universität innsbruck

WS 2024/2025

Advanced Functional Programming

Week 7 – Parsing in General, Parsec

René Thiemann

Department of Computer Science

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

Parsing in General

Towards Parsing of Context Free Languages

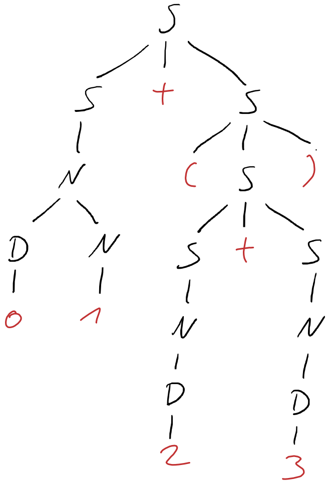
- languages can be described by formal grammars
- most basic version: context free grammars (CFG)
- $G = (V, \Sigma, R, S)$ is a CFG where
  - V is a finite set of non-terminal symbols
  - Σ is a finite set of terminal symbols
  - R is a finite set of rules of the form $A \to w$ with $A \in V$ and $w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

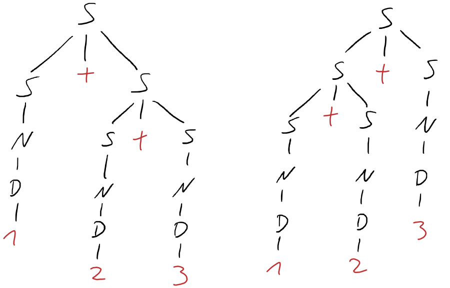
Example: Syntax Tree

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



Ambiguity

- consider $G = \{S \to (S) \mid S + S \mid N, \quad N \to DN \mid D, \quad D \to 0 \mid \cdots \mid 9\}$ 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 $\{S \to T + S \mid T, \quad T \to (S) \mid N\}$
- 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'

- $G = \{S \to (S) \mid S + S \mid N, \quad N \to DN \mid D, \quad D \to 0 \mid \cdots \mid 9\}$
- $G' = \{S \to T + S \mid T, \quad T \to (S) \mid N, \quad N \to DN \mid D, \quad D \to 0 \mid \cdots \mid 9\}$
- 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 $size(S \to S_1 + S_2) = 1 + size(S_1) + 2 \cdot size(S_2)$, and the size of all other trees is 1
    - if t uses $S \to N \in G$ at the root, then $S \to N \in G'$ is also possible
    - if t uses $S \to (S_1) \in G$ at the root, then we can simulate it by $S \to T \to (S_1) \in G'$ and then apply the IH for the tree with root S_1
    - if t uses $S \to S_1 + S_2 \in G$ at the root and S_1 is continued by $S_1 \to N \in G$, then we simulate it by $S \to T + S_2 \to N + S_2 \in G'$ and apply the IH for the tree with root S_2
    - if t uses $S \to S_1 + S_2 \in G$ at the root and S_1 is continued by $S_1 \to (S_3) \in G$, then we simulate it by $S \to T + S_2 \to (S_3) + S_2 \in G'$ and apply IHs for S_2 and S_3
    - if t uses $S \to S_1 + S_2 \in G$ at the root and S_1 is continued by $S_1 \to S_3 + S_4 \in G$, then we rotate the tree to $S \to S_3 + S_5$ where $S_5 \to S_4 + S_2$, 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 $S \to (S)$ and $S \to T$ 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 $G = \{G = \{S \to (S) \mid S + S \mid N, \quad N \to DN \mid D, \quad D \to 0 \mid \cdots \mid 9\}$ to $G' = \{S \to (S) \mid S + S \mid number\}$ 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 then 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
    - initially: choose no user state, so `u = ()`
  - `a` is return type upon successful parsing, e.g. `Int`, `String`, `Expr`, `AbstractSyntaxTree`
- `type Parser = Parsec String ()` parsing from a string without user state
- `type GenParser tok u = Parsec [tok] u` parsing from a token list with user state `u`
- `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
```

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)

```haskell
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
```

```haskell
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
```

Explanations

- `many :: GenParser tok u a -> GenParser tok u [a]`
  - `many p` applies `p` iteratively, until it fails
  - `many p` always succeeds, results are stored and returned as list
- `eof :: Show tok => GenParser tok u ()`
  - successful, if and only if the input stream has been fully consumed
- `noneOf :: [Char] -> GenParser Char u Char`
  - `noneOf f` reads the next character from the input
  - successful, if and only this character is not among the forbidden characters `f`
  - on failure, no character is consumed
- `char :: Char -> GenParser Char u Char`
  - similar to `noneOf`, except that one provides exactly the character that is expected
- `(<|>) :: Parsec s u a -> Parsec s u a -> Parsec s u a`
  - `p1 <|> p2` first tries `p1`
  - if `p1` is successful, then the result of `p1` is returned
  - otherwise, `p2` is executed and that result is returned
  - `p1` should not consume input if it fails (will be discussed later); hint:
    `parse ((many (noneOf "ab") >> char 'a') <|> char 'c') "file" "ceeeb"` fails

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

Towards Tuning the Parser: `sepBy`

- upcoming: write more concise parsers by using further combinators
- `sepBy :: Parsec s u a -> Parsec s u sep -> Parsec s u [a]`
  - `sepBy p1 p2` takes a parser `p1` for elements of type `a` and a parser `p2` for separators `:: sep`
  - first `p1` is invoked to parse the first element
    - if this first invocation fails, then `[]` is returned
  - otherwise, alternating, a separator and a next element is parsed until no separator is occurring any more, and the gathered elements are returned
  - if during this process `p1` fails, then also `sepBy p1 p2` fails
- 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`
  - same type, takes element parser and separator parser
  - iteratively parses `p1` and `p2` in sequence, until `p1` fails
  - all gathered elements will be returned
  - if during this process `p2` fails, then also `endBy p1 p2` fails
- examples for `pEnd = parse (sepBy (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`
- try 1: `eol = string "\n" <|> string "\n\r"`
  - problem: `parse (eol >> eof) "(unknown)" "\n\r"` fails
  - reason: only `"\n"` is consumed
- try 2: `eol = string "\n\r" <|> string "\n"`
  - problem: `parse (eol >> eof) "(unknown)" "\n"` fails
  - reason: `"\n"` is consumed while trying parser `string "\n\r"`
- 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 failing state with the remaining input `"as is a name for Greece"`
- solution: `try :: GenParser tok u a -> GenParser tok u a`
  - if `p` succeeds, then `try p` succeeds
  - if `p` fails, then `try p` fails, but 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 does not modify the input
- 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: `(<?>) :: Parsec s u a -> String -> Parsec s u a`
  - `p <?> msg` is similar to `p <|> fail msg`
  - difference: if `p` fails and does not consume input, 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`

Overview of Primitives and Combinators

- `space` (or `spaces`): parse a (or many) white space
- `char c`: parse the single character `c`
- `noneOf bad`: parse any character that is not forbidden
- `oneOf good`: parse any allowed character
- `string s`: parse the given string `s` (beware of partial consumption)
- `many p`: apply `p` as often as possible
- `many1 p`: apply `p` as often as possible, but at least once
- `between pOpen p pClose`: applies all three parsers in sequence, returns result of `p`
- `p1 <|> p2`: apply `p1` first; if that fails, 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`
- https://hackage.haskell.org/package/parsec/docs/Text-Parsec-Char.html
- https://hackage.haskell.org/package/parsec/docs/Text-Parsec-Combinator.html

Exercises

- develop a parser for the ARI format
- see `Exercise07.hs` for further details
- see https://project-coco.uibk.ac.at/ARI/ and https://project-coco.uibk.ac.at/ARI/trs.php for the format
- example

```
; @author some one
; @author another one
; @origin some location
; just a comment
(format TRS)
(fun + 2)
(fun 0 0)
(fun s 1)
(rule (+ x 0) x)
(rule (+ x (s y)) (s (+ x y)))
```

Literature

- Real World Haskell, Chapter 16