Grammars and Trees

Dr. Vadim Zaytsev aka @grammarware
2015
Compilers
Principles, Techniques, and Tools
Aho Lam Sethi Ullman
Second Edition
Recap

✓ Lexical analysis
✓ Syntactic analysis
✓ Semantic analysis
✓ Intermediate representation
✓ Code generation
✓ Optimisation
✓ . . .
WHY

✓ Formats everywhere
✓ DSLs are easy
✓ SLs have many faces
✓ 90% automated, 10% hard work
Models of Languages

✓ How can a language be defined?
Models of Languages

✓ Actual (in)finite set
✓ \{“a”, “b”, “c”\}
✓ \{0^i1^n\...\}
✓ English
✓ set arithmetic works
✓ concatenation, union, difference, intersection, complement, closure
Models of Languages

✓ Formal grammar
✓ term rewriting system
✓ “semi-Thue”
✓ all about rewriting rules
✓ $\alpha \rightarrow \beta$
Models of Languages

✓ Recognising automaton
✓ states
✓ transitions
✓ extra stuff
Models of Languages

✓ Declarative
  ✓ enumeration / description
  ✓ characteristic function

✓ Analytic
  ✓ recogniser / parser
  ✓ analytic grammar

✓ Generative
  ✓ term rewriting system
  ✓ generative grammar
Language

instance of

Program
The diagram illustrates the relationships between 

- **Language**
- **Automaton**
- **Grammar**
- **Sentences**

- **Language** is modelled by **Automaton** and **Grammar**.
- **Automaton** accepts **Sentences**.
- **Sentences** generates **Grammar**.

This diagram helps explain the concept of how different models and automata can be used to understand and generate language.
Language is modelled by Grammar.

Grammar generates Language.

Language is accepted by Automaton.

Automaton is parseable by Program.

Program is an element of Sentences.

Sentences conforms to Grammar.
Language defined by Grammar conforms to Program.
Example: XML

✓ X ::= ![<>]+
   | '<' ![<>]+ '>' X* '<' '/' ![<>]+ '>'

✓ X ::= D
   | '<' T A* '>' X* '<' '/' T '>'

✓ <!ELEMENT dir (#PCDATA)>
   <!ATTLIST dir xml:space (def|preserve) 'preserve'>

✓ <xsd:element name="tag">
   <xsd:complexType>
     . . .
Conclusion

✓ “Language” is intangible
✓ Grammars hide in:
  ✓ data types
  ✓ API and libraries
  ✓ protocols and formats
  ✓ structural commitments
  ✓ . . .
✓ Not all grammars are equally “good”
Fig. 2.33. The silhouette of a rose, approximated by Type 3 to Type 0 grammars

Unrestricted grammars

\[ \alpha \rightarrow \beta \]

Context-sensitive grammars

\[ \alpha X \beta \rightarrow \alpha \gamma \beta \]

Context-free grammars

\[ X \rightarrow \gamma \]

Regular grammars

\[ X \rightarrow a \]
\[ X \rightarrow aB \]

Noam Chomsky
(b.1928)

Unrestricted grammars
Decidable grammars
Context-sensitive grammars
Indexed grammars
Context-free grammars
Deterministic CFG
Nested word
Regular grammars
Non-recursive grammars

\[ \alpha \rightarrow \beta \]

\[ \alpha \chi \beta \rightarrow \alpha \gamma \beta \]

\[ A[\sigma] \rightarrow \alpha[\sigma] \]
\[ A[\sigma] \rightarrow B[f\sigma] \]
\[ A[f\sigma] \rightarrow \alpha[\sigma] \]

\[ X \rightarrow \gamma \]

\[ X \rightarrow a \]
\[ X \rightarrow aB \]

Noam Chomsky
(b. 1928)
<table>
<thead>
<tr>
<th>Unrestricted grammars</th>
<th>Recursively enumerable languages</th>
<th>Turing machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decidable grammars</td>
<td>Recursive languages</td>
<td>Terminating automata</td>
</tr>
<tr>
<td>Context-sensitive grammars</td>
<td>Context-sensitive languages</td>
<td>Linear-bounded automata</td>
</tr>
<tr>
<td>Indexed grammars</td>
<td>Languages with macros</td>
<td>Nested stack automata</td>
</tr>
<tr>
<td>Context-free grammars</td>
<td>Context-free languages</td>
<td>Pushdown automata</td>
</tr>
<tr>
<td>Deterministic CFG</td>
<td>Deterministic CFL</td>
<td>Deterministic PDA</td>
</tr>
<tr>
<td>Nested word</td>
<td>Nested word</td>
<td>Visibly PDA</td>
</tr>
<tr>
<td>Regular grammars</td>
<td>Regular languages</td>
<td>FSMs</td>
</tr>
<tr>
<td>Non-recursive grammars</td>
<td>Finite languages</td>
<td>FSMs without cycles</td>
</tr>
</tbody>
</table>
Finite languages

✓ Examples:
  ✓ Boolean values
  ✓ languages
  ✓ countries
  ✓ cities
  ✓ postcodes

LIST

ALL

THE ENTITIES
Regular languages

✓ Regular sets by Stephen Kleene in 1956
✓ $\emptyset$, $\epsilon$, letters from $\Sigma$
✓ concatenation
✓ iteration
✓ alternation
✓ Precisely fit the regular class


photo from: Konrad Jacobs, S. C. Kleene, 1978, MFO.
Regular languages

✓ PCRE
✓ “Perl-compatible regular expressions”
✓ (not compatible with Perl)
✓ (not regular)
✓ C library
✓ (backrefs, recursion, assertions...
Context-free

✓ FSM + memory (stack)
✓ Modular composition
  ✓ A ::= "[" B "]" ;
  ✓ B ::= A? ;

✓ Forget intersection & diff
✓ Closed under substitution

John Backus (1924–2007)
Context-sensitive

✓ Explainable only in context
✓ Sentence → List End
✓ List → Name;
✓ List → List “,” Name;
✓ “,” Name End → “and” Name
✓ Parsing in exponential time
Unbounded

✓ (almost) anything
✓ recognising is impossible
✓ parsing is impossible
Which is which?

- Substring search
  - grep, contains(), find(), substring(), ...
- Substring replacement
  - sed, awk, perl, vim, replace(), replaceAll(), ...
- Pretty-printing
  - VS.NET, Sublime, TextMate, ...
Which is which?

- Counting [non-empty] lines in a file
  - `wc -l, grep -c ""`
  - `grep -v "^\$", sed -n \./\./p | wc -l`

- Parsing HTML
  - `<BODY><TABLE><P><A HREF="...">

- Parsing a postcode
  - `1098 XG, ...`
Popular languages

✓ \{ a^i b^n \ldots \}
✓ 0 counters
✓ 1 counter
✓ n counters
✓ \infty counters
✓ Dyck language
✓ parentheses
✓ named parentheses

Walther von Dyck
(1856–1934)

Popular parsers

✓ **Bottom-up**
  ✓ Reduce the input back to the start symbol
  ✓ Recognise terminals
  ✓ Replace terminals by nonterminals
  ✓ Replace terminals and nonterminals by left-hand side of rule
  ✓ LR, LR(0), LR(1), LR(k), LALR, SLR, GLR, SGLR, CYK, ...

✓ **Top-down**
  ✓ Imitate the production process by rederivation
  ✓ Each nonterminal is a goal
  ✓ Replace each goal by subgoals (= elements of its rule)
  ✓ Parse tree is built from top to bottom
  ✓ LL, LL(1), LL(k), LL(*), GLL, DCG, RD, Packrat, Earley
Popular parsers

✓ Bottom-up
✓ YACC / bison
✓ Beaver
✓ SableCC
✓ GDK
✓ Tom
✓ ASF+SDF
✓ Spoofax
✓ Top-down
✓ JavaCC
✓ ANTLR
✓ ModelCC
✓ Rascal
✓ TXL
✓ Rats!
✓ PetitParser
Popular data structures

✓ Lists (of tokens)
✓ Trees (hierarchy!)
✓ Forests (many trees)
✓ Graphs (loops!)
✓ Relations (tables)
Conclusion

✓ Parsing recognises structure
✓ Can be many models of a language
✓ Hierarchy of classes
✓ 90% automated, 10% hard work
Lexical syntax

✓ Terminal symbols
✓ finite sublanguage
✓ regular sublanguage
✓ Keywords
✓ Layout
✓ whitespace
✓ comments
Lexical syntax

**lexical** Boolean = "True" | "False";

**lexical** Id = [a–z]+ !>> [a–z];

**keyword** Reserved = "if" | "while";
**lexical** Id = [a–z]+ \ \\ Reserved !>> [a–z];

**lexical** WS = [\ \t\n\r];

**lexical** Cm = "--" ... $;

**layout** L = (WS|Cm)* !>> [\\ \t\n\r] !>> "--";
Lexical syntax

XML

layout L = [\t\n\r]* !>> [\t\n\r];
lexical D = ![[<\>]* !>> ![<\>];
lexical T = [a-z][a-z0-9]* !>> [a-z0-9];
lexical A = [a-z]+ [=] [\"] ![\"]* [\"];
lexical X = D
    | "\<" T A* "\>" X+ "\<" "\/>" T "\/>";
Beyond lexical

XML

layout L = [\t\n]* !>> [\t\n];
lexical D = ![<>]* !>> ![<>];
lexical T = [a-z][a-z0-9]* !>> [a-z0-9]:
lexical A = [a-z]+ [=] ["" ] ![["]
lexical X = D
  | " " T L{A L}* "">" X+ "" <"
Beyond lexical

XML

layout L = [\t\n\r]* !>> [\t\n\r];
lexical D = ![<>]* !>> ![<>];
lexical T = [a-z][a-z0-9]* !>> [a-z0-9]:
lexical A = [a-z]+ [=] [\"] ![\"]
lexical X = D | "<>" T L {A L}* "\>" X+ "\<"
Beyond lexical

XML

layout L = [ \t\n\r]* !>> [ \t\n\r];
syntax D = W+;
lexical W = ![\t\n\r<\>]+ !>> ![\t\n\r<\>];
lexical T = [a-z][a-z0-9]* !>> [a-z0-9];
lexical A = [a-z]+ [=] ["" ] !["" ]* ["" ];
syntax X = D
    | "\<" T A* "\>" X* "\<" "\//" T "\>";
Recap: lexical

✓ Terminal: "if"
✓ Character class: [a−z]
✓ Inverse: ![a−z]
✓ Kleene closures: [a−z]+, [a−z]∗
✓ Optionals: [a−z]? 
✓ Reserve: [a−z]+ \ Keywords 
✓ Follow: [a−z]+ !>>> [a−z]
Beyond lexical

✓ Choice: |  
✓ Priority: >  
✓ Associativity: left, right, non-assoc  
✓ Named alternatives: foo: x  
✓ Named symbols: E left "+" E right  
✓ Regular combinators: X*, X+, X?
✓ parse(#N, s)
✓ try parse(#N, s) catch: . . .
✓ vis::ParseTree::renderParseTree(t)
✓ /amb(_) !:= t
✓ t is foo
✓ t.x
✓ if (pattern := tree) . . .
✓ (E)`<E e1> + <E e2>`
✓ /regexp/