Textual modeling languages

Oszkár Semeráth, Kristóf Marussy

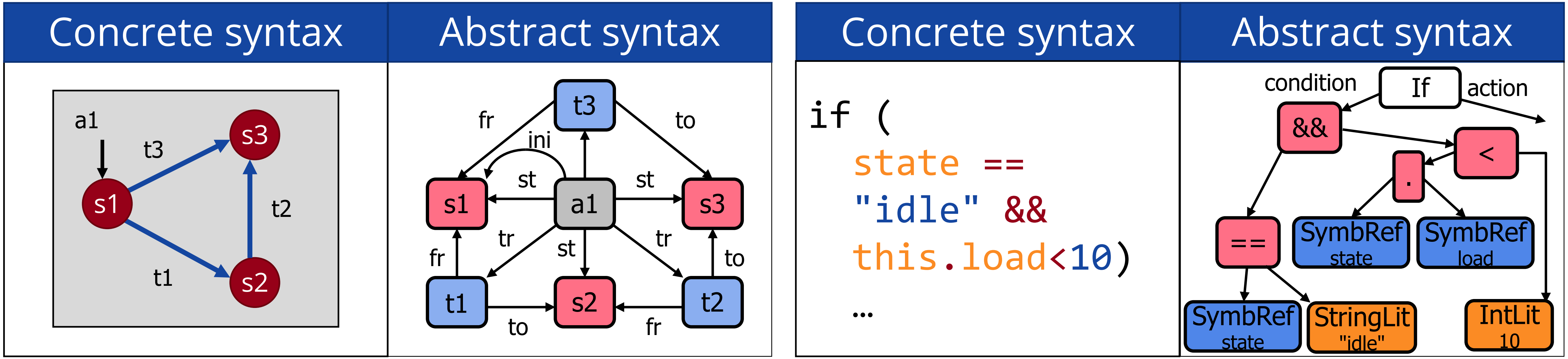
MŰEGYETEM 1782

BME FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

ftsrg Critical Systems Research Group

Last lecture: Concrete and Abstract Syntax

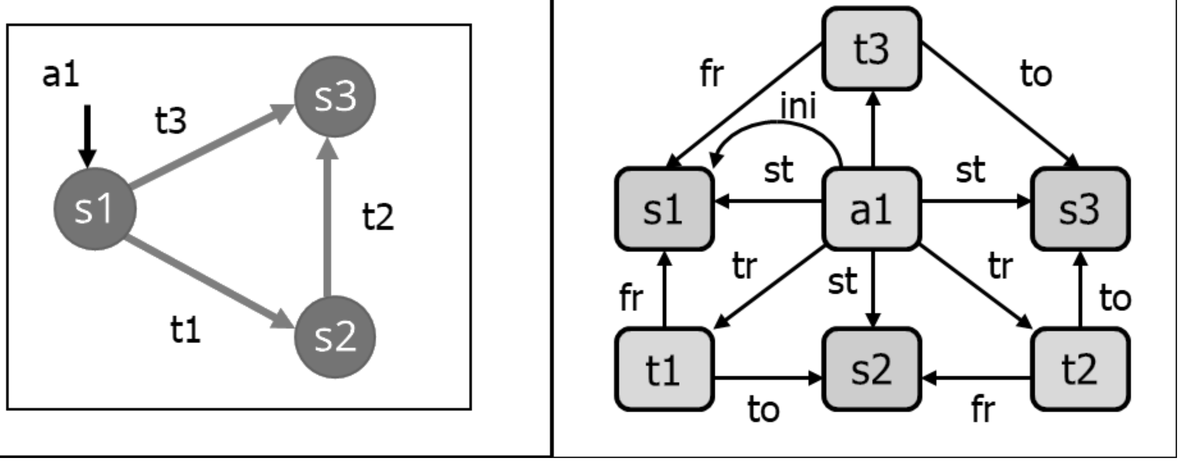
- **Question:** How to capture models?
- **Answer:** graph-based structures!



Last lecture: Concrete and Abstract Syntax

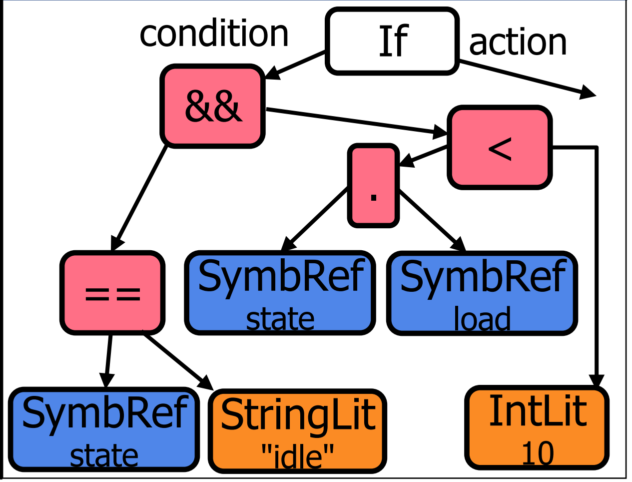
- **Question:** How to capture models?
- **Answer:** graph-based structures!

Concrete syntax	Abstract syntax



Concrete syntax	Abstract syntax

```
if (
  state ==
  "idle" &&
  this.load<10)
  …
```



- **Todays Topic:** Textual syntax and textual editors

ftsrg

Grammars, Context-Free Parsing

Theoretical Background

Conceptual overview

1. There are textual documents that needs to be parsed.
2. Some documents are valid, some are invalid. We want to efficiently capture which textual documents are valid.
   → **Grammars**
3. We want to create editors for valid document.
   → **Parsers**
4. For each document, we want to construct a model.
   → **Abstract syntax graphs**

Terminology

- Alphabet (Σ): Set of possible symbols in the document, e.g.,
  - *{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}* a numerical field
  - ASCII characters textual documents
  - *{a, b}*, then we can use only those two letters
- Words (Σ^*): Sequence of symbols, e.g.,
  - *“0012”, “222”, “1”*, or *“ ”* also called ε
  - *“class person { }”*
  - worlds like *“babababbbba”*
- Language (L): a subset of all words ($L \subseteq \Sigma^*$)
  - We do not like *“0012”*, but we like “222”
  - We do like *“class person { }”* but not *“class person { { }”*
- **How to capture a language?**

Formal Grammar

- Formal grammar
  G = (N, T, P, S)
  - N: nonterminal symbols
  - T: terminal symbols (alphabet)
  - P: production rules
  - S: start symbol ($S \in N$)

Example: G = (N, T, P, S)

- Num → Digit Num ← *Traditionally first rule is Start symbol*
- Num → Digit
- Digit → 0 |1 | 2 … | 9

Formal Grammar

- Notation:
  - A, B, C: nonterminals in N,
  - a, b, c: terminals in T
  - $\alpha,\beta,\gamma \in (T \cup N)^*$
- Regular rules:
  - $B \rightarrow a$
  - $B \rightarrow aC$

EASY

- Context-free (CF) rules:
  - $B \rightarrow \alpha$
  - $B \rightarrow \varepsilon$ (empty)

Theoretical background
+ Good technologies

- Condext-dependent rules:
  - $\alpha \rightarrow \beta$

Same power as a computer

- There are languages that cannot be captured by rules ☠

Derivation and Language

Num	
Digit Num	Num → Digit Num
1 Num	Digit → 1
1 Digit Num	Num → Digit Num
1 9 Num	Digit → 9
1 9 Digit Num	Num → Digit Num
1 9 Digit Digit	Num → Digit
1 9 Digit 6	Digit → 6
1 9 7 6	Digit → 7

- **Derivation step** using grammar G = (T,N,P,S)
  - $\alpha A \gamma \Rightarrow \alpha\beta\gamma$
  - applying production rule: $A \rightarrow \beta$
  - $\alpha A \gamma$, $\alpha\beta\gamma$: sentential forms
- **Derivation over G**: $S \Rightarrow^* w$ where
  - S: start symbol
  - $\Rightarrow^*$ transitive closure (apply as long as possible)
  - $w \in T^*$: sentence, i.e. string of terminals only
- **Language generated by G**
  - $L(G) = \{w \in T^* \mid$ there exists a derivation $S \Rightarrow^* w$ of $G\}$
  - Set of sentences derivable from S
- Parsing is polynomical algorithm for regular and context-free grammars
- In general, nonterminals can be resolved in arbitrary order (non-deterministic)
- Leftmost vs. Rightmost derivation: always resolve the left/rightmost nonterminal as next step

Binary Operations

Example: G = (N, T, P, S)

Exp → Num

Exp → Exp "+" Exp

Exp → Exp "-" Exp

Exp → Exp "*" Exp

Exp → Exp "/" Exp

Exp → "(" Exp ")"

Two Derivations of 1+2*3:

Exp	
Exp + Exp	Exp → Exp "+" Exp
1 + Exp	Exp → Num
1 + Exp * Exp	Exp → Exp "*" Exp
1 + 2 * Exp	Exp → Num
1 + 2 * 3	Exp → Num
Exp	
Exp * Exp	Exp → Exp "*" Exp
Exp * 3	Exp → Num
Exp + Exp * 3	Exp → Exp "+" Exp
Exp + 2 * 3	Exp → Num
1 + 2 * 3	Exp → Num

Lexing and Parsing

Lexer

- **Input**:
  - Regular grammar / RegExp
  - Character sequence: l,e,t, ,x,=,1,3,4,
- **Output**:
  - Token sequence: let,x,=,134,
  - Identify keywords, numbers, variables, comments

Grammar:

Num→ 0 Num|...| 9 Num Num→ 0 |...| 9

RegExp:

Num = Digit Digit*
Digit = (0| 1 | ... |9)

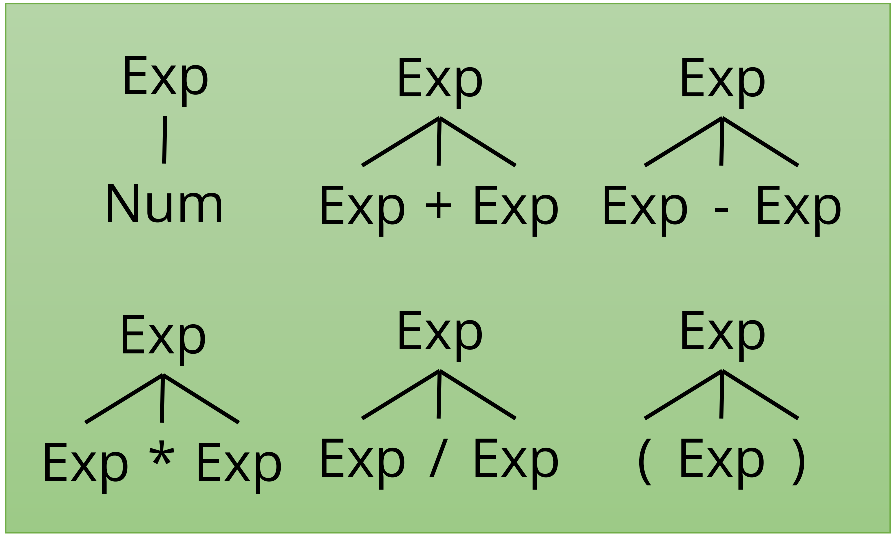
Parser

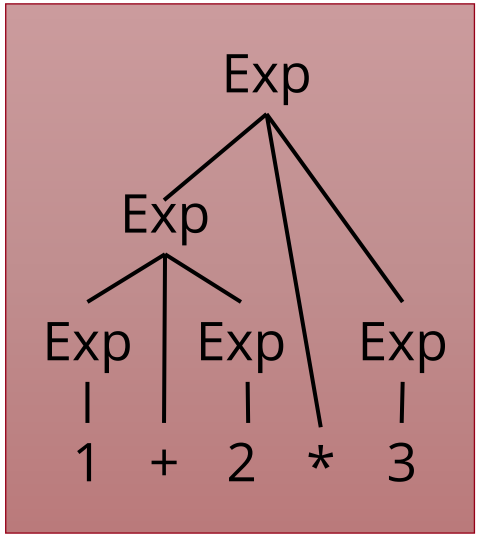
- **Input**:
  - CF grammar
  - Token sequence: let,x,=,134,
- **Output**
  - Abstract syntax tree (AST)
  - How to derive the token sequence according to grammar?

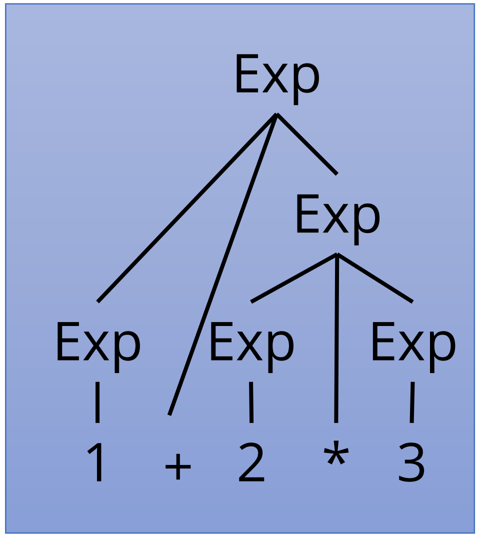
Grammar:

Exp → Num | Exp "+" Exp
| Exp "-" Exp | Exp "*" Exp
| Exp "/" Exp | "(" Exp ")"

Parse Tree Construction







Ambiguous derivation!
How to disambiguate?

- **Parse Tree:**
  - Parent node: nonterminals
  - Child node: nonterminals/terminals
  - Built up according to productions of the grammar

- **Ambiguous grammar**:
  - Generates two distinct parse trees
  - Serious problem for a parser!

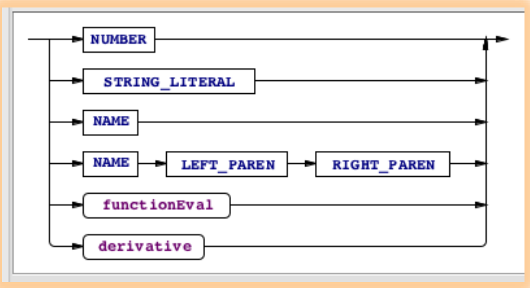
Idea: Change the grammar

Looking inside advanced IDEs

From parsers to development tools
Roundtrip property
Modern services

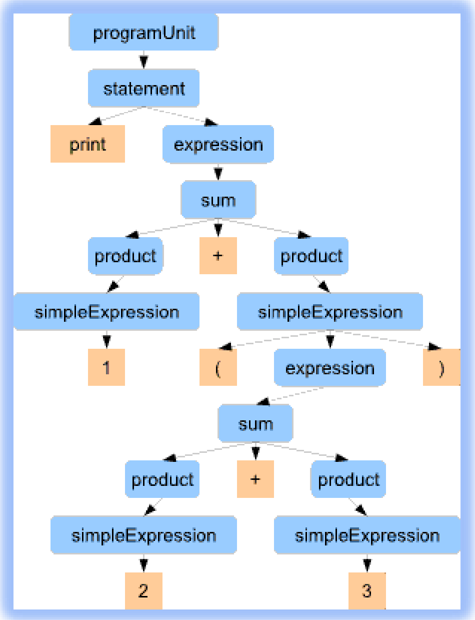
Parsers: The Traditional Setup

Grammar



Parsing
(lexer + parser)

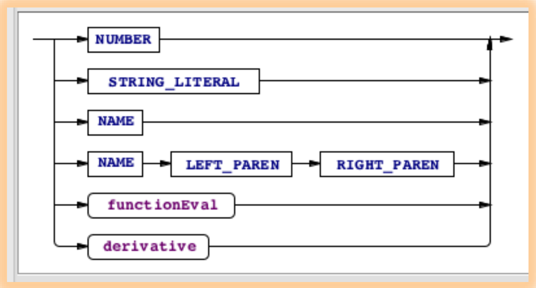
Source code of program



Abstract
syntax tree (AST)

Parsers: Setup with errors

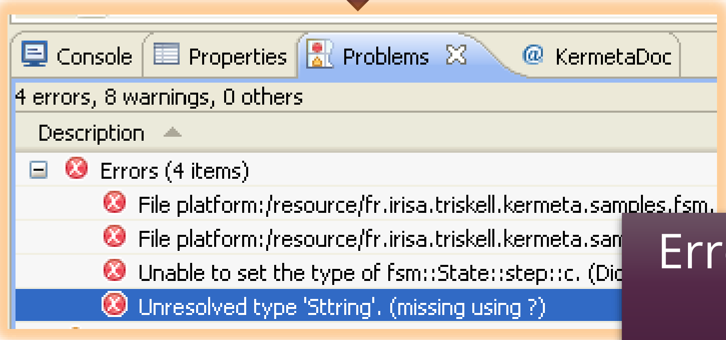
Grammar



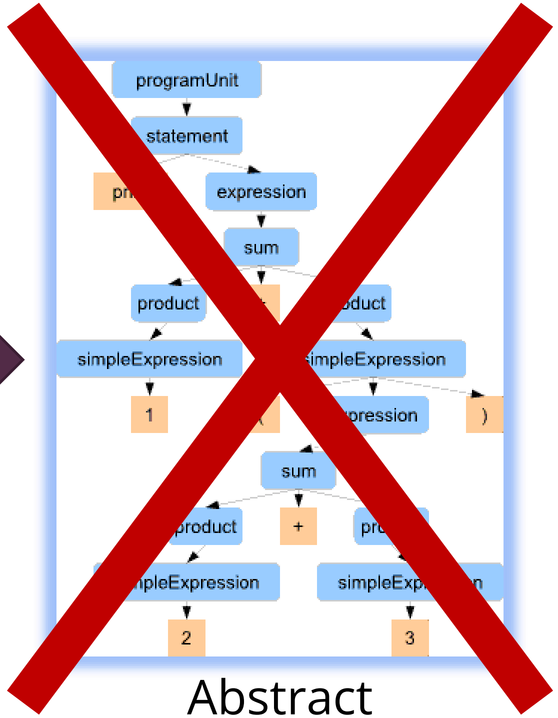
Parsing
(lexer + parser)

Source code of program

Error
report



Error recovery
parsing

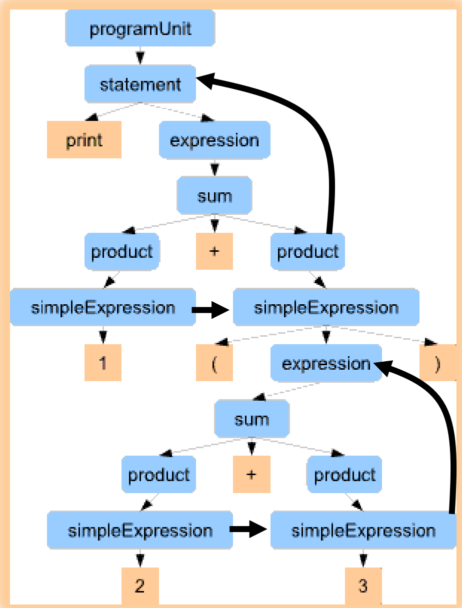


Abstract
syntax tree (AST)

View Generation + Program Analysis

Source code of program

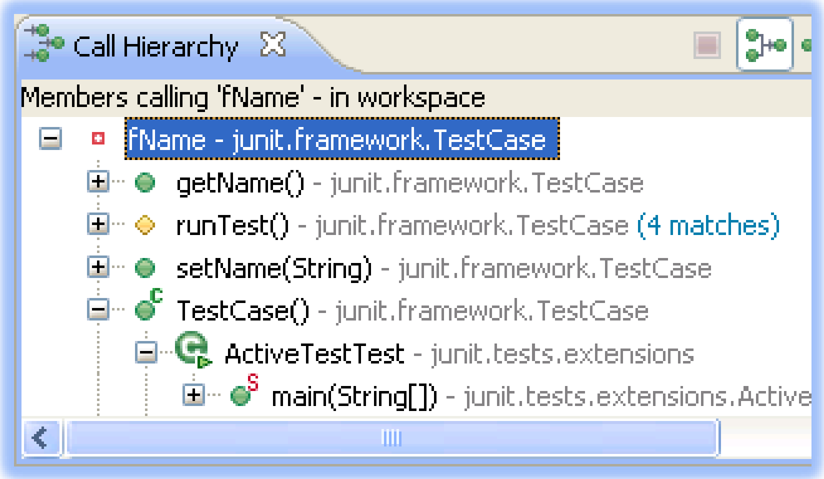
Parsing



AST: Abstract syntax tree

„Visitor”

Call graph



„Visitor”

Type hierarchy

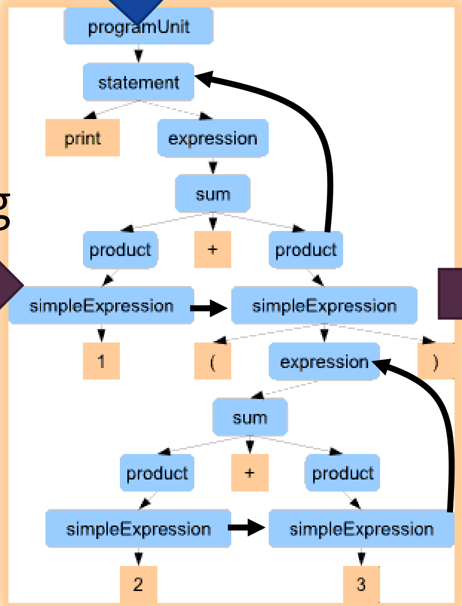
AST vs DOMs

Defined by a grammar

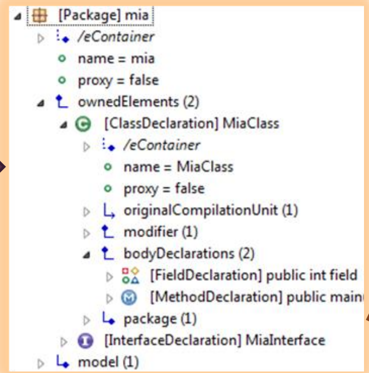
Defined by a metamodel

Source code of program

Parsing



AST: Abstract syntax tree

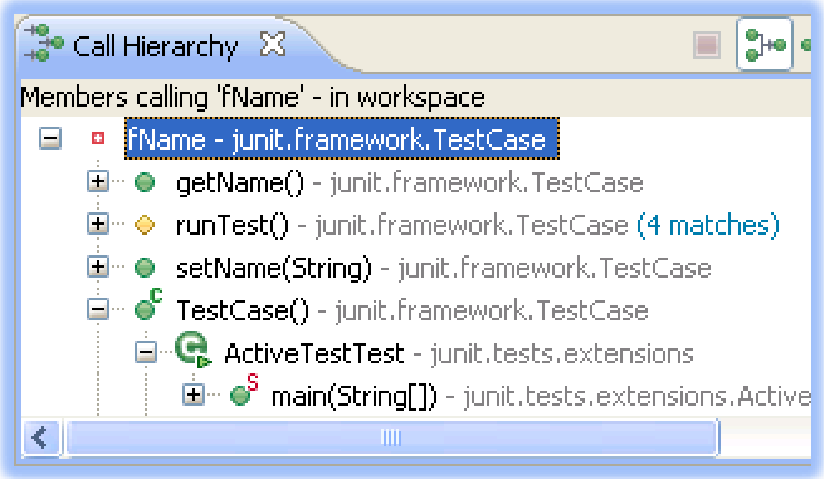


DOM: Document Object Model DOM

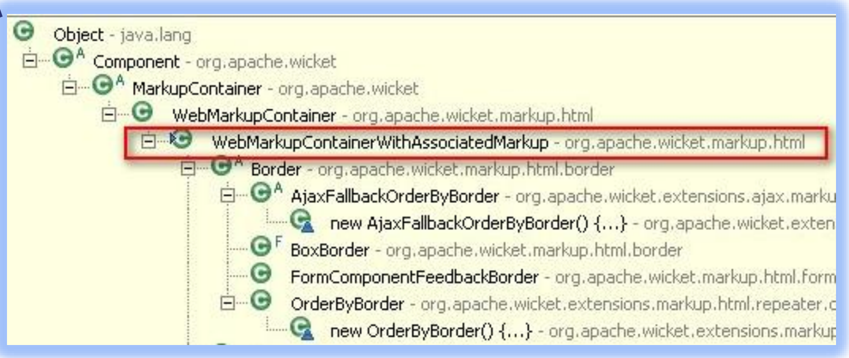
„Visitor"

„Visitor"

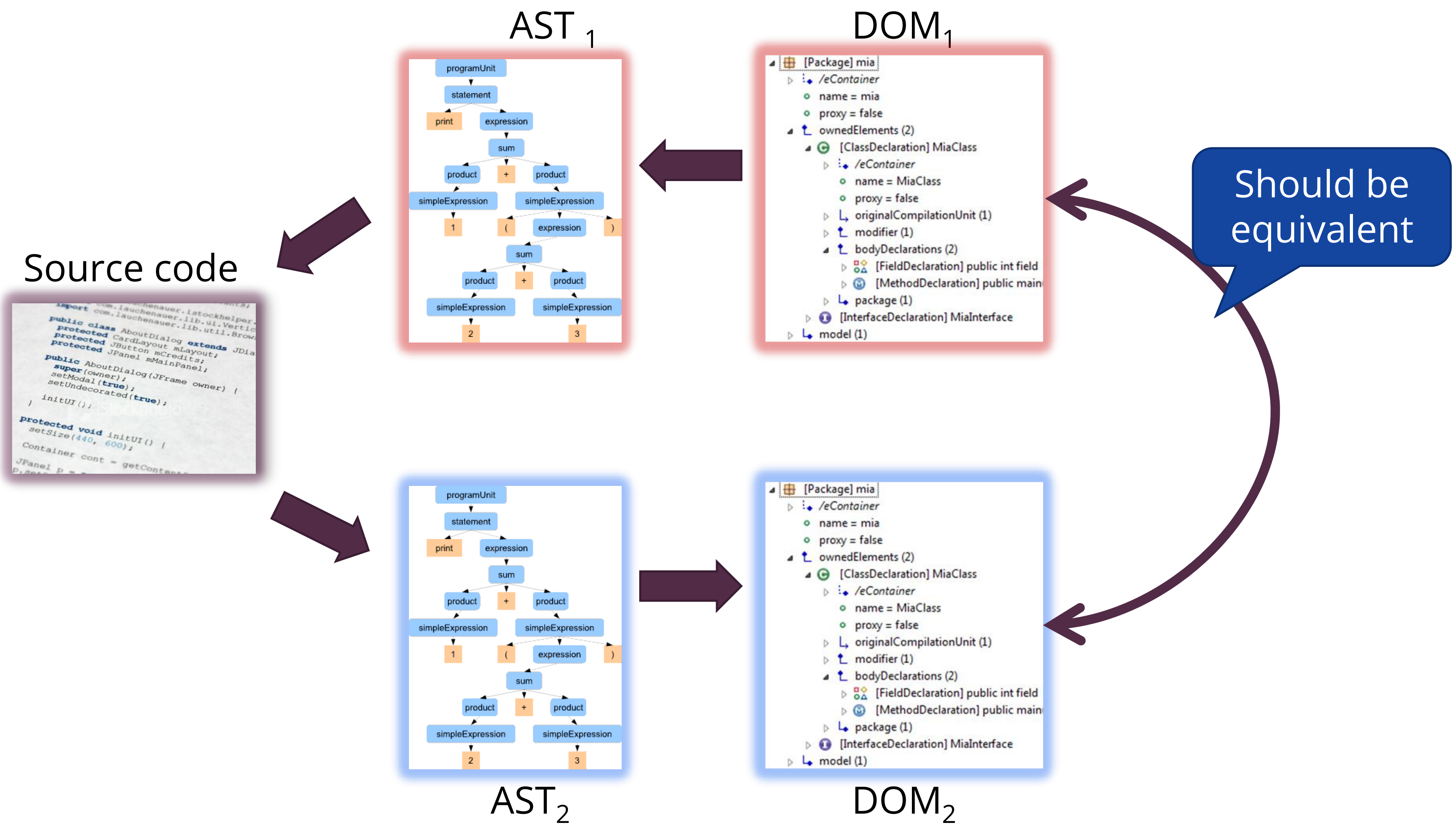
Call graph



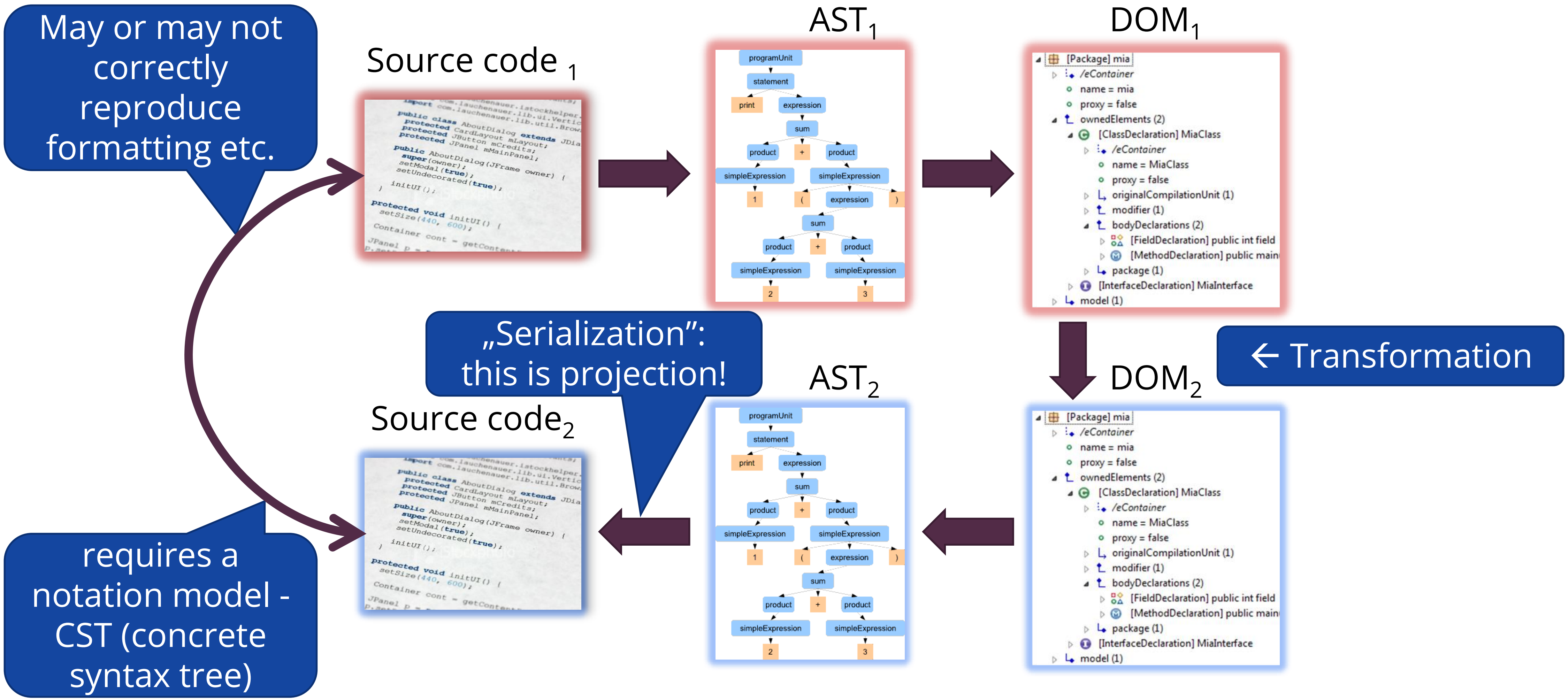
Type hierarchy



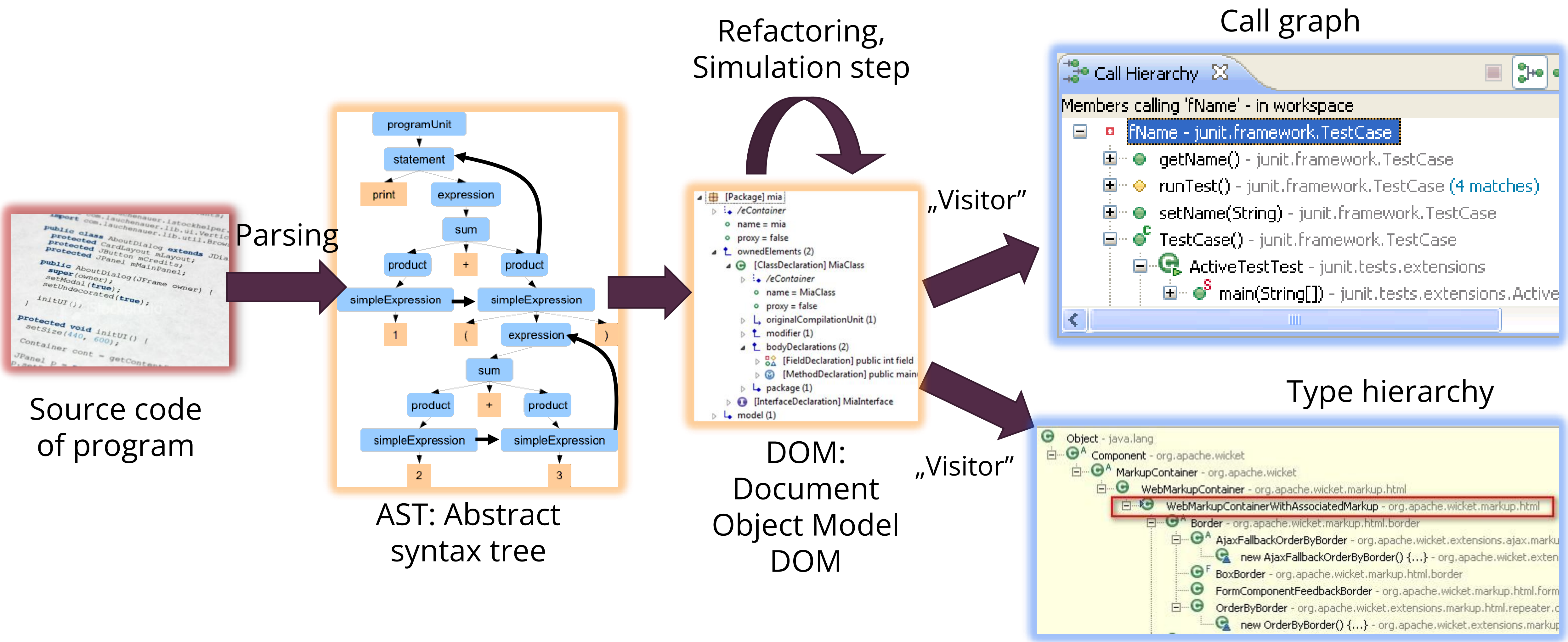
AST + DOMs: persistence roundtrip



Model editing roundtrip

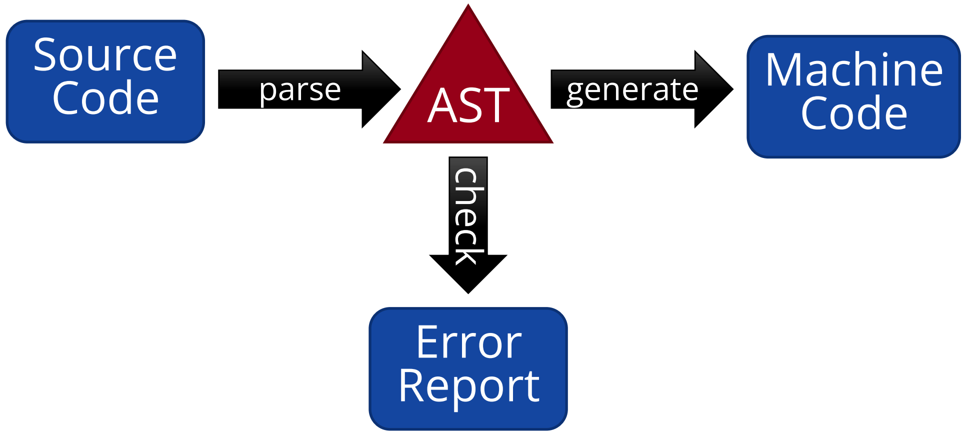


Textual DSM Languages: An Overview



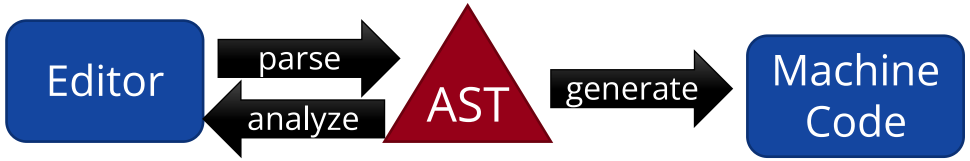
Traditional Architecture of Compilers

- On demand parsing
  - Explicit user's request
  - E.g. javac myClass.java
- Parsing:
  - Successful: AST generated
  - Failed: Errors reported (no AST)
- Semantic checks
  - Successful: Machine code generated
  - Failed: Errors reported



Modern IDEs

- Auto-parse-and-check
  - During typing: Parse
  - Upon save: Analyze
- Parsing:
  - AST always generated
  - Error markers on failure
- Semantic analysis
  - Successful:
    Machine code generated
  - Failed: Errors reported



Syntactic editor services:
- syntax checking
- syntax highlighting
- outline view
- code folding
- bracket matching…

Semantic editor services:
- error checking
- reference resolving
- hover help
- „code mining”
- code completion
- refactoring…

Source: Guido Wachsmuth (Compiler Construction at TU Delft)

Textual syntax specification

Regular grammars

Context-free grammars

Textual Domain-specific Languages

- Idea
  - Describing models as text files
- Textual development
  - Long history (30+ years)
  - Well-researched theory
  - Mature tools

(Bad) idea: Regular expressions

- Pattern matching for strings
  - Good support
    - Most programming languages
    - More or less the same syntax
  - Calculates and returns matches
- Usable as DSL parser?
  - Weak expressive power
    - Example: balanced parentheses (see pumping lemma)
  - Output is a single boolean variable (does it match?)
    - Missing: interpretability of contents, error localization
  - In some cases, not very concise…

RegExp: Validation of email addresses

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t] )+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?: \r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:( ?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0 31]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\ ](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+ (?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z |(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n) ?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\ r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n) ?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t] )*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])* )(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*) *:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+ |\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r \n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?: \r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t ]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031 ]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]( ?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(? :(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(? :\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(? :(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)? [ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]| \\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<> @,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t] )*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\ ".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
\t])*)(?:\.(?:(?:\r\n)?[ \t])*(? :[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[ \]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000- \031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|( ?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,; :\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\" .\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\ ]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\ [\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\ r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
\t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\] |\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \0 00-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\
.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@, ;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(? :[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[
\t])*))*@(?:(?:\r\n)?[ \t])* (?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\". \[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[ ^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\] ]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s*( ?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:( ?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[ \["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t
\t])*(?:[^()<>@,;:\\".
\t])+|\Z|(?=[\["()<>
()<>@,;:\\".\[\]]))|\[                                                                                          r\n)?[
\t])*))*(?:,@(?:(?:\r\                                                                                          ".\[\] \000-
\031]+(?:(?:(?:\r\n)?                                                                                           >@,;:\\".
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?: \r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[ "()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t]) *))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]) +|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\ .(?:(?:\r\n)?[
\t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z |(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:( ?:\r\n)?[ \t])*))*)?;\s*)
```

This regular expression will only validate addresses that have had any comments stripped and replaced with whitespace

Source: http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html

Regexp: validation

- Output is a single boolean variable
  - Decides whether the string matches the language
- What is missing?

Error localization!

Debugging!

(it has a bug with “00”)

Context-Free (CF) grammars

Wide-spread adoption

- Sufficient expressive power for describing models
- Intuitive semantics and usage
- Advanced parsing techniques
  - Performance
  - Error localization
  - …
- Automated tooling for parser generation

Example: name lists CF grammar

- Terminal Symbols
  - "*Dániel*", "*István*", "*Zoltán*", "*and*", ","
- Non-terminal Symbols
  - «Name», «Sentence», «List»
- Metalanguage

  `::=,   |,   « »`

```
«Sentence»::= «Name» | «List» and «Name»
«List»::= «List», «Name»| «Name»
«Name»::=  Zoltán | István | Dániel
```

Examples: *Dániel* *Dániel, Zoltán* *Dániel, Zoltán and Dániel*

Example from: Dick Grune, Ceriel J.H. Jacobs: Parsing Techniques

The Parsing Process

Derivation trees

Lexer vs parser

Linking & scoping

Example: derivation / parse tree

- Terminal Symbols
  - "*Dániel*", "*István*", "*Zoltán*", "*and*", ","
- Non-terminal Symbols
  - «Name», «Sentence», «List»
- Metalanguage

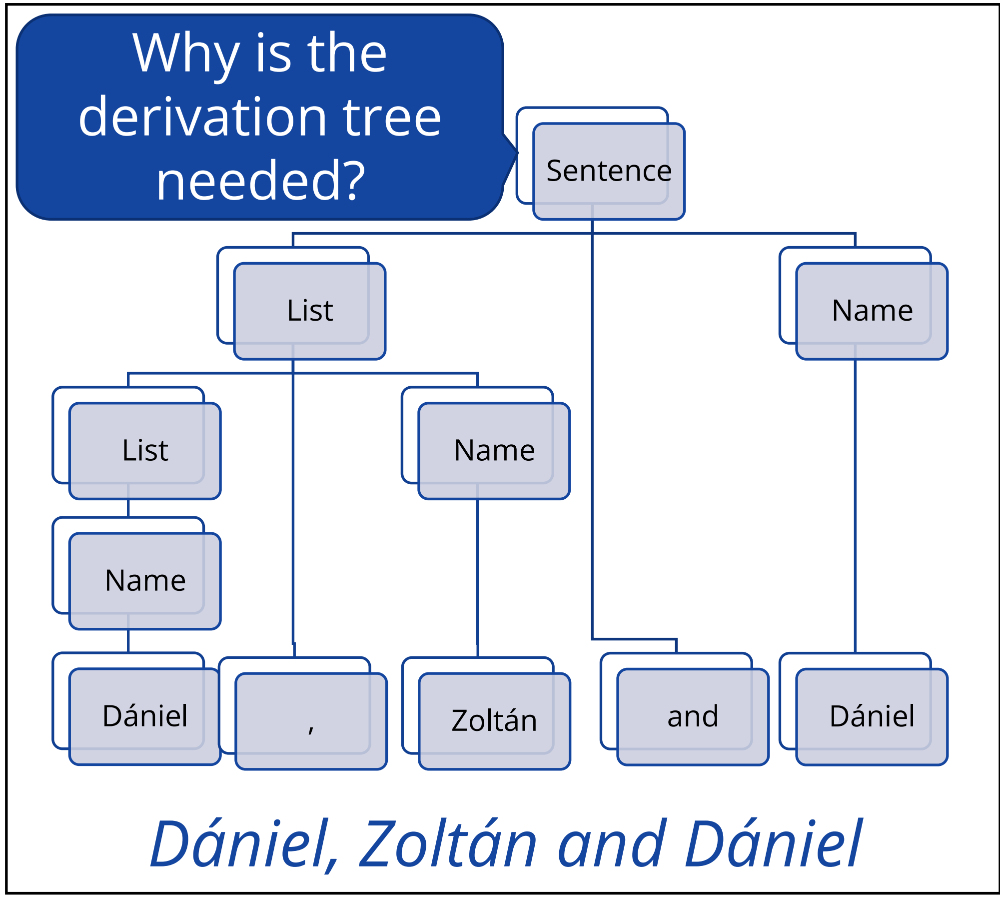
  ::=, |, « »

«Sentence»::= «Name» | «List» *and* «Name»

«List»::= «List», «Name»| «Name»

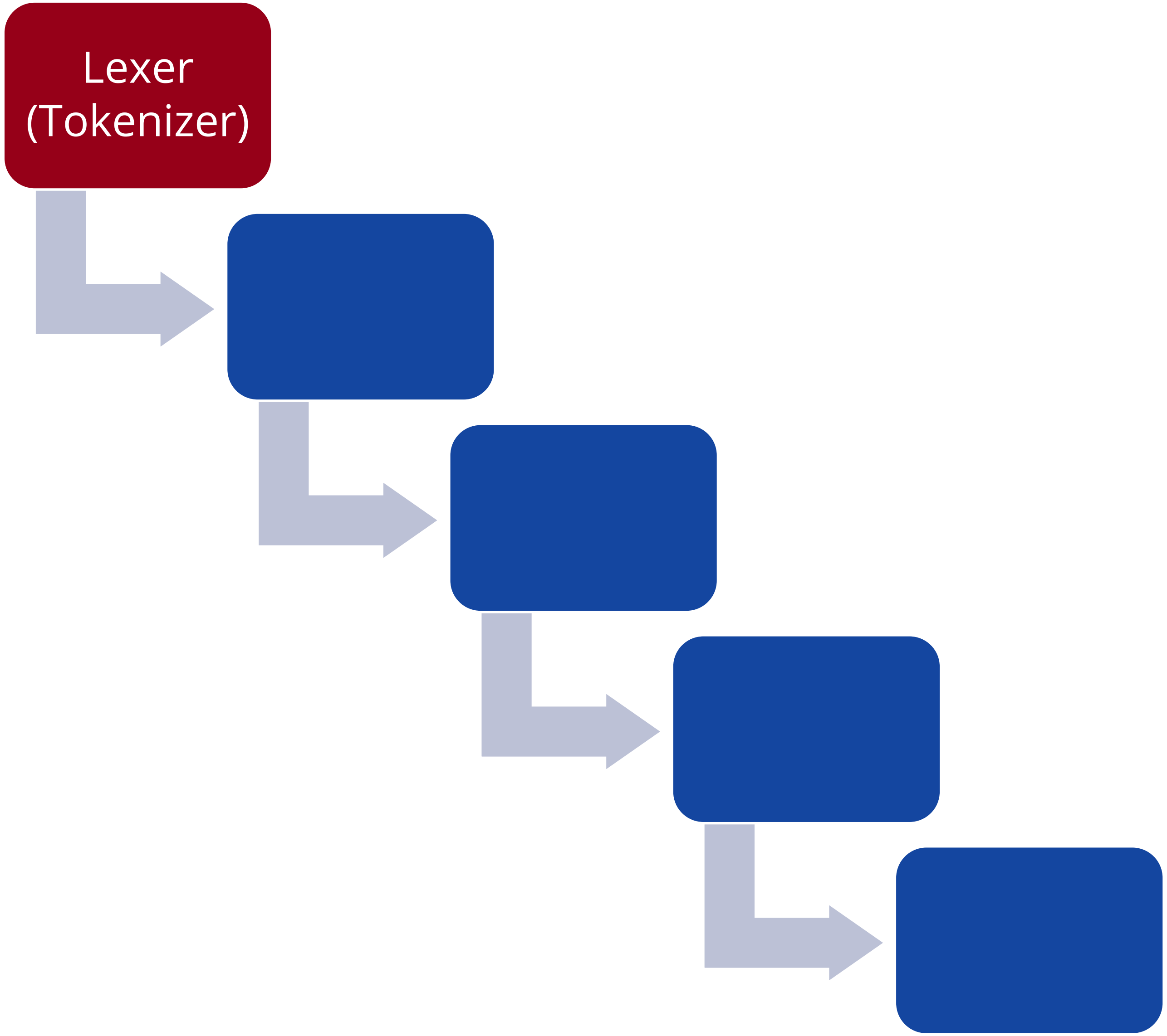
«Name»::= *Zoltán* | *István* | *Dániel*

Examples: *Dániel* *Dániel, Zoltán*



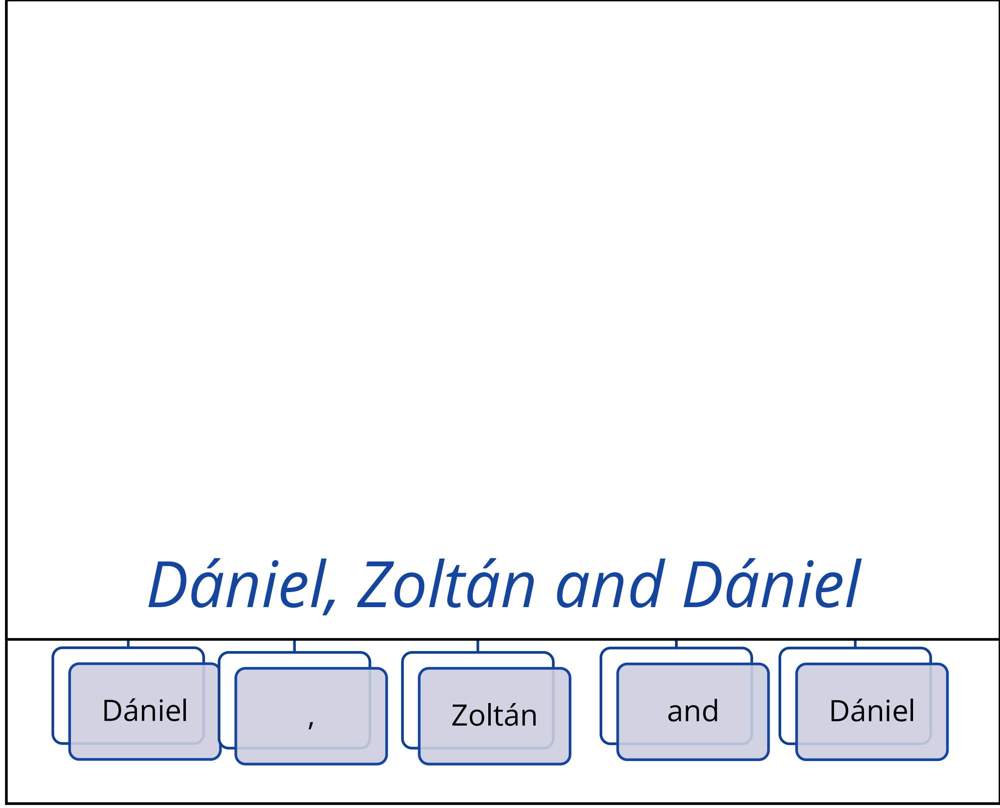
Example from: Dick Grune, Ceriel J.H. Jacobs: Parsing Techniques

The parsing process



Lexer (Tokenizer)

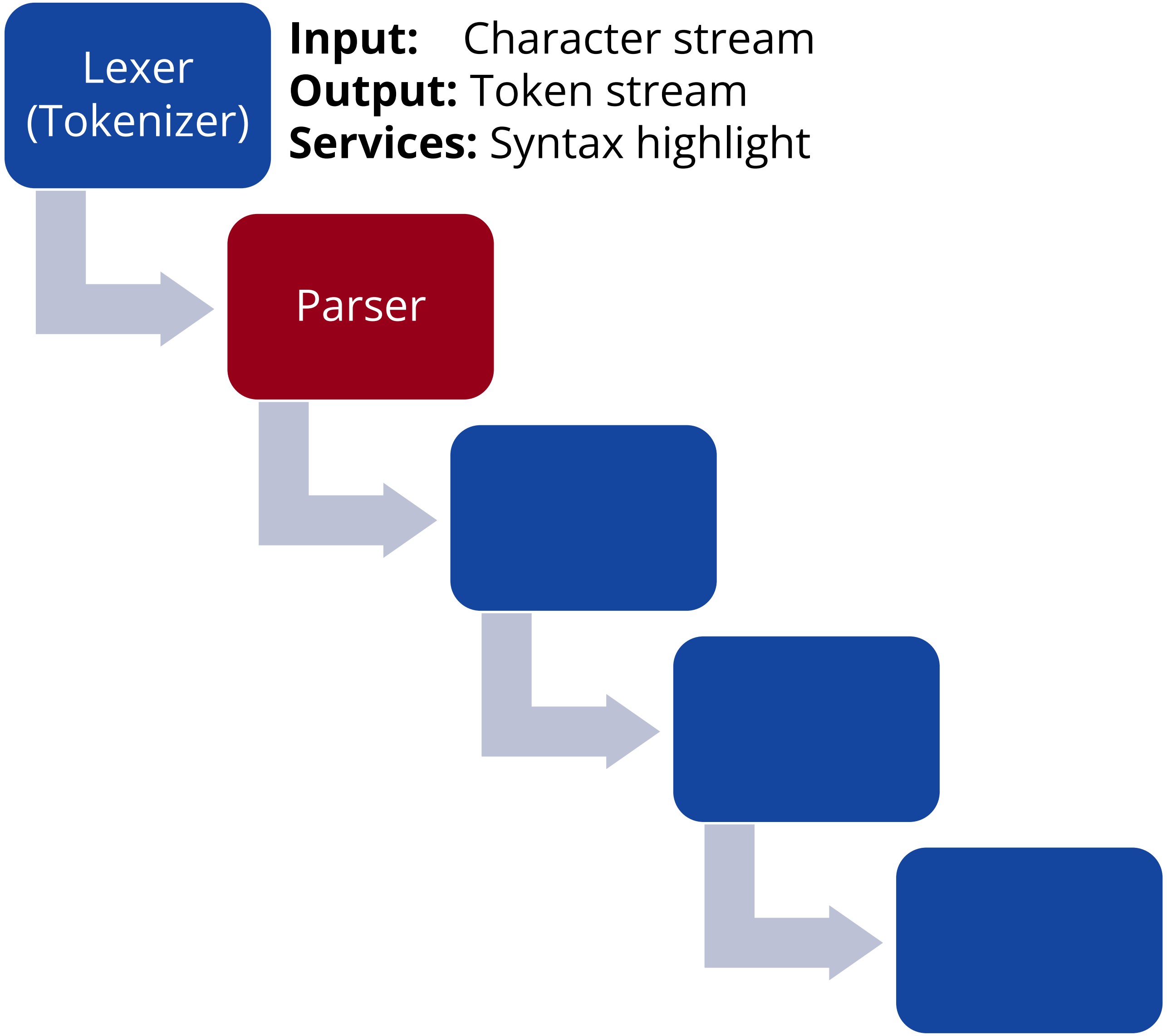
- **Input:** Character stream
- **Output:** Token stream



- Tokenizing the input character stream
- Similar to the parsing problem
  - But usually simpler – Typically regular expressions
  - Only word/token identification
  - Optional task: leaving out comments

Error handling, Performance, Problem decomposition

The parsing process



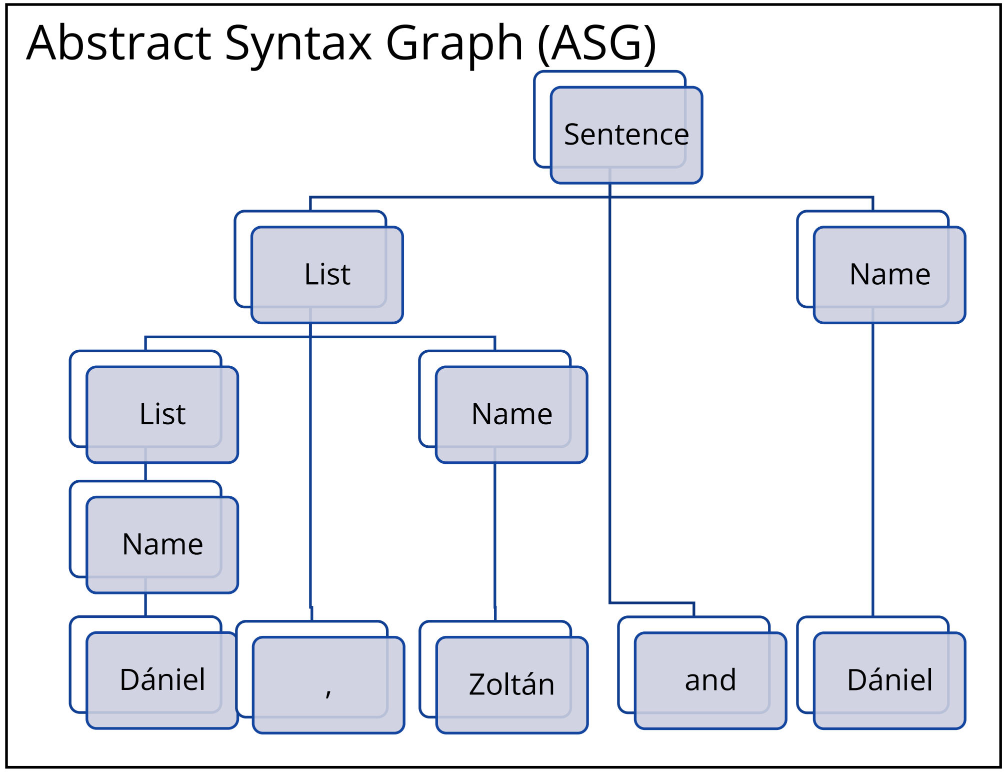
Input: Character stream
Output: Token stream
Services: Syntax highlight

Parser

Grammar:

«Sentence»::= «Name» | «List» *and* «Name»
«List»::= «List», «Name»| «Name»
«Name»::= *Zoltán* | *István* | *Dániel*

Token stream:



The parsing process

Lexer (Tokenizer)

Input: Character stream
Output: Token stream
Services: Syntax highlight

Parser

Input: Token stream
Output: AST (well, CST/parse tree first, ignore that for now…)
Services: Outline, formatting, partial semantic search

Resolver (Linker)

Input: AST
Output: ASG (abstract syntax graph) = AST + crosslinks
Services: Semantic search, navigation

Variable / Identifier Handling

- Variables
  - At runtime: Value calculation/substitution
  - Editing/analysis time: Refers to other parts of the AST
- ***Variable declaration***:
  - A declaration of a variable
  - Unique naming: Variable definitions must be resolvable → Extra phase
- ***Variable reference***: Use of an already defined variable

```
int a=3;                  Declaration
System.out.println(a);    Rerference
```

- Parser checks: "Can a variable be named 'a'?"
- Reference resolution: "Is a variable defined?" → Scoping problem

Scoping/Linking problem

```
private int value;

public void setValue(int value)
{
        this.value = value;
}
```

Which variable declaration is referred by 'value'?

- Possible approaches
  - Most specific declaration
    - **Hierarchical scopes**
  - Conflict is error
  - Qualified references
  - …
- Linking may be *lazy*

Scope: the set of elements (variable names, identifiers) that can be referenced at a given point. Used:

- During parsing (resolving/linking),
- During editing (content assist)

The parsing process

Lexer (Tokenizer)

Input: Character stream
Output: Token stream
Services: Syntax highlight

Parser

Input: Token stream
Output: AST (well, CST/parse tree first, ignore that for now...)
Services: Outline, formatting, partial semantic search

Resolver (Linker)

Input: AST
Output: ASG (abstract syntax graph) = AST + crosslinks
Services: Semantic search, navigation

Value Converter

Input: ASG
Output: DOM (document object model),
- Conform to metamodel, E.g. use Integer in place of „1976”

Services: Refactor, code generation

Optional;
May come before *Linker*

The parsing process

Lexer (Tokenizer)

Input: Character stream
Output: Token stream
Services: Syntax highlight

Parser

Input: Token stream
Output: AST (well, CST/parse tree first, ignore that for now…)
Services: Outline, formatting, partial semantic search

Resolver (Linker)

Input: AST
Output: ASG (abstract syntax graph) = AST + crosslinks
Services: Semantic search, navigation

Value Converter

Input: ASG
Output: DOM (document object model),

- Conform to metamodel, E.g. use Integer in place of „1976”

Services: Refactor, code generation

Analysis

Input: DOM
Arbitrary analysis techniques, E.g., type checking
Services: Validation, optimization…