MATHFUN Lecture FP4
Strings, Tuples and Lists

Matthew Poole

`moodle.port.ac.uk`

School of Computing
University of Portsmouth

2014/15

Introduction to Lecture

- The main aim of this lecture is to introduce Haskell's list type.
- Lists are the main data structure in functional programming languages, and correspond to arrays in imperative languages.

Introduction to Lecture

- The main aim of this lecture is to introduce Haskell's list type.
- Lists are the main data structure in functional programming languages, and correspond to arrays in imperative languages.
- We'll first introduce:
    - characters;
    - strings, which are lists of characters;
    - tuples.

Introduction to Lecture

- The main aim of this lecture is to introduce Haskell's list type.
- Lists are the main data structure in functional programming languages, and correspond to arrays in imperative languages.
- We'll first introduce:
  - characters;
  - strings, which are lists of characters;
  - tuples.
- We'll use Haskell's standard list functions to illustrate **polymorphism** in functional programming.

Introduction to Lecture

- The main aim of this lecture is to introduce Haskell's list type.
- Lists are the main data structure in functional programming languages, and correspond to arrays in imperative languages.
- We'll first introduce:
  - characters;
  - strings, which are lists of characters;
  - tuples.
- We'll use Haskell's standard list functions to illustrate **polymorphism** in functional programming.
- This lecture is supported by Chapter 5 of Thompson's book.

Introduction to Lecture

- The main aim of this lecture is to introduce Haskell's list type.
- Lists are the main data structure in functional programming languages, and correspond to arrays in imperative languages.
- We'll first introduce:
  - characters;
  - strings, which are lists of characters;
  - tuples.
- We'll use Haskell's standard list functions to illustrate **polymorphism** in functional programming.
- This lecture is supported by Chapter 5 of Thompson's book.
- In the following lecture we look at how to write list-processing functions using recursion.

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci>
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
'a' :: Char
ghci>
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
'a' :: Char
ghci> :type "Sam"
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
'a' :: Char
ghci> :type "Sam"
"Sam" :: String
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
'a' :: Char
ghci> :type "Sam"
"Sam" :: String
```

- The `Char` module defines some useful functions on chars:

```
ghci>
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
'a' :: Char
ghci> :type "Sam"
"Sam" :: String
```

- The `Char` module defines some useful functions on chars:

```
ghci> :m + Data.Char
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
'a' :: Char
ghci> :type "Sam"
"Sam" :: String
```

- The `Char` module defines some useful functions on chars:

```
ghci> :m + Data.Char
ghci>
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
'a' :: Char
ghci> :type "Sam"
"Sam" :: String
```

- The `Char` module defines some useful functions on chars:

```
ghci> :m + Data.Char
ghci> toUpper 'a'
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
'a' :: Char
ghci> :type "Sam"
"Sam" :: String
```

- The `Char` module defines some useful functions on chars:

```
ghci> :m + Data.Char
ghci> toUpper 'a'
'A'
ghci>
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
'a' :: Char
ghci> :type "Sam"
"Sam" :: String
```

- The `Char` module defines some useful functions on chars:

```
ghci> :m + Data.Char
ghci> toUpper 'a'
'A'
ghci> isDigit 'a'
```

Characters and Strings

- We begin by noting that Haskell has `Char` and `String` types.
- We use single quotes for chars and double quotes for strings:

```
ghci> :type 'a'
'a' :: Char
ghci> :type "Sam"
"Sam" :: String
```

- The `Char` module defines some useful functions on chars:

```
ghci> :m + Data.Char
ghci> toUpper 'a'
'A'
ghci> isDigit 'a'
False
```

Tuples

- Tuples are used to **combine** pieces of data into one.

Tuples

- Tuples are used to **combine** pieces of data into one.
- Suppose we're writing a program to analyse student marks; we might combine the student names and marks into **tuples**:

  `("Steve",46)` or `("Sam",62)`

Tuples

- Tuples are used to **combine** pieces of data into one.
- Suppose we're writing a program to analyse student marks; we might combine the student names and marks into **tuples**:

  `("Steve",46)` or `("Sam",62)`

- The type of these values is the tuple type `(String,Int)`.

Tuples

- Tuples are used to **combine** pieces of data into one.
- Suppose we're writing a program to analyse student marks; we might combine the student names and marks into **tuples**:

  `("Steve",46)` or `("Sam",62)`

- The type of these values is the tuple type `(String,Int)`.
- An example function definition that processes such tuples is:

```
betterStu :: (String,Int) -> (String,Int) -> String
betterStu (s1,m1) (s2,m2)
    | m1 >= m2           = s1
    | otherwise          = s2
```

Tuples

- Tuples are used to **combine** pieces of data into one.
- Suppose we're writing a program to analyse student marks; we might combine the student names and marks into **tuples**:

  `("Steve",46)` or `("Sam",62)`

- The type of these values is the tuple type `(String,Int)`.
- An example function definition that processes such tuples is:

```
betterStu :: (String,Int) -> (String,Int) -> String
betterStu (s1,m1) (s2,m2)
    | m1 >= m2          = s1
    | otherwise         = s2
```

- For example,

```
ghci> betterStu ("Jim",45) ("Fred",87)
"Fred"
```

Tuples

- It's often a good idea to define a **type synonym**:

```
type StudentMark = (String, Int)
```

  and use `StudentMark` in place of `(String,Int)`.

Tuples

- It's often a good idea to define a **type synonym**:

  ```
  type StudentMark = (String, Int)
  ```

  and use `StudentMark` in place of `(String,Int)`. We re-write:

  ```
  betterStu :: StudentMark -> StudentMark -> String
  betterStu (s1,m1) (s2,m2)
      | m1 >= m2          = s1
      | otherwise         = s2
  ```

Tuples

- It's often a good idea to define a **type synonym**:

  ```
  type StudentMark = (String, Int)
  ```

  and use `StudentMark` in place of `(String,Int)`. We re-write:

  ```
  betterStu :: StudentMark -> StudentMark -> String
  betterStu (s1,m1) (s2,m2)
      | m1 >= m2          = s1
      | otherwise         = s2
  ```

- Tuples can be used in the result type of a function to enable it to return more than one value; e.g.:

  ```
  minAndMax :: Int -> Int -> (Int,Int)
  minAndMax x y
      | x <= y            = (x,y)
      | otherwise         = (y,x)
  ```

Lists

- Lists are used to store any number of data values of the same type; they are the main data structure in Haskell.
- For example:

  ```
  [12, 64, -92, 85, 12]
  ```

  is a list of integers, and

  ```
  ["This", "is", "a", "list"]
  ```

  is a list of strings.

Lists

- Lists are used to store any number of data values of the same type; they are the main data structure in Haskell.
- For example:

  ```
  [12, 64, -92, 85, 12]
  ```

  is a list of integers, and

  ```
  ["This", "is", "a", "list"]
  ```

  is a list of strings.
- For any type `t`, we denote the type of lists of elements from `t` by `[t]`. For example:

  ```
  ghci> :type [True, False, False]
  [True, False, False] :: [Bool]
  ```

- The empty list `[]` is an element of any list type.

Strings as lists of Chars

- Strings in Haskell are simply lists of characters: the type `String` is declared as:

```
type String = [Char]
```

Strings as lists of Chars

- Strings in Haskell are simply lists of characters: the type `String` is declared as:

```
type String = [Char]
```

- Therefore, the following two expressions are the same:

```
['h', 'e', 'l', 'l', 'o']          "hello"
```

Strings as lists of Chars

- Strings in Haskell are simply lists of characters: the type `String` is declared as:

```
type String = [Char]
```

- Therefore, the following two expressions are the same:

```
['h', 'e', 'l', 'l', 'o']          "hello"
```

- The list operations that we see later (e.g. for concatenating lists) thus also apply to strings.

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci>
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci> :type ("Hello", [True, False, True])
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci> :type ("Hello", [True, False, True])
("Hello",[True,False,True]) :: ([Char],[Bool])
ghci>
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci> :type ("Hello", [True, False, True])
("Hello",[True,False,True]) :: ([Char],[Bool])
ghci> :type 3
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci> :type ("Hello", [True, False, True])
("Hello",[True,False,True]) :: ([Char],[Bool])
ghci> :type 3
3 :: Num a => a
ghci>
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci> :type ("Hello", [True, False, True])
("Hello",[True,False,True]) :: ([Char],[Bool])
ghci> :type 3
3 :: Num a => a
ghci> :type [4, 5, 7]
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci> :type ("Hello", [True, False, True])
("Hello",[True,False,True]) :: ([Char],[Bool])
ghci> :type 3
3 :: Num a => a
ghci> :type [4, 5, 7]
[4,5,7] :: Num a => [a]
ghci>
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci> :type ("Hello", [True, False, True])
("Hello",[True,False,True]) :: ([Char],[Bool])
ghci> :type 3
3 :: Num a => a
ghci> :type [4, 5, 7]
[4,5,7] :: Num a => [a]
ghci> :type [(7, "Yes"), (4, "No")]
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci> :type ("Hello", [True, False, True])
("Hello",[True,False,True]) :: ([Char],[Bool])
ghci> :type 3
3 :: Num a => a
ghci> :type [4, 5, 7]
[4,5,7] :: Num a => [a]
ghci> :type [(7, "Yes"), (4, "No")]
[(7,"Yes"),(4,"No")] :: Num a => [(a,[Char])]
ghci>
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci> :type ("Hello", [True, False, True])
("Hello",[True,False,True]) :: ([Char],[Bool])
ghci> :type 3
3 :: Num a => a
ghci> :type [4, 5, 7]
[4,5,7] :: Num a => [a]
ghci> :type [(7, "Yes"), (4, "No")]
[(7,"Yes"),(4,"No")] :: Num a => [(a,[Char])]
ghci> :type []
```

More on types

- What are the types of the following expressions?

```
ghci> :type ["This", "is", "a", "sentence"]
["This","is","a","sentence"] :: [[Char]]
ghci> :type ("Hello", [True, False, True])
("Hello",[True,False,True]) :: ([Char],[Bool])
ghci> :type 3
3 :: Num a => a
ghci> :type [4, 5, 7]
[4,5,7] :: Num a => [a]
ghci> :type [(7, "Yes"), (4, "No")]
[(7,"Yes"),(4,"No")] :: Num a => [(a,[Char])]
ghci> :type []
[] :: [a]
```

Lists from ranges

- We can form lists of numbers/strings using a range format:

  `[n .. m]` gives the list `[n, n+1, ..., , m]`

Lists from ranges

- We can form lists of numbers/strings using a range format:

  `[n .. m]` gives the list `[n, n+1, ..., , m]`

- For example:

```
[3 .. 9]        = [3,4,5,6,7,8,9]
[3.1 .. 9]      = [3.1,4.1,5.1,6.1,7.1,8.1,9.1]
['a' .. 'z']    = "abcdefghijklmnopqrstuvwxyz"
```

Lists from ranges

- We can form lists of numbers/strings using a range format:

  `[n .. m]` gives the list `[n, n+1, ..., , m]`

- For example:

```
[3 .. 9]         = [3,4,5,6,7,8,9]
[3.1 .. 9]       = [3.1,4.1,5.1,6.1,7.1,8.1,9.1]
['a' .. 'z']     = "abcdefghijklmnopqrstuvwxyz"
```

- We can also add an argument to give steps different from 1:

```
[3, 5 .. 15]      = [3,5,7,9,11,13,15]
[0, 0.1 .. 0.5]   = [0.0,0.1,0.2,0.3,0.4,0.5]
```

List comprehensions

- Suppose we define a list `aList` as:

  ```
  aList = [1, 2, 3, 4, 5]
  ```

- Then the **list comprehension**:

  ```
  [ 2*i | i <- aList ]
  ```

  will have the value:

  ```
  [2, 4, 6, 8, 10]
  ```

List comprehensions

- Suppose we define a list `aList` as:

```
aList = [1, 2, 3, 4, 5]
```

- Then the **list comprehension**:

```
[ 2*i | i <- aList ]
```

  will have the value:

```
[2, 4, 6, 8, 10]
```

- We read this as: "take all `2*i` where `i` comes from `aList`."
- (The `<-` is meant to resemble the set member symbol $\in$.)

List comprehensions

- Suppose we define a list `aList` as:

```
aList = [1, 2, 3, 4, 5]
```

- Then the **list comprehension**:

```
[ 2*i | i <- aList ]
```

  will have the value:

```
[2, 4, 6, 8, 10]
```

- We read this as: "take all `2*i` where `i` comes from `aList`."
- (The `<-` is meant to resemble the set member symbol $\in$.)
- List comprehensions are a powerful feature (almost) unique to functional programming languages.
- (A few imperative languages (e.g. Python) have some functional programming facilities including list comprehensions.)

List comprehensions

- Let's look at some further examples; suppose here that the list `aList` is defined as:

```
aList = [2,3,6,9,4,7]
```

List comprehensions

- Let's look at some further examples; suppose here that the list `aList` is defined as:

```
aList = [2,3,6,9,4,7]
```

- Then:

```
ghci> [ mod i 2 == 0 | i <- aList ]
[True,False,True,False,True,False]
```

List comprehensions

- Let's look at some further examples; suppose here that the list `aList` is defined as:

```
aList = [2,3,6,9,4,7]
```

- Then:

```
ghci> [ mod i 2 == 0 | i <- aList ]
[True,False,True,False,True,False]
```

- We can add a **test** at the end of the **generator** `i <- aList`:

```
ghci> [ i*2 | i <- aList, i < 5 ]
[4,6,8]
```

- Here, the generator/test has given all the values in `aList` that are less than 5; i.e. 2, 3, 4.

List comprehensions

- Let's look at some further examples; suppose here that the list `aList` is defined as:

```
aList = [2,3,6,9,4,7]
```

- Then:

```
ghci> [ mod i 2 == 0 | i <- aList ]
[True,False,True,False,True,False]
```

- We can add a **test** at the end of the **generator** `i <- aList`:

```
ghci> [ i*2 | i <- aList, i < 5 ]
[4,6,8]
```

- Here, the generator/test has given all the values in `aList` that are less than 5; i.e. 2, 3, 4.

List comprehensions

- Instead of a single variable, we can use a **pattern** on the left hand side of the <-.
- For example, consider the definition:

```
addPairs :: [(Int,Int)] -> [Int]
addPairs pairList = [ i+j | (i,j) <- pairList ]
```

will choose all pairs in `pairList` and add their components;

List comprehensions

- Instead of a single variable, we can use a **pattern** on the left hand side of the <-.
- For example, consider the definition:

```
addPairs :: [(Int,Int)] -> [Int]
addPairs pairList = [ i+j | (i,j) <- pairList ]
```

  will choose all pairs in `pairList` and add their components; e.g.

```
ghci> addPairs [(1,2), (4,8), (6,3)]
[3,12,9]
```

List comprehensions

- Instead of a single variable, we can use a **pattern** on the left hand side of the `<-`.
- For example, consider the definition:

```
addPairs :: [(Int,Int)] -> [Int]
addPairs pairList = [ i+j | (i,j) <- pairList ]
```

  will choose all pairs in `pairList` and add their components; e.g.

```
ghci> addPairs [(1,2), (4,8), (6,3)]
[3,12,9]
```

- For some final examples, recall the type synonym:

```
type StudentMark = (String, Int)
```

  for student name/mark data such as `("Steve", 46)`.

List comprehensions

- Instead of a single variable, we can use a **pattern** on the left hand side of the `<-`.
- For example, consider the definition:

```
addPairs :: [(Int,Int)] -> [Int]
addPairs pairList = [ i+j | (i,j) <- pairList ]
```

  will choose all pairs in `pairList` and add their components; e.g.

```
ghci> addPairs [(1,2), (4,8), (6,3)]
[3,12,9]
```

- For some final examples, recall the type synonym:

```
type StudentMark = (String, Int)
```

  for student name/mark data such as `("Steve", 46)`.
- An example of the type `[StudentMark]` of name/mark lists is:

```
[("Sam", 67), ("Kate", 35), ("Jill", 75)]
```

List comprehensions

- The following definition will generate a list of just the marks:

```
marks :: [StudentMark] -> [Int]
marks stMarks = [ mk | (st,mk) <- stMarks ]
```

List comprehensions

- The following definition will generate a list of just the marks:

```
marks :: [StudentMark] -> [Int]
marks stMarks = [ mk | (st,mk) <- stMarks ]
```

- For example,

```
ghci> marks [("Sam",67), ("Kate",35), ("Jill",75)]
[67,35,75]
```

List comprehensions

- The following definition will generate a list of just the marks:

```
marks :: [StudentMark] -> [Int]
marks stMarks = [ mk | (st,mk) <- stMarks ]
```

- For example,

```
ghci> marks [("Sam",67), ("Kate",35), ("Jill",75)]
[67,35,75]
```

- The following gives the names of students who have passed:

```
pass :: [StudentMark] -> [String]
pass stMarks = [ st | (st,mk) <- stMarks, mk >= 40 ]
```

List comprehensions

- The following definition will generate a list of just the marks:

```
marks :: [StudentMark] -> [Int]
marks stMarks = [ mk | (st,mk) <- stMarks ]
```

- For example,

```
ghci> marks [("Sam",67), ("Kate",35), ("Jill",75)]
[67,35,75]
```

- The following gives the names of students who have passed:

```
pass :: [StudentMark] -> [String]
pass stMarks = [ st | (st,mk) <- stMarks, mk >= 40 ]
```

- For example:

```
ghci> pass [("Sam", 67), ("Kate", 35), ("Jill", 75)]
["Sam","Jill"]
```

Polymorphic functions

- We now look at some list functions defined in the Prelude.
- Doing so also provides a good opportunity to introduce the concept of **polymorphism**.

Polymorphic functions

- We now look at some list functions defined in the Prelude.
- Doing so also provides a good opportunity to introduce the concept of **polymorphism**.
- Consider first the `length` function, which gives the number of elements in a list (i.e. it returns an `Int`).

Polymorphic functions

- We now look at some list functions defined in the Prelude.
- Doing so also provides a good opportunity to introduce the concept of **polymorphism**.
- Consider first the `length` function, which gives the number of elements in a list (i.e. it returns an `Int`).
- This function works for **any** type of list; we might say:

```
length :: [String] -> Int
length :: [Bool] -> Int
```

- We call a function that has many types a **polymorphic** function.

Polymorphic functions

- We now look at some list functions defined in the Prelude.
- Doing so also provides a good opportunity to introduce the concept of **polymorphism**.
- Consider first the `length` function, which gives the number of elements in a list (i.e. it returns an `Int`).
- This function works for **any** type of list; we might say:

```
length :: [String] -> Int
length :: [Bool] -> Int
```

- We call a function that has many types a **polymorphic** function.
- The actual type of `length` is given as:

```
length :: [a] -> Int
```

- Here `a` is a **type variable** that stands for an **arbitrary type** (i.e. its possible values are types).

Polymorphic functions

- (By convention, `a`, `b`, `c`, ... are used as type variables.)
- Types `[String] -> Int` and `[Bool] -> Int` are **instances** of type `[a] -> Int` (they are found by replacing `a` with a type).
- `[a] -> Int` is known as the **most general type** for `length`.

Polymorphic functions

- (By convention, `a`, `b`, `c`, ... are used as type variables.)
- Types `[String] -> Int` and `[Bool] -> Int` are **instances** of type `[a] -> Int` (they are found by replacing `a` with a type).
- `[a] -> Int` is known as the **most general type** for `length`.
- We can define our own polymorphic functions; the easiest way to do this is to omit a type declaration.

Polymorphic functions

- (By convention, `a`, `b`, `c`, ... are used as type variables.)
- Types `[String] -> Int` and `[Bool] -> Int` are **instances** of type `[a] -> Int` (they are found by replacing `a` with a type).
- `[a] -> Int` is known as the **most general type** for `length`.
- We can define our own polymorphic functions; the easiest way to do this is to omit a type declaration.
- Supposing, for example, we define:

```
square n = n * n
```

- Haskell will infer the most general type by looking at the structure of the function.

Polymorphic functions

- (By convention, `a`, `b`, `c`, ... are used as type variables.)
- Types `[String] -> Int` and `[Bool] -> Int` are **instances** of type `[a] -> Int` (they are found by replacing `a` with a type).
- `[a] -> Int` is known as the **most general type** for `length`.
- We can define our own polymorphic functions; the easiest way to do this is to omit a type declaration.
- Supposing, for example, we define:

```
square n = n * n
```

- Haskell will infer the most general type by looking at the structure of the function. Here:

```
square :: Num a => a -> a
```

  says that the type is `a -> a` where `a` can be any **numeric type**.

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci>
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
5
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
5
```

- The most used operation is `:` which adds an element to a list:

```
ghci>
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
5
```

- The most used operation is `:` which adds an element to a list:

```
ghci> :type (:)
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
5
```

- The most used operation is `:` which adds an element to a list:

```
ghci> :type (:)
(:) :: a -> [a] -> [a]
ghci>
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
5
```

- The most used operation is `:` which adds an element to a list:

```
ghci> :type (:)
(:) :: a -> [a] -> [a]
ghci> 3:[5, 7, 2]
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
5
```

- The most used operation is `:` which adds an element to a list:

```
ghci> :type (:)
(:) :: a -> [a] -> [a]
ghci> 3:[5, 7, 2]
[3,5,7,2]
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
5
```

- The most used operation is `:` which adds an element to a list:

```
ghci> :type (:)
(:) :: a -> [a] -> [a]
ghci> 3:[5, 7, 2]
[3,5,7,2]
```

- The ++ operator joins two lists together, and `!!` returns an element at a given position. (Question: what are their types?)

```
ghci>
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
5
```

- The most used operation is `:` which adds an element to a list:

```
ghci> :type (:)
(:) :: a -> [a] -> [a]
ghci> 3:[5, 7, 2]
[3,5,7,2]
```

- The ++ operator joins two lists together, and `!!` returns an element at a given position. (Question: what are their types?)

```
ghci> "hello" ++ "there"
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
5
```

- The most used operation is `:` which adds an element to a list:

```
ghci> :type (:)
(:) :: a -> [a] -> [a]
ghci> 3:[5, 7, 2]
[3,5,7,2]
```

- The ++ operator joins two lists together, and `!!` returns an element at a given position. (Question: what are their types?)

```
ghci> "hello" ++ "there"
"hellothere"
ghci>
```

List functions

- Let's try `length` and a few of the Prelude's list operators:

```
ghci> length "hello"
5
```

- The most used operation is `:` which adds an element to a list:

```
ghci> :type (:)
(:) :: a -> [a] -> [a]
ghci> 3:[5, 7, 2]
[3,5,7,2]
```

- The ++ operator joins two lists together, and `!!` returns an element at a given position. (Question: what are their types?)

```
ghci> "hello" ++ "there"
"hellothere"
ghci> [5, 7, 8, 4] !! 2
8
```