Parsing with Haskell

Lennart Andersson
Computer Science
Lund University

October 28, 2001

Introduction

This will be an extended example that will provide the basis for a small program project resulting in an interpreter for an imperative language. The example will show how abstraction and higher order functions can be used to make the parsing program look like the grammar for the imperative language.

The example will decompose a nontrivial problem into a large number of very small problems. Most of them can be solved with almost trivial functions. Actually, the longest function in the example has just 7 lines of code. To do this decomposition requires a fair amount of experience; it should be a lot easier to understand the functions and the program. We believe it is a good programming strategy to decompose a nontrivial problem into problems that are small and almost trivial.

The programming language will have statements and integer expressions as its main building stones. Expressions will be represented in the interpreter with the following data type.

```
data Expr = Num Int | Var String | Add Expr Expr
          | Sub Expr Expr | Mul Expr Expr | Div Expr Expr
```

As an example the expression `2*(x+3)` will be represented by

```
Mul (Num 2) (Add (Var "x") (Num 3))
```

Statements will be represented using

```
data Statement =
  Assignment String Expr |
  ...
```

so that the assignment statement `count := count + 1` is represented by

```
Assignment "count" (Add (Var "count") (Num 1))
```

We are going to define parsing functions that takes such strings as arguments and returns the corresponding representation. We are going to introduce a few very simple parsers and some operators for combining parsers that may be used to construct complex parsers.

A parsing function will usually not parse all of the input string, but just a prefix. The remainder of the string will be the input to another parser. Suppose that we have a parser which accepts an identifier consisting of letters, another one accepting the string ":=" and a third one accepting an expression and returning a pair with some representation of the accepted string and a remainder string:

```
ident "count:=count+1;" -> ("count", ":=count+1;")
becomes ":=count+1:" -> (":=", "count+1;")
expr "count+1;" -> (Add (Var "count") (Num 1), ";")
```

The arrow `->` denotes evaluation. We can combine these parsers to a parser for an assignment statement:

```
assignment str = (Assignment id e, rest3) where
  (id, rest1) = ident str
  (_, rest2) = becomes rest1
  (e, rest3) = expr rest2
```

The type of such parsers would be

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

A `Parser a` takes a string argument and returns a pair. We will call the first component the result of the parser and second the remainder string. Thus we have

```
ident :: Parser String
becomes :: Parser String
expr :: Parser Expr
assignment :: Parser Statement
```

A parser should either accept or reject its input. The Prelude defines a data type for handling two such alternatives.

```
data Maybe a = Nothing | Just a
```

This type can be used in any context where a computation either fails or returns some valid result. We redefine our parser type

```
type Parser a = String -> Maybe (a,String)
```

and the parsers so that e.g.

```
ident "count:=count+1;" -> Just("count",":=count+1;")
ident "123:=count+1;" -> Nothing
```

The `type` definitions just give synonyms to type expressions; there are no constructors as in the case of `data` definitions.

Exercise 1 *Define* `semicolon :: Parser Char` *so that*

```
semicolon "; skip" -> Just(';', " skip")
semicolon "skip" -> Nothing
```

Exercise 2 *Define* `becomes`*.*

Exercise 3 *Define a parser* `char :: Parser Char` *which accepts one character. The solution appears on the next page, don't look!*

Basic parsers

The basic parsers are almost trivial. The first one will accept one character and is defined by

```
char :: Parser Char
char (c:cs) = Just(c,cs)
char [] = Nothing
```

`Parser Char` is a synonym for `String -> Maybe(Char,String)`. The function fails if the input string is empty. Some examples

```
char "" -> Nothing
char "1" -> Just('1',"")
char "abc" -> Just('a',"bc")
```

The next parser is even simpler.

```
fail :: Parser a
fail cs = Nothing
```

This parser will never accept any input, so it seems to be useless. Actually we will later define a similar parser which will print an error message and abort the computation. The name `fail` is defined in the Prelude so we have hide it when importing the Prelude. If you test `fail` by applying it to a string the system refuses to print `Nothing`:

```
> fail "abc"
ERROR: Cannot find "show" function for:
*** expression : fail "abc"
*** of type    : Maybe (a,[Char])
```

In order to print (show) the result the Haskell system must know how to print values of type `a` which occurs in `Maybe (a,[Char])` even if no such value is present in `Nothing`. Supplying a printable type fixes the problem:

```
> fail "abc" :: Maybe(Int,String)
Nothing
```

The same error is reported if you enter an empty list to the system:

```
> []
ERROR: Cannot find "show" function for:
*** expression : []
*** of type    : [a]
```

While `fail` is extreme in one way the next parser, `return`, is extreme in opposite sense; it will always succeed without inspecting the input string. It returns its first argument that may be of any type.

```
return :: a -> Parser a
return a cs = Just(a,cs)
```

Examples

```
return 0 "abc" -> Just(0,"abc")
return (Add (Var "count") (Num 1)) ";" -> Just(Add (Var "count") (Num 1)), ";")
```

The `return` function is defined in the Prelude for a similar but different purpose.

Parser operators

The first parser operator will be an infix operator denoted by `?`. If `m :: Parser a` is a parser and `p :: a -> Bool` is a predicate then `(m ? p)` is a parser which applies `m` to the input string and tests if the result satisfies `p`.

```
infix 7 ?
(?) :: Parser a -> (a -> Bool) -> Parser a
(m ? p) cs =
  case m cs of
    Nothing -> Nothing
    Just(a,cs) -> if p a then Just(a,cs) else Nothing
```

The `infix` directive declares `?` to be an infix operator with precedence 7. The precedence levels for the parser operators will be chosen so that most parentheses may be omitted.

The Prelude contains a predicate for deciding if a character is a digit, `isDigit`. We can define a parser accepting one digit

```
digit :: Parser Char
digit cs = (char ? isDigit) cs
```

Examples

```
digit "123" -> Just('1',"23")
digit "abc" -> Nothing
```

If the right member of a function definition applies an expression not containing the last argument to the last argument then this argument may be removed from both sides of the definition.

```
digit = char ? isDigit
```

The second parser operator corresponds to alternatives in the grammar. We will use an infix operator, `!`, for this parser combinator. If `m` and `n` are parsers of the same type then `(m ! n)` will be a parser which first applies `m` to the input string which will be the result unless `m` fails. In that case `n` is applied to the input.

The definition is simple:

```
infixl 3 !
(!) :: Parser a -> Parser a -> Parser a
(m ! n) cs =
  case m cs of
    Nothing -> n cs
    mcs -> mcs
```

The `infixl` directive tells the Haskell system `!` will be an infix operator which associates to the left. Left association means that `m ! n ! k` is evaluated as `(m ! n) ! k`. Some examples:

```
(char ! digit) "abc" -> Just('a', "bc")
(digit ! char) "abc" -> Just('a', "bc")
(digit ! return '0') "abc" -> Just('0', "abc")
(digit ! char) "" -> Nothing
```

We have assigned precedences to the operators so that `m ? p ! n` will mean `(m ? p) ! n` and not `m ? (p ! n)` which would give a type mismatch error.

Exercise 4 *The Prelude defines the predicates* `isAlpha, isSpace:: Char -> Bool` *which decide if a character is a letter or space character (blank, tab, newline). Define parsers accepting a letter and a space character,* `letter, space :: Parser Char`*.*

Exercise 5 *Define a parser* `alphanum :: Parser Char` *which accepts a letter or a digit.*

A parser accepting a given character is easily defined.

```
lit :: Char -> Parser Char
lit c = char ? (==c)
```

The operator section `(==c)` is equivalent to the predicate `(\x -> x==c)`. The predicate decides if the given character is the the same as the one accepted by `char`.

```
lit 'a' "abc" -> Just ('a',"bc")
lit 'b' "abc" -> Nothing
```

Exercise 6 *Redefine* `semicolon :: Parser Char` *using* `lit`*. The definition should not include the string argument.*

The next parser combinator applies two parsers in sequence where the remainder string from the first one is fed into the other. The results of the two parsers are combined into a pair.

```
infixl 6 #
(#) :: Parser a -> Parser b -> Parser (a, b)
(m # n) cs =
    case m cs of
      Nothing -> Nothing
      Just(p, cs') ->
             case n cs' of
               Nothing -> Nothing
               Just(q, cs'') -> Just((p,q), cs'')
```

A parser accepting two characters follows.

```
(char # char) ":=1" -> Just((':', '='), "1")
```

We may name it:

```
twochars :: Parser (Char, Char)
twochars = char # char
```

Exercise 7 *Redefine* `becomes :: Parser (Char, Char)` *using* `twochars` *and the* `?` *operator. The definition should not include the string argument.*

Sometimes we would like to transform the result of a parser. This may be done with

```
infixl 5 >->
(>->) :: Parser a -> (a -> b) -> Parser b
(m >-> k) cs =
  case m cs of
    Just(a,cs') -> Just(k a, cs')
    Nothing -> Nothing
```

The right operand of `>->` named `k` is the function defining the transformation. The result of the operation is a new parser. We may use it to define a parser which accepts a digit and returns an `Int` using `digitToInt` from the Prelude.

```
digitVal :: Parser Int
digitVal = digit >-> digitToInt
```

Example

```
digitVal "123" -> Just(1, "23")
digitVal "abc" -> Nothing
```

Exercise 8 *Define a parser that accepts a letter and returns an upper case letter. The Prelude function* `toUpper :: Char -> Char` *transforms any letter to its uppercase form.*

Exercise 9 *Define* `sndchar :: Parser Char` *which accepts two characters and returns the second one. Use* `twochars` *and the transformation operator. Examples*

```
sndchar "abc" -> Just('b', "c")
sndchar "a" -> Nothing
```

Exercise 10 *Redefine* `twochars :: Parser String` *so that it returns a* `String` *with two characters instead of a pair of characters.*

Exercise 11 *Define an operator* `(-#) :: Parser a -> Parser b -> Parser b` *which applies two parsers in sequence as* `#` *but throws away the result from the first one. Use* `#` *and* `>->` *in the definition. Examples*

```
(char -# char) "abc" -> Just('b', "c")
(char -# char) "a" -> Nothing
```

Exercise 12 *Define a similar operator* `(#-) :: Parser a -> Parser b -> Parser a` *which applies two parsers in sequence as* `#` *but throws away the result from the second one.*

It is useful to be able to apply a parser iteratively to the input string. First we define `iterate m i` which will apply the parser `m` to the input string `i` times with the result in a list with `i` elements. Example

```
iterate digitVal 3 "123456" -> Just([1, 2, 3], "456")
iterate letter 3 "a123" -> Nothing
```

The definition uses primitive recursion over natural numbers.

```
iterate :: Parser a -> Int -> Parser [a]
iterate m 0 = return []
iterate m i = m # iterate m (i-1) >-> cons
```

where

```
cons :: (a, [a]) -> [a]
cons (hd, tl) = hd:tl
```

The name `iterate` is defined in the Prelude so it is necessary to hide it or to rename the new function.

It is even more useful to be able to iterate a parser as long as it succeeds. Assuming that `iter` does this we could use it like follows

```
iter digitVal "123abc" -> Just([1, 2, 3], "abc")
iter digit "123abc" -> Just(['1', '2', '3'], "abc") -> Just("123", "abc")
iter digit "abc" -> Just([], "abc") -> Just("", "abc")
```

The definition will again be recursive but termination will be signaled by the failure of the recursive branch. In that case we just return the empty list.

```
iter :: Parser a -> Parser [a]
iter m = m # iter m >-> cons ! return []
```

Inserting parentheses to indicate precedences of the operators we get

```
iter m = ((m # (iter m)) >-> cons) ! (return [])
```

Exercise 13 *Why is it an error to take the alternatives in the opposite order?*

```
iter m = return [] ! m # iter m >-> cons
```

We can use `iter` to define a parser for a string of letters. Our first attempt is

```
letters :: Parser String
letters  = iter letter
```

This is OK provided we are willing to accept an empty string. If this is not the case we define it as

```
letters  = letter # iter letter >-> cons
```

We now consider the problem of whitespace, i.e. blanks and newline characters, in the input. We define `token` which for a given parser will return a parser which will remove any whitespace after the accepted string.

```
token :: Parser a -> Parser a
token m  = m #- iter space
```

We now define two very useful parsers.

```
word :: Parser String
word = token letters

accept :: String -> Parser String
accept w = token (iterate char (size w) ? (==w))
```

Examples

```
word "count := count+1" -> Just("count", ":= count+1")
word ":= count+1" -> Nothing
accept "while" "while x do x:=x-1" -> Just("while", "x do x:=x-1")
accept "if" "while x do x:=x-1" -> Nothing
```

Sometimes we need to make the result from one parser available to another. The following operator provides the means.

```
infix 4 #>
(#>) :: Parser a -> (a -> Parser b) -> Parser b
(m #> k) cs =
    case m cs of
     Nothing -> Nothing
     Just(a,cs') -> k a cs'
```

After applying the parser `m` to the input string both the result and the remainder input string are given to the second operand, `k :: a -> Parser b`.

This operator is very useful for parsing numbers and arithmetic expressions with operators that associate to the left like `-` and `/`. These parsers will be nontrivial so we start with a simpler and not so useful parser. It will accept two characters provided they are equal and returning just one character.

```
double :: Parser Char
double = char #> lit
```

Remember that `lit :: Char -> Parser Char` needs on argument to become a parser which will accept this character. The character is provided as the result of the first parser. Examples

```
double "aab" -> Just('a', "b")
double "abb" -> Nothing
```

In order to construct a number parser we define three functions, `bldNumber`, `number'` and `number`.

```
bldNumber :: Int -> Int -> Int
bldNumber n d = 10*n+d
```

`bldNumber n d` just “appends” the digit `d` after `n`.

```
bldNumber 123 4 -> 1234

number' :: Int -> Parser Int
number' n =
  digitVal >-> bldNumber n #> number'
  ! return n

number :: Parser Int
number = token (digitVal #> number')
```

The argument in `number'` will serve as an accumulator which will be returned when no more input can be consumed. Otherwise the next digit is "appended" to the accumulator and recursively given to `number'`. The `number` parser accepst the first digit with `digitVal` and sends it to `number'` and finally removes any trailing whitespace. Examples

```
number' 1 "abc" -> Just(1, "abc")
number' 1 "23 abc" -> Just(123, " abc")
number "123 abc" -> Just(123, "abc")
```

Exercise 14 *Define the operator* `#` *using* `#>`*. The solution should not contain* `Just` *or* `Nothing`*.*

An expression parser

We are going to construct a parser for arithmetic expressions with integer numbers, variables, additions, subtractions, multiplications, divisions and parentheses with the usual laws for operator precedence. A grammar is given by

```
expr  ::= term expr'
expr' ::= addOp term expr' | empty
term  ::= factor term'
term' ::= mulOp factor term' | empty
factor::= num | var | "(" expr ")"
addOp ::= "+" | "-"
mulOp ::= "*" | "/"
```

The word `empty` denotes the empty string. In the definition of `factor`, `num` is a string of digits and `var` a string of letters. There are several grammars describing ordinary expressions. The current one has been chosen since it will be possible to define a parser which is very similar. The result of the parser will be a value of the following type.

```
data Expr =
  Num Int | Var String | Add Expr Expr |
  Sub Expr Expr | Mul Expr Expr | Div Expr Expr
```

We are going to build the parser from the bottom starting with parsers for `mulOp` and `addOp`. `mulOp` will accept either `*` or `/` and will return `Mul` or `Div`. These parsers are trivial to define without our parser operators:

```
mulOp ('*':rest) = Just(Mul, rest)
mulOp ('/':rest) = Just(Div, rest)
mulOp _ = Nothing
```

We prefer, however, to make the definition on the abstract level. The reason is that if we do not exploit the fact that the representation by the `Maybe` type is visible outside the `Parser` module then we may change it without changing any module using the `Parser` module. Such a change is suggested at the end of this paper.

```
mulOp, addOp :: Parser ((Expr, Expr) -> Expr)
mulOp = lit '*' >-> (\_ -> Mul)
      ! lit '/' >-> (\_ -> Div)
```

Examples

```
mulOp "*2*3" -> Just(Mul, "2*3")
mulOp "/2*3" -> Just(Div, "2*2")
mulop "+2*3" -> Nothing
```

The `addOp` parser is analogous.

The `num` and `var` parsers are simple; we just give the result from `word` and `number` to the relevant `Expr` constructor.

```
var :: Parser Expr
var = word >-> Var
num :: Parser Expr
num = number >-> Num
```

Examples

```
var "count + 1" -> Just(Var "count", "+ 1")
num "123*456" -> Just(Num 123, "*456")
```

The `expr`, `term` and `factor` parsers are going to be mutually recursive. It may be simpler to define and test such functions if one temporarily eliminates the recursion. We do it by simplifying the factor production.

```
factor :: num | var | "(" var ")"
```

We already have parsers for the two first alternatives. A parser for the last alternative is given by

```
lit '(' -# var #- lit ')'
```

We conclude with

```
factor :: Parser Expr
factor = num ! var ! lit '(' -# var #- lit ')'
```

A straightforward implementation of the rule for terms would parse e.g. the expression 6/3/2 as 6/(3/2) contrary to the ordinary rules. In order to get the interpretation (6/3)/2 we may use the same technique as in the definition of `number'` with a parameter to accumulate the result.

```
term' :: Expr -> Parser Expr
term' e =
   mulOp # factor >-> bldOp e #> term' !
   return e
```

The parameter `e` will be the result of the previous parser. If the input doesn't start with a `mulOp` we just return `e`. Otherwise `mulOp # factor` results in a pair which is given to `bldOp e`. The transformed result is given recursively to `term'`. The transformation `bldOp` is defined by

```
bldOp :: Expr -> (Expr -> Expr -> Expr, Expr) -> Expr
bldOp e (oper,e') = oper e e'
```

Example

```
bldOp (Num 1) (Mul, Num 2) -> Mul Num 1 Num 2
```

The result of all this is given recursively to `term'` which will find a new `mulOp` or terminate.

```
term' (Num 1) "*2" ->
  (mulOp # factor >-> bldOp (Num 1) #> term') "*2" ->
  (term' (bldOp (Num 1) (Mul, Num 2))) "" ->
  (term' (Mul (Num 1) (Num 2))) "" ->
  Just(Mul (Num 1) (Num 2), "")
```

The definition of `term` is simple; parse a factor and send the result to `term'`.

```
term :: Parser Expr
term = factor #> term'
```

Examples

```
term' (Num 1) "" -> Just(Num 1, "")
term "1*2" -> Just(Mul (Num 1) (Num 2), "")
term "6/3/2" -> Just(Div (Div (Num 6) (Num 3)) (Num 2), "")
```

The definitions of `expr'` and `expr` are analogous.

```
expr' :: Expr -> Parser Expr
expr' e =
    addOp # term >-> bldOp e #> expr' !
    return e
expr :: Parser Expr
expr = term #> expr'
```

We may now compose the `factor`, `term` and `expr` parsers resetting the first one to make them mutually recursive.

```
factor =
    num !
    var !
    lit '(' -# expr #- lit ')' !
    err "illegal factor"
term' e =
    mulOp # factor >-> bldOp e #> term' !
    return e
term = factor #> term'
expr' e =
    addOp # term >-> bldOp e #> expr' !
    return e
expr = term #> expr'
```

We have added another alternative parser in `factor` in order to get an informative message for illegal input. Without it, any invalid string argument will just return `Nothing`.

The `err` parser is defined by

```
err :: String -> Parser a
err message cs = error (message ++ " near " ++ cs ++ "\n")
```

It will print the message and the offending input and abort the computation.

Exercise 15 *Define a parser* `require :: String -> Parser a` *that behaves as* `accept` *but emits an error message instead of returning* `Nothing`*.*

Program structure

The program parts that we have developed so far should be divided into two parser modules and one expression module. The first parser module should contain the parser type, the basic parsers and parser operators, while the second one contains derived parsers and parser operators. The representation of parsers as values of type `String -> Maybe(a, String)` will not be exploited outside the first parser module.

Since we are going to define parsers for more than one data type it is useful to have a type class declaring the names `parse` and `toString` and defining `fromString`. We can then use these names for all the data types and the relevant one will be chosen using the context where the name is used. We use the name `Parse` for the type class.

It is inconvenient to use qualified import for the parser module, so we refrain from giving the parser type the conventional name `T`.

```
module CoreParser(Parser, char, return, fail, (#), (!), (?), (#>), (>->),
                  Parse, parse, toString, fromString) where
import Prelude hiding (return, fail)
infixl 3 !
infixl 7 ?
infixl 6 #
infixl 5 >->
infixl 4 #>

class Parse a where
    parse :: Parser a
    fromString :: String -> a
    fromString cs =
        case parse cs of
               Just(s, []) -> s
               Just(s, cs) -> error ("garbage '"++cs++"'")
               Nothing -> error "Nothing"
    toString :: a -> String

type Parser a = String -> Maybe (a, String)

char :: Parser Char
char []= Nothing
char (c:cs) = Just (c, cs)

return :: a -> Parser a
return a cs = Just (a, cs)

fail ::  Parser a
fail cs = Nothing
```

```
(!) :: Parser a -> Parser a -> Parser a
(m ! n) cs = case m cs of
             Nothing -> n cs
             mcs -> mcs

(?) :: Parser a -> (a -> Bool) -> Parser a
(m ? p) cs =
    case m cs of
    Nothing -> Nothing
    Just(r, s) -> if p r then Just(r, s) else Nothing

(#) :: Parser a -> Parser b -> Parser (a, b)
(m # n) cs =
    case m cs of
    Nothing -> Nothing
    Just(a, cs') ->
        case n cs' of
        Nothing -> Nothing
        Just(b, cs'') -> Just((a, b), cs'')

(>->) :: Parser a -> (a -> b) -> Parser b
(m >-> b) cs =
    case m cs of
    Just(a, cs') -> Just(b a, cs')
    Nothing -> Nothing

(#>) :: Parser a -> (a -> Parser b) -> Parser b
(p #> k) cs =
    case p cs of
    Nothing -> Nothing
    Just(a, cs') -> k a cs'
```

The derived parsers appear in `Parser.hs`.

```
module Parser(module CoreParser, digit, digitVal, chars, letter, err,
              lit, number, iter, accept, require, token,
              spaces, word, (-#), (#-)) where
import Prelude hiding (return, fail)
import CoreParser
infixl 7 -#, #-

type T a = Parser a

err :: String -> Parser a
err message cs = error (message++" near "++cs++"\n")

iter :: Parser a -> Parser [a]
iter m = m # iter m >-> cons ! return []

cons(a, b) = a:b
. . .
```

The expression module will have a few other functions not mentioned previously. We rename the main type to `T`. The `value` function should evaluate an expression. It will need a "dictionary" to find the values of the variables.

The expression module will make heavy use of the parser functions so it is convenient to import the parser module without qualification.

```
module Expr(Expr, T, parse, fromString, value, toString) where
import Prelude hiding (return, fail)
import Parser
import qualified Dictionary

data Expr = Num Integer | Var String | Add Expr Expr
       | Sub Expr Expr | Mul Expr Expr | Div Expr Expr
         deriving Show

type T = Expr
        ...
instance Parse Expr where
    parse = expr
    toString = shw 0
```

Parsing with ambiguous grammars

This description is mainly inspired by an article by Phil Wadler, [4]. He uses an elegant trick to code the failure and the success of a parser and finding all possible parses of a string. The `Maybe` data type may be viewed as a special case of the list type, but with the list either being empty, `Nothing`, or having one element, `Just a`. Failure will be represented by the empty list while a nonempty list contains all possible successes. He defines the parser type by

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

As an example the number parser will find

```
number "123abc" -> [(123,"abc"),(12,"3abc"),(1,"23abc")]
number "abc" -> []
```

A few modifications are needed in the Parser module. The basic parser must be changed to return lists instead of `Maybe` values.

```
char [] = []
char (c:cs) = [(c,cs)]

return a cs = [(a,cs)]

fail cs = []
```

The operators must also be adapted. Using list comprehension they become "one-liners"

```
(m ? p) cs = [(a,cs') | (a,cs') <- m cs, p a]

(m ! n) cs = (m cs) ++ (n cs)
```

```
(m # n) cs = [((a,b),cs'') | (a,cs') <- m cs, (b,cs'') <- n cs']

(m >-> b) cs = [(b a, cs') | (a,cs') <- m cs]

(p #> k) cs = [(b,cs'') | (a,cs') <- p cs, (b,cs'') <- k a cs']
```

The `err` and hence the `require` parsers can no longer be used in `Expr` and `Program` to give adequate error messages.

Parsec

There is a "industrial stength" parser library called *Parsec*, [2]. It uses basically the same technique as above, but is larger and uses `do` notation extensively. An expression parser using this library follows.

```
import Parsec
import ParsecExpr

data Expr = Num Int | Var String | Add Expr Expr
       | Sub Expr Expr | Mul Expr Expr | Div Expr Expr
         deriving Show

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

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

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

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

variable :: Parser Expr
variable = do{ ds<- many1 letter
            ; return (Var ds)}
       <?> "variable"
```

The library has special support for parsing expressions. Operator precedences and associativities are given using a table. `do` notation is used to express sequential composition of parsers. Alternatives are separated by `<|>` and `<?>` is used as our `err` parser.

References

[1] Hutton, G. and Meier, E., *Monadic Parser Combinators*, Technical report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[2] Leijen, D., *Parsec, a fast combinator parser*, Oct 2001, http://www.cs.ruu.nl/˜daan/parsec.html.

[3] Wadler P., *How to replace failure by a list of successes*, 2nd International Conference on Functional Programming Languages and Computer Architecture, Springer-Verlag, Nancy France, September 1985. http://cm.bell-labs.com/cm/cs/who/wadler/.

[4] Wadler, P., *Monads for functional programming*, Marktoberdorf Summer School on Program Design Calculi, Springer Verlag, NATO ASI Series F: Computer and systems sciences, Volume 118, August 1992. http://cm.bell-labs.com/cm/cs/who/wadler/.