ABSTRACT
In this paper we develop, in a stepwise fashion, a set of parser combinators that may be used in the construction of deterministic, error correcting parsers. The only restriction on the grammar is that it should not be left recursive. The set of grammars that may be described are the \( LL(\infty) \) grammars. In the development extensive use is made of lazy evaluation, and parsers are constructed such that they “analyze themselves”. We claim that our new combinators are adequate for the construction of large parsers to be used in compilers to be used by real users.

Categories and Subject Descriptors
D.1.1 [Programming techniques]: Applicative (functional) programming; D.3.3 [Programming languages]: Language Constructs and features; D.3.4 [Programming languages]: Processors—parsing, translator writing systems and compiler generators; D.3.4 [Programming languages]: Language Classification—Applicative (functional) languages

General Terms
parser combinators, error correction, deterministic, partial evaluation, parser generators, program analysis, lazy evaluation, polymorphism

1. INTRODUCTION
Parsers constructed with conventional parser combinators have two disadvantages: the larger the grammar, the slower the parsing gets and when the input is not a sentence of the language they return an empty set of successful parses, with no clue about where or what error was encountered. In [6] we presented a set of parser combinators that did not exhibit such shortcomings, provided the grammar was \( LL(1) \); this property makes it possible to decide what direction to take during top-down parsing by looking at the next symbol in the input.

For many languages grammars may be constructed, by performing a process called left factoring, that have this property (maybe with using some post processing), but unfortunately the resulting grammars often bear little resemblance to what the language designer had in mind, and as a consequence extending such transformed grammars with functions for processing the recognized result becomes cumbersome and the elegance that one expects when using parser combinators is lost.

In order to alleviate this problem we set out to extend our previous combinators in such a way that we would be able to use a longer look-ahead sequences. This resulted in a completely different set of combinators, for which the \( LL(1) \) restriction does not hold. The only requirement we still impose on a grammar is that it is neither directly nor indirectly left-recursive. This can easily be achieved by the use of appropriate chain-combinators, which in most cases express the intention of the language designer even better than the explicit recursive formulations.

In section 2 we will recapitulate the conventional combinators (in the non-monadic formulation) and explain where the problems mentioned above arise. In section 3 we will provide a different approach, that makes it possible to add error correction to the basic combinators. The resulting combinators are still very short and might be used for small grammars. In section 4 we will show how to include the computation of look-ahead information in the combinators. In this process we try to minimize the number of times a symbol is inspected. Only in those cases where information of the follower set is needed, this may happen. We do not consider this unreasonable since precomputing this kind of information is in general impossible in the kind of programs we write, and furthermore this is also the kind of information that may make the usage of memory explode. In this sense we consider our implementation to be optimal. The final implementation has been used in constructing some fairly large parsers, that perform at quite reasonable efficiency, especially when compared with other combinator based versions. Especially the costs of being able to correct errors is negligible, especially when errors are absent. So one only pays for the extra features, when they are used. Finally we present some further extensions (section 5 and conclusions (section 6).

2. CONVENTIONAL PARSER COMBINATORS
There exist many different implementations of the basic parser combinators; some use basic functions ([2]), whereas others make use of a monadic formulation (in the Haskell distribution). In figure 1 we present the basic interface of the combinators, and a straightforward implementation. We will define new implementations and new types, but will do this in such a way that already constructed parsers can use these new definitions with minimal changes. Parsers constructed using these combinators in essence perform a depth-first search through the set of all possible parse trees, and return all possible ways in which to succeed. Note that we have taken a really “functional” approach when constructing the result of a sequential composition. Instead of constructing a value of the more complicated type \((a, b)\) out of the two more simpler types \(a\) and \(b\), we have chosen to construct a value of a simpler type \(b\) out of the more complicated types \(a \rightarrow b\) and \(a\). Based on these basic combinators more complicated combinators can be constructed. For an examples of the use of such combinators, and the description of more complicated ones expressed in these basic ones see [2, 3, 4, 5].

The main shortcoming of this set is that when the input cannot be parsed, this is signalled by the parser returning the empty list, without a single indication about where things are most likely to be wrong. A consequence the combinators in this form are unusable for any input of significant size. It is however a straightforward exercise to change the definitions in such a way that track is kept of the furthest point in the input reached during parsing. But from modern compilers we expect more: the compiler should correct simple typing mistakes such as adding missing closing brackets, inserting missing semicolons etc. Furthermore it should do so with proper error messages about what changes were made to the input in order to convert it into a proper sentence of the language.

A second disadvantage of parsers constructed in this way is that when the grammar becomes larger, and has many alternatives, all alternatives are tried in a sequential way at each branching point, thus necessitating a large number of symbol comparisons. This effect becomes worse when a naive user uses the combinators to describe very large grammars as in:

\[
\text{fold1 (<|>) (map pSym [1..1000])}
\]

Here on the average 500 comparisons are needed in order to recognize a symbol.

A further source of potential inefficiency is caused by non-determinism. When many alternatives recognize strings with a common prefix, this prefix may be parses several times, with usually only one of those alternatives resulting in a success. So for highly “non-deterministic” grammars the price paid may be high. Although it is quite well known how to construct deterministic automata out of non-deterministic ones this knowledge is not used in this implementation nor easily incorporated.

We will now start our description of a new implementation that solves all of the problems mentioned.

### 3. ERROR CORRECTION

#### 3.1 Continuation based parsing

If we extend the combinators from the previous section to keep track of the farthest point in the input that was reached, the parser returns that value after backtracking has completed. Unfortunately we have by then lost all state information that might enable us to decide on the proper error correcting steps to be taken and to continue parsing from there. So our first step will be to convert our combinators into a form that allows us to work on all possible alternatives concurrently, thus changing from a depth-first to a breadth-first exploration of the search space. As a first step
we introduce the combinators in figure 2, which are constructed using a continuation based style. As we will see this will make it possible to return information about how the parsing process is progressing before we have actually finished a complete parse. For the time being we ignore the result to be computed, and simply return a boolean value indicating whether the sentence belongs to the language or not. Later we will include the semantic processing into the functions. The parameter \( r \) represents the rest of the parsing process after the current parser succeeds, and can be seen as a stack of remaining symbols from the right hand sides of productions, against which the remaining part of the input is to be matched. Note also that we have defined a function \( \text{parse} \) that must be used to start the parsing process. It gets passed the function \( \text{null} \) that checks whether the input has indeed been totally consumed when parsing has finished. The introduction of this function \( \text{parse} \) is the only extension we will make to the interface.¹

3.2 Parsing histories
An essential design decision now is not just to return a result, but to combine this with the parsing history, consisting of a sequence of parsing steps, that led to this result. We consider two different kind of parsing steps:

- **Ok-steps**, that represent the successful recognition of an input symbol
- **Fail-steps**, that represent a corrective step during the parsing process; such a step will either correspond to the insertion into or the deletion of a symbol from the input stream

```
data Steps v = Ok (Steps v)  
| Fail (Steps v)  
| Stop v

getresult (Ok l) = getresult l
getresult (Fail l) = getresult l
getresult (Stop v) = v
```

For the combination of the result and its parsing history we do not simply take a cartesian product, since then we can only construct this pair after having reached the end of the parsing process and thus having constructed the final result. Instead we introduce a more intricate data type, that allows us to start constructing the required information before parsing has been completed. Ideally one would like to select the result with the fewest \text{Fail}-ing steps, i.e. that sequence that represents a minimal editing distance to the input. Unfortunately this will be a very costly operation, since it implies that at all possible positions in the input all possible corrective steps have to be taken. Suppose e.g. that an unmatched \text{then}-symbol is encountered, and that we want to find the optimal place to insert the missing \text{if}-symbol. In this case there may be many points where this might be inserted, and all those points might be equivalent in a certain sense.

In order to cope with this problem we take a greedy approach, giving preference to the longest parsing without error correcting steps. So we define an order between the parsing results, based on longest successful prefixes of \text{Ok}-steps:

```
(OK l) 'best' (Ok r) = Ok (l 'best' r)
(Fail l) 'best' (Fail r) = Fail (l 'best' r)
l@(Ok _) 'best' (Fail _) = l
(Fail _) 'best' r@(Ok _) = r
l@(Stop _) 'best' _ = l
_ 'best' r@(Stop _) = r
```

There is a very important observation to be made here: when there is no preference between two sequences based on the first step, we may postpone the decision about which operand to return, but still return information about the first step in the selected result, as shown in the first two patterns in the definition of the function \text{best} above!

3.3 Error correcting steps
Let us now discuss the possible error correcting steps. We have to take such a step when the next symbol in the input is different from the symbol we expect or when we expect at least one more symbol and the input is exhausted. In this case we have two possible correcting steps:

- pretend that the symbol was there anyway, which is equivalent to inserting it in the input stream
- delete the current input symbol, and try again to see whether the expected symbol is present
In both of these cases we have to report a \texttt{Fail}\texttt{-step}. If we add this error recovery to the combinators defined before, we get the code in figure 3. Note that if we have any input left at the end of the parsing process this input is deleted, resulting in a number of failing steps \((\text{foldr} \ (\text{const} \ \text{Fail}) \ (\text{Stop} \ \text{True}))\). The operator \(||\) that was used before to find out whether at a branching point at least one of the alternatives finally had led to success has been replace by the \texttt{best} operator that selects the “best” result. It is here that the change from a depth first approach to a breadth-first approach is made: the function \(||\) returns only a result after at least its first operand has been evaluated, whereas the function \texttt{best} returns its result in a stepwise fashion. It is the function \texttt{getresult} at the top level that is actually driving the computation by asking for the constructors used at the head of the returned \texttt{Steps} value.

### 3.4 Computing more interesting results

Note also that the combinators as defined here are quite useless, since now we have made sure that we will always find a complete parse and thus the result of the parsing will always be \texttt{True}. In order to make the parsers more interesting we now have to add two more things:

1. compute a kind of result similar to the original combinators
2. a reporting facility for generating error messages, indicating what corrective steps were taken.

Both these components can be handled in a similar fashion, viz. by accumulating the results computed thus far in an extra argument to the parsing functions.

#### 3.4.1 Computing a result

Top down parsers maintain two kinds of stacks:

- one for keeping track of what still is to be recognized (here represented by the continuation)
- one for storing the right hand side elements that have been recognized, but cannot yet be used in a reduction

Note that our parsers (or grammars if you prefer), although this may not be realized at first sight, are in a normal form, in which each right hand side has length 2 since each occurrence of a \(<\leftrightarrow\) combinator introduces an (implicit) non-terminal. If the length of a right hand side is larger than 2, it is the left-associativity of \(<\leftrightarrow\) that makes this happen.

We decide to represent the stack of recognized left arguments to \(<\leftrightarrow\>-occurrences with a function too, since it may contain elements of very different types. The invariant we maintain is that each parser takes as an extra parameter a function that has to be applied to its parsing result, and the result of that application is to be passed on to the continuation. So the type of the reduced item stack is:

\[
\text{type Stack a b} = a \rightarrow b
\]

The interesting combinator here is the one taking care of sequential composition, which now, with the new definition of \texttt{Parser}\textsuperscript{3}, becomes:

\[
\begin{align*}
\text{type Parser s a =} \forall b \text{ result .} \\
& \quad \forall s \text{ result .} \quad \text{the future} \\
& \quad \forall a \rightarrow b \quad \text{the history} \\
& \quad \forall [s] \quad \text{the input sequence} \\
& \quad \forall \text{Steps result} \\
& ((p \leftrightarrow q) \text{ r stack input = p (q r) (stack.) input}
\end{align*}
\]

When \(p v\) is the value computed by the parser \(p\) and \(q v\) the value computed by the parser \(q\), the value passed on to \(r\) will be:

\[
(((\text{stack .}) \ p v) \ q v) = (\text{stack.} \ pv) \ q v = \text{stack} \ (pv \ qv)
\]

\textsuperscript{3}Note that the type is currently not correct Haskell.
have been included). The form of a data structure, which we make an instance of their own native language) we return the error messages in a way that is exactly what we would expect. If the stack was of the form stack', then the resulting value is stack'. (qv), corresponding to a push of the value on the stack. Otherwise it has been paired with its left neighbor, corresponding to a push operation. Finally we have to adapt he function parse once more, so it incorporates the constructed result in the returned result, and it initializes the stack (id):

```
parse p = getresult
  (p \ v inp -> foldr (const Fail) (Stop v) inp)
  id
)
```

We will not give the new versions of the other combinators here, since they will show up almost in the same form in figure 4.

3.4.2 Error reporting

Note that at the point where we introduce the error correcting steps we cannot be sure whether these corrections will actually be on the finally chosen path in the search tree, and so we cannot directly add the error messages to the result: keep in mind that it is a fundamental property of our strategy that we may return information about the result without actually having made a choice yet. Including error messages with the Ok and Fail constructors would enforce us to prematurely take a decision about which path to choose. So we decide to pass the error messages in an accumulating parameter too, only to be included in the result at the end of the parsing process. In order to make it possible for users to generate their own error messages (say in their own native language) we return the error messages in the form of a data structure, which we make an instance of Show (see figure 4, in which also the previous modifications have been included). The Maybe s component is introduced to store information about where the corrective step was taken, with the Nothing alternative representing the end of the input sequence.

4. IMPROVING EFFICIENCY

In the previous section we have solved the first of the mentioned problems, i.e. we made sure that a result will always be returned, together with a message about what error correcting steps were taken. In this section we will solve the two remaining problems, that both have to do with the low efficiency, in one sweep.

Thus far the parsing functions were all defined as functions that do the parsing, but about which we cannot easily get any further information. An example of such information may be the set of symbols that may be recognized as first symbols by a parser, or whether the parser can recognize an empty sequence. Since we cannot obtain this information from the parser itself we decide to compute this information separately, and to tuple this information with a parser that was constructed using this information.

To see what such kind of information might look like we first introduce yet another formulation of the basic combinators, which in this case do not construct a parser but a trie-structure representing all possible sentences in the language of the represented grammar (see figure 5). For a while we forget again about computing results and error messages. The case /: nodes describe nodes that are both Choice node (contained in the left operand) and an End node, contained in its right operand. We could have encoded this using a slightly different structure, but this would have resulted in a more elaborate program text later. Now notice that for the following grammar

```
Combination
```

a Choice node is constructed with two outgoing branches that are labelled with ‘a’, so it is an obvious optimization to merge these two branches into a single one, thus removing the backtracking from the parsing process: we replace the expression Choice (as++bs) with the expression combine as bs, where combine is defined as:

```
combine xss@(x@(s,ct):xs) yss@(y@(s',ct'):ys)
  = case compare s s' of
      LT -> x:combine xs yss
      GT -> y:combine xss ys
      EQ -> (a, ct <|> ct'):combine xs ys
combine [] cs' = cs'
combine cs [] = cs
```

Note that for:

```
x = asb <|> asc :: Sents
asb = pSym 'a' <> asb <|> pSym 'b'
asb = pSym 'a' <> asb <|> pSym 'c'
```

this may be a non terminating process. Fortunately lazy evaluation takes care of this problem, and the merging process only proceeds far enough to recognize the current sentence. A shortcoming of this approach however is that we perform a tremendous amount of copying, because of the way sequential composition has been modelled. We do not only use the structure to make the decision process deterministic, but also to represent the stack of symbols still to be recognized.

We now compare the two different approaches taken:

- continuation based parsers that compute a value, and work on all alternatives in parallel
- the parser that interprets a trie data structure which makes that each symbol in the input is inspected only once

In our final solution we will try to merge these two approaches. We will use a data structure similar to Sents to decide how to proceed with parsing, and use the continuation based parsers to actually accept the symbols. Since the
data Errors s = Deleted s (Maybe s) (Errors s)
                   | Inserted s (Maybe s) (Errors s)

type Errs s = (Errors s -> Errors s)

instance Show s => Show (Errors s) where
    show (Deleted s w e) = "\ndeleted " ++ show s ++ " before " ++ location w ++ show e
    show (Inserted s w e) = "\ninserted " ++ show s ++ " before " ++ location w ++ show e
    show (NotUsed Nothing) = ""
    show (NotUsed w) = "\nsymbols starting at " ++ location w ++ " were discarded "

location Nothing = " end of input"
location (Just s) = show s

position [] = Nothing
position (s:_:_) = Just s

p <|> q = \ r stack errors input -> p r stack errors input 'best' q r stack errors input
p <*> q = \ r stack errors input -> p (q r) (stack . errors input)
pSym a
  = let pr = \ r st e input ->
    case input
      of [] -> Fail (r (st a) (e.Inserted a Nothing) input)
          (s:ss) ->
          if s == a then Ok (r (st s) e s ss)
          else Fail ((pr r st (e.Deleted s (position ss)) ss)
                            'best'
                           (r (st a) (e.Inserted a (Just s)) input)
    in pr

pSucceed v = \ r stack errors input -> r (stack v) errors input

parse p
  = getresult ( p (\ v errors inp -> foldr (const Fail) (Stop (v,errors)) inp) id)

Figure 4: Correcting and error reporting combinators
information represented in the new data structure closely resembles the information stored in a state of an LR(0) automaton, we will use that terminology. So instead of building the complete \texttt{Sents} structure, we will construct a similar structure that may be used to select the parser to continue with.

Before proceeding, let us consider the following grammar fragment:

\begin{verbatim}
\texttt{pSucceed (\\{ x y -> x \} <> pSym 'a' <> pSym 'b'<>} \texttt{)}
\texttt{pSucceed (\\{ x y -> x \} <> pSym 'a' <> pSym 'c'<>} \texttt{)}
\end{verbatim}

The problem that arises is what to do with the parsers preceding the respective \texttt{pSym 'a'} occurrences: we can only decide which one to take after a symbol \texttt{'b'} or \texttt{'c'} has been encountered, because only then we will have a definite answer about which alternative to take. This problem is solved by pushing such actions inside the trie structure to a point where the merging of the different alternatives has stopped: at that point we are again working on a single alternative and can safely perform the postponed computations.

We now discuss the full version of the structure we have defined to do the book keeping of our look-ahead information (see the definition in figure 6), and the actions to be taken once a decision has been made. The \texttt{p} components of the \texttt{Look} structure are the parsing functions that actually accept symbols. We will discuss the different alternative in reverse order:

- \texttt{Found p (Look s p)}: this alternative indicates that the only possible parser that applies at this point is the parser \texttt{p}. It corresponds to a non-merged node in the trie structure. The \texttt{(Look s p)} part is used when the structure is merged with further alternatives and the uniqueness of the parser \texttt{p} no longer holds. This is the case when that other alternative contains a path similar to the path that leads to this \texttt{Found} node. As one can see in the function \texttt{merge ch} this \texttt{Found} constructor is removed when the structure is merged with other alternatives.

- \texttt{Reduce p}: this indicates that in using the look ahead information we have used all the information in the right hand side of a production (and have reached a \texttt{reduce} state in LR terminology), and that the parser \texttt{p} is the function to be called.

- \texttt{(|:)}: this corresponds to the situation where we either may continue with using further symbols to make a decision, or we will have to use information about the followers of this nonterminal. This will be the only place where we continue with a possibly non-deterministic parsing process. It corresponds to a shift-reduce conflict in an LR(0) automaton.

- \texttt{Shift p [(s, Look s p)]}: this corresponds to a shift state, in which we need at least one more symbol in order to be able to decide what to do. The parser \texttt{p} contained in this alternative is the error correcting parser to be called when the next input symbol is not a key in the next table of this \texttt{Shift} point, and thus no symbol can be shifted without taking corrective steps.

Before giving the description about how to construct such a Look-ahead structure we will first explain how they are going to be used (see figure 7).
data Look s p = Shift p [(s, Look s p)] -- like Choice
| (Look s p) :|: (Look s p) -- Shift/Reduce
| Reduce p -- End
| Found p (Look s p) -- Uniquely determined

cata_Look (sem_Shift, sem_Or, sem_Reduce, sem_Found)
= let r = \ c -> case c of
  (Shift p csr) -> sem_Shift p [(s,r ch) | (s, ch) <- csr]
  (left :|: right ) -> sem_Or (r left) (r right)
  (Reduce p ) -> sem_Reduce p
  (Found p cs ) -> sem_Found p (r cs)
in r

map_Look f = cata_Look ( \ qp csr -> Shift (f qp) csr
, \ left right -> left :|: right
, \ qp -> Reduce (f qp)
, \ qp csr -> Found (f qp) csr )

merge_ch :: Look s p -> Look s p -> Look sp
l@(Shift p pcs) 'merge_ch' right
= case right of
  Shift q qcs -> Shift (p 'bestp' q) (combine pcs qcs)
  (left :|: right ) -> (l 'merge_ch' left) :|: right
  (Reduce p ) -> l :|: (Reduce p)
  (Found _ c) -> l 'merge_ch' c
  Found _ c 'merge_ch' r = c 'merge_ch' r
l 'merge_ch' _ = error "ambiguous grammar"

Figure 6: Computing the look-ahead information

In order to minimize the interpretive overhead associated with inspecting the look-ahead data structures, we pair each such structure with the function that given the input sequence:

- first locates the parsing function to be called
- and then calls this function with the input sequence.

In a proper Haskell implementation this implies that the constructors used in the look-ahead structure are being “compiled away”. Note that the functions constructed in this way (of type Realparser) will be the real parsers to be called: the look-ahead structures merely play an intermediate role in their construction, but may be discarded as soon as the functions have been constructed. Note that we really need the type extensions that are now common in most Haskell implementations; without these extensions we could not have expressed this pairing process. The first argument to a Realparser is the continuation, the second one the function to be applied to its result, the third one the error messages accumulated thus far, the fourth one its input sequence, and its result will finally be a Steps sequence, containing the parsing result at the very end.

The function mkparser takes a Look structure and pairs it with its corresponding Realparser. The function mkparser constructs a function choose that is used in the resulting Realparser to select (choose input) a Realparser (p) making use of the current input. Once selected this parser p is then called (p r st e input). So the function choose, that is the result of the homomorphism over the Look structure, has type [s] -> Realparser s a. We will now discuss how this selection process takes place, again taking the alternatives into account from bottom to top:

- Found: no further selection is needed so we return the function that, given the rest of the input, returns the parser contained in this alternative.
- End: this alternative can be dealt with just as the Found alternative; no further symbols of the input have to be inspected.
- :|: : in this case we return a parser that is going to choose dynamically between the two possible alternatives: either we reduce by calling the parser contained in this alternative (p), or we continue with the parser located by using further look-ahead information (css).
- Look: this alternative is the most interesting one. We dealing with the case that we have to inspect the next input symbol to decide how to proceed. Since performing a linear search here may be very expensive when we deal with many different starting symbols, we first construct a binary search tree out of the table, and partially evaluate the function pFind with respect to this constructed search tree. The returned function now is used to continue the selection process (and if this fails returns the error correcting parser p).
newtype RealParser s a = P (forall b result
    . (b -> Errs s -> [s] -> Steps result)
    -> (a -> b)
    -> Errs s
    -> [s] -> Steps result
    )

data Parser s a = Parser (Look s (RealParser s a)) (RealParser s a)

mkparser :: Look s a -> Parser s a
mkparser cs
  = let choose
      = cata_Look
        (\ p css
           -> let locfind = pFind (tab2tree css)
               in \ inp -> case inp of
               [] -> p
               (s:ss) -> case locfind s of
               Just cp -> (cp ss)
               Nothing -> p
           ,\ css p -> (p 'bestp'). css
           ,\ p -> const p
           ,\ p cs -> const p
         )
        in Parser cs (P (\ r st e input -> let (P p) = choose input
                         in  p r st e input ))

(P p) 'bestp' (P q)
= P (\ r st e input -> p r st e input 'best' q r st e input)

Figure 7: Constructing parsers out of Look ahead structures

(P p) 'merge_ch' (P q)
= mkparser (\ css p r st e i - > p (q r) (f.) e i)

(P p) 'fwby' (P q) = P (\ r st e i - > p (q r) (f.) e i)

(P p) 'fwby' (P q)
= mkparser$\pp csr -> Shift (pp 'fwby' qp) csr
 ,\ csr rp -> csr 'merge_ch' rp
 ,\ pp -> map_Look (pp 'fwby') csq
 ,\ pp csr -> Found (pp 'fwby' qp) csr
 ) cp
where (P p) 'fwby' (P q) = P (\ r st e i - > p (q r) (st.) e i)

Figure 8: Constructing Look structures for <|> and <*>
We given the code for the additional functions in figure 10. They speak for themselves, and are not really important for understanding the overall selection process. Note however how we have assured that this searching function is only computed once and is included in the Realparser. The interpretation overhead associated with all the table stuff is thus only performed once. In this respect our combinators really function as a parser generator. Having come thus far we can now describe how the Look structures can be constructed for the different basic combinators. The code for the \(<\mid\rangle\) and the \(<\leftrightarrow\rangle\) combinators is given in figure 8. The code for the \(<\mid\rangle\) combinator is quite straightforward: we construct the trie structure by merging the two trie structures, as described before, and invoke mkparser in order to tuple it with its associated Realparser. The code for the \(<\leftrightarrow\rangle\) combinator is a bit more involved:

- **Found**: we replace the parser already present, and which may be selected without further look-ahead, with the sequential composition (\("fwbv\)) of the two Realparsers.
- **Reduce**: apparently further information about the followers of this node is available from the context (viz. the right hand side operand of the sequential composition), and we use this to provide the badly needed information about what symbols may follow. So we replace this Reduce node with the trie structure of the right hand side parser, but with all element in it replaced by a parser prefixed with the reducing parser from the original left hand side node\(^4\).
- **:\(\mid\):** we merge both alternatives.
- **Shift**: we update the error correcting parser, and postlfix all parsers contained in the choice structure with the fact that they are followed by the second parser.

This definition seems to be horrendously costly, but again we are saved by lazy evaluation. Keep in mind that these Look structures are only being used in the function mkparser, and are only inspected for the branches until a Found or Reduce node is reached. If the grammar is \(LL(1)\) this will only be one step! As soon as mkparser has done its job the whole structure may be garbage collected.

The code for the two remaining combinators, pSym and pSucceed is given in figure 9. We will discuss the code for pSym, since the code for pSucceed leaves little imagination. In the code for pSym we see two local functions:

- **pr**: the original error correcting parser
- **pr'**: this function is only called when a function constructed by the aforementioned choose has indeed found that the expected symbol is present. In this case the check that it is present and the error correcting behavior can be skipped. So all we have to do is to take a single symbol from the input, incorporate it into the

\[\text{result, and continue with parsing. We record the successful step by adding an OK-step.}\]

If we do not make use of look ahead information we have to apply the function pr, and this is the function that is contained in the first Found construct. When this parser is merged with other parsers, this wrapper is be removed and the parser will only be called after it has been decided that it will succeed: hence the occurrences of pr' in the rest of the text. Only when the test somehow fails to find a proper look ahead resort is taken to the old pr again in order to suggest corrective steps.

For completeness sake we incorporate the additional functions used in figure 10.

5. EXTENSIONS

In the full set of combinators\(^5\), we have included some further extensions.

Not always does one want to compute the full look ahead, and is one willing to resort to the non-deterministic case; e.g. when the choice structures that are computed become very large, and are not used very often. For this purpose also dynamic versions are available.

We also note that the process of passing a value and error messages around can be extended to incorporate the accumulation of any further needed information; examples of such kind of information are the name of the file being parsed, a line number, an environment in which to locate specific identifiers etc. In that case the state should at least be able to deal with the acceptance of error messages, and recognized symbols. New combinators may be incorporated to update the state otherwise.

The production version of our combinators contains numerous further small improvements. As an example of such a subtle improvement consider the code of the function choose. Once it cannot locate the next symbol in the shift table it resorts to the error correcting version. This parser will try all alternatives, and compare all those results. But we know for sure that the first step of each result will be a Fail and thus that the first step of the selected result will be too. So instead of first finding out what is the best way to fail, and only then reporting that we have failed, it is better to immediately report a fail step and to remove the first fail step from the actual result; it is quite likely that we are dealing with a shift/reduce conflict and have gone over to the dynamic comparison of the two alternatives, and that, since we fail, the other alternative will succeed.

In a sequential composition we always incorporate the call to the second parser in the trie structure of the first one. In general it is undecidable in our approach whether this is really needed for getting a deterministic parser; it may be used to resolve a shift/reduce conflict, but computing whether such a conflict exists in the (possibly infinite) trie structure may lead to a non-terminating computation. An example of this is the parser \(X\) we have seen before: in our approach it is not possible to discover that, when building

\[\text{see www.cs.uu.nl/groups/ST/Software/UUparsing}\]
pSym a
  = let pr = ... as before
     pr' = (\ r -> \ st e (s:ss) -> Ok (r (st s) e ss))
     in mkparser (Found (P pr) (Shift (P pr) [(a, Found (P pr') (Reduce (P pr'))) ] ))

pSucceed v = mkparser ( End (P (\ r -> \ st e input -> r (st v) e input)) )

Figure 9: The code for pSym and pSucceed.

data BinSearchTree a v
  = Node (BinSearchTree a v) a v (BinSearchTree a v)
  | Leaf a v
  | Nil

tab2tree tab = tree
  where
    (tree,[]) = sl2bst (length tab) (tab)
    sl2bst 0 list = (Nil , list)
    sl2bst 1 ((a,v):rest) = (Leaf a v, rest)
    sl2bst n list
      = let
         ll = (n - 1) ‘div’ 2 ; rl = n - 1 - ll
         (lt,(a,v):list1) = sl2bst ll list
         (rt, list2) = sl2bst rl list1
       in (Node lt a v rt, list2)

pFind Nil = \i -> Nothing
pFind (Leaf r v) = \i -> case compare i r of {EQ -> Just v; otherwise -> Nothing}
pFind (Node left r v right) = let findleft = pFind left
                                  findright= pFind right
                                  in \i -> case compare i r of {EQ -> Just v;
                                  LT -> findleft i
                                  GT -> findright i

Figure 10: The additional function for search tree construction and inspection
the trie structure, we have reached a situation equivalent to the root after taking the ‘a’ branch. We may however take an approximate approach, in which we try to find out whether we can be sure it is not is not needed to create an updated version of the trie structure. As an example of such a situation consider:

```haskell
number = foldl (<<>) (map pSym [1..1000])
plus = pSucceed (+) <$> number <$> number
```

In this case an (unneeded) copy is made of the first number parser, in order to incorporate the call to +, and then once more a copy of this structure is made in order to incorporate the second number parser. Since we can immediately see that a number cannot be empty, and all alternatives are disjoint, and leading in one step to a Found node, we can postpone the updating process, and create a parser using the ‘fvery’ operator immediately.

Sometimes, once an erroneous situation has been detected, a better correction can be produced by not only taking the future but also some of the past into account. We have produced versions of our combinators that do so, and which base the decision about how to proceed by comparing sequences of parsing steps, instead of taking the greedy approach, and looking just ahead far enough to see a difference.

Another problem that occurs is that the number of different possible error corrections may get quite high, and worse, that they are all equivalent. If an operator is missing between two operands there are usually quite a number of candidates to be inserted, all resulting in a single failing step. In the approach given this would imply that the rest of the input is parsed once for all these different insertions, trying all possible corrections, and creating a parser using the ‘fvery’ operator immediately.

Finally we note that with respect to the kind of symbols we are able to handle we may easily include ranges of symbols, in this making the combinators also quite usable for describing lexers [1].

6. CONCLUSIONS
We have shown that it is possible to analyze grammars and construct parsers, that are both efficient and do correct errors dynamically. The overhead for the error correction is only a few reductions per symbol in the absence of errors: adding the Ok step and removing it again.

When comparing the current approach with the one for the LL(1) grammars we see that instead of treating the empty alternatives separately, we have now included all look-ahead information in one single data structure, thus getting a more uniform approach, and fewer special cases to deal with. Furthermore the decision about whether to insert or delete a symbol is done in a more local way, using more precise look-ahead, especially in the case of using follower symbols. This in general makes parsers continue longer in a satisfactory way; perviously a parser might prematurely decide to insert a sequence of symbols to complete a program, and throw away the rest of the input as being not needed.

When comparing the parsers coded using our library with those written in Yacc and similar formalisms, it appears that, even when using a top-down parsing method, the inputs look much nicer. Reduce-reduce conflict resulting from inserting of semantic actions e.g. do not occur, since always a sufficient context is taken into account to solve the problems occurring. We conclude by referring to the title of this paper by claiming that finally parser combinators have reached the adult state and are the tool to be chosen when to construct a parser: already they were nice, but now they have become useful.

7. ACKNOWLEDGMENTS
I want to thank Pablo Azero, Sergio Mello Schneider, Han Leushuis and David Barton for their willingness to be exposed to an almost unending sequence of different variants of the library. I thank Johan Jeuring and Pin Kars for commenting on the paper.

8. REFERENCES