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

4. Parser combinators and grammar transformations

This lecture

Parser combinators and grammar transformations

- Parser combinators summary
- Grammar transformations
- Separators and operators

4.1 Parser combinators summary

Parser combinator interface

These are from the last lecture:

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

$$\begin{aligned}
&\text{epsilon} :: \text{Parser s ()} \\
&(<|>) :: \text{Parser s a} \to \text{Parser s a} \to \text{Parser s a} \\
&(<*>) :: \text{Parser s (a} \to \text{b)} \to \text{Parser s a} \to \text{Parser s b} \\
&(<\$>) :: (a \to b) \to \text{Parser s a} \to \text{Parser s b} \\
&\text{satisfy} :: (s \to \text{Bool}) \to \text{Parser s s}
\end{aligned}$$

Parser combinator interface

These are from the last lecture:

```
type Parser s r     (abstract)

epsilon :: Parser s ()
(<|>)  :: 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
```

Parser combinator interface

These are from the last lecture:

```
type Parser s r    (abstract)
epsilon :: Parser s ()
(<|>)  :: 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
```

And the parser for the empty language (also called failp):

```
empty :: Parser s a
empty = Parser (const [])
```

Derived parser combinators

We have seen more functions, but these can be defined in terms of the basic combinators:

```
symbol :: Eq s ⇒ s → Parser s s
symbol x = satisfy (== x)

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

some :: Parser s a → Parser s [a]    -- also called many₁
some p = (:) <$> p <*> many p
```

Derived parser combinators

We have seen more functions, but these can be defined in terms of the basic combinators:

```
symbol :: Eq s ⇒ s → Parser s s
symbol x = satisfy (== x)

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

some :: Parser s a → Parser s [a]    -- also called many1
some p = (:) <$> p <*> many p
```

Similarly:

```
option :: Parser s a → a → Parser s a
option p default = p <|> const default <$> epsilon
```

Example: matching parentheses

Consider this grammar:

$$S \rightarrow (\; S \;) \; S \mid \varepsilon$$

Example: matching parentheses

Consider this grammar:

$$S \rightarrow (\ S\ )\ S \mid \varepsilon$$

Haskell datatype:

```haskell
data Parens = Match Parens Parens | Empty
```

Example: matching parentheses

Consider this grammar:

$$S \rightarrow (S) S \mid \varepsilon$$

Haskell datatype:

```
data Parens = Match Parens Parens | Empty
```

Parser:

```
parens :: Parser Char Parens
parens  =
               <$> symbol '(' <*> parens <*> symbol ')'
               <*>  parens
       <|>                   <$> epsilon
```

Universiteit Utrecht

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

Example: matching parentheses

Consider this grammar:

$$S \rightarrow (S) S \mid \varepsilon$$

Haskell datatype:

```
data Parens = Match Parens Parens | Empty
```

Parser:

```
parens :: Parser Char Parens
parens  =  (λ_ x _ y → Match x y)
             <$> symbol '(' <*> parens <*> symbol ')'
             <*>  parens
         <|> const Empty <$> epsilon
```

More derived combinators

We often need to fill in a result for ε:

```
succeed :: a → Parser s a
succeed x = const x <$> epsilon
```

More derived combinators

We often need to fill in a result for ε:

```
succeed :: a → Parser s a
succeed x = const x <$> epsilon
```

We often do not need all the results of a sequence:

```
(<$) :: a → Parser b → Parser a
x <$ p = const x <$> p
(<*) :: Parser a → Parser b → Parser a
p <* q = const <$> p <*> q
(*>) :: Parser a → Parser b → Parser b
p *> q = flip const <$> p <*> q
```

Matched parentheses again

We can now improve the parser from

```
parens :: Parser Char Parens
parens  =  (λ_ x _ y → Match x y)
              <$> symbol '(' <*> parens <*> symbol ')'
              <*>  parens
           <|> const Empty <$> epsilon
```

to

```
parens :: Parser Char Parens
parens =
   Match <$ symbol '(' <*> parens <* symbol ')' <*> parens
    <|> succeed Empty
```

4.2 Grammar transformations

Removing duplicate productions

Example:

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

can be transformed into

$A \rightarrow u \mid v$

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
```

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

- Removes a source of ambiguity.
- Simplifies the grammar and the code.
- Improves efficiency of the parser.

Inlining nonterminals

Example:

$$A \rightarrow uBv \mid z$$
$$B \rightarrow x \mid w$$

can be transformed into

$$A \rightarrow uxv \mid uwv \mid z$$

$$B \rightarrow x \mid w$$

Inlining nonterminals

Example:

$A \to uBv \mid z$
$B \to x \mid w$

can be transformed into

$A \to uxv \mid uwv \mid z$

$B \to x \mid w$

Parser:

```
a = u <*> b <*> v <|> z
b = x <|> w
```

becomes

```
a =         u <*> x  <*> v
        <|> u <*> w <*> v <|> z
b = x <|> w
```

Inlining nonterminals

Example:

$$A \to uBv \mid z$$
$$B \to x \mid w$$

can be transformed into

$$A \to uxv \mid uwv \mid z$$
$$B \to x \mid w$$

Parser:

```
a = u <*> b <*> v <|> z
b = x <|> w
```

becomes

```
a =      u <*> x <*> v
     <|> u <*> w <*> v <|> z
b = x <|> w
```

- Mainly attractive if the nonterminal is used in only a few places, and after inlining becomes unreachable.
- The reverse transformation – introducing a new nonterminal – can also be useful.
- No effect on efficiency of the parser.

Removing unreachable productions

Example:

$$A \to \mathsf{uxv} \mid \mathsf{uwv} \mid \mathsf{z}$$

$$B \to \mathsf{x} \mid \mathsf{w}$$

can be transformed into

$$A \to \mathsf{uxv} \mid \mathsf{uwv} \mid \mathsf{z}$$

Removing unreachable productions

Example:

$A \rightarrow uxv \mid uwv \mid z$

$B \rightarrow x \mid w$

can be transformed into

$A \rightarrow uxv \mid uwv \mid z$

Parser:

```
a =     u <*> x <*> v
    <|> u <*> w <*> v <|> z
b = x <|> w
```

becomes

```
a =     u <*> x <*> v
    <|> u <*> w <*> v <|> z
```

Removing unreachable productions

Example:

$A \rightarrow uxv \mid uwv \mid z$

$B \rightarrow x \mid w$

can be transformed into

$A \rightarrow uxv \mid uwv \mid z$

Parser:

```
a =       u <*> x  <*> v
      <|> u <*> w <*> v <|> z
b = x <|> w
```

becomes

```
a =       u <*> x  <*> v
      <|> u <*> w <*> v <|> z
```

- Only if B is unreachable and not needed as a (secondary) start symbol.
- Simplifies the grammar.
- Corresponds to dead code removal.

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?

Left factoring – contd.

Consider the grammar:

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

Let us develop a parser for this grammar.

Left factoring – contd.

Consider the grammar:

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

Let us develop a parser for this grammar.

After left factoring, we obtain:

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

Left factoring – contd.

Consider the grammar:

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

Let us develop a parser for this grammar.

After left factoring, we obtain:

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

Left factoring again:

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

Left factoring – contd.

Consider the grammar:

$$S \rightarrow xSy \mid xSx \mid x$$

Let us develop a parser for this grammar.

After left factoring, we obtain:

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

Left factoring again:

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

- Left factoring corresponds to an optimization of the parser.
- Depending on the grammar and the parser combinators used, it can be absolutely essential.

Left recursion

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

Example:

$A \to Az$

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:

$$A \rightarrow Ax_1 \mid Ax_2 \mid \ldots \mid A\ x_n$$
$$A \rightarrow y_1 \mid y_2 \mid \ldots \mid y_m \quad \text{(none of the } y_i \text{ start with A)}$$

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:

$$A \rightarrow Ax_1 \mid Ax_2 \mid \ldots \mid A\ x_n$$
$$A \rightarrow y_1 \mid y_2 \mid \ldots \mid y_m \quad \text{(none of the } y_i \text{ start with A)}$$

This grammar can be transformed to:

$$A \rightarrow y_1 \mid y_1 Z \mid y_2 \mid y_2 Z \mid \ldots \mid y_m \mid y_m Z$$
$$Z \rightarrow x_1 \mid x_1 Z \mid x_2 \mid x_2 Z \mid \ldots \mid x_n \mid x_n Z$$

Example: Removing left recursion

Consider:

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

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

Example: Removing left recursion

Consider:

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

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

Applying the transformation yields:

$$S \to \mathtt{s} \mid \mathtt{s}Z$$
$$Z \to S \mid SZ$$

4.3 Separators and operators

Associative separator/operator

Consider:

Decls → Decls ; Decls
Decls → Decl

Associative separator/operator

Consider:

> Decls → Decls ; Decls
> Decls → Decl

This grammar is left-recursive and ambiguous.

Associative separator/operator

Consider:

> Decls → Decls ; Decls
> Decls → Decl

This grammar is left-recursive and ambiguous.

From a language specification point, one can argue that ambiguity is not problematic if the intended meaning of the different parse trees is the same.

Associative separator/operator

Consider:

> Decls → Decls ; Decls
> Decls → Decl

This grammar is left-recursive and ambiguous.

From a language specification point, one can argue that ambiguity is not problematic if the intended meaning of the different parse trees is the same.

In this case, the meaning will be the same if ';' is intended to be **associative**, i.e., if

> d_1 ; (d_2 ; d_3) and (d_1 ; d_2) ; d_3

have the same meaning.

Associative separator/operator – contd.

If the operator is associative, we can freely choose how to remove the ambiguity by choosing either

> Decls → Decl ; Decls
> Decls → Decl

or

> Decls → Decls ; Decl
> Decls → Decl

Associative separator/operator – contd.

If the operator is associative, we can freely choose how to remove the ambiguity by choosing either

> Decls → Decl ; Decls
> Decls → Decl

or

> Decls → Decls ; Decl
> Decls → Decl

Note that the former grammar can be left-factored, and the latter is still left-recursive (but they are no longer ambiguous).

Parsing separated sequences

Separated sequences occur often, so it makes sense to define an abstraction.

Parsing separated sequences

Separated sequences occur often, so it makes sense to define an abstraction.

Left-factoring

$$\begin{aligned}\text{Decls} &\rightarrow \text{Decl}\ ;\ \text{Decls}\\ \text{Decls} &\rightarrow \text{Decl}\end{aligned}$$

yields

$$\begin{aligned}\text{Decls} &\rightarrow \text{Decl Decls}'\\ \text{Decls}' &\rightarrow\ ;\ \text{Decls} \mid \varepsilon\end{aligned}$$

Parsing separated sequences

Separated sequences occur often, so it makes sense to define an abstraction.

Left-factoring

$$\begin{aligned} &\text{Decls} \rightarrow \text{Decl} \ ; \ \text{Decls} \\ &\text{Decls} \rightarrow \text{Decl} \end{aligned}$$

yields

$$\begin{aligned} \text{Decls} &\rightarrow \text{Decl Decls}' \\ \text{Decls}' &\rightarrow \ ; \ \text{Decls} \mid \varepsilon \end{aligned}$$

We can inline Decls in Decls′:

$$\begin{aligned} \text{Decls} &\rightarrow \text{Decl Decls}' \\ \text{Decls}' &\rightarrow \ ; \ \text{Decl Decls}' \mid \varepsilon \end{aligned}$$

Parsing separated sequences – contd.

$$\begin{aligned} \text{Decls} &\rightarrow \text{Decl Decls}' \\ \text{Decls}' &\rightarrow \ ; \ \text{Decl Decls}' \mid \varepsilon \end{aligned}$$

Here Decls′ is just a sequence:

$$\begin{aligned} \text{Decls} &\rightarrow \text{Decl Decls}' \\ \text{Decls}' &\rightarrow (\ ; \ \text{Decl})^* \end{aligned}$$

Parsing separated sequences – contd.

$$\begin{aligned} \text{Decls} &\rightarrow \text{Decl Decls}' \\ \text{Decls}' &\rightarrow \;;\; \text{Decl Decls}' \mid \varepsilon \end{aligned}$$

Here Decls′ is just a sequence:

$$\begin{aligned} \text{Decls} &\rightarrow \text{Decl Decls}' \\ \text{Decls}' &\rightarrow (\;;\; \text{Decl})^* \end{aligned}$$

We can inline Decls′ now:

$$\begin{aligned} \text{Decls} &\rightarrow \text{Decl } (\;;\; \text{Decl})^* \\ \text{Decls}' &\rightarrow (\;;\; \text{Decl})^* \end{aligned}$$

Parsing separated sequences – contd.

$$\begin{aligned}\text{Decls} &\rightarrow \text{Decl Decls}' \\ \text{Decls}' &\rightarrow \;;\; \text{Decl Decls}' \mid \varepsilon\end{aligned}$$

Here Decls' is just a sequence:

$$\begin{aligned}\text{Decls} &\rightarrow \text{Decl Decls}' \\ \text{Decls}' &\rightarrow (\,;\; \text{Decl})^*\end{aligned}$$

We can inline Decls' now:

$$\begin{aligned}\text{Decls} &\rightarrow \text{Decl}\;(\,;\; \text{Decl})^* \\ \text{Decls}' &\rightarrow (\,;\; \text{Decl})^*\end{aligned}$$

and remove Decls' because it is unreachable:

$$\text{Decls} \rightarrow \text{Decl}\;(\,;\; \text{Decl})^*$$

Parsing separated sequences – contd.

$$\text{Decls} \rightarrow \text{Decl}\ (;\ \text{Decl})^*$$

Abstracting from Decl and ';', we can construct a parser for separated sequences from this grammar:

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

We drop the results from parsing with s and collect the results of the elements in a list.

Operator chains

Consider

$$E \to E\ O\ E \mid Nat$$
$$O \to + \mid -$$

Operator chains

Consider

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

Here, we cannot ignore the meaning of the operators because we have to distinguish them.

Operator chains

Consider

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

Here, we cannot ignore the meaning of the operators because we have to distinguish them.

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

Operator chains

Consider

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

Here, we cannot ignore the meaning of the operators because we have to distinguish them.

Also, '−' 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:

Operator chains

Consider

$E \rightarrow E\ O\ E \mid \mathrm{Nat}$
$O \rightarrow + \mid -$

Here, we cannot ignore the meaning of the operators because we have to distinguish them.

Also, '−' 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:

```
data E = Plus E E | Minus E E | 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 would like to parse

```
1+2-3+4
```

as

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

Questions

What are the types of the following expressions?

Plus
(‘Plus‘ Nat 2)

Operator chains – contd.

We want:

> ((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

What does the following evaluate to?

> foldl (flip ($)) (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?

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

We can obtain this result as follows:

```
chainl :: Parser s a → Parser s (a → a → a) → Parser s a
chainl p s = foldl (flip ($)) <$> p <*> many (flip <$> s <*> p)
```

Operator chains – contd.

We want:

```
((Nat 1 `Plus` Nat 2) `Minus` Nat 3) `Plus` Nat 4
```

What does the following evaluate to?

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

We can obtain this result as follows:

```
chainl :: Parser s a → Parser s (a → a → a) → Parser s a
chainl p s = foldl (flip ($)) <$> p <*> many (flip <$> s <*> p)

e = chainl (Nat <$> natural) o
o = Plus <$ symbol '+' <|> Minus <$ symbol '-'
```

Chain combinators

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

```
chainl :: Parser s a → Parser s (a → a → a) → Parser s a
chainr :: Parser s a → Parser s (a → a → a) → 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:

```
chainl :: Parser s a → Parser s (a → a → a) → Parser s a
chainr :: Parser s a → Parser s (a → a → a) → 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:

```
chainl :: Parser s a → Parser s (a → a → a) → Parser s a
chainr :: Parser s a → Parser s (a → a → a) → 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:

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

Operator priorities

Consider:

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

This is a typical grammar for expressions with operators.

For the same reasons as before, it is ambiguous.

Operator priorities

Consider:

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

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 parse operators of different priorities sequentially.

Operator priorities – contd.

The basic idea is to parse operators of different priorities sequentially.

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 parse operators of different priorities sequentially.

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)

The highest level contains the remaining productions.

Operator priorities – contd.

The basic idea is to parse operators of different priorities sequentially.

For each priority level i, we get

> $\mathsf{E_i} \rightarrow \mathsf{E_i}\ \mathsf{Op_i}\ \mathsf{E_{i+1}} \mid \mathsf{E_{i+1}}$ (for left-associative operators)
>
> or
>
> $\mathsf{E_i} \rightarrow \mathsf{E_{i+1}}\ \mathsf{Op_i}\ \mathsf{E_i} \mid \mathsf{E_{i+1}}$ (for right-associative operators)
>
> or
>
> $\mathsf{E_i} \rightarrow \mathsf{E_{i+1}}\ \mathsf{Op_i}\ \mathsf{E_{i+1}} \mid \mathsf{E_{i+1}}$ (for non-associative operators)

The highest level contains the remaining productions.

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:

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

```
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:

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

```
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:

Grammar	Datatype
$E \rightarrow E + E$	**data** $E = \text{Plus}\ E\ E$
$E \rightarrow E - E$	$\mid \text{Minus}\ E\ E$
$E \rightarrow E * E$	$\mid \text{Times}\ E\ E$
$E \rightarrow (\ E\ )$	
$E \rightarrow \text{Nat}$	$\mid \text{Nat}$

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

Parsers for operator expressions – contd.

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

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

Parsers for operator expressions – contd.

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

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

Parser:

```
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:

```
type Op a = (Char, a → a → a)
gen :: [Op a] → Parser Char a → Parser Char a
gen ops p =
   chainl p (choice (map (λ(s, c) → c <$ symbol s) ops))
```

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

A general operator parser

We can abstract even further from this pattern:

```
type Op a = (Char, a → a → a)
gen :: [Op a] → Parser Char a → Parser Char a
gen ops p =
   chainl p (choice (map (λ(s, c) → c <$ symbol s) ops))
```

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

Now:

```
e1 = gen [('+', Plus), ('-', Minus)] e2
e2 = gen [('*', Times)]              e3
```

A general operator parser – contd.

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

A general operator parser – contd.

$$e_1 = \text{gen}\ [(\text{'+'}, \text{Plus}), (\text{'-'}, \text{Minus})]\ e_2$$
$$e_2 = \text{gen}\ [(\text{'*'}, \text{Times})]\ e_3$$

We do not even need the intermediate levels anymore:

$$e_1 = \text{foldr gen}\ e_3\ [[(\text{'+'}, \text{Plus}), (\text{'-'}, \text{Minus})], [(\text{'*'}, \text{Times})]]$$

A general operator parser – contd.

$$e_1 = \text{gen}\ [(\text{'+'}, \text{Plus}), (\text{'-'}, \text{Minus})]\ e_2$$
$$e_2 = \text{gen}\ [(\text{'*'}, \text{Times})]\ e_3$$

We do not even need the intermediate levels anymore:

$$e_1 = \text{foldr gen}\ e_3\ [[(\text{'+'}, \text{Plus}), (\text{'-'}, \text{Minus})], [(\text{'*'}, \text{Times})]]$$

Remarks:

- Numeric levels not required, just the relative ordering.
- Extra functionality can be added (such as the possibility of right-associative or unary operators).
- User-defined abstractions are very useful.

Next lecture

Developing a larger parser from scratch – a case study.

Including common pitfalls and practical problems.