Preserving order in non-order preserving parsers

Rui Guerra

Arthur Baars

Doaitse Swierstra

João Saraiva

institute of information and computing sciences,
utrecht university

technical report UU-CS-2005-025

www.cs.uu.nl

Preserving order in non-order preserving parsers

Rui Guerra Arthur Baars
Doaitse Swierstra João Saraiva

Institute of Information and Computing Sciences
Utrecht University
The Netherlands
{rui,arthurb,doaitse}@cs.uu.nl

Universidade do Minho,
Departamento de Informática,
Campus de Gualtar, 4710-057 Braga, Portugal
jas@di.uminho.pt

June 21, 2005

Abstract

A *non-order preserving* (NOP) parser returns a result that cannot be mapped back to the original input, because the order of elements may have changed. For reasons of reporting errors it is desirable to preserve the initial order in some way. We show how to combine several existing parsers to construct a NOP parser that preserves the initial order by building a reordering functions while parsing. Besides other examples, we show how to use our technique in a parser for permutation phrases.

1 Introduction

There are many good reasons why the concrete syntax of a language differs from the abstract syntax. The former has been designed to make programs easy to read and write and maintain, and to make them look beautiful, whereas the latter has been designed to facilitate program analysis and code generation. Sometimes the abstract syntax closely follows the structure of the concrete syntax and it is rather straightforward to recover the original program structure from the abstract structure, but unfortunately it is more often the case that information got lost in building the abstract syntax tree. One of the detrimental effects of this kind of program transformations is that when we want to give feedback to the user about the outcome of the analysis performed on his transformed program, e.g. in the form of error messages, this is usually can no longer be done in terms of the original program structure, but only by mentioning line numbers or similar information that was explicitly preserved for this purpose. This problem becomes more aggravating if we have an editor that is integrated with error reporting in order to give the programmer continuous feedback about the status of his program text.

A good error message should make a reference to the place in the original code where the error occurred, tells what happened, why it happened and how the error can be corrected or avoided. To be able to produce good error messages we thus need to keep somehow track of all the original information. On the other hand, we need to abstract from the language redundancy to keep the semantic analysis as simple as possible.

An example of the kind of re-orderings we want to deal with here can be found in the programming language Haskell, in which one can have type definitions, type specifications and function definitions in the program text, with no restriction on the order in which those elements appear. This has obvious advantages for the programmer since this enables him to group related definitions closely together in the program text. For semantic processing on the other hand it would be nice if the parser returned as a result three separate lists: one containing all the type definitions, a second one containing all the types for the functions to be defined, and a final one containing all the function definitions themselves.

A second example can be found in the input language for an attribute grammar system. The input basically is a long list of definitions of non-terminals, productions, attribute declarations and semantic functions, grouped in such a way that definitions that somehow belong together are lexically close. Also here semantic processing prefers a parser that returns a group of lists, one for each kind of element mentioned.

As a third example we take a small subset of the Java language, in which each **class** contains fields, methods and one constructor. These components can appear in any order. Is common after parsing to return the components in a pre-defined order. Indeed, the vast majority of parser libraries has shown that reordering results is advantageous for semantic analysis.

In this paper we show how to instrument an existing parser combinator library in such a way that information about the way a specific parser reorders elements is preserved, and may be used to map the abstract structure, that may be enriched with error messages and other feedback information, back into the original concrete structure.

The nice thing about using such combinators is that, once implemented, the techniques are available for free to everyone using the combinators: no further extra data for computing the inverse mapping has to be constructed and maintained explicitly.

We call parsers that reorder the recognized elements *non-order preserving(NOP).* A *NOP* parser is usually defined by composing several alternatives: one for each allowed order. The reordering used in constructing the result depends on the alternative used. As the most simple example of a *NOP* parser we define a parser that recognizes two elements, of which the order does not matter, but that have to be returned in a predefined order:

$$\begin{array}{lcl} \mathit{split} & :: & (\mathit{Parser}\ a, \mathit{Parser}\ b) \rightarrow \mathit{Parser}\ (a, b) \\ \mathit{split}\ (\mathit{parser}_a, \mathit{parser}_b) & = & \quad (,) \mathbin{<\$>} \mathit{parser}_a \mathbin{<*>} \mathit{parser}_b \\ & <|> & \mathit{flip}\ (,) \mathbin{<\$>} \mathit{parser}_b \mathbin{<*>} \mathit{parser}_a \\ & & \\ (,)\ x\ y & = & (x, y) \\ \mathit{flip}\ f\ x\ y & = & f\ y\ x \end{array}$$

The result of applying the parser *split* $(\mathit{parser}_{alp}, \mathit{parser}_{int})$ to the input `"1a"` is a pair $(\texttt{'a'}, 1)$, where parser_{alp} is a parser that recognizes a letter and parser_{int} is a parser that recognizes a digit.

In the pair that is constructed as the witness of a successful parse the information about which element came first in the input is lost. The function that gives meaning to the input syntactic structure is usually called *semantic* function, in this case is the function $(,)$ and the function *flip* $(,)$. The new information given by the semantic function can be useful for further analysis. Although this is a simple example, there are useful parser combinators that have similar reordering behavior. An example of this are the parsers for permutation phrases, that generalize the above case for two elements to any number of elements[1].

In this paper, we propose to build functions during parsing that remember the initial order of the input. The parser thus produces two different results: the abstract syntax tree and the inverse function. By not maintaining this inverse information as part of the result returned we keep things cleanly separated, and do not clutter further semantic processing with passing around this kind of information.

We tackle this problem in two steps: first we perform an abstract interpretation of the parser description, in which we combine the different parsers that constitute the *NOP* parser and subsequently we construct the parsers we are interested in, with the inverse functionality.

Our approach makes use of advanced features of modern functional programming languages: existentially quantified data types are used to combine parsers of different types. Additionally, we use lazy evaluation to make the resulting implementation efficient. We thus have chosen Haskell as an implementation language. Existential types are not part of the Haskell 98 standard [3], but are supported by almost all Haskell implementations.

The report is organized as follows: Section 2 explains the parser combinators we build upon. Section 3 presents the idea of combining parsers to obtain a *NOP* parser and explains how to extend them to recover the initial order. Section 4 we customize the result of the parser combinator. In section 5 we redefine the inverse function to handle inputs of different types. In section 6 we present an example that shows how to parse java syntax. Section 7 concludes.

2 Parsing using combinator libraries

Parser combinator libraries for functional programming languages are well-known and subject of active research. Features like higher-orderness 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. In this project we will focus on a specific kind of parsers, *NOP* parsers. For this reason we just briefly present the interface we will assume in subsequent sections. We want to stress, however, that our approach is not tied to any specific library.

We make use of a simple [4] interface 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 [2] and [5]. The parser interface used here is presented in figure 1; note that the precise definition of the type constructor *Parser* is not relevant, and may even differ from implementation to implementation.

The function *fail* represents the parser that always fails, whereas *succeed* never consumes any input and always returns the given argument as a result value. The parser *symbol* accepts solely its argument as input. If this character is encountered, *symbol* consumes and returns it, otherwise it fails. The <*> operator denotes the sequential composition of two parsers in which, in order to obtain the result of the combined parser, the result of the first parser is applied to the result of the second. The operator <|> expresses a choice between two parsers. Finally, the application operator <$> is a parser transformer that can be used to apply a semantic function to a parse result. It can be defined in terms of succeed and <*>. As an example of the use of these basic combinators we present the definition of the *pFoldr* combinator that will be used extensively in this presentation:

```
pFoldr                        :: (a → b → b) → b → Parser a → Parser b
pFoldr binop empty parser = pRest
                    where pRest = binop <$> parser <*> pRest <|> succeed empty
```

The function *pFoldr* is a *foldr* for parsers, i.e., it applies recursively the input *parser* until it fails, then it returns *empty*. Results are combined with the function *binop*, which has *empty* as its unit element. Many useful combinators can be built on the top of these basic ones.

3 Combining and inverting parsers

3.1 Combining and inverting two parsers

We start with a simple example. Suppose our input consists of a list of elements of type *a* and *b* in no specific order, and we want to construct a parser that returns a pair of lists: one containing all the elements of type *a* and one containing all the elements of type *b*. We start by defining a parser that recognizes

$$\begin{array}{lll}
\textbf{infixl}\ 3 \lhd\!\rhd & & \\
\textbf{infixl}\ 4 \lhd\!*\!\rhd, \lhd\!\$\!\rhd & & \\
\mathit{symbol} & :: & \mathit{Char} \to \mathit{Parser\ Char} \\
\mathit{fail} & :: & \mathit{Parser\ a} \\
\mathit{succeed} & :: & a \to \mathit{Parser\ a} \\
(\lhd\!\rhd) & :: & \mathit{Parser\ a} \to \mathit{Parser\ a} \to \mathit{Parser\ a} \\
(\lhd\!*\!\rhd) & :: & \mathit{Parser}\ (a \to b) \to \mathit{Parser\ a} \to \mathit{Parser\ b} \\
(\lhd\!\$\!\rhd) & :: & (a \to b) \to \mathit{Parser\ a} \to \mathit{Parser\ b} \\
f \lhd\!\$\!\rhd p & = & \mathit{succeed}\ f \lhd\!*\!\rhd p \\
\mathit{parse} & :: & \mathit{Parser\ a} \to \mathit{String} \to \mathit{Maybe\ a}
\end{array}$$

Figure 1: parser combinators

one element, either of type a or b, and that returns a function that inserts this element in the corresponding list in a pair of lists:

$$\begin{array}{l}
\mathit{a\_or\_b} :: (\mathit{Parser\ a}, \mathit{Parser\ b}\ ) \to \mathit{Parser}\ (([a],[b]) \to ([a],[b])) \\
\mathit{a\_or\_b}\ (\mathit{parser}_a, \mathit{parser}_b) = \ (\ \mathit{put}_a \lhd\!\$\!\rhd \mathit{parser}_a \lhd\!\rhd \mathit{put}_b \lhd\!\$\!\rhd \mathit{parser}_b) \\
\qquad\qquad \textbf{where}\ \mathit{put}_a\ a\ (\mathit{res}_a, \mathit{res}_b) = (a : \mathit{res}_a, \mathit{res}_b) \\
\qquad\qquad\qquad\quad \mathit{put}_b\ b\ (\mathit{res}_a, \mathit{res}_b) = (\mathit{res}_a, b : \mathit{res}_b)
\end{array}$$

We can use the function *pFoldr* to apply this parser repeatedly. When it fails or reaches the end of the input, *pFoldr* returns a pair of empty lists $([\,],[\,])$. We define the parser combinator *pMerged* that constructs a *NOP* parser from a pair of parsers.

$$\begin{array}{lll}
\mathit{pMerged} & :: & (\mathit{Parser\ a}, \mathit{Parser\ b}) \to \mathit{Parser}\ ([a],[b]) \\
\mathit{pMerged}\ (\mathit{parser}_a, \mathit{parser}_b) & = & \mathit{pFoldr\ apply}\ ([\,],[\,])\ (\mathit{a\_or\_b}\ (\mathit{parser}_a, \mathit{parser}_b)) \\
\mathit{apply\ element\ rest} & = & \mathit{element\ rest}
\end{array}$$

As we can see, the put_a function inserts the value returned by parser_a in the list res_a, that contains all the values of type a. The put_b function does the same as put_a but for values of type b. Suppose now that $\mathit{parser}_{\mathit{int}}$ and $\mathit{parser}_{\mathit{alp}}$ are parsers that recognize and return digits and letters, respectively. Then the *NOP* parser that combines these parsers is represented by:

$$\begin{array}{l}
\mathit{int\_alp} :: \mathit{Parser}\ ([\mathit{Int}],[\mathit{Char}]) \\
\mathit{int\_alp} = \mathit{pMerged}\ (\mathit{parser}_{\mathit{int}}, \mathit{parser}_{\mathit{alp}})
\end{array}$$

The parser *int_alp* recognizes a list of integers and characters and returns them in a pair of lists. Thus the result of applying *int_alp* to the input string `"A1bC2"` is the pair $([1,2],$ `"AbC"`$)$. Using the parser combinator *pMerged* we can define any *NOP* made up from two simple parsers.

Having such a parser, the question that arises now is how to recover the original order of the input? The obvious answer is to build a function that memorizes that order and it is able to map back the original input. In other words, we are looking for an *invert* function that takes an input like $([1, 2],$ `"AbC"`$)$ and reconstructs the initial order `"A1bC2"`.

The function that memorizes the order of the original input has to be constructed when the result is re-ordered. That is to say, it has to be constructed during parsing. So the parser function has to perform three tasks:

- Detecting whether the input follows the syntactic rules specified by underlying grammar, or not;
- Giving semantic meaning to the input (which can preserve the input order or not);
- Constructing an inverse function that recovers the initial order of the input (only in the case of a *NOP* parser).

To recover the initial order of the input, we have to construct an *invert* function, during parsing. The *invert* function will take a pair of lists and reorders its elements into a single list. Standard lists, however, require that all elements have the same type. This is not the case in our running example. To simplify our introductory explanation we take for granted that the two lists have the same type. We will return to this subject in section 5. The *invert* function takes a pair of lists and returns them in the initial order and we use a type synonym to represent its type:

$$\textbf{type}\ \mathit{Invert}\ s = ([s], [s]) \rightarrow [s]$$

The *invert* function has to remember in which side of the pair the parsing result was stored. When the semantic function inserts the parser result in the left side of the pair, the *invert* function will take an element from the left side and otherwise it will take an element from the right side. We define two *put* and two *get* functions. As before, we combine each parser with a *put* function using the combinator $<\$>$. Each *put* function will update the *inv* function using the *get* function. The functions get_a and get_b update the invert function by taking an element from the left side of the input or from the right side, respectively. For a parser that failed or reached the end of the input, the resulting *invert* function returns an empty list.

$$\begin{array}{l}
\mathit{pMerged} :: (\mathit{Parser}\ a, \mathit{Parser}\ b) \rightarrow \mathit{Parser}\ (([a],[b]), \mathit{Invert}\ s)\\
\mathit{pMerged}\ (\mathit{parser}_a, \mathit{parser}_b) = \mathit{pFoldr}\ (\$)\ (([\,],[\,]), \mathit{const}\ [\,])\\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{put}_a <\$> \mathit{parser}_a <|> \mathit{put}_b <\$> \mathit{parser}_b)\\
\qquad \textbf{where}\\
\qquad\quad \mathit{put}_a\ a\ ((\mathit{res}_a, \mathit{res}_b), \mathit{inv}) = ((a : \mathit{res}_a, \mathit{res}_b), \mathit{get}_a\ \mathit{inv})\\
\qquad\quad \mathit{put}_b\ b\ ((\mathit{res}_a, \mathit{res}_b), \mathit{inv}) = ((\mathit{res}_a, b : \mathit{res}_b), \mathit{get}_b\ \mathit{inv})
\end{array}$$

$$\begin{array}{ll}
\mathit{get}_a\ \mathit{inv}\ ([\,], \mathit{res}_b) & = \mathit{inv}\ ([\,], \mathit{res}_b)\\
\mathit{get}_a\ \mathit{inv}\ (a : \mathit{res}_a, \mathit{res}_b) & = a : \mathit{inv}\ (\mathit{res}_a, \mathit{res}_b)\\
\mathit{get}_b\ \mathit{inv}\ (\mathit{res}_a, [\,]) & = \mathit{inv}\ (\mathit{res}_a, [\,])\\
\mathit{get}_b\ \mathit{inv}\ (\mathit{res}_a, b : \mathit{res}_b) & = b : \mathit{inv}\ (\mathit{res}_a, \mathit{res}_b)
\end{array}$$

The resulting *NOP* parser returns a pair where the first element is another pair with the recognized symbols and the second element is the *invert* function.

Suppose that we want to build a *NOP* parser that recognizes letters and distinguishes upper case from lower case letters. If $parser_{low}$ and $parser_{upp}$ are parsers that recognize lower case and upper case letters, respectively, then our parser is easily represented by:

$$\begin{aligned} &low\_upp :: Parser\ ((String, String), Invert\ a) \\ &low\_upp = pMerged\ (parser_{low}, parser_{upp}) \end{aligned}$$

The parser *low_upp* besides returning the two lists with the parser results, it also returns the *invert* function. Thus applying the parser *low_upp* to the string `"AaBb"` produces the pair $((\texttt{"ab"}, \texttt{"AB"}), <Invert>)$.

To recover the initial input, we only have to apply the invert function to the pair of recognized values.

We should realize that the *invert* function remembers the order of the symbols but not the symbols themselves. As a result, we can change the initial values and still recover the initial order. Thus if we change the value 'B' in the previous result pair for a 'Z', for instance, the inverted input will have a 'Z' in the place of the 'B'.

Note that abstractly, this can be seen as a typical error correcting task, where the parser replaces a symbol, occurring in the wrong position, by the expected one.

3.2 Combining n parsers

In the previous section,we saw how to combine and invert two parsers. However, a *NOP* parser can be build up with a combination of any number of parsers. In this section, we define a parser combinator that may be used to combine any number of parsers into a *nop* parser.

When we combine two parsers, for instance *Parser a* and *Parser b*, the parsing result of the new parser is a pair of type (a, b). If we combine this new parser with another parser, for instance, *Parser c*, the parsing result, following the same logic, is of type (a, b, c). The problem of this approach is that the type of the resulting parser depends on the number of combined parsers. Our aim is to define a function that combines a variable number of parsers. Because the types of the combined parsers are typically not the same, we cannot represent the result by lists. A nested cartesian product is more suitable for this situation. We use combinators only in a left associative way, so all products are of the type $((x, y), z)$ instead of (x, y, z)[1].

In the previous section, both parser results were returned as lists. This is, however, a severe limitation. It would be convenient to abstract from the result of the parsers, in order to allow the use of data structures other than lists.

[1]This was also the very first approach taken in the design of parser combinator libraries [2]

In constructing lists we have used the predefine insert function on lists, and the (*const* [ ]) to construct the empty list. Thus in order to use a custom data structure we have to provide its constructors, i.e., the function that constructs the empty data structure and the respective insert function. We provide this information in a pair like (*empty*, *insert* <\$> *parser*). Now the parser result is a function that knows how to insert the recognized value in the data structure. The information pair can be represented by a type synonym:

$$\textbf{type}\ \mathit{ParseInf}\ \mathit{res} = (\mathit{res}, \mathit{Parser}\ (\mathit{res} \rightarrow \mathit{res}))$$

To return the parsing result in the predefined type *lists*, we have to build the information pair with the insert function (:) and with the empty structure ([ ]):

$$\begin{aligned}&\mathit{listOf} && :: \mathit{Parser}\ \mathit{res} \rightarrow \mathit{ParseInf}\ [\mathit{res}]\\ &\mathit{listOf}\ \mathit{parser} && = ([\,], (:) \mathbin{<\$>} \mathit{parser})\end{aligned}$$

Any data structure can be used. Thus, for instance, we can define a data type for balanced binary trees.

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf} \mid \mathit{Node}\ \mathit{Int}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)$$

Now we only need to compose the information pair, where the *Leaf* is the empty structure and the *insert* function is the balanced insert function for binary trees.

$$\begin{aligned}&\mathit{treeOf} && :: \mathit{Parser}\ \mathit{res} \rightarrow \mathit{ParseInf}\ (\mathit{Tree}\ \mathit{res})\\ &\mathit{treeOf}\ \mathit{parser} && = (\mathit{Leaf}, \mathit{insert} \mathbin{<\$>} \mathit{parser})\end{aligned}$$

Now instead of combining parsers, we have to combine information pairs that contain parsers. To be able to construct one parser that combines all the other parsers, we go through a two phase process. First we perform an abstract interpretation in which we combine all the information pairs into a single structure. Second we construct the parser we are interested in, using the result pair of the first phase.

To accomplish the first phase, we define a new combinator <||>, that combines the information pairs out of which the *NOP* parser is to be constructed. First we combine the empty structures in a pair. The result of applying the parser combinator <||> to a variable number of pairs, is a nested pair of empty structures. Next we have to compose the parser results. Following the approach in which we combined a parser with an insert function, the parser result becomes a function that knows how to insert the recognized value in the respective data structure. Thus we only apply the parsing result to the correct side of the pair.

$$\begin{aligned}&(\mathbin{<||>}) && :: \mathit{ParseInf}\ a \rightarrow \mathit{ParseInf}\ b \rightarrow \mathit{ParseInf}\ (a, b)\\ &(e_a, p_a) \mathbin{<||>} (e_b, p_b) && = ((e_a, e_b), \mathit{comb}_a \mathbin{<\$>} p_a \mathbin{<|>} \mathit{comb}_b \mathbin{<\$>} p_b)\\ &\quad \textbf{where}\\ &\quad\quad \mathit{comb}_a\ \mathit{insert}_a\ (\mathit{res}_a, \mathit{res}_b) = (\mathit{insert}_a\ \mathit{res}_a, \quad \mathit{res}_b)\\ &\quad\quad \mathit{comb}_b\ \mathit{insert}_b\ (\mathit{res}_a, \mathit{res}_b) = (\quad \mathit{res}_a, \mathit{insert}_b\ \mathit{res}_b)\end{aligned}$$

The result of combining all parsers is a single information pair, of which the first element is a nested pair with all the empty structures and the second element is an alternative combination of all the parsers. We can always combine the result constructed thus far with one more information pair.

Now we pass on to the second phase, where the result of the first phase is used to construct the final parser. We can use the parser combinator *pFoldr* to construct the parser we are interested in. The second element of the input pair, *alter* is a parser that returns a list of function. The combinator *pFoldr* will apply each function from this list backwards, starting with the empty structure, *units*.

$$\begin{aligned} &\mathit{pMerged} &&:: \mathit{ParseInf\ res} \rightarrow \mathit{Parser\ res} \\ &\mathit{pMerged}\ (\mathit{units}, \mathit{alter}) &&= \mathit{pFoldr\ apply\ units\ alter} \end{aligned}$$

Suppose we want to parse a string and to return the digits and upper case letters in two separate lists, and the lower case letters in a balanced binary tree. Then, instead of writing a new parser we can easily combine three predefined parsers. First, we use the functions *listOf* and *treeOf* to store in a pair the parser and the respective information about the data structure. Then we merge the three pairs in a single nested pair, using the new combinator $\Leftrightarrow$. Finally we construct the parser we are interested in using the *pMerged* function.

$$\begin{aligned} &\mathit{int\_low\_upp} :: \mathit{Parser}\ (([\mathit{Int}], [\mathit{Char}]), \mathit{Tree\ Char}) \\ &\mathit{int\_low\_upp} = \mathit{pMerged}\ (\mathit{listOf\ parser}_{\mathit{int}} \Leftrightarrow \\ &\qquad\qquad\qquad\qquad\quad \mathit{listOf\ parser}_{\mathit{low}} \Leftrightarrow \\ &\qquad\qquad\qquad\qquad\quad \mathit{treeOf\ parser}_{\mathit{upp}}) \end{aligned}$$

This new parser returns the parsing result in a nested pair, where the two first elements are lists of characters and the last one is a tree of characters. The result of applying the parser *int_low_upp* to the input string `"A1bC2"` is the nested pair $(([1, 2], \texttt{"b"}), \mathit{Node}\ 2\ (\mathit{Node}\ 1\ \mathit{Leaf}\ \texttt{'A'}\ \mathit{Leaf})\ \texttt{'C'}\ \mathit{Leaf})$.

3.3 Inverting n parsers

Having generalized the *pMerged* function to merge any number of parsers, we now will generalize the invert function as well. In the previous section, the semantic functions splitted the result into different data structures. To accomplish this, we needed to know the empty structure and the insert function of that data structure. Now, to be able to recover the initial order of the input, we also need to know how to get a value from that data structure. Thus, if for instance we are using *lists*, the function that takes one element from a non empty list is:

$$\begin{aligned} &\mathit{getVal} &&:: [v] \rightarrow (v, [v]) \\ &\mathit{getVal}\ (v : vs) &&= (v, vs) \end{aligned}$$

The new insert function has to return the updated data structure and the respective *getVal* function for that element. The new information pair has to be of the following shape $(\mathit{empty}, ((\lambda x\ xs \rightarrow (\mathit{inserted}, \mathit{getVal})) \mathbin{<\$} \mathit{parser}))$. The updated *ParseInf* type is:

$$\textbf{type}\ \mathit{ParseInf\ res\ ds\ inv} = (\mathit{res}, \mathit{Parser}\ (\mathit{res} \rightarrow (\mathit{res}, \mathit{ds} \rightarrow (\mathit{inv}, \mathit{ds}))))$$

To return the parsing result as *lists*, the information pair is:

$$\begin{aligned}
&\mathit{listOf} && :: \mathit{Parser\ res} \to \mathit{ParseInf}\ [\mathit{res}]\ [\mathit{inv}]\ \mathit{inv}\\
&\mathit{listOf\ parser} && = ([\,], (\lambda x\ l \to (x : l, \mathit{getVal})) \mathbin{<\$>} \mathit{parser})
\end{aligned}$$

As we can see, the function applied to the parsing result returns a pair with the updated *list* of results and the *getVal* function. In the case of *lists*, we always insert the new element in the *head*, thus is easy to define a function that takes the last inserted element. But in many other cases, like with balanced trees, taking an element from the data structure depends on the insert function. Thus the *getVal* function has to be defined during the insert action.

The *insert* function for balanced trees adds a new element to that side of the tree that has the least number of elements. Firstly we test which tree is the smallest one, then we recursively insert the new value in that tree. The result is a pair with the *newTree* and the *get* function. The updated *invert* function will apply the *get* function to the same side of the tree where we inserted the new value. Since the values are taken in the same order as inserted, the result tree will always be a balanced tree.

$$\begin{aligned}
&\mathit{insert} && :: a \to \mathit{Tree}\ a \to (\mathit{Tree}\ a, \mathit{Tree}\ b \to (b, \mathit{Tree}\ b))\\
&\mathit{insert\ val\ Leaf} && = (\mathit{Node}\ 1\ \mathit{Leaf\ val\ Leaf}, \mathit{invert}_{\mathit{leaf}})\\
&\mathit{insert\ val}\ (\mathit{Node}\ s\ \mathit{tree}_l\ \mathit{value}\ \mathit{tree}_r) && = (\mathit{Node}\ (s+1)\ l\ \mathit{value}\ r, i)\\
&\quad \mathbf{where}\ (l, r, i) \mid \mathit{size}\ \mathit{tree}_l \leqslant \mathit{size}\ \mathit{tree}_r = (\mathit{newTree}_l, \mathit{tree}_r, \mathit{invert}_l\ \mathit{get}_l)\\
&\qquad\qquad\qquad\quad \mid \mathit{otherwise} = (\mathit{tree}_l, \mathit{newTree}_r, \mathit{invert}_r\ \mathit{get}_r)\\
&\qquad\quad (\mathit{newTree}_l, \mathit{get}_l) = \mathit{insert\ val}\ \mathit{tree}_l\\
&\qquad\quad (\mathit{newTree}_r, \mathit{get}_r) = \mathit{insert\ val}\ \mathit{tree}_r
\end{aligned}$$

$$\begin{aligned}
&\mathit{invert}_{\mathit{leaf}}\ (\mathit{Node}\ 1\ \mathit{Leaf\ value\ Leaf}) = (\mathit{value}, \mathit{Leaf})\\
&\mathit{invert}_l\ \mathit{get}_l\ (\mathit{Node}\ s\ \mathit{tree}_l\ v\ \mathit{tree}_r) && = \mathbf{let}\ (\mathit{val}, \mathit{tree}) = \mathit{get}_l\ \mathit{tree}_l\\
& && \quad\ \mathbf{in}\ (\mathit{val}, \mathit{Node}\ (s-1)\ \mathit{tree}\ v\ \mathit{tree}_r)\\
&\mathit{invert}_r\ \mathit{get}_r\ (\mathit{Node}\ s\ \mathit{tree}_l\ v\ \mathit{tree}_r) && = \mathbf{let}\ (\mathit{val}, \mathit{tree}) = \mathit{get}_r\ \mathit{tree}_r\\
& && \quad\ \mathbf{in}\ (\mathit{val}, \mathit{Node}\ (s-1)\ \mathit{tree}_l\ v\ \mathit{tree})
\end{aligned}$$

Now the definition of the information pair is similar to the previous one:

$$\begin{aligned}
&\mathit{treeOf} && :: \mathit{Parser\ res} \to \mathit{ParseInf}\ (\mathit{Tree\ res})\ (\mathit{Tree\ inv})\ \mathit{inv}\\
&\mathit{treeOf\ parser} && = \ (\mathit{Leaf}, \mathit{insert} \mathbin{<\$>} \mathit{parser})
\end{aligned}$$

Like in the previous section, we proceed in two steps. The semantic functions comb_a and comb_b are extended, and thus the type of the combinator $\mathbin{<\cdot\!\!\cdot>}$ is adapted to the new *ParseInf* pair.

$$\begin{aligned}
&(\mathbin{<\cdot\!\!\cdot>}) && :: \mathit{ParseInf}\ \mathit{res}_a\ \mathit{ds}_a\ \mathit{inv}\\
& && \to \mathit{ParseInf}\ \mathit{res}_b\ \mathit{ds}_b\ \mathit{inv}\\
& && \to \mathit{ParseInf}\ (\mathit{res}_a, \mathit{res}_b)\ (\mathit{ds}_a, \mathit{ds}_b)\ \mathit{inv}\\
&(e_a, p_a) \mathbin{<\cdot\!\!\cdot>} (e_b, p_b) && = ((e_a, e_b), \mathit{comb}_a \mathbin{<\$>} p_a \mathbin{<|>} \mathit{comb}_b \mathbin{<\$>} p_b)
\end{aligned}$$

Both parsers, p_a and p_b, return a function. First, we apply that function to the correct side of the pair, that is to say, in comb_a we apply *insert* to values_a

and in $comb_b$ we apply it to $values_b$. This function returns a pair of which the first element is the data structure updated with the new parsed element and the second one the *get* function that removes the corresponding element from the data structure. Secondly we update the *invert* function by applying the *get* function to the correct side of the pair. The result is a pair with the recovered element and the non inverted elements represented by $(e, (rest, inp_b))$, in the case of $comb_a$ or $(e, (inp_a, rest))$ in the case of $comb_b$.

```
comb_a insert (values_a, values_b)     = ((newVal_a, values_b), invert_a)
      where
            (newVal_a, get_a)          = insert values_a
            invert_a (inp_a, inp_b)    = let (e, rest) = get_a inp_a
                                         in (e, (rest, inp_b))

comb_b insert (values_a, values_b)     = ((values_a, newVal_b), invert_b)
      where
            (newVal_b, get_b)          = insert values_b
            invert_b (inp_a, inp_b)    = let (e, rest) = get_b inp_b
                                         in (e, (inp_a, rest))
```

In the previous section, in order to obtain the combined parser, we only needed to use the parser combinator *pFoldr* with the function *apply*. This was possible because each parser result was a function that knew how to insert its result in the data structure.

Now we need to apply the result function to the data structure and update the invert function, as well. The *acceptNextVal* function applies the result function *insRes* to the data structure *recognVals*. The result is a pair with the updated data structure *newValues* and the function *getVal* . This function is used to get the first inverted value and subsequently we append it to the rest of the inverted values computed by *getRest*.

```
acceptNextVal :: (a → (a, c → (d, c))) → (a, c → [d]) → (a, c → [d])
acceptNextVal insert (recognVals, getRest) = (newValues, invert)
     where
          (newValues, getVal) = insert recognVals
          invert inp          = let (v, rest) = getVal inp
                                in v : (getRest rest)
```

The function *pFoldr* applies *acceptNextVal* to the result of the combined parsers, *alternatives*. When the parser fails or reaches the end of the input, *pFoldr* returns a pair with the empty structure and a function that always returns an empty list, the invert function.

```
pMerged                :: ParseInf a c d → Parser (a, c → [d])
pMerged (units, alter) = pFoldr acceptNextVal (units, const []) alter
```

The parser *dig_low_upp* is defined in a similar way as in the previous section, but now we get the invert function, as well.

$$\begin{aligned}
\mathit{dig\_low\_upp} &:: \mathit{Parser}\ ((([\mathit{Char}], [\mathit{Char}]), \mathit{Tree\ Char}),\\
&\qquad\qquad (([a], [a]), \mathit{Tree}\ a) \rightarrow [a])\\
\mathit{dig\_low\_upp} &= \ \mathit{pMerged}\ (\mathit{listOf}\ \mathit{parser}_{dig} \mathbin{<\!\!\!\diamond\!\!\!>}\\
&\qquad\qquad \mathit{listOf}\ \mathit{parser}_{low} \mathbin{<\!\!\!\diamond\!\!\!>}\\
&\qquad\qquad \mathit{treeOf}\ \mathit{parser}_{upp})
\end{aligned}$$

The usage of the parser is the same. Its result is a pair where the first element is a nested pair of parsing results and the second element the *invert* function. Thus applying the parser *dig_low_upp* to the input string `"A1bC2"` results in $((($`"12"`$,$ `"b"`$), \mathit{Node}\ 2\ (\mathit{Node}\ 1\ \mathit{Leaf}$ `'A'` $\mathit{Leaf})$ `'C'` $\mathit{Leaf}), <\mathit{invert}>)$.

The *invert* function is able to recover the initial order from the parsing result, by just applying it to the nested pair of recognized values. Thus, to recover the input string, we simply apply the *invert* function to the nested pair $(($`"12"`$,$ `"b"`$), \mathit{Node}\ 2\ (\mathit{Node}\ 1\ \mathit{Leaf}$ `'A'` $\mathit{Leaf})$ `'C'` $\mathit{Leaf})$.

4 Customizing the parser result

In the previous sections, we implemented a function, that combines a variable number of parsers. To achieve our aim we constructed the parsing result as a nested pair. Consequently the input of the invert function should be a nested pair, as well. This data type is useful to return a variable number of results, but it is difficult to understand by human beings. It would be convenient if the user could specify the type of the parsing result and the *invert* input. Therefore, we will now develop a variant in which the nested pairs are only used internally and thus eliminate the inconvenience of showing them to the outside world.

To be able to customize the parsing result, first we have to 'unpack' the nested pair, i.e., to take each element out of the nested pair. Next we have to transform the result to the custom one. Taking as example the previous combination of three parsers, we realize that the parsing result is a nested pair of three elements. To customize it we need a function that takes each different result and returns it in the new type. If for instance, we want to return this parsing result as a tuple, we have to supply the $(,,)$[2] function. This function takes three separate arguments and returns them in a tuple. In the approach followed thus far we actually need a function of the type $((a, b), c) \rightarrow (a, b, c)$. Haskell has defined the *uncurry*[3] function. It takes a function with two arguments and returns a function that takes a pair of arguments instead. If we apply the *uncurry* function to $(,,)$, we get:

$$\mathit{uncurry}\ (,,) :: (a, b) \rightarrow c \rightarrow (a, b, c)$$

Composing two *uncurry* function we construct the function we need:

$$(\mathit{uncurry}.\mathit{uncurry})\ (,,) :: ((a, b), c) \rightarrow (a, b, c)$$

[2] $(,,) :: a \rightarrow b \rightarrow c \rightarrow (a, b, c)$
[3] $\mathit{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c$

By applying two *uncurry* functions to the $(,,)$ function we manage to adapt it to handle nested pairs. The input of the *invert* function is a nested pair as well, $invert :: ((l, l), l) \to [l]$. We need a *pack* function that adapts it to take three inputs instead of a nested pair. The type of the invert function becomes $invert :: l \to l \to l \to [l]$. Haskell has defined the *curry*[4] function, as well. It takes a function with one pair of arguments and returns another function that takes two arguments. Like in the previous case if we compose two *curry* function, we are able to customize the invert function: The *curry* function transforms a function, that takes one pair of arguments, to another one that accepts flattened inputs.

$$(curry.curry)\ invert :: l \to l \to l \to [l]$$

The definition of both functions, *pack* and *unpack*, depends on the number of parsers we combine. Thus they have to be constructed during the merge process, and the *listOf* and the *treeOf* functions have to include a *pack* and an *unpack* function too. Since they store a single parser, the result does not need to be packed or unpacked, thus the *unpack* and *pack* function are defined as the *identity* function.

$$\begin{aligned} listOf\ parser &= ([\,], (\lambda x\ l \to (x : l, getVal)) \mathbin{<\$>} parser, id, id) \\ treeOf\ parser &= (Leaf, insert \mathbin{<\$>} parser, id, id) \end{aligned}$$

Whenever we combine two parsers, we have to compose the *pack* function with another *curry* function. We do the same thing to the *unpack* function, but instead of using the *curry* function we use the *uncurry*. Initially both *pack* and *unpack* functions are the *identity* function. After combining the two first parsers, the resulting *pack* and *unpack* are equal to *curry* and *uncurry* functions respectively, as we saw before. In the end we will have $(n-1)$ *curry* and *uncurry* composed functions, where n is the number of parsers combined.

$$\begin{aligned} &(e_a, p_a, unpack_a, pack_a) \mathbin{<+>} (e_b, p_b, unpack_b, pack_b) = \\ &\quad ((e_a, e_b) \\ &\quad , comb_a \mathbin{<\$>} p_a \mathbin{<|>} comb_b \mathbin{<\$>} p_b \\ &\quad , unpack_b.uncurry.unpack_a \\ &\quad , pack_b.curry.pack_a \\ &\quad) \end{aligned}$$

Now we eliminate the nested pair of the input of the *invert* function by applying the *pack* function to it. The semantic function *sem* that returns the parsing result in a predefined data type is adapted to the nested pair by applying the *unpack* function to it.

$$\begin{aligned} &sem\ `pMerged`\ (units, alternatives, unpack, pack) = \\ &\qquad (\lambda(result, invert\ ) \to (unpack\ sem\ result, pack\ invert)) \\ &\qquad \mathbin{<\$>} \\ &\qquad pFoldr\ acceptNextVal\ (units, const\ [\,])\ alternatives \end{aligned}$$

[4] $curry :: ((a, b) \to c) \to a \to b \to c$

If for instance, we want to change the previous example, to return the parser result in a tuple, instead of a nested pair, the definition of *dig_low_upp* will be:

$$\begin{aligned} dig\_low\_upp = (,,)\ `pMerged`\ (&listOf\ parser_{dig} \Leftrightarrow \\ &listOf\ parser_{low} \Leftrightarrow \\ &listOf\ parser_{upp}) \end{aligned}$$

The parser usage remains the same, but the result is a customized data structure. The *invert* function takes as many arguments as we have composed parsers.

5 Inverting parsers of different types

As we realized in the previous section, the type of the invert function for three combined parsers is:

$$invert :: l \to l \to l \to [\,l\,]$$

The three inputs have to be of the same type since they will be inserted in a list. However, parsers of different types can be combined, for instance the parser *dig_alp*, defined in the section 3.1. Then the *invert* function has to deal with inputs of different types. There are two ways to deal with this drawback. The most obvious one, is to return the inverted values in a data structure where the elements can be of different types, instead of a list. To achieve this solution, we need to replace the list insert function (:), used in the function *acceptNextVal*. We also have to specify the empty structure that *pFoldr* should return when the parser fails or reaches the end of input.

If we decide to return the inverted elements in a list, the user will have to supply functions, that homogenize all the parser results, in other words, all the inputs have to be transformed in values of the same type. This functions will be called *transformation* functions. The new *invert* function takes n *transformation* functions and n parser results. Each function homogenizes the respective parser result.

$$invert\ f_1 \ldots f_n\ res_1 \ldots res_n$$

Whenever we recover another element we have to homogenize it using the *transformation* function. All the *transformation* functions are stored in a nested pair. The function we should use depends on the parser that we used to recognize that element. We will encapsulate the type of each inverted value using a new type definition. In this data type, we will store two functions. The first function, $funcs \to (inv \to res)$, knows how to choose the right *transformation function*, from a nested pair of functions. The second function, $ds \to (inv, ds)$, is the function that takes the next inverted value from the parsing result.

$$\begin{aligned} &\textbf{data}\ Pair\ funcs\ res\ val = \\ &\qquad \exists\ inv.\ Pair\ (funcs \to (inv \to res))\ (val \to (inv, val)) \end{aligned}$$

With the second function $val \to (inv, val)$ we get an inverted value of type inv. Then with the first function $funcs \to (inv \to res)$, we get a function that transform the inverted value in a value of type *res*. The $\exists$ states that the inverted value, *inv*, can be of any type if we also provide a function that can transform that value to a value of type *res*. Using this data type we make the type of the inverted value transparent.

The new *invert* function takes n *transformation* functions and n parser results. Now we have to merge the *transformation* functions in a nested pair, as we do to the input of the *invert* function. Thus we need two pack functions.

As a practical example, we will redefine the *listOf* and *treeOf* functions. Now, instead of storing an invert function, we have to store a *Pair* with the function that is able to choose the *transformation* function for the respective inverted value, and the *invert* function itself.

```
listOf parser      = ([], ((λx l → (x : l, Pair id getVal)) <$> parser), id, id, id)

treeOf parser      = (Leaf, f <$> parser, id, id, id)
       where f x t = let (res, getVal) = insert x t
                     in (res, Pair id getVal)
```

The new definition of the combinator <+> is extended with an extra pack function. This *pack* function will be used to pack the *transformation* functions, before being used by the invert function.

As you can realize, the two *pack* functions are similar. It will be convenient to have a single polymorphic function. That is possible, but it is out of scope of this paper:

```
(e_a, p_a, punpack, ppack, ppack') <+> (e_b, p_b, qunpack, qpack, qpack') =
    ((e_a, e_b)
    , comb_a <$> p_a <|> comb_b <$> p_b
    , qunpack.uncurry.punpack
    , qpack.curry.ppack
    , qpack'.curry.ppack'
    )
```

The combined functions are adapted to the new type *Pair*. In $comb_a$ first we apply *insert* to the left side of the pair. Then we update the *choose* function by applying the *ch* function to the left side of the pair of the *transformation* functions. The invert function takes a value from the left side, exactly how we did it before. The $comb_b$, has the same behavior but applies both functions to the right side of the pair.

```
comb_a insert (values_a, values_b)    = ((newVal_a, values_b), Pair choose invert_a)
  where
     (newVal_a, Pair ch getVal) = insert values_a
     choose (func_a, func_b)    = ch func_a
     invert_a (inp_a, inp_b)    = let (e, rest) = getVal inp_a
                                  in (e, (rest, inp_b))
comb_b insert (values_a, values_b)    = ((values_a, newVal_b), Pair choose invert_b)
  where
     (newVal_b, Pair ch getVal) = insert values_b
     choose (func_a, func_b)    = ch func_b
     invert_b (inp_a, inp_b)    = let (e, rest) = getVal inp_b
                                  in (e, (inp_a, rest))
```

The *acceptNextVal* function uses the function *choose* to take the correct *transformation* function from the nested pair *funcs*, where all the *transformation* functions are stored. Then the chosen function is applied to the inverted value e and the result is inserted in the list of inverted values.

```
acceptNextVal f (recognVals, getRest)         = (newValues, invert)
     where
        (newValues, Pair choose getVal) = f recognVals
        invert funcs inp = let (e, rest)   = getVal inp
                               in (choose funcs e) : (getRest funcs rest)
```

The *acceptNextVal* is used by *pMerged* to compose the parsing results. By applying the two *pack* functions to *invert* we obtain an invert function that takes several inputs instead of two nested pairs.

```
sem `pMerged` (units, alternatives, unpack, pack, pack') =
     (λ(r, invert) → (unpack sem r, pack' (λval → pack (invert val))))
     <$>
     pFoldr acceptNextVal (units, const (const [])) alternatives
```

Now we can combine parsers of different types. We can define a parser, that includes, for instance, the parser $parser_{int}$ that recognizes and returns digits, as well as, the parser $parser_{low}$ returns strings and the parser $parser_{upp}$ that returns a balanced tree of characters.

```
int_low_upp = (,,) `pMerged` (listOf parser_int <+>
                              listOf parser_low <+>
                              treeOf parser_upp)
```

The usage of the parser remains the same, but the result is a list of digits, a string and a tree of characters. To recover the order of the initial input, we supply to the invert function the functions that homogenize the input, for instance, the function *show*.

6 Example: Parsing Java syntax

Modern languages are liberal about the order of declarations. Taking as example a small subset of the java language specification, each **class** can be composed by fields, methods and constructors. These components can appear in any order.

We start by defining the abstract syntax for Java classes:

```
data Class = Class String Elements
data Elements = Elements [Field] [Constructor] [Method]
```

Assume we have three parsers *pField*, *pConstructor* and *pMethod*, that parse fields, constructors and methods, respectively.

Then we use the *pMerged* function to build the final parser *pClass*. Each Java field, constructor and method is stored in a list, using the function *listOf*.

```
pClass = Class <$  pToken "class" <*> identifier <*>
         braces (Elements `pMerged` (
                                  listOf pField <||>
                                  listOf pConstructor <||>
                                  listOf pMethod
                                  ))

braces p = (\_ v _ -> v) <$> pToken "{" <*> p <*> pToken "}"
```

To recover the initial Java code we only need to define a pretty-printing for class and use the resulting *invert* function. This function will return the input in its original order.

```
ppClass invert (Class name (Elements fs cs ms)) =
      "class " ++ name ++
      ppBraces (invert ppField ppConstructor ppMethod fs cs ms)
```

7 Conclusion

We have shown how to define a *NOP* parser by combining several parsers. We extended the parser combinator with the inverse functionality.

We tried to make the interface of the combinators and the invert function as simple as possible. A user can easily make use of this combinator to obtain a *NOP* parser and the invert function, with the advantage of having the parsing result in a customized data type.

We have shown how to use existentially quantified data types to customize the type of the parser result. The invert function can be used to generate properly located error messages or even to repair errors found at parsing time.

The functions presented in this project have been incorporated into the "UU_Parsing" library. This library is used by a large community of functional programmers worldwide. Thus the results of this project will also have an impact in that community and we do hope to contribute to make programming and writing parsers a more effective, easy and grateful task.

References

[1] A.I. Baars, A. Löh, and S.D. Swierstra. Parsing permutation phrases. In R. Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 171–182. Elsevier, 2001.

[2] J. Fokker. Functional parsers. In J.T. Jeuring and H.J.M. Meijer, editors, *Advanced Functional Programming*, number 925 in LNCS, pages 1–52, 1995.

[3] Simon Peyton-Jones, John (eds.) Hughes, et al. Report on the programming language Haskell 98. February 1998.

[4] Doaitse Swierstra and Luc Duponcheel. Deterministic, error correcting combinator parsers. In *Advanced Functional Programming, Second International Spring School*, volume 1129 of *LNCS*, pages 184–207. Springer-Verlag, 1996.

[5] Philip Wadler. How to replace failure with a list of successes. In *Functional Programming Languages and Computer Architecture*, volume 201 of *LNCS*, pages 113–128. Springer-Verlag, 1985.

A Parsing a permutation phrase with inverse functionality

In [1] is shown how to represent a permutation phrase in a tree by expanding and factorizing its elements. 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.

The data type *Perms* represents our tree. 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.

```
data Perms a  = Choice{ defaults :: (Maybe a), branches :: [Branch a] }
data Branch a = ∀ x.Br (Parser (x, GetVal)) (Perms (x → a))
```

The first parser added to the tree, will have an invert function that takes the first element of the input list. The *n* parser, where *n* denotes the number of parsers added to the tree plus one, will take the element in the position number *n* from the input list.

```
data GetVal = GetVal (∀ a.[a] → a)
```

The data type *GetVal* keeps record of the function that takes the *n* element of a list.

```
data Undo a = Undo GetVal (Perms a)
```

By keeping this information outside the tree, we avoid to replicate it through all the branches.

All the parsers of the permutation phrase has to be added to the tree. Each path of the tree represents a parser of a possible permutation. The function *permute* follows the correct path to obtain the pretended parser. The new parser returns the parsing result and the *invert* function.

```
permute                             :: Undo a → Parser (a, [b] → ([b], [b]))
permute (Undo g (Choice def bs))    =  foldr (<|>) exit (map pars bs)
  where exit = case def of
                 Nothing → fail
                 Just x → succeed (x, λinp → (inp, []))
        pars (Br p perm) = (λ(x, GetVal u) (f, l) → (f x, (nextVal u).l))<$>
                                   p <*> permute (Undo g perm)
```

```
nextVal getVal (vals, inverted) = (vals, (getVal vals) : inverted)
```

The empty tree stores the initial *invert* function, (*GetVal head*).

```
empty   :: a → Undo a
empty x = Undo (GetVal head) (Choice (Just x) [])
```

The *add* function takes a pair of arguments, where the first value specify if the element is optional or not. Besides that, it has to update the *invert* function. The new *invert* function will take the next element of the list, by composing the old *invert* function with the *tail* function.

```
add :: (Maybe a, Parser a) → Undo (a → b) → Undo b
add (def, p) (Undo (GetVal undo) perm) =
        Undo (GetVal (undo.tail))
                (addPerms (def, ((λx → (x, (GetVal undo))) <$> p)) perm)
```

The *invert* function is combined with the parser and added to the permutation tree using the function *addPerms*. A new element is added to the permutation tree by inserting it in all possible positions to every permutation that is already in the tree.

```
addPerms :: (Maybe a, Parser ( a, GetVal)) → Perms (a → b) → Perms b
addPerms (d, p) perm@(Choice ds bs) =
      Choice (ds ‘ap‘ d) (Br p perm : map ins bs)
   where
      ins (Br p′ perm′) = Br p′ (addPerms (d, p) (mapPerms flip perm′))
```

The *mapPerms* is the map on permutation trees.

```
mapPerms                   :: (a → b) → Perms a → Perms b
mapPerms f (Choice d bs) = Choice (fmap f d) (map (mapBranch f) bs)

mapBranch                  :: (a → b) → Branch a → Branch b
mapBranch f (Br p perm)  = Br p (mapPerms (f.) perm)
```

This is an example of a parser for a permutation of an optional character ’a’ and ’c’ and zero or more characters ’b’.

```
perm_abc :: Parser ((Char, [Char], Char), [b] → ([b], [b]))
perm_abc = permute ((, , )<<$>> option('_', (symbol 'a'))
                              <|> option([], many (symbol 'b'))
                              <|> option('_', (symbol 'c')))
```