Universiteit Utrecht

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

Talen en Compilers

2023 - 2024

David van Balen

Department of Information and Computing Sciences
Utrecht University

2024-11-25

5. Parser combinators (iii)

This lecture

Parser combinators (iii)

- Parser Combinators: recap
- Parser Combinators: new primitives
- Parser Combinators: new abstractions
- Grammar transformations
- Operators

5.1 Parser Combinators: recap

Library

```
module ParseLib (Parser, parse)
data Parser s a = Parser { runParser :: [s] → [(a, [s])] }
parse = runParser
```

Recap

Primitive parser combinators:

```
type Parser s r = [s] → [(r, [s])]
epsilon :: Parser s ()
empty   :: Parser s a
(<|>)   :: Parser s a → Parser s a → Parser s a
(<*>)   :: Parser s (a → b) → Parser s a → Parser s b
(<$>)   :: (a → b) → Parser s a → Parser s b
satisfy :: (s → Bool) → Parser s s
```

Recap

Derived parser combinators:

$$
\begin{array}{ll}
\textbf{type}\ \text{Parser s r} = \text{(hidden)} & \\
(<\$) & :: \text{a} \rightarrow \text{Parser s b} \rightarrow \text{Parser s a} \\
(<*) & :: \text{Parser s a} \rightarrow \text{Parser s b} \rightarrow \text{Parser s a} \\
(*>) & :: \text{Parser s a} \rightarrow \text{Parser s b} \rightarrow \text{Parser s b} \\
\text{succeed} & :: \text{a} \rightarrow \text{Parser s a} \\
\text{symbol} & :: \text{Eq s} \Rightarrow \text{s} \rightarrow \text{Parser s s}
\end{array}
$$

5.2 Parser Combinators: new primitives

Choice, again

$$(\lt|\gt) :: \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s\ a}$$
$$(\mathsf{p} \lt|\gt \mathsf{q}) = \lambda\mathsf{xs} \rightarrow \mathsf{p\ xs} +\!\!+\ \mathsf{q\ xs}$$

Choice, again

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

$(\ll|>) :: \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s\ a}$
$(\mathsf{p} \ll|> \mathsf{q}) = \lambda \mathsf{xs} \rightarrow \mathbf{if}\ \mathsf{null\ (p\ xs)}\ \mathbf{then}\ \mathsf{q\ xs}\ \mathbf{else}\ \mathsf{p\ xs}$

Another primitive combinator

We defined guard as a primitive last lecture:

```
guard :: (a → Bool) → Parser s a → Parser s a
guard cond parser input =
  [(result, tail) | (result, tail) ← parser input
  , cond result]
```

Another primitive combinator

We defined guard as a primitive last lecture:

$$
\begin{aligned}
&\mathsf{guard} :: (\mathsf{a} \to \mathsf{Bool}) \to \mathsf{Parser\ s\ a} \to \mathsf{Parser\ s\ a} \\
&\mathsf{guard\ cond\ parser\ input} = \\
&\quad [(\mathsf{result}, \mathsf{tail}) \mid (\mathsf{result}, \mathsf{tail}) \leftarrow \mathsf{parser\ input} \\
&\quad , \mathsf{cond\ result}]
\end{aligned}
$$

We can do better!

Another primitive combinator – contd.

We introduce ($\ggg\!=$) – pronounced "bind":

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

Another primitive combinator – contd.

We introduce ($\gg\!=$) – pronounced "bind":

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

Now:

$$
\begin{aligned}
&\mathsf{guard} :: (\mathsf{a} \to \mathsf{Bool}) \to \mathsf{Parser\ s\ a} \to \mathsf{Parser\ s\ a}\\
&\mathsf{guard\ cond\ parser} = \mathsf{parser} \gg\!= \lambda \mathsf{a} \to\\
&\quad \textbf{if}\ \mathsf{cond\ a}\ \textbf{then}\ \mathsf{succeed\ a}\ \textbf{else}\ \mathsf{empty}
\end{aligned}
$$

Another primitive combinator – contd.

$$
\begin{array}{l}
\mathsf{guard} :: (\mathsf{a} \rightarrow \mathsf{Bool}) \rightarrow \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s\ a} \\
\mathsf{guard\ cond\ parser} = \mathsf{parser} \ggg\!= \lambda \mathsf{a} \rightarrow \\
\quad \mathbf{if}\ \mathsf{cond\ a}\ \mathbf{then}\ \mathsf{succeed\ a}\ \mathbf{else}\ \mathsf{empty}
\end{array}
$$

Another primitive combinator – contd.

```
guard :: (a → Bool) → Parser s a → Parser s a
guard cond parser = parser >>= λa →
   if cond a then succeed a else empty
```

```
guard :: (a → Bool) → Parser s a → Parser s a
guard cond parser = do
   a ← parser
   if cond a then return a else empty
```

Another primitive combinator – example

Parse a number, then parse that many lines:

```
3
Hello
World
!
asdf
```

Another primitive combinator – example

Parse a number, then parse that many lines:

```
3
Hello
World
!
asdf
```

```
parseNLines :: Parser Char [String]
parseNLines = do
  n <- natural
  _ <- symbol '\n'
  sequence $ replicate n parseLine
    where parseLine = many (satisfy (/= '\n')) <* symbol '\n'
```

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 and alternative functors, you have probably seen monads before.

Monads

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

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

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

5.3 Parser Combinators: new abstractions

Option

```
option :: Parser s a → a → Parser s a
option p def = p <|> succeed def
```

Lists

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

Lists

$\text{many} :: \text{Parser s a} \rightarrow \text{Parser s [a]}$
$\text{many p} = (:) <\$> \text{p} <\!\!*\!\!> \text{many p} <|> \text{succeed} []$

$\text{some} :: \text{Parser s a} \rightarrow \text{Parser s [a]}$ `-- also called` many_1
$\text{some p} = (:) <\$> \text{p} <\!\!*\!\!> \text{many p}$

Lists

$\text{many} :: \text{Parser s a} \rightarrow \text{Parser s [a]}$
$\text{many p} = (:) <\$> \text{p} <\!\!*\!\!> \text{many p} <|> \text{succeed} \, []$

$\text{some} :: \text{Parser s a} \rightarrow \text{Parser s [a]}$ `-- also called` many_1
$\text{some p} = (:) <\$> \text{p} <\!\!*\!\!> \text{many p}$

$\text{listOf} :: \text{Parser s a} \rightarrow \text{Parser s b} \rightarrow \text{Parser s [a]}$
$\text{listOf p s} = (:) <\$> \text{p} <\!\!*\!\!> \text{many (s} *\!\!> \text{p)}$

Greedy lists

$$\text{greedy} :: \text{Parser s a} \rightarrow \text{Parser s [a]}$$
$$\text{greedy p} = (:) <\$> \text{p} <*> \text{greedy p} \ll|> \text{succeed} []$$

Greedy lists

$$\text{greedy} :: \text{Parser s a} \to \text{Parser s [a]}$$
$$\text{greedy p} = (:) <\$> \text{p} <*> \text{greedy p} \ll\!|\!> \text{succeed} \; []$$

$$\text{greedy}_1 :: \text{Parser s a} \to \text{Parser s [a]} \quad \texttt{-- also called many}_1$$
$$\text{greedy}_1 \; \text{p} = (:) <\$> \text{p} <*> \text{greedy p}$$

Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

5.4 Grammar transformations

From Grammar to Parser

$$
\begin{aligned}
S &\rightarrow A \\
S &\rightarrow B \\
A &\rightarrow \mathtt{c} \\
A &\rightarrow AA \\
B &\rightarrow \mathtt{d} \\
B &\rightarrow BB
\end{aligned}
$$

From Grammar to Parser

$$\begin{aligned} S &\rightarrow A \\ S &\rightarrow B \\ A &\rightarrow \mathtt{c} \\ A &\rightarrow AA \\ B &\rightarrow \mathtt{d} \\ B &\rightarrow BB \end{aligned}$$

$$\begin{aligned} S &\rightarrow A \\ S &\rightarrow B \\ A &\rightarrow \mathtt{c} \end{aligned}$$

From Grammar to Parser

$$
\begin{aligned}
S &\rightarrow A \\
S &\rightarrow B \\
A &\rightarrow \mathtt{c} \\
A &\rightarrow AA \\
B &\rightarrow \mathtt{d} \\
B &\rightarrow BB
\end{aligned}
$$

$$
\begin{aligned}
S &\rightarrow A \\
S &\rightarrow B \\
A &\rightarrow \mathtt{c}
\end{aligned}
$$

$$
\begin{aligned}
S &\rightarrow A \\
S &\rightarrow B \\
A &\rightarrow \mathtt{c} \\
B &\rightarrow \varepsilon
\end{aligned}
$$

Removing duplicate productions

Example:

$$A \to u \mid u \mid v$$

can be transformed into

$$A \to u \mid v$$

Removing duplicate productions

Example:

> $A \rightarrow u \mid u \mid v$

can be transformed into

> $A \rightarrow u \mid v$

Parser:

> a = u <|> u <|> v

becomes

> a = u <|> v

Left factoring

Example:

$$A \rightarrow xy \mid xz \mid v$$

can be transformed into

$$A \rightarrow xQ \mid v$$
$$Q \rightarrow y \mid z$$

Left factoring

Example:

$A \to xy \mid xz \mid v$

can be transformed into

$A \to xQ \mid v$
$Q \to y \mid z$

Parser:

$a = x <*> y <|> x <*> z <|> v$

becomes

$a = x <*> q <|> v$
$q = y <|> z$

Left factoring

Example:

$A \rightarrow xy \mid xz \mid v$

can be transformed into

$A \rightarrow xQ \mid v$
$Q \rightarrow y \mid z$

Parser:

```
a = x <*> y <|> x <*> z <|> v
```

becomes

```
a = x <*> q <|> v
q = y <|> z
```

- Note that x can be an arbitrarily long sequence of symbols. The longer the sequence, and the more alternatives have the same prefix, the more useful this transformation is.
- What is the effect on the parsers?

Universiteit Utrecht

Left factoring – contd.

$S \to xSy \mid xSx \mid x$

Left factoring – contd.

$$\mathsf{S} \rightarrow \mathtt{x}\mathsf{S}\mathtt{y} \mid \mathtt{x}\mathsf{S}\mathtt{x} \mid \mathtt{x}$$

$$\begin{aligned} \mathsf{S} &\rightarrow \mathtt{x}\mathsf{T} \\ \mathsf{T} &\rightarrow \mathsf{S}\mathtt{y} \mid \mathsf{S}\mathtt{x} \mid \varepsilon \end{aligned}$$

Left factoring – contd.

$$S \to xSy \mid xSx \mid x$$

$$\begin{aligned} S &\to xT \\ T &\to Sy \mid Sx \mid \varepsilon \end{aligned}$$

$$\begin{aligned} S &\to xT \\ T &\to SU \mid \varepsilon \\ U &\to y \mid x \end{aligned}$$

Left recursion

A production is called **left-recursive** if the right hand side **starts** with the nonterminal of the left hand side.

Example:

> $A \rightarrow Az$

Left recursion

A production is called **left-recursive** if the right hand side **starts** with the nonterminal of the left hand side.

Example:

$$A \rightarrow Az$$

A grammar is called left-recursive if $A \Rightarrow^{+} Az$ for some nonterminal A of the grammar.

Left recursion

A production is called **left-recursive** if the right hand side **starts** with the nonterminal of the left hand side.

Example:

> A $\rightarrow$ Az

A grammar is called left-recursive if A $\Rightarrow^{+}$ Az for some nonterminal A of the grammar.

Question

Can a grammar be left-recursive if it does not have any left-recursive productions?

Left recursion

A production is called **left-recursive** if the right hand side **starts** with the nonterminal of the left hand side.

Example:

> $A \rightarrow Az$

A grammar is called left-recursive if $A \Rightarrow^{+} Az$ for some nonterminal A of the grammar.

Question

Can a grammar be left-recursive if it does not have any left-recursive productions?

Yes, grammars can be indirectly left-recursive.

Left recursion and parsers

The production

> $A \rightarrow Az$

corresponds to a parser

> $a = a <*> z$

What happens here?

Left recursion and parsers

The production

> $A \rightarrow Az$

corresponds to a parser

> a = a <*> z

What happens here?

- The parser loops!
- Removing left recursion is essential for a combinator parser.

Removing left recursion

Transforming a (directly) left-recursive nonterminal A such that the left recursion is removed is relatively simple.

Removing left recursion

Transforming a (directly) left-recursive nonterminal A such that the left recursion is removed is relatively simple.

First, split the productions for A into left-recursive and others:

$$\begin{aligned}&\mathsf{A} \rightarrow \mathsf{A}\mathsf{x}_1 \mid \mathsf{A}\mathsf{x}_2 \mid \ldots \mid \mathsf{A}\ \mathsf{x}_\mathsf{n}\\ &\mathsf{A} \rightarrow \mathsf{y}_1 \mid \mathsf{y}_2 \mid \ldots \mid \mathsf{y}_\mathsf{m} \qquad \{\text{-(none of the } \mathsf{y}_\mathsf{i} \text{ start with A) -}\}\end{aligned}$$

Removing left recursion

Transforming a (directly) left-recursive nonterminal A such that the left recursion is removed is relatively simple.

First, split the productions for A into left-recursive and others:

$$
\begin{array}{l}
A \rightarrow Ax_1 \mid Ax_2 \mid \ldots \mid A\ x_n \\
A \rightarrow y_1 \mid y_2 \mid \ldots \mid y_m \qquad \{\text{-(none of the } y_i \text{ start with A) -}\}
\end{array}
$$

This grammar can be transformed to:

$$
\begin{array}{l}
A \rightarrow y_1 Z \mid y_2 Z \mid \ldots \mid y_m Z \\
Z \rightarrow \varepsilon \mid x_1 Z \mid x_2 Z \mid \ldots \mid x_n Z
\end{array}
$$

Example: Removing left recursion

Consider:

$S \rightarrow SS$
$S \rightarrow \mathtt{s}$

One left-recursive production, one other – already split.

Universiteit Utrecht

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

Example: Removing left recursion

Consider:

$$\begin{aligned} &\mathsf{S} \rightarrow \mathsf{SS} \\ &\mathsf{S} \rightarrow \mathtt{s} \end{aligned}$$

One left-recursive production, one other – already split.

Applying the transformation yields:

$$\begin{aligned} &\mathsf{S} \rightarrow \mathtt{s}\mathsf{Z} \\ &\mathsf{Z} \rightarrow \varepsilon \mid \mathsf{SZ} \end{aligned}$$

5.5 Operators

Operator chains

Consider

$$\begin{aligned} \mathsf{E} &\rightarrow \mathsf{E}\ \mathsf{O}\ \mathsf{E} \mid \mathsf{Nat} \\ \mathsf{O} &\rightarrow \texttt{+} \mid \texttt{-} \end{aligned}$$

Universiteit Utrecht

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

Operator chains

Consider

E $\rightarrow$ E O E | Nat
O $\rightarrow$ + | -

'–' is not an associative operator. It is usually defined as associating to the left (i.e. **left-associative**).

Operator chains

Consider

$$\begin{aligned} E &\rightarrow E\ O\ E \mid Nat \\ O &\rightarrow + \mid - \end{aligned}$$

'–' is not an associative operator. It is usually defined as associating to the left (i.e. **left-associative**).

We inline O and remove it to obtain an abstract syntax:

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

Operator chains

Consider

$$\begin{aligned} E &\rightarrow E\ O\ E \mid Nat \\ O &\rightarrow + \mid - \end{aligned}$$

'–' is not an associative operator. It is usually defined as associating to the left (i.e. **left-associative**).

We inline O and remove it to obtain an abstract syntax:

$$\textbf{data}\ E = Plus\ E\ E \mid Minus\ E\ E \mid Nat\ Int$$

Operator chains – contd.

We would like to parse

> 1+2−3+4

as

> ((Nat 1 ‘Plus‘ Nat 2) ‘Minus‘ Nat 3) ‘Plus‘ Nat 4

Operator chains – contd.

We want:

> ((Nat 1 ‘Plus‘ Nat 2) ‘Minus‘ Nat 3) ‘Plus‘ Nat 4

Operator chains – contd.

We want:

$((\text{Nat } 1 \text{ `Plus` Nat } 2) \text{ `Minus` Nat } 3) \text{ `Plus` Nat } 4$

What does the following evaluate to?

$\text{foldl (flip (\$)) (Nat } 1)$
$\quad [(\text{`Plus` Nat } 2), (\text{`Minus` Nat } 3), (\text{`Plus` Nat } 4)]$

Operator chains – contd.

We want:

$((\mathsf{Nat}\ 1\ \text{`Plus`}\ \mathsf{Nat}\ 2)\ \text{`Minus`}\ \mathsf{Nat}\ 3)\ \text{`Plus`}\ \mathsf{Nat}\ 4$

What does the following evaluate to?

$$\begin{array}{l}\mathsf{foldl}\ (\mathsf{flip}\ (\$))\ (\mathsf{Nat}\ 1) \\ \qquad [(\text{`Plus`}\ \mathsf{Nat}\ 2), (\text{`Minus`}\ \mathsf{Nat}\ 3), (\text{`Plus`}\ \mathsf{Nat}\ 4)]\end{array}$$

We can obtain this result as follows:

$$\begin{array}{l}\mathsf{chainl} :: \mathsf{Parser\ s\ a} \to \mathsf{Parser\ s}\ (\mathsf{a} \to \mathsf{a} \to \mathsf{a}) \to \mathsf{Parser\ s\ a} \\ \mathsf{chainl\ p\ s} = \mathsf{foldl}\ (\mathsf{flip}\ (\$)) <\$> \mathsf{p} <*> \mathsf{many}\ (\mathsf{flip} <\$> \mathsf{s} <*> \mathsf{p})\end{array}$$

Operator chains – contd.

We want:

$$((\text{Nat } 1 \text{ `Plus` Nat } 2) \text{ `Minus` Nat } 3) \text{ `Plus` Nat } 4$$

What does the following evaluate to?

$$\text{foldl (flip (\$)) (Nat } 1)$$
$$[(\text{`Plus` Nat } 2), (\text{`Minus` Nat } 3), (\text{`Plus` Nat } 4)]$$

We can obtain this result as follows:

$$\text{chainl} :: \text{Parser s a} \to \text{Parser s } (\text{a} \to \text{a} \to \text{a}) \to \text{Parser s a}$$
$$\text{chainl p s} = \text{foldl (flip (\$))} <\$> \text{p} <*> \text{many (flip} <\$> \text{s} <*> \text{p)}$$

$$\text{e} = \text{chainl (Nat} <\$> \text{natural) o}$$
$$\text{o} = \text{Plus} <\$ \text{ symbol } \text{'+'} <|> \text{Minus} <\$ \text{ symbol } \text{'-'}$$

Chain combinators

There are combinators for left-associative and right-associative chains:

$$\mathsf{chainl} :: \mathsf{Parser}\ s\ a \rightarrow \mathsf{Parser}\ s\ (a \rightarrow a \rightarrow a) \rightarrow \mathsf{Parser}\ s\ a$$
$$\mathsf{chainr} :: \mathsf{Parser}\ s\ a \rightarrow \mathsf{Parser}\ s\ (a \rightarrow a \rightarrow a) \rightarrow \mathsf{Parser}\ s\ a$$

```
chainl p s =
  foldl (flip ($)) <$> p <*> many (flip <$> s <*> p)
chainr p s =
  flip (foldr ($)) <$> many (flip ($) <$> p <*> s) <*> p
```

Chain combinators

There are combinators for left-associative and right-associative chains:

$$\mathsf{chainl} :: \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s\ (a \rightarrow a \rightarrow a)} \rightarrow \mathsf{Parser\ s\ a}$$
$$\mathsf{chainr} :: \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s\ (a \rightarrow a \rightarrow a)} \rightarrow \mathsf{Parser\ s\ a}$$

```
chainl p s =
  foldl (flip ($)) <$> p <*> many (flip <$> s <*> p)
chainr p s =
  flip (foldr ($)) <$> many (flip ($) <$> p <*> s) <*> p
```


Universiteit Utrecht

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

Chain combinators

There are combinators for left-associative and right-associative chains:

$$\mathsf{chainl} :: \mathsf{Parser\ s\ a} \to \mathsf{Parser\ s}\ (\mathsf{a} \to \mathsf{a} \to \mathsf{a}) \to \mathsf{Parser\ s\ a}$$
$$\mathsf{chainr} :: \mathsf{Parser\ s\ a} \to \mathsf{Parser\ s}\ (\mathsf{a} \to \mathsf{a} \to \mathsf{a}) \to \mathsf{Parser\ s\ a}$$

```
chainl p s =
   foldl (flip ($)) <$> p <*> many (flip <$> s <*> p)
chainr p s =
   flip (foldr ($)) <$> many (flip ($) <$> p <*> s) <*> p
```

Use chainl and chainr for some of the most common occurrences of left recursion in grammars.

Operator priorities

Consider:

$$
\begin{aligned}
E &\rightarrow E + E \\
E &\rightarrow E - E \\
E &\rightarrow E * E \\
E &\rightarrow (E) \\
E &\rightarrow \mathsf{Nat}
\end{aligned}
$$

Operator priorities

Consider:

$$
\begin{aligned}
E &\rightarrow E + E\\
E &\rightarrow E - E\\
E &\rightarrow E * E\\
E &\rightarrow (E)\\
E &\rightarrow \text{Nat}
\end{aligned}
$$

This is a typical grammar for expressions with operators.

For the same reasons as before, it is ambiguous.

Operator priorities

Consider:

$$E \to E + E$$
$$E \to E - E$$
$$E \to E * E$$
$$E \to (E)$$
$$E \to Nat$$

This is a typical grammar for expressions with operators.

For the same reasons as before, it is ambiguous.

Given the priorities of the operators and their associativity, we can transform this grammar such that the ambiguity is removed.

Operator priorities – contd.

The basic idea is to associate operators of different priorities with different non-terminals.

Operator priorities – contd.

The basic idea is to associate operators of different priorities with different non-terminals.

For each priority level i, we get

$E_i \rightarrow E_i\ Op_i\ E_{i+1} \mid E_{i+1}$ (for left-associative operators)

or

$E_i \rightarrow E_{i+1}\ Op_i\ E_i \mid E_{i+1}$ (for right-associative operators)

or

$E_i \rightarrow E_{i+1}\ Op_i\ E_{i+1} \mid E_{i+1}$ (for non-associative operators)

Operator priorities – contd.

The basic idea is to associate operators of different priorities with different non-terminals.

For each priority level i, we get

$\mathsf{E_i} \to \mathsf{E_i}\ \mathsf{Op_i}\ \mathsf{E_{i+1}} \mid \mathsf{E_{i+1}}$ (for left-associative operators)

or

$\mathsf{E_i} \to \mathsf{E_{i+1}}\ \mathsf{Op_i}\ \mathsf{E_i} \mid \mathsf{E_{i+1}}$ (for right-associative operators)

or

$\mathsf{E_i} \to \mathsf{E_{i+1}}\ \mathsf{Op_i}\ \mathsf{E_{i+1}} \mid \mathsf{E_{i+1}}$ (for non-associative operators)

The highest level contains the remaining productions.

Operator priorities – contd.

The basic idea is to associate operators of different priorities with different non-terminals.

For each priority level i, we get

$E_i \to E_i\ Op_i\ E_{i+1} \mid E_{i+1}$ (for left-associative operators)

or

$E_i \to E_{i+1}\ Op_i\ E_i \mid E_{i+1}$ (for right-associative operators)

or

$E_i \to E_{i+1}\ Op_i\ E_{i+1} \mid E_{i+1}$ (for non-associative operators)

The highest level contains the remaining productions.

All forms of brackets point to the outer (lowest) level of expressions.

Operator priorities – contd.

Applied to

$E \rightarrow E + E$
$E \rightarrow E - E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow Nat$

we obtain:

$E_1 \rightarrow E_1\ Op_1\ E_2 \mid E_2$
$E_2 \rightarrow E_2\ Op_2\ E_3 \mid E_3$
$E_3 \rightarrow (\ E_1\ ) \mid Nat$
$Op_1 \rightarrow + \mid -$
$Op_2 \rightarrow *$

Parsers for operator expressions

Since the abstract syntax tree structure makes the nesting explicit, it typically makes sense to derive the Haskell datatype from the ambiguous grammar:

$$
\begin{aligned}
E &\rightarrow E + E \\
E &\rightarrow E - E \\
E &\rightarrow E * E \\
E &\rightarrow (E) \\
E &\rightarrow Nat
\end{aligned}
$$

Parsers for operator expressions

Since the abstract syntax tree structure makes the nesting explicit, it typically makes sense to derive the Haskell datatype from the ambiguous grammar:

$$
\begin{aligned}
E &\rightarrow E + E \\
E &\rightarrow E - E \\
E &\rightarrow E * E \\
E &\rightarrow (\ E\ ) \\
E &\rightarrow Nat
\end{aligned}
$$

```haskell
data E = Plus  E E
       | Minus E E
       | Times E E
       | Parens E
       | Nat
```

Parsers for operator expressions

Since the abstract syntax tree structure makes the nesting explicit, it typically makes sense to derive the Haskell datatype from the ambiguous grammar:

$$
\begin{aligned}
E &\rightarrow E + E\\
E &\rightarrow E - E\\
E &\rightarrow E * E\\
E &\rightarrow (E)\\
E &\rightarrow \mathrm{Nat}
\end{aligned}
$$

```
data E = Plus  E E
       | Minus E E
       | Times E E

       | Nat
```

Parsers for operator expressions

Since the abstract syntax tree structure makes the nesting explicit, it typically makes sense to derive the Haskell datatype from the ambiguous grammar:

$$
\begin{aligned}
\mathsf{E} &\rightarrow \mathsf{E}\ \texttt{+}\ \mathsf{E}\\
\mathsf{E} &\rightarrow \mathsf{E}\ \texttt{-}\ \mathsf{E}\\
\mathsf{E} &\rightarrow \mathsf{E}\ \texttt{*}\ \mathsf{E}\\
\mathsf{E} &\rightarrow \texttt{(}\ \mathsf{E}\ \texttt{)}\\
\mathsf{E} &\rightarrow \mathsf{Nat}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{data}\ \mathsf{E} = &\ \mathsf{Plus}\quad \mathsf{E}\ \mathsf{E}\\
\mid &\ \mathsf{Minus}\ \mathsf{E}\ \mathsf{E}\\
\mid &\ \mathsf{Times}\ \mathsf{E}\ \mathsf{E}\\
\\
\mid &\ \mathsf{Nat}
\end{aligned}
$$

We can now use chainl and chainr again for each of the levels.

Parsers for operator expressions – contd.

$$
\begin{aligned}
\mathsf{E}_1 &\rightarrow \mathsf{E}_1\ \mathsf{Op}_1\ \mathsf{E}_2 \mid \mathsf{E}_2\\
\mathsf{E}_2 &\rightarrow \mathsf{E}_2\ \mathsf{Op}_2\ \mathsf{E}_3 \mid \mathsf{E}_3\\
\mathsf{E}_3 &\rightarrow (\ \mathsf{E}_1\ ) \mid \mathsf{Nat}\\
\mathsf{Op}_1 &\rightarrow \texttt{+} \mid \texttt{-}\\
\mathsf{Op}_2 &\rightarrow \texttt{*}
\end{aligned}
$$

```
data E = Plus  E E
       | Minus E E
       | Times E E
       | Nat   Int
```


Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Parsers for operator expressions – contd.

$$
\begin{array}{lcl}
E_1 & \rightarrow & E_1\ Op_1\ E_2 \mid E_2 \\
E_2 & \rightarrow & E_2\ Op_2\ E_3 \mid E_3 \\
E_3 & \rightarrow & (\ E_1\ ) \mid \mathsf{Nat} \\
Op_1 & \rightarrow & \texttt{+} \mid \texttt{-} \\
Op_2 & \rightarrow & \texttt{*}
\end{array}
$$

```haskell
data E = Plus  E E
       | Minus E E
       | Times E E
       | Nat   Int
```

Parser:

```haskell
e1, e2, e3 :: Parser Char E
e1  = chainl e2 op1
e2  = chainl e3 op2
e3  = parenthesised e1 <|> Nat <$> natural
op1, op2 :: Parser Char (E -> E -> E)
op1 = Plus  <$ symbol '+' <|> Minus <$ symbol '-'
op2 = Times <$ symbol '*'
```

A general operator parser

We can abstract even further from this pattern:

$$\textbf{type}\ \mathsf{Op\ a} = (\mathsf{Char}, \mathsf{a} \to \mathsf{a} \to \mathsf{a})$$

$$\mathsf{gen} :: [\mathsf{Op\ a}] \to \mathsf{Parser\ Char\ a} \to \mathsf{Parser\ Char\ a}$$

$$\mathsf{gen\ ops\ p} =$$

$$\quad \mathsf{chainl\ p\ (choice\ (map\ } (\lambda(\mathsf{s}, \mathsf{c}) \to \mathsf{c} <\$\ \mathsf{symbol\ s})\ \mathsf{ops}))$$

where choice combines a list of parsers using $(<|>)$.

A general operator parser

We can abstract even further from this pattern:

$$\textbf{type}\ \mathsf{Op\ a} = (\mathsf{Char}, \mathsf{a} \to \mathsf{a} \to \mathsf{a})$$

$$\mathsf{gen} :: [\mathsf{Op\ a}] \to \mathsf{Parser\ Char\ a} \to \mathsf{Parser\ Char\ a}$$

$$\mathsf{gen\ ops\ p} =$$

$$\quad \mathsf{chainl\ p\ (choice\ (map\ } (\lambda(\mathsf{s}, \mathsf{c}) \to \mathsf{c} <\$\ \mathsf{symbol\ s})\ \mathsf{ops}))$$

where choice combines a list of parsers using $(<|>)$.

Now:

$$\mathsf{e}_1 = \mathsf{gen}\ [(\texttt{'+'}, \mathsf{Plus}), (\texttt{'-'}, \mathsf{Minus})]\ \mathsf{e}_2$$

$$\mathsf{e}_2 = \mathsf{gen}\ [(\texttt{'*'}, \mathsf{Times})]\ \mathsf{e}_3$$

A general operator parser – contd.

$$\begin{aligned} e_1 &= \mathsf{gen}\ [(\texttt{'+'}, \mathsf{Plus}), (\texttt{'-'}, \mathsf{Minus})]\ e_2 \\ e_2 &= \mathsf{gen}\ [(\texttt{'*'}, \mathsf{Times})]\ e_3 \end{aligned}$$

A general operator parser – contd.

$$
\begin{aligned}
e_1 &= \mathsf{gen}\ [(\texttt{'+'}, \mathsf{Plus}), (\texttt{'-'}, \mathsf{Minus})]\ e_2 \\
e_2 &= \mathsf{gen}\ [(\texttt{'*'}, \mathsf{Times})]\ \qquad\qquad\qquad e_3
\end{aligned}
$$

We do not even need the intermediate levels anymore:

$$
\begin{aligned}
e_1 = {}& \mathsf{foldr}\ \mathsf{gen}\ e_3 \\
& \quad [[(\texttt{'+'}, \mathsf{Plus}), (\texttt{'-'}, \mathsf{Minus})], [(\texttt{'*'}, \mathsf{Times})]]
\end{aligned}
$$

A general operator parser – contd.

$$
\begin{aligned}
e_1 &= \mathsf{gen}\ [(\texttt{'+'}, \mathsf{Plus}), (\texttt{'-'}, \mathsf{Minus})]\ e_2 \\
e_2 &= \mathsf{gen}\ [(\texttt{'*'}, \mathsf{Times})]\ \qquad\qquad\qquad e_3
\end{aligned}
$$

We do not even need the intermediate levels anymore:

$$
\begin{aligned}
e_1 = {} & \mathsf{foldr}\ \mathsf{gen}\ e_3 \\
& \qquad [[(\texttt{'+'}, \mathsf{Plus}), (\texttt{'-'}, \mathsf{Minus})], [(\texttt{'*'}, \mathsf{Times})]]
\end{aligned}
$$

Remarks:

- Extra functionality can be added (such as the possibility of right-associative or unary operators).
- User-defined abstractions are like macros or meta-level programming.