C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 6: Haskell intro, Haskell basics

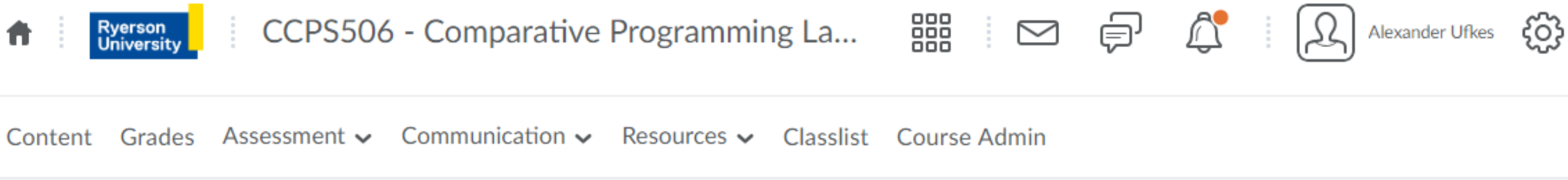
Ryerson University

Notice!

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

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

Course Administration



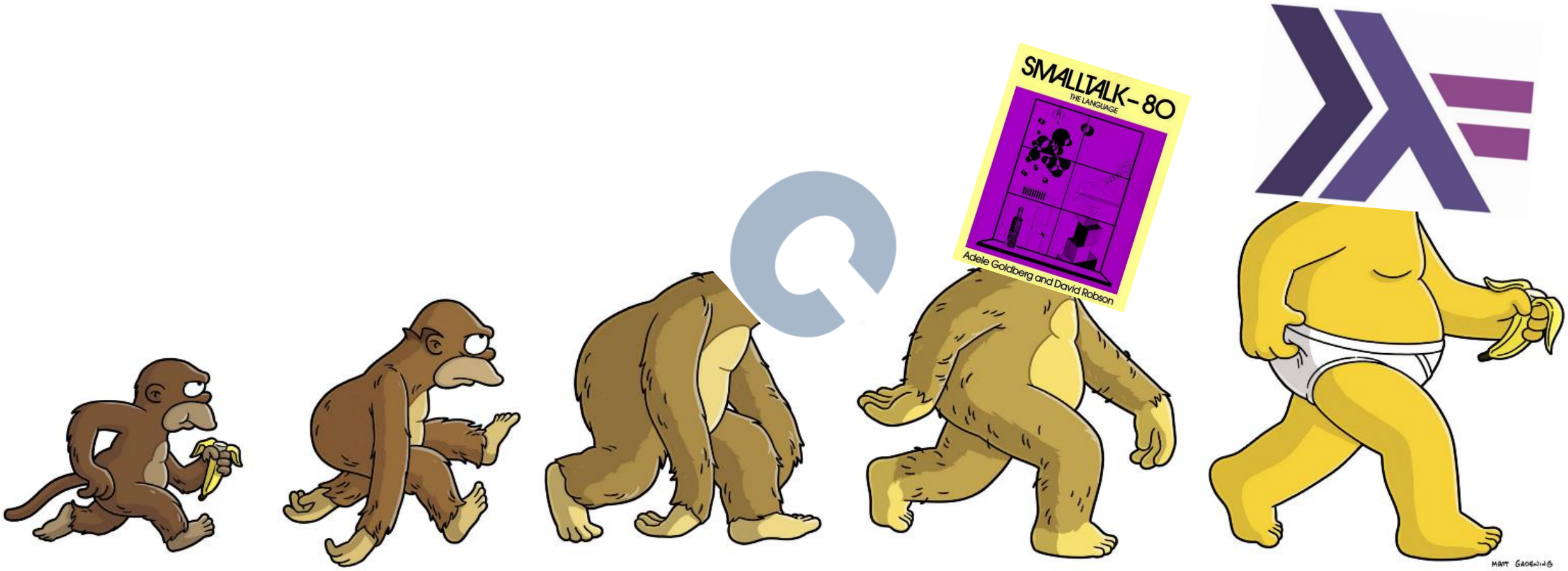
Two languages down, two to go!

Today

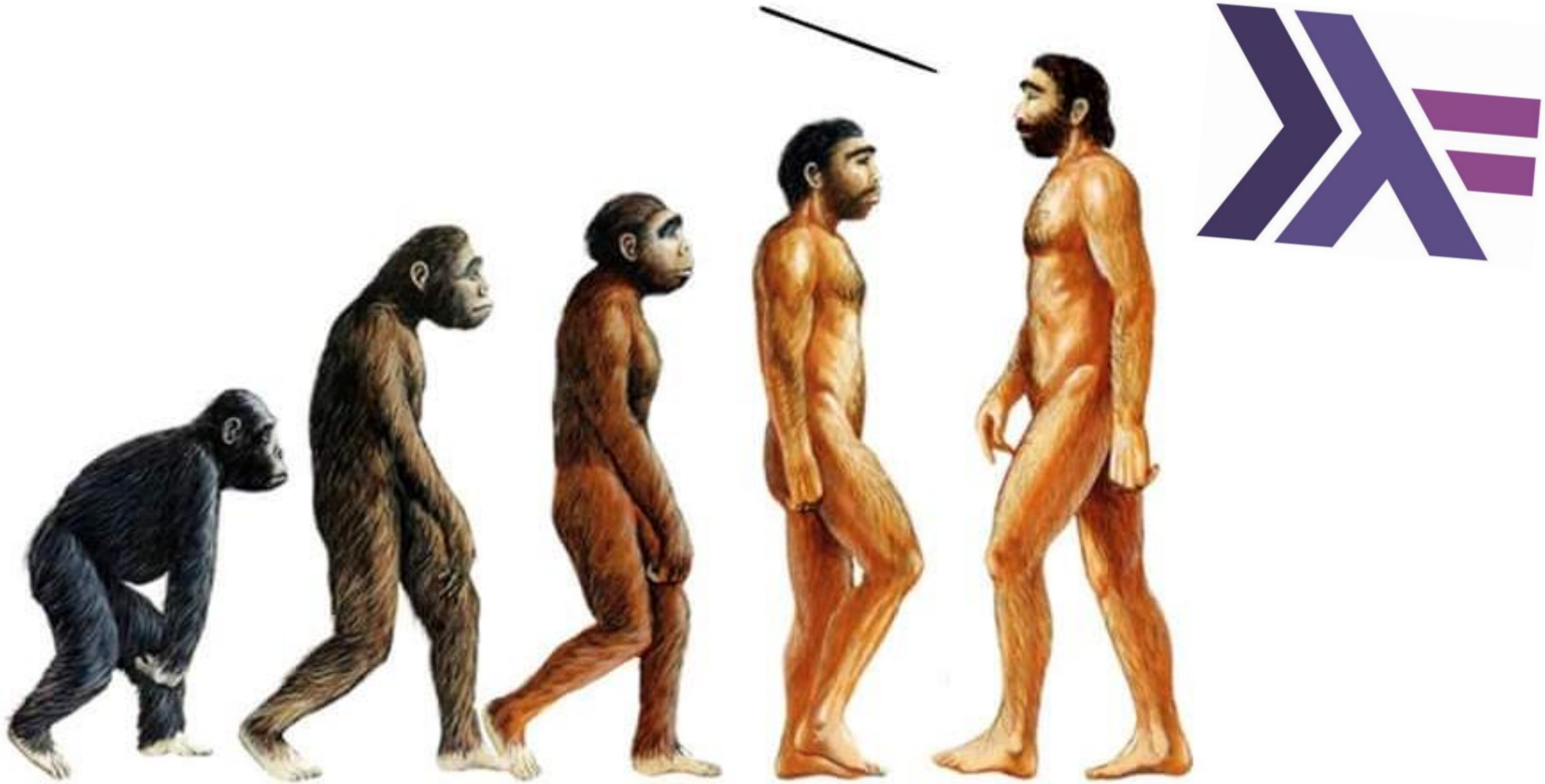
Intro to Haskell

- Pure functional
- Haskell basics
- Functions
- Control flow

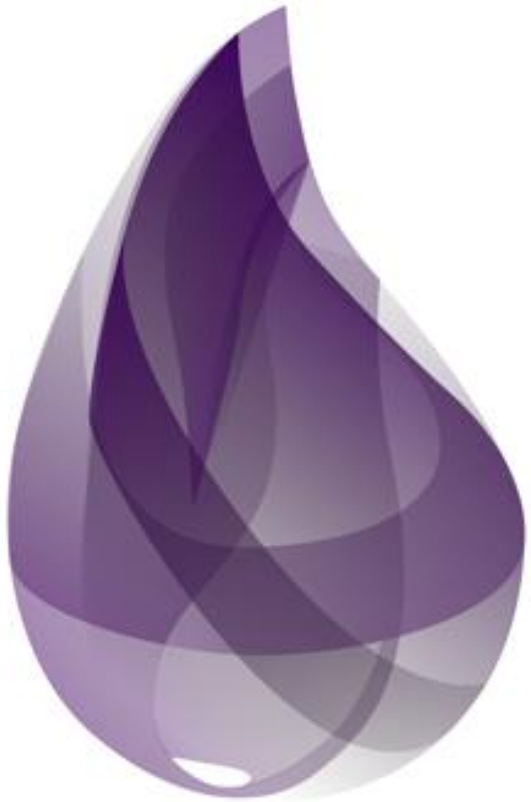
Functional Programming



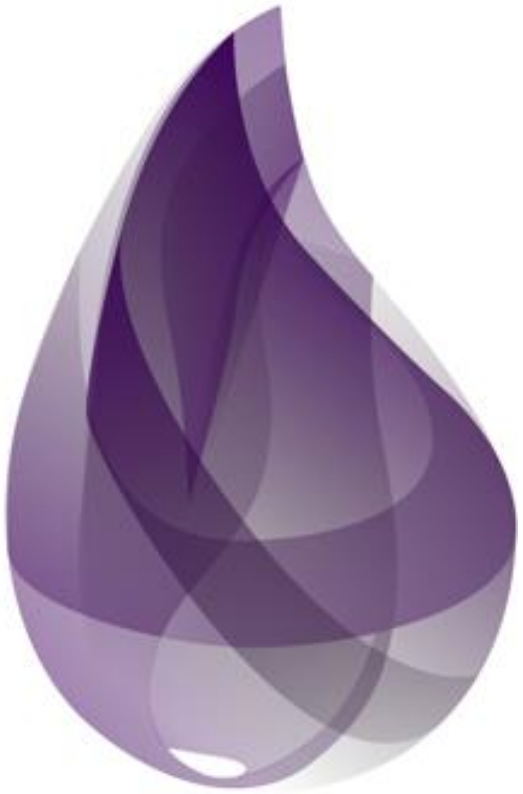
Go back. We f*cked up.



Functional Programming



Functional Programming



Higher-order functions:

- Can return functions or accept them as arguments.

First class functions:

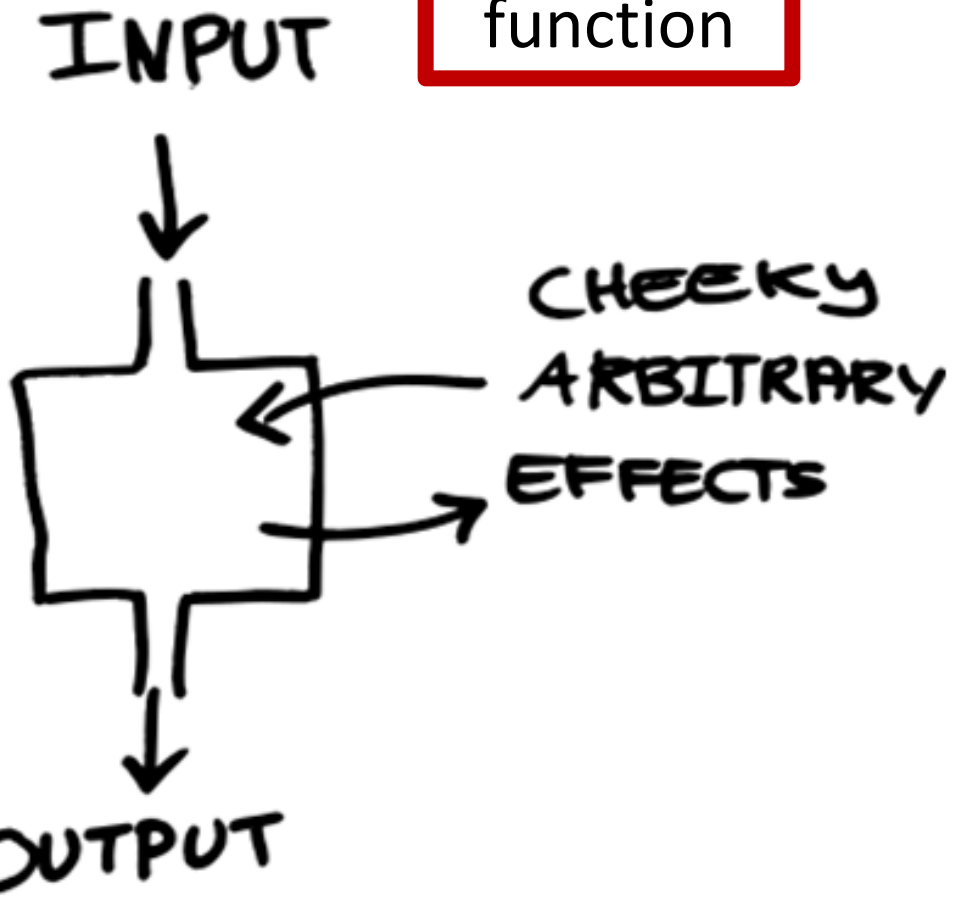
- Can be passed as arguments, returned as values.
- Think of them as ***values***, just like integers or floats

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

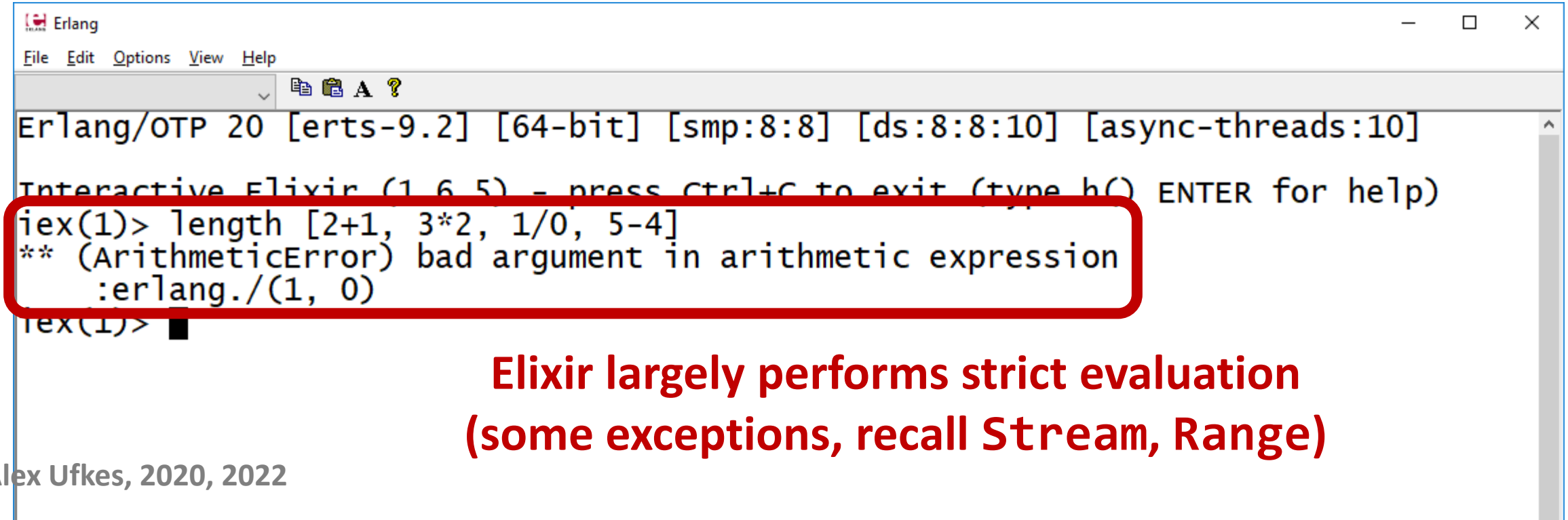
Impure function



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.

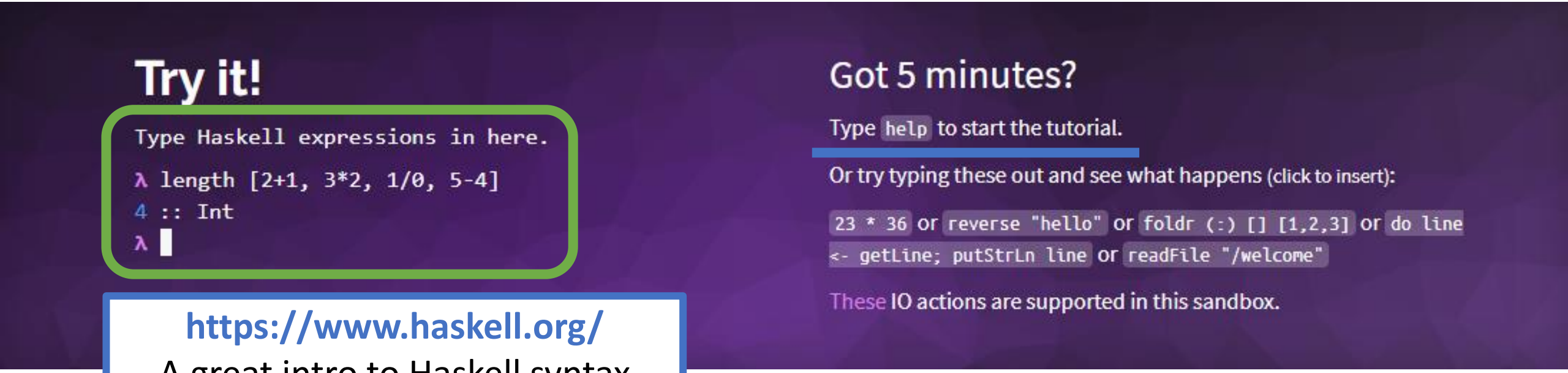


Elixir largely performs strict evaluation
(some exceptions, recall `Stream`, `Range`)

Functional Programming

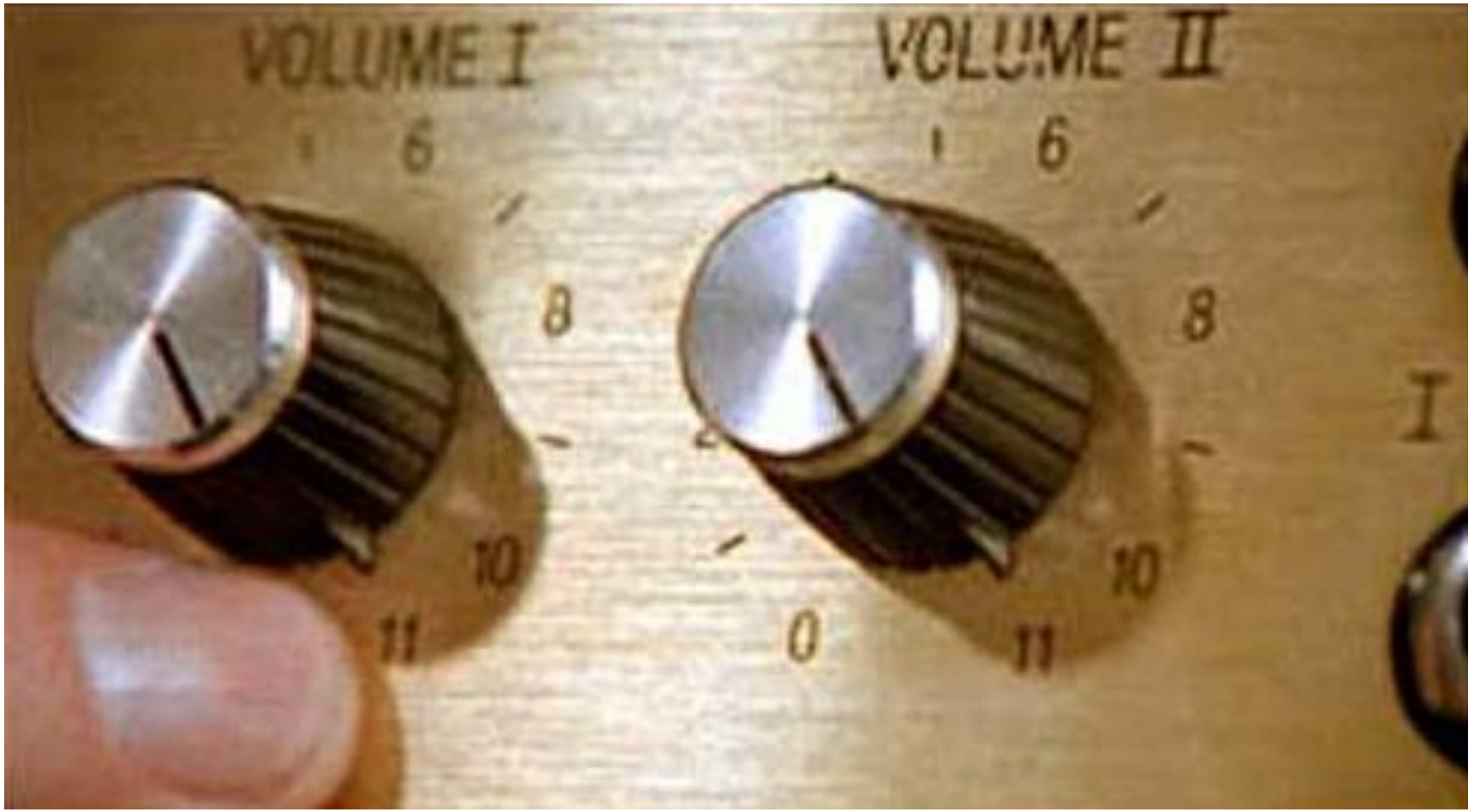
Strict (eager) VS. non-strict (lazy) evaluation:

- 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
- 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
- Control constructs written as functions
- Easy to fuse chains of functions together
- 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.

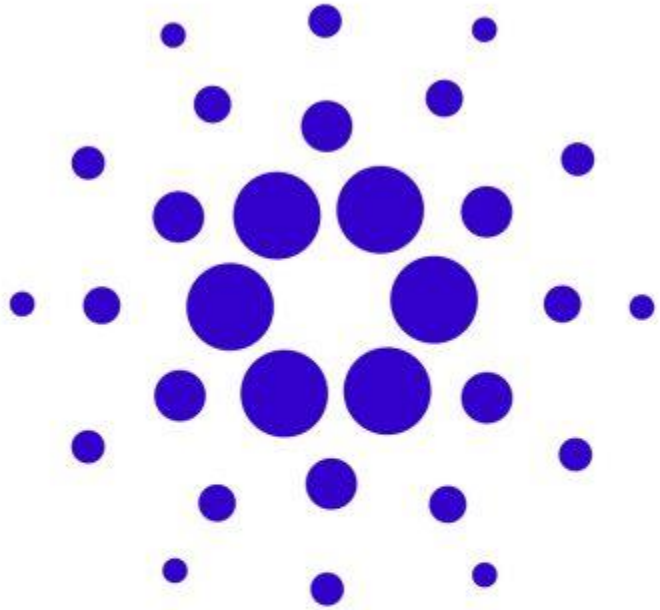
- Alcatel-Lucent

  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:

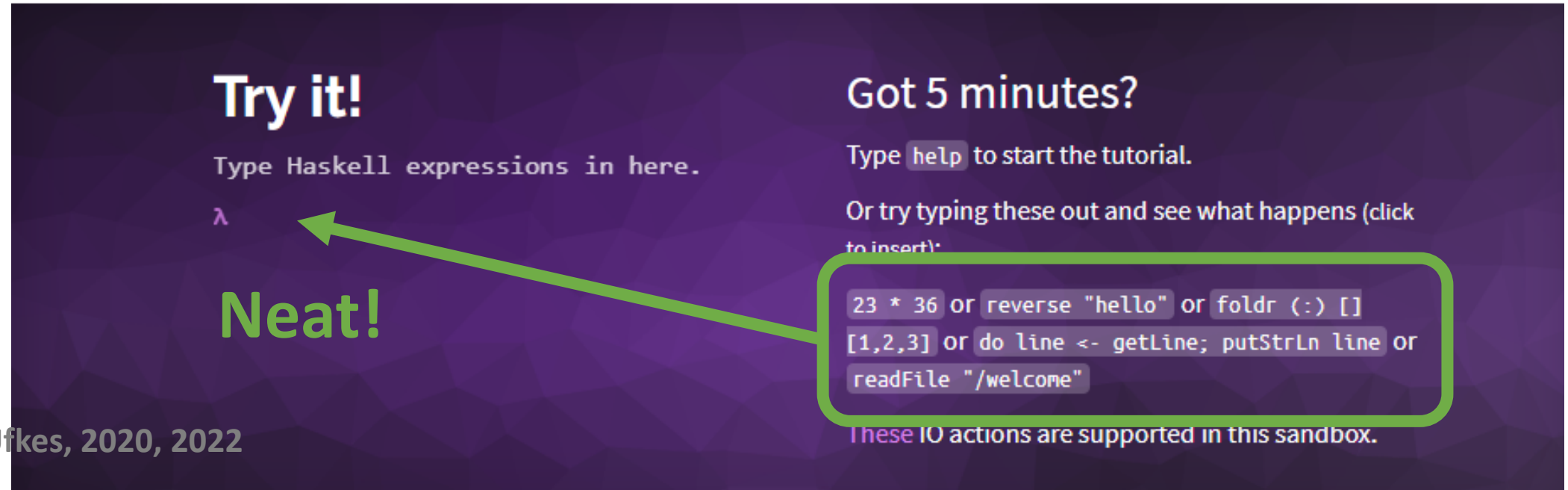
https://www.haskell.org/documentation/

Haskell

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]
```



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.

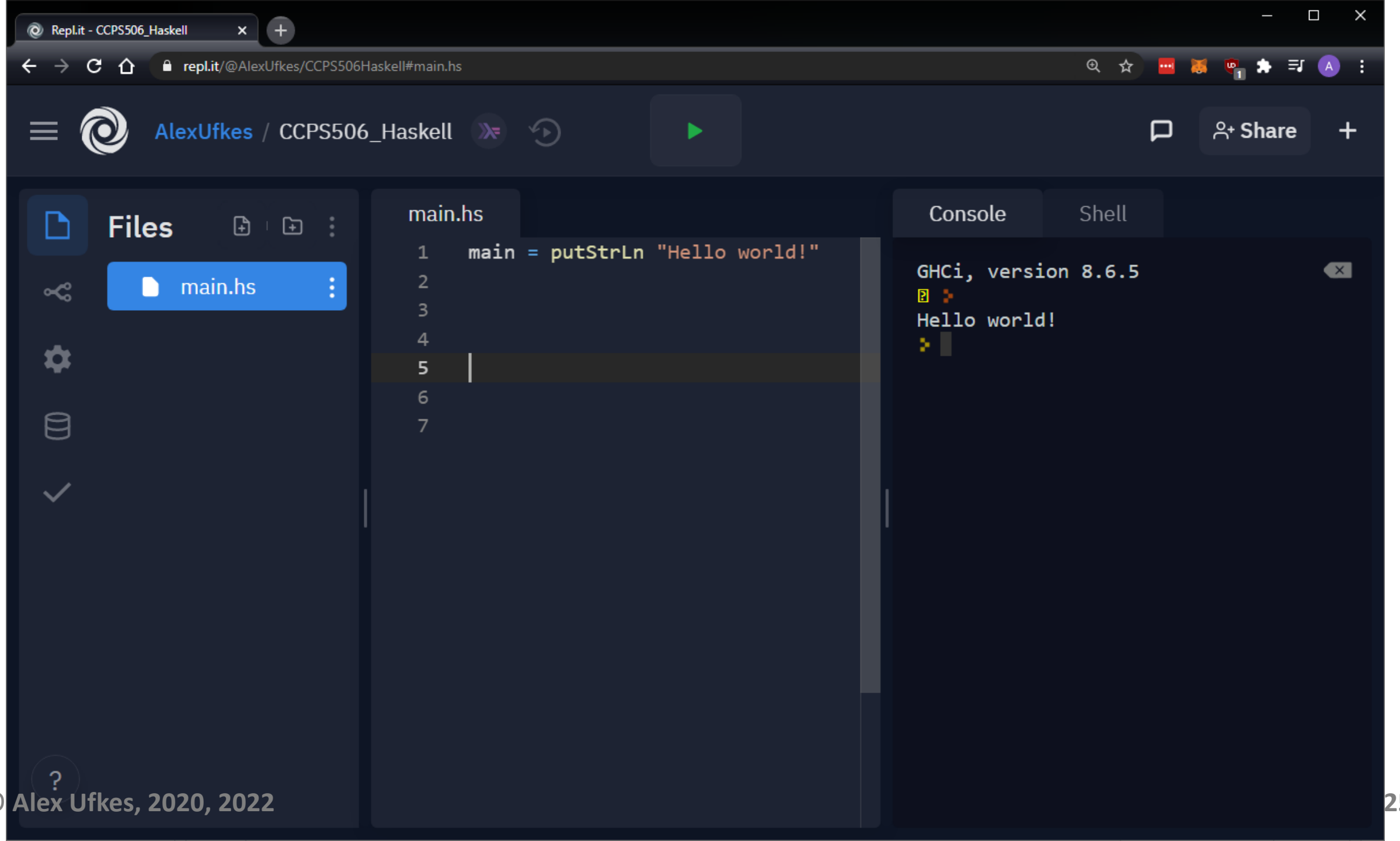
What you get

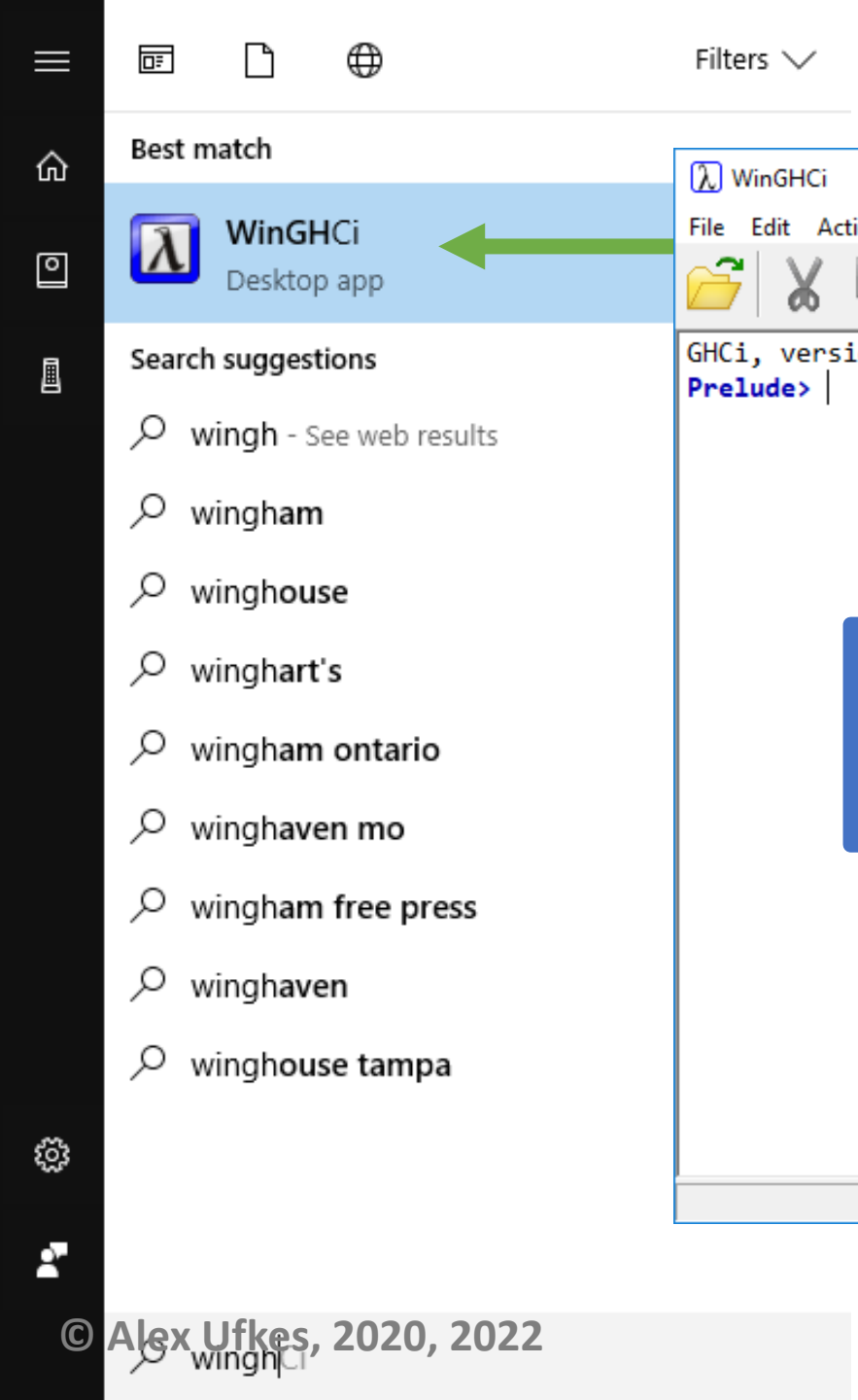
- The Glasgow Haskell Compiler
- The Cabal build system, which can install new packages, and by default fetches from Hackage, the central Haskell package repository.
- the Stack tool for developing projects
- Support for profiling and code coverage analysis
- 35 core & widely-used packages

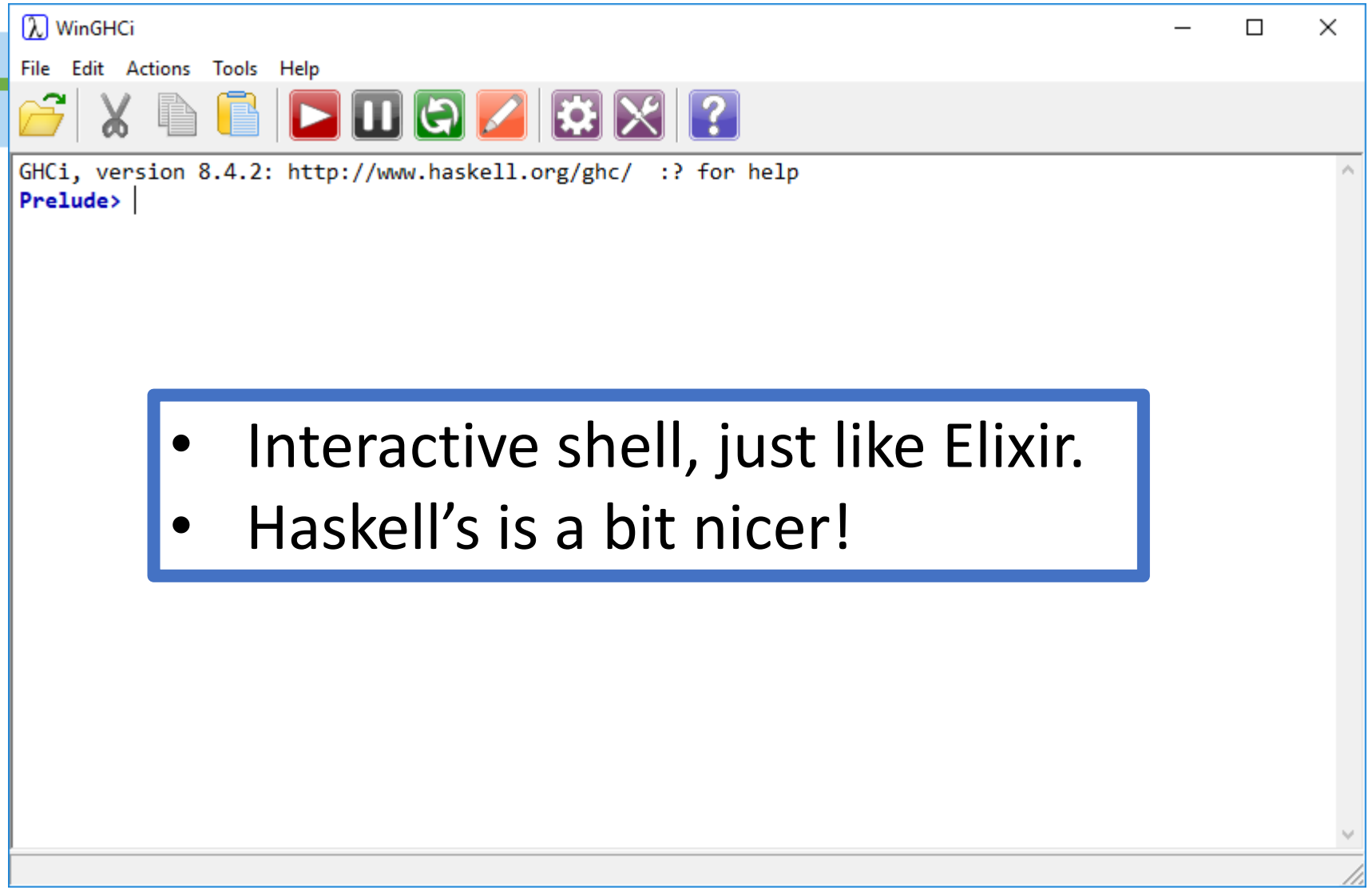
How to get it

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

- Linux
- OS X
- Windows

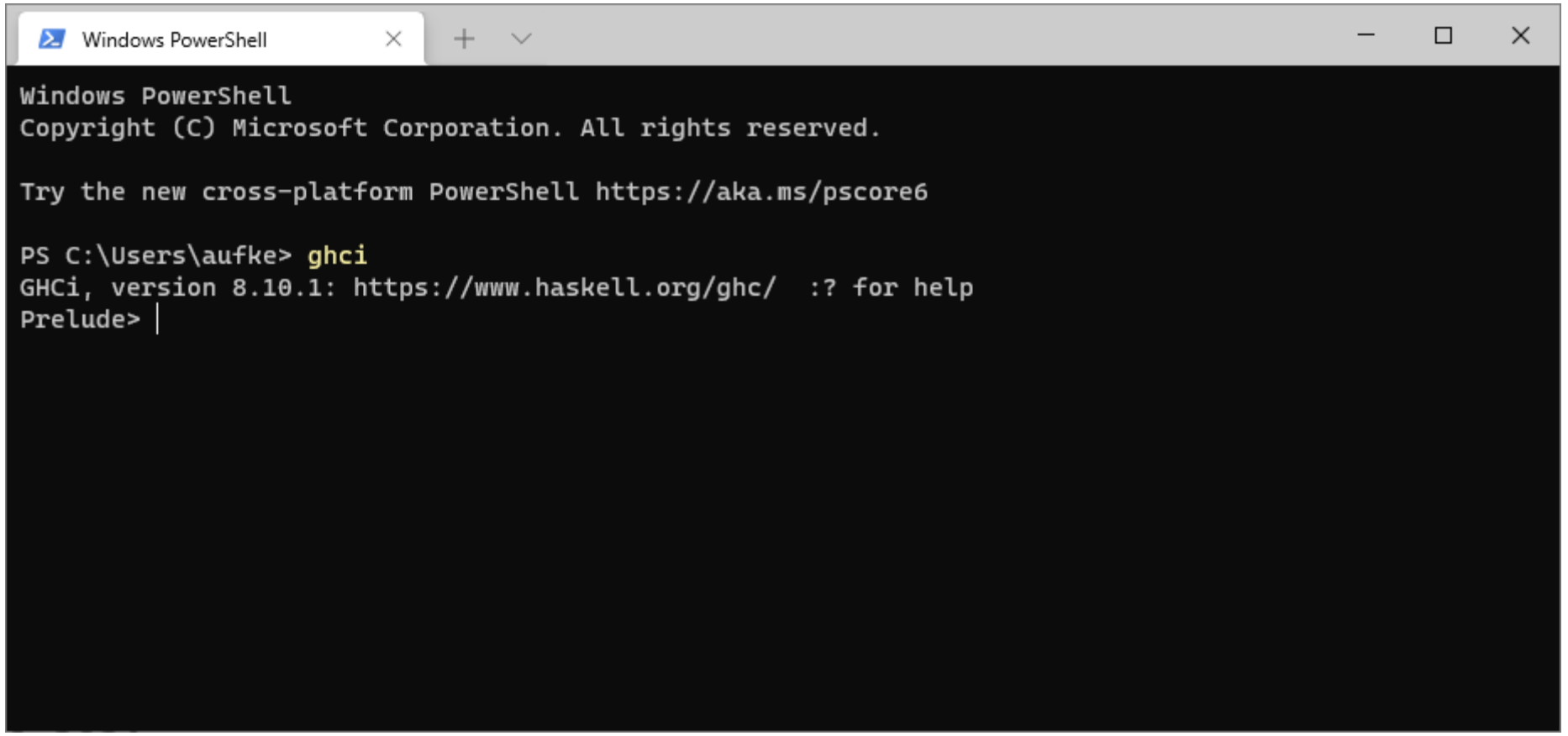




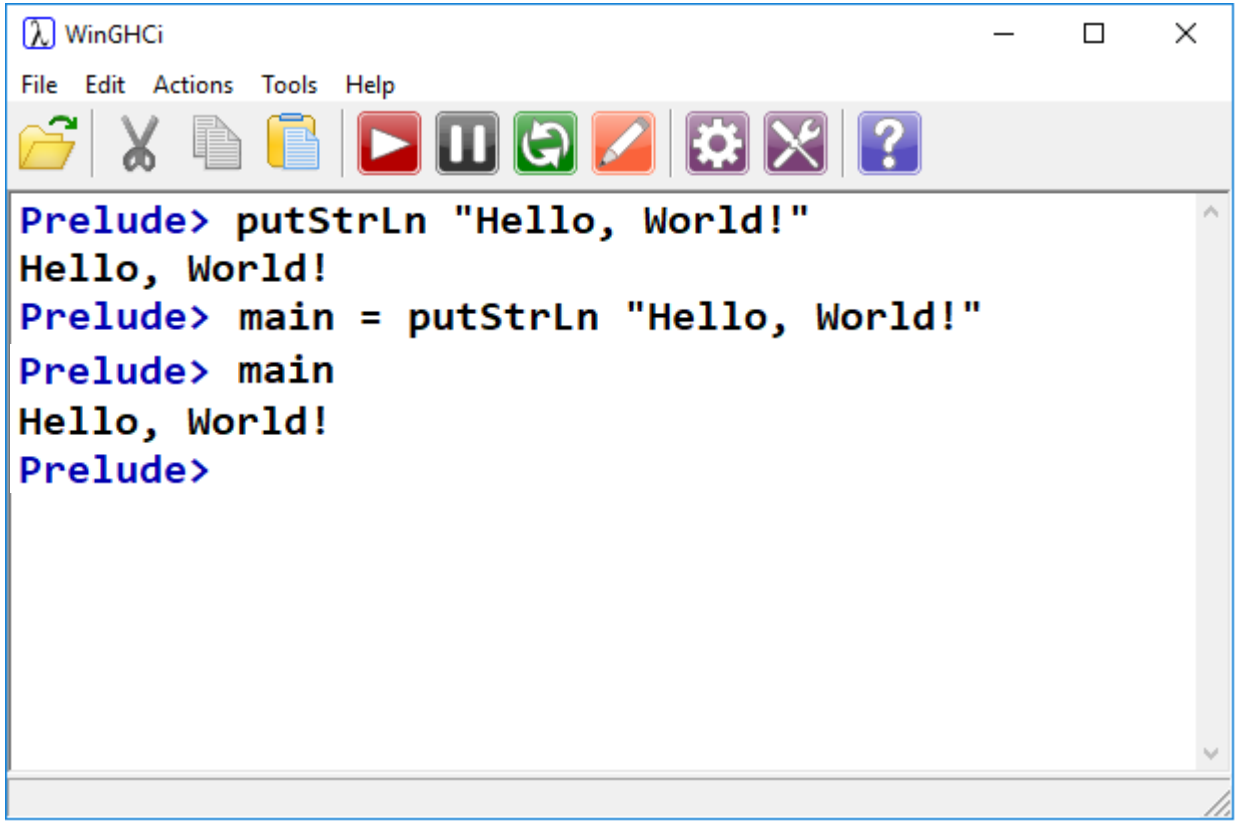


- Interactive shell, just like Elixir.
- Haskell's is a bit nicer!

- Apparently WinGHCi doesn't exist anymore
- Isn't included in newer versions of Haskell.
- No matter, you can get a GHCi shell in a regular terminal:

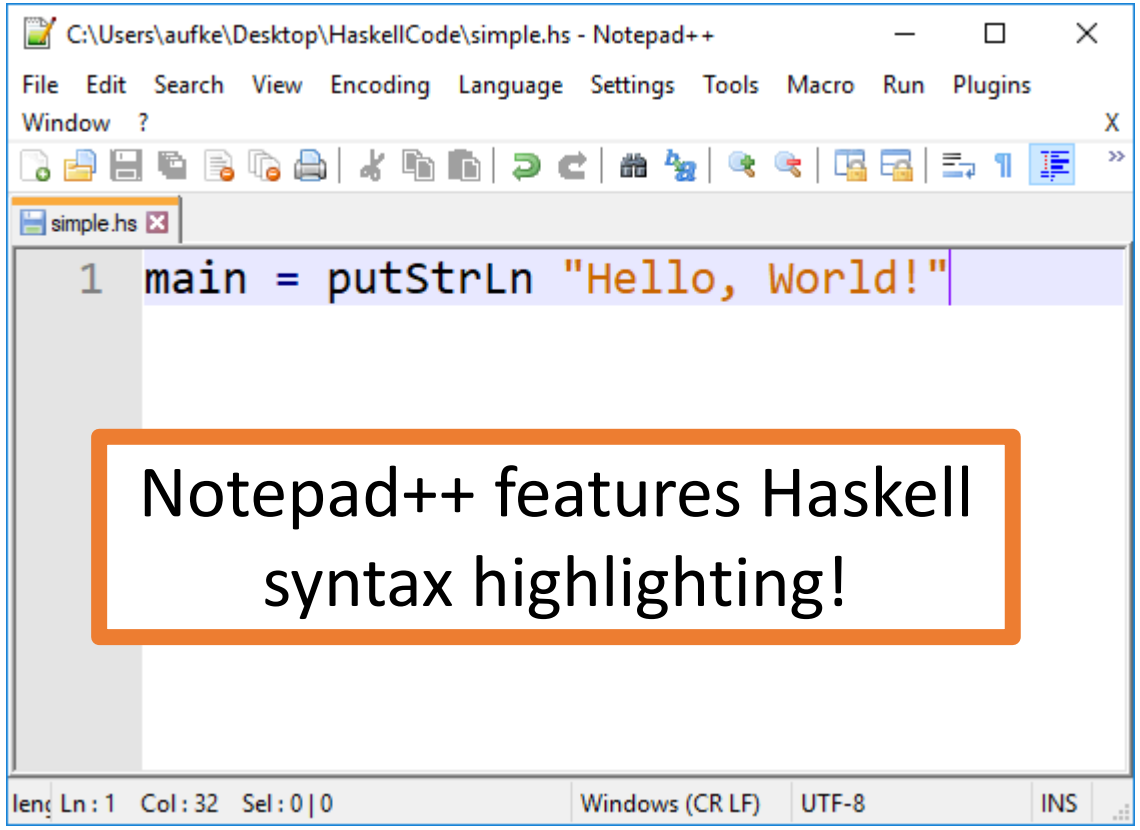


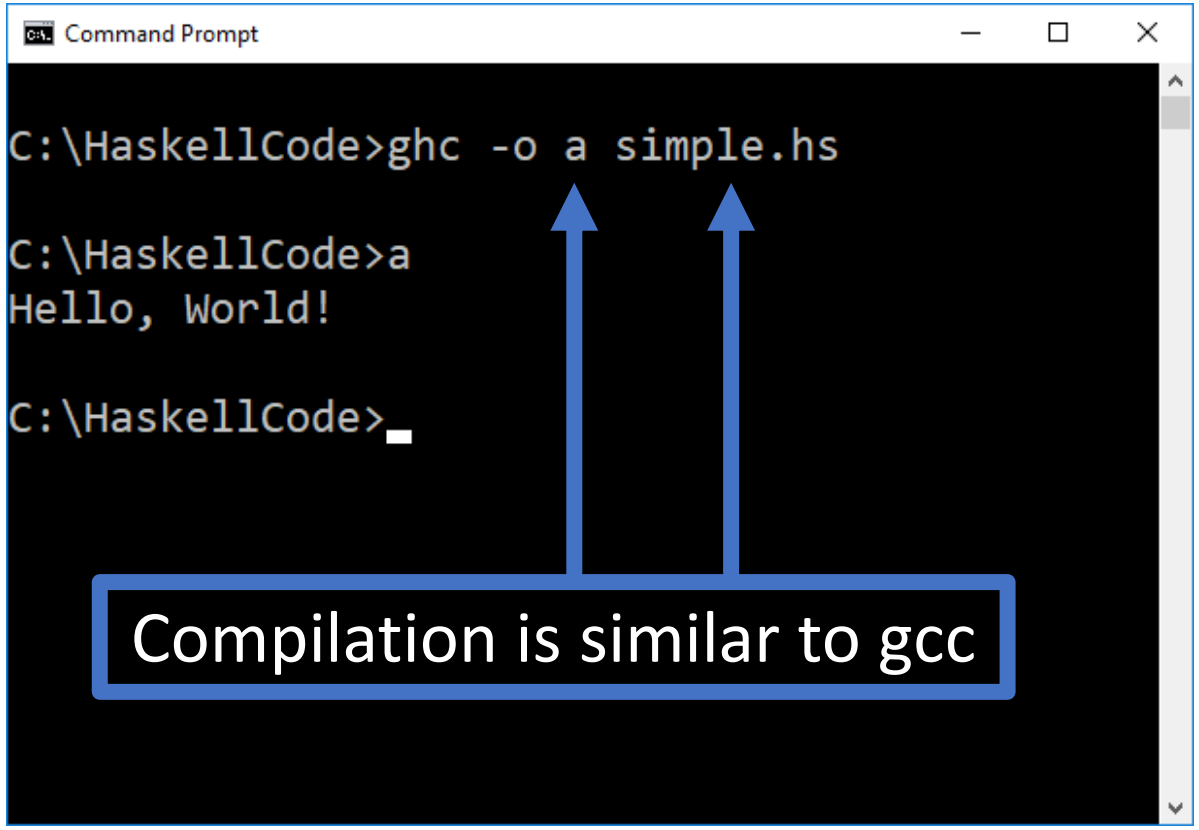
Hello, World!

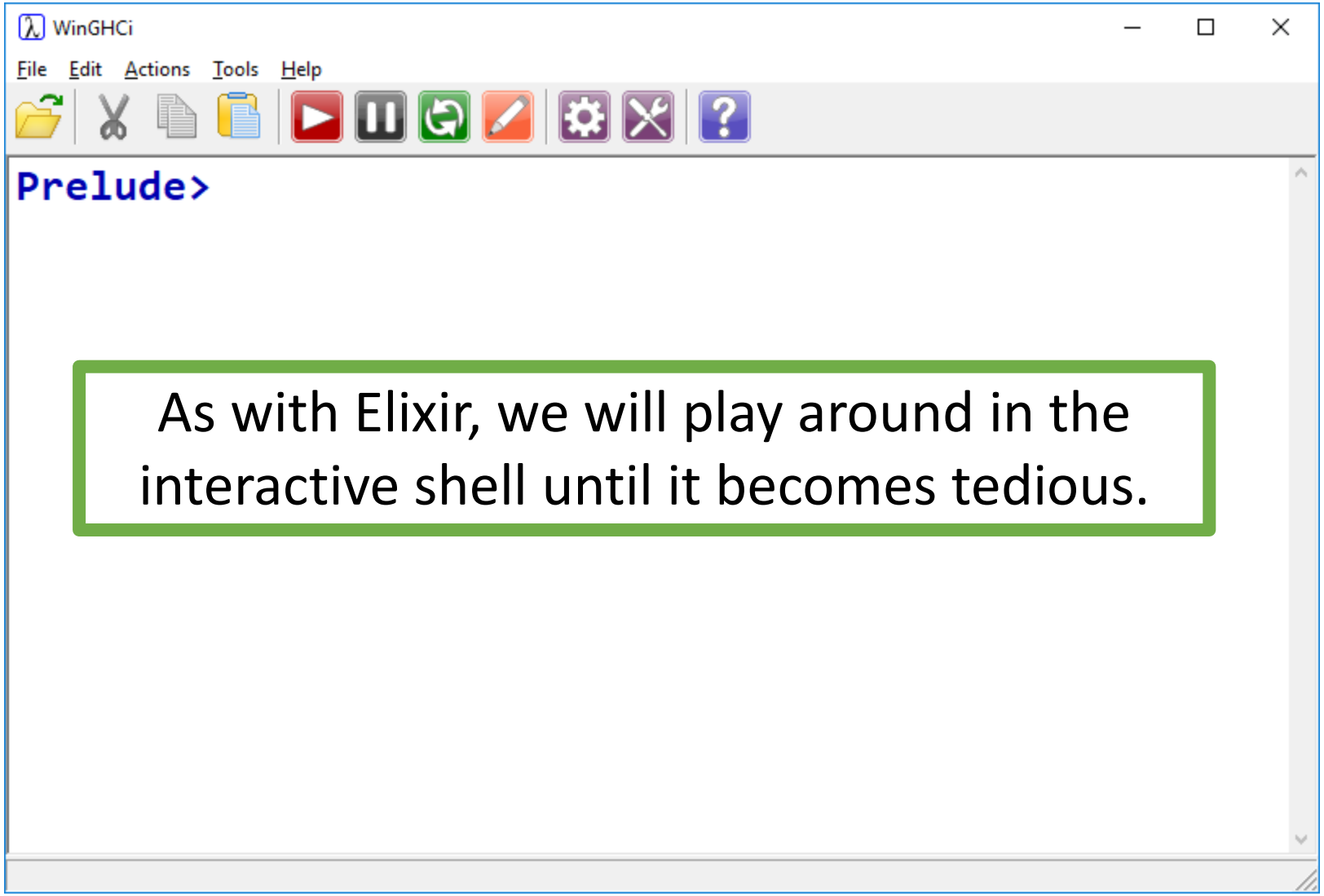


- Define a main function.
- main() is the entry point of a Haskell program
- Just like C or Java

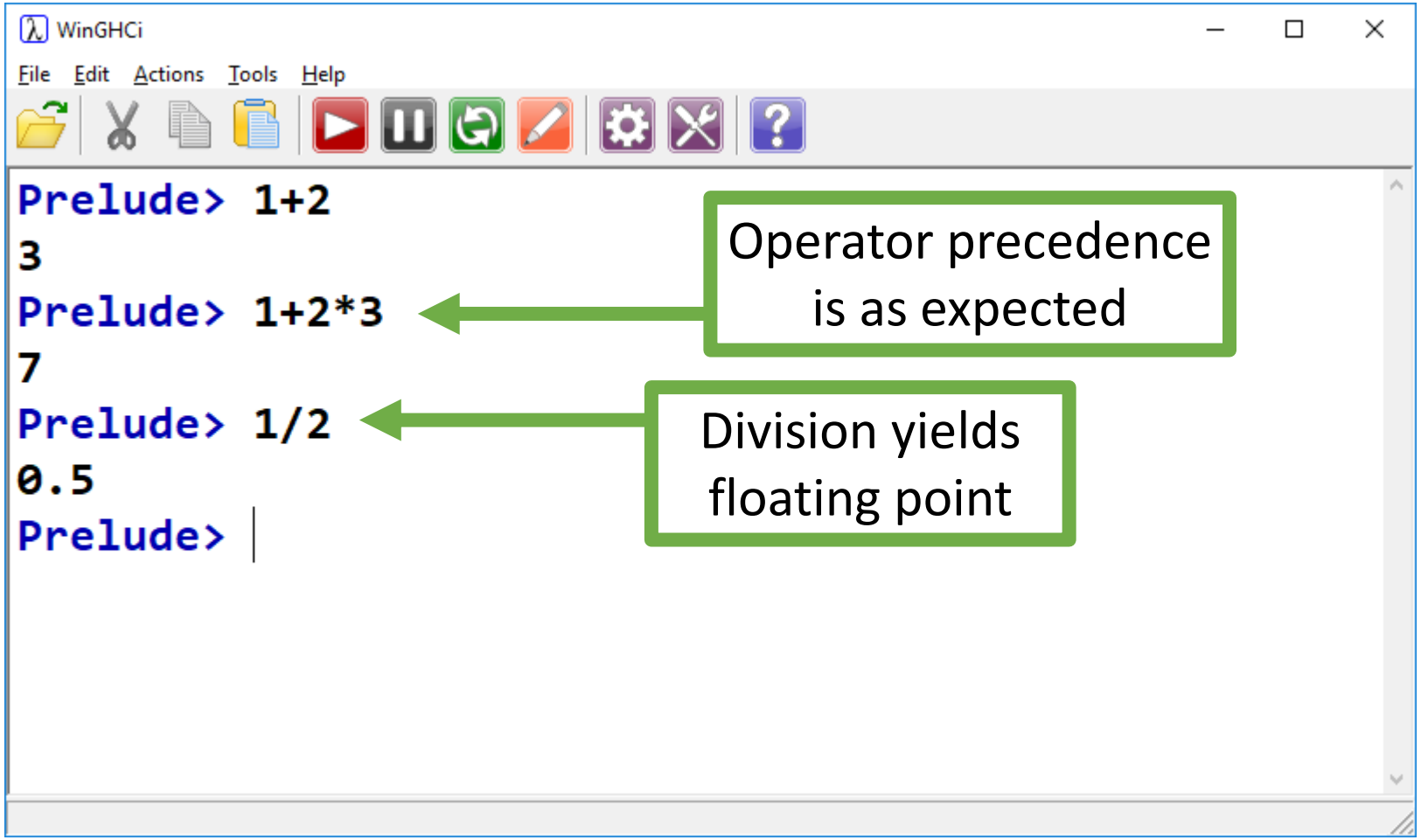
Compiling Haskell



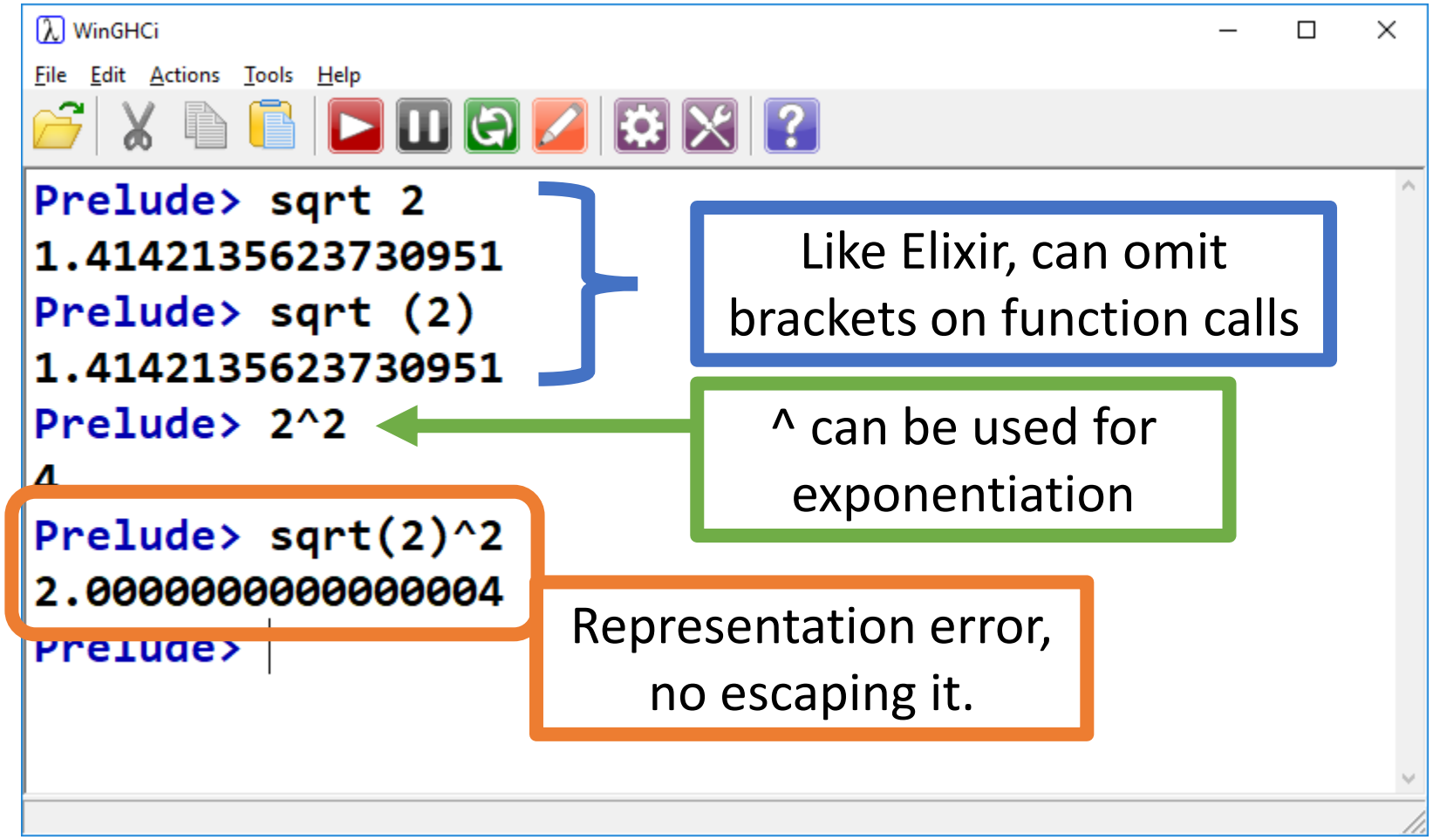




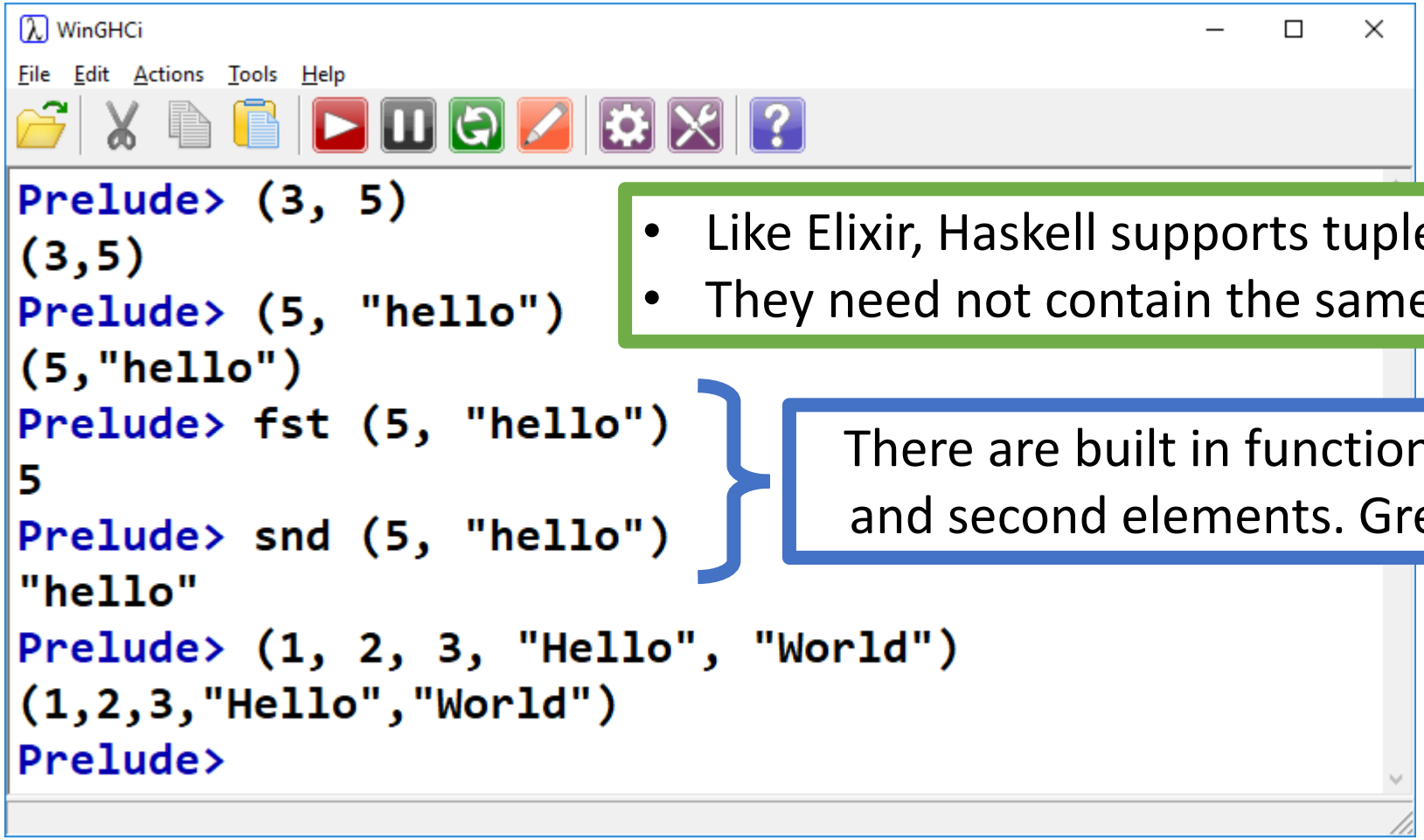
Literals & Arithmetic



Literals & Arithmetic

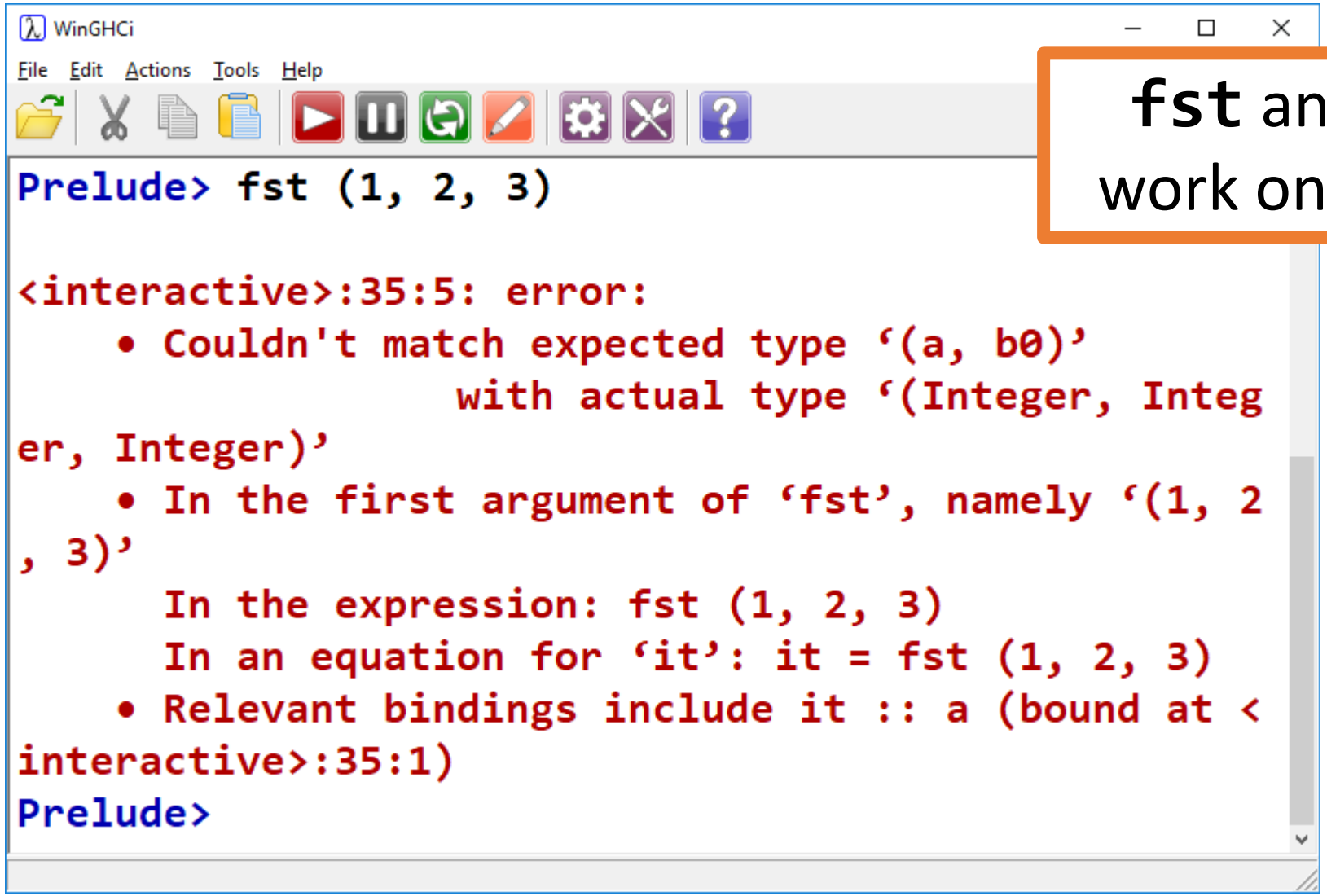


Tuples



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

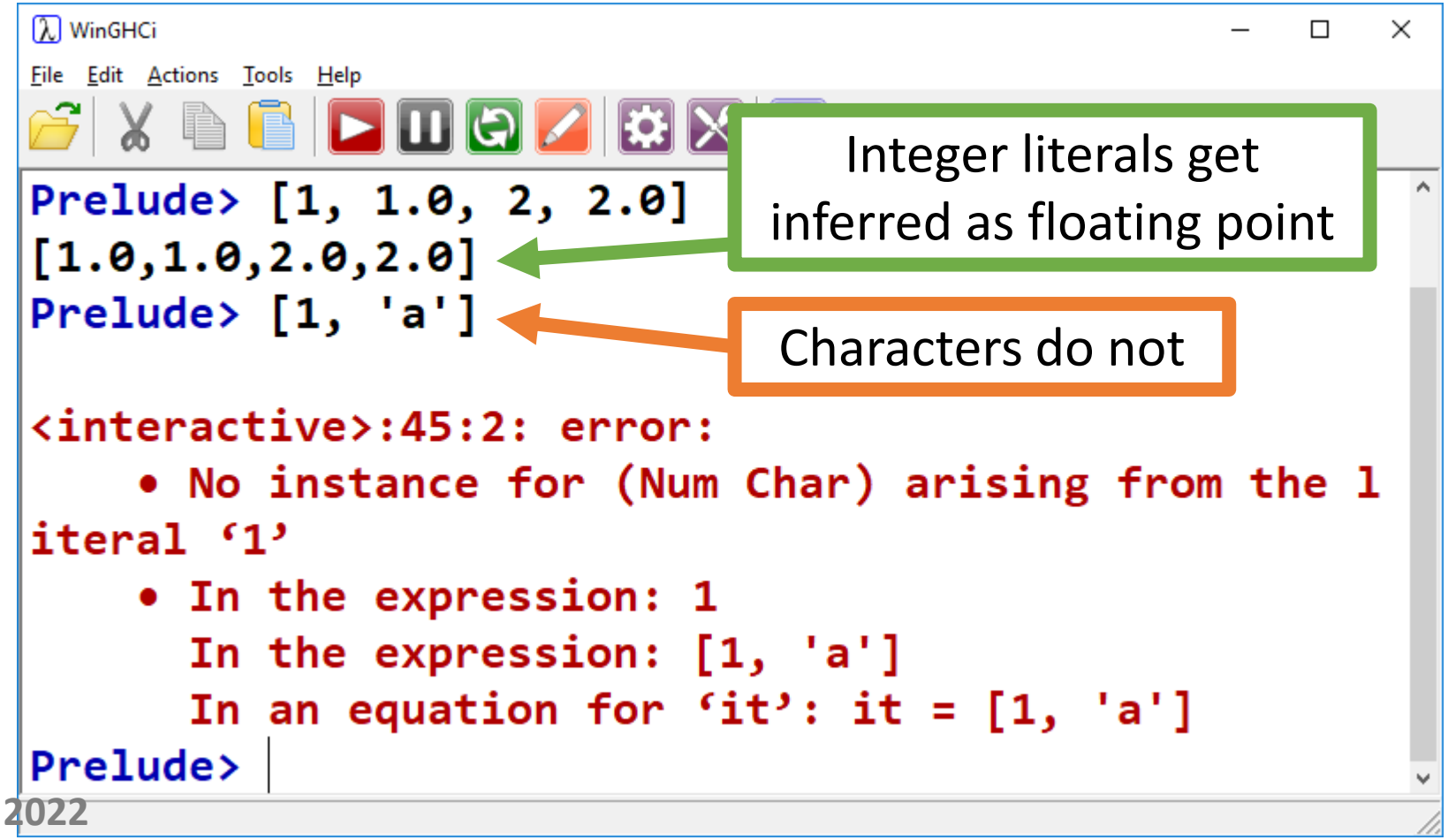
There are built in functions for accessing first and second elements. Great for coordinates.



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

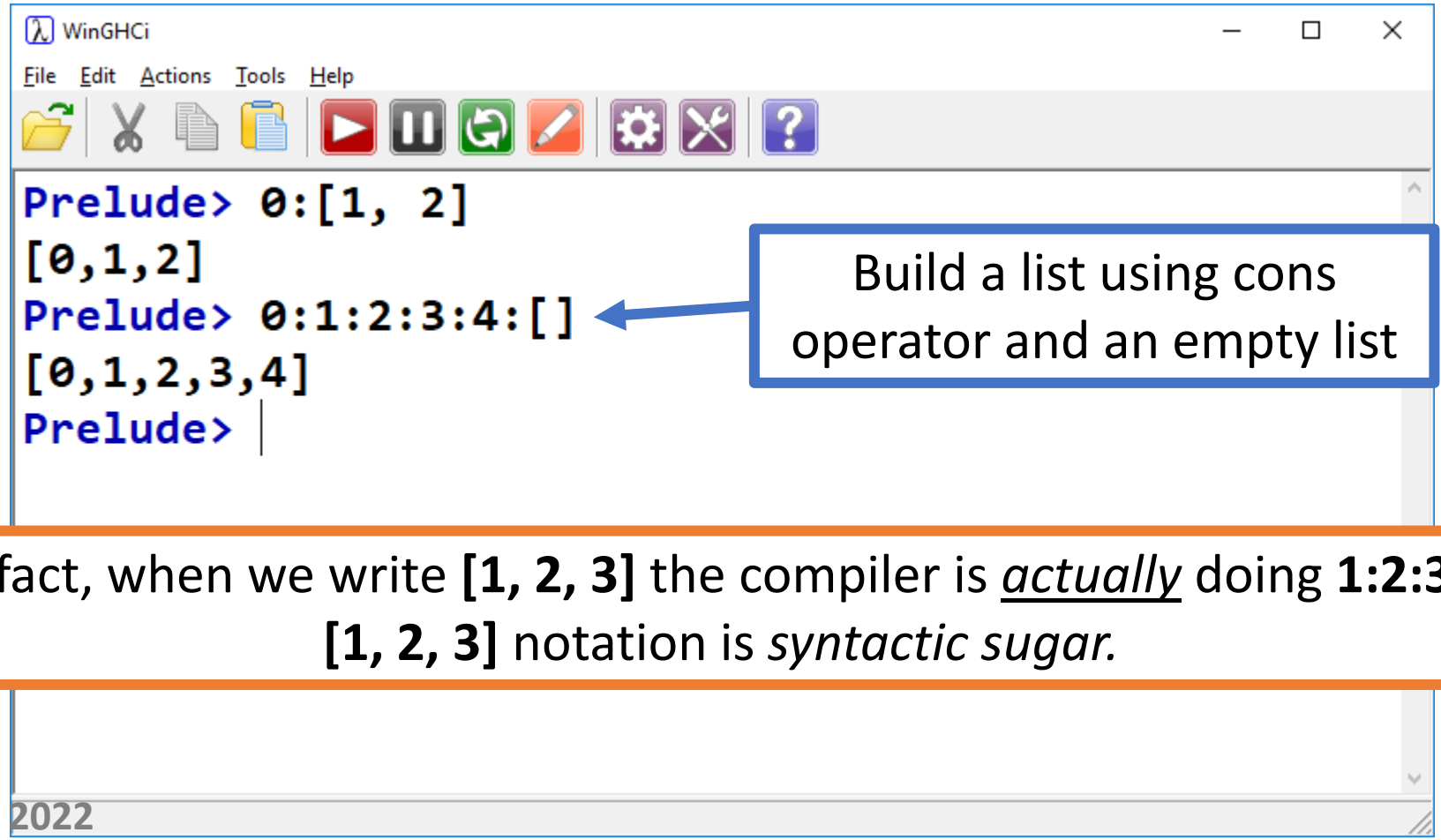
Lists

Must be ***homogeneous***:



Lists

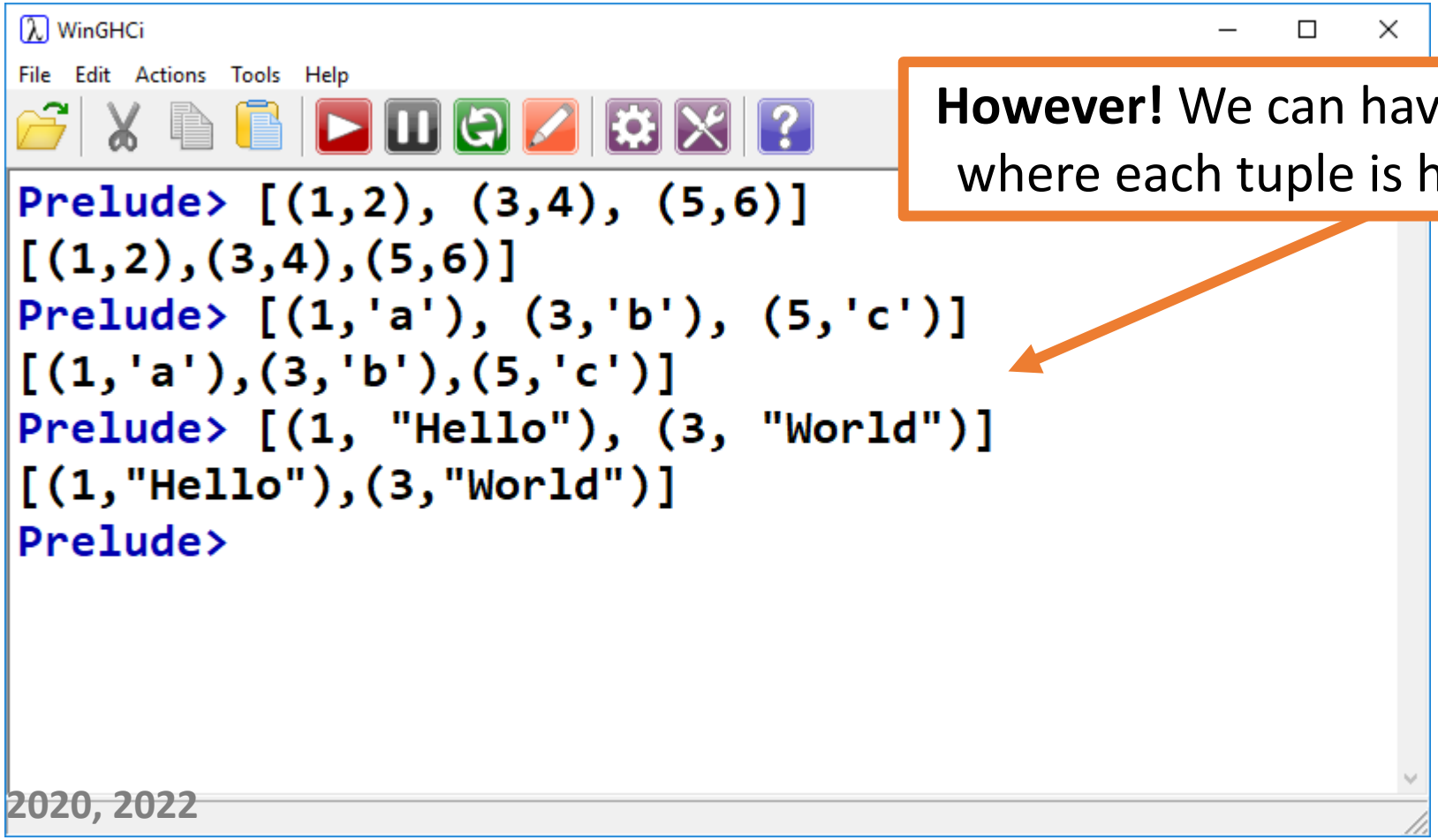
Elements can be added to the *beginning* of a list with the **cons** (:) operator



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

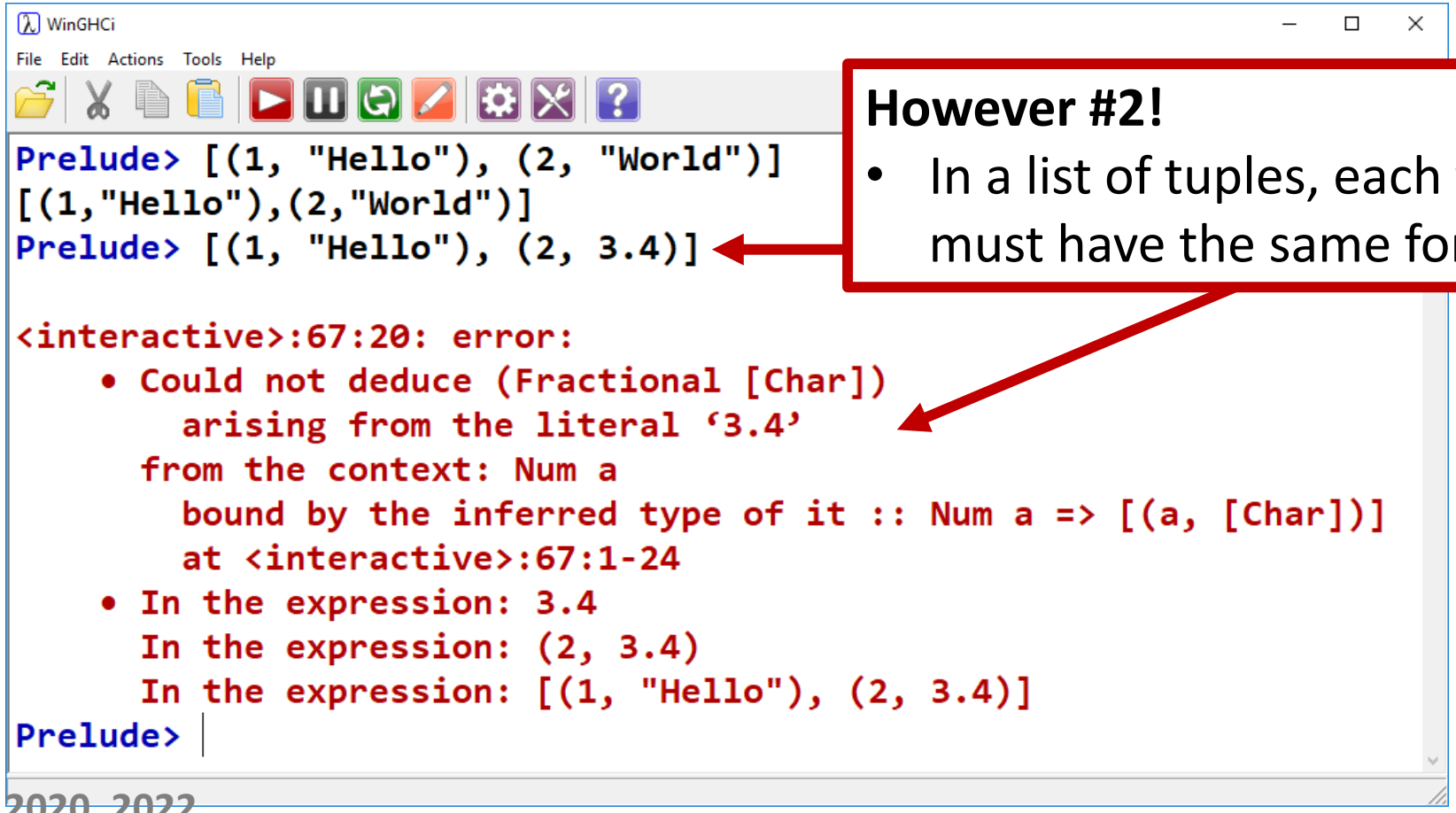
Tuples can be heterogeneous; lists must be homogeneous.



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

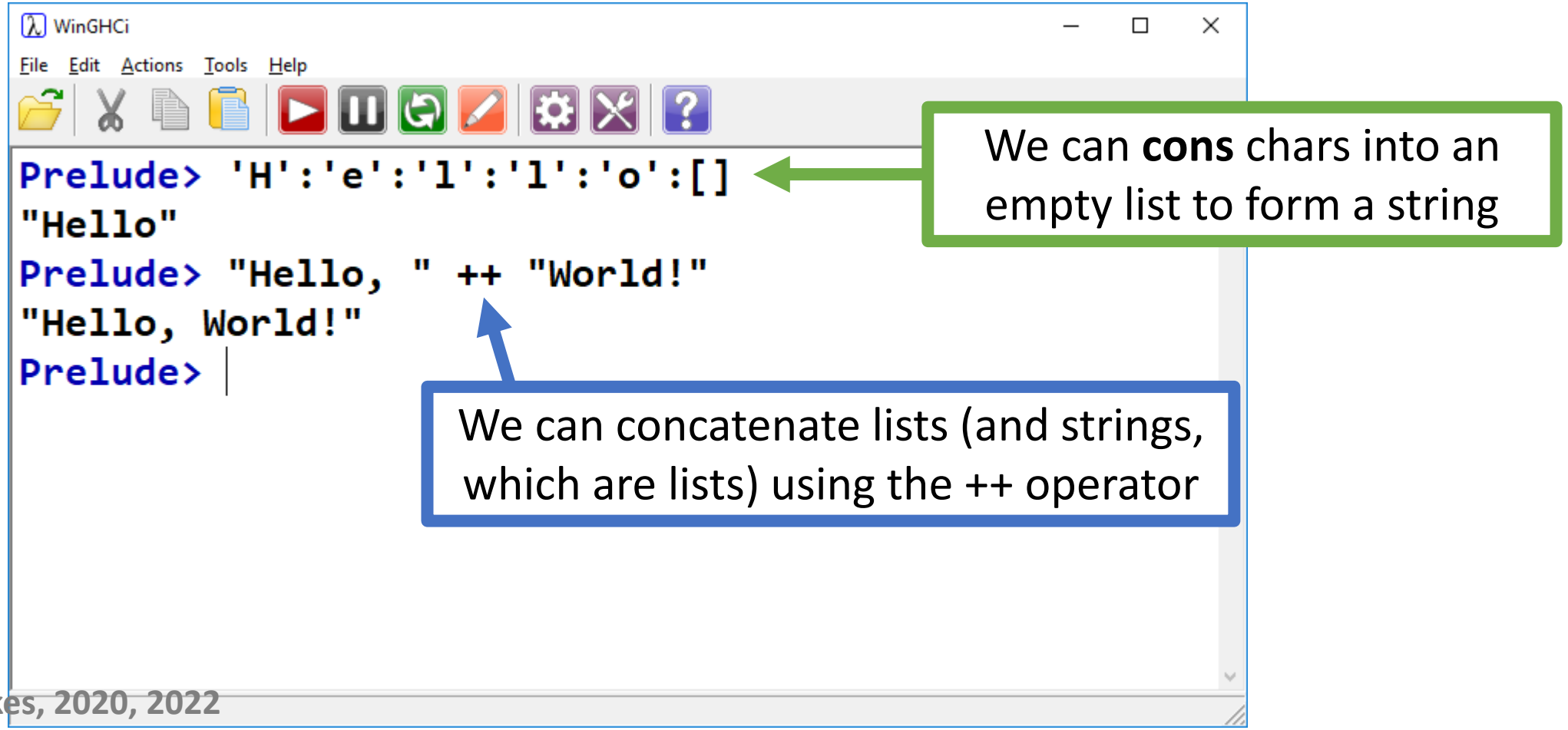
Lists & Tuples

Tuples can be heterogeneous, lists must be homogeneous.



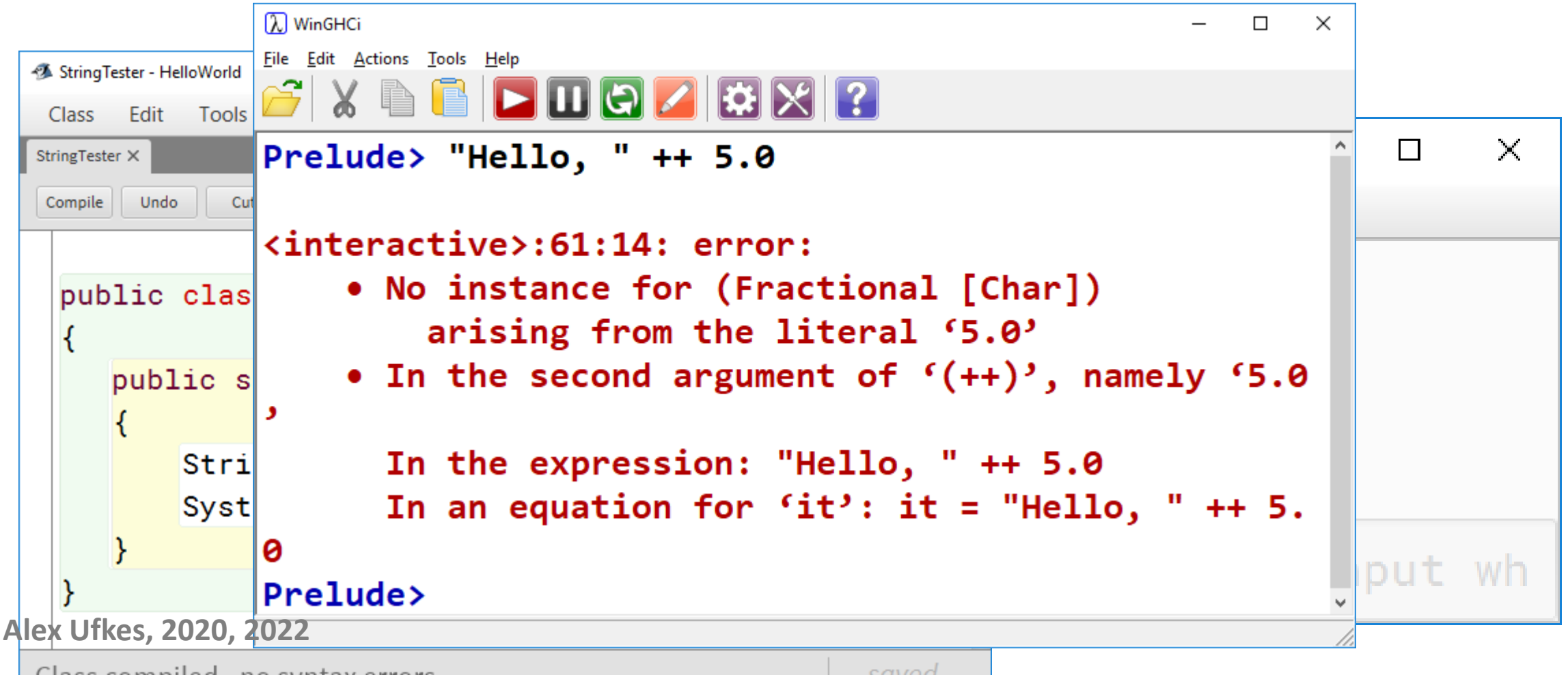
Strings

Strings are simply lists of chars:



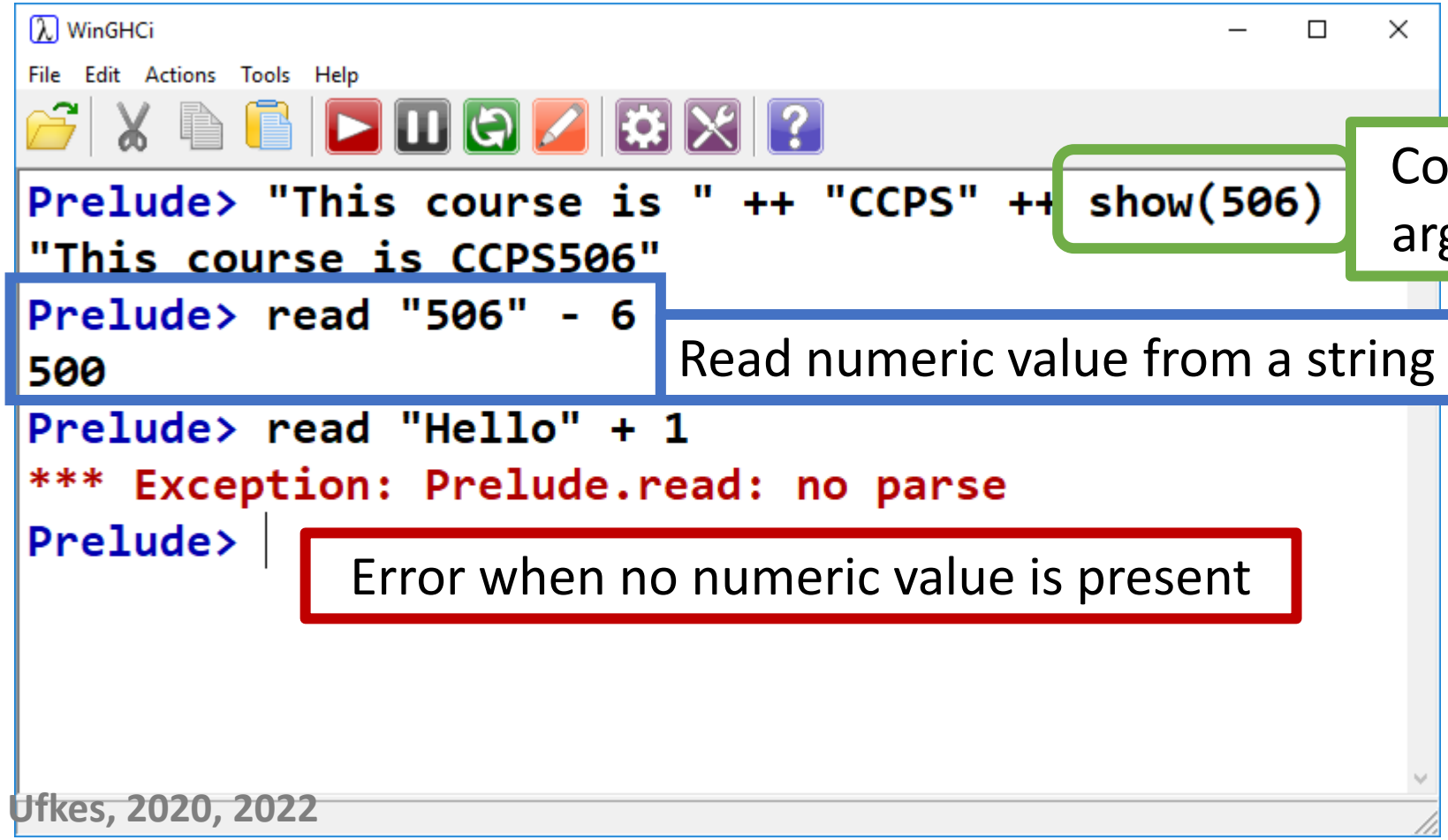
Strings

Concatenate multiple types? Java lets us…



Strings

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

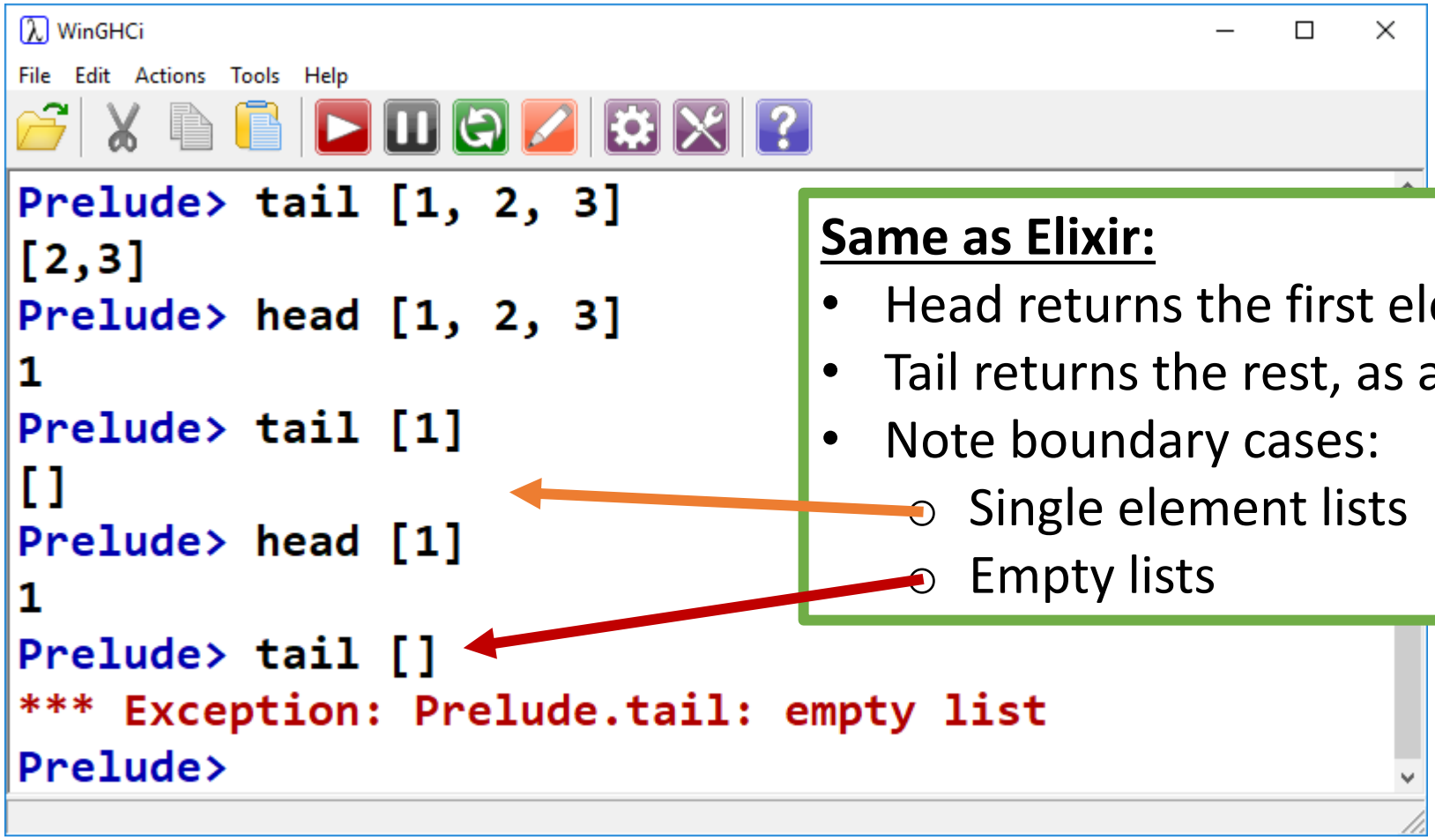


Operations on Lists

- 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.

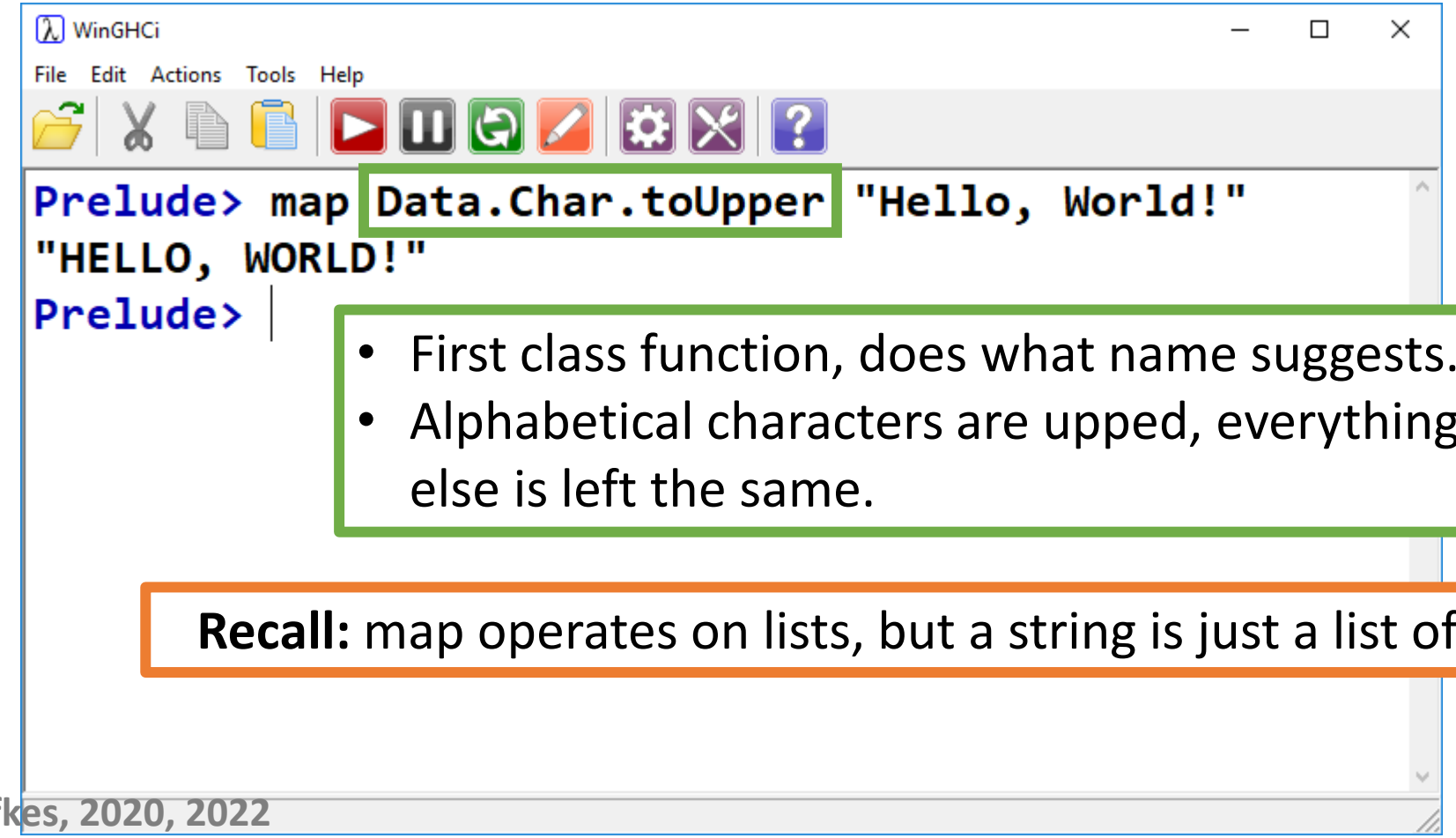
Three primary list-processing functions: **`map`**, **`filter`**, **`foldr`** (and **`foldl`**)

Head & Tail



map

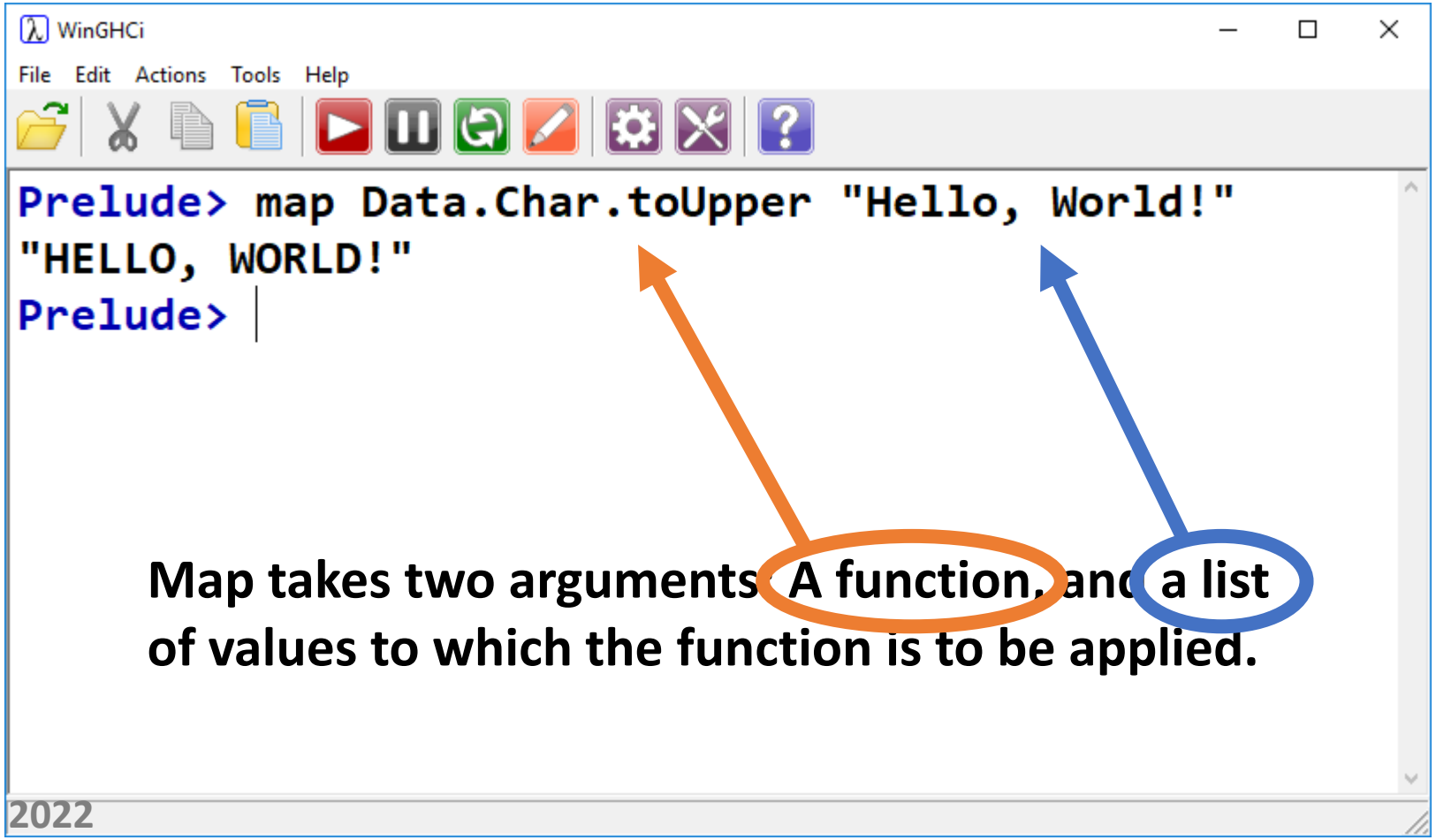
Similar to Elixir's **Enum.map**



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

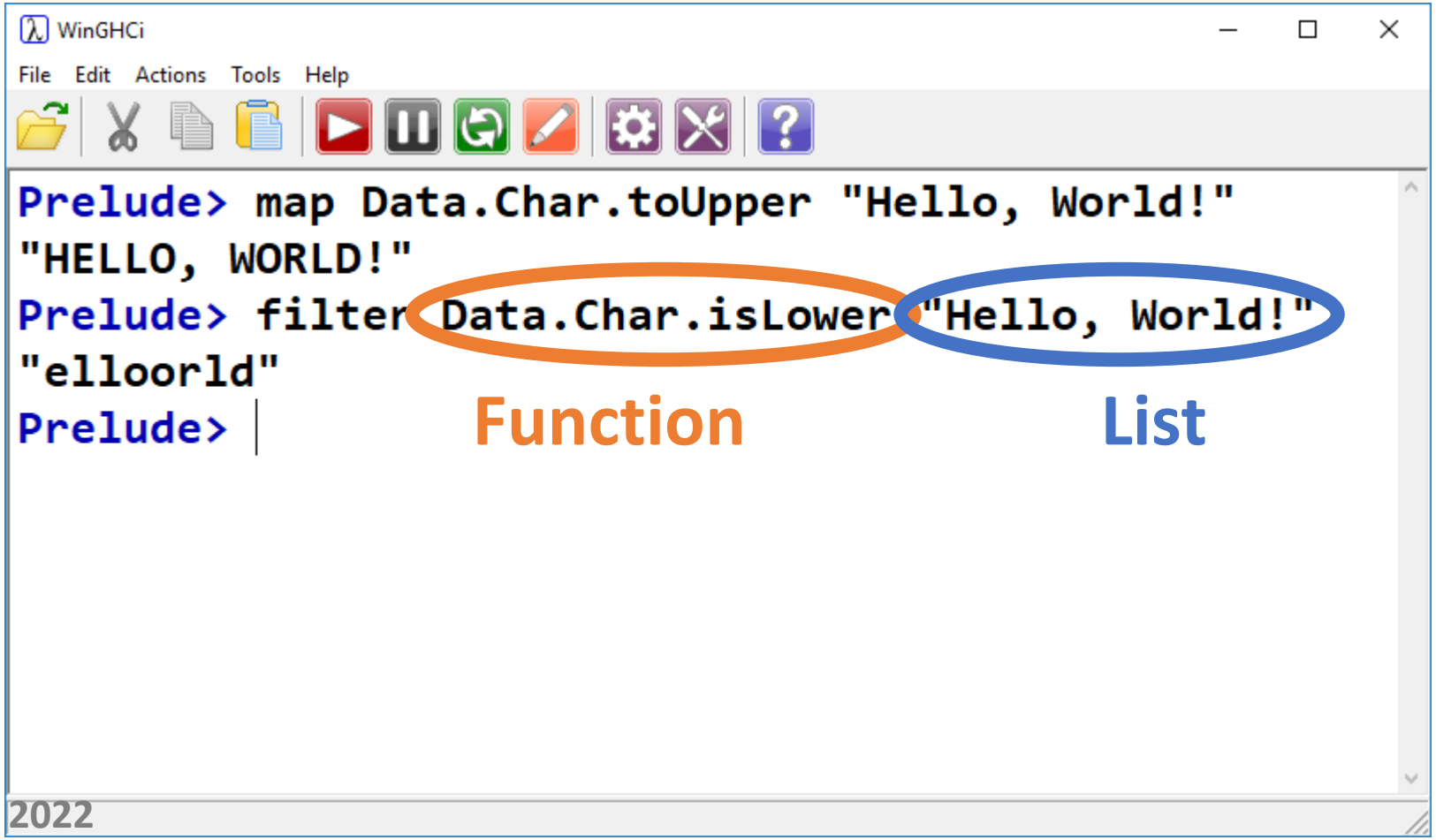
map

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



filter

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



`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.

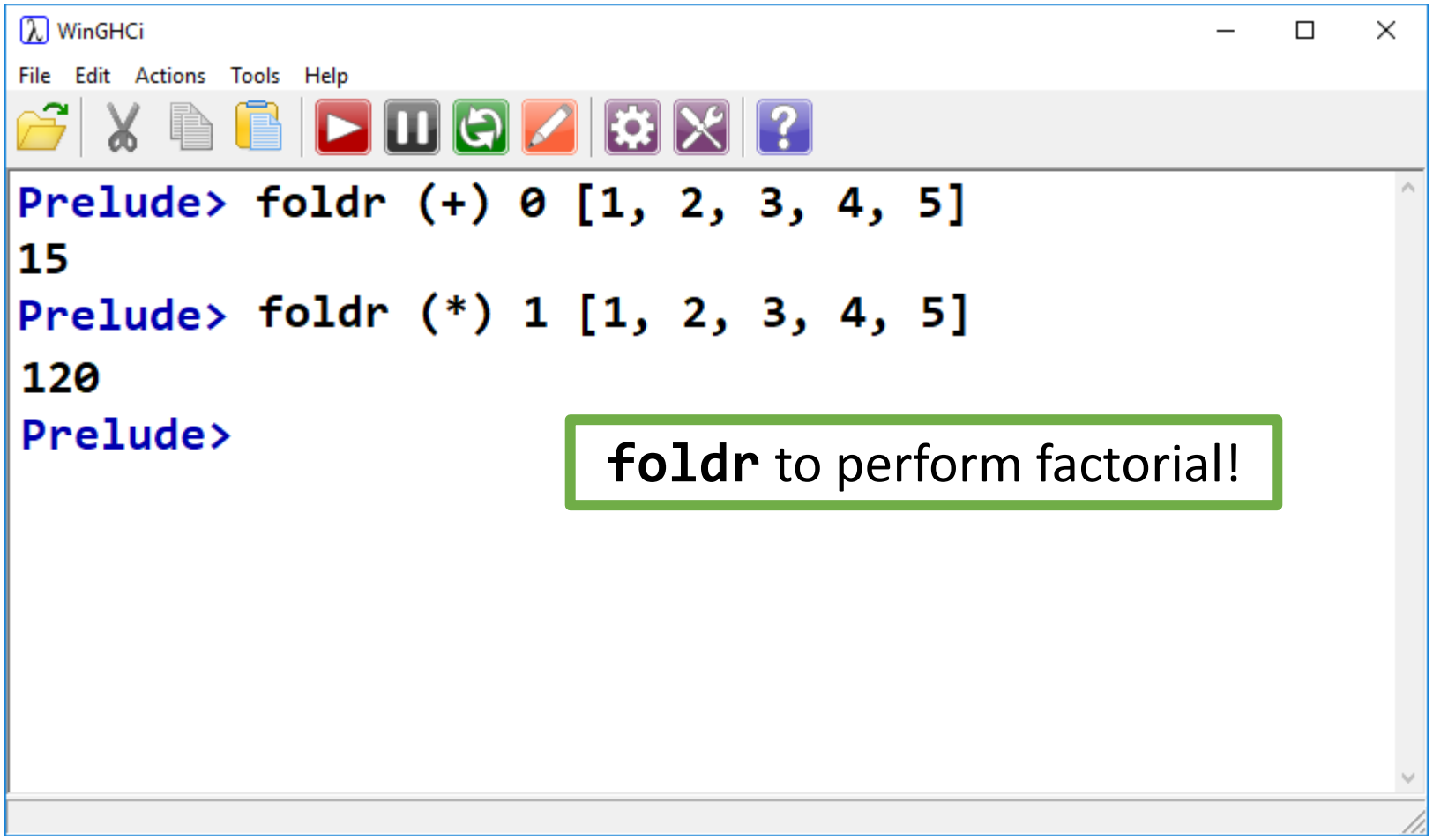
```
foldr (+) 0 [1, 2, 3, 4, 5]
```

```
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]
```

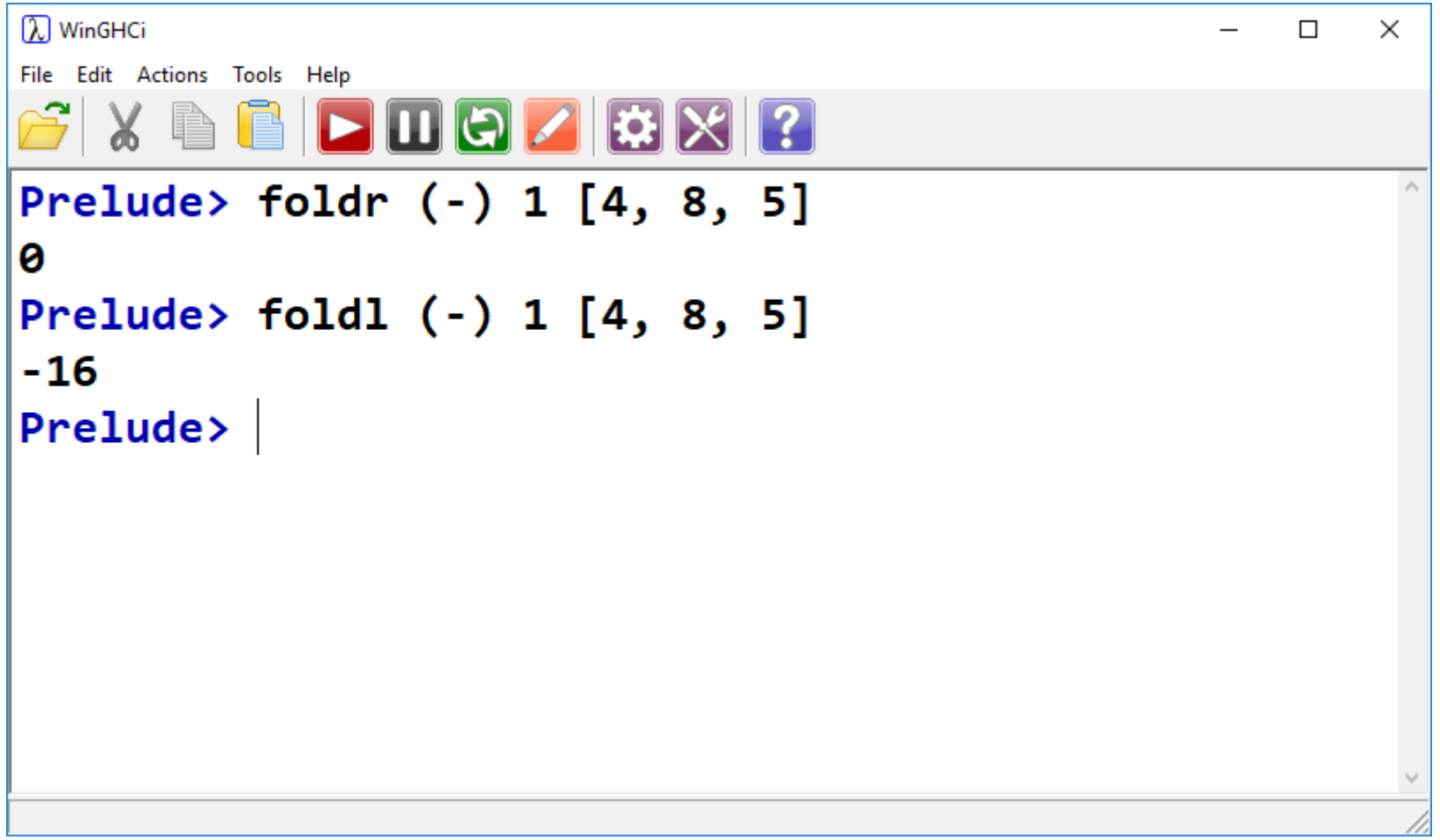
```
1 - 4 – 8 - 5
```

Is actually:

```
(((1 - 4) – 8) - 5)
```

```
-16
```

foldl VS foldr



List Generation

Syntactic sugar:

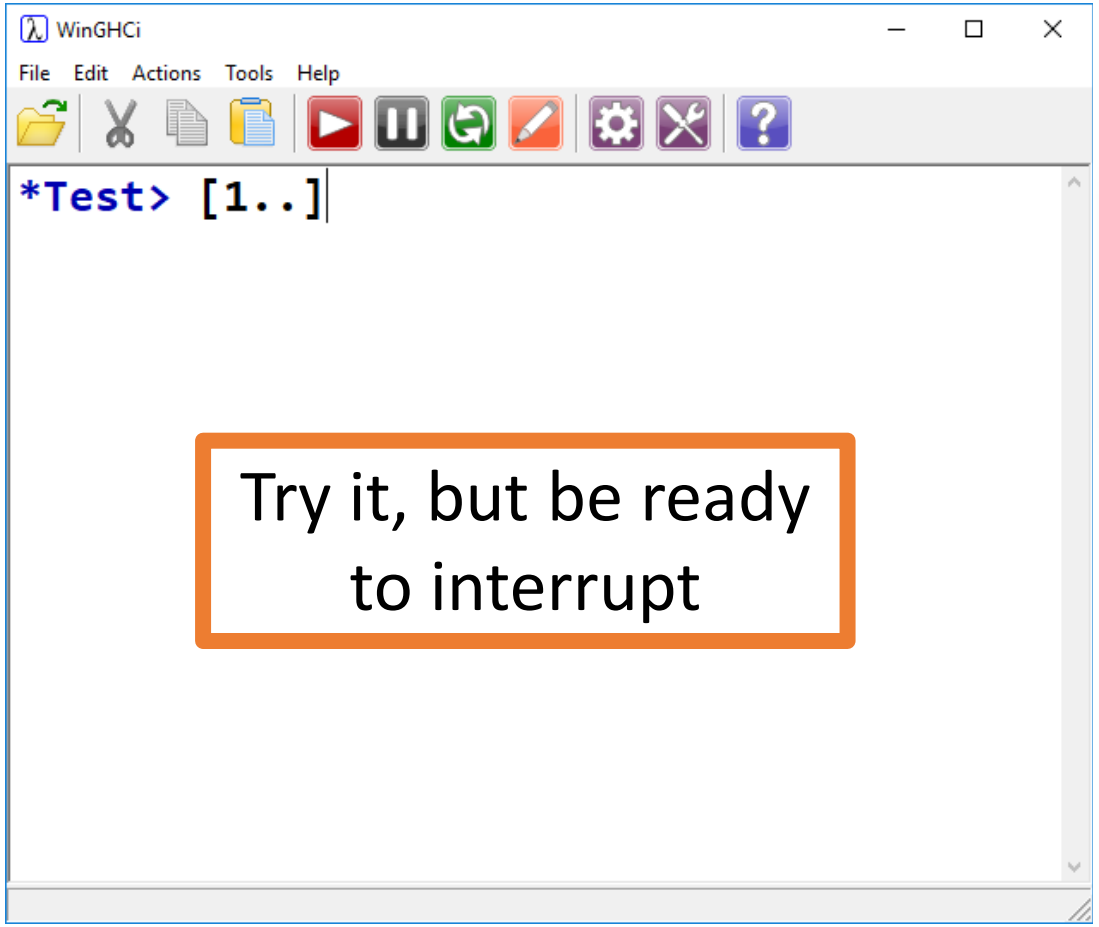
List declaration: `list = [1, 2, 3, 4, 5, 6, 7, 8, 9]`

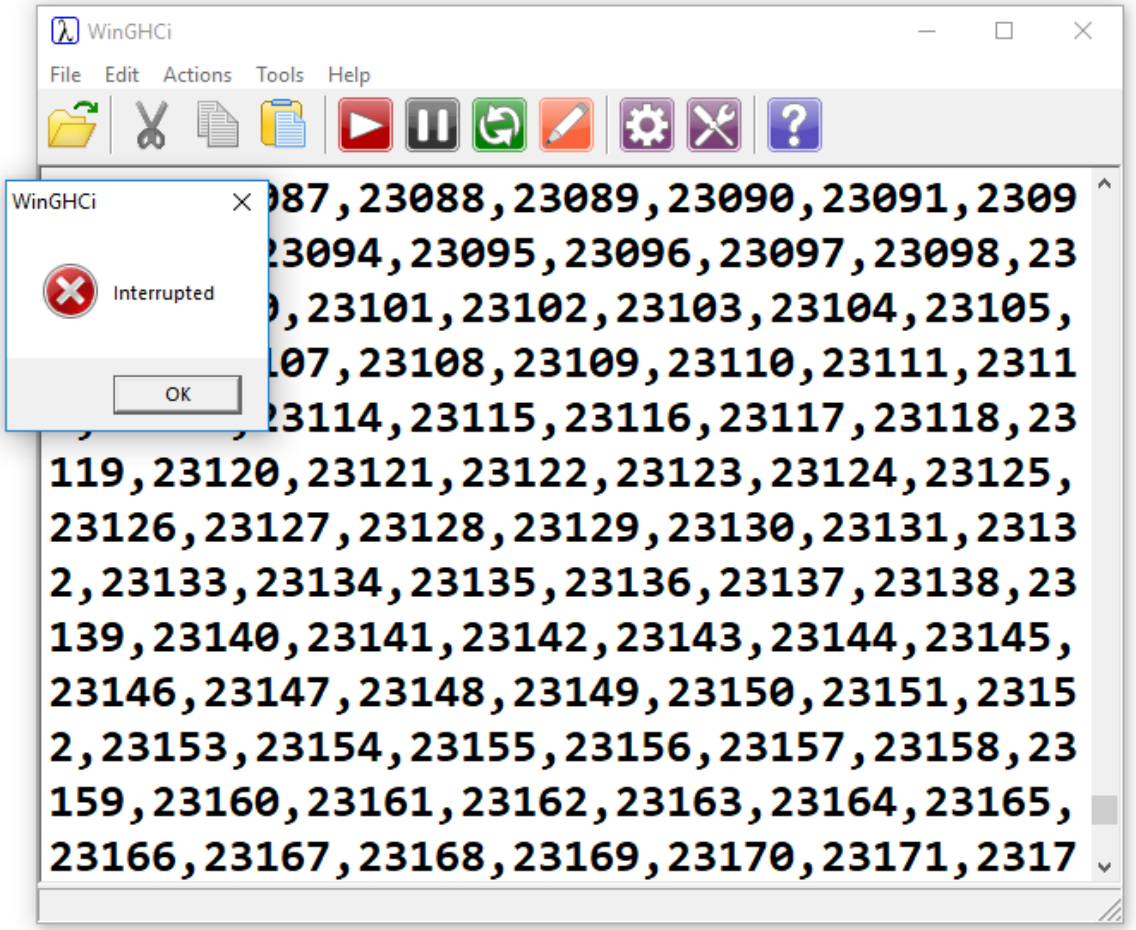
Can be written: `list = [1..9]`

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

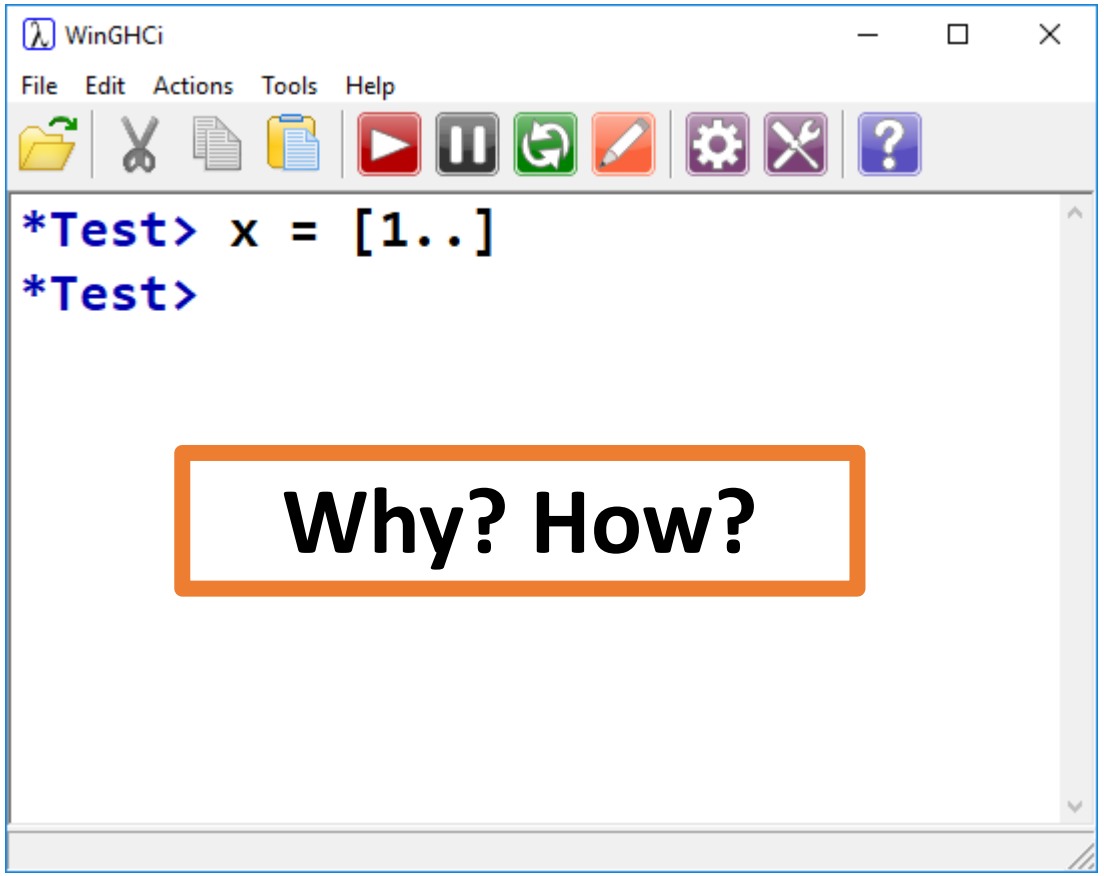
Interval is discerned from difference between first two elements

Infinite Lists?





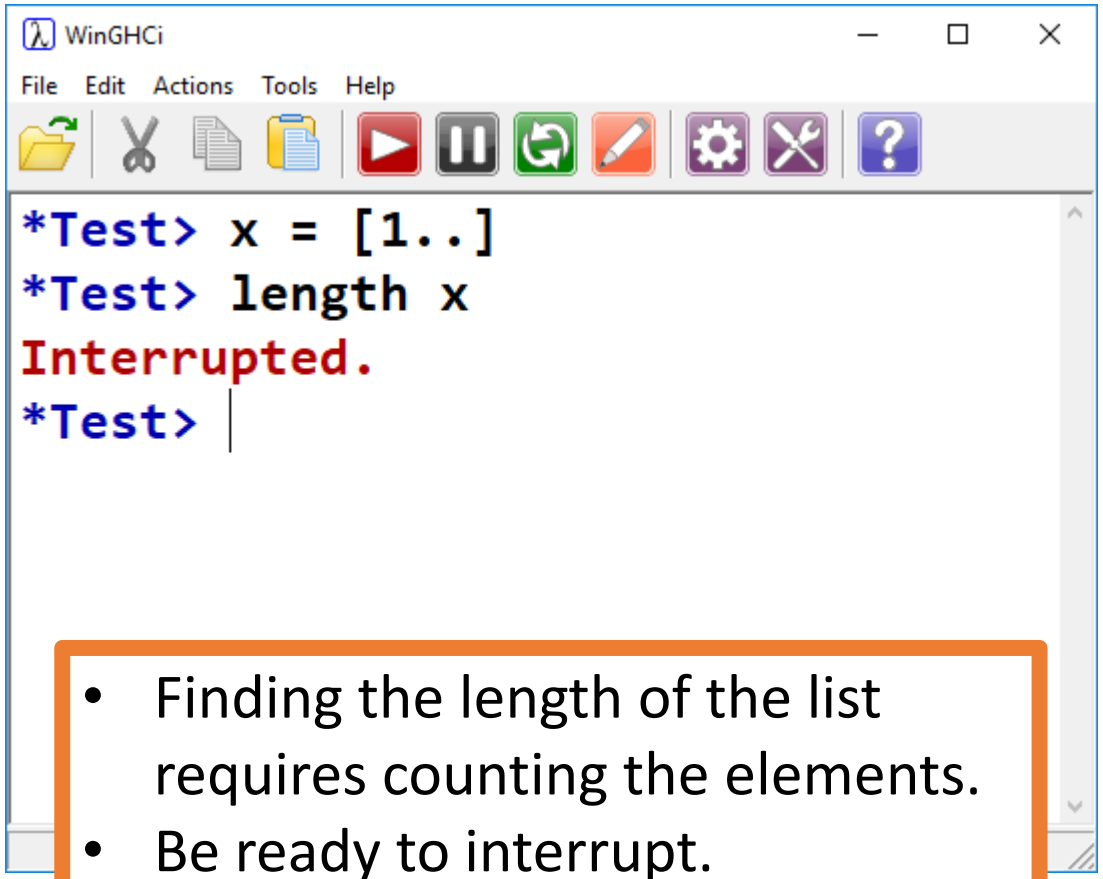
Infinite Lists?



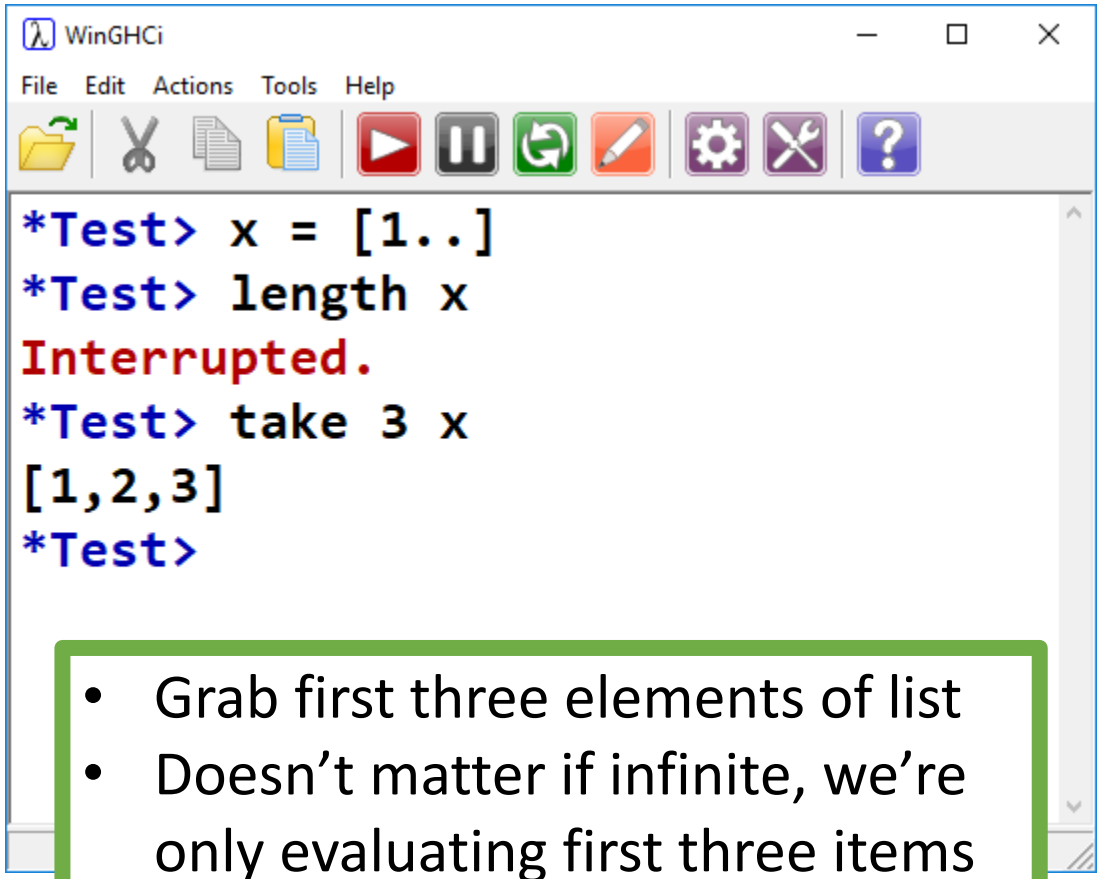
Haskell is lazy!

- 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?



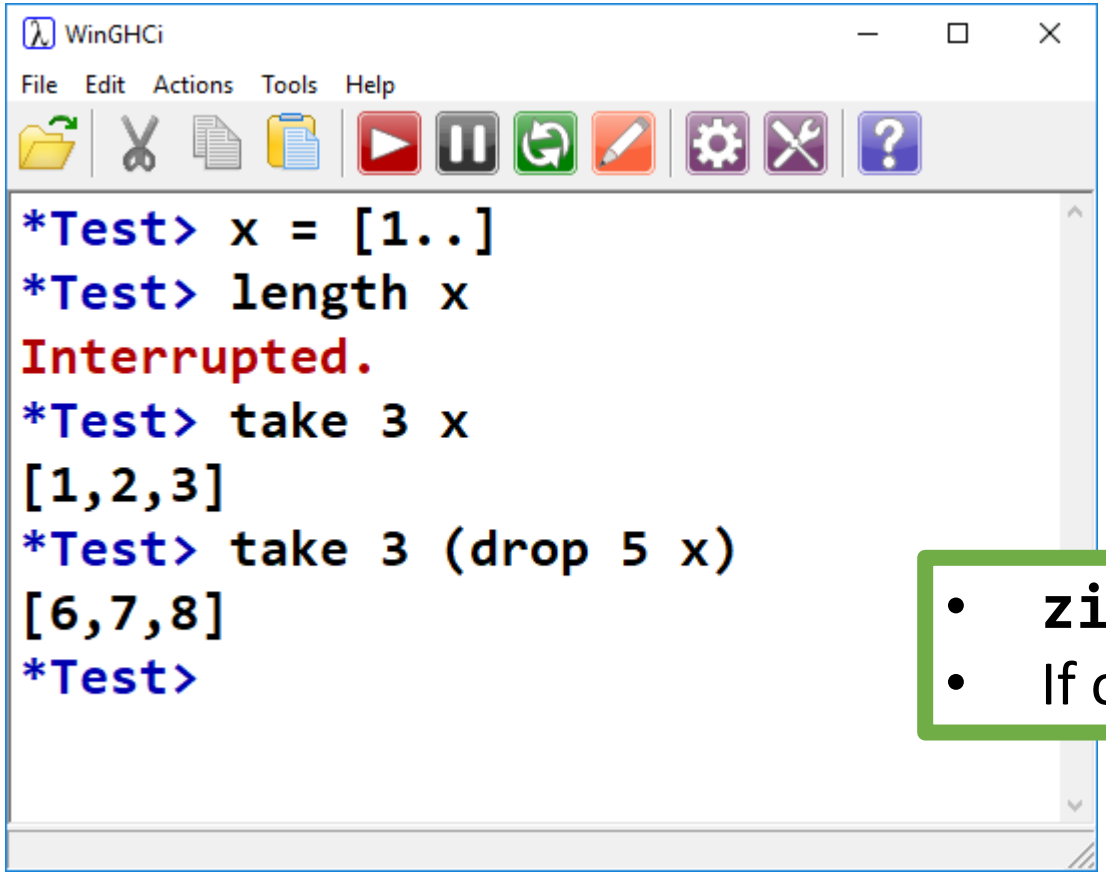
- Finding the length of the list requires counting the elements.
- Be ready to interrupt.

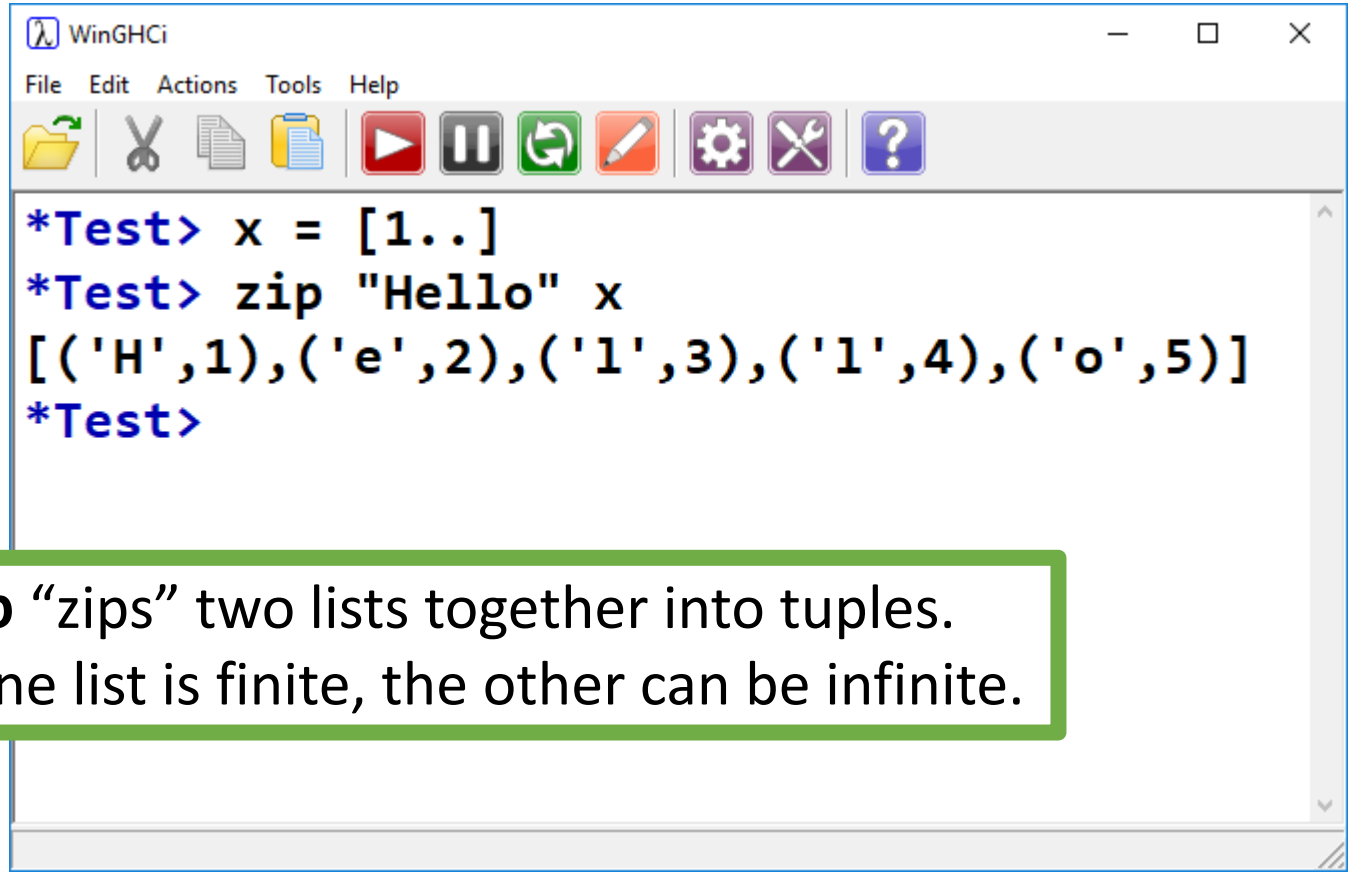


- Grab first three elements of list
- Doesn't matter if infinite, we're only evaluating first three items

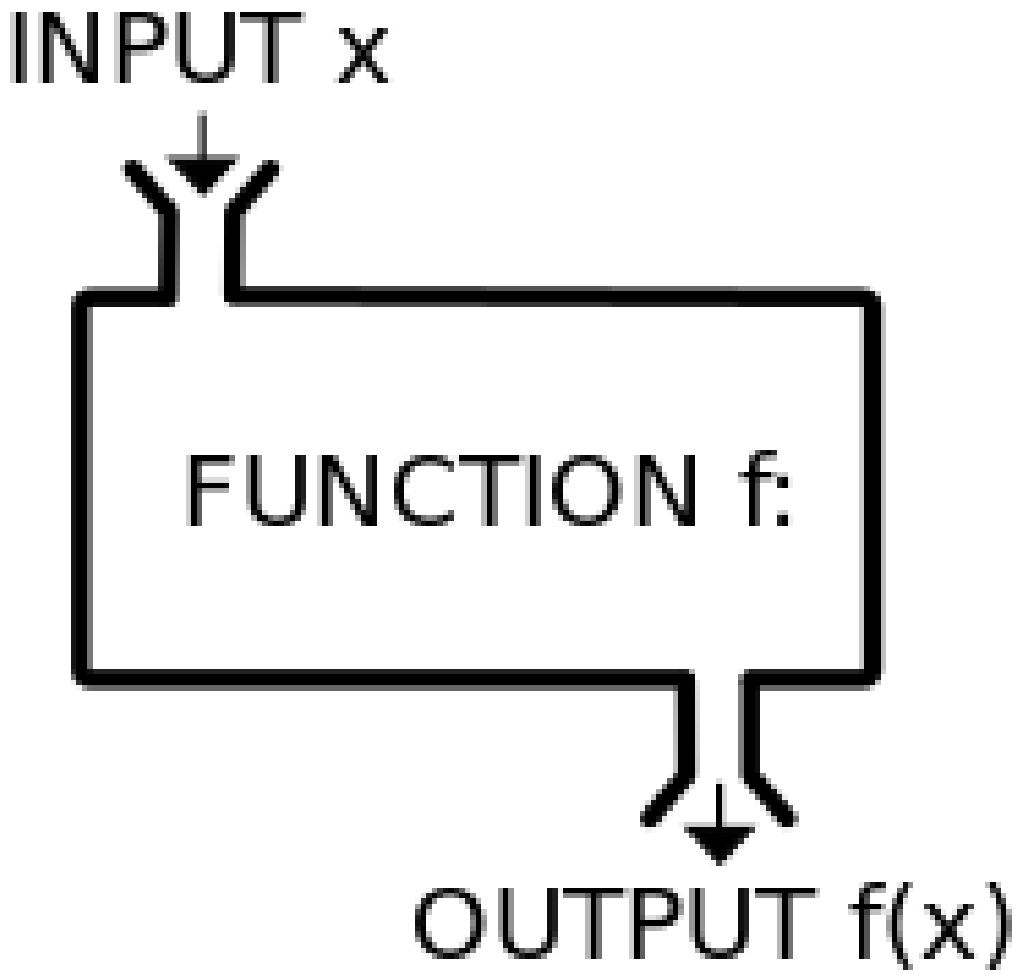
Infinite Lists?

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



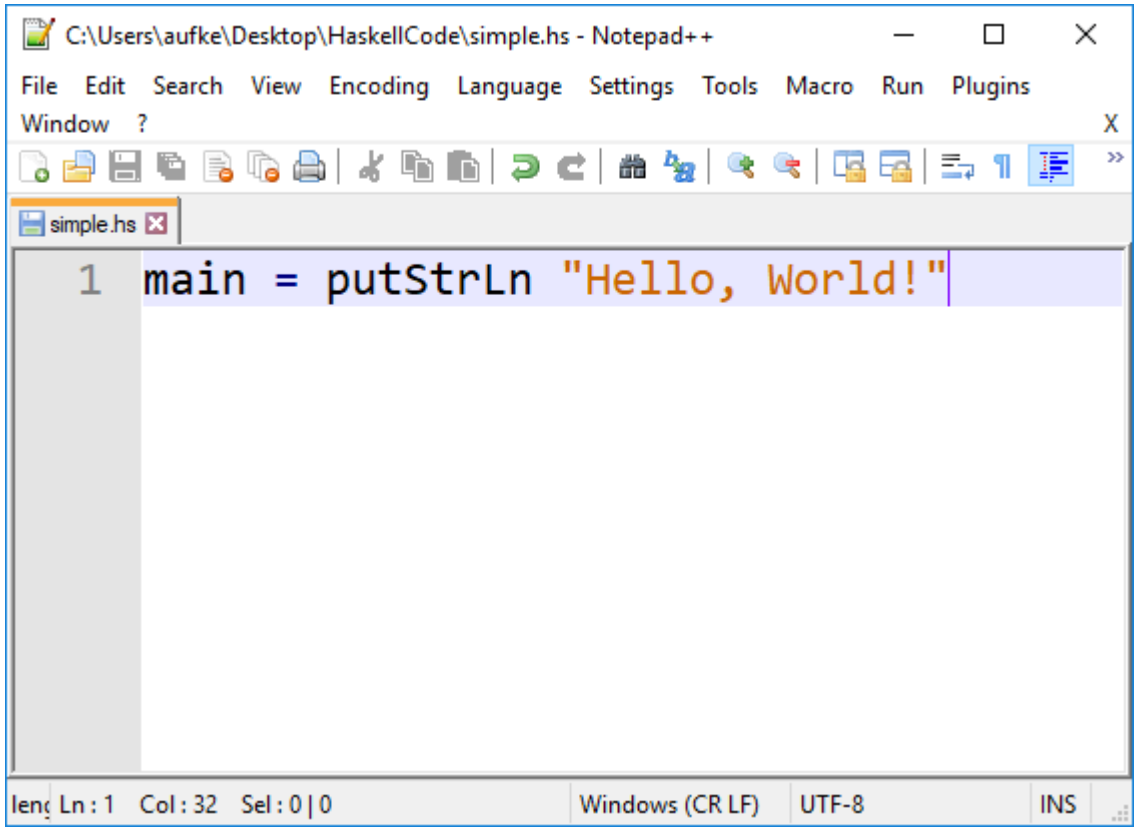


- **`zip`** "zips" two lists together into tuples.
- If one list is finite, the other can be infinite.



Functions in Haskell

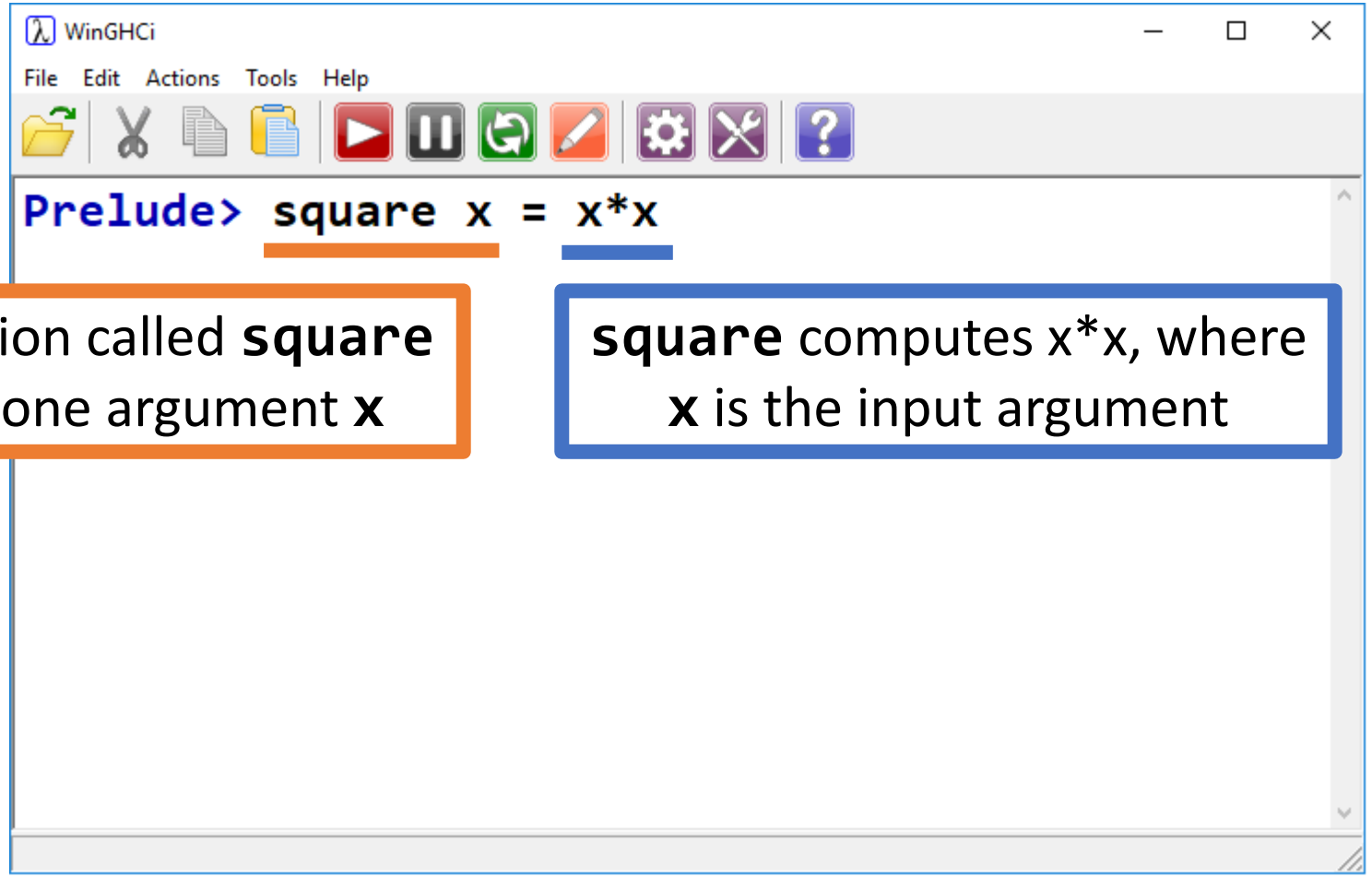
As expected of a pure functional language, functions are central in Haskell



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

Functions in Haskell

Let's start simple:

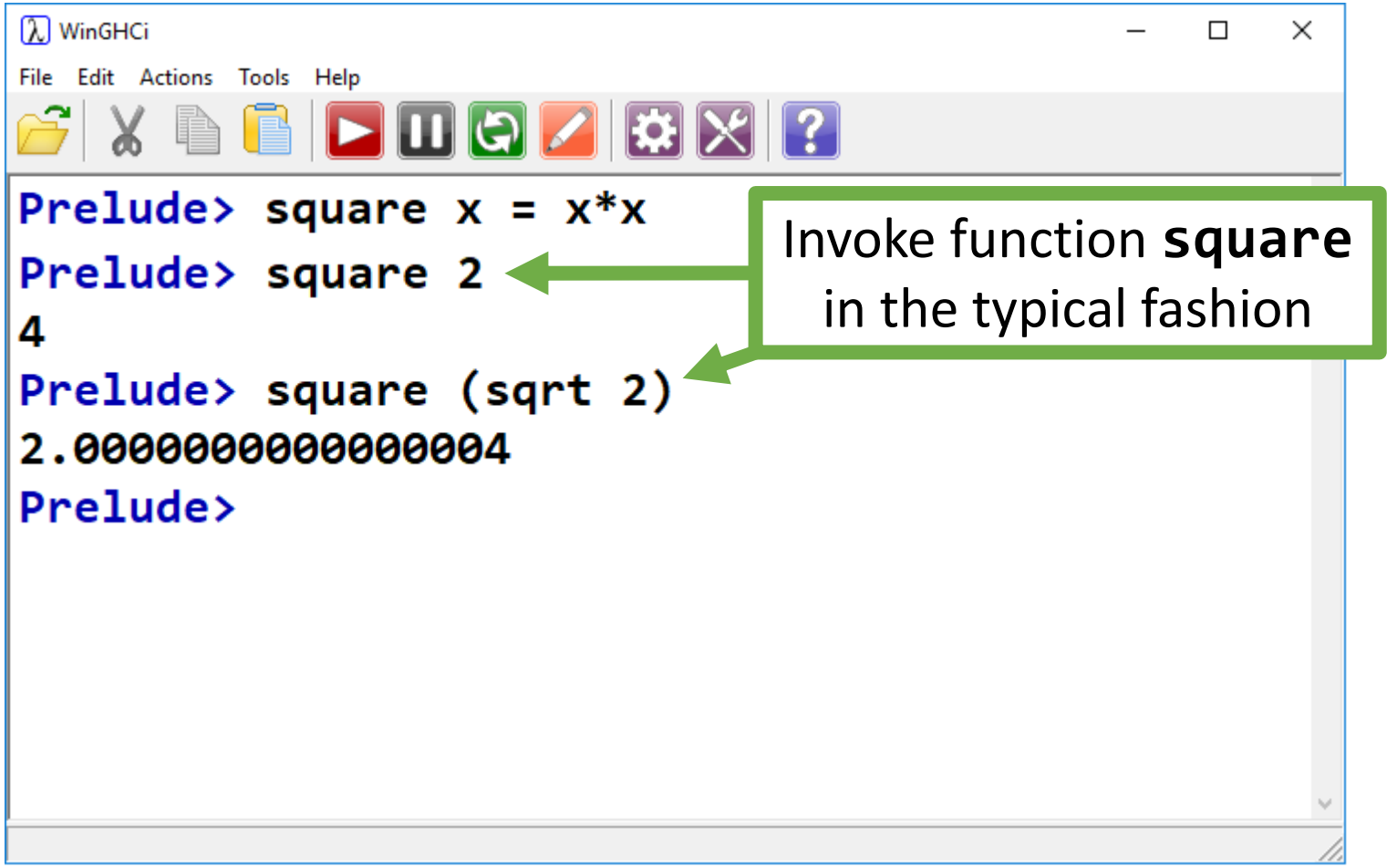


Define function called **square** that takes one argument **x**

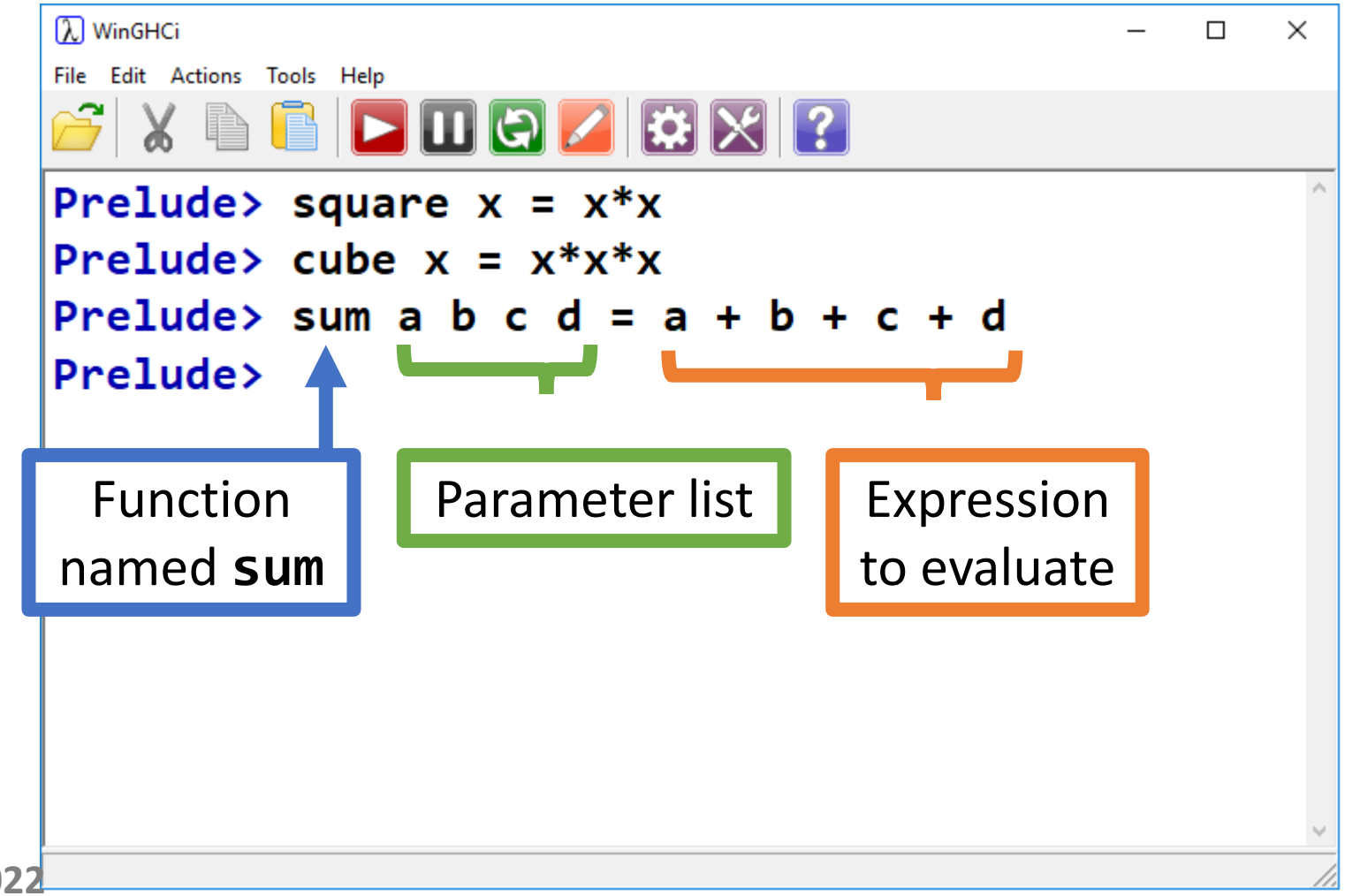
square computes x*x, where **x** is the input argument

Functions in Haskell

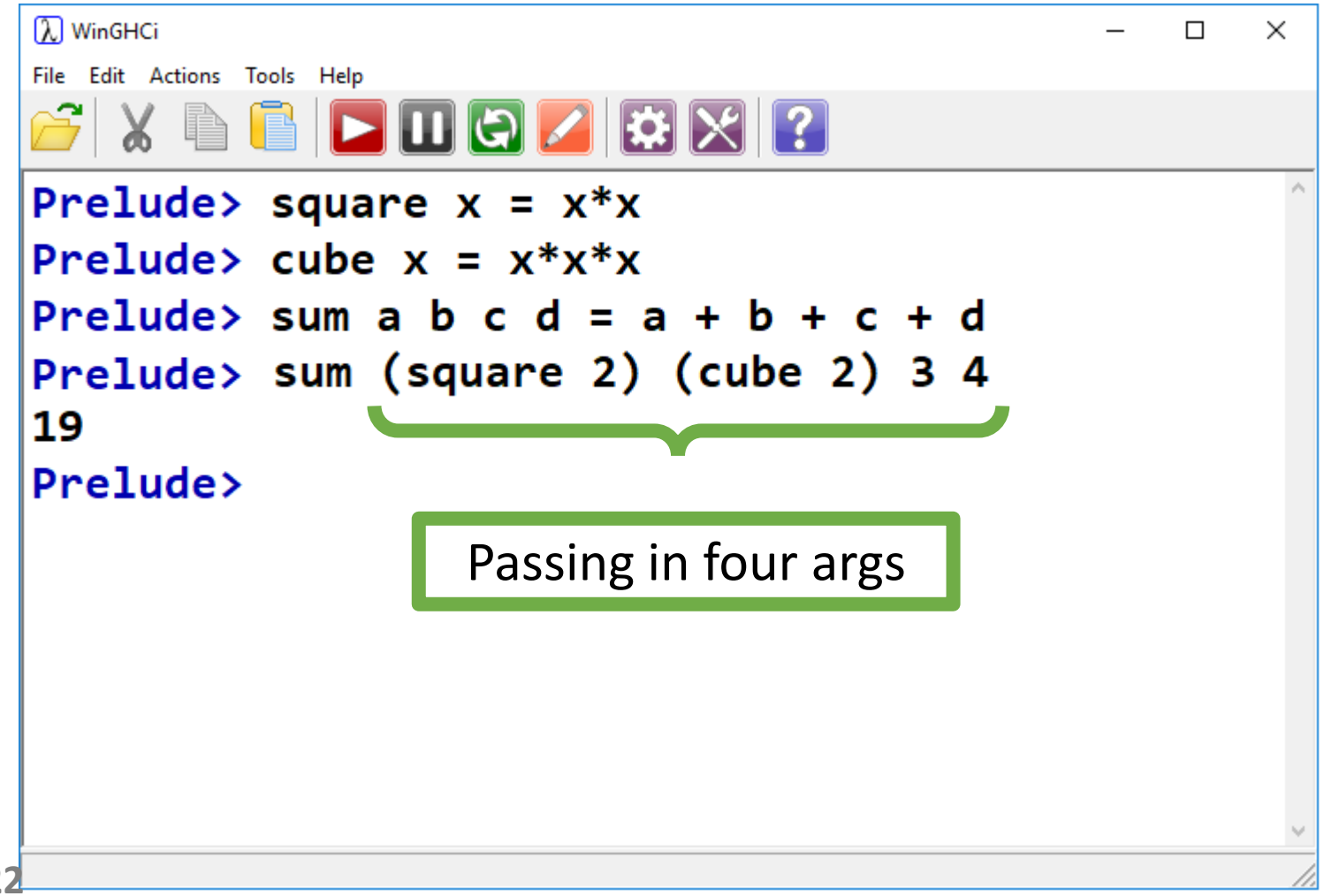
Let's start simple:



Functions in Haskell

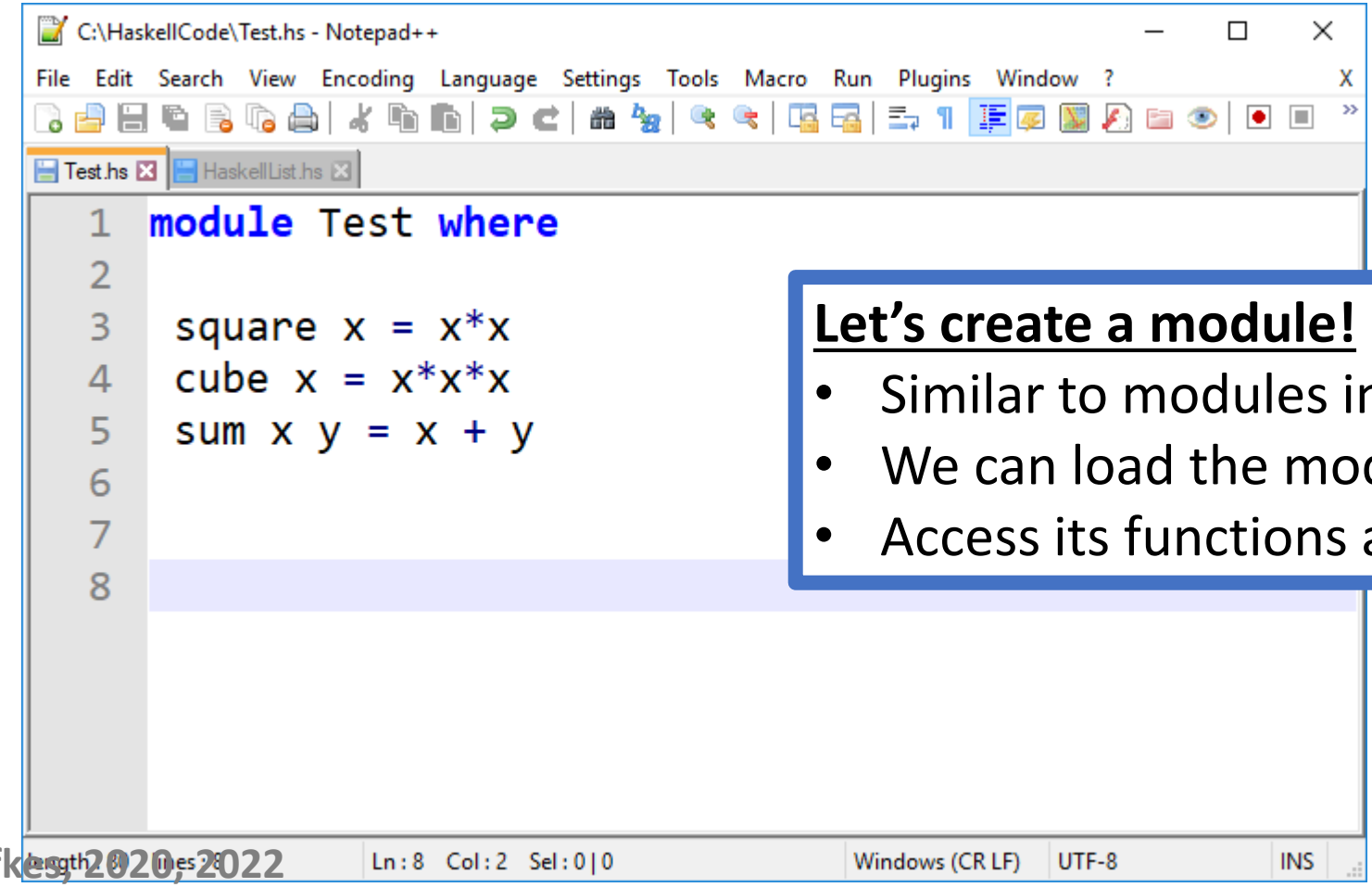


Functions in Haskell



Haskell Modules

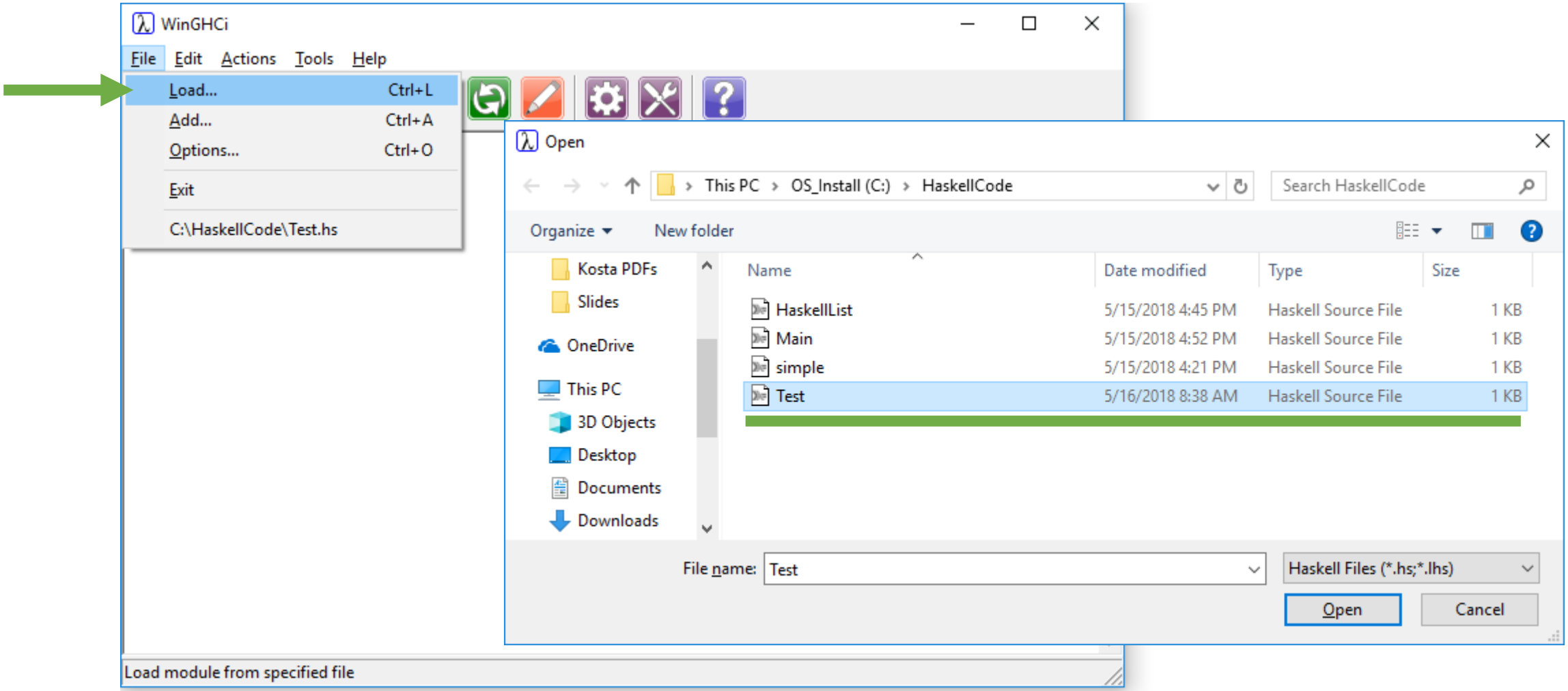
This is getting tedious to type interactively.

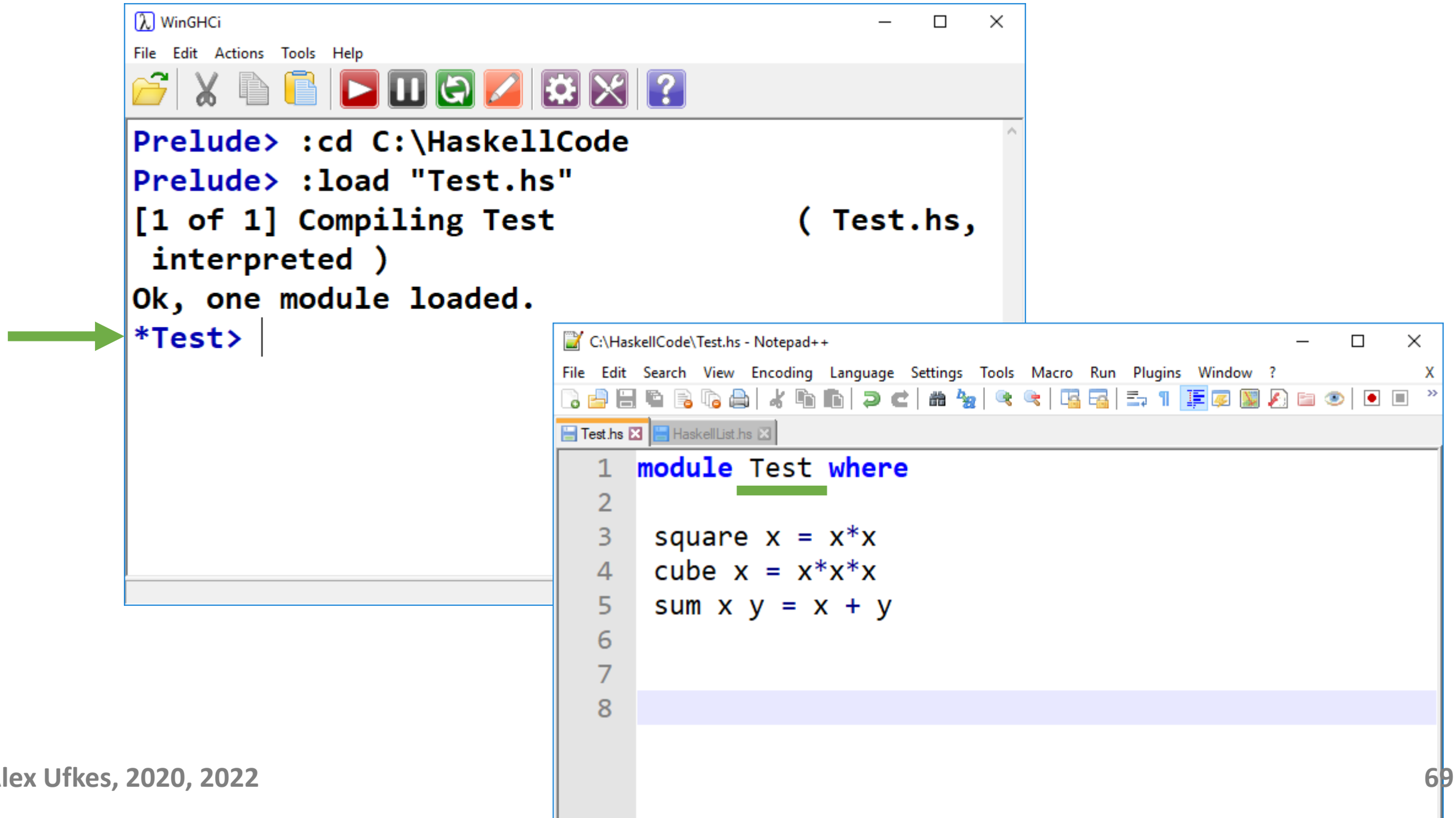


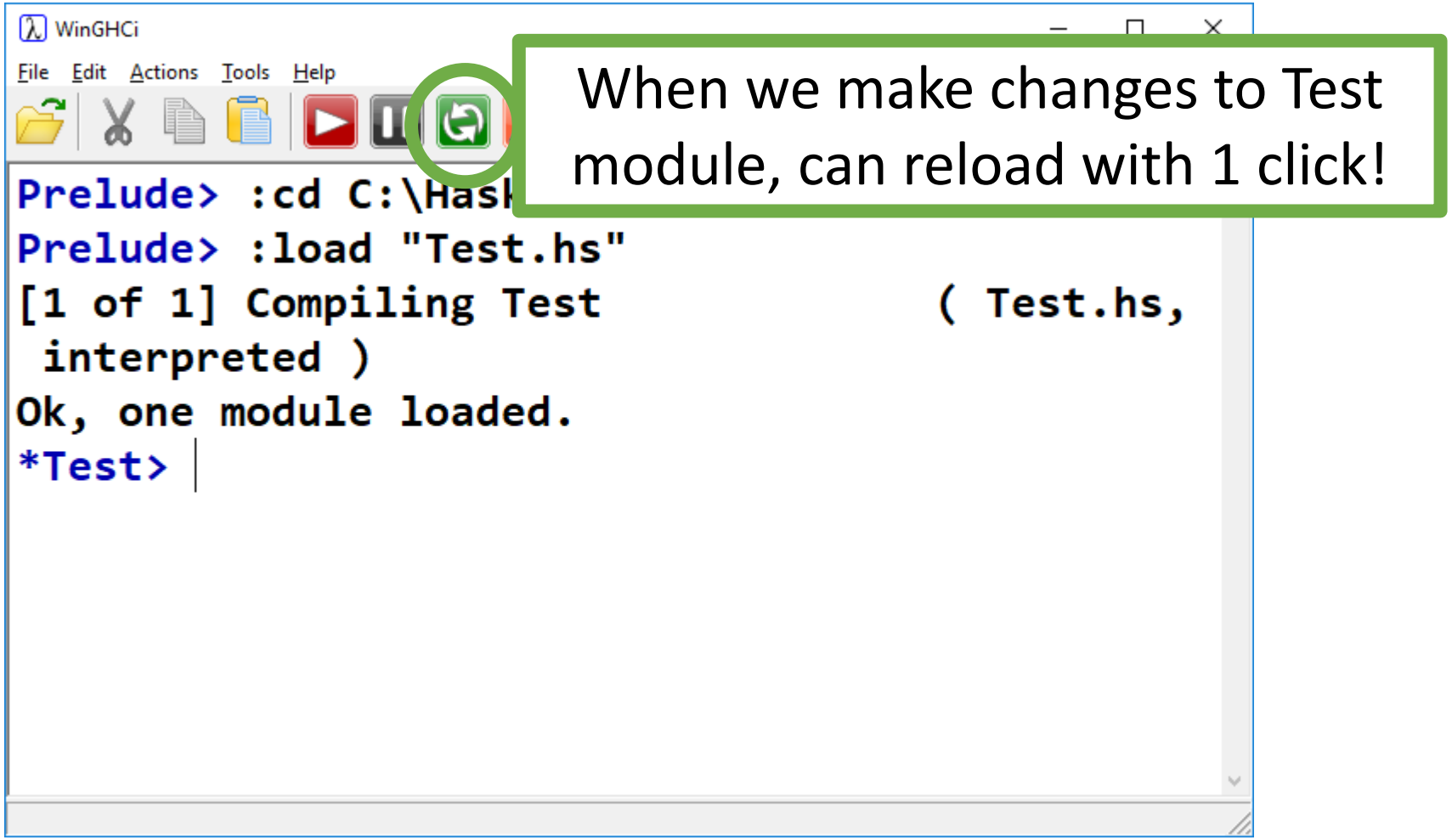
Let's create a module!

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

Loading a Module

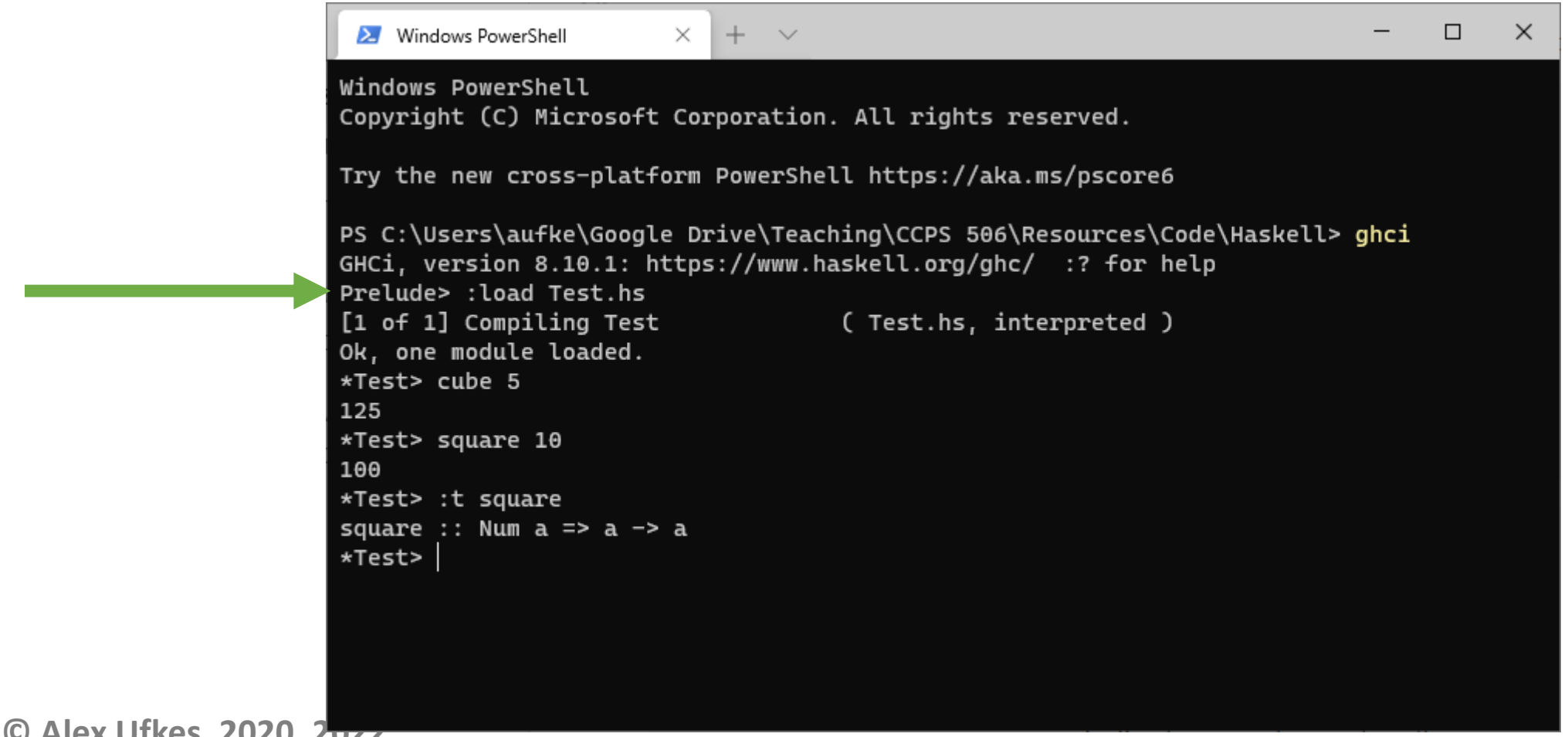






Loading a Module

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

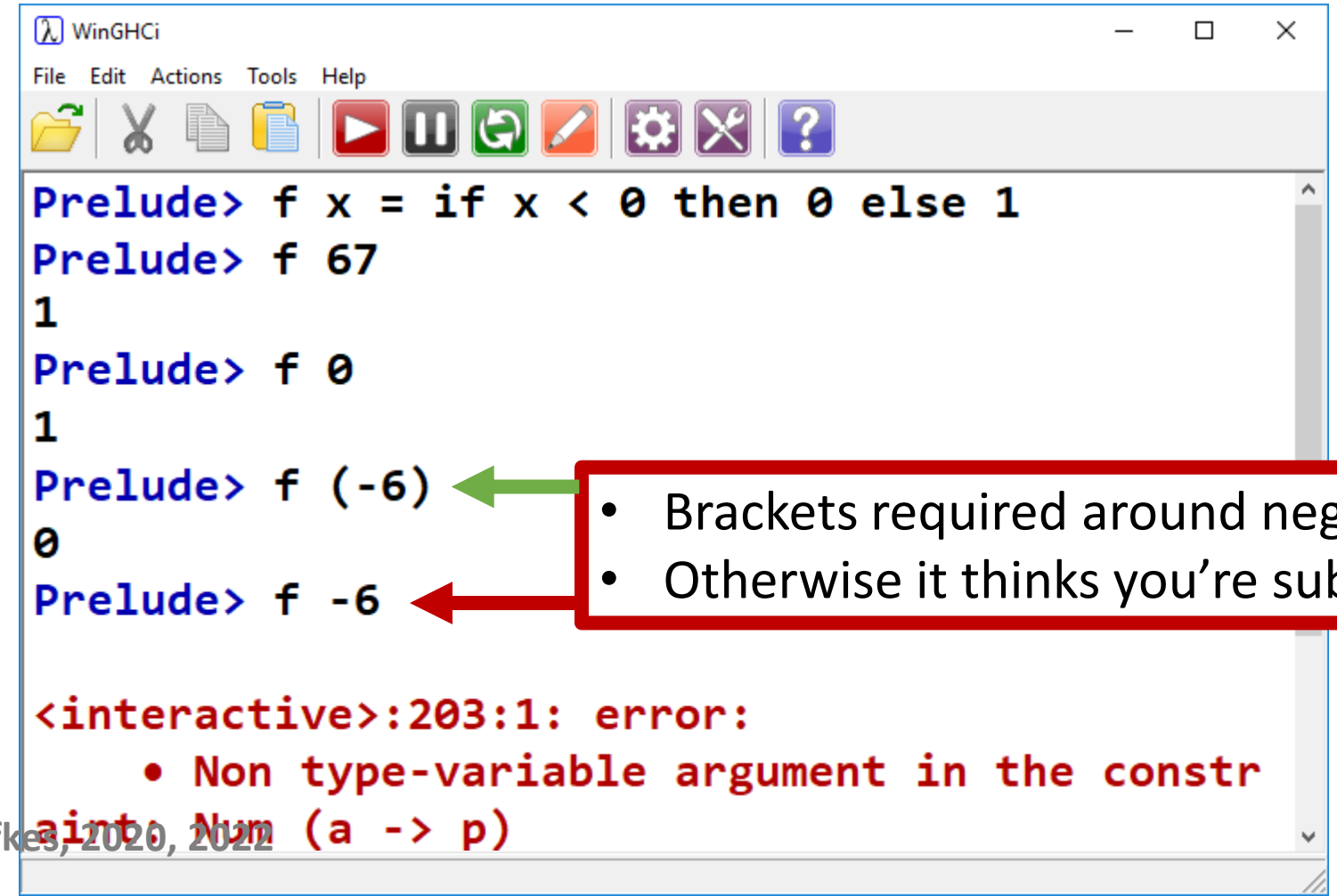


Control Structures

`if-then-else` `case` `let-in`

Control Structures

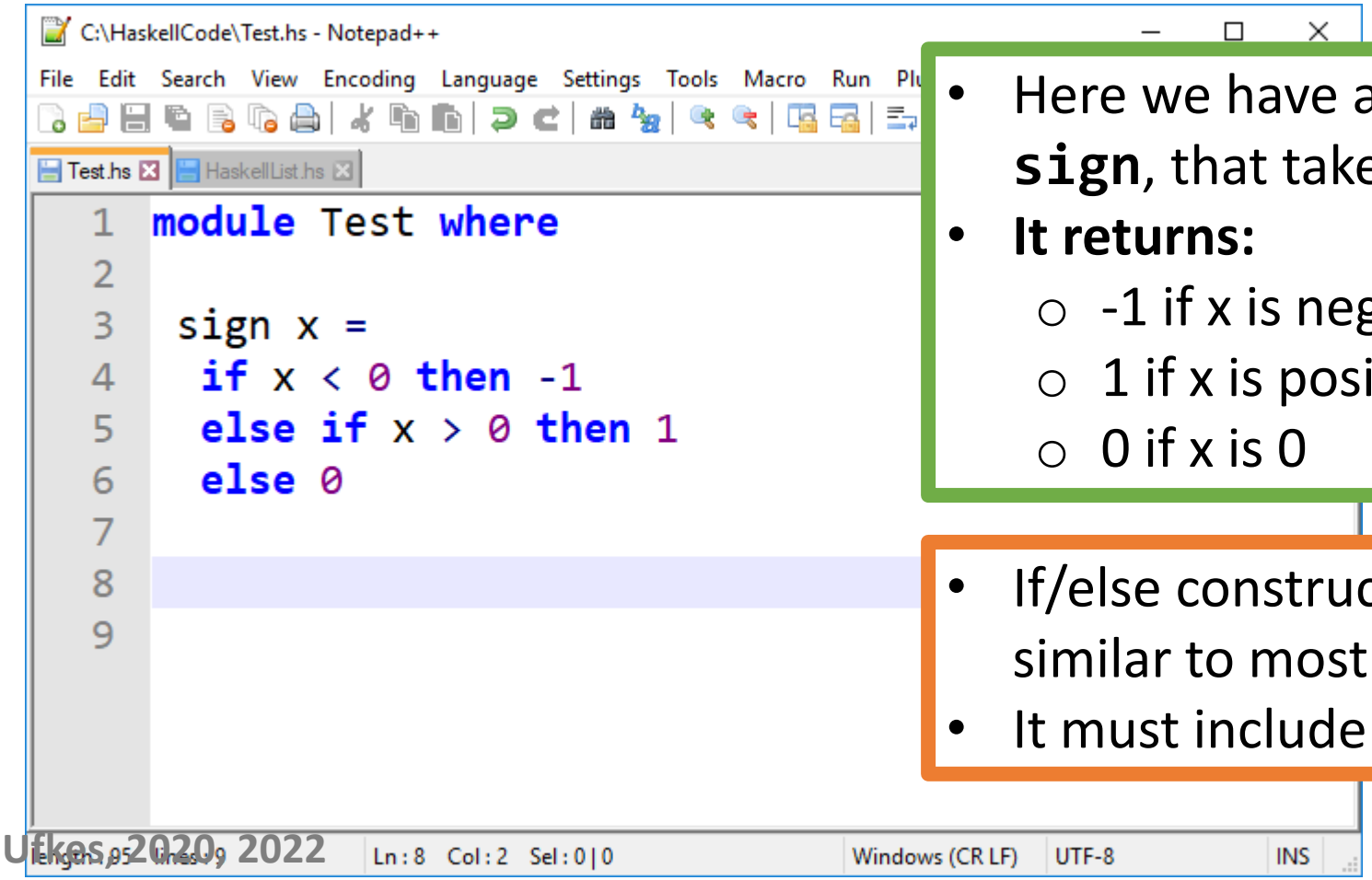
`if then else`



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

Control Structures

`if then else if then else`

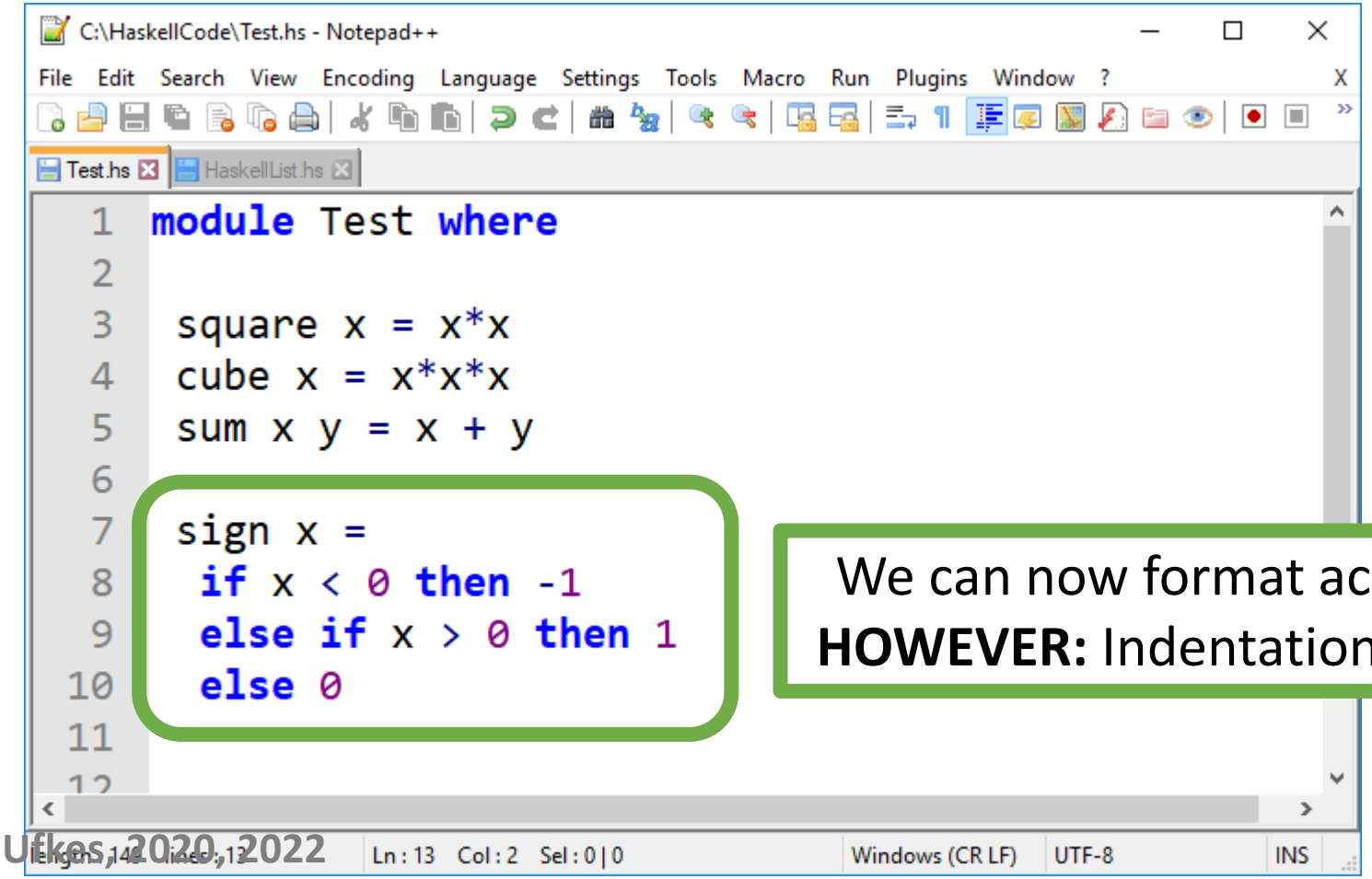


- Here we have a function named **`sign`**, that takes one argument x
- **It returns:**
  - -1 if x is negative
  - 1 if x is positive
  - 0 if x is 0

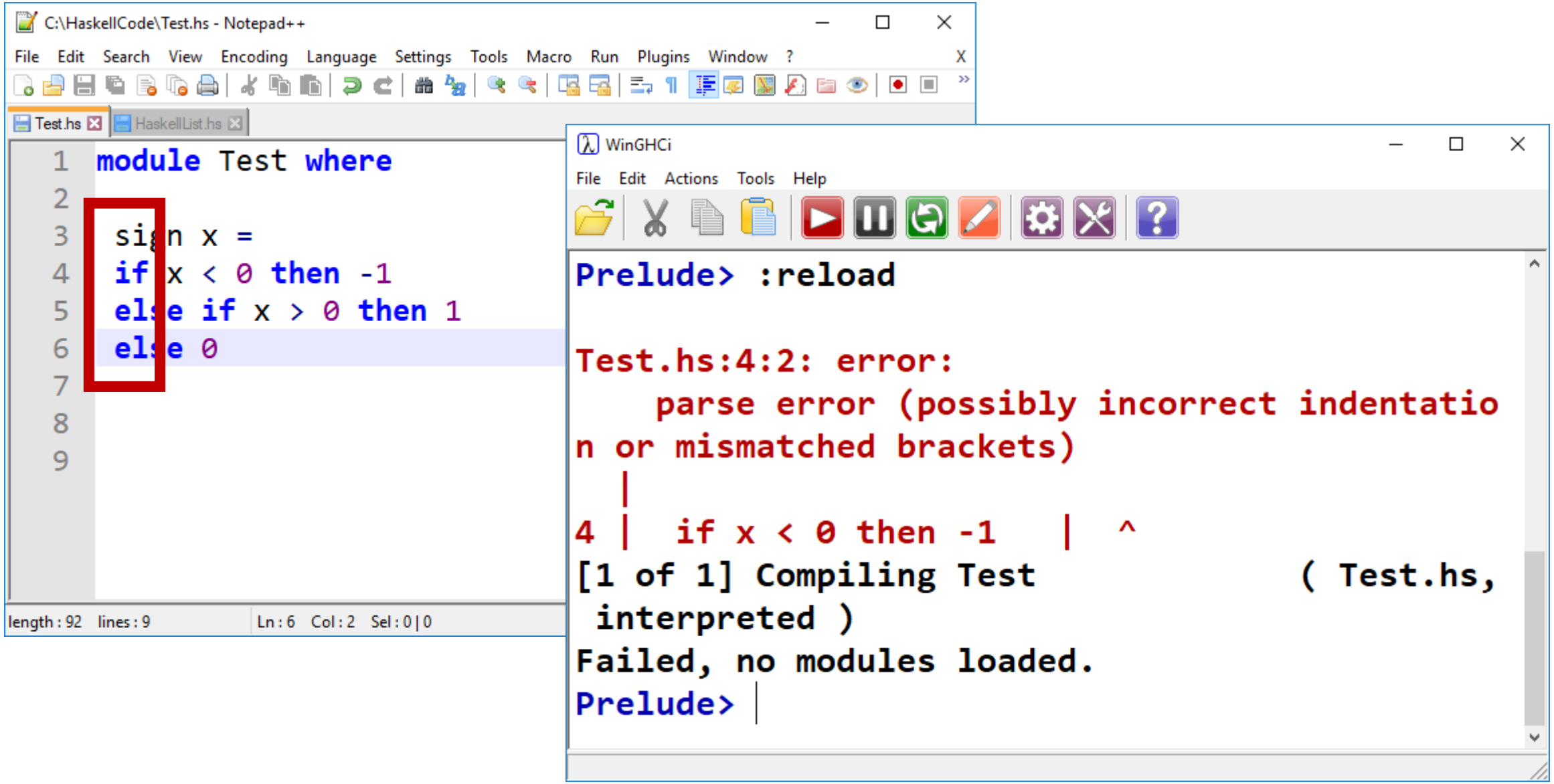
- If/else construct in Haskell is similar to most other languages.
- It must include a **then** and an **else**

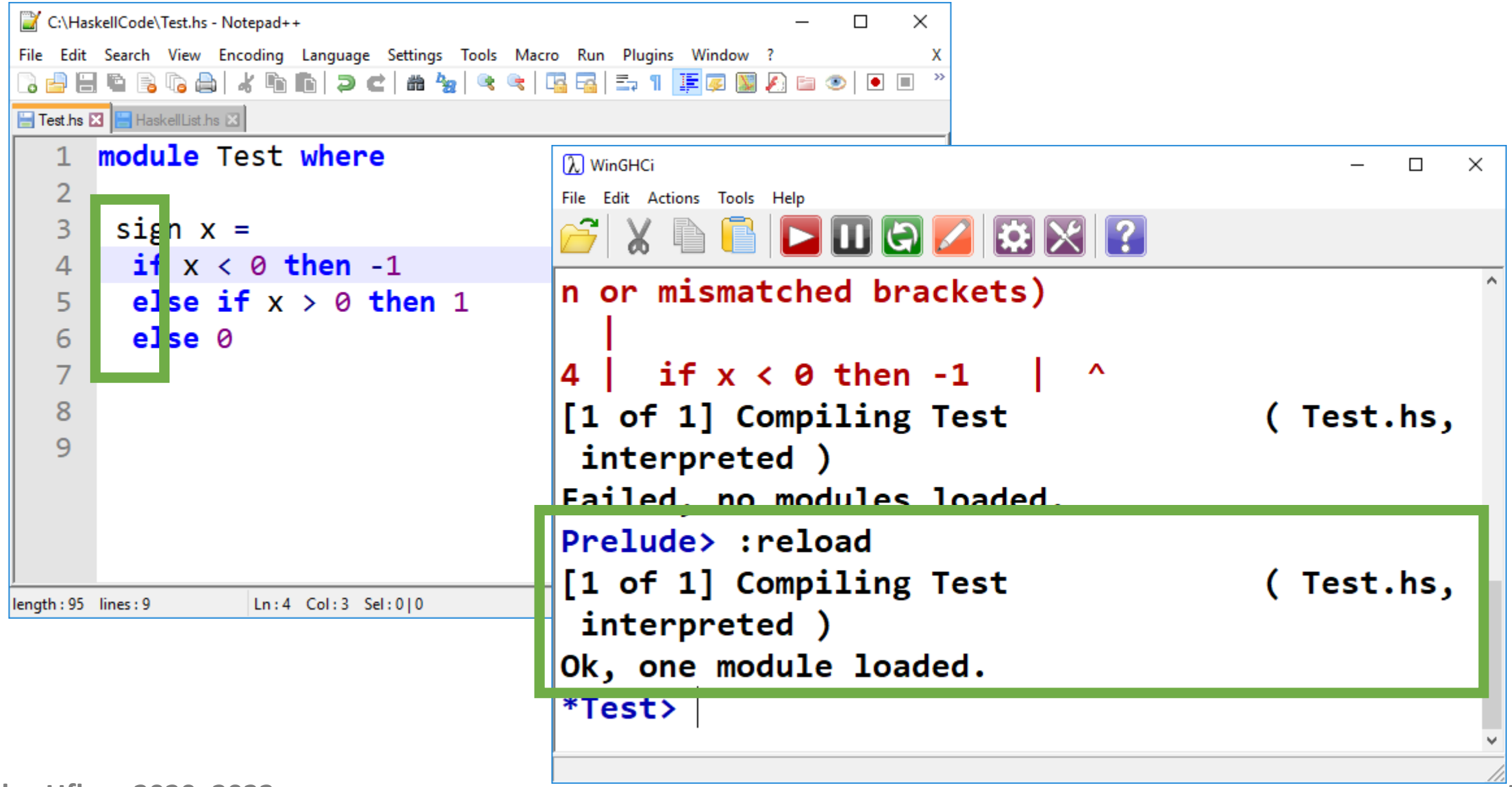
Control Structures

`if then else if then else`

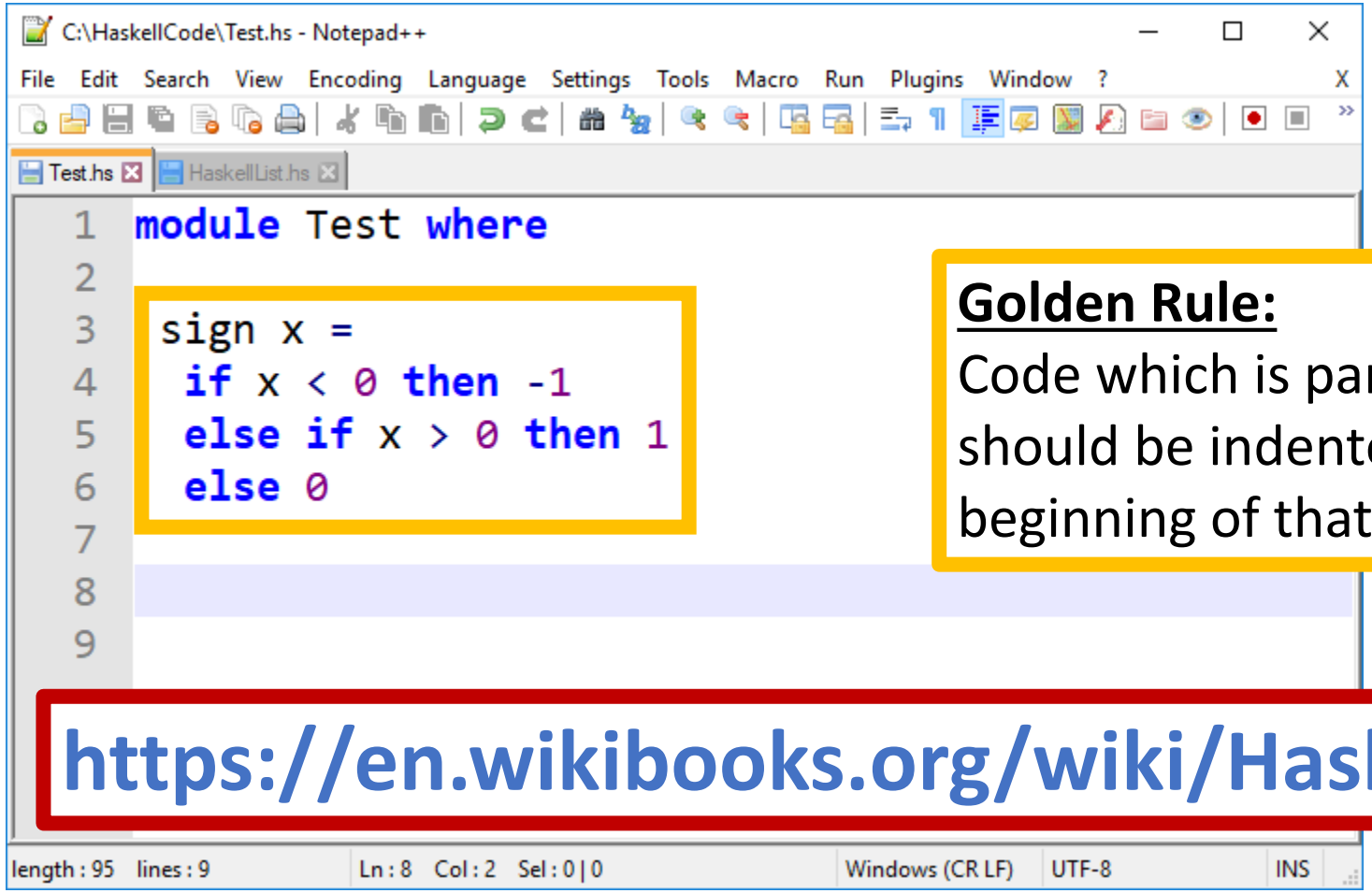


We can now format across multiple lines.
HOWEVER: Indentation matters in Haskell!





Indenting in Haskell



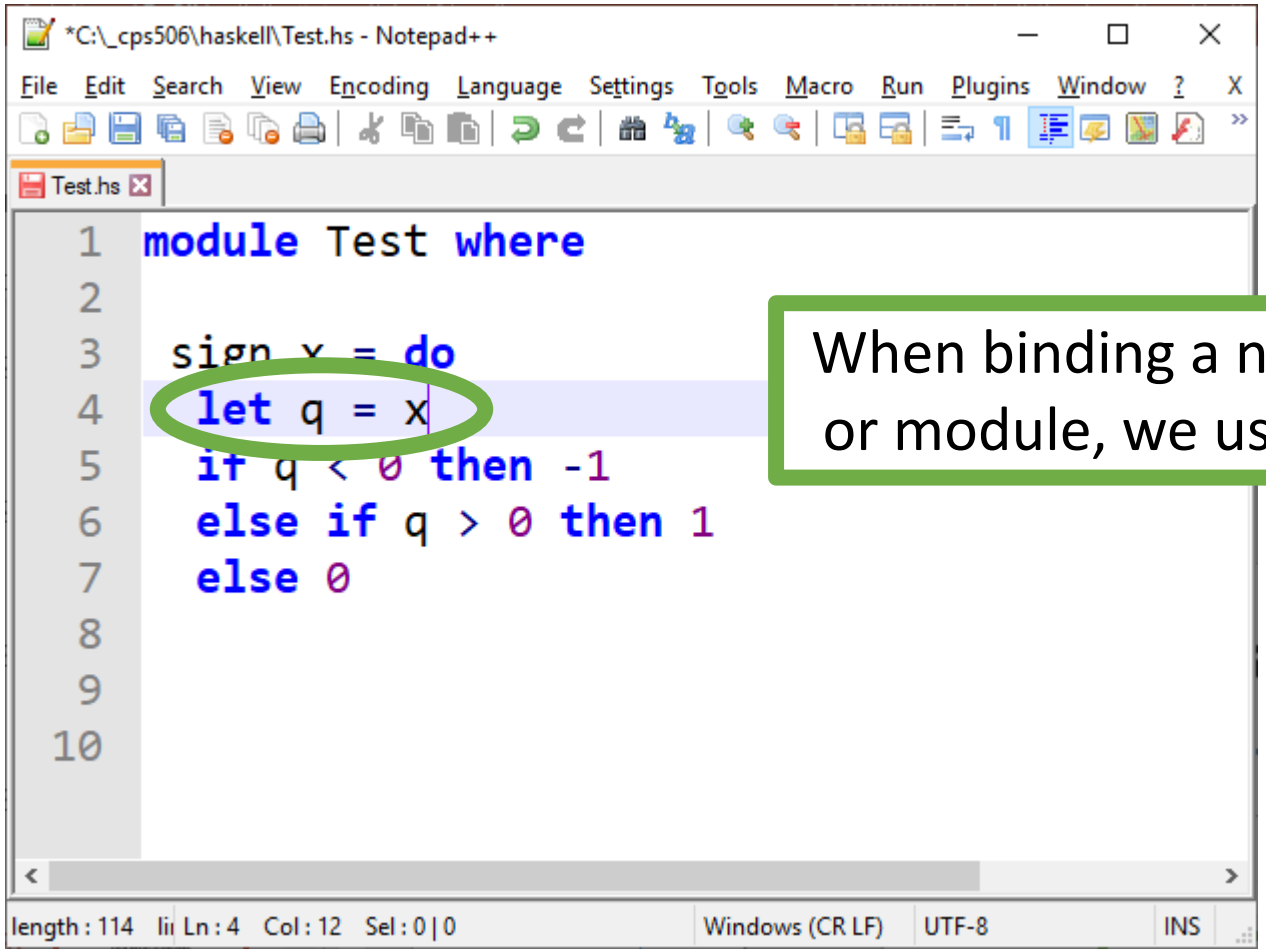
Golden Rule:

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

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

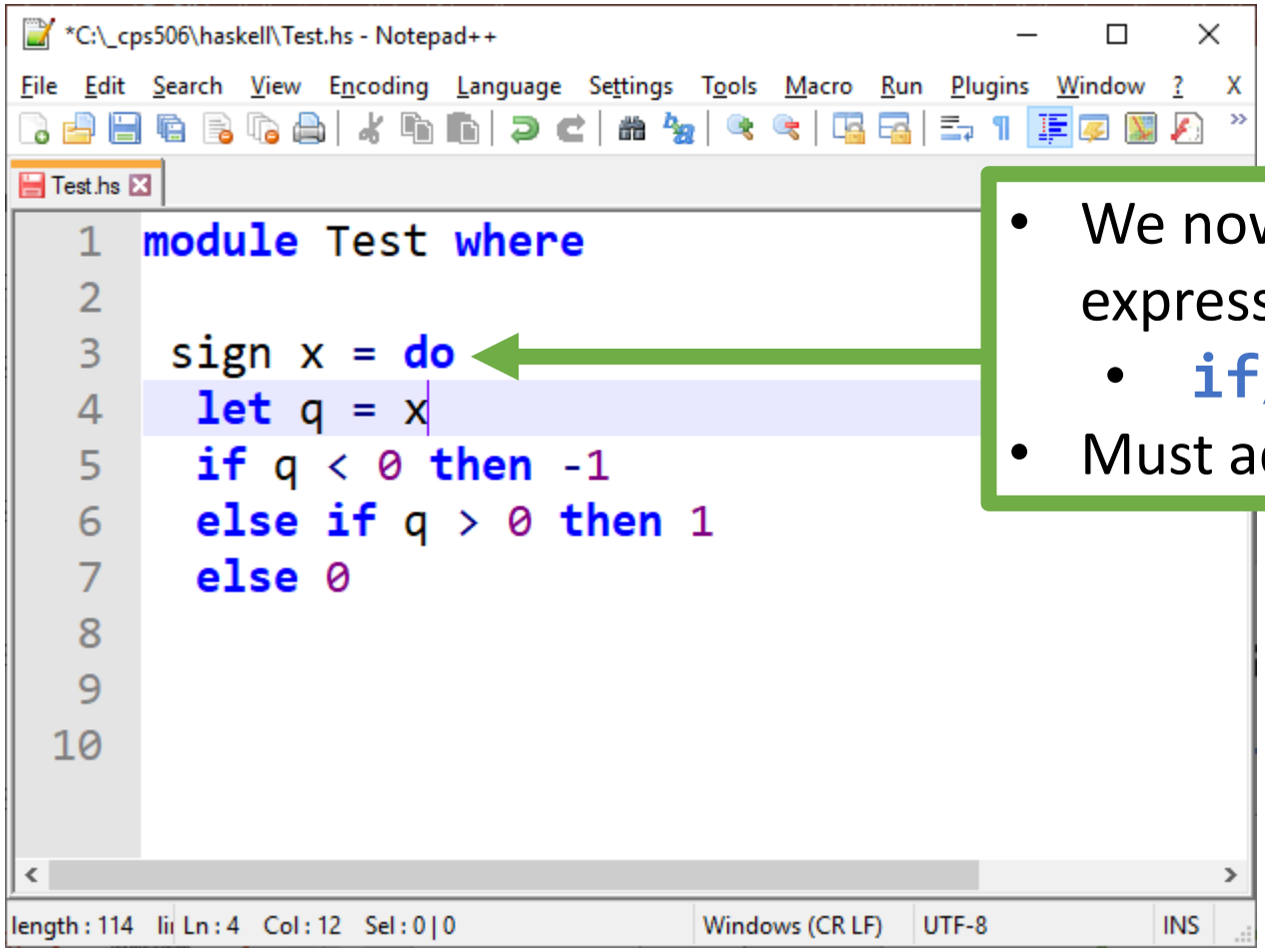
If all that weren’t enough, Tabs don’t work properly unless they’re 8 spaces exactly.

Local Names in Functions



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

Multiple Expressions



- 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

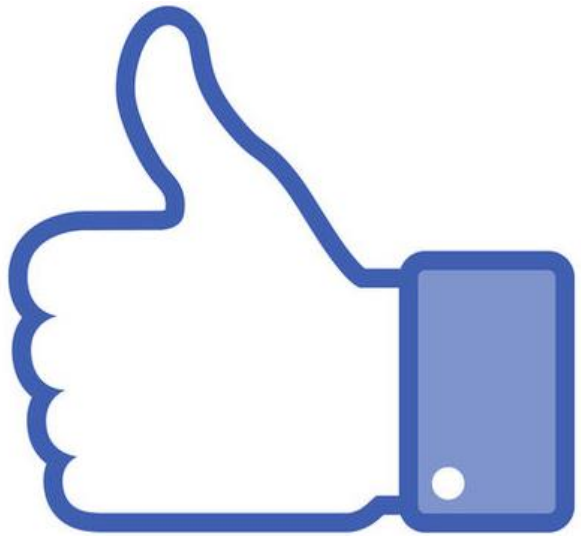
```
1  module Test where
2
3   sign x = do
4    let q = x
5    if q < 0 then -1
6    else if q > 0 then 1
7    else 0
8
9
10
```

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

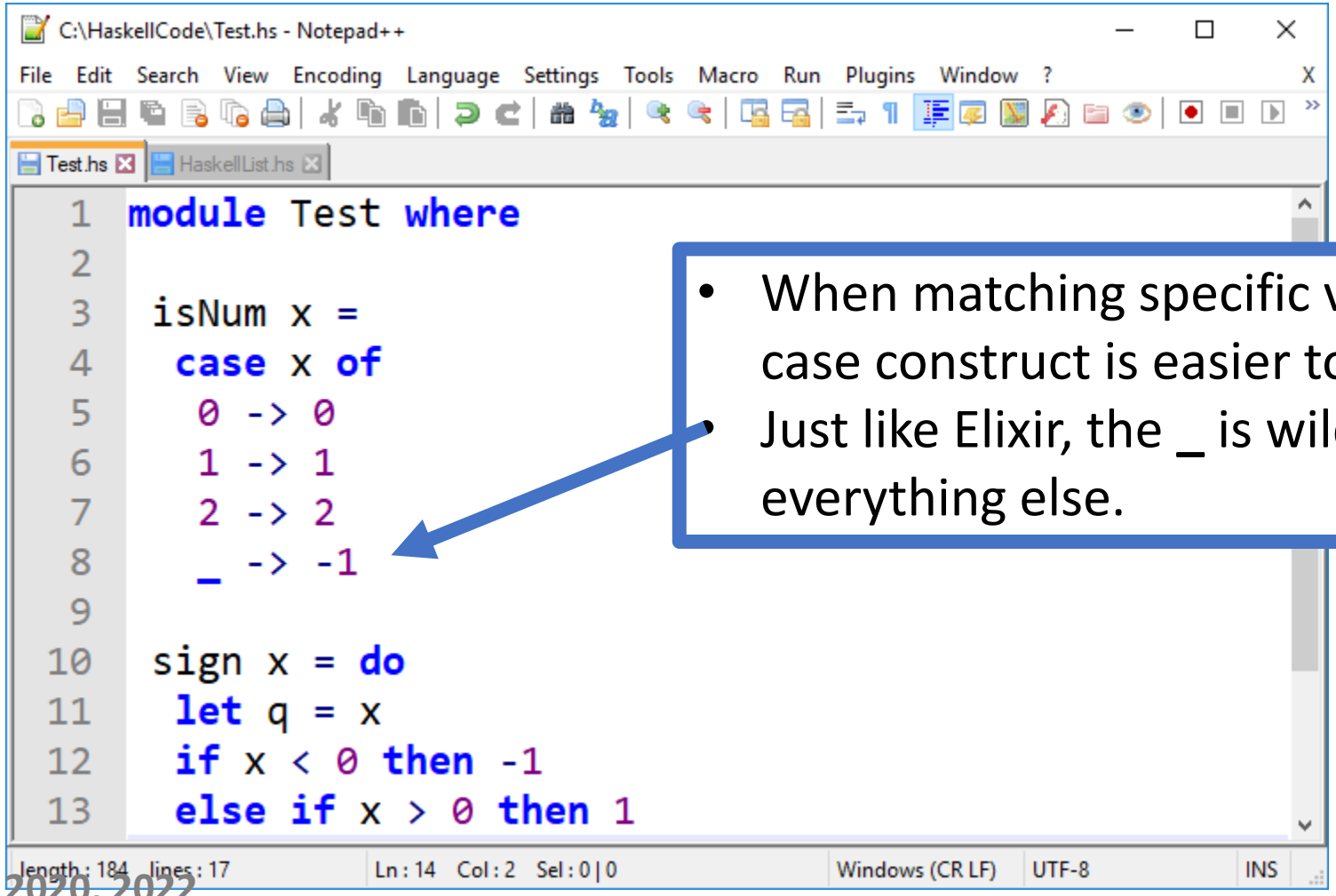
WinGHCi

File Edit Actions Tools Help

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

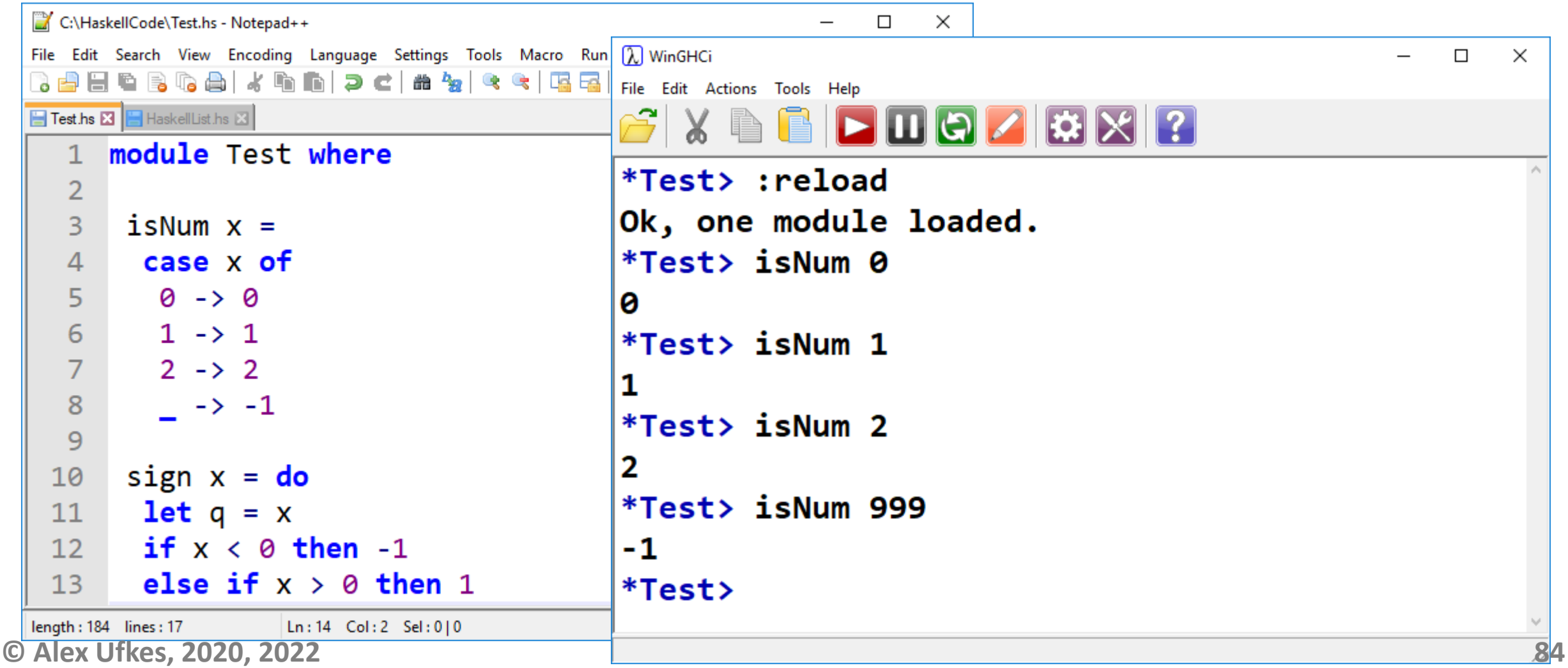


Case Expression

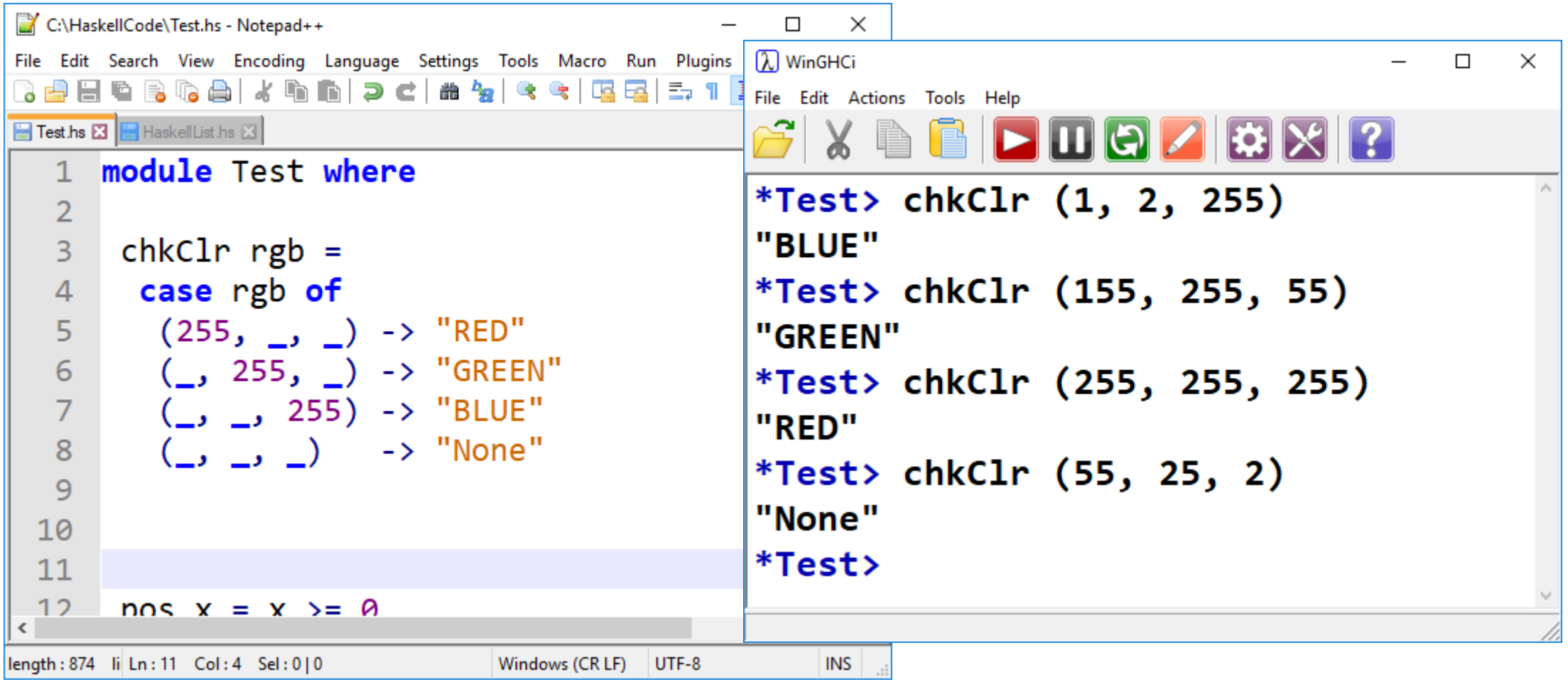


- When matching specific values, a case construct is easier to write.
- Just like Elixir, the _ is wild. It catches everything else.

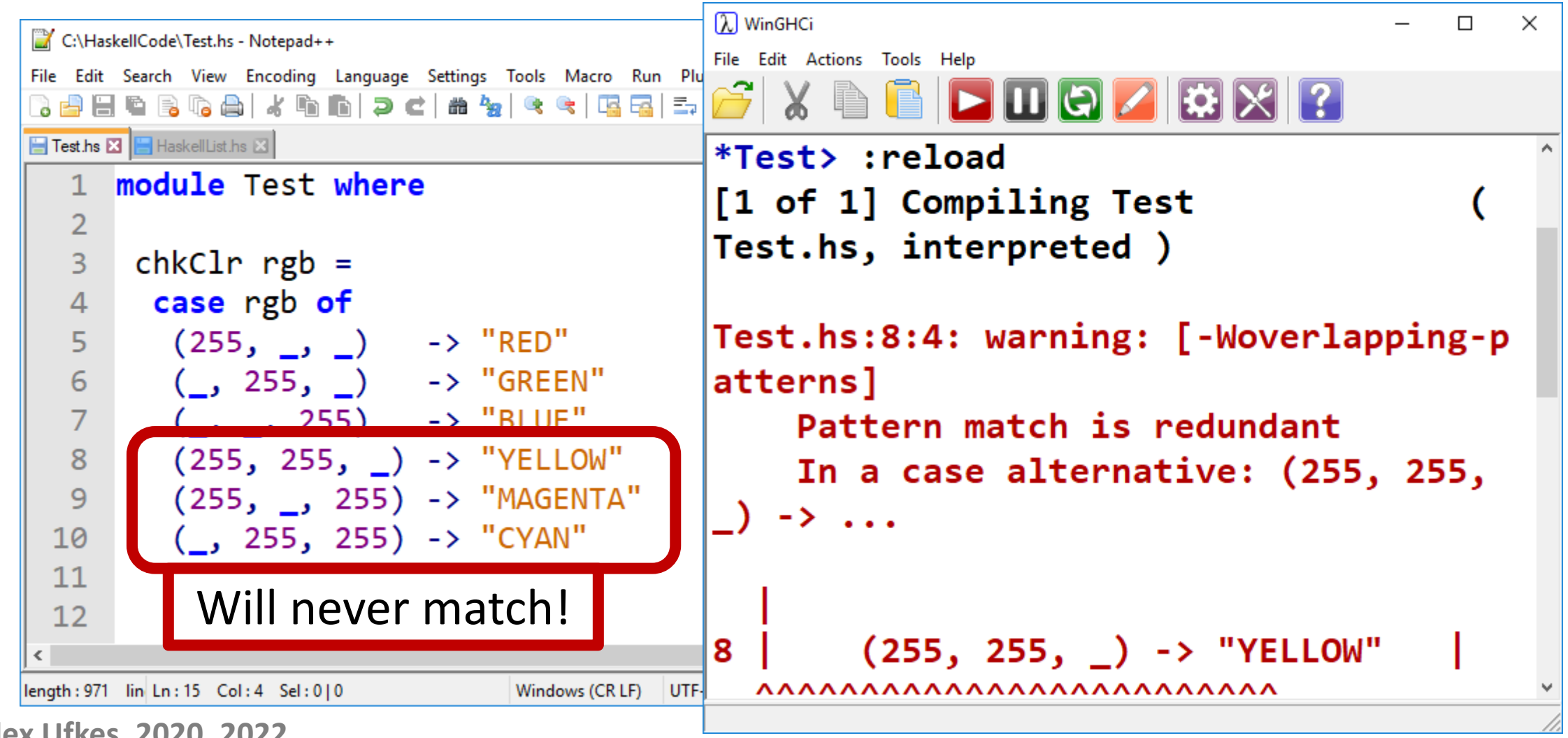
Case Expression



Pattern Matching: Case



Pattern Matching: Case



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

File Edit Search View Encoding Language Settings Tools Macro Run Plugins

Test.hs HaskellList.hs

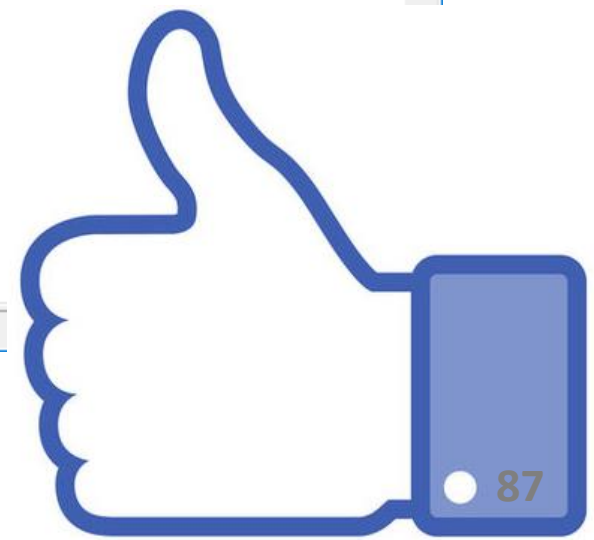
```
1  module Test where
2
3   chkClr rgb =
4    case rgb of
5     (255, 255, _) -> "YELLOW"
6     (255, _, 255) -> "MAGENTA"
7     (_, 255, 255) -> "CYAN"
8     (255, _, _)   -> "RED"
9     (_, 255, _)   -> "GREEN"
10    (_, _, 255)   -> "BLUE"
11    (_, _, _)     -> "None"
12
```

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

WinGHCi

File Edit Actions Tools Help

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



Unlike Elixir…

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

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

Test.hs | HaskellList.hs

```
1  module Test where
2
3   chkClr rgb =
4    case rgb of
5     (255, _, _) -> "RED"
6     (_, 255, _) -> "GREEN"
7     (_, _, 255) -> "BLUE"
8     x -> "None"
9
10
```

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

WinGHCi

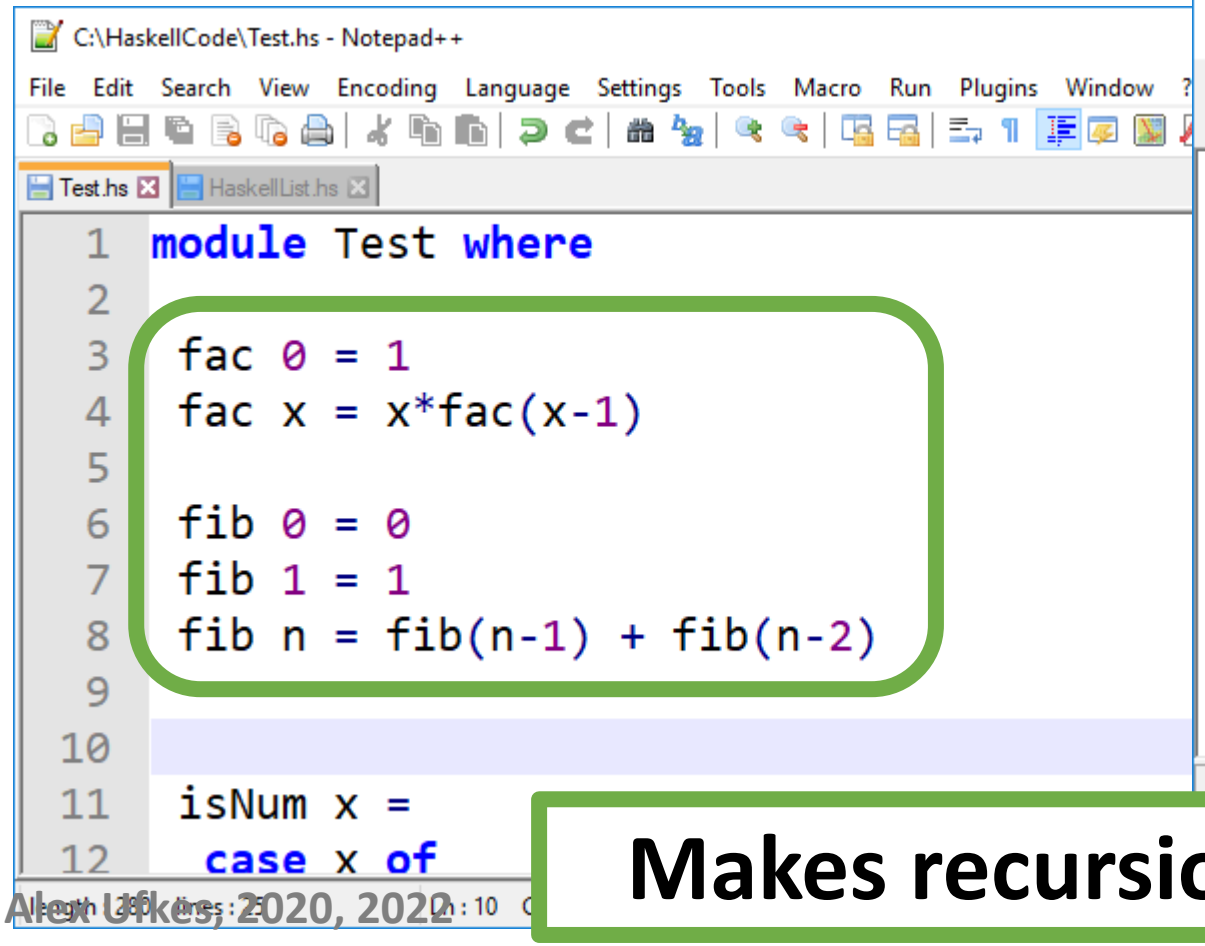
File Edit Actions Tools Help

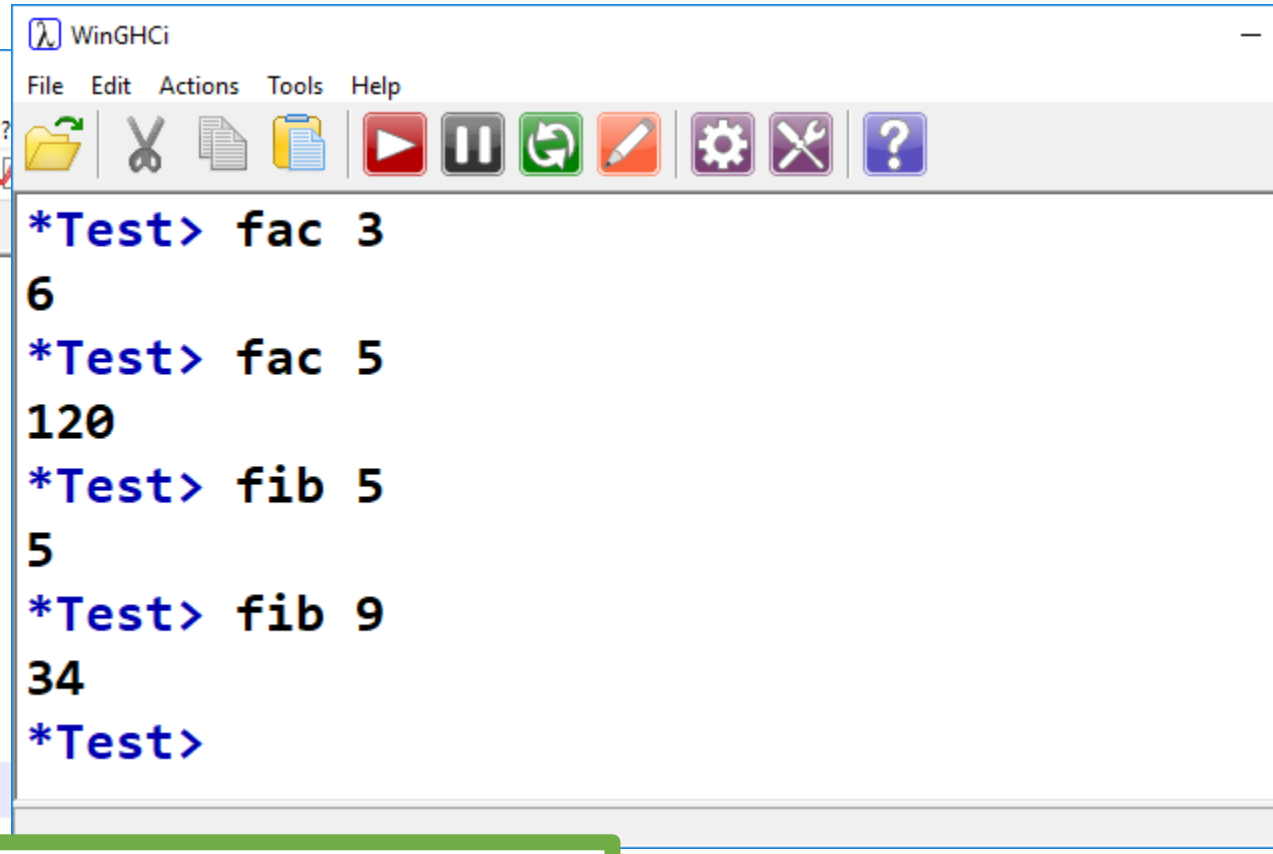
```
*Test> chkClr (3, 3, 3)
"None"
*Test> chkClr (3, 3)

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

Piecewise Functions

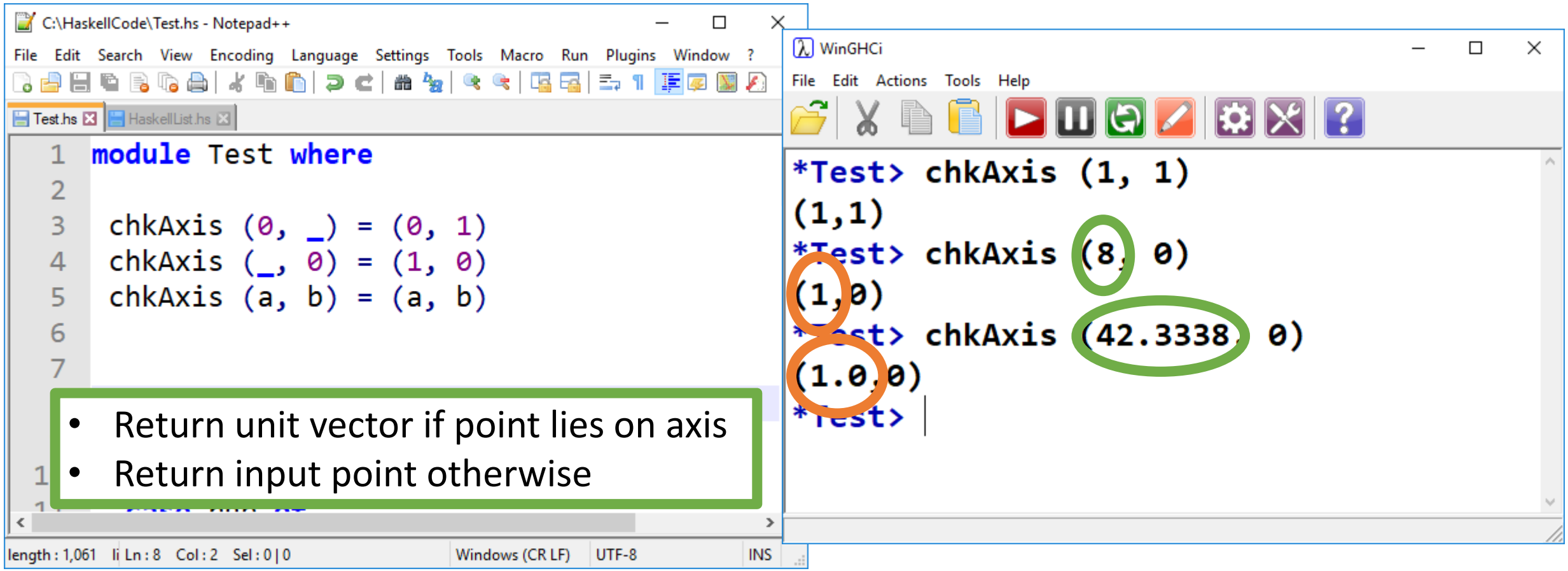
Just like Elixir's function signature pattern matching



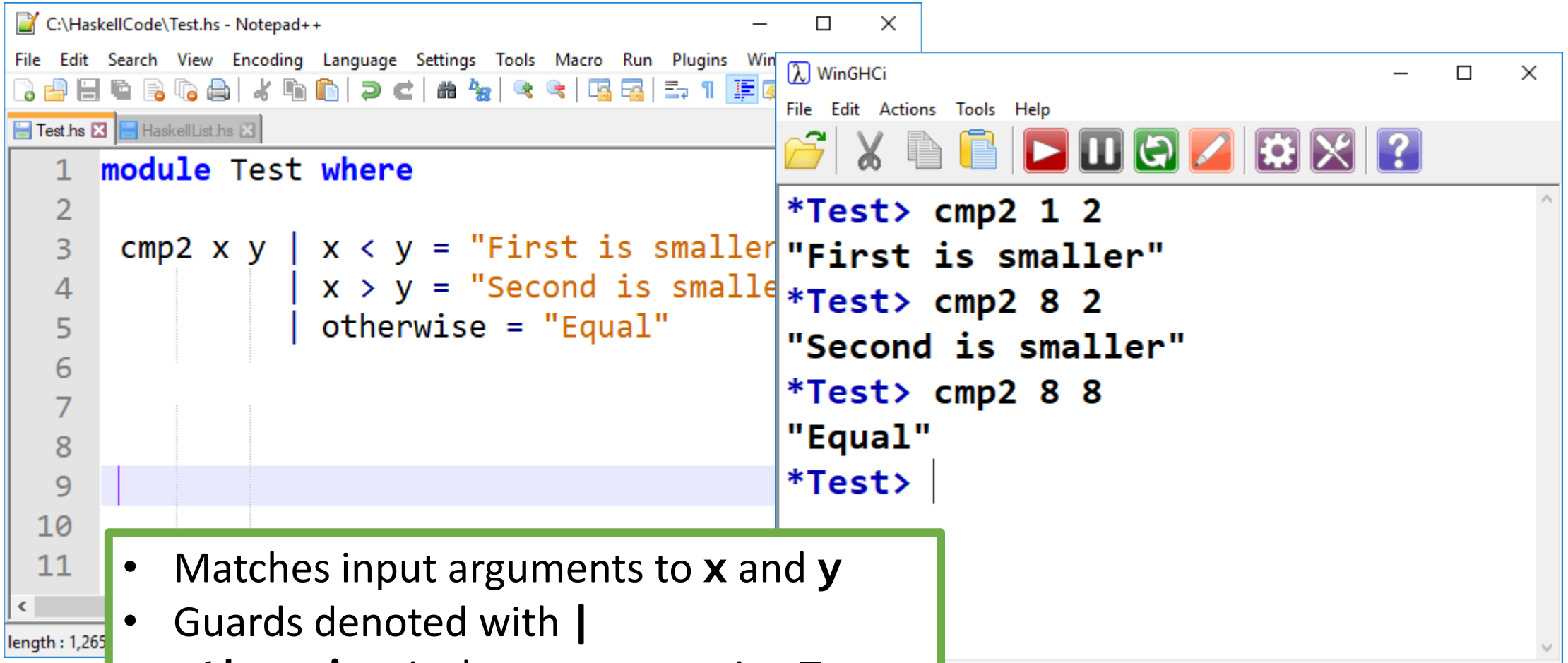


Makes recursion very easy!

Piecewise Functions

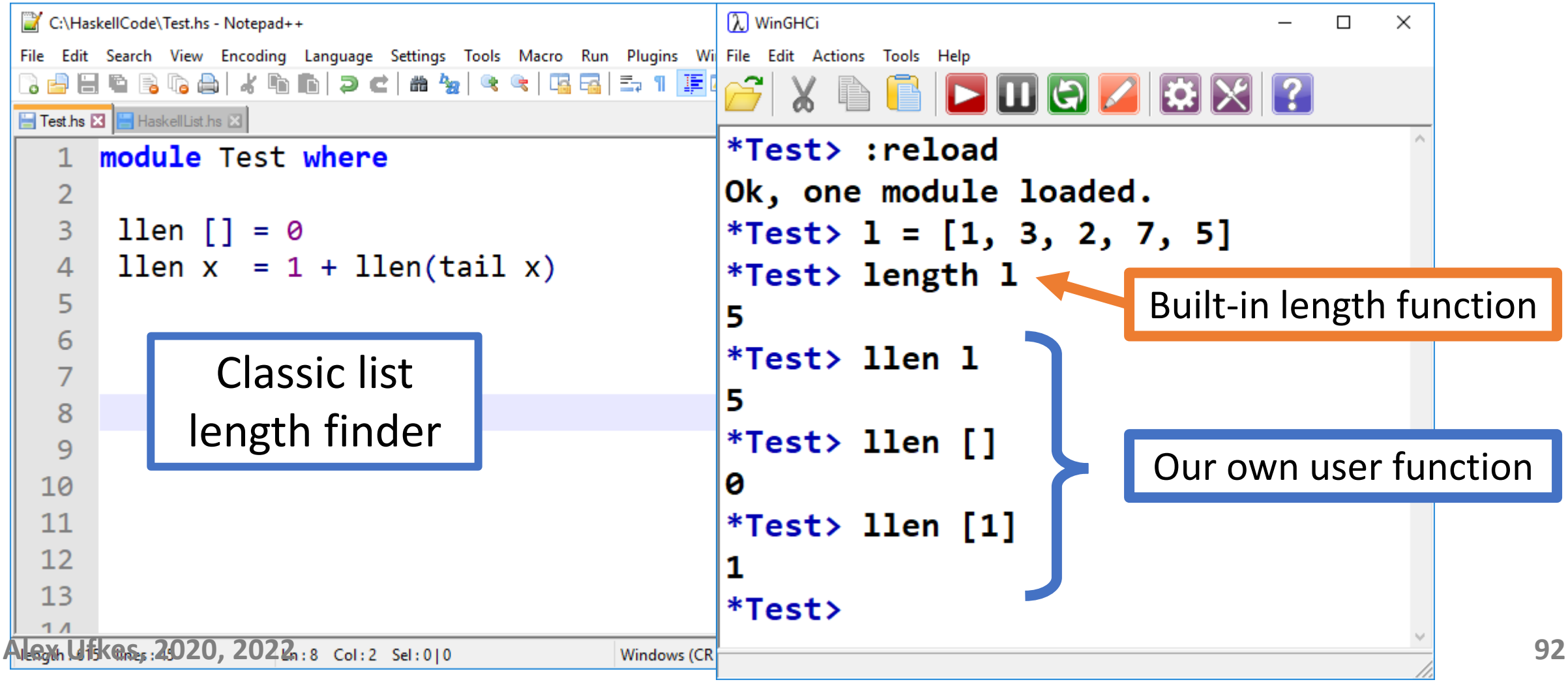


Functions: Guards



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

Recursion

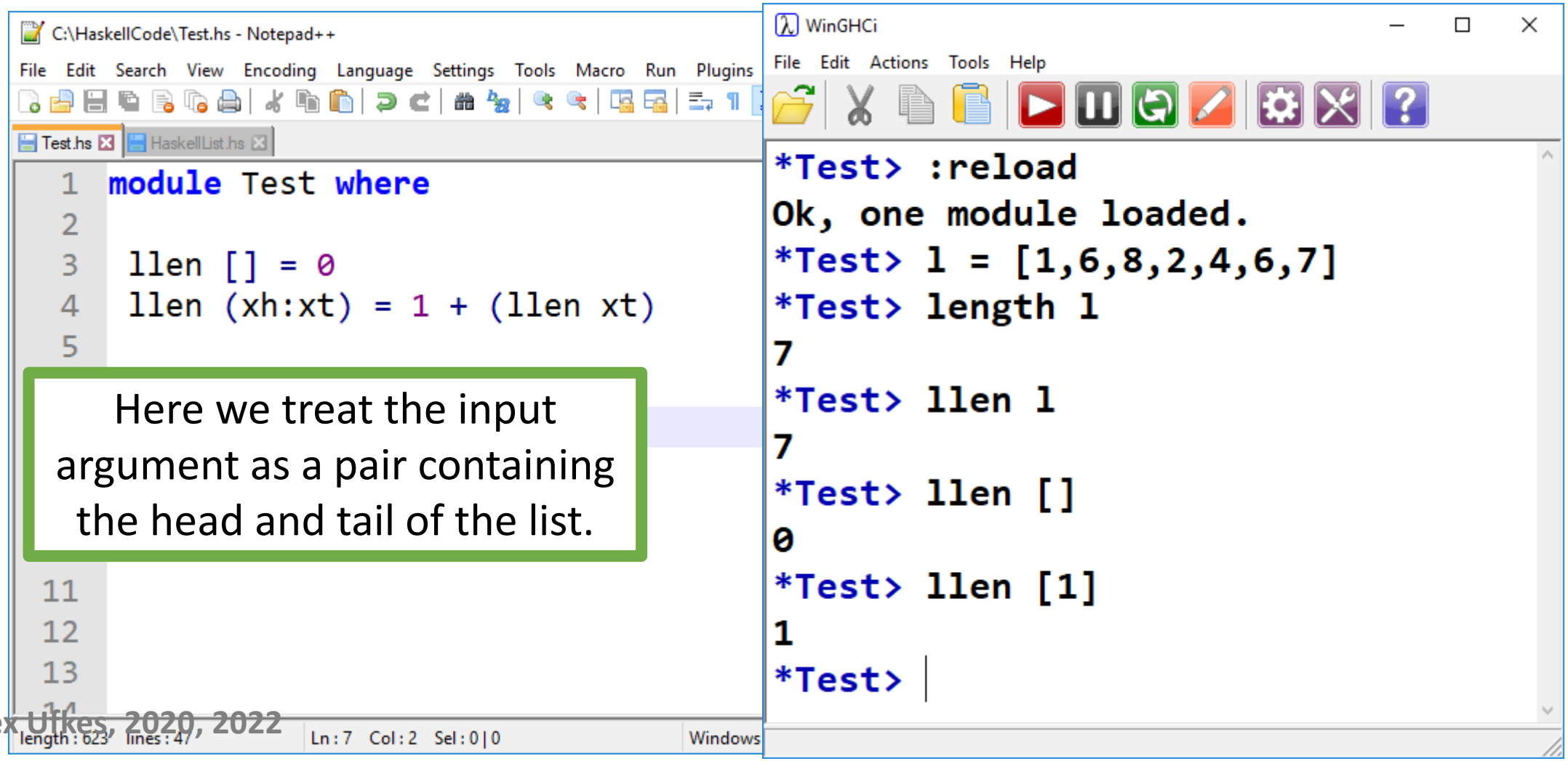


Tail Recursion?

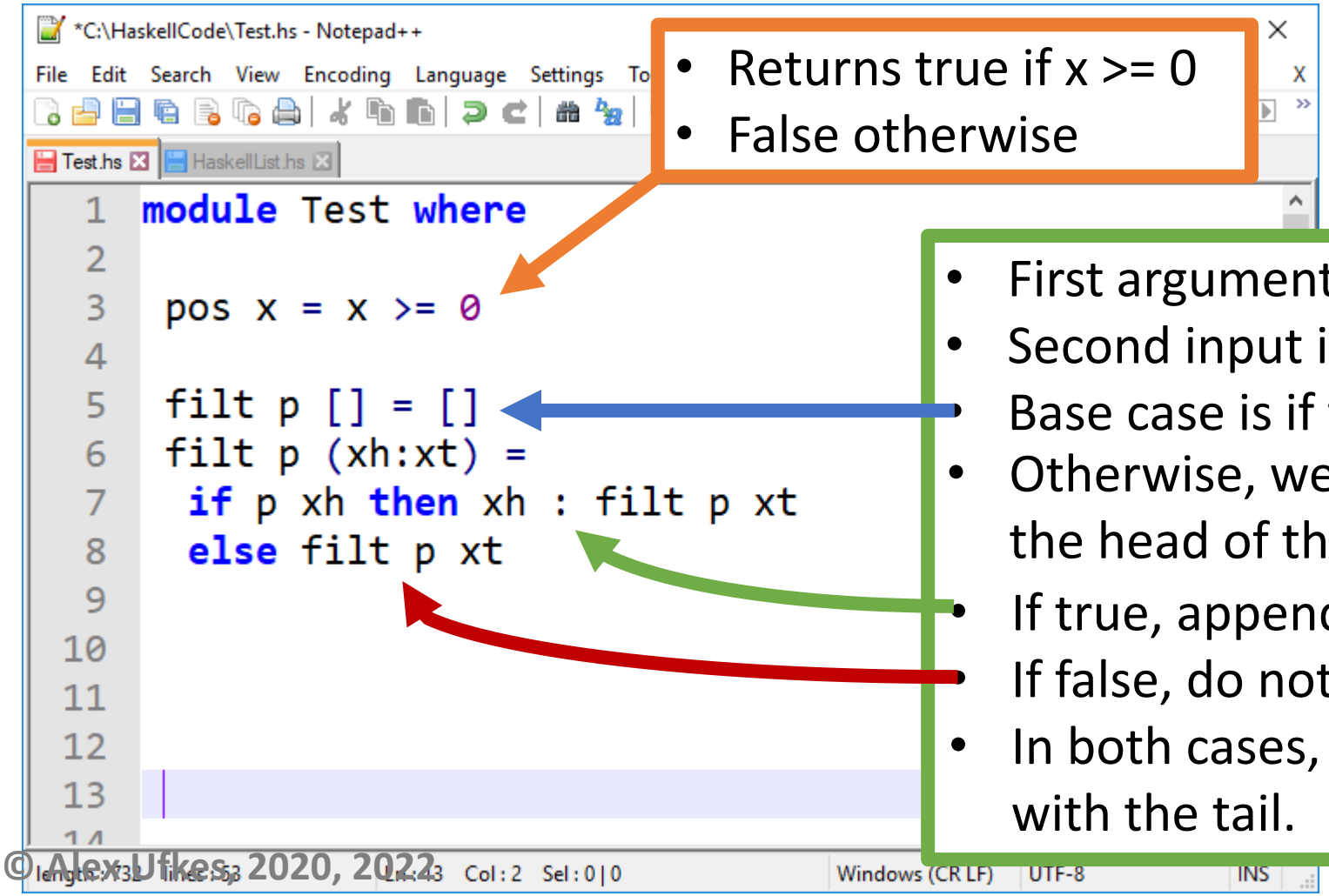
Less important in Haskell

- 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



Recursion: `filter`



- 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

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

Ln : 43 Col : 2 Sel : 0 | 0 Windows (CR LF)

WinGHCi

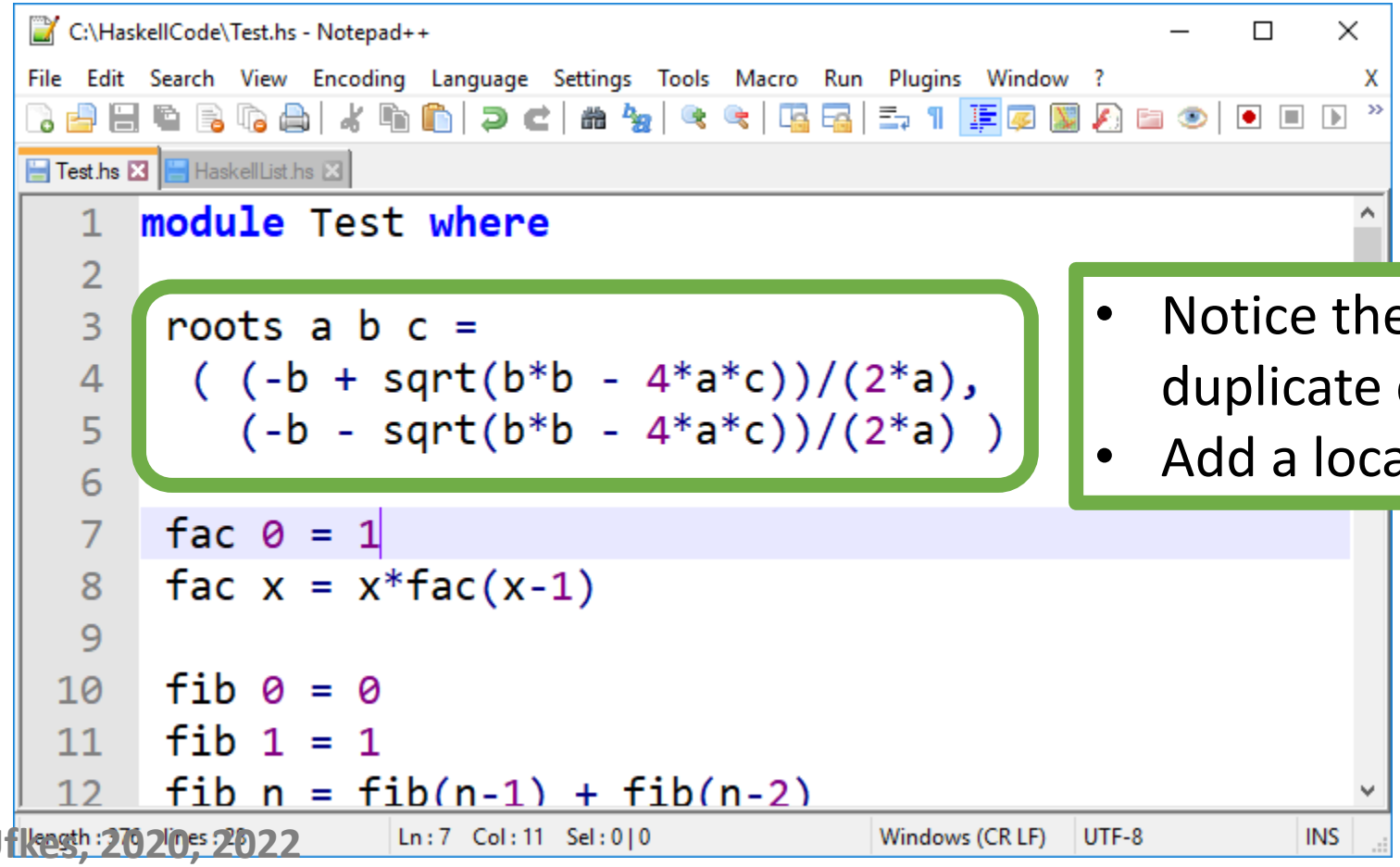
File Edit Actions Tools Help

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

Test **pos** function

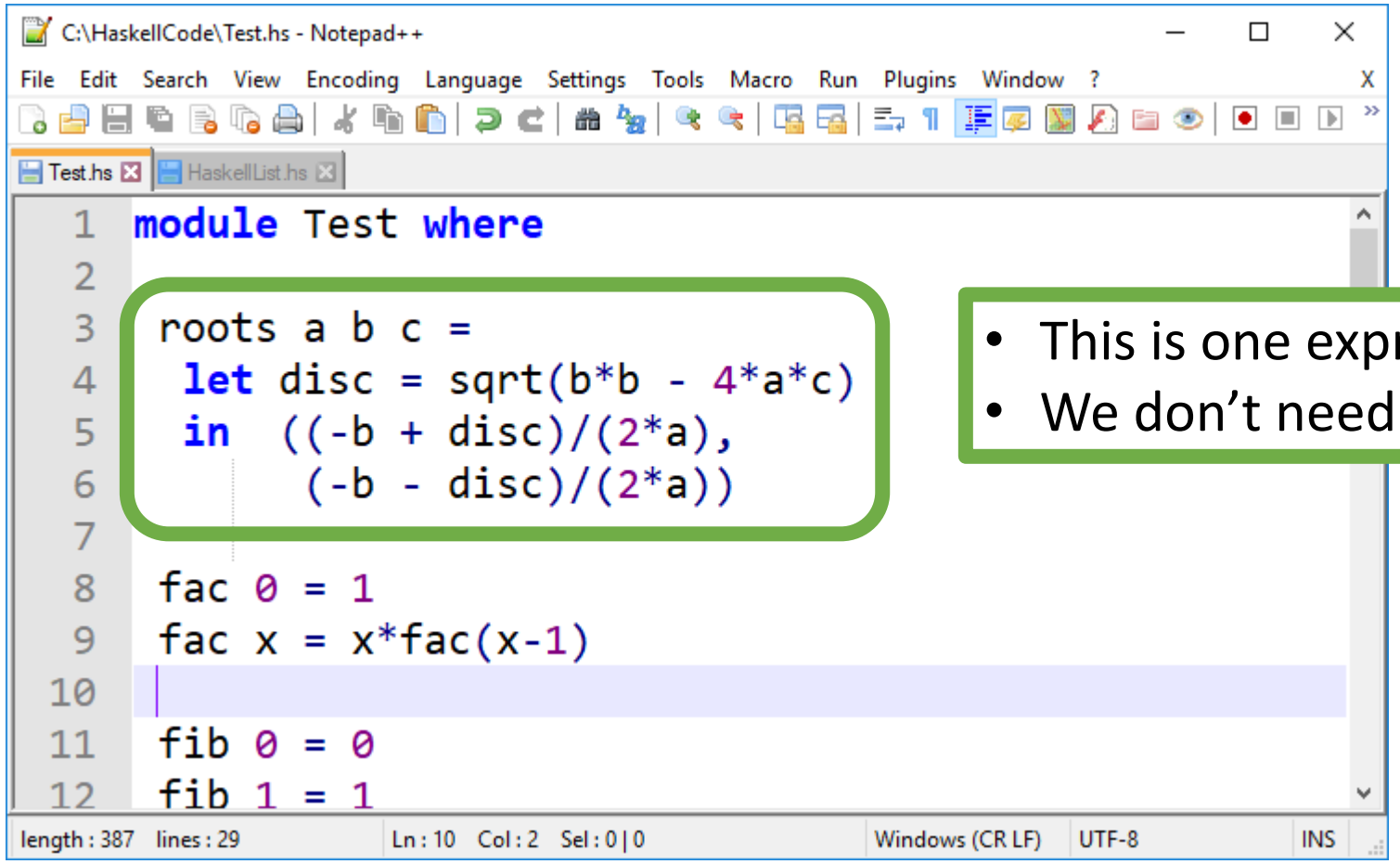
Return Multiple *Things*?

Lists/tuples to the rescue!

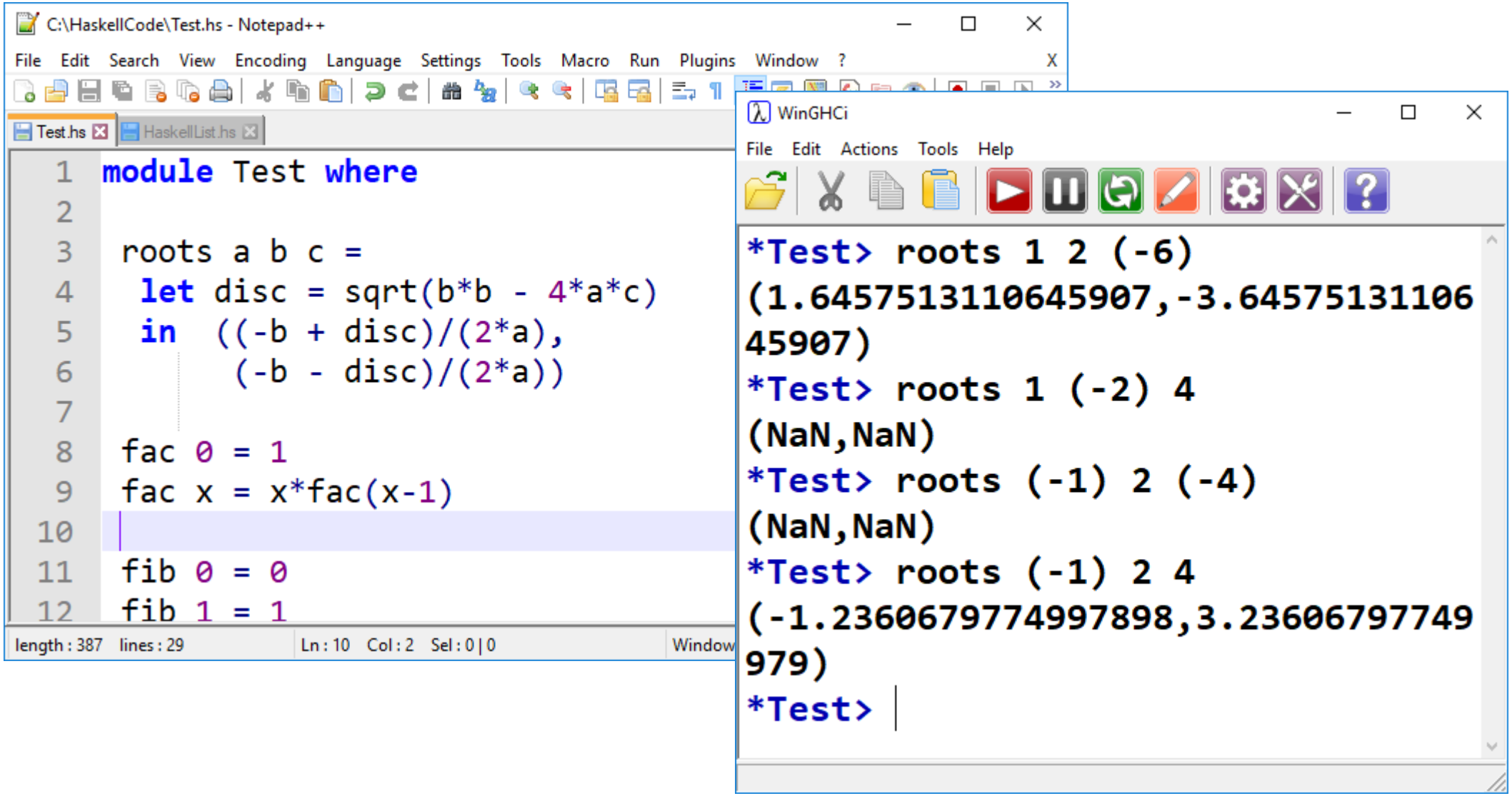


- Notice there is a lot of duplicate computation here.
- Add a local variable?

`let/in` Expression

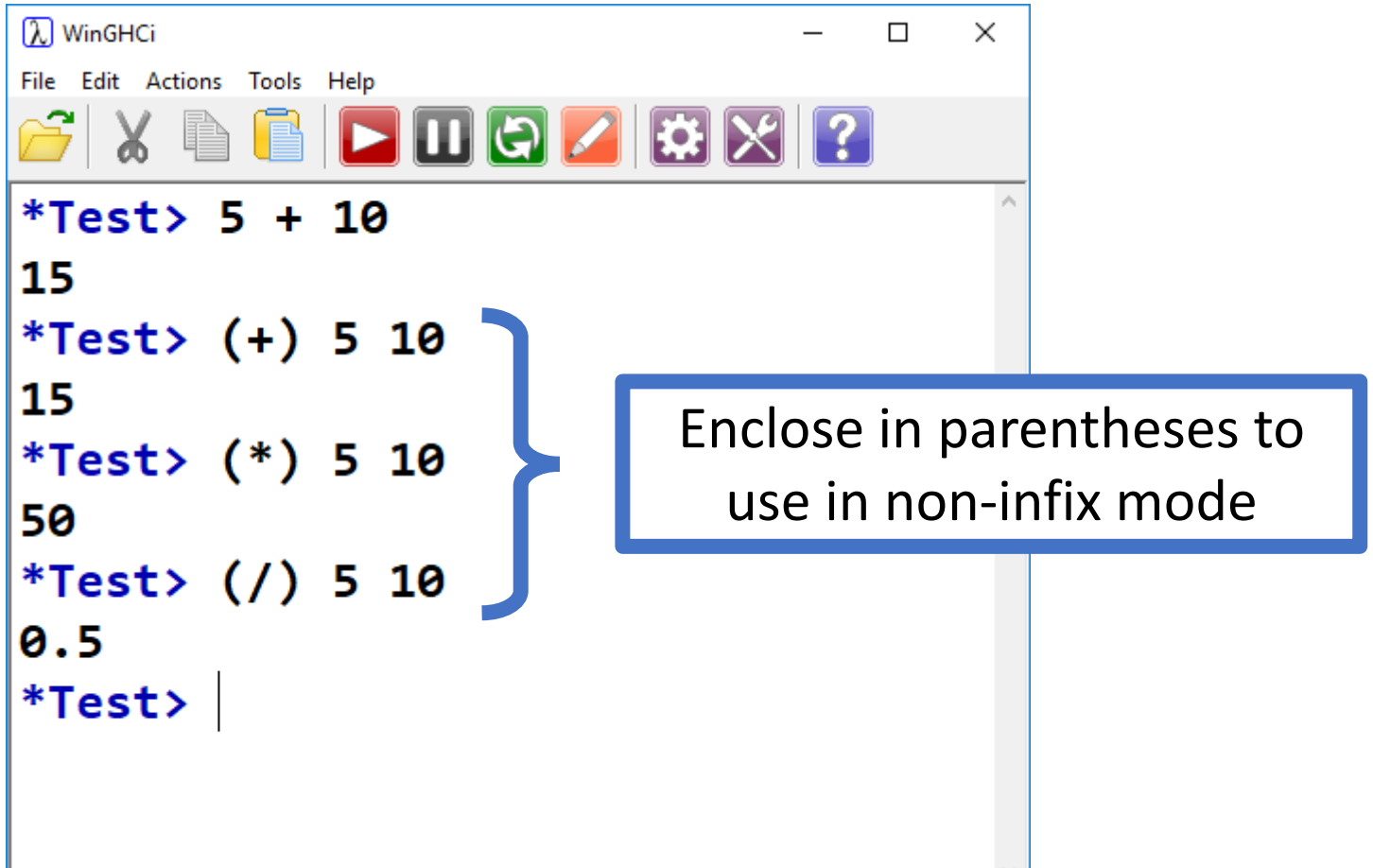


- This is one expression (`let`/`in`)
- We don't need to add `do` keyword

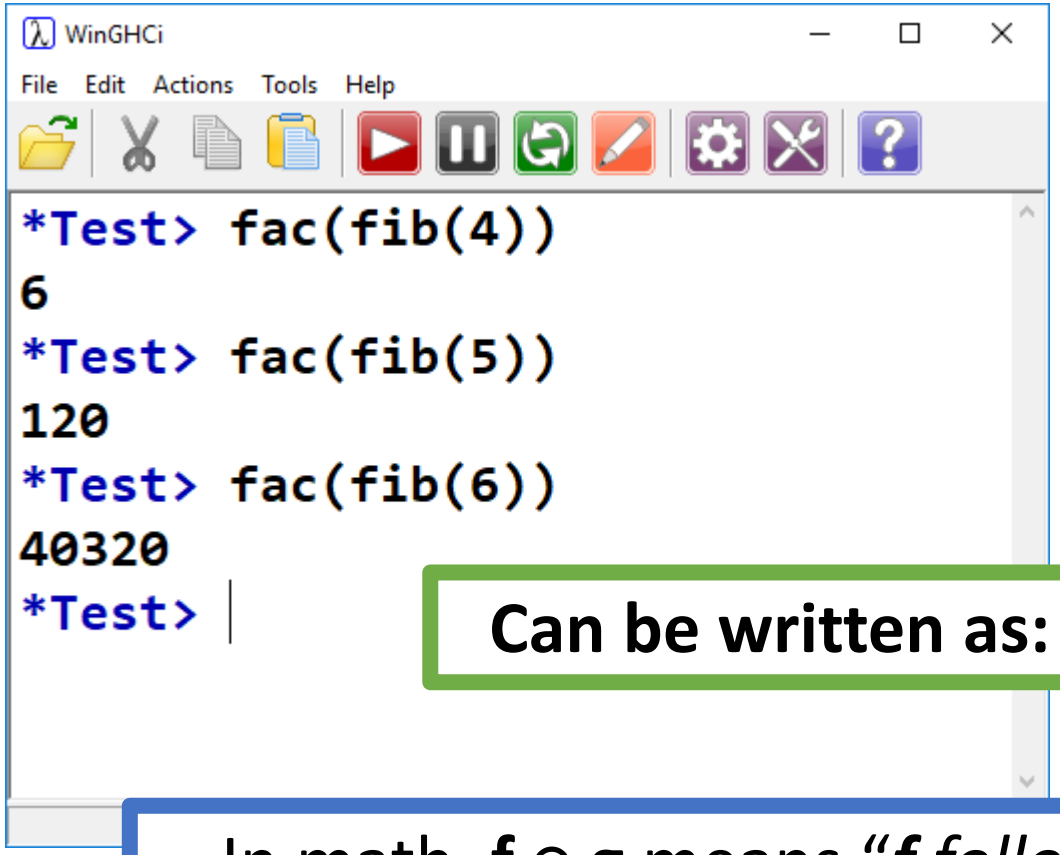


Infix Functions

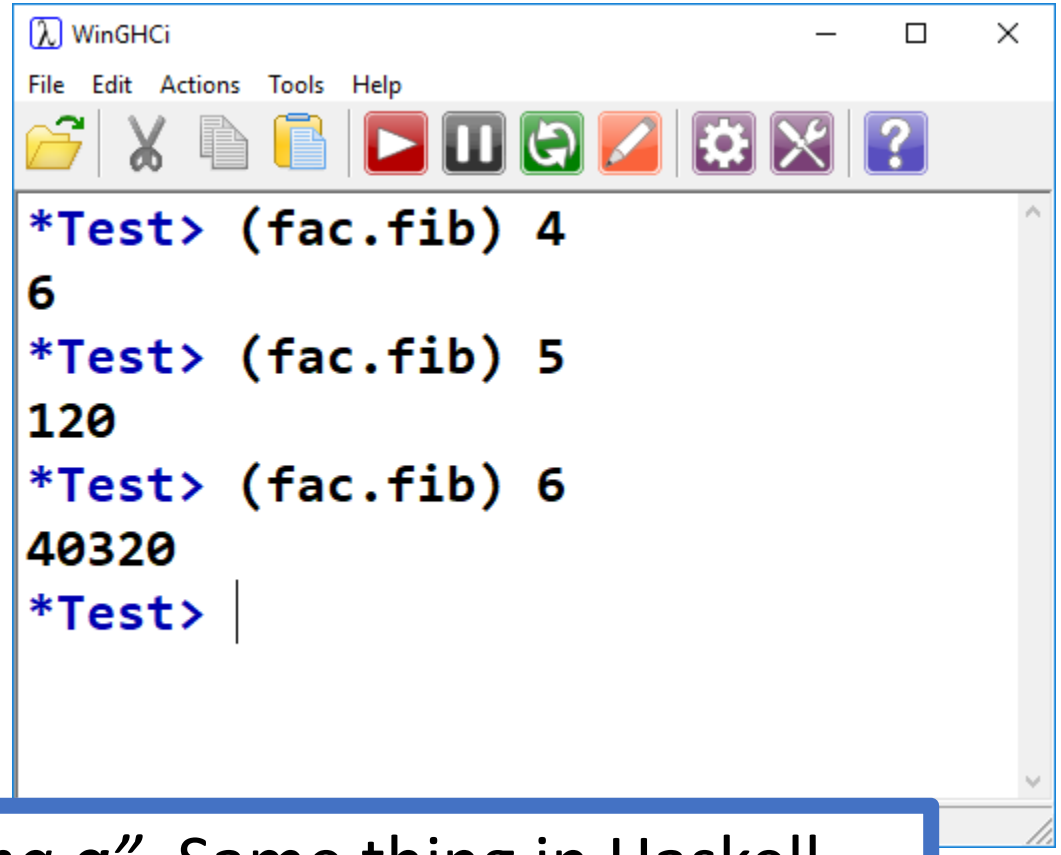
Use symbolic operators as functions:



Function Composition



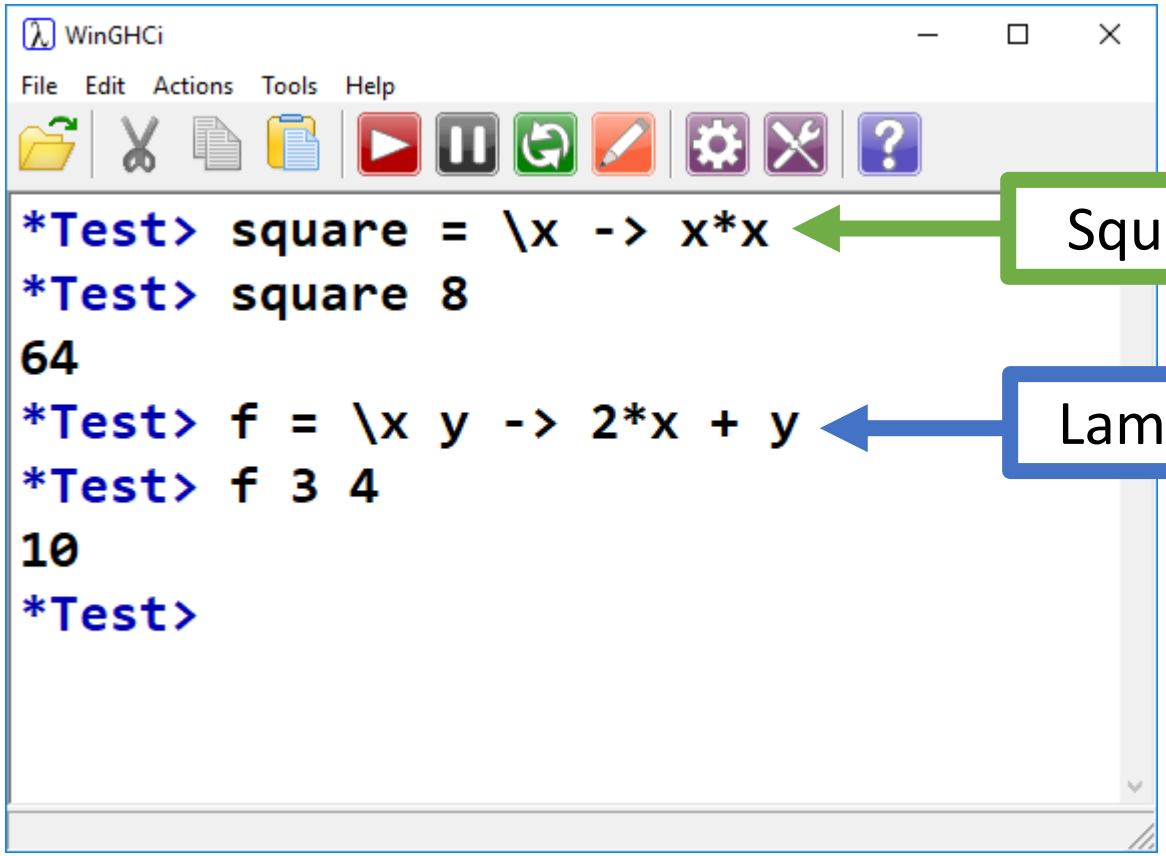
Can be written as:



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

Lambda Functions

Like anonymous functions in Elixir:

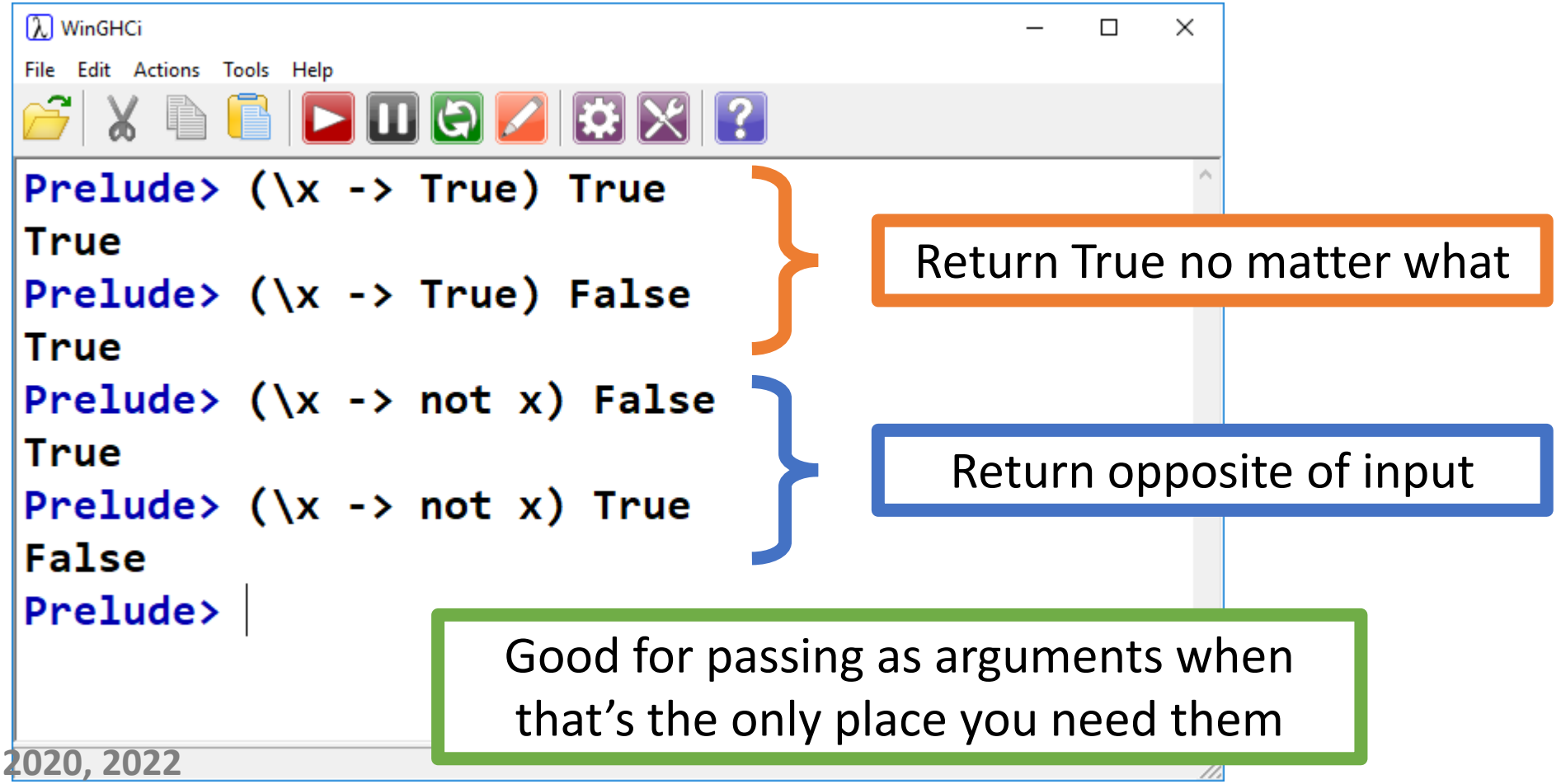


Square as Lambda function

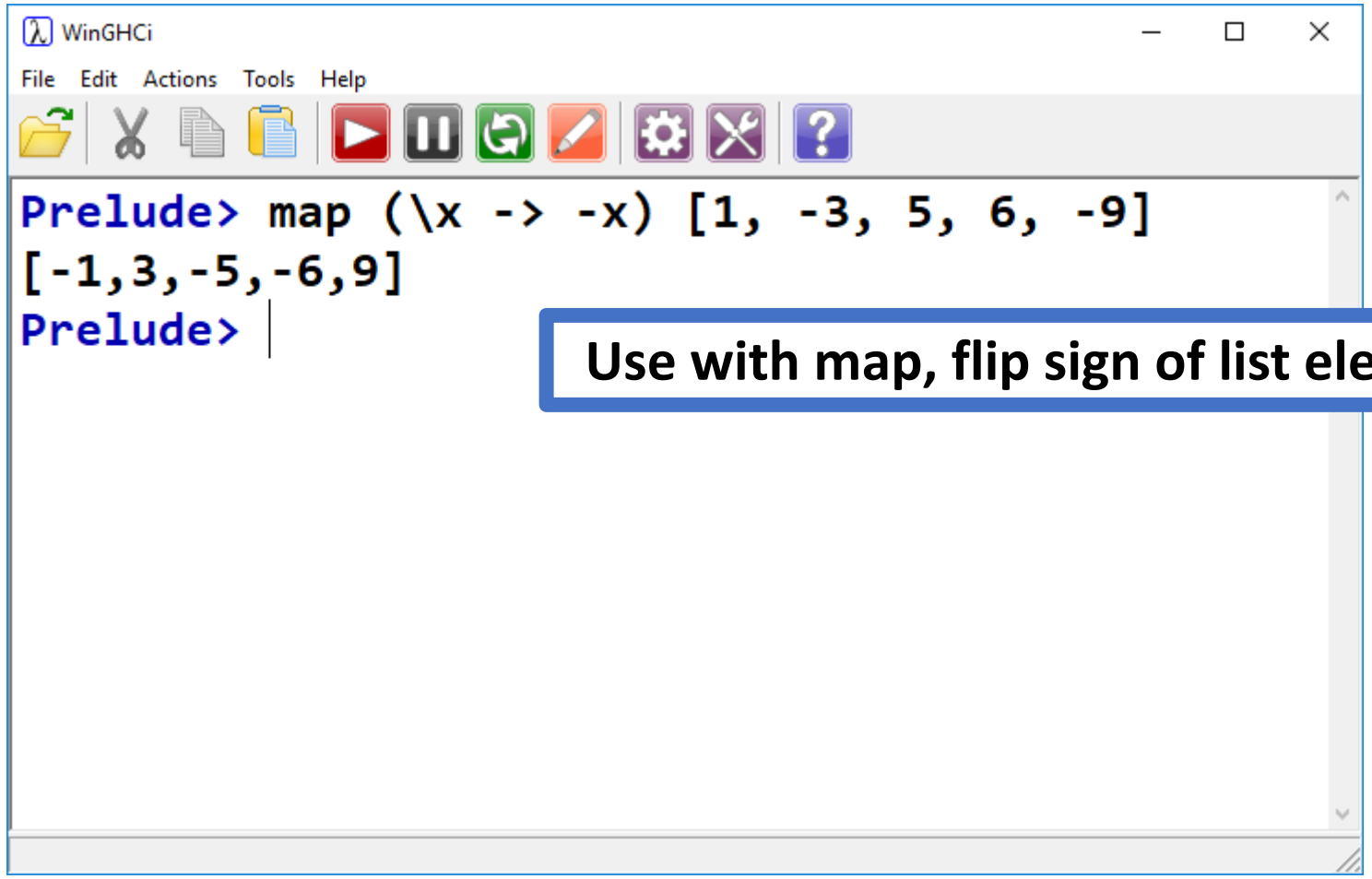
Lambda function with two args

Lambda Functions

They don't need names!



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



Haskell Tutorials/References:

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

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