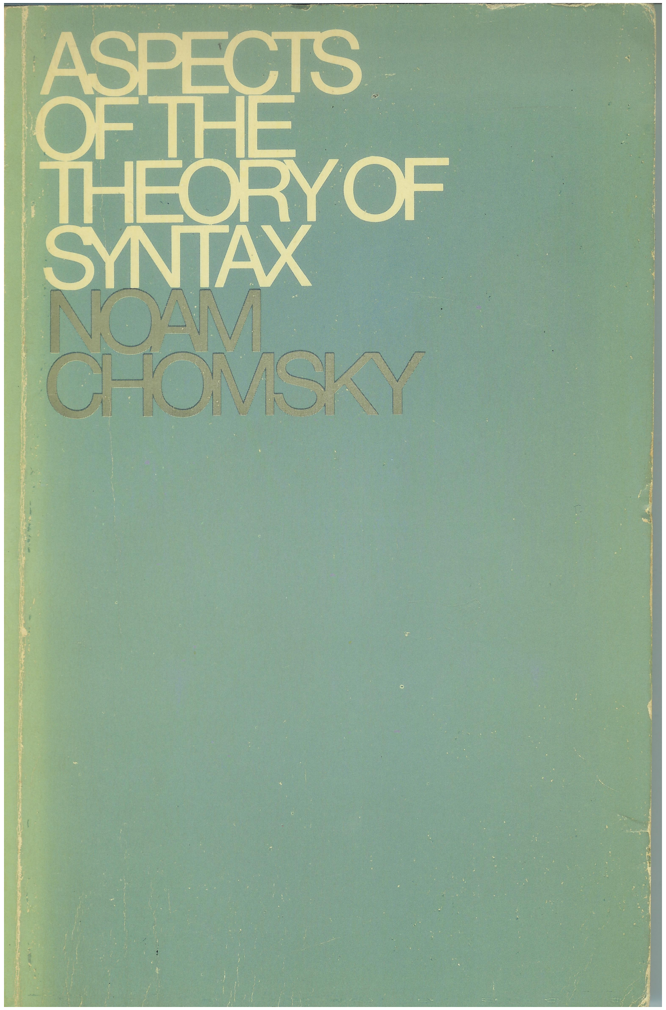
Grammars and Parsing

Paul Klint

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



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

A Language ...

- ... is a (possibly infinite) set of sentences
- Exercise:
  - Give examples of finite languages
  - Give examples of infinite languages

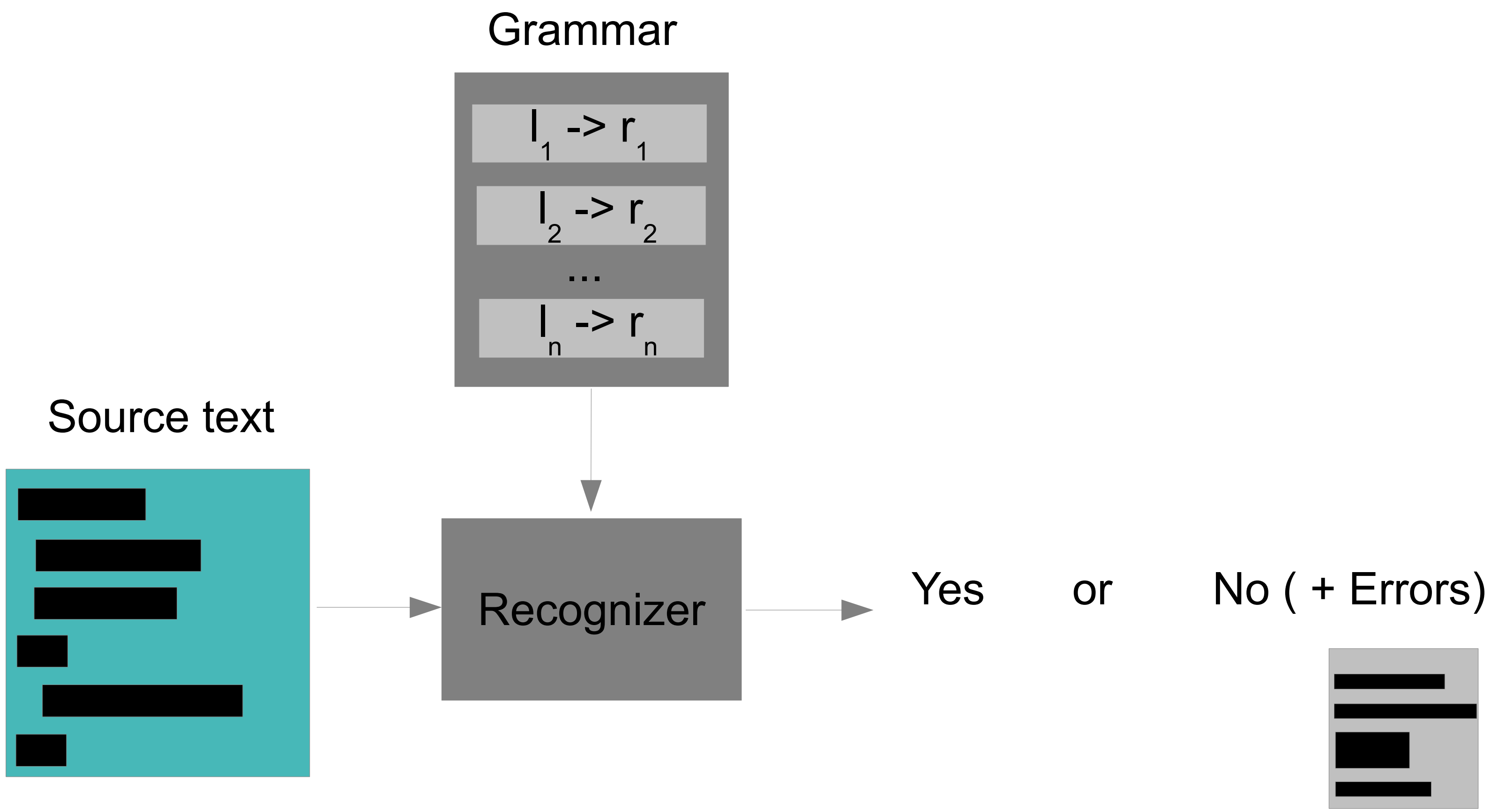
A Grammar ...

- ... is a set of formation rules to describe the sentences in a language
- The Chomsky hierarchy:
  - Context-sensitive languages
    - Natural language processing
  - Context-free languages
    - Syntax of programming languages
  - Regular languages
    - Regular expressions, grep, lexical syntax

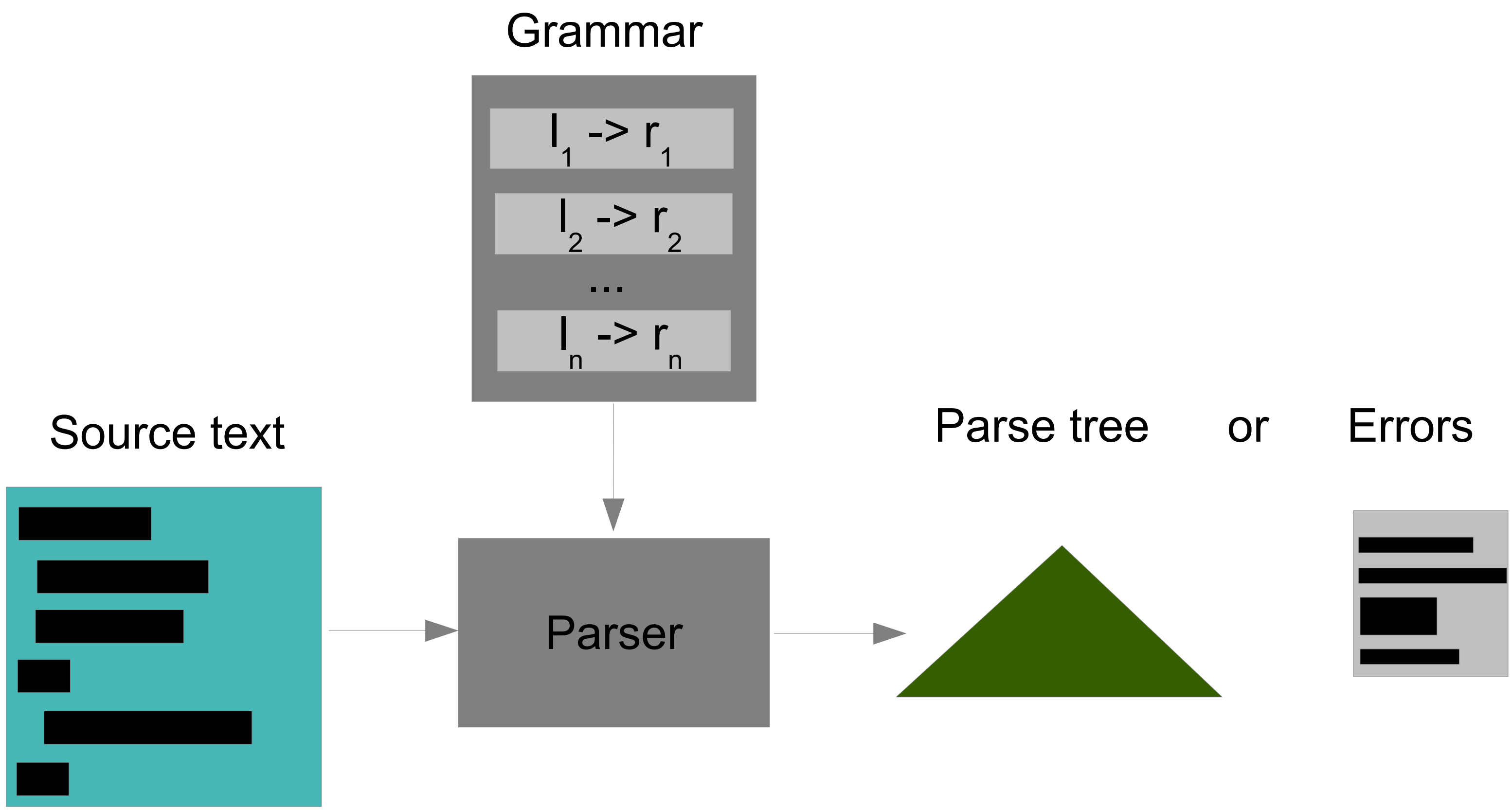
Syntax Analysis (aka Parsing) ...

- ... is the process of analyzing the syntactic structure of a sentence.
- A recognizer only says Yes or No (+ messages) to the question:
  - Does this sentence belong to language L?
- A parser also builds a structured representation when the text is syntactically correct.
  - Such a “syntax tree” or “parse trees” is a proof how the grammar rules can be sued to derive the sentence.

A Recognizer

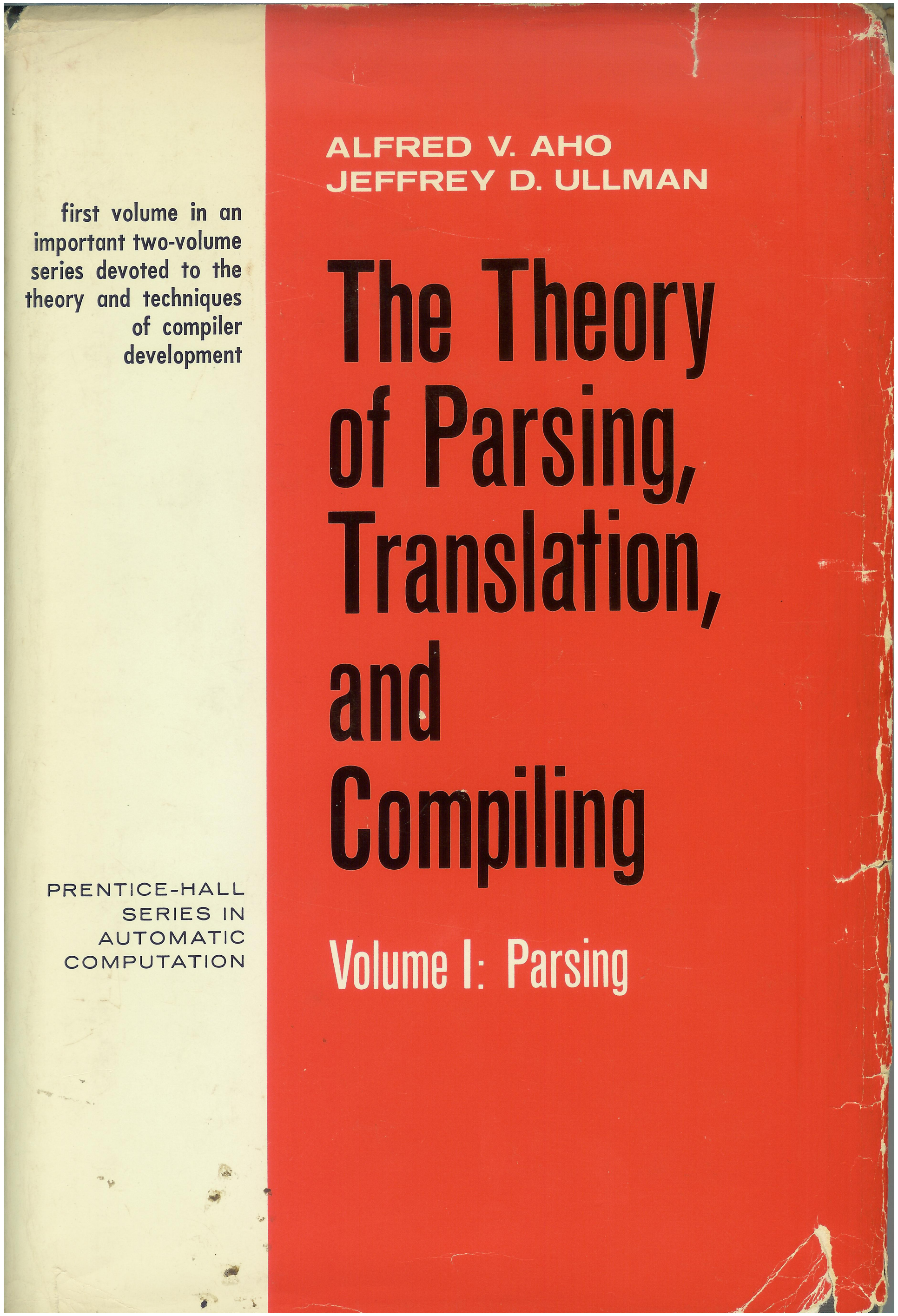


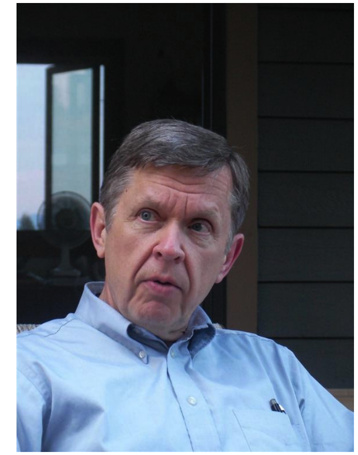
A Parser

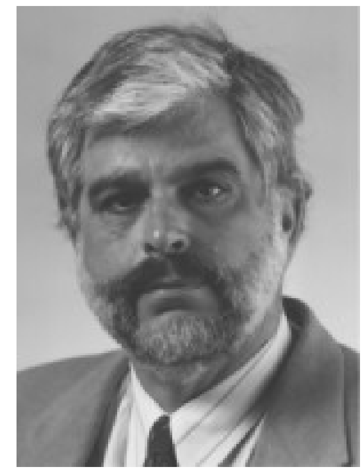

Grammar
l_1 -> r_1
l_2 -> r_2
...
l_n -> r_n
Source text
Parser
Parse tree
or
Errors

Why are Grammars and Parsing Techniques relevant?

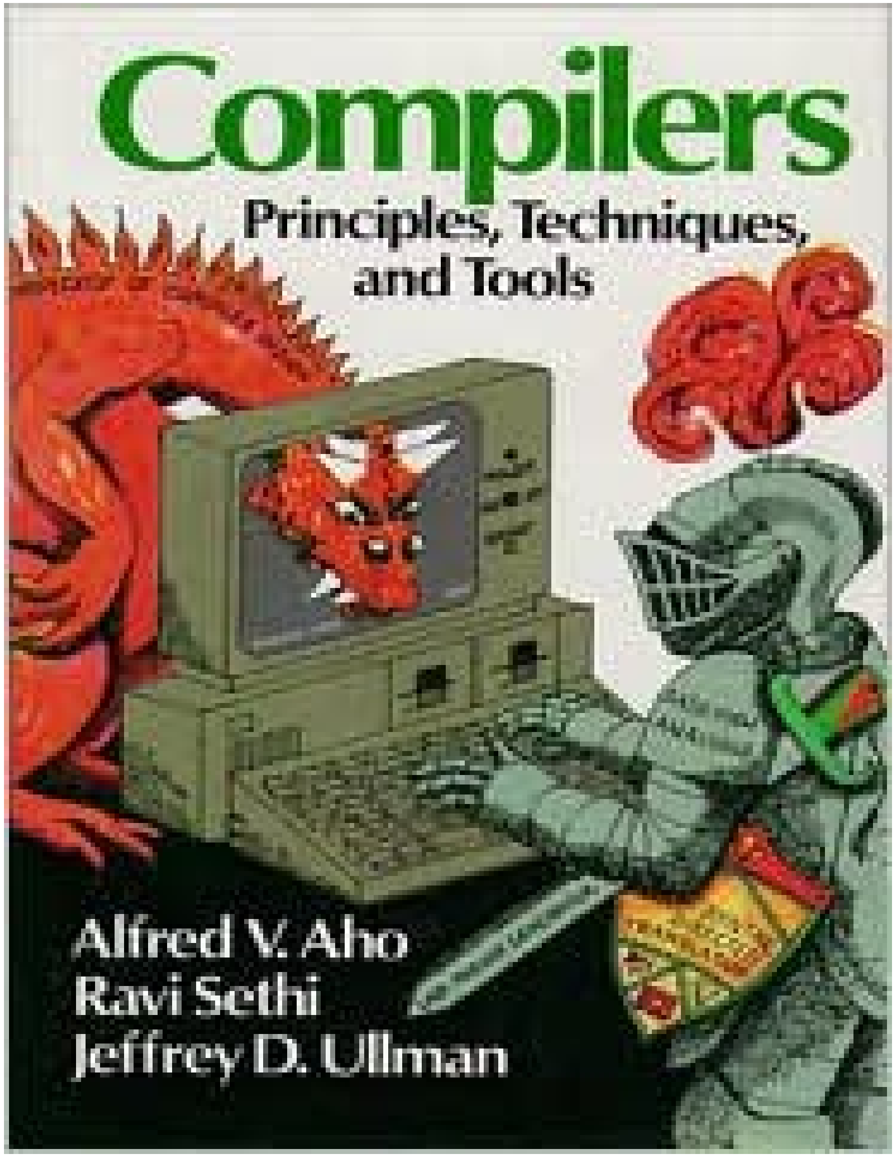
- A *grammar* is a formal method to describe a (textual) *language*
  - Programming languages: C, Java, C#, JavaScript
  - Domain-specific languages: BibTex, Mathematica
  - Data formats: log files, protocol data
- *Parsing*:
  - Tests whether a text conforms to a grammar
  - Turns a correct text into a parse tree



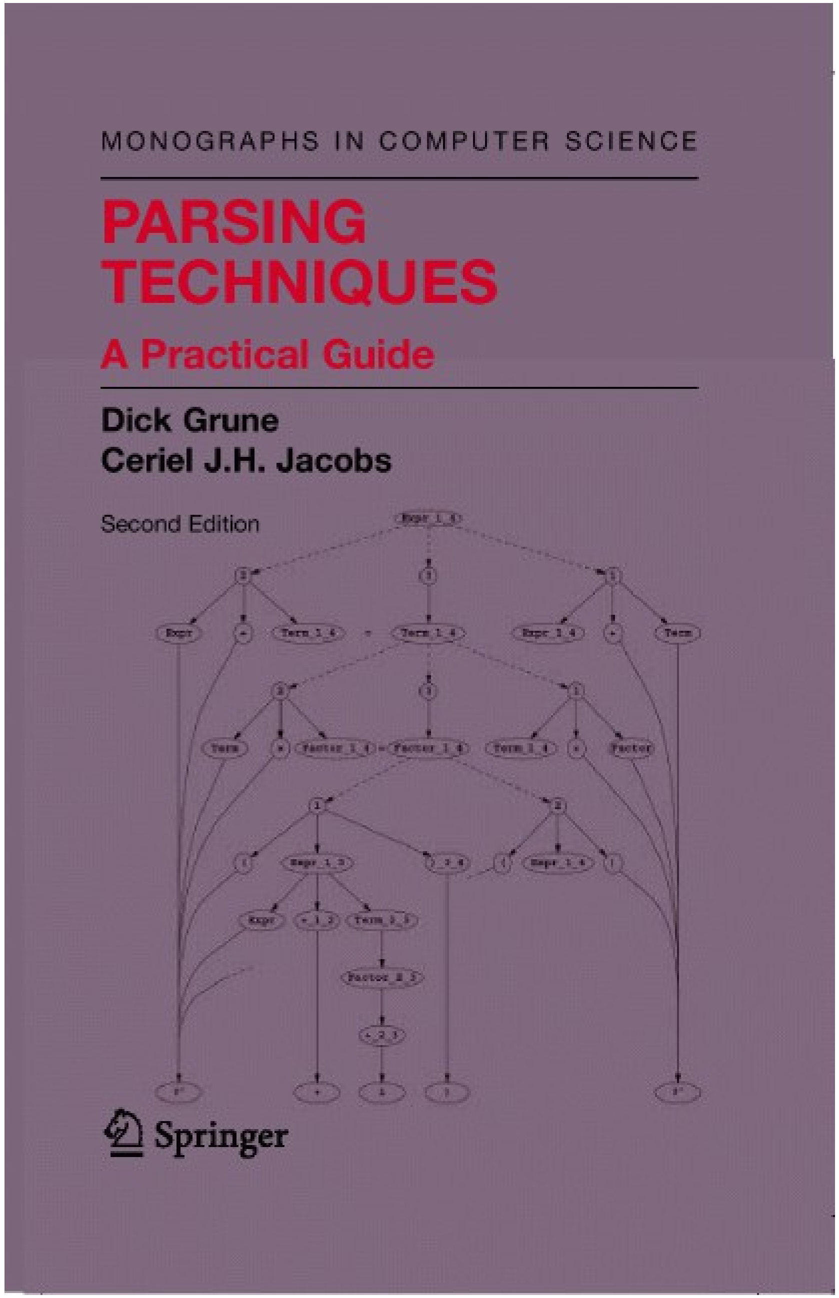




A.V. Aho & J.D. Ullman,
The Theory of Parsing,
Translation and
Compiling,
Parts I + II, 1972



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



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

What is Syntax Analysis about?

- Syntax analysis (or parsing) is about recognizing structure in text (or the lack thereof)
- The question “Is this a textually correct Java program?” can be answered by syntax analysis.
- Note: other correctness aspects are outside the scope of parsing:
  - Has this variable been declared?
  - Is this expression type correct?
  - Is this method called with the right parameters?

When is Syntax Analysis Used?

- Compilers
- IDEs
- Software analysis
- Software transformation
- DSL implementations
- Natural Language processing
- Genomics (parsing of DNA fragments)

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

Example

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

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

Variations in Notation

- Name -> **tom | dick | harry**
- <Name> ::= "tom" | "dick" | "harry"
- "tom" | "dick" | "harry" -> Name
- In Rascal:
  - syntax Name = "tom" | "dick" | "harry";

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

Exercise: What changed and Why?

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

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

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

We start with text

Consider the assignment statement:

Position := initial + rate * 60

First approximation, this is a string of characters:

p	o	s	i	t	i	o	n		:	=		i	n	i	t	i	a	l		+		r	a	t	e		*		6	0

From text to tokens

p	o	s	i	t	i	o	n		:	=		i	n	i	t	i	a	l		+		r	a	t	e		*		6	0

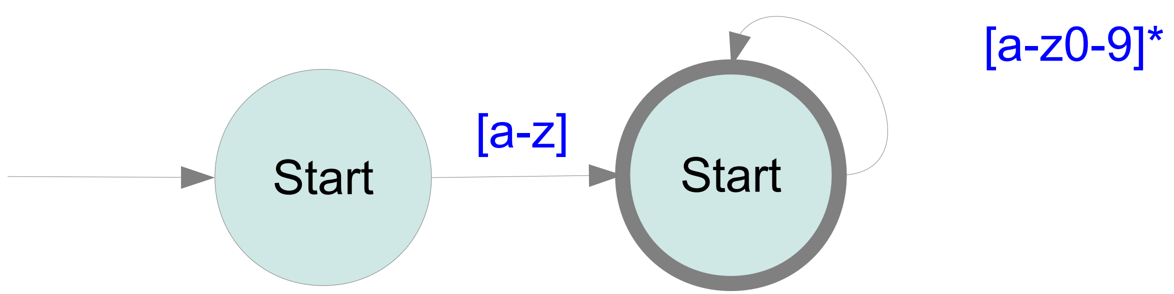
- The identifier position
- The assignment symbol :=
- The identifier initial
- The addition operator +
- The identifier rate
- The multiplication operator *
- The number 60

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

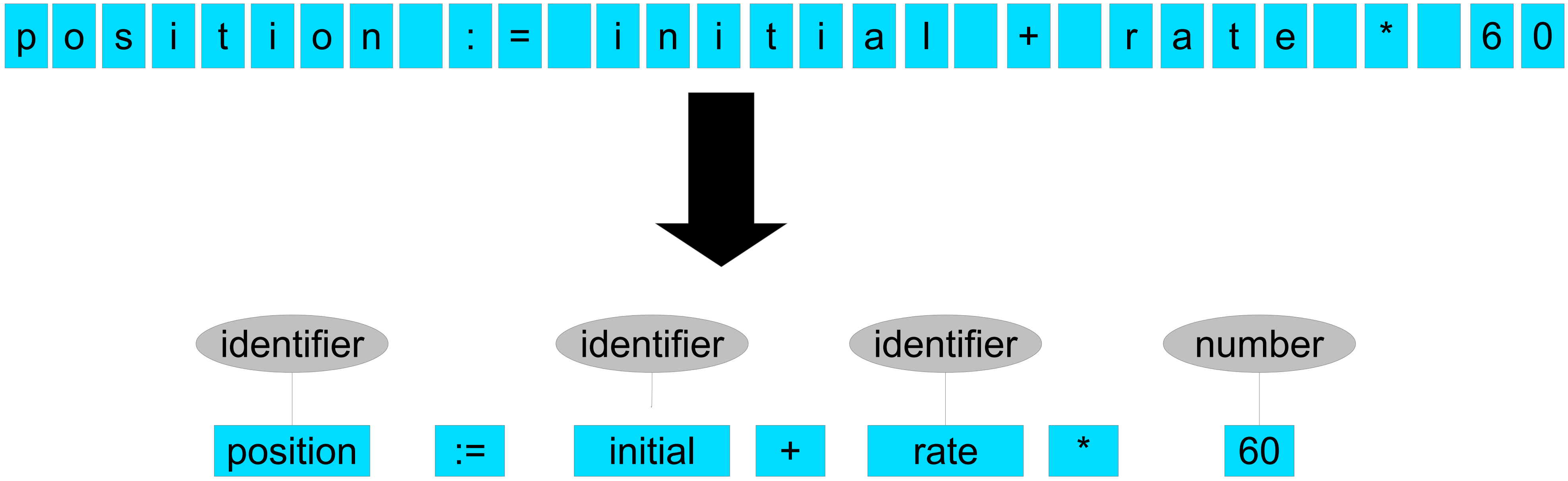
- Regular expressions can be implemented with a finite automaton
- Consider [a-z][a-z0-9]*



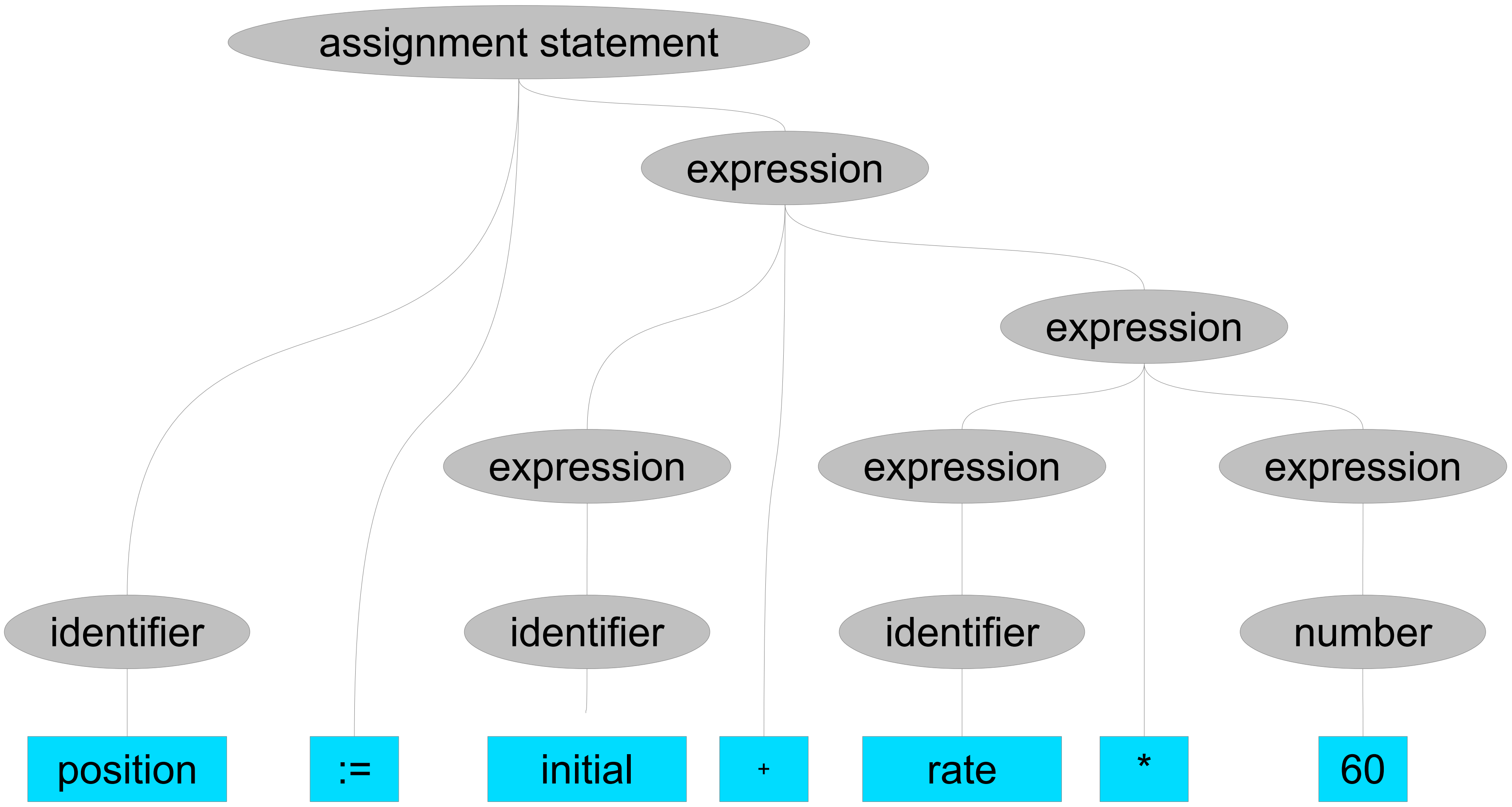
From text to tokens

Classify characters by lexical category:

- Tokenization
- Lexical analysis
- Lexical scanning



From Tokens to Parse Tree



Expression Grammar

The hierarchical structure of expressions can be described by recursive rules:

1. Any Identifier is an *expression*
2. Any Number is an *expression*
3. If Expression_1 and Expression_2 are *expressions* then so are:
   - Expression_1 + Expression_2
   - Expression_1 * Expression_2
   - (Expression_1)

Statement Grammar

1. If $Identifier_1$ is an identifier and $Expression_1$ is an expression then the following is a statement:
    - $Identifier_1$:= $Expression_1$
2. If $Expression_1$ is an expression and $Statement_1$ is an statement then the following are statements:
    - while ($Expression_1$) $Statement_1$
    - if ($Expression_1$) then $Statement_1$

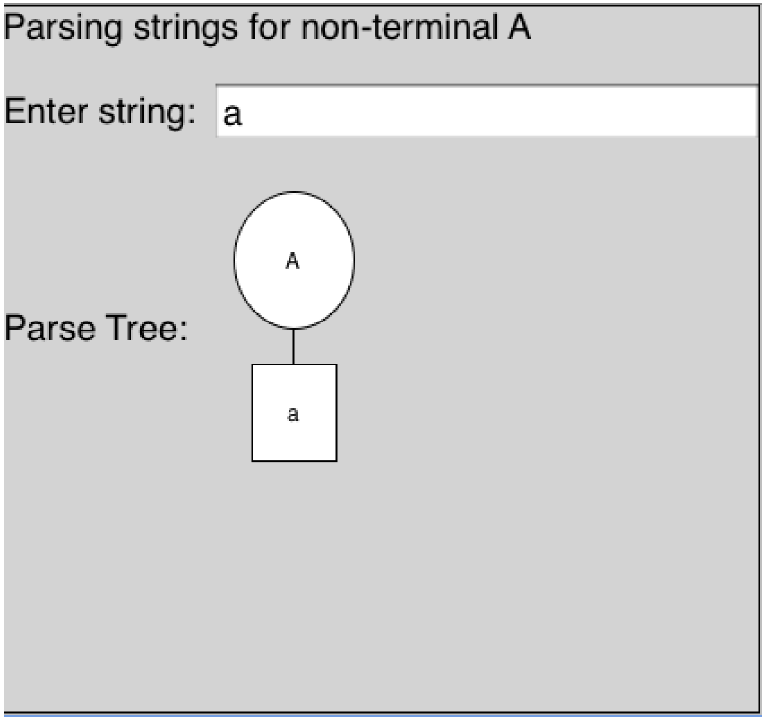
How do we get a parser?

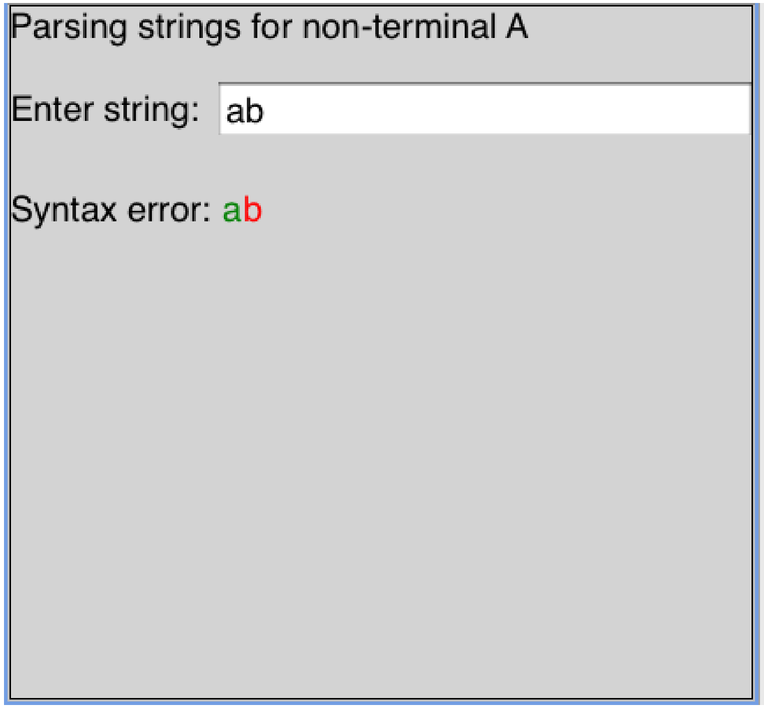
Use a parser generator

- Pro: regenerate when grammar changes
- Pro; recognized language is exactly known
- Pro: less effort
- Con: Grammar has to fit in the grammar class accepted by the parser generator (this may be very hard!)
- Con: mixing of parsing and other actions somewhat restricted
- Con: limited error recovery

Language A

```
start syntax A = "a";
```

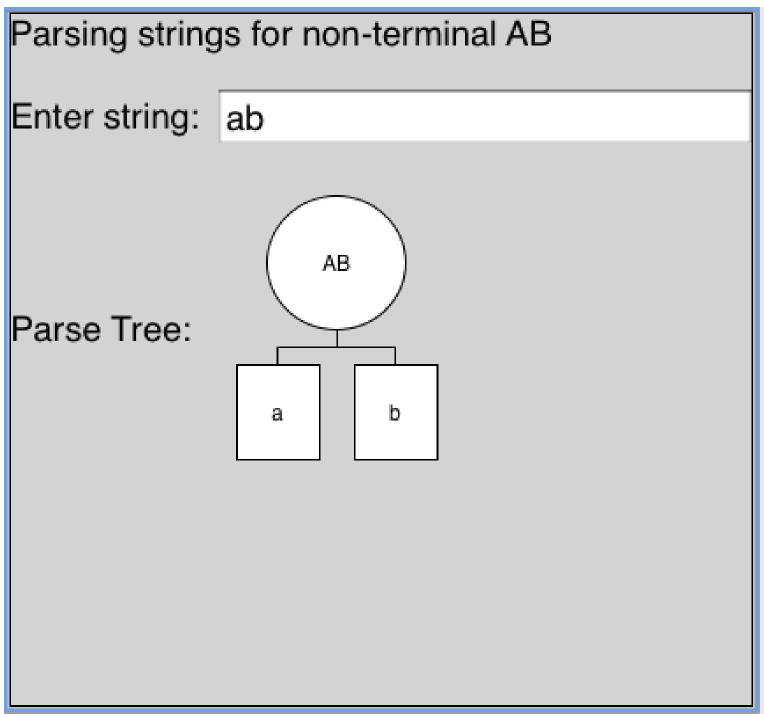




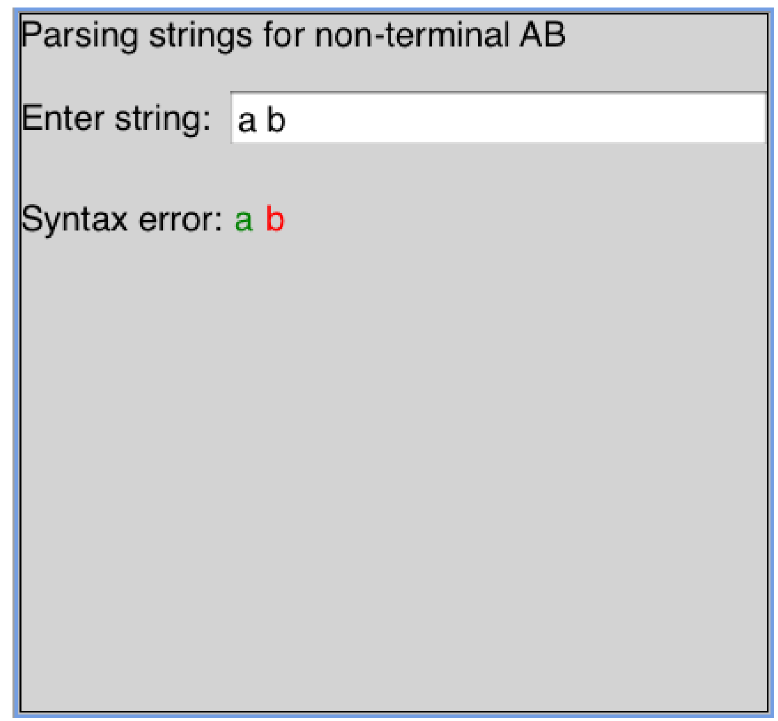

```
parseTreeViewer(#start[A])
```

Language AB

```
start syntax AB = "a" "b";
```

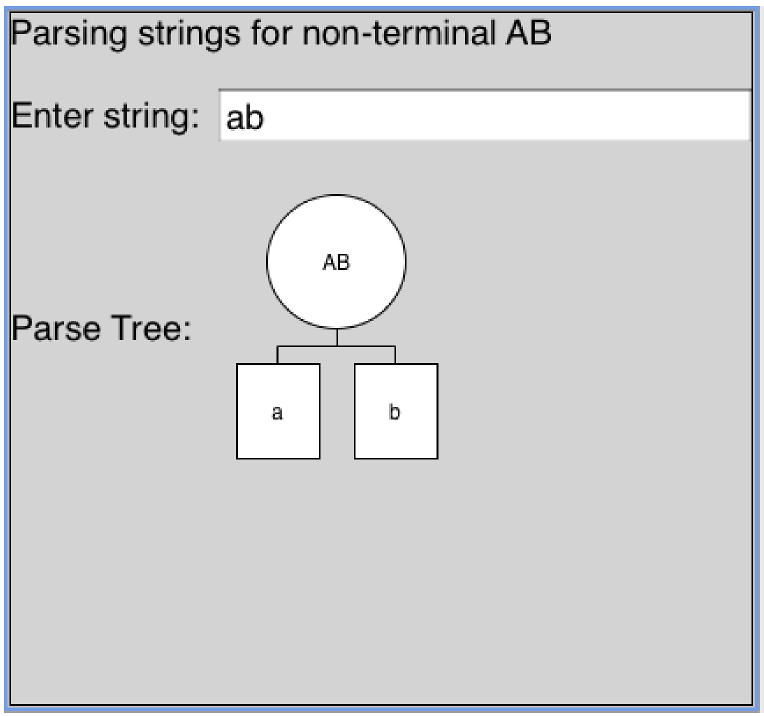


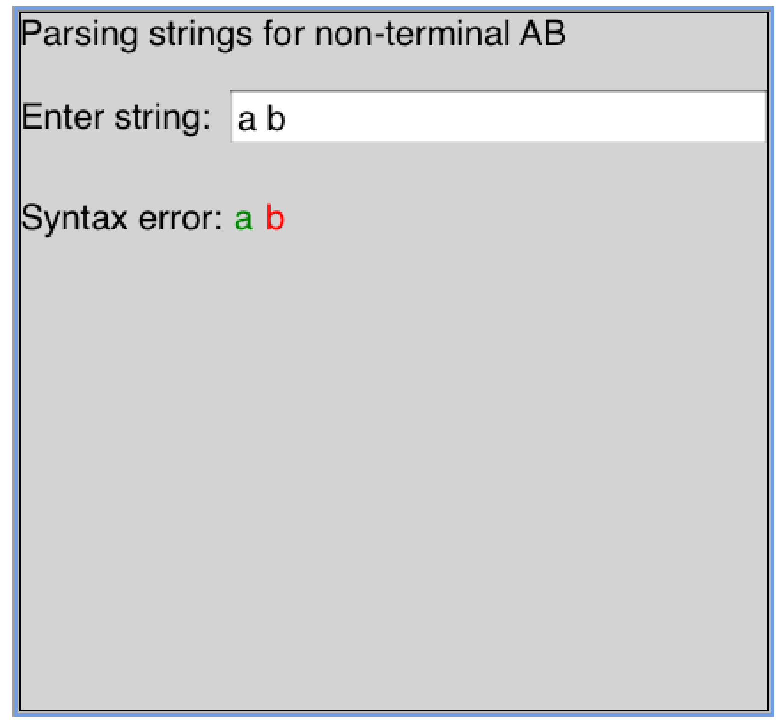

```
parseTreeViewer(#start[AB])
```



Language AB

```
start syntax AB = "a" "b";
```

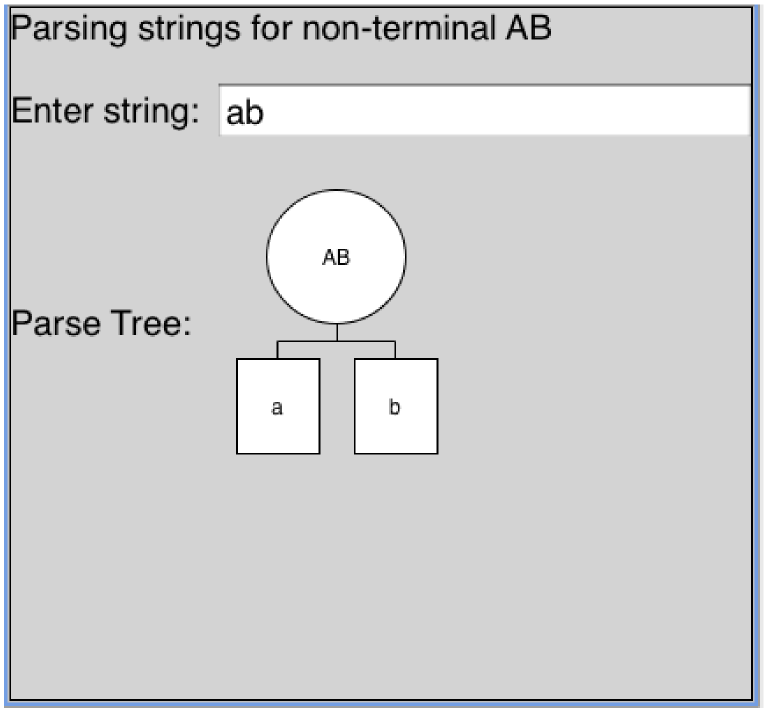


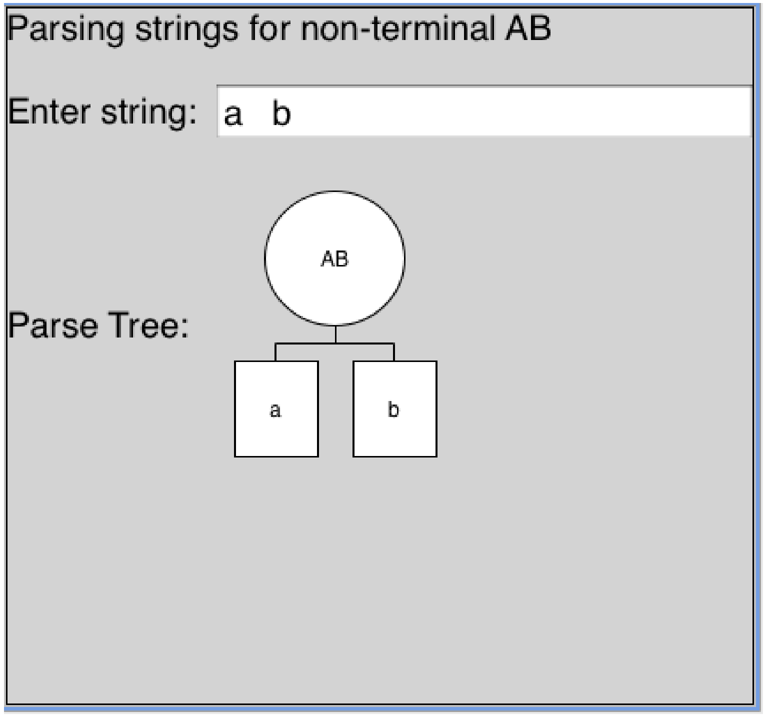



```
parseTreeViewer(#start[AB])
```

Language AB (with layout)

```
layout Whitespace = [\ \t\n]*;
start syntax AB = "a" "b";
```

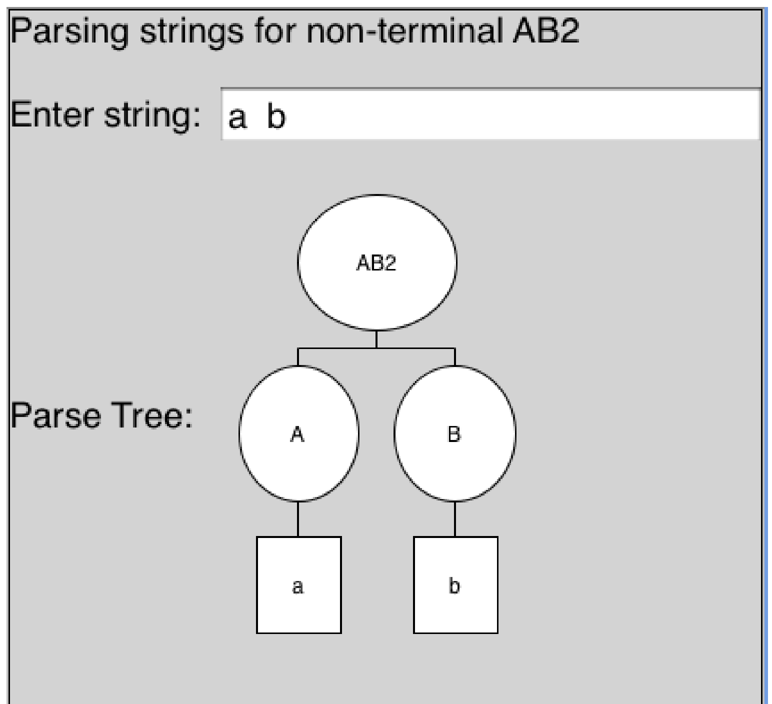





```
parseTreeViewer(#start[AB])
```

Language AB2

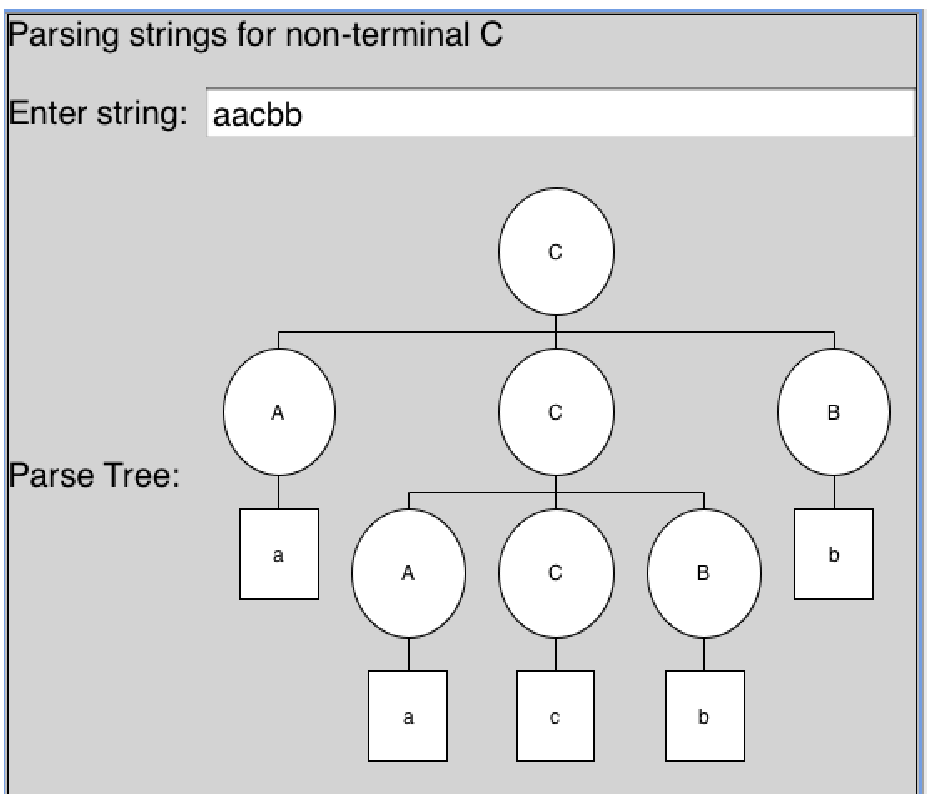
```
layout Whitespace = [\ \t\n]*;
syntax A = "a";
syntax B = "b";
start syntax AB2 = A B;
```

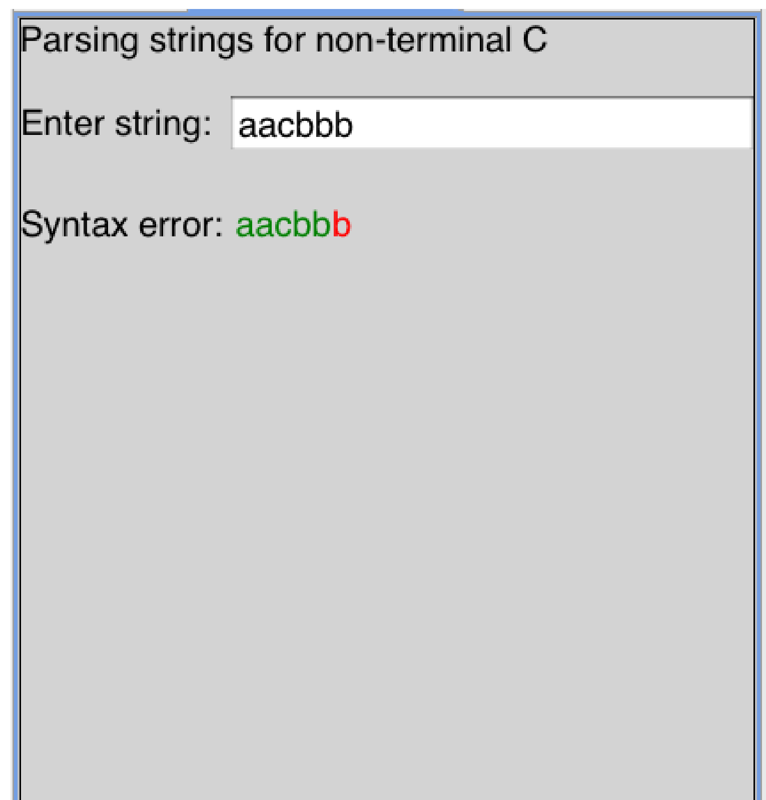



```
parseTreeViewer(#start[AB2])
```

Language C

```
layout Whitespace = [\ \t\n]*;
syntax A = "a";
syntax B = "b";
start syntax C = "c" | A C B;
```

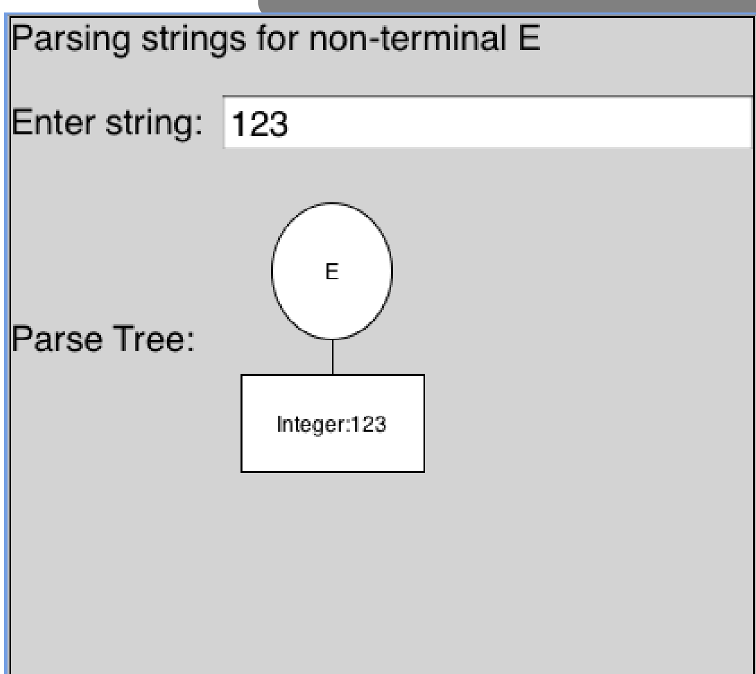


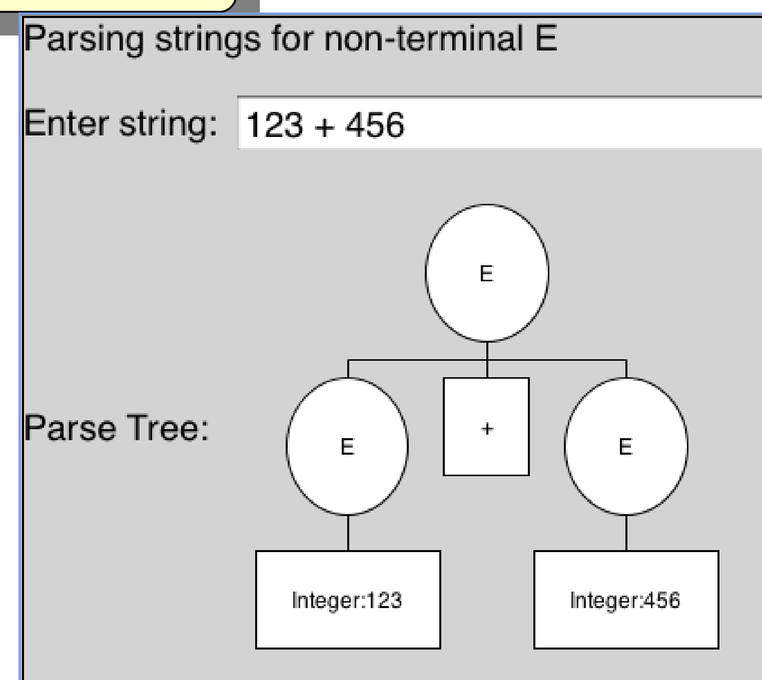



```
parseTreeViewer(#start[C])
```

Language E

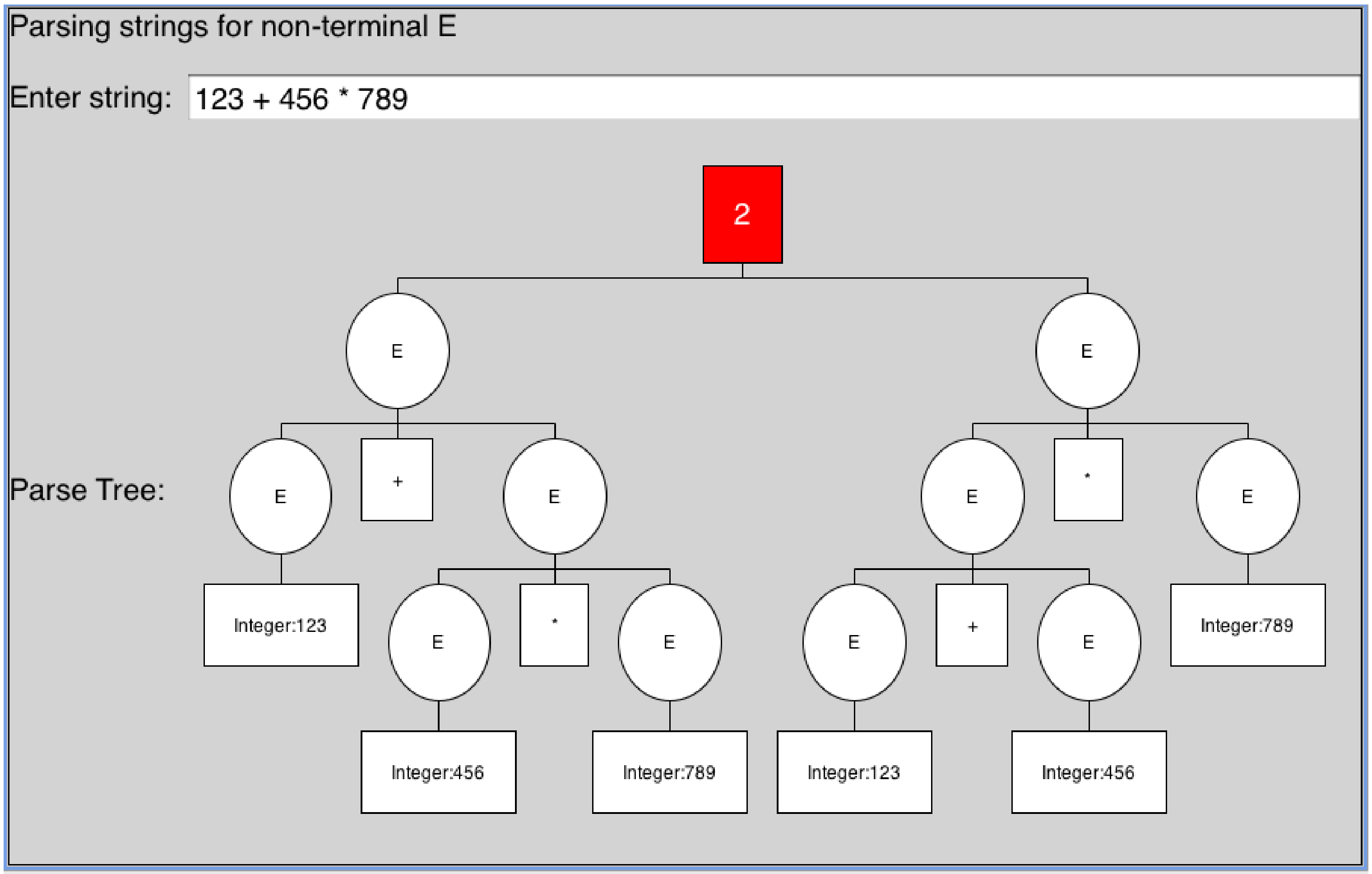
```
layout Whitespace = [\ \t\n]*;
lexical Integer = [0-9]+;
start syntax E = Integer
               | E "*" E
               | E "+" E
               | "(" E ")"
               ;
```





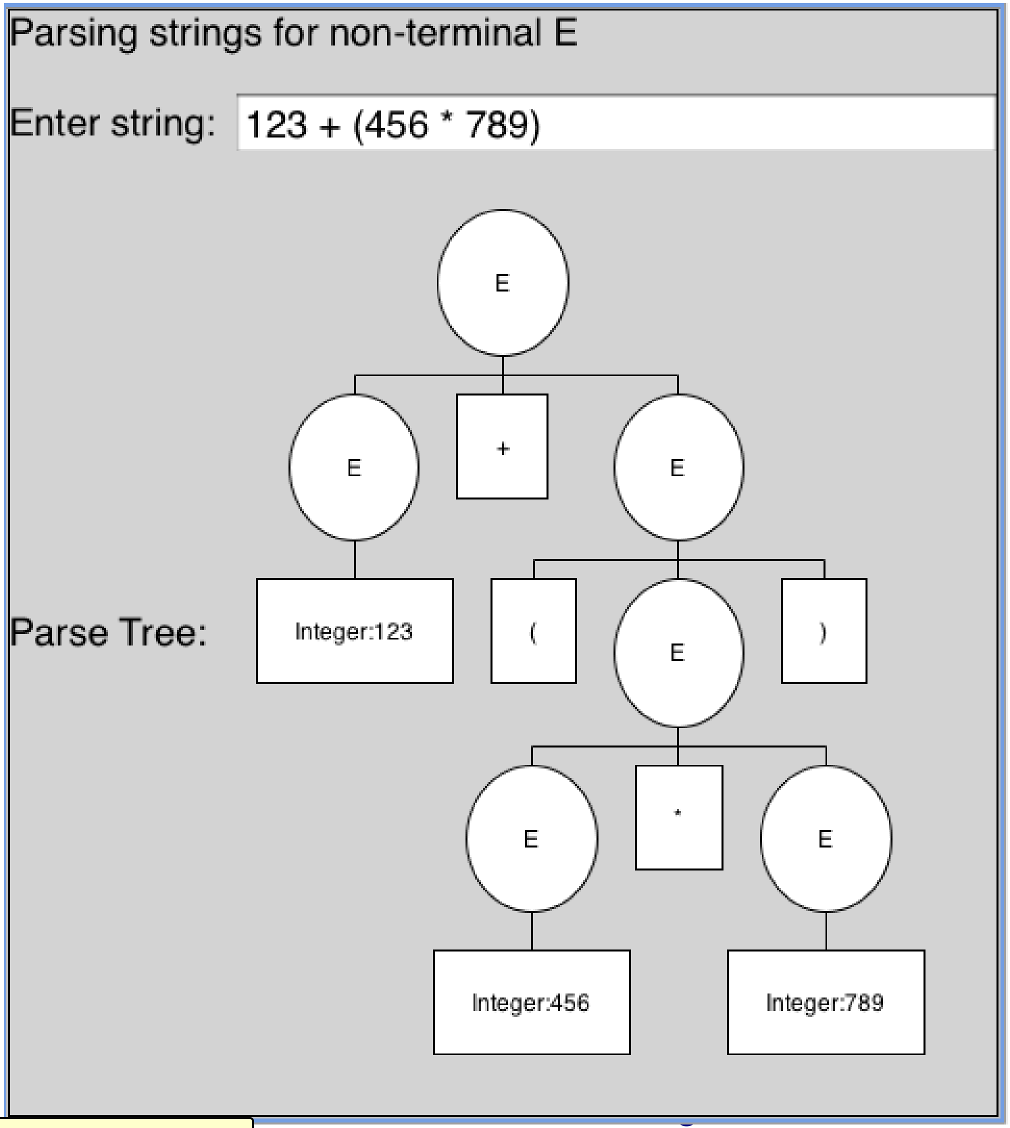

```
parseTreeViewer(#start[E])
```

Language E: ambiguity




```
parseTreeViewer(#start[E])
```

Language E: Using Parentheses

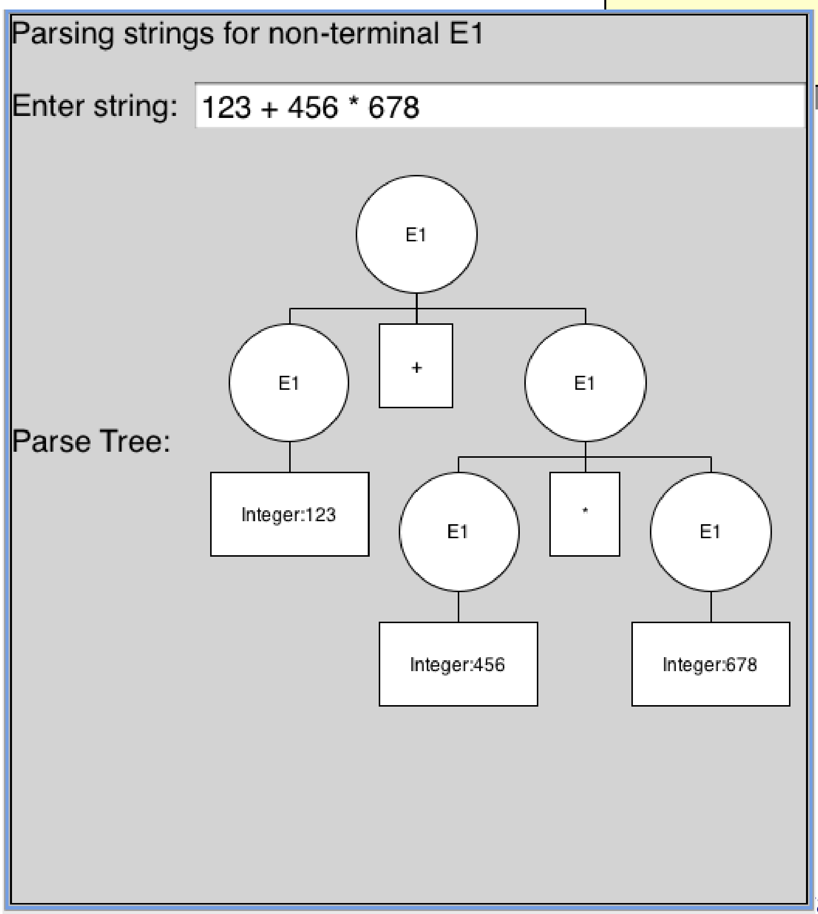



```
parseTreeViewer(#start[E])
```

Language E1: Define Priority

```
layout Whitespace = [\ \t\n]*;
lexical Integer = [0-9]+;
start syntax E1 = Integer
               | E1 "*" E1
               > E1 "+" E1
               | "(" E1 ")"
               ;
```

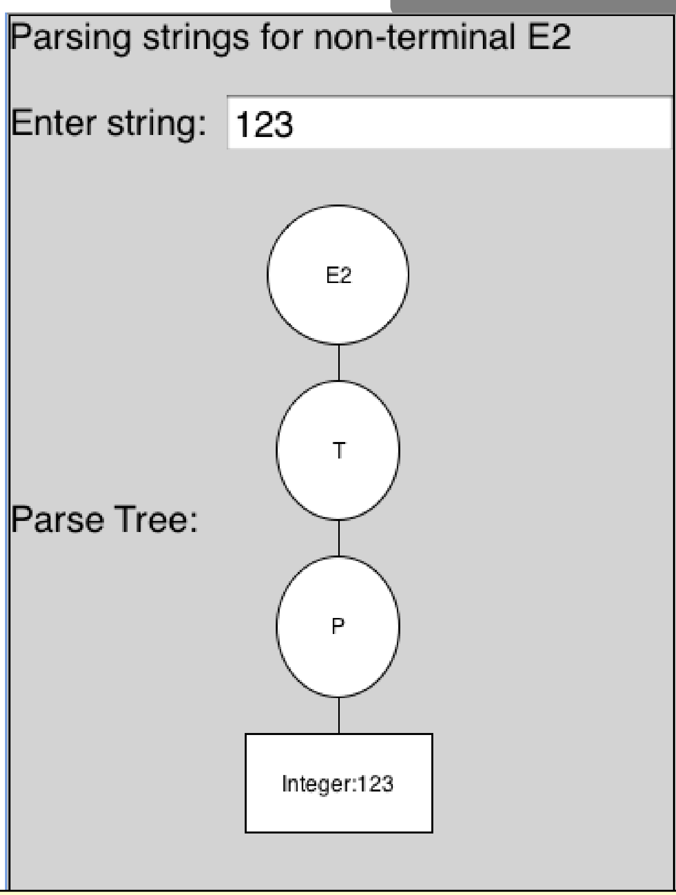
> defines that E1 “*” E1 has higher priority than E1 “+” E1



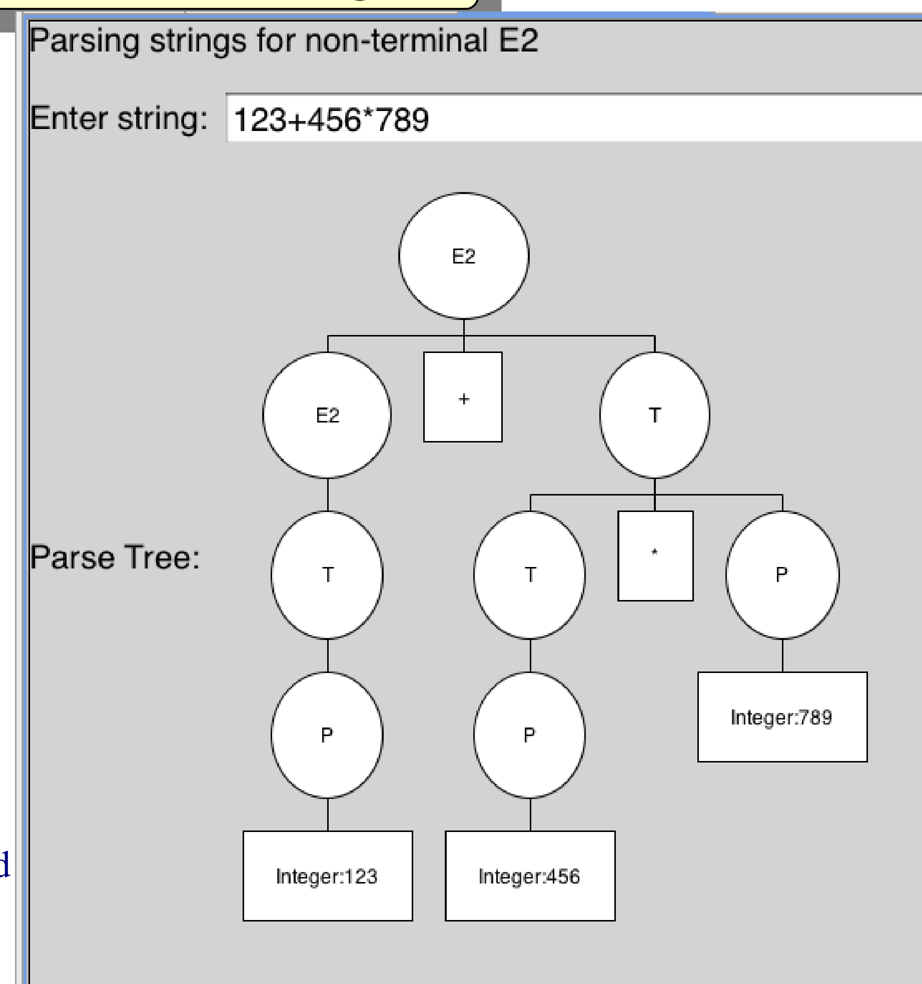

```
parseTreeViewer(#start[E1])
```

Language E2: Extra non-terminals

```
layout Whitespace = [\ \t\n]*;
lexical Integer = [0-9]+;
start syntax E2 = E2 "+" T | T;
syntax T = T "*" P | P;
syntax P = "(" E2 ")" | Integer;
```

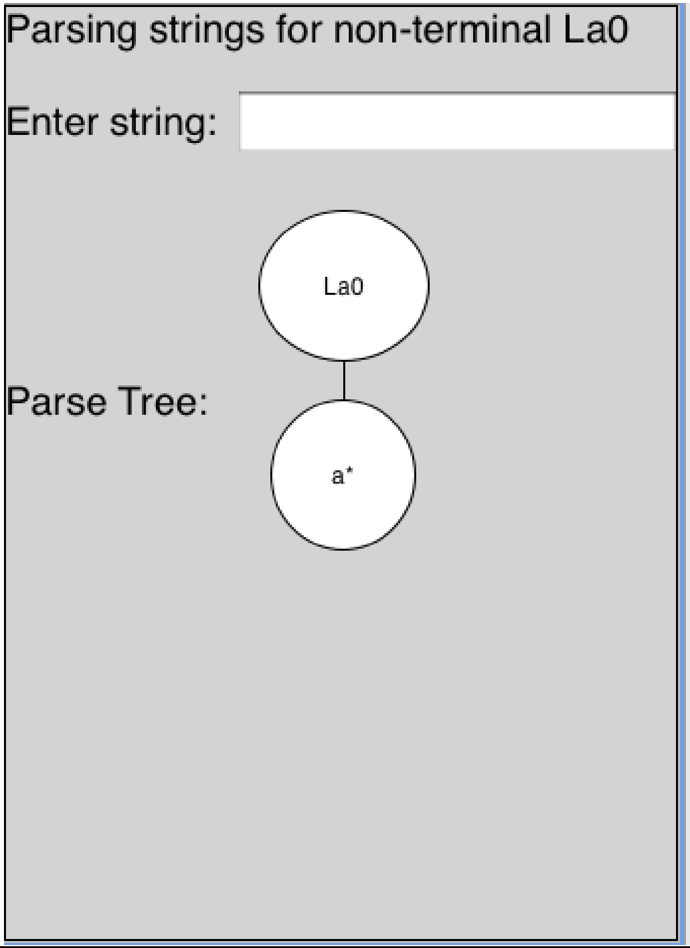


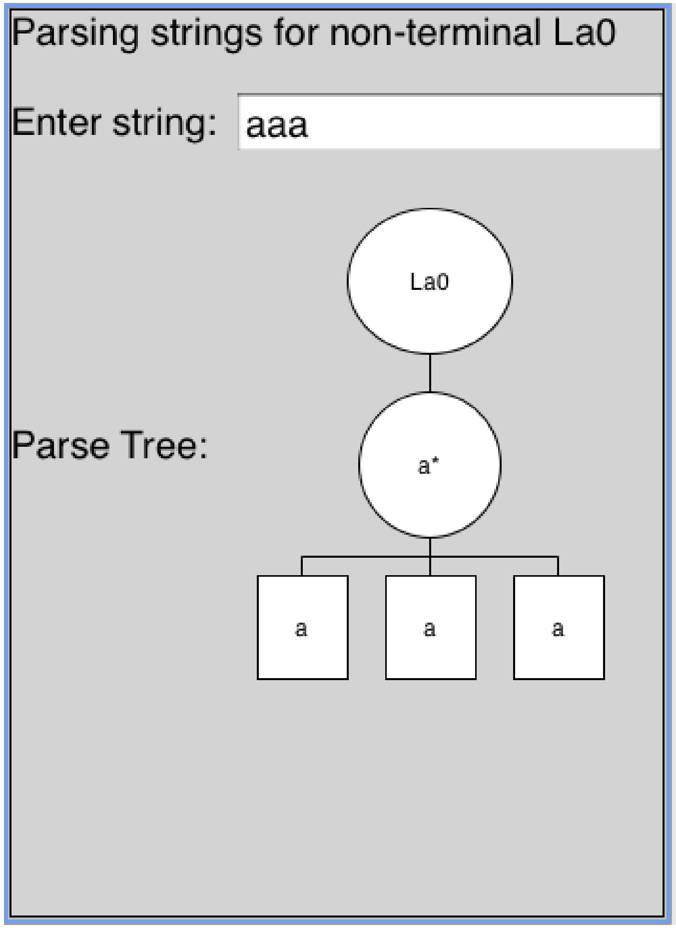

```
parseTreeViewer(#start[E2])
```



Language La0: List of zero or more a's

```
start syntax La0 = "a"*;
```

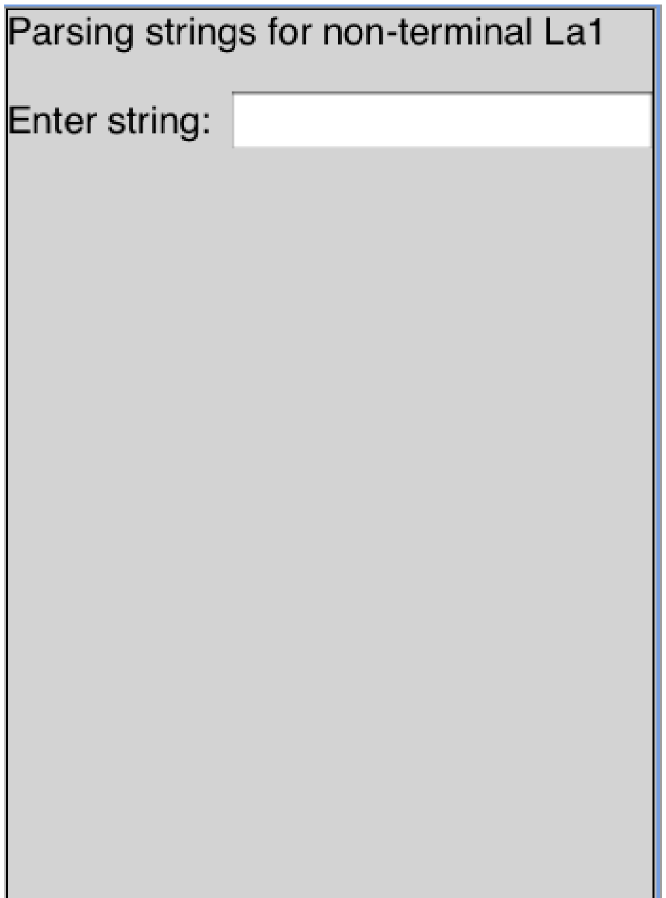


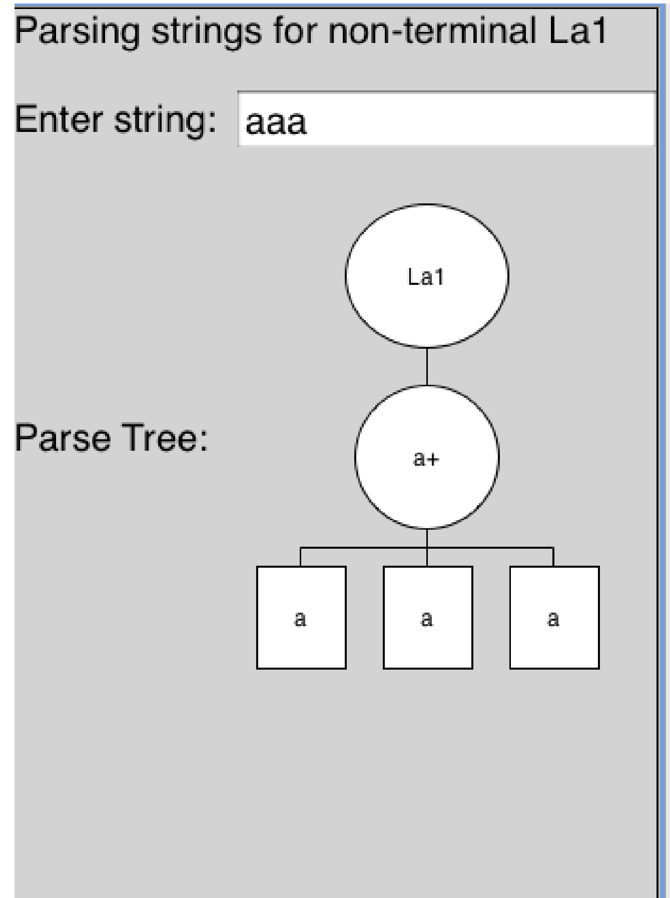



```
parseTreeViewer(#start[La0])
```

Language La0: List of one or more a's

```
start syntax La1 = "a"+;
```

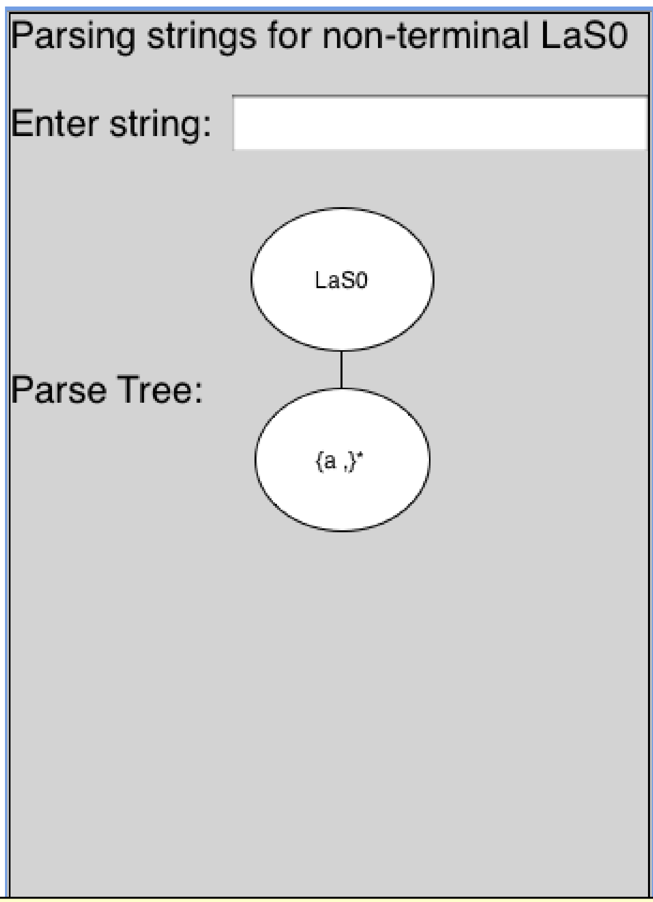


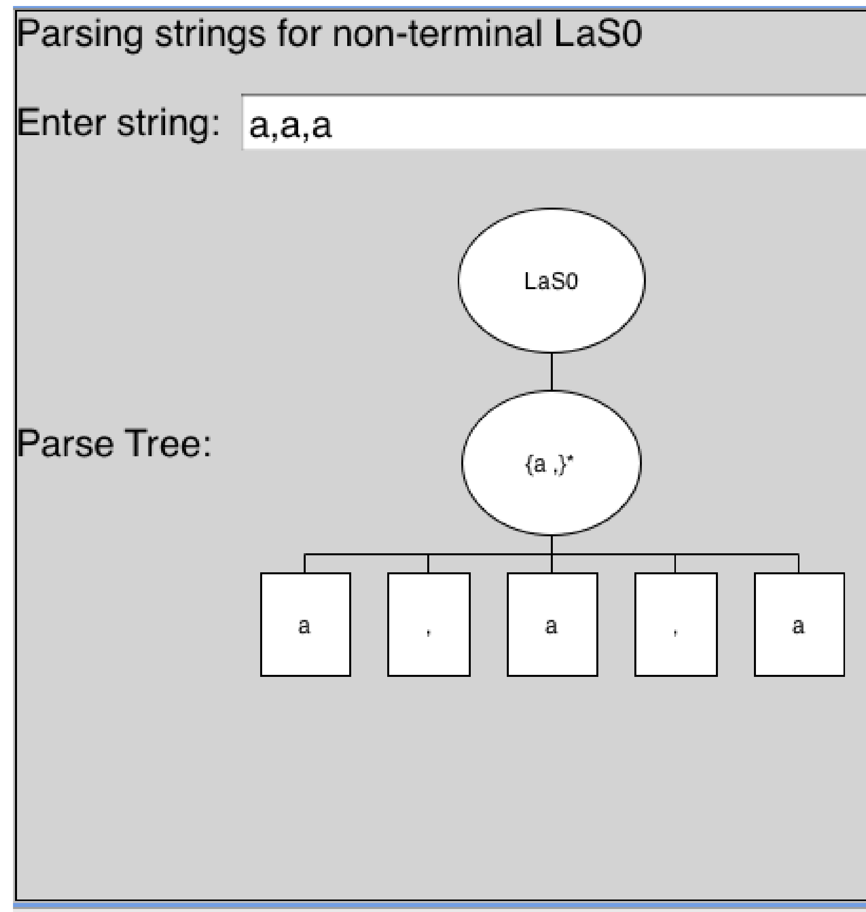



```
parseTreeViewer(#start[La1])
```

Language LaS0: List of zero or more a's separated by comma's

```
start syntax LaS0 = {"a" ","}*;
```

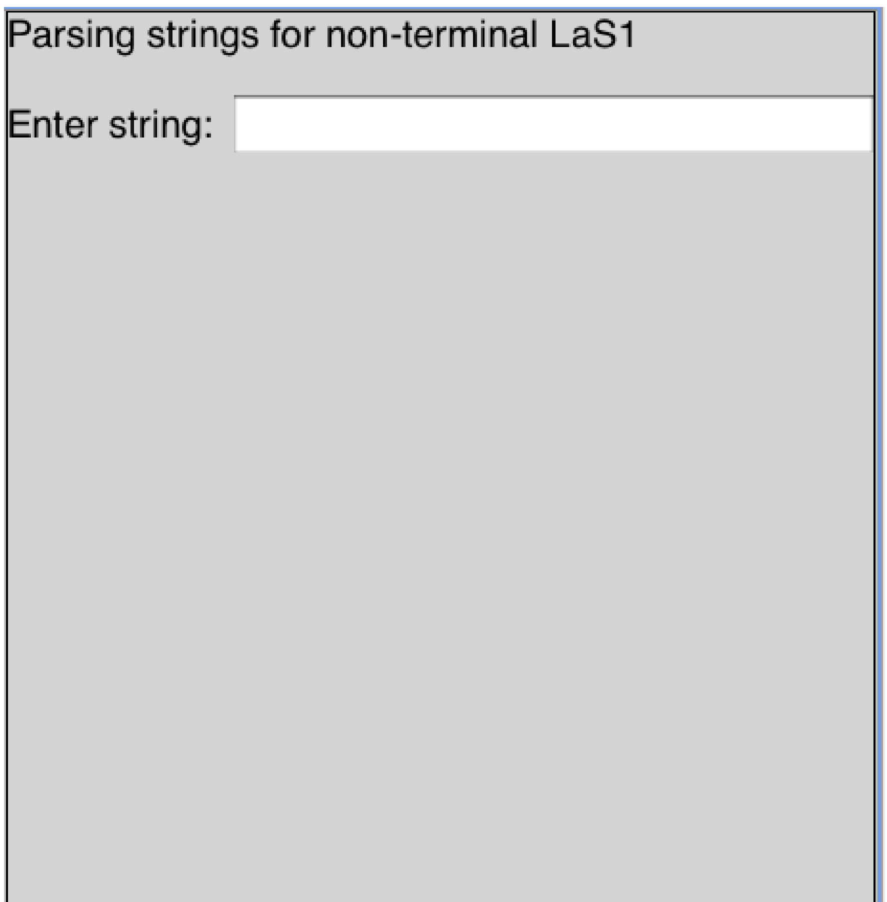


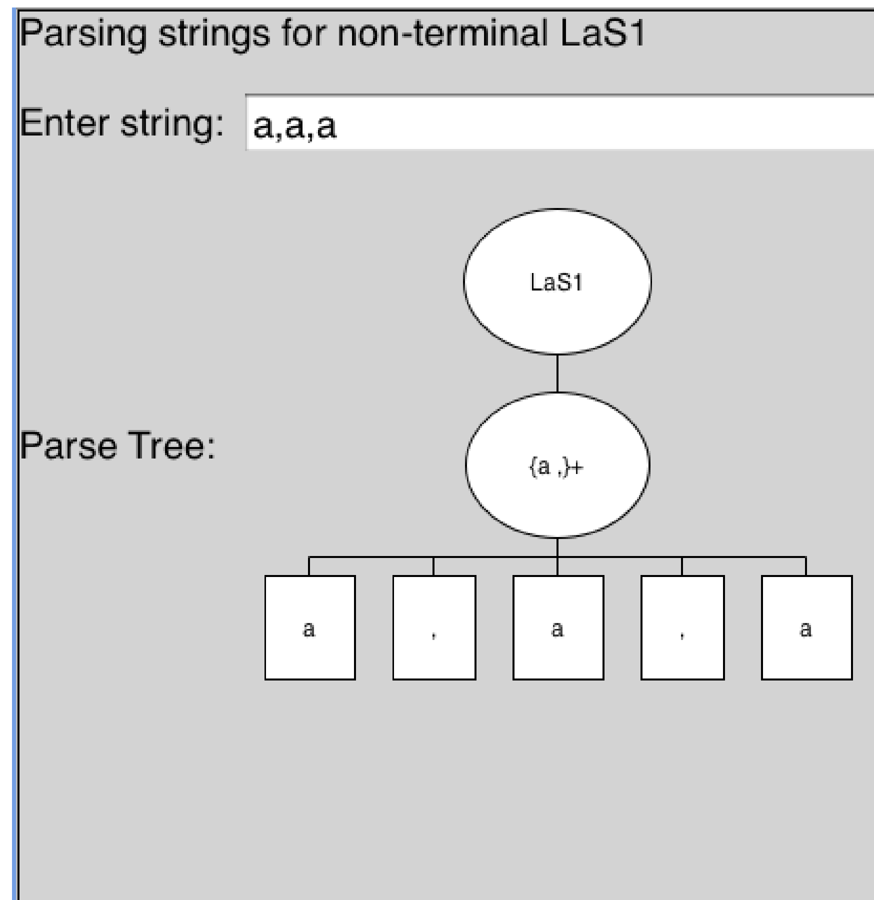



```
parseTreeViewer(#start[LaS0])
```

Language LaS1: List of one or more a's separated by comma's

```
start syntax LaS1 = {"a" ","}+;
```

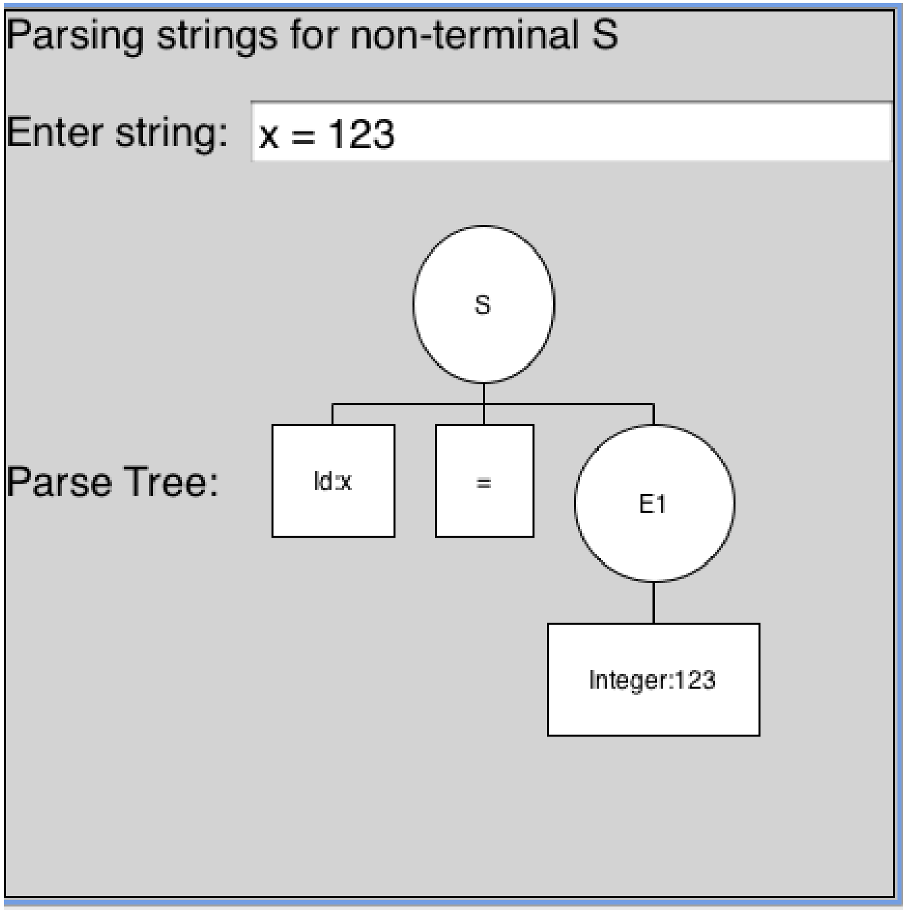





```
parseTreeViewer(#start[LaS1])
```

Language S: Statement-like

```
layout Whitespace = [\ \t\n]*;

lexical Integer = [0-9]+;
syntax E1 = Integer
          | E1 "*" E1
          > E1 "+" E1
          | "(" E1 ")"
          ;
lexical Id = [a-z][a-z0-9]*;

start syntax S =
     Id "=" E1
   | "while" E1 "do" {S ";"}+ "od"
   ;
```




```
parseTreeViewer(#start[S])
```

What is a grammar?

- A context-free grammar is 4-tuple G=(N,Σ,P,S)
- N is a set of nonterminals
- Σ is a set of terminals (literal symbols, disjoint from N)
- P is a set of production rules of the form (A, α) with A a nonterminal, and α a list of zero or more terminals or nonterminals. Notation:
  - A ::= α (in BNF)
  - `syntax A = α;` (in Rascal)
- S ε N, is the start symbol.

Derivations

- A grammar is a formal system with one proof rule:
  - $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if A ::= γ is a production
  - A is a nonterminal, α, β, γ possibly empty lists of (non)terminals

Example

- N = {E}
- Σ = { +, *, (,), -, a}
- S = E
- P = {E::=E+E, E::=E*E, E::=(E), E::=-E, E::= a}
- A derivation:
  - E => - E => - (E) => - (E + E) => - (a + E) => - (a + a)
- A derivation generates a sentence from the start symbol

Exercise

- Give a derivation for a + a * a

Language defined by a Grammar

- Extend the one step derivation $\Rightarrow$ to
  - $\Rightarrow^*$ derive in *zero* or more steps
  - $\Rightarrow^+$ derive in *one* or more steps
- The language defined by a grammar
  $G = (N,\Sigma,P,S)$ is:
  - $L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^+ w \}$
- A sentence $w \in L(G)$ only contains terminals

Derivations

- At each derivation step there are choices:
  - Which nonterminal will we replace?
  - Which alternative of the selected nonterminal will we apply?
- Two choices:
  - Leftmost: always select leftmost nonterminal
  - Rightmost: always select leftmost nonterminal

Examples

- Recall our -(a+a) example
- Leftmost derivation of -(a+a):
  - E => - E => - (E) => - (E + E) => - (a + E) => - (a + a)
- Rightmost derivation of -(a+a):
  - E => - E => - (E) => - (E + E) => - (E + a) => - (a + a)

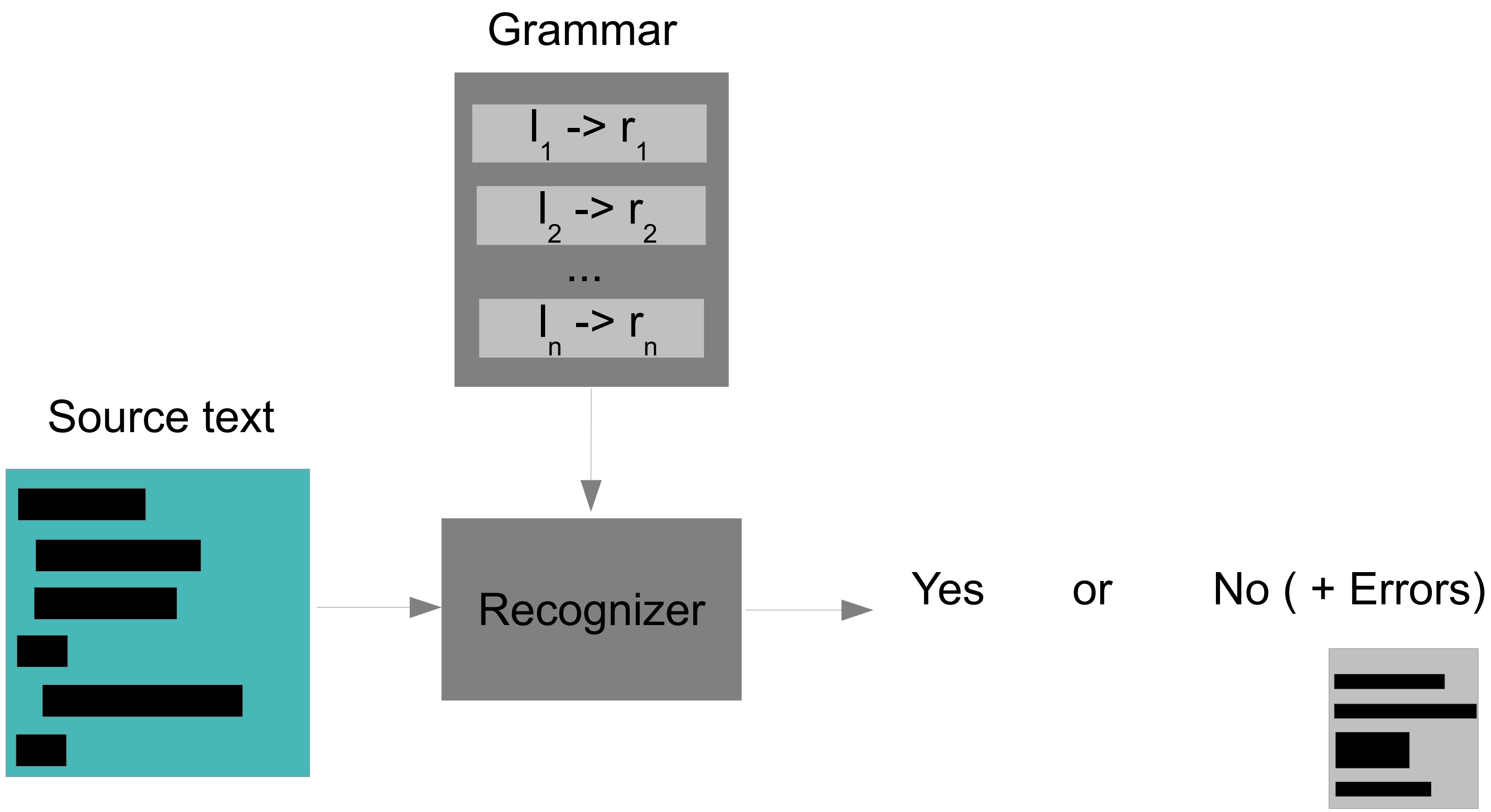
Derivation versus parsing

- A derivation generates a sentence from the start symbol
- A recognizer does the inverse: it deduces the start symbol from the sentence
- Leftmost derivation leads to a topdown recognizer (LL parser)
- Rightmost derivation leads to a bottom-up recognizer (LR parser)

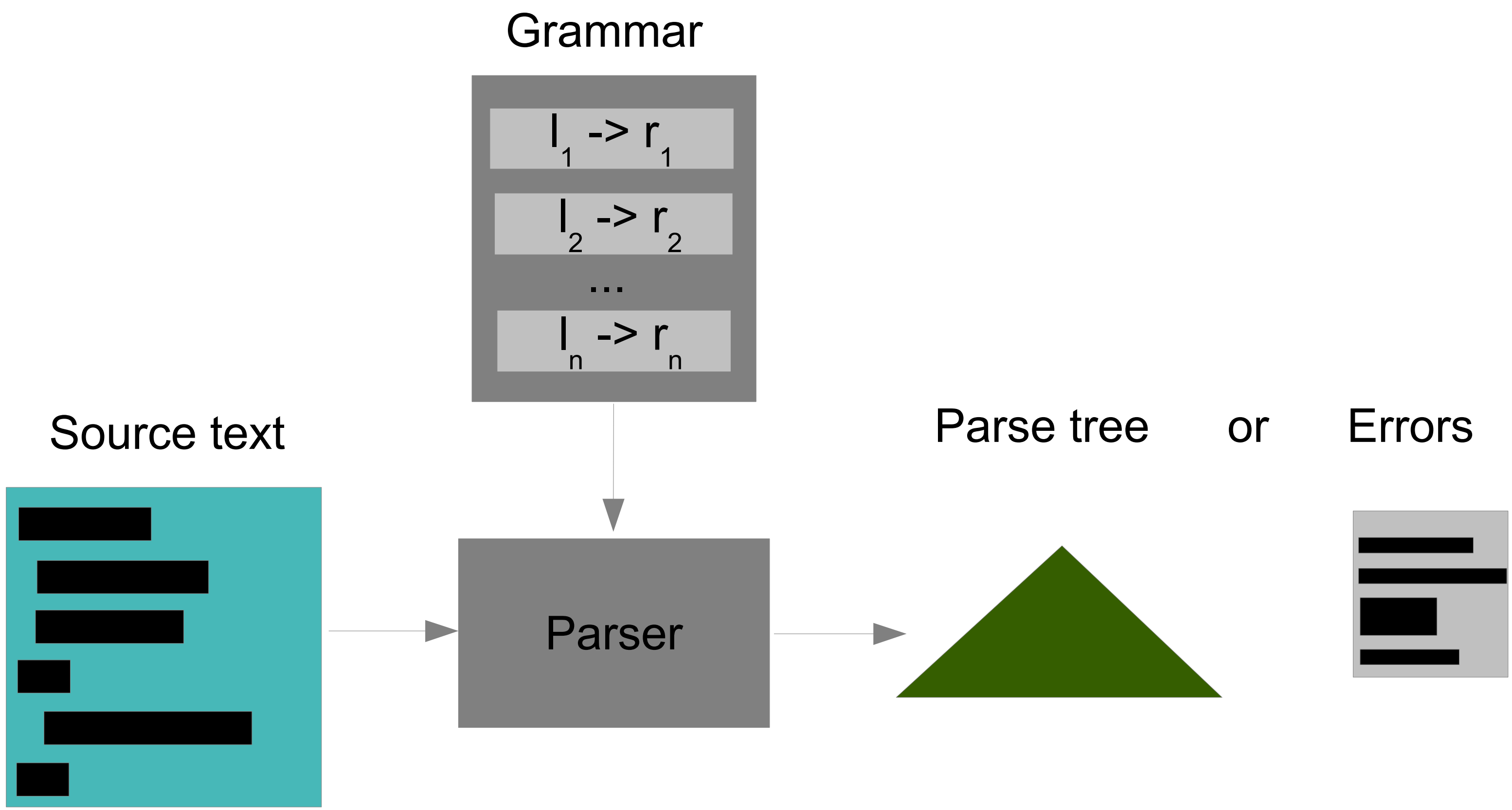
Recognizing versus Parsing

- Recognizer:
  - Is this string in the language?
- Parser:
  - Is this string in the language?
  - If so, return a syntax tree
- Generalized Parser:
  - Idem, but may return more than one tree
  - Accepts larger class of grammars

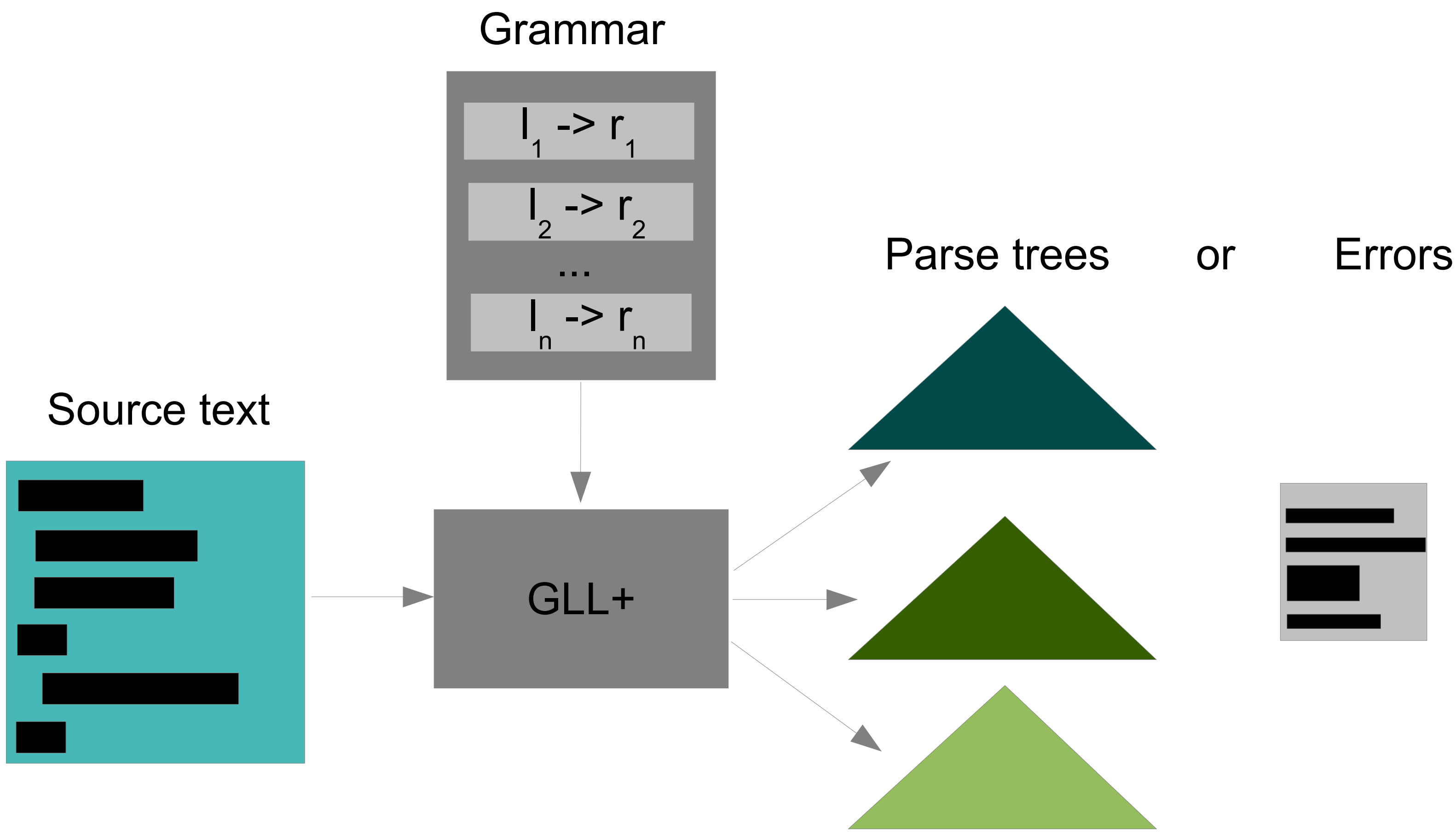
A Recognizer



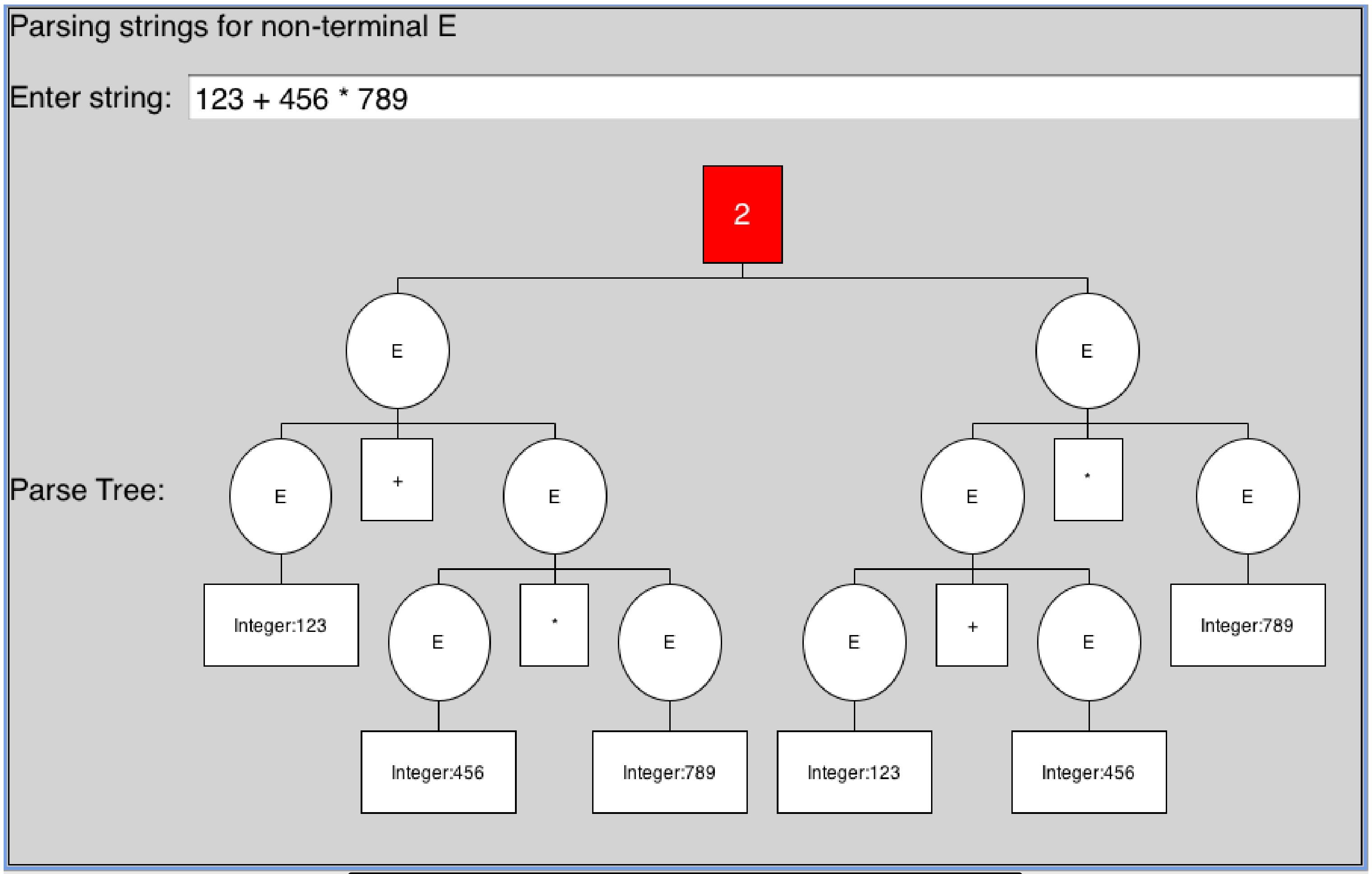
A Parser


Grammar
l1 -> r1
l2 -> r2
...
ln -> rn
Source text
Parser
Parse tree
or
Errors

Generalized Parser (as used in Rascal)



Recall Language E



`parseTreeViewer(#start[E])`

General Parsing Approaches

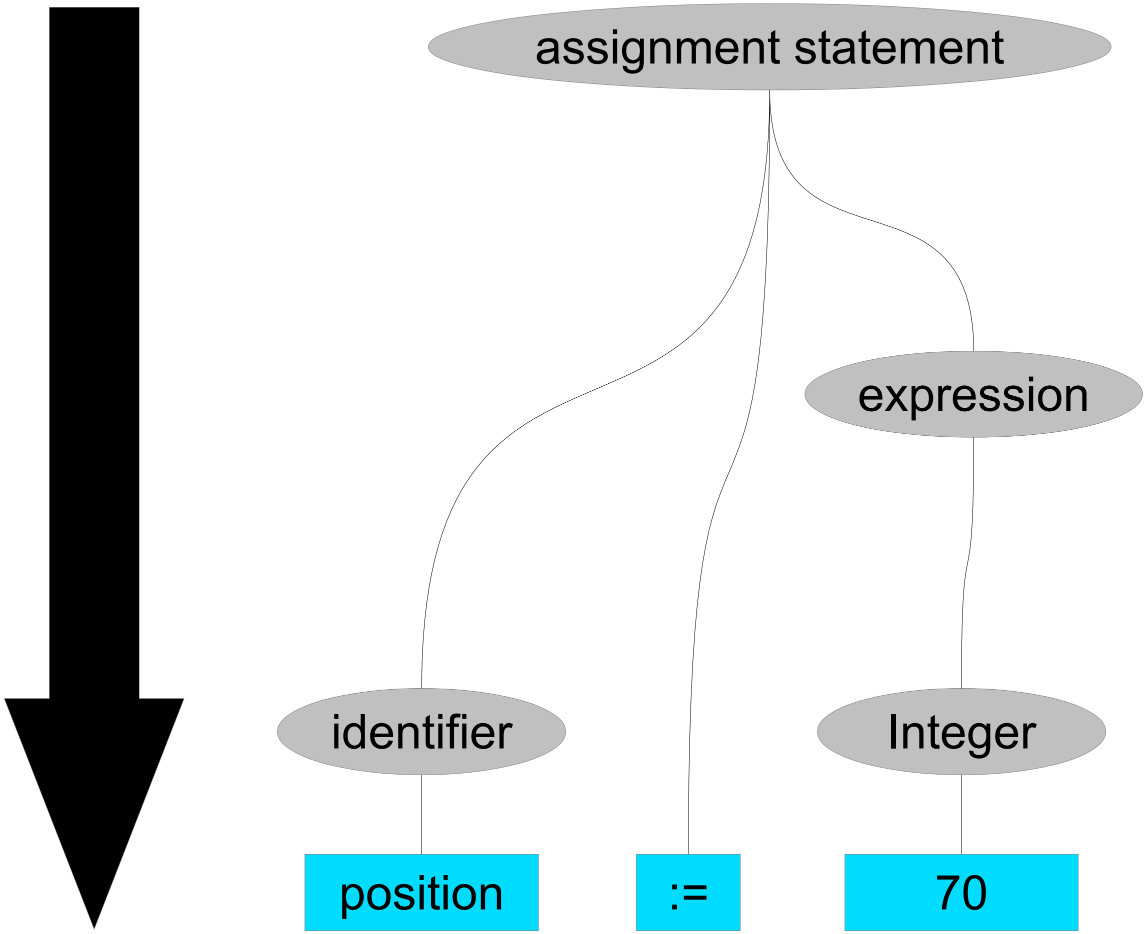
- Top-Down (predictive)
  - Predict what you want to parse, and verify the input
  - Leftmost derivation
- Bottom-Up
  - Recognize token by token and infer what you are recognizing by combining these tokens.
  - Rightmost derivation
- The type of grammar determines the parsing techniques that can be used

Parsing techniques

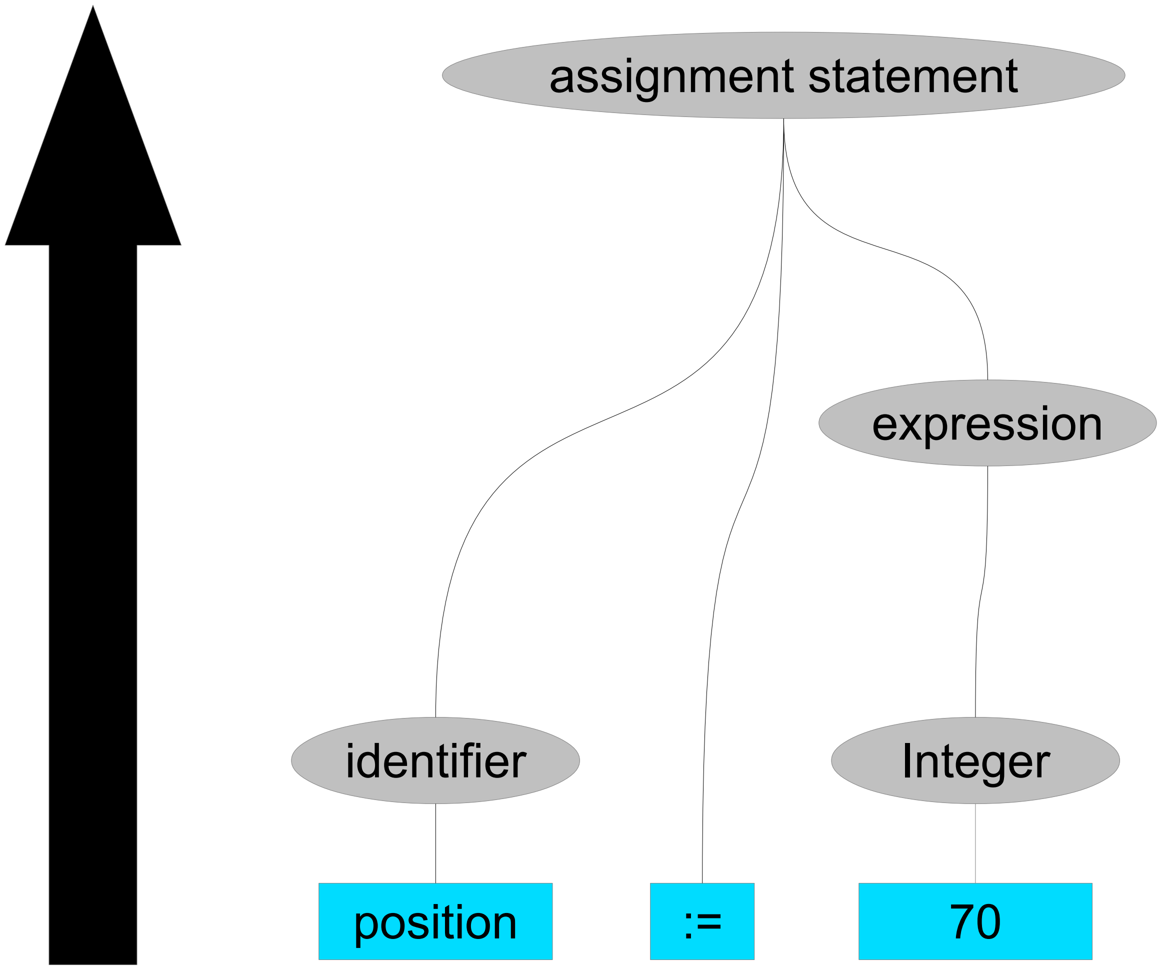
- Top-Down
- **LL**(**1**): **L**eft-to-right, **L**eftmost derivation, 1 symbol lookahead
- LL(k), k symbols lookahead
- ...

- Bottom-Up
- **LR**(1): **L**eft-to-right, **R**ightmost, 1 symbol lookahead
- LR(k)
- LALR(k)
- SLR(k)
- ...

Top-Down Parser



Bottom-Up Parser



How do we get a parser?

Write it by hand

- Pro: large flexibility
- Pro: each mixing of parsing with other actions
  - Type checking
  - Tree building
- Pro: specialized error messages/error recovery
- Con: more effort
- Con: reprogramming needed when grammar changes
- Con: unclear which language is recognized

Example: Writing a Parser

```
syntax A = "a";
syntax B = "b";
start syntax C = "c" | A C B;
```

Idea: implement three functions

bool parseA()

bool parseB()

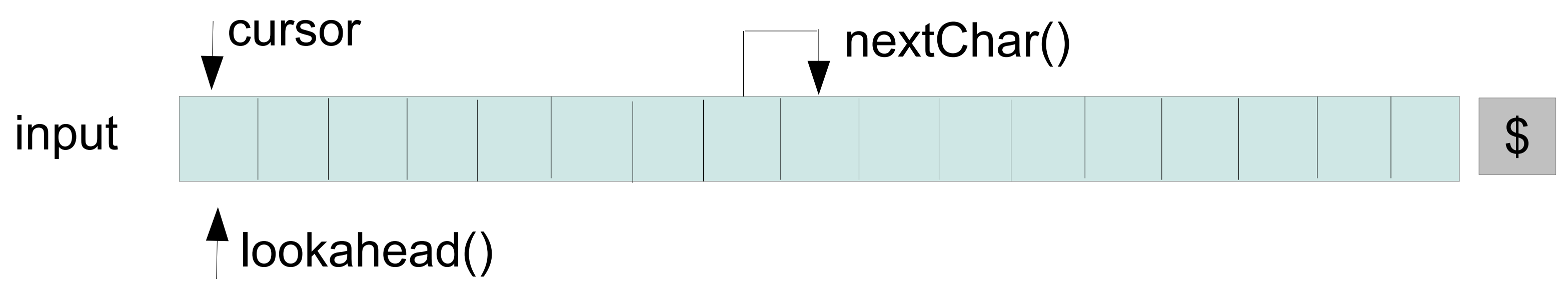
bool parseC()

that parse the corresponding non-terminal

Infrastructure

```
syntax A = "a";
syntax B = "b";
start syntax C = "c" | A C B;
```

```
str input = "";
int cursor = -1;

private void initParse(str s) { input = s; cursor = 0; }

private str lookahead() = cursor < size(input) ? input[cursor] : "$";

private void nextChar(){
     cursor += 1;
}

public bool endOfString() = lookahead() == "$";

public bool parseC(str s){
  initParse(s);
  return parseC() && endOfString();
}
```



Parser

```
syntax A = "a";
syntax B = "b";
start syntax C = "c" | A C B;
```

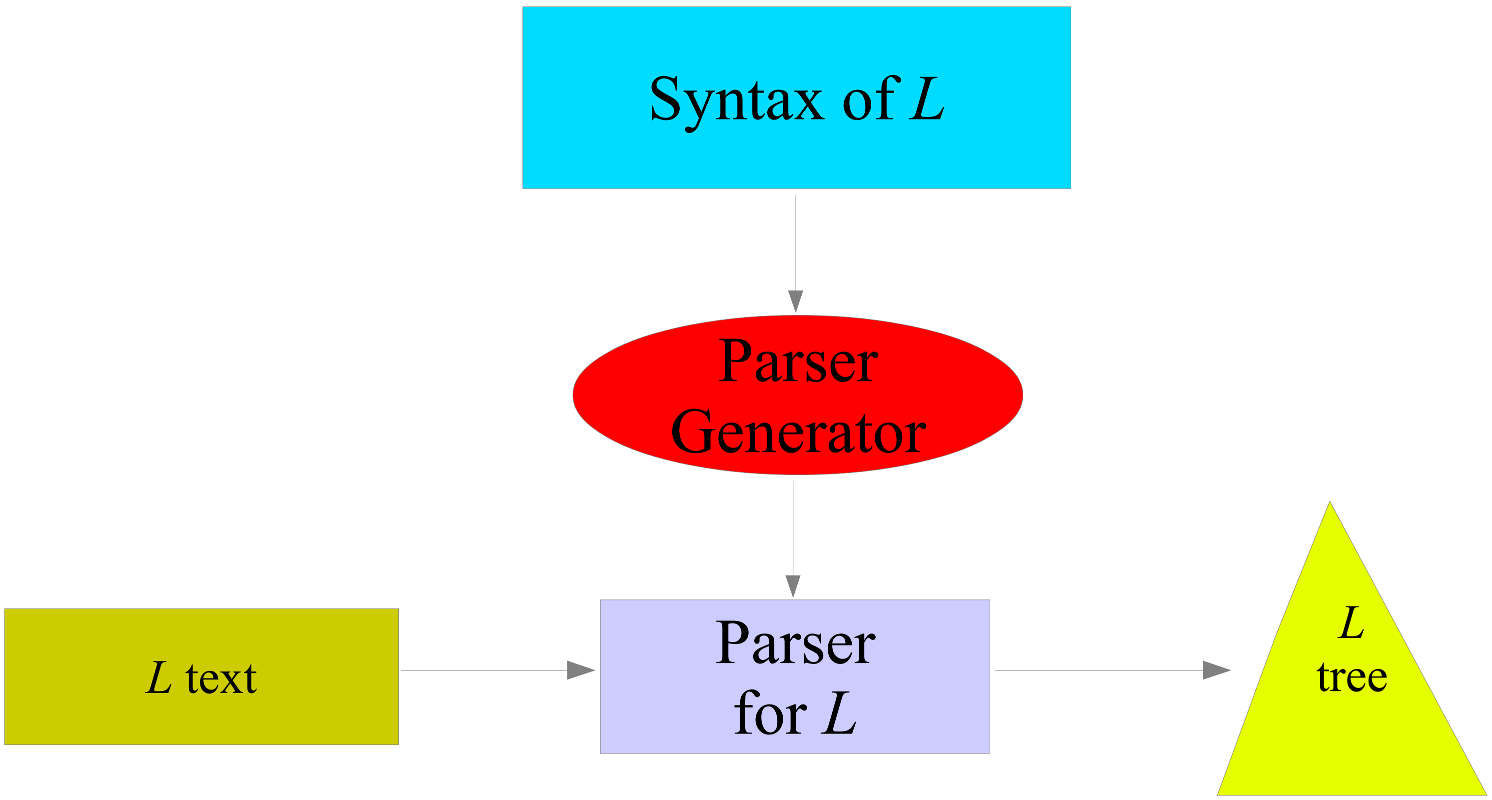
```
public bool parseTerm(str term){
  if(lookahead() == term){
     nextChar();
     return true;
  }
  return false;
}

public bool parseA() = parseTerm("a");

public bool parseB() = parseTerm("b");

public bool parseC(){
   if(lookahead() == "c")
      return parseTerm("c");
   if(lookahead() == "a"){
      parseA();
      if(parseC()){
         if(lookahead() == "b"){
            return parseB();
         }
      }
   }
   return false;
}
```

```
rascal>parseC("aaacbbb")
bool: true

rascal>parseC("aaacbb")
bool: false
```

Trickier Cases

- Fixed lookahead > 1 characters
- No fixed lookahead
- Alternatives overlap partially:
  - Naive approach tries first alternative and then fails but another alternative may match.
  - A backtracking approach tries each alternative and if it fails it restores the input position and tries other alternatives.
- Generalized parsing approach: try alternatives in parallel

Automatic Parser Generation



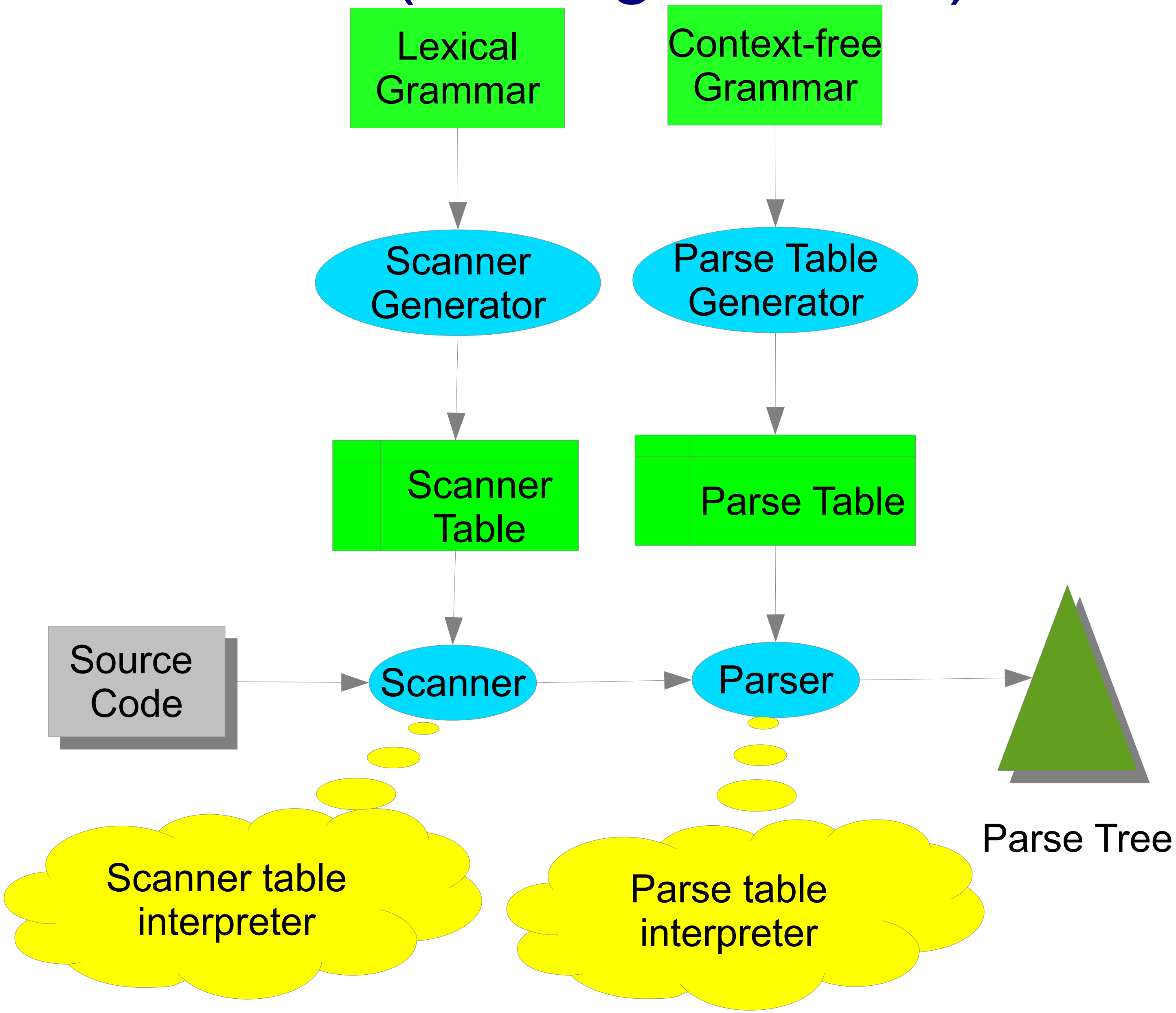
Some Parser Generators

- Bottom-up
  - Yacc/Bison, LALR(1)
  - CUP, LALR(1)
  - SDF, SGLR
- Top-down:
  - ANTLR, LL(k)
  - JavaCC, LL(k)
  - Rascal, GLL+
- Except SDF and Rascal, all depend on a scanner generator

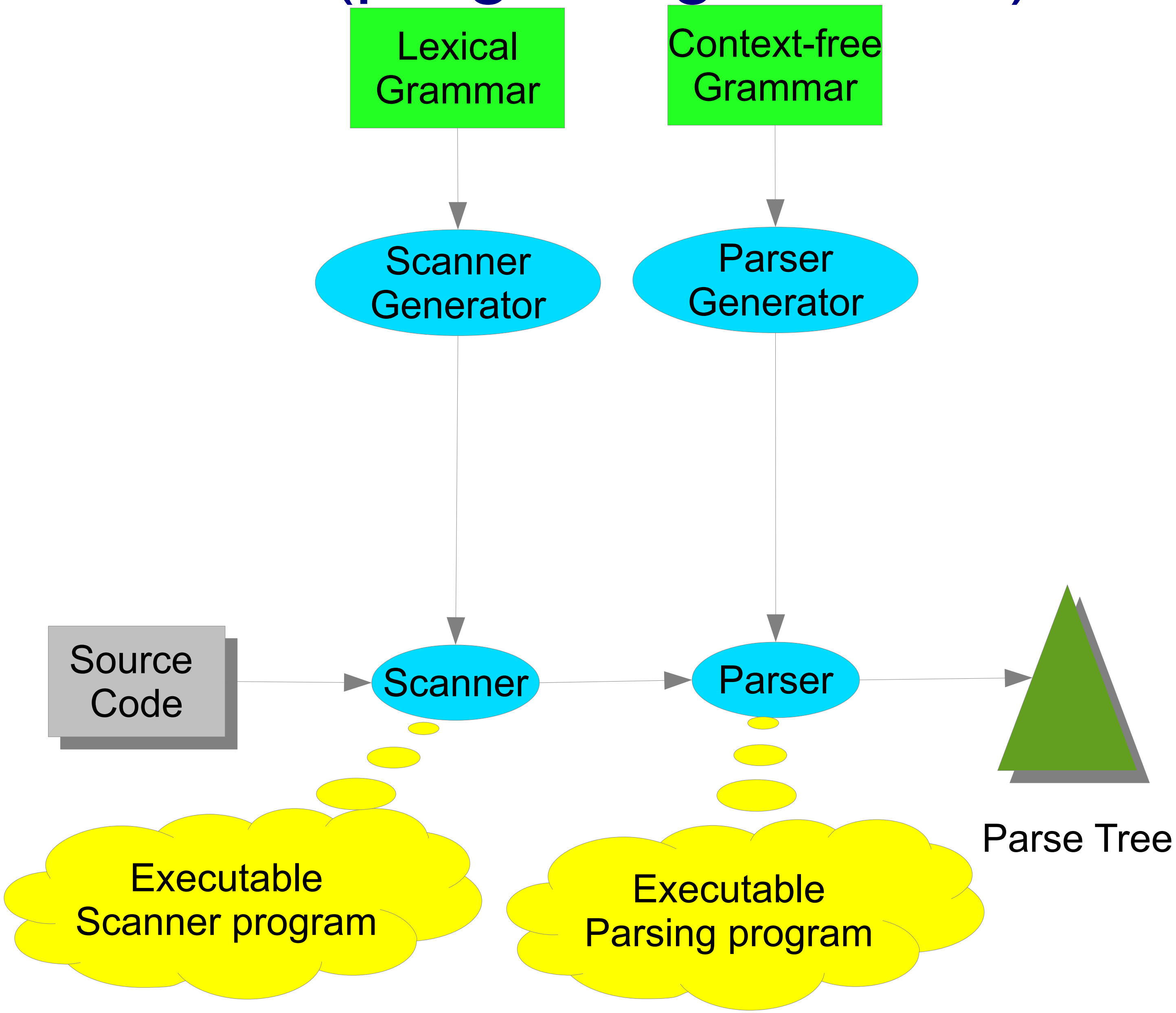
Assessment parser implementation

- 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

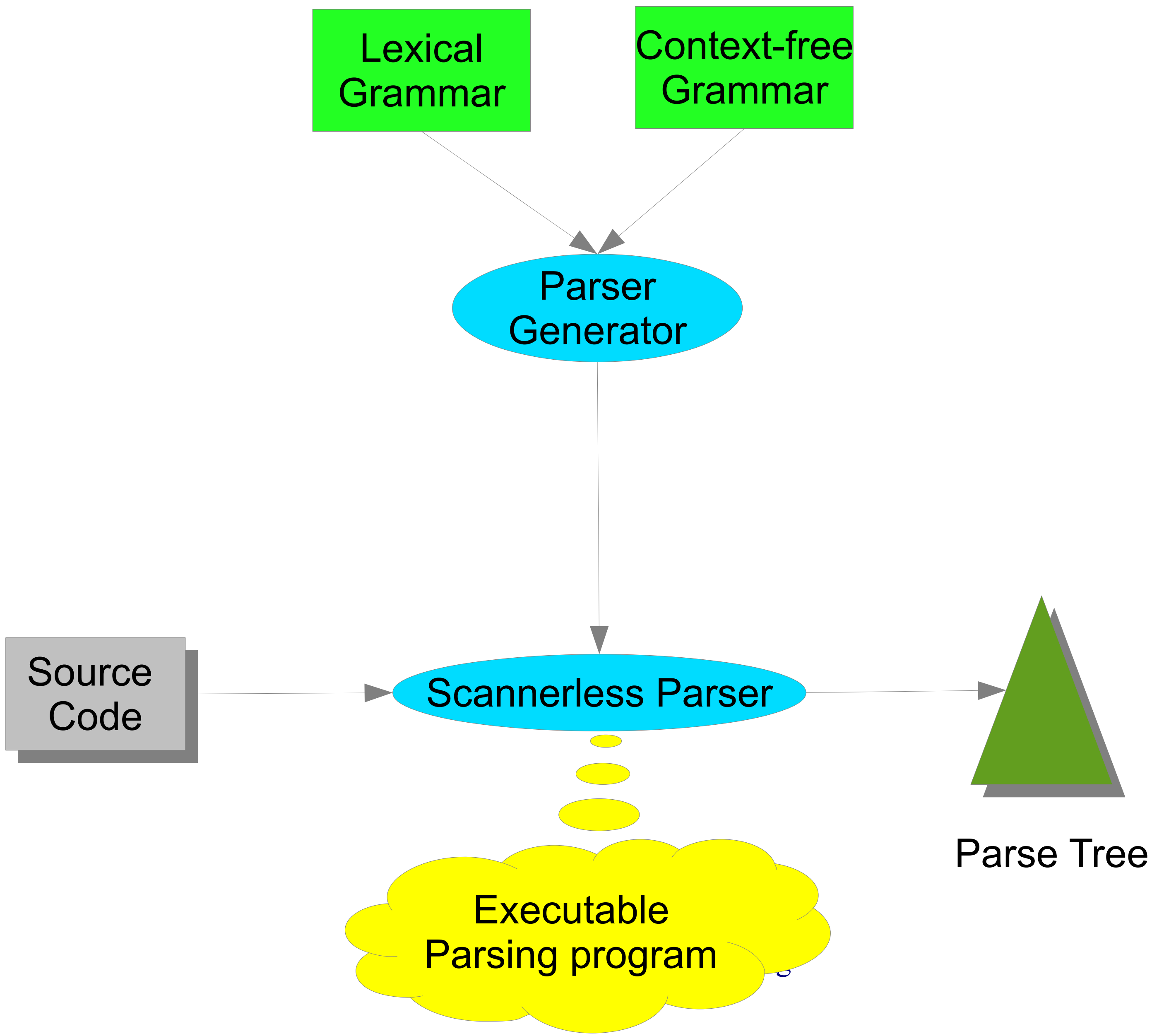
Parser Generation Architecture
(table-generator)



Parser Generation Architecture
(program-generator)



Parser Generation Architecture



Pragmatic Issues

- How do I get my grammar in a form accepted by the parser generator:
  - Rewriting, refactoring, renaming, ...
  - May be very hard (or impossible!)
- How does the scanner get its input?
- How are scanner and parser interfaced?
- How are actions attached to grammar rules?
  - Semantic actions in C/Java code + Interface variables
- How to define error recovery?

Parsing in Rascal

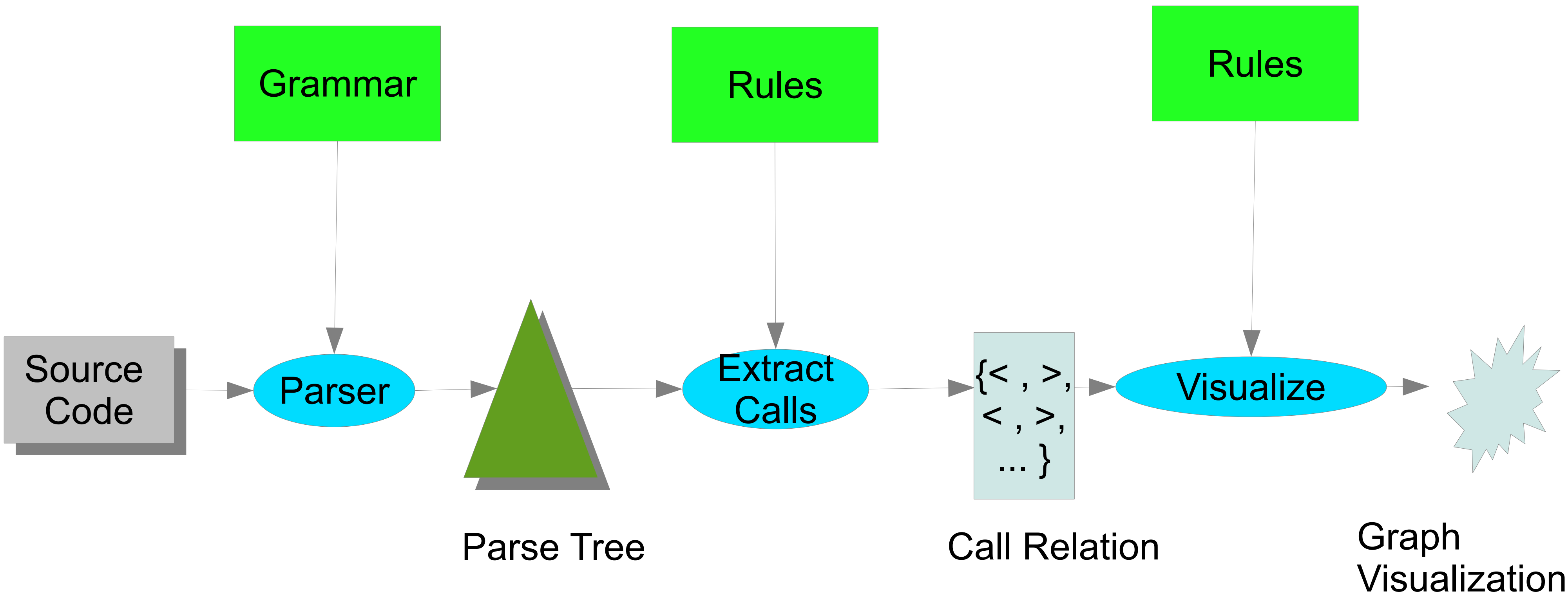
- Scannerless, GLL+ parser
- Grammars can easily be composed (this is not possible with other technologies)
- Parsing and executing Rascal code can be mixed.
- Work in progress: error recovery.

Conclusions

- Parsing is a vital ingredient for many systems
- Formal languages, grammars, etc. are a well-established but rather theoretical part of computer science. Learn the basic notions!
- Not always easy to get to grips with a specific parsing technology
  - Grammar rewriting/refactoring is difficult
- Rascal's scannerless GLL+ parser makes this unnecessary.

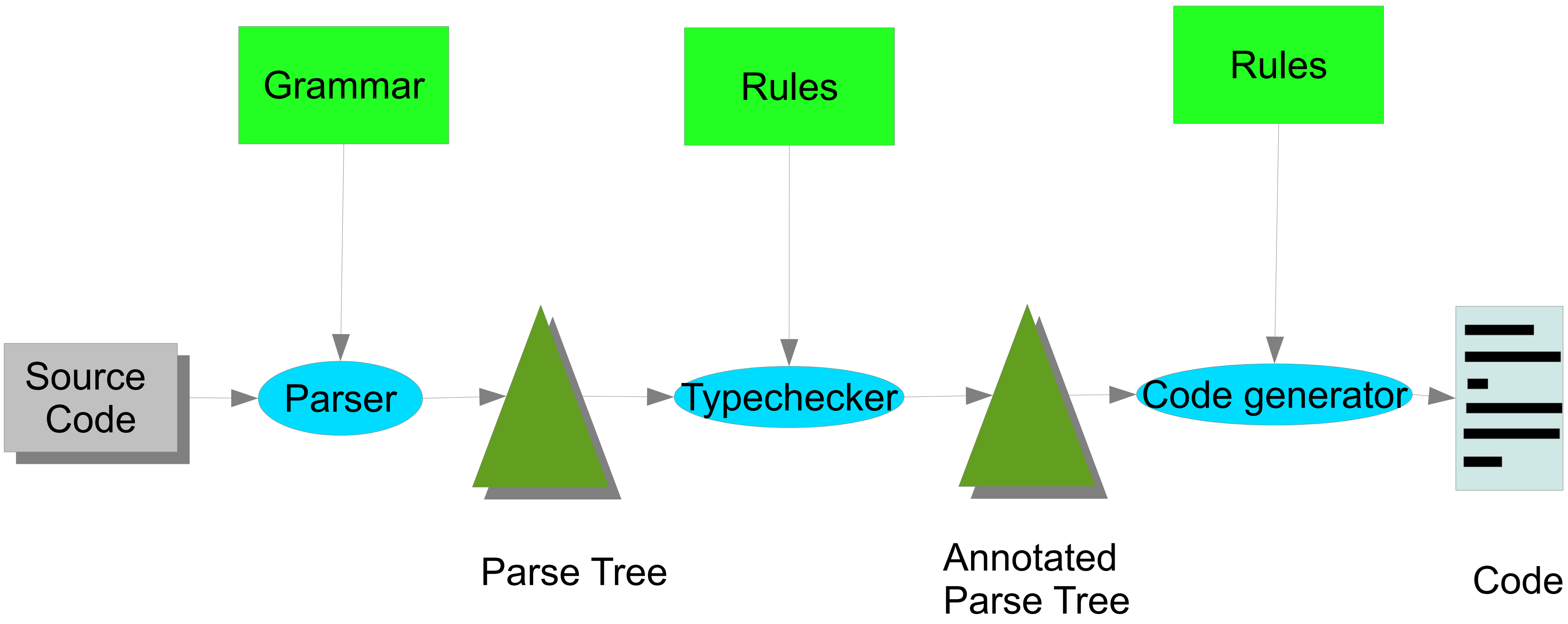
Parser in a Bigger Picture

Call Graph Visualization

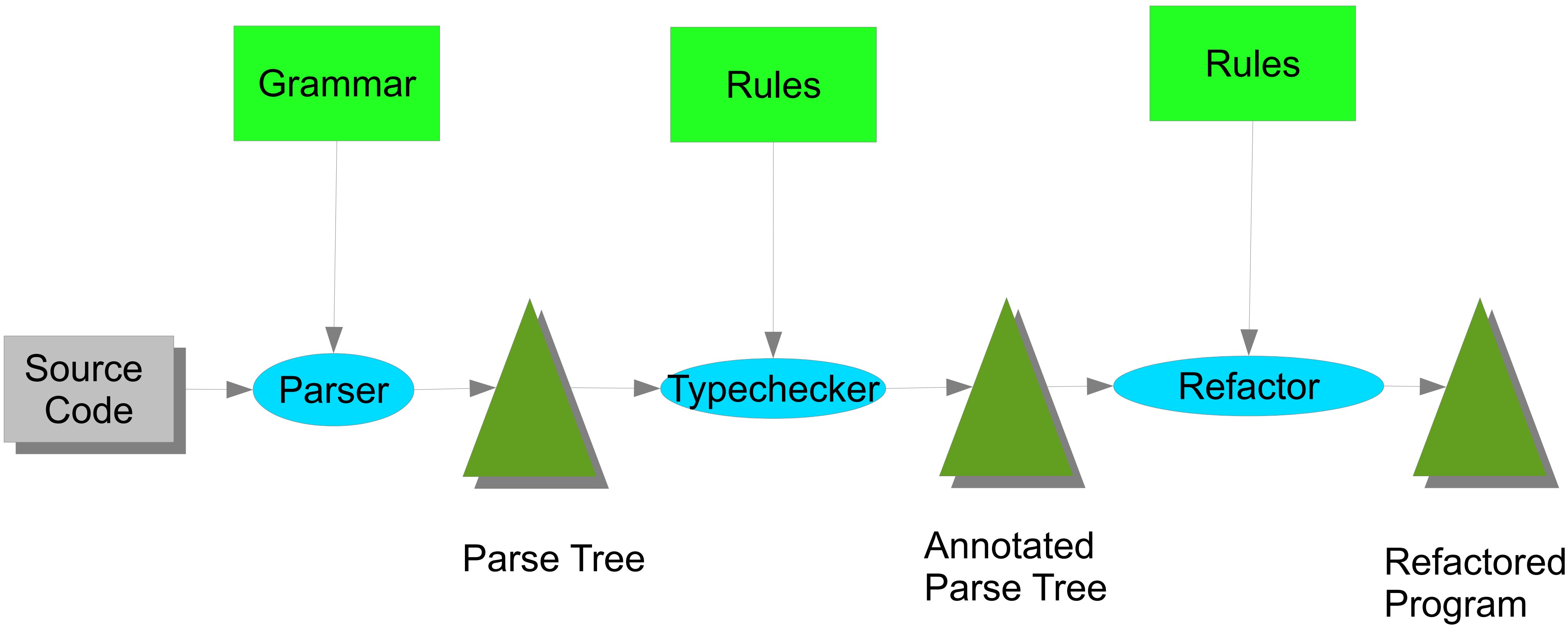


Parser in a Bigger Picture

Compiler



Parser in a Bigger Picture
Refactoring



Further Reading

- http://en.wikipedia.org/wiki/Chomsky_hierarchy
- D. Grune & C.J.H. Jacobs, *Parsing Techniques: A Practical Guide*, Second Edition, Springer, 2008
- Tutor/Rascalopedia (Grammar, Language, LanguageDefinition)
- Tutor/Rascal (Concepts/SyntaxDefinitionAndParsing, Declarations/Syntaxdefinition)
- Tutor/Recipes/Languages