Parsec, a fast combinator parser

DAAN LEIJEN
University of Utrecht
Dept. of Computer Science
PO.Box 80.089, 3508 TB Utrecht
The Netherlands
daan@cs.uu.nl, *http://www.cs.uu.nl/~daan*

4 Oct 2001

1 Introduction

Parsec is an industrial strength, monadic parser combinator library for Haskell. It can parse *context-sensitive*, *infinite look-ahead* grammars but it performs best on predictive (LL[1]) grammars. Combinator parsing is well known in the literature and offers several advantages to YACC or event-based parsing.

- Combinator parsers are written and used within the same programming language as the rest of the program. There is no gap between the grammar formalism (Yacc) and the actual programming language used (C). Users learn just one language! We can also enjoy all the benefits that exist in the host language: type checking, module system, development environment, etc.
- Parsers are first-class values within the language. They can be put into lists, passed as parameters and returned as values. It is easy extend the available set of parsers with custom made parsers specific for a certain problem, see for example `many` or `buildExpressionParser`.

Most 'real' parsers are built using bottom-up parser generators like happy. This is mostly for reasons of speed but also because most parser libraries are written for research and inadequate for practical applications. Parsec is designed as a 'real world' library from scratch:

- *Speed.* Most combinator libraries lack the speed necessary to be competetive with bottom-up parser generators. Parsec uses some novel techniques to improve its performance. The library is fast, parsing thousands of lines a second on today's machines (see the benchmark for test results).
- *Error reporting.* Parsec has extensive error messages; giving position, unexpected input, expected productions and general user messages. The error messages are given in terms of high-level grammar productions and can be localized for different (natural) languages.
- *Libraries.* Parsec provides a set of basic parsers and provides three extension libraries, a lexical analysis library, a generalized expression parser and a library for parsing permutation phrases.
- *Documentation.* Parsec is documented with a reference- and user-guide and example grammar implementations.

- *Simplicity.* Parsec is designed to be as simple as possible, making it accessable to both novice and expert users. Parsec is predictive by default, only backtracking where an explicit `try` is inserted. This makes the operational behaviour easier to understand than the 'normal' approach where backtracking is the default behaviour.

1.1 About this document

This document ships in the following formats:

- *Postscript (PS).* High quality output, best for printing.
- *Acrobat (PDF).* High quality output and hyperlinked. Preferred for online browsing.
- *HTML.* Poor man's PDF – hyperlinked but it lacks an index.

After the introduction, this document contains a simple user guide and a than a full reference guide for the library.

1.2 Compatibility

The core library is written in Haskell98. The extension library `ParsecToken` uses the `forall` keyword (to emulate a first-class module) but, to my best knowledge, this extension is supported by every Haskell98 system. The library is successfully compiled with GHC 4.x, GHC 5.x, Hugs98 and nhc98, but probably works with any Haskell98 compliant compiler or interpreter.

1.2.1 Compiling with GHC

Parsec is distributed as a package with GHC. This means that you can simply use the `-package text` switch to use the Parsec library. Unfortunately, Ghc versions up to 5.02 include an old version of Parsec. In this case, you might want to link with the newest Parsec library. Parsec ships with a makefile that creates a Haskell library – just type `make` at the command prompt.

Suppose that the library is compiled in the directory `c:\parsec`. When you compile a Haskell source that uses Parsec, you should tell GHC where it can find the interface files (`*.hi`).

```
ghc -c myfile -ic:\parsec
```

When your linking the files together, you need to tell GHC where it can find libraries (`-L`) and to link with the Parsec library too (`-l`):

```
ghc -o myprogram myfile1.o myfile2.o -Lc:\parsec -lparsec
```

1.2.2 Compiling with Hugs

If you use the `ParsecToken` or `ParsecPerm` modules, you need to enable the `forall` extension. Hugs should be started with the `-98` option to enable Haskell98 extensions.

1.3 Reporting bugs

Any bugs (or fixes!) can be reported to the author, Daan Leijen (`daan@cs.uu.nl`). Any feedback on:

- Which parsers are lacking;
- Naming of the parsers;
- Performance, both time and space.

is greatly appreciated too. Besides that, I am also quite interested in any applications, especially in a commercial setting, that are created using this library.

The PARSEC homepage can be found at: `http://www.cs.uu.nl/~daan/parsec.html`. Any useful additions to the library or example grammars will be put on this www-page. If you have written a parser for some language, don't hesitate and send it to author so it can be put on the www-page too.

The library has been tested using the *Pbench* benchmark. It can be found at `http://www.cs.uu.nl/people/daan/pbench.html`.

1.4 History

- 4 Oct 2001: Parsec can now work on any input stream which allows one to write a seperate lexer. There is also a user definable state – for example for creating identifiers from strings. Non-compatible changes:
  - `brackets` is now called `angles`, while `squares` is now called `brackets`.
  - The type of many parsers is now more general, ie. `GenParser tok st a` instead of `Parser a`.
  - The types of `Operator` and `OperatorTable` have changed into a more general type.
  - The `ParsecToken` module is now parameterized with a record instead of an imported module.
  - The `token` parser now parses single tokens.
  - Added new functionality, like `sepEndBy`.
  - Added an experimental module for parsing permutations (`ParsecPerm`).

1.5 License

I would like to see this library as widely used as possible. It is therefore distributed with an unrestrictive BSD style license. If there is any problem using this software with this license, I would like to hear about it and see if we can work out a solution.

2 Users guide

All the examples given in this user guide can be found in the `parsec/examples/userguide` directory of the parser library.

2.1 Running a parser

First of all, import the library into your Haskell module:

```
module Main where

import Parsec
```

Let's start with a very simple parser. The parser that recognizes a single letter (a character between `'a'` and `'z'`) is written as:

```
simple :: Parser Char
simple  = letter
```

The type of `simple` shows that `simple` is a parser that will return a `Char` (the parsed letter) whenever it succeeds. A parser can be executed using the `parse` function. This function takes three arguments: the parser (`simple`), the name of the input (`""`) and the input itself. The name of input is only used for error messages and can be empty in our case. The result of `parse` is either an error (`Left`) or a succesful result (`Right`). We define a simple `run` function that simply prints either the error value or the result value for a specific parser:

```
run :: Show a => Parser a -> String -> IO ()
run p input
        = case (parse p "" input) of
            Left err -> do{ putStr "parse error at "
                          ; print err
                          }
            Right x  -> print x
```

(The `run` function is actually defined in the `Parsec` module as `parseTest`). When we load this module into hugs, we can directly and interactively test our parsers on different inputs:

```
Reading file "d:\cvs\parsec\examples\userguide\main.hs":

Hugs session for:
d:\programs\hugs98\lib\Prelude.hs
d:\programs\hugs98\lib\Maybe.hs
d:\programs\hugs98\lib\List.hs
d:\cvs\parsec\ParsecError.hs
d:\programs\hugs98\lib\Monad.hs
```

```
d:\programs\hugs98\lib\Char.hs
d:\cvs\parsec\Parsec.hs
d:\cvs\parsec\examples\userguide\Main.hs
Main> run simple "a"
'a'

Main> run simple ""
parse error at (line 1, column 1):
unexpected end of input
expecting letter

Main> run simple "123"
parse error at (line 1, column 1):
unexpected "1"
expecting letter
```

2.2 Sequence and choice

Two important operations for specifying grammars (and parsers) are sequence and choice. In our case, parsers can be sequenced using the monadic `do`-notation. For example, the following parser parses an open parenthesis followed by a closing parenthesis (`"()"`):

```
openClose :: Parser Char
openClose = do{ char '('
              ; char ')'
              }
```

Two simple parenthesis are not too interesting but in combination with the choice operator `(<|>)` we can specify a parser that recognizes matching pairs of parenthesis. Matching parenthesis consist of either an open parenthesis followed by a matching pair of parenthesis, a closing parenthesis and another matching pair of parenthesis, or it is empty; The empty alternative is implemented using the `(return x)` parser which always succeeds with value `x` without consuming any input.

```
parens  :: Parser ()
parens  = do{ char '('
            ; parens
            ; char ')'
            ; parens
            }
        <|> return ()
```

Let's run the `parens` parser on some input to test its behaviour:

```
Main> run parens "(())()"
()

Main> run parens "(()()"
parse error at (line 1, column 6):
unexpected end of input
expecting "(" or ")"
```

2.3 Predictive parsers

For reasons of speed, the `(<|>)` combinator is *predictive*; it will only try its second alternative if the first parser hasn't consumed any input. Consider the following example parser:

```
testOr  =   string "(a)"
        <|> string "(b)"
```

The `(string s)` parser accepts input that equals the string `s`. Since both strings in the example start with the same initial character, the behaviour is probably not what the user expected:

```
Main> run testOr "(b)"
parse error at (line 1, column 2):
unexpected 'b'
expecting 'a'
```

Since the first parser has already consumed some input (the `'('`), the second alternative is never tried allthough it would succeed! The preferred solution to this problem is to *left-factor* the grammar by merging common prefixes. The following parser works as expected:

```
testOr1 = do{ char '('
            ; char 'a' <|> char 'b'
            ; char ')'
            }
```

Unfortunately, it is not always possible to left-factor the grammar and in many other cases the grammar becomes more complex to understand. There is a primitive combinator called `try` that handles these cases in an elegant manner. The parser `(try p)` behaves just like `p` except that it pretends that it hasn't consumed any input whenever `p` fails. In combination with `(<|>)` this allows for infinite look-ahead in the grammar. Here is the above parser written using `try`:

```
testOr2 =   try (string "(a)")
        <|> string "(b)"
```

or an even better version:

```
testOr3 =   do{ try (string "(a"); char ')'; return "(a)" }
        <|> string "(b)"
```

Allthough the implementation of `try` is quite efficient there is a penalty when it fails, ie. it shouldn't be used as an alternative to careful grammar design!

2.4 Adding semantics

Right now, the `parens` parser simply recognizes matching pairs of parenthesis but doesn't return a useful value. We will now extend the `parens` parser with 'semantic' actions which compute the maximal nesting level of the parenthesis. The `(<-)` operator will be used to bind the intermediate values returned by the parsers.

The `nesting` parser returns the nesting level of the parenthesis. The empty alternative returns zero. The other alternative returns the maximal nesting level of both other parsers:

```
nesting :: Parser Int
nesting = do{ char '('
            ; n <- nesting
            ; char ')'
            ; m <- nesting
            ; return (max (n+1) m)
            }
        <|> return 0
```

Here is an example session with `nesting`:

```
Main> run nesting "(())()"
2

Main> run nesting "(()(()))"
3

Main> run nesting "(()(())"
parse error at (line 1, column 8):
unexpected end of input
expecting "(" or ")"
```

2.5 Sequences and seperators

Consider the `word` parser that parses words: a sequence of one or more letters:

```
word    :: Parser String
word    = do{ c  <- letter
            ; do{ cs <- word
                ; return (c:cs)
                }
              <|> return [c]
            }
```

After parsing at a letter, it is either followed by a word itself or it returns the single character as a string. Allthough it is not too hard to learn to recognize these patterns, it is not immediately obvious that a sequence of one or more letters is parsed by this grammar. Even widely used parser generators like YACC and JavaCC do not provide any abstraction from these commonly occurring patterns. Luckily, we have the full power of functional programming at our finger tips, it is easy to provide a custom parser combinator that parses a sequence of one or more parsers; it is called `many1`:

```
word    :: Parser String
word    = many1 letter
```

Now it is obvious from the code what `word` is supposed to do and it is much easier to specifiy; no strange recursive patterns! Beside `many1`, the library provides the `many` parser which parses *zero* or more occurrences of its argument. The combinators `skipMany` and `skipMany1` behave like `many` and `many1` but do not return the values of their argument parser.

Another useful pair of combinators are `sepBy` and `sepBy1` which parse a sequence of parsers separated by some separator. The combinators `sepEndBy` and `sepEndBy1` parse a sequence of parsers that are either all seperated or terminated by some seperator (ie. Haskell declarations).

For example, sentences are a sequence of one or more words separated by one or more spaces or commas and ended by a dot, question mark or exclamation mark:

```
sentence    :: Parser [String]
sentence    = do{ words <- sepBy1 word separator
                ; oneOf ".?!"
                ; return words
                }

separator   :: Parser ()
separator   = skipMany1 (space <|> char ',')
```

Let's parse some example sentences:

```
Main> run sentence "hi,di,hi."
["hi","di","hi"]

Main> run sentence "hi,di hi!"
["hi","di","hi"]

Main> run sentence "hi,123"
parse error at (line 1, column 4):
unexpected "1"
expecting ",", space or letter
```

2.6 Improving error messages

Allthough the last error message of the previous hugs session is correct, it is not very user friendly. A better message would simply be `"expecting word"`. The error combinator `(<?>)` attaches a high-level error description to parsers. Errors can now be given in terms of high-level grammar productions instead of at the level of characters. We will first attach a description to the `word` parser:

```
word    :: Parser String
word    = many1 letter <?> "word"
```

Whenever the `word` parser fails without consuming input, it will now issue an error message `"expecting word"`:

```
Main> run sentence "hi,123"
parse error at (line 1, column 4):
unexpected "1"
expecting ",", space or word
```

It is easy to get rid of the comma and space messages; just use an empty description:

```
separator   :: Parser ()
separator   = skipMany1 (space <|> char ',' <?> "")
```

Note that the `(<|>)` combinator binds stronger than the `(<?>)` operator. The error message only reports the expected word now:

```
Main> run sentence "hi,123"
parse error at (line 1, column 4):
unexpected "1"
expecting word
```

Another improvement can be made to the 'end of sentence' parser:

```
sentence    :: Parser [String]
sentence    = do{ words <- sepBy1 word separator
                ; oneOf ".?!" <?> "end of sentence"
                ; return words
                }
```

Let's test the new parser on an uncompleted sentence:

```
Main> run sentence "hi,di"
parse error at (line 1, column 6):
unexpected end of input
expecting letter or end of sentence
```

Allthough the message is again correct, one further improvement can be made. The expected letter is a bit confusing; indeed a letter can be added to the last word but it is probably not what the user intended. Note that the message reports that a letter can be expected and not a word. This is because a message that is attached to a parser using `(<?>)` will only be issued if the parser can not even be started, i.e. when it hasn't consumed any input. Since the word parser has already consumed two letters (`di`) the expect message for letter is issued.

What we want to do is to attach an empty message to the `letter` parser in `word` but maintain the `"word"` message on the parser as a whole.

```
word    :: Parser String
word    = many1 (letter <?> "") <?> "word"
```

Finally we have a highly customised error messages which behave quite nicely:

```
Main> run sentence "hi di"
parse error at (line 1, column 6):
unexpected end of input
expecting end of sentence

Main> run sentence "hi di,"
parse error at (line 1, column 7):
unexpected end of input
expecting word
```

After this section, it might seem that error handling is quite involved, but don't be discouraged; This example grammar was chosen to be tricky and experience shows that for most grammars just a few error descriptions are necessary.

2.7 Expressions

Expressions are probably one of the most common constructs for which a parser is needed. Consider the following standard EBNF grammar for expressions:

```
expr   ::= expr '+' term | term
term   ::= term '*' factor | factor
factor ::= '(' expr ')' | digit+

digit  ::= '0' | '1' | ... | '9'
```

The grammar uses different productions to specify priorities (`'*'` has higher priority than `'+'`) and uses left-recursion to make both star and plus left-associative.

Unfortunately, left-recursive grammars can not be specified directly in a combinator library. If you accidently write a left recursive program, the parser will go into an infinite loop! However, every left-recursive grammar can be rewritten to a non-left-recursive grammar. The library provides combinators which do this automatically for you (`chainl` and `chainl1`).

Even when using these combinators to deal with the left-recursion, it is still a nuisance to code the associativity and priorities of the operators in such a primitive way. The parser library provides a standard module ParsecExpr to deal more elegantly with expression style grammars. Using this library, the above expression grammar, extended with minus and divide, is specified using the buildExpressionParser function:

```
import ParsecExpr

expr    :: Parser Integer
expr    = buildExpressionParser table factor
        <?> "expression"

table   = [[op "*" (*) AssocLeft, op "/" div AssocLeft]
          ,[op "+" (+) AssocLeft, op "-" (-) AssocLeft]
          ]
        where
          op s f assoc
             = Infix (do{ string s; return f}) assoc

factor  = do{ char '('
            ; x <- expr
            ; char ')'
            ; return x
            }
        <|> number
        <?> "simple expression"

number  :: Parser Integer
number  = do{ ds <- many1 digit
            ; return (read ds)
            }
        <?> "number"
```

The function buildExpressionParser takes two arguments. The second argument is the basic expression term and the first a table of operators of decreasing priority (the higher in the list, the higher the priority). Operators can be Infix, Prefix or Postfix. Infix operators also have an associativity: AssocLeft, AssocRight or AssocNone.

Here is an example session with the `expr` parser:

```
Main> run expr "1+2*3"   -- '*' has higher priority
7

Main> run expr "(1+2)*3"
9

Main> run expr "8/4/2"   -- '/' is left associative
1

Main> run expr "8/(4/2)"
```

```
4

Main> run expr "1 + 2"  -- wrong!
1

Main> run expr "1+ 2"   -- wrong!
parse error at (line 1, column 3):
unexpected " "
expecting simple expression
```

2.8 Lexical analysis

The previous hugs session with expressions shows that the expression parser can not yet deal with whitespace. Most parsers take their input from a lexical analyzer or scanner which filters white space and comments and returns a list tokens for the parser. How Parsec can work on such token streams is described in a later section. Parsec can also merge both passes into one parser specification which allows us to use all the normal parser combinators to deal with lexical analysis too. The standard module `ParsecToken` deals with issues normally delegated to a seperate scanner: white space, comments, identifiers, reserved words, numbers, strings and other tokens.

Unfortunately, most languages disagree on the syntax of comments, the form of identifiers and for example case sensitivity. The `ParsecToken` module should be parameterized with those differences, but unfortunately, Haskell doesn't support parameterized modules. The module therefore exports a single function `makeTokenParser` that takes paremeterizable language features as an argument (`LanguageDef`) and returns a (rather large) record that contains a set of lexical parsers. For your convenience, the module `ParsecLanguage` contains a set of common language definitions. For the expression parser, we will use haskell-style comments and identifiers

```
module Main where
import Parsec
import qualified ParsecToken as P
import ParsecLanguage( haskellStyle )

lexer :: TokenParser ()
lexer  = makeTokenParser
         (haskellDef
         { reservedOpNames = ["*","/","+","-"]
         }
```

For efficiency, we will bind all the used lexical parsers at toplevel.

```
whiteSpace= P.whiteSpace lexer
lexeme    = P.lexeme lexer
symbol    = P.symbol lexer
natural   = P.natural lexer
parens    = P.parens lexer
semi      = P.semi lexer
```

```
identifier= P.identifier lexer
reserved  = P.reserved lexer
reservedOp= P.reservedOp lexer
```

Every lexical parser from the ParsecToken module will skip whitespace after each symbol parsed; parsers which skip trailing whitespace are called *lexeme* parsers (the lexeme combinator can be used to define them). By skipping trailing whitespace, it is garanteed that every parser starts at valid input. For the expression parser, the lexeme parser reservedOp is used instead of the char and string parsers:

```
import ParsecToken

...

expr    :: Parser Integer
expr    = buildExpressionParser table factor
        <?> "expression"

table   = [[op "*" (*) AssocLeft, op "/" div AssocLeft]
          ,[op "+" (+) AssocLeft, op "-" (-) AssocLeft]
          ]
        where
          op s f assoc
             = Infix (do{ reservedOp s; return f} <?> "operator") assoc
```

The ParsecToken module also defines a set of high-level combinators like parens to parse something enclosed in parenthesis and natural to parse natural numbers in decimal, octal or hexadecimal notation:

```
factor  =   parens expr
        <|> natural
        <?> "simple expression"
```

Finally the run function will be adapted to deal with whitespace. The main parser needs to skip any whitespace at the start of the input (since all other parsers only skip trailing whitespace) and needs to end with the eof parser which only succeeds at the end of the input. This parser garantees that all input will be parsed; if it is left out, a parser is allowed to parse only part of the input.

```
runLex :: Show a => Parser a -> String -> IO ()
runLex p input
        = run (do{ whiteSpace
                 ; x <- p
                 ; eof
                 ; return x
                 }) input
```

And that's it. In about 30 lines of code we have a fully functional, extensible expression parser with comments, priority and associativity rules and infinite precision numbers in hexadecimal, octal and decimal representation.

```
Main> runLex expr "1 + 2"
3

Main> runLex expr "1 + {- comment -} 2 * 3 --multiply has higher priority"
7

Main> runLex expr "  0xAA / 0o37 / 2"
2

Main> runLex expr "0xAA / 0o37 2 "
parse error at (line 1, column 13):
unexpected "2"
expecting operator or end of input
```

2.9 Receipts: Lexeme parsers and Reserved words

Consider the following EBNF grammar for receipts:

```
receipt     ::= product* total

produkt     ::= "return" price ";"
              | identifier price ";"
total       ::= price "total"

price       ::= digit+ "." digit digit
```

A receipt is a sequence of semicolon terminated produkts and ends with the total price of all products. A produkt is either a name of a produkt with its price or a return produkt in which case the client receives the specified price. We will construct a parser for receipts which checks if the total amount is actually correct.

2.9.1 Lexeme parsers

The `price` parser is a nice exercise in specifying lexeme parsers. The price parser doesn't allow whitespace inside; For example, we can't use the `decimal` parser to parse the digits in front of the dot since the `decimal` parser would allow whitespace between those digits and the dot. The appropiate solution is to create a lexeme parser ourselves.

```
price   :: Parser Int   -- price in cents
price   = lexeme (do{ ds1 <- many1 digit
                    ; char '.'
                    ; ds2 <- count 2 digit
                    ; return (convert 0 (ds1 ++ ds2))
                    })
          <?> "price"
          where
            convert n []     = n
            convert n (d:ds) = convert (10*n + digitToInt d) ds
```

The implementation of the other productions is almost a literal translation of the grammar rules:

```
receipt :: Parser Bool
receipt = do{ ps <- many produkt
            ; p  <- total
            ; return (sum ps == p)
            }

produkt = do{ symbol "return"
            ; p <- price
            ; semi
            ; return (-p)
            }
      <|> do{ identifier
            ; p  <- price
            ; semi
            ; return p
            }
      <?> "produkt"

total   = do{ p <- price
            ; symbol "total"
            ; return p
            }
```

Let's try the parser on a few receipts:

```
Main> runLex receipt "book 12.00; plant 2.55; 14.55 total"
True

Main> runLex receipt "book 12.00; plant 2.55; 12.55 total"
False

Main> runLex receipt "book 12.00; plant 2; 12.55 total"
parse error at (line 1, column 20):
unexpected ";"
expecting digit or "."

Main> runLex receipt "book 12.00; return 2.00; plant 2.55; 12.55 total"
True
```

2.9.2 Reserved words

Unfortunately, the parser will incorrectly return an error on the following input:

```
Main> runLex receipt "book 12.00; reader 2.00; plant 1.00; 15.00 total"
parse error at (line 1, column 13):
unexpected "a"
expecting "return"
```

What happens here? The `produkt` parser will try to recognize the keyword `return` when it starts scanning `"reader"`. When the `"a"` is encountered an error is returned. The parser will not try the second alternative which starts with an identifier since the it has already consumed input (i.e. `"re"`).

This is a good example of a grammar which is hard to left-factor. The use of the `try` combinator is the appropiate way of solving this problem:

```
produkt = do{ try (symbol "return")
            ; p <- price
            ; semi
            ; return (-p)
            }
        <|> do{ identifier
            ; p  <- price
            ; semi
            ; return p
            }
        <?> "produkt"
```

The test run now gives the expected result:

```
Main> runLex receipt "book 12.00; reader 2.00; plant 1.00; 15.00 total"
True
```

However, we are still not done as the following example shows:

```
Main> runLex receipt "book 12.00; returns 2.00; plant 1.00; 15.00 total"
parse error at (line 1, column 19):
unexpected "s"
expecting price
```

The current parser recognises the keyword `return` now and it complains about the letter `'s'` following it. A keyword however shouldn't be recognised as a keyword when it is only a prefix of an identifier. The parser `notFollowedBy` can be used to make sure that a keyword is not followed by a legal identifier character. Just like checking that a keyword is not a prefix of an identifier, we should also check that an identifier doesn't make up a keyword, in order to make the parser independent of the order of choices.

Since adding the `try` combinator in the right places, using `notFollowedBy` and checking for reserved words is quite subtle, the `ParsecToken` module implements all of this complexity for you; simply add the reserved words to your token definition:

```
lexer :: TokenParser ()
lexer  = makeTokenParser
         (haskellDef
         { reservedNames   = ["return","total"]
         , reservedOpNames = ["*","/","+","-"]
         }
```

Instead of `symbol`, the combinator `reserved` is used to scan reserved words. The `try` combinator is not necessary anymore since the `reserved` combinator already takes care of it.

```
receipt :: Parser Bool
receipt = do{ ps <- many produkt
            ; p  <- total
            ; return (sum ps == p)
            }

produkt = do{ reserved "return"
            ; p <- price
            ; semi
            ; return (-p)
            }
      <|> do{ identifier
            ; p  <- price
            ; semi
            ; return p
            }
      <?> "produkt"

total   = do{ p <- price
            ; reserved "total"
            ; return p
            }
```

Finally, everything works as expected:

```
Main> runLex receipt "book 12.00; returns 2.00; plant 1.00; 15.00 total"
True

Main> runLex receipt "book 12.00; total 2.00; plant 1.00; 15.00 total"
parse error at (line 1, column 13):
unexpected reserved word "total"
expecting produkt or price

Main> runLex receipt "book 12.00; totals 2.00; return 1.00; 13.00 total"
True
```

2.10 Examples

The following example parsers are shipped with Parsec:

- **Tiger** and **While**. Both parsers are written by Jurriaan Hage, who used the languages during CS cources at Utrecht University. The *Tiger* language is the example language in the compiler book by Andrew Appel (1997). The *While* language is the example language in the semantics book by Nielson, Nielson and Hankin (1999).

- **Henk 2000**, based on Pure Type Systems. Written by Jan-Willem Roorda and Daan Leijen. A parser for the Henk language as descibed by Peyton Jones and Meijer (1997). The master thesis of Jan-Willem is online available together with a full interpreter for pure-type-systems.
- **Mondrian**, the internet programming language (Mondrian, 2000). An example of a Java-style language.

2.11 Advanced: Seperate scanners

In the previous examples we used an integrated approach – the lexer was specified using normal parser combinators. This is not always desirable, sometimes there is already a scanner available or sometimes the input needs pre-processing. Another advantage of writing a scanner by hand is that the performance of the parser generally improves about 30% (see the benchmark for test results).

Suppose that we have written a seperate scanner for receipts. The `scanner` function returns a list of tokens and a list of error messages.

```
type Token  = (SourcePos,Tok)
data Tok    = Identifier String
            | Reserved   String
            | Symbol     String
            | Price      Int
            deriving Show

scanner :: [Char] -> ([Token],[String])
```

The parsers should now work on these token streams instead of the normal chararacter streams. This is reflected in the type. The type of *general* parsers is `GenParser tok st a`, where `tok` is the type of the tokens, `st` the type of the local user state and `a` is the type of the result. Indeed, `Parser` is just a type synonym for parsers that work on characters and have no state:

```
type Parser a   = GenParser Char () a
```

The `token` parser is used to define a parser that scans a single token. The function takes three arguments: a function to show tokens, a function to extract the source position of a token and finally a test function that determines if the token is accepted or not. The first two arguments are the same for all our tokens so we define a little abstraction:

```
type MyParser a   = GenParser Token () a

mytoken :: (Tok -> Maybe a) -> MyParser a
mytoken test
  = token showToken posToken testToken
  where
    showToken (pos,tok)   = show tok
    posToken  (pos,tok)   = pos
    testToken (pos,tok)   = test tok
```

Now, it is easy to define the basic parser that work on tokens.

```
identifier :: MyParser String
identifier
  = mytoken (\tok -> case tok of
                       Identifier name -> Just name
                       other           -> Nothing)

reserved :: String -> MyParser ()
reserved name
  = mytoken (\tok -> case tok of
                       Reserved s   | s == name  -> Just ()
                       other        -> Nothing)

...
```

For an extended example, you might want to look at the implementation of the `ParsecChar` module.

2.12 Advanced: User state

Since Parsec is already a state monad, it is easy to include a user definable state. This state is very useful when building a symbol table – strings can immediately be converted to identifiers [1].

Parsec provides three combinators to handle state. The function `getState` gets the state, `setState st` sets the state to `st`, and `updateState f` updates the state by applying function `f` to it. As an example, we show how we can count the number of identifiers in a parser.

```
run input
  = case runParser parser 0 "" input of
      Right n  -> putStrLn ("counted " ++ show n ++ " identifiers!")
      Left err -> do{ putStr "parse error at "
                    ; print err
                    }

parser :: CharParser Int Int
parser
  = do{ ...
      ; n <- getState
      ; return n
      }
```

[1]I have to warn the reader though that experience with the HaskellLight compiler has shown that it hardly pays off in practice to use special identifier representations instead of normal strings

```
...

myIdentifier :: CharParser Int String
myIdentifier
  = do{ x <- identifier
      ; updateState (+1)
      ; return x
      }
```

2.13 Advanced: Permutation phrases

A permutation phrase is sequence of elements of possibly different types in which each element occurs at most once and the order is irrelevant. Parsec provides the module `ParsecPerm` to parse such free-order constructs. Possible applications include parsers for XML attributes and Haskell style records.

Since each element can be of a different type, it is hard to find a typable approach and we have to resort to judicious use of permutation parser combinators. Parsers are combined into a permutation parser with the `(<||>)` operator. At the end, all the elements are combined using a combination function `f` that is applied using the `(<$$>)` combinator. The whole permutation parser is than converted into a normal parser using the `permute` function. Suppose we want to parse a permutation of the characters `a`, `b`, and `c`. We can write this as:

```
perm0 = permute (f (<$$>) char 'a'
                   (<||>) char 'b'
                   (<||>) char 'c')
      where
        f a b c  = [a,b,c]
```

Let's try the parser against some real input!

```
Main> run perm0 "abc"
"abc"

Main> run perm0 "cba"
"abc"

Main> run perm0 "b"
parse error at (line 1, column 2):
unexpected end of input
expecting "c" or "a"
```

There is an implicit restriction that the parsers in a permutation should never succeed on empty input – ie. you can't use `(many (char 'a'))` to parse an optional string of `'a'` characters. If there are optional elements in the permutation, you need to use the special `(<|?>)` operator that takes both a parser and a default value. If the first parser can be optional, you can use the `(<$?>)` operator. For example, suppose we parse a permutation of: an optional string of `a`'s, the character `b` and an optional `c`. This can be described by:

```
perm1 :: Parser (String,Char,Char)
perm1 = permute (tuple <$?> ("",many1 (char 'a'))
                       <||> char 'b'
                       <|?> ('_',char 'c'))
      where
        tuple a b c  = (a,b,c)
```

At the command prompt we can see how error correction works with permutations.

```
Main> run perm1 "caaaaab"
("aaaaa",'b','c')

Main> run perm1 "cb"
("",'b','c')

Main> run perm1 "b"
("",'b','_')

Main> run perm1 ""
parse error at (line 1, column 1):
unexpected end of input
expecting "c", "b" or "a"

Main> run perm1 "c"
parse error at (line 1, column 2):
unexpected end of input
expecting "b" or "a"

Main> run perm1 "ca"
parse error at (line 1, column 3):
unexpected end of input
expecting "a" or "b"
```

3 Reference guide

Parsec consists of the following main modules:

- `Parsec` implements the core parser library.
- `ParsecExpr` is an extension library for parsing expression grammars.
- `ParsecToken` and `ParsecLanguage` are extension libraries for parsing lexical tokens.
- `ParsecPerm` is an extension library for parsing permutation phrases.

3.1 Module `Parsec`

This module is the core parser library. It exports functions from following primitive modules:

- `ParsecPos`. Source positions.
- `ParsecError`. Error messages.
- `ParsecPrim`. The main parser type and primitive parser combinators, for example `parse`.
- `ParsecCombinator`. General polymorphic parser combinators, for example `many`.
- `ParsecChar`. A range of parsers for characters, for example `digit`.

3.2 Module `ParsecPrim`

Parser a

> A type synonym for `GenParser Char () a`, i.e. a parser for character streams without a user state.

GenParser tok st a

> The General Parser `GenParser tok st a` data type represents a parser that parses tokens of type `tok` with a user supplied state `st` and returns a value of type `a` on success. `GenParser` is an instance of the `Functor`, `Monad` and `MonadPlus` classes.

runParser :: `GenParser tok st a -> st -> FilePath -> [tok] -> Either ParseError a`

> The most general way to run a parser. `(runParser p state filePath input)` runs parser `p` on the input list of tokens `input`, obtained from source `filePath` with the initial user state `st`. The `filePath` is only used in error messages and may be the empty string. Returns either a `ParseError` (`Left`) or a value of type `a` (`Right`).

```
parseFromFile p fname
  = do{ input <- readFile fname
      ; return (runParser p () fname input)
      }
```

parse :: `Parser a -> FilePath -> String -> Either ParseError a`

`(parse p filePath input)` runs a character parser `p` without user state. The `filePath` is only used in error messages and may be the empty string. Returns either a `ParseError` (`Left`) or a value of type `a` (`Right`).

```
main    = case (parse numbers "" "11, 2, 43") of
            Left err  -> print err
            Right xs  -> print (sum xs)

numbers = commaSep integer
```

parseFromFile :: `Parser a -> FilePath -> IO (Either ParseError a)`

`(parseFromFile p filePath)` runs a character parser `p` on the input read from `filePath`. Returns either a `ParseError` (`Left`) or a value of type `a` (`Right`).

```
main    = do{ result <- parseFromFile numbers "digits.txt"
            ; case (result) of
                Left err  -> print err
                Right xs  -> print (sum xs)
            }
```

parseTest :: `Show a => GenParser tok () a -> [tok] -> IO ()`

The expression `parseTest p input` applies a parser `p` against input `input` and prints the result to stdout. Used for testing parsers.

```
parseTest p input
        = case (runParser p () "" input) of
            Left err -> do{ putStr "parse error at "
                          ; print err
                          }
            Right x  -> print x
```

return :: `a -> GenParser tok st a`

The parser `(return x)` always succeeds with value `x` without consuming any input.

`(>>=) :: GenParser tok st a -> (a -> GenParser tok st b) -> GenParser tok st b`
`infixl 1`

This parser is called *bind* and implements sequencing. The parser `(p >>= f)` first applies parser `p`. It than applies `f` to the returned value of `p` and applies the resulting parser.

The `do`-notation of Haskell is syntactic sugar that automatically uses this combinator. Using this notation, we can write `do{ x <- p; q }` instead of the more cumbersome `(p >>= (\x -> q))`. The `do`-notation has a natural operational reading: first apply parser `p`, binding its result to value `x`, and than apply the parser `q`.

Since the second parser can depend on the result of the first parser, we are able to parse *context sensitive* grammars.

Examples of context sensitive parsing are XML tags, variable definition before use and environments in TEX. A simple example of parsing TEX environments is:

```
environment  = do{ name <- envBegin
                 ; environment
                 ; envEnd name
                 }
             <|> return ()

envBegin :: Parser String
envBegin     = do{ reserved "\\begin"
                 ; braces (many1 letter)
                 }

envEnd :: String -> Parser ()
envEnd name  = do{ reserved "\\end"
                 ; braces (string name)
                 }
```

`(<|>) :: GenParser tok st a -> GenParser tok st a -> GenParser tok st a`
`infixr 1`

This combinator implements choice. The parser `(p <|> q)` first applies `p`. If it succeeds, the value of `p` is returned. If `p` fails *without consuming any input*, parser `q` is tried. This combinator is defined equal to the `mplus` member of the `MonadPlus` class.

The parser is called *predictive* since `q` is only tried when parser `p` didn't consume any input (i.e.. the look ahead is 1). This non-backtracking behaviour allows for both an efficient implementation of the parser combinators and the generation of good error messages.

fail :: `String -> GenParser tok st a`

The parser `(fail msg)` always fails with message-error `msg` without consuming any input.

The `return`, `(>>=)` and `fail` combinators are the members of the `Monad` class. The `Parser` is called *monadic* since it both provides an implementation for these three combinators, making it an instance of the `Monad` class, and it satisfies the monad-laws.

pzero :: `GenParser tok st a`

`pzero` always fails without consuming any input. `pzero` is defined equal to the `mzero` member of the `MonadPlus` class.

try :: `GenParser tok st a -> GenParser tok st a`

The parser `(try p)` behaves like parser `p`, except that it pretends that it hasn't consumed any input when an error occurs.

This combinator is used whenever arbitrary look ahead is needed. Since it pretends that it hasn't consumed any input when `p` fails, the `(<|>)` combinator will try its second alternative even when the first parser failed while consuming input.

The `try` combinator can for example be used to distinguish identifiers and reserved words. Both reserved words and identifiers are a sequence of letters. Whenever we expect a certain reserved word where we can also expect an identifier we have to use the `try` combinator. Suppose we write:

```
expr        = letExpr <|> identifier <?> "expression"

letExpr     = do{ string "let"; ... }
identifier  = many1 letter
```

If the user writes `"lexical"`, the parser fails with: `unexpected 'x', expecting 't' in "let"`. Indeed, since the `(<|>)` combinator only tries alternatives when the first alternative hasn't consumed input, the `identifier` parser is never tried (because the prefix `"le"` of the `(string "let")` parser is already consumed). The right behaviour can be obtained by adding the `try` combinator:

```
expr        = letExpr <|> identifier <?> "expression"

letExpr     = do{ try (string "let"); ... }
identifier  = many1 letter
```

Since the use of the `try` combinator with lexical tokens is quite tricky, the ParsecToken module can be used to parse lexical tokens. This module automatically uses the try combinator in the appropiate places.

token `:: (tok -> String) -> (tok -> SourcePos) -> (tok -> Maybe a) -> GenParser tok st a`

The parser `token showTok posFromTok testTok` accepts a token `t` with result `x` when the function `testTok t` returns `Just x`. The source position of the `t` should be returned by `posFromTok t` and the token can be shown using `showTok t`.

This combinator is expressed in terms of `tokenPrim`. It is used to accept user defined token streams. For example, suppose that we have a stream of basic tokens tupled with source positions. We can than define a parser that accepts single tokens as:

```
mytoken :: Show t => t -> GenParser ((Int,Int),t) () t
mytoken x
  = token showTok posFromTok testTok
  where
    showTok (pos,t)     = show t
    posFromTok (pos,t)  = pos
    testTok (pos,t)     = if (x == t) then Just t else Nothing
```

tokenPrim `:: (tok -> String) -> (SourcePos -> tok -> [tok] -> SourcePos) -> (tok -> Maybe a) -> GenParser tok st a`

The parser `(token showTok nextPos testTok)` accepts a token `t` with result `x` when the function `testTok t` returns `Just x`. The token can be shown using `showTok t`. The position of the *next* token should be returned when `nextPos` is called with the current source position `pos`, the current token `t` and the rest of the tokens `toks`, `(nextPos pos t toks)`.

This is the most primitive combinator for accepting tokens. For example, the `char` parser could be implemented as:

```
char :: Char -> GenParser Char st Char
char c
  = tokenPrim showChar nextPos testChar
  where
    showChar x       = "'" ++ x ++ "'"
    testChar x        = if (x == c) then Just x else Nothing
    nextPos pos x xs  = updatePosChar pos x
```

`(<?>)` :: `GenParser tok st a -> String -> GenParser tok st a`
`infix 0`

The parser `p <?> msg` behaves as parser `p`, but whenever the parser `p` fails *without consuming any input*, it replaces expect error messages with the expect error message `msg`.

This is normally used at the end of a set alternatives where we want to return an error message in terms of a higher level construct rather than returning all possible characters. For example, if the `expr` parser from the `try` example would fail, the error message is: `...: expecting expression`. Without the `(<?>)` combinator, the message would be like `...: expecting "let" or letter`, which is less friendly.

unexpected :: `String -> GenParser tok st a`

The parser `(unexpected msg)` always fails with an unexpected error message `msg` without consuming any input.

The parsers `fail`, `(<?>)` and `unexpected` are the three parsers used to generate error messages. Of these, only `(<?>)` is commonly used. For an example of the use of `unexpected`, see the definition of `notFollowedBy`.

getState :: `GenParser tok st st`

Returns the current user state.

setState :: `st -> GenParser tok st ()`

`(setState st)` set the user state to `st`.

updateState :: `(st -> st) -> GenParser tok st ()`

`(updateState f)` applies function `f` to the user state. Suppose that we want to count identifiers in a source, we could use the user state as:

```
expr :: GenParser Char Int Expr
expr  = do{ x <- identifier
          ; updateState (+1)
          ; return (Id x)
          }
```

getPosition :: `GenParser tok st SourcePos`

Returns the current source position. See also `SourcePos`

setPosition :: `SourcePos -> GenParser tok st ()`

`setPosition pos` sets the current source position to `pos`.

getInput :: `GenParser tok st [tok]`

Returns the current input

setInput :: `[tok] -> GenParser tok st ()`

`setInput input` continues parsing with `input`. The `getInput` and `setInput` functions can for example be used to deal with `#include` files.

3.3 Module `ParsecCombinator`

many :: `GenParser tok st a -> GenParser tok st [a]`

`(many p)` applies the parser `p` *zero* or more times. Returns a list of the returned values of `p`.

```
identifier  = do{ c  <- letter
                ; cs <- many (alphaNum <|> char '_')
                ; return (c:cs)
                }
```

many1 :: `GenParser tok st a -> GenParser tok st [a]`

`(many p)` applies the parser `p` *one* or more times. Returns a list of the returned values of `p`.

```
word  = many1 (letter)
```

skipMany :: `GenParser tok st a -> GenParser tok st ()`

`(skipMany p)` applies the parser `p` *zero* or more times, skipping its result.

```
spaces  = skipMany space
```

skipMany1 :: `GenParser tok st a -> GenParser tok st ()`

`(skipMany1 p)` applies the parser `p` *one* or more times, skipping its result.

sepBy :: `GenParser tok st a -> GenParser tok st sep -> GenParser tok st [a]`

`(sepBy p sep)` parses *zero* or more occurrences of `p`, separated by `sep`. Returns a list of values returned by `p`.

```
commaSep p  = p `sepBy` (symbol ",")
```

sepBy1 :: `GenParser tok st a -> GenParser tok st sep -> GenParser tok st [a]` (`sepBy1 p sep`) parses *one* or more occurrences of `p`, separated by `sep`. Returns a list of values returned by `p`.

endBy :: `GenParser tok st a -> GenParser tok st end -> GenParser tok st [a]`

(`endBy p sep`) parses *zero* or more occurrences of `p`, seperated and ended by `sep`. Returns a list of values returned by `p`.

```
cStatements  = cStatement `endBy` semi
```

endBy1 :: `GenParser tok st a -> GenParser tok st end -> GenParser tok st [a]`

(`endBy1 p sep`) parses *one* or more occurrences of `p`, seperated and ended by `sep`. Returns a list of values returned by `p`.

sepEndBy :: `GenParser tok st a -> GenParser tok st sep -> GenParser tok st [a]`

(`sepEndBy p sep`) parses *zero* or more occurrences of `p`, separated and optionally ended by `sep`, ie. haskell style statements. Returns a list of values returned by `p`.

```
haskellStatements  = haskellStatement `sepEndBy` semi
```

sepEndBy1 :: `GenParser tok st a -> GenParser tok st sep -> GenParser tok st [a]`

(`sepEndBy1 p sep`) parses *one* or more occurrences of `p`, separated and optionally ended by `sep`. Returns a list of values returned by `p`.

count :: `Int -> GenParser tok st a -> GenParser tok st [a]`

(`count n p`) parses `n` occurrences of `p`. If `n` is smaller or equal to zero, the parser equals to `(return [])`. Returns a list of `n` values returned by `p`.

between :: `GenParser tok st open -> GenParser tok st close -> GenParser tok st a -> GenParser tok st a`

(`between open close p`) parses `open`, followed by `p` and `close`. Returns the value returned by `p`.

```
braces  = between (symbol "{") (symbol "}")
```

option :: `a -> GenParser tok st a -> GenParser tok st a`

(`option x p`) tries to apply parser `p`. If `p` fails without consuming input, it returns the value `x`, otherwise the value returned by `p`.

```
priority :: Parser Int
priority  = option 0 (do{ d <- digit
                        ; return (digitToInt d)
                        })
```

choice :: `[GenParser tok st a] -> GenParser tok st a` (`choice ps`) tries to apply the parsers in the list `ps` in order, until one of them succeeds. Returns the value of the succeeding parser. `choice` can be defined as:

```
choice ps  = foldl (<|>) pzero ps
```

manyTill :: `GenParser tok st a -> GenParser tok st end -> GenParser tok st [a]`

(`manyTill p end`) applies parser `p` *zero* or more times until parser `end` succeeds. Returns the list of values returned by `p` . This parser can be used to scan comments:

```
simpleComment   = do{ string "<!--"
                    ; manyTill anyChar (try (string "-->"))
                    }
```

Note the overlapping parsers `anyChar` and `string "<!--"`, and therefore the use of the `try` combinator.

chainl :: `GenParser tok st a -> GenParser tok st (a->a->a) -> a -> GenParser tok st a`

(`chainl p op x`) parser *zero* or more occurrences of `p`, separated by `op`. Returns a value obtained by a *left* associative application of all functions returned by `op` to the values returned by `p`. If there are zero occurrences of `p`, the value `x` is returned. .

chainl1 :: `GenParser tok st a -> GenParser tok st (a->a->a) -> GenParser tok st a`

(`chainl1 p op x`) parser *one* or more occurrences of `p`, separated by `op` Returns a value obtained by a *left* associative application of all functions returned by `op` to the values returned by `p`. . This parser can for example be used to eliminate left recursion which typically occurs in expression grammars.

```
expr    = term   `chainl1` mulop
term    = factor `chainl1` addop
factor  = parens expr <|> integer

mulop   =   do{ symbol "*"; return (*)   }
        <|> do{ symbol "/"; return (div) }

addop   =   do{ symbol "+"; return (+) }
        <|> do{ symbol "-"; return (-) }
```

chainr :: `GenParser tok st a -> GenParser tok st (a->a->a) -> a -> GenParser tok st a`

(`chainr p op x`) parser *zero* or more occurrences of `p`, separated by `op` Returns a value obtained by a *right* associative application of all functions returned by `op` to the values returned by `p`. If there are no occurrences of `p`, the value `x` is returned. .

chainr1 :: `GenParser tok st a -> GenParser tok st (a->a->a) -> GenParser tok st a` `(chainr1 p op x)` parser *one* or more occurrences of `p`, separated by `op` Returns a value obtained by a *right* associative application of all functions returned by `op` to the values returned by `p`. .

eof :: `Show tok => GenParser tok st ()`

This parser only succeeds at the end of the input. This is not a primitive parser but it is defined using `notFollowedBy`.

```
eof  = notFollowedBy anyToken <?> "end of input"
```

Wow, I really like this definition!

notFollowedBy :: `Show tok => GenParser tok st tok -> GenParser tok st ()`

`(notFollowedBy p)` only succeeds when parser `p` fails. This parser does not consume any input. This parser can be used to implement the 'longest match' rule. For example, when recognizing keywords (for example `let`), we want to make sure that a keyword is not followed by a legal identifier character, in which case the keyword is actually an identifier (for example `lets`). We can program this behaviour as follows:

```
keywordLet  = try (do{ string "let"
                     ; notFollowedBy alphaNum
                     }
                   )
```

Surprisingly, this parser is not primitive and can be defined as:

```
notFollowedBy p  = try (do{ c <- p; unexpected (show [c]) }
                       <|> return ()
                       )
```

anyToken :: `Show tok => GenParser tok st tok`

The parser `anyToken` accepts any kind of token. It is for example used to implement `eof`. Returns the accepted token.

3.4 Module `ParsecChar`

CharParser st a

A type synonym for `GenParser Char st a`, i.e. a parser for character streams with user state `st`.

oneOf :: `[Char] -> CharParser st Char`

`(oneOf cs)` succeeds if the current character is in the supplied list of characters `cs`. Returns the parsed character. See also `satisfy`.

```
vowel  = oneOf "aeiou"
```

noneOf :: `[Char] -> CharParser st Char`

As the dual of `oneOf`, `(noneOf cs)` succeeds if the current character *not* in the supplied list of characters `cs`. Returns the parsed character.

```
consonant = noneOf "aeiou"
```

char :: `Char -> CharParser st Char`

`(char c)` parses a single character `c`. Returns the parsed character (i.e. `c`).

```
semiColon  = char ';'
```

string :: `String -> CharParser st String`

`(string s)` parses a sequence of characters given by `s`. Returns the parsed string (i.e. `s`).

```
divOrMod    =   string "div"
            <|> string "mod"
```

anyChar :: `CharParser st Char`

This parser succeeds for any character. Returns the parsed character.

upper :: `CharParser st Char`

Parses an upper case letter (a character between `'A'` and `'Z'`). Returns the parsed character.

lower :: `CharParser st Char`

Parses a lower case character (a character between `'a'` and `'z'`). Returns the parsed character.

letter :: `CharParser st Char`

Parses a letter (an upper case or lower case character). Returns the parsed character.

alphaNum :: `CharParser st Char`

Parses a letter or digit (a character between `'0'` and `'9'`). Returns the parsed character.

digit :: `CharParser st Char`

Parses a digit. Returns the parsed character.

hexDigit :: `CharParser st Char`

Parses a hexadecimal digit (a digit or a letter between `'a'` and `'f'` or `'A'` and `'F'`). Returns the parsed character.

octDigit :: `CharParser st Char`

Parses an octal digit (a character between `'0'` and `'7'`). Returns the parsed character.

newline :: `CharParser st Char`

Parses a newline character (`'\n'`). Returns a newline character.

tab :: `CharParser st Char`

Parses a tab character (`'\t'`). Returns a tab character.

space :: `CharParser st Char`

Parses a white space character (any character in `" \v\f\t\r\n"`). Returns the parsed character.

spaces :: `CharParser st ()`

Skips *zero* or more white space characters. See also `skipMany`.

satisfy :: `(Char -> Bool) -> CharParser st Char`

The parser `(satisfy f)` succeeds for any character for which the supplied function `f` returns `True`. Returns the character that is actually parsed.

```
digit     = satisfy isDigit
oneOf cs  = satisfy (\c -> c ‘elem‘ cs)
```

3.5 Module ParsecPos

SourcePos

The abstract data type `SourcePos` represents source positions. It contains the name of the source (i.e. file name), a line number and a column number. `SourcePos` is an instance of the `Show`, `Eq` and `Ord` class.

Line

A type synonym for `Int`

Column

A type synonym for `Int`

sourceName :: `SourcePos -> FilePath`

Extracts the name of the source from a source position.

sourceLine :: `SourcePos -> Line`

Extracts the line number from a source position.

sourceColumn :: `SourceColumn -> Column`

Extracts the column number from a source position.

incSourceLine :: `SourcePos -> Int -> SourcePos`

Increments the line number of a source position.

incSourceColumn :: `SourcePos -> Int -> SourcePos`

Increments the column number of a source position.

setSourceLine :: `SourcePos -> SourceLine -> SourcePos`

Set the line number of a source position.

setSourceColumn :: `SourcePos -> SourceColumn -> SourcePos`

Set the column number of a source position.

setSourceName :: `SourcePos -> SourceName -> SourcePos`

Increments the line number of a source position.

updatePosChar :: `SourcePos -> Char -> SourcePos`

Update a source position given a character. If the character is a newline (`'\n'`) or carriage return (`'\r'`) the line number is incremented by 1. If the character is a tab (`'\t'`) the column number is incremented to the nearest 8'th column, ie. `(column + 8 - ((column-1) `mod` 8))`. In all other cases, the column is incremented by 1.

updatePosString :: `SourcePos -> String -> SourcePos`

The expression `(updatePosString pos s)` updates the source position `pos` by calling `updatePosChar` on every character in `s`, ie. `(foldl updatePosChar pos string)`.

3.6 Module ParsecError

ParseError

The abstract data type `ParseError` represents parse errors. It provides the source position (`SourcePos`) of the error and a list of error messages (`Message`). A `ParseError` can be returned by the function `parse`. `ParseError` is an instance of the `Show` class.

errorPos :: `ParseError -> SourcePos`

Extracts the source position from the parse error

errorMessages :: `ParseError -> [Message]`

Extracts the list of error messages from the parse error

Message

This abstract data type represents parse error messages. There are four kinds of messages:

```
data Message = SysUnExpect String
             | UnExpect String
             | Expect String
             | Message String
```

The fine distinction between different kinds of parse errors allows the system to generate quite good error messages for the user. It also allows error messages that are formatted in different languages. Each kind of message is generated by different combinators:

A `SysUnExpect` message is automatically generated by the `satisfy` combinator. The argument is the unexpected input.

- A `UnExpect` message is generated by the `unexpected` combinator. The argument describes the unexpected item.
- A `Expect` message is generated by the `<?>` combinator. The argument describes the expected item.
- A `Message` message is generated by the `fail` combinator. The argument is some general parser message.

messageString :: `Message -> String`

Extract the message string from an error message

messageCompare :: `Message -> Message -> Ordering`

Compares two error messages without looking at their content. Only the constructors are compared where:

`SysUnExpect < UnExpect < Expect < Message`

messageEq :: `Message -> Message -> Bool`

`(messageEq m1 m2)` equals `True` if `(messageCompare m1 m2)` equals `EQ`, in all other cases it equals `False`

showErrorMessages :: `[Message] -> String`

The standard function for showing error messages. Formats a list of error messages in English. This function is used in the `Show` instance of `ParseError`. The resulting string will be formatted like:

```
unexpected {The first UnExpect or a SysUnExpect message};
expecting {comma separated list of Expect messages};
{comma separated list of Message messages}
```

3.7 Module `ParsecToken`

The extension module `ParsecToken` is used to parse lexical tokens (for example: identifiers, comments and numbers). Since programming languages all have slightly different grammar rules for lexical tokens, the module should be parameterized with the most common differences between languages (for example: valid identifier characters, reserved words etc). Since Haskell doesn't have parametrized modules, we resort to a trick. The module exports a single function (`makeTokenParser`) that returns a rather large record that contains all the lexical parse functions.

Unfortunately, some of those functions (like `parens`) are polymorphic which implies that we need to use the `forall` keyword to give their type signature inside the record. This keyword is *not* part of the Haskell98 standard but almost all Haskell systems support it nowadays. You might need to add some flags to your compiler options to make this work, for example, GHC needs `-fglasgow-exts` and Hugs the `-98` option.

All lexical features that are not parameterizable are implemented according to the Haskell lexical rules (for example: escape codes and floating point numbers). Luckily most language grammars match the Haskell grammar on these tokens quite closely (if not completely).

TokenParser st

The type of the record that holds lexical parsers that work on character streams with state `st`. Each function of this record is seperately described as if it was a toplevel function in a later section.

```
data TokenParser st
 = TokenParser{
     identifier  :: CharParser st String
   , reserved    :: String -> CharParser st ()
   , operator    :: CharParser st String
   , reservedOp  :: String -> CharParser st ()

   , charLiteral :: CharParser st Char
   , stringLiteral :: CharParser st String
   , natural     :: CharParser st Integer
   , integer     :: CharParser st Integer
   , float       :: CharParser st Double
   , naturalOrFloat:: CharParser st (Either Integer Double)
   , decimal     :: CharParser st Integer
   , hexadecimal :: CharParser st Integer
   , octal       :: CharParser st Integer

   , symbol      :: String -> CharParser st String
   , lexeme      :: forall a. CharParser st a -> CharParser st a
   , whiteSpace  :: CharParser st ()

   , parens      :: forall a. CharParser st a -> CharParser st a
   , braces      :: forall a. CharParser st a -> CharParser st a
   , brackets    :: forall a. CharParser st a -> CharParser st a
   , squares     :: forall a. CharParser st a -> CharParser st a

   , semi        :: CharParser st String
   , comma       :: CharParser st String
   , colon       :: CharParser st String
   , dot         :: CharParser st String
   , semiSep     :: forall a . CharParser st a -> CharParser st [a]
   , semiSep1    :: forall a . CharParser st a -> CharParser st [a]
   , commaSep    :: forall a . CharParser st a -> CharParser st [a]
   , commaSep1   :: forall a . CharParser st a -> CharParser st [a]
   }
```

LanguageDef st

The `LanguageDef` type is a record that contains all parameterizable features of the `ParsecToken` module. The module `ParsecLanguage` contains some default definitions. The members of `LanguageDef` are described in a later section.

```
data LanguageDef st
  = LanguageDef
  { commentStart   :: String
  , commentEnd     :: String
  , commentLine    :: String
  , nestedComments :: Bool
  , identStart     :: CharParser st Char
  , identLetter    :: CharParser st Char
  , opStart        :: CharParser st Char
  , opLetter       :: CharParser st Char
  , reservedNames  :: [String]
  , reservedOpNames:: [String]
  , caseSensitive  :: Bool
  }
```

makeTokenParser :: `LanguageDef st -> TokenParser st`

The expression `(makeTokenParser language)` creates a `TokenParser` record that contains lexical parsers that are defined using the definitions in the `language` record.

The use of this function is quite stylized – one imports the appropiate language definition and selects the lexical parsers that are needed from the resulting `TokenParser`.

```
module Main where

import Parsec
import qualified ParsecToken as P
import ParsecLanguage (haskellDef)

-- The parser
...

expr  =   parens expr
      <|> identifier
      <|> ...


-- The lexer
lexer       = P.makeTokenParser haskellDef

parens      = P.parens lexer
braces      = P.braces lexer
identifier  = P.identifier lexer
reserved    = P.reserved lexer
...
```

3.8 The members of `TokenParser`

The following functions are all members of the `TokenParser` record but they are described as if they are top-level functions since this how they are normally used (as shown in the example of `makeTokenParser`).

whiteSpace :: `CharParser st ()`

Parses any white space. White space consists of *zero* or more occurrences of a `space`, a line comment or a block (multi line) comment. Block comments may be nested. How comments are started and ended is defined in the `LanguageDef` that is passed to `makeTokenParser`.

lexeme :: `CharParser st a -> CharParser st p`

`(lexeme p)` first applies parser `p` and than the `whiteSpace` parser, returning the value of `p`. Every lexical token (lexeme) is defined using `lexeme`, this way every parse starts at a point without white space. Parsers that use `lexeme` are called *lexeme* parsers in this document.

The only point where the `whiteSpace` parser should be called explicitly is the start of the main parser in order to skip any leading white space.

```
mainParser  = do{ whiteSpace
                ; ds <- many (lexeme digit)
                ; eof
                ; return (sum ds)
                }
```

symbol :: `String -> CharParser st String`

Lexeme parser `(symbol s)` parses `string` `s` and skips trailing white space.

parens :: `CharParser st a -> CharParser st a`

Lexeme parser `(parens p)` parses `p` enclosed in parenthesis, returning the value of `p`. It can be defined as:

```
parens p  = between (symbol "(") (symbol ")") p
```

braces :: `CharParser st a -> CharParser st a`

Lexeme parser `(braces p)` parses `p` enclosed in braces (`'{'` and `'}'`), returning the value of `p`.

brackets :: `CharParser st a -> CharParser st a`

Lexeme parser `(brackets p)` parses `p` enclosed in brackets (`'<'` and `'>'`), returning the value of `p`.

squares :: `CharParser st a -> CharParser st a`

Lexeme parser `(squares p)` parses `p` enclosed in square brackets (`'['` and `']'`), returning the value of `p`.

semi :: `CharParser st String`

Lexeme parser `semi` parses the character `';'` and skips any trailing white space. Returns the string `";"`.

comma `:: CharParser st String`

Lexeme parser `comma` parses the character `','` and skips any trailing white space. Returns the string `","`.

colon `:: CharParser st String`

Lexeme parser `colon` parses the character `':'` and skips any trailing white space. Returns the string `":"`.

dot `:: CharParser st String`

Lexeme parser `dot` parses the character `'.'` and skips any trailing white space. Returns the string `"."`.

semiSep `:: CharParser st a -> CharParser st [a]`

Lexeme parser `(semiSep p)` parses *zero* or more occurrences of `p` separated by `semi`. Returns a list of values returned by `p`. This combinator can be defined as:

```
semiSep p  = sepBy p semi
```

semiSep1 `:: CharParser st a -> CharParser st [a]`

Lexeme parser `(semiSep1 p)` parses *one* or more occurrences of `p` separated by `semi`. Returns a list of values returned by `p`.

commaSep `:: CharParser st a -> CharParser st [a]`

Lexeme parser `(commaSep p)` parses *zero* or more occurrences of `p` separated by `comma`. Returns a list of values returned by `p`.

commaSep1 `:: CharParser st a -> CharParser st [a]`

Lexeme parser `(commaSep1 p)` parses *one* or more occurrences of `p` separated by `comma`. Returns a list of values returned by `p`.

charLiteral `:: CharParser st Char`

This lexeme parser parses a single literal character. Returns the literal character value. This parsers deals correctly with escape sequences. The literal character is parsed according to the grammar rules defined in the Haskell report (which matches most programming languages quite closely).

stringLiteral `:: CharParser st String`

This lexeme parser parses a literal string. Returns the literal string value. This parsers deals correctly with escape sequences and gaps. The literal string is parsed according to the grammar rules defined in the Haskell report (which matches most programming languages quite closely).

decimal `:: CharParser st Integer`

Parses a positive whole number in the decimal system. Returns the value of the number.

hexadecimal :: `CharParser st Integer`

Parses a positive whole number in the hexadecimal system. The number should be prefixed with `"0x"` or `"0X"`. Returns the value of the number.

octal :: `CharParser st Integer`

Parses a positive whole number in the octal system. The number should be prefixed with `"0o"` or `"0O"`. Returns the value of the number.

natural :: `CharParser st Integer`

This lexeme parser parses a natural number (a positive whole number). Returns the value of the number. The number can be specified in `decimal`, `hexadecimal` or `octal`. The number is parsed according to the grammar rules in the Haskell report.

integer :: `CharParser st Integer`

This lexeme parser parses an integer (a whole number). This parser is like `natural` except that it can be prefixed with sign (i.e. `'-'` or `'+'`). Returns the value of the number. The number can be specified in `decimal`, `hexadecimal` or `octal`. The number is parsed according to the grammar rules in the Haskell report.

float :: `CharParser st Double`

This lexeme parser parses a floating point value. Returns the value of the number. The number is parsed according to the grammar rules defined in the Haskell report.

naturalOrFloat :: `CharParser st (Either Integer Double)`

This lexeme parser parses either `natural` or a `float`. Returns the value of the number. This parsers deals with any overlap in the grammar rules for naturals and floats. The number is parsed according to the grammar rules defined in the Haskell report.

identifier :: `CharParser st String`

This lexeme parser parses a legal identifier. Returns the identifier string. This parser will fail on identifiers that are reserved words. Legal identifier (start) characters and reserved words are defined in the `LanguageDef` that is passed to `makeTokenParser`. An `identifier` is treated as a single token using `try`.

reserved :: `String -> CharParser st String`

The lexeme parser `(reserved name)` parses `(symbol name)`, but it also checks that the `name` is not a prefix of a valid identifier. A `reserved` word is treated as a single token using `try`.

operator :: `CharParser st String`

This lexeme parser parses a legal operator. Returns the name of the operator. This parser will fail on any operators that are reserved operators. Legal operator (start) characters and reserved operators are defined in the `LanguageDef` that is passed to `makeTokenParser`. An `operator` is treated as a single token using `try`.

reservedOp :: `String -> CharParser st String`

The lexeme parser `(reservedOp name)` parses `(symbol name)`, but it also checks that the `name` is not a prefix of a valid operator. A `reservedOp` is treated as a single token using `try`.

3.9 The members of `LanguageDef`

The following functions are all members of the `LanguageDef` record but they are described as if they are top-level values. These values determine how lexers that are constructed using `makeTokenParser` will behave.

commentStart :: `String`

Describes the start of a block comment. Use the empty string if the language doesn't support block comments. For example `"/*"`.

commentEnd :: `String`

Describes the end of a block comment. Use the empty string if the language doesn't support block comments. For example `"*/"`.

commentLine :: `String`

Describes the start of a line comment. Use the empty string if the language doesn't support line comments. For example `"//"`.

nestedComments :: `Bool`

Set to `True` if the language supports nested block comments.

identStart :: `CharParser st Char`

This parser should accept any start characters of identifiers. For example `(letter <|> char "_")`.

identLetter :: `CharParser st Char`

This parser should accept any legal tail characters of identifiers. For example `(alphaNum <|> char "_")`.

opStart :: `CharParser st Char`

This parser should accept any start characters of operators. For example `(oneOf ":!#$%&*+./<=>?@\\^|- ")`

opLetter :: `CharParser st Char`

This parser should accept any legal tail characters of operators. Note that this parser should even be defined if the language doesn't support user-defined operators, or otherwise the `reservedOp` parser won't work correctly.

reservedNames :: `[String]`

The list of reserved identifiers.

reservedOpNames :: `[String]`

The list of reserved operators.

caseSensitive :: `Bool`

Set to `True` if the language is case sensitive.

3.10 Module `ParsecLanguage`

This module defines some default `LanguageDef` definitions. These definitions provide easy starting points for defining your own `LanguageDef` definitions.

emptyDef :: `LanguageDef st`

This is the most minimal token definition. It is recommended to use this definition as the basis for other definitions. `emptyDef` has no reserved names or operators, is case sensitive and doesn't accept comments, identifiers or operators.

```
emptyDef
 = LanguageDef
   { commentStart   = ""
   , commentEnd     = ""
   , commentLine    = ""
   , nestedComments = True
   , identStart     = letter <|> char '_'
   , identLetter    = alphaNum <|> oneOf "_'"
   , opStart        = opLetter emptyDef
   , opLetter       = oneOf ":!#$%&*+./<=>?@\\^|-~"
   , reservedOpNames= []
   , reservedNames  = []
   , caseSensitive  = True
   }
```

javaStyle :: `LanguageDef st`

This is a minimal token definition for Java style languages. It defines the style of comments, valid identifiers and case sensitivity. It does not define any reserved words or operators.

```
javaStyle
    = emptyDef
    { commentStart   = "/*"
    , commentEnd     = "*/"
    , commentLine    = "//"
```

```
        , nestedComments = True
        , identStart     = letter
        , identLetter    = alphaNum <|> oneOf "_'"
        , reservedNames  = []
        , reservedOpNames= []
        , caseSensitive  = False
        }
```

haskellStyle :: LanguageDef st

This is a minimal token definition for Haskell style languages. It defines the style of comments, valid identifiers and case sensitivity. It does not define any reserved words or operators.

```
haskellStyle
    = emptyDef
    { commentStart   = "{-"
    , commentEnd     = "-}"
    , commentLine    = "--"
    , nestedComments = True
    , identStart     = letter
    , identLetter    = alphaNum <|> oneOf "_'"
    , opStart        = opLetter haskell
    , opLetter       = oneOf ":!#$%&*+./<=>?@\\^|-~"
    , reservedOpNames= []
    , reservedNames  = []
    , caseSensitive  = True
    }
```

mondrianDef :: LanguageDef st

The language definition for the language Mondrian.

```
mondrian
    = javaStyle
    { reservedNames  = [ "case", "class", "default"
                       , "extends"
                       , "import", "in", "let"
                       , "new", "of", "package"
                       ]
    , caseSensitive  = True
    }
```

haskell98Def :: LanguageDef st

The language definition for the language Haskell98.

```
haskell98Def
    = haskellStyle
    { reservedOpNames= ["::","..","=","\\","|"
                       ,"<-","->","@","~","=>"
                       ]
    , reservedNames  = ["let","in","case","of"
                       ,"if","then","else"
                       ,"data","type",
                                   ,"class","default","deriving"
                                   ,"do","import",
                                   ,"infix","infixl","infixr"
                                   ,"instance","module"
                                   ,"newtype","where"
                                   ,"primitive"
                                    -- "as","qualified","hiding"
                                   ]
                 }
```

haskellDef :: `LanguageDef st`

The language definition for the Haskell language.

```
haskellDef
  = haskell98Def
    { identLetter    = identLetter haskell98Def <|> char '#'
    , reservedNames  = reservedNames haskell98Def ++
                       ["foreign","import","export","primitive"
                       ,"_ccall_","_casm_"
                       ,"forall"
                       ]
    }
```

mondrian :: `TokenParser st`

A lexer for the mondrian language.

```
mondrian  = makeTokenParser mondrianDef
```

haskell :: `TokenParser st`

A lexer for the haskell language.

```
mondrian  = makeTokenParser haskellDef
```

3.11 Module `ParsecExpr`

`ParsecExpr` is a small extension module for parsing expression grammars. It imports the `Parsec` module.

Assoc

This data type specifies the associativity of operators: left, right or none.

```
data Assoc  = AssocNone
            | AssocLeft
            | AssocRight
```

Operator tok st a

This data type specifies operators that work on values of type `a`. An operator is either binary infix or unary prefix or postfix. A binary operator has also an associated associativity.

```
data Operator tok st a
  = Infix (GenParser tok st (a -> a -> a)) Assoc
  | Prefix (GenParser tok st (a -> a))
  | Postfix (GenParser tok st (a -> a))
```

OperatorTable tok st a

An `(OperatorTable tok st a)` is a list of `(Operator tok st a)` lists. The list is ordered in descending precedence. All operators in one list have the same precedence (but may have a different associativity).

```
type OperatorTable tok st a  = [[Operator tok st a]]
```

buildExpressionParser `:: OperatorTable tok st a -> GenParser tok st a -> GenParser tok st a`

`(buildExpressionParser table term)` builds an expression parser for terms `term` with operators from `table`, taking the associativity and precedence specified in `table` into account. Prefix and postfix operators of the same precedence can only occur once (i.e. `--2` is not allowed if `-` is prefix negate). Prefix and postfix operators of the same precedence associate to the left (i.e. if `++` is postfix increment, than `-2++` equals `-1`, not `-3`).

The `buildExpressionParser` takes care of all the complexity involved in building expression parser. Here is an example of an expression parser that handles prefix signs, postfix increment and basic arithmetic.

```
expr    = buildExpressionParser table term
        <?> "expression"

term    =  parens expr
        <|> natural
        <?> "simple expression"

table :: OperatorTable Char () Integer
table   = [ [prefix "-" negate, prefix "+" id ]
          , [postfix "++" (+1)]
          , [binary "*" (*) AssocLeft, binary "/" (div) AssocLeft ]
          , [binary "+" (+) AssocLeft, binary "-" (-)   AssocLeft ]
          ]

binary  name fun assoc = Infix (do{ reservedOp name; return fun }) assoc
prefix  name fun       = Prefix (do{ reservedOp name; return fun })
postfix name fun       = Postfix (do{ reservedOp name; return fun })
```

3.12 Module ParsecPerm

This module contains combinators for building parsers that parse permutation phrases. It uses a typed approach that is descibed in (Baars *et al.*, 2001). This library is still experimental and might change.

PermParser tok st a

The type `(PermParser tok st a)` denotes a permutation parser that, when converted by the `permute` function, parses token streams of type `tok` with user state `st` and returns a value of type `a` on success.

Normally, a permutation parser is first build with special operators like `(<||>)` and than transformed into a normal parser using `permute`.

permute :: `PermParser tok st a -> GenParser tok st a`

The parser `(permute perm)` parses a permutation of parser described by `perm`. For example, suppose we want to parse a permutation of: an optional string of `a`'s, the character `b` and an optional `c`. This can be described by:

```
test :: Parser (String,Char,Char)
test  = permute (tuple <$?> ("",many1 (char 'a'))
                       <||> char 'b'
                       <|?> ('_',char 'c'))
      where
        tuple a b c  = (a,b,c)
```

`(<$$>) :: (a -> b) -> GenParser tok st a -> PermParser tok st b`
`infixl 2`

The expression `(f <$$> p)` creates a fresh permutation parser consisting of parser `p`. The the final result of the permutation parser is the function `f` applied to the return value of `p`. The parser `p` is not allowed to accept empty input – use the optional combinator `(<$?>)` instead.

If the function `f` takes more than one parameter, the type variable `b` is instantiated to a functional type which combines nicely with the adds parser `p` to the `(<||>)` combinator. This results in stylized code where a permutation parser starts with a combining function `f` followed by the parsers. The function `f` gets its parameters in the order in which the parsers are specified, but actual input can be in any order.

```
permute $
f <$$> parse field x
  <||> parse field y
where
  f x y = ...
```

`(<||>) :: PermParser tok st (a -> b) -> GenParser tok st a -> PermParser tok st b`
`infixl 1`

The expression `(perm <||> p)` adds parser `p` to the permutation parser `perm`. The parser `p` is not allowed to accept empty input – use the optional combinator `(<|?>)` instead. Returns a new permutation parser that includes `p`.

`(<$?>) :: PermParser tok st (a -> b) -> (a,GenParser tok st a) -> PermParser tok st b`
`infixl 1`

The expression `(f <$?> (x,p))` creates a fresh permutation parser consisting of parser `p`. The the final result of the permutation parser is the function `f` applied to the return value of `p`. The parser `p` is optional – if it can not be applied, the default value `x` will be used instead.

```
(<|?>) :: PermParser tok st (a -> b) -> (a,GenParser tok st a) -> PermParser
tok st b
infixl 1
```

The expression `(perm <||> (x,p))` adds parser `p` to the permutation parser `perm`. The parser `p` is optional – if it can not be applied, the default value `x` will be used instead. Returns a new permutation parser that includes the optional parser `p`.

References

A. Aho, R. Sethi, J. Ullman. (1986) *Compilers: principles, techniques and tools.* Addison-Wesley.

Andrew Appel. (1997) *Mondern compiler implementation in Java/ML/C.* Cambridge University Press, ISDN 0-521-58654-2.

Atsushi Igarishi, Benjamin Pierce, and Philip Wadler. (November 1999) *Featherweight Java: A minimal core calculus for Java and GJ.* OOPSLA, Denver. `http://cm.bell-labs.com/cm/cs/who/wadler/papers/featherweight/featherweight.ps`.

Arthur Baars, Andres Loh, and Doaitse Swierstra. (2001) *Parsing Permutation Phrases.* Proceedings of the ACM SIGPLAN Haskell Workshop, 171–183.

W.H. Burge. (1975) *Recursive programming techniques.* Addison-Wesley.

Jeroen Fokker. (May 1995) *Functional Parsers.* Lecture Notes of the Baastad Spring school on Functional Programming.
`http://www.cs.uu.nl/~jeroen/article/parsers/parsers.ps`.

Andy Gill and Simon Marlow. (1995) *Happy: the parser generator for Haskell.* University of Glasgow. `http://www.haskell.org/happy`.

Steve Hill. (May 1996) *Combinators for parsing expressions.* Journal of Functional Programming **6**(3): 445-463.

Graham Hutton. (1992) *Higher-order functions for parsing.* Journal of Functional Programming **2**: 232-343.
`http://www.cs.nott.ac.uk/Department/Staff/gmh/parsing.ps`.

Graham Hutton and Erik Meijer. (1996) *Monadic Parser Combinators.* Technical report NOTTCS-TR-96-4. Department of Computer Science, University of Nottingham. `http://www.cs.nott.ac.uk/Department/Staff/gmh/monparsing.ps`.

Pieter Koopman and Rinus Plasmeijer. (1999) *Efficient Combinator Parsers.* Implementation of Functional Languages. Springer Verlag, LNCS **1595**: 122-138.
`ftp://ftp.cs.kun.nl/pub/CSI/SoftwEng.FunctLang/papers`.

Torben Mogensen. (1993) *Ratatosk: a parser generator and scanner generator for Gofer.* University of Copenhagen (DIKU).

Nigel Perry, Arjan van IJzendoor, and Erik Meijer. (2000) *Mondrian, the internet scripting language.* `http://www.mondrian-script.org/`.

Flemming Nielson, Hanne Riis Nielson and Chris Hankin. (1999) *Principles of program analysis.* Springer Verlag, ISBN 3-540-65410-0.

Andrew Partridge and David Wright. (1996) *Predictive parser combinators need four values to report errors.* Journal of Functional Programming **6**(2): 355-364.

Simon Peyton Jones and Erik Meijer. (June 1997) *Henk: a typed intermediate language.* Proceedings of the Types in Compilation Workshop, Amsterdam.
`http://www.research.microsoft.com/~simonpj/Papers/henk.ps.gz`.

Niklas Röjemo. (1995) *Garbage collection and memory efficiency in lazy functional languages.* Ph.D. thesis, Chalmers University of Technology.
`http://www.cs.chalmers.se/~rojemo/thesis.html`.

Doaitse Swierstra and Pablo Azero. (November 1999) *Fast, Error Correcting Parser Combinators: A Short Tutorial.* SOFSEM'99 Theory and Practice of Informatics. LNCS **1725**: 111-129. `http://www.cs.uu.nl/groups/ST/Software`.

Doaitse Swierstra and Luc Duponcheel. (1996) *Deterministic, Error-Correcting Combinator Parsers.* Advanced Functional Programming. LNCS **1129**: 185-207.
`http://www.cs.uu.nl/groups/ST/Software`.

Philip Wadler. (1985) *How to replace failure with a list of successes.* Functional Programming Languages and Computer Architecture, LNCS **201**: 113-128.

Philip Wadler. (1990) *Comprehending Monads.* ACM Conference on Lisp and Functional Programming, pages 61-77.
`http://cm.bell-labs.com/cm/cs/who/wadler/topics/monads.html`.

Philip Wadler. (1992) *The essence of functional programming.* Symposium on principles of programming languages, pages 1-14.
`http://cm.bell-labs.com/cm/cs/who/wadler/topics/monads.html`.

Index