C/CPS 506

Compara **Languages**

Prof. Ale

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 [illegible] ex Ufkes. Students registered in course CPSC 506 can use it 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 bout 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 xpected type, a com

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:


```
1.0                    of Num type class.
                       of Fractional type class.
```

- `'a'` is a **Char**
- `"Hello"` is a **[Char]**
- **[Char]** = **String**
- `t` is a **Bool**

`Num p => p ?`

p is a *type v* e value can be any type in type 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?

Types as generic as possible

- If we expli… a variable as integer, it can't be p… nction 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

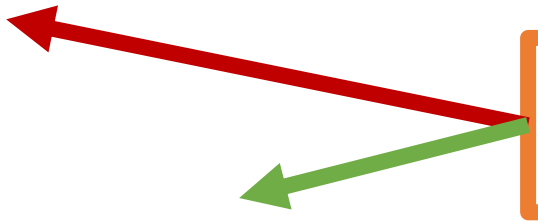
We can explicitly indicate types:


```
Prelude> :t 5::Int
5::Int :: Int
Prelude> :t 5.0::Double
5.0::Double :: Double
Prelude>
```

- Us 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:



When defining a name, can indicate type explicitly:

Type Classes

Type polymorphism and type variables:

Recall: Overloading

- In langu tor is overload ferent types.
- Numeric type equality a uality 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 pes, 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`**

- (==) ta of type **a**, where **a** is a pe 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.

Num Type Class

- This allows nu s 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.

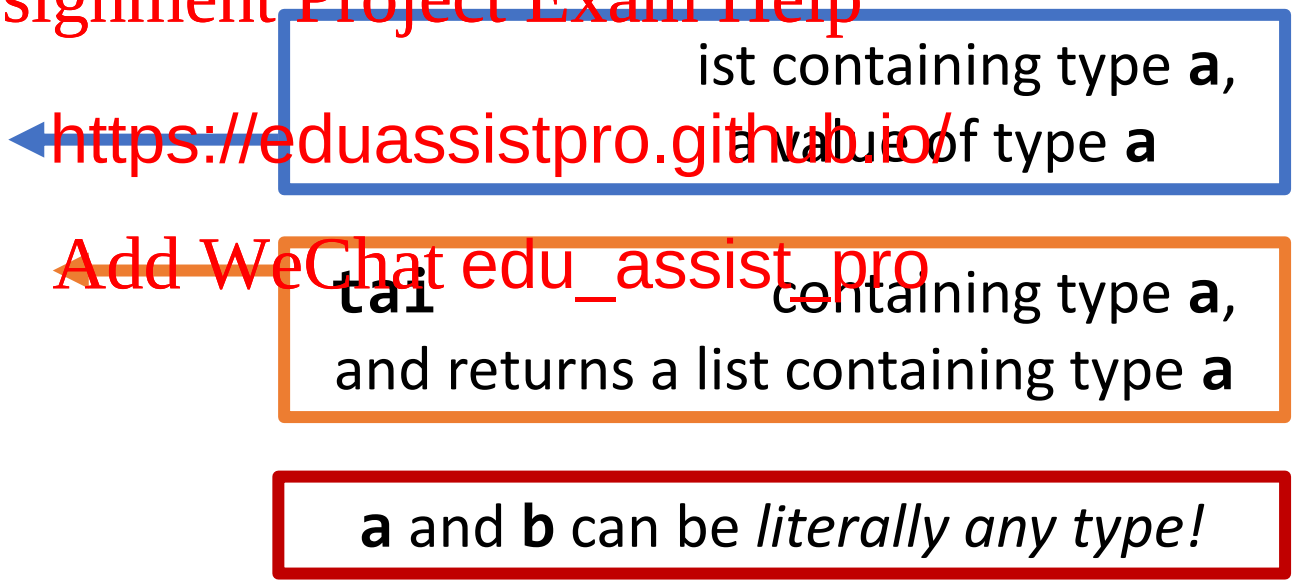
Num Type Class

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

Show Type Class

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

Function Types



Function Types

- ____ function
- tring, returns ***IO action***.
- ut, IO() is output.
- More on IO actions later.

Same as [Char]

We'll create our own types soon, and see how to add t lasses.

Functions in Haskell

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

- If we're compiling our
  de into an executable,
  e 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 x = x*x, where **x** is the input argument

Functions in Haskell

Let's start simple:

function **square**
ypical fashion

Functions in Haskell

Functions in Haskell


```
19
Prelude>
```

Passing in four args

Functions in Haskell

hat we're do **square** and
s ing 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.

Let's cr le!

- Simil s in Elixir
- We can load the module in GHCi
- Access its functions and expressions

Loading a Module

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

Loading a Module

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

Control Structures

if then else


```
        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
```

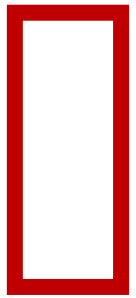

- Here we have a function named
  takes one argument x
  egative
  sitive
  - 0 if x is 0

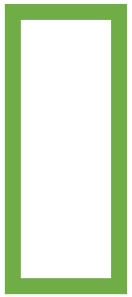
- If/else construct in Haskell is similar to most other languages.
- It must include a **then** and an **else**

Control Structures

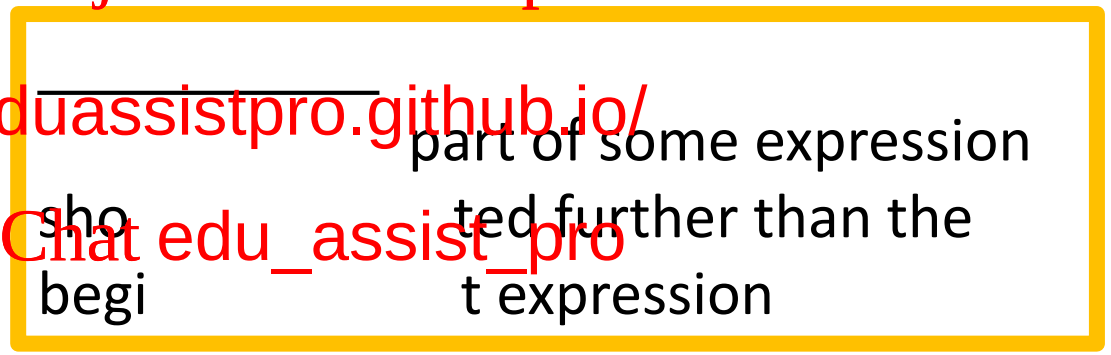
```
if then else if then else
```


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





Indenting in Haskell



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

If all that weren't enough, Tabs don't work properly u aces exactly.

Local Names in Functions

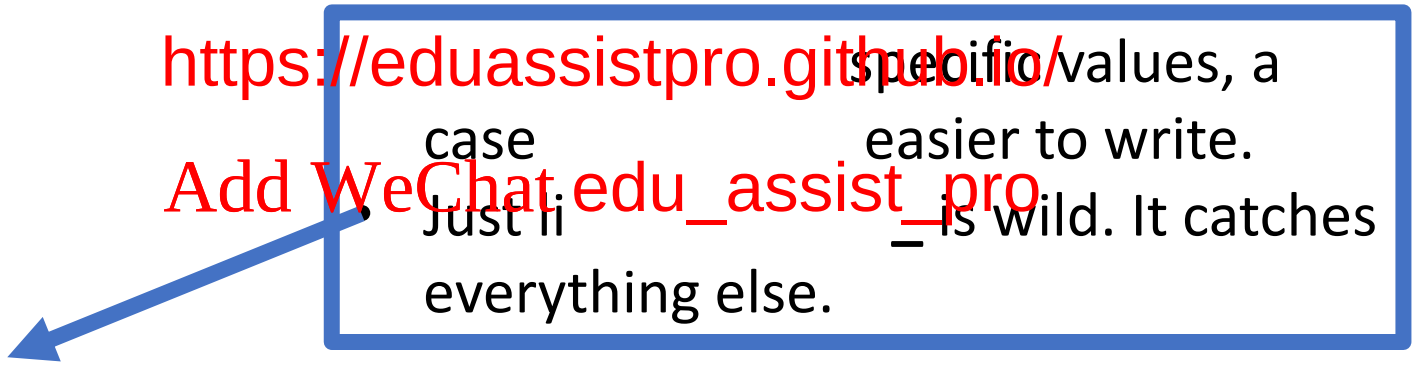
ame inside a function

or mod the "`let`" keyword

Multiple Expressions

- We now have two
  sions in this function!
  and `let`
  the `do` keyword

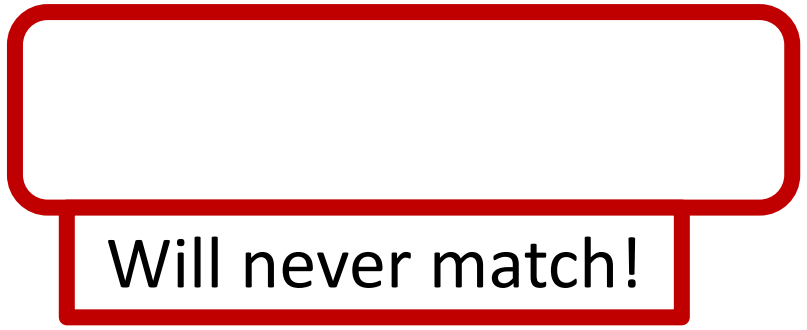
Case Statement



Case Statement

Pattern Matching: Case

Pattern Matching: Case



Unlike Elixir…

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

Piecewise Functions

Just like Elixir's function signature pattern matching

Makes recursion very easy!

Piecewise Functions

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

Functions: Guards

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

Recursion

Built-in length function

Classic list length finder

Our own user function

Tail Recursion?

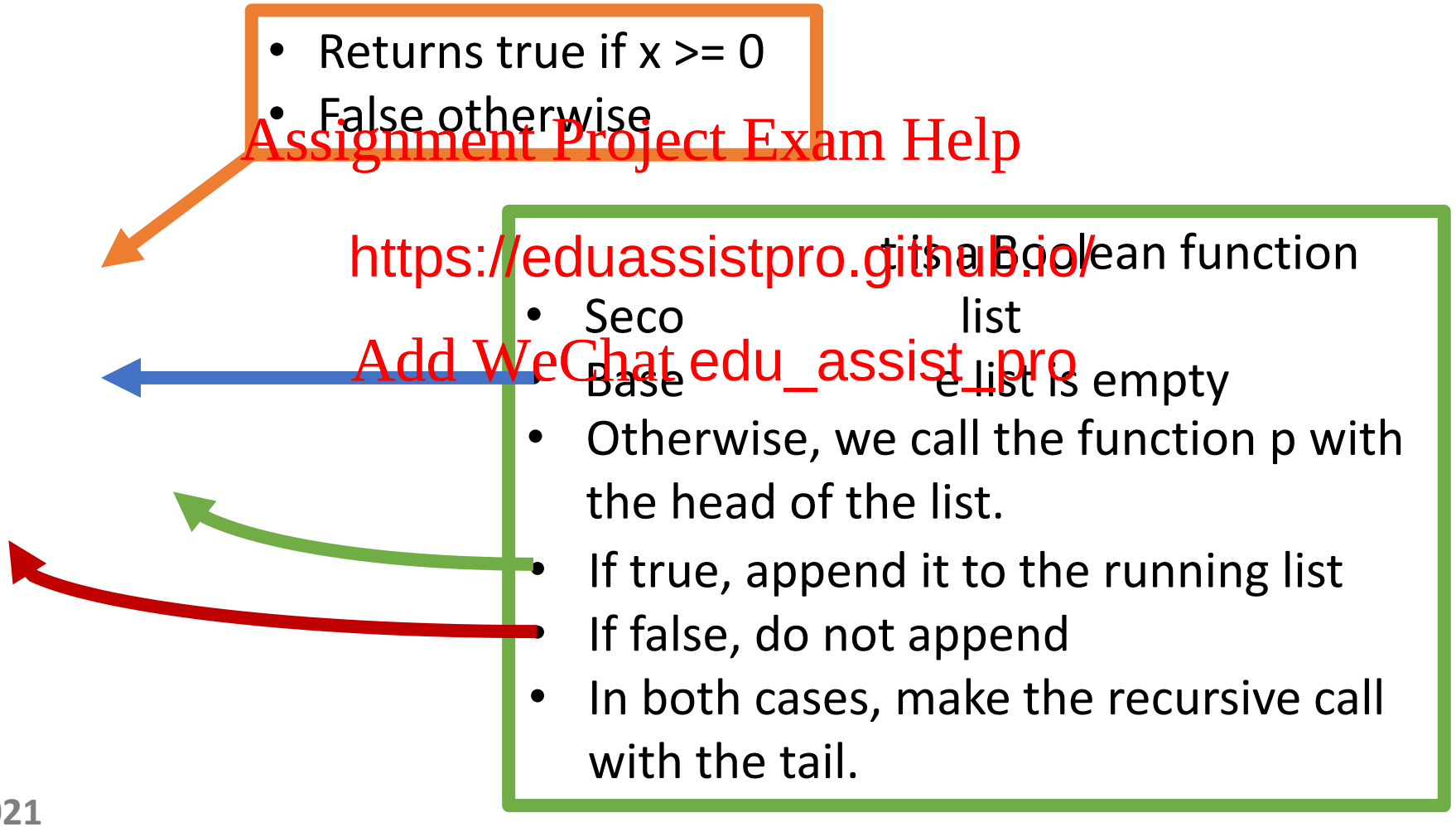
Less important in Haskell

- In Ha s different
- Funct create a new stack frame
- In practice, tail recurs g deal.

Recursion: `cons`

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

Recursion: `filter`



Recursion: `filter`

Test **pos** function

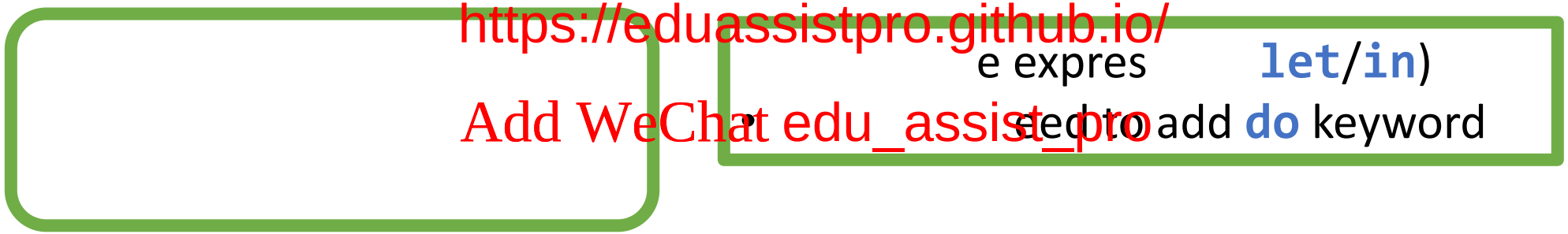
Return Multiple *Things*?

Lists/tuples to the rescue!

there is a lot of
ate computation here.

- Add a local variable?

`let/in` Structure



Infix Functions

Use symbolic operators as functions:

Enclose in parentheses to use in non-infix mode

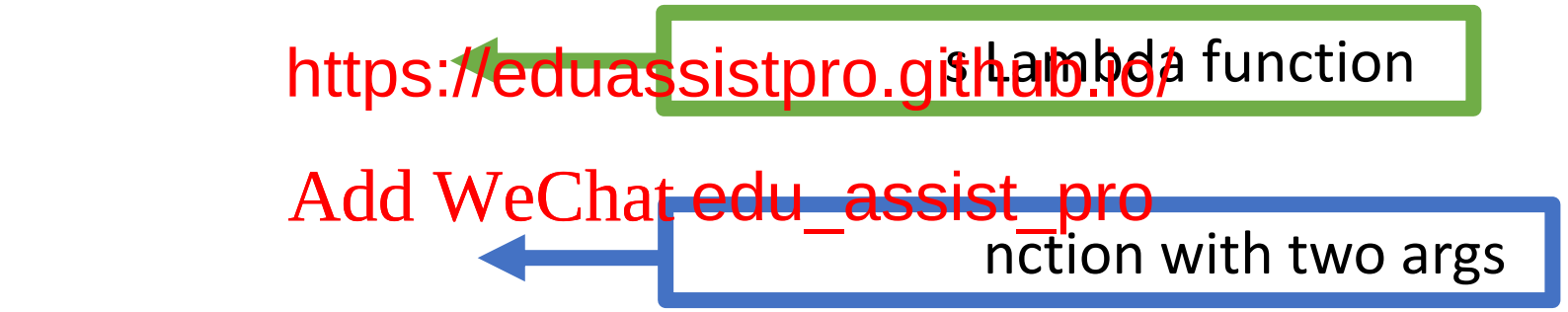
Function Composition

Can be written as:

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!

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

n of list elements

Comments

About time we mentioned them…

Type Inference

Type of `chkClr` is inferred:

- Takes as input a tuple with three values
- The input value in the tuple must be instance class **`Eq`** and **`Num`**.
- The output is a character list.

Type Inference

Based on the contents of chkClr:

- Haskell determined that the output is [Char]
- tuple whose elements must
- **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

- **`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

Thoughts?

```
*Test> :t cmp2
cmp2 :: Ord a => a -> a -> [Char]
```


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

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

- 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

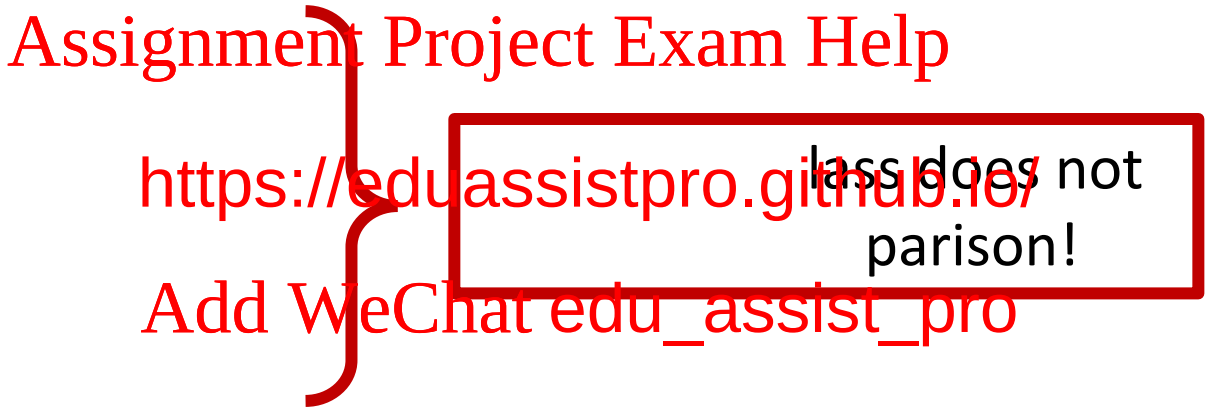
`Ord` Type Class

- 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?



Hmmmm…

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

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

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:

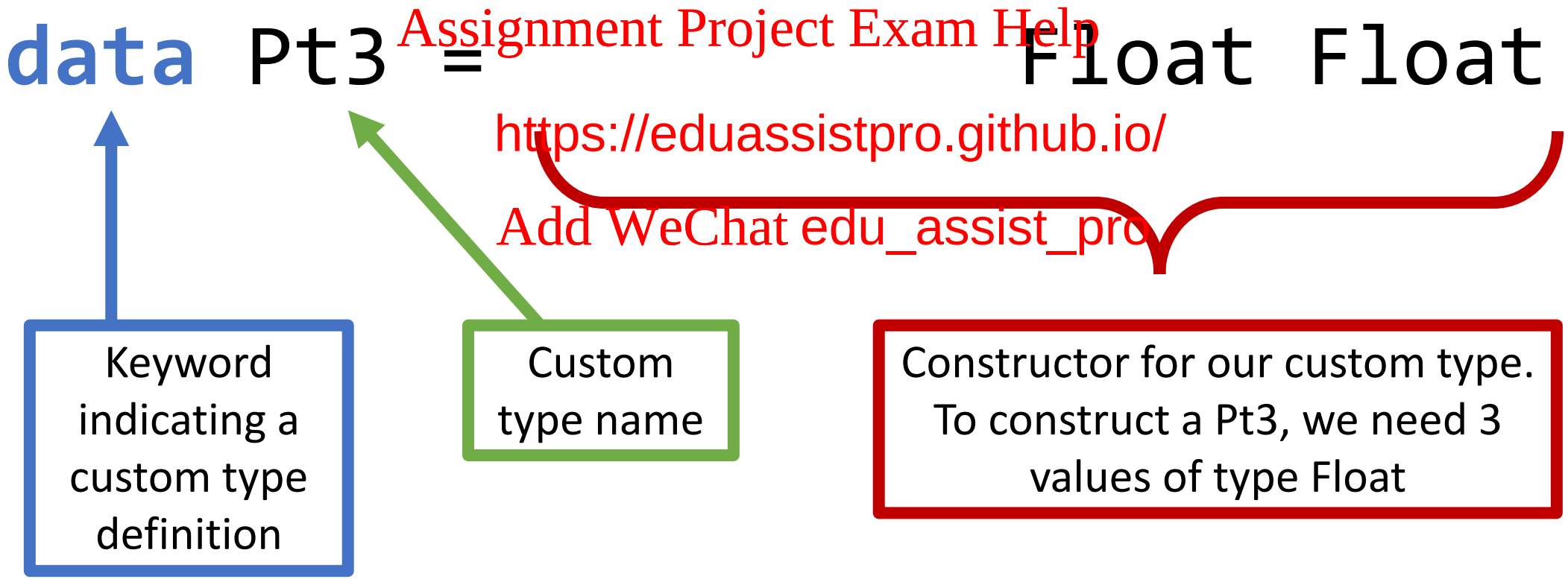
Num:

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 1, 2).
- What if we want ates and compute the sum? The dot product? Etc.?
- Addition is not defined for tuple ore complicated operations.

Custom Coordinate Types



Custom Type Usage

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?

- 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

- Define Pt3 with three parameters
  with two parameters
  r data type is Pt,
  b ave made it more generic.

There is now a problem with our access functions

There is now a problem with our access functions.

Now our access functions work for both Pt2 and Pt3

Deriving Show

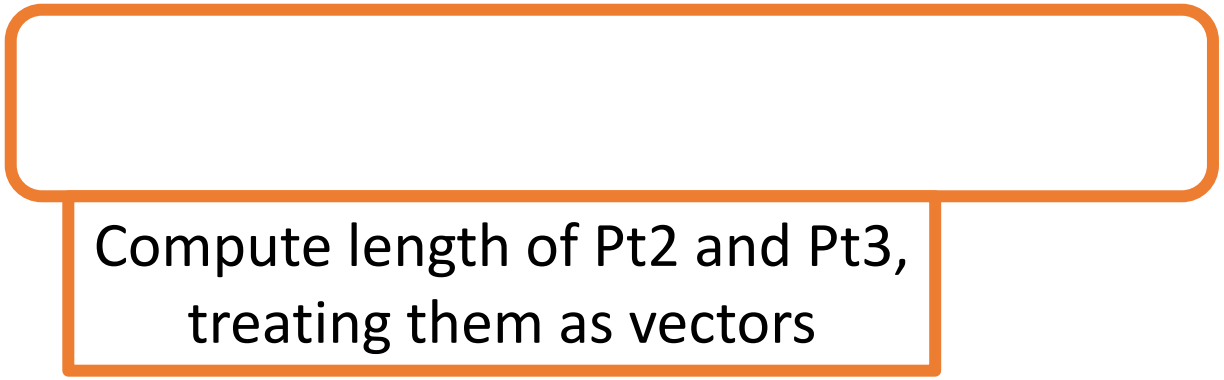
Recall:

Deriving Show

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

Similar to the toString() method in Java!

More Advanced Functions



Addition, Subtraction, Equality?

functions!

- W asily define addition as the sum of each respective X and Y coord
- Likewise for subtraction and equality.

Addition, Subtraction, Equality?

Addition, Subtraction, Equality?

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

We can! Equal s of type class **Eq**

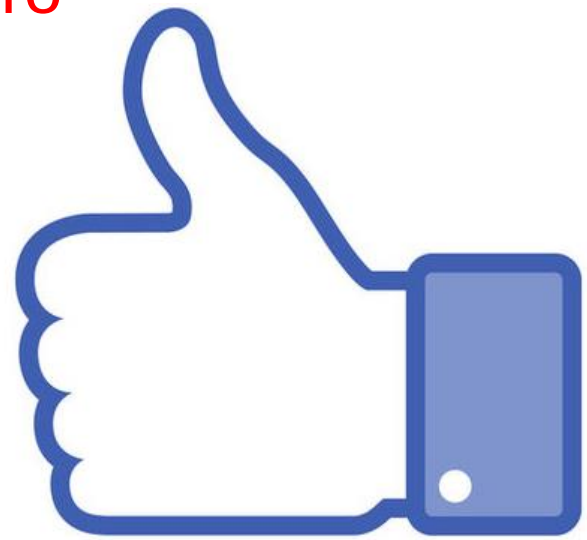
+, -, etc. are defined for ms of type class **Num**.

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

Custom Types & Type Classes

ine what it means for two Pt2
lues to be considered equal

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

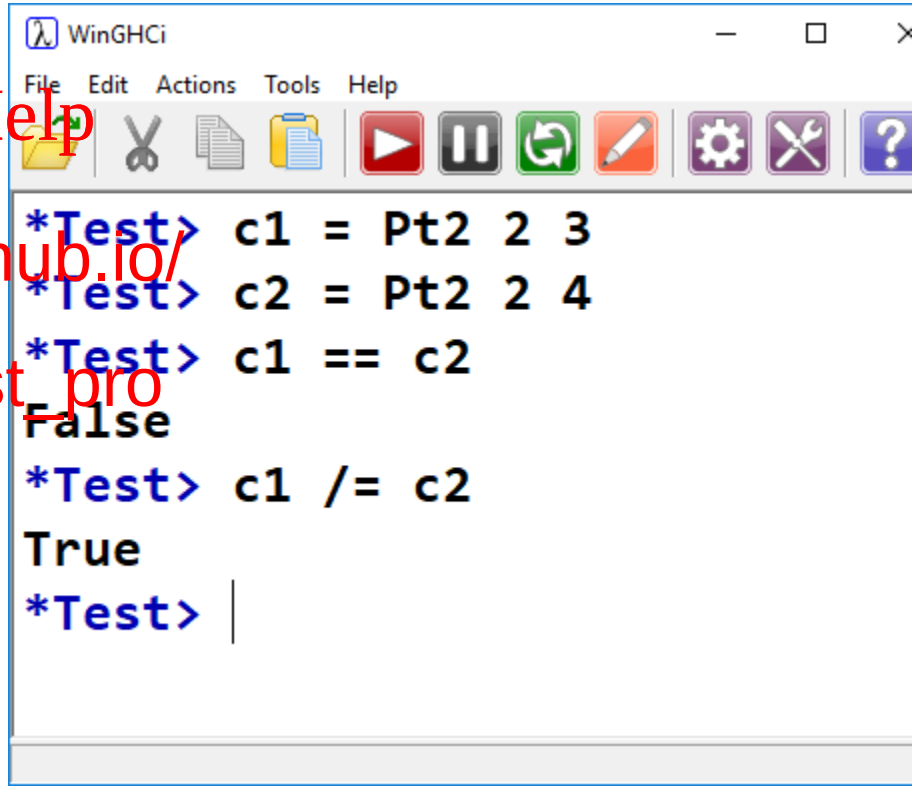


Minimal Definition

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

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

Let's Add /= Anyway



By The Way…

This should remind you of interfaces in Java

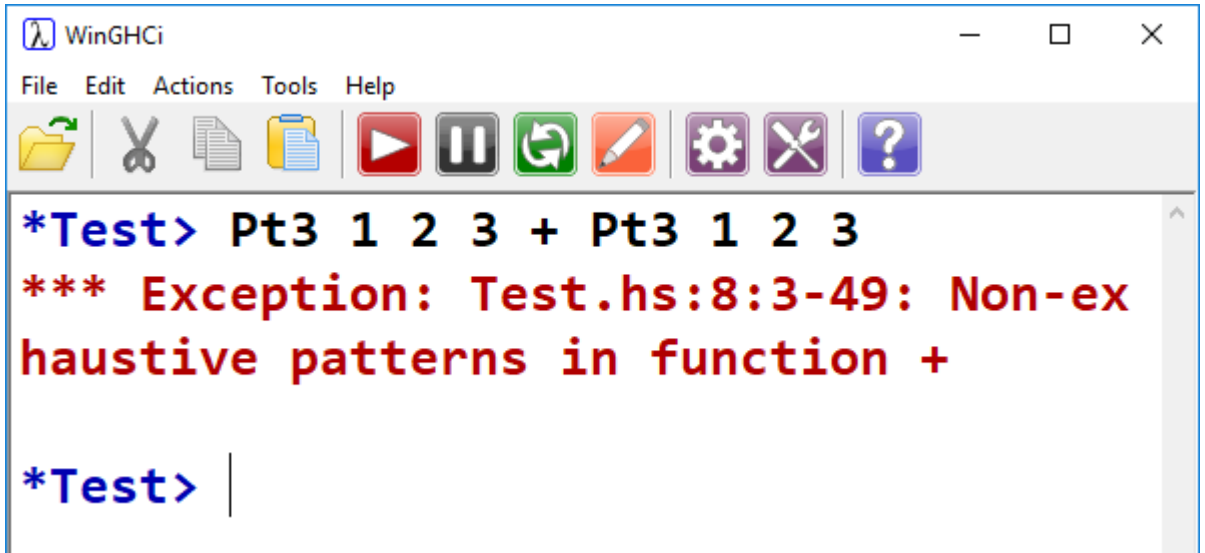
If you create a Ja n interface, it must define all me that interface

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

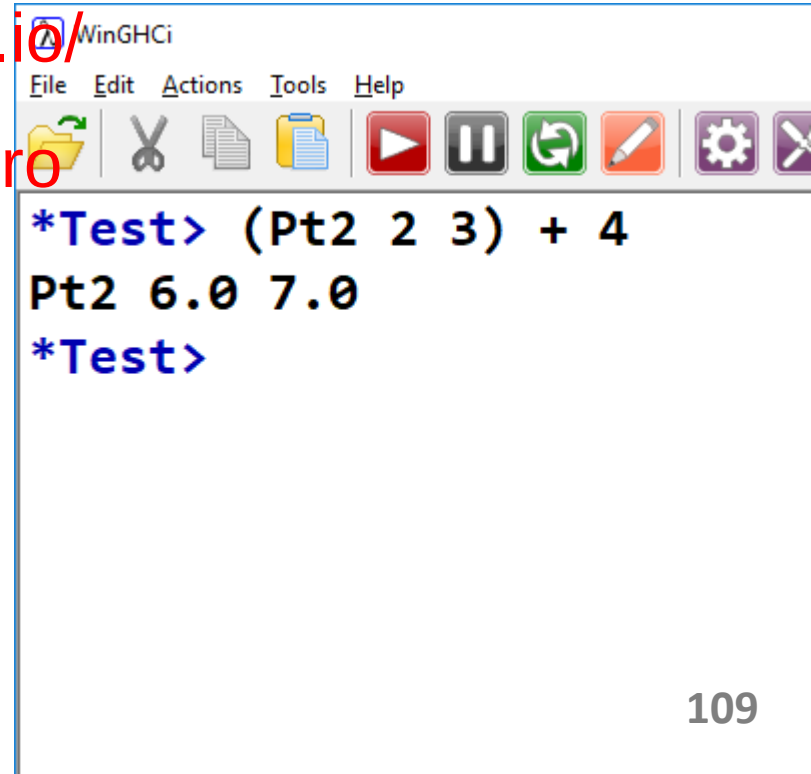
- We're only implementing for Pt2.
- Adding Pt3 follows the same pattern



Instance of Num

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

- `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…



No more warnings!

Instance of Show

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

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

No longer need to derive
, 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

Haskell Tutorials/References:

https://en.wikiboo ther_Haskell_Tutorial

http://cheatsheet.codesto heatSheet.pdf