

C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 6: Haskell intro, Haskell basics




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

        Alexander Ufkes 

[Content](#) [Grades](#) [Assessment](#)  [Communication](#)  [Resources](#)  [Classlist](#) [Course Admin](#)

Two languages down, two to go!

Today

Intro to Haskell

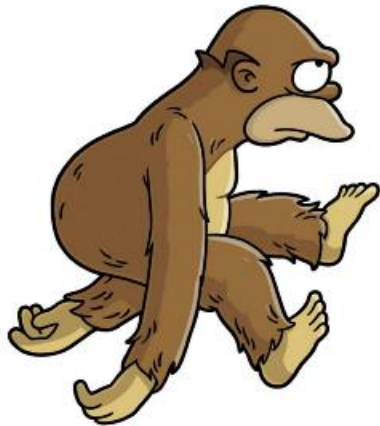
- Pure functional
- Haskell basics
- Functions
- Control flow



Functional Programming



MACHINE



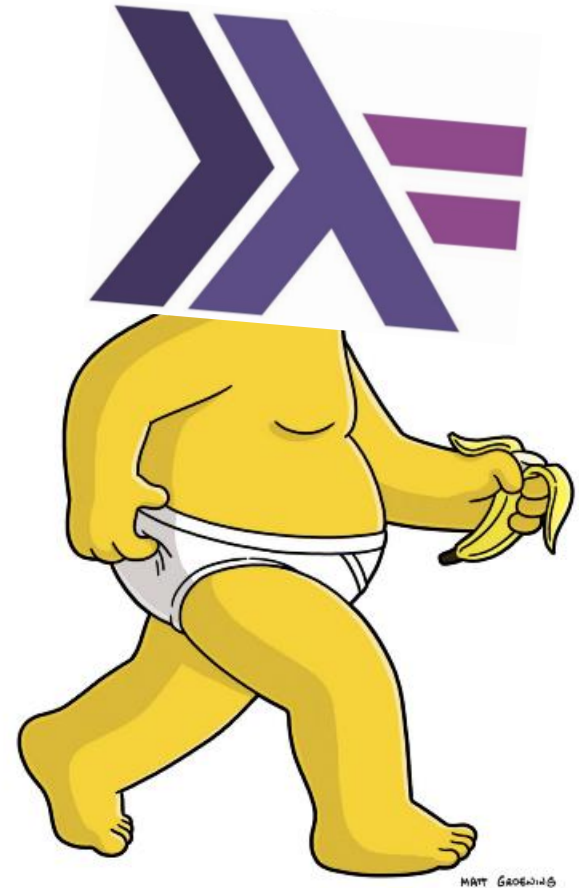
ASSEMBLY



PROCEDURAL

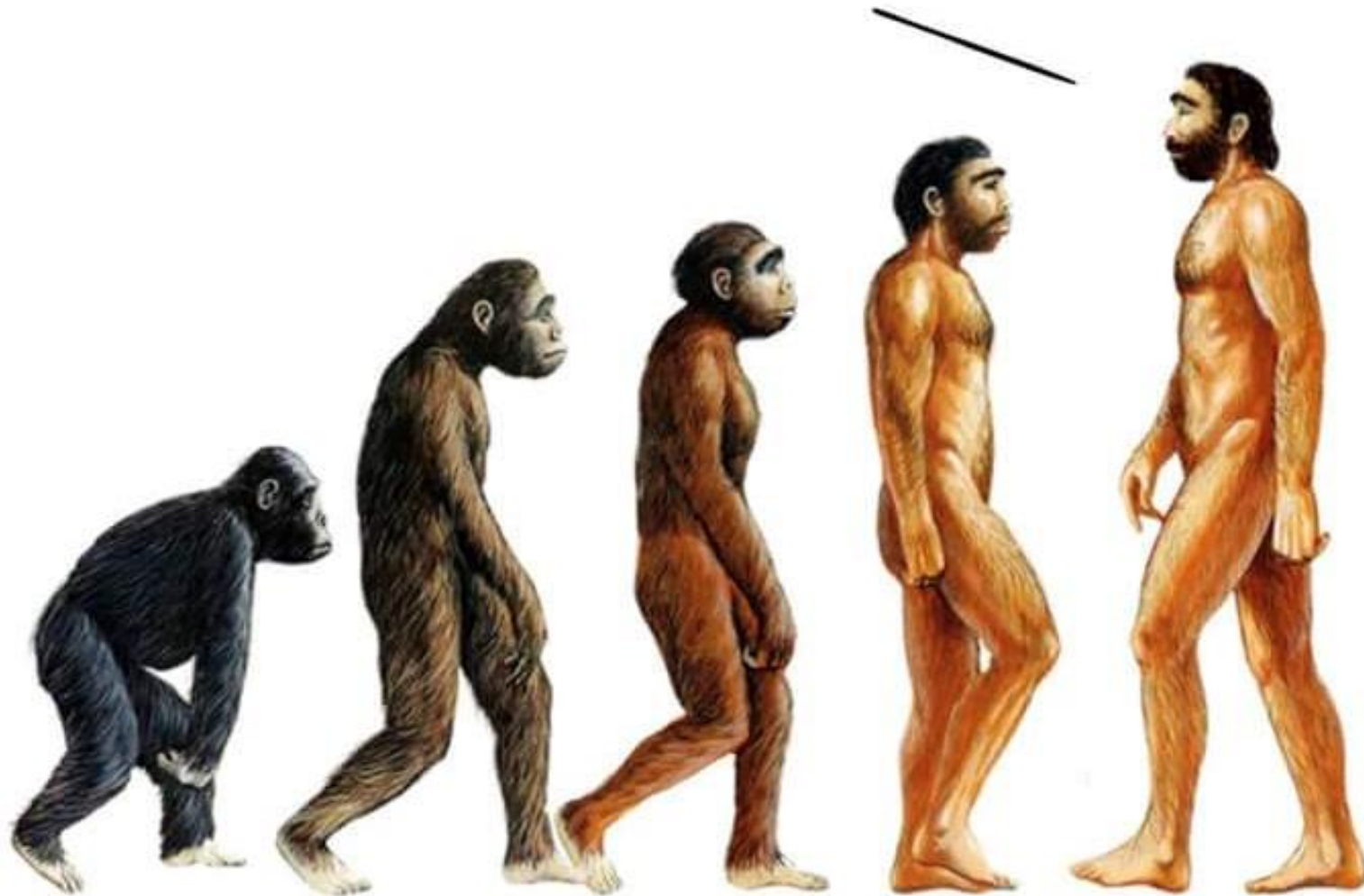


OBJECT ORIENTED

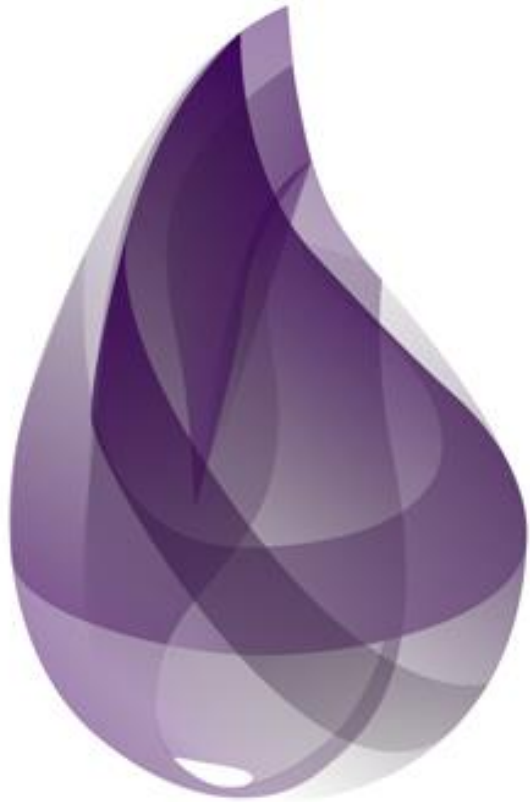


FUNCTIONAL

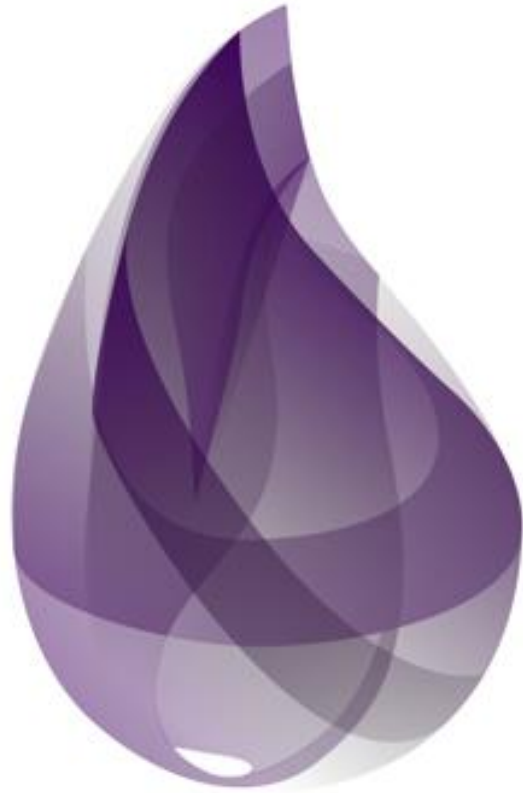
Go back. We f*cked up.



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

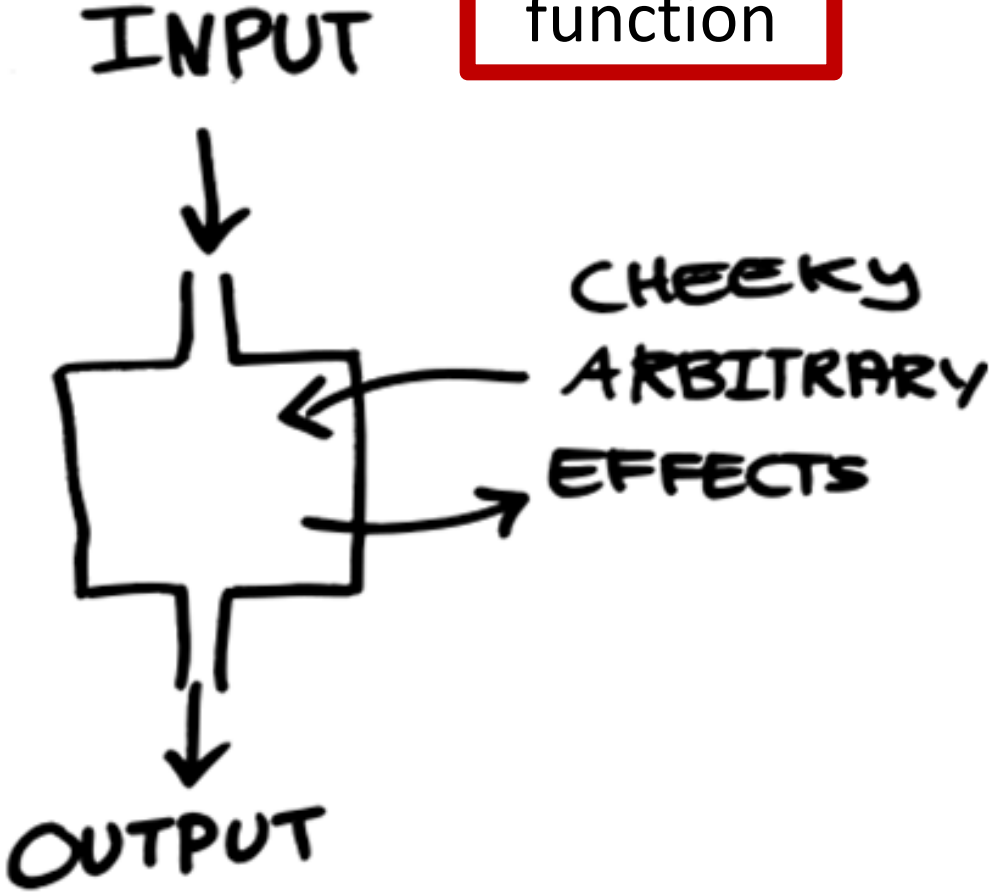
Pure Functions:

- 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



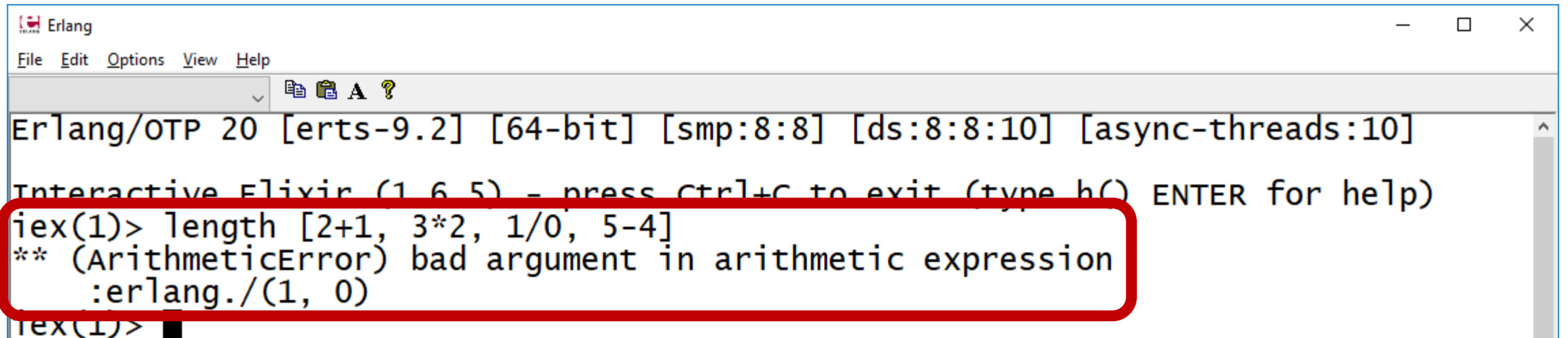
Impure
function



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.

A screenshot of an Erlang shell window. The window title is "Erlang". The menu bar includes "File", "Edit", "Options", "View", and "Help". The main text area shows the Erlang/OTP version (20) and various system parameters. Below that, it says "Interactive Elixir (1.6.5) - press Ctrl+C to exit (type h() ENTER for help)". The user has entered the command `length [2+1, 3*2, 1/0, 5-4]`. The shell has responded with an error: `** (ArithmeticError) bad argument in arithmetic expression` and `:erlang./(1, 0)`. The error message and the command line are highlighted with a red rounded rectangle. The prompt `iex(1)>` is visible at the beginning and end of the input line.

```
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

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 █
```

<https://www.haskell.org/>

A great intro to Haskell syntax

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.

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 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. DSEs allowed the AI to be specified very declaratively.

[Aetion's CUFP talk](#).

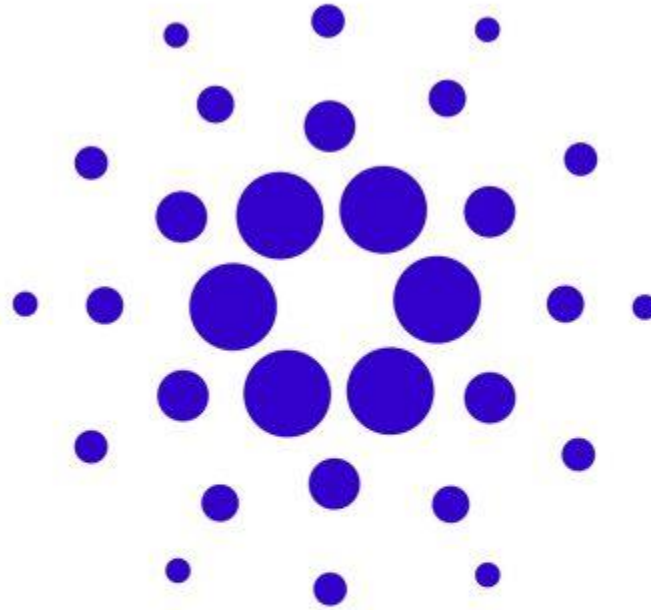
- [Alcatel-Lucent](#)

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]
```

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.



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]
```

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

Files

- main.hs

```
main.hs
1  main = putStrLn "Hello world!"
2
3
4
5 |
6
7
```

Console Shell

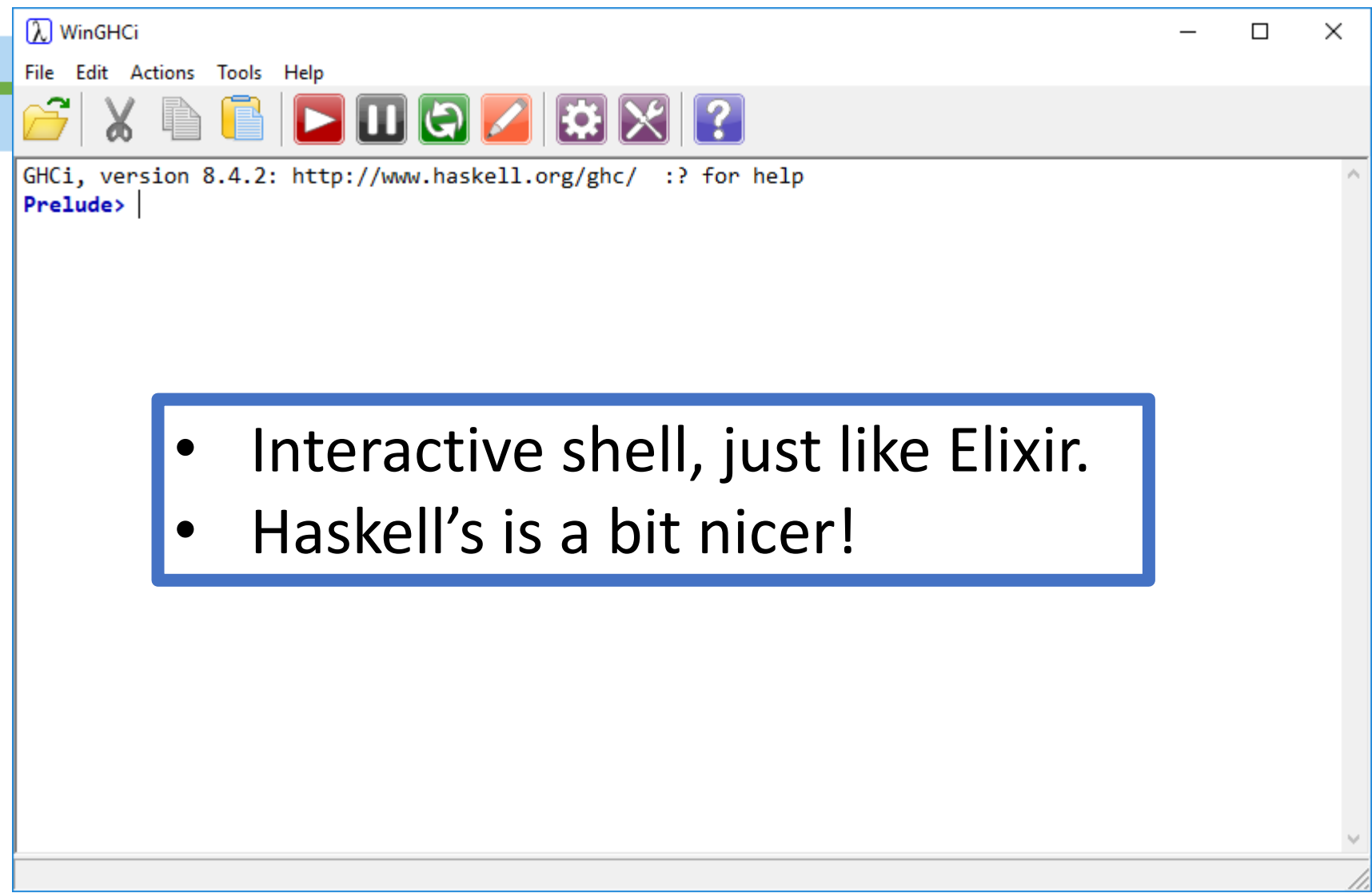
```
GHCi, version 8.6.5
>
Hello world!
>
```

Best match

 **WinGHCi**
Desktop app

Search suggestions

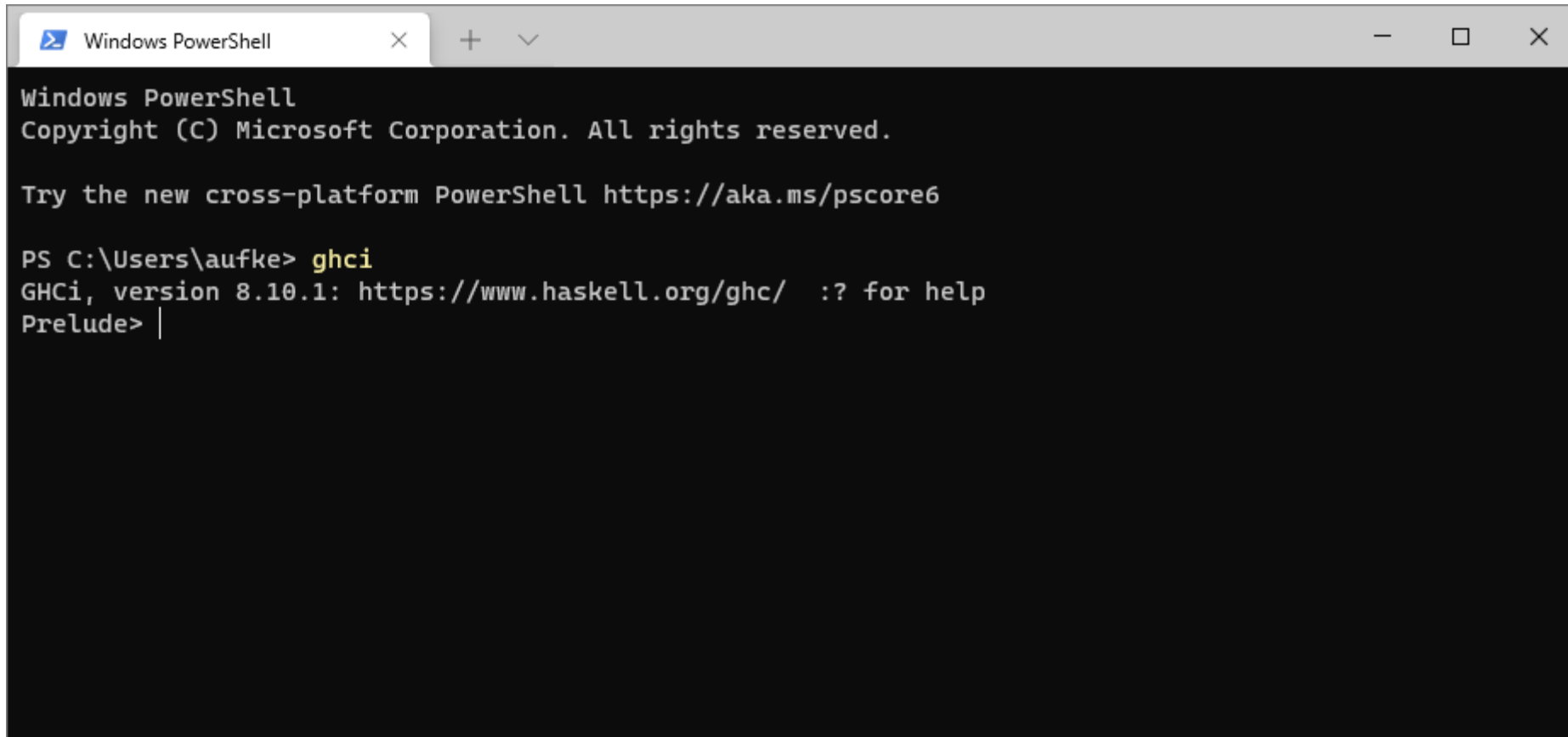
- 🔍 wingh - See web results
- 🔍 wingham
- 🔍 winghouse
- 🔍 winghart's
- 🔍 wingham ontario
- 🔍 winghaven mo
- 🔍 wingham free press
- 🔍 winghaven
- 🔍 winghouse tampa



The screenshot shows the WinGHCi application window. The title bar reads "WinGHCi". The menu bar includes "File", "Edit", "Actions", "Tools", and "Help". The toolbar contains icons for file operations (copy, paste, save, delete), playback (play, pause, refresh), and settings (gear, wrench, question mark). The main content area displays the text: "GHCi, version 8.4.2: <http://www.haskell.org/ghc/> :? for help" followed by a prompt "Prelude> |". A blue-bordered box is overlaid on the text, containing a bulleted list.

- 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:

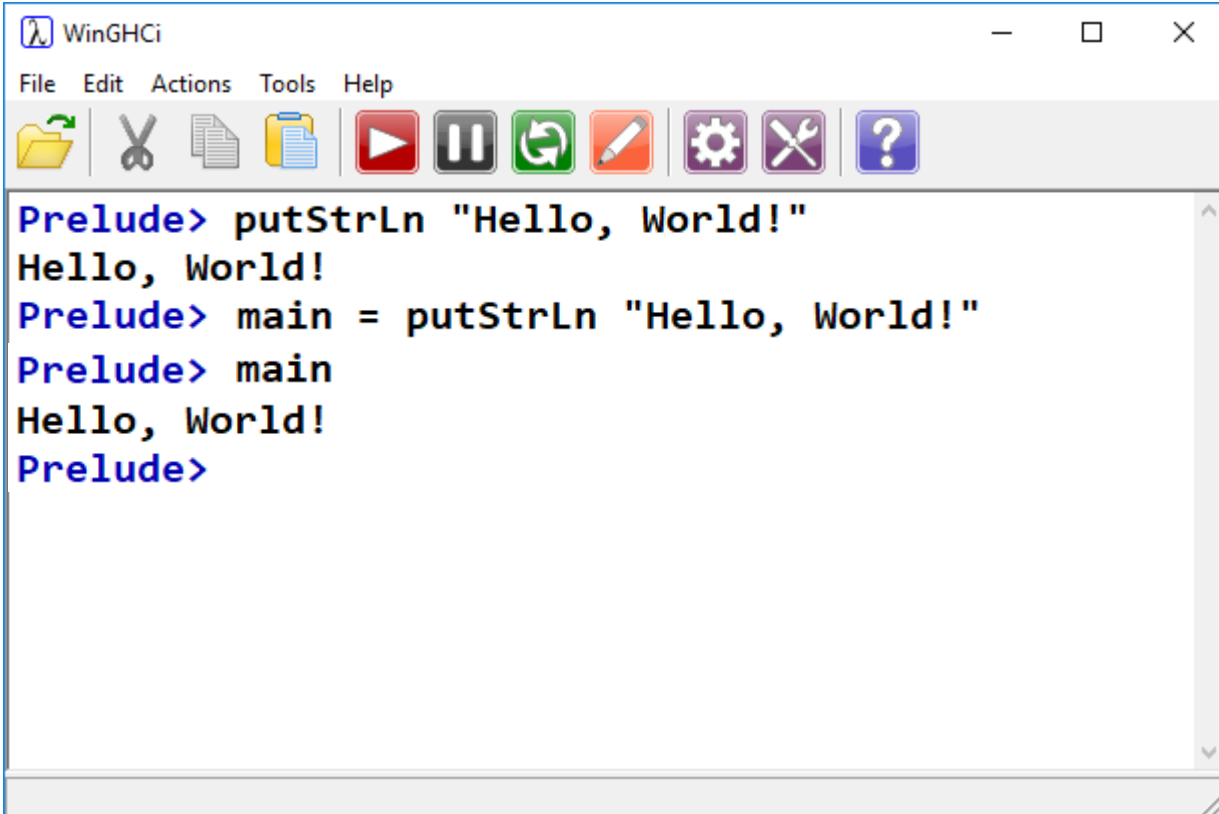


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\aufke> ghci
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
Prelude> |
```

Hello, World!

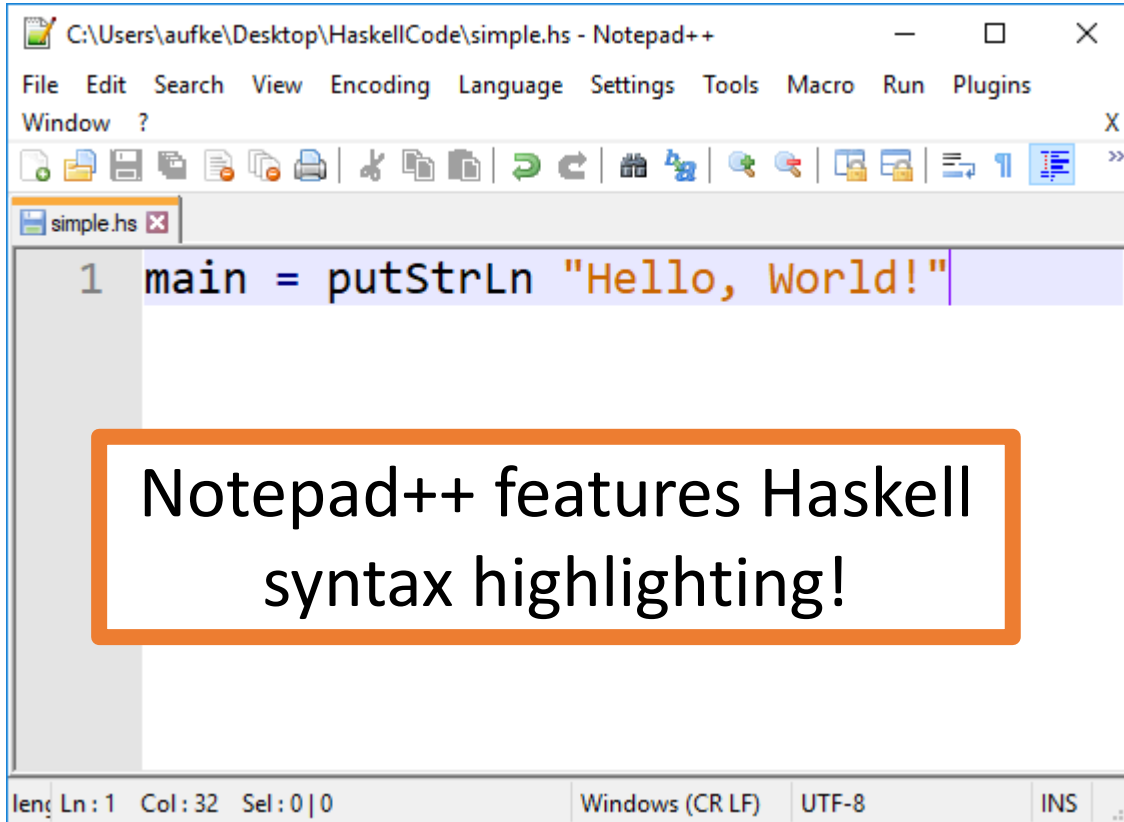


The screenshot shows a window titled "WinGHCi" with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar with icons for file operations and execution. The main text area contains the following Haskell code and its output:

```
Prelude> putStrLn "Hello, World!"
Hello, World!
Prelude> main = putStrLn "Hello, World!"
Prelude> main
Hello, World!
Prelude>
```

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

Compiling Haskell

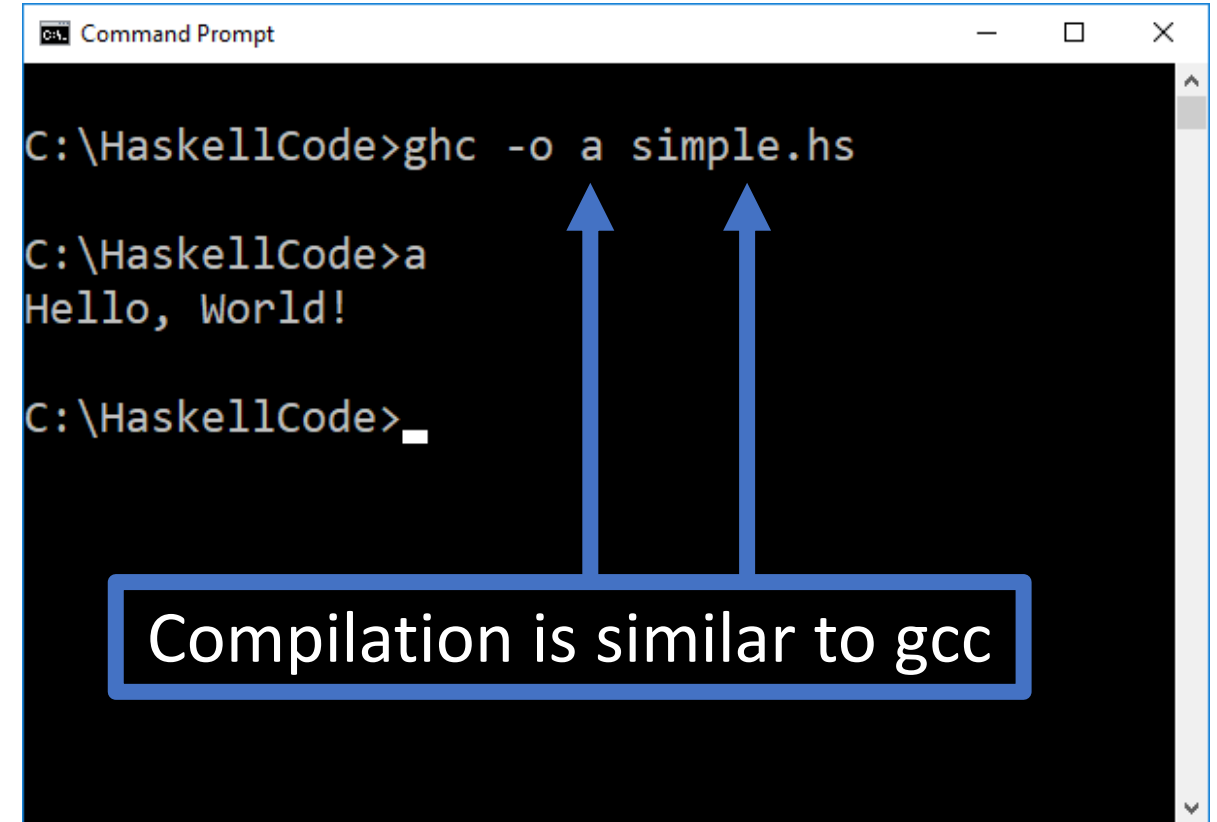


C:\Users\aufke\Desktop\HaskellCode\simple.hs - Notepad++

```
File Edit Search View Encoding Language Settings Tools Macro Run Plugins
Window ?
simple.hs
1 main = putStrLn "Hello, World!"
```

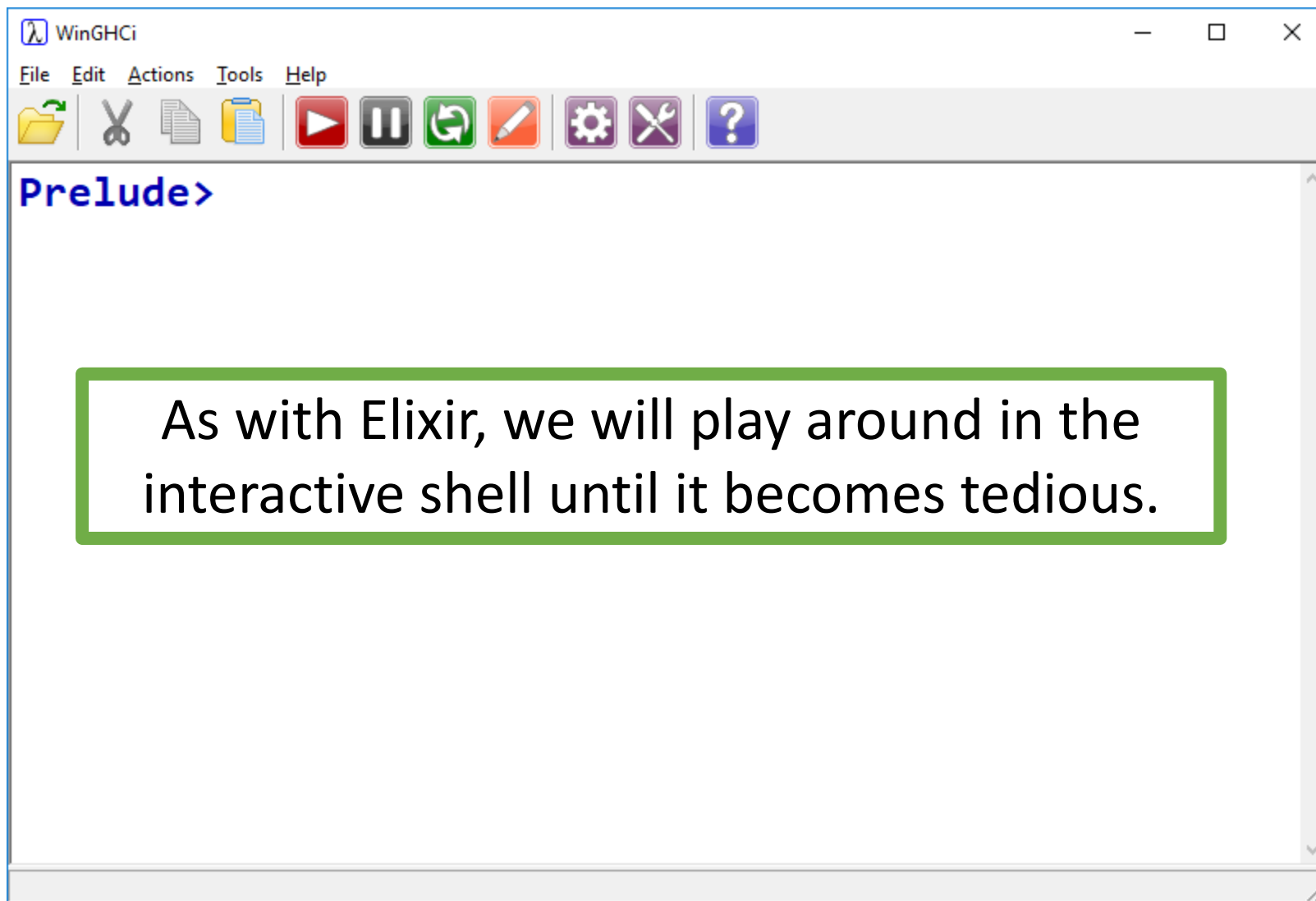
Notepad++ features Haskell syntax highlighting!

len: Ln: 1 Col: 32 Sel: 0 | 0 Windows (CR LF) UTF-8 INS

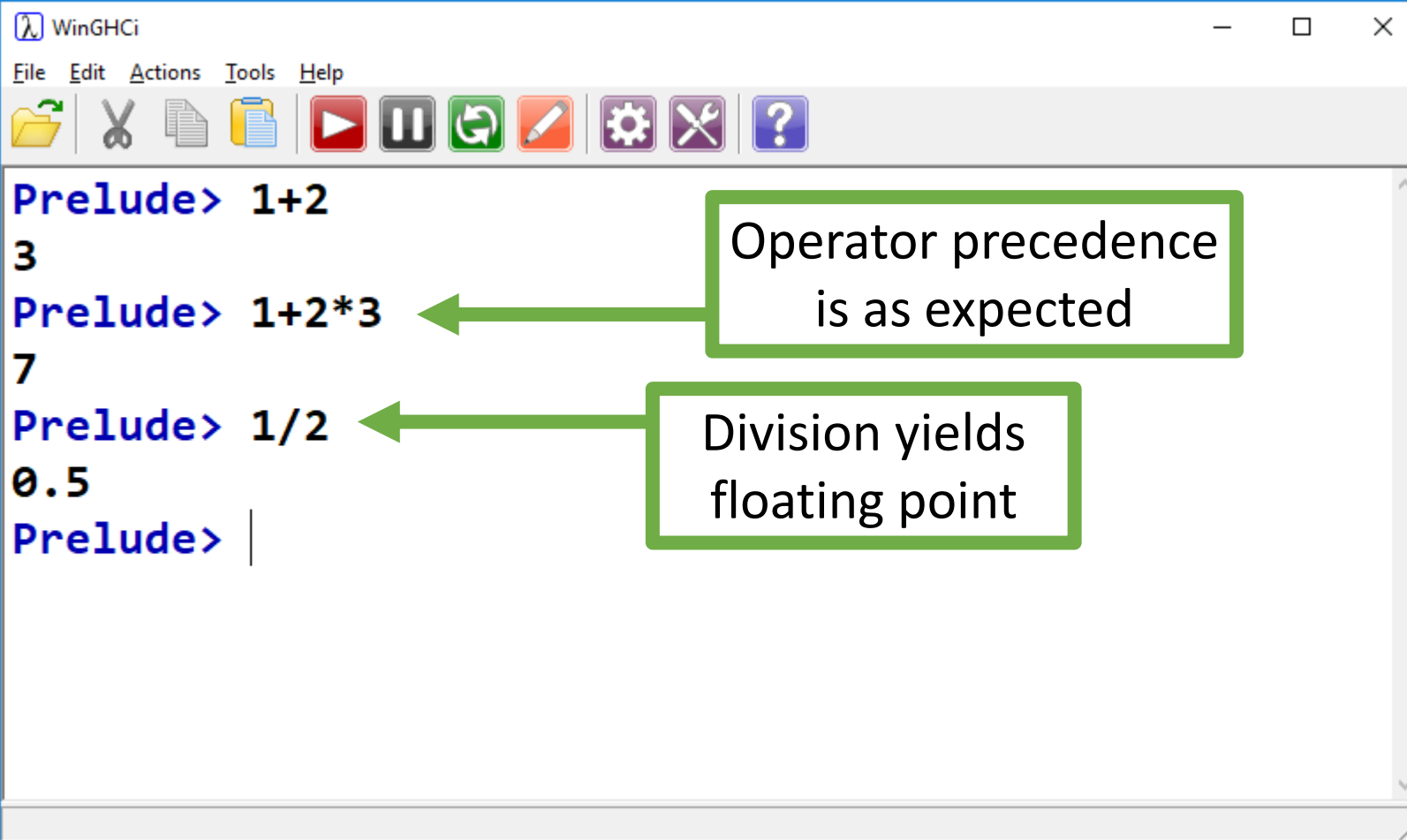


```
Command Prompt
C:\HaskellCode>ghc -o a simple.hs
C:\HaskellCode>a
Hello, World!
C:\HaskellCode>_
```

Compilation is similar to gcc



Literals & Arithmetic



The screenshot shows the WinGHCi terminal window with the following content:

```
WinGHCi
File Edit Actions Tools Help
[Icons: Folder, Scissors, Document, Clipboard, Play, Pause, Refresh, Eraser, Gear, Wrench, Question Mark]

Prelude> 1+2
3
Prelude> 1+2*3
7
Prelude> 1/2
0.5
Prelude> |
```

Annotations:

- A green box containing the text "Operator precedence is as expected" has a green arrow pointing to the result of the expression `1+2*3`.
- A green box containing the text "Division yields floating point" has a green arrow pointing to the result of the expression `1/2`.

Literals & Arithmetic

The screenshot shows the WinGHCi terminal window with the following content:

```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> sqrt 2
1.4142135623730951
Prelude> sqrt (2)
1.4142135623730951
Prelude> 2^2
4
Prelude> sqrt(2)^2
2.0000000000000004
Prelude>
```

Annotations:

- A blue box with a bracket on the right side of the first two lines: "Like Elixir, can omit brackets on function calls".
- A green box with an arrow pointing to the `2^2` line: "`^` can be used for exponentiation".
- An orange box with a bracket on the right side of the last line: "Representation error, no escaping it."

Tuples

```
WinGHCi
File Edit Actions Tools Help
[Icons]
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.

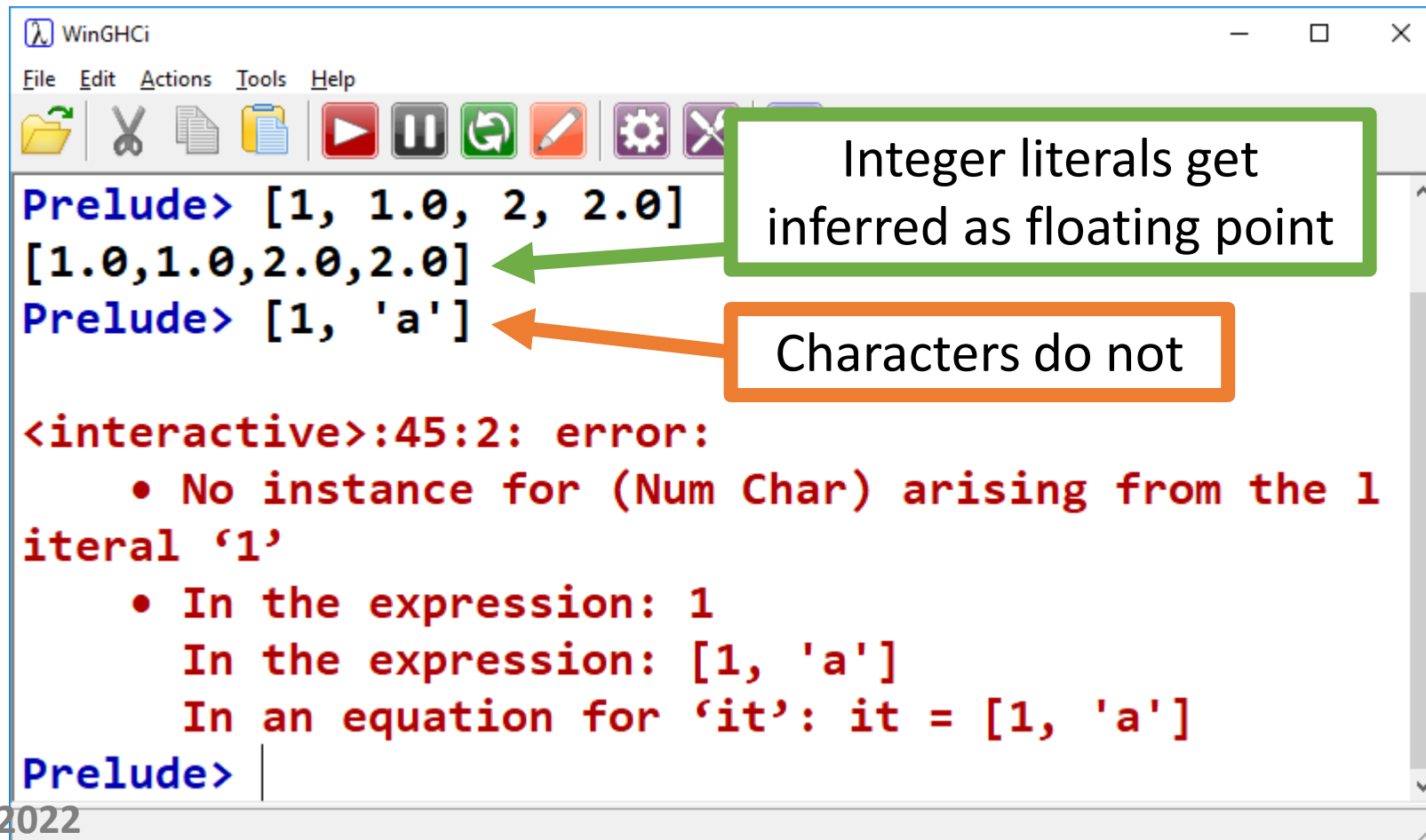
```
WinGHCi
File Edit Actions Tools Help
[Icons: Folder, Scissors, Document, Print, Play, Pause, Refresh, Eraser, Gear, Wrench, Question Mark]
Prelude> fst (1, 2, 3)

<interactive>:35:5: error:
  • Couldn't match expected type '(a, b0)'
    with actual type '(Integer, Integer, 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*:



The screenshot shows a WinGHCi terminal window with the following content:

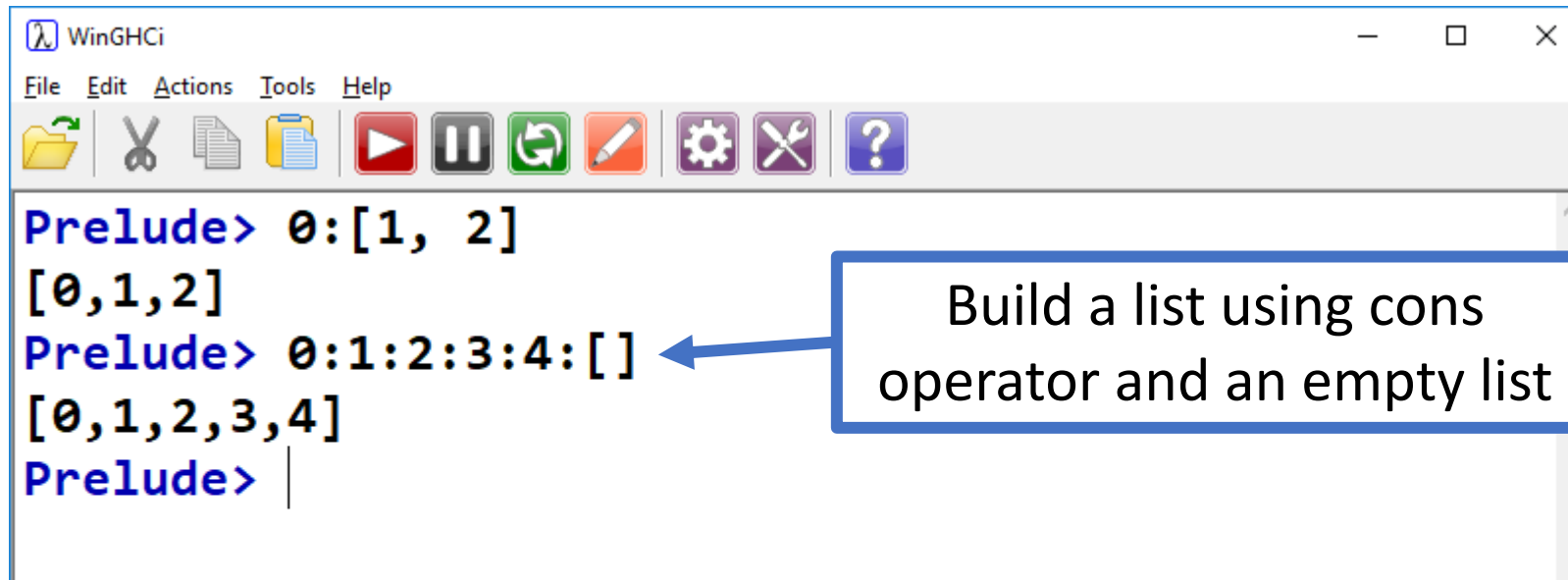
```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> [1, 1.0, 2, 2.0]
[1.0,1.0,2.0,2.0]
Prelude> [1, 'a']
<interactive>:45:2: error:
  • No instance for (Num Char) arising from the 1
  literal '1'
  • In the expression: 1
    In the expression: [1, 'a']
    In an equation for 'it': it = [1, 'a']
Prelude>
```

Annotations in the image:

- A green box highlights the output `[1.0,1.0,2.0,2.0]` with the text "Integer literals get inferred as floating point". A green arrow points from this box to the output.
- An orange box highlights the input `[1, 'a']` with the text "Characters do not". An orange arrow points from this box to the input.

Lists

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



```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> 0:[1, 2]
[0,1,2]
Prelude> 0:1:2:3:4:[]
[0,1,2,3,4]
Prelude> |
```

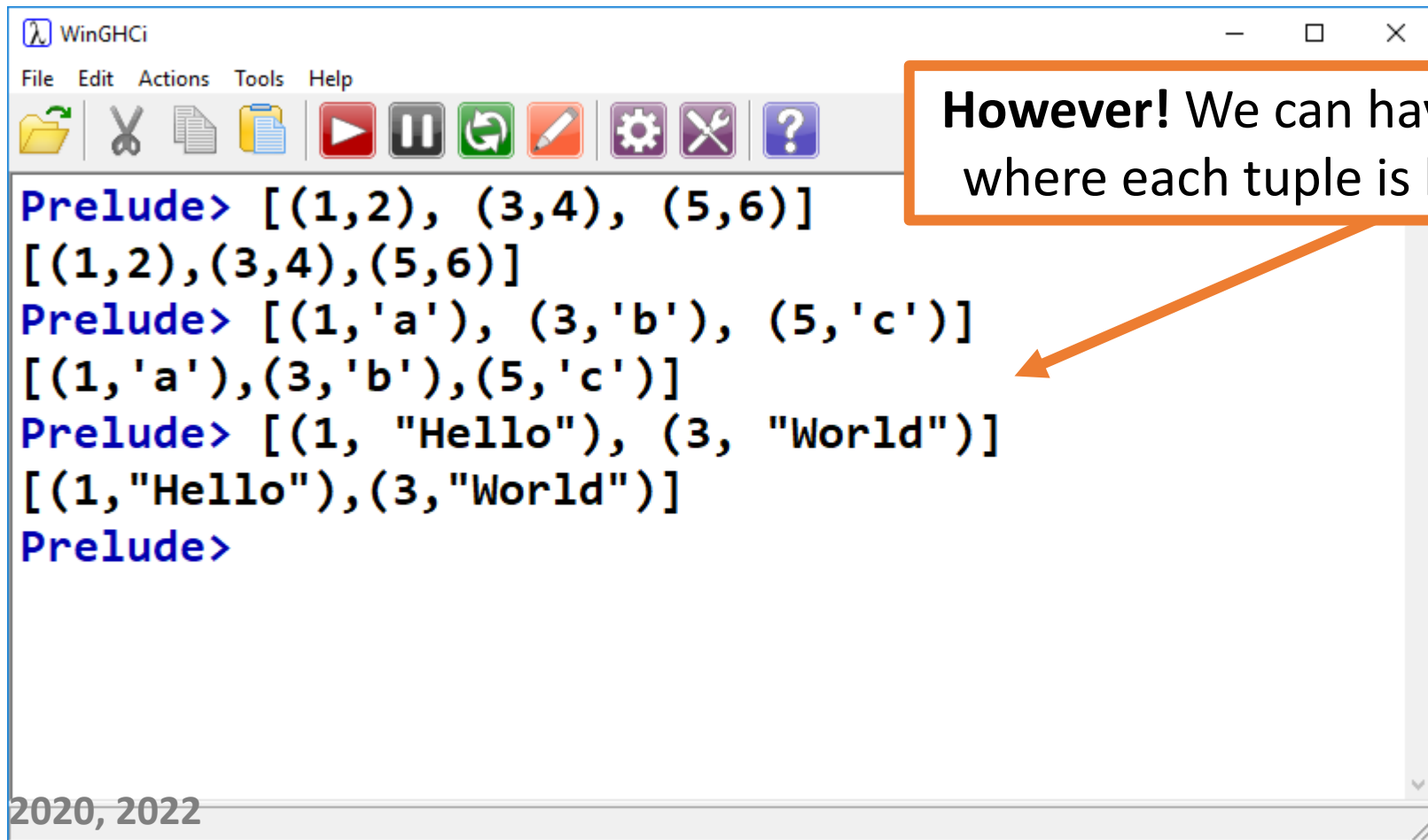
A blue arrow points from a text box to the `0:1:2:3:4:[]` command in the terminal.

Build a list using cons operator and an empty list

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.



```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> [(1,2), (3,4), (5,6)]
[(1,2),(3,4),(5,6)]
Prelude> [(1,'a'), (3,'b'), (5,'c')]
[(1,'a'),(3,'b'),(5,'c')]
Prelude> [(1, "Hello"), (3, "World")]
[(1,"Hello"),(3,"World")]
Prelude>
```

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



Lists & Tuples

Tuples can be heterogeneous, lists must be homogeneous.

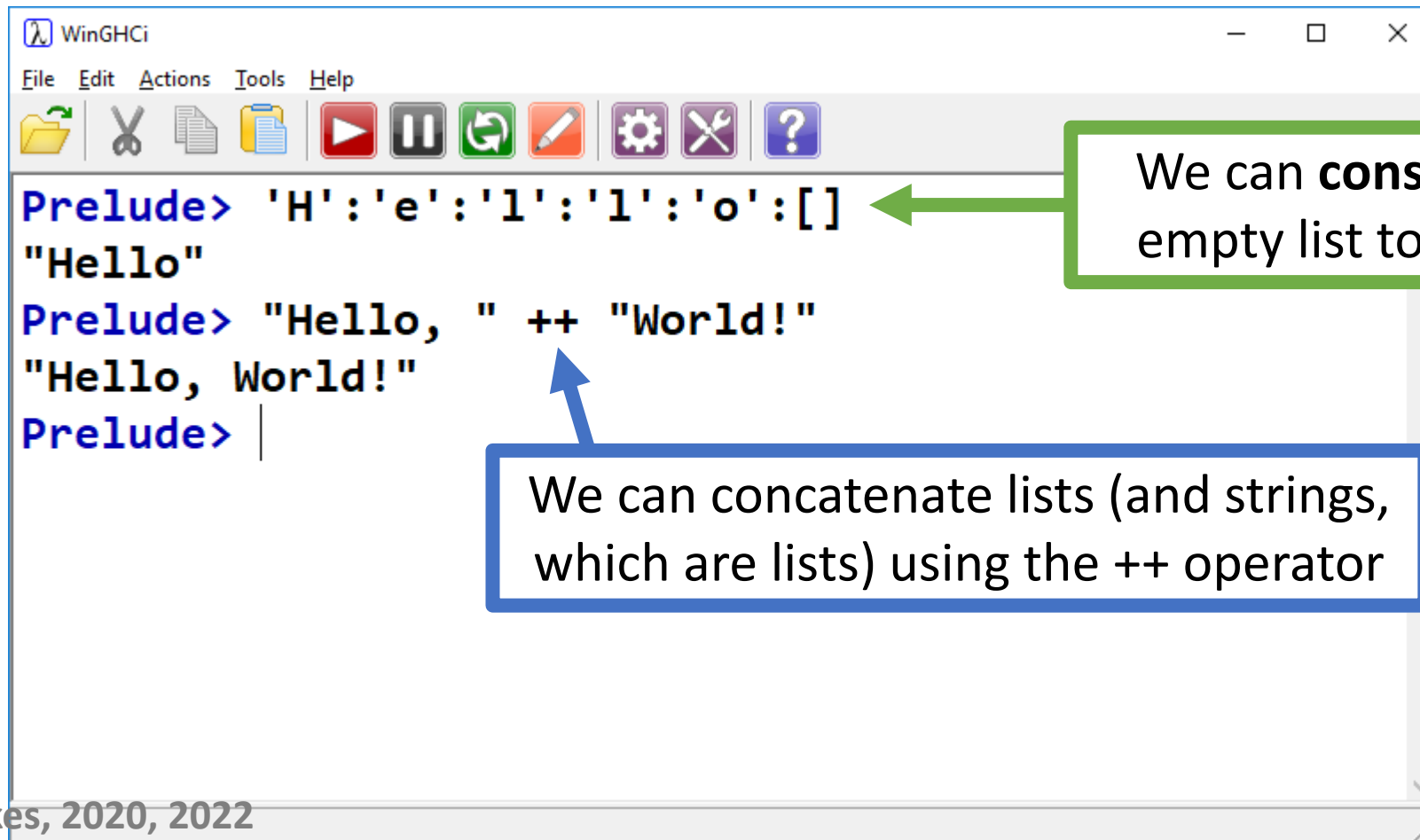
```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> [(1, "Hello"), (2, "World")]
[(1, "Hello"), (2, "World")]
Prelude> [(1, "Hello"), (2, 3.4)]
<interactive>:67:20: error:
• Could not deduce (Fractional [Char])
  arising from the literal '3.4'
  from the context: Num a
                    bound 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:



```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> 'H': 'e': 'l': 'l': 'o': []
"Hello"
Prelude> "Hello, " ++ "World!"
"Hello, World!"
Prelude> |
```

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...

The screenshot shows a Haskell IDE window titled "WinGHCi" with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar. The main editor area displays the following code and error message:

```
Prelude> "Hello, " ++ 5.0
```

<interactive>:61:14: error:

- No instance for (Fractional [Char]) arising from the literal '5.0'
- In the second argument of '(++)', namely '5.0'

In the expression: "Hello, " ++ 5.0
In an equation for 'it': it = "Hello, " ++ 5.0

```
0  
Prelude>
```

On the left, a code editor window titled "StringTester - HelloWorld" shows a snippet of Haskell code:

```
public class  
{  
    public s  
    {  
        Stri  
        Syst  
    }  
}
```

At the bottom of the IDE, a status bar indicates "Class compiled, no syntax errors" and "saved".

Strings

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

```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> "This course is " ++ "CCPS" ++ show(506)
"This course is CCPS506"
Prelude> read "506" - 6
500
Prelude> read "Hello" + 1
*** Exception: Prelude.read: no parse
Prelude>
```

`show(506)`

Convert non-string argument to string

`read "506" - 6`
`500`

Read numeric value from a string (like *scanf* 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

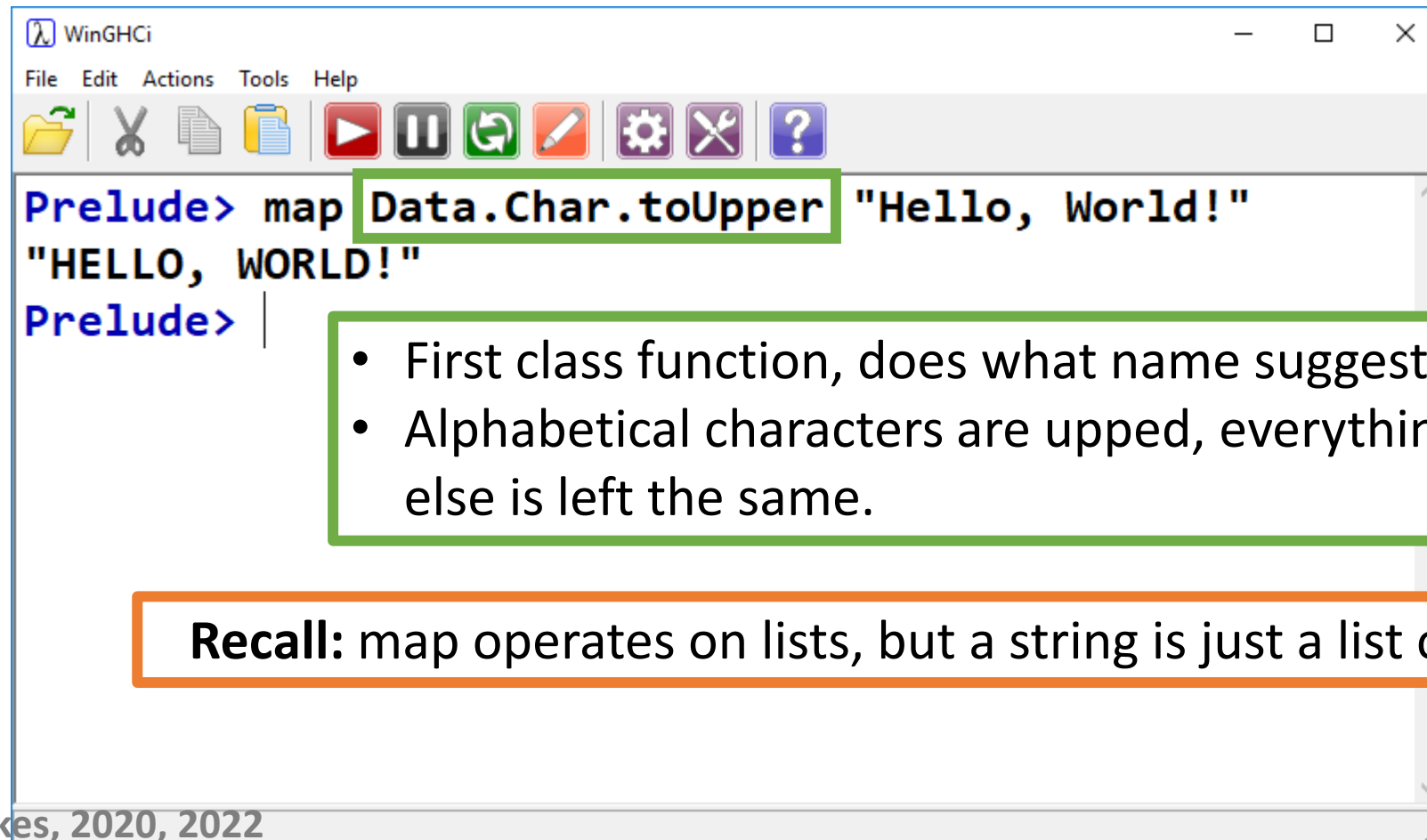
```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> tail [1, 2, 3]
[2,3]
Prelude> head [1, 2, 3]
1
Prelude> tail [1]
[]
Prelude> head [1]
1
Prelude> tail []
*** Exception: Prelude.tail: empty list
Prelude>
```

Same as Elixir:

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

map

Similar to Elixir's `Enum.map`



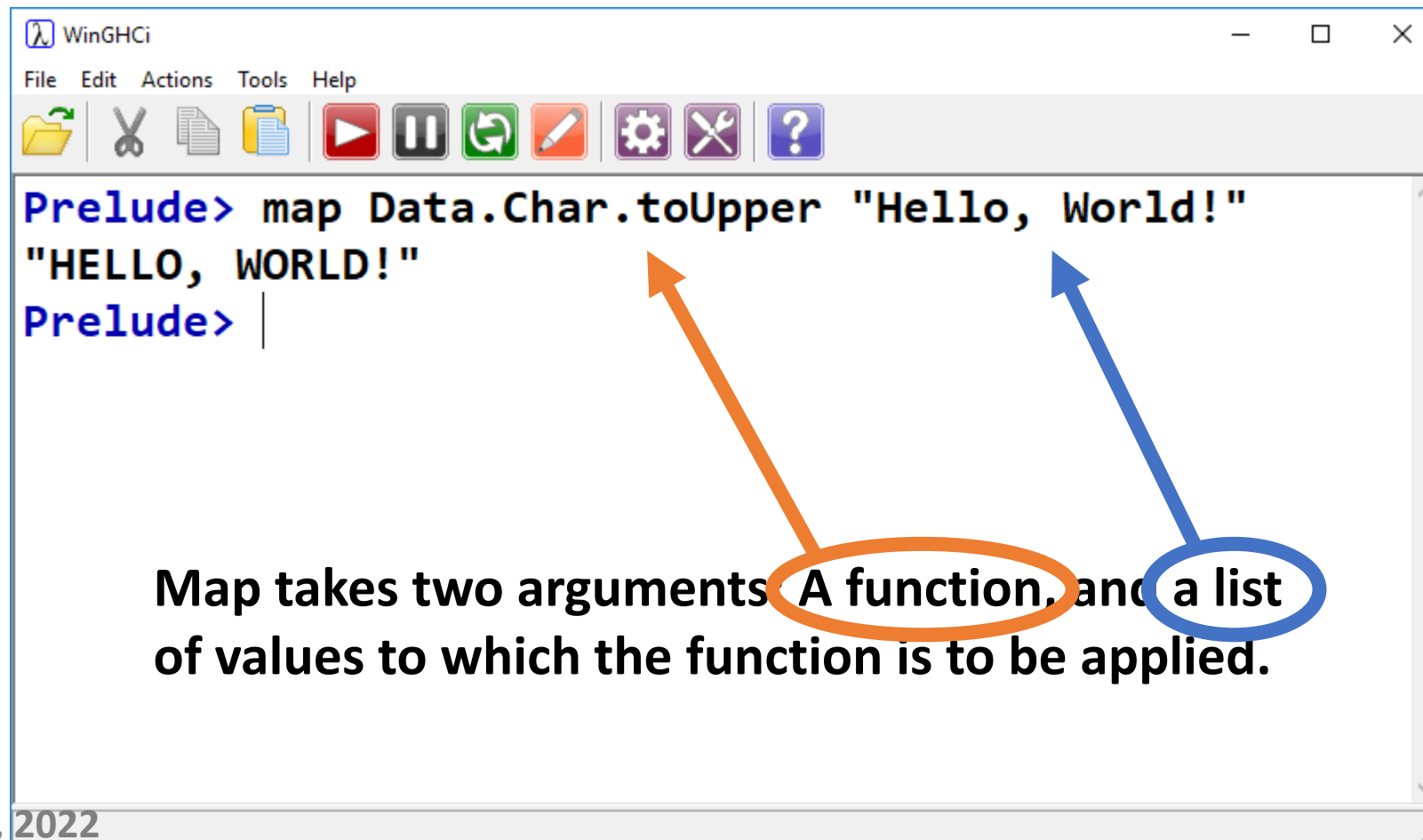
```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> map Data.Char.toUpper "Hello, World!"
"HELLO, WORLD!"
Prelude> |
```

- 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

Similar to Elixir's `Enum.map`

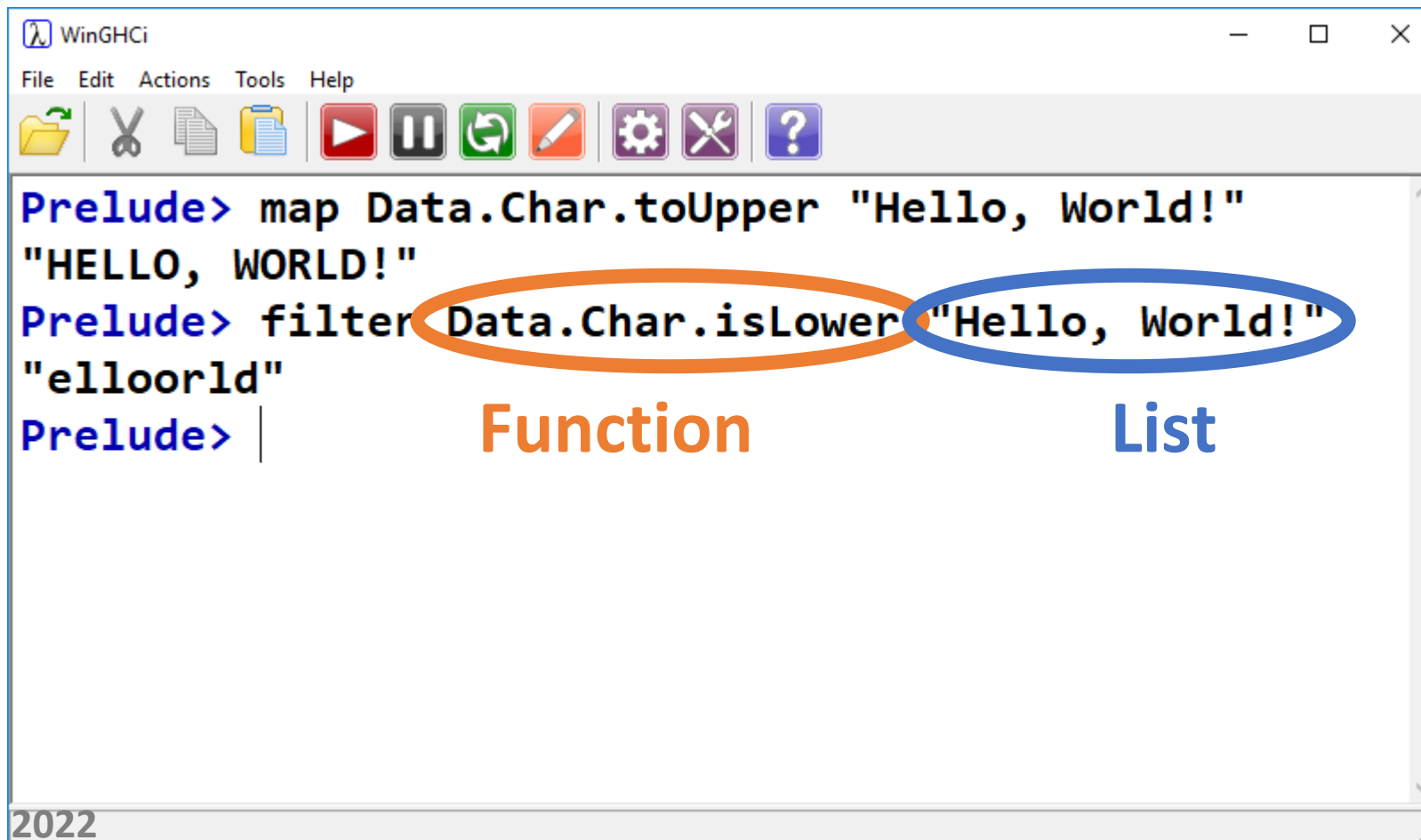


```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> map Data.Char.toUpper "Hello, World!"
"HELLO, WORLD!"
Prelude> |
```

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:



The screenshot shows a WinGHCi terminal window with the following content:

```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> map Data.Char.toUpper "Hello, World!"
"HELLO, WORLD!"
Prelude> filter Data.Char.isLower "Hello, World!"
"elloorld"
Prelude> |
```

Annotations in the image:

- An orange oval highlights the expression `Data.Char.isLower`, with the word **Function** written below it in orange.
- A blue oval highlights the string `"Hello, World!"`, with the word **List** written below it in blue.

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:[]`

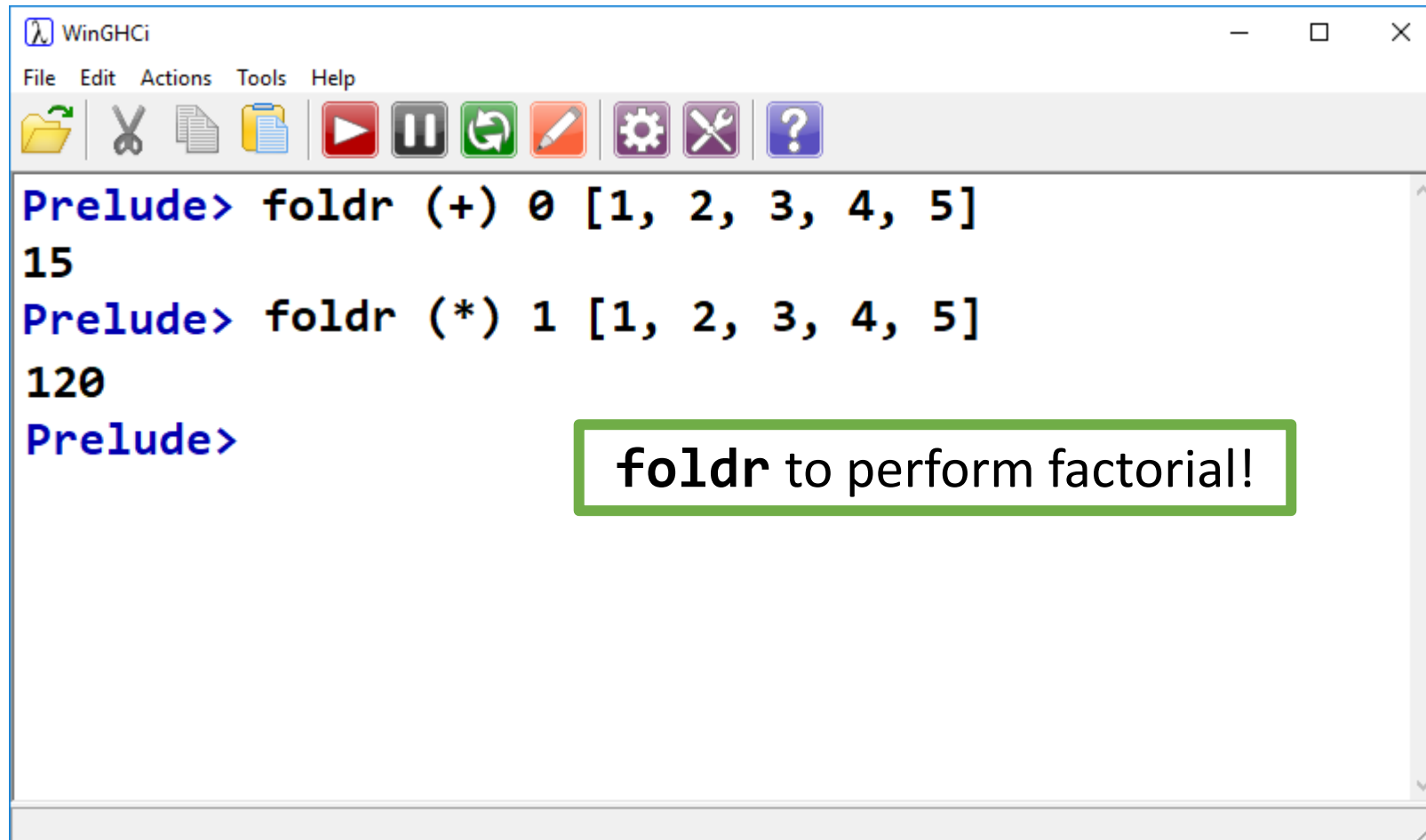


`1 + 2 + 3 + 4 + 5 + 0`



`15`

foldl, foldr



The screenshot shows a terminal window titled "WinGHCi" with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar with various icons. The terminal content is as follows:

```
Prelude> foldr (+) 0 [1, 2, 3, 4, 5]
15
Prelude> foldr (*) 1 [1, 2, 3, 4, 5]
120
Prelude>
```

A green-bordered box highlights the text "foldr to perform factorial!" in the terminal area.

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`

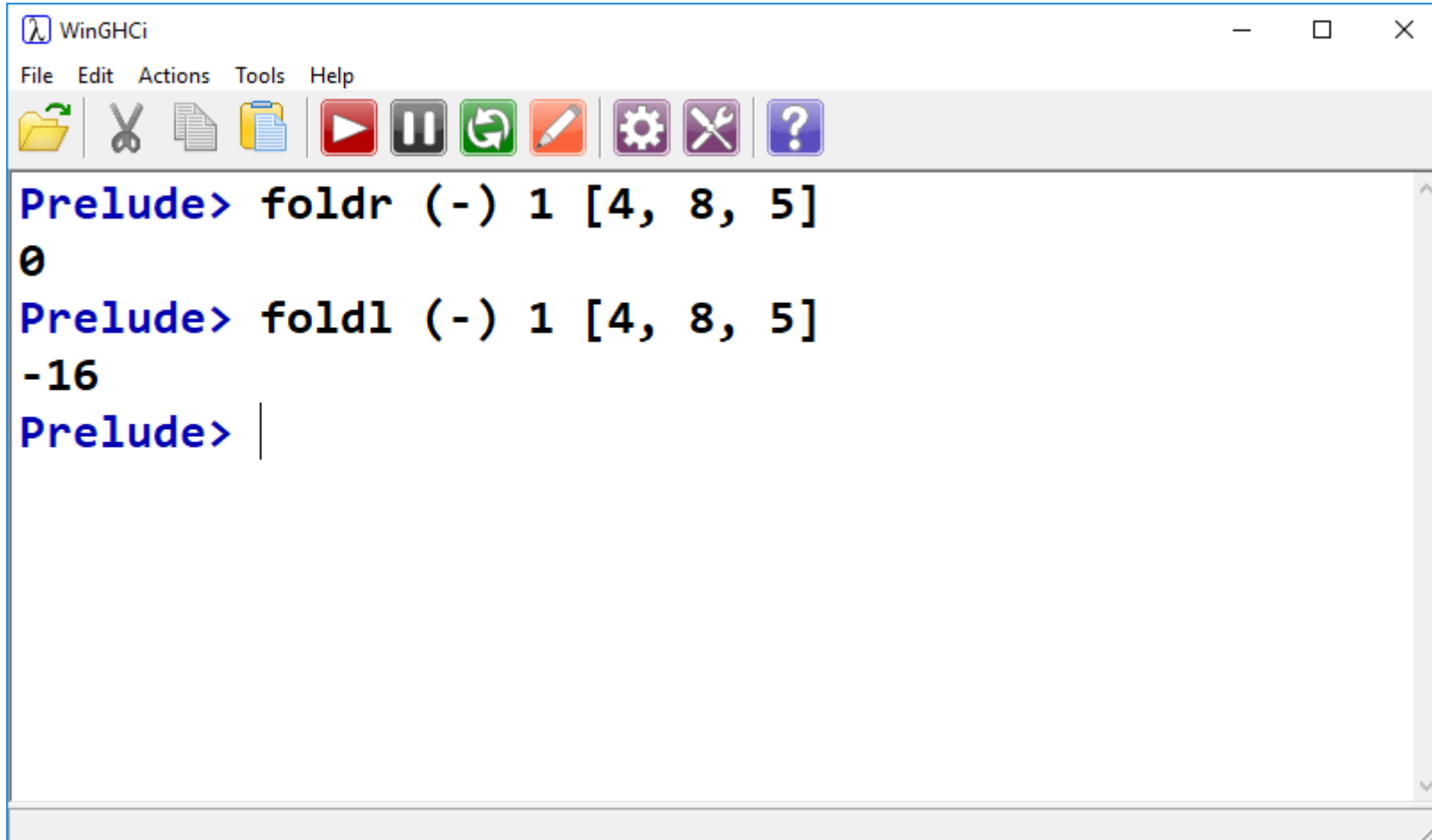
Is actually:

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



`-16`

foldl VS foldr

A screenshot of the WinGHCi Haskell interpreter window. The window title is "WinGHCi" and it has standard window controls (minimize, maximize, close). The menu bar includes "File", "Edit", "Actions", "Tools", and "Help". Below the menu bar is a toolbar with icons for file operations (folder, scissors, document), execution (play, pause, refresh), editing (pencil), settings (gear), and help (question mark). The main text area shows the following interaction:

```
Prelude> foldr (-) 1 [4, 8, 5]
0
Prelude> foldl (-) 1 [4, 8, 5]
-16
Prelude> |
```

List Generation

Syntactic sugar:

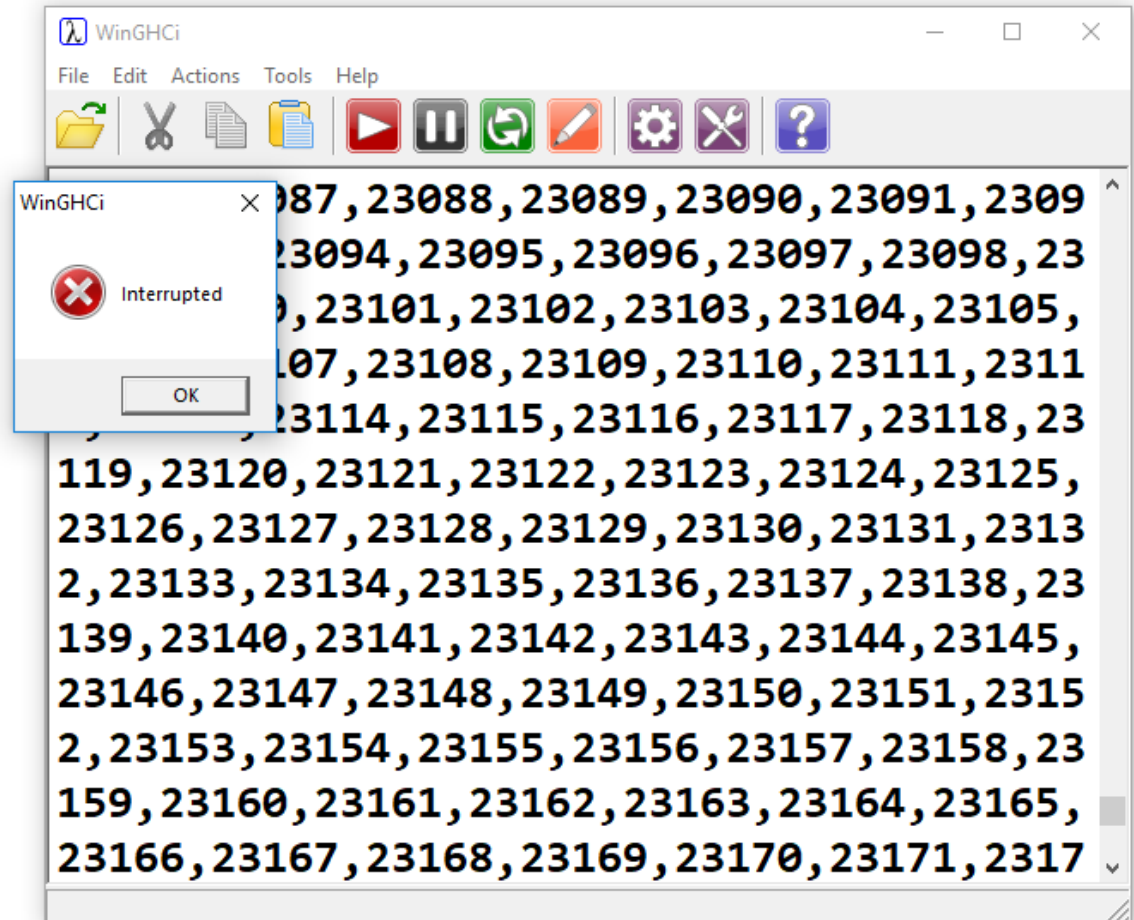
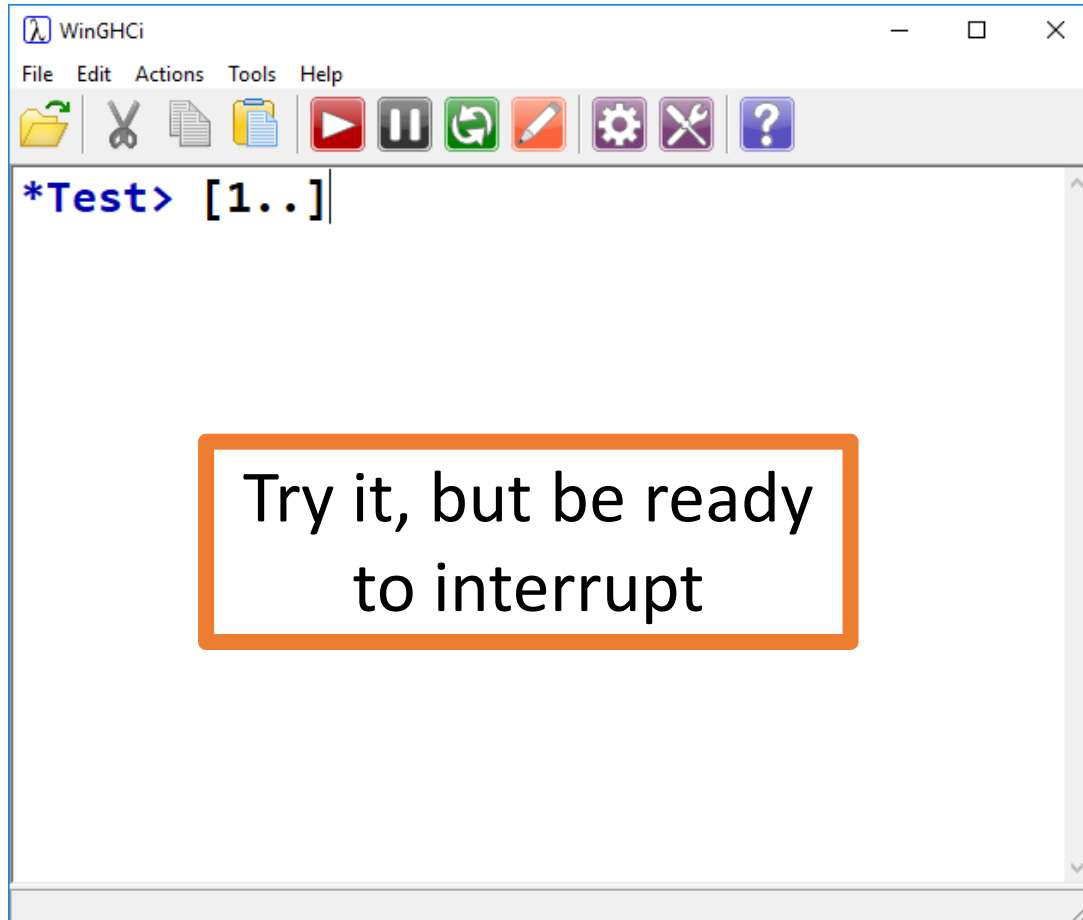
List declaration: `list = [1, 2, 3, 4, 5, 6, 7, 8, 9]`

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

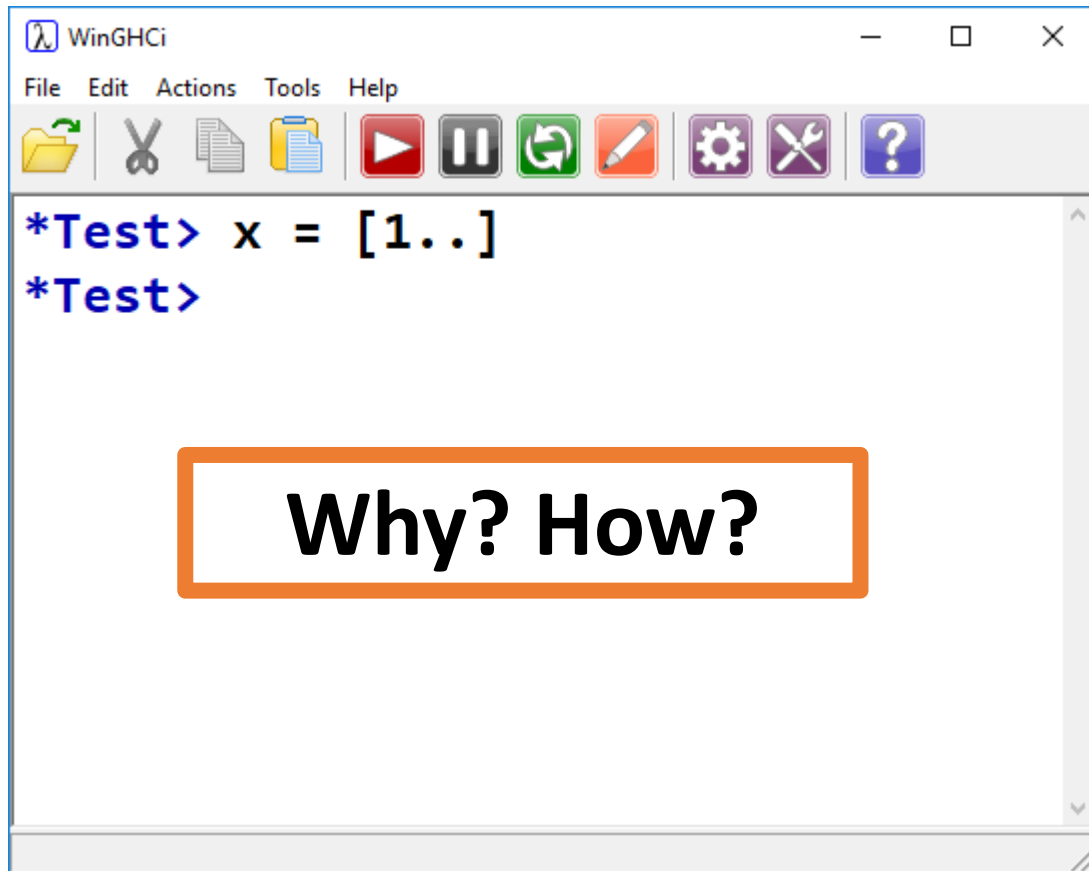
Specify interval:
`list = [1, 3..9]`
`= [1, 3, 5, 7, 9]`

Interval is discerned from difference between first two elements

Infinite Lists?



Infinite Lists?



The screenshot shows a terminal window titled "WinGHCi" with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar with icons for file operations and execution. The terminal content is as follows:

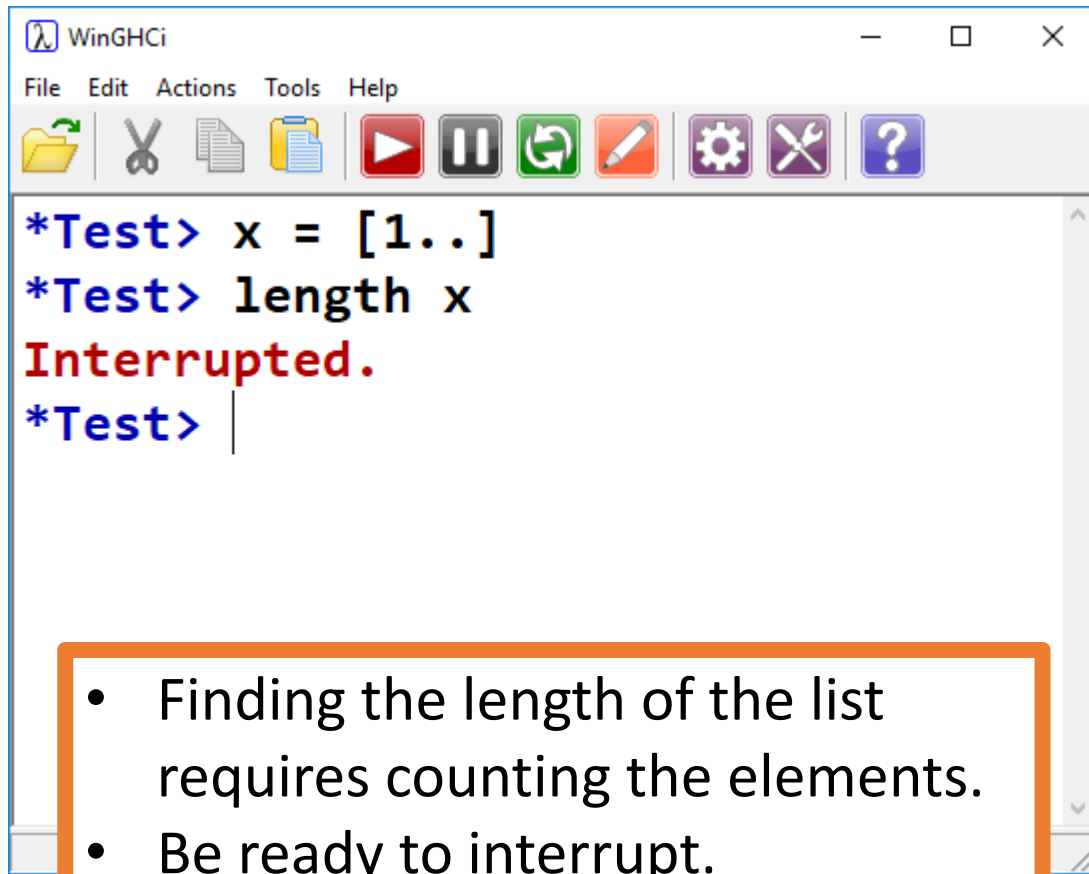
```
*Test> x = [1..]  
*Test>
```

Below the terminal content, there is a callout box with an orange border containing the text "Why? How?".

Haskell is lazy!

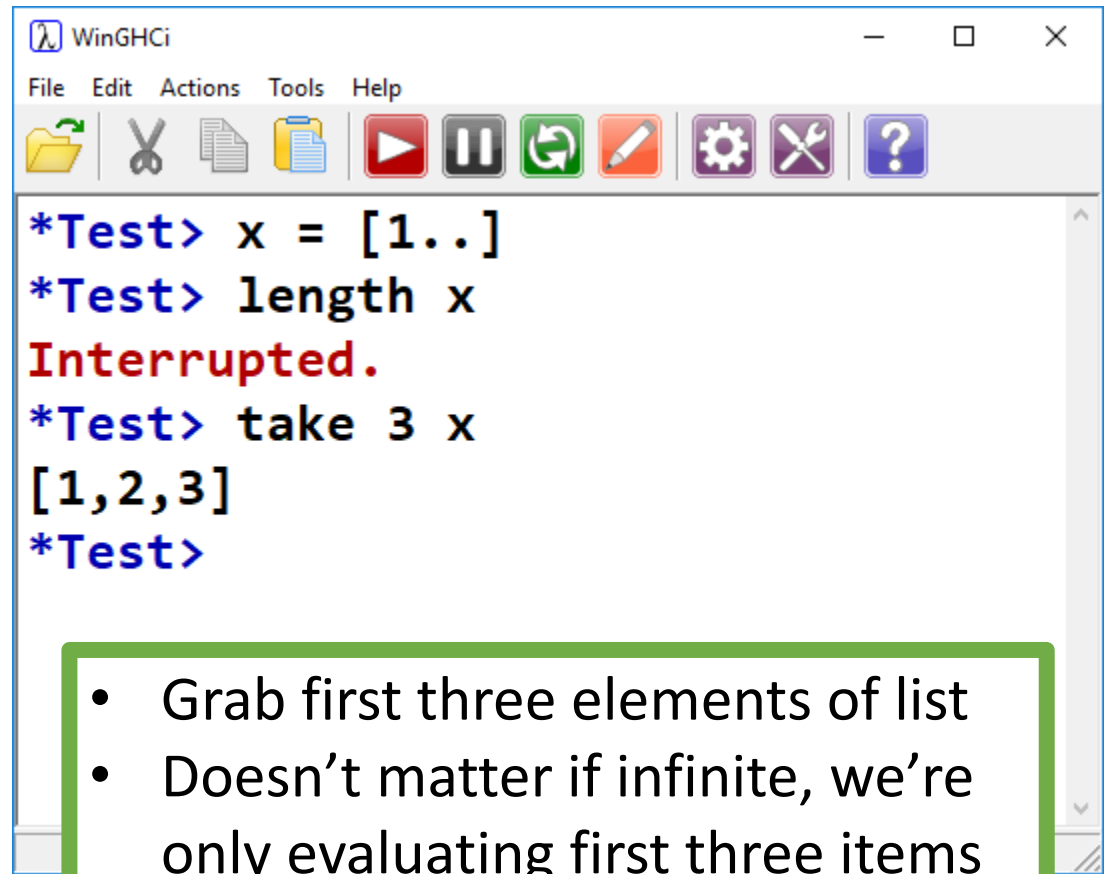
- We bind x to the expression to generate an infinite list.
- We don't have to *evaluate* this list to do so!
- Displaying the list, however, requires evaluation.

Infinite Lists?



```
WinGHCi
File Edit Actions Tools Help
[Icons]
*Test> x = [1..]
*Test> length x
Interrupted.
*Test> |
```

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



```
WinGHCi
File Edit Actions Tools Help
[Icons]
*Test> x = [1..]
*Test> length x
Interrupted.
*Test> take 3 x
[1,2,3]
*Test>
```

- 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
[Icons]

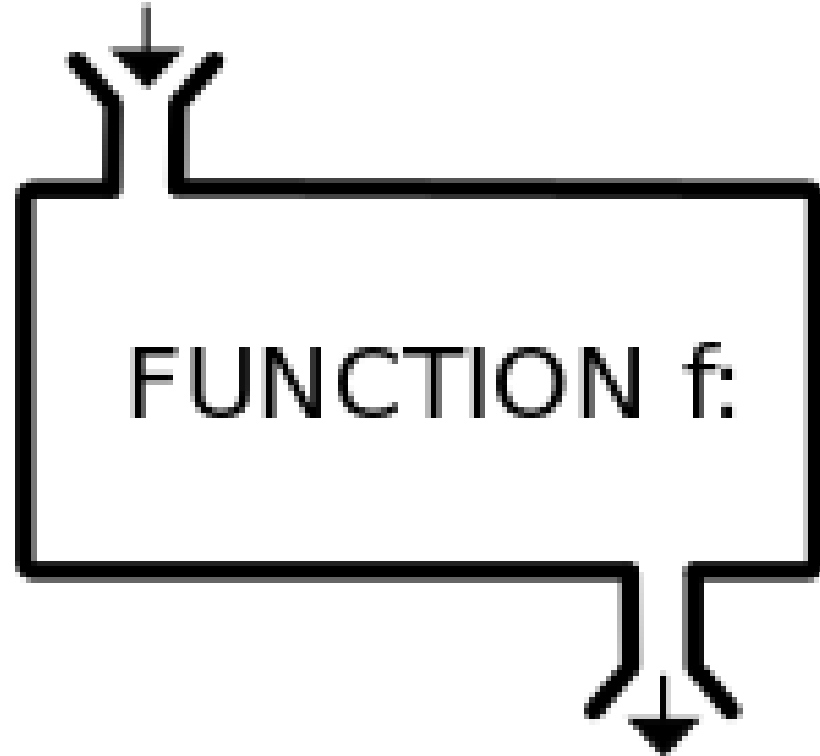
*Test> x = [1..]
*Test> length x
Interrupted.
*Test> take 3 x
[1,2,3]
*Test> take 3 (drop 5 x)
[6,7,8]
*Test>
```

```
WinGHCi
File Edit Actions Tools Help
[Icons]

*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.

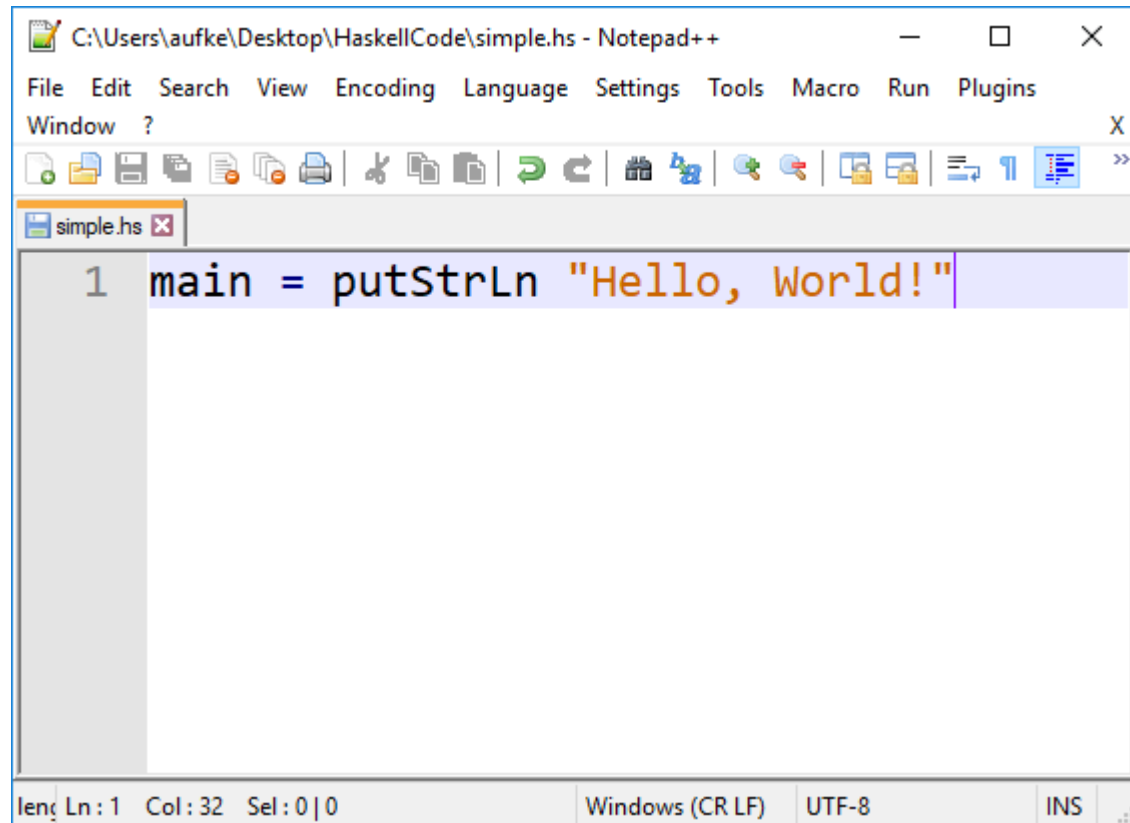
INPUT x



OUTPUT $f(x)$

Functions in Haskell

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

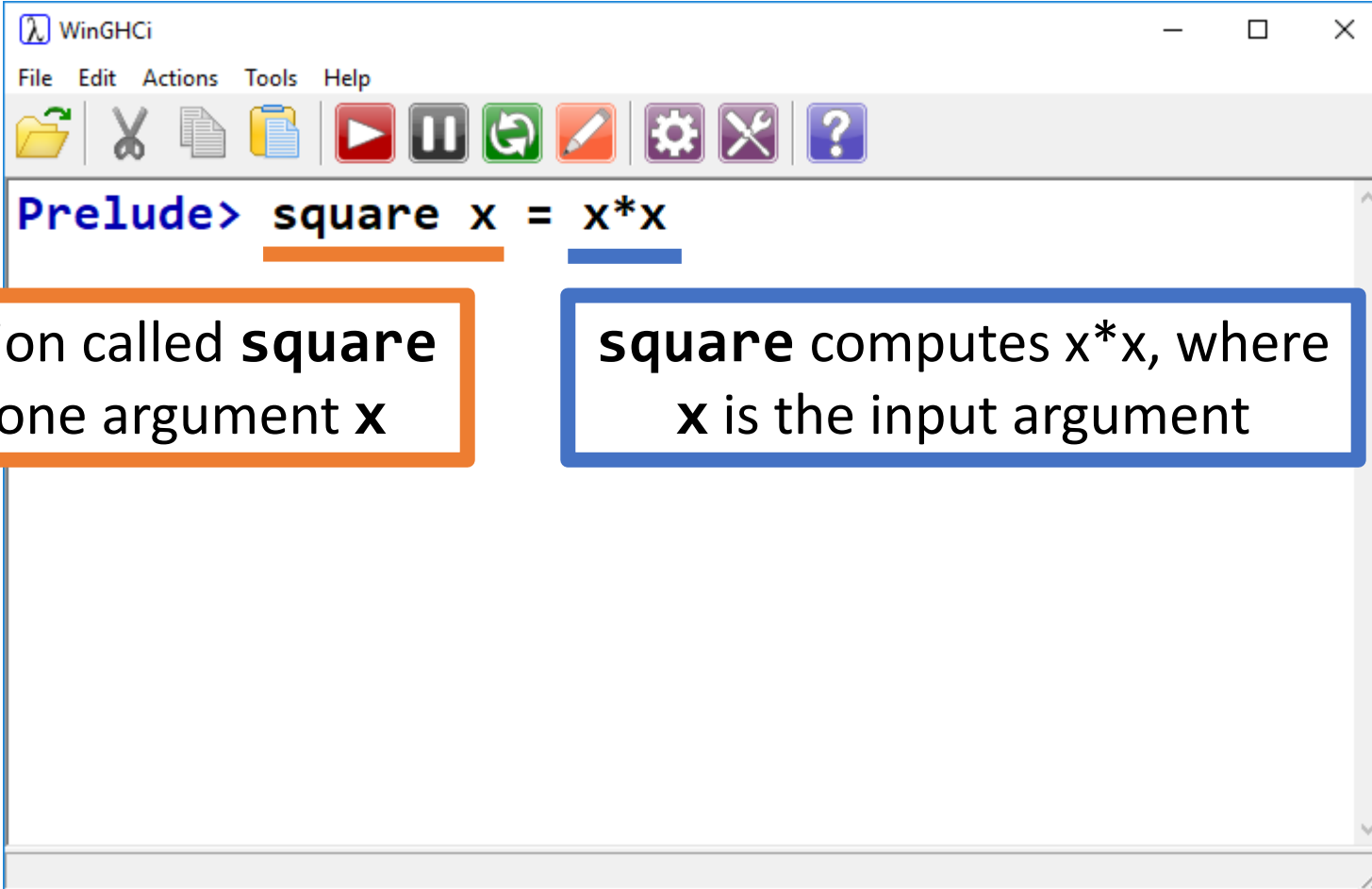


```
1 main = putStrLn "Hello, World!"
```

- 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:



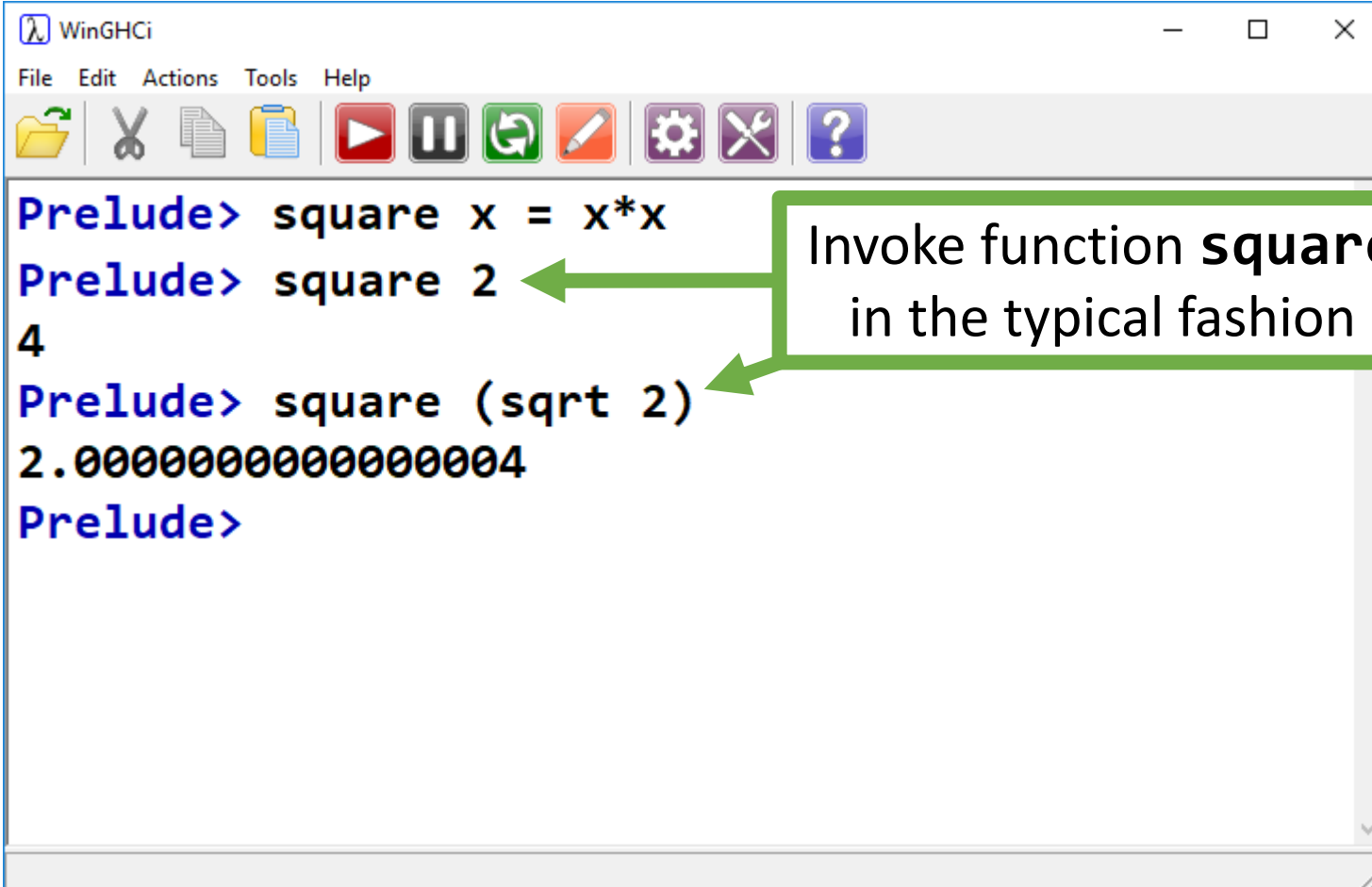
```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> square x = x*x
```

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:

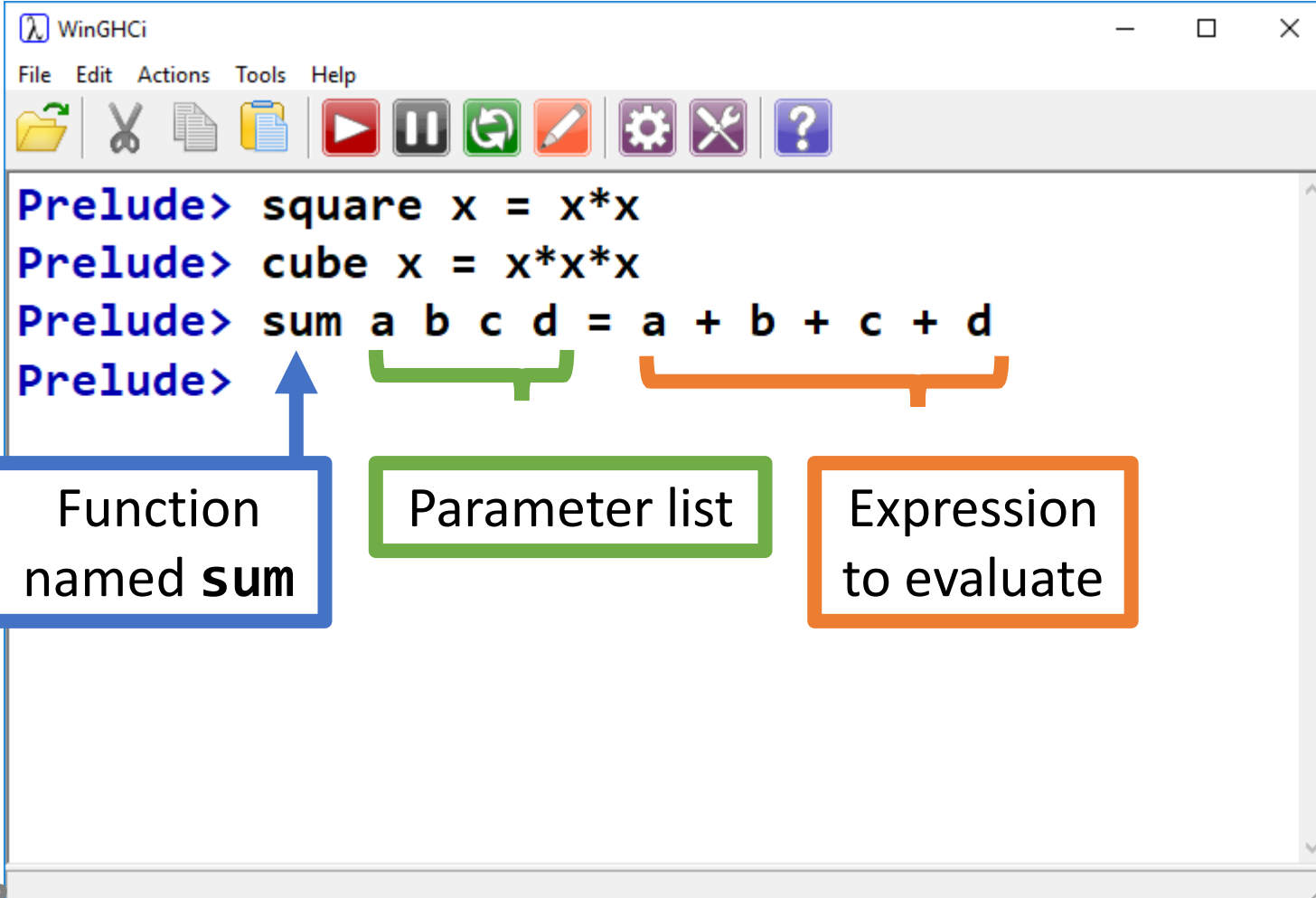


The screenshot shows a terminal window titled "WinGHCi" with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar with icons for file operations, execution, and help. The terminal content is as follows:

```
Prelude> square x = x*x
Prelude> square 2
4
Prelude> square (sqrt 2)
2.0000000000000004
Prelude>
```

A green callout box with a white background and a green border contains the text "Invoke function **square** in the typical fashion". Two green arrows point from this box to the `square 2` and `square (sqrt 2)` lines in the terminal.

Functions in Haskell



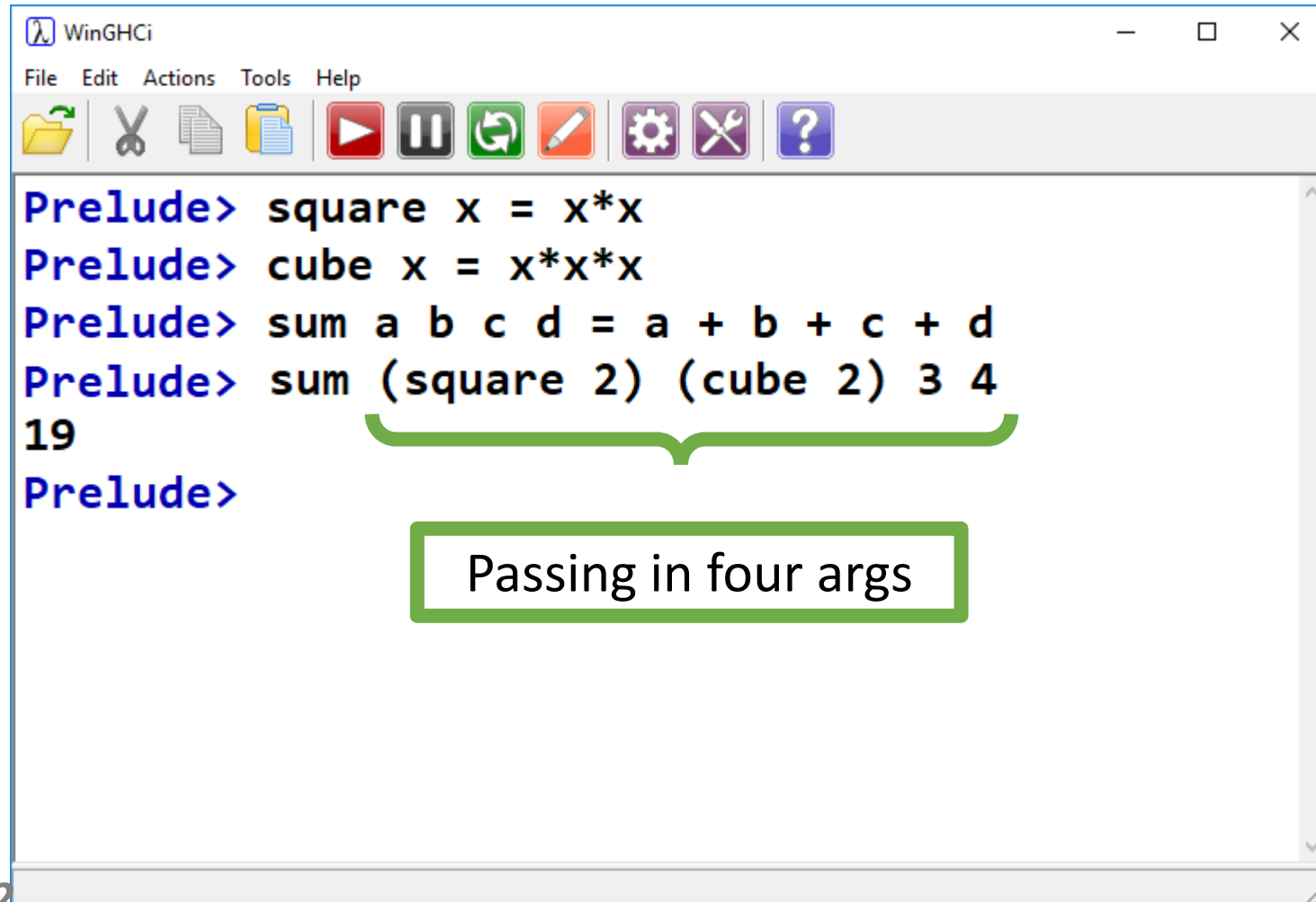
The screenshot shows the WinGHCi window with the following content:

```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> square x = x*x
Prelude> cube x = x*x*x
Prelude> sum a b c d = a + b + c + d
Prelude>
```

Annotations in the image:

- A blue box labeled "Function named **sum**" has an arrow pointing to the `sum` keyword in the third line.
- A green box labeled "Parameter list" has a bracket underneath `a b c d` in the third line.
- An orange box labeled "Expression to evaluate" has a bracket underneath `a + b + c + d` in the third line.

Functions in Haskell



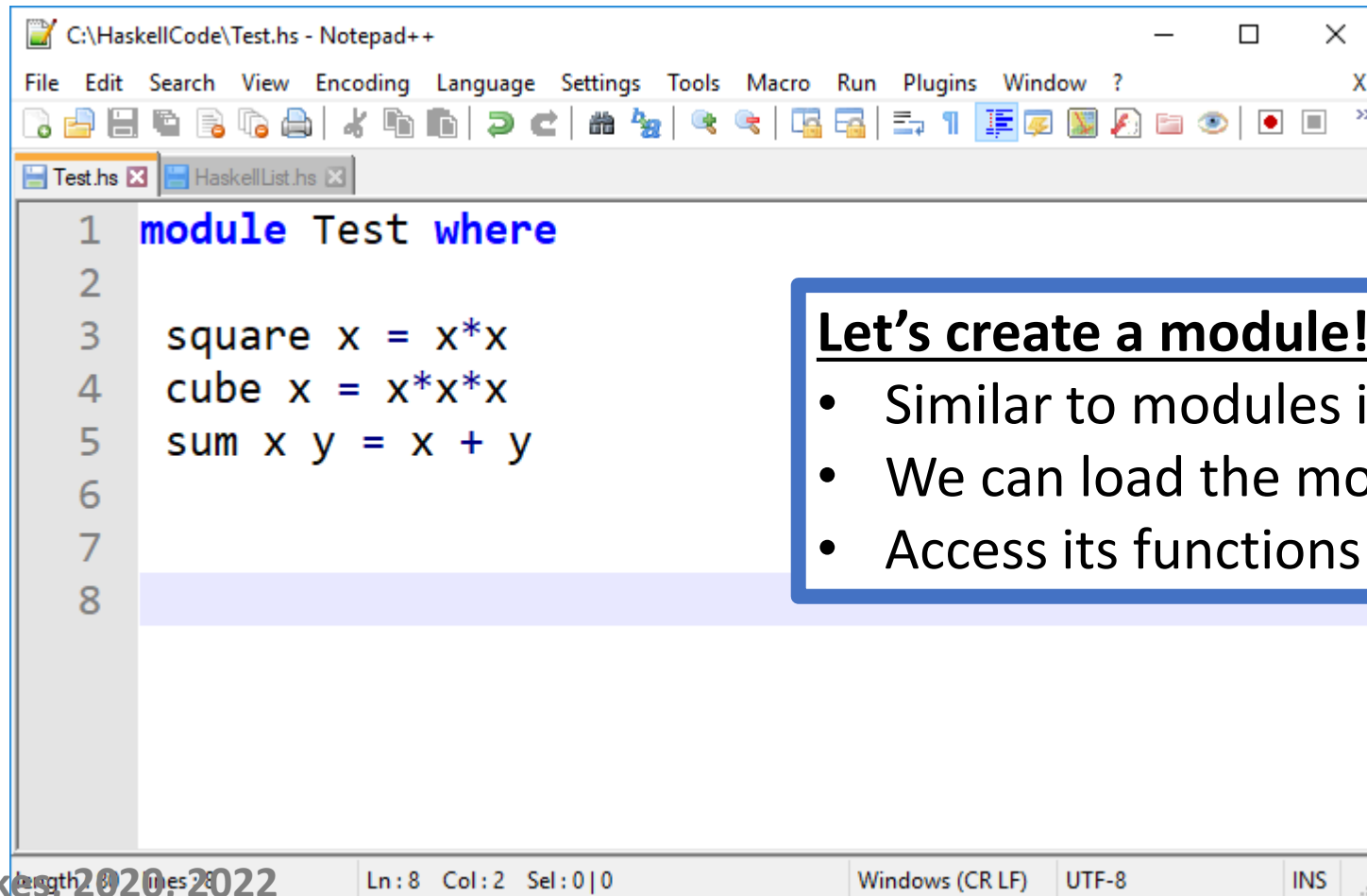
The screenshot shows the WinGHCi terminal window with the following content:

```
WinGHCi
File Edit Actions Tools Help
[Icons]
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>
```

A green bracket highlights the arguments `(square 2) (cube 2) 3 4` in the `sum` function call. Below the terminal, a green-bordered box contains the text "Passing in four args".

Haskell Modules

This is getting tedious to type interactively.



The screenshot shows a Notepad++ window with the following content:

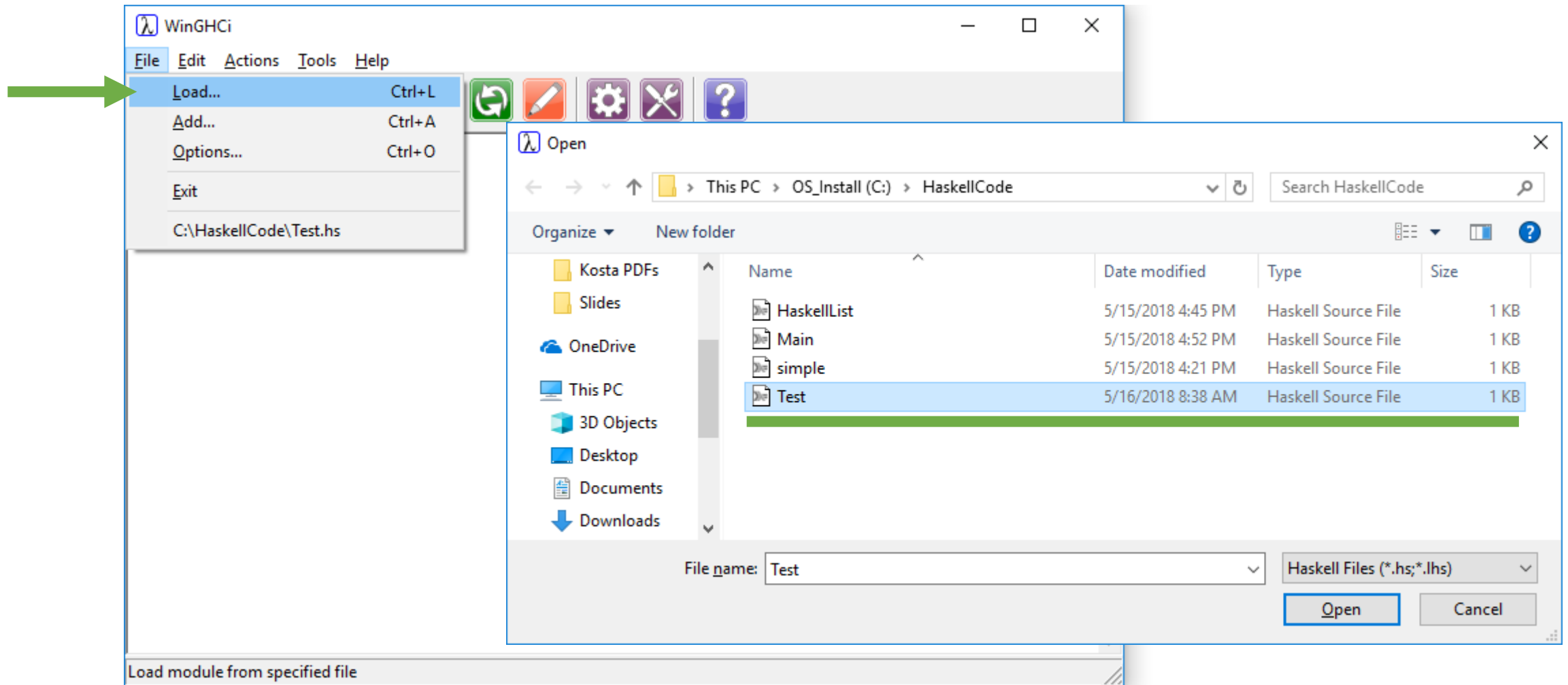
```
1 module Test where
2
3 square x = x*x
4 cube x = x*x*x
5 sum x y = x + y
6
7
8
```

The status bar at the bottom indicates: Length: 80, Lines: 8, Ln: 8, Col: 2, Sel: 0|0, Windows (CR LF), UTF-8, INS.

Let's create a module!

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

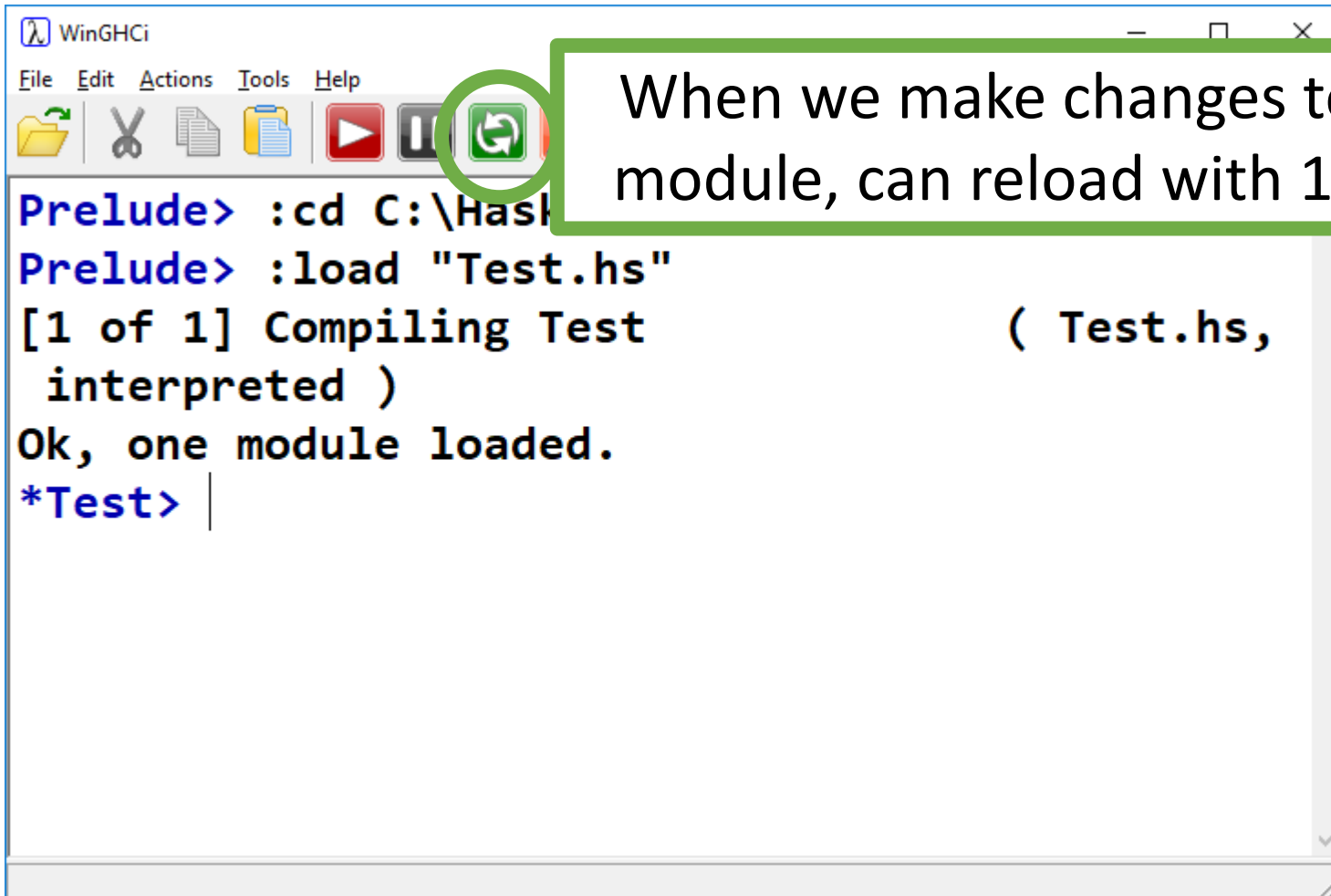
Loading a Module



```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> :cd C:\HaskellCode
Prelude> :load "Test.hs"
[1 of 1] Compiling Test                ( Test.hs,
interpreted )
Ok, one module loaded.
*Test> |
```



```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
[Icons]
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
8
```



WinGHCi

File Edit Actions Tools Help

[1 of 1] Compiling Test (Test.hs, interpreted)

Ok, one module loaded.

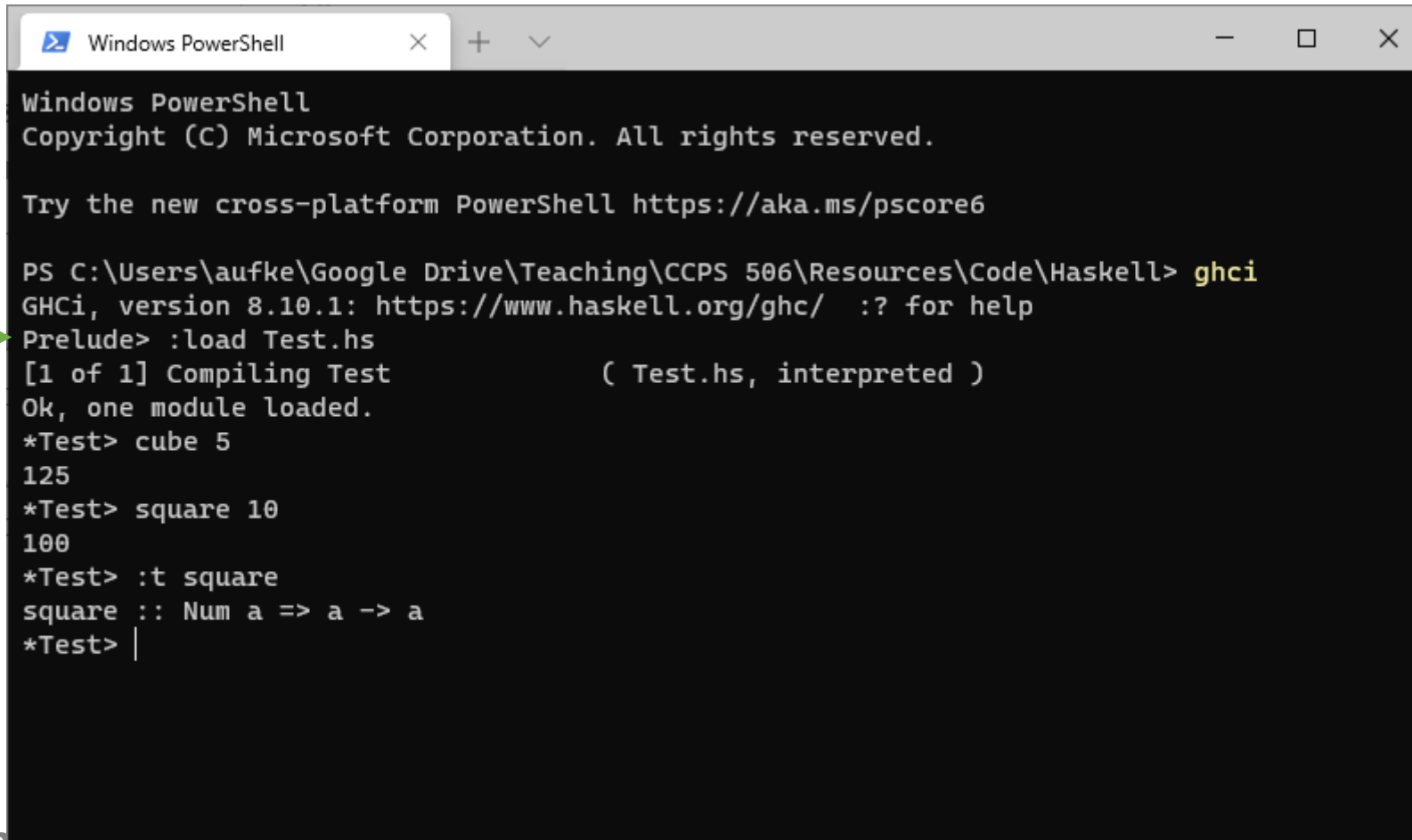
*Test> |

The screenshot shows a terminal window titled 'WinGHCi' with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar. The toolbar contains icons for file operations (folder, copy, paste) and execution (play, stop, reload). The reload icon, a green square with a white circular arrow, is circled in green. A green-bordered text box to the right of the toolbar contains the text: 'When we make changes to Test module, can reload with 1 click!'. The terminal output shows the user navigating to a directory and loading 'Test.hs', which is then compiled and interpreted. The prompt changes from 'Prelude>' to '*Test>'.

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

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

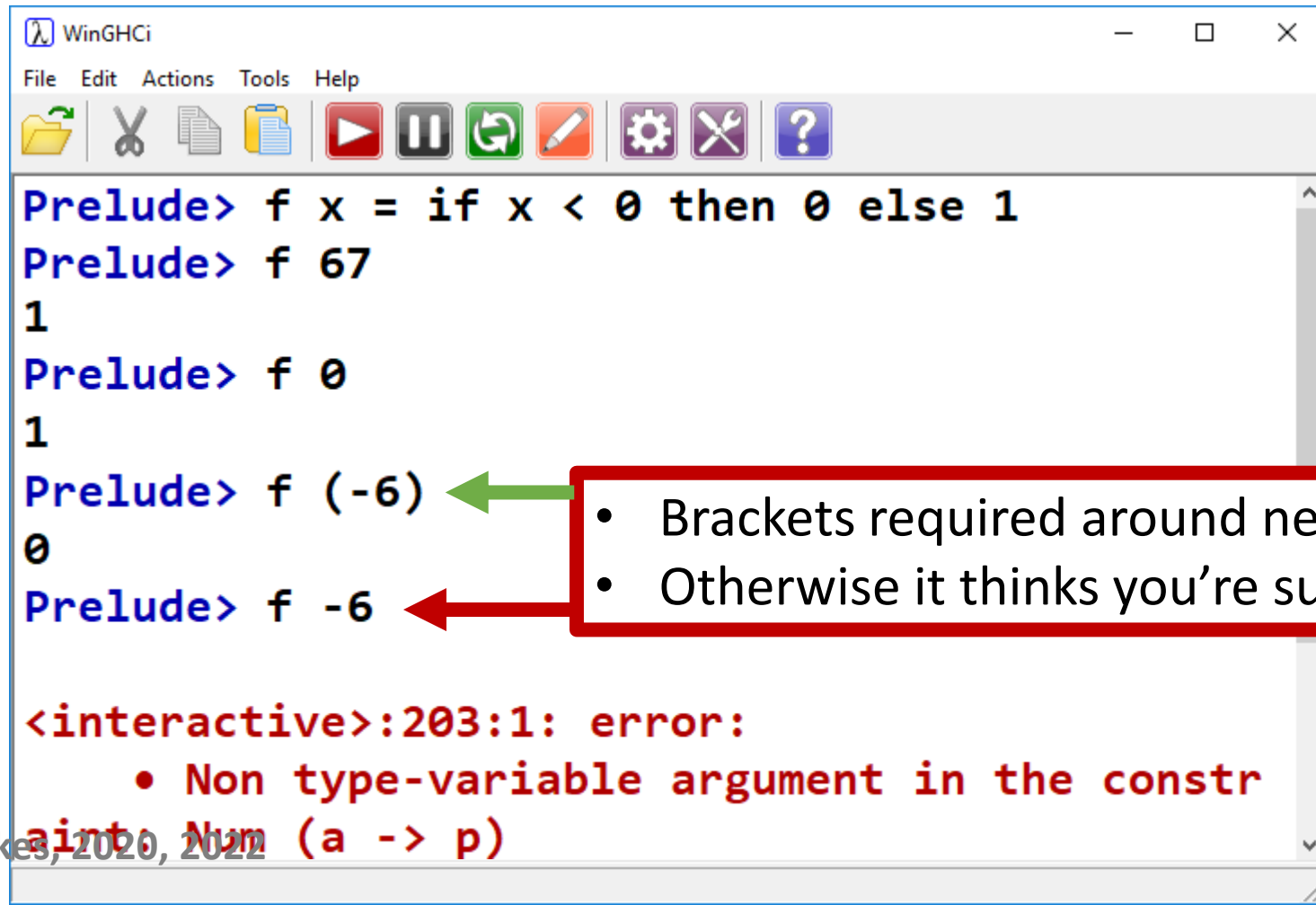
PS C:\Users\aufke\Google Drive\Teaching\CCPS 506\Resources\Code\Haskell> ghci
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
Prelude> :load Test.hs
[1 of 1] Compiling Test                ( Test.hs, interpreted )
Ok, one module loaded.
*Test> cube 5
125
*Test> square 10
100
*Test> :t 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
[Icons]
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 constraint 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

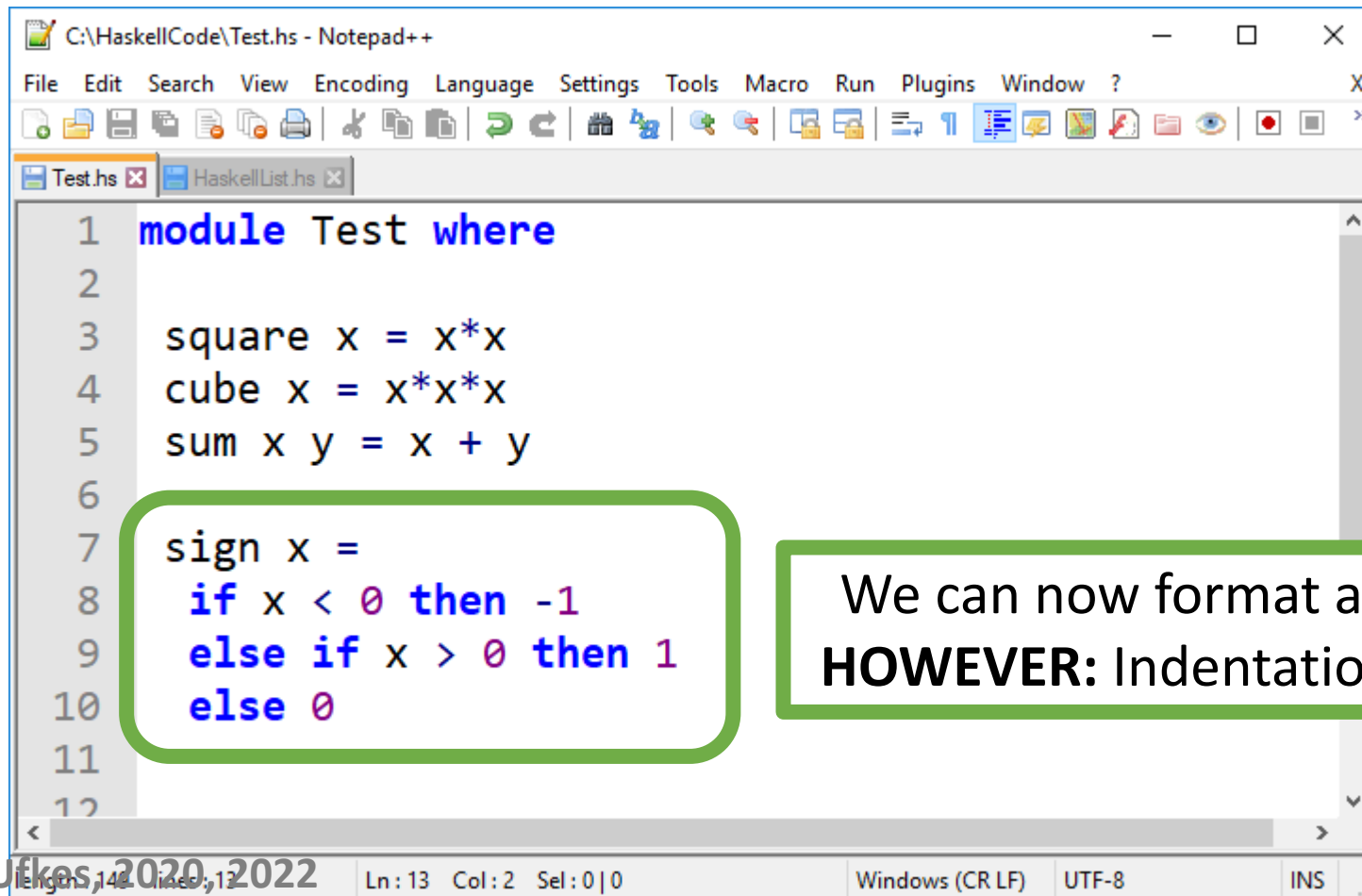
```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plu
Test.hs HaskellList.hs
1 module Test where
2
3 sign x =
4   if x < 0 then -1
5   else if x > 0 then 1
6   else 0
7
8
9
length: 95 lines: 9
Windows (CR LF) UTF-8 INS
```

- 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



```
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
12
```

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

© Alex Ufkes, 2020, 2022

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs HaskellList.hs
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: 92 lines: 9 Ln: 6 Col: 2 Sel: 0|0
```

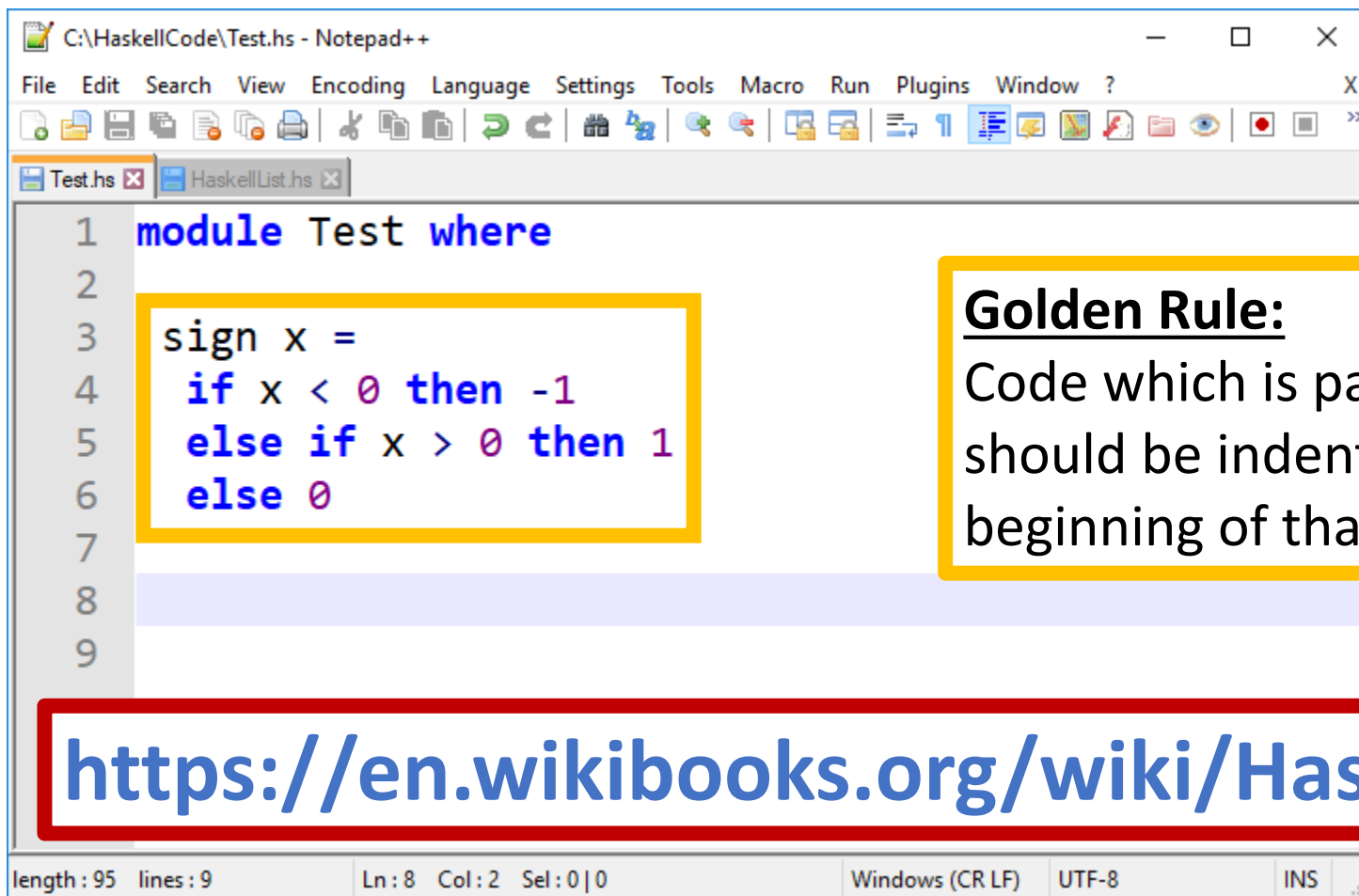
```
WinGHCi
File Edit Actions Tools Help
Prelude> :reload

Test.hs:4:2: error:
    parse error (possibly incorrect indentation
or mismatched brackets)
   |
4 |  if x < 0 then -1 | ^
[1 of 1] Compiling Test      ( Test.hs,
interpreted )
Failed, no modules loaded.
Prelude> |
```

```
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: 4 Col: 3 Sel: 0|0
```

```
WinGHCi
File Edit Actions Tools Help
n or mismatched brackets)
4 | if x < 0 then -1 | ^
[1 of 1] Compiling Test (Test.hs,
interpreted )
Failed, no modules loaded.
Prelude> :reload
[1 of 1] Compiling Test (Test.hs,
interpreted )
Ok, one module loaded.
*Test>
```

Indenting in Haskell



The screenshot shows a Notepad++ window with the following code:

```
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
```

The code is indented to illustrate the 'Golden Rule' of Haskell indentation. The code is displayed in a Notepad++ window with a menu bar (File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, Plugins, Window) and a toolbar. The status bar at the bottom shows '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

```
*C:\_cps506\haskell\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
Test.hs x
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 | Ln: 4 | Col: 12 | Sel: 0 | 0
Windows (CR LF) UTF-8 INS
```

When binding a name inside a function or module, we use the “**let**” keyword

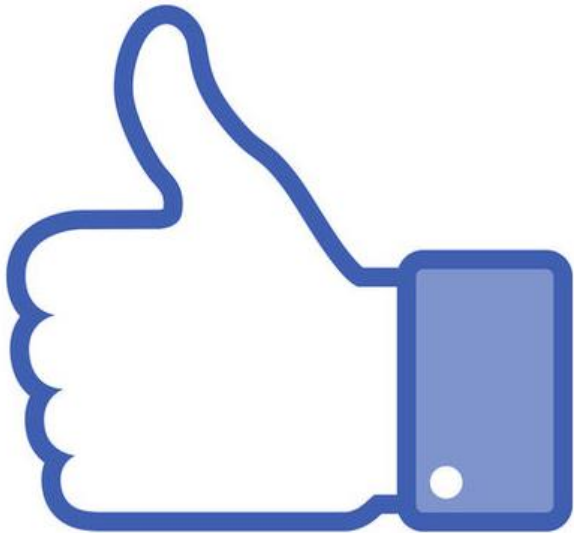
Multiple Expressions

```
*C:\_cps506\haskell\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
Test.hs x
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 lln: 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 x
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 |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

```
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 isNum x =
4   case x of
5     0 -> 0
6     1 -> 1
7     2 -> 2
8     _ -> -1
9
10 sign x = do
11   let q = x
12       if x < 0 then -1
13       else if x > 0 then 1
```

length: 184 lines: 17 Ln: 14 Col: 2 Sel: 0|0 Windows (CR LF) UTF-8 INS

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

Case Expression

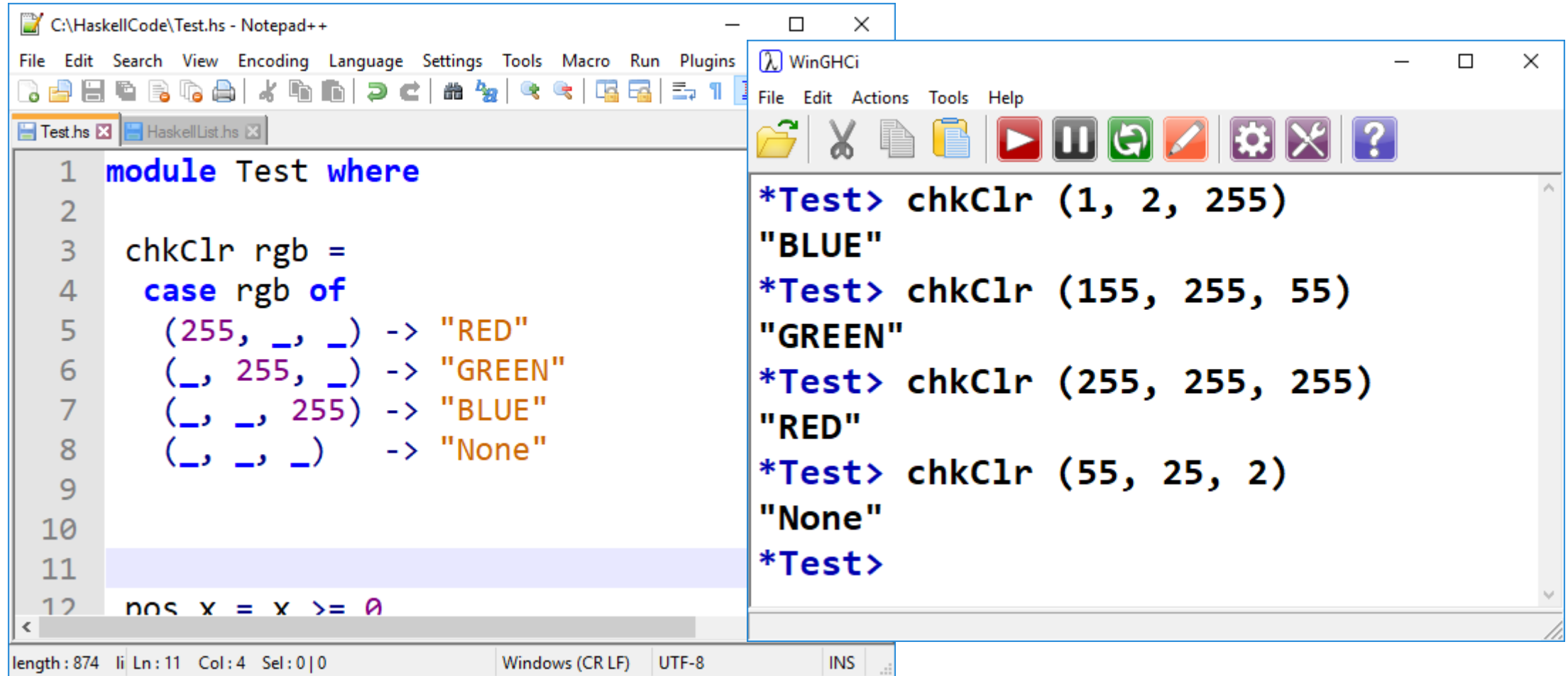
The image shows two windows side-by-side. The left window is Notepad++ editing a file named 'Test.hs'. The code defines a module 'Test' with a 'case' expression 'isNum' and a 'do-let-if' expression 'sign'. The right window is WinGHCi, a Haskell interpreter, showing the execution of the code. It displays the output of ':reload', 'isNum 0', 'isNum 1', 'isNum 2', and 'isNum 999'.

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run
Test.hs HaskellList.hs
1 module Test where
2
3 isNum x =
4   case x of
5     0 -> 0
6     1 -> 1
7     2 -> 2
8     _ -> -1
9
10 sign x = do
11   let q = x
12   if x < 0 then -1
13   else if x > 0 then 1

length: 184 lines: 17 Ln: 14 Col: 2 Sel: 0|0

WinGHCi
File Edit Actions Tools Help
*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



The image shows two windows side-by-side. The left window is Notepad++ editing a file named Test.hs. The code defines a function chkClr that uses a case expression to match RGB values. The right window is WinGHCi, a Haskell interpreter, showing the execution of the function with various arguments and the resulting color strings.

```
1 module Test where
2
3 chkClr rgb =
4   case rgb of
5     (255, _, _) -> "RED"
6     (_, 255, _) -> "GREEN"
7     (_, _, 255) -> "BLUE"
8     (_, _, _)   -> "None"
9
10
11
12 nos x = x >= 0
```

```
*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

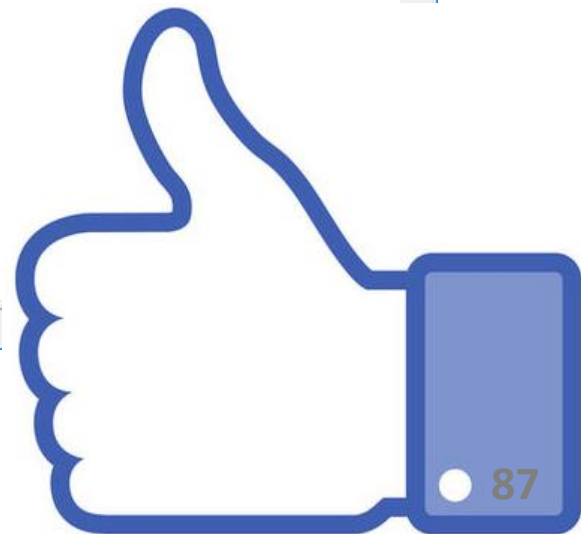
```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plu
Test.hs HaskellList.hs
1 module Test where
2
3 chkClr rgb =
4   case rgb of
5     (255, _, _) -> "RED"
6     (_, 255, _) -> "GREEN"
7     (_, _, 255) -> "BLUE"
8     (255, 255, _) -> "YELLOW"
9     (255, _, 255) -> "MAGENTA"
10    (_, 255, 255) -> "CYAN"
11
12
length: 971 lin Ln: 15 Col: 4 Sel: 0|0 Windows (CR LF) UTF-
```

Will never match!

```
WinGHCi
File Edit Actions Tools Help
[1 of 1] Compiling Test (Test.hs, interpreted)
Test.hs:8:4: warning: [-Woverlapping-patterns]
    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
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
11
12
```

↑

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 'chkClr',
      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
1 module Test where
2
3 fac 0 = 1
4 fac x = x*fac(x-1)
5
6 fib 0 = 0
7 fib 1 = 1
8 fib n = fib(n-1) + fib(n-2)
9
10
11 isNum x =
12 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!

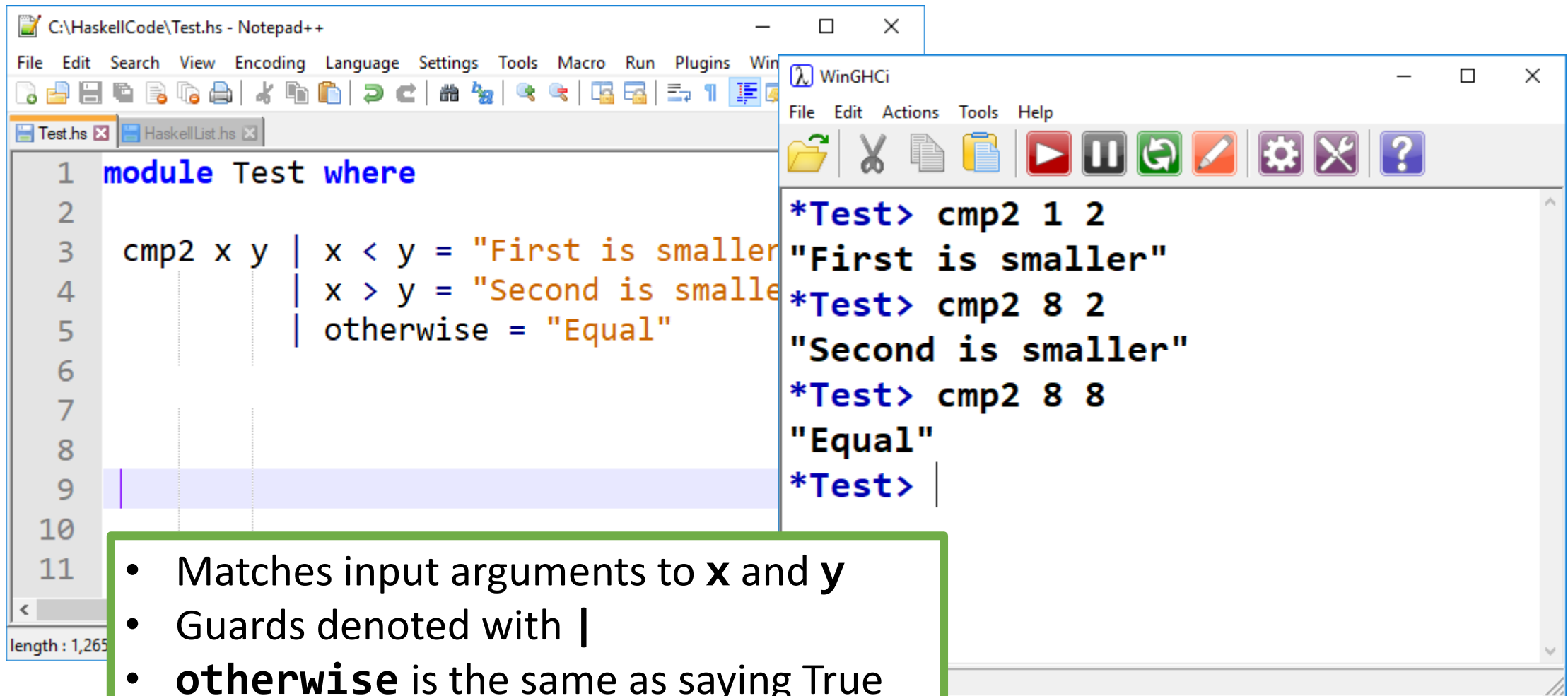
Piecewise Functions

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs HaskellList.hs
1 module Test where
2
3   chkAxis (0, _) = (0, 1)
4   chkAxis (_, 0) = (1, 0)
5   chkAxis (a, b) = (a, b)
6
7
length: 1,061 | Ln: 8 Col: 2 Sel: 0 | 0 Windows (CR LF) UTF-8 INS
```

```
WinGHCi
File Edit Actions Tools Help
*Test> chkAxis (1, 1)
(1,1)
*Test> chkAxis (8, 0)
(1,0)
*Test> chkAxis (42.3338, 0)
(1.0,0)
*Test> |
```

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

Functions: Guards



The image shows two overlapping windows. The background window is Notepad++ with a file named 'Test.hs' open. The code in the editor is as follows:

```
1 module Test where
2
3 cmp2 x y | x < y = "First is smaller"
4          | x > y = "Second is smaller"
5          | otherwise = "Equal"
6
7
8
9
10
11
```

The foreground window is WinGHCi, which shows the execution of the code. The prompt is `*Test>`. The output is:

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

A green box highlights the following list of bullet points:

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

Recursion

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Wi
Test.hs HaskellList.hs
1 module Test where
2
3 llen [] = 0
4 llen x = 1 + llen(tail x)
5
6
7
8
9
10
11
12
13
14
```

Classic list length finder

```
WinGHCi
File Edit Actions Tools Help
*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

The image shows two windows side-by-side. The left window is Notepad++ with a Haskell file named Test.hs. The code defines a recursive function `llen` that calculates the length of a list. The right window is WinGHCi, a Haskell interpreter, showing the execution of the code. A green box highlights the recursive call in the code, and a blue arrow points from this box to the corresponding line in the WinGHCi output.

```
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 llen [] = 0
4 llen (xh:xt) = 1 + (llen xt)
5
11
12
13
14
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> |
```

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

Recursion: filter

```
*C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings To
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
14
```

- Returns true if $x \geq 0$
- 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.

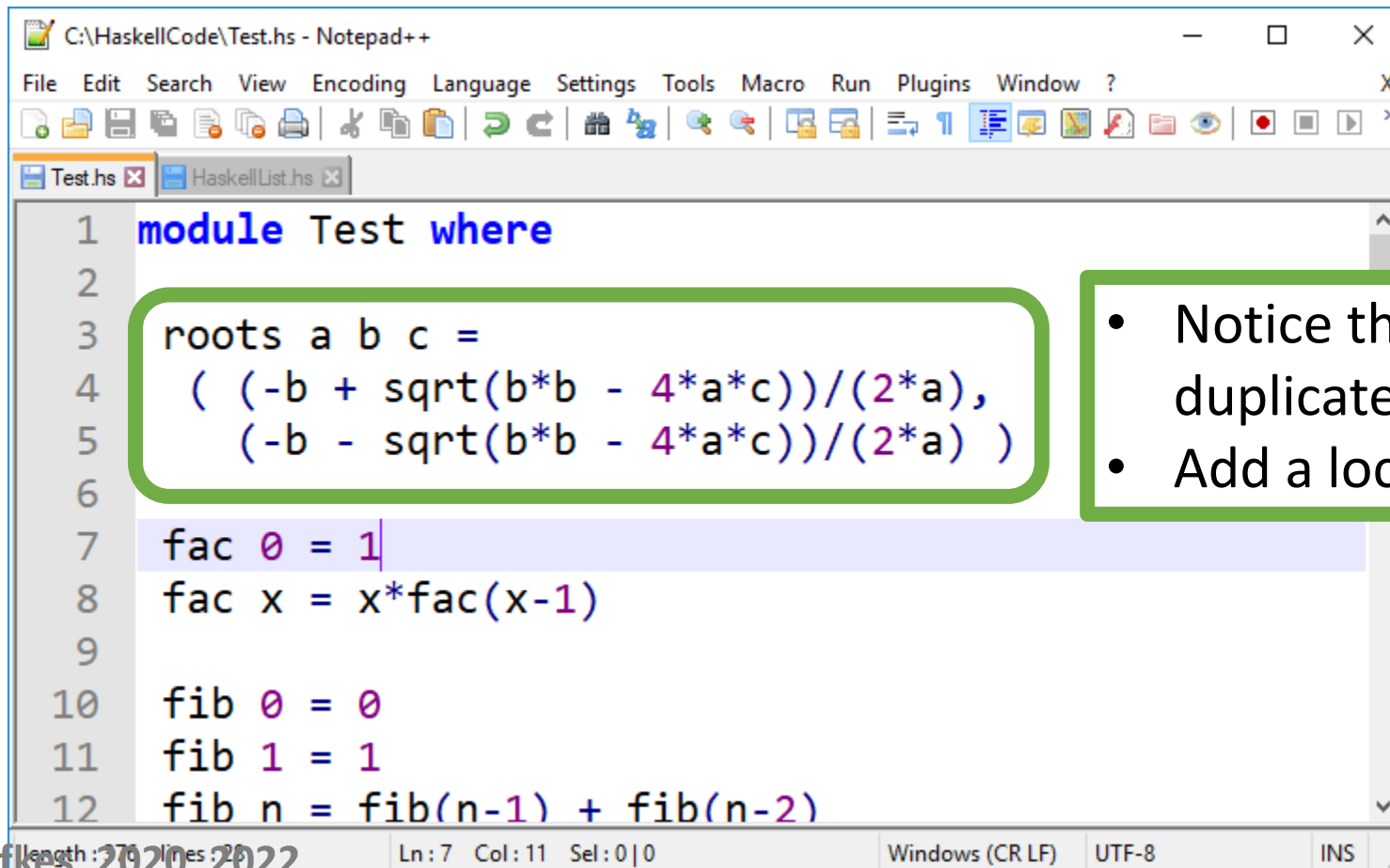
Recursion: filter

```
*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
14
```

```
WinGHCi
File Edit Actions Tools Help
Test pos function
*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> |
```


Return Multiple *Things*?

Lists/tuples to the rescue!



```
1 module Test where
2
3 roots a b c =
4   ( (-b + sqrt(b*b - 4*a*c))/(2*a),
5     (-b - sqrt(b*b - 4*a*c))/(2*a) )
6
7 fac 0 = 1
8 fac x = x*fac(x-1)
9
10 fib 0 = 0
11 fib 1 = 1
12 fib n = fib(n-1) + fib(n-2)
```

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

let/in Expression

```
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 roots a b c =
4   let disc = sqrt(b*b - 4*a*c)
5   in  ((-b + disc)/(2*a),
6       (-b - disc)/(2*a))
7
8 fac 0 = 1
9 fac x = x*fac(x-1)
10
11 fib 0 = 0
12 fib 1 = 1
length: 387 lines: 29 Ln: 10 Col: 2 Sel: 0|0 Windows (CR LF) UTF-8 INS
```

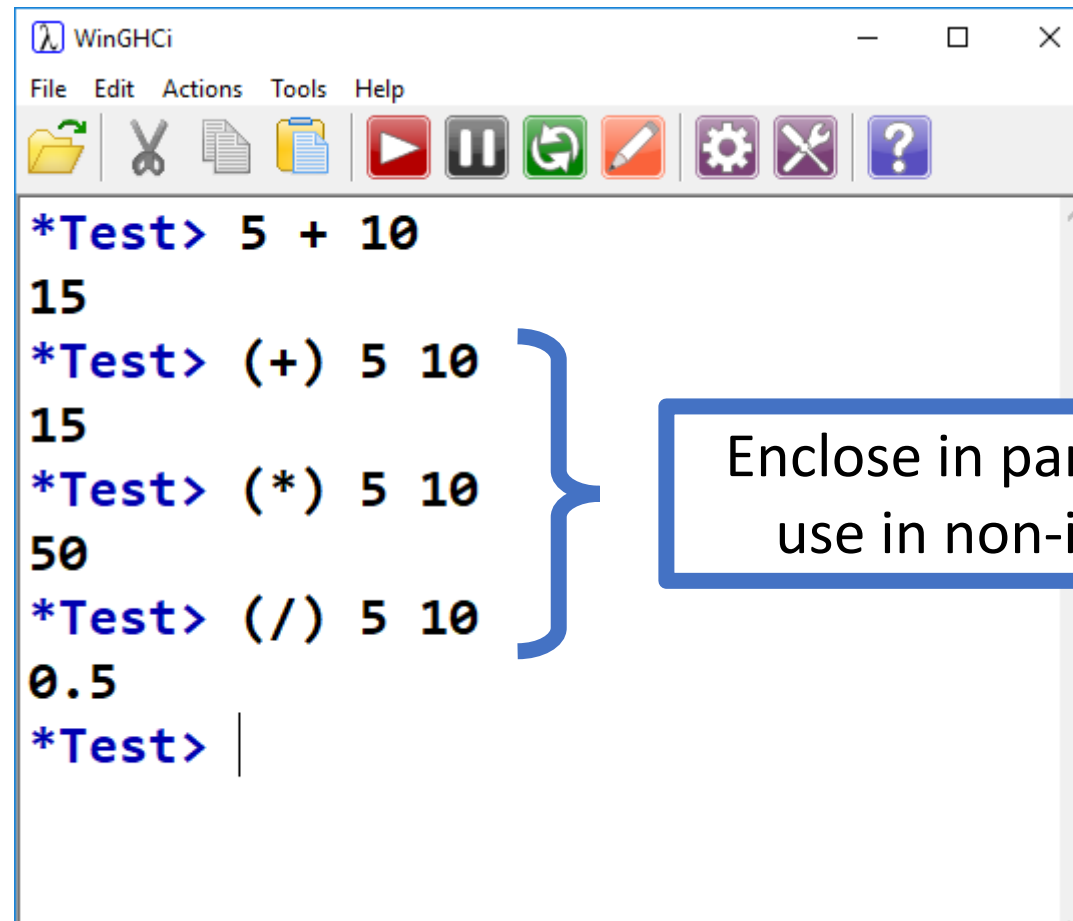
- This is one expression (**let/in**)
- We don't need to add **do** keyword

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs HaskellList.hs
1 module Test where
2
3 roots a b c =
4   let disc = sqrt(b*b - 4*a*c)
5       in ((-b + disc)/(2*a),
6           (-b - disc)/(2*a))
7
8 fac 0 = 1
9 fac x = x*fac(x-1)
10
11 fib 0 = 0
12 fib 1 = 1
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, 3.2360679774997898)
*Test> |
```

Infix Functions

Use symbolic operators as functions:



The screenshot shows a terminal window titled 'WinGHCi' with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar with icons for file operations, execution, and settings. The terminal output is as follows:

```
*Test> 5 + 10
15
*Test> (+) 5 10
15
*Test> (*) 5 10
50
*Test> (/) 5 10
0.5
*Test> |
```

A blue bracket on the right side of the terminal groups the three lines: `(+) 5 10`, `(*) 5 10`, and `(/) 5 10`. A blue-bordered box points to this bracket with the text: "Enclose in parentheses to use in non-infix mode".

Function Composition

```
WinGHCi
File Edit Actions Tools Help
[Icons]
*Test> fac(fib(4))
6
*Test> fac(fib(5))
120
*Test> fac(fib(6))
40320
*Test> |
```

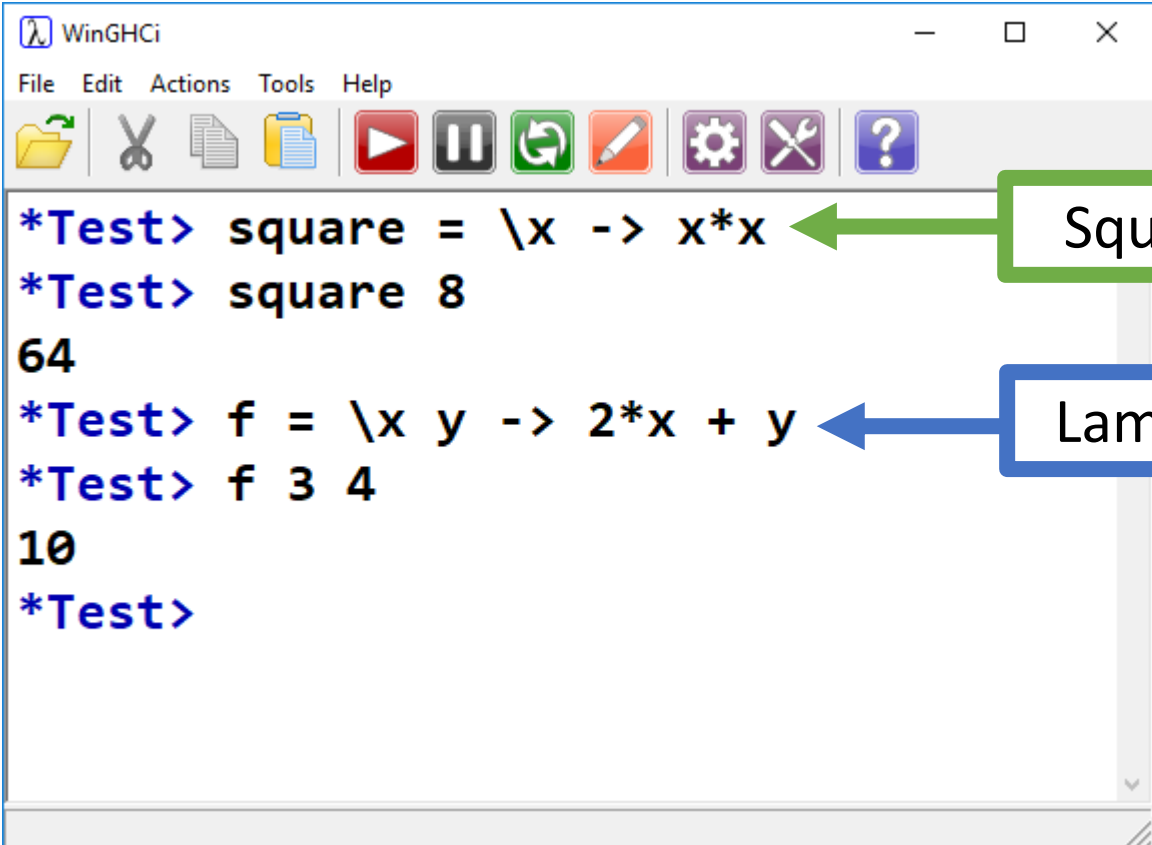
Can be written as:

```
WinGHCi
File Edit Actions Tools Help
[Icons]
*Test> (fac.fib) 4
6
*Test> (fac.fib) 5
120
*Test> (fac.fib) 6
40320
*Test> |
```

In math, $f \circ g$ means “*f* following *g*”. Same thing in Haskell.

Lambda Functions

Like anonymous functions in Elixir:



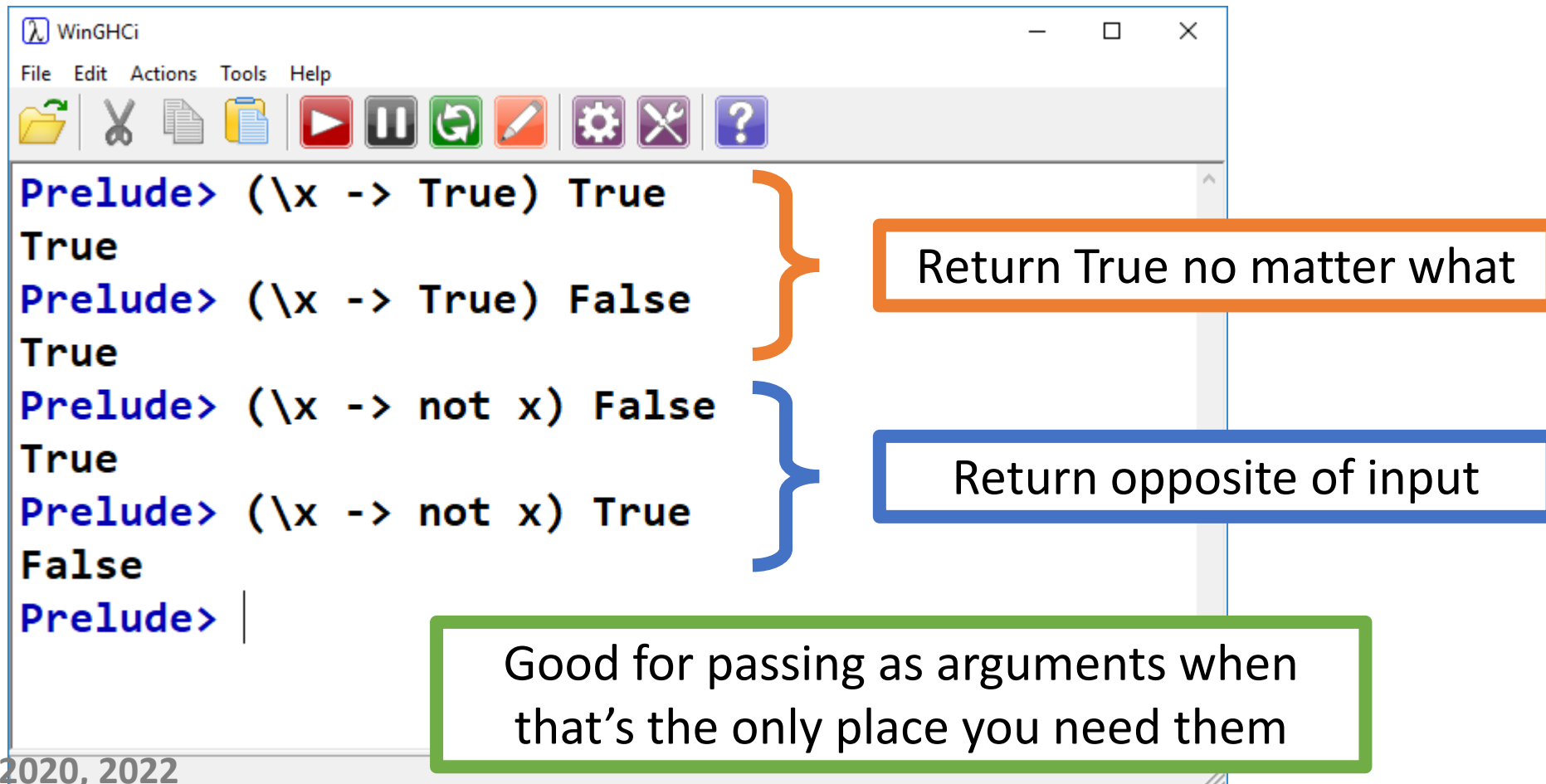
```
WinGHCi
File Edit Actions Tools Help
[Icons]
*Test> square = \x -> x*x
*Test> square 8
64
*Test> f = \x y -> 2*x + y
*Test> f 3 4
10
*Test>
```

Square as Lambda function

Lambda function with two args

Lambda Functions

They don't need names!



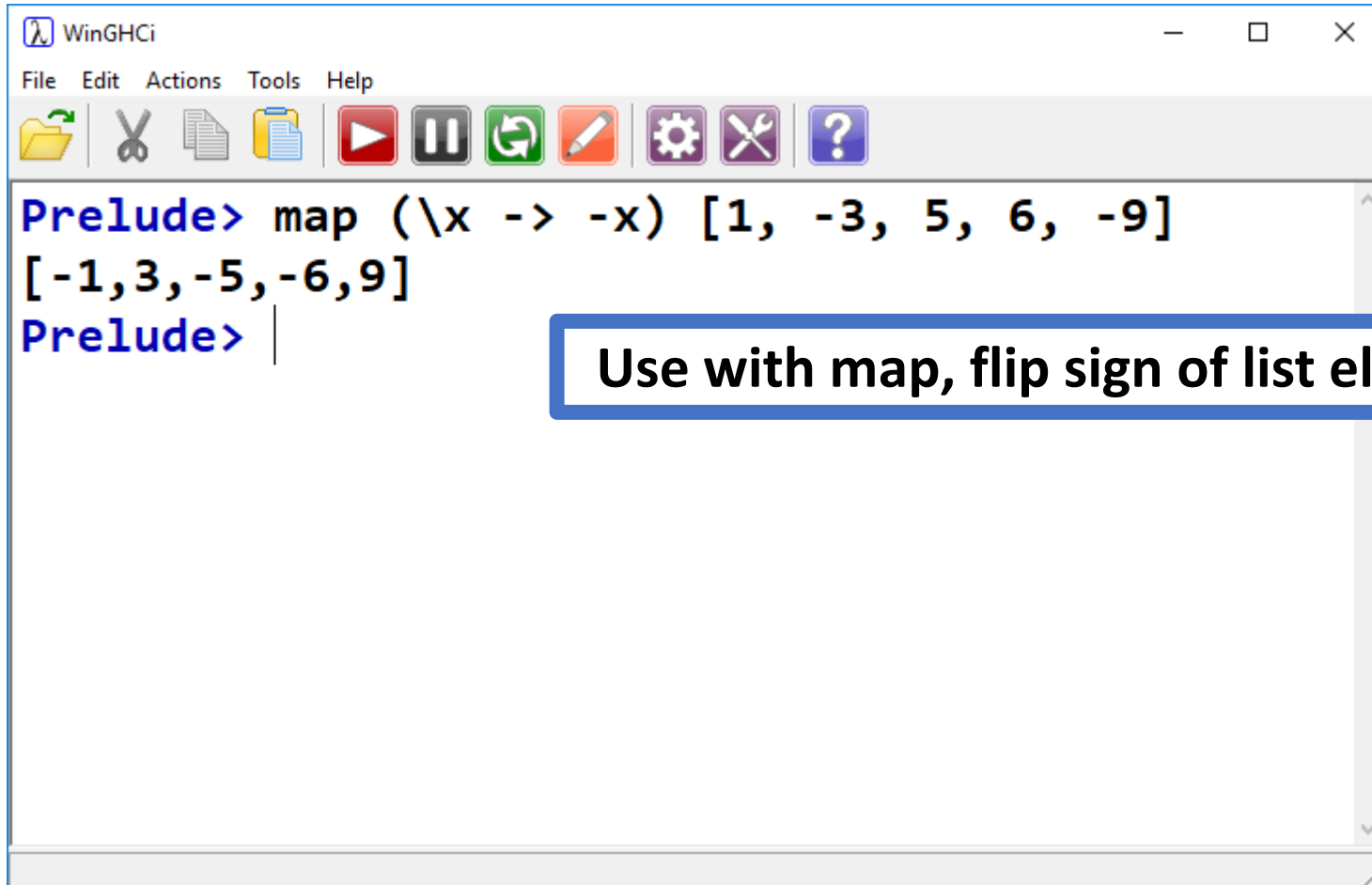
The screenshot shows the WinGHCi terminal window with the following content:

```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> (\x -> True) True
True
Prelude> (\x -> True) False
True
Prelude> (\x -> not x) False
True
Prelude> (\x -> not x) True
False
Prelude> |
```

Callouts:

- An orange bracket groups the first two lines of code, pointing to a box: "Return True no matter what".
- A blue bracket groups the last two lines of code, pointing to a box: "Return opposite of input".
- A green box at the bottom contains the text: "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



```
WinGHCi
File Edit Actions Tools Help
[Icons: Folder, Scissors, Document, Clipboard, Play, Pause, Refresh, Eraser, Gear, Wrench, Question Mark]
Prelude> map (\x -> -x) [1, -3, 5, 6, -9]
[-1,3,-5,-6,9]
Prelude> |
```

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>

