Talen en Compilers

2010/2011, periode 2

Johan Jeuring

Department of Information and Computing Sciences
Utrecht University

November 24, 2010

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 questions to ask 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.
- Analyze the grammar.
- Possibly transform the grammar.
- Decide on the types.
- Construct a basic 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
```

$$
\begin{array}{lcl}
\mathsf{TS} & \to & \mathsf{TS\ Departure\ Arrival\ TS} \\
 & | & \mathsf{Station} \\
\mathsf{Station} & \to & \mathsf{Identifier}^{+} \\
\mathsf{Departure} & \to & \mathsf{Time} \\
\mathsf{Arrival} & \to & \mathsf{Time} \\
\mathsf{Time} & \to & \mathsf{Nat : Nat}
\end{array}
$$

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Step 3: Testing the grammar

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


        Zwolle
```

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

Step 4: Analyzing the grammar

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

Step 4: Analyzing the grammar

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

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

> TS → Station Departure
> (Arrival Station Departure)*
> Arrival Station
> | Station

Another grammar for the language

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

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

Step 5: Transforming the grammar

> $\text{TS} \rightarrow \text{TS Departure Arrival TS}$
> $\quad\quad\;\mid\; \text{Station}$

Step 5: Transforming the grammar

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

The symbols Departure Arrival are an associative separator:

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

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:

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

Step 5: Transforming the grammar

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

The symbols Departure Arrival are an associative separator:

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

Simplification:

$$TS \rightarrow (Station\ Departure\ Arrival)^*\ Station$$

Abstraction:

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

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, possibility to define semantic functions.
- Main criterion for parser: efficiency, i.e., left-factored and no left-recursion.
- If the underlying grammars for abstract syntax and grammar are different, more work is required in the semantic functions.

Deciding on the type – contd.

In our case, the grammar

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

is both readable and suitable for a parser.

Deciding on the type – contd.

In our case, the grammar

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

is both readable and suitable for a parser.

```haskell
data TS        = TS  [Leg] Station
data Leg       = Leg Station Departure Arrival
type Station   = String
type Departure = Time
type Arrival   = Time
data Time      = Time Hours Minutes
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:

$$\text{spaces} :: \text{Parser Char String}$$
$$\text{spaces} = \text{many}\ (\text{satisfy isSpace})$$

Question

What about

$$\text{moreSpaces} :: \text{Parser Char String}$$
$$\text{moreSpaces} = (+\!\!+) <\$> \text{spaces} <*> \text{spaces}$$

?

Observations regarding whitespace

- Where to call spaces? It is best to handle whitespace systematically, in order to prevent uses of spaces all over the place.
- Multiple sequential uses of spaces lead to ambiguity.
- Most often, we want to discard **all** whitespace in a particular place, i.e., 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 subsequent whitespace,
- the parser for the start symbol starts by consuming initial whitespace,
- no other parser consumes any whitespace.

Greedy parsers

In order to prevent partial results, we extend our **primitive** parser combinators with a greedy choice operator.

$$
\begin{array}{l}
(\ll\!|\!>) :: \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s\ a} \\
\mathsf{Parser\ p} \ll\!|\!> \mathsf{Parser\ q} = \mathsf{Parser}\ (\lambda \mathsf{xs} \rightarrow \mathbf{let}\ \mathsf{r} = \mathsf{p\ xs} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in\ if}\ \mathsf{null\ r} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \mathsf{q\ xs} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ \mathsf{r})
\end{array}
$$

Greedy parsers

In order to prevent partial results, we extend our **primitive** parser combinators with a greedy choice operator.

$$
\begin{aligned}
&(\lll\!|\!>) :: \mathsf{Parser}\ s\ a \to \mathsf{Parser}\ s\ a \to \mathsf{Parser}\ s\ a\\
&\mathsf{Parser}\ p \lll\!|\!> \mathsf{Parser}\ q = \mathsf{Parser}\ (\lambda xs \to \mathbf{let}\ r = p\ xs\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathbf{in\ if}\ null\ r\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ q\ xs\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \ r)
\end{aligned}
$$

We can then define greedy versions of many and some:

$$
\begin{aligned}
&greedy, greedy_1 :: \mathsf{Parser}\ s\ a \to \mathsf{Parser}\ s\ [a]\\
&greedy\ \ \ p = (:) <\$> p <\!*\!> greedy\ p \lll\!|\!> succeed\ []\\
&greedy_1\ p = (:) <\$> p <\!*\!> greedy\ p
\end{aligned}
$$

Parsing spaces greedily

$$\begin{array}{l}\mathrm{spaces} :: \mathrm{Parser}\ \mathrm{Char}\ \mathrm{String}\\ \mathrm{spaces} = \mathrm{greedy}\ (\mathrm{satisfy}\ \mathrm{isSpace})\end{array}$$

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:

$$\text{Station} \rightarrow \text{Identifier}^{+}$$

Parser:

$$\begin{aligned}&\text{station} :: \text{Parser Char Station}\\&\text{station} = \text{unwords} <\$> \text{greedy}_1\ (\text{identifier} <* \text{spaces})\end{aligned}$$

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

Tokens in travel schemes

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

Haskell:

$$\begin{aligned}
&\textbf{data}\ \text{Time} &&= \text{Time Hours Minutes}\\
&\textbf{type}\ \text{Hours} &&= \text{Int}\\
&\textbf{type}\ \text{Minutes} &&= \text{Int}
\end{aligned}$$

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:

$$
\begin{aligned}
&\textbf{data}\ \text{TS} &&= \text{TS}\ [\text{Leg}]\ \text{Station} \\
&\textbf{data}\ \text{Leg} &&= \text{Leg Station Departure Arrival}
\end{aligned}
$$

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:

$$\text{start} :: \text{Parser Char TS}$$
$$\text{start} = \text{spaces} \ast\!\!> \text{ts} <\!\!\ast \text{eof}$$

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:

$$\textbf{data}\ \text{Token} = \text{TStation Station} \\ \qquad\qquad\quad \mid \text{TTime} \quad \text{Time}$$

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

$$\text{tstation} :: \text{Parser Char Token}$$
$$\text{tstation} = \text{TStation} \,.\, \text{unwords} <\$> \text{greedy}_1\ (\text{identifier} <* \text{spaces})$$
$$\text{ttime} :: \text{Parser Char Token}$$
$$\text{ttime} = \\ \quad \text{TTime} <\$> \\ \quad (\text{Time} <\$> \text{natural} <* \text{symbol}\ \text{':'} <*> \text{natural} <* \text{spaces})$$

Using an explicit scanner – contd.

The scanner parses any number of tokens:

$$\mathit{anyToken} :: \mathsf{Parser\ Char\ Token}$$
$$\mathit{anyToken} = \mathit{tstation} <|> \mathit{ttime}$$
$$\mathit{scan} :: \mathsf{Parser\ Char\ [Token]}$$
$$\mathit{scan} = \mathit{spaces} *> \mathit{greedy}\ \mathit{anyToken} <* \mathit{eof}$$

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

Parsing a particular type of tokens

$$
\begin{array}{l}
\mathsf{station} :: \mathsf{Parser\ Token\ Station} \\
\mathsf{station} = \mathsf{fromStation} <\$> \mathsf{satisfy\ isStation}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{isStation} :: \mathsf{Token} \to \mathsf{Bool} & \\
\mathsf{isStation\ (TStation\ \_)} & = \mathsf{True} \\
\mathsf{isStation\ \_} & = \mathsf{False}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{fromStation} :: \mathsf{Token} \to \mathsf{Station} & \\
\mathsf{fromStation\ (TStation\ x)} & = \mathsf{x} \\
\mathsf{fromStation\ \_} & = \mathsf{error}\ \texttt{"fromStation"}
\end{array}
$$

Parsing a particular type of tokens

$$\begin{array}{l}
\mathsf{station} :: \mathsf{Parser\ Token\ Station} \\
\mathsf{station} = \mathsf{fromStation} <\$> \mathsf{satisfy\ isStation}
\end{array}$$

$$\begin{array}{ll}
\mathsf{isStation} :: \mathsf{Token} \to \mathsf{Bool} & \\
\mathsf{isStation}\ (\mathsf{TStation}\ _) & = \mathsf{True} \\
\mathsf{isStation}\ _ & = \mathsf{False}
\end{array}$$

$$\begin{array}{ll}
\mathsf{fromStation} :: \mathsf{Token} \to \mathsf{Station} & \\
\mathsf{fromStation}\ (\mathsf{TStation}\ x) & = x \\
\mathsf{fromStation}\ _ & = \mathsf{error}\ \texttt{"fromStation"}
\end{array}$$

Similarly:

$$\begin{array}{l}
\mathsf{time} :: \mathsf{Parser\ Token\ Time} \\
\mathsf{time} = \mathsf{fromTime} <\$> \mathsf{satisfy\ isTime}
\end{array}$$

A token parser for travel schemes

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

$$\mathsf{ts} :: \mathsf{Parser\ Token\ TS}$$
$$\mathsf{ts} = \mathsf{TS} <\$> \mathsf{many\ leg} <*> \mathsf{station}$$
$$\mathsf{leg} :: \mathsf{Parser\ Token\ Leg}$$
$$\mathsf{leg} = \mathsf{Leg} <\$> \mathsf{station} <*> \mathsf{time} <*> \mathsf{time}$$

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

Net travel time

$$
\begin{aligned}
&\mathsf{netTravelTimeTS} :: \mathsf{TS} \rightarrow \mathsf{Minutes}\\
&\mathsf{netTravelTimeTS}\ (\mathsf{TS}\ \mathsf{ls}\ \_) = \mathsf{sum}\ (\mathsf{map}\ \mathsf{netTravelTimeLeg}\ \mathsf{ls})\\
&\mathsf{netTravelTimeLeg} :: \mathsf{Leg} \rightarrow \mathsf{Minutes}\\
&\mathsf{netTravelTimeLeg}\ (\mathsf{Leg}\ \_\ \mathsf{dep}\ \mathsf{arr}) = \mathsf{arr}\ \text{`timeDiff`}\ \mathsf{dep}
\end{aligned}
$$

Net travel time

$$\mathit{netTravelTimeTS} :: \mathit{TS} \rightarrow \mathit{Minutes}$$
$$\mathit{netTravelTimeTS}\ (\mathit{TS}\ \mathit{ls}\ \_) = \mathit{sum}\ (\mathit{map}\ \mathit{netTravelTimeLeg}\ \mathit{ls})$$
$$\mathit{netTravelTimeLeg} :: \mathit{Leg} \rightarrow \mathit{Minutes}$$
$$\mathit{netTravelTimeLeg}\ (\mathit{Leg}\ \_\ \mathit{dep}\ \mathit{arr}) = \mathit{arr}\ \text{`}\mathit{timeDiff}\text{`}\ \mathit{dep}$$

The rest is dealing with times.

Excursion: Time difference

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

$$
\begin{array}{l}
\mathsf{timeDiff} :: \mathsf{Time} \rightarrow \mathsf{Time} \rightarrow \mathsf{Minutes} \\
\mathsf{timeDiff}\ (\mathsf{Time}\ \mathsf{h}_1\ \mathsf{m}_1)\ (\mathsf{Time}\ \mathsf{h}_2\ \mathsf{m}_2) \\
\quad \mid \mathsf{h}_2 > \mathsf{h}_1 \vee (\mathsf{h}_2 == \mathsf{h}_1 \wedge \mathsf{m}_2 > \mathsf{m}_1) \\
\qquad\qquad = (\mathsf{h}_2 \phantom{{}+24} - \mathsf{h}_1) * 60 + \mathsf{m}_2 - \mathsf{m}_1 \\
\quad \mid \mathsf{otherwise} = (\mathsf{h}_2 + 24 - \mathsf{h}_1) * 60 + \mathsf{m}_2 - \mathsf{m}_1
\end{array}
$$

Waiting time

$$
\begin{aligned}
&\mathsf{waitingTimeTS} :: \mathsf{TS} \rightarrow \mathsf{Minutes}\\
&\mathsf{waitingTimeTS}\ (\mathsf{TS}\ \mathsf{ls}\ \_) = \mathsf{waitingTimeLegs}\ \mathsf{ls}\\
&\mathsf{waitingTimeLegs} :: [\mathsf{Leg}] \rightarrow \mathsf{Minutes}\\
&\mathsf{waitingTimeLegs}\ (\mathsf{Leg}\ \_\ \_\ \mathsf{arr} : \mathsf{ls}@(\mathsf{Leg}\ \_\ \mathsf{dep}\ \_ : \_)) =\\
&\quad \mathsf{dep}\ \text{`}\mathsf{timeDiff}\text{`}\ \mathsf{arr} + \mathsf{waitingTimeLegs}\ \mathsf{ls}\\
&\mathsf{waitingTimeLegs}\ \_ = 0
\end{aligned}
$$

Universiteit Utrecht

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

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:

$$\begin{aligned}
&\mathsf{ts} :: \mathsf{Parser\ Token\ Minutes}\\
&\mathsf{ts} = \mathsf{sum} <\$> \mathsf{many\ leg} <\ast\ \mathsf{station}\\
&\mathsf{leg} :: \mathsf{Parser\ Token\ Minutes}\\
&\mathsf{leg} = \mathsf{flip\ timeDiff} <\$\ \mathsf{station} <\ast> \mathsf{time} <\ast> \mathsf{time}
\end{aligned}$$

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.

> Time → Nat : Nat

Parsing times

Both syntax and parser for times are too liberal.

Time → Nat : 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 $(<\$>)$:

$$(<\$>) :: (a \rightarrow b) \rightarrow \text{Parser } s\ a \rightarrow \text{Parser } s\ b$$

Does not work:

$$\text{hours} :: \text{Parser Char Hours}$$
$$\text{hours} = (\lambda x \rightarrow \textbf{if } x < 24 \textbf{ then} \ldots \textbf{else} \ldots) <\$> \text{natural}$$

Another sequencing combinator

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

$$(<\$>) :: (a \to b) \to \text{Parser s a} \to \text{Parser s b}$$

Does not work:

$$\text{hours} :: \text{Parser Char Hours}$$
$$\text{hours} = (\lambda x \to \textbf{if}\ x < 24\ \textbf{then} \ldots \textbf{else} \ldots) <\$> \text{natural}$$

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:

$$
\begin{array}{l}
(\gg\!=) :: \mathsf{Parser}\ s\ a \to (a \to \mathsf{Parser}\ s\ b) \to \mathsf{Parser}\ s\ b \\
\mathsf{Parser}\ p \gg\!= f = \\
\quad \mathsf{Parser}\ (\lambda xs \to [(s, zs) \mid (r, ys) \leftarrow p\ xs, \\
\qquad\qquad\qquad\qquad\qquad\qquad (s, zs) \leftarrow \mathsf{runParser}\ (f\ r)\ ys])
\end{array}
$$

Another sequencing combinator – contd.

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

$$
\begin{array}{l}
(\gg\!=) :: \mathsf{Parser}\ s\ a \to (a \to \mathsf{Parser}\ s\ b) \to \mathsf{Parser}\ s\ b \\
\mathsf{Parser}\ p \gg\!= f = \\
\quad \mathsf{Parser}\ (\lambda xs \to [(s, zs) \mid (r, ys) \leftarrow p\ xs, \\
\qquad\qquad\qquad\qquad\qquad\qquad (s, zs) \leftarrow \mathsf{runParser}\ (f\ r)\ ys])
\end{array}
$$

Now:

$$
\begin{array}{l}
\mathsf{hours} :: \mathsf{Parser}\ \mathsf{Char}\ \mathsf{Hours} \\
\mathsf{hours} = \mathsf{natural} \gg\!= \lambda x \to \\
\qquad\quad \textbf{if}\ x < 24\ \textbf{then}\ \mathsf{succeed}\ x\ \textbf{else}\ \mathsf{empty}
\end{array}
$$

Universiteit Utrecht

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

Using bind

Question

What is the difference between the following two keyword parsers?

$$
\begin{array}{l}
\mathsf{keyword}_1, \mathsf{keyword}_2 :: \mathsf{String} \to \mathsf{Parser\ Char\ String} \\
\mathsf{keyword}_1 \qquad = \mathsf{token} \\
\mathsf{keyword}_2\ \mathsf{xs} = \mathsf{greedy\ isLetter} \ggg\!= \lambda \mathsf{ys} \to \\
\qquad\qquad\qquad \textbf{if}\ \mathsf{xs} == \mathsf{ys}\ \textbf{then}\ \mathsf{succeed\ ys}\ \textbf{else}\ \mathsf{empty}
\end{array}
$$

Using bind

Question

What is the difference between the following two keyword parsers?

$$
\begin{aligned}
&\text{keyword}_1, \text{keyword}_2 :: \text{String} \to \text{Parser Char String}\\
&\text{keyword}_1 \quad\;\; = \text{token}\\
&\text{keyword}_2\ \text{xs} = \text{greedy isLetter} \ggg\!= \lambda \text{ys} \to\\
&\qquad\qquad\qquad \textbf{if}\ \text{xs} == \text{ys}\ \textbf{then}\ \text{succeed ys}\ \textbf{else}\ \text{empty}
\end{aligned}
$$

Hint

Consider keyword `"let"` $*\!\!>$ spaces $*\!\!>$ identifier and the string `"letx"`.

Applicative functors

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

$$
\begin{array}{l}
\textbf{class}\ \text{Functor f}\ \textbf{where} \\
\quad \text{fmap}\ \ :: (a \to b) \to f\ a \to f\ b \\
(<\$>) = \text{fmap} \\
\textbf{class}\ \text{Functor f} \Rightarrow \text{Applicative f}\ \textbf{where} \\
\quad \text{pure}\ \ :: a \to f\ a \\
\quad (<*>) :: f\ (a \to b) \to f\ a \to f\ b \\
\textbf{class}\ \text{Applicative f} \Rightarrow \text{Alternative f}\ \textbf{where} \\
\quad \text{empty} :: f\ a \\
\quad (<|>) :: f\ a \to f\ a \to f\ a
\end{array}
$$

Monads

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

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

Monads

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