Combinator Parsing: A Short Tutorial

S. Doaitse Swierstra

Center for Software Technology, Utrecht University, The Netherlands,
doaitse@swierstra.net

Abstract. There are numerous ways to implement a parser for a given syntax; using parser combinators is a powerful approach to parsing which derives much of its power and expressiveness from the type system and semantics of the host programming language. This tutorial begins with the construction of a small library of parsing combinators. This library introduces the basics of combinator parsing and, more generally, demonstrates how domain specific embedded languages are able to leverage the facilities of the host language. After having constructed our small combinator library, we investigate some shortcomings of the naïve implementation introduced in the first part, and incrementally develop an implementation without these problems. Finally we discuss some further extensions of the presented library and compare our approach with similar libraries.

1 Introduction

Parser combinators [2, 4, 8, 15] occupy a unique place in the field of parsing; they make its possible to write expressions which look like grammars, but actually describe parsers for these grammars. Most mature parsing frameworks entail voluminous preprocessing, which read in the syntax at hand, analyse it, and produce target code for the input grammar. By contrast, a relatively small parser combinator library can achieve comparable parsing power by harnessing the facilities of the language. In this tutorial we develop a mature parser combinator library, which rivals the power and expressivity of other frameworks in only a few hundred lines of code. Furthermore it is easily extended if desired to do so. These advantages follow from the fact that we have chosen to embed context-free grammar notation into a general purpose programming language, by taking the *Embedded Domain Specific Language* (EDSL) approach.

For many areas special purpose programming languages have been defined. The implementation of such a language can proceed along several different lines. On the one hand one can construct a completely new compiler, thus having complete freedom in the choice of syntax, scope rules, type system, commenting conventions, and code generation techniques. On the other hand one can try to build on top of work already done by extending an existing host language. Again, here one can pursue several routes; one may extend an existing compiler, or one can build a library implementing the new concepts. In the latter case one automatically inherits –but is also limited to– the syntax, type system and code generation techniques from the existing language and compilers for that language. The success of this technique thus depends critically on the properties of the host language.

With the advent of modern functional languages like Haskell [11] this approach has become a really feasible one. By applying the approach to build a combinator parsing library we show how Haskell's type system (with the associated class system) makes this language an ideal platform for describing EDSLs . Besides being a kind of user manual for the constructed library this tutorial also serves as an example of how to use Haskell concepts in developing such a library. Lazy evaluation plays a very important role in the semantics of the constructed parsers; thus, for those interested in a better understanding of lazy evaluation, the tutorial also provides many examples. A major concern with respect to combinator parsing is the ability or need to properly define and use parser combinators so that functional values (trees, unconsumed tokens, continuations, etc.) are correctly and efficiently manipulated.

In Sect. 2 we develop, starting from a set of basic combinators, a parser combinator library, the expressive power of which extends well above what is commonly found in *EBNF*-like formalisms. In Sect. 3 we present a case study, describing a sequence of ever more capable pocket calculators. Each subsequent version gives us a chance to introduce some further combinators with an example of their use.

Sect. 4 starts with a discussion of the shortcomings of the naïve implementation which was introduced in Sect.2, and we present solutions for all the identified problems, while constructing an alternative, albeit much more complicated library. One of the remarkable points here is that the basic interface which was introduced in Sect. 2 does not have to change, and that –thanks to the facilities provided by the Haskell class system– all our derived combinators can be used without having to be modified.

In Section 5 we investigate how we can use the progress information, which we introduced to keep track of the progress of parsing process, introduced in Sect. 4 to control the parsing process and how to deal with ambiguous grammars. In Sect. 6 we show how to use the Haskell type and class system to combine parsers which use different scanner and symbol type intertwined. In Sect. 7 we extend our combinators with error reporting properties and the possibility to continue with the parsing process in case of erroneous input. In Sect. 8 we introduce a class and a set of instances which enables us to make our expressions denoting parsers resemble the corresponding grammars even more. In Sect. 9 we touch upon some important extensions to our system which are too large to deal with in more detail, in Sect. 10 we provide a short comparison with other similar libraries and conclude.

2 A Basic Combinator Parser Library

In this section we describe how to embed grammatical descriptions into the programming language Haskellin such a way that the expressions we write closely resemble context- free grammars, but actually are descriptions of parsers for such languages. This technique has a long history, which can be traced back to the use of recursive descent parsers [2], which became popular because of their ease of use, their nice integration with semantic processing, and the absence of the need to (write and) use an off-line parser generator. We assume that the reader has a basic understanding in the concept of a context-free grammar, and probably also has seen the use of parser generators, such as YACC or ANTLR.

Just like most normal programming languages, embedded domain specific languages are composed of two things:

1. a collection of primitive constructs
2. ways to compose and name constructs

and when embedding grammars things are no different. The basic grammatical concepts are *terminal* and *non-terminal* symbols, or *terminals* and *non-terminals* for short. They can be combined by *sequential composition* (multiple constructs occurring one after another) or by *alternative composition* (a choice from multiple alternatives).

Note that one may argue that non-terminals are actually not primitive, but result from the introduction of a naming scheme; we will see that in the case of parser combinators, non-terminals are not introduced as a separate concept, but just are Haskell names referring to values which represent parsers.

2.1 The Types

Since grammatical expressions will turn out to be normal Haskell expressions, we start by discussing the types involved; and not just the types of the basic constructs, but also the types of the composition mechanisms. For most embedded languages the decisions taken here heavily influence the shape of the library to be defined, its extendability and eventually its success.

Basically, a parser takes a list of symbols and produces a tree. Introducing type variables to abstract from the symbol type s and the tree type t, a first approximation of our *Parser* type is:

$$\textbf{type}\ \mathit{Parser}\ s\ t = [s] \rightarrow t$$

Parsers do not need to consume the entire input list. Thus, apart from the tree, they also need to return the part of the input string that was not consumed:

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

The symbol list $[s]$ can be thought of as a *state* that is transformed by the function while building the tree result.

Parsers can be ambiguous: there may be multiple ways to parse a string. Instead of a single result, we therefore have a list of possible results, each consisting of a parser tree and unconsumed input:

```
type Parser s t = [s] → [(t, [s])]
```

This idea was dubbed by Wadler [17] as the "list of successes" method, and it underlies many backtracking applications. An added benefit is that a parser can return the empty list to indicate failure (no successes). If there is exactly one solution, the parser returns a singleton list.

Wrapping the type with a constructor P in a **newtype** definition we get the actual *Parser* type that we will use in the following sections:

```
newtype Parser s t = P ([s] → [(t, [s])])
unP (P p) = p
```

2.2 Basic Combinators: *pSym*, *pReturn* and *pFail*

As an example of the use of the *Parser* type we start by writing a function which recognises the letter `'a'`: keeping the "list of successes" type in mind we realise that either the input starts with an `'a'` character, in which case we have precisely one way to succeed, i.e. by removing this letter from the input, and returning this character as the witness of success paired with the unused part of the input. If the input does not start with an `'a'` (or is empty) we fail, and return the empty list, as an indication that there is no way to proceed from here:

```
pLettera :: Parser Char Char
pLettera = P (λinp → case inp of
                      (s : ss) | s ≡ 'a' → [('a', ss)]
                      otherwise →          []
              )
```

Of course, we want to abstract from this as soon as possible; we want to be able to recognise other characters than `'a'`, and we want to recognise symbols of other types than *Char*. We introduce our first basic parser constructing function *pSym*:

```
pSym ::    Eq s ⇒ s → Parser s s
pSym a = P (λinp → case inp of
                    (s : ss) | x ≡ a → [(s, ss)]
                    otherwise        → []
            )
```

Since we want to inspect elements from the input with terminal symbols of type s, we have added the *Eq* s constraint, which gives us access to equality (≡) for values of type s. Note that the function *pSym* by itself is strictly speaking not a parser, but a function which returns a parser. Since the argument is a run-time value it thus becomes possible to construct parsers at run-time.

One might wonder why we have incorporated the value of s in the result, and not the value a? The answer lies in the use of *Eq s* in the type of *Parser*; one should keep in mind that when $\equiv$ returns *True* this does not imply that the compared values are guaranteed to be bit-wise equal. Indeed, it is very common for a scanner –which pre-processes a list of characters into a list of tokens to be recognised– to merge different tokens into a single class with an extra attribute indicating which original token was found. Consider e.g. the following *Token* type:

```
data Token = Identifier      -- terminal symbol used in parser
           | Ident String    -- token constructed by scanner
           | Number Int
           | If_Symbol
           | Then_Symbol
```

Here, the first alternative corresponds to the terminal symbol as we find it in our grammar: we want to see an identifier and from the grammar point of view we do not care which one. The second alternative is the token which is returned by the scanner, and which contains extra information about which identifier was actually scanned; this is the value we want to use in further semantic processing, so this is the value we return as witness from the parser. That these symbols are the same, as far as parsing is concerned, is expressed by the following line in the definition of the function $\equiv$:

```
instance Eq Token where
  (Ident _) ≡ Identifier = True
  ...
```

If we now define:

```
pIdent = pSym Identifier
```

we have added a special kind of terminal symbol.

The second basic combinator we introduce in this subsection is *pReturn*, which corresponds to the ϵ-production. The function always succeeds and as a witness returns its parameter; as we will see the function will come in handy when building composite witnesses out of more basic ones. The name was chosen to resemble the monadic *return* function, which injects values into the monadic computation:

```
pReturn :: a → Parser s a
pReturn a = P (λinp → [(a, inp)])
```

We could have chosen to let this function always return a specific value (e.g. ()), but as it will turn out the given definition provides a handy way to inject values into the result of the overall parsing process.

The final basic parser we introduce is the one which always fails:

$$pFail = P\ (const\ [\,])$$

One might wonder why one would need such a parser, but that will become clear in the next section, when we introduce *pChoice*.

2.3 Combining Parsers: <*>, <|>, <\$> and *pChoice*.

A grammar production usually consists of a sequence of terminal and non-terminal symbols, and a first choice might be to use values of type $[Parser\ s\ a]$ to represent such productions. Since we usually will associate different types to different parsers, this does not work out. Hence we start out by restricting ourselves to productions of length 2 and introduce a special operator <*> which combines two parsers into a new one. What type should we choose for this operator? An obvious choice might be the type:

$$Parser\ s\ a \rightarrow Parser\ s\ b \rightarrow Parser\ s\ (a, b)$$

in which the witness type of the sequential composition is a pair of the witnesses for the elements of the composition. This approach was taken in early libraries [4]. A problem with this choice is that when combining the resulting parser with further parsers, we end up with a deeply nested binary Cartesian product. Instead of starting out with simple types for parsers, and ending up with complicated types for the composed parsers, we have taken the opposite route: we start out with a complicated type and end with a simple type. This interface was pioneered by Röjemo [12], made popular through the library described by Swierstra and Duponcheel [15], and has been incorporated into the Haskell libraries by McBride and Paterson [10]. Now it is know as the *applicative interface.* It is based on the idea that if we have a value of a complicated type $b \rightarrow a$, and a value of type b, *we can compose them into a simpler type by applying the first value to the second one.* Using this insight we can now give the type of <*>, together with its definition:

$$\begin{array}{lll}
(\circledast) & :: & Parser\ s\ (b \rightarrow a) \rightarrow Parser\ s\ b \rightarrow Parser\ s\ a \\
P\ p_1 \circledast P\ p_2 & = & P\ (\lambda inp \rightarrow [(v_1\ v_2, ss_2) \mid (v_1, ss_1) \leftarrow p_1\ inp \\
 & & \qquad\qquad\qquad\qquad\qquad\quad , (v_2, ss_2) \leftarrow p_2\ ss_1 \\
 & & \qquad\qquad\quad] \\
 & & \quad)
\end{array}$$

The resulting function returns all possible values $v_1\ v_2$ with remaining state ss_2, where v_1 is a witness value returned by parser p_1 with remaining state ss_1. The state ss_1 is used as the starting state for the parser p_2, which in its turn returns the witnesses v_2 and the corresponding final states ss_2. Note how the types of the parsers were chosen in such a way that the value of type $v_1\ v_2$ matches the witness type of the composed parser.

As a very simple example, we give a parser which recognises the letter 'a' twice, and if it succeeds returns the string `"aa"`:

```
pString_aa = (pReturn (:) <*> pLettera)
                  <*>
             (pReturn (λx → [x]) <*> pLettera)
```

Let us take a look at the types. The type of (:) is $a \to [a] \to [a]$, and hence the type of *pReturn* (:) is $Parser\ s\ (a \to [a] \to [a])$. Since the type of *pLettera* is $Parser\ Char\ Char$, the type of *pReturn* (:)<*>*pLettera* is $Parser\ Char\ ([Char] \to [Char])$. Similarly the type of the right hand side operand is $Parser\ Char\ [Char]$, and hence the type of the complete expression is $Parser\ Char\ [Char]$. Having chosen <*> to be left associative, the first pair of parentheses may be left out. Thus, many of our parsers will start out by producing some function, followed by a sequence of parsers each providing an argument to this function.

Besides sequential composition we also need *choice*. Since we are using lists to return all possible ways to succeed, we can directly define the operator <|> by returning the concatenation of all possible ways in which either of its arguments can succeed:

```
(<|>)          :: Parser s a → Parser s a → Parser s a
P p₁ <|> P p₂ = P (λinp → p₁ inp ++ p₂ inp)
```

Now we have seen the definition of <|>, note that *pFail* is both a left and a right unit for this operator:

```
pFail <|> p ≡ p ≡ p <|> pFail
```

which will play a role in expressions like

```
pChoice ps = foldr (<|>) pFail ps
```

One of the things left open thus far is what precedence level these newly introduced operators should have. It turns out that the following minimises the number of parentheses:

```
infixl 5 <*>
infixr 3 <|>
```

As an example to see how this all fits together, we write a function which recognises all correctly nested parentheses – such as `"()(()())"`– and returns the maximal nesting depth occurring in the sequence. The language is described by the grammar $S \to$ `'('` S `')'` $S \mid \epsilon$, and its transcription into parser combinators reads:

```
parens :: Parser Char Int
parens = pReturn (λ_ b _ d → (1 + b) `max` d)
                 <*> pSym '(' <*> parens <*> pSym ')' <*> parens
         <|> pReturn 0
```

Since the pattern *pReturn* ... <*> will occur quite often, we introduce a third combinator, to be defined in terms of the combinators we have seen already. The combinator <\$> takes a function of type $b \to a$, and a parser of type $Parser\ s\ b$, and builds a value of type *Parser s a*, by applying the function to the witness returned by the parser. Its definition is:

```
infix 7 <$>
(<$>) :: (b → a) → (Parser s b) → Parser s a
f <$> p = pReturn f <*> p
```

Using this new combinator we can rewrite the above example into:

```
parens = (λ_ b _ d → (1 + b) `max` d)
                <$> pSym '(' <*> parens <*> pSym ')' <*> parens
           <|> pReturn 0
```

Notice that left argument of the <$> occurrence has type $a \to (Int \to (b \to (Int \to Int)))$, which is a function taking the four results returned by the parsers to the right of the <$> and constructs the result sought; this all works because we have defined <*> to associate to the left.

Although we said we would restrict ourselves to productions of length 2, in fact we can just write productions containing an arbitrary number of elements. Each extra occurrence of the <*> operator introduces an anonymous non-terminal, which is used only once.

Before going into the development of our library, there is one nasty point to be dealt with. For the grammar above, we could have just as well chosen $S \to S$ `'('` S `')'` $\mid \epsilon$, but unfortunately the direct transcription into a parser would not work. Why not? Because the resulting parser is left recursive: the parser *parens* will start by calling itself, and this will lead to a non-terminating parsing process. Despite the elegance of the parsers introduced thus far, this is a serious shortcoming of the approach taken. Often, one has to change the grammar considerably to get rid of the left-recursion. Also, one might write left-recursive grammars without being aware of it, and it will take time to debug the constructed parser. Since we do not have an off-line grammar analysis, extra care has to be taken by the programmer since the system just does not work as expected, without giving a proper warning; it may just fail to produce a result at all, or it may terminate prematurely with a stack-overflow.

2.4 Special Versions of Basic Combinators: <*, *>, <$ and *opt*.

As we see in the *parens* program the values witnessing the recognition of the parentheses themselves are not used in the computation of the result. As this situation is very common we introduce specialised versions of <$> and <*>: in the new operators <$, <* and *>, the missing bracket indicates which witness value is not to be included in the result:

```
infixl 3  `opt`
infixl 5  <*, *>
infixl 7  <$
```

```
f <$ p  = const <$> pReturn f <*> p
p <* q  = const <$> p         <*> q
p *> q  = id    <$  p         <*> q
```

We use this opportunity to introduce two further useful functions, *opt* and *pParens* and reformulate the *parens* function:

```
pParens :: Parser s a → Parser s a
pParens p = id <$ pSym '(' <*> p <* pSym ')'

opt :: Parser s a → a → Parser s a
p `opt` v    = p <|> pReturn v

parens       = (max.(1+)) <$> pParens parens <*> parens `opt` 0
```

As a final combinator, which we will use in the next section, we introduce a combinator which creates the parser for a specific keyword given as its parameter:

```
pSyms [ ] = pReturn [ ]
pSyms (x : xs) = (:) <$> pSym x <*> pSyms xs
```

3 Case Study: Pocket Calculators

In this section, we develop –starting from the basic combinators introduced in the previous section– a series of pocket calculators, each being an extension of its predecessor. In doing so we gradually build up a small collection of useful combinators, which extend the basic library.

To be able to run all the different versions we provide a small driver function *run* :: (*Show t*) ⇒ *Parser Char t* → *String* → *IO* () in appendix A. The first argument of the function *run* is the actual pocket calculator to be used, whereas the second argument is a string prompting the user with the kind of expressions that can be handled. Furthermore we perform a little bit of preprocessing by removing all spaces occurring in the input.

3.1 Recognising a Digit

Our first calculator is extremely simple; it requires a digit as input and returns this digit. As a generalisation of the combinator *pSym* we introduce the combinator *pSatisfy*: it checks whether the current input token satisfies a specific predicate instead of comparing it with the expected symbol:

```
pDigit = pSatisfy (λx → '0' ⩽ x ∧ x ⩽ '9')
pSatisfy :: (s → Bool) → Parser s s
pSatisfy p = P (λinp → case inp of
                         (x : xs) | p x → [(x, xs)]
                         otherwise      → [ ]
               )
pSym a = pSatisfy (≡ a)
```

A demo run now reads:

```
*Calcs> run pDigit "5"
Give an expression like: 5 or (q) to quit
3
Result is: '3'
Give an expression like: 5 or (q) to quit
a
Incorrect input
Give an expression like: 5 or (q) to quit
q
*Calcs>
```

In the next version we slightly change the type of the parser such that it returns an *Int* instead of a *Char*, using the combinator <$>:

```
pDigitAsInt :: Parser Char Int
pDigitAsInt = (λc → fromEnum c − fromEnum '0') <$> pDigit
```

3.2 Integers: *pMany* and *pMany1*

Since single digits are very boring, let's change our parser into one which recognises a natural number, i.e. a (non-empty) sequence of digits. For this purpose we introduce two new combinators, both converting a parser for an element to a parser for a sequence of such elements. The first one also accepts the empty sequence, whereas the second one requires at least one element to be present:

```
pMany, pMany1 :: Parser s a → Parser s [a]
pMany p   = (:) <$> p <*> pMany p `opt` []
pMany1 p  = (:) <$> p <*> pMany p
```

The second combinator forms the basis for our natural number recognition process, in which we store the recognised digits in a list, before converting this list into the *Int* value:

```
pNatural :: Parser Char Int
pNatural = foldl (λa b → a ∗ 10 + b) 0 <$> pMany1 pDigitAsInt
```

From here it is only a small step to recognising signed numbers. A − sign in front of the digits is mapped onto the function *negate*, and if it is absent we use the function *id*:

```
pInteger :: Parser Char Int
pInteger = (negate <$ (pSyms "-") `opt` id) <*> pNatural
```

3.3 More Sequencing: *pChainL*

In our next version, we will show how to parse expressions with infix operators of various precedence levels and various association directions. We start by parsing an expression containing a single + operator, e.g. "2+55". Note again that the result of recognising the + token is discarded, and the operator (+) is only applied to the two recognised integers:

```
pPlus = (+) <$> pInteger <* pSyms "+" <*> pInteger
```

We extend this parser to a parser which handles any number of operands separated by +-tokens. It demonstrates how we can make the result of a parser to differ completely from its “natural” abstract syntax tree.

```
pPlus′ = applyAll <$> pInteger <*> pMany ((+) <$ pSyms "+" <*> pInteger)
applyAll :: a → [a → a] → a
applyAll x (f : fs)  =  applyAll (f x) fs
applyAll x [ ]       =  x
```

Unfortunately, this approach is a bit too simple, since we are relying on the commutativity of + for this approach to work, as each integer recognized in the call to *pMany* becomes the first argument of the (+) operator. If we want to do the same for expressions with − operators, we have to make sure that we *flip* the operator associated with the recognised operator token, in order to make the value which is recognised as second operand to become the right hand side operand:

```
pMinus′ = applyAll <$> pInteger <*> pMany (flip (−) <$ pSyms "-"
                                                      <*> pInteger
                                           )
flip f x y = f y x
```

From here it is only a small step to the recognition of expressions which contain both + and − operators:

```
pPlusMinus = applyAll <$> pInteger
                      <*> pMany (  (  flip (−) <$ pSyms "-"
                                   <|>
                                      flip (+) <$ pSyms "+"
                                   )  <*> pInteger
                                )
```

Since we will use this pattern often we abstract from it and introduce a parser combinator *pChainL*, which takes two arguments:

1. the parser for the separator, which returns a value of type $a \to a \to a$
2. the parser for the operands, which returns a value of type a

Using this operator, we redefine the function *pPlusMinus*:

```
pChainL :: Parser s (a → a → a) → Parser s a → Parser s a
pChainL op p = applyAll <$> p <*> pMany (flip <$> op <*> p)
pPlusMinus′ = ((−) <$ pSyms "-" <|> (+) <$ pSyms "+")
              ‘pChainL‘
              pInteger
```

3.4 Left Factoring: *pChainR*, <**> and <??>

As a natural companion to *pChainL*, we would expect a *pChainR* combinator, which treats the recognised operators right-associatively. Before giving its code, we first introduce two other operators, which play an important role in fighting common sources of inefficiency. When we have the parser p:

$$\begin{aligned} p = {} & \qquad f \mathbin{<\$>} q \mathbin{<*>} r \\ & \mathbin{<|>} g \mathbin{<\$>} q \mathbin{<*>} s \end{aligned}$$

then we see that our backtracking implementation may first recognise the q from the first alternative, subsequently can fail when trying to recognise r, and will then continue with recognising q again before trying to recognise an s. Parser generators recognise such situations and perform a grammar transformation (or equivalent action) in order to share the recognition of q between the two alternatives. Unfortunately, we do not have an explicit representation of the underlying grammar at hand which we can inspect and transform [16], and without further help from the programmer there is no way we can identify such a common *left-factor*. Hence, we have to do the left-factoring by hand. Since this situation is quite common, we introduce two operators which assist us in this process. The first one is a modification of <*>, which we have named <**>; it differs from <*> in that it applies the result of its right-hand side operand to the result of its left-hand side operand:

$$\begin{aligned} & (\mathbin{<**>}) :: \mathit{Parser}\ s\ b \to \mathit{Parser}\ s\ (b \to a) \to \mathit{Parser}\ s\ a \\ & p \mathbin{<**>} q = (\lambda a\ f \to f\ \ a) \mathbin{<\$>} p \mathbin{<*>} q \end{aligned}$$

With help of this new operator we can now transcribe the above example, introducing calls to *flip* because the functions f and g now get their second arguments first, into:

$$p = q \mathbin{<**>} (\mathit{flip}\ f \mathbin{<\$>} r \mathbin{<|>} \mathit{flip}\ g \mathbin{<\$>} s)$$

In some cases, the element s is missing from the second alternative, and for such situations we have the combinator <??>:

$$\begin{aligned} & (\mathbin{<??>}) :: \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ (a \to a) \to \mathit{Parser}\ s\ a \\ & p \mathbin{<??>} q = p \mathbin{<**>} (q\ \text{‘}\mathit{opt}\text{‘}\ \mathit{id}) \end{aligned}$$

Let us now return to the code for *pChainR*. Our first attempt reads:

$$\begin{aligned} & \mathit{pChainR}\ \mathit{op}\ p = \mathit{id} \qquad\quad \mathbin{<\$>} p \\ & \quad \mathbin{<|>} \qquad\qquad \mathit{flip}\ (\$) \mathbin{<\$>} p \mathbin{<*>} (\mathit{flip} \mathbin{<\$>} \mathit{op} \mathbin{<*>} \mathit{pChainR}\ \mathit{op}\ p) \end{aligned}$$

which can, using the refactoring method, be expressed more elegantly by:

$$\mathit{pChainR}\ \mathit{op}\ p = r\ \mathbf{where}\ r = p \mathbin{<??>} (\mathit{flip} \mathbin{<\$>} \mathit{op} \mathbin{<*>} r)$$

3.5 Two Precedence Levels

Looking back at the definition of *pPlusMinus*, we see still a recurring pattern, i.e. the recognition of an operator symbol and associating it with its semantics.

This is the next thing we are going to abstract from. We start out by defining a function that associates an operator terminal symbol with its semantics:

$$\mathit{pOp}\ (\mathit{sem}, \mathit{symbol}) = \mathit{sem} \mathbin{<\$} \mathit{pSyms}\ \mathit{symbol}$$

Our next library combinator *pChoice* takes a list of parsers and combines them into a single parser:

$$\mathit{pChoice} = \mathit{foldr}\ (<\!|\!>)\ \mathit{pFail}$$

Using these two combinators, we now can define the collective additive operator recognition by:

$$\begin{aligned}
&\mathit{anyOp} = \mathit{pChoice}.\mathit{map}\ \mathit{pOp}\\
&\mathit{addops} = \mathit{anyOp}\ [((+), \texttt{"+"}), ((-), \texttt{"-"})]
\end{aligned}$$

Since multiplication has precedence over addition, we can now define a new non-terminal *pTimes*, which can only recognise operands containing multiplicative operators:

$$\begin{aligned}
&\mathit{pPlusMinusTimes} &&= \mathit{pChainL}\ \mathit{addops}\ \mathit{pTimes}\\
&\mathit{pTimes} &&= \mathit{pChainL}\ \mathit{mulops}\ \mathit{pInteger}\\
&\mathit{mulops} &&= \mathit{anyOp}\ [((*), \texttt{"*"})]
\end{aligned}$$

3.6 Any Number of Precedence Levels: *pPack*

Of course, we do not want to restrict ourselves to just two priority levels. On the other hand, we are not looking forward to explicitly introduce a new non-terminal for each precedence level, so we take a look at the code, and try to see a pattern. We start out by substituting the expression for *pTimes* into the definition of *pPlusMinusTimes*:

$$\mathit{pPlusMinusTimes} = \mathit{pChainL}\ \mathit{addops}\ (\mathit{pChainL}\ \mathit{mulops}\ \mathit{pInteger})$$

in which we recognise a *foldr*:

$$\mathit{pPlusMinusTimes} = \mathit{foldr}\ \mathit{pChainL}\ \mathit{pInteger}\ [\mathit{addops}, \mathit{mulops}]$$

Now it has become straightforward to add new operators: just add the new operator, with its semantics, to the corresponding level of precedence. If its precedence lies between two already existing precedences, then just add a new list between these two levels. To complete the parsing of expressions we add the recognition of parentheses:

$$\begin{aligned}
&\mathit{pPack} &&:: \mathit{Eq}\ s \Rightarrow [s] \rightarrow \mathit{Parser}\ s\ a \rightarrow [s] \rightarrow \mathit{Parser}\ s\ a\\
&\mathit{pPack}\ o\ p\ c &&= \mathit{pSyms}\ o \mathbin{*\!>} p \mathbin{<\!*} \mathit{pSyms}\ c
\end{aligned}$$

$$\begin{aligned}
&\mathit{pExpr} &&= \mathit{foldr}\ \mathit{pChainL}\ \mathit{pFactor}\ [\mathit{addops}, \mathit{mulops}]\\
&\mathit{pFactor} &&= \mathit{pInteger} \mathbin{<\!|\!>} \mathit{pPack}\ \texttt{"("}\ \mathit{pExpr}\ \texttt{")"}
\end{aligned}$$

As a final extension we add recognition of conditional expressions. In order to do so we will need to recognise keywords like **if**, **then**, and **else**. This invites us to add the companion to the *pChoice* combinator:

```
pSeq :: [Parser s a] → Parser s [a]
pSeq (p : pp) = (:) <$> p <*> pSeq pp
pSeq [] = pReturn []
```

Extending our parser with conditional expressions is now straightforward:

```
pExpr        = foldr pChainL pFactor [addops, mulops] <|> pIfThenElse
pIfThenElse = choose <$      pSyms "if"
                     <*>     pBoolExpr
                      <*     pSyms "then"
                     <*>     pExpr
                      <*     pSyms "else"
                     <*>     pExpr
choose c t e = if c then t else e
pBoolExpr   = foldr pChainR pRelExpr [orops, andops]
pRelExpr    =        True <$ pSyms "True"
                 <|> False <$ pSyms "False"
                 <|> pExpr <**> pRelOp <*> pExpr
andops  = anyOp [((∧) , "&&")]
orops   = anyOp [((∨) , "||")]
pRelOp = anyOp [((⩽), "<="), ((⩾), ">="),
                ((≡), "=="), ((≢), "/="),
                ((<), "<"), ((>)  , ">")]
```

3.7 Monadic Interface: *Monad* and *pTimes*

The parsers described thus far have the expressive power of context-free grammars. We have introduced extra combinators to capture frequently occurring grammatical patterns such as in the EBNF extensions. Because parsers are normal Haskell values, which are computed at run-time, we can however go beyond the conventional context-free grammar expressiveness by using the result of one parser to construct the next one. An example of this can be found in the recognition of XML-based input. We assume the input be a tree-like structure with tagged nodes, and we want to map our input onto the data type *XML*. To handle situations like this we make our parser type an instance of the class *Monad*:

```
instance Monad (Parser s) where
return = pReturn
P pa ≫= a2pb = P (λinput → [b_input″ | (a, input′) ← pa input
                                      , b_input″   ← unP (a2pb a) input′]
                  )
data XML = Tag String [XML] | Leaf String
pXML =         do t ← pOpenTag
                  Tag t <$> pMany pXML <* pCloseTag t
          <|> Leaf <$> pLeaf
```

```
pTagged p    = pPack "<" p ">"
pOpenTag     = pTagged pIdent
pCloseTag t  = pTagged (pSym '/' *> pSyms t)
pLeaf        = ...
pIdent       = pMany1 (pSatisfy (λc → 'a' ⩽ c ∧ c ⩽ 'z'))
```

A second example of the use of monads is in the recognition of the language $\{a^n b^n c^n | n >= 0\}$, which is well known not to be context-free. Here, we use the number of 'a''s recognised to build parsers that recognise exactly that number of 'b''s and 'c''s. For the result, we return the original input, which has now been checked to be an element of the language:

```
pABC = do as ← pMany (pSym 'a')
          let n = length as
          bs ← p_n_Times n (pSym 'b')
          cs ← p_n_Times n (pSym 'c')
          return (as ++ bs ++ cs)
p_n_Times :: Int → Parser s a → Parser s [a]
p_n_Times 0 p = pReturn [ ]
p_n_Times n p = (:) <$> p <*> p_n_Times (n − 1) p
```

3.8 Conclusions

We have now come to the end of our introductory section, in which we have introduced the idea of a combinator language and have constructed a small library of basic and non-basic combinators. It should be clear by now that there is no end to the number of new combinators that can be defined, each capturing a pattern recurring in some input to be recognised. We finish this section by summing up the advantages of using an EDSL.

full abstraction Most special purpose programming languages have –unlike our host language Haskell– poorly defined abstraction mechanisms, often not going far beyond a simple macro-processing system. Although –with a substantial effort– amazing things can be achieved in this way as we can see from the use of TEX, we do not think this is the right way to go; programs become harder to get correct, and often long detours –which have little to do with the actual problem at hand– have to be taken in order to get things into acceptable shape. Because our embedded language inherits from Haskell –by virtue of being an embedded language– all the abstraction mechanisms and the advanced type system, it takes a head start with respect to all the individual implementation efforts.

type checking Many special purpose programming languages, and especially the so-called scripting languages, only have a weak concept of a type system; simply because the type system was not considered to be important when the design took off and compilers should remain small. Many scripting languages are completely dynamically typed, and some see this even as an advantage since the type system does not get into their way when implementing new abstractions. We feel that this perceived shortcoming is due to the very basic type systems found in most general purpose programming languages. Haskell however has a very powerful type system, which is not easy to surpass, unless one is prepared to enter completely new grounds, as with dependently typed languages such as Agda (see paper in this volume by Bove and Dybjer). One of the huge benefits of working with a strongly typed language is furthermore that the types of the library functions already give a very good insight in the role of the parameters and what a function is computing.

clean semantics One of the ways in which the meaning of a language construct is traditionally defined is by its denotational semantics, i.e. by mapping the language construct onto a mathematical object, usually being a function. This fits very well with the embedding of domain specific languages in Haskell, since functions are the primary class of values in Haskell. As a result, implementing a DSL in Haskell almost boils down to giving its denotational semantics in the conventional way and getting a compiler for free.

lazy evaluation One of the formalisms of choice in implementing the context sensitive aspects of a language is by using attribute grammars. Fortunately, the equivalent of attribute grammars can be implemented straightforwardly in a lazily evaluated functional language; inherited attributes become parameters and synthesized attributes become part of the result of the functions giving the semantics of a construct [14, 13].

Of course there are also downsides to the embedding approach. Although the programmer is thinking he writes a program in the embedded language, he is still programming in the host language. As a result of this, error messages from the type system, which can already be quite challenging in Haskell, are phrased in terms of the host language constructs too, and without further measures the underlying implementation shines through. In the case of our parser combinators, this has as a consequence that the user is not addressed in terms of terminals, non-terminals, keywords, and productions, but in terms of the types implementing these constructs.

There are several ways in which this problem can be alleviated. In the first place, we can try to hide the internal structure as much as possible by using a lot of **newtype** constructors, and thus defining the parser type by:

$$\textbf{newtype}\ \mathit{Parser}'\ s\ a = \mathit{Parser}'\ ([s] \rightarrow [(a, [s])])$$

A second approach is to extend the type checker of Haskell such that the generated error messages can be tailored by the programmer. Now, the library designer not only designs his library, but also the domain specific error messages that come with the library. In the Helium compiler [5], which handles a subset of Haskell, this approach has been implemented with good results. As an example, one might want to compare the two error messages given for the incorrect program in Fig. 1. In Fig. 2 we see the error message generated by a version of Hugs, which does not even point near the location of the error, and in which the internal representation of the parsers shines through. In Fig. 3, taken from [6], we see that Helium, by using a specialised version of the type rules which are provided by the programmer of the library, manages to address the application programmer in terms of the embedded language; it uses the word *parser* and explains that the types do not match, i.e. that a component is missing in one of the alternatives. A final option in the Helium compiler is the possibility to program the search for possible corrections, e.g. by listing functions which are likely to be confused by the programmer (such as <*> and <* in programming parsers, or : and ++ by beginning Haskell programmers). As we can see in Fig. 4 we can now pinpoint the location of the mistake even better and suggest corrective actions.

```
data Expr     = Lambda Patterns Expr    -- can contain more alternatives
type Patterns = [Pattern]
type Pattern  = String

pExpr :: Parser Token Expr
pExpr
  =  pAndPrioExpr
 <|> Lambda <$  pSyms "\\"
            <*> many  pVarid
            <*  pSyms "->"
            <*  pExpr         -- <* should be <*>
```

Fig. 1. Type incorrect program

```
ERROR "Example.hs":7 - Type error in application
*** Expression     : pAndPrioExpr <|> Lambda <$ pSyms "\\"
                        <*> many pVarid <* pSyms "->" <* pExpr
*** Term           : pAndPrioExpr
*** Type           : Parser Token Expr
*** Does not match : [Token] -> [(Expr -> Expr,[Token])]
```

Fig. 2. Hugs, version November 2002

4 Improved Implementations

Since the simple implementation which was used in section 2 has quite a number of shortcomings we develop in this section a couple of alternative implementations of the basic interface. Before doing so we investigate the problems to be solved, and then deal with them one by one.

```
Compiling Example.hs
(7,6): The result types of the parsers in the operands of <|> don't match
  left parser   : pAndPrioExpr
    result type : Expr
  right parser  : Lambda <$ pSyms "\\" <*> many pVarid <* pSyms "->"
                                                       <* pExpr
    result type : Expr -> Expr
```

Fig. 3. Helium, version 1.1 (type rules extension)

```
Compiling Example.hs
(11,13): Type error in the operator <*
  probable fix: use <*> instead
```

Fig. 4. Helium, version 1.1 (type rules extension and sibling functions)

4.1 Shortcomings

Error reporting One of the first things someone notices when starting to use the library is that when erroneous input is given to the parser the result is [ ], indicating that it is not possible to get a correct parse. This might be acceptable in situations where the input was generated by another program and is expected to be correct, but for a library to be used by many in many different situations this is unacceptable. At least one should be informed about the position in the input where the parser got stuck, and what symbols were expected.

Online Results A further issue to be investigated is at what moment the result of the parser will become available for further processing. When reading a long list of records –such as a BiBTeX file–, one is likely to want to process the records one by one and to emit the result of processing it as soon as it has been recognised, instead of first recognising the complete list, storing that list in memory, and finally –after we know that the input does not contain errors– process all the elements.

When we inspect the code for the sequential composition closely however, and investigate when the first element of the resulting list will be produced, we see that this is only the case after the right-hand side parser of <*> returns its first result. For the root symbol this implies that we get only to see the result after we have found our first complete parse. So, taking the observation of the previous subsection into account, at the end of the first complete parse we have stored the complete input and the complete result in memory. For long inputs this may become prohibitively costly, especially since garbage collection will take a lot of time without actually collecting a lot of garbage.

To illustrate the difference consider the parser:

$\mathit{parse}\ (\mathit{pMany}\ (\mathit{pSym}\ \texttt{'a'}))\ (\mathit{listToStr}\ (\texttt{'a'} : \bot))$

The parsers we have seen thus far will produce $\bot$ here. An online parser will return $\texttt{'a'} : \bot$ instead, since the initial `'a'` could be succesfully recognised irrespective of what is behind it in the input.

Error Correction Although this is nowadays less common, it would be nice if the parser could apply (mostly small) error repairing steps, such as inserting a missing closing parenthesis or **end** symbol. Also spurious tokens in the input stream might be deleted. Of course the user should be properly informed about the steps which were taken in order to be able to proceed parsing.

Space Consumption The backtracking implementation may lead to unexpected space consumption. After the parser p in a sequential composition $p \mathbin{<\!*\!>} q$ has found its first complete parse, parsing by q commences. Since this may fail further alternatives for p may have to be tried, even when it is obvious from the grammar that these will all fail. In order to be able to continue with the backtracking process (i.e. go back to a previous choice point) the implementation keeps a reference in the input which was passed to the composite parser. Unfortunately this is also the case for the root symbol, and thus the complete input is kept in memory at least until the first complete parse has been found, and its witness has been selected as the one to use for further processing

This problem is well known from many systems based on backtracking implementations. In Prolog we have the *cut* clause to explicitly indicate points beyond which no backtracking should take place, and also some parser combinator libraries [9] have similar mechanisms.

Conclusions Although the problems at first seem rather unrelated they are not. If we want to have an online result this implies that we want to start processing a result without knowing whether a complete parse can be found. If we add error correction we actually change our parsers from parsers which may fail to parsers which will always succeed (i.e. return a result), but probably with an error message. In solving the problems mentioned we will start with the space consumption problem, and next we change the implementation to produce online results. As we will see special measures have to be taken to make the described parsers instances of the class *Monad*.

We will provide the full code in this tutorial. Unfortunately when we add error reporting and error correction our way of presenting code in an incremental way leads to code duplication. So we will deal with the last two issues separately in Sect. 7.

4.2 Parsing Classes

Since we will be giving many different implementations and our aim is to construct a library which is generally usable, we start out by defining some classes.

Applicative Since the basic interface is useable beyond the basic parser combinators from Sect. 3 we introduce a class for it: *Applicative*. [1]

```
class Applicative p where
  (<*>)   :: p (b → a) → p b → p a
  (<|>)   :: p a       → p a → p a
  (<$>)   :: (b → a)   → p b → p a
  pReturn :: a               → p a
  pFail   ::                   p a
  f <$> p = pReturn f <*> p
instance Applicative p ⇒ Functor p where
  fmap = (<$>)
```

The class *Describes* Although for parsing the input is just a sequence of terminal symbols, in practice the situation is somewhat different. We assume our grammars are defined in terms of terminal *symbols*, whereas we can split our input state into the next *token* and a a new *state*. A token may contain extra position information or more detailed information which is not relevant for the parsing process. We have seen an example already of the latter; when parsing we may want to see an identifier, but it is completely irrelevant which identifier is actually recognised. Hence we want check whether the current token matches with an expected symbol. Of course these values do not have to be of the same type. We capture the relation between input *tokens* and terminal *symbols* by the class *Describes*:

```
class symbol `Describes` token where
  eqSymTok :: symbol → token → Bool
```

Recognising a single symbol: *Symbol* The function *pSym* takes as parameter a terminal *symbol*, but returns a parser which has as its witness an input *token*. Because we again will have many different implementations we make *pSym* a member of a class too.

```
class Symbol p symbol token where
  pSym :: symbol → p token
```

[1] We do not use the class *Applicative* from the module *Control.Applicative*, since it provides standard implementations for some operations for which we want to give optimized implementations, as the possibility arises.

Generalising the Input: *Provides* In the previous section we have taken the input to be a list of tokens. In reality this may also be a too simple approach. We may e.g. want to maintain position information, or extra state which can be manipulated by special combinators. From the parsing point of view the thing that matters is that the input state can provide a *token* on demand if possible:

```
class Provides state symbol token | state symbol → token where
  splitState :: symbol → state → Maybe (token, state)
```

We have decided to pass the expected *symbol* to the function *splitState*. Since we will also be able to switch *state* type we have decided to add a functional dependecy *state symbol* → *token*, stating that the *state* together with the expected *symbol* type determines how a token is to be produced. We can thus switch from one scanning stategy to another by passing a symbol of a different type to pSym!

Calling a parser: *Parser* We will often have to check whether we have read the complete input, and thus we introduce a class containing the function *eof* (end-of-file) which tells us whether more tokens have to be recognised:

```
class Eof state where
  eof :: state → Bool
```

Because our parsers will all have different interfaces we introduce a function *parse* which knows how to call a specific parser and how to retrieve the result:

```
class Parser p where
      parse :: p state a → state → a
```

The instances of this class will serve as a typical example of how to use a parser of type p from within a Haskell program. For specific implementations of p, and in specific circumstances one may want to vary on the given standard implementations.

4.3 From Depth-first to Breadth-first

In this section we will define four instances of the *Parser* class:

1. the type R ('recognisers') in subsection 4.4
2. the type P_h ('history parsers') in subsection 4.5,
3. the type P_f ('future parsers') in subsection 4.6, and
4. the type P_m ('monad parsers') in subsection 4.7.

All four types will be polymorphic, having two type parameters: the type of the state, and the type of the witness of the correct parse. This is a digression from the parser type in Sect. 2, which was polymorphic in the *symbol* type and the witness type.

All four types will be functions, which operate on a state rather than a list of symbols. The state type must be an instance of *Provides* together with a symbol and a token type, and the symbol and the token must be an instance of *Describes*.

A further digression from section 2 is that the parsers in this section are not ambiguous. Instead of a list of successes, they return a single result.

As a final digression, the result type of the parsers is not a pair of a witness and a final state, but a witness only wrapped in a *Steps* datatype. The *Steps* datatype will be introduced below. It is an encoding of whether there is failure or success, and in the case of success, how much input was consumed.

As we explained before, the list-of-successes method basically is a depth-first search technique. If we manage to change this depth-first approach into a breath-first approach, then there is no need to hang onto the complete input until we are finished parsing. If we manage to run all alternative parsers in parallel we can discard the current input token once it has been inspected by all active parsers, since it will never be inspected again.

Haskell's lazy evaluation provides a nice way to drive all the active alternatives in a step by step fashion. The main ingredient for this process is the data type *Steps*, which plays a crucial role in all our implementations, and describes the type of values constructed by all parsers to come. It can be seen as a lazily constructed trace representing the progress of the parsing process.

```
data Steps a where
  Step :: Steps a → Steps a
  Fail ::           Steps a
  Done :: a       → Steps a
```

Instead of returning just a witness from the parsing process we will return a nested application of *Step*'s, which has eventually a *Fail* constructor indicating a failed branch in our breadth-first search, or a *Done* constructor which indicates that parsing completed successfully and presents the witness of that parse. For each successfully recognised symbol we get a *Step* constructor in the resulting *Steps* sequence; thus the number of *Step* constructors in the result of a parser tells us up to which point in the input we have successfully proceeded, and more specifically if the sequence ends in a *Fail* the number of *Step*-constructors tell us where this alternative failed to proceed.

The function driving our breadth-first behaviour is the function *best*, which compares two *Steps* sequences and returns the "best" one:

```
best :: Steps a → Steps a → Steps a
Fail     `best` r        = r
l        `best` Fail     = l
(Step l) `best` (Step r) = Step (l `best` r)
_        `best` _        = error "incorrect parser"
```

The last alternative covers all the situations, where either one parser completes and another is still active (*Step*‘*best*‘ *Done*,*Done*‘*best*‘ *Step*), or where two active parsers complete at the same time (*Done* / *Done*) as a result of an ambiguity in the grammar. For the time being we assume that such situations will not occur.

The alternative which takes care of the conversion from depth-first to breadth-first is the one in which both arguments of the *best* function start with a *Step* constructor. In this case we discover that both alternatives can make progress, so the combined parser can make progress by immediately returning a *Step* constructor; we do however not decide nor reveal yet which alternative eventually will be chosen. The expression *l*‘*best*‘ *r* in the right hand side is lazily evaluated, and only unrolled further when needed, i.e. when further pattern matching takes place on this value, and that is when all *Step* constructors corresponding to the current input position have been merged into a single *Step*. The sequence associated with this *Step* constructor is internally an expression, consisting of further calls to the function *best*. Later we will introduce more elaborate versions of this type *Steps*, but the idea will remain the same, and they will all exhibit the breadth-first behaviour.

In order to retrieve a value from a *Steps* value we write a function *eval* which retrieves the value remembered by the *Done* at the end of the sequence, provided it exists:*

```
eval :: Steps a → a
eval (Step l)   = eval l
eval (Done v) = v
eval Fail       = error "should not happen"
```

4.4 Recognisers

After the preparatory work introducing the *Steps* data type, we introduce our first ‘parser’ type, which we will dubb *recogniser* since it will not present a witness; we concentrate on the recognition process only. The type of *R* is polymorphic in two type parameters: *st* for the state, and *a* for the witness of the correct parse. Basically a recogniser is a function taking a state and returning *Steps*. This *Steps* value starts with the steps produced by the recogniser itself, but ends with the steps produced by a continuation which is passed as the first argument to the recogniser:

```
newtype R st a = R (∀ r.(st → Steps r) → st → Steps r)
unR (R p) = p
```

Note that the type *a* is not used in the right hand side of the definition. To make sure that the recognisers and the parsers have the same kind we have included this type parameter here too; besides making it possible to to make use of all the calsses we introduce for parsers it also introduces extra check on the wellformedness of recognisers. Furthermore we can now, by provinding a top level type specification use the same expression to just recognise something or to parse with building a result.

We can now make *R* an instance of *Applicative*, that is implement the five classic parser combinators for it. Note that the parameter *f* of the operator <\$> is irnored, since it does not play a role in the reognition process, and the same holds for the parameter *a* of *pReturn*.

```
instance Applicative (R st) where
         R p <*> R q = R (λk st → p (q k) st)
         R p <|> R q = R (λk st → p k st ‘best‘ q k st)
         f <$>   R p = R p
         pReturn a   = R (λk st → k st)
         pFail       = R (λk st → Fail)
```

We have abstained from giving point-free definitions, but one can easily see that sequential composition is essentially function composition, and that *pReturn* is the identity function wrapped in a constructor.

Next we provide the implementation of *pSym*, which resembles the definition in the basic library. Note that when a symbol is succesfully recognised this is reflected by prefixing the result of the call to the continuation with a *Step*:

```
instance (symbol ‘Describe‘ s token, Provides state symbol token)
  ⇒ Symbol (R state) symbol token where
  pSym a = R (λk h st → case splitState a st of
                          Just (t, ss) → if a ‘eqSymTok‘ t
                                         then Step (k ss)
                                         else Fail
                          Nothing      → Fail)
```

4.5 History Based Parsers

After the preparatory work introducing the *Steps* data type and the recognisers, we now introduce our first parser type, which we will call *history* parsers. The type P_h takes the same type parameters as the recogniser: *st* for the state, and *a* for the witness of the correct parse. The actual parsing function takes, besides the continuation and the state an extra parameter in its second position..

The second parameter is the ‘history’: a stack containing all the values recognised as the left hand side of a <*> combinator which have thus far not been paired with the result of the corresponding right hand side parser. The first parameter is again the ‘continuation’, a function which is responsible, being passed the history extended with the newly recognised witness, to produce the final result from the rest of the input.

In the type P_h, we have local type variables for the type of the history h, and the type of the witness of the final result r:

```
newtype Pₕ st a = Pₕ (∀ r h  .  ((h, a) → st → Steps r)
                             →    h     → st → Steps r)
unPₕ (Pₕ p) = p
```

We can now make P_h an instance of *Applicative*, that is, implement the five classic parser combinators for it.

In the definition of *pReturn*, we encode that the history parameter is indeed a stack, growing to the right and implemented as nested pairs. The new witness is pushed on the history stack, and passed on to the continuation k.

In the definition of f <\$>, the continuation is modified to sneak in the application of the function f.

In the definition of alternative composition <|>, we call both parsers and exploit the fact that they both return *Steps*, of which we can take the *best*. Of course, *best* only lazily unwraps both *Steps* up to the point where one of them fails.

In the definition of sequential composition <*>, the continuation-passing style is again exploited: we call p, passing it q as continuation, which in turn takes a modification of the original continuation k. The modification is that two values are popped from the history stack: the witness b from parser q, and the witness $b2a$ from parser p; and a new value $b2a$ b is pushed onto the history stack which is passed to the orginal continuation k:

```
instance Applicative (Pₕ state) where
  Pₕ p <*> Pₕ q = Pₕ (λk → p (q applyₕ)
                          where applyₕ = λ((h, b2a), b) → k (h, b2a b))
  Pₕ p <|> Pₕ q = Pₕ (λk h st → p k h st ‘best‘ q k h st)
  f    <$> Pₕ p = Pₕ (λk      → p $ λ(h, a) → k (h, f a))
  pFail         = Pₕ (λk _ _  → Fail                       )
  pReturn a     = Pₕ (λk h    → k (h, a)                   )
```

Note that we have given a new definition for <\$>, which is slightly more efficient than the default one; instead of pushing the function f on the stack with a *pReturn* and popping it off later, we just apply it directly to recognised result of the parser p. In Fig. 5 we have given a pictorial representation of the flow of data associated with this parser type. The top arrows, flowing right, correspond to the accumulated history, and the arrows directly below them to the state which is passed on. The bottom arrows, flowing left, correspond to the final result which is returned through all the continuation calls.

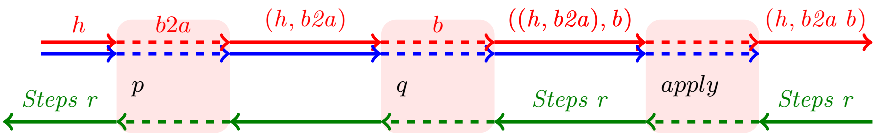

Fig. 5. Sequential composition of history parsers

In a slightly different formulation the stack may be respresented implicitly using extra continuation functions. From now on we will use a somewhat simpler type for $P - h$ and thus we also provide a new instance definition for the class *Applicative*. It is however useful to keep te pictorial representation of the earlier type in mind.:

```
Pₕ st a = Pₕ (∀ r.(a → st → Steps r) → st → Steps r
instance Applicative (Pₕ state) where
  (Pₕ p) <*> (Pₕ q) = Pₕ (λk → p (λf → q (λa → k (f a))))
  (Pₕ p) <|> (Pₕ q) = Pₕ (λk inp → p k inp ‘best‘ q k inp)
  f <$> (Pₕ p)      = Pₕ (λk → p (λa → k (f a)))
  pFail             = Pₕ (λk → const noAlts)
  pReturn a         = Pₕ (λk → k a)
```

The definition of *pSym* is straightforward; the recognised token is passed on to the continuation:

```
instance (symbol‘Describes′ token, Provides state symbol token)
  ⇒ Symbol (Pₕ state) symbol token where
  pSym a = Pₕ (λk st → case splitState a st of
                         Just (t, ss) → if a ‘eqSymTok‘ t
                                        then Step (k t ss)
                                        else Fail
                         Nothing      → Fail)
```

Finally we make P_h an instance of *Parser* by providing a function *parse* that checks whether all input was consumed; if so we initialise the return sequence with a *Done*with the final conctructed witness.

```
instance Eof state ⇒ Parser (Pₕ state) where
  parse (Pₕ p)
    = eval.p (λr rest → if eof rest then Done r else Fail)
```

Since we will later be adding error recovery to the parsers constructed in this chapter, which will turn every illegal input into a legal one, we will assume in this section that there exists *always precisely one way of parsing the input.* If there is more than one way then we have to deal with ambiguities, which we will also show how to deal with in section 5.

4.6 Producing Results Online

The next problem we are attacking is producting the result online. The history parser accumulates its result in an extra argument, only to be inserted at the end of the parsing process with the *Done* constructor. In this section we introduce the counterpart of the history parser, the *future* parser, which is named this way because the “stack” we are maintaining contains elements which still have to come into existence. The type of future parsers is:

```
newtype P_f st a = P_f (∀ r.(st → Steps r) → st → Steps (a, r))
unP_f (P_f p) = p
```

We see that the history parameter has disappeared and that the parameter of the *Steps* type now changes; instead of just passing the result constructed by the call to the continuation unmodified to the caller, the constructed witness a is pushed onto the stack of results constructed by the continuation; this stack is made an integral part of the data type *Steps* by not only representing progress information but also constructed values in this sequence.

In our programs we will make the stack grow from the right to the left; this maintains the suggestion introduced by the history parsers that the values to the right correspond to input parts which are located further towards the end of the input stream (assuming we read the stream from left to right). One way of pushing such a value on the stack would be to traverse the whole future sequence until we reach the *Done* constructor and then adding the value there, but that makes no sense since then the result again will not be available online. Instead we extend our *Steps* data type with an extra constructor. We remove the *Done* constructor, since it can be simulated with the new *Apply* constructor. The *Apply* constructor makes it possible to store function values in the progress sequence:

```
data Steps a where
    Step  ::              Steps a → Steps a
    Fail  ::                        Steps a
    Apply :: (b → a) → Steps b → Steps a

eval :: Steps a → a
eval (Step     l)  = eval l
eval (Fail     ls) = error "no result"
eval (Apply f  l ) = f (eval l)
```

As we have seen in the case of the history parsers there are two operations we perform on the stack: pushing a value, and popping two values, applying the one to the other and pushing the result back. For this we define two auxiliary functions:

```
push    :: v → Steps r → Steps (v, r)
push v = Apply (λs → (v, s))

apply_f  :: Steps (b → a, (b, r)) → Steps (a, r)
apply_f = Apply (λ(b2a, ~(b, r)) → (b2a b, r))
```

One should not confuse the *Apply* constructor with the $apply_f$ function. Keep in mind that the *Apply* constructor is a very generally applicable construct changing the value (and possibly the type) represented by the sequence by prefixing the sequence with a function value, whereas the $apply_f$ function takes care of combining the values of two sequentially composed parsers by applying the result of the first one to the result of the second one. An important rôle is played by the ~-symbol. Normally, Haskell evaluates arguments to functions far enough to check that it indeed matches the pattern. The tilde prevents this by making Haskell assume that the pattern always matches. Evaluation of the argument is thus slightly more lazy, which is critically needed here: the function *b2a* can already return that part of the result for which evaluation of its argument is not needed!

The code for the the function *best* now is a bit more involved, since there are extra cases to be taken care of: a *Steps* sequence may start with an *Apply* step. So before calling the actual function *best* we make sure that the head of the stream is one of the constructors that indicates progress, i.e. a *Step* or *Fail* constructor. This is taken care of by the function *norm* which pushes *Apply* steps forward into the progress stream until a progress step is encountered:

```
norm :: Steps a → Steps a
norm (Apply f (Step l      )) = Step (Apply f l)
norm (Apply f Fail          ) = Fail
norm (Apply f (Apply g l))    = norm (Apply (f.g) l)
norm steps                    = steps
```

Our new version of *best* now reads:

```
l ‘best‘ r = norm l ‘best′‘ norm r
  where Fail     ‘best′‘ r        = r
        l        ‘best′‘ Fail     = l
        (Step l) ‘best′‘ (Step r) = Step (l ‘best‘ r)
        _        ‘best′‘ _        = Fail
```

We as well make P_f an instance of *Applicative*:

```
instance Applicative (P_f st) where
  P_f p <⊛> P_f q = P_f (λk st → apply_f (p (q k) st))
  P_f p <|> P_f q = P_f (λk st → p k st ‘best‘ q k st )
  pReturn a       = P_f (λk st → push a (k st)        )
  pFail           = P_f (λ_ _  → Fail                 )
```

Just as we did for the history parsers we again provide a pictorial representation of the data flow in case of a sequential composition <⊛> in Fig. 6:

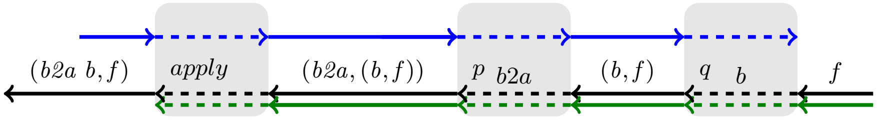

Fig. 6. Sequential composition of future parsers

Also the definitions of *pSym* and *parse* pose no problems. The only question is what to take as the initial value of the *Steps* sequence. We just take ⊥, since the types guarantee that it will never be evaluated. Notice that if the parser constructs the value b, then the result of the call to the parser in the function *parse* will be $(b, \bot)$ of which we select the first component after converting the returned sequence to the value represented by it.

```
instance (symbol `Describes` token, state `Provides` token)
           ⇒ Symbol (P_f state) symbol token where
  pSym a = P_f (λk st → case splitState a st of
                           Just (t, ss) → if a `eqSymTok` t
                                          then Step (push t (k ss))
                                          else Fail
                           Nothing      → Fail
               )
instance Eof state ⇒ Parser (P_f state)
where
  parse (P_f p) = fst.eval.p (λinp → if eof inp then ⊥ else error "end")
```

4.7 The Monadic Interface

As with the parsers from the introduction we want to make our new parsers instances of the class *Monad* too, so we can again write functions like *pABC* (see page 15). Making history parsers an instance of the class *Monad* is straightforward:

```
instance Applicative (P_h state) ⇒ Monad (P_h state) where
  P_h p ≫= a2q = P_h (λk → p (λa → unP_h (a2q a) k))
  return = pReturn
```

At first sight this does not seem to be a problem to proceed similarly for future parsers. Following the pattern of sequential composition, we call p with the continuation $unP_h\ (a2q\ a)\ k$; the only change is that instead of applying the result of p to the result of q we use the result of p to build the continuation in $a2q\ a$. And indeed the following code type-checks perfectly:

```
instance Applicative (P_f state) ⇒ Monad (P_f state) where
  P_f p ≫= pv2q = P_f (λk st →
                        let steps = p (q k) st
                            q     = unP_f (pv2q pv)
                            pv    = fst (eval steps)
                        in Apply snd steps
                       )
  return = pReturn
```

Unfortunately execution of the above code may lead to a black hole, i.e. a non-terminating computation, as we will explain with the help of Fig. 7. Problems occur when inside p we have a call to the function *best* which starts to compare two result sequences. Now suppose that in order to make a choice the parser p does not provide enough information. In that case the continuation q is called

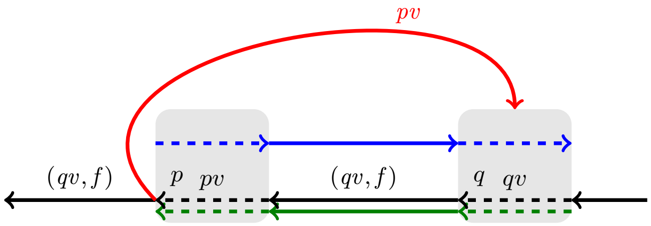

Fig. 7. Erroneous implementation of monadic future parsers

once for each branch of the choice process, in order to provide further steps of which we hope they will lead us to a decision. If we are lucky the value of pv is not needed by $q\ pv$ in order to provide the extra needed progress information. But if we are unlucky the value is needed; however the *Apply* steps contributing to pv will have been propagated into the sequence returned by q. Now we have constructed a loop in our computation: pv depends on the outcome of *best*, *best* depends on the outcome of $q\ pv$, and $q\ pv$ depends on the value of pv.

The problem is caused by the fact that each branch taken in p has its own call to the continuation q, and that each branch *may lead to a different value for* pv, but we get only one in our hands: the one which belongs to the successful alternative. So we are stuck.

Fortunately we remember just in time that we have introduced a different kind of parser, the history based ones, which have the property that they pass the value produced along the path taken inside them to the continuation. Each path splitting somewhere in p can thus call the continuation with the value which will be produced if this alternative wins eventually. That is why their implementation of *Monad*'s operations is perfectly fine. This brings us to the following insight: the reason we moved on from history based parsers to future based parsers was that we wanted to have an online result. But the result of the left-hand side of a monadic bind is not used at all in the construction of the result. Instead it is removed from the result stack in order to be used as a parameter to the right hand side operand of the monadic bind. So the solution to our problem lies in *using a history based parser as the left hand side of a monadic bind*, and a future based parser at the right hand side. Of course we have to make sure that they share the *Steps* data type used for storing the result. In Fig. 8 we have given a pictorial representation of the associated data flow.

Unfortunately this does not work out as expected, since the type of the $\gg\!=$ operator is $Monad\ m \Rightarrow m\ b \rightarrow (b \rightarrow m\ a) \rightarrow m\ a$, and hence requires the left and right hand side operands to be based upon the same functor m. A solution is to introduce a class *GenMod*, which takes two functor parameters instead of one:

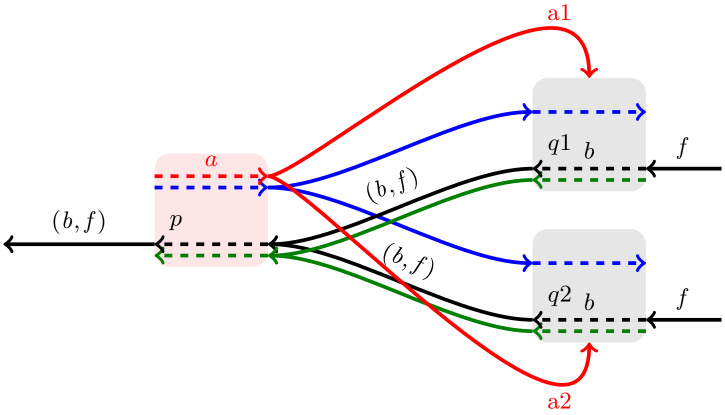

Fig. 8. Combining future based and history based parsers

```
infixr 1 ≫≫=
class GenMonad m_1 m_2 where
  (≫≫=) :: m_1 b → (b → m_2 a) → m_2 a
```

Now we can create two instances of *GenMonad*. In both cases the left hand side operand is the history parser, and the right hand side operand is either a history or a future based parser:

```
instance Monad (P_h state)
  ⇒       GenMonad (P_h state) (P_h state) where
  (≫≫=) = (≫=)   -- the monadic bind defined before
instance GenMonad (P_h state) (P_f state) where
  (P_h p) ≫≫= pv2q = P_f (λk → p (λpv → unP_h (pv2q pv) k))
```

Unfortunately we are now no longer able to use the **do** notation because that is designed for *Monad* expressions rather than for *GenMonad* expressions which was introduced for monadic expressions, and thus we still cannot replace the implementation in the basic library by the more advanced one we are developing. Fortunately there is a trick which makes this still possible: *we pair the two implementations, and select the one which we need*:

```
data P_m state a = P_m (P_h state a) (P_f state a)
unP_m_h (P_m (P_h h) _       ) = h
unP_m_f (P_m _       (P_f f)) = f
```

Our first step is to make this new type again instance of *Applicative*:

```
instance  ( Applicative (P_h st), Applicative (P_f st))
            ⇒ Applicative (P_m st) where
(P_m hp fp)    <*> ~(P_m hq fq) = P_m (hp <*> hq)  (fp <*> fq)
(P_m hp fp)    <|> (P_m hq fq)  = P_m (hp <|> hq)  (fp <|> fq)
pReturn a                       = P_m (pReturn a) (pReturn a)
pFail                           = P_m pFail        pFail
instance    (symbol `Describes` token, state `Provides` token)
  ⇒         Symbol (P_m state) symbol token where
  pSym a = P_m (pSym a) (pSym a)
instance Eof state ⇒ Parser (P_m state) where
  parse (P_m _ (P_f fp))
     = fst.eval.fp (λrest → if eof rest     then ⊥
                                             else error "parse")
```

This new type can now be made into a monad by:

```
instance Applicative (P_m st) ⇒ Monad (P_m st) where
  (P_m (P_h p) _) ≫= a2q =
       P_m (P_h (λk → p (λa → unP_m_h (a2q a) k)))
           (P_f (λk → p (λa → unP_m_f (a2q a) k)))
  return = pReturn
```

Special attention has to be paid to the occurrence of the ~ symbol in the left hand side pattern for the <*> combinator. The need for it comes from recursive definitions like:

```
pMany p = (:) <$> p <*> pMany p `opt` [ ]
```

If we match the second operand of the <*> occurrence strictly this will force the evaluation of the call *pMany p*, thus leading to an infinite recursion!

5 Exploiting Progress Information

Before continuing discussing the mentioned shortcomings such as the absence of error reporting and error correction which will make the data types describing the result more complicated, we take some time to show how the introduced *Steps* data type has many unconventional applications, which go beyond the expressive power of context-free grammars. Because both our history and future parsers now operate on the same *Steps* data type we will focus on extensions to that data type only.

5.1 Greedy Parsing

For many programming languages the context-free grammars which are provided in the standards are actually ambiguous. A common case is the dangling **else**. If we have a production like:

```
stat ::= "if" expr "then" stat ["else" stat]
```

then a text of the form `if ... then ... if ... then ... else ...` has two parses: one in which the **else** part is associated with the first **if** and one in which it is associated with the second. Such ambiguities are often handled by accompanying text in the standard stating that the second alternative is the interpretation to be chosen. A straightforward way of implementing this, and this is how it is done in quite a few parser generators, is to apply a greedy parsing strategy: if we have to choose between two alternatives and the first one can make progress than take that one. If the greedy strategy fails we fall back to the normal strategy.

In our approach we can easily achieve this effect by introducing a biased choice operator ≪|>, which for all purposes acts like <|>, but chooses its left alternative if it starts with the successful recognition of a token:

```
class Greedy p where
  (≪|>) :: p a → p a → p a
best_gr :: Steps a → Steps a → Steps a
l@(Step _ _) `best_gr` _ = l
l            `best_gr` r = l `best` r
instance Best_gr (P_h st) where
  P_h p ≪|> P_h q = P_h (λk st → p k st `best_gr` q k st)
```

The instance declarations for the other parser types are similar.

This common solution usually solves the problem adequately. It may however be the case that we only want to take a specific alternative if we can be sure that some initial part can completely be recognised. As a preparation for the discussion on error correction we show how to handle this. We extend the data type *Steps* with one further alternative:

```
data Steps a = ...
             | Success (Steps a)
```

and introduce yet another operator ⋘|> which performs its work in cooperation with a function *try*. In this case we only provide the implementation for the P_f case:

```
class Try p where
  (⋘|>) :: p a → p a → p a
  try ::      p a → p a
instance Try (P_f state) where
  P_f p ⋘|> P_f q = P_f (λk st → let l = p k st
                              in maybe (l `best` q k st) id (hasSuccess id l)
                             )
          where hasSuccess f (Step l     ) = hasSuccess (f.Step)      l
                hasSuccess f (Apply g l) = hasSuccess (f.Apply g) l
                hasSuccess f (Success l ) = Just (f l)
                hasSuccess f (Fail       ) = Nothing
      try (P_f p) = P_f (p.(Success.))
```

The function *try* does little more than inserting a *Success* marker in the result sequence, once its argument parser has completed successfully. The function *hasSuccess* tries to find such a marker. If found then the marker is removed and success (*Just*) reported, otherwise failure (*Nothing*) is returned. In the latter case our good old friend *best* takes its turn to compare both sequences in parallel as before. One might be inclined to think that in case of failure of the first alternative we should just take the second, but that is a bit too optimistic; the right hand side alternative might fail even earlier.

Unfortunately this simple approach has its drawback: what happens if the programmer forgets to mark an initial part of the left hand side alternative with *try*? In that case the function will never find a *Success* constructor, and our parsing process fails. We can solve this problem by introducing yet another parser type which guarantees that *try* has been used and thus that such a *Success* constructor may occur. We will not pursue this alternative here any further, since it will make our code even more involved.

5.2 Ambiguous Grammars

One of the big shortcomings of the combinator based approach to parsing, which is aggravated by the absence of global grammar analysis, is that we do not get a warning beforehand if our underlying grammar is ambiguous. It is only when we try to choose between two result sequences in the function *best* and discover that both end successfully, that we find out that our grammar allows more than one parse. Worse however is that parse times also may grow exponentially. For each successful parse for a given non-terminal the remaining part of the input is completely parsed. If we were only able to memoise the calls to the continuations, i.e. we can see that the same function is called more than once with the same argument, we could get rid of the superfluous work. Unfortunately continuations are anonymous functions which are not easily compared. If the programmer is however prepared to do some extra work by indicating that a specific non-terminal may lead to more than a single parse, we can provide a solution.

The first question to be answered is what to choose for the result of an ambiguous parser. We decide to return a list of all produced witnesses, and introduce a function *amb* which is used to label ambiguous non-terminals; the type of the parser that is returned by *amb* reflects that more than one result can be expected.

```
class Ambiguous p where
  amb :: p a → p [a]
```

For its implementation we take inspiration from the *parse* functions we have seen thus far. For history parsers we discovered that a grammar was ambiguous by simultaneously encountering a *Done* marker in the left and right operand of a call to *best*. So we model our *amb* implementation in the same way, and introduce a new marker End_h which becomes yet an extra alternative in our result type:

```
data Steps a where
     ...
     | End_h :: ([a], [a] → Steps r) → Steps (a, r) → Steps (a, r)
     ...
```

To recognise the end of a potentially ambiguous parse we insert an End_h mark in the result sequence, which indicates that at this position a parse for the ambiguous non-terminal was completed and we should continue with the call to the continuation. Since we want to evaluate the call to the common continuation only once we bind the current continuation k and the current state in the value of type $[a] \rightarrow Steps\ r$; the argument of this function will be the list of all witnesses recognised at the point corresponding to the occurrence of the End_h constructor in the sequence:

```
instance Ambiguous (P_h state) where
  amb (P_h p) =
    P_h (λk → removeEnd_h.p (λa st′ → End_h ([a], λas → k as st′) noAlts))
  noAlts = Fail
```

We thus postpone the call to the continuation itself. The second parameter of the End_h constructor represents the other parsing alternatives that branch within the ambiguous parser, but have not yet completed and thus contain and End_h marker further down the sequence.

All parses which reach their End_h constructor at the same point are collected in a common End_h constructor. We only provide the interesting alternatives in the new function *best*:

```
End_h (as, k_ st) l ‘best′‘ End_h (bs, _) r = End_h (as ++ bs, k_st)
                                                     (l ‘best‘ r)
End_h as l         ‘best′‘ r                 = End_h as (l ‘best‘ r)
l                  ‘best′‘ End_h bs r        = End_h bs (l ‘best‘ r)
```

If an ambiguous parser succeeds at least once it will return a sequence of *Step*'s which has the length of input consumed, followed by an End_h constructor which holds all the results and continuations of the parses that completed successfully at this point, and a sequence representing the best result for all other parses which were successful up-to this point. Note that all the continuations which are stored are the same by construction.

The expression *kas st′* binds the ingredients of the continuation; it can immediately be called once we have constructed the complete list containing the witnesses of all successful parses. The tricky work is done by the function *removeEnd*, which hunts down the result sequence in order to locate the End_h constructors, and to resume the *best* computation which was temporarily postponed until we had collected all successful parses with their common continuations.

```
removeEnd_h :: Steps (a, r) → Steps r
removeEnd_h (Fail                 ) = Fail
removeEnd_h (Step l               ) = Step (removeEnd_h l)
removeEnd_h (Apply f l            ) = error "not in history parsers"
removeEnd_h (End_h (as, k_st) r) = k_st as `best` removeEnd_h r
```

In the last alternative the function $removeEnd_h$ has forced the evaluation of all alternatives which are active up to this point. The result of the completed parsers here have been collected in the value *as*, which can now be passed to the function, thus resuming the parsing process at this point. Other parsers for the ambiguous non-terminal which have not completed yet are all represented by the second component. So the function $removeEnd_h$ still has to force further evaluation of these sequences, and remove the End_h constructor. The parsers terminating at this point of course still have to compete wih the still active parsers to finally reach a decision.

Without making this explicit we have gradually moved from a situation were the calls to the function *best* immediately construct a single sequence, to a situation where we have markers in the sequence which may be used to stop and start evaluation.

The situation for the online parsers is a bit different, since we want to keep as much of the online behaviour as possible. As an example we look at the following set of definitions, where the parser *r* is marked as an ambiguous parser:

```
p <++> q = (++) <$> p <*> q
a = (:[ ]) <$> pSym 'a'
a2 = a <++> a
a3 = a <++> a <++> a
r = amb (a <++> (a2 <++> a3 <|> a3 <++> a2)
```

In section 7 we will introduce error repair, which will guarantee that each parser always constructs a result sequence when forced to do so. This has as a consequence that if we access the value computed by an ambiguous parser we can be sure that this value has a length of at least 1, and thus we should be able to match, in the case of the parser *r* above, the resulting value successfully against the pattern ((a : _) : _) as soon as parsing has seen the first `'a'` in the input. As before we add yet another marker type to the type *Steps*:

```
data Steps a where
  ...
```

```
    End_f :: [Steps a] → Steps a → Steps a
    ...
```

We now give the code for the P_f case:

```
  instance Ambiguous (P_f state) where
    amb (P_f p) = P_f (λk inp → combineValues.removeEnd_f $
                                  p (λst → End_f [k st] noAlts) inp)
  removeEnd_f :: Steps r → Steps [r]
  removeEnd_f (Fail)              = Fail
  removeEnd_f (Step l)            = Step (removeEnd_f l)
  removeEnd_f (Apply f l)         = Apply (map' f) (removeEnd_f l)
  removeEnd_f (End_f (s : ss) r)  = Apply (:(map eval ss)) s
                                        `best`
                                    removeEnd_f r
  combineValues :: Steps [(a, r)] → Steps ([a], r)
  combineValues lar = Apply (λlar' → (map fst lar', snd (head lar'))) lar
  map' f ~(x : xs)    = f x : map f xs
```

The hard work is again done in the last alternative of $\mathit{removeEnd}_f$, where we apply the function *eval* to all the sequences. Fortunately this *eval* is again lazily evaluated, so not much work is done yet. The case of *Apply* is also interesting, since it covers the case of the first *a* in the example; the *map′ f* adds this value to all successful parses. We cannot use the normal *map* since this function is strict in the list constructor of its second argument, and we may already want to expose the call to *f* (e.g. to produce the value `'a':`) without proceeding with the match. The function *map′* exploits the fact that its list argument is guaranteed to be non-empty, as a result of the error correction to be introduced.

Finally we use the function *combineValues* to collect the values recognised by the ambiguous parser, and combine the result of this with the sequence produced by the continuation. It looks all very expensive, but lazy evaluation makes that a lot of work is actually not performed; especially the continuation will be evaluated only once, since the function *fst* does not force evaluation of the second component of its argument tuple.

5.3 Micro-steps

Besides the greedy parsing strategy which just looks at the next symbol in order to decide which alternative to choose, we sometimes want to give precedence to one parse over the other. An example of this is when we use the combinators to construct a scanner. The string `"name"` should be recognised as an identifier, whereas the string `"if"` should be recognised as a keyword, and this alternative thus has precedence over the interpretation as an identifier. We can easily get the desired effect by introducing an extra kind of step, which looses from *Step* but wins from *Fail*. The occurrence of such a step can be seen as an indication that a small penalty has to be paid for taking this alternative, but that we are happy to pay this price if no other alternatives are available. We extend the type *Steps* with a step *Micro* and add the alternatives:

```
(Micro l)   `best'` r@(Step _) = r
l@(Step _) `best'` (Micro _)   = l
(Micro l)   `best'` (Micro r)  = Micro (l `best'` r)
...
```

The only thing still to be done is to add a combinator which inserts this small step into a progress sequence:

```
class Micro p where
  micro :: p a → p a
instance Micro (P_f state) where
  micro (P_f p) = P_f (p.(Micro.))
```

The other instances follow a similar pattern. Of course there are endless variations possible here. One might add a small integer cost to the micro step, in order to describe even finer grained disambiguation strategies.

6 Embedding Parsers

With the introduction of the function *splitState* we have moved the responsibility for the scanning process, which converts the input into a stream of tokens, to the *state* type. Usually one is satisfied to have just a single way of scanning the input, but sometimes one may want to use a parser for one language as sub-parser in the parser for another language. An example of this is when one has a Haskell parser and wants to recognise a *String* value. Of course one could offload the recognition of string values to the tokeniser, but wouldn't it be nice if we could just call the parser for strings as a sub-parser, which uses single characters as its token type? A second example arises when one extend a language like Java with a sub-language like AspectJ, which again has Java as a sub-language. Normally this creates all kind of problems with the scanning process, but if we are able to switch from scanner type, many problems disappear.

In order to enable such an embedding we introduce the following class:

```
class Switch p where
  pSwitch :: (st1 → (st2, st2 → st1)) → p st2 a → p st1 a
```

It provides a new parser combinator *pSwitch* that can temporarily parse with a different state type *st2* by providing it with a splitting function which splits the original state of type *st1* into a state of type *st2* and a function which will convert the final value of type *st2* back into a value of type *st1*:

```
instance Switch P_h where
  pSwitch split (P_h p) = P_h (λk st1 → let (st2, b) = split st1
                                        in p (λst2' → k (b st2')) st2)
instance Switch P_f where
```

```
  pSwitch split (P_f p) = P_f (λk st1 → let (st2, b) = split st1
                                        in p (λst2′ → k (b st2′)) st2)
instance Switch P_m where
  pSwitch split (P_m (p, q)) = P_m (pSwitch split p, pSwitch split q)
```

Using the function *pSwitch* we can map the state to a different state and back; by providing different instances we can thus use different versions of *splitState*.

A subtle point to be addressed concerns the breadth-first strategy; if we have two alternatives working on the same piece of input, but are using different scanning strategies, the two alternatives may get out of sync by accepting a different number of tokens for the same piece of input. Although this may not be critical for the breadth-first process, it may spoil our recognition process for ambiguous parsers, which depend on the fact that when *End* markers meet the corresponding input positions are the same. We thus adapt the function *splitState* such that it not only returns the next token, but also an *Int* value indicating how much input was consumed. We also adapt the *Step* alternative to record the progress made:

```
type Progress = Int
data Steps a where
      ...
      Step :: Progress → Steps a
```

Of course also the function *best′* needs to be adapted too. We again only show the relevant changes:

```
Step n l ‘best′‘   Step m r
      | n ≡ m = Step n (l ‘best′‘ r)
      | n < m = Step n (l ‘best′‘ Step (m − n) r)
      | n > m = Step m (Step (n − m) l ‘best′‘ r)
```

The changes to all other functions, such as *eval*, are straightforward.

7 Error Reporting and Correcting

In this section we will address two issues: the reporting of errors and the automatic repair of errors, such that parsing can continue.

7.1 Error Reporting

An important feature of proper error reporting is an indication of the longest valid prefix of the input, and which symbols were expected at that point. We have seen already that the number of *Step* constructors provides the former. So we will focus on the latter. For this we change the *Fail* alternative of the *Steps* data type, in order to record symbols that were expected at the point of failure:

```
data Steps a where
   ...
  Fail :: [String] → Steps a
   ...
```

In the functions *pSym* we replace the occurrences of *Fail* with the expression *Fail* [*show a*], where *a* is the symbol we were looking for, i.e. the argument of *pSym*. The reason that we have chosen to represent the information as a collection of *String*'s makes it possible to combine *Fail* steps from parsers with different symbol types, which arises if we embed one parser into another.

In the function *best* we have to change the lines accordingly; the most interesting line is one where two failing alternatives are merged, which in the new situation becomes:

```
Fail ls `best` Fail rs = Fail (ls ++ rs)
```

An important question to be answered is how to deal with erroneous situations. The simplest approach is to have the function *eval* emit an *error* message, reporting the number of accepted tokens and the list of expected symbols. One might be tempted to change the function *eval* to return an *Either a* [*String*], returning either the evaluated result or the list of expected symbols. Keep in mind however that this would completely defeat all the work we did in order to get online results. If one is happy to use the history parsers this is however a perfect solution.

7.2 Error Repair

The situation becomes more interesting if we want to perform some form of error repair. We distinguish two actions we can perform on the input [18], *inserting* an expected symbol and *deleting* the current token. Ideally one would like to try all possible combinations of such actions, and continue parsing to see which combination leads to the least number of error messages. Unfortunately this soon becomes infeasible. If we encounter e.g. the expression `"2 4"` then it can be repaired by inserting a binary operator between both integers, and from the parser's point of view these are all equivalent, leading us to the situation we encountered in the case of the ambiguous non-terminals: a non-terminal may not be ambiguous, but its corrections may turn it into one which behaves like an ambiguous one. The approach we will take is to generate a collection of possible repairs, each with an associated cost, and then select the best one out of these, using a limited look-ahead.

To get an impression of the kind of repairs we will be implementing consider the following program:

```
test p inp = parse ((,) <$> p <*> pEnd) (listToStr inp)
```

The function *test* calls its parameter parser followed by a call to *pEnd* which returns the list of constructed errors and deleted possibly unconsumed input. The constructor (,) pairs the error messages with the result of the parser and the function *listToStr* convert a list of characters into an appropriate input stream type.

We define the following small parsers to be tested, including an ambiguous parser and a monadic parser to show the effects of the error correction:

```
a        = (λa → [a]) <$> pSym 'a'
b        = (λa → [a]) <$> pSym 'b'
p <++> q = (++) <$> p <*> q
a2       =     a <++> a
a3       =     a <++> a2
pMany p = (λa b → b + 1) <$> p <*> pMany p ≪|> pReturn 0
pCount 0 p = pReturn []
pCount n p = p <++> pCount (n − 1) p
```

Now we have three calls to the function *test*, all with erroneous inputs:

```
main = do print (test a2                                "bbab"     )
          print (test (do {l ← pMany a; pCount l b}) "aaacabbb")
          print (test (amb   (  (++) <$> a2 <*> a3
                             <|> (++) <$> a3 <*> a2)) "aaabaa")
```

Running the program will generate the following outputs, in which each result tuple contains the constructed witness and a list of error messages, each reporting the correcting action, the position in the input where it was performed, and the set of expected symbols:

```
("aa",              [  Deleted  'b' 0 ["'a'"],
                       Deleted  'b' 1 ["'a'"],
                       Deleted  'b' 3 ["'a'"],
                       Inserted 'a' 4 ["'a'"]])
["bbbb"],           [  Deleted  'c' 3 ["'a'","'b'"],
                       Inserted 'b' 8 ["'b'"]])
(["aaaaa"],         [  Deleted  'b' 3 ["'a'","'a'"]])
```

Before showing the new parser code we have to answer the question how we are going to communicate the repair steps. To allow for maximal flexibility we have decided to let the *state* keep track of the accumulated error messages, which can be retrieved (and reset) by the special parser *pErrors*. We also add an extra parser *pEnd* which is to be called as the last parser, and which deletes superfluous tokens at the end of the input:

```
class p `AsksFor` errors where
  pErrors :: p errors
  pEnd :: p errors
class Eof state where
  eof         :: state → Bool
  deleteAtEnd :: state → Maybe (Cost, state)
```

In order to cater for the most common case we introduce a new class *Stores*, which represents the retreival of errors, and extend the class *Provides* with two more functions which report the corrective actions tken to the *state*:

```
class state ‘Stores‘ errors where
  getErrors :: state → (errors, state)
class Provides state symbol token where
where
splitState  :: symbol → state →                          Maybe (token, state)
insertSym :: symbol → state →             Strings → Maybe (Cost, token, state)
deleteTok  :: token   → state → state → Strings → Maybe (Cost,          state)
```

The function *getErrors* returns the accumulated error messages and resets the maintained set. The function *insertSym* takes as argument the symbol to be inserted, the current state and a set of strings describing what was expected at this location. If the state decides that the symbol is acceptable for insertion, it returns the costs associated with the insertion, a token which should be used as the witness for the successful insertion action, and a new state. The function *deleteTok* takes as argument the token to be deleted, the old state which was passed to *splitState* –which may e.g. contain the position at which the token to be deleted is located–, and the new state that was returned from *splitState*. It returns the cost of the deletion, and the new state with the associated error message included.

In Fig. 9 we give a reference implementation which lifts, using *listToStr*, a list of tokens to a state which has the required interface and provides a stream of tokens. One fine point remains to be discussed, which is the commutativity of insert and delete actions. Inserting a symbol and then deleting the current token has the same effect as first deleting the token and then inserting a symbol. This is why the function *deleteTok* returns a *Maybe*; if it is called on a state into which just a symbol has been inserted it should return *Nothing*. The data type *Error* represents the error messages which are stored in the state, and *pos* maintains the current input position. Note also that the function *splitState* returns the extra integer, which represents how far the input state was "advanced"; here the value is always 1.

Given the defined interfaces we can now define the proper instances for the parser classes we have introduced. Since the code is quite similar we only give the version for P_f. The occurrence of the *Fail* constructor is a bit more involved than expected, and will be explained soon. The function *pErrors* uses *getErrors* to retrieve the error messages, which are inserted into the result sequence using a *push*. The function *pEnd* uses the recursive function *del* to remove any remaining tokens from the input, and to produce error messages for these deletions. Having reached the end of the input it retrieves all pending error messages and hands them over to the result:

```
instance (Eof state, Stores state errors) ⇒ AsksFor (P_f state) errors where
  pErrors = P_f (λk inp → let (errs, inp′) = getErrors inp
instance Eq a ⇒ Describes a a where
  eqSymTok = (≡)

data Error t s pos = Inserted s pos Strings
                   | Deleted t pos Strings
                   | DeletedAtEnd t
          deriving Show

data Str t = Str      { input    :: [t]
                      , msgs     :: [Error t t Int]
                      , pos      :: !Int
                      , deleteOk :: !Bool}

listToStr ls = Str ls [] 0 True

instance Provides (Str a) a where
  splitState _ (Str     []      _ _      _ ) = Nothing
  splitState _ (Str     (t : ts) msgs pos ok) = Just (t, Str ts msgs (pos + 1) True, 1)

instance Eof (Str a) where
  eof (Str i            _ _ _)           = null i
  deleteAtEnd    (Str (i : ii) msgs pos ok)
                  = Just (5, Str ii (msgs ++ [DeletedAtEnd i]) pos ok)
  deleteAtEnd    _
                  = Nothing

instance Corrects (Str a) a a where
  insertSym s (Str i    msgs pos       ok) exp
          = Just (5, s, Str i (msgs ++ [Inserted s pos exp]) pos False)
  deleteTok i (Str  ii   _          pos True)
            (Str    _   msgs pos' True)   exp
          = (Just (5, Str ii (msgs ++ [Deleted i pos' exp]) pos True))
  deleteTok _ _ _             _
          = Nothing

instance Stores (Str a) [Error a a Int] where
  getErrors (Str    inp msgs pos ok      ) = (msgs, Str inp [] pos ok)
```

Fig. 9. A reference implementation of *state*.

```
                              in push errs (k inp′))
pEnd = P_f (λk inp →
          let del inp =       case deleteAtEnd inp of
                              Nothing → let (errors, state) = getErrors inp
                                            in push errors (k state)
                              Just (i, inp′) → Fail [ ] [const (Just (i, del inp′))]
          in del inp
          )
```

Of course, if we want to base any decision about how to proceed with parsing on what errors have been produced thus far, the P_h version of *pErrors* should be used. If we just want to decide whether to proceed or not, the fact that results are produced online can be used too. If we find a non-empty error message embedded in the resulting value, we may decide not to inspect the rest of the returned value at all; and since we do not inpect it, parsing will not produce it either.

7.3 Repair strategies

As we have seen we have associated a cost with each repair step. In order to decide how to proceed we change the type *Step* once more. Since this will be the final version we present its complete definition here:

```
data Steps a where
Step   ::        Progress → Steps a                              → Steps a
Fail   ::        [String]  → [[String] → Maybe (Int, Steps a)] → Steps a
Apply  :: ∀ b.(b → a)  → Steps b                                  → Steps a
End_h  ::        [(a, [a] → Steps r)]  → Steps (a, r)           → Steps (a, r)
End_f  ::        [Steps a] → Steps a                              → Steps a
```

In the first component of the *fail* alternative the *String*'s describing the expected symbols are collected. The interesting part is the second component of the *Fail* alternative, which is a list of functions, each taking the list of expected symbols, and possibly returning a repair step containing an *Int* cost for this step and the result sequence corresponding to this path. The interesting alternative of *best′*, where all this information is collected, is:

```
Fail sl fl ‘best′‘ Fail sr fr = Fail (sl ++ sr) (fl ++ fr)
```

In figure Fig. 10 we give the final definition of *pSym* for the P_f case. The local functions *del* and *ins* take care of the deletion of the current token and the insertion of the expected symbol, and are returned where appropriate if recognition of the expected symbol a fails.

In the *best′* alternative just given we see that the function stops working and just collects information about how to proceed. Now it becomes the task of the function *eval* to start the suspended parsing process:

```
eval (Fail ss fs) = eval (getCheapest 3 [c | f ← fs, let Just c = f ss])
```

```
instance (Show symbol, Describes symbol token, Corrects state symbol token)
            ⇒ Symbol (P_f state) symbol token where
  pSym a = P_f (
    let p = λk inp →
      let ins ex =          case insertSym a inp ex of
                              Just (c_i, v, st_i) →    Just (c_i, push v (k st_i))
                              Nothing                → Nothing
          del s ss ex
                           = case deleteTok s ss inp ex of
                              Just (c_d, st_d) → Just (c_d, p k st_d)
                              Nothing →           Nothing
      in case splitState a inp of
          Just (s, ss, pr) → if a `eqSymTok` s
                             then    Step pr (push s (k ss))
                             else    Fail [show a] [ins, del s ss]
          Nothing →                  Fail [show a] [ins]
    in p)
```

Fig. 10. The definition of *pSym* for the P_f case.

Once *eval* is called we know that all expected symbols and all information how to proceed has been merged into a single *Fail* constructor. So we can construct all possible ways how to proceed by applying the elements from *ls* to the set of expected symbols *ss*, and selecting those cases where actually something can be repaired. The returned progress sequences themselves of course can contain further *Fail* constructors, and thus each alternative actually represents a tree of ways of how to proceed; the branches of such a tree are either *Step*'s with which we associate cost 0, or further repair steps each with its own costs. For each tree we compute the cheapest path up-to n steps away from the root using the function *traverse*, and use the result to select the progress sequence containing the path with the lowest accumulated cost. The first parameter of *traverse* is the number of tree levels still to be inspected, the second argument the tree, the third parameter the accumulated costs from the root up-to the current node, and the last parameter the best value found for a tree thus far, which is used to prune the search process.

```
getCheapest :: Int → [(Int, Steps a)] → Steps a
getCheapest _ [] = error "no correcting alternative found"
getCheapest n l = snd $ foldr (λ(w, ll) btf@(c, l)
                                 → if w < c
                                   then let new = (traverse n ll w c)
                                        in if new < c then (new, ll) else btf
                                   else btf
                                ) (maxBound, error "getCheapest") l
traverse :: Int → Steps a → Int → Int → Int
traverse 0 _              =  λv c → v
traverse n (Step ps l)    =  traverse (n − 1) l
traverse n (Apply _ l)      =  traverse n l
traverse n (Fail m m2ls) =
   λv c → foldr (λ(w, l) c′ →  if v + w < c′
                                then traverse (n − 1) l (v + w) c′
                                else c′
               ) c (catMaybes $ map ($m) m2ls)
traverse n (End_h ((a, lf) : _) r) = traverse n (lf [a] `best` removeEnd_h r)
traverse n (End_f (l : _)        r) = traverse n (l `best` r)
```

8 An Idiomatic Interface

McBride and Paterson [10] investigate the *Applicative* interface we have been using throughout this tutorial. Since this extension of the pattern of sequential composition is so common they propose an intriguing use of functional dependencies to enable a very elegant way of writing applicative expressions. Here we shortly re-introduce the idea, and give a specialised version for the type *Parser* we introduced for the basic library.

Looking at the examples of parsers written with the applicative interface we see that if we want to inject a function into the result then we will always do this with a *pReturn*, and if we recognise a keyword then we always throw away the result. Hence the question arises whether we can use the types of the components of the right hand side of a parser to decide how to incorporate it into the result. The overall aim of this exercise is that we will be able to replace an expression like:

```
choose < $ pSyms "if" <*> pExpr <* pSyms "then" <*> pExpr
                                <* pSyms "else" <*> pExpr
```

by the much shorter expression:

```
start choose "if" pExpr "then" pExpr "else" pExpr stop
```

or by nicely formatting the start and stop tokens as a pair of brackets by:

```
[: choose "if" pExpr "then" pExpr "else" pExpr :]
```

The core idea of the trick lies in the function *idiomatic* which takes two arguments: an accumulating argument in which it constructs a parser, and the next element from the expression. Based on the type of this next element we decide what to do with it: if it is a parser too then we combine it with the parser constructed thus far by using sequential composition, and if it is a *String* then we build a keyword parser out of it which we combine in such a way with the thus far constructed parser that the witness is thrown away. We implement the choice based on the type by defining a collection of suitable instances for the class *Idiomatic*:

```
class Idiomatic f g | g → f where
  idiomatic :: Parser Char f → g
```

We start by discussing the standard case:

```
instance Idiomatic f r ⇒ Idiomatic (a → f) (Parser Char a → r) where
  idiomatic isf is = idiomatic (isf <*> is)
```

which is to be read as follows: if the next element in the sequence is a parser returning a witness of type a, and the parser we have constructed thus far expects a value of that type a to build a parser of type f, and we know how to combine the rest of g with this parser of type f, then we combine the accumulated parser recognising a value of type $a \to f$ and the argument parser recognising an a, and call the function *idiomatic* available from the context to consume further elements from the expression.

If we encounter the *stop* marker, we return the accumulated parser. For this marker we introduce a special type *Stop*, and declare an instance which recognises this *Stop* and returns the accumulated parser.

```
data Stop = Stop
stop = Stop
instance Idiomatic x (Stop → Parser Char x) where
  idiomatic ix Stop = ix
```

Now let us assume that the next element in the input is a function instead of a parser. In this case the *Parser Char a* in the previous instance declaration is replaced by a function of some type $a \to b$, and we expect our thus far constructed parser to accept such a value. Hence we get:

```
instance Idiomatic f g ⇒ Idiomatic ((a → b) → f) ((a → b) → g) where
  idiomatic isf a2b = idiomatic (isf <*> pReturn a2b)
```

Once we have this instance it is now easy to define the function *start*. Since we can prefix every parser with a *id*<$> fragment, we can define *start* as the initialisation of the accumulated parser by the parser which always succeeds with an *id*:

```
start :: ∀ a g.(Idiomatic (a → a) g) ⇒ g
start = idiomatic (pReturn id)
```

Finally we can provide extra instances at will, as long as we do not give more than one for a specific type. Otherwise we would get an overloading ambiguity. As an example we define two further cases, one for recognising a keyword and once for recognising a single character:

```
instance Idiomatic f g ⇒ Idiomatic f (String → g) where
  idiomatic isf str = idiomatic (isf <* pKey str)
instance Idiomatic f g ⇒ Idiomatic f (Char   → g) where
  idiomatic isf c   = idiomatic (isf <* pSym c)
```

9 Further Extensions

In the previous sections we have developed a library which provides a lot of basic functionality. Unfortunately space restrictions prevent us from describing many more extensions to the library in detail, so we will sketch them here. Most of them are efficiency improvements, but we will also show an example of how to use the library to dynamically generate large grammars, thus providing solutions to problems which are infeasible when done by traditional means, such as parser generators.

9.1 Recognisers

In the basic library we had operators which discarded part of the recognised result since it was not needed for constructing the final witness; typical examples of this are e.g. recognised keywords, separating symbols such as commas and semicolons, and bracketing symbols. The only reason for their presence in the input is to make the program readable and unambiguously parseable.

Of course it is not such a great idea to first perform a lot of work in constructing the result, only having to even more work to get rid of it again. Fortunately we have already introduced the *recognisers* which can be combined with the other types of parsers P_h, P_f and P_m. We introduce yet another class:

```
class Applicative p ⇒ ExtApplicative p st where
  (<*) :: p a      → R st b → p a
  (*>) :: R st b → p a      → p a
  (<$) :: a        → R st b → p a
```

The instances of this class again follow the common pattern. We only give the implementation for P_h:

```
instance ExtApplicative (P_h st) st where
  P_h p <* R r = P_h (p.(r.) )
  R r *> P_h p = P_h (r.p    )
  f <$ R r     = P_h (r.($f))
```

9.2 Parsing Permutation Phrases

A nice example of the power of parser combinators is when we want to recognise a sequence of elements of different type, in which the order in which they appear in the input does not matter; examples of such a situation are in the recognition of a BibTeX entry or the attribute declarations allowed at a specific node in an XML-tree. In [1] we show how to proceed in such a case, so here we only sketch the idea which heavily depends on lazy evaluation.

We start out by building a data structure which represents all possible permutations of the parsers for the individual elements to be recognised. This structure is a tree, in which each path from the root to a leaf represents one of the possible permutations. From this tree we generate a parser, which initially is prepared to accept any of the elements; after having recognised the first element it continues to recognise a permutation of the remaining elements, as described by the appropriate subtree. Since the tree describing all the permutations and the parser corresponding to it are constructed lazily, only the parsers corresponding to a permutation actually occurring in the input will be generated. All the chosen alternative has to do in the end is to put the elements in some canonical order.

9.3 Look-ahead computations

Conventional parser generators analyse the grammar, and based on the results of this analysis try to build efficient recognisers. In an earlier paper [15] we have shown how the computation of *first sets*, as known from the theory about $LL(1)$ grammar analysis, can be performed for grammars described by combinator parsers. We plan to add such an analysis to our parsers too, thus speeding up the parsing process considerably in cases where we have to deal with a larger number of alternatives.

A subtle point here is the question how to deal with monadic parsers. As we described in [15] the static analysis does not go well with monadic computations, since in that case we dynamically build new parses based on the input produced thus far: the whole idea of a static analysis is that it is static. This observation has lead John Hughes to propose *arrows* for dealing with such situations [7]. It is only recently that we realised that, although our arguments still hold in general, they do not apply to the case of the $LL(1)$ analysis. If we want to compute the symbols which can be recognised as the first symbol by a parser of the form $p \ggg= q$ then we are only interested in the starting symbols of the right hand side if the left hand side can recognise the empty string; the good news is that in that case we statically know what value will be returned as a witness, and can pass this value on to q, and analyse the result of this call statically too. Unfortunately we will have to take special precautions in case the left hand side operator contains a call to *pErrors* in one of the empty derivations, since then it is no longer true that the witness of this alternative can be determined statically.

10 Conclusions

We have come to end of a fairly long tutorial, which we hope to extend in the future with sections describing the yet missing abstract interpretations. We hope nevertheless that the reader has gained a good insight in the possibilities of using Haskell as a tool for embedding domain specific languages. There are a few final remarks we should like to make.

In the first place we claim that the library we have developed can be used outside the context of parsing. Basically we have set up a very general infrastructure for describing search algorithms, in which a a tree is generated representing possible solutions. Our combinators can be used for building such trees and searching such trees for possible solutions in a breadth-first way.

In the second place the library we have described is by far not the only one existing. Many different (Haskell) libraries are floating around, some more mature than others, some more dangerous to use than others, some resulting in faster parsers, etc. One of the most used libraries is *Parsec*, originally constructed by Daan Leijen, which gained its popularity by packaged with the distribution of the GHC compiler. The library distinguishes itself from our approach in that the underlying technique is the more conventional back-tracking technique, as described in the first part of our tutorial. In order to alleviate some of the mentioned disadvantages of that approach, the programmer has the possibility to commit the search process at specific points, thus cutting away branches from the search tree. Although this technique can be very effective it is also more dangerous: unintentionally branches which should remain alive may be pruned away. The programmer really has to be aware of how his grammar is parsed in order to know where to safely put the annotations. But if he knows what he is doing, fast parsers can be constructed. Another simplifying aspect is that *Parsec* just stops if it cannot make further progress; a single error message is produced, describing what was expected at the farthest point reached.

A relatively new library was constructed by Malcolm Wallace [19], which contains many of the aspects we are dealing with: building results online, and combing a monadic interface with an applicative one. It does however not perform error correction.

Another library which implements a breadth-first strategy are Koen Claessen's parallel parsers [3], which are currently being used in the implementation of the GHC *read* functions. They are based on a rewriting process, and as a result do not lend themselves well to an optimising implementation.

Concluding we may say that parser combinators are providing an ever lasting source of inspiration for research into Haskell programming patterns which has given us a lot of insight in how to implement Embedded Domain Specific Languages in Haskell.

Acknowledgements I thank current and past members of the Software Technology group at Utrecht University for commenting on earlier versions of this paper, and for trying out the library described here. I want to thank Alesya Sheremet for working out some details of the monadic implementation, and the anonymous referee for his/her comments, and Magnus Carlsson for many suggestions for improving the code.

A Driver function for pocket calculators

The driver function for the pocket calculators:

```
run :: (Show t) ⇒ Parser Char t → String → IO ()
run p c =
   do putStrLn ("Give an expression like: "
               ⧺ c ⧺ " or (q) to quit")
      inp ← getLine
      case inp of
        "q" → return ()
        _    → do putStrLn (case unP p (filter (≢ ' ') inp) of
                               ((v, "") : _) → "Result is: " ⧺ show v
                               _             → "Incorrect input")
      run p c
```

References

1. Arthur I. Baars, Andres Löh, and S. Doaitse Swierstra. Parsing permutation phrases. *J. Funct. Program.*, 14(6):635–646, 2004.
2. W.H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.
3. Koen Claessen. Parallel parsing processes. *Journal of Functional Programming*, 14(6):741–757, 2004.
4. J. Fokker. Functional parsers. In J.T. Jeuring and H.J.M. Meijer, editors, *Advanced Functional Programming, First International Spring School*, number 925 in LNCS, pages 1–23, 1995.
5. Bastiaan Heeren. *Top Quality Type Error Messages*. PhD thesis, Utrecht University, 2005.
6. Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.
7. John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.
8. Graham Hutton and Erik Meijer. Monadic parsing in haskell. *J. Funct. Program.*, 8(4):437–444, 1998.
9. Daan Leijen. Parsec, a fast combinator parser. Technical Report UU-CS-2001-26, Institute of Information and Computing Sciences, Utrecht University, 2001.
10. Conor Mcbride and R. O. S. S. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(01):1–13, 2007.
11. Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, April 2003. http://www.haskell.org/report.
12. Niklas Röjemo. *Garbage collection and memory efficiency in lazy functional languages*. PhD thesis, Chalmers University of Technology, 1995.
13. S. Doaitse Swierstra, Arthur Baars, Andres Löh, and Arie Middelkoop. uuag - utrecht university attribute grammar system.
14. S.D. Swierstra, P.R. Azero Alocer, and J. Saraiava. Designing and implementing combinator languages. In S. D. Swierstra, Pedro Henriques, and José Oliveira, editors, *Advanced Functional Programming, Third International School, AFP'98*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.
15. S.D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.

16. Marcos Viera, S. Doaitse Swierstra, and Eelco Lempsink. Haskell, do you read me?: constructing and composing efficient top-down parsers at runtime. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 63–74, New York, NY, USA, 2008. ACM.
17. 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.
18. Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.
19. Malcolm Wallace. Partial parsing: Combining choice with commitment. In *Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers*, pages 93–110, Berlin, Heidelberg, 2008. Springer-Verlag.