# C/CPS 506

## Comparative Programming Languages

**Prof. Alex Ufkes**

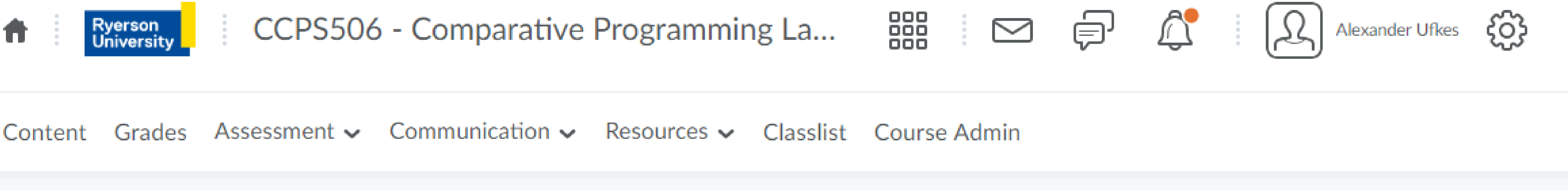
**Topic 7:** Types, type classes, custom types.

Ryerson University

# Notice!

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

# Course Administration (CCPS)



Haskell labs released today!

# Any Questions?

# Let’s Get Started!

# Types in Haskell

**Statically Typed:**

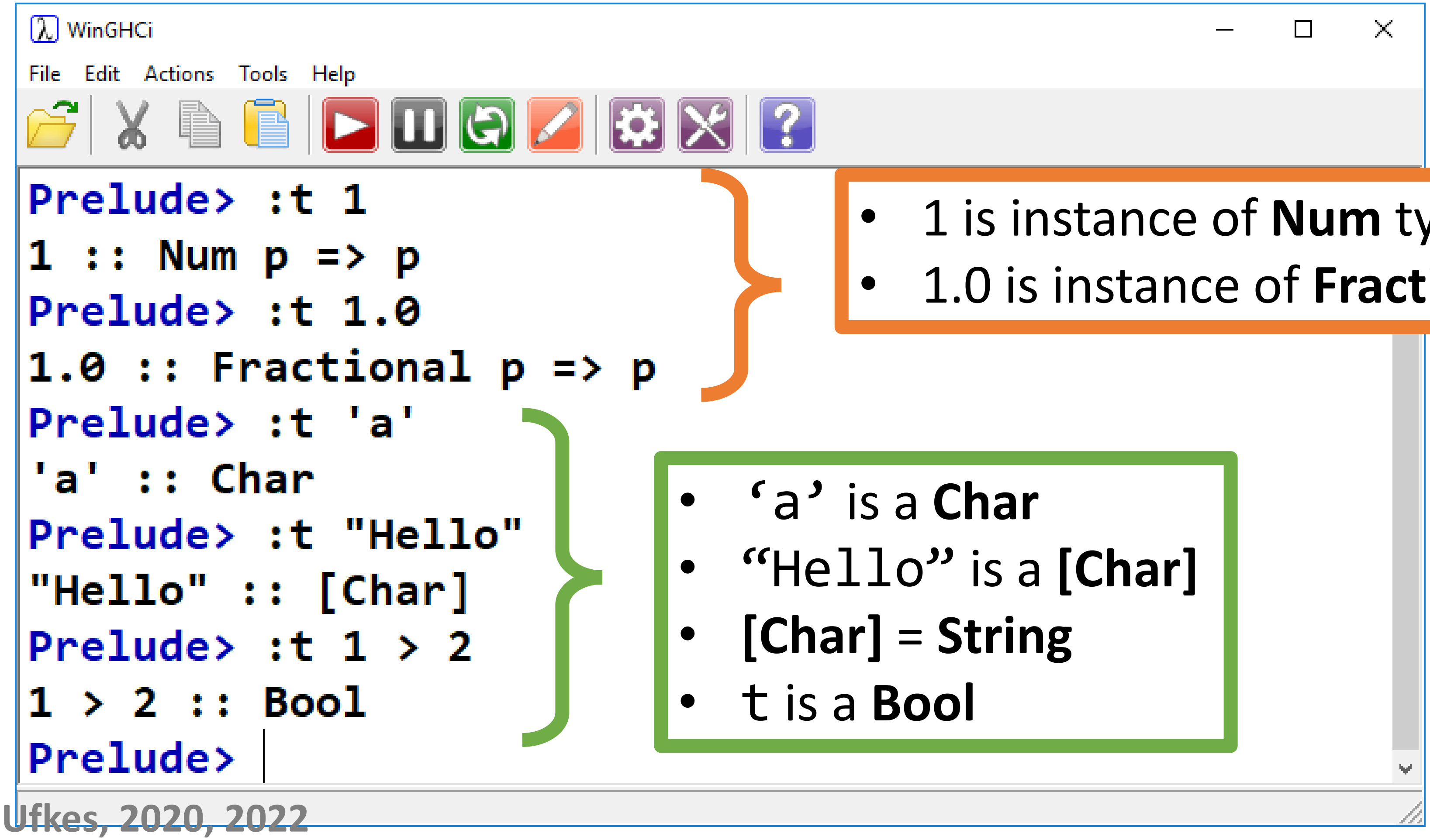
- Haskell uses static type checking.
- Every expression is assigned a type.
- If a function's arguments aren't the expected type, a compile error occurs.

**Type Inference**

- In Haskell, we need not specify type explicitly.
- 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:

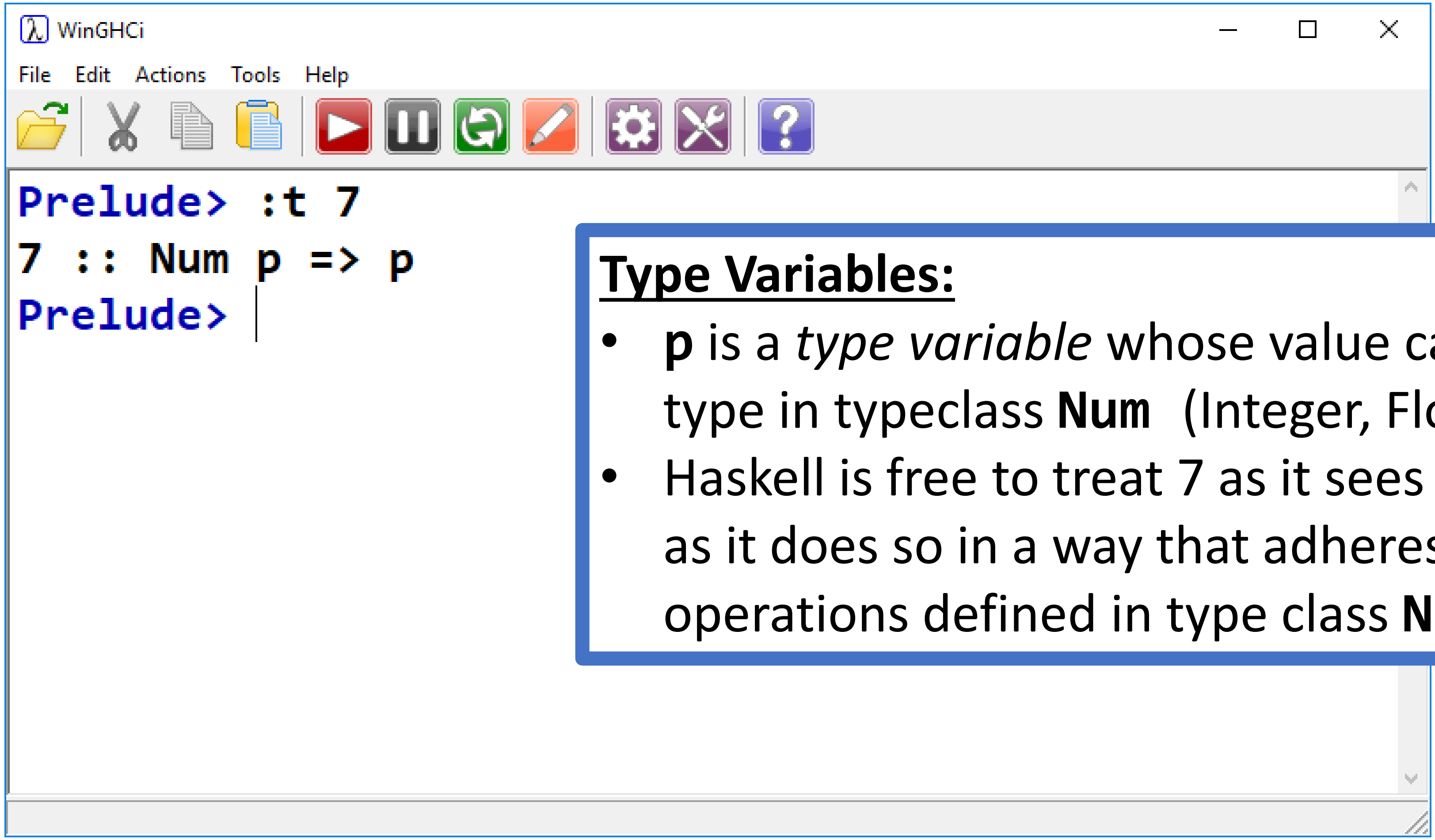


```
Prelude> :t 1
1 :: Num p => p
Prelude> :t 1.0
1.0 :: Fractional p => p
Prelude> :t 'a'
'a' :: Char
Prelude> :t "Hello"
"Hello" :: [Char]
Prelude> :t 1 > 2
1 > 2 :: Bool
Prelude>
```

- 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 ?

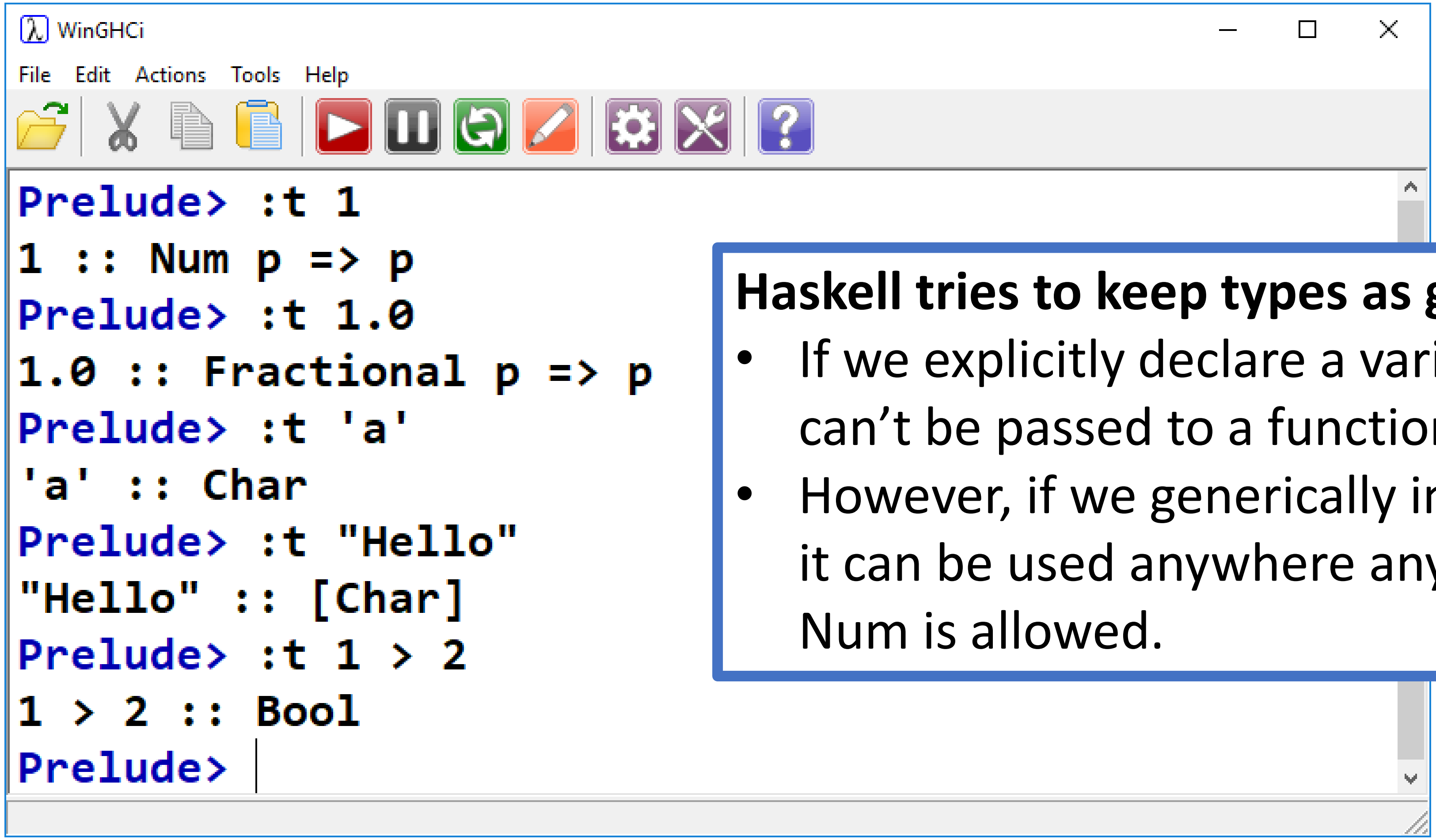


```
Prelude> :t 7
7 :: Num p => p
Prelude>
```

**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?



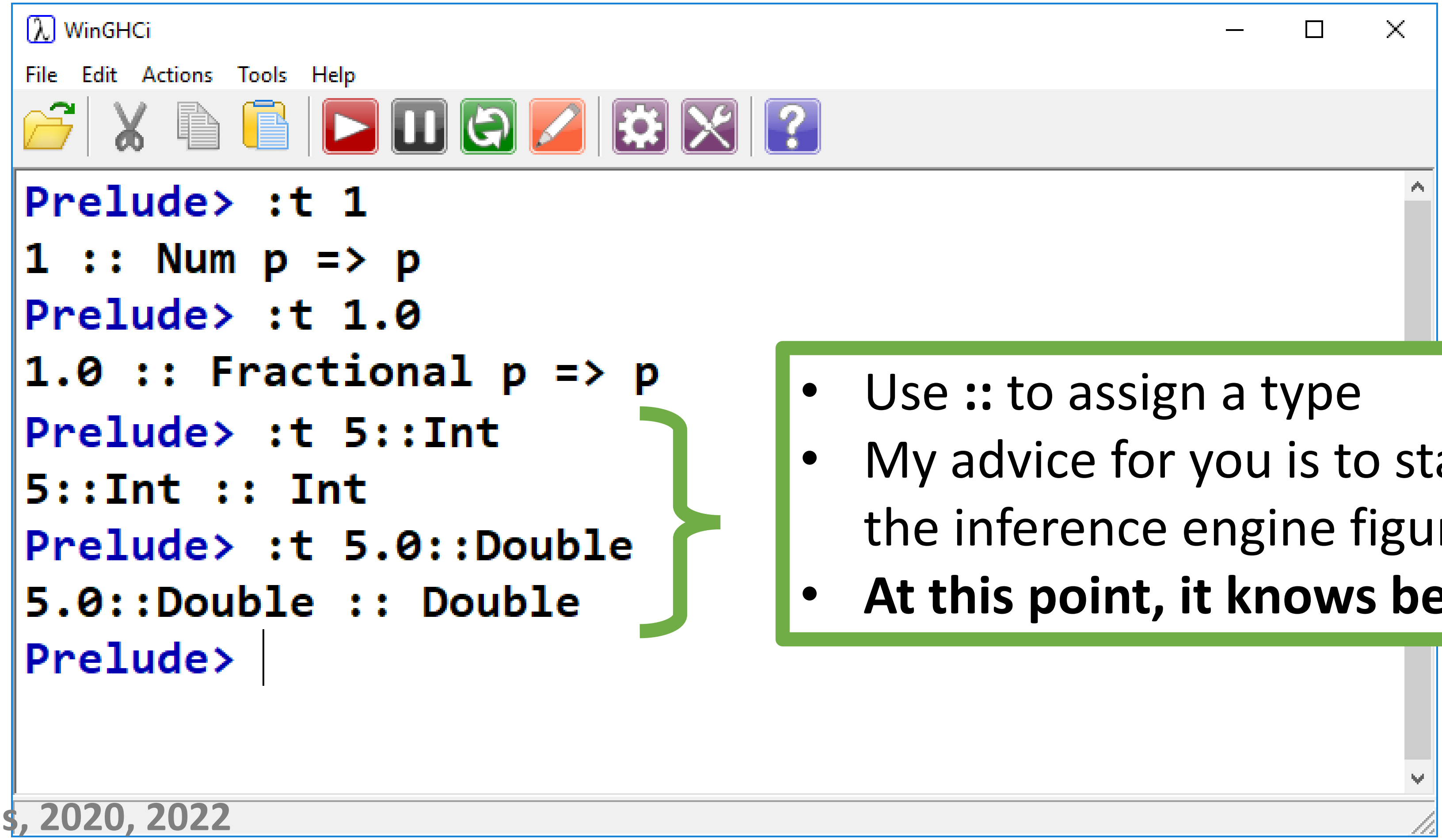
```
Prelude> :t 1
1 :: Num p => p
Prelude> :t 1.0
1.0 :: Fractional p => p
Prelude> :t 'a'
'a' :: Char
Prelude> :t "Hello"
"Hello" :: [Char]
Prelude> :t 1 > 2
1 > 2 :: Bool
Prelude>
```

**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:

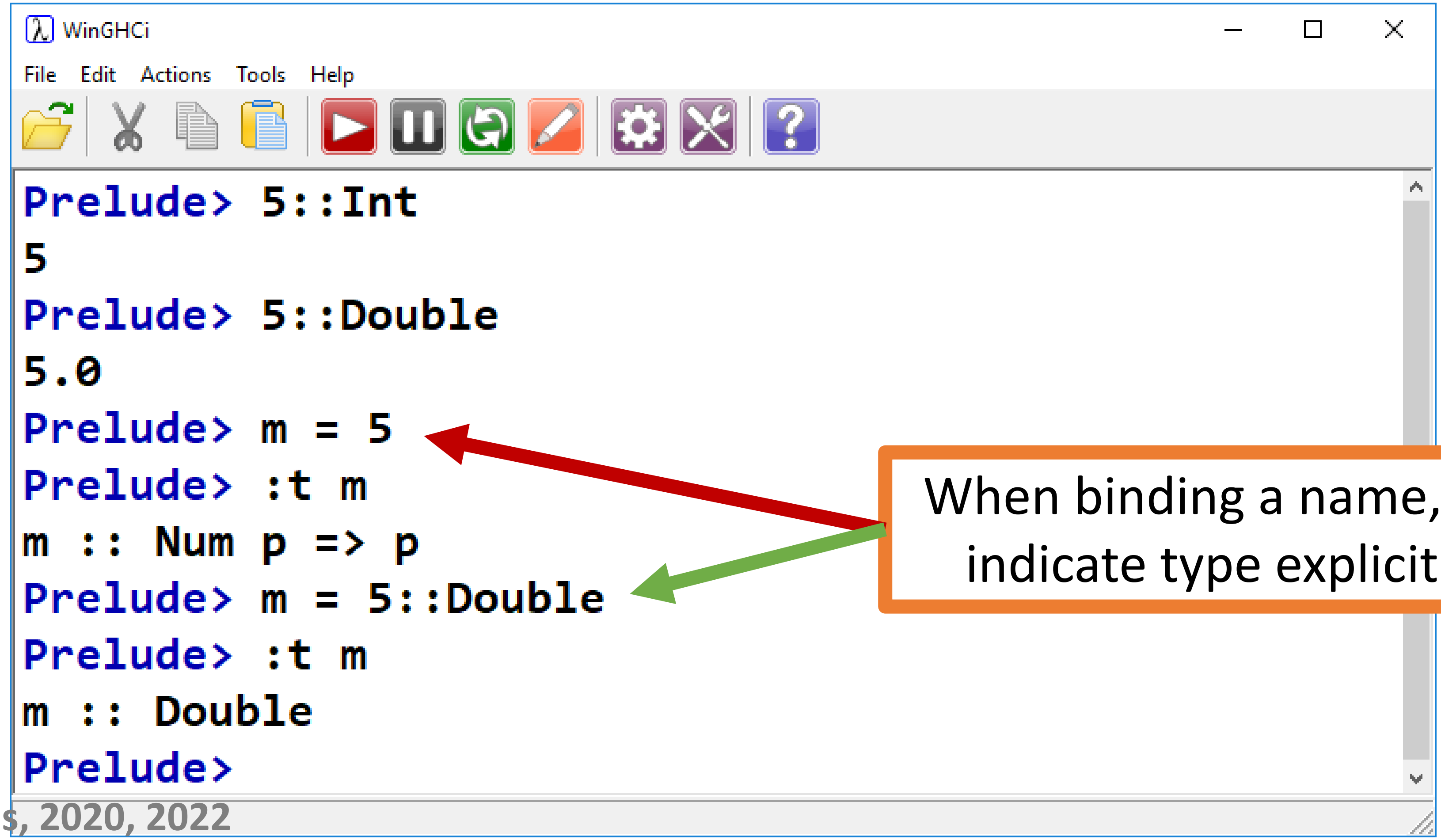


```
Prelude> :t 1
1 :: Num p => p
Prelude> :t 1.0
1.0 :: Fractional p => p
Prelude> :t 5::Int
5::Int :: Int
Prelude> :t 5.0::Double
5.0::Double :: Double
Prelude> 
```

- 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:



```
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 binding 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:

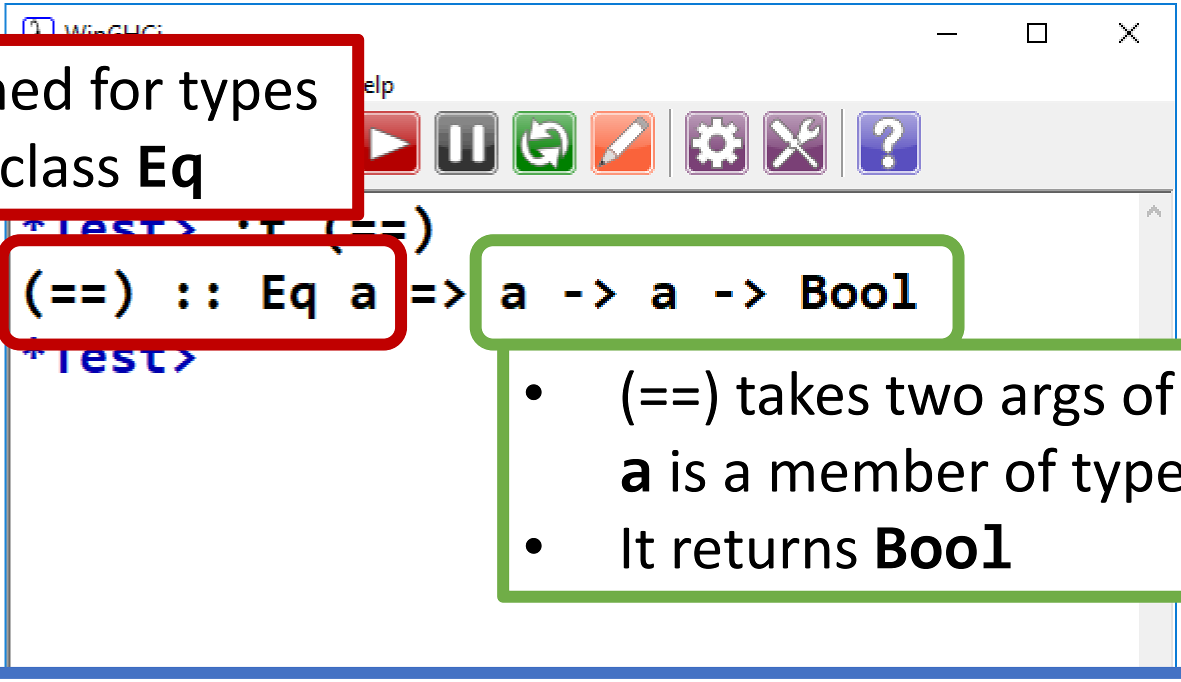
Takes two arguments, each of the same type (call it α), and returns a Boolean

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

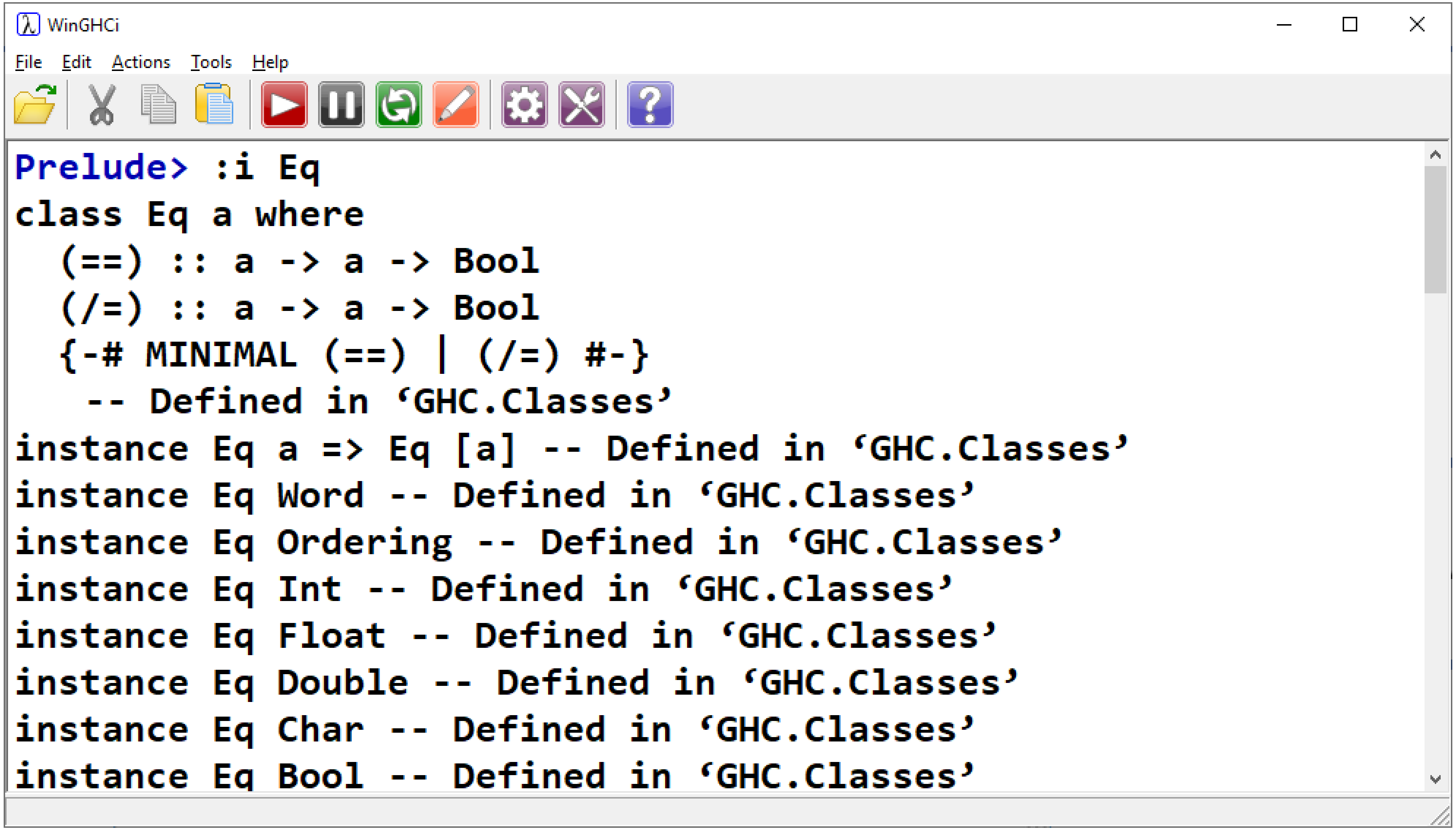


```
*Test> :t (==)
(==) :: Eq a => a -> a -> Bool
*Test>
```

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

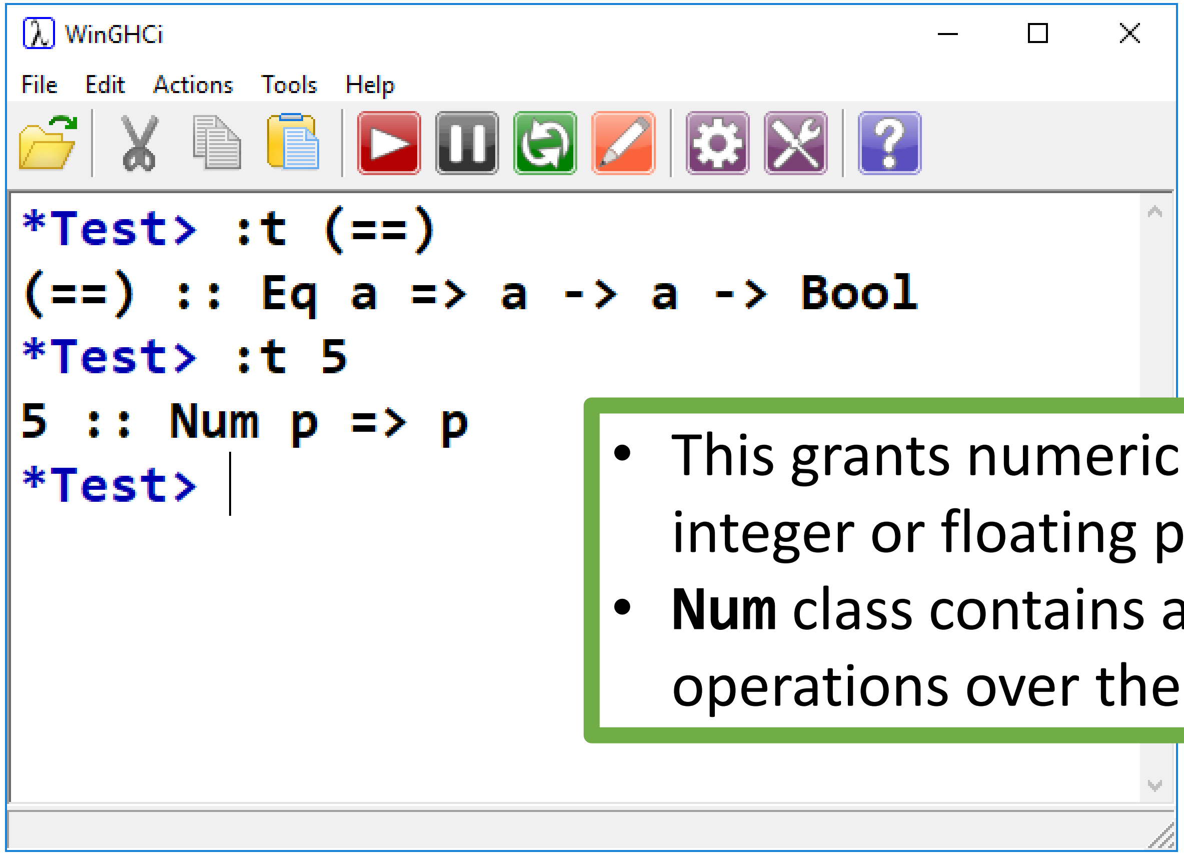
- (==) takes two args of type **a**, where **a** is a member of type class **Eq**
- It returns **Bool**

- 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.



```
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’
```

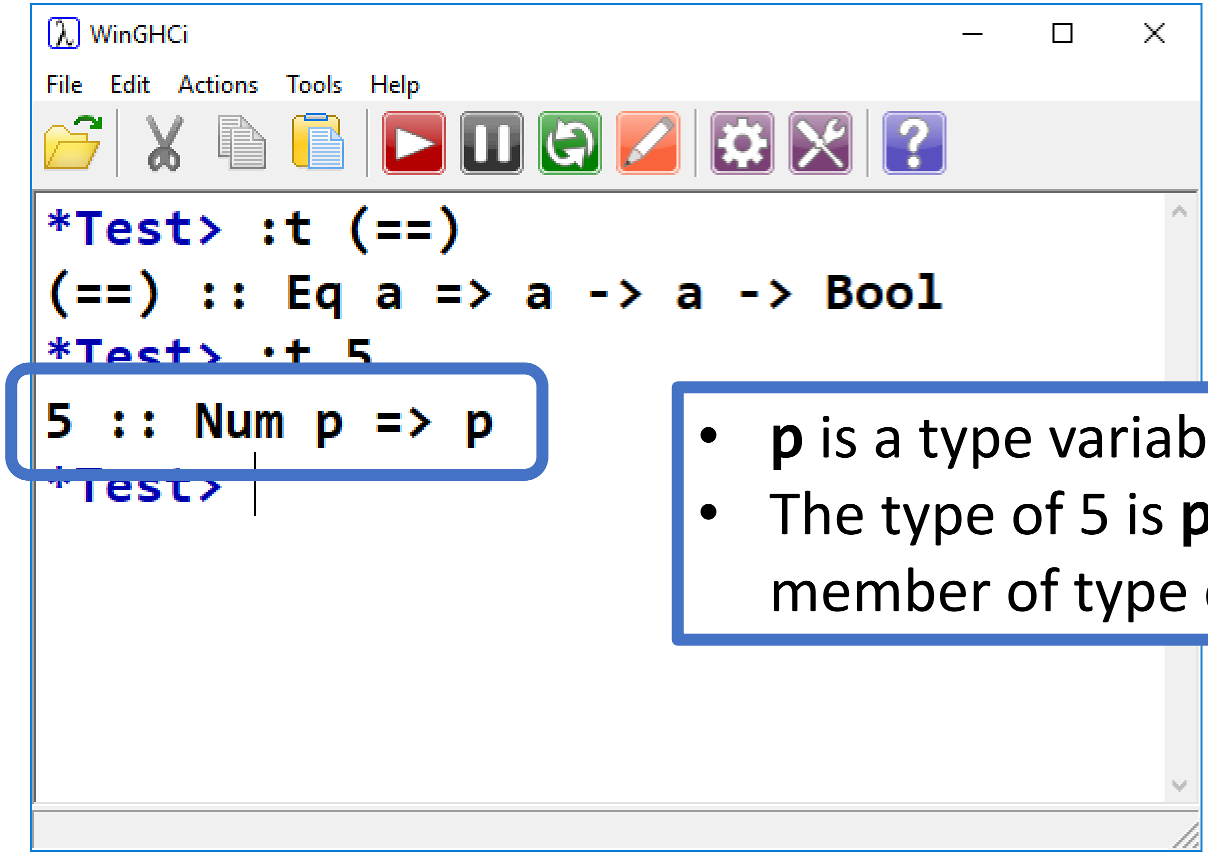
# Num Type Class



```
*Test> :t (==)
(==) :: Eq a => a -> a -> Bool
*Test> :t 5
5 :: Num p => p
*Test>
```

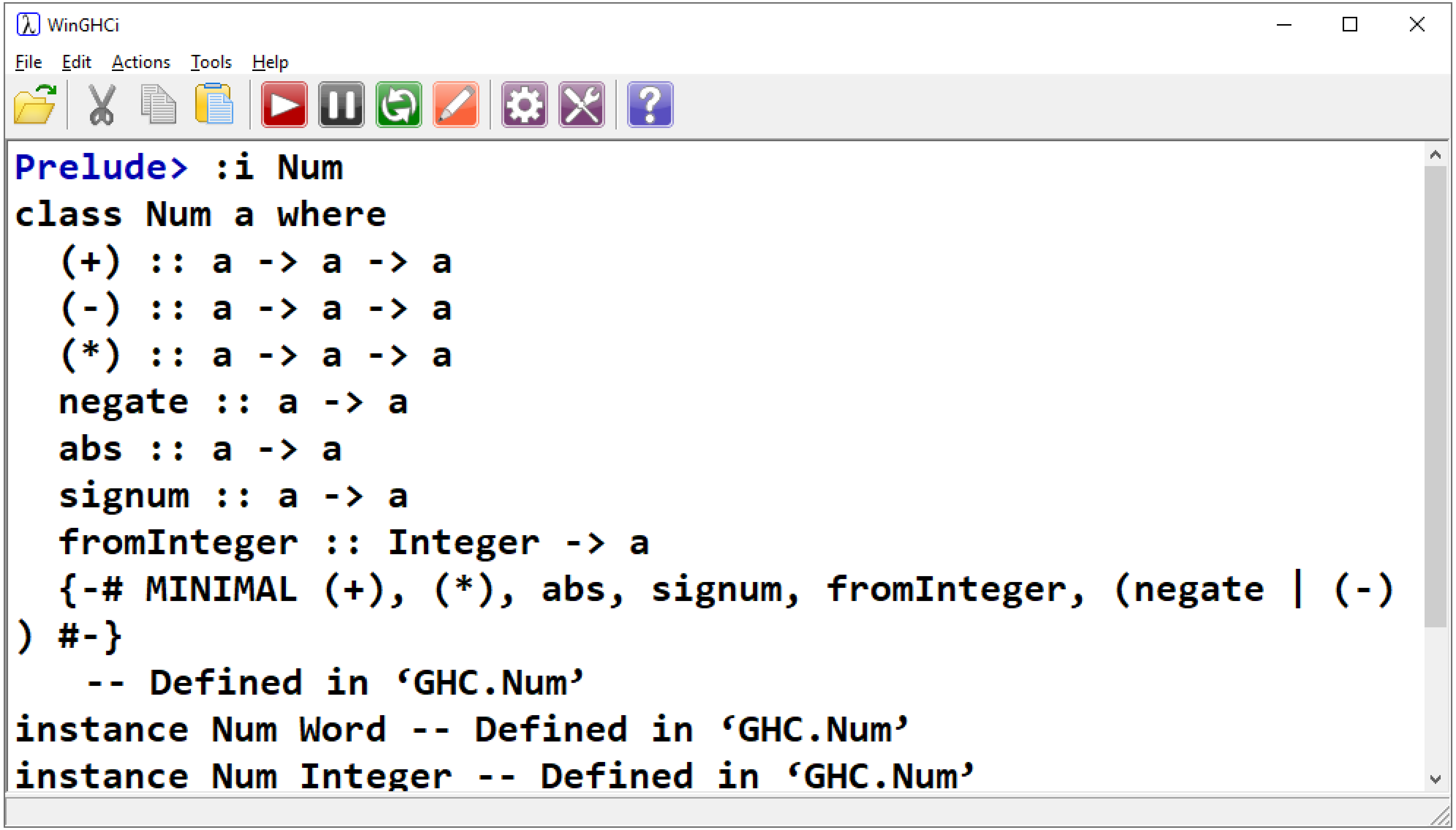
- This grants numeric values freedom to be an integer or floating point as the compiler sees fit.
- **Num** class contains all numbers, and certain operations over them such as addition.

# Num Type Class



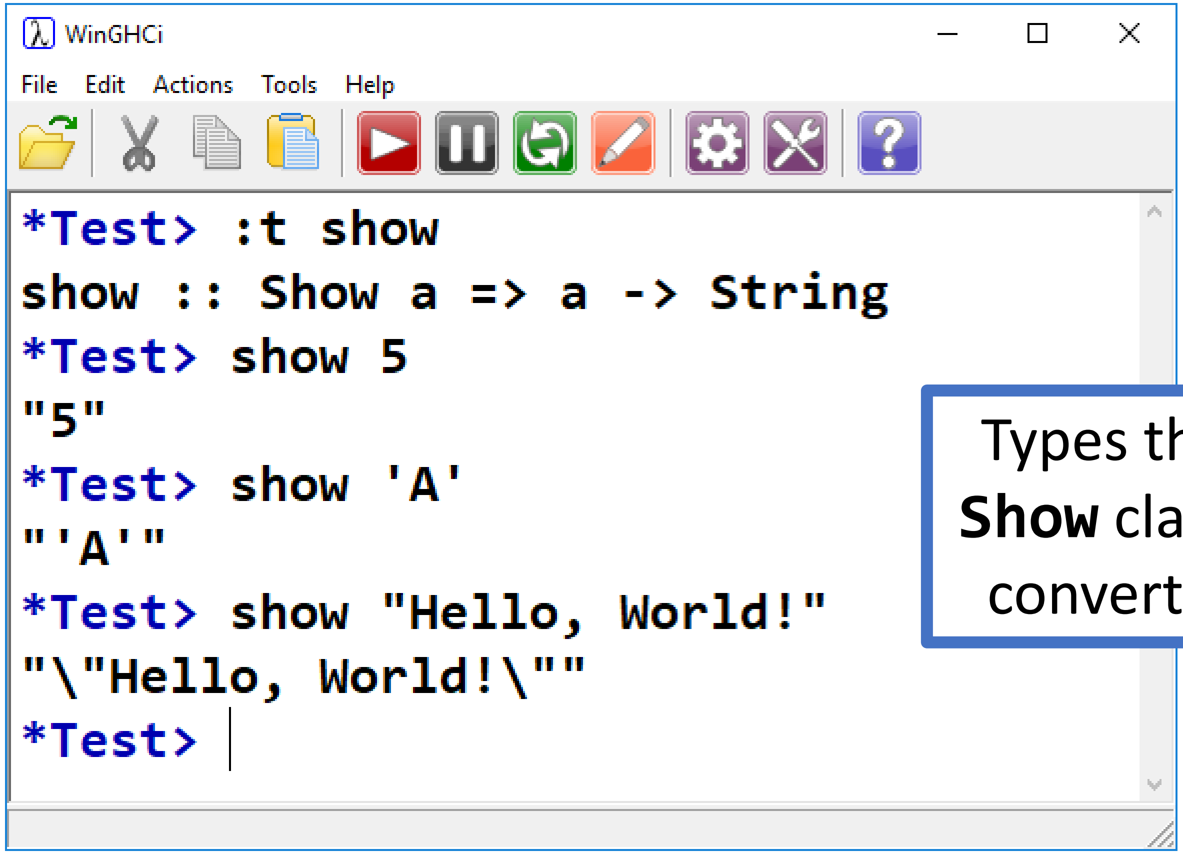
```
*Test> :t (==)
(==) :: Eq a => a -> a -> Bool
*Test> :t 5
5 :: Num p => p
*Test>
```

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



```
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

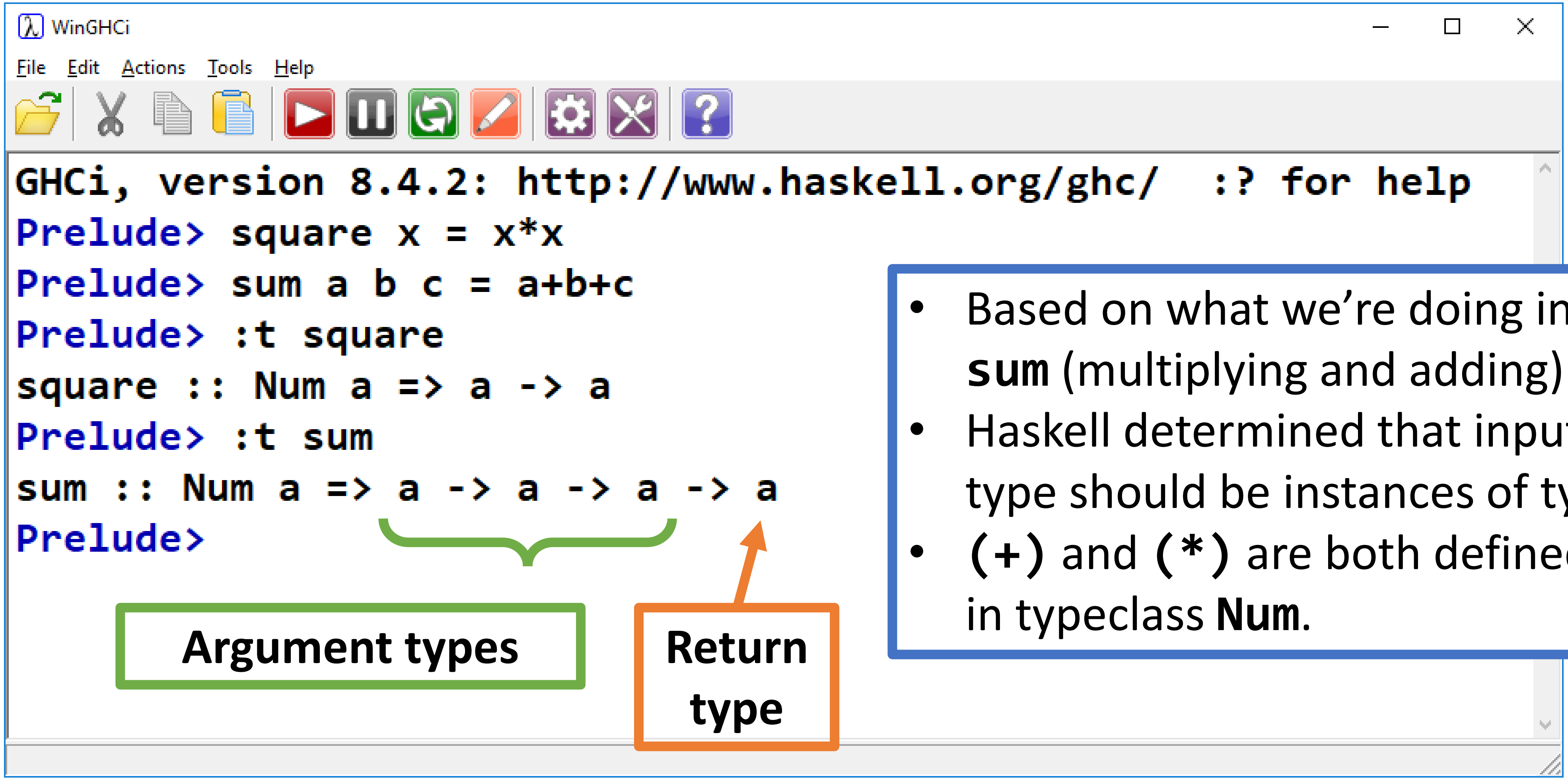


```
*Test> :t show
show :: Show a => a -> String
*Test> show 5
"5"
*Test> show 'A'
"'A'"
*Test> show "Hello, World!"
"\"Hello, World!\""
*Test>
```

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

© Alex Ufkes, 2020, 2022

# Functions & Typeclasses



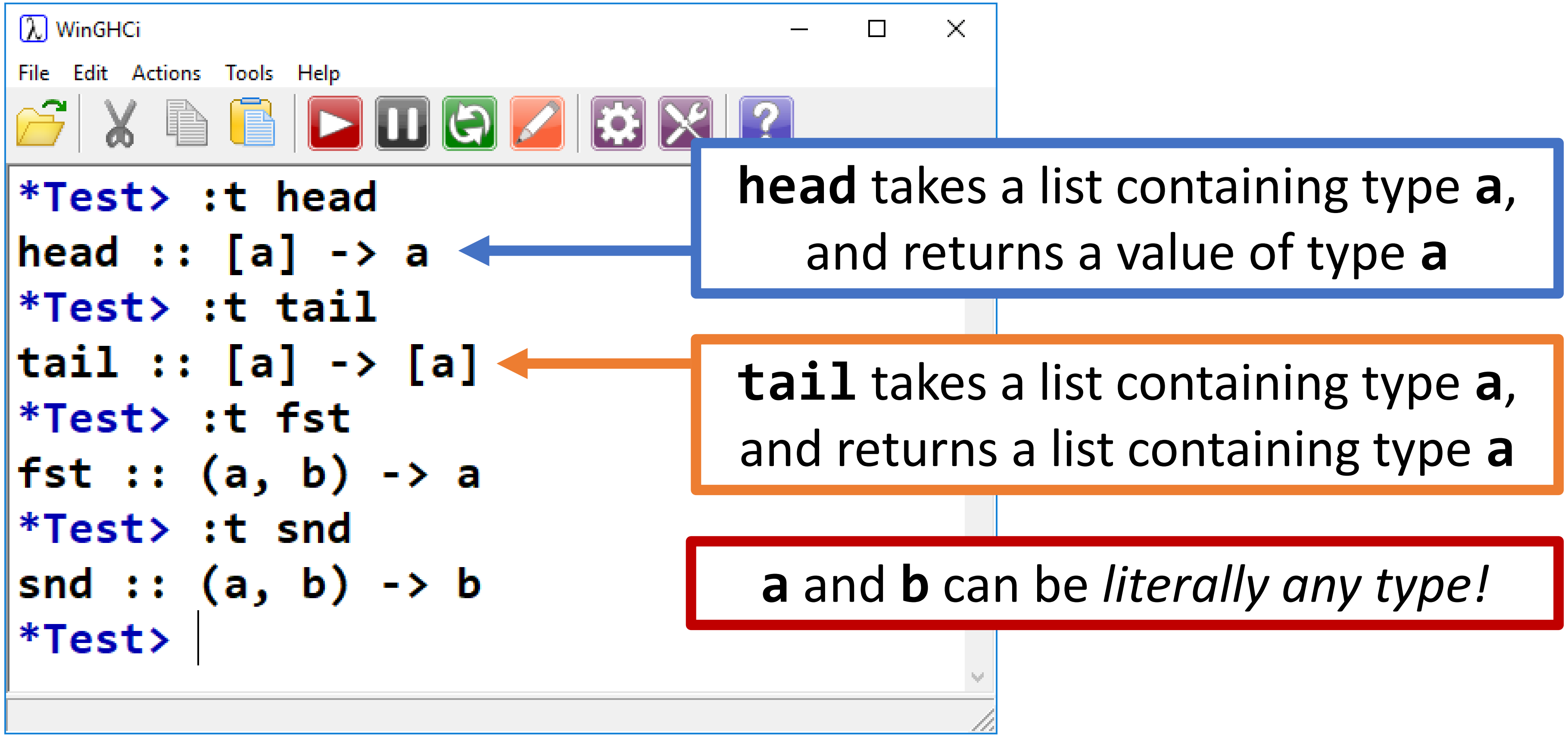
```
GHCi, version 8.4.2: http://www.haskell.org/ghc/  :? for help
Prelude> square x = x*x
Prelude> sum a b c = a+b+c
Prelude> :t square
square :: Num a => a -> a
Prelude> :t sum
sum :: Num a => a -> a -> a -> a
Prelude>
```

Argument types

Return type

- 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**.

# Function Type Signatures



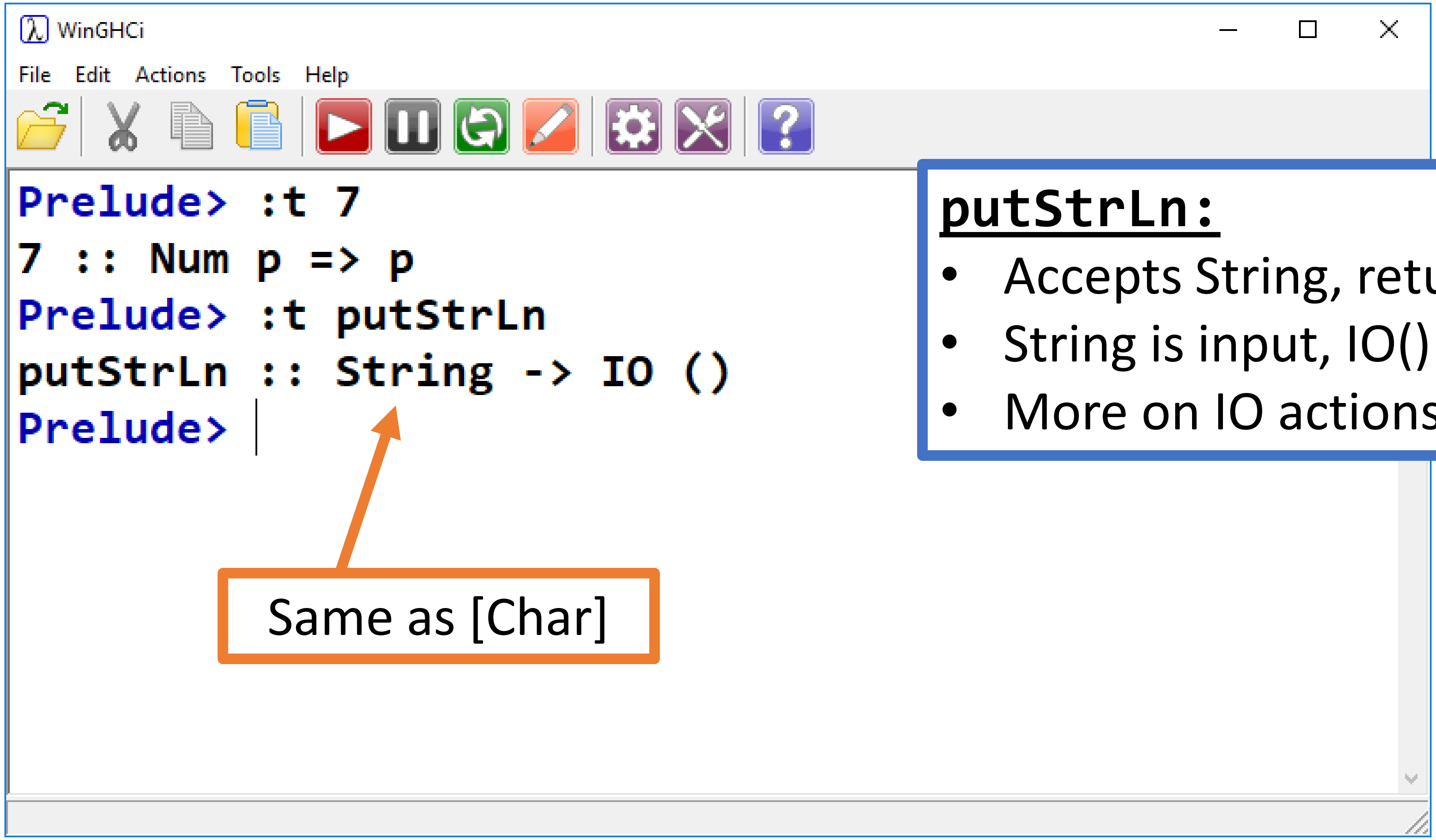
```
*Test> :t head
head :: [a] -> a
*Test> :t tail
tail :: [a] -> [a]
*Test> :t fst
fst :: (a, b) -> a
*Test> :t snd
snd :: (a, b) -> b
*Test>
```

**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 Type Signatures



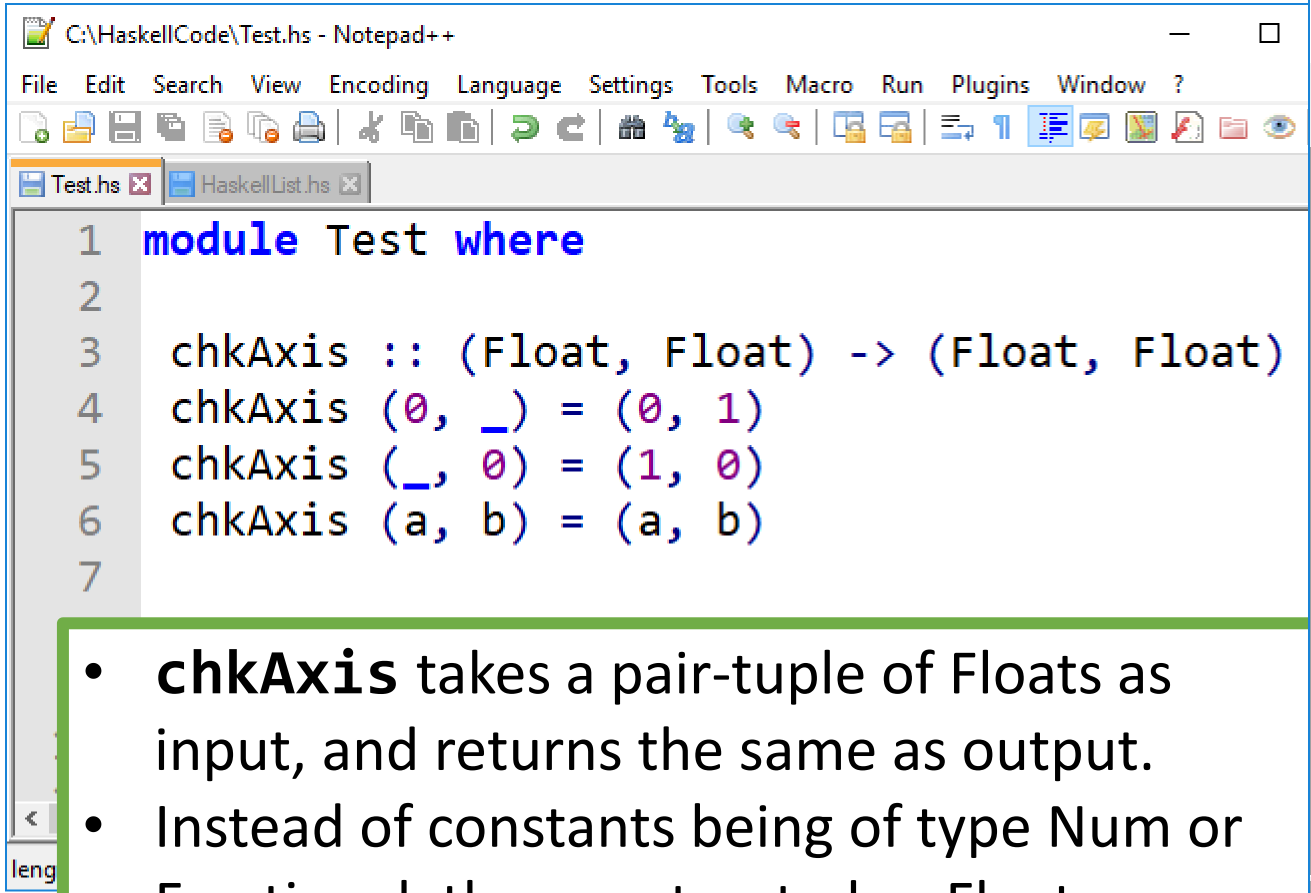
```
Prelude> :t 7
7 :: Num p => p
Prelude> :t putStrLn
putStrLn :: String -> IO ()
Prelude>
```

Same as [Char]

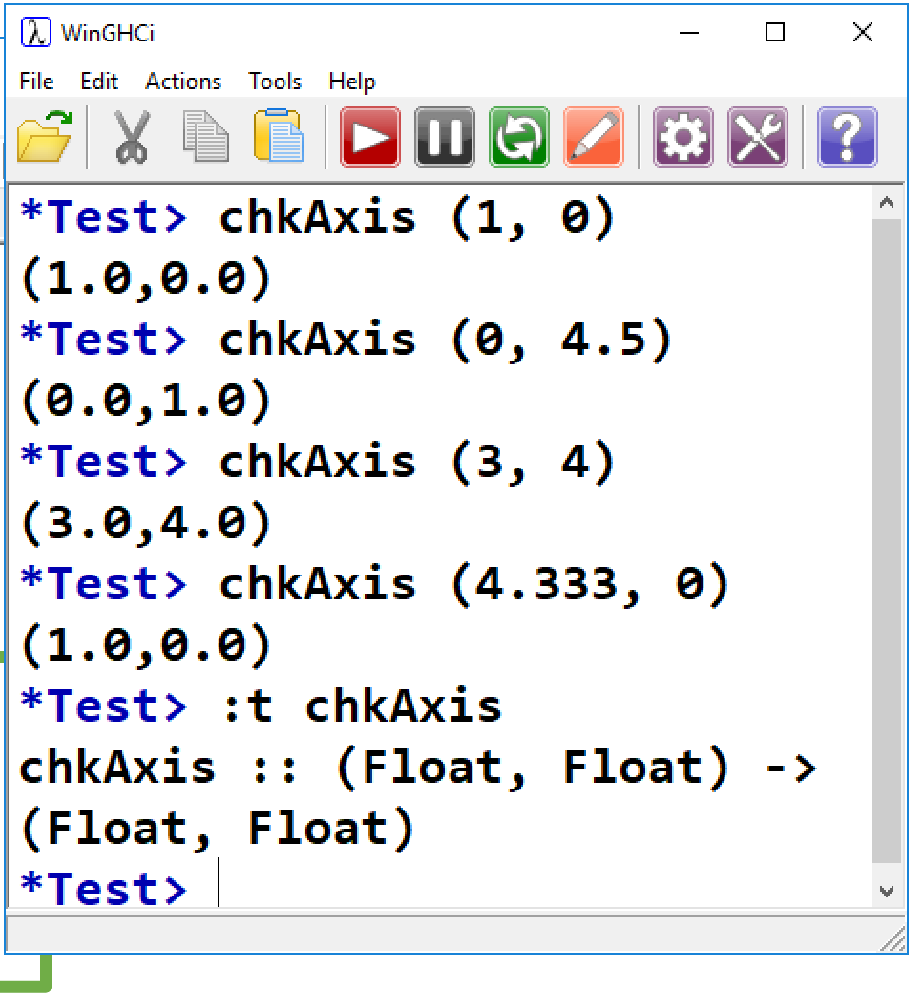
**putStrLn:**

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

# Specify Function Type


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
leng

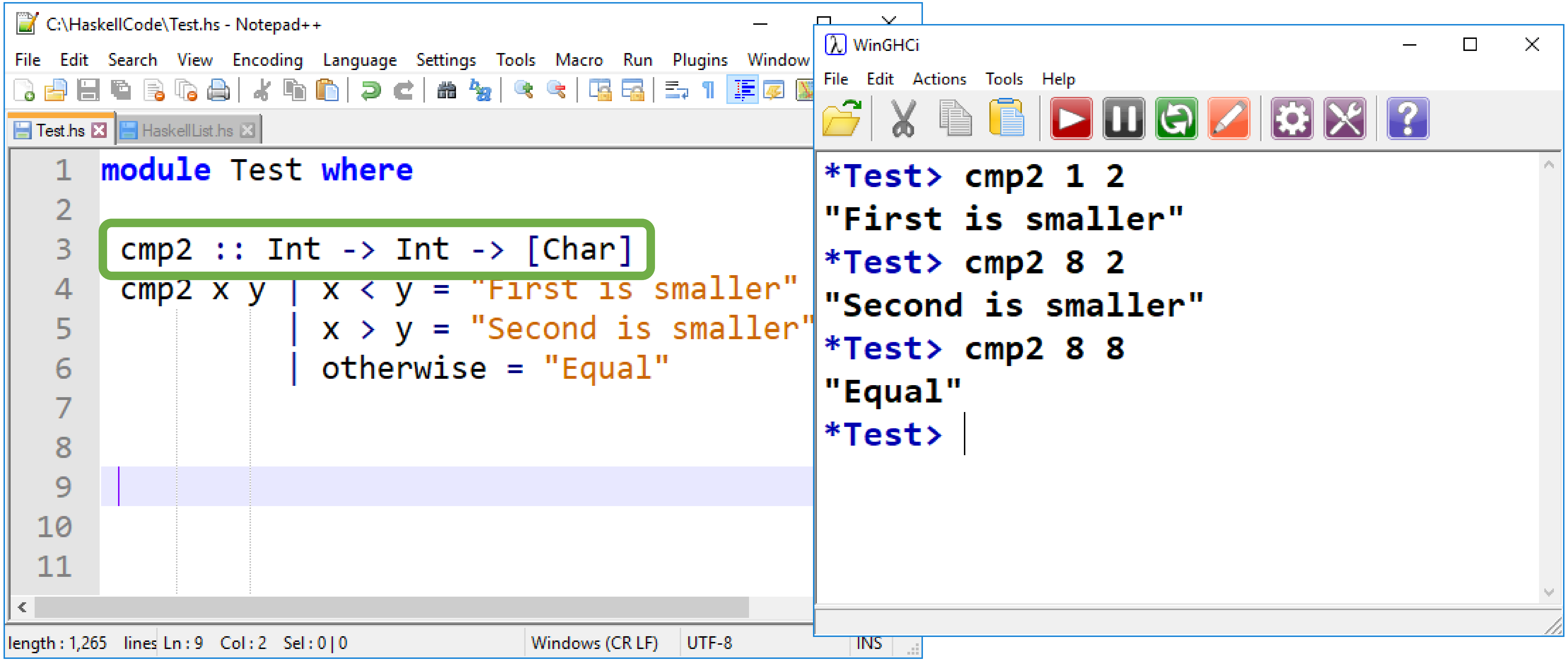
```haskell
module Test where

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

```



```
*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

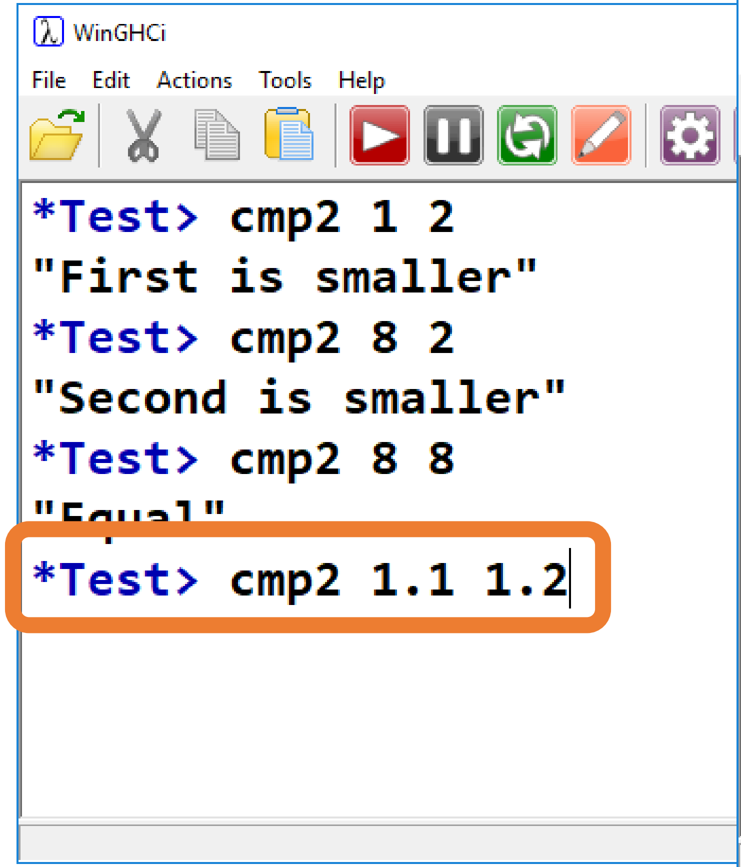


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window
Test.hs
HaskellList.hs
length : 1,265 lines Ln : 9 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS
WinGHCi
File Edit Actions Tools Help

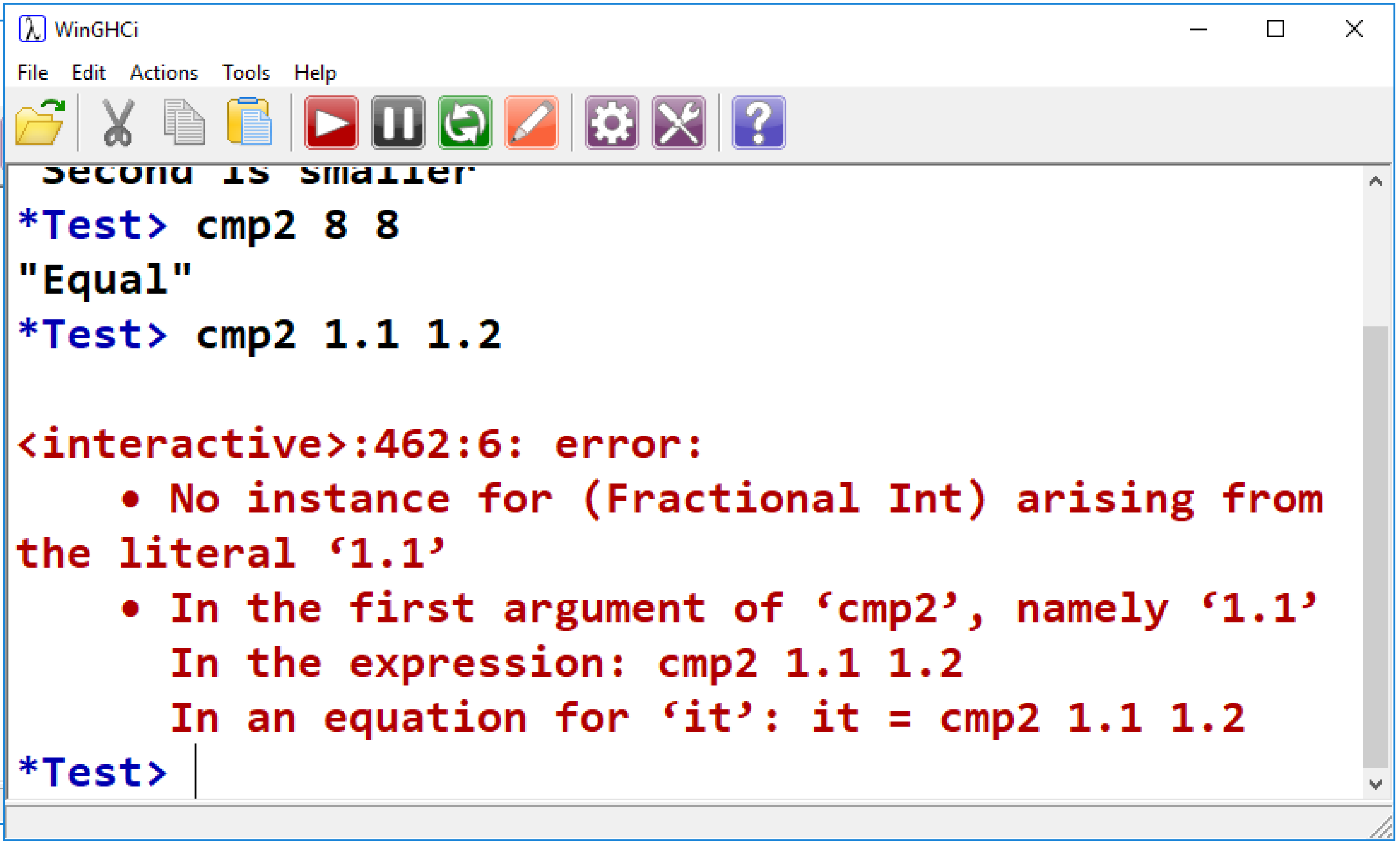
```haskell
module Test where

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

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

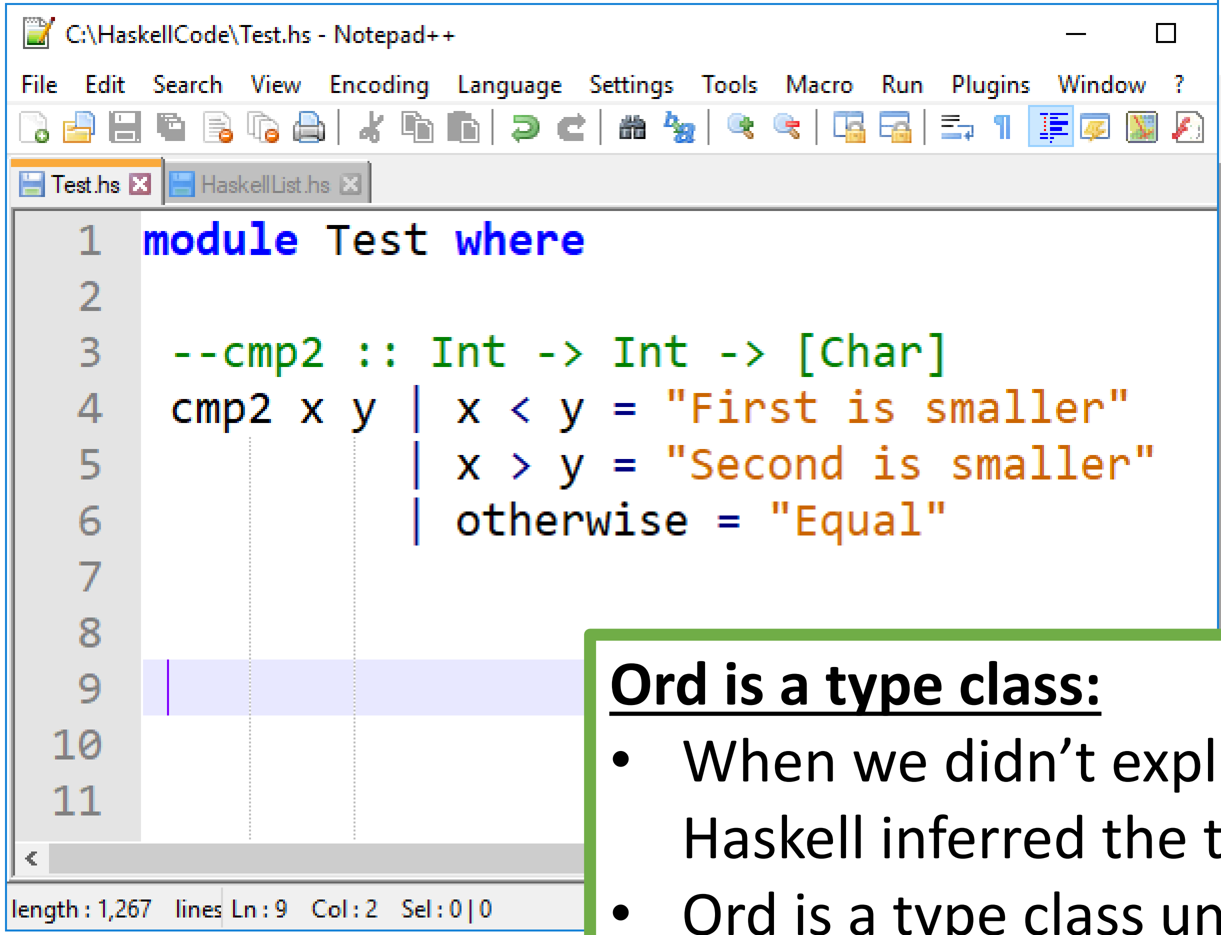
# Thoughts?



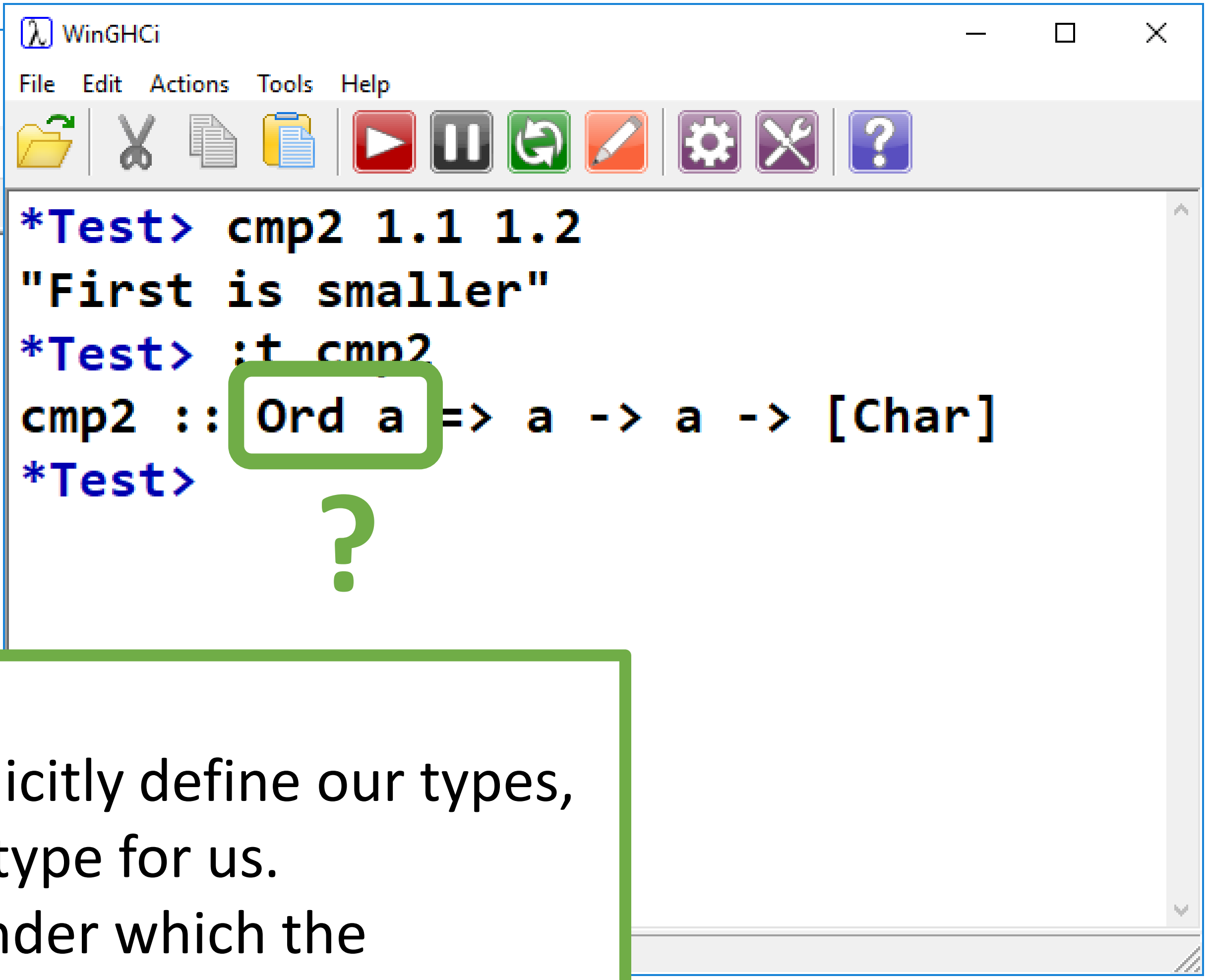
```
*Test> cmp2 1 2
"First is smaller"
*Test> cmp2 8 2
"Second is smaller"
*Test> cmp2 8 8
"Equal"
*Test> cmp2 1.1 1.2
```



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

<interactive>:462:6: 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
length : 1,267 lines Ln : 9 Col : 2 Sel : 0 | 0

```haskell
module Test where

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

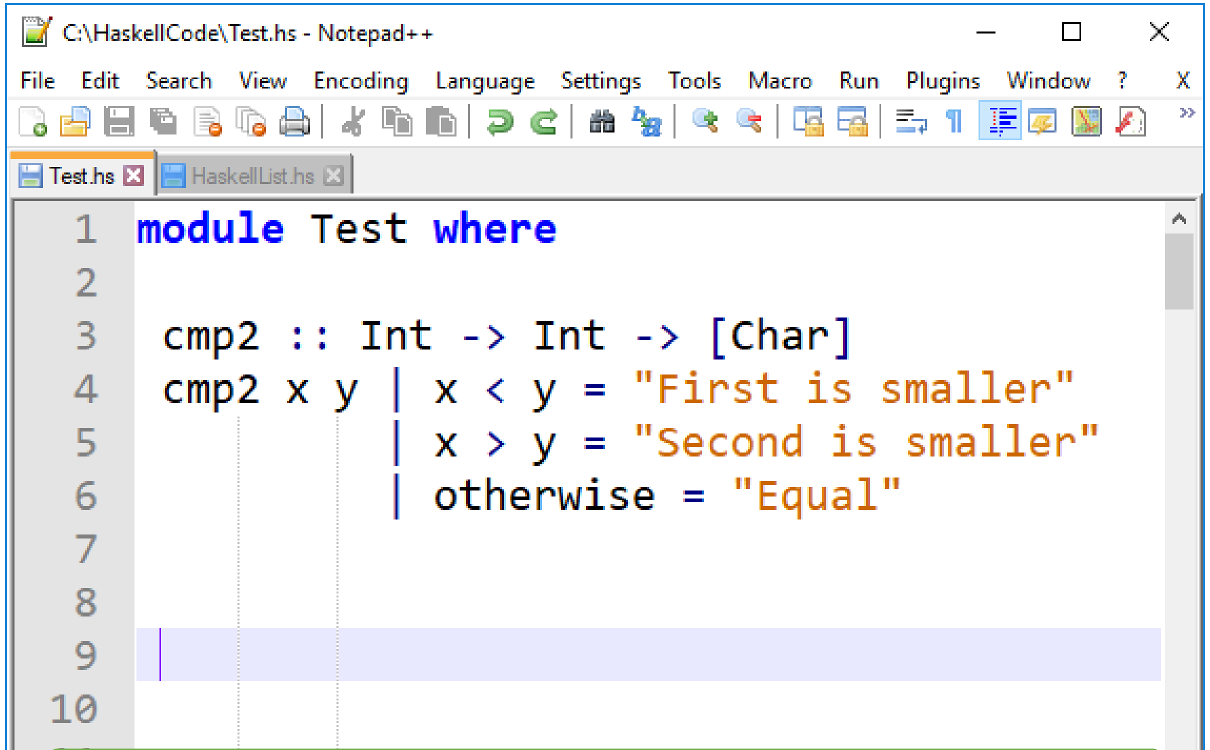


```
*Test> cmp2 1.1 1.2
"First is smaller"
*Test> :t cmp2
cmp2 :: Ord a => a -> a -> [Char]
*Test>
```

**Ord is a type class:**

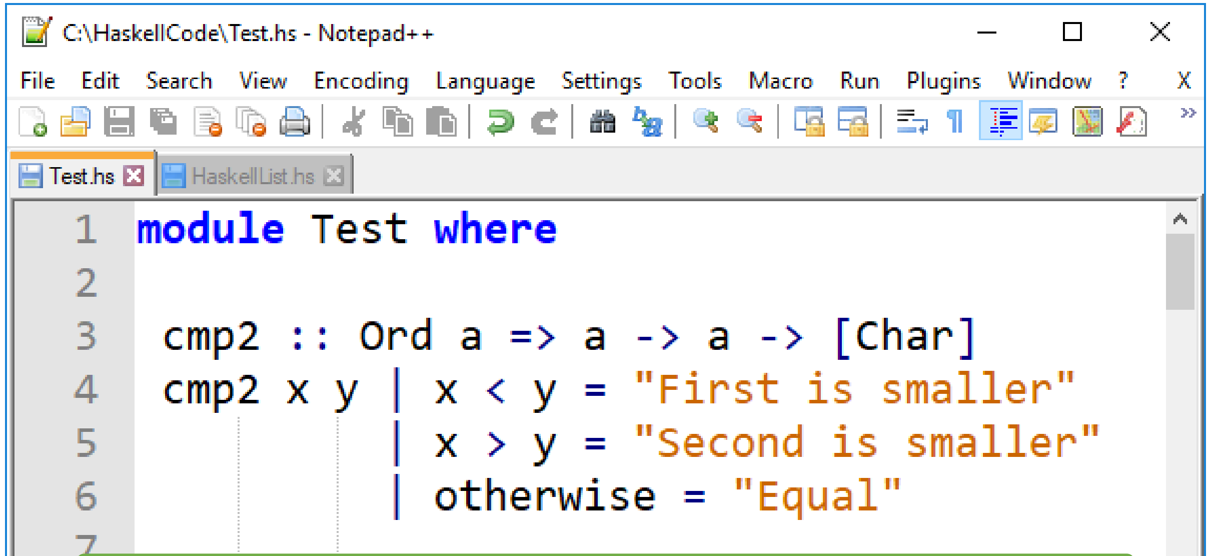
- When we didn’t explicitly define our types, Haskell inferred the type for us.
- Ord is a type class under which the operations used on our inputs are defined.
- I.e., comparison operators.

# Type VS Type Class


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

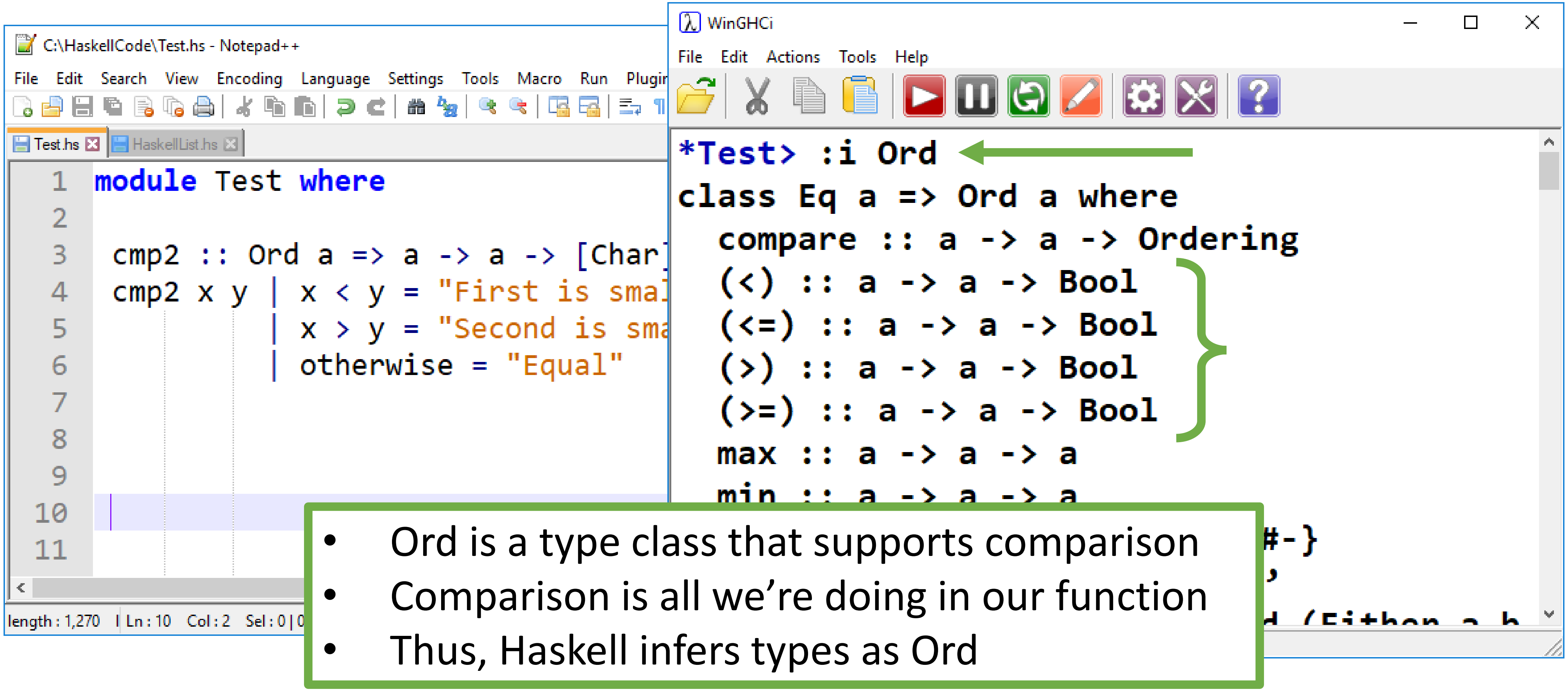
- Int & Char are <u>types</u>, ***not*** type classes
- We can use the above notation


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 :: 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++
File Edit Search View Encoding Language Settings Tools Macro Run Plugir
Test.hs
HaskellList.hs

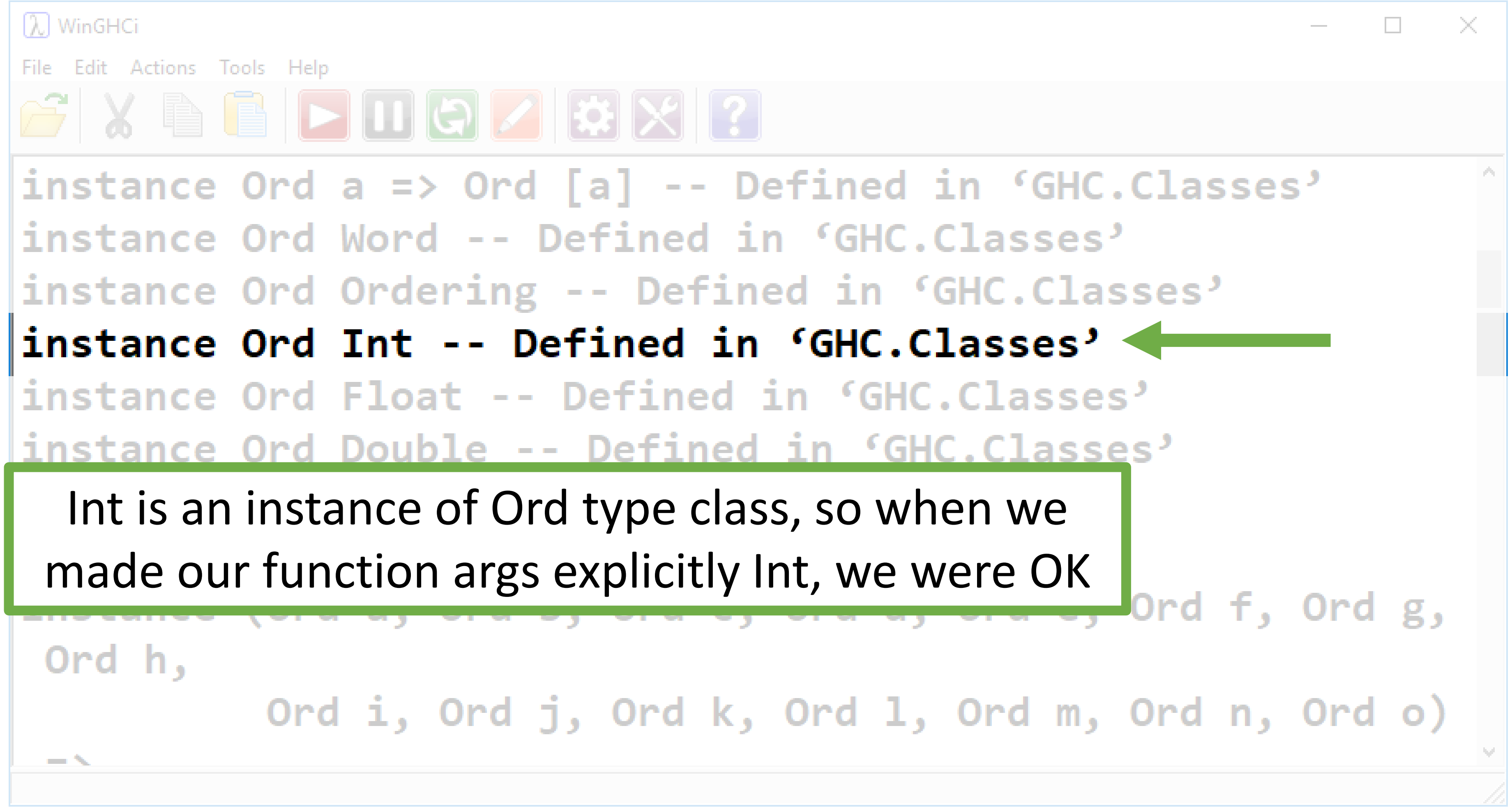
```haskell
module Test where

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

length : 1,270 Ln : 10 Col : 2 Sel : 0 | 0
WinGHCi
File Edit Actions Tools Help

```
*Test> :i Ord
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
#-}
,
d (Either a b
```

- 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

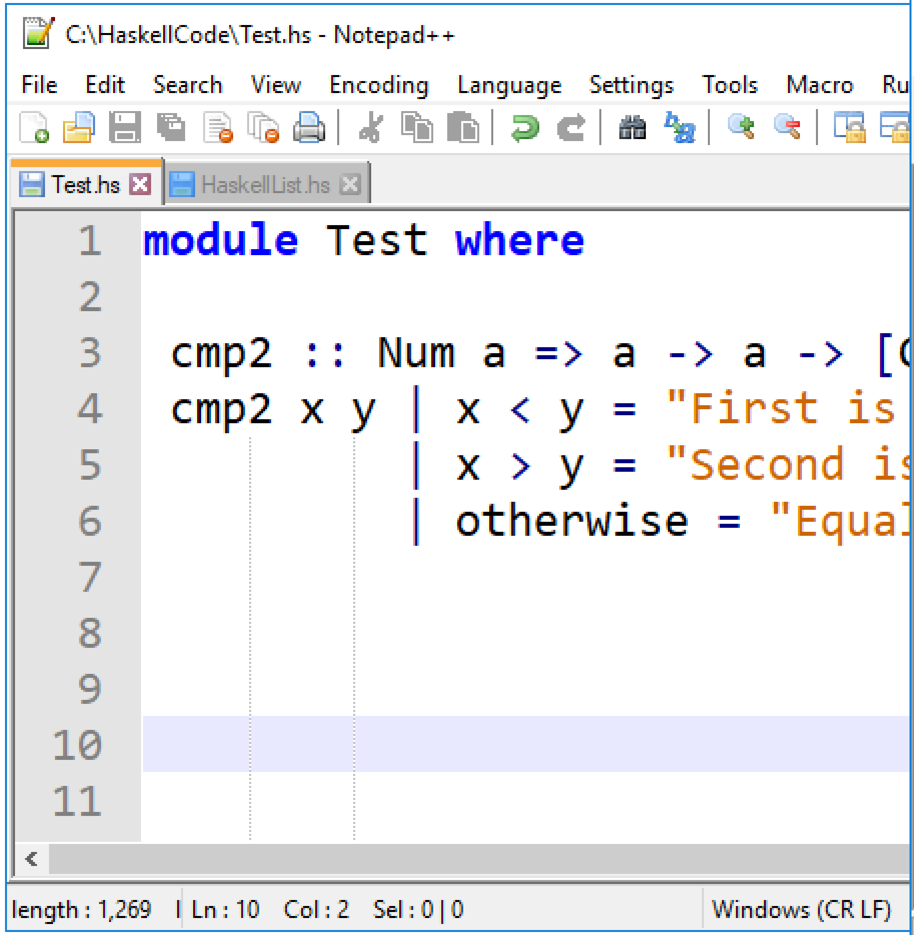


```
instance Ord a => Ord [a] -- Defined in ‘GHC.Classes’
instance Ord Word -- Defined in ‘GHC.Classes’
instance Ord Ordering -- Defined in ‘GHC.Classes’
instance Ord Int -- Defined in ‘GHC.Classes’
instance Ord Float -- Defined in ‘GHC.Classes’
instance Ord Double -- Defined in ‘GHC.Classes’
```

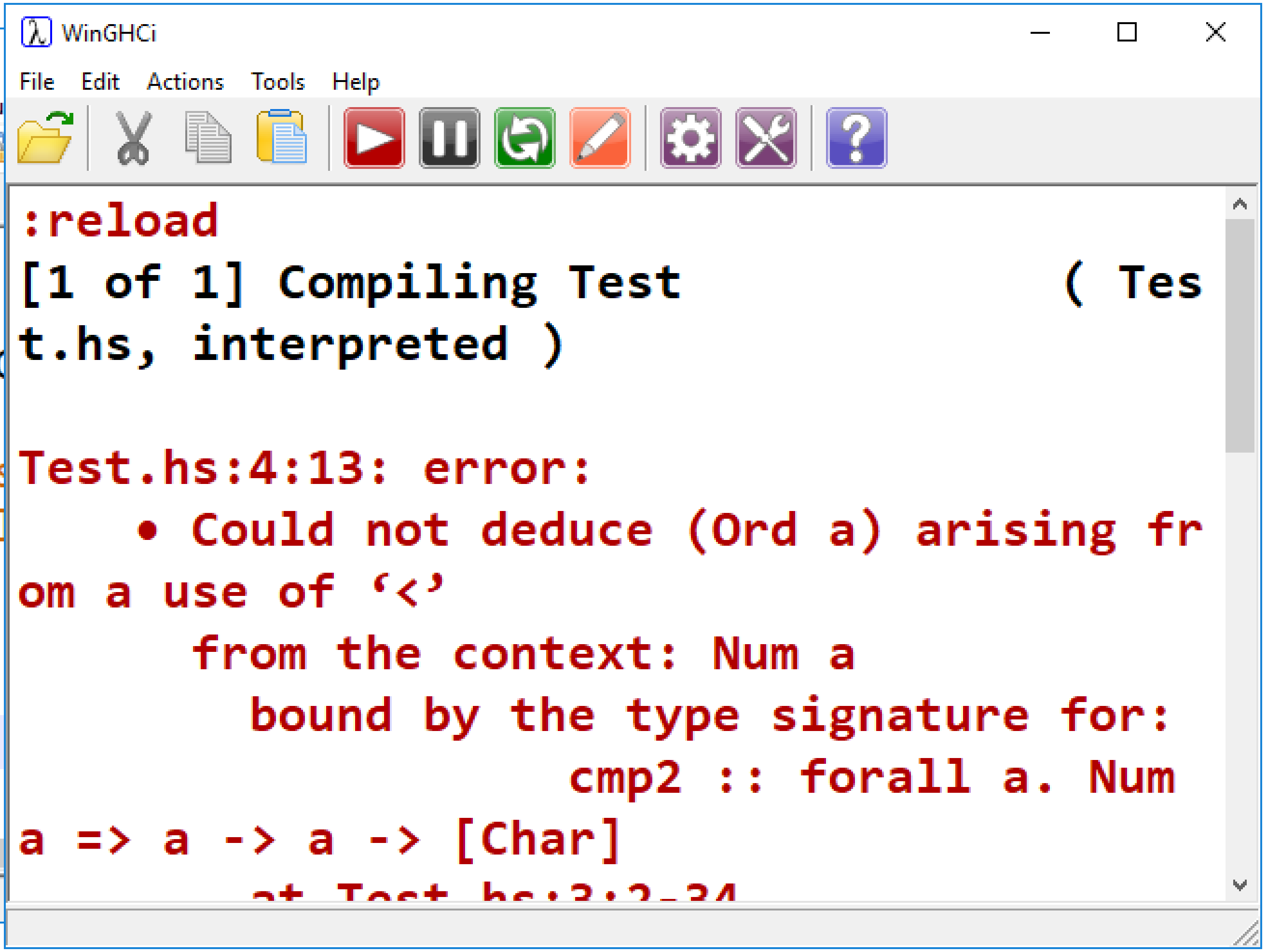
Int is an instance of Ord type class, so when we made our function args explicitly Int, we were OK

```
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, Ord g,
 Ord h,
         Ord i, Ord j, Ord k, Ord l, Ord m, Ord n, Ord o)
 =>
```

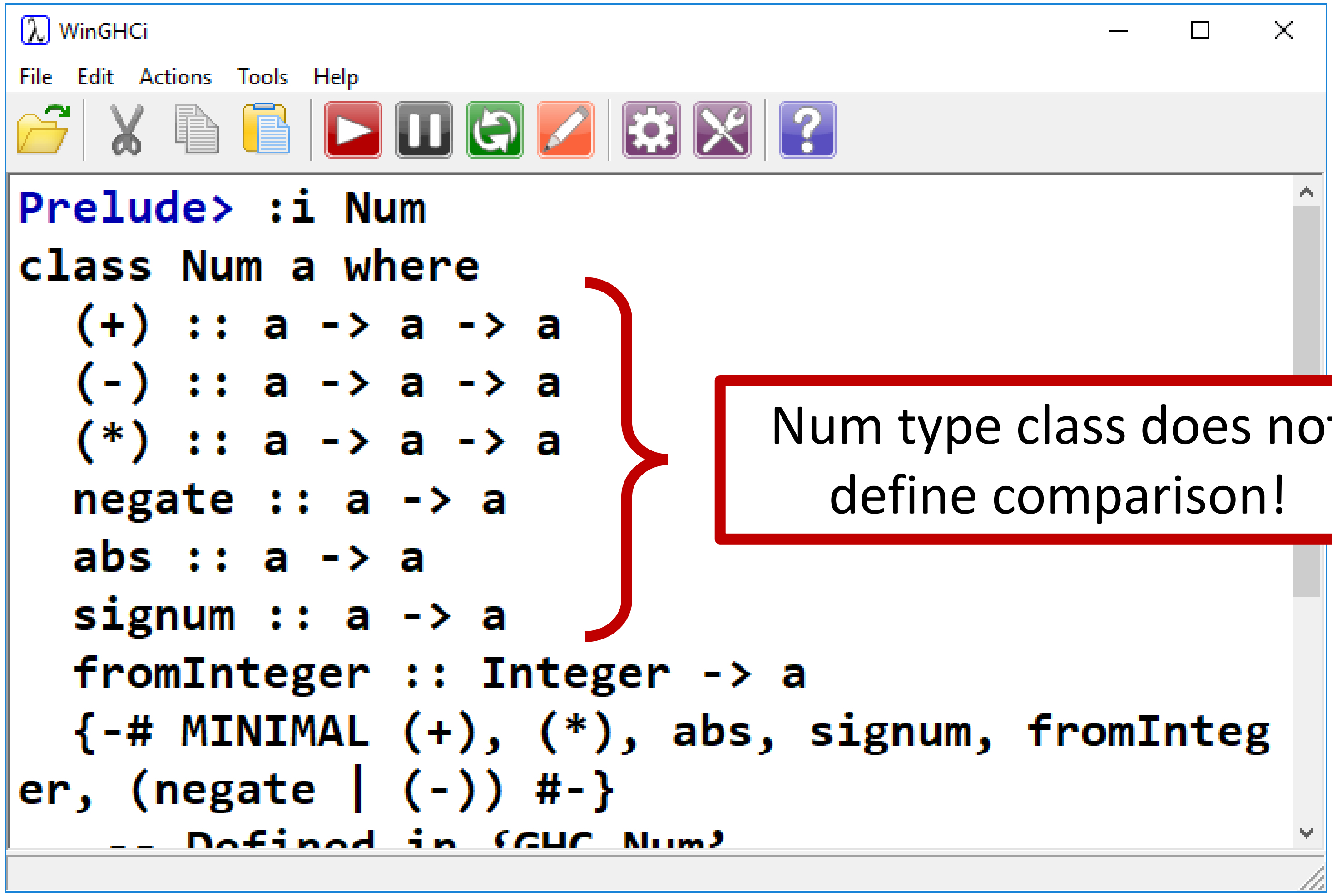
# How About This?


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Ru
Test.hs
HaskellList.hs
length : 1,269 Ln : 10 Col : 2 Sel : 0 | 0
Windows (CR LF)

```haskell
module Test where

 cmp2 :: Num a => a -> a -> [C
 cmp2 x y | x < y = "First is
          | x > y = "Second is
          | otherwise = "Equal
```



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

Test.hs:4:13: 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
```

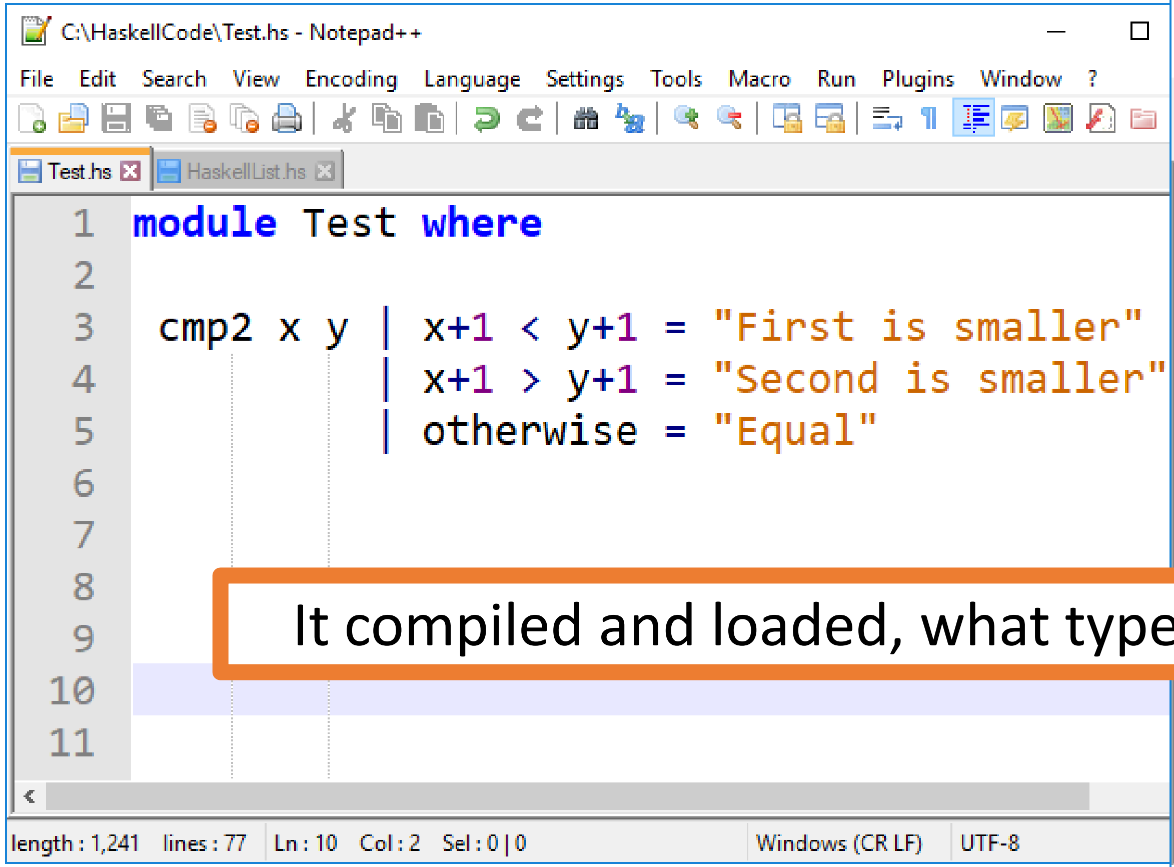


```
Prelude> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteg
er, (negate | (-)) #-}
      -- Defined in ‘GHC.Num’
```

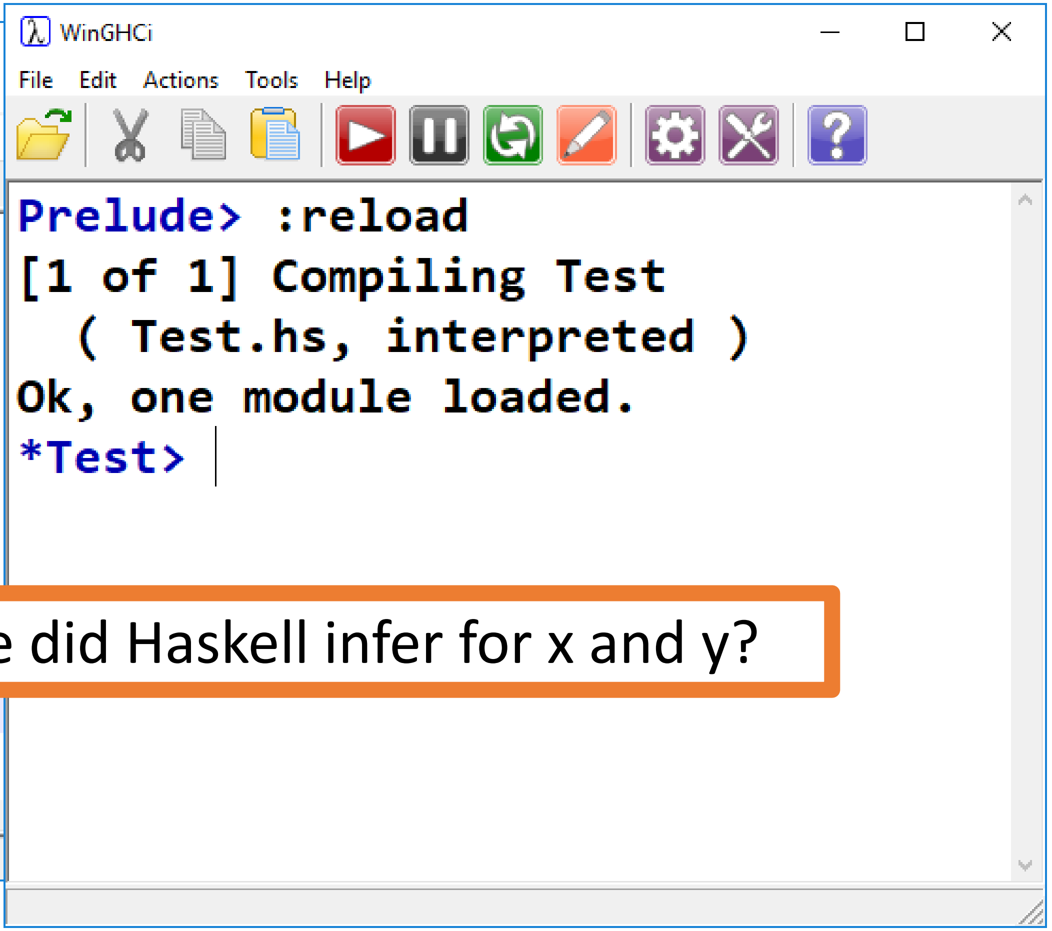
Num type class does not define comparison!

# Hmmmm…

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

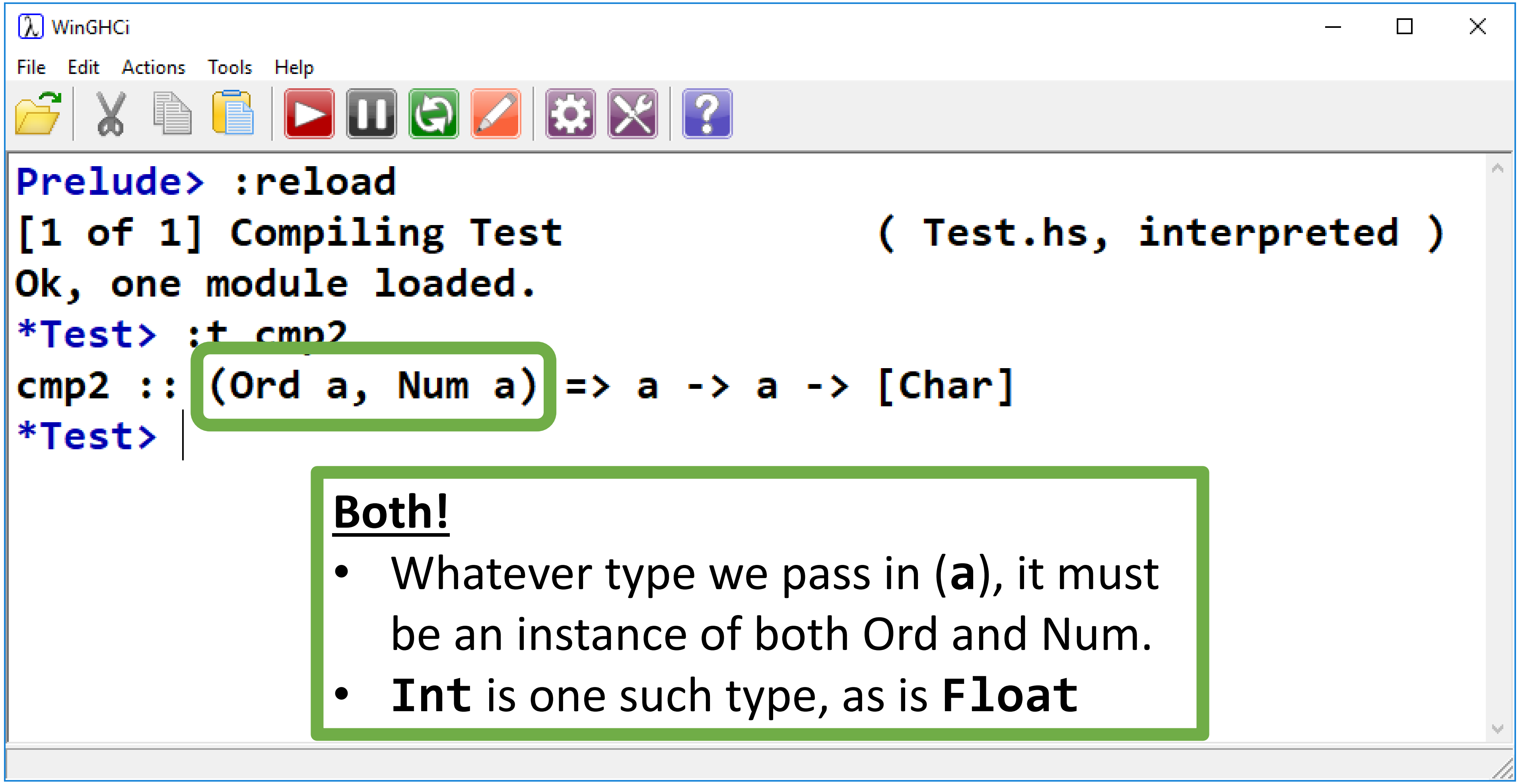

C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
length : 1,241 lines : 77 Ln : 10 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8

```haskell
module Test where

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



```
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?

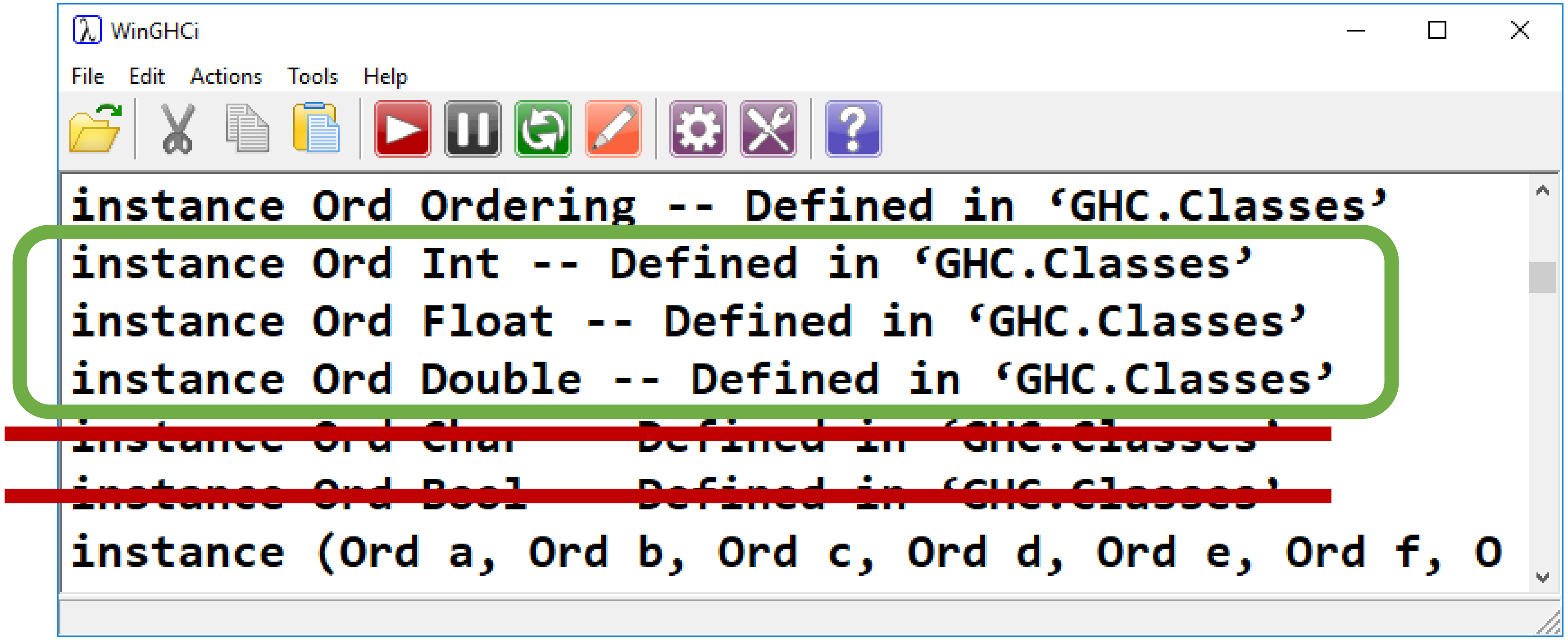


```
Prelude> :reload
[1 of 1] Compiling Test             ( Test.hs, interpreted )
Ok, one module loaded.
*Test> :t cmp2
cmp2 :: (Ord a, Num a) => a -> a -> [Char]
*Test>
```

**Both!**

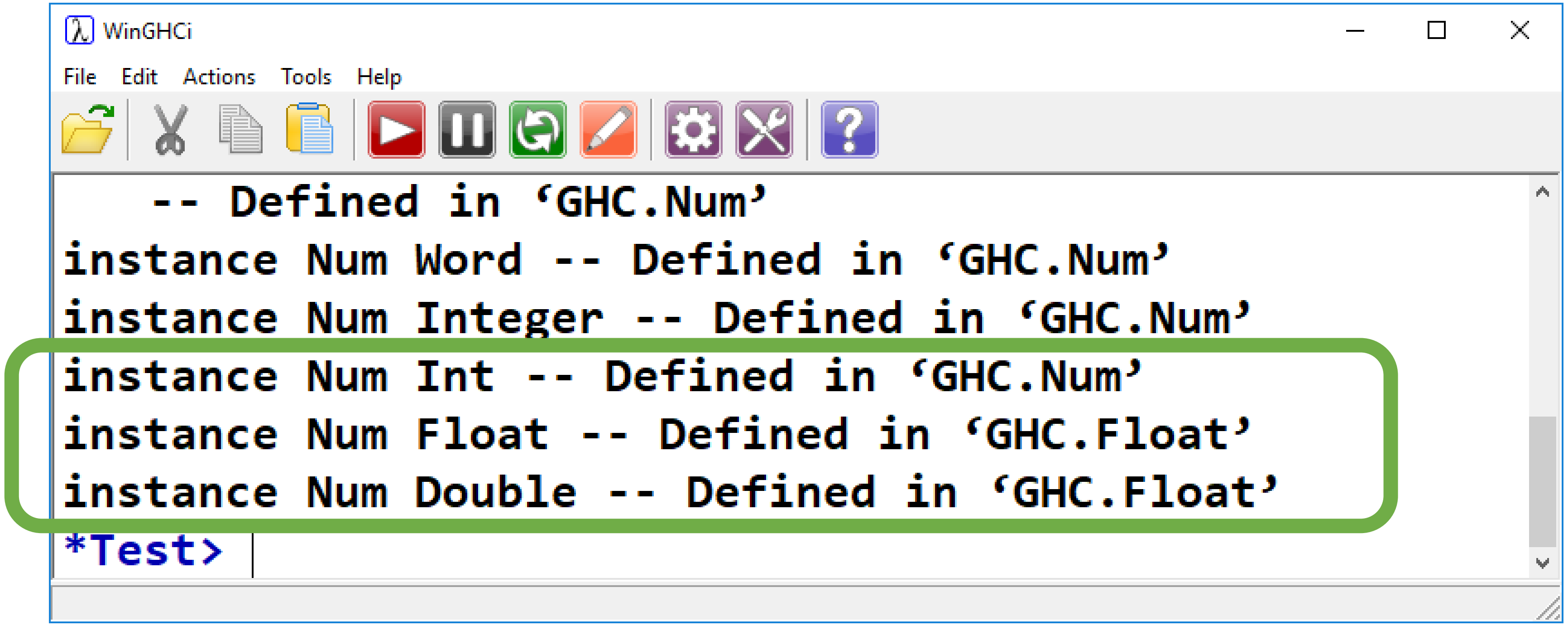
- Whatever type we pass in (**a**), it must be an instance of both Ord and Num.
- **Int** is one such type, as is **Float**

# 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 Bool -- Defined in ‘GHC.Classes’~~
instance (Ord a, Ord b, Ord c, Ord d, Ord e, Ord f, O
```

# Num:



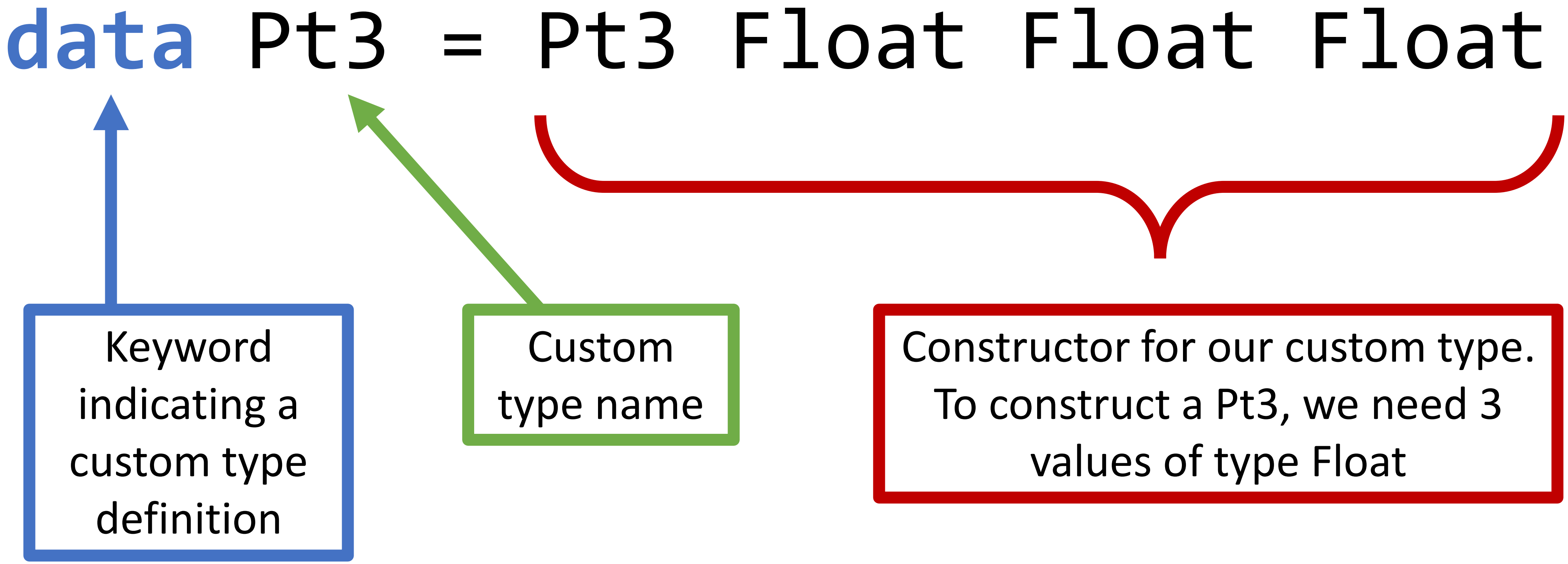
```
    -- Defined in ‘GHC.Num’
instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
*Test>
```

# 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

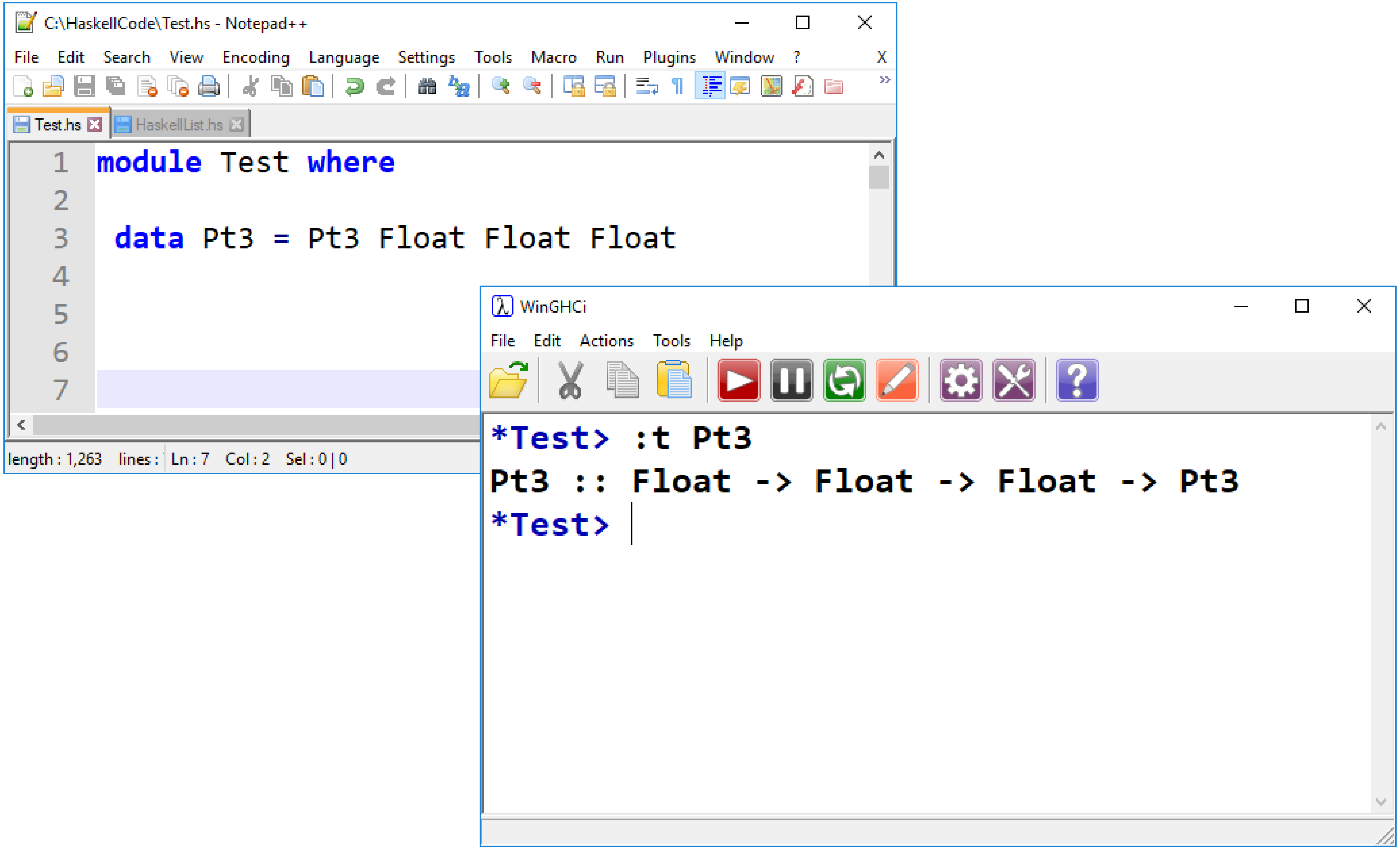


```
data Pt3 = Pt3 Float Float Float
```

Keyword indicating a custom type definition

Custom type name

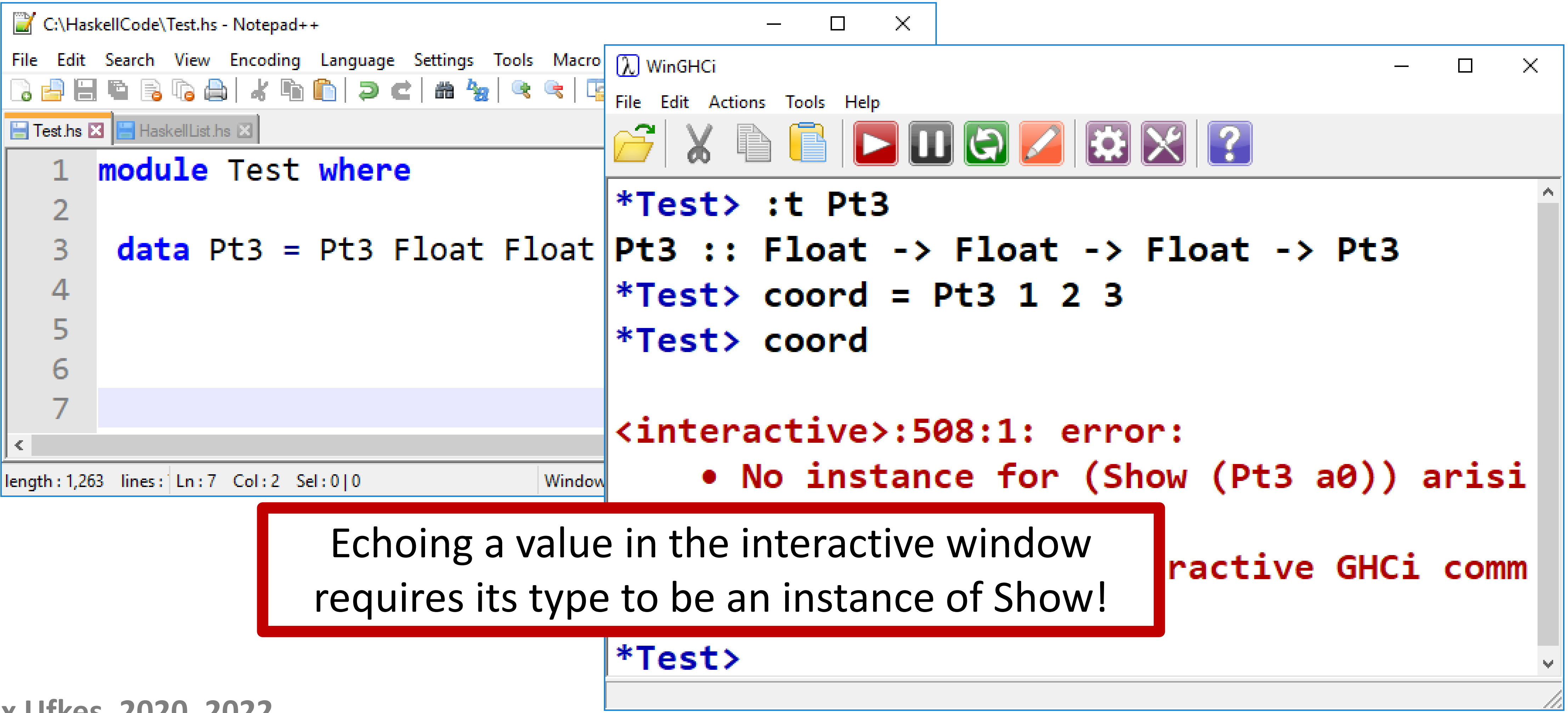
Constructor for our custom type. To construct a Pt3, we need 3 values of type Float


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
length : 1,263 lines : Ln : 7 Col : 2 Sel : 0 | 0
WinGHCi
File Edit Actions Tools Help

```haskell
module Test where

 data Pt3 = Pt3 Float Float Float
```

```
*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
length : 1,263 lines : Ln : 7 Col : 2 Sel : 0 | 0 Window
WinGHCi
File Edit Actions Tools Help

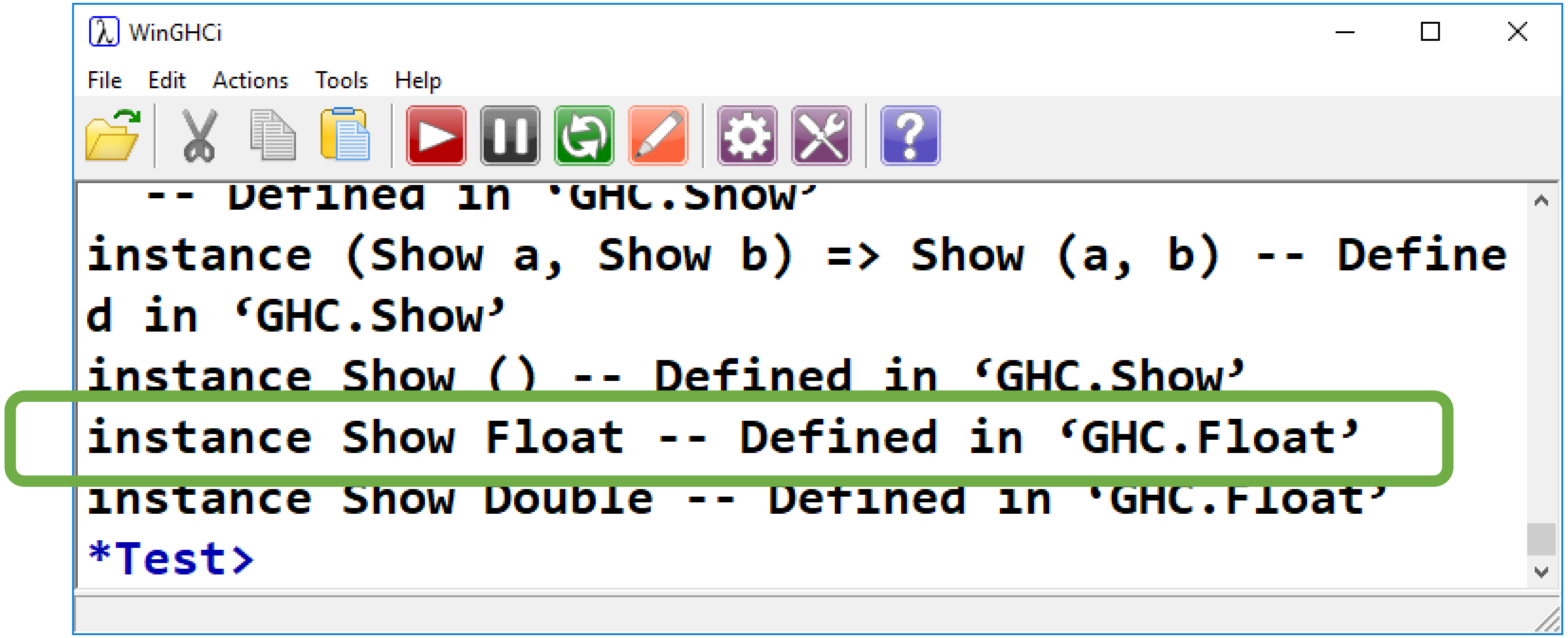
```haskell
module Test where

 data Pt3 = Pt3 Float Float
```

```
*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>
```

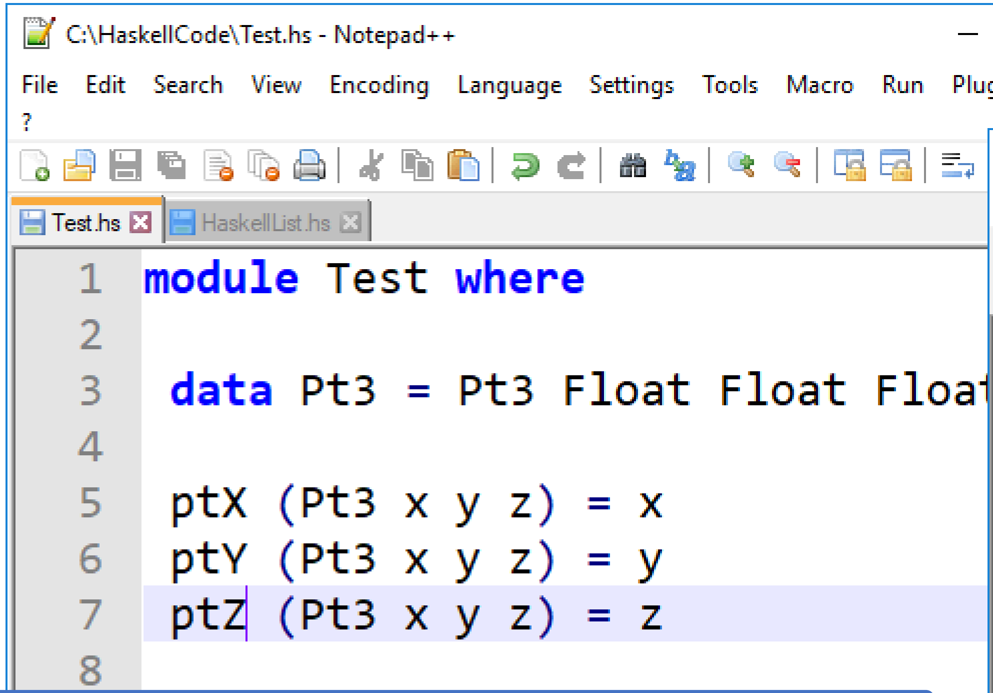
Echoing a value in the interactive window requires its type to be an instance of Show!

# Hmmm…

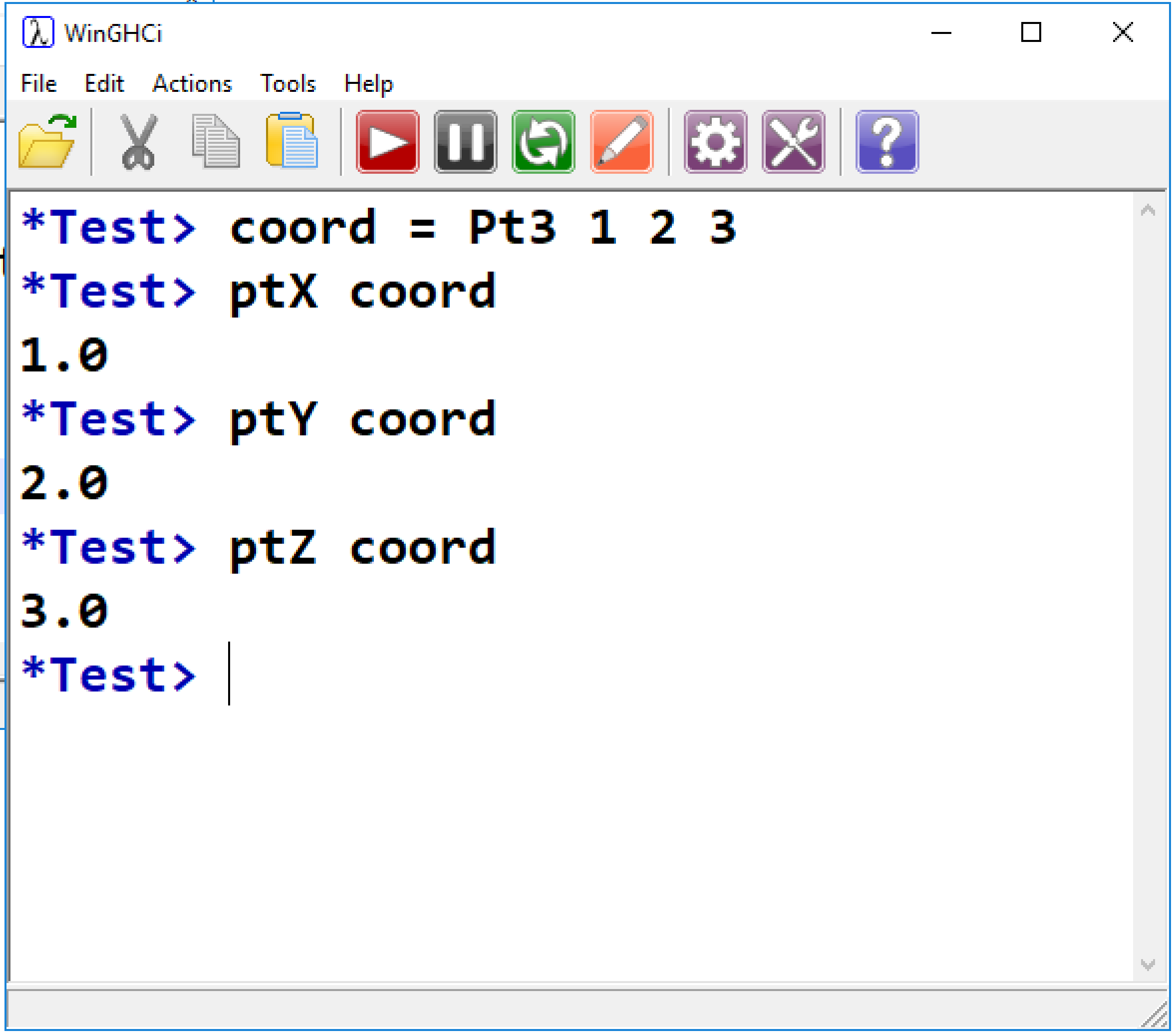
- The values contained in Pt3 are Float, and we know that Float is an instance of Show.
- How can we access the individual elements of Pt3?



```
  -- Defined in ‘GHC.Show’
instance (Show a, Show b) => Show (a, b) -- Define
d in ‘GHC.Show’
instance Show () -- Defined in ‘GHC.Show’
instance Show Float -- Defined in ‘GHC.Float’
instance Show Double -- Defined in ‘GHC.Float’
*Test>
```


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs

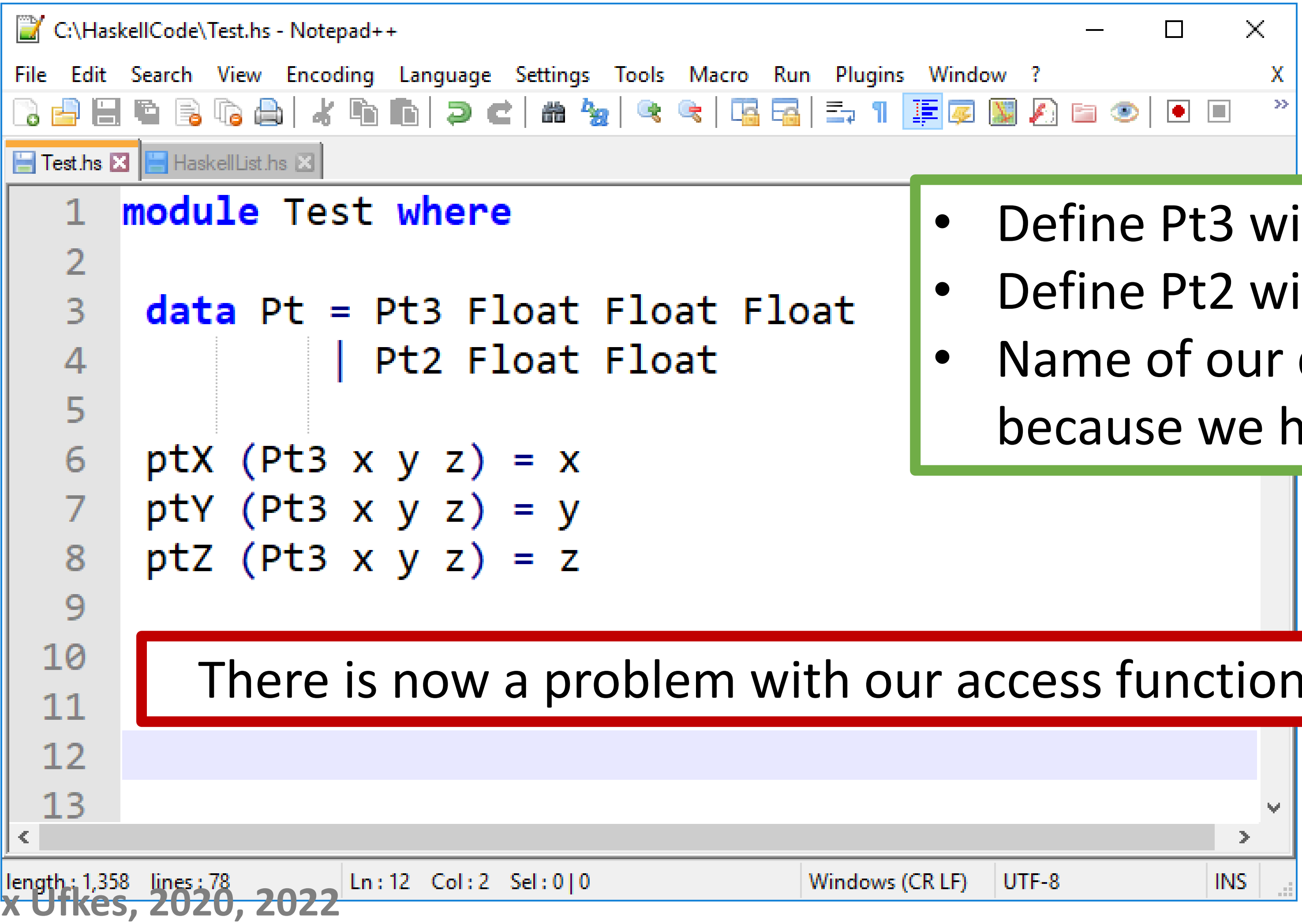
```haskell
module Test where

 data Pt3 = Pt3 Float Float Floa

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

```



```
*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

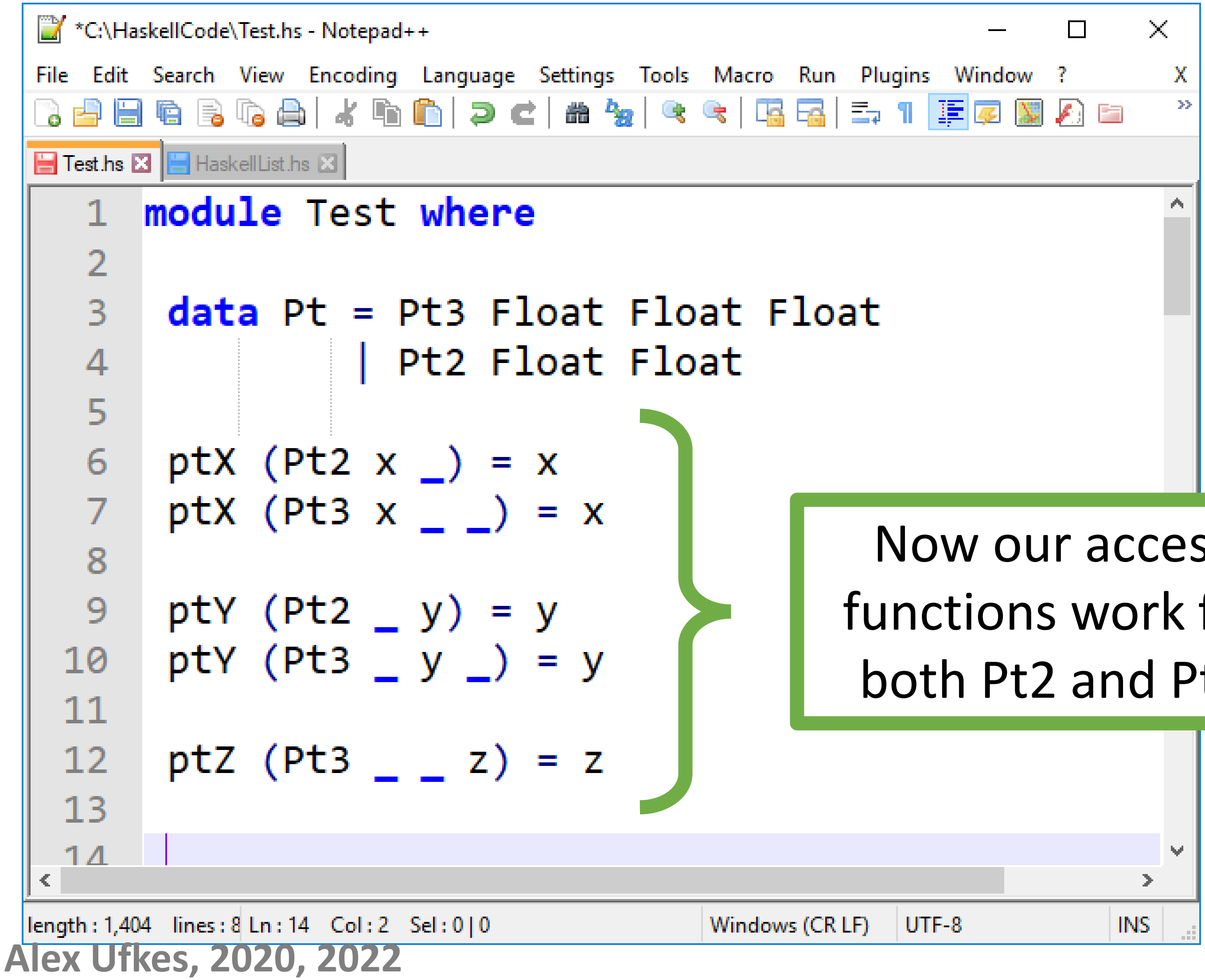

C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
length : 1,358 lines : 78
Ln : 12 Col : 2 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS

```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
```

- 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.

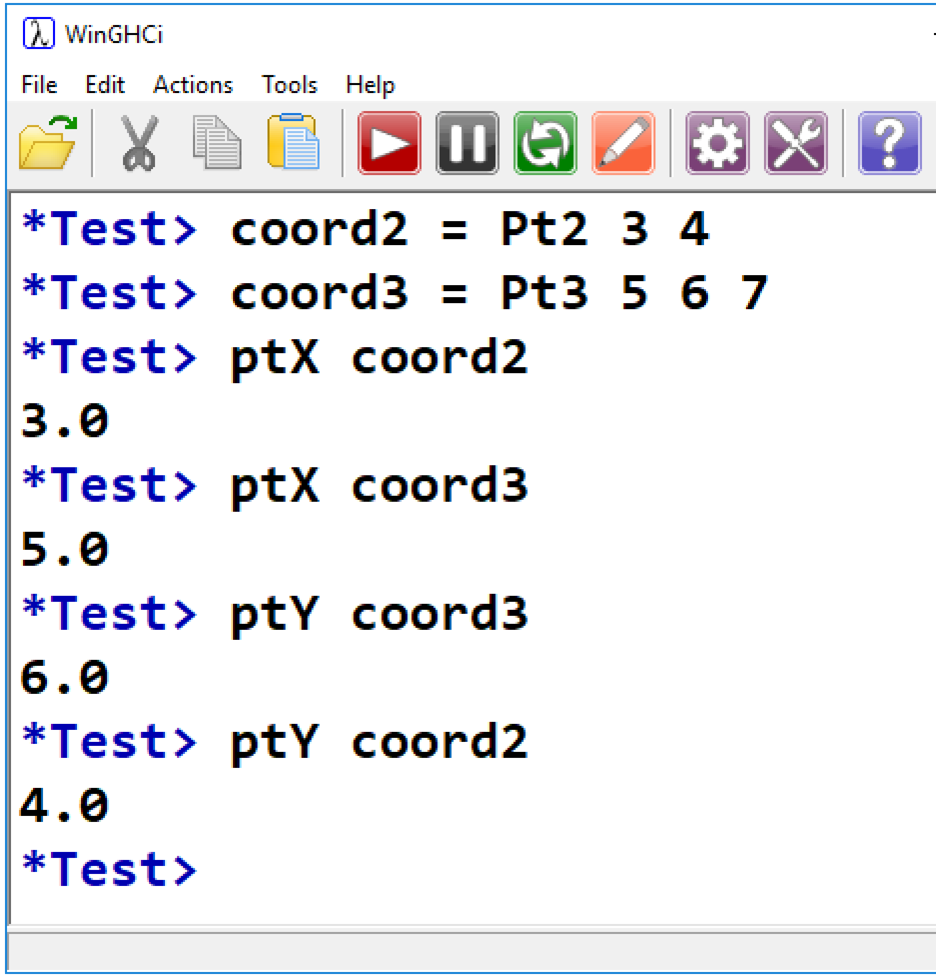
There is now a problem with our access functions

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

```haskell
module Test where

 data Pt = Pt3 Float Float Float
         | Pt2 Float Float

 ptX (Pt2 x _) = x
 ptX (Pt3 x _ _) = x

 ptY (Pt2 _ y) = y
 ptY (Pt3 _ y _) = y

 ptZ (Pt3 _ _ z) = z

```

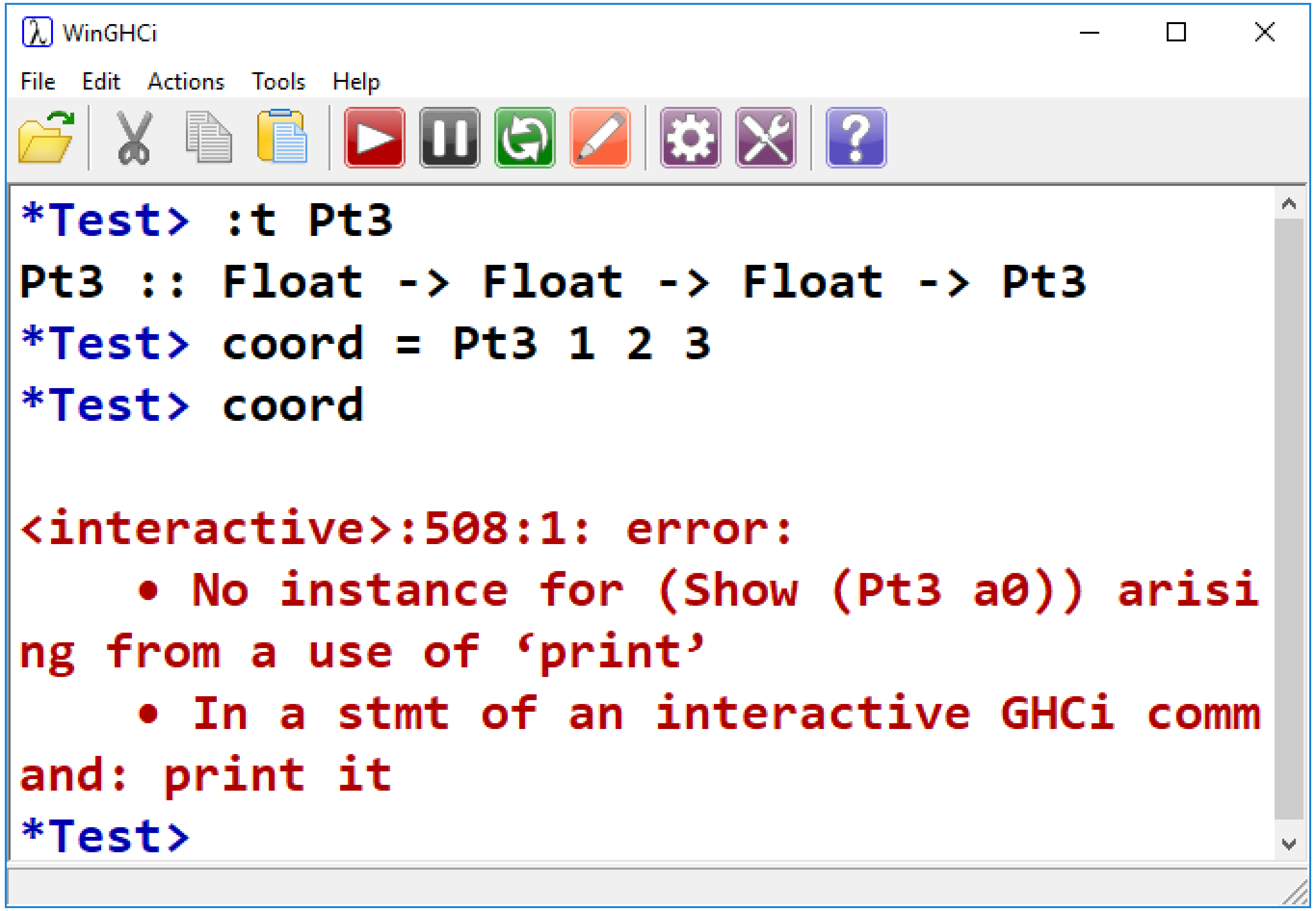
Now our access functions work for both Pt2 and Pt3



```
*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>
```

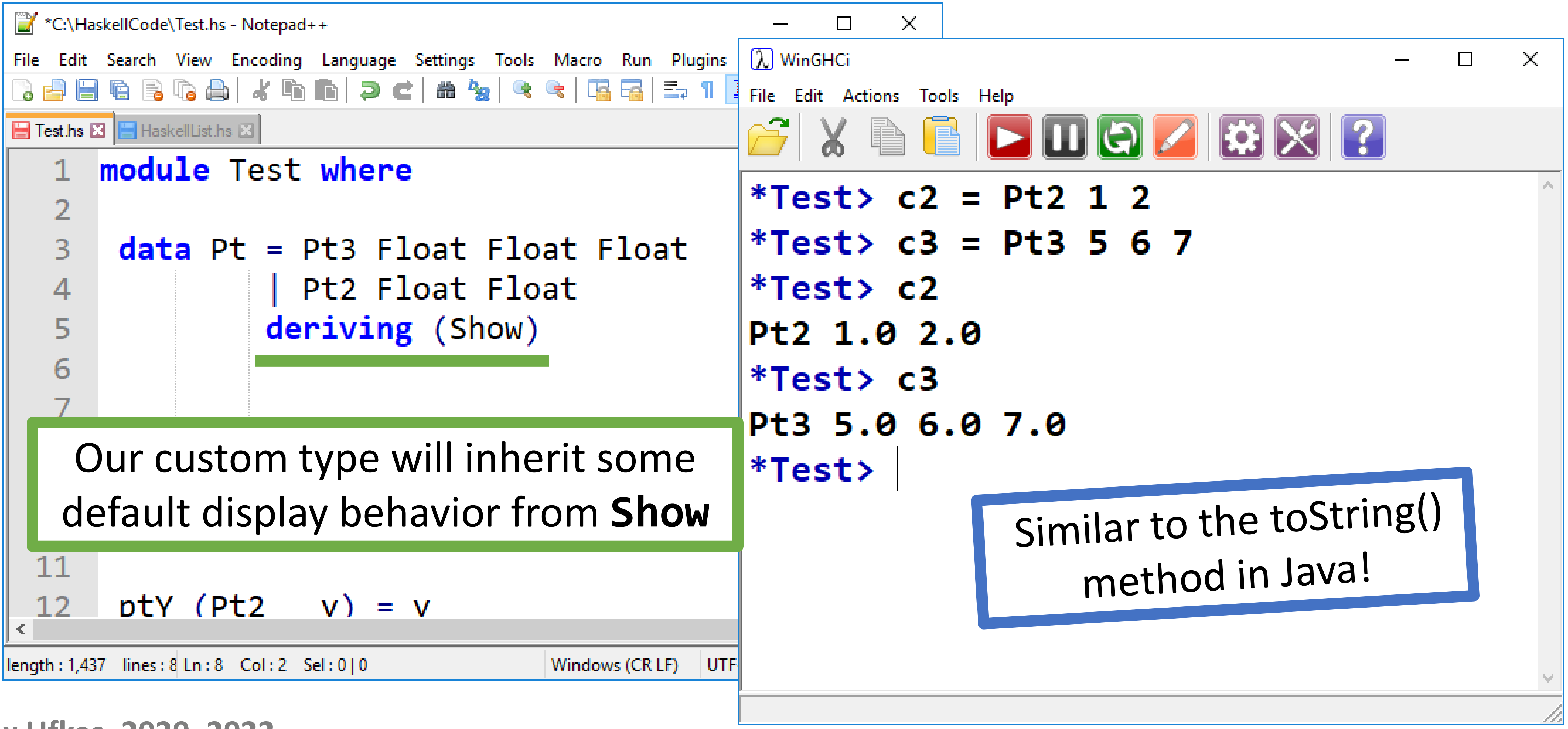
# Deriving Show

Recall:



```
*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
ng from a use of ‘print’
    • In a stmt of an interactive GHCi comm
and: print it
*Test>
```

# Deriving **Show**


*C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins
Test.hs
HaskellList.hs
length : 1,437 lines : 8 Ln : 8 Col : 2 Sel : 0 | 0
Windows (CR LF)
UTF
WinGHCi
File Edit Actions Tools Help

```haskell
module Test where

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

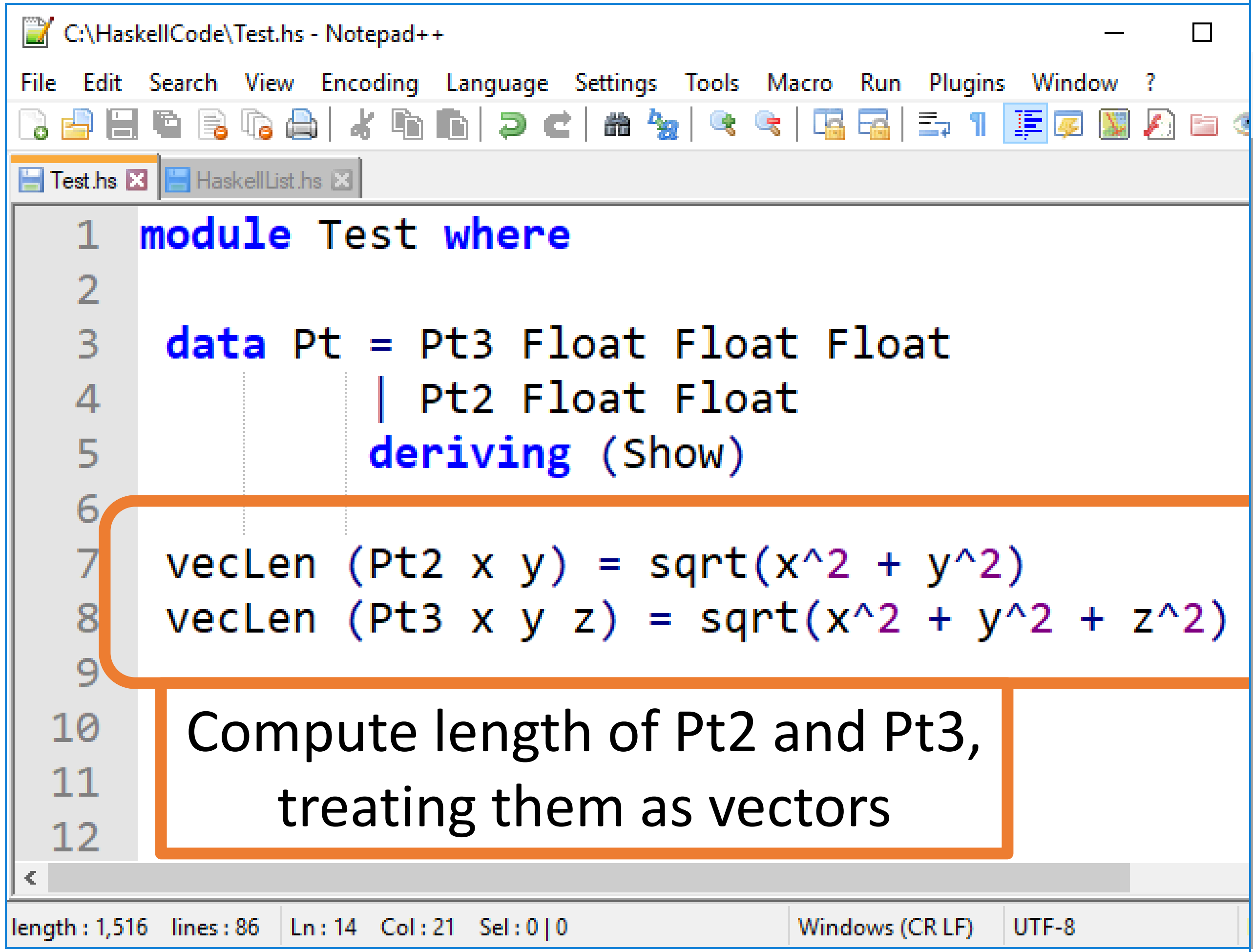
```haskell
 ptY (Pt2   y) = y
```

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

```
*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>
```

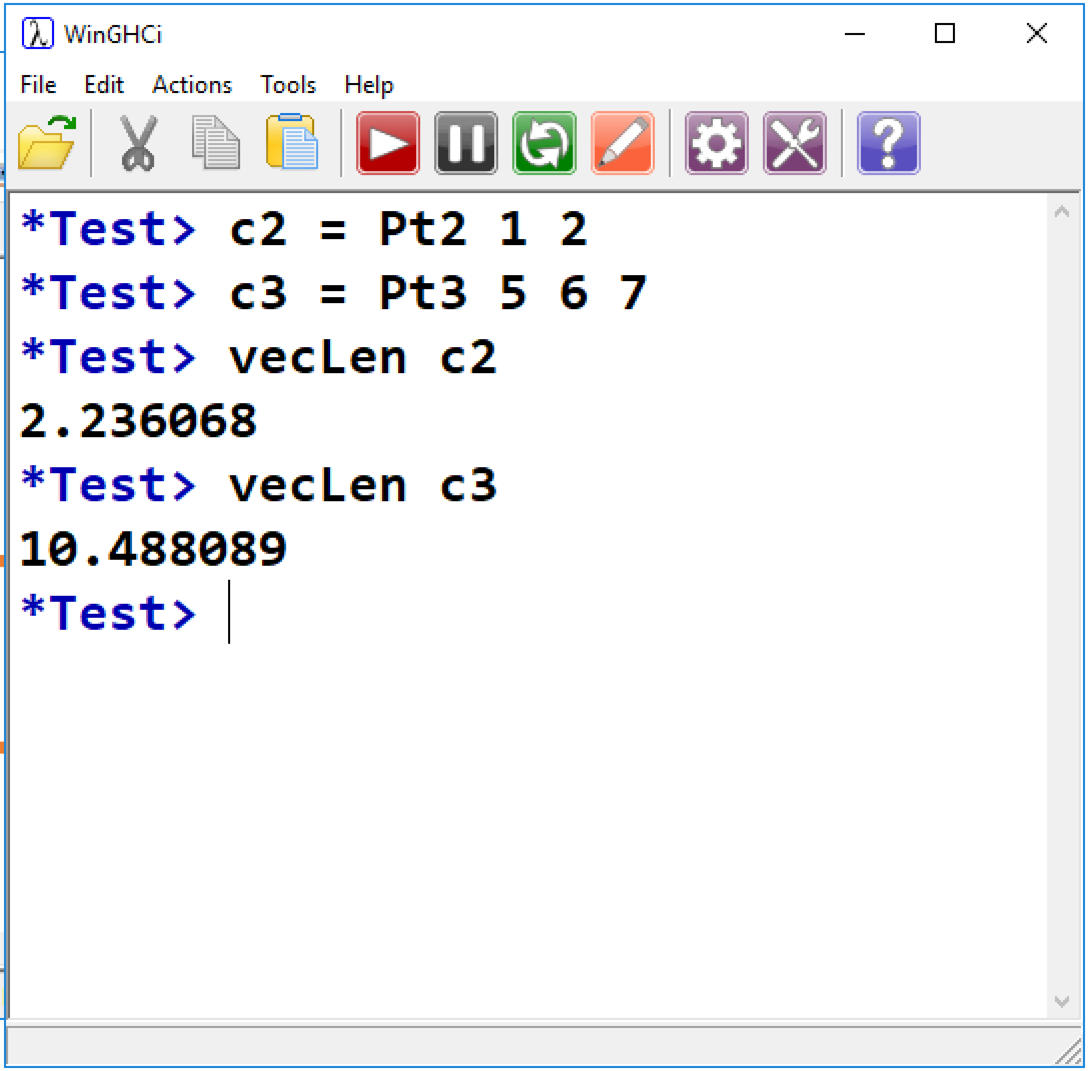
Similar to the toString() method in Java!

# More Advanced Functions


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
length : 1,516 lines : 86 Ln : 14 Col : 21 Sel : 0 | 0 Windows (CR LF) UTF-8

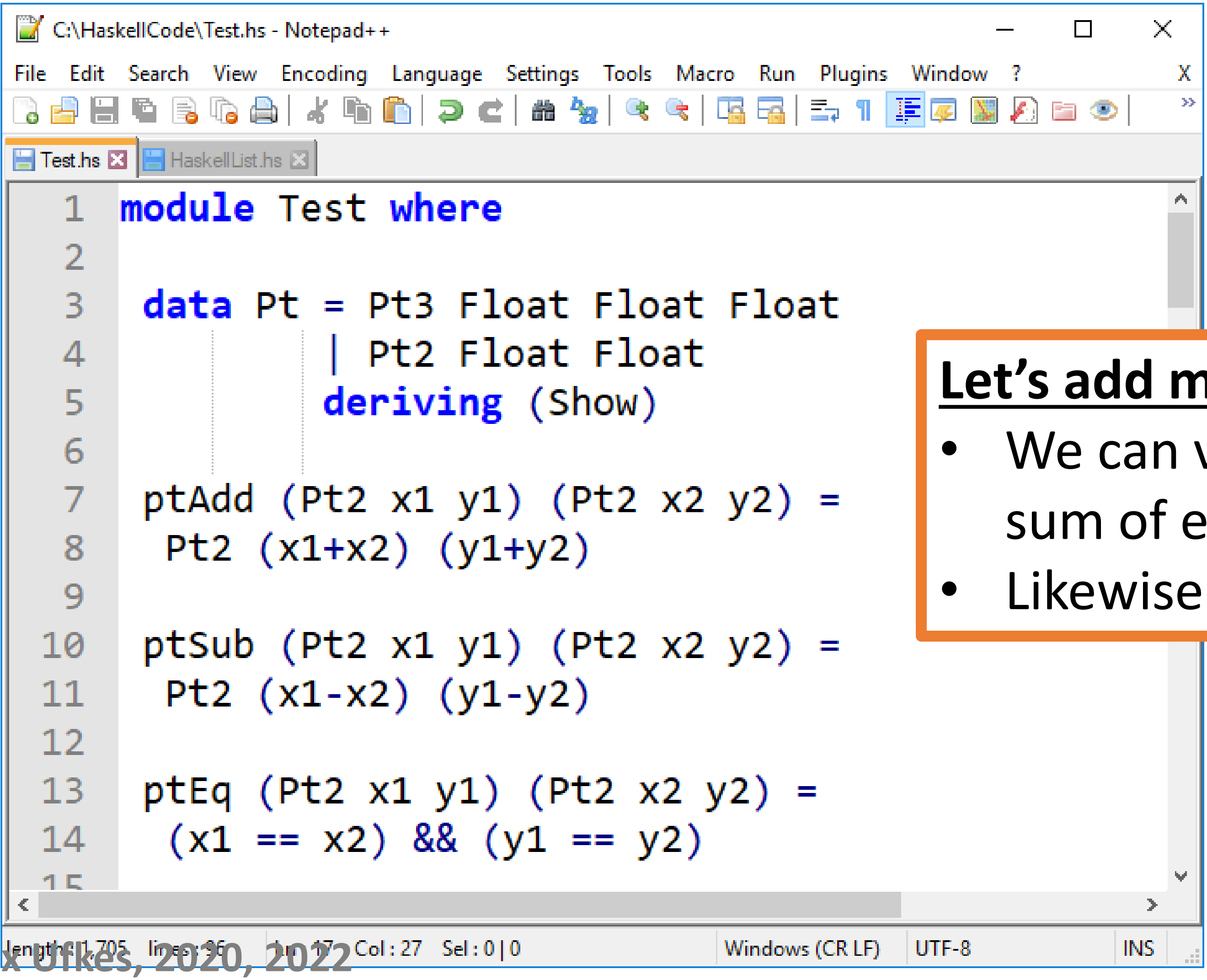
```haskell
module Test where

 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



```
*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?

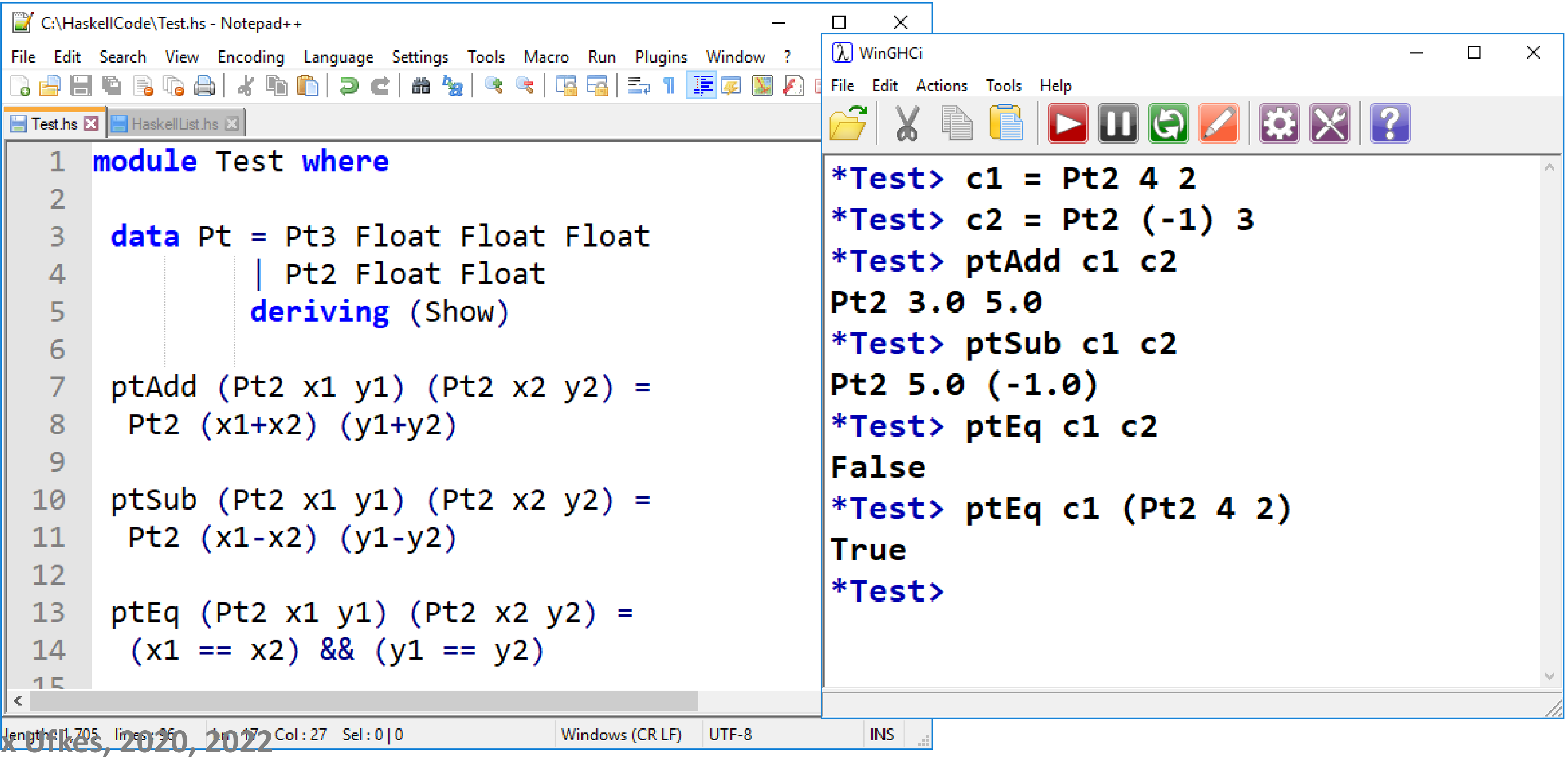

C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
Col : 27 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS

```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)
```

**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++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
length : 1,705 lines : 36 Ln : 17 Col : 27 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS
WinGHCi
File Edit Actions Tools Help

```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)
```

```
*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>
```

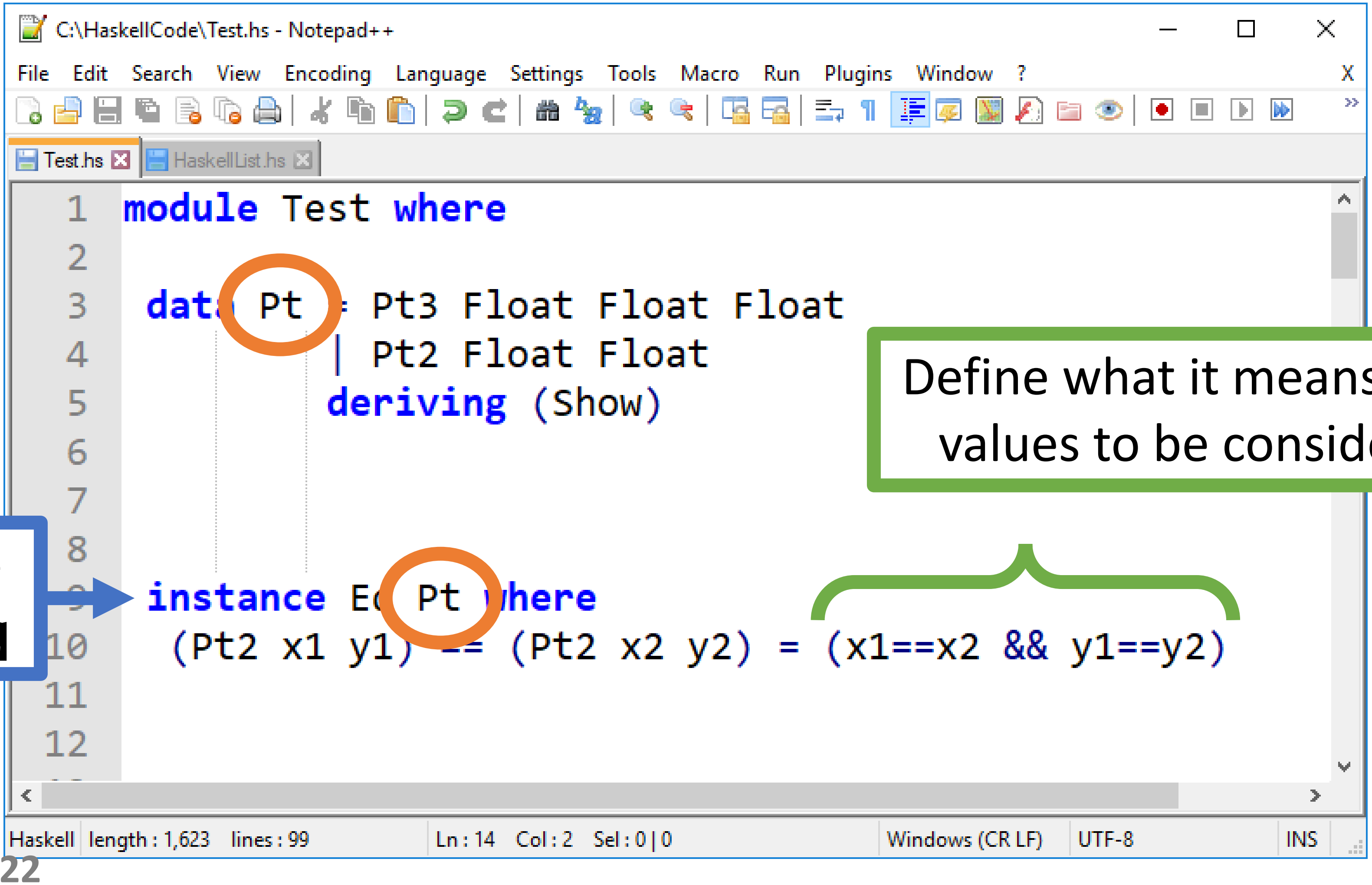
# Addition, Subtraction, Equality?

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?

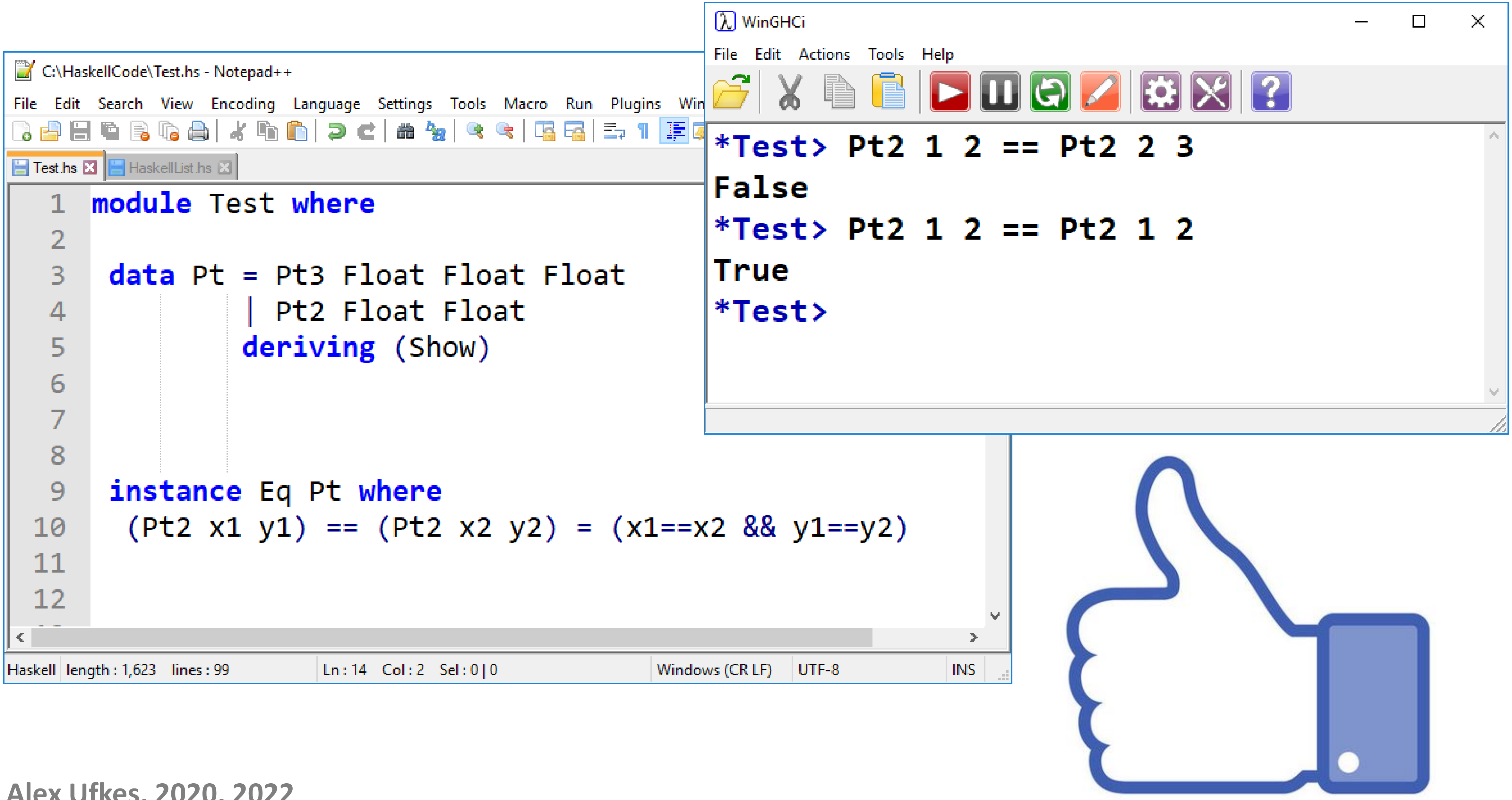
# Custom Types & Type Classes


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
Haskell length : 1,623 lines : 99 Ln : 14 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

```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)
```

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

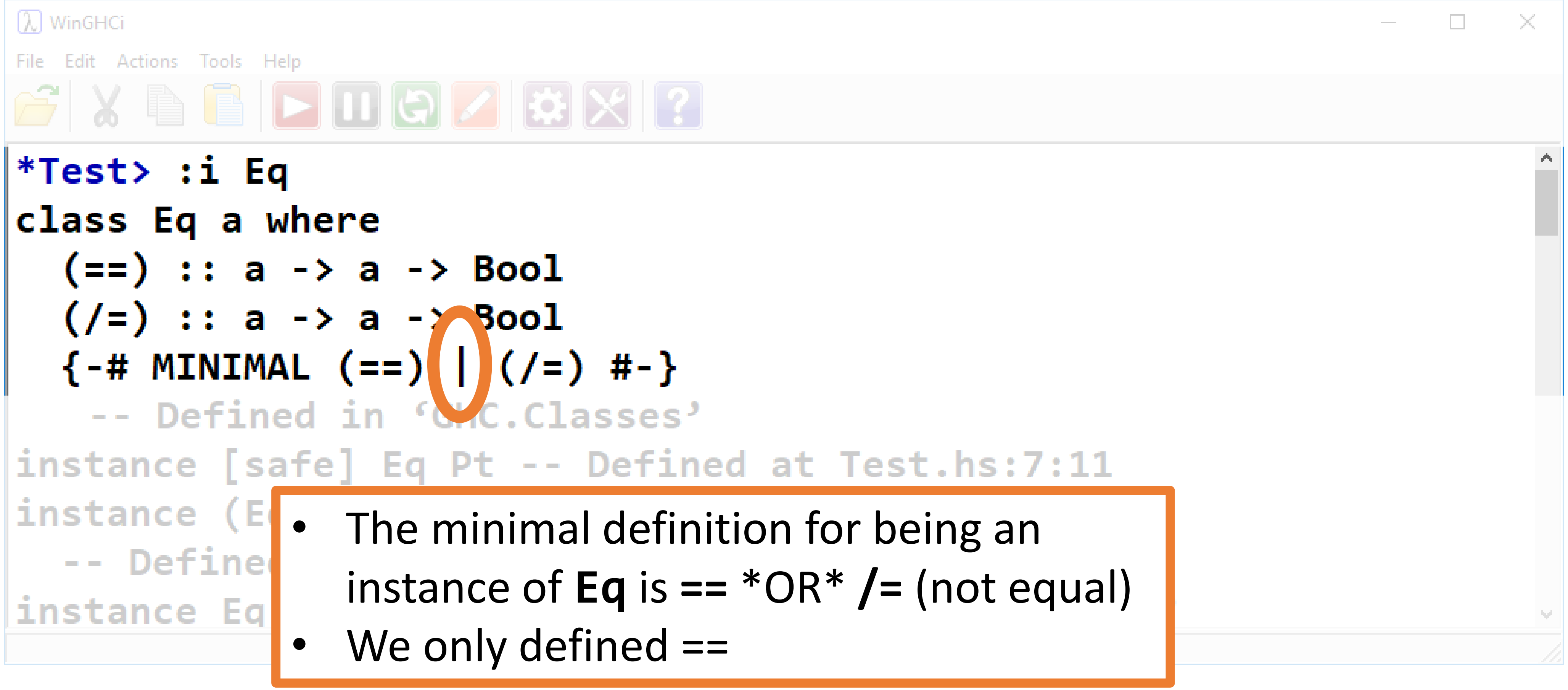
Define what it means for two Pt2 values to be considered equal


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Win
Test.hs
HaskellList.hs
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

```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)


```

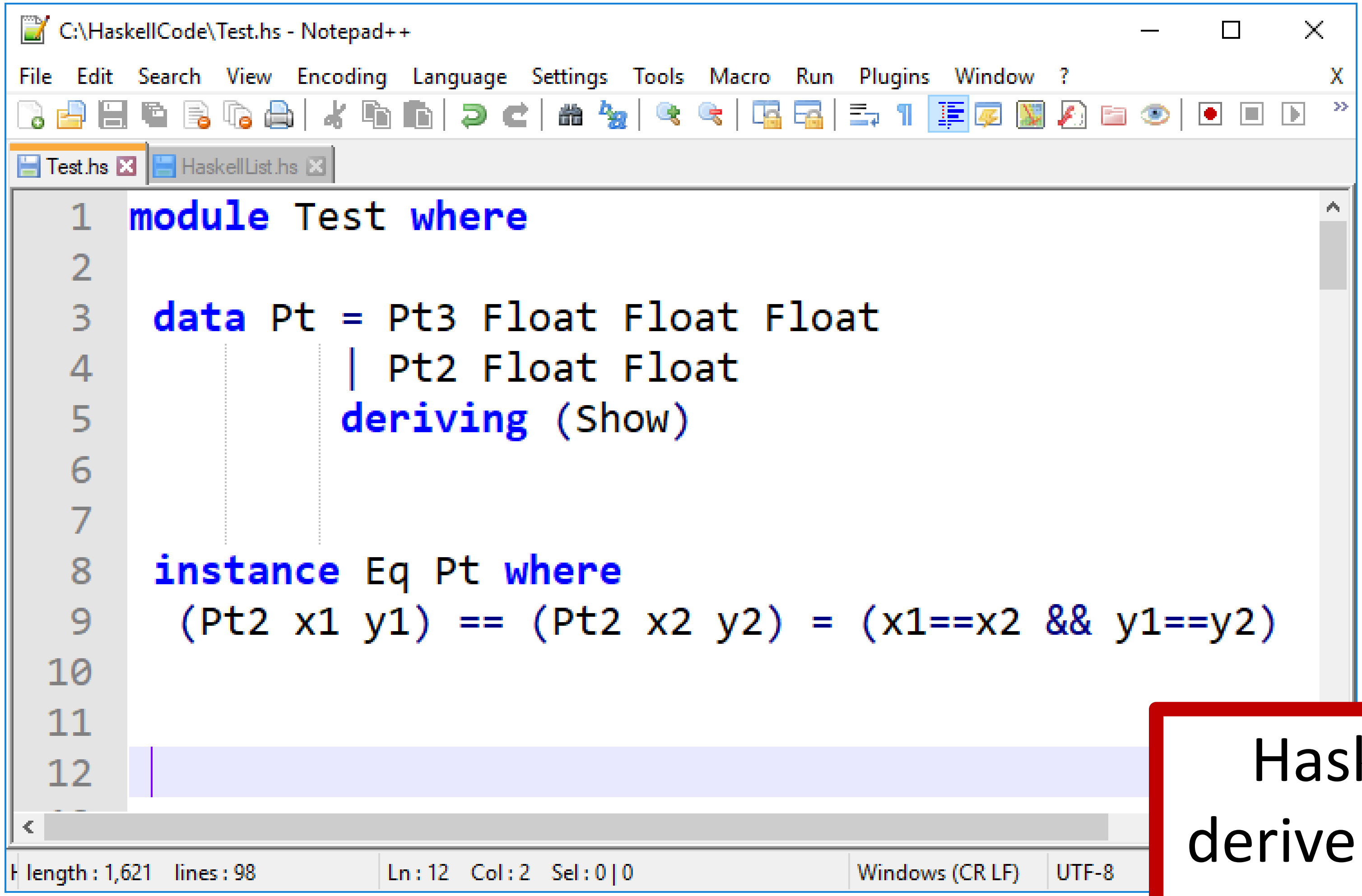
```
*Test> Pt2 1 2 == Pt2 2 3
False
*Test> Pt2 1 2 == Pt2 1 2
True
*Test>
```

# Minimal Definition

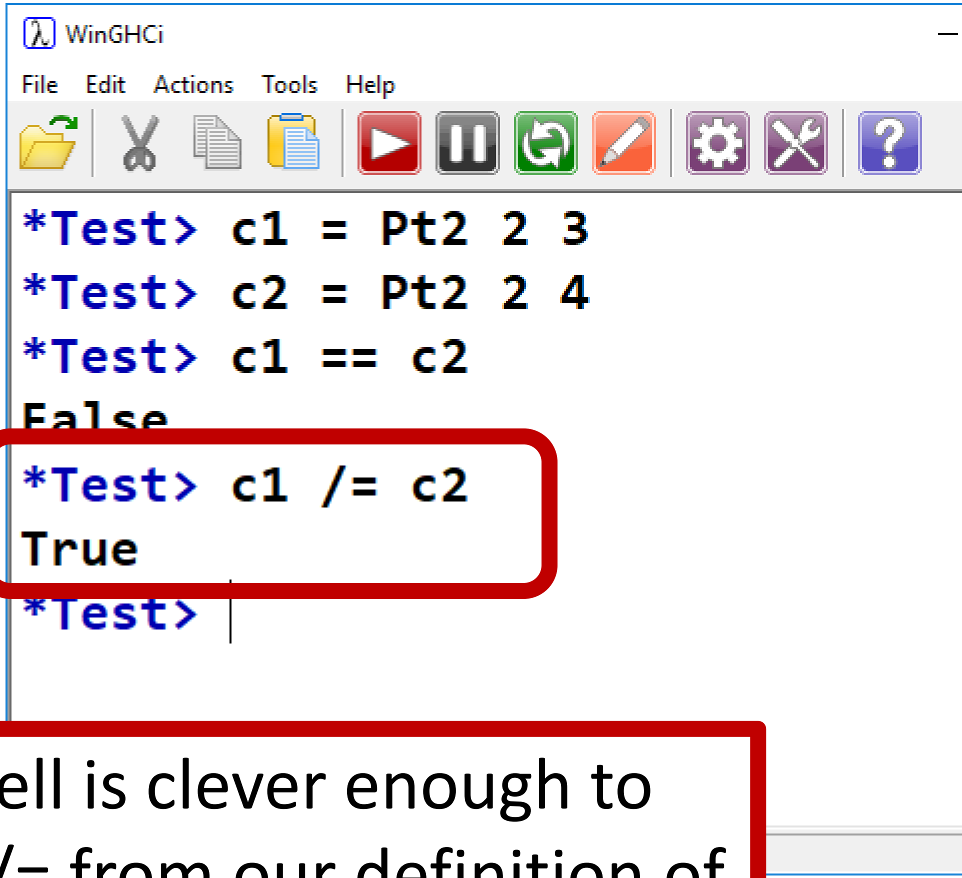


```
*Test> :i Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
    -- Defined in ‘GHC.Classes’
instance [safe] Eq Pt -- Defined at Test.hs:7:11
instance (E
  -- Define
instance Eq
```

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

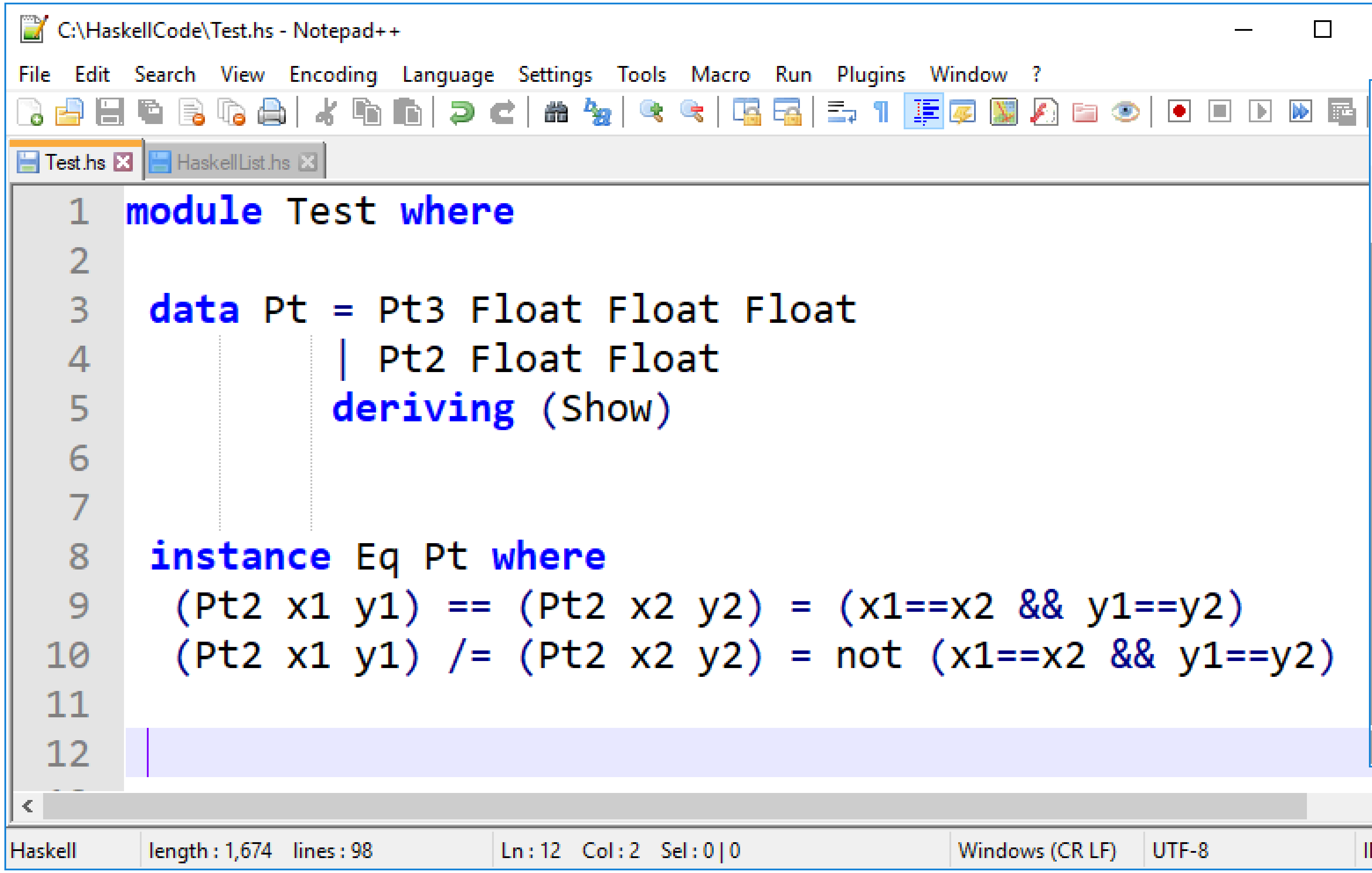
```haskell
module Test where

 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)


```



```
*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.

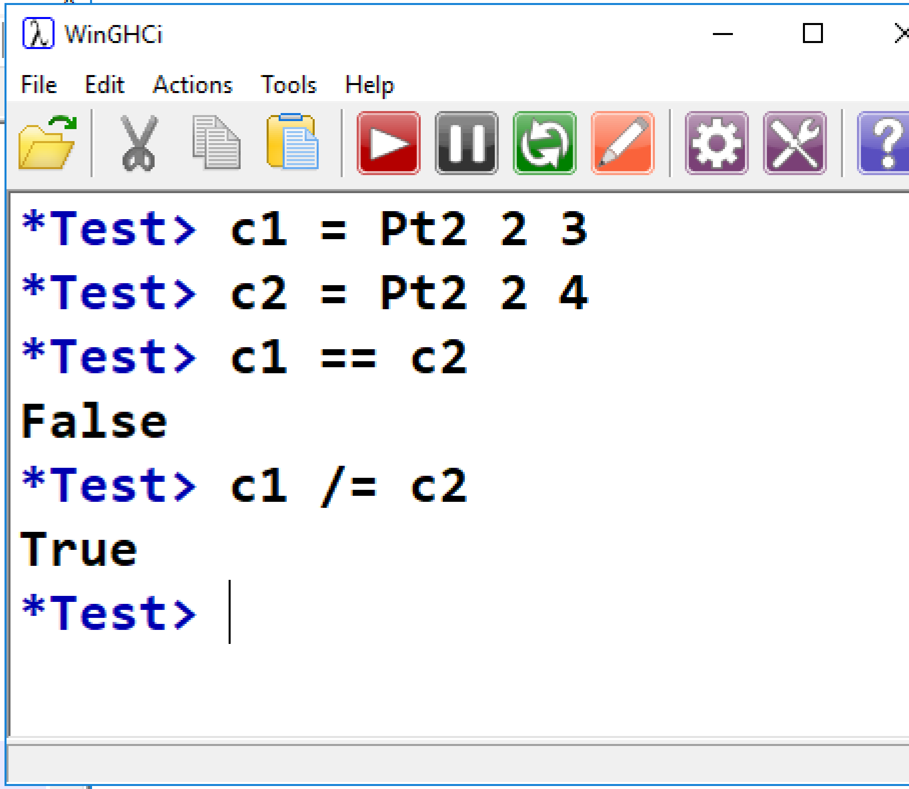
# Let’s Add /= Anyway



C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs

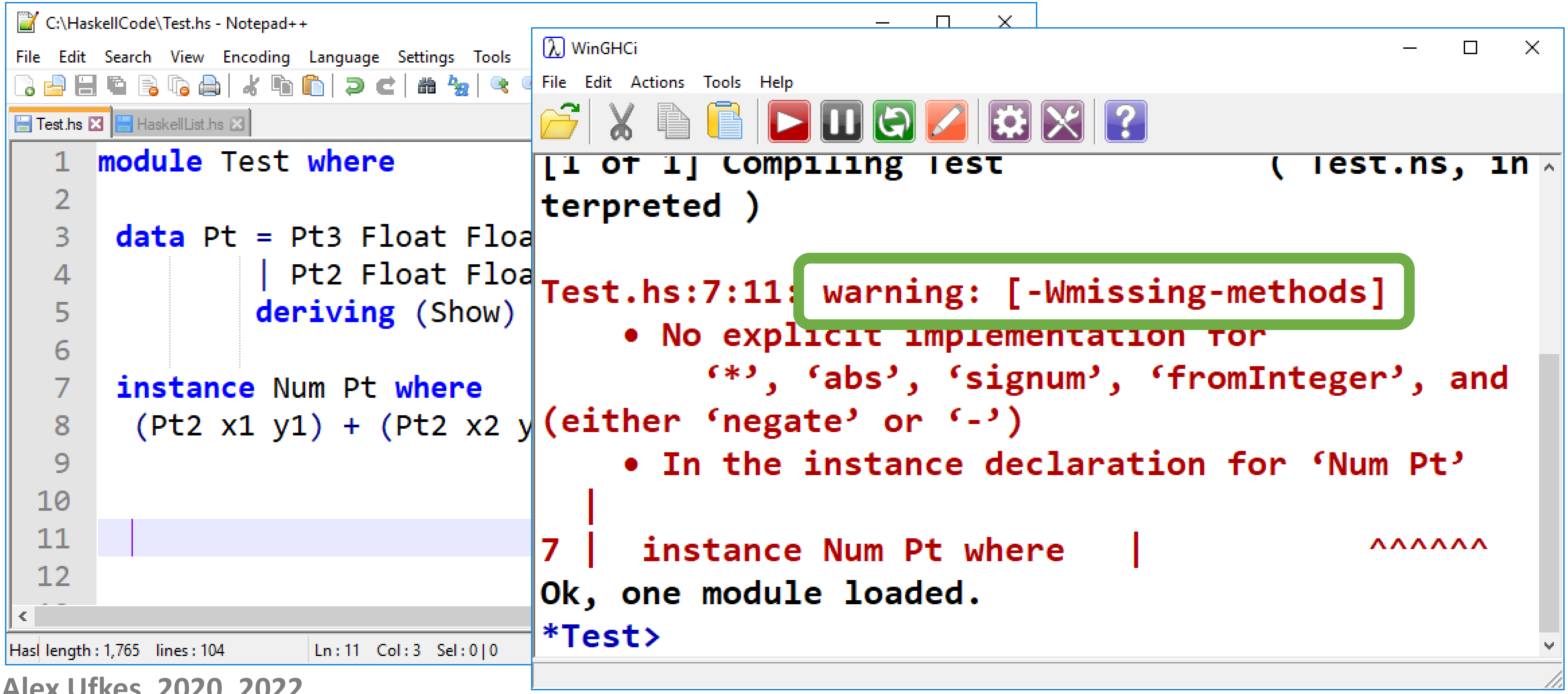
```haskell
module Test where

 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)
  (Pt2 x1 y1) /= (Pt2 x2 y2) = not (x1==x2 && y1==y2)

 
```

Haskell
length : 1,674 lines : 98
Ln : 12 Col : 2 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS



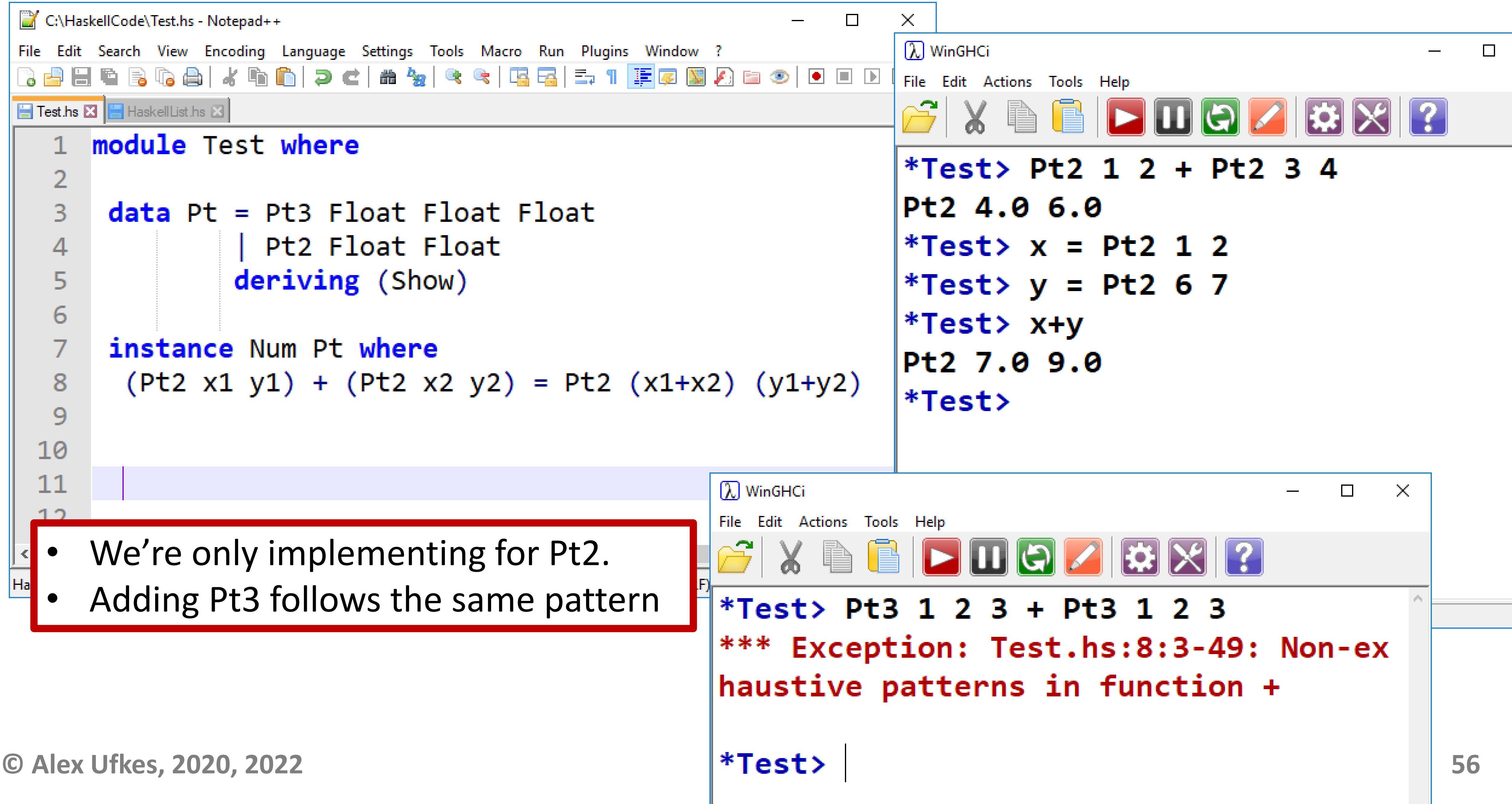
```
*Test> c1 = Pt2 2 3
*Test> c2 = Pt2 2 4
*Test> c1 == c2
False
*Test> c1 /= c2
True
*Test>
```

# Instance of Num


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools
Test.hs
HaskellList.hs
Hasl length : 1,765 lines : 104 Ln : 11 Col : 3 Sel : 0 | 0
WinGHCi
File Edit Actions Tools Help

```haskell
module Test where

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

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

```
[1 of 1] Compiling Test             ( Test.hs, 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
WinGHCi
File Edit Actions Tools Help

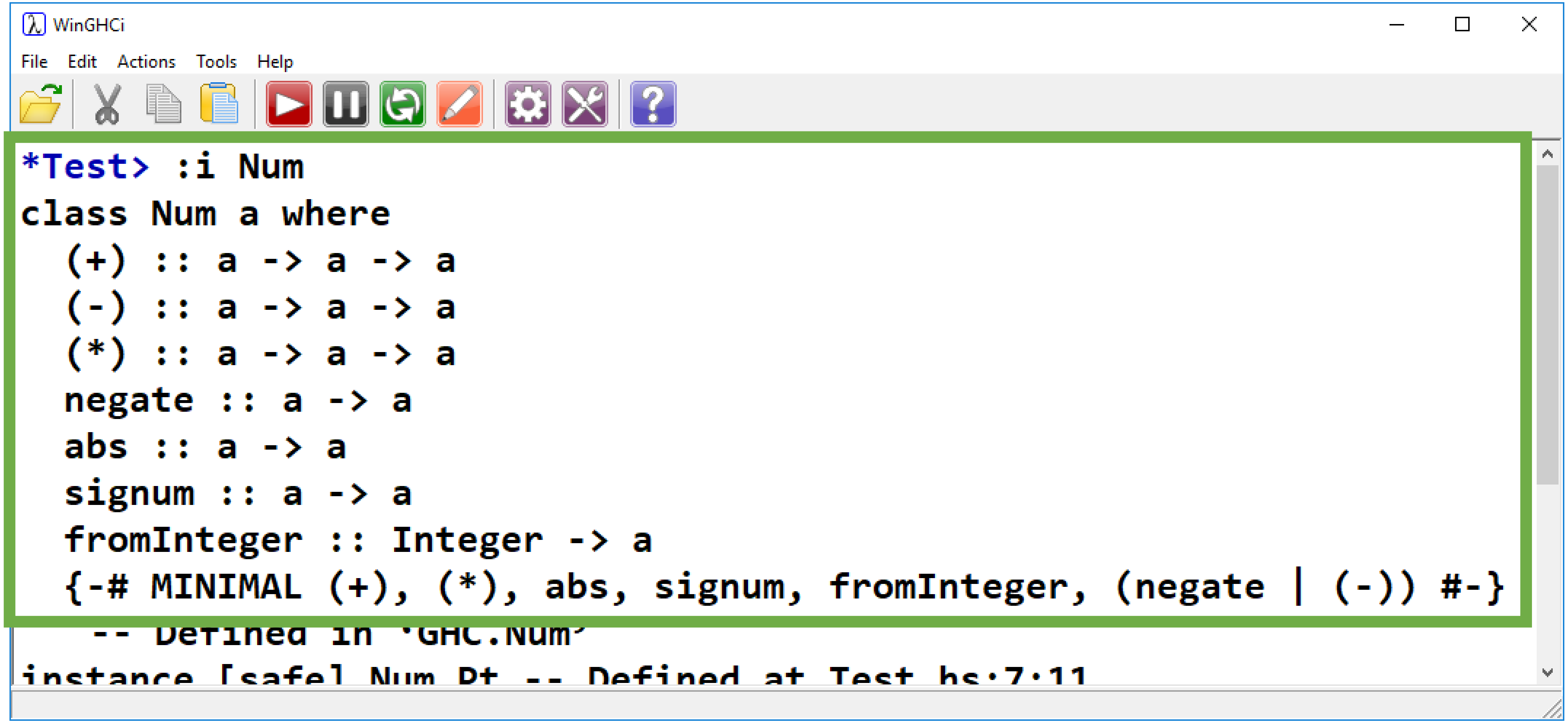
```haskell
1   module Test where
2
3    data Pt = Pt3 Float Float Float
4            | Pt2 Float Float
5            deriving (Show)
6
7    instance Num Pt where
8     (Pt2 x1 y1) + (Pt2 x2 y2) = Pt2 (x1+x2) (y1+y2)
9
10
11
12
```

```
*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>
```

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

*Test>
```

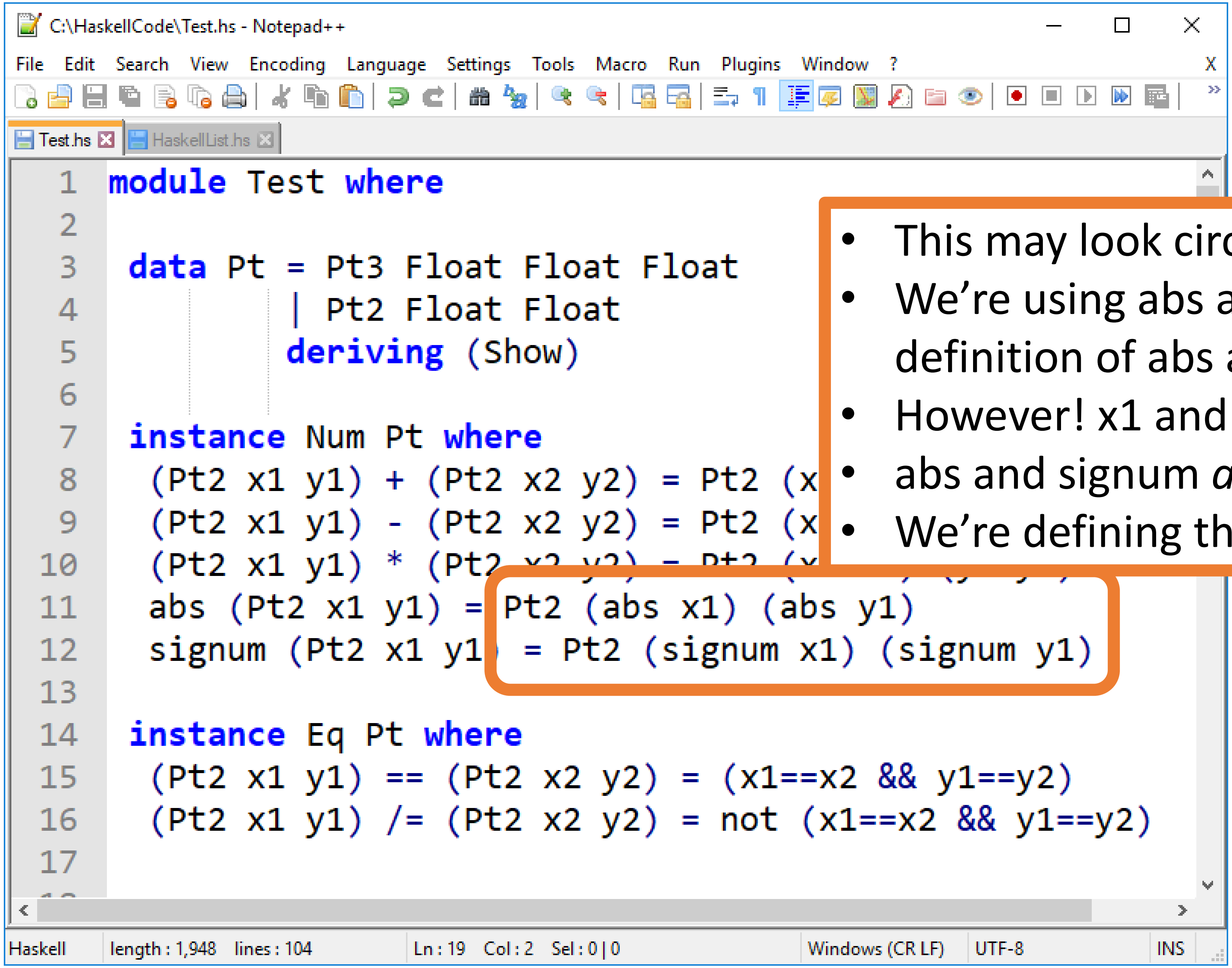
- We’re only implementing for Pt2.
- Adding Pt3 follows the same pattern

# Instance of Num



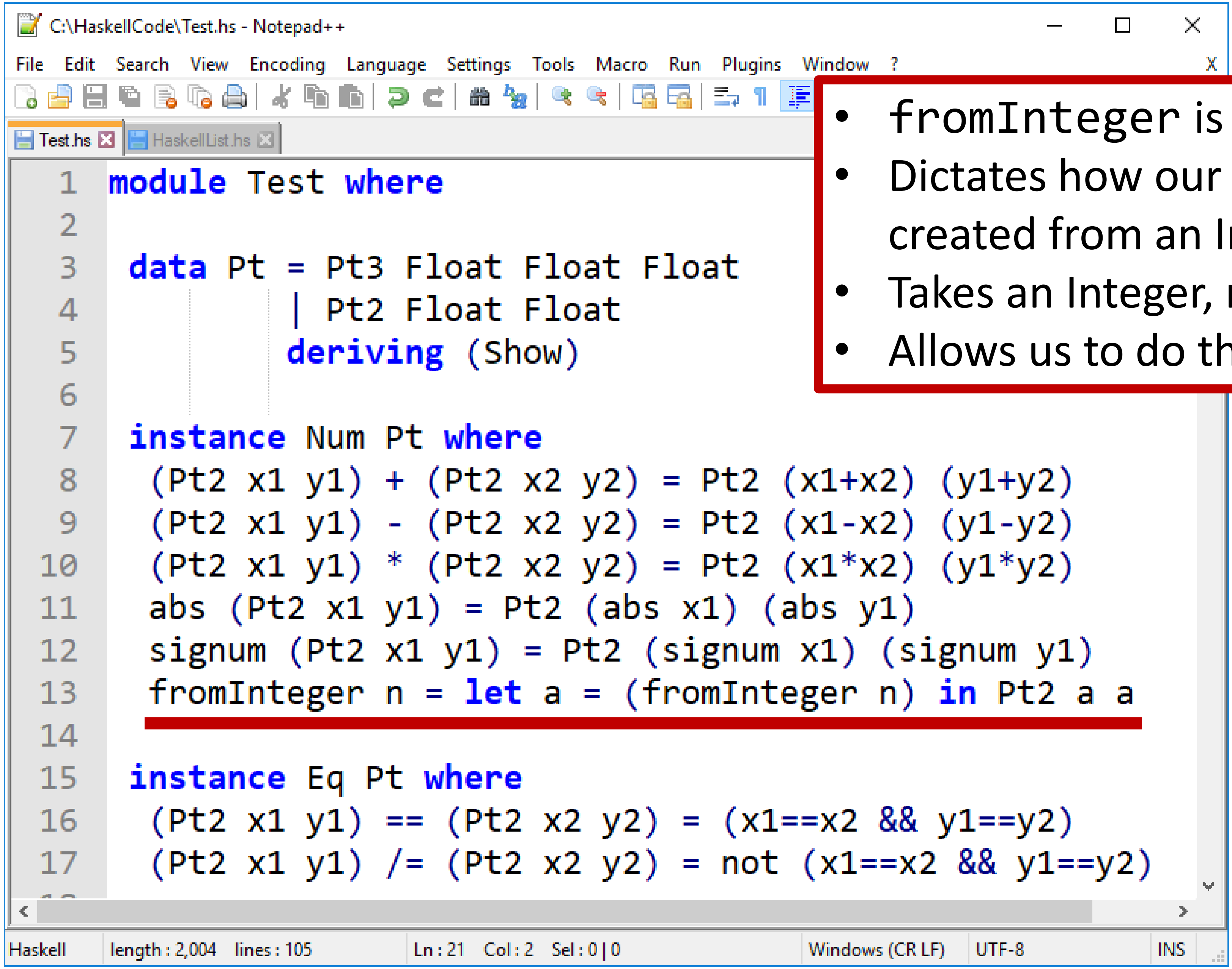
```
*Test> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
    -- Defined in 'GHC.Num'
instance [safe] Num Pt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

© Alex Ufkes, 2020, 2022


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
Haskell length : 1,948 lines : 104 Ln : 19 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

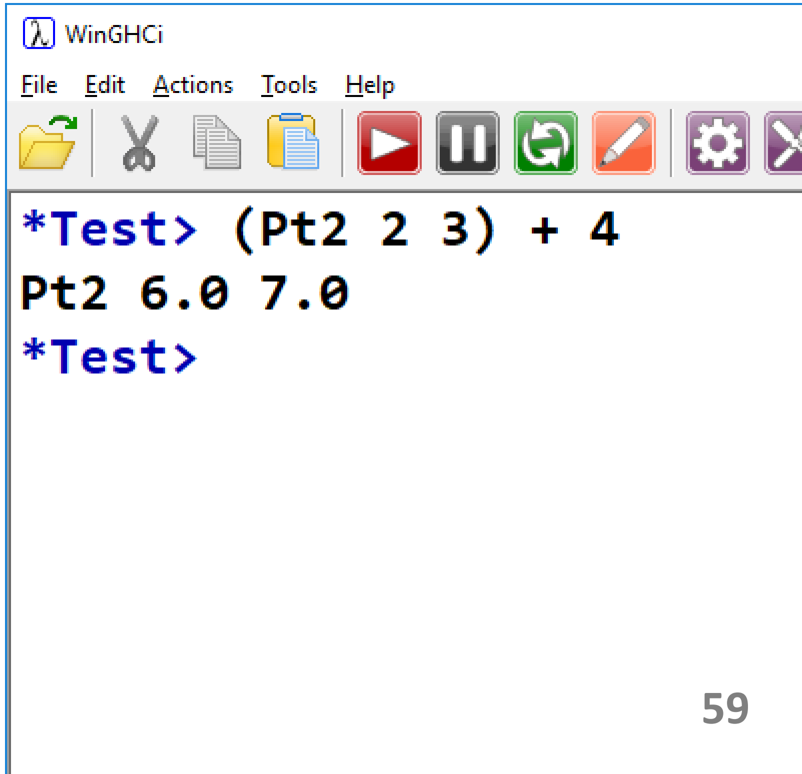
```haskell
module Test where

 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)
```

- 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

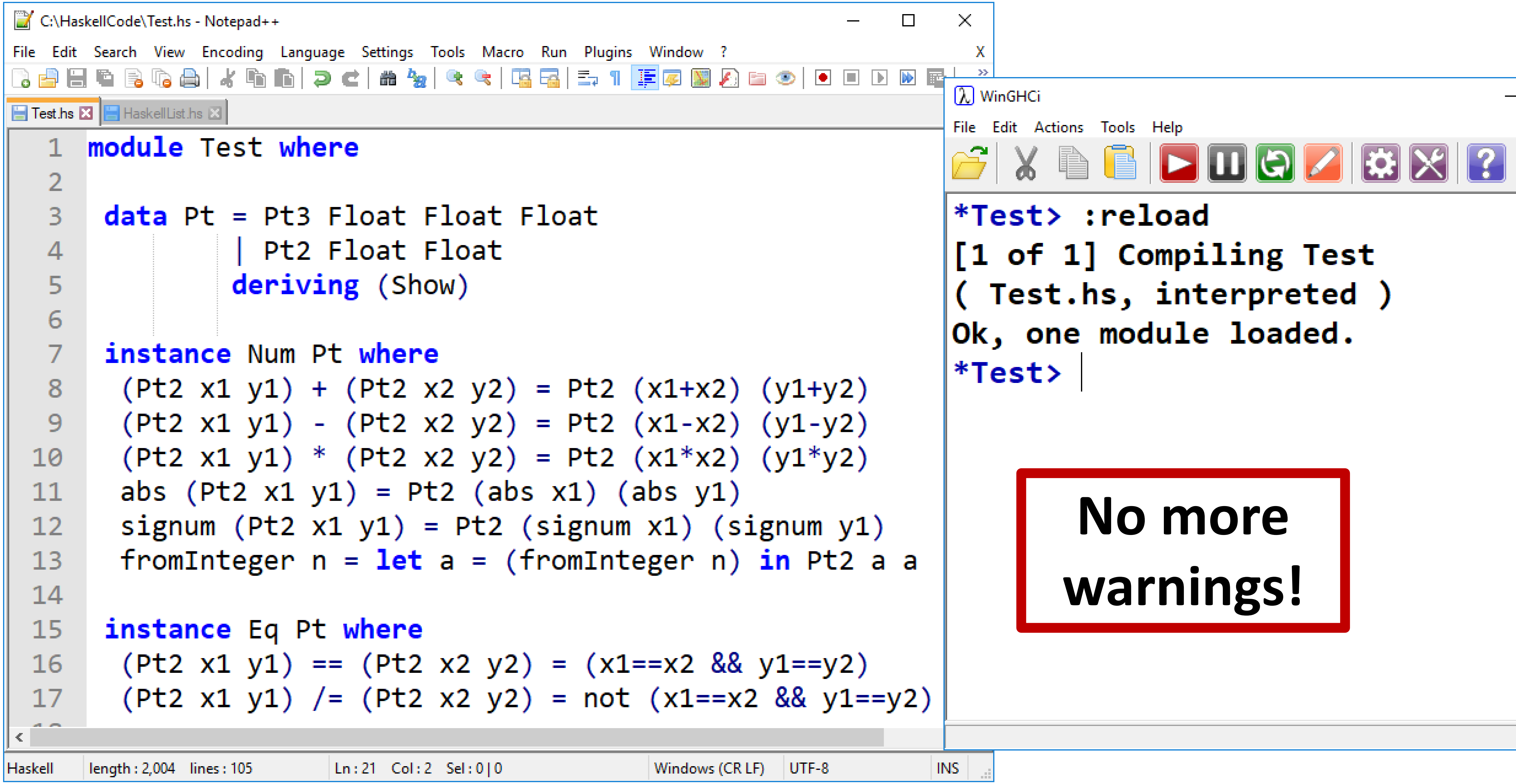

C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
Haskell
length : 2,004 lines : 105
Ln : 21 Col : 2 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS

```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)
```

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



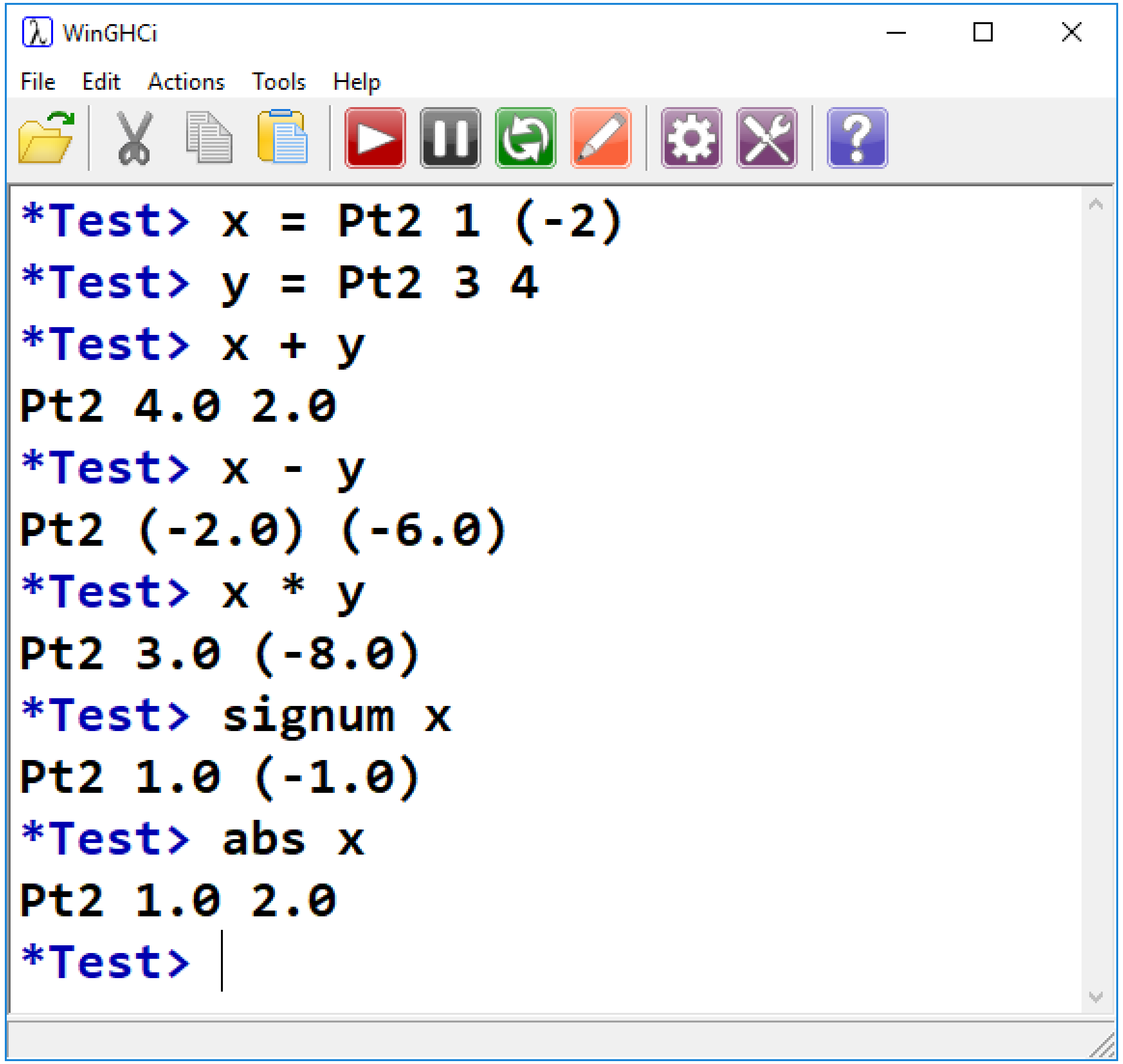
```
*Test> (Pt2 2 3) + 4
Pt2 6.0 7.0
*Test>
```


C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs
HaskellList.hs
Haskell length : 2,004 lines : 105 Ln : 21 Col : 2 Sel : 0 | 0 Windows (CR LF) UTF-8 INS
WinGHCi
File Edit Actions Tools Help

```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)
```

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

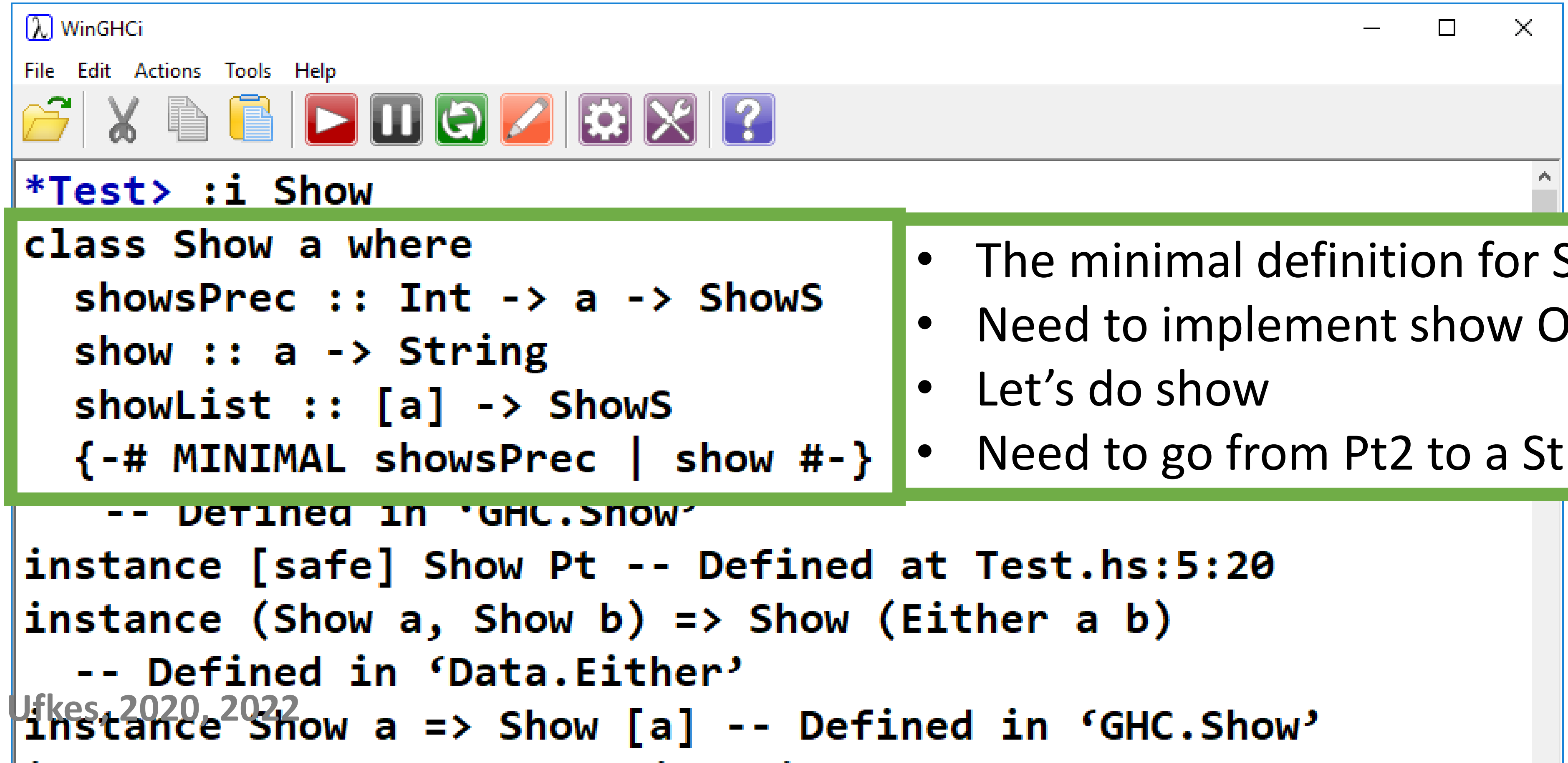
No more warnings!



```
*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> 
```

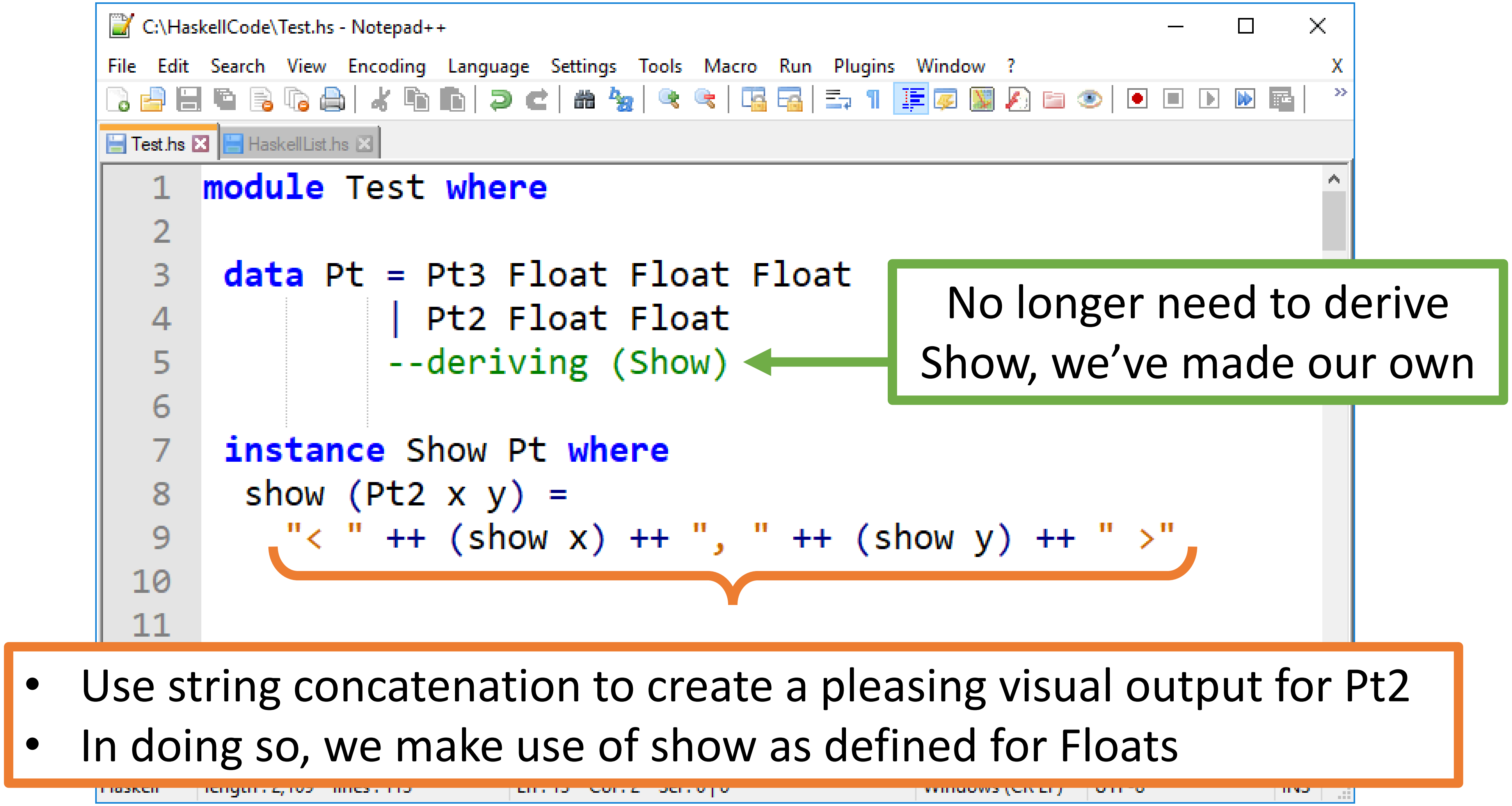
# Instance of Show

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



```
*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’
```

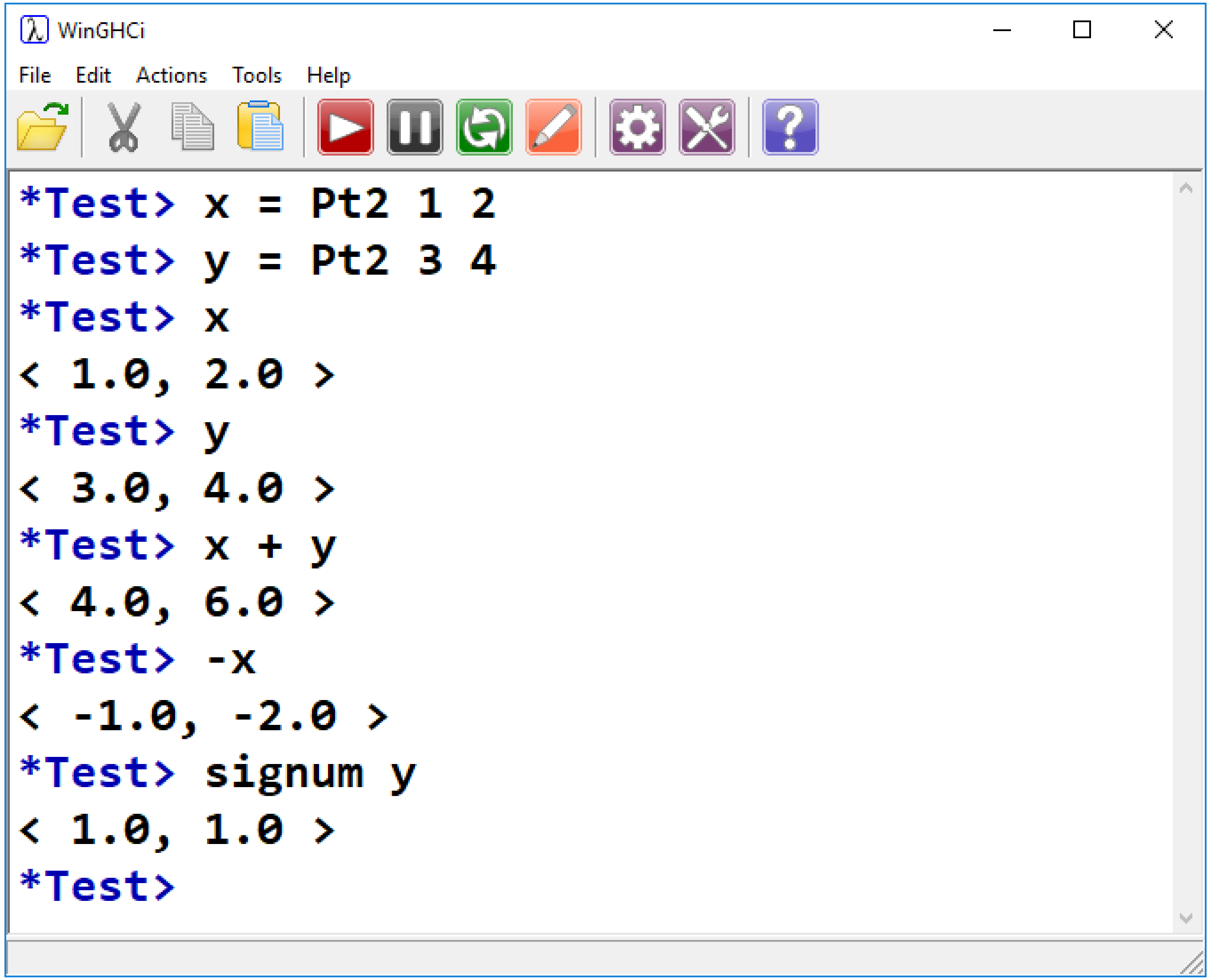
- The minimal definition for Show is easy
- Need to implement show OR showsPrec
- Let’s do show
- Need to go from Pt2 to a String


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



```
*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