Combinator Parsers: From Toys to Tools

S. Doaitse Swierstra
Department of Computer Science
Utrecht University
P.O. Box 80.089
3508 TB UTRECHT, the Netherlands
doaitse@cs.uu.nl

ABSTRACT

We develop, in a stepwise fashion, a set of parser combinators for constructing deterministic, error-correcting parsers. The only restriction on the grammar is that it is not left recursive. Extensive use is made of lazy evaluation, and the parsers constructed "analyze themselves". Our new combinators may be used for the construction of large parsers to be used in compilers in practical use.

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

There exist many different implementations of the basic parser combinators; some use basic functions [3], whereas others make use of a monadic formulation [4].

Parsers constructed with such conventional parser combinators have two disadvantages: when the grammar gets larger parsing gets slower and when the input is not a sentence of the language they break down. In [7] we presented a set of parser combinators that did not exhibit such shortcomings, provided the grammar had the so-called $LL(1)$ property; this property makes it possible to decide how to proceed during top-down parsing by looking at the next symbol in the input.

For many grammars an $LL(1)$ equivalent grammar may be constructed through *left factoring*, but unfortunately the resulting grammars often bear little resemblance to what the language designer had in mind. Extending such transformed grammars with functions for semantic processing is cumbersome and the elegance offered by combinator-based parsers is lost.

To alleviate this problem we set out to extend our previous combinators in a way that enables the use of longer look-ahead sequences The new and completely different implementation is both efficient and deals with incorrect input sequences. The only remaining restriction is that the encoded grammar is neither directly nor indirectly left-recursive: something which can easily be circumvented by the use of appropriate *chain*-combinators; we do not consider this to be a real shortcoming since usually the use of such combinators expresses the intention of the language designer better than explicit left-recursive formulations.

The final implementation has been used in the construction of some large parsers. The cost added for repairing errors is negligible.

In Section 2 we recapitulate the conventional parser combinators and investigate where the problems mentioned above arise. In Section 3 we present different basic machinery which adds error correction; the combinators resulting from this are still very short and may be used for small grammars. In Section 4 we show how to extend the combinators with the (demand driven) computation of look-ahead information. In this process we minimize the number of times that a symbol is inspected. Finally we present some further extensions in Section 5 and conclusions in Section 7.

2. CONVENTIONAL PARSER COMBINATORS

In Figure 1 we present the basic interface of the combinators together with a straightforward implementation. We will define new implementations and new types, but always in such a way that already constructed parsers can be reused with these new definitions with minimal changes. To keep the presentation as simple as possible, we assume all inputs to be sequences of `Symbol`s.

Parsers constructed using these combinators perform a depth-first search through all possible parse trees, and return all ways in which a parse can be found, an idea already found in [1]. Note that we have taken a truly "functional" approach in constructing the result of a sequential composition. Instead of constructing a value of the more complicated type `(b, a)` out of the two simpler types `b` and `a`, we have chosen to construct a value of a simpler type `a` out of the more complicated types `b -> a` and `b`. Based on these basic combinators more complicated combinators can be constructed. For examples of the use of such combinators, and the definition of more complicated combinators, see [3, 5, 6] and the

```
infixl 3 <|>    -- choice combinator
infixl 4 <*>    -- sequential combinator

type Symbol = ...
type Input = [Symbol]
type Parser a = Input -> [(a,Input)]

succeed  ::  a                             -> Parser a
symbol   ::  Symbol                        -> Parser Symbol
(<|>)    ::  Parser a        -> Parser a -> Parser a
(<*>)    ::  Parser (b -> a) -> Parser b -> Parser a
parser   ::  Parser a        -> Input    -> Result a

infixl 3 <$>    -- a derived combinator using the interface
(<$>)    ::  (b -> a)        -> Parser b -> Parser a
f <$> p = succeed f <*> p
-- straightforward implementation
succeed v  input  = [ (v    , input)]
symbol a   (b:bs) = if a == b then [(b,bs)] else []
symbol a   []     = []
(p <|> q)  input  = p input ++ q input
(p <*> q)  input  = [ (pv qv, rest  )
                    | (pv   , qinput) <- p input
                    , (qv   , rest  ) <- q qinput
                    ]
type Result a     = Either a String
parser p s = case p s of
               []              -> Right "Erroneous input"
               ((res,rest):rs) -> Left res
```

Figure 1: The basic combinators

web site for our combinators[1].

As an example of how to construct parsers using these combinators and of what they return consider (for `Symbol` we take `Int`):

```
p = (            symbol 3
   <|>  (+) <$> symbol 3 <*> symbol 4
    )

parser p [3, 4, 5]?
> [(3, [4, 5]), (7, [5])]
```

The main shortcoming of the standard implementation is that when the input cannot be parsed the parser returns an empty list, without any indication about where in the input things are most likely to be wrong. As a consequence the combinators in this form are unusable for any input of significant size. From modern compilers we expect even more than just such an indication: the compiler should correct simple typing mistakes by deleting superfluous closing brackets, inserting missing semicolons etc. Furthermore it should do so while providing proper error messages.

A second disadvantage of parsers constructed in this way is that parsing gets slow when productions have many alternatives, since all alternatives are tried sequentially at each branching point, thus causing a large number of symbol comparisons. This effect becomes worse when a naive user uses the combinators to describe very large grammars as in:

```
foldl (<|>) (map symbol [1..1000])
```

Here on the average 500 comparisons are needed in order to recognize a symbol. Such parsers may easily be implicitly constructed by the use of more complicated derived combinators, without the user actually noticing.

A further source of potential inefficiency is caused by non-determinism. When many alternatives may recognize strings with a common prefix, this prefix will be parsed several times, with usually only one of those alternatives eventually succeeding. So for highly "non-deterministic" grammars the price paid may be high, and even turn out to be exponential in the size of the input. Although it is well known how to construct deterministic automata out of non-deterministic ones, this knowledge is not used in this implementation, nor may it easily be incorporated.

We 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 only after backtracking has

[1] see `www.cs.uu.nl/groups/ST/Software/UU_Parsing`

been completed. Unfortunately, we have by then lost all context information which might enable us to decide on the proper error correcting steps. So we will start by converting 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. This breadth-first approach might be seen as a way of making many parsers work in parallel, each exploring one of the possible routes to be taken.

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 provide information about how the parsing processes are progressing before a complete parse has been constructed. 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. The continuation parameter `r` represents the rest of the parsing process, which is to be called when the current parser succeeds. It can be seen as encapsulating a stack of unaccounted-for symbols from the right hand sides of partially recognized productions, against which the remaining part of the input is to be matched. We have again defined a function `parse` that starts the parsing process. Its continuation parameter is the function `null`, which checks whether the input has indeed been consumed totally when the stack of pending symbols has been depleted.

3.2 Parsing histories

An essential design decision now is not just to return a final result, but to combine this with the parsing history, thus enabling us to trace the 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 corresponds either to the insertion into or the deletion of a symbol from the input stream

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

getresult :: Steps result -> result
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 this pair can only be constructed after having reached the end of the parsing process and thus having access to the final result. Instead, we introduced a more intricate data type, which allows us to start producing tracing information before parsing has completed. Ideally, one would like to select the result with the fewest `Fail` steps, i.e., that sequence that corresponds to the one with a minimal editing distance to the original 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 into consideration. Suppose e.g. that an unmatched `then` symbol is encountered, and that we want to find the optimal place to insert the missing `if` symbol. In this case there may be many points where it might be inserted, and many of those points are equivalent with respect to editing distance to some correct input.

To prevent a combinatorial explosion we take a greedy approach, giving preference to the parsing with the longest prefix of `Ok` steps. So we define an ordering between the `Steps`, based on longest successful prefixes of `Ok` steps:

```
best :: Steps rslt -> Steps rslt -> Steps rslt
_@(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 an essential observation to be made here: when there is no preference between two sequences based on their first step, we postpone the decision about which of the operands to return,*while still returning information about the first step in the selected result.*

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 any symbol we expect or when we expect at least one more symbol and the input is exhausted. We consider 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 report a `Fail` step. If we add this error recovery to the combinators defined before, we get the code in Figure 3. Note that if any input left at the end of the parsing process is left it is deleted, resulting in a number of failing steps (`Fail(Fail(... (Stop True)))`). This may seem superfluous, but is needed to indicate that not all input was consumed. The operator `||`, that was used before to find out whether at a branching point at least one of the alternatives finally led to success, has been replaced by the `best` operator which selects the "best" result. It is here that the change from a depth-first to a breadth-first approach is made: the function `||` only returns a result after at least its first operand has been completely evaluated, whereas the function `best` returns its result in an incremental way. It is the function `getresult` at the top level that is actually driving the computation by repeatedely asking for the constructor at the head of the `Steps` value.

3.4 Computing semantic results

The combinators as just defined are quite useless, because the added error correction makes the parser always return `True`. We now have to add two more things:

```
type Result a  = Bool
type Parser    = (Input -> Bool) -> (Input -> Bool)

succeed   = \ r input -> r input
symbol a  = \ r input -> case input of
                          (b:bs) -> a == b && r bs
                          []     -> False
p <*> q   = \ r input -> p (q r) input
p <|> q   = \ r input -> p r input || q r input

parse p input = p null input -- null checks for end of input
```

Figure 2: The continuation-based combinators

```
type Result a =  Steps Bool

symbol a      = \ r input -> case input
                             of (b:bs) -> if a == b then Ok(r bs)
{- insert the symbol a -}                else Fail (r input
                                                     `best`
{- delete the symbol b -}                           symbol a r bs
                                                   )
{- insert the symbol a -}        []     -> Fail (r input)

succeed    = \ r input -> r input

p <|> q       = \ r input -> p r input `best` q r input
p <*> q       = \ r input -> p (q r) input

parse p = getresult . p (foldr (const Fail) (Stop True))
```

Figure 3: Error correcting parsers

1. the computation of a result, like done in the original combinators
2. the generation of error messages, indicating what corrective steps were taken.

Both these components can be handled by accumulating the results computed thus far in extra arguments 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 "pending" elements, that is, elements of the right hand side of productions that have been recognized and are waiting to be used in a reduction (which in our case amounts to the application of the first element to the second).

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 alternative has length at most 2: each occurrence of a `<*>` combinator introduces an (anonymous) non-terminal. If the length of a right hand side is larger than 2, the left-associativity of `<*>` determines how normalization is defined. So there is an element pending on the stack for each recognized left operand of some `<*>` parser whose right hand side part has not been recognised yet.

We decide to represent the stack of pending elements with a function too, since it may contain elements of very different types. The types of the stack contaniing the reduced items and of the continuation now become:

```
type Stack a b = a -> b
type Future b result = b -> Input -> Steps result
```

Together this gives us the following new definition of the type `Parser`:

```
type Parser a =
  forall b result .
     Future b result   -- the continuation
  -> Stack a b         -- the stack of pending values
  -> Input
  -> Steps result
```

This is a special type that is not allowed by the Haskell98 standard, since it contains type variables `b` and `result` that are not arguments of the type `Parser`. By quantifying with the `forall` construct we indicate that the the type of the parser does not depend on these type variables, and it is only through passing functions that we link the type to its environment. This extension is now accepted by most Haskell compilers. So the parser that recognises a value of type `a` combines this value with the stack of previously found values which will result in a new stack of type `b`, which in its turn is passed to the continuation as the new stack.

The interesting combinator here is the one taking care of sequential composition which now becomes:

```
((p <*> q) r stack input = p (q r) (stack.) input
```

When `pv` is the value computed by the parser `p` and `qv` the value computed by the parser `q`, the value passed on to `r` will be:

```
(((stack .) pv) qv) = (stack. pv) qv = stack (pv qv)
```

which is exactly what we would expect.

Finally we have to adapt the function `parse` such it transforms the constructed result to the desired result and initializes the stack (`id`):

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

We will not give the new versions of the other combinators here, since they will show up in almost 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 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 produce information about the result without actually having made a choice yet. Including error messages with the `Fail` constructors would force 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 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).

4. INTRODUCING LOOK-AHEAD

In the previous section we have solved the first of the problems mentioned, 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 solve the two remaining problems (backtracking and sequential selection), which both have to do with the low efficiency, in one sweep.

Thus far the parsers were all defined as functions about which we cannot easily get any further information. An example of such useful information is the set of symbols that may be recognized as *first*-symbols by a parser, or whether the parser may 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 is constructed using this information.

4.1 Tries

To see what such information might look like we first introduce yet another formulation of the basic combinators: we construct a trie-structure representing all possible sentences in the language of the represented grammar (see Figure 5). This is exactly what we need for parsing: all sentences of the language are grouped by their common prefix in the trie structure. Thus it becomes possible, once the structure has been constructed, to parse the language in linear time.

For a while we forget again about computing results and error messages. Each node in the trie represents the tails of sentences with a common prefix, which in its turn is represented by the path to the root in the oevrall structure representing the language. A `Choice` node represents the non-empty tails by a mapping of the possible next symbols to the tries representing their corresponding tails. An `End` node represents the end of a sentence. The `:|:` nodes corresponds to nodes that are both a `Choice` node (stored in the left operand) and an `End` node (stored in its right operand) [2]. Notice that the language $ab|ac$ is represented by:

```
Choice [('a', Choice [('b', End), ('c', End)])]
```

in which the common prefix has been factored out. In this way the cost associated with the backtracking of the parser has now been moved to the construction of the `Sents` structure.

For the language $a^*b|a^*c$ constructing the trie is a nonterminating process. Fortunately lazy evaluation takes care of this problem, and the merging process only proceeds far enough for recognising the current sentence.

A shortcoming of this approach, however, is that it introduces 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. Furthermore we may have succeeded in parsing in linear time, but this is only possible because we have shifted the work to the construction of the trie structure. In the sequel we will show how we can construct the equivalent to the trie-structure by combining precomputed trie-structure building blocks.

[2] We could have encoded this using a slightly different structure, but this would have resulted in a more elaborate program text later.

```
type Parser a =
  forall b result .
     Future b result
  -> Stack a b
  -> Errs
  -> Input
  -> Steps result
data Errors = Deleted  Symbol  String Errors
            | Inserted Symbol  String Errors
            | Notused          String
instance Show Errors where
  show (Deleted  s pos e ) = "\ndeleted " ++ show s ++ " before " ++ pos ++ show e
  show (Inserted s pos e ) = "\ninserted "++ show s ++ " before " ++ pos ++ show e
  show (NotUsed ""       ) = ""
  show (NotUsed pos      ) = "\nsymbols starting at "++ pos ++ " were discarded "
eof            = " end of input"
position ss    = if null ss then eof else show (head ss)

symbol a = let pr = \ r st e input ->
                    case input of
                      (b:bs)  ->
                       if a == b then  Ok  (r (st s)  e bs)
                                 else  Fail ( (pr r  st (e . Deleted b (position bs))  bs)
                                                              'best'
                                                 (r (st a) (e . Inserted a (show b))      input)
                                               )
                      []      ->       Fail (r (st a) (e . Inserted a eof) input)
           in pr
succeed v   = \ r stack errors input -> r (stack v) errors input
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
parse p input
 = getresult ( p (\ v errors inp -> foldr (const Fail) (Stop (v, errors.position inp)) inp)
                 id id input
             )
```

Figure 4: Correcting and error reporting combinators

4.2 Merging the different approaches

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, and inspects each symbol in the input only once

In our final solution we will merge these two approaches. We will compute `Sents` fragments on which we base the decision how to proceed with parsing, and use the continuation based parsers to actually accept the symbols. Since the 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 `Sents` structure, we will construct a similar structure which may be used to select the parser to continue with.

Before proceeding, let us consider the following grammar fragment:

```
succeed (\ x y -> x) <*> symbol 'a' <*> symbol 'b'
          <|>
succeed (\ x y -> y) <*> symbol 'a' <*> symbol 'c'
```

The problem that arises here is what to do with the parsers preceding the respective `symbol 'a'` occurrences: we can only decide which one to take after a symbol `'b'` or `'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. As a result of this the actual parsing and the computation of the result may run out of phase.

We now discuss the full version of the structure we have defined to do the bookkeeping of our look-ahead information (see the definition in Figure 6), and the actions to be taken once a decision has been made. The `p` components of the `Look` structure are the parsing functions that actually accept symbols and do the semantic processing. We will discuss the different alternatives in reverse order:

`Found p (Look p)` this alternative indicates that the only

```
type Parser = Sents
data Sents   = Choice [(Symbol, Sents)]
             | Sents :|: Sents -- left is Choice, right is End
             | End

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   -> (s,  ct <|> ct'):combine xs ys
combine [] cs'   = cs'
combine cs []    = cs

symbol a      = Choice [(a, End)]
succeed       = End

l@(Choice as)   <|> r
 = case r
    of  (Choice bs)  -> Choice (combine as bs)
        (p :|: q  )  -> (l <|> p) :|: q
        End          -> l         :|: End
l               <|> r@(Choice _) = r <|> l
_               <|> _            = error "ambiguous grammar"

Choice cs <*> q = Choice [ (s, h <*> q)| (s, h) <- cs ]
l :|: r   <*> q = (l <*> q)  <|>  (r <*> q)
End       <*> q = q

parse :: Parser -> Input -> Bool
parse (Choice cs) (a:as)  = or [ parse future as| (s, future) <- cs, s==a]
parse (p :|: q  ) inp     = parse p inp || parse q inp
parse End          []     = True
parse _            _      = False
```

Figure 5: Representing all possible sentences.

```
data Look p = Shift  p      [(Symbol, Look   p)]   -- like Choice
            | (Look  p) :|: (Look   p)             -- Shift/Reduce
            | Reduce p                             -- End
            | Found  p      (Look   p)             -- Uniquely determined

merge_ch :: Look p -> Look 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` r@(Shift  _ _) = r  `merge_ch` l
_            `merge_ch` _              = error "ambiguous grammar"

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

Figure 6: Computing the look-ahead information

possible parser that applies at this point is the first component of this construct. It corresponds to a non-merged node in the trie structure. As such it marks also the end of a trie fragment out of which the original try structure may be reconstructed without any merging needed. The `(Look p)` part is used when the structure is merged with further alternatives. This is the case when that other alternative contains a path similar to the path that leads to this `Found` node. As one can see in the function `` `merge_ch` `` this `Found` constructor is removed when the structure is merged with other alternatives.

`Reduce p` this indicates that in using the look-ahead information we have encountered all symbols in the right hand side of a production (we have reached a *reduce* state in LR terminology), and that the parser `p` is the parser to be used.

`(:|:)` 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.

`Shift p [(s, Look p)]` this corresponds to a shift state, in which we need at least one more symbol in order to decide how to proceed. The parser `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 `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).

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

1. uses the input sequence to locate the parsing function to be called
2. 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" as a form of partial or stages evaluation. 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, and may be discarded as soon as the functions have been constructed. The first argument to a `Realparser` is the continuation, the second one the accumulated stack, 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` interprets 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 `Input -> Realparser 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.

`Reduce` 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`).

`Shift` this alternative is the most interesting one. We are dealing with the case where we have to inspect the next input symbol. Since performing a linear search here may be very expensive, 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 is now used in the constructed fucntion for the continuation of the selection process (and if this fails returns the error correcting parser `p`).

We give the code for the additional functions in Figure 9. They speak for themselves, and are not really important for understanding the overall selection process. 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 basic combinators is given in Figure 8. The code for the `<|>` 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 `<*>` 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 (`` `fwby` ``) 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

```
newtype RealParser a = P (forall  b   result
                           .    (b -> Errs -> Input -> Steps result)
                             -> (a -> b)
                             -> Errs
                             -> Input-> Steps result
                          )

data Parser a     = Parser (Look (RealParser a)) (RealParser a)
mkparser :: Look a -> Parser a

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
                          )

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
          ) cs
  in Parser cs (P (\ r st e input -> let (P p) = choose input
                                     in   p r st e input
               )  )
```

Figure 7: Constructing parsers out of `Look` ahead structures

```
symbol 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, (Reduce (P pr')) ) ] ))

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

(Parser cp _) <|> (Parser cq _)  = mkparser (cp `merge_ch` cq)

(Parser cp _) <*> ~(Parser csq qp)
      = mkparser$cata_Look ( \ pp csr -> Shift (pp `fwby` qp)  csr
                           , \ csr rp -> csr `merge_look` 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: The final basic combinators

by a parser prefixed with the reducing parser from the original left hand side node[3].

(:|:) we merge both alternatives.

Shift we update the error correcting parser, and postfix 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.

In the code for symbol 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 discovered 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 result, and continue 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 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 we use the old pr again in order to incorporate corrective steps.

For the sake of completeness we incorporate the additional functions used in Figure 9.

5. EXTENSIONS

In the full set of combinators, we have included some further extensions.

Computation of a full look-ahead may be costly, e.g. when the choice structures that are computed become very large, and are not used very often. In such cases one may want to use a non-deterministic approach. For this purpose dynamic versions are also available that have the efficiency of the backtracking approach given before.

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 kinds 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 store error messages, and recognized symbols. Extra combinators have been introduced in the library to update the state.

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 parsing 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. Since the fact that repair is possible may be discovered during the selection in a choice structure we have to make sure that no backtracking is taking place over this prefix if we want to guarantee that all correct symbols are examined only once. In the production version this adds one further accumulating parameter to the construction of the actual parsers.

Sometimes, once an erroneous situation has been detected, a better correction can be produced by taking not only 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. By making different choices for the function best we can incorporate different error correction strategies.

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 may occur in the (possibly infinite) trie structure may lead to a non-terminating computation. An example of this we have seen before with the grammar for the language $a^*b|a^*c$; in our approach it is not possible to discover that, when building 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 needed to create an updated version of the trie structure. As an example of such a situation consider:

```
number = foldl (<|>) (map symbol [1..1000])
plus = succeed (+) <*> 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 'fwby' operator immediately.

[3]Strictly speaking this is only needed when the reduce node actually is the right operand of a :|: construct, indicating the existence of a now resolvable shift-reduce conflict

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

tab2tree tab = tree
 where
  (tree,[]) = sl2bst (length tab) (tab)
  sl2bst 0 list     = (Nil   , list)
  sl2bst 1 ((a,v):rest) = (Node Nil a v Nil, 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    (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 9: The additional function for Braun tree construction and inspection

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 each of these different possible corrections, trying to find out whether there really is no difference. In contrast to generalised LR parsing we do not have access to an explicit representation of the grammar, and especially we cannot compare the functions that represent the different states, and as a consequence we cannot discover that we are comparing many parellel but equivalent parses. In the library there are facilities for limiting such indefinite comparisons by specifying different insertion and deletion costs for symbols and by limiting the distance over which such comparisons are being made. In this way fine-tuning of the error correction process is possible without interfering with the rest of the parsing process.

Finally we have included basic parsers for ranges of symbols, thus making the combinators also quite usable for describing lexers [2].

By defining additional combinators that extend the basic machinery we may deal with ambiguous grammars too. As an example consider a simple lexer defined as follows:

```
many p = let manyp = many p
         in     succeed (:) <*> p  <*> manyp
    <|> succeed []
letter = 'a' `upto` 'z'
identifier = (:) <*> letter <*> many letter
token []     = succeed []
token (s:ss) = succeed (:) <*> symbol s <*> token ss
```

```
lexsym = identifier <|> token "if" <|> ...
```

In this case we have two problems:

1. we want to use a greedy approach when recognising identifiers by taking as many letters as possible into account
2. the string "if" should be parsed as a token and not as an identifier

We may solve this problem by introducing a new combinator:

```
step v n
 = mkparser
   ( End (P (\ r st e input -> Step n (r (st v) e input)
   )     )  )
```

and changing the code above into:

```
many n p = let manyp = many p
           in     succeed (:) <*> p <*> manyp
      <|> step [] n
identifier = (:) <*> letter <*> many 2 letter
token []     = step [] 1
token (s:ss) = succeed (:) <*> symbol s <*> token ss
```

If we have seen the text "if", and there are no further steps possible with cost 0 (e.g. further letters), then we choose the `token` alternative since $1 < 2$.

6. EFFICIENCY

It is beyond the scope of this paper to provide a detailed discussion about the efficiency and space usage of the parsers constructed in this way. We notice however the following:

- it follows directly from the construction process that the number of constructed parsers can be equivalent to the number of $LR(0)$ states in a bottom-up parser, so there is no space leak to be expected from that; this is however only the case if the programmer has performed left factorisation by hand. If not the system will perform the left factorisation, but since it cannot identify equality of parsers it will do so by merging two identical copies of the same look-ahead information.
- the number of times a symbol is inspected may be more than once: where a bottom-up parser would have a shift-reduce conflict we basically pursue both paths until a difference is found; if the grammar is $LR(1)$ this implies we are inspecting one symbol twice in such a situation. We feel that in all practical circumstances this is to be preferred over the construction of full $LR(1)$ parsers, where the number of states easily explodes. It is the relatively low number of states of the $LR(0)$ automaton, and thus of our approach, that makes the $LALR(1)$ handled by Yacc to be preferred over $LR(1)$. Besides that has our approach the advantage of smoothly handling also longer look-aheads when needed. The lazy evaluation takes nicely care of the parallel pursuit for success that would be a nightmare to encode in C or Java.
- using the decision trees makes that at each choice point we have a complexity that is logarithmic in the number of possible next symbols. The simple parser combinators all are linear in that number, a fact that really starts to hurt when dealing with productions that have really many alternatives.

7. CONCLUSIONS

We have shown that it is possible to analyze grammars and construct parsers, that are both efficient and correct errors. 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 we have now included all look-ahead information in one single data structure, thus getting a more uniform approach. Furthermore the decision about whether to insert or delete a symbol is done in a more local way, using precise look-ahead, especially in the case of follower symbols. This in general makes parsers continue longer in a more satisfactory way; previously 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, the inputs look much nicer. Reduce-reduce conflict resulting from the incorporation of semantic actions do not occur, since always a sufficient context is taken into account. 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 one wishes to construct a parser: already they were nice, but now they have become useful.

8. 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 variants of the library. I thank Johan Jeuring and Pim Kars and anonymous referees for commenting on the paper.

9. REFERENCES

[1] W. H. Burge. *Recursvie Programming Techniques.* Addison Wesley, 1975.

[2] M. M. T. Chakravarty. Lazy lexing is fast. In A. Middeldorp and T. Sato, editors, *Fourth Fuji International Symposium on Functional and Logic Programming: FLOPS 99*, number 1722 in LNCS, pages 68–84, 1999.

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

[4] G. Hutton and H. Meijer. Monadic parser combinators. *Journal of Functional Programming*, 8(4):437–444, 1988.

[5] J. Jeuring and S. Swierstra. Lecture notes on grammars and parsing. http://www.cs.uu.nl/people/doaitse/Books/GramPars2000.pdf.

[6] S. Swierstra and P. Azero Alcocer. Fast, error correcting parser combinators: a short tutorial. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99 Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics*, volume 1725 of *LNCS*, pages 111–129, November 1999.

[7] S. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.