Fast, Error Correcting Parser Combinators: A Short Tutorial

S. Doaitse Swierstra[1] and Pablo R. Azero Alcocer[1,2]

[1] Department of Computer Science, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht,The Netherlands, `{doaitse,pablo}@cs.uu.nl`
[2] Univeridad Major de San Simon, Cochabamba, Bolivia

1 Introduction

Compiler writers have always heavily relied on tools: parser generators for generating parsers out of context free grammars, attribute grammar systems for generating semantic analyzers out of attribute grammars, and systems for generating code generators out of descriptions of machine architectures. Since designing such special formalisms and constructing such tools deals with one of the most important issues in computer science, courses on compiler construction have always formed part of the core computer science curriculum.

One of the aspects that make modern functional languages like Haskell [3] and ML [4] so attractive is their advanced type system. Polymorphism and type classes make it possible to express many concepts in the language itself, instead of having to resort to a special formalism, and generating programs out of this. It should come as no surprise that, with the increased expressibility provided by the new type systems, the question arises to what extent such tools may be replaced by so-called combinator libraries. In this paper we will present a combinator library that may be used to replace conventional parser generators.

We will be the first to admit that many existing special purpose tools do a great job, and that our library falls short of performing in an equally satisfying way. On the other hand there are good arguments to further pursue this combinator based route. We will come back to this after we have introduced conventional combinator based parsing, since at that point we will have some material to demonstrate the points we want to make. Let it suffice to say here that the size of our tool boxes is only a fraction of the code size of conventional tools; we will present a library of parsing combinators that takes just less than a 100 lines, whereas it provides many features that conventional tools lack. This paper contains enough room to include the full text of the library; something that can definitely not be said about almost any other existing tool.

Parser combinators have been a joy for functional programmers to use, to study, and to create their own versions of [2,1,5]. In many courses on functional programming they form the culminating point, in which the teacher impresses his students with an unbelievably short description of how to write parsers in a functional language. We will do so here too. Next we will explain the advantages of programming with such combinators, and at the same time present

```
  module BasicCombinators where
  infixl 3 <|>
  infixl 4 <*>

5 type Parser s a = [s] -> [(a,[s])]

  pSucceed v input = [ (v    , input)]
  pFail     input = [                ]

10 pSym  :: Eq  s  => s                              -> Parser s s
  (<|>) :: Eq  s  => Parser s a        -> Parser s a -> Parser s a
  (<*>) :: Eq  s  => Parser s (b -> a) -> Parser s b -> Parser s a

  pSym a  (b:rest) = if a == b then [(b,rest)] else []
15 pSym a  []       = []

  (p <|> q) input = p input ++ q input

  (p <*> q) input = [ (pv qv, rest  )
20                    | (pv   , qinput) <- p input
                    , (qv   , rest  ) <- q qinput
                    ]
```

Listing 1: BasicCombinators0

some extensions. We will however also indicate some of the disadvantages of this
extremely simplistic approach; thus in section 7 we will show how to cure these
disadvantages. Finally we will sketch how we have constructed a slightly more
elaborate version of our combinators, which are usually considerably faster and
are of production quality.

2   Basic Combinator Parsing

In listing 1 we provide the full text of a library for building parsers. A `Parser s a`
is a function (`->`) that takes a sequence of tokens as input (`[s]`) and returns a list
(`[...]`), containing all possible ways in which a prefix of the input can be recog-
nized and converted into a value of type `a`, each tupled with the corresponding
remaining part of the input (`(a, [s])`). A parser `pSucceed v` recognizes the
empty string and returns the value `v` as the witness of this success, whereas
the parser `pFail` always fails, and thus returns an empty list of solutions. The
function `pSym` takes a single token as parameter and returns a parser that either
recognizes just this token or fails. The function `pSym` is thus not a parser but
a function that builds a parser. The parser combinator `<|>` constructs a new
parser out of two alternative parsers, and thus corresponds to the symbol | in
context free grammars; all it has to do is simply applying both parsers to the

```
   module TreeParsing1 where
   import BasicCombinators

   data Tree = Leaf Char
5            | Bin Tree Tree
             deriving Show

   pTree1 =        pSucceed Leaf
               <*> pDigit1
10         <|>    pSucceed (\ _ left right _ -> Bin left  right)
               <*> pSym '('  <*> pTree1 <*> pTree1 <*> pSym ')'

   pDigit1 = foldr (<|>) pFail (map pSym "0123456789")

15 foldr op e (a:x) = a 'op' foldr op e x
   foldr op e []    = e
```

Listing 2: TreeParsing1

input and to concatenate the two lists of found parses. The combinator `<*>`, that takes care of the sequential composition, requires a bit more explanation. The construct used in its definition is called list comprehension. First the parser `p` is applied to the input, and for all possible ways (`<-`) in which this parser `p` succeeds with value `pv` and remaining input `qinput`, parser `q` is applied to this remaining input `qinput`. This in its turn results in a list of (`qv, rest`) pairs. There is now some freedom in combining a result `pv` with its corresponding `qv` values. We have decided here to use function application: the parser `p` has to return a function that is applied to the result of the parser `q`. This fact is clearly expressed in the type of the combinator `<*>`.

   In listing 2 we use the combinators for defining a parser `pTree1` for binary trees that have a digit at their leafs. By loading this module into the Haskell interpreter Hugs we may now call our first parser:

```
   TreeParsing1> pTree1 "(2(34))"
   [(Bin (Leaf '2') (Bin (Leaf '3') (Leaf '4')),"")]
```

   You can see that there is only one way of making the input into a `Tree` and that all input was consumed in doing so (`""`).

   You may not immediately see how the second alternative of `pTree1` works. For this we have to carefully inspect the fixity declaration of `<*>`, i.e. the text `infixl 4 <*>`; this defines `<*>` to be left-associative and thus something of the form `f <*> a <*> b <*> c <*> d` is interpreted as `((((f <*> a) <*> b) <*> c) <*> d)`. In our case `pSucceed (\e _ left right _ -> Bin left  right)` always returns a function that takes 4 arguments, and that is precisely the number of components recognized by the remaining part of this alternative.

3   Extending the Library

We will now exploit an important property of our approach: the combinators are not operators in some special formalism, but instead are just functions defined in a general purpose programming language. This implies that we can write functions returning parsers and pass parsers as argument to other functions. This enables us to introduce a wealth of derived combinators, that take care of often occurring patterns in writing parsers. We have already seen a small example of such new possibilities when we defined the parser that recognizes a single digit. Instead of writing down a whole lot of parsers for the individual digits and explicitly combining these, we have taken the sequence of digits `"0123456789"`, have converted each element of that sequence (`map`) into a sequence of parsers by applying `pSym` to it, and finally have combined all these parsers into a single one by applying `foldr (<|>) pFail`. This function `foldr` builds a result by starting off with the unit element `pFail` and then combining this with all elements in the list using the binary function (`<|>`). Since this is an often occurring pattern the real functional programmer immediately sees an opportunity for abstraction here:

```
pAnyOf :: Eq s => [s] -> Parser s s
pAnyOf = foldr (<|>) pFail . map pSym
pDigit2 = pAnyOf "01234567890"
```

In listing 3 we have given definitions for some more often occurring situations, and our tree parser might, using these new combinators also be defined as:

```
pTree2 =           Leaf <$> pDigit2
       <|> pParens (Bin  <$> pTree2 <*> pTree2)
```

,which now has become almost isomorphic to the data type definition. It should be clear from this example that there is now no limit to extending this library.

4   Advantages of this Approach

As a final example of what can be done we will now show how to construct parsers dynamically. by writing a parser for an expression language with infix operators. An example input is:

```
(L+R*)a+b*(c+d)
```

and the code we want to generate is:

```
abcd+*+
```

which is the reversed Polish notation of the input expressions.

The text (`L+R*`) indicates that + is left (L) associative and has lower priority than *, which is right (R) associative. In this way an unlimited number of operators may be specified, with relative priorities depending on their position in this list.

We start by defining a function that parses a single character identifier and returns as its result that identifier in the form of a string:

```
   module ExtendedCombinators where
   import BasicCombinators
   infixl 4 <$>, <$, <*, *>, <**>, <??>
   infixl 2 'opt'
5
   pAnyOf :: Eq s => [s]                            -> Parser s s
   opt    :: Eq s => Parser s a  -> a               -> Parser s a
   (<$>)  :: Eq s => (b -> a)    -> Parser s b      -> Parser s a
   (<$  ) :: Eq s => a           -> Parser s b      -> Parser s a
10 (<*  ) :: Eq s => Parser s a  -> Parser s b      -> Parser s a
   (  *>) :: Eq s => Parser s a  -> Parser s b      -> Parser s b
   (<**>) :: Eq s => Parser s b  -> Parser s (b->a) -> Parser s a
   (<??>) :: Eq s => Parser s b  -> Parser s (b->b) -> Parser s b

15 pAnyOf toks = foldr (<|>) pFail (map pSym toks)
   p 'opt' v = p <|> pSucceed v
   f <$> p   = pSucceed f       <*> p
   f <$  p   = const f          <$> p
   p <*  q   = (\ x _ -> x)     <$> p <*> q
20 p *>  q   = (\ _ x -> x)     <$> p <*> q
   p <**> q  = (\ x f -> f x) <$> p <*> q
   p <??> q  =                   p <**> (q 'opt' id)

   pFoldr         alg@(op,e)      p
25   = pfm where pfm = (op <$> p <*> pfm)   'opt' e
   pFoldrSep      alg@(op,e) sep p
     = (op <$> p <*> pFoldr alg (sep *> p)) 'opt' e
   pFoldrPrefixed alg@(op,e) c    p
     = pFoldr alg (c *> p)

30
   pList         p = pFoldr         ((:), [])  p
   pListSep     s p = pFoldrSep      ((:), []) s p
   pListPrefixed c p = pFoldrPrefixed ((:), []) c p

35 pSome p      =  (:) <$> p <*> pList p
   pChainr op x =  r where r = x <**> (flip <$> op <*> r 'opt' id)
   pChainl op x =  f <$> x <*> pList (flip <$> op <*> x)
                where
                  f x [] = x
40                f x (func:rest) = f (func x) rest

   pPacked l r x  =   l *>  x <*   r

   -- some ad hoc extensions
45
   pOParen = pSym '('
   pCParen = pSym ')'
   pParens  = pPacked pOParen pCParen
```

Listing 3: ExtendedCombinators

```
pVar = (\c -> [c]) <$> pAnyOf ['a'..'z']     .
```

The next step is to define a function that, given the name of an operator, recognizes that operator and as *a result returns a function that will concatenate the two arguments of that operator and postfix it with the name of the operator*, thus building the reversed Polish notation:

```
pOp name = (\ left right -> left++right++[name]) <$ pSym name
```

Note that, by using the operator `<$` we indicate that we are not interested in the recognized operator; we already know what this is since it was passed as a parameter.

Next we define the function `compile`. For this we introduce a new combinator `<@>`, that takes as its left hand side operand a parser constructor `f` and as its right hand side operand a parser `g`. The results `v` of parsing a prefix of the input with `g`, are used in calling `f`; these calls, in their turn, result in new parsers which are applied to the `rest` of the input:

```
(f <@> g) input  = [ f v rest | (v, rest) <- g input ]
```

Since our input consists of two parts, the priority declarations and the expression itself , we postulate that the function compile reads:

```
compile = pRoot <@> pPrios
```

First we focus on the function `pRoot`, that should take as argument the result of recognizing the priorities. Here we will assume that this result is a function that, given how to parse an operand, parses an expression constructed out of operands and the defined operators:

```
pRoot prios = let pExpr = prios (pVar <|> pParens pExpr) in  pExpr
```

There is a difference between an operator that occurs in the declaration part of the input and one in the expression part: the former may be any operator, whereas the latter can only be an operator that has been declared before. For the priority declaration part we thus introduce a new parser that recognizes any operator, and returns a parser that compiles the just recognized operator using the function `pOp` defined before:

```
pAnyOp = pOp <$> pAnyOf "+*/-^"     -- just some possible operators
```

Now suppose we have recognized a left and a right associative operator resulting in an operator compilers `pLeft` and `pRight`. Out of these we can construct a function that, given the operand parser, parses infix expressions containing `pLeft` and `pRight` occurrences:

```
pLR factor = (pChainl pLeft . pChainr pRight) factor.
```

Generalizing this pattern to an unlimited number of operators we now deduce the definition:

```
pPrios  = pFoldr ((.), id) ((   pChainl <$ pSym 'L'
                             <|> pChainr <$ pSym 'R')  <*> pAnyOp)
```

Let us now compare once more this approach with the situation where we would have used a special parser generator. In the combinator approach we can freely introduce all kinds of abbreviations by defining new combinators in terms

of existing ones; furthermore we may define higher order combinators that take arguments and return values that may be parsers. This is a property we get for free here, and is absent from most tools: always the syntax is fixed and at most some form of macro processing is available as an abstraction mechanism.

Another important consequence from embedding our parser construction in an existing language is that type checking and error reporting can directly be done at the program level, and not at the level of some generated program.

5   Analysis

Before going on let us reflect for a while on why this all works so neatly; somehow we managed to define a new language within an existing one. There are many important aspects to be distinguished here.

In the first place having *polymorphic types* is essential. We managed to keep the types of the parser and the types of the result completely separated. There is no way in which the parsers can inspect those values or mutilate them. All they know is that they have types like `a` and `a -> b`. If these are the only things known, the only thing a parser can do is apply the one to the other, but that was the intention of providing these types. So the combinators really are a conservative extension of the rest of the program.

In the second place the *type classes* make it possible to precisely define the interfaces between the parsing part and the rest of the program. For the parsing it is necessary to know whether tokens are equal or not, and precisely this fact is thus specified in the context part (`Eq s =>`) of the type of the combinators. This is the only needed, but also the only available property of tokens in the parsing part of your programs.

The third issue that makes things look nice is of a more syntactic nature: by being able to define new infix operators, parser definitions can be made to resemble grammars, thus taking away another reason for having a special formalism.

Although the combinators defined before look very attractive, they have some serious shortcomings, that make them almost unusable in practice.

Because we have used the list of successes approach, the result for incorrect inputs will be an empty list, and no indication what went wrong and where is given. Furthermore we rely on backtracking for finding all possible parses; our (extended) combinators were cleverly defined in such a way that they return the longest possible parse first, and this is usually what one wants. When the input contains errors or we are not primarily interested in the greedy solution, there is a high backtracking overhead to be paid.

In the next section we will attack these two problems, and will come up with a set of basic combinators that not only report errors but also repair the input. We will sketch the implementation of an equivalent set of combinators that do not even suffer from the backtracking overhead. Of course there is a price to be paid too: they are far more complicated and the number of lines needed for describing them is almost tenfold; these hundred lines however still compare

favorably with the size of equivalent special purpose tools such as YACC which have far less expressive power.

6   Error Locating Parsing Combinators

Before going into techniques for error correction we first spend some time on investigating how to get some form of error reporting. Since the two aspects are deeply intertwined however we will only briefly do so.

In case the input is erroneous we will not be able to return a complete parse, but we may try to report on how far we got in the input. In order to do so we change the type of the parsers such that they not only return a value, but also how many tokens were accepted in the parsing process. To this end an extra argument is tupled with the input: the number of tokens accepted so far. The new combinators are given in listing 4.

The main disadvantage of this approach is that, once we have discovered where the erroneous point is, we have lost the computation that led us there; this implies that we have both lost the necessary contextual information for deciding what kind of repair to make, and the information to continue with the parsing process from that point on.

This suggests that we do the error correction as soon as we discover an erroneous situation, because then we still have the contextual information at our disposal. This however is not trivial. We may locally discover that we got stuck, but may be there is some other alternative that will bring us much further; in that case the current state can just be discarded and no further time should be wasted on it. In order to make this decision we have to convert our depth-first backtracking strategy into a breadth-first strategy, in which we work on all possible parses in parallel. Only then will we be able to discover whether correcting actions are worthwhile to be taken, or whether there are still other alternatives present that can make progress without having to perform such corrective actions.

The basic step we take now is to look at the corrective actions and the decision whether they were needed or not separately: we generate a set of candidates containing all possible parses and all possible corrections, and decide elsewhere which candidate wins. This approach may look horrendously expensive, but we will exploit lazy evaluation to prevent the full computation of all these (possibly corrected) parses. This will also take care of another problem: once we start changing the input by adding or deleting symbols, the set of parses is likely to become infinite, and we had better to avoid computing this whole collection!

7   Error Correcting Parsing Combinators

Since we have decided to deal with the error correction as an integrated part of the parsing process, we will start with a closer inspection of the kind of corrections we want to make. The point where we discover that no progress can be made is in the function pSym, where we expect to see a specific token at the

```
    type Parser s a = ( Integer, [s]) -> [Result s a]

    data Result s a  = Until Integer
                     | Succeed a  (Integer, [s])
 5                   deriving Show

    pSucceed v (n, input) = [Succeed v (n, input)]
    pFail      (n, input) = [Until n]

10  pSym a  (n, (b:rest)) = [if a == b then Succeed b (n+1, rest) else Until n]
    pSym a  (n, []      ) = [                                            Until n]

    (p <|> q) ninput = p ninput ++ q ninput

15  (p <*> q) ninput = let presult = p ninput
                       in  [ Until np | Until np <- presult]
                           ++
                           [ Succeed (pv qv) nqrest
                           | Succeed pv nprest <- presult
20                         , Succeed qv nqrest <- q nprest
                           ]
                           ++
                           [ Until nq
                           | Succeed pv nprest <- presult
25                         , Until nq          <- q nprest
                           ]

    parse p input = foldr1 best (p (0,input))
                    where a `best` b = if pos a > pos b then a else b
30                        pos (Until n)        = n
                          pos (Succeed _ (n,_)) = n
```

Listing 4: BasicCombinators1

head of the input stream, but unfortunately find something different. In this case there are basically two ways to make progress:

- *insert* the expected token at this position
- *delete* the unexpected token at this position and try again

So a first attempt for the function pSym is something of the following form:

```
  pSym a input@(b:rest)
     = if a == b then [(b, rest)]
       else [(a, input)]   -- pretend that the token was seen
            ++             --   or
            pSym a rest    -- delete the unexpected token from the input
                           -- and try again
  pSym a [] = [(a, [])]    -- pretend the token was there
```

```
  module BasicCombinators2 where
  infixl 3 <|>
  infixl 4 <*>

5 data Step  = Okstep
             | Failstep
  type Steps = [Step]

  failsalways = Failstep:failsalways
10
  type Parser s a = [s] -> [(a, Steps, [s])]

  pSucceed v input = [(v          , []            , input)]
  pFail     input = [(undefined, failsalways, input)]
15
  pSym a input@(b:rest)
    = if a == b
      then [(b, [Okstep], rest)]
      else [(a, [Failstep], input)]
20                   ++
          [(v, Failstep:steps, r) | (v, steps, r) <- pSym a rest]

  pSym a [] = [(a, [Failstep], [])]

25 (p <|> q) input = p input ++ q input

  (p <*> q) input = [ (pv qv, psteps++qsteps, rest)
                    | (pv    , psteps        , qinput) <- p input
                    , (qv    , qsteps        , rest  ) <- q qinput
30                  ]

  parse p input = foldr1 best (p input)
```

Listing 5: BasicCombinators2

On a close inspection one sees that this version of pSym actually constructs all possible inputs and and tries to match those to the given input.

The question that arises now is which parse to select; the given approach generates a tremendous number of parses, most of them corresponding to heavily mutilated versions of the input. So in our next step in the development we combine each result with information about its quality. For this we introduce a new data type that represents a parsing history as a sequence of acceptance (Okstep) and correction (Failstep) steps. A first attempt of this approach is given in listing 5. Instead of passing around an integer indicating how many steps were successfully taken in the past, each result is now tupled with how many steps were successfully taken in its recognition: so "counting" starts at the

end of the recognized part, instead of at the beginning. Of course the length of the list of steps will, once we return the value at the root, be the same as the integer tupled with the result before, that indicated how far we went.

We have not given here the function `best` yet, since this solution is erroneous anyway. A moment of thought will show that the final result of the parsing process is, since most languages are infinite, likely to be infinite too: the given input can, with a sufficient number of correcting actions, be changed into each of the sentences of the language. A further shortcoming of this approach is that after recognizing a `p<*>q`, each sequence of steps of `p` is prefixed to many `q`-steps. As a consequence many resulting sequences will have long common prefixes, which makes the comparison process prohibitively expensive. Also the blunt way of concatenating the steps is extremely costly, since building sequences by repeated concatenation of parts tends to be quadratic in the length of the list. It is tempting to first select the best element from the different `q`-solutions, and only append that solution to the corresponding `p`-solution, but that is wrong too: a short `q`-solution may lead to a better starting point for a parser that is invoked after `p<*>q`. Finally we are likely not to get a result at all, since before we are able to construct part of a result of the form `psteps++qsteps` we have to find an appropriate `qsteps`. As soon as we get in an infinite branch of the construction process no more solutions will become available!

8   Continuation Based Parser Combinators

A new insight has now popped up, however. If we could, after recognizing a `p<*>q`-structure, peek into the future to see what the consequences are of taking specific alternatives, we could report back to our caller about it's future by combining our local information with that information about our future. Experienced functional programmers will smell the use of continuation based techniques here. So the question that now arises is: What should be our new `Parser`-type?

Before answering this question let us look for a moment at the data structures used in a conventional description of a top-down parser for context free languages:

- The stack of the symbols recognized thus far, capturing the history of the parsing process and to be used in the construction of the final result.
- The state of the parser, consisting of a stack of symbols still to be recognized.
- The unconsumed part of the input.

In a continuation passing style all such data has to be passed around on by means of parameters. So a parser that should recognize something of type `a`, takes the following arguments:

- a *history*, that may be combined with the recognized value of type `a` into a new history of type `b`, that is to be passed on to
- a *future* that will eventually construct something of type `d`, when passed
- the *remaining input*

So our parser should be of the following type:

```
Future s b d -> History a b -> [s] -> Result d
```

with appropriate definitions for `Future`, `History` and `Result`. The obvious choice for the history is

```
type History a b = a -> b ,
```

because this is the simplest type that can hold values that convert `a`'s into `b`'s. The type for the future does not leave us much choice either, since it has to accept the newly constructed history of type `b` and the remaining input of type `[s]` and should produce something tat contains a type `d` value:

```
type Future s b d = b -> [s] -> Result d
```

We might have taken the liberty to let the future return a value of type `Result' s d`, but that does not turn out to be necessary.

Finally let us try to design the type `Result d`. A parser gets this value back from the future (i.e. the called continuation), and has to return it on to its past (i.e. its caller): the type `Result` in listing 6, has been designed in such a way that it both represents s final result and the parsing steps in finding that result. The `Cost` field will, for the time being, be chosen to be 0 when an input token was successfully recognized, and 1 whenever a symbol was inserted or deleted.

Note that we do not use a conventional continuation passing style, in which continuation calls are usually a so-called tail-calls, in which the result of the continuation call becomes the result of the calling function. Here we take the opportunity to add some information to the result, before returning it to our own caller: i.e. we add information about the parsing steps that were taken between being called and calling our continuation.

In listing 6 we present the final solution, and we will go through this solution step by step. In lines 1-9 we repeat the types introduced thus far. The type `Parser` is a bit peculiar since it contains two type variables that do not occur as a parameter. In many extensions of Haskell98 however it is possible to denote such universally quantified types, provided we locate them inside a `newtype` definition; here such a newtype definition is for all practical purposes equivalent to a normal type definition, with the exception that it introduces an extra constructor (`P`).

The function definition of `pSucceed` is straightforward: it combines its history `h` with the witness of its success (`v`) into a new history (`h v`), and passes this, together with the `input` on to its own future `f`. The function `pFail` simply returns an infinite list of fail steps.

The sequential composition `p<*>q` starts the parsing of the composition of `p` and `q` by calling `p`. Since after `p` first `q` and then their common future `f` should be parsed, we construct the future for `p` by partially applying `q` to `f`. The history of the call to `p` should be such that when it is applied to `pv` its result is a function, that when applied to `qv` results in `h (pv qv)`, and we see that the function (`h .`) does the job since `h (pv qv) == (h . pv) qv == (h .) pv qv`. For `p<|>q` we simply call both alternatives with the same history and future and choose the `best` result of the two.

The function `pSym` checks the first symbol of the input; if this is the sought symbol the continuation `f` is called with the new history (`h s`) and the rest of the input `ss`. Once this returns a result, the fact that this was a successful parsing step is recorded by applying `Step 0` to the final result, and that value is returned

```
   newtype Parser s a  = P (forall b, d
                             .    Future s b d
                              -> History a b
                              -> [s]
5                             -> Result d
                             )
   type Future s b d = b -> [s] -> Result d
   type History  a b = a -> b
   type Cost         = Int
10 data Result d      = Step Cost (Result d)
                     | Stop d
   --THE PARSER COMBINATORS
   pSucceed v       = P (\ f h input -> f (h v)                    input  )
   pFail            = P (\ f h input -> fails where fails = Step 1 fails)
15 (P p) <*> (P q) = P (\ f h input -> p (q f) (h .)              input  )
   (P p) <|> (P q) = P (\ f h input -> p f h input `best` q f h input  )

   pSym a      =
    \ f h -> let pr = \ input ->
20                    case input
                      of  (s:ss)  ->
                             if s == a then Step 0 (f (h s) ss  )
                                       else Step 1 (pr      ss  )  -- delete
                                               `best`
25                                       Step 1 (f (h a) input) -- insert
                          []       ->      Step 1 (f (h a) input) -- insert
            in pr

   -- SELECTING THE BEST RESULT
30 best :: Steps v -> Steps v -> Steps v
   left@(Step l aa) `best` right@(Step r bb)  = if     l < r then left
                                                   else if l > r then right
                                                   else Step l ( aa `best` bb)
   (Stop v   ) `best` _            = Stop v
35 _           `best` (Stop v)     = Stop v
```

Listing 6: BasicCombinators3

to the caller. If the sought symbol is not present both possible corrections are performed. Of course, when we have reached the end of the input, the only possible action is to try to insert the expected symbol.

We have kept the most subtle point for desert, and that is the definition of the function best. Its arguments are two lists with information about future parsing steps, and ideally it should select the one containing the fewest corrections. Since computing this optimal solution implies computing all possible futures that at least extend to the end of the input, this will be very expensive. So we choose

to approximate this with a greedy algorithm that selects that list that is better than its competitor at the earliest possible point (i.e. that list that comes first in a lexicographic ordering). We should be extremely careful however, since it may easily be the case that both lists start with an infinite number of failing (`Step 1`) steps, in which case it will take forever before we see a difference between the two. The function `best` is this carefully formulated such that, even when a decision has not been taken yet, already information is returned the result. In this way we are able to do the comparison and the selection on an incremental basis. We just return the common prefix for as far as needed, but postpone the decision what branch is to be preferred, and thus what value has to be returned, as long as possible. Since this partial result will most likely again be compared with other lists, most such lists will be discarded before the full comparison has been made and a decision has been taken! It is this lazy formulation of the function `best` that converts the underlying depth-first backtracking algorithm into one that works on all possible alternatives in parallel; it are all the calls to `best`, and the demand driven production of partial results, that are driving the overall computation. In the next section we will complete our discussion by describing how to call our parsers, and what functions to pass to our initial parsers.

9   Further Details

Having constructed a basic version of the parser combinators, there still is some opportunity for further polishing. In this section we will describe how error reporting may be added and how constructed parsers have to be called. We finish by pointing out some subtle points where the innocent user might be surprised.

9.1   Error Reporting

Despite the fact that we have managed to parse and correct erroneous inputs, we do not know yet, even though we get a result, whether any or what corrections were made to the input. To this end we now slightly extend the parser type once more as show in listing 7. All parsing functions take one extra argument, representing the corrections made in constructing its corresponding history. When further corrections are made this fact is added to the current list of errors. Errors are represented as a value of type `Errors s -> Errors s`, so actually we are passing a function that may be used to construct the `Errors s` that we are interested in. As a consequence all combinators change a bit, in the sense that they all get an extra argument that is just passed on to the called continuations. Only when corrections are applied in order to be able to continue we have to do something, and we have seen that this is all local to the function `pSym`. The new version of `pSym` is given in listing 7.

The errors are returned in the form of a data type, that may be convert into a string using the function show.

```
   newtype Parser s a
    = P (forall b, d.   Future s b d
                    -> History a b
                    -> Errs s
5                   -> [s]
                    -> Result s d
        )

   type Errs s = (Errors s -> Errors s)
10
   data Errors s = Deleted  s s       (Errors s)
                 | Inserted s s       (Errors s)
                 | InsertedBeforeEof s (Errors s)
                 | DeletedBeforeEof  s (Errors s)
15               | NotUsed [s]

   instance Show s => Show (Errors s) where
     show (Deleted  s w e      ) = msg "deleted "  s (show w)          e
     show (Inserted s w e      ) = msg "inserted " s (show w)          e
20   show (InsertedBeforeEof s e) = msg "inserted " s "(virtual) eof" e
     show (DeletedBeforeEof  s e) = msg "deleted "  s "(virtual) eof" e
     show (NotUsed [])           ) = ""
     show (NotUsed ss           ) = "\nsymbols starting with "
                                    ++ show (head ss)
25                                  ++ " were discarded "

   msg txt sym pos resterrors = "\n" ++ txt ++ show sym
                                ++ " before " ++ pos ++ show resterrors

30 -- the new version of pSym
   pSym a f h
     = P( let pr =
            = \ errs input
             -> case input
35              of  (s:ss) ->
                       if s == a
                       then Step 0 (f (h s) errs ss  )
                       else Step 1 (pr (errs . (if null ss
                                                then DeletedBeforeEof s
40                                               else Deleted s (head ss)
                                             )        )
                                       ss
                                    )
                                               `best`
45                     Step 1 (f (h a) (errs .Inserted a s        ) input)
                   [] -> Step 1 (f (h a) (errs .InsertedBeforeEof a) input)
          in pr )
```

Listing 7: ErrorReporting

9.2   How to Stop?

Before we can actually parse something we still have to decide what kind of continuation to pass to the parser corresponding to the root symbol of the grammar. We could have defined this function once and for all, but we have decided to provide some extra flexibility here. In our expression example we have already seen that one may not only want to stop parsing at the end of the input. The function `parse` takes a Boolean function that indicates whether parsing has reached a point that may be interpreted as the end of the input. If this is the case then no error message is generated. Otherwise it is reported that there were unconsumed tokens (which are assumed to have been deleted). Furthermore not only the witness of the parsing is stored in the resulting value, but also the accumulated errors and the remaining part of the input.

```
parse (P p) user_eof input
   = let eof v e input
         = if user_eof input
           then       (Stop (v, input, e  (NotUsed []   )))
           else foldr  (\ _ t -> Step 1  t)
                       (Stop (v, input, e  (NotUsed input)))
                       input
       stepsresult (Step _ s)  = stepsresult s
       stepsresult (Stop v)    = v
    in stepsresult ( p    -- the parser
                     eof  -- its future
                     id   -- its history
                     id   -- no errors thus far
                     input )
```

10   Pitfalls

One of the major shortcomings of programming in the way we do, i.e. with many higher order functions and lazy - and possibly infinite- data structures is that a feeling for what is actually going on and how costly things are, gets easily lost.

The first example of such a pitfall occurs when in the input a simple binary operator is missing between two operands. Usually there are many operators that might be inserted here, all leading to a correct context free sentence. Unfortunately the system is, without any further help, not able to distinguish between all these possible repairs. As a consequence it will parse the rest of the program once for each possibility, frantically trying to discover a difference that will never show up. If several of such repairs occur the overall parsing time explodes. It is for this reason that we have included the cost of a repair step in an `Int`-value. If this phenomenon shows up, all operators but one can be given a higher insertion cost, and as a consequence one continuation is immediately selected without wasting time on the others. In the complete version of our library[1] furthermore

---

[1] http://www.cs.uu.nl/groups/ST/Software/Parse/

the lookahead for the `best` function is limited somewhat, once it has been discovered that two sequences that both contain a repair step are being compared. It is clear that for the function `best` there is still a lot of experimentation possible.

A second problem arises if we have a grammar that may need a long lookahead in order to decide between different alternatives. An example of such a grammar is:

```
p =     count <$> pSym 'a' <*> l <*> pSym 'b'
    <|> count <$> pSym 'a' <*> l <*> pSym 'c'
    <|> count <$> pSym 'a' <*> l <*> pSym 'd'
    <|> count <$> pSym 'a' <*> l <*> pSym 'e'
    where count _ n _ = n+1
```

Here some heavy backtracking will take place, once we try to parse an input that starts with many `a`'s. Of course this problem is easily solved here by rewriting the grammar a bit using left-factorization, but one may not always be willing to do so, especially not when semantic functions like `count` are different. Fortunately it is possible to exchange time for space as we will show in the next section, and this will be done by the combinators from our library.

11   Speeding Up

Despite its elegance, the process of deciding which alternative to take is rather expensive. At every choice point all possible choices are explored and usually all but one are immediately discarded by the calls to `best`. In most parsers the decision what to do with the next input symbol is a simple table lookup, where the table represents the state the parser is in. Using the technique of tupling we will sketch how in our parsers we may get a similar performance; since the precise construction process is quite complicated, time and space forbid a detailed description here. Furthermore we assume that the reader is familiar with the construction of LR(0) items, as being described in every book on compiler construction.

The basic data structure around which we center our efforts is the trie data type `Choices` in listing 8. This data structure describing a parser is tupled with that of concrete parser using the function `mkParser`; this is an extension of the techniques described for LL(1) grammars in [5].

The four alternatives of `Choices` represent the following four cases:

Found In this case there is no need to inspect any further symbols in the input; the parser `P s a` should be applied at this point in the input.

Choose Based on the look-ahead inspected thus far it is not yet possible to decide which parser to call, so we have to use the `[(s, Choices s a)]` structure to continue the selection process. If the next input symbol however is not a key in this table the corresponding `P s a` is the error correcting parser that applies here. This state corresponds to a pure shift state, in LR(0) terminology.

```
data Choices s a = Choose (P s a)     [(s,  Choices s a)]
                 | Split  (P   a)           (Choices s a)
                 | End    (P s a)
                 | Found  (P s a)           (Choices s a)

cata_Choices (sem_Choose, sem_Split, sem_End, sem_Found)
  = let r
      = \ c -> case c of
              (Choose p     csr) -> sem_Choose p [ (s, r ch)
                                                 | (s,   ch) <- csr
                                                 ]
              (Split  p cs     ) -> sem_Split  p (r cs)
              (End    p        ) -> sem_End    p
              (Found  p cs     ) -> sem_Found  p (r cs)
    in r

mkparser cs
 =let choices
      = cata_Choices
         (\ (P p) css                            -- shift
            -> \inp ->  case inp
                        of []     ->  p
                           (s:ss) ->  case find cmp css s of
                                      Just (_, cp)  -> (cp ss)
                                      Nothing       ->  p
         ,\ (P p) css -> \inp -> p `bestp` (css inp)  -- reduce and shift
         ,\ (P p)     -> \_   -> p                -- reduce
         ,\ (P p) cs  -> \_   -> p                -- only candidate
         ) cs
 in (cs, (P (\ f h e input -> (choices input) f h e input))) -- tuple

p `bestp` q  = ( \ f h e input -> p f h e input `best` q f h e input )
```

Listing 8: MakeParser

End This corresponds to a pure reduce state. The parser P s a is the parser we
   have to call, and we can be sure it will succeed since in the selection process
   we have seen all the symbols of its right-hand side.
Split This corresponds to a shift-reduce state and we have two possibilities. So
   we continue with the selection process in the Choices s a component, and
   apply both the parser found there and the P s a component of the Split,
   and compare the results using the function best.

The function cata_Choices is a homomorphism over the initial data type
Choices, and thus replaces all data type constructors in a systematic way with
one of the four functions given as argument. The function mkparser is defined
that tuples a Choices s a structure cs with a real parser. This demonstrates

another important technique that can often be applied when writing functions that can be seen as an interpreter: *partial evaluation*. In our case the "program" corresponds to the choice structure, and the input of the program to the input of the parser. The important step here is the call to `cata_Choices` that maps the choice structure to a function that chooses which parser to call. This resulting function is then used in the actual parser to select the parser that applies at this position (`(choices input)`), which parser is then called: `(\e f h e input -> (choices input) f h e input)`.

12   Conclusions and Further Reading

We have shown how, by making use of polymorphism, type classes, higher order functions and lazy evaluation we can write small libraries for constructing efficient parsers. In defining parsers with this library all features of a complete programming language are available.

Essential for the description of such libraries is the availability of an advanced type system. In our case we needed the possibility to incorporate universally quantified types in data structures.

We expect that, with more advanced type systems becoming available, special purpose tools will gradually be replaced by combinator libraries.

References

1. Graham Hutton and Erik Meijer. Monadic parser combinators. *Journal of Functional Programming*, 8(4):437–444, July 1998.   pages 1
2. Fokker J. Functional parsers. In Jeuring J. and Meijer E., editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 1–52. Springer-Verlag, Berlin, 1995.   pages 1
3. Hammond K. and Peterson J. (eds).   Haskell 1.4 report.   Available at: `http://haskell.org!`, May 1997.   pages 1
4. Milder R., Tofte M., and Harper R. *The Definition of Standard ML*. MIT Press, 1990.   pages 1
5. Swierstra S.D. and Duponcheel L.   Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.   pages 1, 17

Table of Contents