Programming Language Technology

Putting Formal Languages to Work

Andreas Abel

Department of Computer Science and Engineering
Chalmers and Gothenburg University

Finite Automata Theory and Formal Languages
TMV027/DIT321, LP4 2017
11 May 2017

This Lecture: a Taste of PLT

- Lecture material: `http://www.cse.chalmers.se/~abela/`
- A taste of an application of formal languages and automata

  Programming Language Technology
- Parsing, type-checking, interpretation, compilation
- DAT151 / DIT231
- Next edition: 2017/2018 LP2 (November-Jan)

Task: Implement Calculator With Variables

```
$ Calc
1+2
==> 3
1+2*3+4*5
==> 27
x where x = 1
==> 1
(x*y where x=2) where y=3
==> 6
x where x = y where y=42
==> 42
x where x = x
==> Calc: undefined variable x
```

Calculator Master Plan

- Read string from stdin.
- Parse input.
- Calculate result.
- Print result to stdout.
- Start over.

Parsing

- latin / old french *pars* = part(s) (of speech)
- A parser for a formal language
  1. takes input stream of characters
  2. checks if input forms word of language
  3. outputs typically one of:
     - Yes/no (*accepting* parser).
     - Parse tree.
     - Abstract syntax tree.
     - Result of interpreting input (e.g. for our calculator).
- You already encountered accepting parsers: automata, CYK.

Natural Language Parsing

- Recognizing a sentence in two phases:

  Marce, veni!

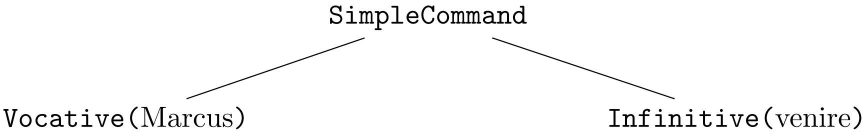
- Lexical analysis (2): recognize lexical structure: words, punctuation.

  Marce, veni!

- The lexical analysis returns a token stream.

  `Name`(Marce) `Comma` `Word`(veni) `Bang`

- Grammatical analysis (2): recognize grammatical structure.



`SimpleCommand`

`Vocative`(Marcus)

`Infinitive`(venire)

Formal Language Parsing

- Formal word:

  `wher+1 * 2where wher= (42)`

- Lexical analysis:

  `wher+1 * 2where wher= (42)`

  `Ident(wher) Plus Number(1) Times Number(2) Where`

  `Ident(wher) Equals LParen Number(42) RParen`

- Grammatical analysis:

  `Local(wher,Num(42),Plus(Var(wher),Times(Num(1),Num(2))))`

Calculator Grammar

- Naive Grammar

```
Expr ::= Ident | Number | ( Expr )
       | Expr * Expr | Expr + Expr
       | Expr where Ident = Expr
```

- This grammar is ambiguous:
  1+2*3 could be parsed as product `1+2 * 3` or sum `1 + 2*3`.
- Disambiguated grammar:

```
Atom    ::= Ident   | Number  | ( Expr )
Product ::= Atom    | Product * Atom
Sum     ::= Product | Sum + Product
Expr    ::= Sum     | Sum where Ident = Expr
```

From Grammars to Parser Generators

- Most programming languages adhere to a context-free grammar (CFG) suitable for efficient LR-parsing
- Division of labor:
  1. Lexer: transforms character string into token stream.
     - Discards whitespace and comments.
     - Recognizes numbers, string literals etc. via finite automata.
  2. Parser: processes token stream according to grammar.
- Automation:
  1. Lexers are generated from regular expressions.
  2. Parsers are generated from CFGs.

Lexical Analyzers

- Lexer is short for lexical analyzer.
- Big finite automaton with output: In accepting states, a token (depending on the state) is output.
- Typical form: $A = (A_1 + A_2 + \cdots + A_n)^*$
- Each automaton A_i has a specific output, e.g.:
  - A_1 recognizes whitespace, produces no output.
  - A_2 recognizes numbers, outputs the number.
  - A_3 recognizes `(`, outputs token `LParen`.
  - ...
- Longest match takes priority, then first match. E.g.:
  - `whereas`: Identifier RE has longer match than keyword `where`
  - `where`: Matches both identifier and keyword

Alex: a Lexer Generator for Haskell

- `.x` file maps regular expressions to output actions.

```
$white+    ;  -- no action
"where"    { \ s -> Where    }
@ident     { \ s -> Ident s }
@number    { \ s -> Number (read s) }
"+"        { \ s -> Plus     }
"*"        { \ s -> Times    }
"("        { \ s -> LParen   }
")"        { \ s -> RParen   }
"="        { \ s -> Equals   }
```

- Abbreviations (macros) for REs:

```
$digit  = 0-9           @number = 0 | $digit1 ( $digit * )
$digit1 = 1-9           @ident  = $lower +
$lower  = a-z
```

Example Tokens (Haskell code)

```
data Token
  = Ident  String     -- E.g. x
  | Number Integer    -- E.g. 123
  | Plus              -- +
  | Times             -- *
  | LParen            -- (
  | RParen            -- )
  | Equals            -- =
  | Where             -- where
```

LR Parsers

- LR = Left-to-right Rightmost-derivation.
- Efficient $O(n)$ bottom-up parsing using stack. (CYK: $O(n^3)$)
- Actions:
  1. Shift: put input token onto stack.
  2. Reduce: replace topmost stack symbols by a non-terminal, according to a grammar rule.
- Decision whether to shift or to reduce is taken by a finite automaton running over the stack contents.
- States of this FA are the parser states.

Run of a LR-Parser

```
Stack  Input   Action
       1+2*3   shift
1       +2*3   reduce  Atom    ::= Number
A       +2*3   reduce  Product ::= Atom
P       +2*3   reduce  Sum    ::= Product
S       +2*3   shift(2)
S+2       *3   reduce  Atom    ::= Number
S+A       *3   reduce  Product ::= Atom
S+P       *3   shift(2)
S+P*3          reduce  Atom    ::= Number
S+P*A          reduce  Product ::= Product * Atom
S+P            reduce  Sum     ::= Sum + Product
S              reduce  Expr    ::= Sum
E              accept
```

Happy: A Parser Generator for Haskell

- `https://www.haskell.org/happy/`
- `.y`-file contains token definitions and *grammar with actions*

```
Sum     : Product               { $1 }
        | Sum '+' Product       { plus $1 $3 }

Product : Atom                  { $1 }
        | Product '*' Atom      { times $1 $3 }

Atom    : num                   { number $1 }
        | '(' Expr ')'          { $2 }
```

- Haskell code inside the `{ braces }`.
- `$n` refers to value of *n*th item in rule.
- This parser directly computes the value of the parsed expression.

Happy: Token definitions

- Connect tokens accepted by Happy parser to the ones produced by the Alex lexer.

```
%tokentype { Token  }
%token
  '+'       { Plus   }
  '*'       { Times  }
  '('       { LParen }
  ')'       { RParen }
  '='       { Equals }
  'where'   { Where  }
  num       { Number $$ }  -- $$ holds the value of the tok
  id        { Ident  $$ }
```

BNFC: A BNF Compiler

- Usually, a parser should output the abstract syntax tree (AST).
- Calculating its value can be done in a second pass (interpretation).
- BNFC `http://bnfc.digitalgrammars.com/` gives additional convenience.
- `.cf` file contains BNF-grammar with rule names.
- BNFC produces input for several lexer/parser generators from the same grammar.
- The generated parsers produce ASTs.
- BNFC also produces pretty-printers and visitors for these ASTs.
- Supported languages include: C, C++, Haskell, Java.

Conclusions

- Suggested exercises:
- Implement the calculator in your favorite programming language using its lexer and parser generators.
- Extend the calculator by subtraction, division, etc.
- Extend the lexer towards single-line and block comments.
- Implement the calculator using BNFC.

Implementing Parsers

- We can write a parser directly, e.g. in Haskell.

```
parseNumber :: String -> Either Error (Integer, String)
```

- Parses a number and returns the remaining input.

```
parseNumber "345"     = Right (345, "")
parseNumber "1 + 2"   = Right (1, " + 2")
parseNumber "1hello" = Right (1, "hello")
parseNumber "hello"   = Left ExpectedNumber
```

- Should skip whitespace.

```
parseNumber "  345 " = Right (345, " ")
```

Composing Parsers

- Parsers can be combined (google: *parser combinators*)

```
type Parser a = String -> Either Error (a, String)
orP   :: Parser a -> Parser a -> Parser a
thenP :: Parser a -> Parser b -> Parser (a, b)
```

- Can we represent grammar as parser directly!?

```
parseAtom = parseNumber ‘orP‘
   (parseLParen ‘thenP‘ parseExpr ‘thenP‘ parseRParen)
```

- Parser combinators became popular with higher-order programming languages (Haskell, ML)
- However, there are some caveats . . .

Problems of Parser Combinators

- Naive translation of grammar fails

  ```
  parseExpr = parseProduct `orP`
     (parseExpr `thenP` parsePlus `thenP` parseProduct)
  ```

  `parseExpr "hello"` loops.
- Need to write grammar in a form suitable for *recursive-decent* aka *LL* (Left-to-right Left-most-derivation) parsing.
- Backtracking for alternative `orP` can be expensive.
  Parser might become exponential time.
- Let's put our formal language theory to work for efficient parsing!