C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

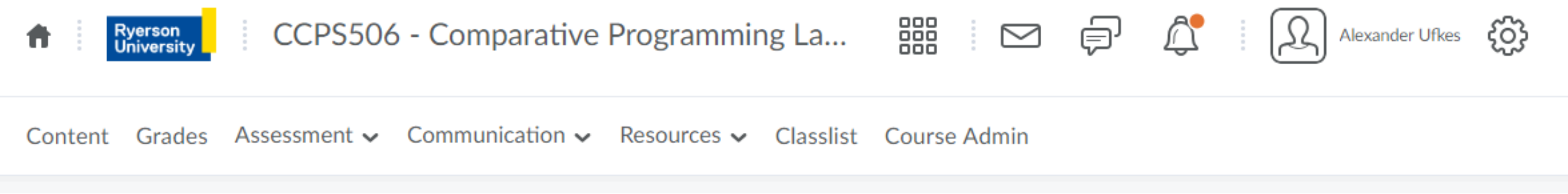
Topic 6: Haskell intro, Haskell basics

Toronto Metropolitan University

Notice!

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

Course Administration



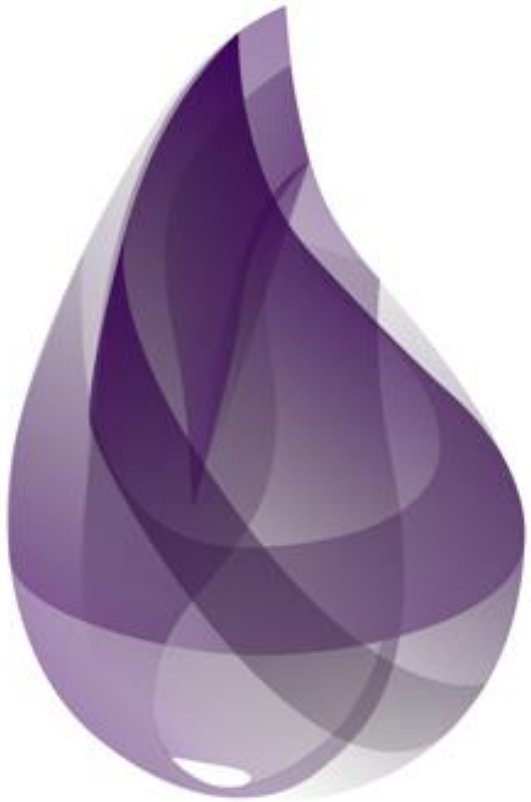
Two languages down, two to go!

Today

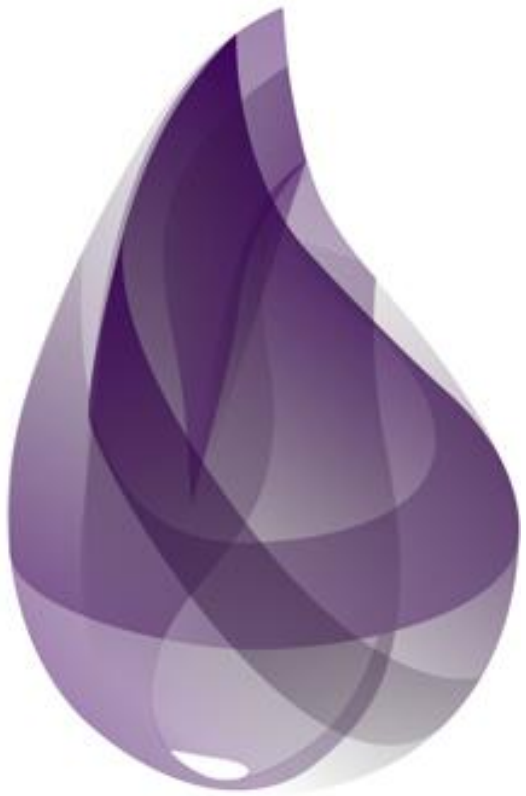
Intro to Haskell

- Pure functional
- Haskell basics
- Functions
- Control flow

Functional Programming



Functional Programming



Higher-order functions:

- Can return functions or accept them as arguments.

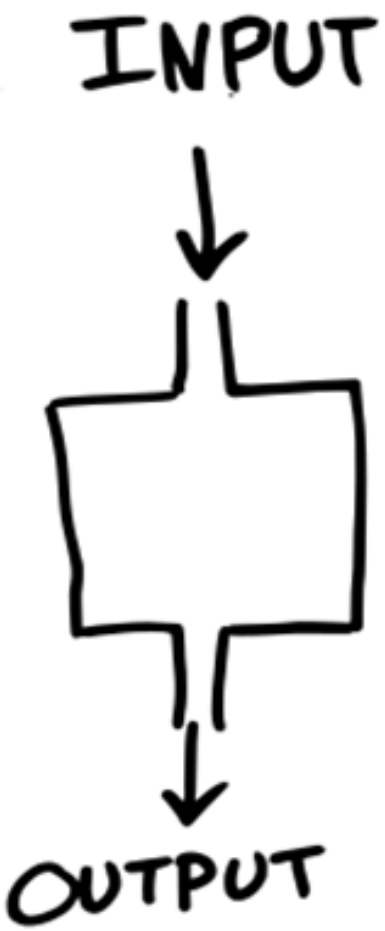
First class functions:

- Can be passed as arguments, returned as values.
- Think of them as ***values***, just like integers or floats

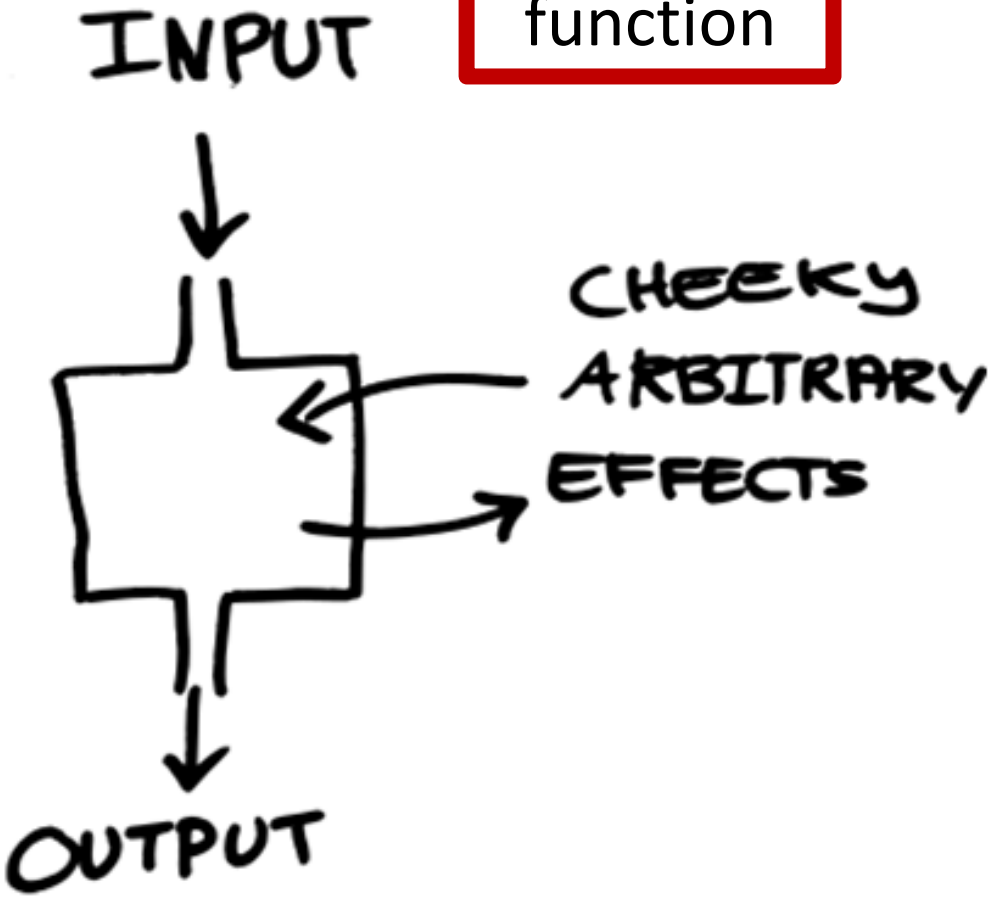
Pure Functions:

- 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

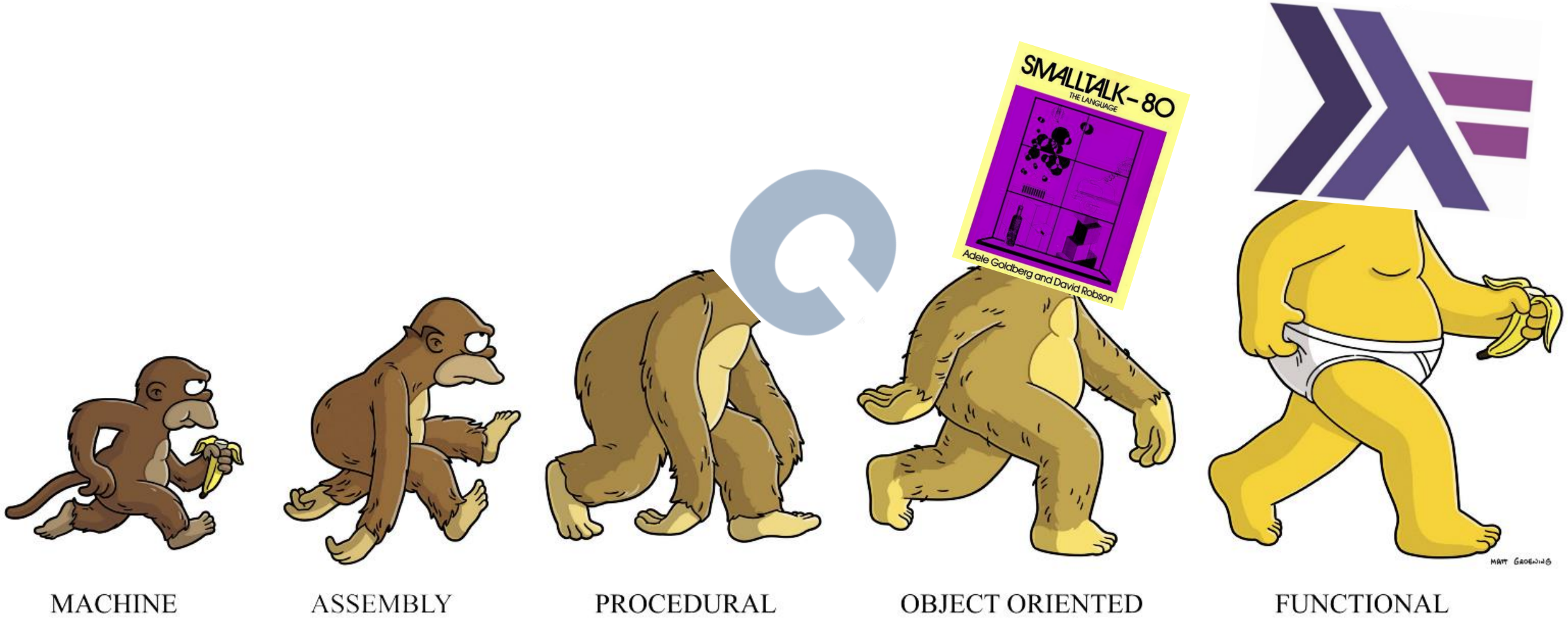
Strict (eager) VS. non-strict (lazy) evaluation:

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

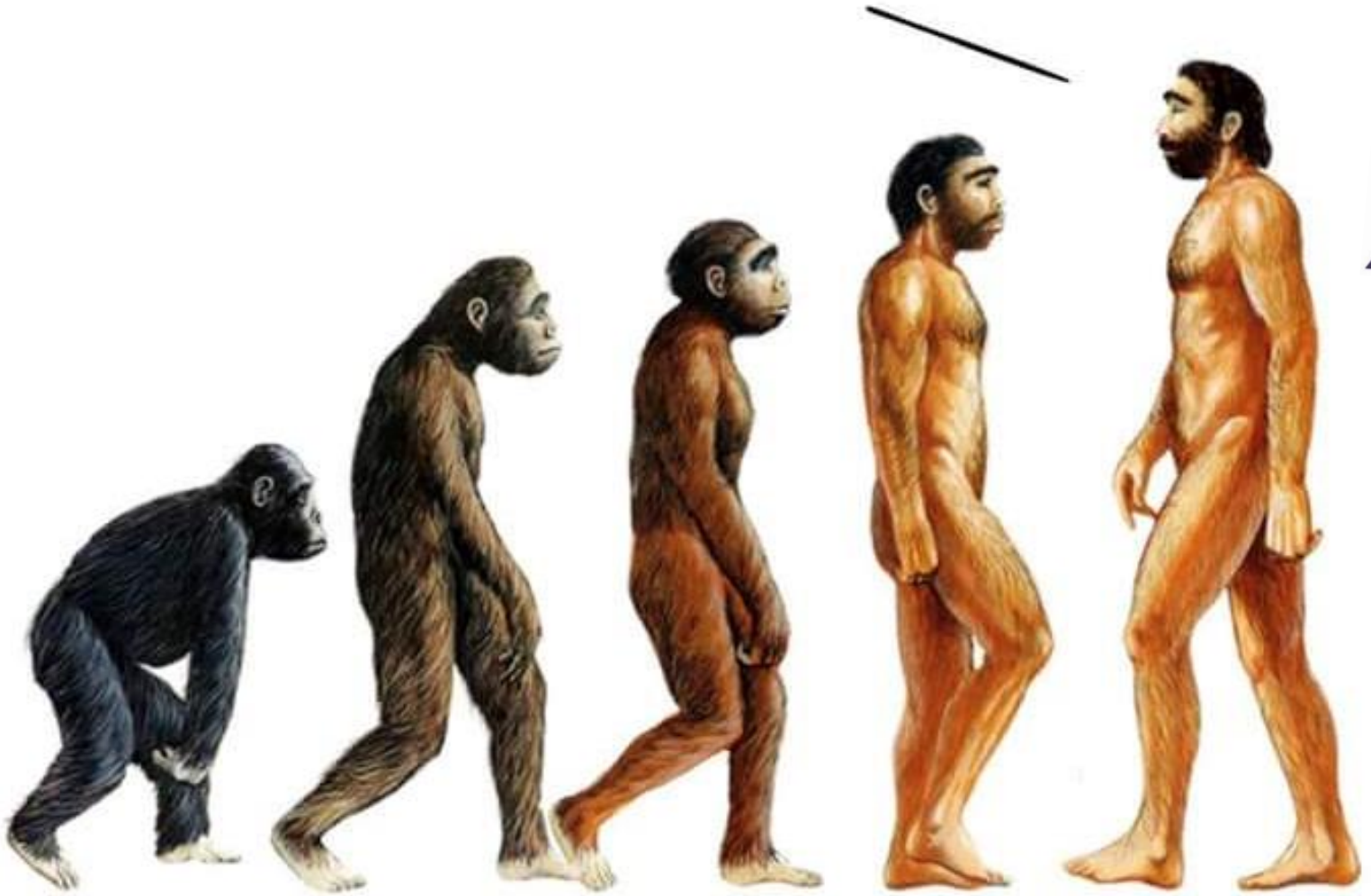
```
Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10]

Interactive Elixir (1.6.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> length [2+1, 3*2, 1/0, 5-4]
** (ArithmeticError) bad argument in arithmetic expression
    :erlang./(1, 0)
iex(1)>
```

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

Functional Programming



Go back. We f*cked up.



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.

Try it!

```
Type Haskell expressions in here.
λ length [2+1, 3*2, 1/0, 5-4]
4 :: Int
λ
```

Got 5 minutes?

Type `help` to start the tutorial.

Or try typing these out and see what happens (click to insert):

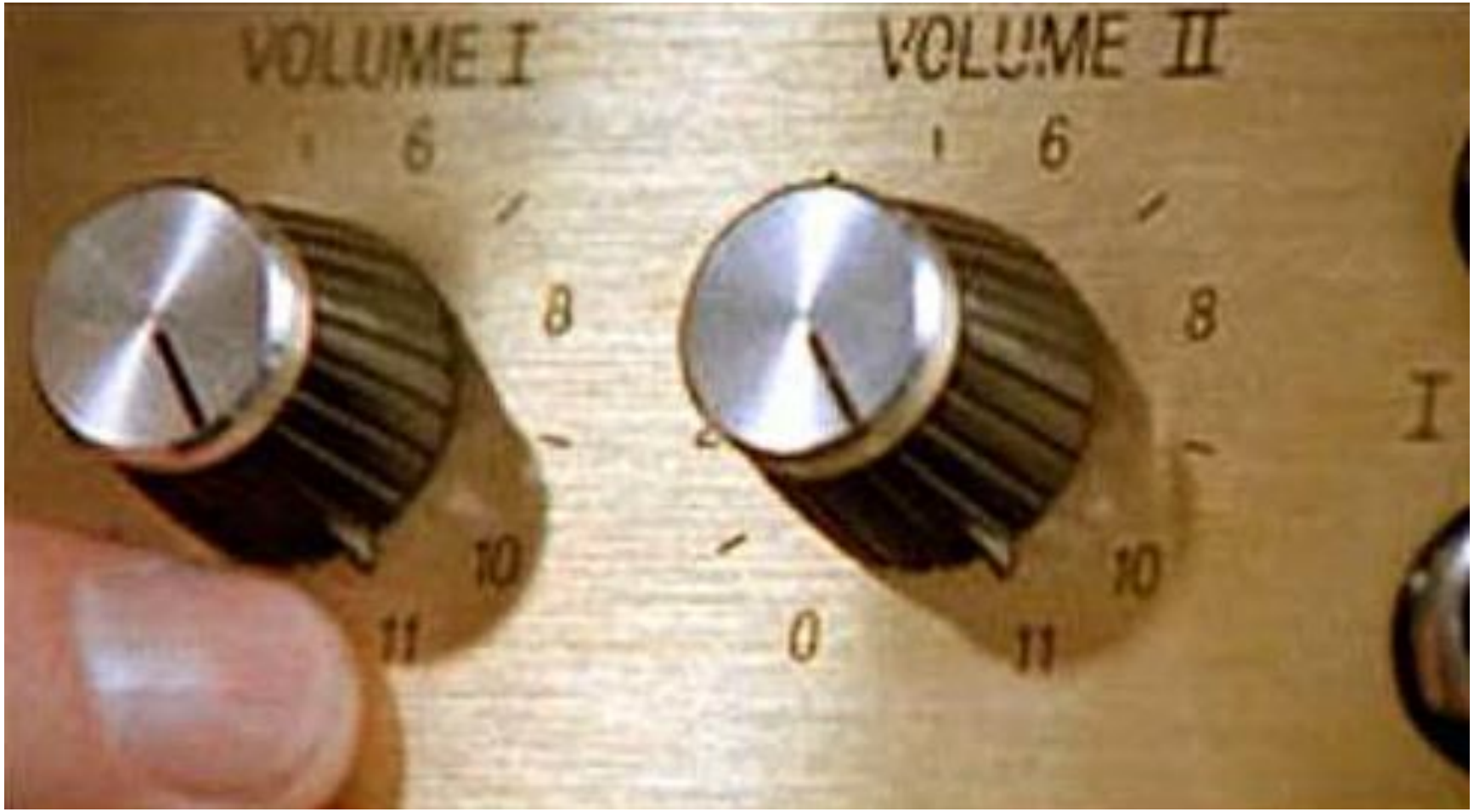
`23 * 36` or `reverse "hello"` or `foldr (:) [] [1,2,3]` or `do line <- getLine; putStrLn line` or `readFile "/welcome"`

These IO actions are supported in this sandbox.

https://www.haskell.org/

A great intro to Haskell syntax

Haskell: Functional Programming cranked up to 11



History

- Named after logician Haskell Curry
- In the late 80s, interest in lazy functional languages was growing
- There was a strong consensus to define an open standard 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*
- Haskell 2010 was published in July 2010
  - Contained uncontroversial features previously enabled via compiler flags
- Haskell 2020 was intended for 2020
  - GHC2021 finally released on Oct 29, 2021

Features

Purely Functional:

- Every function is *pure*
- No statements, only expressions
- Cannot mutate variables (local or global)
- Supports pattern matching
- Even side-effect inducing operations are produced by pure code
- Side effects are handled using *monads*

Features

Statically Typed:

- Every expression has a type
  - Determined at compile time
- Types composing expressions must match
  - If not, compile error

Type Inference:

- 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
- Expressions bound to variables don’t evaluate unless the result is used
- Computation ***never*** takes place unless a result is used.

Concurrency:

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

Haskell in Industry?

https://wiki.haskell.org/Haskell_in_industry

Haskell in industry

Haskell has a diverse range of use commercially, from aerospace and defense, to finance, to web startups, hardware design firms and a lawnmower manufacturer. This page collects resources on the industrial use of Haskell.

- The main user conference for industrial Haskell use is CUFP - the Commercial Users of Functional Programming Workshop.
- The Industrial Haskell Group supports commercial users.
- There is a well-maintained (as of 2018) github repository that collects information on companies using Haskell.
- The commercial Haskell group is a special interest group for companies and individuals interested in commercial usage of Haskell.

The Reddit page 72 would-be commercial Haskell users: what Haskell success stories we need to see has several stories of commercial Haskell users.

1 Haskell in Industry

Many companies have used Haskell for a range of projects, including:

- ABN AMRO Amsterdam, The Netherlands

  ABN AMRO is an international bank headquartered in Amsterdam. For its investment banking activities it needs to measure the counterparty risk on portfolios of financial derivatives.

  ABN AMRO's CUFP talk.

- Aetion Technologies LLC, Columbus, Ohio

  Aetion was a defense contractor in operation from 1999 to 2011, whose applications use artificial intelligence. Rapidly changing priorities make it important to minimize the code impact of changes, which suits Haskell well. Aetion developed three main projects in Haskell, all successful. Haskell's concise code was perhaps most important for rewriting: it made it practicable to throw away old code occasionally. DSELs allowed the AI to be specified very declaratively.

  Aetion's CUFP talk.

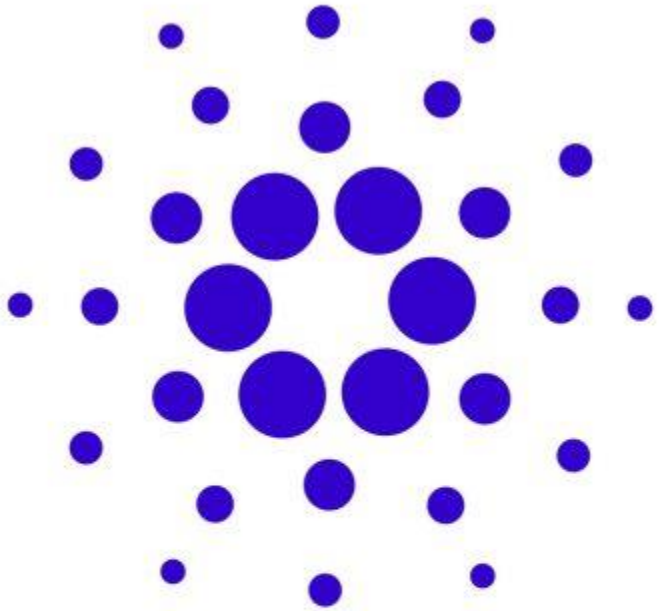
- Alcatel-Lucent

  A consortium of groups, including Alcatel-Lucent, have used Haskell to prototype narrowband software radio systems, running in (soft) real-time.

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.



CARDANO

https://medium.com/@cardano.foundation/why-cardano-chose-haskell-and-why-you-should-care-why-cardano-chose-haskell-and-why-you-should-f97052db2951

Installing Haskell:

https://www.haskell.org/

Haskell Documentation:

https://www.haskell.org/documentation/

Haskell Basics:

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

Haskell

An advanced, purely functional programming language

Declarative, statically typed code.

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mod` p /= 0]
```

Try it!

Type Haskell expressions in here.

λ

Neat!

Got 5 minutes?

Type `help` to start the tutorial.

Or try typing these out and see what happens (click to insert):

`23 * 36` or `reverse "hello"` or `foldr (:) []` `[1,2,3]` or `do line <- getLine; putStrLn line` or `readFile "/welcome"`

These IO actions are supported in this sandbox.

Downloads Community Documentation News

Hask

An advanced, purely functional programm

Declarative, statically typed code.

```
primes = filterPrime [2..]
  where filterPrime (p:xs) =
          p : filterPrime [x | x <- xs, x `mo
```

Haskell Platform

What it is

The Haskell Platform is a self-contained, all-in-one installer. After download, you will have everything necessary to build Haskell programs against a core set of useful libraries. It comes in both minimal versions with tools but no libraries outside of GHC core, or full versions, which include a broader set of globally installed libraries.

What you get

- The Glasgow Haskell Compiler
- The Cabal build system, which can install new packages, and by default fetches from Hackage, the central Haskell package repository.
- the Stack tool for developing projects
- Support for profiling and code coverage analysis
- 35 core & widely-used packages

How to get it

The Platform is provided as a single installer, and can be downloaded at the links below.

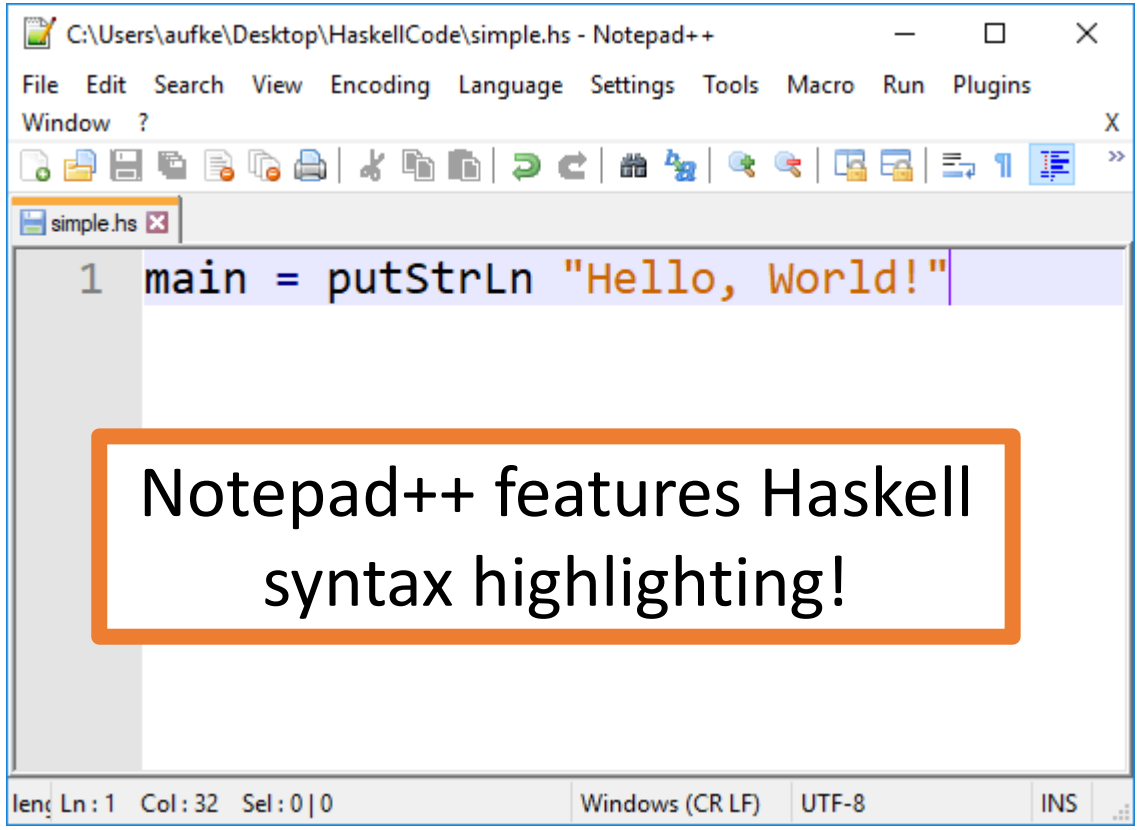
- Linux
- OS X
- Windows

```
PS C:\Users\aufke> ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/  :? for help
Prelude>
```

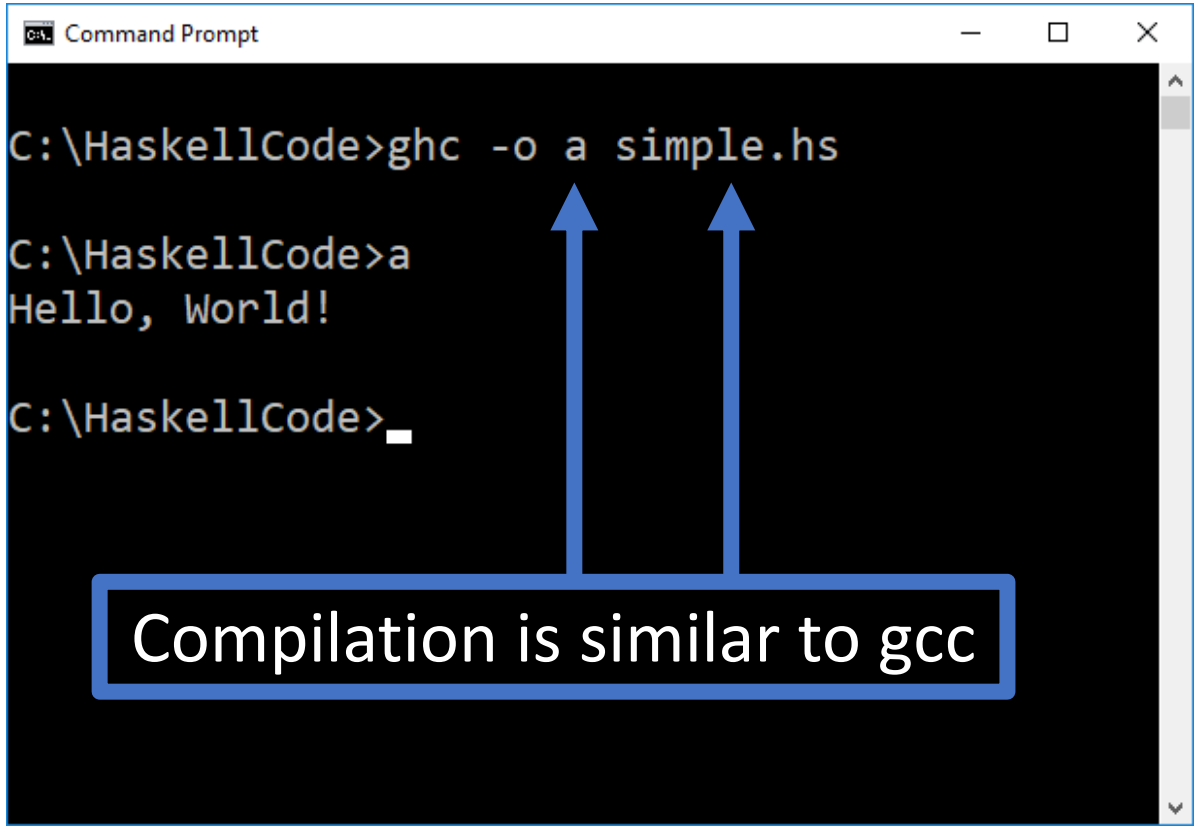
- Interactive shell, just like Elixir.
- Run **`ghci`** in the terminal of your choice.

Hello, World!

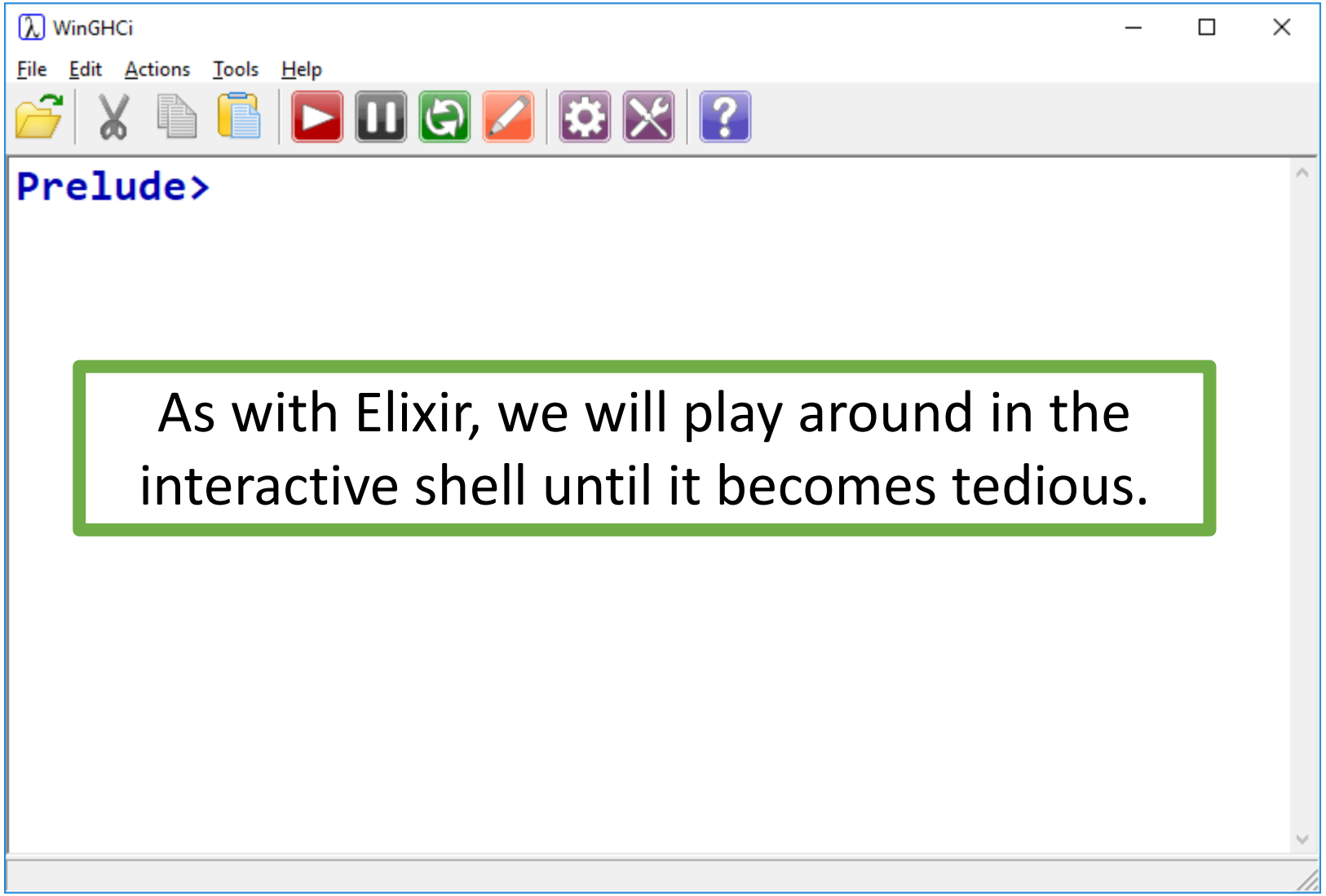
`main()` is the entry point of a Haskell program



Notepad++ features Haskell syntax highlighting!

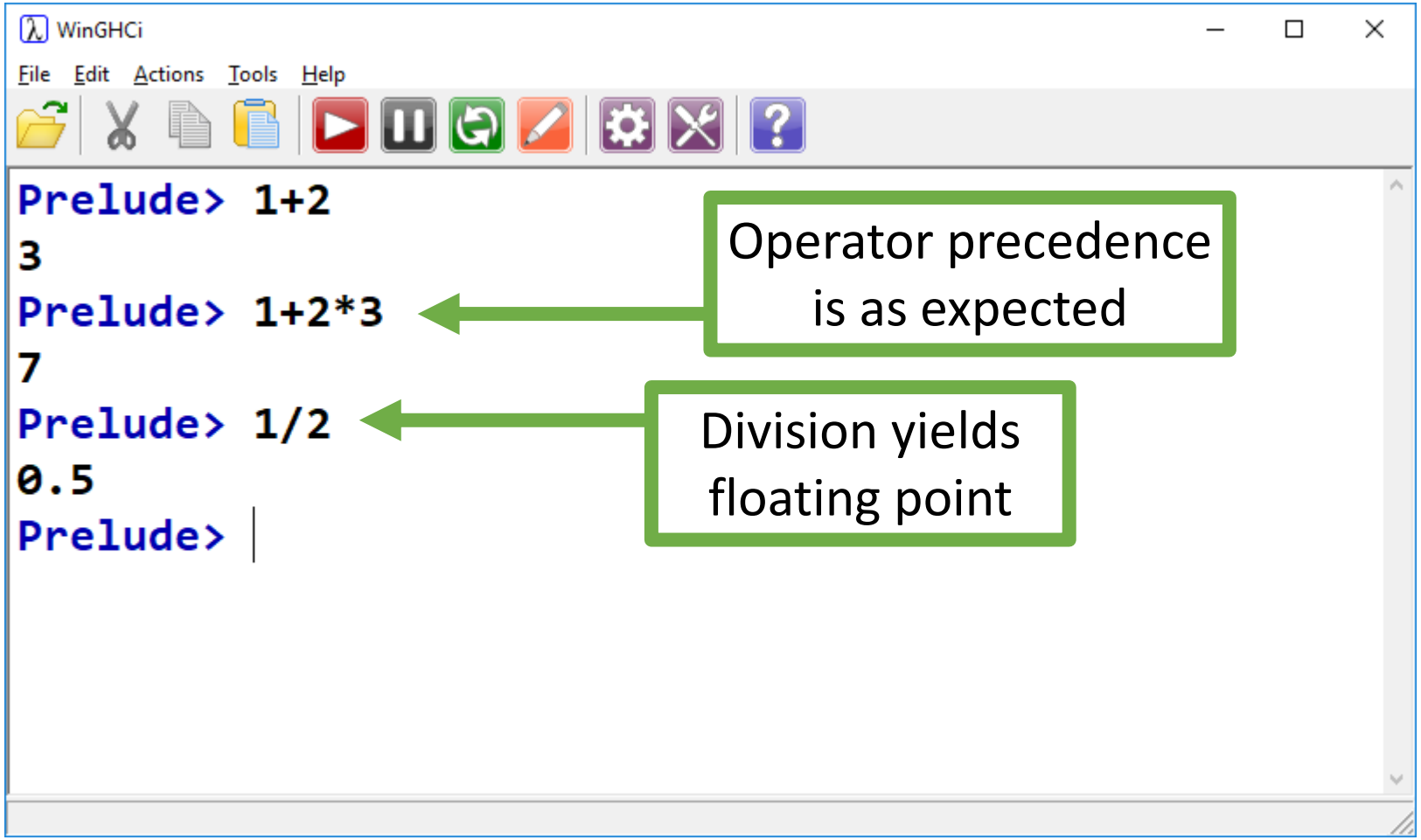


Compilation is similar to gcc

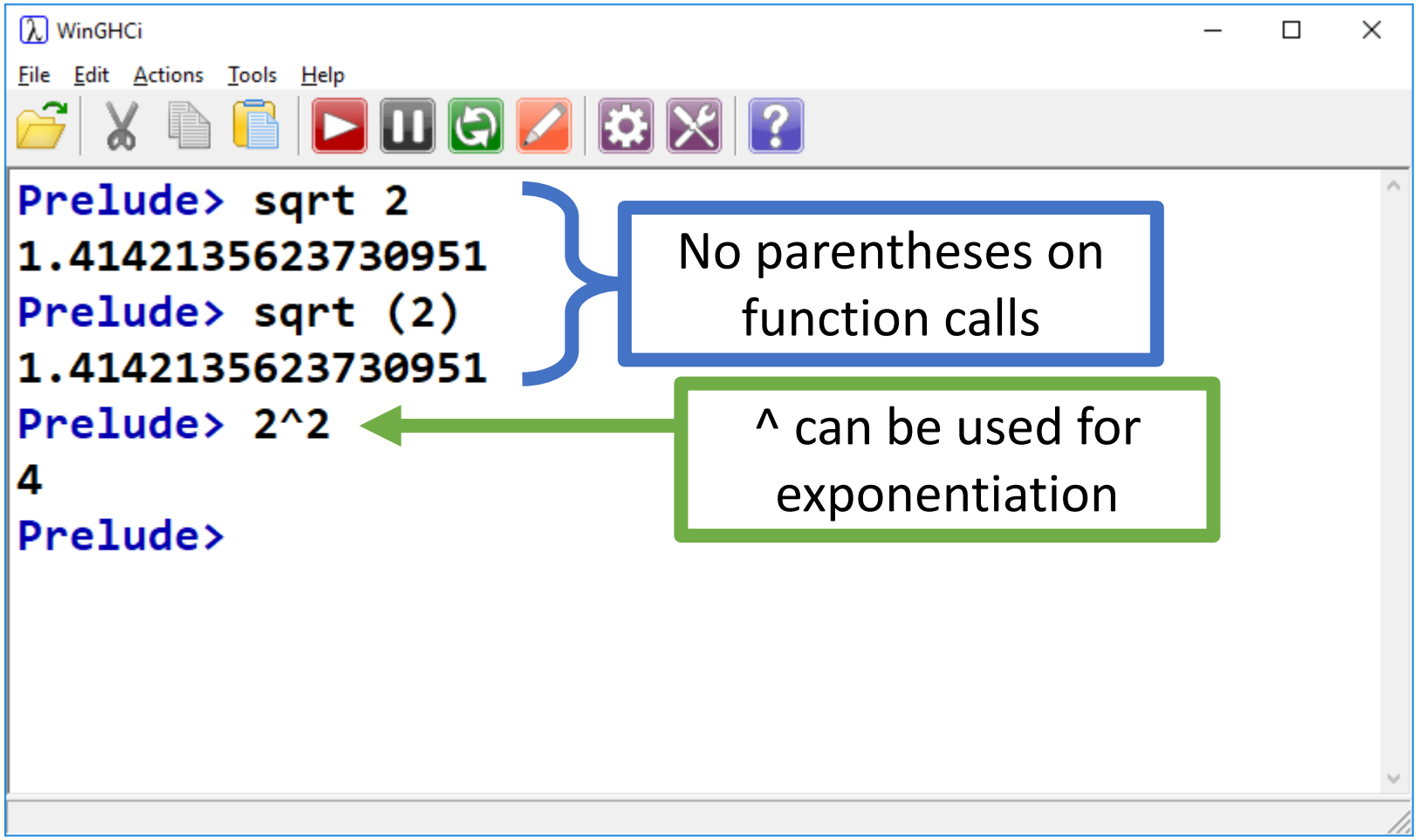


As with Elixir, we will play around in the interactive shell until it becomes tedious.

Literals & Arithmetic



Literals & Arithmetic



Tuples

WinGHCi

File Edit Actions Tools Help

```
Prelude> (3, 5)
(3,5)
Prelude> (5, "hello")
(5,"hello")
Prelude> fst (5, "hello")
5
Prelude> snd (5, "hello")
"hello"
Prelude> (1, 2, 3, "Hello", "World")
(1,2,3,"Hello","World")
Prelude>
```

- Like Elixir, Haskell supports tuples.
- They need not contain the same types.

There are built in functions for accessing first and second elements. Great for coordinates.

WinGHCi

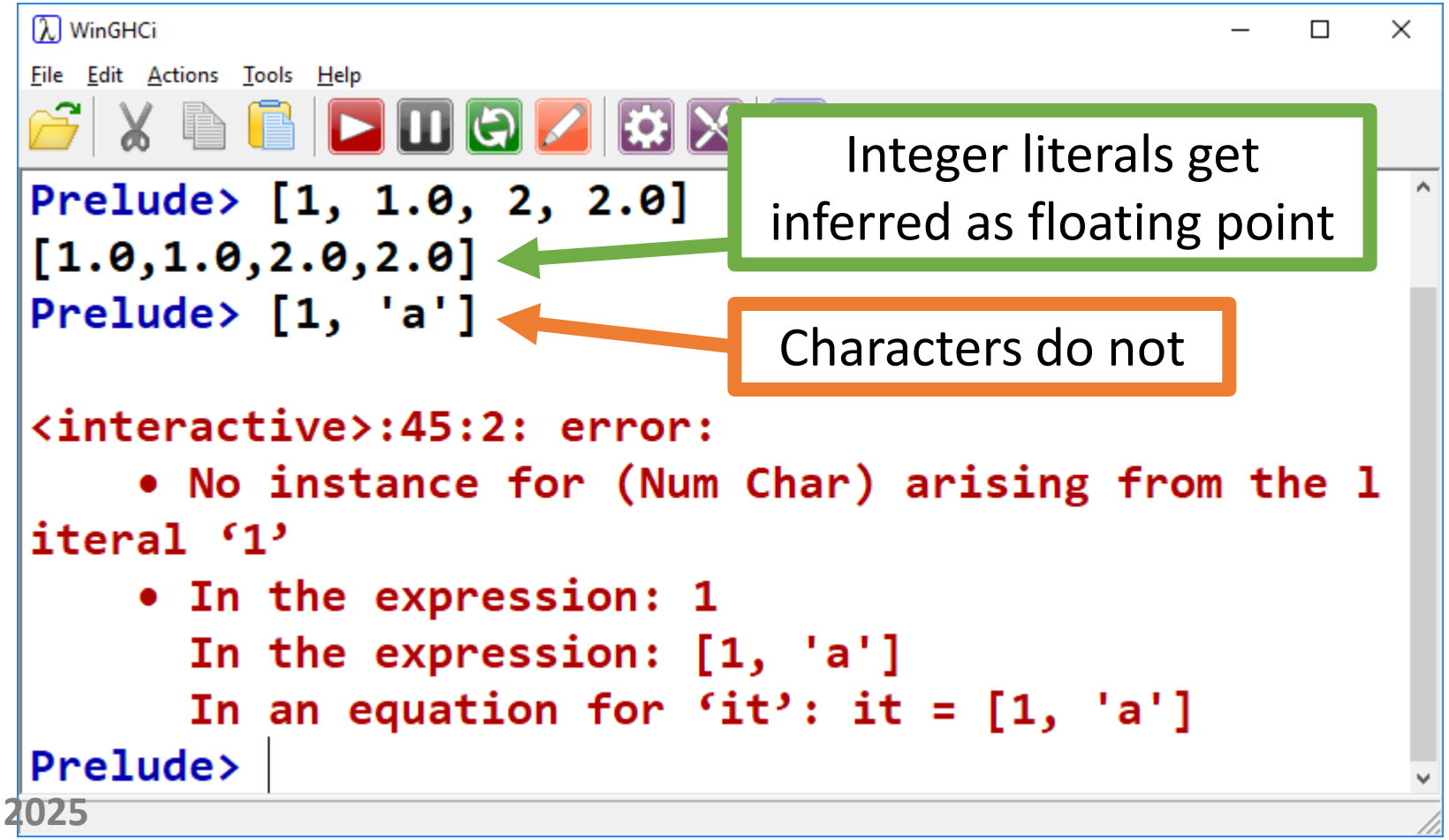
File Edit Actions Tools Help

```
Prelude> fst (1, 2, 3)

<interactive>:35:5: error:
    • Couldn't match expected type ‘(a, b0)’
                  with actual type ‘(Integer, Integ
er, Integer)’
    • In the first argument of ‘fst’, namely ‘(1, 2
, 3)’
      In the expression: fst (1, 2, 3)
      In an equation for ‘it’: it = fst (1, 2, 3)
    • Relevant bindings include it :: a (bound at <
interactive>:35:1)
Prelude>
```

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

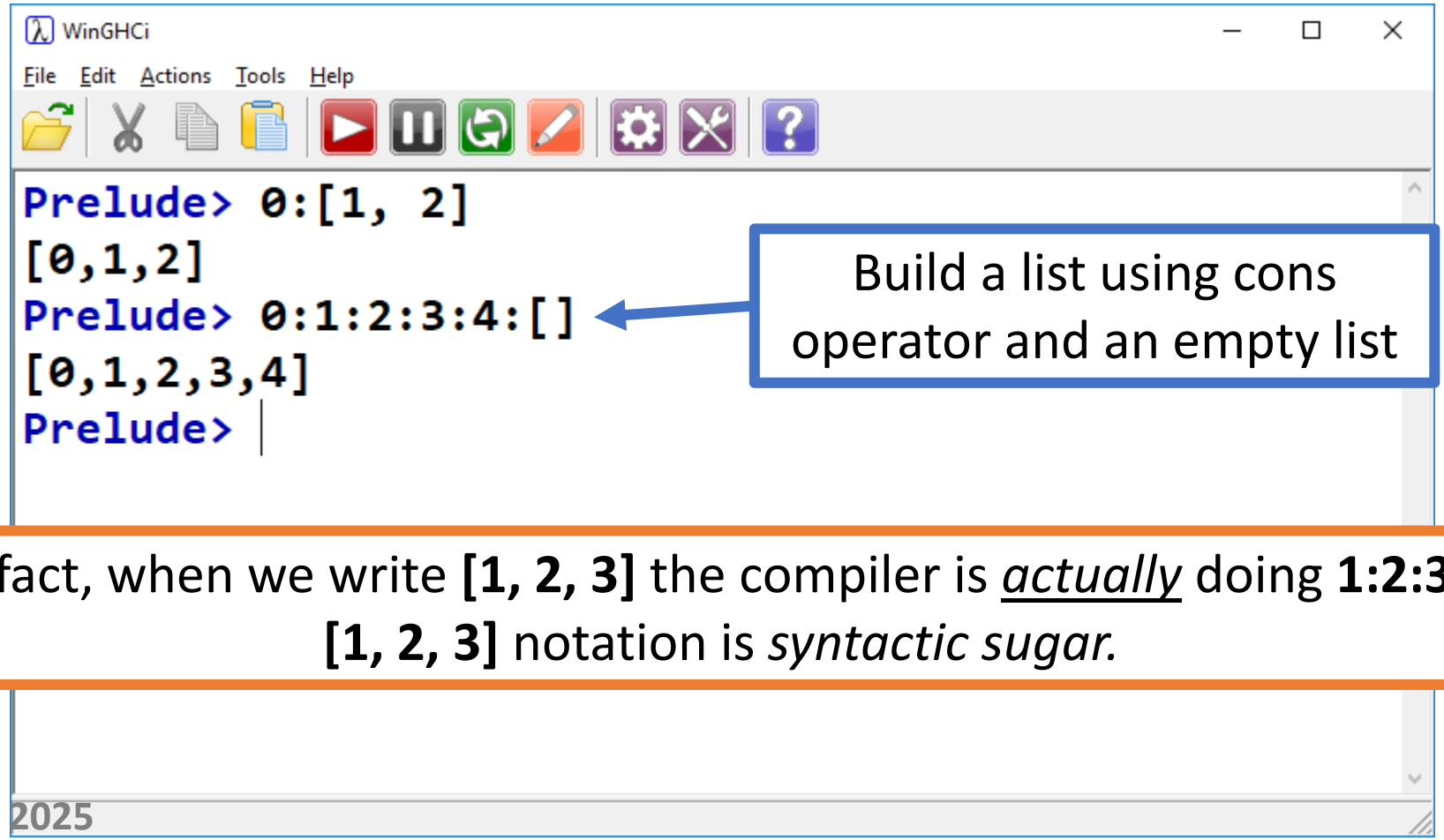
Lists

Must be ***homogeneous***:



Lists

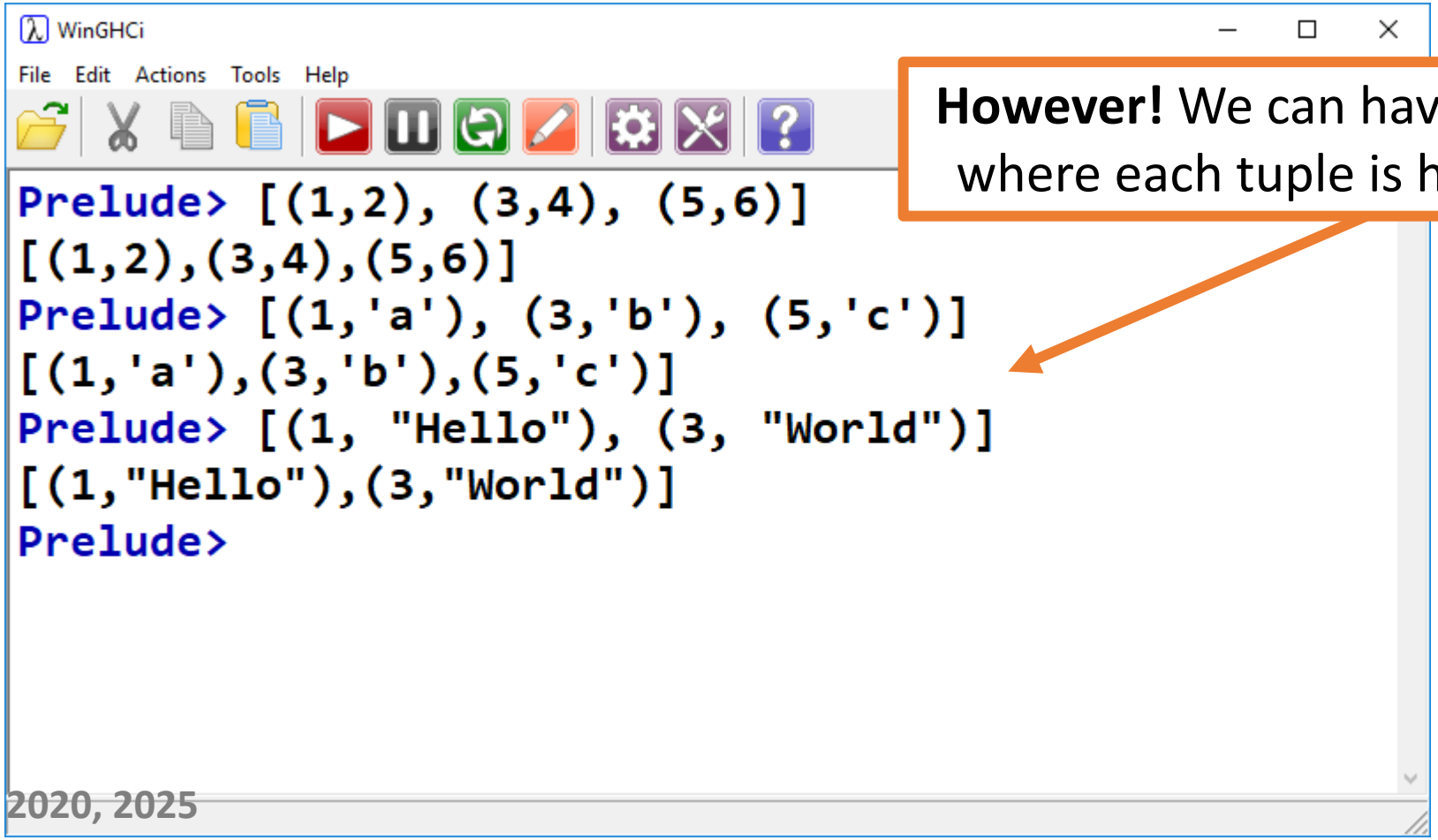
Elements can be added to the *beginning* of a list with the **cons** (:) operator



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, where each tuple is heterogeneous.

Lists & Tuples

Tuples can be heterogeneous, lists must be homogeneous.

WinGHCi

File Edit Actions Tools Help

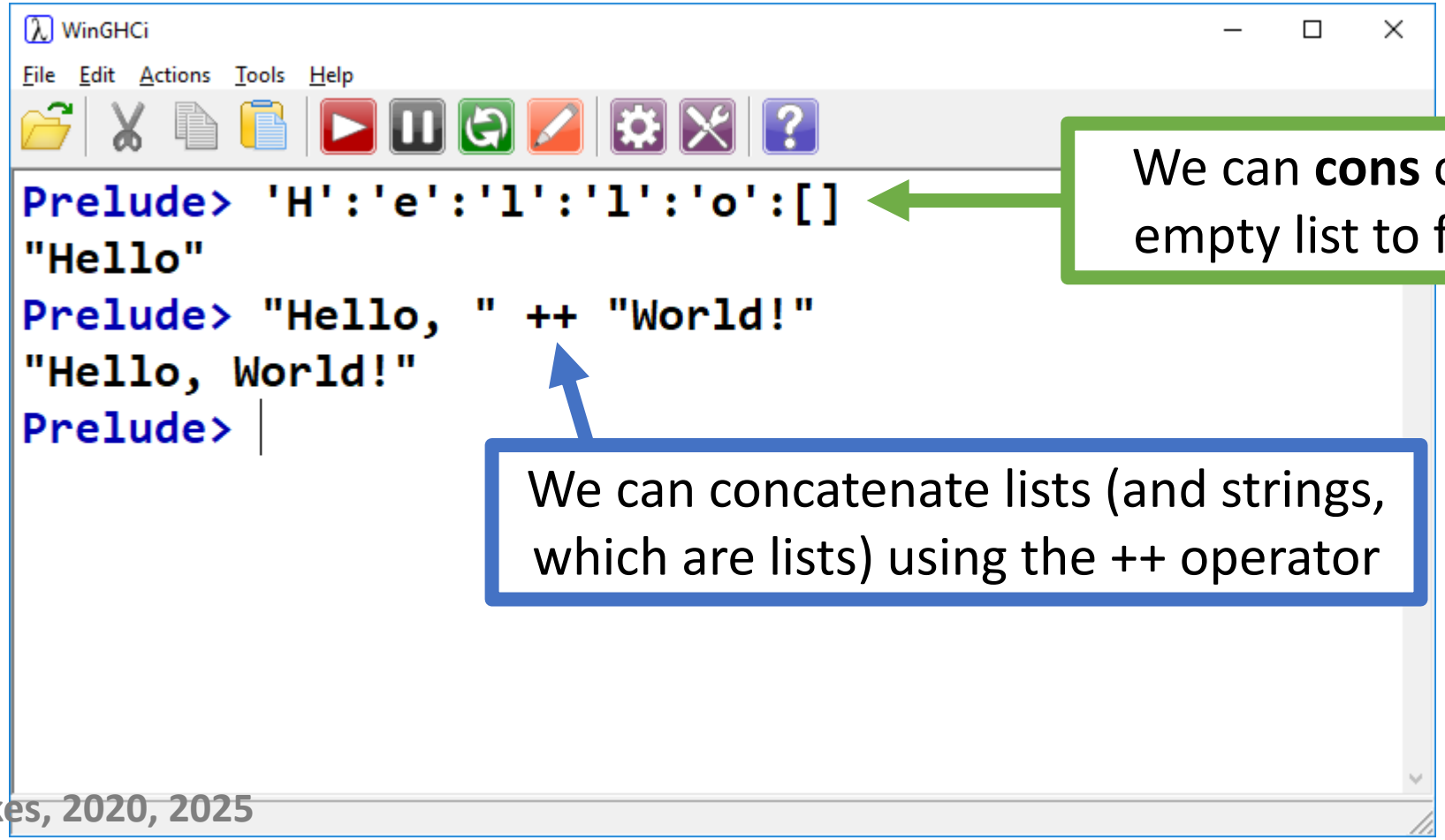
```
Prelude> [(1, "Hello"), (2, "World")]
[(1,"Hello"),(2,"World")]
Prelude> [(1, "Hello"), (2, 3.4)]

<interactive>:67:20: error:
    • Could not deduce (Fractional [Char])
        arising from the literal ‘3.4’
      from the context: Num a
        bound by the inferred type of it :: Num a => [(a, [Char])]
        at <interactive>:67:1-24
    • In the expression: 3.4
      In the expression: (2, 3.4)
      In the expression: [(1, "Hello"), (2, 3.4)]
Prelude>
```

However #2!

- In a list of tuples, each tuple must have the same format:

Strings

Strings are simply lists of chars:

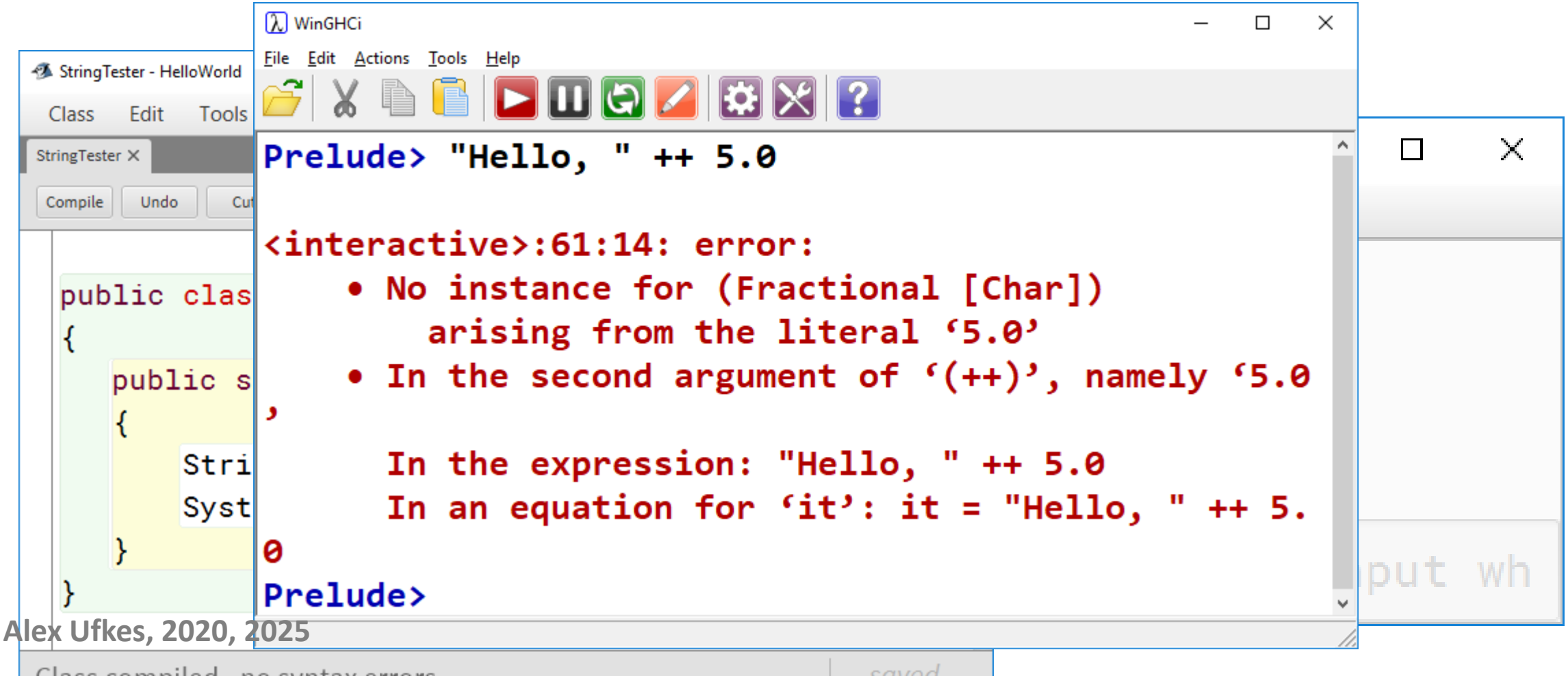


We can **cons** chars into an empty list to form a string

We can concatenate lists (and strings, which are lists) using the ++ operator

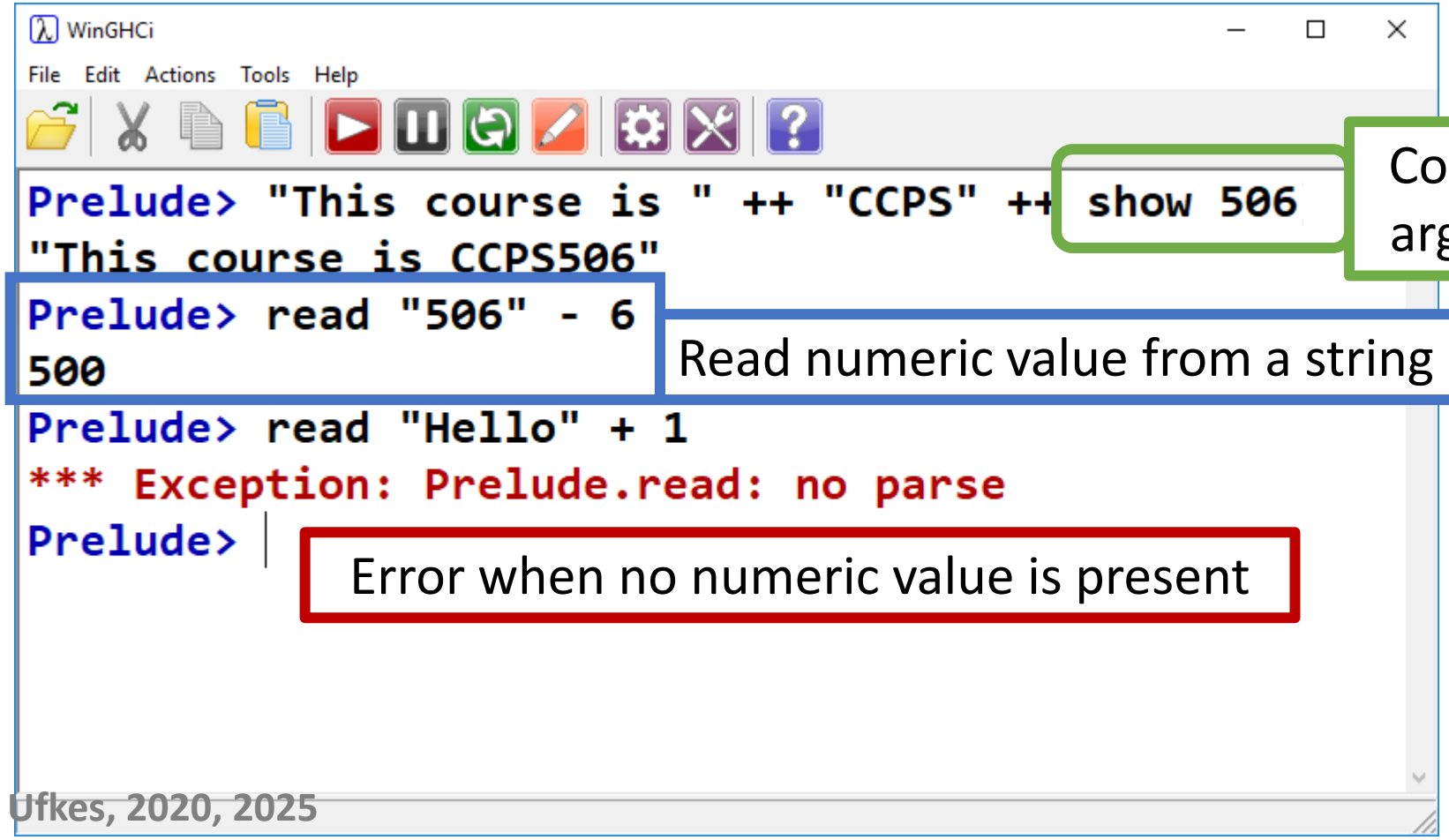
Strings

Concatenate multiple types? Java lets us…



Strings

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



Convert non-string argument to string

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

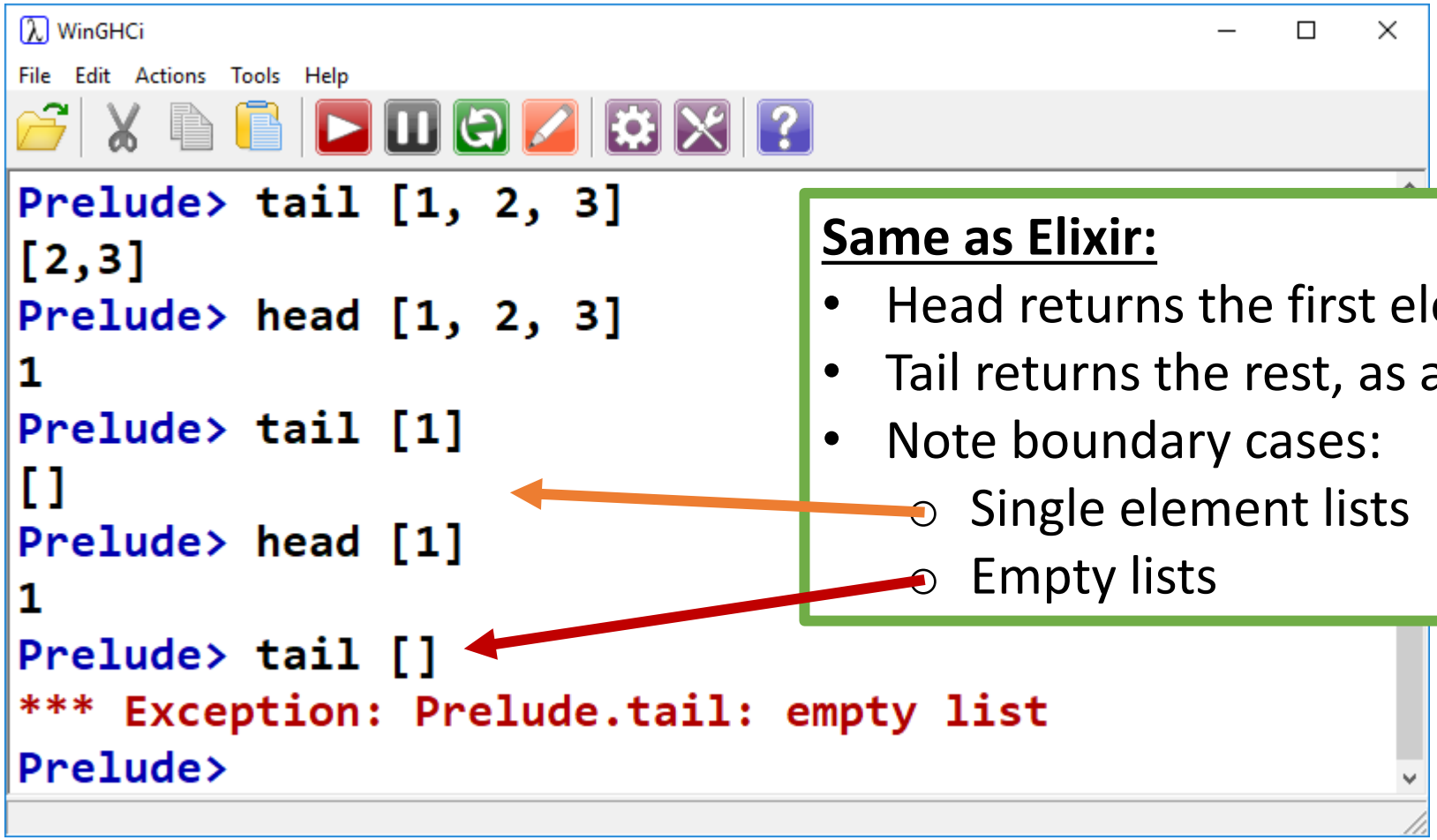
Error when no numeric value is present

Operations on Lists

- In functional programming, computation is done in large part by operating on lists.
- We saw the **`hd`**, **`tl`**, **`|`**, and **`Enum`** in Elixir.
- Haskell has a similar set of operations.

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

Head & Tail

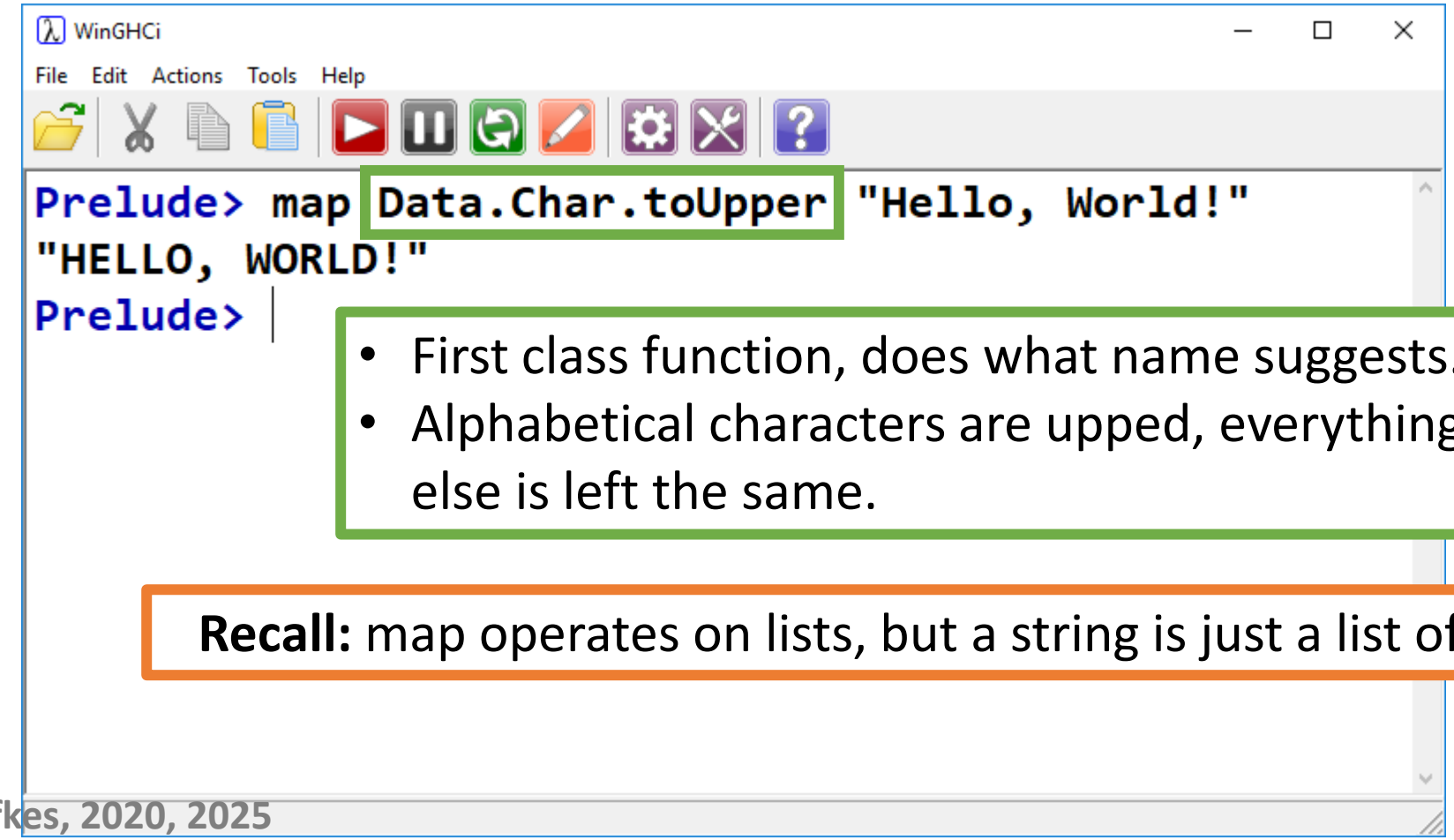


Same as Elixir:

- Head returns the first element
- Tail returns the rest, as a list
- Note boundary cases:
  - Single element lists
  - Empty lists

map

Like Elixir's **Enum.map**

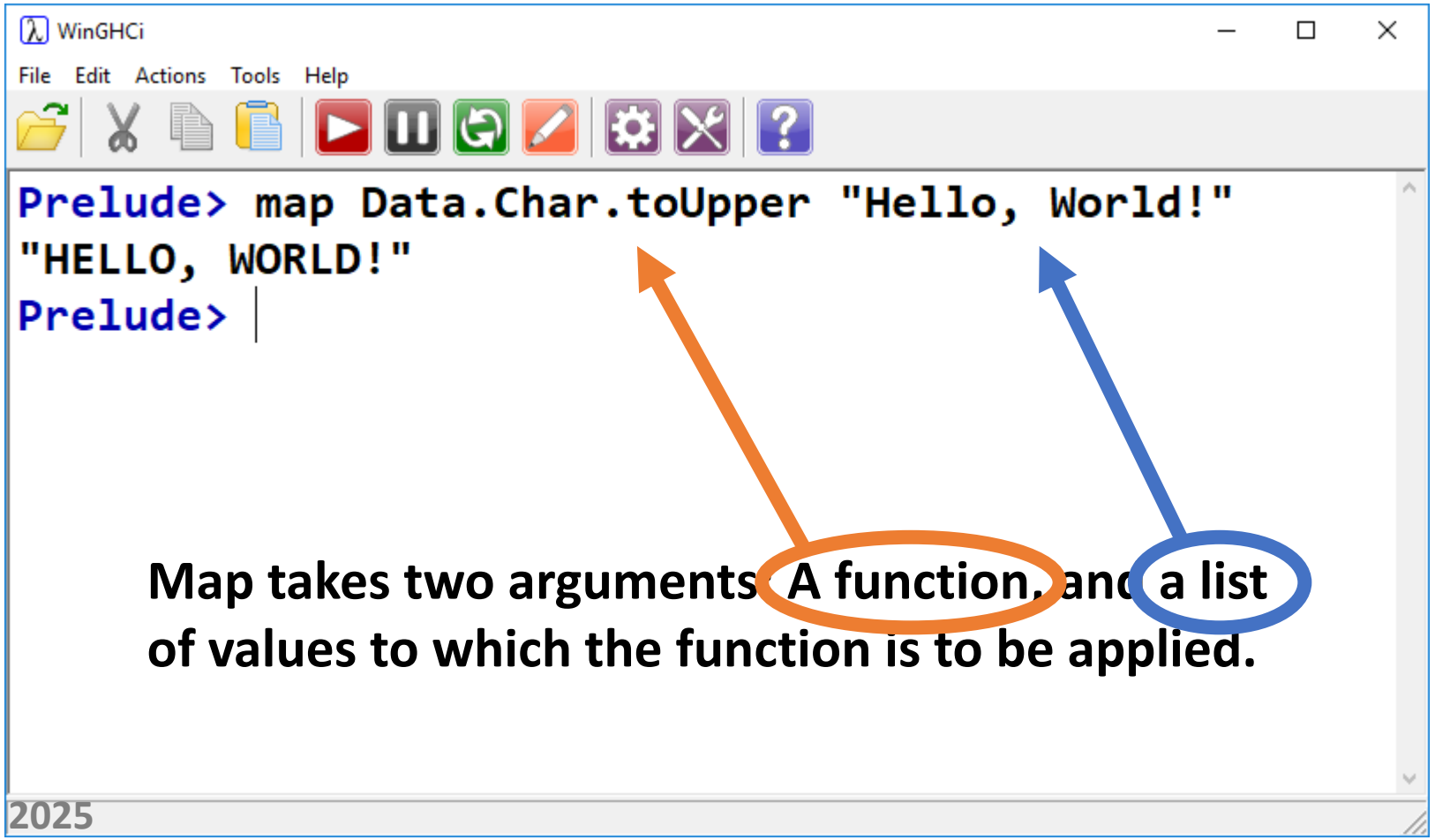


- First class function, does what name suggests.
- Alphabetical characters are upped, everything else is left the same.

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

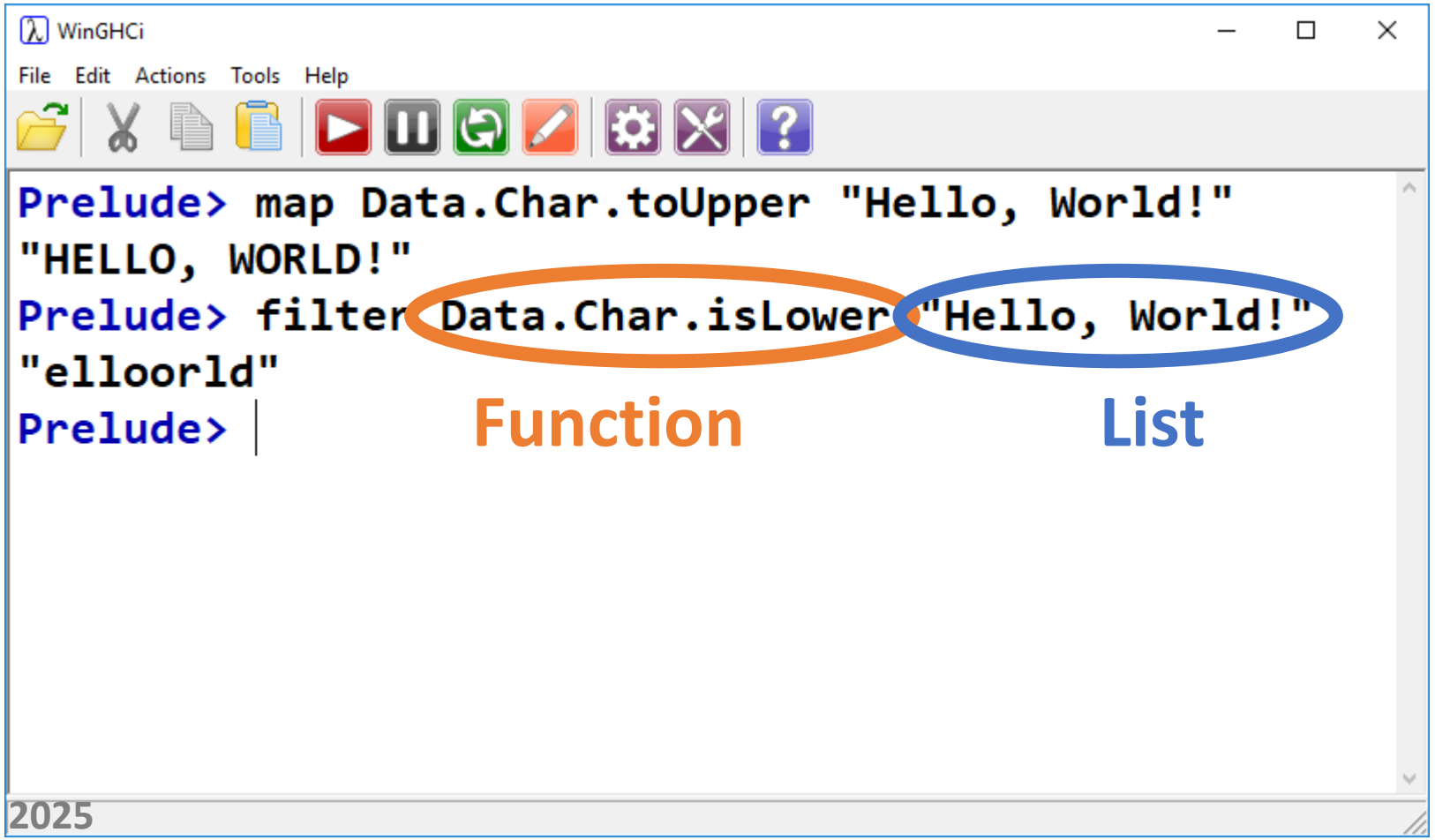
map

Like Elixir's **Enum.map**



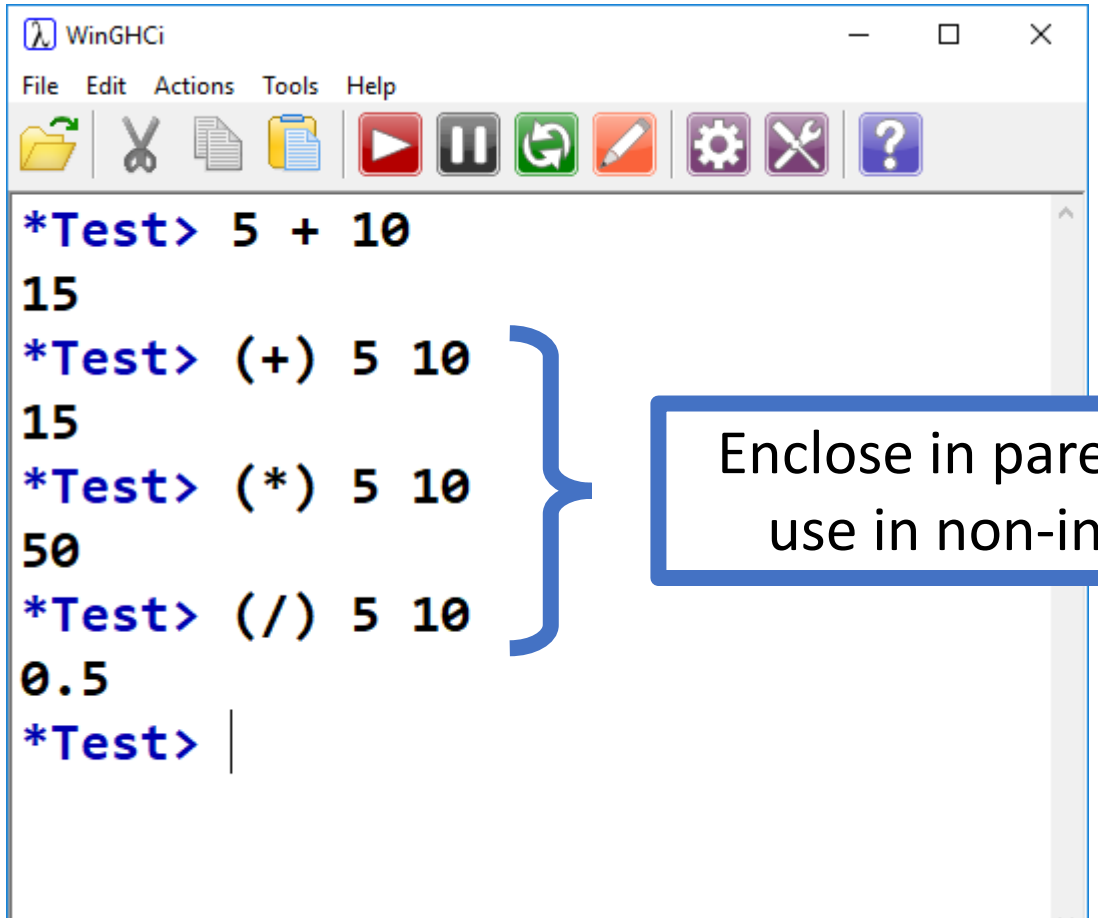
filter

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



Infix Functions

Use symbolic operators as functions:



Enclose in parentheses to use in non-infix mode

`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.

`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`** in effect replaces the cons operator with another function of our choosing.
- This is similar to **`Enum.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 function of our choosing.
- This is similar to **`Enum.reduce`** in Elixir.
- The empty list is replaced with some 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 1:2:3:4:5:[]
```

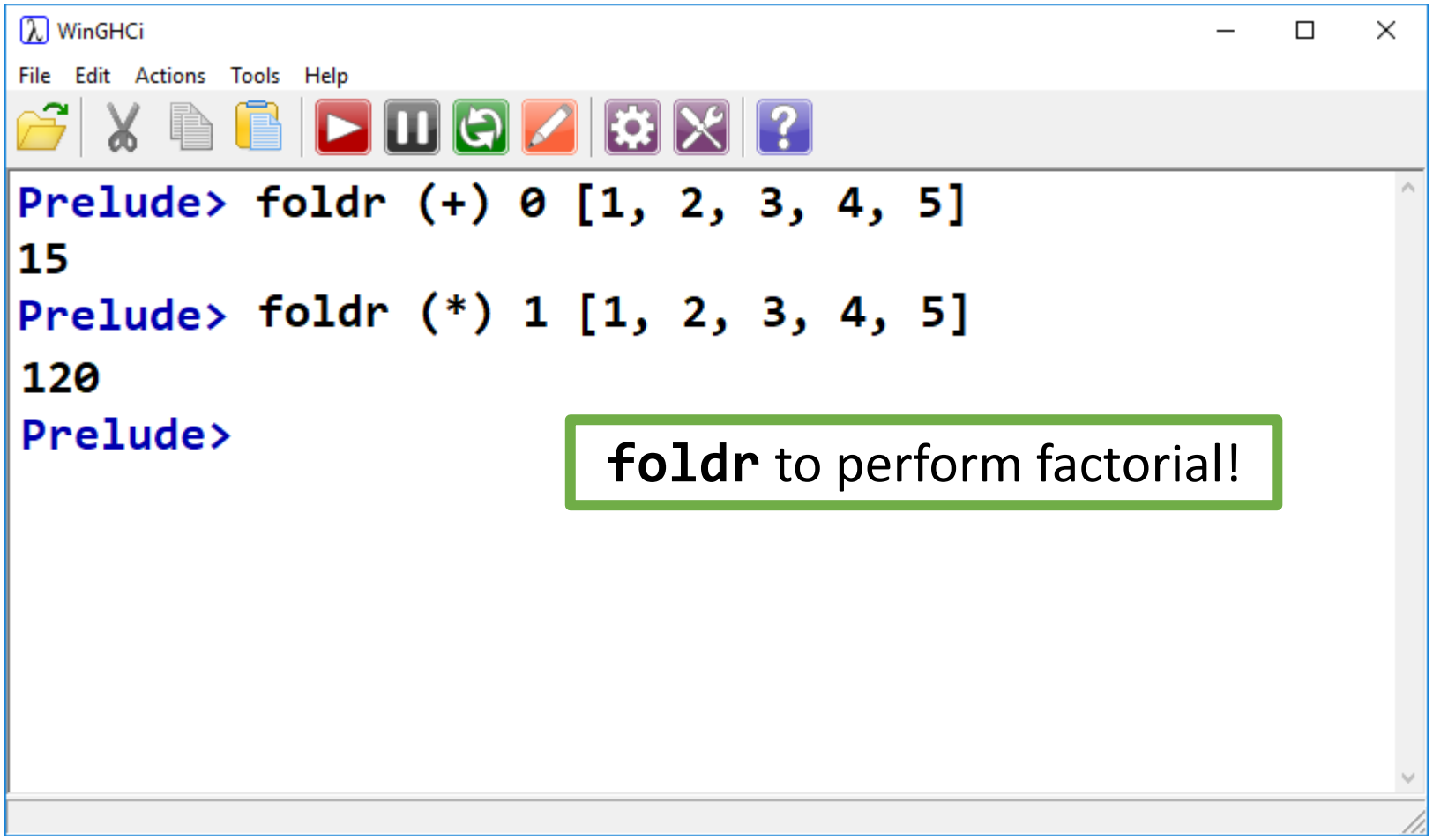
↓

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

↓

```
15
```

foldl, foldr



`foldl` VS `foldr`

`foldr` is *right associative*. Meaning:

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

```
1 + 2 + 3 + 4 + 5 + 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 - 5 - 1
```

Is actually:

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

↓

```
0
```

`foldl` VS `foldr`

`foldl` is *left associative*. Meaning:

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

↓

```
1 - 4 – 8 - 5
```

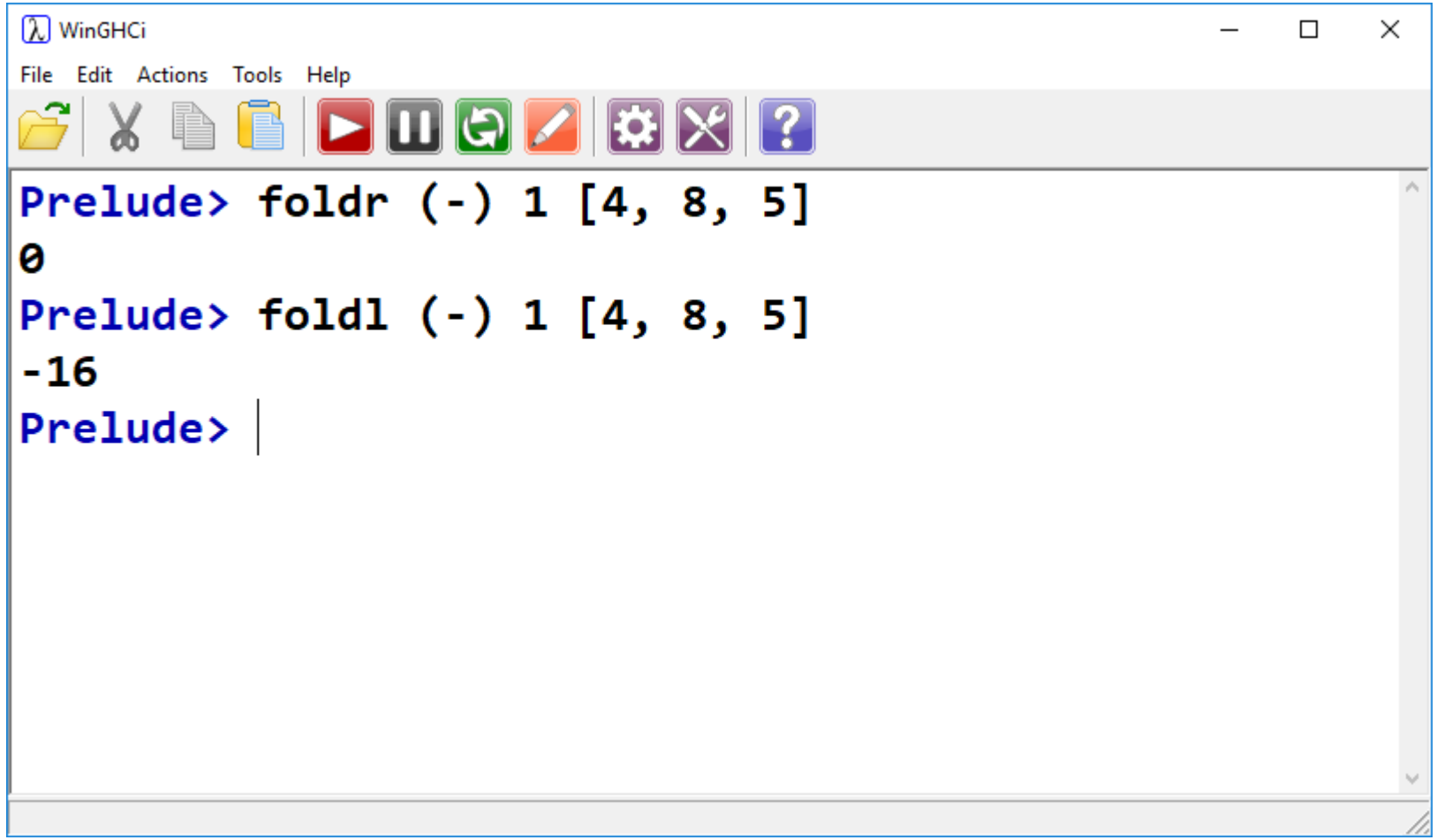
Is actually:

```
(((1 - 4) – 8) - 5)
```

↓

```
-16
```

foldl VS foldr



List Generation

Syntactic sugar:

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

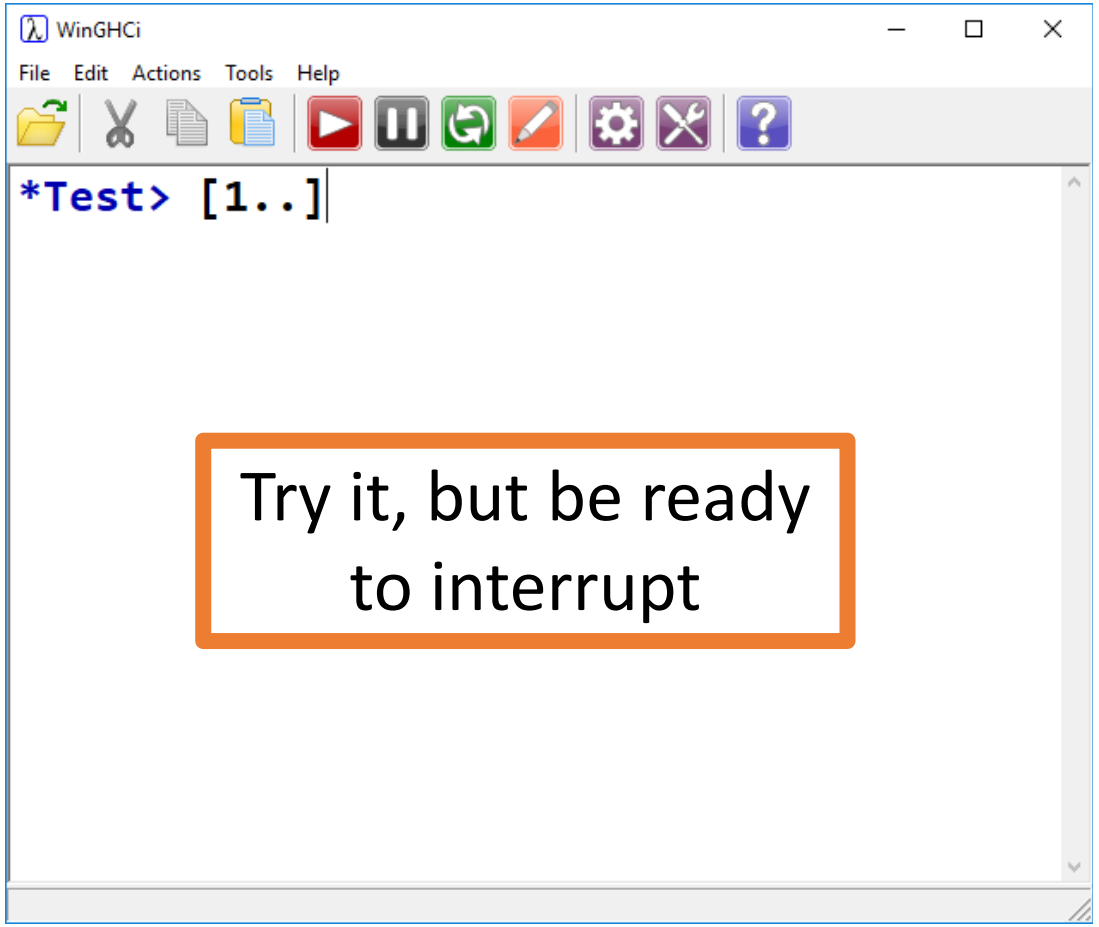
Can be written: `list = [1..9]`

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

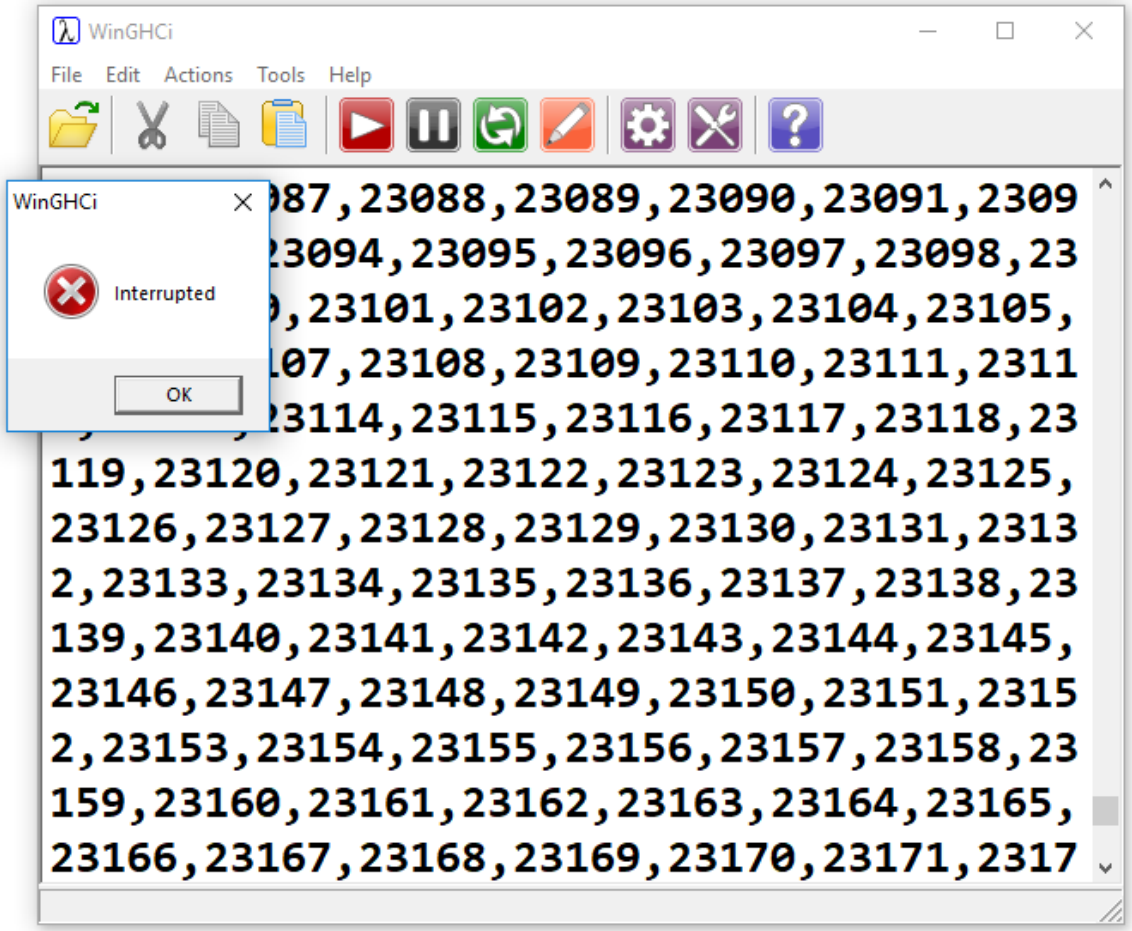
`= [1,3,5,7,9]`

Interval is discerned from difference between first two elements

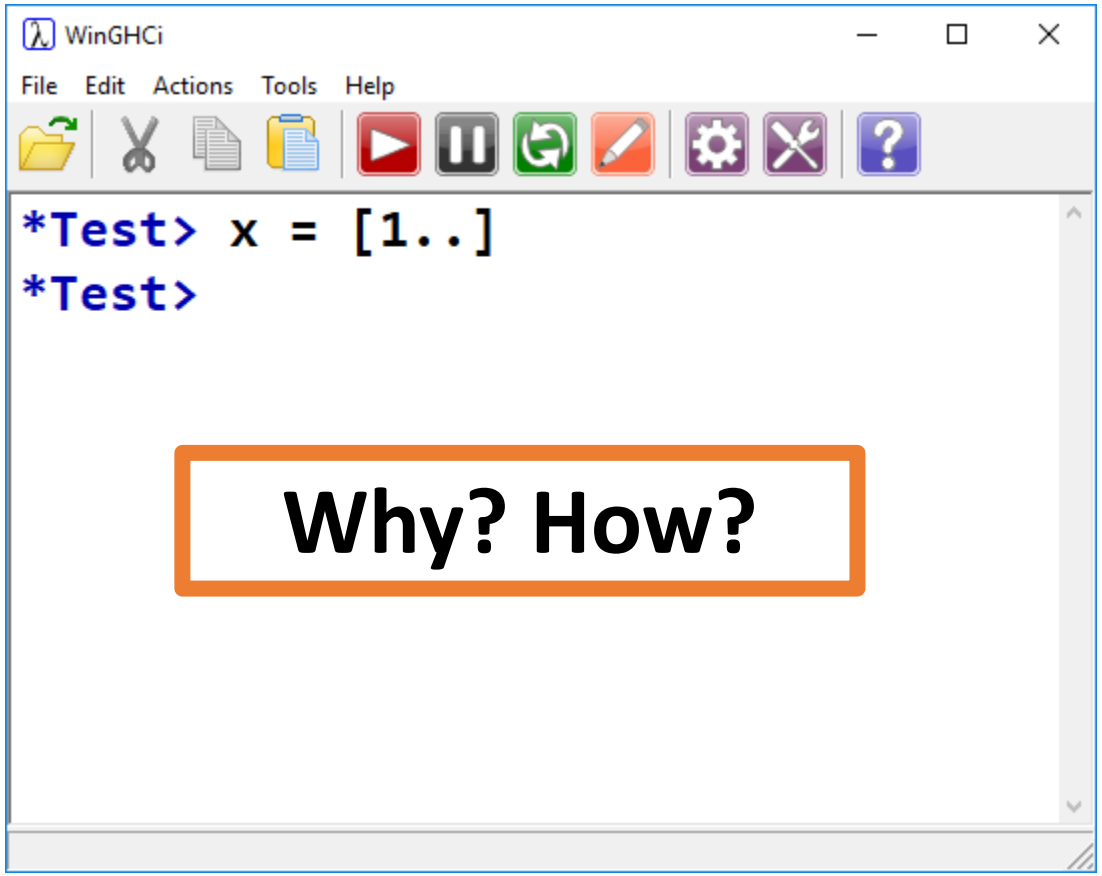
Infinite Lists?



Try it, but be ready to interrupt



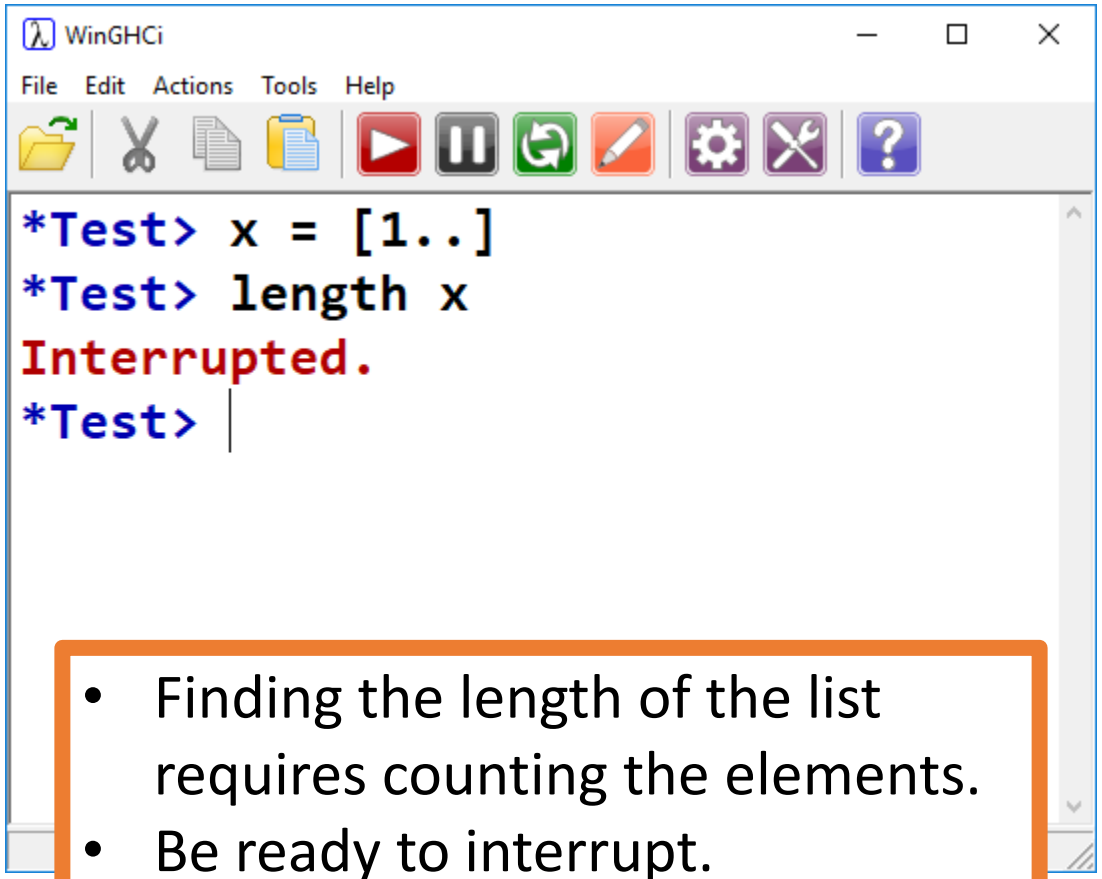
Infinite Lists?



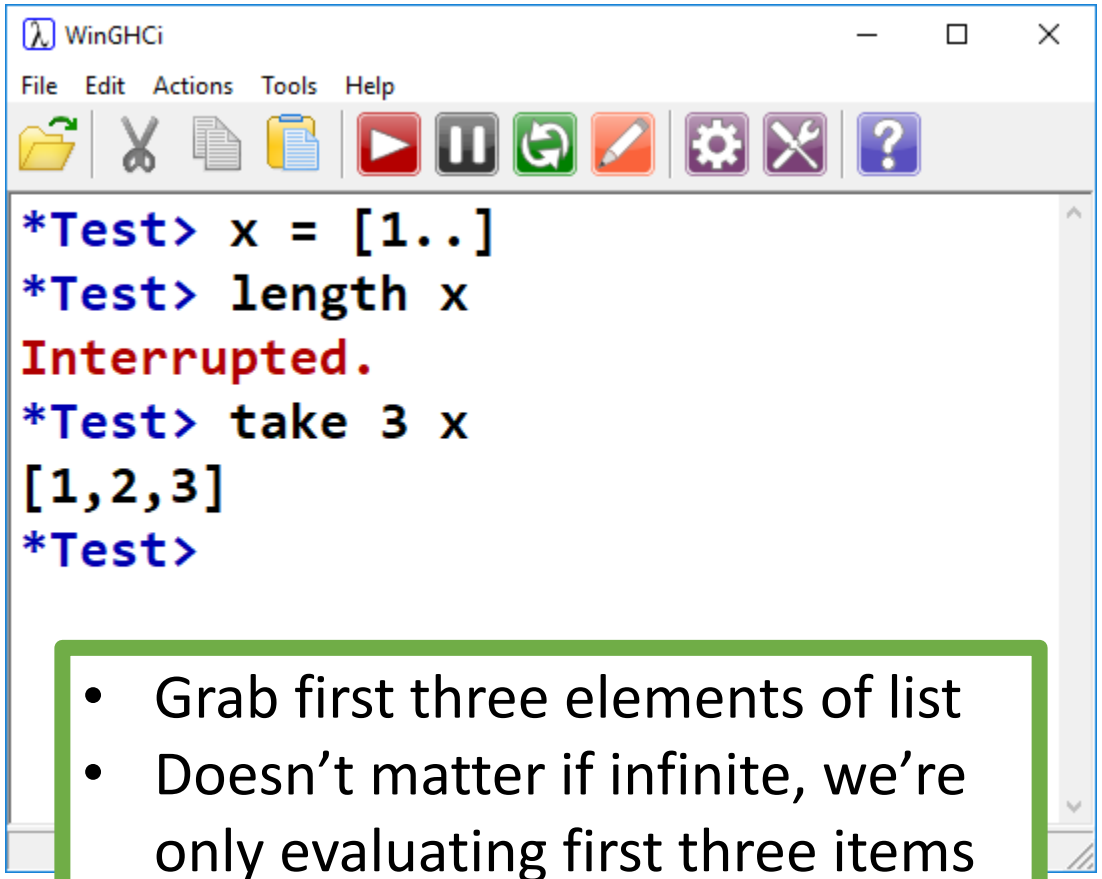
Haskell is lazy!

- We bind x to the expression to generate an infinite list.
- We don't have to *evaluate* this list to do so!
- Displaying the list, however, requires evaluation.

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.

WinGHCi

File Edit Actions Tools Help

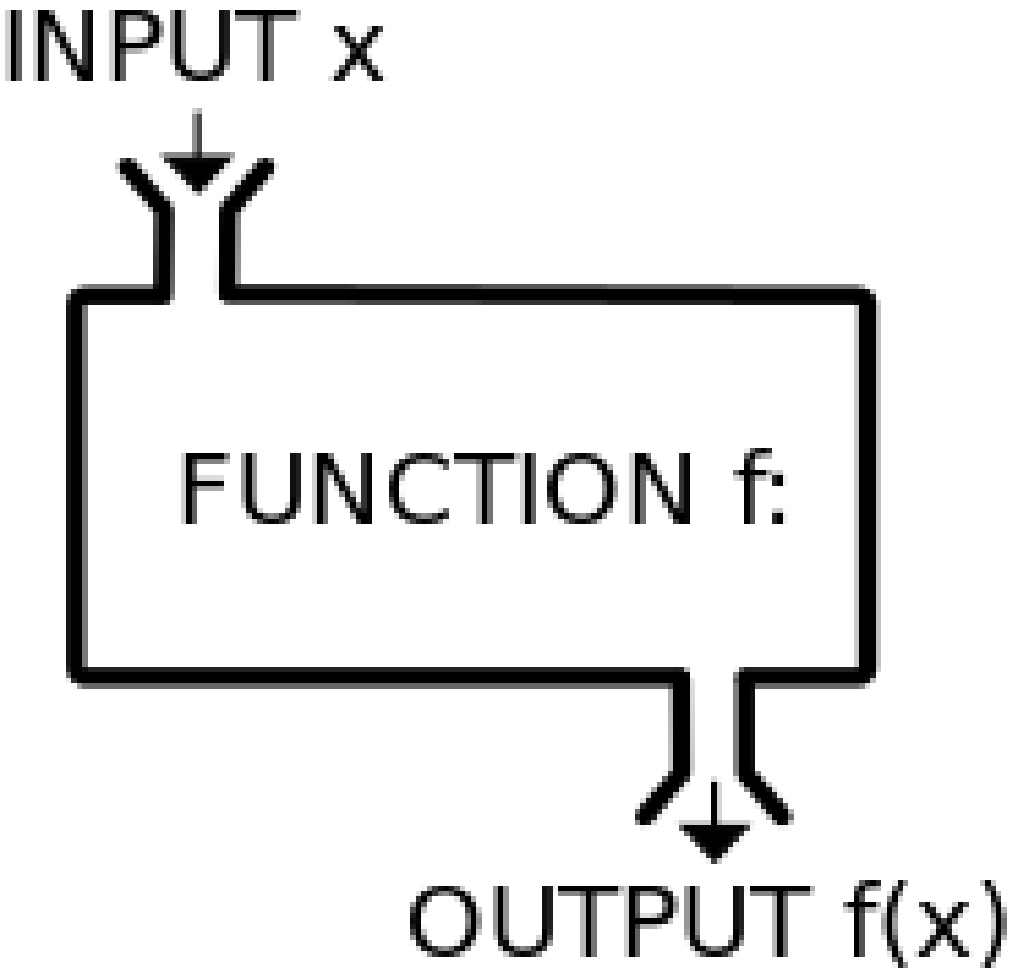
```
*Test> x = [1..]
*Test> length x
Interrupted.
*Test> take 3 x
[1,2,3]
*Test> take 3 (drop 5 x)
[6,7,8]
*Test>
```

WinGHCi

File Edit Actions Tools Help

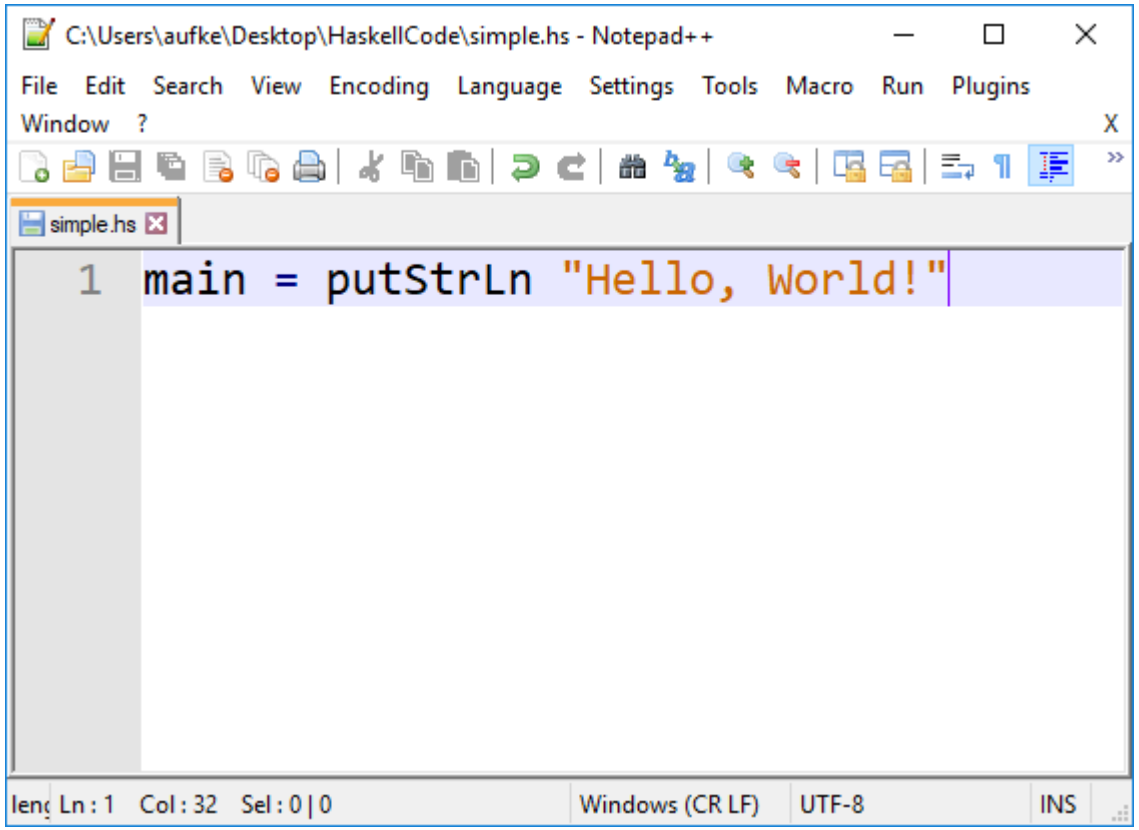
```
*Test> x = [1..]
*Test> zip "Hello" x
[('H',1),('e',2),('l',3),('l',4),('o',5)]
*Test>
```

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


INPUT x
FUNCTION f:
OUTPUT f(x)

Functions in Haskell

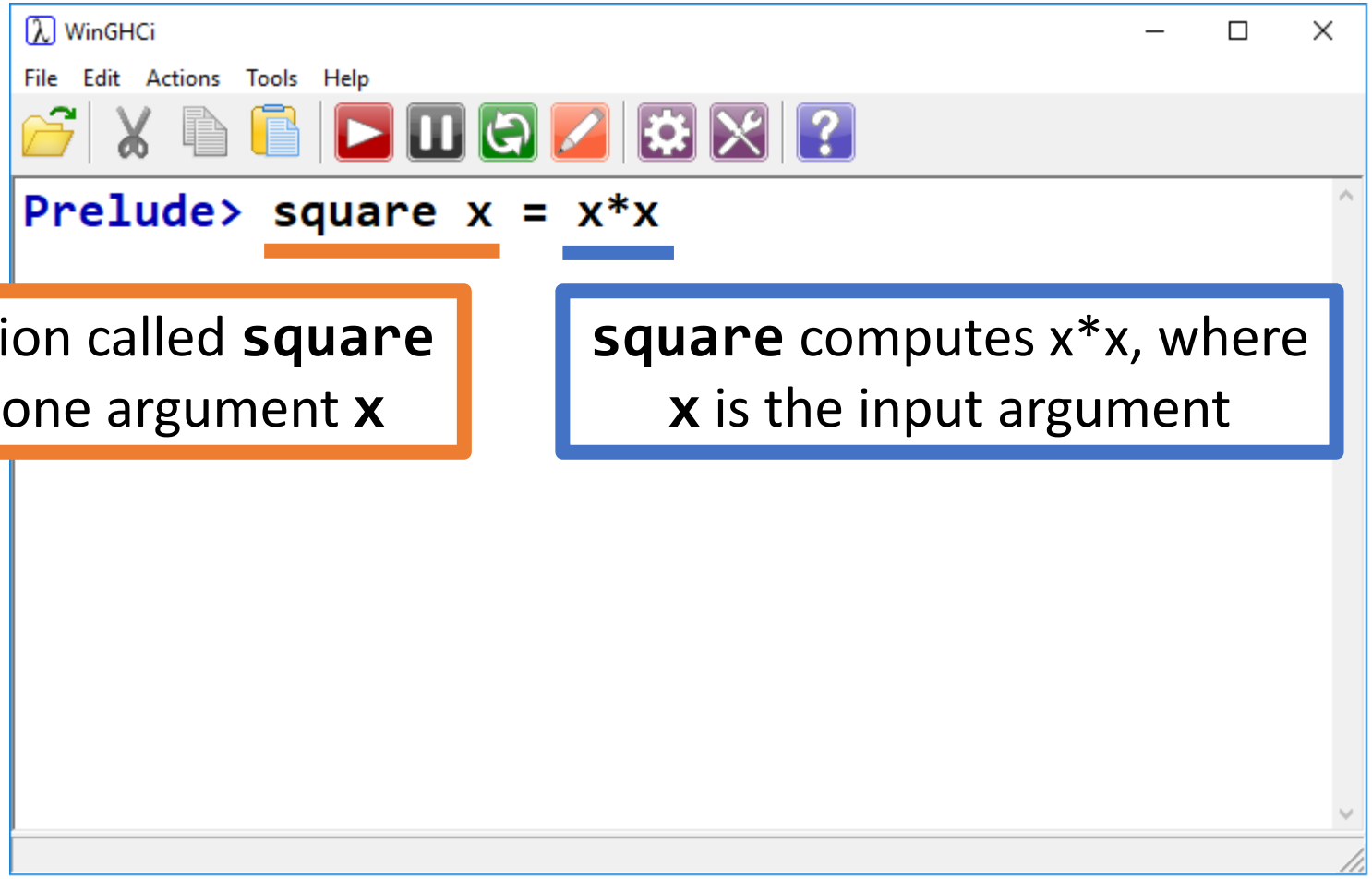
As expected of a pure functional language, functions are central in Haskell



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

Functions in Haskell

Let's start simple:

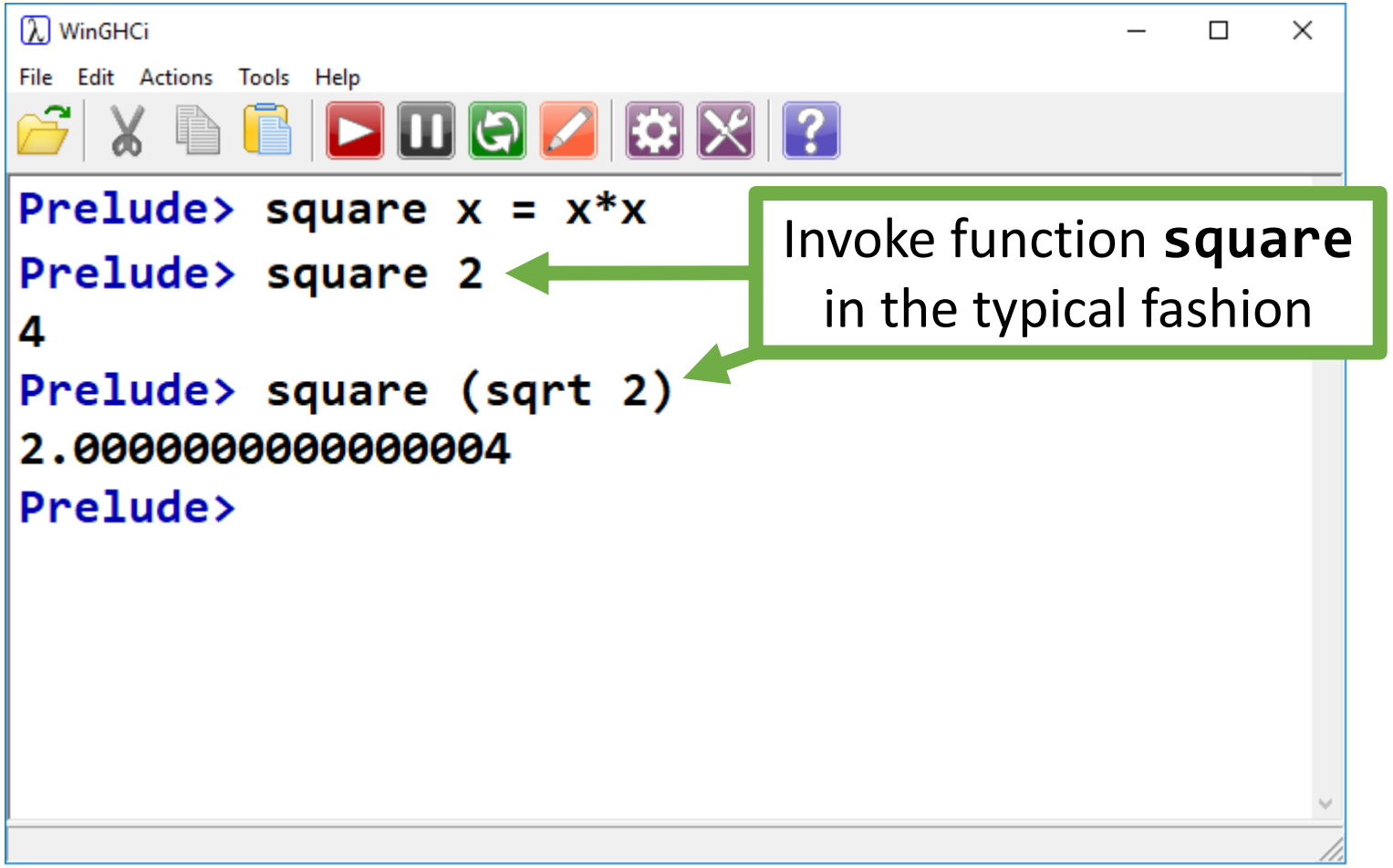


Define function called **square** that takes one argument **x**

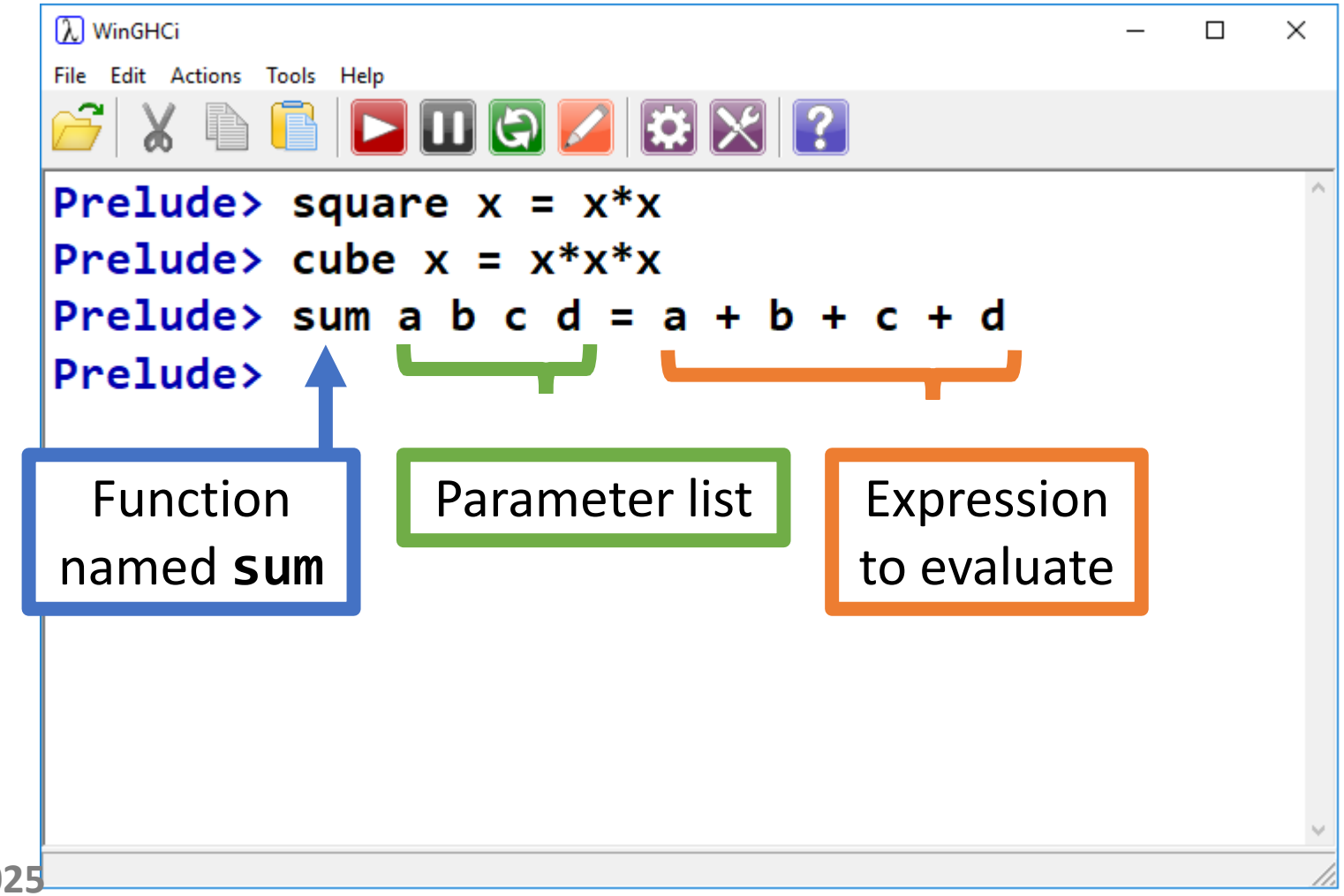
square computes x*x, where **x** is the input argument

Functions in Haskell

Let's start simple:



Functions in Haskell



Functions in Haskell

WinGHCi

File Edit Actions Tools Help

```
Prelude> square x = x*x
Prelude> cube x = x*x*x
Prelude> sum a b c d = a + b + c + d
Prelude> sum (square 2) (cube 2) 3 4
19
Prelude>
```

Passing in four args

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!

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

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` `case` `let-in`

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 Plu

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
```

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

```
1  module Test where
2
3   square x = x*x
4   cube x = x*x*x
5   sum x y = x + y
6
7   sign x =
8    if x < 0 then -1
9    else if x > 0 then 1
10   else 0
11
```

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

We can 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>
```

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
```

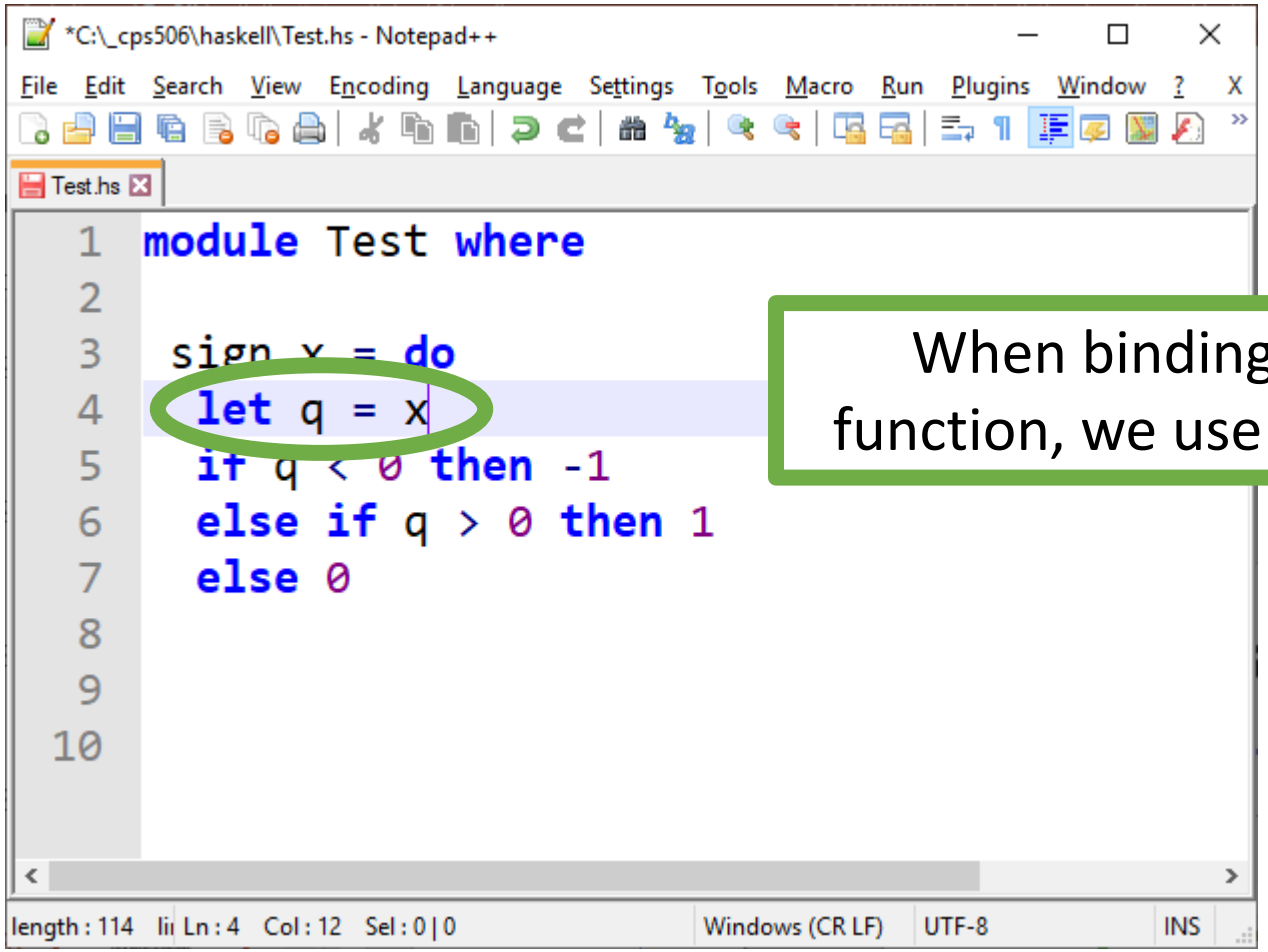
length : 95 lines : 9 | Ln : 8 Col : 2 Sel : 0 | 0 | Windows (CR LF) | UTF-8 | INS

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

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

Local Names in Functions



When binding a name inside a function, 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

```
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

- 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>
```

Return Multiple *Things*?

Lists/tuples to the rescue!

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

```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) )
```

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

`let/in` Expression

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

```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))
```

- 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))
```

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>
```

Case Expression

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

```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.

Case Expression

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

```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
```

WinGHCi

```
*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++

```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

```
*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 smalle
         | 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++

```haskell
module Test where

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

Classic list length finder

WinGHCi

```
*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

```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
```

- 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
```

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

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

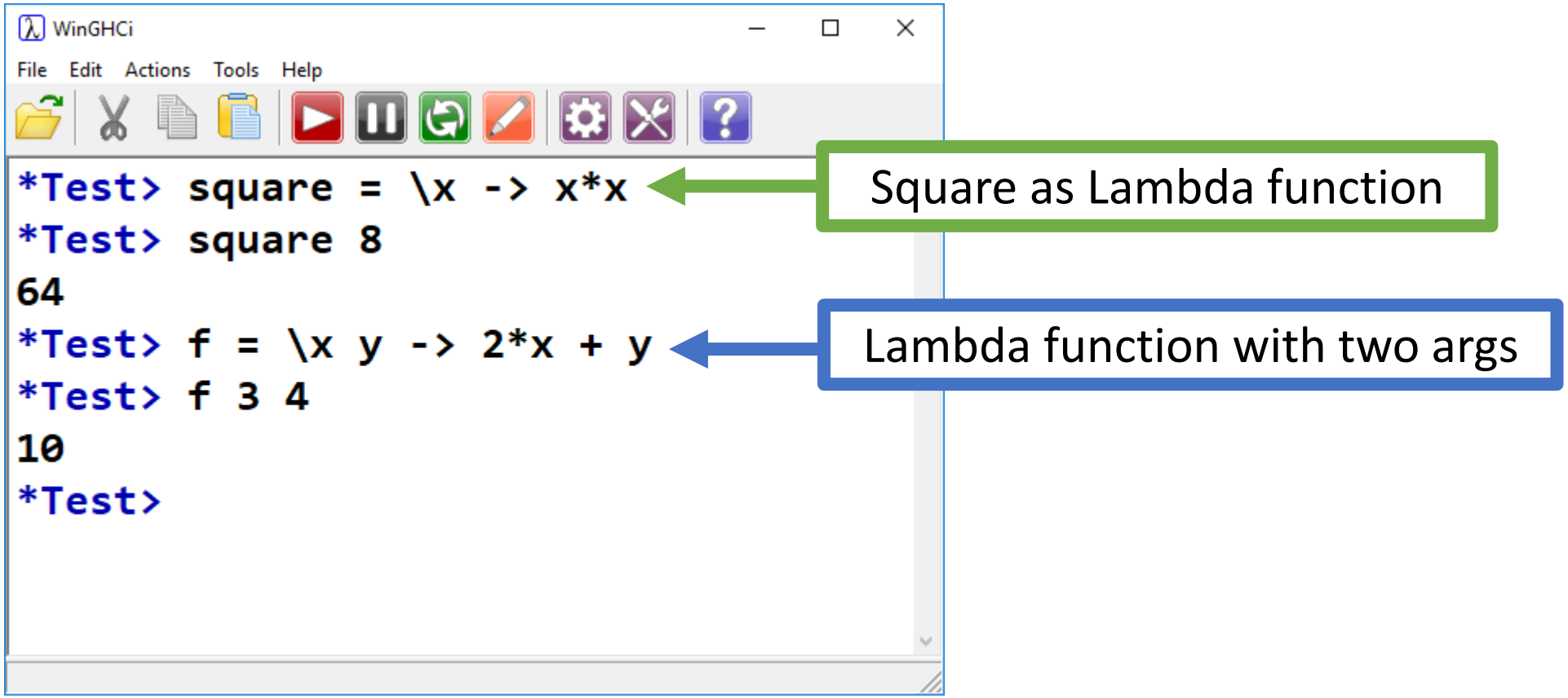
File Edit Actions Tools Help

```
*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.

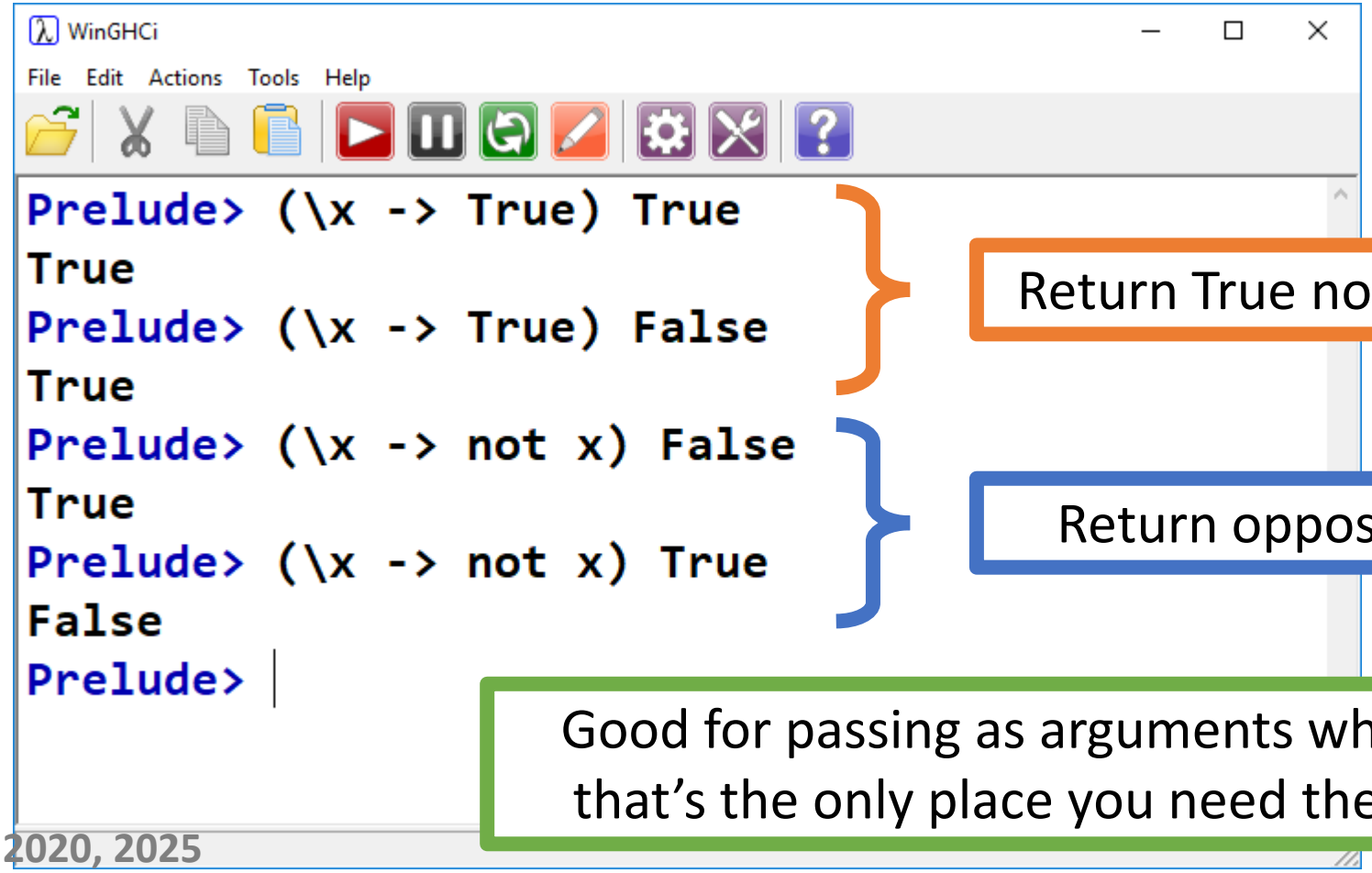
Lambda Functions

Like anonymous functions in Elixir:



Lambda Functions

They don't need names!

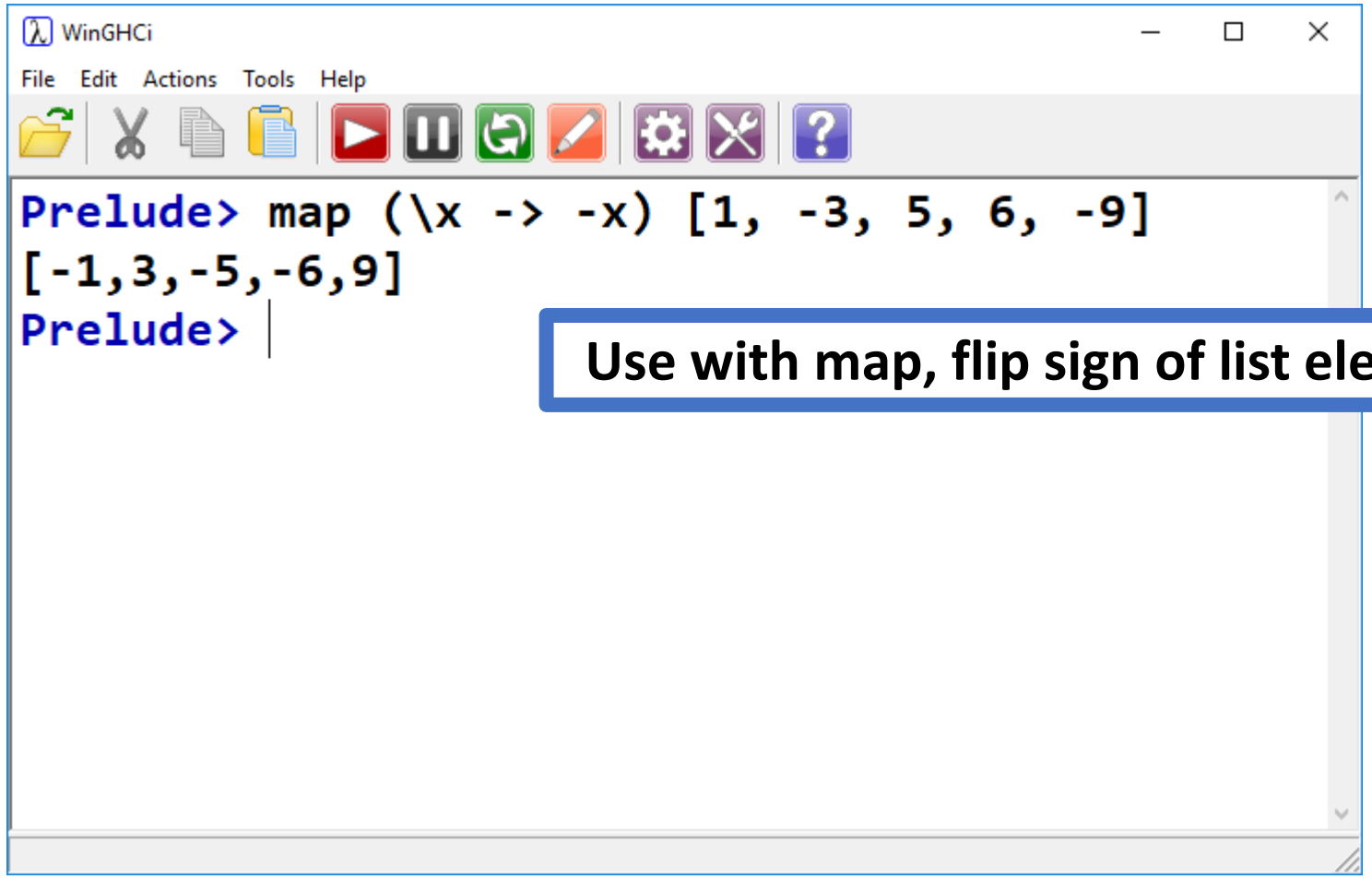


Return True no matter what

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

Haskell Tutorials/References:

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

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