Parsing with Grammars

UvA MSc SE: Software Construction 2015

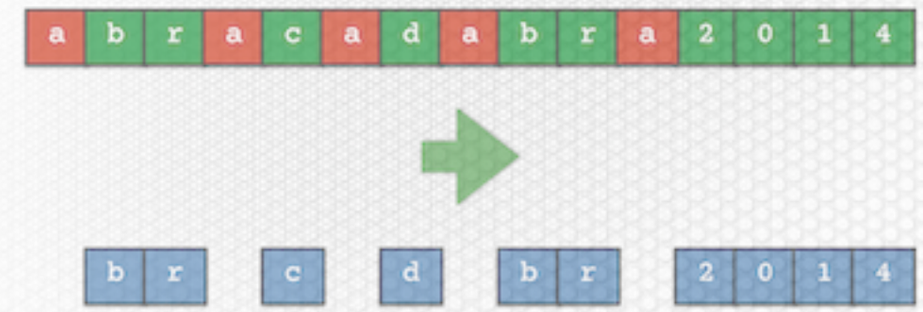
Dr. Vadim Zaytsev, Universiteit van Amsterdam

What is parsing?

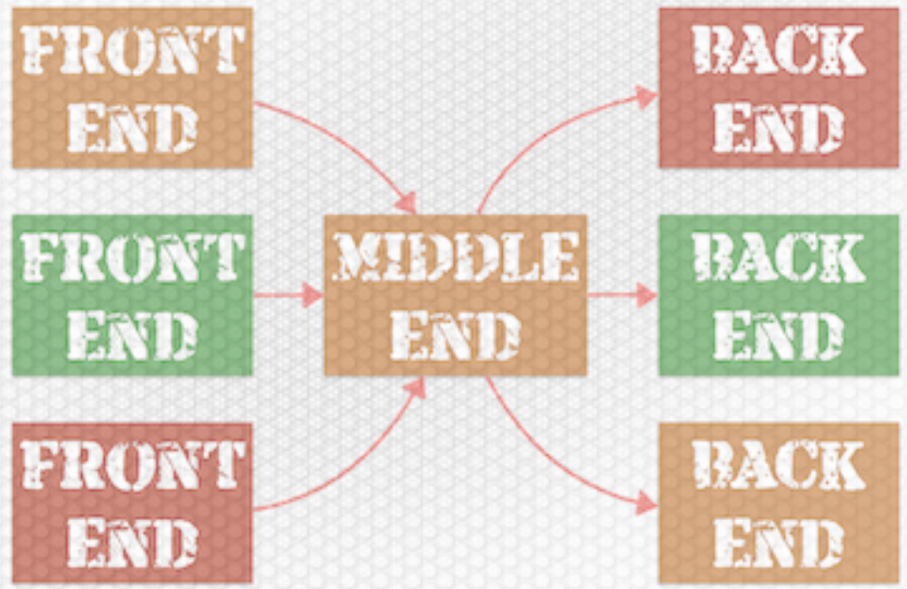
* deja vu from Software Evolution

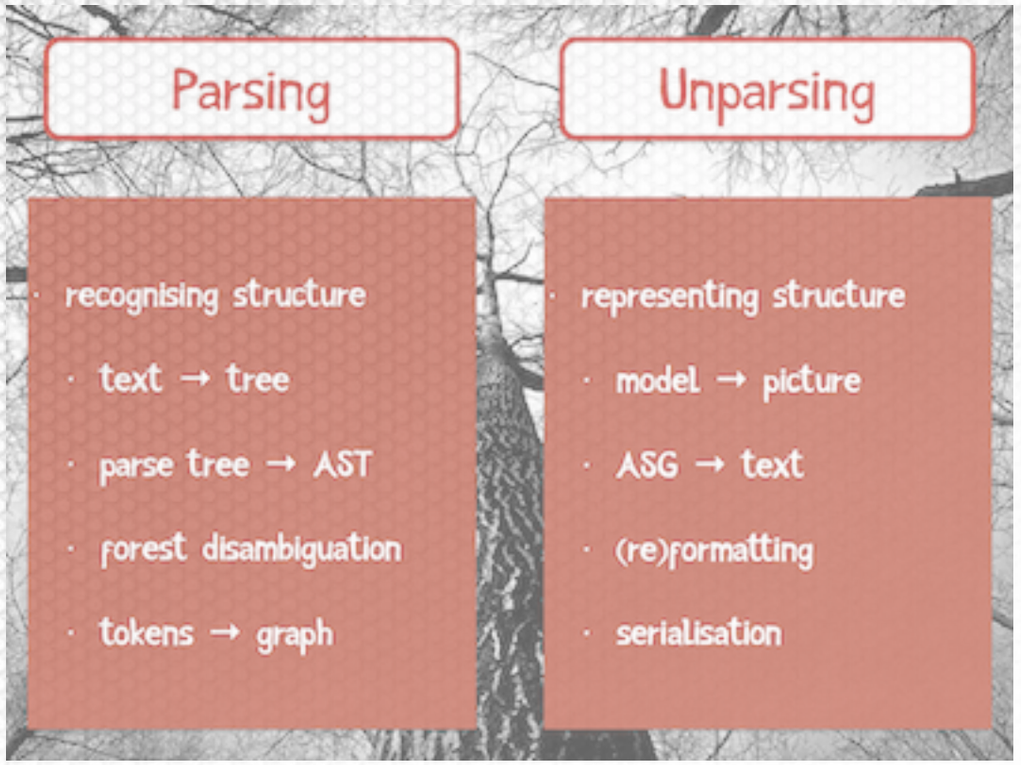
Parsing or not??

```
[t | t <- split(s," "), t!="a"]
```



Examples of grammarware?



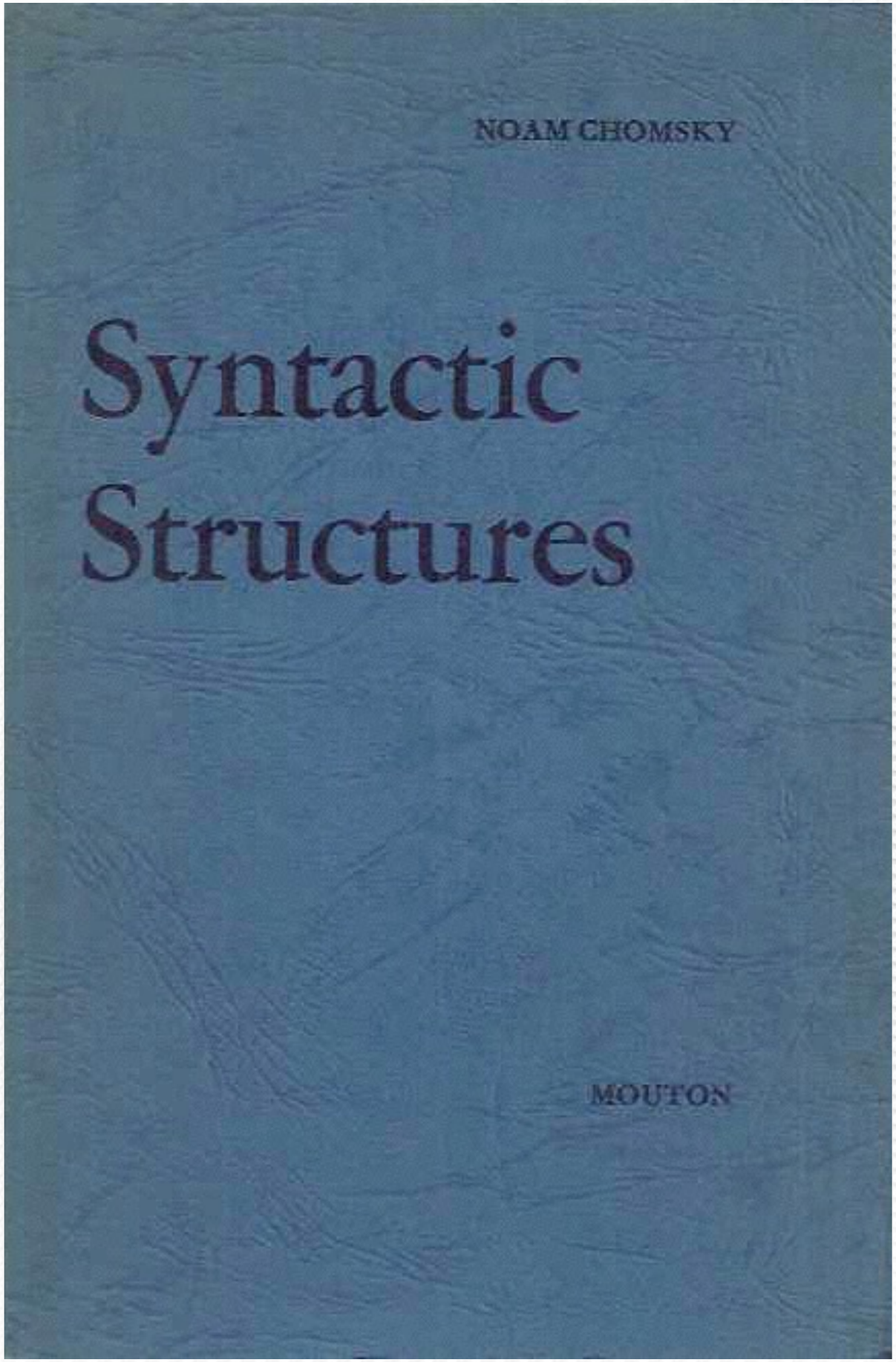


Software Parsing and Modelling, 3 December 2014

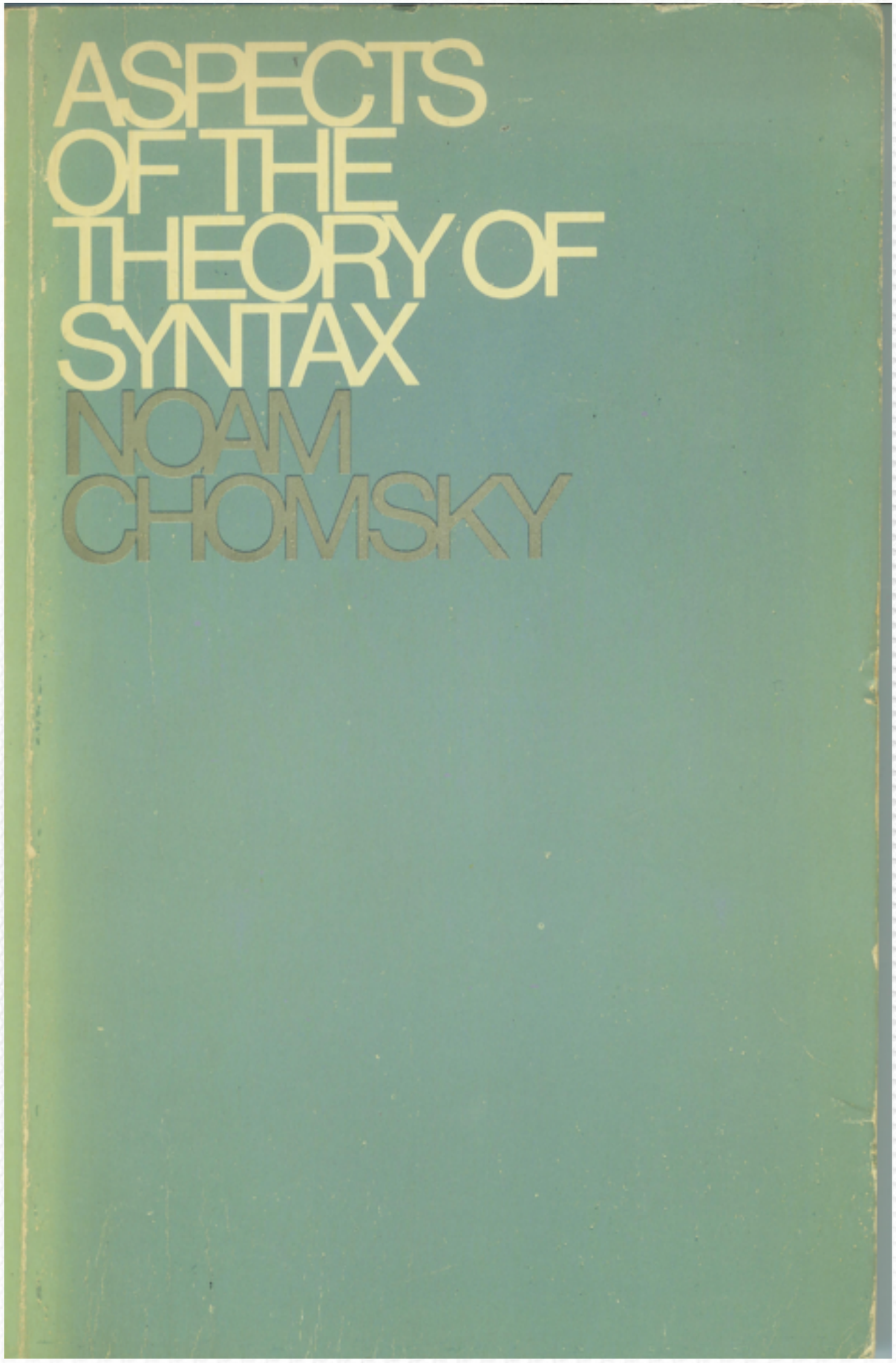
Grammars & parsing

are among the most

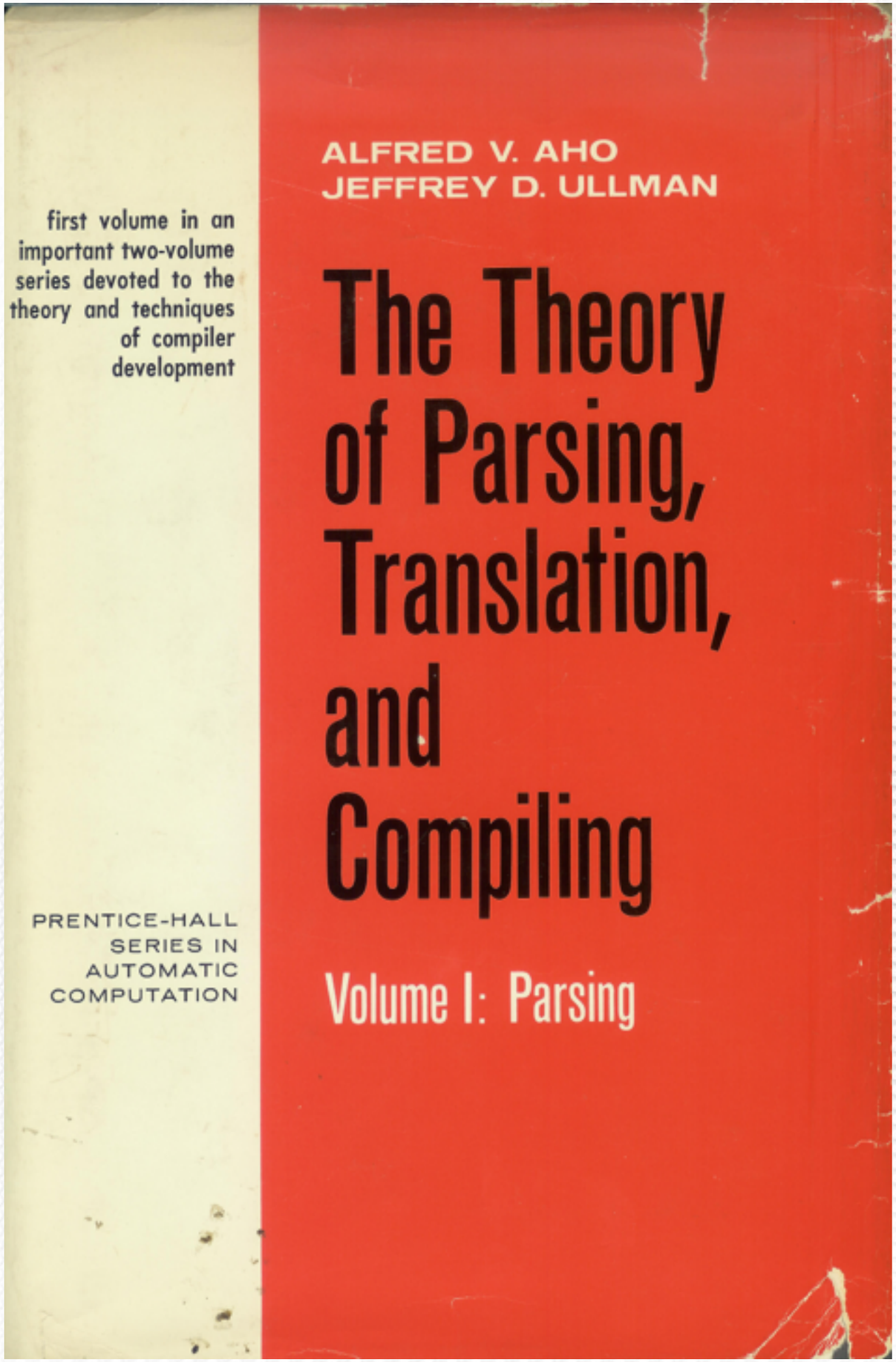
established areas of CS/SE



N. Chomsky, Syntactic Structures, 1957

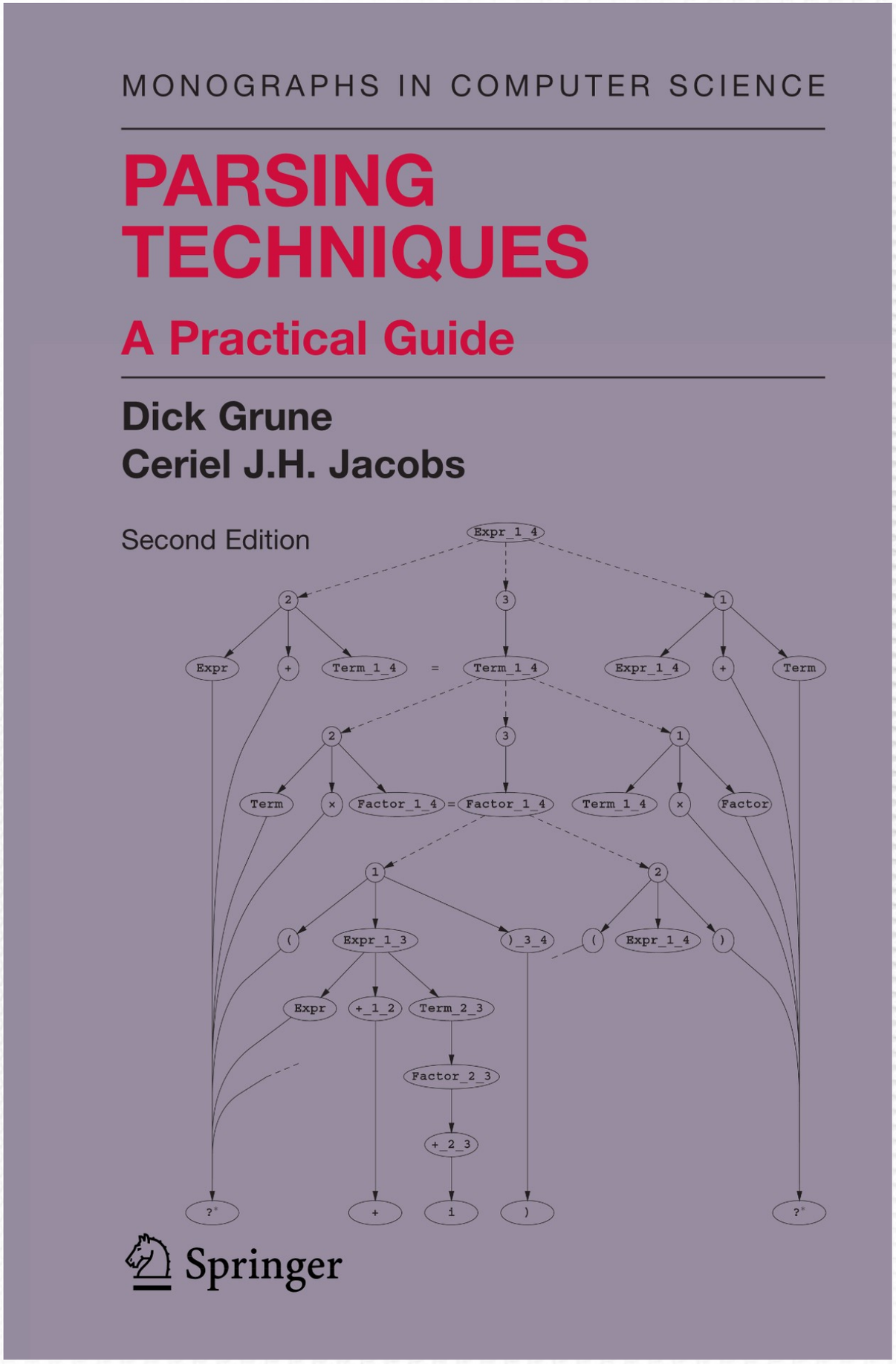


N. Chomsky, Aspects of the Theory of Syntax, 1965



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

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



D. Grune, C.J.H. Jacobs, Parsing Techniques: A Practical Guide, 2 ed, 2008

Why are grammars and parsing relevant?

Language

- Programming languages: C, Java, C#, JavaScript
- Markup languages: HTML, XML, TeX, Markdown, wikis
- Domain-specific languages: BibTeX, CSS, SQL, QL
- Data formats: JSON, log files, protocol data, bytecode
- …
- (formally: a set of strings)

How to define a language?

- List all the sentences!
- Infinite languages?
- Finite recipes = grammars
- Infinite grammars?
- Two level grammars

Adriaan van Wijngaarden

Example

- Valid sentences/programs/instances:
  - Alice
  - Alice and Bob
  - Alice, Bob and Coen
  - Alice, Bob, Coen and Daenerys
  - …
- How to define a recipe?

ABCD Grammar

Name → Alice

Name → Bob

Name → Coen

Name → Daenerys

Sentence → List End

List → Name

List → List , Name

, Name End → and Name

ABCD Grammar

Terminal symbols

Name → Alice

Name → Bob

Name → Coen

Name → Daenerys

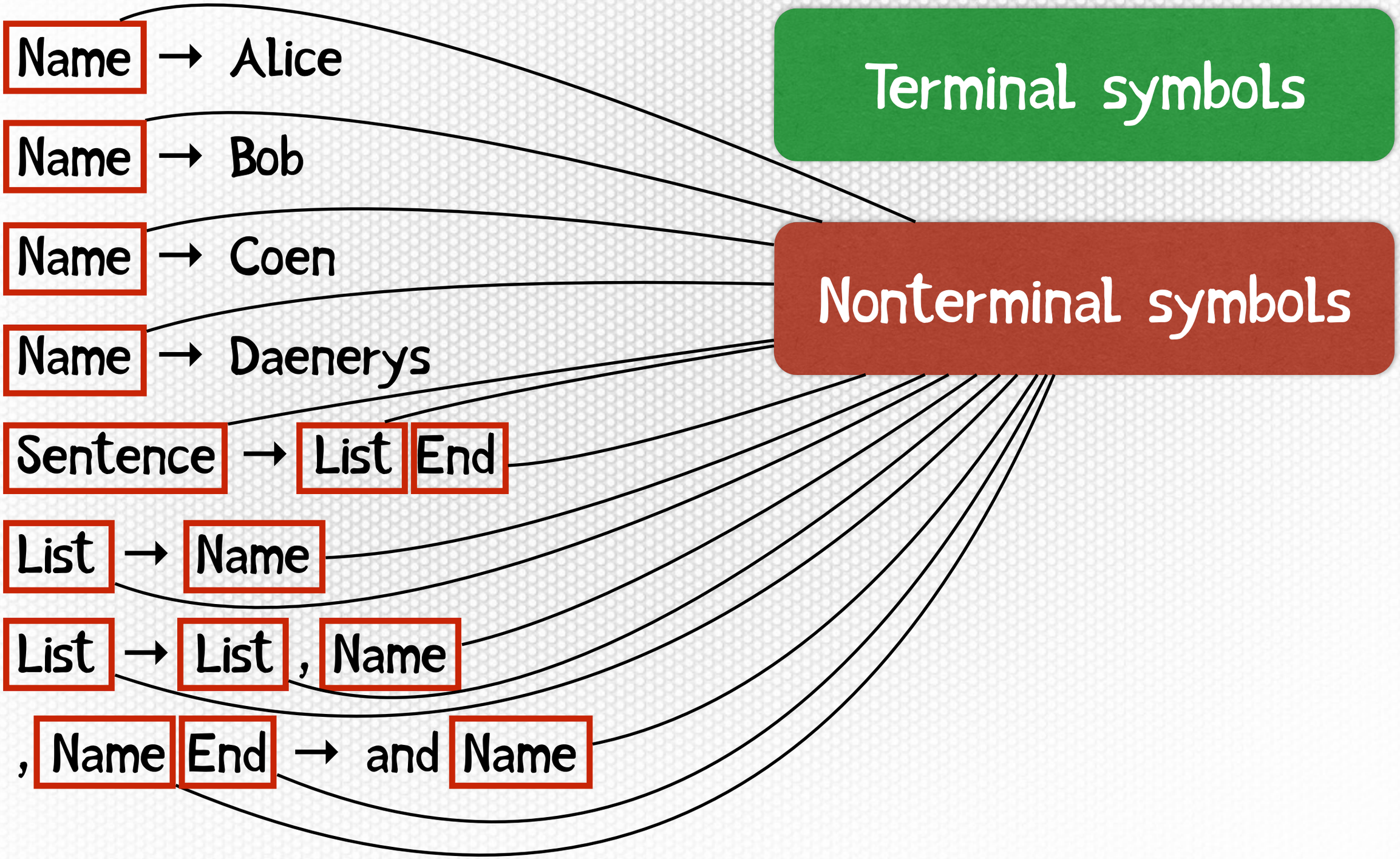
Sentence → List End

List → Name

List → List , Name

, Name End → and Name

ABCD Grammar



ABCD Grammar

Name → Alice

Name → Bob

Name → Coen

Name → Daenerys

Sentence → List End

List → Name

List → List , Name

, Name End → and Name

Terminal symbols

Nonterminal symbols

Starting symbol

ABCD Grammar

Name → Alice

Name → Bob

Name → Coen

Name → Daenerys

Sentence → List End

List → Name

List → List , Name

, Name End → and Name

Terminal symbols

Nonterminal symbols

Starting symbol

Production rules

Using ABCD

- Alice and Bob
- Sentence → List End →
  List , Name End →
  Name , Name End → Name
  and Name → Alice and Name
  → Alice and Bob
- (generative semantics)

- Alice and Bob
- Alice and Bob → Name and
  Bob → Name and Name →
  Name , Name End →
  List , Name End → List End
  → Sentence
- (analytic semantics)

Production

Notations

- Name → Alice | Bob | Coen | Daenerys
- Name → "Alice" | "Bob" | "Coen" | "Daenerys"
- Name → “Alice” | “Bob” | “Coen” | “Daenerys”
- ⟨Name⟩ → Alice | Bob | Coen | Daenerys
- ⟨Name⟩ → “Alice” | “Bob” | “Coen” | “Daenerys”

Notations

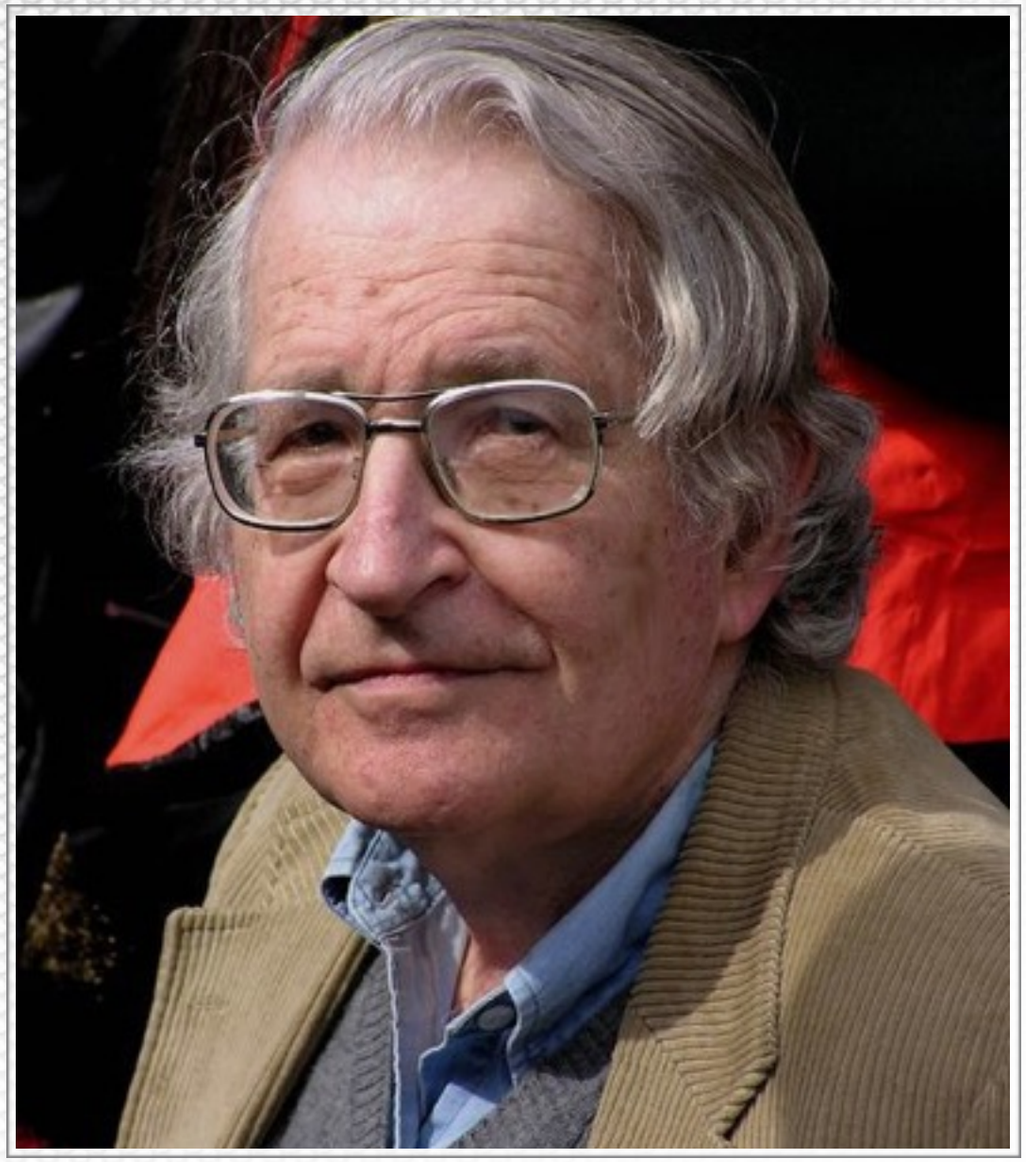
- List → List , Name
- ⟨List⟩ ::= ⟨List⟩ "," ⟨Name⟩ ;
- List "," Name -> List
- List <- List ',' Name
- define List [List] , [Name] end define
- syntax List = List "," Name;
- List -> List ',' Name : ['$1' | '$3'].

Common metaconstructs

- Optional symbols
  - A?, [A]
- Zero or more (Kleene star)
  - A*, {A}
- One or more
  - A+
- Choice (disjunction)
  - A | B, A / B
- Less common (careful!)
  - conjunction
  - negation
  - exact repetition
  - reference naming
  - priorities

Chomsky-Schützenberger 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 $A \rightarrow aB$



Noam Chomsky. On Certain Formal Properties of Grammars, Information & Control 2(2):137–167, 1959.

CFG for ABCD

⟨Name⟩ → “Alice” | “Bob” | “Coen” | “Daenerys”

⟨Sentence⟩ → ⟨Name⟩ | ⟨List⟩ “and” ⟨Name⟩

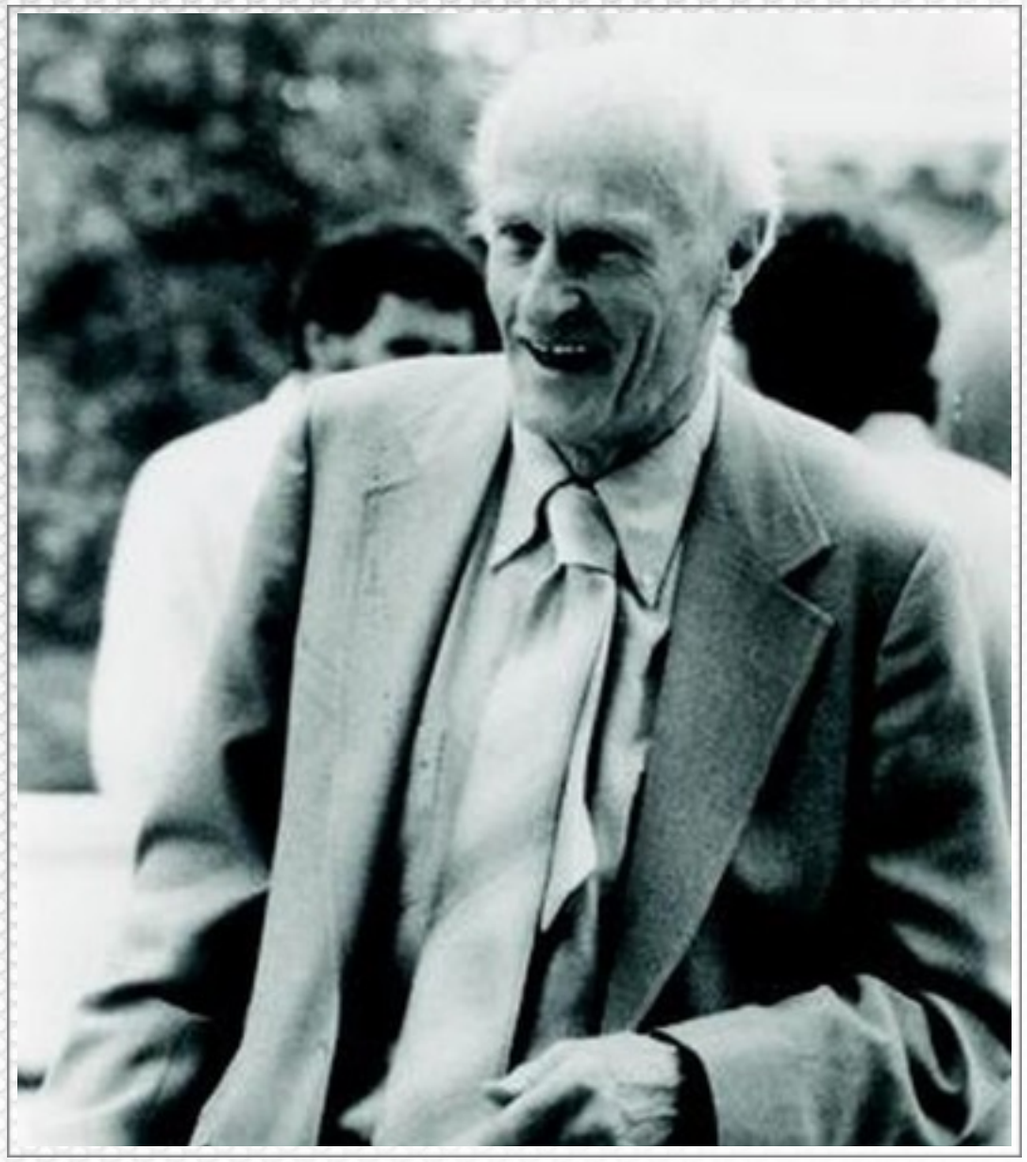
⟨List⟩ → ⟨Name⟩ “,” ⟨List⟩ | ⟨Name⟩

⟨List⟩ → ⟨Name⟩ (“,” ⟨Name⟩)*

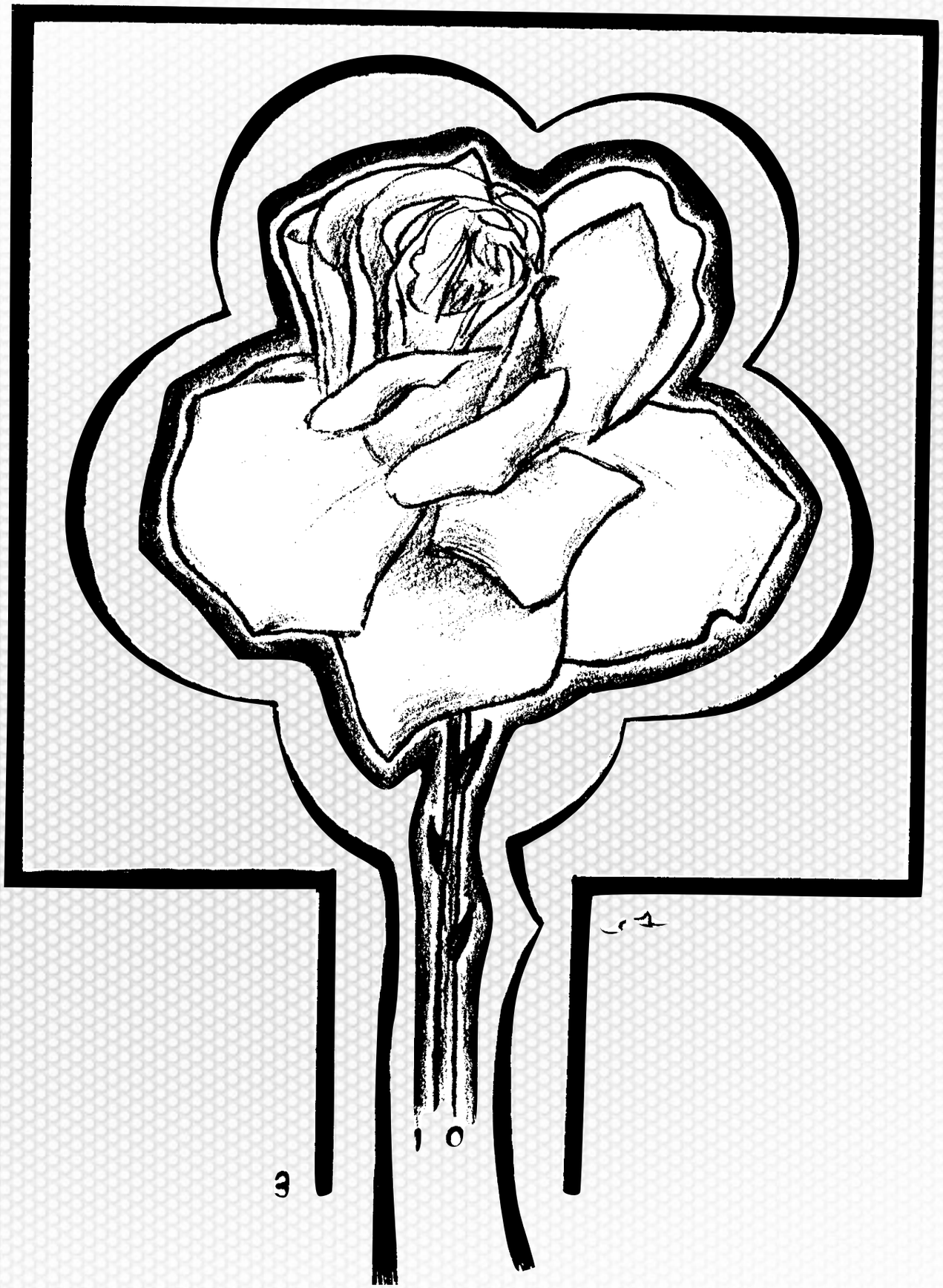
⟨List⟩ → {⟨Name⟩ “,”}+

Regexp for ABCD

^\w+((, \w+)* and \w+)?$



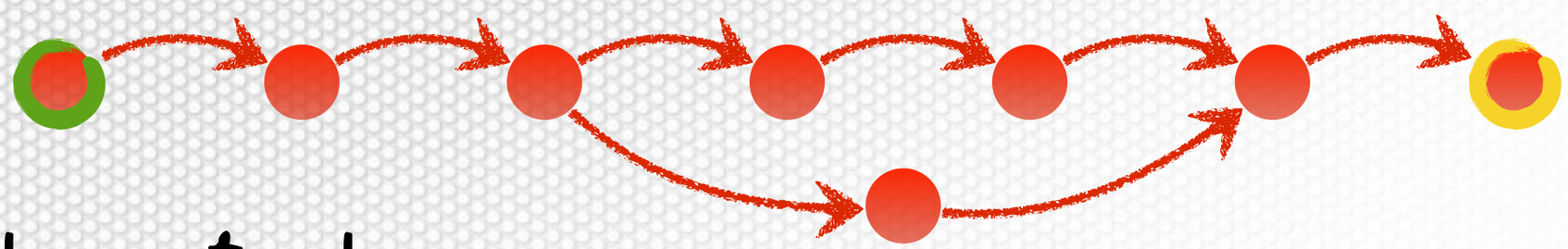
S. C. Kleene, Representation of Events in Nerve Nets and Finite Automata. In Automata Studies, pp. 3–42, 1956.
photo from: Konrad Jacobs, S. C. Kleene, 1978, MFO.



Rose by Arwen Grune; p.58 of Grune/Jacobs' "Parsing Techniques", 2008

Finite world

- Explicitly given lists
- Acyclic automata
- Finite choice grammars (non-recursive, non-iterating)

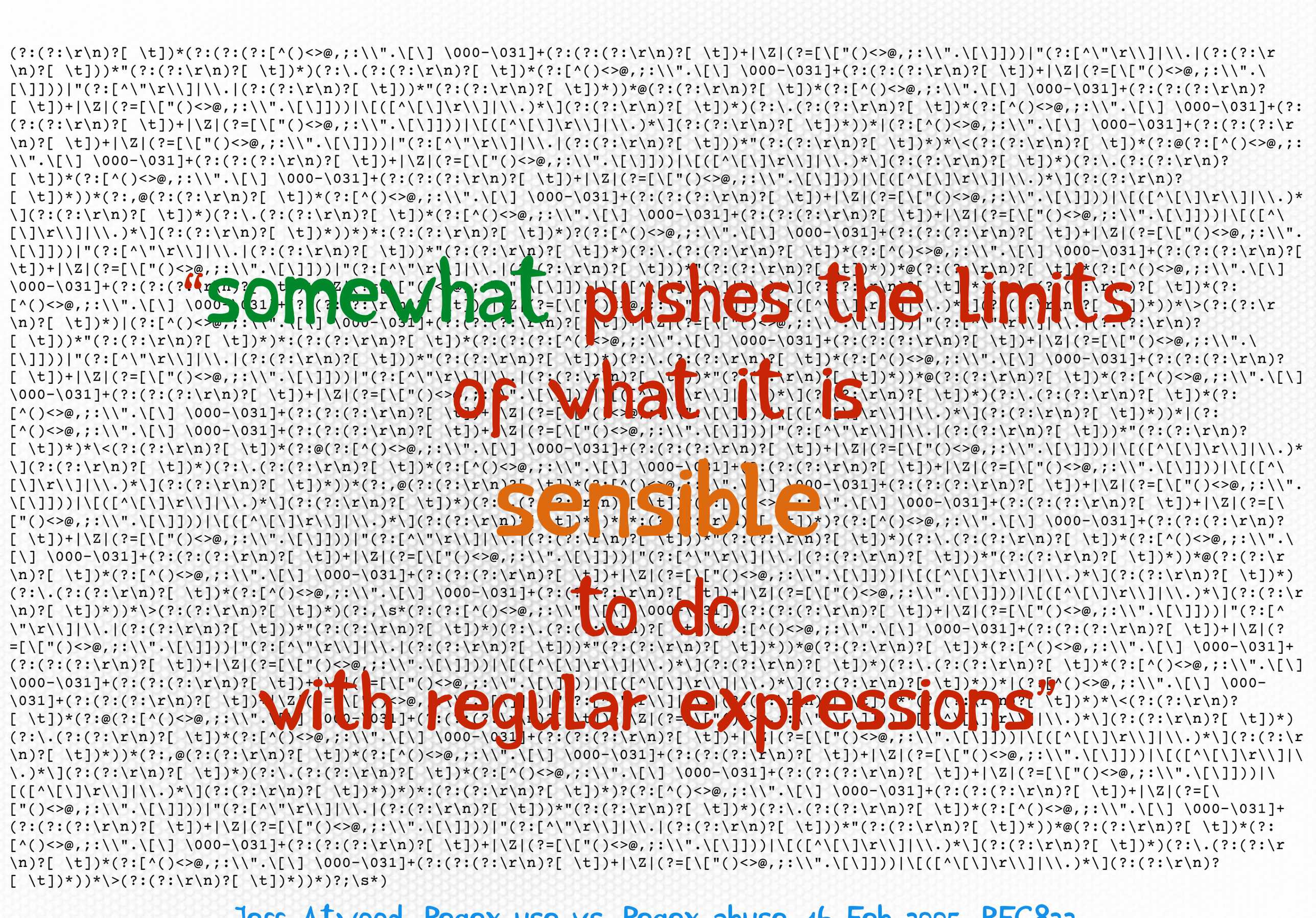


- i.e., users, keywords, postcodes

Regular world

- Regular expressions
- Finite automata
- Grammars:
  - A → a
  - A → aB

- i.e., substring search, substring replace, counting

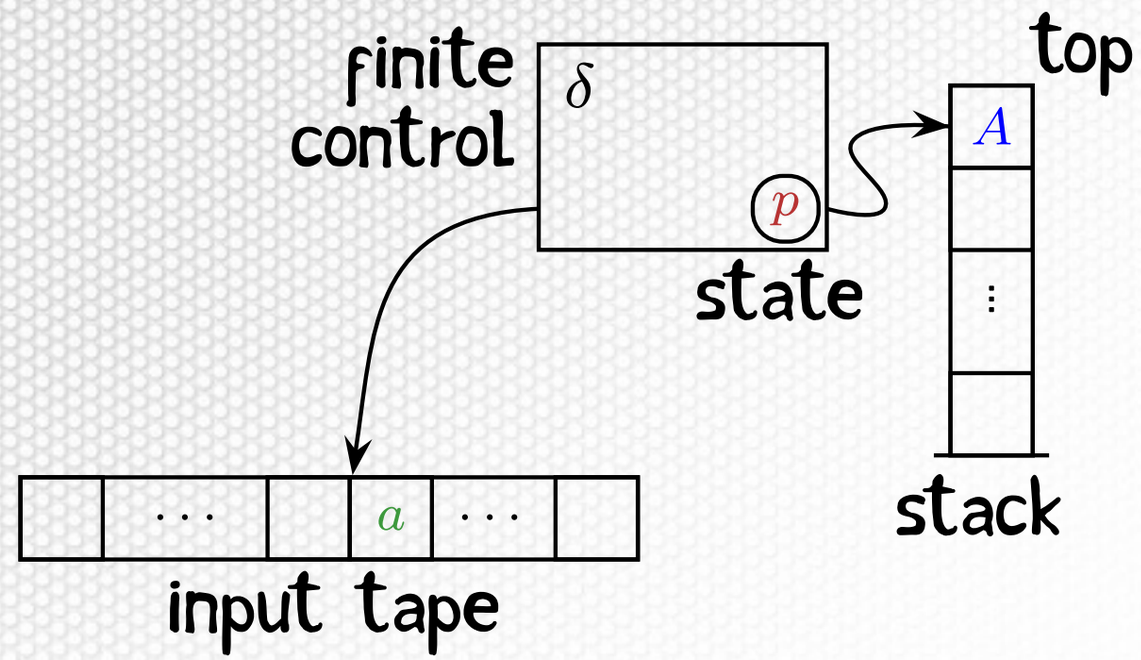


Jeff Atwood, Regex use vs. Regex abuse, 16 Feb 2005. RFC822.

Paul Warren, Mail::RFC822::Address: regexp-based address validation, 17/09/2012.

Context-free world

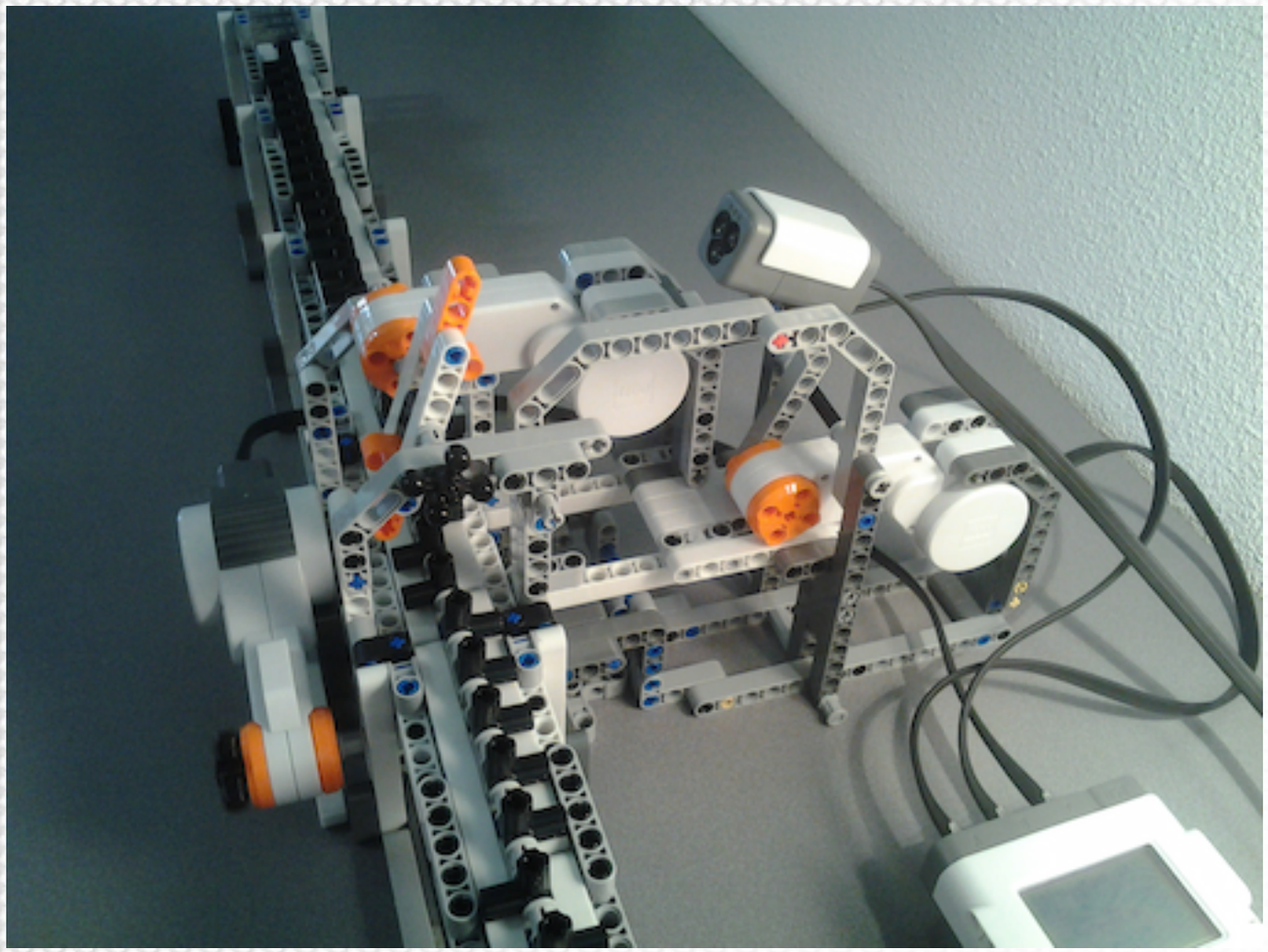
- Grammarware and software languages
- Nondeterministic pushdown automata
- Grammars:
  - $A \to \gamma$
    - $A \to BC$
    - $A \to a$
    - $A \to \varepsilon$
- i.e., parsing, pretty-printing, etc



Jochgem, https://commons.wikimedia.org/wiki/File:Pushdown-overview.svg, CC-BY-SA, 2008.

Context-sensitive world

- Computer
- Linear-bounded automata
- Grammars:
  - $\alpha A \beta \rightarrow \alpha \gamma \beta$
- i.e., anything practical



http://www.legoturingmachine.org

Unrestricted world

- Imaginary machines
- Turing machine, λ-calculus, semi-Thue rewriting systems, Lindenmayer systems, Markov algorithm, ...
- Grammars:
  - $\alpha \rightarrow \beta$
- i.e., almost anything

recognising is impossible

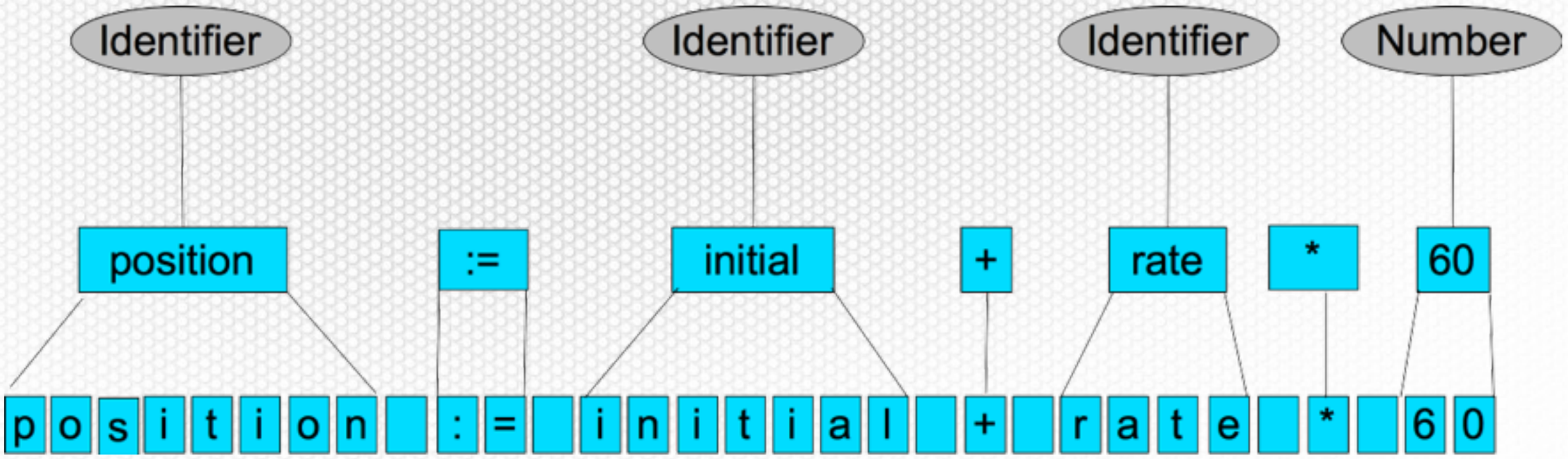
In practice...

- Regular grammars are used for lexical analysis
  - keywords
  - constants
  - comments (if not nested)
- Context-free grammars: for structured/nested constructs
  - class declaration
  - if statement
  - ...
- Everything else: annotations + hacking

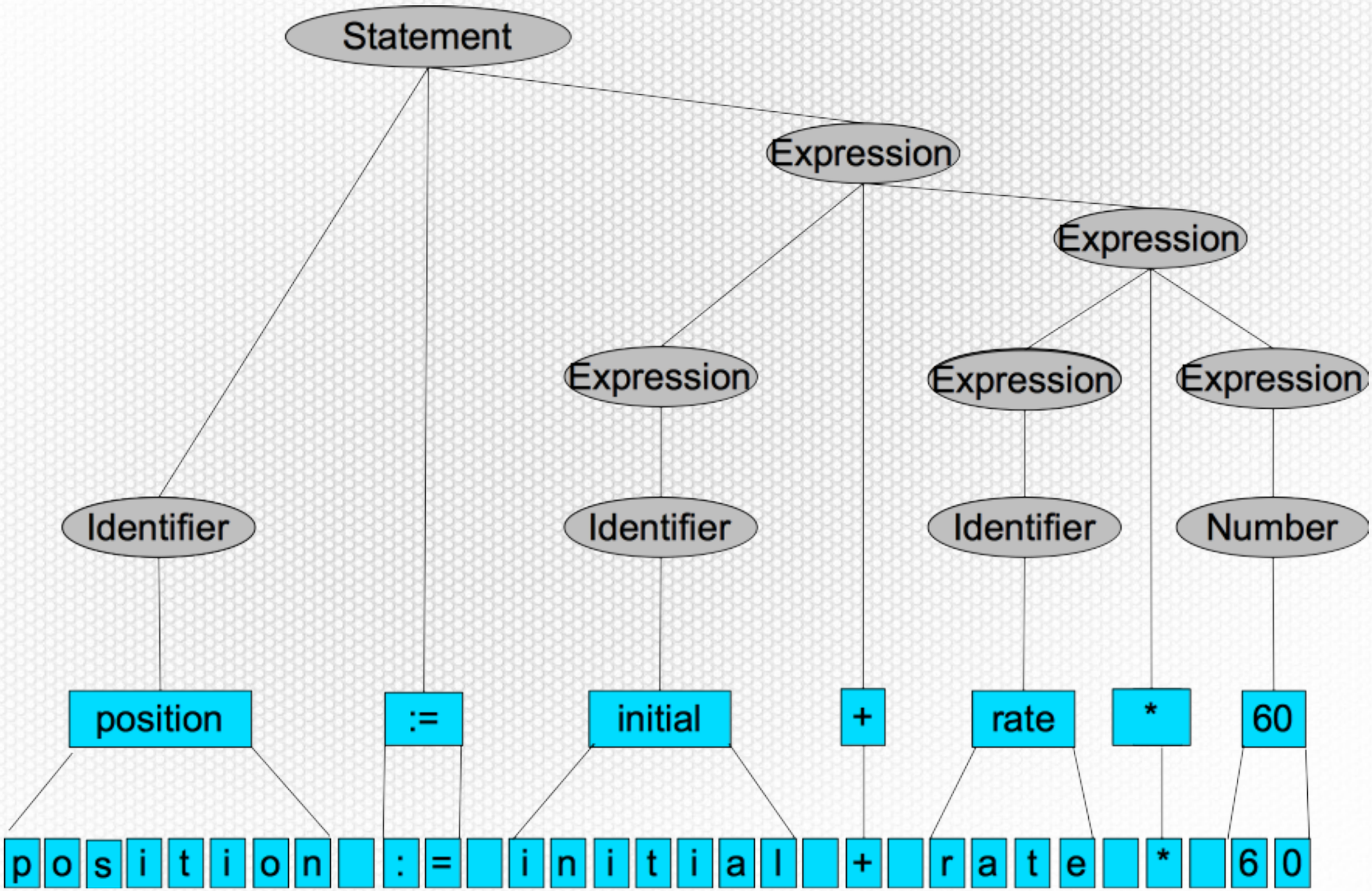
A sentence

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

Lexical tokens



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

PEG

- Parsing Expression Grammars, introduced in 2004
  - Analytic grammars; top-down unambiguous recognition
  - Explicit backtracking, ordered disjunction
  - Linear parsing time (with memoisation)
  - Do not fit into the Chomsky-Schützenberger hierarchy
  - Conjunction and negation are purely lookahead-based

PEG example

- S ← &X 'a'+ Y
- X ← Z 'c'
- Z ← 'a' Z? 'b'
- Y ← 'b' Y? 'c'

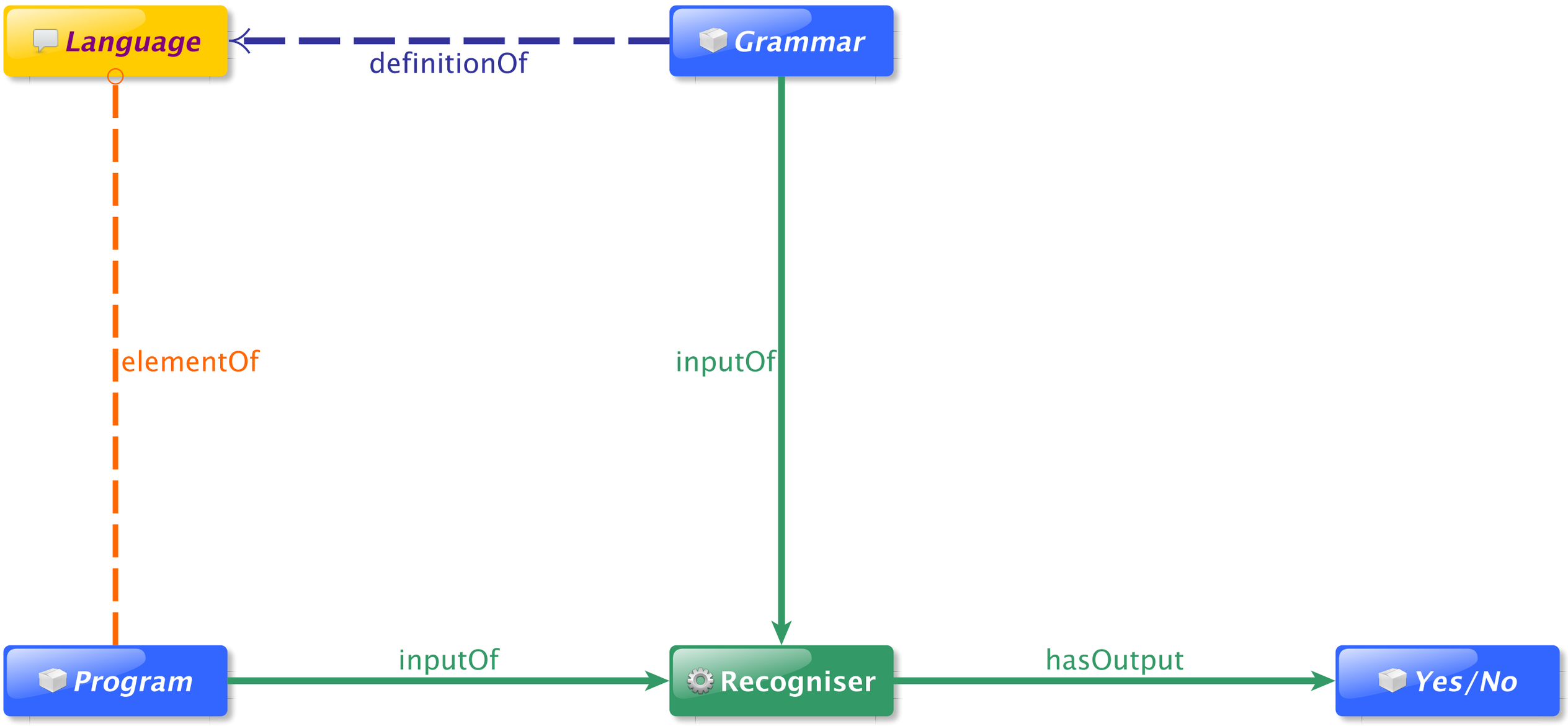
$\{a^n b^n c^n \mid n>0\}$

fixed from https://en.wikipedia.org/wiki/Parsing_expression_grammar

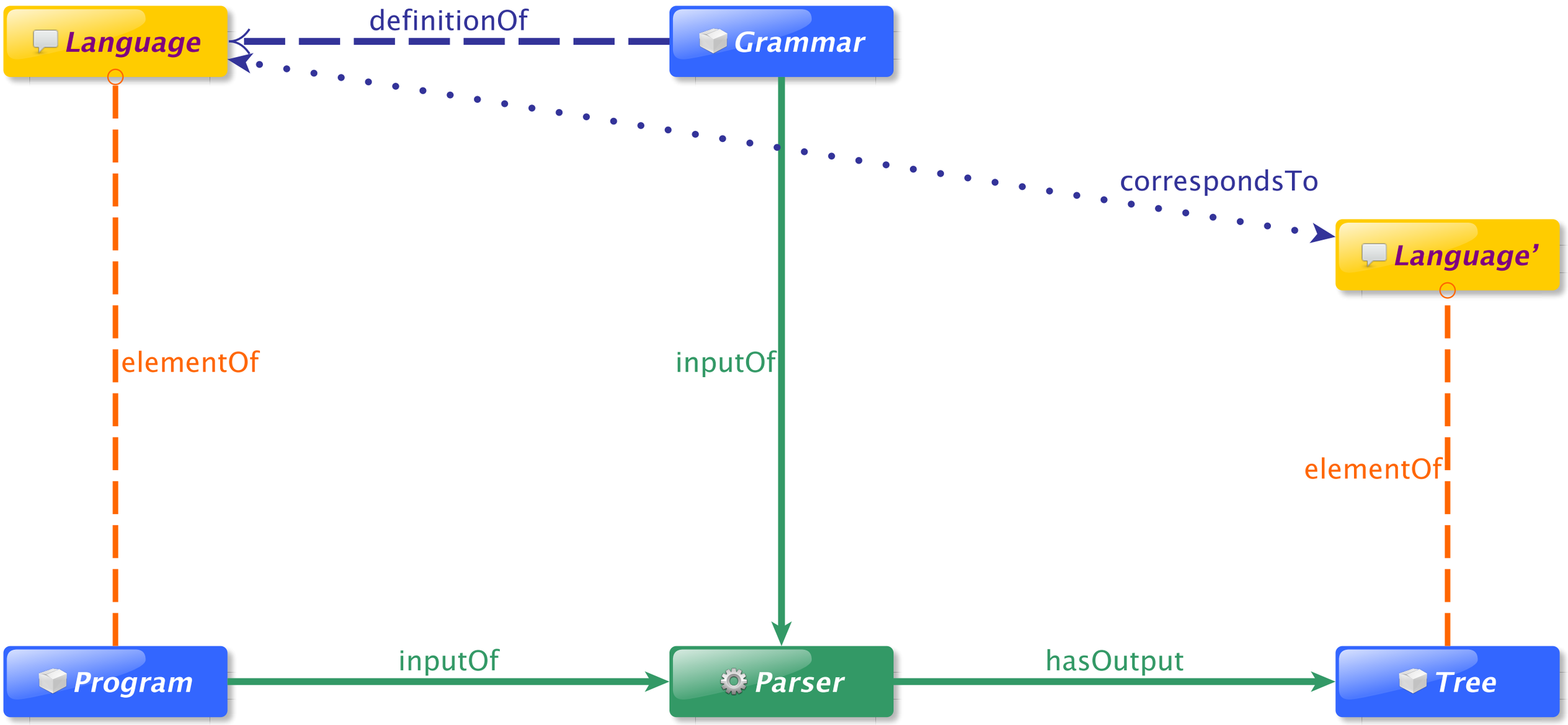
What is a parser?

(recogniser, compiler, ...)

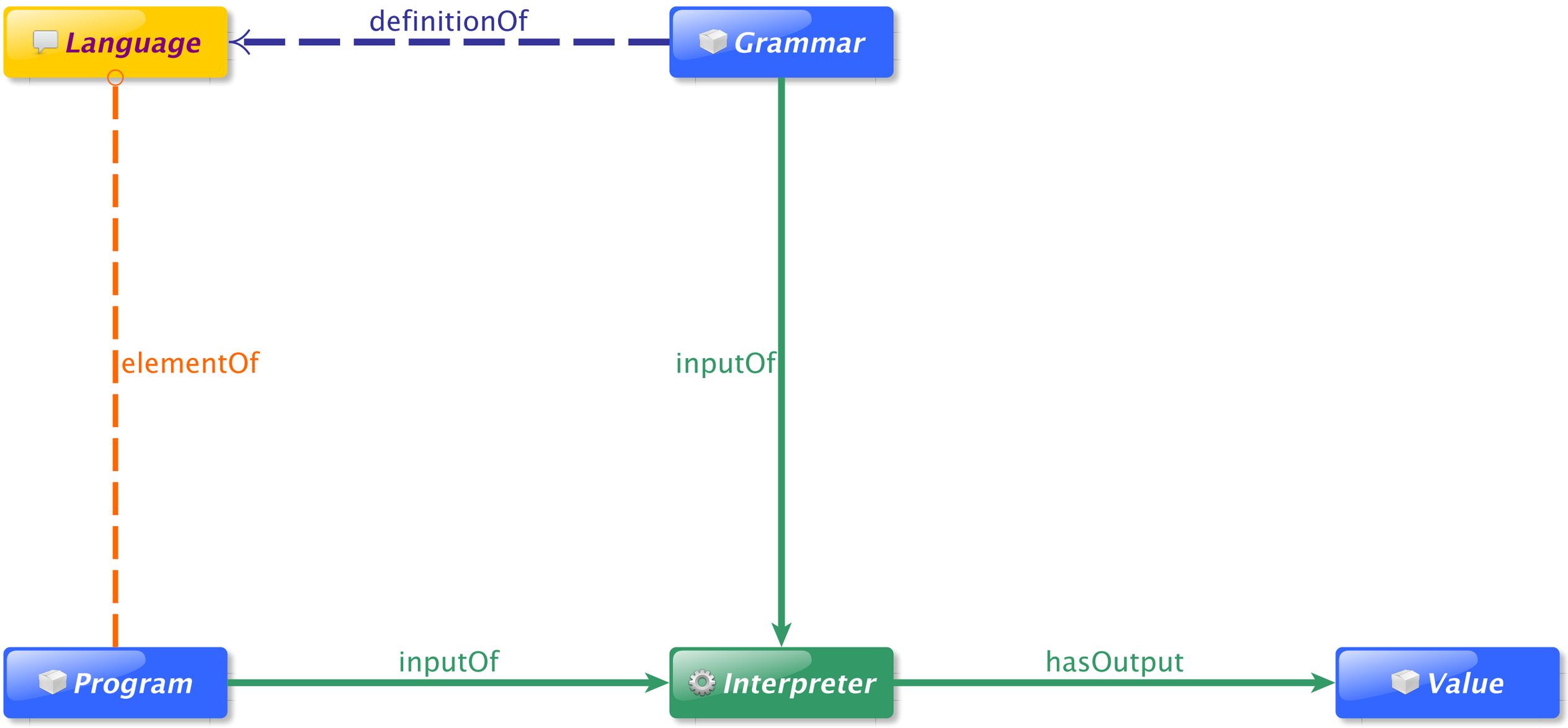
Recogniser


Language
definitionOf
Grammar
elementOf
inputOf
Program
inputOf
Recogniser
hasOutput
Yes/No

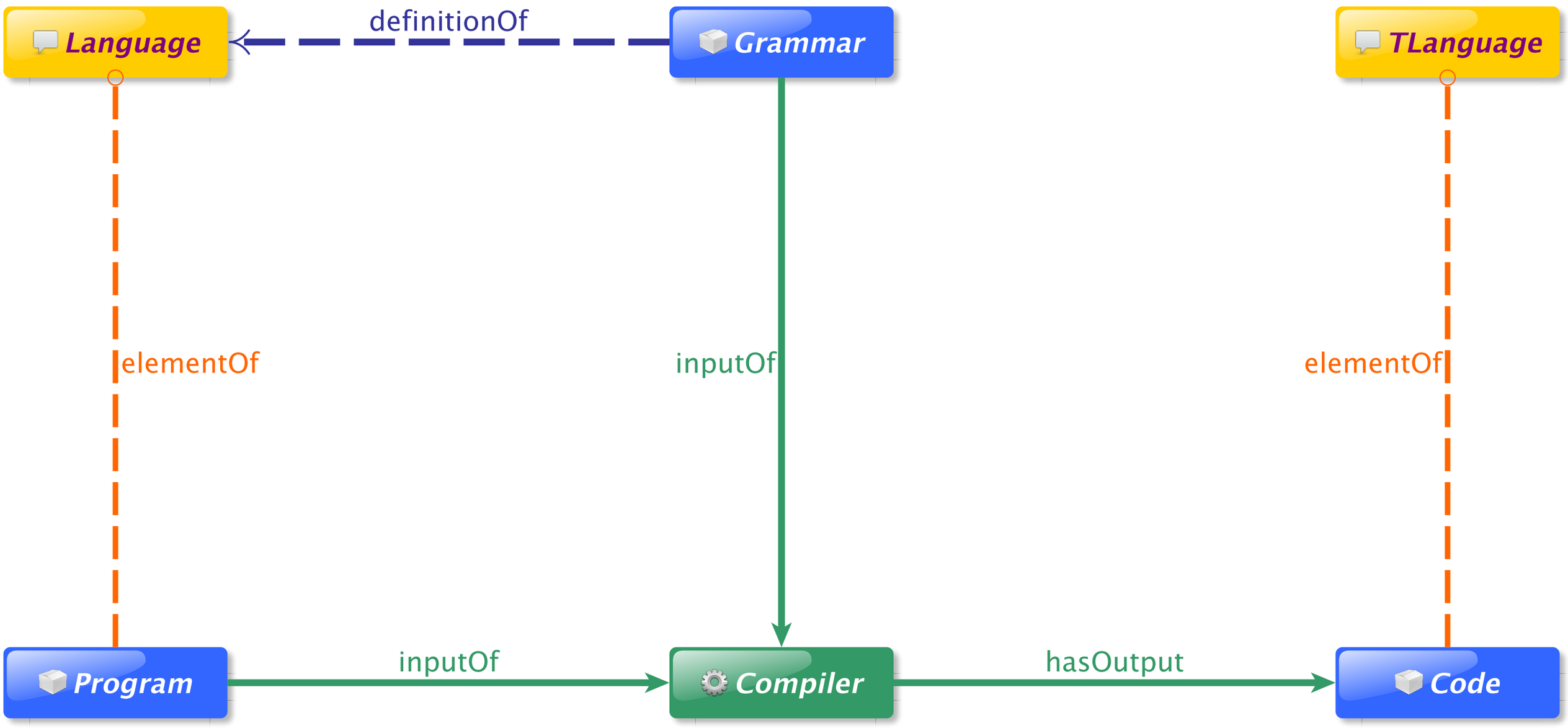
Parser


Language
definitionOf
Grammar
correspondsTo
Language'
elementOf
inputOf
elementOf
Program
inputOf
Parser
hasOutput
Tree

Interpreter



Compiler


Language
definitionOf
Grammar
TLanguage
elementOf
inputOf
elementOf
Program
inputOf
Compiler
hasOutput
Code

Guess what

gcc?

A: Recogniser

B: Parser

C: Interpreter

D: Compiler

Linker?

A: Recogniser

B: Parser

C: Interpreter

D: Compiler

Java

A: Recogniser

B: Parser

C: Interpreter

D: Compiler

Eclipse

A: Recogniser

B: Parser

C: Interpreter

D: Compiler

make

A: Recogniser

B: Parser

C: Interpreter

D: Compiler

HTML validator

A: Recogniser

B: Parser

C: Interpreter

D: Compiler

Parsing

Bottom-up parsing

- 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 parsing

- 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, rec. descent, Packrat, Earley

How to parse with a grammar?

- Write a parser manually
  - good error handling
  - possible fine-tuning
  - a lot of work (seriously, years)
- Generate with a parser generator
  - less work (maybe)
  - complex, rigid, idiosyncratic frameworks
  - difficult error handling

Manually: recursive descent

- Grammar:
  - A ::= x B C;
- Parser:
  - A() { match('x'); B(); C(); }

Manually: recursive descent

- Grammar:
  - A ::= x B C | y D | ε ;
- Parser:
  - A() { if (lookahead= = ‘x’) { match(‘x’); B(); C(); }
    else if (lookahead= = ‘y’) { match(‘y’); D(); }
    else ; }

Manually: recursive descent

- Grammar:
  - A ::= x B C | x D ;
- Parser:
  - A() { if (lookahead == 'x') { match('x'); B(); C(); }
    else { match('x'); D(); } }

solved by backtracking:
try and if fail, return

Left factoring

- For some parsers, this is inefficient:
  - S -> if E then S else S | if E then S
- Can be rewritten as this:
  - S -> if E then S P
  - P -> else S | ε

sometimes the only way

Manually: recursive descent

- Grammar:
  - Expr ::= Expr ‘+’ Expr ;
- Parser:
  - Expr() { Expr(); match(‘+’); Expr(); }

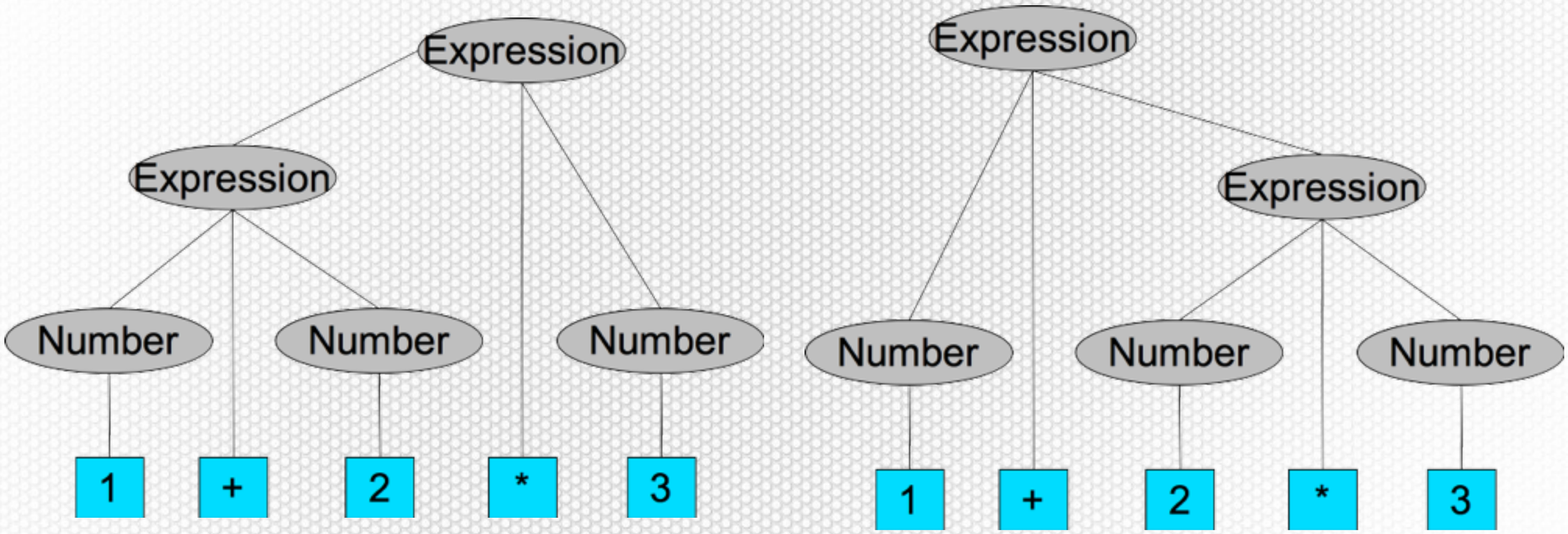
solved by ‘optimising’
the grammar

Left recursion removal

- Expr -> Expr + Term
- Expr -> Expr – Term
- Expr -> Term
- Term -> [0-9]

- Expr -> Term Rest
- Rest -> + Term Rest
  | - Term Rest
  | ε
- Term -> [0-9]

Ambiguity:
one sentence, several possible trees



sometimes possible to annotate the grammar with priorities

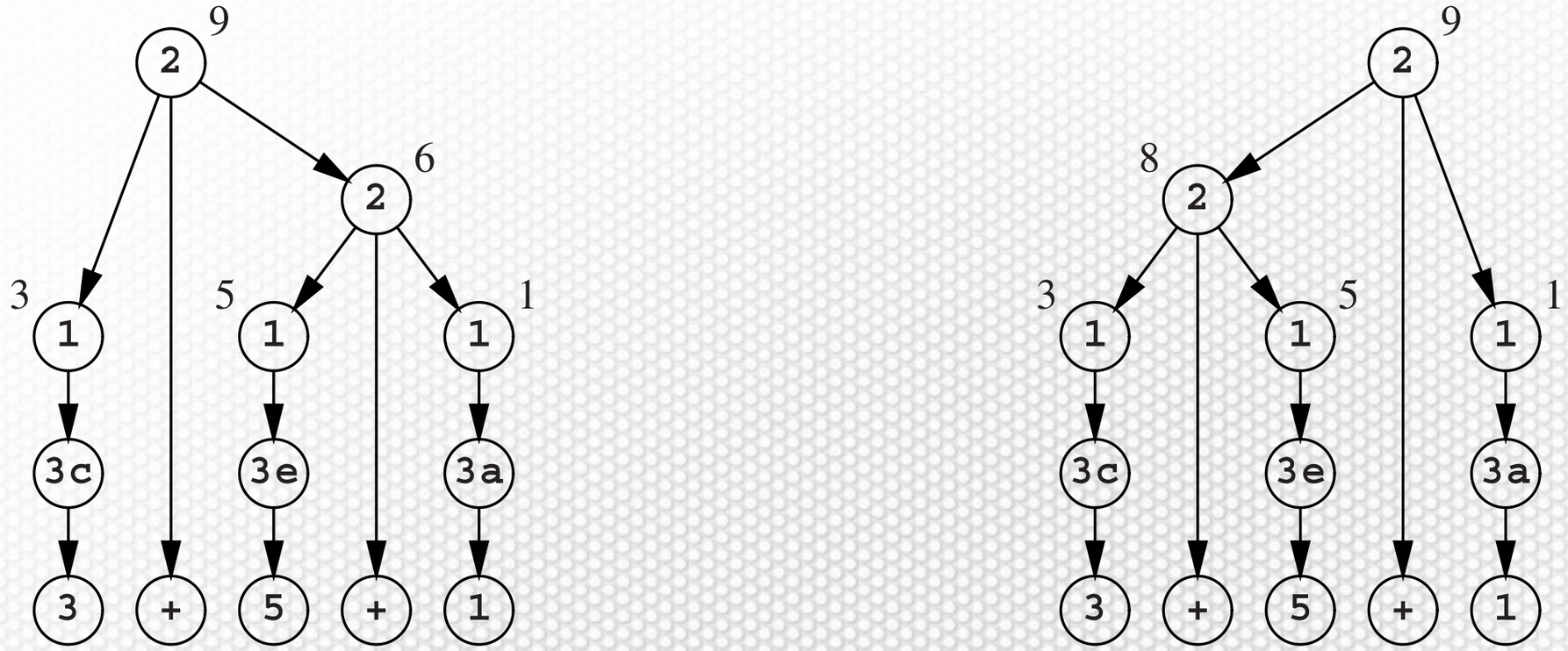


Fig. 3.2. Spurious ambiguity: no change in semantics

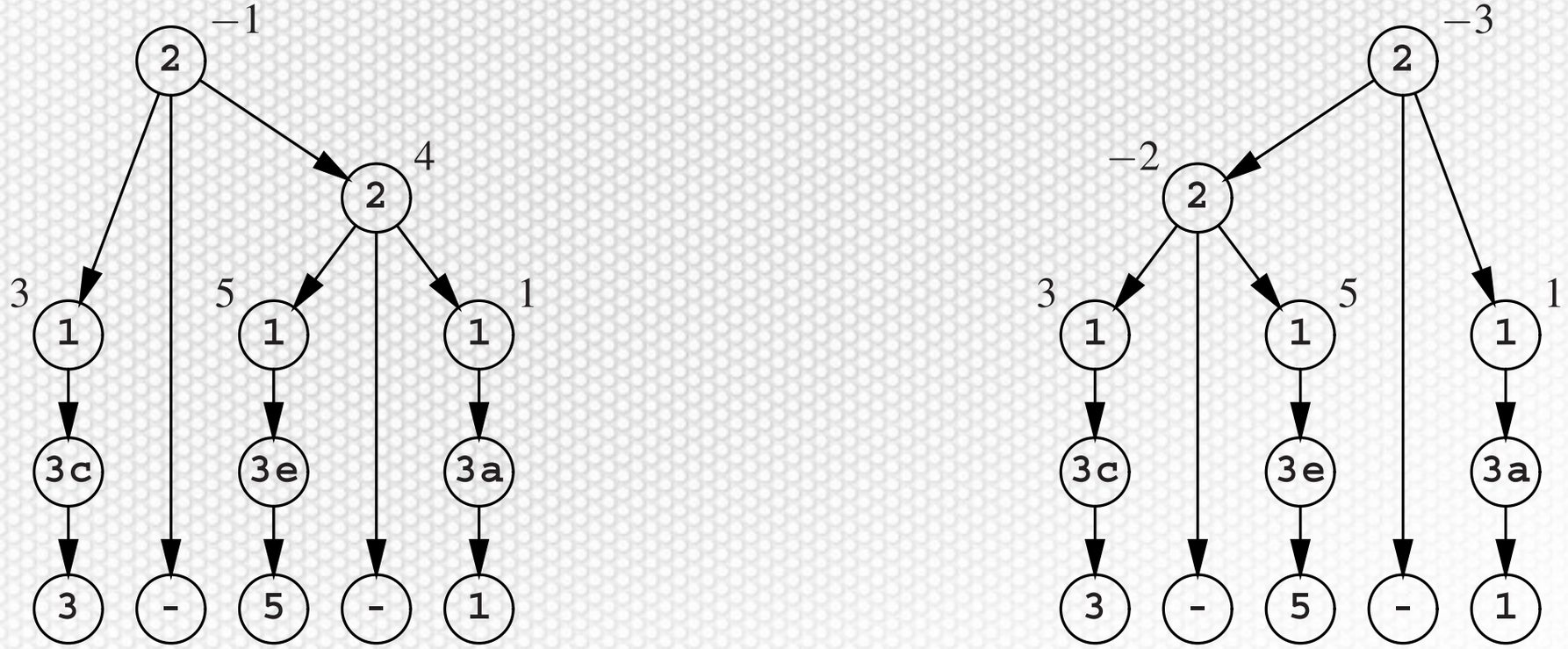


Fig. 3.3. Essential ambiguity: the semantics differ

p.64 of Grune/Jacobs' "Parsing Techniques", 2008

Disambiguation

- Expr -> Expr + Atom
- Expr -> Expr - Atom
- Expr -> Atom
- Atom -> Number
- Atom -> Variable

- Expr -> Expr + Term
- Expr -> Term
- Term -> Term * Primary
- Term -> Primary
- Primary -> Number
- Primary -> Variable

```
conditional-and-expression ::=
        inclusive-or-expression
        conditional-and-expression "&&" inclusive-or-expression
```

```
conditional-or-expression ::=
        conditional-and-expression
        conditional-or-expression "||" conditional-and-expression
```

```
null-coalescing-expression ::=
        conditional-or-expression
        conditional-or-expression "??" null-coalescing-expression
```

```
conditional-expression ::=
        null-coalescing-expression
        null-coalescing-expression "?" expression ":" expression
```

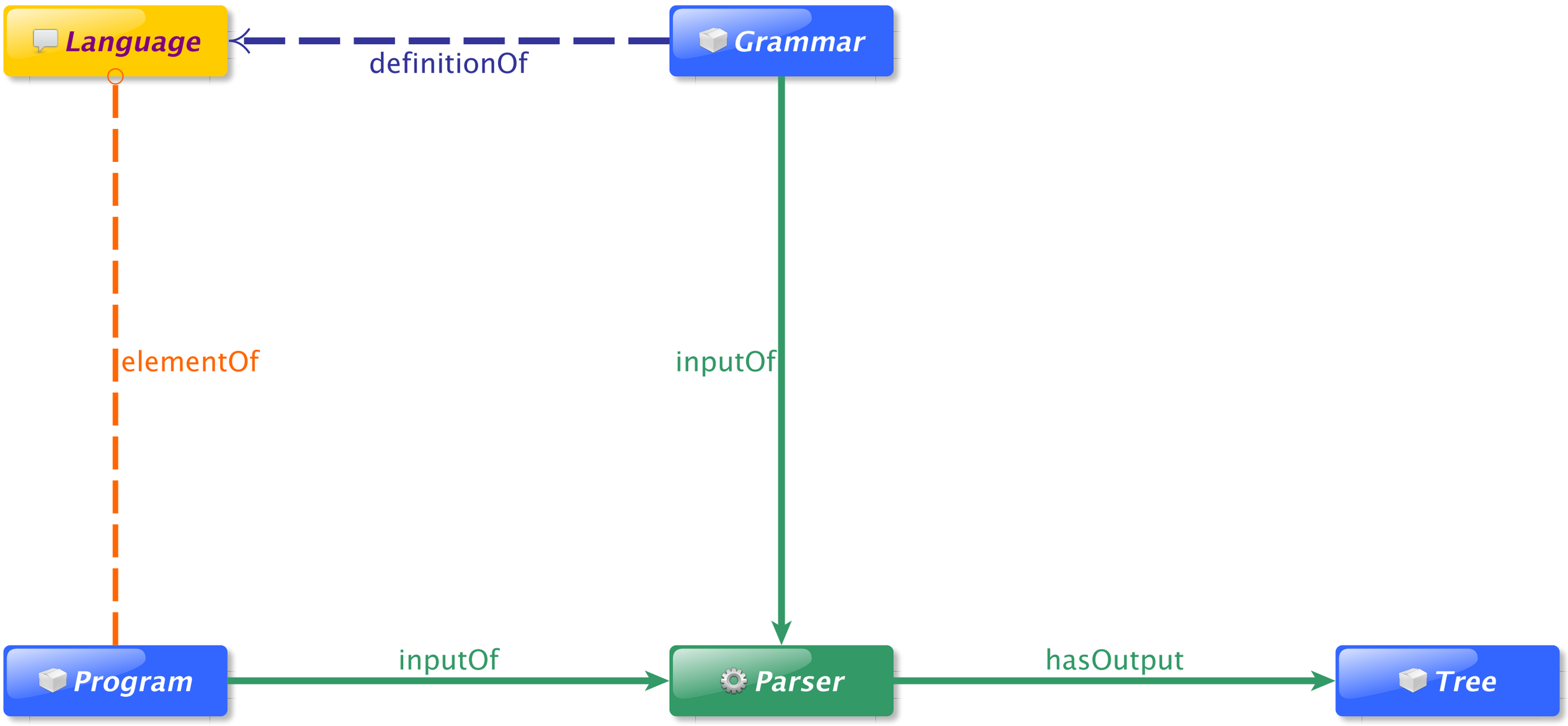
```
lambda-expression ::=
        anonymous-function-signature "=>" anonymous-function-body
```

```
expression ::=
        non-assignment-expression
        assignment
```

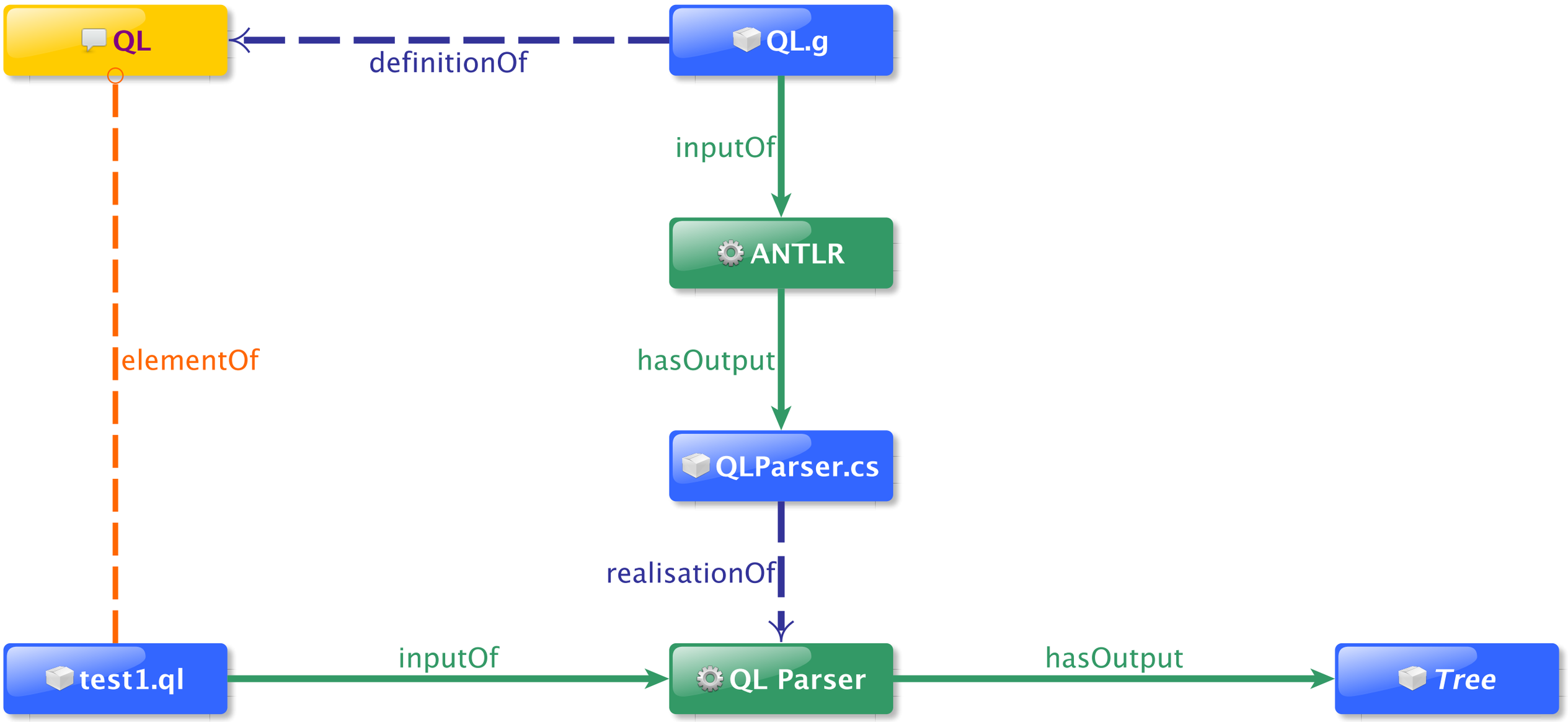
```
non-assignment-expression ::=
        conditional-expression
        lambda-expression
        query-expression
```

ALL "optimisations" are parsing tech-specific! (and make grammars uglier)

Recall: parser



Parser generator



Parser generators: ↑

- LALR(1)
  - Beaver
  - YACC, byacc, bison, etc
  - ELi
  - Irony
  - SableCC
  - yecc
- GLR
  - bison
  - DMS
  - GDK
  - Tom
- SGLR
  - ASF+SDF MetaEnv
  - Spoofax, Stratego/XT

Parser generators: ↓

- LL(k)
  - JavaCC
- LL(*)
  - ANTLR
- Earley
  - Marpa
  - ModelCC
- GLL
  - Rascal – SGTDBF
  - gll-combinators in Scala
- Packrat
  - Rats!
  - OMeta
  - PetitParser
- Others
  - TXL

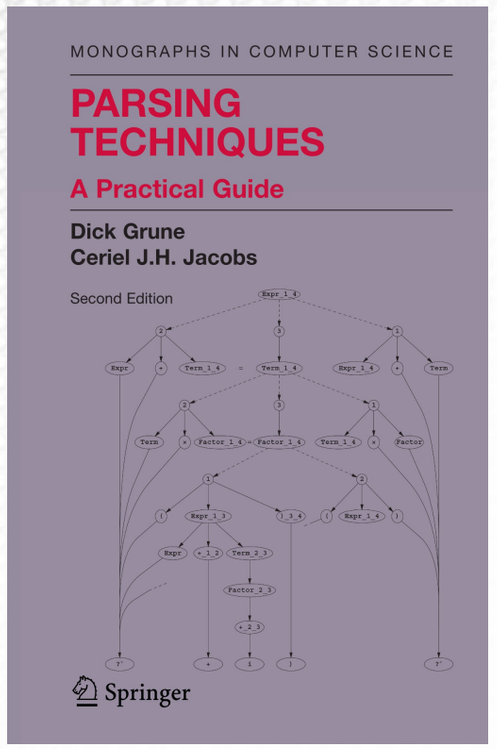
Summary of parsing techniques

- Top-down
  - predict-match and variants/improvements
  - backtracking is good; memoisation is good
  - left recursion is generally problematic
- Bottom-up
  - shift-reduce and variants
  - deterministic CFLs in linear time

Conclusion

- "Making a linear-time parser for an arbitrary given grammar is 10% hard work; the other 90% can be done by computer".
  [Grune/Jacobs, Parsing Techniques, 2008, p.81]
- Every notation/metatool defines a class of Gs/Ls
- Parsing should never be more complex than $O(n^3)$
- Many books/papers exist; beware of bullshit!

follow @grammarware!



Credits

- Given on the bottom of each slide
  - unless self-made or public domain
- Font
  - Intuitive by Bruno de Souza Leão, OFL
    http://openfontlibrary.org/en/font/intuitive
- Feedback
  - http://grammarware.net, http://grammarware.github.io, …