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...

---

Parsing – Implementation Variants

Two variants...
- **Combinator parsing**
  - *recursive descent parsing*
- **Monadic parsing**

---

Reference

The following presentation is based on...
- Chapter 17
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...

```haskell
type Parse a b = [a] -> [(b,[a])] -- Parser Input Expected Output

-- 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.

```
Remark:
• The capability of delivering multiple results enables the analysis of ambiguous grammars

    ∼ list of successes technique

• Each element in the output list represents a successful parse.
```

Basic Parsers 1(3)

Primitive, input-independent parsing functions

- The always failing parsing function

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

- The always successful parsing function

  ```haskell
  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)...

  ```haskell
  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...

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

Remark:
- 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 is the input not yet considered.

Advanced functional Programming (SS 2008) / Part 6 (Thu, 06/05/08)
Basic Parsers 3(3)

Application:

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

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

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

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

Combining Parsers 1(4)

...to obtain (more) complex parsing functions

\(\sim\) Combinator Parsing

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

- Alternatives

```haskell
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:

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

\(\sim\) ...the alt parser combines the results of the parses given by parsers p1 and p2

Combining Parsers 2(4)

- Sequential composition of parsers

```haskell
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

Example:

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

```haskell
(number >>= bracket) "24"
```

```haskell
-> [(y,z),rem2) | (y,rem1) <- [(2,"4"), (24,""),
(z,rem2) <- bracket rem1 ]
```

```haskell
-> [(2,z),rem2) | (z,rem2) <- bracket "4" ] ++
[(24,z),rem2) | (z,rem2) <- bracket "" ]
```

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

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

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

```haskell
-> [ ((24,'('), "" ) ]
```

Combining Parsers 3(4)

Example:

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

```haskell
(number >>= bracket) "24("
```

```haskell
-> [(y,z),rem2) | (y,rem1) <- [(2,"4("), (24,"")],
(z,rem2) <- bracket rem1 ]
```

```haskell
-> [(2,z),rem2) | (z,rem2) <- bracket "4(" ] ++
[(24,z),rem2) | (z,rem2) <- bracket "" ]
```

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

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

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

```haskell
-> [ ((24,'('), "" ) ]
```
Combining Parsers 4(4)

• Transformation/Modification
  ~ change the item returned by the parser, or build something from it...

  \[
  \text{build} :: \text{Parse } a \to (b \to c) \to \text{Parse } a c
  \]

  \[
  \text{build } p \ f \ \text{inp} = [ (f \ x, \ \text{rem}) | (x,\text{rem}) <\!\!\!< p \ \text{inp} ]
  \]

Example:

  \[
  (\text{digList } '\text{build}' \ \text{digsToNum}) '21a3'
  \]

  \[
  \to [ (\text{digsToNum } x,\text{rem}) | (x,\text{rem}) <\!\!\!< \text{digList } '21a3' ]
  \]

  \[
  \to [ (\text{digsToNum } '2', '1a3') , (\text{digsToNum } '21', 'a3') ]
  \]

  \[
  \to [ (2,'1a3') , (21,'a3') ]
  \]

The Clou

The combinators

• \textit{alt}
• \textit{>>=}
• \textit{build}

The combinators \textit{alt} and \textit{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:

  \[
  \text{list} :: \text{Parse } a \rightarrow \text{Parse } [b]
  \]

  \[
  \text{list } p = (\text{succeed } [] '\text{alt}'}
  \]

  \[
  (p \gg= \text{list } p) '\text{build}' \ (\text{uncurry } (::))
  \]

Intuition:

• A list can be empty.
  ~ ...recognized by the parser succeed []

• A list can be non-empty, i.e., it consists of an object followed by a list of objects.
  ~ ...recognized by the combined parser \text{p} \gg= \text{list } p, where we use \text{build} to turn a pair \((x,\text{xs})\) into the list \((x:\text{xs})\).

Summary and Conclusion

...about combining parsers (\textit{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 \textit{combinating forms}, or, as a short hand, \textit{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 :: a -> Parse a b
succeed val inp = [(val,inp)]

Overview of the Parsing Functions 2(4)

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

-- Recognizing single objects satisfying a particular property
spot :: (a -> Bool) -> Parse a b
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 [b]
list p = (succeed []) 'alt'
  ((p >> list p) 'build' (uncurry ()))
**Application: Back to the Initial Example**

We consider expressions of the form...

```haskell
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...

```haskell
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**

```haskell
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

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

**Part III: Parsing literals (numerals)**

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

---

**A Parser for Expressions 3(3)**

Note that a number of supporting functions used such as...

- **isOp**
- **charlistToExpr**
- ...

are yet to be defined (~ exercise).
The Top-level Parser

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,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.

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...

\[
\text{item} :: \text{Parser Char}
\text{item} = \text{Parser} \ (\cs \to \text{case } \cs \text{ of} \ \"
\text{\textbackslash s} \to [] \ \ (c:cs) \to [(c,cs)] \)
\]

Compare: item vs. token

Properties of return and (\texttt{>=>})

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

\[
\begin{align*}
\text{return } a \text{ } \texttt{>=>} f & = f \ a \\
\text{p } \texttt{>=>} \text{ return } & = p \\
\text{p } \texttt{>=>} (\text{f } a \text{ } \texttt{>=>} g) & = (\text{p } \texttt{>=>} (\text{f } a \text{ } \texttt{>=>} g)) \text{ } \texttt{>=>} g
\end{align*}
\]

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 (\texttt{>=>})
  - …allows a simpler and more concise definition of some parsers
  - …(\texttt{>=>}) is associative
  - …allows suppression of parentheses when parsers are applied sequentially

The Parser Monad

Reminder: The class monad...

\[
\begin{align*}
\text{class Monad m where} \\
\text{return} & : a \to m a \\
\text{(\texttt{>=>})} & : m a \to (a \to m b) \to m b
\end{align*}
\]

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

\[
\begin{align*}
\text{instance Monad Parser where} \\
\text{return} a & = \text{Parser} \ (\cs \to [(a,cs)]) \\
\text{-- Sequences} \\
\text{p } \texttt{>=>} f & = \text{Parser} \ (\cs \to \text{concat } [\text{parse } (f \ a) \ cs' \mid \ (a,cs') \leftarrow \text{parse } p \ cs])
\end{align*}
\]

Compare: return vs. succeed and (\texttt{>=>}) vs. infixr

Typical Structure of a Parser 1(2)

...using the operator (\texttt{>=>})

\[
\begin{align*}
p_1 & \text{ } \texttt{>=>} \ \text{\textbackslash a1} \to \\
p_2 & \text{ } \texttt{>=>} \ \text{\textbackslash a2} \to \\
\ldots \\
p_n & \text{ } \texttt{>=>} \ \text{\textbackslash an} \\
f \ a1 \ a2 \ldots \ an
\end{align*}
\]

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

- $a_i \leftarrow p_i$ are called *generators* (since they generate values for the variables $a_i$)

**Remark**:

A generator of the form $a_i \leftarrow p_i$ can be

- replaced by $p_i$, 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 into instances of these two classes as follows:

- The parser which always fails...
  
  ```haskell```
  ```
  instance MonadZero Parser where
  zero = Parser (\cs -> [])
  ```
  ```
  ```

- The parser which non-deterministically selects...
  
  ```haskell```
  ```
  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 >>= (\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...

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

- single objects
  
  \[
  \text{char :: Char -> Parser Char} \\
  \text{char c = sat (c ==)}
  \]

- 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
  
  \[
  \text{string :: String -> Parser String} \\
  \text{string "" = return ""} \\
  \text{string (c:cs) = do \{char c; string cs; return (c:cs)\}}
  \]

- Parse repeated applications of a parser \( p \)
  
  \[
  \text{many :: Parser a -> Parser [a]} \\
  \text{many p = many1 p +++ return []} \\
  \text{many1 :: Parser a -> Parser [a]} \\
  \text{many1 p = do \{a <- p; as <- many p; return (a:as)\}}
  \]

Recursion Combinators 2(3)

- Similar to the parser \( \text{many} \) but with interspersed applications of the parser \( \text{sep} \), whose result values are thrown away
  
  \[
  \text{sepby :: Parser a -> Parser b -> Parser [a]} \\
  \text{p 'sepby' sep = (p 'sepby1' sep) +++ return []}
  \]

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

- Parsing of a symbol token
  
  ```
  symb :: String -> Parser Stringsymb 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 ::= + | -
mulp ::= * | /
```

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)
mulp :: Parser (Int -> Int -> Int)

expr = term 'chainl1' addop

term = factor 'chainl1' mulp

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

Example: Parsing of Expressions 3(3)

`Example`:

Evaluating

```java
apply expr " 1 - 2 * 3 + 4 "
gives the singleton list

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

as desired.
Further Readings 1(3)

On combinator parsing...


Further Readings 2(3)

On error-correcting parsing...


Further Readings 3(3)

On parser libraries...


Next lecture...

- Thu, June 12, 2008, lecture time: 4.15 p.m. to 5.45 p.m., lecture room on the ground floor of the building Argentinierstr. 8

Sixth assignment (as well as previous assignments)...

- Please check out the homepage of the course for details.