Lesson 23. Working with text and Unicode

After reading lesson 23, you'll be able to

- Use the Text type for more-efficient text processing
- Change Haskell's behavior with language extensions
- Program by using common text functions
- Use Text to properly handle Unicode text

So far in this book, you've made heavy use of the String type. In the preceding lesson, you saw that you can even view an I/O stream as a lazy list of type Char, or a String. String has been useful in helping you explore many topics in this book. Unfortunately, String has a huge problem: it can be woefully inefficient.

From a philosophical standpoint, nothing could be more perfect than representing one of the more important types in programing as one of the most foundational data structures in Haskell: a list. The problem is that a list isn't a great data structure to store data for heavy string processing. The details of Haskell performance are beyond the scope of this book, but it suffices to say that implementing Strings as a linked list of characters is needlessly expensive in terms of both time and space.

In this lesson, you'll take a look at a new type, Text. You'll explore how to replace String with Text for more-efficient text processing. Then you'll learn about the functions common to both String and Text for processing text. Finally, you'll learn about how Text handles Unicode by building a function that can highlight search text, even in Sanskrit!

CONSIDER THIS

In Haskell, String is a special case of a List. But in most programming languages, string types are stored much more efficiently as arrays. Is there a way in Haskell to use the tools you already know about for working with the String type, but still have the efficiency of an array-based implementation?

livebook features:
highlight, annotate, and bookmark

Select a piece of text and click the appropriate icon to comment, bookmark, or highlight

view how

23.1. The Text type

For practical and commercial Haskell programming, the preferred type for working with text data is the type Text. The Text type can be found in the module Data.Text. In practice, Data.Text is almost always imported as a qualified import by using a single letter, usually T:

```
import qualified Data.Text as T
```

copy

Unlike String, Text is implemented as an array under the hood. This makes many string operations faster and much more memory-efficient. Another major difference between Text and String is that Text *doesn't* use lazy evaluation. Lazy evaluation proved to be helpful in the preceding lesson, but in many real-world cases it can lead to performance headaches. If you do need lazy text, you can use Data.Text.Lazy, which has the same interface as Data.Text.

23.1.1. When to use Text vs. String

In the commercial Haskell community, Data.Text is strongly preferred over String. Some members of the Haskell community argue that the standard Prelude should be thrown out for anything practical due to the heavy dependency on String. While learning Haskell, String is useful for two reasons. First, as mentioned, many of the basic string utilities are baked into the standard Prelude. Second, lists are to Haskell what arrays are to C. Many concepts in Haskell are nicely demonstrated with lists, and strings are useful lists. For learning purposes, feel free to stick with String. But for anything beyond exercises, use Data.Text as much as possible. You'll continue to use String in many places in this book but will start to use Data.Text more often.

livebook features:

discuss

Ask a question, share an example, or respond to another reader. Start a thread by selecting any piece of text and clicking the discussion icon.

view how

open discussions

23.2. Using Data.Text

The first thing you need to do is learn how to use the Text type. Data.Text has two functions, pack and unpack, which can be used to convert String -> Text and Text -> String. Figuring out which function does what can easily be determined by their type signatures:

```
T.pack :: String -> T.Text
```

```
T.unpack :: T.Text -> String
```
copy
Here are some examples of converting a String to Text and back again.
Listing 23.1. Converting back and forth between String and Text types
```
1
2
3
4
5
6
7
8
firstWord :: String
firstWord = "pessimism"

secondWord :: T.Text
secondWord = T.pack firstWord

thirdWord :: String
thirdWord = T.unpack secondWord
```
copy
It's important to note that conversion isn't computationally cheap, because you have to traverse the entire string. Avoid converting back and forth between Text and String.
QUICK CHECK 23.1
Q1:
Create fourthWord once again, making the String type T.Text.
QC 23.1 ANSWER
1:
```
fourthWord :: T.Text
fourthWord = T.pack thirdWord
```
copy
23.2.1. OverloadedStrings and Haskell extensions
An annoying thing about T.Text is that this code throws an error.
Listing 23.2. The problem with using literal strings to define Text
```
1
2
myWord :: T.Text
myWord = "dog"
```

copy
The error you get reads as follows:
Couldn't match expected type 'T.Text' with actual type '[Char]'
copy
This error occurs because the literal "dog" is a String. This is particularly annoying because you don't have this problem with numeric types. Take, for example, these numbers.

Listing 23.3. The same numeric literal used in three types

```
myNum1 :: Int
myNum1 = 3

myNum2 :: Integer
myNum2 = 3

myNum3 :: Double
myNum3 = 3
```

copy
This code will compile just fine even though you've used the same literal, 3, for three different types.
Clearly this isn't a problem that you can solve with clever coding, no matter how powerful Haskell may be. To fix this issue, you need a way to fundamentally change how GHC reads your file. Surprisingly, an easy fix for this exists! GHC allows you to use *language extensions* to alter the way Haskell itself works. The specific extension you're going to use is called OverloadedStrings.
There are two ways to use a language extension. The first is by using it when compiling with GHC. To do this, use the flag -X followed by the extension name. For a program named text.hs, this looks like the following:

```
$ ghc text.hs -XOverloadedStrings
```

copy
This can also be used as an argument to GHCi, to start an instance of GHCi by using the language extension.

The trouble is that someone who is using your code (and that someone could be you) might not remember to use this flag. A preferred method is to use a LANGUAGE pragma. The pragma looks like this:

```
{-# LANGUAGE <Extension Name> #-}
```

copy

Here's a text.hs file that will allow you to use literal values for Text types.

Listing 23.4. Using OverloadedStrings to easily assign Text using a literal

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T

aWord :: T.Text
aWord = "Cheese"

main :: IO ()
main = do
  print aWord
```

copy

With the LANGUAGE pragma, you can compile this program just like any other Haskell program.

Language extensions are powerful and range from practical to experimental. In real-world Haskell, a few extensions are common and useful.

OTHER USEFUL LANGUAGE EXTENSIONS

Language extensions are common in practical Haskell. They're powerful, as they allow you to use features of Haskell that may not be available as a default in the language for years, if ever. OverloadedStrings is the most common. Here are a few others you may come across or find useful:

- ViewPatterns—Allows for more-sophisticated pattern matching.
- TemplateHaskell—Provides tools for Haskell metaprogramming.
- DuplicateRecordFields—Solves the annoying problem from lesson 16, where using the same field name for different types using record syntax

causes a conflict.

- NoImplicitPrelude—As mentioned, some Haskell programmers prefer to use a custom Prelude. This language extension allows you to not use the default Prelude.

QUICK CHECK 23.2

Q1:

There's a language extension called TemplateHaskell. How would you compile templates.hs to use this extension? How would you add it using a LANGUAGE pragma?

QC 23.2 ANSWER

1:

```
$ghc templates.hs -XTemplateHaskell

{-# LANGUAGE TemplateHaskell -#}
```

copy

23.2.2. Basic Text utilities

The trouble with using Text instead of String is that most useful functions for working with text are intended to be used with the String type. You definitely don't want to be converting Text back to String in order to use functions such as lines. Luckily, nearly every important String function has its own version for working on Text in Data.Text. Here's some sampleInput you'll work with to show how these functions work.

Listing 23.5. sampleInput of type Text

```
1
2
sampleInput :: T.Text
sampleInput = "this\nis\ninput"
```

copy

To use lines on this example, all you have to do is make sure you preface lines with T., because of your qualified import. Here's an example in GHCi:

```
GHCi>T.lines sampleInput
["this","is","input"]
```

copy

The following are a few other useful functions that exist for both Text and String.

words

The words function is the same as lines, but it works for any whitespace characters, rather than just new lines.

Listing 23.6. someText as a sample input for words

```
1
```

```
2
someText :: T.Text
someText = "Some\ntext for\t you"
```

copy

In GHCi, you can easily see how this works:

```
GHCi> T.words someText
["Some","text","for","you"]
```

copy

splitOn

Lesson 22 briefly mentioned splitOn. For strings, splitOn is part of the Data.List.Split module. Thankfully, the text version is included in Data.Text so no additional import is needed. splitOn lets you split up text by any substring of text.

Listing 23.7. Code for splitOn example

```
1
2
3
4
5
breakText :: T.Text
breakText = "simple"

exampleText :: T.Text
exampleText = "This is simple to do"
```

copy

And in GHCi:

```
GHCi> T.splitOn breakText exampleText
["This is "," to do"]
```

copy

unwords and unlines

Breaking up Text by using whitespace is fairly common when working with I/O. The inverse is also common, so two functions can undo what you've just done, conveniently called unlines and unwords. Their usage is fairly obvious, but they're useful functions to have in your tool belt:

```
GHCi> T.unlines (T.lines sampleInput)
"this\nis\ninput\n"
GHCi> T.unwords (T.words someText)
"Some text for you"
```

copy

Intercalate

You've used the string version of intercalate before in lesson 18. It's the opposite of splitOn:

```
GHCi> T.intercalate breakText (T.splitOn breakText exampleText)
"This is simple to do"
```

copy

Almost any useful function for working with strings works on text and has its own Text version.

Monoid operations

The exception to the rule that most useful functions on strings work on text is the ++ operator. So far, you've used ++ to combine strings:

```
combined :: String
combined = "some" ++ " " ++ "strings"
```

copy

Unfortunately, ++ is defined only on the List type, so it won't work for Text. In lesson 17, we discussed the Monoid and Semigroup type classes, which allow you to combine like types and concatenate lists of the same type. This provides a general solution to combining both strings and text. You can either import Semigroup and use <> to combine text, or use mconcat:

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T
import Data.Semigroup

combinedTextMonoid :: T.Text
combinedTextMonoid = mconcat ["some"," ","text"]

combinedTextSemigroup :: T.Text
combinedTextSemigroup = "some" <> " " <> "text"
```

copy

Because String is also an instance of Monoid and Semigroup, strings can be combined in the same way.

QUICK CHECK 23.3

Q1:

Create your own version of T.lines and T.unlines by using splitOn and T.intercalate.

QC 23.3 ANSWER

1:

```
myLines :: T.Text -> [T.Text]
myLines text = T.splitOn "\n" text

myUnlines :: [T.Text] -> T.Text
```

myUnlines textLines = T.intercalate "\n" textLines
copy

livebook features:
settings

Update your profile, view your dashboard, tweak the text size, or turn on dark mode.

view how

23.3. Text and Unicode

The Text type has excellent support for working seamlessly with Unicode text. At one point, programmers could largely ignore the complications of working with non-ASCII text. If input had accents or umlauts, it could be squashed out of existence; it was acceptable to change *Charlotte Brontë to Charlotte Bronte*. But ignoring Unicode today and in the future is a recipe for disaster. There's no reason to be unable to record a user's name that includes diacritical marks, or to fail to handle Japanese Kanji.

23.3.1. Searching Sanskrit

To demonstrate how seamlessly you can use Text for working with Unicode characters, you'll build a simple program that highlights words in text. The trick is that you're going to be highlighting Sanskrit words written in Devanagari script! The Unicode text can be easily copied from this link if you want to paste this into your editor to follow along: https://gist.github.com/willkurt/4bced09adc2ff9e7ee366b7ad681cac6.

All of your code will go in a file named bg_highlight.hs. Your program will take a text query and a body of text, and use curly braces, {}, to highlight all cases of the word you're looking for. For example, if *dog* is your query text, and your main text is *a dog walking dogs*, you'd expect this output:

a {dog} walking {dog}s
copy

In this task, you want to highlight the Sanskrit word *dharma* in a sample text from the *Bhavagad Gita*. The word *dharma* has many meanings in Sanskrit, ranging from *duty* to references of cosmic order and divine justice. Sanskrit is a language that has no singular writing system. The most popular today is Devanagari, an alphabet used by more than 120 languages, including Hindi. Here's the Sanskrit word *dharma* written in Devanagari script.

Listing 23.8. A Unicode text variable for dharma written in Devanagari script

1
2

```
dharma :: T.Text
dharma = ""
```

copy

Next you'll take an excerpt from the *Bhavagad Gita*, itself a part of the Indian epic, *The Mahabharata*. Here's our section.

Listing 23.9. Your search text from the Bhavagad Gita

```
1
2
bgText :: T.Text
bgText = ""
```

copy

Your goal here is to highlight everywhere in your bgText where the word *dharma* appears. In English, your first thought might be to split a sentence by using T.words, and then look for the word you're looking for. But Sanskrit is more complicated. Because Sanskrit was a spoken language long before it was written, whenever words are naturally combined when speaking a sentence, they end up combined in text. To solve this, you can split your text on the target text query, wrap the query in brackets, and then put it all back together. You can use T.splitOn to split up the text, mconcat to add brackets to your query string, and T.intercalate to piece your words back together.

Here's your highlight function.

Listing 23.10. The highlight function for highlighting text segments

```
1
2
3
4
highlight :: T.Text -> T.Text -> T.Text
highlight query fullText = T.intercalate highlighted pieces
  where pieces = T.splitOn query fullText
        highlighted = mconcat ["{",query,"}"]
```

1
2
3

copy

-

Finally, you can put this all together in your main. But first you have to learn how to use IO with your Text type.

23.4. Text I/O

Now that you have a highlight function, you want to print the results of your highlighting back to the users. The trouble is that so far you've always used an IO String type to send output to the user. One solution would be to unpack your end text back into a string. What you want is to have a putStrLn for Text; this way, you never have to convert your text to a string (and can hopefully forget about strings altogether). The Data.Text module includes only functions for manipulating text. To perform text I/O, you need to import the Data.Text.IO package. You'll do another qualified import:

```
import qualified Data.Text.IO as TIO
```


With TIO.putStrLn, you can print your Text type just as you would String. Any IO action you've used related to the String type has an equivalent in Data.Text.IO. Now you can put together your main, which calls your highlight function on your data. Here's your full file, including the necessary imports and LANGUAGE pragma.

Listing 23.11. Full file for your program

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

```
15
16
17
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T
import qualified Data.Text.IO as TIO

dharma :: T.Text
dharma :: ""

bgText :: T.Text
bgText = ""

highlight :: T.Text -> T.Text -> T.Text
highlight query fullText = T.intercalate highlighted pieces
  where pieces = T.splitOn query fullText
        highlighted = mconcat ["{",query,"}"]

main = do
  TIO.putStrLn (highlight dharma bgText)
```

copy

You can compile your program and see the highlighted text:

```
$./bg_highlight
```

copy

Now you have a program that easily handles Unicode and also works with text data much more efficiently than String.

Summary

In this lesson, our objective was to teach you how to efficiently process text (including Unicode) in Haskell by using Data.Text. Although strings as lists of characters are a useful tool for teaching Haskell, in practice they can lead to poor performance. The preferred alternative whenever you're working with text data is to use the Data.Text module. One issue you came across was that Haskell, by default, doesn't know how to understand string literals as Data.Text. This can be remedied by using the OverloadedStrings language extension. Let's see if you got this.

Q23.1

Rewrite the hello_world.hs program (reproduced here) from lesson 21 to use Text instead of String types.

```
helloPerson :: String -> String
```

```
helloPerson name = "Hello" ++ " " ++ name ++ "!"

main :: IO ()
main = do
  putStrLn "Hello! What's your name?"
  name <- getLine
  let statement = helloPerson name
  putStrLn statement
```
copy

Q23.2

Use Data.Text.Lazy and Data.Text.Lazy.IO to rewrite the lazy I/O section from lesson 22 by using the Text type.

```
toInts :: String -> [Int]
toInts = map read . lines

main :: IO ()
main = do
 userInput <- getContents
 let numbers = toInts userInput
 print (sum numbers)
```
copy