Layered Combinator Parsers with a Unique State

Draft

Pieter Koopman & Rinus Plasmeijer

Nijmegen Institute for Information and Computing Sciences, The Netherlands
{pieter, rinus}@cs.kun.nl

Although it is possible to construct recursive descent parser very elegantly using parser combinators, this is not often done in practise. This is a pity since the combinator approach to construct parsers has a number of clear advantages: like the availability of full power of the high level functional languages to manipulate parsers and their results, arbitrary look ahead, and context sensitivity. Using continuations enables us to implement parsers efficiently. Remaining problems are the error handling, the use of efficient data structures in the parse state, and for some applications the lack of a separate scanner level. In this paper we show how all these problems are solved using a set of parser combinators that is based on a multi parameter type class. All existing advantages of combinator parser are preserved.

1. Introduction

In the literature a rather large number of approaches is described to construct parsers in functional programming languages. We can distinguish two approaches in the construction of parsers.

The first approach uses a program generator [??, ??]. The generator is feed with some description of the syntax of the language to be parsed and generates an appropriate parser. The advantage of this approach is that it is relative easy for the generator to check properties of the syntax, like left-recursion and LL(1), and to manipulate the syntax.

The second method to construct parsers uses a library of functions [??, ??]. The functions in such a library are usually called *parser combinators*. The drawback of this methodology is that it becomes the responsibility of the programmer to enforce constraints, like not left-recursive, on the syntax. However, there are also some big advantages associated with using a library of parser combinators. For instance, there is no separate language needed to describe the syntax. The main benefit however, is that the full power of the high level functional language can be used to construct parsers and handle the parsed objects. For instance parsers can be produced by other functions, be stored in data-structures, and combined like any other functions in the language. As a result it becomes very easy to write context sensitive parsers with a look ahead as big as is required. The constructed parsers usually combine the work classically done by a scanner, recognising tokens, and a conventional parser, combining

tokens to sentences. As shown earlier [??] these parsers can be made also efficient by an appropriate implementation and restricting the non-determinism to the spots where it is really needed. Swierstra describes some sets of parser combinators that are able to detect and correct errors in the language accepted by the constructed parser [??].

Despite of the advantages mentioned, combinatory parsers are not heavily used for real world programs. We think there are a couple of reasons for this. One of the main problems is that error handling remains cumbersome. For languages with a rich input syntax and few keywords, like functional programming languages, suitable error correction is next to impossible. Further more we need possibilities to maintain parse tables and other data structures efficiently. In Clean[1] this implies that the parser should have a *unique state*. In such a unique state we can store heaps, updateable arrays and the like to construct efficient syntax trees and files to contain generated messages. Finally, on occasions it is convenient to introduce a separate scanner, for instance to implement an offside rule. The scanner and parser perform very similar tasks on a different sequence of input symbols, characters for the scanner and tokens for the parser. It is very desirable that the same set of combinators can be used for the scanner and the parser.

In this paper we present a set of parser combinators that is by default deterministic, has a unique state and can have multiple levels. It combines the advantages of existing combinators approaches and removes some hindrances for large-scale applications. The possibilities for error reporting and recovery are significantly improved, but error recovery is not yet automatically derived from the syntax description.

In section 2 we recapitulate the conventional parser combinators. In section 3 we present a quick overview of uniqueness in Clean. Section 4 explains the architecture of our parser combinatory system. The next section shows how a set of basic operations can be implemented which can read input from a unique file. In section 7 we show how parser combinators are defined and used. In section 8 we add a new level to the constructed parser. The combinators defined on the previous level can be applied in this level as well. Error handling is discussed in section 9. Finally, we show how non-determinism can be introduced if it is desired, and we draw some conclusions.

2. Conventional Parser Combinators

In the conventional approach parser combinators are non-deterministic. A parser takes a list of symbols as input and produces a list of results. Each element in the list represents a successful way to parse the current input. Such a success consists of the remaining input and the result produced by the parser.

```
:: Parser      s r :== [s] -> ParseResult s r
:: ParseResult s r :== [([s],r)]
```

The simplest parser combinator is `yield`. It takes a result as argument and produces a parse result independent of the form of the input.

[1] In this paper we will use Clean 2.0 since we need multi parameter type classes, although it is not yet public available.

```
yield :: r -> Parser s r
yield r = \ss = [(ss,r)]
```

The basic combinatory to recognize a given symbol in the input is `symbol`.

```
symbol :: s -> Parser s r | == s
symbol sym = p
where p [s:ss] | s==sym = [(ss,sym)]
      p _               = []
```

The non-deterministic or-combinator combines the results of two parsers. Note that the input is distributed over both parsers. Therefore the input is shared and cannot have a unique type, see section 3, in this approach.

```
(<||>) infixr 4 :: (Parser s r) (Parser s r) -> Parser s r
(<||>) p1 p2 = \ss = p1 ss ++ p2 ss
```

In many situations one is only interested in a single successful parse of the input. In this situation we only apply `p2` to the input if `P1` fails. The corresponding or-combinator is:

```
(<|>) infixr 4 :: (Parser s r) (Parser s r) -> Parser s r
(<|>) p1 p2 = \ss = case p1 ss of
                      []  = p2 ss
                      res = res
```

Using this version of the or-combinator can reduce the complexity of the parser from polynomial to linear [??].

There are many variations of the and-combinator. This combinator receives two parsers as argument. The first parser is applied to the input and the second parser is applied to the rest of the input. Here we supply the result of the first parser as argument to the second parser. This is sometimes called the *monadic style*.

```
(<&>) infixr 4 :: (Parser s r) (r->Parser s t) -> Parser s t
(<&>) p1 p2 = \ss1 = [  tuple
                     \\ (ss2,r1) <- p1 ss1
                     ,  tuple    <- p2 r1 ss2
                     ]
```

As a very simple example we implement a parser that accepts the symbol `'a'` followed by a `'b'` or a `'c'`. Since it is obvious that we can only recognize a `'c'` if there was no `'b'`, we use `<|>` as or-combinator.

```
a_and_b_or_c :: Parser Char (Char,Char)
a_and_b_or_c =  symbol 'a'                 <&> \r1 =
               (symbol 'b' <|> symbol 'c') <&> \r2 =
                yield (r1,r2)
```

We have used the conventional implementation of parser combinators here, although it is much slower than an implementation using continuations. This implementation is very simple and makes it easy to grasp the idea.

3. Uniqueness

The lazy functional programming language Clean has since many years a *uniqueness-type-system* [??]. An object is said to be unique if it can be assured at compile-time that there is always at most one single reference to that object. This is used to enforce

the single threaded use of real world objects that cannot be copied, like files and windows. If large data-structures, like arrays, are known to be unique the can be updated destructively without compromising referential transparency. In Clean it is only possible to update an array if it is known to be unique.
We notate uniqueness by prepending the type with an asterisk. For instance `File` is the ordinary type for files. Since reading does not change a file we can read from objects with type `File`. In order to define the sequence of write operations on a file we have to guarantee that the file is used in a single threaded way. In Clean this is done by requiring an object of type `*File`. This object is unique, hence there are no other references to it, and so it is impossible to apply multiple write operation to the same object. The write operation yields a new unique file. This new file can be used for the next write operation. In Haskell, the required single threaded use of the file is achieved by using a special monad.

The type of the function to write a single character to a file is:

```
fwritec :: !Char !*File -> *File
```

Using this function we implement a function that writes a list of characters to a file.

```
writeList :: [Char] *File -> *File
writeList []    file = file
writeList [c:r] file = writeList r (fwritec c file)
```

Some functions are polymorphic in uniqueness. When they receive a unique argument they produce the unique result. However, they can also have a non-unique argument and, the result will than also be non-unique. We notate this by prefixing the type by a dot. The best-known example is the identity function.

```
id :: .t -> .t
id x = x
```

Unique elements can also be part of data-structures. It is obvious that elements of a data-structure can only be unique if the encapsulating structure is unique. If the surrounding structure is not unique, it is always possible to create additional reference via this surrounding structure. Such an additional reference spoils uniqueness.

The Clean system is able to derive and verify uniqueness properties of functions.

4. System Architecture

As explained in the introduction we require that the parser combinators are able to do error handling, which can be used in a multi-level approach. We also require that the parsers are able to handle unique objects, like symbol tables.

In order to generate proper error messages the parser should have a notion of location, i.e. what is the input and what is the current position inside this input. Furthermore, we want to have a file available to write messages to. This is achieved by replacing the list of input symbols by a more elaborated input state. This state contains at least the actual input, a notion of position inside this input and the messages file. Since the input is usually stored in a file it might be worthwhile to see if it is possible to avoid the construction of a list of input characters. This parse state contains a unique object, the messages file, and hence has to be unique. The fact that we want to

be able to handle unique objects in the parser is another reason to make the parse state unique.

Since we want to be able to have a multi-level approach we model the parse state by a type constructor class rather than a direct class. Although parsers of more than two levels are rare, there is no reason to restrict us to two levels. Using a type constructor class also enables us to use one and the same parser on different types of input. The input can modelled for instance by a list of characters, a string (an unboxed array of characters in Clean), or a file.

This results in the following approach:

- A type constructor class represents the parser state.
- We define instances of this class to handle different kinds of input. In this paper we will start with a parser state where the input come form a unique file.
- The parser combinators are defined in terms of the manipulation functions of the type constructor class.
- Parsers are as usual defined in terms of the parser combinators.
- A second level of the parser is defined by a new instance of the type constructor class that contains the lower level parser, the scanner.

We will threat these topics in separate sections.

5. Parser State

The parser state is a type constructor class that is parameterised by the type of symbols it contains. In order to have an extendable parser state we will provide a second type argument that represents a user-defined extension of the parse state. In this paper we will use this extension to hold the offside-administration and the hashtable for symbols. This field will be used to hold unique symbol tables and the like. Hence, this field and the whole state are unique. Due to this requirement we will need multi-parameter type classes and hence version 2.0 of Clean.

The basic function for the parse state is to read a new symbol.

```
class PSread ps sym st :: (*ps sym *st)-> (sym,*ps sym *st)
```

It appears to be convenient to have the current input symbol always easily available. This is achieved by the uniform use of the tuple `(symbol,*ps symbol *state)` instead of a state `*ps symbol *state` that contains the current symbol.

The next three functions are intended to make a branch in the parsing process, to return to that point as required by the or-operator and to remove the bracng. Due to the uniqueness of the state we cannot simply copy the current state to both branche.

```
class PSsplit ps sym st :: (sym,*ps sym *st) -> (sym,*ps sym *st)
class PSback  ps sym st :: (sym,*ps sym *st) -> (sym,*ps sym *st)
class PSclear ps sym st :: (sym,*ps sym *st) -> (sym,*ps sym *st)
```

Applying the function `PSsplit` indicates that it is possible to return to the current spot in the input. Applying the function `PSback` indicates that the parser wants to return to the earlier indicated spot in the input. It is obvious that splitting and going back in the input can be nested to arbitrary depth.

The current position in the input can be obtained by applying:

```
class PSpos ps sym st :: (sym,*ps sym *st)->(Pos,(sym,*ps sym *st))

:: Pos = {pos_inp::String, pos_col::Int, pos_line::Int}
```

The error message `m` can be written to the output file inside the parser state by:

```
class PSerror ps sym st::m (sym,*ps sym *st)->(sym,*ps sym *st)|<<= m
```

There are two higher order functions to apply functions to the user-defined state within the parser state. The first is used to apply functions that produce a result and a new local state. The second just applies a function to the local state.

```
class PSacc ps sym st
 :: (*st->(r,*st)) (sym,*ps sym *st)->(r,(sym,*ps sym *st))

class PSapp ps sym st::(*st->*st)(sym,*ps sym *st)->(sym,*ps sym *st)
```

In order to produce better error messages we supply a stack of strings that is used to indicate the current parsing context. This context is used during the production of error messages.

```
class PSpush ps sym st
 :: String (sym,*ps sym *st) -> (sym,*ps sym *st)

class PSpop  ps sym st :: (sym,*ps sym *st) -> (sym,*ps sym *st)
```

The actual library contains some additional operations, but for the purpose of this paper we define the parser state as:

```
class ParserState ps symbol state
    | PSread, PSsplit, PSback, PSerror, PSacc, PSapp,
      PSpush, PSpop ps symbol state
```

6. File Input

The first instance of `ParserState` that we define contains a unique file used to read characters from. For efficiency reasons we read a whole line of charters at a time instead of reading characters one by one. This line of charters from the file is stored as a string in the parser state. We maintain in index in this string to indicate the current character. Due to the existence of tabs the index is not necessary equal the column in the position.

For this parser state we assume that we never need to go back to a previous line. If the parser needs to go back in the input it will always remain on the current line. Since this parser state corresponds to a scanner this is a very realistic approach. Note however, that this is not a restriction of the system, but a restriction of this implementation of the parser state. Using seeks it is possible to go arbitrary far back in the input file without having to store a huge part of the file.

```
:: *FileInput symbol state =
        { fi_file  :: *File
        , fi_pos   :: Pos
        , fi_line  :: String
        , fi_index :: Int
        , fi_hist  :: [(Index,Column)]
        , fi_error :: ErrorState
        , fi_cont  :: [String]
```

```
        , fi_state :: state
        }
```

The manipulations functions of the class `ParserState` are implemented for `FileInput`. The implementations are all rather straight forward. We show some illustrative examples.

```
instance PSpos FileInput Char state
where PSpos (c,fi=:{fi_pos}) = (fi_pos, (c,fi))

instance PSsplit FileInput char state
where PSsplit (c,fi=:{fi_index,fi_hist})
       = (c,{fi&fi_hist=[(fi_index-1,fi.fi_pos.pos_col-1):fi_hist]})

instance PSback FileInput Char state | PSread FileInput Char state
where PSback (c,fi=:{fi_hist})
       = case fi_hist of
          [(index,col):t]
           = PSread {fi & fi_index = index
                        , fi_hist = t
                        , fi_pos={fi.fi_pos&pos_col=col}
                    }
```

7. Parser Combinators

The parser combinators are built on the parser state type class. In order to prevent the construction of unnecessary intermediate data structures we equip the parsers with two continuations.

The first continuation determines what should be done if the current parser succeeds. It receives the result of the current parser, the fail continuation and the tuple containing the current input symbol and the parser state as arguments.

The second continuation determines what should be done when the current parser fails. This continuation has only the current input tuple as argument.

The final argument of each parser is the current input tuple.

```
:: *Tuple   sym ps st   :== *(sym,ps sym st)
:: FailCont sym ps st r :== *(Tuple sym ps st)->*(r,*Tuple sym ps st)
:: SuccCont sym ps st t r
 :==t (FailCont sym ps st r)*(Tuple sym ps st)->*(r,*Tuple sym ps st)

:: Parser sym ps st tmp r
   :==  (SuccCont sym ps st tmp r)
        (FailCont sym ps st r)
       *(Tuple sym ps st)  ->  *(r,*Tuple sym ps st)
```

We show some simple parser combinators corresponding to combinators shown in section 2. The simplest combinator is `yield`. It succeeds with the given value. A parser succeeds by applying the success continuation to its result, the fail continuation and the input.

```
yield :: tmp -> Parser sym *ps .st tmp result
yield r = \succ fail tuple = succ r fail tuple
```

The counter part of the parser that always succeeds is the parser that always fails. It is much less useful, but illustrative.

```
failComb :: Parser sym *ps st tmp result
failComb = \succ fail tuple = fail tuple
```

A more serious parser primitive checks whether the first symbol in the input is equal to the given symbol.

```
symbol::sym->Parser sym *ps st sym r | == sym & ParserState ps sym st
symbol wanted = p
where p succ fail t=:(sym,ps)
          | sym == wanted
          = succ wanted fail (PSread ps)
          = fail t
```

If you look at this function you recognise the `yield` function in the first alternative of `symbol` and the fail combinator in its second alternative.

There are two basic combinators to combine parsers. The first one denotes a sequence of parsers that should be applied. The second one denotes a choice. Both parser combinators are implemented by inserting the other parser at the correct place in the continuations of the first one.

For the sequence combinator we insert the second parser in the success continuation of the first parser. The second parser should be invoked to the remaining input if the first parser succeeds. The standard monadic parser combinator is:

```
(<\>) infixl 6 :: (Parser sym *ps .st a r)
                  (a->Parser sym *ps .st b r)->Parser sym *ps .st b r
(<\>) p1 p2 = \succ fail t = p1 (\a = p2 a succ) fail t
```

It appears that it is often more convenient to have a sequence combinator where the first parser yields a function that takes the result of the next parser as argument. This reduces the number of lambda functions needed significantly.

```
(<+>) infixl 6 :: (Parser sym *ps .st (a->b) r)
                  (Parser sym *ps .st a r) -> Parser sym *ps .st b r
(<+>) p1 p2 = \succ fail t -> p1 (\f = p2 (\x = succ (f x))) fail t
```

To parse syntax elements, like keywords and parentheses, that does not contribute to the syntax tree conveniently, we also introduce the operators <+ en +> that yield only the result of the first and second parser respectively.

We choose to have deterministic parsers. For a choice combinator this implies that the second parser is only invoked if the first one fails. For the choice combinator we insert the second parser in the fail continuation of the first one.

```
(<|>) infixr 4 :: (Parser sym *ps .st t r)
                  (Parser sym *ps .st t r) -> Parser sym *ps .st t r
                  | ParserState ps sym st
(<|>) p1 p2 = \succ fail tuple -> p1 (\r f t=succ r fail (PSclear t))
              (\t2 -> p2 succ fail (PSback t2)) (PSsplit tuple)
```

Using these combinators we define equivalent of the simple parser from section 2 that recognises an 'a' followed by a 'b' or a 'c'.

```
a_and_b_or_c
 =     yield (\x y=(x,y))
   <+> symbol 'a'
   <+> (symbol 'b' <|> symbol 'c')
```

For convenience a rich set of operators is defined in the library. We define the following operator as a frequently used special case of this combinator. It constructs a list element combining the results of the parsers `p1` and `p2`.

```
(<:>) infixl 6
(<:>) p1 p2 :== yield (\h t=[h:t]) <+> p1 <+> p2
```

It is often needed to insert a function as starting point of a sequence of parsers. In the example above we did this by `yield (\x y=(x,y))`. It is worthwhile to introduce a special infix operator.

```
(@>) infixl 7 :: (a->b) (Parser s *ps .st a r)-> Parser s *ps .st b r
(@>) f p = yield f <+> p
```

A cousin of this operator applies a function to the parsed item.

```
(<@) infixl 5 :: (Parser s *ps .st a r) (a->b)-> Parser s *ps .st b r
(<@) p f = \succ fail tuple = p (\r -> succ (f r)) fail tuple
```

Another relative applies a function to the parse state if the parser succeeds.

```
(<<@@) infixl 5
(<<@@) p f = \succ fail t = p (\x fail t = succ x fail (f t)) fail t
```

The Kleene star operator applies a given parser as often as possible to the input. The results of applying the parser are collected in a list.

```
star :: (Parser sym *ps .st a r) -> Parser sym *ps .st [a] r
        | ParserState ps sym st
star p = p <:> star p <|> yield []
```

The final operator shown here is token, it recognises a sequence of symbols.

```
token :: [s] -> Parser sym *ps .st [s] r | == s & ParserState ps s st
token []       = yield []
token [s:rest] = symbol s <:> token rest
```

Using the given parser combinators we can define a scanner for a simple functional programming language elegantly. The tokens are represented by an algebraic data-type. We use a data-type called `OffsideState` for the administration of offside positions. The function `setOffside` adds the current column as offside position. An offside token is generated by `generateOffsideToken` if the current position is in the input is offside. This function also takes care about removing offside positions.

```
scanner :: Parser Char FileInput OffsideState Token Token
scanner
 =      skipSpace +>
    (   generateOffsideToken
    <|> (token ['of']  <@ K OfToken)  <+ skipSpace <<@@ setOffside
    <|> (token ['let'] <@ K LetToken) <+ skipSpace <<@@ setOffside
    <|> satisfy isAlpha
        <:> star (satisfy isAlphanum) <@ check_reserved o toString
    <|> plus (satisfy isDigit)        <@ IntToken o to_number 0
    <|> symbol '='                    <@ K EqualToken
    <|> symbol '('                    <@ K OpenToken
    <|> symbol ')'                    <@ K CloseToken
    )
```

8. Token Input

The next step is to construct a parser that is able to parse the tokens constructed by the scanner. The token input is very similar to the file input. Since it is the second layer of

the parser, the first layer is a component of this parser state. The state of the scanner and the scanner itself are fields in the record representing the token input. Here we do maintain a list of tokens to go back in the input of the high level parser.

```
:: *TokenInput symbol state
 =  { ti_fi      :: *(Char,*FileInput Char OffsideState)
    , ti_scanner :: *(Char,*FileInput Char OffsideState) ->
                    *(symbol,*(Char,*FileInput Char OffsideState))
    , ti_buffer  :: [symbol]
    , ti_history :: [[symbol]]
    , ti_state   :: state
    }
```

Note that we can stack parsers to arbitrary height if we do not give a concrete type for the lower level state and parser function. We use an existentional type for the variables in the state and lower level parse function.

```
:: *TokenInput token state
 = E. ps s st:
    { ti_fi      :: *(s,*ps s st)
    , ti_scanner :: *(s,*ps s st) -> *(token,*(s,*ps s st))
    , ti_buffer  :: [token]
    , ti_history :: [[token]]
    , ti_state   :: state
    }
```

Implementing the access functions of the class `ParserState` is a simple exercise.

9. Error handling

In practise a parser is most of the time applied to incorrect programs. This makes error handling necessary. Although these parser combinators can be used to construct any parser, they are constructed initially for a functional programming language. Such a language is characterised by a very rich syntax and small number of keywords. This makes a general form of error correction unfeasible. Hence, we will limit ourselves to proper error messages and some rudimentary error correction.

The infix operator ::> is used to assign a name to a parser. The name is pushed on the context stack before the parser starts and removed when it is finished.

```
(::>) infix 1 :: String (Parser s *ps .st t r)->Parser s *ps .st t r
                 | ParserState ps s st
```

Due to the stack of parsing environments and the current position maintained by the parser state it is possible to generate error messages like:

```
Error [t.icl,20,[Case_alt,Expression]]: ) expected instead of <<EOD>>
```

This indicates that the parser discovered an error in the file t.icl on line 20 while parsing an expression in side a case alternative. This error is produced by the function

```
parseError :: e a -> Parser s *ps st a t
              | ParserState ps s st & toString e & toString s
```

The first argument is the expected value, the second argument is the value used as result placeholder. Using this function we can define parser combinators that accepts the given token or languages construction if it is present, if the current symbol is not

the desired symbol an error message is generates and the parser continues as if the symbol was recognised:

```
wantSymbol sym = symbol sym <|> parseError sym sym
want p message default = p <|> parseError message default
```

The `wantSymbol` combinator appears useful for symbols like equal signs in function definitions and closing parenthesis. The combinator `want` is used for instance to recognise right hand side of function definitions.

To synchronise the parser with the input we define

```
skipToSymbol sym
 =     symbol sym
   <|> parseError sym sym +> star (satisfy (\s=s<>sym)) +> symbol sym
```

This skips symbols until the given symbol is recognised. If some symbols have to be skipped an appropriate messages is produced. This is applied to find the end of definitions indicated by the automatically generated offside-tokens.

Using the token parser and this error handling, we can construct a parser that recognises a simple expression and yields an appropriate syntax tree.

```
pExpression :: Parser Token *ps HashTable Expression t
             | ParserState ps Token HashTable
pExpression
 =  "Expression" ::>
        match mBasicValue <@ BV
    <|> pIdentifier
    <|> symbol OpenToken +> pCompoundExpression <+ symbol CloseToken
    <|>     symbol CaseToken
        +>  (\e ps d = Case {expr=e, patterns=ps, default=d})
        @>  pCompoundExpression
        <+  wantSymbol OfToken
        <+> star pCaseAlt
        <+> opti (symbol EqualToken +> pCompoundExpression)
        <+  skipToSymbol EndOfGroupToken
```

10. Non-deterministic Parsers

If for some reason we really want to have a parser for a non-deterministic syntax we need to implement a new or-combinator. This operator appends all results of the first parser to all results of the second parser. A naive implementation just appends the lists of successes. Since the parser used here does not yield list of successes we use the combinator `opt` to transform an ordinary parser to a parser producing a list of successes.

```
(<||>) infix 4::(Parser sym *ps .st t r) (Parser sym *ps .st t r)
                 -> Parser sym *ps .st [t] r | ParserState ps sym st
(<||>) p1 p2 = (\x y=x++y) @> opt p1 <+> opt p2

opt p :== (\x=[x]) @> p <|> yield []
```

A more advanced implementation prevents the construction of intermediate lists by employing the existing continuations directly. Also the composition of more than two parsers can and should be improved for serious applications. The given implementation just illustrates the possibilities.

11. Discussion

In this paper we introduced a set of parser combinators that inherits the advantages of previous developed combinator libraries [??]. The combinators enable us to write recursive descent parsers fast and clear. Since the parser combinators are ordinary functions the parsers and their results can be manipulated very flexible in a high level functional programming language. If desirable the parsers can be context sensitive, have an arbitrary look ahead and the syntax can be non-deterministic.

The library of combinators introduced in this paper offers some additional features. A very important contribution to the efficiency of real world parser is the ability to handle unique objects like symbol tables in the parse state. Another nice property is that a separate scanner level can be introduced if that appears to be desirable, for instance to implement an offside rule elegantly. If a separate scanner is not needed we do not have to introduce such a level. Moreover, it is possible to create as many levels as convenient and the same parser combinators can be used on each level. Also the possibilities for error handling are significantly improved.

The execution time of parsers constructed with these combinators does not differ much from ad-hoc written parser/scanner combinations and is very acceptable. Depending on the exact input and post-processing we measured up to 3500 lines containing simple function definition per second on a 1 GHz Windows-ME machine, with an AMD processor. The version of the parser constructed with parser combinators is much better readable, easier to change and more concise than the ad-hoc parser. In order to obtain a similar memory use we have added a combinator that eliminates the backtrack administration as soon as the programmer indicates that the current branch of the parser should not fail. It appears to be sufficient to use this in some strategically chosen top-level parse alternatives. For instance after the double double-colon symbol, `::`, in a type.

12. References

to be added