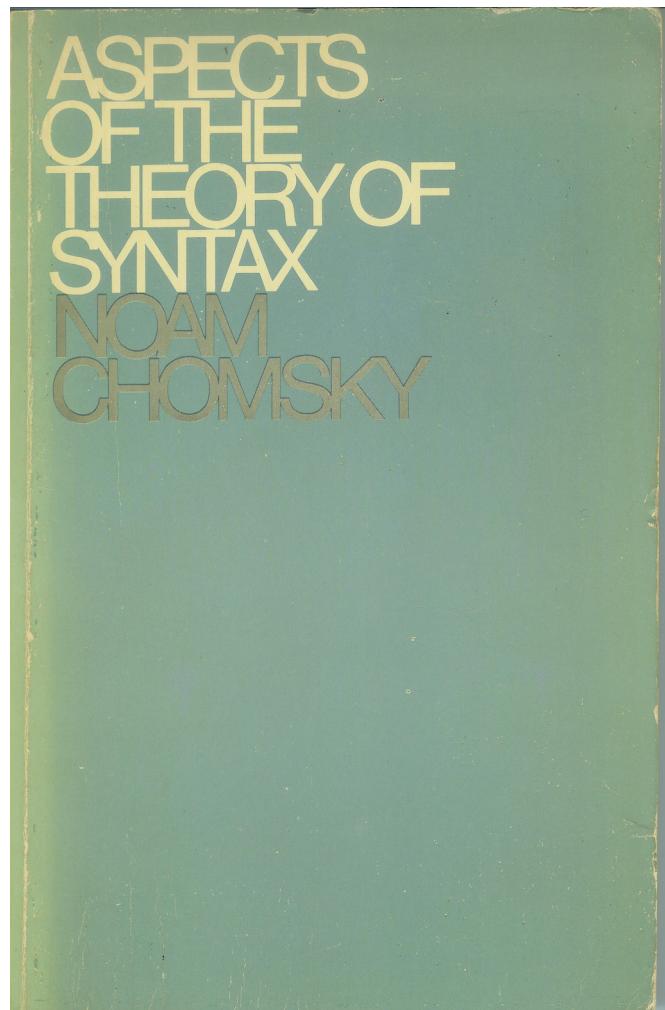


Introduction to Grammars and Parsing Techniques

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

**ALFRED V. AHO
JEFFREY D. ULLMAN**

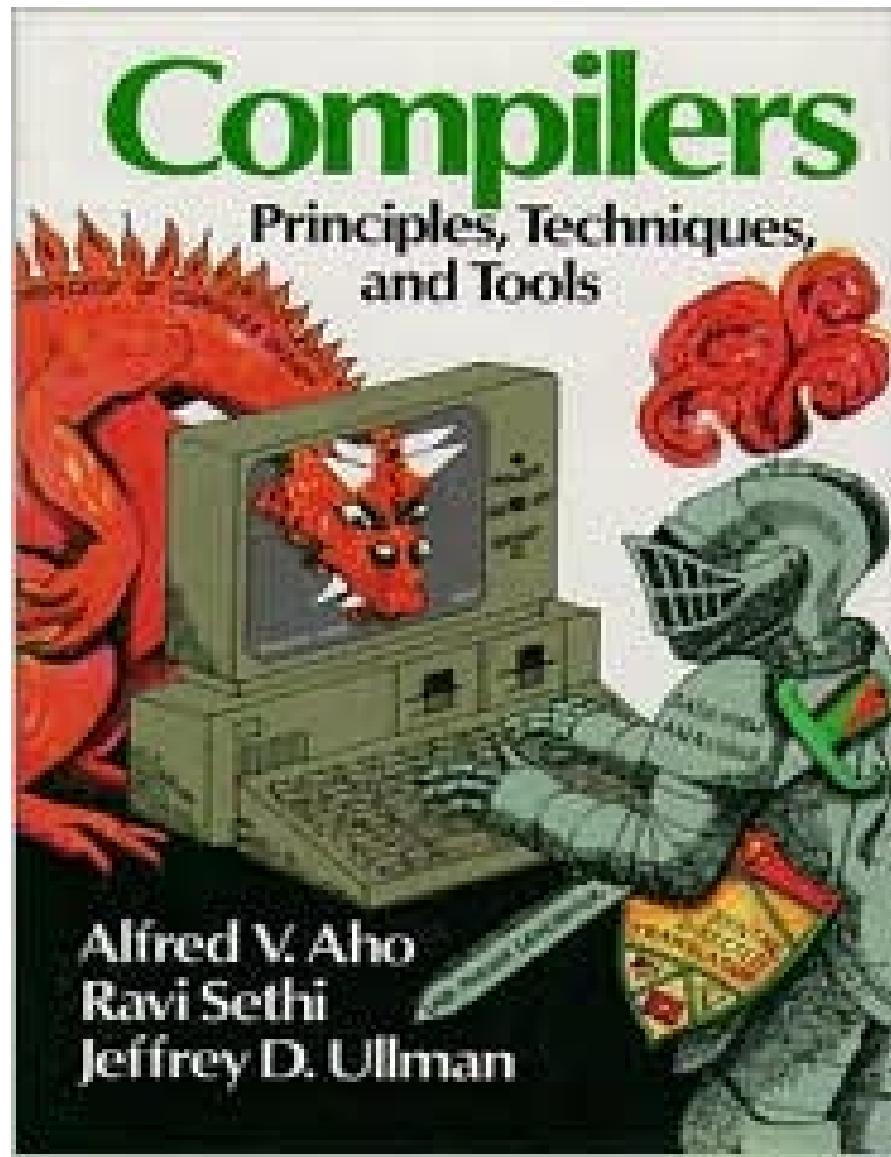
first volume in an
important two-volume
series devoted to the
theory and techniques
of compiler
development

The Theory of Parsing, Translation, and Compiling

Volume I: Parsing

PRENTICE-HALL
SERIES IN
AUTOMATIC
COMPUTATION

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



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

MONOGRAPHS IN COMPUTER SCIENCE

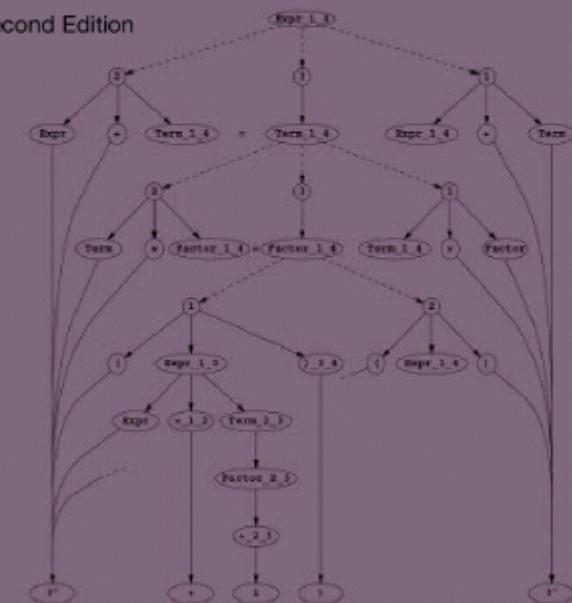
PARSING TECHNIQUES

A Practical Guide

Dick Grune

Ceriel J.H. Jacobs

Second Edition



 Springer

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: $\alpha \rightarrow \beta$ (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 \rightarrow **tom** | **dick** | **harry**
- Sentence \rightarrow Name | List and Name
- List \rightarrow 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



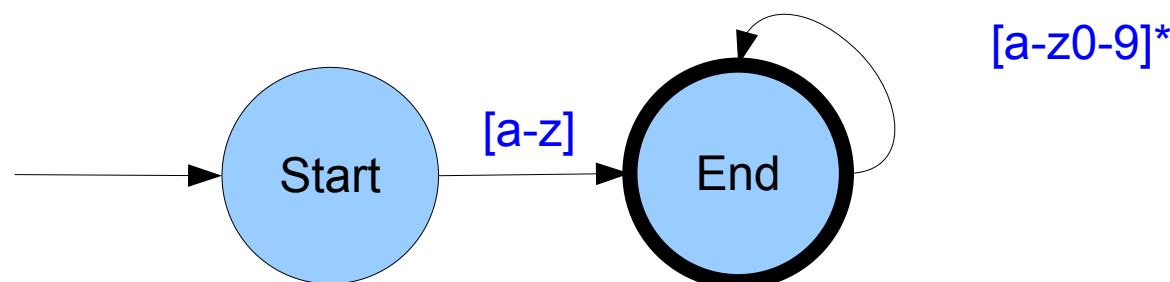
position := initial + rate * 60

Lexical syntax

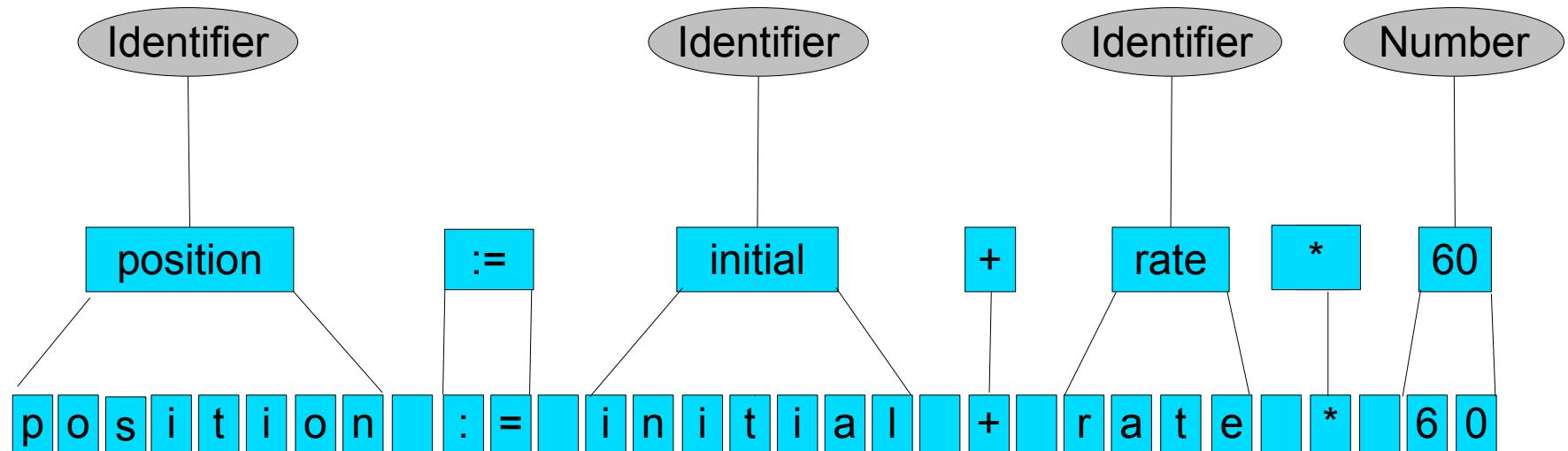
- Regular expressions define lexical syntax:
 - Literal characters: a,b,c,1,2,3
 - Character classes: [a-z], [0-9]
 - Operators: sequence (space), repetition (* or +), option (?)
- Examples:
 - Identifier: [a-z][a-zA-Z0-9]*
 - Number: [0-9]+
 - Floating constant: [0-9]*.[0-9]*([eE][0-9]+)

Lexical syntax

- Regular expressions can be implemented with a finite automaton
- Consider $[a-z][a-zA-Z0-9]^*$



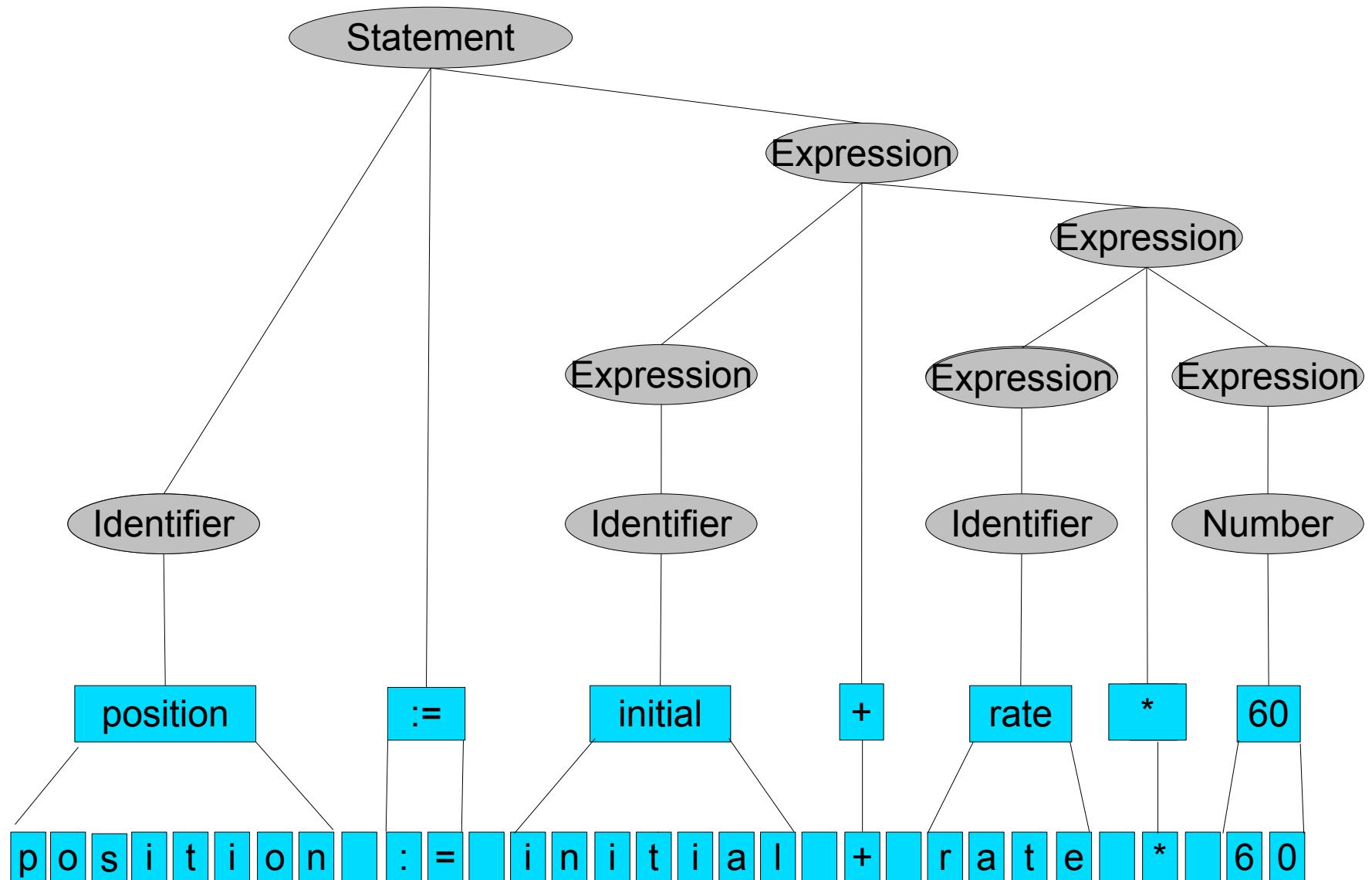
Lexical Tokens



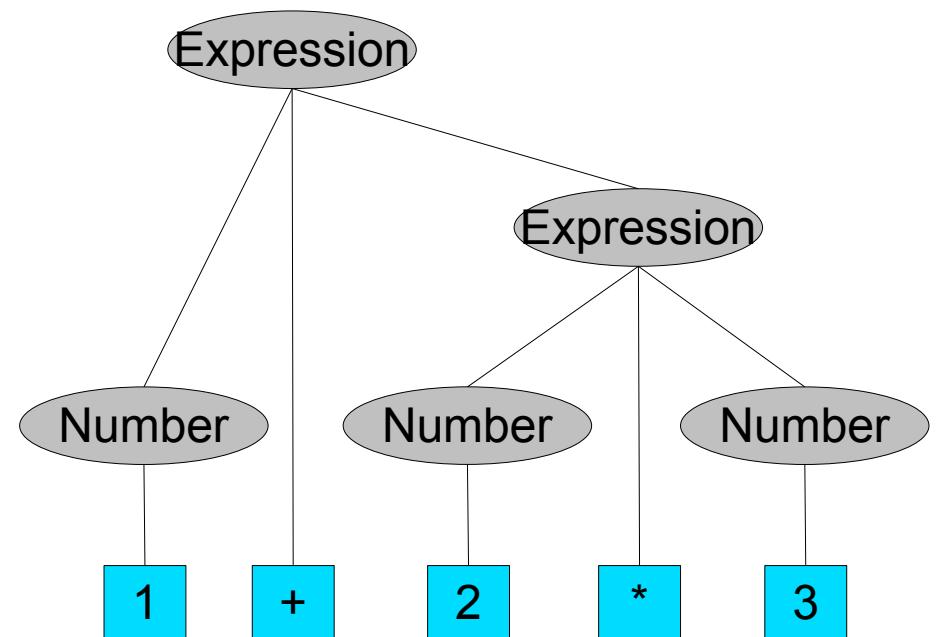
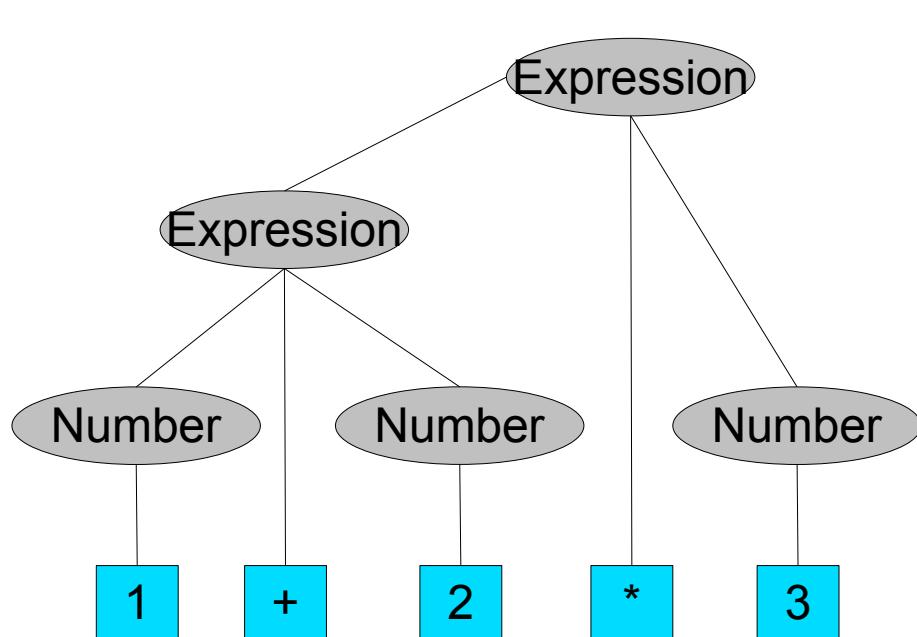
Context-free syntax

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

Parse Tree



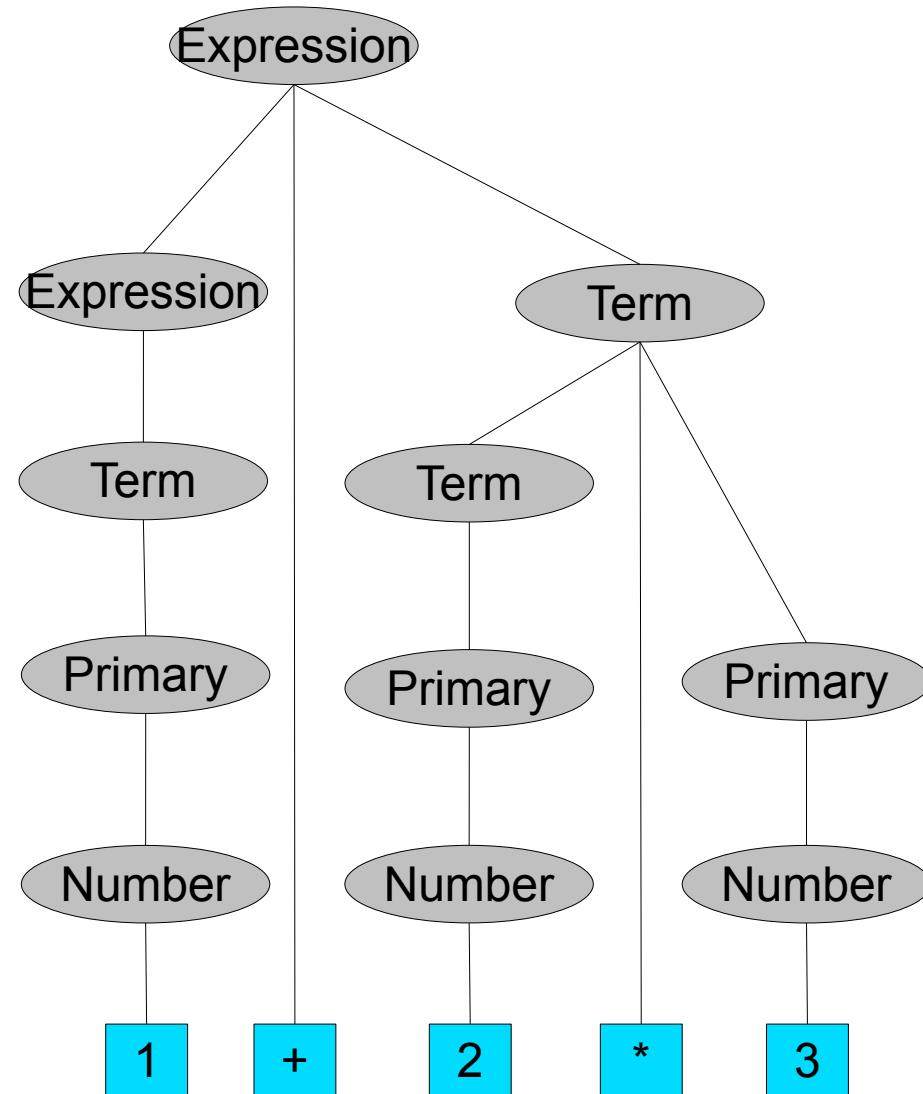
Ambiguity: one sentence, but several trees



Two solutions

- Add priorities to the grammar:
 - $* > +$
- Rewrite the grammar:
 - Expression \rightarrow Expression + Term
 - Expression \rightarrow Term
 - Term \rightarrow Term * Primary
 - Term \rightarrow Primary
 - Primary \rightarrow Number
 - Primary \rightarrow Identifier

Unambiguous Parse Tree



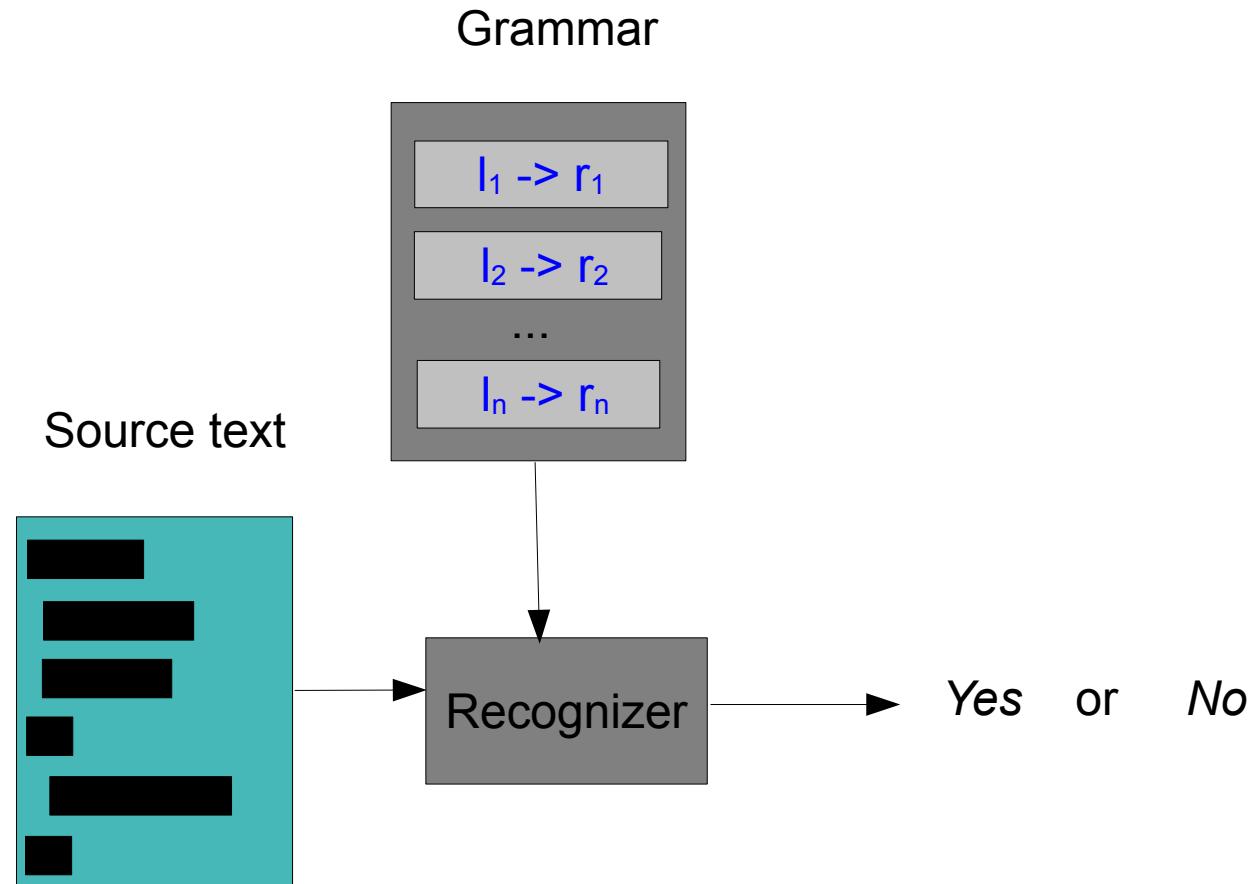
Some Grammar Transformations

- **Left recursive production:**
 - $A \rightarrow A\alpha | \beta$
 - Example: $\text{Exp} \rightarrow \text{Exp} + \text{Term} | \text{Term}$
- Left recursive productions lead to loops in some kinds of parsers (recursive descent)
- **Removal:**
 - $A \rightarrow \beta R$
 - $R \rightarrow \alpha R | \varepsilon$ (ε is the empty string)

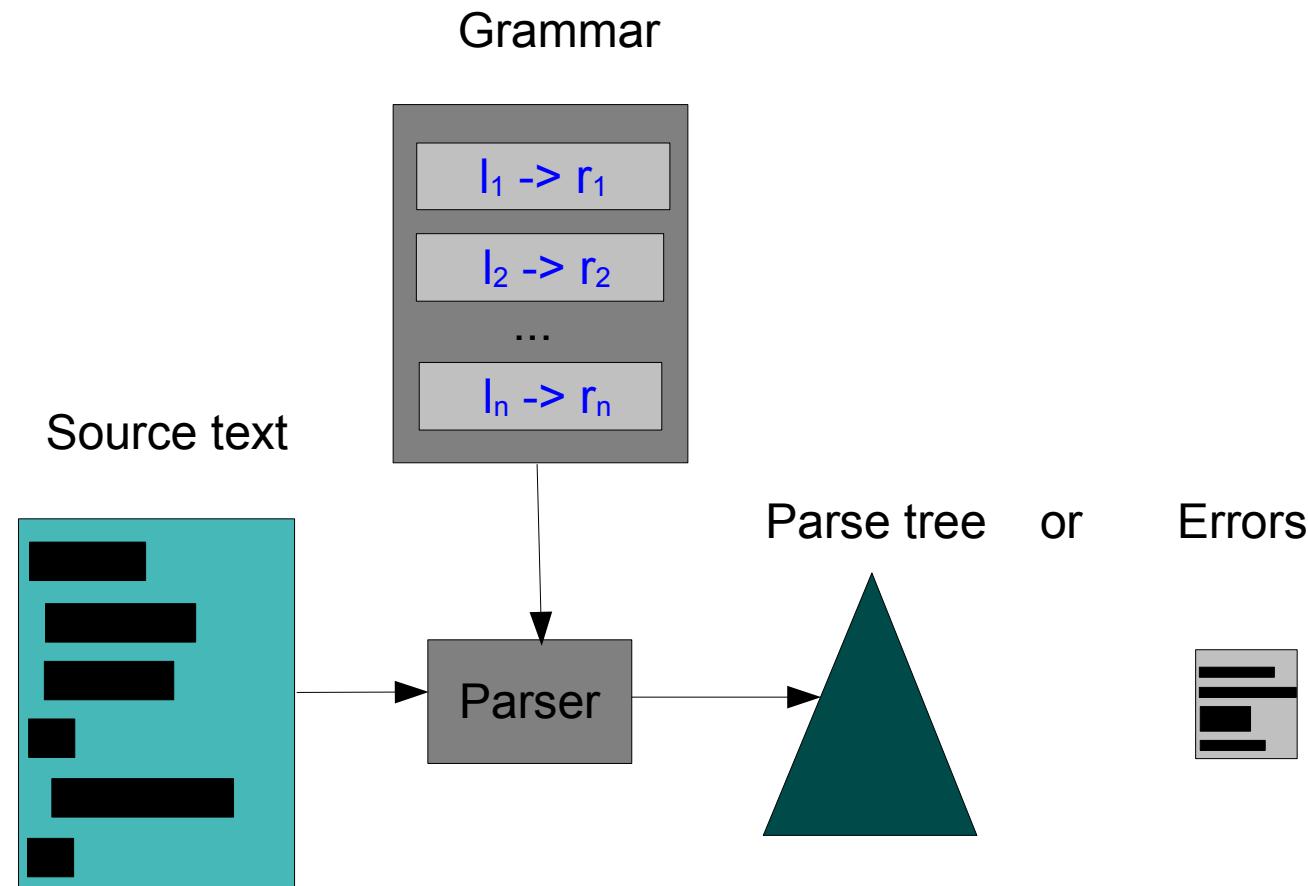
Some Grammar Transformations

- 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



A Parser

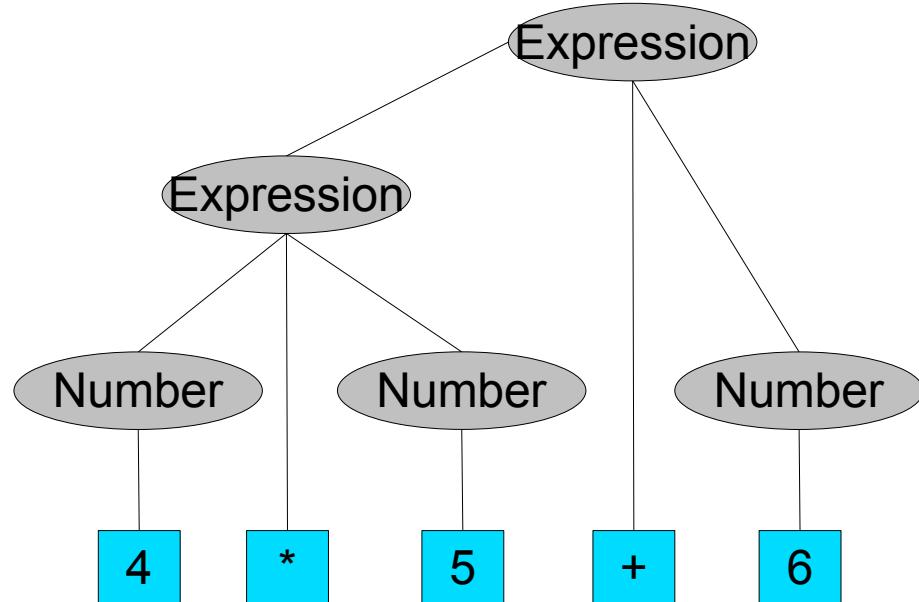


General Approaches to Parsing

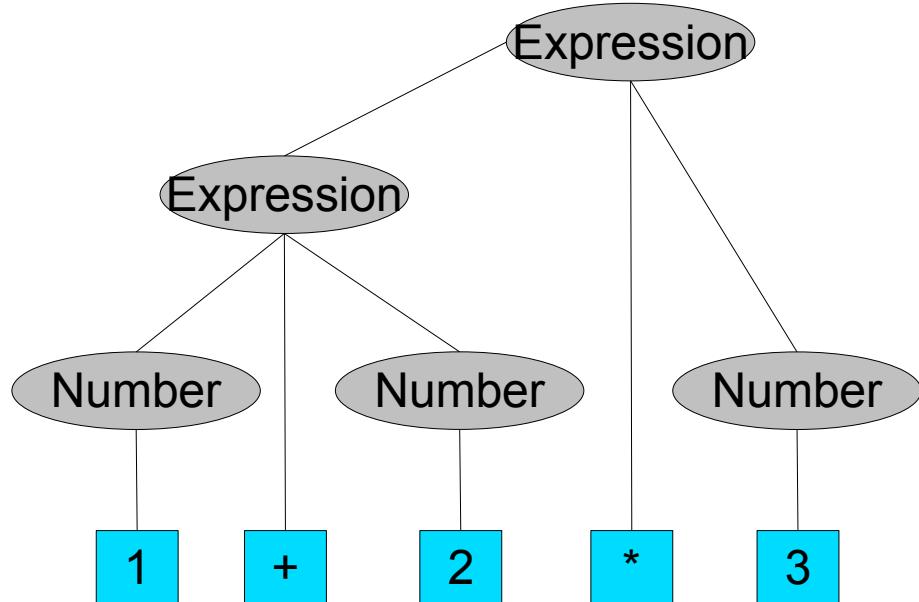
- **Top-Down (Predictive)**
 - Each non-terminal is a goal
 - Replace each goal by subgoals (= elements of rule)
 - Parse tree is built from top to bottom
- **Bottom-Up**
 - Recognize terminals
 - Replace terminals by non-terminals
 - Replace terminals and non-terminals by left-hand side of rule

Top-down versus Bottom-up

Top-down



Bottom-up



How to get a parser?

- Write parser manually
- Generate parser from grammar

Example

- Given grammar:
 - $\text{Expr} \rightarrow \text{Expr} + \text{Term}$
 - $\text{Expr} \rightarrow \text{Expr} - \text{Term}$
 - $\text{Expr} \rightarrow \text{Term}$
 - $\text{Term} \rightarrow [0-9]$
- Remove left recursion:
 - $\text{Expr} \rightarrow \text{Term Rest}$
 - $\text{Rest} \rightarrow + \text{Term Rest} \mid - \text{Term Rest} \mid \epsilon$
 - $\text{Term} \rightarrow [0-9]$

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

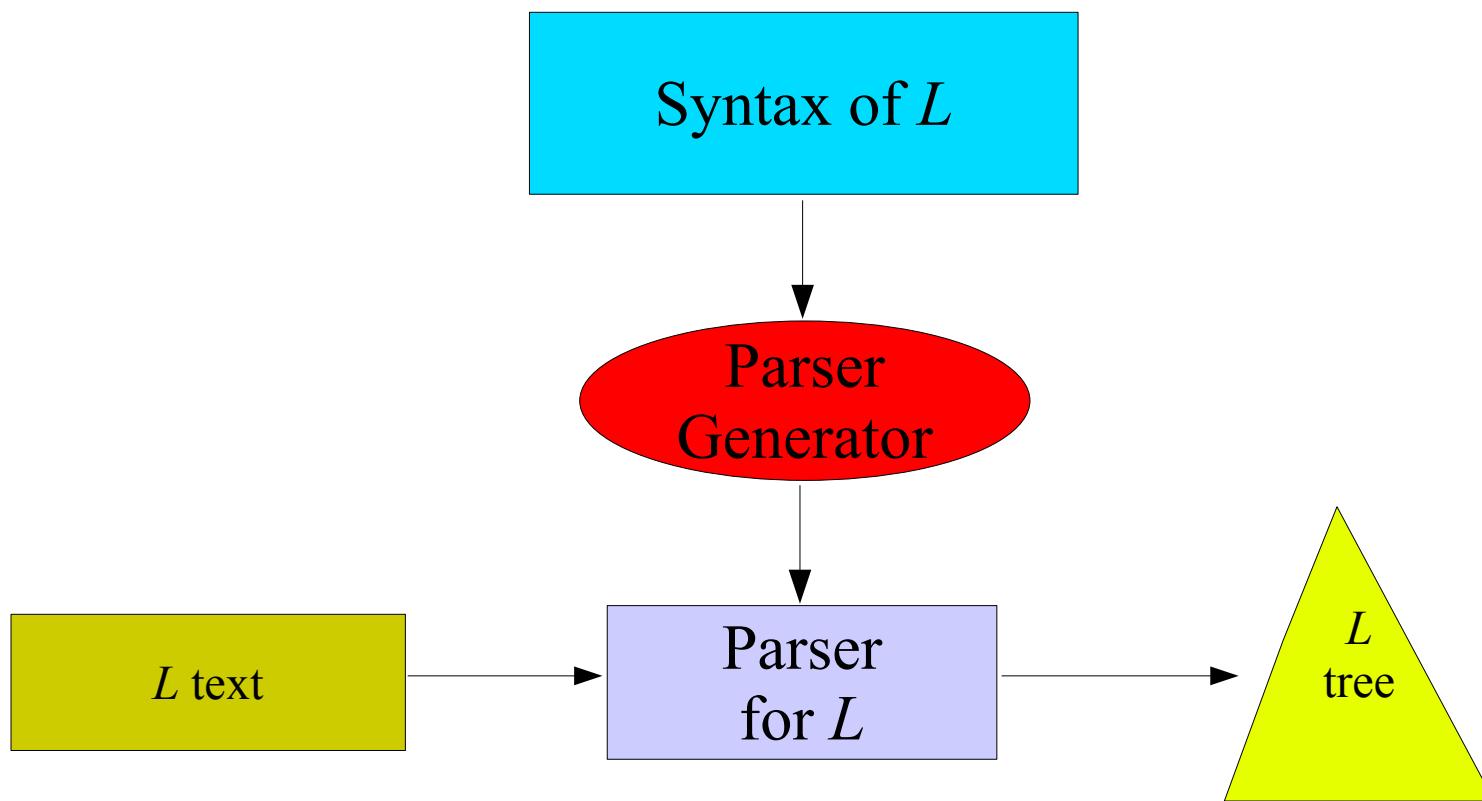
            Error();

}
```

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



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