Talen en Compilers

2010/2011, periode 2

Johan Jeuring

Department of Information and Computing Sciences
Utrecht University

November 24, 2010

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:

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

$$\begin{array}{l}
\mathsf{epsilon} :: \mathsf{Parser\ s}\ ()\\
(<|>)\ :: \mathsf{Parser\ s\ a} \to \mathsf{Parser\ s\ a} \to \mathsf{Parser\ s\ a}\\
(<*>)\ :: \mathsf{Parser\ s}\ (\mathsf{a} \to \mathsf{b}) \to \mathsf{Parser\ s\ a} \to \mathsf{Parser\ s\ b}\\
(<\$>) :: (\mathsf{a} \to \mathsf{b}) \to \mathsf{Parser\ s\ a} \to \mathsf{Parser\ s\ b}\\
\mathsf{satisfy}\ :: (\mathsf{s} \to \mathsf{Bool}) \to \mathsf{Parser\ s\ s}
\end{array}$$

Parser combinator interface

These are from the last lecture:

$\mathbf{type}$ Parser s r $\quad$ **(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:

$$
\begin{aligned}
&\textbf{type}\ \text{Parser s r} \quad \textbf{(abstract)}\\
&\text{epsilon} :: \text{Parser s } ()\\
&(<|>) :: \text{Parser s a} \rightarrow \text{Parser s a} \rightarrow \text{Parser s a}\\
&(<*>) :: \text{Parser s } (\text{a} \rightarrow \text{b}) \rightarrow \text{Parser s a} \rightarrow \text{Parser s b}\\
&(<\$>) :: (\text{a} \rightarrow \text{b}) \rightarrow \text{Parser s a} \rightarrow \text{Parser s b}\\
&\text{satisfy} :: (\text{s} \rightarrow \text{Bool}) \rightarrow \text{Parser s s}
\end{aligned}
$$

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

$$
\begin{aligned}
&\text{empty} :: \text{Parser s a}\\
&\text{empty} = \text{Parser } (\text{const } [\,])
\end{aligned}
$$

Derived parser combinators

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

$$\mathsf{symbol} :: \mathsf{Eq\ s} \Rightarrow \mathsf{s} \rightarrow \mathsf{Parser\ s\ s}$$
$$\mathsf{symbol} = \mathsf{satisfy}\ (== \mathsf{x})$$

$$\mathsf{many} :: \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s}\ [\mathsf{a}]$$
$$\mathsf{many\ p} = (:)\ {<}\${>}\ \mathsf{p}\ {<}{*}{>}\ \mathsf{many\ p}\ {<}|{>}\ \mathsf{const}\ []\ {<}\${>}\ \mathsf{epsilon}$$

$$\mathsf{some} :: \mathsf{Parser\ s\ a} \rightarrow \mathsf{Parser\ s}\ [\mathsf{a}] \quad \text{-- also called } \mathsf{many}_1$$
$$\mathsf{some\ p} = (:)\ {<}\${>}\ \mathsf{p}\ {<}{*}{>}\ \mathsf{many\ p}$$

Derived parser combinators

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

$$\begin{aligned}
&\mathsf{symbol} :: \mathsf{Eq}\ \mathsf{s} \Rightarrow \mathsf{s} \to \mathsf{Parser}\ \mathsf{s}\ \mathsf{s}\\
&\mathsf{symbol} = \mathsf{satisfy}\ (== \mathsf{x})\\
&\mathsf{many} :: \mathsf{Parser}\ \mathsf{s}\ \mathsf{a} \to \mathsf{Parser}\ \mathsf{s}\ [\mathsf{a}]\\
&\mathsf{many}\ \mathsf{p} = (:) <\$> \mathsf{p} <\!*\!> \mathsf{many}\ \mathsf{p} <|> \mathsf{const}\ []\ <\$> \mathsf{epsilon}\\
&\mathsf{some} :: \mathsf{Parser}\ \mathsf{s}\ \mathsf{a} \to \mathsf{Parser}\ \mathsf{s}\ [\mathsf{a}] \quad \text{-- also called } \mathsf{many}_1\\
&\mathsf{some}\ \mathsf{p} = (:) <\$> \mathsf{p} <\!*\!> \mathsf{many}\ \mathsf{p}
\end{aligned}$$

Similarly:

$$\begin{aligned}
&\mathsf{option} :: \mathsf{Parser}\ \mathsf{s}\ \mathsf{a} \to \mathsf{a} \to \mathsf{Parser}\ \mathsf{s}\ \mathsf{a}\\
&\mathsf{option}\ \mathsf{p}\ \mathsf{default} = \mathsf{p} <|> \mathsf{const}\ \mathsf{default} <\$> \mathsf{epsilon}
\end{aligned}$$

Example: matching parentheses

Consider this grammar:

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

Universiteit Utrecht

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

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:

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

Parser:

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

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:

$$\begin{aligned}
&\text{succeed} :: a \to \text{Parser } s\ a\\
&\text{succeed } x = \text{const } x < \$ > \text{epsilon}
\end{aligned}$$

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

$$\begin{aligned}
&(<\$) :: a \to \text{Parser } b \to \text{Parser } a\\
&x <\$\ p = \text{const } x < \$ > p\\
&(<*) :: \text{Parser } a \to \text{Parser } b \to \text{Parser } a\\
&p <*\ q = \text{const} < \$ > p <*> q\\
&(*>) :: \text{Parser } a \to \text{Parser } b \to \text{Parser } b\\
&p *> q = \text{flip const} < \$ > p <*> q
\end{aligned}$$

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:

$$\begin{aligned} A &\rightarrow uBv \mid z \\ B &\rightarrow x \mid w \end{aligned}$$

can be transformed into

$$\begin{aligned} A &\rightarrow uxv \mid uwv \mid z \\ B &\rightarrow x \mid w \end{aligned}$$

Parser:

$$\begin{aligned} a &= u \mathbin{<\!\!*\!\!>} b \mathbin{<\!\!*\!\!>} v \mathbin{<\!|\!>} z \\ b &= x \mathbin{<\!|\!>} w \end{aligned}$$

becomes

$$\begin{aligned} a = {} & \phantom{<\!|\!>}\ u \mathbin{<\!\!*\!\!>} x \mathbin{<\!\!*\!\!>} v \\ & <\!|\!>\ u \mathbin{<\!\!*\!\!>} w \mathbin{<\!\!*\!\!>} v \mathbin{<\!|\!>} z \\ b = {} & x \mathbin{<\!|\!>} w \end{aligned}$$

Inlining nonterminals

Example:

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

can be transformed into

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

Parser:

$$a = u \mathbin{<\!\!*\!\!>} b \mathbin{<\!\!*\!\!>} v \mathbin{<\!|\!>} z$$
$$b = x \mathbin{<\!|\!>} w$$

becomes

$$\begin{aligned} a = {} & \quad u \mathbin{<\!\!*\!\!>} x \mathbin{<\!\!*\!\!>} v \\ & \mathbin{<\!|\!>} u \mathbin{<\!\!*\!\!>} w \mathbin{<\!\!*\!\!>} v \mathbin{<\!|\!>} z \\ b = {} & x \mathbin{<\!|\!>} w \end{aligned}$$

- 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:

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

$$B \rightarrow x \mid w$$

can be transformed into

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

Removing unreachable productions

Example:

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

$$B \to x \mid w$$

can be transformed into

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

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

$$\begin{aligned} A &\rightarrow xQ \mid v \\ Q &\rightarrow y \mid z \end{aligned}$$

Left factoring

Example:

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

can be transformed into

$$\begin{aligned} A &\rightarrow xQ \mid v \\ Q &\rightarrow y \mid z \end{aligned}$$

Parser:

$$a = x \mathbin{<\!\!*\!\!>} y \mathbin{<\!|\!>} x \mathbin{<\!\!*\!\!>} z \mathbin{<\!|\!>} v$$

becomes

$$\begin{aligned} a &= x \mathbin{<\!\!*\!\!>} q \mathbin{<\!|\!>} v \\ q &= y \mathbin{<\!|\!>} z \end{aligned}$$

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:

> $S \rightarrow \mathrm{xSy} \mid \mathrm{xSx} \mid \mathrm{x}$

Let us develop a parser for this grammar.

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 – contd.

Consider the grammar:

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

Let us develop a parser for this grammar.

After left factoring, we obtain:

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

Left factoring again:

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

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:

> $A \rightarrow Az$

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

$$A \to Az$$

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?

- Parsers systematically derived from left-recursive grammars loop.
- Removing left recursion is thus essential if we want 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:

$$A \rightarrow A x_1 \mid A x_2 \mid \ldots \mid A\ x_n$$
$$A \rightarrow y_1 \mid y_2 \mid \ldots \mid y_m \quad \text{(none of the } y_i \text{ start with A)}$$

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:

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

This grammar can be transformed to:

$$
\begin{array}{l}
A \rightarrow y_1 \mid y_1Z \mid y_2 \mid y_2Z \mid \ldots \mid y_m \mid y_mZ \\
Z \rightarrow x_1 \mid x_1Z \mid x_2 \mid x_2Z \mid \ldots \mid x_n \mid x_nZ
\end{array}
$$

Example: Removing left recursion

Consider:

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

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

Example: Removing left recursion

Consider:

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

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

Applying the transformation yields:

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

4.3 Separators and operators

Associative separator/operator

Consider:

> Decls → Decls ; Decls
> Decls → Decl

Associative separator/operator

Consider:

$$\mathrm{Decls} \rightarrow \mathrm{Decls}\ ;\ \mathrm{Decls}$$
$$\mathrm{Decls} \rightarrow \mathrm{Decl}$$

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 $\rightarrow$ Decl ; Decls
> Decls $\rightarrow$ Decl

or

> Decls $\rightarrow$ Decls ; Decl
> Decls $\rightarrow$ 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

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

yields

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

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.

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

Here Decls' is just a sequence:

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

Parsing separated sequences – contd.

$$\begin{aligned} \mathsf{Decls} &\to \mathsf{Decl}\ \mathsf{Decls}' \\ \mathsf{Decls}' &\to ;\ \mathsf{Decl}\ \mathsf{Decls}' \mid \varepsilon \end{aligned}$$

Here Decls′ is just a sequence:

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

We can inline Decls′ now:

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

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.

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:

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

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

Question

What if the separators/operators are supposed to carry meaning?

Operator chains

Consider

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

Operator chains

Consider

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

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

Operator chains

Consider

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

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

Also, '−' associates to the left, so we are dealing with a left-recursive grammar.

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, '−' associates to the left, so we are dealing with a left-recursive grammar.

We inline and remove O and obtain this Haskell datatype:

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, '–' associates to the left, so we are dealing with a left-recursive grammar.

We inline and remove O and obtain this Haskell datatype:

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

Operator chains – contd.

We would like to parse

```
1+2−3+4
```

as

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

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

Universiteit Utrecht

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

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?

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

We can obtain this result as follows:

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

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?

```
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:

$$\mathsf{chainl} :: \mathsf{Parser}\ s\ a \rightarrow \mathsf{Parser}\ s\ (a \rightarrow a \rightarrow a) \rightarrow \mathsf{Parser}\ s\ a$$
$$\mathsf{chainl}\ p\ s = \mathsf{foldl}\ (\mathsf{flip}\ (\$))\ \mathrel{<\$>}\ p \mathrel{<*>} \mathsf{many}\ (\mathsf{flip} \mathrel{<\$>} s \mathrel{<*>} p)$$
$$\mathsf{chainr} :: \mathsf{Parser}\ s\ a \rightarrow \mathsf{Parser}\ s\ (a \rightarrow a \rightarrow a) \rightarrow \mathsf{Parser}\ s\ a$$
$$\mathsf{chainr}\ p\ s = \mathsf{foldr}\ (\$) \mathrel{<\$>} \mathsf{many}\ (\mathsf{flip}\ (\$) \mathrel{<\$>} p \mathrel{<*>} s) \mathrel{<*>} p$$

Chain combinators

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

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

Chain combinators

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

$$\mathsf{chainl} :: \mathsf{Parser}\ s\ a \rightarrow \mathsf{Parser}\ s\ (a \rightarrow a \rightarrow a) \rightarrow \mathsf{Parser}\ s\ a$$
$$\mathsf{chainl}\ p\ s = \mathsf{foldl}\ (\mathsf{flip}\ (\$))\ <\$>\ p\ <\!\!*\!\!>\ \mathsf{many}\ (\mathsf{flip}\ <\$>\ s\ <\!\!*\!\!>\ p)$$
$$\mathsf{chainr} :: \mathsf{Parser}\ s\ a \rightarrow \mathsf{Parser}\ s\ (a \rightarrow a \rightarrow a) \rightarrow \mathsf{Parser}\ s\ a$$
$$\mathsf{chainr}\ p\ s = \mathsf{foldr}\ (\$)\ <\$>\ \mathsf{many}\ (\mathsf{flip}\ (\$)\ <\$>\ p\ <\!\!*\!\!>\ s)\ <\!\!*\!\!>\ p$$

The presence of chainl catches one of the most common cases of left recursion in grammars.

Operator priorities

Consider:

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

Operator priorities

Consider:

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

This is a typical grammar for expressions with operators.

For the same reasons as before, it is ambiguous.

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.

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

Operator priorities – contd.

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

Operator priorities – contd.

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

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)

Operator priorities – contd.

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

For each priority level i, we get

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

or

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

or

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

The highest level contains the remaining productions.

Operator priorities – contd.

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

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

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

we obtain:

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

Parsers for operator expressions

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

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

Parsers for operator expressions

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

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

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

Parsers for operator expressions

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

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

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

       | Nat
```

Parsers for operator expressions

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

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

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

       | 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.

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

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

Parser:

$$
\begin{array}{l}
e_1, e_2, e_3 :: \text{Parser Char E} \\
e_1 = \text{chainl } e_2 \; op_1 \\
e_2 = \text{chainl } e_3 \; op_2 \\
e_3 = \text{parenthesised } e_1 <|> \text{Nat} <\$> \text{natural} \\
op_1, op_2 :: \text{Parser Char } (E \rightarrow E \rightarrow E) \\
op_1 = \text{Plus} \;\; <\$ \text{ symbol '+'} <|> \text{Minus} <\$ \text{ symbol '-'} \\
op_2 = \text{Times} <\$ \text{ symbol '*'}
\end{array}
$$

A general operator parser

We can abstract even further from this pattern:

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

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

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

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

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.

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

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:

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

A general operator parser – contd.

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

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:

- 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.