Deterministic, Error-Correcting Combinator Parsers

S. Doaitse Swierstra and Luc Duponcheel

Dept. of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
e-mail swierstra@cs.ruu.nl and luc@cs.ruu.nl

Abstract. We show how error-correcting, deterministic, combinator parsers can be constructed for grammars which have the LL(1) property. The normal disadvantages of conventional combinator parsers, such as their lack of speed and their poor error reporting, are remedied. Parsers constructed with these combinators are implicitly specialised for the grammar they describe. Because of this property the combinators act as a parser generator for LL(1) grammars. The techniques used to construct the combinators illustrate how partial parametrisation and careful ordering of computations can be used to achieve implicit partial evaluation in a lazy language.

1 Introduction

Somewhere along the road, everyone who is introduced to the use of recursion in imperative languages gets introduced to recursive descent parsers. In a similar fashion everyone who gets introduced to functional programming gets introduced to combinator parsers. In both cases one is impressed by the elegant formulation of the underlying ideas. When compared to the expressive power of recursive descent parsers, combinator parsers are even more expressive. What they allow in addition is the elegant formulation of parsers corresponding to the conventional regular extensions of grammatical formalisms. For an introduction to such combinator based parsers one may consult [1, 2]. In this paper we will follow the line taken by Fokker[1], and stick to his notation as closely as possible.

As soon as one starts to use normal combinator based parsers, disappointments arise. The parsers constructed may be unexpectedly slow, and since combinators are usually used to describe non-deterministic parsers, they do not perform any form of error-reporting, let alone error-recovery. Even the smallest mistake in the input may lead to a lengthy parsing process which finally produces an empty list of successful parses, with no clue as to where where the mistake is located. Although it is possible to extend the parsers in such a way that they report the furthest point in the input reached during the backtracking process([8, 5]), it is still a nuisance to have to run the parser over and over again in order to discover all the mistakes in the source. Furthermore no other information is usually given about the nature of the error.

For recursive descent parsing, techniques have been developed to incorporate forms of error recovery. For example, in the original Pascal compiler a simple error recovery mechanism was introduced in an ad hoc fashion. In this paper, based on the concept of deterministic top-down parsers (i.e. based on grammars which have the so-called LL(1) property), we will develop an new set of combinators which will construct parsers which repair the input, produce useful error messages, and continue to parse the rest of the input.

An interesting feature of our approach is that we only redefine the basic combinators for *sequencing*, *choice*, and the parsers which recognise a single *symbol* and the *empty* string. All the other combinators, which are expressed in terms of these basic combinators, can thus be used without any further change. As a result of this approach there is no need for a separate parser generator, which performs an elaborate analysis of the grammar, and then generates code, which later has to be either incorporated in a program as text or has to be linked as a separate module.

We will develop the new combinators in a stepwise fashion, starting from the conventional combinators. In order to do this we will rely on the use of *constructor classes*. Thus all our programs will be given in Gofer, a language which closely resembles Haskell, which supports this abstraction feature. Our use of constructor classes is relatively self-contained. The interested reader can see [3] for a more detailed description of constructor classes.

This paper can also be used as a literal program (all the lines prefixed with > together form a program).

In the conclusion we will elaborate on the general strategy which we apply in the paper and argue that it is much more widely applicable.

2 Conventional Combinator Parsers

We start with a quick review of conventional combinator parsers and point out some of their problems.

A functional parser is a function which takes as input a sequence of symbols, and returns a list of successes, i.e. a list of pairs in which the first component of the pair represents the semantic value corresponding to the segment of input which has been consumed by the parser, and the second component of the pair represents the part of the input which was not used, and should thus be consumed by subsequent parses. If the list contains more than one pair, this indicates that the input can be parsed in several possible ways.

We will not be very specific about the exact type of input which we will be able to parse. We must be able to compare symbols, and to use them as an index in a data structure, thus an equality on symbols is needed. For reasons of efficiency we require an ordering on symbols, and we also require that symbols have a textual representation for error reporting. We start with a `class`-definition which captures the requirements above.

```
> class Symbol s
> instance (Ord s, Text s) => Symbol s
```

Parsers can be combined into new parsers using combinators. The basic parsers are: `empty` for recognising the empty string and `symbol` for recognising a single symbol. The basic combinators for constructing new parsers from existing ones are: `<*>` for sequencing and `<|>` for choice (the combinator `<?>` will be explained later on). We group these combinators together in a `class`-definition. In each step we develop ever more complicated `instance`'s of this class.

```
> infixl 2 <?> ; infixl 3 <|> ; infixl 4 <*>

> class Parsing p where
>   empty  :: Symbol s => a -> p s a
>   symbol :: Symbol s => s -> p s s
>
>   (<?>) :: Symbol s => p s a -> (a,String) -> p s a
>   (<|>) :: Symbol s => p s a      -> p s a -> p s a
>   (<*>) :: Symbol s => p s (b->a) -> p s b -> p s a
```

Traditionally the result of a parser combinator expression for sequencing: `p <*> q` is the Cartesian product of the results of parsers `p` and `q`. Note that we have taken a slightly different approach, following Rojemo[8], in which the `p`-parser returns a function which is applied to the result of the `q`-parser. Note furthermore that we have chosen the `<*>` combinator to be left associative, so that associativity corresponds to associativity of application (i.e. `f x y = (f x) y`).

Using the basic parsers and the combinators other parsers and combinators can be constructed. The combinator `<$>` takes a function and a parser and applies the function to the result of the parse; in this way a meaning can be given to the recognised structure, avoiding the explicit construction of a parse tree, which usually would be inspected later on in a syntax directed translation process. The combinator `<$>` is defined in terms of `(<*>)` and `empty`. A combinator `opt` for optional input can be defined in a similar way in terms of `(<|>)` and `empty`.

```
> infixl 4 <$> ; infixl 2 `opt`

> (<$>) :: (Parsing p, Symbol s) => (b->a) -> p s b -> p s a
> f <$> p = empty f <*> p

> opt :: (Parsing p, Symbol s) => p s a -> a -> p s a
> p `opt` v = p <|> empty v
```

Recursion can used to construct more involved parsers and combinators, like the combinator for recognising the same structure many times or the combinator for recognising a chain of operands separated by operators:

```
> many p = (\a as -> a:as) <$> p <*> many p `opt` []

> chainr x op = (\x f -> f x) <$> x <*> f where
>  f = (\op x -> (`op` x)) <$> op <*> chainr x op `opt` id
```

Exercise 1. What are the types of `many` and `chainr`? The `chainr` combinator is used for right associative operators. Define a `chainl` combinator which is used for left associative operators.

Exercise 2. Write a simple combinator parser which can be used to test the correctness of `chainr` and `chainl`.

Finally we give an example grammar for statements which is described using the `Parsing` class. The grammar just returns the parsed input string. The grammar makes use of a parser `sym` which is a variant of the parser `symbol`.

```
> sym  :: (Parsing p, Symbol s) => s -> p s [s]
> sym s = (\x -> [x]) <$> symbol s
```

In this grammar statements are separated by semicolons. To keep things simple we consider only two compound statements: if-statements and while-statements. Since we wish to avoid lengthy explanations about lexical issues in this example we assume that reserved words are just single upper case characters. The other basic structures, conditions and assignments, are single lower case characters.

```
> stats :: Parsing p => p Char String
> stats  = chainr stat ((\s x y ->x++s++y) <$> sym ';')

> stat :: Parsing p => p Char String
> stat  =     if_stat
>         <|> while_stat
>         <|> assignment
>         <?> ("<stat>",inserted "<stat>")

> if_stat :: Parsing p => p Char String
> if_stat
>  = (\i c tp ep f -> i++c++tp++ep++f) <$>
>    sym 'I' <*> cond <*> then_part <*> else_part <*> sym 'F'

> then_part :: Parsing p => p Char String
> then_part  = (\t ss -> t++ss) <$> sym 'T' <*> stats

> else_part :: Parsing p => p Char String
> else_part  = (\e ss -> e++ss) <$> sym 'E' <*> stats `opt` []

> while_stat :: Parsing p => p Char String
> while_stat
>  = (\w c d ss o -> w++c++d++ss++o) <$>
>    sym 'W' <*> cond <*> sym 'D' <*> stats <*> sym 'O'

> assignment :: Parsing p => p Char String
> assignment  = sym 'a'

> cond :: Parsing p => p Char String
> cond  = sym 'c'
```

The `<?>` combinator is used for repairing the input and describing what error messages should be constructed when a statement cannot be recognised in the input. We will see the details of this later on. Before going on we illustrate the general form of the output we are aiming for. By invoking the parser **stats** in different ways we obtain different results. The function **invokeEmpty** tests if the parser can recognise the empty string (see 3.1). The function **invokeFirst** computes the set of symbols the parser is willing to accept as the first symbol of the input sequence (see 3.2). The function **invokeDet** invokes the parser in a deterministic way. This function does not deal with errors yet (see 3.3). Finally, the function **invokeErr** invokes the parser that deals with error recovery.

```
? invokeEmpty stats
False

? invokeFirst stats
IWa

? invokeDet stats "WcDaO"
WcDaO

? invokeDet stats "WcDaE"

Program error: Illegal input symbol: 'E'

? display (invokeErr stats "WcDaE")
"WcDaO"
'E' deleted
'O' inserted

? display (invokeErr stats "WcDIcTEaO")
"WcDIcT<stat>EaFO"
"<stat>" inserted
'F' inserted
```

Note how the error messages indicate in a clean fashion what went wrong. The following two auxiliary functions are useful for generating the error messages above.

```
> inserted s = show s ++ " inserted\n"
> deleted  s = show s ++ " deleted\n"
```

Exercise 3. Define a small grammar for your own favorite language using combinator parsers.

2.1 Non-Deterministic Parsing

The fact that combinator parsers are non-deterministic has several unfortunate consequences. In the first place it implies that even though a successful alternative has been recognised, all other alternatives must also be tried, thus leading to a slow parsing process. Worse however is the fact that because backtracking may take us back to the very beginning of the parsing process, usually leading to no further interesting results, the whole input has to be kept, and cannot be discarded before parsing has been completed. Some attempts have been made to adapt the parsing process to give preference to longer parses, but this does not solve the problem in all cases [8].

Since we are interested in top-down deterministic parsing we turn to the theory of LL(k) grammars: for such grammars it is always possible to determine what alternative to take, based on inspecting at most the first k input symbols. We will show how the information necessary for making such a choice in case of an LL(1) grammar can be computed and used in the construction of such deterministic parsers.

2.2 No Error Recovery

A further disadvantage of brute force non-deterministic parsing is it is cumbersome to find the latest point where an error has occurred. In addition since this discovery is usually made only after all other alternatives have been tried, (most of which do not reach as far into the input) we must somehow bring back the parser back to the state where it was when reaching the ultimate point before performing some sort of error correction on the input in order to continue from that point on with the parsing process. This is not so easily done, and when done, will be costly.

So we reach the conclusion that it is a good idea to shy away from non-deterministic parsers whenever possible, and to make good use of the fact that one knows that the underlying grammar has some useful properties.

In the next section we will derive the deterministic parser combinators in a sequence of steps. In the section following that error recovery will be added.

3 Deterministic parsing

To find out if a grammar is LL(1) we must determine several things about that grammar. First we must be able to compute whether a parser based upon that grammar can accept the *empty string*. Second we must be able to compute the *first sets* which contain for each alternative the symbols it is prepared to accept first, and third we must be able to compute the *follow sets* which contain for each nonterminal the symbols which may follow that nonterminal in a derivation.

Based on this information we can predict whether a deterministic choice can always be made. Since we are constructing our parsers as functions we do not have an explicit representation of the grammar available. This will make it impossible for us to compute the follow sets, so we cannot guarantee that when actually parsing there will never be more than one alternative which may be chosen. We will however compute *dynamic follow sets*, which contain the symbols which can actually follow the non-terminal at hand, given the parsing history thus far.

In a sequence of steps we will now start to construct such parsers.

3.1 Computing the Emptyness of Parsers

The basic idea we follow is that we do not compute parsers, but parsers tupled with some of the properties of the corresponding grammatical construct. The important data type we will use (which will store both the actual parsers and their properties) is always of the abstract form `P s a`. Although in this subsection this type looks more complicated than needed, (i.e. we have incorporated some type parameters `s` and `a` without motivating why) this will allow us, later on, to avoid having to repeatedly change the type definitions of the basic functions. We will always construct parsers by defining functions `pempty`, `psymbol`, `perr`, `palt` and `pseq`: they implement the members `empty`, `symbol`, `(<?>)`, `(<|>)` and `(<*>)` of the `Parsing` class.

In this section we will illustrate how to compute whether a parser can accept the empty string or not. We do this by constructing the first instance of our `Parsing` class. In this instance the type constructor `Empty` plays the role of the abstract `P` above. For the purpose of using this parser instance (used in the function `invokeEmpty`) we do not need any input sequence or for that matter any value to be returned by the parser. The instance performs a static analysis of the grammar. The types of the abstract type parameters for symbols, `s`, and values, `a`, are ignored. Here we go,

```
> type Empty s a = Bool in
>  eempty, esymbol, eerr, ealt, eseq,
>  invokeEmpty,
>  dpalt, dpseq,
>  combine

> instance Parsing Empty where
>   empty  = eempty
>   symbol = esymbol
>
>   (<?>) = eerr
>   (<|>) = ealt
>   (<*>) = eseq

> eempty  :: a -> Empty s a
> esymbol :: s -> Empty s s
>
> eerr :: Empty s a      -> (a,String) -> Empty s a
> ealt :: Empty s a      -> Empty s a  -> Empty s a
> eseq :: Empty s (b->a) -> Empty s b  -> Empty s a

> eempty  _ = True
> esymbol _ = False
>
> eerr _ _ = False
> ealt     = (||)
```

```
> eseq     = (&&)

> invokeEmpty  :: Empty s a -> Bool
> invokeEmpty p = p
```

Apart from all the type declarations (which are needed to turn `Empty s a` into an abstract data type) all we have actually said is that a sequence accepts the empty string if both of its components do, and that a choice accepts the empty string if at least one of the components does. So far nothing very complicated.

One might even wonder whether it is really that simple, so we spend some time to see why this definition works. Both conventional combinator parsers and our parsers only work for grammars which are *neither directly nor indirectly left-recursive*. For the function `stat` it holds that before reaching a recursive invocation of `stat` always at least one symbol will be accepted, e.g. an `I` or a `W`. Now because `&&` *does not evaluate its second argument when its first argument evaluates to* `False` we avoid an infinite recursion. Our definition is well-defined, exactly when the grammar it implements is not left-recursive! We are thus saved by the lazy semantics of our programming language.

3.2 Computing First Sets

As explained earlier the *first set* of a parser is the set of symbols which it is prepared to accept as the first symbol of the input. We tuple this computation with the computation of the emptyness which we described before since we need this information in the function `fseq` for computing the first set of a sequential composition. If the first component is empty the second component can actually be the one which accepts the first input symbol (this idea is encoded in the function `combine`).

```
> combine :: Symbol s => Empty s a -> [s] -> [s] -> [s]
> combine e s1 s2 = s1 `union` (if e then s2 else [])
```

None of the other functions dealing with first sets need the emptyness information. The type constructor `First` plays the role of the abstract `P`. For the purpose of using this parser instance (used in the function `invokeFirst`) we again do not need any input sequence or any value to be returned by the parser. The instance performs a static analysis of the grammar. The type of values, `a`, is thus ignored.

```
> type First s a = [s]

> fempty _  = []
> fsymbol s = [s]
>
> ss `ferr` _   = ss
> ss `falt` ss' = ss `union` ss'
```

We are now ready to construct the second instance of our `Parsing` class:

```
> type EmpFir s a = (Empty s a, First s a) in
```

```
>  efempty, efsymbol, eferr, efalt, efseq,
>  invokeFirst,
>  dpalt, dpseq

> instance Parsing EmpFir where
>   empty  = efempty
>   symbol = efsymbol
>
>   (<?>) = eferr
>   (<|>) = efalt
>   (<*>) = efseq

> efempty  :: Symbol s => a -> EmpFir s a
> efsymbol :: Symbol s => s -> EmpFir s s
>
> eferr :: Symbol s => EmpFir s a      -> (a,String) -> EmpFir s a
> efalt :: Symbol s => EmpFir s a      -> EmpFir s a -> EmpFir s a
> efseq :: Symbol s => EmpFir s (b->a) -> EmpFir s b -> EmpFir s a

> efempty  v = (eempty  v, fempty  v)
> efsymbol s = (esymbol s, fsymbol s)
>
> (e,  f)  `eferr`  x        = (e  `eerr` x,  f  `ferr` x)
> (e1, f1) `efalt`  (e2, f2) = (e1 `ealt` e2, f1 `falt` f2)
> (e1, f1) `efseq` ~(e2, f2) = (e1 `eseq` e2, f1 `fseq` f2)
>                               where fseq = combine e1

> invokeFirst  :: Symbol s => EmpFir s a -> First s a
> invokeFirst (_,f) = f
```

There is a subtlety in this example we want to point out. Notice that when straightforwardly defining the `efseq` operator, pattern matching is performed on both arguments. Due to the fact that matching against an undefined value diverges we have to take counter measures: hence the irrefutable pattern in the right hand side of the definition of the infix function `efseq`. Normally even if we do not use any of the information from either of the fields `e2` or `f2`, some computation is still being done when calling the function. In order to construct the cartesian product pattern matching is needed for the nested calls, and thus our computation ends up in an indefinite recursion as soon as the grammar is in some way recursive, even if it is not left-recursive.

3.3 Parsers

Now by using the first sets, deterministic parsers can be constructed. We again define a new type `DetPar`, which now contains a parser which parses the input deterministically upto the first error. This parser incorporates both empty and first information.

```
> type Input s = [s]
> type Follow s = [s]
> type DetParFun s a = Input s -> Follow s -> (a,Input s)

> type DetPar s a = (EmpFir s a, DetParFun s a) in
>  dpempty, dpsymbol, dperr, dpalt, dpseq,
>  invokeDet

> instance Parsing DetPar where
>   empty  = dpempty
>   symbol = dpsymbol
>
>   (<?>) = dperr
>   (<|>) = dpalt
>   (<*>) = dpseq

> dpempty  :: Symbol s => a -> DetPar s a
> dpsymbol :: Symbol s => s -> DetPar s s
>
> dperr :: Symbol s => DetPar s a      -> (a,String) -> DetPar s a
> dpalt :: Symbol s => DetPar s a      -> DetPar s a -> DetPar s a
> dpseq :: Symbol s => DetPar s (b->a) -> DetPar s b -> DetPar s a
```

Notice that the parser now takes a second parameter of type `Follow s` in addition to the input sequence. When parsing, this will be the possible set of symbols, with which parsing can continue after this parser has succeeded. When the parser may accept the empty string, these symbols are used to decide whether this empty alternative should be taken. If the current input symbol is not in this set, apparently there is no reason to take this empty alternative. This will be especially important when we add error correction in the next section. We should not be too eager here to try to make some progress with parsing, without accepting an input symbol. Taking this empty alternative without justification lessens our possibilities for proper error correction.

The functions `dpempty` and `dpsymbol` are easy. Since we will assure elsewhere that a parser is only called in a situation where the first symbol on the input is indeed one of the symbols of its first set, or belongs to the dynamic follow set, there is no reason for `dpsymbol s` to test if the first symbol is `s`.

```
> pempty  v = \inp     _ -> (v,inp)
> psymbol s = \(_:inp) _ -> (s,inp)

> dpempty  v = (efempty  v, pempty  v)
> dpsymbol s = (efsymbol s, psymbol s)
```

The parser returned by `dperr` still does not do anything useful, but is included for reasons of consistency with the other declarations in this paper.

```
> dp `dperr` (v,_) = dp `dpalt` (efempty v, pempty v)
```

The parser returned by the combinator `dpalt` is the first parser for which the first and follow information is used. When the input sequence has become empty, empty parsers may still succeed, and finally lead to a succesful complete parse. If there are still symbols present a sequence of tests is being made to see if one of the constituents can be called, with progess of the parsing process guaranteed; when all alternatives fail apparently the furthest point in the input from which parsing cannot be continued has been reached. Notice that this can be done safely because the grammar is assumed to be LL(1).

```
> (ef1@(e1, f1), p1) `dpalt` (ef2@(e2, f2), p2)
>   = (ef1 `efalt` ef2, p1 `palt` p2) where
>  p1 `palt` p2 = p where
>   p [] follow =
>    if      e1 then p1 [] follow
>    else if e2 then p2 [] follow
>    else error "Unexpected Eof"
>   p inp@(s:_) follow =
>    if      s `elem` f1           then p1 inp follow
>    else if s `elem` f2           then p2 inp follow
>    else if e1 && s `elem` follow then p1 inp follow
>    else if e2 && s `elem` follow then p2 inp follow
>    else error ("Illegal input symbol: " ++ show s)
```

Notice however also that the process of making the choice is, albeit extremely simple, also rather expensive. It may take many evaluations of the function `elem`, before actually a choice has been made for a parser which accepts a symbol.

Exercise 4. Try to find a better approach for associating parsers with their first sets.

The definition of `dpseq` is surprisingly similar to previous sequence combinators. Its only interesting part is the computation of the follow set of the first parser `p1`, which depends on the follow set of the combined parser, the first set of `p2`, and the possible emptyness of `p2`:

```
> (ef1, p1) `dpseq` ~(ef2@(e2, f2), p2)
>   = (ef1 `efseq` ef2, p1 `pseq` p2) where
>  p1 `pseq` p2
>   = \inp follow ->
>       let comb = combine e2
>           (v1, inp1) = p1 inp (f2 `comb` follow)
>           (v2, inp2) = p2 inp1 follow
>       in  (v1 v2, inp2)
```

The function `invokeDet` returns the result of the the deterministic parser.

```
> invokeDet :: Symbol s => DetPar s a -> Input s -> a
> invokeDet (_,p) inp = case p inp [] of (a,_) -> a
```

Exercise 5. When some of the first sets are not disjoint the parsing process is apparently ambiguous at this point. Although a choice will be made due to the non-deterministic semantics of the programming language, this may be undesirable. Change the program in such a way that a useful error message will be produced. These error conditions can be computed statically, i.e. before parsing actually has begun. Be careful: since many combinators may have been introduced to construct new parsers it is very likely that the generated error message is not very helpful to the writer of the grammar since it comes from somewhere hidden in those combinators. A suggestion is to build up its path to the root symbol of the grammar, which will point out where the problem is located.

We may wonder why we cannot compute the union of all possible follow sets, so that we may statically decide if parsing will always be deterministic. In order to be able to compute this set however, we need access to all the places in the code where the function corresponding to this non-terminal is called. Unfortunately this information can be neither directly nor indirectly computed from the functional parsers since we do not have an explicit representation of the parsers at hand. The programs we write may look like a grammar, but keep in mind that they actually are function definitions. The fact that at a specific location we call a specific function, which corresponds to the use of a specific non-terminal in the right hand side of production, is information which is not available to us explicitly. For this we would need a language with some form of reflection.

Fortunately in parser generators for LL(1) grammars the follow sets are only used to issue warning messages at parser construction time, indicating that no deterministic choice might be made at a specific point when parsing, so we are not completely lost.

4 Error Correcting Parsers

We now come to the last instance of our `Parsing` class, which will contain the error-correcting parsers.

4.1 The Basic Idea

When encountering an unexpected symbol in the input there are two possibilities:

- it is superfluous
- there are symbols missing

In the first case the action to be taken is simple: delete the symbol from the input sequence and construct an error message. In the second case the missing symbols can be inserted. The only problem now is to decide which of these two alternatives we should take, and in case it is the second one which symbols to insert.

Our choice will be based on the usefulness of the unexpected symbol. If we foresee that, using our parsing history thus far, this symbol can be of use in the future, we decide to keep it and to insert a piece of input which brings us to the point where the useful symbol will actually be used, and parsing can continue normally. Suppose now, that the parsing of a `while_stat` has started. This occurs in the body of a parse in which the parsing of an `if_stat` has also started. Assume that parsing has reached the point indicated by the `^` in `WcDIc^aEa;aF`

Since the `T`-symbol is missing from the input, apparently the important symbols not to be skipped are the first symbols of an assignment, i.e. the `E`-, the `F`- and `O`- symbols, which are the first symbols of the tails of the currently active parsers. These are the first symbols of the stack of parsers which are still pending to accept their parts of the input, i.e. the tails of the entered while- and if-statements. We will call this set the *noskip set*. The set computed in this way is a superset of the follow set, in the sense that it not only contains the directly following symbols, but also future symbols, which have become visible by belonging to the first set of a pending parser. The strategy is based upon the assumption that it is unlikely that someone forgets to include in the input the first symbol of an alternative.

The recovery strategy is now as follows: if the next input symbol is part of this noskip set then the symbols which are occurring in front of this symbol are inserted in the input sequence (i.e. we take actions as if they were present), otherwise the current input symbol is deleted since there is no easy way to decide what to do with it. Since we do not always want to insert a complete set of symbols, which may be a highly artificial choice, we provide for the possibility to insert also non-terminal symbols in the input.

Exercise 6. The associativity of the sequential composition plays an important role here. Fortunately it is left associative. Can you indicate why this is fortunate?

4.2 Inserting Symbols

A non-terminal which accepts the empty string will never have to be inserted, because the parser can be pretend that it accepted the empty string. For those parsers for which this is not the case a pseudo alternative is introduced, which is taken when the corresponding non-terminal symbol is inserted. Furthermore provisions are made for error messages. The accumulated error messages are tupled with the remaining input, to form together the `state` of the parser.

Instead of computing a Boolean value indicating the possible emptiness we compute a value of the type defined in the data definition `EmptyDescr` below. The `IsEmpty` alternative of the definition contains the semantic value of an empty parser. The `Insert` alternative contains the semantic value of a nonempty parser that is inserted, and the error message which will be returned in the list of error messages when this parser is inserted. Here are all the relevant definitions:

```
> data EmptyDescr s a
>   = IsEmpty a
>   | Insert a String
```

```
> edempty  v = IsEmpty v
> edsymbol s = Insert s (inserted s)

> ederr (v,msg) = Insert v msg

> edp `edalt` edq =
>  case edp of
>   IsEmpty _  -> edp
>   Insert _ _ -> edq

> edp `edseq` edq =
>  case edp of
>   IsEmpty pv
>    -> case edq of
>        IsEmpty qv    -> IsEmpty (pv qv)
>        Insert  qv sq -> Insert (pv qv) sq
>   Insert pv sp ->
>    Insert (case edq of
>              IsEmpty qv   -> pv qv
>              Insert  qv _ -> pv qv
>           )
>           (case edq of
>              IsEmpty _    -> sp
>              Insert  _ sq -> sp++sq
>           )
```

Notice that, when combining two of these values in a sequential composition, attention again has been paid to the laziness of the definition. The definition of `edseq` may look a bit clumsy, but notice now that our requirement, which states that in the case that `edp` indicates that the first parser will never accept the empty sequence `edq` will not be evaluated, does not trivially hold. This problem can be solved by making the second alternative of the outer case expression more lazy. In the way it has been coded the `Insert`-constructor can be returned as soon as it has been decided that the first component is non-empty. Since we do not need the components of the second parser yet, using irrefutable patterns will again save us.

The combined values are statically computed as far as possible, which may add something to the efficiency of the generated parsers; again some form of partial evaluation is taking place here.

Notice that when a non-terminal is inserted in the input stream there is some liberty in choosing which alternative of that non-terminal should be chosen to represent the corresponding semantic action. We have deliberately chosen to give preference to the last one. A necessary condition for this to work well is that all possibly recursive non-terminals have as their last alternative a non-recursive production; if this is not the case an infinite sequence of insertions would take place. In order to avoid this we make use of the `<?>` combinator which points out the alternative to be taken when error recovery is taking place. The <?> combinator is used as the last alternative in a sequence of alternatives. The first component of its right hand side parameter represents the semantic value corresponding to this alternative, and its second component is a text which will be used in generating the message that this non-terminal has been inserted.

Exercise 7. Can you change the types in such a way that it is guaranteed by the type checker that there is always such an alternative constructed using <?>?

4.3 Using Parser Tables

As was indicated in exercise 4, the process of selecting a parser from a set of alternatives is rather cumbersome, since the current input symbol is repeatedly being sought for in an ever decreasing set. In this subsection some improvements are presented in order to cure this problem.

In the first place we note that the operator <|> is associative, so there is no reason to have the `elem` tests being performed in the same order as the alternatives have been grouped and ordered in the program text. Only one alternative should apply. Since the set of symbols in which the search starts is known statically we adapt the program in such a way that instead of searching through these sets, we construct a table in which each expected input symbol is associated with its corresponding parser. As a consequence the combinators do not construct parsers which make choices, but construct parser tables which will be used to make choices later when actually parsing. The `State` of the parsing process now no longer contains only the unconsumed part of the input, but also the error messages generated thus far. Since the `noskip`-sets are dynamic structures we take a simple approach to their implementation in which they are implemented as a stack of sets, the union of which is actually the set of non-skippable symbols. Since we will use this set only to see whether a specific symbol is present, and the symbol being sought will most likely be contained at or near the top of this stack, we prefer not to make this set into an actual set, but just to search for its presence in a linear way, from the top down to the bottom of the stack of sets.

```
> type State s  = ([s],String)
> type Noskip s = [[s]]

> type ErrParFun s a = State s -> Noskip s -> (a, State s)

> data Look s a      = Look s a
> type ParserTab s a = [Look s (ErrParFun s a)]
```

Here are the relevant definitions for dealing with parser tables. The table of the parser for a single symbol contains only one alternative, and since it is guaranteed that its contained parser will only be called when this symbol is indeed present, there is no need for a test for its presence anymore.

```
> tempty  _ = []
```

```
> tsymbol s = [Look s (\((_:inp), str) _ -> (s, (inp, str)))]

> talt = (++)

> mapParser tab f = [Look s (f p) | Look s p <- tab]
```

4.4 The Final Code

As usual we start with the type definition of the functions to be defined, in which the `ep` stands for *(deterministic), error (correcting) parser*:

```
> type ErrPar s a = (EmptyDescr s a, First s a, ParserTab s a) in
>  epempty, epsymbol, eperr, epalt, epseq,
>  invokeErr

> instance Parsing ErrPar where
>   empty  = epempty
>   symbol = epsymbol
>
>   (<?>) = eperr
>   (<*>) = epseq
>   (<|>) = epalt

> epempty  :: Symbol s => a -> ErrPar s a
> epsymbol :: Symbol s => s -> ErrPar s s
>
> eperr :: Symbol s => ErrPar s a      -> (a,String) -> ErrPar s a
> epseq :: Symbol s => ErrPar s (b->a) -> ErrPar s b -> ErrPar s a
> epalt :: Symbol s => ErrPar s a      -> ErrPar s a -> ErrPar s a
```

The last components of the results have now been converted into tables of parsers, each tagged with the symbol that they will accept first. For all parsers tagged in this way it holds that they cannot accept any other symbol first, and will indeed always accept the tag symbol when called.

The optimisation does not influence the definition of the empty parser, the symbol parser and the error parser. In the first part of the symbol parser, text is constructed which will report that this symbol was inserted in the input.

```
> epempty  v = (edempty  v, fempty  v, tempty  v)
> epsymbol s = (edsymbol s, fsymbol s, tsymbol s)
>
> ep `eperr` x  = ep `epalt` (ederr x, fempty x, tempty x)
> (edp, fp, tp) `epalt` (edq, fq, tq)
>  = (edp `edalt` edq, fp `falt` fq, tp `talt` tq)
```

For the choice combinator there is only a small change in the sense that the third part of the result is now a composition of lookup tables. Note that the actual choice is not being made here now, but later at the outermost choice level, i.e. at the point where all the possible alternatives to choose from have been collected.

```
> (edp, fp, tp) `epalt` (edq, fq, tq)
>  = (edp `edalt` edq, fp `falt` fq, tp `talt` tq)
```

Before presenting the function for sequential composition, one more function is introduced: the function which actually makes a choice or, in case no choice is possible, takes care of the error recovery. It takes as parameters the `EmptyDescr` of a parser and a binary search tree containing the non-empty parsers. Before choices have to be made the tables are converted into balanced binary search trees, so that the right parser can be quickly located. The code for this can be found in the appendix, since it is not an essential part of our line of reasoning.

First it is checked whether the end of the input has been reached. If so, and if the parser can accept the empty string, there is no problem: we return the semantic value corresponding to this parser; if not, the value corresponding to the inserted non-terminal is returned and its corresponding error message is left in the list of error messages.

If there is still an input symbol to be recognised and it can be found in the binary search tree, then there is no problem either and the corresponding parser is called. Otherwise, if the symbol is element of the noskip set, either the empty string should be recognised or a (number of) inserts should take place to reach the point where parsing can continue in a regular way. This is encoded in the `insorempty` function below. If the symbol is not in the noskip set it is discarded, and an error message is generated in the `state`.

```
> choose (edp,_) state@([],_) noskip = insorempty edp state
> choose x@(edp, ptree) state@((s:inp), errors) noskip
>  = find s ptree
>     (\_ _
>        -> if any (s `elem`) noskip then insorempty edp state
>           else choose x (inp, errors ++ deleted s) noskip
>     ) state noskip
> choose _ _ _ = error "no good alternative"

> insorempty edp state@(input,errors)
>  = case edp of
>     IsEmpty pv      -> (pv, state)
>     Insert pv error -> (pv, (input, errors ++ error))
```

Now we have finally reached the most complicated part of our derivation: the point where the sequential composition can be defined in its final form. The returned parser table consists of two parts:

- the first half consists of the parsers from the first operand, in which each parser has been extended with a `choose` from the tree `treeq`, constructed from the table of the second parser
- if the first parser can accept empty the second half consists of the second table, in which each parser has been changed in such a way that its result is subjected to the semantic value corresponding to the first empty parser

```
> (edp, fp, tp) `epseq` ~(edq, fq, tq)
>   = (edp `edseq` edq, fp `fseq` fq, tp `tseq` tq)
>    where
>     treeq = tab2tree tq
>     fp `fseq` fq
>      = fp `union` case edp of IsEmpty _ -> fq ; Insert _ _ -> []
>     tp `tseq` tq
>      = mapParser tp
>          (\p ss noskip ->
>             let
>              (pv,rs)  = p ss (fq:noskip)
>              (qv,rrs) = choose (edq, treeq) rs noskip
>             in (pv qv, rrs)
>          )
>       ++ case edp of
>            IsEmpty pv
>              -> mapParser tq
>                 (\q ss noskip ->
>                     let (qv, rs) = q ss noskip
>                     in (pv qv, rs)
>                 )
>            Insert _ _ -> []
```

Notice how the construction of the binary search tree `treeq` is done outside of the actual parsers, and is thus only done once, instead of whenever the parser is called.

Exercise 8. When there is only one alternative for the second parser the construct is rather cumbersome, because a tree is constructed, and an attempt is made in `choose` to find the appropriate symbol. Change the program in such a way that this situation is handled more efficiently.

Finally we can now define the function `invokeErr`. When starting to parse, the parsing table of the root symbol must first be subjected to the `choose` function, converting it into a real parser:

```
> invokeErr :: Symbol s => ErrPar s a -> Input s -> (a, String)
> invokeErr (edp, _, tp) input =
>  let (a, (_, errors)) = choose (edp, tab2tree tp) (input, []) []
>  in (a, errors)

> display (a, errors) = show a ++ "\n" ++ errors
```

Exercise 9. Is the computation of the first set still needed?

Exercise 10. The `<?>` instances have always been defined in terms of the `<|>` instances. This implies that (for the purposes of the parsers considered in this paper) we can do without the `<?>` combinator and use a `fail` parser as a last alternative instead. Change the `Parsing` class and its instances to incorporate this fact.

4.5 Discussion of Error Recovery Properties

Before starting the final discussion of what has been achieved we will briefly discuss the error recovery technique which was chosen. The technique chosen is very general, but it may be a bit too agressive. A famous example can be found in the Pascal grammar, which at the outer level reads something like:

program := **program** ident ; declarations ; **begin** statements **end**.

When, using our recovery techniques, someone writes a superfluous dot (.) in a record selection (e.g. `a..b`) then the parser will conclude that the second dot is actually the dot terminating the program, and it will quickly insert all pending non-terminals and think it is done. This can be prevented by a slight rewrite of the grammar to contain an extra non-terminal:

program_end ::= **end**.

Now the final dot no longer belongs to the first set of a pending parser, and the superfluous dot will be deleted.

Exercise 11. Change the combinators in such a way that if too many symbols are being inserted, an attempt is made to see whether a symbol should be deleted. Many strategies are possible here.

Exercise 12. There is a situation where the naive application of the techniques presented may lead to unexpected or undesired results. It is not uncommon to forget to write a separator in a `chainr` construct. E.g. semicolons tend to be forgotten. Due to the way our `chainr` combinator has been defined however the first symbols of the element after the separator are not visible until the separator has been recognised. This can be cured by spending some more effort on the definition of `chainr` by explicitly adding the first symbols of the elements to the table of the separator, making them in this way visible in the noskip sets. Change the definition of `chainr` such that it becomes a member of the `Parsing` class with a default definition and change this default definition for the `ErrPar` instance.

5 Discussion

5.1 Partial Evaluation

We have now reached the end of the development. We want to point out some important aspects of our approach:

- an attempt was made to move expressions which do not in some way depend on the input out of the actual parsers. As a consequence they will only be evaluated once. When running the 'generated/described' parsers more and more parts of the grammar will become analysed and the parser will run faster. What we have constructed, is a parser generator which is generating parsers on demand.
- the technique presented can be applied in many other situations in which a static and a dynamic computation are intricately intertwined. We reap the benefits of partial evaluation [4], without having to resort to complicated off-line techniques. In the figure a profile of the heap is given when using the described parser. We see that after an initial startup phase there are no new nodes being produced by the combinators.

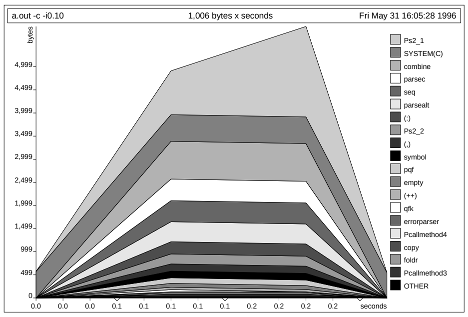

Fig. 1. Incremental behaviour of described parsers

One may be suprised by the small number of nodes actually being used; usually combinator parsers use a huge amount of memory by hanging on to their input. This problem has been solved by us, since we do not perform backtracking. There is however a second cause for this memory hungriness. When parsing, the symbols which have been recognised are consumed somehow by the application of the functions which have been inserted into the parsers using the `<$>` combinator. The expressions which are implicitely constructed in this way however, are not evaluated until there actually is a need for the top-value, i.e. the values corresponding to the synthesized attributes of the root, when expressing ourselves in attribute grammar based terminology.

This can be cured by incorporating some strictness into our parsers. In the Haskell library there exists a function `seq`, which forces the evaluation of its first element as far as possible before actually returning its second parameter. When applying the value `pv` to the value `qv`, we can better write this as:

```
let pvqv = pv qv in seq pvqv (state'', (pvqv))
```

One might be tempted to write this function as:

```
seq a b = if a == a then b
```

but this necessitates that the parameter `a` is an instance of the class `Eq`, which in our case will in general not be the case, since it is most likely a partially parametrised function.

5.2 Problems with Monad-based Formulations

Unfortunately the techniques which we have used do not easily extend to the monad-based combinator parsers[9]. Due to the monadic formulation the dynamic and the static values are tupled together, and thus cause the evaluation of the parser construction over and over again when parsing. To see what happens we inspect the following program:

```
f x = (a1 `expensive` b1, a2 `cheap` b2)
      where (a1, a2 ) = g x
            (b1, b2)  = h x
g x = (5, x+x)
h x = (3, x*x)
z   = f 7
```

On close inspection we see that the first component of the result of a call to `f` does not depend on the parameter `x` at all, and thus is static. Nevertheless it will be recomputed for each call of `f` again: an unfortunate situation. Of course one might argue that this program should not have been written in this way in the first place, but this is exactly what happens when one tries to write the functions we have presented in a monadic style. Of course the program above could easily be rewritten into:

```
f1   = (a1 `expensive` b1) where (a1, b1)  = (g1  , h1  )
f2 x = (a2 `cheap`     b2) where (a2, b2 ) = (g2 x, h2 x)
g1   = 5
h1   = 3
g2 x = x+x
h2 x = x*x
(z1, z2) = (f1, f27)
```

but that is exactly what we have done in our parsing combinators. Using techniques from the area of attribute grammars [7, 6] such rewriting can be done automatically in many situations. The techniques used imply however some form of global analysis of the program, and are not easily added to existing implementations of functional languages.

Exercise 13. This exercise is a little (but non-trivial) programming project. You are asked to write a code generator for a block structured language using the parsers developed in the paper. For the purposes of the exercise a block is a list of statements. A statement is a dummy statement, a variable declaration, a variable usage or a nested block. The concrete representation of a block of the block structured language looks as follows:

```
X;x;(Z;Y;;x;z;X);y;();Y
```

Statements are separated by semicolons. Nested blocks are surrounded by parentheses. The usage `Z` refers to the local declaration `z`. The usage `Y` refers to the global declaration `y`. The local usage `X` refers to the local declaration `x` and the global usage `X` refers to the global declaration `x`. Note that it is allowed to use variables before they are declared. The generated code should look at follows

```
? compile "X;x;(Z;Y;;x;z;X);y;();Y"
Enter 2
Access (1,1)
Enter 2
Access (2,2)
Access (1,2)
Dummy
Access (2,1)
Leave 2
Enter 0
Dummy
Leave 0
Access (1,2)
Leave 2
```

– Write a combinator parser which returns the parsed input sequence.
– Write a combinator parser which generates code for the parsed input sequence. The recover action for an ill formed statement should be such that it enables you to continue generating code. Do not write a parser which directly generates code, write a parser which returns a function which generates code using appropriate inherited and synthesised attributes.

6 Acknowledgements

We want to thank Pablo Azero, Jeroen Fokker, and Erik Meijer for their extensive comments on the paper. The first author wants to thank the computer science department of Chalmers university and its personnel for their hospitality when working on a preliminary version of this paper.

References

1. Jeroen Fokker. Functional parsers. In Johan Jeuring and Erik Meijer, editors, *Advanced functional programming, Baastad Summerschool Tutorial Text*, volume 925 of *LNCS*, pages 1–23. Springer, 1995.
2. Graham Hutton. Higher-order functions for parsing. *Journal of functional programming*, 2:323–343, 1992.

3. Mark P. Jones A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism *Conference on Functional Programming Languages and Computer Architecture, Copenhagen*, Denmark, pages 52-64, ACM, 1993.
4. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program generation.* Prentice-Hall, 1993.
5. Andrew Partridge and David Wright. Parser combinators need four values to report errors. Technical report, Department of Computer Science, Tasmania, 1994.
6. M. Pennings, S.D. Swierstra, and H.H. Vogt. Using cached functions and constructors for incremental attribute evaluation. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 130–144. Springer, 1992.
7. Maarten Pennings. *Generating Incremental Attribute Evaluators.* PhD thesis, Utrecht University, Dept. of Computer Science, 1994. www.cs.ruu.nl.
8. Niklas Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages.* PhD thesis, Chalmers, rojemo@cs.chalmers.se, 1995.
9. Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced functional programming, Baastad Summerschool Tutorial Text*, volume 925 of *LNCS*, pages 24–52. Springer, 1995.

7 Appendix: Utility Functions

For building the efficient mappings from first symbols to parsers we have used binary search trees. For the sake of completeness we give here the code for constructing such trees.

```
> data BinSearchTree a
>  = Node (BinSearchTree a) a (BinSearchTree a)
>  | Leaf a
>  | Nil
>
> type ParserTree s a = BinSearchTree (Look s (ErrParFun s a))

> tab2tree :: Symbol s => ParserTab s a -> ParserTree s a
> tab2tree tab = tree
>  where
>   (tree,[]) = sl2bst (length tab) (qsort tab)
>   qsort [] = []
>   qsort (look@(Look s _):tab)
>    = qsort  [look | look@(Look t _) <- tab, t <= s ]
>          ++ [look] ++
>      qsort [look | look@(Look t _) <- tab, t > s]
>   sl2bst 0 list     = (Nil   , list)
>   sl2bst 1 (v:rest) = (Leaf v, rest)
>   sl2bst n list
>    = let
>       ll = (n - 1) `div` 2 ; rl = n - 1 - ll
>       (lt,a:list1) = sl2bst ll list
```

```
>       (rt,  list2) = sl2bst rl list1
>      in (Node lt a rt, list2)
```

The function **find** tries to find the parser indexed by a specific symbol; if the symbol cannot be found in the tree the continuation **notfound** is returned.

```
> find i Nil notfound = notfound
> find i (Leaf (Look s p)) notfound
>       = if i == s then p else notfound
> find i (Node left (Look s p) right) notfound
>       | i==s = p
>       | i<s  = find i left  notfound
>       | i>s  = find i right notfound
```

We end with the function **union** for computing the union of two sets.

```
> union :: Eq x => [x] -> [x] -> [x]
> xs `union` ys = nub(xs++ys)
```

Exercise 14. Can you write the function **find** without the argument **notfound**?

This article was processed using the LaTeX macro package with LLNCS style