



Advanced Functional Programming

Week 7 – Parsing in General, Parsec

René Thiemann

Department of Computer Science

Parsing in General

Last Week

- evaluation of monadic code heavily depends on underlying monad
 - example 1: difference in strictness of ;
 - example 2: in some monads consecutive ; might result in nested loops
- combining multiple `State`-monads using datatypes with record syntax
- combination of monads using the `RWS`-monad
- example application: Tseitin
- error monads, `MonadFail` and irrefutable patterns

RT (DCS @ UIBK)

Week 7

2/31

Towards Parsing of Context Free Languages

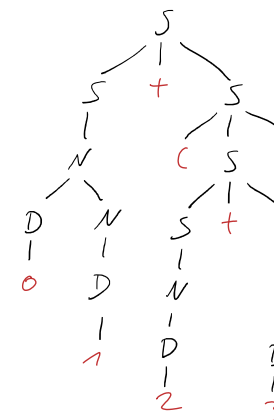
- languages can be described by formal grammars
- most basic version: **context free grammars** (CFG)
- $G = (V, \Sigma, R, S)$ is a CFG where
 - V is a finite set of non-terminal symbols
 - Σ is a finite set of terminal symbols
 - R is a finite set of rules of the form $A \rightarrow w$ with $A \in V$ and $w \in (V \cup \Sigma)^*$
 - S is the starting symbol
- in examples we often just indicate the rules of a grammar; moreover we write $A \rightarrow w_1 \mid w_2 \mid \dots$ to indicate rules $A \rightarrow w_1, A \rightarrow w_2, \dots$
- example: $G = \{S \rightarrow (S) \mid S + S \mid N, \quad N \rightarrow DN \mid D, \quad D \rightarrow 0 \mid \dots \mid 9\}$
 - implicit $V = \{S, N, D\}$
 - implicit $\Sigma = \{0, \dots, 9, +, (,)\}$
 - D generates digits
 - N generates natural numbers
 - S generates arithmetic expressions involving numbers, additions, and parenthesis

Context Free Languages and Syntax Trees

- given a CFG $G = (V, \Sigma, R, S)$ a **syntax tree** t of G has all of the following properties
 - the root of t is S
 - for every subtree u of t with root $A \in V$ there is a rule $A \rightarrow w \in R$ such that u has $|w|$ many subtrees and the roots of these subtrees are exactly w
 - every subtree u of t with root $a \in \Sigma$ is a leaf
- a syntax tree produces the word that is obtained when traversing the terminal symbols from left to right
- $L(G) \subseteq \Sigma^*$ is the **language** of the grammar; it consists of those words that are produced by the set of all syntax trees of G
- a **language** L' is **context free** if there is some CFG G such that $L' = L(G)$

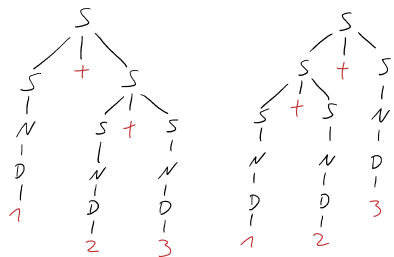
Example: Syntax Tree

- consider $G = \{S \rightarrow (S) \mid S + S \mid N, \quad N \rightarrow DN \mid D, \quad D \rightarrow 0 \mid \dots \mid 9\}$ from before
- the following syntax tree proves $01 + (2 + 3) \in L(G)$



Ambiguity

- consider $G = \{S \rightarrow (S) \mid S + S \mid N, \quad N \rightarrow DN \mid D, \quad D \rightarrow 0 \mid \dots \mid 9\}$ from before
- there is only 1 syntax tree that produces $01 + (2 + 3)$, but in general there might be several syntax trees for the same word
- example: there are two syntax trees for $1 + 2 + 3$



- if there are multiple syntax trees for some word, then G is **ambiguous**
- another example of ambiguity: if b then if c then x++ else y++

Some Facts About CFGs

- given w and CFG G , the question $w \in L(G)$ is decidable in cubic time (CYK algorithm)
- sometimes ambiguous CFGs can be transformed into language-equivalent non-ambiguous CFGs
 - example grammar G from before is equivalent to the following non-ambiguous grammar G'
 - G' copies rules for N and D from G
 - G' also has rules $\{S \rightarrow T + S \mid T, \quad T \rightarrow (S) \mid N\}$
- given CFGs G and G' , it is undecidable whether $L(G) = L(G')$
- given CFG G , it is undecidable whether G is ambiguous
- there are inherently ambiguous context free languages where no non-ambiguous CFG exists
 - $L' = \{a^n b^n c^m d^m \mid n, m > 0\} \cup \{a^n b^m c^m d^n \mid n, m > 0\}$ is context free
 - if $L(G) = L'$ and $n > 0$ then the word $a^n b^n c^n d^n$ has at least two syntax trees in G
- further literature: Hopcraft, Ullman: Introduction to Automata Theory, Languages and Computation

Equivalence of G and G'

- $G = \{S \rightarrow (S) \mid S + S \mid N, \quad N \rightarrow DN \mid D, \quad D \rightarrow 0 \mid \dots \mid 9\}$
- $G' = \{S \rightarrow T + S \mid T, \quad T \rightarrow (S) \mid N, \quad N \rightarrow DN \mid D, \quad D \rightarrow 0 \mid \dots \mid 9\}$
- equivalence is proved in two steps
- we write $L_G(A)$ for the language produced by CFG G where the start symbol is replaced by A
 - $L_{G'}(S) \cup L_{G'}(T) \subseteq L_G(S)$: by induction on the size of the syntax tree
 - $L_G(S) \subseteq L_{G'}(S)$: by induction on the following size of the syntax tree t where $size(S \rightarrow S_1 + S_2) = 1 + 2 \cdot size(S_1) + size(S_2)$, $size(S \rightarrow (S_1)) = 1 + size(S_1)$, and the size of all other trees is 1
 - if t uses $S \rightarrow N \in G'$ at the root, then $S \rightarrow N \in G'$ is also possible
 - if t uses $S \rightarrow (S_1) \in G'$ at the root, then we can simulate it by $S \rightarrow T \rightarrow (S_1) \in G'$ and then apply the IH for the tree with root S_1
 - if t uses $S \rightarrow S_1 + S_2 \in G$ at the root and S_1 is continued by $S_1 \rightarrow N \in G$, then we simulate it by $S \rightarrow T + S_2 \rightarrow N + S_2 \in G'$ and apply the IH for the tree with root S_2
 - if t uses $S \rightarrow S_1 + S_2 \in G$ at the root and S_1 is continued by $S_1 \rightarrow (S_3) \in G$, then we simulate it by $S \rightarrow T + S_2 \rightarrow (S_3) + S_2 \in G'$ and apply IHs for S_2 and S_3
 - if t uses $S \rightarrow S_1 + S_2 \in G$ at the root and S_1 is continued by $S_1 \rightarrow S_3 + S_4 \in G$, then we rotate the tree to $S \rightarrow S_3 + S_5$ where $S_5 \rightarrow S_4 + S_2$, and apply the IH

Parsing of CFGs

- general task: given a CFG and some input word w
 - return the unique syntax tree that generates w
 - or report that this is not possible (none or more than one syntax trees)
- problems and challenges
 - efficiency: general case has at least cubic complexity, but one wants to have linear algorithm; often: traverse the input from left to right exactly once
 - more expressive forms of grammars are welcome: transforming G to G' for getting non-ambiguity is tedious and not very readable
 - Backus-Naur-Form (BNF) is more concise than CFG
 - use grammars such as G and specify priorities and left/right associativity of operators
 - full syntax tree is often too verbose
 - drop $S \rightarrow (S)$ and $S \rightarrow T$ in G'
 - collapse N subtrees to a single number
 - in general: provide not only grammar specification, but also result specification
- detailed and helpful error reporting

Approaches to Getting a Parser

- use parser generators (in Haskell: happy)
 - disadvantage
 - might require to specify grammars in specific shape (LL(1), LALR(1))
 - error reporting requires technical knowledge (resolve shift-reduce conflict, ...)
 - fixed feature set
 - advantages
 - static checks on grammar
 - guaranteed linear time
 - take care of user error messages in generated parser
- use parser combinators (in Haskell: Parsec)
 - disadvantages
 - less automation
 - might become inefficient, no static checks
 - advantages
 - no formal specification of an input grammar required: the code is the spec
 - many building blocks that simplify the task of writing a parser and reading it
 - full flexibility of the programming language (arbitrary features)
 - adjustment of parsing possible on the fly, e.g., when reading new infix-operator from user
 - easy to control generated output

Parsing in Phases: Lexical Analysis

- parsing can be performed in two phases: **lexical analysis** (lexing, **tokenization**) and parsing
- lexical analysis is often done using regular languages
- purpose of tokenization: simplify latter parsing phase
- examples
 - simplify $G = \{S \rightarrow (S) \mid S + S \mid N, \quad N \rightarrow DN \mid D, \quad D \rightarrow 0 \mid \dots \mid 9\}$ to $G' = \{S \rightarrow (S) \mid S + S \mid \text{number}\}$ where tokenization converts list of digits into single **number** token (with integer stored inside)
 - tokenization can **remove all comments** and can **take care of whitespace**
 - tokenization can **identify keywords** and distinguish them from standard names
 - example: tokenizer might convert string

```
"if someBool then foo else 832"
```

into token list

```
[KeywIf, Name "someBool", KeywThen, Name "foo", KeywElse, Number 832]
```

- tool examples: flex does lexical analysis and bison does parsing

Parsec

Parsec

- **Parsec** is a Haskell library for parsing based on parser combinators
- it can be used both to write single phase parsers, but also supports many phase parsing
- Parsec has been used in other projects, e.g., to write parsers for CSV, JSON and bibtex
- documentation: <https://hackage.haskell.org/package/parsec>
- alternatives to obtain parsers in Haskell that are not (further) discussed in this course
 - use parser generator such as Happy
 - use alternative parser combinator library such as Attoparsec
 - use advanced fork of Parsec such as Megaparsec
 - don't use any library, e.g., for parsing raw PGM files
- **parser combinators**: assemble complex parsers from simpler ones via combinators

Important Types in Parsec

- `type Parsec s u a = ...`
 - `Parsec s u` is an instance of `MonadFail`
 - `s` is the type of input stream, e.g., `ByteString`, `String`, `[Token]`, ...
 - `u` is the user state type
 - Parsec has its own state, e.g., to keep track of position in input
 - `u` can be used as an additional state that is under user control; often `u = ()`
 - `a` is return type upon successful parsing, e.g. `Int`, `String`, `Expr`, `AbstractSyntaxTree`
- `type GenParser t u = Parsec [t] u` parsing from a token list `[t]` with user state `u`
- `type Parser = GenParser Char ()` parsing from a string without user state
- `data ParseError = ...` type to encapsulate error, instance of `Show`
- `type SourceName = String`
- running a parser, where `s` needs to be stream type

```
parse :: Parsec s () a -> SourceName -> s -> Either ParseError a
parse :: Parser      a -> SourceName -> String -> Either ParseError a
runParser :: GenParser t u a -> u -> SourceName -> [t] -> -||-
runParser :: GenParser Char u a -> u -> SourceName -> String -> -||-
```

First Example: Parsing CSV Files

- CSV = comma separated values
- heavily used for importing and exporting data of spread sheets
- CSV file is ASCII file
 - each line represents one row in table, and must be terminated by end-of-line
 - each line consists of cells that are separated by commas (,)
 - special treatment for cells whose content contains comma
- example content of CSV file

```
name,matrikel number,skz,email
max m.,123456,521,max@uibk.at
nina n.,654321,921,nina@uibk.at
junior,,junior@school.at
```
- we will develop several versions of parsers for CSV, first ignoring cells with comma
- note: input to `parse` is `String`, getting file content must be done separately

First Version (Demo07_Parser_CSV_V1)

```
csv :: Parser [[String]]
csv = do
  result <- many line
  eof >> return result

line :: Parser [String]
line = do
  result <- cells
  eol >> return result

cells :: Parser [String]
cells = do
  firstC <- cellContent
  nextC <- remainingCells
  return $ firstC : nextC

remainingCells :: Parser [String]
remainingCells =
  (char ',' >> cells)
  <|> return []

cellContent :: Parser String
cellContent = many (noneOf ",\n")

eol :: Parser ()
eol = char '\n' >> return ()

parseCSV :: String ->
  Either ParseError [[String]]
parseCSV input =
  parse csv "(unknown)" input
```

RT (DCS @ UIBK)

Week 7

17/31

Explanations

- when applying a parser there are three different outcomes
 - **Ok**: parser succeeds (and may have consumed some parts of the input)
 - **F0**: parser failed, and did not consume input
 - **F+**: parser failed, and consumed at least one token from the input
- **Ok** and **F0** often appear during successful parsing
 - parsers resulting in **F0** indicate that they are not applicable
 - useful when handling alternatives
- **F+** usually results in overall parse error, alternatives are not tried
- note: names **Ok**, **F0**, **F+** are not official

RT (DCS @ UIBK)

Week 7

18/31

Explanations

- **many** :: **GenParser** **t u a** -> **GenParser** **t u** [**a**]
 - **many p** applies **p** iteratively while **p** results in **Ok**
 - if final result of **p** is **F0**, then **many p** results in **Ok** and returns parsed elements as list
 - if final result of **p** is **F+**, then **many p** results in **F+**
- **eof** :: **Show** **t** => **GenParser** **t u** ()
 - returns **Ok**, if and only if the input stream has been fully consumed, otherwise **F0**
- **noneOf** :: [**Char**] -> **GenParser** **Char u** **Char**
 - **noneOf f** reads the next character from the input
 - returns **Ok**, if and only this character is not among the forbidden characters **f**
 - otherwise, result is **F0**
- **char** :: **Char** -> **GenParser** **Char u** **Char**
 - similar to **noneOf**, except that one provides exactly the character that is expected
- (**<|>**) :: **GenParser** **t u a** -> **GenParser** **t u a** -> **GenParser** **t u a**
 - **p1 <|> p2** first tries **p1**
 - if **p1** returns **Ok**, then this will be the result of **p1 <|> p2**
 - if **p1** returns **F0**, then **p2** is executed and that result is returned
 - if **p1** returns **F+**, then the result is **F+** and **p2** is not tried

RT (DCS @ UIBK)

Week 7

19/31

Example Invocations

- **parseCSV** "Hello,Parsec\n"
Right [["Hello","Parsec"]]
- **parseCSV** "a,,b\n\nc,d\n"
Right [["a","","b"],[""],["c","d"]]
- **parseCSV** "Hello,Parsec"
Left "(unknown)" (line 1, column 13):
unexpected end of input
expecting ",", or "\n"
- first examples illustrate correct behavior on sample CSV strings
- last example shows that we get useful error messages by using existing framework

RT (DCS @ UIBK)

Week 7

20/31

Towards Tuning the Parser: `sepBy`

- upcoming: write more concise parsers by using further combinators
- `sepBy :: GenParser t u a -> GenParser t u sep -> GenParser t u [a]`
 - `sepBy p1 p2` parses zero or more elements of type `a` via `p1` that are separated via `p2`
 - if any invocation of `p1` and `p2` result in `F+`, then the result will be `F+`
 - first `p1` is invoked to parse the first element
 - if this first invocation results in `F0`, then `[]` is returned with `Ok`
 - otherwise, then alternating, a separator and a next element is parsed until separator parser delivers `F0`, and the gathered elements are returned by `sepBy`
 - if during this process `p1` results in `F0`, then `sepBy p1 p2` fails with `F+`
- examples for `pSep = parse (sepBy (noneOf ",c") (char ',')) "unk"`
 - `pSep "b"` succeeds and returns `"b"`
 - `pSep "ba"` succeeds and returns `"b"`
 - `pSep "c"` succeeds and returns `" "`
 - `pSep "b,a,d,e"` succeeds and returns `"bade"`
 - `pSep "b,a,"` fails
 - `pSep "b,a,c"` fails

Towards Tuning the Parser: `endBy`

- `endBy` is similar to `sepBy`, but separators are at the end
 - same type, takes element parser and separator parser
 - iteratively parses `p1` and `p2` in sequence, until `p1` results in `F0`
 - all gathered elements will be returned
 - if during this process `p2` results in `F0`, then `endBy p1 p2` results in `F+`
- examples for `pEnd = parse (endBy (noneOf ".c") (char '.')) "unk"`
 - `pEnd "b"` fails
 - `pEnd "bb"` fails
 - `pEnd "b."` succeeds and returns `"b"`
 - `pEnd "b. ."` succeeds and returns `"b"`
 - `pEnd "b.b"` fails
 - `pEnd "c"` succeeds and returns `" "`
 - `pEnd "b.a.d.e."` succeeds and returns `"bade"`

A More Concise Parser

```
csv = endBy line eol
eol = char '\n'
line = sepBy cell (char ',')
cell = many (noneOf ",\n")
```

```
parseCSV :: String -> Either ParseError [[String]]
parseCSV input = parse csv "(unknown)" input
```

- parser definition can be read as specification of CSV
- no formal grammar required

Extending the Parsing of EOL

- currently: `eol = char '\n'`
- problem: depending on OS, end-of-line might also be `"\n\r"`
- extended primitive of `char`: `string :: String -> GenParser Char u String`
- behavior: `string s` either fully consumes `s` and results in `Ok`, or consumes as many chars until a mismatch is detected and then results in `F+` or `F0`
- attempt 1: `eol = string "\n" <|> string "\n\r"`
 - problem: `parse (eol >> eof) "(unknown)" "\n\r"` fails
 - reason: only `"\n"` is consumed
- attempt 2: `eol = string "\n\r" <|> string "\n"`
 - problem: `parse (eol >> eof) "(unknown)" "\n"` fails
 - reason: `string "\n\r"` results in `F+`
- **lookahead task**: peek at the upcoming symbol(s) without consuming them
- Parsec's mechanism for lookahead will be explained on next slides
- solution without this mechanism

```
eol = char '\n' >> (char '\r' <|> return '\n') >> return ()
```

Try

- situation: parser might fail, but still consume some input
 - running `string "Hello"` on input `"Hellas is a name for Greece"` will lead to `F+` state with the remaining input `"as is a name for Greece"`
- solution: `try :: GenParser t u a -> GenParser t u a`
 - if `p` succeeds, then `try p` succeeds
 - if `p` fails, then `try p` fails with `F0`, and the parsing state is modified in such a way as if `p` did not consume any input at all
- consequence: `try (string "Hello")` either succeeds or results in `F0`
- usually `try` is used on left-hand sides of `<|>`
 - there are exceptions, since some functions might use `<|>` internally
- improved parser for end of line

```
eol = (try (string "\n\r")
  <|> try (string "\r\n")
  <|> string "\n"      -- for single char parsers, try has no effect
  <|> string "\r") >> return ()
```

Error Handling: fail

- situation: parser can accept multiple line endings
 - `parseCSV "line1\r\nline2\nline3\n\rline4\rline5\n"`
Right `[["line1"],["line2"],["line3"],["line4"],["line5"]]`
- error message are not optimal: too low level
 - `parseCSV "line1"`
Left "(unknown)" (line 1, column 6):
unexpected end of input
expecting ",", "\n\r", "\r\n", "\n" or "\r"
- since `Parsec s u` is an instance of `MonadFail` we may use `fail "message"`

```
eol = (try (string "\n\r")
  <|> try (string "\r\n")
  <|> string "\n"
  <|> string "\r"
  <|> fail "Couldn't find EOL") >> return ()
```
- problem: error message is just added when using `fail`

Error Handling: <?> (Demo07_Parser_CSV_V2)

- error message are still not optimal
 - `parseCSV "line1"`
Left "(unknown)" (line 1, column 6):
unexpected end of input
expecting ",", "\n\r", "\r\n", "\n" or "\r"
Could not find EOL
- solution: `(<?>) :: GenParser t u a -> String -> GenParser t u a`
 - `p <?> msg` is similar to `p <|> fail msg`
 - difference: if `p` returns `F0`, then `msg` is used as high-level error message

```
eol = (try (string "\n\r") <|> try (string "\r\n")
  <|> string "\n" <|> string "\r"
  <?> "end of line") >> return ()
```
- `parseCSV "line1"`
Left "(unknown)" (line 1, column 6):
unexpected end of input
expecting ",", " or end of line

Extended Example: Full CSV

- CSV cells might also contain commas
- standard solution: put double quotation marks around cells
- next problem: how to handle double quotation marks
- standard solution: use double double quotation marks
 - example CSV file

```
Ralph,"chess, reading and swimming",18
John Michael "Ozzy" Osbourne,music,??
some,"""easy""", nice exercise,"hello
world"
```
 - expected output of `parseCSV` on this input

```
Right [
  ["Ralph","chess, reading and swimming","18"],
  ["John Michael \"Ozzy\" Osbourne","music","??"],
  ["some","\"easy\"", nice exercise,"hello\nworld"]
]
```

Extended Parser (Demo07_Parser_CSV_V3)

- only one change required in parser: the `cell` parser
- previous solution: `cell = many (noneOf ",\n")`
- new, extended cell parser
`cell = quotedCell <|> many (noneOf ",\n")`

```
quotedCell =  
  do _ <- char '"'  
      content <- many quotedChar  
      _ <- char '"' <?> "missing closing quote at end of cell"  
      return content  
  
quotedChar =  
  noneOf "\""  
  <|> try (string "\"\"") >> return ''
```

- note: `try` on rhs of `<|>`; usage required, since `quotedChar` is used inside `many`

Literature

- Real World Haskell, Chapter 16

Overview: Selection of Primitives and Combinators

- `space` (or `spaces`): parse a (or many) white space
- `char c`: parse the single character `c` (`Ok` or `F0`)
- `noneOf bad`: parse any character that is not forbidden (`Ok` or `F0`)
- `oneOf good`: parse any allowed character (`Ok` or `F0`)
- `string s`: parse the given string `s` (`Ok` or `F0` or `F+`)
- `many p`: apply `p` as often as possible until `p` delivers `F0`
- `many1 p`: apply `p` as often as possible, but at least once
- `between pOpen pClose p`: parses `pOpen p pClose` in sequence, returns result of `p`
- `p1 <|> p2`: apply `p1` first; if that fails with `F0`, apply `p2`
- `p1 <?> msg`: drop potential error of `p1` in `p1 <|> fail msg`
- `choice [p1, ..., pn]`: same as `p1 <|> ... <|> pn`
- `eof`: check whether input has completely been consumed
- `try p`: if `p` fails, restore the consumed input of `p` and fail with `F0`
- <https://hackage.haskell.org/package/parsec/docs/Text-Parsec-Char.html>
- <https://hackage.haskell.org/package/parsec/docs/Text-Parsec-Combinator.html>