Class 11: Parsing in Haskell

(guest lecture)

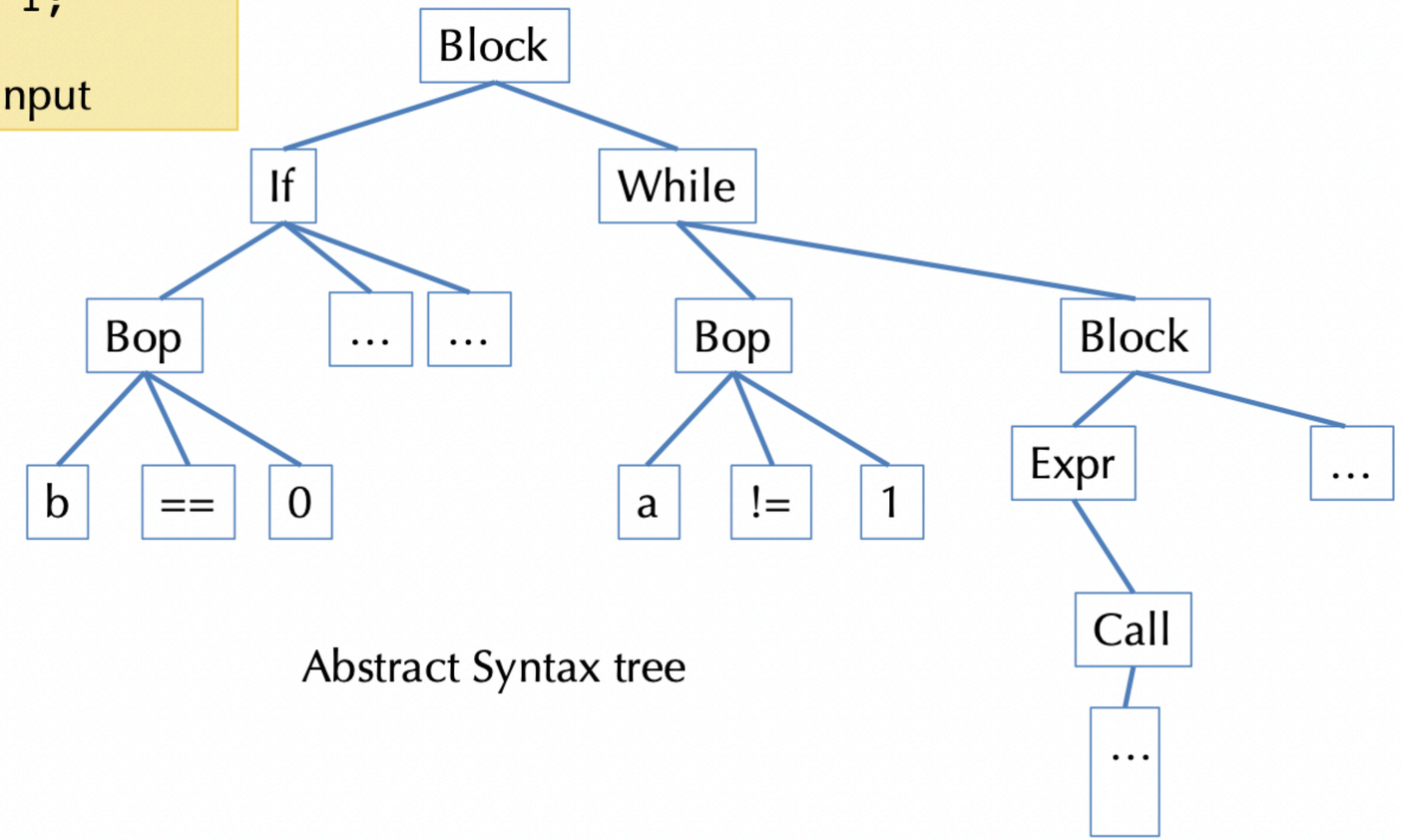
April 4

What is a parser?

`type Parser :: String → StructuredObject`

```
{
  if (b == 0) a = b;
  while (a != 1) {
    print_int(a);
    a = a – 1;
  }
}
```

Source input



Abstract Syntax tree

Diagram by Steve Zdancewic (CIS 3410)

Why do parsers matter?

Parsing is a fundamental aspect of many system programming tasks:

CIS 2400 / 3410: Compilers (parse programs)

CIS 3800: Operating Systems (parse terminal input)

CIS 5530: Networks (parsing packets)

Haskell makes writing parsers simple (& fun!)

How do we define a Parser type in Haskell?

Defining the `Parser` type

A `Parser` is a function:

```
type Parser :: String → StructuredObject
```

A `Parser` doesn't need to consume all of its input:

```
type Parser = String → (StructuredObject, String)
```

remainder of input

A `Parser` should be polymorphic over the type of structured object that it returns:

```
type Parser a = String → (a, String)
```

structured object

A `Parser` should be able to fail (not all strings are parseable!)

```
type Parser a = String → Maybe (a, String)
```

possibility of getting `Nothing`

How do we separate what a `Parser` *is* from what a `Parser` *does*?

we use *records*!

Crash course on Haskell records

Motivation: how do we give *names* to the arguments of data constructors?

Record example

```
data Student = MkStudent {
  name :: String,
  age :: Int
}
```

record type definition

constructor for
the record type

```
data Student = MkStudent {
  name :: String,
  age :: Int
}
```

```
ghci> :t MkStudent
MkStudent :: String → Int → Student
```

name age

Example: defining a record

```
MkStudent :: String → Int → Student
data Student = MkStudent { name :: String, age :: Int }


ernest :: Student
ernest = MkStudent { name = "Ernest", age = 22 }


— alternative syntax (order matters!)
ernest :: Student
ernest = MkStudent "Ernest" 22
```

Each record field defines a *selector*

```
ernest :: Student
ernest = MkStudent { name = "Ernest", age = 22 }
```

```
ghci> name ernest
“ernest”
```

```
name :: Student → String
```

```
ghci> age ernest
22
```

```
age :: Student → Int
```

Defining the `Parser` type

Recall our working type definition for a `Parser`:

```
type Parser a = String → Maybe (a, String)
```

Let's package this into a record

Separating values of type `Parser a` from their parsing functionality:

```
newtype Parser a =
  P { doParse :: String → Maybe (a, String) }
```

actual function that does the parsing

```
ghci> :t P
P :: (String → Maybe (a, String)) → Parser a
```

the `doParse` function

the constructor `P` takes in a function called `doParse` & returns a `Parser`

The `doParse` field in the record defines a record selector:

```
newtype Parser a =
  P { doParse :: String → Maybe (a, String) }
```

```
ghci> :t doParse
doParse :: Parser a → (String → Maybe (a, String))
```

The `doParse` record selector takes a `Parser` & returns the underlying function

Simple parsers

Parsing one single Char

```
get :: Parser Char
get = P $ \s →
  case s of
    (c : cs) → Just (c, cs)
    [] → Nothing
```

the stuff after $ is the `doParse` function associated with this parser

(in Haskell, you can use $ to avoid using parentheses)

```
get :: Parser Char
get = P $ \s →                          s :: String
  case s of
    (c : cs) → Just (c, cs)
    [] → Nothing
```

`P :: (String → Maybe (Char, String)) → Parser Char`

Exercise: parsing one single digit

```
oneDigit :: Parser Int
```

Exercise solution: Parsing a single digit

```
oneDigit :: Parser Int
oneDigit = P $ \s →
  case s of
    (c : cs) → do
      i ← readMaybe [c]
      return (i, cs)
    [] → Nothing
```

Exercise solution: Parsing a single digit

```
oneDigit :: Parser Int
oneDigit = P $ \s →
  case s of
    (c : cs) → do
      i ← readMaybe [c]
      return (i, cs)
    [] → Nothing
```

```
s :: String = [Char]
```

What if instead of `Parsers` for Ints / Chars, we want to build a `Parser` for a function?

Parsing arithmetic operations

```
oneOp :: Parser (Int → Int)
oneOp = P $ \s →
  case s of
    ('-' : cs) → Just (negate, cs)
    ('+' : cs) → Just (id, cs)
    _ → Nothing
```

```
oneOp :: Parser (Int → Int)
oneOp = P $ \s →
  case s of
    ('-' : cs) → Just (negate, cs)
    ('+' : cs) → Just (id, cs)
    _ → Nothing
```

What if we want to build a `Parser` that parses *only* when a condition is satisfied?

Conditional parsing

```
satisfy :: (Char → Bool) → Parser Char
satisfy f = P $ \s → do
  (c, cs) ← doParse get s
  guard (f c)
  return (c, cs)
```

here, we're using the Maybe monad

`Parser` is a monad

Parser is a **Monad**

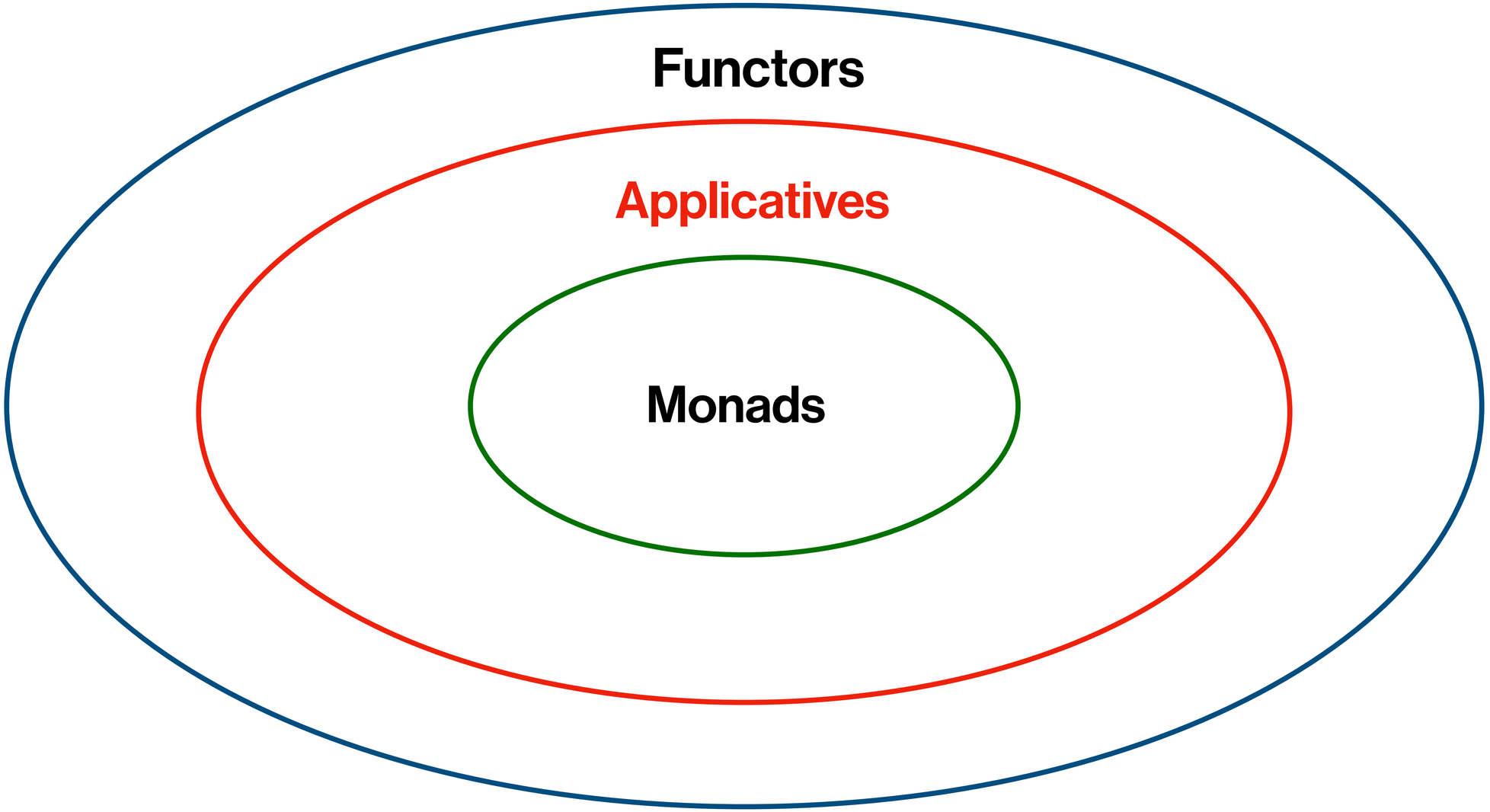
```
instance Monad Parser where
  return :: a → Parser a
  return a = P $ \s → Just (a, s)

  (>>=) :: Parser a → (a → Parser b) → Parser b
  p >>= k = P $ \s → do
    (a, s') ← doParse p s
    doParse (k a) s'
```

As it turns out,
there's a generalisation of Monads that is more useful for parsing!

Detour: `Applicative` functors

(or, how to compose parsers)

Monad ⊆ Applicative ⊆ Functor



Recall the Functor typeclass:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

If you give me:

a "container" of a's

& a function from a → b

I can give you:

a "container" of b's

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

What if the `a → b` function itself is contained inside a Functor?

`Applicative` functors
= “functors with application”

```
class Functor f ⇒ Applicative f where
      pure    :: a → f a
      (<*>)   :: f (a → b) → f a → f b
```

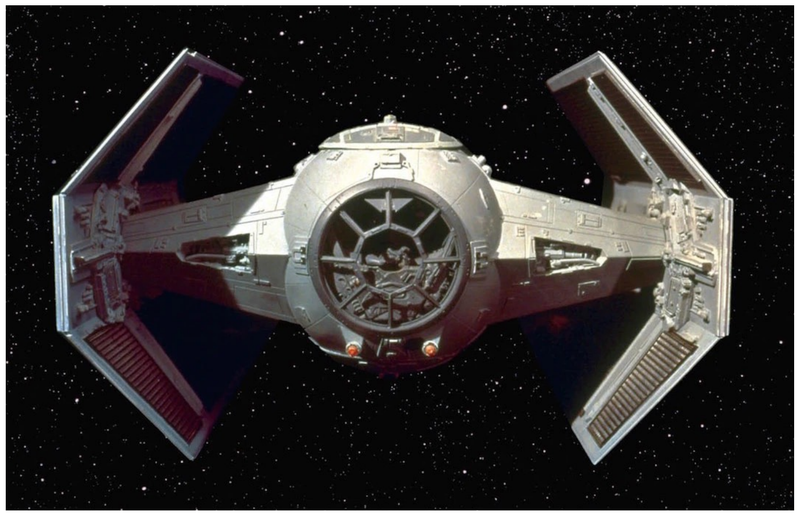
Inject a value into the `Applicative` type

```
pure :: Applicative f ⇒ a → f a
```

analogous to `return` for Monads

Contextual application of functions

```
(<*>) :: Applicative f ⇒ f (a → b) → f a → f b
```



TIE fighter from Star Wars

<*>

“tie fighter”

Source: *Haskell Programming from First Principles*
(Chris Allen & Julie Moronuki)

Parser is an **Applicative**

```haskell
instance Applicative Parser where
  pure :: a → Parser a
  pure x = P $ \s → Just (x, s)

  (<*>) :: Parser (a → b) → Parser a → Parser b
  p1 <*> p2 = P $ \s → do
    (f, s') ← doParse p1 s
    (x, s'') ← doParse p2 s'
    return (f x, s'')
```

Composing parsers

```
(<*>) :: Parser (a → b) → Parser a → Parser b
p1 <*> p2 = P $ \s → do
```

Composing parsers

```
(<*>) :: Parser (a → b) → Parser a → Parser b
p1 <*> p2 = P $ \s → do
  (f, s') ← doParse p1 s
```

1. Use parser `p1` to extract a function `f`

Composing parsers

```
(<*>) :: Parser (a → b) → Parser a → Parser b
p1 <*> p2 = P $ \s → do
  (f, s') ← doParse p1 s
  (x, s'') ← doParse p2 s'
```

1. Use parser `p1` to extract a function `f`
2. Using the remaining input `s'`, use parser `p2` to extract a value `x`

Composing parsers

```
(<*>) :: Parser (a → b) → Parser a → Parser b
p1 <*> p2 = P $ \s → do
  (f, s') ← doParse p1 s
  (x, s'') ← doParse p2 s'
  return (f x, s'')
```

1. Use parser `p1` to extract a function `f`
2. Using the remaining input `s'`, use parser `p2` to extract a value `x`
3. **Apply `f` to `x`, and return `s''` (the new remainder of the input)**

Example: parsing signed digits

```
oneOp :: Parser (Int → Int)
```

parses + and -

```
oneDigit :: Parser Int
```

parses digits

```
signedDigit :: ParserInt
signedDigit = oneOp <*> oneDigit
```

```
(<*>) :: Parser (a → b) → Parser a → Parser b
```

Monad vs Applicative

Monads vs Applicatives

Monads

```
(>>=) :: Parser a → (a → Parser b) → Parser b
```

Applicatives

```
(<*>) :: Parser (a → b) → Parser a → Parser b
```

Monads vs Applicatives

Monads

```
(>>=) :: Parser a → (a → Parser b) → Parser b
```

the result of the 1st parser gets to
influence how the 2nd parser behaves

Applicatives

```
(<*>) :: Parser (a → b) → Parser a → Parser b
```

the results of the 1st parser are not visible

need to declare beforehand how the parsed objects will be combined
(the structure of the computation remains fixed throughout)

Conor McBride, originator of the Applicative typeclass:

"Applicative computations come from the 1960s: you choose all things you're going to compute today and you collect your line-printer output in the morning; **you can't look at the output of one computation and use it to choose a later computation**."

"Monadic computations come from the 1970s: you sit at your teletype, and **you get to see the response to your previous command before you choose your next command**"

(CIS 5520 discussion)

Monads are more powerful than Applicatives,
but we don't need the full monadic structure for parsing!

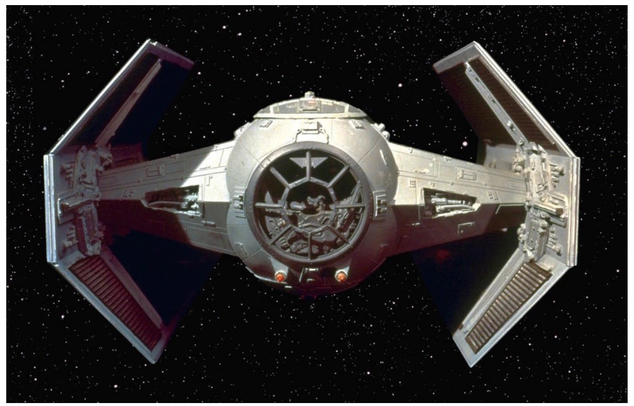
Defining Applicative operators alone is sufficient for parsers*

(in fact, we only give you an Applicative instance for Parser in this week's HW — no Monad instance!)

*technically, parsers for any *context-free grammar*

Applicative operators

`Applicative` operators



TIE fighter from Star Wars

`<*>` “tie-fighter”

`<*` “left tie”

`*>` “right tie”

Source: *Haskell Programming from First Principles*
(Chris Allen & Julie Moronuki)

```
(<*) :: Applicative f ⇒ f a → f b → f a
```

result is discarded

keep the *1st* action's result

```
(*>) :: Applicative f ⇒ f a → f b → f b
```

result is discarded

keep the *2nd* action's result

akin to >> for monads

Exercise: using *> and <* to parse between parentheses

```
parenP :: Parser a → Parser a
```

Example: parsing stuff within (...)

Parser for a specific char

```
char :: Char → Parser Char
```

```
parenP :: Parser a → Parser a
parenP p = char '(' *> p <* char ')'
```

parser for (

original parser

parser for)

Case study:
Translating SQL to Pandas

An automated tool for translating SQL queries into Pandas (joint work with Jason Hom)

https://github.com/homjason/sql-to-pandas

(Heavy usage of Applicative-based parsers for parsing SQL)

SQL

```
SELECT day, COUNT(total_bill) as num_bills
FROM tips_df
WHERE tip > 2
GROUP BY day
```

Pandas

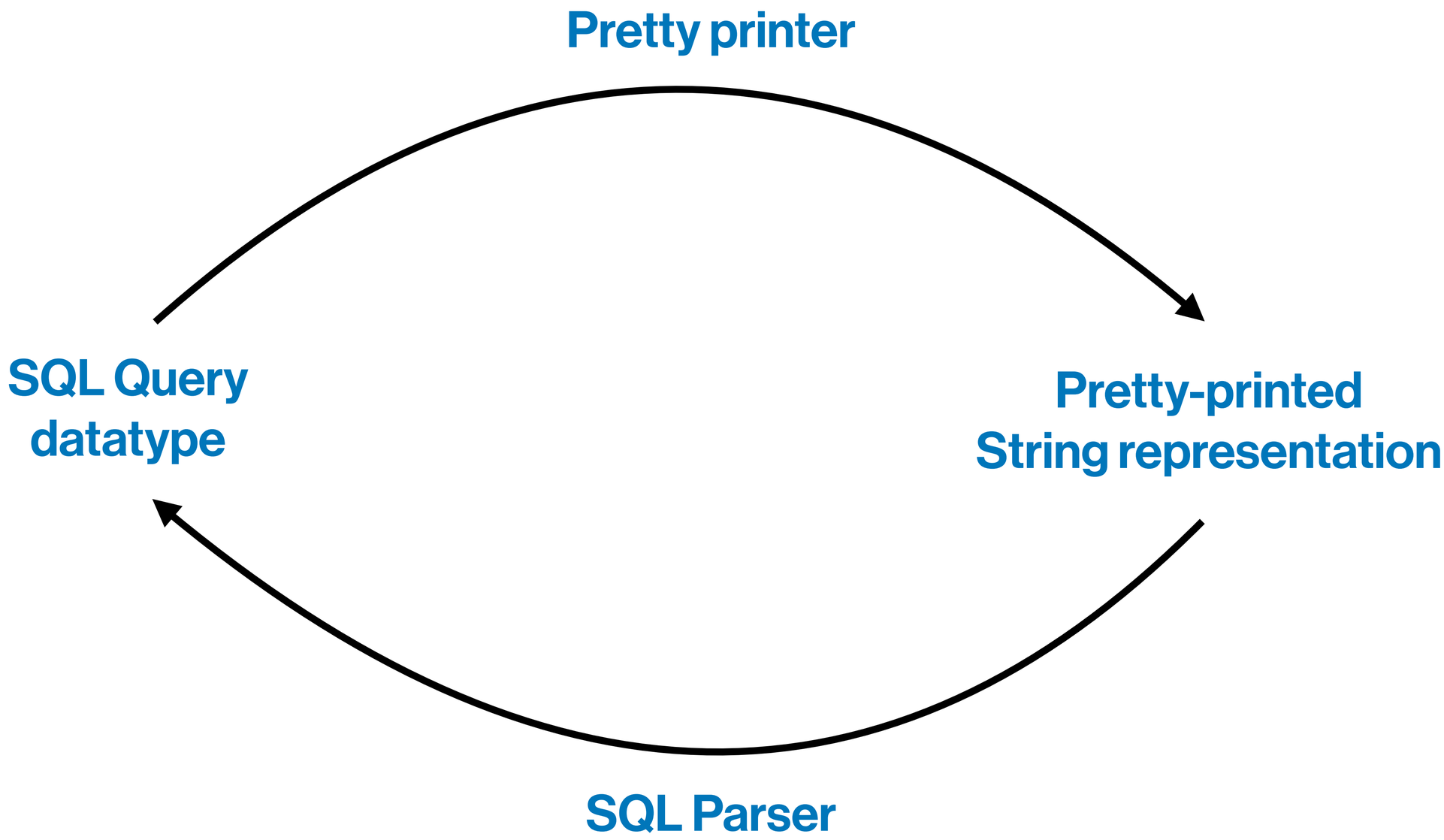
```
tips_df[tips_df["tip"] > 2]
    .groupby(by="day")
    .agg({"total_bill": "size"})
    .reset_index()
    .rename(columns={"total_bill": "num_bills"})
```

Parsing a SQL query

```
queryP :: Parser Query
queryP = Query <$> selectExpP
     <*> fromExpP
     <*> ((Just <$> whereExpP) <|> pure Nothing)
     <*> ((Just <$> groupByP) <|> pure Nothing)
     <*> ((Just <$> orderByP) <|> pure Nothing)
     <*> ((Just <$> limitP) <|> pure Nothing)
```

(you'll see in the HW what the (<|>) operator does)

Example QuickCheck round-trip property



Example QuickCheck round-trip property

```
prop_roundtrip :: Query → Bool
prop_roundtrip query =
  doParse queryParser (printQuery query) == Just (query, "")
```

fin

Lecture adapted from:

- **Stephanie Weirich's lectures for CIS 5520**
- **Brent Yorgey's lectures for CIS 1940 (Spring 2013)**