C/CPS 506

Compara **Languages**

Prof. Ale

Topic 6: Type systems, pure functional with 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

Elixir assignment 9

Today

Type systems:

Intro to Has

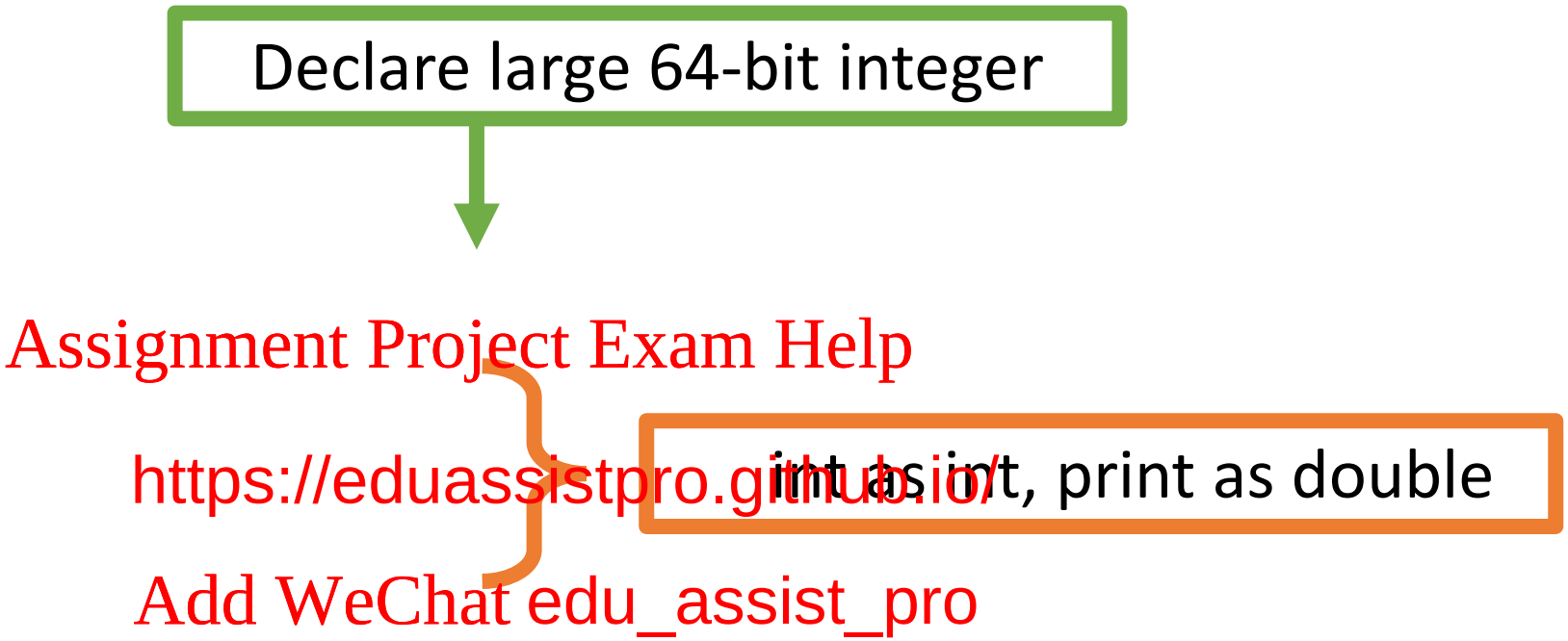
- Pure functional
- Typing in Haskell

Type System

- A set of rules that assigns a property called ***type*** to constructs of a program.
- These constructs include variables, functions, expressions, etc.

The whole point is to reduce bu

- For example, if a pattern of 3 [illegible] en encoded using 2s complement, we don't want to read it using IEEE 754
- And we *can* do this in many languages!



- The 2s comp bit pattern was read as an IEEE 754 double.
- (The integer constant was deliberately picked to produce a bit pattern that would yield 1.000000 as double)

Type Checking

Clearly, type checking isn't performed in the context of a **`printf`** statement in C++

- Think of type checking as a puzzle: pieces together.
- Does the output type of a fun the variable we're trying to store it in?
- Do the input arguments to a function match the types indicated in the parameter list?
- If no, will we allow implicit conversion?

Static VS Dynamic

When are types checked?

Statically typed languages perform type checking at *compile time*

- Checked while converting source code to machine (or byte) code

Dynamically typed l ecking at *run-time*

- Checked on the fly while instructi g executed.

Statically Typed languages: C/C++, Java, Haskell, Rust

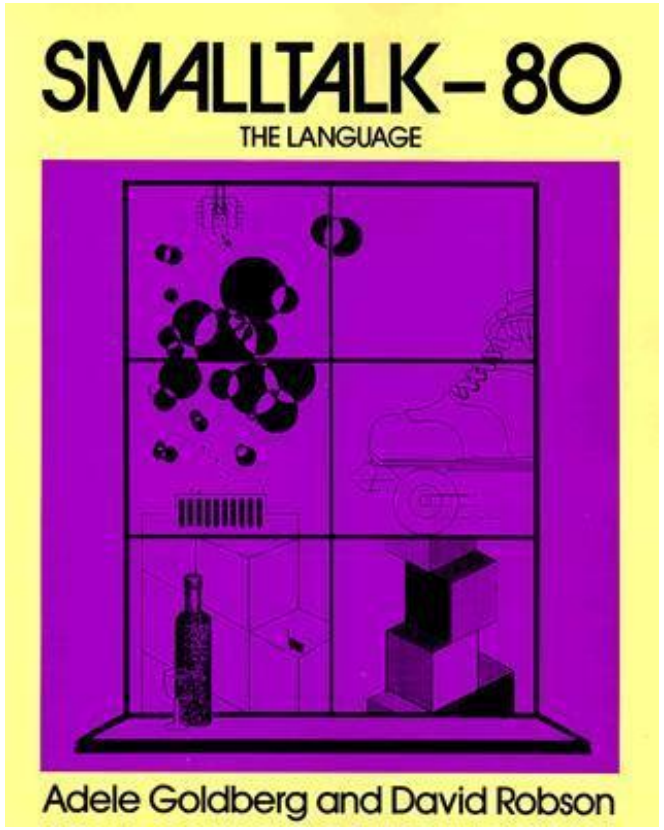
Dynamically Typed languages: Python, Smalltalk, Elixir

Static Type Checking

```
incompatible types: java.lang.String cannot be
converted to double
```

- In dynamically typed languages, every operation knows the types for which it is valid.
- Providing invalid arguments or operands will yield a run-time erro t be recoverable
- Such things can be anticipa itigated in various ways, such as verify xplicitly

Dynamic Type Checking




```
factorial: n
   | fac |
   fac := 1.
   n
      ifTrue: [1
         [:a | fac          .].
         ^fac
      ]
      ifFalse: [
         ^'Bad input'
      ].
```

- In Java, the parameter would be defined as **`int`**
- Compile error if arg isn't **`int`**, or
  't be implicitly cast as an **`int`**.

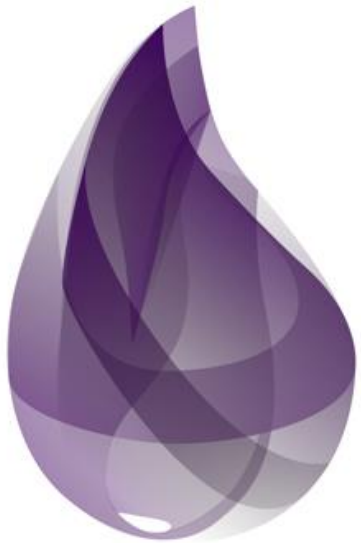
- Of course, polymorphism in Java complicates this.
- Still statically typed.

Dynamic Type Checking…?

```
#(1 2 3 4) + 18.2
```


- Does Smalltalk have type errors in the strict sense?
- Different obj messages.
- A “type error oesn’t have a method to handle a partial
- “Type” errors in Smalltalk are as a result of not finding a method (DNU, Did Not Understand).
- Above, the error occurs because the Array class doesn’t have an instance ***method*** for symbolic operator **#+**
- Smalltalk enthusiasts debate this.

Dynamic Type Checking



```
defmodule UserMath do
  def fib(n) when not is_integer(n) or n < 0 do
    :error
  end
  de
  de
  def fib(n), do: fib(n-1)
  def fac(n) when not is_integer(n) or n < 0 do
    :error
  end
  def fac(0), do: 1
  def fac(n), do: n*fac(n-1)

end
```

Static VS Dynamic

Advantages? Disadvantages?

Static:

- Reliably find errors at c
- Code will execute faste
  assumed to be correct at run time.
- Type-specific optimization can be performed at compile time.
- I.e., integer arithmetic is faster than floating point

Dynamic:

- pilers run faster
- n dynamically load
  - de
  - Smalltalk, MATLAB, iex
- Easier code reuse

Static VS Dynamic

Advantages? Disadvantages?

- There is much disagreement among programmers about just how much of a proble and scheme of things.
- Does the added c ically typed language make sense if type-related bugs a fraction?
- Of the type-related bugs that occ portion of those would have been solved by a type checker anyway?
- They aren't perfect after all.

Strong VS Weak Typing

Strong VS Weak (or *Loose*)

Refers to how strict statically typed languages are at compile time

There is actually no universally accepted definition of what
co typing

Of strongly typed

1974: “Whenever an object is pa calling function to a called function, its type must be ***compatible*** with the type declared in the called function."

Compatible is open to interpretation. Is float compatible with double? Integer with short integer?

Strong VS Weak (or *Loose*)

Refers to how strict statically typed languages are at compile time

1974: "Whenever an object is passed from a calling function to a called function, its type must be compatible with the type ction."

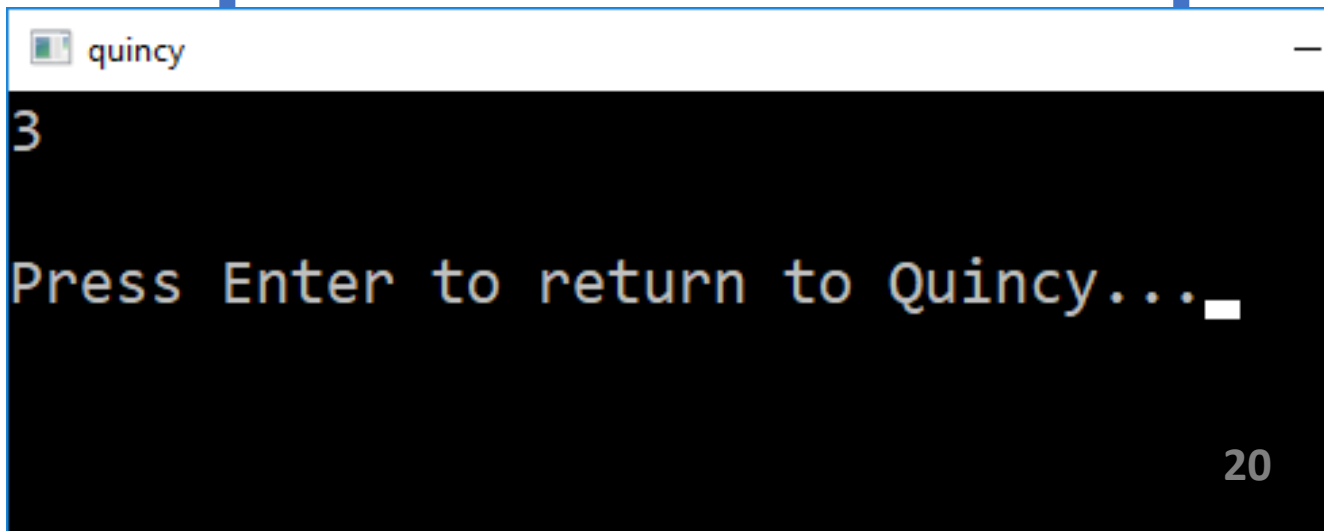
1977: "In a strongly typed lan ata area will have a distinct type and each process will state its ***communication requirements*** in terms of these types."

Parameter lists, return types, etc.

THE C PROGRAMMING LANGUAGE

Strong VS Weak (or *Loose*)

To what degree does a statically typed language allow implicit type conversion?



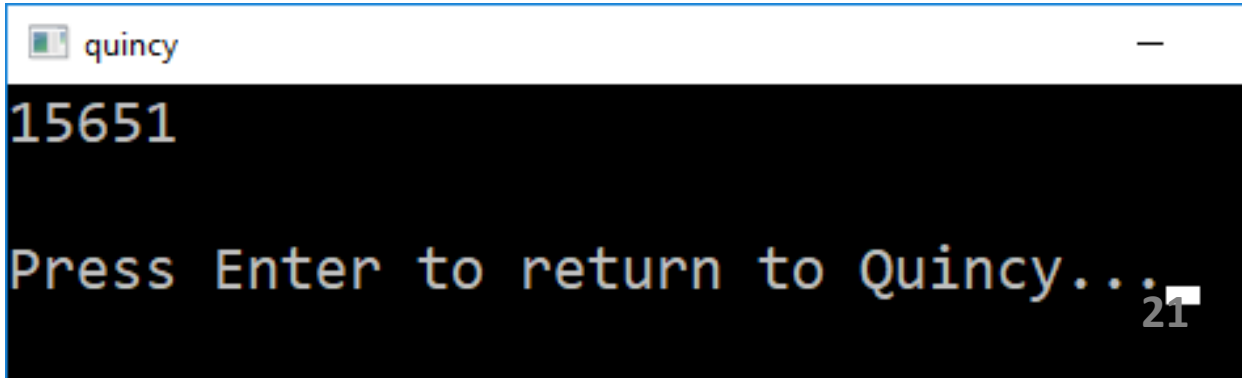
- C is weakly typed.
- Happy to perform all manner of implicit conversion without warning or error.

THE C PROGRAMMING LANGUAGE

Strong VS Weak (or *Loose*)

In C, pointer arithmetic can be used to ***completely bypass*** the type system:

- We're using pointer arithmetic to read the first 2 bytes of an **`int`** as a **`short`**.
- ne with any two types.
- he rightmost 4 by
- det **`int`** etc.
- We can treat memory any way we want

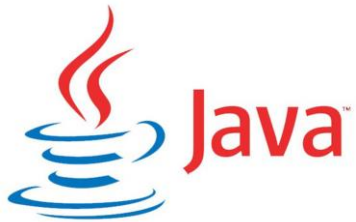


Strong VS Weak (or *Loose*)

C++ will give warnings where C did not, but still compiles and runs in this case:


```
guration: Debug Win32 ------

 warning C4244: 'initializing' : conversion from 'double' to 'int', possible loss of data
est\Debug\test.exe
-to-date, 0 skipped ==========
```



Strong VS Weak (or *Loose*)

Java will throw compile errors when a ***loss of precision*** occurs:

- No implicit rom floating point to integer
- Floating point constants are double precision
- Need to indicate single precision explicitly

Java will throw compile errors when a loss of precision occurs:

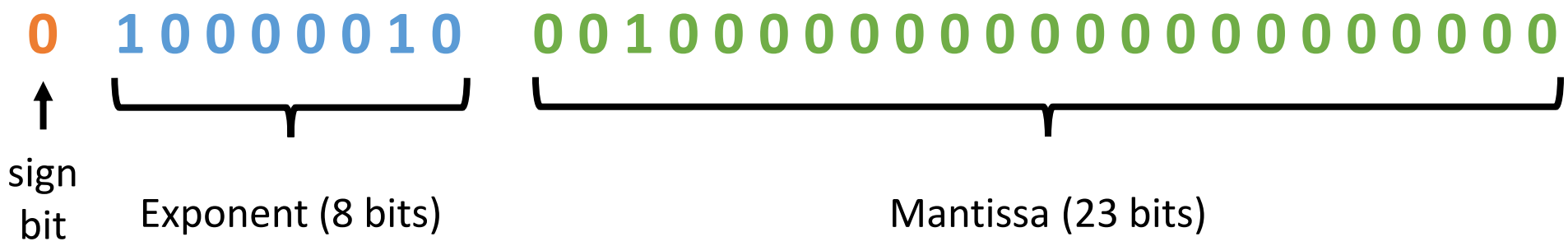
Careful! Loss of precision does not ***only*** occur when going from floating point type to integer type!

`int` is 32 bits two'

`float` is a 23-bit

here!

32-bit:



Imprecision of Floating Point

- Integers are represented *precisely.* The integer 42 is **exactly** 42.
- The single-precision (32 bits) floating point value 0.1 is ***actually*** 0.100000001490116119384765625
- *Double*-precision ues are more accurate, but still not perfe

But why?

- Floating point values exist on an infinite continuum.
- Between any two floating point values are an ***infinite*** number of additional floating point values.
- Integers are discrete. Between any two integers are a ***finite*** number of integers.

Imprecision of Floating Point

- A double-precision float is represented using 64 bits.
- A ***finite*** number of bits cannot represent an ***infinite*** number of floating point values.

0100101110000010100011011000110000111000011

- There are **2^64** ways to arrange 64 bits. A large number to be sure, but certainly not infinite.

Infinite Integers?

But there are an infinite number of integers!

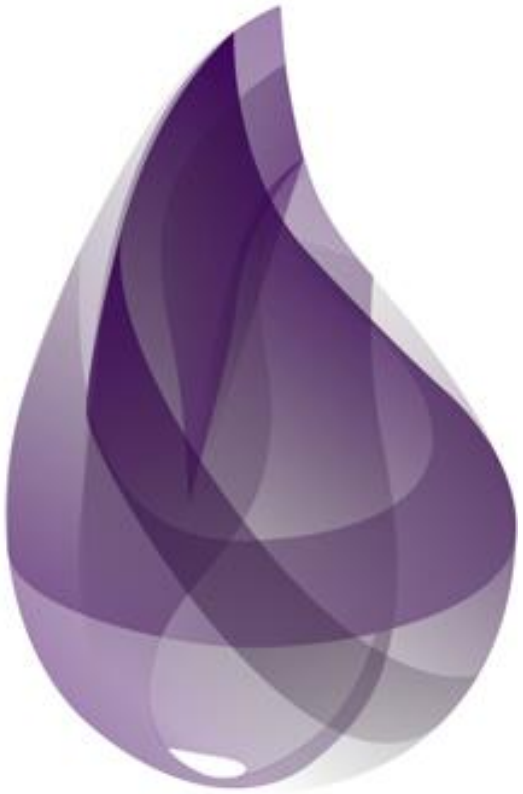
- 100% correct. We can't represent every possible integer either.
- Rather, there is integer has a range of −2,147,483,64
- Every integer within this range is precisely.
- Anything outside this range can't be represented using 32 bits
- If we try, we overflow.

IEEE-754

Functional Programming

Functional Programming

Functional Programming



Higher-order functions:

- Can return functions or accept them as arguments.

F

- ents, returned as values.
- Think of them st like integers or floats

Pure Functions:

- Functions that have no side effects. No interaction with world outside of local scope
- Easier to verify correctness, thread-safe when no data dependency is present.

Pure function

Impure function

Functional Programming

Strict (eager) VS. non-strict (lazy) evaluation:

- Strict: evaluate function arguments before invoking the function.
- Lazy: Evaluates arguments if their value is required to invoke the function.

Elixir largely performs strict evaluation
(some exceptions, recall `Stream`, `Range`)

Functional Programming

Strict (eager) VS. non-strict (lazy) evaluation:

- Strict: evaluate function arguments before invoking the function.
- Lazy: Evaluates arguments if their value is required to invoke the function.

https://www.haskell.org/

A great intro to Haskell syntax

Haskell: Functional Programming cranked up to 11

History

- Named after logician Haskell Curry
  interest in lazy functional
  lowing
- There
  g consensus to define
  an op
  for such languages

History

- Haskell 1.0 was defined in 1990
  - Continued with version 1.1, 1.2, 1.3, etc.
  - Culminated with *Haskell 98* published in July 2010
- Next version controversial features pre led via compiler flags
  - Though progress seems to have stalled worked on – Haskell 2020
  - Perhaps it should be called Haskell 202X

Features

Purely Functional:

- Every function is *pure*
- Even side-effect inducing operations are
  e code
  only expressions
- Canno
  riables (local or global)
- Suppo
  atching
- Side effects are handled using *monads*

Features

Statically Typed:

- Every expression has a type
  - Determined at compile time
- g an expressions must match
- ile error

Type Inf

- Types don't have to be written out explicitly
  - Though you can if you want
- They will be inferred at compile time

Features

Lazy Evaluation:

- Functions don't evaluate their arguments
- Control constructs written as functions

ins of functions together

ver takes place unless a result

Concurrency:

- GHC (Haskell compiler) includes high performance parallel garbage collector
- Light-weight concurrency library

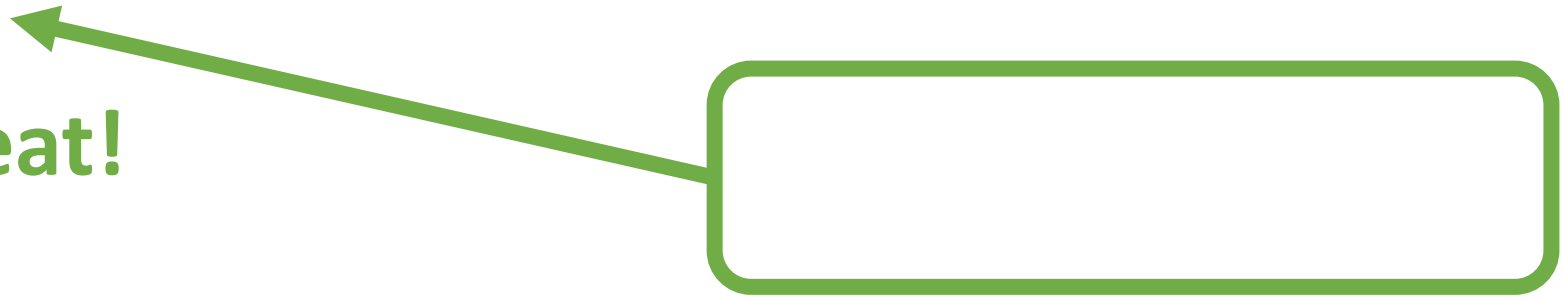
Haskell in Industry?

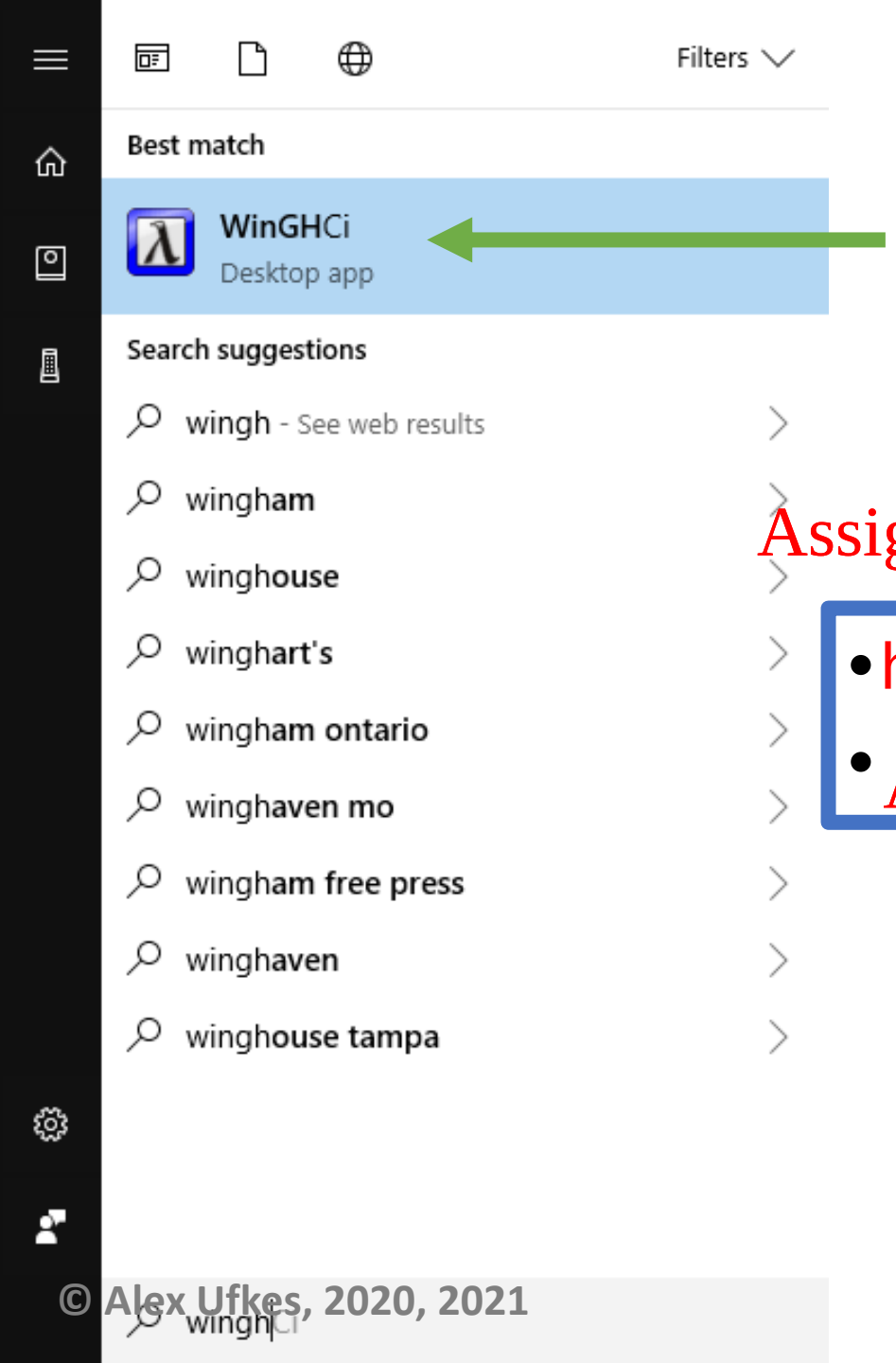
Notable companies that use or have used Haskell:

- Nvidia
- AT&T
- Ericsson
- Facebook
- Google
- Intel
- Microsoft

Typically Haskell is used on specialized internal projects or research. Not necessarily company-wide.

Installing Haskell:





- https://eduassistpro.github.io/ just like Elixir.
- Haskell's i

Hello, World!


```
Prelude> main
Hello, World!
Prelude>
```

putStrLn == System.out.println()
putStr == System.out.print()

efine a main function.
hen executing a Haskell program, main is the entry point

- Just like C or Java

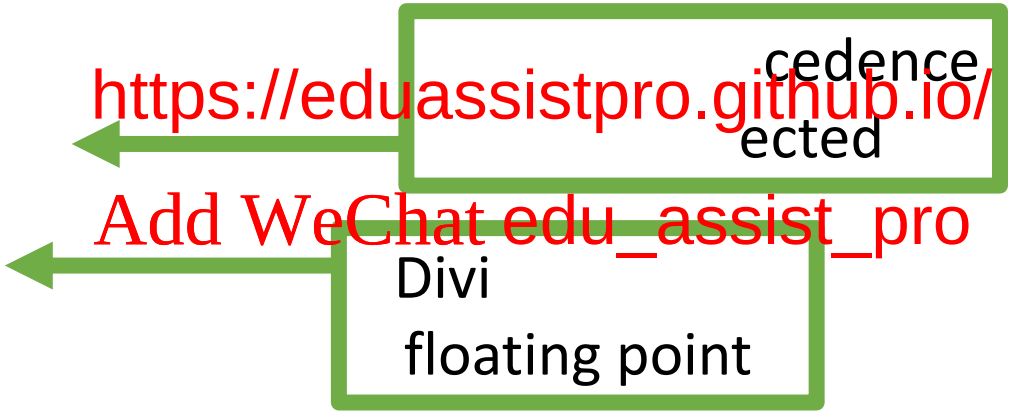
Execute main function

Compiling Haskell

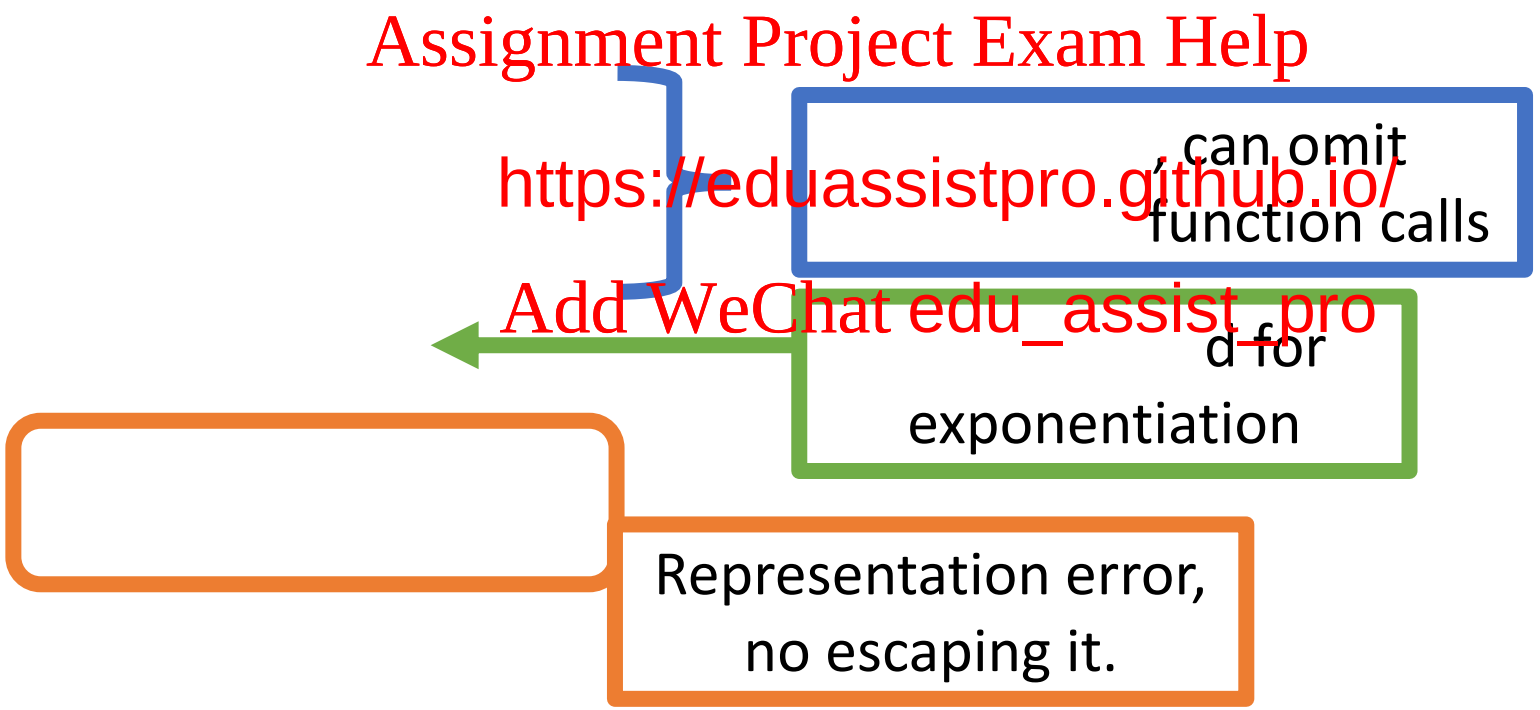
Notepad++ features Haskell syntax highlighting!

As with E nd in the interactive shell until it edious.

Literals & Arithmetic



Literals & Arithmetic



Tuples

- orts tuples.
- he same types.

There a nctions for accessing first and second elements. Great for coordinates.

`fst` and **`snd`** only work on pair tuples!

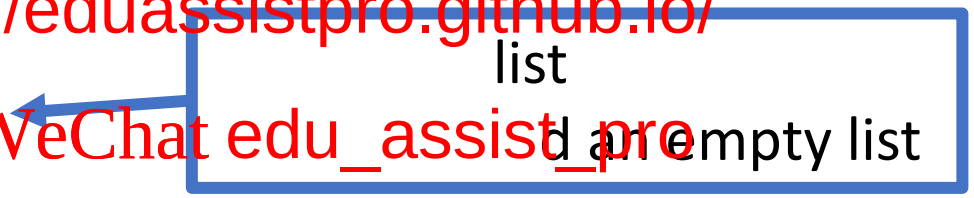
Lists

Must be ***homogeneous***:

r literals get
floating point
do not

Lists

Elements can be added to the *beginning* of a list with the **cons** (:) operator



In fact, when we write **[1, 2, 3]** the compiler is *actually* doing **1:2:3:[]**
[1, 2, 3] notation is *syntactic sugar.*

Lists & Tuples

Tuples can be heterogeneous, lists must be homogeneous.

However! We can have lists of tuples, each tuple is heterogeneous.

Lists & Tuples

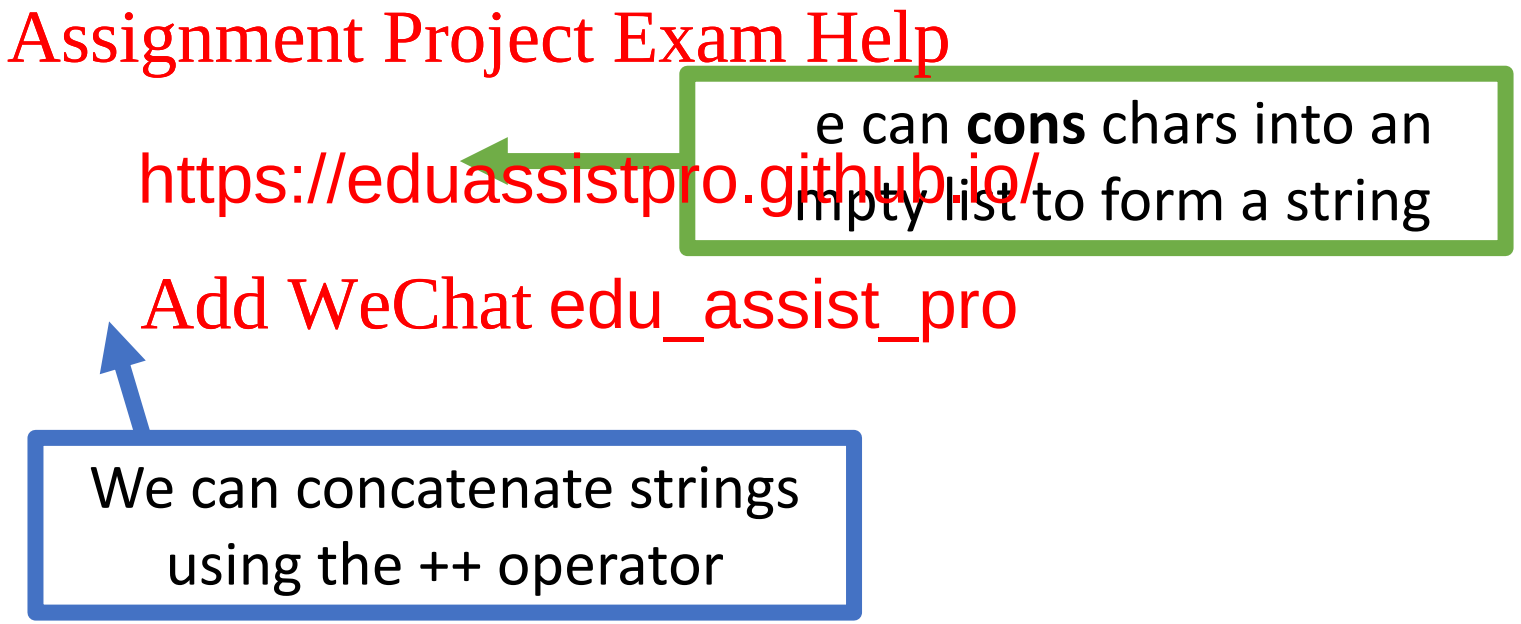
Tuples can be heterogeneous, lists must be homogeneous.

However #2!

of tuples, each tuple
ve the same format:

Strings

Strings are simply lists of chars:



Strings

Concatenate multiple types? Java lets us…

Strings

`show()` and **`read()`** functions

Convert non-string argument to string

Read numeric a string (like *sscanf* in C)

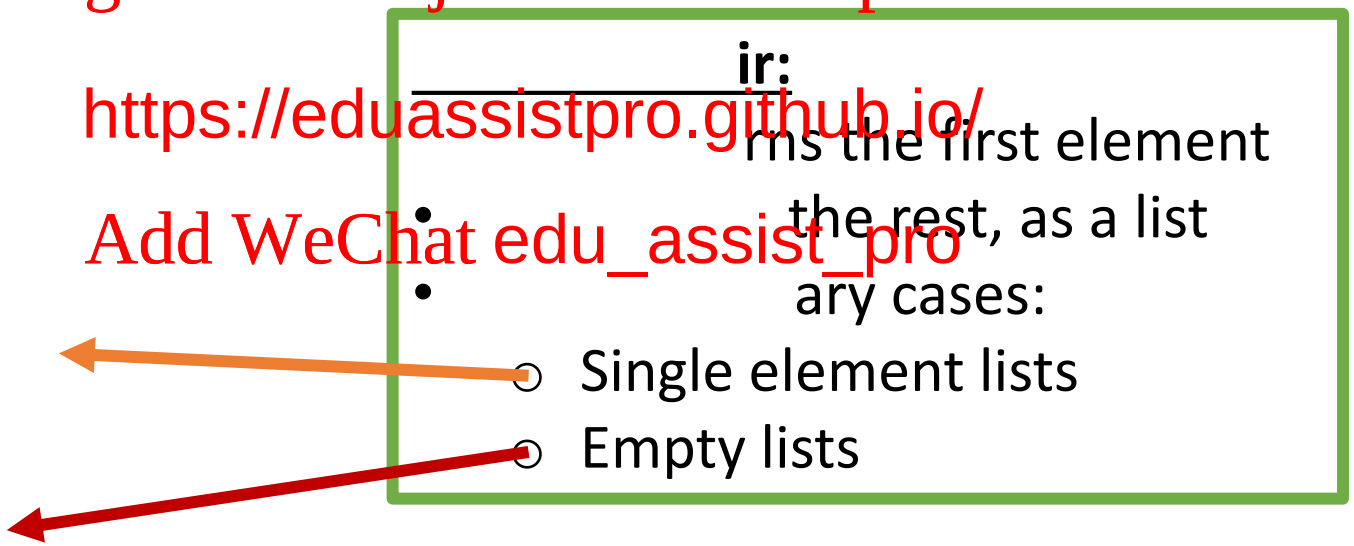
Error when no numeric value is present

Operations on Lists

- In functional programming, computation is done in large part by operating on lists.
- We sa in Elixir.
- Haske ations.

Three primary list-processing functions: **`map`**, **`filter`**, **`foldr`** (and **`foldl`**)

Head & Tail



`map`

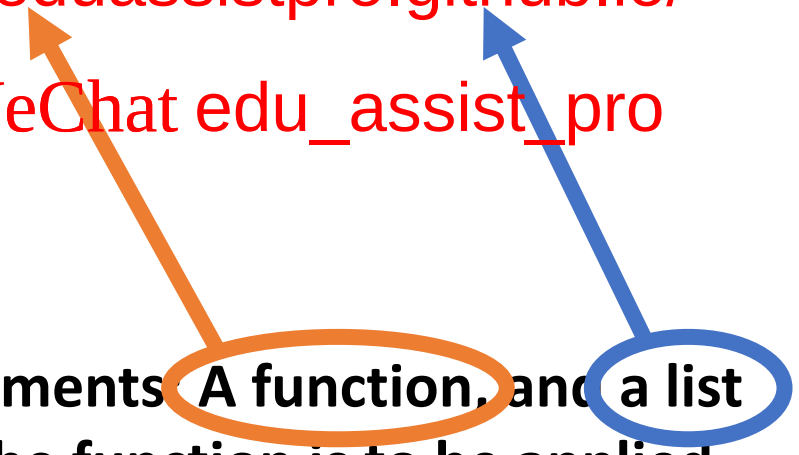
Similar to Elixir's **`Enum.map`**

- First class function, doe suggests.
- Alphabetical characters are upped, everything else is left the same.

Recall: map operates on lists, but a string is just a list of characters

`map`

Similar to Elixir's **`Enum.map`**



Map takes two arguments: A function, and a list of values to which the function is to be applied.

`filter`

"Remove" items from a list based on some criteria:

Function **List**

`foldl, foldr`

Replaces the cons operator with some other function. This takes some explaining.

Is actually s

```
1:2:3:4:5:[]
```

By the compiler.

`foldl, foldr`

Replaces the cons operator with some other function. This takes some explaining.

Recall that the list:

```
[1, 2, 3, 4, 5]
```

Is actually seen as:

```
1:2:3:4:5:[]
```

By the compiler.

`foldr` replaces the cons operator with a function of our choosing.

- This is `lum.reduce` in Elixir.
- The empty list is replaced with some initial value.

`foldl, foldr`

Replaces the cons operator with some other function. This takes some explaining.

- **`foldr`** in effect replaces the cons operator with another .
- This is s Elixir.
- The empty list is replac e initial value.


```
foldr (+) 0 [1, 2, 3, 4, 5]
```

Three arguments: **function**, **initial value**, **list**

foldl, foldr

Replaces the cons operator with some other function. This takes some explaining.


```
foldr (+) 0 [1, 2, 3, 4, 5]
```


```
foldr (+) 0          :5:[]
```


```
1 + 2 + 3 + 4 + 5 + 0
```

```
15
```

foldl, foldr


```
120
Prelude>
```

foldr to perform factorial!

`foldl` VS `foldr`

`foldr` is *right associative*. Meaning:

```
foldr (+) 0 [1, 2, 3, 4, 5]
```



```
1 + 2 + 3 +         + 0
```


Is actually:

```
(1 + (2 + (3 + (4 + (5 + 0)))))
```

Doesn't matter for addition, but subtraction…

`foldl` VS `foldr`

`foldr` is *right associative*. Meaning:

```
foldr (-) 1 [4, 8, 5]
```



```
  4 - 8 -
```


Is actually:

```
(4 - (8 - (5 - 1 )))
```

```
0
```

`foldl` VS `foldr`

`foldl` is *left associative*. Meaning:

```
foldl (-) 1 [4, 8, 5]
```



```
  1 - 4 -
```


Is actually:

```
(((1 - 4) - 8) - 5)
```

```
-16
```

`foldl` VS `foldr`

List Generation

Syntactic sugar:

List declaration:
```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Can be written:

Specify interval:
```
list = [1,
       = [1,3,5,7,9]
```


Interval is discerned from difference between first two elements

Infinite Lists?

Try it, but be ready to interrupt

Infinite Lists?

Haskell is lazy!

ind x to the expression
nerate an infinite list.
n't have to *evaluate*
to do so!

- Displaying the list, however, requires evaluation.

Why? How?

Infinite Lists?

- Finding the length of the list requires counting the elements.
- Be ready to interrupt.

- Grab first three elements of list
- Doesn't matter if infinite, we're only evaluating first three items

Infinite Lists?

We're allowed to perform operations on a *finite subset* of an infinite list.

- **`zip`** "zips" two lists together into tuples.
- If one list is finite, the other can be infinite.

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:


```
+ :: Num a => a -> a -> a
1.0 :: Fractional a => a
```

- ‘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 nteger, 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

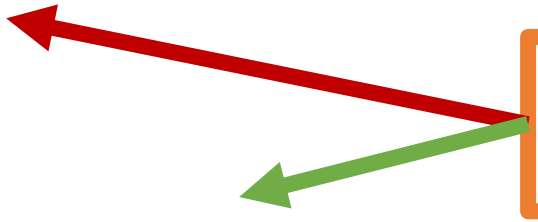
We can explicitly indicate types:


```
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.

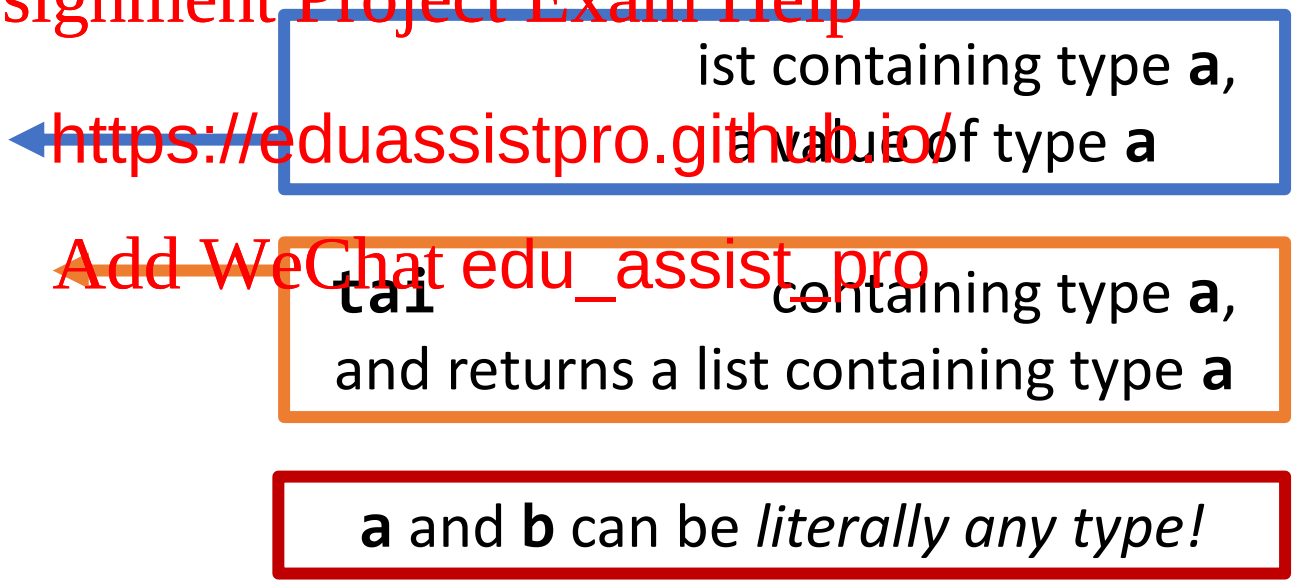
Num Type Class

- **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

- _____ tring, returns ***IO action***.
- function
- 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:

Use **`:reload`** for previously loaded module

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
```

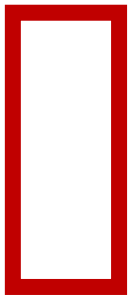

- Here we have a function named
  takes one argument x
  egative
  sitive
  - 0 if x is 0

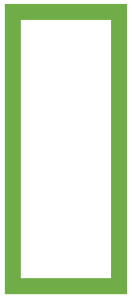
- If/else construct in Haskell is similar to most other languages.
- It must include a **then** and an **else**

Control Structures

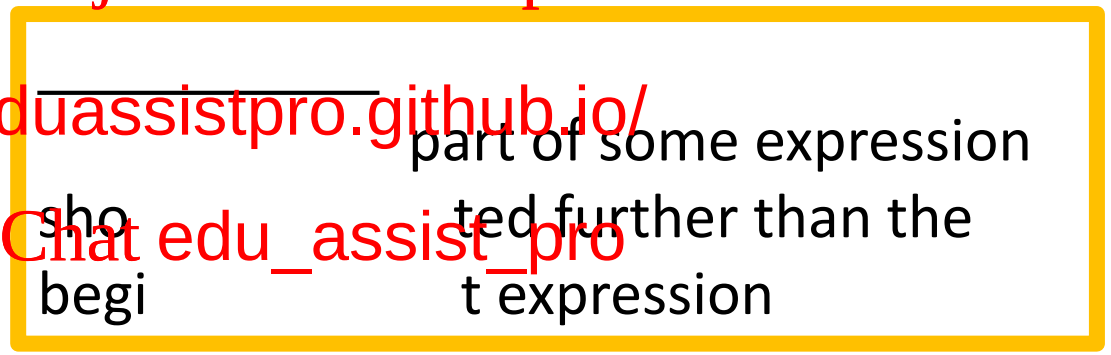
```
if then else if then else
```


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.

Haskell Tutorials/References:

https://en.wikiboo ther_Haskell_Tutorial

http://cheatsheet.codesto heatSheet.pdf