Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Talen en Compilers

2019 - 2020, period 2

Jurriaan Hage

Department of Information and Computing Sciences
Utrecht University

2019-11-04

5. Grammar and parser design

This lecture

Grammar and parser design

- Grammar
- Lexing / scanning and parsing
- Semantics
- Loose ends

Example: Travel schemes

	Groningen	8:37
9:44	Zwolle	9:49
10:15	Utrecht	10:21
11:05	Den Haag	

Example: Travel schemes

	Groningen	8:37
9:44	Zwolle	9:49
10:15	Utrecht	10:21
11:05	Den Haag	

Some possible questions for a given travel scheme:

- What is the net travel time?
- What is the total waiting time?
- What is the minimal change time?

Designing a grammar and a parser

- Give some example inputs.
- Construct a grammar from example inputs.
- Test the grammar on the example inputs.
- Analyse the grammar.
- Transform the grammar (if necessary).
- Decide on the types.
- Construct a parser.
- Define semantic functions.
- Check the results.

5.1 Grammar

Step 1: Give some example inputs

```
        Groningen    8:37
 9:44   Zwolle       9:49
10:15   Utrecht     10:21
11:05   Den Haag

        Zwolle
```

Step 2: Constructing a grammar

```
        Groningen    8:37
 9:44   Zwolle       9:49
10:15   Utrecht     10:21
11:05   Den Haag
```

$$
\begin{array}{lll}
\text{TS} & \rightarrow & \text{TS Departure Arrival TS} \\
 & | & \text{Station} \\
\text{Station} & \rightarrow & \text{Identifier}^{+} \\
\text{Departure} & \rightarrow & \text{Time} \\
\text{Arrival} & \rightarrow & \text{Time} \\
\text{Time} & \rightarrow & \text{Nat:Nat}
\end{array}
$$

Step 3: Testing the grammar

```
        Groningen    8:37
 9:44   Zwolle       9:49
10:15   Utrecht     10:21
11:05   Den Haag
```

```
        Zwolle
```

$$
\begin{array}{lll}
\text{TS} & \rightarrow & \text{TS Departure Arrival TS} \\
 & | & \text{Station} \\
\text{Station} & \rightarrow & \text{Identifier}^{+} \\
\text{Departure} & \rightarrow & \text{Time} \\
\text{Arrival} & \rightarrow & \text{Time} \\
\text{Time} & \rightarrow & \text{Nat:Nat}
\end{array}
$$

Step 4: Analysing the grammar

$$
\begin{array}{lll}
\mathsf{TS} & \rightarrow & \mathsf{TS\ Departure\ Arrival\ TS} \\
 & \mid & \mathsf{Station} \\
\mathsf{Station} & \rightarrow & \mathsf{Identifier}^{+} \\
\mathsf{Departure} & \rightarrow & \mathsf{Time} \\
\mathsf{Arrival} & \rightarrow & \mathsf{Time} \\
\mathsf{Time} & \rightarrow & \mathsf{Nat} : \mathsf{Nat}
\end{array}
$$

Step 4: Analysing the grammar

$$
\begin{array}{lll}
\text{TS} & \rightarrow & \text{TS Departure Arrival TS} \\
 & | & \text{Station} \\
\text{Station} & \rightarrow & \text{Identifier}^{+} \\
\text{Departure} & \rightarrow & \text{Time} \\
\text{Arrival} & \rightarrow & \text{Time} \\
\text{Time} & \rightarrow & \text{Nat}:\text{Nat}
\end{array}
$$

Observations

- The grammar is not explicit about the use of whitespace.
- A single station is a valid travel scheme according to the grammar.
- The grammar for times is imprecise.
- The grammar is left-recursive.

Another grammar for the language

$$
\begin{aligned}
\text{TS} \rightarrow\ & \text{Station Departure} \\
& (\text{Arrival Station Departure})^* \\
& \text{Arrival Station} \\
\mid\ & \text{Station}
\end{aligned}
$$

Another grammar for the language

$$
\begin{aligned}
\mathrm{TS} \rightarrow\ & \text{Station Departure} \\
& (\text{Arrival Station Departure})^* \\
& \text{Arrival Station} \\
\mid\ & \text{Station}
\end{aligned}
$$

- Has a different focus.
- Not left-recursive, but can be left-factored.

Step 5: Transforming the grammar

$$\begin{aligned} TS \rightarrow\ & TS\ \text{Departure Arrival}\ TS \\ \mid\ & \text{Station} \end{aligned}$$

Step 5: Transforming the grammar

$$\begin{aligned} TS &\rightarrow TS\ \text{Departure Arrival}\ TS \\ &\mid \text{Station} \end{aligned}$$

The symbols Departure Arrival are an associative separator:

$$\begin{aligned} TS &\rightarrow \text{Station Departure Arrival}\ TS \\ &\mid \text{Station} \end{aligned}$$

Step 5: Transforming the grammar

$$\begin{aligned} TS &\rightarrow TS\ \text{Departure Arrival}\ TS \\ &\mid \text{Station} \end{aligned}$$

The symbols Departure Arrival are an associative separator:

$$\begin{aligned} TS &\rightarrow \text{Station Departure Arrival}\ TS \\ &\mid \text{Station} \end{aligned}$$

Simplification:

$$TS \rightarrow (\text{Station Departure Arrival})^{*}\ \text{Station}$$

Step 5: Transforming the grammar

$$\begin{aligned} \text{TS} &\rightarrow \text{TS Departure Arrival TS} \\ &\mid \text{Station} \end{aligned}$$

The symbols Departure Arrival are an associative separator:

$$\begin{aligned} \text{TS} &\rightarrow \text{Station Departure Arrival TS} \\ &\mid \text{Station} \end{aligned}$$

Simplification:

$$\text{TS} \rightarrow (\text{Station Departure Arrival})^* \text{ Station}$$

Abstraction:

$$\begin{aligned} \text{TS} &\rightarrow \text{Leg}^* \text{ Station} \\ \text{Leg} &\rightarrow \text{Station Departure Arrival} \end{aligned}$$

Step 6: Deciding on the types

Which grammar do we use as a basis for the abstract syntax?

Which grammar do we use as a basis for the parser?

Step 6: Deciding on the types

Which grammar do we use as a basis for the abstract syntax?

Which grammar do we use as a basis for the parser?

- No need to use the same grammar for both purposes.
- Main criteria for abstract syntax:
  - Readability
  - Appropriate for semantic functions
- Main criteria for parser:
  - Termination (e.g. no left-recursion)
  - Efficiency (e.g. left-factored)

Deciding on the type – contd.

In our case, the grammar

> TS $\rightarrow$ Leg* Station
> Leg $\rightarrow$ Station Departure Arrival

is suitable for both the abstract syntax and the parser.

Deciding on the type – contd.

In our case, the grammar

> TS $\rightarrow$ Leg* Station
> Leg $\rightarrow$ Station Departure Arrival

is suitable for both the abstract syntax and the parser.

```
data TS        = TS  [Leg] Station
data Leg       = Leg Station Departure Arrival
data Time      = Time Hours Minutes

type Station   = String
type Departure = Time
type Arrival   = Time
type Hours     = Int
type Minutes   = Int
```

5.2 Lexing / scanning and parsing

Dealing with whitespace

In the grammar, we left the handling of spaces implicit.

Intuitively, any two **tokens** (semantically connected sequences of characters) can be separated by spaces.

Dealing with whitespace

In the grammar, we left the handling of spaces implicit.

Intuitively, any two **tokens** (semantically connected sequences of characters) can be separated by spaces.

There are (at least) three possiblities to deal with spaces:

- write a scanner by hand that produces tokens,
- construct a scanner using parser combinators that produces tokens,
- deal with spaces in the parser itself.

We have a look at the latter two options.

A parser for whitespace

First attempt:

```
spaces :: Parser Char String
spaces = many (satisfy isSpace)
```

A parser for whitespace

First attempt:

```
spaces :: Parser Char String
spaces = many (satisfy isSpace)
```

Question

What about

```
moreSpaces :: Parser Char String
moreSpaces = (++) <$> spaces <*> spaces
```

?

Observations regarding whitespace

- Sequential uses of spaces lead to ambiguity.
- To prevent this, it is best to handle whitespace systematically.
- Often, the whitespace itself is not significant, and we want to discard **all** whitespace in a particular place.
- In other words, we are not interested in the partial results returned by many.

Whitespace policy

We can systematically handle whitespace if we agree that

- Whenever a parser consumes a complete token from the input, it is responsible for consuming all subsequent whitespace.
- The parser for the start symbol starts by consuming initial whitespace.
- Whitespace is not consumed anywhere else.

Greedy parsers

In order to avoid unused partial results, we define a primitive parser combinator for "greedy" choice:

```
(<<|>) :: Parser s a → Parser s a → Parser s a
Parser p <<|> Parser q = Parser (λxs → let r = p xs
                                       in if null r
                                          then q xs
                                          else  r)
```

Greedy parsers

In order to avoid unused partial results, we define a primitive parser combinator for "greedy" choice:

```
(<<|>) :: Parser s a → Parser s a → Parser s a
Parser p <<|> Parser q = Parser (λxs → let r = p xs
                                       in if null r
                                          then q xs
                                          else r)
```

We can then define greedy versions of many and some:

```
greedy, greedy1 :: Parser s a → Parser s [a]
greedy  p = (:) <$> p <*> greedy p <<|> succeed []
greedy1 p = (:) <$> p <*> greedy p
```

Parsing spaces greedily

```
spaces :: Parser Char String
spaces = greedy (satisfy isSpace)
```

Parsing spaces greedily

```
spaces :: Parser Char String
spaces = greedy (satisfy isSpace)
```

Now

```
moreSpaces :: Parser Char String
moreSpaces = (++) <$> spaces <*> spaces
```

is not even all that problematic.

Parsing spaces greedily

```
spaces :: Parser Char String
spaces = greedy (satisfy isSpace)
```

Now

```
moreSpaces :: Parser Char String
moreSpaces = (++) <$> spaces <*> spaces
```

is not even all that problematic.

In general, use of greedy can improve parser efficiency. But careful in cases where backtracing may be needed!

Step 7: A direct parser for travel schemes

We have two kinds of tokens that occur in travel schemes:

- station names,
- times.

Step 7: A direct parser for travel schemes

We have two kinds of tokens that occur in travel schemes:

- station names,
- times.

Let us first write parsers for these.

Grammar:

> Station $\rightarrow$ Identifier$^+$

Parser:

```
station :: Parser Char Station
station = unwords <$> greedy1 (identifier <* spaces)
```

Note how we consume spaces **at the end**.

Tokens in travel schemes

$$\text{Time} \to \text{Nat} : \text{Nat}$$

Haskell:

```
data Time    = Time Hours Minutes
type Hours   = Int
type Minutes = Int
```

Parser:

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

We choose not to allow spaces between the numbers and the colon.

A direct parser for travel schemes

$$\begin{aligned} \text{TS} &\rightarrow \text{Leg}^* \ \text{Station} \\ \text{Leg} &\rightarrow \text{Station Departure Arrival} \end{aligned}$$

Haskell:

```
data TS  = TS  [Leg] Station
data Leg = Leg Station Departure Arrival
```

A direct parser for travel schemes

$$\begin{aligned} \text{TS} &\rightarrow \text{Leg}^* \ \text{Station} \\ \text{Leg} &\rightarrow \text{Station Departure Arrival} \end{aligned}$$

Haskell:

```
data TS  = TS  [Leg] Station
data Leg = Leg Station Departure Arrival
```

Parser:

```
ts :: Parser Char TS
ts = TS <$> many leg <*> station

leg :: Parser Char Leg
leg = Leg <$> station <*> time <*> time
```

A direct parser for travel schemes

$$\begin{aligned} \text{TS} &\rightarrow \text{Leg}^* \ \text{Station} \\ \text{Leg} &\rightarrow \text{Station Departure Arrival} \end{aligned}$$

Haskell:

```
data TS  = TS  [Leg] Station
data Leg = Leg Station Departure Arrival
```

Parser:

```
ts :: Parser Char TS
ts = TS <$> many leg <*> station

leg :: Parser Char Leg
leg = Leg <$> station <*> time <*> time
```

No space handling required!

Running the parser

As agreed, we parse initial spaces in the top-level parser:

```
start :: Parser Char TS
start = spaces *> ts <* eof
```

We can now run parse start on an input string and get a result.

Using an explicit scanner

If we want to define a separate scanner, we start by defining a datatype for Tokens:

```
data Token = TStation Station
           | TTime    Time
```

Using an explicit scanner

If we want to define a separate scanner, we start by defining a datatype for Tokens:

```
data Token = TStation Station
           | TTime    Time
```

We adapt the parsers for stations and times to produce tokens:

```
tstation :: Parser Char Token
tstation = TStation . unwords <$> greedy1 (identifier <* spaces)

ttime :: Parser Char Token
ttime =
  TTime <$>
  (Time <$> natural <* symbol ':' <*> natural <* spaces)
```

Using an explicit scanner – contd.

The scanner parses any number of tokens:

```
anyToken :: Parser Char Token
anyToken = tstation <|> ttime

scan :: Parser Char [Token]
scan = spaces *> greedy anyToken <* eof
```

In the subsequent parsing phase, we then work with Token as the type of symbols.

Parsing a particular type of tokens

```
station :: Parser Token Station
station = fromStation <$> satisfy isStation

isStation :: Token → Bool
isStation (TStation _) = True
isStation _            = False

fromStation :: Token → Station
fromStation (TStation x) = x
fromStation _            = error "fromStation"
```

Parsing a particular type of tokens

```
station :: Parser Token Station
station = fromStation <$> satisfy isStation

isStation :: Token → Bool
isStation (TStation _) = True
isStation _            = False

fromStation :: Token → Station
fromStation (TStation x) = x
fromStation _            = error "fromStation"
```

Similarly:

```
time :: Parser Token Time
time = fromTime <$> satisfy isTime
```

A token parser for travel schemes

The rest of the parser is unchanged apart from the symbol type:

```
ts :: Parser Token TS
ts = TS <$> many leg <*> station

leg :: Parser Token Leg
leg = Leg <$> station <*> time <*> time
```

Use a scanner or not?

Whether to use a scanner or not is mostly a matter of taste.

Use a scanner or not?

Whether to use a scanner or not is mostly a matter of taste.

Advantages

- Easier to keep whitespace handling localized.
- Easier to maintain decent efficiency.
- Easier to produce good error messages.

Use a scanner or not?

Whether to use a scanner or not is mostly a matter of taste.

Advantages

- Easier to keep whitespace handling localized.
- Easier to maintain decent efficiency.
- Easier to produce good error messages.

Disadvantages

- Extra work required.
- All of the advantages can also be gained without a separate scanner.

5.3 Semantics

Step 8: Adding semantic functions

On the abstract syntax, we can define semantic functions.

Semantic functions typically follow the datatypes closely:

```
data TS      = TS  [Leg] Station
data Leg     = Leg Station Departure Arrival
data Time    = Time Hours Minutes

type Station   = String
type Departure = Time
type Arrival   = Time
type Hours     = Int
type Minutes   = Int
```

Net travel time

```
netTravelTimeTS :: TS → Minutes
netTravelTimeTS (TS ls _) = sum (map netTravelTimeLeg ls)

netTravelTimeLeg :: Leg → Minutes
netTravelTimeLeg (Leg _ dep arr) = arr `timeDiff` dep
```

Net travel time

```
netTravelTimeTS :: TS → Minutes
netTravelTimeTS (TS ls _) = sum (map netTravelTimeLeg ls)

netTravelTimeLeg :: Leg → Minutes
netTravelTimeLeg (Leg _ dep arr) = arr `timeDiff` dep
```

The rest is dealing with times.

Excursion: Time difference

We assume that no two times are more than a day apart:

$$
\begin{aligned}
&\mathsf{timeDiff} :: \mathsf{Time} \to \mathsf{Time} \to \mathsf{Minutes} \\
&\mathsf{timeDiff}\ (\mathsf{Time}\ h_1\ m_1)\ (\mathsf{Time}\ h_2\ m_2) \\
&\quad \mid h_2 > h_1 \vee (h_2 == h_1 \wedge m_2 > m_1) \\
&\qquad\qquad = (h_2 \quad\ - h_1) * 60 + m_2 - m_1 \\
&\quad \mid \mathsf{otherwise} = (h_2 + 24 - h_1) * 60 + m_2 - m_1
\end{aligned}
$$

Waiting time

```
waitingTimeTS :: TS → Minutes
waitingTimeTS (TS ls _) = waitingTimeLegs ls

waitingTimeLegs :: [Leg] → Minutes
waitingTimeLegs (Leg _ _ arr : ls@(Leg _ dep _ : _)) =
   dep `timeDiff` arr + waitingTimeLegs ls
waitingTimeLegs _ = 0
```

Waiting time

```
waitingTimeTS :: TS → Minutes
waitingTimeTS (TS ls _) = waitingTimeLegs ls

waitingTimeLegs :: [Leg] → Minutes
waitingTimeLegs (Leg _ _ arr : ls@(Leg _ dep _ : _)) =
   dep `timeDiff` arr + waitingTimeLegs ls
waitingTimeLegs _ = 0
```

Minimal waiting time is a variation of this theme: first compute a list of waiting times per station, then compute the minimum thereof.

Eliminating the abstract syntax tree

If we are only interested in one particular semantic function, there is no need to compute an abstract syntax tree first – we can plug the semantic function directly into the parser.

Eliminating the abstract syntax tree

If we are only interested in one particular semantic function, there is no need to compute an abstract syntax tree first – we can plug the semantic function directly into the parser.

For net travel time:

```
ts :: Parser Token Minutes
ts = sum <$> many leg <* station

leg :: Parser Token Minutes
leg = flip timeDiff <$ station <*> time <*> time
```

Step 9: Checking and improving

Of course, we should test the parser on lots of examples, and make adaptions if necessary.

5.4 Loose ends

Parsing times

Both syntax and parser for times are too liberal.

> $\mathsf{Time} \to \mathsf{Nat} : \mathsf{Nat}$

Parsing times

Both syntax and parser for times are too liberal.

> $\mathsf{Time} \to \mathsf{Nat} : \mathsf{Nat}$

Two options:

- Adapt the grammar, and hence the parser – morally correct, but tedious.
- A more pragmatic solution: first parse liberally, then check and perhaps reject afterwards.

Another sequencing combinator

We cannot reject a result using $(<\$>)$:

```
(<$>) :: (a → b) → Parser s a → Parser s b
```

Does not work:

```
hours :: Parser Char Hours
hours = (λx → if x < 24 then ... else ... ) <$> natural
```

Another sequencing combinator

We cannot reject a result using $(<\$>)$:

```
(<$>) :: (a → b) → Parser s a → Parser s b
```

Does not work:

```
hours :: Parser Char Hours
hours = (λx → if x < 24 then ... else ... ) <$> natural
```

What if we could build a new parser based on a previous result?

Another sequencing combinator – contd.

We introduce $(\gg\!=)$ – pronounced "bind". This is another **primitive** parser combinator:

```
(>>=) :: Parser s a -> (a -> Parser s b) -> Parser s b
Parser p >>= f =
   Parser (\xs -> [(s, zs) | (r, ys) <- p xs,
                             (s, zs) <- runParser (f r) ys])
```

Another sequencing combinator – contd.

We introduce $(\gg\!=)$ – pronounced "bind". This is another **primitive** parser combinator:

```
(>>=) :: Parser s a → (a → Parser s b) → Parser s b
Parser p >>= f =
   Parser (λxs → [(s, zs) | (r, ys) ← p xs,
                            (s, zs) ← runParser (f r) ys])
```

Now:

```
hours :: Parser Char Hours
hours = natural >>= λx →
          if x < 24 then succeed x else empty
```

Using bind

Question

What is the difference between token and keyword?

```
token :: String → Parser Char String
token k xs | k == take n xs = [(k, drop n xs)]
           | otherwise      = []
  where n = length k

keyword :: String → Parser Char String
keyword xs = greedy isLetter >>= λys →
               if xs == ys then succeed ys else empty
```

Using bind

Question

What is the difference between token and keyword?

```
token :: String → Parser Char String
token k xs | k == take n xs = [(k, drop n xs)]
           | otherwise      = []
  where n = length k

keyword :: String → Parser Char String
keyword xs = greedy isLetter >>= λys →
               if xs == ys then succeed ys else empty
```

Hint

Consider p `"let"` *> spaces *> identifier (where p is token or keyword) and the input string `"letx"`.

Applicative functors

The operations parsers support are very common – many other types support the same interface(s):

```
class Functor f where
  fmap  :: (a → b) → f a → f b
(<$>) = fmap
class Functor f ⇒ Applicative f where
  pure  :: a → f a
  (<*>) :: f (a → b) → f a → f b
class Applicative f ⇒ Alternative f where
  empty :: f a
  (<|>) :: f a → f a → f a
```

Monads

```
class Monad m where
  (>>=) :: m a → (a → m b) → m b
```

In contrast to applicative functors, you have probably seen monads before.

Monads

```
class Monad m where
  (>>=) :: m a → (a → m b) → m b
```

In contrast to applicative functors, you have probably seen monads before.

More about applicative functors and monads in the master course on **Advanced Functional Programming**.

A common pitfall

Question

What happens here?

```
many (option (symbol 'x') ' ')
```

A common pitfall

Question

What happens here?

```
many (option (symbol 'x') ' ')
```

In general, be very careful not to call many on anything that can succeed on the empty string.

A common pitfall

Question

What happens here?

```
many (option (symbol 'x') ' ')
```

In general, be very careful not to call many on anything that can succeed on the empty string.

In particular, many (many p) will always go wrong.