

C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 6: Haskell intro, Haskell basics

The logo for Toronto Metropolitan University, featuring a blue square with the text "Toronto Metropolitan University" in white, and a yellow square to its right.









**Toronto
Metropolitan
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

  CCPS506 - Comparative Programming La...      Alexander Ufkes 

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

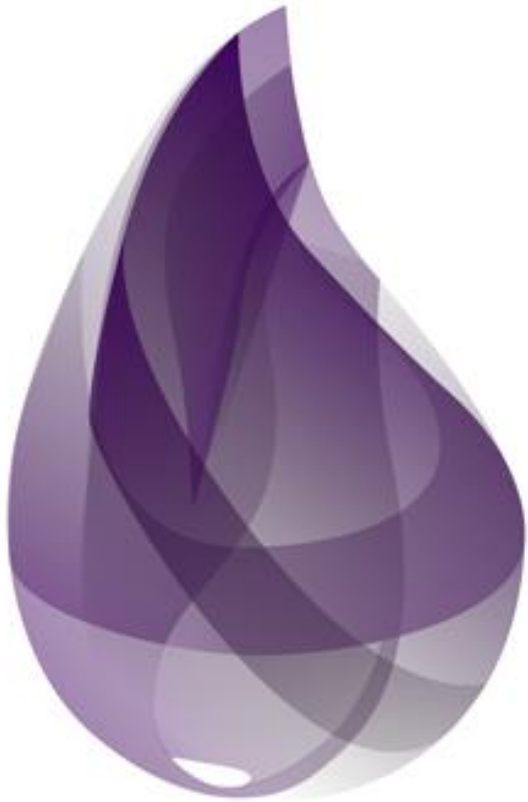
Two languages down, two to go!

Today

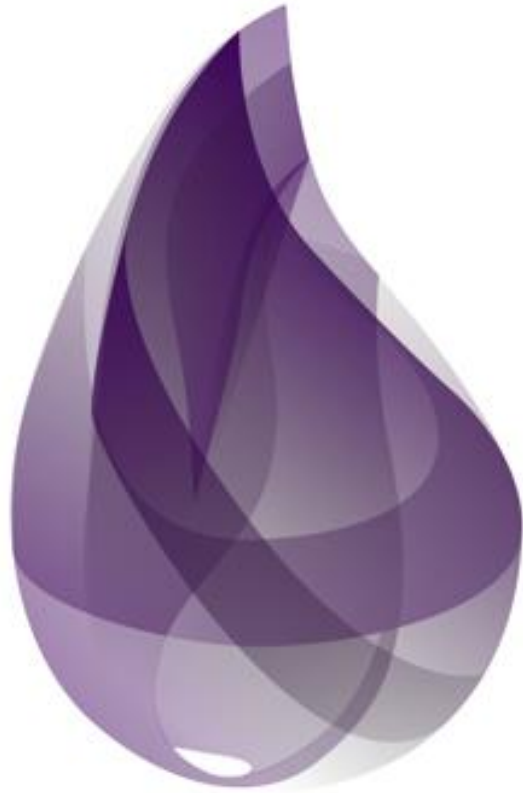
Intro to Haskell

- Pure functional
- Haskell basics
- Functions
- Control flow

Functional Programming



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

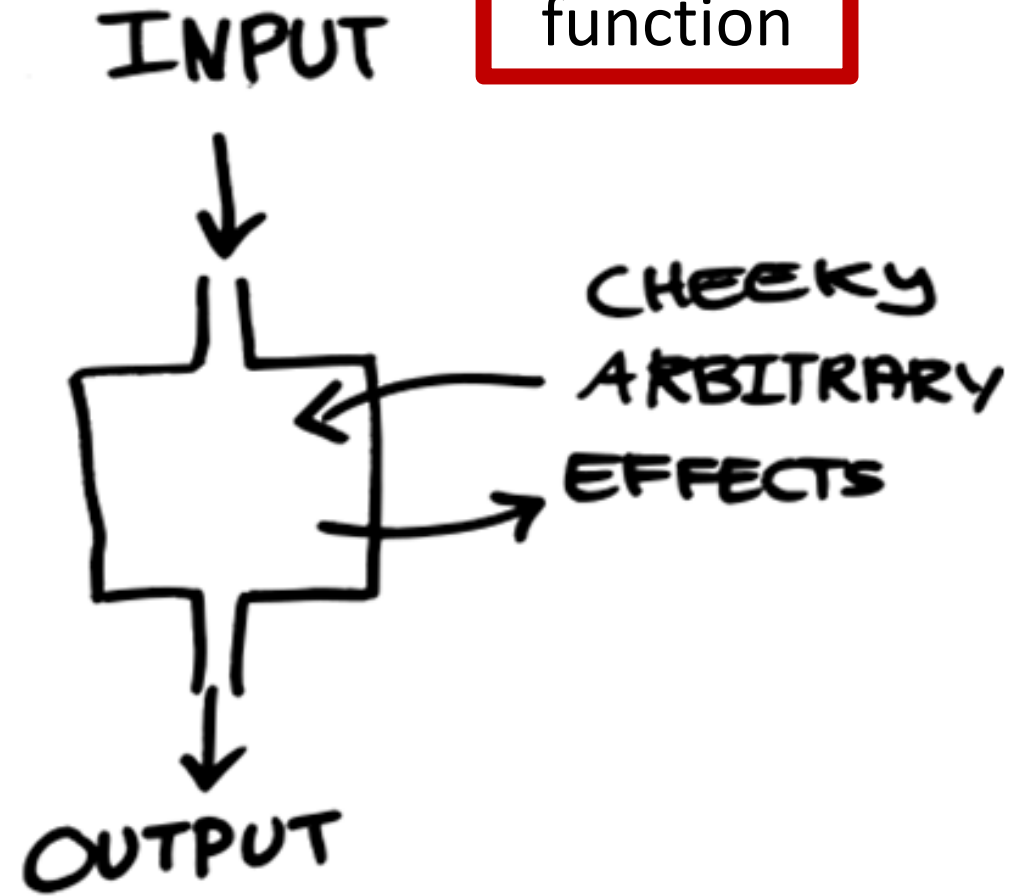
Pure Functions:

- 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



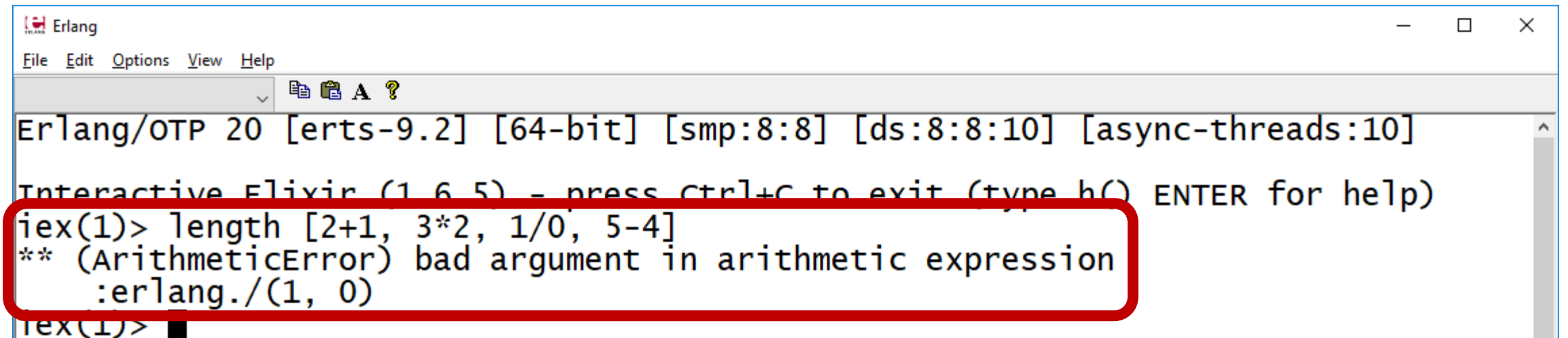
Impure
function



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 status bar shows icons for a file, a folder, a magnifying glass, and a question mark. The main text area displays the following text:

```
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)>
```

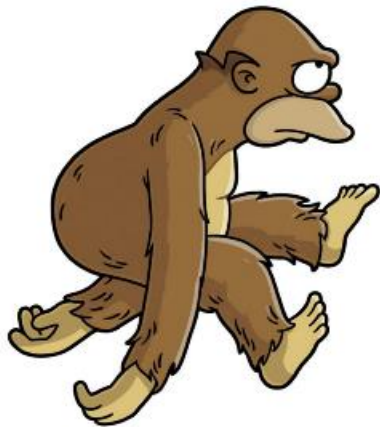
The error message is highlighted with a red rounded rectangle.

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

Functional Programming



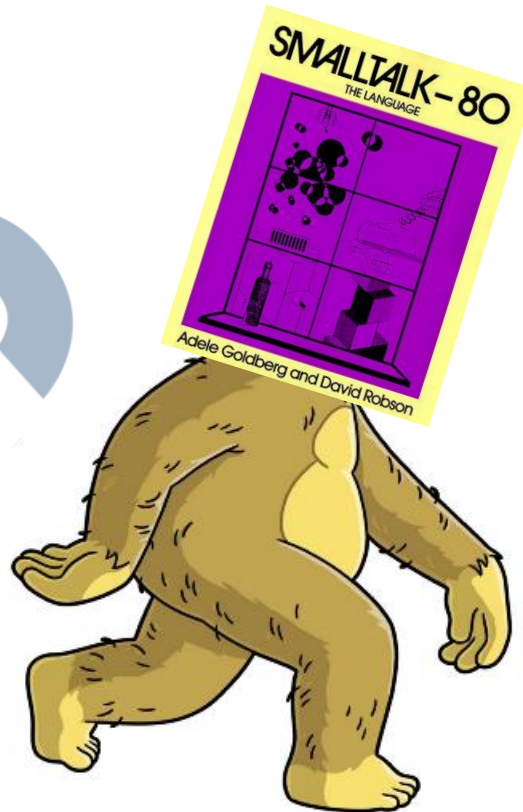
MACHINE



ASSEMBLY



PROCEDURAL

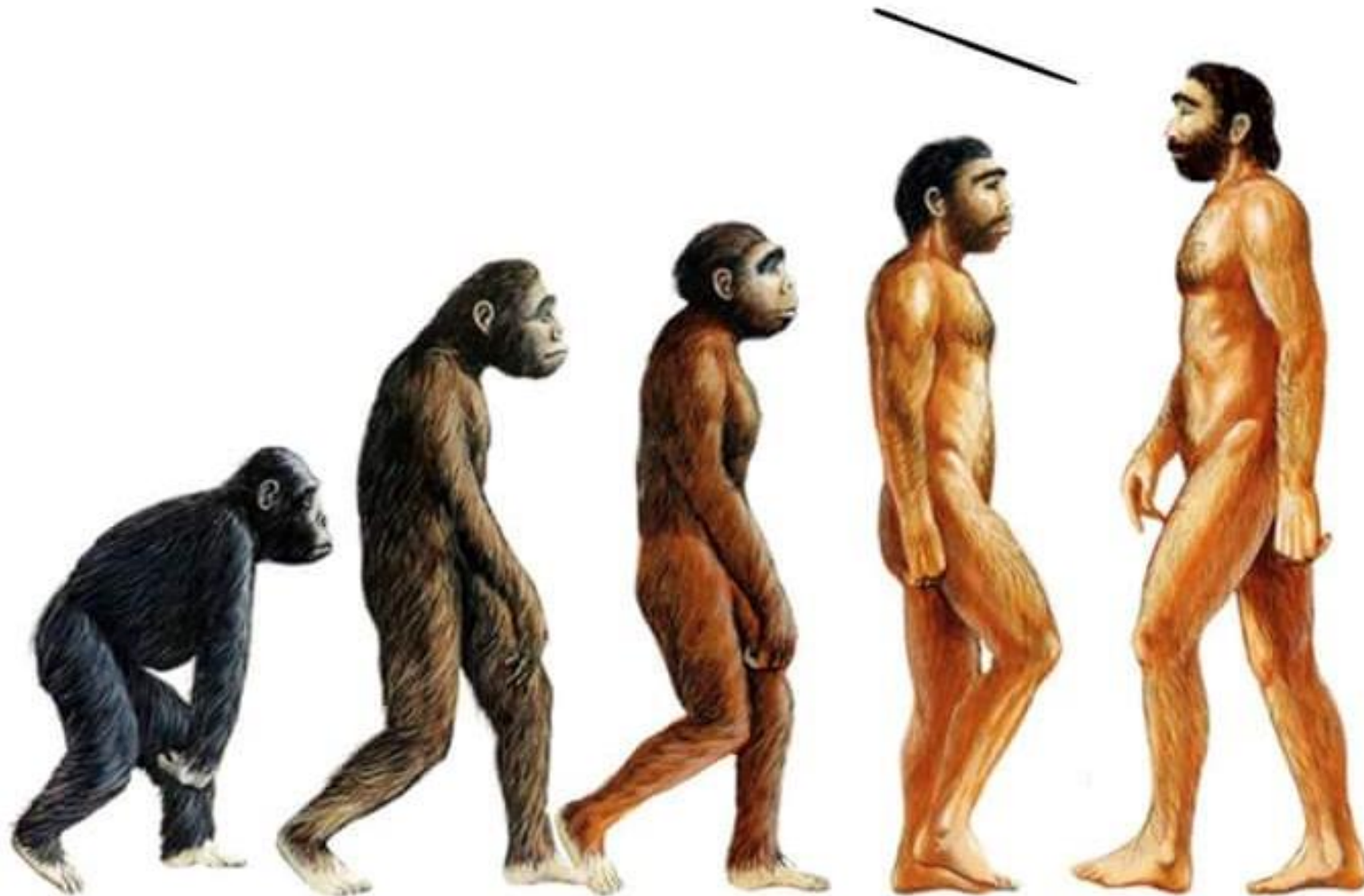


OBJECT ORIENTED



FUNCTIONAL

Go back. We f*cked up.



Functional Programming

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

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

Try it!

Type Haskell expressions in here.

```
λ length [2+1, 3*2, 1/0, 5-4]
4 :: Int
λ █
```

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

Concurrency:

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

Haskell in Industry?

https://wiki.haskell.org/Haskell_in_industry



Haskell 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.



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

Installing Haskell:

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

Haskell Documentation:

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

Haskell Basics:

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



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.

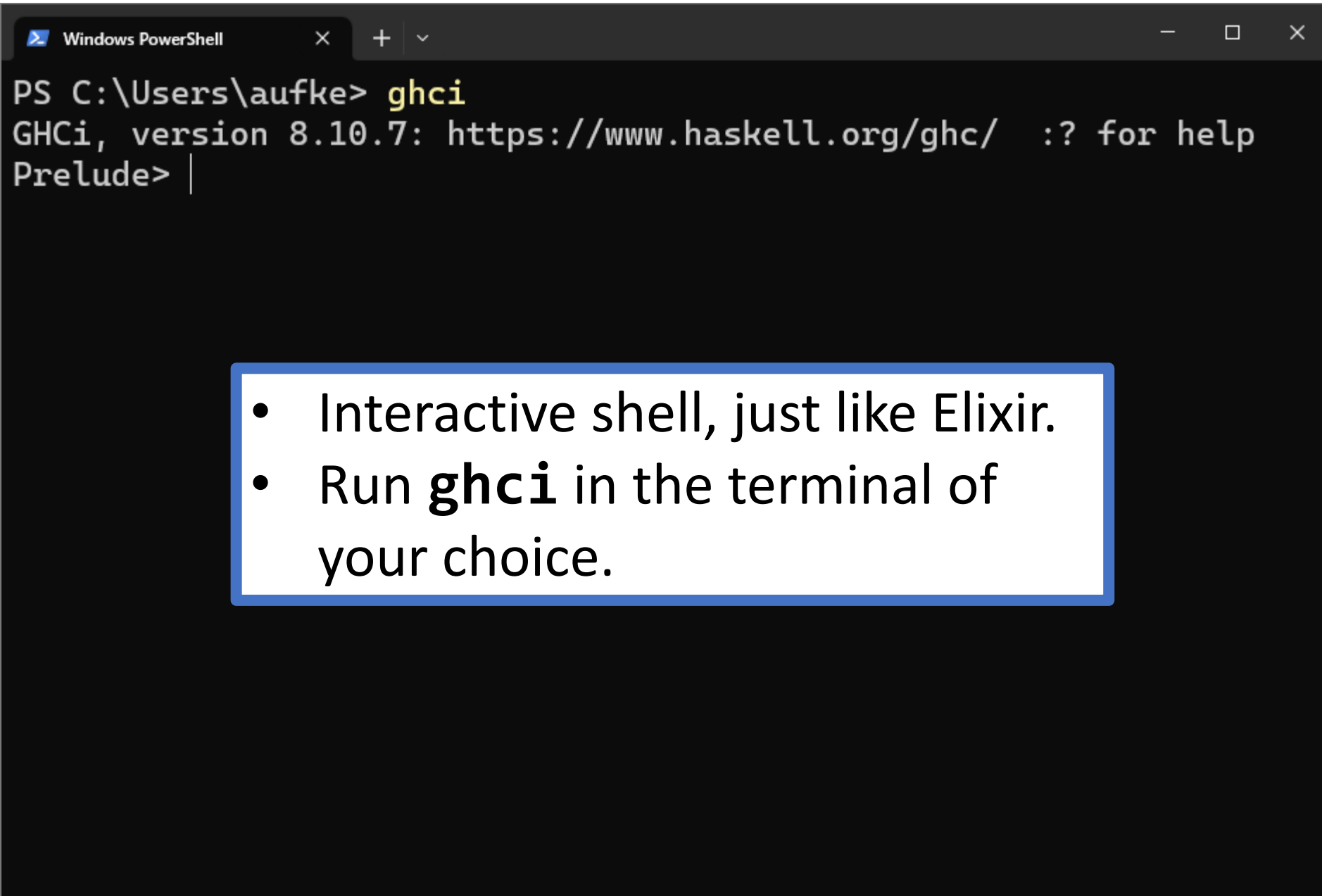
What you get

- 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

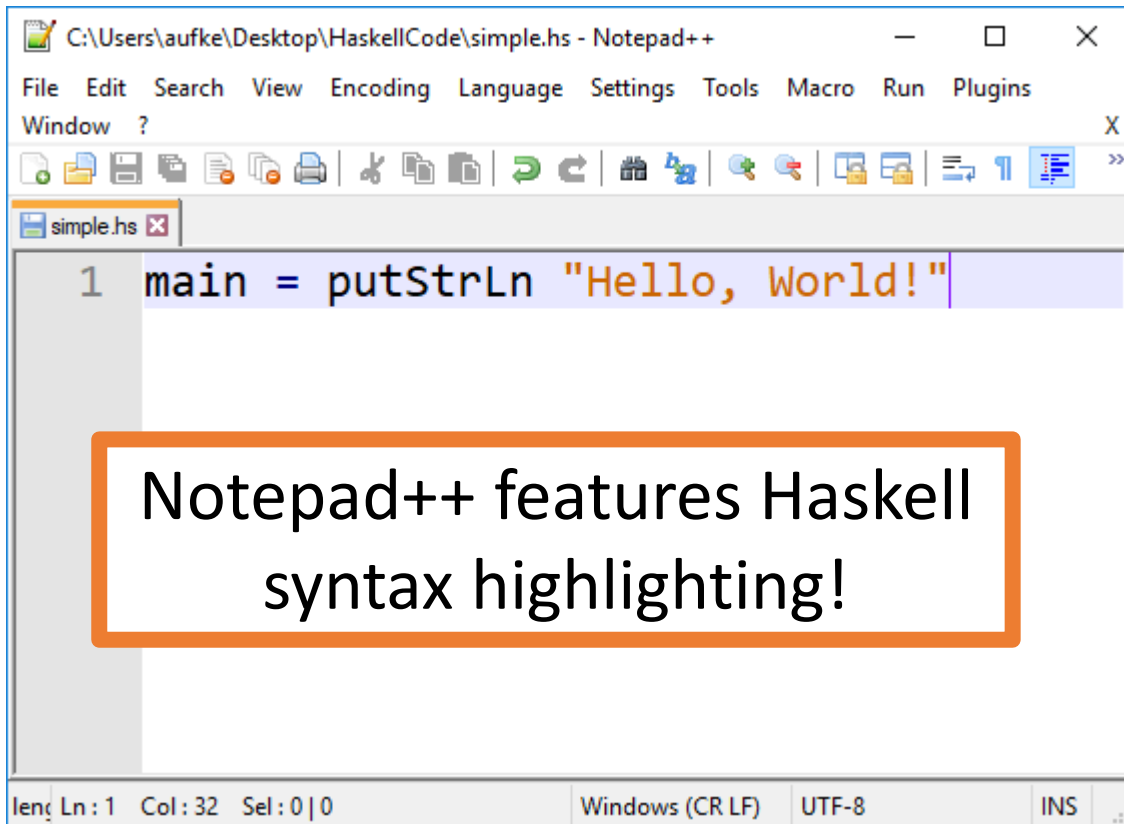


```
Windows PowerShell
PS C:\Users\aufke> ghci
GHCi, version 8.10.7: https://www.haskell.org/ghc/ :? for help
Prelude> |
```

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

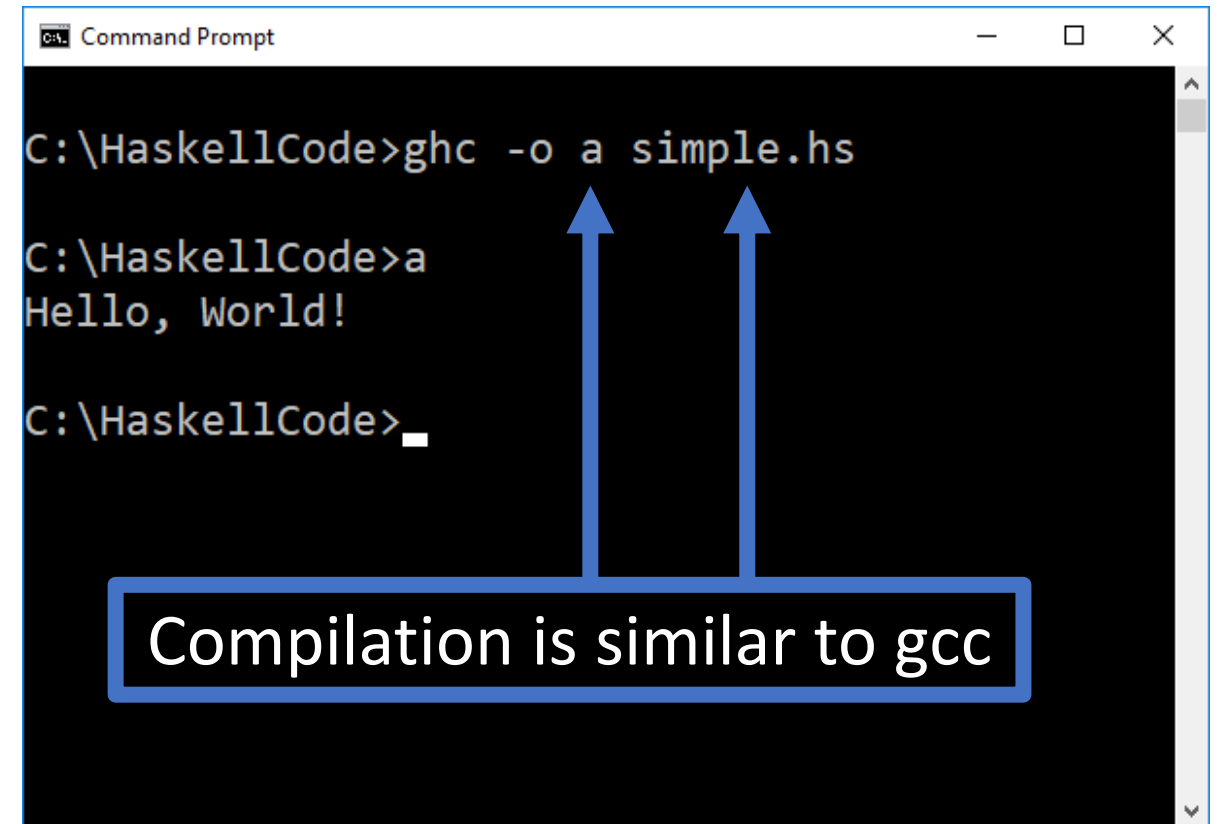
Hello, World!

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



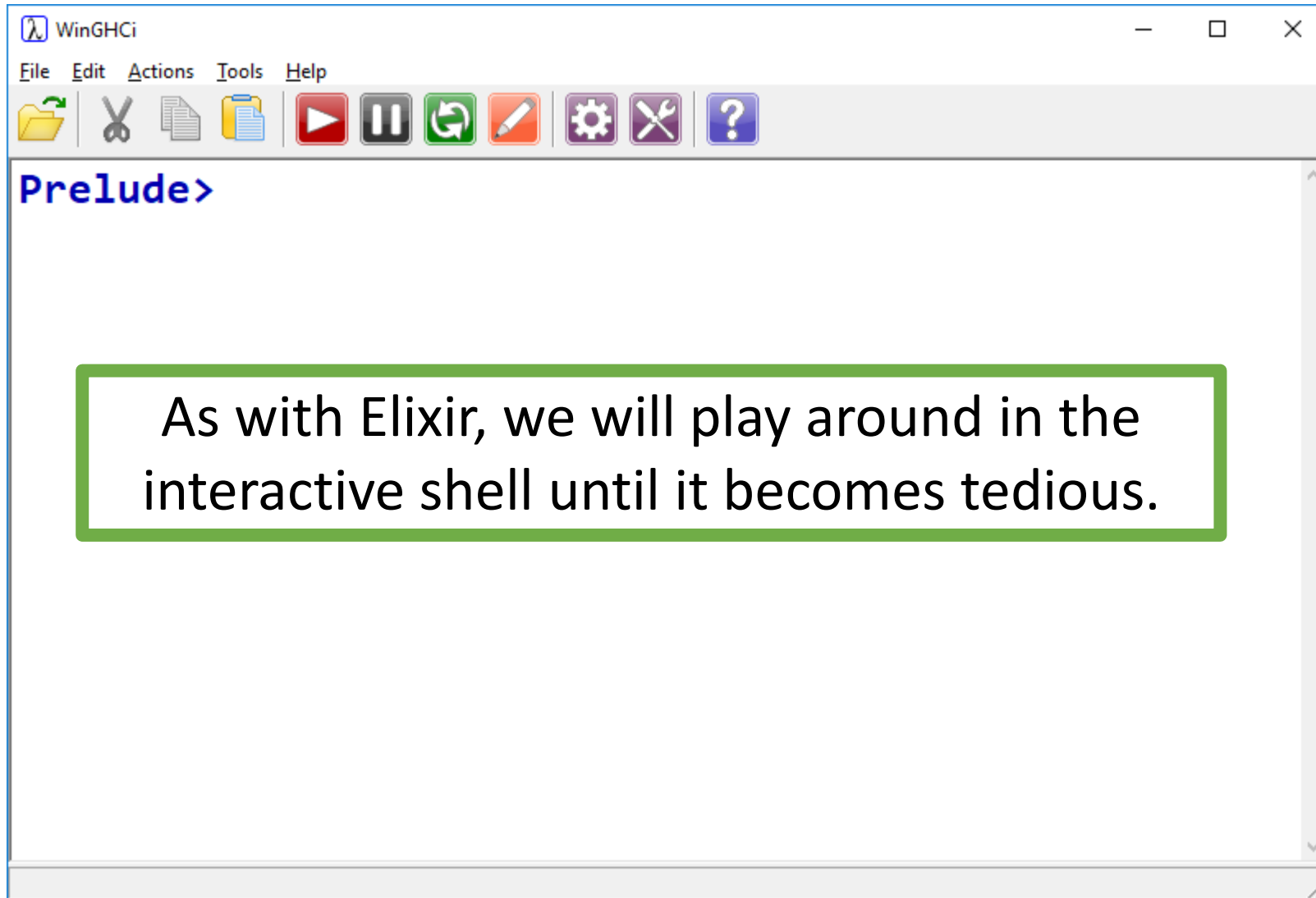
A screenshot of the Notepad++ text editor. The title bar shows the file path: `C:\Users\aufke\Desktop\HaskellCode\simple.hs - Notepad++`. The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, and Plugins. The toolbar contains various icons for file operations and editing. The editor window shows a single line of Haskell code: `1 main = putStrLn "Hello, World!"`. The text is color-coded: `main` is blue, `=` is black, `putStrLn` is blue, and the string `"Hello, World!"` is orange. An orange-bordered box is overlaid on the bottom half of the editor window, containing the text "Notepad++ features Haskell syntax highlighting!". The status bar at the bottom shows "Ln: 1 Col: 32 Sel: 0 | 0", "Windows (CR LF)", "UTF-8", and "INS".

Notepad++ features Haskell syntax highlighting!

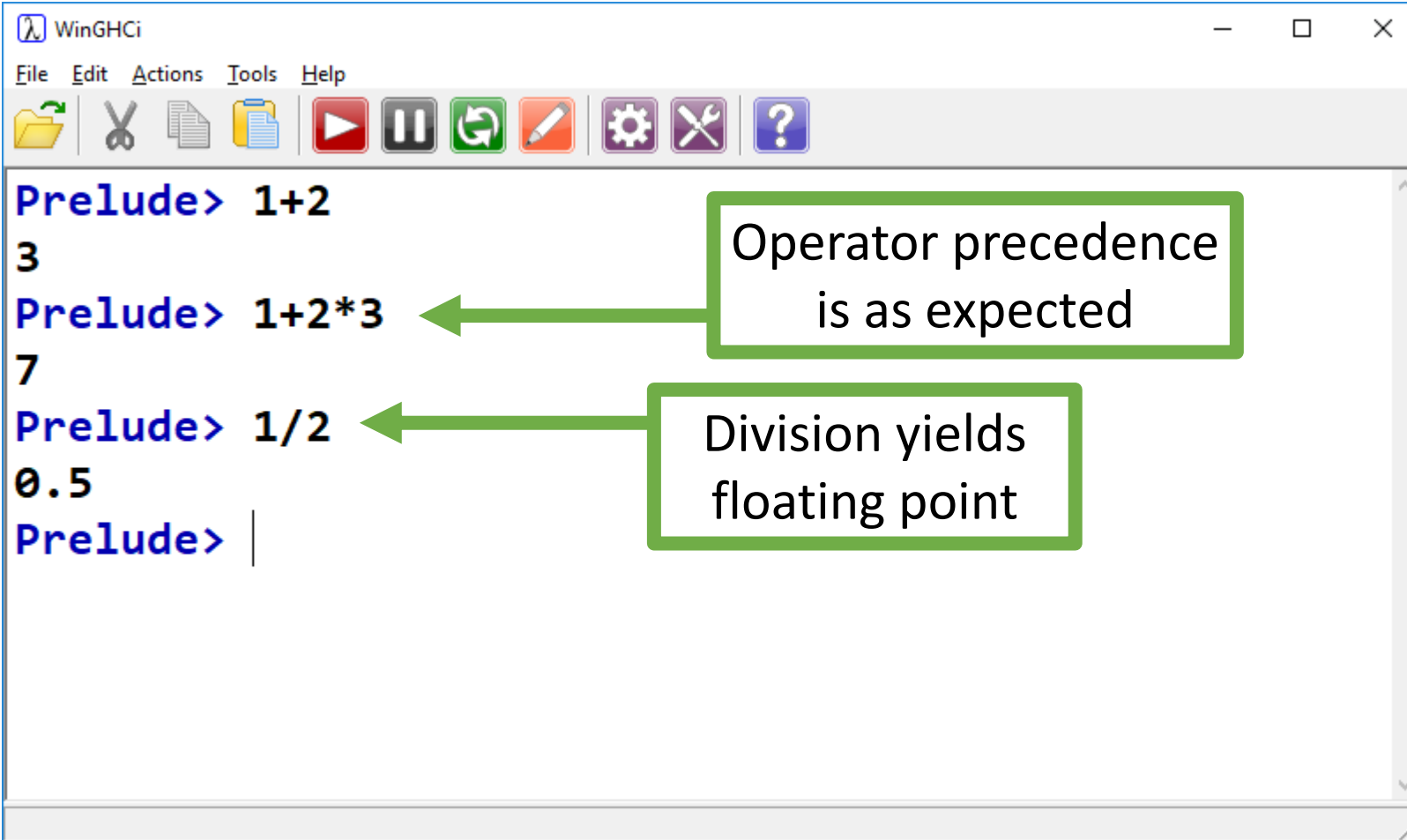


A screenshot of a Windows Command Prompt window. The title bar says "Command Prompt". The prompt is `C:\HaskellCode>`. The first command entered is `ghc -o a simple.hs`. The second command is `a`, which results in the output `Hello, World!`. The third prompt is `C:\HaskellCode>` followed by a cursor. Two blue arrows point from a blue-bordered box at the bottom to the `ghc` and `a` commands in the previous lines. The box contains the text "Compilation is similar to gcc".

Compilation is similar to gcc



Literals & Arithmetic



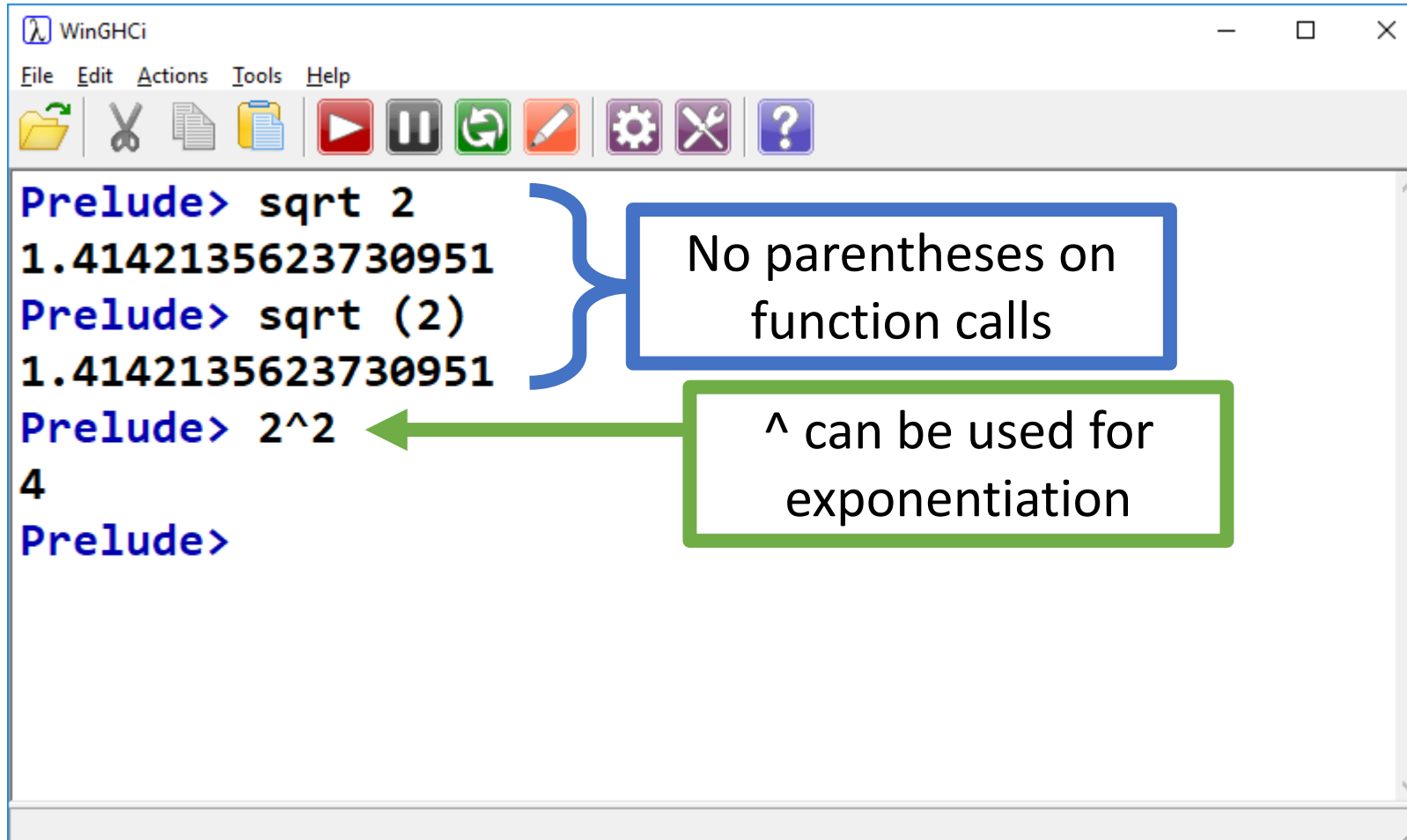
The screenshot shows the WinGHCi window with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar with icons for file operations, execution, and settings. The main text area displays the following interactions:

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

Two green callout boxes with arrows pointing to the input lines provide additional context:

- A box containing "Operator precedence is as expected" points to the expression `1+2*3`.
- A box containing "Division yields floating point" points to the expression `1/2`.

Literals & Arithmetic



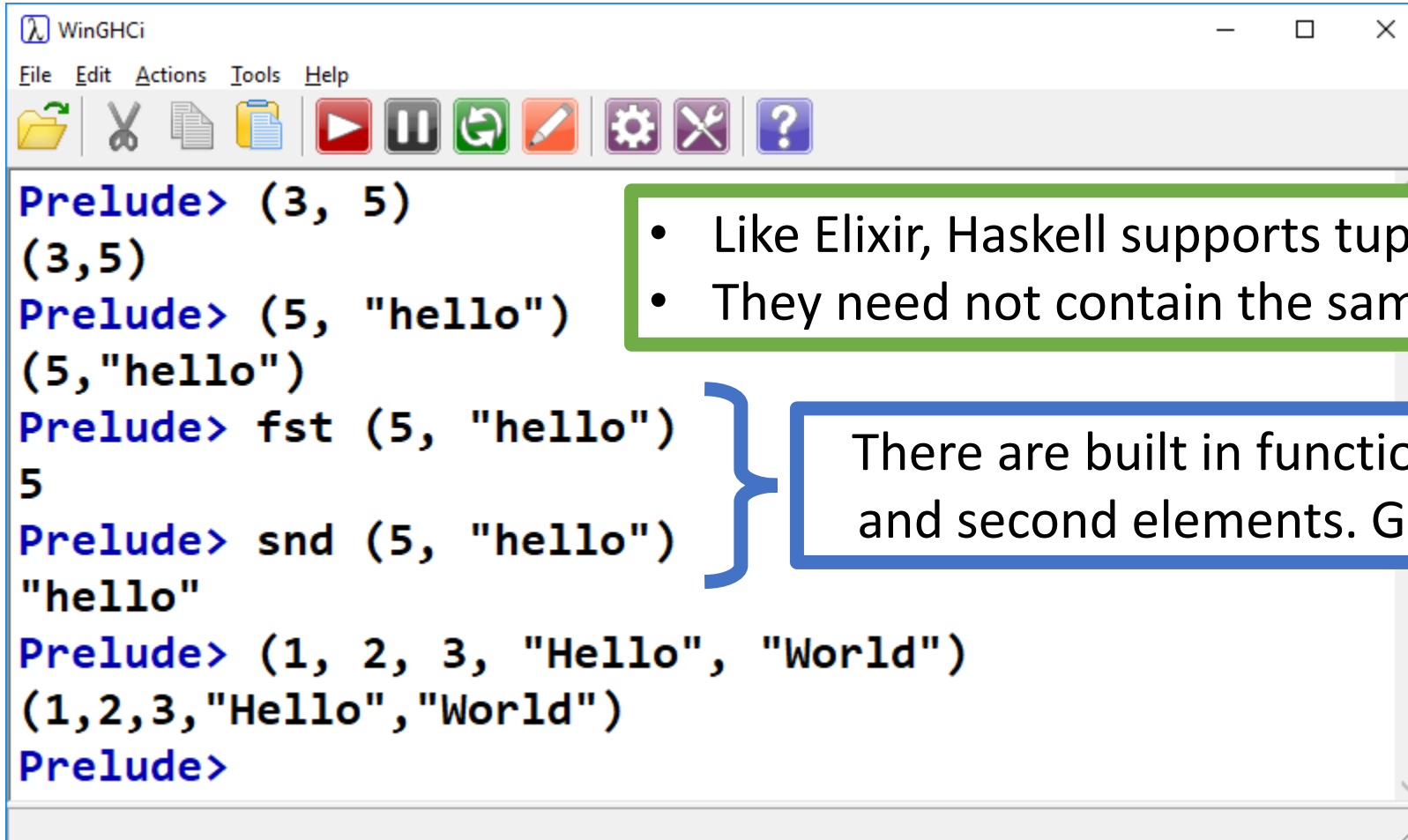
The screenshot shows the WinGHCi window with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar with icons for file operations, execution, and settings. The main text area contains the following interactions:

```
Prelude> sqrt 2
1.4142135623730951
Prelude> sqrt (2)
1.4142135623730951
Prelude> 2^2
4
Prelude>
```

Annotations on the image:

- A blue bracket groups the two `sqrt` function calls, with a callout box stating: "No parentheses on function calls".
- A green arrow points from the `2^2` expression to a callout box stating: "`^` can be used for exponentiation".

Tuples

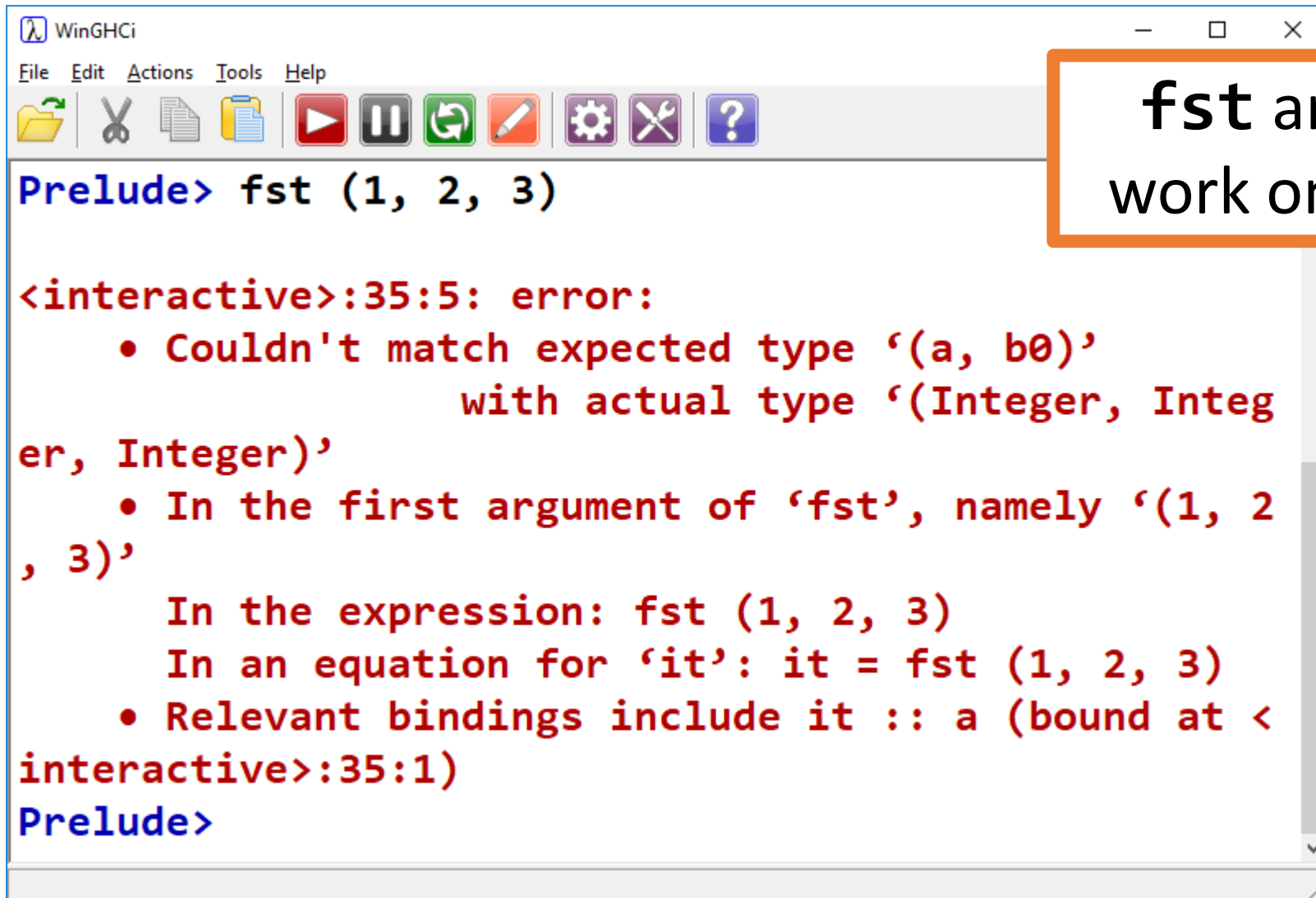


```
WinGHCi
File Edit Actions Tools Help
[Icons]

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



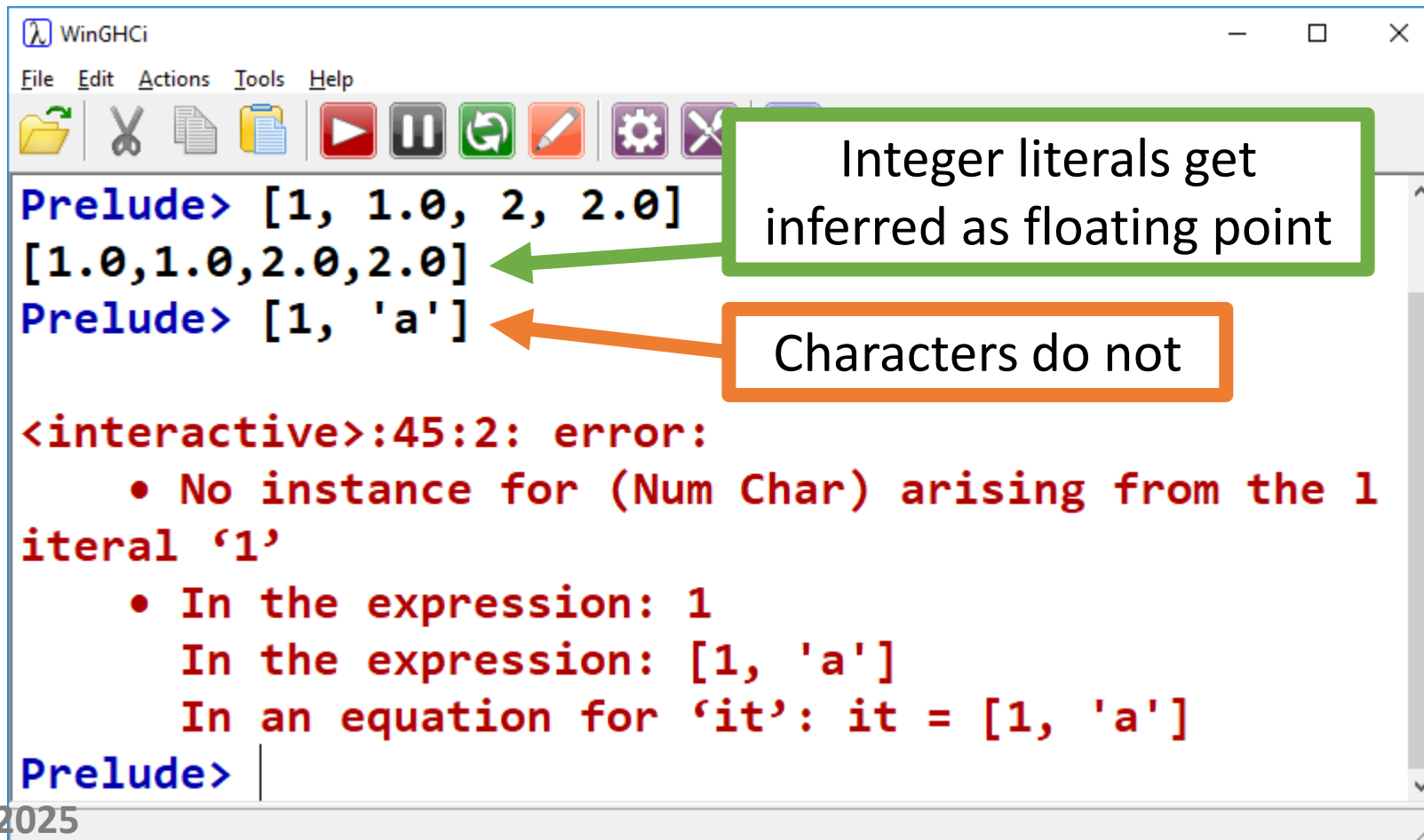
```
Prelude> fst (1, 2, 3)
```

```
<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 the WinGHCi window with the following content:

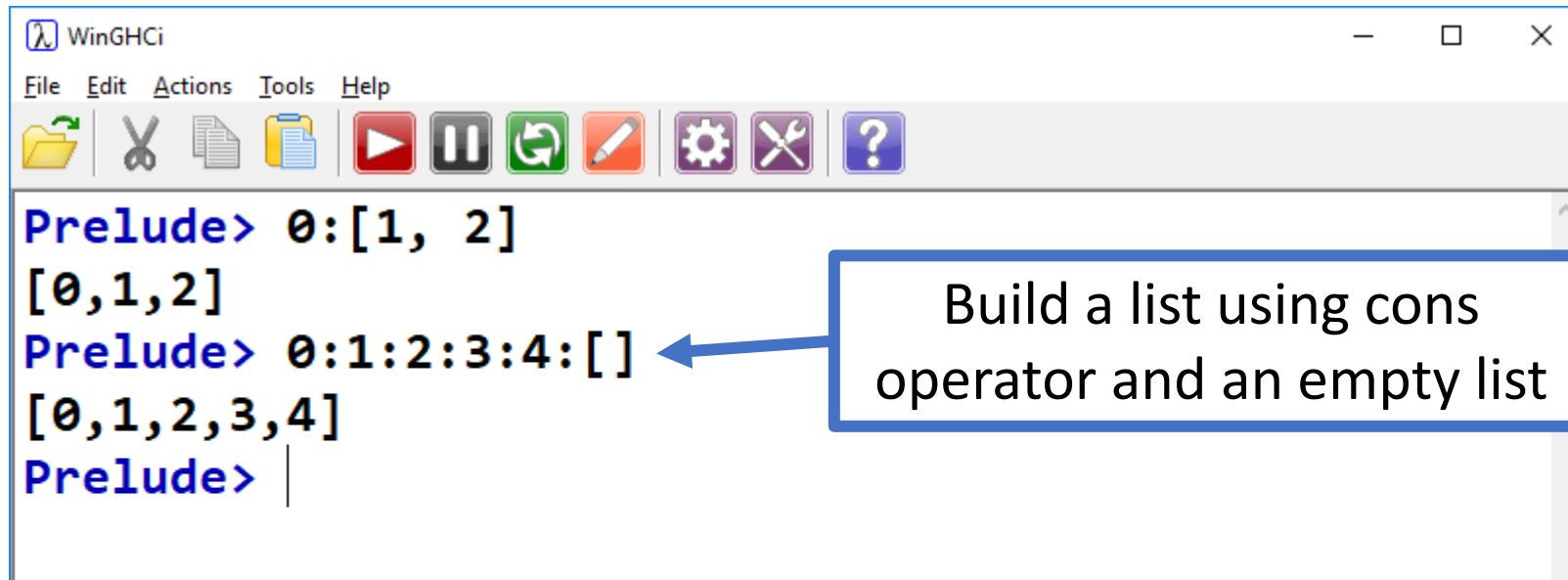
```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> [1, 1.0, 2, 2.0]
[1.0,1.0,2.0,2.0]
Prelude> [1, 'a']
<interactive>:45:2: error:
  • No instance for (Num Char) arising from the 1
    literal '1'
  • In the expression: 1
    In the expression: [1, 'a']
    In an equation for 'it': it = [1, 'a']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

Annotations in the image:

- A green box with the text "Integer literals get inferred as floating point" has a green arrow pointing to the second list `[1.0,1.0,2.0,2.0]`.
- An orange box with the text "Characters do not" has an orange arrow pointing to the list `[1, 'a']`.

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> |
```

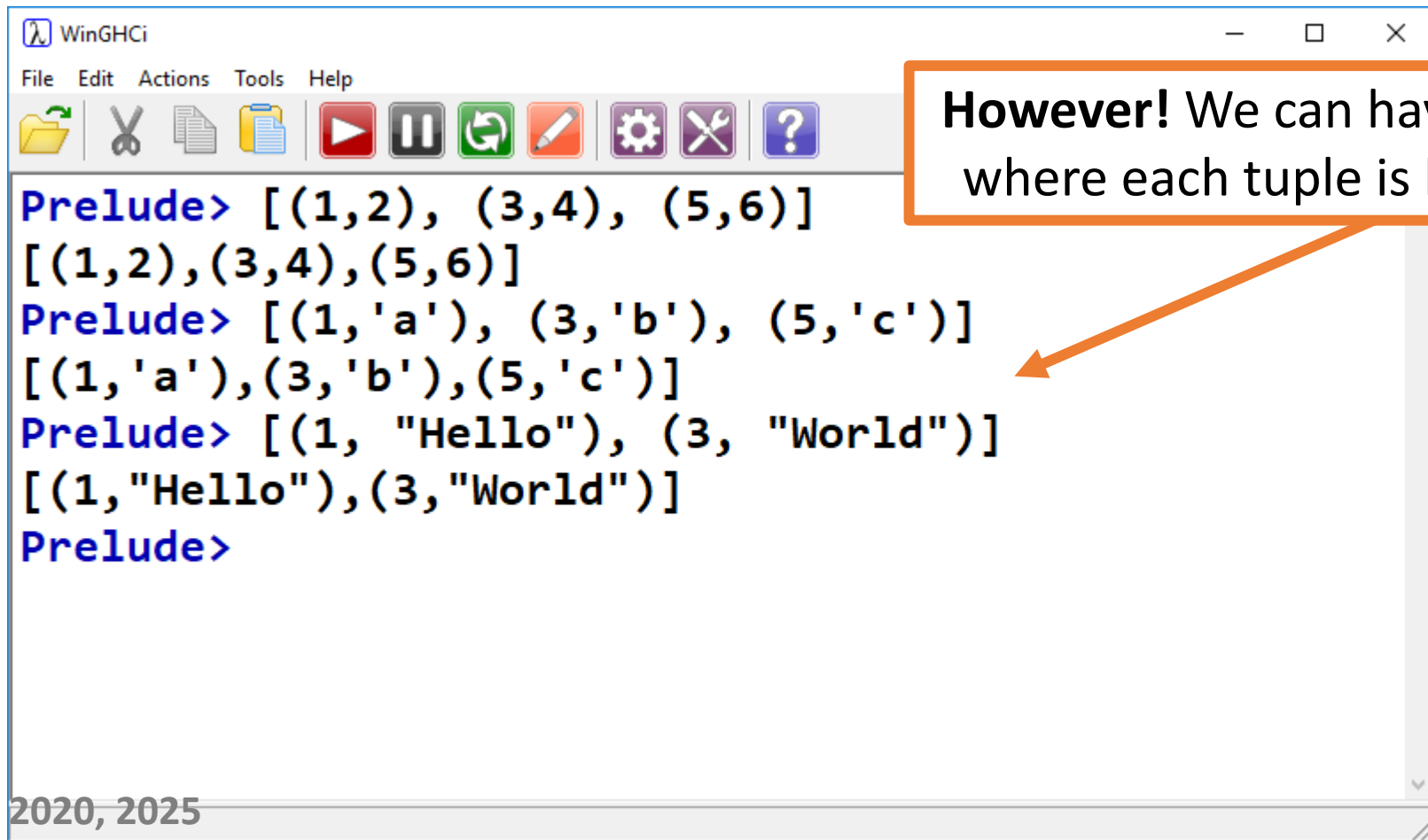
A blue arrow points from the text box on the right to the expression `0:1:2:3:4:[]` in the code block.

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.



A screenshot of the WinGHCi terminal window. The window has a title bar 'WinGHCi' and a menu bar 'File Edit Actions Tools Help'. Below the menu bar is a toolbar with icons for file operations (folder, copy, paste, save), execution (play, pause, refresh), and settings (gear, wrench, question mark). The terminal shows the following interactions:

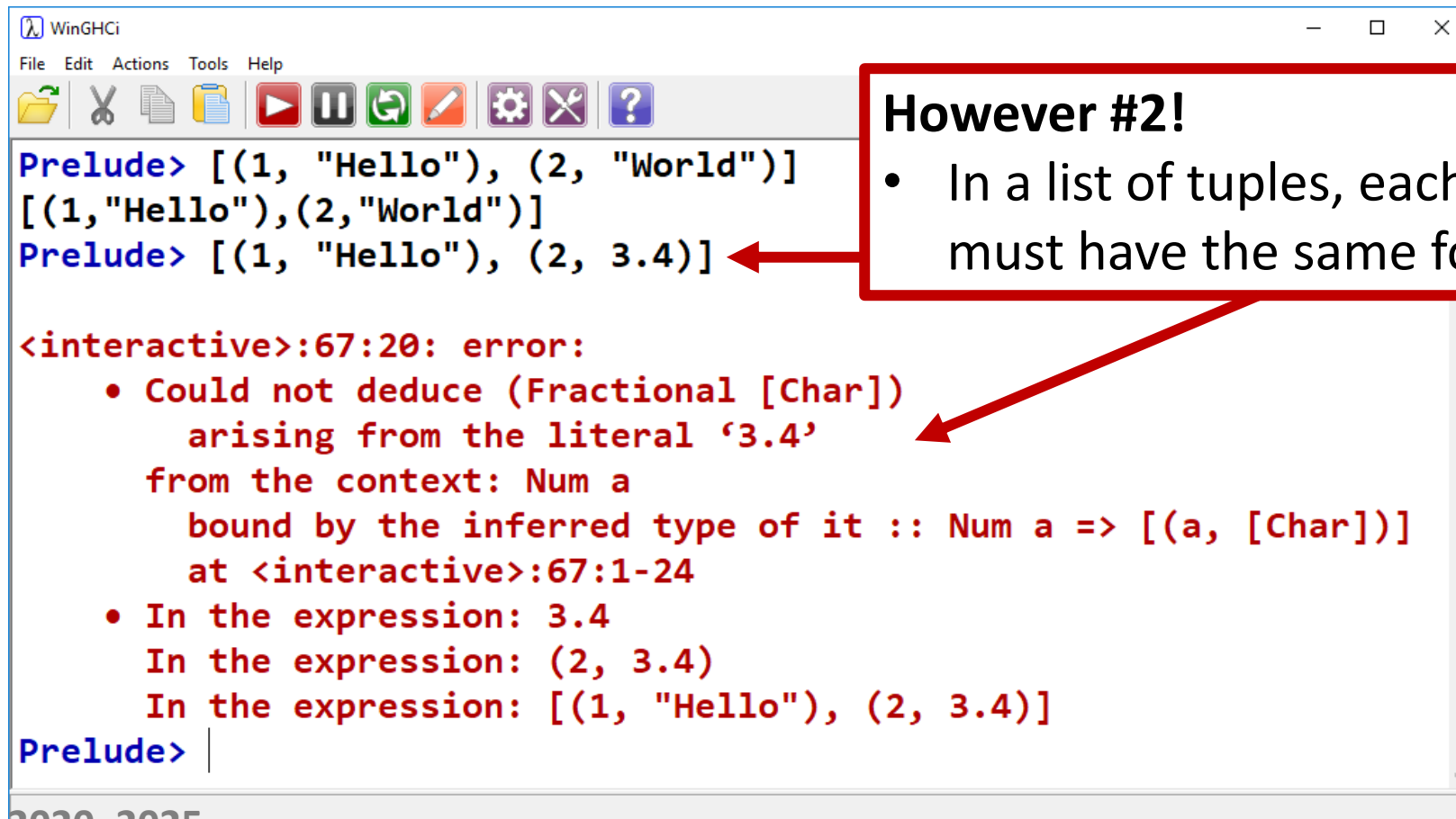
```
Prelude> [(1,2), (3,4), (5,6)]
[(1,2),(3,4),(5,6)]
Prelude> [(1,'a'), (3,'b'), (5,'c')]
[(1,'a'),(3,'b'),(5,'c')]
Prelude> [(1, "Hello"), (3, "World")]
[(1,"Hello"),(3,"World")]
Prelude>
```

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



Lists & Tuples

Tuples can be heterogeneous, lists must be homogeneous.



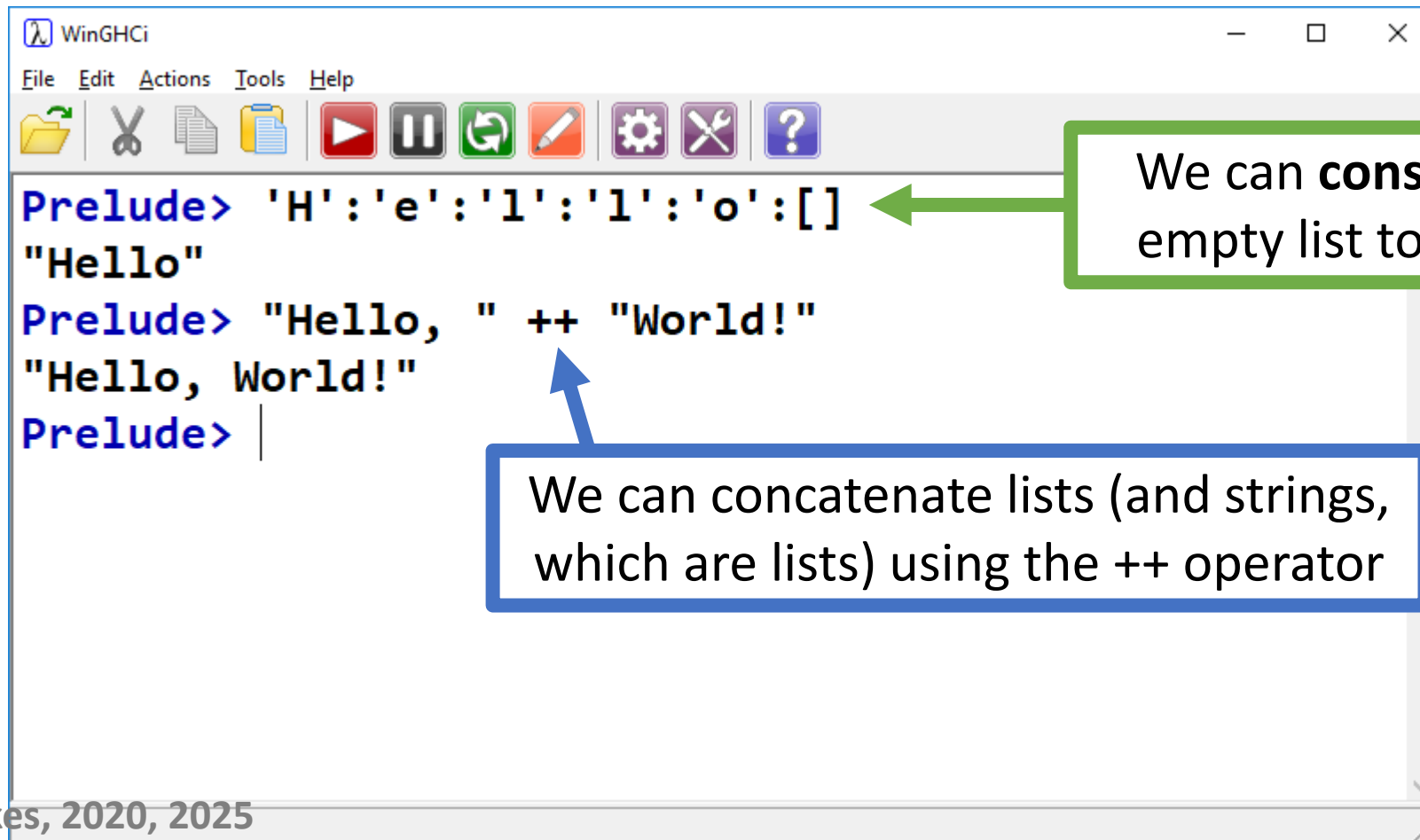
```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> [(1, "Hello"), (2, "World")]
[(1, "Hello"), (2, "World")]
Prelude> [(1, "Hello"), (2, 3.4)]
<interactive>:67:20: error:
• Could not deduce (Fractional [Char])
  arising from the literal '3.4'
  from the context: Num a
    bound by the inferred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



The screenshot shows the WinGHCi window with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar. The command line contains the following Haskell code:

```
Prelude> 'H':'e':'l':'l':'o':[]  
"Hello"  
Prelude> "Hello, " ++ "World!"  
"Hello, World!"  
Prelude> |
```

Two callout boxes provide explanations:

- A green box with a green arrow pointing to the first line of code: "We can **cons** chars into an empty list to form a string".
- A blue box with a blue arrow pointing to the second line of code: "We can concatenate lists (and strings, which are lists) using the ++ operator".

Strings

Concatenate multiple types? Java lets us...

```
StringTester - HelloWorld
Class Edit Tools
StringTester X
Compile Undo Cut

public class StringTester {
    public static void main(String[] args) {
        System.out.println("Hello, " ++ 5.0);
    }
}
```

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

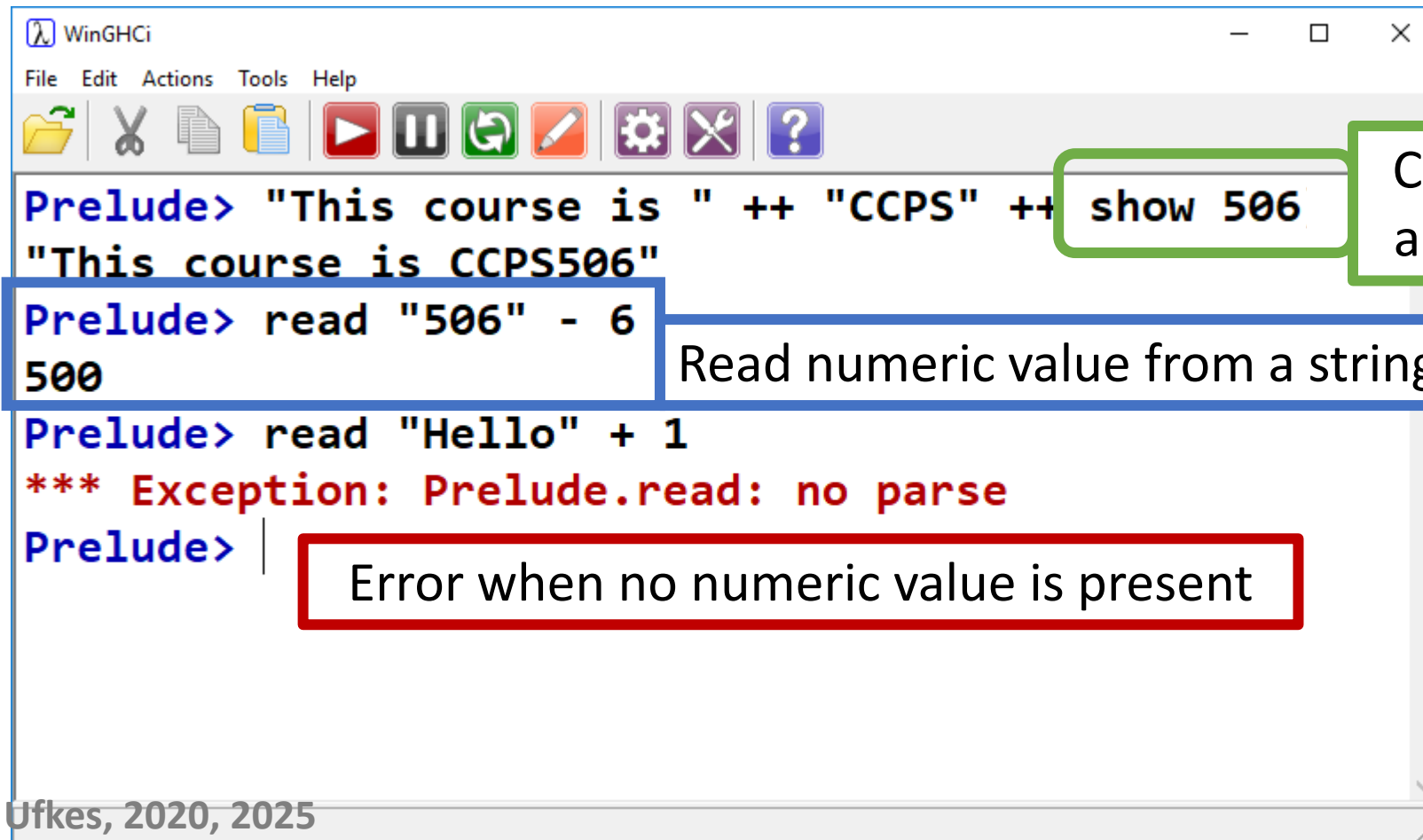
Prelude> "Hello, " ++ 5.0

<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>
```

put wh

Strings

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



A screenshot of the WinGHCi window. The window has 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> "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> |
```

Convert non-string argument to string

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

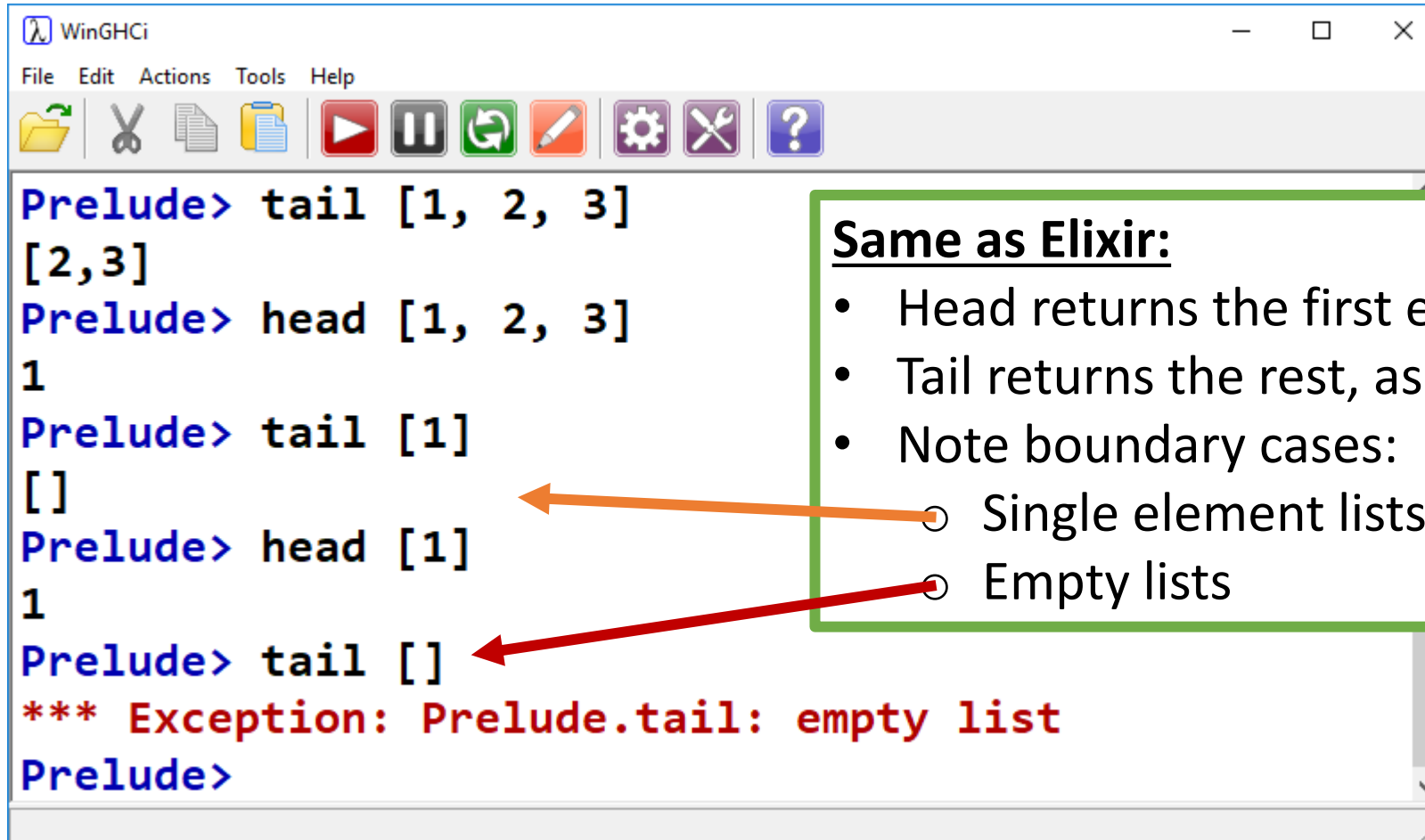
Error when no numeric value is present

Operations on Lists

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

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

Head & Tail

A screenshot of a WinGHCi window. The window has a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Below the menu is a toolbar with icons for file operations and execution. The main text area contains Erlang code. A green-bordered callout box on the right contains the text 'Same as Elixir:' followed by a bulleted list. Two arrows point from the callout box to the code: an orange arrow points from 'Single element lists' to the 'tail [1]' line, and a red arrow points from 'Empty lists' to the 'tail []' line.

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

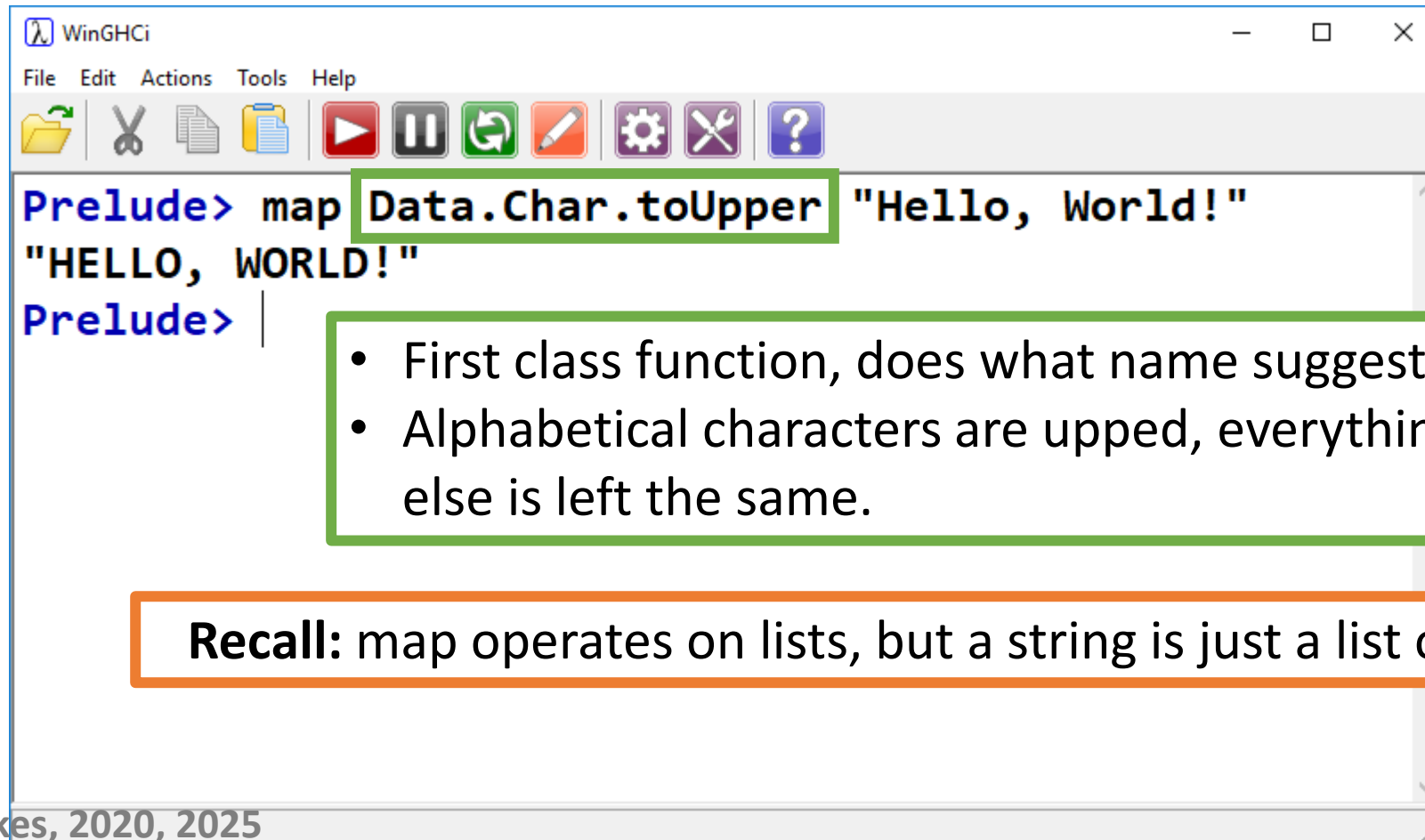
Prelude> tail [1, 2, 3]
[2,3]
Prelude> head [1, 2, 3]
1
Prelude> tail [1]
[]
Prelude> head [1]
1
Prelude> tail []
*** Exception: Prelude.tail: empty list
Prelude>
```

Same as Elixir:

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

map

Like Elixir's `Enum.map`

A screenshot of a WinGHCi terminal window. The window has a title bar with the WinGHCi logo and standard window controls. Below the title bar is a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Underneath the menu bar is a toolbar with icons for file operations (folder, copy, paste, save), execution (play, pause, refresh), editing (undo, redo, delete), and settings (gear, wrench, question mark). The terminal text shows a Haskell prompt 'Prelude>' followed by the command 'map Data.Char.toUpper "Hello, World!"'. The result '"HELLO, WORLD!"' is displayed on the next line. A green box highlights the 'Data.Char.toUpper' part of the command. Below the terminal output, there is a green-bordered box containing a bulleted list of two points: 'First class function, does what name suggests.' and 'Alphabetical characters are upped, everything else is left the same.' Below that, an orange-bordered box contains the text 'Recall: map operates on lists, but a string is just a list of characters'.

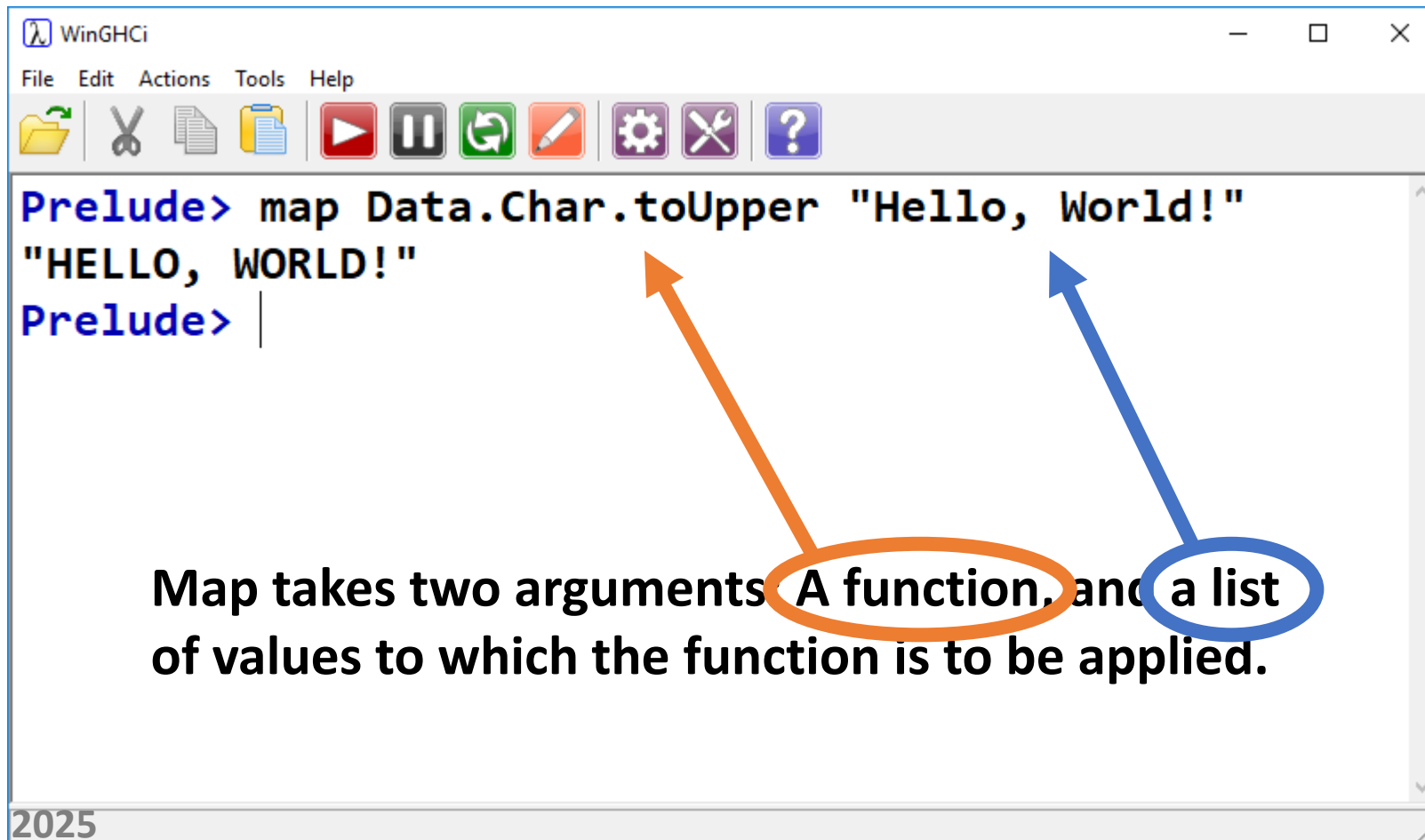
```
WinGHCi
File Edit Actions Tools Help
[Icons]
Prelude> map Data.Char.toUpper "Hello, World!"
"HELLO, WORLD!"
Prelude> |
```

- 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

Like Elixir's `Enum.map`



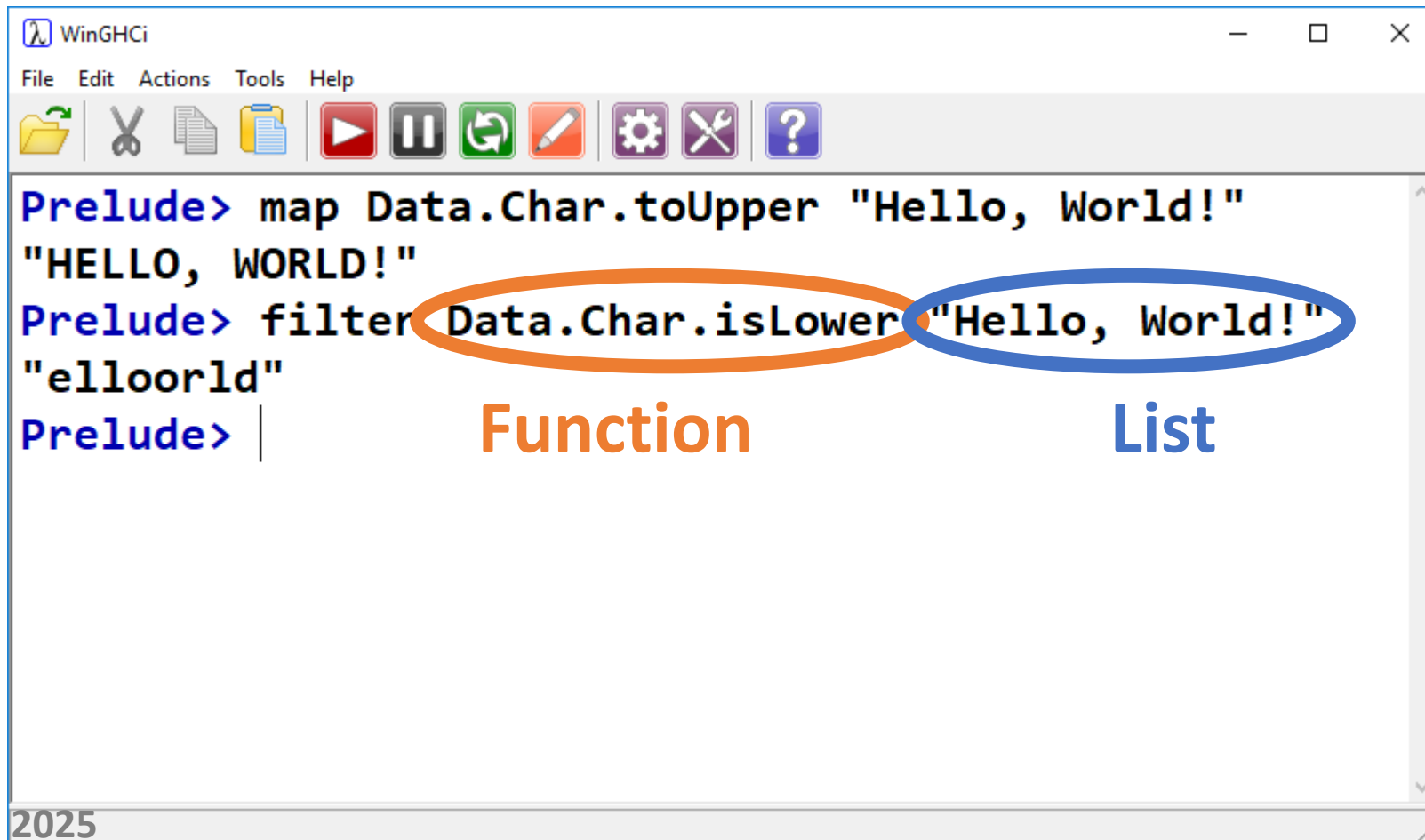
The image shows a WinGHCi terminal window with a menu bar (File, Edit, Actions, Tools, Help) and a toolbar with icons for file operations and execution. The terminal displays the following interaction:

```
Prelude> map Data.Char.toUpper "Hello, World!"  
"HELLO, WORLD!"  
Prelude> |
```

Below the terminal, a text overlay explains the function: "Map takes two arguments: A function, and a list of values to which the function is to be applied." The words "A function," are circled in orange, and "a list" is circled in blue. An orange arrow points from the orange circle to the function `Data.Char.toUpper` in the terminal command. A blue arrow points from the blue circle to the string `"Hello, World!"` in the terminal command.

filter

“Remove” items from a list based on some criteria:

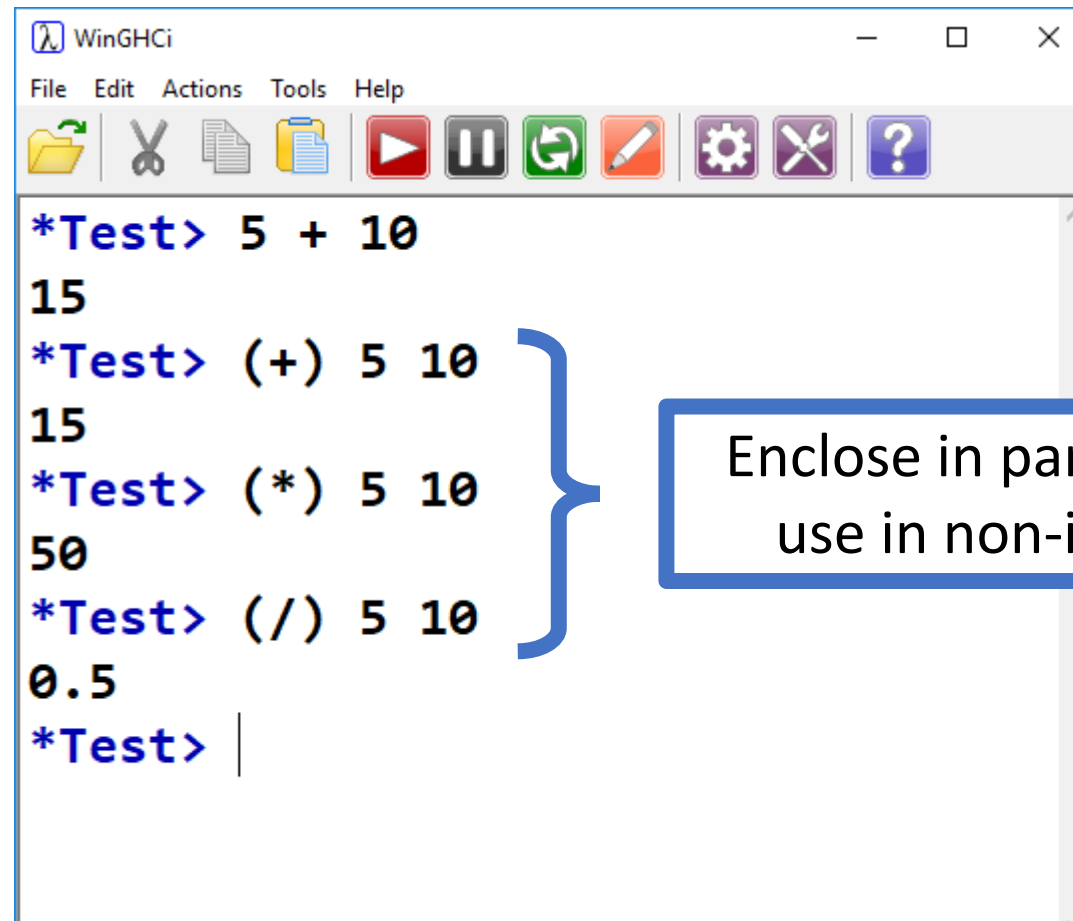
A screenshot of a WinGHCi window. The window has a title bar with the text 'WinGHCi' and standard window controls. Below the title bar is a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Under the menu bar is a toolbar with icons for file operations (folder, scissors, document), execution (play, pause, refresh), editing (pencil), settings (gear, wrench), and help (question mark). The main text area contains three lines of Haskell code. The first line is 'Prelude> map Data.Char.toUpper "Hello, World!"' followed by the output '"HELLO, WORLD!"'. The second line is 'Prelude> filter Data.Char.isLower "Hello, World!"' followed by the output '"elloorld"'. The third line is 'Prelude> |'. The text 'Data.Char.isLower' is circled in orange, and the text '"Hello, World!"' is circled in blue. Below the code, the word 'Function' is written in orange and 'List' is written in blue, corresponding to the circled parts of the code.

```
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> |
```

Function List

Infix Functions

Use symbolic operators as functions:

A screenshot of the WinGHCi window. The window has a menu bar with '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, wrench), and help (question mark). The main text area shows the following interactions:

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

A blue bracket groups the three lines where operators are used in parentheses: `(+)`, `(*)`, and `(/)`.

```


```

Enclose in parentheses to
use in non-infix mode

foldl, foldr

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

Recall that the list:

[1, 2, 3, 4, 5]

Is actually seen as:

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

By the compiler.

foldl, foldr

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

Recall that the list:

[1, 2, 3, 4, 5]

Is actually seen as:

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

By the compiler.


- **foldr** in effect replaces the cons operator with another function of our choosing.
- This is similar to **Enum.reduce** in Elixir.
- The empty list is replaced with some initial value.

foldl, foldr

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

- **foldr** in effect replaces the cons operator with another function of our choosing.
- This is similar to **Enum.reduce** in Elixir.
- The empty list is replaced with some initial value.

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



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

foldl, foldr

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

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



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

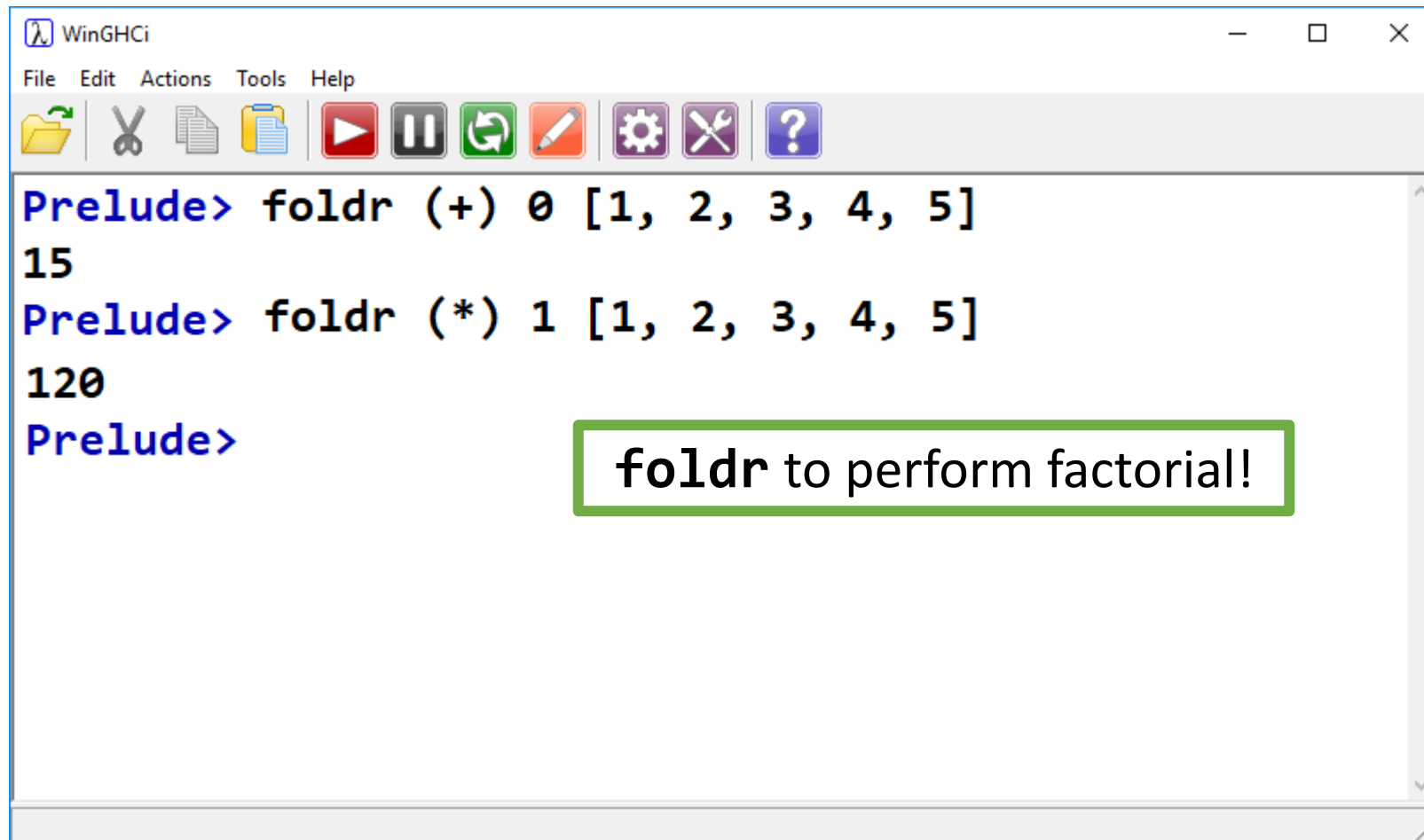


`1 + 2 + 3 + 4 + 5 + 0`



`15`

foldl, foldr

A screenshot of a WinGHCi terminal window. The window has a title bar with the WinGHCi logo and standard window controls. Below the title bar is a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Under the 'Actions' menu, there is a toolbar with icons for file operations (copy, paste, save), execution (run, pause, refresh), and settings (gear, wrench, help). The terminal area shows the following text:

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

A green rectangular box is drawn over the text 'foldr to perform factorial!'.

foldr to perform factorial!

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`

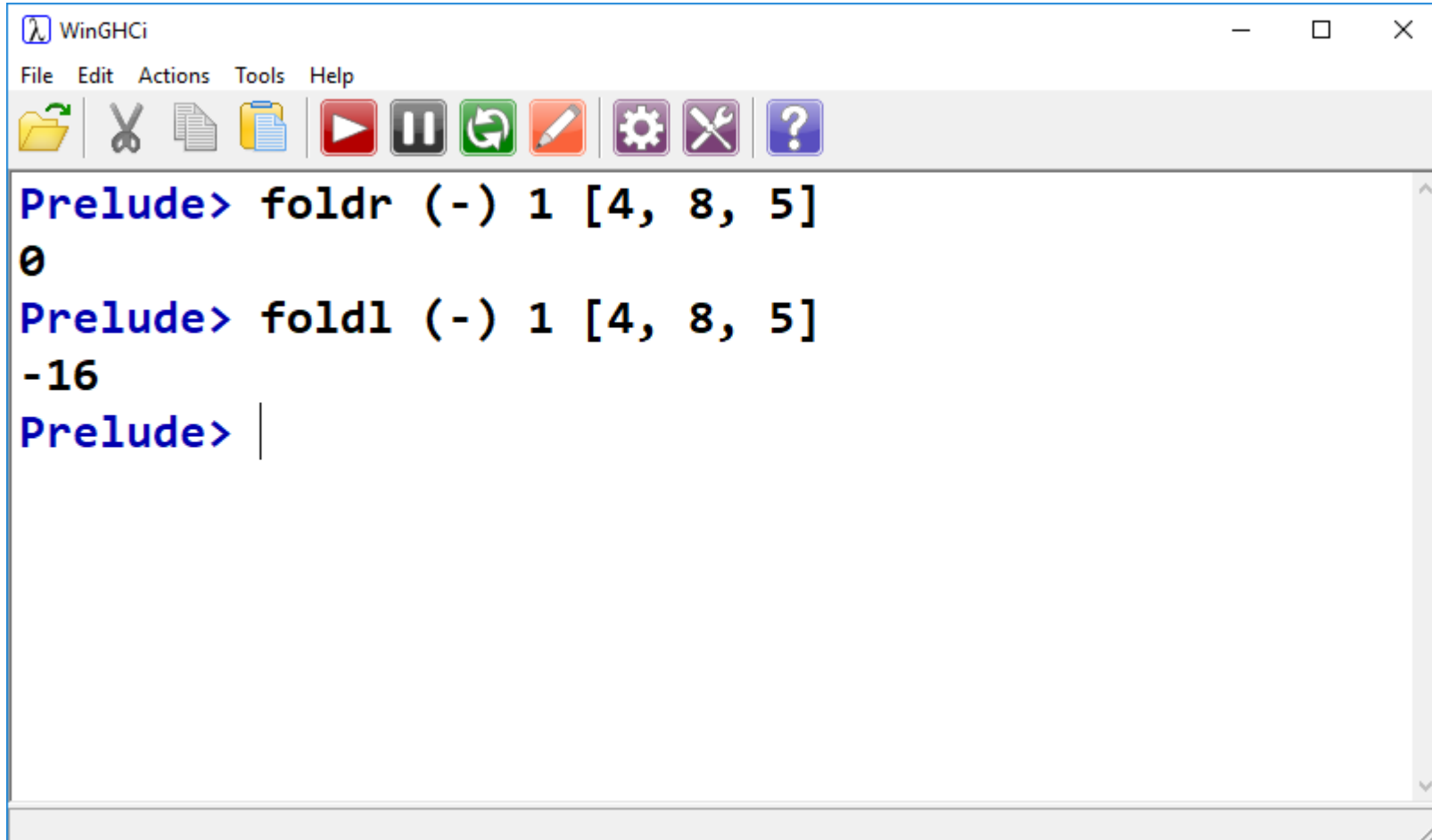
Is actually:

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



`-16`

foldl VS foldr

A screenshot of the WinGHCi Haskell interpreter window. The window has a title bar 'WinGHCi' and standard window controls. Below the title bar is a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Under the 'Actions' menu, there is a toolbar with icons for file operations (folder, copy, paste), execution (play, pause, refresh), editing (undo, redo), and settings (gear, wrench, question mark). The main text area shows the following interactions:

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

The first command, `foldr (-) 1 [4, 8, 5]`, returns `0`. The second command, `foldl (-) 1 [4, 8, 5]`, returns `-16`. The third command is a prompt with a cursor.

List Generation

Syntactic sugar:

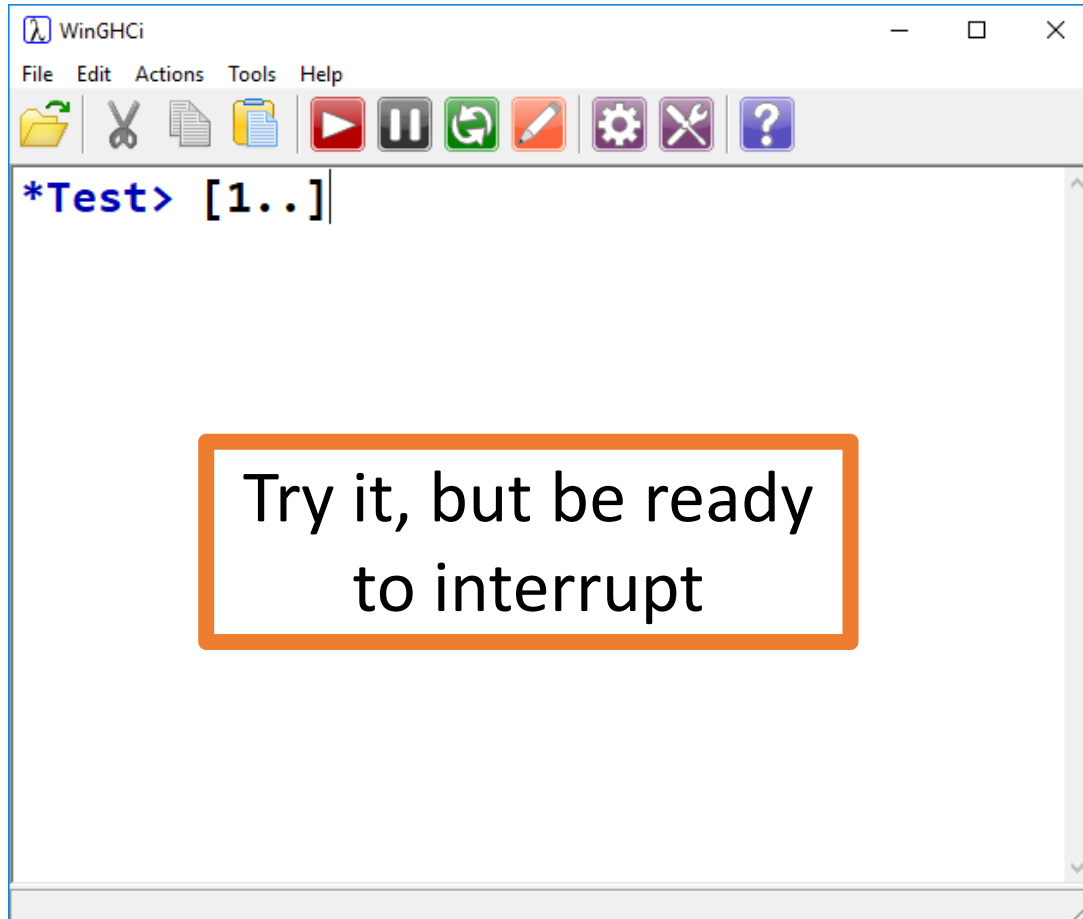
List declaration: `list = [1, 2, 3, 4, 5, 6, 7, 8, 9]`

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

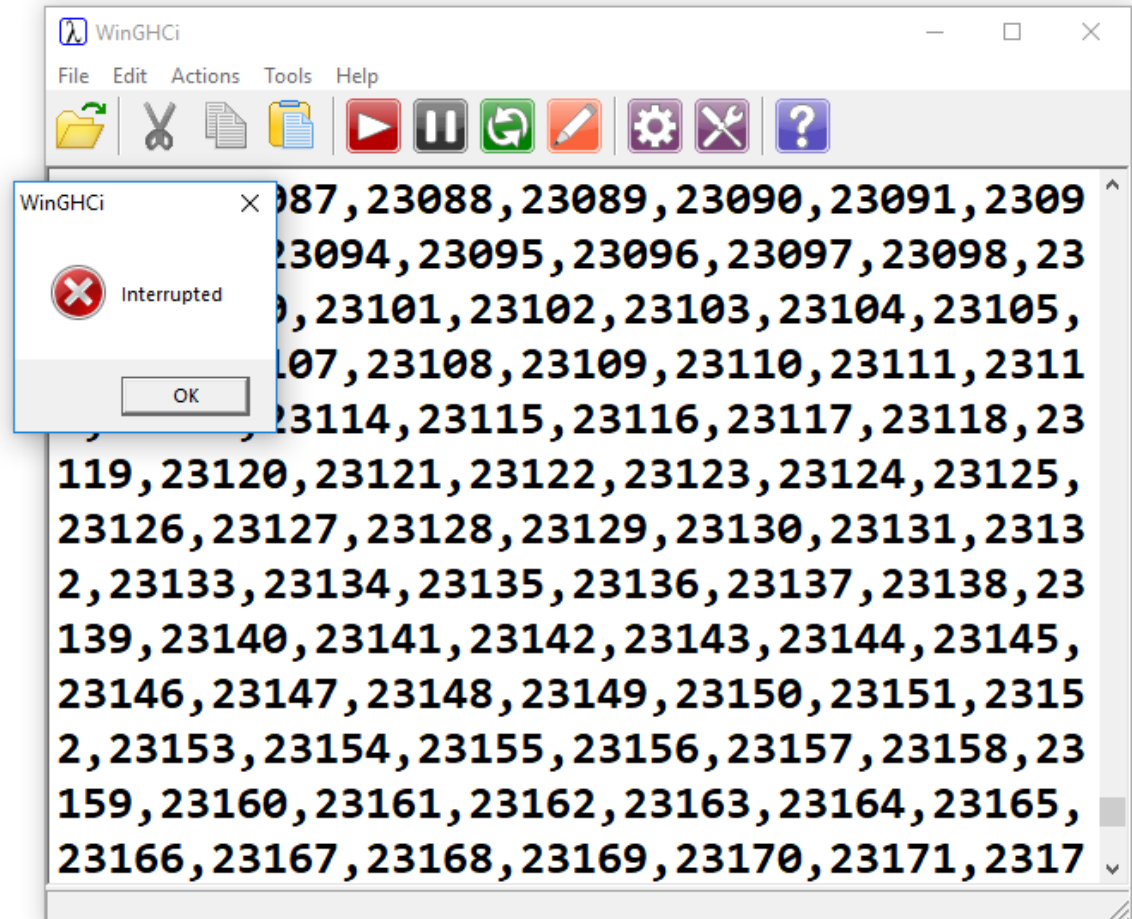
Specify interval: `list = [1,3..9]`
`= [1,3,5,7,9]`

Interval is discerned from difference between first two elements

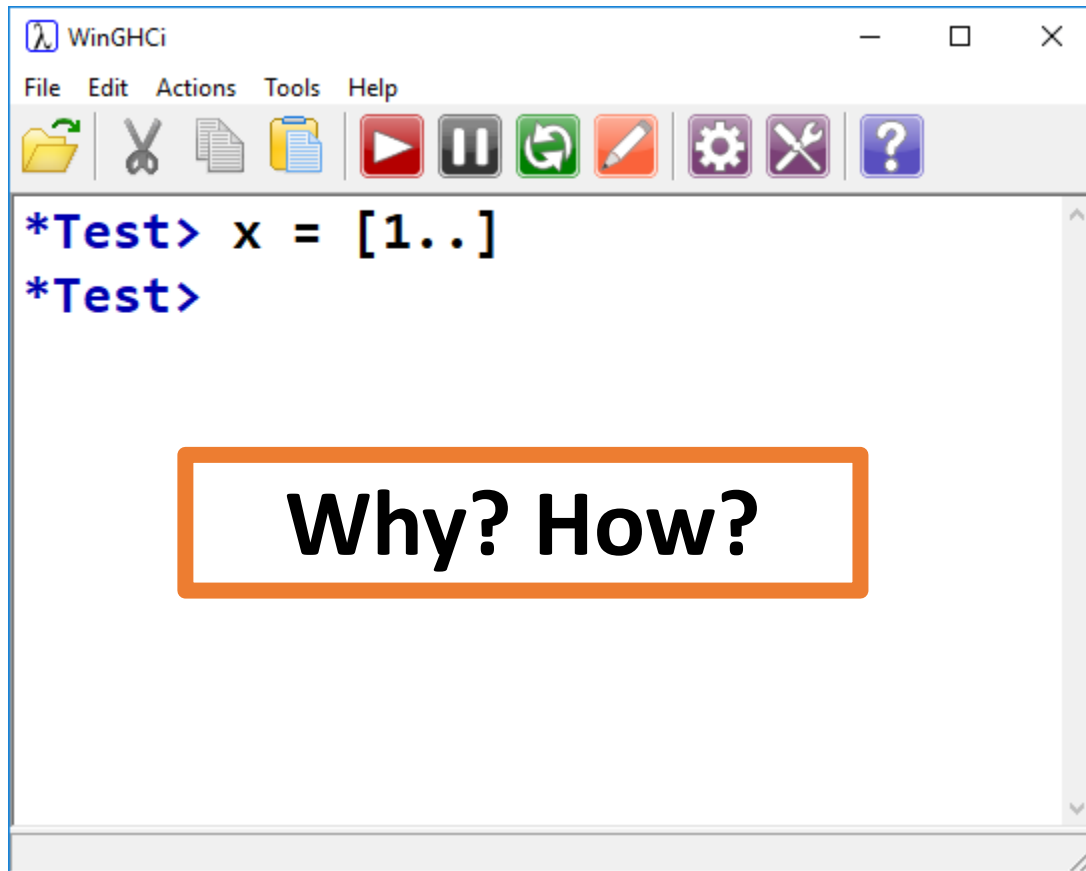
Infinite Lists?



Try it, but be ready
to interrupt



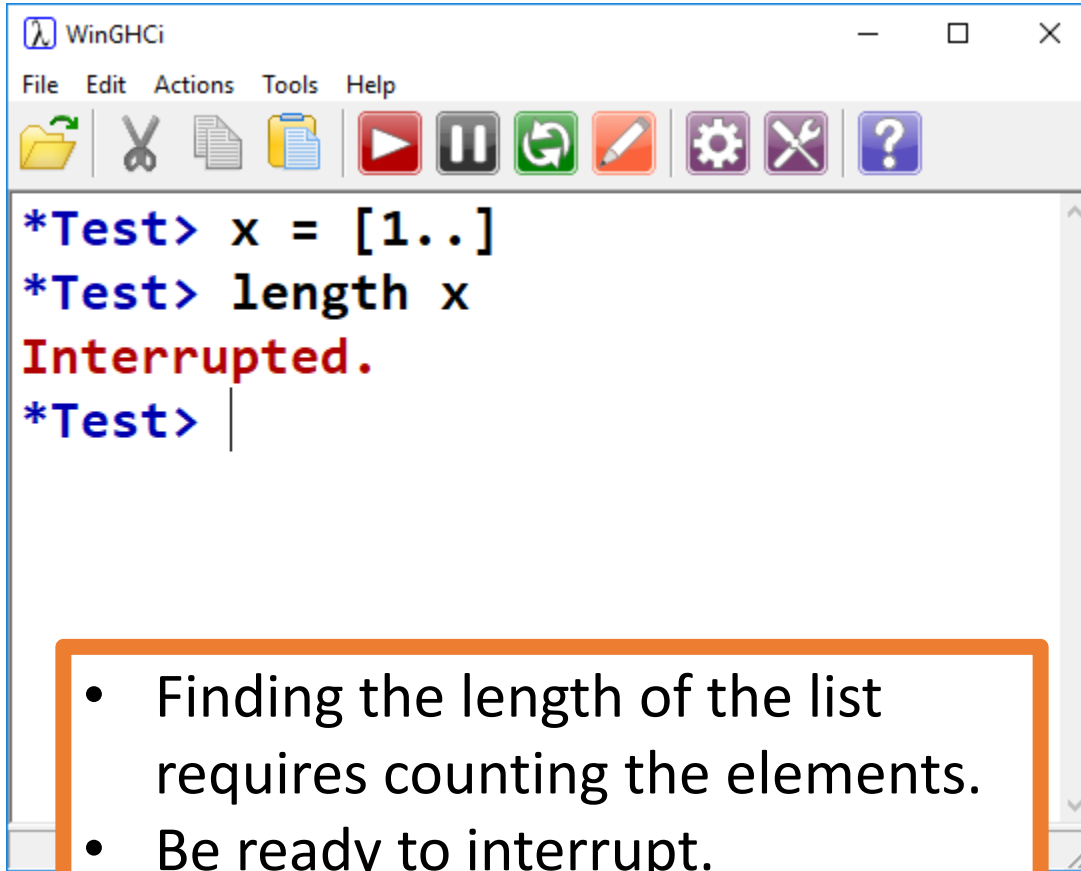
Infinite Lists?



Haskell is lazy!

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

Infinite Lists?

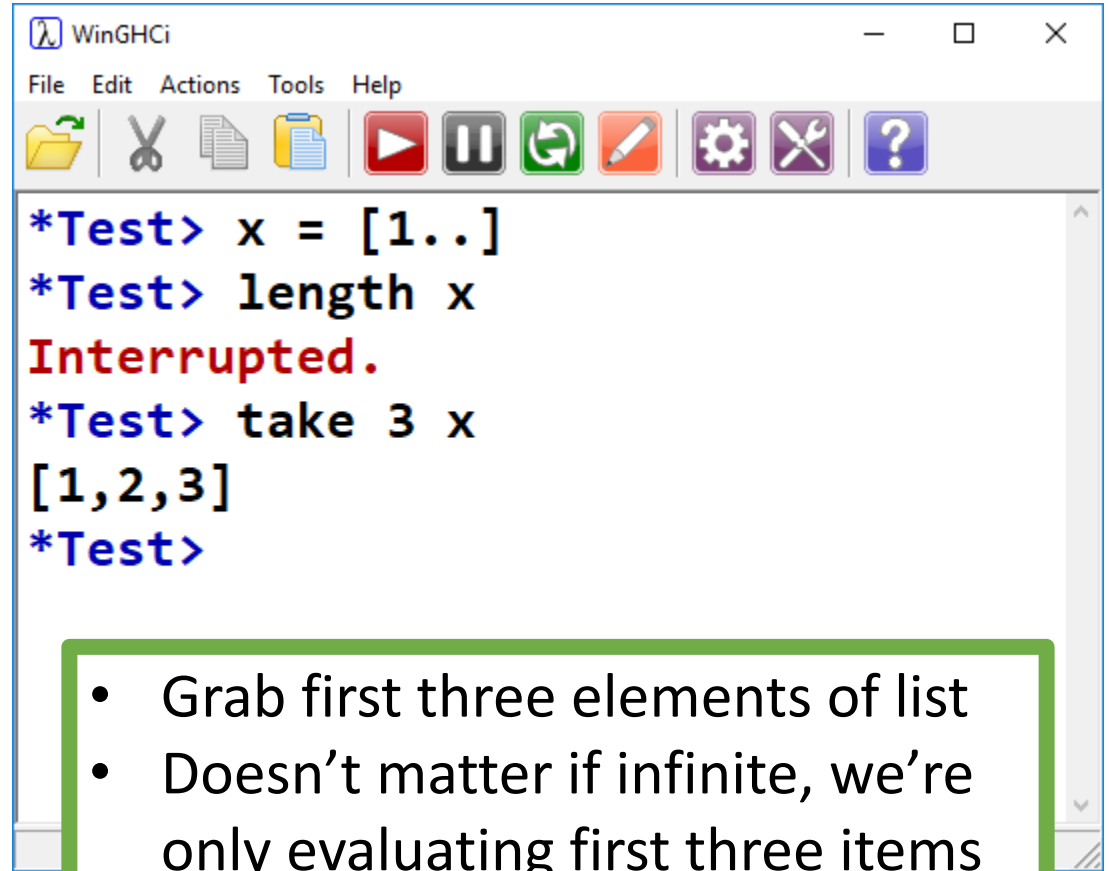


```
WinGHCi
File Edit Actions Tools Help
[Icons]

*Test> x = [1..]
*Test> length x
Interrupted.
*Test> |
```

The terminal window shows the creation of an infinite list `x = [1..]`. When the command `length x` is entered, it results in an `Interrupted.` error. The prompt `*Test>` is followed by a vertical bar, indicating the user is waiting for input.

- 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>
```

The terminal window shows the same infinite list `x = [1..]`. After an interrupted `length x` command, the user enters `take 3 x`, which successfully returns the list `[1,2,3]`. The prompt `*Test>` is followed by a vertical bar, indicating the user is waiting for input.

- 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.

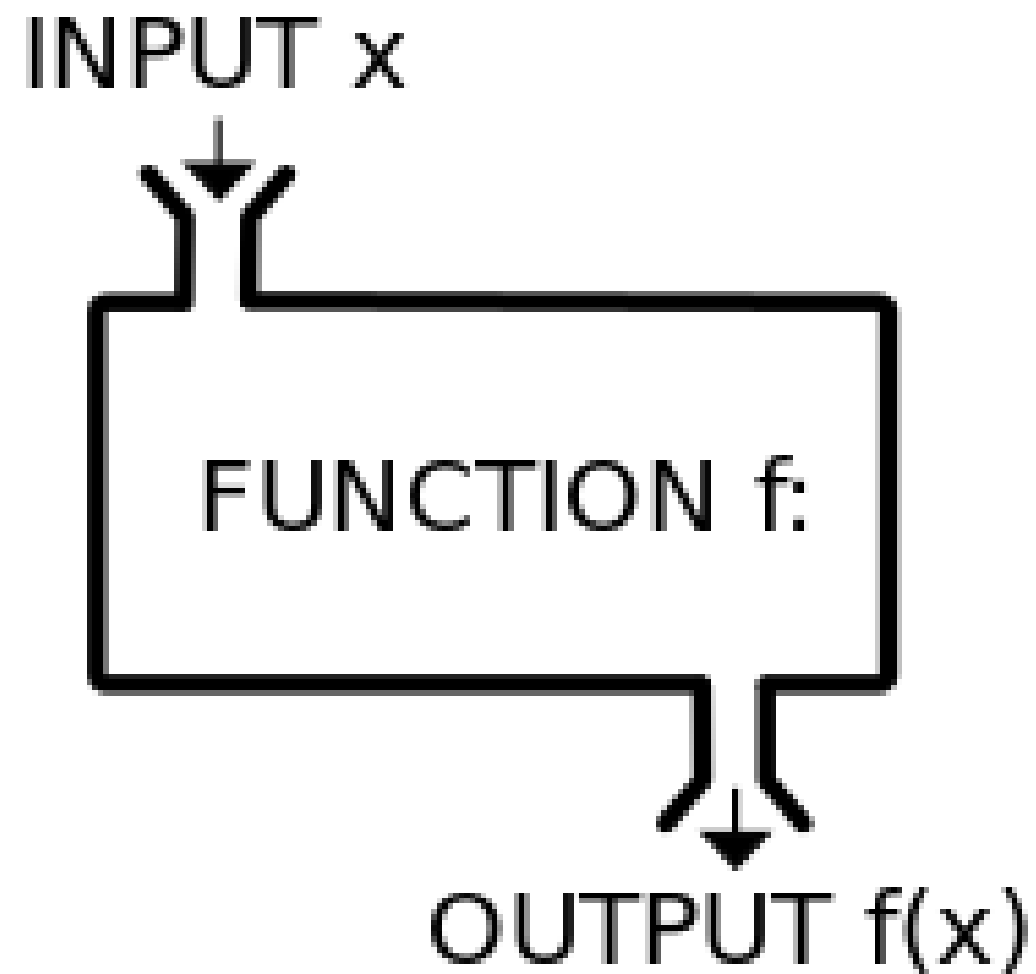
```
WinGHCi
File Edit Actions Tools Help
[Icons]

*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]

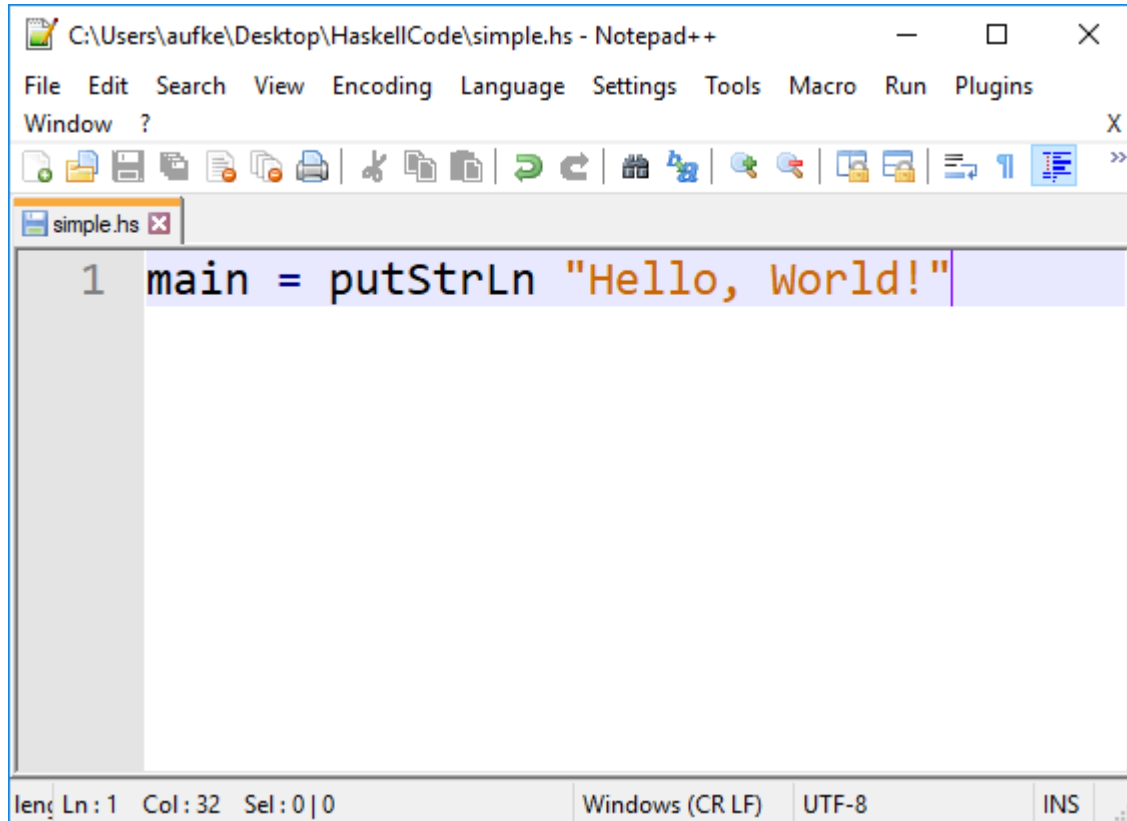
*Test> x = [1..]
*Test> zip "Hello" x
[('H',1),('e',2),('l',3),('l',4),('o',5)]
*Test>
```

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



Functions in Haskell

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

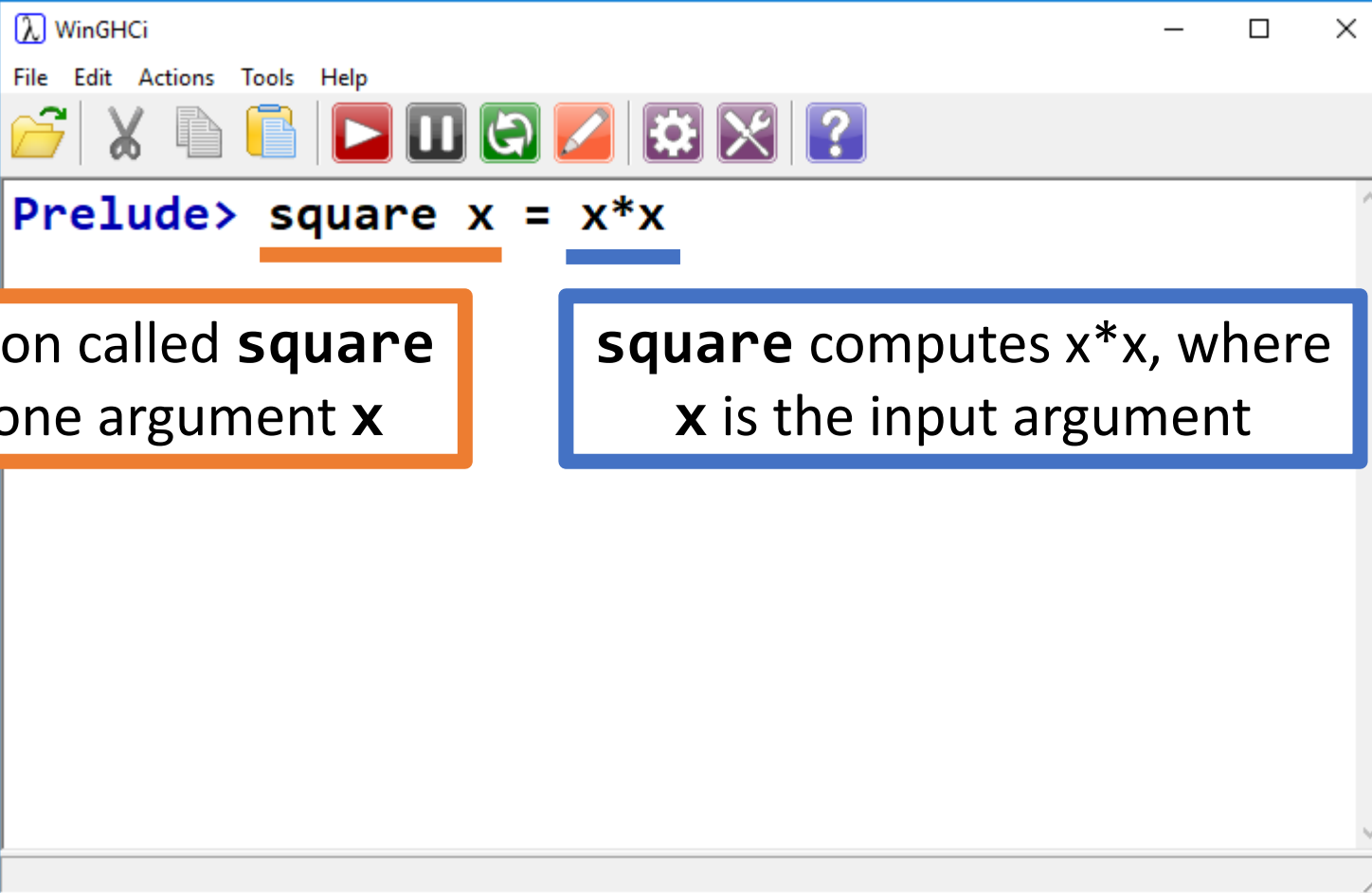


```
1 main = putStrLn "Hello, World!"
```

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

Functions in Haskell

Let's start simple:



The image shows a screenshot of the WinGHCi Haskell interpreter window. The window has a title bar 'WinGHCi' and a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for file operations (folder, copy, paste, save), execution (play, pause, refresh), and settings (gear, wrench, question mark). The main text area displays the Haskell code `Prelude> square x = x*x`. The word `square` is underlined with an orange line, and the variable `x` in the expression `x*x` is underlined with a blue line. Two callout boxes are present: an orange box on the left and a blue box on the right, both containing explanatory text about the function definition.

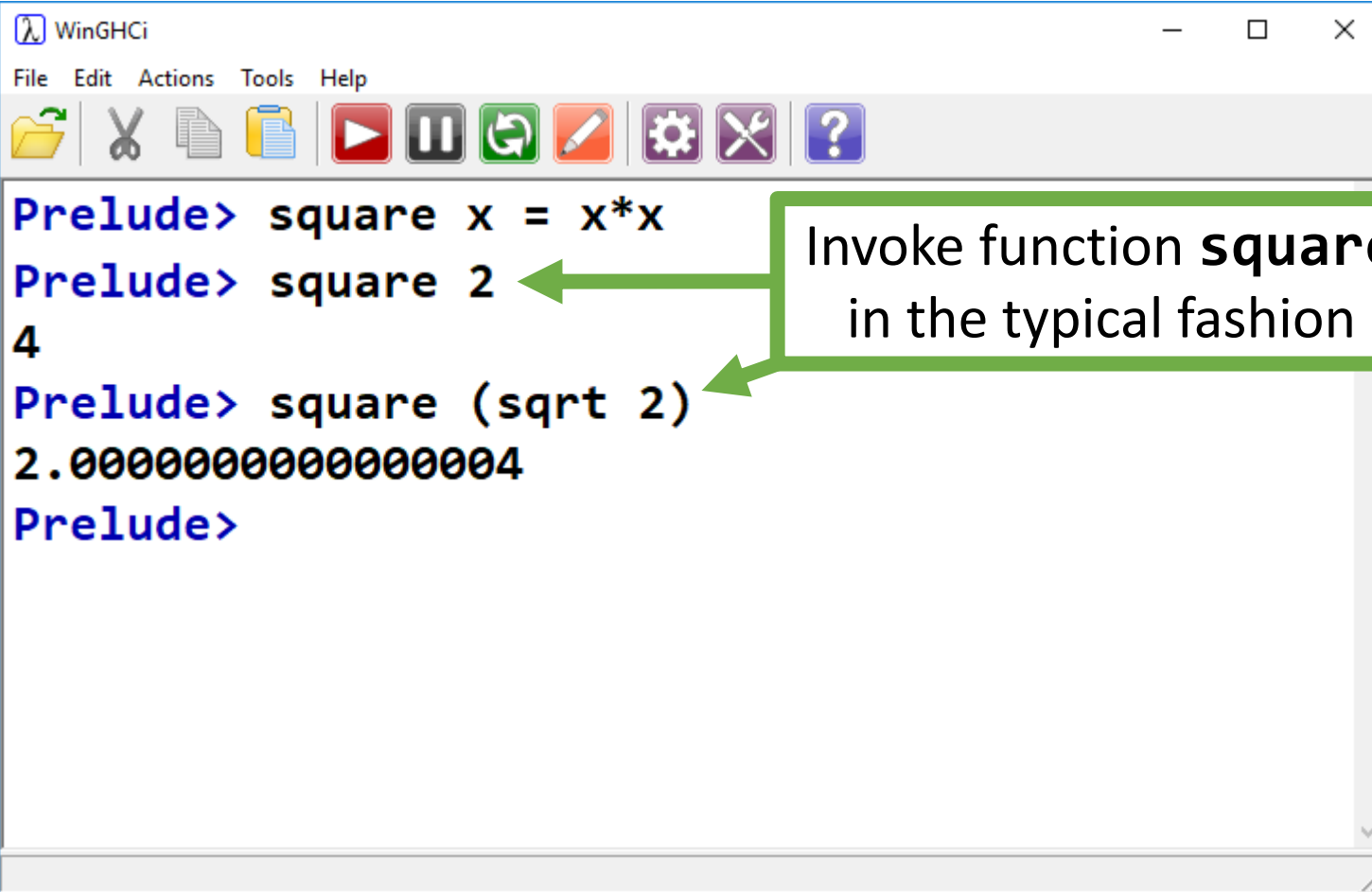
```
Prelude> square x = x*x
```

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:

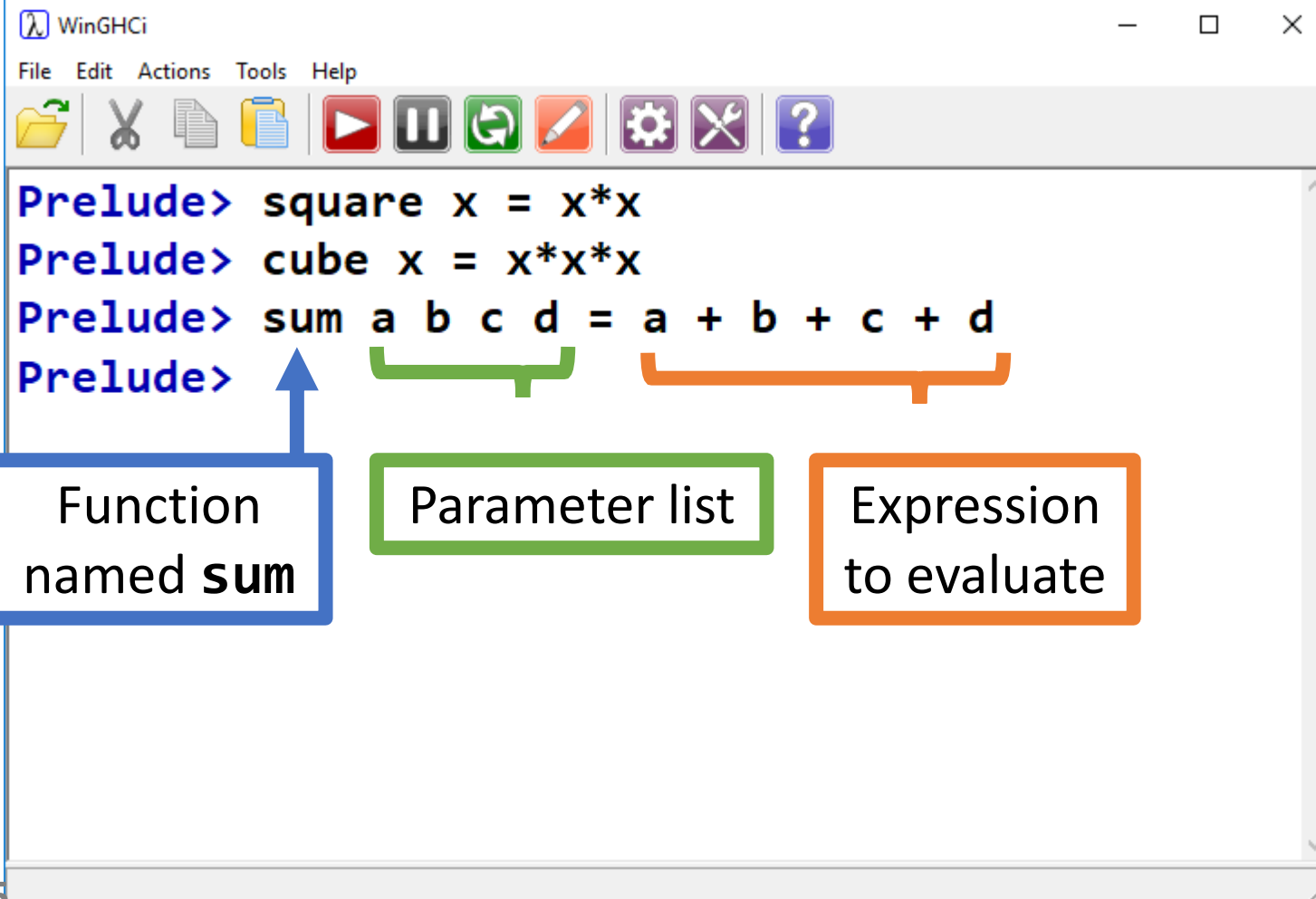


The image shows a screenshot of the WinGHCi terminal window. The window has a title bar 'WinGHCi' and a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for file operations (folder, copy, paste, save), execution (play, pause, refresh), editing (undo, redo, delete), and settings (gear, wrench, question mark). The terminal content shows the following interactions:

```
Prelude> square x = x*x
Prelude> square 2
4
Prelude> square (sqrt 2)
2.00000000000000004
Prelude>
```

A green callout box with a white border contains the text 'Invoke function **square** in the typical fashion'. Two green arrows point from this box to the 'square' function calls in the terminal: one to 'square 2' and another to 'square (sqrt 2)'.

Functions in Haskell



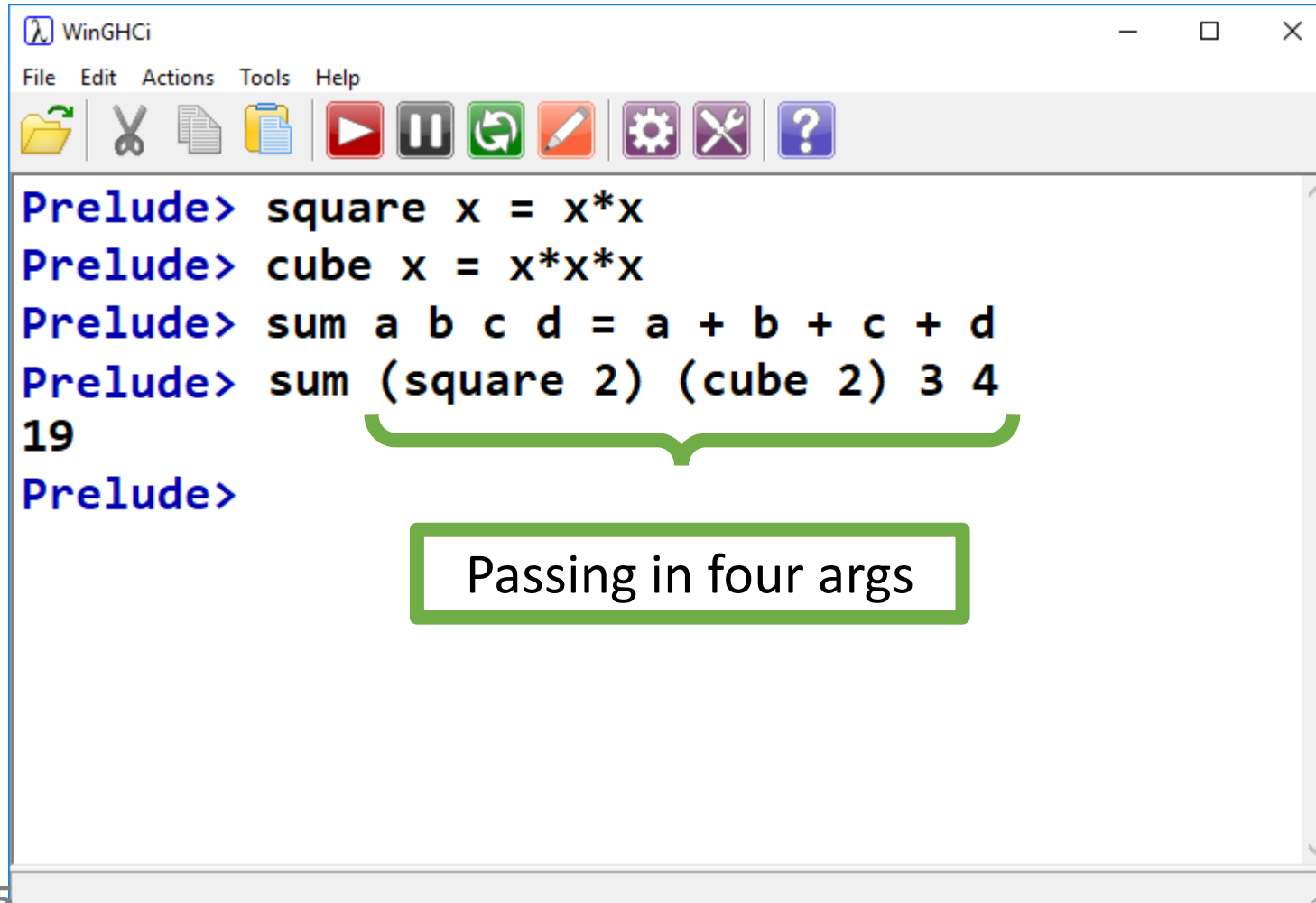
The image shows a screenshot of the WinGHCi window. The window title is "WinGHCi". The menu bar includes "File", "Edit", "Actions", "Tools", and "Help". The toolbar contains icons for file operations (folder, scissors, document), execution (play, pause, refresh), editing (pencil), settings (gear), and help (question mark). The main text area contains the following Haskell code:

```
Prelude> square x = x*x  
Prelude> cube x = x*x*x  
Prelude> sum a b c d = a + b + c + d  
Prelude>
```

Annotations are present on the code:

- A blue arrow points from the text "Function named **sum**" to the `sum` keyword in the third line.
- A green bracket underlines the parameters `a b c d` in the third line, with a label "Parameter list" in a green box below it.
- An orange bracket underlines the expression `a + b + c + d` in the third line, with a label "Expression to evaluate" in an orange box below it.

Functions in Haskell



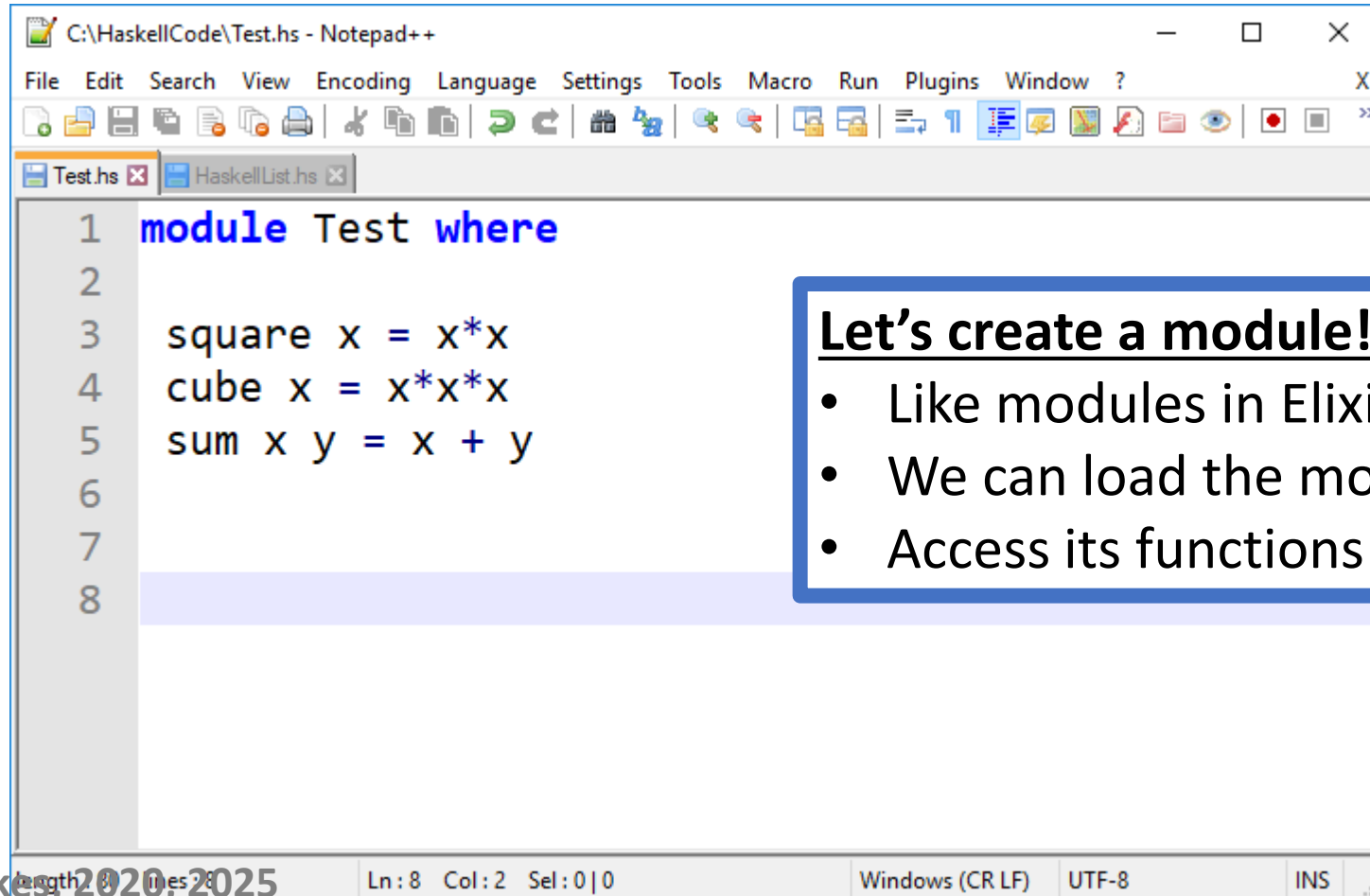
A screenshot of the WinGHCi Haskell interpreter window. The window has a title bar 'WinGHCi' and a menu bar with 'File', 'Edit', 'Actions', 'Tools', and 'Help'. Below the menu bar is a toolbar with icons for file operations and execution. The main text area contains the following Haskell code:

```
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 is drawn under the arguments `(square 2) (cube 2) 3 4` in the fourth line. Below this bracket, 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 titled 'C:\HaskellCode\Test.hs - Notepad++'. The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, Plugins, Window, and ?. The toolbar contains various icons for file operations and editing. The tab bar shows 'Test.hs' and 'HaskellList.hs'. The code editor contains the following Haskell code:

```
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
```

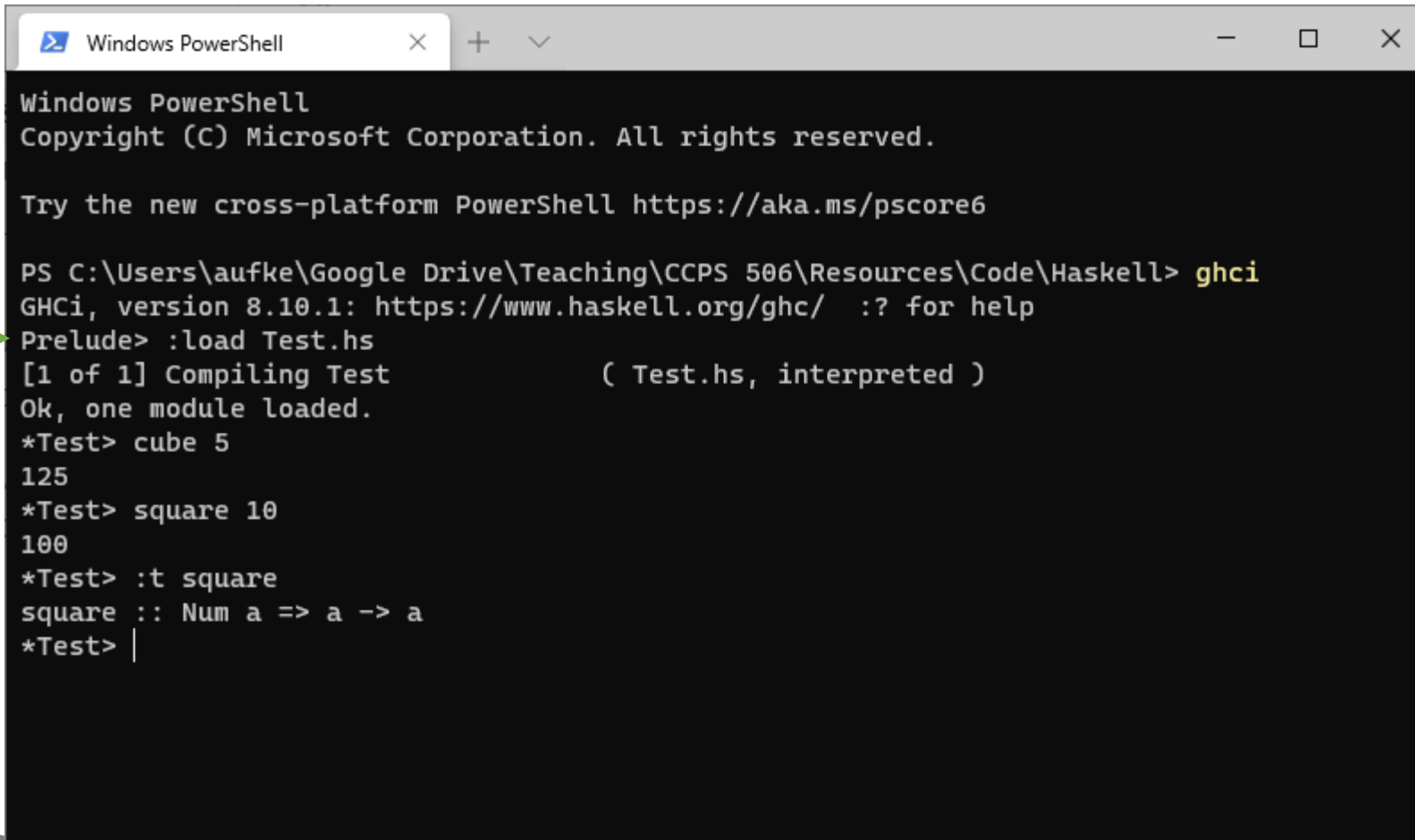
The status bar at the bottom shows 'Ln: 8 Col: 2 Sel: 0 | 0', 'Windows (CR LF)', 'UTF-8', and 'INS'.

Let's create a module!

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

Loading a Module

Use `:load` in terminal GHCi:



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

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

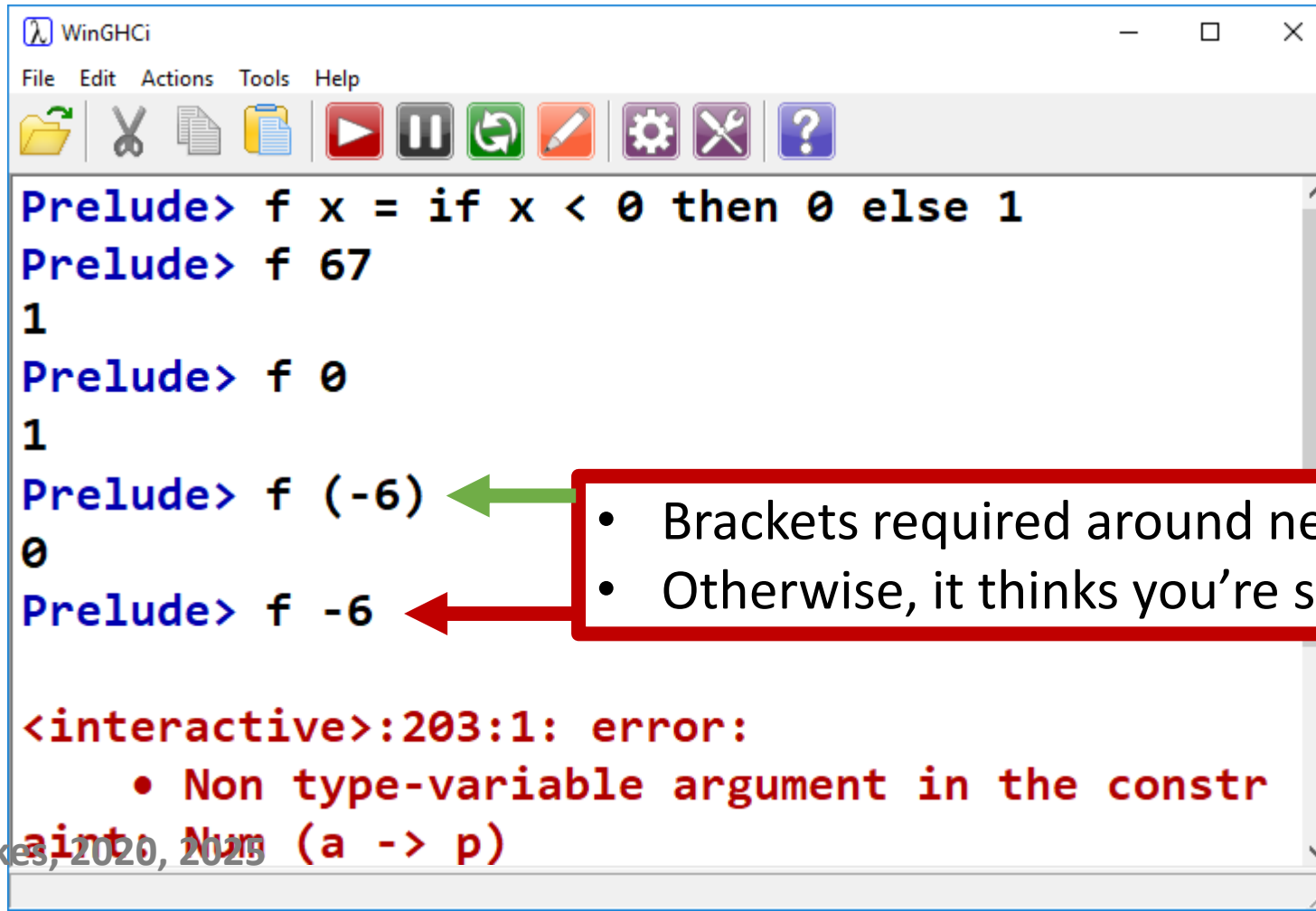
PS C:\Users\aufke\Google Drive\Teaching\CCPS 506\Resources\Code\Haskell> ghci
GHCi, version 8.10.1: https://www.haskell.org/ghc/  :? for help
Prelude> :load Test.hs
[1 of 1] Compiling Test                ( Test.hs, interpreted )
Ok, one module loaded.
*Test> cube 5
125
*Test> square 10
100
*Test> :t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

Control Structures

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

Control Structures

if then else



```
WinGHCi
File Edit Actions Tools Help
[Icons: File Explorer, Copy, Paste, Undo, Redo, Run, Break, Refresh, Erase, Settings, Wrench, Help]

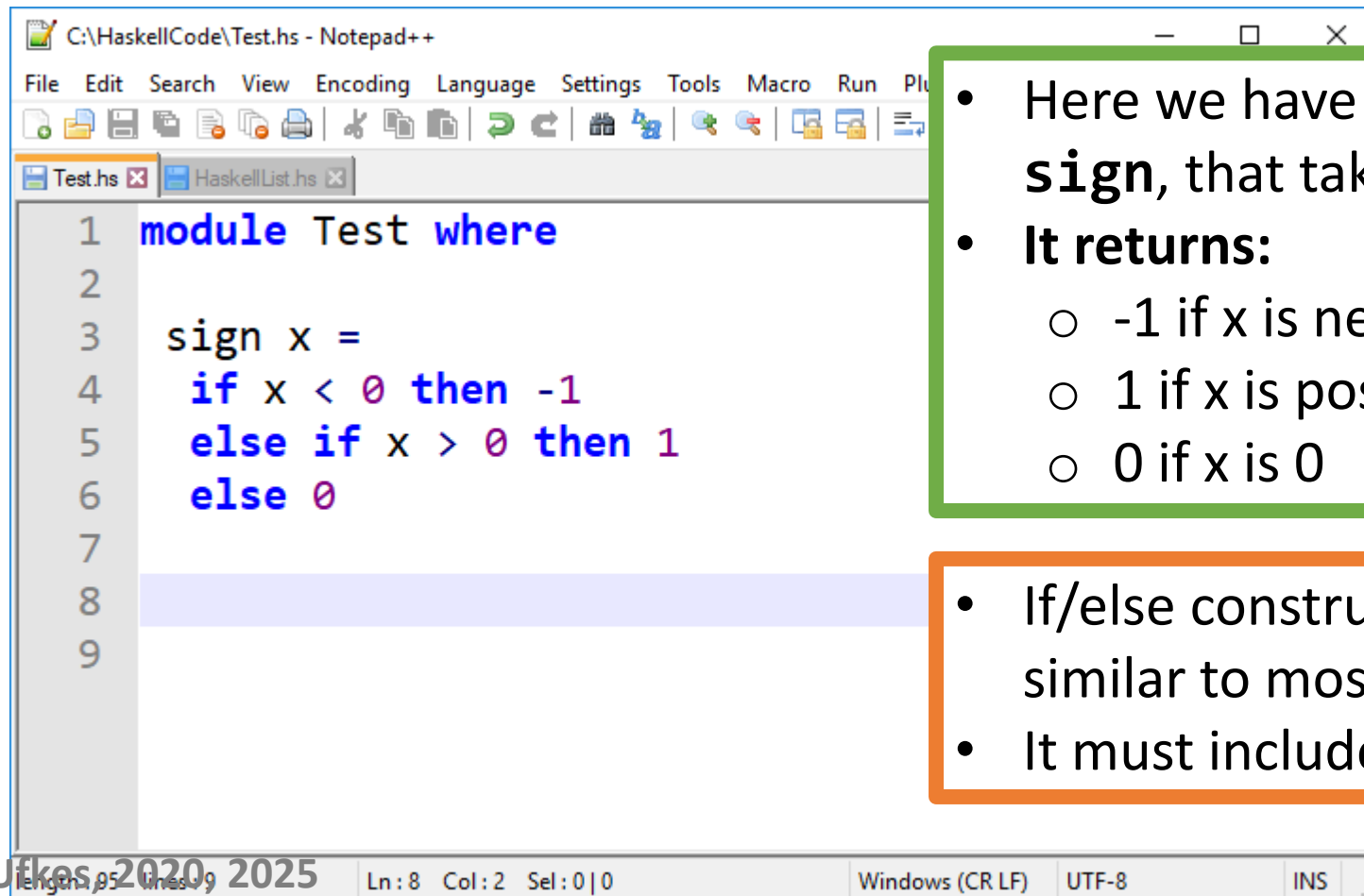
Prelude> f x = if x < 0 then 0 else 1
Prelude> f 67
1
Prelude> f 0
1
Prelude> f (-6)
0
Prelude> f -6
<interactive>:203:1: error:
  • Non type-variable argument in the constraint: Num (a -> p)
    >
```

A red box highlights the error message and the two negative arguments. A green arrow points from the box to the `f (-6)` line, and a red arrow points from the box to the `f -6` line.

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

Control Structures

if then else if then else



The screenshot shows a Notepad++ window titled 'C:\HaskellCode\Test.hs - Notepad++'. The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Tools, Macro, Run, and Plug-ins. The toolbar contains various icons for file operations and editing. Two tabs are open: 'Test.hs' and 'HaskellList.hs'. The 'Test.hs' tab is active, displaying the following Haskell 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
```

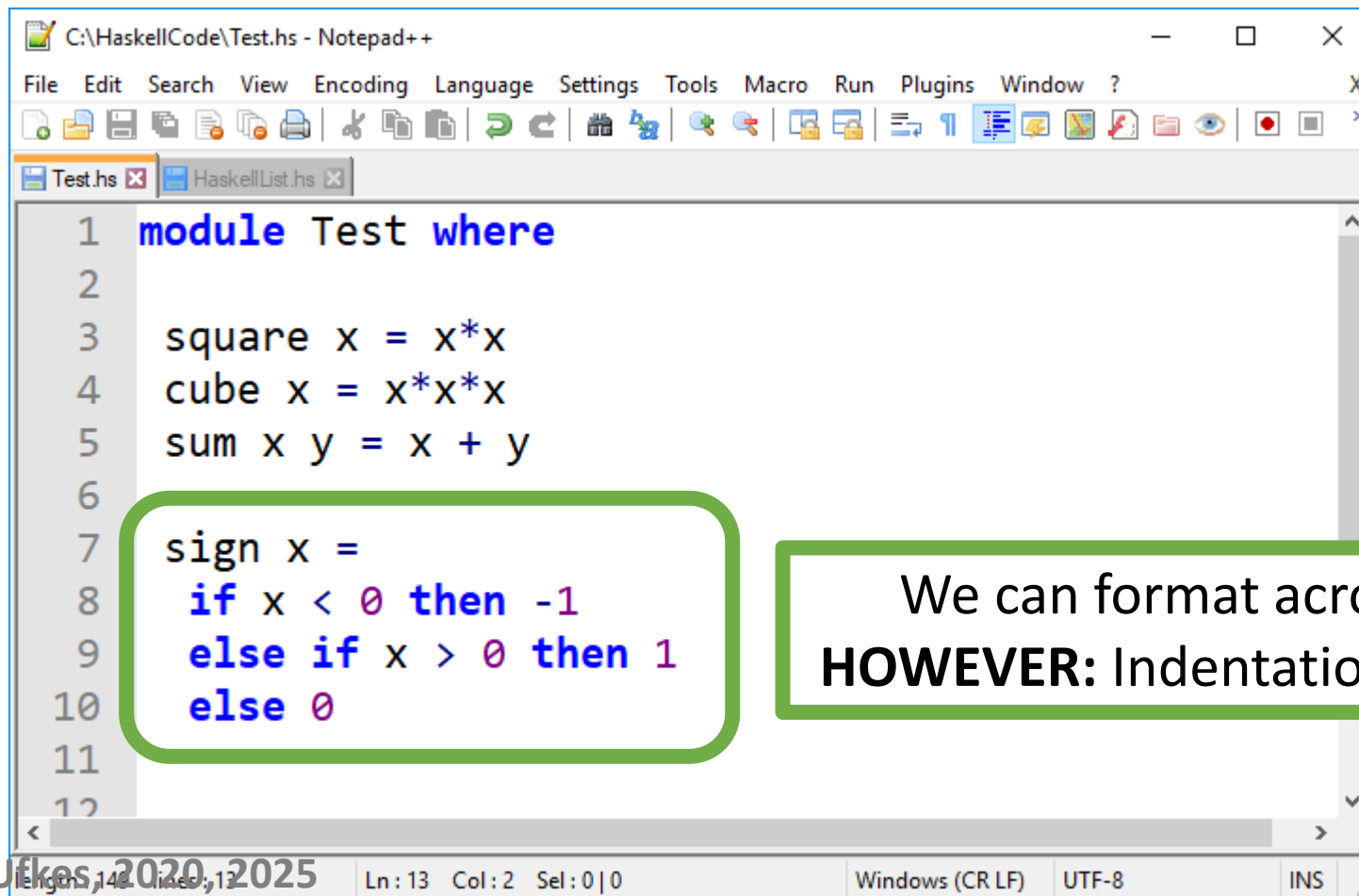
The status bar at the bottom shows 'length: 95 lines: 9', 'Ln: 8 Col: 2 Sel: 0 | 0', 'Windows (CR LF)', 'UTF-8', and 'INS'.

- 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
```

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

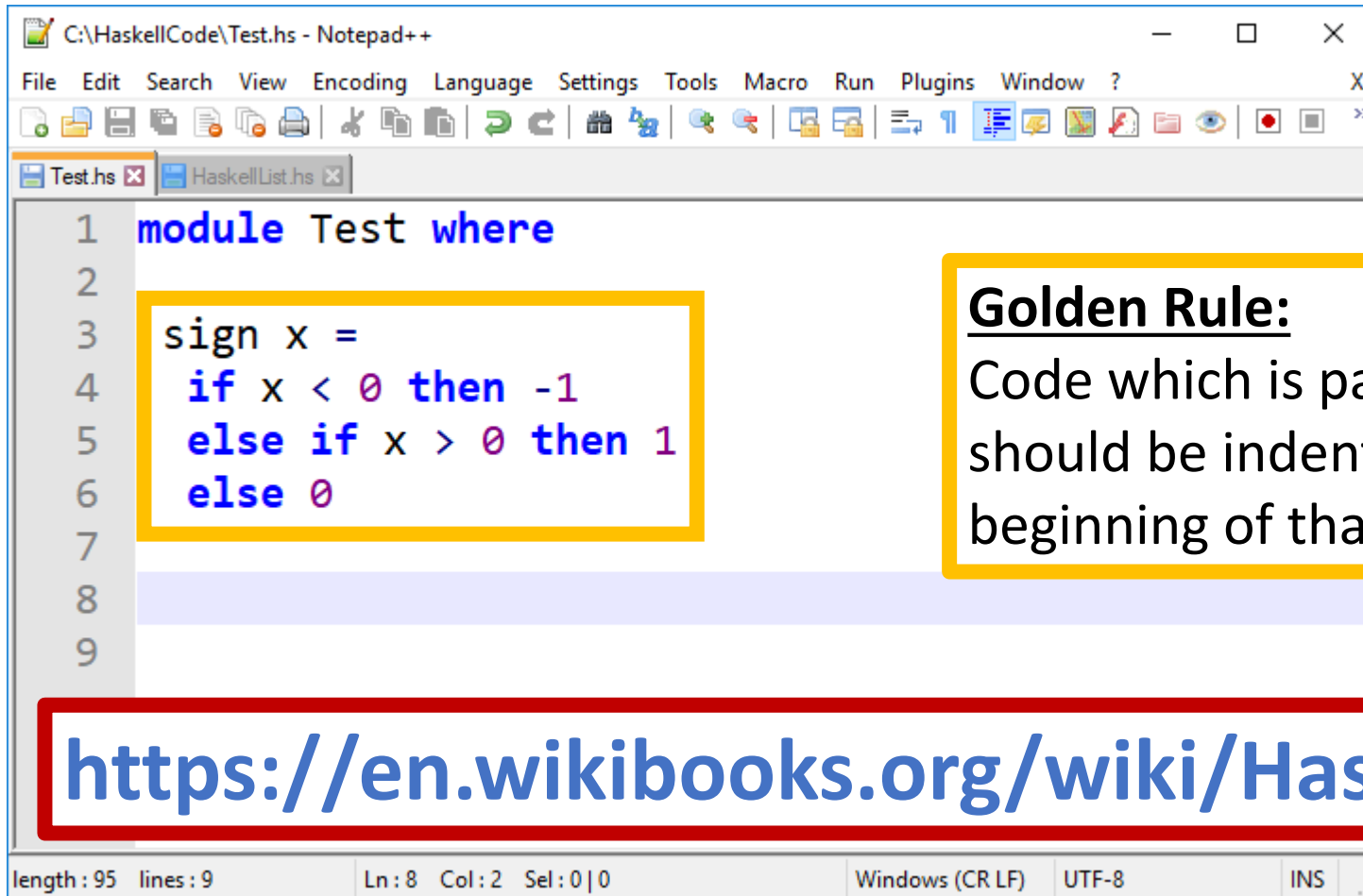
© Alex Ufkes, 2020, 2025

```
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> |
```

Indenting in Haskell

A screenshot of a Notepad++ window titled 'C:\HaskellCode\Test.hs - Notepad++'. The window shows a Haskell code file with line numbers 1 through 9 on the left. The code is as follows:

```
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 block from line 3 to line 6 is highlighted with a yellow box. The status bar at the bottom shows 'length : 95 lines : 9', 'Ln : 8 Col : 2 Sel : 0 | 0', 'Windows (CR LF)', 'UTF-8', and 'INS'.

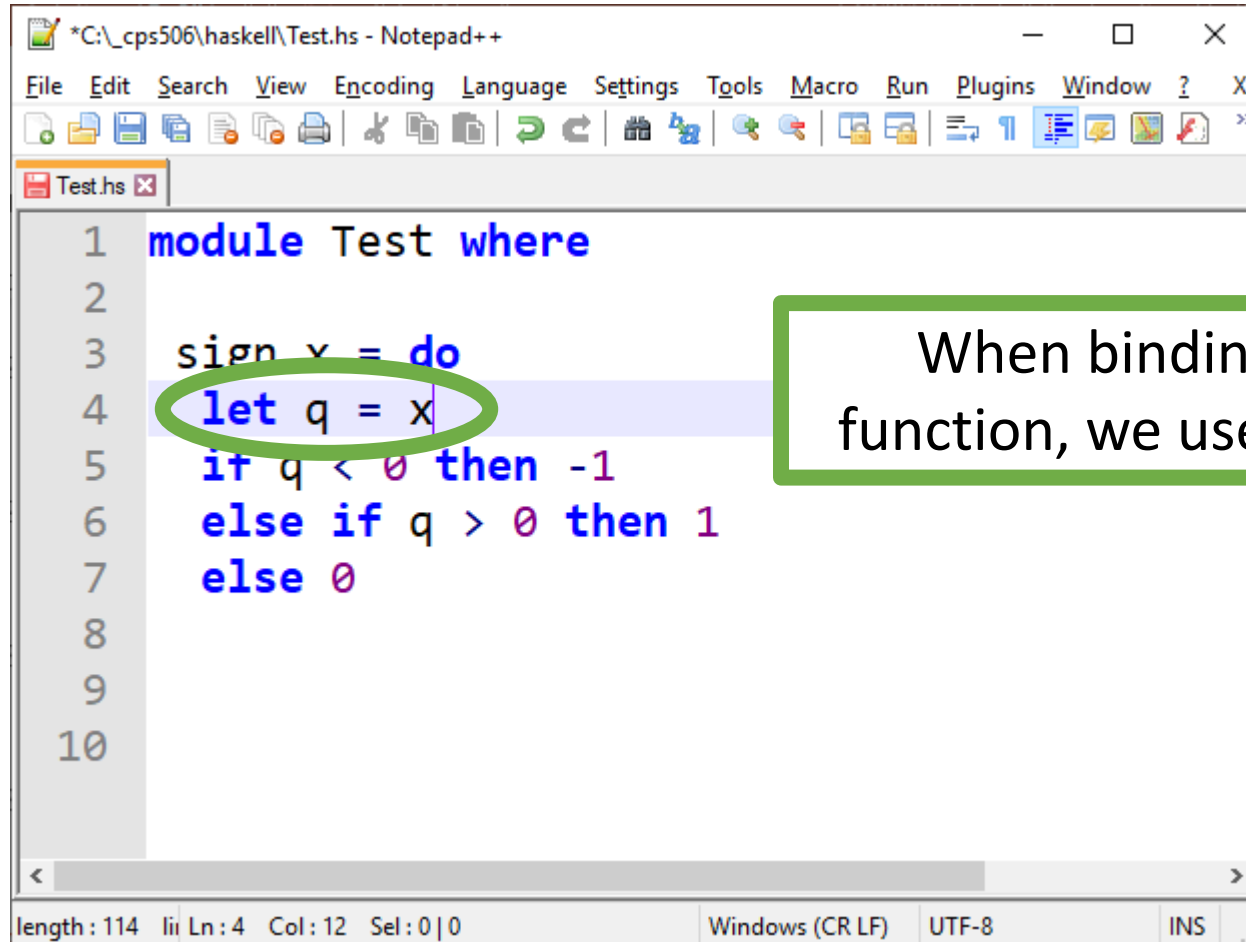
Golden Rule:

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.

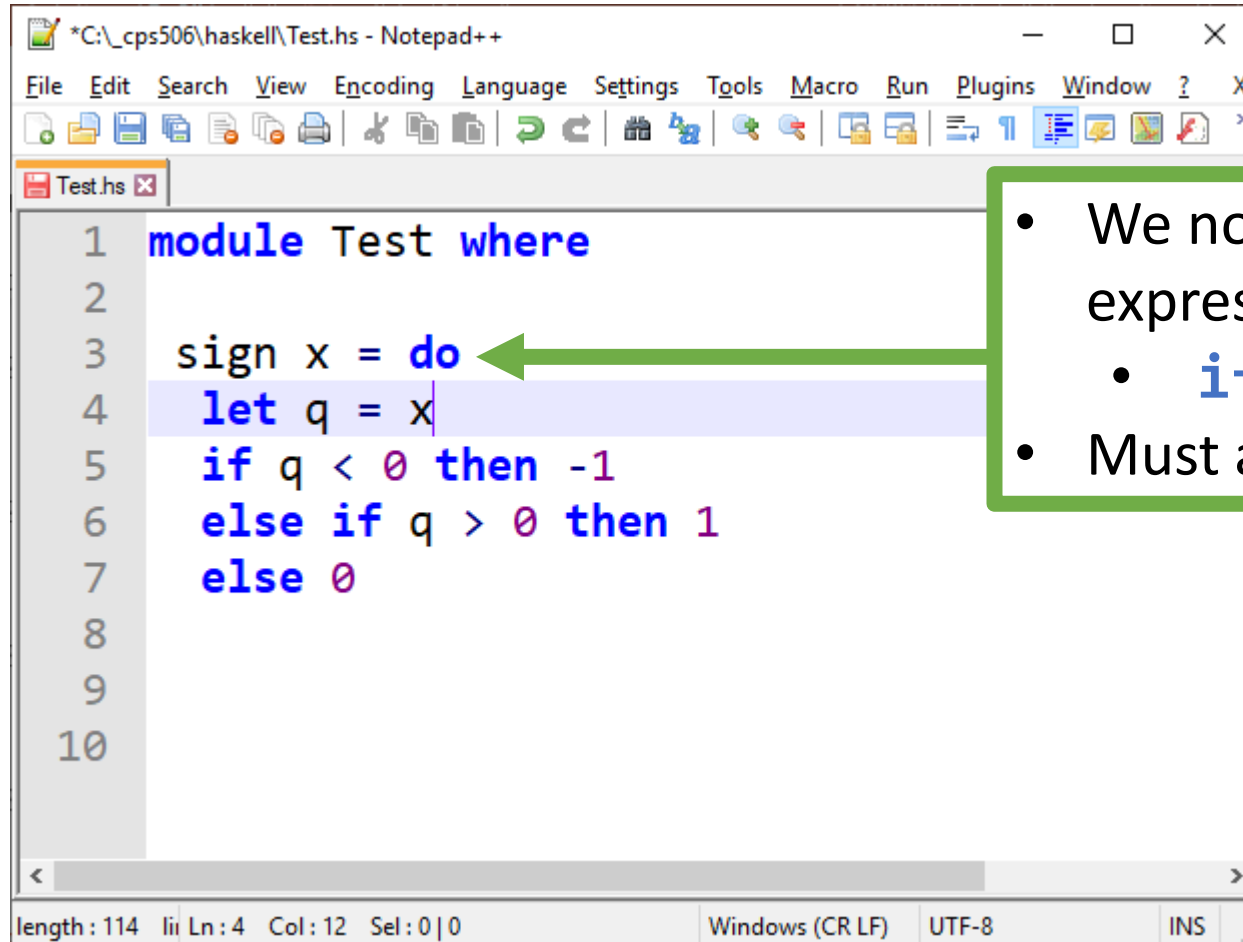
Local Names in Functions



```
*C:\cps506\haskell\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
Test.hs
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
```

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

Multiple Expressions



```
*C:\cps506\haskell\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X

Test.hs 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
```

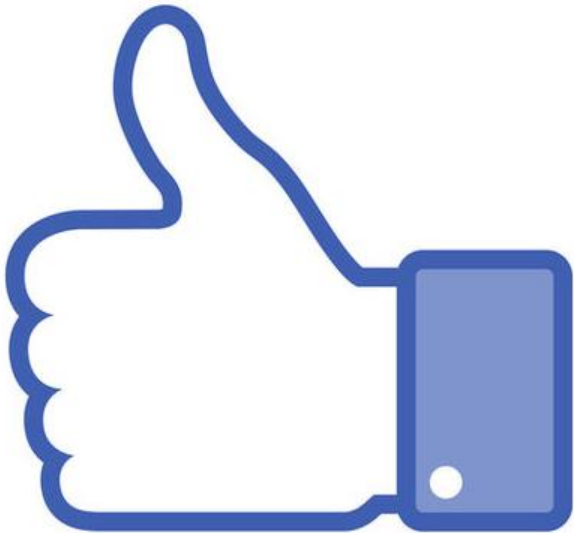
- We now have two expressions in this function!
 - **if/else** and **let**
- Must add the **do** keyword

```
*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
```

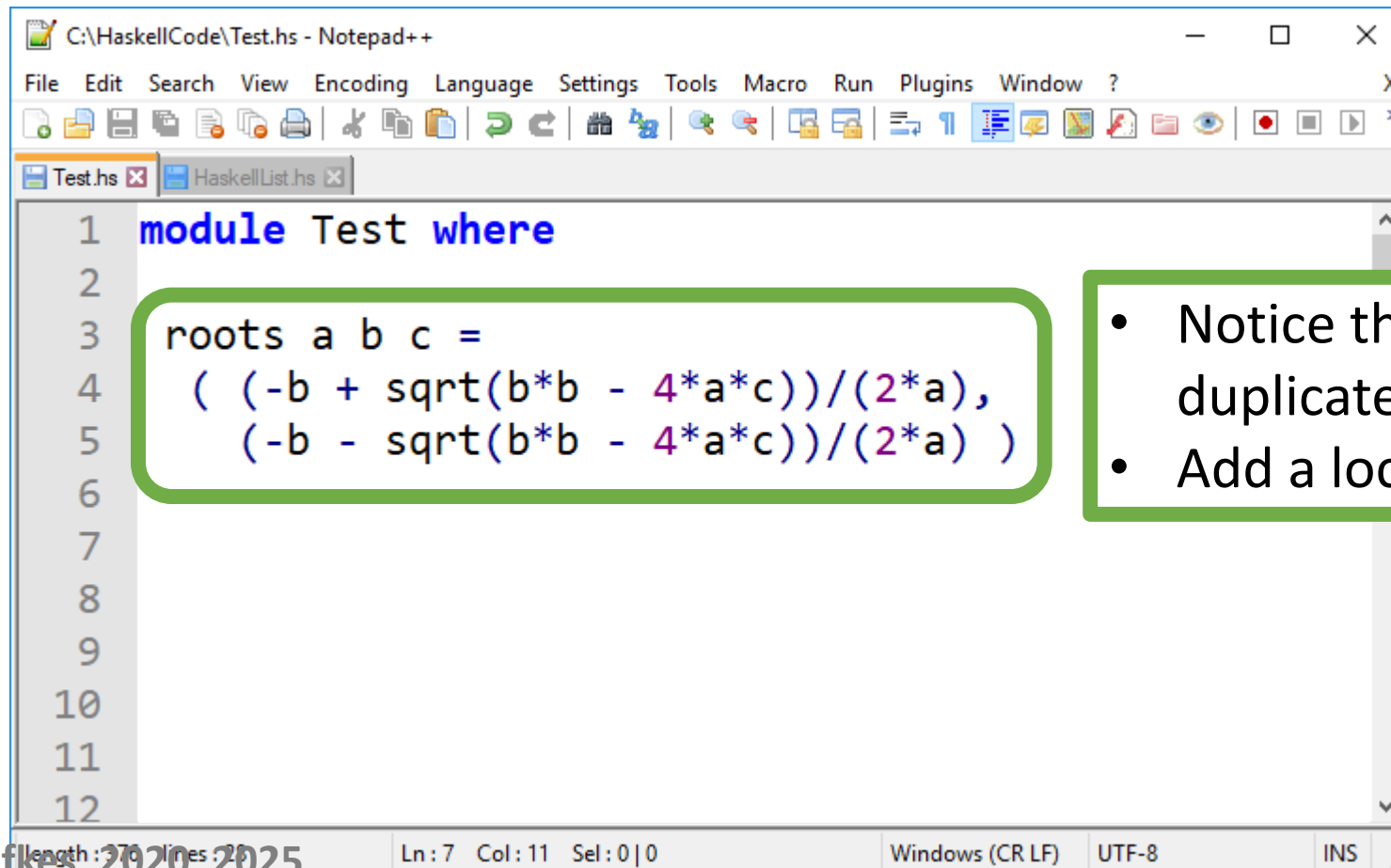
```
WinGHCi
File Edit Actions Tools Help

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



Return Multiple *Things*?

Lists/tuples to the rescue!

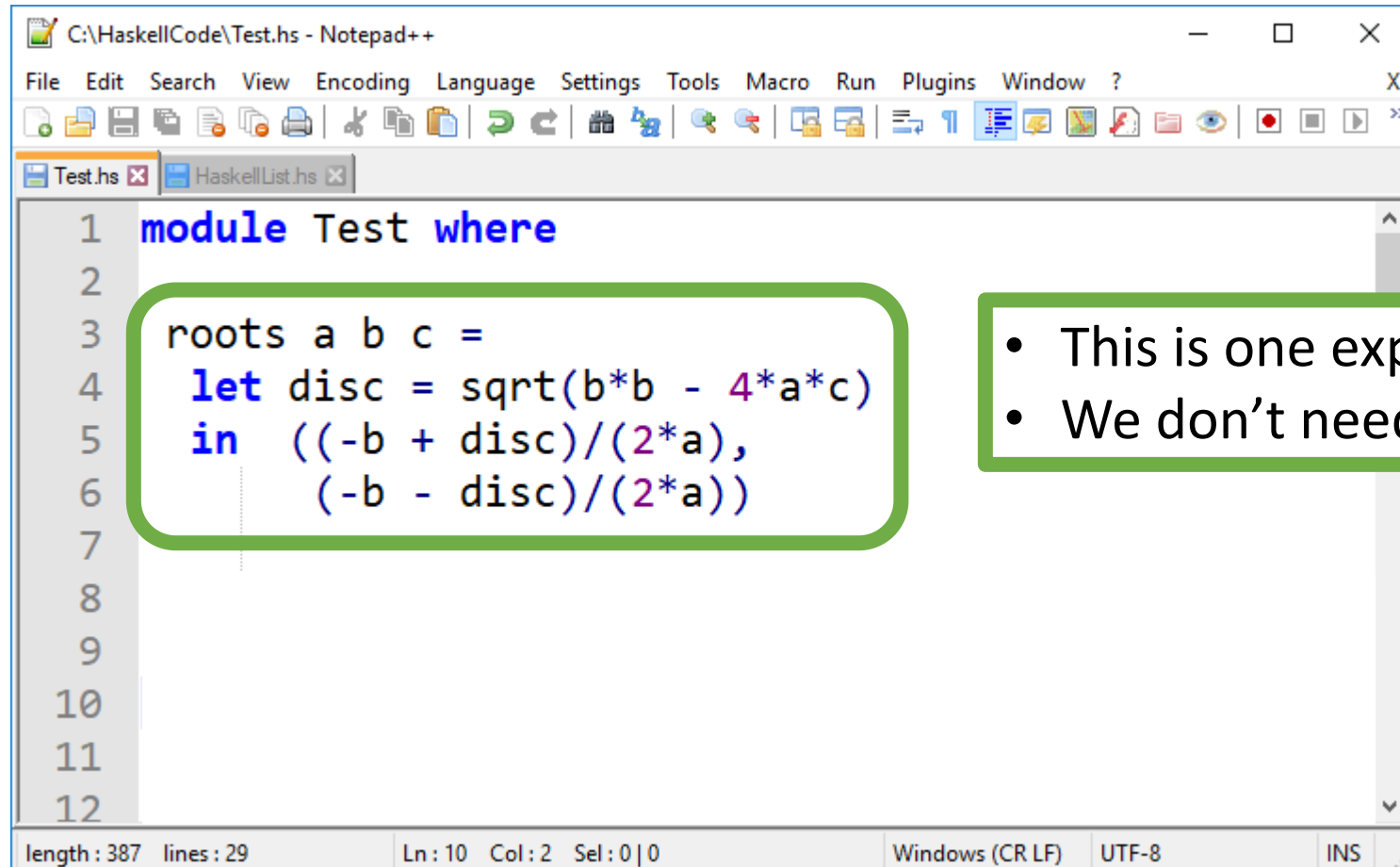


```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs HaskellList.hs
1 module Test where
2
3 roots a b c =
4   ( (-b + sqrt(b*b - 4*a*c))/(2*a),
5     (-b - sqrt(b*b - 4*a*c))/(2*a) )
6
7
8
9
10
11
12
```

length: 270 lines: 20 Ln: 7 Col: 11 Sel: 0|0 Windows (CR LF) UTF-8 INS

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

let/in Expression



```
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
9
10
11
12
```

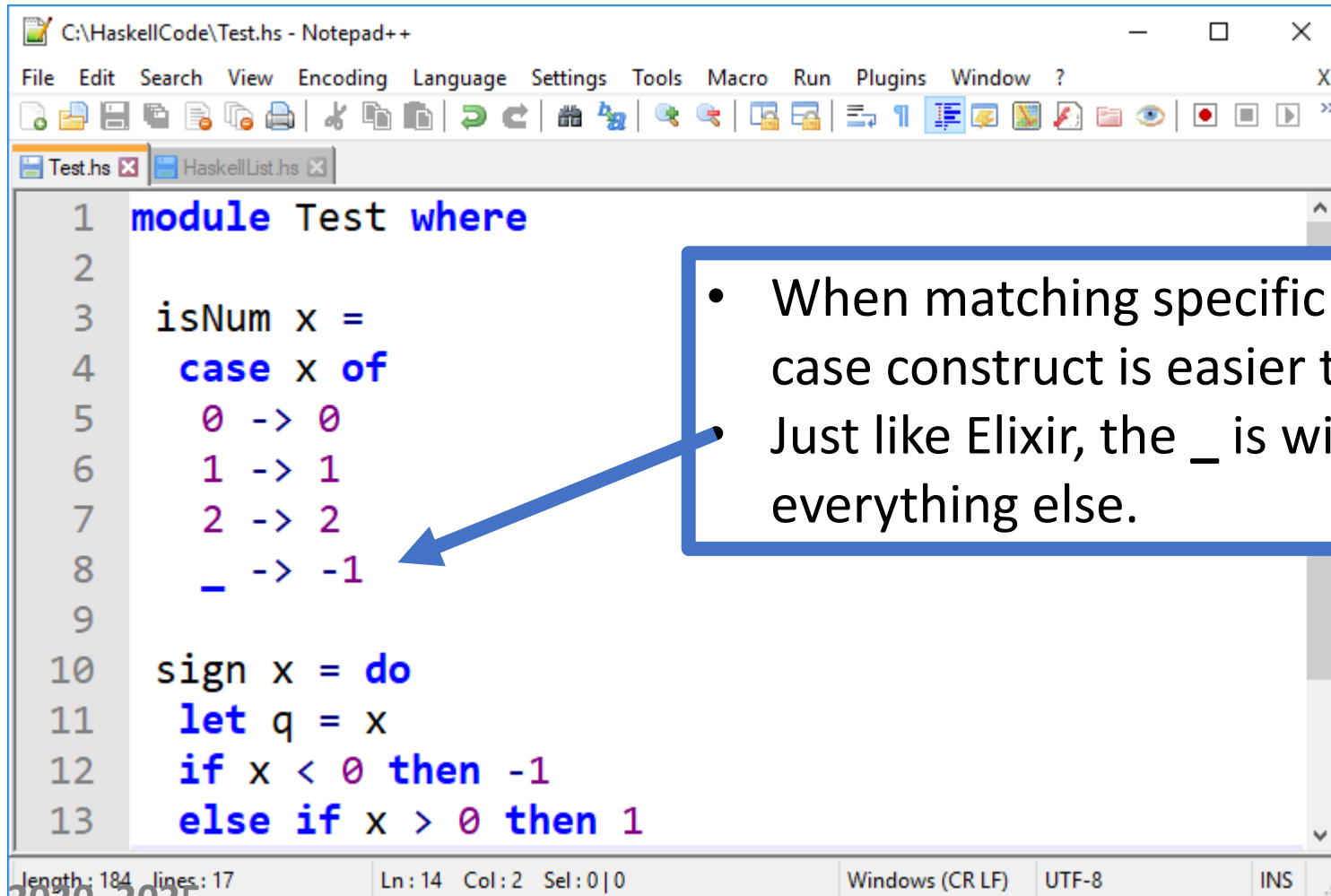
length: 387 lines: 29 Ln: 10 Col: 2 Sel: 0|0 Windows (CR LF) UTF-8 INS

- 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
9
10
11
12
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> |
```

Case Expression



```
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 a Haskell development environment with two windows. The left window, titled 'C:\HaskellCode\Test.hs - Notepad++', displays a Haskell source file with a case expression. The right window, titled 'WinGHCi', shows the interactive interpreter running the code.

Source File (Test.hs):

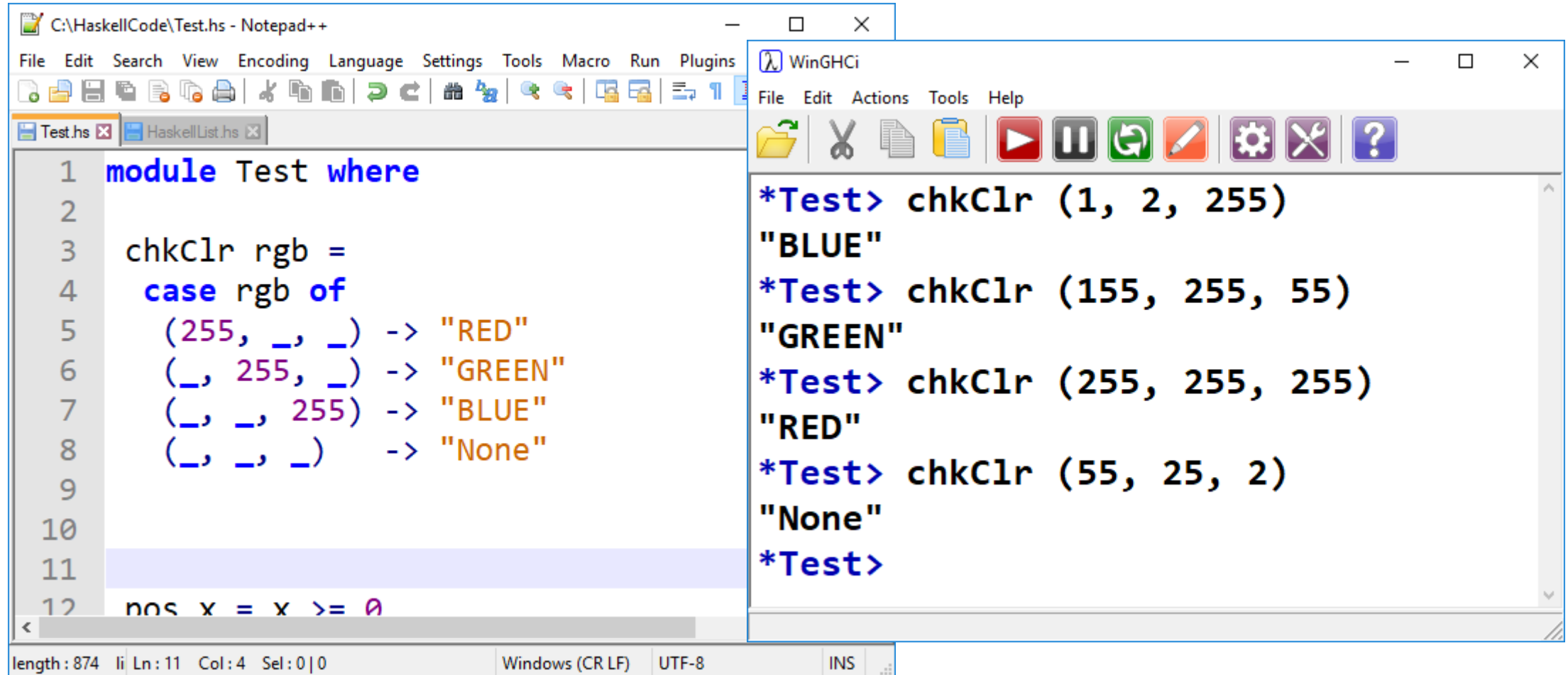
```
1 module Test where
2
3 isNum x =
4   case x of
5     0 -> 0
6     1 -> 1
7     2 -> 2
8     _ -> -1
9
10 sign x = do
11   let q = x
12   if x < 0 then -1
13   else if x > 0 then 1
```

WinGHCi Output:

```
*Test> :reload
Ok, one module loaded.
*Test> isNum 0
0
*Test> isNum 1
1
*Test> isNum 2
2
*Test> isNum 999
-1
*Test>
```

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

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' using a 'case' expression for pattern matching on RGB values. The right window is WinGHCI, showing the execution of the 'chkClr' function with various arguments, demonstrating how the pattern matching logic returns 'RED', 'GREEN', 'BLUE', or 'None'.

```
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>
```

length: 874 | Ln: 11 Col: 4 Sel: 0 | 0 Windows (CR LF) UTF-8 INS

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-p
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
Ok, one module loaded.
*Test> :reload
*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

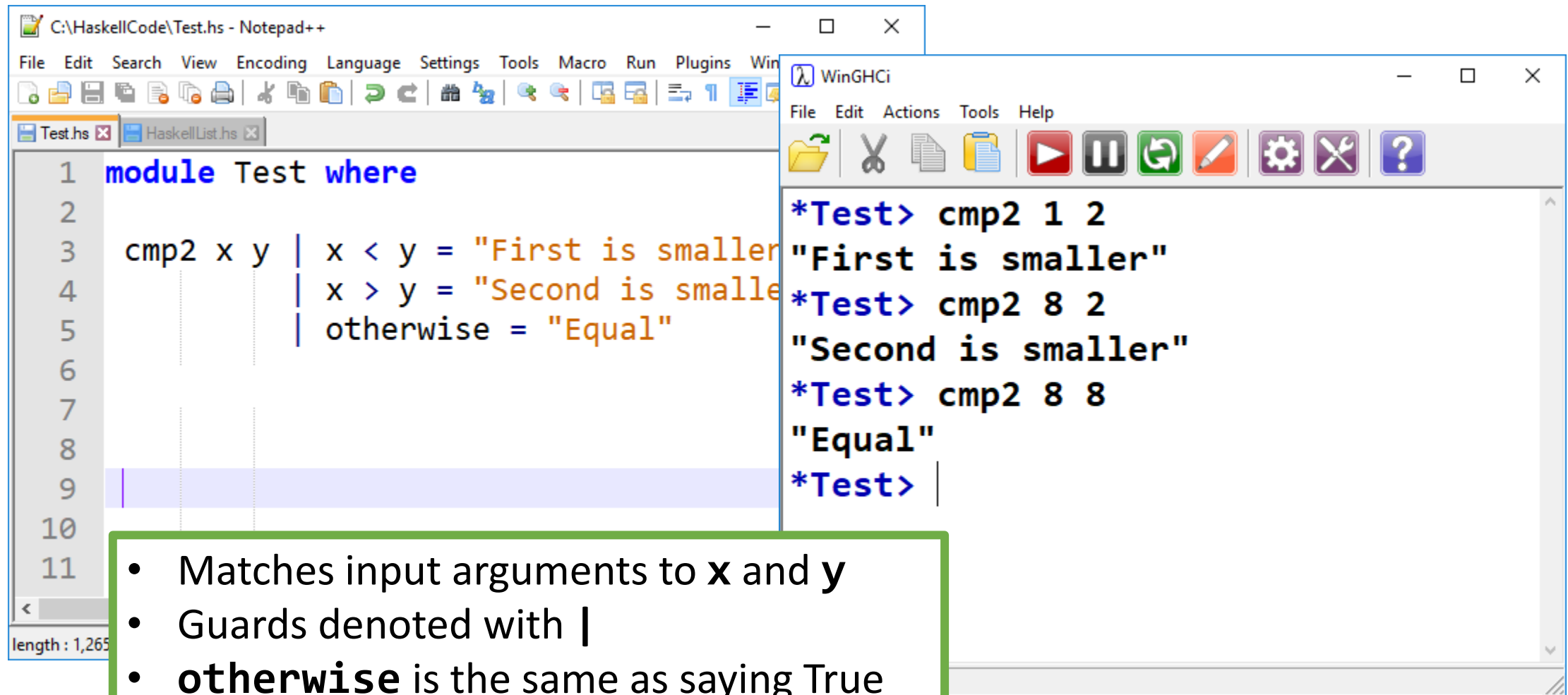
```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
Test.hs HaskellList.hs
1 module Test where
2
3 chkAxis (0, _) = (0, 1)
4 chkAxis (_, 0) = (1, 0)
5 chkAxis (a, b) = (a, b)
6
7
```

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

length: 1,061 | Ln: 8 Col: 2 Sel: 0 | 0 Windows (CR LF) UTF-8 INS

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

Functions: Guards



The image shows a Notepad++ window with a Haskell module named 'Test' and a WinGHCi window showing the execution of the 'cmp2' function. The Haskell code defines 'cmp2' with three guards: 'x < y', 'x > y', and 'otherwise'. The WinGHCi window shows three test cases: 'cmp2 1 2' returns 'First is smaller', 'cmp2 8 2' returns 'Second is smaller', and 'cmp2 8 8' returns 'Equal'.

```
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
```

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

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

Recursion

```
C:\HaskellCode\Test.hs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Win
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

© Alex Ufkes, 2020, 2025

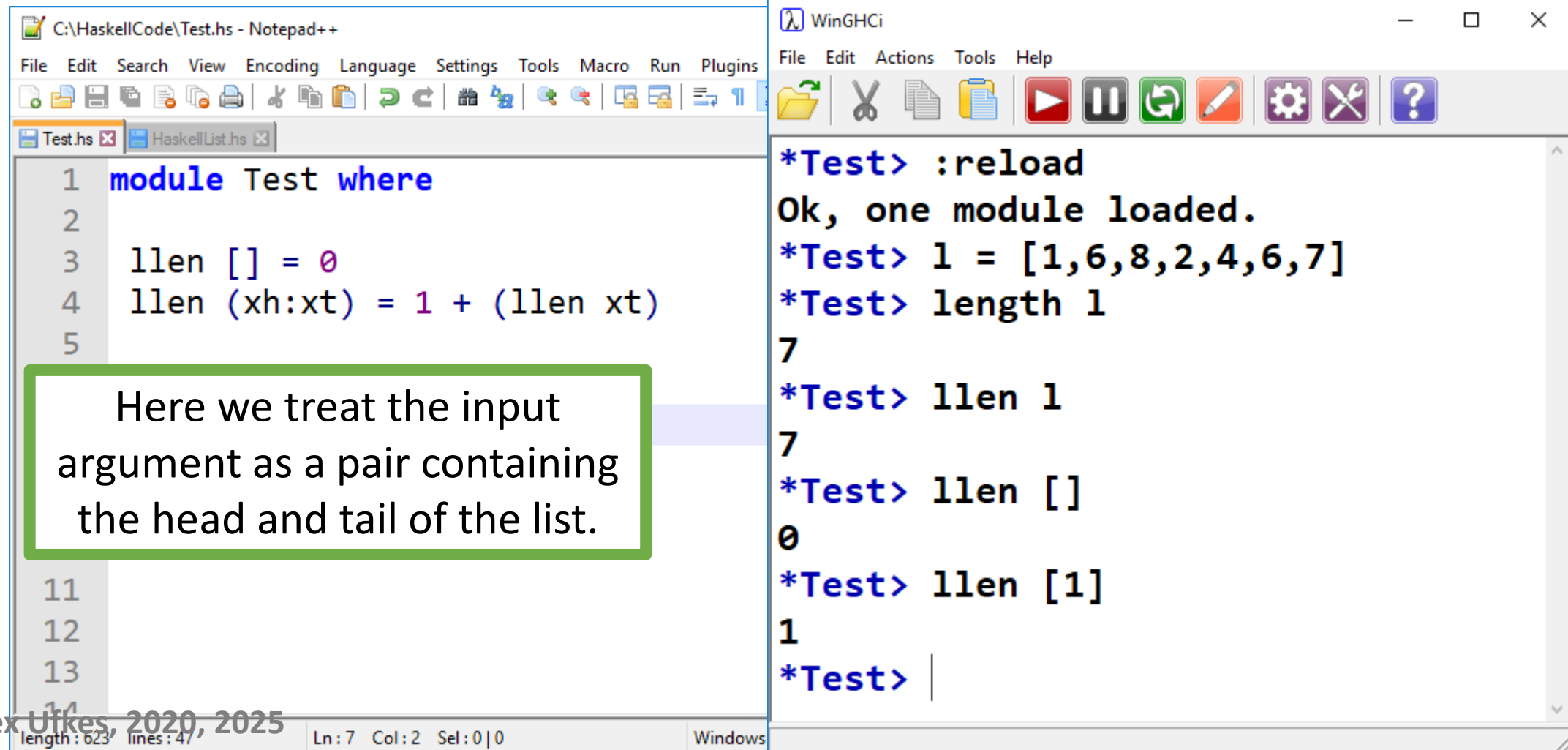
```
WinGHCi
File Edit Actions Tools Help
Built-in length function
*Test> :reload
Ok, one module loaded.
*Test> l = [1, 3, 2, 7, 5]
*Test> length l
5
Our own user function
*Test> llen l
5
*Test> llen []
0
*Test> llen [1]
1
*Test>
```


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++ editing a Haskell file named Test.hs. It contains 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 function definition in the code, with a callout explaining the `xh:xt` pattern.

Test.hs - Notepad++

```
1 module Test where
2
3 llen [] = 0
4 llen (xh:xt) = 1 + (llen xt)
5
11
12
13
14
```

length: 623 lines: 47

WinGHCi

```
*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.

© Alex Ufkes, 2020, 2025

Recursion: filter

- Returns true if $x \geq 0$
- False otherwise

```
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
```

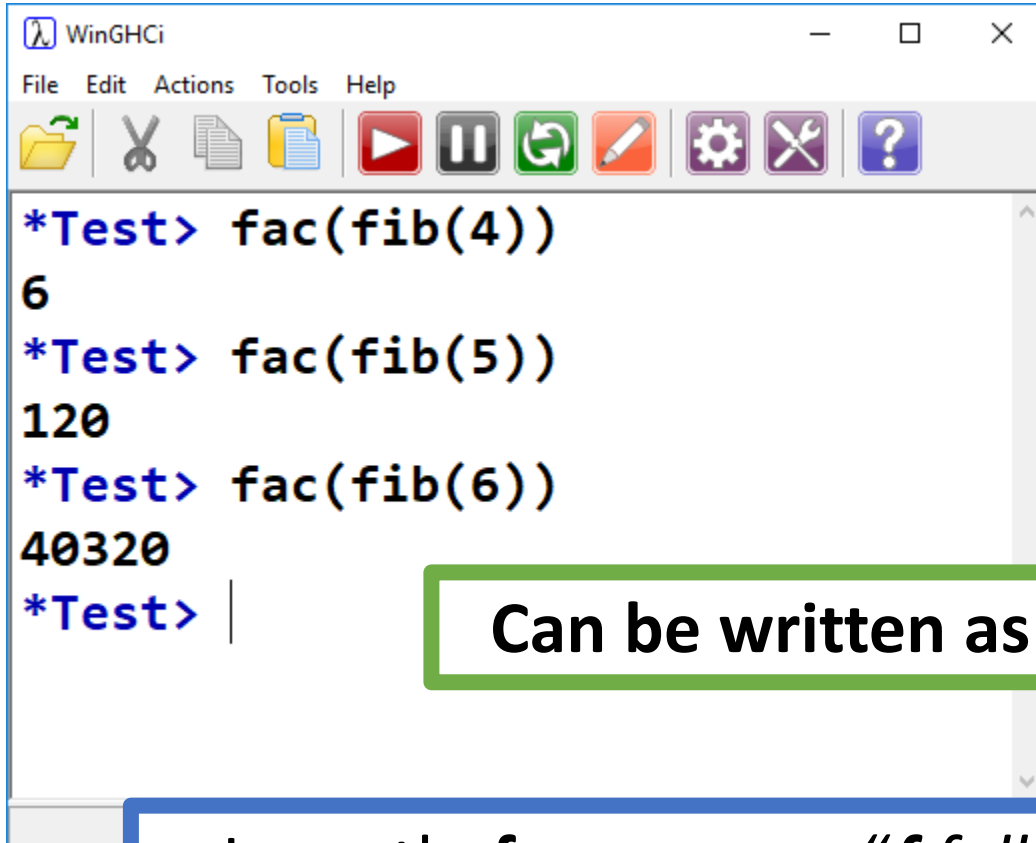
- First argument is a Boolean function
- Second input is a list
- Base case is if the list is empty
- Otherwise, we call the function p with the head of the list.
- If true, append it to the running list
- If false, do not append
- In both cases, make the recursive call with the tail.

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
```

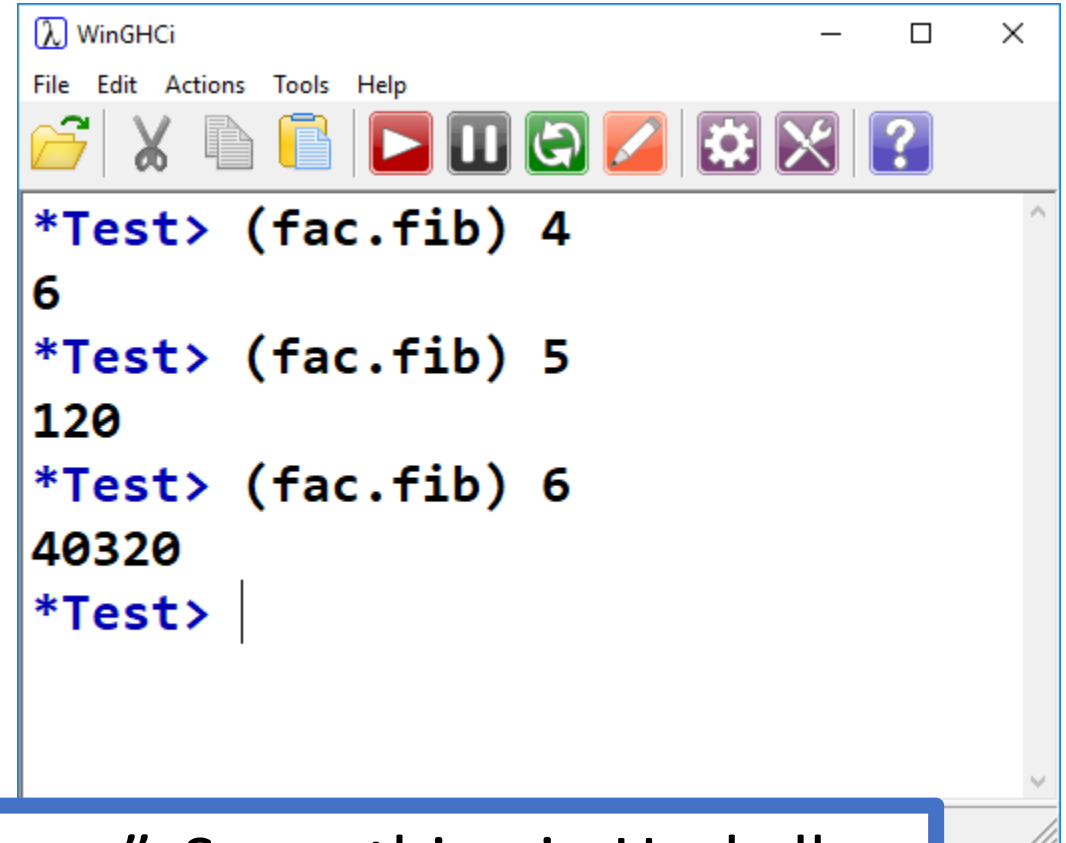
```
WinGHCi
File Edit Actions Tools Help
Test pos function
*Test> pos 4
True
*Test> pos (-5)
False
*Test> l = [-1, 2, -3, 4, -5, 6]
*Test> filt pos l
[2,4,6]
*Test> filt pos [-1]
[]
*Test> filt pos []
[]
*Test>
```

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:

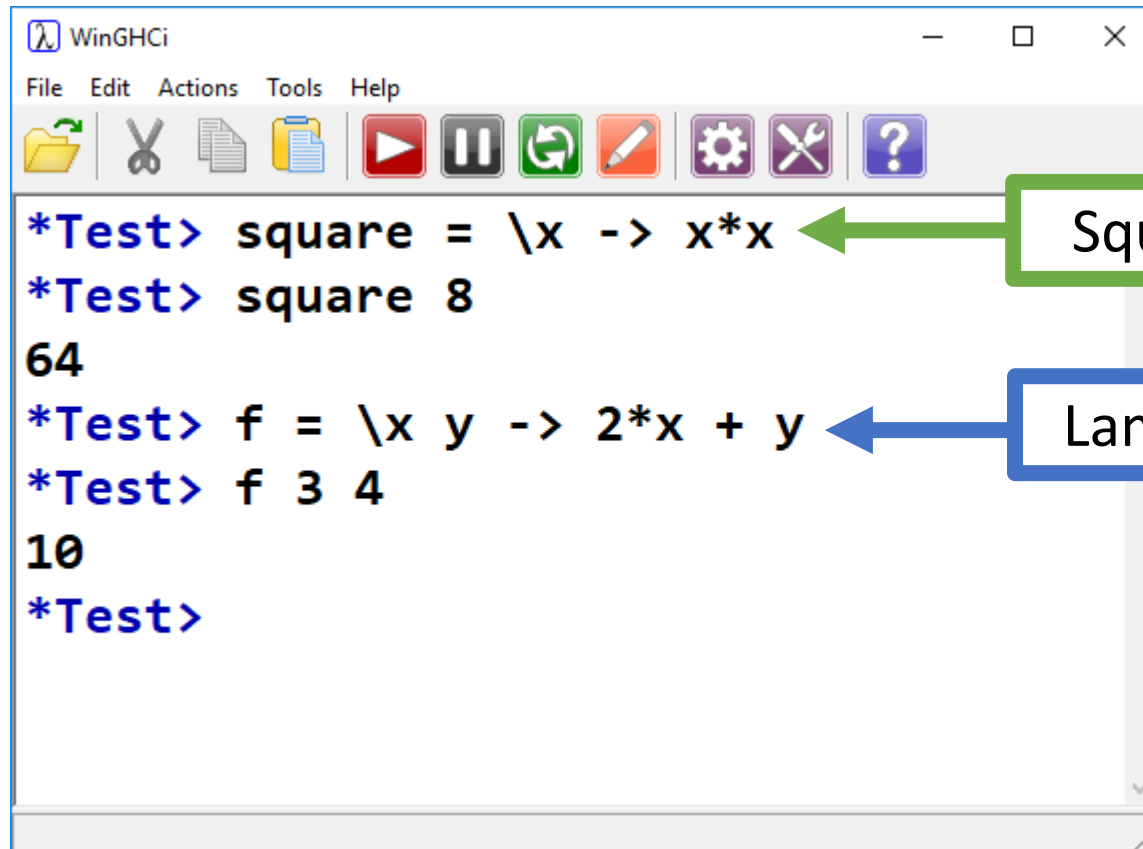


```
WinGHCi
File Edit Actions Tools Help
[Icons]
*Test> (fac.fib) 4
6
*Test> (fac.fib) 5
120
*Test> (fac.fib) 6
40320
*Test> |
```

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

Lambda Functions

Like anonymous functions in Elixir:



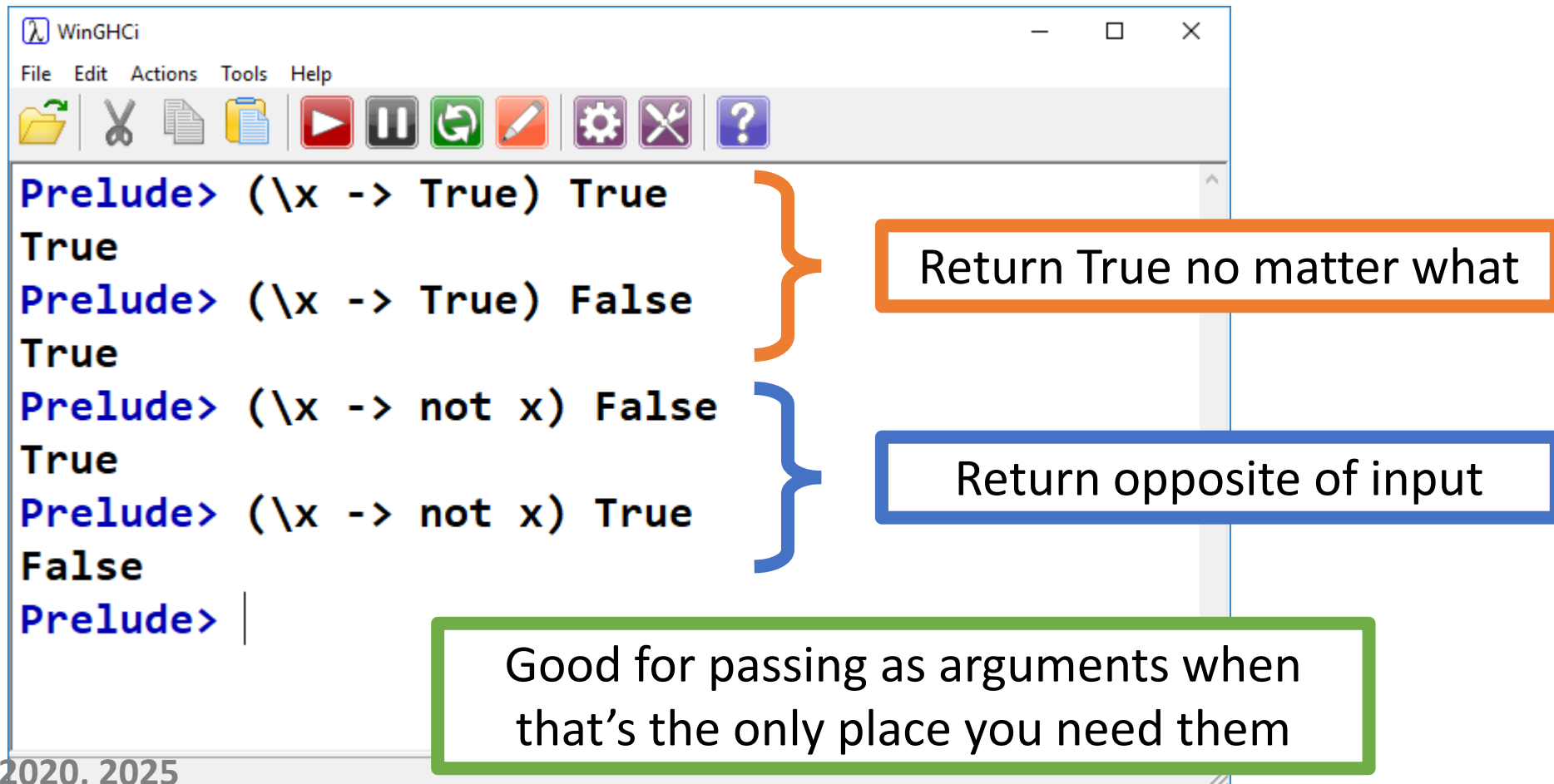
```
WinGHCi
File Edit Actions Tools Help
[Icons]
*Test> square = \x -> x*x
*Test> square 8
64
*Test> f = \x y -> 2*x + y
*Test> f 3 4
10
*Test>
```

Square as Lambda function

Lambda function with two args

Lambda Functions

They don't need names!



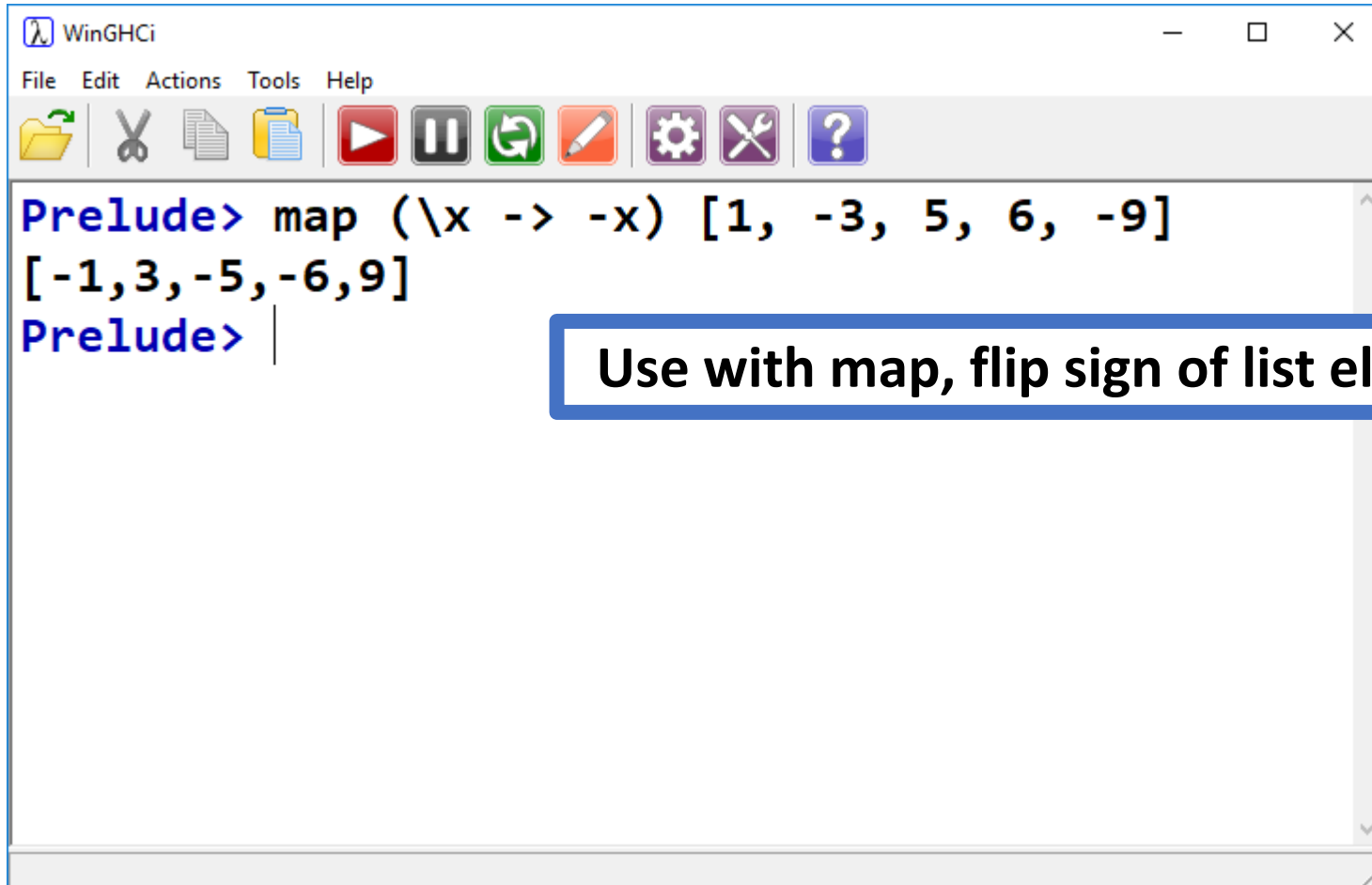
The image shows a WinGHCi terminal window with the following content:

```
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> |
```

Annotations:

- An orange bracket groups the first two lines of code, with a callout box stating: "Return True no matter what".
- A blue bracket groups the next two lines of code, with a callout box stating: "Return opposite of input".
- A green box at the bottom states: "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

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

```
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>

