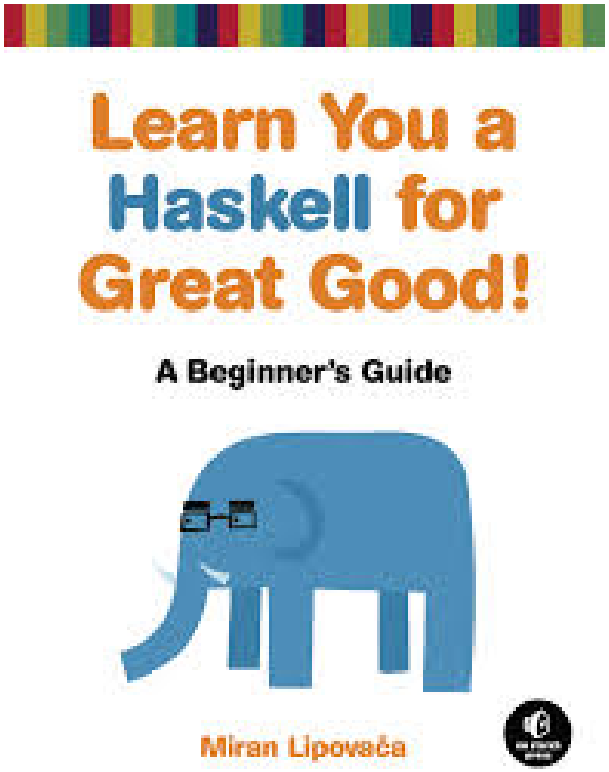
Learning Haskell

1. Using `ghci`. Interactivity, directives.
2. Expressions for each of the basic data types: `Integer`, `Float`, `Bool`, `Char`, `()` — unit.
3. Tuples. Lists: arithmetic sequences, list comprehensions.
4. Bindings. The `let` expression.
5. Simple functions. Anonymous functions, functions as data, special declaration syntax, patterns, cases, guards.
6. Using `ghc`. Writing, compiling, and running a main program, `interact`.

Erik Meijer, OSCON '09 [14min] YouTube
Simon Peyton Jones, POP 2003, "Retrospective" [ppt]

Learning Haskell

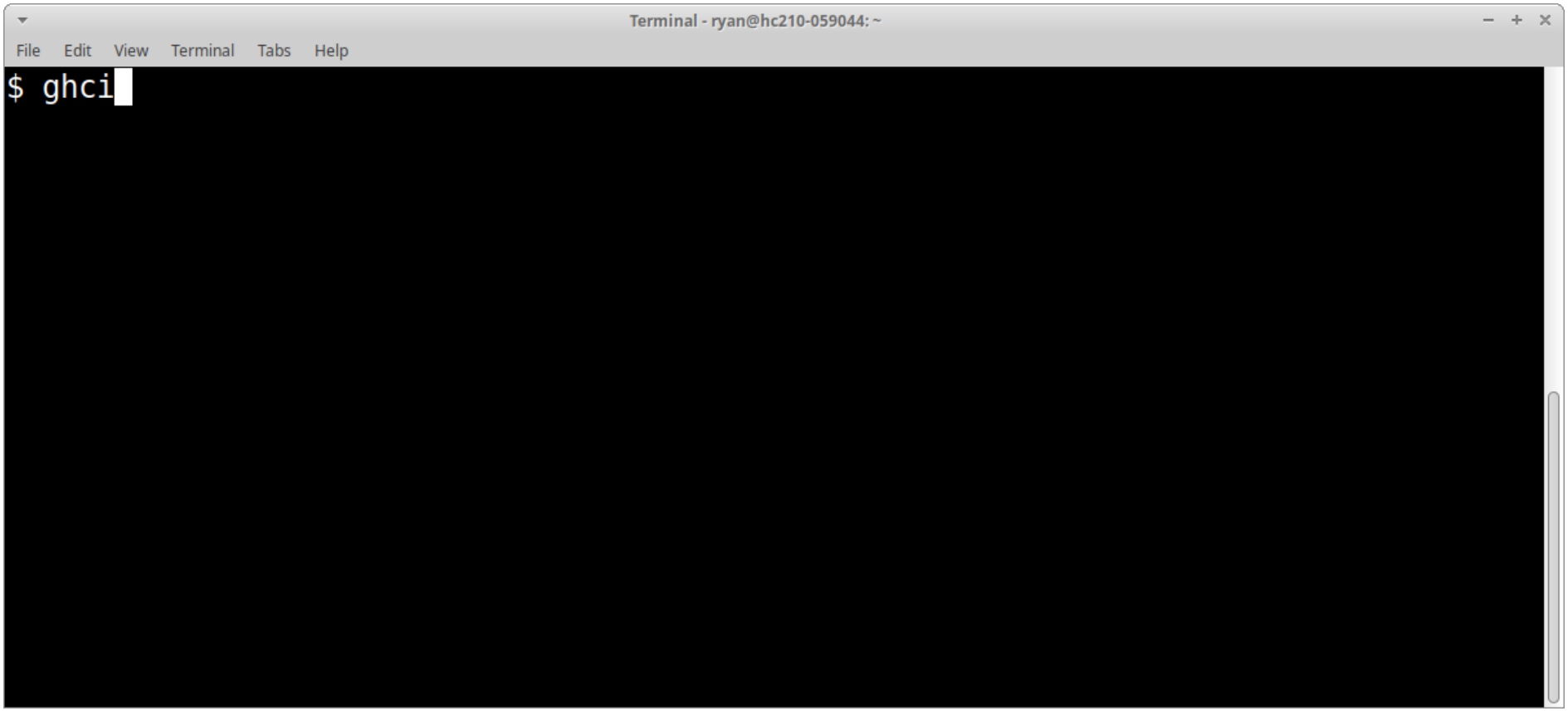
- All things Haskell: `haskell.org`
- Tutorial: *Learn You a Haskell for Great Good!* by Miran Lipovača
- Searching for functions: Hoogle



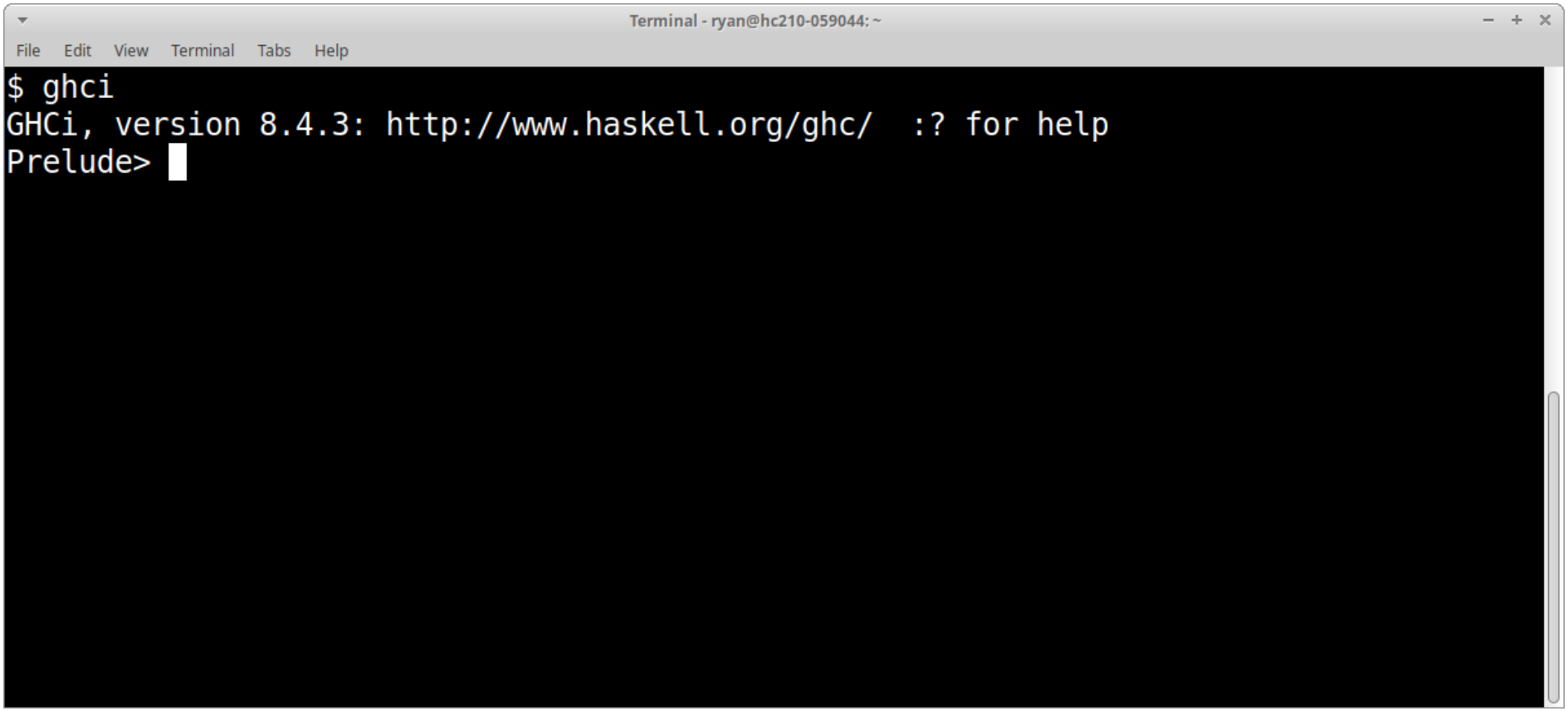
Learning Haskell

- Higher-order functions
- Data structures
- Type inference

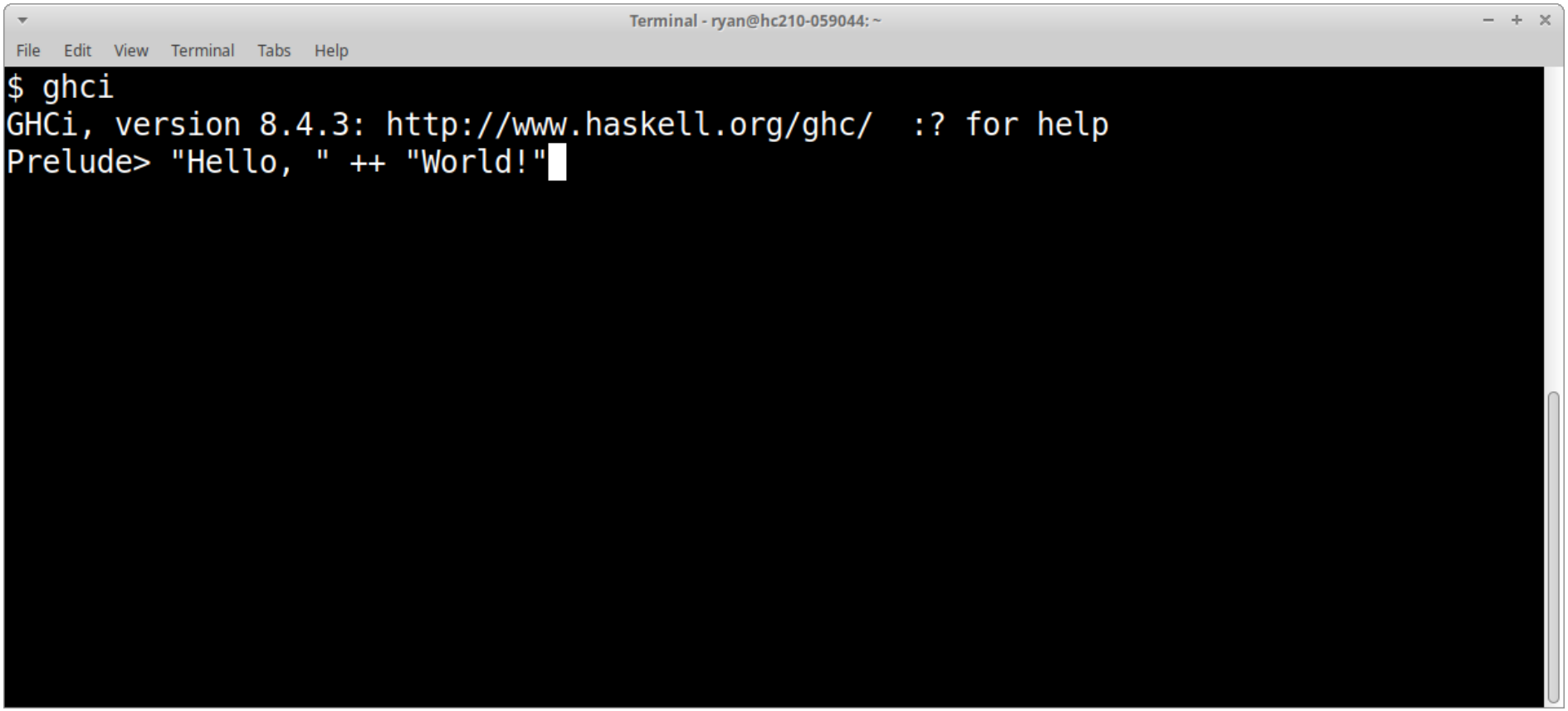
Using GHCi


Terminal - ryan@hc210-059044: ~
File Edit View Terminal Tabs Help
$ ghci

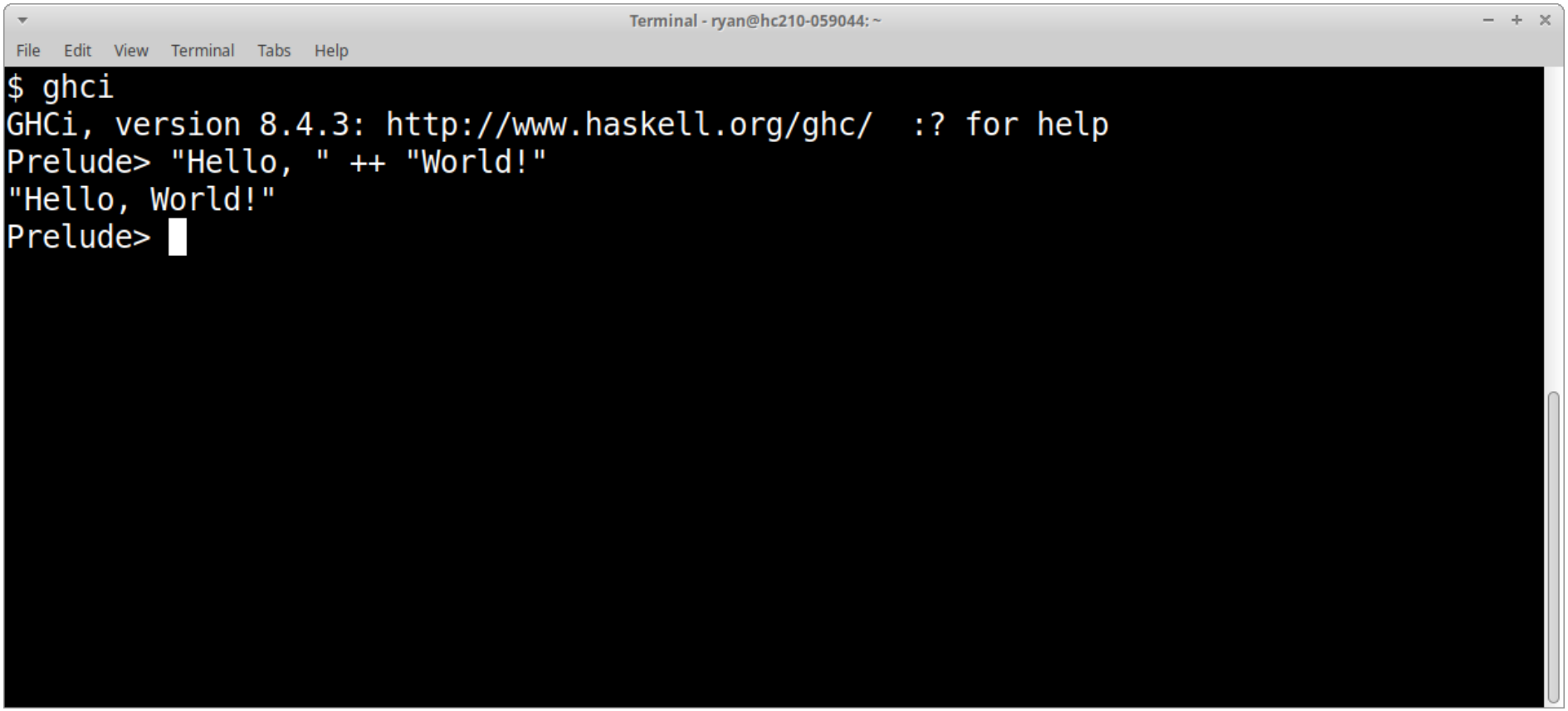
Using GHCi


Terminal - ryan@hc210-059044: ~
File Edit View Terminal Tabs Help
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Prelude>

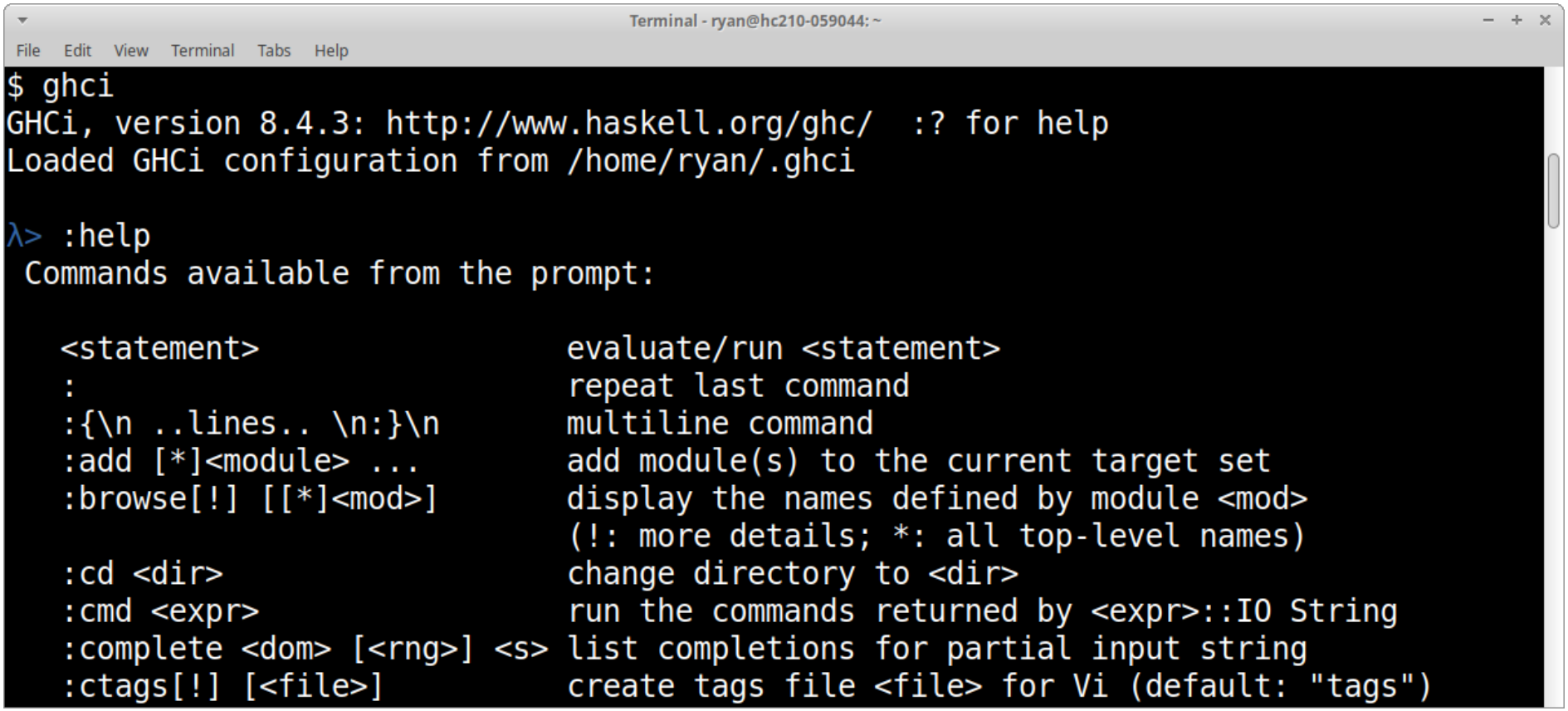
Using GHCi


Terminal - ryan@hc210-059044: ~
File Edit View Terminal Tabs Help
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Prelude> "Hello, " ++ "World!"

Using GHCi


Terminal - ryan@hc210-059044: ~
File Edit View Terminal Tabs Help
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/ :? for help
Prelude> "Hello, " ++ "World!"
"Hello, World!"
Prelude>

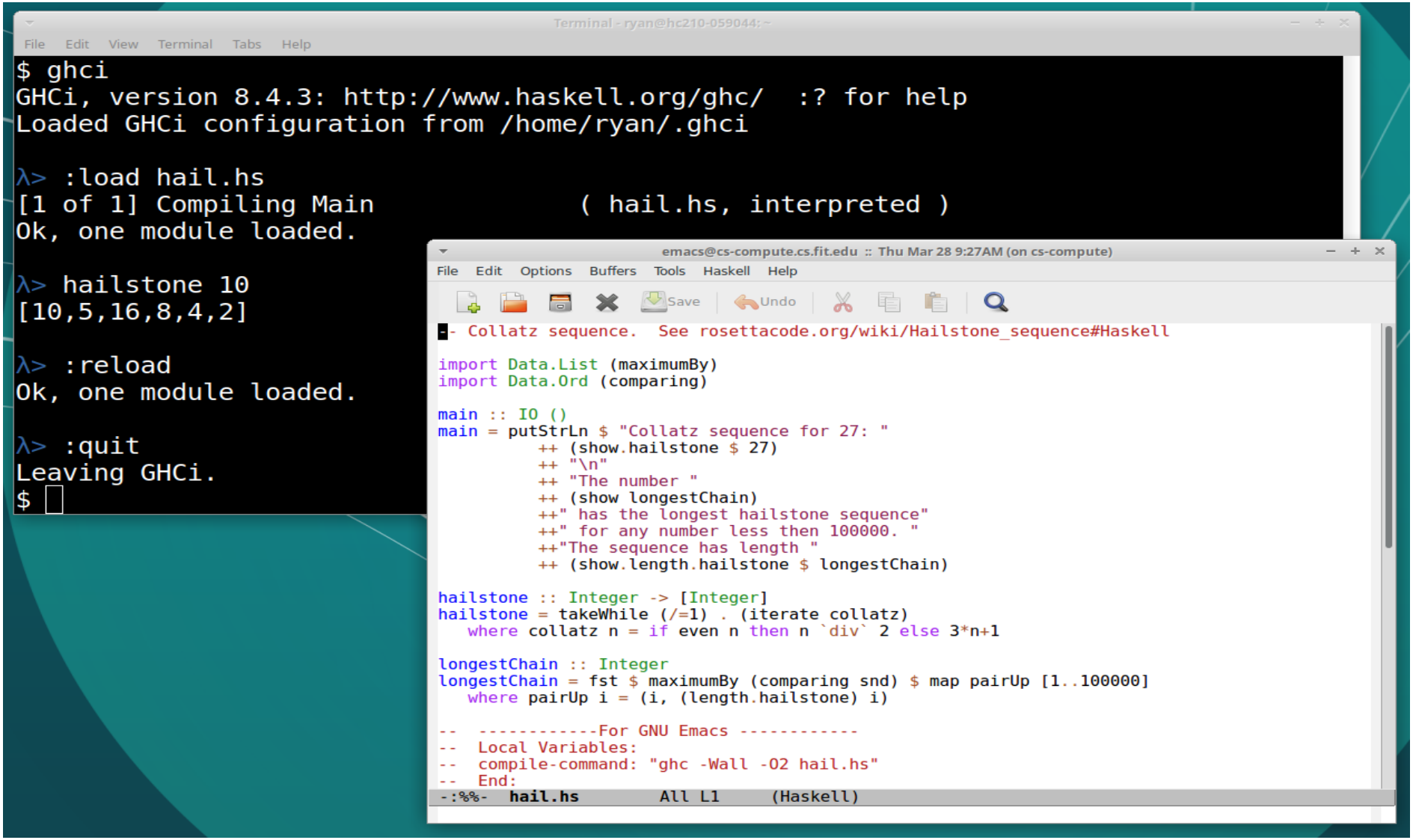
GHCI directives



GHCI directives

```
<statement>                    evaluate/run <statement>
:                              repeat last command
:{\n ..lines.. \n:}\n          multiline command
:add [*]<module> ...           add module(s) to the current target set
:browse[!] [[*]<mod>]          display the names defined by module <mod>
                               (!: more details; *: all top-level names)
:cd <dir>                      change directory to <dir>
:complete <dom> [<rng>] <s>    list completions for partial input string
:help, :?                      display this list of commands
:info[<name> ...]              display information about the given names
:kind <type>                   show the kind of <type>
:load    [*]<module> ...       load module(s) and their dependents
:main [<arguments> ...]        run the main function with the given arguments
:module [+/-] [*]<mod> ...     set the context for expression evaluation
:quit                          exit GHCi
:reload                        reload the current module set
:type <expr>                   show the type of <expr>
:type +d <expr>                show the type of <expr>, defaulting type variab
```

Interactive development



load — edit — reload – test; and repeat

Overview to skim quickly, or
Summary for later

Data

Integers

$$\langle integer \rangle ::= \langle digit \rangle^{+}$$

Lists

$$\langle list \rangle ::= []$$

$$::= \langle expr \rangle : \langle list \rangle$$

Tuples

$$\langle tuple \rangle ::= (\langle expr \rangle , \langle expr \rangle)$$

$$::= (\langle expr \rangle , \langle expr \rangle , \langle expr \rangle)$$

$$::= (\langle expr \rangle , \langle expr \rangle , \langle expr \rangle , \langle expr \rangle)$$

Functions

$$\langle function \rangle ::= \backslash \langle name \rangle \texttt{->} \langle expr \rangle$$

Integers

$$\langle integer \rangle ::= \langle digit \rangle^{+}$$

Some Integers

The Type of Integers

Integer

Integer

Integer

Integer

Integer

Integer

Tuples

$$\langle tuple \rangle ::= (\langle expr \rangle, \langle expr \rangle)$$

$$::= (\langle expr \rangle, \langle expr \rangle, \langle expr \rangle)$$

$$::= (\langle expr \rangle, \langle expr \rangle, \langle expr \rangle, \langle expr \rangle)$$

```
(5,3+4)
('z',1,1,1)
(True,2,False,3)
('a',True)
(4*7,('a',True),'p')
(2+3,2*3,2-3)
```

```
(Integer,Integer)
(Char,Integer,Integer,Integer)
(Bool,Integer,Bool,Integer)
(Char,Bool)
(Integer,(Char,Bool),Char)
(Integer,Integer,Integer)
```

Lists

$$\langle list \rangle ::= []$$

$$::= \langle expr \rangle : \langle list \rangle$$

```
1:1:[]
'a':'b':'c':[]
():():[]
3:5:(3+6):7:[]
False:False:True:[]
True:False:True:[]
```

[Integer]

[Char] = String

[()]

[Integer]

[Bool]

[Bool]

Lists Have Special Syntax

```
[1,1]

['a','b','c'] = "abc"

[(),()]

[3,5,3+6,7]

[False,False,True]

[True,False,True]
```

Functions

$$\langle function \rangle ::= \backslash \langle name \rangle \texttt{->} \langle expr \rangle$$

lambda
formal parameter
body

```
                                \xs ->2:xs
      \i ->max i 0

                                      \s ->s++", "++s

                          \i ->i+2

\x ->sin (x+pi)

                                \c ->ord c
```

```
[Integer]->[Integer]
```

```
Integer->Integer
```

```
String->String
```

```
Integer->Integer
```

```
Double->Double
```

```
Char->Integer
```

Data, all data exists apart from names. Data

```
meaningOfLive = 42

myVerySpecialPair = ('a', 0)

myShortList = [True,False,True]

add2 = \ i -> i+2

cons2 = \ xs -> 2:xs

double = \ s -> s ++ ", " ++ s
```

Function Declarations Have Special Syntax

```
cons2 xs = 2:xs

cutOff i = max i 0

doubleWord s = s ++ ", " ++ s

add2 i = i+2

f x = sin (x+pi)

toOrd c = ord c
```

Complex Canonical Values

```
(2,('a',5), "abc", True)  :: (Integer, (Char,Intege
([1],('a',5), "abc", True, "xyz")  :: ([Integer], (
( \i->i+2 , \x->sind(x+pi) ) :: (Integer->Integer,

[(1,True,"abc"), (2,True,"mno"), (3,True,"xyz")] ::

\ i -> (i,i)  :: Integer -> (Integer,Integer)
\ p -> (fst p + snd p, fst p * snd p, fst p - snd p
```

End of the quick overview
(proceed to basics), or
End of summary
(proceed to writing a program with GHC)

The Basics

1. Primitive: Integer, floating-point, character, boolean, unit
2. Structures: Tuple, lists, functions
3. Text: list of character

Integer

```
143
succ 34
3+4
5*7
9-4
negate 8

---- watch out: 3 * -8  NO!
3 * (-8)
```

Integer

```
min 17 34
max 17 34

div 24 7
24 `div` 7
24 `rem` 7
mod 36 5
   ---> 1

quot 36 5
   ---> 7
div 36 5
   ---> 7
quot 36 (-5)
   ---> -7
div 36 (-5)
   ---> -8

      36 5
```

Integer

```
quotRem 36 (-5)
  ---> (-7,1)

divMod 36 -5
  ---> (-8,-4)
```

Floating-Point

```
--- pi,exp,sqrt,log,(**),sin,tan,cos,
--- truncate, ceiling, round, truncate, floor
3.2 + 43.1
5.2 * 43.2345236
9.9 - 2.3
2345.2345 / 34.34
34.4 ^ 34
4254 ** 4.345
sqrt (4.59)
sin (1.7172)
```

Char

```
'a'

--- :browse Data.Char
ord :: Char -> Int
chr :: Int -> Char
digitToInt :: Char -> Int
intToDigit :: Int -> Char
toLower :: Char -> Char
toUpper :: Char -> Char
isAlphaNum :: Char -> Bool
isDigit :: Char -> Bool
isAlpha :: Char -> Bool
isLower :: Char -> Bool
isUpper :: Char -> Bool
isSpace :: Char -> Bool
```

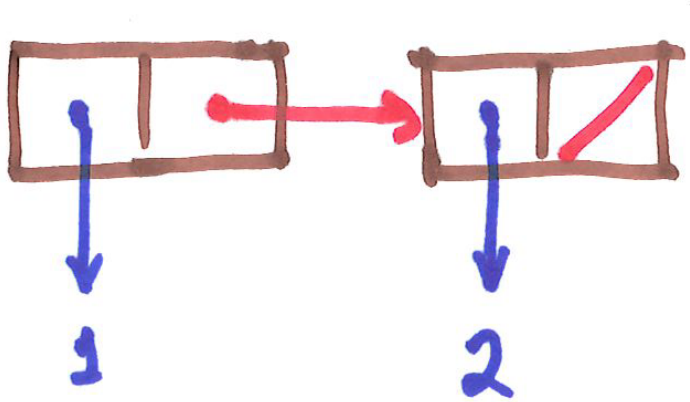
Bool

```
False
True
False && True
False || True
not True
-- otherwise is defined to be the value True
-- for the purposes of making guards more readable
otherwise

if True then 4 else 5

-- important predicates
-- (==), (/=), (<), (>=), (>), (<=), compare
```

Tuples

```
(2'a')
(4.34, 'a', 456)
(True, (), 1)
(2, "ab", 2*2, 5.0)
( 2 ) -- not a tuple
((())) -- unit

fst (2,'b')
snd (1,'a')

(2,('a',3,4),"abcd")
((2,3,4), (True, 3.3))
```

What is a List?



How do you make a List in Haskell?

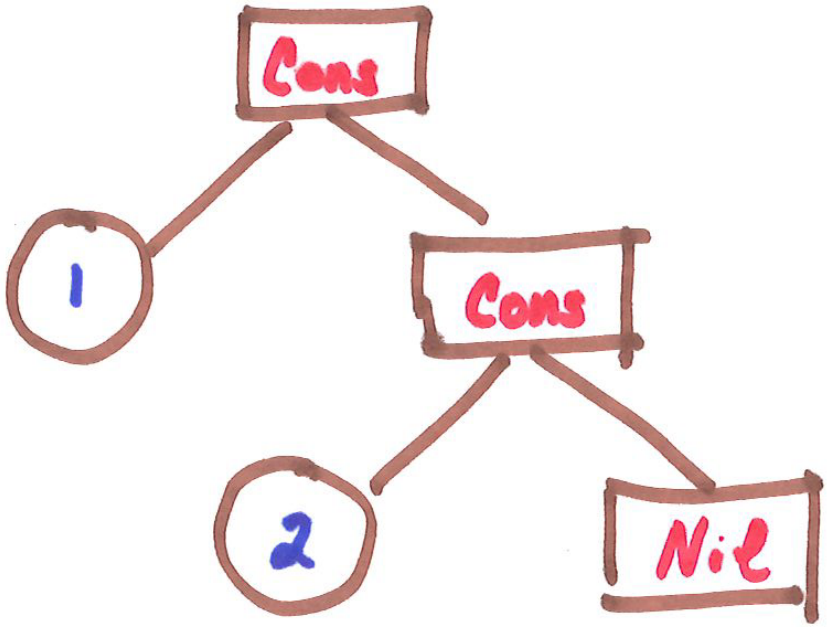
Two constructors “nil” and “cons”

```
[]

1:[]

1 : (2 : [])

1 : (2 : (3 : []))

1 : (2 : (3 : 4 : []))
```

How do you make a List in Haskell?

"Cons" is right associative; and, anyway, lists have special syntax.

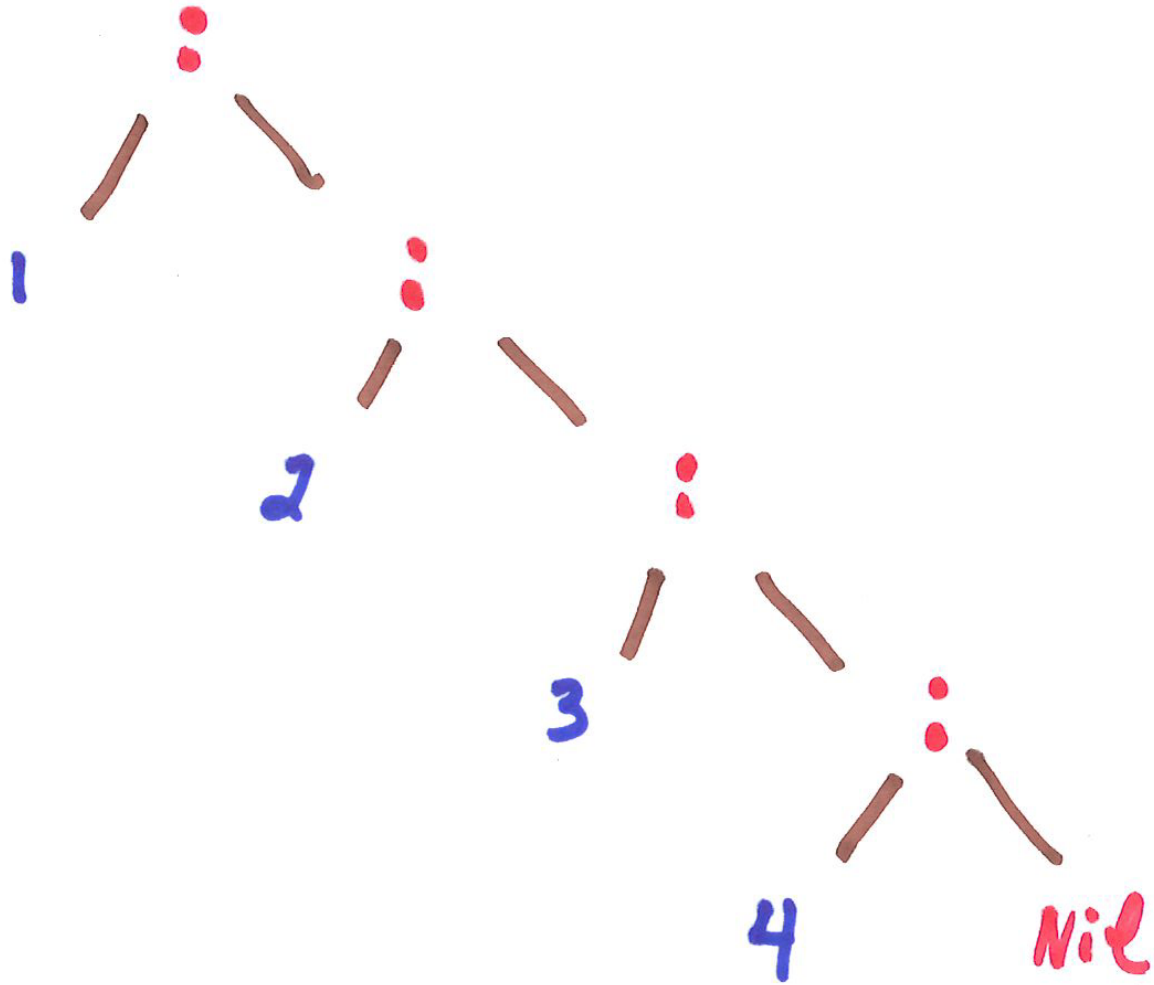
```
[]                          []                      []
1:[]                        1:[]                    [1]
1 : (2 : [])                1:2:[]                  [1,2]
1 : (2 : (3 : []))          1:2:3:[]                [1,2,3]
1 : (2 : (3 : 4 : []))      1:2:3:4:[]              [1,2,3,4]
```



Cons 1 (cons 2 Nil)

1 : 2 : Nil

[1, 2]



[1, 2, 3, 4]

What can you do with a list?

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

tail [1,2,3,4]

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

Lists are polymorphic but homogeneous.

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

['a', 'b', 'c']

[(1,'a'), (2,'b'), (3,'c')]

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

There are no mutable arrays in Haskell, so lists are the "go to" data structure for collections.

Every list function one can think up has been pre-defined in Haskell. See lists at Wiki Haskell.

```
length
(++)    -- append
elem    -- member
(!!)    -- get element
concat  -- flatten
last init
splitAt
take drop
sort nub
reverse
sum product
minimum maximum
and or
all any
```

List and Arithmetic Sequences

```
[-1,-1 .. 0]
[3,2 .. 8]
[0,2 .. 1]
[1,1 .. 1]
[3,3 .. (-4)]
[2,1 .. (-4)]
[3,3 ..(-12)]
[-1,-1 ..     8]
[-6,-6 ..12]
[1,2..(-12)]
[-6,-5 .. (-4)]
[1,0 .. 5]
[-6,-6 .. 8]
[-1,-2 .. ( -4)]
[1,2 .. 5]
```

List Comprehension

Video; 31 minutes
Presenter: E Meijer
based on Hutton's book, chapter 5
Channel 9 lectures at YouTube

List Comprehension

Defining sets by their properties is sometimes known as set comprehension and found often in mathematical texts. For example,

$$\{x \in \mathbb{R} \mid x > 0\}$$

In mathematical notation we write

$$\{\, x^2 \mid x \in \{1, 2, \ldots, 5\} \,\}$$

to mean the set $\{1, 4, 9, 16, 25\}$.

In Haskell a similar syntax is used for lists. The special square brackets syntax is expanded to include generators and filters.

Haskell List Comprehension

A list comprehension has the form:

$$[\, expr \mid qual_1 \, , \ldots , \, qual_n \,]$$

where $1 \geq n$

There are three types qualifiers

- generators of the form *pat*<-*expr*, where p is a pattern (see Section 3.17) of type t and e is an expression of type [t]
- local bindings that provide new definitions for use in the generated expression *expr* or subsequent boolean guards and generators
- boolean guards, which are arbitrary expressions of type `Bool`.

See the section on List Comprehensions in the Haskell 2010 Language Report.

Also, there are intriguing language extensions:

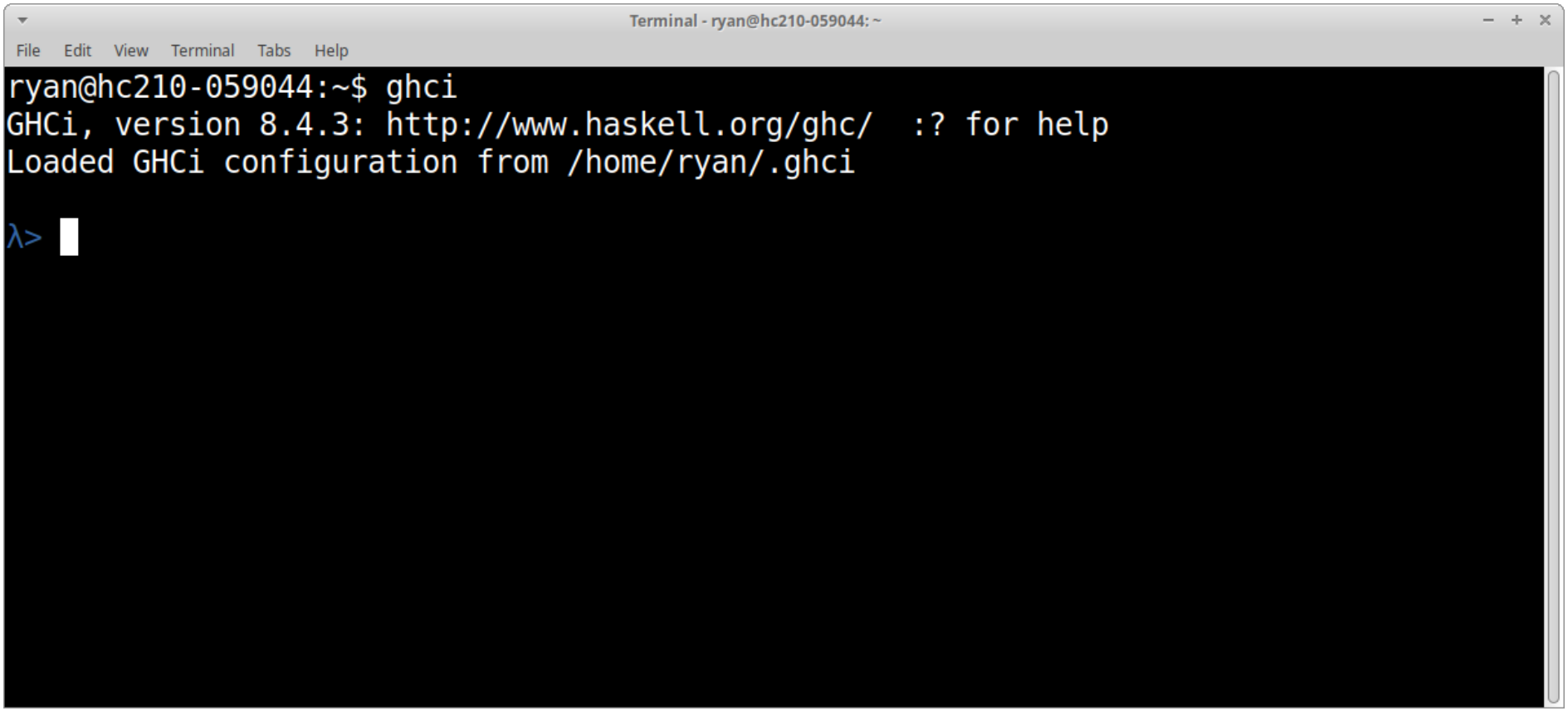
- ParallelListComp. Syntax: `[ expr | qualifier, ... | qualifier, ... ]`
- TransformListComp. Three new keywords: group, by, and using.

See a paper by Simon Peyton Jones.

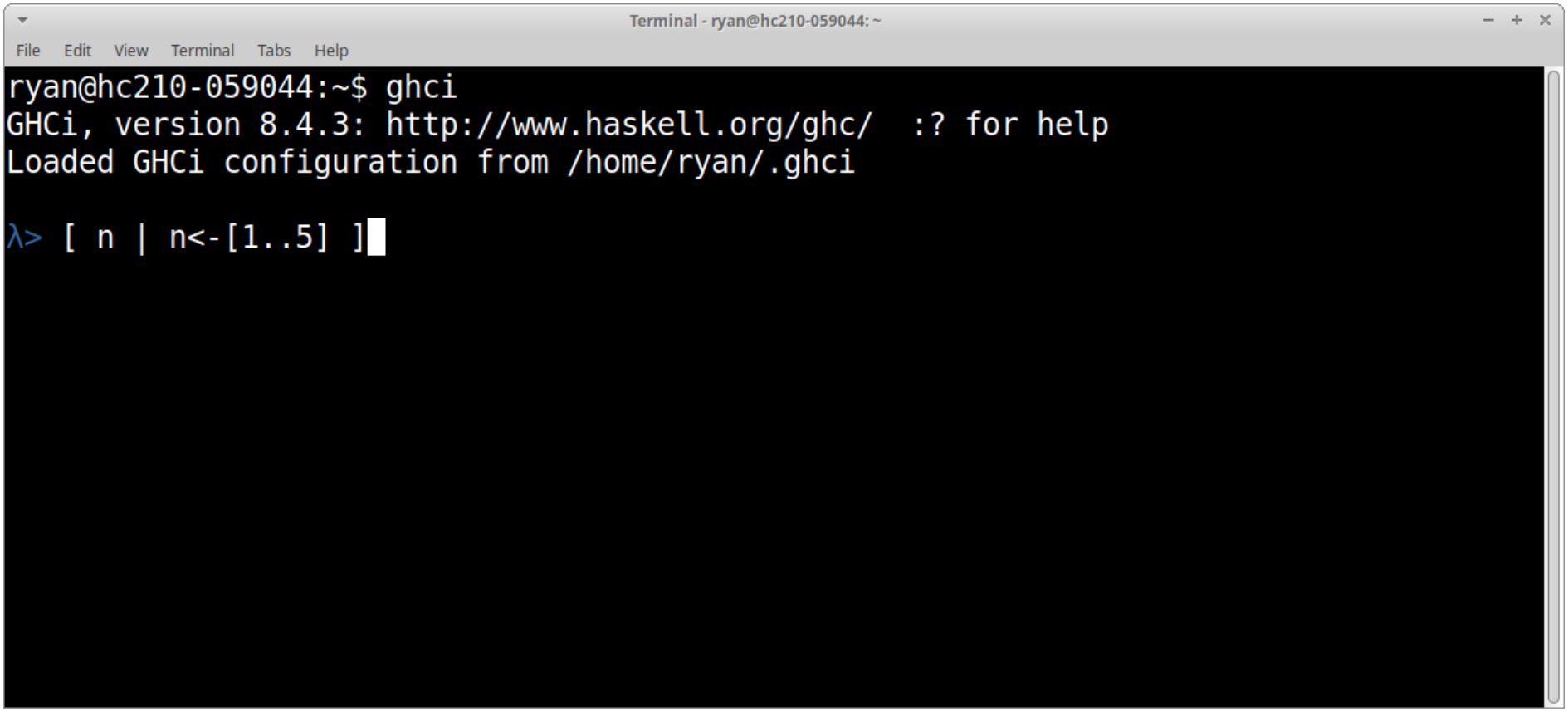
Upcoming script

```
[ n | n <- [1..5] ]
[ (n+2,5*n) | n <-[1..3] ]    -- list of pairs
:mod + Data.Char
[ toUpper c | c <- "one fish" ]
```

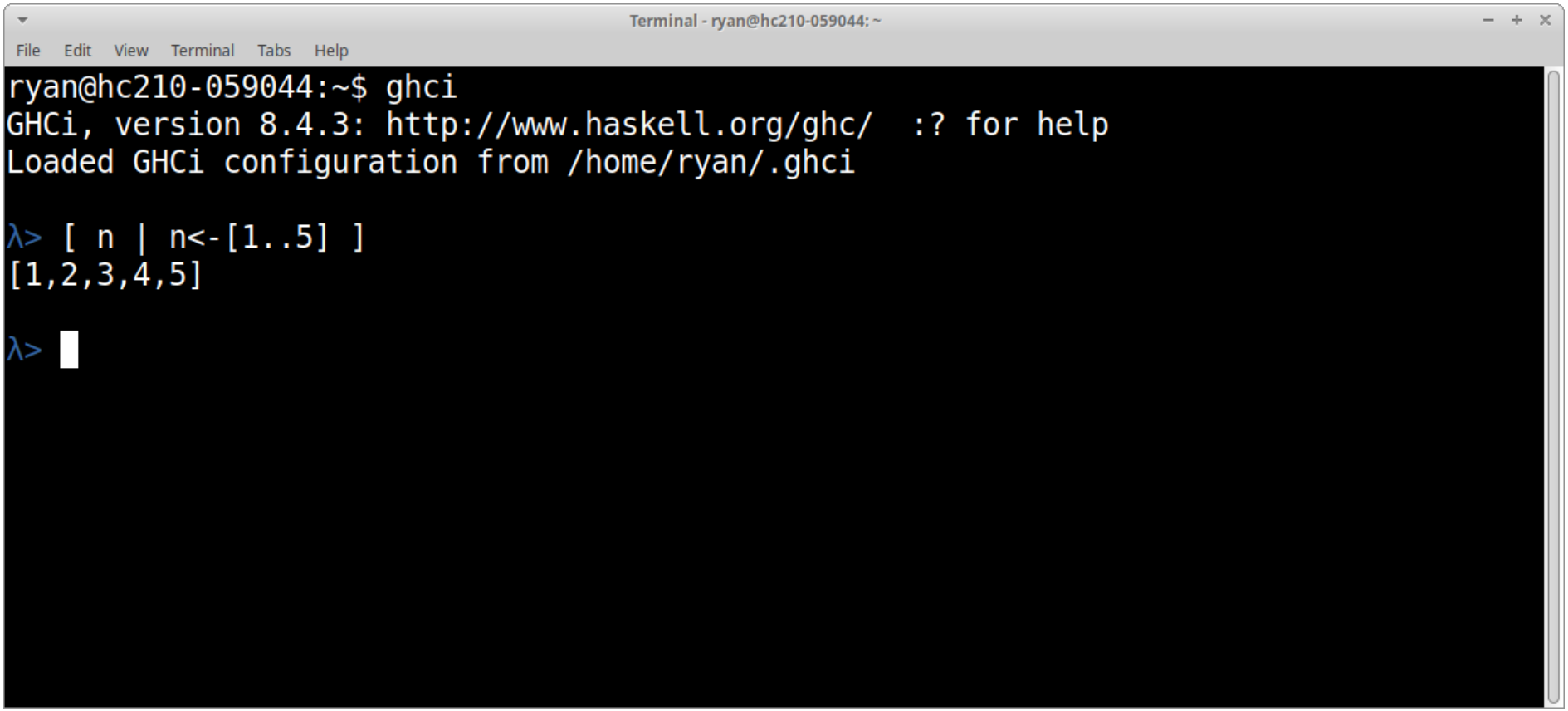
List Comprehension



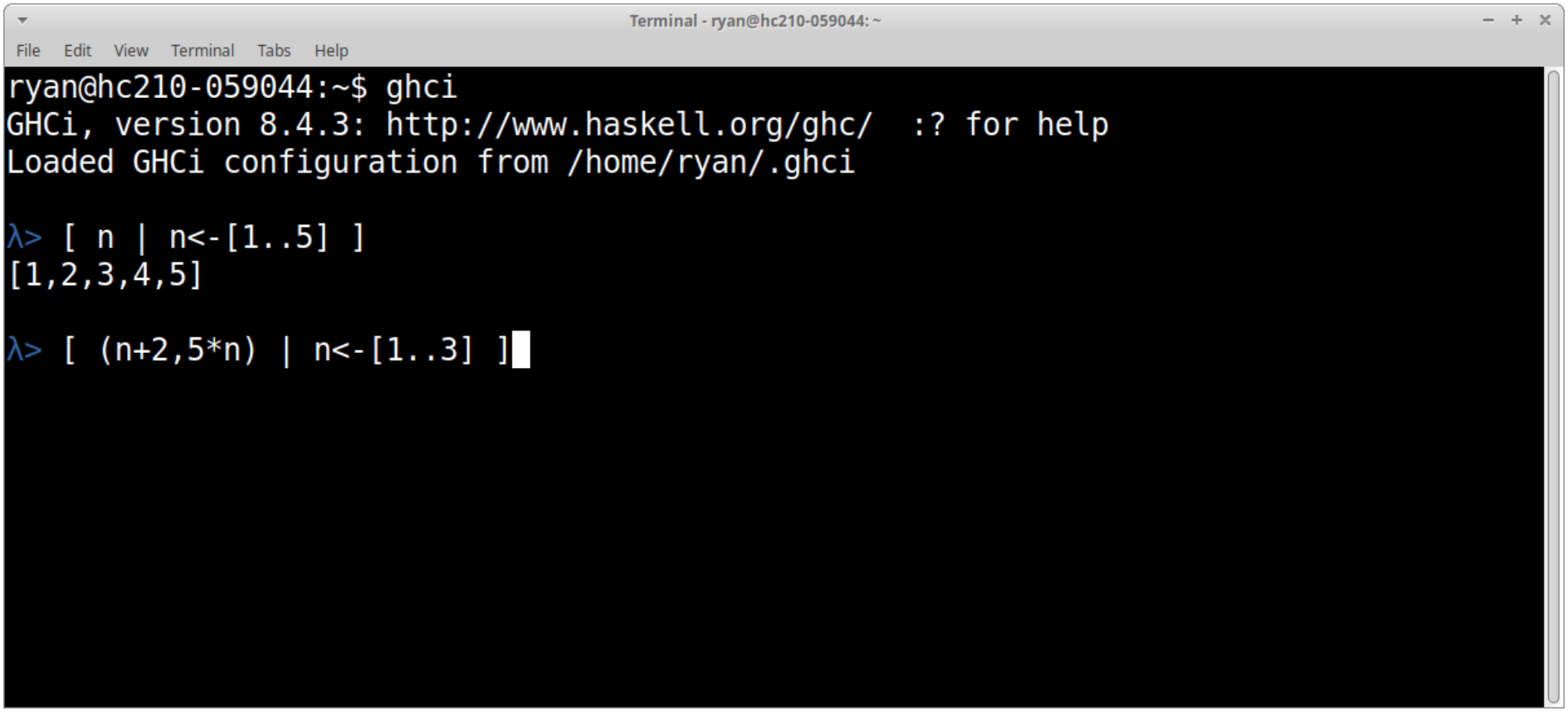
List Comprehension



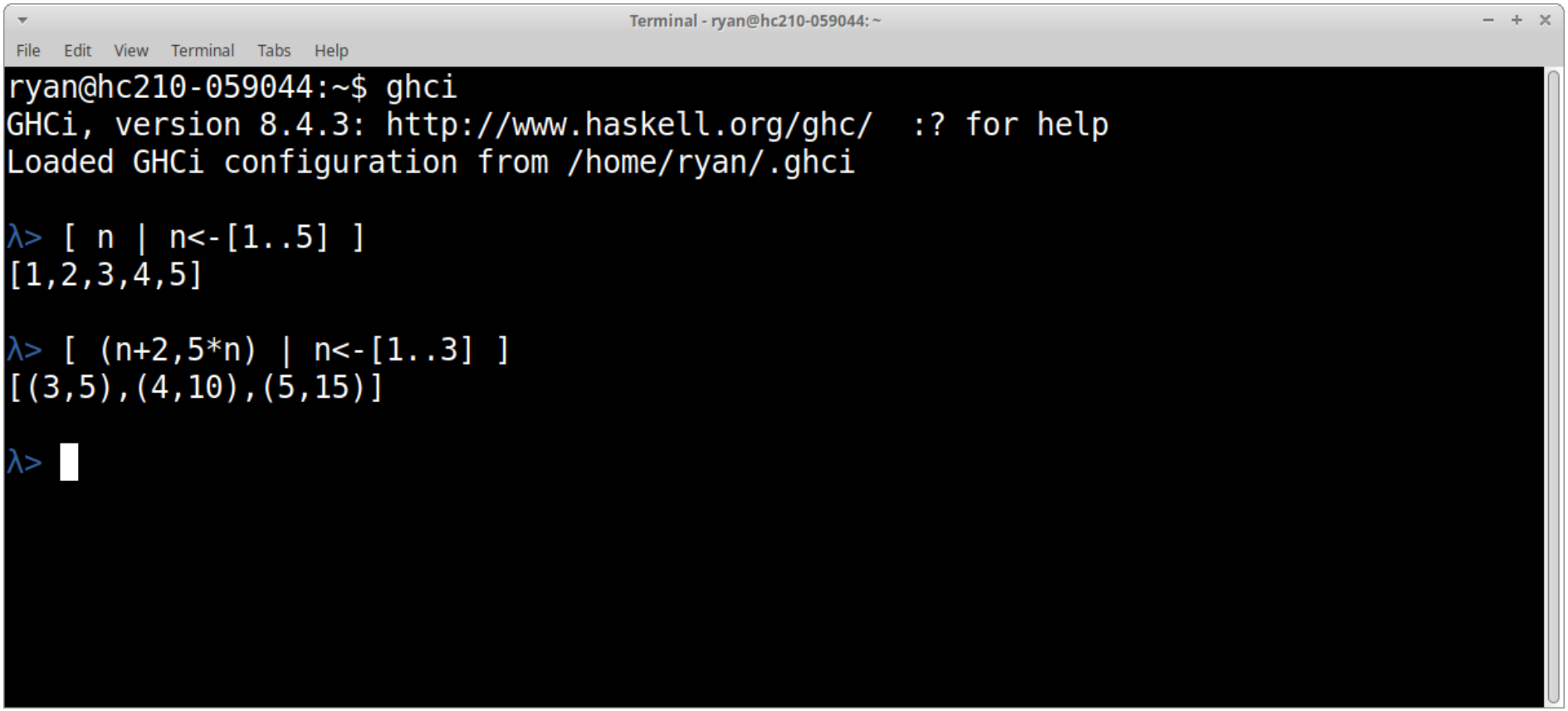
List Comprehension



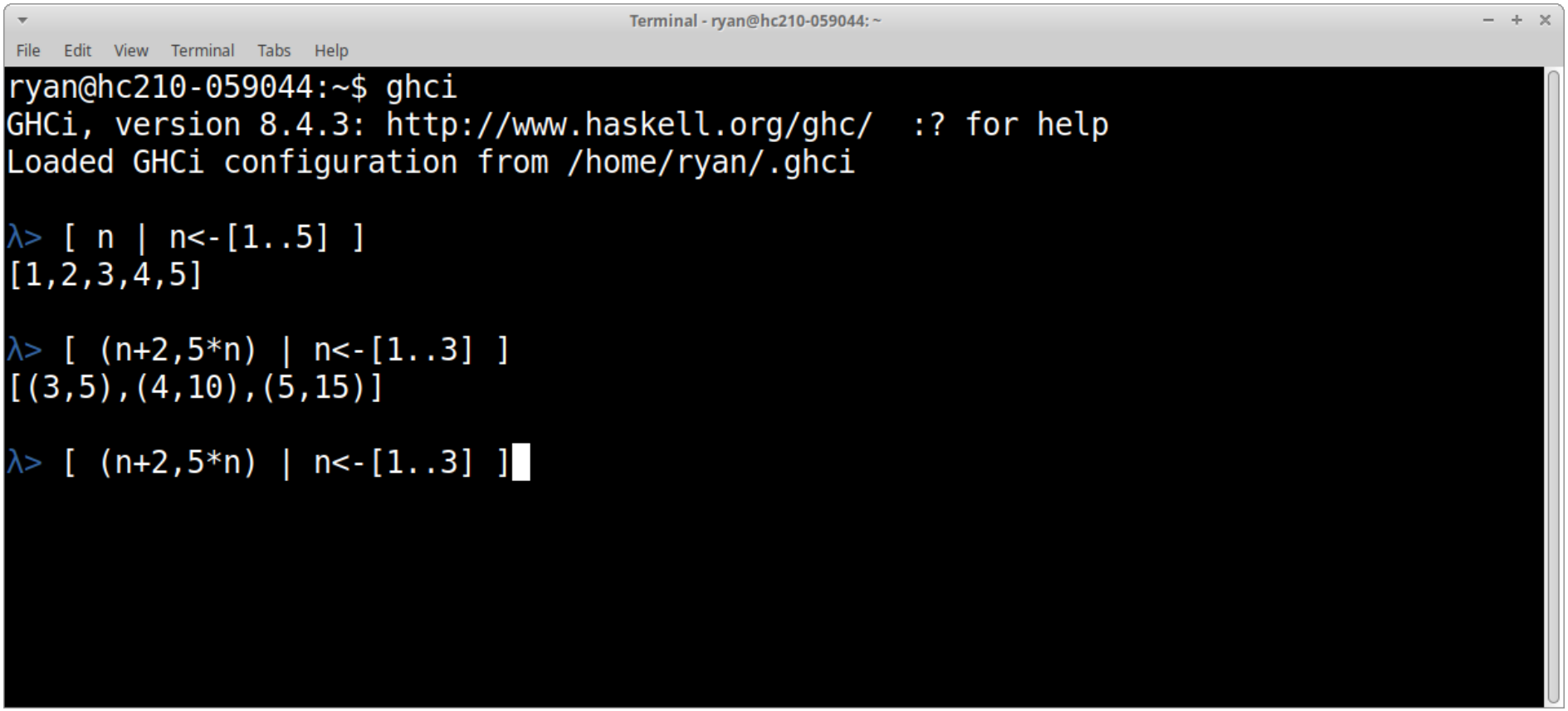
List Comprehension



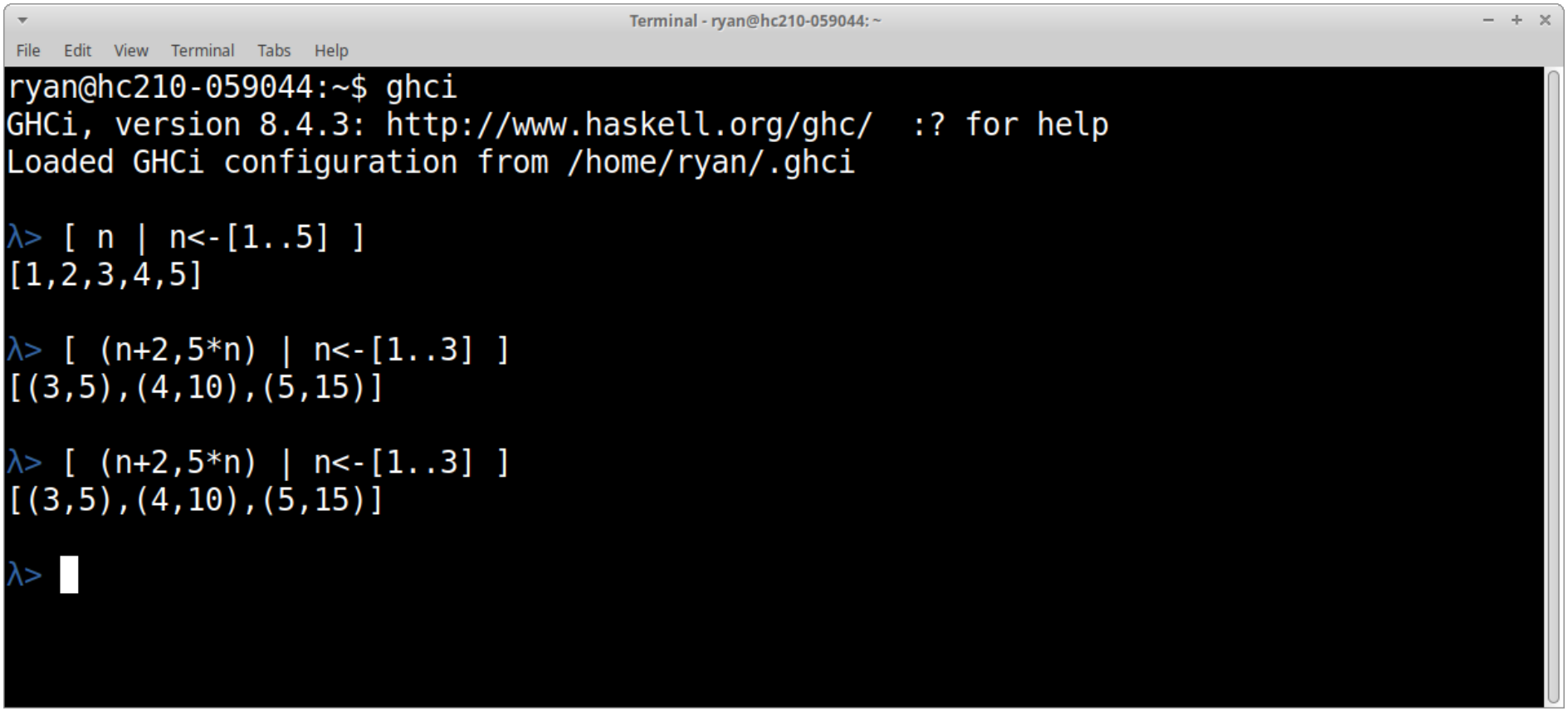
List Comprehension



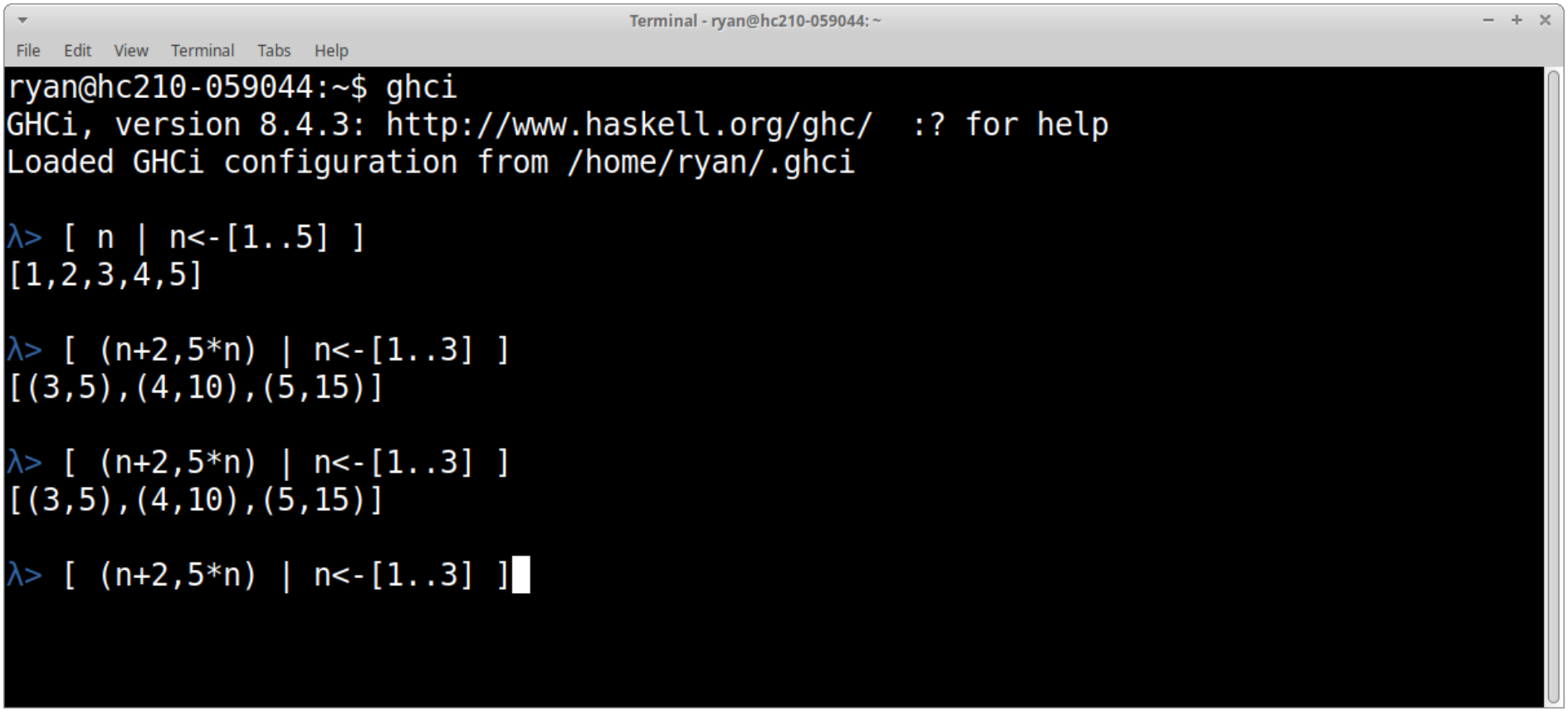
List Comprehension



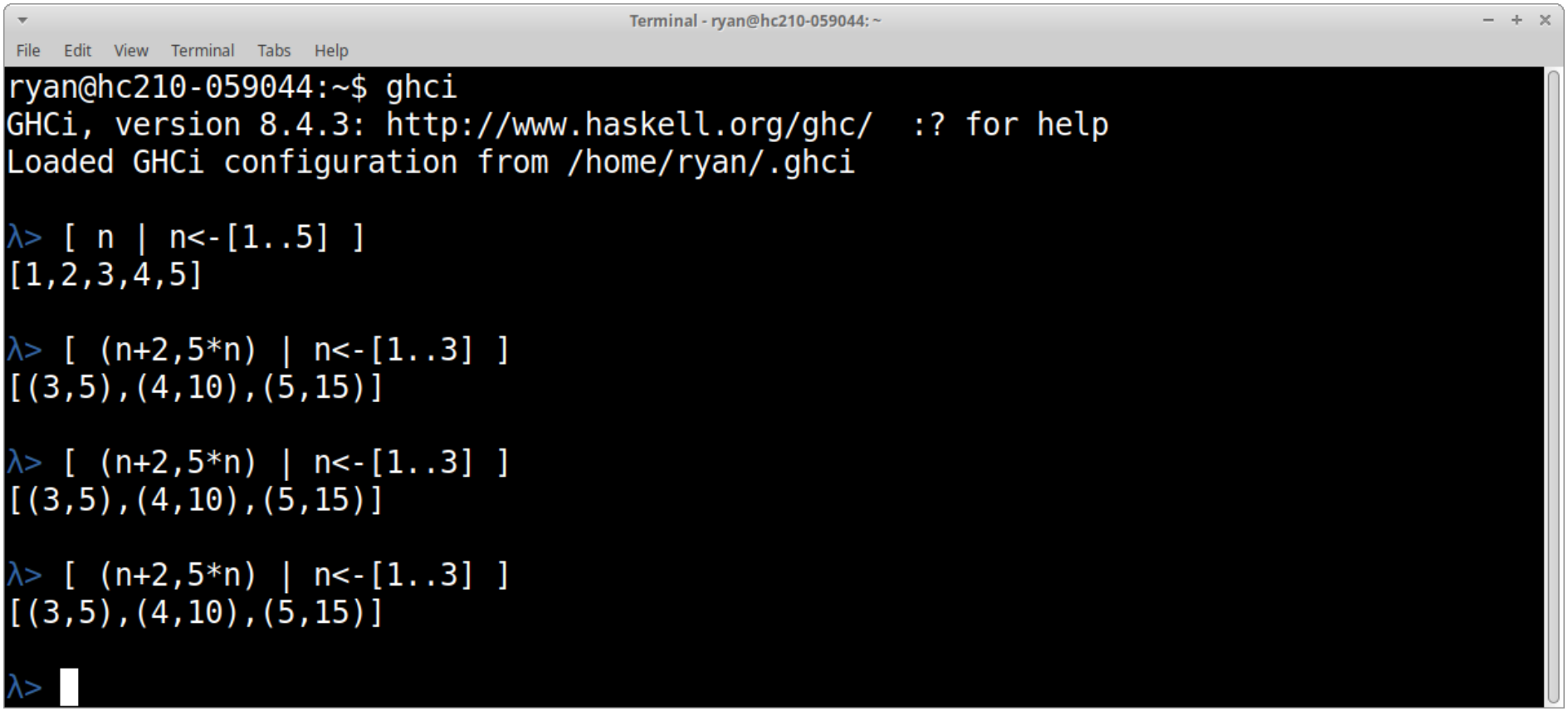
List Comprehension



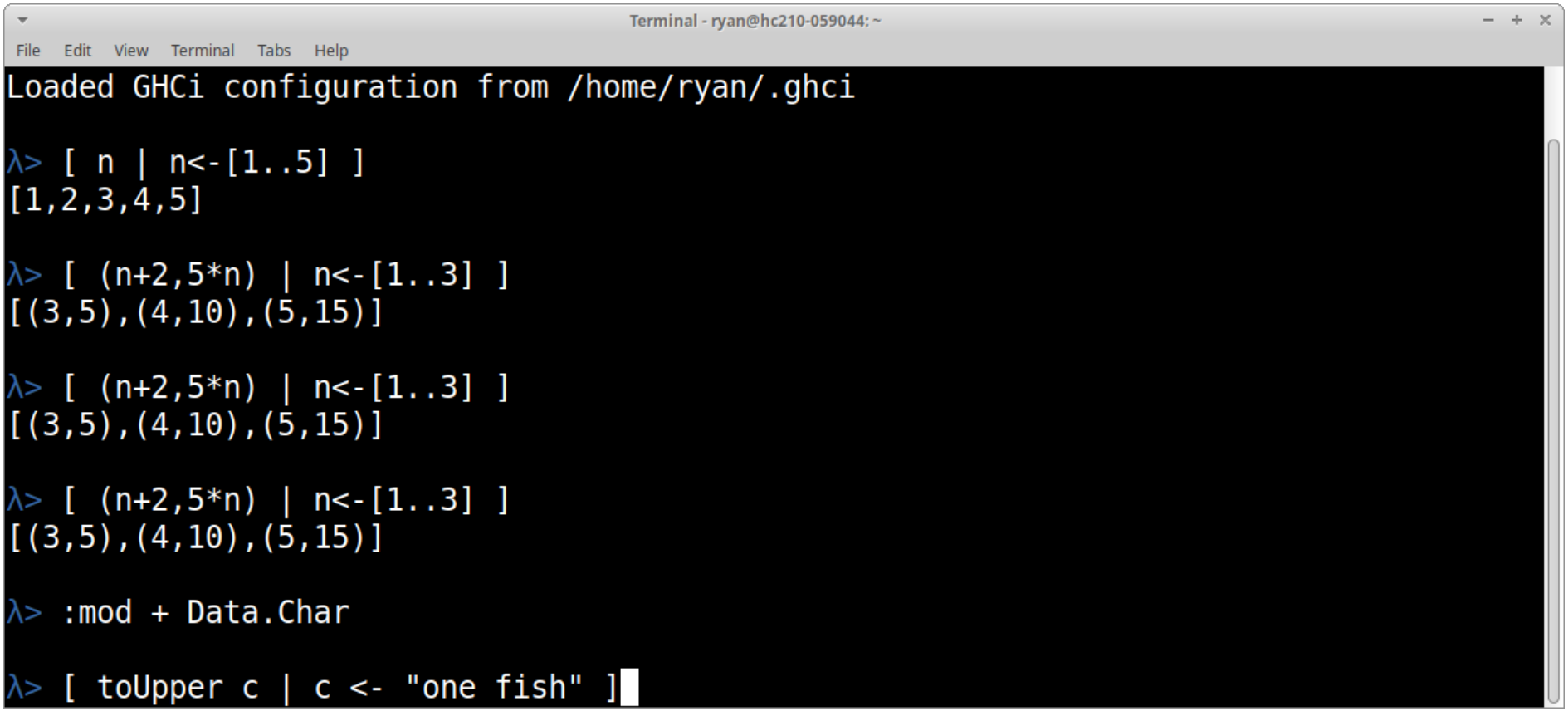
List Comprehension



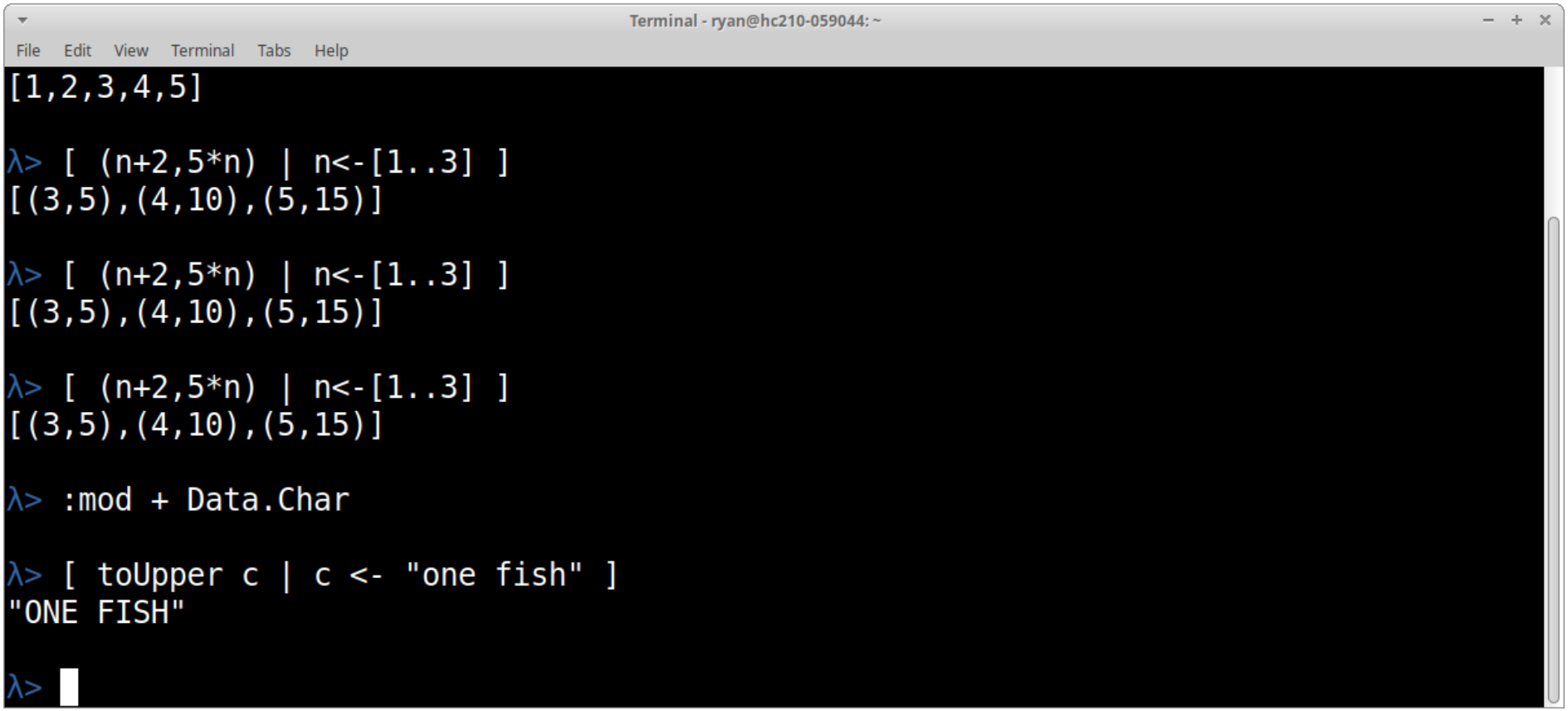
List Comprehension



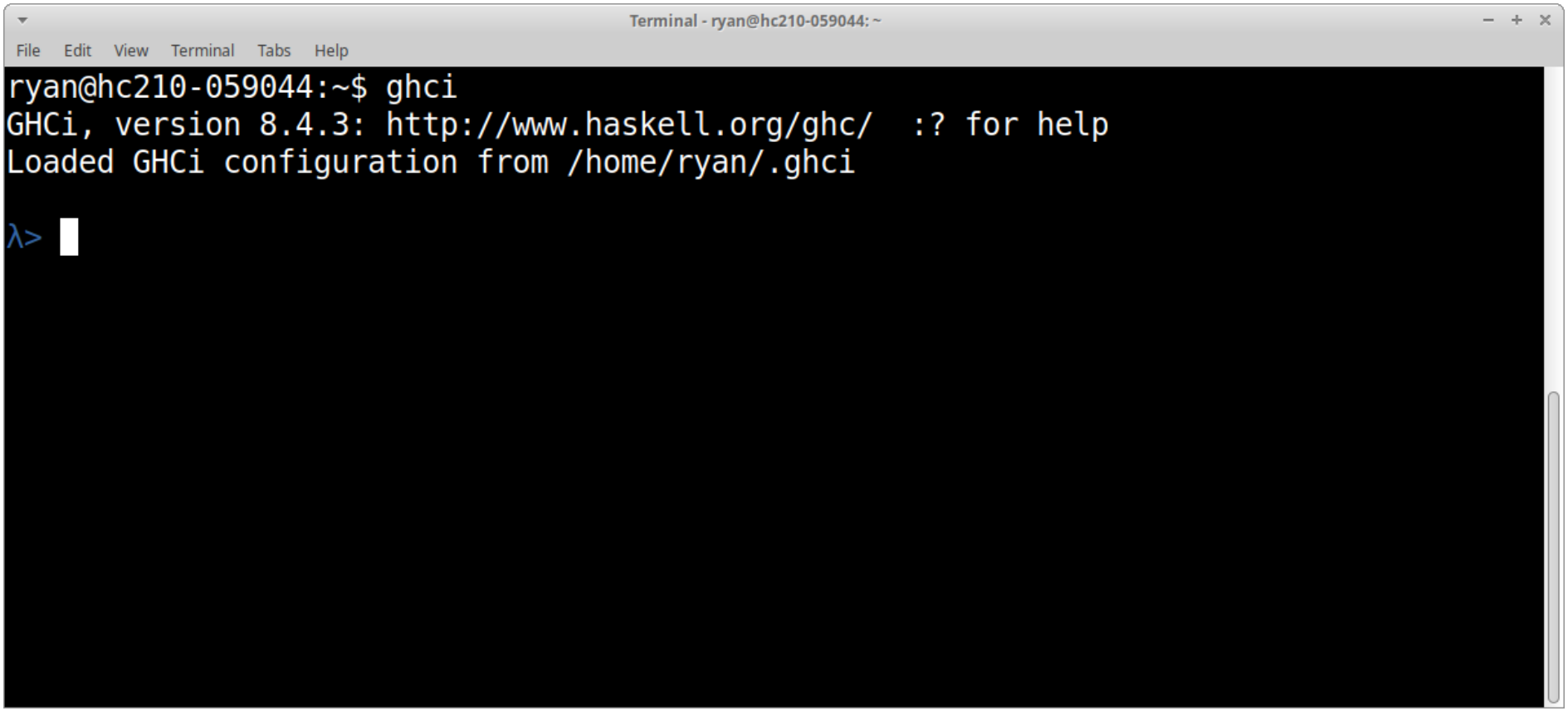
List Comprehension



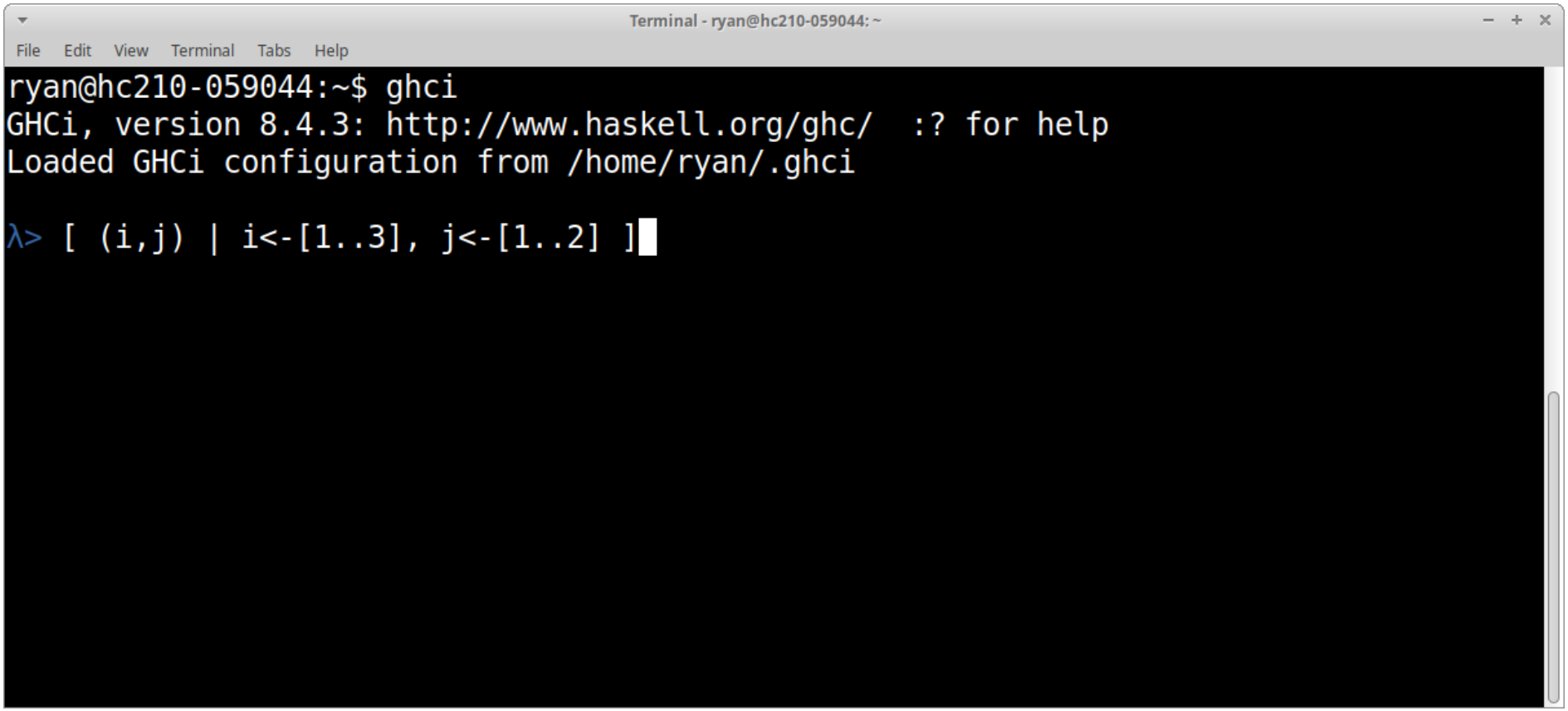
List Comprehension



List Comprehension (Multiple Generators)

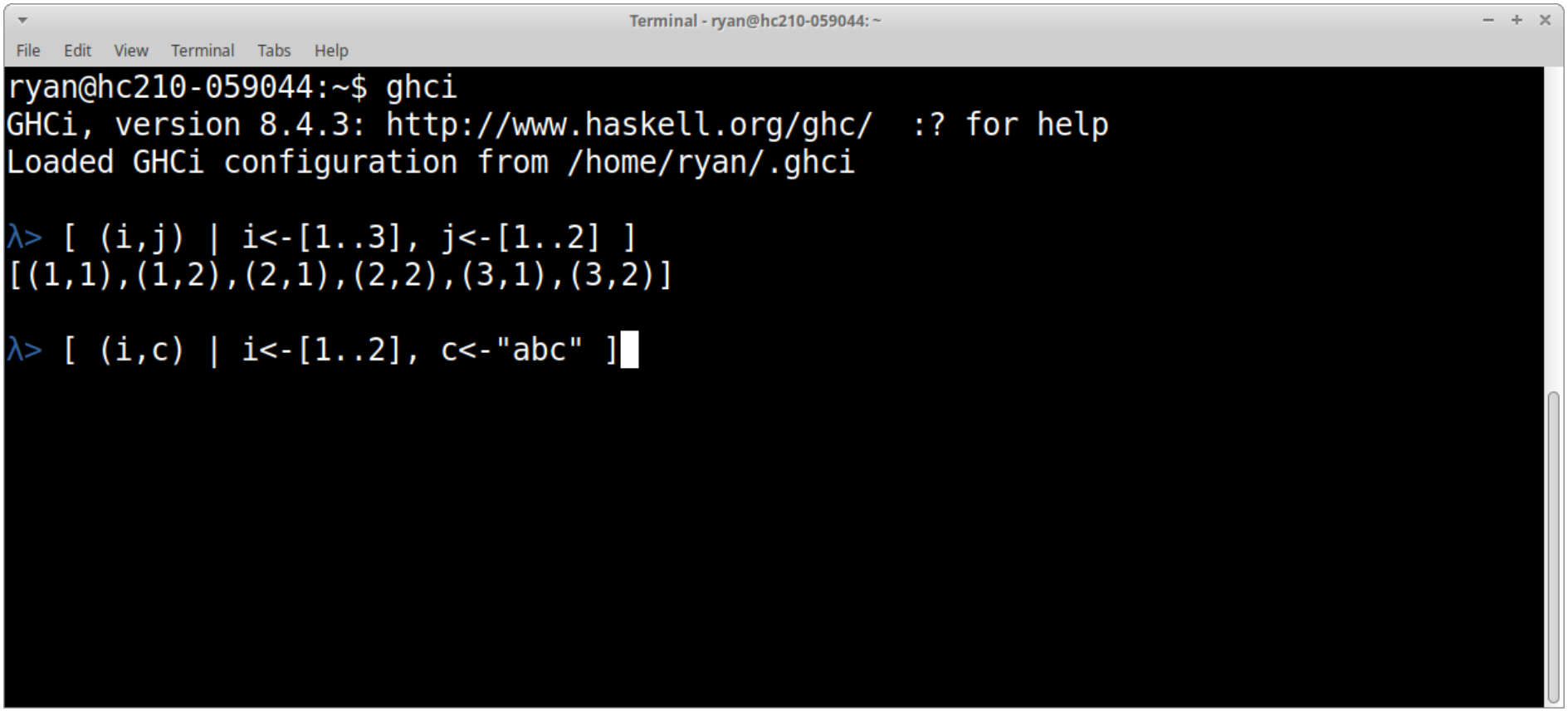


List Comprehension (Multiple Generators)

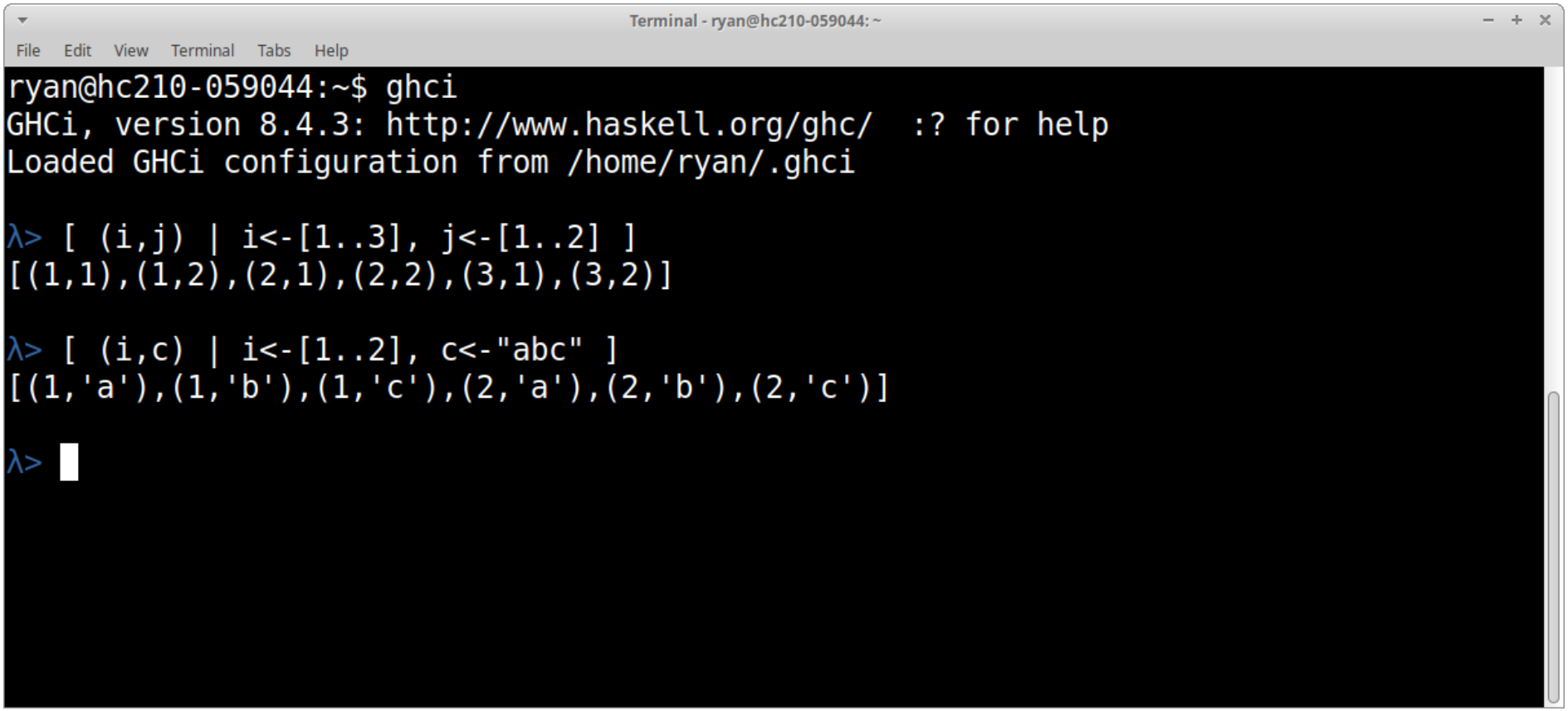


List Comprehension (Multiple Generators)

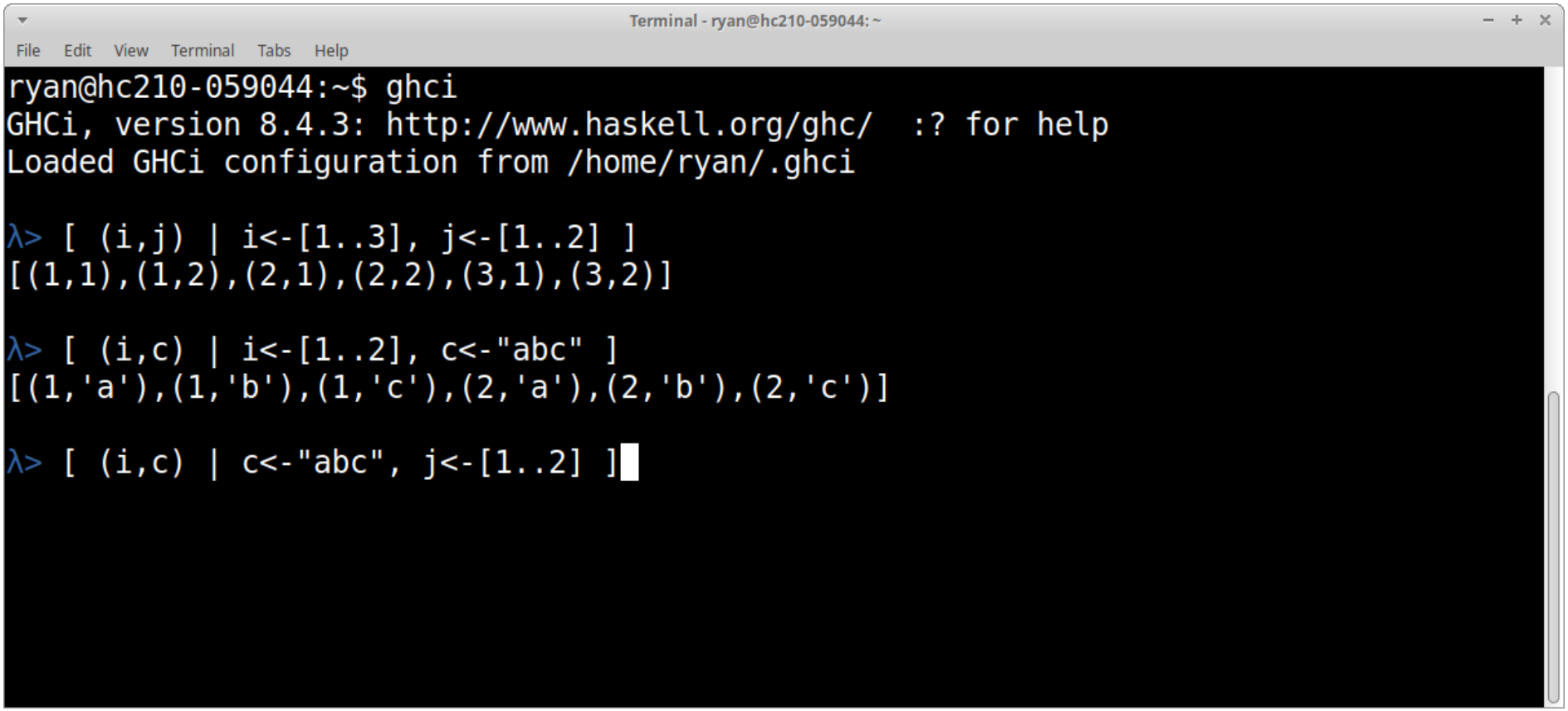
List Comprehension (Multiple Generators)



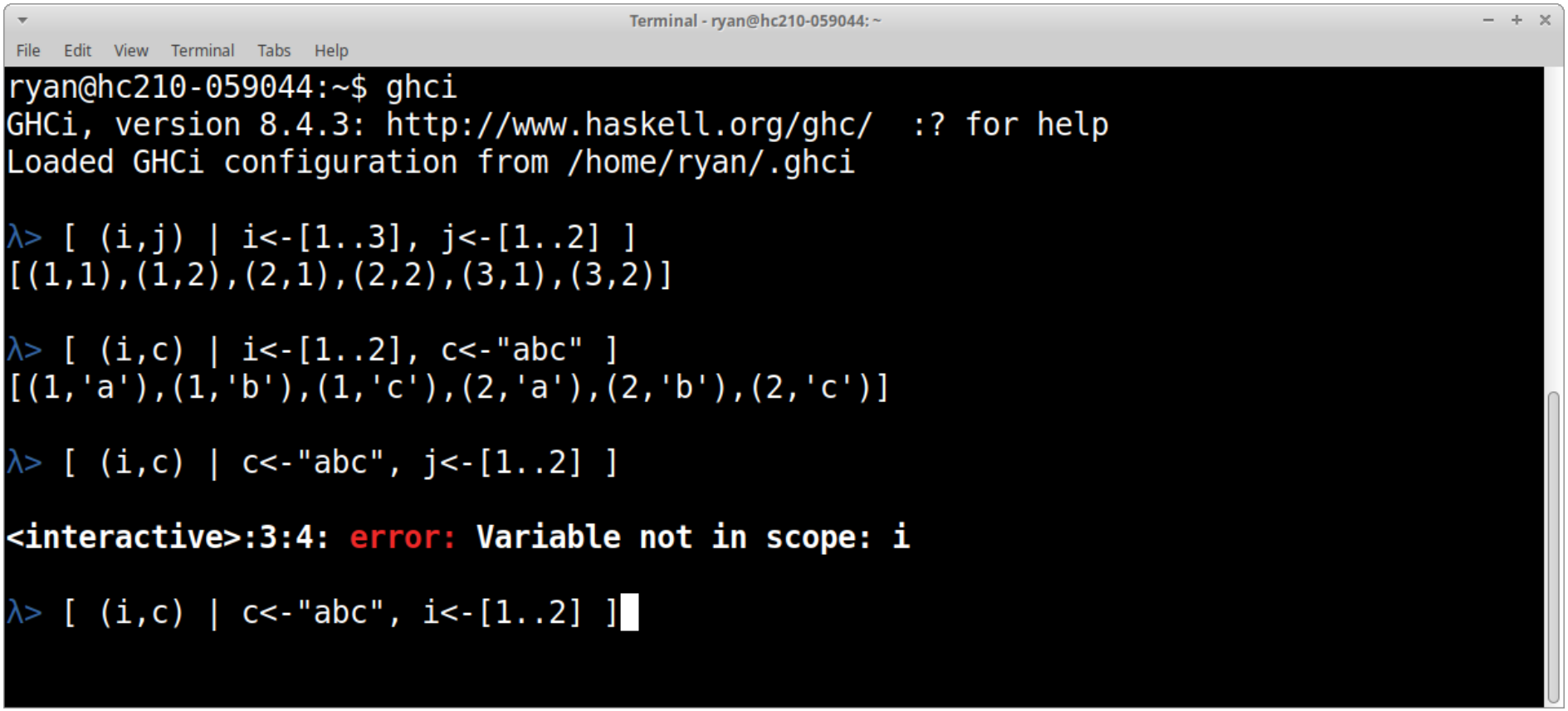
List Comprehension (Multiple Generators)



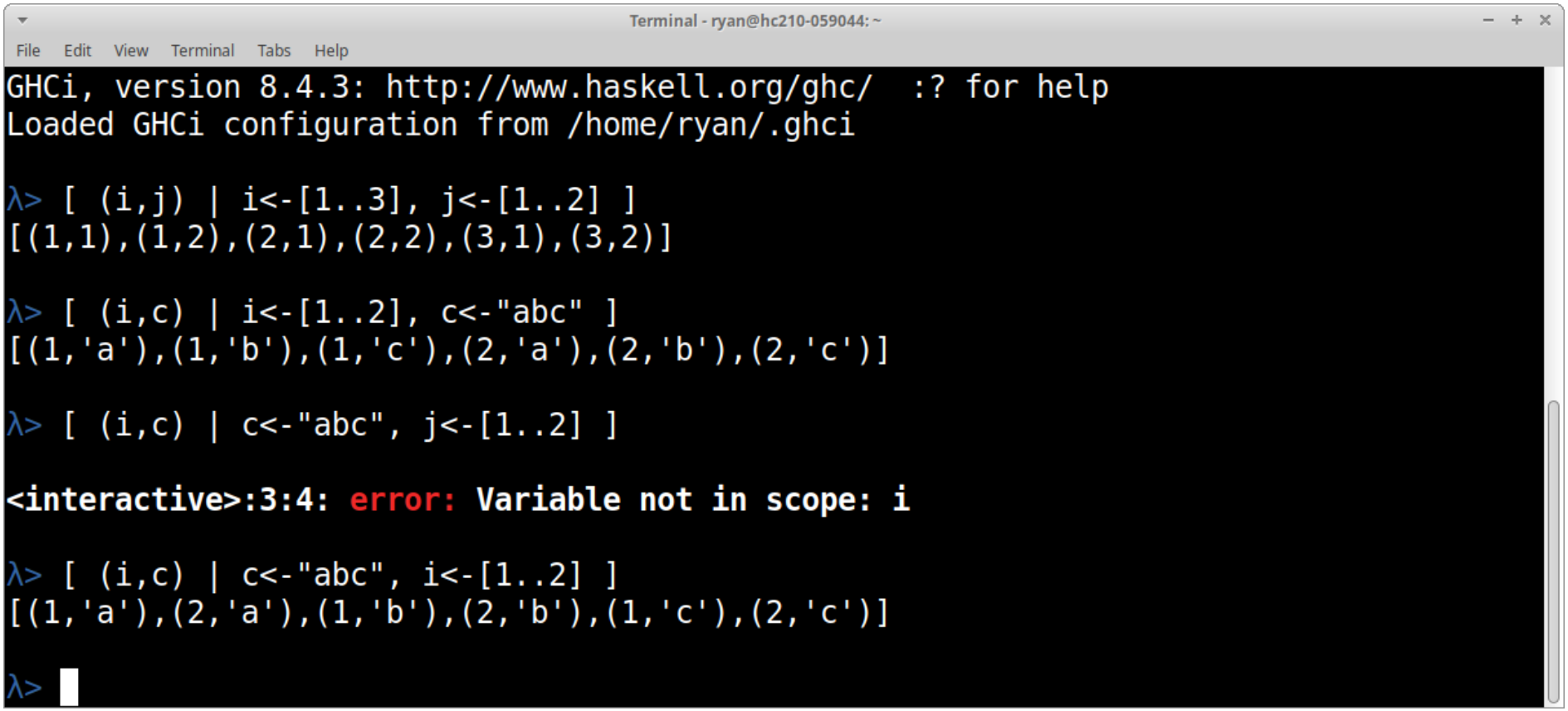
List Comprehension (Multiple Generators)



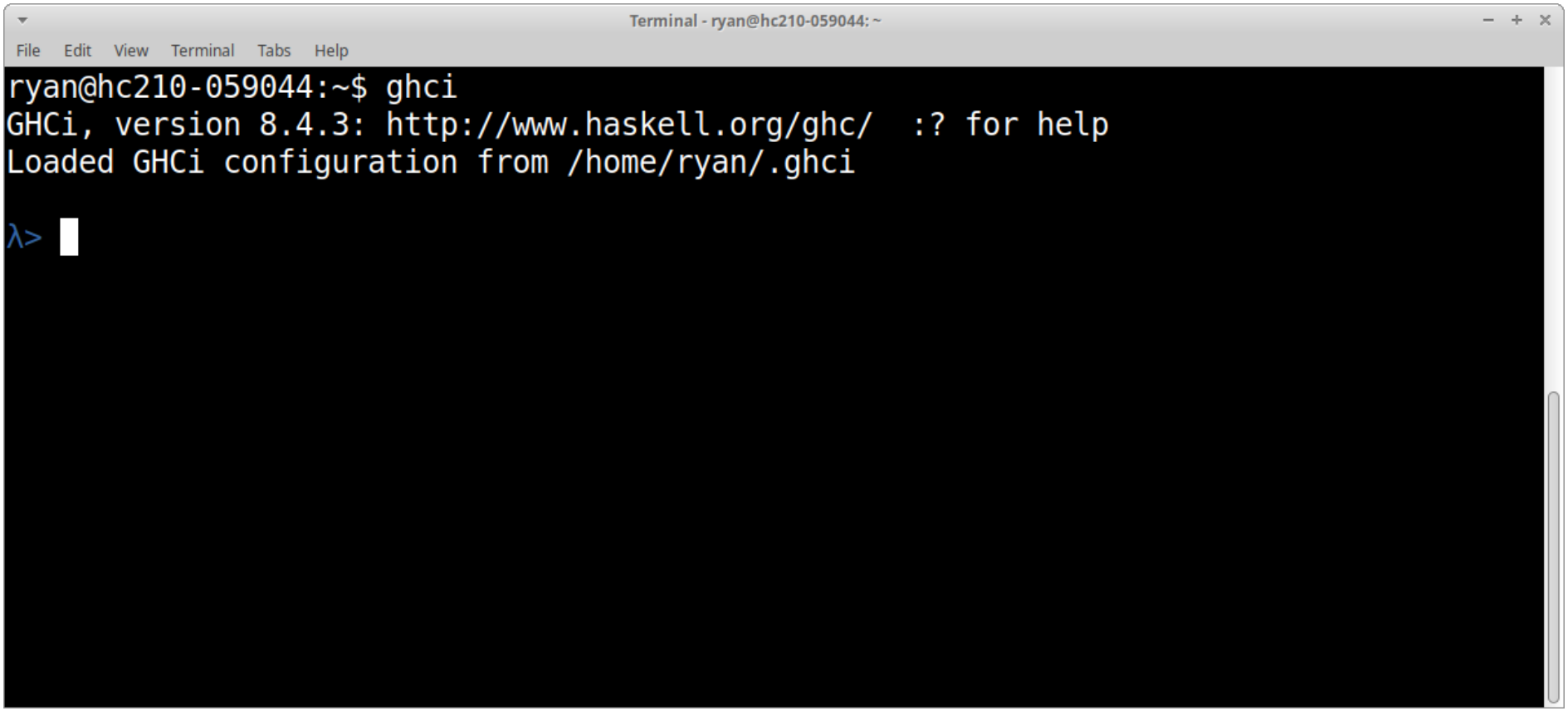
List Comprehension (Multiple Generators)



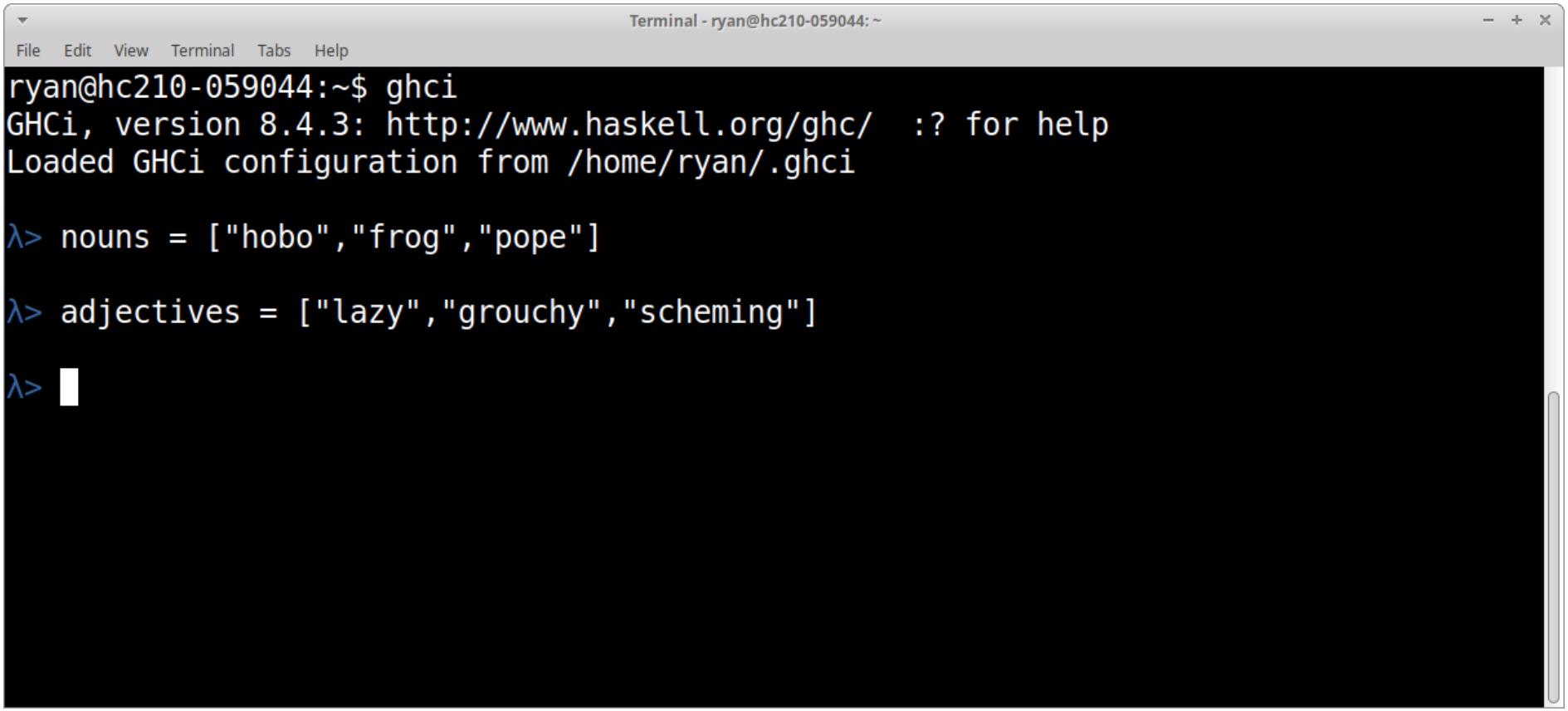
List Comprehension (Multiple Generators)



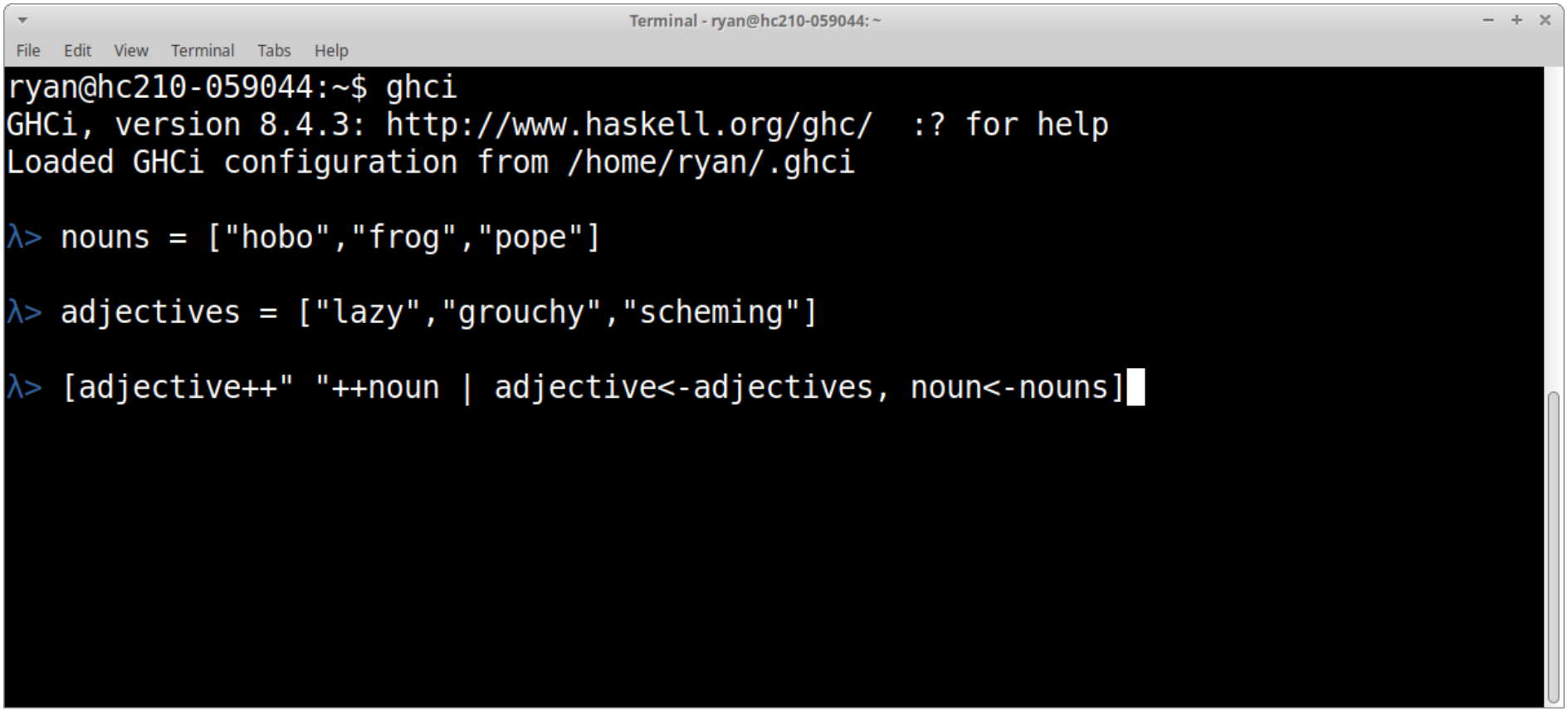
List Comprehension (Multiple Generators)



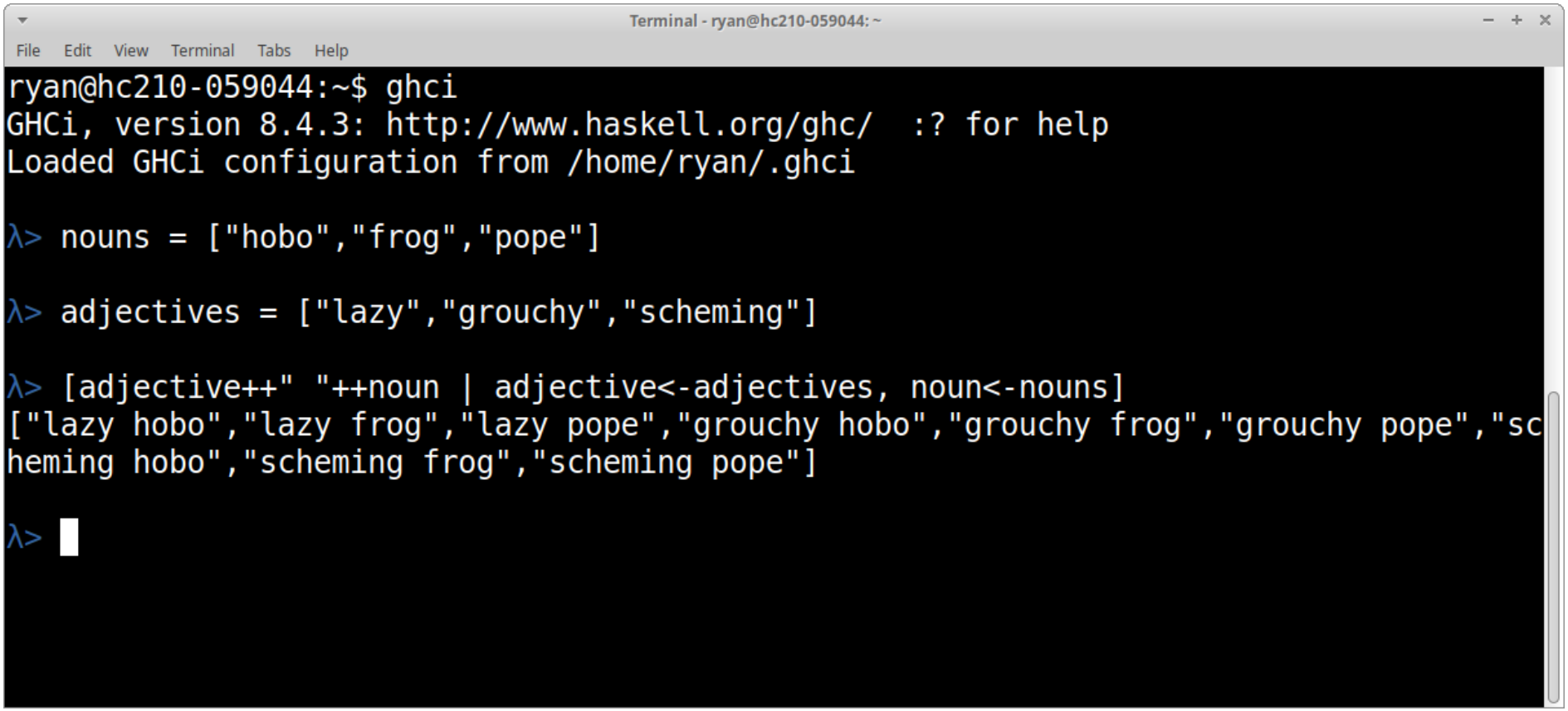
List Comprehension (Multiple Generators)



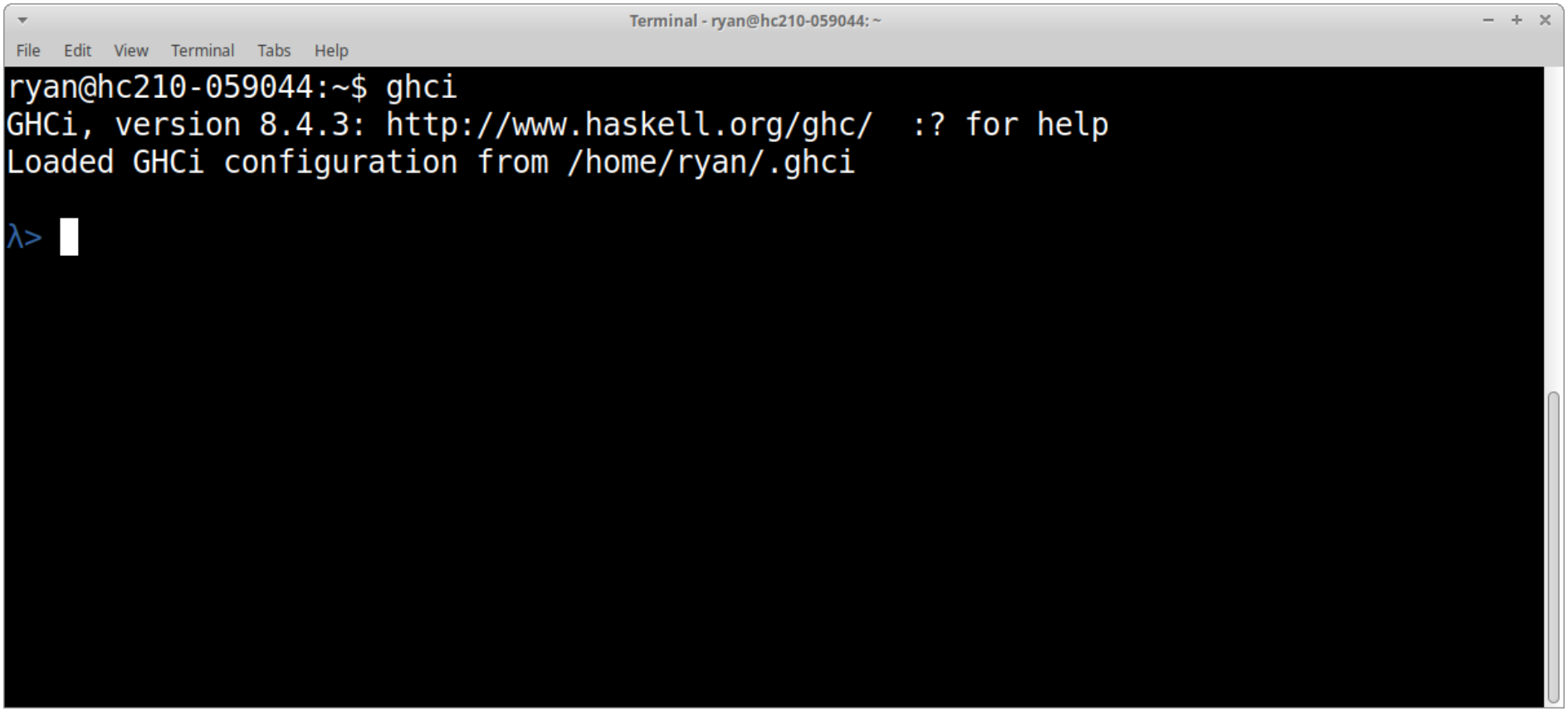
List Comprehension (Multiple Generators)



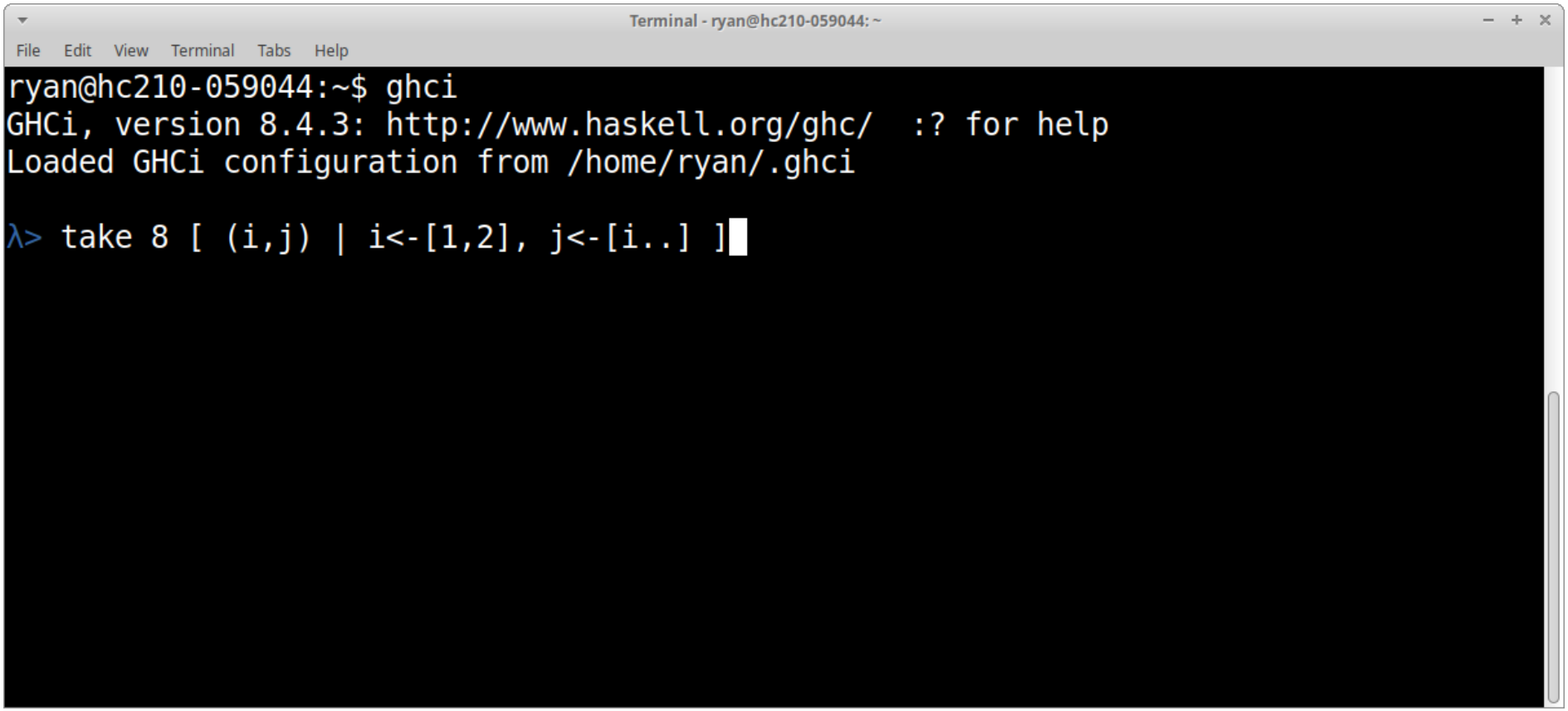
List Comprehension (Multiple Generators)



List Comprehension (Infinite Lists, Nested)



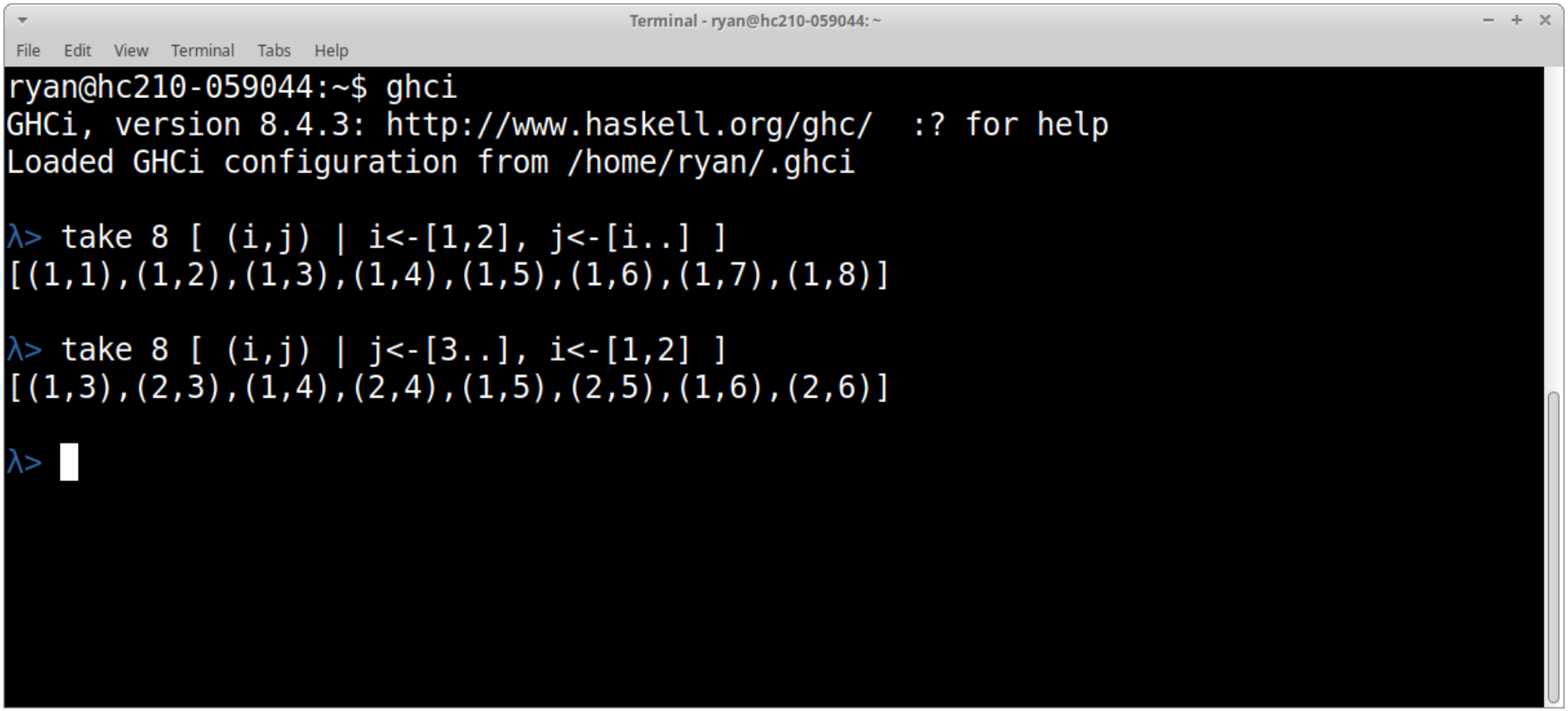
List Comprehension (Infinite Lists, Nested)



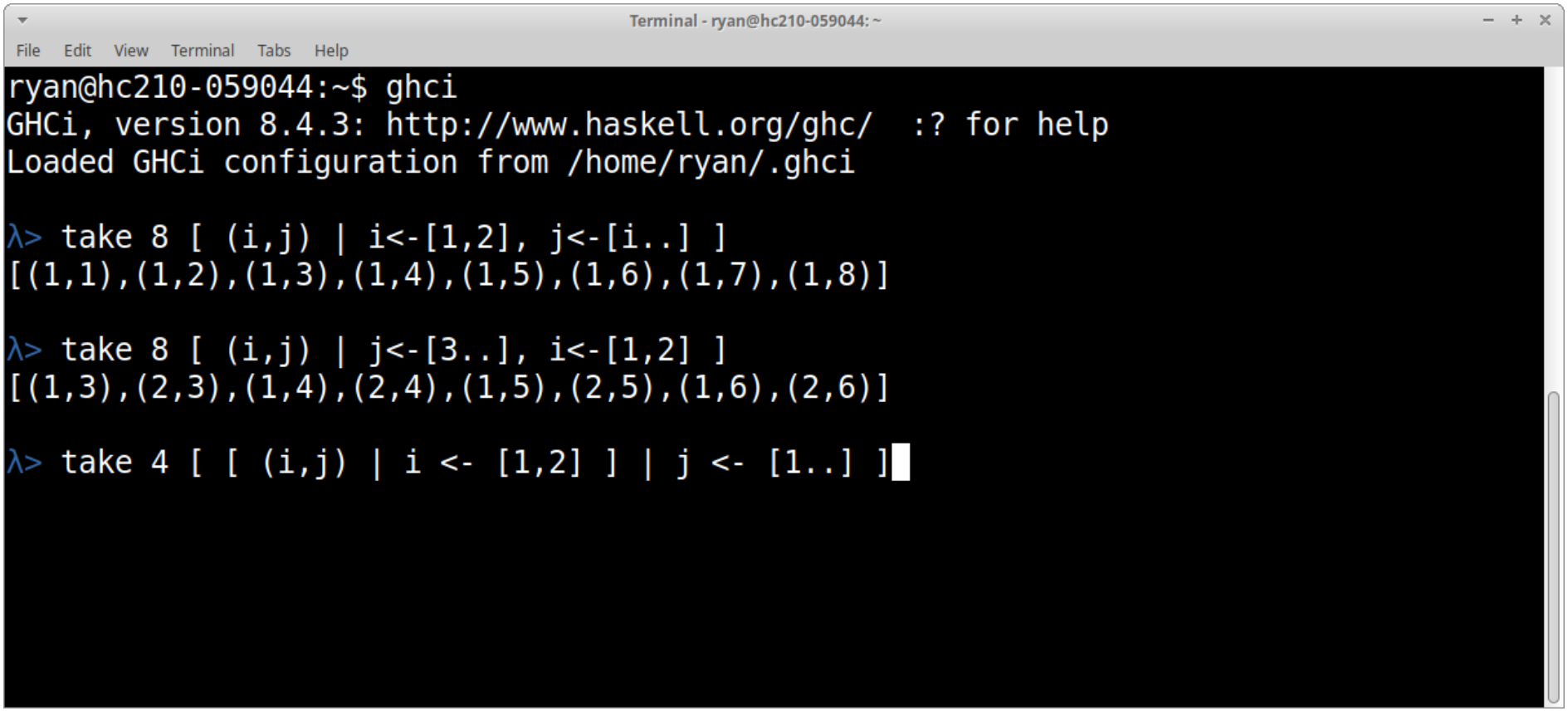
List Comprehension (Infinite Lists, Nested)

List Comprehension (Infinite Lists, Nested)

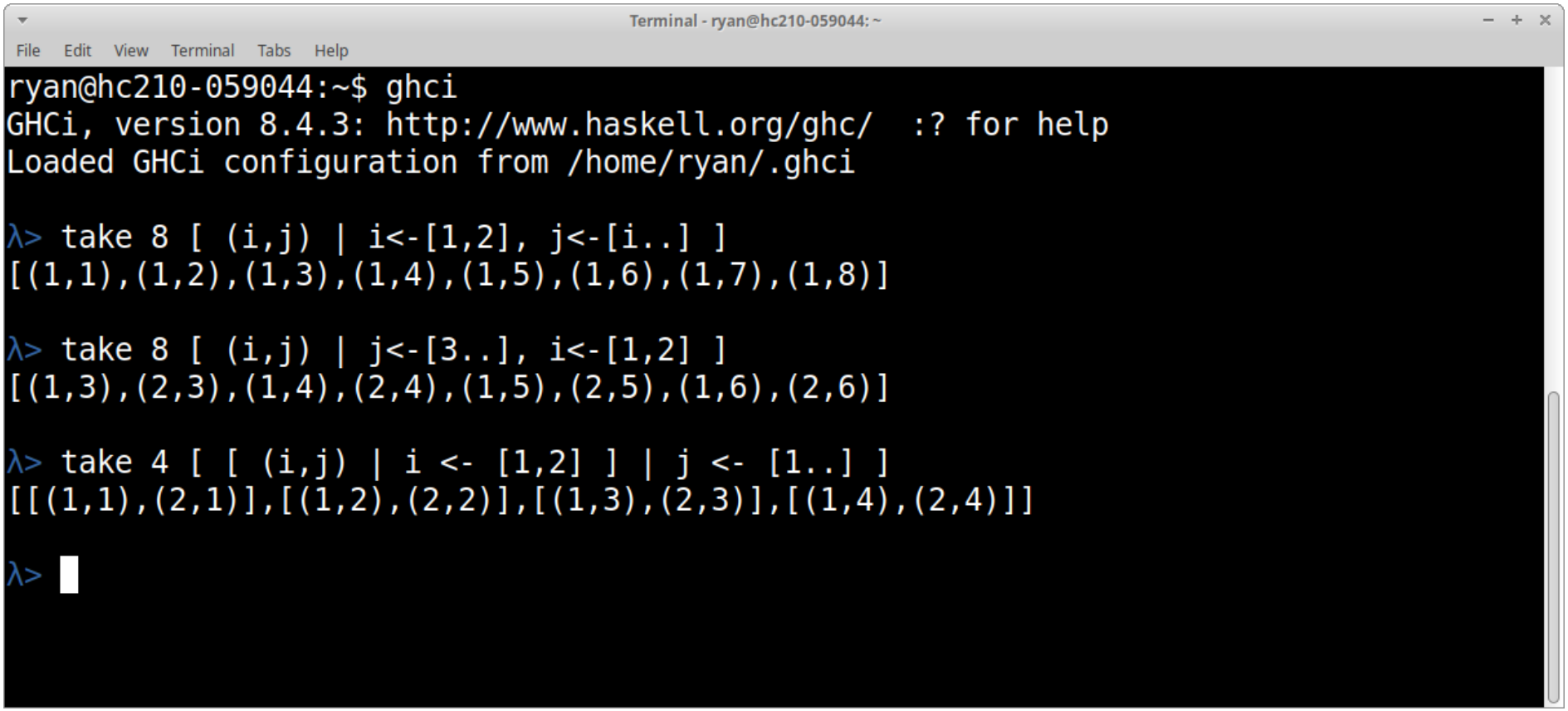
List Comprehension (Infinite Lists, Nested)



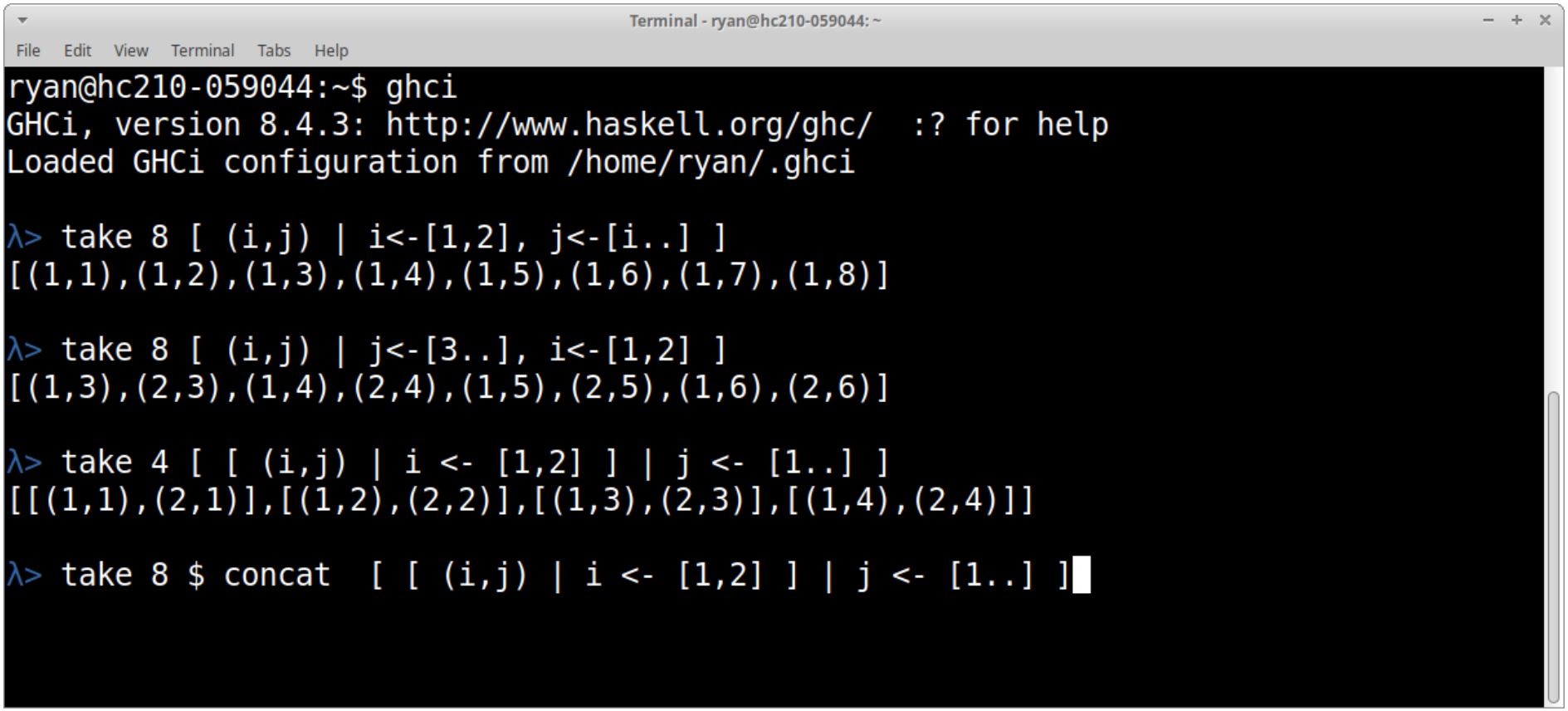
List Comprehension (Infinite Lists, Nested)



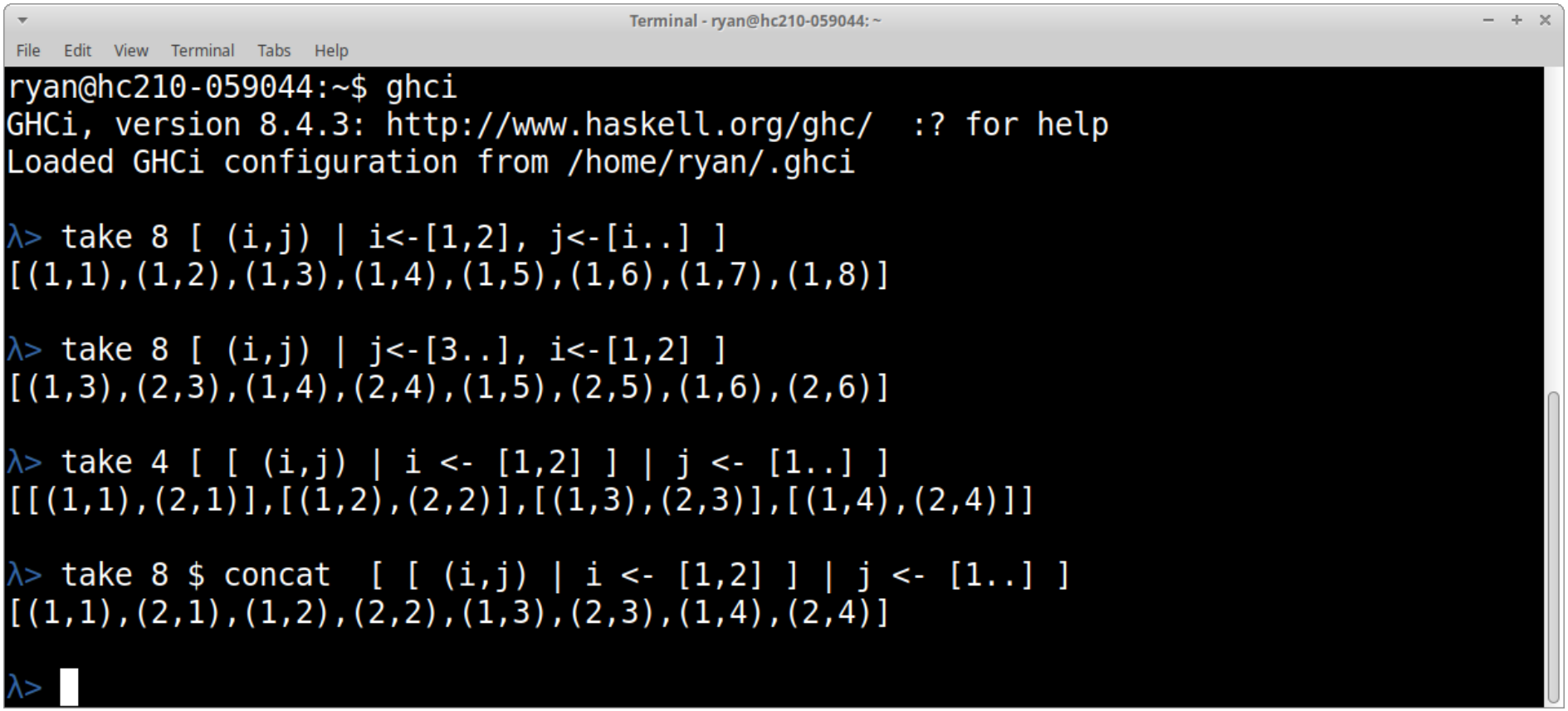
List Comprehension (Infinite Lists, Nested)



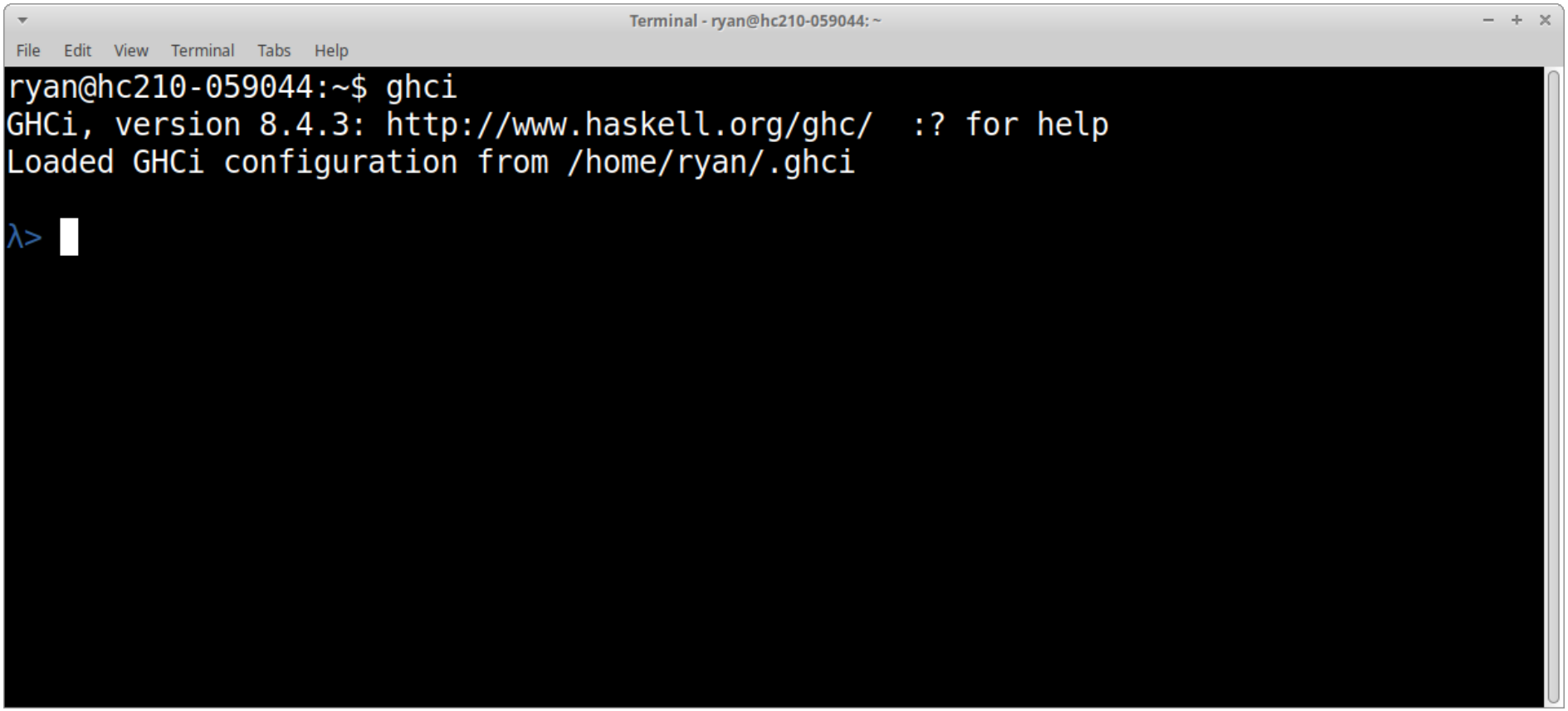
List Comprehension (Infinite Lists, Nested)



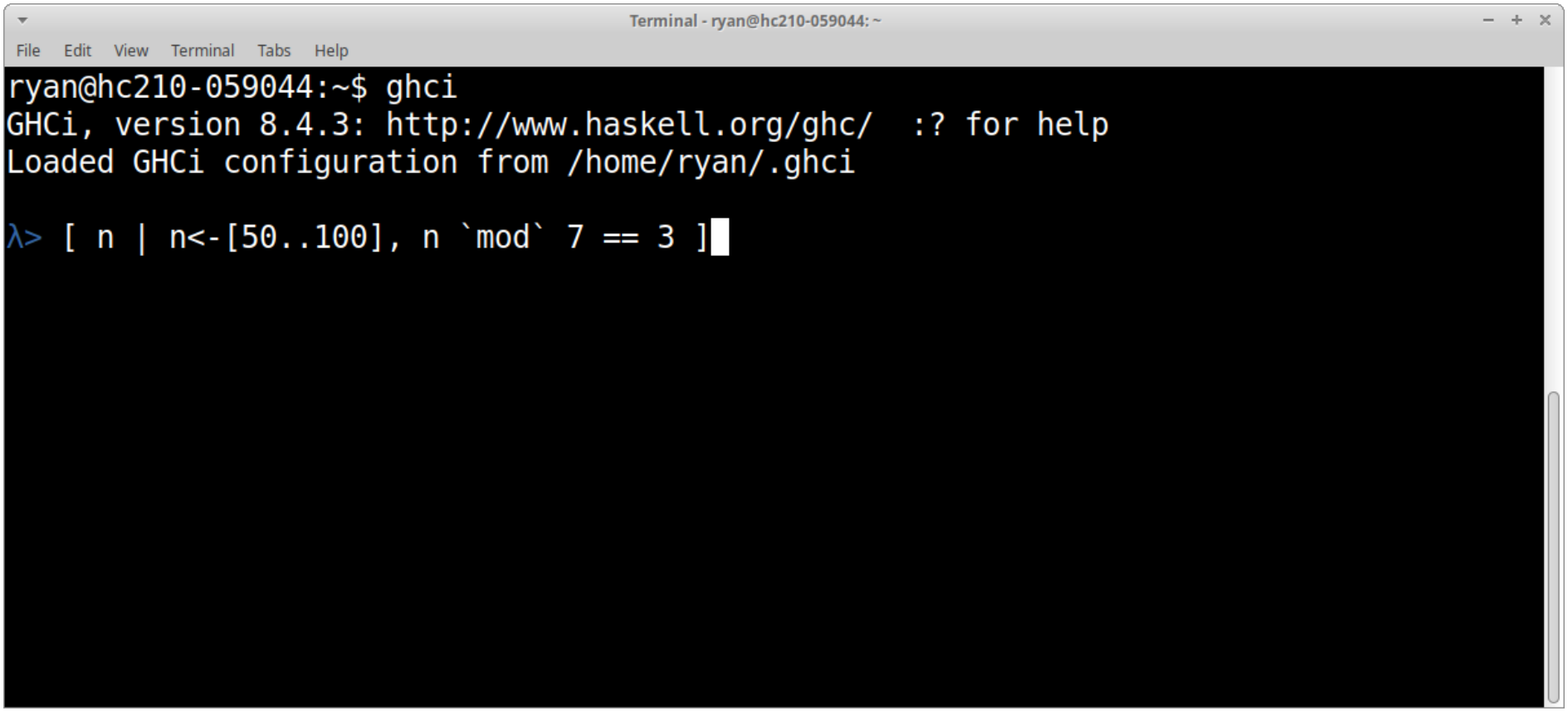
List Comprehension (Infinite Lists, Nested)



List Comprehension (Guards aka Filters)



List Comprehension (Guards aka Filters)



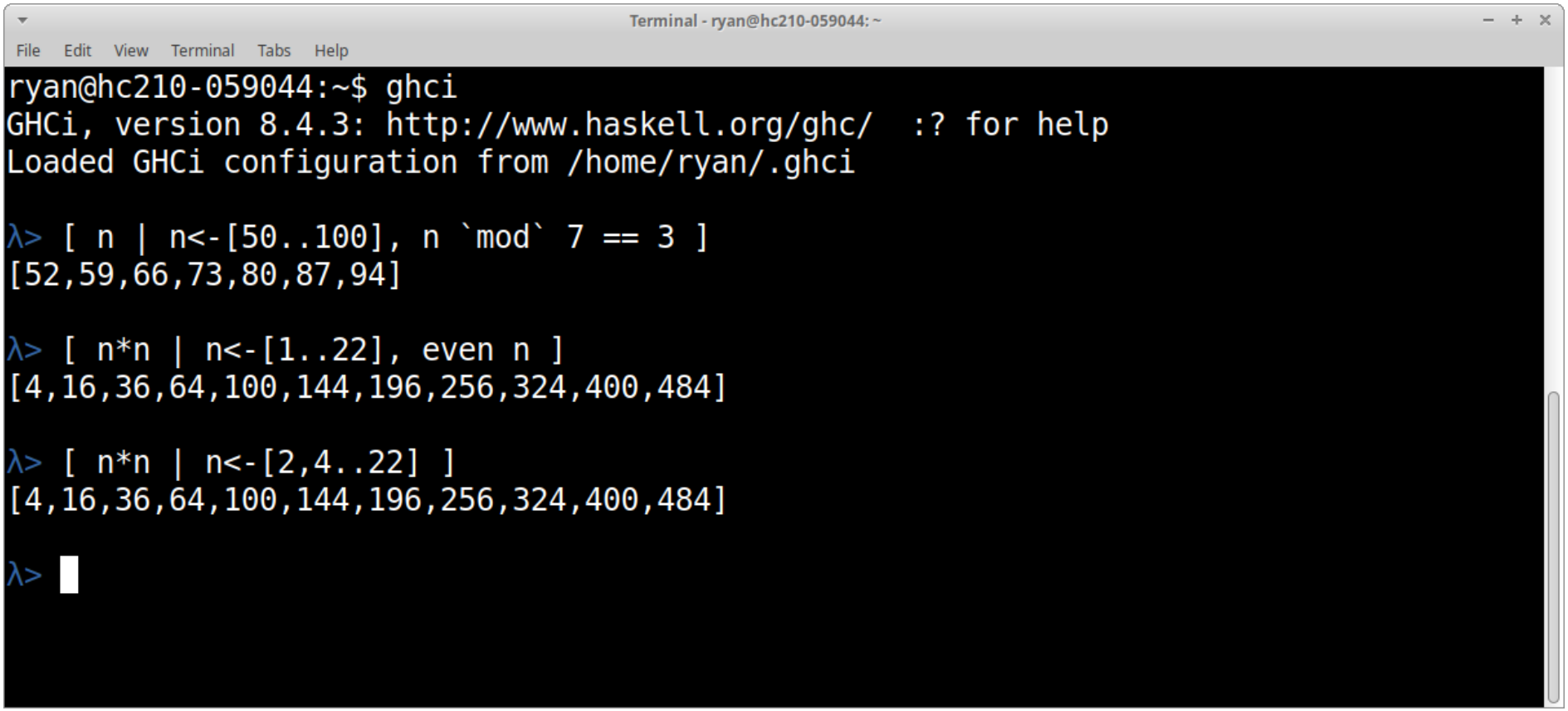
List Comprehension (Guards aka Filters)

List Comprehension (Guards aka Filters)

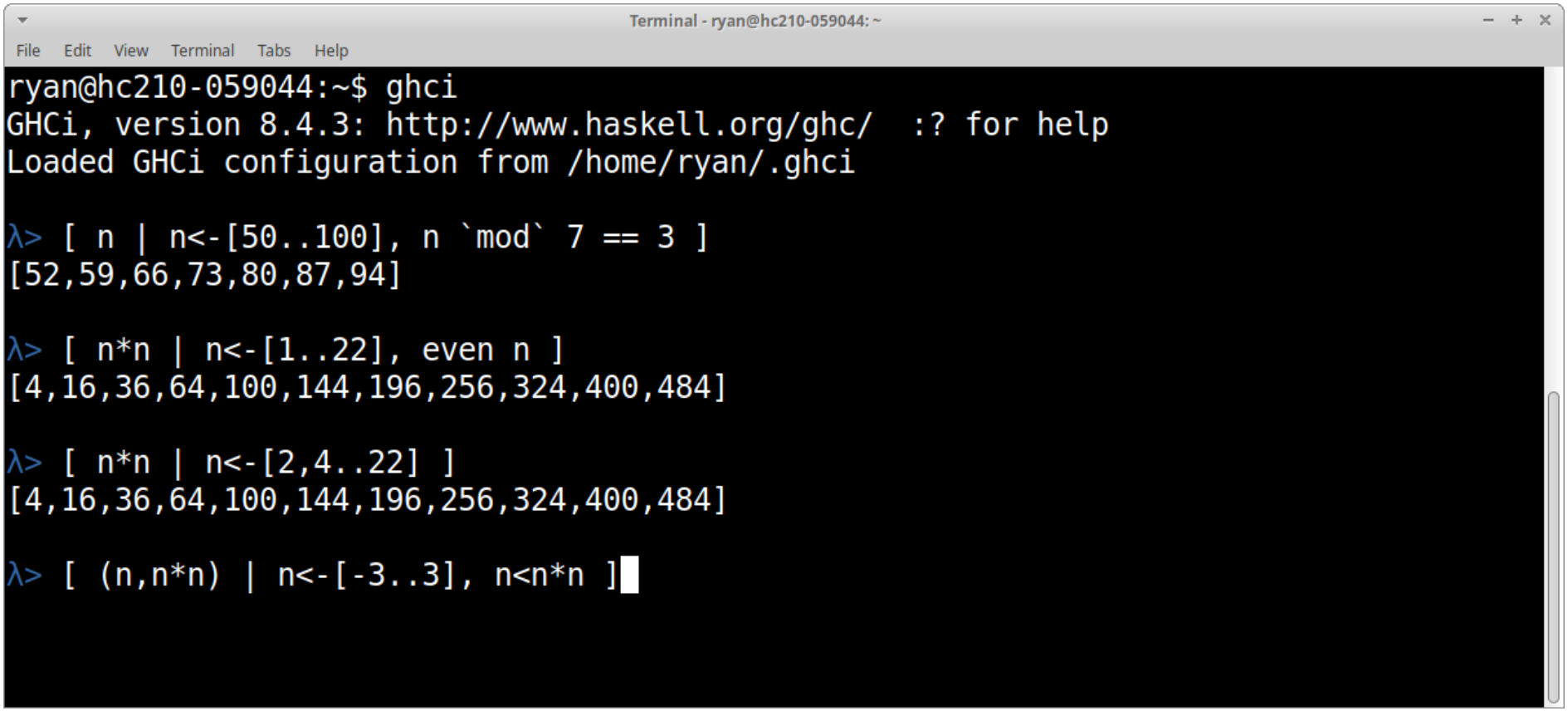
List Comprehension (Guards aka Filters)

List Comprehension (Guards aka Filters)

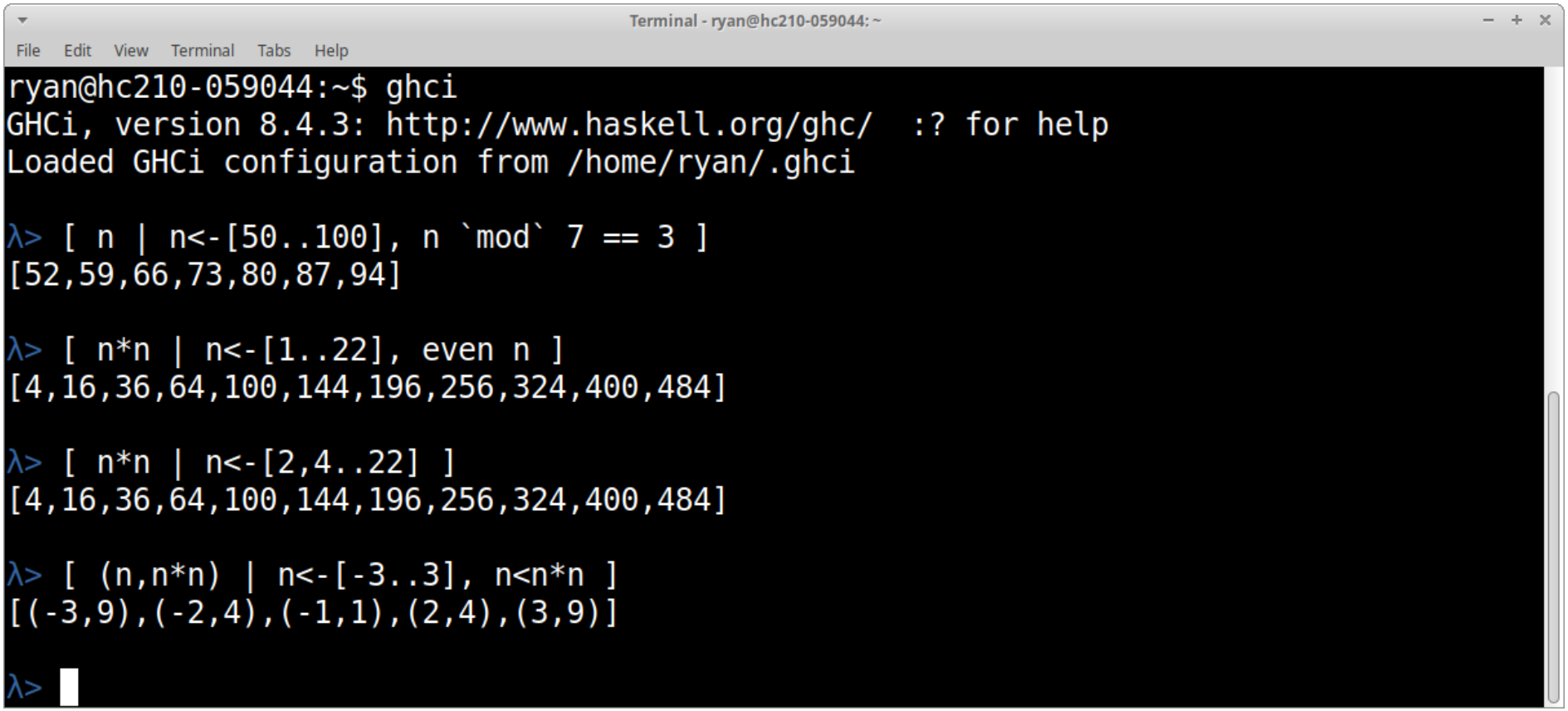
List Comprehension (Guards aka Filters)



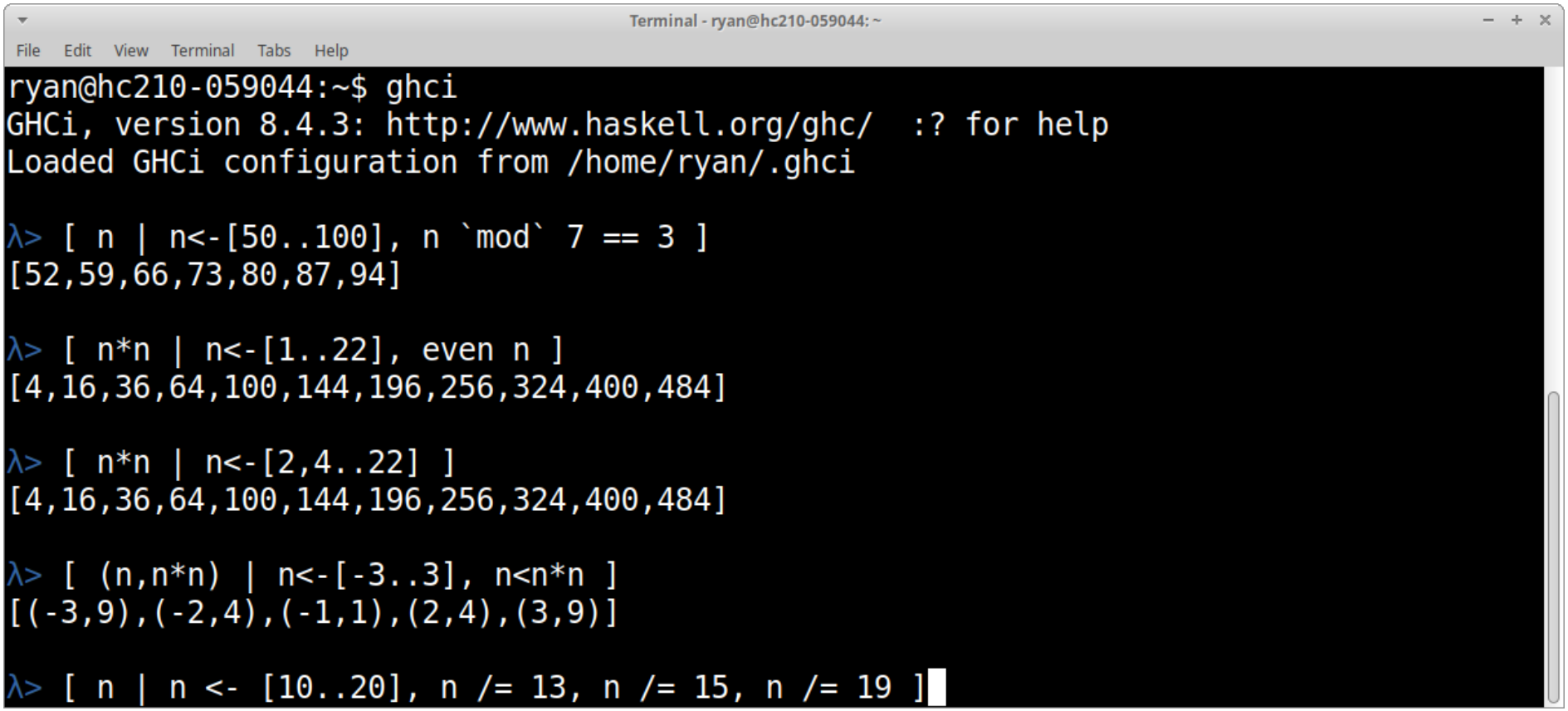
List Comprehension (Guards aka Filters)



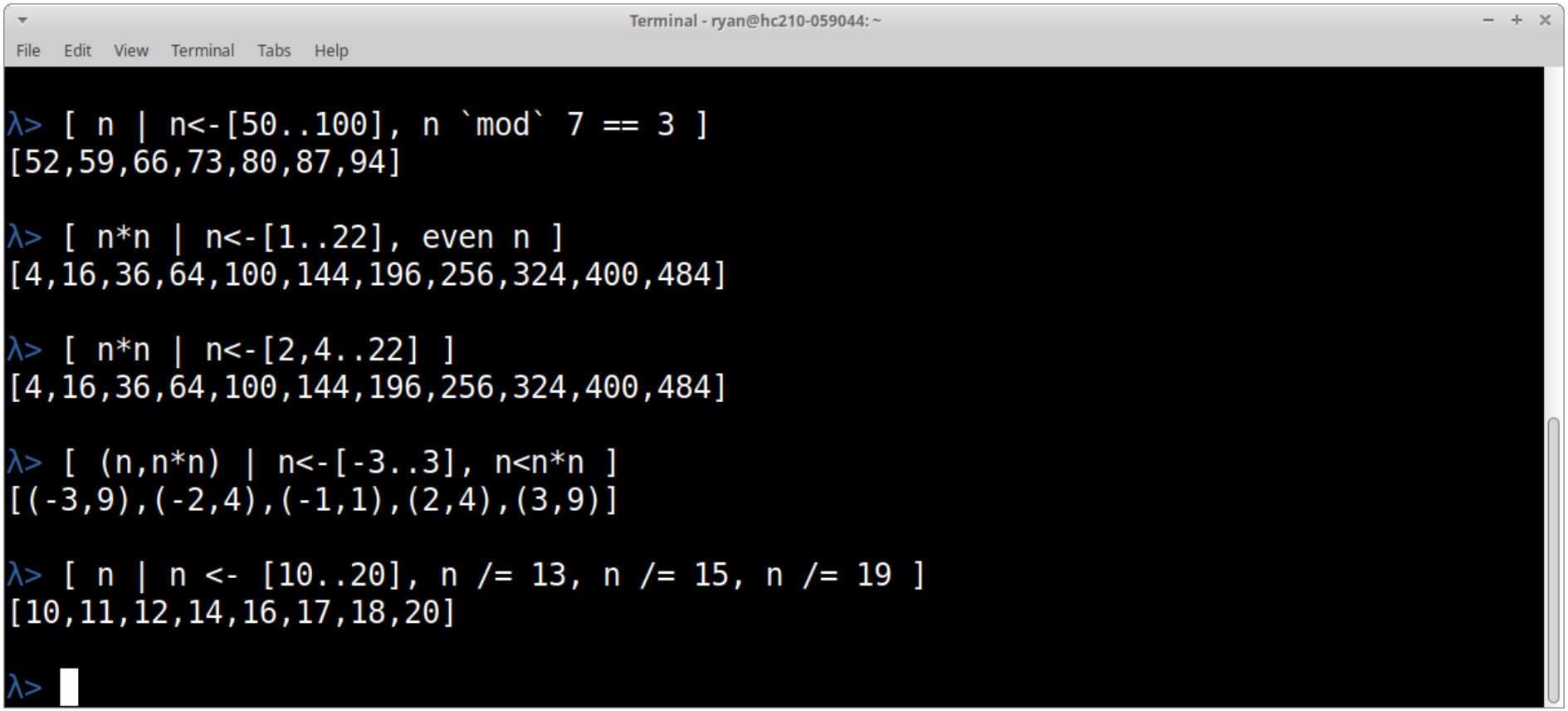
List Comprehension (Guards aka Filters)



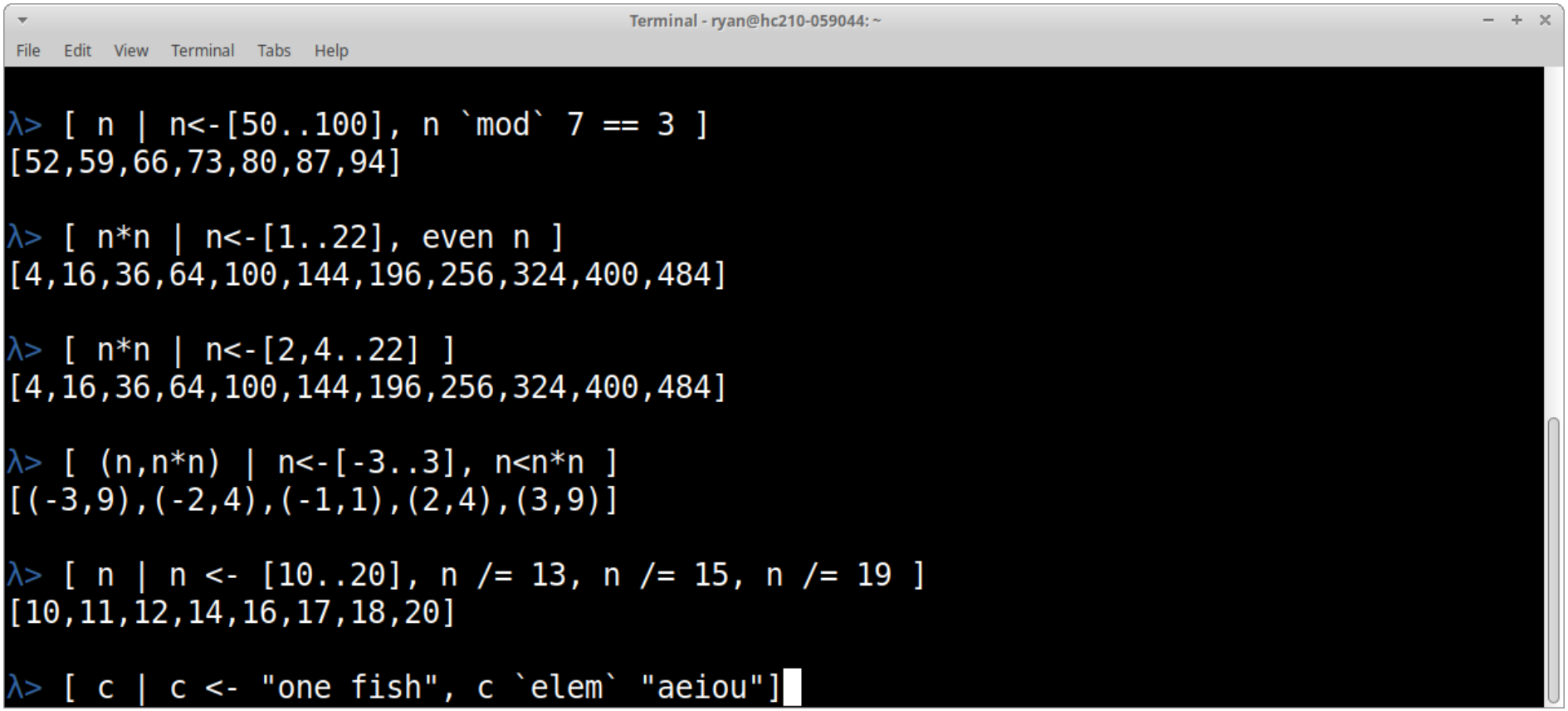
List Comprehension (Guards aka Filters)



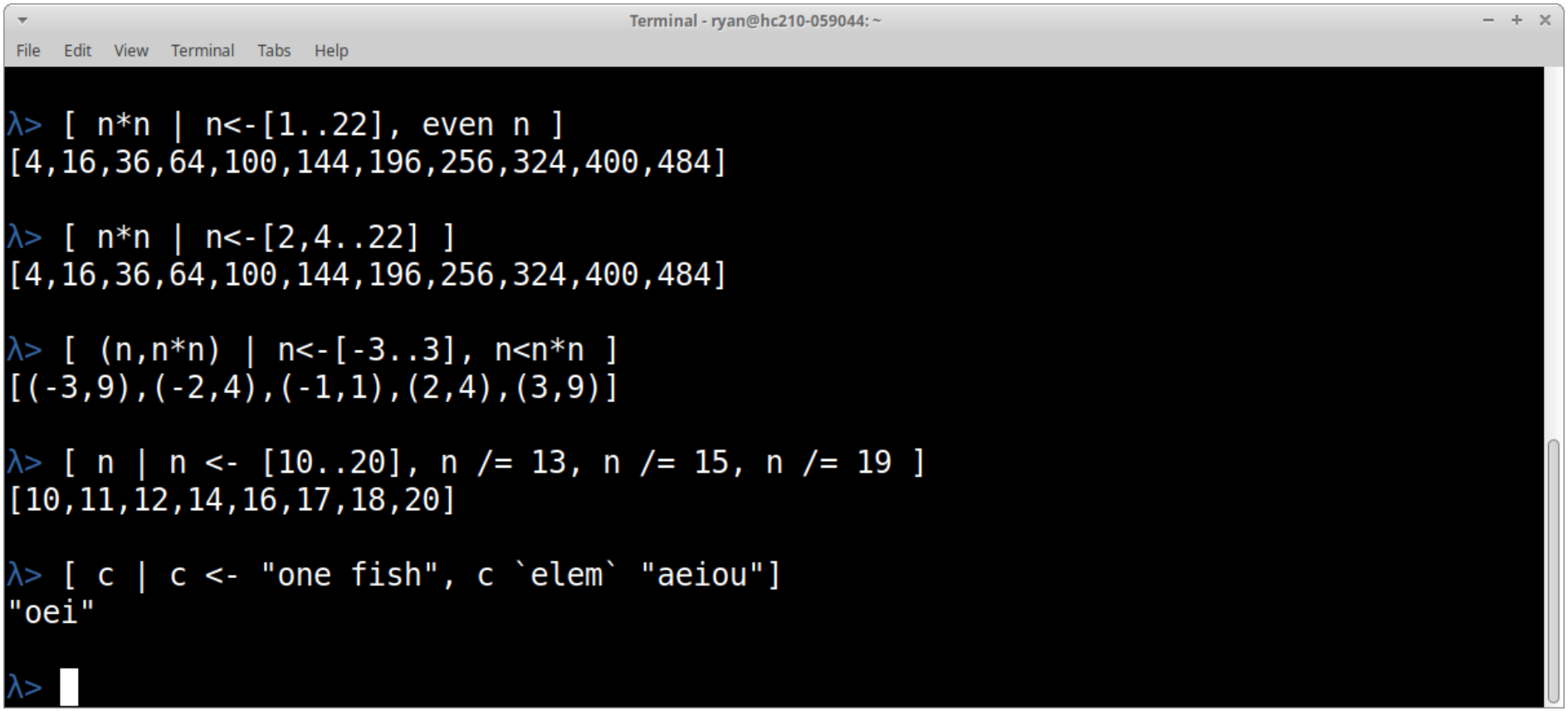
List Comprehension (Guards aka Filters)



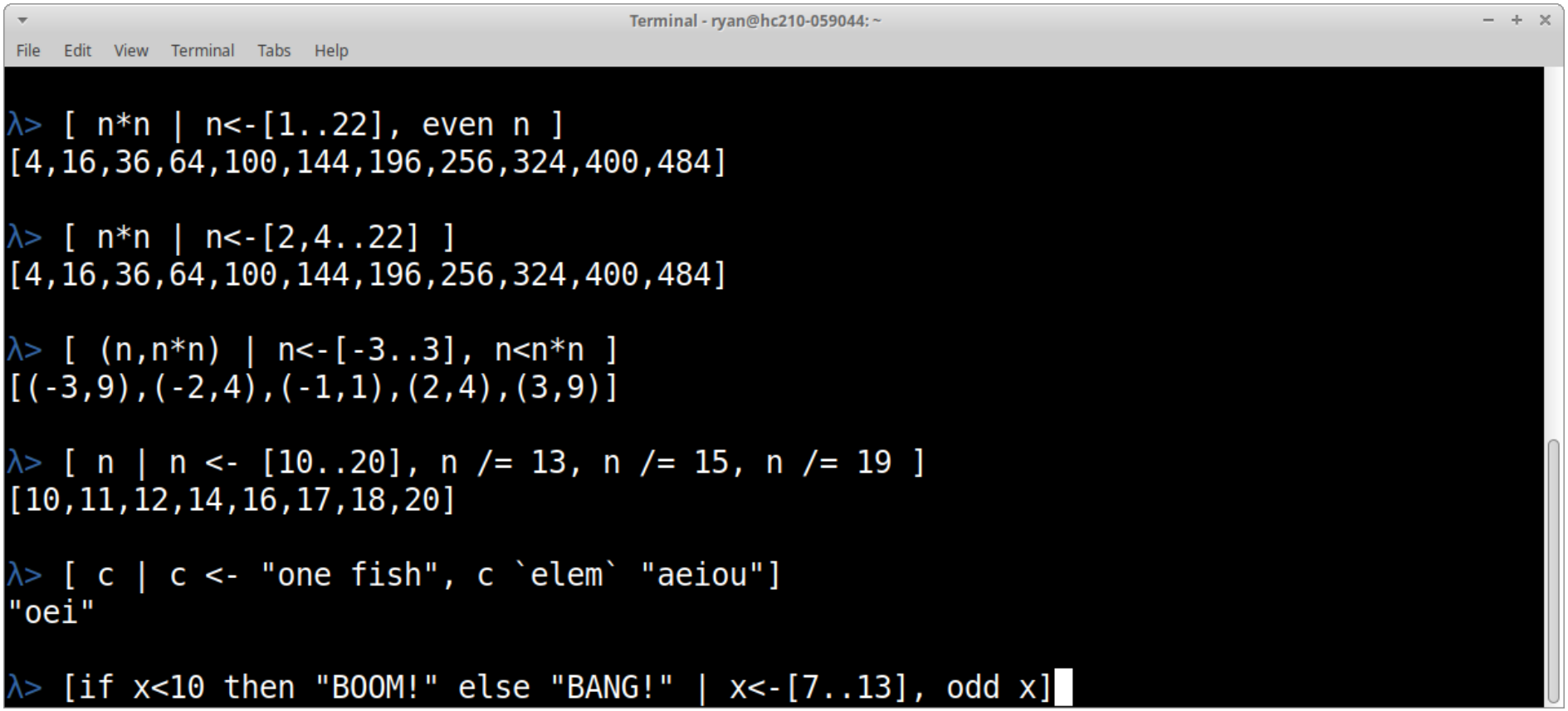
List Comprehension (Guards aka Filters)



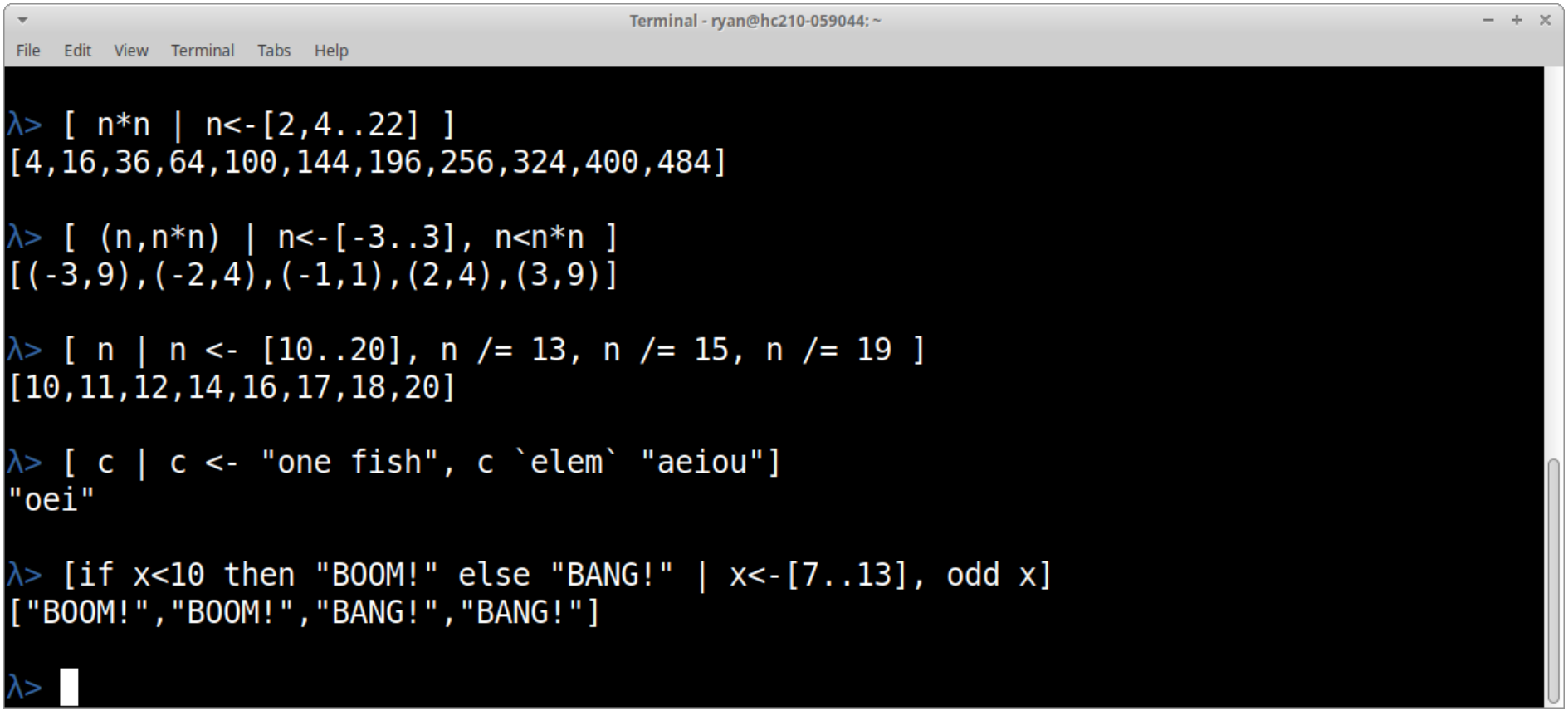
List Comprehension (Guards aka Filters)



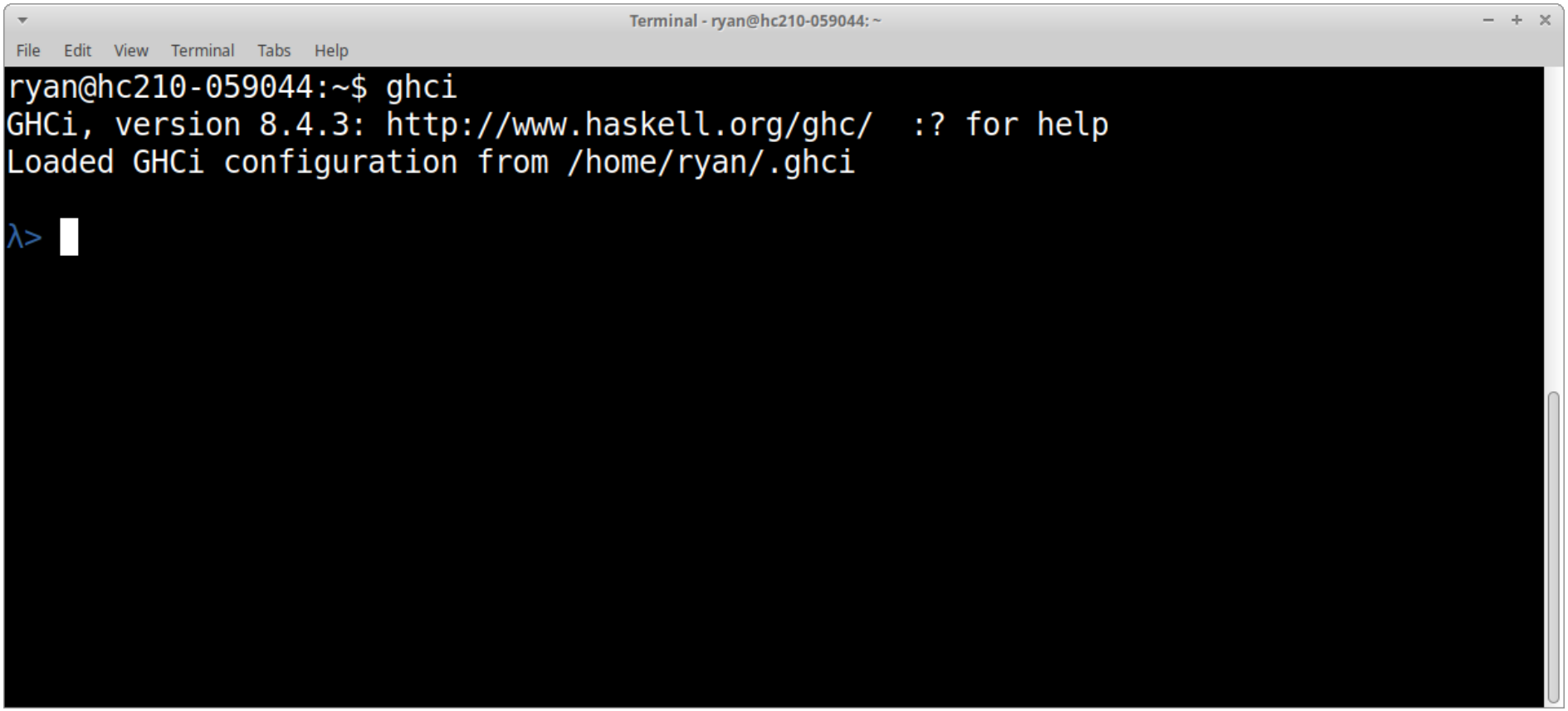
List Comprehension (Guards aka Filters)



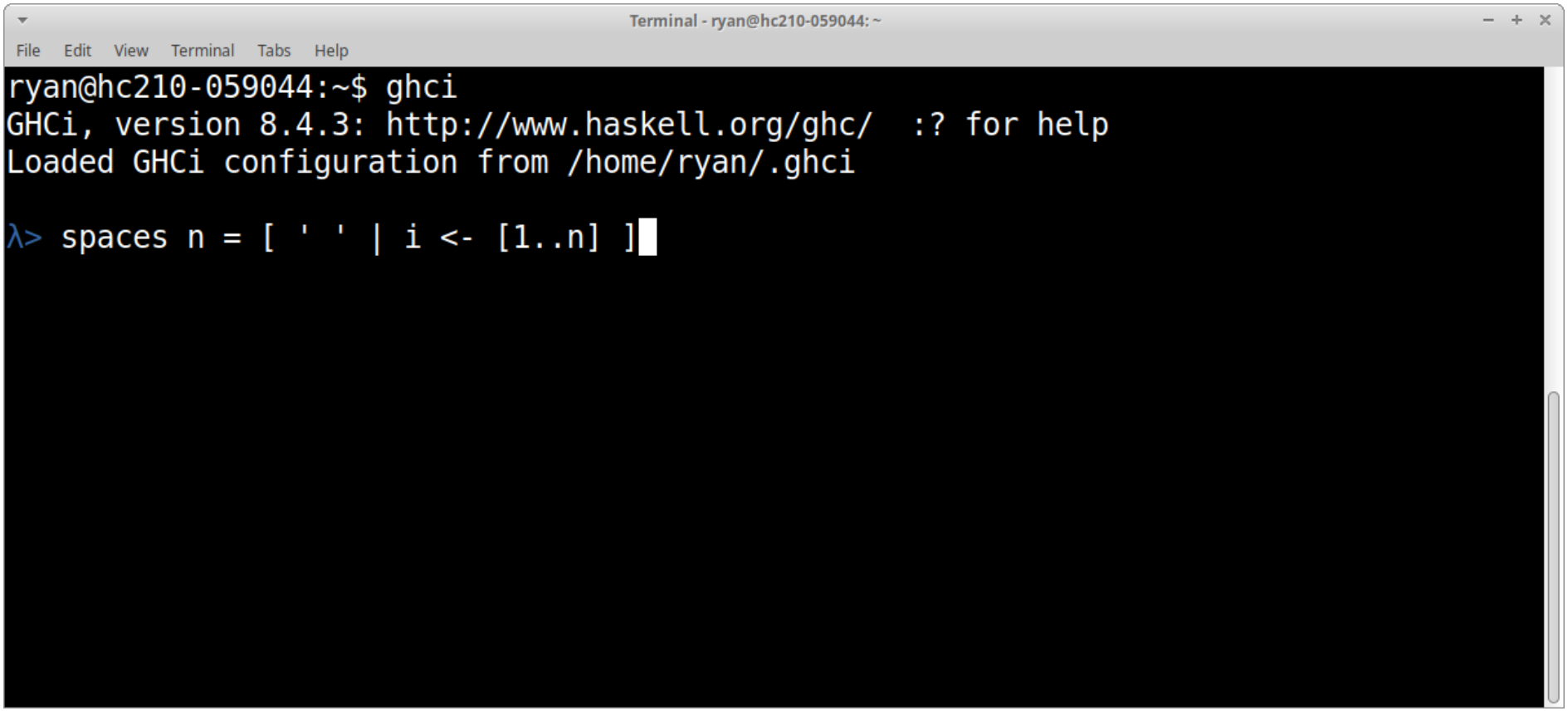
List Comprehension (Guards aka Filters)



List Comprehension and Functions

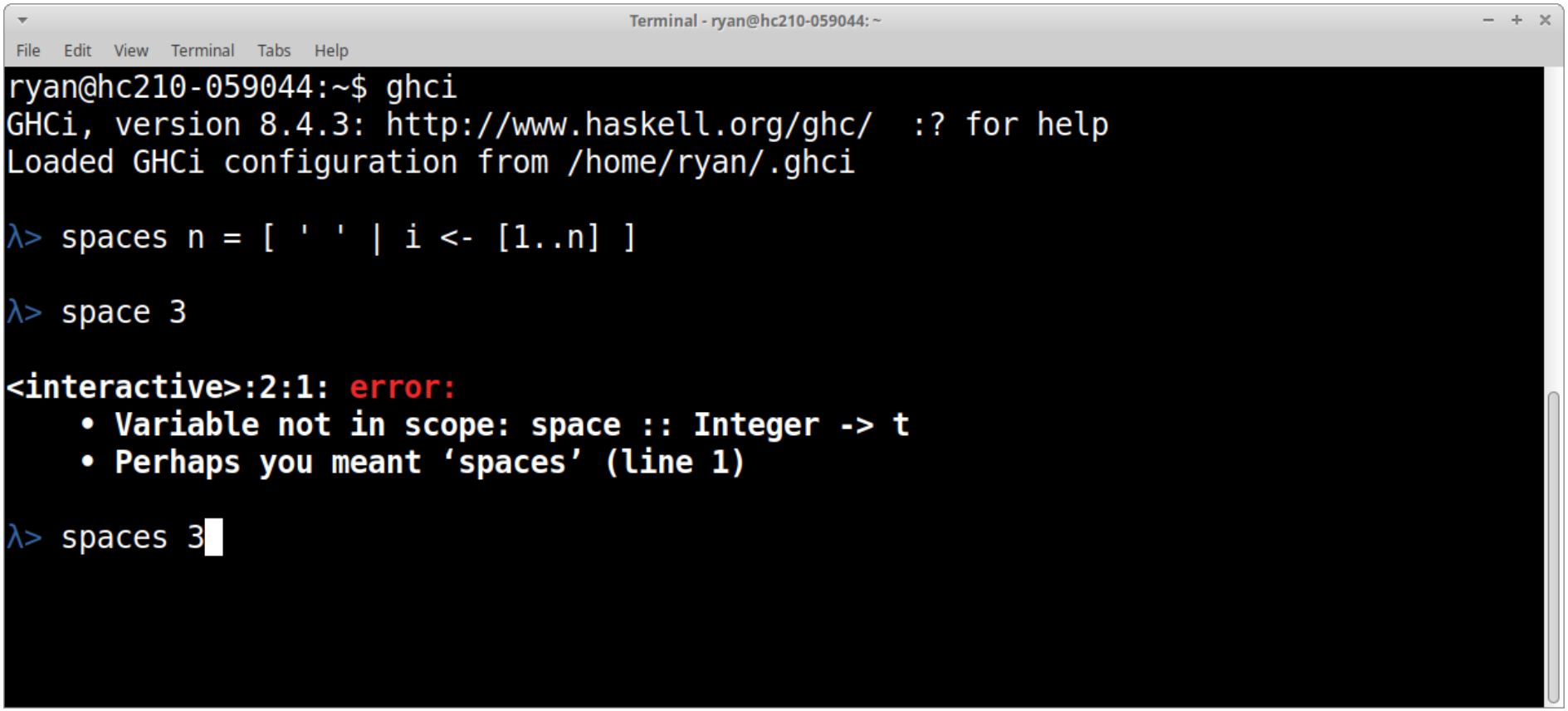


List Comprehension and Functions

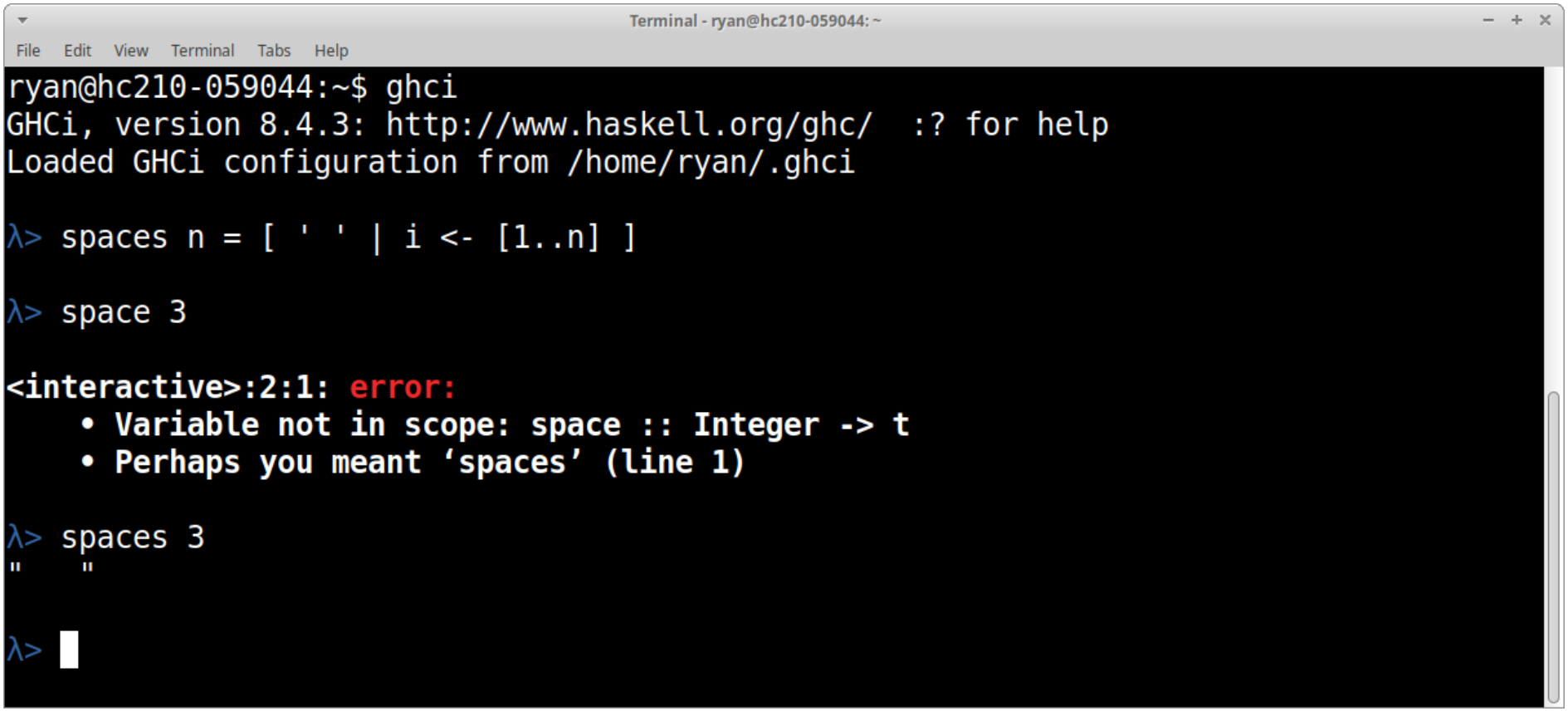


List Comprehension and Functions

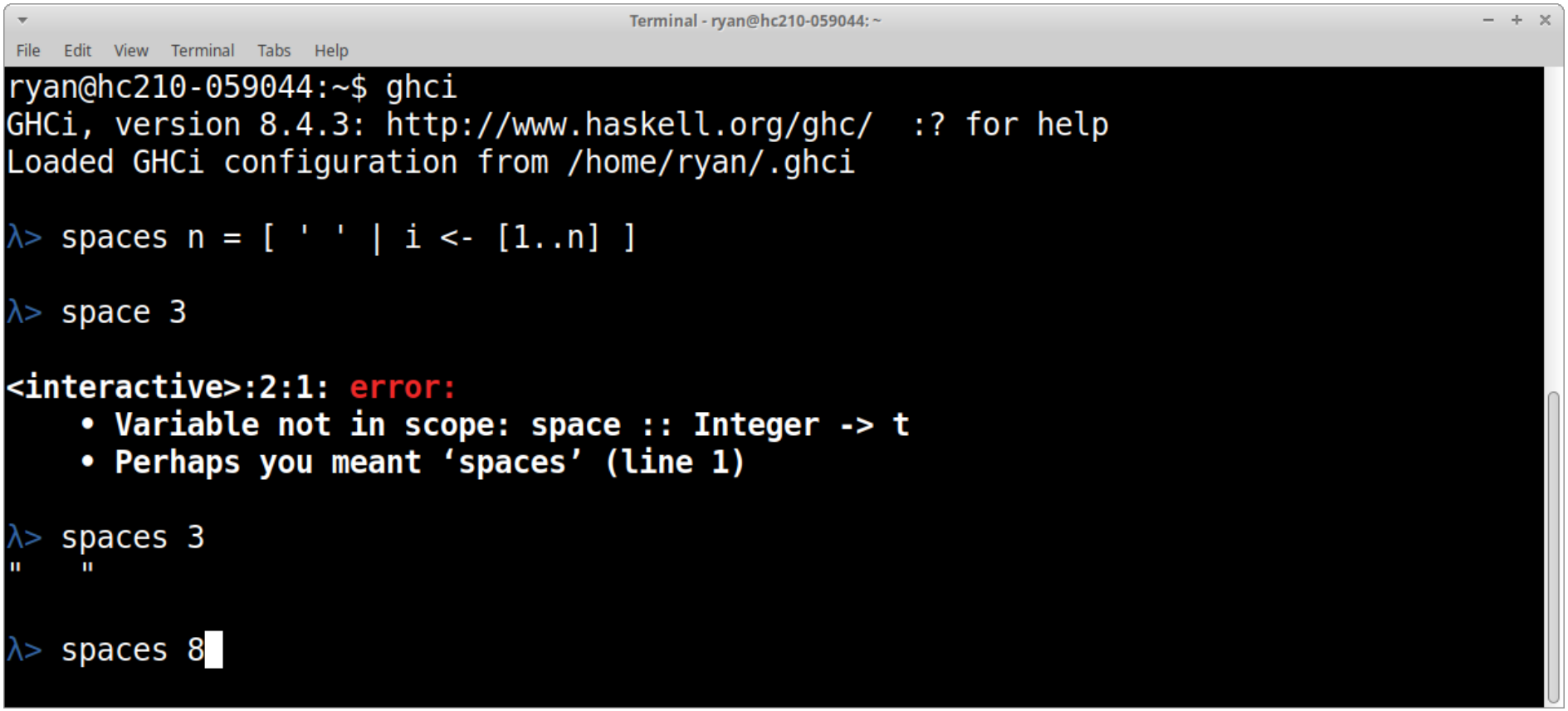
List Comprehension and Functions



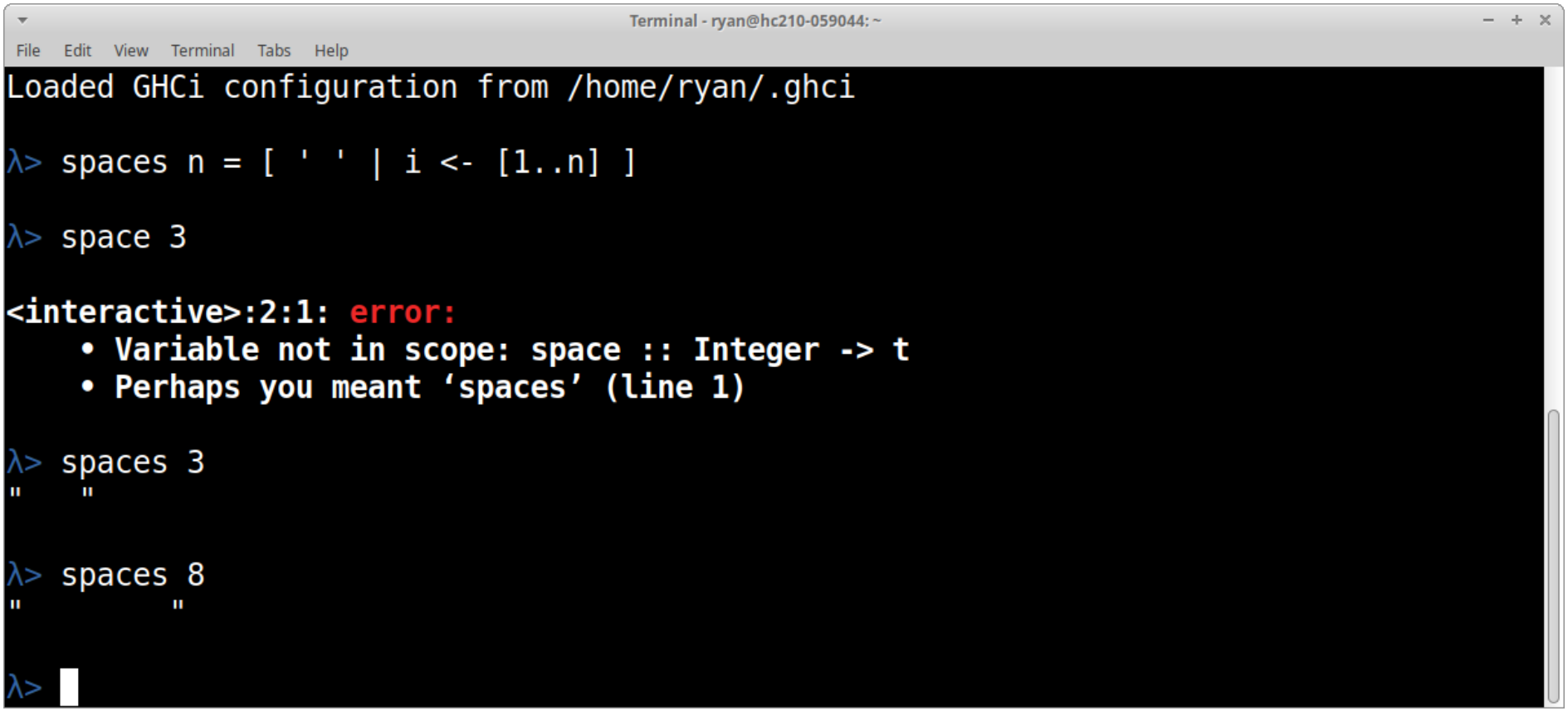
List Comprehension and Functions



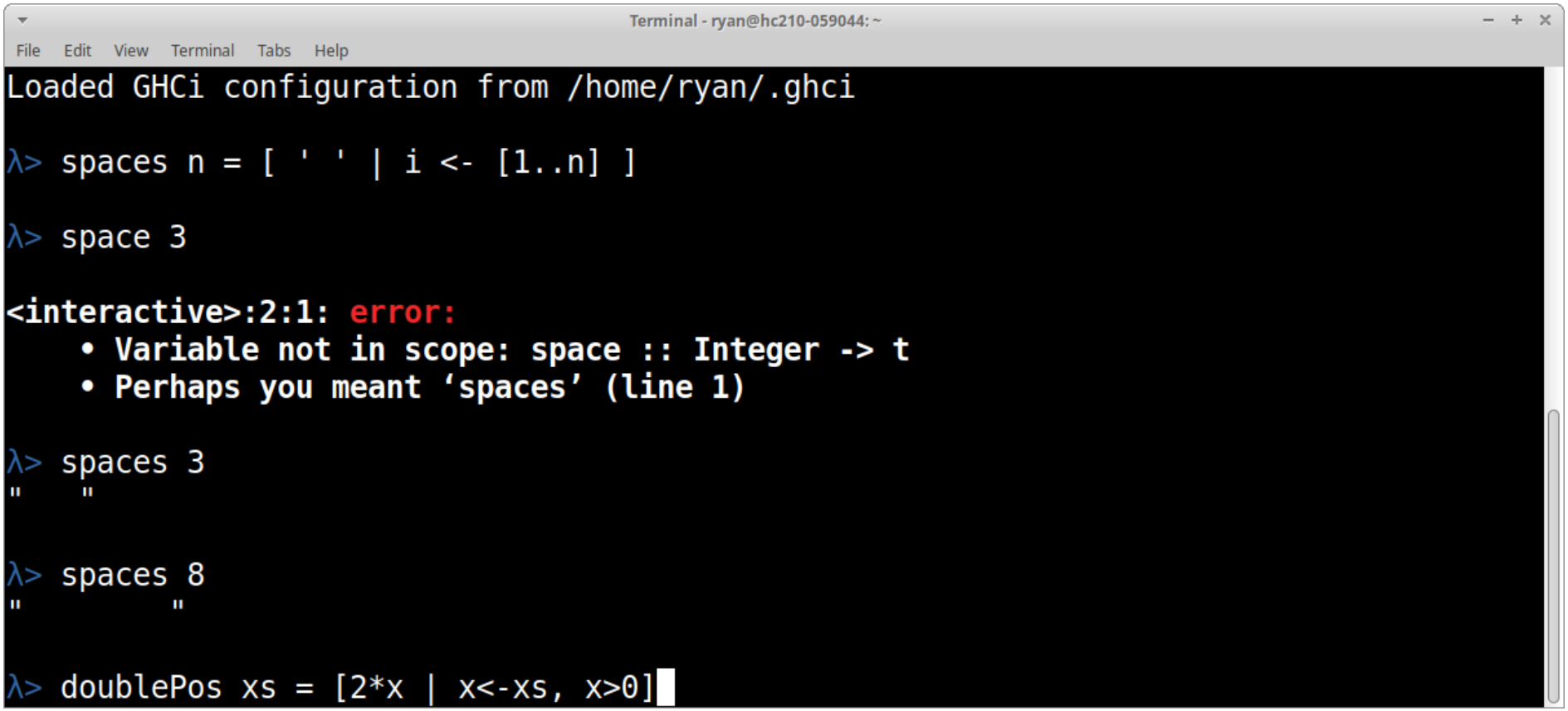
List Comprehension and Functions



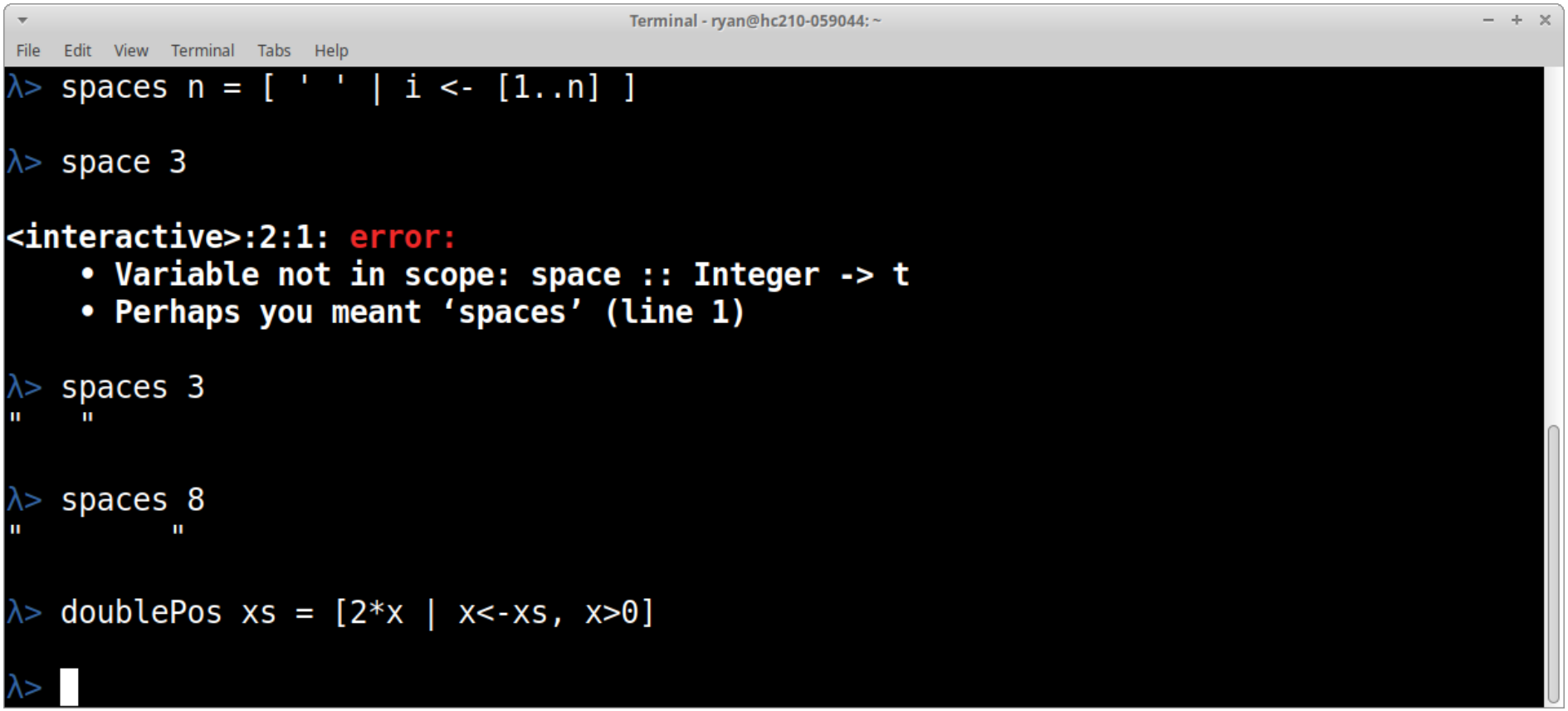
List Comprehension and Functions



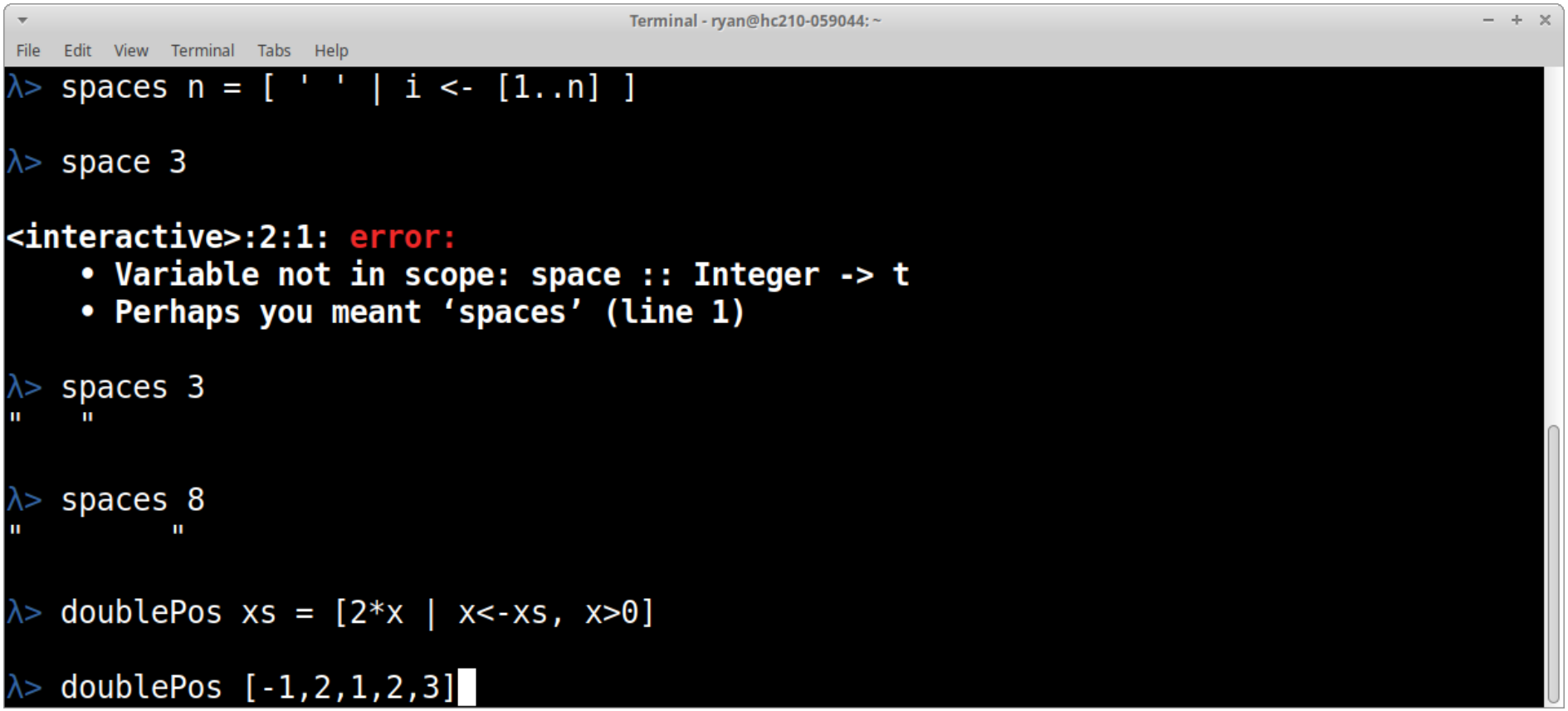
List Comprehension and Functions



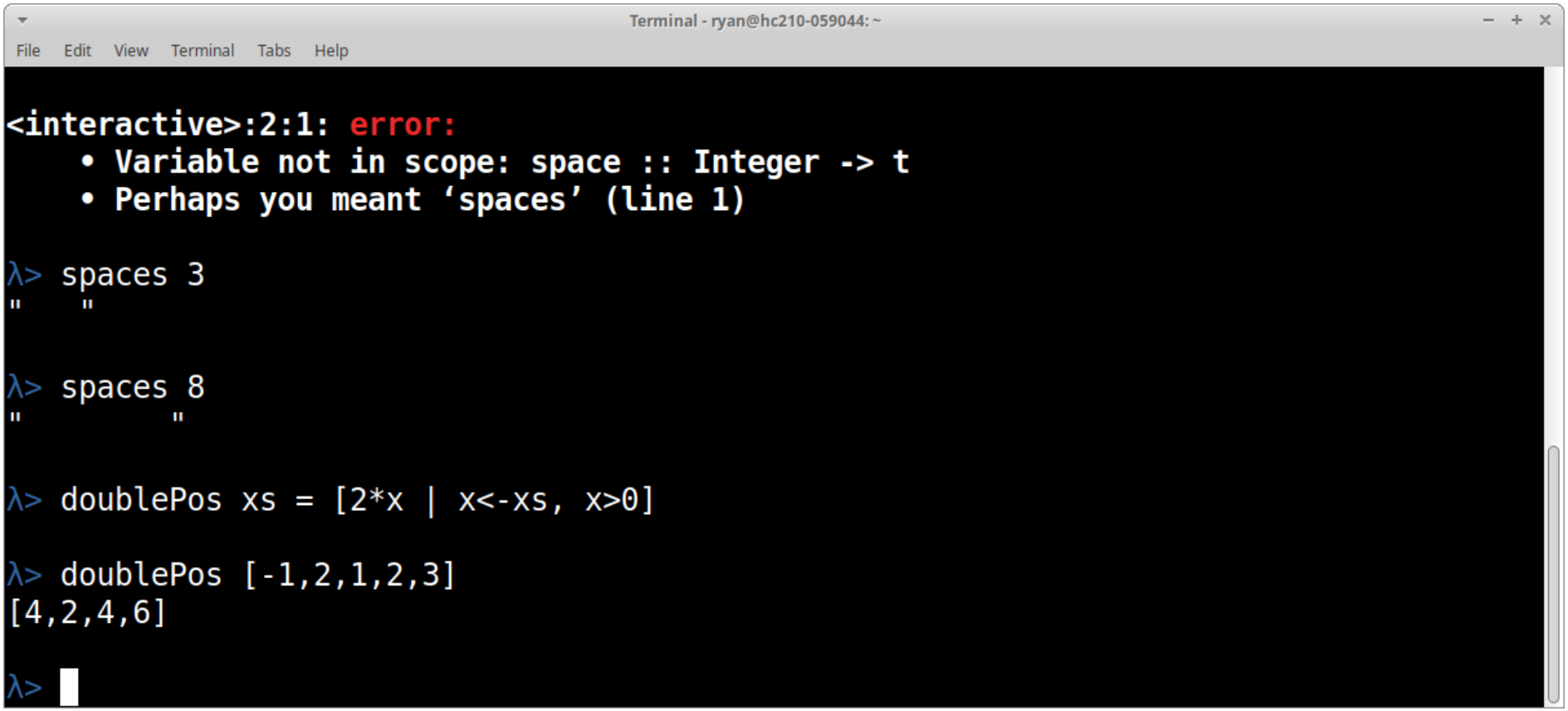
List Comprehension and Functions



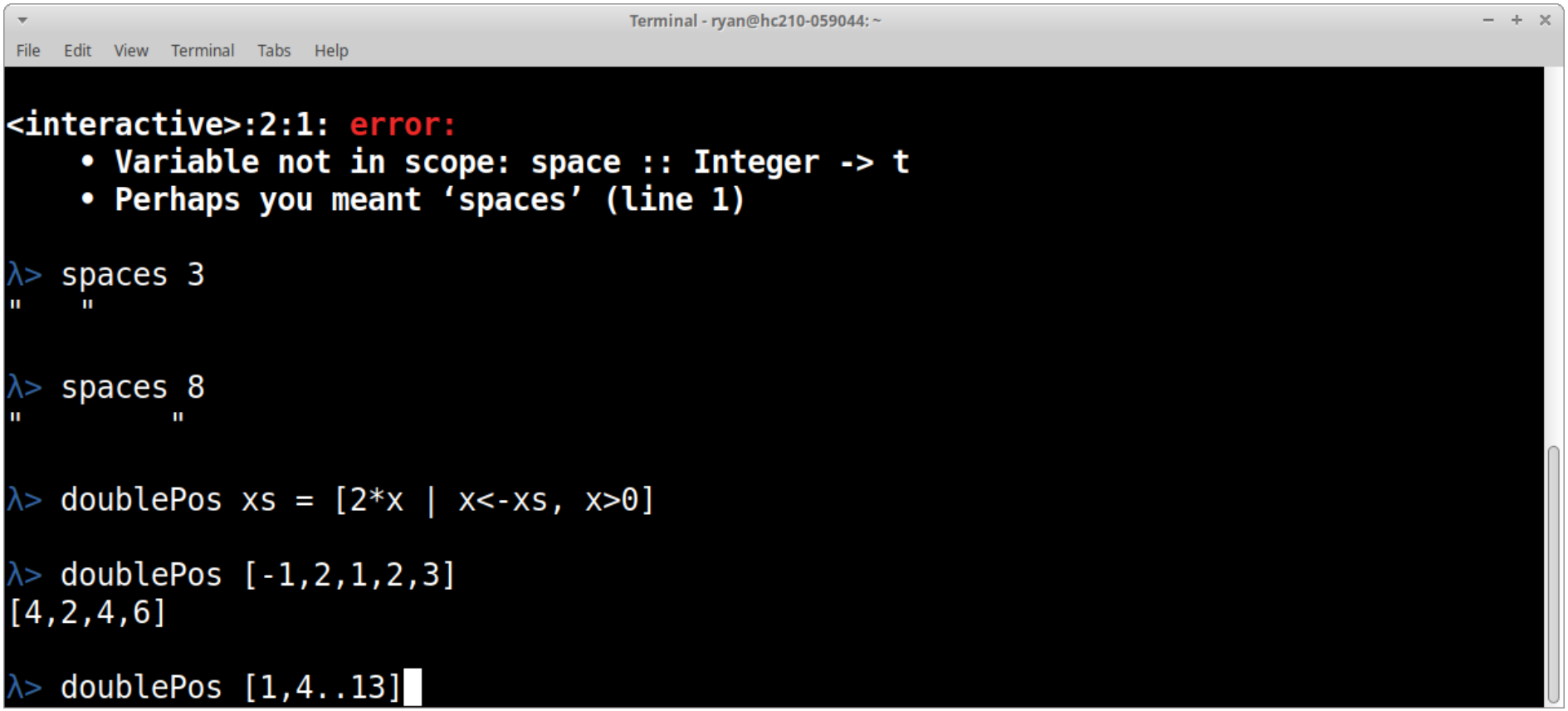
List Comprehension and Functions



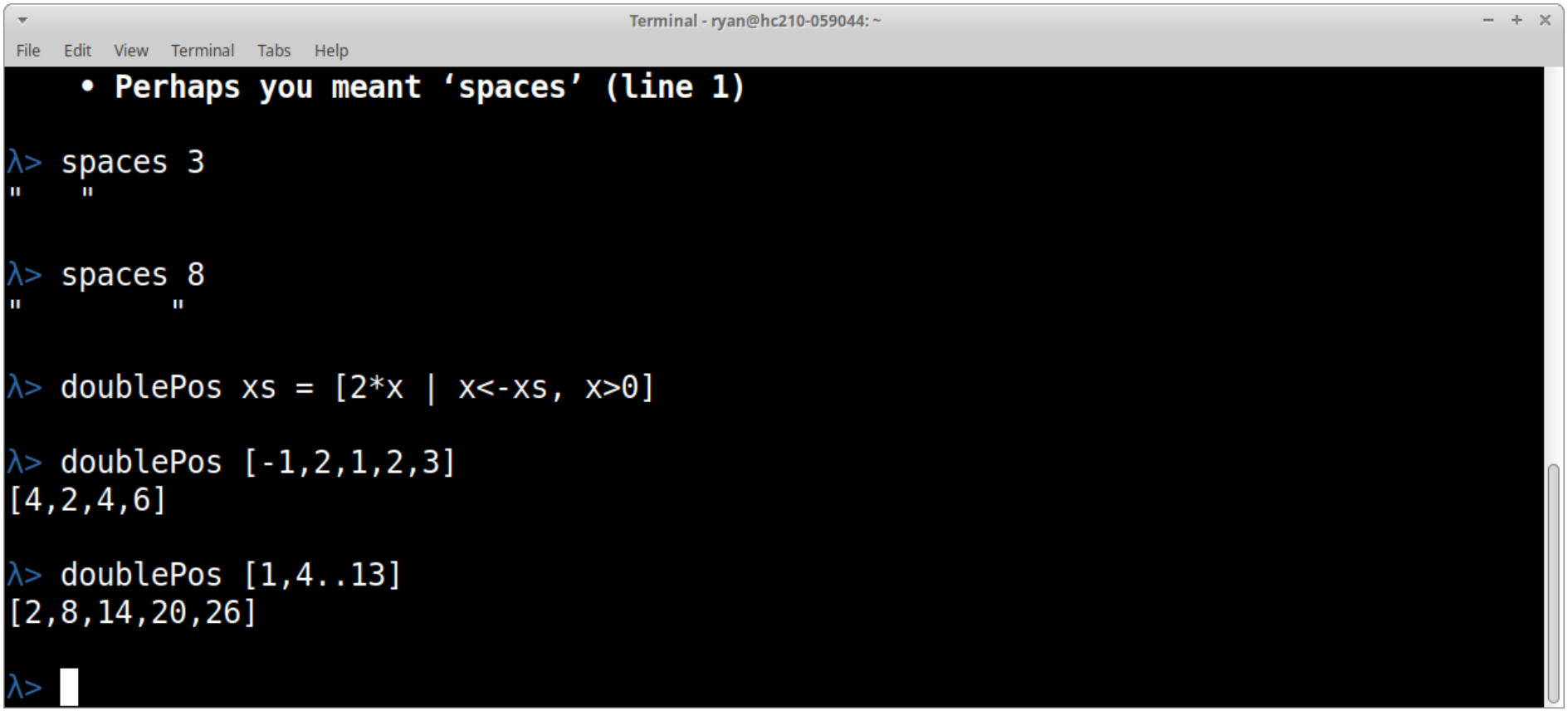
List Comprehension and Functions



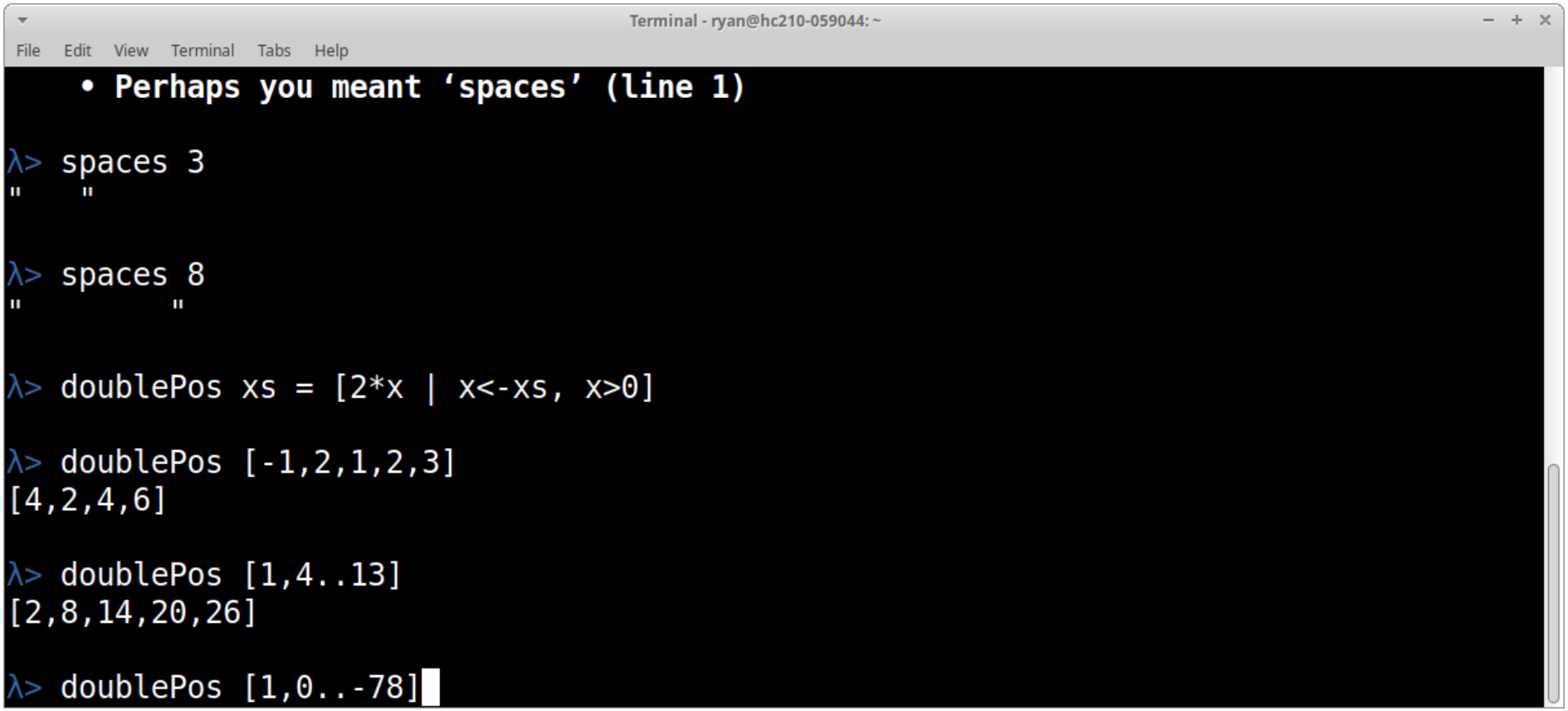
List Comprehension and Functions



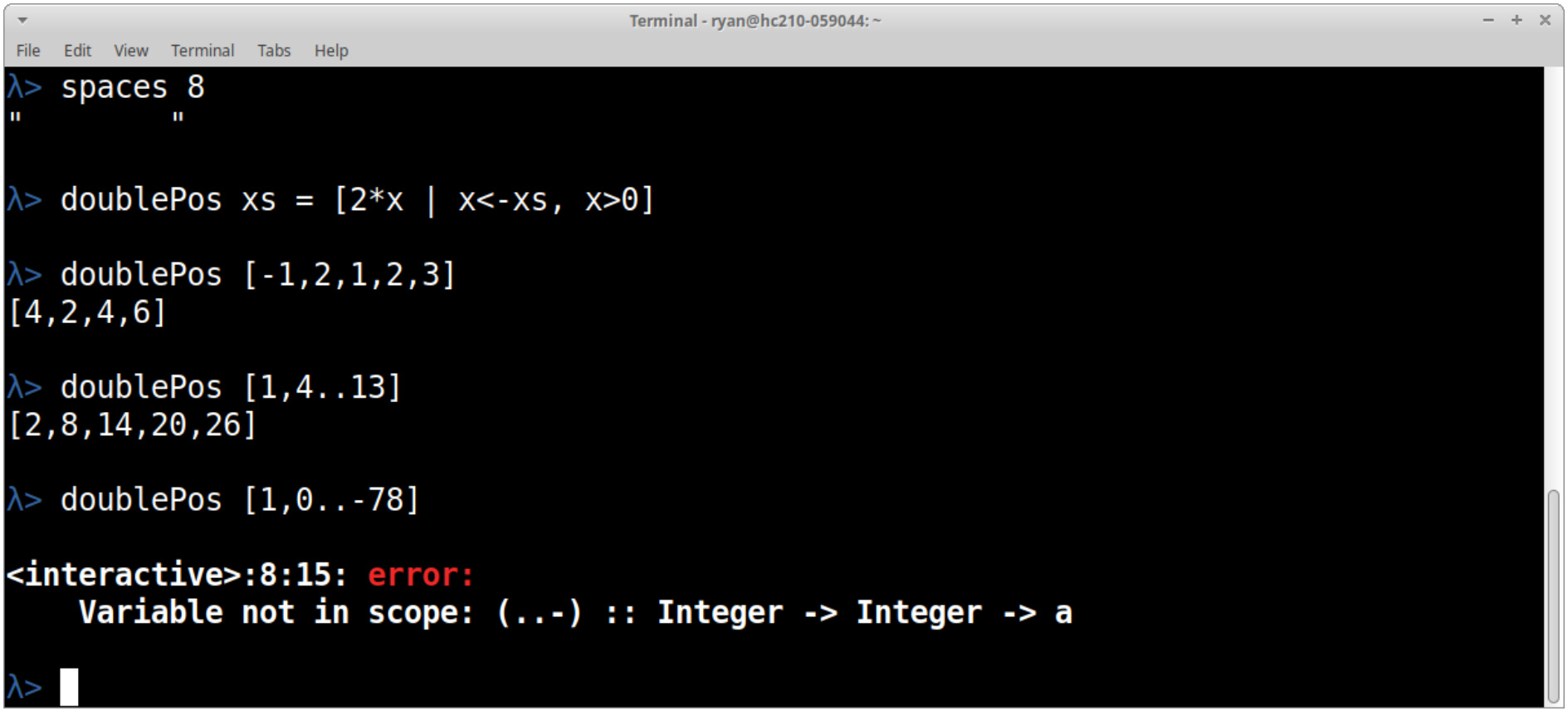
List Comprehension and Functions



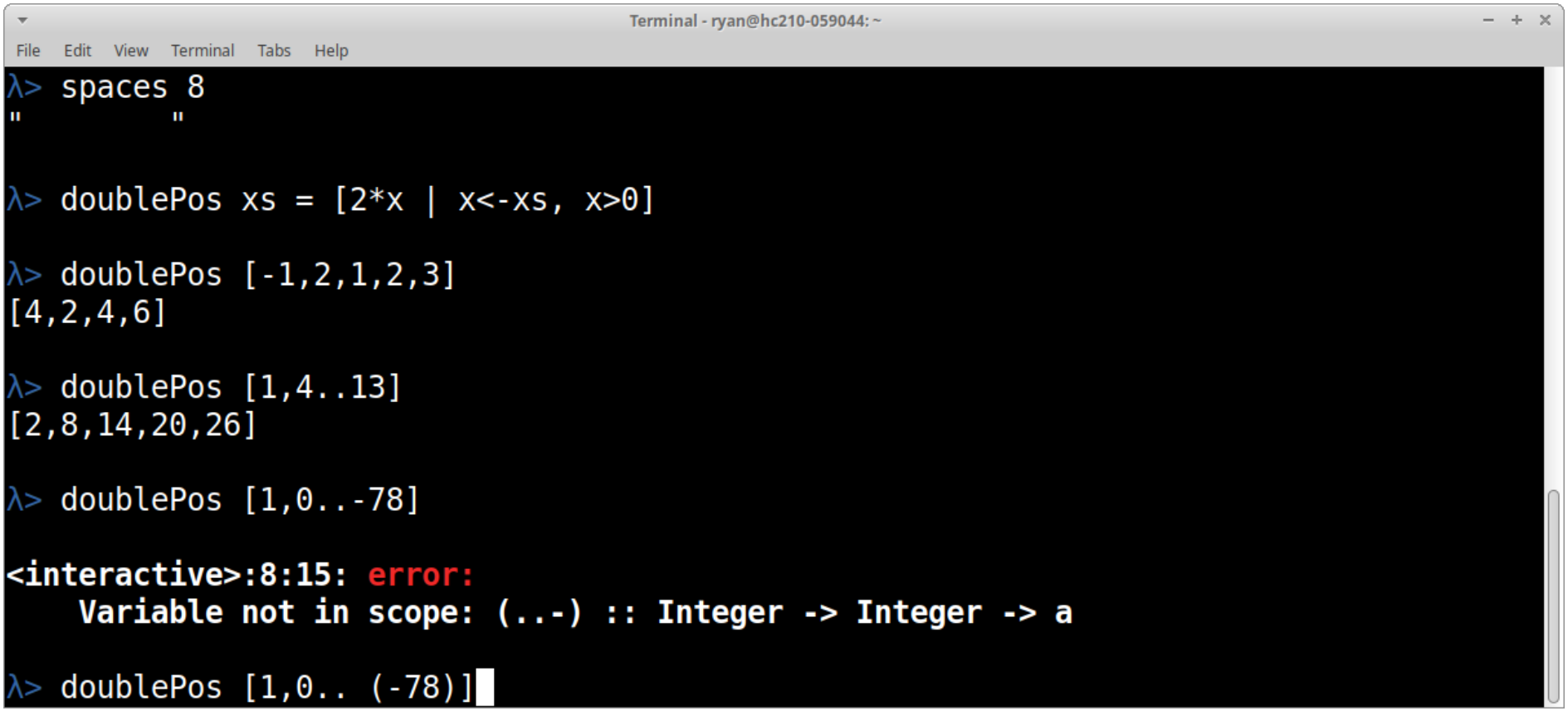
List Comprehension and Functions



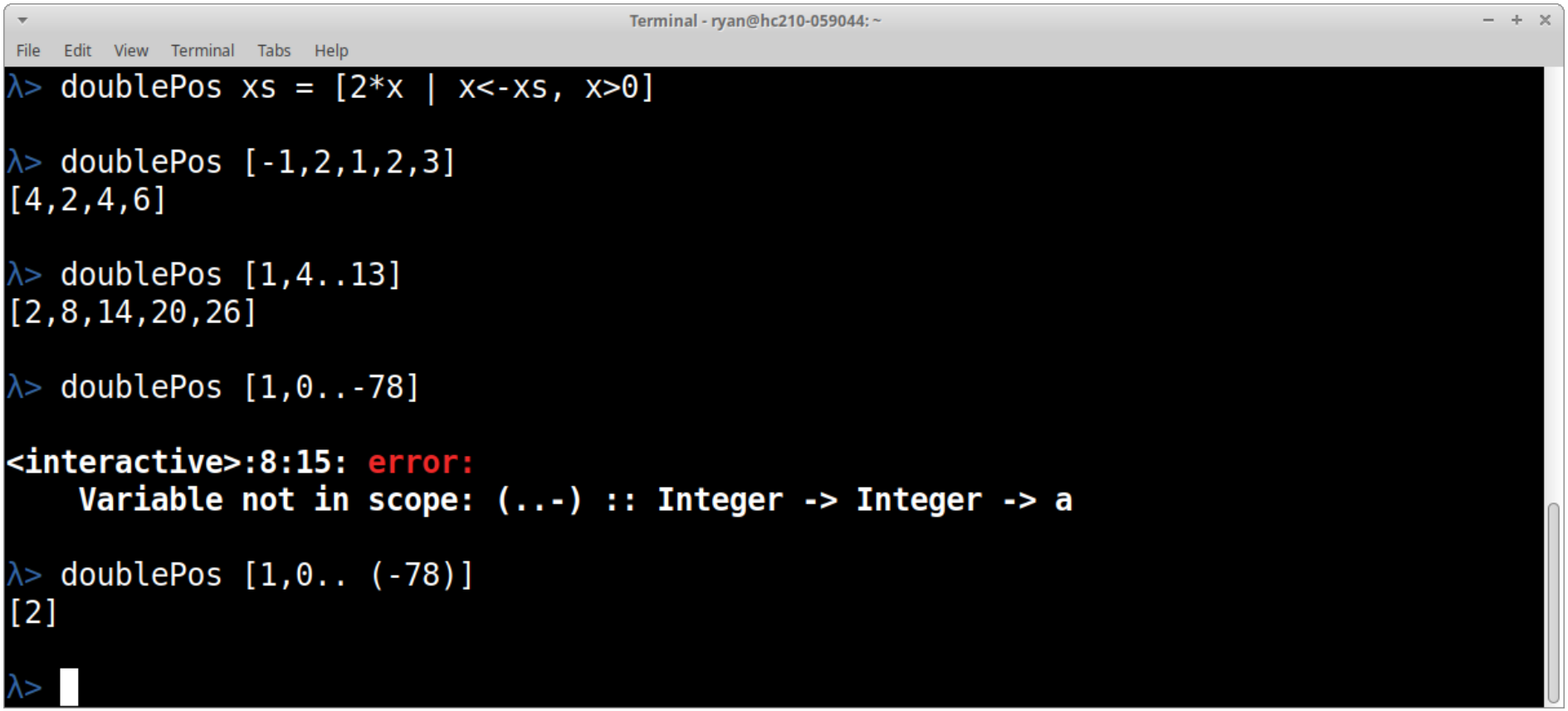
List Comprehension and Functions



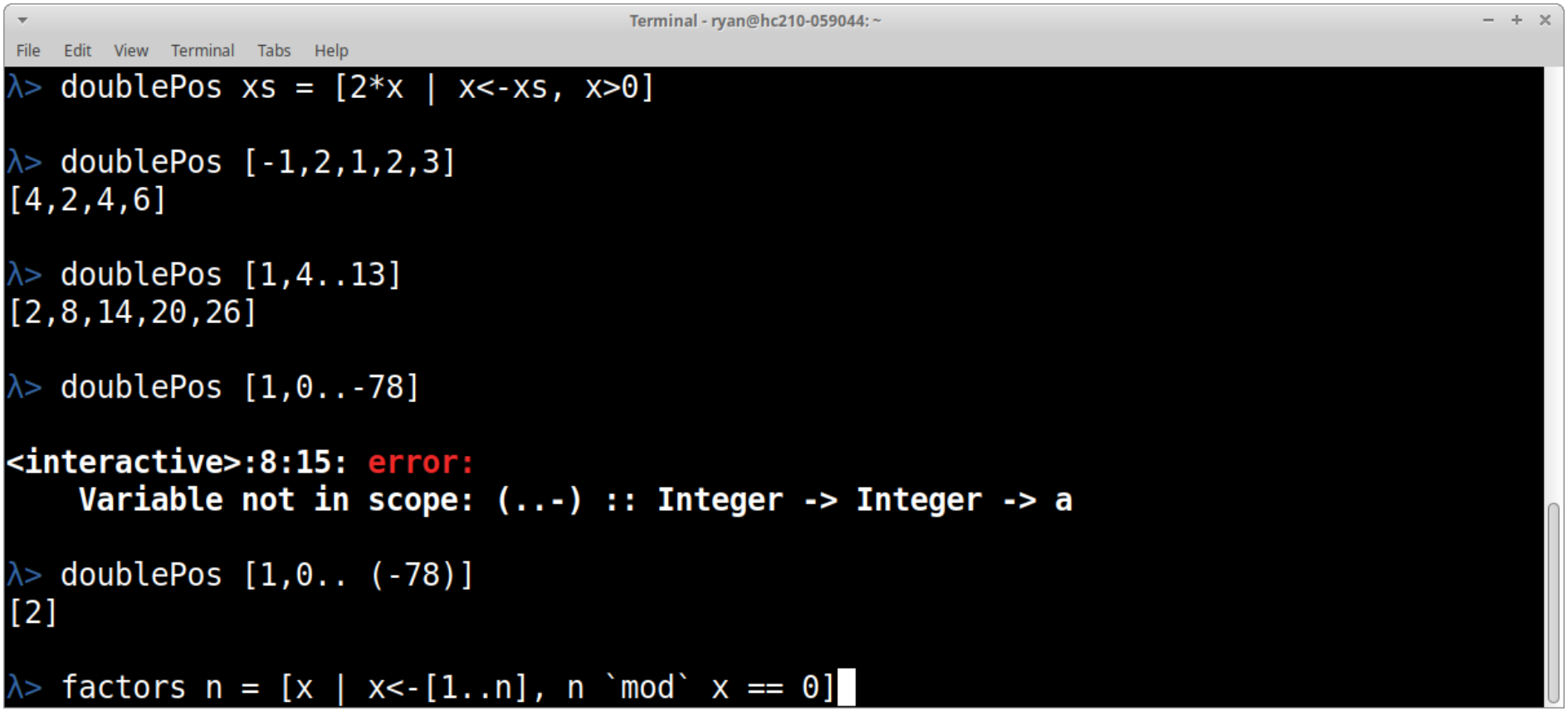
List Comprehension and Functions



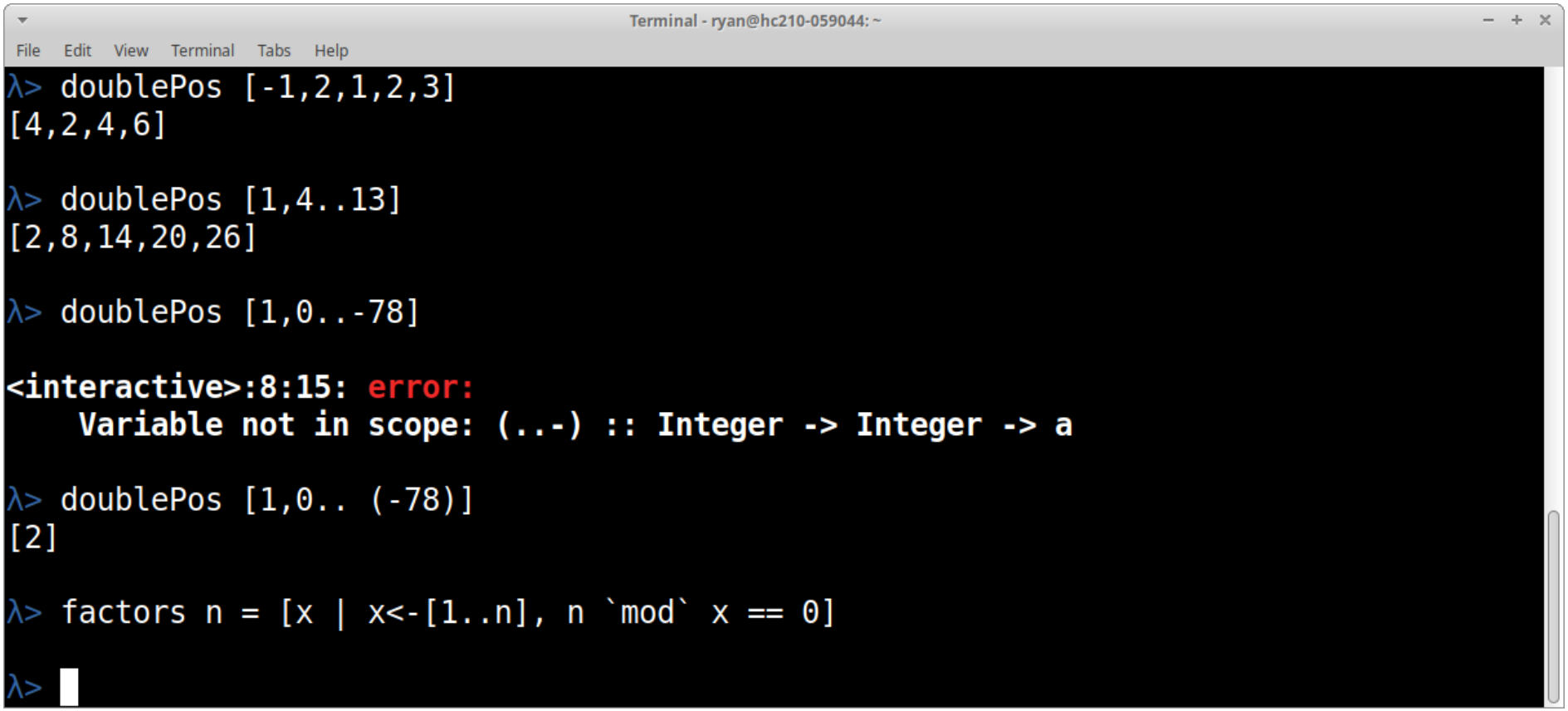
List Comprehension and Functions



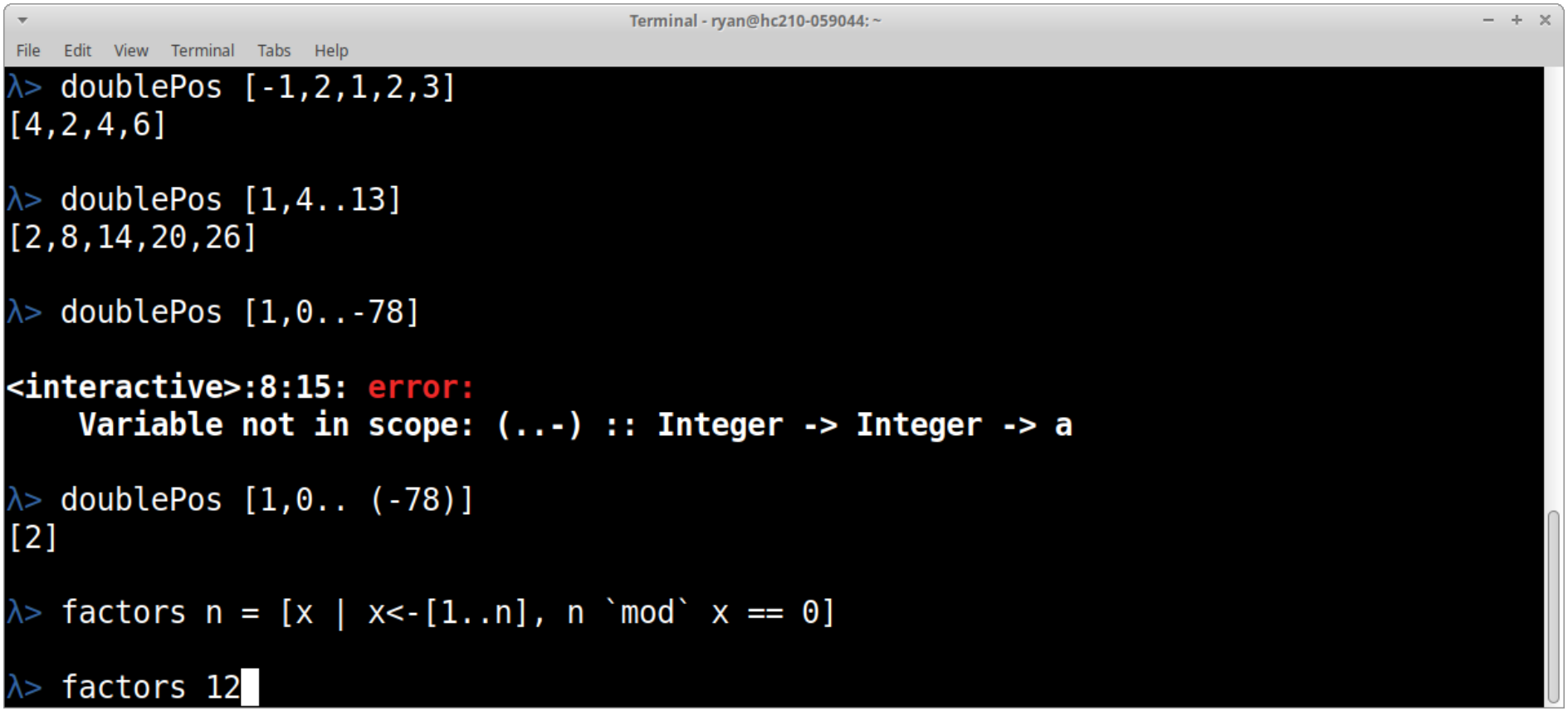
List Comprehension and Functions



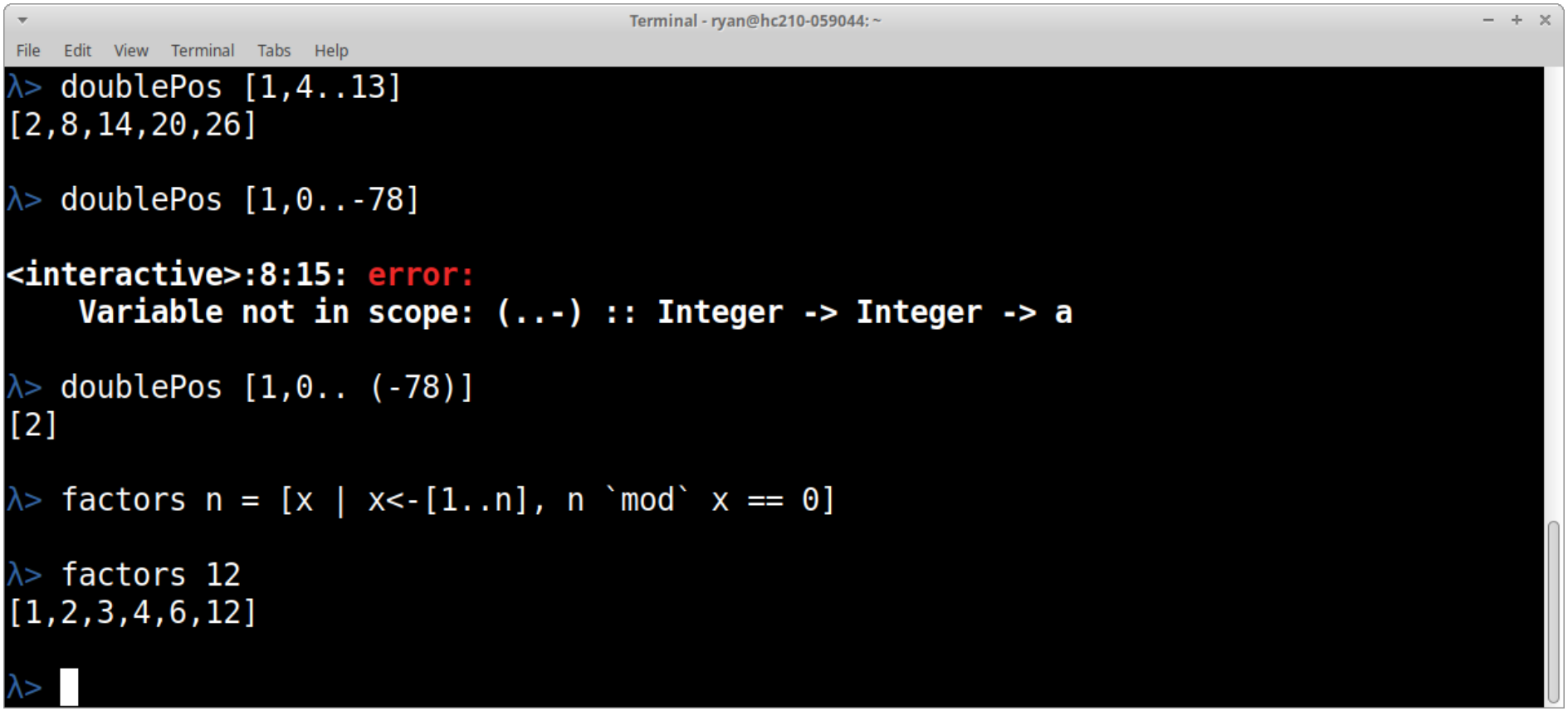
List Comprehension and Functions



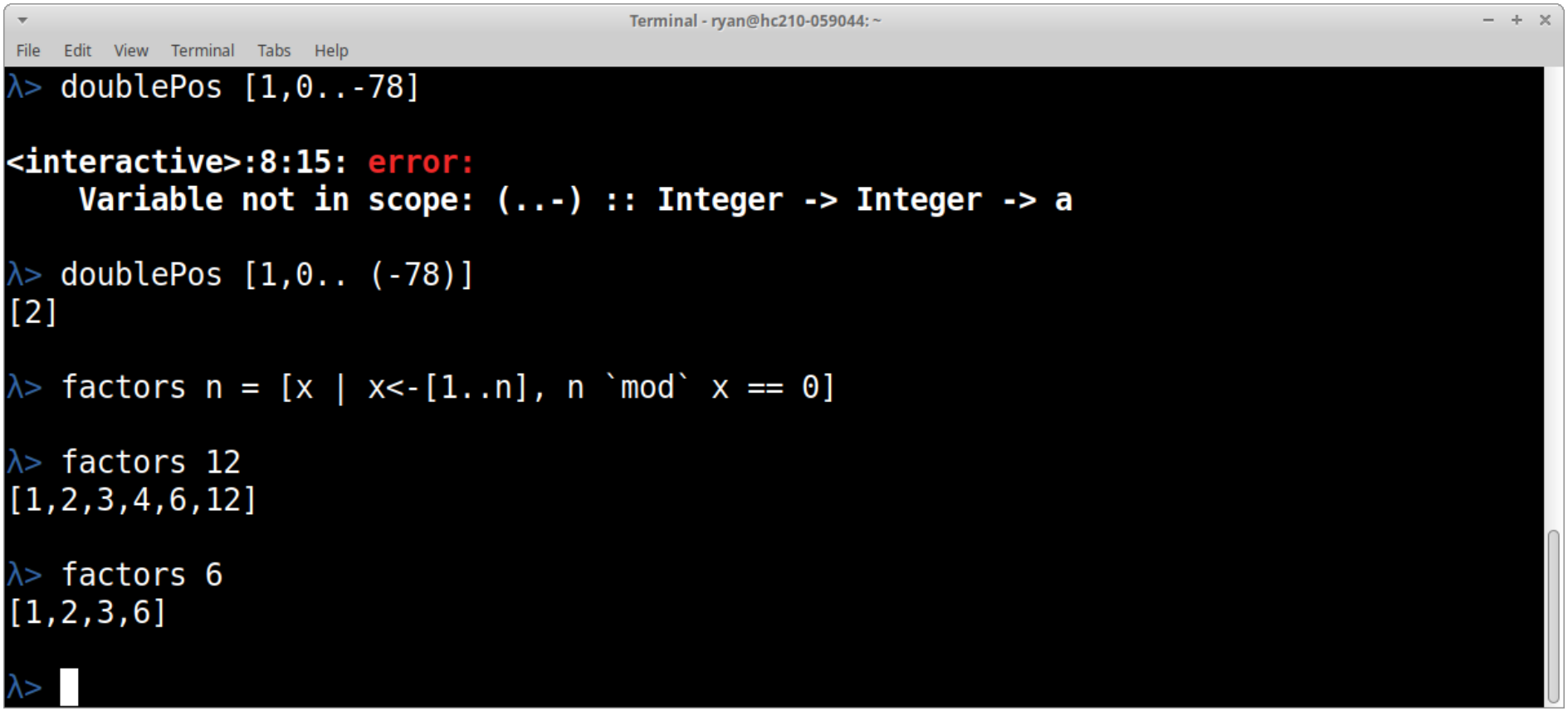
List Comprehension and Functions



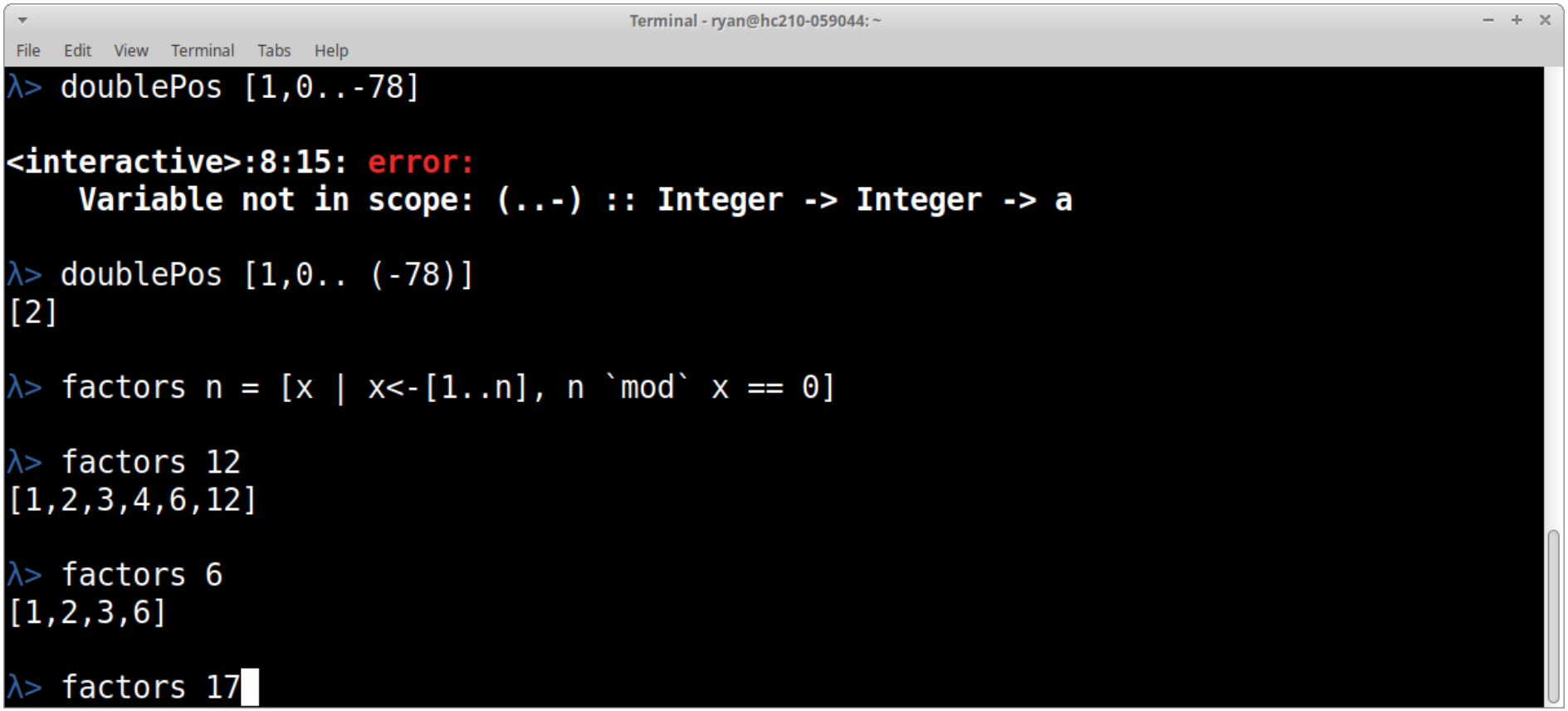
List Comprehension and Functions



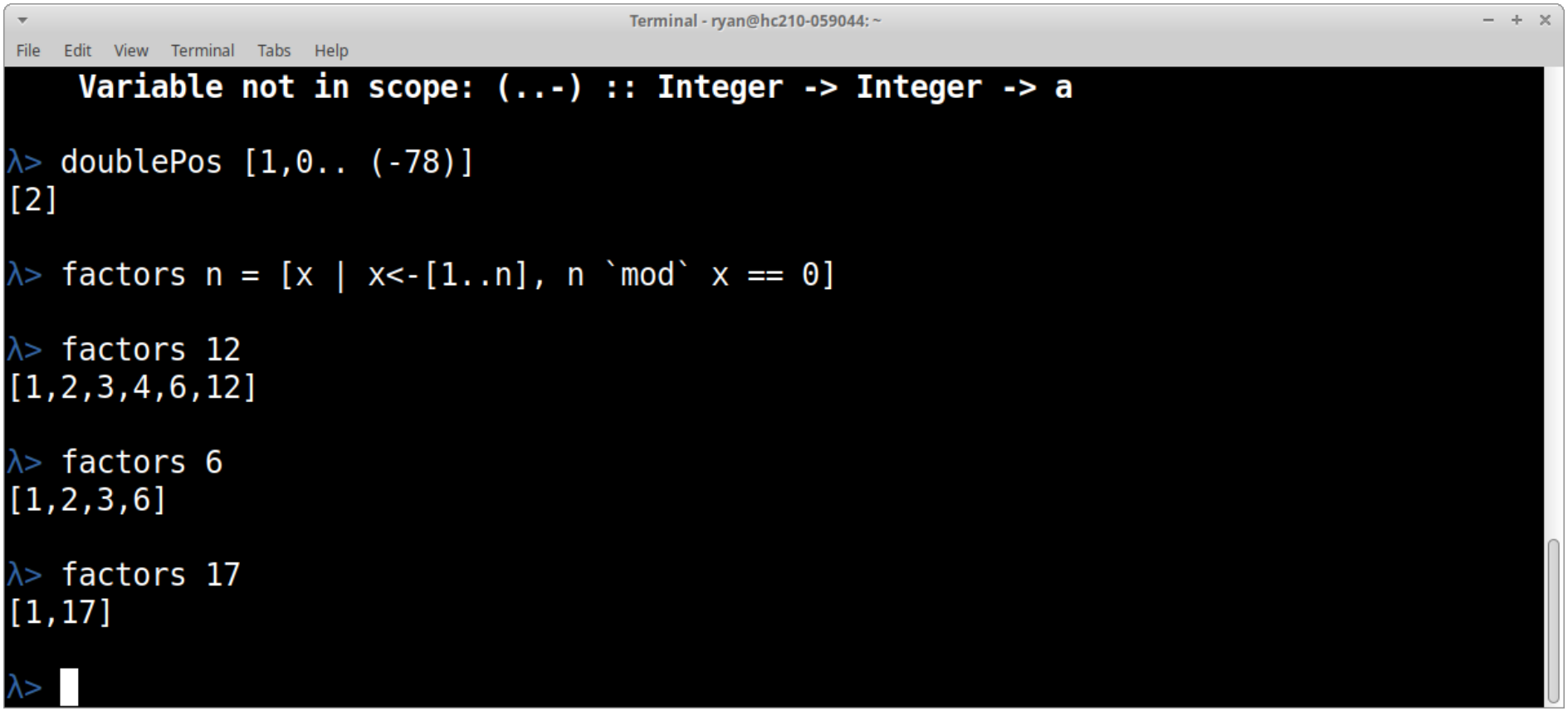
List Comprehension and Functions



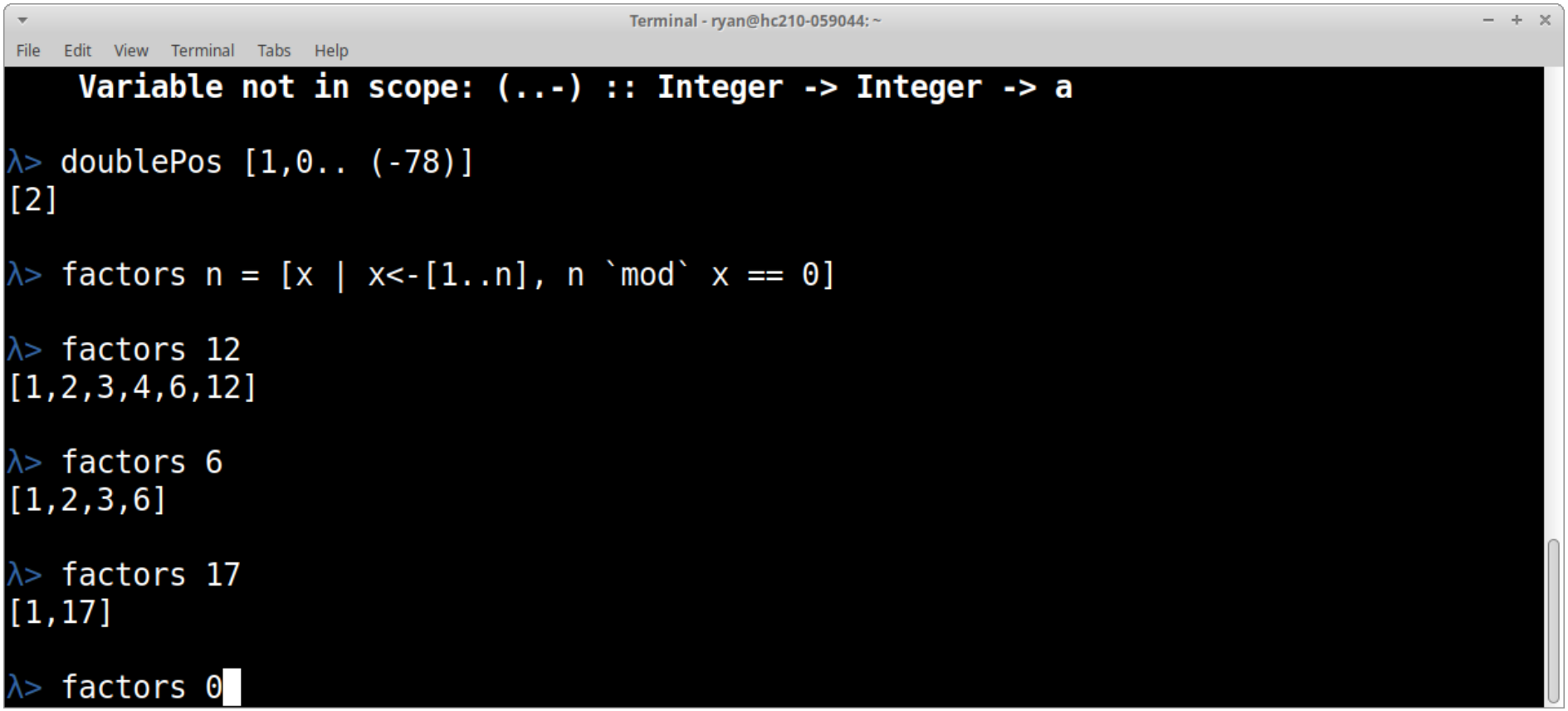
List Comprehension and Functions



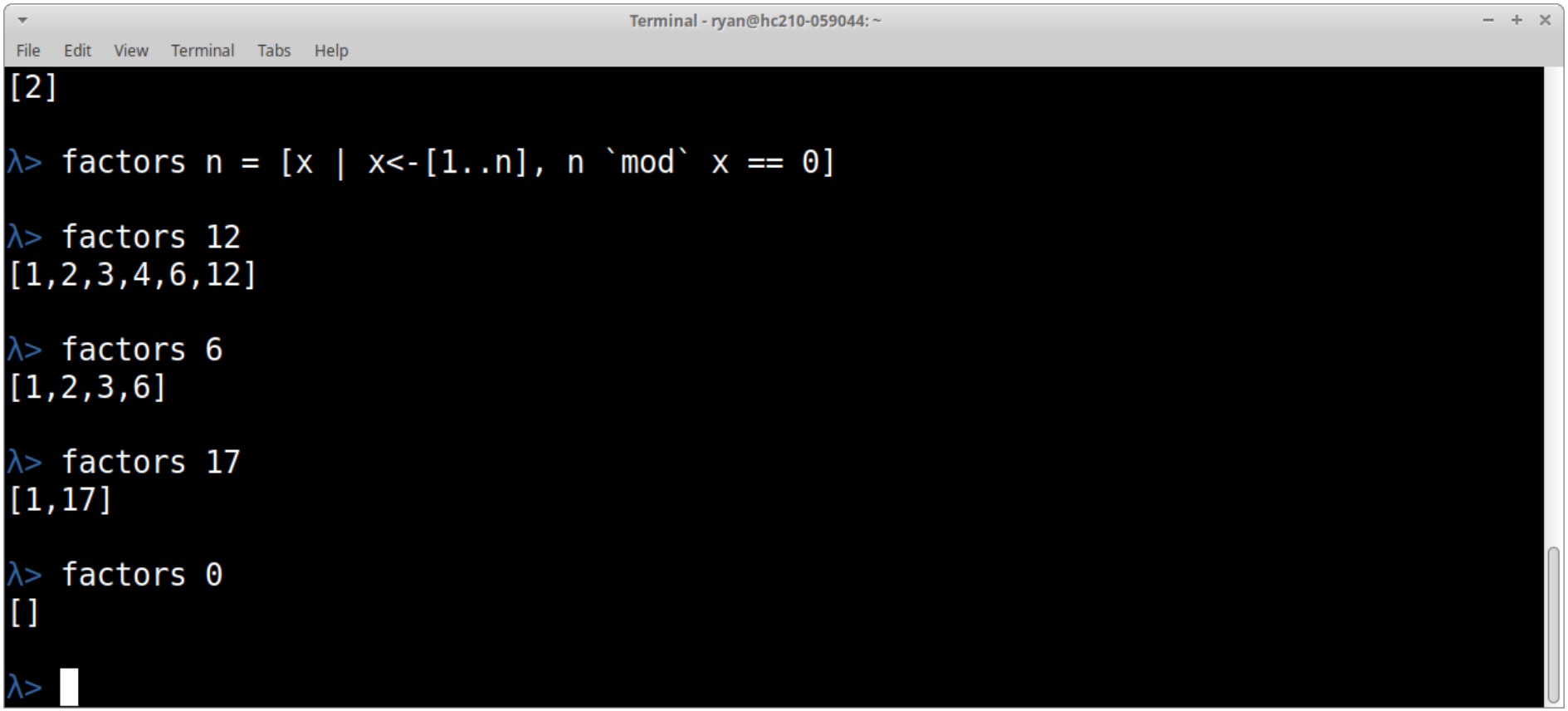
List Comprehension and Functions



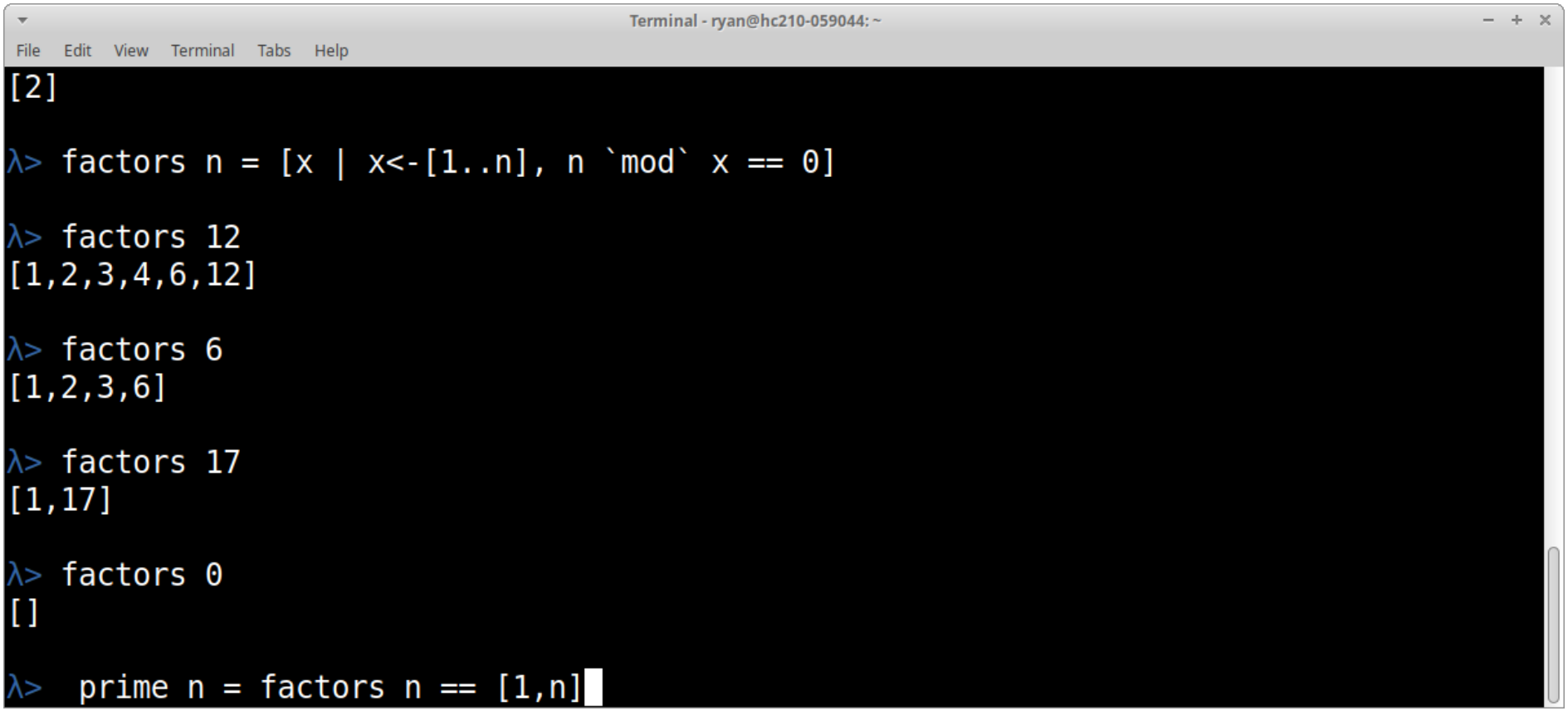
List Comprehension and Functions



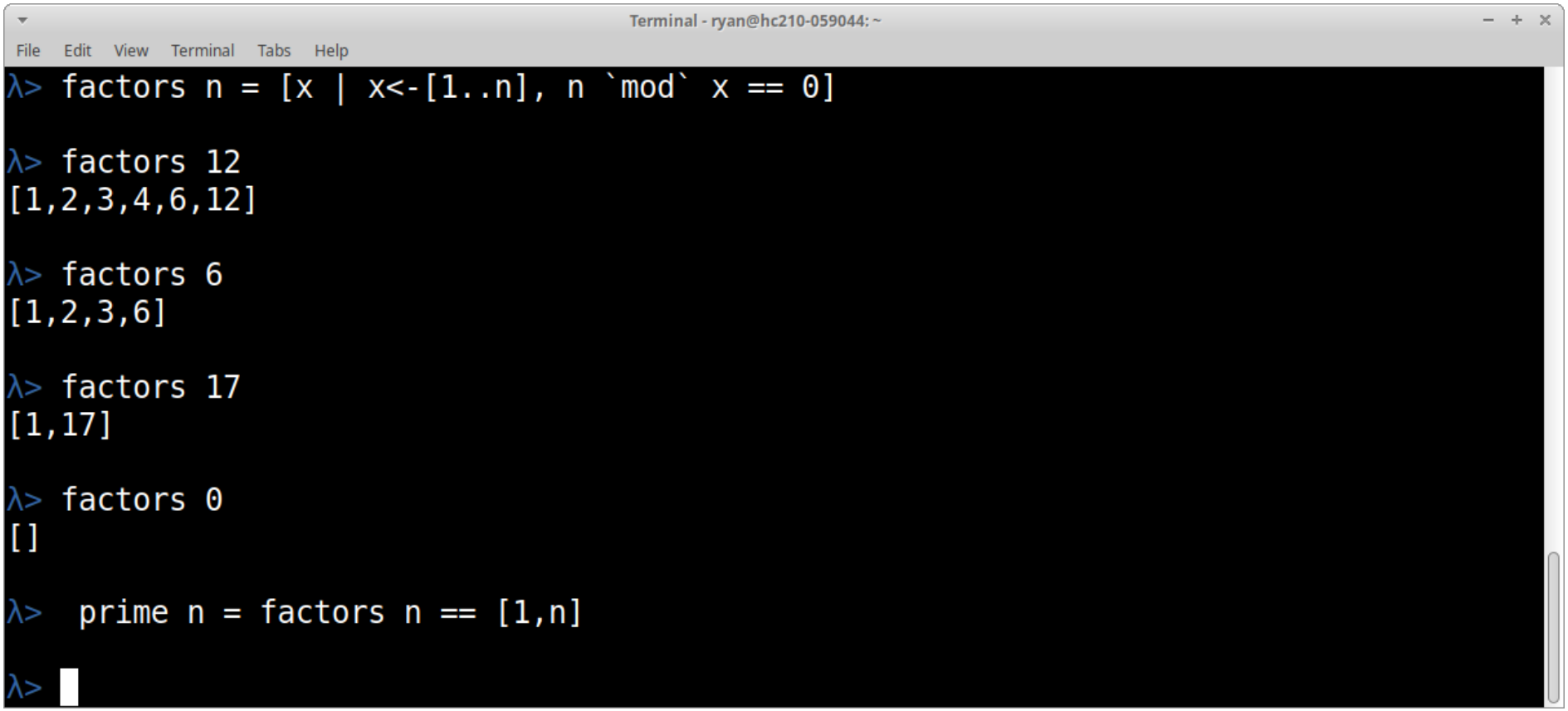
List Comprehension and Functions



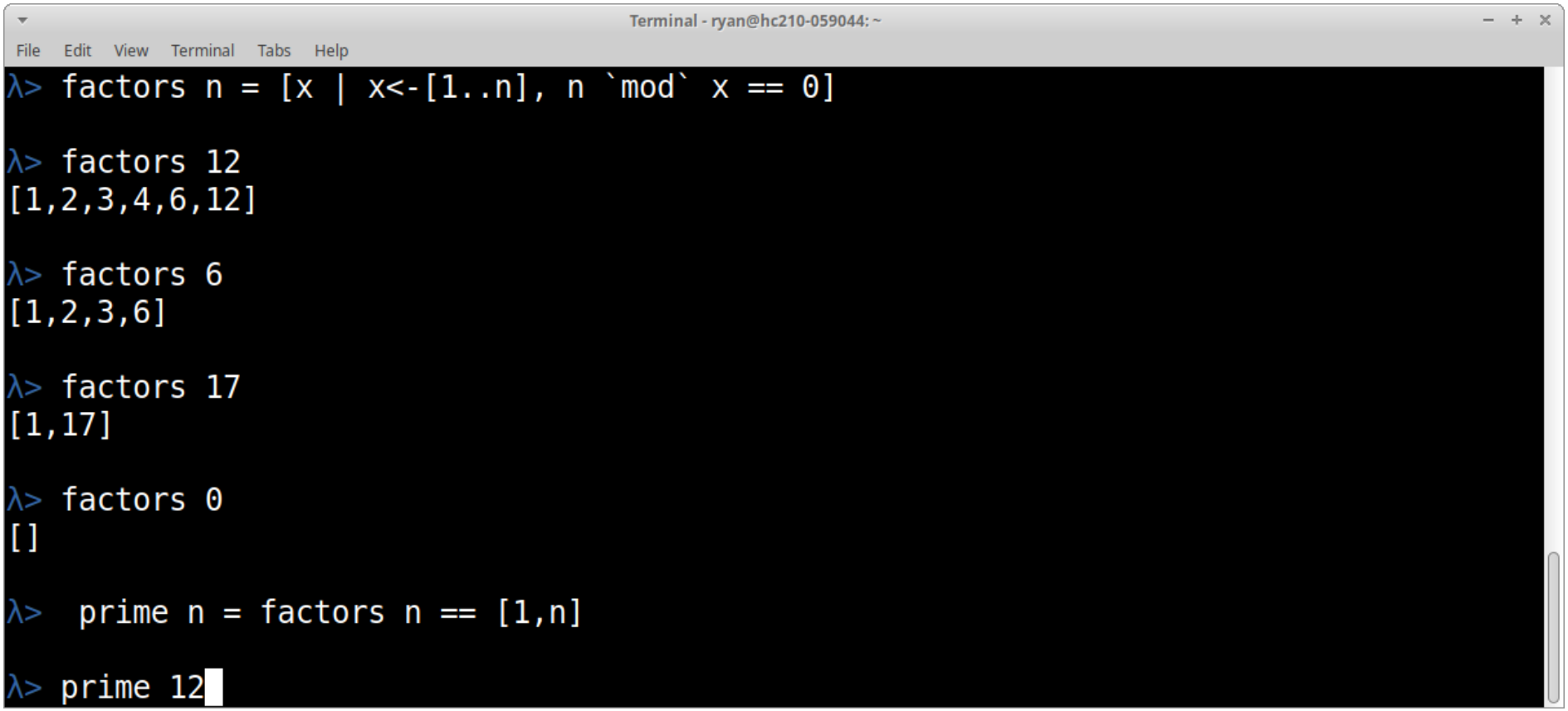
List Comprehension and Functions



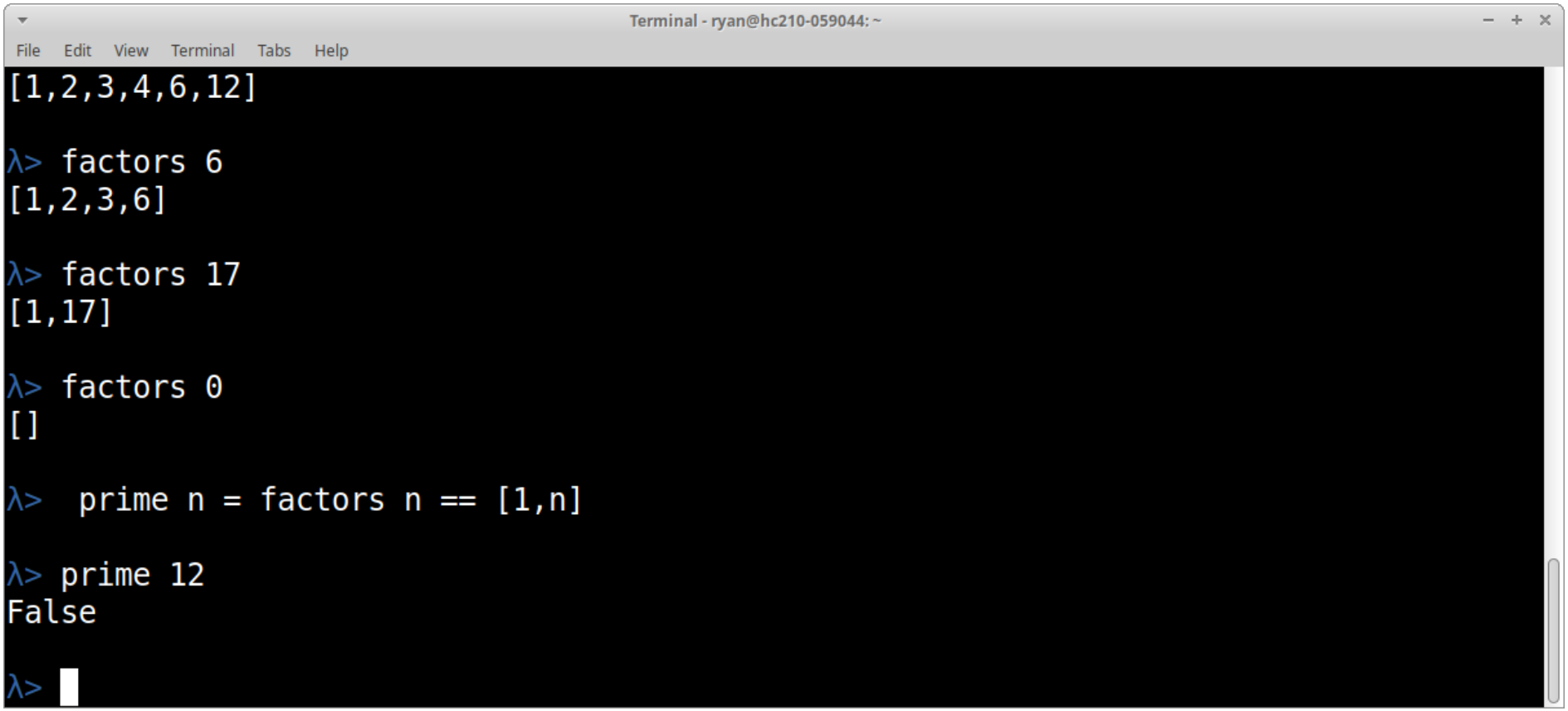
List Comprehension and Functions



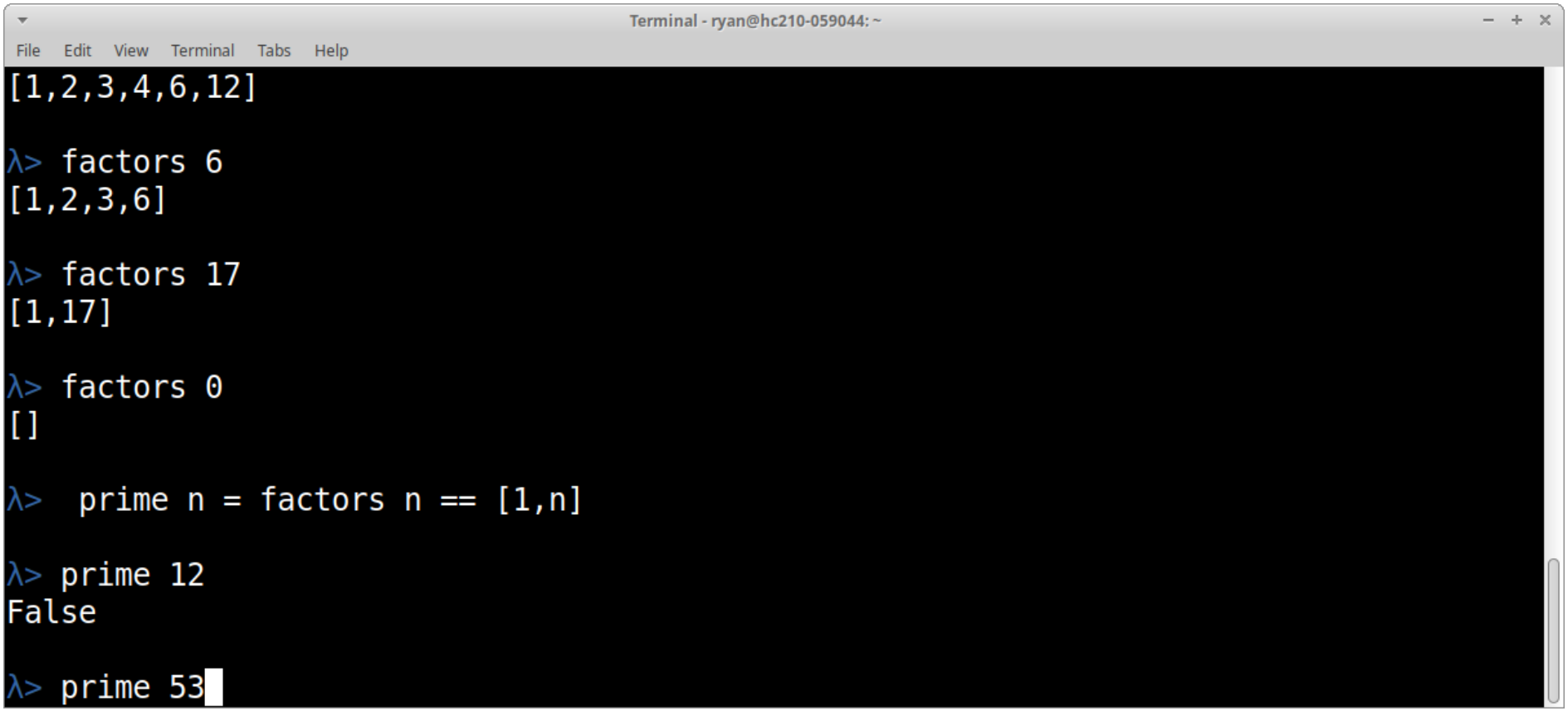
List Comprehension and Functions



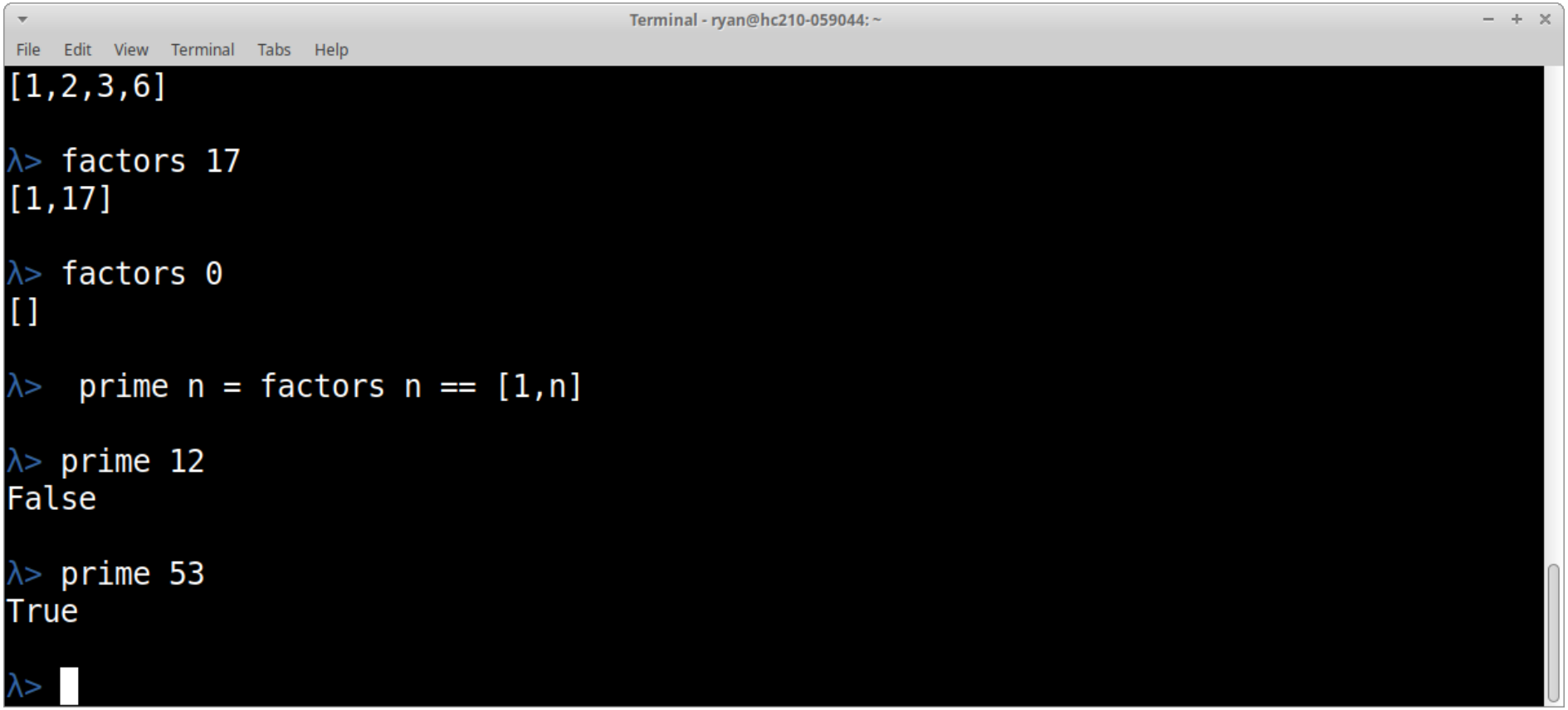
List Comprehension and Functions



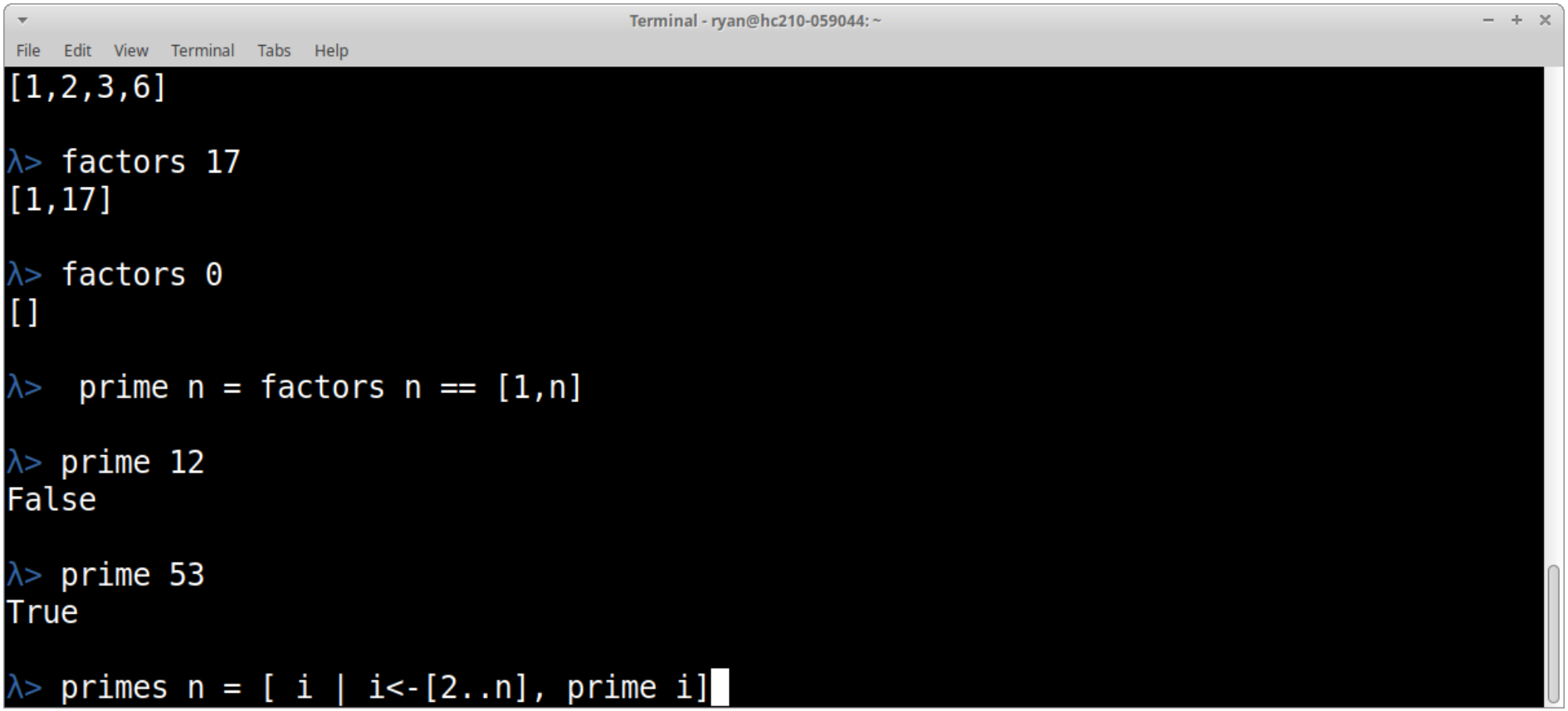
List Comprehension and Functions



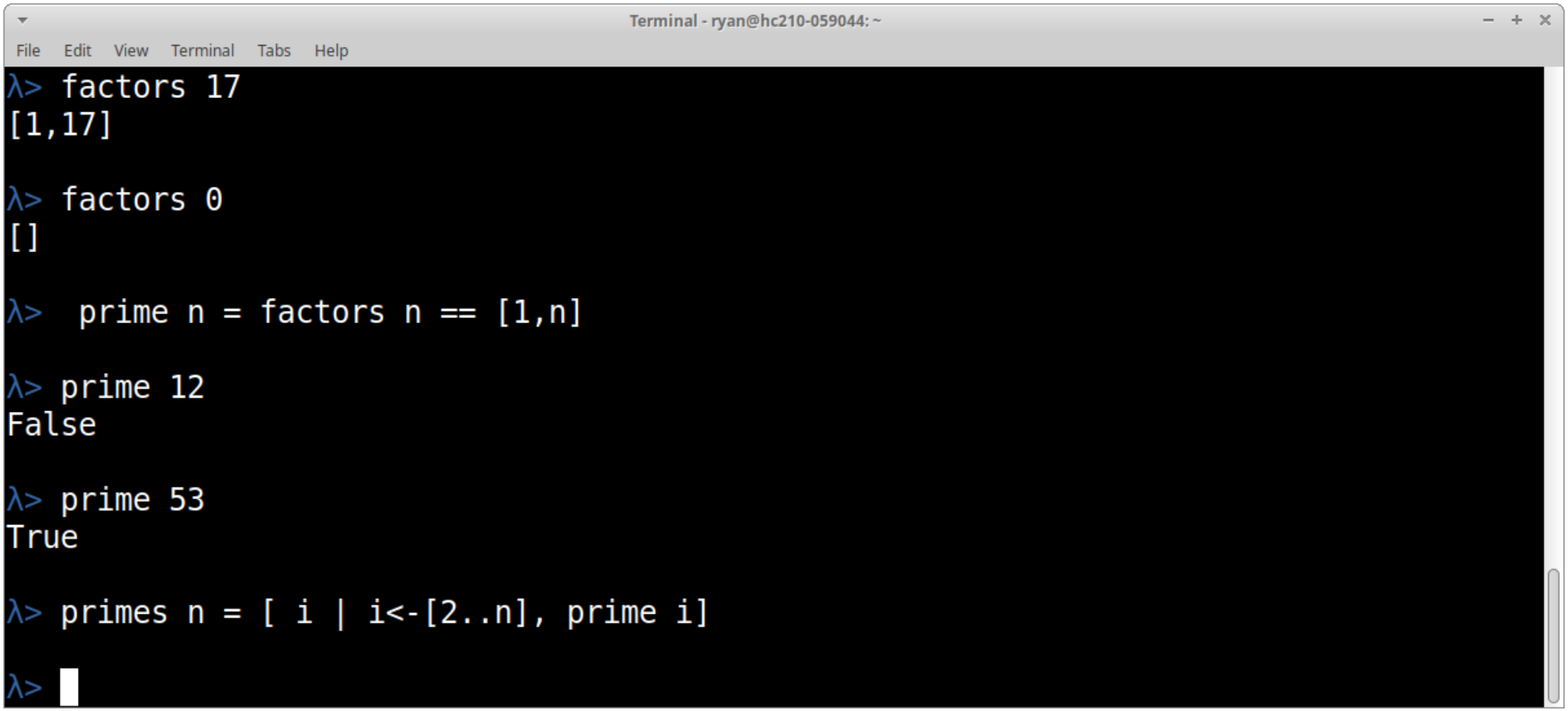
List Comprehension and Functions



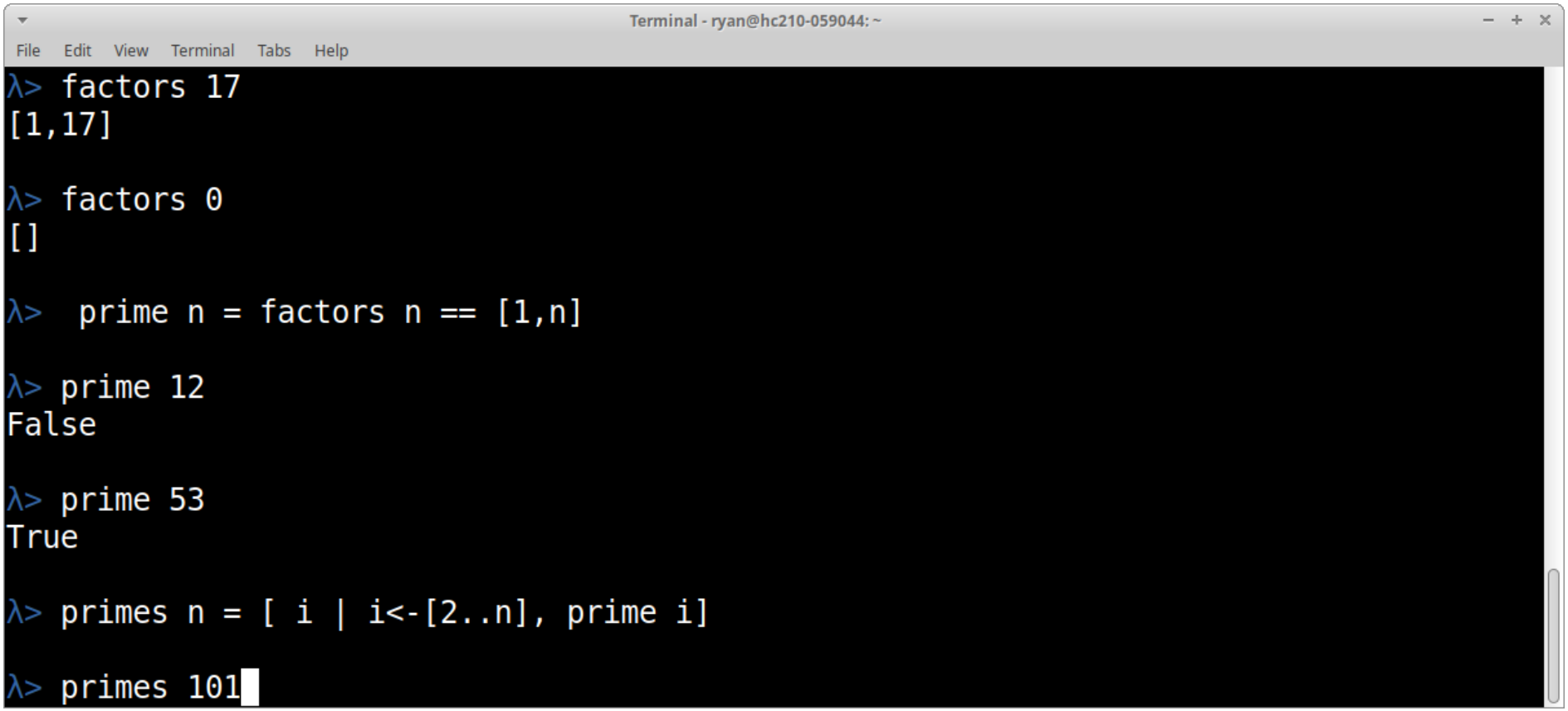
List Comprehension and Functions



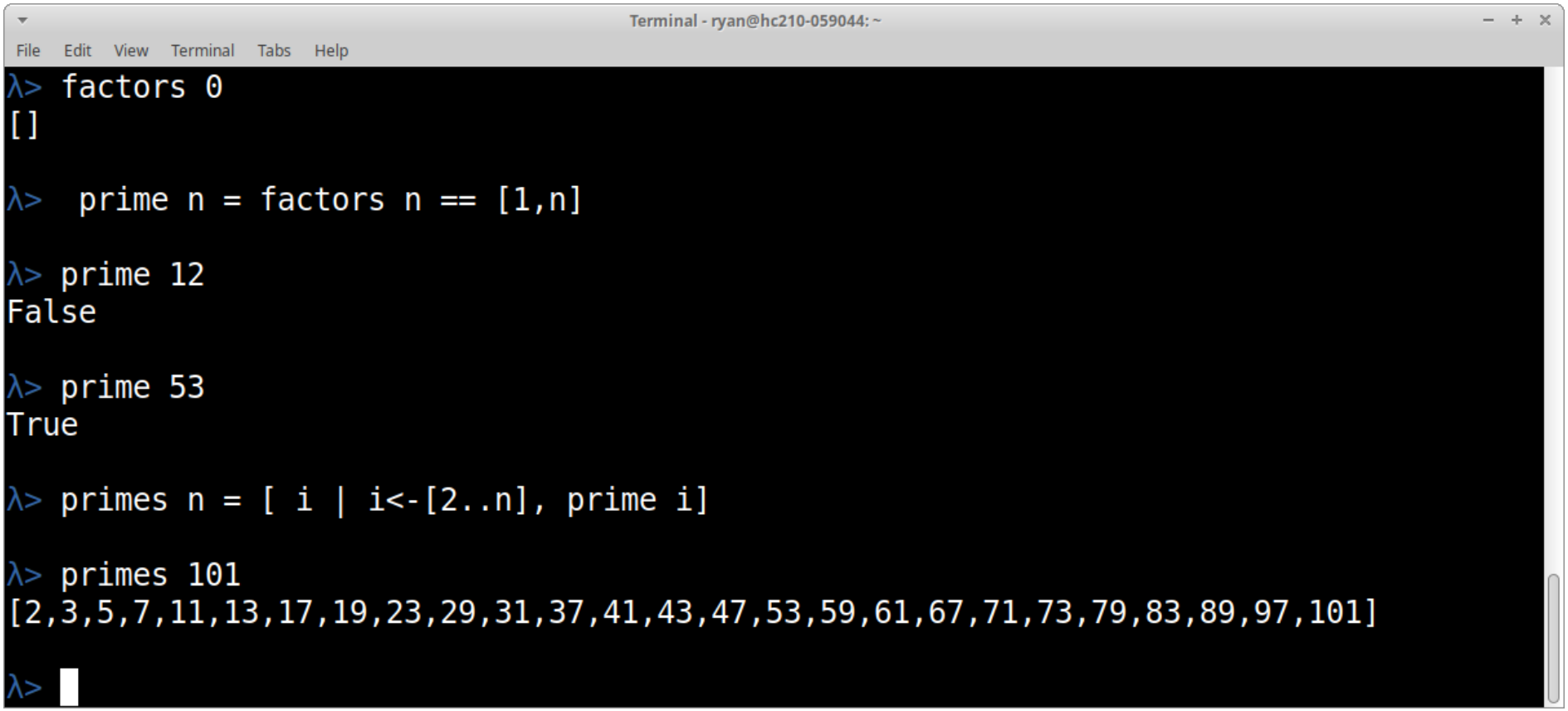
List Comprehension and Functions



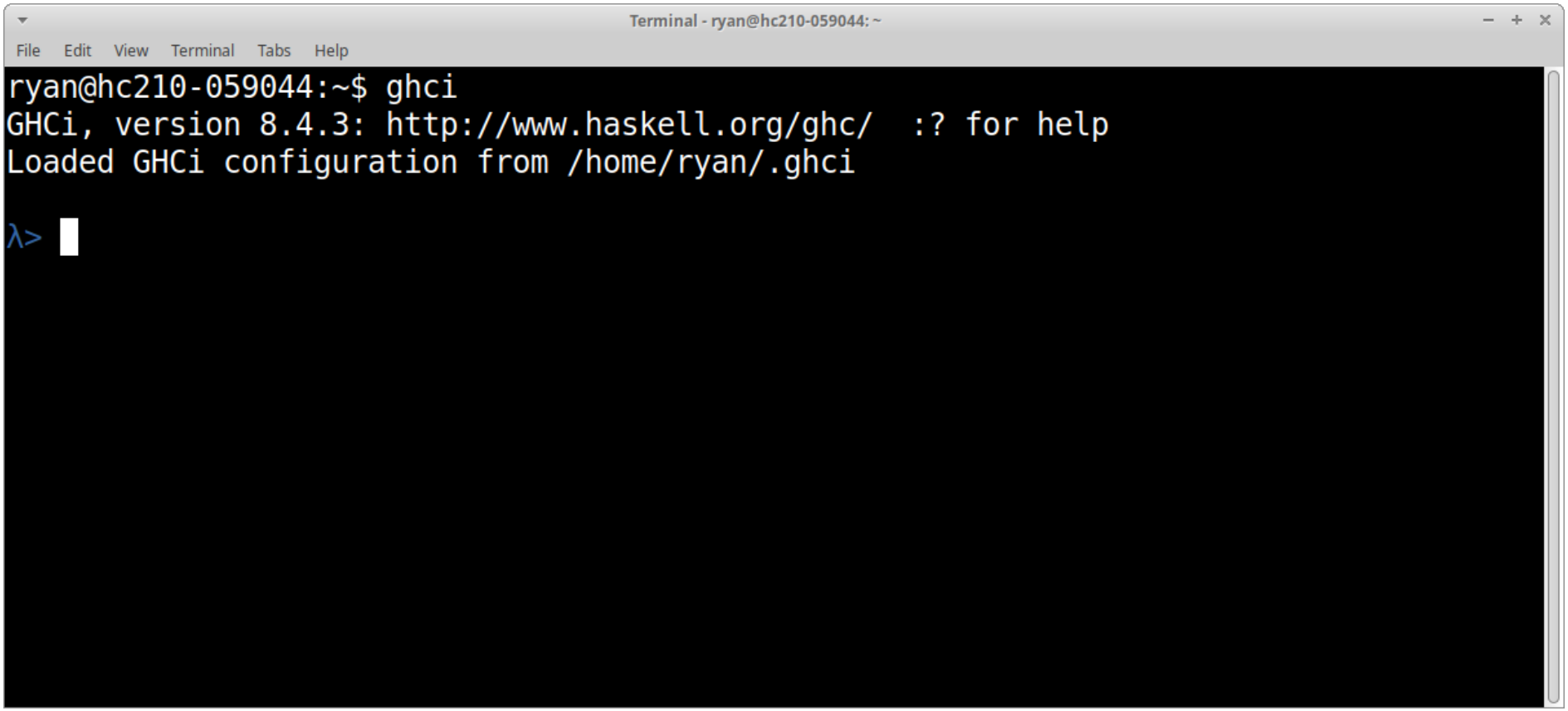
List Comprehension and Functions



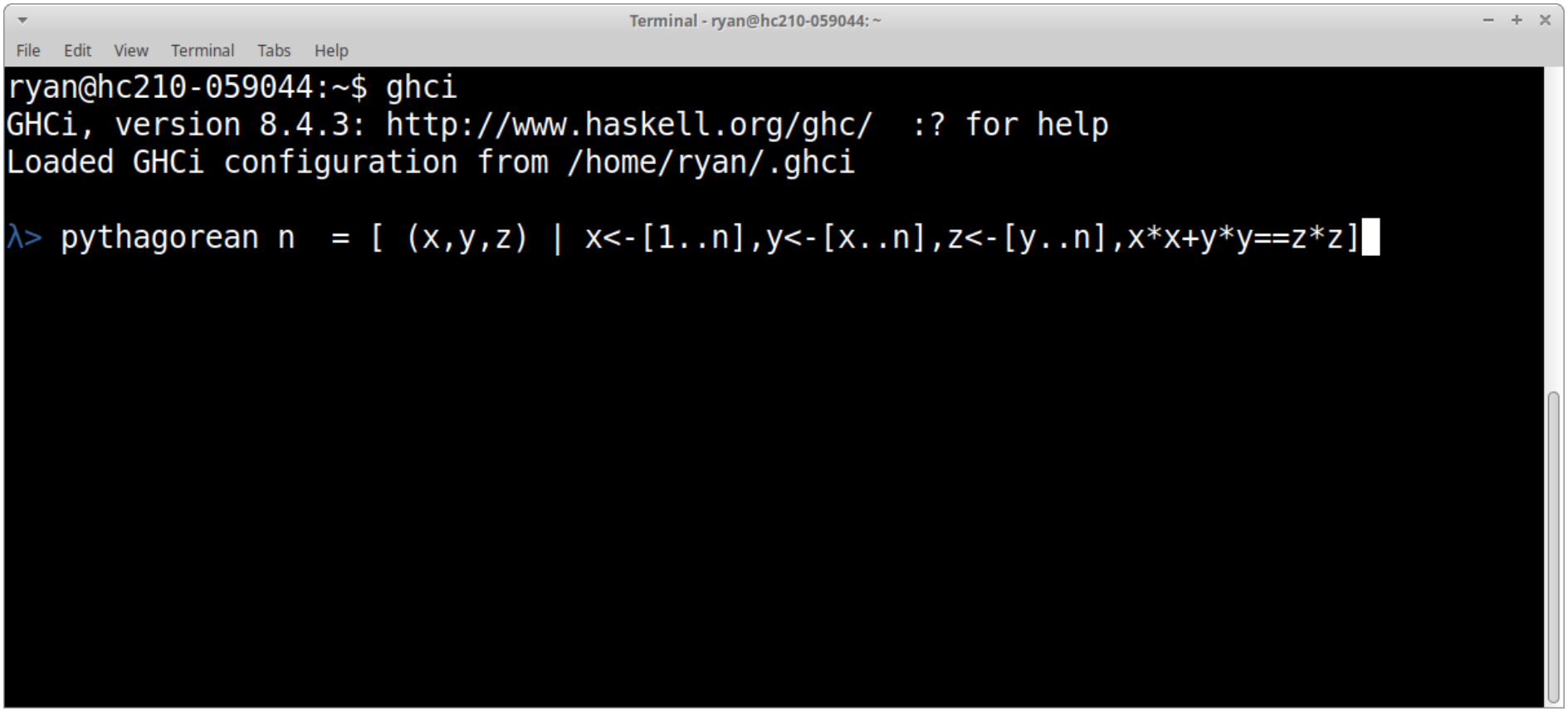
List Comprehension and Functions



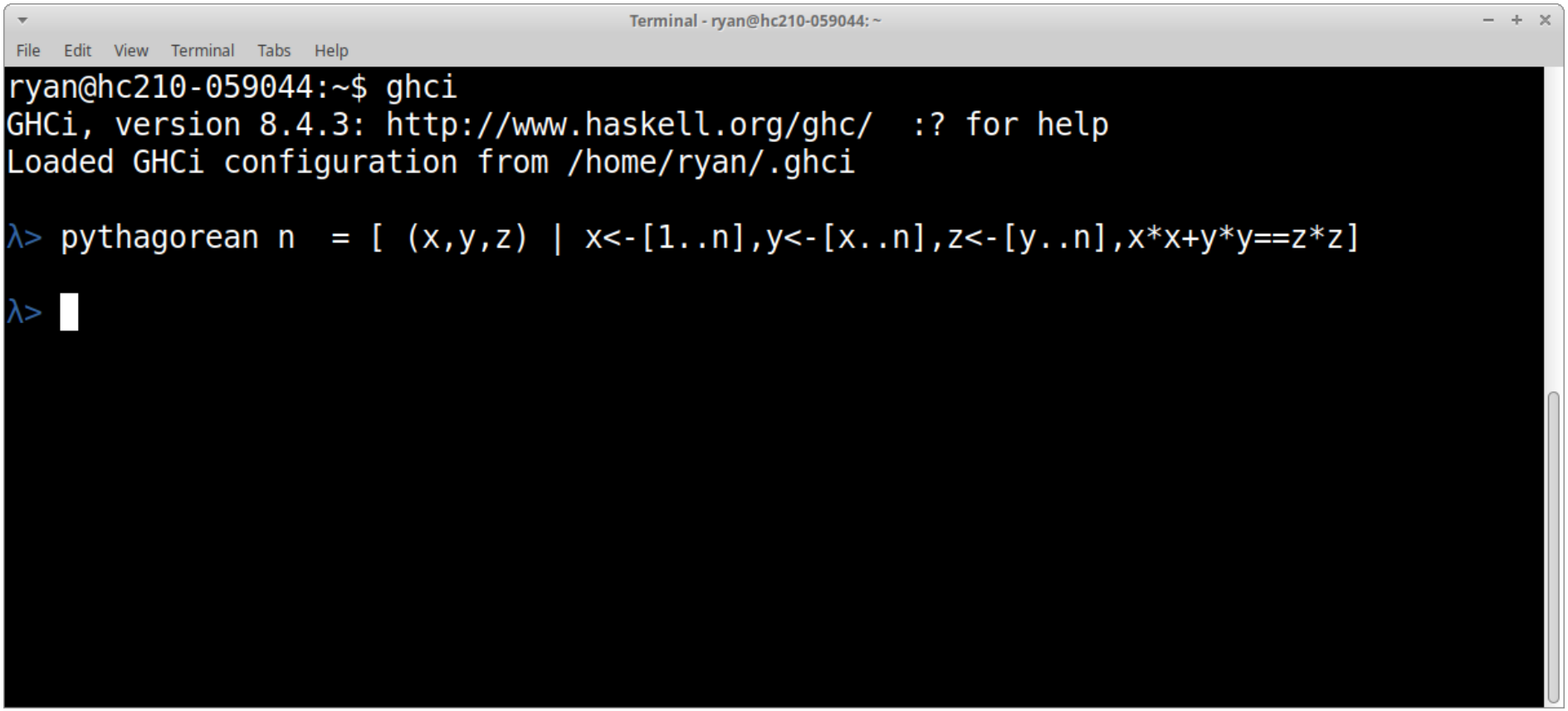
List Comprehension, Pythagorean Triples



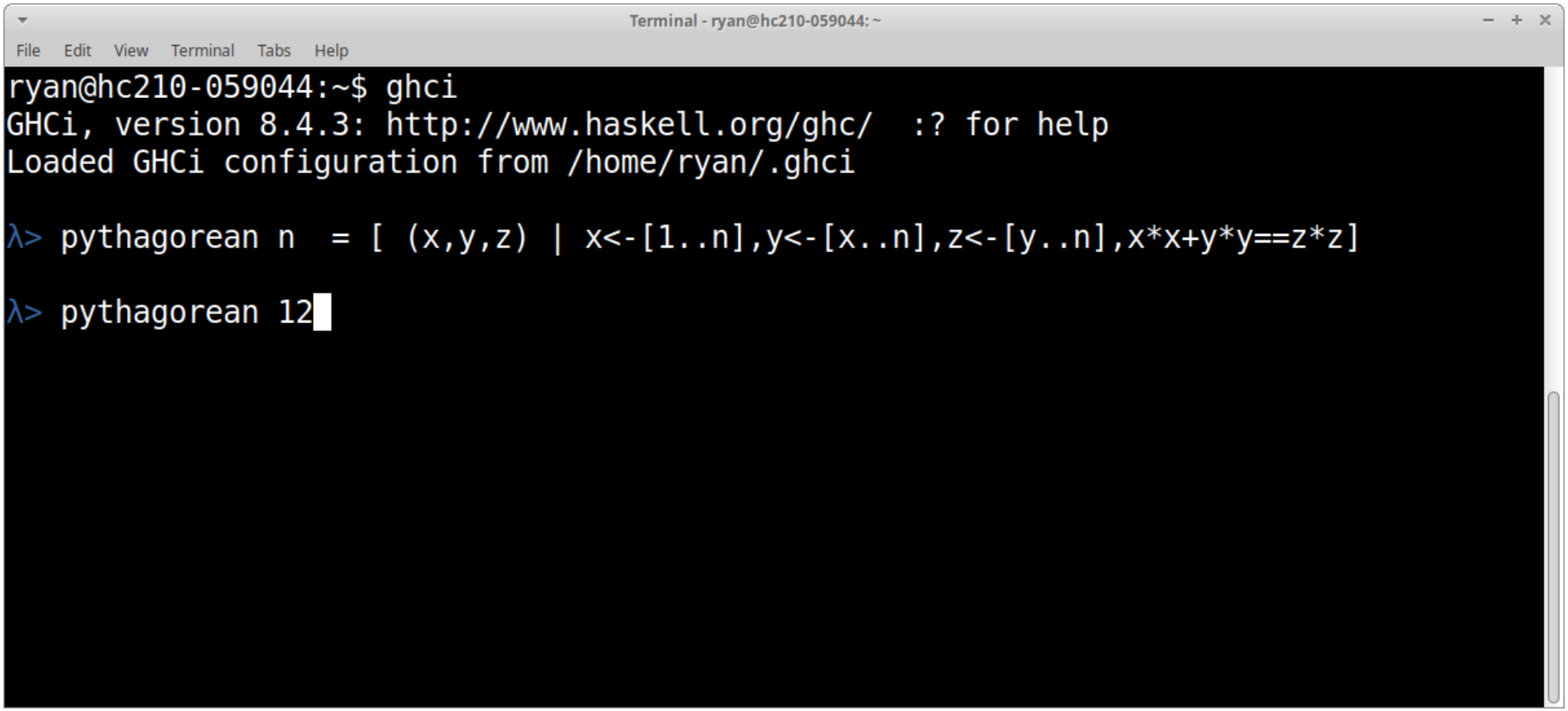
List Comprehension, Pythagorean Triples



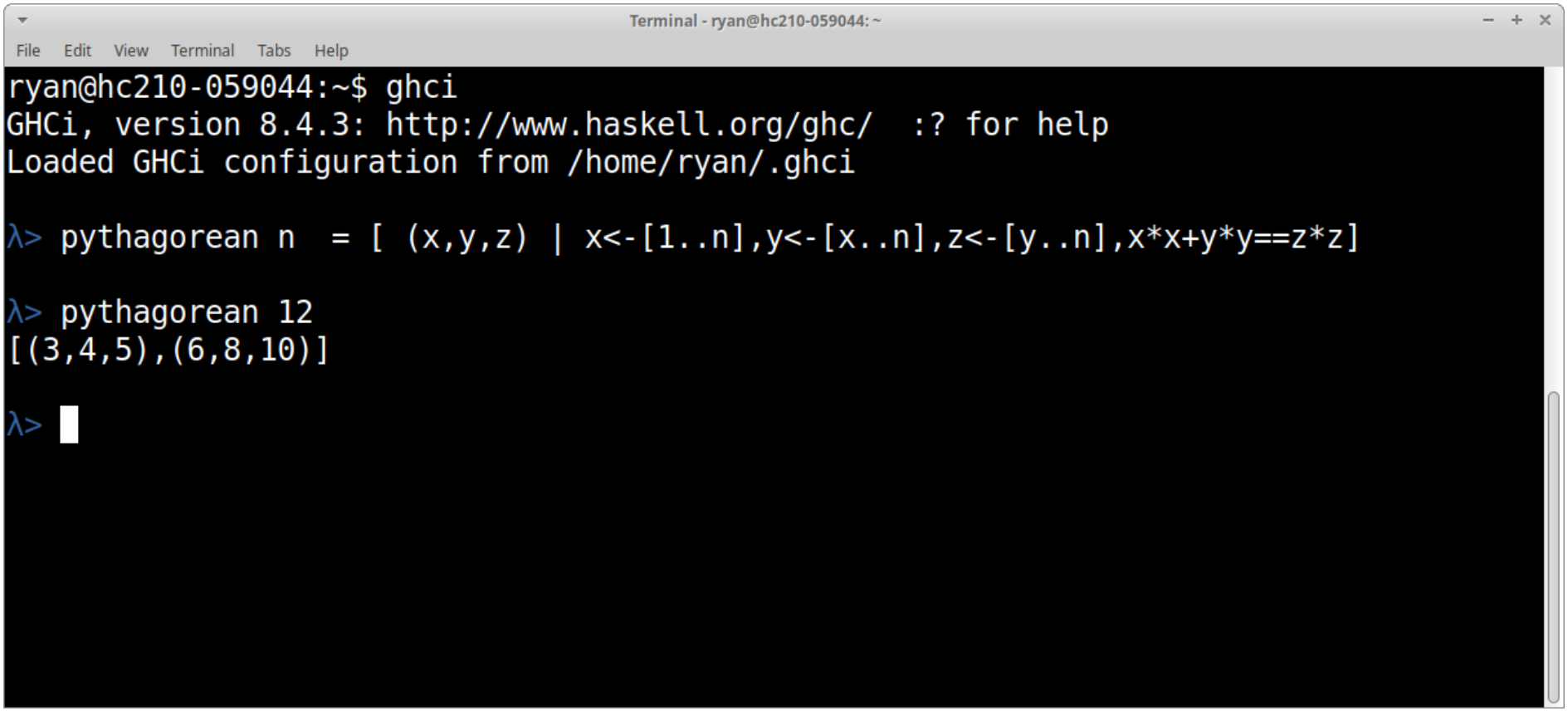
List Comprehension, Pythagorean Triples



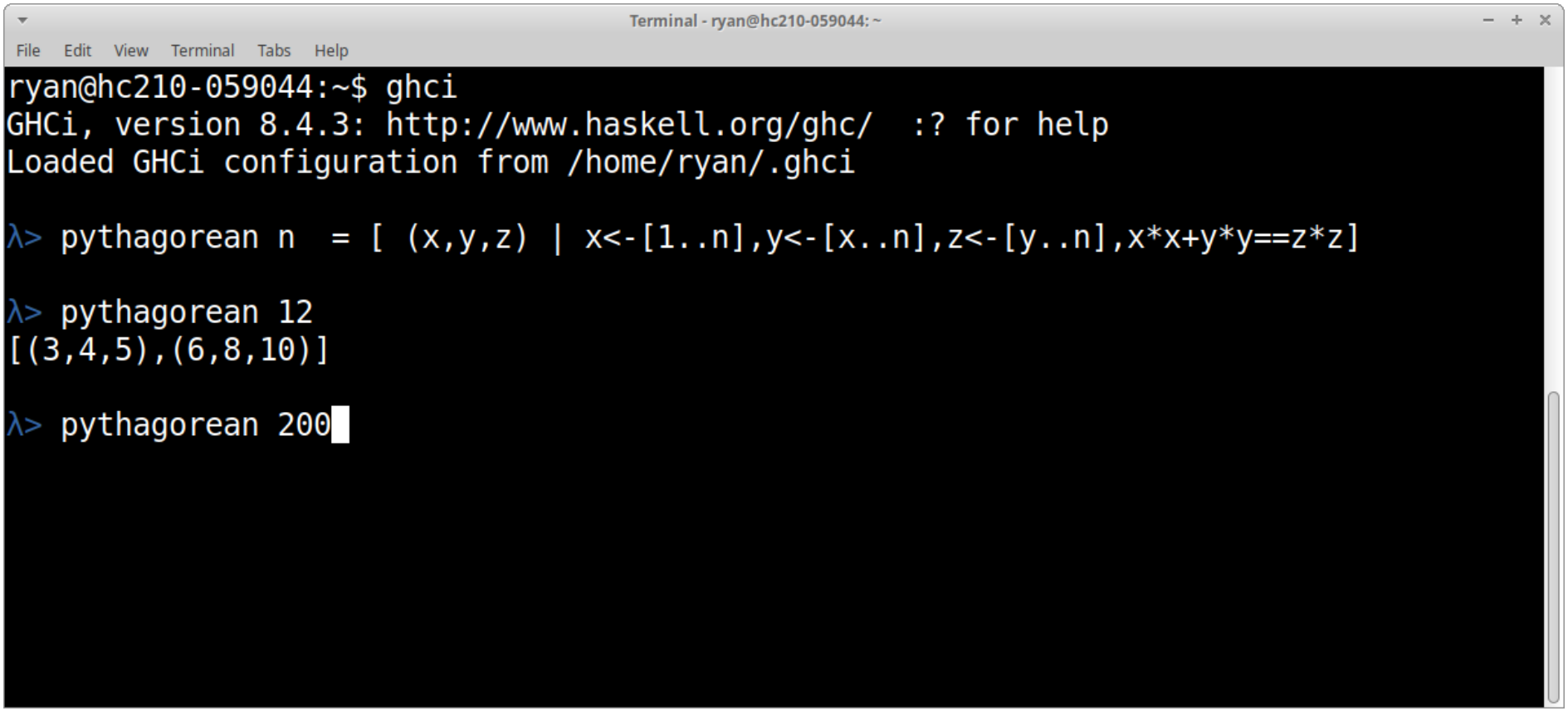
List Comprehension, Pythagorean Triples



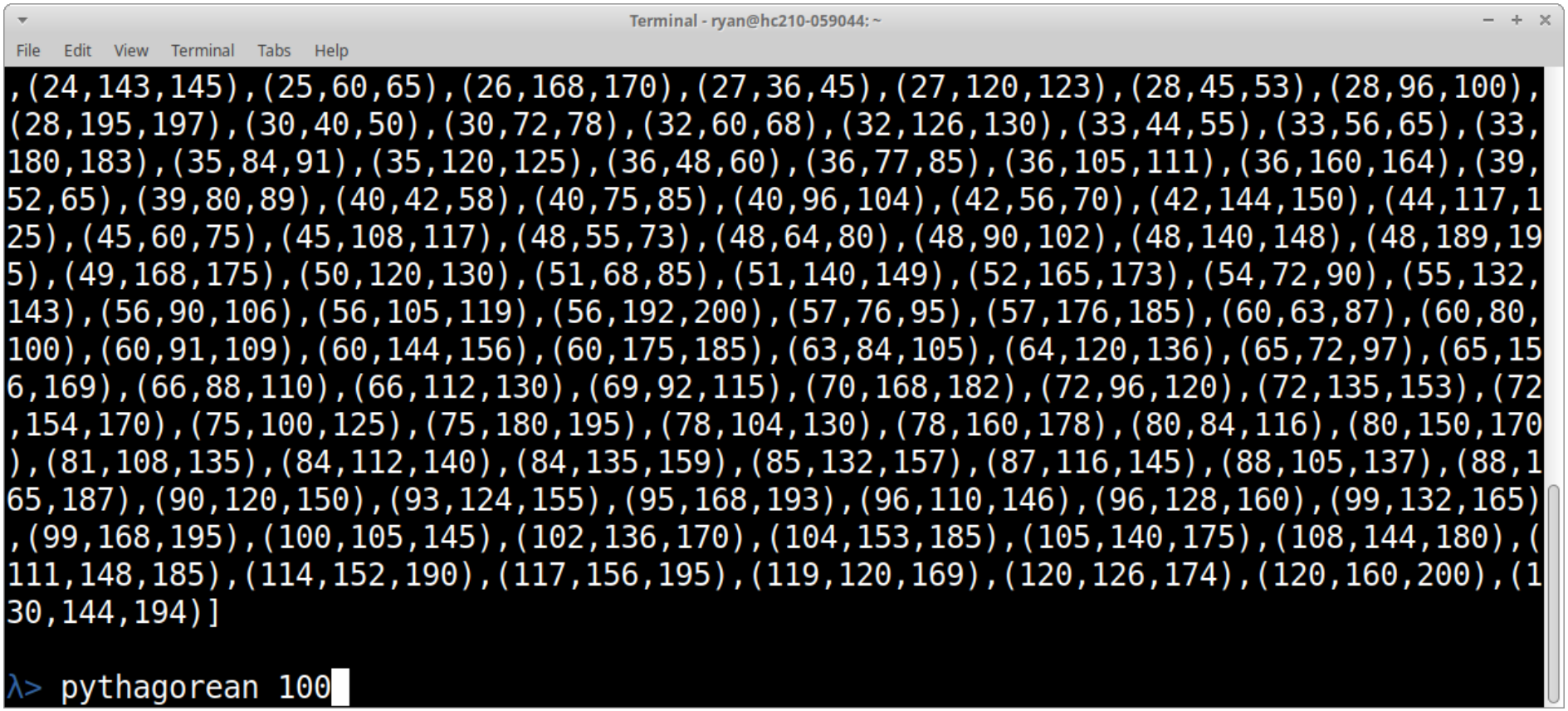
List Comprehension, Pythagorean Triples



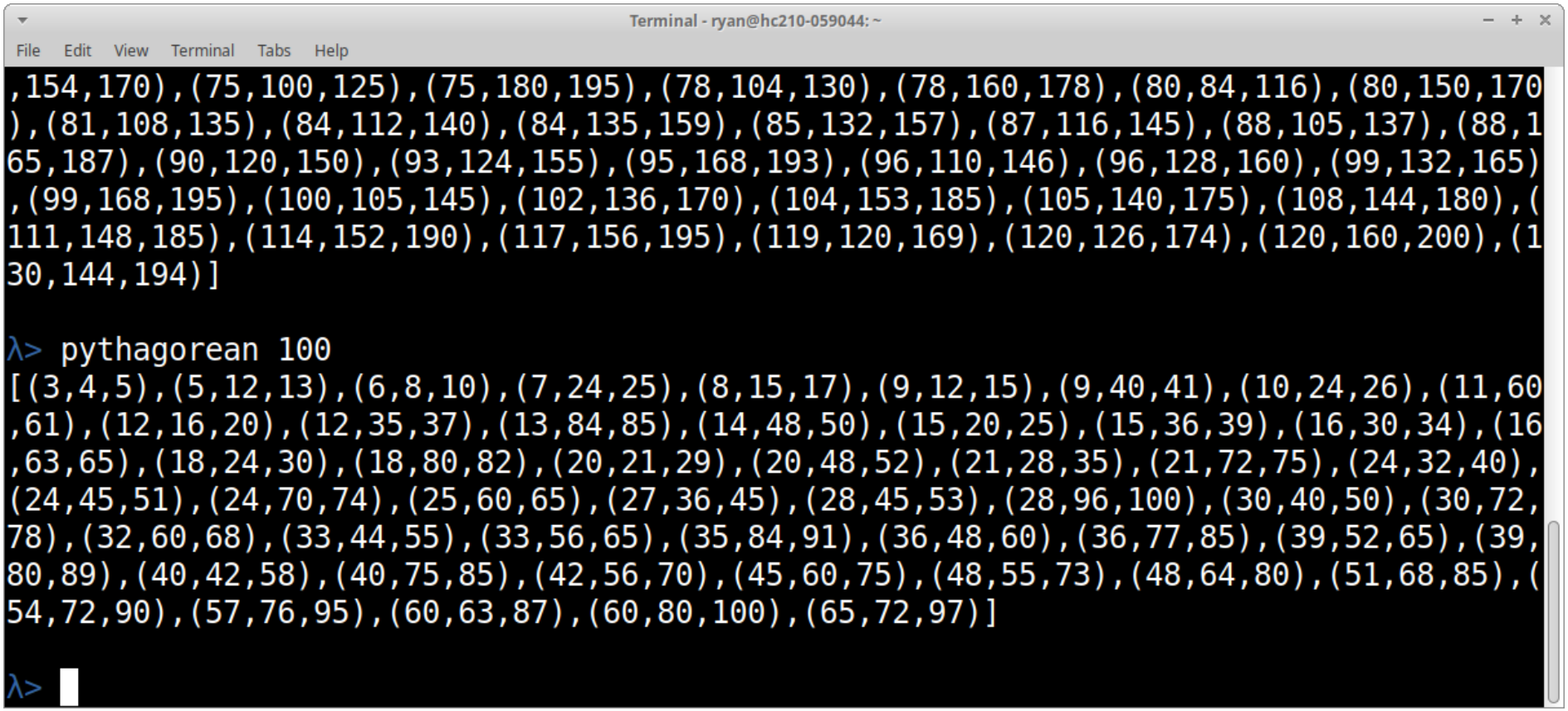
List Comprehension, Pythagorean Triples



List Comprehension, Pythagorean Triples



List Comprehension, Pythagorean Triples

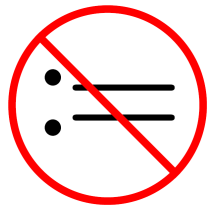


Before continuing on the last and most important data value: functions, we consider the problem of giving data values names.

It is important psychologically to give the programmer a means of refering to data and not just creating data.

- Every object of computation/value gets a name in the same way:

```
theMeaningOfLife = 42
```

NB. Not assignment! 

- Control of scope using `let` expression.

```
let <declaration> in <expression>
```

[We don't need any examples now.]

Syntax of Names

Names consist of either letters and digits, or of all symbols. All symbolic names are treated as infix operators by the parser. All non-symbolic names are treated as prefix functions by the parser.

```
:
!!
++
*

div
quotRem
Integer
```

Syntax of Names

Enclosing a name with () tells the parser to drop the infix assumption.
Enclosing a name with backquotes tells the parser to assume infix assumption.

	prefix	infix
alphabetic	elem	`elem`
symbolc	(+)	+

```
div 24 7
24 `div` 7

elem 5 [1,2,3,4,5,6]
5 `elem` [1,2,3,4,5,6]

(+) 2 3
2 + 3
```

Declarations

A name is given a value by a declaration of the simple form:

```
name = value
```

In haskell this declaration is actually a specific case of a more general declartion of the form:

```
pattern "=" value
```

Patterns (section 3.17.1 of the reference 2010 manual) are more naturally discussed later in the context of functions.

There are declarations for data types and classes. These will be discussed later.

Declarations

Many languages have declarations for many kinds of things: constants, variables, procedures, and functions.

Haskell does not talk about memory locations (however, see boxed versus unboxed), hence no need for variable declarations.

Fundamentally, given a datum a name in Haskell is the same for any value—functions included.

Overview of Simple Functions

1. syntax (lambda)
2. Two Haskell quirks
3. anonymous (functions as regular data)
4. special syntax
5. patterns (generalization of formal parameters)
6. definition by cases
7. guards [omit]

Syntax of Functions

```
\ x -> 2 * x
```

Syntax of Functions

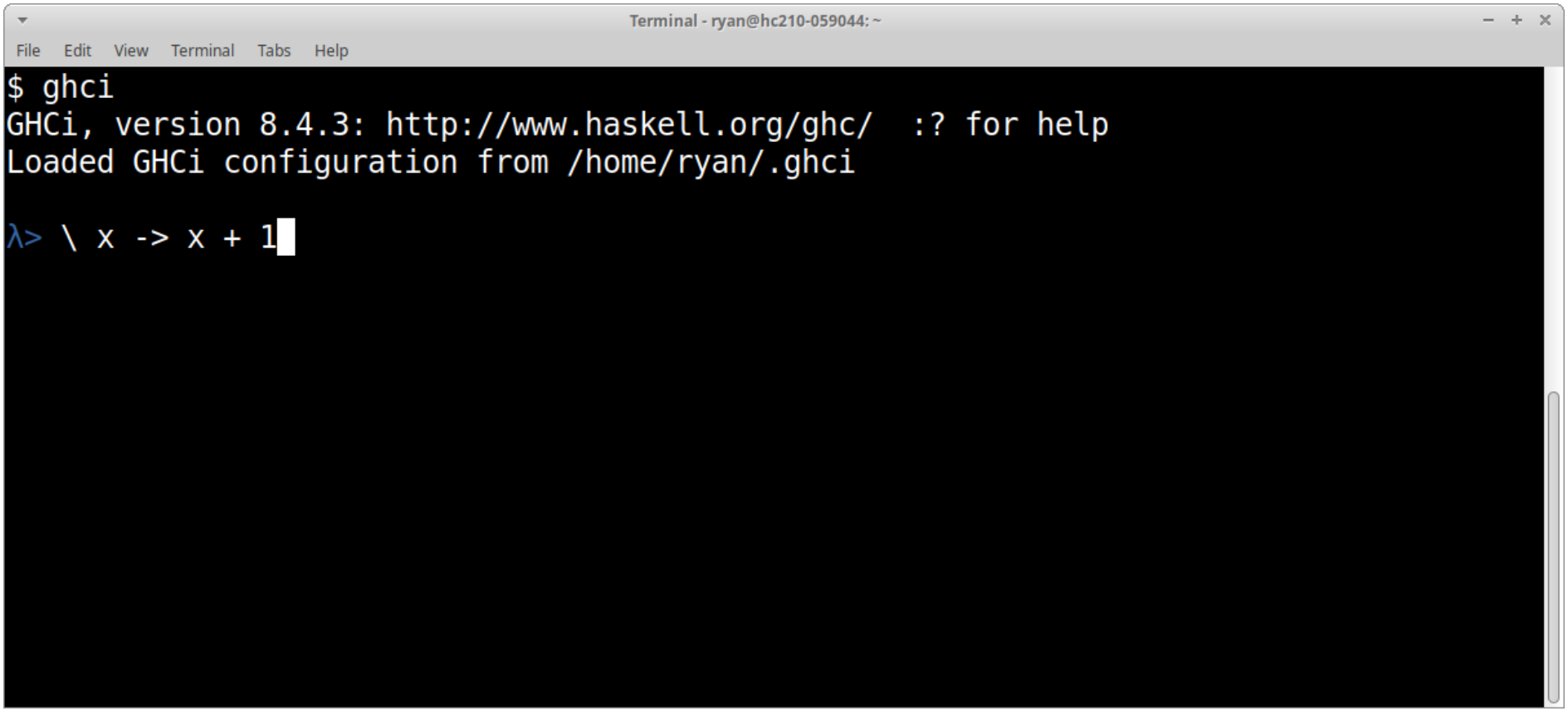
- backslash; pronounced “lambda”
- identifier: name of formal parameter (but generalized to patterns later!)
- arrow separates teh body of the fucntion from the parameter

Using GHCi with Functions

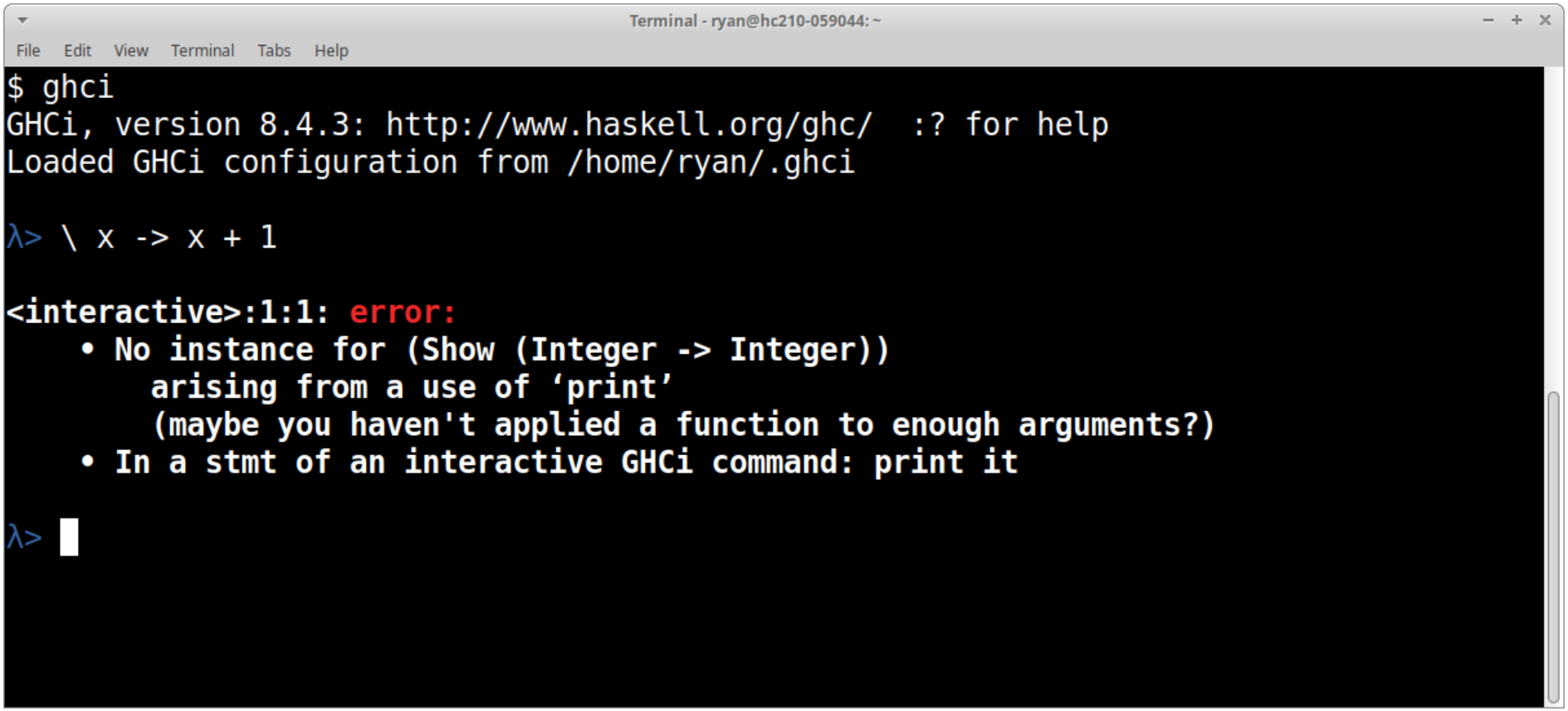
Before we can begin illustrating functions we must deal with two quirks in Haskell which totally distract and even obscure the main issues.

1. Haskell does not print anything reasonable to represent a function by default.
2. The types of many function contain class constraints, and there is no need to discuss classes in a simple introduction to Haskell. *(Use* `:type +d` *directive.)*

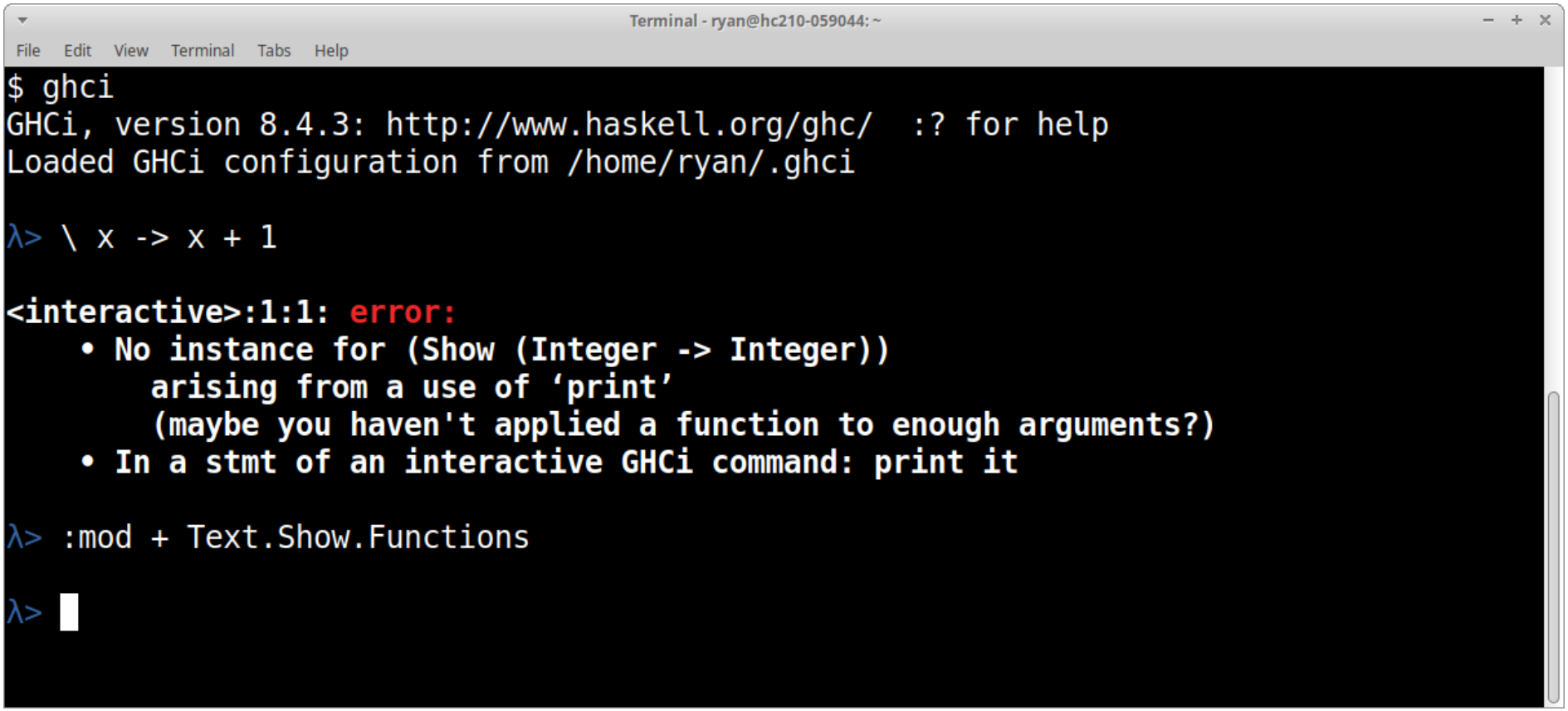
Using GHCi with Functions



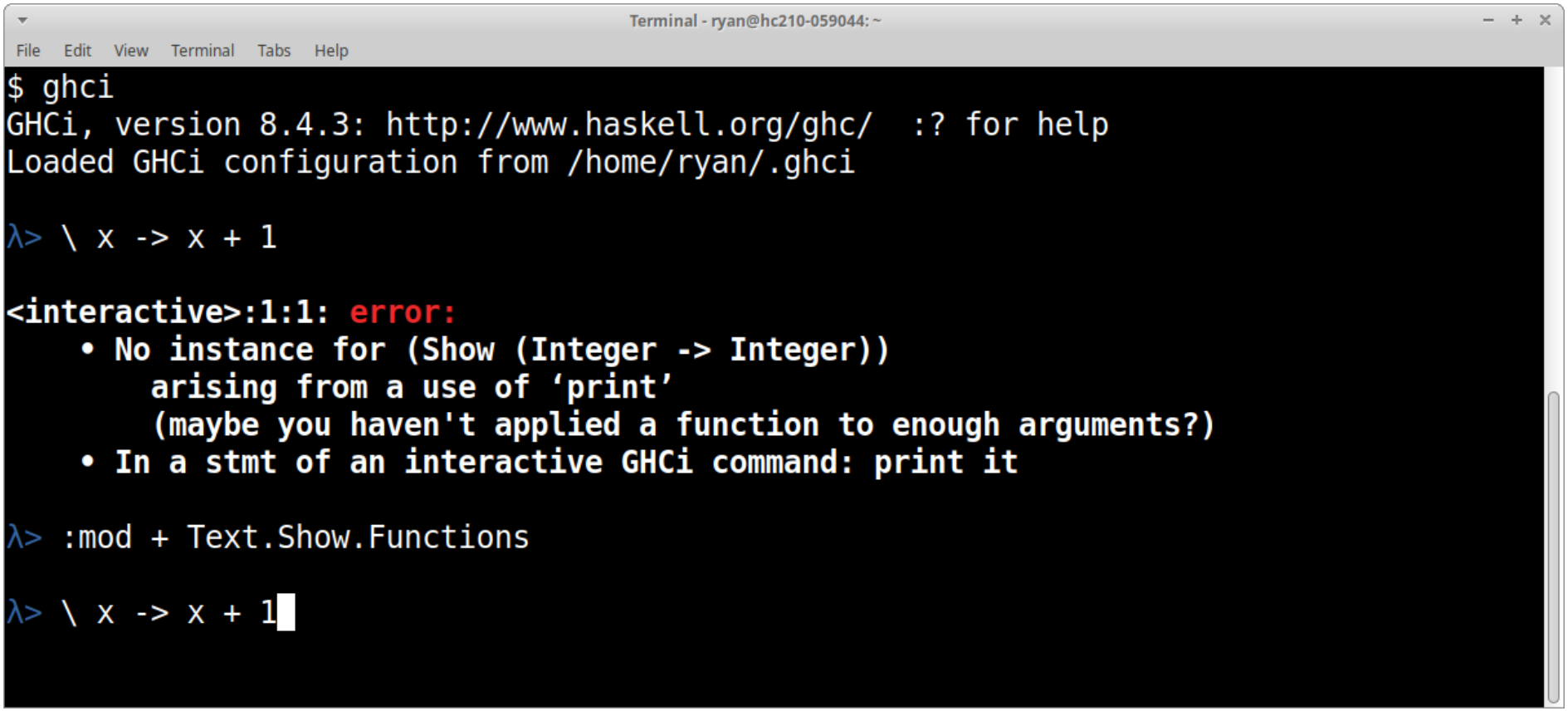
Using GHCi with Functions



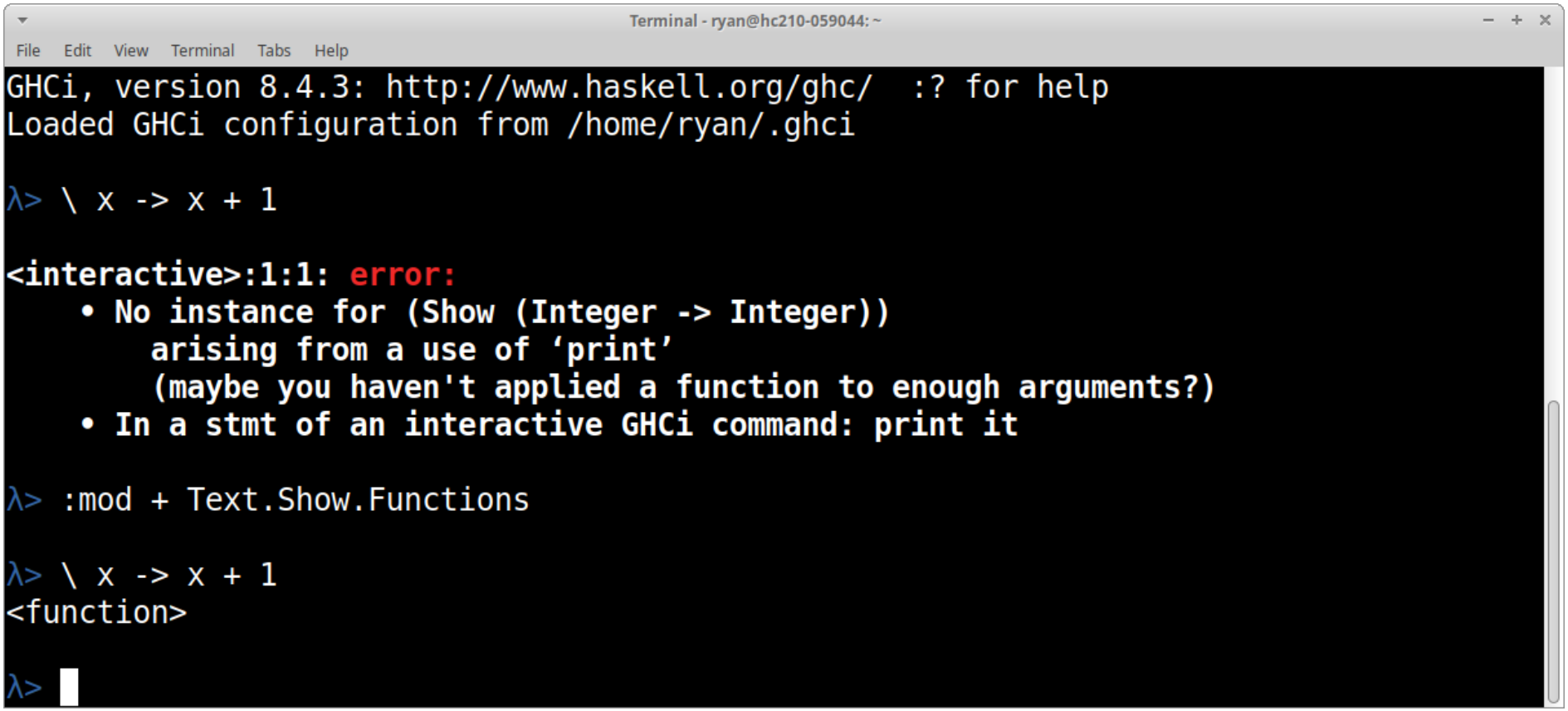
Using GHCi with Functions



Using GHCi with Functions



Using GHCi with Functions



Using GHCi with Functions

Haskell magical `show` function will show/print anything, *but not functions*.

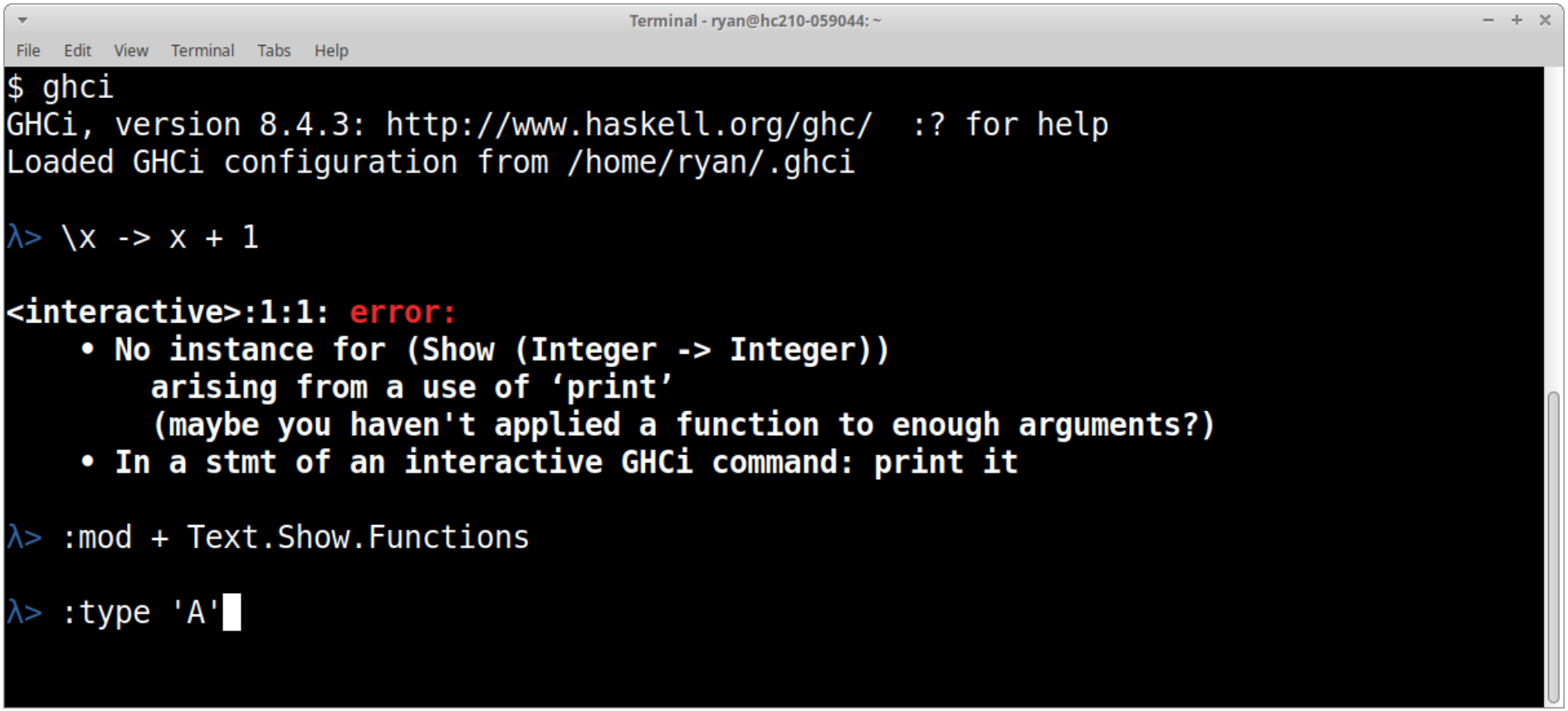
In Haskell's defense, what is the printable representation of a function? Is the Intel assembly code? The abstract syntax tree of the code? Should it just parrot back out the input source code?

One can get the Haskell interactive system to print the word `<function>` which seems like a really better idea than one of its inscrutable error messages.

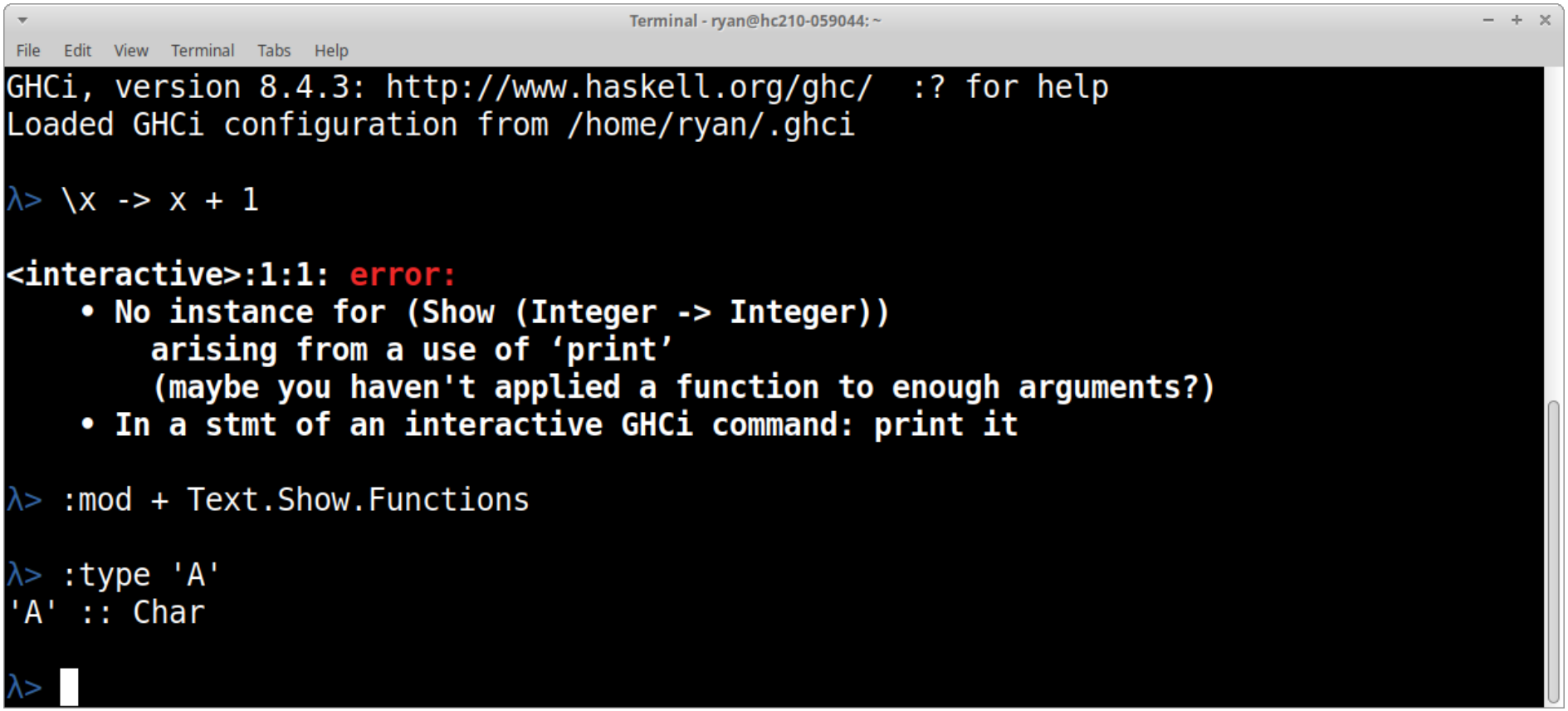
Upcoming script

```
:type 'A'
:type 3
:type \ b -> not b
:type \ x -> not x
:set +t
\ x -> x + 1
```

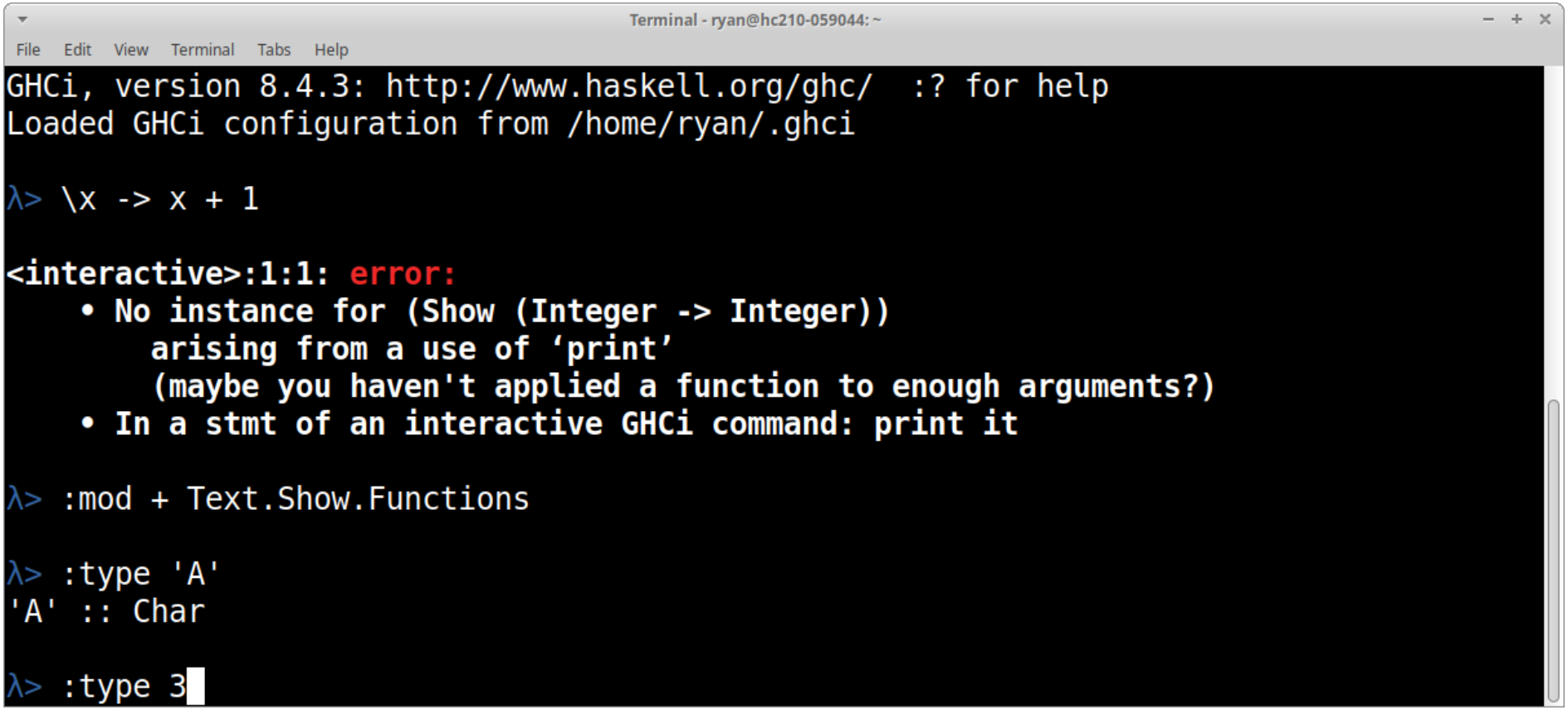
Using GHCi with Functions



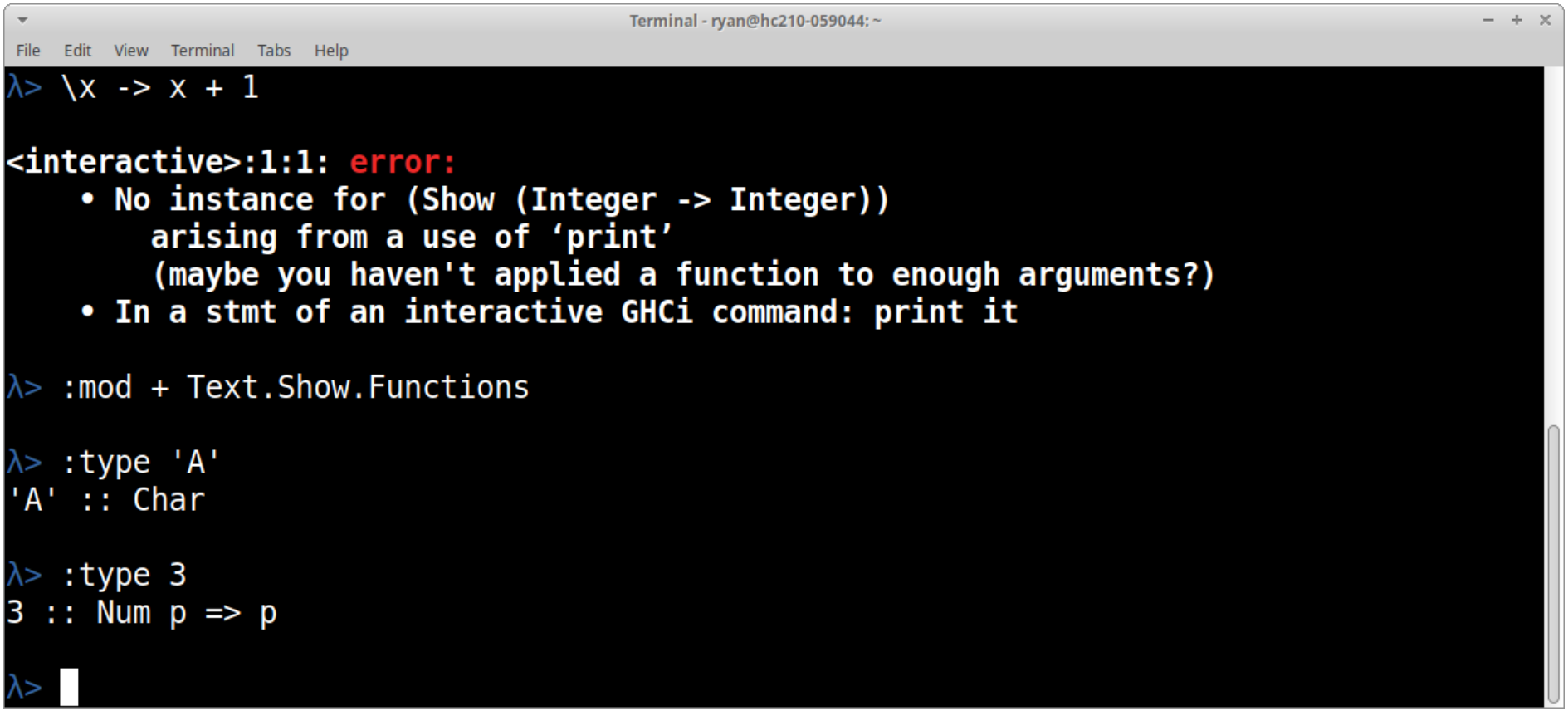
Using GHCi with Functions



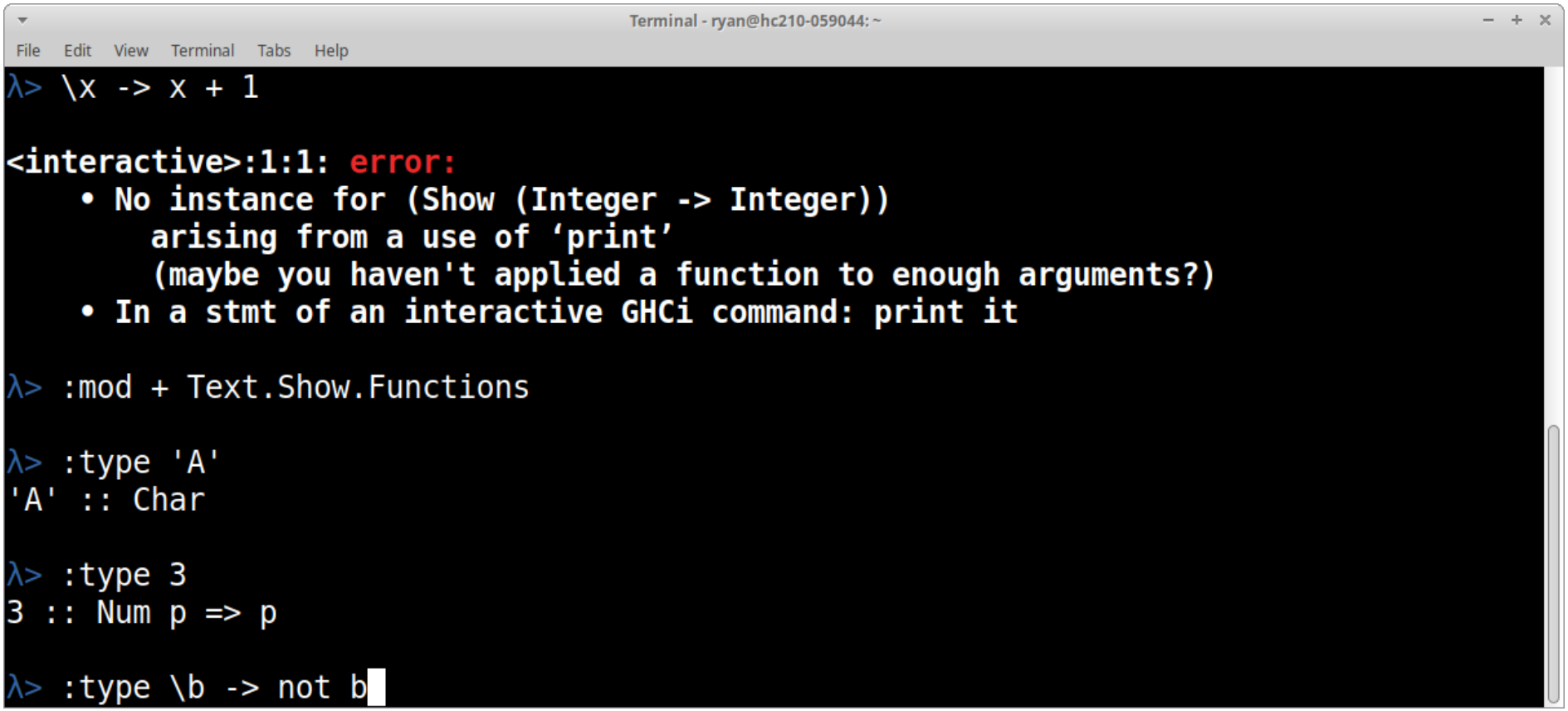
Using GHCi with Functions



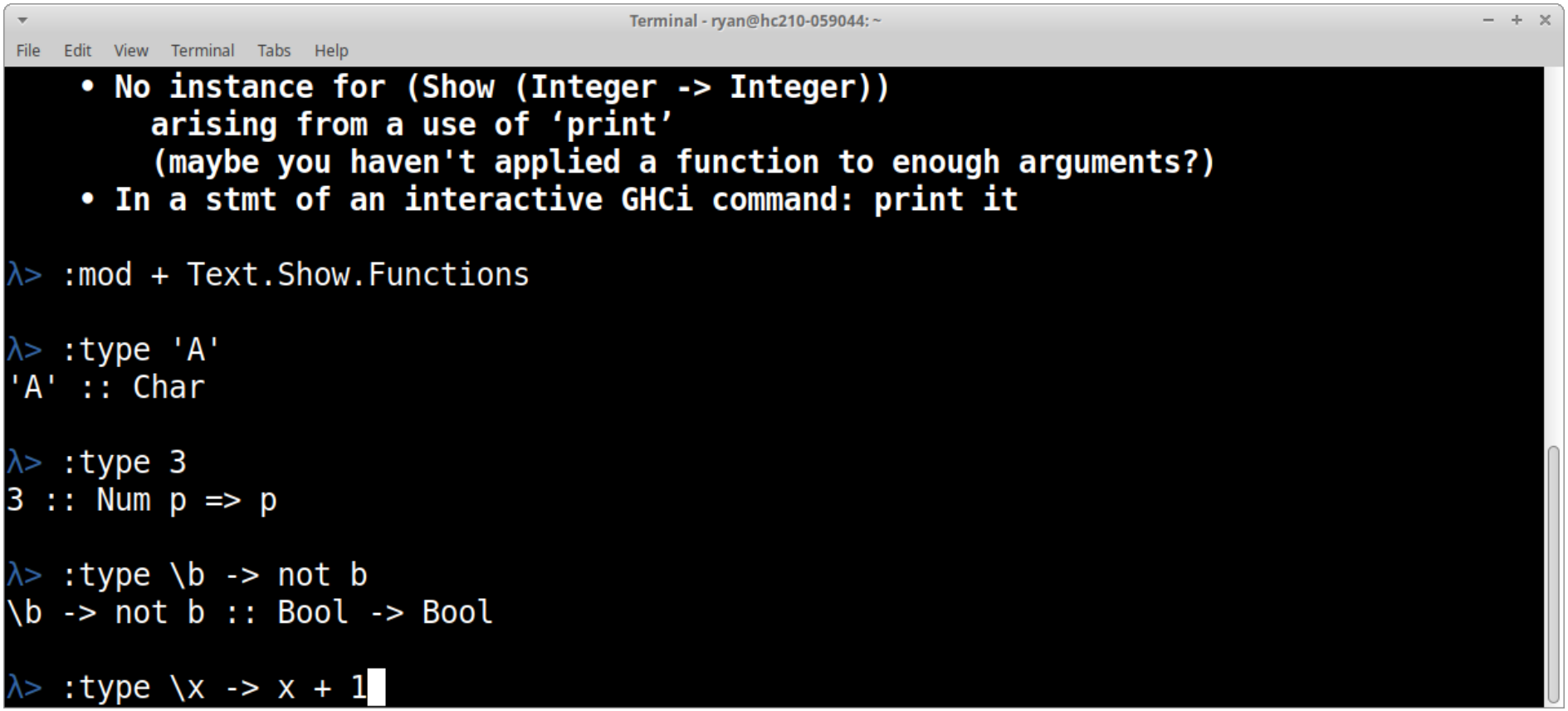
Using GHCi with Functions



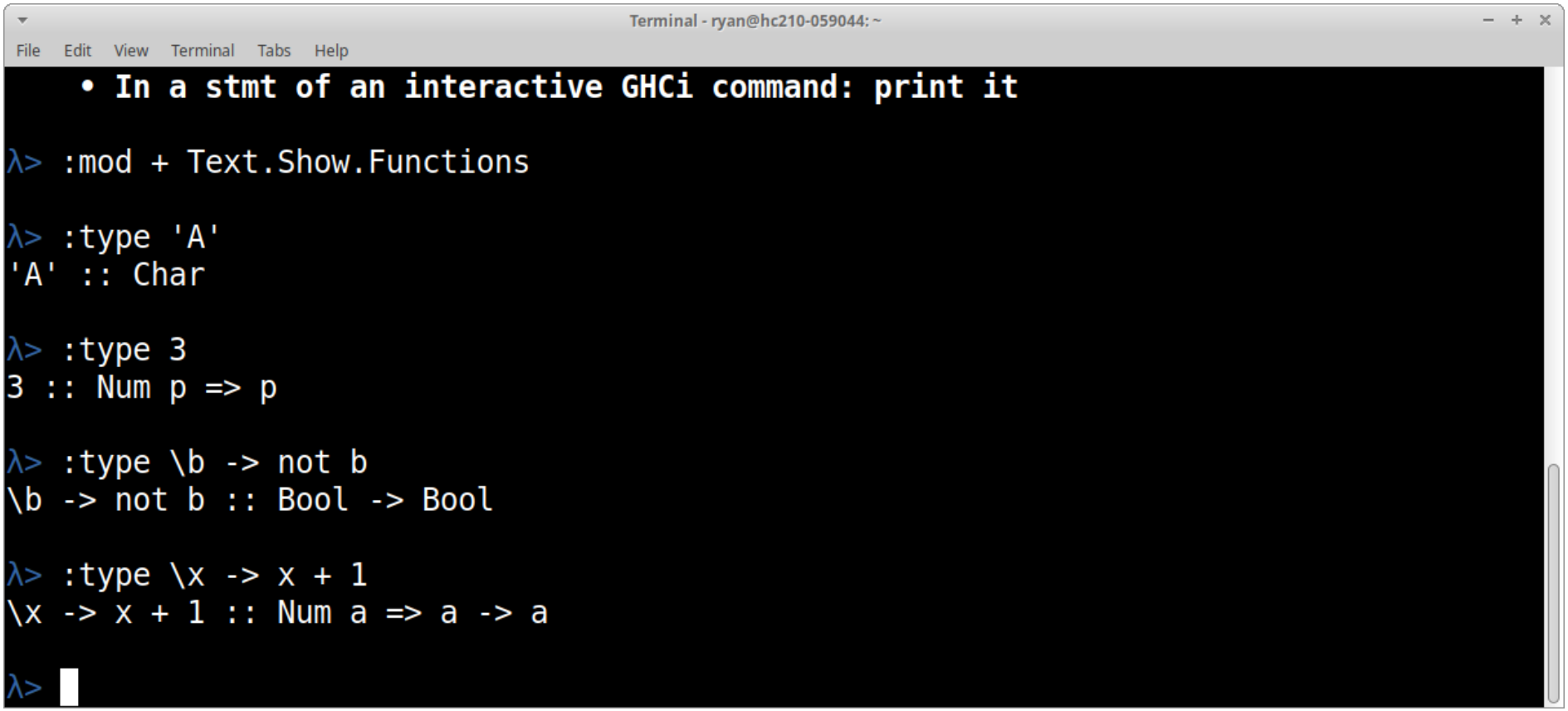
Using GHCi with Functions



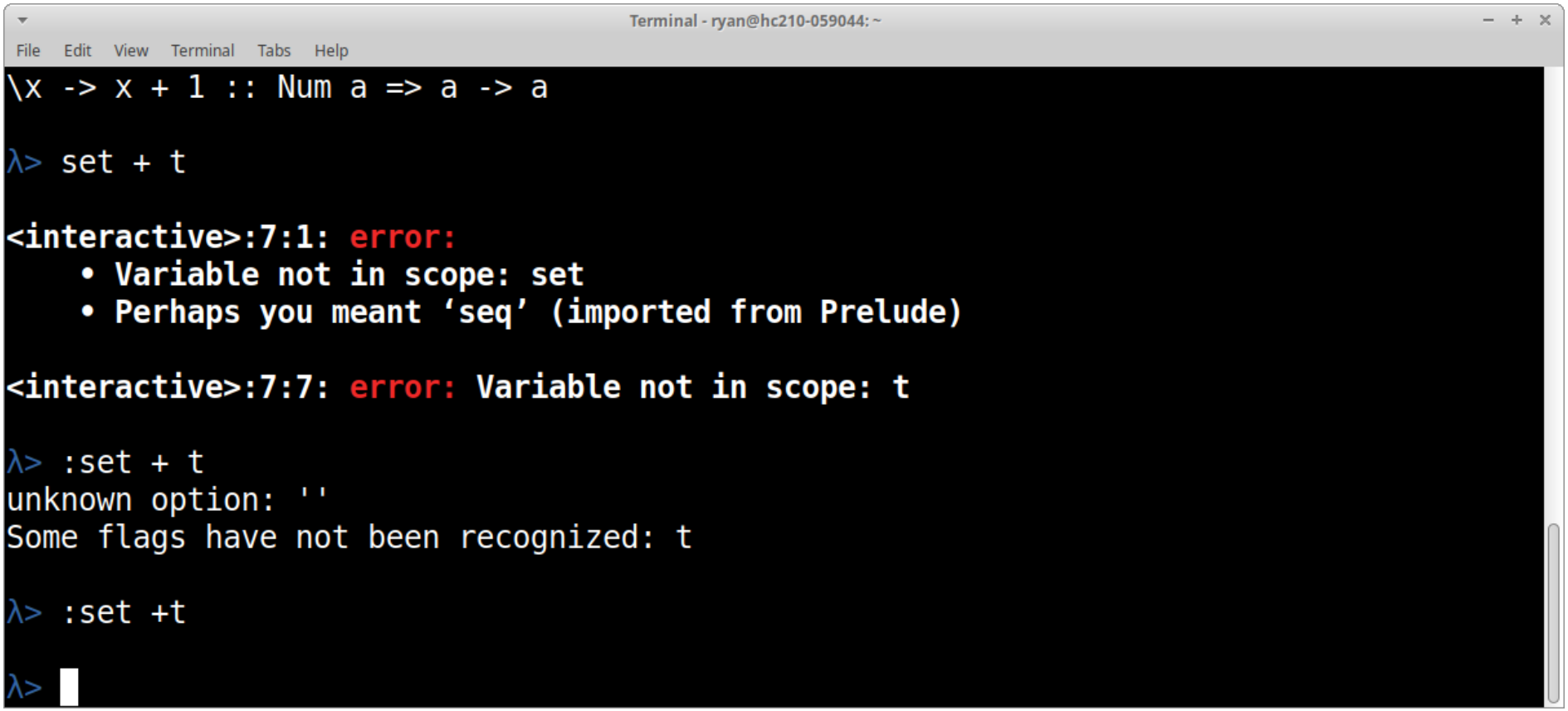
Using GHCi with Functions



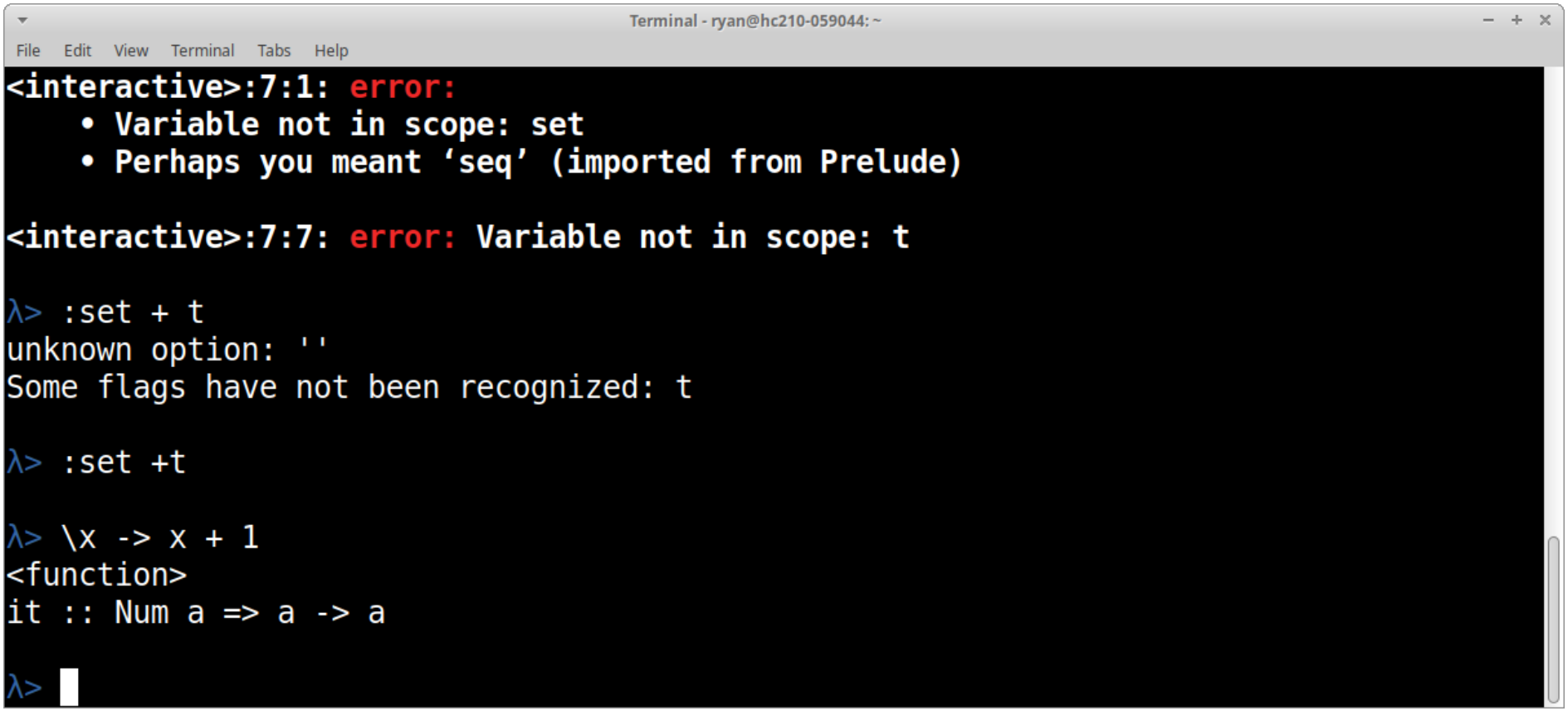
Using GHCi with Functions



Using GHCi with Functions



Using GHCi with Functions

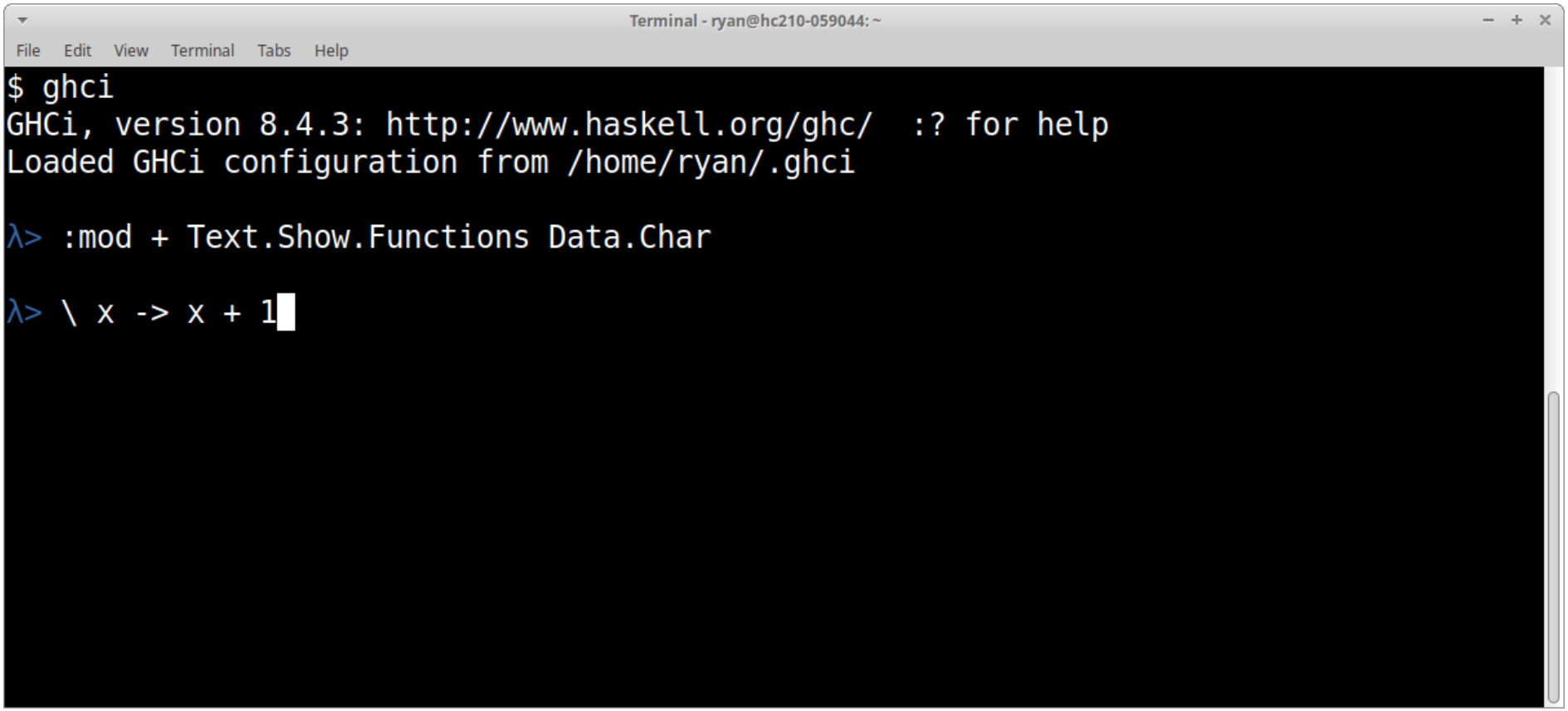


Functions Are Data

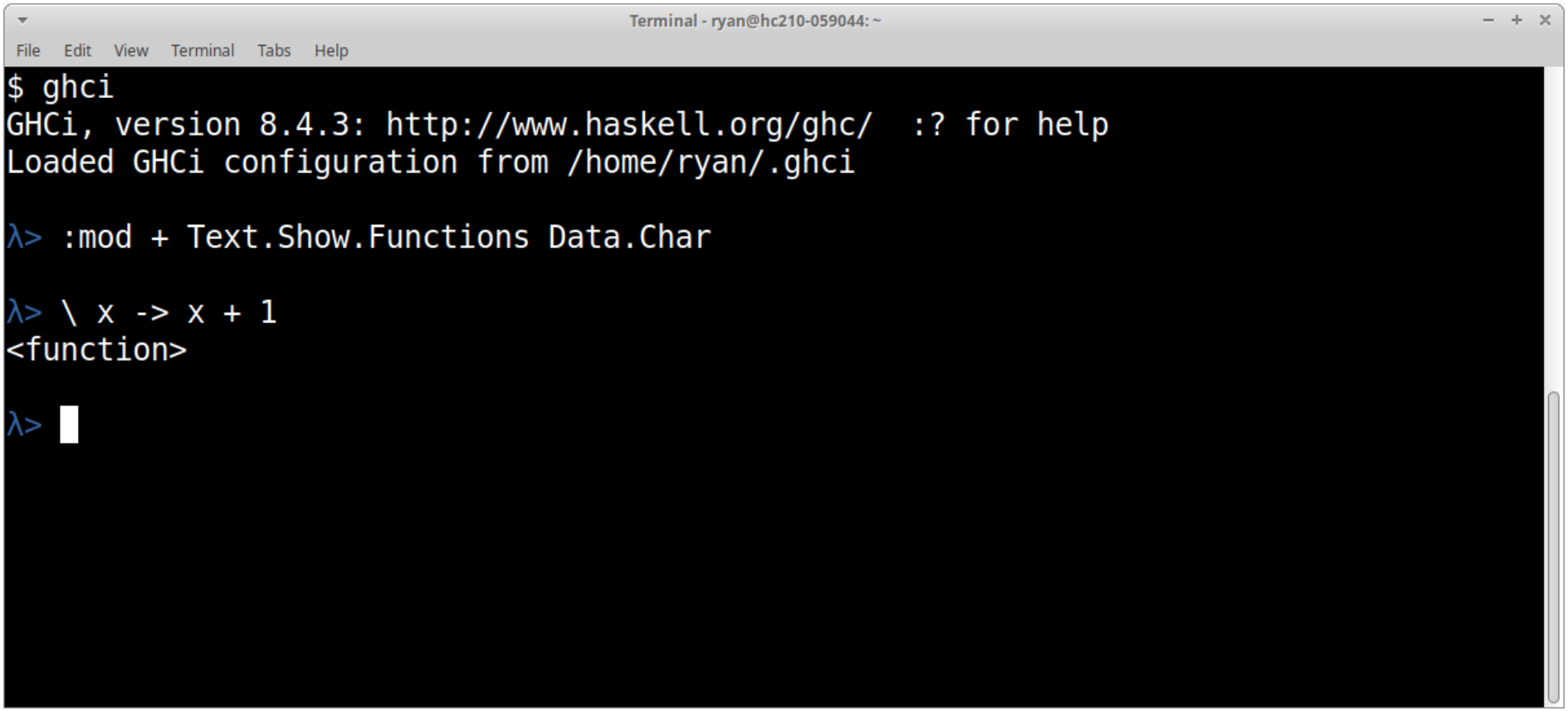
Upcoming script

```
:mod + Text.Show.Functions Data.Char

\ x -> x + 1

\ i -> max i 0

\ x -> sin (x+pi)

\ c -> prd c

\ s -> s ++ ", " ++ s

\ xs -> 2:xs
```

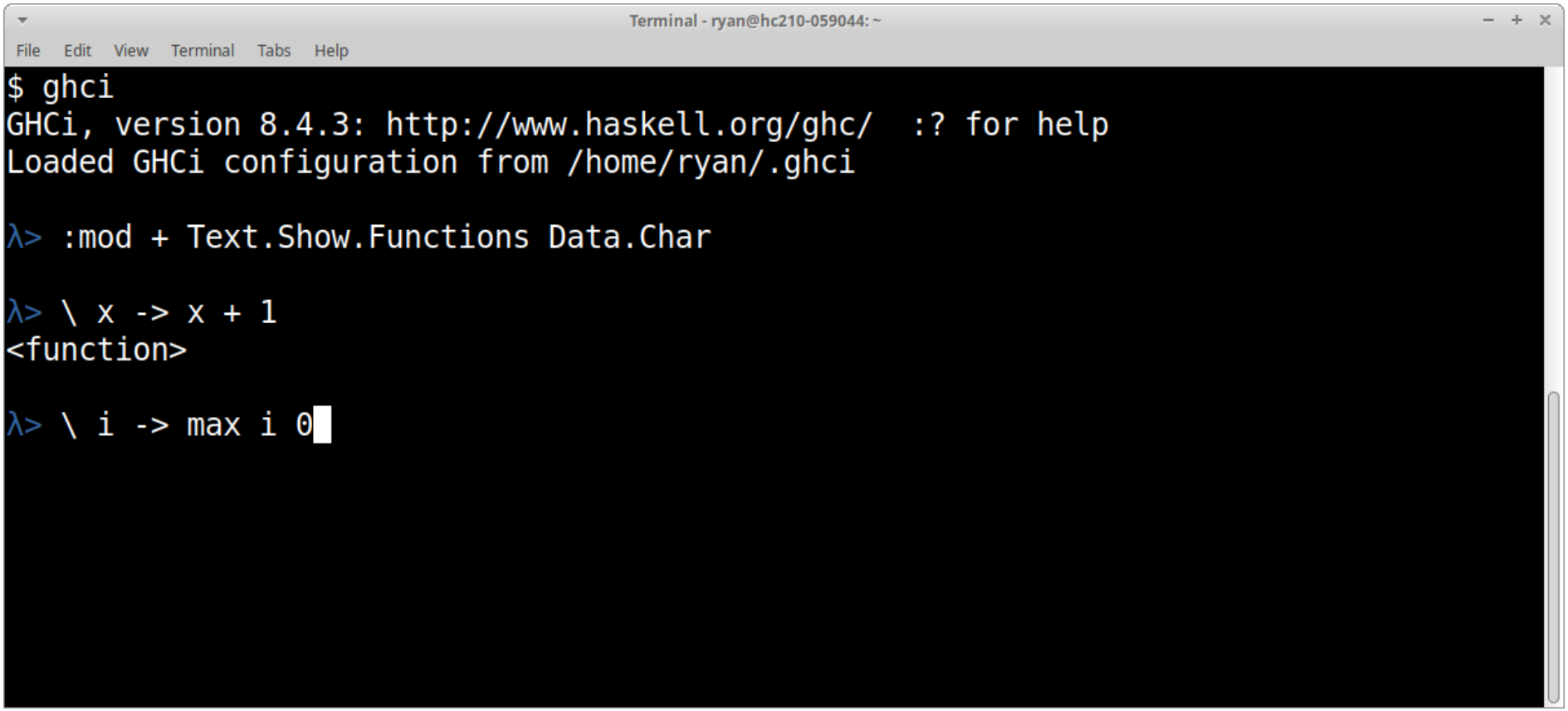
Functions Definitions



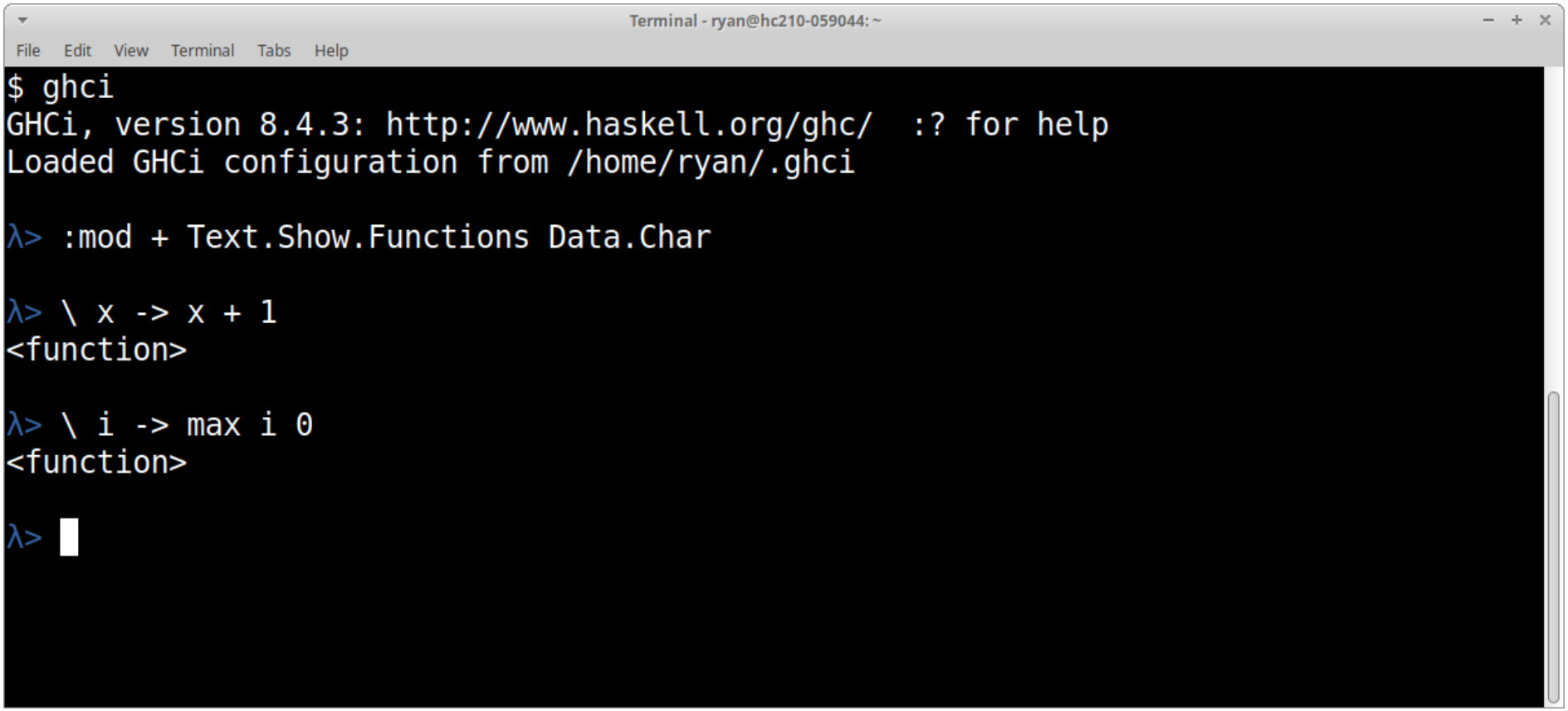
Functions Definitions



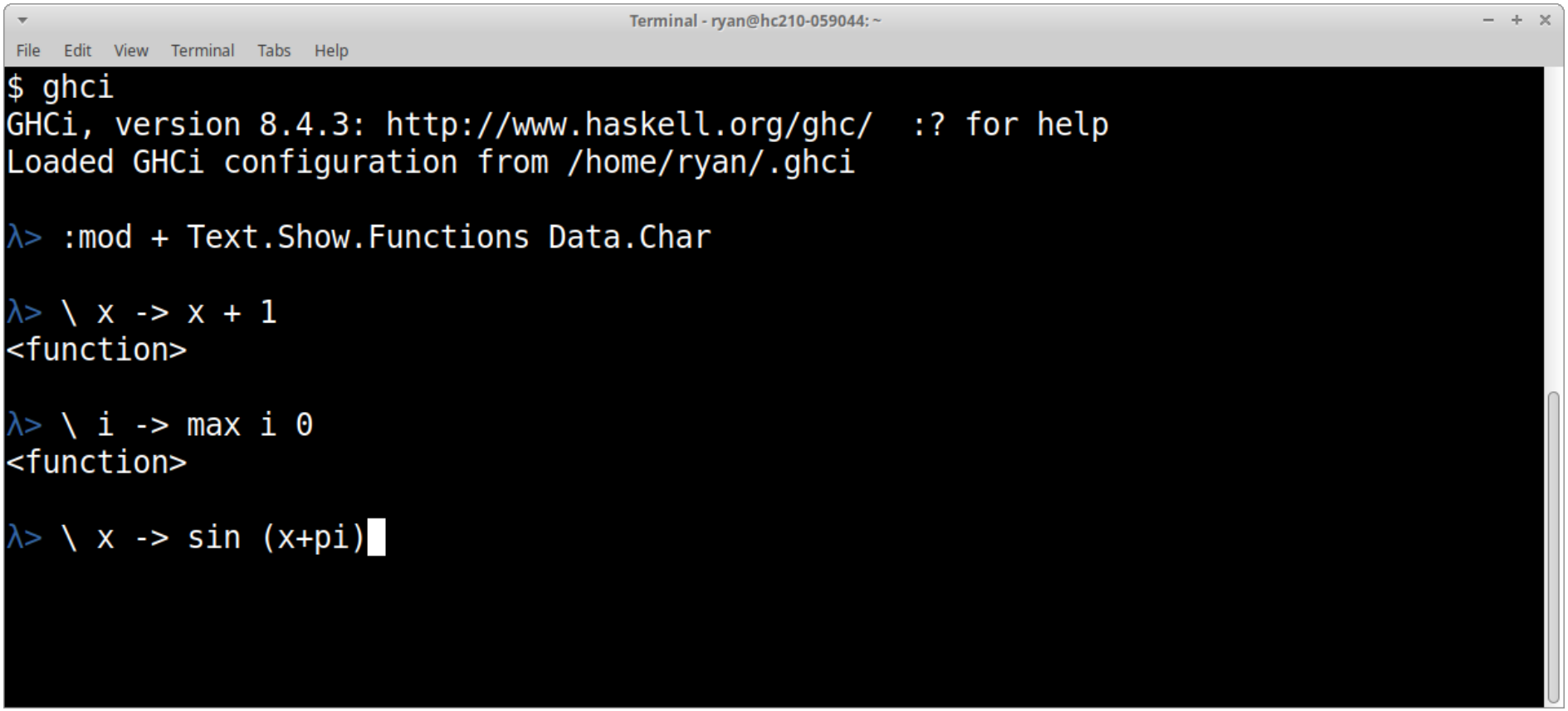
Functions Definitions



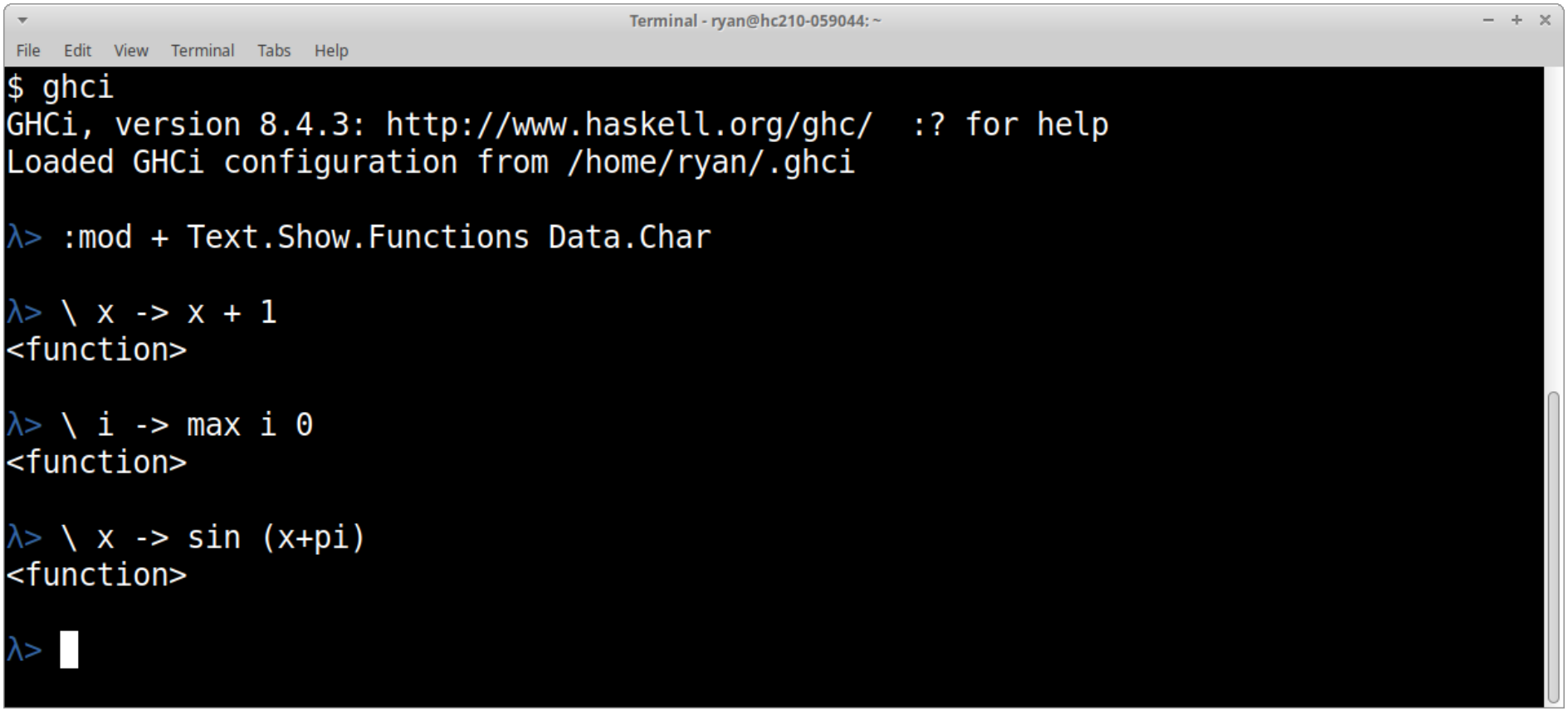
Functions Definitions



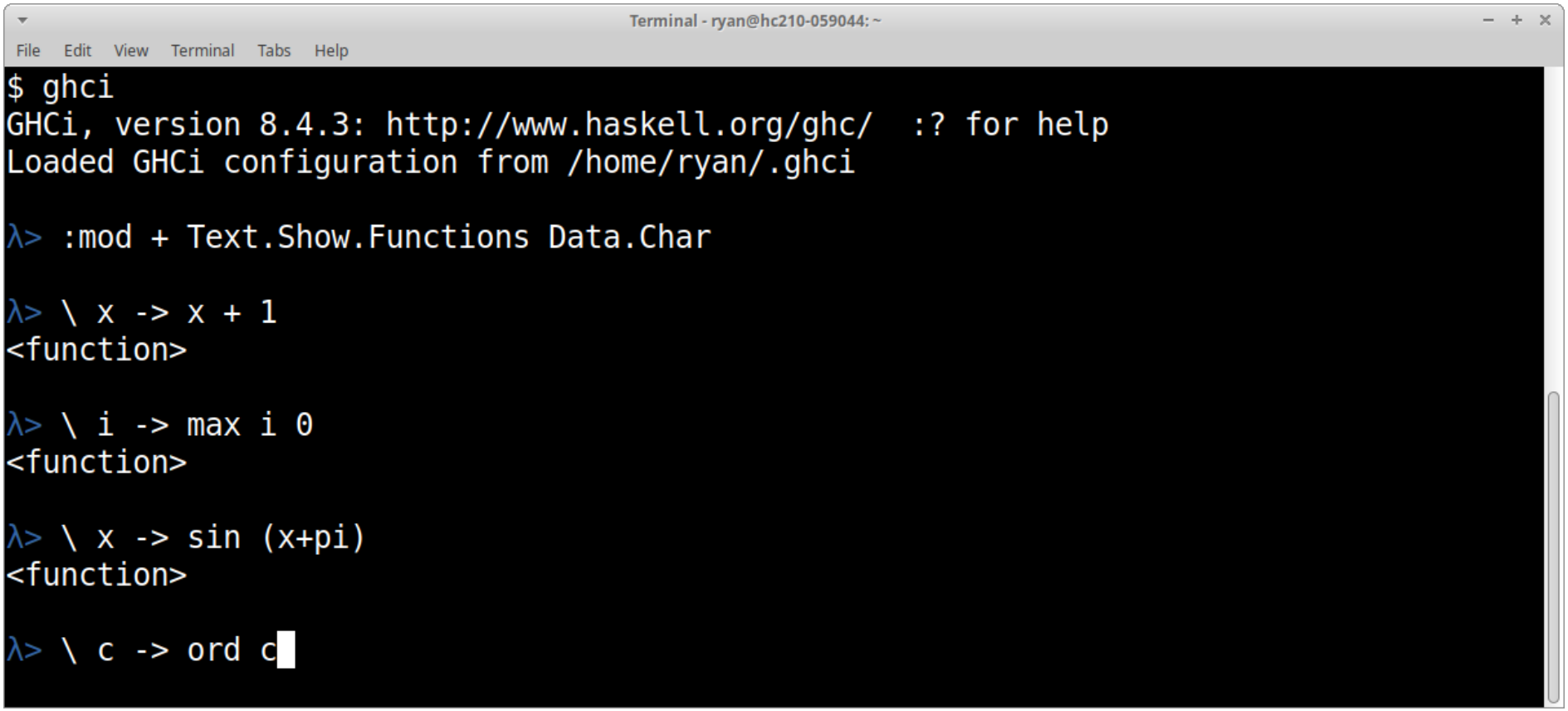
Functions Definitions



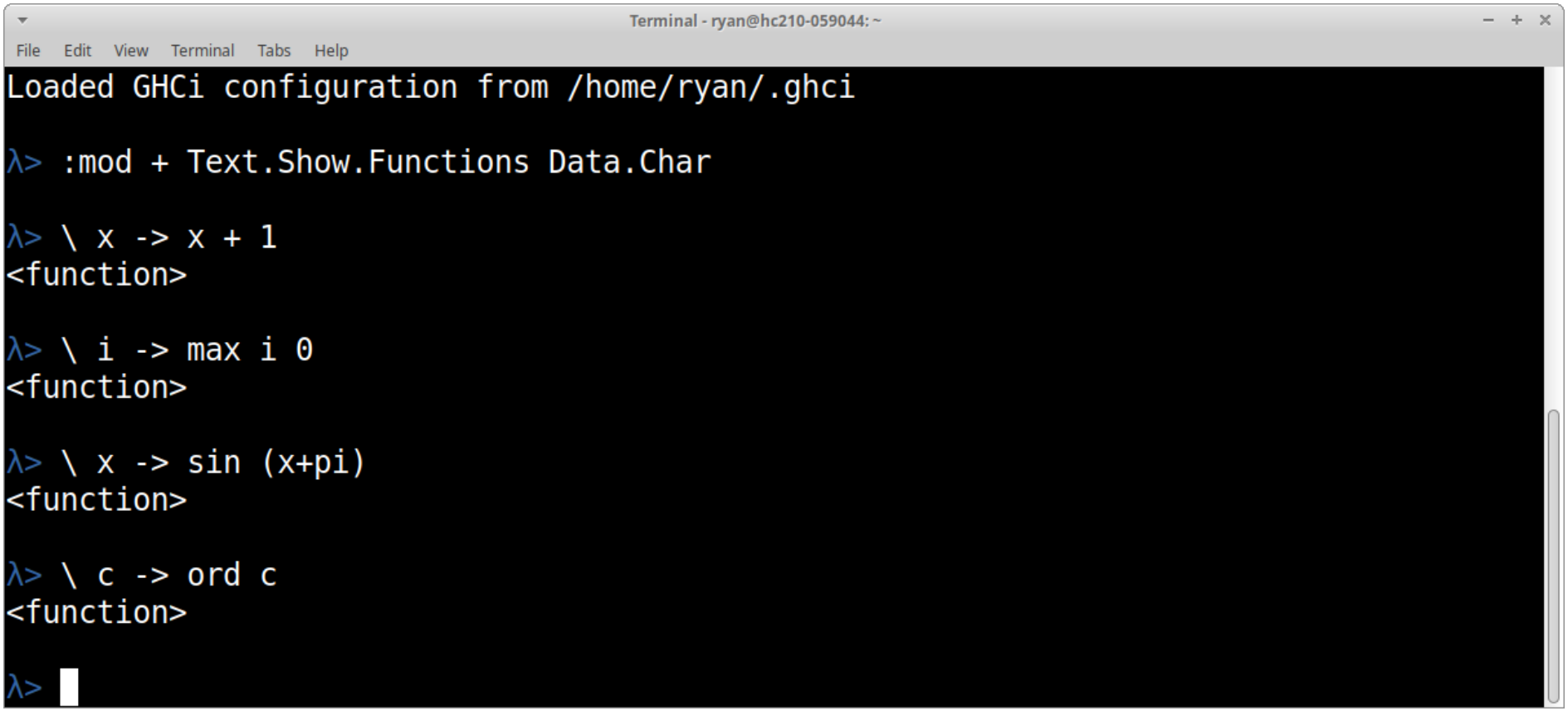
Functions Definitions



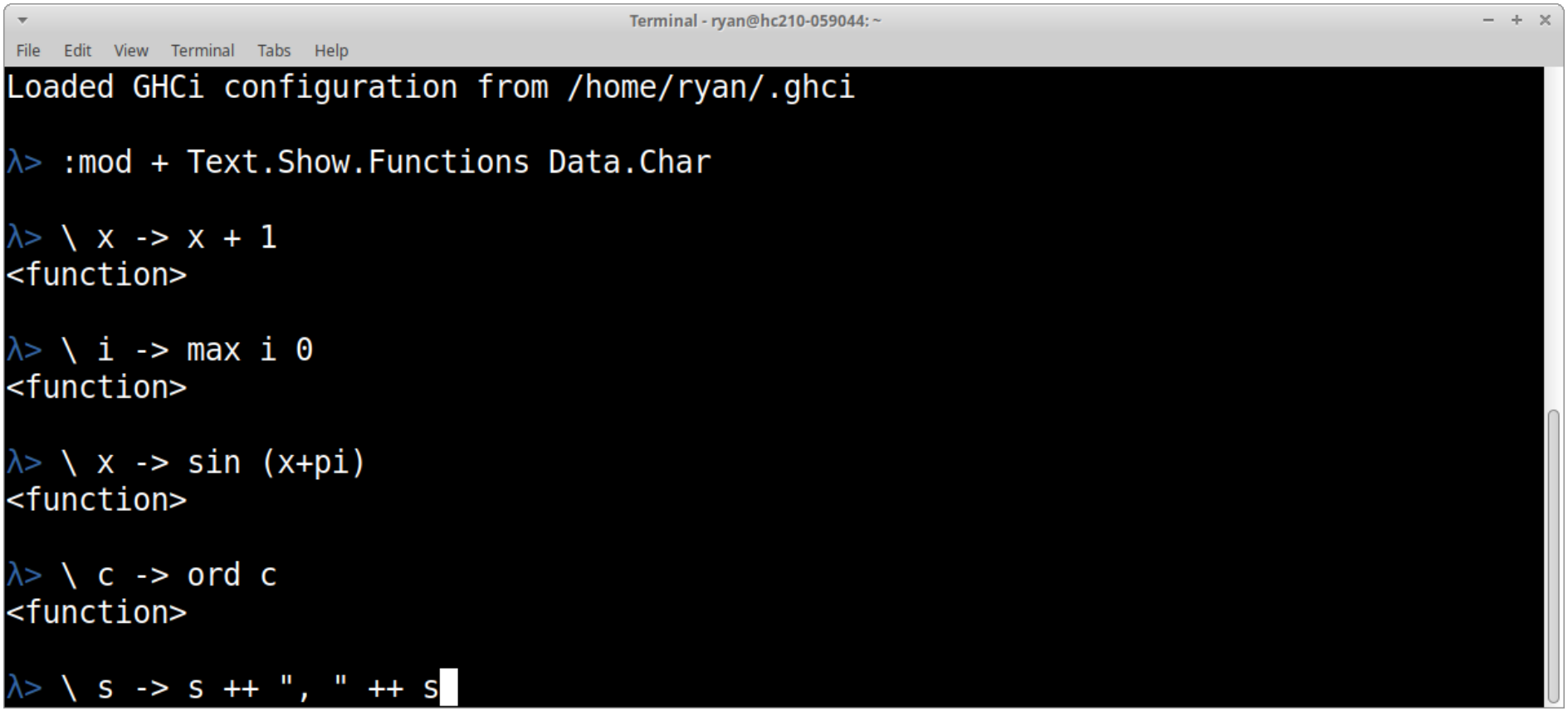
Functions Definitions



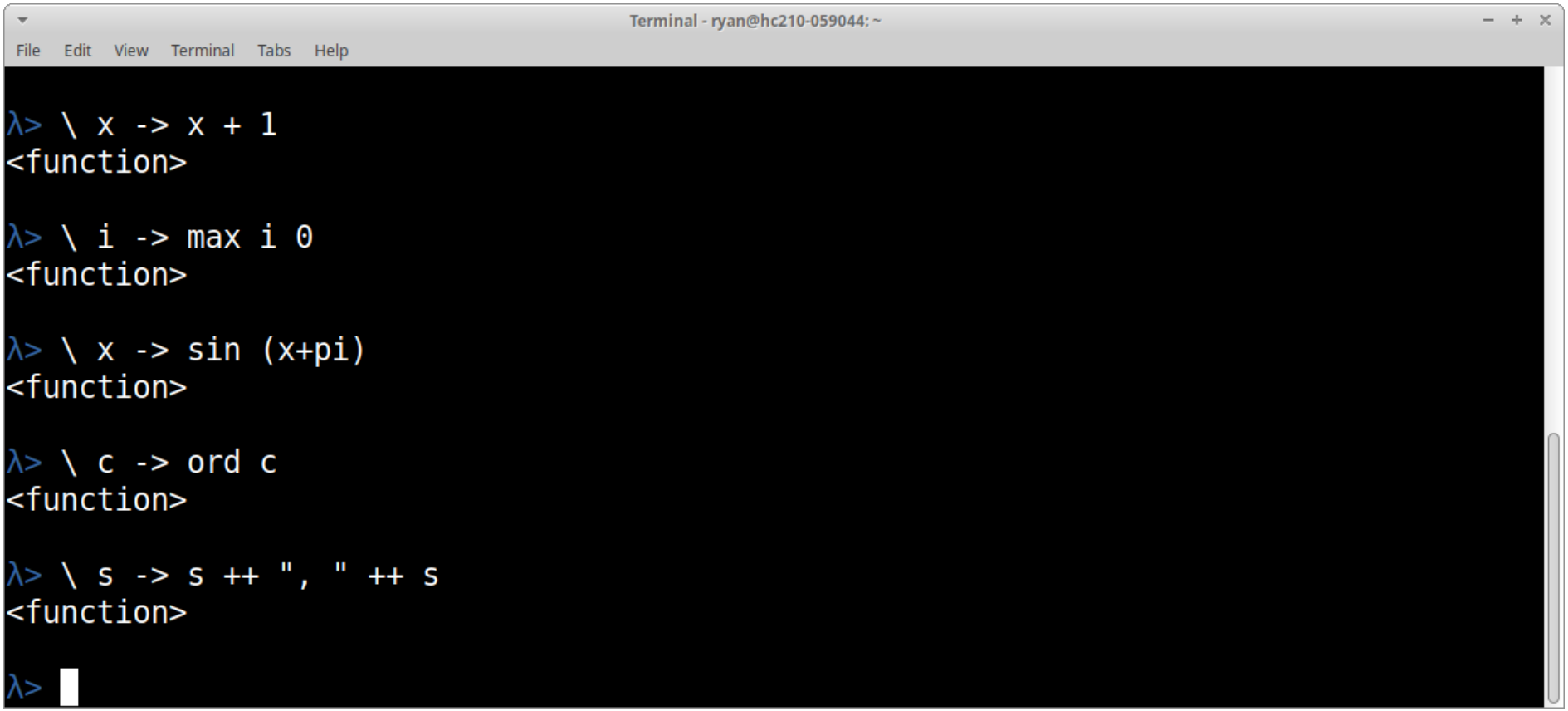
Functions Definitions



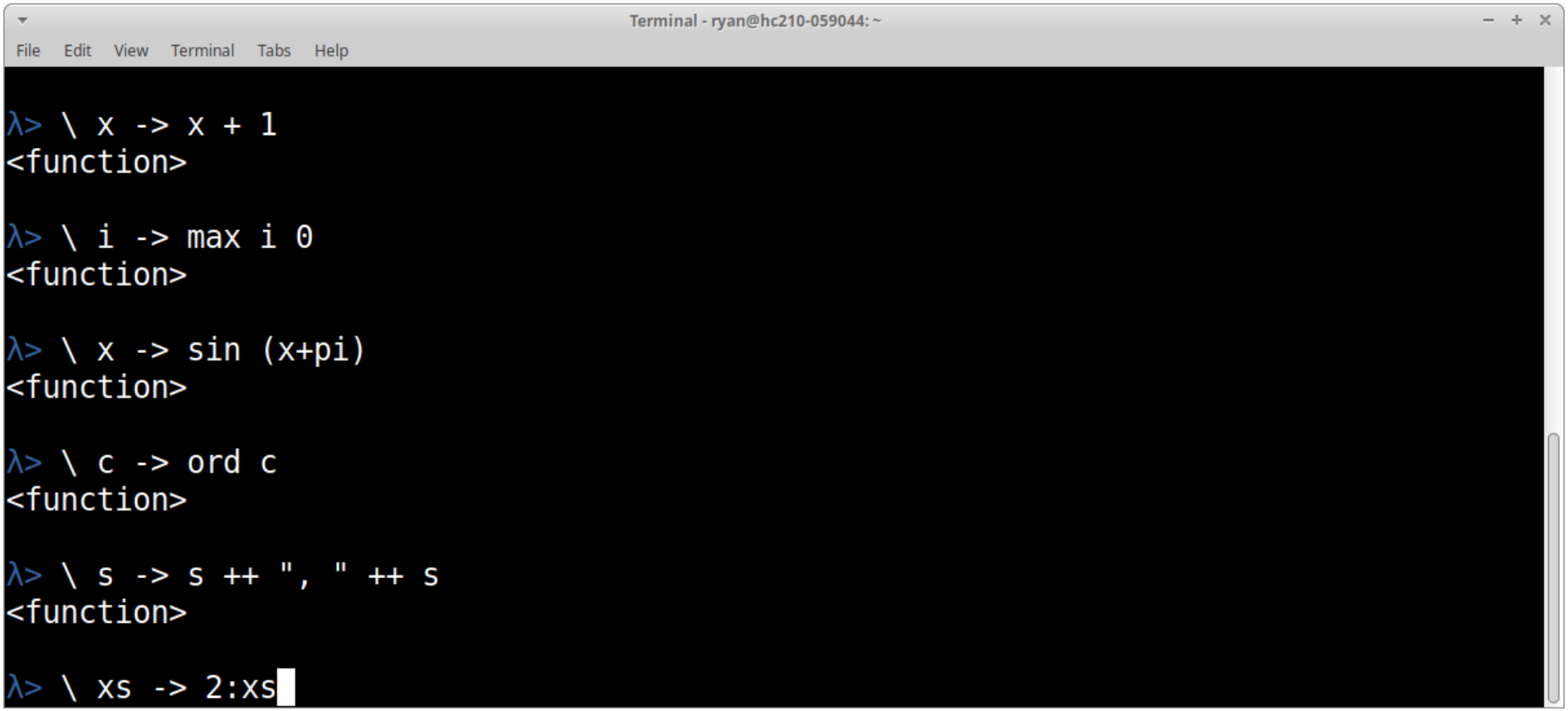
Functions Definitions



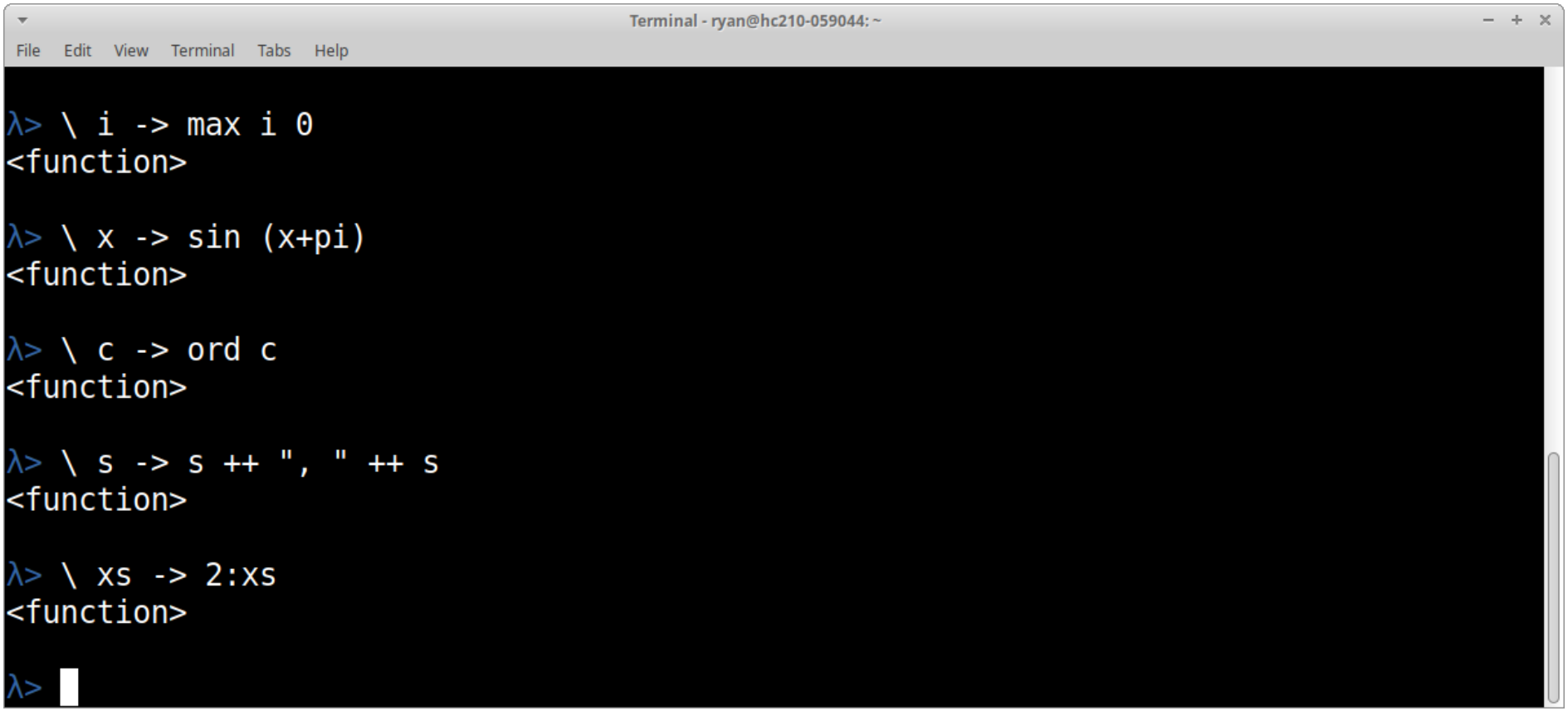
Functions Definitions



Functions Definitions



Functions Definitions



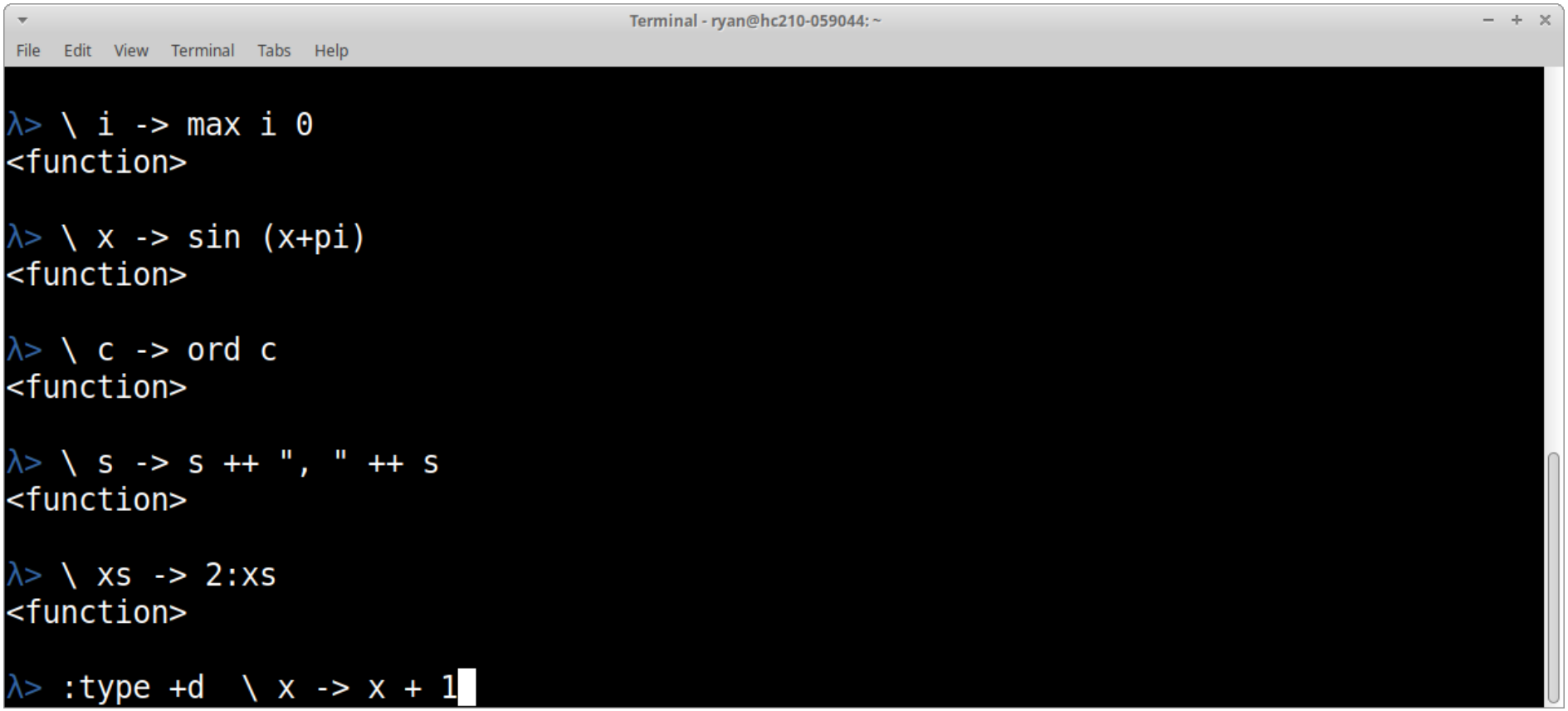
What Can You Do With a Function?

- Integer: get the next integer
- Tuple: get the first and the second part
- List: get the rest of the lsit
- Function: use/apply it to an argument

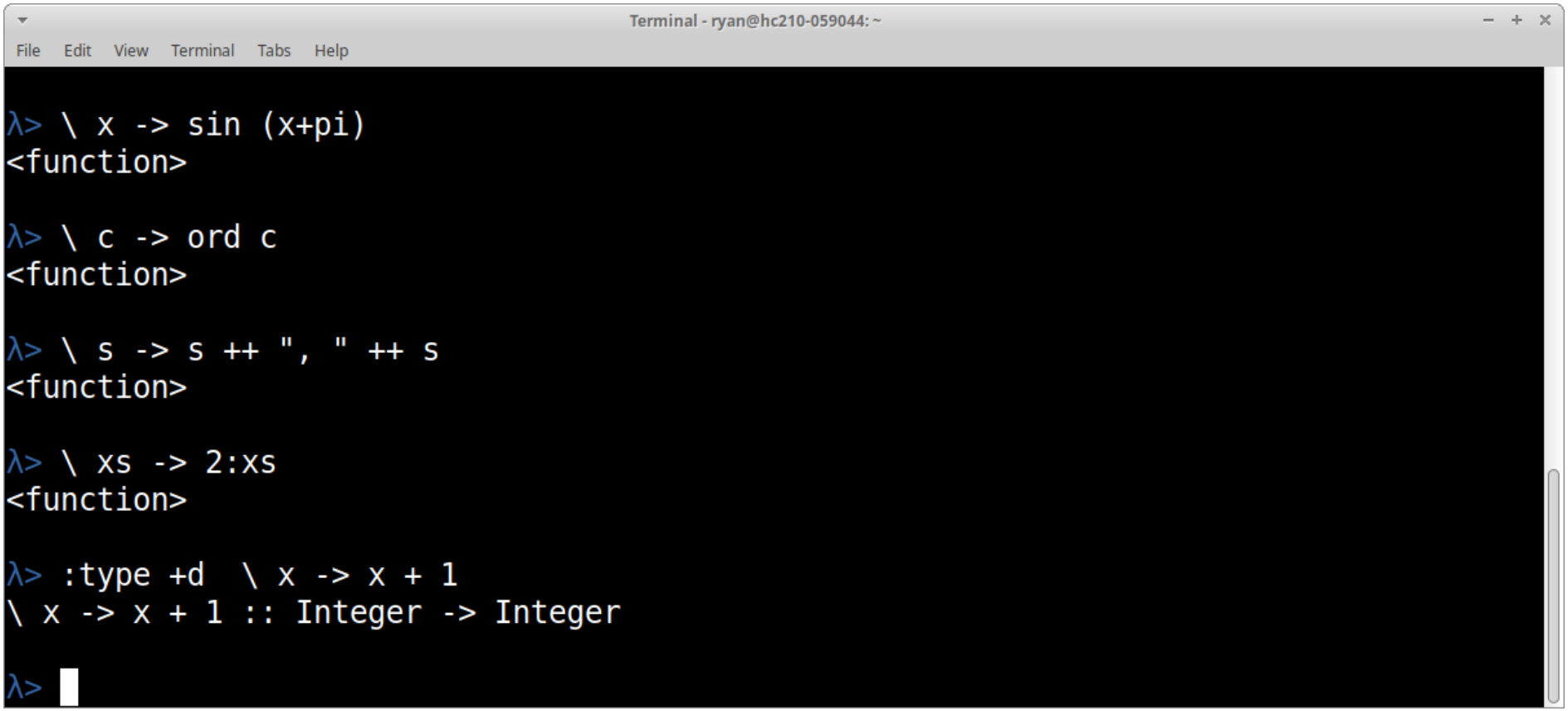
Functions Application

We ask what is the type of the function, and then we apply it to some element of the domain.

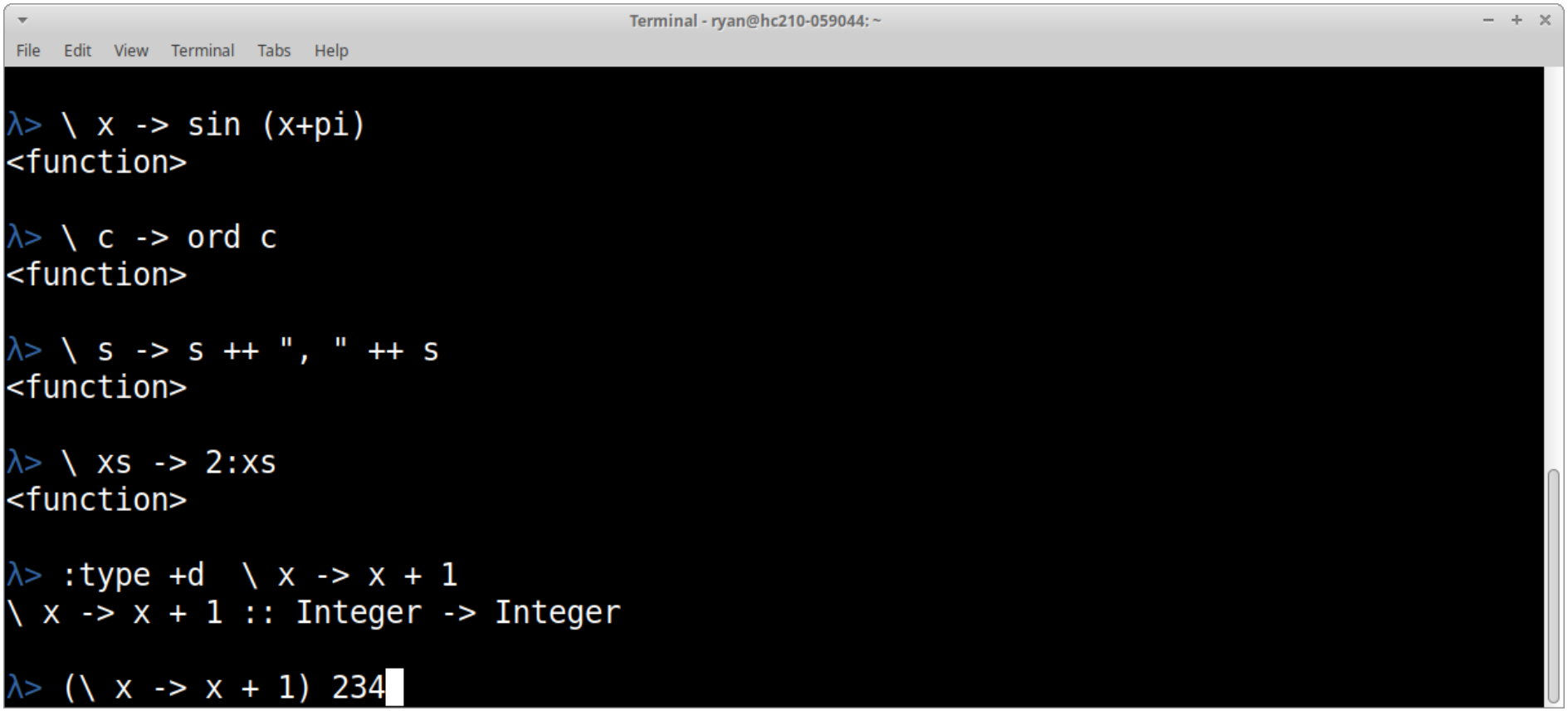
Function Application



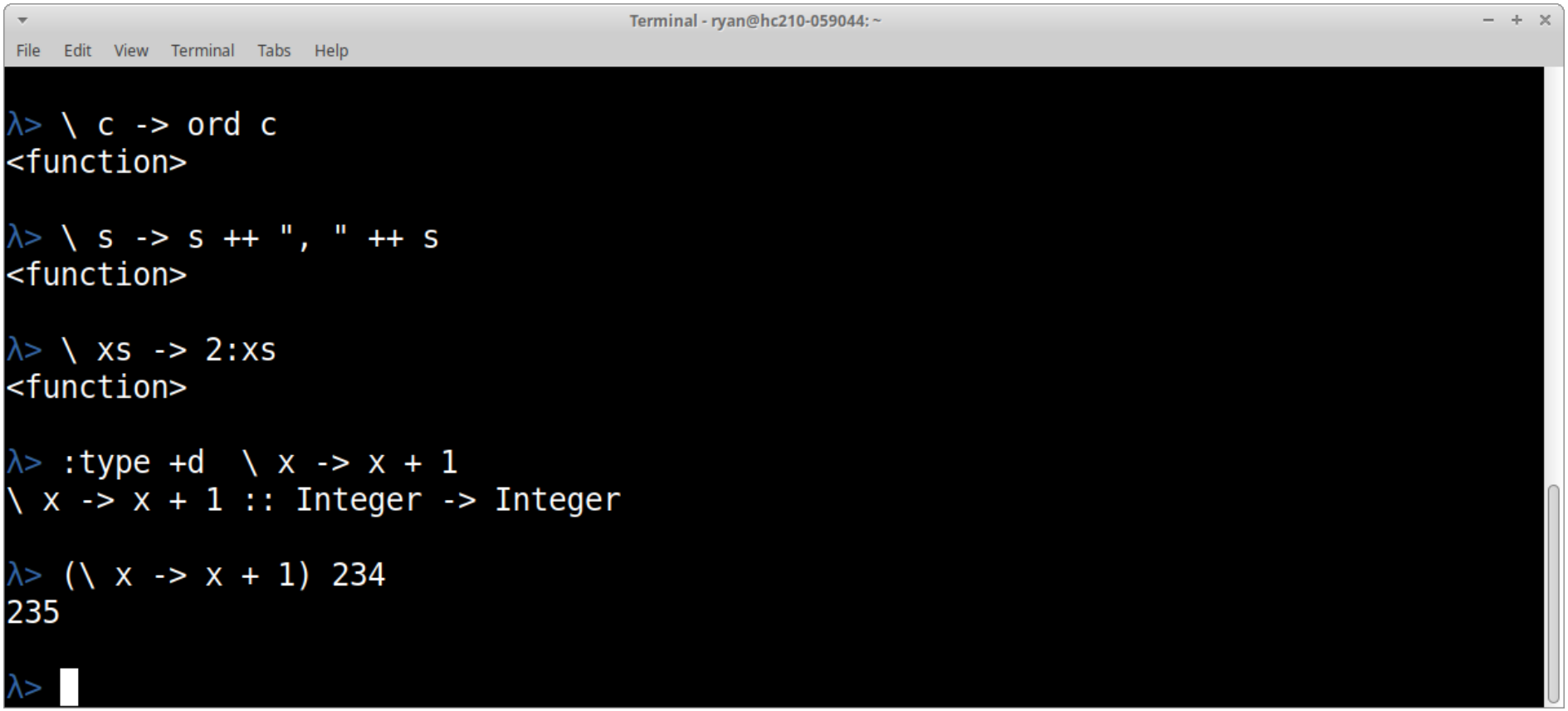
Function Application



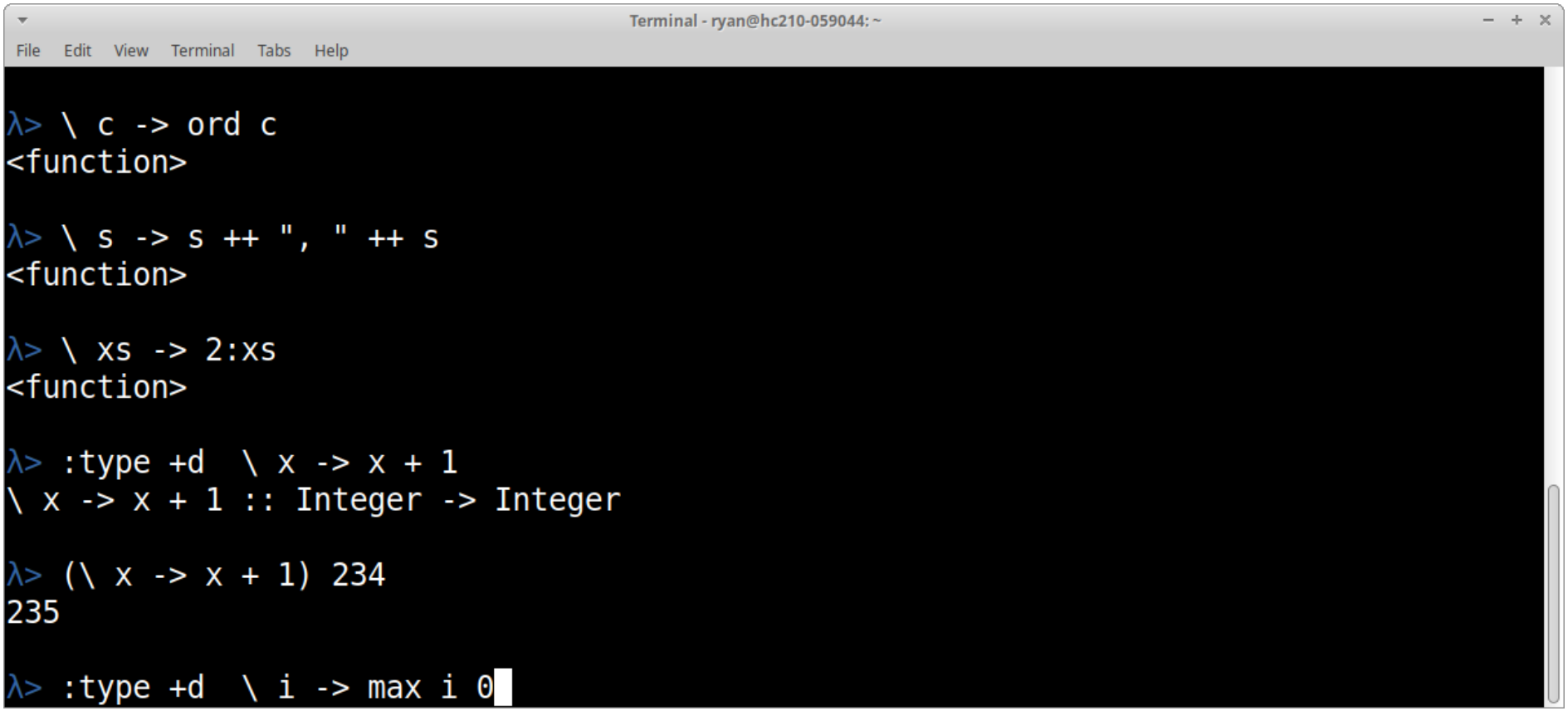
Function Application



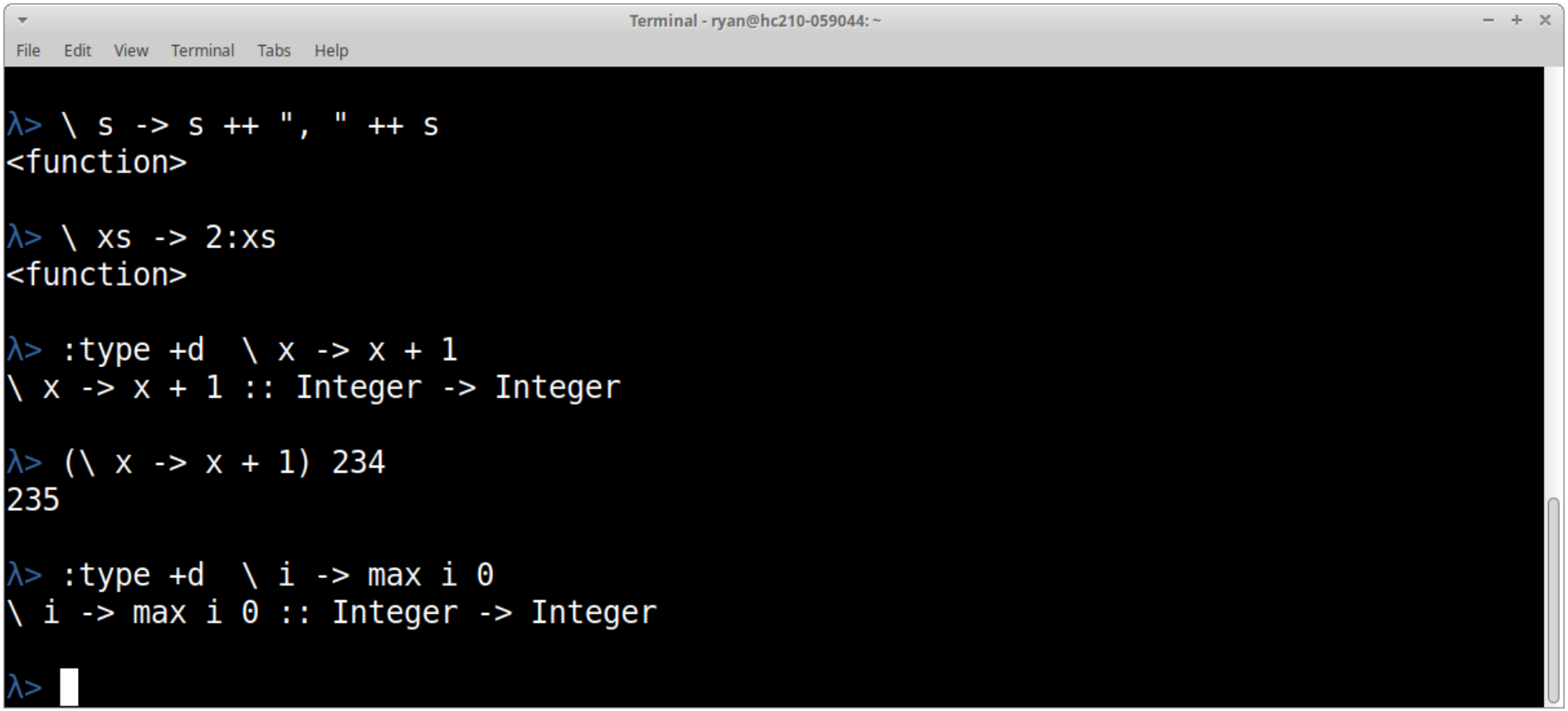
Function Application



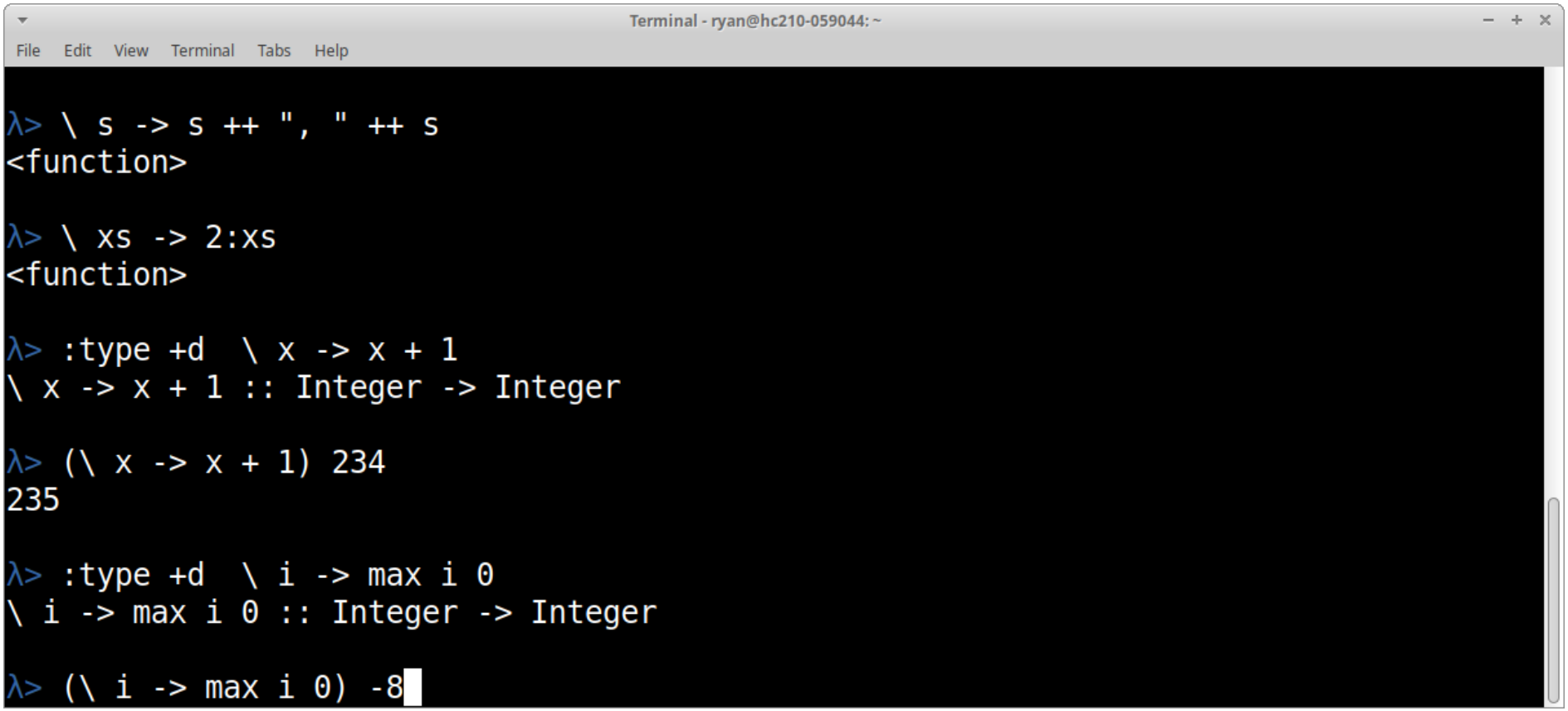
Function Application



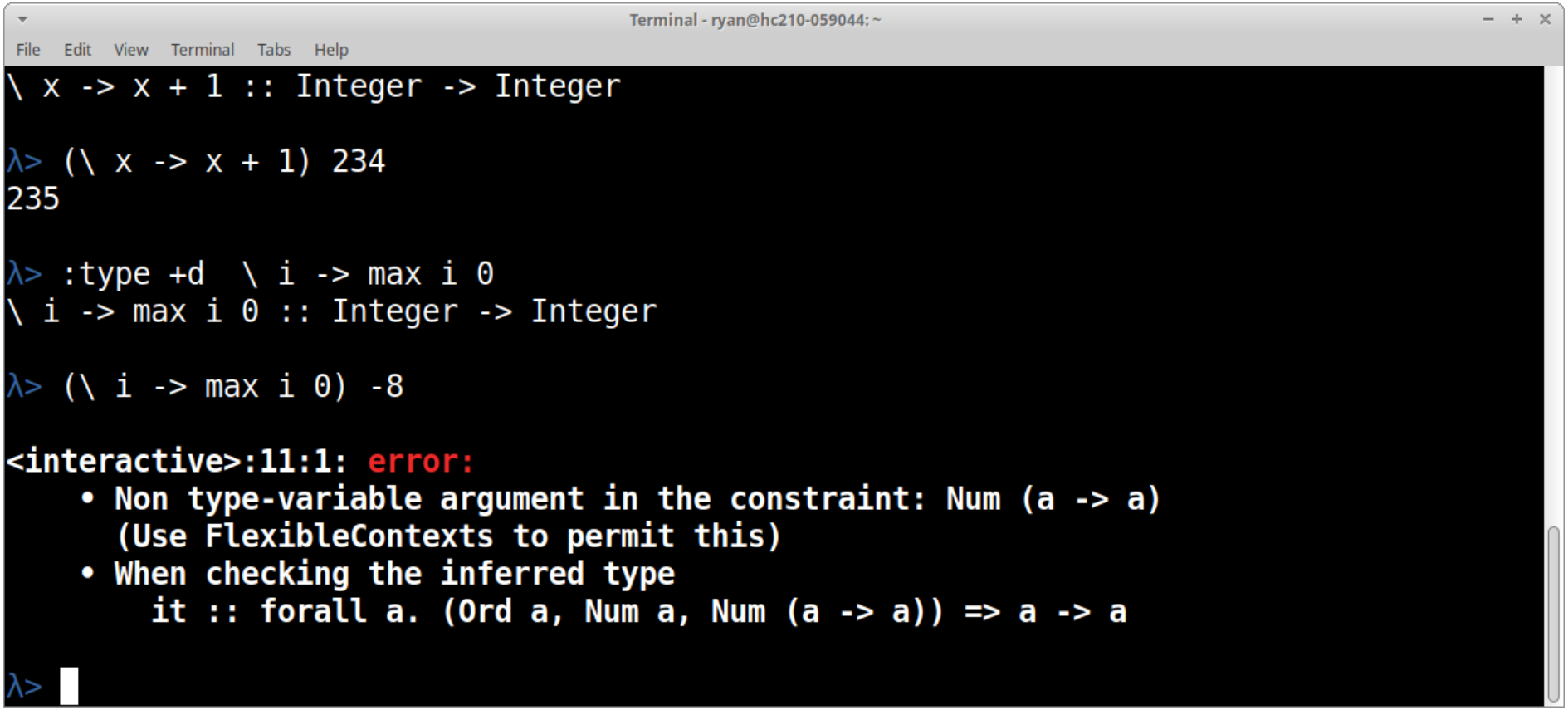
Function Application



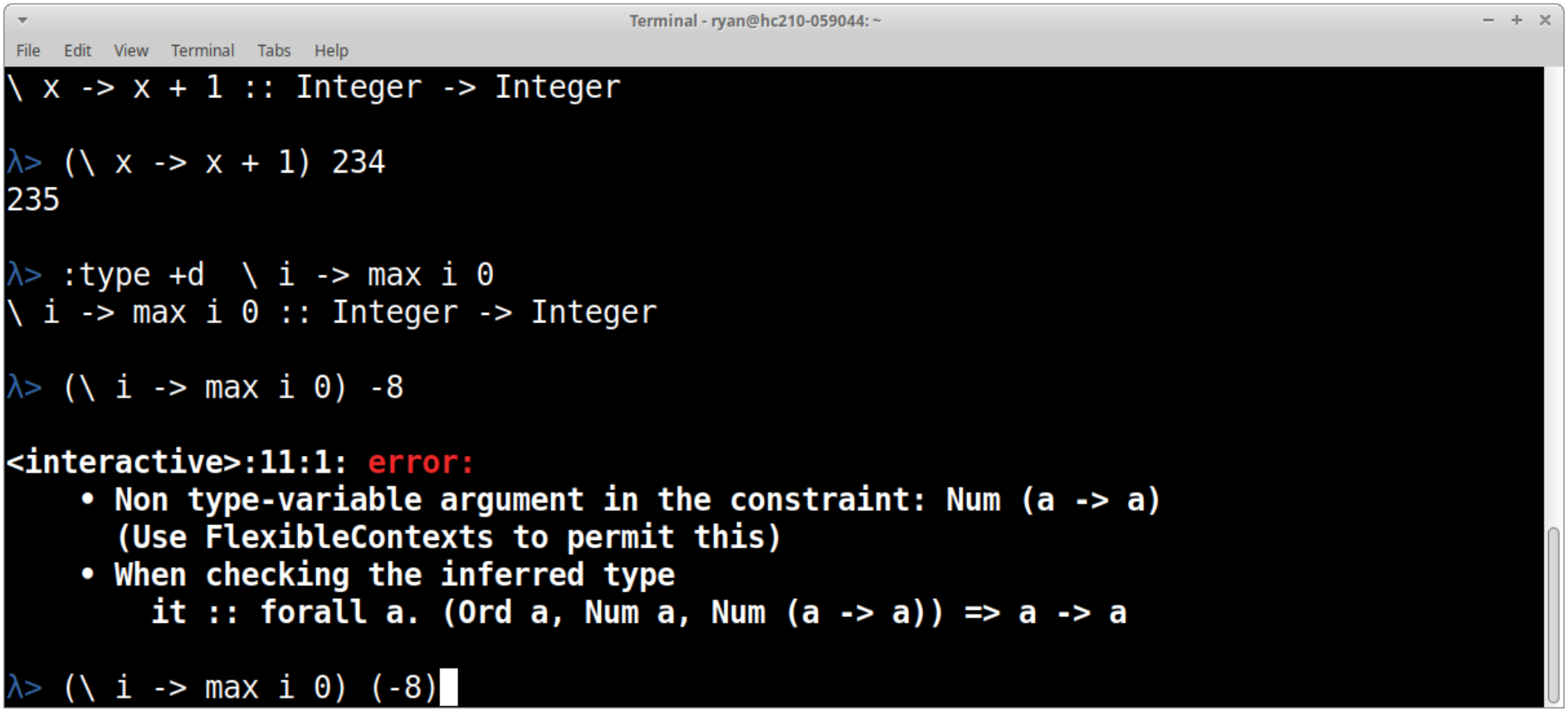
Function Application



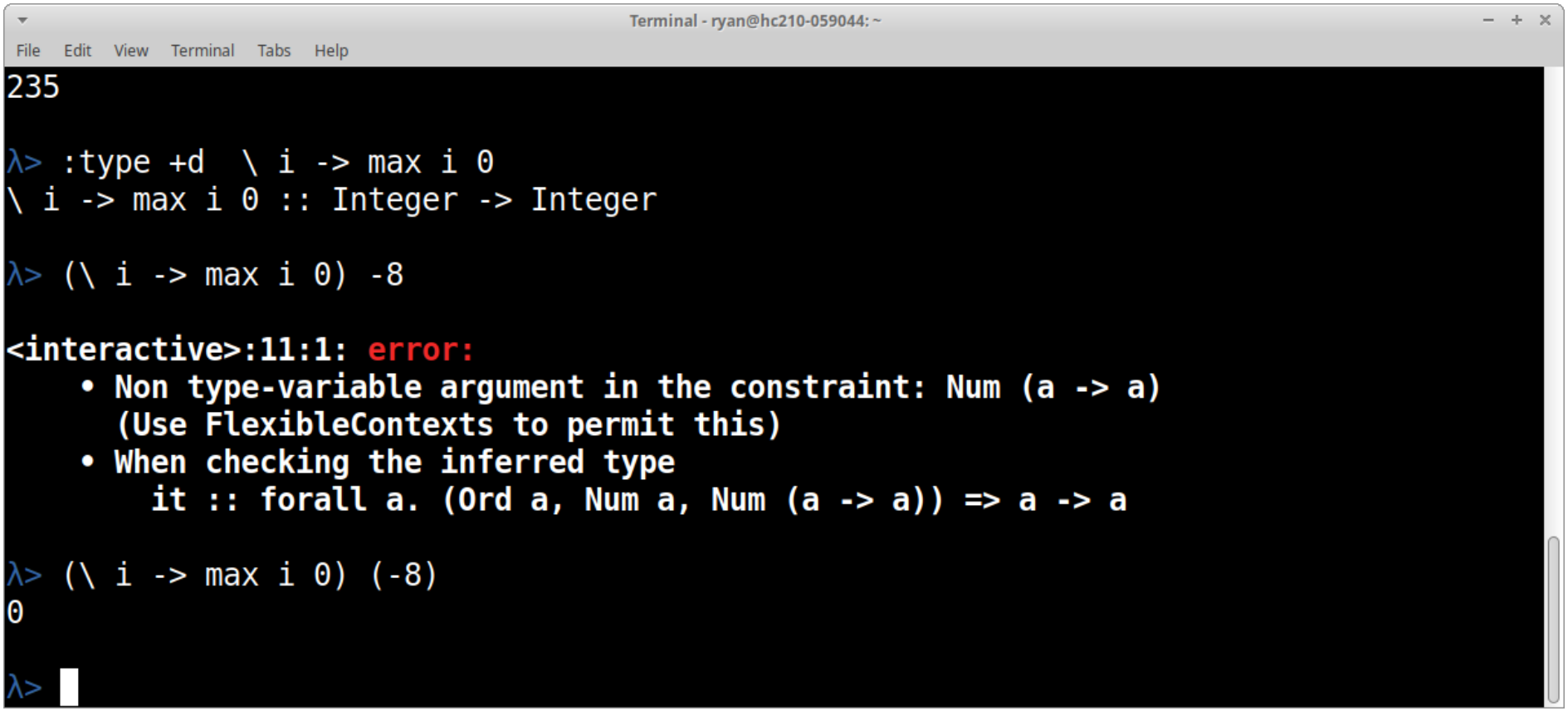
Function Application



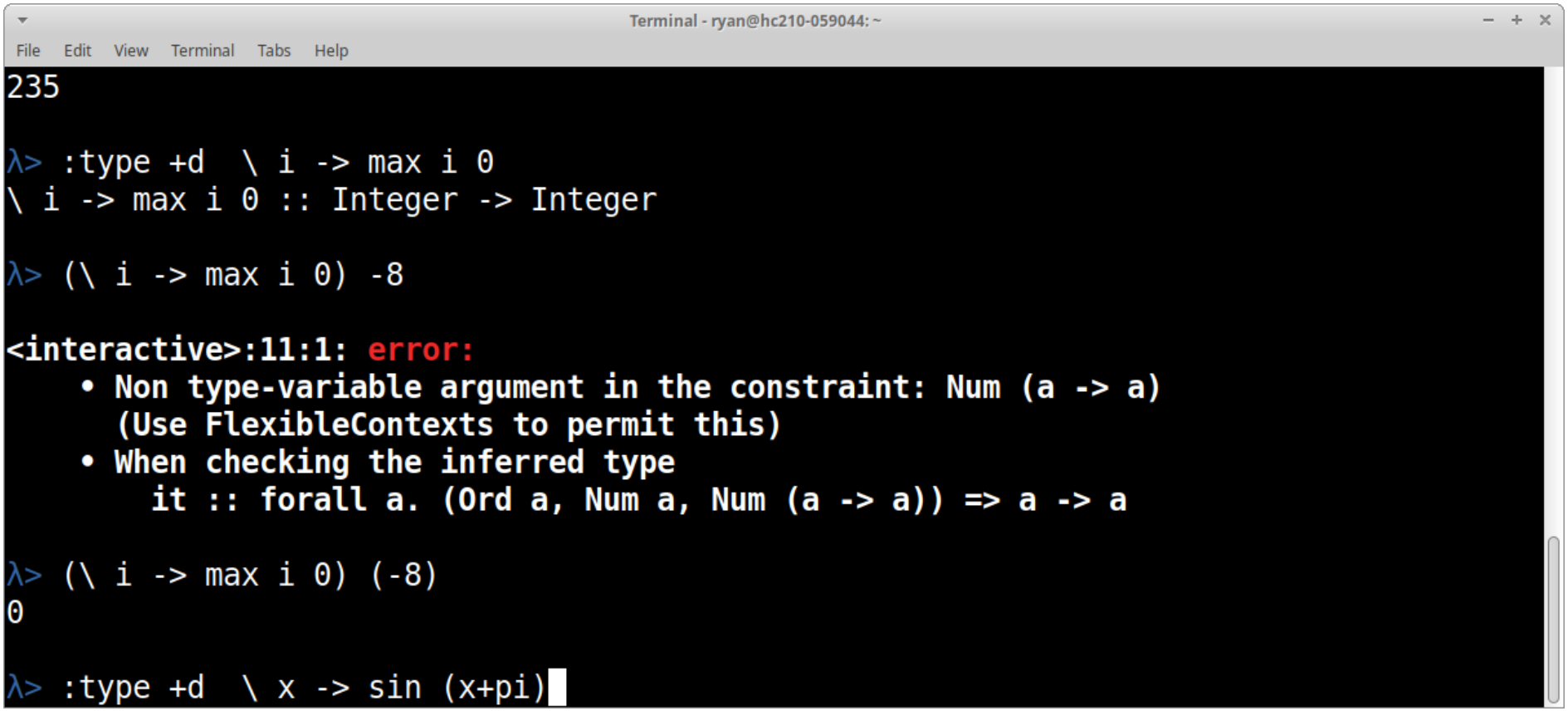
Function Application



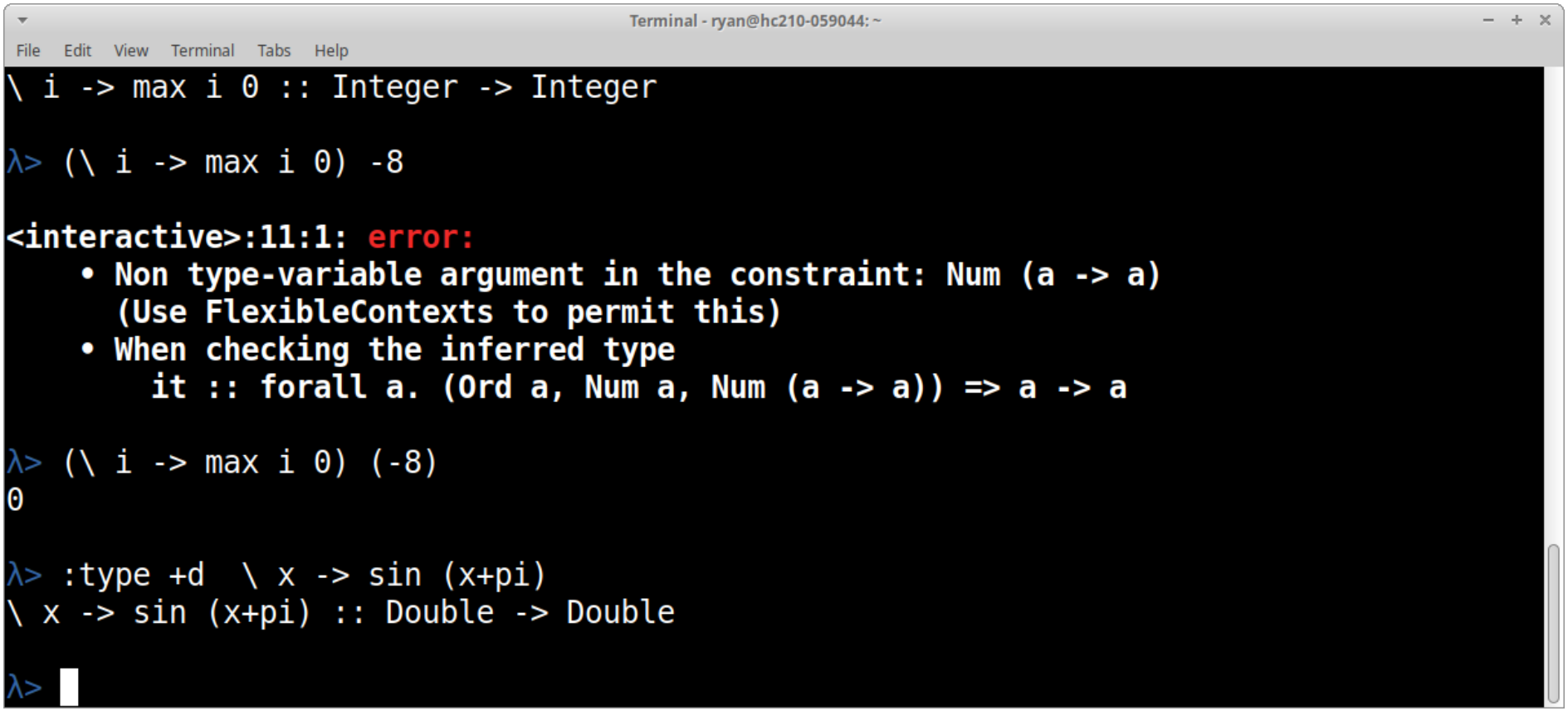
Function Application



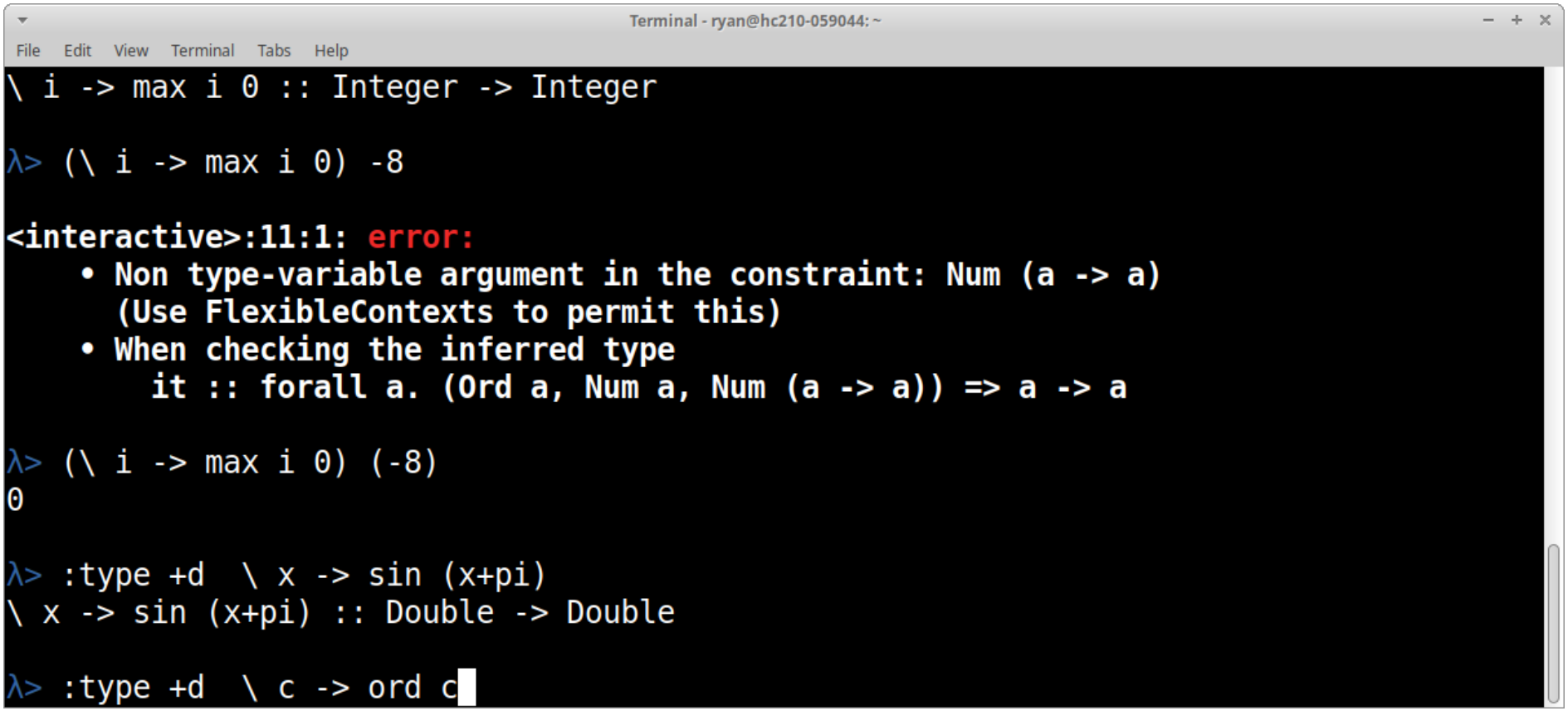
Function Application



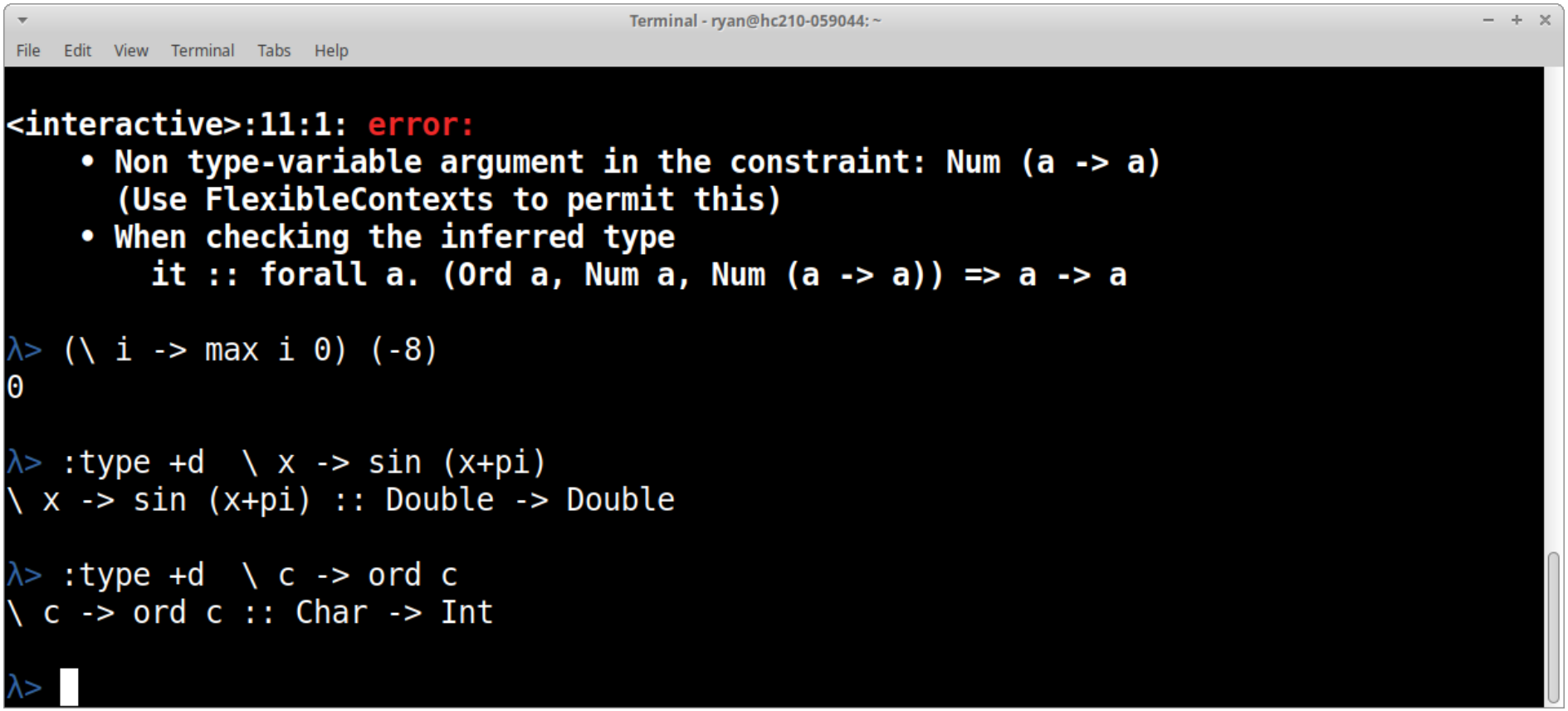
Function Application



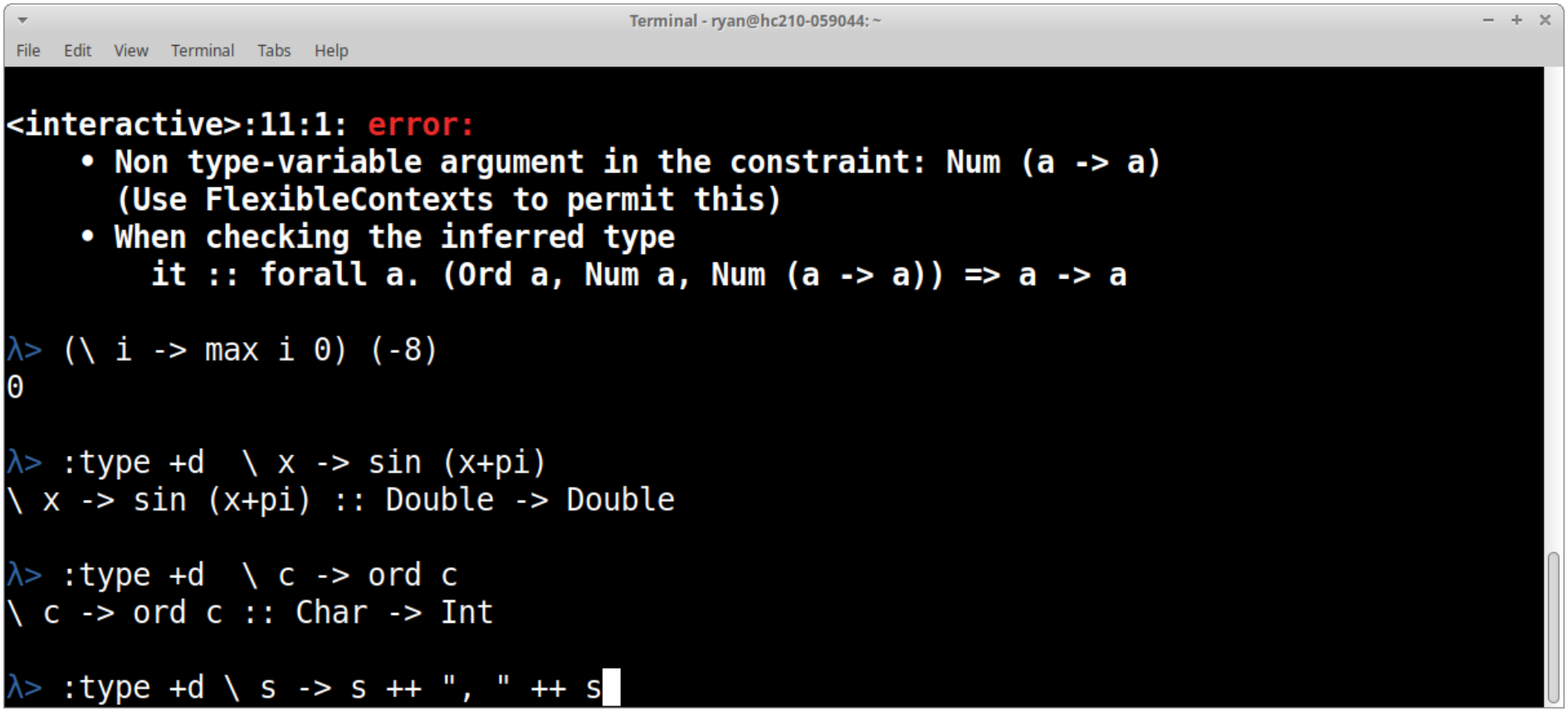
Function Application



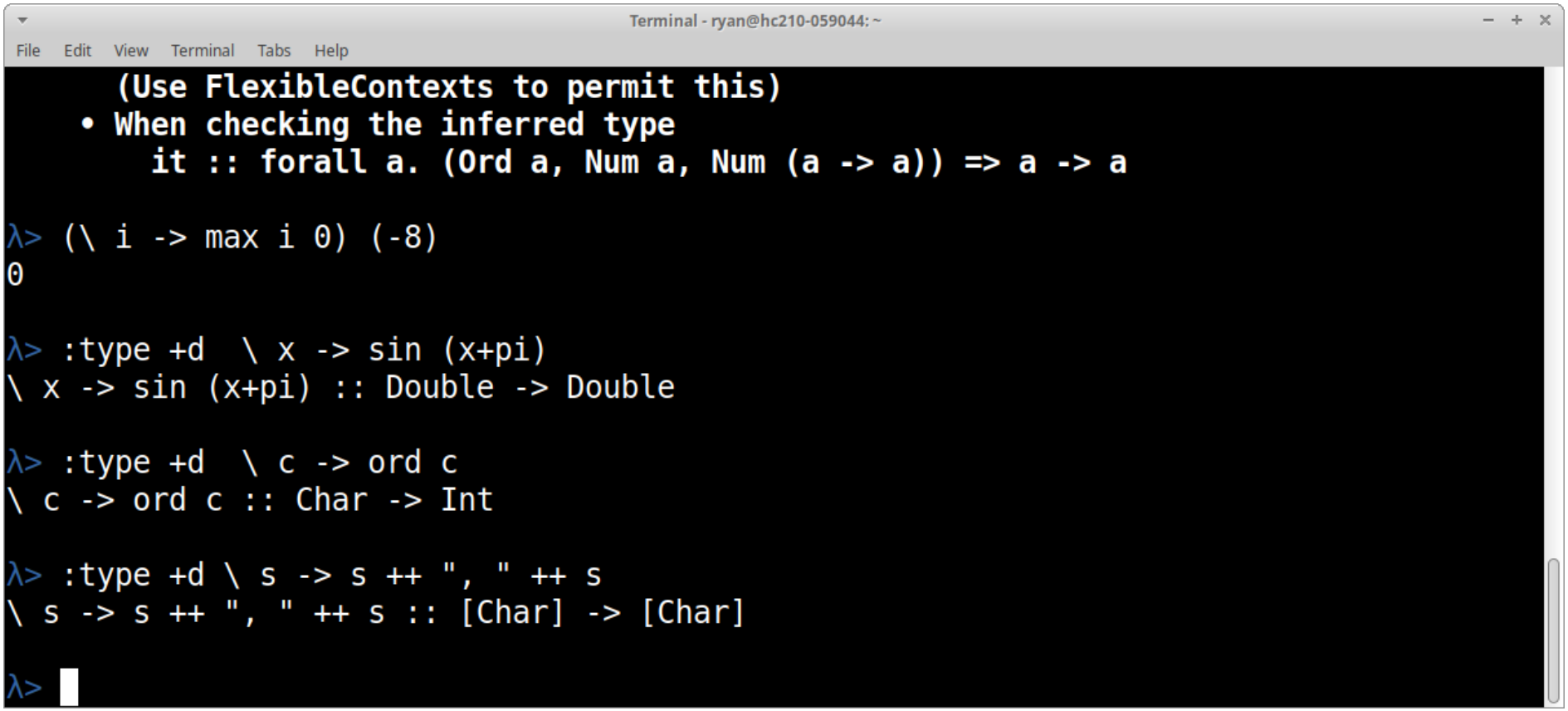
Function Application



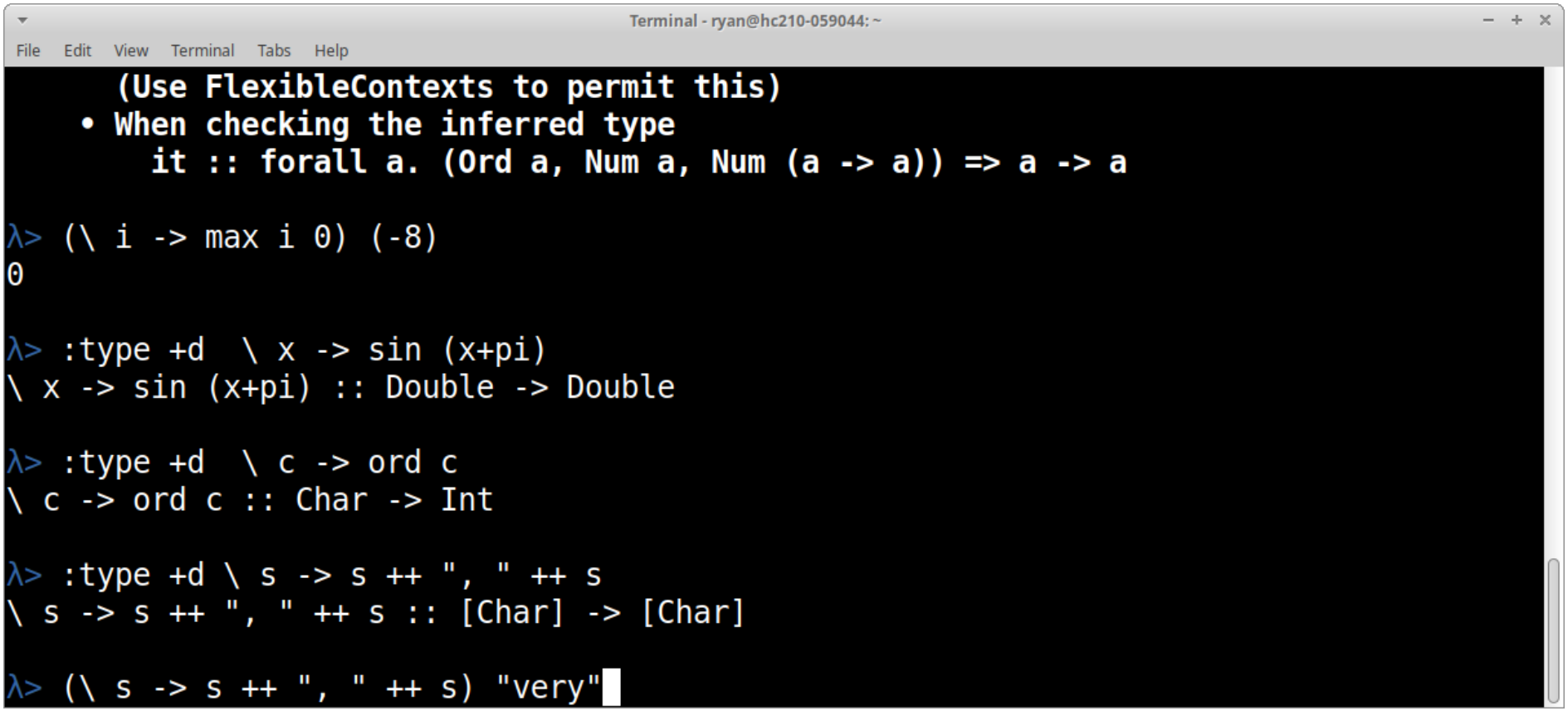
Function Application



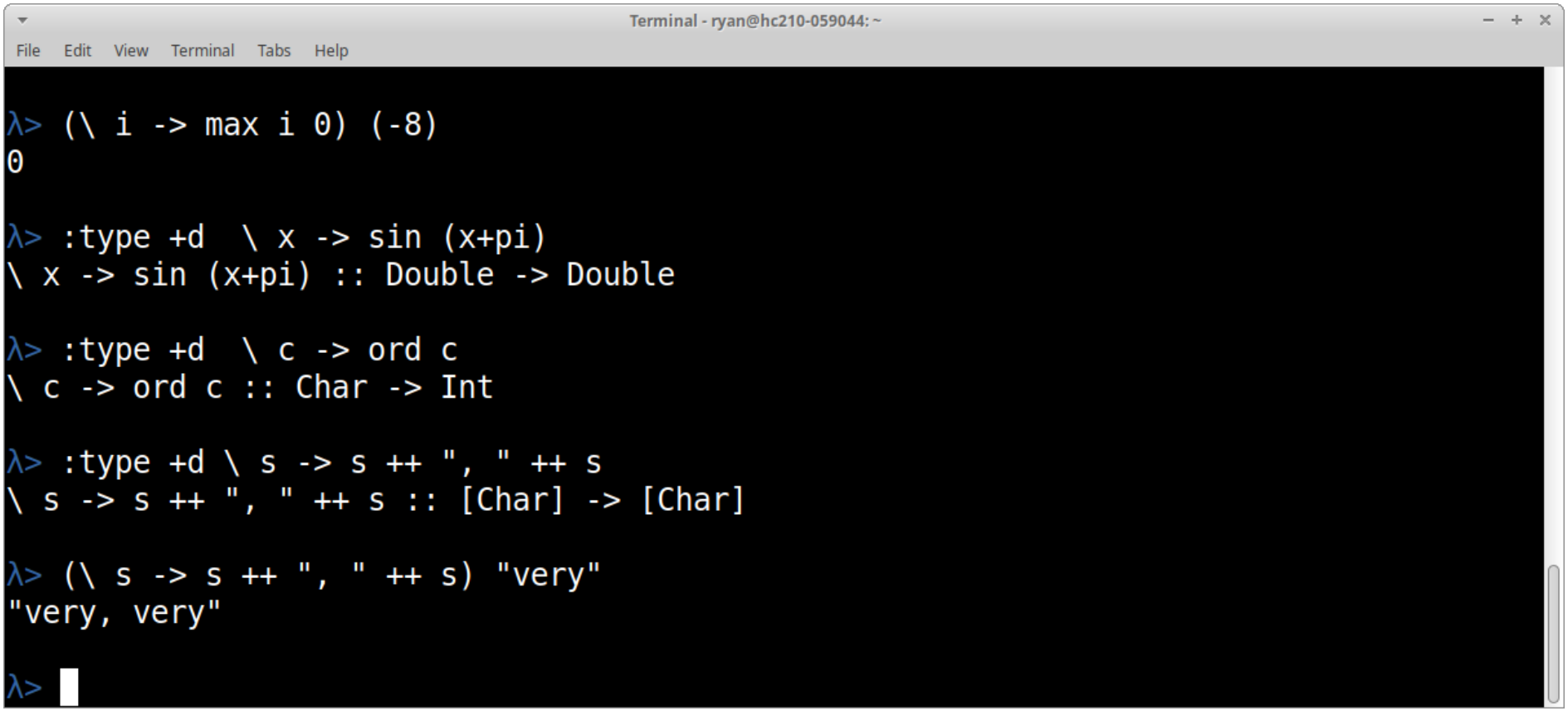
Function Application



Function Application



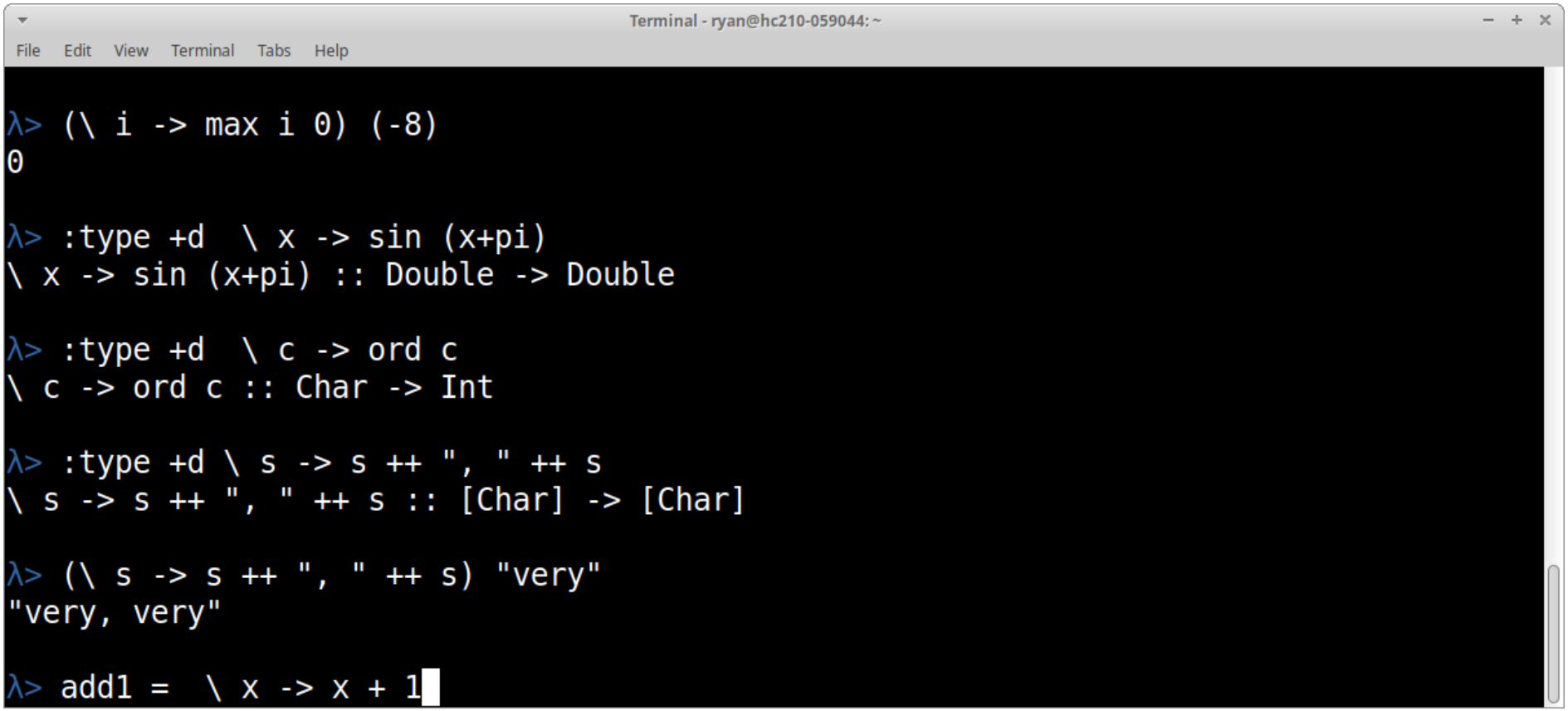
Function Application



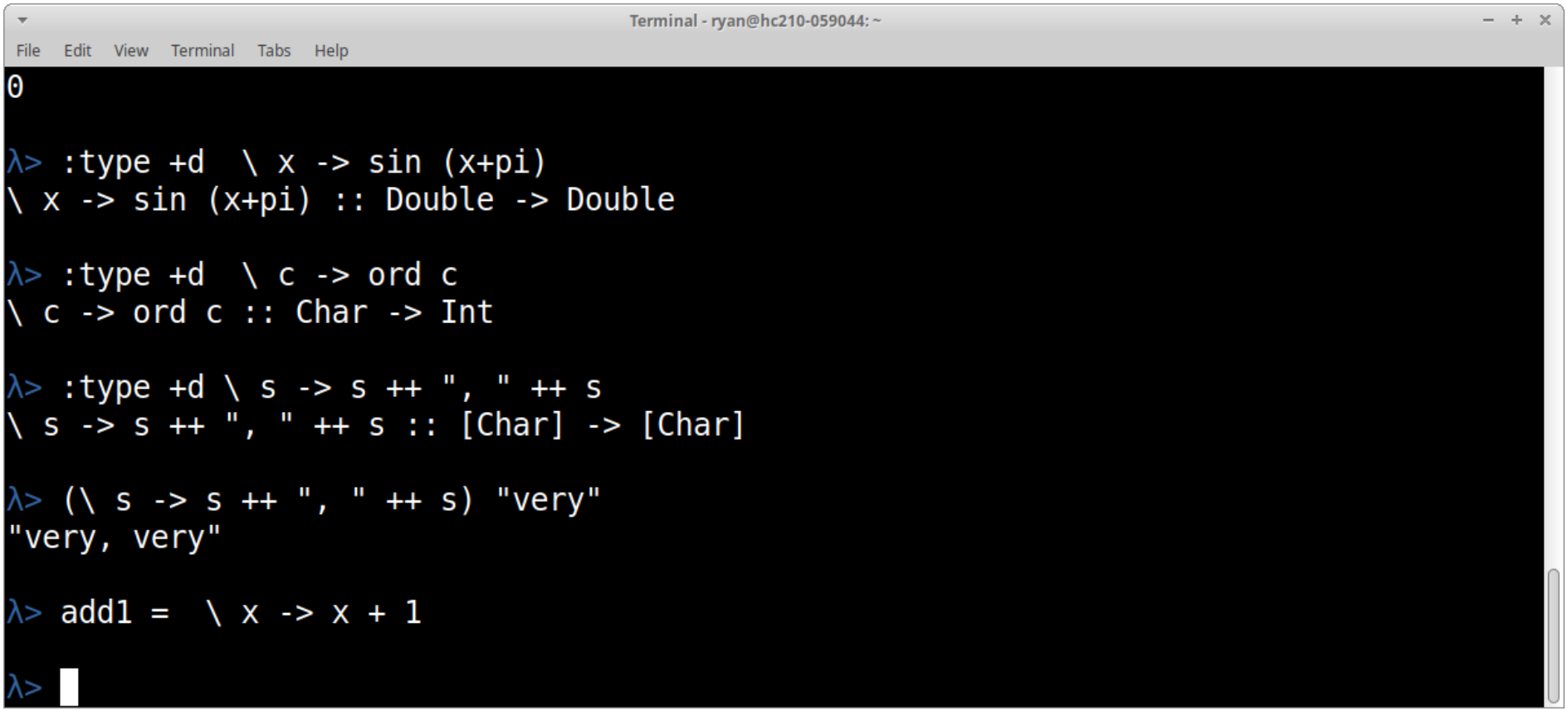
How do functions get names?
The same way anything gets a name!

```
add1 = \ x -> x + 1

cutOff = \ i -> max i 0

g = \ x -> sin (x+pi)

f = \ c -> ord c

double = \ s -> s ++ ", " ++ s

cons2 = \ xs -> 2:xs
```

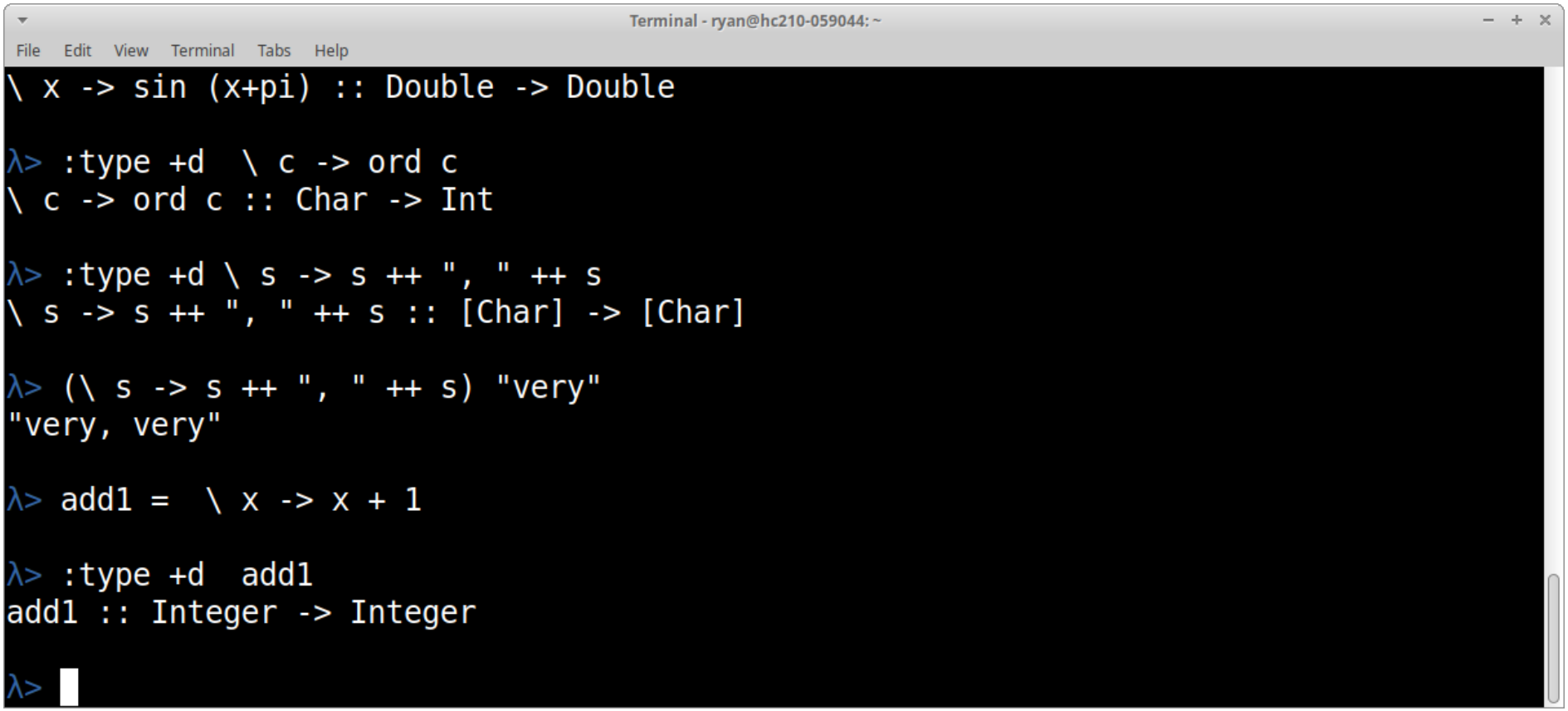
Naming Functions



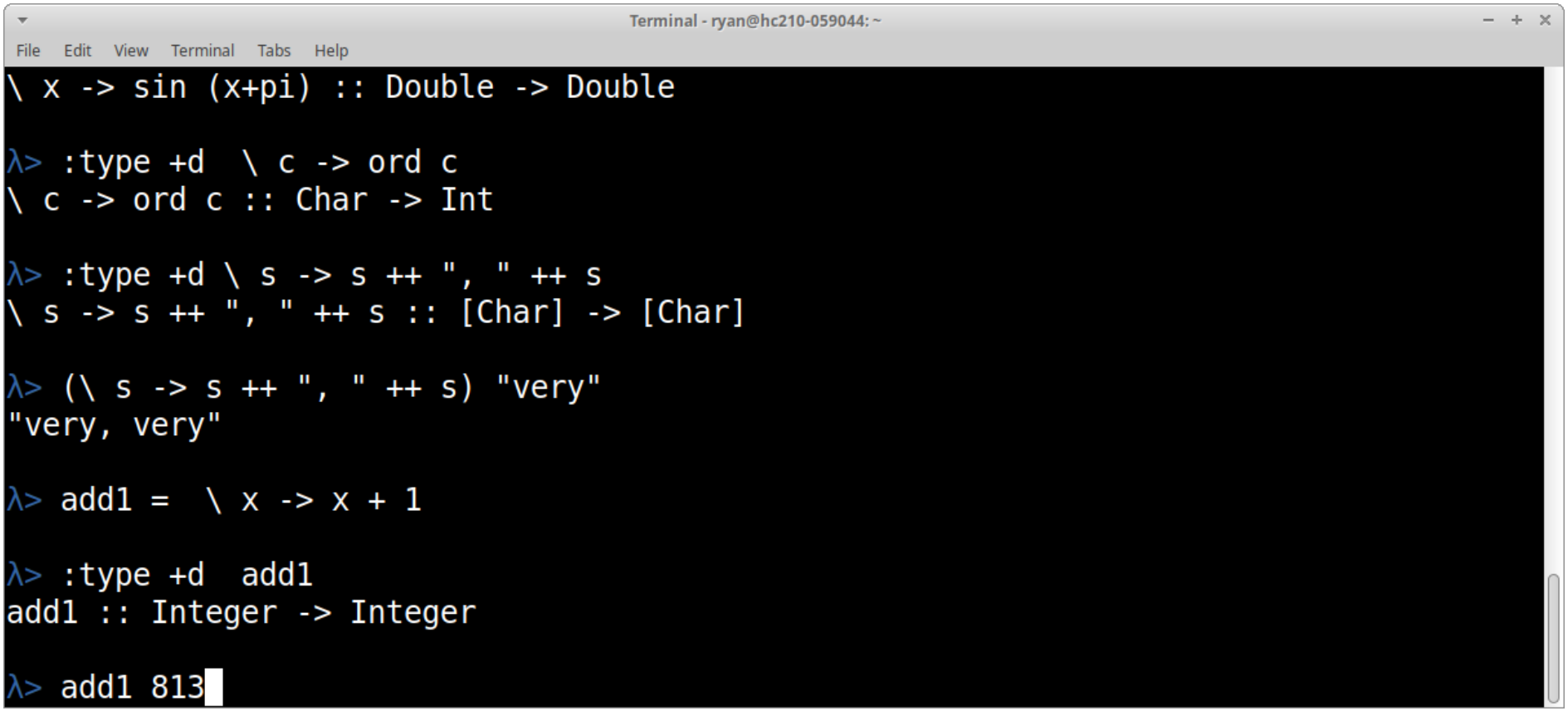
Naming Functions



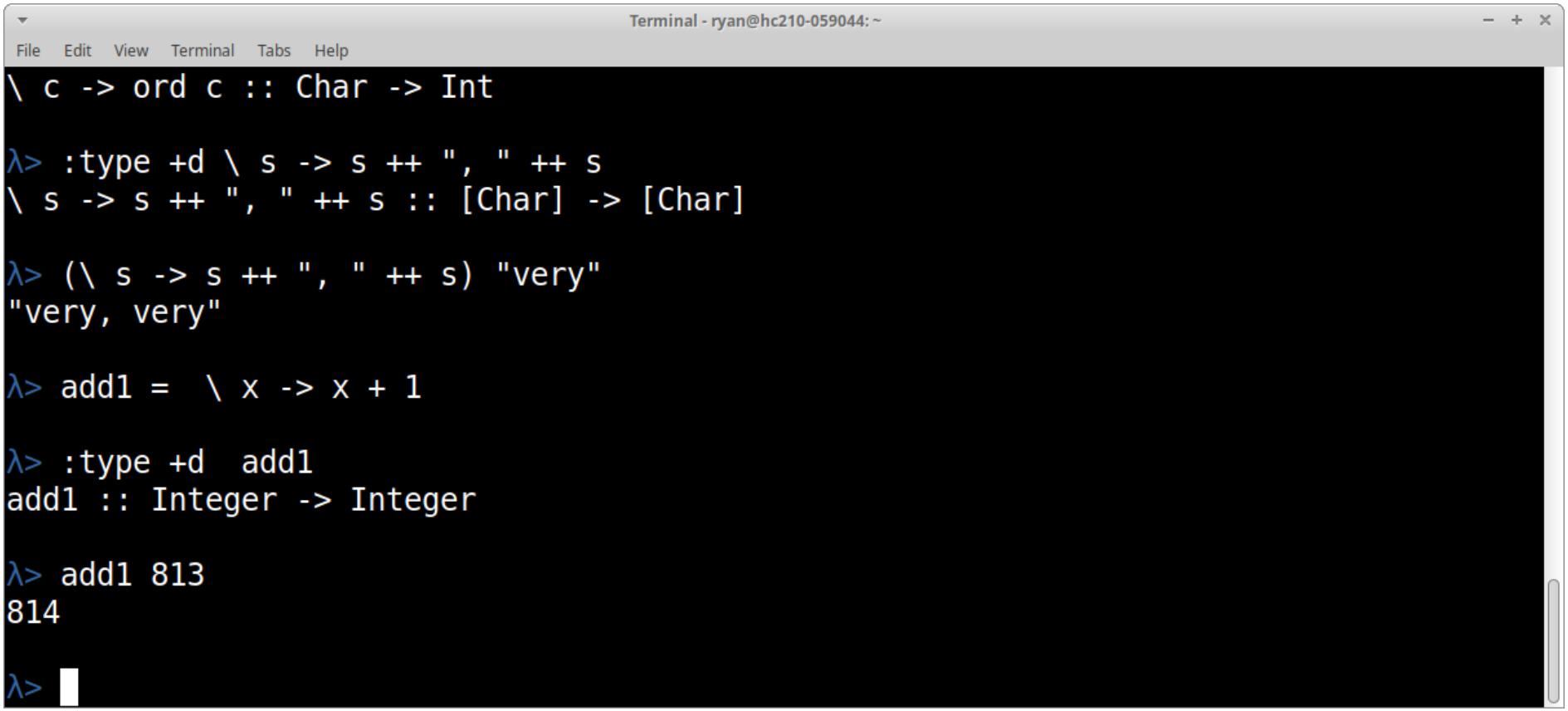
Naming Functions



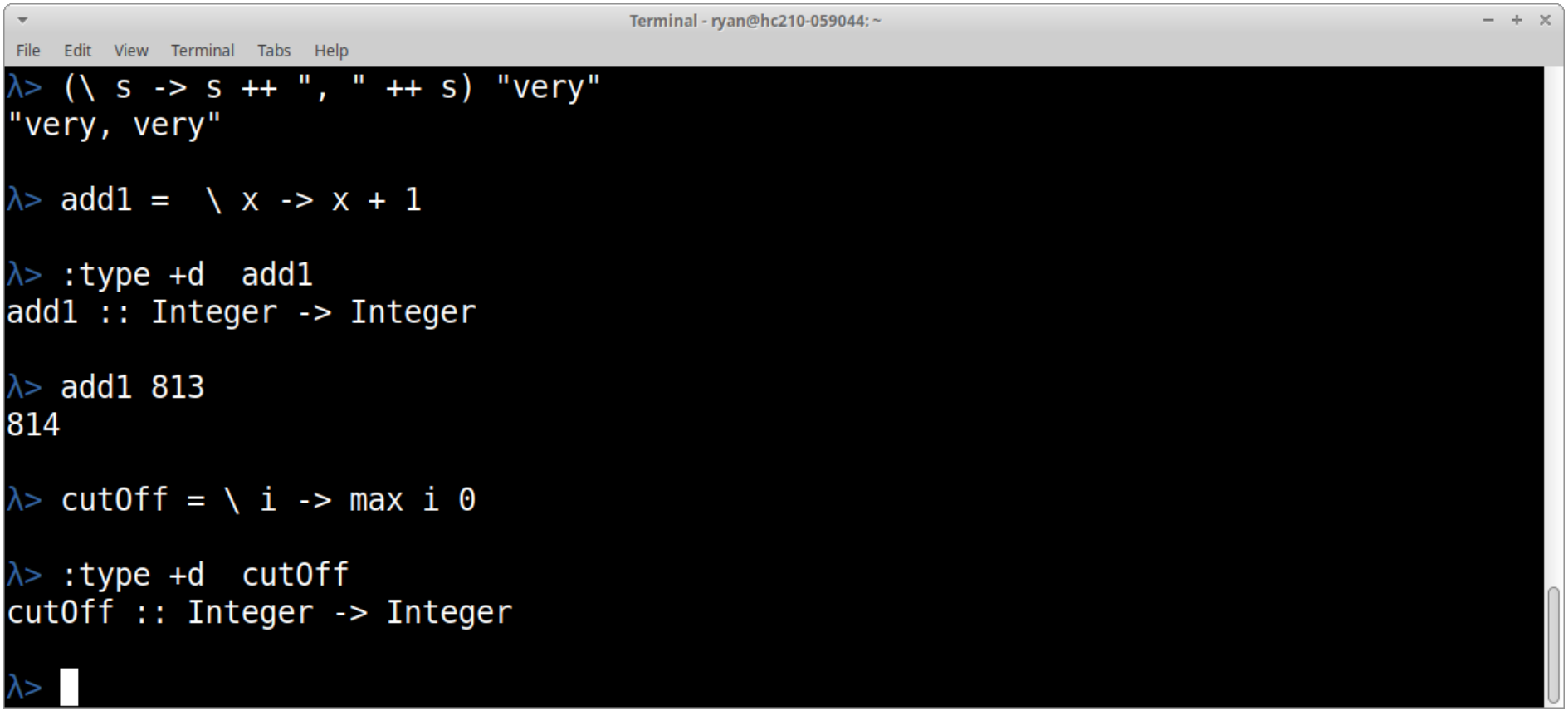
Naming Functions



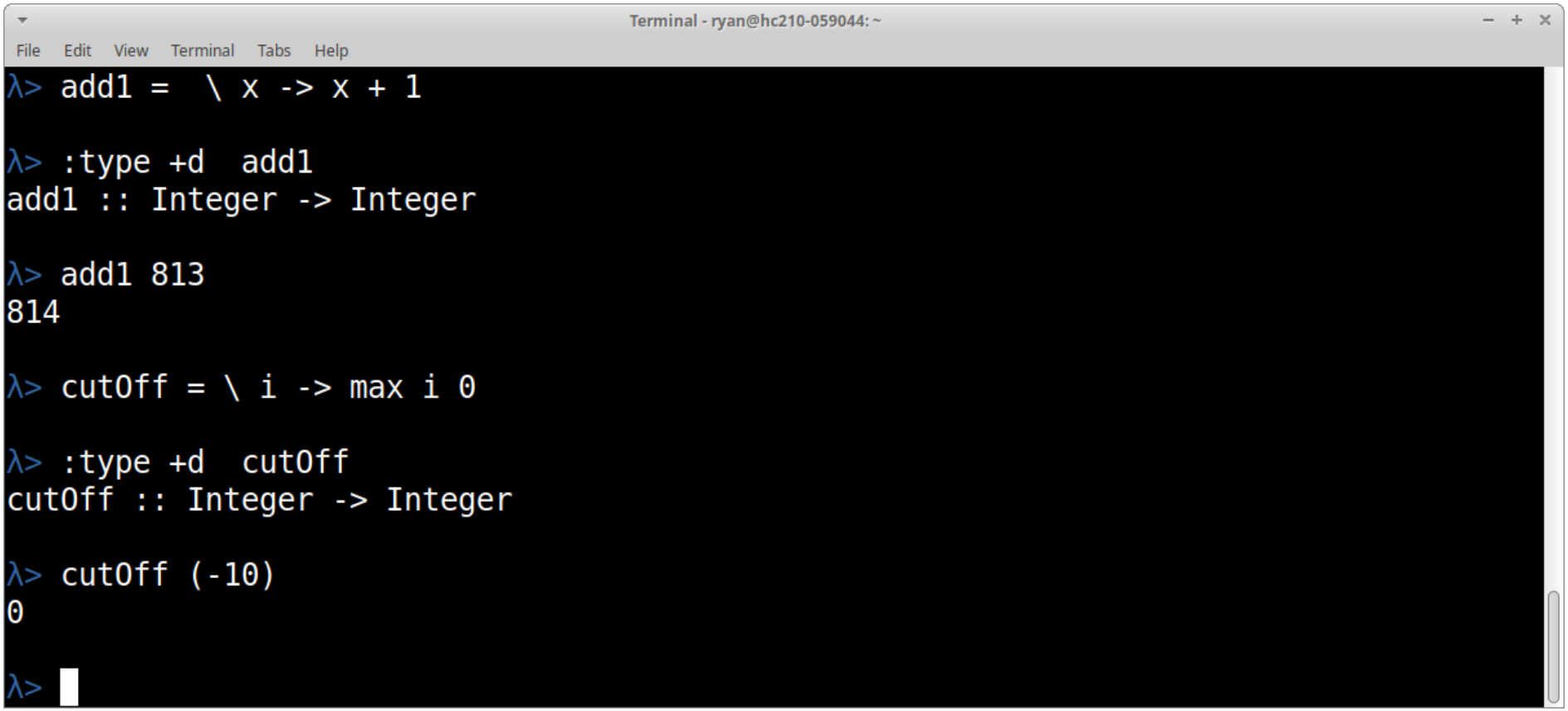
Naming Functions



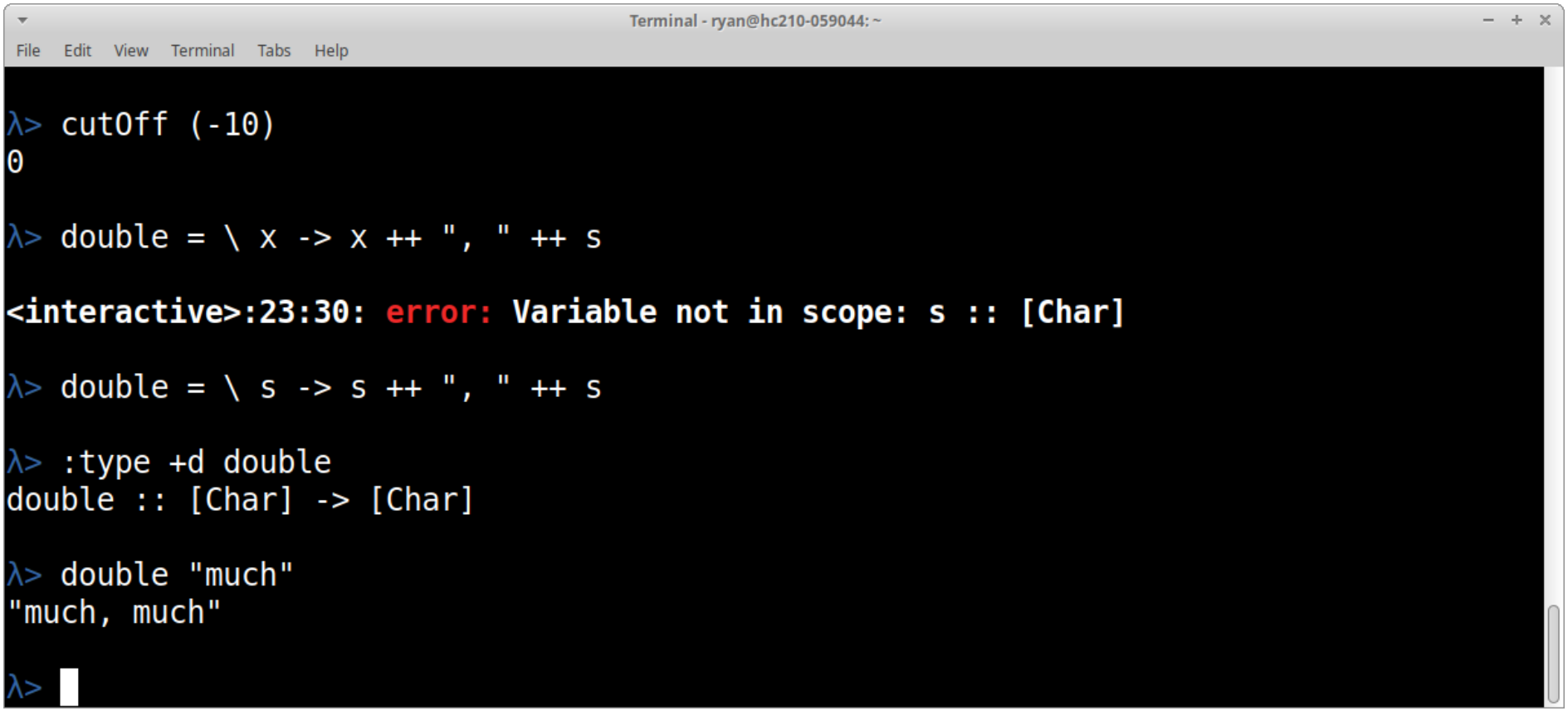
Naming Functions



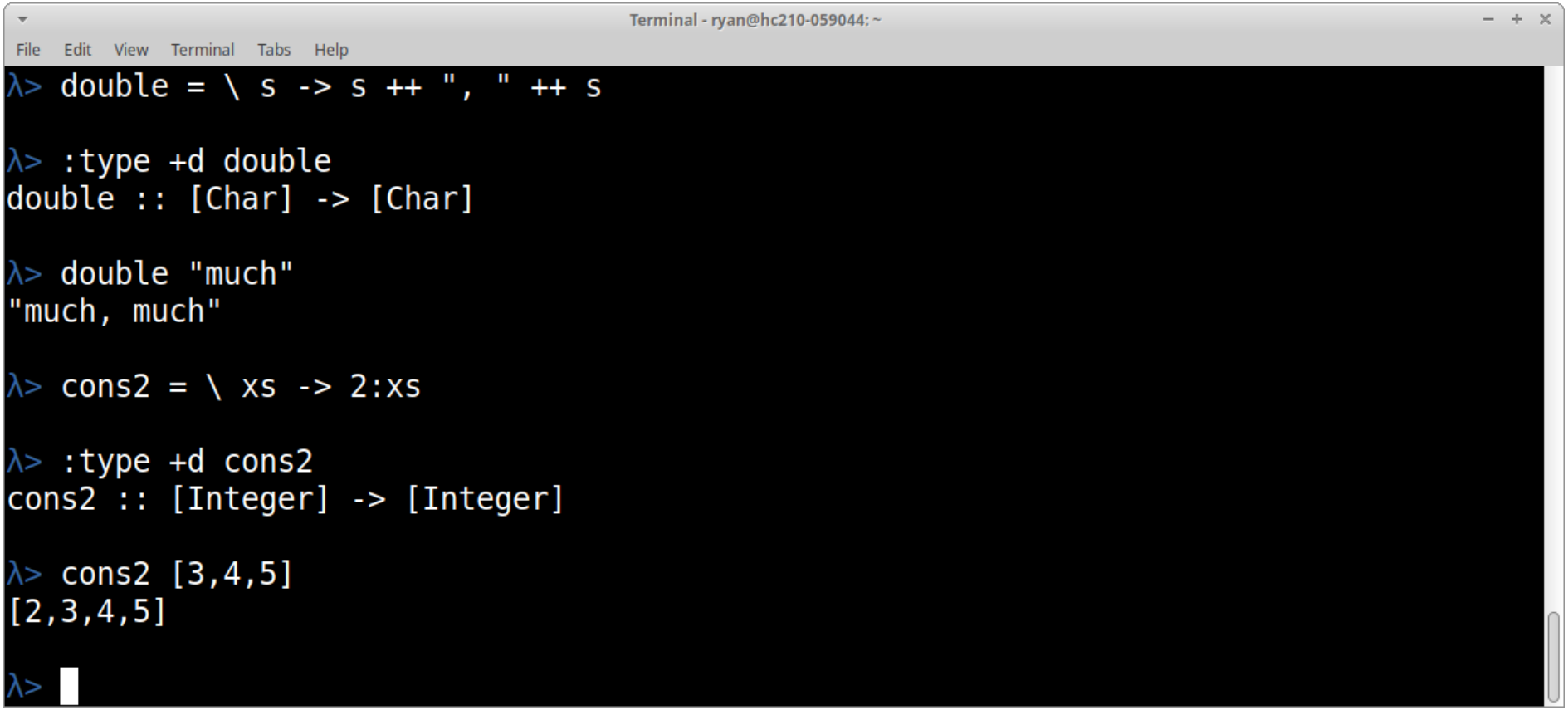
Naming Functions



Naming Functions



Naming Functions



Function Declaration Has Special Syntax

```
add1 = \ x -> x + 1
```

```
add1 x = x + 1
```

```
cutOff = \ i -> max i 0
```

```
cutOff i = max i 0
```

```
g = \ x -> sin (x+pi)
```

```
g x = sin (x+pi)
```

```
f = \ c -> ord c
```

```
f c = ord c
```

```
double=\s->s++", "++ s
```

```
double s = s++", "++s
```

```
cons2 = \ xs -> 2:xs
```

```
cons2 xs = 2:xs
```

Functions

```
f x = if null x then "Empty!" else "Not Empty!"

factorial n = if n<2 then 1 else n * factorial (n-1)
```

Patterns

Think of a formal argmument as a name that matches any value in the domain of the function.

Patterns as formal arguments use constructors to match against the value given to the function as the actual argument.

If the value does not match, you are out of luck.

Patterns

```
p3 (x,y,z) = z   -- definition of function p3

p3 (1,2,3) -- evaluates to 3

tr = (1,2,3)

p3 tr   -- evaluates to 3

f (x:2:y:rest) = x+y -- defintion of f

f (1:2:3:9:[]) -- evaluates to 4

f [1,2,3,9]      -- evaluates to 4

l = [12,3,9]

f l -- evaluates to 4
```

Wildcard Patterns

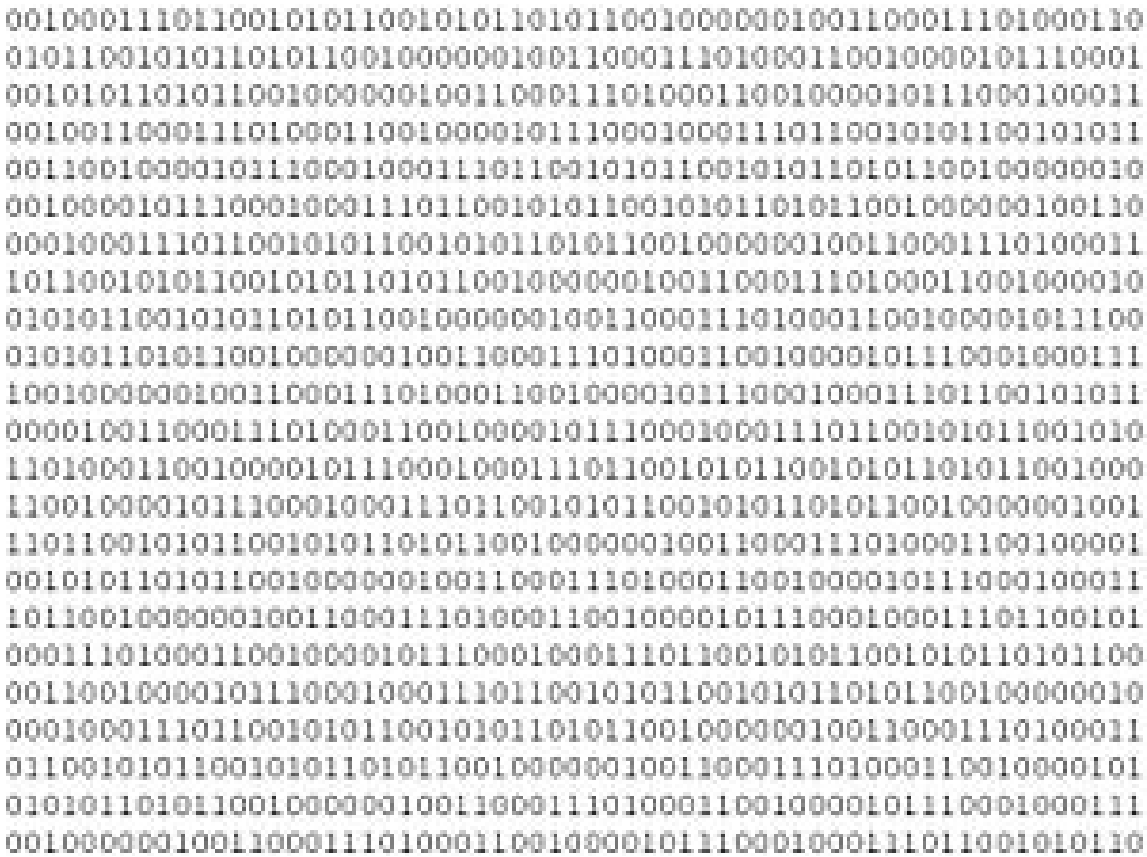
```
p3 (_,_,z) = z   -- definition of function p3

f (_:2:_:_) = x+y -- definition of f
```

Definition by cases

```
f  []           = "empty"
f  (x:[])       = "single"
f  (x:y:[])     = "small"
f  (x:y:z:[])   = "medium"
f  (_)          = "large"
```

Overview

- Everything is digital (all files are binary)
- Unix streams and pipes (function composition is important)
- What is a program?
- GHC
  - writing a program
  - compiling a program (also `-Wall`)
  - running a program (also `+RTS -RTS`)
- `interact`: boilerplate to turn a function into a working program



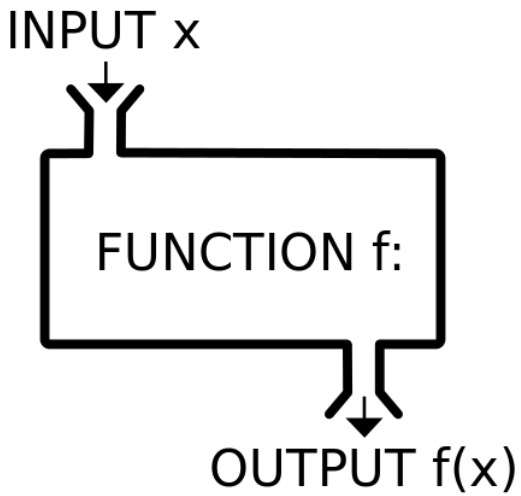
Everything is Digital

Streams and Pipes

Along the Stream by Sharon France

We know what a function is: it maps an element in the domain to an element in the range.

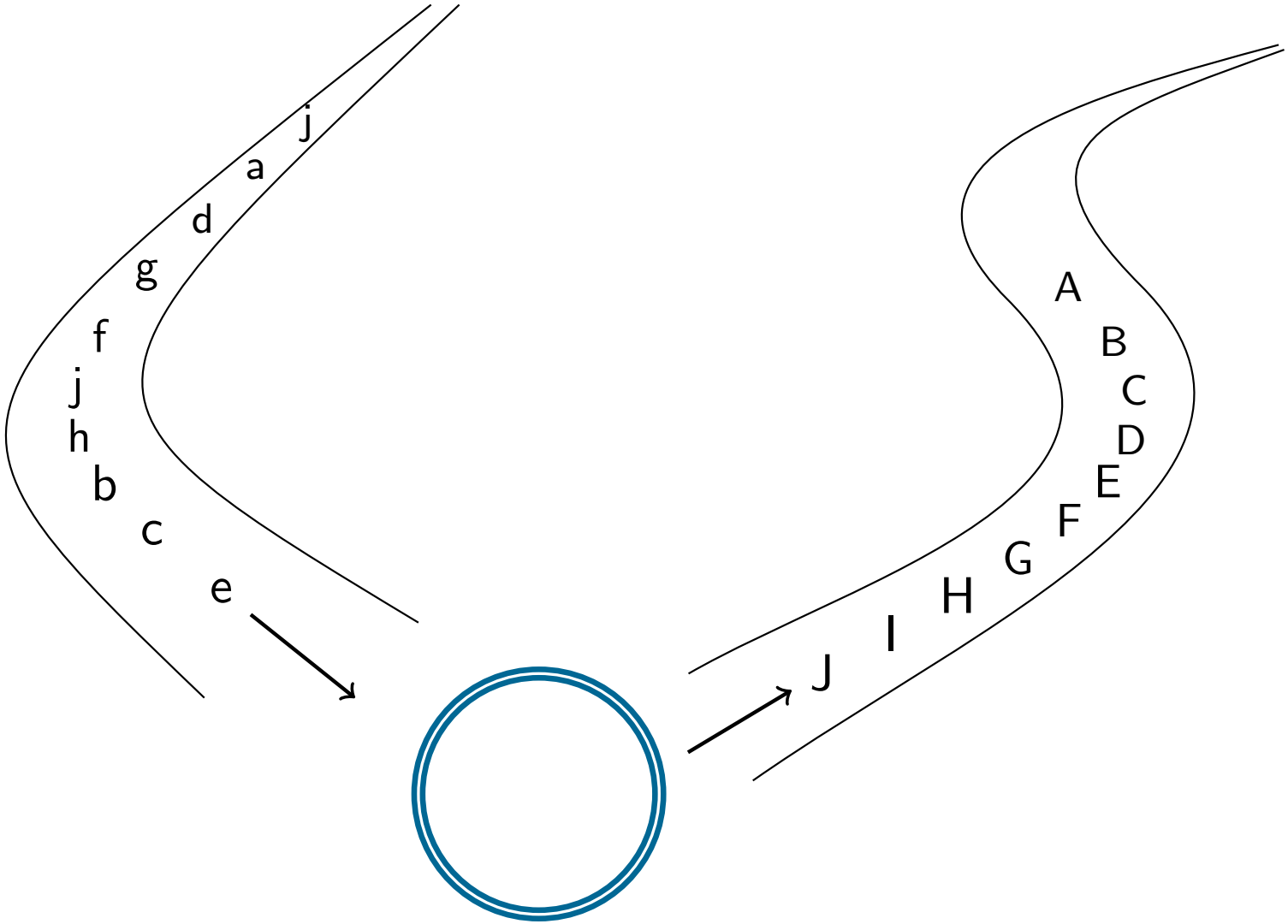
When we think of a program as a function we generally think of something like the factorial function from numbers to numbers.



But this is not a "real" program. A real program reads input and writes output.

Limiting ourselves to the important case of programs from US-ASCII text to US-ASCII text, a real program is really a *function* from a stream/file/string of text to a stream/file/string of text.

Input and Output Stream



Using the Glasgow Haskell Compiler

In Haskell, just like numerous other programming languages, one creates a source program in a text file, one invokes the compiler to create an executable file, and then one commands the computer to run the executable file on some text input and observe or collect the text output.

```
$ ghc -Wall -o main Main.hs
$ main < input.txt > output.txt
```

Warning!

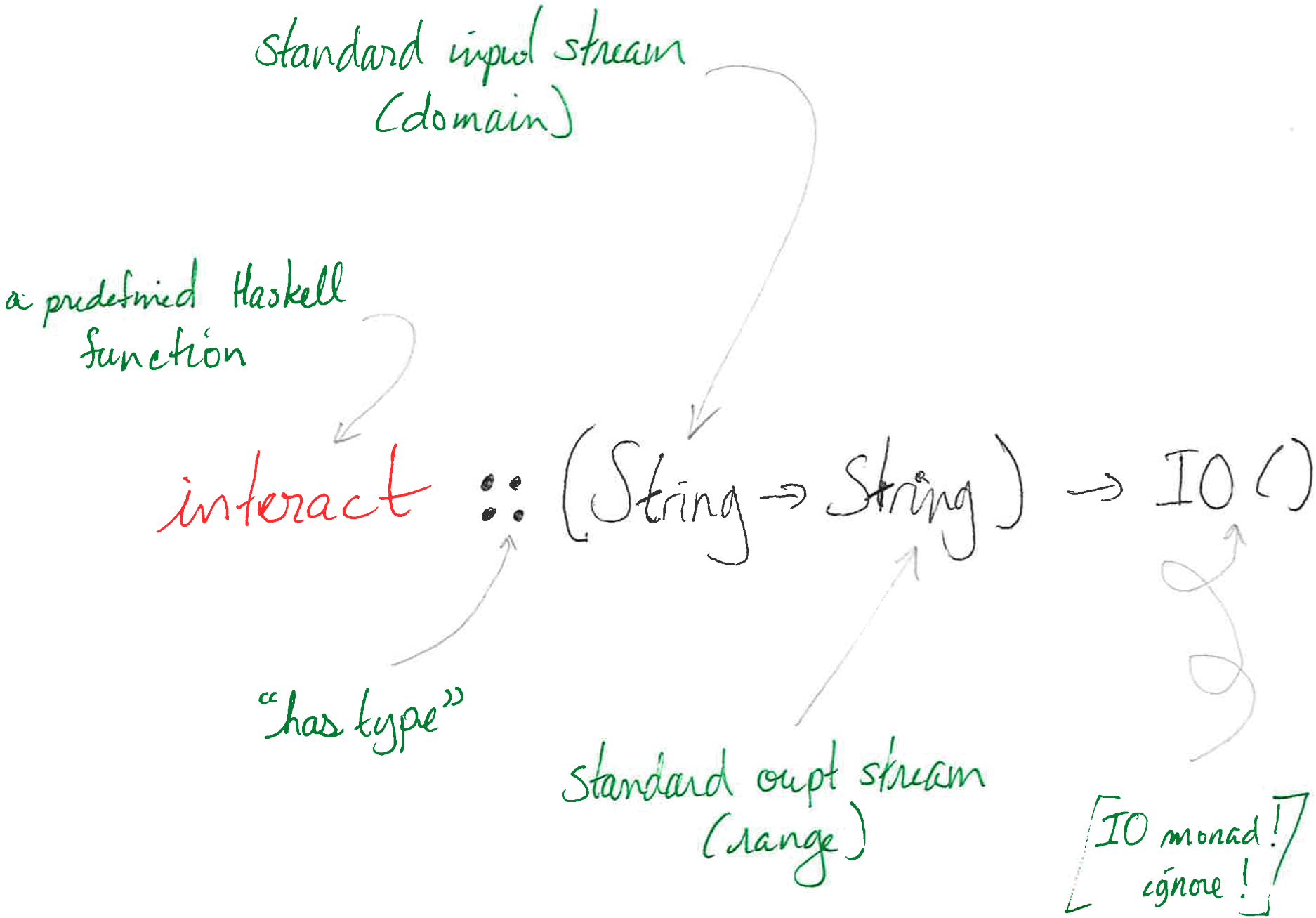
Ask the compiler to help you write a more beautiful program by turning on all warnnigs.

```
$ ghc -Wall -o main Main.hs
```

Don't turn in a program with warnings.

How do do we create a Haskell program that reads the input and writes the output?

Some programming languages have elaborate IO mechanisms which one must learn to simple process the the input and produce the output.
In Haskell the simplest approach is to use the function `interact` to transform the conceptually pure program (a function from the input stream to the output stream) to an actual, working program on the computer.


standard input stream
(domain)
a predefined Haskell
function
interact :: (String → String) → IO ()
"has type"
standard output stream
(range)
IO monad!
ignore!

Examples On-Line

README