Functional Programming with Haskell, Part 2

Kenneth Lai

Brandeis University

September 19, 2022

Announcements

- Student hours this week
  - Tue 9/20 5-6pm, Ken, remote only
  - Wed 9/21 11am-noon, Bingyang, hybrid
  - Thu 9/22 4-5pm, Ken, remote only
  - Fri 9/23 2:15-3:15pm, Bingyang, hybrid
- For Wednesday:
  - Review van Eijck and Unger Chapter 4.4, 5.2, and 5.3
  - Read van Eijck and Unger Chapter 4.5, 4.6, and 5.5
- For 9/28:
  - HW1 due

Today's Plan

- Functional Programming with Haskell

Prefix and Infix Operators

- Most Haskell operators (functions) are prefix operators
  - Write the function first, followed by its arguments

Prefix and Infix Operators

- Most Haskell operators (functions) are prefix operators
  - Write the function first, followed by its arguments
- Some binary operators are infix operators
  - Write the function between its arguments
  - `1 + 1` instead of `+ 1 1`

Prefix and Infix Operators

- Most Haskell operators (functions) are prefix operators
  - Write the function first, followed by its arguments
- Some binary operators are infix operators
  - Write the function between its arguments
  - `1 + 1` instead of `+ 1 1`
- Parentheses change an infix operator into a prefix operator
  - `1 + 1` or `(+) 1 1`

Prefix and Infix Operators

- Most Haskell operators (functions) are prefix operators
  - Write the function first, followed by its arguments
- Some binary operators are infix operators
  - Write the function between its arguments
  - `1 + 1` instead of `+ 1 1`
- Parentheses change an infix operator into a prefix operator
  - `1 + 1` or `(+) 1 1`
- Backticks change a prefix operator into an infix operator
  - `elem 1 [1,2]` or `1 ‘elem‘ [1,2]`

Currying

- Currying is the conversion of a function of multiple arguments into a sequence of functions of one argument

Currying

- Currying is the conversion of a function of multiple arguments into a sequence of functions of one argument
  - `elem 1 [1,2]`: `elem` takes an element and a list (or list-like object), and outputs whether the element is in the list

Currying

- Currying is the conversion of a function of multiple arguments into a sequence of functions of one argument
  - `elem 1 [1,2]`: `elem` takes an element and a list (or list-like object), and outputs whether the element is in the list
  - `elem 1 [1,2]`: `elem` takes an element (1), and outputs a function `(elem 1)` that takes a list (or list-like object), and outputs whether `1` is in the list

Currying

- Currying is the conversion of a function of multiple arguments into a sequence of functions of one argument
  - `elem 1 [1,2]`: `elem` takes an element and a list (or list-like object), and outputs whether the element is in the list
  - `elem 1 [1,2]`: `elem` takes an element (1), and outputs a function `(elem 1)` that takes a list (or list-like object), and outputs whether `1` is in the list
  - These two descriptions are equivalent

String Functions in Haskell

```
Prelude> (\ x -> x ++ " emeritus") "professor"
"professor emeritus"
```

String Functions in Haskell

```
Prelude> (\ x -> x ++ " emeritus") "professor"
"professor emeritus"
```

This combines **lambda abstraction** and **concatenation**.

String Functions in Haskell

```
Prelude> (\ x -> x ++ " emeritus") "professor"
"professor emeritus"
```

This combines **lambda abstraction** and **concatenation**.

The types:

```
Prelude> :t (\ x -> x ++ " emeritus")
\x -> x ++ " emeritus" :: [Char] -> [Char]
Prelude> :t "professor"
"professor" :: String
Prelude> :t (\ x -> x ++ " emeritus") "professor"
(\x -> x ++ " emeritus") "professor" :: [Char]
```

Concatenation

The type of the concatenation function:

```
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
```

(or `(++) :: forall a. [a] -> [a] -> [a]`?)

Concatenation

The type of the concatenation function:

```
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
```

(or `(++) :: forall a. [a] -> [a] -> [a]`?)

The type (with type variable(s)) indicates that `(++)` not only concatenates strings. It works for lists in general

- This is called type polymorphism

More String Functions in Haskell

```
Prelude> (\ x -> "nice " ++ x) "guy"
"nice guy"
Prelude> (\ f -> \ x -> "very " ++ (f x))
              (\ x -> "nice " ++ x) "guy"
"very nice guy"
```

More String Functions in Haskell

```
Prelude> (\ x -> "nice " ++ x) "guy"
"nice guy"
Prelude> (\ f -> \ x -> "very " ++ (f x))
                (\ x -> "nice " ++ x) "guy"
"very nice guy"
```

The types:

```
Prelude> :t "guy"
"guy" :: [Char]
Prelude> :t (\ x -> "nice " ++ x)
(\ x -> "nice " ++ x) :: [Char] -> [Char]
Prelude> :t (\ f -> \ x -> "very " ++ (f x))
(\ f -> \ x -> "very " ++ (f x))
  :: forall t. (t -> [Char]) -> t -> [Char]
```

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.
- The type `[Char]` is abbreviated as `String`.

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.
- The type `[Char]` is abbreviated as `String`.
- Examples of characters are `'a'`, `'b'` (note the single quotes).

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.
- The type `[Char]` is abbreviated as `String`.
- Examples of characters are `'a'`, `'b'` (note the single quotes).
- Examples of strings are `"Montague"` and `"Chomsky"` (note the double quotes).

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.
- The type `[Char]` is abbreviated as `String`.
- Examples of characters are `'a'`, `'b'` (note the single quotes).
- Examples of strings are `"Montague"` and `"Chomsky"` (note the double quotes).
- In fact, `"Chomsky"` can be seen as an abbreviation of the following character list:

  `['C','h','o','m','s','k','y']`.

Booleans

- "The type `Bool` of Booleans (so-called after George Boole) consists of the two truth-values `True` and `False`."

Booleans

- “The type `Bool` of Booleans (so-called after George Boole) consists of the two truth-values `True` and `False`.”
- Logical operators in Haskell
  - Conjunction is `&&`
  - Disjunction is `||`
  - Negation is `not`

Booleans

- "The type `Bool` of Booleans (so-called after George Boole) consists of the two truth-values `True` and `False`."
- Logical operators in Haskell
  - Conjunction is `&&`
  - Disjunction is `||`
  - Negation is `not`
- Types
  - `(&&) :: Bool -> Bool -> Bool`
  - `(||) :: Bool -> Bool -> Bool`
  - `not :: Bool -> Bool`

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

Properties of Strings

- If strings have type [Char] (or String), properties of strings have type [Char] -> Bool.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

- This definition uses *pattern* matching: (x:xs) is the prototypical non-empty list.

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

- This definition uses *pattern* matching: `(x:xs)` is the prototypical non-empty list.
- The `head` of `(x:xs)` is `x`, the `tail` is `xs`.

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

- This definition uses *pattern* matching: `(x:xs)` is the prototypical non-empty list.
- The `head` of `(x:xs)` is `x`, the `tail` is `xs`.
- The head and tail are glued together by means of the operation `:`, of type `a -> [a] -> [a]`.

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

- This definition uses *pattern* matching: `(x:xs)` is the prototypical non-empty list.
- The `head` of `(x:xs)` is `x`, the `tail` is `xs`.
- The head and tail are glued together by means of the operation `:`, of type `a -> [a] -> [a]`.
- The operation combines an object of type `a` with a list of objects of the same type to a new list of objects, again of the same type.

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.
- What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character `a`, or the tail of the sring is an `aword`.

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.
- What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character `a`, or the tail of the sring is an `aword`.
- The list pattern `[]` matches only the empty list,

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.
- What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character a, or the tail of the sring is an `aword`.
- The list pattern `[]` matches only the empty list,
- the list pattern `[x]` matches any singleton list,

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.
- What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character `a`, or the tail of the sring is an `aword`.
- The list pattern `[]` matches only the empty list,
- the list pattern `[x]` matches any singleton list,
- the list pattern `(x:xs)` matches any non-empty list.

Recursion

- Recursive definitions always have a base case, a case that can be computed without calling the function

Recursion

- Recursive definitions always have a base case, a case that can be computed without calling the function
- **Exercise 3.6** Why is the definition of ‘GNU’ as ‘GNU’s Not Unix’ not a recursive definition?

Sentences can go on . . .

Sentences can go on and on and on and on and on and on and on

```
gen :: Int -> String
gen 0 = "Sentences can go on"
gen n = gen (n-1) ++ " and on"

genS :: Int -> String
genS n = gen n ++ "."
```

Recursion

- But a base case is not always enough...

```
story :: Int -> String
story 0 =
 "Let's cook and eat that final missionary, and off to bed."
story k =
 "The night was pitch dark, mysterious and deep.\n"
 ++ "Ten cannibals were seated around a boiling cauldron.\n"
 ++ "Their leader got up and addressed them like this:\n'"
 ++ story (k-1) ++ "'"
```

Recursion

- But a base case is not always enough...

```
story :: Int -> String
story 0 =
 "Let's cook and eat that final missionary, and off to bed."
story k =
 "The night was pitch dark, mysterious and deep.\n"
 ++ "Ten cannibals were seated around a boiling cauldron.\n"
 ++ "Their leader got up and addressed them like this:\n'"
 ++ story (k-1) ++ "'"
```

- **Exercise 3.5** What happens if you ask for `putStrLn (story (-1))`? Why?

List Reversal

CHOMSKY

EUGATNOM

List Reversal

CHOMSKY YKSMOHC

EUGATNOM

List Reversal

CHOMSKY YKSMOHC

EUGATNOM MONTAGUE

List Reversal

CHOMSKY YKSMOHC

EUGATNOM MONTAGUE

```
reversal :: [a] -> [a]
reversal []     = []
reversal (x:t) = reversal t ++ [x]
```

List Reversal

CHOMSKY YKSMOHC

EUGATNOM MONTAGUE

```
reversal :: [a] -> [a]
reversal []     = []
reversal (x:t) = reversal t ++ [x]
```

Reversal works for any list, not just for strings.