String Types

NOTE This tutorial contains content from two different documents. It still needs to be harmonized.

The two primary packages for dealing with string-like data in Haskell are bytestring and text. The former is for working with binary data, and the latter for textual data. While there can be some overlap between these two, for the most part the choice between them is straightforward.

This document will demonstrate how to use these libraries, motivate when to choose different representations, address some concerns o backwards compatibility with other string representations, and recommend how to deal with more complicated cases.

Synopsis (bytestring)

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as S
import Data.Monoid ((<>))
import Data.Word (Word8)

main :: IO ()
main = do
    S.writeFile "content.txt" "This is some sample content"
    bs <- S.readFile "content.txt"
    print bs
    print $ S.takeWhile (/= space) bs
    print $ S.take 5 bs
    print $ "File contents: " <> bs

    putStrLn $ "Largest byte: " ++ show (S.foldl1' max bs)
    -- Or just use S.maximum

    putStrLn $ "Spaces: " ++ show (S.length (S.filter (== space) bs))
  where
    space :: Word8
    space = 32 -- ASCII code
```

Synopsis (text)

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import Data.Monoid ((<>))

main :: IO ()
main = do
    TIO.writeFile "content.txt" "This is some sample content"
    text <- TIO.readFile "content.txt"
    print text
    print $ T.takeWhile (/= ' ') text
    print $ T.take 5 text
    print $ "File contents: " <> text

    putStrLn $ "Largest character: " ++ show (T.foldl1' max text)
    -- Or just use T.maximum

    putStrLn $ "Spaces: " ++ show (T.length (T.filter (== ' ') text))
```

OverloadedStrings

The first thing worth pointing out here is the usage of `{-# LANGUAGE OverloadedStrings #-}` in both examples above. This language extension allows us to generalize string literals like `"foo"` to be treated by the GHC compiler as arbitrary string-like data types including `ByteString` and `Text`. We can also explicitly perform this conversion from string literal to the relevant datatypes using `pack` functions:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Data.ByteString (ByteString)
import qualified Data.ByteString as S
import qualified Data.ByteString.Char8 as S8
import Data.Text (Text)
import qualified Data.Text as T

main :: IO ()
main = do
    print (S8.pack "This is now a ByteString" :: ByteString)
    print (T.pack "This is now a Text" :: Text)
```

We had to use the `Data.ByteString.Char8` module for packing the string into a `ByteString`. This module will implicitly coerce characters into bytes, and for any characters outside the range of a byte will perform truncation. This is our first important point in the distinction between `ByteString` and `Text`, consider:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import Data.ByteString (ByteString)
import qualified Data.ByteString as S
import qualified Data.ByteString.Char8 as S8
import Data.Text (Text)
import qualified Data.Text as T

main :: IO ()
main = do
    print (S8.pack "ByteString: Non Latin characters: שלום" :: ByteString)
    print (T.pack  "Text:       Non Latin characters: שלום" :: Text)
```

These two lines output different results:

```
"ByteString: Non Latin characters: \233\220\213\221"
"Text:       Non Latin characters: \1513\1500\1493\1501"
```

The `ByteString` version has to truncate the characters to the 0-255 range, whereas the `Text` version represents the full range of Unicode points.

List-like API

Both `Text` and `ByteString` are quite similar to lists, in that they represent a sequence of values. All three data types have very similar APIs, which you've already seen some of in the synopses above. Functions you're already used to like `take`, `drop`, `break`, `foldl'`, and others are all available. For the most part, if you're familiar with the list API, you can stick a `S.` or `T.` in front of the function name and sta working with a `ByteString` or `Text`.

Reading through the Data.ByteString and Data.Text API documentation will provide an exhaustive list of available functions.

Differences from lists

For all the similarity with lists, there are certainly some differences which need to be pointed out:

- Lists are polymorphic, in that they can hold multiple types. `ByteString` and `Text` are both monomorphic, holding only `Word8` and `Char`, respectively. This affects the type of functions like `map`, and prevents instances of typeclasses like `Functor` and `Foldable`
- Both of these datatypes are strict, as opposed to lists, which are lazy. You can demonstrate this with:

  ```
  #!/usr/bin/env stack
  -- stack --resolver lts-12.21 script
  import qualified Data.ByteString as S
  import qualified Data.ByteString.Char8 as S8
  import qualified Data.Text as T

  main :: IO ()
  main = do
      print $ take 5 ['h', 'e', 'l', 'l', 'o', undefined]
      print $ T.take 5 $ T.pack ['h', 'e', 'l', 'l', 'o', undefined]
      print $ S.take 5 $ S8.pack ['h', 'e', 'l', 'l', 'o', undefined]
  ```

  The first line based on lists will print `"hello"`, whereas both of the following lines will throw an exception.

- The memory representation of `ByteString` and `Text` is packed into a memory buffer, avoiding heap objects, pointer indirection, and improving cache friendliness of algorithms. Lists have a far higher memory overhead for the same data payload.
- As a result of the previous two points, `ByteString` and `Text` are both finite data structures, as opposed to lists which provide for infinite data representation. (Though see below for comments on lazy `ByteString` and `Text`.)

Converting between ByteString and Text

A `Text` value represents a sequence of Unicode code points, whereas a `ByteString` represents a sequence of bytes. In order to conver between these two concepts, you need to select a character encoding. Many common encodings are available in `Data.Text.Encodin` Let's play with some of this:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as S
import qualified Data.Text as T
import qualified Data.Text.Encoding as TE

main :: IO ()
main = do
    let text = "This is some text, with non-Latin chars: שלום"
        bs = TE.encodeUtf8 text
    S.writeFile "content.txt" bs
    bs2 <- S.readFile "content.txt"
    let text2 = TE.decodeUtf8 bs2
    print text2
```

Probably the most common character encoding in the world is UTF8, especially due to its compatibility with ASCII data. The `encodeUtf` function converts a `Text` into a `ByteString`, while `decodeUtf8` converts from a `ByteString` to `Text`. Unfortunately, there's a slight problem with using this function: it's partial, meaning that if an invalid UTF8 character sequence is detected, it will generate an exception For example:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text.Encoding as TE

main :: IO ()
main = do
    let bs = "Invalid UTF8 sequence\254\253\252"
    print $ TE.decodeUtf8 bs
```

This will output:

```
"foo.hs: Cannot decode byte '\xfe': Data.Text.Internal.Encoding.decodeUtf8: Invalid UTF-8 stream
```

There are two recommended ways to avoid this. One is to do lenient decoding:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text.Encoding as TE
import qualified Data.Text.Encoding.Error as TEE

main :: IO ()
main = do
    let bs = "Invalid UTF8 sequence\254\253\252"
    print $ TE.decodeUtf8With TEE.lenientDecode bs
```

This will replace invalid encoding sequences with the Unicode replacement character. Another is to use a function which is explicit in the error condition, like `decodeUtf8'`:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text.Encoding as TE

main :: IO ()
main = do
    let bs = "Invalid UTF8 sequence\254\253\252"
    case TE.decodeUtf8' bs of
        Left e -> putStrLn $ "An exception occurred: " ++ show e
        Right text -> print text
```

So many types!

One complaint you'll often hear is that Haskell has too many string types, which may seem at odds with the breakdown in this article so fa of just two types with a clear delineation between them (binary vs textual). Usually there are five types referenced: `String`, strict `ByteString`, lazy `ByteString`, strict `Text`, and lazy `Text`.

One reason for this difference is the presence of lazy datatypes in these packages (Data.ByteString.Lazy and Data.Text.Lazy). While these data types certainly have some use cases, my overall recommendation is: don't use them. If you need to deal with data too large to fit into memory, you should use a streaming data library instead, like conduit.

By avoiding the strict/lazy choice: we've removed two of the five types listed above. The final choice is `String`, which is the only string-like type defined in the base package. It is actually just a type synonym `type String = [Char]`. While this type is conceptually simpl it is highly inefficient (as mentioned above). You should avoid using it whenever possible, replacing instead with `Text`.

There are unfortunately some functions and typeclasses from base - `error`, `Show`, `Read` - that rely on `String`, making it impossible to completely avoid using it. The best approach is to keep `Text` internal to your program as long as possible, and convert to and from `String` only when absolutely necessary for compatibility with existing libraries.

I/O and character encodings

There is a plethora of ways to interact with files and file handles for `Text` and `ByteString`. I'll start off with the recommendations, and then explain:

- When interacting over standard input/output/error, use the functions from `Data.Text.IO`.
- When reading from files, sockets, or any other source, always use `ByteString`-based functions and then, if necessary, use explicit

character decoding functions to convert to `Text`.

When performing `Handle`-based I/O, the text package API will respect environment variables specifying the language settings and character encodings for the user. When interacting on standard input/output/error, this usually makes sense*. However, in my experience when this logic is applied to files, it leads to bugs far more often than not, usually when running a program on a different system with different environment variables. For example, consider this program:

* Though not always, when dealing with input redirection from a file this can cause confusion.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as S
import qualified Data.Text.IO as TIO
import qualified Data.Text.Encoding as TE

main :: IO ()
main = do
    S.writeFile "utf8-file.txt" $ TE.encodeUtf8 "hello hola שלום"
    text <- TIO.readFile "utf8-file.txt"
    print text
```

This program uses the bytestring API to write to a file, and writes data which is explicitly UTF8 encoded. It then uses the text API to read from that same file. On many systems, running this program will work just fine. However, it's trivial to break. For example, if I compile this program and run it inside a Docker image without any explicit language environment variables set, the result is:

```
utf8-file.txt: hGetContents: invalid argument (invalid byte sequence)
```

The basic premise is: file formats should typically be explicit in their character encoding.

By contrast, if you're going to work with stdin/stdout/stderr, using the character encoding determination logic works pretty well. As an example:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Data.Monoid ((<>))
import qualified Data.Text.IO as TIO

main :: IO ()
main = do
    TIO.putStrLn "What is your name?"
    name <- TIO.getLine
    TIO.putStrLn $ "Hello, " <> name
```

FFI and internal representation

Executive summary: whenever working with external APIs, you'll end up using `ByteString`.

There are a few differences in the internal representations of `Text` and `ByteString`. For the most part, you don't need to be aware of them, but when dealing with external APIs, they become important:

- A `Text` value is internally a UTF-16 encoded representation, whereas a `ByteString` - as arbitrary binary data - has no character encoding associated with it. While this is true at the time of writing, there is no inherent guarantee about the character encoding used by text, and in the past, efforts have been made to switch to different encodings.
- A `Text` value uses *unpinned memory* - meaning that it is controlled by the GHC garbage collector and can be moved around at will By contrast, a `ByteString` value uses *pinned memory*, meaning that it is locked down to a single memory address and cannot be changed.

These two points combine to something important for foreign function calls: you cannot use a `Text` value directly. If you somehow did ge the memory address of a `Text` value, it may be changed while your code is making an FFI call, leading to corrupted data or, worse, a segfault. And if you rely on UTF-16 encoding, you may be safe, but it's possible that your code will break in the future.

Instead, if you need to interact with an FFI and you have a `Text` value, convert it to a `ByteString` and then use the appropriate functio e.g.:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ForeignFunctionInterface #-}
import Data.Monoid ((<>))
import qualified Data.Text.Encoding as TE
import qualified Data.Text.IO as TIO
import Foreign.Ptr (Ptr)
import Foreign.C.Types (CChar)
import Data.ByteString.Unsafe (unsafeUseAsCStringLen)

foreign import ccall "write"
    c_write :: Int -> Ptr CChar -> Int -> IO ()

main :: IO ()
main = do
    TIO.putStrLn "What is your name?"
    name <- TIO.getLine
    let msg = "Hello, " <> name <> "\n"
        bs = TE.encodeUtf8 msg
    unsafeUseAsCStringLen bs $ \(ptr, len) ->
        c_write stdoutFD ptr len
  where
    stdoutFD = 1
```

Builders

What's wrong with the following snippet of code?

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Data.Monoid ((<>))
import Data.Text (Text)

main :: IO ()
main = print ("Hello " <> "there " <> "world" :: Text)
```

When we evaluate `"there " <> "world"`, the text package will:

1. Allocate a new memory buffer capable of holding the combined value
2. Copy the first value into the buffer
3. Copy the second value into the buffer

When we then evaluate `"Hello " <> "there world"`, the text package will do the same thing over again. While combining only thre values this isn't too bad, as the number of values increases this becomes increasingly inefficient. We need to allocate too many memory buffers and recopy the same data multiple times. (The same applies to `ByteString`s.)

Both packages provide a `Builder` datatype, which allows for efficient construction of larger values from smaller pieces. Those coming from the Java world are likely already familiar with the `StringBuilder` class. Let's start off with an example of a text builder:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Data.Monoid ((<>))
import Data.Text.Lazy.Builder (Builder, toLazyText)

main :: IO ()
main = print (toLazyText ("Hello " <> "there " <> "world" :: Builder))
```

Due to memory representation issues, this is one of the corner cases where using a lazy `Text` value makes sense. The reason for this is that, when evaluating a `Buffer`, it is most efficient to allocate a series of buffers, and then represent them as a collection of strict chunk instead of needing to constantly resize just a single buffer.

We can do something similar with bytestring:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Data.Monoid ((<>))
import Data.ByteString.Builder (Builder, toLazyByteString)

main :: IO ()
main = print (toLazyByteString ("Hello " <> "there " <> "world" :: Builder))
```

However, the bytestring builder concept has a lot more flexibility to it. For example, instead of allocating a lazy `ByteString`, you can wo in a fully streaming manner (such as via Data.Streaming.ByteString.Builder):

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Data.Monoid ((<>))
import qualified Data.ByteString as S
import Data.ByteString.Builder (Builder)
import Data.Streaming.ByteString.Builder (toByteStringIO)

main :: IO ()
main = toByteStringIO S.putStr ("Hello " <> "there " <> "world" :: Builder)
```

There are many functions available in both text and bytestring for converting various values into `Builder`s, and in particular with

bytestring you can have very fine-grained control of endianness and other low-level binary format settings.

Text reading

If you're going to be processing large chunks of text, you're best off using a parsing library like attoparsec. However, for many common cases you can use the `Data.Text.Read` module together with some simple `Data.Text` functions, especially by leveraging the `Maybe` type in `do`-notation. As a simple example:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Data.Text (Text)
import qualified Data.Text as T
import qualified Data.Text.Read as TR

alice :: Text
alice = T.unlines
    [ "Name: Alice"
    , "Age: 30"
    , "Score: 5"
    ]

bob :: Text
bob = T.unlines
    [ "Name: Bob"
    , "Age: 25"
    , "Score: -3"
    ]

invalid :: Text
invalid = "blah blah blah"

parsePerson :: Text -> Maybe (Text, Int, Int)
parsePerson t0 = do
    t1 <- T.stripPrefix "Name: " t0
    let (name, t2) = T.break (== '\n') t1
    t3 <- T.stripPrefix "\nAge: " t2
    Right (age, t4) <- Just $ TR.decimal t3
    t5 <- T.stripPrefix "\nScore: " t4
    Right (score, "\n") <- Just $ TR.signed TR.decimal t5
    return (name, age, score)

main :: IO ()
main = do
    print (parsePerson alice)
    print (parsePerson bob)
    print (parsePerson invalid)
```

As you can see, this approach can be far more error-prone than just using a parser, but sometimes the trade-off is worth it.

text-icu

If you're going to be dealing with any significant Unicode topics, like normalization, collation, or others, you should definitely check out th

text-icu package, which provides a binding to the very widely used ICU library. We're not going to go into detail on that package here, simply provide the pointer. For those of you on Windows, the easiest way to install that package's C dependencies is:

```
stack exec -- pacman -Sy mingw64/mingw-w64-x86_64-icu
stack build text-icu
```

ShortByteString?

- Pinned vs unpinned
- Wait till we discuss vector (storable vs unboxed)
- Can be useful... but maybe just used unboxed `Vector Word8`?
- Not nearly as many convenience functions for `ShortByteString`

bytestring basics

Perform some I/O

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import Data.Monoid ((<>))

main :: IO ()
main = do
  let fp = "somefile.txt"
  B.writeFile fp $ "Hello " <> "World!"
  contents <- B.readFile fp
  B.putStr $ B.takeWhile (/= 32) contents <> "\n"
```

Question How are our string literals being treated as `ByteString`s?

Magic numbers like 32 are ugly, `word8` to the rescue!

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import Data.Monoid ((<>))
import Data.Word8 (_space)

main :: IO ()
main = do
  let fp = "somefile.txt"
  B.writeFile fp $ "Hello " <> "World!"
  contents <- B.readFile fp
  B.putStr $ B.takeWhile (/= _space) contents <> "\n"
```

Or assume ASCII directly.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as B8
import Data.Monoid ((<>))

main :: IO ()
main = do
  let fp = "somefile.txt"
  B.writeFile fp $ "Hello " <> "World!"
  contents <- B.readFile fp
  B8.putStrLn $ B8.takeWhile (/= ' ') contents
```

Downsides of the `Char8` modules

- Lots of breakage for non-ASCII data
- Less efficient than working on `Word8` (conversion to/from `Char` can be costly)
- Can be very convenient for some use cases (e.g., HTTP headers)

Questions

- Is our code exception safe?
- Is there any laziness in the code?
- Discuss: to `Char8` or not `Char8` for `Handle` I/O

Printing fibs

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as B8

fibs :: [Int]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fibsBS :: [B.ByteString]
fibsBS = map (B8.pack . show) fibs

main :: IO ()
main = B8.putStr $ B8.unlines $ take 5 fibsBS
```

- Propose alternative implementations
- Discuss: performance and assumptions
- `putStr` vs `putStrLn`

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import qualified Data.ByteString.Builder as BB
import System.IO (stdout)
import Data.Monoid ((<>))

fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

main = BB.hPutBuilder stdout $ foldr
  (\i rest -> BB.intDec i <> "\n" <> rest)
  mempty
  (take 5 fibs)
```

- What does the performance of this look like?
- `foldr` or `foldl`?

Unicode

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import Data.ByteString (ByteString)
import qualified Data.ByteString.Char8 as B8

main :: IO ()
main = do
  let bs = "Non Latin characters: שלום"
  B8.putStrLn bs
  print bs
```

Output:

```
Non Latin characters: ????
"Non Latin characters: \233\220\213\221"
```

- Implicit data loss!
- Terminal doesn't like incorrect character encodings

Laziness and undefined

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as B8
import qualified Data.ByteString.Lazy as BL
import qualified Data.ByteString.Lazy.Char8 as BL8
import UnliftIO.Exception (pureTry)

main :: IO ()
main = do
    let bomb = ['h', 'e', 'l', 'l', 'o', undefined]
    print $ pureTry $ take 5 bomb
    print $ pureTry $ B.take 5 $ B8.pack bomb
    print $ pureTry $ BL.take 5 $ BL8.pack bomb
```

Guess the output!

```
Just "hello"
Nothing
Nothing
```

Let's try again, a little bit bigger.

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import qualified Data.ByteString as B
import qualified Data.ByteString.Char8 as B8
import qualified Data.ByteString.Lazy as BL
import qualified Data.ByteString.Lazy.Char8 as BL8
import UnliftIO.Exception (pureTry)

main :: IO ()
main = do
    let bomb = concat $ replicate 10000 "hello" ++ [undefined]
    print $ pureTry $ take 5 bomb
    print $ pureTry $ B.take 5 $ B8.pack bomb
    print $ pureTry $ BL.take 5 $ BL8.pack bomb
```

Guess the output this time:

```
Just "hello"
Nothing
Just "hello"
```

Exercise: file copy

- Copy `source.txt` to `dest.txt`
- Step 1: make it work
- Step 2: make it work for large files
    - Hint: you'll want `withBinaryFile` and `hGetSome`
    - Use Stackage to search for them

Solution 1

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import qualified Data.ByteString as B

main = B.readFile "source.txt" >>= B.writeFile "dest.txt"
```

- Problem: reads entire file into memory

Solution 2

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
import qualified Data.ByteString as B
import System.IO
import Data.Function (fix)
import Control.Monad (unless)
main =
  withBinaryFile "source.txt" ReadMode $ \hIn ->
  withBinaryFile "dest.txt" WriteMode $ \hOut ->
  fix $ \loop -> do
    bs <- B.hGetSome hIn 4096
    unless (B.null bs) $ do
      B.hPut hOut bs
      loop
```

- Problem: allocates lots of buffers
- Advanced: can use lower-level FFI functions with reused buffer

Exercise

Count how many lines are in a file.

Exercise

Find the largest byte available on standard input.

Exercise

Find the length of the longest line in a file.

Exercise

Write out a file with the line "ABC...Z\n" 1000 times.

text

(Kinda just like bytestring)

Example: dumb CSV parser

(Please actually use a CSV library for this kind of thing.)

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}
import Data.Text (Text)
import qualified Data.Text as T

input :: Text
input =
    "Alice,165cm,30y,15\n\
    \Bob,170cm,35y,-20\n\
    \Charlie,175cm,40y,0\n"

parseRow :: Text -> [Text]
parseRow = T.splitOn ","

main :: IO ()
main = mapM_ print $ map parseRow $ T.lines input
```

Did that feel too easy? Good. Consider quoted fields and embedded commas:

```
"Adams, Adams",165cm,30y,15
"Biggs, Bob",170cm,35y,-20
"Carter, Charlie",175cm,40y,0
```

This is *much* harder to parse with standard text functions, consider it extra credit.

Example: Data.Text.Read for silly format

(Please actually use a real parser library for this kind of thing.)

Consider the following format:

```
Alice 165cm 30y 15
Bob 170cm 35y -20
Charlie 175cm 40y 0
```

We want to parse it to a list of `Person` values:

```haskell
data Person = Person
    { name    :: !Text
    , height  :: !Int
    , age     :: !Int
    , balance :: !Int
    }
```

Give it a shot.

```haskell
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE RecordWildCards #-}
import Data.Text (Text)
import qualified Data.Text as T
import Data.Text.Read (signed, decimal)
import Data.Maybe (mapMaybe)

input :: Text
input =
    "Alice 165cm 30y 15\n\
    \Bob 170cm 35y -20\n\
    \Charlie 175cm 40y 0\n"

data Person = Person
    { name    :: !Text
    , height  :: !Int
    , age     :: !Int
    , balance :: !Int
    }
    deriving Show

parseLine :: Text -> Maybe Person
parseLine t0 = do
    let (name, t1) = T.break (== ' ') t0
    t2 <- T.stripPrefix " " t1
    Right (height, t3) <- Just $ decimal t2
    t4 <- T.stripPrefix "cm " t3
    Right (age, t5) <- Just $ decimal t4
    t6 <- T.stripPrefix "y " t5
    Right (balance, "") <- Just $ signed decimal t6
    Just Person {..}

main :: IO ()
main = mapM_ print $ mapMaybe parseLine $ T.lines input
```

Internal representation

- Irrelevant: you don't need to know what a Text looks like!
- But since you're curious...
- UTF-16 encoded
- Uses unpinned memory
- `data Text = Text !Array !Int !Int`, payload, offset, length
- By contrast, `ByteString` is:
- `data ByteString = PS !(ForeignPtr Word8) !Int !Int`

You cannot pass a `Text` directly to FFI or directly look at its internal data (without diving into deep magic). Consider it fully opaque!

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ForeignFunctionInterface #-}
import Data.Monoid ((<>))
import qualified Data.Text.Encoding as TE
import qualified Data.Text.IO as TIO
import Foreign.Ptr (Ptr)
import Foreign.C.Types (CChar)
import Data.ByteString.Unsafe (unsafeUseAsCStringLen)

foreign import ccall "write"
    c_write :: Int -> Ptr CChar -> Int -> IO ()

main :: IO ()
main = do
    TIO.putStrLn "What is your name?"
    name <- TIO.getLine
    let msg = "Hello, " <> name <> "\n"
        bs = TE.encodeUtf8 msg
    unsafeUseAsCStringLen bs $ \(ptr, len) ->
        c_write stdoutFD ptr len
  where
    stdoutFD = 1
```

Character encoding

- Use Text for in-memory representation
- Use ByteString for serialization and I/O
- Convert with `encodeUtf8`/`decodeUtf8`
    - Oh, `decodeUtf8` is partial :(

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Text.Encoding as TE

main = do
  let text = "This is some text, with non-Latin chars: שלום"
      bs = TE.encodeUtf8 text
  B.writeFile "content.txt" bs
  bs2 <- B.readFile "content.txt"
  let text2 = TE.decodeUtf8 bs2
  TIO.putStrLn text2
```

Total decoding:

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Text.Encoding as TE
import qualified Data.Text.Encoding.Error as TEE

main = do
  let text = "This is some text, with non-Latin chars: שלום"
      bs = TE.encodeUtf8 text
  B.writeFile "content.txt" bs
  bs2 <- B.readFile "content.txt"
  let text2 = TE.decodeUtf8With TEE.lenientDecode bs2
  TIO.putStrLn text2
```

Use or `decodeUtf8'`, which returns an `Either`.

Question What character encoding did `TIO.putStrLn` use?

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import qualified Data.Text as T
import qualified Data.Text.IO as TIO
import qualified Data.Text.Encoding as TE

main = do
    let text = "This is some text, with non-Latin chars: שלום"
        bs = TE.encodeUtf8 text
    B.writeFile "content.txt" bs
    text2 <- TIO.readFile "content.txt"
    TIO.putStrLn text2
```

- Good luck!
- https://www.snoyman.com/blog/2016/12/beware-of-readfile

The char encoding debacle

- File I/O: use bytestring
- Standard handles
    - Interacting with user: use text
    - Interacting with pipe: use bytestring
- Also: text I/O is slow, you can use the `say` package

Exercise

Take a UTF-8 encoded file and generate a UTF-16 encoded file

```
#!/usr/bin/env stack
-- stack --resolver lts-12.21 script
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.ByteString as B
import qualified Data.Text.Encoding as TE

main :: IO ()
main = do
    bsUtf8 <- B.readFile "utf8.txt"
    let text = TE.decodeUtf8 bsUtf8
        bsUtf16 = TE.encodeUtf16LE text
    B.writeFile "utf16.txt" bsUtf16
```

Laziness

- Don't read files lazily
- Writing lazy data isn't actually lazy I/O
- We'll get to streaming data/conduit later

Further reading

https://haskell-lang.org/tutorial/string-types