Introduction to Grammars and Parsing Techniques

Paul Klint
Grammars and Languages are one of the most established areas of Computer Science
N. Chomsky,
Aspects of the theory of syntax,
1965
Why are Grammars and Parsing Techniques relevant?

- **A grammar** is a formal method to describe a (textual) *language*
  - Programming languages: C, Java, C#, JavaScript
  - Domain-specific languages: BibTex, Mathematica
  - Data formats: log files, protocol data

- **Parsing**:
  - Tests whether a text conforms to a grammar
  - Turns a correct text into a parse tree
How to define a grammar?

- Simplistic solution: finite set of acceptable sentences
  - **Problem**: what to do with infinite languages?
- Realistic solution: *finite recipe* that describes all acceptable sentences
- A grammar is a finite description of a possibly infinite set of acceptable sentences
Example: Tom, Dick and Harry

Suppose we want describe a language that contains the following legal sentences:

- Tom
- Tom and Dick
- Tom, Dick and Harry
- Tom, Harry, Tom and Dick
- ...

How do we find a finite recipe for this?
The Tom, Dick and Harry Grammar

- **Name** -> **tom**
- **Name** -> **dick**
- **Name** -> **harry**
- **Sentence** -> **Name**
- **Sentence** -> **List End**
- **List** -> **Name**
- **List** -> **List**, **Name**
- **, Name End** -> **and** **Name**

Non-terminals: Name, Sentence, List, End

Terminals: **tom**, **dick**, **harry**, **and**, **,**

Start Symbol: **Sentence**
Variations in Notation

- **Name** -> `tom` | `dick` | `harry`
- `<Name>` ::= “tom” | “dick” | “harry”
- “tom” | “dick” | “harry” -> Name
Chomsky’s Grammar Hierarchy

• Type-0: *Recursively Enumerable*
  • Rules: $\alpha \rightarrow \beta$ (unrestricted)

• Type-1: *Context-sensitive*
  • Rules: $\alpha A\beta \rightarrow \alpha \gamma \beta$

• Type-2: *Context-free*
  • Rules: $A \rightarrow \gamma$

• Type-3: *Regular*
  • Rules: $A \rightarrow a$ and $A \rightarrow aB$
Context-free Grammar for TDH

- **Name** -> tom | dick | harry
- **Sentence** -> Name | List and Name
- **List** -> Name , List | Name
In practice ...

- **Regular grammars** used for lexical syntax:
  - Keywords: if, then, while
  - Constants: 123, 3.14, “a string”
  - Comments: /* a comment */

- **Context-free grammars** used for structured and nested concepts:
  - Class declaration
  - If statement
A sentence

\[
\text{position} := \text{initial} + \text{rate} \times 60
\]
Lexical syntax

• Regular expressions define lexical syntax:
  • Literal characters: a,b,c,1,2,3
  • Character classes: [a-z], [0-9]
  • Operators: sequence (space), repetition (* or +), option (?)

• Examples:
  • Identifier: [a-z][a-z0-9]*
  • Number: [0-9]+
  • Floating constant: [0-9]*.[0-9]*\(e-[0-9]+\)
Lexical syntax

- Regular expressions can be implemented with a finite automaton
- Consider \([a-z][a-z0-9]*\)
Lexical Tokens

```
position := initial + rate * 60
```

- `Identifier`:
  - `position`
  - `initial`
  - `rate`
  - `Number`

- `operator`:
  - `:=`
  - `+`
  - `*`
Context-free syntax

• Expression -> Identifier
• Expression -> Number
• Expression -> Expression + Expression
• Expression -> Expression * Expression
• Statement -> Identifier := Expression
Parse Tree

position := initial + rate * 60
Ambiguity: one sentence, but several trees

```
Expression
    /   \  \
Expression Number
     / \   /
Number Number Number
  / |  / |  /
1  +  2  *  3

Expression
    /   \  \
Expression Number
     / \   /
Number Number Number
  / |  / |  /
1  +  2  *  3
```
Two solutions

• Add priorities to the grammar:
  • * > +

• Rewrite the grammar:
  • Expression -> Expression + Term
  • Expression -> Term
  • Term -> Term * Primary
  • Term -> Primary
  • Primary -> Number
  • Primary -> Identifier
Unambiguous Parse Tree

```
Expression
  /       \\
Term     Term
   / \\
Primary Primary Primary
  / \\
Number Number Number
```

1 + 2 * 3
Some Grammar Transformations

- **Left recursive production:**
  - $A \rightarrow A\alpha | \beta$
  - Example: $Exp \rightarrow Exp + Term | Term$

- Left recursive productions lead to loops in some kinds of parsers (recursive descent)

- **Removal:**
  - $A \rightarrow \beta R$
  - $R \rightarrow \alpha R | \varepsilon$  (\(\varepsilon\) is the empty string)
Some Grammar Transformations

- Left factoring:
  - $S \rightarrow \text{if } E \text{ then } S \text{ else } S \mid \text{if } E \text{ then } S$
- For some parsers it is better to factor out the common parts:
  - $S \rightarrow \text{if } E \text{ then } S \text{ P}$
  - $P \rightarrow \text{else } S \mid \epsilon$
A Recognizer

Grammar

\[ l_1 \rightarrow r_1 \]
\[ l_2 \rightarrow r_2 \]
\[ \ldots \]
\[ l_n \rightarrow r_n \]

Source text

Recognizer

Yes or No
A Parser

Grammar

Source text

Parser

Parse tree or Errors

Introduction to Grammars and Parsing Techniques

27
General Approaches to Parsing

- **Top-Down** (Predictive)
  - Each non-terminal is a goal
  - Replace each goal by subgoals (= elements of rule)
  - Parse tree is built from top to bottom

- **Bottom-Up**
  - Recognize terminals
  - Replace terminals by non-terminals
  - Replace terminals and non-terminals by left-hand side of rule
Top-down versus Bottom-up

Top-down

Expression

Expression

Number 4

Number 5

Expression

Number

Expression

Number

Number 6

Bottom-up

Expression

Expression

Number

Number

Expression

Number 1

Expression

Number

Expression

Number 2

Expression

Number

Number 3
How to get a parser?

- Write parser manually
- Generate parser from grammar
Example

- Given grammar:
  - $\text{Expr} \to \text{Expr} + \text{Term}$
  - $\text{Expr} \to \text{Expr} - \text{Term}$
  - $\text{Expr} \to \text{Term}$
  - $\text{Term} \to [0-9]$  

- Remove left recursion:
  - $\text{Expr} \to \text{Term} \text{Rest}$
  - $\text{Rest} \to + \text{Term} \text{Rest} | - \text{Term} \text{Rest} | \varepsilon$
  - $\text{Term} \to [0-9]$
Recursive Descent Parser

Expr(){ Term(); Rest(); }

Rest(){ if(lookahead == '+'){ Match('+'); Term(); Rest(); } else if( lookahead == '-'){ Match('-'); Term(); Rest(); } else ; }

Term(){ if(isdigit(lookahead)){ Match(lookahead); } else Error(); }
A Trickier Case: Backtracking

- Example
  - $S \rightarrow aSbS \mid aS \mid c$
- Naive approach (input $ac$):
  - Try $aSbS$, but this fails hence error
- Backtracking approach:
  - First try $aSbS$ but this fails
  - Go back to initial input position and try $aS$, this succeeds.
Automatic Parser Generation

Syntax of $L$

Parser Generator

$L$ text

Parser for $L$

$L$ tree
Some Parser Generators

- **Bottom-up**
  - Yacc/Bison, LALR(1)
  - CUP, LALR(1)
  - SDF, SGLR

- **Top-down:**
  - ANTLR, LL(k)
  - JavaCC, LL(k)
  - Rascal

- Except Rascal and SDF, all depend on a scanner generator
Assessment parser implementation

- **Manual parser construction**
  - Good error recovery
  - Flexible combination of parsing and actions
  - A lot of work

- **Parser generators**
  - *May* save a lot of work
  - Complex and rigid frameworks
  - Rigid actions
  - Error recovery more difficult
Further Reading

Further Reading

