JFP **14** (6): 635–646, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796804005143 Printed in the United Kingdom

FUNCTIONAL PEARL
Parsing permutation phrases

ARTHUR I. BAARS, ANDRES LÖH and S. DOAITSE SWIERSTRA
Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands
(*e-mail:* {arthurb,andres,doaitse}@cs.uu.nl)

Abstract

A permutation phrase is a sequence of elements (possibly of different types) in which each element occurs exactly once and the order is irrelevant. Some of the permutable elements may be optional. We show how to extend a parser combinator library with support for parsing such free-order constructs. A user of the library can easily write parsers for permutation phrases and does not need to care about checking and reordering the recognized elements. Applications include the generation of parsers for attributes of XML tags and Haskell's record syntax.

1 Introduction

Parser combinator libraries highlight the strengths of functional programming languages: higher-order functions and the possibility to define new infix operators allow parsers to be expressed in a concise and natural notation that closely resembles the syntax of EBNF grammars. At the same time, the user has the full abstraction power of the underlying programming language at hand. Complex, often recurring patterns can be expressed by defining new combinators.

A specific parsing problem is the recognition of permutation phrases. A permutation phrase is a sequence of elements (possibly of different types) in which each element occurs exactly once and the order is irrelevant. Some of the permutable elements may be optional. Since permutation phrases are not easily expressed by a context-free grammar, the usual approach is to tackle this problem in two steps: first parse a relaxed version of the grammar, then check whether the recognized elements form a permutation of the expected elements. This method, however, has a number of disadvantages. Dealing with a permutation of typed values is quite cumbersome, and the problem is often avoided by encoding the values in a universal representation, thus adding an extra level of interpretation. Furthermore, because of the two steps involved, error messages cannot be produced until a larger part of the input has been consumed, and special care has to be taken to make them point to the right position in the code.

Permutation phrases have been proposed by Cameron (1993) as an extension to EBNF grammars, not aiming at greater expressive power, but at more clarity. Cameron also presents a pseudo-code algorithm to parse permutation phrases with optional elements efficiently in an imperative setting. It fails, however, to address the types of the constituents.

We show how to extend any existing parser combinator library with support for parsing permutations of typed, potentially optional elements.

Possible applications include the implementation of Haskell's *read* function where it is desirable to parse the fields of a data type with labelled fields in any order, and the parsing of XML tags which have large sets of potentially optional attributes that may occur in any order. For instance, a parser for the XHTML image tag with some of its attributes can be written as follows:

```
imgtag       = token "<"  *> token "img"  *> attrs <*  token "/>"
   where attrs = permute (Img <$$>              field "src"      uri
                              <||>              field "alt"      string
                              <||> optional (field "longdesc" uri)
                              <||> optional (field "height"   int)
                              <||> optional (field "width"    int)
                         )
```

The combinator <||> is used to separate parsers for permutable elements, and <$$> can be used to apply a semantic function. Parsers for permutation phrases have to be enclosed by a call to *permute*.

Our approach makes use of two features that are not provided by all functional programming languages: existentially quantified data types are used to encode reordering information that permutes the recognized elements to a canonical order. Additionally, we utilize lazy evaluation to make the resulting implementation efficient. The administrative part of parsing permutation phrases has a quadratic time complexity in the number of permutable elements. The size of the code, however, is linear in the number of permutable elements.

We therefore choose Haskell as implementation language. Existential types are not part of the Haskell 98 standard (Peyton Jones, 2003), but are supported by several current Haskell implementations.

The paper is organized as follows: section 2 explains the parser combinators we build upon. Section 3 presents the idea of dealing with permutations in terms of permutation trees and explains how such trees are built and converted into parsers. In section 4, we take a brief look at the applications mentioned above: the parsing of data types with labelled fields and the parsing of XML attribute sets. Section 5 concludes.

2 Parsing using combinator libraries

The use of a combinator library for describing parsers instead of writing them by hand or generating them from a separate formalism is a well-known technique in functional programming. As a result, there are several excellent libraries around. For this reason we just briefly present the interface we will assume in subsequent sections of this paper, but do not go into the details of the implementation. We want to stress, however, that our extension is not tied to any specific library.

```
infixl 3 <|>
infixl 4 <*>
class Parser p where
  fail    :: p a
  succeed :: a → p a
  symbol  :: Char → p Char
  (<*>)   :: p (a → b) → p a → p b
  (<|>)   :: p a → p a → p a
```

Fig. 1. Type class for parser combinators.

```
infixl 4 <$>, <$ , *>, <*
(<$>)      :: Parser p ⇒ (a → b) → p a → p b
f <$> p    = succeed f <*> p
(<$ )      :: Parser p ⇒ a → p b → p a
x <$  p    = const x <$> p
(<* )      :: Parser p ⇒ p a → p b → p a
p <*  q    = const <$> p <*> q
( *>)      :: Parser p ⇒ p a → p b → p b
p  *> q    = flip const <$> p <*> q
parens     :: Parser p ⇒ p a → p a
parens p = symbol '('  *> p <*  symbol ')'
```

Fig. 2. Some useful parser combinators.

We make use of a simple interface (Röjemo, 1995; Swierstra & Duponcheel, 1996) that is parametrized by the result type of the parsers and assumes a list of characters as input. It can easily be implemented by straightforward list-of-successes parsers (Fokker, 1995; Wadler, 1985). Our permutation parsers have also been implemented for more advanced libraries, such as the fast, error-correcting parser combinators of Swierstra (2001) and the monadic-style (Hutton & Meijer, 1988) combinator library Parsec (Leijen, 2001).

The parser interface used here is given as a type class declaration in Figure 1. The function *fail* represents the parser that always fails, whereas *succeed* never consumes any input and always returns the given result value. The parser *symbol* accepts solely the given character as input. If this character is encountered, *symbol* consumes and returns this character, otherwise it fails. The <*> operator denotes the sequential composition of two parsers, where the result of the first parser is applied to the result of the second. Finally, the operator <|> expresses a choice between two parsers.

Many useful combinators can be built on top of these basic ones. A small selection that we use in this paper is presented in Figure 2. The most notable of the derived combinators is the application operator <$>, a parser transformer that can be used to apply a semantic function to a parse result. It is defined in terms of *succeed* and <*>.

3 Permutation parsers

We compute the parser for a permutation phrase not directly, but from an intermediate tree. We first introduce an auxiliary data type to represent such

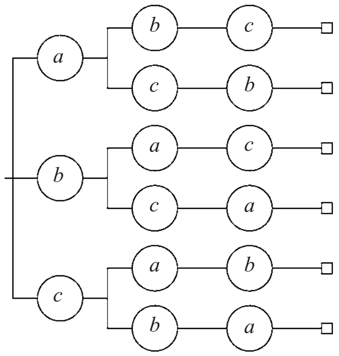

Fig. 3. A permutation tree containing three elements.

permutation trees, and show how they can be converted to parsers. Later, we introduce special combinators that provide a convenient notation for constructing permutation trees.

3.1 Permutation trees

A permutation phrase of a set of elements can be expanded into an EBNF definition by summing up all possible permutations of the elements. Consider for example the permutation phrase of three elements a, b, and c. Using Cameron's notation for permutation phrases, we can write it as:

$$s ::= \langle\!\langle a \mid\mid b \mid\mid c \rangle\!\rangle$$

Expanding and subsequently left-factorizing this permutation phrase gives us the following EBNF production rule:

$$\begin{array}{rl} s ::= & a\ b\ c \mid a\ c\ b \\ \mid & b\ a\ c \mid b\ c\ a \\ \mid & c\ a\ b \mid c\ b\ a \end{array} \quad \rightsquigarrow \quad \begin{array}{rl} s ::= & a\ (b\ c \mid c\ b) \\ \mid & b\ (a\ c \mid c\ a) \\ \mid & c\ (a\ b \mid b\ a) \end{array}$$

The left-factorized production rule can be represented by a tree as illustrated in Figure 3. Each path from the root to a leaf in the tree represents a particular permutation. Permutations with a common prefix share the same subtree, hence the number of choices in each node is limited by the number of permutable elements. Such a permutation tree is very suitable as intermediate data structure for a permutation parser. Parsing a permutation phrase boils down to checking whether the input matches one of the paths in the permutation tree.

If the grammar (and thus the permutation tree) is ambiguous, large parts of the tree might need to be evaluated before it can be decided which path must be followed. Therefore, ambiguous grammars should (as always) be avoided. However, if the ambiguity in the grammar stems from optional elements in the permutation phrase, the permutation tree can be modified in a simple way to resolve the ambiguity.

In section 3.2, we develop an implementation for permutations without optional elements, and in section 3.3 extend this solution to cover optionality.

3.2 Permutation trees without optional elements

We introduce a data type *Perms* for permutation trees, which is parametrized by a type constructor p (e.g. the parser type) and a result type a.

$$\begin{array}{lll}
\textbf{data}\ \mathit{Perms}\ p\ a & = & \mathit{Empty}\ a \\
 & \mid & \mathit{Choice}\ [\mathit{Branch}\ p\ a] \\
\textbf{data}\ \mathit{Branch}\ p\ a & = & \exists\, x\, .\ \mathit{Br}\ (p\ x)\ (\mathit{Perms}\ p\ (x \to a))
\end{array}$$

The *Empty* constructor represents a leaf of the tree, the *Choice* constructor stores a branching node. A single branch is constructed using *Br* and consists of an element, represented as a parser, plus a subtree. As the types of the elements may differ between branches, we hide their types by existentially quantifying the x in the definition of *Br*. The subtrees have an element type that is different from the type of the original tree, making *Perms* a non-regular data type. Each subtree must contain information how to construct a value of the result type a from a value of the element's type. This is achieved by storing at the end of each path a function that effectively reorders the elements on the path.

To show that reordering is determined by the type of the components, we will henceforth write the type (or the element type for type constructors) of a variable as an index to its name.

The idea that each path in the tree represents the parser for one of the possible permutations is reflected by the following simple conversion function from permutation trees to parsers. A leaf, i.e. an *Empty*, is interpreted as an always succeeding parser; a *Choice* constructor is converted into a choice between parsers. For each branch, we have to supply the value resulting from the parser to the reordering function resulting from the subtree.

$$\begin{array}{ll}
\mathit{permute} & :: \mathit{Parser}\ p \Rightarrow \mathit{Perms}\ p\ a \to p\ a \\
\mathit{permute}\ (\mathit{Empty}\ v_a) & = \mathit{succeed}\ v_a \\
\mathit{permute}\ (\mathit{Choice}\ bs_a) & = \mathit{choice}\ (\mathit{map}\ \mathit{branch}\ bs_a) \\
\mathit{branch} & :: \mathit{Parser}\ p \Rightarrow \mathit{Branch}\ p\ a \to p\ a \\
\mathit{branch}\ (\mathit{Br}\ p_x\ t_{x\to a}) & = (\lambda x\ f \to f\ x) \mathbin{\langle\$\rangle} p_x \mathbin{\langle*\rangle} \mathit{permute}\ t_{x\to a} \\
\mathit{choice} & :: \mathit{Parser}\ p \Rightarrow [p\ a] \to p\ a \\
\mathit{choice} & = \mathit{foldr}\ (\langle|\rangle)\ \mathit{fail}
\end{array}$$

Lazy evaluation plays an important role here, in that it ensures that the permutation tree is never computed completely. In fact, for a permutation of n elements, just the n tree elements immediately below the root of the tree are required to decide which subtree will be used to parse the rest of the permutation. From then on, only that subtree (a permutation tree of size $n-1$) is relevant. Iterating this argument leads to a complexity of only $O(n^2)$.

There are two potential problems here. First, the underlying parser combinator library might try to optimize parsing behaviour by evaluating different possible paths.

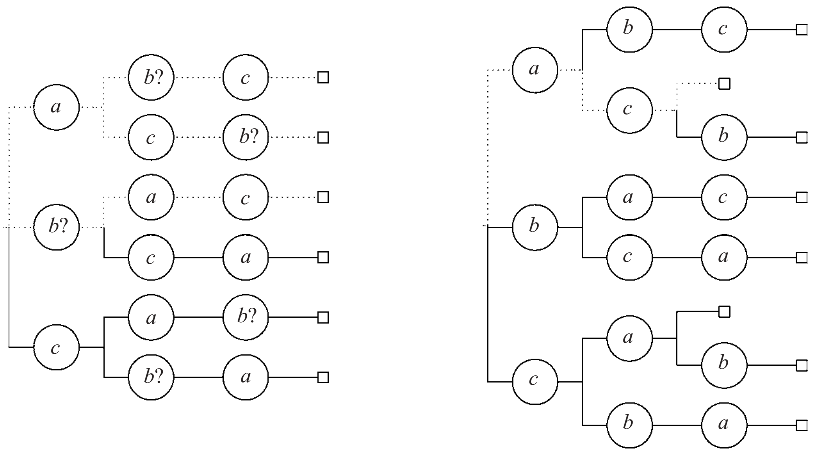

Fig. 4. The ambiguous and the adapted permutation tree for optional *b*.

This is problematic because permutation trees are so large that the precomputation is clearly undesireable. If the library has such features, they should be locally disabled for permutation trees. Second, repeated parsing of different permutations might cause multiple paths of the tree to be evaluated. In practice, however, the number of different permutations that actually occur in the input is small compared to the number of possible permutations.

3.3 Permutation trees with optional elements

Optional elements can be represented by parsers that can recognize the empty string and return a default value for this element. However, if *permute* is called on a permutation tree that contains such parsers, the resulting parser is ambiguous.

Consider the left tree in Figure 4, which contains all permutations of *a*, an optional *b* and *c*. Suppose we want to recognize *ac*. This can be done in three different ways since the empty *b* can be recognized before *a*, after *a* or after *c*. The three possible parses are shown as dotted paths in the figure. Fortunately, it is irrelevant for the result of a parse where exactly the empty *b* is derived, since order is not important. This allows us to use a strategy similar to the one proposed by Cameron (1993): parse nonempty constituents as they are seen and allow the parser to stop if all remaining elements are optional. When the parser stops the default values are returned for all optional elements that have not been recognized. The right tree in Figure 4 depicts this strategy for our three element example. The additional leaves mark the positions where the parser is allowed to stop. The string *ac* can now be parsed in only one way.

To implement this strategy we need to be able to determine whether a parser can derive the empty string and split it into its default value and its non-empty part, i.e. a parser that behaves the same except that it does not recognize the empty string. Both parts are represented as *Maybe* values: the first component is *Nothing* if and only if the parser cannot recognize the empty string. If the second component is *Nothing*, the parser does not consume any input. Hence, it is either a *succeed* (i.e. it just carries semantics) or a *fail*. The splitting of parsers is represented by the *ParserSplit* class that is an extension of the normal *Parser* class. If the underlying parser combinator library cannot be easily adapted to cover this extension, one can alternatively introduce additional combinators, similar to $\triangleleft\!\triangleright$ and $\lll\!\$\!\ggg$, and let the user mark the optional elements explicitly (Leijen, 2001).

$$\begin{array}{l}\textbf{class}\ \mathit{Parser}\ p \Rightarrow \mathit{ParserSplit}\ p\ \textbf{where}\\ \quad \mathit{split} :: p\ a \to (\mathit{Maybe}\ a, \mathit{Maybe}\ (p\ a))\end{array}$$

In the solution that does not deal with optional elements a parser for a permutation follows a path from the root of a permutation tree to a leaf, i.e. an *Empty* node. In the presence of optional elements, however, a parser may stop in any node that stores only optional elements. We adapt the *Perms* data type to incorporate this additional information. If all elements stored in a tree are optional then their default values are stored in *defaults*, otherwise *defaults* is *Nothing*. The parser stored in each *Branch* is not allowed to derive the empty string. We can express the former *Empty* constructor as a function now.

$$\begin{array}{ll}\textbf{data}\ \mathit{Perms}\ p\ a & = \mathit{Choice}\ \{\mathit{defaults} :: (\mathit{Maybe}\ a), \mathit{branches} :: [\mathit{Branch}\ p\ a]\}\\ \mathit{empty}\ x & = \mathit{Choice}\ (\mathit{Just}\ x)\ []\end{array}$$

The function *permute* is straightforwardly adapted to the generalized data type:

$$\begin{array}{lcl}\mathit{permute} & :: & \mathit{Parser}\ p \Rightarrow \mathit{Perms}\ p\ a \to p\ a\\ \mathit{permute}\ (\mathit{Choice}\ d_a\ bs_a) & = & \mathit{exit}\ d_a\\ & \triangleleft\!\triangleright & \mathit{choice}\ (\mathit{map}\ \mathit{branch}\ bs_a)\\ \mathit{exit} & :: & \mathit{Parser}\ p \Rightarrow \mathit{Maybe}\ a \to p\ a\\ \mathit{exit}\ (\mathit{Just}\ v_a) & = & \mathit{succeed}\ v_a\\ \mathit{exit}\ \mathit{Nothing} & = & \mathit{fail}\end{array}$$

Note the similarity to the old definition. The only difference is the possibility to exit early where a default value is present in the tree.

3.4 Building a permutation tree

Permutation trees are created by adding the elements of the permutation one by one to an initially empty tree. The function *add* takes a pair consisting of an optional default value and a parser that does not recognize the empty string, and adds it to an existing tree.

$$\begin{array}{ll}\mathit{add} & :: (\mathit{Maybe}\ a, p\ a) \to \mathit{Perms}\ p\ (a \to b) \to \mathit{Perms}\ p\ b\\ \mathit{add}\ (d_a, p_a)\ t_{a\to b} & = \textbf{case}\ t_{a\to b}\ \textbf{of}\\ & \quad\ \mathit{Choice}\ d_{a\to b}\ bs_{a\to b} \to\\ & \qquad \mathit{Choice}\ (\mathit{ap}\ d_{a\to b}\ d_a)\\ & \qquad\qquad (\mathit{Br}\ p_a\ t_{a\to b} : \mathit{map}\ \mathit{ins}\ bs_{a\to b})\\ \multicolumn{2}{l}{\quad\textbf{where}\ \mathit{ins}\ (\mathit{Br}\ p_x\ t_{x\to a\to b}) = \mathit{Br}\ p_x\ (\mathit{add}\ (d_a, p_a)\ (\mathit{mapPerms}\ \mathit{flip}\ t_{x\to a\to b}))}\end{array}$$

Having a default value means that the parser described by the permutation tree can accept the empty string. Surely, for the constructed tree, that is only possible if both the original tree has a default value and d_a is not *Nothing*. Then, the new default value can be built from the two using function application. The function *ap* does exactly this – it is function application lifted to *Maybe* types.

$$\begin{array}{lcl}
\mathit{ap} & :: & \mathit{Maybe}\ (a \to b) \to \mathit{Maybe}\ a \to \mathit{Maybe}\ b \\
\mathit{ap}\ (\mathit{Just}\ f)\ (\mathit{Just}\ x) & = & \mathit{Just}\ (f\ x) \\
\mathit{ap}\ \_\ \ \ \ \ \ \ \ \_ & = & \mathit{Nothing}
\end{array}$$

We can add a new element (d_a, p_a) to a permutation tree by inserting it in all possible positions to every permutation that is already in the tree. The function *add* explicitly constructs the tree that represents the permutation in which p_a is the top element. Additionally, for each branch of the original tree, the top element is left unchanged, and (d_a, p_a) is inserted everywhere (by a recursive call to *add*) in the subtree. Because the new element and the top element of the branch are now swapped, the function resulting from the subtree of the branch gets its arguments passed in the wrong order, which is repaired by applying *flip* to that function.

The function *mapPerms* is a mapping function on permutation trees. In a branch, $f_{a\to b}$ is composed with the function that results from the subtree.

$$\begin{array}{lcl}
\mathit{mapPerms} & :: & (a \to b) \to \mathit{Perms}\ p\ a \to \mathit{Perms}\ p\ b \\
\mathit{mapPerms}\ f_{a\to b}\ (\mathit{Choice}\ d_a\ \mathit{bs}_a) & = & \mathit{Choice}\ (\mathit{fmap}\ f_{a\to b}\ d_a) \\
 & & \qquad\qquad (\mathit{map}\ (\mathit{mapBranch}\ f_{a\to b})\ \mathit{bs}_a) \\
\mathit{mapBranch} & :: & (a \to b) \to \mathit{Branch}\ p\ a \to \mathit{Branch}\ p\ b \\
\mathit{mapBranch}\ f_{a\to b}\ (\mathit{Br}\ p_x\ t_{x\to a}) & = & \mathit{Br}\ p_x\ (\mathit{mapPerms}\ (f_{a\to b}\circ)\ t_{x\to a})
\end{array}$$

We now define two operators for building permutation trees. The first is an operator that extends a permutation tree with a new element.

$$\begin{array}{l}
(\lhd\!\mid\!\rhd)\quad :: \mathit{ParserSplit}\ p \Rightarrow \mathit{Perms}\ p\ (a \to b) \to p\ a \to \mathit{Perms}\ p\ b \\
t \lhd\!\mid\!\rhd p = \\
\quad \mathbf{case}\ \mathit{split}\ p\ \mathbf{of} \\
\qquad (\mathit{Just}\ e\ \ \ ,\mathit{Just}\ \mathit{ne}\ ) \to \mathit{add}\ (\mathit{Just}\ e\ \ ,\mathit{ne})\ t \qquad\quad \text{-- optional element} \\
\qquad (\mathit{Nothing},\mathit{Just}\ \mathit{ne}\ ) \to \mathit{add}\ (\mathit{Nothing},\mathit{ne})\ t \qquad\quad \text{-- required element} \\
\qquad (\mathit{Just}\ e\ \ \ ,\mathit{Nothing}) \to \mathit{mapPerms}\ (\lambda f \to f\ e)\ t \quad \text{-- pure semantics} \\
\qquad (\mathit{Nothing},\mathit{Nothing}) \to \mathit{Choice}\ \mathit{Nothing}\ [\,] \qquad\quad \text{-- fail}
\end{array}$$

The second provides an empty permutation tree with initial semantics. It has a similar functionality as the $\lhd\!\$\!\rhd$ for normal parsers.

$$\begin{array}{l}
(\ll\!\$\!\gg)\quad :: \mathit{ParserSplit}\ p \Rightarrow (a \to b) \to p\ a \to \mathit{Perms}\ p\ b \\
f \ll\!\$\!\gg p = \mathit{empty}\ f \lhd\!\mid\!\rhd p
\end{array}$$

An example with three permutable elements, corresponding to the tree in Figure 3, can now be realized by:

$$\mathit{permute}\ ((,,) \ll\!\$\!\gg \mathit{int} \lhd\!\mid\!\rhd \mathit{char} \lhd\!\mid\!\rhd \mathit{bool})$$

where *int*, *char*, and *bool* are parsers for literals of type *Int*, *Char*, and *Bool*, respectively. Then all permutations of an integer, a character and a boolean are accepted, and the results of a successful parse are combined using the triple constructor (,,), thus yielding a value of type (*Int*, *Char*, *Bool*).

3.5 Separators

Permutable elements are often separated by symbols that do not carry meaning – typically commas or semicolons. Consider extending the three-element example to the Haskell tuple syntax: not just the elements, but also the parentheses and the commas should be parsed. Since there is one separation symbol less than there are permutable elements, our current variant of *permute* cannot handle this problem.

Therefore we define *permuteSep* as a generalization of *permute* that accepts an additional parser for the separator as an argument. The semantics of the separators are ignored for the result.

$$\begin{array}{ll}
\mathit{permuteSep} & :: \mathit{Parser}\ p \Rightarrow p\ b \rightarrow \mathit{Perms}\ p\ a \rightarrow p\ a \\
\mathit{permuteSep}\ s\ \mathit{perm} & = \mathit{permuteSep}'\ (\mathit{succeed}\ ())\ s\ \mathit{perm}
\end{array}$$

The function *permuteSep*′ now converts a permutation tree into a parser in almost the same way as the former *permute*, except that before each permutable element a separator is parsed. To prevent that a separator is expected before the first permutable element, we make use of the following simple trick. The *permuteSep*′ function expects two extra arguments: the first one will be parsed immediately before the first element, and the second will be used subsequently. Using *succeed* () as first extra argument in *permuteSep* leads to the desired result.

$$\begin{array}{lll}
\mathit{permuteSep}' & :: & \mathit{Parser}\ p \Rightarrow p\ c \rightarrow p\ b \rightarrow \mathit{Perms}\ p\ a \rightarrow p\ a \\
\mathit{permuteSep}'\ \mathit{sf}\ s\ (\mathit{Choice}\ d_a\ \mathit{bs}_a) & = & \mathit{exit}\ d_a \\
 & \mathbin{<\!|\!>} & \mathit{sf}\ \mathbin{*\!>}\ \mathit{choice}\ (\mathit{map}\ (\mathit{branchSep}\ s)\ \mathit{bs}_a) \\
\mathit{branchSep} & :: & \mathit{Parser}\ p \Rightarrow p\ b \rightarrow \mathit{Branch}\ p\ a \rightarrow p\ a \\
\mathit{branchSep}\ s\ (\mathit{Br}\ p_x\ t_{x\rightarrow a}) & = & \mathit{flip}\ (\$) \mathbin{<\!\$\!>} p_x \mathbin{<\!*\!>} \mathit{permuteSep}'\ s\ s\ t_{x\rightarrow a}
\end{array}$$

The *permute* function can now be implemented in terms of *permuteSep*.

$$\begin{array}{l}
\mathit{permute} :: \mathit{Parser}\ p \Rightarrow \mathit{Perms}\ p\ a \rightarrow p\ a \\
\mathit{permute} = \mathit{permuteSep}\ (\mathit{succeed}\ ())
\end{array}$$

To return to the small example, triples of an integer, a character, and a boolean – in any order – are parsed by:

$$\mathit{parens}\ (\mathit{permuteSep}\ (\mathit{symbol}\ \text{'},\text{'})\ ((,,) \mathbin{<\!<\!\$\!>\!>} \mathit{int} \mathbin{<\!|\!|\!>} \mathit{char} \mathbin{<\!|\!|\!>} \mathit{bool}))$$

4 Applications

4.1 XML attributes

We now demonstrate the use of the permutation parsers by showing how to parse XML tags with attributes. For simplicity, we just consider one tag (the `img` tag of XHTML) and only deal with a subset of the attributes allowed. In a Haskell program, this tag might be represented by the following data type.

```
data XHTML = Img{src      :: URI
                ,alt      :: String
                ,longdesc :: Maybe URI
                ,height   :: Maybe Int
                ,width    :: Maybe Int
                }
           | ...
```

Our variant of the `img` tag has five attributes of three different types. We use Haskell's record syntax to keep track of the names. The first two attributes are mandatory whereas the others are optional. We choose the *Maybe* variant of their types to reflect this optionality. Our parser should be able to parse the attributes in any order, where any of the optional arguments may be omitted. For the parsing process, we ignore whitespace and assume that there is a parser

```
token :: Parser p => String -> p String
```

that consumes just the given token and fails on any other input.

Using the *permute* combinator, writing the parser for the `img` tag is easy:

```
imgtag        :: ParserSplit p => p XHTML
imgtag        = token "<"  *> token "img"  *> attrs <*  token "/>"
   where attrs = permute $ Img <$$>           field "src"       uri
                               <||>           field "alt"       string
                               <||> optional (field "longdesc"  uri)
                               <||> optional (field "height"    int)
                               <||> optional (field "width"     int)
optional      :: Parser p => p a -> p (Maybe a)
optional p    = Just <$> p <|> succeed Nothing
```

The order in which we denote the attributes determines the order in which the results are returned. Therefore, we can apply the *Img* constructor to form a value of the *XHTML* data type. The helper function *field* is used to parse a single attribute.

```
field      :: Parser p => String -> p a -> p a
field s p = token s  *> symbol '='  *> p
```

4.2 Haskell's record syntax

Haskell allows data types to contain labelled fields. If one wants to construct a value of that data type, one can make use of these names. The advantage is that the user does not need to remember the order in which the fields of the constructor have been defined. Furthermore, all fields are considered as optional. If a field is not explicitly set to a value, it is silently assumed to be $\perp$.

Whereas compilers support order-free syntax (the record fields in a program can be ordered arbitrarily), the *read* function expects the fields in the same order as in the data type declaration. The resulting asymmetry is unfortunate. Using the *permuteSep* combinator, it is an easy task to write more flexible parsers for data types with labelled fields.

```
justOrNothing   :: Parser p => p a -> p (Maybe a)
justOrNothing p  = Just <$  token "Just" <*> p
                <|> Nothing <$  token "Nothing"
img             :: ParserSplit p => p XHTML
img              = token "Img"  *> token "{"  *> fields <*  token "}"
  where fields   = permuteSep (symbol ',') $
                     Img <<$>> recordfield "src"      uri
                         <|>   recordfield "alt"      string
                         <|>   recordfield "longdesc" (justOrNothing uri)
                         <|>   recordfield "height"   (justOrNothing int)
                         <|>   recordfield "width"    (justOrNothing int)
```

We use *recordfield* here to parse a single optional record field, returning ⊥ if it is not present.

```
recordfield     :: Parser p => String -> p a -> p a
recordfield f p = field f p <|> succeed ⊥
```

5 Conclusion

We have shown how to extend a parser combinator library with the functionality to parse free-order constructs. It can be placed on top of any combinator library that implements the *Parser* interface. A user of the library can easily write parsers for free-order constructs and does not need to care about checking and reordering the parsed elements. Due to the use of existentially quantified types the implementation of reordering is type safe and hidden from the user.

The underlying parser combinators can be used to handle errors, such as missing or duplicate elements, since the extension inherits their error-reporting or error-repairing properties.

We have shown how our extension can be used to parse XML attributes and Haskell records. Other interesting examples mentioned by Cameron (1993) include citation fields in BIBTEX bibliographies and attribute specifiers in C declarations. Cameron's pseudo-code algorithm uses a similar strategy. It does not show, however, how to maintain type safety by undoing the change in semantics resulting from reordering, nor can it deal with the presence of separators between free-order constituents.

References

Cameron, R. D. (1993) Extending context-free grammars with permutation phrases. *ACM Lett. Program. Lang. Syst.* **2**(4), 85–94.

Fokker, J. (1995) Functional parsers. *Advanced Functional Programming, First International Spring School: LNCS 925*, pp. 1–23. Springer-Verlag.

Hutton, G. and Meijer, H. (1988) Monadic parser combinators. *J. Funct. Program.* **8**(4), 437–444.

Leijen, D. (2001) *Parsec, a fast combinator parser.* `http://www.cs.uu.nl/~daan/parsec.html`.

Peyton Jones, S. (2003) *Haskell 98 Language and Libraries.* Cambridge University Press. `http://www.haskell.org/report`.

Röjemo, N. (1995) *Garbage collection and memory efficiency in lazy functional languages.* PhD thesis, Chalmers University of Technology.

Swierstra, D. (2001) Combinator parsers: From toys to tools. In: Hutton, G. (editor), *Electronic Notes in Theoretical Computer Science*, vol. 41. Elsevier.

Swierstra, D. and Duponcheel, L. (1996) Deterministic, error correcting combinator parsers. *Advanced Functional Programming, Second International Spring School: LNCS 1129*, pp. 184–207. Springer-Verlag.

Wadler, P. (1985) How to replace failure with a list of successes. *Functional Programming Languages and Computer Architecture: LNCS 201*, pp. 113–128. Springer-Verlag.