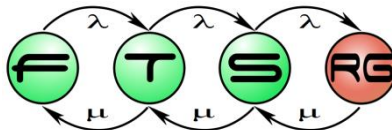


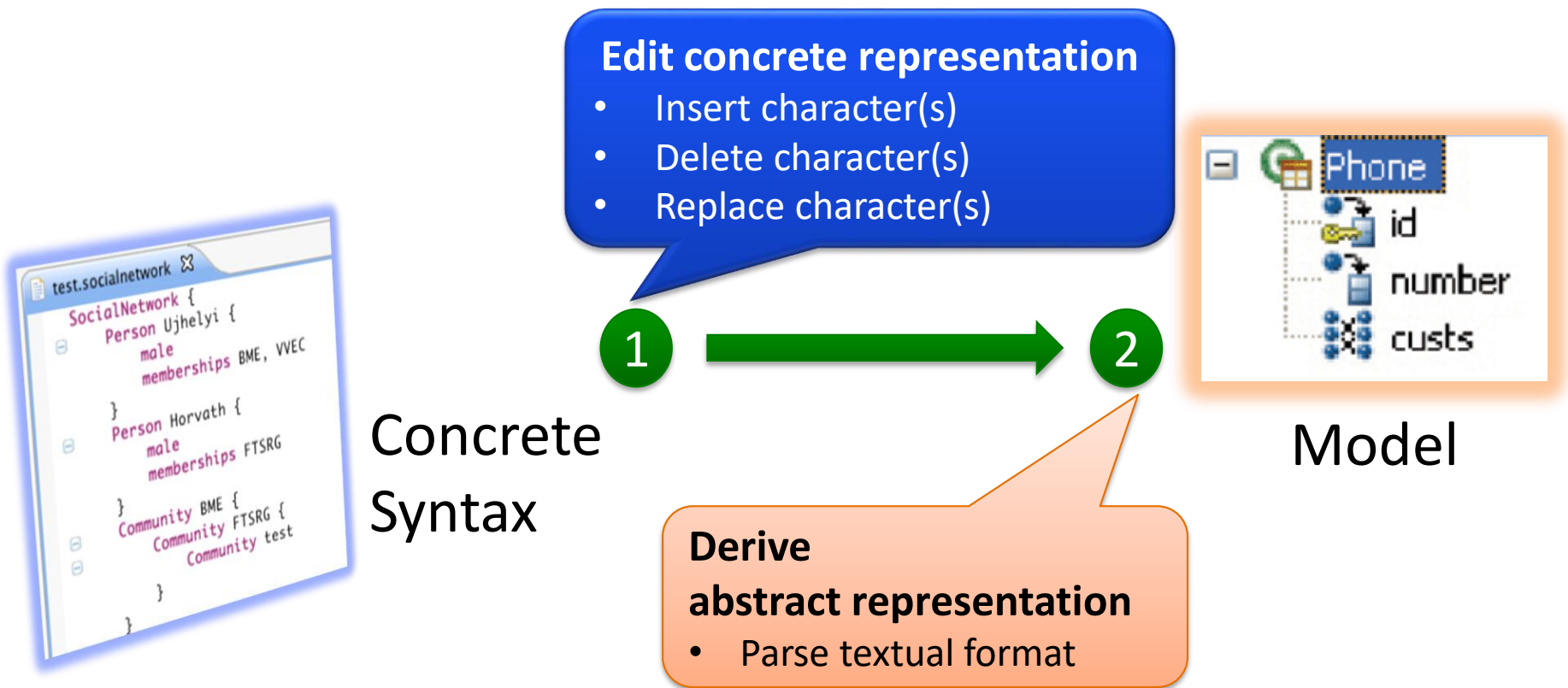
Textual Modeling Languages

Model Driven Software Development Lecture 6



Recall: raw editing

- Workflow 2: **raw editing** (w. textual syntax)
 - AKA source editing

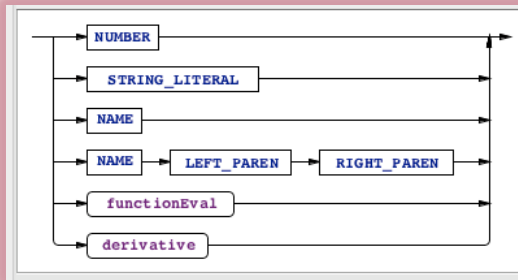


Looking inside advanced IDEs

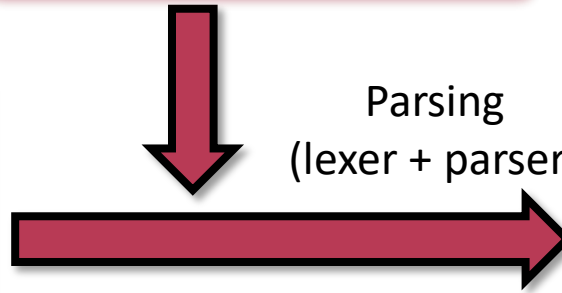
From parsers to development tools
Roundtrip property
Modern services

Parsers: The Traditional Setup

Grammar

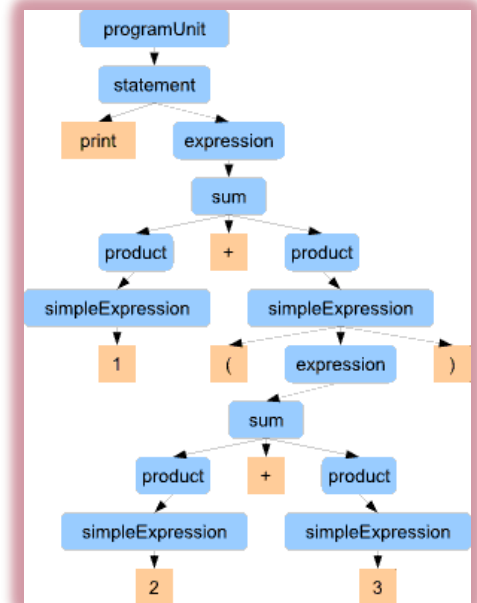


Parsing
(lexer + parser)



```
import com.lauchenauer.istockhelper.  
import com.lauchenauer.lib.ui.Vertic  
import com.lauchenauer.lib.util.Brow  
  
public class AboutDialog extends JDia  
protected CardLayout mLayout;  
protected JButton mCredits;  
protected JPanel mMainPanel;  
  
public AboutDialog(JFrame owner) {  
    super(owner);  
    setModal(true);  
    setUndecorated(true);  
    initUI();  
}  
  
protected void initUI() {  
    setSize(440, 600);  
    Container cont = getContentPane;  
    JPanel p = ...  
}
```

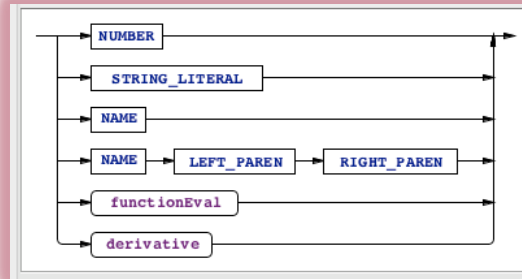
Source code of program



Abstract
syntax tree (AST)

Parsers in Software Engineering Practice

Grammar

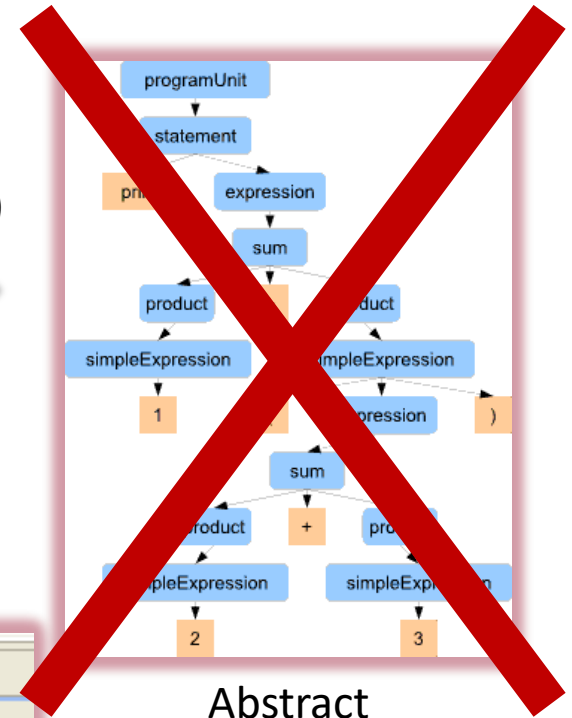
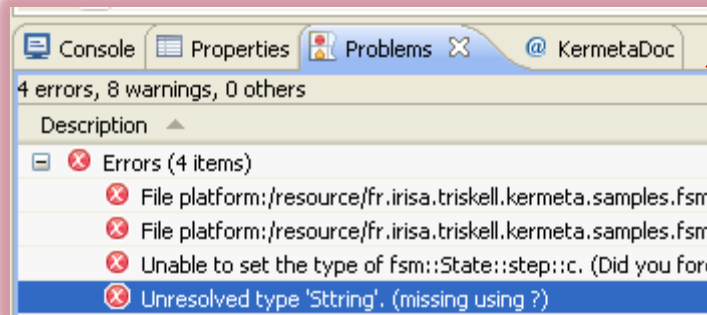


Parsing
(lexer + parser)

Error report

```
import com.lauchenauer.istockhelper.  
import com.lauchenauer.lib.ui.Vertic  
import com.lauchenauer.lib.util.Brow  
  
public class AboutDialog extends JDia  
protected CardLayout mLayout;  
protected JButton mCredits;  
protected JPanel mMainPanel;  
  
public AboutDialog(JFrame owner) {  
    super(owner);  
    setModal(true);  
    setUndecorated(true);  
    initUI();  
}  
  
protected void initUI() {  
    setSize(440, 600);  
    Container cont = getContentPane  
    JPanel p =
```

Source code of program



Abstract
syntax tree (AST)

Error recovery parsing

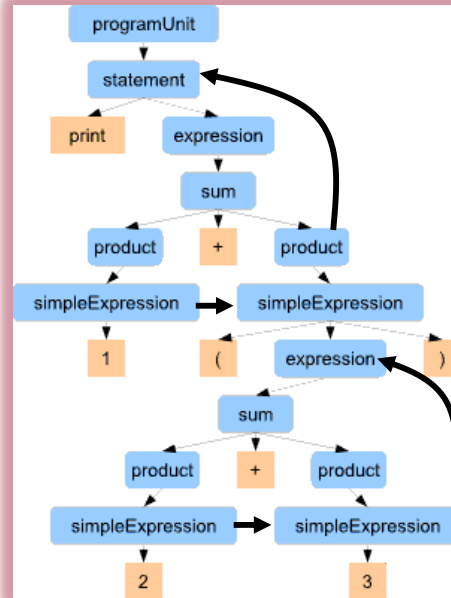
View Generation + Program Analysis

Call graph

```
import com.lauchnauer.iStockHelper.  
public class AboutDialog extends JDialog  
protected CardLayout mLayout;  
protected JButton mCredits;  
protected JPanel mMainPanel;  
public AboutDialog(JFrame owner) {  
    super(owner);  
    setModal(true);  
    setUndecorated(true);  
    initUI();  
}  
protected void initUI() {  
    setSize(440, 600);  
    Container cont = getContentPane();  
    JPanel p = ...  
}
```

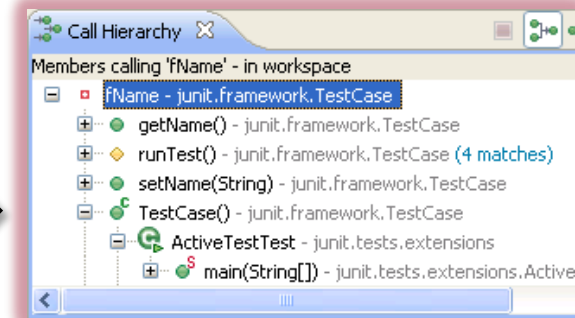
Source code of
program

Parsing



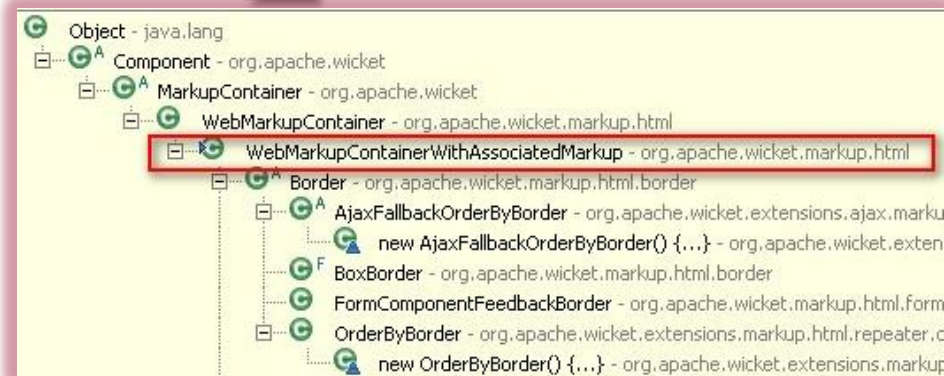
AST: Abstract
syntax tree

„Visitor”



„Visitor”

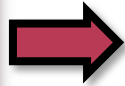
Type hierarchy



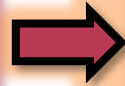
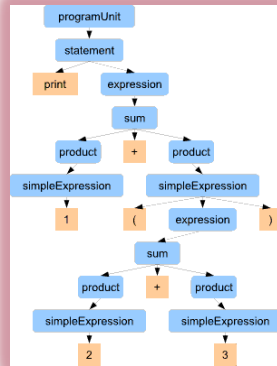
ASTs vs DOMs

Source code of program

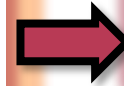
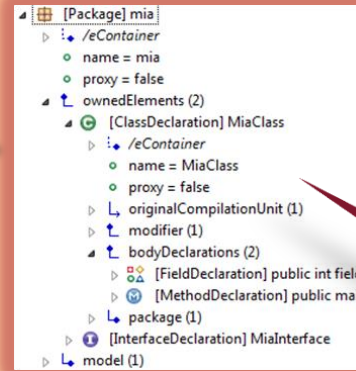
```
import com.lauchhammer.istockhelper.  
public class AboutDialog extends JDialog  
protected CardLayout mLayout;  
protected JButton mCredits;  
public AboutDialog(JFrame owner) {  
    super(owner);  
    setModal(true);  
    setUndecorated(true);  
    initUI();  
}  
protected void initUI() {  
    setSize(440, 600);  
    Container cont = getContentPane();  
    JPanel p = ...  
}
```



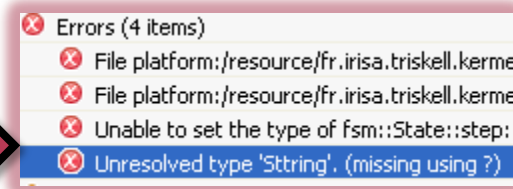
AST: Abstract syntax tree



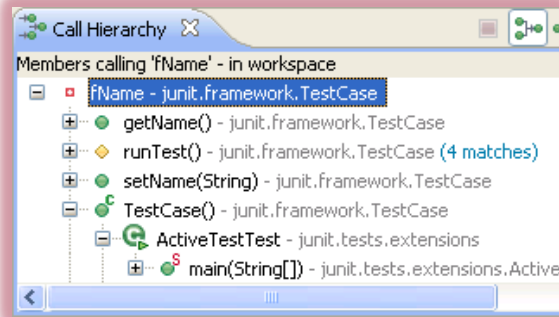
DOM: Document Object Model



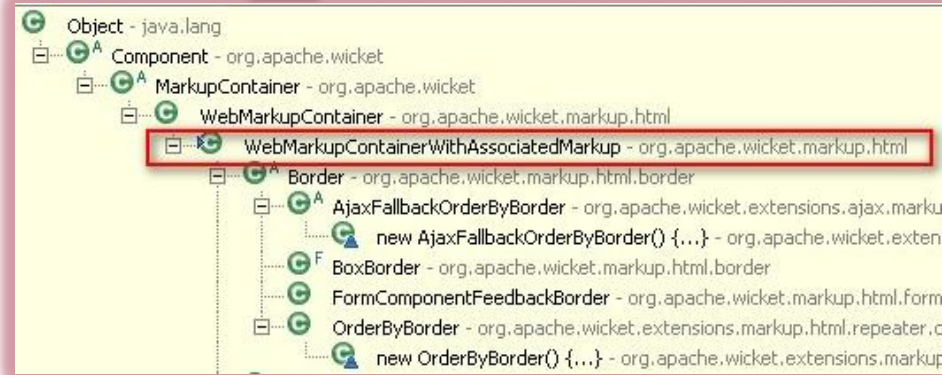
Well-formedness constraints



Defined by a metamodel

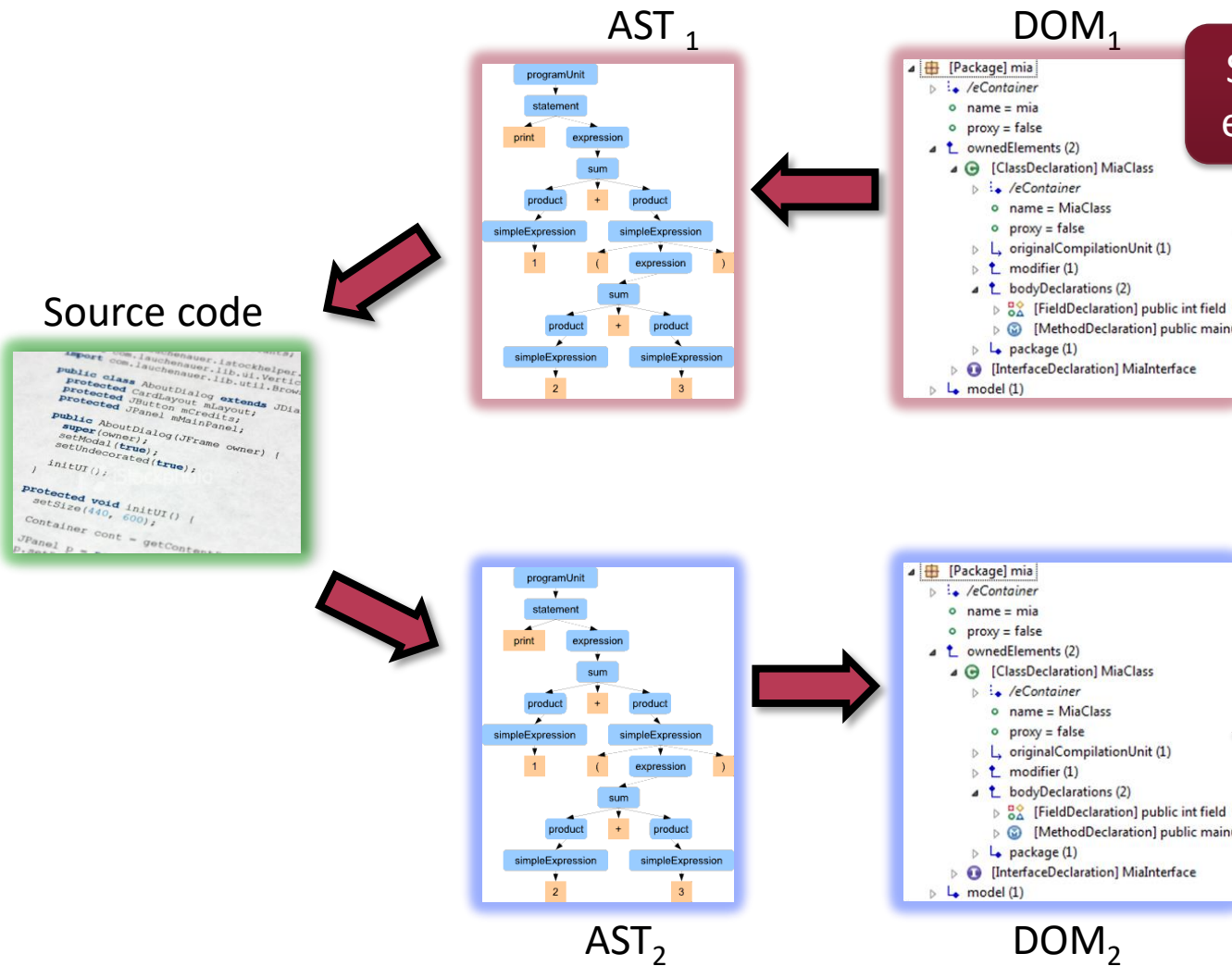


Call graph (View)

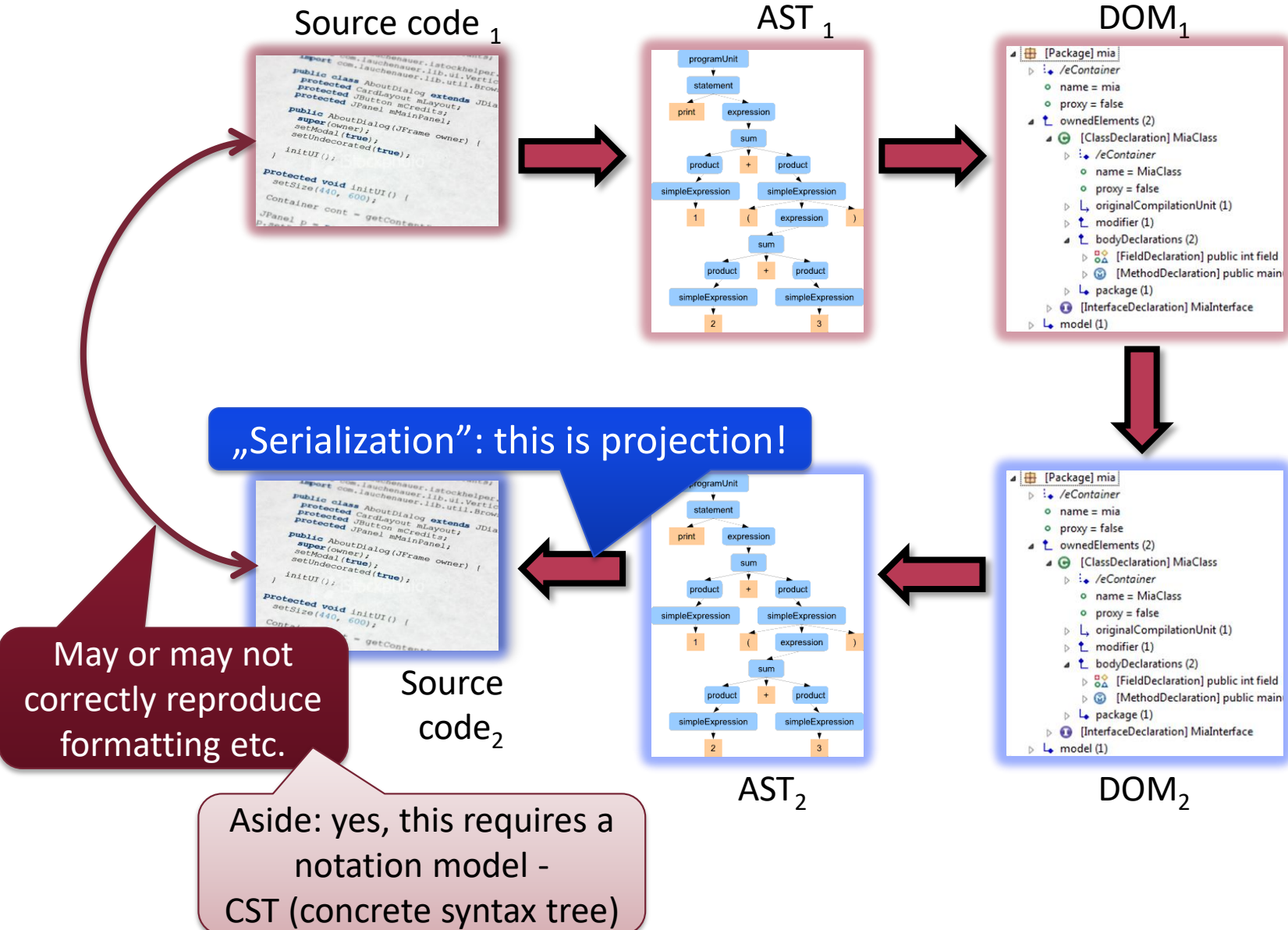


Type hierarchy (View)

Persistance Roundtrip



Model Editing Roundtrip (e.g refactoring)



Textual DSM Languages: An Overview

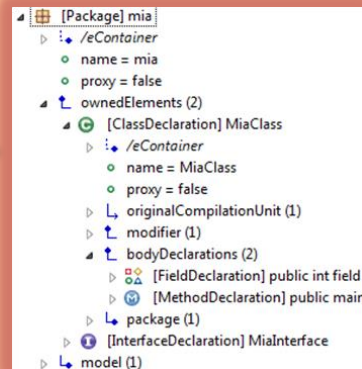
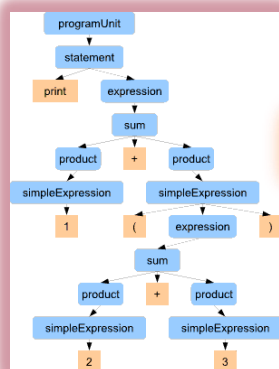
Source code

AST

DOM /
Abstract syntax

Well-formedness
constraints

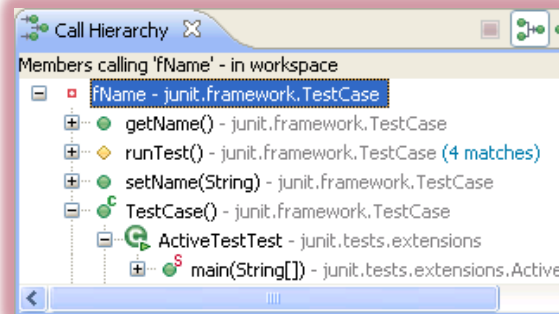
```
import com.lauchhammer.istockhelper.  
public class AboutDialog extends JDialog  
protected CardLayout mLayout;  
protected JButton mCredits;  
public AboutDialog(JFrame owner) {  
    super(owner);  
    setModal(true);  
    setUndecorated(true);  
    initUI();  
}  
protected void initUI() {  
    setSize(440, 600);  
    Container cont = getContentPane();  
    JPanel p = ...  
}
```



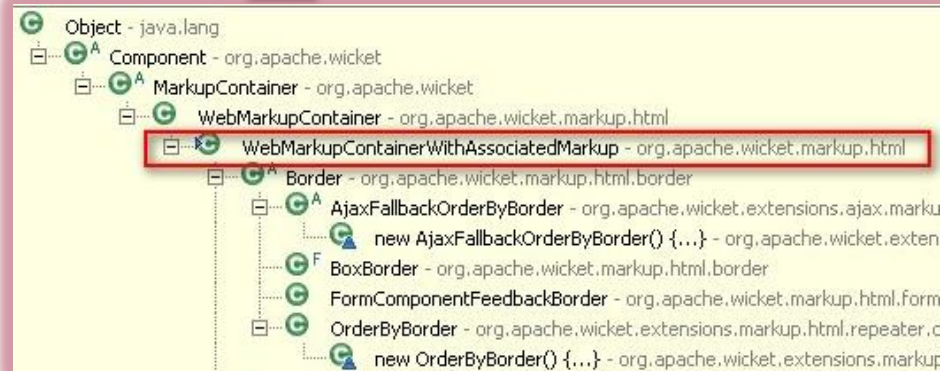
Errors (4 items)

- File platform:/resource/fr.irisa.triskell.kerne
- File platform:/resource/fr.irisa.triskell.kerne
- Unable to set the type of fsm::State::step:
- Unresolved type 'Sttring'. (missing using ?)

Refactoring,
Simulation step



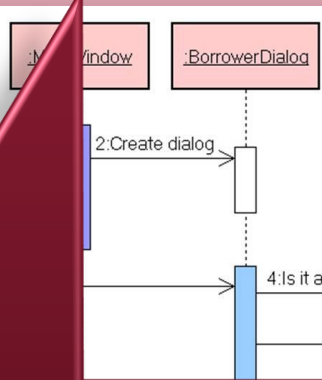
Call graph
(Analysis model / View)



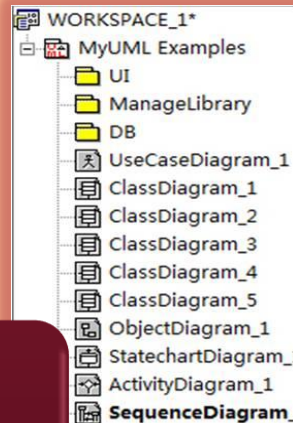
Type hierarchy
(Analysis model / View)

Graphical DSM Languages

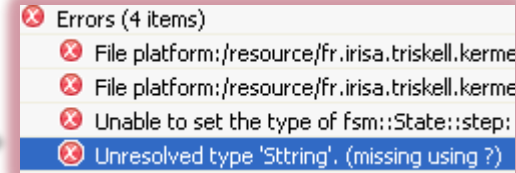
Diagram model



Abstract syntax

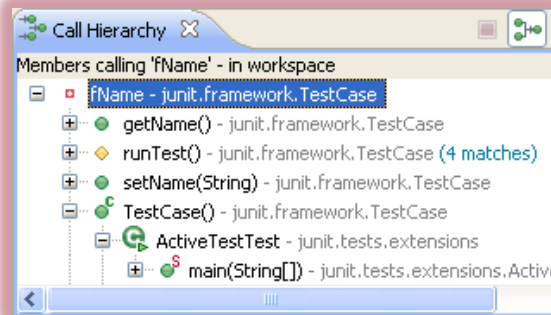


Well-formedness constraints

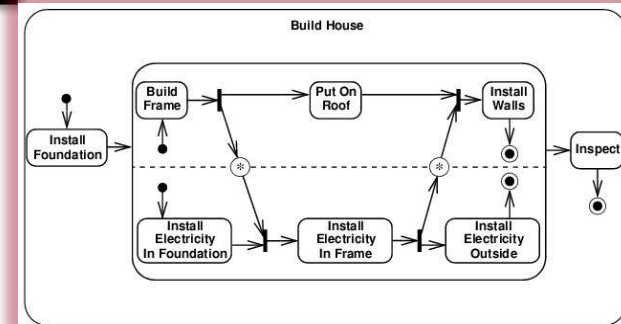


Refactoring,
Simulation

This is the notation model
The diagram image itself is not parsed!



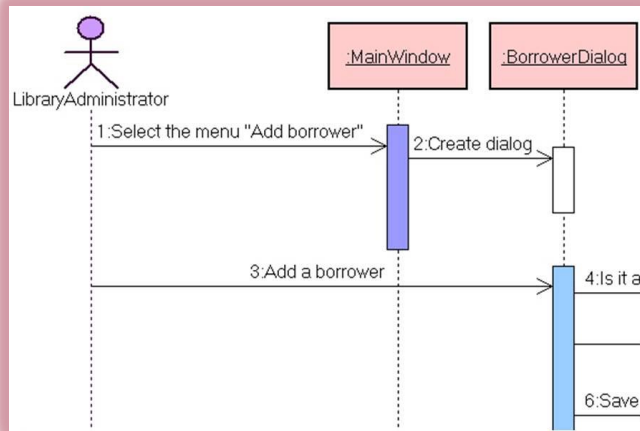
Call graph (View)



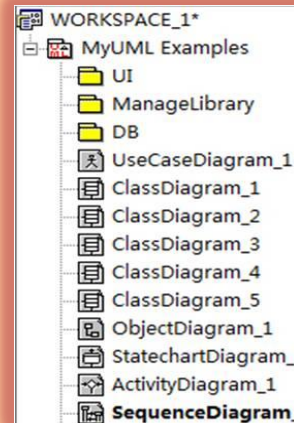
Statechart
(other DSM)

Graphical DSM Languages

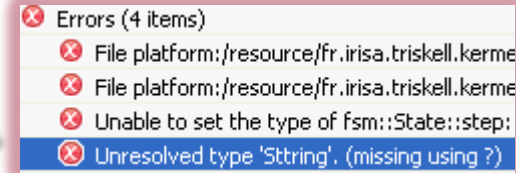
Diagram image



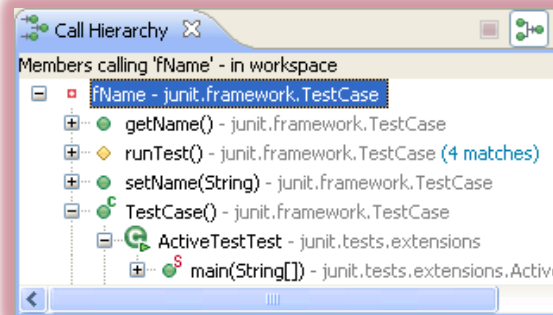
Abstract syntax
incl. notation



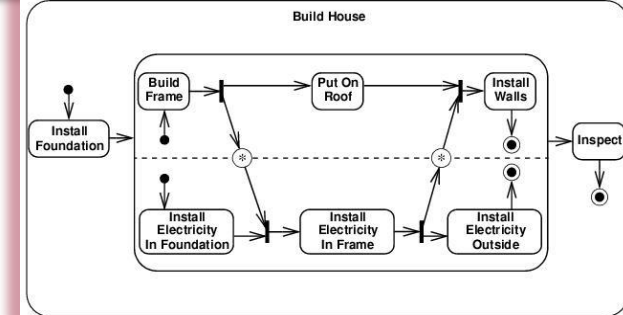
Well-formedness
constraints



Refactoring,
Simulation

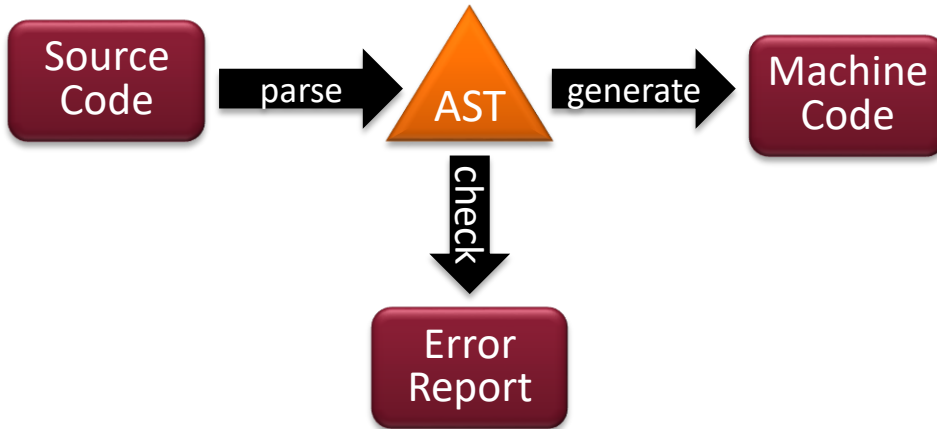


Call graph (View)



Statechart
(other DSM)

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 Compilers in IDEs



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...

- 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

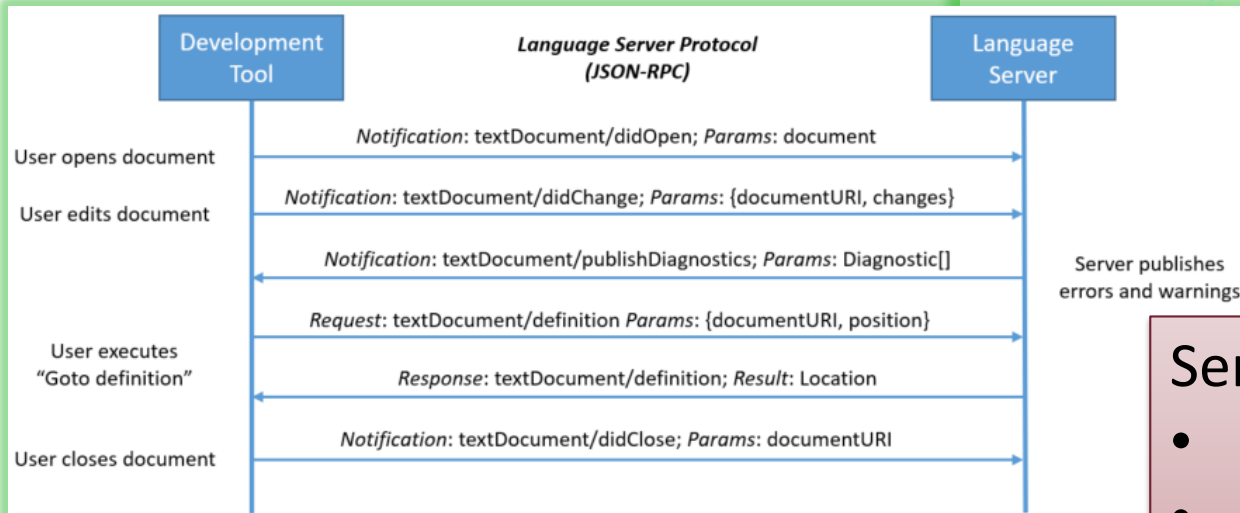
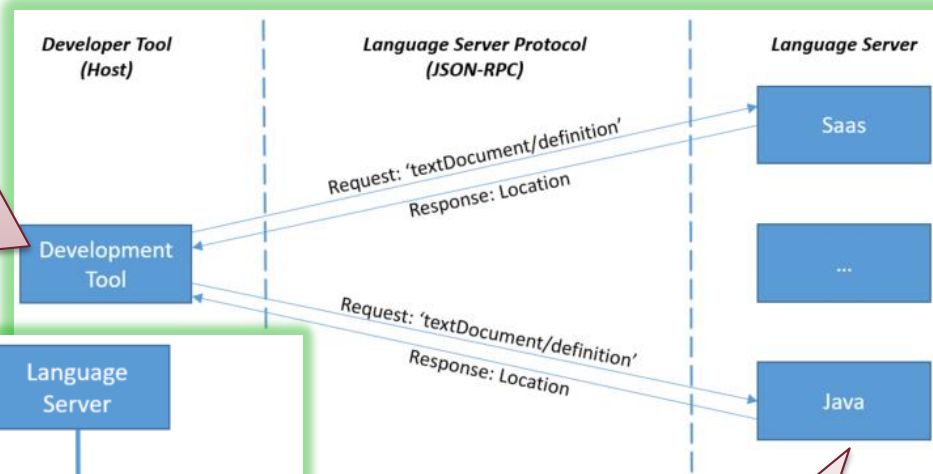
Source: Guido Wachsmuth (Compiler Construction at TU Delft)

„Hot topic”: Language Server Protocol (LSP)

- Language-agnostic editor SW + language server
 - Delegate some editor services to server
 - Completion, diagnostics, hover, folding, rename, navigation...



VS Code, Atom, Eclipse, Theia, Che, Sublime Text, Monaco JS, ...



Servers available in...

- Many languages
- Several technologies

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

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

[illegible]

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

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: CF grammar for describing name lists

- Terminal Symbols

- “*Dániel*”, “*István*”, “*Zoltán*”, “*and*”, “,”

- Non-terminal Symbols

- «Name», «Sentence», «List»

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

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

«List» ::= «List», «Name» | «Name»

Expressive power is not everything



```
typeDeclaration class {  
  name: "HelloWorld"  
  public  
  memberDeclaration: method {  
    name: "main"  
    public static  
    returns: builtinTypeRef void  
    parameterDeclaration: {  
      name: "args"  
      type: arrayType {  
        baseType: classRef "java.lang.String"  
      }  
    }  
  }  
  body {  
    statement: instanceMethodInvocation {  
      methodName: "println"  
      argument: stringLiteral "Hello, World"  
      receiver: staticFieldAccessExpression {  
        fieldName: "out"  
        receiver: classRef "java.lang.System"  
      }  
    }  
  }  
}
```

```
public class HelloWorld {  
  public static void main(String[] args) {  
    System.out.println("Hello, World");  
  }  
}
```

Human aspects in grammar design:

- Conciseness
- Readability
- Intuitive meaning
- ...

~ Default syntax
generated from
metamodel

The Parsing Process

Derivation trees

Lexer vs parser

Linking & scoping

Example: CF grammar for describing name lists

- Terminal Symbols

- “*Dániel*”, “*István*”, “*Zoltán*”, “*and*”, “,”

- Non-terminal Symbols

- «Name», «Sentence», «List»

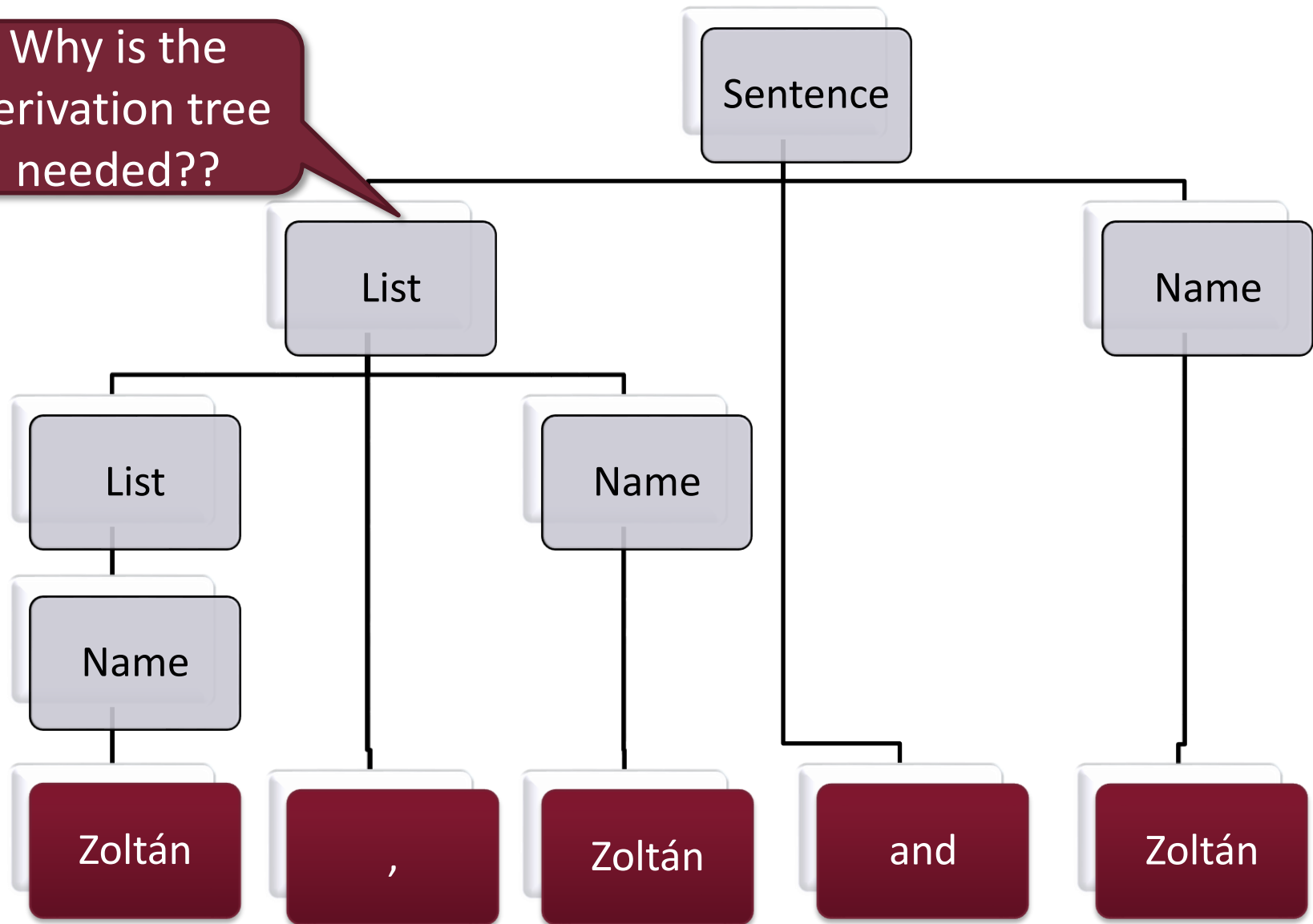
«Name» ::= *Zoltán* | *István* | *Dániel*

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

«List» ::= «List», «Name» | «Name»

Derivation / Parse Tree

Why is the
derivation tree
needed??



Additional practical tasks

- Handling input strings
- Variable handling
- High level analysis

Input

Input string:

'Z' 'o' 'l' 't' 'á' 'n' ' ' ' , ' 'Z'
'o' 'l' 't' 'á' 'n' ' ' 'a' 'n' 'd'
' ' 'Z' 'o' 'l' 't' 'á' 'n'

Zoltán

,

Zoltán

and

Zoltán

Input handling

Input:

Character stream

Parser
input:

Higher level tokens

- «Name»
- ‘;’
- “and”

Input gap

Filled by ‘lexer’

- Why is this indirection useful?
 - Error handling
 - Performance
 - Problem decomposition

Lexer

- Goal:
 - 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
 - Simplifies parsing significantly

Variable / Identifier Handling

■ Variables

- At runtime:
Value calculation/substitution
- Editing/analysis time:
Refers to other parts of the AST

■ *Variable definition*

- A declaration of a variable
- Unique naming
 - Variable definitions must be resolvable
 - Extra phase required after parsing

■ *Variable reference*

- Use of an already defined variable

```
int a=3;
```

```
System.out.println(a);
```

■ Parser checks

- “Can a variable be named ‘a’?”

■ Reference resolution

- “Is a variable defined?”
 - Scoping problem
- “Is a variable uniquely defined?”

Scoping Problem

```
private int value;
```

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

Which variable
declaration is referred
by 'value'?

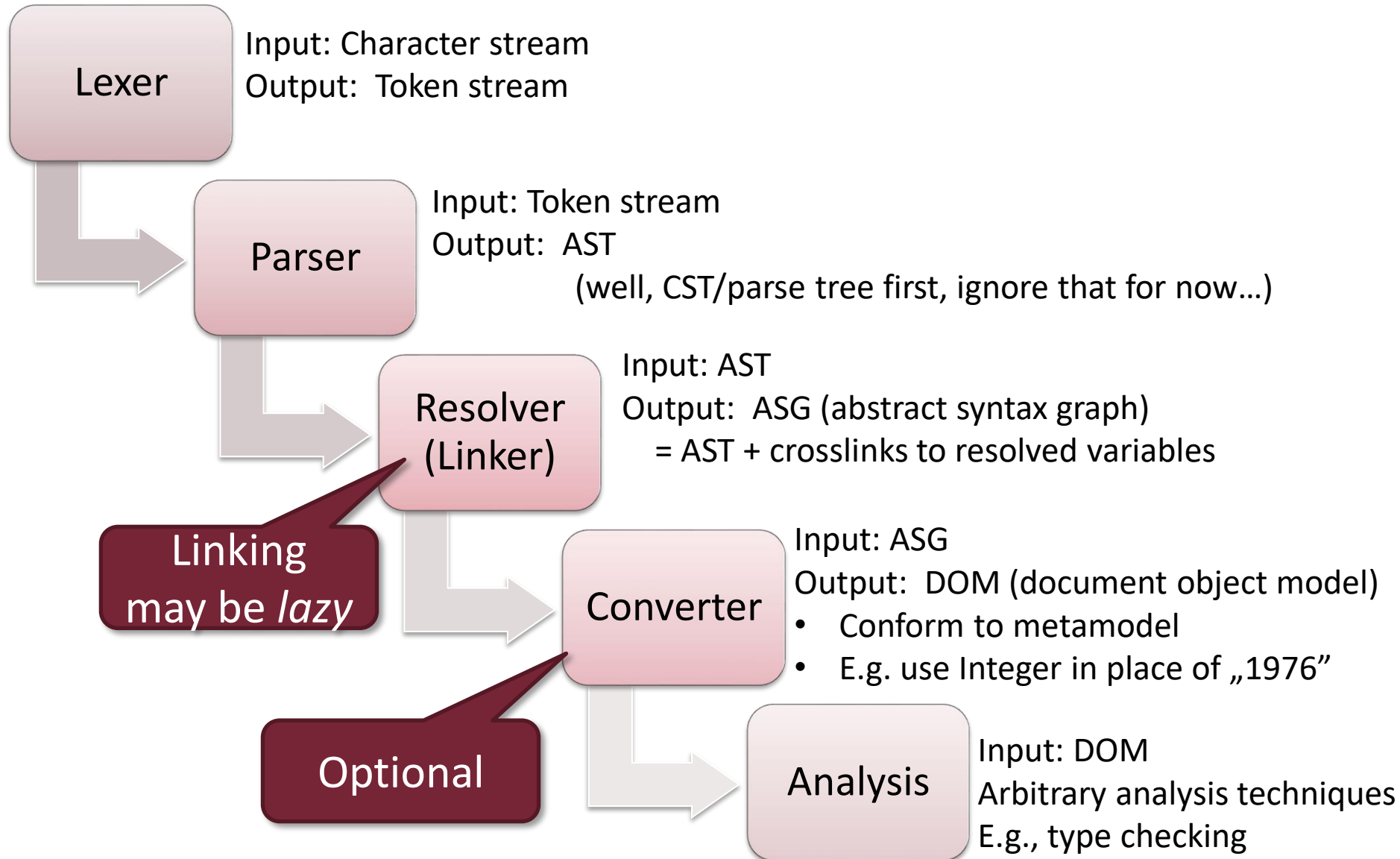
- Possible approaches
 - Most specific declaration
 - Hierarchical scopes
 - Conflict is error
 - Qualified references
 - ...

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



Technologies

for textual syntax editors in Eclipse

Technologies 1. – IMP

- IDE Meta-tooling platform
 - Goal:
 - Language editor creation
 - Every parser-generator should be re-usable
 - Manual coding required
 - Project is close to dead

Technologies 2. - EMFText

- Editor generation
 - Based on existing EMF model
 - Different generated grammar styles
 - Manually modifiable
 - Limited grammar support
- Syntax Zoo
 - ~100 different syntax examples available
 - Including a Java implementation (called JaMoPP)

Technologies 3. – Xtext

- Editor and EMF model generation
 - Based on high-level grammar
 - Optionally can be initiated from an existing EMF model
- Easy to work with but highly customizable
 - Both the grammar and the generated code
- New developments:
 - Xbase expression language – DSLs over JVM
 - JVM Model Inference instead of direct code generation
 - Interpreter also available
 - Can generate **LSP** language server (fewer features)

LSP4E Client (Generic Editor)

- Eclipse **Generic Text Editor** extensibility
 - Syntax highlight from *TextMate Grammar* (TM4E)
 - Editing services from an LSP *Language Server* (LSP4E)
 - Diagnostics
 - Completion
 - Doc hover
 - Navigation
 - Find references
 - Rename
 - Outline
 - ...



```
1 use std::env;
2
3 fn main() {
4     let args: Vec<String> = env::args().collect();
5
6     let query = &args[1];
7     let filename = &args[2];
8     fil
9
10    pr query);
11    pr me);
12 }
```

Grammars, CF Parsing

Theoretical Background

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

- $\text{Num} \rightarrow \text{Digit Num}$
- $\text{Num} \rightarrow \text{Digit}$
- $\text{Digit} \rightarrow 0 \mid 1 \mid 2 \mid \dots \mid 9$

■ Notation:

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

■ Regular rules:

- $B \rightarrow a$
- $B \rightarrow aC$

■ Context-free (CF) rules:

- $B \rightarrow \alpha$
- $B \rightarrow \varepsilon$

Empty symbol

Derivation and Language

- **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**
 - $L(G) = \{w \in T^* \mid \text{there exists a derivation } S \rightarrow^* w \text{ of } G\}$
 - Set of sentences derivable from S

Example derivation

Num	Num $\rightarrow$ Digit Num
Digit Num	Digit $\rightarrow$ 1
1 Num	Num $\rightarrow$ Digit Num
1 Digit Num	Digit $\rightarrow$ 9
1 9 Num	Num $\rightarrow$ Digit Num
1 9 Digit Num	Num $\rightarrow$ Digit
1 9 Digit Digit	Digit $\rightarrow$ 6
1 9 Digit 6	Digit $\rightarrow$ 7
1 9 7 6	

- **Remarks:**
 - In general, nonterminals can be resolved in arbitrary order (non-deterministic)
 - Leftmost vs. Rightmost derivation: always resolve the left/rightmost nonterminal as next step
- Parsing is polynomial algorithm for regular and context-free grammars

Binary Operations over Numbers

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

$\text{Exp} \rightarrow \text{Num}$

$\text{Exp} \rightarrow \text{Exp} \text{ "+" } \text{Exp}$

$\text{Exp} \rightarrow \text{Exp} \text{ "-" } \text{Exp}$

$\text{Exp} \rightarrow \text{Exp} \text{ "*" } \text{Exp}$

$\text{Exp} \rightarrow \text{Exp} \text{ "/" } \text{Exp}$

$\text{Exp} \rightarrow \text{"(" Exp "}"$

Two Derivations:

Exp

$\text{Exp} + \text{Exp}$

$1 + \text{Exp}$

$1 + \text{Exp} * \text{Exp}$

$1 + 2 * \text{Exp}$

$1 + 2 * 3$

$\text{Exp} \rightarrow \text{Exp} \text{ "+" } \text{Exp}$

$\text{Exp} \rightarrow \text{Num}$

$\text{Exp} \rightarrow \text{Exp} \text{ "*" } \text{Exp}$

$\text{Exp} \rightarrow \text{Num}$

$\text{Exp} \rightarrow \text{Num}$

Exp

$\text{Exp} * \text{Exp}$

$\text{Exp} * 3$

$\text{Exp} + \text{Exp} * 3$

$\text{Exp} + 2 * 3$

$1 + 2 * 3$

$\text{Exp} \rightarrow \text{Exp} \text{ "*" } \text{Exp}$

$\text{Exp} \rightarrow \text{Num}$

$\text{Exp} \rightarrow \text{Exp} \text{ "+" } \text{Exp}$

$\text{Exp} \rightarrow \text{Num}$

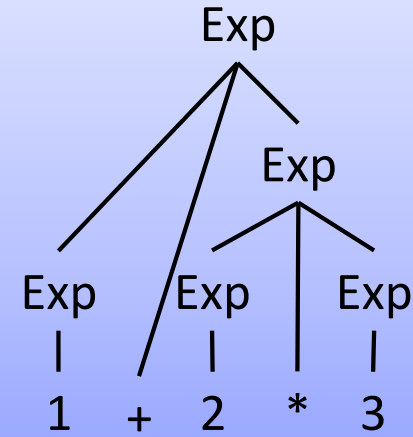
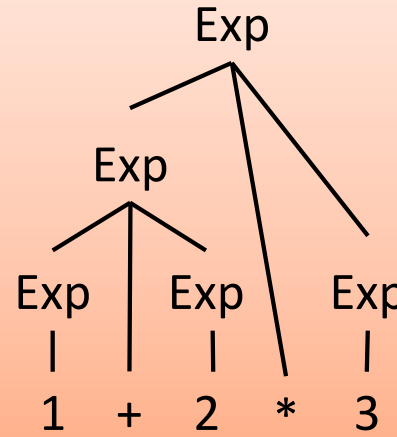
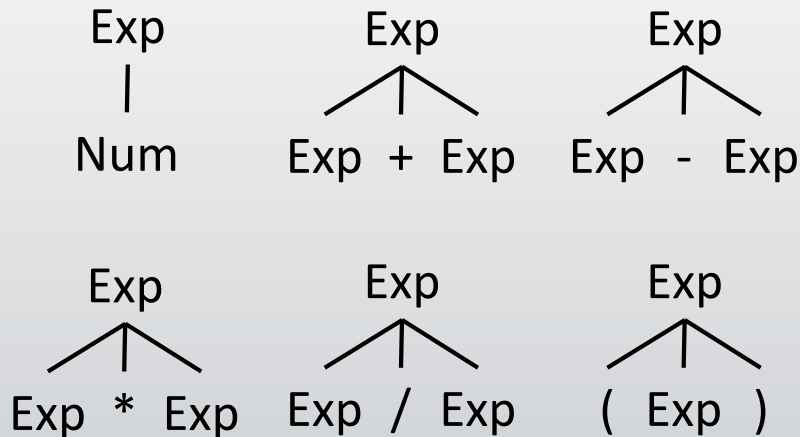
$\text{Exp} \rightarrow \text{Num}$

Problem:

$\text{Exp} * 3$

$1 + 2 * 3$

Parse Tree Construction



■ Parse Tree:

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

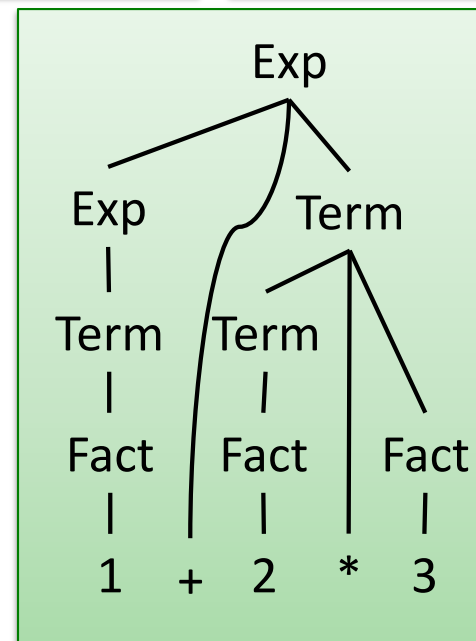
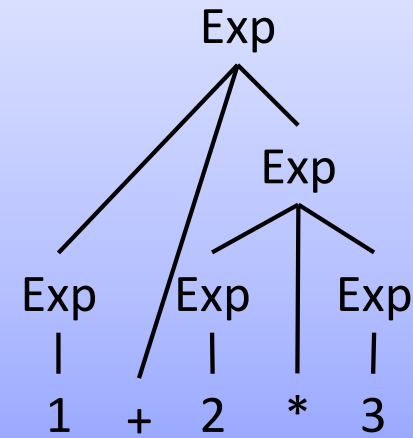
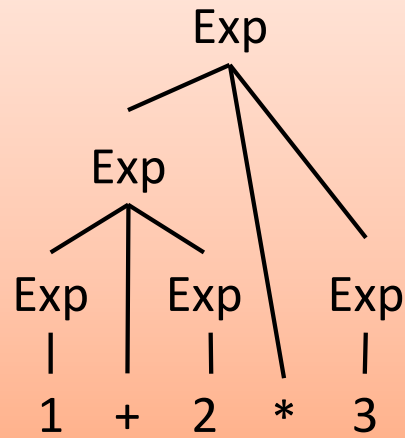
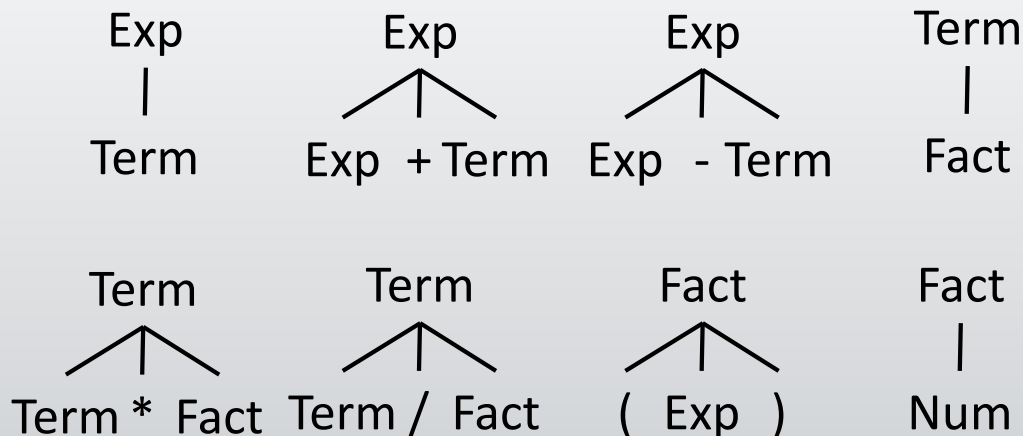
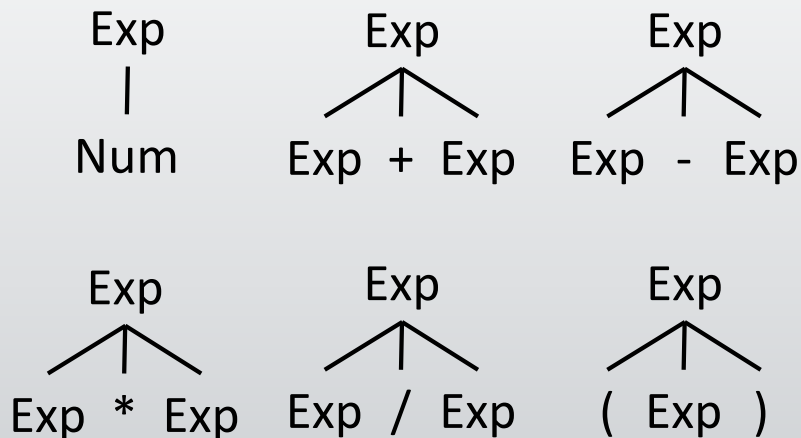
Ambiguous derivation!
How to disambiguate?

■ Ambiguous grammar:

- Generates two distinct parse trees
- Serious problem for a parser!
- Difficult to disambiguate!

Idea: Change the grammar to force the correct parse tree!

Disambiguation: Binary Expression

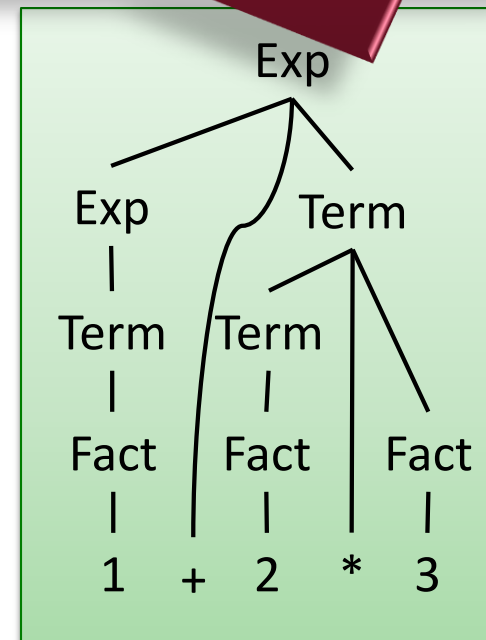
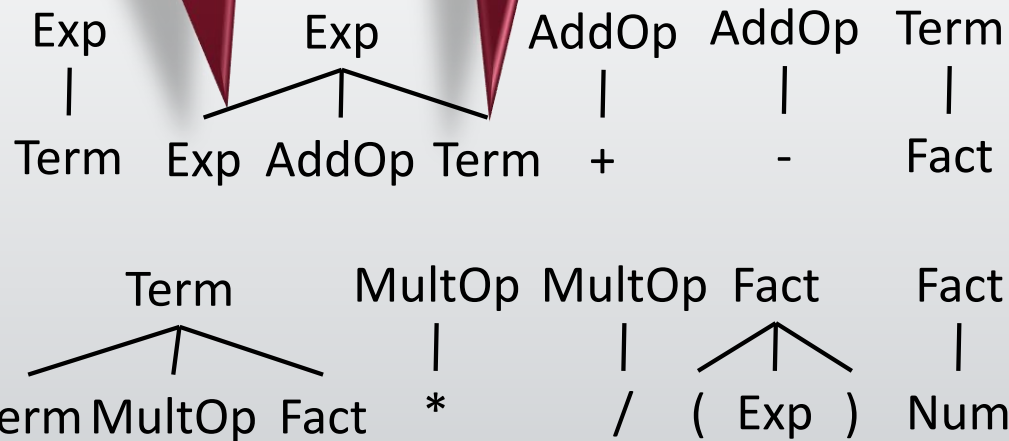


Simplifying Grammar: Binary Expression

Precedence:
multiplicatives bind
stronger than additives

Parse tree respects **associativity**
and **operator precedence**

Left
associative



Disambiguation: If-then-else

Grammar: (bold=terminals)

$\text{Stmt} \rightarrow \text{IfStmt} \mid \mathbf{other}$

$\text{IfStmt} \rightarrow$

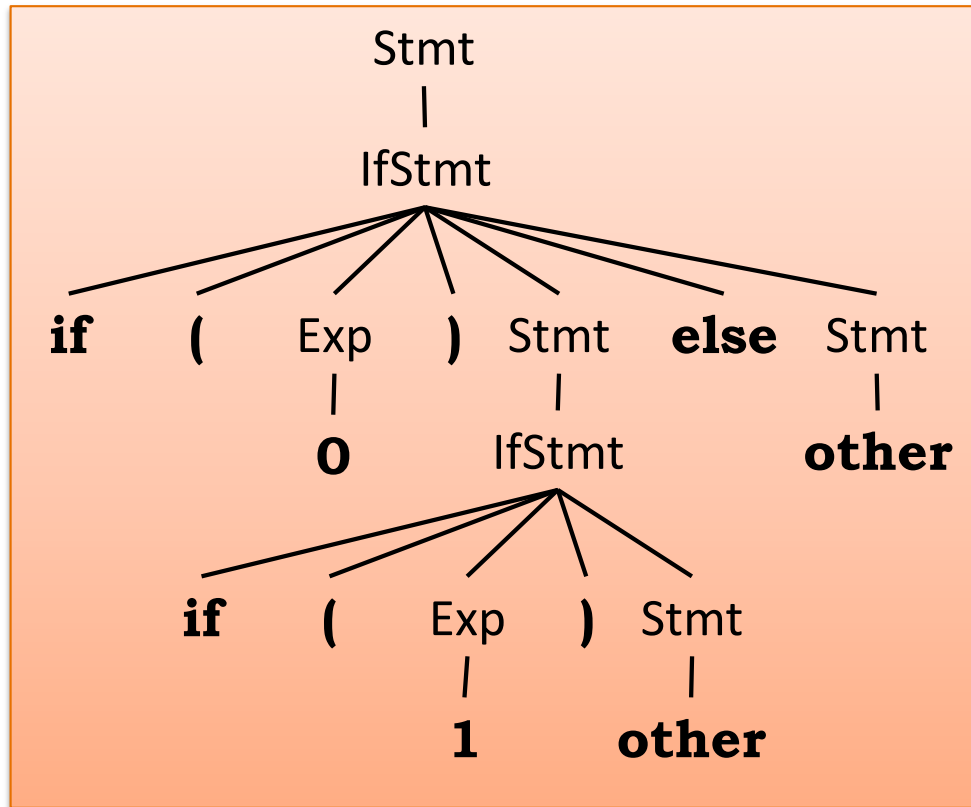
$\mathbf{if} (\text{Exp}) \text{Stmt} \mid$

$\mathbf{if} (\text{Exp}) \text{Stmt} \mathbf{else} \text{Stmt}$

$\text{Exp} \rightarrow \mathbf{0} \mid \mathbf{1}$

Ambiguous sample program:

if (0) if (1) other else other



Disambiguation: If-then-else

Grammar: (bold=terminals) **Ambiguous sample program:**

$\text{Stmt} \rightarrow \text{IfStmt} \mid \text{other}$

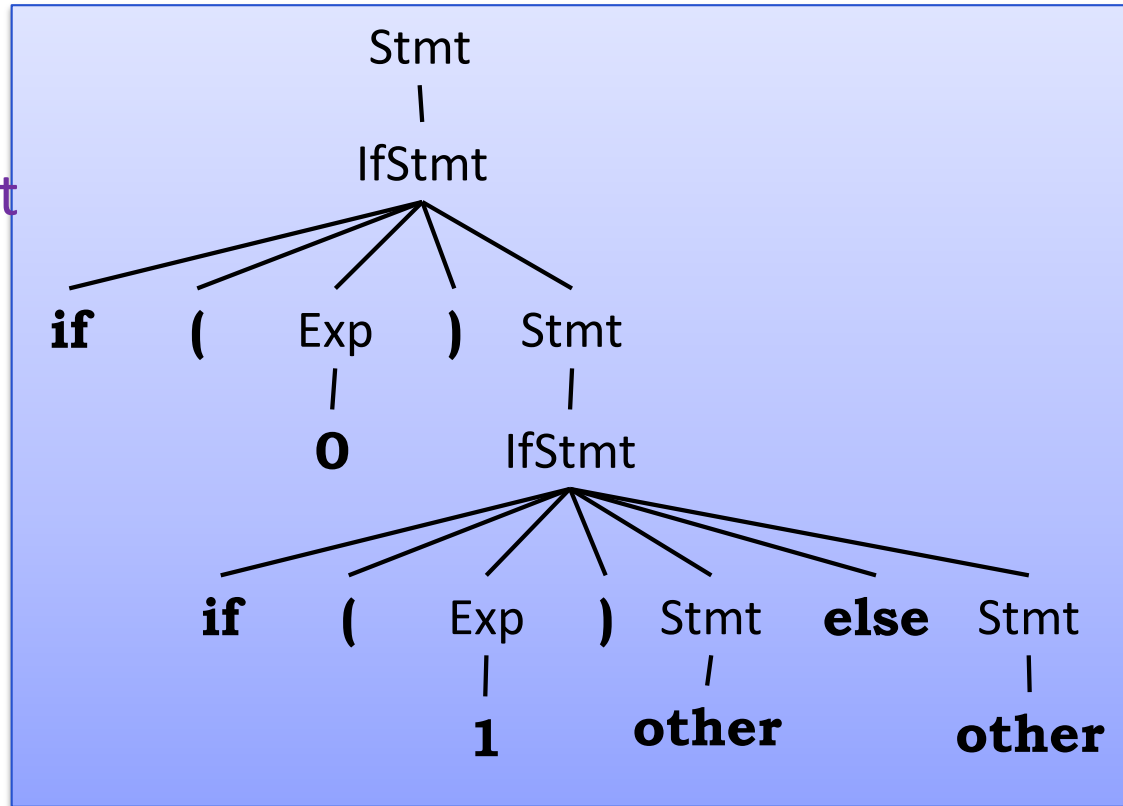
if (0) if (1) other else other

$\text{IfStmt} \rightarrow$

if (Exp) Stmt |

if (Exp) Stmt **else** Stmt

$\text{Exp} \rightarrow 0 \mid 1$



Disambiguation rule:
Most closely nested else

Disambiguation: If-then-else

Grammar: (bold=terminals)

Stmt →

UnMatched | Matched

Matched →

if (Exp) Matched **else** Matched
| **other**

UnMatched →

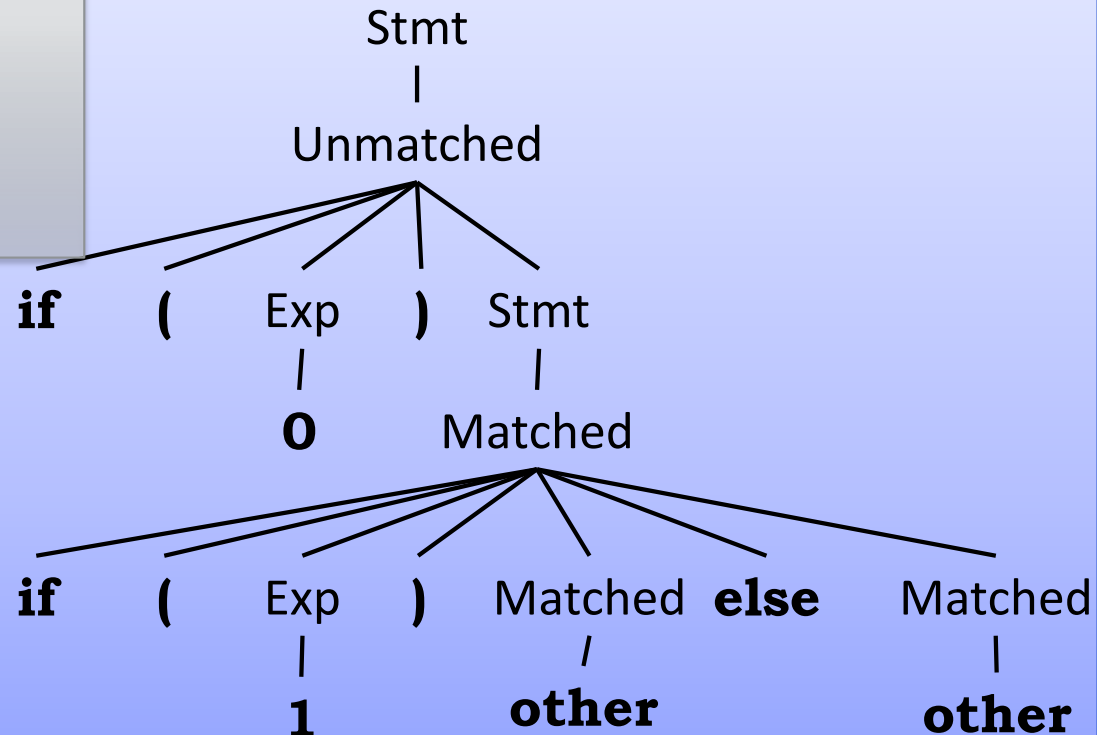
if (Exp) Stmt |
if (Exp) Matched **else**
UnMatched

Exp → **0** | **1**

Unambiguous program:

(else matched to 2nd if construct)

if (0) if (1) other else other



More on Associativity and Left Recursion

■ Grammar variants for ternary ?:

○ Naive solution

- Ambiguity: $a?b:c?d:e?f:g$

○ Bind to right (right associative)

- $a?b:(c?d:(e?f:g))$
- Use same trick as if-then-else

○ Bind to left (left associative)

- $((a?b:c)?d:e)?f:g$
- Problem: **left recursive grammar**

Difficult to parse (LR or backtracking LL needed)
as first few characters don't determine depth

If-then-else is earlier: # of ifs tells depth

Grammar: (bold=terminals)

$\text{Exp} \rightarrow$

$\text{Var} \mid \text{Exp} ? \text{Exp} : \text{Exp}$

Grammar: (bold=terminals)

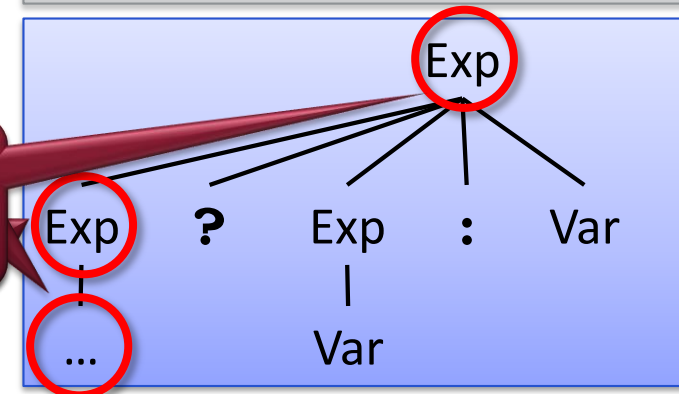
$\text{Exp} \rightarrow$

$\text{Var} \mid \text{Var} ? \text{Exp} : \text{Exp}$

Grammar: (bold=terminals)

$\text{Exp} \rightarrow$

$\text{Var} \mid \text{Exp} ? \text{Exp} : \text{Var}$



Factoring

Grammar: (bold=terminals)

Stmt →

UnMatched | Matched

Matched →

if (Exp) Matched **else** Matched
| **other**

UnMatched →

if (Exp) Stmt |
if (Exp) Matched **else**
UnMatched

Exp → **0** | **1**

- When parser reads „if ”
 - Can be the prefix of...
 - Matched₁
 - UnMatched₁
 - UnMatched₂
 - Can the parser avoid speculating?

Factoring transformation
automated (see later)
Don't worry about it

Factored grammar: (bold=terminals)

Stmt →

other | **if** (Exp) IfSuffix

...

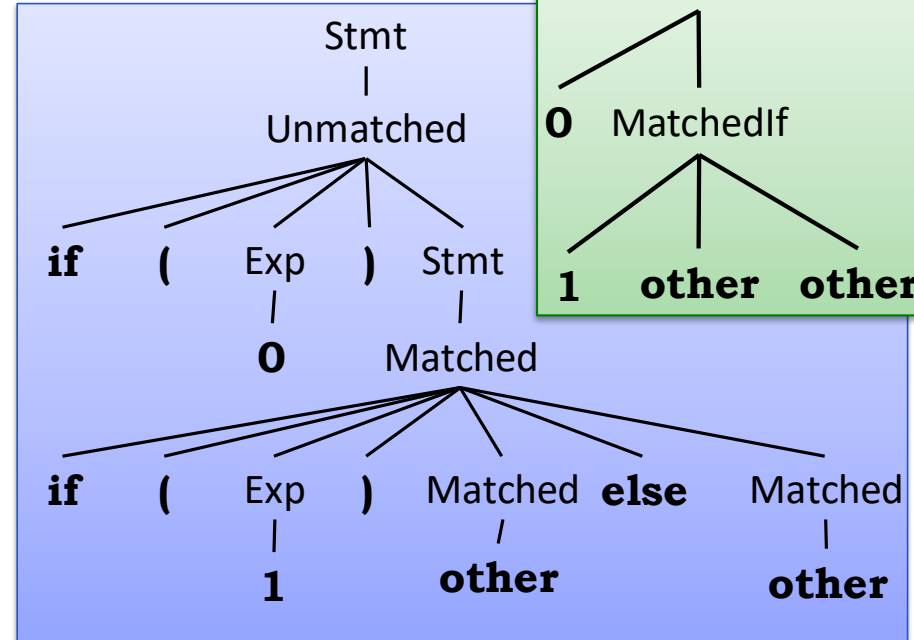
Parse Tree vs. Abstract Syntax Tree

■ Parse tree (Concrete syntax)

- Contains more information than necessary for a compiler
- ~ Notation model

■ Abstract Syntax Tree:

- Simplifications
 - Whitespace, syntactic sugar, etc.
 - Individual tokens cannot be recovered
 - Factoring, „Subclasses“ e.g. Stmt → Unmatched
 - „Optional“/„Repeating“ rules collapsed
- Defined by **high-level grammar** (EBNF)
 - **Block → Stmt+** **Block → Stmt | Stmt Block**
 - Low-level generated from high-level



Used in
practical technologies
(e.g. Xtext)

no need to understand
low-level

Grammar Design Conclusions

- Design grammar according to...
 - Precedence of constructs
 - Rules delegate to each other in order of precedence
 - Left/right binding (associativity)
 - Force associativity by controlling which branch is recursive
- Keep your parser happy
 - Avoid ambiguity → single derivation for single string
 - Try to avoid left recursion by...
 - Recursion through discriminator e.g. `”(”...”)`, or
 - Force right associativity
- Keep your human happy (be concise yet readable)