Haskell Basics

Max Levatich & Stephen A. Edwards

Columbia University

Fall 2025

Arithmetic and Booleans

Lists

The sum and length Functions

Fold and Map

List Ranges and Comprehensions

Infinite Lists

Tuples

Primes: The Sieve of Eratosthenes

Useful Websites

- `https://www.haskell.org/`

  Downloads, documentation

  E.g., the Haskell Wiki, the GHC User's Guide, The Haskell 2010 language report, Hackage (package library), Hoogle (Haskell API search)

- `http://docs.haskellstack.org`

  The Haskell Tool Stack: a powerful system for downloading and installing packages, etc.

  We will be using the Haskell Stack to make sure everybody's environment is consistent.

GHCi

GHC is the Glasgow Haskell Compiler (the major Haskell compiler release)

GHCi is the REPL (Read-Eval-Print Loop, a.k.a., command-line interface)

Run `ghci` with stack:

```
$ stack config set resolver lts-22.33
$ stack ghci
Configuring GHCi with the following packages:
GHCi, version 9.6.6: https://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /home/sedwards/...
ghci> :?
 Commands available from the prompt:

   <statement>                 evaluate/run <statement>
   :quit                       exit GHCi
```

Comments

Single-line comments start with two dashes: `--`

```
ghci> -- Single-line comment
```

Multi-line comments start with `{-`, end with `-}`, and may nest.

In GHCi only, multi-line definitions, etc. may be written with `:{` and `:}`; these are unnecessary in source (.hs) files.

```
ghci> :{
ghci| {- This is a
ghci|    multi-line comment -}
ghci| :}
```

Alternately enable multi-line input mode in GHCi:

```
ghci> :set +m
ghci> {-
ghci|     A multi-line
ghci|     Comment
ghci| -}
ghci> {- Another
ghci|     one -}
```

Basic Arithmetic

```
ghci> 2 + 15
17
ghci> 42 - 10
32
ghci> 1 + 2 * 3
7
ghci> 5 / 2
2.5
ghci> 3 + -2

<interactive>:4:1: error:
  Precedence parsing error
   cannot mix '+' [infixl 6] and prefix '-' [infixl 6] in the same
   infix expression
ghci> 3 + (-2)
1
```

Booleans and Equality

Haskell is case-sensitive

```
ghci> True && False
False
ghci> False || True
True
ghci> not True || True
True
ghci> not (True || True)
False
```

```
ghci> 5 == 5
True
ghci> 5 == 0
False
ghci> 5 /= 5
False
ghci> 5 /= 0
True
ghci> "hello" == "hello"
True
```

```
ghci> "llama" == 5
<interactive>:25:12: error:
    * No instance for (Num [Char]) arising from the literal '5'
    * In the second argument of '(==)', namely '5'
      In the expression: "llama" == 5
      In an equation for 'it': it = "llama" == 5
```

Function Application

Juxtaposition indicates function application. Don’t use parentheses or commas for arguments

```
ghci> succ 41
42
ghci> min 42 17
17
ghci> 42 `max` 17    -- `max` is infix operator of the function
42
```

Juxtaposition binds tightly; use parentheses to group arguments

```
ghci> succ 3 * 2
8
ghci> succ (3 * 2)
7
ghci> (*) 3 (succ 4)   -- (*) is prefix function of the operator
15
```

Lists: Homogeneous Sequences (Singly-linked Lists)

`[]` is the empty list

```
ghci> []
[]
```

The “Cons” operator `:` adds an element to the front of a list.
The `[ , ]` notation is syntactic sugar for multiple `:` operations

```
ghci> 42 : []
[42]
ghci> 1 : (2 : [])    -- The right argument must be a list
[1,2]
ghci> 2 : 3 : 4 : []  -- Associates right-to-left
[2,3,4]
ghci> [2,3,4]         -- Abbreviation
[2,3,4]
```

Strings: Lists of characters

```
ghci> 'H' : 'e' : 'l' : 'l' : 'o' : []   -- Character literals
"Hello"
ghci> ['H','e','l','l','o']
"Hello"
ghci> 'H' : "ello"
"Hello"
```

Cons cannot join lists because the first argument must be an element

```
ghci> "He" : "llo"
<interactive>:43:8: error: [GHC-83865]
    * Couldn't match type 'Char' with '[Char]'
      Expected: [String]
        Actual: String     -- Assumed the first argument was correct
    * In the second argument of '(:)', namely '"llo"'
      In the expression: "He" : "llo"
      In an equation for 'it': it = "He" : "llo"
```

Functions on Lists

Function that puts “re” in front of a string:

```
ghci> re s = 'r' : 'e' : s  -- User-defined function w/ argument s
ghci> re "animator"
"reanimator"
ghci> re "ject"
"reject"

ghci> :type re              -- What is the type of this function?
re :: [Char] -> [Char]      -- Haskell inferred the type
```

“re has the type ’a function from a list of characters to a list of characters’“

```
ghci> :t re                  -- Abbreviation
re :: [Char] -> [Char]
ghci> :t re "ject"           -- Type of an expression
re "ject" :: [Char]
```

Challenge: Sum the Elements of a List

```
ghci> sum [40, 2, 8000]
8042
```

Challenge: Sum the Elements of a List

```
ghci> sum [40, 2, 8000]
8042
```

```
sum l =
```

User-defined function called `sum` with a single argument `l`

Let's try a recursive solution

Challenge: Sum the Elements of a List

```
ghci> sum [40, 2, 8000]
8042
```

```
sum l = case l of
```

Pattern-match on the list. Sort of like “switch-case” in C/Java.

Challenge: Sum the Elements of a List

```
ghci> sum [40, 2, 8000]
8042
```

```
sum l = case l of
        []     -> 0
```

The base case: when `l` is the empty list, return 0

The **`of`** keyword starts a multi-line block

The next token (here, `[]`) sets the indentation of the block

Challenge: Sum the Elements of a List

```
ghci> sum [40, 2, 8000]
8042
```

```
sum l = case l of
        []     -> 0
        x : xs ->
```

The recursive case: when `l` is a non-empty list whose first element (its "head") is `x` and everything else (the "tail") is `xs`

This is the Cons operator `:` being used as a pattern:
"If this is a list that had been constructed with `x : xs`"

Challenge: Sum the Elements of a List

```
ghci> sum [40, 2, 8000]
8042
```

```
sum l = case l of
        []     -> 0
        x : xs -> x + sum xs
```

Return the head of the list plus the sum of the rest of the list

Challenge: Sum the Elements of a List

```
ghci> sum [40, 2, 8000]
8042
```

```
sum []       = 0
sum (x : xs) = x + sum xs
```

More concise: function arguments may be patterns

Challenge: Sum the Elements of a List

```
ghci> sum [40, 2, 8000]
8042
```

```
ghci> :{
ghci| sum []       = 0
ghci| sum (x : xs) = x + sum xs
ghci| :}
ghci> sum [5,10,300]
315
ghci> sum [1,2,3]
6
ghci> sum [1,2,3,4,5,6,7,8,9,10]
55
ghci> sum [10,9,8,7,6,5,4,3,2,1]
55
```

The `:{ :}` notation tells GHCi that multiple lines are coming

Haskell Layout Syntax

Internally, the Haskell compiler intreprets

```
sum l = case l of
        []     -> 0
        x : xs -> x + sum xs
```

as

```
sum l = case l of { [] -> 0 ; x : xs -> x + sum xs }
```

The only effect of layout is to insert `{ ; }` tokens.

Manually inserting `{ ; }` overrides the layout rules

Haskell Layout Syntax

- Layout blocks begin after *let*, *where*, *do*, and *of* unless there's a `{`
- The first token after the keyword sets the indentation of the block
- Every following line at that indentation gets a leading `;`
- Every line indented more is part of the previous line
- The block ends (an implicit `}`) when anything is indented less

```
sum l =
 case l of
 []     -> 0
 x : xs -> x + sum xs
```

```
sum l = case l of
 []     -> 0
 x : xs -> x
       + sum xs
```

```
sum l = case l of []     -> 0
                  x : xs -> x + sum xs
```

```
sum l = case l of []     -> 0
                  x : xs -> x
                  + sum xs      -- No
```

```
sum l = case l of []     -> 0
                x : xs -> x + sum xs -- No
```

The Type of Sum: Type variables and Typeclasses

```
sum []       = 0
sum (x : xs) = x + sum xs
```

```
ghci> :t sum
sum :: Num a => [a] -> a
```

`sum` is polymorphic: the type of `sum` includes type variable `a`

`Num a =>` means “type `a` must be of the `Num` typeclass”: a number

“sum is a function that, provided `a`’s are number-like, takes a list of `a`’s and returns an `a`”

```
ghci> sum [1.1, 20.02, 300.003]  -- Works on doubles, too
321.123
```

Challenge: The Length of a List

```
ghci> length [1,3,5,7,9]
5
```

```
length []       = 0
length (x : xs) = 1 + length xs
```

```
ghci> :{
ghci| length [] = 0
ghci| length (x : xs) = 1 + length xs
ghci| :}
ghci> :t length
length :: Num a1 => [a2] -> a1
```

`length` is polymorphic: the result (type `a1`) must be a `Num`, but the list itself may be of any type (`a2`)

Multi-argument functions are Curried

Haskell functions have exactly one argument. Functions with “multiple arguments” are actually functions that return functions that return functions.

Such “currying” is named after Haskell Brooks Curry, who is also known for the Curry-Howard Correspondence (“programs are proofs”).

```
ghci> add x y = x + y
ghci> add 17 25
42
ghci> :t add
add :: Num a => a -> a -> a
ghci> :t (+)
(+) :: Num a => a -> a -> a
```

```
ghci> addtwo = add 2
ghci> :t addtwo
addtwo :: Num a => a -> a
ghci> addtwo 40
42
ghci> :t (+) 5
(+) 5 :: Num a => a -> a
ghci> ((+) 5) 30
35
```

Curried Functions: Partial Application

```
ghci> ise a b = a : b : "ise"
ghci> :t ise
ise :: Char -> Char -> [Char]
ghci> ise 'a' 'r'
"arise"
ghci> ise 'n' 'o'
"noise"
ghci> aise = ise 'a'    -- Give it the first argument
ghci> :t aise
aise :: Char -> [Char]
ghci> aise 'n'
"anise"
```

-> is right-to-left: `Char -> Char -> [Char] = Char -> (Char -> [Char])`

“A function that takes a Char and returns
a function that takes a Char and returns a list of Chars”

Now Just Fold Everything

"Walk through the elements of a list accumulating a result" appears frequently; let's make a function for it

```
sum []       = 0
sum (x : xs) = x + sum xs

length []       = 0
length (x : xs) = 1 + length xs
```

```
sum []       = 0
sum (x : xs) = x + sum xs
```

Let's generalize this into something that can implement sum, length, etc.

Now Just Fold Everything

"Walk through the elements of a list accumulating a result" appears frequently; let's make a function for it

```
sum []       = 0
sum (x : xs) = x + sum xs

length []       = 0
length (x : xs) = 1 + length xs
```

```
foldr []       = 0
foldr (x : xs) = x + foldr xs
```

In Haskell, this is called "foldr" because it works left-to-right

Now Just Fold Everything

"Walk through the elements of a list accumulating a result" appears frequently; let's make a function for it

```
sum []       = 0
sum (x : xs) = x + sum xs

length []       = 0
length (x : xs) = 1 + length xs
```

```
foldr z []       = z
foldr z (x : xs) = x + foldr z xs
```

Let's call the base case result `z`

Now Just Fold Everything

"Walk through the elements of a list accumulating a result" appears frequently; let's make a function for it

```
sum []       = 0
sum (x : xs) = x + sum xs

length []       = 0
length (x : xs) = 1 + length xs
```

```
foldr _ z []       = z                    -- The _ argument is ignored
foldr f z (x : xs) = x `f` foldr f z xs
```

Let's call the accumulation function `f`

```
foldr f z [x1, x2, ..., xn] = x1 `f` (x2 `f` ( ... (xn `f` z) ...))
```

Now Just Fold Everything

```
ghci> :{
ghci| foldr _ z [] = z
ghci| foldr f z (x : xs) = x `f` foldr f z xs
ghci| :}
ghci> :t foldr
foldr :: (t1 -> t2 -> t2) -> t2 -> [t1] -> t2
ghci> foldr (+) 0 [1,2,3,4,5,6,7,8,9,10] -- sum
55
ghci> add1 _ x = succ x                   -- succ 8 = 9
ghci> foldr add1 0 [1,1,1,1]              -- length
4
ghci> foldr (\_ x -> succ x) 0 [1,1,1,1] -- A lambda expression
4
ghci> foldr (:) "World" "Hello "          -- List concatenation
"Hello World"
ghci> "Hello " ++ "World"                 -- ++ is list concat
"Hello World"
```

Map: A particularly useful fold

Folding and accumulating a list lets you modify list elements

```
ghci> foldr (\x xs -> x + 5 : xs) [] [1,2,3,5,7]
[6,7,8,10,12]
```

This is called `map`

```
ghci> map f = foldr (\x xs -> f x : xs) []  -- Apply f and prepend
ghci> :t map
map :: (t -> a) -> [t] -> [a]
ghci> map (+10) [0,1,2,3,4,5]   -- (+10) is the "add ten" function
[10,11,12,13,14,15]
```

Another way to code it:

```
map _ []     = []
map f (x:xs) = f x : map f xs
```

The list toolbox

The basics:

- Empty list ([])
- Cons (:) and pattern matching (x:xs)
- Recursion!

Basic transformations:

- map, fold, filter, zip, cartesian product

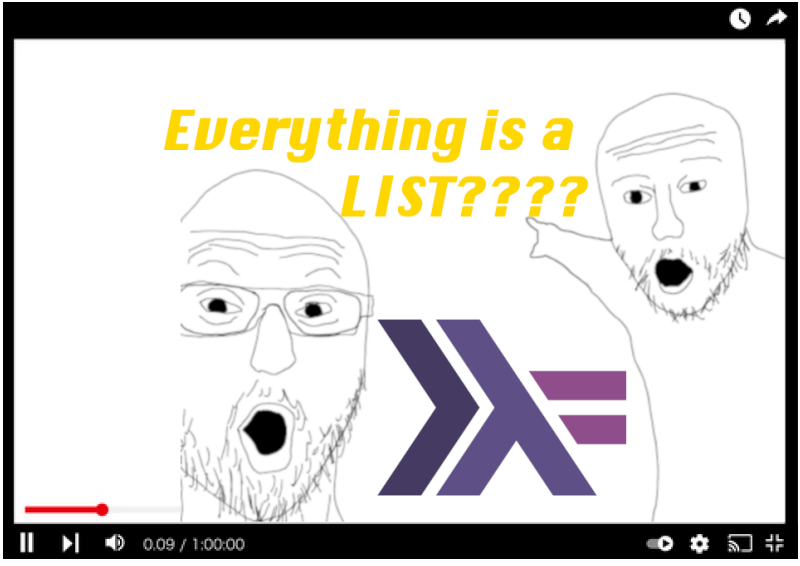
And some nice utilities built from these:

- take, drop, length, sum, head, tail, elem, repeat, (++)...

Clickbait

In Haskell, everything is a list.

Clickbait



Slightly less clickbait

In Haskell, everything is *like* a list.

Honest

In Haskell, we operate on most data the same way we operate on lists.

Haskell style

Do's:

- Cons (:) and pattern matching, case expressions, guards for branching
- lambdas and partial application for brevity
- If the same operation appears multiple times, factor it out into a function with let/where
- List comprehensions
- Advice: https://stackoverflow.com/questions/9345589/guards-vs-if-then-else-vs-cases-in-haskell

Dont's:

- Long lines with lots of parentheses
- "unsafe" operations: head, tail, init, last, !!
- If-then-else instead of cases or patterns, (++) where (:) would suffice
- Manually write a recursive form when a map/filter/fold would suffice

List Ranges

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

Linear sequences only

Floating point numbers don’t work well

List Comprehensions

[*expression* | *generator-guard-let*, *generator-guard-let*, ...]

```
ghci> [ x^2 | x <- [1..19] ]
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,324,361]

ghci> [ x^2 | x <- [1..20], (x^2) `mod` 2 == 0 ]
[4,16,36,64,100,144,196,256,324,400]

ghci> [ x^2 | x <- [1..20], even (x^2) ]
[4,16,36,64,100,144,196,256,324,400]

ghci> [ y | x <- [1..20], let y = x^2, even y ]
[4,16,36,64,100,144,196,256,324,400]
```

`let` introduces a new name whose scope ends at the `]`. This is the only place where a *let* does not have a matching *in*.

List Comprehensions

Multiple guards must all be true

```
ghci> [ x | x <- [1..100], x `mod` 7 == 0 ]
[7,14,21,28,35,42,49,56,63,70,77,84,91,98]

ghci> [ x | x <- [1..100], x `mod` 7 == 0, x `mod` 5 == 0 ]
[35,70]
```

Multiple generators apply right-to-left:

```
ghci> [ x + y | x <- [100,200..400], y <- [0..3] ]
[100,101,102,103,200,201,202,203,300,301,302,303,400,401,402,403]
```

Application: CS Research Jargon Generator

```
ghci> :set +m
ghci> [ adjective ++ " " ++ noun |
ghci|   adjective <- ["An integrated","A type-safe"],
ghci|   noun <- ["network","architecture","hypervisor"] ]
["An integrated network","An integrated architecture",
 "An integrated hypervisor","A type-safe network",
 "A type-safe architecture","A type-safe hypervisor"]
```

https://www.cs.purdue.edu/homes/dec/essay.topic.generator.html

Application: CS Research Jargon Generator

```
ghci> :set +m
ghci> [ adjective ++ " " ++ noun |
ghci|   adjective <- ["Deep","Practical","Large Language",
ghci|                "Generative"],
ghci|   noun <- ["Learning","AI"] ]
["Deep Learning","Deep AI","Practical Learning","Practical AI",
 "Large Language Learning", "Large Language AI",
 "Generative Learning","Generative AI"]
```

https://www.cs.purdue.edu/homes/dec/essay.topic.generator.html

List Comprehensions

An alternative way to code the list length function:

```
ghci> length xs = sum [ 1 | _ <- xs ]
ghci> length [5,6,2,1,0]
5
ghci> length (replicate 11 []) -- List of eleven empty lists
11
```

Names (variable identifiers) start with a lowercase letter followed by zero or more letters, digits, underscores, and single quotes.

```
ghci> onlyLetters s = [ c | c <- s,
ghci|                       c `elem` ['A'..'Z'] ++ ['a'..'z'] ]
ghci> onlyLetters "Does this do what I think it 5hould?"
"DoesthisdowhatIthinkithould"
```

Let Expressions: Introducing Local Names

let <bindings> **in** <expression>

```
cylinder r h = let sideArea = 2 * pi * r * h
                   topArea = pi * r^2
               in sideArea + 2 * topArea
```

This example can be written “more mathematically” with **where**

```
cylinder r h = sideArea + 2 * topArea
  where sideArea = 2 * pi * r * h
        topArea = pi * r^2
```

Semantically equivalent; *let*...*in* is an expression; *where* only comes after bindings. Only *where* works across guards.

let...*in* Is an Expression and More Local

A contrived example:

```
f a = a + let a = 3 in a
```

This is the “add 3” function. The scope of `a = 3` is limited to the *let*...*in*

let bindings are recursive. E.g.,

```
ghci> let a = '?' : a in a
"????????????????????????????????????????????????????????????????????????
????????????????????????????????????????????????????????????????????????
^C
```

is an infinite list of question mark characters.

The take function: Return the first n elements of a list

```
take 0 _         = []
take _ []        = []
take n (x : xs) = x : take (n-1) xs
```

```
ghci> take 5 [1,2,3]
[1,2,3]
ghci> take 5 [1..10]
[1,2,3,4,5]
ghci> take 0 [1..10]
[]
ghci> take 5 [1..3]
[1,2,3]

ghci> let a = '?' : a in take 5 a   -- Haskell is lazy
"?????"
```

Infinite Lists

Haskell supports infinite lists (and other infinite data structures).
Hint: don't print out the whole thing. E.g., use `take` to see the first elements

```
ghci> take 5 [1..]
[1,2,3,4,5]
ghci> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
ghci> take 10 [1,2,3]
[1,2,3]
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
ghci> take 16 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1,2,3,1,2,3,1]
ghci> take 17 (repeat 5)
[5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5]
ghci> replicate 15 6
[6,6,6,6,6,6,6,6,6,6,6,6,6,6,6]
```

Tuples: Pairs and More of Heterogeneous Objects

Lists are zero or more things of the same type; a tuple is two or more of (potentially) different types.

```
ghci> (5,10)
(5,10)
ghci> ("a",15)
("a",15)
ghci> ("Douglas","Adams",42)
("Douglas","Adams",42)
ghci> sae = ("Stephen", "Edwards")
ghci> fst sae
"Stephen"
ghci> snd sae
"Edwards"
```

Zip and Pythagorean Triples

Form a list of pairs from two lists. Shorter of the two lists dominates; convenient with infinite lists

```
ghci> zip [1,2,3] [100,200,300]
[(1,100),(2,200),(3,300)]

ghci> zip "Stephen" [1..]
[('S',1),('t',2),('e',3),('p',4),('h',5),('e',6),('n',7)]
```

```
ghci> [ (a,b,c) | c <- [1..20], b <- [1..c], a <- [1..b],
ghci|             a^2 + b^2 == c^2 ]
[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17),(12,16,20)]
```

The Handshake Problem

Number of handshakes among a group of *n* friends?

```
ghci> handshakes n = [ (a,b) | a <- [1..n-1], b <- [a+1..n] ]
ghci> handshakes 3
[(1,2),(1,3),(2,3)]
ghci> handshakes 5
[(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),(3,4),(3,5),(4,5)]
ghci> length (handshakes 5)
10
ghci> [ length (handshakes n) | n <- [1..10] ]
[0,1,3,6,10,15,21,28,36,45]
ghci> [ n * (n-1) `div` 2 | n <- [1..10] ]
[0,1,3,6,10,15,21,28,36,45]
```

The Sieve of Eratosthenes

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

`[2..]`	The infinite list [2,3,4,...]
`where` `filterPrime`	*Where* clause defining *filterPrime*
`(p:xs)`	Pattern matching on head and tail of list
`p : filterPrime` ...	Recursive function application
`[x | x <- xs, x ‘mod‘ p /= 0]`	List comprehension: everything in xs not divisible by `p`

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

```
primes = fp [2..] where
                  fp (p:ps) = p : fp [x | x <- ps, x `mod` p /= 0]
       = fp (2 : [3..])
```

```
primes = fp [2..] where
                  fp (p:ps) = p : fp [x | x <- ps, x `mod` p /= 0]
       = fp (2 : [3..])
       = 2 : fp [ x | x <- [3..], x `mod` 2 /= 0]
```

```
primes = fp [2..] where
                  fp (p:ps) = p : fp [x | x <- ps, x `mod` p /= 0]
       = fp (2 : [3..])
       = 2 : fp [ x | x <- [3..], x `mod` 2 /= 0]
       = 2 : fp [3,5,7,9..]
```

```
primes = fp [2..] where
                  fp (p:ps) = p : fp [x | x <- ps, x `mod` p /= 0]
       = fp (2 : [3..])
       = 2 : fp [ x | x <- [3..], x `mod` 2 /= 0]
       = 2 : fp [3,5,7,9..]
       = 2 : fp (3 : [ x | x <- [5..], x `mod` 2 /= 0])
```

```
primes = fp [2..] where
                  fp (p:ps) = p : fp [x | x <- ps, x `mod` p /= 0]
       = fp (2 : [3..])
       = 2 : fp [ x | x <- [3..], x `mod` 2 /= 0]
       = 2 : fp [3,5,7,9..]
       = 2 : fp (3 : [ x | x <- [5..], x `mod` 2 /= 0])
       = 2 : 3 : fp [ x | x <- [5..], x `mod` 2 /= 0,
                                      x `mod` 3 /= 0]
```

```
primes = fp [2..] where
                  fp (p:ps) = p : fp [x | x <- ps, x `mod` p /= 0]
       = fp (2 : [3..])
       = 2 : fp [ x | x <- [3..], x `mod` 2 /= 0]
       = 2 : fp [3,5,7,9..]
       = 2 : fp (3 : [ x | x <- [5..], x `mod` 2 /= 0])
       = 2 : 3 : fp [ x | x <- [5..], x `mod` 2 /= 0,
                                      x `mod` 3 /= 0]
       = 2 : 3 : fp [5,7, 11,13, 17,19, 23,25, 29,31, 35,37, 41,
```

```
primes = fp [2..] where
                  fp (p:ps) = p : fp [x | x <- ps, x `mod` p /= 0]
       = fp (2 : [3..])
       = 2 : fp [ x | x <- [3..], x `mod` 2 /= 0]
       = 2 : fp [3,5,7,9..]
       = 2 : fp (3 : [ x | x <- [5..], x `mod` 2 /= 0])
       = 2 : 3 : fp [ x | x <- [5..], x `mod` 2 /= 0,
                                      x `mod` 3 /= 0]
       = 2 : 3 : fp [5,7, 11,13, 17,19, 23,25, 29,31, 35,37, 41,
       = 2 : 3 : fp (5 : [ x | x <- [7..], x `mod` 2 /= 0,
                                           x `mod` 3 /= 0])
```

```
primes = fp [2..] where
                  fp (p:ps) = p : fp [x | x <- ps, x `mod` p /= 0]
       = fp (2 : [3..])
       = 2 : fp [ x | x <- [3..], x `mod` 2 /= 0]
       = 2 : fp [3,5,7,9..]
       = 2 : fp (3 : [ x | x <- [5..], x `mod` 2 /= 0])
       = 2 : 3 : fp [ x | x <- [5..], x `mod` 2 /= 0,
                                      x `mod` 3 /= 0]
       = 2 : 3 : fp [5,7, 11,13, 17,19, 23,25, 29,31, 35,37, 41,
       = 2 : 3 : fp (5 : [ x | x <- [7..], x `mod` 2 /= 0,
                                         x `mod` 3 /= 0])
       = 2 : 3 : 5 : fp [ x | x <- [7..], x `mod` 2 /= 0,
                                        x `mod` 3 /= 0,
                                        x `mod` 5 /= 0])
```

```
primes = fp [2..] where
                  fp (p:ps) = p : fp [x | x <- ps, x `mod` p /= 0]
       = fp (2 : [3..])
       = 2 : fp [ x | x <- [3..], x `mod` 2 /= 0]
       = 2 : fp [3,5,7,9..]
       = 2 : fp (3 : [ x | x <- [5..], x `mod` 2 /= 0])
       = 2 : 3 : fp [ x | x <- [5..], x `mod` 2 /= 0,
                                      x `mod` 3 /= 0]
       = 2 : 3 : fp [5,7, 11,13, 17,19, 23,25, 29,31, 35,37, 41,
       = 2 : 3 : fp (5 : [ x | x <- [7..], x `mod` 2 /= 0,
                                         x `mod` 3 /= 0])
       = 2 : 3 : 5 : fp [ x | x <- [7..], x `mod` 2 /= 0,
                                        x `mod` 3 /= 0,
                                        x `mod` 5 /= 0])
       = 2 : 3 : 5 : fp [7,11,13,17,19,23, 29,31, 37,41,43,47,49,
```

```
primes = fp [2..] where
                  fp (p:ps) = p : fp [x | x <- ps, x `mod` p /= 0]
       = fp (2 : [3..])
       = 2 : fp [ x | x <- [3..], x `mod` 2 /= 0]
       = 2 : fp [3,5,7,9..]
       = 2 : fp (3 : [ x | x <- [5..], x `mod` 2 /= 0])
       = 2 : 3 : fp [ x | x <- [5..], x `mod` 2 /= 0,
                                      x `mod` 3 /= 0]
       = 2 : 3 : fp [5,7, 11,13, 17,19, 23,25, 29,31, 35,37, 41,
       = 2 : 3 : fp (5 : [ x | x <- [7..], x `mod` 2 /= 0,
                                         x `mod` 3 /= 0])
       = 2 : 3 : 5 : fp [ x | x <- [7..], x `mod` 2 /= 0,
                                        x `mod` 3 /= 0,
                                        x `mod` 5 /= 0])
       = 2 : 3 : 5 : fp [7,11,13,17,19,23, 29,31, 37,41,43,47,49,
       = 2 : 3 : 5 : fp (7 : [ x | x <- [11..], x `mod` 2 /= 0,
                                              x `mod` 3 /= 0,
                                              x `mod` 5 /= 0])
```