

# Lazy Functional Parser Combinators in Java

Atze Dijkstra<sup>1</sup> and Doaitse S. Swierstra<sup>2</sup>

<sup>1</sup> Institute of Information and Computing Sciences, Utrecht University,  
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands,

`atze@cs.uu.nl`,

`http://www.cs.uu.nl/~atze`

<sup>2</sup> `doaitse@cs.uu.nl`,

`http://www.cs.uu.nl/~doaitse`

**Abstract.** A parser is a program that checks if a text is a sentence of the language as described by a grammar. Traditionally, the program text of a parser is generated from a grammar description, after which it is compiled and subsequently run. The language accepted by such a parser is, by the nature of this process, hardcoded in the program. Another approach, primarily taken in the context of functional languages, allows parsers to be constructed at runtime, thus dynamically creating parsers by combining elements from libraries of higher level parsing concepts; this explains the the name “parser combinators”. Efficient implementation of this concept relies heavily on the laziness that is available in modern functional languages [13, 14]. This paper shows how to use parser combinators in a functional language as well as Java, and shows how parser combinators can be implemented in Java. Implementing parser combinators is accomplished by combining two libraries. The first one, written in Haskell, defines error-correcting and analysing parser combinators [2]. The second one consists of a small Java library implementing lazy functional behavior. The combinator library is straightforwardly coded in Java, using lazy behavior where necessary. In this paper all three aspects, the two libraries and its combination, are explained.

## 1 Introduction

Creating a parser for a grammar normally is a two step process. As a starting point some tool specific notation for a grammar specification is used. From this grammar specification an executable specification in a programming language is generated using a parser generator, subsequently compiled into an executable format. For example, using Java [4], one could use JavaCC [1] to generate a Java program to be compiled subsequently.

This two step process makes it possible to create highly efficient parsers. The parser generation phase usually analyses a grammar and takes advantage of programming language properties. However, this efficiency does not come without a price.

Generally, all information about the original grammar has been lost when a separate program is generated. Though this information can be added as ‘debug’

information to the generated parser, it will not repair the fact that the structure of a grammar has been hardcoded into the generated program. In general, this prevents the programmatic (runtime) manipulation of the original parser (and grammar) as this would require (runtime) rebuilding of the generated information.

Losing the ability to perform runtime manipulation and creation of parsers often is not such a high price to pay, except in situations where the input language described by a grammar may influence the grammar itself. For example, Haskell [5, 13] allows the programmer to define operators with their priority as well as associativity. The following lines, specifying the priority and left associativity of some operators, are from the prelude of Hugs [9]:

```
infixl 7 *, /, 'quot', 'rem', 'div', 'mod', :%, %
infixl 6 +, -
```

These declarations influence the way expressions are parsed and the abstract syntax tree for expressions is constructed. A parser needs this runtime gathered information about the precedence of operators to do its job. This can be accomplished via precedence parsing (see e.g. [3]) in the form of additional information steering the parsing process. However, such a solution is tailored for this special problem, that is, parsing expressions. A more general solution would be to construct a parser at runtime using the declared fixity and priority information about the operators. This is generally not possible when a generator is used.

In contrast, runtime manipulation of parsers is one of the strong points of parser combinators [7, 6, 8, 10, 16, 15]. In the context of parser combinators, a parser is a first class value, to be used or combined as part of other parsers and passed around like any other (programming) language value.

Another advantage of parsers being first class citizens is that it allows the definition of building blocks and the construction of abstract grammatical constructs out of these building blocks. This is similar to regular expressions (see e.g. [3]) where regular expressions can be optional (using `?`), be grouped (using `( , )`) and repeated (using `*`, `+`). Parser combinators allow us to go one step further by letting the programmer define his own abstractions. These abstractions then can ease the construction of a parser for a grammar.

The flexibility of parser combinators however has its usual price: decreased performance. Straightforward implementations of parser combinators [7, 6] rely on backtracking, to determine which alternatives in a grammar production rule are the ones matching a given input. Because of the non-linear (w.r.t. input length and/or grammar size) runtime costs this is unacceptable except for demonstration purposes. Monad based parser combinators [8, 10] allow the restriction of backtracking but this requires a careful grammar design. Only when (runtime) analysis is done on combined parsers backtracking can be avoided [16, 15] and a useful way of error recovery can be offered.

Parser combinator implementations which provide error correction and reporting as well as grammar analysis thus offer a solution to the grammar and parser writer in which both flexibility and performance are at an acceptable level. Efficient implementations for parser combinators however are generally written

in a functional language like Haskell, mainly because of the easy embedding of parser combinators into the language. Also, the advanced type systems offered by functional languages assist in detection of errors in an already complex implementation. And, last but not least, parser combinator implementations need laziness to allow “infinite” grammars, and to avoid unnecessary computations while retaining their notational flexibility.

The aim of this paper is to make parser combinator implementations available for imperative programming environments, and more specific, for Java. This paper elaborates on several different, but related subjects. First, we want to show how parser combinators can be used (section 2) and implemented (section 3). Second, we want to show how parser combinators can be used in Java (section 4). Finally, we will discuss how this can be implemented in Java (section 5).

It is assumed that the reader is familiar with Haskell as well as Java. Some familiarity with parser combinators would be helpful. This is also the case for implementation techniques for functional languages, especially graph reduction. This paper does not go further than marginally explaining these subjects as it is the goal to tie together different aspects of different paradigms.

As a running example we will use a small grammar, the foobar of parsing, expressed in EBNF:

```
<expr> ::= <term> (('+' | '-') <term>)* .
<term> ::= <factor> (('*' | '/') <factor>)* .
<factor> ::= ('0'..'9')+ | '(' <expr> ')'
```

## 2 Parsing with parser combinators

### 2.1 Basics

Conceptually, a parser is something which takes textual input and returns a value which is calculated using the recognised structure of the textual input. Using generated parsers, this is often done explicitly by using a stack of (intermediate) results. Each recognised nonterminal results in a value, which is pushed onto the stack. The arithmetic value of the expression is used as an example throughout this paper. A parser combinator captures this notion more directly by defining a parser to be a function returning a result. The basic idea of such a parser’s functionality is written in Haskell as:

```
type Parser = String -> Int
expr :: Parser
expr = ...
```

Thus a parser is a function taking an input string and yielding an integer result.

This definition of what a parser is (i.e. its type) turns out to be insufficient as it is unknown how much of the input has been used by a parser. A parser only consumes part of the input, so, what is left over after a parser returns its result should also be returned:

```
type Parser = String -> (Int,String)
```

---

	BNF	Combinator	Result
symbol	's'	pSym 's'	's'
choice	x   y	x '< >' y	result of x or result of y
sequence	x y	x '<*>' y	result of x applied to result of y
empty		pSucceed v	v

---

**Fig. 1.** Basic parser combinators and BNF.

---

The result of a parse now has become a tuple containing the result and the remainder of the input.

A second problem is the handling of errors. We will look at error recovery later (section 3.2) and for the time being consider an error to be a situation where no result can be computed. This can be encoded by letting a parser return a list of results. The empty list indicates an error:

```
type Parser = String -> [(Int,String)]
```

This also allows a parser to return multiple results, a feature which is used by the simplest implementation to handle alternatives of a grammar production rule. This definition is the most commonly ([7,6]) used. Finally, we generalise the `Parser` type by parameterising it with the type of the input symbols and the type of the result:

```
type Parser sym res = Eq sym => [sym] -> [(res,[sym])]
```

In this setting the BNF constructs have their parser combinator counterparts defined as functions. A parser for terminal symbol 'a' is constructed using the basic function `pSym`:

```
pSucceed :: a -> Parser s a
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
```

Sequencing and choice are explicitly constructed using `<*>` and `<|>` respectively (see figure 1). The `<factor>` nonterminal of the example grammar now translates to

```
pFact = pNat
        <|> pSym '(' <*> pExpr <*> pSym ')'
```

This definition is not yet correct. Apart from a missing definition for `pNat` it is unclear what the result of `pFact` is. BNF does not enforce the grammar writer to specify anything about results since a BNF grammar definition only says something about the concrete syntax, not the underlying semantics.

Each parser returns a value as the result of parsing a piece of input. The symbol parser `pSym` just returns the parsed symbol itself, whereas the choice parser

combinator `<|>` conceptually returns the result of the chosen alternative<sup>1</sup>. The parser for the empty string `pSucceed` returns the passed parameter as its result because no input was parsed to derive a value from. The sequence combinator requires the result of its left argument to be a function, which is then applied to the result of the second parser. With these rules in mind the definition for `pFact` can be written as:

```
pFact = pNat
      <|> pSucceed (\_ e _ -> e)
      <*> pSym '(' <*> pExpr <*> pSym ')'
```

The result of `pNat` now is a natural number or the value of another expression, both an `Int` in the example. The result values of the parenthesis are not used.

## 2.2 Higher order parser combinators

One of the nicest features of using parser combinators embedded in a general purpose programming language is that it allows the programmer to define his own combinators and abstractions. For example, the second alternative of the definition for `pFact` in the preceding section parses an expression surrounded by parenthesis. We could abstract over this ‘surrounded by parenthesis’ by defining:

```
pParens x = pSucceed (\_ e _ -> e) <*> pSym '(' <*> x <*> pSym ')'
```

The combinator `pParens` itself is a special case of the combinator `pPacked` representing the abstraction ‘surrounded by ...’:

```
pPacked l r x = pSucceed (\_ e _ -> e) <*> l <*> x <*> r
pParens      = pPacked (pSym '(') (pSym ')')
```

The combinator `pPacked` itself may be built upon several ‘throw a result away’ abstractions:

```
pPacked l r x = l *> (x <*> r)
```

The `*>` and `<*>` combinators are variants of the sequence combinator `<*>` which throw away the result of the left respectively right parser given as argument to the respective combinators:

```
f <$> p = pSucceed f <*> p
p <*> q = (\ x _ -> x) <$> p <*> q
p *> q = (\ _ x -> x) <$> p <*> q
```

The application parser combinator `<$>` is a shorthand for the already used combination of `pSucceed f` followed by an arbitrary parser `p`, used for applying a function to the result of `p`.

Even more useful are combinators which encapsulate repetition, as the counterpart of `*` and `+` in the EBNF grammar at the end of section 1. For example, the parser `pNat` for an integer may be written as

---

<sup>1</sup> When using the implementation based on lists, as in this section, a list of results will be returned.

```

pDigit = (\d -> ord d - ord '0') <$> pAnySym ['0'..'9']
pNat   = foldl (\a b -> a*10 + b) 0 <$> pList1 pDigit

```

The combinator `pList1` takes a parser for a single element of a repetition and uses it to parse a sequence of such elements. The result of `pList1` is a non-empty list of result values of the single element parser. In this example this list is then converted to an `Int` value.

All the combinator variants parsing a sequence behave similarly to `pList1`. The variants differ in the handling of the single element results and the minimum number of repetitions:

```

p 'opt' v          = p <|> pSucceed v
pFoldr alg@(op,e) p = pfm where pfm = (op <$> p <*> pfm) 'opt' e
pList             p = pFoldr ((:), []) p
pList1           p = (:) <$> p <*> pList p

```

The basic building block of these sequencing combinators is the folding combinator `pFoldr` which works similar to the `foldr` from Haskell. A unit value `e` is used as the result for an empty list and a result combining operator `op` is used to combine two result values. The combinator `pList` then uses `pFoldr` to build a list from the sequence. This list may be empty, i.e. no elements may be parsed, whereas `pList1` parses a sequence non-zero length.

For `pFoldr` and its derivatives the way elements of a sequence are combined is fixed. That is, the programmer specifies `alg@(op,e)` and the parsed input does not influence this. However, this does not work for:

```

<term> ::= <factor> (('*' | '/') <factor>)* .

```

For `<term>` the combination of two factors within a sequence of factors is determined by the operator in between. In other words, the result of parsing an operator determines how two factors are to be combined. The parsing result of `('*' | '/')` thus somehow has to be used to combine two factors. This is done by the chain combinator `pChain1` used to define `pTerm` and `pExpr` as follows:

```

pTerm = pChain1 (  (*) <$ pSym '*'
                  <|> div <$ pSym '/'
                  )
        pFact

pExpr = pChain1 (  (+) <$ pSym '+'
                  <|> (-) <$ pSym '-'
                  )
        pTerm

```

The combinator `pChain1` (and its right associative variant `pChainr`) take two parsers, one for the elements of a sequence and one for the separator between them. For `pTerm` the elements of the sequence are `pFact`'s, separated by either a `*` or a `/`. The chain combinators expect the result of the separator parser to be a function accepting (at least) two arguments. This is precisely what `(*) <$ pSym '*'` and `div <$ pSym '/'` return.

---

```

module Extended0 where
import Basic0
infixl 4 <$>, <$, <*, *>, <*>, <??>
infixl 2 'opt'

pAnySym :: 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
(<* )   :: 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

pAnySym = foldr (<|>) pFail . map pSym
p 'opt' v = p <|> pSucceed v
f <$> p   = pSucceed f <*> p
f <$ p    = const f <$> p
p <* q    = (\ x _ -> x) <$> p <*> q
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
  = 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)

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

pList1 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
    f x (func:rest) = f (func x) rest

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

pOParen = pSym '('
pCParen = pSym ')'
pParens = pPacked pOParen pCParen

```

**Fig. 2.** Higher order parser combinator functions.

---

---

```

module Basic0 where
infixl 3 <|>
infixl 4 <*>

type Parser s a = ...

pSucceed :: a -> Parser s a
pFail    :: Parser s a
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

pSucceed v input = ...
pFail    input = ...
pSym a    input = ...
(p <|> q) input = ...
(p <*> q) input = ...

data Reports = Error      String Reports
             | NoReports
             deriving Eq

instance Show Reports where
  show (Error      msg errs) = msg ++ "\n" ++ (show errs)
  show (NoReports  ) = ""

parse :: Parser s a -> [s] -> (a,Reports)
parse p inp = ...

```

**Fig. 3.** Basic parser combinator functions.

---

The definition for the chain combinator as well as the other combinators can be found in figure 2.

As we will consider different implementations of parser combinators in section 3 we will interface with a parser via a function `parse` which hides the invocation details and returns a result as well as error messages, see figure 3. In figure 4 it is shown how the function `parse` is used in a small calculator based on the preceding definitions for `pExpr` (see figure 5 for the complete listing).

### 3 Implementing parser combinators

Implementations of parser combinators have to take care of several aspects:

- checking if input is accepted by a grammar (error detection).
- returning a value as directed by the input (syntax directed computation).

---

```

on :: Show a => Parser Char a -> [Char] -> IO ()
on p inp -- run parser p on input inp
  = do let (res, msgs) = parse p inp
        putStr (if msgs == NoReports then "" else "Errors:\n" ++ show msgs)
        putStr ("Result:\n" ++ show res ++ "\n")

main :: IO ()
main = do putStr "Enter expression: "
         inp <- getLine
         pExpr 'on' inp
         main

```

**Fig. 4.** Usage of 'parse' to interface with parser.

---



---

```

pDigit = (\d -> ord d - ord '0') <$> pAnySym ['0'..'9']

pNat = foldl (\a b -> a*10 + b) 0 <$> pList1 pDigit

pFact =  pNat
        <|> pParens pExpr

pTerm = pChain1 (  (*) <$ pSym '*'
                 <|> div <$ pSym '/'
                 )
        pFact

pExpr = pChain1 (  (+) <$ pSym '+'
                 <|> (-) <$ pSym '-'
                 )
        pTerm

```

**Fig. 5.** Expression parser.

---

- if input is not accepted make attempts to repair (error recovery).
- minimizing the amount of unnecessary parsing steps by performing grammar analysis.

The first and second of these aspects are easily implemented [7, 6] by following the suggested representation of section 2. Error recovery attempts to repair a parse by adding symbols to the inputstream or deleting symbols from the inputstream. Finally, by analysing a grammar lookahead information can be extracted. Of these subjects all except the grammar analysis are covered in the following paragraphs.

### 3.1 Backtracking

When backtracking, a parser just returns all possible parses and makes no attempt to predict if one will fail or not. In particular, a combinator just makes an attempt to parse by trying out its components. The combinator `pSym` is the only combinator really making a decision about the correctness of the input. The parser actually works like a recursive descent parser and will -if necessary- check all the returned solutions.

```

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

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

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

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

parse :: Parser s a -> [s] -> (a,Reports)
parse p inp
  = let results = p inp
      filteredResults = filter (null . snd) results
      in case filteredResults of
        [          ]-> (undefined,Error "no correct parses" NoReports)
        [(res,_)  ]-> (res,NoReports)
        ((res,_) :rs)-> (res,Error "ambiguous parses" NoReports)

```

This ‘reference’ implementation (based on [7, 6]) of the building blocks for parser combinators is not efficient. For smaller examples without many alternative productions for a nonterminal this approach still works. A grammar having many alternatives will lead to a recursive descent parsing process where all alternatives are descended, without making an attempt beforehand to determine which alternatives surely will yield no valid parse.

Another deficiency of this implementation is that an error in the input is not handled at all; the parser simply concludes that no parse tree can be built and

will return an empty list of results. No indication of the location of an error is given.

Both of these problems can be remedied by passing extra information around. In the basic ideas for passing this information around are discussed. As an example of the error correction is taken. The grammar analysis required for preventing unnecessary tryouts of alternatives is left out and can be found in [2].

### 3.2 Error recovery

Handling errors as well as performing grammar analysis requires a non-trivial definition of a parser. This section only serves to give an idea of the required structures and leaves out details. For an understanding of parser combinator usage and its Java counterpart, this section may be skipped.

A parser still is something which accepts input and produces a value as a result of parsing the given input. In the following definition of `Parser` this is expressed by `([s] -> Result b)` and `([s] -> Result (a,b))`, which are the functions performing the actual parsing.

```
type Result c = ((c,String),[Int])
type Parser s a b = ([s] -> Result b)
                  -> ([s] -> Result (a,b))
```

A difference is that output not only contains a result, but error messages (a `String`) and a list of costs (`Int`'s) as well. This is expressed in the definition of `Result`. Each element of the list of costs describes the cost of a parsing step.

Another difference is that each parser is given a continuation representing the stack of symbols that still have to be recognised. This can best be seen by looking at the definition of `pSucceed`:

```
addressult v ~(~(r,msgs), ss) = ((v,r), msgs), ss)
pSucceed v = \k input -> (addressult v) (k input)
```

The combinator `pSucceed` is a function accepting the continuation parser `k` and input. The combinator returns both the given `v` as well as the result of parsing the rest of the input via the invocation `k input`. The construction of this combination is performed in `addressult`. The result is obtained by applying the continuation to the input. The input is by definition not modified by `pSucceed` and passed along unmodified.

This definition for `pSucceed` and other parser combinators works because an expression like `k input` is lazily evaluated. So it can be referred to, and be returned as a result without being evaluated at all. Laziness becomes even more important when the result of `k input` is inspected. This happens when choices have to be made, in particular when a choice has to be made between error corrections. The given construction then allows to inspect the parsing future.

The combinator `pSym` is the combinator where the actual comparison with input takes place. Consequently, this is also the place where possible error corrections can be tried:

```

addstep s ~( v , ss) = (v , s:map (+s) ss)
addmsg m ~(~(r,msgs), ss) = (( r , m++msgs), ss)
insert a = address result a.addstep (penalty a)
          .addmsg (" Inserted:" ++ show a++"\n")
delete b =
          addstep (penalty b)
          .addmsg (" Deleted :" ++ show b++"\n")
pSym a k inp@(b:bs) | a == b = addstep 0.address result b.k $ bs
                    | otherwise = best ((insert a) (k inp))
                               ((delete b) (pSym a k bs))
pSym a k inp@[] = (insert a) (k inp)
penalty s = if s == '\EOT' then 1000 else (ord s -ord 'a')::Int
best = ...

```

Though more complicated than the previous version, `pSym` still inspects a symbol of the input. If a matching symbol is found it is added as a result (via `address result`) with zero cost (via `addstep`). If the end of the input is reached an insertion in the inputstream of the expected symbol will be made, followed by a parse attempt of the continuation `k` on the input. If the expected symbol does not match the actual input symbol two repair actions are possible. Either the expected symbol is missing and should be inserted, or the actual input symbol should be deleted from the inputstream. In all the cases where a correction is made, a non-zero cost (penalty) is added to the result. The correction attempts are tried and compared using the function `best`:

```

best left@(lvm, []) _ = left
best _ right@(rvm,[]) = right
best left@(lvm, 0:ls) right@(rvm, 0:rs) =
    addstep 0 (best (lvm, ls) (rvm, rs))
best left@(lvm, ls) right@(rvm, rs)
    = ( if (ls 'beats' rs) 4 then lvm else rvm
      , zipWith min ls rs)

([], 'beats' rs ) _ = True
(_ 'beats' [] ) _ = False
([1] 'beats' (r:_)) _ = 1 < r
((1:_)'beats' [r] ) _ = 1 < r
((1:ls)'beats' (r:rs)) n = (if n == 0 then 1 < r
                           else (ls 'beats' rs) (n-1))

```

The function `best` compares two parses by comparing their costs and choosing the parse with the lowest cost. If necessary, `best` looks into the 'future' until it finds non-zero costs. These are then compared, but only a limited number of steps ahead (here: 4) in order to avoid excessive tryouts of corrections.

Selecting between alternatives using `best` is seen more clearly in the definition of the choice combinator `<|>`:

```

p <|> q = \k input -> p k input 'best' q k input
p <*> q = \k input -> let ((pv, (qv, r)),m),st) = p (q k) input
                        in (((pv qv, r), m), st)
pFail = \_ _ -> ((undefined, []), repeat 10000)

```

The combinator `<*>` is relatively simple since no comparisons have to be made, only extraction of results and applying the result of `p` (`pv`) to the result of `p` (`pv`). The messages and the costs are passed unmodified.

Further discussion of the implementation of these parser combinators falls outside the scope of this paper. Part of its origin can be found in [16, 15]. However, it should be noted that the Haskell library for parser combinators is constructed along the lines discussed here, and also allows different functions `best` to be used. The Java version of the library consequently also allows this.

## 4 Parser combinators in Java

Writing a parser in Java, using parser combinators, (currently) boils down to compiling the Haskell definition to its Java equivalent by hand. A library of functions on top of a small lazy functional engine (section 5) has to be used for this purpose. This library provides the same functionality as the parser combinator library, combined with a minimal necessary subset of the Haskell prelude.

### 4.1 Lazy functional programming in Java

When using parser combinators, parsers are functions, and functions are represented by `Objects` which can be `apply`'d to arguments. Before we look at the Java equivalent of the Haskell expression parser combinators, we first show how lazy evaluation is realised in Java. The definition and usage of factorial in Haskell is used to show how this is done:

```
fac n = if n > 0
        then n * fac (n-1)
        else 1

main = fac 10
```

We will give several equivalents in Java with the purpose of showing how laziness can be used in varying degrees.

In Java, the factorial is normally (that is, imperatively) written as:

```
int fac( int n )
{
    if ( n > 0 )
        return n * fac( n-1 ) ;
    else
        return 1 ;
}
```

However, Java is a strict language, all arguments are computed before being passed to a method. This behavior has to be avoided because laziness is required instead. Since it is not possible to rely on basic Java evaluation mechanisms, basic functionality like integer arithmetic and method invocation is offered in a functional Java equivalent, packaged in a small Java library.

First, we have to define the factorial function. The Java library for the functional machinery contains a `Function` class which can be subclassed to define a new function. To be more precise, for `fac` we have to subclass `Function1`, a subclass of `Function` for defining one-argument functions. It is required to define the method `eval1` for the subclass of `Function1`. This method is used by the evaluation mechanism to perform the actual evaluation of a function once a parameter has been bound:

```
import uu.jazy.core.* ;
import uu.jazy.prelude.* ;

public class Fac
{
    static Eval fac =
        new Function1()
        {
            public Object eval1( Object n )
            {
                return
                    Prelude.ifThenElse.apply3
                    ( Prelude.gt.apply2( n, Int.Zero )
                    , Prelude.mul.apply2
                    ( n
                    , fac.apply1( Prelude.sub.apply2( n, Int.One ) )
                    )
                    , Int.One
                    ) ;
            }
        } ;
    ...
}
```

The definition uses the `core` package because the class `Function1` belongs to it. The `prelude` package offers a subset of the Haskell prelude. For example the test `n > 0` is written as `Prelude.gt.apply2( n, Int.Zero )`. Basically, all function definitions in a Haskell program are translated to subclasses of `Function` and all function applications are translated to the invocation of an appropriate variant of `apply` on an instance of such a subclass.

The big difference between the two given Java solutions is that the first one computes the result when invoked and the latter one creates an application data structure describing the computation. Only when explicitly asked for, this data structure is evaluated and returns the value represented by the data structure. This is done by calling the method `eval` from class `Eval`:

```
public class Fac
{
    ...

    public static void main(String args[])
```

```

    {
        System.out.println( fac( 10 ) ) ;
        System.out.println
            ( ((Int)Eval.eval( fac.apply1( Int.valueOf( 10 ) ) )).intValue() ) ;
    }
}

```

Both Java variants are shown for comparison. For the lazy variant, an application of `fac` to 10 is built by wrapping the integer in a `Int` object<sup>2</sup>. The function `fac` is then applied to this `Int` and the resulting application structure is passed to `eval` for evaluation. The result is known to be an `Int` and downcasted as such for printing.

The given program can be written in different varieties. For example, the library provides builtin `showing` of values used by the library. The last line of `main` could also have been written as

```
IO.showln( fac.apply1( Int.valueOf( 10 ) ) ) ;
```

The method `showln` offers the equivalent of Java's `println`, but for the lazy values used by the library.

It also is possible to mix the two programming paradigms. For example, it is not necessary to delay the computation of the `if n > 0` expression:

```

static Eval fac2 =
    new Function1()
    {
        public Object eval1( Object n )
        {
            if ( Int.evalToI( n ) > 0 )
                return
                    Prelude.mul.apply2
                        ( n
                          , fac2.apply1( Prelude.sub.apply2( n, Int.One ) )
                        ) ;
            else
                return Int.One ;
        }
    } ;

```

The decision made in the `if n > 0` expression has to be made anyway, so, it may as well be done immediately after entering `eval1`. The overhead of laziness is avoided by using the strict evaluation of Java. These optimisations eventually will lead to the Java only solution. It is up to the programmer and the need for laziness to decide how much laziness is required. As a conclusion of the discussion of these mechanisms a final variant as an optimisation example:

```

static Eval fac3 =
    new Function1()
    {

```

---

<sup>2</sup> `Int` is the equivalent of `java.lang.Integer`.

```

public Object eval1( Object n )
{
    int nn = Int.evalToI( n ) ;
    if ( nn > 0 )
        return
            Prelude.mul.apply2
                ( n
                  , fac3.apply1( Int.valueOf( nn-1 ) )
                ) ;
    else
        return Int.One ;
}
} ;

```

The lazy subtraction is replaced by a strict one. This can also be done for the multiplication.

As a final note, one can observe that all values manipulated by the library are `Object`'s. As a consequence typing information is irretrievably lost. In practice, this easily leads to difficult to detect bugs. This situation can best be avoided by first making the program work in Haskell and then compile it by hand to Java using this informally introduced compilation scheme.

## 4.2 Parser combinators

Let us now look at the definition of parser combinators in Java.

```

ParsingPrelude p = new ParsingPrelude( new ParsingListsCore() ) ;

/*
    pExpr = pChainl (    (+) <$ pSym '+'
                       <|> (-) <$ pSym '-'
                       )
                pTerm
*/
Object pExpr =
    p.pChainl
        ( p.pOr
            ( p.pAppL( Int.add, p.pSym( '+' ) )
              , p.pAppL( Int.sub, p.pSym( '-' ) )
            )
          , pTerm
        ) ;

```

This definition resembles the corresponding Haskell definition as closely as possible. First, a parsing library `ParsingPrelude` is constructed. It is necessary to do this because the parsing library itself is parameterised with the basic parser combinators. The derived combinators are built on top of these core combinators. In this case it is parameterised with the backtracking implementation using lists (section 3.1).

Wherever possible, the fact that functions are `Objects` is hidden by using Java wrapper methods. For example, the Java function `pChain1` actually is defined to apply its arguments to the Java representation of a Haskell function:

```
public final Eval pChain1 = ... ;

public final Eval pChain1( Object op, Object x )
{
    return pChain1.apply2( op, x ) ;
}
```

Other functions like integer addition are defined in separate libraries, most of them can be found in the `Prelude` class or a specific class associated with the type of a value. For example, the Java class `Int` defines the integer addition function `add` used in the definition of `pExpr`. This function also can be found in the `Prelude` but is defined more generically using a simple implementation of the Haskell class mechanism. This is not further explained here.

Functions can also be defined by subclassing from subclasses of class `Function`:

```
/*
    pNat = foldl (\a b -> a*10 + b) 0 <$> pList1 pDigit
*/
Object pNat =
    p.pApp
        ( Prelude.foldl
            ( new Function2()
                {
                    public Object eval2( Object a, Object b )
                    {
                        return
                            Int.valueOf
                                ( Int.evalToI(a) * 10
                                    + Int.evalToI(b) ) ;
                    }
                }
            , Int.Zero
        )
        , p.pList1( pDigit )
    ) ;
```

Here, a function taking two arguments, expressed by instantiating a subclass of the Java class `Function2`, is passed to `foldl`. This new subclass of `Function2` is required to define method `eval2`. The method `eval2` (and similar ones with similar names) computes the function result, in this case strictly by evaluating the result immediately.

Finally, the parser built using the preceding definitions is used with some input:

```
/*
    on :: Show a => Parser Char a -> [Char] -> IO ()
*/
```

```

on p inp -- run parser p on input inp
  = do let (res, msgs) = parse p inp
        putStr (if msgs == NoReports
                 then ""
                 else "Errors:\n" ++ show msgs)
        putStr ("Result:\n" ++ show res ++ "\n")

*/
protected static void on( ParsingPrelude parsing, Object parser, String inp )
{
    Tuple pres =
        Tuple.evalToT
            ( parsing.parse( parser, Str.valueOf( inp ) ) ) ;
    Object errors = Eval.eval( pres.second() ) ;
    if ( errors != Reports.NoReports )
        IO.putStr
            ( Prelude.concat2
              ( Str.valueOf( "Errors:\n" )
                , Prelude.show( errors )
              ) ) ;
    IO.showln( pres.first() ) ;
}

```

The Java method `on` contains calls to `eval`, either directly or indirectly via `evalToT` (evaluate to `Tuple`). As mentioned before, the method `eval` performs the actual computation of function applications. The preceding definitions only define the function applications but do not yet evaluate them.

## 5 Mapping lazy functional behavior to Java

To make parser combinators useable in Java we have chosen for a rather straightforward solution, namely to write down the Java solution in terms of functions. In order to be able to use functions as they are used in a functional language, we have to be able to treat them as first class citizens, that is, we have to be able to pass functions as parameters, and return them as result. In Java, the only kind of first class 'thing' available is `Object`. Therefore, functions are modelled as objects. Their basic usage has been shown in the previous section.

In a language like Java the application of a function to parameters, the evaluation of parameters and the evaluation of a function are performed in one action, the method invocation. For parser combinators, this does not work. Results of a parse are already used before they are completely evaluated (see section 3.2). A lazy implementation generally allows this. Though laziness is not always considered an essential ingredient of functional languages, it is essential to make our parser combinators work.

## 5.1 The basic lazy functional engine

Lazy implementations of functional languages come in different flavours [11] among which the STG implementation [12] is considered to be the fastest. This approach is also taken by [17] to compile for the Java virtual machine. The approach taken here is to provide a graph reduction engine, without explicit usage of a stack, constructed in such a way that the Java machinery is used as effectively as possible while at the same time offering ease of use from the Java programmers point of view.

As a starting point, let us look at the following Haskell program:

```
addOne :: [Int] -> [Int]
addOne [] = []
addOne (1:ls) = 1+1 : addOne ls

main = addOne [1,2,3]
```

The function `addOne` takes a list of integers and returns a list where each integer element has been incremented by 1. The result of `main` is `[2,3,4]`. Using the lazy functional Java library, the definition of `addOne` is expressed in Java as:

```
static Eval addOne =
    new Function1()
    {
        public Object eval1( Object l )
        {
            List ll = List.evalToL( l ) ;
            if ( ll.isEmpty() )
                return List.Nil ;
            else
                return
                    List.Cons
                        ( Prelude.add.apply2( ll.head(), Int.One )
                          , addOne.apply1( ll.tail() )
                        ) ;
        }
    } ;
```

The function uses the predefined class `List` to construct a new list. `Nil` denotes the empty list. The method `Cons` constructs a new cons cell with head and tail; it is the equivalent of Haskell's `:`. The function is then used by passing it the list `[1,2,3]` which is a convenient notation for `(1:(2:(3:[])))` where `:` constructs a cons cell used in list representations. The application of `addOne` to the list is subsequently evaluated and shown via `IO.showln`.

```
import uu.jazy.core.* ;
import uu.jazy.prelude.* ;

public class AddOne
{
    ...
}
```

```

public static void main(String args[])
{
    List l123 = List.Cons
        ( Int.One, List.Cons
          ( Int.Two, List.Cons
            ( Int.Three, List.Nil
              )
            )
          ) ;
    IO.showln( addOne.apply1( l123 ) ) ;
}
}

```

The function `addOne` can also be written using a Java method. The recursive invocation of `addOne` then is done before the result of `addOne` is returned; the essential difference can be found in the line marked with `/*<-*`.

```

static Object addOne( Object l )
{
    List ll = List.evalToL( l ) ;
    if ( ll.isEmpty() )
        return List.Nil ;
    else
        return
            List.Cons
                ( Prelude.add.apply2( ll.head(), Int.One )
                  , addOne( ll.tail() )
                ) ;
}

public static void main(String args[])
{
    List l123 = ...
    IO.showln( addOne( l123 ) ) ;
}

```

This produces the same result for the list `[1,2,3]` but is computed in a different way. The recursive invocation of `addOne` is done strictly (non-lazy), by computing the result of the invocation before passing it to the list constructor `List.Cons`. Alternatively, strictness could also have been achieved by replacing the line marked with `/*<-*` by:

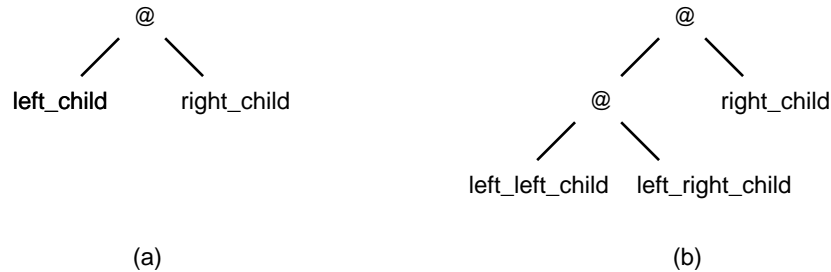
```

static Eval addOne2 =
    new Function2()
    {...
        , eval( addOne2.apply1( ll.tail() ) ) /*<-*
    ...}

```

The strict evaluation order is enforced by the invocation of `eval`.

Strict evaluation order can be encoded more efficiently, but poses two problems when compared with lazy computation order. First, strict evaluation order may compute more than is necessary. For example, suppose that from `addOne`



**Fig. 6.** Reduction graph.

---

[1,2,3] only the first element is needed. Using a strict evaluation order all elements are computed before the result [2,3,4] is returned, whereas lazy evaluation returns the partially evaluated list (2:(addOne 2:(3:[]))) instead.

A second problem arises when `addOne` is passed an infinite list, as in

```
take 5 (addOne (repeat 1))
```

The Haskell function `repeat` produces an infinite list of 1's. The function `take` takes a certain amount of elements from a list (passed as arguments), here producing the result [2,2,2,2,2]. Because strict evaluation evaluates the argument to `addOne` first an attempt is made to compute the infinite list [1,1,...]. In Java, the invocation of `addOne` prints the expected output, but the strict variation `addOne2` gives a stack or memory overflow:

```
Object lInfinite = Prelude.repeat.apply1( Int.One );
IO.showln( Prelude.take.apply2( Int.Five, addOne.apply1( lInfinite ) ) );
IO.showln( Prelude.take.apply2( Int.Five, addOne2.apply1( lInfinite ) ) );
```

Figure 7(a) shows a graph representation of `addOne [1,2,3]`. The graph encodes the structure of the computation of `addOne [1,2,3]`. In this graph representation nodes either consists of an application, denoted by @, or a node consists of plain data. Figure 6 shows a simple usage of @. The @ should be read as “apply the left child to the right”, as indicated in figure 6(a). Its Java equivalent is

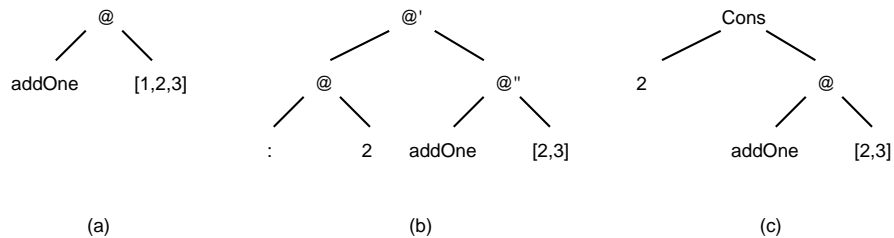
```
left_child.apply1( right_child )
```

If the left child itself is also such an application (figure 6(b)) it reads as

```
(left_left_child.apply1( left_right_child )).apply1( right_child )
```

which is equivalent to

```
left_left_child.apply2( left_right_child, right_child )
```



**Fig. 7.** Reduction graphs for `addOne`.

---

A computation step consists of the evaluation of a `@` node in the graph describing the computation. The evaluation process replaces a `@` by its result, which may be a plain value or yet another `@` node. In figure 7(c) the result `(2:(addOne 2:(3:[])))` of the application `addOne [1,2,3]` can be seen. The graph for `addOne [1,2,3]` has been replaced by its result consisting of a `Cons` cell. The `Cons` cell still contains an unevaluated value, the remaining application `addOne [2,3]`.

The `Cons` cell is considered to be in weak head normal form, because the cell itself cannot be further evaluated, even though it still refers to unevaluated applications. Only when the value of such an unevaluated application really is needed, it has to be evaluated.

The difference between strictness and laziness can be found at the intermediate stage of the computation of `addOne [1,2,3]`, shown in figure 7(b). The evaluation of `@'` returns the `Cons` cell. Strictness requires the right child `@'` of `@'` to be evaluated before `@'` is evaluated, laziness does not.

Such is the power and convenience of laziness. This is consequently also the mechanism which has to be imitated by the Java lazy functional library.

Figure 8 shows the Java class structure used to model a reducible graph. Figure 9 shows instances of such graphs, the Java counterparts of figure 7. The graph is used by an evaluator which considers anything except `Apply` objects to be non-reducible (normal form).

The structure of the class diagram as well as its interpreter are derived from a small evaluator for lambda expressions, see figure 10 for an overview of the core functionality of the evaluator written in Haskell and figure 11 for the Java variant. Only the Java variant is discussed here. The basic idea of the evaluator is that it is given a graph representing a computation. This graph contains either application nodes, that is, instances of (a subclass of) class `Apply`, or it contains something else. Only if a node is an application node further computation steps are taken, otherwise it cannot be evaluated further and the node is simply returned. This work is done in the method `eval`:

```
public static Object eval( Apply av )
{
```

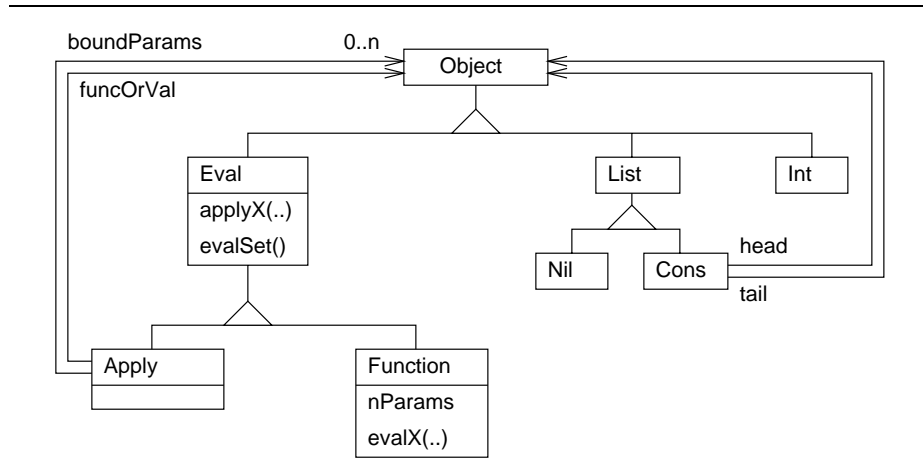


Fig. 8. Class structure for lazy objects.

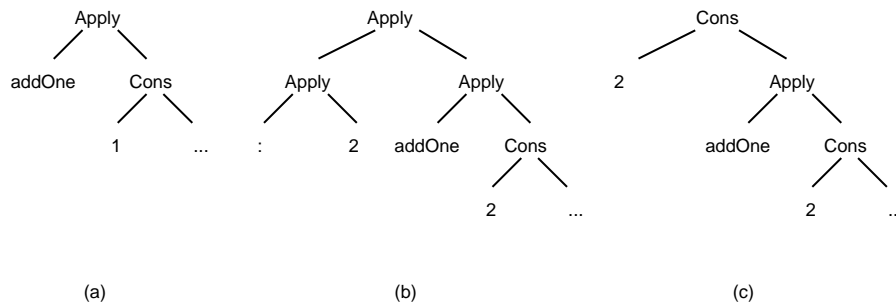


Fig. 9. Example of a reducible graph.

```

    if ( av.nrNeededParams == 0 )
    {
        av.evalSet() ;
        Object vv = av.funcOrVal ;
        av.nrNeededParams = -1 ;
        if ( vv instanceof Apply )
            return av.funcOrVal = eval( (Apply)vv ) ;
    }
    else if ( av.nrNeededParams > 0 )
        return av ;
    else
        return av.funcOrVal ;
}

public static Object eval( Object v )
{
    if ( v instanceof Apply )
        return eval( (Apply)v ) ;
    return v ;
}

```

An `Apply` object contains a field `nrNeededParams` used for remembering the state an `Apply` instance is in. A value  $< 0$  is used to indicate that it is already evaluated,  $> 0$  means that not enough arguments are available and  $= 0$  means that it should be evaluated. The actual evaluation is delegated to the `Apply` object itself via method `evalSet`. This allows subclasses of `Apply` to provide an optimised version of their evaluation.

If the result of the evaluation is another `Apply`, the process is repeated, in this case by recursively invoking `eval`.<sup>3</sup>

The default implementation of `evalSet` extracts the left child of the of an `Apply` node in the reduction graph. This left child result in a function, which is subsequently applied to its arguments:

```

public abstract class Apply extends Eval
{
    protected Object funcOrVal ;
    protected int nrNeededParams = 0 ;

    protected void evalSet()
    {
        funcOrVal = ((Eval)eval(funcOrVal)).evalOrApplyN
                    ( getBoundParams() ) ;
    }
}

```

If the left child does not evaluate to a function it is an error. The application of the function to its arguments (retrieved via `getBoundParams`) either performs

---

<sup>3</sup> The library contains evaluator variants which avoid the usage of the Java stack by reversing pointers between `Apply` nodes.

---

```

data Expr = Var String
          | Val Int
          | Nil
          | Cons Expr Expr
          | ApplyN Expr [Expr]
          | FunctionN [String] Expr

substN :: Env -> Expr -> Expr
substN env e@(Var s          )
  = case lookup s env of {Nothing -> e; Just v -> v}
substN env (ApplyN func args )
  = ApplyN (substN env func) (map (substN env) args)
substN env l@(FunctionN formals body )
  = FunctionN formals (substN (filter (not.('elem' formals).fst) env) body)
substN _ e
  = e

eval appn@(ApplyN func args)
  = let (val, restargs)
      = case eval func of
          (FunctionN formals body)
            -> ( eval (substN (zip formals args) body)
                , (drop (length formals) args)
              )
        in if null restargs
           then val
           else eval (ApplyN val restargs)
eval whnf
  = whnf

```

**Fig. 10.** Lambda expression evaluator.

---

the actual function evaluation, if enough arguments are available, or returns a new application node:

```
public abstract class Function extends Eval
{
    protected int nrParams ;
    protected abstract Object evalN( Object[] vn ) ;

    protected Object evalOrApplyN( Object[] vn )
    {
        Object res = null ;
        if ( nrParams > vn.length )
            res = applyN( vn ) ;
        else
        {
            res = ((FunctionN)this).evalN
                ( Utils.arrayTake( nrParams, vn ) ) ;
            if ( nrParams != vn.length )
                res = ((Eval)res).applyN
                    ( Utils.arrayDrop( nrParams, vn ) ) ;
        }
        return res ;
    }
}
```

If there are leftover arguments, these are applied to the result of the function application. Further evaluation is performed by the method `eval`.

The actual function invocation is done by method `evalN`. A subclass of `Function` is expected to implement this method. In practice, the Java lazy functional library offers convenience classes and methods for functions with 1, 2, 3, 4, 5, or more (N) arguments. The corresponding names of the function classes consist of “Function” suffixed with the number of arguments taken. The names of the evaluation methods defined by the ‘Functional’ programmer use “eval” as a prefix, as already shown in previous examples.

## 5.2 Optimisations

The basic implementation as shown in figure 11 can be significantly improved in terms of efficiency by exploiting knowledge about the number of required and given parameters for a `Function`. For example, the addition of two `Int`’s, assuming the existence of attribute `value` holding the integer value, can be defined as:

```
new Function2()
{
    protected Object eval2( Object v1, Object v2 )
    {
        return
            new Int
                ( ((Int)eval(v1)).value
```

---

```

public abstract class Eval
{
    public Apply applyN( Object[] vn )
    {
        return new ApplyN( this, vn ) ;
    }

    public static Object eval( Apply av )
    {
        if ( av.nrNeededParams == 0 )
        {
            av.evalSet() ;
            Object vv = av.funcOrVal ;
            av.nrNeededParams = -1 ;
            if ( vv instanceof Apply )
                return av.funcOrVal = eval( (Apply)vv ) ;
        }
        else if ( av.nrNeededParams > 0 )
            return av ;
        else
            return av.funcOrVal ;
    }

    public static Object eval( Object v )
    {
        if ( v instanceof Apply )
            return eval( (Apply)v ) ;
        return v ;
    }

    public abstract Object[] getBoundParams() ;
}

public abstract class Function extends Eval
{
    protected int nrParams ;

    protected abstract Object evalN( Object[] vn ) ;

    protected Object evalOrApplyN( Object[] vn )
    {
        Object res = null ;
        if ( nrParams > vn.length )
            res = applyN( vn ) ;
        else
        {
            res = ((FunctionN)this).evalN
                ( Utils.arrayTake( nrParams, vn ) ) ;
            if ( nrParams != vn.length )
                res = ((Eval)res).applyN
                    ( Utils.arrayDrop( nrParams, vn ) ) ;
        }
        return res ;
    }
}

public abstract class Apply extends Eval
{
    protected Object funcOrVal ;
    protected int nrNeededParams = 0 ;

    protected void evalSet()
    {
        funcOrVal = ((Eval)eval(funcOrVal)).evalOrApplyN( getBoundParams() ) ;
    }
}

class ApplyN extends Apply
{
    protected Object[] pN ;

    public ApplyN( Object f, Object[] p )
    {
        super( f ) ;
        pN = p ;
    }

    public Object[] getBoundParams()
    {
        return pN ;
    }
}

```

**Fig. 11.** Lambda expression evaluator in Java (a sketch of).

---

```

        + ((Int)eval(v2)).value
      ) ;
    }
  } ;

```

It now is statically known that this function takes exactly two arguments. If an application `ApplyN` only contains one argument for the function, the evaluator does not need to evaluate the application. The call to `evalSet` can then be avoided. If exactly two arguments are passed, an even greater positive effect on performance can be achieved by redefining the method `evalSet` to call the method `eval2` of the `Function2` directly. In this way the overhead of `evalOrApplyN` can be avoided.

Both cases (not enough and exact number of arguments) are optimised by the definition of appropriate subclasses, shown in figure 12 for `Function2`. The first case -not enough arguments- is dealt with by administering that still one argument more is needed; `nrNeededParams` is set to 1:

```

public abstract class Function2 extends Function
{
    public Apply apply1( Object v1 )
    {
        return new Apply1F2( this, v1 ) ;
    }
}

class Apply1F2 extends Apply1
{
    public Apply1F2( Object f, Object p1 )
    {
        super( f, p1 ) ;
        nrNeededParams = 1 ;
    }
}

```

The second case -exact number of arguments- is dealt with by redefining `evalSet`:

```

public abstract class Function2 extends Function
{
    public Apply apply2( Object v1, Object v2 )
    {
        return new Apply2F2( this, v1, v2 ) ;
    }
}

class Apply2 extends Apply
{
    protected Object p1, p2 ;
    ...
}

```

```

class Apply2F2 extends Apply2
{
    protected void evalSet()
    {
        funcOrVal = ((Function2)funcOrVal).eval2( p1, p2 ) ;
    }
}

```

The method `evalSet` is now defined more efficiently.

### 5.3 Performance

Due to the interpretative nature of the implementation, a parser using parser combinators in Java is no speed demon. No extensive testing has been done, so only an indication of performance is given. For testing, a small parser copying input characters to output was used, giving the following measurement on a 4940 byte file:

```
3198 ms., 275832 evaluations, 0.011594013 ms. per eval
```

Each character took approximately 56 evaluations (calls to `evalSet`). The test was performed on an Apple Powerbook G3 with a 500Mhz PowerPC running Java 1.1.8. The error correcting limited lookahead variant of the parser combinators was used. The obtained speed is roughly equivalent to (some 25% slower than) the speed of running the interpreted Haskell variant using Hugs on the same platform.

## 6 Conclusions

Parser combinators with error correction and on-the-fly grammar analysis allow the grammar writer an easy, flexible way of writing an executable grammar. Though performance in Haskell using a Haskell compiler has an acceptable performance this cannot be said for a straightforward Java implementation based on the literal translation in this paper.

Another aspect is that all information about the type of a parser is lost because of the limitations of the Java typing system. All values as manipulated by parser combinators are of type `Object`, or at best of those displayed in figure 8. This makes use of parser combinators typeless, generally leading to difficult to detect bugs. It is advisable to first make a working Haskell version and then compile this by hand to Java, or, preferably, let a compiler do this work.

With these observations in mind, the authors feel that there may well be a place for parser combinators in Java, as described here. Especially when performance is a lesser issue, but flexibility is more important, for example in interactive systems where compilation is done for small pieces at a time. Furthermore, a better efficiency can be achieved by performing as much as possible in Java. A first candidate would be tokenisation, not necessarily a task requiring laziness.

---

```
public abstract class Function2 extends Function
{
    public Apply apply1( Object v1 )
    {
        return new Apply1F2( this, v1 ) ;
    }

    public Apply apply2( Object v1, Object v2 )
    {
        return new Apply2F2( this, v1, v2 ) ;
    }
}

class Apply2 extends Apply
{
    protected Object p1, p2 ;
    ...
}

class Apply1F2 extends Apply1
{
    public Apply1F2( Object f, Object p1 )
    {
        super( f, p1 ) ;
        nrNeededParams = 1 ;
    }
}

class Apply2F2 extends Apply2
{
    public Apply2F2( Object f, Object p1, Object p2 )
    {
        super( f, p1, p2 ) ;
    }

    protected void evalSet()
    {
        funcOrVal = ((Function2)funcOrVal).eval2( p1, p2 ) ;
    }
}
```

**Fig. 12.** Expression evaluator optimisations for Function2.

---

## References

1. JavaCC. <http://www.metamata.com/>, 2001.
2. Software Technology. <http://www.cs.uu.nl/groups/ST/Software/index.html>, 2001.
3. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.
4. Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
5. Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.
6. Jeroen Fokker. *Functional Parsers*. Utrecht University, Institute of Information and Computing Sciences, 1995.
7. Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2:323–343, July 1992.
8. Graham Hutton and Erik Meijer. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8, 1998.
9. Mark P. Jones and John C. Peterson. *Hugs 98. A functional programming system based on Haskell 98. User Manual*. Oregon Graduate Institute, 1999.
10. Daan Leijen. *Parsec, a fast combinator parser*. Utrecht University, Institute of Information and Computing Sciences, 1999.
11. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
12. Simon L. Peyton-Jones. *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*. Department of Computer Science, University of Glasgow, 1992.
13. Simon Peyton Jones (editor) and John Hughes (editor). Report on the Programming Language Haskell 98. A Non-strict, Purely Functional Language, 1999.
14. Rinus Plasmeijer and Marko v. Eekelen. *Concurrent Clean. Language Report. Version 1.3*. HILT B.V. and University of Nijmegen, 1998.
15. S.D. Swierstra. Parser Combinators: from Toys to Tools. In *Haskell Workshop*, 2000.
16. S.D. Swierstra and P.R. Azero Alcocer. Fast, error correcting parser combinators: A short tutorial. In *SOFSEM'99, 26th Seminar on Current Trends in Theory and Practice of Informatics*, pages 111–129, November 1999.
17. D. Wakeling. Compiling Lazy Functional Programs for the Java Virtual Machine. *Journal of Functional Programming*, 9:579–603, Nov 1999.