Parser Design

common pitfalls, and how to avoid them

Announcements

Lab 1: Group registration

- Ask Johan.

Lab 1: `base` == `4.19.2.0`

- The autograder uses this.
- Bump at your own risk.

Werkcollege today: 5cp bonus

- Work on lab 1 or exercises.

Today: last parsing lecture before Lab 1

Previously...

String

e.g.

```
          Groningen  8:37
10:15 Utrecht   10:21
11:05 Den Haag  11:23
11:34 Leiden
```

Previously…

Language

e.g.

> *“the set of all well-formed travel schedules”*

Previously...

Grammar

e.g.

$$
\begin{aligned}
S &\rightarrow Start\ Wait^*\ End \\
Start &\rightarrow StationName\ Time \\
Wait &\rightarrow Time\ StationName\ Time \\
End &\rightarrow Time\ StationName
\end{aligned}
$$

Previously...

Abstract Syntax

e.g.

```
data TSched = TSched
  { start ::          (String, Time)
  , waits :: [(Time, String, Time)]
  , end   ::  (Time, String)
  }
```

Previously...

Parsers

e.g.

```
parseTSched :: String
            → [(TSched, String)]
```

Quiz: parser for travel schedules?

```
        Groningen  8:37
10:15 Utrecht   10:21
11:05 Den Haag  11.23
11:34 Leiden
```

wooclap.com PEQPEH

```
{- 1 -}                 many (pStart <|> pWait <|> pEnd)
{- 2 -} TSched <$> (many (pStart <|> pWait <|> pEnd))
{- 3 -} TSched <$> pStart <*>     pWaits <*> pEnd
{- 4 -} TSched <$> pStart <*> many pWait <*> pEnd
{- 5 -} TSched <$> pStart <$> many pWait <$> pEnd
{- 6 -} TSched <*> pStart <$  many pWait <*  pEnd
```

Not 1

```haskell
many :: ... -> Parser c [a]
```

```haskell
--                      many (pStart <|> pWait <|> pEnd)
{- 2 -} TSched <$> (many (pStart <|> pWait <|> pEnd))
{- 3 -} TSched <$> pStart <*>     pWaits <*> pEnd
{- 4 -} TSched <$> pStart <*> many pWait <*> pEnd
{- 5 -} TSched <$> pStart <$> many pWait <$> pEnd
{- 6 -} TSched <*> pStart <$  many pWait <*  pEnd
```

Not 2

```
TSched :: ... → ... → ... → TSched
many :: ... → Parser c [a]
```

```
--                    many (pStart <|> pWait <|> pEnd)
--       TSched <$> (many (pStart <|> pWait <|> pEnd))
{- 3 -} TSched <$> pStart <*>     pWaits <*> pEnd
{- 4 -} TSched <$> pStart <*> many pWait <*> pEnd
{- 5 -} TSched <$> pStart <$> many pWait <$> pEnd
{- 6 -} TSched <*> pStart <$  many pWait <$  pEnd
```

Yes 3,4

```
TSched :: ... → ... → ... → TSched
many :: ... → Parser c [a]
```

```
--                    many (pStart <|> pWait <|> pEnd)
--       TSched <$> (many (pStart <|> pWait <|> pEnd))
{- C -} TSched <$> pStart <*>     pWaits <*> pEnd
{- D -} TSched <$> pStart <*> many pWait <*> pEnd
{- E -} TSched <$> pStart <$> many pWait <$> pEnd
{- F -} TSched <*> pStart <$  many pWait <$  pEnd
```

Not 5,6

```
TSched :: ... → ... → ... → TSched
infixl (<$>) ::          (a → b) → Parser c a → Parser c b
infixl (<*>) :: Parser c (a → b) → Parser c a → Parser c b
infixl (<$)  ::          a → Parser c b → Parser c a
infixl (<*)  :: Parser c a → Parser c b → Parser c a

--                  many (pTStart <|> pTEnd <|> pTLine)
--       TSched <$> (many (pTStart <|> pTEnd <|> pTLine))
{- 3 -} TSched <$> pStart <*> pTLine <*>     pWaits <*> pEnd
{- 4 -} TSched <$> pStart <*> pTLine <*> many pWait <*> pEnd
--       TSched <$> pStart <$> pTLine <$> many pWait <$> pEnd
--       TSched <*> pStart <$  pTLine <*> many pWait <$  pEnd
```

Combinator zoo

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

Combinator zoo, abbreviated

```
(<$>) ::          (a → b) →         f a →         f b
(<*>) ::        f (a → b) →         f a →         f b
(<$)  ::          a →           f b → f a
(<*)  ::        f a →           f b → f a
```

Combinator zoo, grouped

```
(<$>) ::          (a → b) →        f a →        f b
(<*>) ::        f (a → b) →        f a →        f b
```

```
(<$)  ::          a →          f b → f a
(<*)  ::        f a →          f b → f a
```

Combinator zoo, expanded

```
 ($)  ::          (a → b) →          a →          b
(<$>) ::          (a → b) →        f a →        f b
(<*>) ::        f (a → b) →        f a →        f b
```

```
(<$)  ::          a →        f b → f a
(<*)  ::        f a →        f b → f a
```

Combinator zoo, expanded

```
 ($)  ::          (a → b) →          a →          b
(<$>) ::          (a → b) →        f a →        f b
(<*>) ::        f (a → b) →        f a →        f b
```

```
const  ::          a →          b →   a
 (<$)  ::          a →        f b → f a
 (<*)  ::        f a →        f b → f a
```

Combinator zoo, classified

```
-- Application-like
      ($)  ::          (a → b) →          a →          b
     (<$>) ::          (a → b) →        f a →        f b
     (<*>) ::        f (a → b) →        f a →        f b


-- Const-like
    const  ::          a →          b →   a
     (<$)  ::          a →        f b → f a
     (<*)  ::        f a →        f b → f a
```

More combinators?

```
-- Application-like
      ($)  ::          (a → b) →          a →          b
     (<$>) ::          (a → b) →        f a →        f b
     (<*>) ::        f (a → b) →        f a →        f b
```

More combinators.

```
-- Application-like
      ($)  ::          (a → b) →          a →          b
     (<$>) ::          (a → b) →        f a →        f b
     (<*>) ::        f (a → b) →        f a →        f b
     flap  ::        f (a → b) →          a →        f b
```

More combinators.

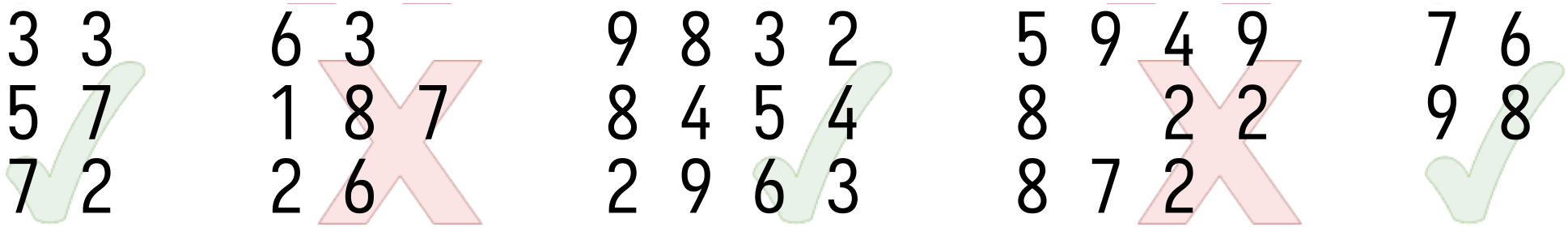
```
-- Application-like
      ($)  ::          (a → b) →          a →          b
     (<$>) ::          (a → b) →        f a →        f b
     (<*>) ::        f (a → b) →        f a →        f b
     flap  ::        f (a → b) →          a →        f b
     (=<<) ::        (a → f b) →        f a →        f b
```

Flipped combinators.

```
-- Application-like
      ($)  ::          (a → b) →          a →          b
     (<$>) ::          (a → b) →        f a →        f b
     (<*>) ::        f (a → b) →        f a →        f b
     flap  ::        f (a → b) →          a →        f b
     (=<<) ::        (a → f b) →        f a →        f b
-- Application-like, arguments flipped
      (&)  ::          a →          (a → b) →          b
     (<&>) ::        f a →          (a → b) →        f b
    (<**>) ::        f a →        f (a → b) →        f b
     (??)  ::          a →        f (a → b) →        f b
     (>>=) ::        f a →        (a → f b) →        f b
```

`(>>=)` allows more flexible parsing

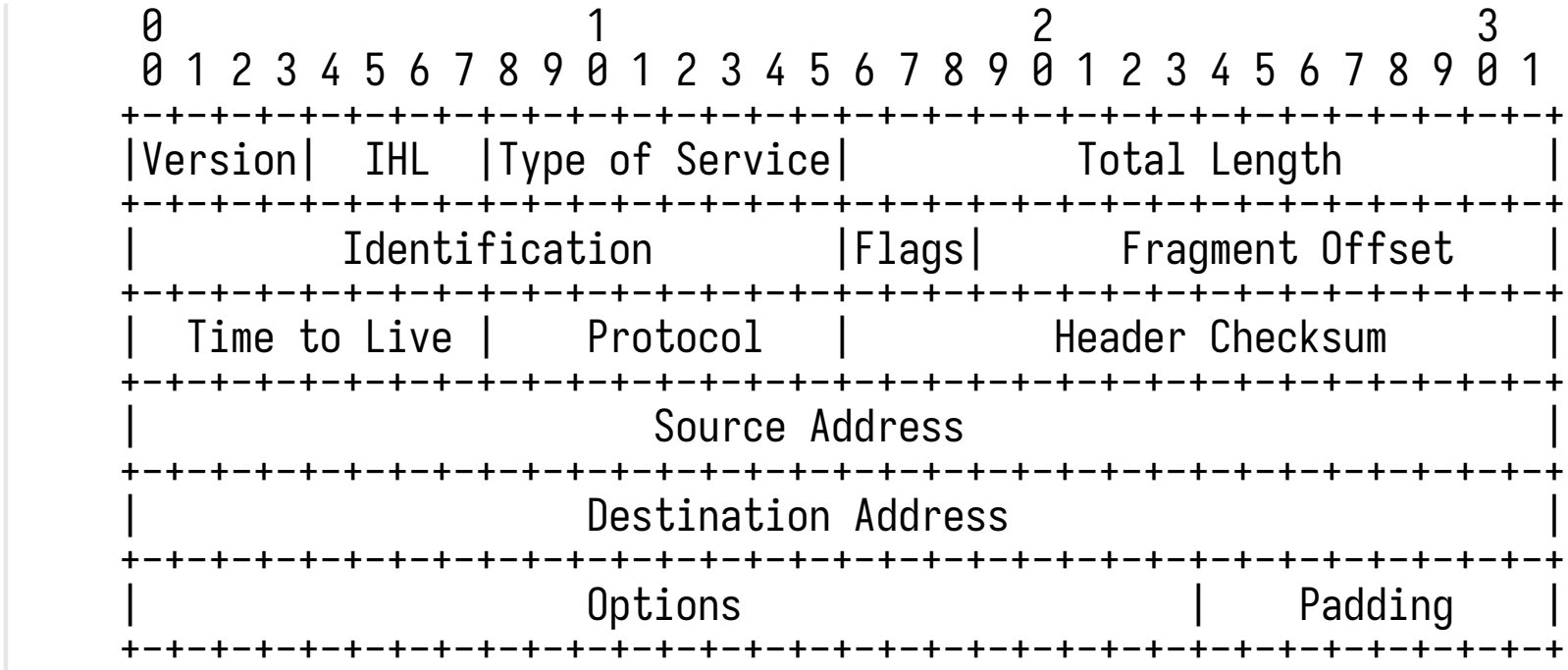
Example: matrices



Numer of columns constant but not known in advance.

```haskell
parseMatrix :: Parser Char [[Int]]
parseMatrix = parseRowOfAnyLength >>= \r →
  (r :) <$> parseRowsOfFixedLength (length r)
```

Example: IPv4 packets



Payload length not known in advance.

```
parseIpPacket : Parser Byte IpPacket
parseIpPacket = parseIpHeader >>= \h ->
  (IpPacket h) <$> parseIpPayload (h .totalLength)
```

Example: nestable markdown divs

```
:::::::::: columns

![](duck.png)

::::: bold
It quacks like a duck...
:::::

::::::::::
```

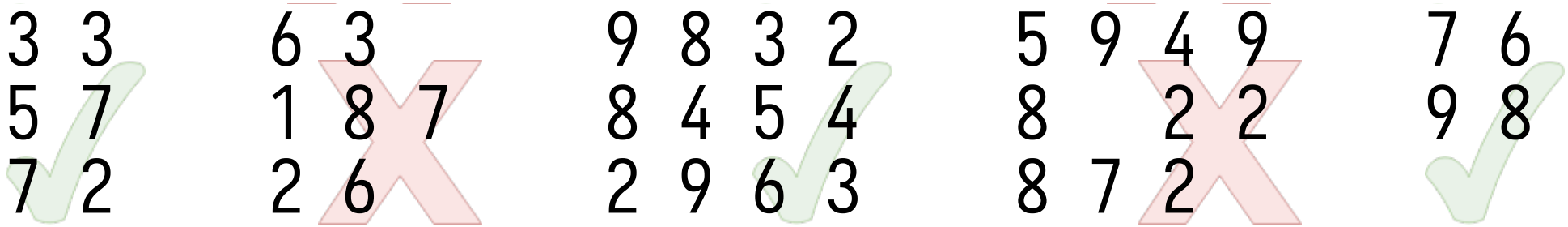
```haskell
parseMarkdown :: Int -> Parser Line MarkdownBlock
parseMarkdown d = text <|> image <|> (parseDivHeaderShorterThan d >>= \h ->
    Div h <$> many (parseMarkdown (h .length))
          <*> parseDivFooter (h .length) )
```

Haskell has special syntax for (>>=)

```haskell
-- You write
do
  a ← foo
  b ← bar
  baz
```

```haskell
-- Equivalent to
foo >>= (\a →
bar >>= (\b →
baz ))
```

Quiz: grammar of well-formed matrices?



1. Matrix → Int**
2. Matrix → Int* Int**
3. Matrix → Int*
4. Matrix → Row Row*
   Row → Int Int*

wooclap.com PEQPEH

Answer: none of the above!

- The matrix language is **not** context-free.
- (>>=) can parse it anyway.
- Sometimes (>>=) can be avoided, e.g.
  - `parseMatrix = many (many integer)`
  - Check row-length later

Back to (>>=)-free parsers

Quiz: parser for wait lines?

> *10:15 Utrecht 10:21*

wooclap.com PEQPEH

```
(,,) :: a → b → c → (a,b,c)
 {- 1 -} (,,) <$> parseTime <$> parseString <$> parseTime
 {- 2 -} (,,) <*  parseTime <$> parseString <$> parseTime
 {- 3 -} (,,) <*> parseTime <$> parseString <$> parseTime
 {- 4 -} (,,) <*> parseTime <$  parseString <$  parseTime
 {- 5 -} (,,) <$> parseTime <*> parseString <*> parseTime
 {- 6 -} (,,) <$> parseTime <*  parseString <*  parseTime
```

“Applicative style”

> *10:15 Utrecht 10:21*

```
(,,) :: a → b → c → (a,b,c)
 --      (,,)  <$> parseTime <$> parseString <$> parseTime
 --      (,,)  <*  parseTime <$> parseString <$> parseTime
 --      (,,)  <*> parseTime <$> parseString <$> parseTime
 --      (,,)  <*> parseTime <$  parseString <$  parseTime
 {- 5 -} (,,)  <$> parseTime <*> parseString <*> parseTime
 --      (,,)  <$> parseTime <*  parseString <*  parseTime
```

Easy enough?

```
"10:15 Utrecht   10:21"
```

```
(,,)  <$> parseTime
      <*> parseString
      <*> parseTime
  :: Parser Char (Time,String,Time)
```

Whitespace trouble

```
"10:15 Utrecht   10:21"
```

```
(,,)  <$> parseTime
      <*> parseString
      <*> parseTime
  :: Parser Char (Time,String,Time)
```

```
(AM 10 15, " Utrecht   ", AM 10 21)
```

Whitespace lexed away

```
["10:15","Utrecht","10:21"]
```

```
(,,)  <$> parseTime
      <*> parseString
      <*> parseTime
  :: Parser String (Time,String,Time)
```

```
(AM 10 15,  "Utrecht"    , AM 10 21)
```

Lexing can be inconvenient

```
["10:15","Den","Haag","10:21"]
```

```
(,,)  <$> parseTime
      <*> parseString
      <*> parseTime
  :: Parser String (Time,String,Time)
```

No parse.

Whitespace parsing

```
"10:15 Utrecht   10:21"
```

```
(,,)  <$> parseTime   <* some space
      <*> parseString <* some space
      <*> parseTime
  :: Parser Char (Time,String,Time)
```

```
(AM 10 15,  "Utrecht"   , AM 10 21)
```

Demo time

wooclap.com PEQPEH

Quiz: what does GHCi return?

```
ghci> parse tline "10:15 Utrecht   10:21"
```

```haskell
import ParseLib.Derived

time :: Parser Char Time
time = AM
    <$> natural <* symbol ':'
    <*> natural

space  = symbol ' '
string = many anySymbol

tline :: Parser Char
          (Time,String,Time)
tline = (,,)
  <$> time    <* some space
  <*> string  <* some space
  <*> time    <* eof
```

Too many results!

```haskell
time :: Parser Char Time
time = AM
    <$> natural <* symbol ':'
    <*> natural

space  = symbol ' '
string = many anySymbol

tline :: Parser Char
          (Time,String,Time)
tline = (,,)
  <$> time    <* some space
  <*> string  <* some space
  <*> time    <* eof
```

wooclap.com PEQPEH

Quiz: what does GHCi return?

```
ghci> parse tline "10:15 Utrecht   10:21"
[((AM 10 15,"Utrecht  ",AM 10 21),"")
,((AM 10 15,"Utrecht ",AM 10 21),"")
,((AM 10 15,"Utrecht",AM 10 21),"")]
```

Quiz: why are too many parses returned?

Definition of some

```haskell
some :: Parser c a → Parser c [a]
some p = (:) <$> p <*> many p

many :: Parser c a → Parser c [a]
many p = some p <|> succeed []
```

Expanding some

```
some space
== (:) <$> space <*> many space
== (:) <$> space <*> (some space
                      <|> succeed [])
== (:) <$> space <*> ((:) <$> space <*> many space
                      <|> succeed [])
== (:) <$> space <*> ((:) <$> space <*> (some space
                                         <|> succeed [])
                      <|> succeed [])
```

Expanding some

```
some space
==  (:) <$> space <*> ((:) <$> space <*> ((:) <$> space <*> some space))
<|> (:) <$> space <*> ((:) <$> space <*> ((:) <$> space <*> succeed []))
<|> (:) <$> space <*> ((:) <$> space <*> succeed [])
<|> (:) <$> space <*> succeed []
```

<|> takes results from both parsers

```
some space
==  (:) <$> space <*> ((:) <$> space <*> ((:) <$> space <*> some space))
<|> (:) <$> space <*> ((:) <$> space <*> ((:) <$> space <*> succeed []))
<|> (:) <$> space <*> ((:) <$> space <*> succeed [])
<|> (:) <$> space <*> succeed []
```

```
(<|>) :: Parser s a → Parser s a → Parser s a
(p1 <|> p2) s = (p1 s) ++ (p2 s)
```

But we only want the first result!

<<|> avoids calling second parser

```
some space
==  (:) <$> space <*> ((:) <$> space <*> ((:) <$> space <*> some space))
<|> (:) <$> space <*> ((:) <$> space <*> ((:) <$> space <*> succeed []))
<|> (:) <$> space <*> ((:) <$> space <*> succeed [])
<|> (:) <$> space <*> succeed []
```

```
(<|>) :: Parser s a -> Parser s a -> Parser s a
(p1 <|> p2) s = (p1 s) ++ (p2 s)
```

```
(<<|>) :: Parser s a -> Parser s a -> Parser s a
(p1 <<|> p2) s = if (p1 s) /= [] then (p1 s) else (p2 s)
```

Definition of greedy,greedy1

```haskell
greedy1 :: Parser c a → Parser c [a]
greedy1 p = (:) <$> p <*> greedy p

greedy :: Parser c a → Parser c [a]
greedy p = greedy1 p <<|> succeed []
```

Quiz: which parsers always return exactly 1 result?

```
{- 1 -} many   space
{- 2 -} greedy space
{- 3 -} (greedy space  <* greedy space)
{- 4 -} (greedy space  <* many space)
{- 5 -} (many   space  <* greedy space)
{- 6 -} (many (greedy space))
{- 7 -} (greedy (greedy space))
{- 8 -} (greedy empty)
{- 9 -} (many empty)
```

wooclap.com PEQPEH

These parsers always return exactly 1 result

```
--       many   space
{- B -} greedy space
{- C -} (greedy space  <* greedy space)
{- D -} (greedy space  <* many space)
--       (many   space  <* greedy space)
--       (many (greedy space))
{- G -} (greedy (greedy space))
--       (greedy empty)
--       (many empty)
```

(many empty) recurses forever

```
(many empty)
== some empty <|> succeed []
```

(many empty) recurses forever

```
(many empty)
== some empty <|> succeed []
== ((:) <$> empty <*> many empty) <|> succeed []
```

(many empty) recurses forever

```
(many empty)
== some empty <|> succeed []
== ((:) <$> empty <*> many empty) <|> succeed []
== (map ((():) (many empty)) <|> succeed []
```

(many empty) recurses forever

```
(many empty)
== some empty <|> succeed []
== ((:) <$> empty <*> many empty) <|> succeed []
== (map ((():)) (many empty)) <|> succeed []
== (map ((():)) (..........)) <|> succeed []
```

(many empty) recurses forever

```
(many empty)
== some empty <|> succeed []
== ((:) <$> empty <*> many empty) <|> succeed []
== (map ((():)) (many empty)) <|> succeed []
== (map ((():)) (..........)) <|> succeed []
```

- Laziness was not enough!

(many empty) recurses forever

```
(many empty)
== some empty <|> succeed []
== ((:) <$> empty <*> many empty) <|> succeed []
== (map ((():)) (many empty)) <|> succeed []
== (map ((():)) (..........)) <|> succeed []
```

- Laziness was not enough!
- Beware left-recursion.

Summary (today)

- `(>>=)`, `(<*>)`, `(<$>)`, `($)`
- `(<*)`, `(<$)`, `const`
- greedy, many
- Excess results
- Left-recursion

How to build parsers

1. Design grammar
2. Write parser
3. Discover bugs
4. Fix parser, transform grammar
5. GOTO 3