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 2016
16 May 2016

This Lecture: a Taste of PLT

- A taste of an application of formal languages and automata

  Programming Language Technology
- Parsing, type-checking, interpretation, compilation
- DAT151 / DIT230
- Next edition: 2016/2017 LP2 (November-Jan)

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:
     - Parse tree
     - Abstract syntax tree
     - Result of interpreting input (if it is a program)

Running Example: Calculator

- This lecture: write a parser for a calculator

  `Expr ::= Number | Expr + Expr | Expr * Expr | ( Expr )`
- This grammar is ambiguous:
  `1+2*3` could be parsed as product `1+2 * 3` or sum `1 + 2*3`.
- Disambiguated grammar (left-associative):

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

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!

From Grammars to Parser Generators

- Parsing programming language is one of the foundations of IT
- Most programming languages adhere to a context-free grammar (CFG) suitable for efficient LR-parsing
- Division of task:
  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: $A = (A_1 + \cdots + A_n)^*$
- 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`.
  - ...

Alex: a Lexer Generator for Haskell

- `https://www.haskell.org/alex/`
- `.x` file maps regular expressions to output actions.

```
$white+   ;  -- no action
@number   { \ s -> Number (read s) }
@nulls    { \ s -> error ("invalid number " ++ s) }
"+"       { \ s -> Plus   }
"*"       { \ s -> Times  }
"("       { \ s -> LParen }
")"       { \ s -> RParen }
```

- Abbreviations (macros) for REs can be given:

```
$digit = 0-9
$digit1 = 1-9
@number = 0 | $digit1 ( $digit * )
@nulls  = 0 ( 0 + )
```

Example tokens (Haskell code)

```
data Token
  = Number Integer
  | Plus
  | Times
  | LParen
  | RParen
```

LR Parsers

- LR = Left-to-right Rightmost-derivation.
- Efficient bottom-up parsing using stack.
- Two actions:
  1. Shift: put input token onto stack.
  2. Reduce: replace topmost stack symbol by 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 Expr ::= Product
E	+2*3	shift(2)
E+2	*3	reduce Atom ::= Number
E+A	*3	reduce Product ::= Atom
E+P	*3	shift(2)
E+P*3		reduce Atom ::= Number
E+P*A		reduce Product ::= Product * Atom
E+P		reduce Expr ::= Expr + Product
E		accept

Happy: A Parser Generator for Haskell

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

```
Expr    : Product               { $1 }
        | Expr '+' Product      { $1 + $3 }

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

Atom    : num                   { $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 }
  num { Number $$ }  -- $$ holds the value of the token
```

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.
- Extend the calculator by variables and let-bindings.
- Implement the calculator using BNFC.