Talen en Compilers

2010/2011, periode 2

Johan Jeuring

Department of Information and Computing Sciences
Utrecht University

November 22, 2010

3. Parser combinators

This lecture

Parser combinators

- Recap
- Parsing
- Developing parser combinators

3.1 Recap

Parsing problem

Given a grammar G and a string s, the **parsing problem** is to decide whether or not s $\in L(\mathsf{G})$.

Furthermore, if s $\in L(\mathsf{G})$, we want evidence/proof/an explanation why this is the case, usually in the form of a parse tree.

Parse trees in Haskell

$$\begin{array}{l} S \rightarrow S\text{–}D \mid D \\ D \rightarrow 0 \mid 1 \end{array}$$

```haskell
data S = Minus S D | SingleDigit D
data D = Zero | One
```

Concrete syntax: context-free grammar.

Abstract syntax: (Haskell datatype), does no longer contain information about terminals that can easily be reconstructed.

Grammars

Context-free grammars can be used to describe lots of interesting languages.

Several grammars can describe the same language, not all of them being equally suited as a starting point for parsing.

Ambiguity is an example of an undesirable property of grammars.

3.2 Parsing

Approaches to parsing

Parser generators

- External program
- based on a bottom-up algorithm, usually LL or LR
- complex theory
- limited look-ahead, usually one token
- only built-in abstractions
- generated parsers are extremely fast

Parser combinators

- Library
- based on a top-down algorithm
- underlying theory is simple
- in principle unlimited look-ahead
- user-definable abstractions
- fast as long as certain constructs are not used

Both approaches place certain (but different) constraints on the grammars being used.

Approaches to parsing – contd.

In the first part of the course, we will work with combinators.

Approaches to parsing – contd.

In the first part of the course, we will work with combinators.

Towards the end of the course, we will learn the theory of parser generators.

Approaches to parsing – contd.

In the first part of the course, we will work with combinators.

Towards the end of the course, we will learn the theory of parser generators.

In the practicum tasks, you will use both parser generators and parser combinators.

Aside: Combinators

The term combinator denotes a self-contained function in **lambda calculus**, the formal system that is the basis of Haskell and other functional programming languages.

Parser combinators are thus a set of (small) library functions that can be used to construct parsers.

Lexing and parsing

Often, parsing is split into a two-phase process:

Lexing and parsing

Often, parsing is split into a two-phase process:

Lexing

In a first phase, whitespace and comments are removed and the input is organized into a sequence of **tokens** – small entities that belong together such as keywords, identifiers or operators.

Lexing and parsing

Often, parsing is split into a two-phase process:

Lexing

In a first phase, whitespace and comments are removed and the input is organized into a sequence of **tokens** – small entities that belong together such as keywords, identifiers or operators.

Parsing

In the second phase, an abstract syntax tree is constructed from the list of tokens rather than from the original list of characters.

Lexing and parsing – contd.

In the world of generators, lexing and parsing is often performed by separate generators. In Haskell, for example: Alex (lexer) and Happy (parser). For C: flex (lexer), yacc/bison (parser).

Lexing and parsing – contd.

In the world of generators, lexing and parsing is often performed by separate generators. In Haskell, for example: Alex (lexer) and Happy (parser). For C: flex (lexer), yacc/bison (parser).

With parser combinators, there are different options:

- use only one phase,
- use the same parser combinators for both phases,
- use dedicated lexer combinators for lexing,
- use a hand-written special-purpose lexer,
- combine a lexer generator with parser combinators.

3.3 Developing parser combinators

Words of warning

We are going to **develop** a suitable type of parsers.

Along the path, we will make several suboptimal or even wrong attempts.

First attempt: a predicate on strings

$\textbf{type}\ \mathsf{Parser}_1 = \mathsf{String} \rightarrow \mathsf{Bool}$

First attempt: a predicate on strings

$$\textbf{type}\ \mathrm{Parser}_1 = \mathrm{String} \rightarrow \mathrm{Bool}$$

With this type, we can write very simple parsers:

$$
\begin{aligned}
&\mathrm{manyLetters}_1 :: \mathrm{Parser}_1\\
&\mathrm{manyLetters}_1\ \mathrm{xs} = \mathrm{all}\ \mathrm{isLetter}\ \mathrm{xs}\\
&\mathrm{someDigits}_1 :: \mathrm{Parser}_1\\
&\mathrm{someDigits}_1\ \mathrm{xs} = \mathrm{all}\ \mathrm{isDigit}\ \mathrm{xs} \wedge \mathrm{not}\ (\mathrm{null}\ \mathrm{xs})
\end{aligned}
$$

First attempt – contd.

type $\text{Parser}_1 = \text{String} \rightarrow \text{Bool}$

Disadvantages:

- Only yes or no as answer.
- Works only on strings.
- Difficult to combine.

First attempt – contd.

type $\text{Parser}_1 = \text{String} \rightarrow \text{Bool}$

Disadvantages:

- Only yes or no as answer.
- Works only on strings.
- Difficult to combine.

We first look into combining parsers, later at the other points.

Motivation: sequencing parsers

Assume we want to combine:

$$\begin{aligned} &\mathsf{manyLetters}_1 :: \mathsf{Parser}_1 \\ &\mathsf{someDigits}_1 :: \mathsf{Parser}_1 \end{aligned}$$

and parse many letters followed by many digits.

Motivation: sequencing parsers

Assume we want to combine:

$$\begin{aligned} \text{manyLetters}_1 &:: \text{Parser}_1 \\ \text{someDigits}_1 &:: \text{Parser}_1 \end{aligned}$$

and parse many letters followed by many digits.

We cannot, because:

- both parsers work on the complete input string
- there is no way to split the input,
- we do not (in general) know where to split the input without running the first parser.

Universiteit Utrecht

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

Second attempt: returning the remaining string

Idea

- Parsers can consume an initial part of the input.
- Parsers only look at the initial part of the input.
- Parsers return the rest of the string.

Second attempt: returning the remaining string

Idea

- Parsers can consume an initial part of the input.
- Parsers only look at the initial part of the input.
- Parsers return the rest of the string.

What type to choose?

$$\textbf{type}\ \mathsf{Parser}_2 = \mathsf{String} \to \ \ldots$$

Second attempt: returning the remaining string

Idea

- Parsers can consume an initial part of the input.
- Parsers only look at the initial part of the input.
- Parsers return the rest of the string.

What type to choose?

$\textbf{type}\ \text{Parser}_2 = \text{String} \rightarrow\ \ldots$

We need the remaining string only if parsing was successful:

$\textbf{data}\ \text{Maybe a} = \text{Nothing} \mid \text{Just a}$

Second attempt: returning the remaining string

Idea

- Parsers can consume an initial part of the input.
- Parsers only look at the initial part of the input.
- Parsers return the rest of the string.

What type to choose?

$\textbf{type}\ \text{Parser}_2 = \text{String} \rightarrow \text{Maybe String}$

We need the remaining string only if parsing was successful:

$\textbf{data}\ \text{Maybe a} = \text{Nothing} \mid \text{Just a}$

Example

We now have to write the parsers such that they work on the initial part of the string:

$$\text{manyLetters}_2 :: \text{Parser}_2$$
$$\text{manyLetters}_2\ \text{xs} = \text{Just}\ (\text{dropWhile isLetter xs})$$

Example

We now have to write the parsers such that they work on the initial part of the string:

$$\mathsf{manyLetters}_2 :: \mathsf{Parser}_2$$
$$\mathsf{manyLetters}_2\ \mathsf{xs} = \mathsf{Just}\ (\mathsf{dropWhile}\ \mathsf{isLetter}\ \mathsf{xs})$$

Note that $\mathsf{manyLetters}_2$ cannot fail.

Example

We now have to write the parsers such that they work on the initial part of the string:

$$\begin{aligned}&\mathsf{manyLetters}_2 :: \mathsf{Parser}_2\\&\mathsf{manyLetters}_2\ \mathsf{xs} = \mathsf{Just}\ (\mathsf{dropWhile\ isLetter\ xs})\end{aligned}$$

Note that $\mathsf{manyLetters}_2$ cannot fail.

$$\begin{aligned}&\mathsf{someDigits}_2 :: \mathsf{Parser}_2\\&\mathsf{someDigits}_2\ \mathsf{xs} = \mathbf{case}\ \mathsf{span\ isDigit\ xs}\ \mathbf{of}\\&\qquad\qquad\qquad ([\,], \_\,) \rightarrow \mathsf{Nothing}\\&\qquad\qquad\qquad (\_, \mathsf{ys}) \rightarrow \mathsf{Just\ ys}\end{aligned}$$

Example – contd.

We can now sequence $\mathrm{manyLetters}_2$ and $\mathrm{someDigits}_2$:

```
lettersThenDigits₂ :: Parser₂
lettersThenDigits₂ xs =
   case manyLetters₂ xs of
      Nothing → Nothing
      Just ys  → someDigits₂ ys
```

Example – contd.

We can now sequence manyLetters_2 and someDigits_2:

```
lettersThenDigits₂ :: Parser₂
lettersThenDigits₂ xs =
  case manyLetters₂ xs of
    Nothing → Nothing
    Just ys  → someDigits₂ ys
```

We can abstract from the sequencing operation.

Universiteit Utrecht
[Faculty of Science
Information and Computing Sciences]

Example – contd.

We can now sequence $\mathsf{manyLetters}_2$ and $\mathsf{someDigits}_2$:

```
lettersThenDigits₂ :: Parser₂
lettersThenDigits₂ xs =
  case manyLetters₂ xs of
    Nothing → Nothing
    Just ys → someDigits₂ ys
```

We can abstract from the sequencing operation.

```
(<*>) :: Parser₂ → Parser₂ → Parser₂
(p <*> q) xs =
  case p xs of
    Nothing → Nothing
    Just ys → q ys
```

Example – contd.

We can now sequence $\mathsf{manyLetters}_2$ and $\mathsf{someDigits}_2$:

$$
\begin{array}{l}
\mathsf{lettersThenDigits}_2 :: \mathsf{Parser}_2 \\
\mathsf{lettersThenDigits}_2\ \mathsf{xs} = \\
\quad \textbf{case}\ \mathsf{manyLetters}_2\ \mathsf{xs}\ \textbf{of} \\
\qquad \mathsf{Nothing} \rightarrow \mathsf{Nothing} \\
\qquad \mathsf{Just\ ys} \rightarrow \mathsf{someDigits}_2\ \mathsf{ys}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{lettersThenDigits}_2 :: \mathsf{Parser}_2 \\
\mathsf{lettersThenDigits}_2 = \\
\qquad \mathsf{manyLetters}_2 \\
\quad <\!\!*\!\!> \mathsf{someDigits}_2
\end{array}
$$

We can abstract from the sequencing operation.

$$
\begin{array}{l}
(<\!\!*\!\!>) :: \mathsf{Parser}_2 \rightarrow \mathsf{Parser}_2 \rightarrow \mathsf{Parser}_2 \\
(\mathsf{p} <\!\!*\!\!> \mathsf{q})\ \mathsf{xs} = \\
\quad \textbf{case}\ \mathsf{p\ xs}\ \textbf{of} \\
\qquad \mathsf{Nothing} \rightarrow \mathsf{Nothing} \\
\qquad \mathsf{Just\ ys} \rightarrow \mathsf{q\ ys}
\end{array}
$$

The end of the input

The $\mathsf{lettersThenDigits}_2$ parser works as follows:

$\mathsf{lettersThenDigits}_2$	`"abc123"`	evaluates to
$\mathsf{lettersThenDigits}_2$	`"abc"`	evaluates to
$\mathsf{lettersThenDigits}_2$	`"123"`	evaluates to
$\mathsf{lettersThenDigits}_2$	`"a1x"`	evaluates to

The end of the input

The $\mathsf{lettersThenDigits}_2$ parser works as follows:

$\mathsf{lettersThenDigits}_2$ `"abc123"`	evaluates to	Just `""`
$\mathsf{lettersThenDigits}_2$ `"abc"`	evaluates to	Nothing
$\mathsf{lettersThenDigits}_2$ `"123"`	evaluates to	Just `""`
$\mathsf{lettersThenDigits}_2$ `"a1x"`	evaluates to	Just `"x"`

The end of the input

The $\mathsf{lettersThenDigits}_2$ parser works as follows:

$\mathsf{lettersThenDigits}_2$ `"abc123"`	evaluates to	Just `""`
$\mathsf{lettersThenDigits}_2$ `"abc"`	evaluates to	Nothing
$\mathsf{lettersThenDigits}_2$ `"123"`	evaluates to	Just `""`
$\mathsf{lettersThenDigits}_2$ `"a1x"`	evaluates to	Just `"x"`

We define a special parser that expects the input to be empty:

$$\mathsf{eof}_2 :: \mathsf{Parser}_2$$
$$\mathsf{eof}_2\ [\,] = \mathsf{Just}\ [\,]$$
$$\mathsf{eof}_2\ \_ = \mathsf{Nothing}$$

The end of the input

The $\mathsf{lettersThenDigits}_2$ parser works as follows:

$\mathsf{lettersThenDigits}_2$ `"abc123"`	evaluates to	Just `""`
$\mathsf{lettersThenDigits}_2$ `"abc"`	evaluates to	$\mathsf{Nothing}$
$\mathsf{lettersThenDigits}_2$ `"123"`	evaluates to	Just `""`
$\mathsf{lettersThenDigits}_2$ `"a1x"`	evaluates to	Just `"x"`

We define a special parser that expects the input to be empty:

$$\begin{array}{l} \mathsf{eof}_2 :: \mathsf{Parser}_2 \\ \mathsf{eof}_2\ [\,] = \mathsf{Just}\ [\,] \\ \mathsf{eof}_2\ \_ = \mathsf{Nothing} \end{array}$$

Now we can reject `"a1x"`:

$$\mathsf{lettersThenDigits}_2' = \mathsf{manyLetters}_2 \mathbin{<\!\!*\!\!>} \mathsf{someDigits}_2 \mathbin{<\!\!*\!\!>} \mathsf{eof}_2$$

Another example

Consider the grammar

$\mathsf{S} \rightarrow \mathsf{Letter}^{*}\ \mathsf{a}$

Another example

Consider the grammar

> S → Letter* a

Is this grammar ambiguous?

Another example

Consider the grammar

> S → Letter* a

Is this grammar ambiguous?

No, but it is problematic to parse with our current approach.

Another example

Consider the grammar

$S \rightarrow \text{Letter}^* \ a$

Is this grammar ambiguous?

No, but it is problematic to parse with our current approach.

We can easily define a parser for a single a:

$$
\begin{aligned}
&\text{singleA}_2 :: \text{Parser}_2 \\
&\text{singleA}_2 \ (\texttt{'a'} : xs) = \text{Just } xs \\
&\text{singleA}_2 \ \_ \qquad\qquad\ \ = \text{Nothing}
\end{aligned}
$$

Can you now see the problem?

Ambiguity revisited

> $(\text{manyLetters}_2 \mathbin{<\!\!*\!\!>} \text{singleA}_2)$ "cba"   evaluates to   Nothing

There are multiple prefixes of "cba" that can be seen as a sequence of letters, yet manyLetters_2 is greedy and returns only one.

Such cases of ambiguity can arise during the parsing process even if the grammar as a whole is unambiguous.

Ambiguity revisited

> (manyLetters$_2$ <*> singleA$_2$) `"cba"` evaluates to Nothing

There are multiple prefixes of `"cba"` that can be seen as a sequence of letters, yet manyLetters$_2$ is greedy and returns only one.

Such cases of ambiguity can arise during the parsing process even if the grammar as a whole is unambiguous.

Our solution

Let parsers return multiple results instead of just one.

- Allows us to deal with the above case and also ambiguous grammars.
- Potential source of inefficiency.

Third attempt: multiple results

What type to choose?

$\textbf{type}\ \mathrm{Parser}_3 = \mathrm{String} \rightarrow \ \ldots$

Third attempt: multiple results

What type to choose?

> **type** $\text{Parser}_3 = \text{String} \rightarrow \ \ldots$

We can use a list. Failure is now represented as the empty list. Successful results are represented by their corresponding remaining strings.

Third attempt: multiple results

What type to choose?

$$\textbf{type}\ \text{Parser}_3 = \text{String} \rightarrow [\text{String}]$$

We can use a list. Failure is now represented as the empty list. Successful results are represented by their corresponding remaining strings.

The technique of using a list of successful results as a return value is called **list of successes** method.

Choice and sequence

The new parser type gives us an easy way to write down a choice between two parsers:

$$
\begin{aligned}
&(<\!|\!>) :: \mathsf{Parser}_3 \to \mathsf{Parser}_3 \to \mathsf{Parser}_3\\
&(\mathsf{p} <\!|\!> \mathsf{q})\ \mathsf{xs} = \mathsf{p}\ \mathsf{xs} +\!\!+\ \mathsf{q}\ \mathsf{xs}
\end{aligned}
$$

Choice and sequence

The new parser type gives us an easy way to write down a choice between two parsers:

$$
\begin{aligned}
&(<|>) :: \mathsf{Parser}_3 \rightarrow \mathsf{Parser}_3 \rightarrow \mathsf{Parser}_3 \\
&(p <|> q)\ xs = p\ xs +\!\!+\ q\ xs
\end{aligned}
$$

On the other hand, sequencing becomes a bit more difficult, because we have to deal with multiple results:

$$
\begin{aligned}
&(<\!\!*\!\!>) :: \mathsf{Parser}_3 \rightarrow \mathsf{Parser}_3 \rightarrow \mathsf{Parser}_3 \\
&(p <\!\!*\!\!> q)\ xs = [zs \mid ys \leftarrow p\ xs, zs \leftarrow q\ ys]
\end{aligned}
$$

Choice and sequence

The new parser type gives us an easy way to write down a choice between two parsers:

$$(<|>) :: \mathsf{Parser}_3 \rightarrow \mathsf{Parser}_3 \rightarrow \mathsf{Parser}_3$$
$$(\mathsf{p} <|> \mathsf{q})\ \mathsf{xs} = \mathsf{p}\ \mathsf{xs} +\!\!+\ \mathsf{q}\ \mathsf{xs}$$

On the other hand, sequencing becomes a bit more difficult, because we have to deal with multiple results:

$$(<\!\!*\!\!>) :: \mathsf{Parser}_3 \rightarrow \mathsf{Parser}_3 \rightarrow \mathsf{Parser}_3$$
$$(\mathsf{p} <\!\!*\!\!> \mathsf{q})\ \mathsf{xs} = [\mathsf{zs} \mid \mathsf{ys} \leftarrow \mathsf{p}\ \mathsf{xs}, \mathsf{zs} \leftarrow \mathsf{q}\ \mathsf{ys}]$$

We define that $(<\!\!*\!\!>)$ binds stronger than $(<|>)$:

$$\textbf{infixl}\ 4\ <\!\!*\!\!>$$
$$\textbf{infixr}\ 3\ <|>$$

Revisiting the examples

We can build manyLetters_3 out of smaller blocks!

$$\text{ManyLetters} \rightarrow \text{Letter ManyLetters} \mid \varepsilon$$

Universiteit Utrecht

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

Revisiting the examples

We can build $\mathsf{manyLetters}_3$ out of smaller blocks!

$$\mathsf{ManyLetters} \rightarrow \mathsf{Letter}\ \mathsf{ManyLetters} \mid \varepsilon$$

We can easily define parsers for Letter and ε:

```
epsilon₃ :: Parser₃
epsilon₃ xs = [xs]

letter₃ :: Parser₃
letter₃ (x : xs) | isLetter x = [xs]
letter₃ _                     = []
```

Revisiting the examples

We can build $\mathsf{manyLetters}_3$ out of smaller blocks!

$$\mathsf{ManyLetters} \rightarrow \mathsf{Letter}\ \mathsf{ManyLetters} \mid \varepsilon$$

We can easily define parsers for Letter and ε:

```
epsilon₃ :: Parser₃
epsilon₃ xs = [xs]

letter₃ :: Parser₃
letter₃ (x : xs) | isLetter x = [xs]
letter₃ _                     = []
```

Now we can define a parser for ManyLetters:

```
manyLetters₃ :: Parser₃
manyLetters₃ = letter₃ <*> manyLetters₃ <|> epsilon₃
```

More abstraction

$$
\begin{aligned}
&\mathsf{satisfy}_3 :: (\mathsf{Char} \to \mathsf{Bool}) \to \mathsf{Parser}_3\\
&\mathsf{satisfy}_3\ \mathsf{p}\ (\mathsf{x} : \mathsf{xs}) \mid \mathsf{p}\ \mathsf{x} = [\mathsf{xs}]\\
&\mathsf{satisfy}_3\ \_\ \_ \qquad\qquad\quad\ = []
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{letter}_3 &= \mathsf{satisfy}_3\ \mathsf{isLetter}\\
\mathsf{digit}_3 &= \mathsf{satisfy}_3\ \mathsf{isDigit}
\end{aligned}
$$

More abstraction

$$
\begin{aligned}
&\mathsf{satisfy}_3 :: (\mathsf{Char} \rightarrow \mathsf{Bool}) \rightarrow \mathsf{Parser}_3 \\
&\mathsf{satisfy}_3\ \mathsf{p}\ (\mathsf{x} : \mathsf{xs}) \mid \mathsf{p}\ \mathsf{x} = [\mathsf{xs}] \\
&\mathsf{satisfy}_3\ \_\ \_ \qquad\qquad\quad = []
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{letter}_3 = \mathsf{satisfy}_3\ \mathsf{isLetter} \\
&\mathsf{digit}_3\ \ = \mathsf{satisfy}_3\ \mathsf{isDigit}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{many}_3 :: \mathsf{Parser}_3 \rightarrow \mathsf{Parser}_3 \\
&\mathsf{many}_3\ \mathsf{p} = \mathsf{p} \mathrel{<\!\!*\!\!>} \mathsf{many}_3\ \mathsf{p} \mathrel{<\!|\!>} \mathsf{epsilon}_3
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{some}_3 :: \mathsf{Parser}_3 \rightarrow \mathsf{Parser}_3 \\
&\mathsf{some}_3\ \mathsf{p}\ = \mathsf{p} \mathrel{<\!\!*\!\!>} \mathsf{some}_3\ \mathsf{p} \mathrel{<\!|\!>} \mathsf{p}
\end{aligned}
$$

Universiteit Utrecht

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

More abstraction

$$\begin{aligned}
&\mathsf{satisfy}_3 :: (\mathsf{Char} \to \mathsf{Bool}) \to \mathsf{Parser}_3\\
&\mathsf{satisfy}_3\ \mathsf{p}\ (\mathsf{x} : \mathsf{xs}) \mid \mathsf{p}\ \mathsf{x} = [\mathsf{xs}]\\
&\mathsf{satisfy}_3\ _\ _ \qquad\qquad\quad = []
\end{aligned}$$

$$\begin{aligned}
&\mathsf{letter}_3 = \mathsf{satisfy}_3\ \mathsf{isLetter}\\
&\mathsf{digit}_3\ \ = \mathsf{satisfy}_3\ \mathsf{isDigit}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{many}_3 :: \mathsf{Parser}_3 \to \mathsf{Parser}_3\\
&\mathsf{many}_3\ \mathsf{p} = \mathsf{p} <\!\!*\!\!> \mathsf{many}_3\ \mathsf{p} <\!|\!> \mathsf{epsilon}_3
\end{aligned}$$

$$\begin{aligned}
&\mathsf{some}_3 :: \mathsf{Parser}_3 \to \mathsf{Parser}_3\\
&\mathsf{some}_3\ \mathsf{p}\ = \mathsf{p} <\!\!*\!\!> \mathsf{some}_3\ \mathsf{p}\ <\!|\!> \mathsf{p}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{manyLetters}_3 &&= \mathsf{many}_3\ \mathsf{letter}_3\\
&\mathsf{someDigits}_3 &&= \mathsf{some}_3\ \mathsf{digit}_3\\
&\mathsf{lettersThenDigits}_3 &&= \mathsf{manyLetters}_3 <\!\!*\!\!> \mathsf{someDigits}_3
\end{aligned}$$

Another example

$$
\begin{aligned}
I &\rightarrow 0 \mid 1 \mid B \\
B &\rightarrow [\ E\ ] \\
E &\rightarrow I\ ,\ E \mid I
\end{aligned}
$$

Another example

$$
\begin{array}{lll}
I & \rightarrow 0 \mid 1 \mid B \\
B & \rightarrow [\, E \,] \\
E & \rightarrow I \, , \, E \mid I
\end{array}
$$

```
[0,[[1,0],[0,1,1]]]
1
[[[[0,1,0,1]]]]
```

Another example

$$
\begin{aligned}
I &\rightarrow 0 \mid 1 \mid B \\
B &\rightarrow [\ E\ ] \\
E &\rightarrow I\ ,\ E \mid I
\end{aligned}
$$

```
[0,[[1,0],[0,1,1]]]
1
[[[[0,1,0,1]]]]
```

We need one additional abstraction:

$\mathsf{symbol}_3 :: \mathsf{Char} \rightarrow \mathsf{Parser}_3$
$\mathsf{symbol}_3\ x = \mathsf{satisfy}_3\ (== x)$

Another example

$$\begin{array}{lll} I & \rightarrow 0 \mid 1 \mid B \\ B & \rightarrow [\ E\ ] \\ E & \rightarrow I\ ,\ E \mid I \end{array}$$

```
[0,[[1,0],[0,1,1]]]
1
[[[[0,1,0,1]]]]
```

We need one additional abstraction:

$$\begin{array}{l} \mathrm{symbol}_3 :: \mathrm{Char} \rightarrow \mathrm{Parser}_3 \\ \mathrm{symbol}_3\ x = \mathrm{satisfy}_3\ (==x) \end{array}$$

The rest is entirely systematic:

$$\begin{array}{l} i, b, e :: \mathrm{Parser}_3 \\ i\ = \mathrm{symbol}_3\ \texttt{'0'} <\mid> \mathrm{symbol}_3\ \texttt{'1'} <\mid> b \\ b = \mathrm{symbol}_3\ \texttt{'['} <\ast> e <\ast> \mathrm{symbol}_3\ \texttt{']'} \\ e = i <\ast> \mathrm{symbol}_3\ \texttt{','} <\ast> e <\mid> i \end{array}$$

Intermediate summary

We have

- a small library of basic parser combinators,
- parsers for larger grammars can be constructed easily,
- new abstractions can be defined,
- we can follow the grammar structure in order to build a parser systematically.

Intermediate summary

We have

- a small library of basic parser combinators,
- parsers for larger grammars can be constructed easily,
- new abstractions can be defined,
- we can follow the grammar structure in order to build a parser systematically.

Still problematic:

- Only yes or no as answer.
- Works only on strings.

Intermediate summary

We have

- a small library of basic parser combinators,
- parsers for larger grammars can be constructed easily,
- new abstractions can be defined,
- we can follow the grammar structure in order to build a parser systematically.

Still problematic:

- Only yes or no as answer.
- Works only on strings.

Let us address the answers next.

Fourth step: adding results

Last lecture, we have seen that we can represent parse trees as values of specifically defined Haskell datatypes.

Fourth step: adding results

Last lecture, we have seen that we can represent parse trees as values of specifically defined Haskell datatypes.

Therefore, it is clear that different parsers should return different types of results.

Fourth step: adding results

Last lecture, we have seen that we can represent parse trees as values of specifically defined Haskell datatypes.

Therefore, it is clear that different parsers should return different types of results.

We parameterize the type of parsers over the type of the result. For each successful parse, we now return the result and the remaining string:

$$\textbf{data}\ \mathsf{Parser}_4\ \mathsf{r} = \mathsf{String} \to [(\mathsf{r}, \mathsf{String})]$$

Simple parsers with results

$$
\begin{aligned}
&\mathsf{epsilon}_4 :: \mathsf{Parser}_4\ ()\\
&\mathsf{epsilon}_4\ \mathsf{xs} = [((), \mathsf{xs})]\\
&\mathsf{satisfy}_4 :: (\mathsf{Char} \to \mathsf{Bool}) \to \mathsf{Parser}_4\ \mathsf{Char}\\
&\mathsf{satisfy}_4\ \mathsf{p}\ (\mathsf{x} : \mathsf{xs})\ |\ \mathsf{p}\ \mathsf{x} = [(\mathsf{x}, \mathsf{xs})]\\
&\mathsf{satisfy}_4\ \_\ \_ \qquad\qquad\quad = []
\end{aligned}
$$

Simple parsers with results

$$
\begin{array}{l}
\mathsf{epsilon}_4 :: \mathsf{Parser}_4\ ()\\
\mathsf{epsilon}_4\ \mathsf{xs} = [((), \mathsf{xs})]
\end{array}
$$

$$
\begin{array}{l}
\mathsf{satisfy}_4 :: (\mathsf{Char} \to \mathsf{Bool}) \to \mathsf{Parser}_4\ \mathsf{Char}\\
\mathsf{satisfy}_4\ \mathsf{p}\ (\mathsf{x} : \mathsf{xs})\ |\ \mathsf{p}\ \mathsf{x} = [(\mathsf{x}, \mathsf{xs})]\\
\mathsf{satisfy}_4\ \_\ \_ \qquad\qquad\quad = []
\end{array}
$$

As before (except for the types):

$$
\begin{array}{l}
\mathsf{letter}_4, \mathsf{digit}_4 :: \mathsf{Parser}_4\ \mathsf{Char}\\
\mathsf{letter}_4 = \mathsf{satisfy}_4\ \mathsf{isLetter}\\
\mathsf{digit}_4\ \ = \mathsf{satisfy}_4\ \mathsf{isDigit}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{symbol}_4 :: \mathsf{Char} \to \mathsf{Parser}_4\ \mathsf{Char}\\
\mathsf{symbol}_4\ \mathsf{x} = \mathsf{satisfy}_4\ (== \mathsf{x})
\end{array}
$$

Choice with results

We can easily combine parsers with $(<|>)$ if they have the same result type:

$$(<|>) :: \mathsf{Parser}_4\ \mathsf{a} \to \mathsf{Parser}_4\ \mathsf{a} \to \mathsf{Parser}_4\ \mathsf{a}$$
$$(\mathsf{p} <|> \mathsf{q})\ \mathsf{xs} = \mathsf{p}\ \mathsf{xs} +\!\!+\ \mathsf{q}\ \mathsf{xs}$$

(Definition is unchanged.)

Choice with results

We can easily combine parsers with $(<|>)$ if they have the same result type:

$$(<|>) :: \mathsf{Parser}_4\ a \rightarrow \mathsf{Parser}_4\ a \rightarrow \mathsf{Parser}_4\ a$$
$$(p <|> q)\ xs = p\ xs +\!\!+\ q\ xs$$

(Definition is unchanged.)

Question

What if the parsers have different result types?

Universiteit Utrecht

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

Chaniging the result of a parser

We define a new function

$$(<\$>) :: (a \rightarrow b) \rightarrow \mathrm{Parser}_4\ a \rightarrow \mathrm{Parser}_4\ b$$
$$(f <\$> p)\ xs = [(f\ r, ys) \mid (r, ys) \leftarrow p\ xs]$$

that changes the results of a parser. It has the same priority as $(<\!\!*\!\!>)$:

$$\textbf{infixl}\ 4\ (<\$>)$$

This function is similar to map for lists:

$$\mathrm{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

Example: bits

> $\text{Bit} \rightarrow 0 \mid 1$

Example: bits

$\text{Bit} \rightarrow 0 \mid 1$

$\textbf{data}\ \text{Bit} = \text{Zero} \mid \text{One}$

Example: bits

$\text{Bit} \rightarrow 0 \mid 1$

data Bit = Zero | One

Parser:

```
bit :: Parser₄ Bit
bit  =     symbol₄ '0'
     <|>   symbol₄ '1'
```

Does not produce a Bit without adapting the results.

Example: bits

$\text{Bit} \rightarrow 0 \mid 1$

data Bit = Zero | One

Parser:

```
bit :: Parser4 Bit
bit  =  const Zero <$> symbol4 '0'
    <|> const One  <$> symbol4 '1'
```

Does not produce a Bit without adapting the results.

Example: bits

$\mathsf{Bit} \rightarrow 0 \mid 1$

data Bit = Zero | One

Parser:

```
bit :: Parser₄ Bit
bit   =   const Zero  <$>  symbol₄ '0'
     <|> const One   <$>  symbol₄ '1'
```

Does not produce a Bit without adapting the results.

Recall:

```
const :: a → b → a
const x y = x
```

Combining results

How does $(<\!\!*\!\!>)$ work in the presence of results?

Combining results

How does $(\lt\!\ast\!\gt)$ work in the presence of results?

One option is to return a pair of results:

$$
\begin{aligned}
&(\lt\!\ast\!\gt) :: \mathsf{Parser}_4\ \mathsf{a} \to \mathsf{Parser}_4\ \mathsf{b} \to \mathsf{Parser}_4\ (\mathsf{a}, \mathsf{b}) \\
&(\mathsf{p} \lt\!\ast\!\gt \mathsf{q})\ \mathsf{xs} = [((\mathsf{r}, \mathsf{s}), \mathsf{zs}) \mid (\mathsf{r}, \mathsf{ys}) \leftarrow \mathsf{p}\ \mathsf{xs}, (\mathsf{s}, \mathsf{zs}) \leftarrow \mathsf{q}\ \mathsf{ys}]
\end{aligned}
$$

Combining results

How does $(<\!\!*\!\!>)$ work in the presence of results?

One option is to return a pair of results:

$$(<\!\!*\!\!>) :: \mathsf{Parser}_4\ a \rightarrow \mathsf{Parser}_4\ b \rightarrow \mathsf{Parser}_4\ (a, b)$$
$$(p <\!\!*\!\!> q)\ xs = [((r, s), zs) \mid (r, ys) \leftarrow p\ xs, (s, zs) \leftarrow q\ ys]$$

Unfortunately, this is unconvenient for long sequences:

$$\mathsf{letter} <\!\!*\!\!> \mathsf{letter} <\!\!*\!\!> \mathsf{letter} <\!\!*\!\!> \mathsf{letter} <\!\!*\!\!> \mathsf{letter}$$
$$:: \mathsf{Parser}\ ((((\mathsf{Char}, \mathsf{Char}), \mathsf{Char}), \mathsf{Char}), \mathsf{Char})$$

Combining results

How does $(<\!\!*\!\!>)$ work in the presence of results?

One option is to return a pair of results:

$$(<\!\!*\!\!>) :: \mathsf{Parser}_4\ \mathsf{a} \rightarrow \mathsf{Parser}_4\ \mathsf{b} \rightarrow \mathsf{Parser}_4\ (\mathsf{a}, \mathsf{b})$$
$$(\mathsf{p} <\!\!*\!\!> \mathsf{q})\ \mathsf{xs} = [((\mathsf{r}, \mathsf{s}), \mathsf{zs}) \mid (\mathsf{r}, \mathsf{ys}) \leftarrow \mathsf{p}\ \mathsf{xs}, (\mathsf{s}, \mathsf{zs}) \leftarrow \mathsf{q}\ \mathsf{ys}]$$

Unfortunately, this is unconvenient for long sequences:

$$\mathsf{letter} <\!\!*\!\!> \mathsf{letter} <\!\!*\!\!> \mathsf{letter} <\!\!*\!\!> \mathsf{letter} <\!\!*\!\!> \mathsf{letter}$$
$$:: \mathsf{Parser}\ ((((\mathsf{Char}, \mathsf{Char}), \mathsf{Char}), \mathsf{Char}), \mathsf{Char})$$

We have to pattern match on these nested pairs in a subsequent function applied via $(<\$>)$. But since we are applying $(<\$>)$ anyway, there is a better option.

Universiteit Utrecht

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

Combining results – contd.

We use the following definition instead:

$$(\mathbin{<\!\!*\!\!>}) :: \mathsf{Parser}_4\ (\mathsf{a} \rightarrow \mathsf{b}) \rightarrow \mathsf{Parser}_4\ \mathsf{a} \rightarrow \mathsf{Parser}_4\ \mathsf{b}$$
$$(\mathsf{p} \mathbin{<\!\!*\!\!>} \mathsf{q})\ \mathsf{xs} = [(\mathsf{f}\ \mathsf{r}, \mathsf{zs}) \mid (\mathsf{f}, \mathsf{ys}) \leftarrow \mathsf{p}\ \mathsf{xs}, (\mathsf{r}, \mathsf{zs}) \leftarrow \mathsf{q}\ \mathsf{ys}]$$

Now $(\mathbin{<\!\!*\!\!>})$ is like function application lifted to parsers.

Example: Dutch zip codes

ZipCode → Digit Digit Digit Digit Letter Letter

Example: Dutch zip codes

> ZipCode → Digit Digit Digit Digit Letter Letter

Haskell abstract syntax:

```
data ZipCode = Zip Digit Digit Digit Digit Letter Letter
type Digit   = Char   -- convenient, but not precise
type Letter  = Char   -- convenient, but not precise
```

Example: Dutch zip codes

$$\text{ZipCode} \rightarrow \text{Digit Digit Digit Digit Letter Letter}$$

Haskell abstract syntax:

```
data ZipCode = Zip Digit Digit Digit Digit Letter Letter
type Digit   = Char   -- convenient, but not precise
type Letter  = Char   -- convenient, but not precise
```

Parser:

```
zipCode :: Parser₄ ZipCode
zipCode = Zip <$> digit <*> digit <*> digit <*> digit
              <*> letter <*> letter
```

Why is this function type-correct?

Example: Dutch zip codes – contd.

Both operators associate to the left, so zipCode is in fact:

$$\text{zipCode} = ((((((\text{Zip} <\$> \text{digit}) <*> \text{digit}) <*> \text{digit}) <*> \text{digit}) <*> \text{letter}) <*> \text{letter})$$

Example: Dutch zip codes – contd.

Both operators associate to the left, so zipCode is in fact:

$$\begin{aligned} zipCode = ((((((Zip \mathbin{<\$>} digit) \mathbin{<*>} digit) \mathbin{<*>} digit) \mathbin{<*>} digit) \\ \mathbin{<*>} letter) \mathbin{<*>} letter) \end{aligned}$$

Now consider the types:

$$\begin{array}{l} Zip \\ \quad :: Digit \to Digit \to Digit \to Digit \to Letter \to Letter \to ZipCode \\ Zip \mathbin{<\$>} digit \\ \quad :: Parser_4\ (Digit \to Digit \to Digit \to Letter \to Letter \to ZipCode) \\ (Zip \mathbin{<\$>} digit) \mathbin{<*>} digit \\ \quad :: Parser_4\ (Digit \to Digit \to Letter \to Letter \to ZipCode) \\ \ldots \\ zipCode \\ \quad :: Parser_4\ ZipCode \end{array}$$

Universiteit Utrecht

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

Are we done yet?

With Parser_4, we have completed all the hard work. All that remains are some final touches.

Other symbol types

Nothing in our parser design depends on the fact that we are working on strings. All we need is a list of symbols as an input, so we can move to

$$\textbf{type}\ \mathsf{Parser}_5\ \mathsf{s}\ \mathsf{r} = [\mathsf{s}] \to [(\mathsf{r}, [\mathsf{s}])]$$

Some of the types change (but not the implementation), for example:

$$\mathsf{satisfy}_5 :: (\mathsf{s} \to \mathsf{Bool}) \to \mathsf{Parser}_5\ \mathsf{s}\ \mathsf{s}$$

Are we done yet? – contd.

Not in the lecture notes, but recommended:

Making parsers abstract

It is better to hide the implementation of parsers:

$$\textbf{newtype}\ \mathsf{Parser}_6\ \mathsf{s}\ \mathsf{r} = \mathsf{Parser}\ ([\mathsf{s}] \rightarrow [(\mathsf{r}, [\mathsf{s}])])$$
$$\mathsf{runParser}\ (\mathsf{Parser}\ \mathsf{p}) = \mathsf{p}$$

Allows us to replace the implementation with a better one later.

Summary

Despite the long development, the final version is still simple:

$$\textbf{newtype}\ \mathsf{Parser}_6\ \mathsf{s}\ \mathsf{r} = \mathsf{Parser}\ ([\mathsf{s}] \rightarrow [(\mathsf{r}, [\mathsf{s}])])$$

We have combinators representing the constructs of grammars:

- parsing individual symbols,
- choice, sequence,
- empty strings,
- repetition.

Furthermore, we can produce results and modify intermediate results.

Next lecture

- Summary of the interface of the parser combinators.
- Constructing parsers from grammars.
- Pitfalls and limitations.
- Grammar transformations.