Stringology and text formatting

Haskell has multiple text string types

- String
  - `[Char]` — huge per-character overhead, everything is boxed
  - each item is an independent Unicode codepoint (incl. surrogates!)
  - for larger bodies of text, String gets impractical

Haskell has multiple text string types

- String
  - `[Char]` — huge per-character overhead, everything is boxed
  - each item is an independent Unicode codepoint (incl. surrogates!)
  - for larger bodies of text, String gets impractical
- ByteString
  - like `char*`, implementation is roughly `Vector Word8`
  - no Unicode!
  - vector speed, input of AttoParsec, can be `mmap`-ed

Haskell has multiple text string types

- String
  - `[Char]` — huge per-character overhead, everything is boxed
  - each item is an independent Unicode codepoint (incl. surrogates!)
  - for larger bodies of text, String gets impractical
- ByteString
  - like `char*`, implementation is roughly `Vector Word8`
  - no Unicode!
  - vector speed, input of AttoParsec, can be `mmap`-ed
- Text
  - Unicode-aware (stores unicode scalar values, i.e., no surrogates!)
  - implemented as an `Array`-slice, stores encoded UTF-8
  - negligible overhead of all string operations

Haskell has multiple text string types

- String
  - `[Char]` — huge per-character overhead, everything is boxed
  - each item is an independent Unicode codepoint (incl. surrogates!)
  - for larger bodies of text, String gets impractical
- ByteString
  - like `char*`, implementation is roughly `Vector Word8`
  - no Unicode!
  - vector speed, input of AttoParsec, can be `mmap`-ed
- Text
  - Unicode-aware (stores unicode scalar values, i.e., no surrogates!)
  - implemented as an `Array`-slice, stores encoded UTF-8
  - negligible overhead of all string operations

IsString provides polymorphic strings

```
{-# LANGUAGE OverloadedStrings #-}
class IsString a where
  fromString :: String → a

"hello" :: IsString s ⇒ s

"hello" :: Text       ⇝ OK
"hello" :: ByteString ⇝ OK
```

ByteString is close to C strings

```
import qualified Data.ByteString as B

B.pack :: [Word8] → ByteString
B.unpack :: ByteString → [Word8]

B.cons :: Word8 → ByteString → ByteString
B.snoc :: ByteString → Word8 → ByteString
B.uncons :: ByteString → Maybe (Word8, ByteString)
B.unsnoc :: ByteString → Maybe (ByteString, Word8)

take, drop, splitAt, takeEnd, split, elem, find, ...
```

Internally, all operations are implemented as index calculations atop a single big buffer. If you want to get rid of the unnecessary buffer space, use *B.copy*.

I/O:

B.getLine :: IO ByteString
B.hGetLine :: Handle → IO ByteString
B.hPutStr :: Handle → ByteString → IO ()
B.hGetContents :: Handle → IO ByteString

B.hGet :: Handle → Int → IO ByteString
B.hPut :: Handle → ByteString → IO ()

etc...

In general, strings are not byte sequences

ByteStrings have to be ***decoded*** in order to get a String or Text. (Note: you can encode Unicode into UTF-8, UTF-16LE, UTF-16BE, UTF-32LE, ..., with losses as Latin1, ISO8859-1, Codepage 1250, KOI-8, ...)

```
import Data.Text.Encoding

decodeUtf8 :: ByteString → Text
decodeUtf16BE :: ByteString → Text
decodeUtf32LE :: ByteString → Text

encodeUtf8 :: Text → ByteString
encodeUtf16LE :: Text → ByteString
```

Text is the usual fast string container

Text is very optimized (almost all functions get fused and/or disappear).

```
import qualified Data.Text as T

T.pack :: String → Text
T.unpack :: Text → String
```

...

(*cons*, *snoc*, *append*, *take*, *takeEnd*, *intercalate*, *toLower*, *justifyLeft*, *splitOn*, ...)

IO-related functionality is in Data.Text.IO

Text is the usual fast string container

Text is very optimized (almost all functions get fused and/or disappear).

```
import qualified Data.Text as T

T.pack :: String → Text
T.unpack :: Text → String
```

...

(*cons*, *snoc*, *append*, *take*, *takeEnd*, *intercalate*, *toLower*, *justifyLeft*, *splitOn*, ...)

IO-related functionality is in Data.Text.IO

String choice depends on use-case

situation	approach
I need speed and have my own code to interpret the data by bytes	ByteString
I use AttoParsec and ignore Unicode	ByteString
I have an infinite word made of undecidable letters	String
String overhead in my application is unconcerning	String
All other cases	Text

Show and Read provide canonical encoding of values

To convert between values and text, we can use *show* and *read*:

- by convention, the text representation should parse as Haskell
- we can serialize values very nicely with just these
- *read* is an actual parser (`Text.ParsingCombinators.ReadP`)
  (and thus survives some typical human-caused fuzz)
- we can format long *show* output with `hindent` to make it readable

Show is not suitable for pretty-printing!

Pretty-printing is formatting for display

Pretty-printing differs from Show-ing in several important aspects:

- the output doesn't need to be parsed (usually)
- the output can contain some redundancy (or hide details)
- the output must be aligned and indented for human eyes

Pretty-printing is formatting for display

Pretty-printing differs from Show-ing in several important aspects:

- the output doesn't need to be parsed (usually)
- the output can contain some redundancy (or hide details)
- the output must be aligned and indented for human eyes

Haskell approach:

```
import Text.PrettyPrint
renderStyle (style {lineLength = 80}) :: Doc → String
```

Pretty-printing is formatting for display

Pretty-printing differs from Show-ing in several important aspects:

- the output doesn't need to be parsed (usually)
- the output can contain some redundancy (or hide details)
- the output must be aligned and indented for human eyes

Haskell approach:

```
import Text.PrettyPrint
renderStyle (style {lineLength = 80}) :: Doc → String
```

Text.PrettyPrint builds output from small document pieces

Doc in an abstraction for various combinations of text fields (similar to TEX vlists/hlists).

empty :: Doc
text :: String → Doc
sizedText :: Int → String → Doc
int :: Int → Doc
float :: Float → Doc

Decorations:

semi, *equals*, *lparen*, *rparen*, *comma*, *colon*, *space*, ... :: Doc

parens, *brackets*, *quotes*, ... :: Doc → Doc

punctuate :: Doc → [Doc] → [Doc]

Glue:

$(\diamond)$ :: Doc → Doc → Doc – next to each other
$(\langle + \rangle)$ :: Doc → Doc → Doc – next to, with (some) space
($$) :: Doc → Doc → Doc – above each other
($+$) :: Doc → Doc → Doc – above with no nesting
nest :: Int → Doc → Doc – indentation

list versions: *hcat*, *hsep*, *vcat*, *vsep*

smart versions (v/h): *cat*, *sep*

paragraph versions: *fcat*, *fsep*

Text.PrettyPrint separates document structure and rendering

```
import Text.PrettyPrint

data Scheme = Ident String | Number Int | Seq [Scheme]

class PDoc a where
  pdoc :: a → Doc

ppshow :: PDoc a ⇒ a → String
ppshow = renderStyle (style {lineLength = 80}) ∘ pdoc

instance PDoc Scheme where
  pdoc (Ident a) = text a
  pdoc (Number i) = int i
  pdoc (Seq xs) = parens $ sep (map pdoc xs)
```

```
short k = Seq $ Ident "*" : map Number [k .. k + 3]
long = Seq $ Ident "+" : map short [1 .. 10]
main = putStrLn . ppshow $ Seq [Ident "factorial", long]
```

```
(factorial
 (+
  (* 1 2 3 4)
  (* 2 3 4 5)
  (* 3 4 5 6)
  (* 4 5 6 7)
  (* 5 6 7 8)
  (* 6 7 8 9)
  (* 7 8 9 10)
  (* 8 9 10 11)
  (* 9 10 11 12)
  (* 10 11 12 13)))
```

$$pdoc\ (\text{Seq}\ [\,]) = parens\ empty$$
$$pdoc\ (\text{Seq}\ (x : xs)) = parens\ \$\ pdoc\ x\ \langle + \rangle\ sep\ (map\ pdoc\ xs)$$

```
(factorial (+ (* 1 2 3 4)
              (* 2 3 4 5)
              (* 3 4 5 6)
              (* 4 5 6 7)
              (* 5 6 7 8)
              (* 6 7 8 9)
              (* 7 8 9 10)
              (* 8 9 10 11)
              (* 9 10 11 12)
              (* 10 11 12 13)))
```

Important side note

Let’s talk about Pandoc.

Pandoc represents markup documents

```
data Pandoc = Pandoc Meta [Block]
newtype Meta = Meta (Map Text MetaValue)

data MetaValue = MetaMap (M.Map Text MetaValue)
               | MetaList [MetaValue]
               | MetaBool Bool
               | MetaString Text
               | MetaInlines [Inline]
               | MetaBlocks [Block]
```

- Attributes: identifier, classes, key-value pairs

```
type Attr = (Text, [Text], [(Text, Text)])
```

Pandoc text consists of inline and block elements

```
data Block
    = Plain [Inline]
    | Para [Inline]
    | LineBlock [[Inline]]
    | CodeBlock Attr Text
    | RawBlock Format Text
    | BlockQuote [Block]
    | OrderedList ListAttributes [[Block]]
    | BulletList [[Block]]
    | DefinitionList [([Inline], [[Block]])]
    | Header Int Attr [Inline]
    | HorizontalRule
    | Table Attr Caption [ColSpec]
            TableHead [TableBody] TableFoot
    | Figure Attr Caption [Block]
    | Div Attr [Block]
```

```
data Inline
    = Str Text
    | Emph [Inline]
    | Underline [Inline]
    | Strong [Inline]
    | Strikeout [Inline]
    | Superscript [Inline]
    | Subscript [Inline]
    | SmallCaps [Inline]
    | Quoted QuoteType [Inline]
    | Cite [Citation] [Inline]
    | Code Attr Text
    | Space
    | SoftBreak
    | LineBreak
    | Math MathType Text
    | RawInline Format Text
    | Link Attr [Inline] Target
    | Image Attr [Inline] Target
    | Note [Block]
    | Span Attr [Inline]
```

Pandoc conversions are “easy” parsers and formatters

```
readMarkdown ::
  (PandocMonad m, ToSources a)
  ⇒ ReaderOptions
  → a
  → m Pandoc

readHtml ::
  (PandocMonad m, ToSources a)
  ⇒ ReaderOptions
  → a
  → m Pandoc
```

```
writeMarkdown ::
  PandocMonad m
  ⇒ WriterOptions
  → Pandoc
  → m Text

writeHtml5String ::
  PandocMonad m
  ⇒ WriterOptions
  → Pandoc
  → m Text
```

...and like 50 other readers and 75 other writers.

Let’s transform some monads

We sometimes want combined functionality

What if we want a monad that does more things at once?

- State (*put*, *get*, ...) + IO (*print*)?
- Parsec + IO?
- IO + Either?
- State + IO + list non-determinism? (a.k.a. prolog)

Software engineering: let's code all combinations!

We sometimes want combined functionality

What if we want a monad that does more things at once?

- State (*put*, *get*, ...) + IO (*print*)?
- Parsec + IO?
- IO + Either?
- State + IO + list non-determinism? (a.k.a. prolog)

Software engineering: let's code all combinations!

That's 2^n combinations of monads!

Some monads can be combined quite mechanically

Can we patch a functionality of one monad atop another?

Let's try IO with a failure semnatic (Maybe + IO):

```
newtype MaybeIO a =
  MaybeIO {runMaybeIO :: IO (Maybe a)}

instance Functor MaybeIO where
  fmap f (MaybeIO m) = MaybeIO $ fmap (fmap f) m
instance Applicative MaybeIO where
  pure = MaybeIO . pure . Just
  MaybeIO a <*> MaybeIO b = MaybeIO $
    a >>= maybe (return Nothing)
                ((<$>b) . fmap)
```

NB.: the applicative instance is slightly asymmetric

```
instance Monad MaybeIO where
  return = pure
  MaybeIO a >>= f = MaybeIO $
    a >>= maybe (return Nothing)
                (runMaybeIO . f)
```

MaybeIO can do IO and failures at the same time

How do we use MaybeIO?

```
runMaybeIO :: MaybeIO a → IO (Maybe a)
```

How to run an IO action?

```
liftIO :: IO a → MaybeIO a
liftIO io = MaybeIO (return ⟨$⟩ io)
```

How to run a Maybe action?

```
stop :: MaybeIO a
stop = MaybeIO (return Nothing)
```

```
io = liftIO
main = runMaybeIO $ do
  io $ putStrLn "give number!"
  a ← io (readLn :: IO Float)
  if a < 0 then stop
    else io $ putStrLn "can be sqrt'd!"
  io . print $ sqrt a
```

(The last line does not print out if there's a problem.)

But wait...

We did not even touch the IO!

```
newtype MaybeIO    a = MaybeIO
                         { runMaybeIO :: IO (Maybe a) }
newtype MaybeT m a = MaybeT
                         { runMaybeT  :: m  (Maybe a) }
```

Transformer is a monad glue-in

```
newtype MaybeT m a =
  MaybeT {runMaybeT :: m (Maybe a)}

instance Functor m ⇒ Functor (MaybeT m) where
  fmap f = MaybeT ∘ fmap (fmap f) ∘ runMaybeT

instance Monad m ⇒ Applicative (MaybeT m) where
  pure = MaybeT ∘ pure ∘ Just
  MaybeT a ⟨∗⟩ MaybeT b = MaybeT $
    a ≫= maybe (return Nothing)
               ((⟨$⟩b) ∘ fmap)
```

```
instance Monad m ⇒ Monad (MaybeT m) where
  return = pure
  MaybeT a >>= f = MaybeT $
    a >>= maybe (return Nothing)
                (runMaybeT . f)

stop :: Monad m ⇒ MaybeT m a
stop = MaybeT (return Nothing)
```

We don’t really need to care which monad is inside

```
class MonadTrans t where
  lift :: Monad m ⇒ m a → t m a

instance MonadTrans MaybeT where
  lift = MaybeT • fmap Just
```

(previously this was *liftIO*)

Maybe-with-IO works

```
main = runMaybeT $ do
  io $ putStrLn "give number!"
  a ← lift $ (readLn :: IO Float)
  if a < 0 then stop
    else lift $ putStrLn "can be sqrt'd!"
  lift . print $ sqrt a
```

Maybe-with-State works too!

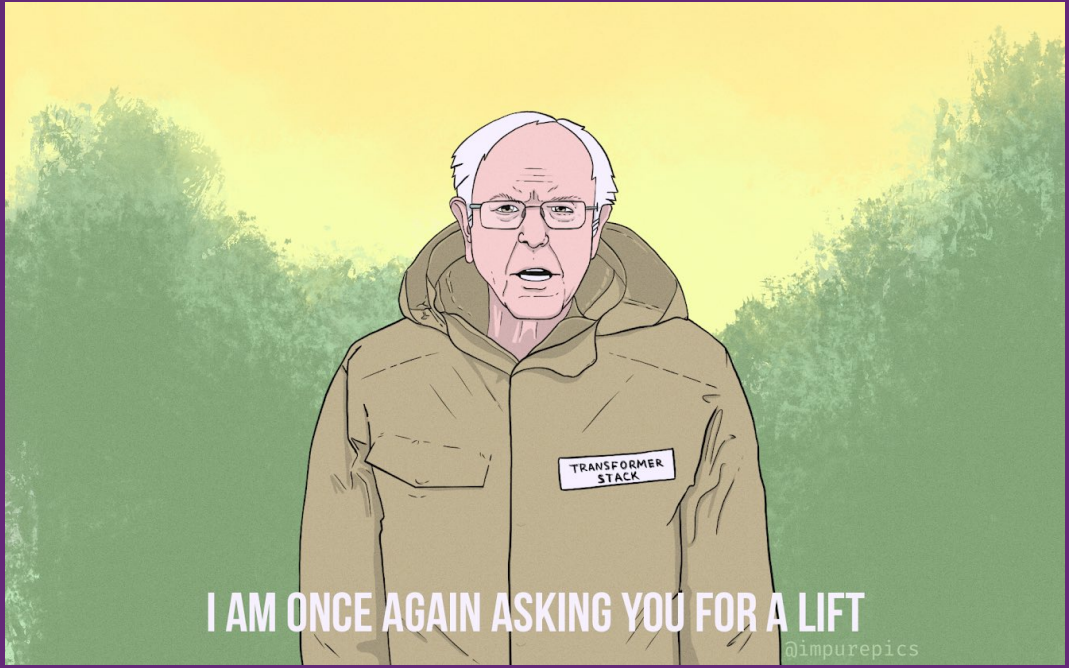
```
type Parser a = MaybeT (State String) a

parseExactChar c = do
  s ← lift get
  case s of
    [ ] → stop
    (c' : cs)
      | c ≡ c' → stop
      | otherwise → do lift $ put cs
                       pure c

runParser :: Parser a → String → Maybe a
runParser p s = execState (runMaybeT p) s
```

Can we apply more transformers at once?

Can we do this with all monads?


TRANSFORMER STACK
I AM ONCE AGAIN ASKING YOU FOR A LIFT
@impurepics

Manual lifting is boring (and error-prone)

```
class Monad m ⇒ MonadIO m where liftIO :: IO a → m a
instance MonadIO IO where          liftIO = id
instance (MonadIO m, MonadTrans t)
         ⇒ MonadIO (t m) where     liftIO = lift . liftIO
```

(The instances recursively lift the IO action until it executes in an actual IO.)

Use in some libraries: *putStrLn* :: MonadIO *m* ⇒ String → *m* ()

There's plenty of other monad transformers

Obviously there is already a StateT, ReaderT, EitherT, ...

Transformer libraries such as `mtl` define typeclasses for easier work within monad stacks:

```
class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
  state :: (s → (a, s)) → m a

class (Monoid w, Monad m)
      ⇒ MonadWriter w m | m → w where
  tell :: w → m ()

class Monad m ⇒ MonadError e m | m → e where
  throwError :: e → m a
  catchError :: m a → (e → m a) → m a
```

Libraries:

- `mtl` — older, uses multi-parameter typeclasses
- `transformers` — newer, standard-compliant, no generic interfaces
- `mtl-tf` — generic interfaces for `transformers` via type families

Recommendation (2023): use `mtl` only for compatibility reasons

There's a transformer for everything

- Identity — doesn't do anything but helps reducing the amount of code:
  SomeMonad = SomeMonadT Identity
- MaybeT, EitherT (Not as great for actual error handling!)
- ExceptT (also ErrorT, but that is deprecated)
- StateT — stateful computation with *get* and *put*
- ReaderT — global environment that you can *ask* for
- WriterT — global monoid that stores what you *tell* it
- RWST — ReaderWriterStateT (quite a typical combo over IO)
- ListT — nondeterminism (but the failures behave weird!)

Reader trivia

Instance for `Monad (Reader r)` is moreless equivalent to `Monad ((->) r)`.

We typically refer to any such parameter broadcasting as Reader-style.

Reader trivia

Instance for $\mathtt{Monad}\ (\mathtt{Reader}\ r)$ is moreless equivalent to $\mathtt{Monad}\ ((\rightarrow)\ r)$.

We typically refer to any such parameter broadcasting as Reader-style.

Human-style: $\lambda x \rightarrow (f\ x, g\ x)$

Reader-style: $\mathit{liftA2}\ (,)\ f\ g$

Side note: There are helper functors too

- Identity
- Const c
- Compose f g — 2 (applicative) functors in one
- Sum, Product
- Reverse — reverses folding and traversing
- Backwards — reverses side-effect order in ($\langle * \rangle$)

How to do backtracking? (like in parsers)

Approach 1: stuff with Alternative instance

Approach 2: MonadPlus

```
class (Alternative m, Monad m) ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

In certain situations *mplus* can behave much more sensibly than ⟨|⟩ thanks to the monad powers.

LogicT provides advanced state space search

```
class MonadPlus m ⇒ MonadLogic m where
  msplit :: m a → m (Maybe (a, m a))    - first result

  ifte :: m a → (a → m b) → m b → m b    - soft cut
  once :: m a → m a    - hard cut

  (≫-) :: m a → (a → m b) → m b    - diagonal conjunction
  interleave :: m a → m a → m a    - diagonal disjunction (Prolog coroutines!)

instance Monad m ⇒ MonadLogic (LogicT m) where ...
```

- (≫=) is like the Prolog comma
- '*mplus*' is like the Prolog semicolon
- *guard* works like with lists

Can we do some useful stuff with continuations?

Continuation-passing style is a method of programming without ‘returning’ values.

- Normal code: $f\ x = \mathbf{do}\ \{r \leftarrow g\ x; return\ (2 * r);\ \}$
- CPS code: $f\ x\ cont = g\ x\ (\lambda r \rightarrow cont\ (2 * r))$
- monadic binding is already quite CPSish: $f \ggg cont$

`Cont` and `ContT` monads are (among other) quite useful for changing control flow.

Use CPS if you want `goto`

```
import Control.Monad
import Control.Monad.Trans.Class
import Control.Monad.Trans.Cont

getCC = callCC (return . fix)

main = evalContT $ do
  lift $ putStrLn "Hello!"
  restart ← getCC
  lift $ putStrLn "Guess a number!"
  a ← lift (readLn :: IO Int)
  unless (a ≡ 42) restart
  lift $ putStrLn "Finally okay!"
```

Use CPS if you hate bracket nesting

```
withFile f = bracket (openFile f readMode) hHclose
muchFileWork = withFile "1.txt" $ λx → withFile "2.txt" $ λy → ...
```

Holding a lot of files at once:

```
muchFileWork = evalContT $ do
  x ← ContT $ withFile "1.txt"
  y ← ContT $ withFile "2.txt"
  z ← ContT $ withFile "3.txt"
  ...
```

...since it's a monad:

```
muchFileWork = evalContT $ do
  handles ← traverse (ContT • withFile) ["1.txt", "2.txt", "3.txt", ...]
  ...
```

Use CPS if you miss imperative `return`

Conditions at the beginning of long monadic actions are annoying:

```
doSomething = do
  x ← parseFile
  case x of
    Nothing → pure False
    Just a → do putStrLn "parsed OK"
                ...
```

– 2 extra levels of nesting

CPS:

```
doSomething = evalContT . callCC $ λfinish → do
  x ← lift parseFile
  a ← case x of Nothing → finish False
                Just a → pure a
  lift $ putStrLn "parsed OK"
  ...
```

– lift is not needed with e.g. ClassyPrelude

Summary

Generic:

$$\mathit{lift} :: (\mathbf{MonadTrans}\ t, \mathbf{Monad}\ m) \Rightarrow m\ a \rightarrow t\ m\ a$$

Specific:

$$\mathit{runMyT} :: \mathbf{Monad}\ m \Rightarrow \mathbf{MyT}\ \mathit{xxx}\ m\ a \rightarrow m\ (\mathit{yyy}\ a)$$
$$\mathit{lift} :: \mathbf{Monad}\ m \Rightarrow m\ a \rightarrow \mathbf{MyT}\ \mathit{xxx}\ m\ a$$

Semantics-driven:

$$\mathbf{instance}\ (\mathbf{MonadMy}\ m, \mathbf{MonadTrans}\ t) \Rightarrow \mathbf{MonadMy}\ (t\ m)$$
$$\mathit{liftMy} :: \mathbf{MonadMy}\ m \Rightarrow \mathbf{My}\ \mathit{xxx}\ a \rightarrow m\ a$$

Extra example: We can make a stateful IO

```
import Control.Monad.Trans.State

type CountingMonad a = StateT Int IO a

doCountdown :: CountingMonad ()
doCountdown = do
  i ← get
  if i > 0 then do put (i – 1)
                   liftIO $ print i
                   doCountdown
  else return ()

main = evalStateT 10 doCountdown
```

Extra example: We can make a stateful IO

```
import Control.Monad.Trans.State

type CountingMonad a = StateT Int IO a

doCountdown :: CountingMonad ()
doCountdown = do
  i ← get
  if i > 0 then do put (i – 1)
                   liftIO $ print i
                   doCountdown
  else return ()

main = evalStateT 10 doCountdown
```

Extra example: We can make a stateful IO

```
import Control.Monad.Trans.State

type CountingMonad a = StateT Int IO a

doCountdown :: CountingMonad ()
doCountdown = do
  i ← get
  if i > 0 then do put (i – 1)
                   liftIO $ print i
                   doCountdown
  else return ()

main = evalStateT 10 doCountdown
```

Extra example: We can make a stateful IO

```
import Control.Monad.Trans.State

type CountingMonad a = StateT Int IO a

doCountdown :: CountingMonad ()
doCountdown = do
  i ← get
  if i > 0 then do put (i – 1)
                   liftIO $ print i
                   doCountdown
  else return ()

main = evalStateT 10 doCountdown
```

Common source of danger: transformation order is relevant!

StateT *s* (ParserT IO) *a*

vs.

ParserT (StateT *s* IO) *a*

Common source of danger: transformation order is relevant!

StateT *s* (ParserT IO) *a*

vs.

ParserT (StateT *s* IO) *a*

⟵ local effects global effects ⟶

Common source of danger: transformation order is relevant!

StateT *s* (ParserT IO) *a*

vs.

ParserT (StateT *s* IO) *a*

⟵ local effects global effects ⟶

Corollary: IOT doesn't really make sense.

Transformation order is relevant!

```
import Control.Applicative
import Control.Monad.Trans.Class
import Control.Monad.Trans.Maybe
import Control.Monad.Trans.State
import Data.Functor.Identity

test1 = runIdentity . runMaybeT . flip runStateT 0
                    $ (put 1 >> empty) <|> pure "ok"
test2 = runIdentity . flip runStateT 0 . runMaybeT
                    $ (lift (put 1) >> empty) <|> pure "ok"

test1  ~> (Just ("ok", 0))
test2  ~> (Just  "ok", 1)
```

Appendix: MPTCs are relations on types

Multiparameter typeclasses may express relations between types:

class X *a b c* | *a b* → *c* **where** ...

Types before the arrow uniquely determine the ones behind the arrow, reducing ambiguity

class MonadState *m s* | *m* → *s*
instance Monad *m* ⇒ MonadState (StateT *s m*) *s*
popGlobalStack :: MonadState *m* [*a*] ⇒ *m* (Maybe *a*)

class KeyValueContainer *c k v* | *c* → *k v*
instance KeyValueContainer (Map Int String) Int String
capitalizeValues :: KeyValueContainer *c k* String ⇒ *c* → *c*

class Convertible *a b*
instance Convertible Int Float
plusAnything :: (Num *a*, Convertible *x a*, Convertible *y a*) ⇒ *x* → *y* → *a*

MPTCs are like Prolog

Haskell:

```
data Z
data S a

class Number a
instance Number Z
instance Number a ⇒ Number (S a)

class Succ a b | a → b
instance Succ a (S a)

class Add a b c | a b → c
instance Add a b c ⇒ Add (S a) b (S c)
instance Add Z b b
```

Prolog:

```
number(z).
number(s(A)) :- number(A).

% succ(+A, -A1)
succ(X, s(X)).

% add(+X, +Y, -Z)
add(z, X, X).
add(s(X), Y, s(Z)) :- add(X, Y, Z).
```

Appendix 2: Open type families are overloadable typelevel functions

```
class Monad m ⇒ MonadState m where
  type StateType m
  get :: m (StateType m)
  put :: StateType m → m ()
```

```
instance Monad m ⇒ MonadState (StateT s m) where
  type StateType (StateT s m) = s
  get = lift get
  put = lift . put
```

```
class KeyValueContainer a where
  type Key a
  type Value a
  keys :: a → [Key a]
  values :: a → [Value a]
```

```
instance KeyValueContainer [(a, b)] where
  type Key [(a, b)] = a
  type Value [(a, b)] = b
  keys = map fst
  values = map snd
```

Appendix 3: Closed type families are normal typelevel functions

```
ghci> :set -XTypeFamilies
ghci> data family X a
ghci> data instance X Int = String
ghci> data instance X Bool = Double
ghci> _ :: X Bool

<interactive>:13:1: error:
    - Found hole: _ :: X Bool
      ...
    - Valid hole fits include
        Double :: X Bool (defined at <interactive>:5:24)
```

Multiparameter type classes and type families are 2 different views of the same typesystem capability.

Bonus: Both can easily encode an undecidable problem.

Final note: Types are values

```
data Z
data S n

data family Succ a
data instance Succ a = S a
  - ...

data Vec / where      - GADT syntax
  Nil :: Vec Z
  Cons :: Int → Vec n → Vec (S n)
```

```
{-# LANGUAGE DataKinds #-}
data Nat = S Nat | Z

succ a = S a

data Vec :: Nat → *
  where
    Nil :: Vec Z
    Cons :: Int → Vec n → Vec (S n)
```

Result: *vec3d* :: Vec (S (S (S Z)))

GHC.TypeLits can help you to get common values (numbers, strings) to type level.