Chomsky Revisited

– or –

Everything you always wanted to know about *Parsing a Permutation*

Arthur Baars, Andres Löh and Doaitse Swierstra

Institute of Information and Computing Science
Utrecht University
e-mail: {arthurb,andres,doaitse}@cs.uu.nl

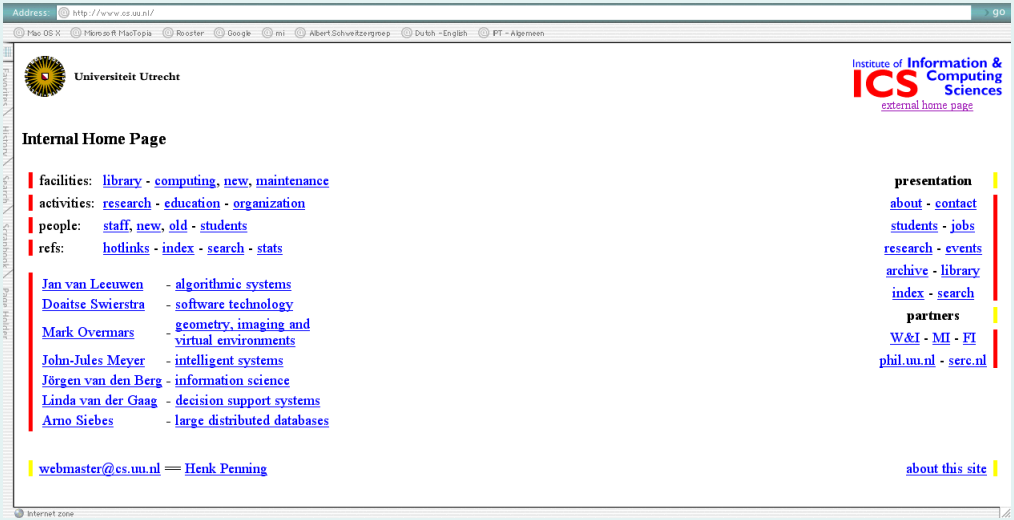
May 6, 2002

The Chomsky Hierarchy

The Chomsky hierarchy:

- Regular
- Context-free
- Context-sensitive
- Recursively enumerable

Example 1: HTML




```
<img width=190                  -- optional, integer
     height=54                  -- optional, integer
     src="/icons/ics.gif"       -- a string
     alt="HOME"                 -- a string
>
```

Attributes: order is irrelevant, each occurs exactly once

Example 2: BibTeX

```
@inProceedings
{ BaarsLoehSwierstra2001,
   author    = {Baars, Arthur and Loeh, Andres
                and Swierstra, S. Doaitse},
   title     = {Parsing Permutation Phrases},
   booktitle = {Preliminary proceedings of
                Haskell workshop 2001,
                # techreport # UU-CS-2001-23},
   year      = 2001,
   pages     = {171--182},
   editor    = {Hinze, Ralf},
}
```

Fields: order is irrelevant, each occurs exactly once

Approach 1

$Attributes \quad ::= (Name \text{ '='} \ Value)*$

Step 1(context-free):

- Parse input as a sequence of assignments

Approach 1

$Attributes \quad ::= (Name \text{ '=' } Value)*$

Step 1(context-free):

- Parse input as a sequence of assignments

Step 2(semantic action):

- Check for valid names
- Check whether each name occurs only once
- Check whether each value has the right type

Approach 2

$$
\begin{array}{ll}
\mathit{Attributes} & ::= \mathit{Field}* \\
\mathit{Field} & ::= \mathit{Width} \mid \mathit{Height} \mid \mathit{Src} \mid \mathit{Alt}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Width} & ::= \texttt{"width="}\ \mathit{Number} \\
\mathit{Height} & ::= \texttt{"height="}\ \mathit{Number} \\
\mathit{Src} & ::= \texttt{"src="}\ \mathit{String} \\
\mathit{Alt} & ::= \texttt{"alt="}\ \mathit{String}
\end{array}
$$

- Name and type checks moved to syntactic level
- Occurrence check still in second step

Approach 3

$$
\begin{array}{lll}
\mathit{Attributes} & ::= & \mathit{Width}\ \mathit{Height}\ \mathit{Src}\ \mathit{Alt} \\
 & | & \mathit{Width}\ \mathit{Height}\ \mathit{Alt}\ \mathit{Src} \\
 & | & \mathit{Width}\ \mathit{Src}\ \mathit{Height}\ \mathit{Alt} \\
 & | & \mathit{Width}\ \mathit{Src}\ \mathit{Alt}\ \mathit{Height} \\
 & | & \mathit{Width}\ \mathit{Alt}\ \mathit{Height}\ \mathit{Src} \\
 & | & \mathit{Width}\ \mathit{Alt}\ \mathit{Src}\ \mathit{Height} \\
 & | & \mathit{Height}\ \mathit{Width}\ \mathit{Src}\ \mathit{Alt} \\
 & | & \mathit{Height}\ \mathit{Width}\ \mathit{Alt}\ \mathit{Src} \\
 & | & \mathit{Height}\ \mathit{Src}\ \mathit{Width}\ \mathit{Alt} \\
 & | & \mathit{Height}\ \mathit{Src}\ \mathit{Alt}\ \mathit{Width} \\
 & | & \mathit{Height}\ \mathit{Alt}\ \mathit{Width}\ \mathit{Src} \\
 & | & \mathit{Height}\ \mathit{Alt}\ \mathit{Src}\ \mathit{Width} \\
 & | & \mathit{Src}\ \mathit{Width}\ \mathit{Height}\ \mathit{Alt} \\
 & | & \mathit{Src}\ \mathit{Width}\ \mathit{Alt}\ \mathit{Height} \\
 & | & \mathit{Src}\ \mathit{Height}\ \mathit{Width}\ \mathit{Alt} \\
 & | & \ldots
\end{array}
$$

EBNF grammars grow too large

BNF:

- alternative "|"
- composition "␣"
- terminals "'s'"
- empty string ϵ

EBNF grammars grow too large

BNF:

- alternative "|"
- composition "␣"
- terminals "'s'"
- empty string ϵ

EBNF:

- repetition "*" and "+"
- option "?"
- grouping "(" ... ")"

EBNF grammars grow too large

BNF:

- alternative "|"
- composition "␣"
- terminals "'s'"
- empty string ϵ

EBNF:

- repetition "*" and "+"
- option "?"
- grouping "(" ... ")"

EEBNF:

- permutations "||"

EBNF grammars grow too large

BNF:

- alternative "|"
- composition "␣"
- terminals "'s'"
- empty string ϵ

EBNF:

- repetition "*" and "+"
- option "?"
- grouping "(" ... ")"

EEBNF:

- permutations "||"

...

EEEEEEBNF:

What do we need?

- Formalism with basic operators
- Extension mechanism in the formalism

What do we need?

- Formalism with basic operators
- Extension mechanism in the formalism $\Longleftarrow$ **Programming language**

What do we need?

- Formalism with basic operators $\Longleftarrow$ **Library, Package**
- Extension mechanism in the formalism $\Longleftarrow$ **Programming language**

Combinator language

- Basic operators "combine" entities from a formalism
- Extension mechanism is borrowed from a programming language

Combinator language

- Basic operators "combine" entities from a $\underbrace{\text{formalism}}_{EBNF}$
- Extension mechanism is borrowed from a programming language

Combinator language

- Basic $\underbrace{\text{operators}}_{|,*,?,\epsilon}$ "combine" entities from a $\underbrace{\text{formalism}}_{EBNF}$
- Extension mechanism is borrowed from a programming language

Combinator language

- Basic $\underbrace{\text{operators}}_{|,*,?,\epsilon}$ "combine" $\underbrace{\text{entities}}_{parsers}$ from a $\underbrace{\text{formalism}}_{EBNF}$
- Extension mechanism is borrowed from a programming language

Combinator language

- Basic $\underbrace{\text{operators}}_{|,*,?,\epsilon}$ "combine" $\underbrace{\text{entities}}_{parsers}$ from a $\underbrace{\text{formalism}}_{EBNF}$
- Extension mechanism is borrowed from a $\underbrace{\text{programming language}}_{Haskell}$

A Parser Combinator Library

Basic operators:

	EBNF	Haskell
alternative	$a \mid b$	
composition	$a\ b$	
terminals	's'	
empty string	ϵ	

A Parser Combinator Library

Basic operators:

	EBNF	Haskell
alternative	$a \mid b$	$a \mid b$
composition	$a\ b$	
terminals	's'	
empty string	ϵ	

A Parser Combinator Library

Basic operators:

	EBNF	Haskell
alternative	$a \mid b$	a <|> b
composition	$a\ b$	
terminals	's'	
empty string	ϵ	

A Parser Combinator Library

Basic operators:

	EBNF	Haskell
alternative	$a \mid b$	a <|> b
composition	$a\ b$	a <*> b
terminals	's'	
empty string	ϵ	

A Parser Combinator Library

Basic operators:

	EBNF	Haskell
alternative	$a \mid b$	a <|> b
composition	$a\ b$	a <*> b
terminals	's'	$symbol$'s'
empty string	ϵ	

A Parser Combinator Library

Basic operators:

	EBNF	Haskell
alternative	$a \mid b$	a <|> b
composition	$a\ b$	a <*> b
terminals	's'	$symbol$'s'
empty string	ϵ	$epsilon$

A Parser Combinator Library

Basic operators:

	EBNF	Haskell
alternative	$a \mid b$	a `<|>` b
composition	$a\ b$	a `<*>` b
terminals	`'s'`	*symbol* `'s'`
empty string	ϵ	*epsilon*

Other operators:

repetition	$a*$	*many* a
option	$a?$	*option* a
grouping	$(\ldots)$	$(\ldots)$

A Parser Combinator Library

Basic operators:

	EBNF	Haskell
alternative	$a \mid b$	a `<|>` b
composition	$a\ b$	a `<*>` b
terminals	`'s'`	*symbol* `'s'`
empty string	ϵ	*epsilon*

Other operators:

repetition	$a*$	*many* a
option	$a?$	*option* a
grouping	$(\ldots)$	$(\ldots)$

Example of use:

```
symbol 'H'<*>
symbol 'o'<*>
symbol 'i'
```

A Parser Combinator Library

Basic operators:

	EBNF	Haskell
alternative	$a \mid b$	a `<|>` b
composition	$a\ b$	a `<*>` b
terminals	`'s'`	*symbol* `'s'`
empty string	ϵ	*epsilon*

Other operators:

repetition	$a*$	*many* a
option	$a?$	*option* a
grouping	$(\ldots)$	$(\ldots)$

Example of use:

```
symbol 'H'<*>
option (symbol 'o')<*>
symbol 'i'
```

A Parser Combinator Library

Basic operators:

	EBNF	Haskell
alternative	$a \mid b$	a `<|>` b
composition	$a\ b$	a `<*>` b
terminals	`'s'`	*symbol* `'s'`
empty string	ϵ	*epsilon*

Other operators:

repetition	$a*$	*many* a
option	$a?$	*option* a
grouping	$(\ldots)$	$(\ldots)$

Example of use:

```
many (symbol 'H'<*>
      option (symbol 'o')<*>
      symbol 'i'
     )
```

What is a Parser

$String \longrightarrow a$

What is a Parser

$String \longrightarrow [a]$

What is a Parser

$$String \rightarrow [(a, String)]$$

What is a Parser

$$String \rightarrow [(a, String)]$$

Abbreviation:

$$Parser\ a$$

Types of the Combinators

<|>	$:: Parser\ a \longrightarrow Parser\ a \longrightarrow Parser\ a$

Types of the Combinators

<|>	$:: Parser\ a \rightarrow Parser\ a \rightarrow Parser\ a$

<*>	$:: Parser\ a \rightarrow Parser\ b \rightarrow Parser\ (a, b)$

Types of the Combinators

<|>	$:: Parser\ a \to Parser\ a \to Parser\ a$
<*>	$:: Parser\ a \to Parser\ b \to Parser\ (a, b)$
<*>	$:: Parser\ (b \to c) \to Parser\ b \to Parser\ c$

Types of the Combinators

<|>	$:: Parser\ a \to Parser\ a \to Parser\ a$
<*>	$:: Parser\ a \to Parser\ b \to Parser\ (a, b)$
<*>	$:: Parser\ (b \to c) \to Parser\ b \to Parser\ c$
symbol	$:: Char \to Parser\ Char$

Types of the Combinators

`<|>`	$:: Parser\ a \rightarrow Parser\ a \rightarrow Parser\ a$
`<*>`	$:: Parser\ a \rightarrow Parser\ b \rightarrow Parser\ (a, b)$
`<*>`	$:: Parser\ (b \rightarrow c) \rightarrow Parser\ b \rightarrow Parser\ c$
symbol	$:: Char \rightarrow Parser\ Char$
epsilon	$:: a \rightarrow Parser\ a$

Types of the Combinators

`<|>`	$:: Parser\ a \rightarrow Parser\ a \rightarrow Parser\ a$
`<*>`	$:: Parser\ a \rightarrow Parser\ b \rightarrow Parser\ (a, b)$
`<*>`	$:: Parser\ (b \rightarrow c) \rightarrow Parser\ b \rightarrow Parser\ c$
symbol	$:: Char \rightarrow Parser\ Char$
epsilon	$:: a \rightarrow Parser\ a$
`<$>`	$:: (a \rightarrow b) \rightarrow Parser\ a \rightarrow Parser\ b$

Make your own extensions

EBNF:

$$
\begin{aligned}
&\mathit{many} && :: \quad \mathit{Parser}\ a \rightarrow \mathit{Parser}\ [a] \\
&\mathit{many}\ x && = \quad x\ \texttt{<*>}\ \mathit{many}\ x \\
& && \texttt{<|>}\ \mathit{epsilon} \\
&\mathit{option}\ x && = \ldots
\end{aligned}
$$

Make your own extensions

EBNF:

$$
\begin{aligned}
&\mathit{many} && :: \ \mathit{Parser}\ a \to \mathit{Parser}\ [a] \\
&\mathit{many}\ x && = \ (:)\ \texttt{<\$>}\ x\ \texttt{<*>}\ \mathit{many}\ x \\
& && \texttt{<|>}\ \mathit{epsilon} \\
&\mathit{option}\ x && = \ldots
\end{aligned}
$$

Make your own extensions

EBNF:

$$
\begin{aligned}
&\mathit{many} && :: \quad \mathit{Parser}\ a \to \mathit{Parser}\ [a] \\
&\mathit{many}\ x && = \quad (:)\ \texttt{<\$>}\ x\ \texttt{<*>}\ \mathit{many}\ x \\
& && \texttt{<|>}\ \mathit{epsilon}\ [\,] \\
&\mathit{option}\ x && = \ldots
\end{aligned}
$$

Make your own extensions

EBNF:

$$
\begin{aligned}
&many && :: \; Parser\ a \rightarrow Parser\ [a] \\
&many\ x && = \; (:)\ \texttt{<\$>}\ x\ \texttt{<*>}\ many\ x \\
& && \texttt{<|>}\ epsilon\ [\,] \\
&option\ x && = \ldots
\end{aligned}
$$

EEBNF:

$$
\begin{aligned}
&token && :: [Char] \rightarrow Parser\ [Char] \\
&token\ (s : rest) && = symbol\ s\ \texttt{<*>}\ token\ rest \\
&token\ [\,] && = epsilon
\end{aligned}
$$

Make your own extensions

EBNF:

```
many       :: Parser a → Parser [a]
many x     = (:) <$> x <*> many x
           <|> epsilon []
option x   = ...
```

EEBNF:

```
token      :: [Char] → Parser [Char]
token (s : rest) = (:) <$> symbol s <*> token rest
token []   = epsilon ""
```

Make your own extensions

EBNF:

```
many       :: Parser a → Parser [a]
many x     = (:) <$> x <*> many x
           <|> epsilon []
option x   = ...
```

EEBNF:

```
token      :: [Char] → Parser [Char]
token (s : rest) = (:) <$> symbol s <*> token rest
token []   = epsilon ""
```

EEEBNF:

```
permute    = ...
<||>       = ...
```

Example: IMG-tag attributes

Parser for attributes:

```
permute (field "width" integer
        <||>field "height" integer
        <||>field "src" string
        <||>field "alt" string
        )
```

Helper function for parsing assignments:

```
field name val = glue<$>token name <*> symbol '=' <*> val
  where glue _ _ v = v
```

Implementation idea

$$s = permute\ (a \texttt{<|>}\ b \texttt{<|>}\ c)$$

should be equivalent to:

$$\begin{aligned} s = {} & a \texttt{<*>}\ b \texttt{<*>}\ c \\ & \texttt{<|>} a \texttt{<*>}\ c \texttt{<*>}\ b \\ & \texttt{<|>} b \texttt{<*>}\ a \texttt{<*>}\ c \\ & \texttt{<|>} b \texttt{<*>}\ c \texttt{<*>}\ a \\ & \texttt{<|>} c \texttt{<*>}\ a \texttt{<*>}\ b \\ & \texttt{<|>} c \texttt{<*>}\ b \texttt{<*>}\ a \end{aligned}$$

Implementation idea

$$s = permute\ (a \mathbin{<|>} b \mathbin{<|>} c)$$

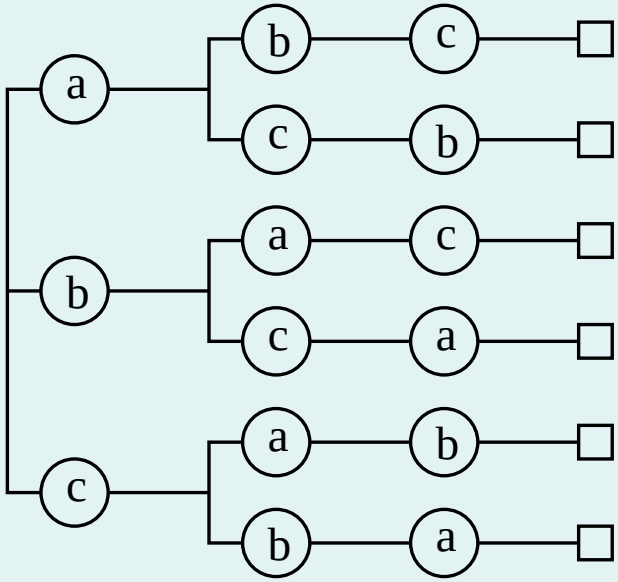
should be equivalent to:

$$\begin{aligned} s = {} & a \mathbin{<*>} b \mathbin{<*>} c \\ \mathbin{<|>} & a \mathbin{<*>} c \mathbin{<*>} b \\ \mathbin{<|>} & b \mathbin{<*>} a \mathbin{<*>} c \\ \mathbin{<|>} & b \mathbin{<*>} c \mathbin{<*>} a \\ \mathbin{<|>} & c \mathbin{<*>} a \mathbin{<*>} b \\ \mathbin{<|>} & c \mathbin{<*>} b \mathbin{<*>} a \end{aligned}$$

or, with left-factorization:

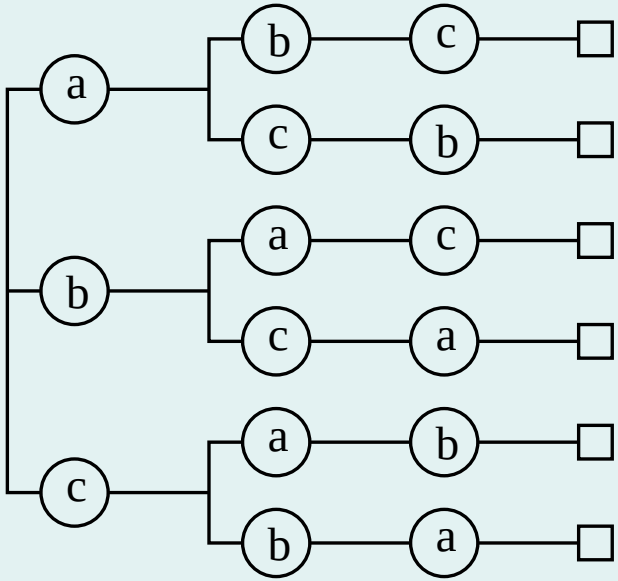
$$\begin{aligned} s = {} & a \mathbin{<*>} (b \mathbin{<*>} c \mathbin{<|>} c \mathbin{<*>} b) \\ \mathbin{<|>} & b \mathbin{<*>} (a \mathbin{<*>} c \mathbin{<|>} c \mathbin{<*>} a) \\ \mathbin{<|>} & c \mathbin{<*>} (a \mathbin{<*>} b \mathbin{<|>} b \mathbin{<*>} a) \end{aligned}$$

Implementation idea



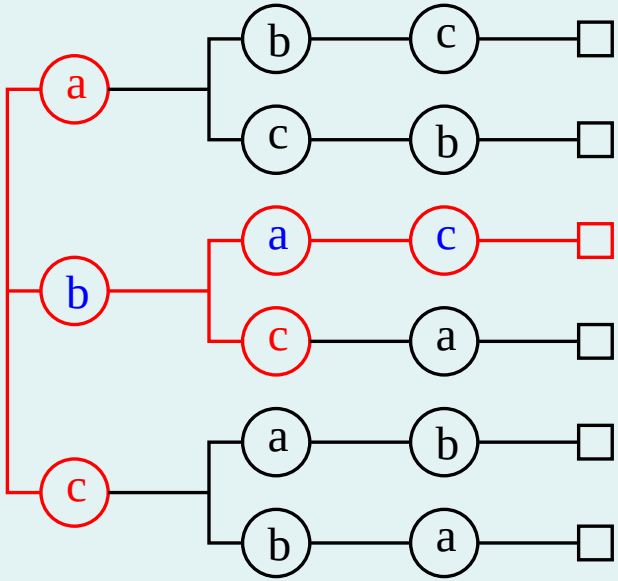
- <| |>: constructs an intermediate data structure(decision tree)
- *permute*: transforms it into a parser

Complexity



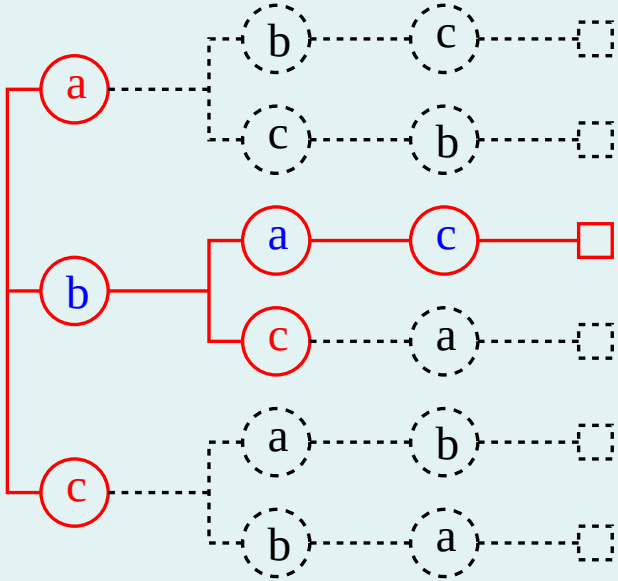
Size of the intermediate structure: O(n!)

Complexity



Example: parse string "bac"

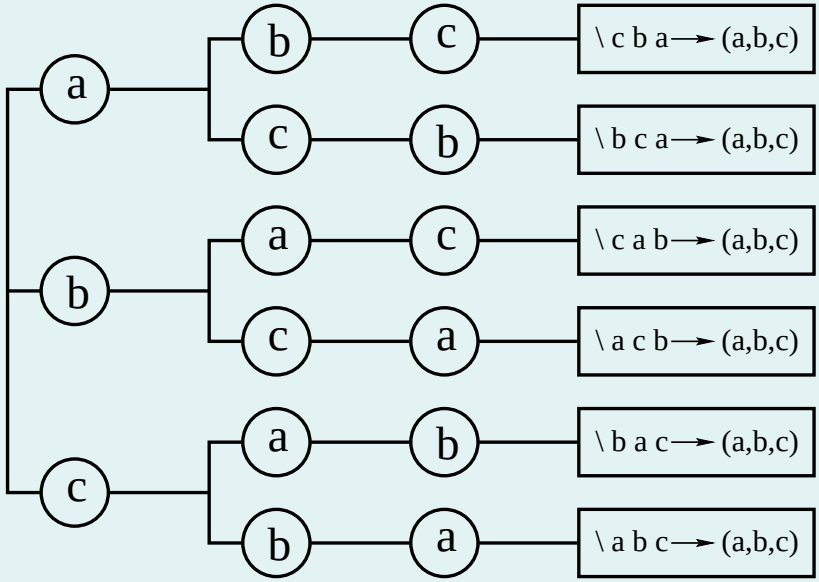
Complexity



Thanks to lazy evaluation only one path is constructed

Size of the intermediate structure: $O(n^2)$

Finishing Touch: Reordering



Each leaf contains a special reordering function

Recap

- <| |>: combinator to construct large permutation grammar
  - Provides notation for writing such grammars easily
  - Makes programmer happy
- *permute*: uses grammar to parse input
- Lazy evaluation:
  - Only small part of the grammar is computed
  - Makes computer happy

Intermediate data structure (1)

Structure of the tree:

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Tree} & = & \mathit{Choice}\ [\mathit{Branch}] \\
 & | & \mathit{Leaf} \\
\textbf{data}\ \mathit{Branch} & = & \mathit{Branch}\ (\mathit{Parser?})\ \mathit{Tree}
\end{array}
$$

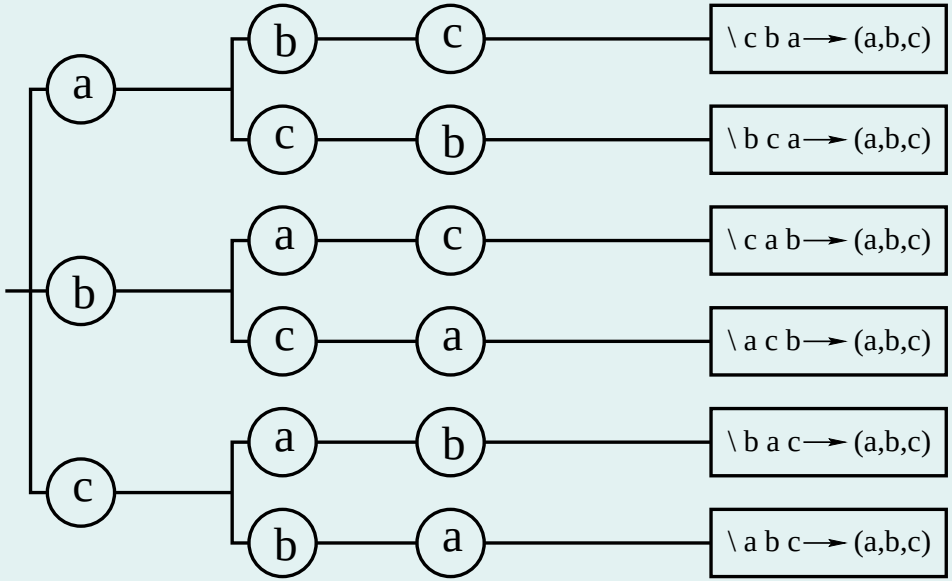
Intermediate data structure (2)

Structure of the tree:

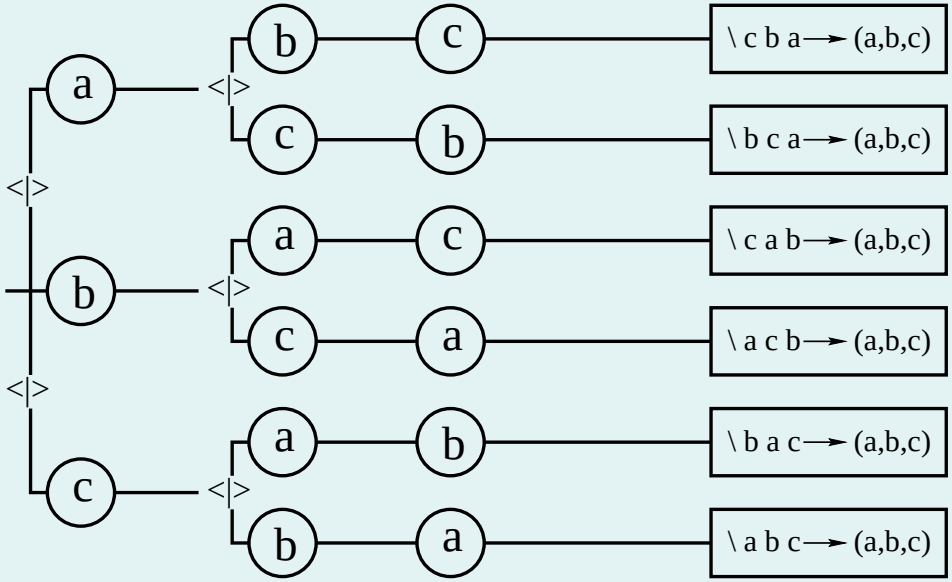
$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree}\ a & = & \mathit{Choice}\ [\mathit{Branch}\ a] \\
 & | & \mathit{Leaf}\ a \\
\textbf{data}\ \mathit{Branch}\ a & = & \exists x.\mathit{Branch}\ (\mathit{Parser}\ x)\ (\mathit{Tree}\ (x \to a))
\end{array}
$$

an element of type a is constructed by applying the result of the rest of the path (of type $x \to a$) to the result of the parser in the branch (of type x)!

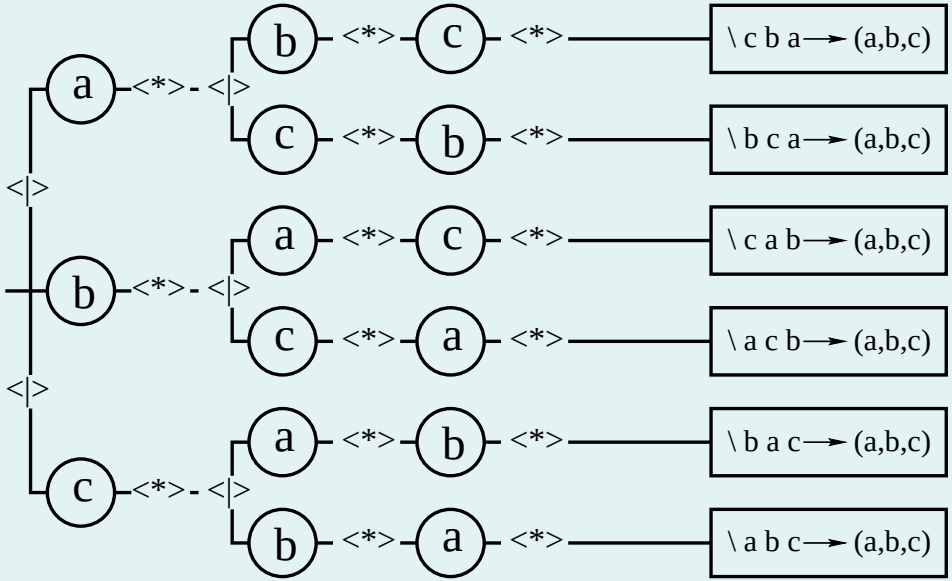
Transforming intermediate structure



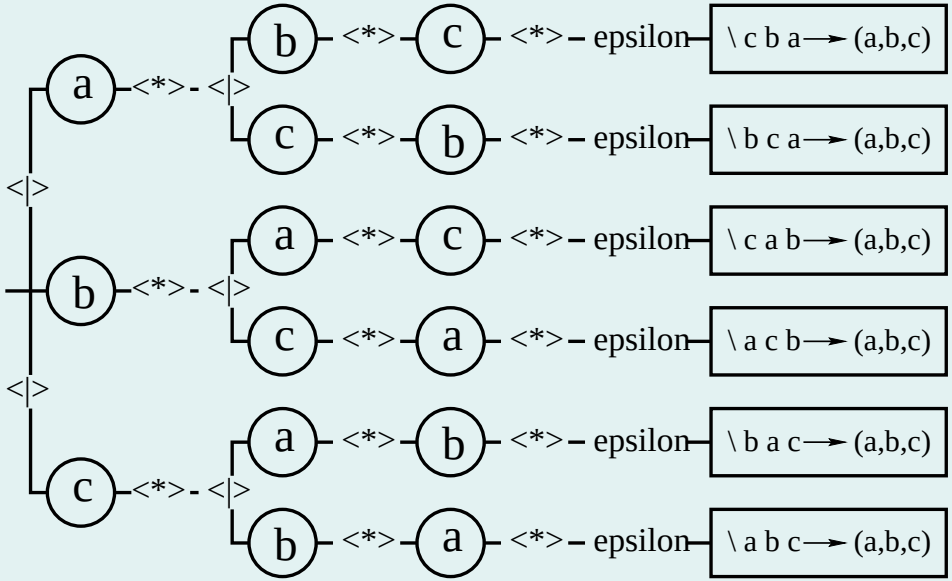
Transforming intermediate structure



Transforming intermediate structure



Transforming intermediate structure



Transforming intermediate structure

permute traverses the grammar and constructs a parser:

```
permute (Leaf f)      = epsilon f
permute (Choice bs)   = choice [glue <$> p <*> permute rest
                                | Branch p rest ← bs
                               ]
  where glue x f = f x
```

The function *choice* takes a list of parsers and puts <|> between them

Summary

- Permutation parsing can be done context-free
- Necessary large grammars can be generated
  (by <|| > and $permute$ combinators)
- Lazy evaluation reduces complexity from O(n!) to O(n^2)

Conclusions

Combinators are cool

- Formalisms are programs
- Extendible
- Executable

Haskell is hot

- Higher order functions
- Lazy evaluation
- User defined operators
- ...