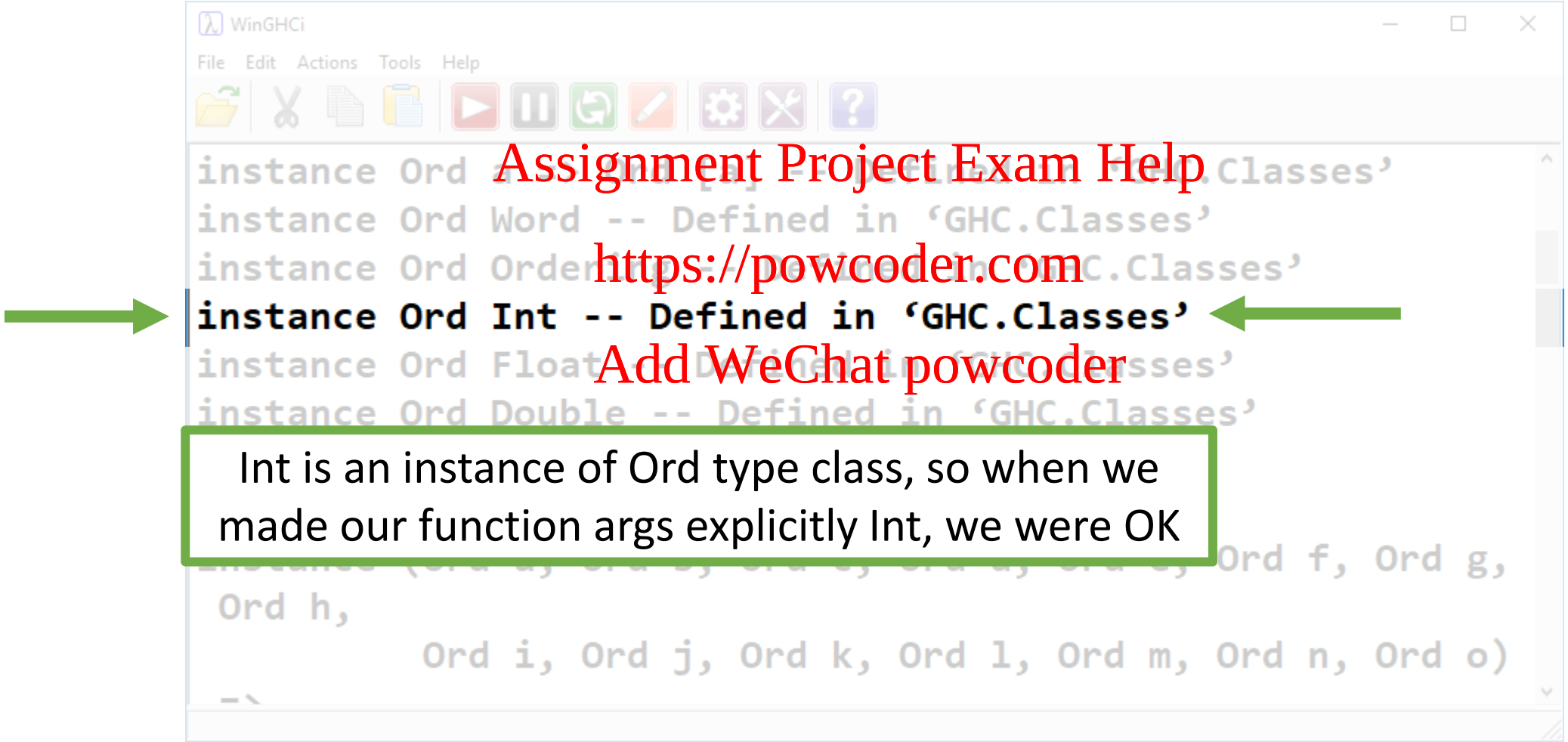
```haskell
module Test where

 cmp2 :: Ord a => a -> a -> [char
 cmp2 x y | x < y = "First is smal
          | x > y = "Second is sma
          | otherwise = "Equal"
```

WinGHCi

```haskell
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```


- Ord is a type class that supports comparison
- Comparison is all we're doing in our function
- Thus, Haskell infers types as Ord

Ord Type Class



Int is an instance of Ord type class, so when we made our function args explicitly Int, we were OK

How About This?

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Ru

Test.hs | HaskellList.hs

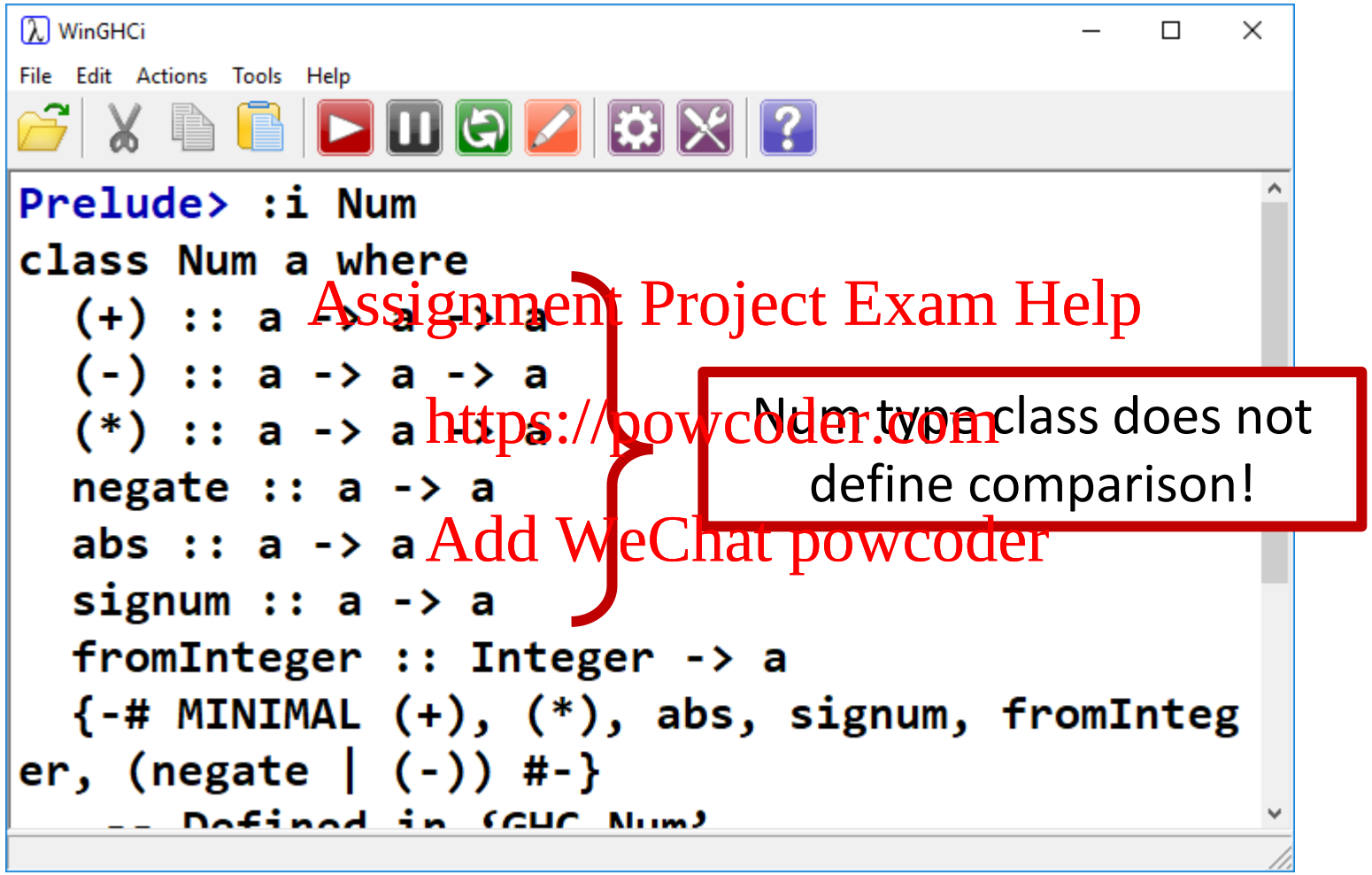
```
1  module Test where
2
3   cmp2 :: Num a => a -> a -> [C
4   cmp2 x y | x < y = "First is
5            | x > y = "Second is
6            | otherwise = "Equal
7
8
9
10
11
```

length : 1,269 | Ln : 10 Col : 2 Sel : 0 | 0 Windows (CR LF)

WinGHCi

File Edit Actions Tools Help

```
:reload
[1 of 1] Compiling Test             ( Tes
t.hs, interpreted )

Test.hs:3:4:error:
    • Could not deduce (Ord a) arising fr
om a use of ‘<’
      from the context: Num a
        bound by the type signature for:
                   cmp2 :: forall a. Num
a => a -> a -> [Char]
        at Test.hs:3:2-34
```



Num type class does not define comparison!

Hmmmm…

Num doesn't have comparison, **Ord** doesn't have addition

C:\HaskellCode\Test.hs - Notepad++

```haskell
module Test where

 cmp2 x y | x+1 < y+1 = "First is smaller"
          | x+1 > y+1 = "Second is smaller"
          | otherwise = "Equal"
```

WinGHCi

```
Prelude> :reload
[1 of 1] Compiling Test
   ( Test.hs, interpreted )
Ok, one module loaded.
*Test>
```

It compiled and loaded, what type did Haskell infer for x and y?

WinGHCi

File Edit Actions Tools Help

```
Prelude> :reload
[1 of 1] Compiling Test             ( Test.hs, interpreted )
Ok, one module loaded.
*Test> :t cmp2
cmp2 :: (Ord a, Num a) => a -> a -> [Char]
*Test>
```


Both!

- Whatever type we pass in (**a**), it must be an instance of both Ord and Num.
- **`Int`** is one such type, as is **`Float`**

Ord:

```
instance Ord Ordering -- Defined in ‘GHC.Classes’
instance Ord Int -- Defined in ‘GHC.Classes’
instance Ord Float -- Defined in ‘GHC.Classes’
instance Ord Double -- Defined in ‘GHC.Classes’
instance Ord Char -- Defined in ‘GHC.Classes’
instance Ord [a] -- Defined in ‘GHC.Classes’
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, O
```


Num:

```
    -- Defined in ‘GHC.Num’
instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
*Test>
```

Custom Data Types

Custom Data Types

- Lists and tuples are already quite powerful for organizing data
- What if we want to add custom behaviors over our data?
- For example, we can declare a pair tuple (1, 2).
- What if we want to treat these as coordinates and compute the sum? The dot product? Etc.?
- Addition is not defined for tuples, let alone more complicated operations.

Custom Coordinate Types

```
data Pt3 = Pt3 Float Float Float
```


Keyword indicating a custom type definition

Custom type name

Constructor for our custom type. To construct a Pt3, we need 3 values of type Float

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs HaskellList.hs

```haskell
module Test where

 data Pt3 = Pt3 Float Float Float
```

length : 1,263 lines : Ln : 7 Col : 2 Sel : 0 | 0

WinGHCi

File Edit Actions Tools Help

```
*Test> :t Pt3
Pt3 :: Float -> Float -> Float -> Pt3
*Test>
```

Custom Type Usage

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro

Test.hs | HaskellList.hs

```
1  module Test where
2
3   data Pt3 = Pt3 Float Float
4
5
6
7
```

length : 1,263 lines : Ln : 7 Col : 2 Sel : 0 | 0 Window

WinGHCi

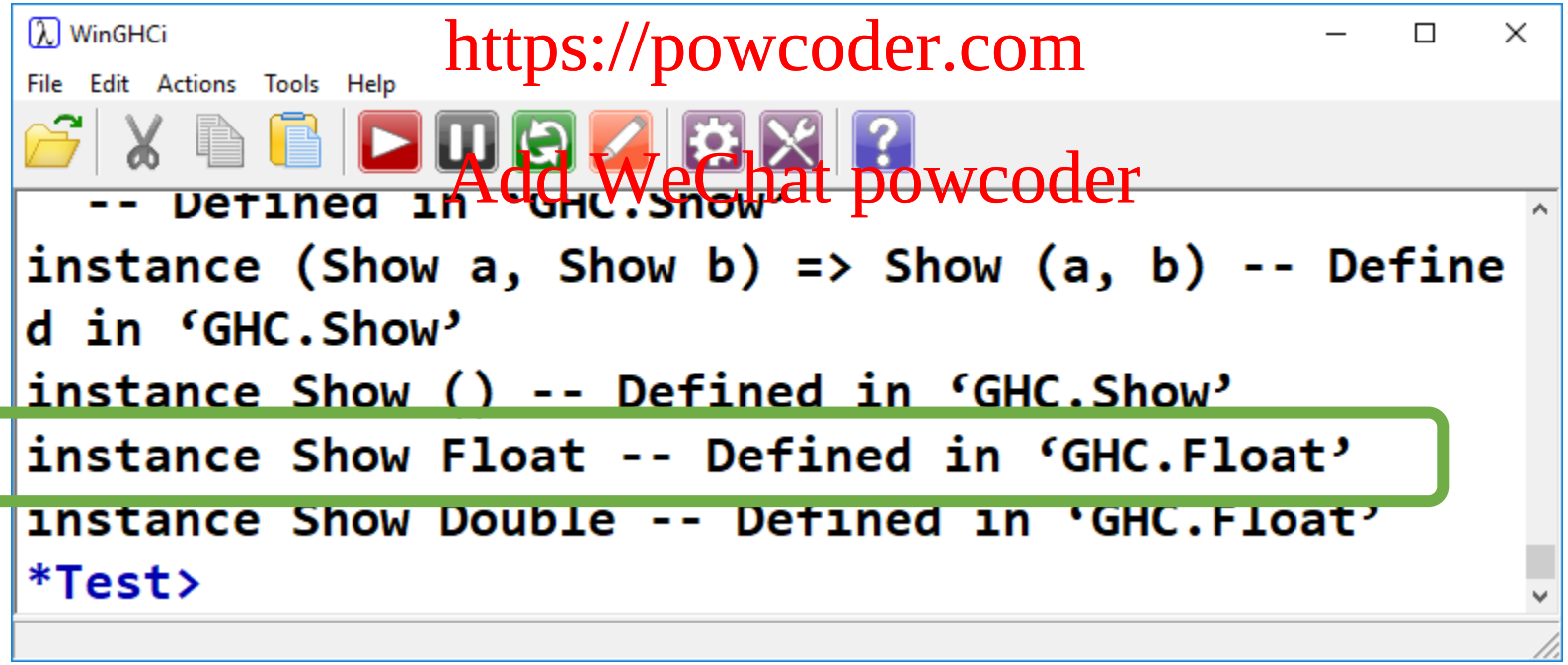
File Edit Actions Tools Help

```
*Test> :t Pt3
Pt3 :: Float -> Float -> Float -> Pt3
*Test> coord = Pt3 1 2 3
*Test> coord

<interactive>:508:1: error:
    • No instance for (Show (Pt3 a0)) arisi

                                  ractive GHCi comm

*Test>
```


Echoing a value in the interactive window requires its type to be an instance of Show!

Hmmm…

- The values contained in Pt3 are Float, and we know that Float is an instance of Show.
- How can we access the individual elements of Pt3?



C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt3 = Pt3 Float Float Float

 ptX (Pt3 x y z) = x
 ptY (Pt3 x y z) = y
 ptZ (Pt3 x y z) = z
```

WinGHCi

File Edit Actions Tools Help

```
*Test> coord = Pt3 1 2 3
*Test> ptX coord
1.0
*Test> ptY coord
2.0
*Test> ptZ coord
3.0
*Test>
```


- Three access functions, one for each of the three values.
- Take as arguments Pt3 (and by extension its three members)
- Return x, y, or z coordinate respectively.

Overloading Constructor

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float

 ptX (Pt3 x y z) = x
 ptY (Pt3 x y z) = y
 ptZ (Pt3 x y z) = z
```

length : 1,358 lines : 78 | Ln : 12 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

- Define Pt3 with three parameters
- Define Pt2 with two parameters
- Name of our data type is now simply Pt, because we have made it more generic.

There is now a problem with our access functions

There is now a problem with our access functions.

*C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```
1  module Test where
2
3   data Pt = Pt3 Float Float Float
4           | Pt2 Float Float
5
6   ptX (Pt2 x _) = x
7   ptX (Pt3 x _ _) = x
8
9   ptY (Pt2 _ y) = y
10  ptY (Pt3 _ y _) = y
11
12  ptZ (Pt3 _ _ z) = z
13
14
```

length : 1,404 lines : 8 Ln : 14 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Now our access functions work for both Pt2 and Pt3

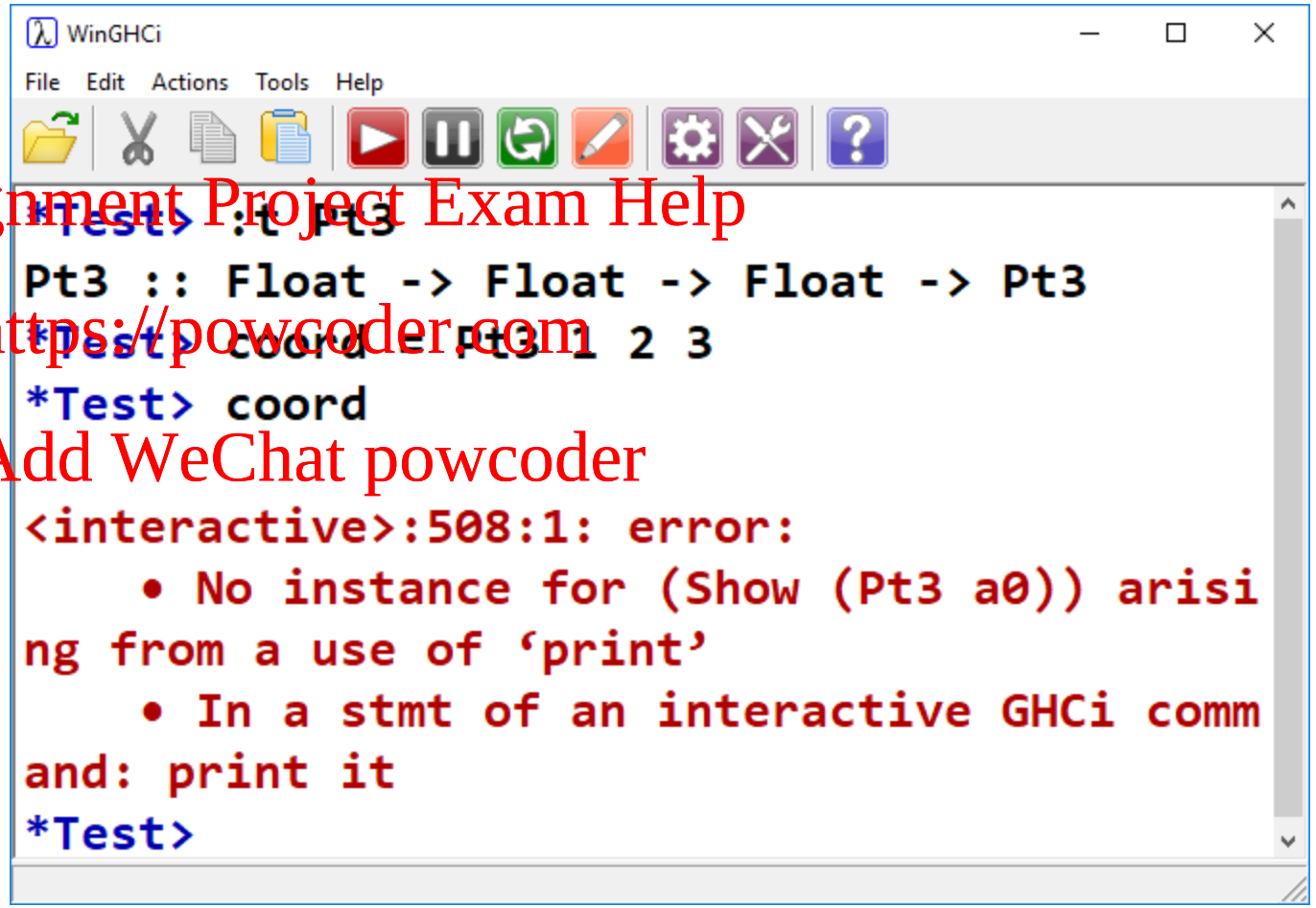
WinGHCi

File Edit Actions Tools Help

```
*Test> coord2 = Pt2 3 4
*Test> coord3 = Pt3 5 6 7
*Test> ptX coord2
3.0
*Test> ptX coord3
5.0
*Test> ptY coord3
6.0
*Test> ptY coord2
4.0
*Test>
```

Deriving Show

Recall:



Deriving Show

*C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins

Test.hs HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         deriving (Show)

ptY (Pt2    y) = y
```

length : 1,437 lines : 8 Ln : 8 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF

Our custom type will inherit some default display behavior from **Show**

WinGHCi

File Edit Actions Tools Help

```
*Test> c2 = Pt2 1 2
*Test> c3 = Pt3 5 6 7
*Test> c2
Pt2 1.0 2.0
*Test> c3
Pt3 5.0 6.0 7.0
*Test>
```

Similar to the toString() method in Java!

More Advanced Functions

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         deriving (Show)

 vecLen (Pt2 x y) = sqrt(x^2 + y^2)
 vecLen (Pt3 x y z) = sqrt(x^2 + y^2 + z^2)
```

Compute length of Pt2 and Pt3, treating them as vectors

length : 1,516 lines : 86 Ln : 14 Col : 21 Sel : 0 | 0 Windows (CR LF) UTF-8

WinGHCi

File Edit Actions Tools Help

```
*Test> c2 = Pt2 1 2
*Test> c3 = Pt3 5 6 7
*Test> vecLen c2
2.236068
*Test> vecLen c3
10.488089
*Test>
```

Addition, Subtraction, Equality?

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         deriving (Show)

 ptAdd (Pt2 x1 y1) (Pt2 x2 y2) =
  Pt2 (x1+x2) (y1+y2)

 ptSub (Pt2 x1 y1) (Pt2 x2 y2) =
  Pt2 (x1-x2) (y1-y2)

 ptEq (Pt2 x1 y1) (Pt2 x2 y2) =
  (x1 == x2) && (y1 == y2)
```

Ln : 17 Col : 27 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Let's add more functions!

- We can very easily define addition as the sum of each respective X and Y coord
- Likewise for subtraction and equality.

Addition, Subtraction, Equality?

C:\HaskellCode\Test.hs - Notepad++

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         deriving (Show)

 ptAdd (Pt2 x1 y1) (Pt2 x2 y2) =
  Pt2 (x1+x2) (y1+y2)

 ptSub (Pt2 x1 y1) (Pt2 x2 y2) =
  Pt2 (x1-x2) (y1-y2)

 ptEq (Pt2 x1 y1) (Pt2 x2 y2) =
  (x1 == x2) && (y1 == y2)
```

WinGHCi

```
*Test> c1 = Pt2 4 2
*Test> c2 = Pt2 (-1) 3
*Test> ptAdd c1 c2
Pt2 3.0 5.0
*Test> ptSub c1 c2
Pt2 5.0 (-1.0)
*Test> ptEq c1 c2
False
*Test> ptEq c1 (Pt2 4 2)
True
*Test>
```

Addition, Subtraction, Equality?

This seems very clunky. Why can't we simply add, subtract, or check equality with the symbolic operators (+, -, ==)?

We can! Equality is defined for instances of type class **Eq**

+, -, etc. are defined for instances of type class **Num**.

How do we make Pt2 and Pt3 instances of another type class?

Custom Types & Type Classes

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs


```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
        deriving (Show)


instance Eq Pt where
  (Pt2 x1 y1) == (Pt2 x2 y2) = (x1==x2 && y1==y2)
```


Haskell | length : 1,623 lines : 99 | Ln : 14 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

Define what it means for two Pt2 values to be considered equal

Declare **Pt** to be an instance of **Eq**

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Win

Test.hs | HaskellList.hs

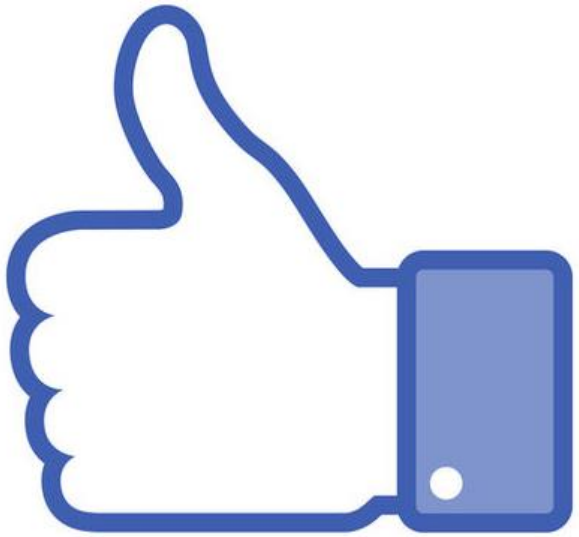
```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
        deriving (Show)



 instance Eq Pt where
  (Pt2 x1 y1) == (Pt2 x2 y2) = (x1==x2 && y1==y2)
```

Haskell | length : 1,623 lines : 99 | Ln : 14 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

WinGHCi

File Edit Actions Tools Help

```
*Test> Pt2 1 2 == Pt2 2 3
False
*Test> Pt2 1 2 == Pt2 1 2
True
*Test>
```

Minimal Definition

WinGHCi

File Edit Actions Tools Help

```
*Test> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
   -- Defined in ‘GHC.Classes’
instance [safe] Eq Pt -- Defined at Test.hs:7:11
instance (E
  -- Define
instance Eq
```


- The minimal definition for being an instance of **Eq** is **==** *OR* **/=** (not equal)
- We only defined ==

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         deriving (Show)


 instance Eq Pt where
  (Pt2 x1 y1) == (Pt2 x2 y2) = (x1==x2 && y1==y2)
```

length : 1,621 lines : 98 | Ln : 12 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8

WinGHCi

File Edit Actions Tools Help

```
*Test> c1 = Pt2 2 3
*Test> c2 = Pt2 2 4
*Test> c1 == c2
False
*Test> c1 /= c2
True
*Test>
```

Haskell is clever enough to derive /= from our definition of ==, and vice versa.

Let's Add /= Anyway

C:\HaskellCode\Test.hs - Notepad++

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         deriving (Show)


 instance Eq Pt where
  (Pt2 x1 y1) == (Pt2 x2 y2) = (x1==x2 && y1==y2)
  (Pt2 x1 y1) /= (Pt2 x2 y2) = not (x1==x2 && y1==y2)
```

WinGHCi

```
*Test> c1 = Pt2 2 3
*Test> c2 = Pt2 2 4
*Test> c1 == c2
False
*Test> c1 /= c2
True
*Test>
```

By The Way…

This should remind you of interfaces in Java

If you create a Java class that implements an interface, it must define all method signatures present in that interface

Most common is the **Comparable** interface

If our Java class implements Comparable, we must define some way of comparing two instances of our class

Instance of Num

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools

Test.hs HaskellList.hs

```
1  module Test where
2
3   data Pt = Pt3 Float Floa
4           | Pt2 Float Floa
5          deriving (Show)
6
7   instance Num Pt where
8    (Pt2 x1 y1) + (Pt2 x2 y
9
10
11
12
```

Hasl length : 1,765 lines : 104 Ln : 11 Col : 3 Sel : 0 | 0

WinGHCi

File Edit Actions Tools Help

```
[2 of 2] Compiling Test             ( Test.ns, in
terpreted )

Test.hs:7:11: warning: [-Wmissing-methods]
    • No explicit implementation for
        ‘*’, ‘abs’, ‘signum’, ‘fromInteger’, and
(either ‘negate’ or ‘-’)
    • In the instance declaration for ‘Num Pt’
  |
7 |  instance Num Pt where    |          ^^^^^^
Ok, one module loaded.
*Test>
```

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         deriving (Show)

 instance Num Pt where
  (Pt2 x1 y1) + (Pt2 x2 y2) = Pt2 (x1+x2) (y1+y2)
```

- We're only implementing for Pt2.
- Adding Pt3 follows the same pattern

WinGHCi

File Edit Actions Tools Help

```
*Test> Pt2 1 2 + Pt2 3 4
Pt2 4.0 6.0
*Test> x = Pt2 1 2
*Test> y = Pt2 6 7
*Test> x+y
Pt2 7.0 9.0
*Test>
```

WinGHCi

File Edit Actions Tools Help

```
*Test> Pt3 1 2 3 + Pt3 1 2 3
*** Exception: Test.hs:8:3-49: Non-ex
haustive patterns in function +

*Test>
```

Instance of Num

WinGHCi

File Edit Actions Tools Help

```
*Test> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
    -- Defined in 'GHC.Num'
instance [safe] Num Pt -- Defined at Test.hs:7:11
```

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         deriving (Show)

 instance Num Pt where
  (Pt2 x1 y1) + (Pt2 x2 y2) = Pt2 (x
  (Pt2 x1 y1) - (Pt2 x2 y2) = Pt2 (x
  (Pt2 x1 y1) * (Pt2 x2 y2) = Pt2 (x
  abs (Pt2 x1 y1) = Pt2 (abs x1) (abs y1)
  signum (Pt2 x1 y1) = Pt2 (signum x1) (signum y1)

 instance Eq Pt where
  (Pt2 x1 y1) == (Pt2 x2 y2) = (x1==x2 && y1==y2)
  (Pt2 x1 y1) /= (Pt2 x2 y2) = not (x1==x2 && y1==y2)
```

Haskell | length : 1,948 lines : 104 | Ln : 19 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

- This may look circular
- We're using abs and signum in our definition of abs and signum.
- However! x1 and y1 are Float.
- abs and signum *are* defined for Float
- We're defining them for Pt2

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         deriving (Show)

 instance Num Pt where
  (Pt2 x1 y1) + (Pt2 x2 y2) = Pt2 (x1+x2) (y1+y2)
  (Pt2 x1 y1) - (Pt2 x2 y2) = Pt2 (x1-x2) (y1-y2)
  (Pt2 x1 y1) * (Pt2 x2 y2) = Pt2 (x1*x2) (y1*y2)
  abs (Pt2 x1 y1) = Pt2 (abs x1) (abs y1)
  signum (Pt2 x1 y1) = Pt2 (signum x1) (signum y1)
  fromInteger n = let a = (fromInteger n) in Pt2 a a

 instance Eq Pt where
  (Pt2 x1 y1) == (Pt2 x2 y2) = (x1==x2 && y1==y2)
  (Pt2 x1 y1) /= (Pt2 x2 y2) = not (x1==x2 && y1==y2)
```

Haskell | length : 2,004 lines : 105 | Ln : 21 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

- `fromInteger` is a *coercion* function.
- Dictates how our custom type can be created from an Integer
- Takes an Integer, returns a Pt
- Lets us do this…

WinGHCi

File Edit Actions Tools Help

```
*Test> (Pt2 2 3) + 4
Pt2 6.0 7.0
*Test>
```

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         deriving (Show)

 instance Num Pt where
  (Pt2 x1 y1) + (Pt2 x2 y2) = Pt2 (x1+x2) (y1+y2)
  (Pt2 x1 y1) - (Pt2 x2 y2) = Pt2 (x1-x2) (y1-y2)
  (Pt2 x1 y1) * (Pt2 x2 y2) = Pt2 (x1*x2) (y1*y2)
  abs (Pt2 x1 y1) = Pt2 (abs x1) (abs y1)
  signum (Pt2 x1 y1) = Pt2 (signum x1) (signum y1)
  fromInteger n = let a = (fromInteger n) in Pt2 a a

 instance Eq Pt where
  (Pt2 x1 y1) == (Pt2 x2 y2) = (x1==x2 && y1==y2)
  (Pt2 x1 y1) /= (Pt2 x2 y2) = not (x1==x2 && y1==y2)
```

Haskell | length : 2,004 lines : 105 | Ln : 21 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

WinGHCi

File Edit Actions Tools Help

```
*Test> :reload
[1 of 1] Compiling Test
( Test.hs, interpreted )
Ok, one module loaded.
*Test>
```

No more warnings!

WinGHCi

File Edit Actions Tools Help

```
*Test> x = Pt2 1 (-2)
*Test> y = Pt2 3 4
*Test> x + y
Pt2 4.0 2.0
*Test> x - y
Pt2 (-2.0) (-6.0)
*Test> x * y
Pt2 3.0 (-8.0)
*Test> signum x
Pt2 1.0 (-1.0)
*Test> abs x
Pt2 1.0 2.0
*Test>
```

Instance of Show

In Java-speak, define our own toString(), instead of deriving the default

WinGHCi

File Edit Actions Tools Help


```
*Test> :i Show
class Show a where
  showsPrec :: Int -> a -> ShowS
  show :: a -> String
  showList :: [a] -> ShowS
  {-# MINIMAL showsPrec | show #-}
  -- Defined in 'GHC.Show'
instance [safe] Show Pt -- Defined at Test.hs:5:20
instance (Show a, Show b) => Show (Either a b)
  -- Defined in 'Data.Either'
instance Show a => Show [a] -- Defined in 'GHC.Show'
```


- The minimal definition for Show is easy
- Need to implement show OR showsPrec
- Let's do show
- Need to go from Pt2 to a String

C:\HaskellCode\Test.hs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

Test.hs | HaskellList.hs

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float
         --deriving (Show)

 instance Show Pt where
  show (Pt2 x y) =
    "< " ++ (show x) ++ ", " ++ (show y) ++ " >"
```

No longer need to derive Show, we've made our own

- Use string concatenation to create a pleasing visual output for Pt2
- In doing so, we make use of show as defined for Floats

WinGHCi

File Edit Actions Tools Help

```
*Test> x = Pt2 1 2
*Test> y = Pt2 3 4
*Test> x
< 1.0, 2.0 >
*Test> y
< 3.0, 4.0 >
*Test> x + y
< 4.0, 6.0 >
*Test> -x
< -1.0, -2.0 >
*Test> signum y
< 1.0, 1.0 >
*Test>
```

Haskell Tutorials/References:

https://en.wikibooks.org/wiki/Yet_Another_Haskell_Tutorial

http://cheatsheet.codeslower.com/CheatSheet.pdf