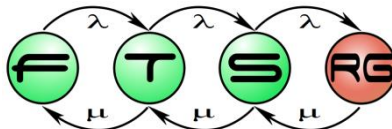


# Textual Modeling Languages

## Model Driven Software Development Lecture 7

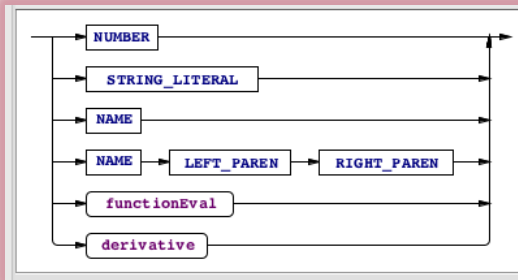


# Looking Inside Advanced IDEs

From parsers to development tools

# Parsers: The Traditional Setup

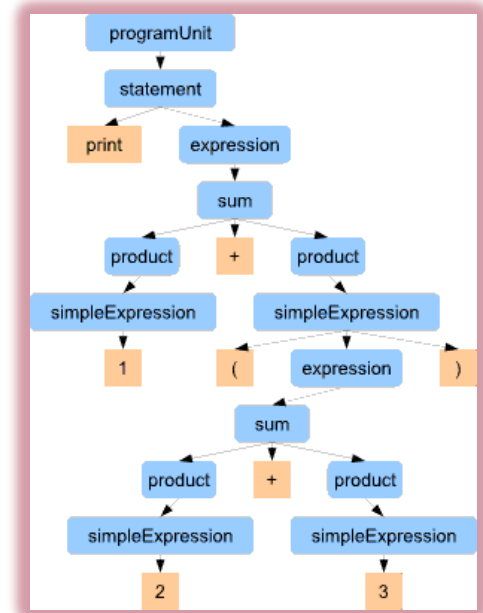
Grammar



```
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

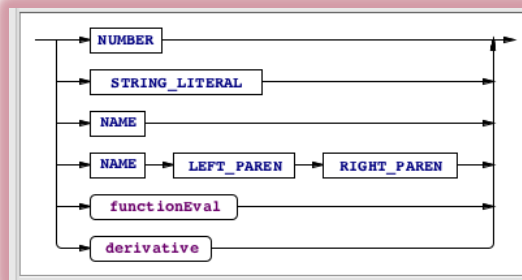
Parsing  
(lexer + parser)



Abstract  
syntax tree (AST)

# Parsers in Software Engineering Practice

Grammar

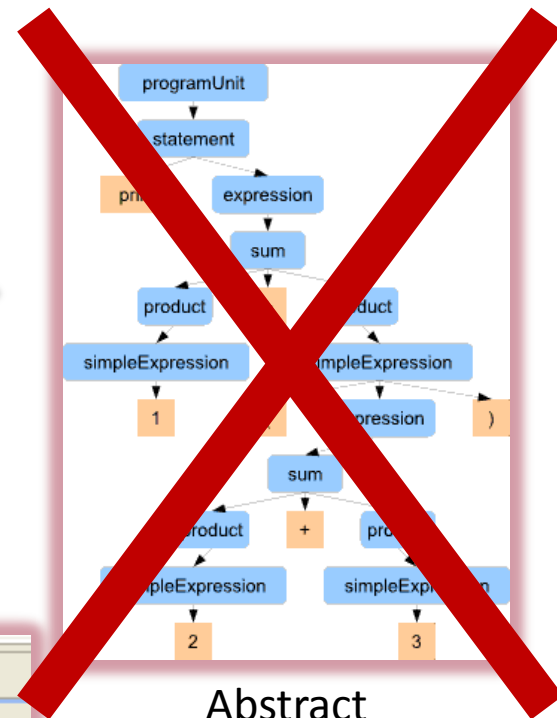
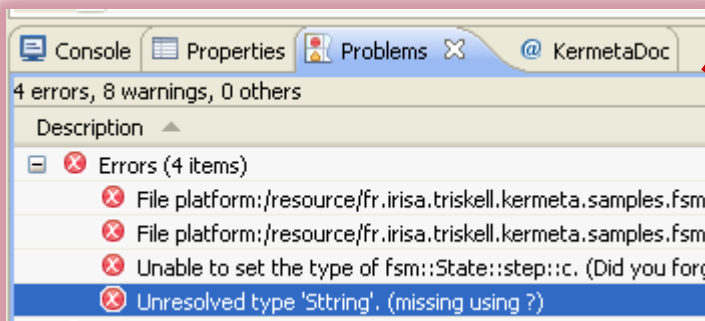


```
import com.lauchenauer.istockhelper.  
import com.lauchenauer.lib.ui.Vertic  
public class AboutDialog extends JDia  
protected CardLayout mLayout;  
protected JButton mCredits;  
protected JPanel mMainPanel;  
public AboutDialog(JFrame owner) {  
    super(owner);  
    setModal(true);  
    setUndecorated(true);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

Source code of program

Parsing  
(lexer + parser)

Error report



Abstract  
syntax tree (AST)

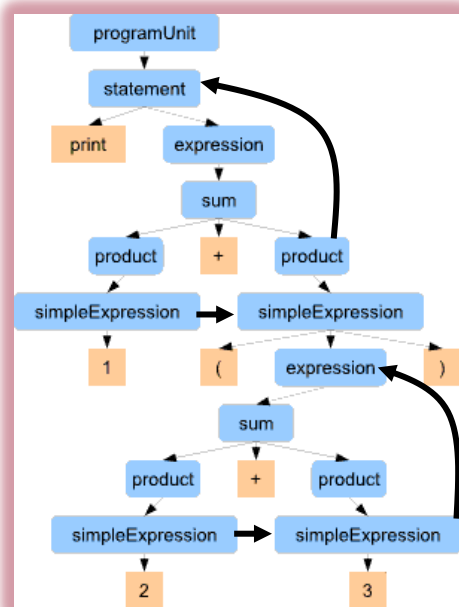
Error recovery parsing

# View Generation + Program Analysis

Call graph

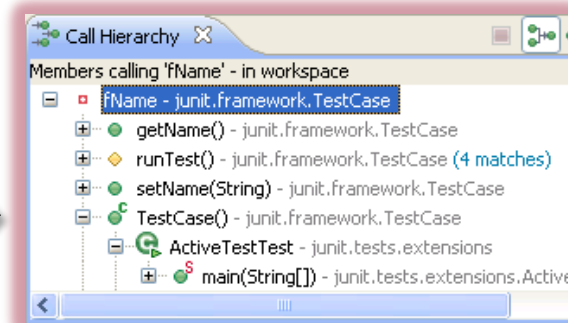
```
import com.lauchhammer.istockhelper...
public class AboutDialog extends JDialog
protected BorderLayout layout;
protected JButton mCredits;
protected JPanel mMainPanel;
public AboutDialog(JFrame owner) {
super(owner);
setModal(true);
setUndecorated(true);
initUI();
}
protected void initUI() {
setSize(440,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

Parsing



AST: Abstract syntax tree

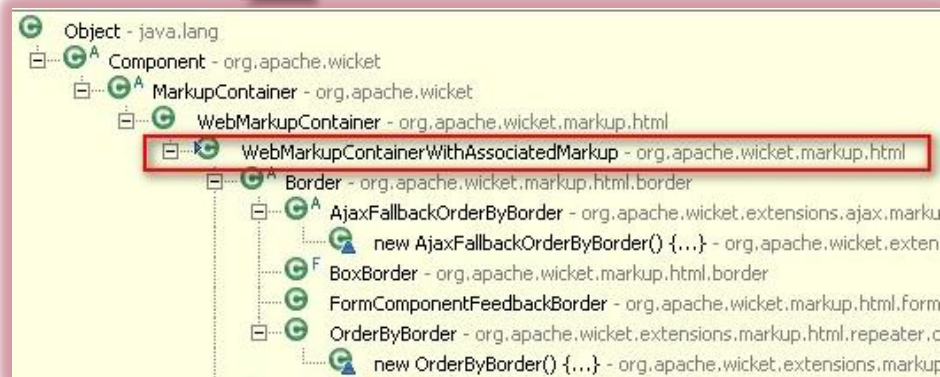
„Visitor”



„Visitor”



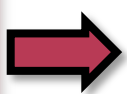
Type hierarchy



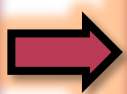
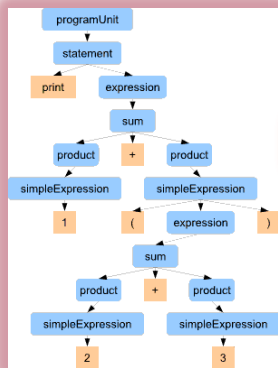
# ASTs vs DOMs

Source code of program

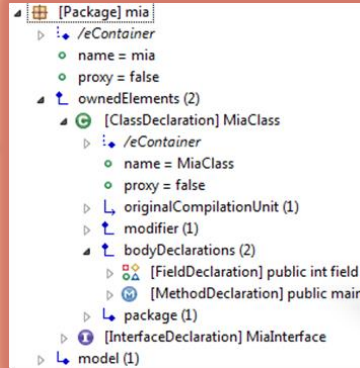
```
import com.lauchhammer.IStockHelper;
public class AboutDialog extends JDialog
protected CardLayout mLayout;
protected JButton mCredits;
protected JPanel mMainPanel;
public AboutDialog(JFrame owner) {
super(owner);
setModal(true);
setUndecorated(true);
initUI();
}
protected void initUI() {
setSize(440,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



AST: Abstract syntax tree



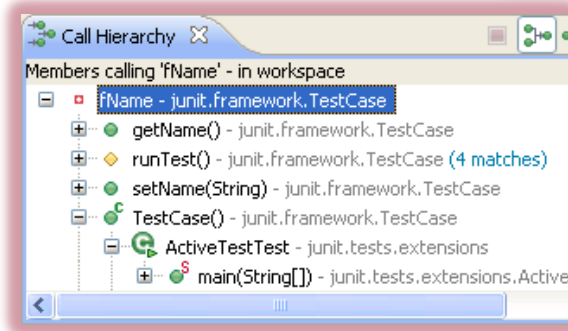
DOM: Document Object Model



Well-formedness constraints

```
Errors (4 items)
File platform:/resource/fr.iris.a.triskell.kerne
File platform:/resource/fr.iris.a.triskell.kerne
Unable to set the type of fsm::State::step:
Unresolved type 'Sstring'. (missing using ?)
```

Defined by a metamodel



Call graph (View)



Type hierarchy (View)

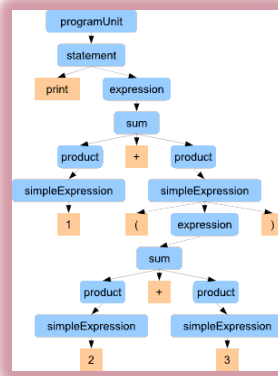
Eclipse IMP framework

# Refactoring

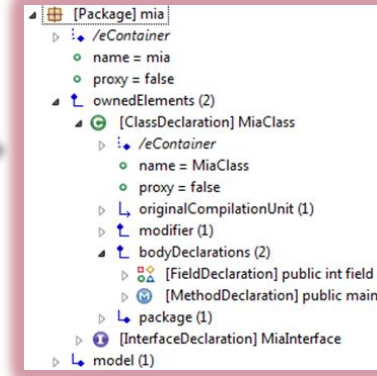
Source code <sub>1</sub>

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

AST <sub>1</sub>



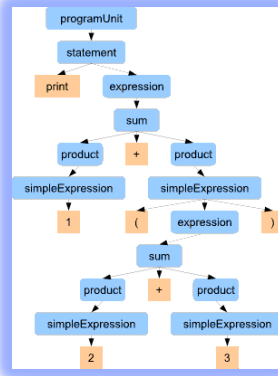
DOM <sub>1</sub>



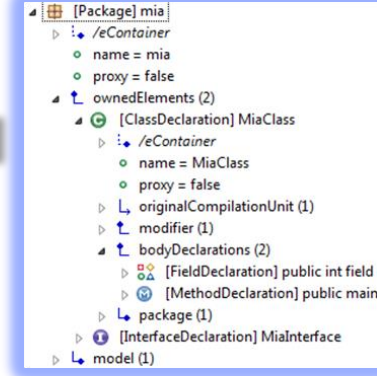
Source code <sub>2</sub>

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

AST <sub>2</sub>



DOM <sub>2</sub>



# Textual DSM Languages: An Overview

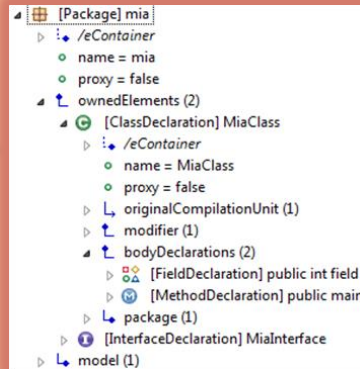
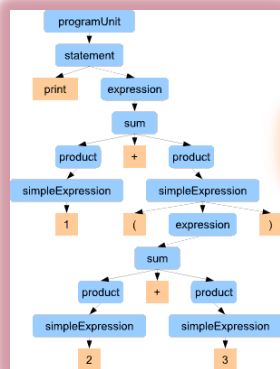
Source code

AST

DOM /  
Abstract syntax

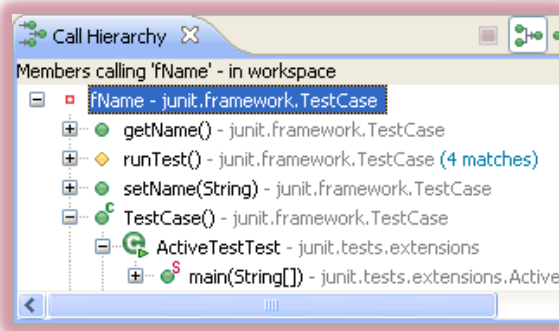
Well-formedness  
constraints

```
import com.laughnauer.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 = ...
}
```

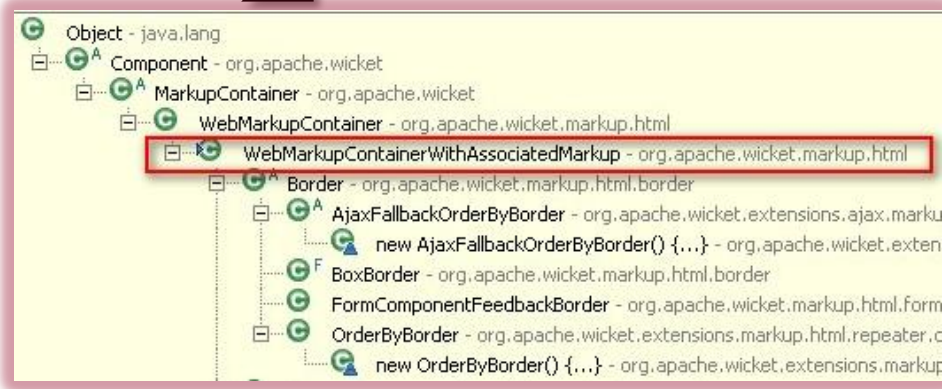


```
Errors (4 items)
File platform:/resource/fr.iris.a.triskell.kerne
File platform:/resource/fr.iris.a.triskell.kerne
Unable to set the type of fsm::State::step:
Unresolved type 'Sstring'. (missing using ?)
```

Refactoring,  
Simulation step



Call graph  
(Analysis model / View)

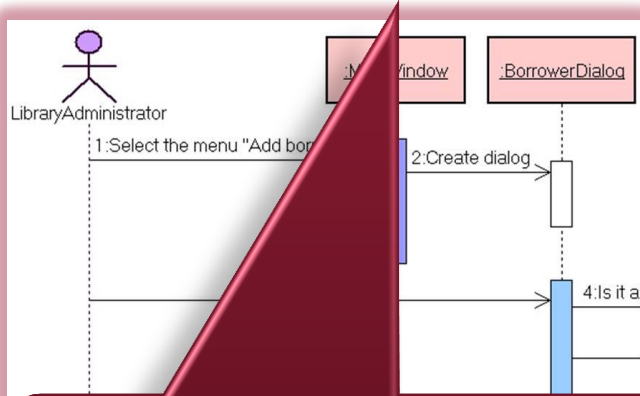


Type hierarchy  
(Analysis model / View)

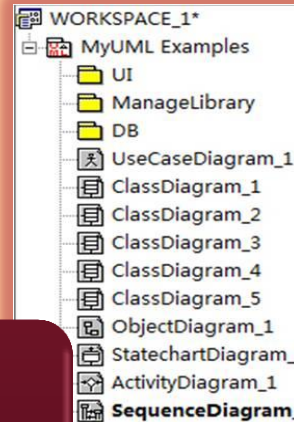


# Graphical DSM Languages

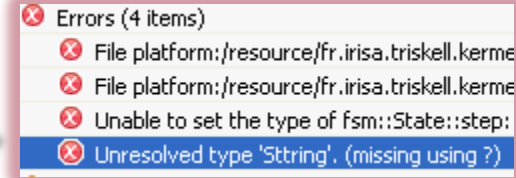
## Diagram model



## Abstract syntax

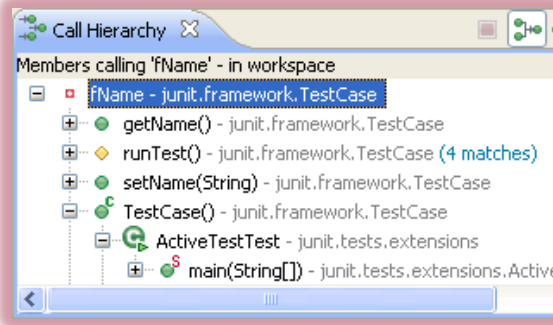


## Well-formedness constraints

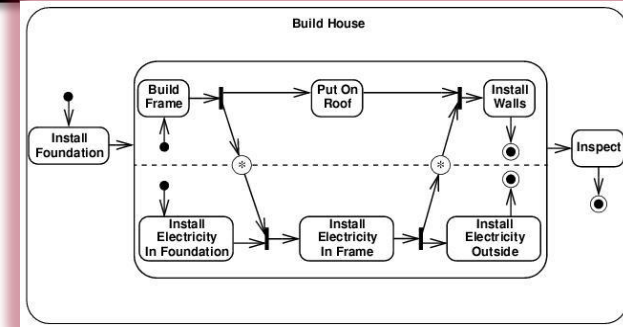


## Refactoring, Simulation

We said diagram model  
The diagram image itself is not parsed!



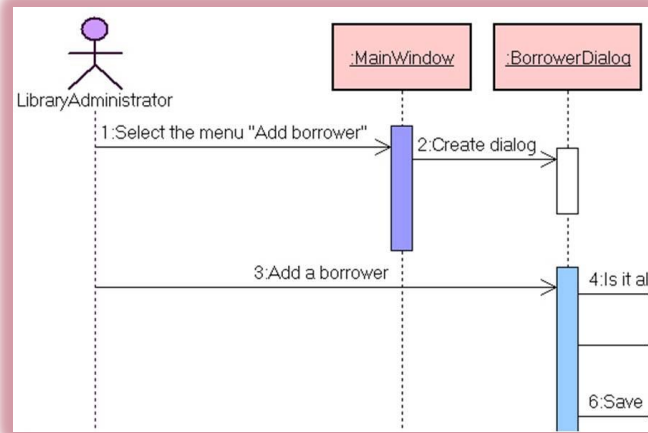
Call graph (View)



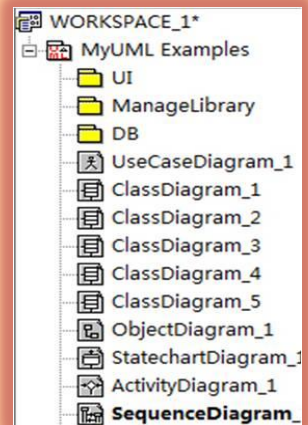
Statechart (other DSM)

# Graphical DSM Languages

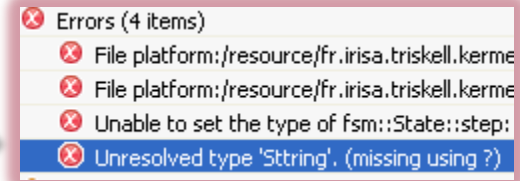
Diagram image



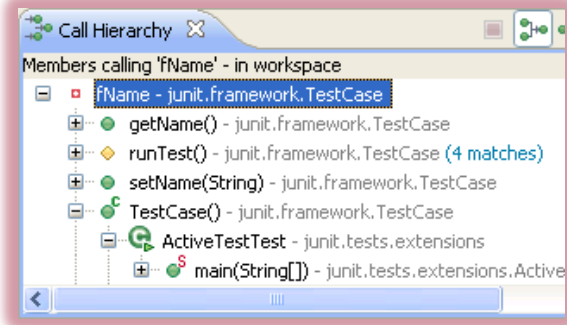
Abstract syntax  
incl. notation



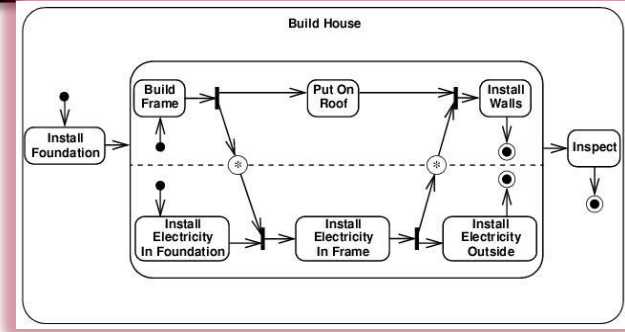
Well-formedness  
constraints



Refactoring,  
Simulation



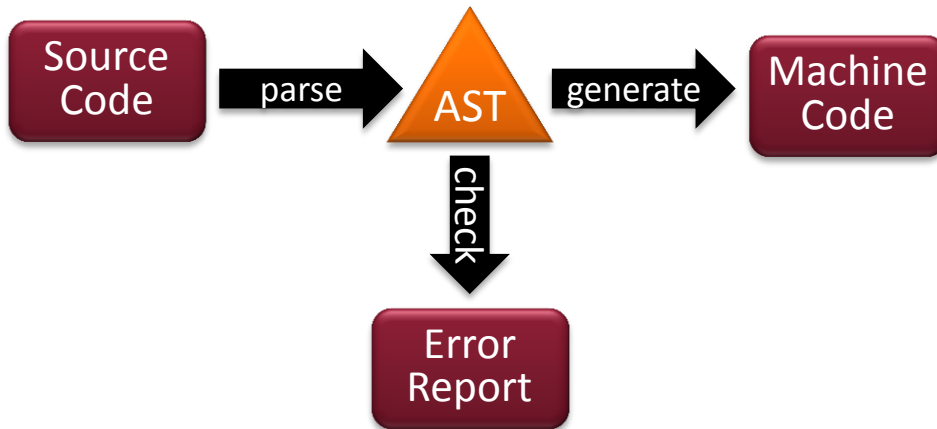
Call graph (View)



Statechart  
(other DSM)

# Architecture of Compilers

# 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 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)

# Textual modeling languages

# 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?





# Additionally

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

## Error localization!

# Example: Grammar for describing name lists

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

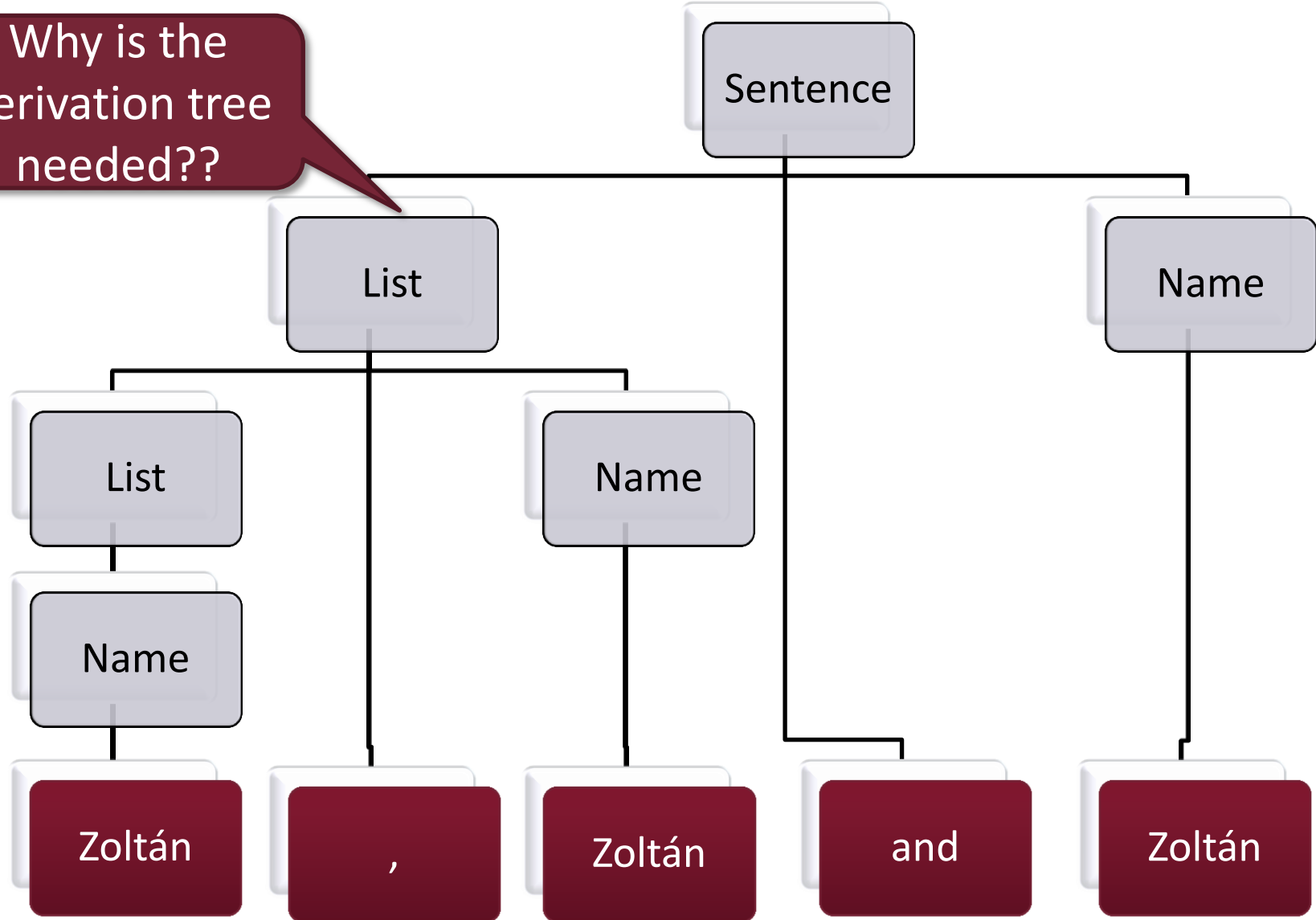
«Name» ::= *Zoltán* | *István* | *Dániel*

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

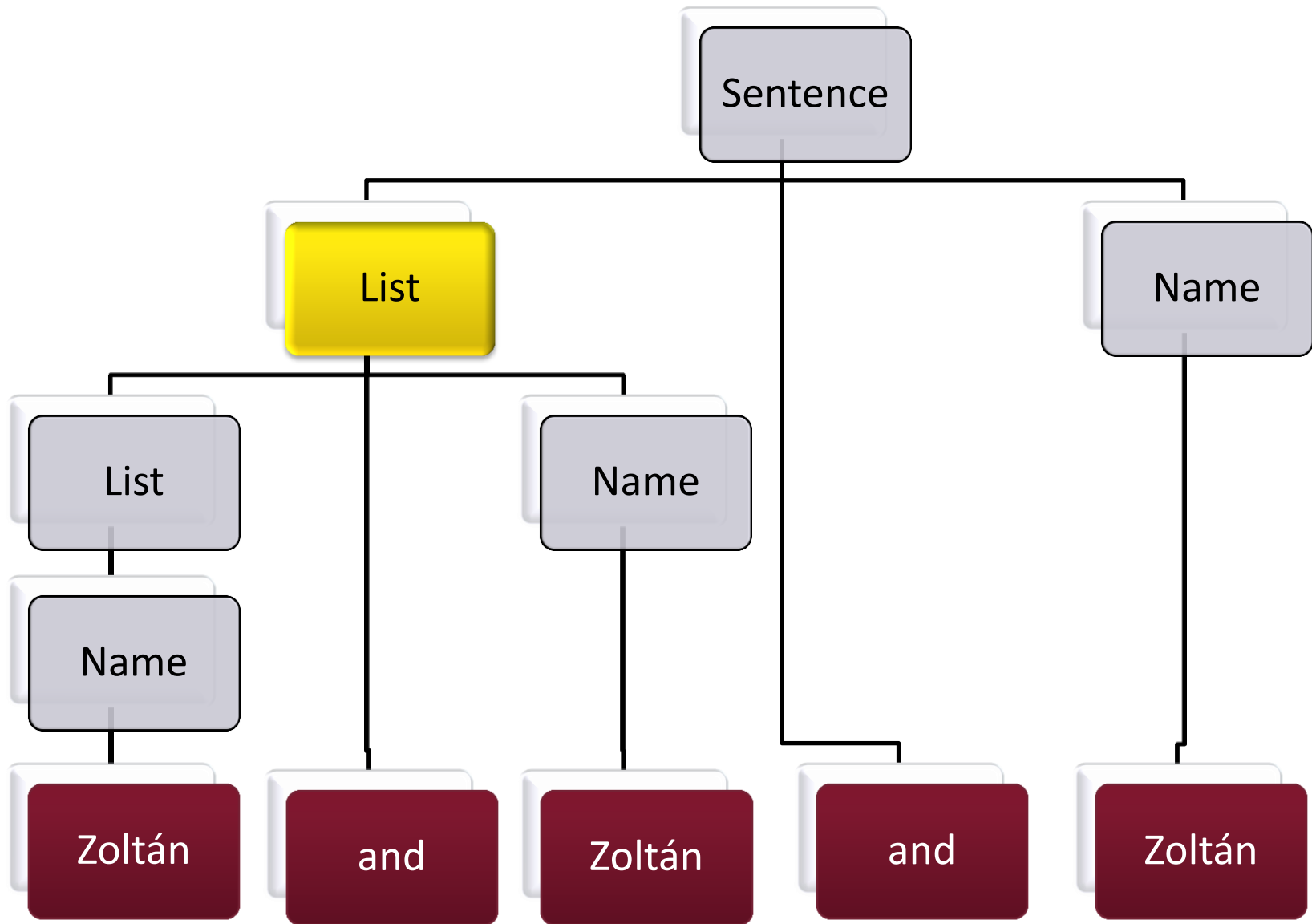
«List» ::= «List», «Name» | «Name»

# Derivation / Parse Tree

Why is the derivation tree needed??



# Derivation / Parse Tree



# 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 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;
```

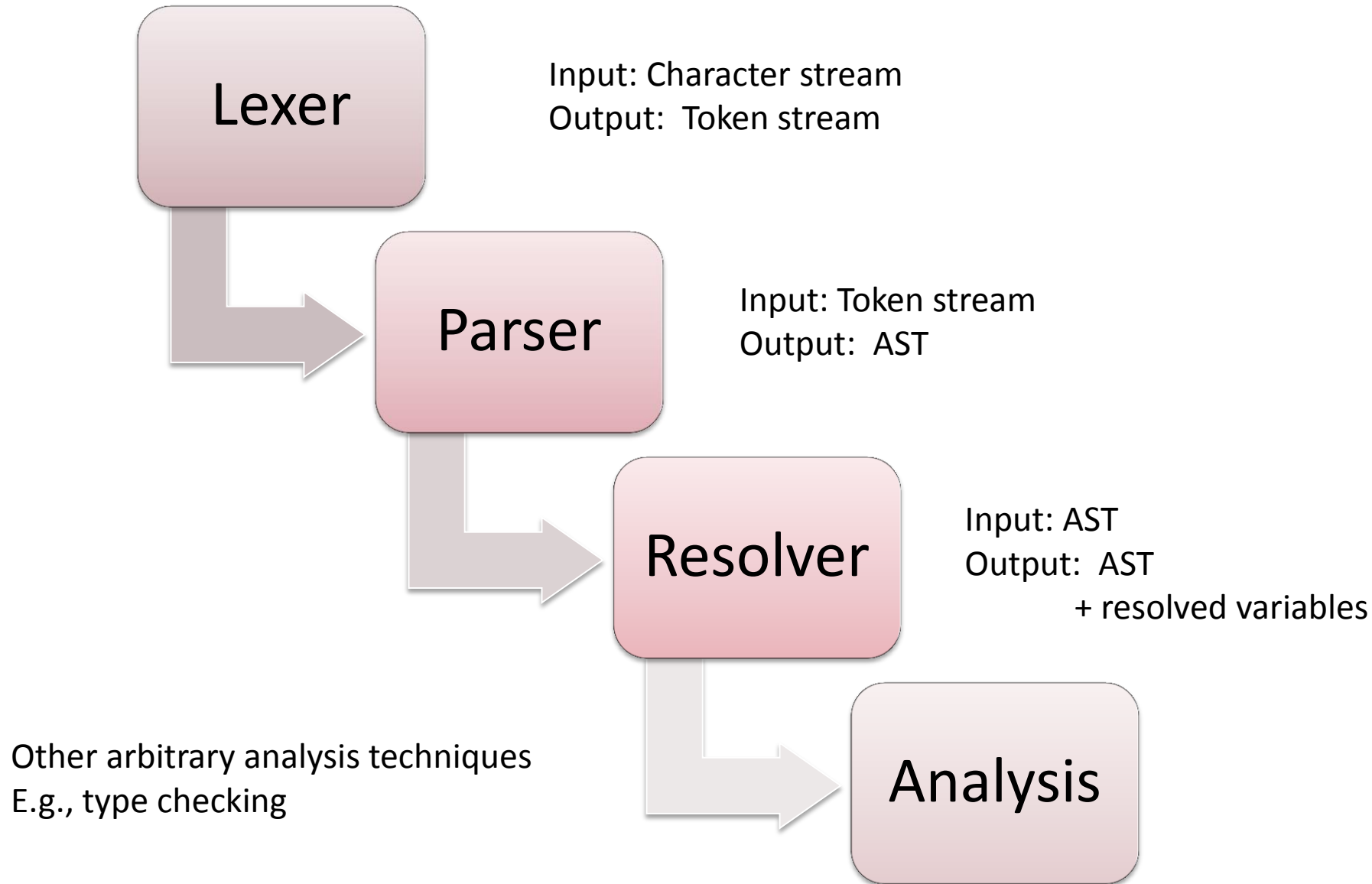
```
public void setValue(int value)
```

```
{  
    this.value = value;  
}
```

Which variable  
declaration is referred  
by 'value'?

- Solution
  - Resolver should define this
- Possible approaches
  - Most specific declaration
  - Conflict is error
  - Qualified references
  - ...

# The Parsing Process



# 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 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 development – DSLs over JVM:
  - Xbase expression language
  - JVM Model Inference instead of code generation

# Languages and Grammars

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

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

■ 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 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:

$\text{Exp}$

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

$1 + \text{Exp}$

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

$1 + 2 * \text{Exp}$

$1 + 2 * 3$

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

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

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

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

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

$\text{Exp}$

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

$\text{Exp} * 3$

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

$\text{Exp} + 2 * 3$

$1 + 2 * 3$

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

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

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

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

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

Problem:

$\text{Exp} * 3$

$1 + 2 * 3$

# Scanning and Parsing

## Tokenizer / Lexer / Scanner

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

Grammar:

$\text{Num} \rightarrow 0 \text{ Num} \mid \dots \mid 9 \text{ Num}$   
 $\text{Num} \rightarrow 0 \mid \dots \mid 9$

RegExp:

$\text{Num} = \text{Digit Digit}^*$   
 $\text{Digit} = (0 \mid 1 \mid \dots \mid 9)$

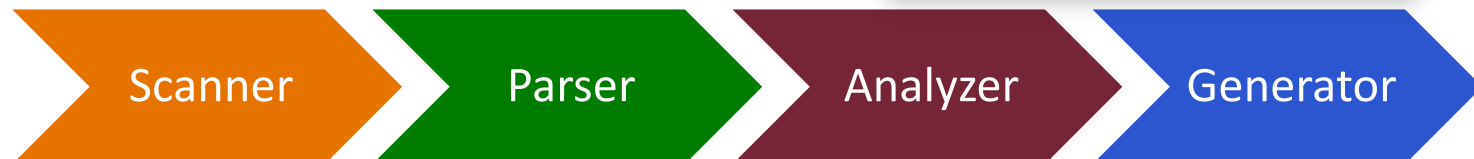
## Parser

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

Grammar:

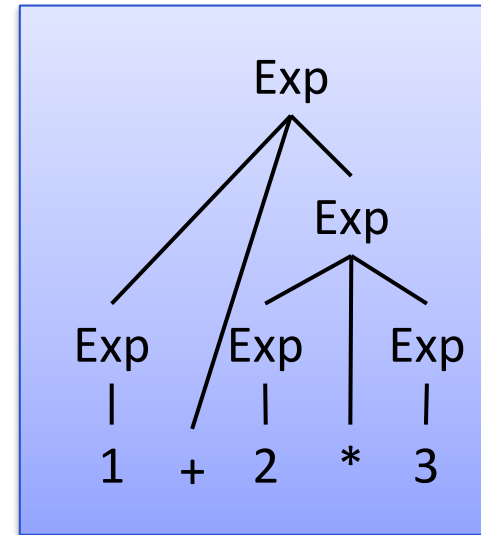
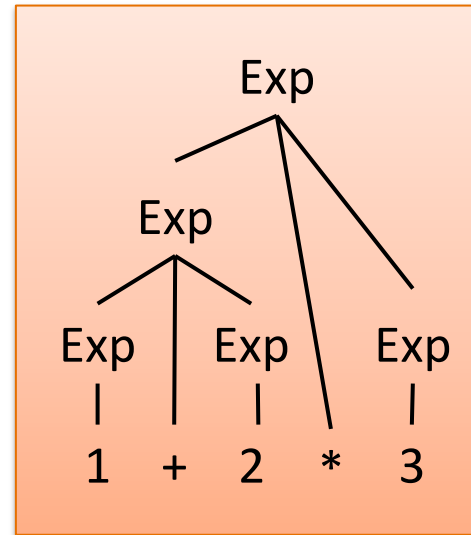
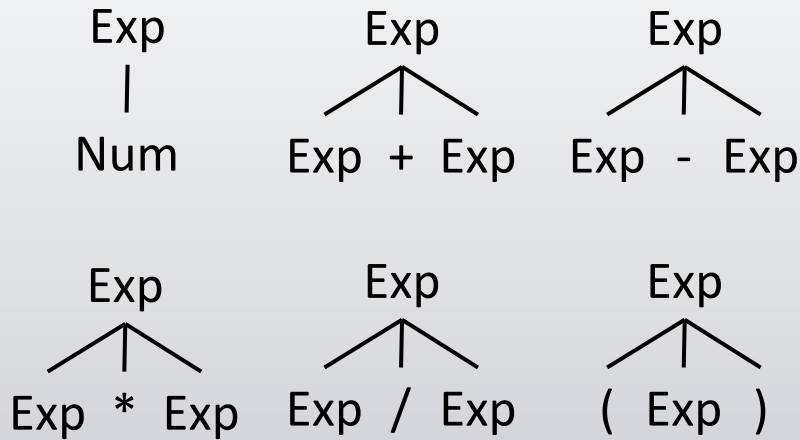
$\text{Exp} \rightarrow \text{Num} \mid \text{Exp} \text{ "+" } \text{Exp}$   
 $\mid \text{Exp} \text{ "-" } \text{Exp} \mid \text{Exp} \text{ "*" } \text{Exp}$   
 $\mid \text{Exp} \text{ "/" } \text{Exp} \mid \text{"(" Exp ")"}$

EBNF Notation



# Parse Trees

# Parse Tree Construction



## Parse Tree:

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

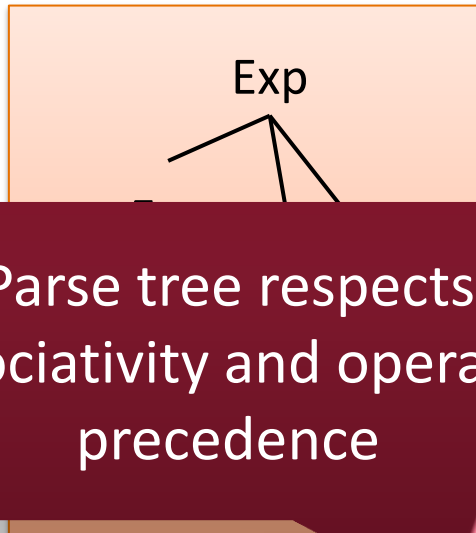
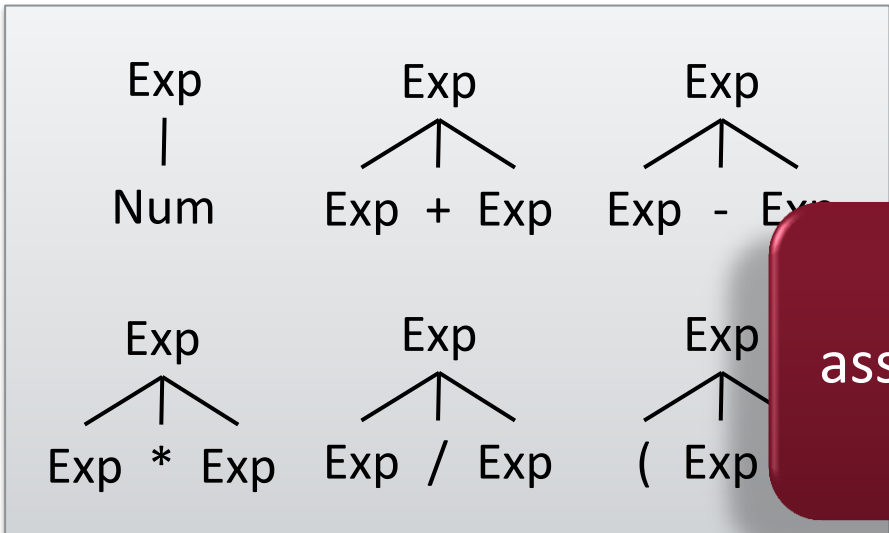
**Ambiguous derivation!**  
**How to disambiguate?**

## Ambiguous grammar:

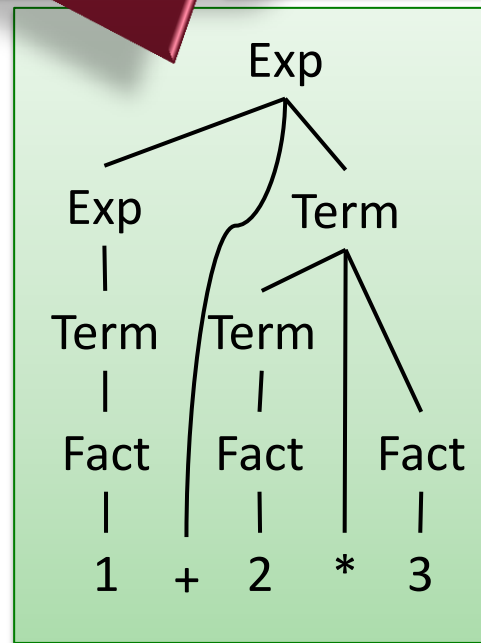
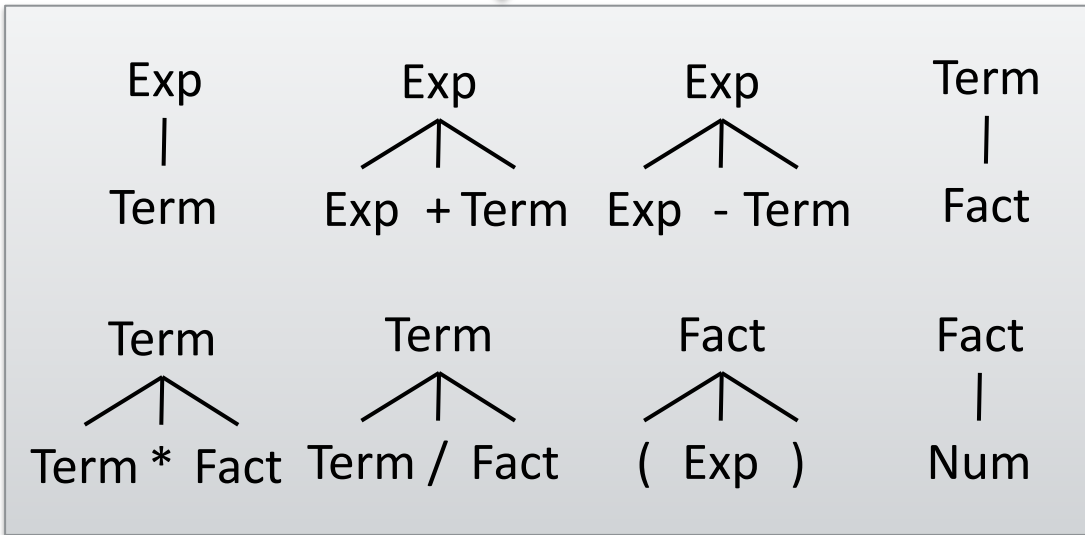
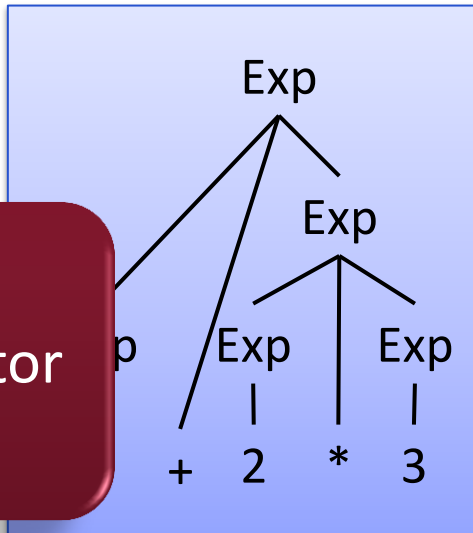
- Generates two distinct parse trees
- Serious problem for a parser!
- Difficult to disambiguate!

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

# Disambiguation: Binary Expression

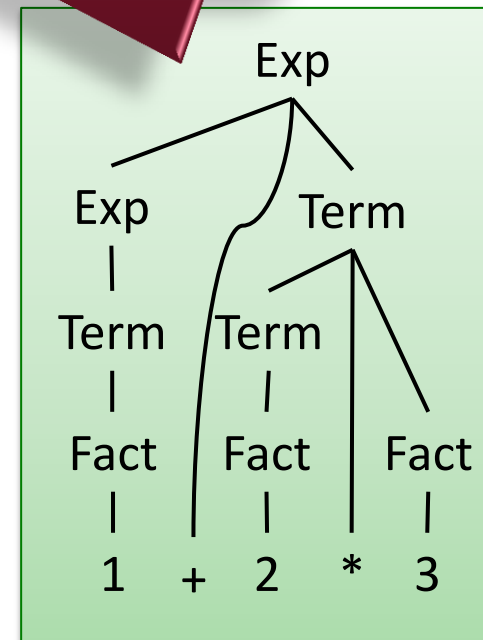
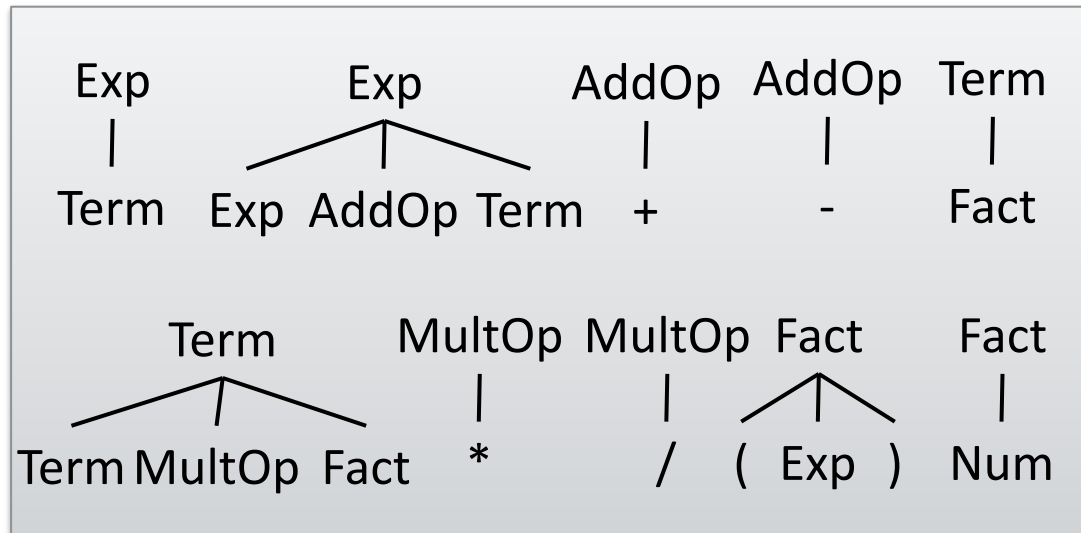


Parse tree respects associativity and operator precedence



# Simplifying Grammar: Binary Expression

Parse tree respects associativity and operator precedence





# Disambiguation: If-then-else

**Grammar:** (bold terminals)

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

$\text{IfStmt} \rightarrow$

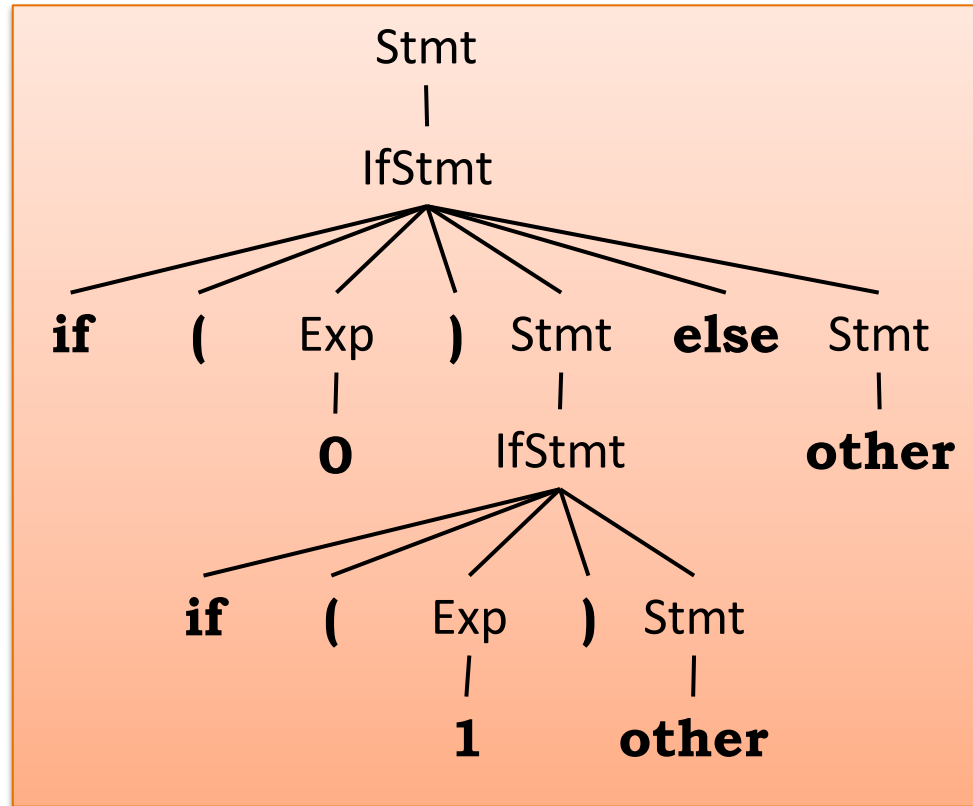
$\mathbf{if} ( \text{Exp} ) \text{Stmt} \mid$

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

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

Ambiguous sample program:

**if (0) if (1) other else other**



# Disambiguation: If-then-else

**Grammar:** (bold terminals)

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

$\text{IfStmt} \rightarrow$

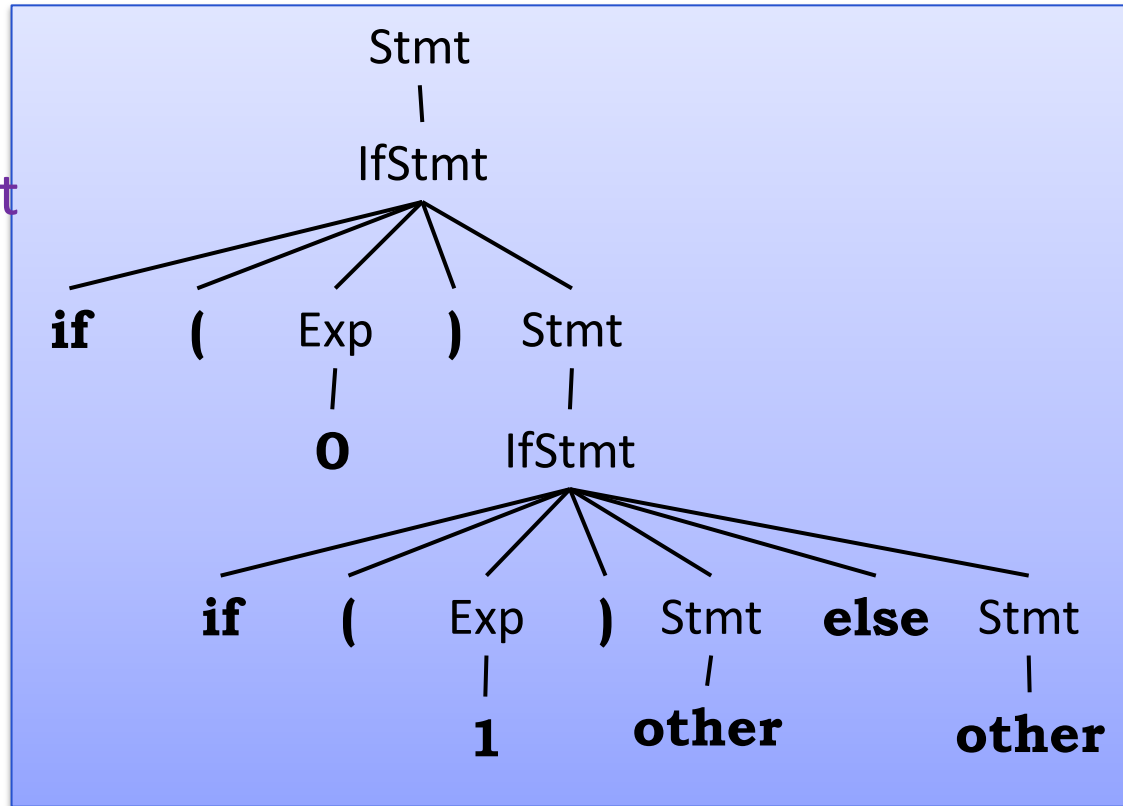
$\mathbf{if} ( \text{Exp} ) \text{ Stmt} \mid$

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

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

Ambiguous sample program:

**if (0) if (1) other else other**



Disambiguation rule:  
Most closely nested else

# Disambiguation: If-then-else

**Grammar:** (bold terminals)

Stmt  $\rightarrow$

UnMatched | Matched

Matched  $\rightarrow$

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

UnMatched  $\rightarrow$

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

Exp  $\rightarrow$  **0** | **1**

Unambiguous program:

(else matched to 2nd if construct)

**if (0) if (1) other else other**

