Today's Topic

Parsing: Lexical and syntactical analysis

- Combinator parsing
- Monadic parsing

Lexical and Syntactical Analysis

- ...in the following summarized as *parsing*

...an application of functional programming typically used to demonstrate its power and elegance.

Enjoys a long history. As an example of early work see e.g...

- W. Burge. *Recursive Programming Techniques*, Addison-Wesley, 1975.

Parsing – Implementation Variants

Two variants...

- *Combinator parsing*
  $\leadsto$ *recursive descent* parsing
  - Graham Hutton. *Higher-Order Functions for Parsing*. Journal of Functional Programming 2(3):323-343, 1992.
- *Monadic parsing*
  - Graham Hutton, Erik Meijer. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4, Dept. of Computer Science, University of Nottingham, 1996.

Reference

The following presentation is based on...

- Chapter 17
  Simon Thompson. *Haskell – The Craft of Functional Programming*, Addison-Wesley, 2nd edition, 1999.
- Graham Hutton, Erik Meijer. *Monadic Parsing in Haskell*. Journal of Functional Programming 1(1), 1993.

Parsing informally

The basic problem...

- Read a sequence of objects of type `a` and
- extract from this sequence an object or a list of objects of type `b`.

Example: Parsing of Expressions

Consider...

- Expressions

```
data Expr = Lit Int | Var Name | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod

Op Mul (Op Add (Lit 2) (Lit 3)) (Lit 3)
                                   corresponds to  ((2+3)*3)
```

The parsing task to be solved...

- Read an expression of the form `((2+3)*5)` and yield the corresponding expression of type `expr`.

  (Note: This can be considered the reverse of the `show` function. It is similar to the derived `read` function, but differs in the arguments it takes (expressions of the form `((2+3)*5)` vs. expressions of the form `Op Mul (Add (Lit 2) (Lit 3)) (Lit 5)`).

Initial Considerations 1(2)

What should be the type of a parsing function?

```
type BSParse1 a b = [a] -> b

-- Parser  Input        Expected Output
bracket    "(xyz"  -->  '('
number     "234"   -->  2 or 23 or 234 ?
bracket    "234"   -->  no result, failure?
```

We have to answer...

How shall the parser behave if there ...

- ...are multiple results?
- ...is a failure?

Initial Considerations 2(2)

```
type BSParse2 a b = [a] -> [b]

-- Parser  Input        Expected Output
bracket    "(xyz"  -->  ['(']
number     "234"   -->  [2, 23, 234]
bracket    "234"   -->  []
```

Now we have to answer...

- What shall be done with the remaining input?

Type of the Parser 1(2)

The conclusion of our initial considerations...

```
type Parse a b = [a] -> [(b,[a])]

-- Parser  Input        Expected Output

bracket    "(xyz"  -->  [('(', "xyz")]
number     "234"   -->  [(2,"34"), (23,"4"), (234,"")]
bracket    "234"   -->  []
```

Remark:

- The capability of delivering multiple results enables the analysis of ambiguous grammars

  $\leadsto$ *list of successes* technique
- Each element in the output list represents a successful parse.

Type of the Parser 2(2)

Convention:

- *Delivery of the empty list* ...signals failure of the analysis.
- *Delivery of a non-empty list* ...signals success of the analysis; each element of the list is a pair, whose first component is the identified object (token) and whose second component is the input not yet considered.

Basic Parsers 1(3)

Primitive, input-*independent* parsing functions

- The always failing parsing function

```
none :: Parse a b
none inp = []
```

- The always successful parsing function

```
succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```

Remark:

- The `none` parser always fails. It does not accept anything.
- The `succeed` parser does not consume its input. In BNF-notation this corresponds to the symbol ε representing the empty word.

Basic Parsers 2(3)

Primitive, input-*dependent* parsing functions

- Recognizing single objects (token)...

```
token :: Eq a => a -> Parse a a
token t (x:xs)
  | t == x     =  [(t,xs)]
  | otherwise  =  []
token t []     =  []
```

- Recognizing single objects satisfying a particular property...

```
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x        =  [(x,xs)]
  | otherwise  =  []
spot p []      =  []
```

Basic Parsers 3(3)

Application:

```
bracket = token '('
dig     = spot isDigit

isDigit :: Char -> Bool
isDigit ch = ('0' <= ch) && (ch <= '9')
```

Note: ...`token` can be defined by means of `spot`

```
token t = spot (== t)
```


Combining Parsers 1(4)

...to obtain (more) complex parsing functions

$\leadsto$ *Combinator Parsing*

...building a library of higher-order polymorphic functions, which are then used to construct parsers

- Alternatives

  ```
  alt :: Parse a b -> Parse a b -> Parse a b
  alt p1 p2 inp = p1 inp ++ p2 inp
  ```

  Underlying intuition:

  ...an expression is *either* a literal, *or* a variable *or* an operator expression

  Example:

  ```
  (bracket 'alt' dig) "234"  -->  [] ++ [(2,"34")]
  ```

  $\leadsto$ ...the `alt` parser combines the results of the parses given by parsers `p1` and `p2`

Combining Parsers 2(4)

- Sequential composition of parsers

  ```
  infixr 5 >*>
  (>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
  (>*>) p1 p2 inp
    = [((y,z),rem2) | (y,rem1) <- p1 inp,
                                  (z,rem2) <- p2 rem1 ]
  ```

  Underlying intuition:

  ...an operator expression starts with a bracket *followed by* a number

Combining Parsers 3(4)

Example:

```
Because of  number "24(" --> [(2,"4("), (24,"(")]  we obtain

(number >*> bracket) "24("
   -->  [((y,z),rem2) | (y,rem1) <- [(2,"4("), (24,"(")],
                        (z,rem2) <- bracket rem1 ]
   -->  [((2,z),rem2) | (z,rem2) <- bracket "4(" ] ++
        [((24,z),rem2) | (z,rem2) <- bracket "(" ]

   -->  [] ++ [((24,z),rem2) | (z,rem2) <- bracket "(" ]

Because of  bracket "("  -->  [('(',"")]  we finally obtain

   -->  [((24,z),rem2) | (z,rem2) <- [('(',"")] ]
   -->  [ ((24,'('), "") ]
```

Combining Parsers 4(4)

- Transformation/Modification

  ⇝ change the item returned by the parser, or build something from it...

```
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x,rem) <- p inp ]
```

Example: (`digList` returns a list of numbers and shall be embedded such that the number represented by it is returned.)

```
(digList 'build' digsToNum) "21a3"
   --> [ (digsToNum x,rem) | (x,rem) <- digList "21a3" ]
   --> [ (digsToNum x,rem) | (x,rem) <-
                              [("2","1a3"),("21","a3")]]
   --> [ (digsToNum "2", "1a3"), (digsToNum "21", "a3") ]
   --> [ (2,"1a3"), (21,"a3") ]
```


The Clou

The combinators

- `alt`
- `>*>`
- `build`

together with the basic parsers constitute a universal "parser basis," i.e., allow to build any parser which might be desired.

Example: A Parser for a List of Objects

We suppose to be given a parser recognizing single objects:

```
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
         ((p >*> list p) 'build' (uncurry (:)))
```

Intuition:

- A list can be empty.
  ⇝ this is recognized by the parser `succeed []`
- A list can be non-empty, i.e., it consists of an object followed by a list of objects.
  ⇝ this is recognized by the combined parser `p >*> list p`, where we use `build` to turn a pair `(x,xs)` into the list `(x:xs)`.

Summary and Conclusion

...about combining parsers (*parser combinators*)

- Parsing functions in the above fashion are structurally similar to grammars in BNF-form. For each operator of the BNF-grammar there is a corresponding (higher-order) parsing function.
- These higher-order functions *combine* simple(r) parsing functions to (more) complex parsing functions.
- They are thus also called *combining forms*, or, as a short hand, *combinators* (cf. Graham Hutton. *Higher-Order Functions for Parsing*).

Overview of the Parsing Functions 1(4)

```
-- Sequence operator
infixr 5 >*>

-- Parser type
type Parse a b = [a] -> [(b,[a])]

-- Input-independent parsing functions
none :: Parse a b
none inp = []

succeed :: b -> Parse a b
succeed val inp = [(val,inp)]
```


Overview of the Parsing Functions 2(4)

```
-- Recognizing single objects
token :: Eq a => a -> Parse a a
token t = spot (==t)

-- Recognizing single objects satisfying a particular property
spot :: (a -> Bool) -> Parse a a
spot p (x:xs)
  | p x        =  [(x,xs)]
  | otherwise  =  []
spot p []      =  []
```


Overview of the Parsing Functions 3(4)

```
-- Alternatives
alt :: Parse a b -> Parse a b -> Parse a b
alt p1 p2 inp = p1 inp ++ p2 inp

-- Sequences
(>*>) :: Parse a b -> Parse a c -> Parse a (b,c)
(>*>) p1 p2 inp
  = [((y,z),rem2) | (y,rem1) <- p1 inp, (z,rem2) <- p2 rem1 ]

-- Transformation/Modification
build :: Parse a b -> (b -> c) -> Parse a c
build p f inp = [ (f x, rem) | (x,rem) <- p inp ]
```


Overview of the Parsing Functions 4(4)

```
-- Application example
list :: Parse a b -> Parse a [b]
list p = (succeed []) 'alt'
         ((p >*> list p) 'build' (uncurry (:)))
```

Application: Back to the Initial Example

We consider expressions of the form...

```
data Expr = Lit Int | Var Name | Op Ops Expr Expr
data Ops  = Add | Sub | Mul | Div | Mod

Op Add (Lit 2) (Lit 3)     corresponds to   2+3
```

...where the following convention shall hold:

- *Literals* ...67, ~89, etc., where ~ is used for unary minus
- *Names* ...the lower case characters from 'a' to 'z'
- *Applications of the binary operations* ...$+, *, -, /, \%$, where % is used for mod and / for integer division.
- Expressions are fully bracketed, and white space is not permitted.

A Parser for Expressions 1(3)

The parser consists...

```
parser :: Parse Char Expr
parser = litParse 'alt' nameParse 'alt' opExpParse
```

...of three parts corresponding to the three sorts of expressions.

Part I: Parsing names of variables

```
nameParse :: Parse Char Expr
nameParse = spot isName 'build' Name

isName :: Char -> Bool
isName x = ('a' <= x && x <= 'z')
```


A Parser for Expressions 2(3)

Part II: Parsing (fully bracketed binary) operator expressions

```
opExpParse
  = (token '(' >*>
     parser    >*>
     spot isOp >*>
     parser    >*>
     token ')')
     'build' makeExpr
```

Part III: Parsing literals (numerals)

```
litParse
  = ((optional (token '~')) >*>
     (neList (spot isDigit))
    'build' (charlistToExpr . uncurry (++))
```


A Parser for Expressions 3(3)

```
neList   :: Parse a b -> Parse a [b]
optional :: Parse a b -> Parse a [b]
```

such that:

- `neList p` recognizes a non-empty list of the objects which are recognized by `p`
- `optional p` recognizes an object recognized by `p` or succeeds immediately.

Note that `neList` and `optional` as well as a number of other supporting functions used such as...

- `isOp`
- `charlistToExpr`
- ...

are yet to be defined (⇝ exercise).

The Top-level Parser / Putting it all together

Converting a string to the expression it represents...

```
topLevel :: Parse a b -> [a] -> b
topLevel p inp
  = case results of
      [] -> error ‘‘parse unsuccessful’’
      _  -> head results
    where
    results = [ found | (found, []) <- p inp ]
```

Note:

- The input string is provided by the value of `inp`.
- The parse is successful, if the result contains at least one parse, in which all the input has been read.

Summary and Conclusions 1(2)

Parsers of the form...

```
type Parse a b = [a] -> [(b,[a])]

none :: Parse a b
succeed :: b -> Parse a b
spot :: (a -> Bool) -> Parse a a
alt :: Parse a b -> Parse a b -> Parse a b
>*> :: Parse a b -> Parse a c -> Parse a (b,c)
build :: Parse a b -> (b -> c) -> Parse a c
topLevel :: Parse a b -> [a] -> b
```

...support particularly well the construction of so-called *recursive descent* parsers.

Summary and Conclusions 2(2)

The following language features proved invaluable...

- *Higher-order functions* ...`Parse a b` is of a functional type; all parser combinators are thus higher-order functions, too.
- *Polymorphism* ...consider again the type of `Parse a b`: We do need to be specific about either the input or the output type of the parsers we build. Hence, the above parser combinator can immediately be reused for other (token-) and data types.
- *Lazy evaluation* ... *“on demand”* generation of the possible parses, automatical backtracking (the parsers will backtrack through the different options until a successful one is found).

Monadic Parsing

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

We use again the convention:

- *Delivery of the empty list* ...signals failure of the analysis
- *Delivery of a non-empty list* ...signals success of the analysis; each element of the list is a pair, whose first component is the identified object (token) and whose second component the input still to be examined

A Monad of Parsers

Basic Parsers...

- Recognizing single characters...

```
item :: Parser Char
item  = Parser (\cs -> case cs of
                             ""     -> []
                             (c:cs) -> [(c,cs)])
```

Compare: `item` vs. `token`

The Parser Monad

Reminder: The class monad...

```
class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b
```

Note: `Parser` is a type constructor. This allows...

```
instance Monad Parser where
   -- The always successful parser
   return a = Parser (\cs -> [(a,cs)])
   -- Sequences
   p >>= f  = Parser (\cs -> concat [parse (f a) cs' |
                                 (a,cs') <- parse p cs])
```

Compare: `return` vs. `succeed` and `(>>=)` vs. `infixr`

Properties of `return` and `(>>=)`

As required for instances of class `Monad`, we can show...

```
          return a >>= f  =  f a
            p >>= return  =  p
p >>= (\a -> (f a >>= g))  =  (p >>= (\a -> f a)) >>= g
```

Reminder:

- The above properties are required for each instance of class `Monad`, not just for the specific instance of the parser monad
  - ...`return` is left-unit and right-unit for `(>>=)`
    ⇝ ...allows a simpler and more concise definition of some parsers
  - ...`(>>=)` is associative
    ⇝ ...allows suppression of parentheses when parsers are applied sequentially

Typical Structure of a Parser 1(2)

...using the operator `(>>=)`

```
p1 >>= \a1 ->
p2 >>= \a2 ->
...
pn >>= \an ->
f a1 a2 ... an
```

Intuition:
There is a natural operational reading of such a parser...

- Apply parser `p1` and denote its result value `a1`
- Apply subsequently parser `p2` and denote its result value `a2`
- ...
- Apply concludingly parser `pn` and denote its result value `an`
- Combine finally the intermediate result values by applying some suitable function `f`

Typical Structure of a Parser 2(2)

The do-notation allows a more elegant and appealing notation...

```
do a1 <- p1
   a2 <- p2
   ...
   an <- pn
   f a1 a2 ... an
```

Alternatively, in just one line...

```
do {a1 <- p1; a2 <- p2; ...; an <- pn; f a1 a2 ... an}
```


Notational Conventions

Expressions of the form

- `ai <- pi` are called *generators*
  (since they generate values for the variables `ai`)

Remark:

A generator of the form `ai <- pi` can be

- replaced by `pi`, if the generated value will not be used afterwards

Example

A Parser `p`, which...

- reads three characters
- drops the second character of these and
- returns the first and the third character as a pair

Implementation:

```
p :: Parser (Char,Char)
p  = do {c <- item; item; d <- item; return (c,d)}
```


Parser Extensions 1(2)

Monads with a *zero* and a `plus` are captured by two built-in class definitions in Haskell...

```
class Monad m => MonadZero m where
   zero :: m a

class MonadZero m => MonadPlus m where
   (++) :: m a -> m a -> m a
```

Parser Extensions 2(2)

The type constructor `Parser` can be made instances of these two classes as follows:

- The parser which always fails...

```
instance MonadZero Parser where
   zero   = Parser (\cs -> [])
```

- The parser which non-deterministically selects...

```
instance MonadPlus Parser where
   p ++ q = (\cs -> parse p cs ++ parse q cs)
```


Simple Properties 1(2)

We can show...

```
      zero ++ p  =  p
      p ++ zero  =  p
p ++ (q ++ r)  =  (p ++ q) ++ r
```

Remark: The above properties are required to hold for each monad with *zero* and `plus`

Informally:

- ...`zero` is left-unit and right-unit for (++)
- ...(++) is associative

Simple Properties 2(2)

Specifically for the parser monad we can additionally show...

```
             zero >>= f  =  zero
        p >>= const zero  =  zero
          (p ++ q) >>= f  =  (p >>= f) ++ (q >>= f)
p >>= (\a -> f a ++ g a)  =  (p >>= f) ++ (p >>= g)
```

Informally:

- ...`zero` is left-zero and right-zero element for (>>=)
- ...(>>=) distributes through (++)

Deterministic Selection

The parser which deterministically selects...

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = Parser (\cs -> case parse (p ++ q) cs of
                            []     -> []
                            (x:xs) -> [x])
```

Note:

- (+++) shows the same behavior as (++), but yields at most one result
- (+++) satisfies all of the previously mentioned properties of (++)

Further Parsers

Recognizing...

- single objects satisfying a particular property

```
sat  :: (Char -> Bool) -> Parser Char
sat p = do {c <- item; if p c then return c else zero}
```

- single objects

```
char  :: Char -> Parser Char
char c = sat (c ==)
```

- sequences of numbers, lower case and upper case characters, etc.
  ...analogously to `char`

Compare: `sat` and `char` vs. `spot` and `token`

Recursion Combinators 1(3)

Useful parsers can often recursively be defined...

- Parse a specific string

```
string       :: String -> Parser String
string ""     = return ""
string (c:cs) = do {char c; string cs; return (c:cs)}
```

- Parse repeated applications of a parser p

```
(Zero or more applications of p)
many  :: Parser a -> Parser [a]
many p = many1 p +++ return []

(One or more applications of p)
many1  :: Parser a -> Parser [a]
many1 p = do {a <- p; as <- many p; return (a:as)}
```


Recursion Combinators 2(3)

- Similar to the parser `many` but with interspersed applications of the parser `sep`, whose result values are thrown away

```
sepby        :: Parser a -> Parser b -> Parser [a]
p 'sepby' sep = (p 'sepby1' sep) +++ return []

sepby1        :: Parser a -> Parser b -> Parser [a]
p 'sepby1' sep = do a  <- p
                    as <- many (do {sep; p})
                    return (a:as)
```


Recursion Combinators 3(3)

- Parse repeated applications of a parser `p`, separated by applications of a parser `op`, whose result value is an operator that is assumed to associate to the left, and which is used to combine the results from the p parsers

```
chainl       :: Parser a -> Parser (a -> a -> a) -> a -> Parser
chainl p op a = (p 'chainl1' op) +++ return a

chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p 'chainl1' op = do {a <- p; rest a}
                 where
                    rest a = (do f <- op
                                 b <- p
                                 rest (f a b))
                             +++ return a
```

Lexical Combinators

Suitable combinators allow suppression of a lexical analysis (token recognition), which traditionally precedes parsing...

- Parsing of a string with blanks and line breaks

```
space :: Parser String
space  = many (sat isSpace)
```

- Parsing of a token by means of parsers p

```
token  :: Parser a -> Parser a
token p = do {a <- p; space; return a}
```

- Parsing of a symbol token

```
symb   :: String -> Parser String
symb cs = token (string cs)
```

- Application of parser p, removal of initial blanks

```
apply  :: Parser a -> String -> [(a,String)]
apply p = parse (do {space; p}]
```


Example: Parsing of Expressions 1(3)

Grammar:

...for arithmetic expressions built up from single digits using the operators +, -, *, /, and parentheses:

```
expr   ::=  expr addop term | term
term   ::=  term mulop factor | factor
factor ::=  digit | (expr)
digit  ::=  0 | 1 | ... | 9

addop  ::=  + | -
mulop  ::=  * | /
```


Example: Parsing of Expressions 2(3)

Parsing and evaluating expressions (yielding integer values) using the `chainl1` combinator to implement the left-recursive production rules for `expr` and `term`...

```
expr  :: Parser Int
addop :: Parser (Int -> Int -> Int)
mulop :: Parser (Int -> Int -> Int)

expr   = term   'chainl1' addop
term   = factor 'chainl1' mulop
factor = digit +++ do {symb "("; n <- expr; symb ")"; return n}
digit  = do {x <- token (sat isDIgit); return (ord x - ord '0')}

addop  = do {symb "+"; return (+)} +++ do {symb "-"; return (-)}
mulop  = do {symb "*"; return (*)} +++ do {symb "/"; return (div)}
```


Example: Parsing of Expressions 3(3)

Example:

Evaluating

```
apply expr " 1 - 2 * 3 + 4 "
```

gives the singleton list

```
[(-1,"")] as desired
```

as desired.

Further Readings 1(3)

On combinator parsing...

- J. Fokker. *Functional Parsers*. In: *Advanced Functional Programming, First International Summer School*, Springer, LNCS 925 (1995), 1-23.
- S. Hill. *Combinators for Parsing Expressions*. Journal of Functional Programming 6:445-463, 1996.
- P. Koopman, R. Plasmeijer. *Efficient Combinator Parsers*. In Proceedings of Implementation of Functional Languages, Springer, LNCS 1595 (1999), 122-138.

Further Readings 2(3)

On error-correcting parsing...

- P. Wadler. *How to Replace Failure with a List of Successes*, in: Functional Programming Languages and Computer Architectures, Springer, LNCS 201 (1985), 113 - 128.
- D. Swierstra, P. Azero Alcocer. *Fast, Error Correcting Parser Combinators: A Short Tutorial*. In Proceedings SOFSEM'99, Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics, Springer, LNCS 1725 (1999), 111-129.
- D. Swierstra, L. Duponcheel. *Deterministic, Error Correcting Combinator Parsers*. In: *Advanced Functional Programming, Second International Spring School*, Springer, LNCS 1129 (1996), 184-207.

Further Readings 3(3)

On parser libraries...

- Daan Leijen, Erik Meijer. *Parsec: A Practical Parser Library*. Electronic Notes in Theoretical Computer Science 41(1), 2001.
- A. Gill, S. Marlow. *Happy – The Parser Generator for Haskell*. University of Glasgow, 1995.
  `http://www.haskell.org/happy`

Next Course Meeting...

- Thu, May 28, 2009, lecture time: 4.15 p.m. to 5.45 p.m., lecture room on the ground floor of the building Argentinierstr. 8
- No meeting tomorrow!