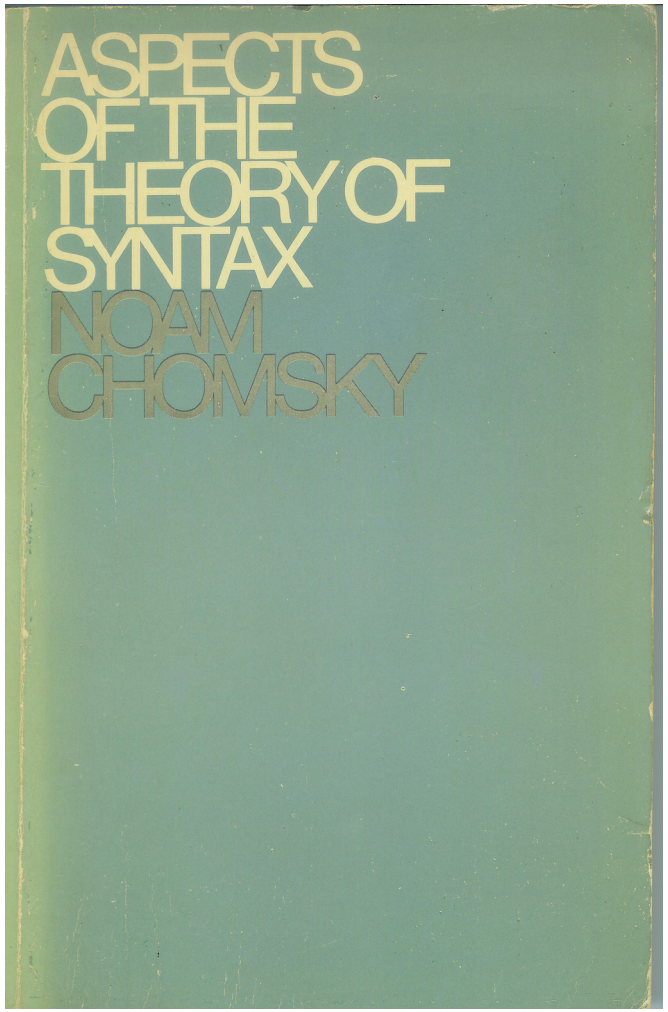
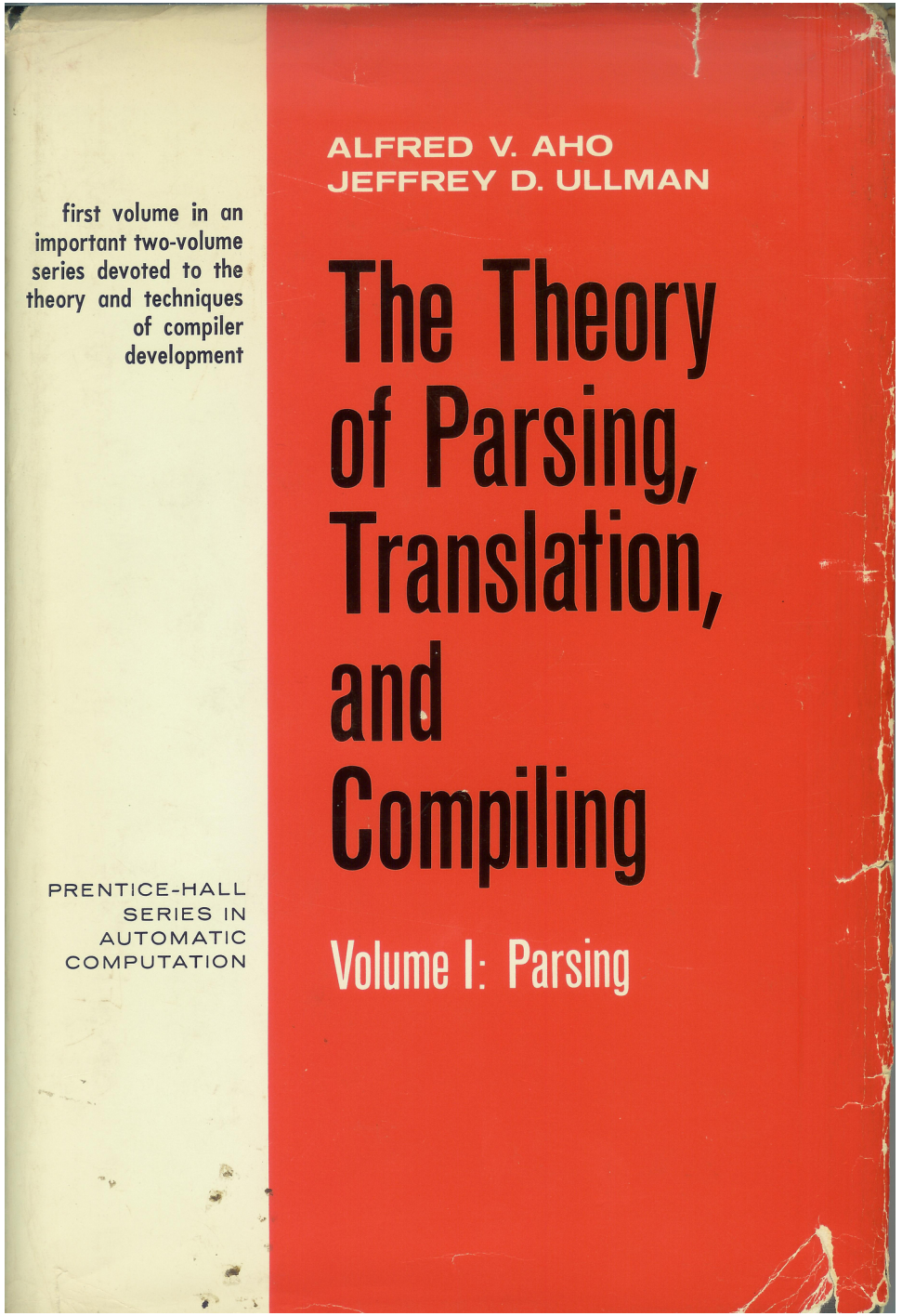
Introduction to Grammars and Parsing Techniques

Tijs van der Storm
(/ht Paul Klint)

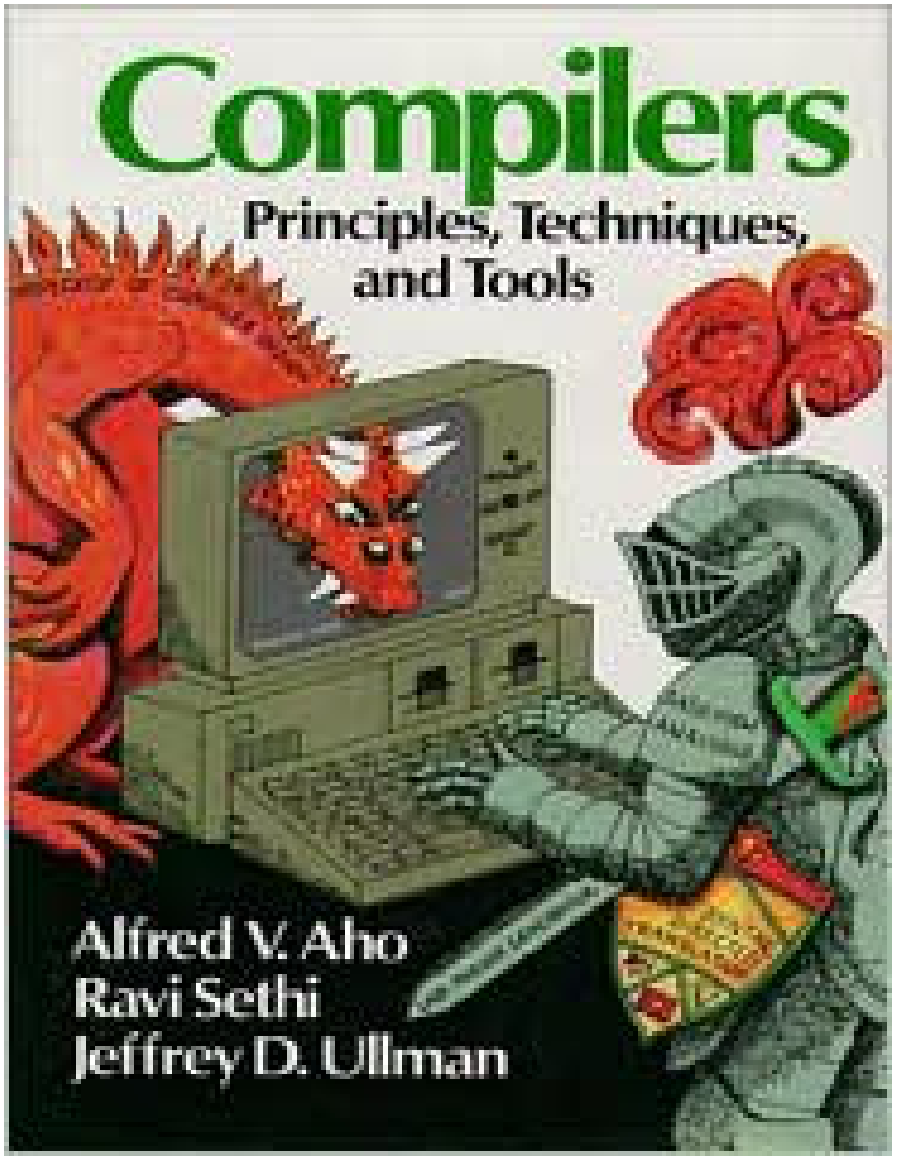
Grammars and Languages are one of the most established areas of Computer Science



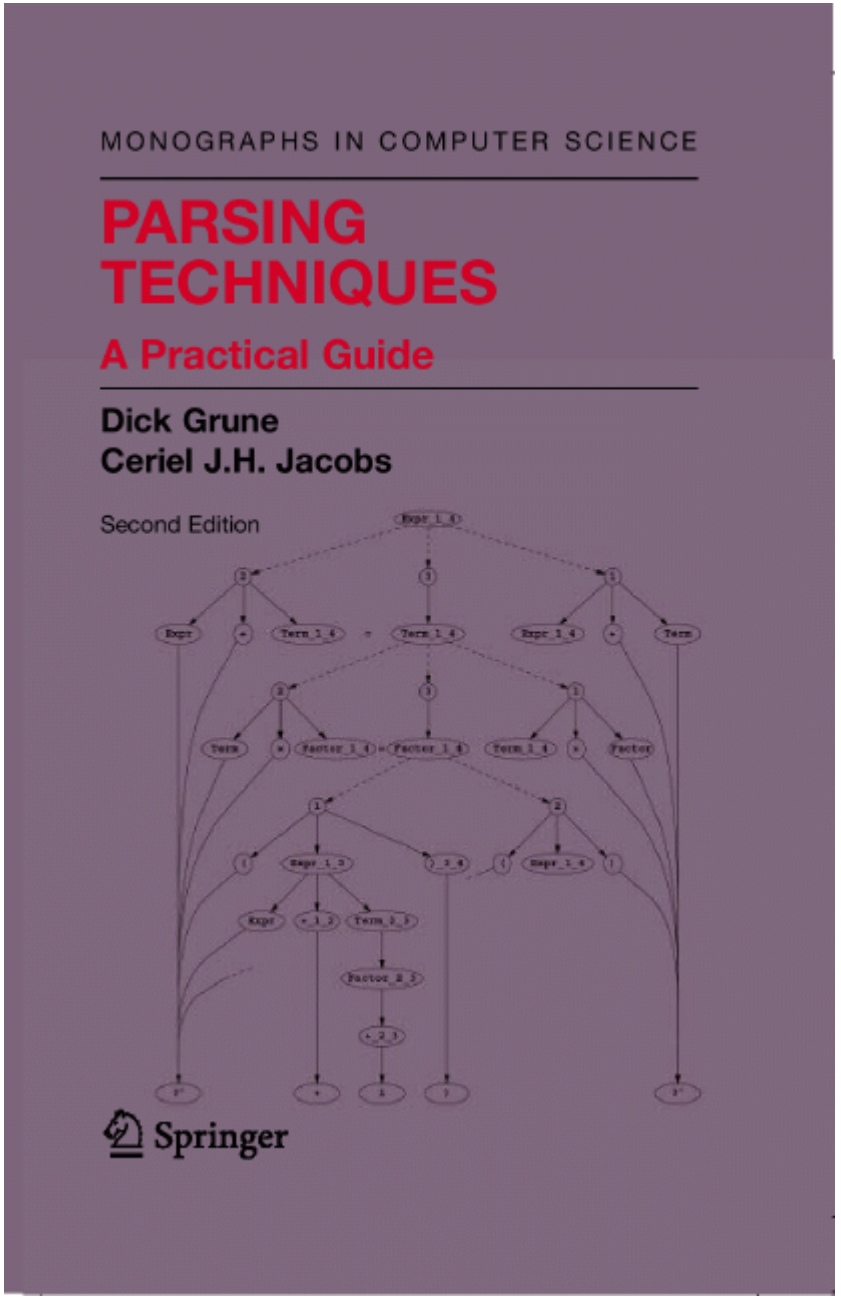
N. Chomsky,
Aspects of the theory of syntax,
1965



A.V. Aho & J.D. Ullman,
The Theory of Parsing,
Translation and Compilimg,
Parts I + II,
1972



A.V. Aho, R. Sethi, J.D. Ullman, Compiler, Principles, Techniques and Tools, 1986



D. Grune, C. Jacobs,
Parsing Techniques,
A Practical Guide,
2008

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: α -> β (unrestricted)
- Type-1: Context-sensitive
  - Rules: αAβ -> αγβ
- Type-2: Context-free
  - Rules: A -> γ
- Type-3: Regular
  - Rules: A -> a and A -> aB

Context-free Grammar for TDH

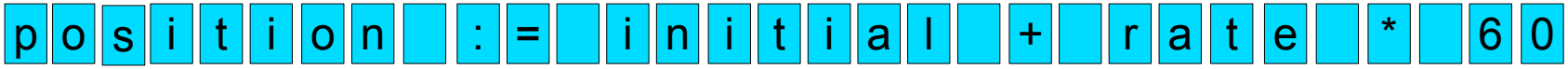
- Name -> **tom** | **dick** | **harry**
- Sentence -> Name | List and Name
- List -> Name , List | Name

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

position := initial + rate * 60

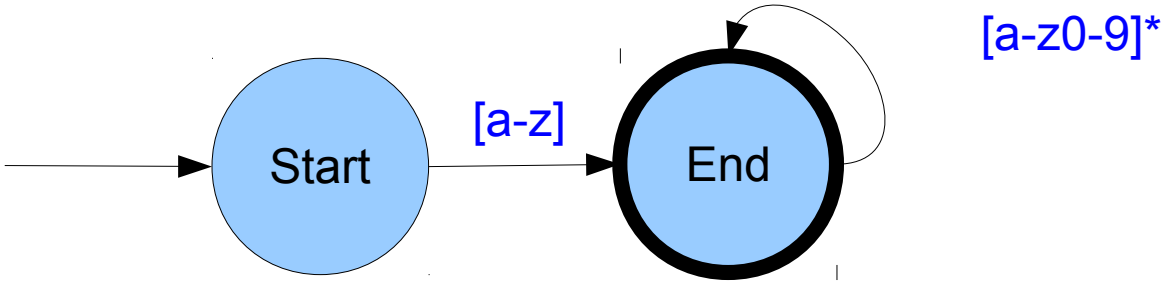


Lexical syntax

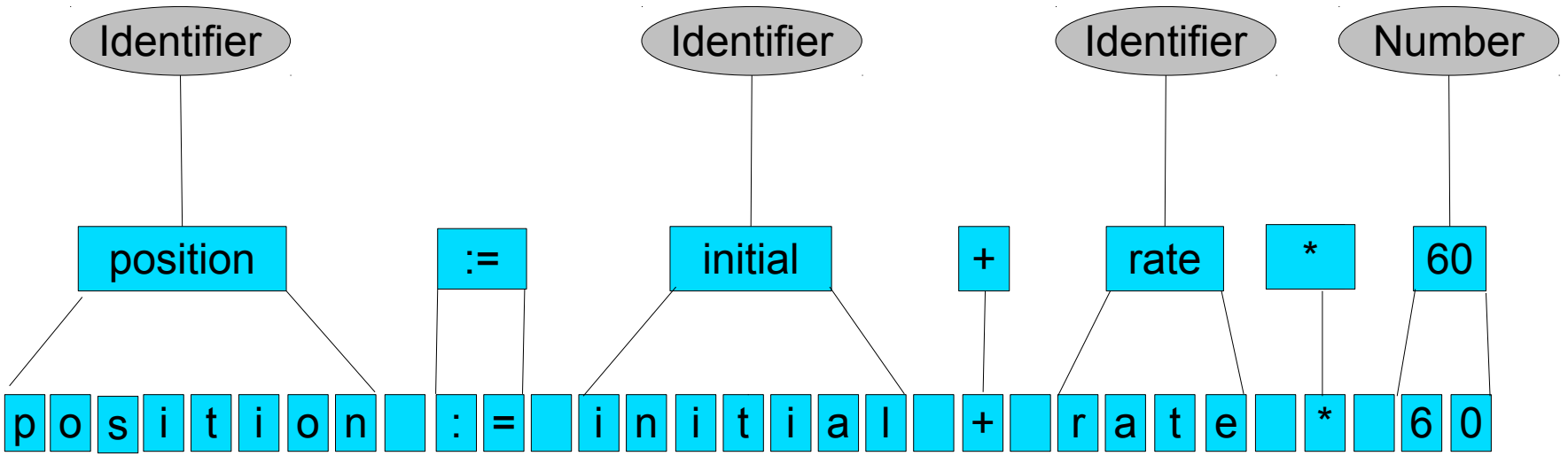
- Regular expressions define lexical syntax:
  - Literal characters: a,b,c,1,2,3
  - Character classes: [a-z], [0-9]
  - Operators: sequence (space), repetition (* or +), option (?)
- Examples:
  - Identifier: [a-z][a-z0-9]*
  - Number: [0-9]+
  - Floating constant: [0-9]*.[0-9]*(e-[0-9]+)

Lexical syntax

- Regular expressions can be implemented with a finite automaton
- Consider [a-z][a-z0-9]*



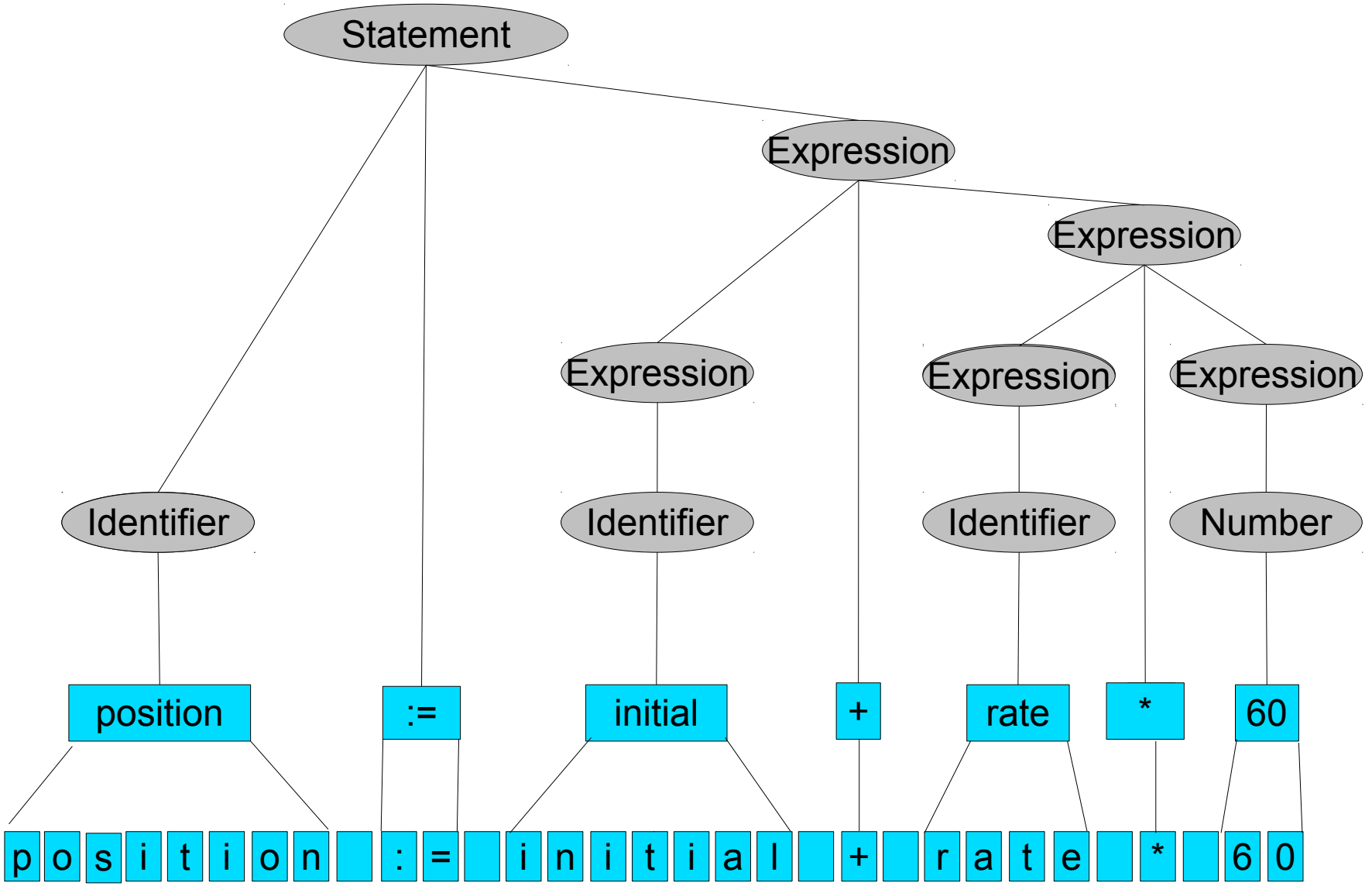
Lexical Tokens



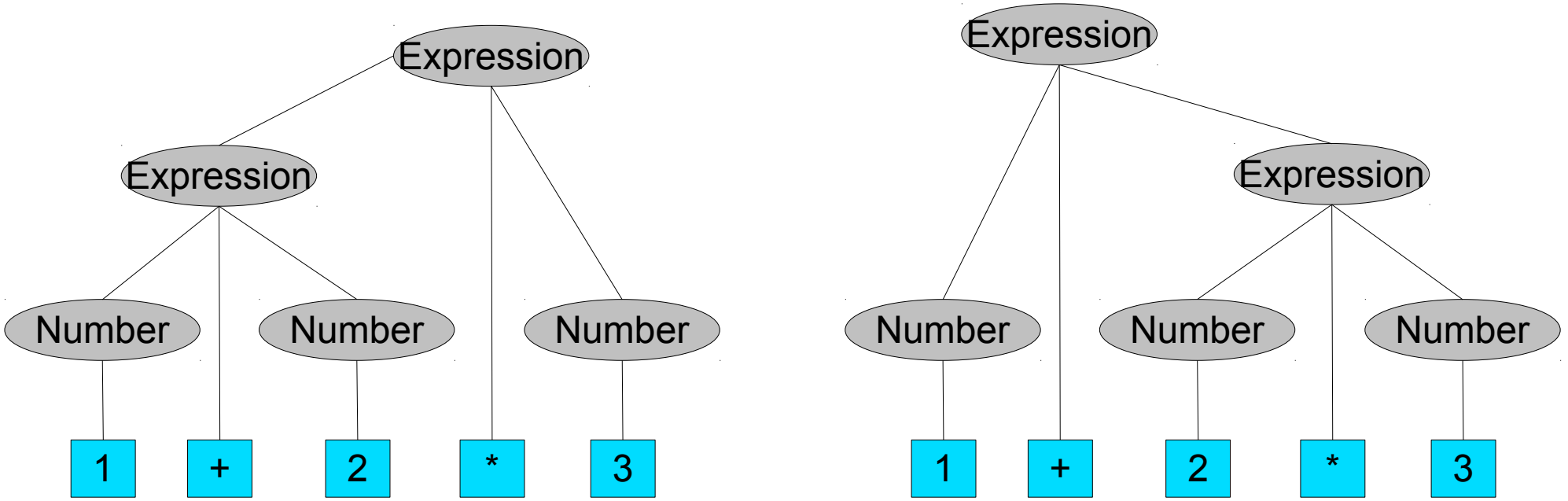
Context-free syntax

- Expression -> Identifier
- Expression -> Number
- Expression -> Expression + Expression
- Expression -> Expression * Expression
- Statement -> Identifier := Expression

Parse Tree


Statement
Expression
Expression
Expression
Expression
Expression
Identifier
Identifier
Identifier
Number
position
:=
initial
+
rate
*
60
p o s i t i o n : = i n i t i a l + r a t e * 6 0

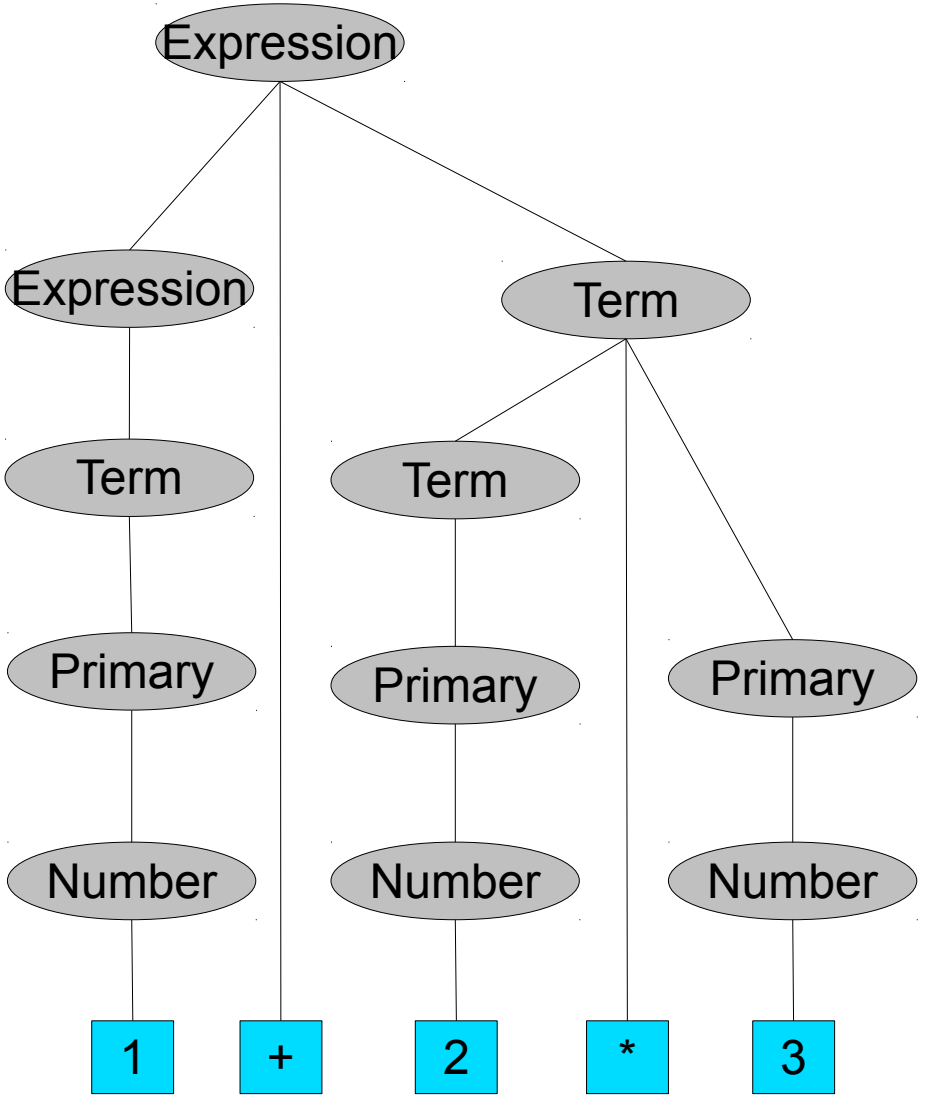
Ambiguity: one sentence, but several trees



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



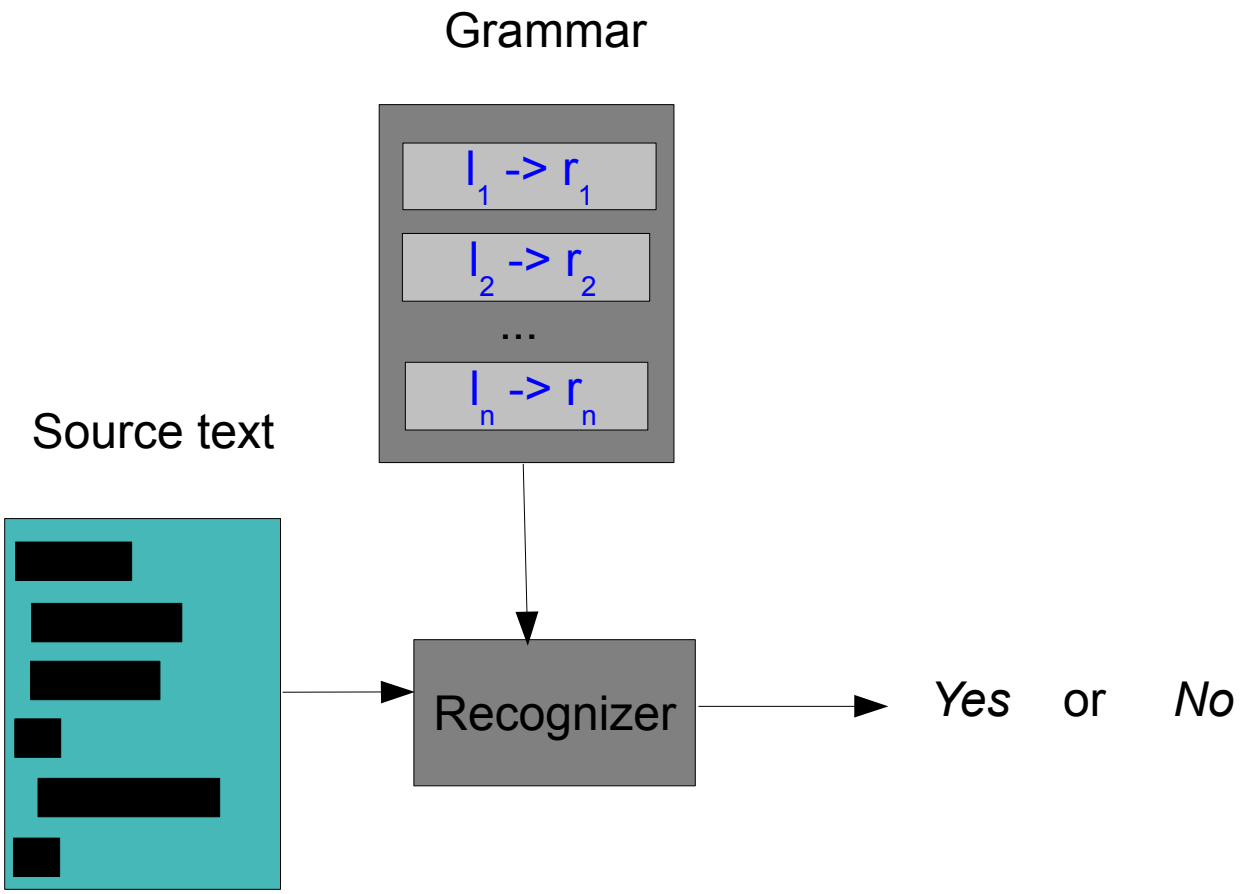
Some Grammar Transformations

- Left recursive production:
  - A -> A α | β
  - Example: Exp -> Exp + Term | Term
- Left recursive productions lead to loops in some kinds of parsers (recursive descent)
- Removal:
  - A -> β R
  - R -> α R | ε (ε is the empty string)

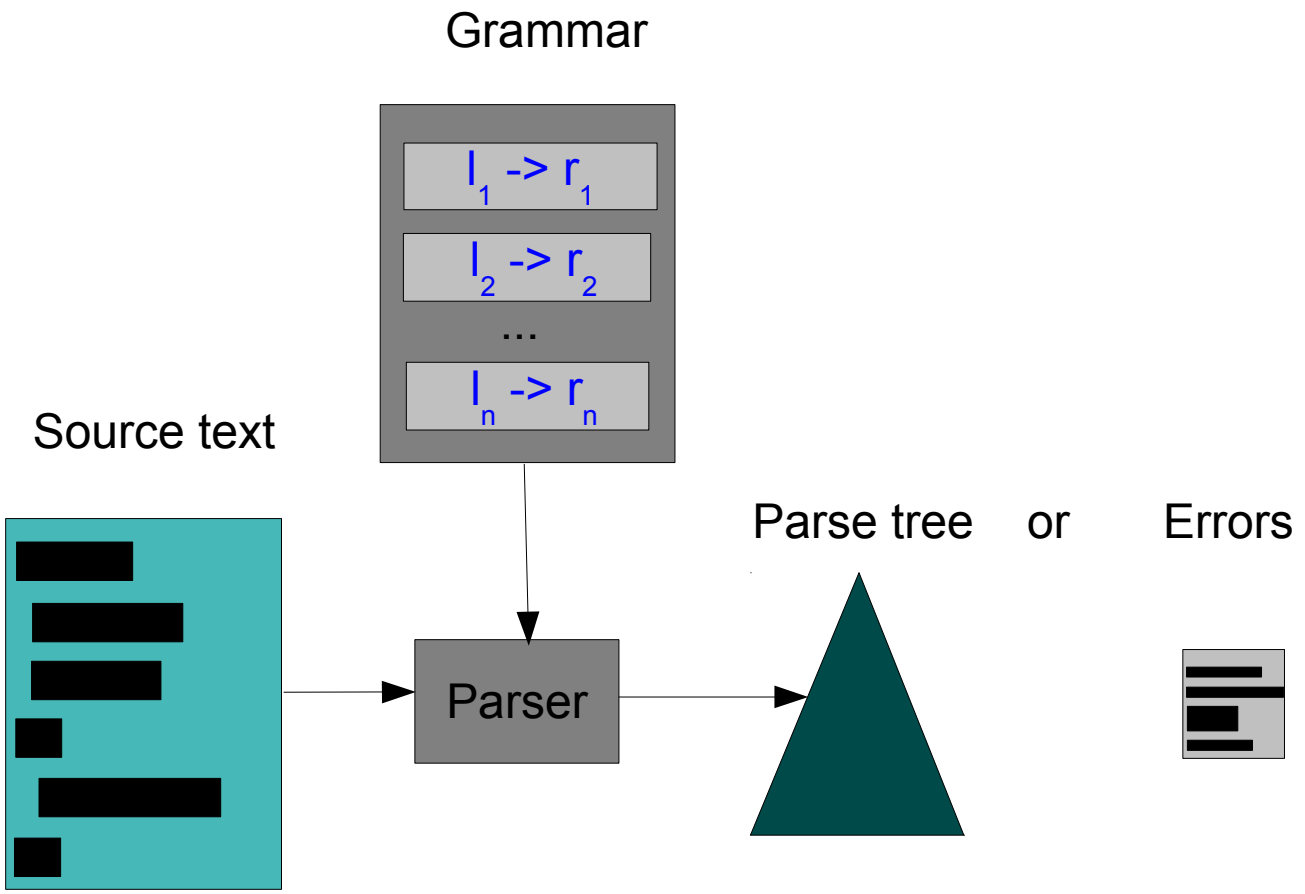
Some Grammar Transformations

- Left factoring:
  - S -> **if** E **then** S **else** S | **if** E **then** S
- For some parsers it is better to factor out the common parts:
  - S -> **if** E **then** S P
  - P -> **else** S | ε

A Recognizer



A Parser

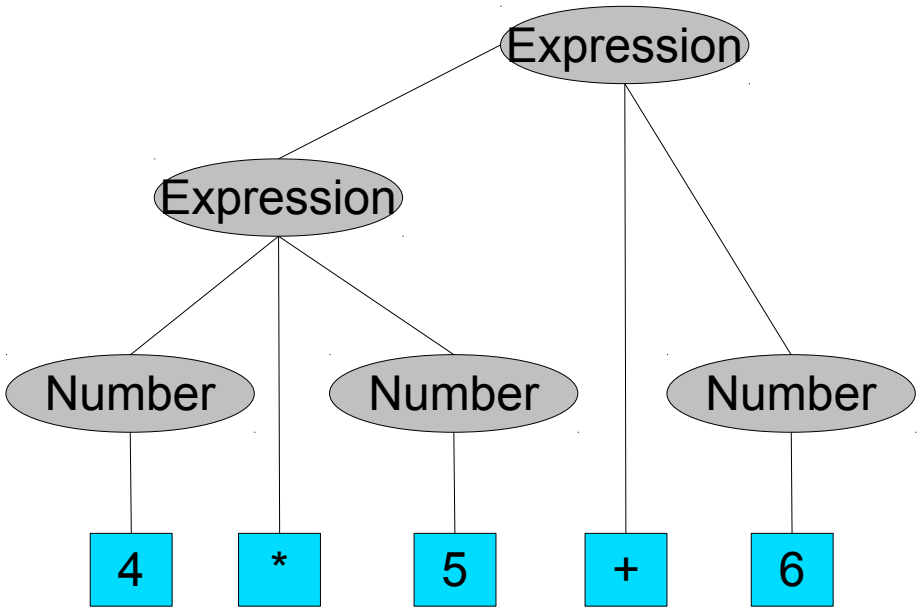

Grammar
l_1 -> r_1
l_2 -> r_2
...
l_n -> r_n
Source text
Parser
Parse tree
or
Errors

General Approaches to Parsing

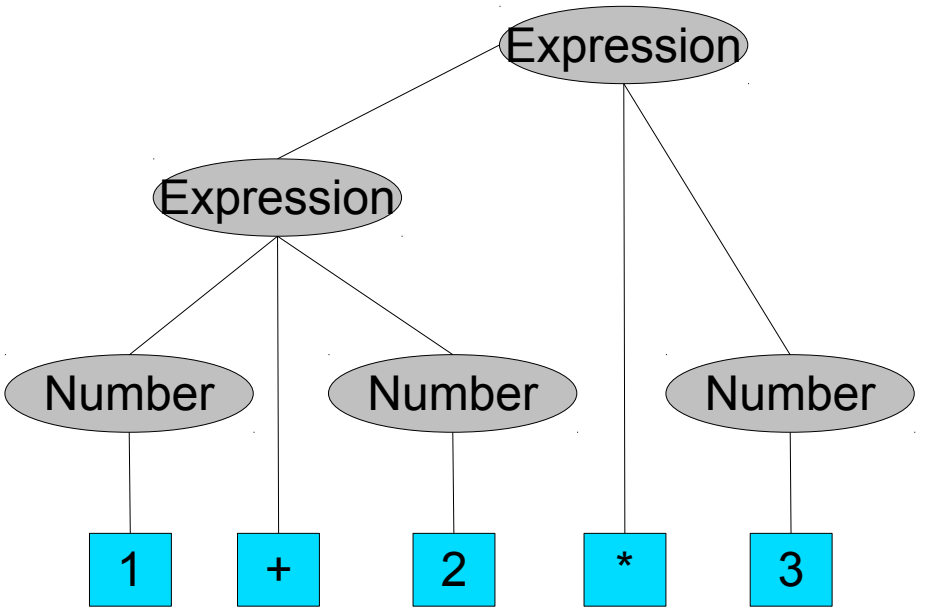
- Top-Down (Predictive)
  - Each non-terminal is a goal
  - Replace each goal by subgoals (= elements of rule)
  - Parse tree is built from top to bottom
- Bottom-Up
  - Recognize terminals
  - Replace terminals by non-terminals
  - Replace terminals and non-terminals by left-hand side of rule

Top-down versus Bottom-up

Top-down



Bottom-up



How to get a parser?

- Write parser manually
- Generate parser from grammar

Example

- Given grammar:
  - Expr -> Expr **+** Term
  - Expr -> Expr **–** Term
  - Expr -> Term
  - Term -> [0-9]
- Remove left recursion:
  - Expr -> Term Rest
  - Rest -> **+** Term Rest | **-** Term Rest | ε
  - Term -> [0-9]

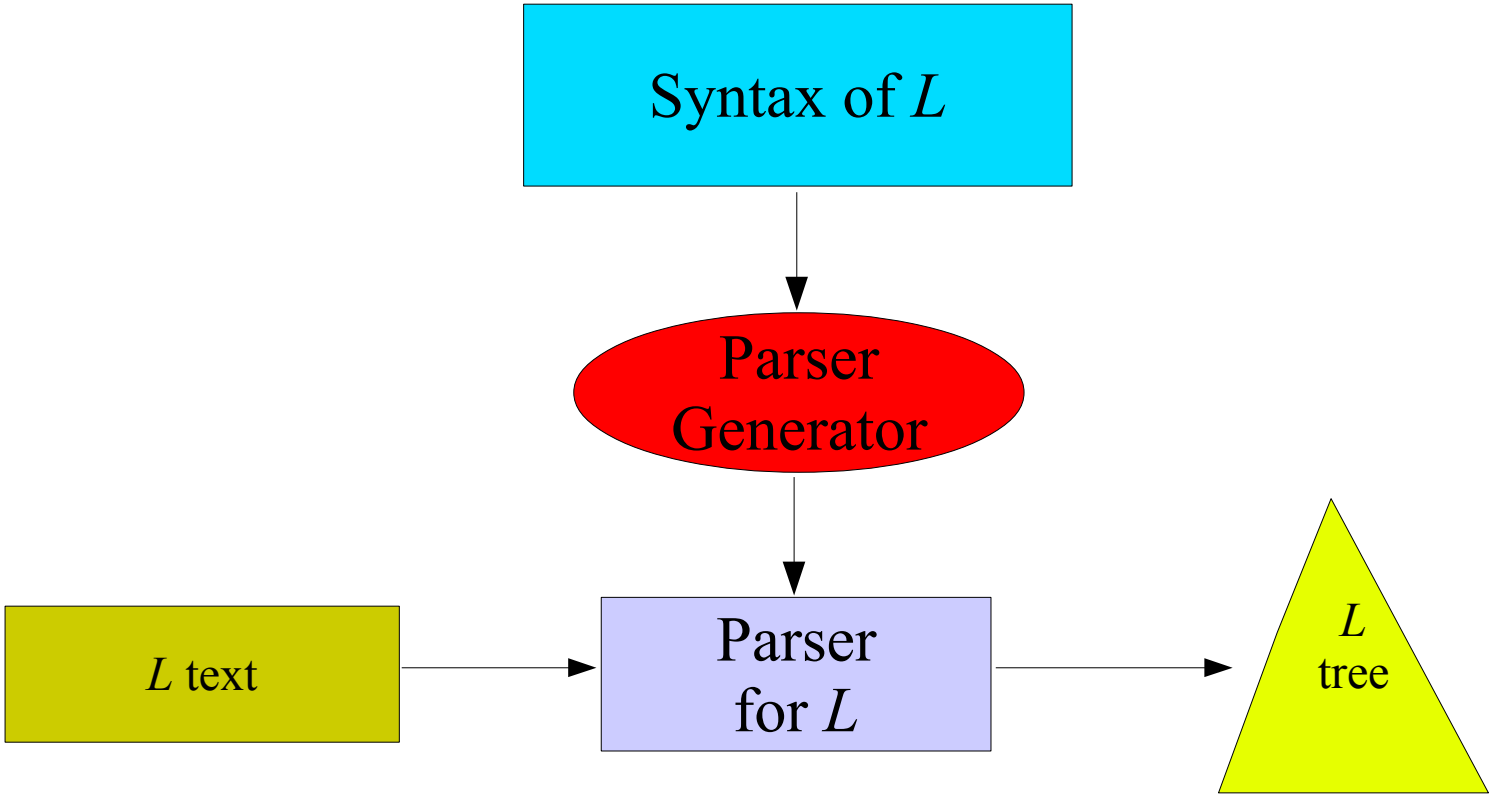
Recursive Descent Parser

```
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 -> aSbS | aS | 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



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

- http://en.wikipedia.org/wiki/Chomsky_hierarchy
- D. Grune & C.J.H. Jacobs, *Parsing Techniques: A Practical Guide*, Second Edition, Springer, 2008

Further Reading

- http://en.wikipedia.org/wiki/Chomsky_hierarchy
- Paul Klint, Quick introduction to Syntax Analysis, http://www.meta-environment.org/doc/books//syntax/syntax-analysis/syntax-analysis.html
- D. Grune & C.J.H. Jacobs, *Parsing Techniques: A Practical Guide*, Second Edition, Springer, 2008