Parsing Permutation Phrases

Arthur Baars, Andres Löh, Doaitse Swierstra

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

September 2, 2001

Permutation phrases

All the following describe the same tag:

```
<img src="/icons/li-dsol.gif" alt="Utrecht University" height=54 width=190>
<img width=190 height=54 alt="Utrecht University" src="/icons/li-dsol.gif">
<img alt="Utrecht University" width=190 src="/icons/li-dsol.gif" height=54>
```

- Attributes of XML tags
- Specifiers in C-declarations
  - `static int signed foo;`
  - `int static signed foo;`
- Derived read-function for Haskell's record syntax
- Citation fields in BibTeX entries
- ...

Previous work

Extending context-free grammars with permutation phrases
Robert D. Cameron:

- Extension of EBNF notation for permutation phrases
  - $\ll A \parallel B \parallel C \gg$
- Pseudo-code algorithm
  - LL(1) parsing
  - Parser generators
  - Ignores reordering and type of different constituents

Interface of our parser combinators

We will present our permutation combinators in the context of the latest version of the UU_Parsing library:

infixl 5 $\langle\!|\!\rangle$
infixl 6 $\langle\!*\!\rangle$
infix 7 $\langle\!\$\!\rangle$

class *IsParser p* **where**

$$
\begin{array}{lcl}
\mathit{pFail} & :: & p\ a \\
\mathit{pSucceed} & :: & a \to p\ a \\
\mathit{pSym} & :: & \mathit{Char} \to p\ \mathit{Char} \\
(\langle\!*\!\rangle) & :: & p\ (a \to b) \to p\ a \to p\ b \\
(\langle\!|\!\rangle) & :: & p\ a \to p\ a \to p\ a \\
(\langle\!\$\!\rangle) & :: & (a \to b) \to p\ a \to p\ b \\
f\ \langle\!\$\!\rangle\ p & = & \mathit{pSucceed}\ f\ \langle\!*\!\rangle\ p \\
\mathit{parse} & :: & p\ a \to \mathit{String} \to \mathit{Result}\ a
\end{array}
$$

How to use parser combinators

```
src = "/icons/li-dsol.gif"
width = 190
```

$$
\begin{array}{ll}
pField & :: \; IsParser\ p \Rightarrow String \rightarrow p\ a \rightarrow p\ a \\
pField\ fld\ p & = (\lambda \_\ \_\ val \rightarrow val) \mathbin{<\$>} pTok\ fld \mathbin{<*>} pSym\ \texttt{'='} \mathbin{<*>} p
\end{array}
$$

Adding parentheses according to associativity and priority of $<\$>$ and $<*>$:

$$
pField\ fld\ p = ((((\lambda \_\ \_\ val \rightarrow val) \mathbin{<\$>} pTok\ fld) \mathbin{<*>} pSym\ \texttt{'='}) \mathbin{<*>} p)
$$

$$
\begin{array}{ll}
pMaybe & :: \; IsParser\ p \Rightarrow p\ a \rightarrow p\ (Maybe\ a) \\
pMaybe\ p & = Just \mathbin{<\$>} p \\
 & \mathbin{<|>} pSucceed\ Nothing
\end{array}
$$

```
UU_Parsing_Demo> parse (pMaybe (pField "width" pLength)) "width␣=␣190"

Result: Just 190

UU_Parsing_Demo> parse (pMaybe (pField "width" pLength)) ""

Result: Nothing
```

How to use the new combinators

```
<img height=54 width=190 src="/icons/li-dsol.gif" alt="Utrecht University">
```

$$
\begin{array}{l}
\textbf{data}\ \mathit{XHTML} \;=\; \mathit{Img}\ \{\,\mathit{src} \;\;\;\;\;\; :: \mathit{URI} \\
\qquad\qquad\qquad\quad ,\,\mathit{alt} \;\;\;\;\;\; :: \mathit{Text} \\
\qquad\qquad\qquad\quad ,\,\mathit{width} \;\;\; :: \mathit{Maybe}\ \mathit{Length} \\
\qquad\qquad\qquad\quad ,\,\mathit{height} \;\; :: \mathit{Maybe}\ \mathit{Length} \\
\qquad\qquad\qquad\quad \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{pImgTag} & :: & \mathit{Parser}\ \mathit{XHTML} \\
\mathit{pImgTag} & = & (\lambda\_\ \_\ x\ \_ \rightarrow x)\ \langle\$\rangle\ \mathit{pTok}\ \texttt{"<"}\ \langle\circledast\rangle\ \mathit{pTok}\ \texttt{"img"}\ \langle\circledast\rangle\ \mathit{attrs}\ \langle\circledast\rangle\ \mathit{pTok}\ \texttt{"/>"} \\
\quad \textbf{where}\ \mathit{attrs} & = & \mathit{pPerms}\ (\mathit{Img}\ \langle\langle\$\rangle\rangle\ \mathit{pField}\ \texttt{"src"}\ \mathit{pURI} \\
 & & \qquad\qquad\qquad\quad \langle\langle\circledast\rangle\rangle\ \mathit{pField}\ \texttt{"alt"}\ \mathit{pText} \\
 & & \qquad\qquad\qquad\quad \langle\langle\circledast\rangle\rangle\ \mathit{pMaybe}\ (\mathit{pField}\ \texttt{"width"}\ \mathit{pLength}) \\
 & & \qquad\qquad\qquad\quad \langle\langle\circledast\rangle\rangle\ \mathit{pMaybe}\ (\mathit{pField}\ \texttt{"height"}\ \mathit{pLength}) \\
 & & \qquad\qquad\qquad\quad)
\end{array}
$$

Interface of new parser combinators

infixl 6 $\langle\!\langle * \rangle\!\rangle$, $\langle\!\langle * \rangle\!\rangle$
infix 7 $\langle\!\langle \$ \rangle\!\rangle$, $\langle\!\langle \$ \rangle\!\rangle$

$$\begin{array}{lll}
\langle\!\langle * \rangle\!\rangle & :: & IsParser\ p \Rightarrow \quad p\ (a \to b) \to p\ a \to \quad p\ b \\
\langle\!\langle * \rangle\!\rangle & :: & IsParser\ p \Rightarrow Perms\ p\ (a \to b) \to p\ a \to Perms\ p\ b
\end{array}$$

$$\begin{array}{lll}
\langle\!\langle \$ \rangle\!\rangle & :: & IsParser\ p \Rightarrow (a \to b) \to p\ a \to \quad p\ b \\
\langle\!\langle \$ \rangle\!\rangle & :: & IsParser\ p \Rightarrow (a \to b) \to p\ a \to Perms\ p\ b
\end{array}$$

$$pPerms \quad :: \quad IsParser\ p \Rightarrow Perms\ p\ a \to p\ a$$

Toy Example

$$S \quad ::= \quad \ll \mathbf{a}* \parallel \mathbf{b} \parallel \mathbf{c}? \gg$$

```
pS    :: Parser (String, Char, Char)
pS    = pPerms ((,,) <$$> pList (pSym 'a')
                     <**> pSym 'b'
                     <**> pOptSym 'c'
               )

pOptSym     :: Char -> Parser Char
pOptSym x   = pSym x <|> pSucceed '_'

UU_Parsing_Demo> parse pS "caab"

Result:
("aa",'b','c')
```

Toy Example (continued)

$$S \quad ::= \quad \ll \mathbf{a}* \parallel \mathbf{b} \parallel \mathbf{c}? \gg$$

```
UU_Parsing_Demo> parse pS ""

Symbol 'b' inserted at end of file; 'b' or 'c' or ('a')* expected.
Result:
("",'b','_')

UU_Parsing_Demo> parse pS "abd"

Errors:
Symbol 'd' at end of file deleted; 'c' or eof expected.
Result:
("a",'b','_')
```

Requirements

- Easy to construct parsers for permutation phrases
- Efficient
- Inheritance of error-reporting/repairing facilities
- Constituents of the permutation phrase may have different types!!!
- Constituents of the permutation phrase may be optional (empty)

Implementation Idea

The permutation phrase:

$$S \quad ::= \quad \ll \mathbf{a} \parallel \mathbf{b} \parallel \mathbf{c} \gg$$

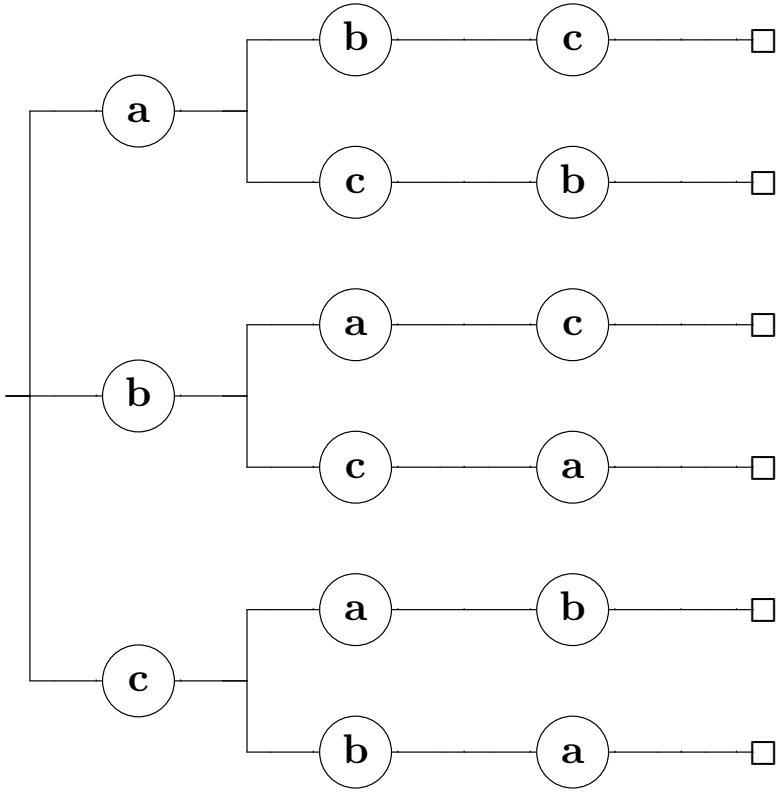
is formally equivalent to:

$$\begin{array}{lll} S & ::= & \mathbf{abc} \\ & | & \mathbf{acb} \\ & | & \mathbf{bac} \\ & | & \mathbf{bca} \\ & | & \mathbf{cab} \\ & | & \mathbf{cba} \end{array}$$

Implementation Idea (continued)

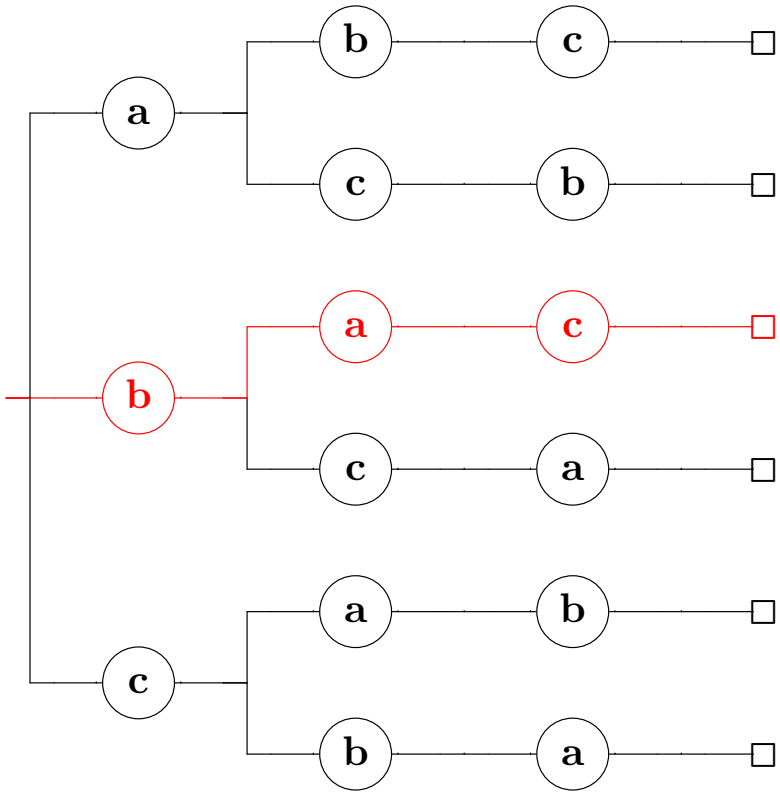
$$
\begin{array}{lcl}
S & ::= & \mathbf{a(bc|cb)} \\
 & | & \mathbf{b(ac|ca)} \\
 & | & \mathbf{c(ab|ba)}
\end{array}
$$

Consider parsing the string "**bac**"



Implementation Idea (continued)

- When recognizing a permutation a parser follows a path through the permutation tree from the root to one of the exits.
- Permutation tree has a potential size of order $n!$, where n is the number of permutable constituents.
- Lazy evaluation ensures that only those parts of the tree are evaluated that correspond to the path followed by the parser.



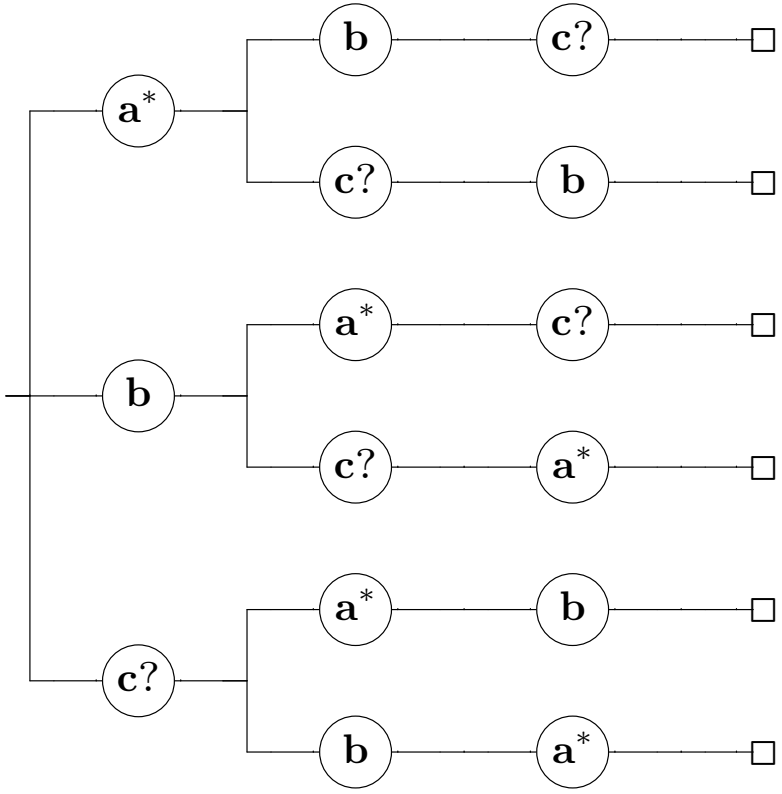
Complications caused by possibly empty elements

$$S \quad ::= \quad \ll \mathbf{a}^* \parallel \mathbf{b} \parallel \mathbf{c}? \gg$$

$$\begin{array}{lll} S & ::= & \mathbf{a}^*(\mathbf{bc}?|\mathbf{c}?\mathbf{b}) \\ & | & \mathbf{b}(\mathbf{a}^*\mathbf{c}?|\mathbf{c}?\mathbf{a}^*) \\ & | & \mathbf{c}?(\mathbf{a}^*\mathbf{b}|\mathbf{ba}^*) \end{array}$$

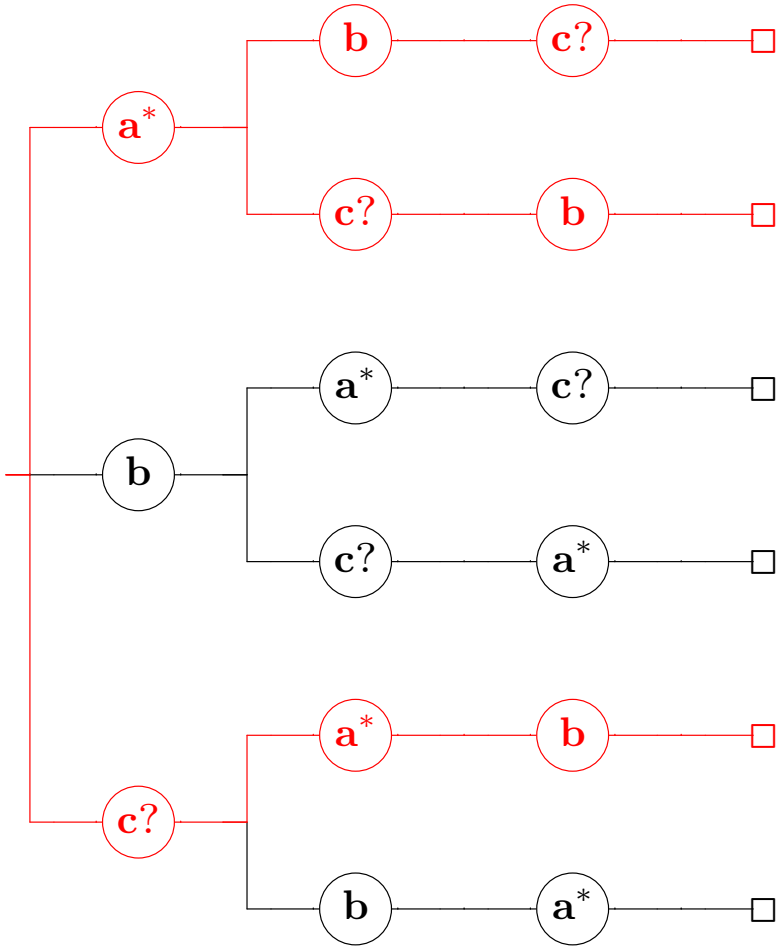
Elements with a ? or a * can be empty.

Problem: "**aab**" is ambiguous.



Complications caused by possibly empty elements (continued)

There are three different paths for "**aab**", because the empty "**c**?" can be derived at different positions:



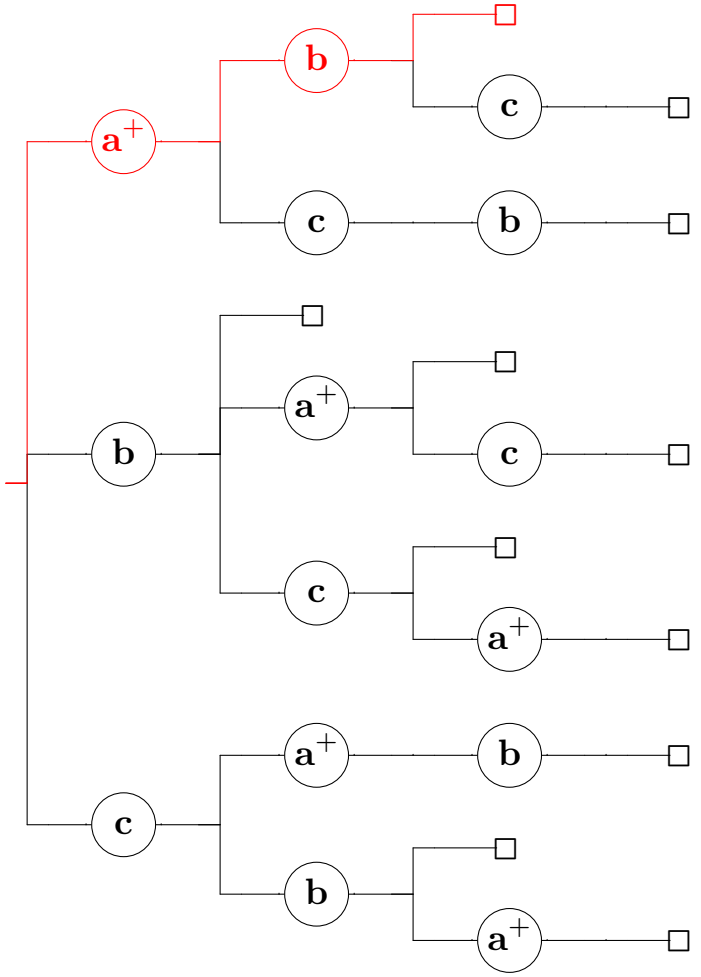
Disambiguating: splitting constituents in empty and non-emtpy part

Solution: optional elements that are not present can only be derived at the end of a parse.

This means: no nodes can derive the empty string, and additional exits (leaves) must be added.

$splitParser$ takes care of splitting an optional element (parser that can derive the empty string) in its empty part and its non-empty part.

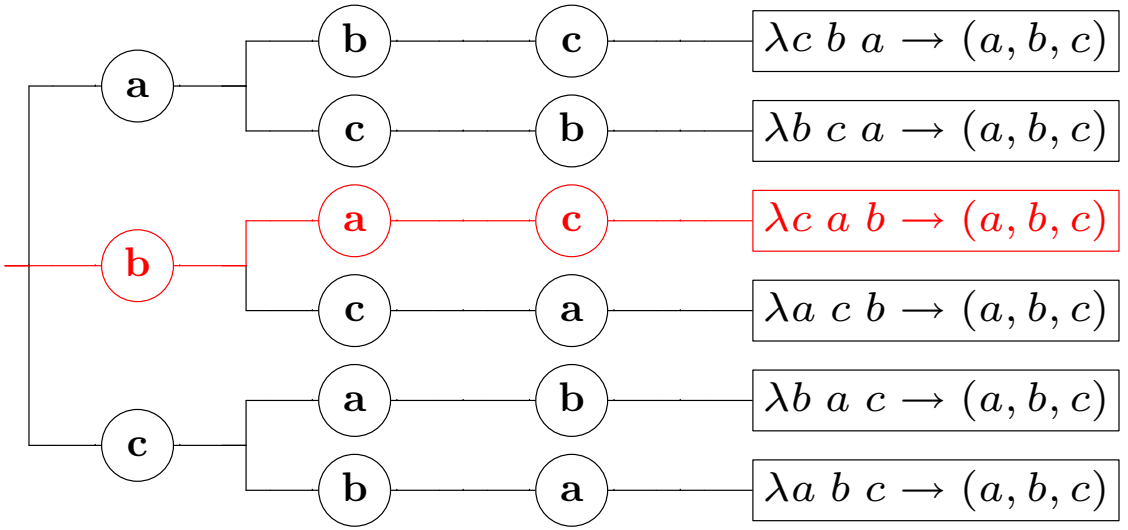
$$splitParser :: p\ a \longrightarrow (Maybe\ a, Maybe\ (p\ a))$$



Semantic functions are stored at the exits

$$
\begin{array}{lcl}
pABC & :: & Perms\ (Char, Char, Char) \\
pABC & = & pPerms\ ((,,) \mathbin{\langle\$\rangle} pSym\ \texttt{'a'} \mathbin{\langle*\rangle} pSym\ \texttt{'b'} \mathbin{\langle*\rangle} pSym\ \texttt{'c'})
\end{array}
$$

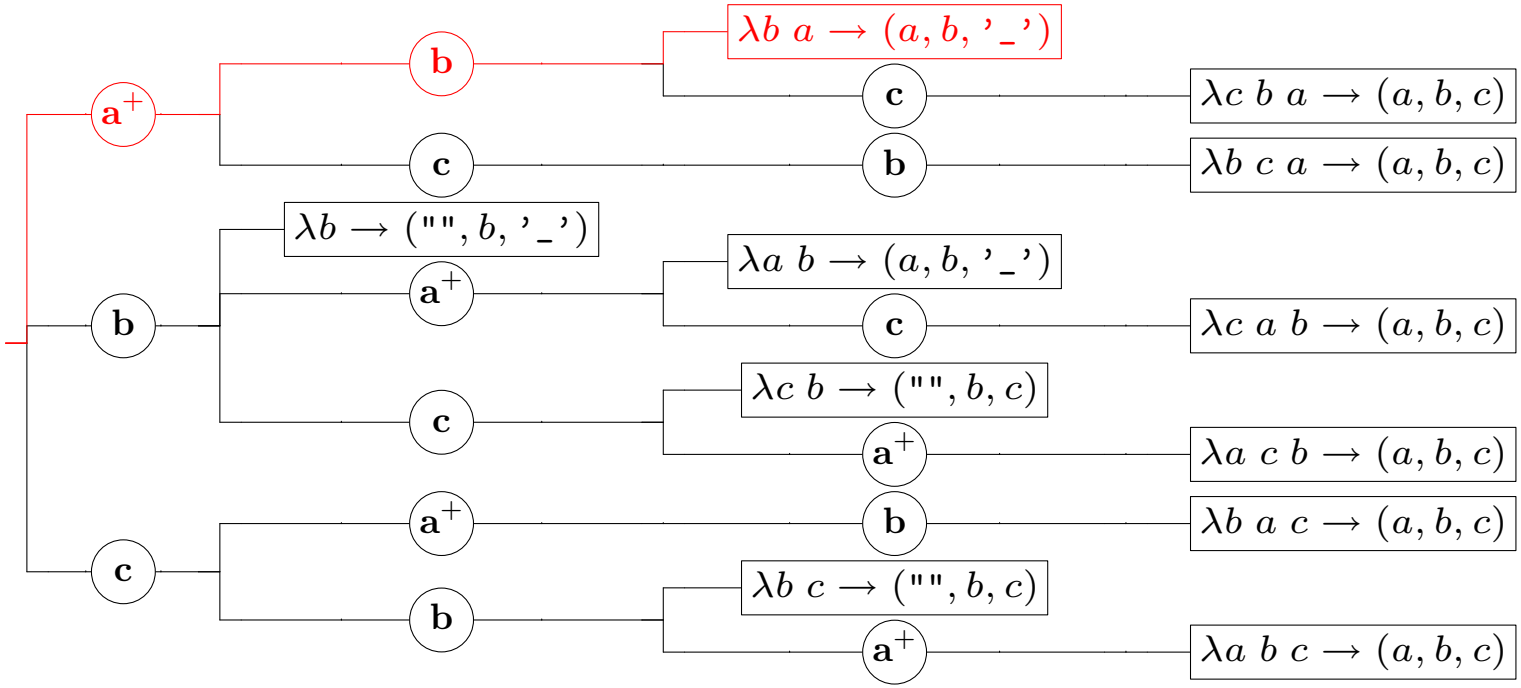
- In every exit a version of the semantic function is stored that that arranges the arguments in a uniform order.



Toy example revisited

$$pS :: Parser\ (String, Char, Char)$$
$$pS = pPerms\ ((,,) \lessdot\$\gtrdot pList\ (pSym\ \texttt{'a'}) \lessdot\ast\gtrdot pSym\ \texttt{'b'} \lessdot\ast\gtrdot pOptSym\ \texttt{'c'})$$

$$pOptSym\ x = pSym\ x \lhd\rhd pSucceed\ \texttt{'\_'}$$



Implementation: $Perms$ datatype

$$\textbf{data}\ Perms\ p\ a \quad = \ Choice\ \{\, exit :: Maybe\ a, branches :: [\, Branch\ p\ a\,]\,\}$$

$$\textbf{data}\ Branch\ p\ a \quad = \ \forall\ x.Br\ (p\ x)\ (Perms\ p\ (x \rightarrow a))$$

Note the use of an existential type to hide the specific order in which the elements occur.

Definitions of $fmap$ for $Perms\ p$ and $Branch\ p$:

$$\begin{aligned}&\textbf{instance}\ Functor\ (Perms\ p)\ \textbf{where}\\ &\quad fmap\ v_{a\rightarrow b}\ (Choice\ e_a\ bs_a) \ = \ Choice\ (fmap\ v_{a\rightarrow b}\ e_a)\\ &\qquad\qquad\qquad\qquad\qquad\qquad [fmap\ v_{a\rightarrow b}\ br_a \mid br_a \leftarrow bs_a]\end{aligned}$$

$$\begin{aligned}&\textbf{instance}\ Functor\ (Branch\ p)\ \textbf{where}\\ &\quad fmap\ v_{a\rightarrow b}\ (Br\ p_x\ t_{x\rightarrow a}) \quad = \ Br\ p_x\ (fmap\ (v_{a\rightarrow b}\circ)\ t_{x\rightarrow a})\end{aligned}$$

Implementation: transforming $Perms$ into a parser

$$
\begin{array}{lll}
pPerms\ t_a & = & foldr\ (\langle\rangle)\ empty\ nonempties \\
\quad \textbf{where}\ empty & = & \textbf{case}\ exit\ t_a\ \textbf{of} \\
 & & \quad Nothing \rightarrow pFail \\
 & & \quad Just\ v_a \rightarrow pSucceed\ v_a \\
nonempties & = & [\ flip\ (\$) \langle\$\rangle\ p_x \langle\circledast\rangle\ pPerms\ t_{x \rightarrow a} \\
 & & \mid Br\ p_x\ t_{x \rightarrow a} \leftarrow branches\ t_a \\
 & &]
\end{array}
$$

Building the $Perms$ structure

$$
\begin{array}{lcl}
succeedPerms & :: & a \to Perms\ p\ a \\
succeedPerms\ x & = & Choice\ (Just\ x)\ [\,] \\
 & & \\
(\ll\$\gg) & :: & IsParser\ p \Rightarrow (a \to b) \to p\ a \to Perms\ p\ b \\
f \ll\$\gg p & = & succeedPerms\ f \ll\circledast\gg p \\
 & & \\
(\ll\circledast\gg) & :: & IsParser\ p \Rightarrow Perms\ p\ (a \to b) \to p\ a \to Perms\ p\ b \\
perms \ll\circledast\gg p & = & add\ (splitParser\ p)\ perms
\end{array}
$$

Example of usage:

```
pS  :: Parser (String, Char, Char)
pS  = pPerms ((,,) <$> pList (pSym 'a')
                   <*> pSym 'b'
                   <*> pOptSym 'c'
             )
```

Building the $Perms$ structure (continued)

empty part	non-empty part	
$Nothing$	$Nothing$	$pFail$
$Just$	$Nothing$	$pSucceed$
$Nothing$	$Just$	required element
$Just$	$Just$	optional element

$$
\begin{array}{lll}
add & :: & (Maybe\ a, Maybe\ (p\ a)) \rightarrow Perms\ p\ (a \rightarrow b) \rightarrow Perms\ p\ b \\
add\ sp_a\ t_{a\rightarrow b}@(Choice\ e_{a\rightarrow b}\ bs_{a\rightarrow b}) & = & \mathbf{case}\ sp_a\ \mathbf{of} \\
\quad (Nothing, Nothing) & \rightarrow & Choice\ Nothing\ [\,] \\
\quad (Just\ v_a, Nothing) & \rightarrow & fmap\ (\$v_a)\ t_{a\rightarrow b} \\
\quad (Nothing, Just\ p_a) & \rightarrow & Choice\ Nothing\ (insert\ p_a) \\
\quad (Just\ v_a, Just\ p_a) & \rightarrow & Choice\ (fmap\ (\$v_a)\ e_{a\rightarrow b})\ (insert\ p_a) \\
\mathbf{where}\ insert\ p_a & = & Br\ p_a\ t_{a\rightarrow b} : [Br\ p_x\ (add\ sp_a\ t_{a\rightarrow x\rightarrow b}) \\
 & & \quad |Br\ p_x\ t_{x\rightarrow a\rightarrow b} \leftarrow bs_{a\rightarrow b} \\
 & & \quad , \mathbf{let}\ t_{a\rightarrow x\rightarrow b} = fmap\ flip\ t_{x\rightarrow a\rightarrow b} \\
 & & \quad]
\end{array}
$$

Conclusions

- Existential types
  - Reordering of constituents is type-safe and hidden from user
- Lazy evaluation
  - Only small part of intermediate data structure is evaluated
- No need for a second phase: errors are reported by underlying parsing library
- Can be made to work with (almost) any parser combinator library
- http://www.cs.uu.nl/groups/ST/Software/UU_Parsing/