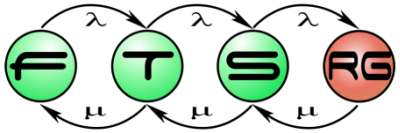
Textual Domain-specific languages



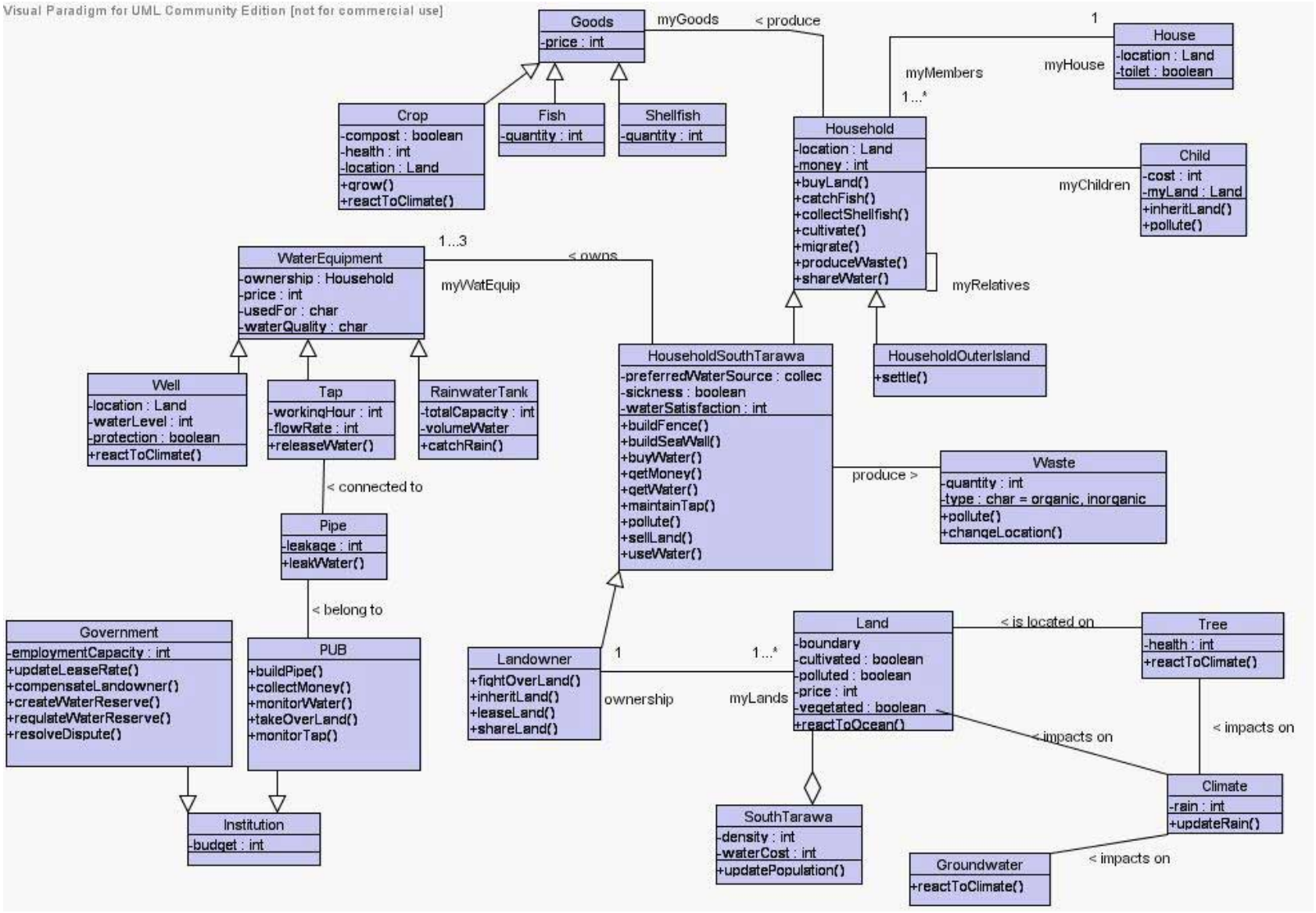
Designing modeling languages

- Metamodel: a model of models
  - Abstract syntax
  - Concrete syntax
    - Graphical model
    - **Textual model**
  - Well-formedness rules
  - Behavioral (dynamic) semantics
  - Translation to other languages

Evaluation of graphical editors

- Readable, easy to understand
- Editing slower
- Larger (20+ element) models: usability issues

Larger example model (Ontology)



Forrás: http://epress.anu.edu.au/cs/html/ch12s04.html

Editors for textual languages

- Alternative: textual editing
- Basic understanding
  - Widget to write text
  - Additional support functions

Levels of support

Levels of support

- Simple text editor
  - See Notepad
  - Basic operations (clipboard, find/replace, etc.)
  - Implemented generically in Eclipse
- Programmers editor
  - See Notepad++, Emacs, Vim, …
  - Language-dependent services (syntax highlight, …)
- Integrated editor
  - See Eclipse JDT, Visual Studio
  - Complex services (project support, autocomplete, documentation, …)

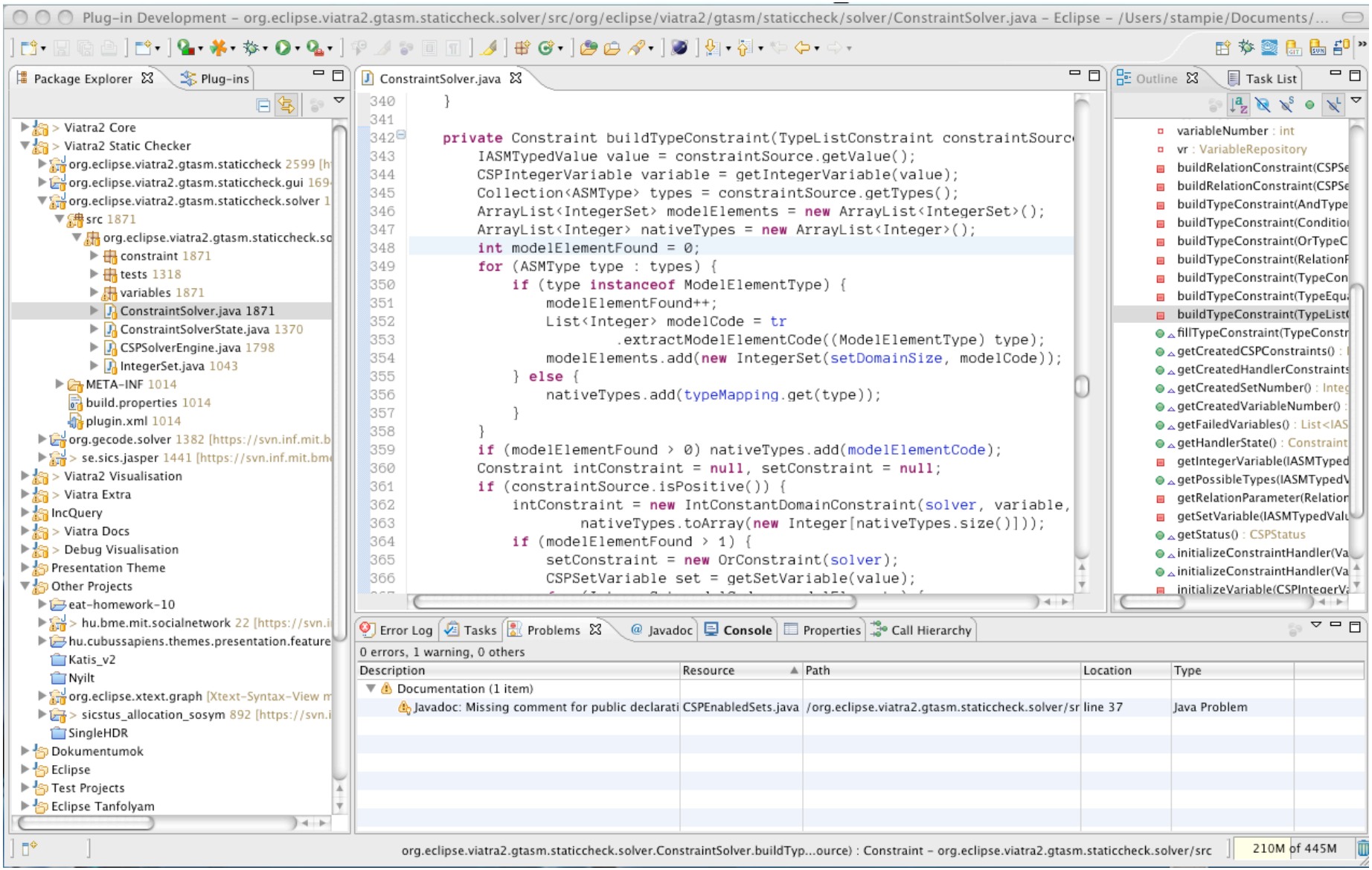
Levels of support

- Simple text editor
  - See Notepad
  - Basic operations (clipboard, find/replace, etc.)
  - Implemented generically in Eclipse
- Programmers editor
  - See Notepad++, Emacs, Vim, …
  - Language-dependent services (syntax highlight, …)
- Integrated editor
  - See Eclipse JDT, Visual Studio
  - Complex services (project support, autocomplete, documentation, …)

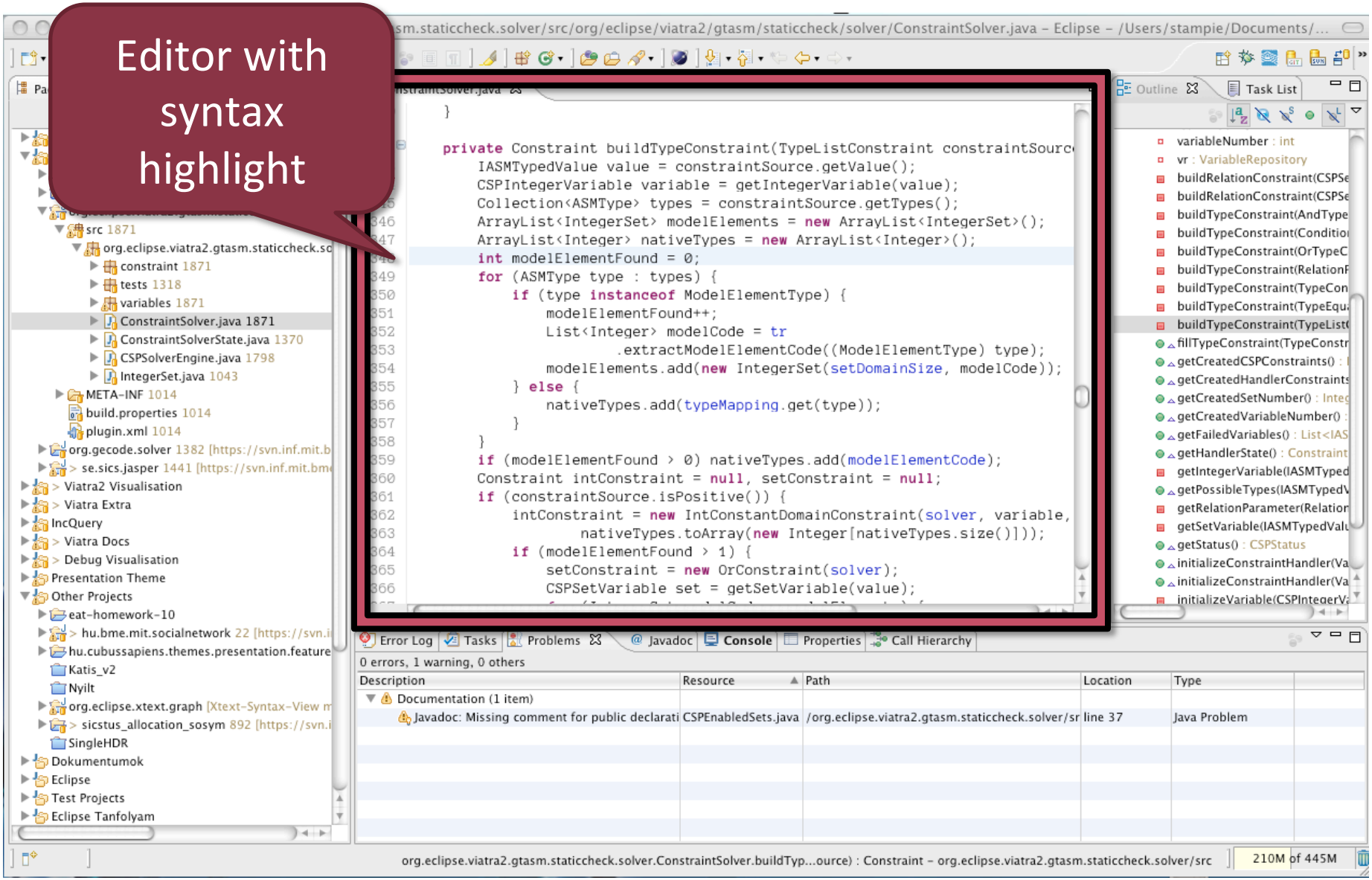
Levels of support

- Simple text editor
  - See Notepad
  - Basic operations (clipboard, find/replace, etc.)
  - Implemented generically in Eclipse
- Programmers editor
  - See Notepad++, Emacs, Vim, …
  - Language-dependent services (syntax highlight, …)
- Integrated editor
  - See Eclipse JDT, Visual Studio
  - Complex services (project support, autocomplete, documentation, …)

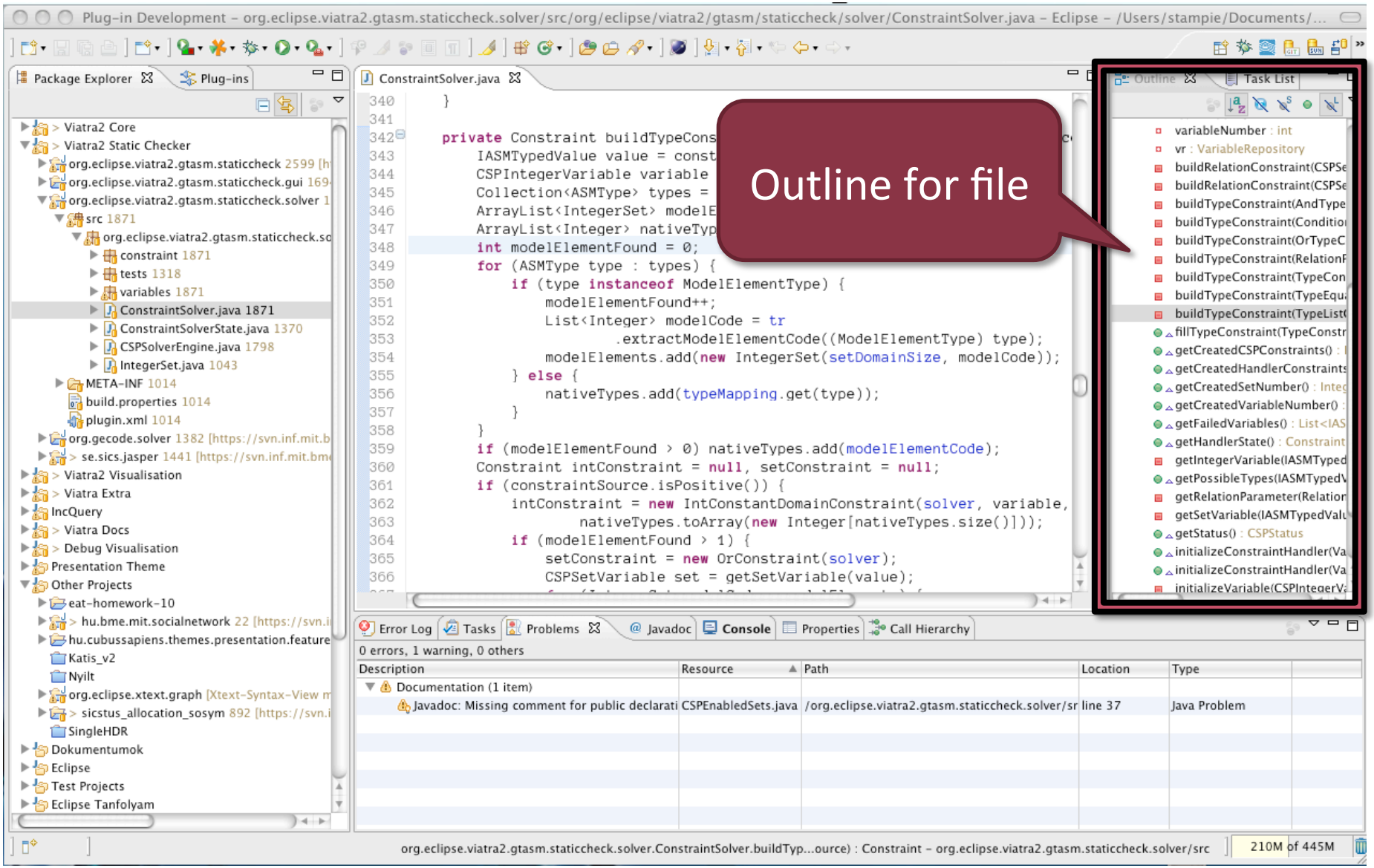
JDT services



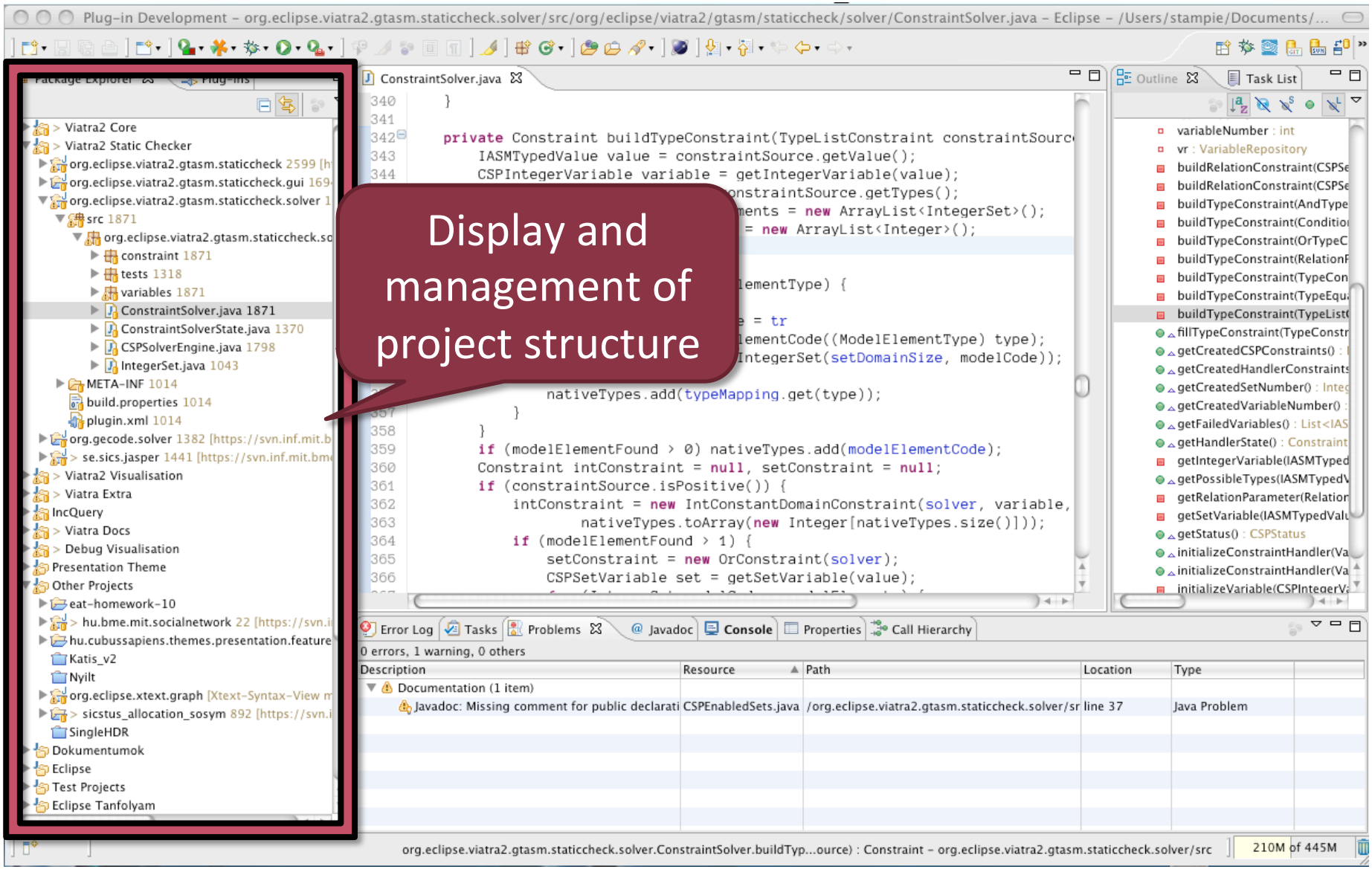
JDT services



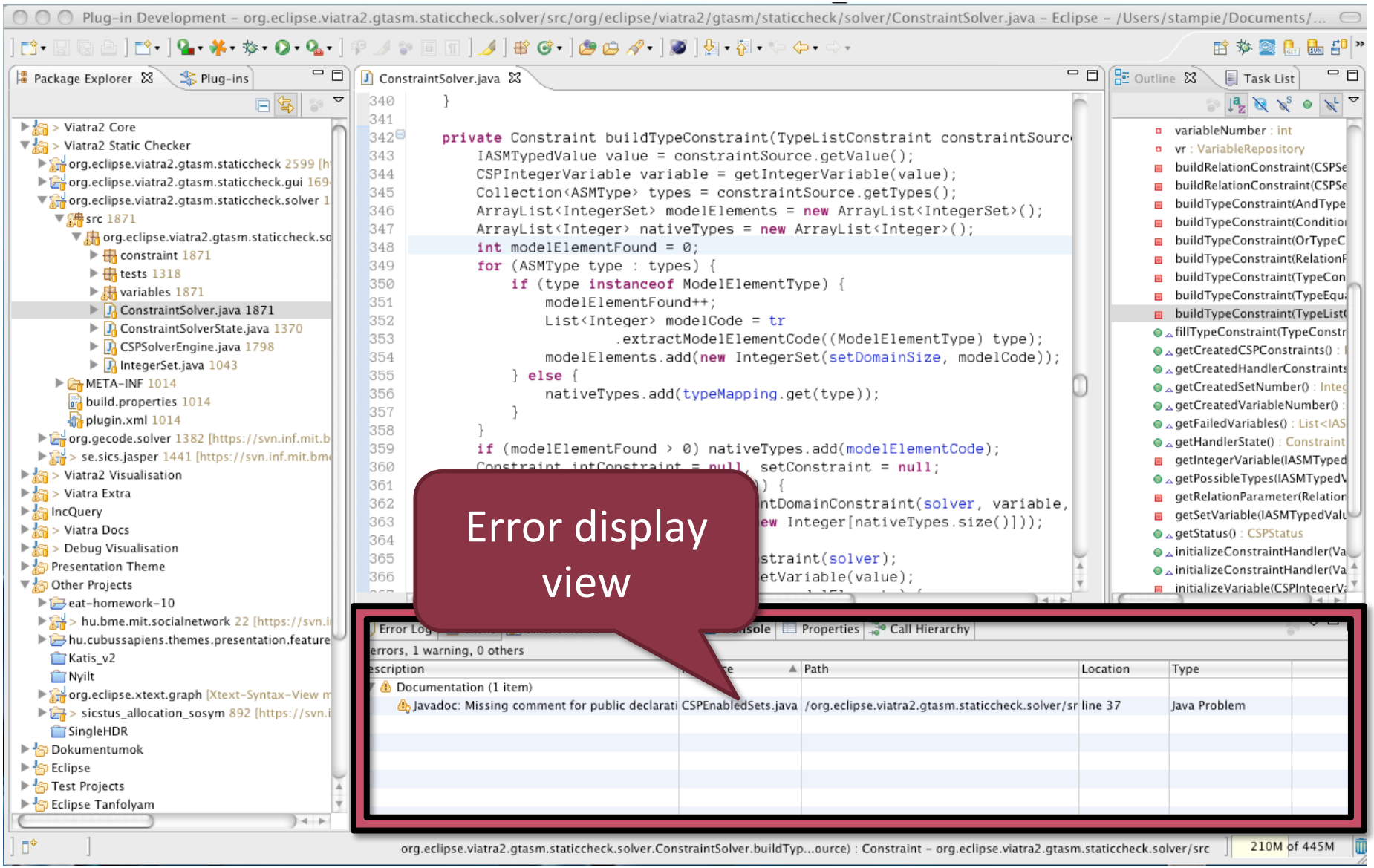
JDT services



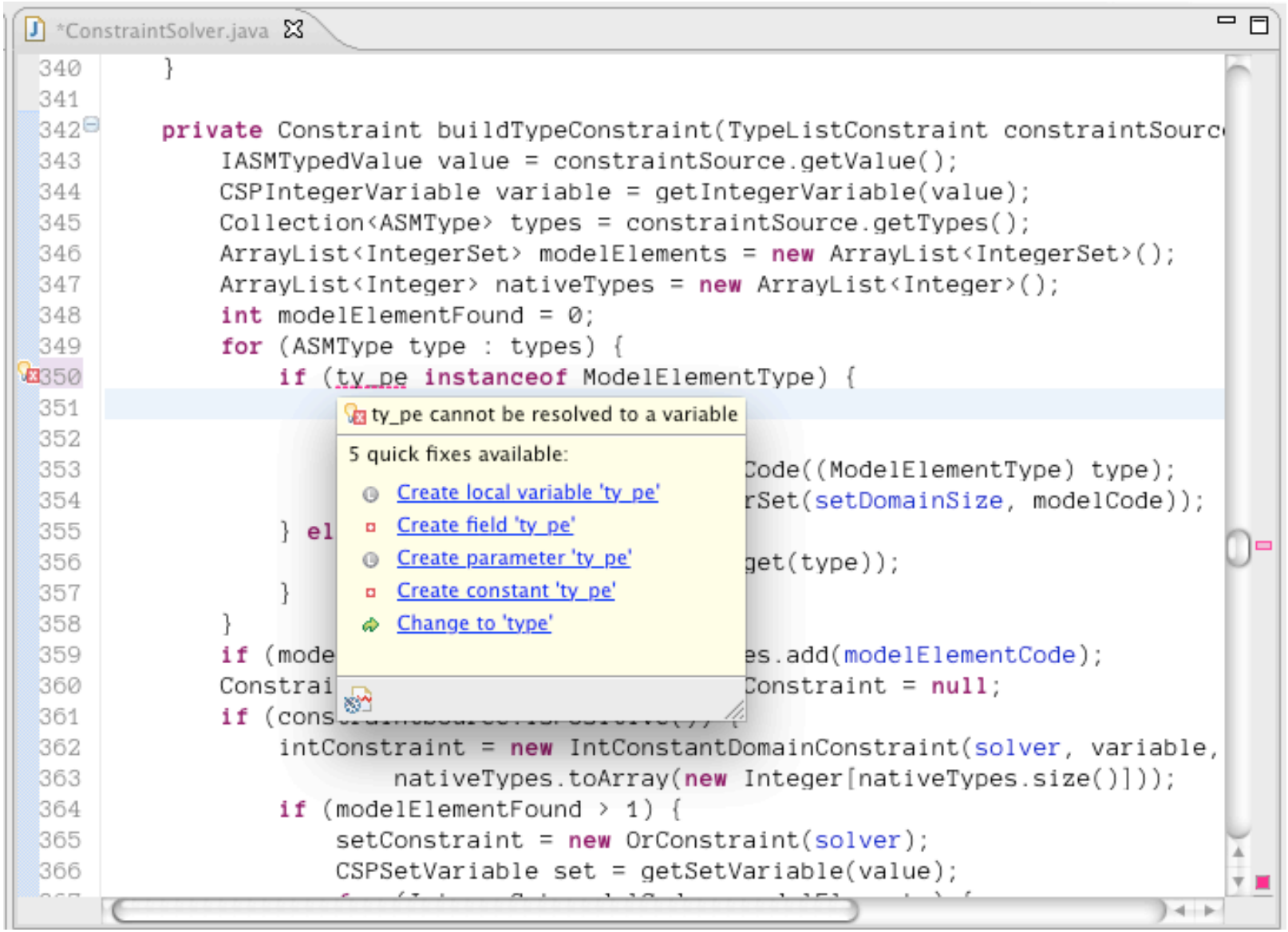
JDT services



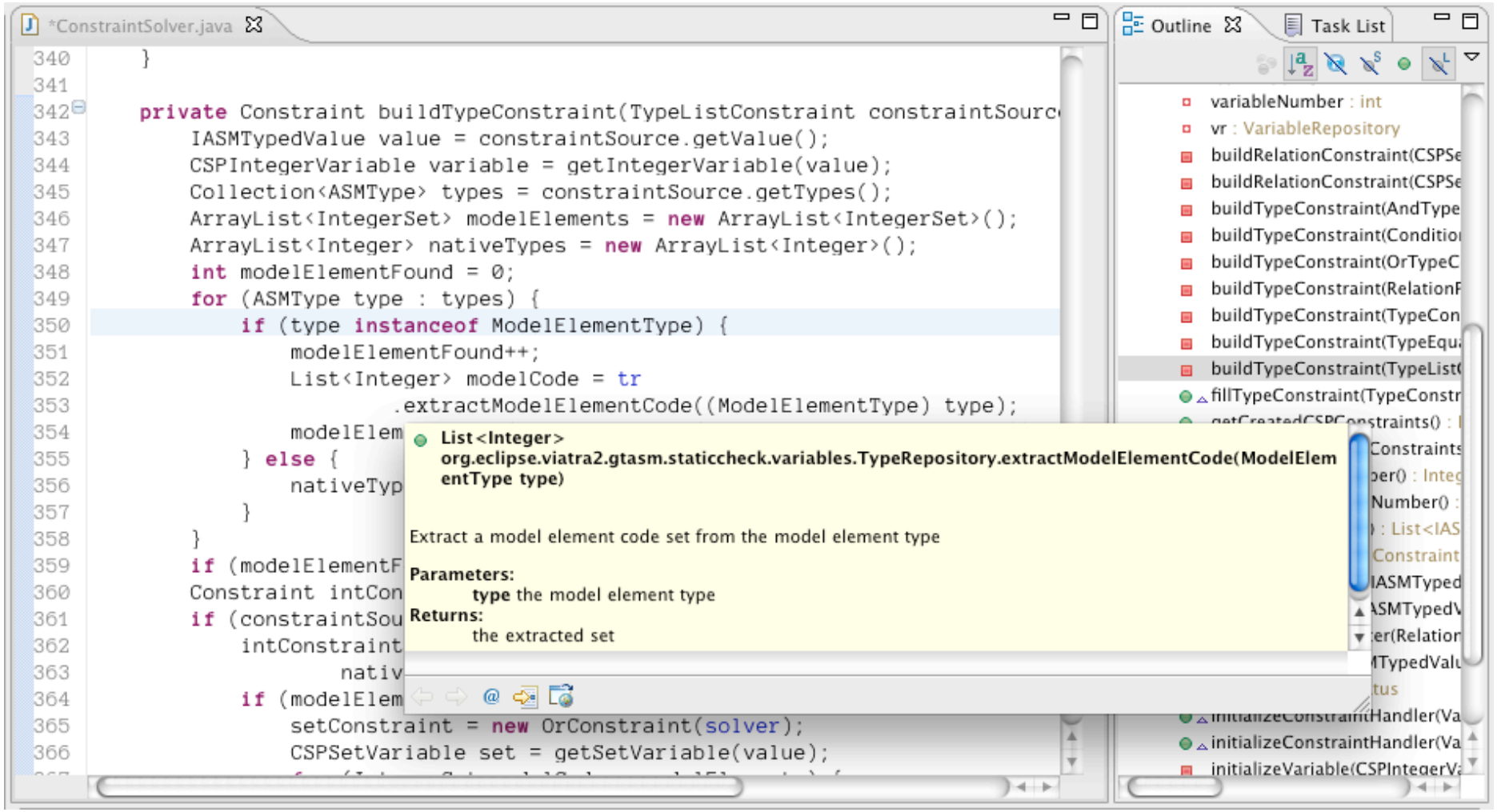
JDT services



JDT services: Error display and quick fix



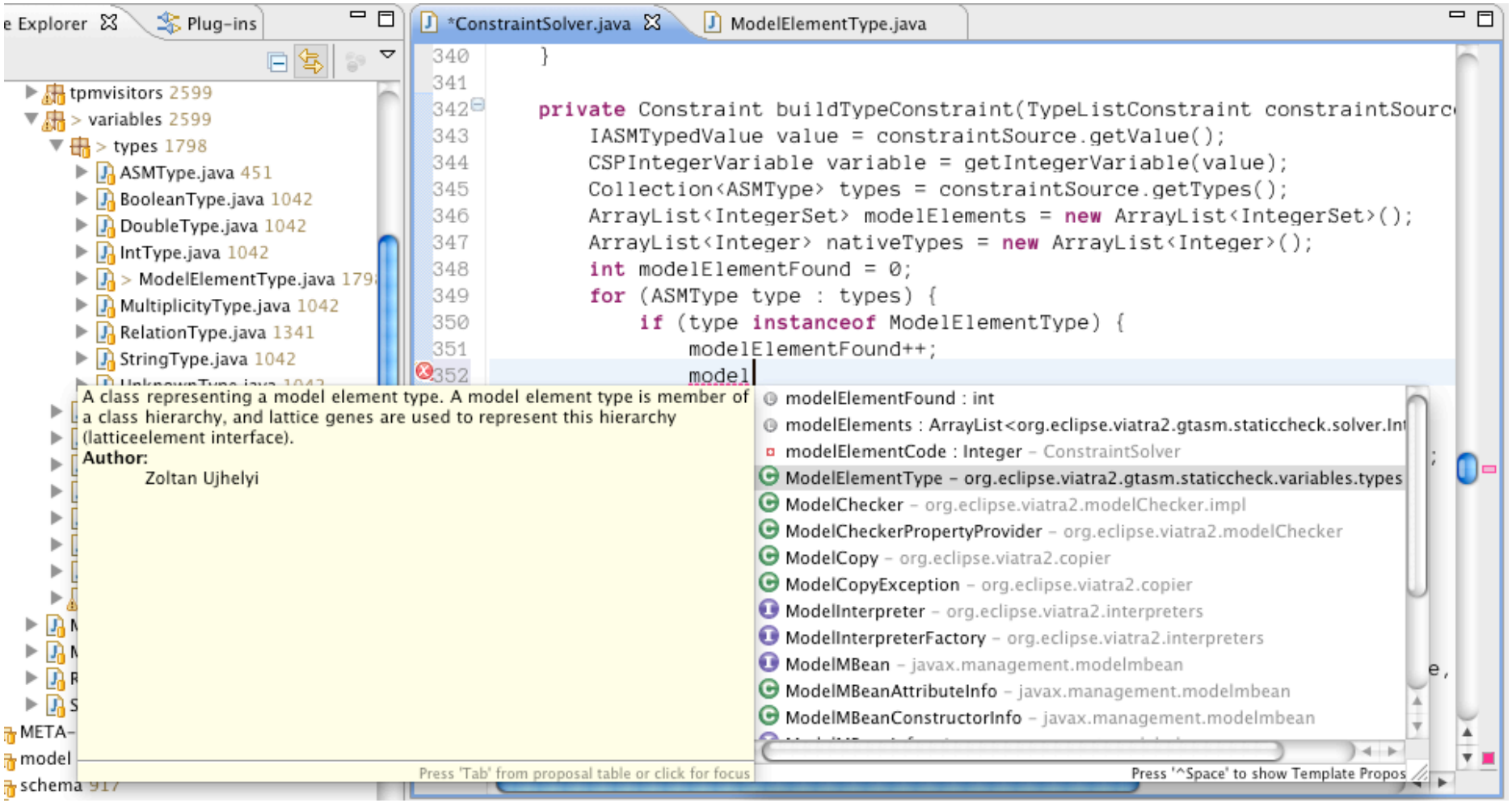
JDT services: Documentation



JDT services: Occurrence marker

```
341
342    private Constraint buildTypeConstraint(TypeListConstraint constraintSourc
343        IASMTypedValue value = constraintSource.getValue();
344        CSPIntegerVariable variable = getIntegerVariable(value);
345        Collection<ASMType> types = constraintSource.getTypes();
346        ArrayList<IntegerSet> modelElements = new ArrayList<IntegerSet>();
347        ArrayList<Integer> nativeTypes = new ArrayList<Integer>();
348        int modelElementFound = 0;
349        for (ASMType type : types) {
350            if (type instanceof ModelElementType) {
351                modelElementFound++;
352                List<Integer> modelCode = tr
353                        .extractModelElementCode((ModelElementType) type);
354                modelElements.add(new IntegerSet(setDomainSize, modelCode));
355            } else {
356                nativeTypes.add(typeMapping.get(type));
357            }
358        }
359        if (modelElementFound > 0) nativeTypes.add(modelElementCode);
360        Constraint intConstraint = null, setConstraint = null;
361        if (constraintSource.isPositive()) {
362            intConstraint = new IntConstantDomainConstraint(solver, variable,
363                    nativeTypes.toArray(new Integer[nativeTypes.size()]));
364            if (modelElementFound > 1) {
365                setConstraint = new OrConstraint(solver);
366                CSPSetVariable set = getSetVariable(value);
```

JDT services: Content assist



JDT services

- Program structure display views
- Navigation links between files
- Wizards
- Refactoring
- Incremental builder
- Run/Debug support
- **Extensibility**

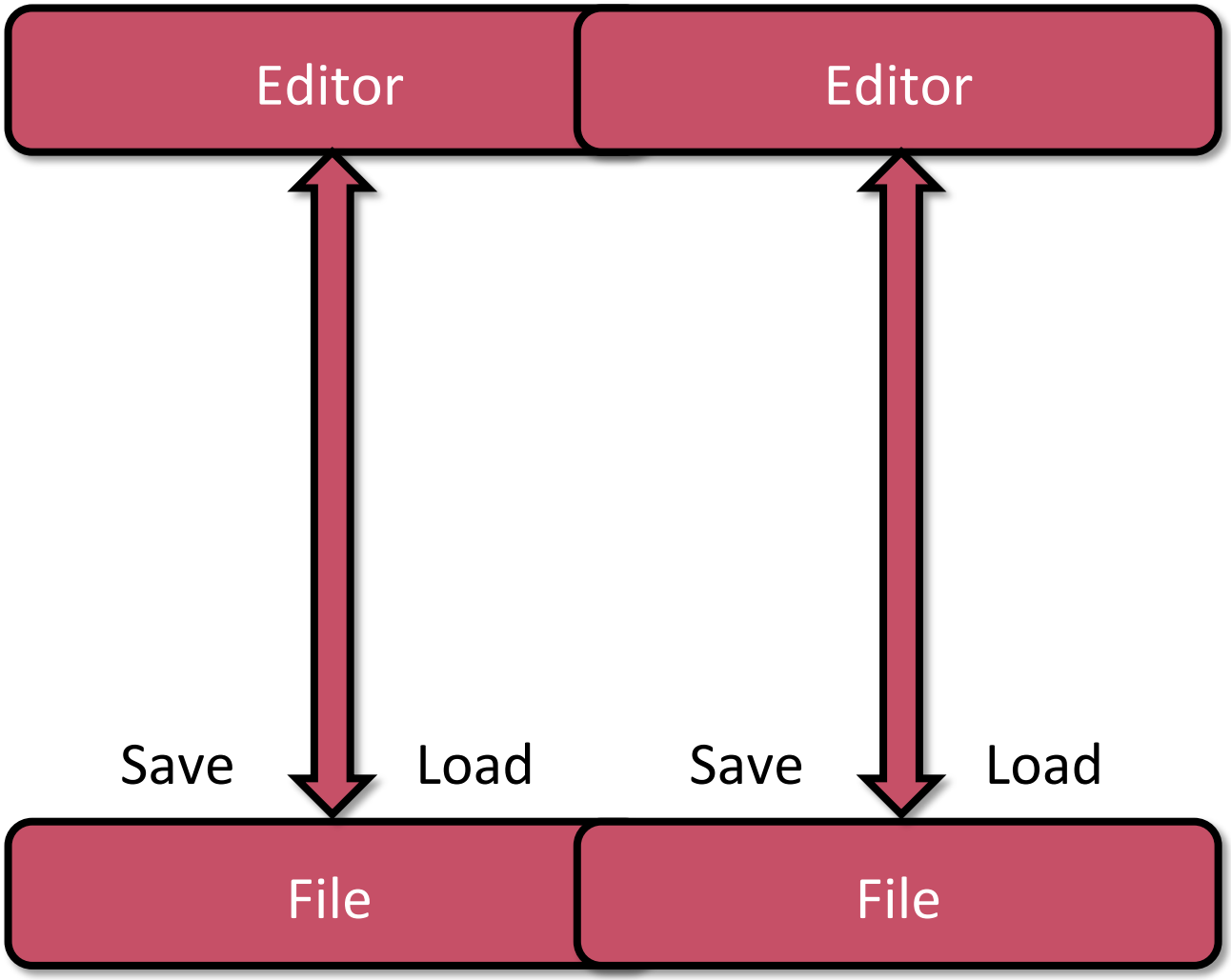
Textual editors in Eclipse

Architecture

- Simple text editor

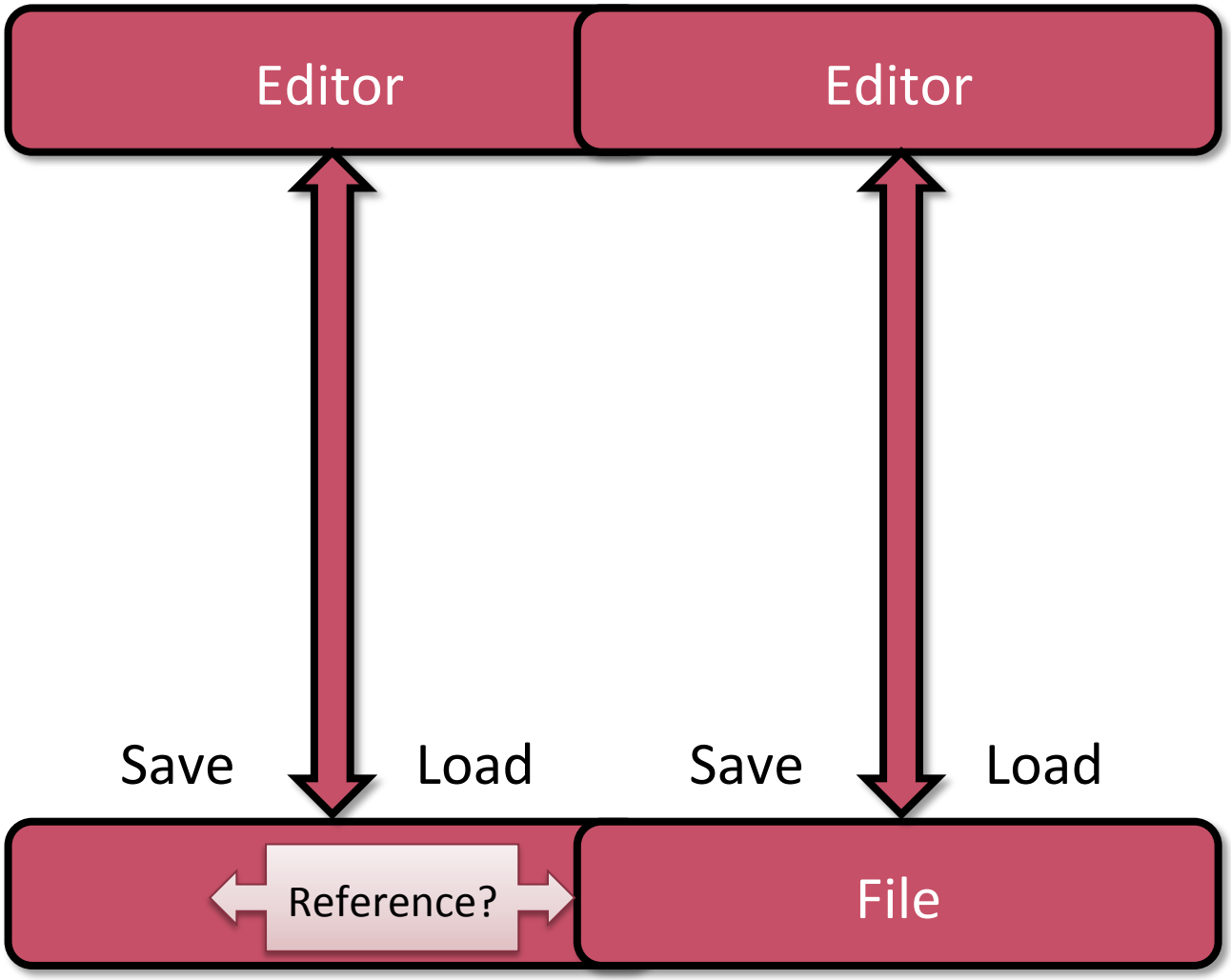
Architecture

- Simple text editor



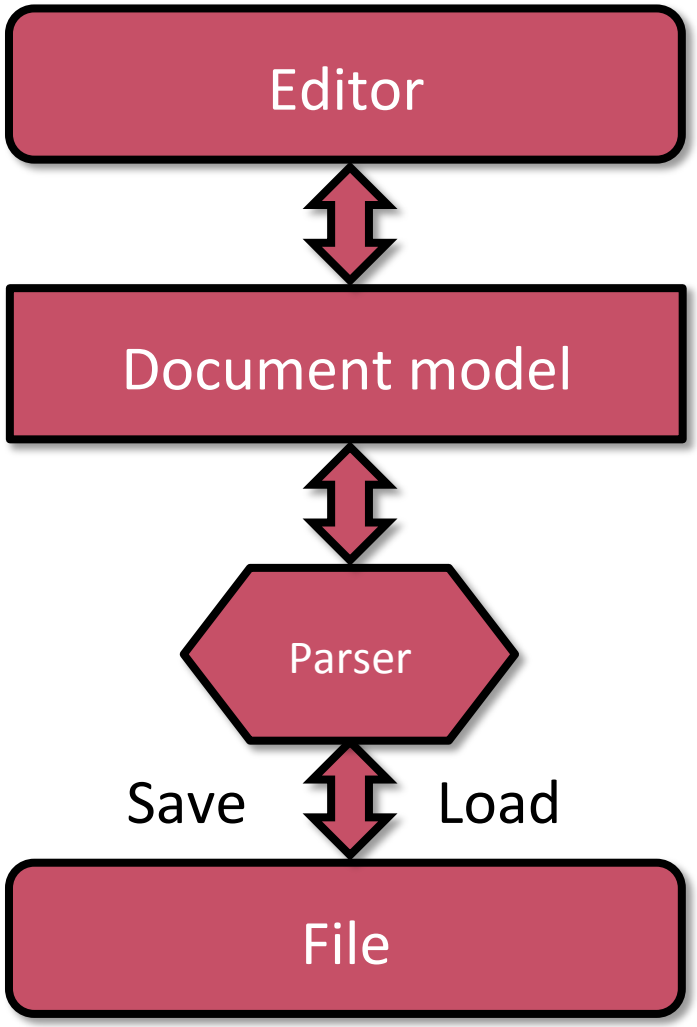
Architecture

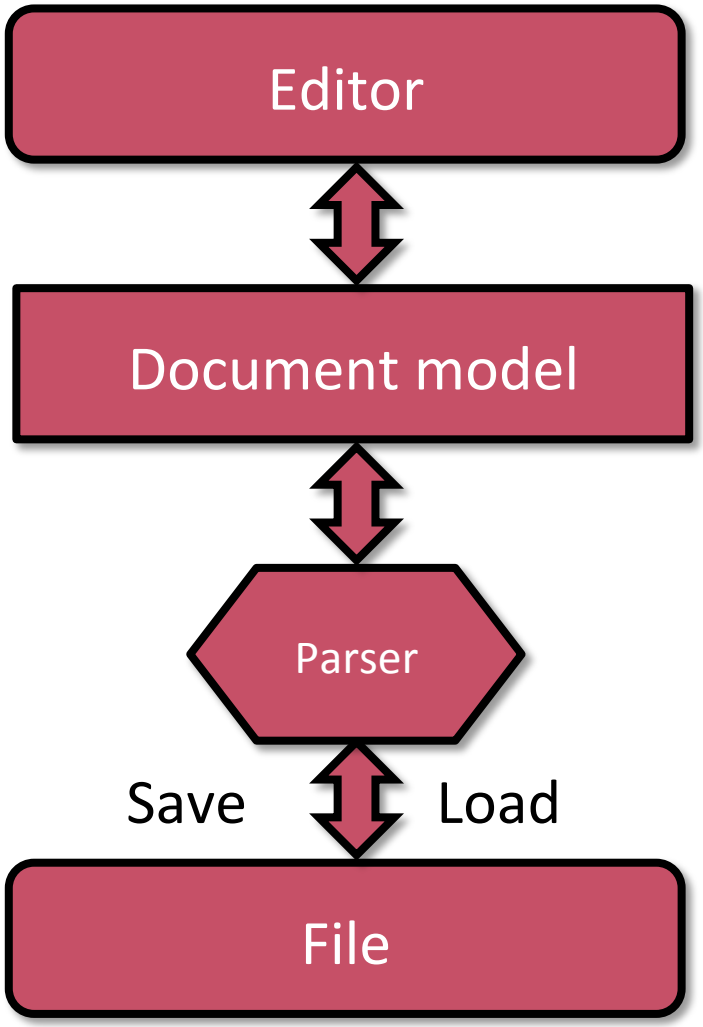
- Simple text editor



Architecture

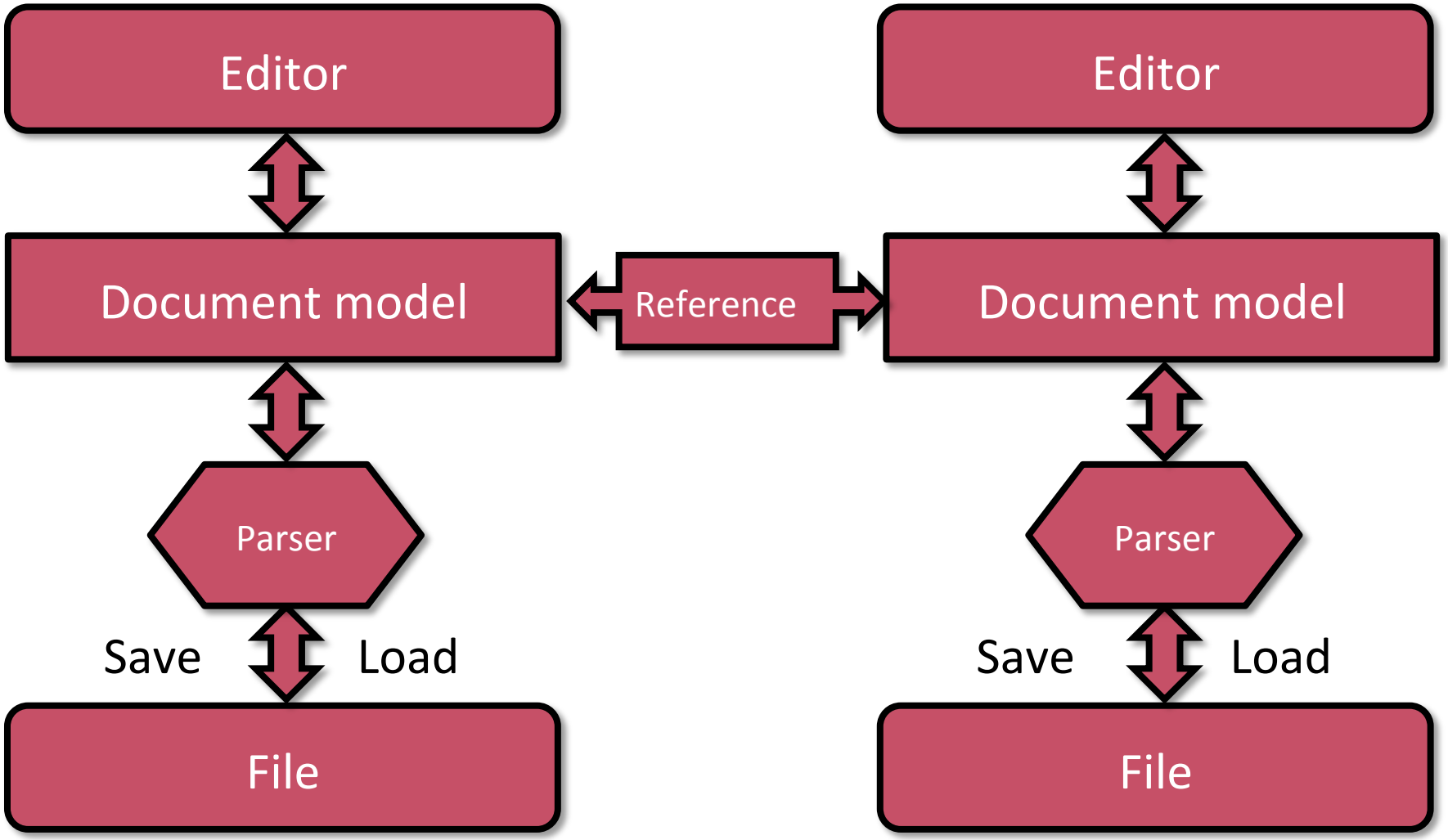
- Document model (MVC)





Architecture

- Document model (MVC)



Document model

- In-memory object representation for files
- What to use for?
  - Reference tracking
    - Inside files
    - Between files
  - Editor services rely on it
    - Content assist
    - Outline
    - …

Document model

- Structural information
  - Typesetting information
    - E.g. whitespaces
  - Comments
  - Abstract syntax
    - Strongly connected to raw text
    - Identifiable text fragments
- Produced by
  - Parsers

Document model

Document model

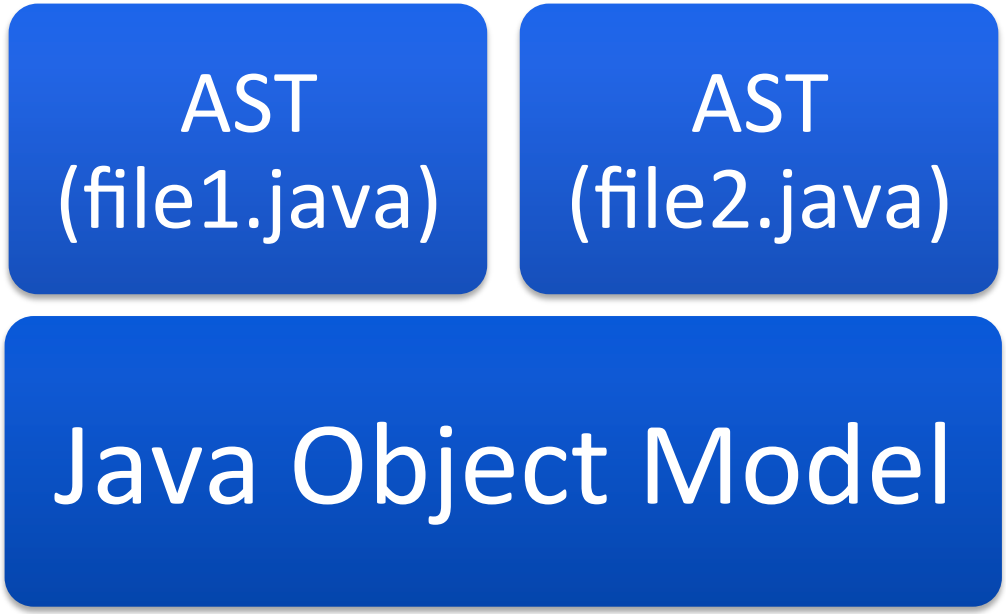
- Full details
  - 1:1 correspondence between model and text
  - Implemented by
    - AST (Abstract Syntax Tree)
    - ASG (Abstract Syntax Graph) - AST with resolved cross-edges
- Overview level
  - Model less detailed than file
  - Supports common overview of different files

Document model

- Full details
  - 1:1 correspondence between model and text
  - Implemented by
    - AST (Abstract Syntax Tree)
    - ASG (Abstract Syntax Graph) - AST with resolved cross-edges
- Overview level
  - Model less detailed than file
  - Supports common overview of different files

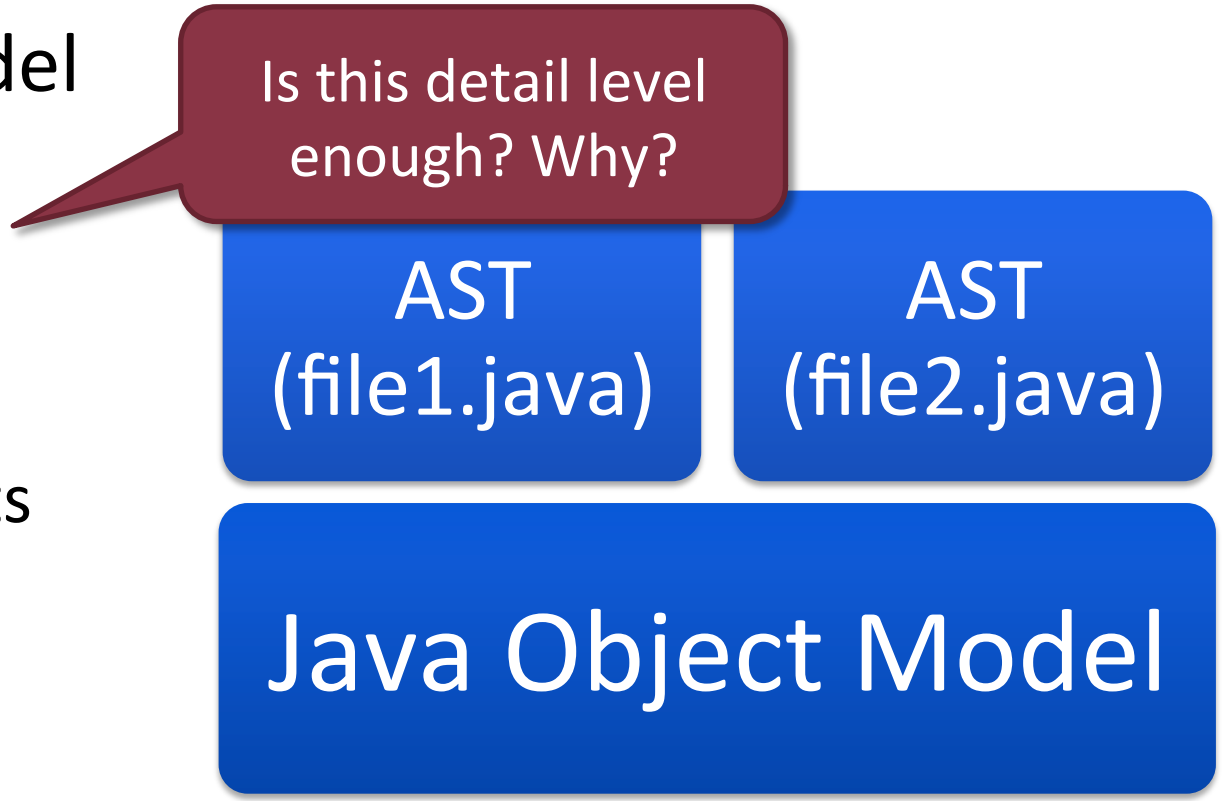
Example: JDT document model

- Java Object Model
  - Classes
  - Methods
  - Attributes
  - For Java projects
- AST
  - Full detail
  - On-demand model building



Example: JDT document model

- Java Object Model
  - Classes
  - Methods
  - Attributes
  - For Java projects
- AST
  - Full detail
  - On-demand model building



Producing a document model

- Using language-dependant parser
- Various important tasks
  - Reading
  - Tokenization
  - Reference resolution
  - Building AST/document models
- Error handling important
  - User want detailed error information
  - Error detection not enough

Markers

- Markers
  - Attaching meta-information to files
  - Supported by Platform Resources API
- Kinds
  - Error marker
  - Bookmark
  - Task marker
  - …
  - Custom markers

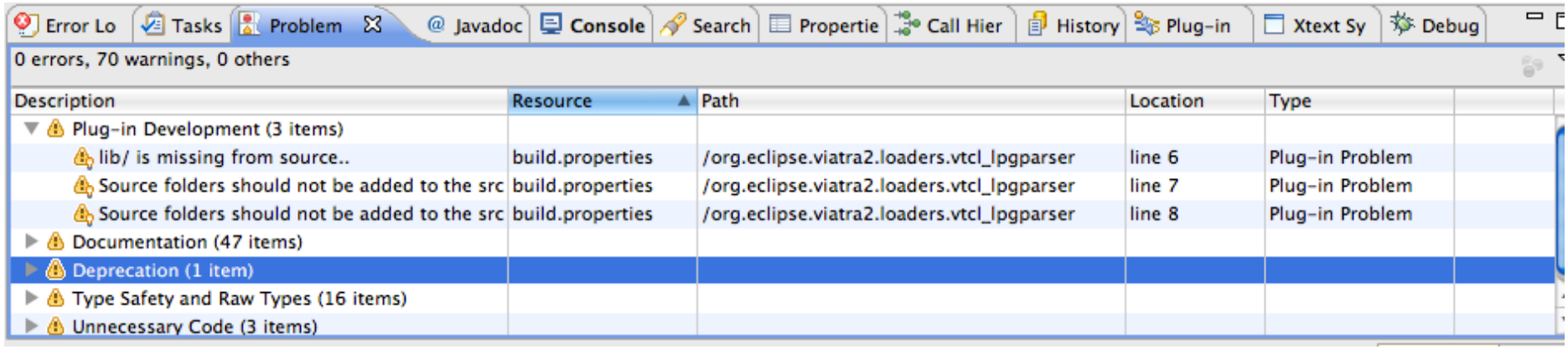
Markers

- Commonly used for feedback
  - Parsers
  - Analysis
  - User-initialized (e.g. bookmarks)
- Free data structure
  - Key-value pairs
  - No display method is defined on platform level
    - Marker users may use what information available
    - Ignores any unexpected data
  - Examples
    - Problems view
    - Bookmarks
    - Tasks

Markers

- Associated to files
  - More specifically to workspace resources
    - Not EMF Resources ☺
- Typical values
  - Location
    - File path
    - Line number/offset code
  - In case of problem markers
    - Severity
    - Message
    - Category

Problem markers



Quick fixes

- Quick fixes can be attached to problem markers
  - Possible file manipulations
  - Goal: fixing the problem
  - In most cases, it is done over the document model

Using the document model

- Content Assist
  - List the defined elements
  - Model specific filtering possible
- Outline support
  - Primitive filtered display of the model
- Navigation through references
  - Occurrence marking
  - Navigation links

Using the document model

- Refactoring
  - Easier to do on the model
    - Changes are easier to understand
  - Model supports serialization
- Pretty printing
  - Model serialization
  - Adding white spaces
    - Good pretty printing greatly helps understanding ☺

Editing textual languages

Processing textual languages

- Processing natural languages
- XML processing
- Regular expressions
- Grammar-based solution
  - Manually coded parsers
  - Generated parsers

Processing natural languages

- “Hard” problems to tackle
  - Context-information not always available
  - Not (well) defined semantics

Context

We gave the monkeys$_1$ the bananas$_2$

... because they$_1$ were hungry.

... because they$_2$ were ripe.

Forrás: http://specgram.com/CLIII.4/08.phlogiston.cartoon.zhe.html

Multiple parse trees based on context

Forrás: http://specgram.com/CLIII.4/08.phlogiston.cartoon.zhe.html

XML processing

- XML: standard textual format
- Schemes available
  - Roughly equivalent to metamodels
- Good tooling
  - Multiple serialization/deserialization techniques
    - DOM, SAX, etc.
  - Validation support
  - Query support

Problems with using XML

- Hard to write
  - End tags
  - Entity escaping (pl. `>` `&#8222;`)
- Hard to read
  - Strictly a tree structure
  - Reference resolution is not automatic

"It has been said that XML is like violence; if a little doesn't solve the problem, use more."
— Sarkos in reddit

The essence of XML is this: the problem it solves is not hard, and it does not solve the problem well.
— Phil Wadler, POPL 2003

http://quotes.cat-v.org/programming/

Regular expressions

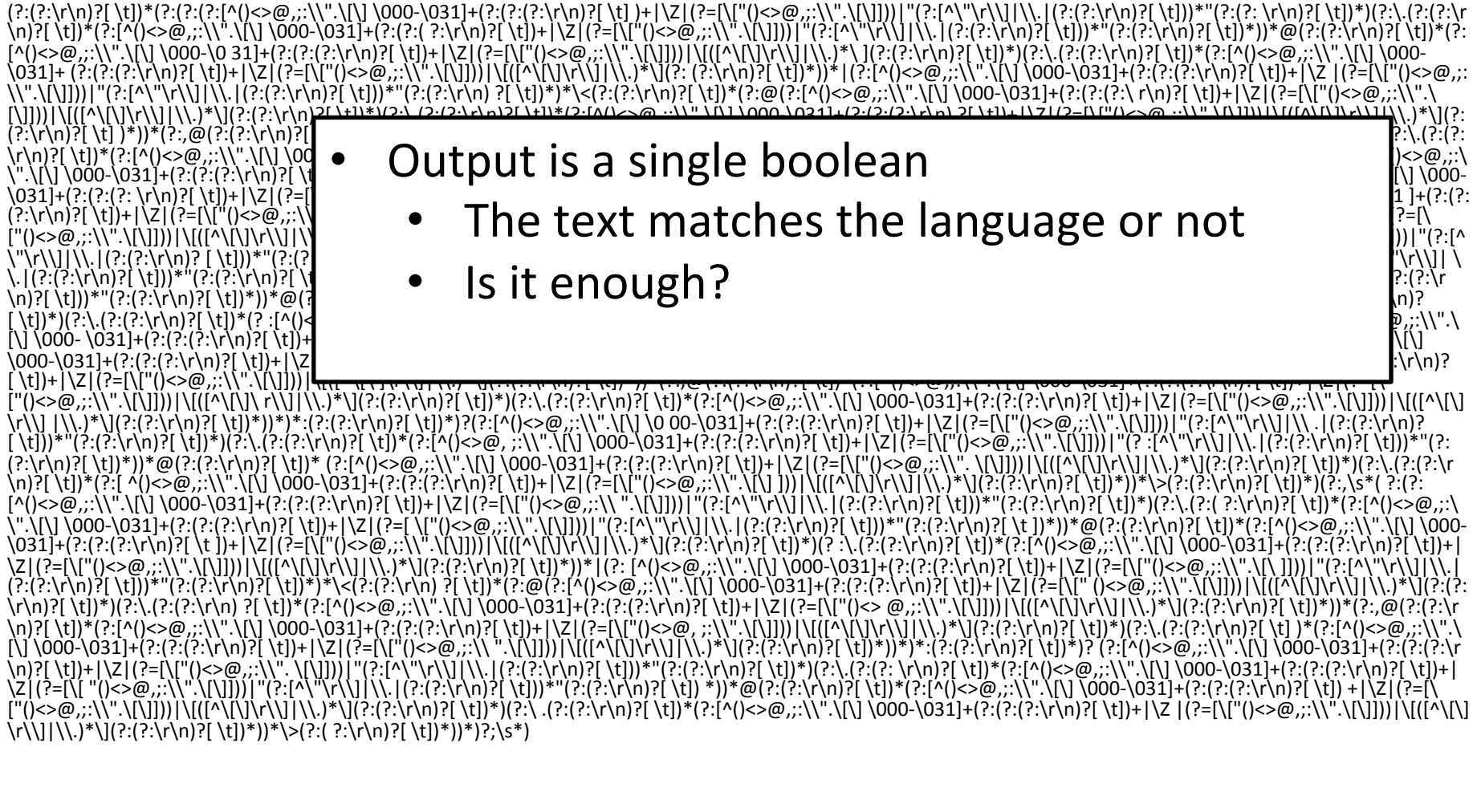
- Pattern matching in character sequences
  - Good support
    - Most programming languages
    - Basically common syntax (with differences)
  - Find/returns matches in the text
- Usable for textual DSL parsing?

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

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

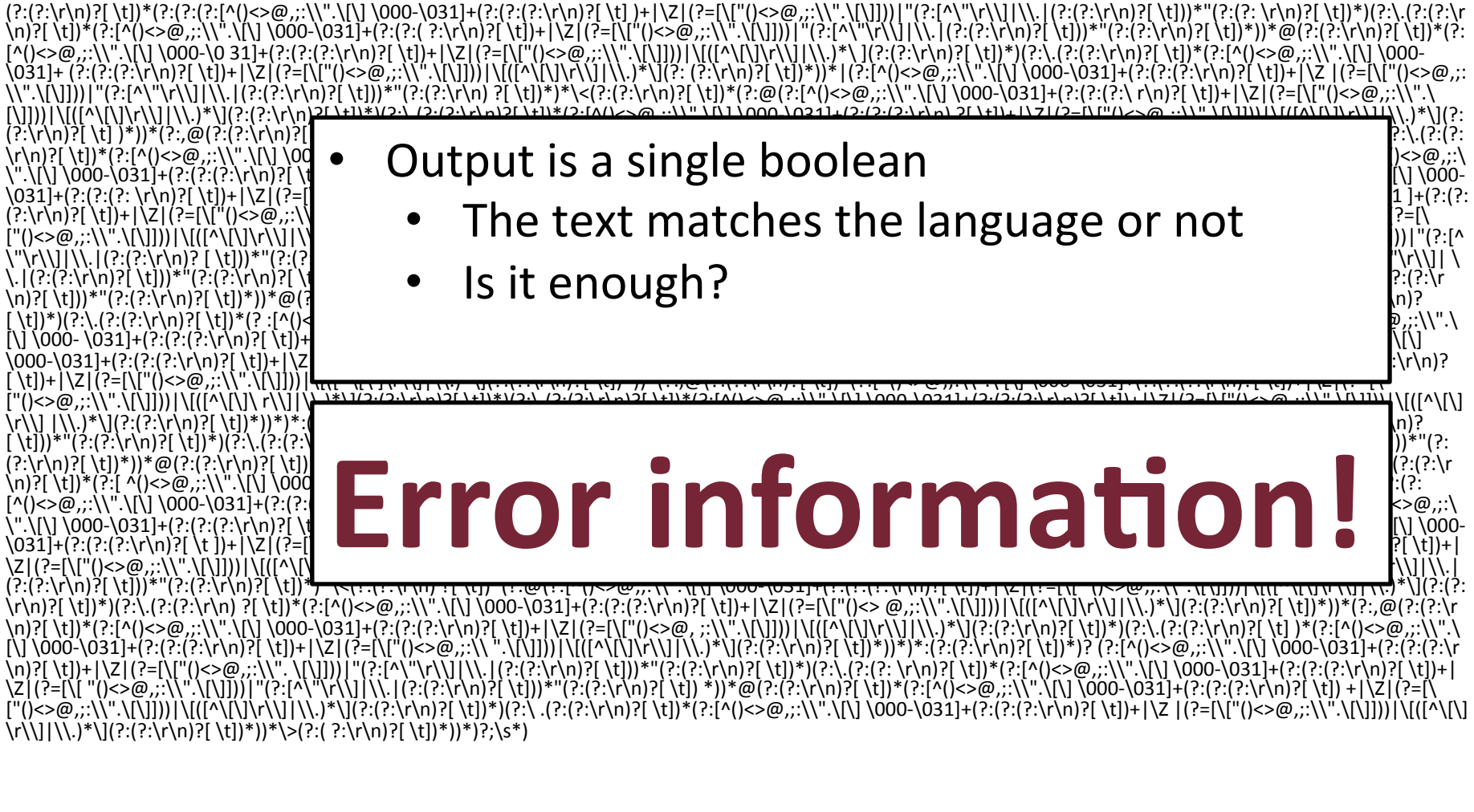
RegExp: Validation email addresses



- Output is a single boolean
  - The text matches the language or not
  - Is it enough?

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

RegExp: Validation email addresses



- Output is a single boolean
  - The text matches the language or not
  - Is it enough?

Error information!

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

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

Jamie Zawinski
http://regex.info/blog/2006-09-15/247

Parsers and grammars

Crash course on formal languages

Formal language

- Definitions (non-precise)
  - Language: set of possible sentences
  - Sentence: series of symbols
  - Alphabet: set of usable symbols

Examples using natural numbers

Examples using natural numbers

- Alphabet: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Language of natural numbers:
  - «0», «1», «2», «3», «4», «5», «6», «7», «8», «9»
- Odd numbers below 15:
  - «0», «2», «4», «6», «8», «10», «12», «14»
- Language of odd positive numbers:
  - «0», «2», «4», «6», «8», «10», «12», «14», «16», «18», «20», «22», «24», «26», «28», …

Examples using natural numbers

- Alphabet: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Language of natural numbers:
  - «0», «1», «2», «3», «4», «5», «6», «7», «8», «9»
- Odd numbers below 15:
  - «0», «2», «4», «6», «8», «10», «12», «14»
- Language of odd positive numbers:
  - «0», «2», «4», «6», «8», «10», «12», «14», «16», «18», «20», «22», «24», «26», «28», …

Examples using natural numbers

- Alphabet: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Language of natural numbers:
  - «0», «1», «2», «3», «4», «5», «6», «7», «8», «9»
- Odd numbers below 15:
  - «0», «2», «4», «6», «8», «10», «12», «14»
- Language of odd positive numbers:
  - «0», «2», «4», «6», «8», «10», «12», «14», «16», «18», «20», «22», «24», «26», «28», …

Examples using natural numbers

- Alphabet: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Language of natural numbers:
  - «0», «1», «2», «3», «4», «5», «6», «7», «8», «9»
- Odd numbers below 15:
  - «0», «2», «4», «6», «8», «10», «12», «14»
- Language of odd positive numbers:
  - «0», «2», «4», «6», «8», «10», «12», «14», «16», «18», «20», «22», «24», «26», «28», …

Specifying languages

- Enumerating valid sentences
  - Only works over finite languages
  - E.g. note language of odd numbers
- Defining a finite algorithm
  - Possible end result are the sentences of grammar
  - If a sentence **can** be a result of the algorithm -> sentence of the language

Example: Name enumerations

- Separate names with commas (‘,’)
  - Except before the last: there should be an ‘and’
  - E.g..: Zoltán, István and Dániel
- Name repeating allowed
  - E.g.: Zoltán, Zoltán and Dániel

Example: Name enumeration

Example: Name enumeration

- Algorithm
  1. Dániel, István, Zoltán are valid names
  2. A single name is a valid sentence
  3. A sentence followed by a comma and a name is also valid
  4. When finishing, if the sentence ends with “, «name»”, replace it with “ and «name»”

Example: Name enumeration

- Algorithm
  1. Dániel, István, Zoltán are valid names
  2. A single name is a valid sentence
  3. A sentence followed by a comma and a name is also valid
  4. When finishing, if the sentence ends with “, «name»”, replace it with “ and «name»”

«name» cannot appear in the sentences

Symbol types

- Terminal symbol
  - Can appear in valid sentences
  - Member of alphabet
- Non-terminal symbol
  - Must not appear in valid sentences
- Parsing algorithm
  - Replace non-terminals by symbol sequences (➔)
  - Algorithm ends
    - Only terminal symbols remain -> valid sentence
    - No more symbol replacement available
      - Either error
      - Or backtracking (if required)

Modified algorithm

- Alphabet
  - “Dániel”, “István”, “Zoltán”, “and”, “,”
- Non-terminal symbols
  - «Name», «Sentence»

① «Name» ➔ *Dániel* | *István* | *Zoltán*

② «Sentence» ➔ «Name»

③ «Sentence» ➔ «Sentence», «Name»

④ «Sentence» ends with “, «Name»” ➔ “and” «Name»

⑤ Start with non-terminal «Sentence»

Modified algorithm

- Alphabet
  - “Dániel”, “István”, “Zoltán”, “and”, “,”
- Non-terminal symbols
  - «Name», «Sentence»

① «Name» ➔ *Dániel* | *István* | *Zoltán*

② «Sentence» ➔ «Name»

③ «Sentence» ➔ «Sentence», «Name»

④ «Sentence» ends with “, «Name»” ➔ “and” «Name»

⑤ Start with non-terminal «Sentence»

This rule is complex

Categorizing formal languages

Categorizing languages

- Different grammars require different parsers
  - Serious complexity differences
  - Some grammars cannot be handled by generic parsers
- Chomsky-hierarchy of grammars
  - Based on rule complexity

Recursively enumerable languages (Type 0)

- Unrestricted rules
  - Equivalent by capabilities of Turing machines
- Generic parsing impossible
  - Undecidability issues
    - E.g. halting problem
  - Parsers for specific languages are still implementable!

Context-sensitive grammars (Type 1)

- Two (equivalent) definitions
  - Context-sensitive
    - $\alpha A \beta \rightarrow \alpha \gamma \beta$
    - α and β: context sequences
    - A: single non-terminal
    - γ: replacement sequence
  - Monotonic
    - RHS of rule contains more symbols then LHS
    - LHS of rule contains non-terminal rules

Context-sensitive grammars (Type 1)

- Two (equivalent) definitions
  - Context-sensitive
    - $\alpha A \beta \rightarrow \alpha \gamma \beta$
    - α and β: context sequences
    - A: single non-terminal
    - γ: replacement sequence
  - Monotonic
    - RHS of rule contains more symbols then LHS
    - LHS of rule contains non-terminal rules

Non-terminal A can be replaced by sequence γ

Context-sensitive grammars (Type 1)

- Generic context-sensitive parsing possible
  - Enumerate all valid sentences
    - Until given length
  - Decidable and finite!
    - If length is exceeded, it cannot be finished
  - Slow

Context-free grammars (Type 2)

- Rules
  - $A \rightarrow \gamma$
    - LHS: Single non-terminal symbol
    - RHS: sequence of terminal and non-terminal symbols
  - Context sensitive grammar with empty contexts

Context-free grammars (Type 2)

- Parsing
  - Multiple approaches available
  - Well researched benefits and disadvantages
  - See later

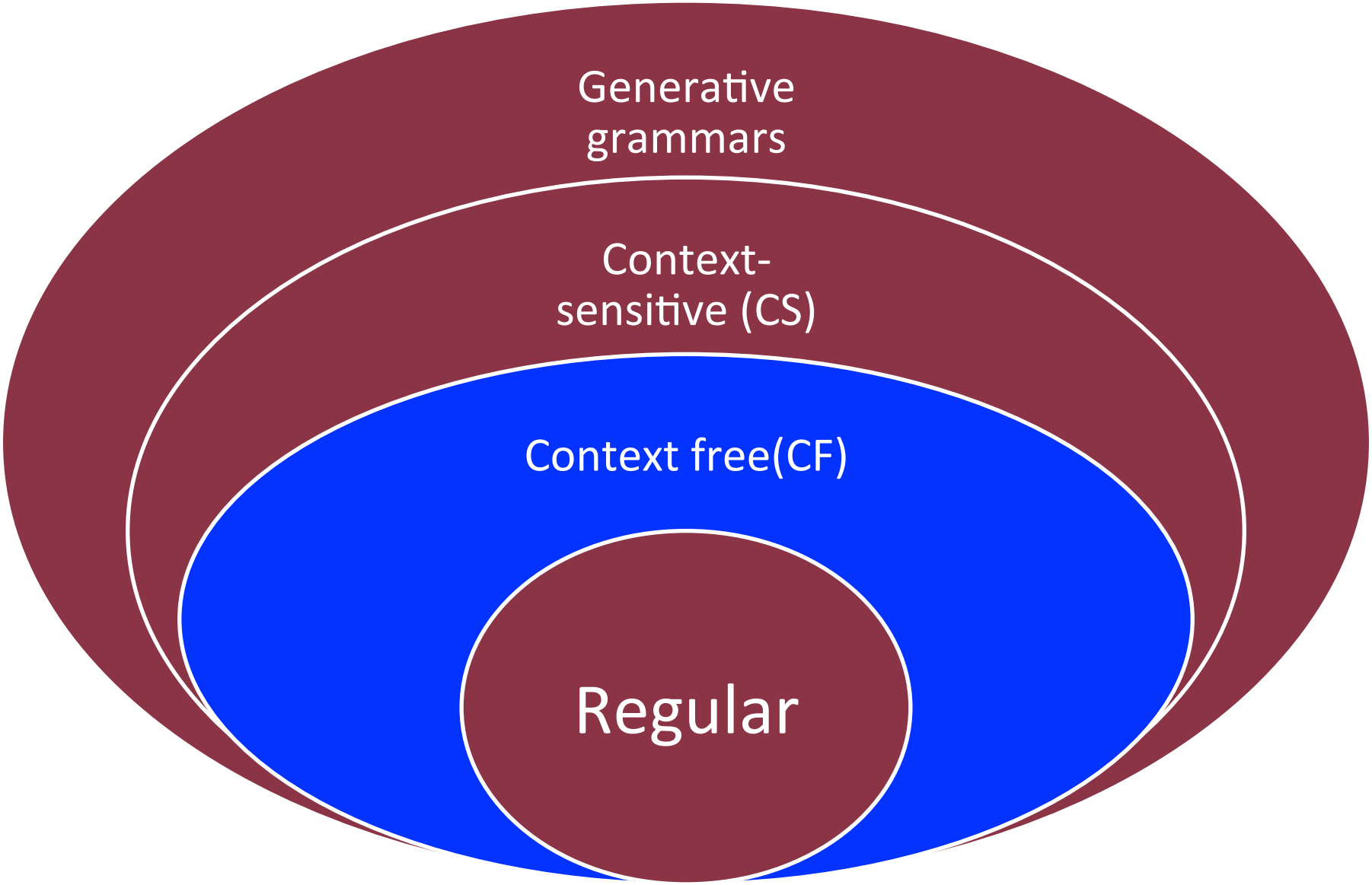
Regular grammars (Type 3)

- Two types of rules:
  - A→α
  - A→αB
- Equivalent to regular expressions
  - Implementations consists of some expressions

Regular grammars (Type 3)

- Parser: finite automaton
- Works for simple problems, but
  - Nesting problematic
    - See pumping lemma for regular languages
  - Required for
    - Parentheses
    - Nested blocks (e.g. expression blocks)
  - Typically required for common DSLs

Chomsky hierarchy for formal grammars



Categorizing languages

- So far:
  - Categorizing grammars
- Category of language:
  - Consider all possible grammars
  - The most strict grammar of the language determines the language complexity

Example: Name enumeration

- Alphabet
  - “Dániel”, “István”, “Zoltán”, “and”, “,”
- Non-terminal symbols
  - «Name», «Sentence»

① «Name» ➔ *Dániel* | *István* | *Zoltán*

② «Sentence» ➔ «Name»

③ «Sentence» ➔ «Sentence», «Name»

④ «Sentence» ends with “, «Name»” ➔ “and” «Name»

⑤ Start with non-terminal «Sentence»

Example: Name enumeration

- Alphabet
  - “Dániel”, “István”, “Zoltán”, “and”, “,”
- Non-terminal symbols
  - «Name», «Sentence»

① «Name» ➔ *Dániel* | *István* | *Zoltán*

② «Sentence» ➔ «Name»

③ «Sentence» ➔ «Sentence» “,” «Name»

④ «Sentence» ends with “,” «Name» ➔ “and” «Name»

⑤ Start with non-terminal «Sentence»

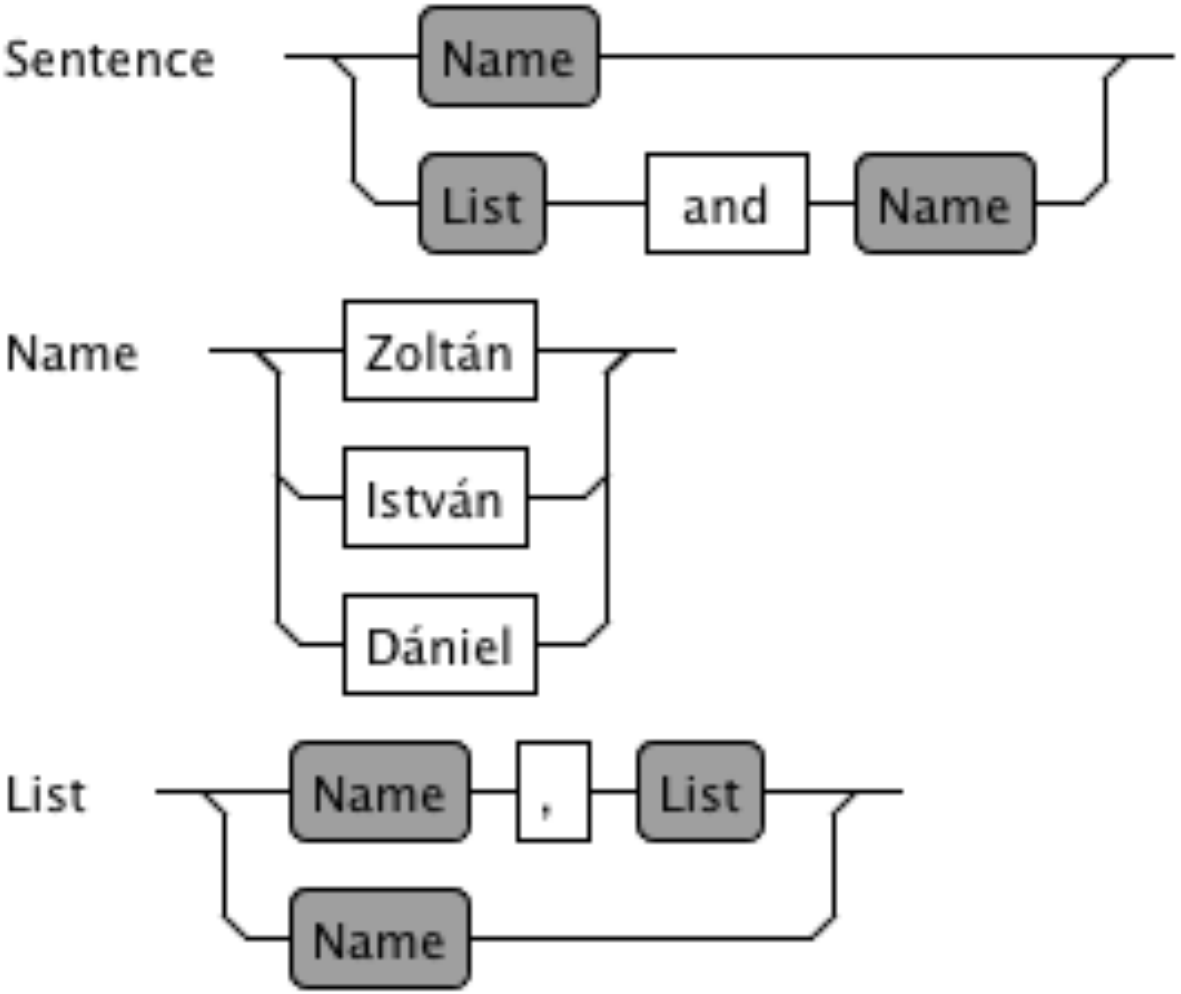
Rule 4: Type 0

Example: CF grammar for names

- Alphabet
  - “Dániel”, “István”, “Zoltán”, “and”, “,”
- Non-terminal symbols
  - «Name», «Sentence»

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

Railroad diagram



Parsing CF languages

Applying grammars

- Select applicable rule
  - Indeterministic selection process
    - Different strategies for different algorithms
    - Might be deterministic for selected algorithms
  - Apply selected rule
  - Repeat until
    - Input sentence is found,
    - Or no rule can be selected

Example: Derivation

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

Example: Derivation

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

«Sentence»

- Apply «Sentence» ->«List» and «Name»

«List» and «Name»

- Apply «List» -> «List», «Name»

«List», «Name» and «Name»

- Apply «List» -> «Name»

«Name», «Name» and «Name»

- Apply 3 times: «Name» -> Zoltán

Zoltán, Zoltán and Zoltán

Example: Derivation

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

«Sentence»

- Apply «Sentence» ->«List» and «Name»

«List» and «Name»

- Apply «List» -> «List», «Name»

«List», «Name» and «Name»

- Apply «List» -> «Name»

«Name», «Name» and «Name»

- Apply 3 times: «Name» -> Zoltán

Zoltán, Zoltán and Zoltán

Example: Derivation

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

«Sentence»

- Apply «Sentence» ->«List» and «Name»

«List» and «Name»

- Apply «List» -> «List», «Name»

«List», «Name» and «Name»

- Apply «List» -> «Name»

«Name», «Name» and «Name»

- Apply 3 times: «Name» -> Zoltán

Zoltán, Zoltán and Zoltán

Example: Derivation

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

«Sentence»

- Apply «Sentence» ->«List» and «Name»

«List» and «Name»

- Apply «List» -> «List», «Name»

«List», «Name» and «Name»

- Apply «List» -> «Name»

«Name», «Name» and «Name»

- Apply 3 times: «Name» -> Zoltán

Zoltán, Zoltán and Zoltán

Example: Derivation

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

«Sentence»

- Apply «Sentence» ->«List» and «Name»

«List» and «Name»

- Apply «List» -> «List», «Name»

«List», «Name» and «Name»

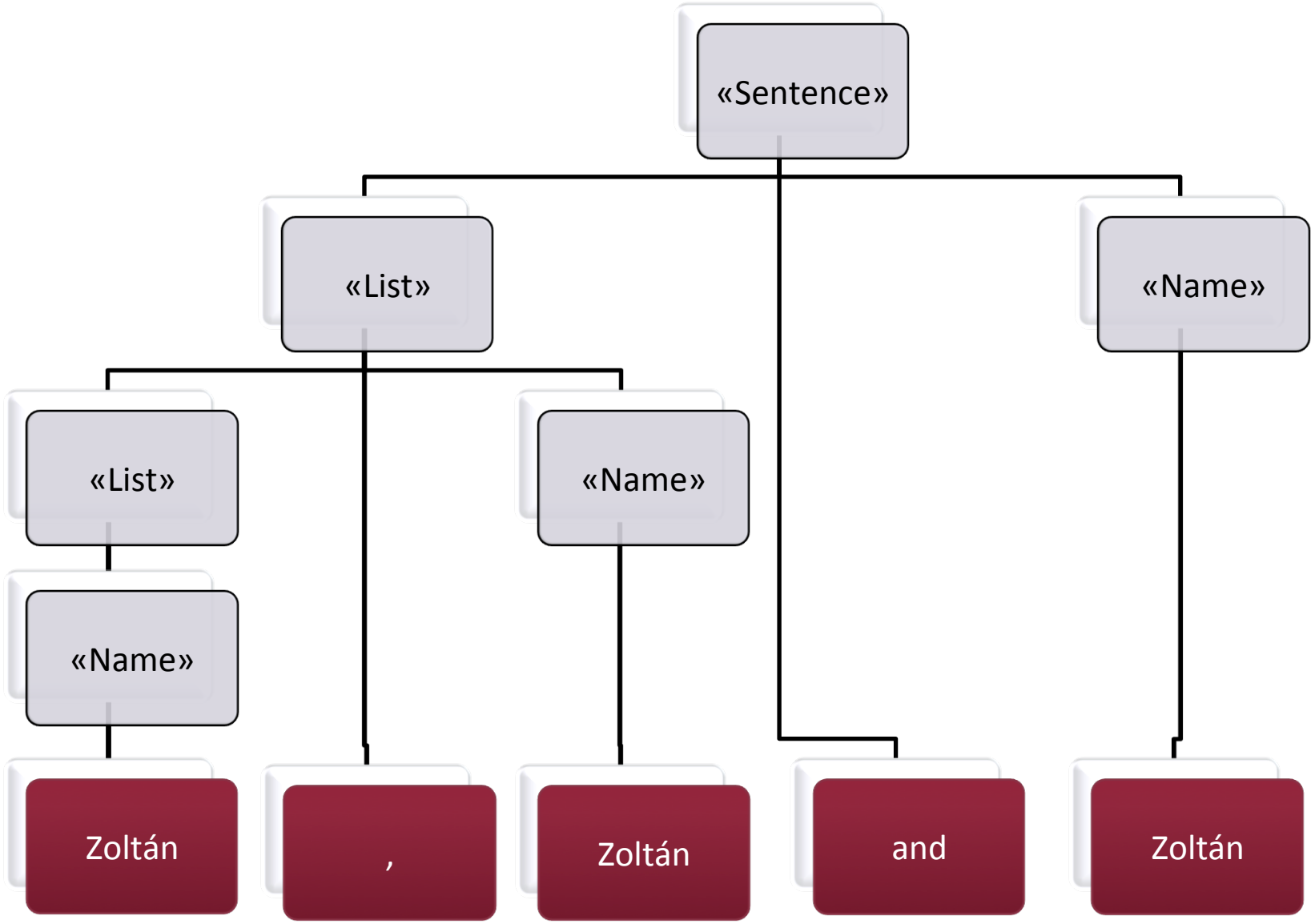
- Apply «List» -> «Name»

«Name», «Name» and «Name»

- Apply 3 times: «Name» -> Zoltán

Zoltán, Zoltán and Zoltán

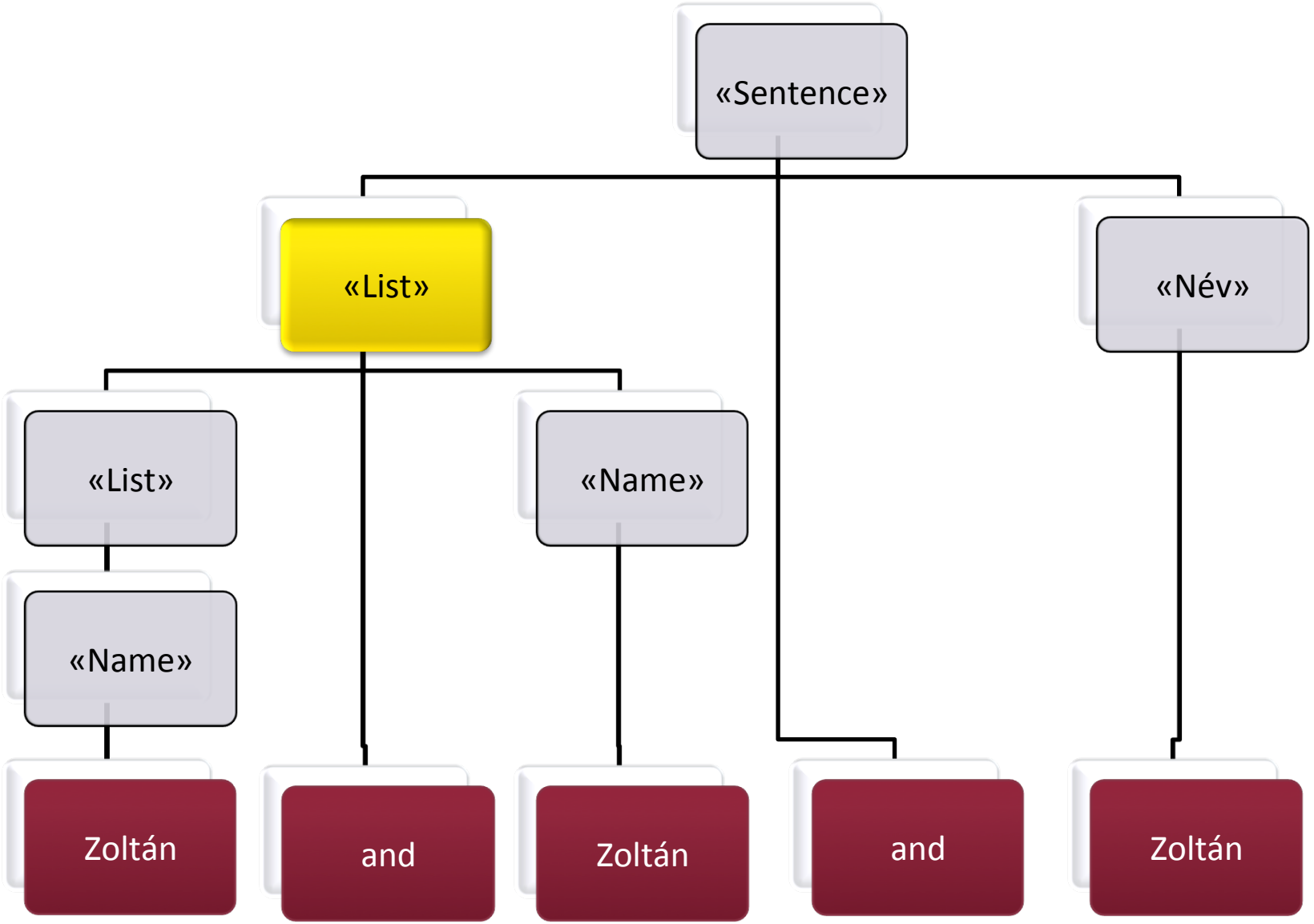
Derivation Tree



Parsing

- Goal:
  - Creation of derivation tree
  - What do we gain with a derivation tree?

Error Detection



Parsing

- Goal:
  - Creation of derivation tree
  - What do we gain with a derivation tree?
- Different approaches
  - Non-directed methods
  - Directed methods
  - Search-based techniques
  - **Top-down approaches**
  - **Bottom-up approaches**

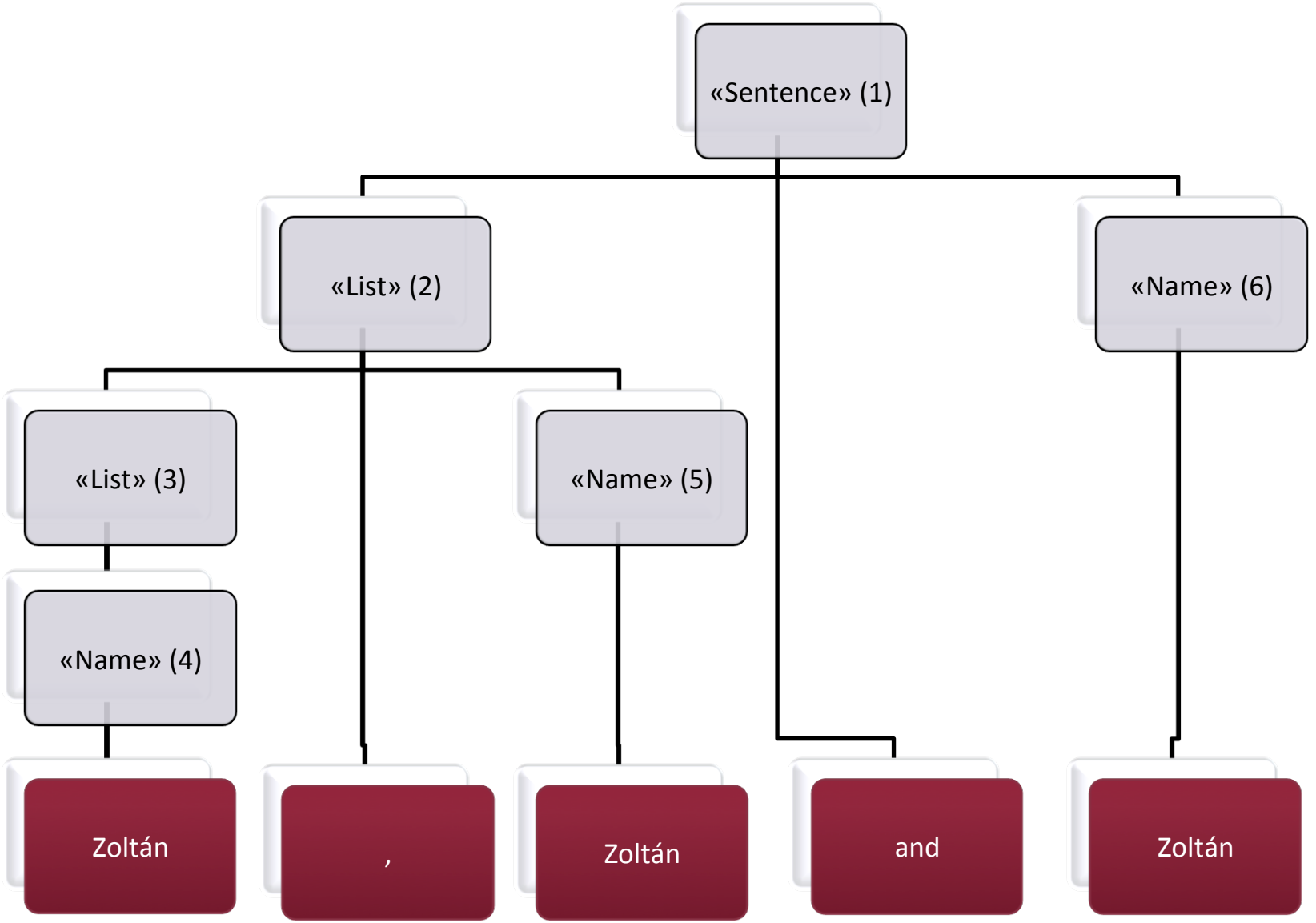
Top-down parsing

- Start: Sentence symbol
- Goal: execute rule applications and reach solution
- In case of required decisions
  - Predict and match
  - Backtracking if required

LL(k) parsers

- Properties
  - Left-to-right (reading order)
  - Leftmost derivation
  - k character look-ahead

LL(k) derivation



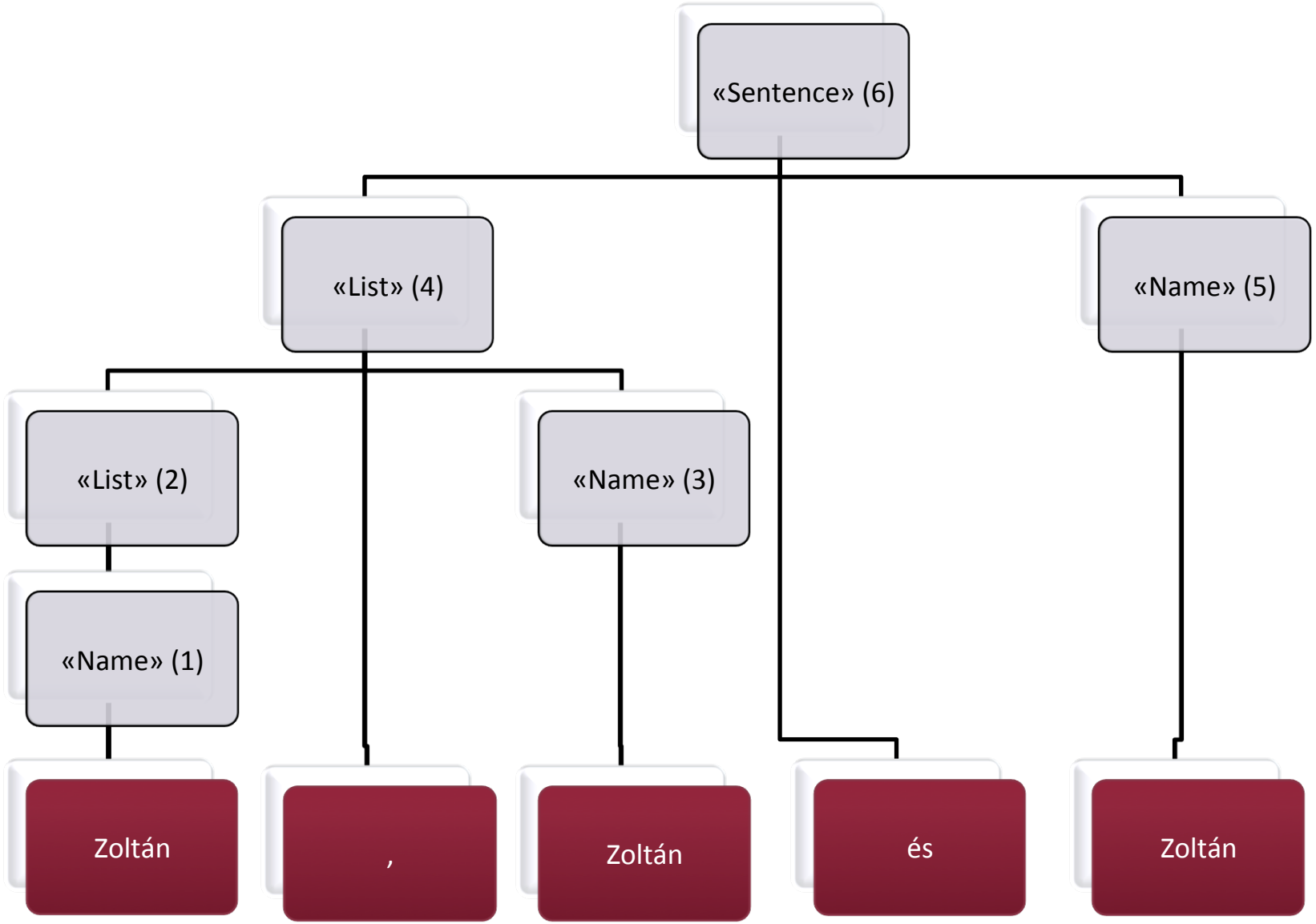
Bottom-up parsing

- Start: Sentence
- Goal: Reach sentence symbol
- Inverse rule applications
  - RHS matched to text

LR(k) parsers

- Properties
  - Left-to-right (reading order)
  - Rightmost derivation ()
  - k character look-ahead

LR(k) derivation



Choosing algorithm

- $LL(k) \subset LR(k)$
- $LR(k) \subset CF$

- Ideas
  - LL(k)
    - Simpler algorithm
    - Similar execution to reading -> more understandable
  - LR(k)
    - Higher expressive power
    - Less memory consumed

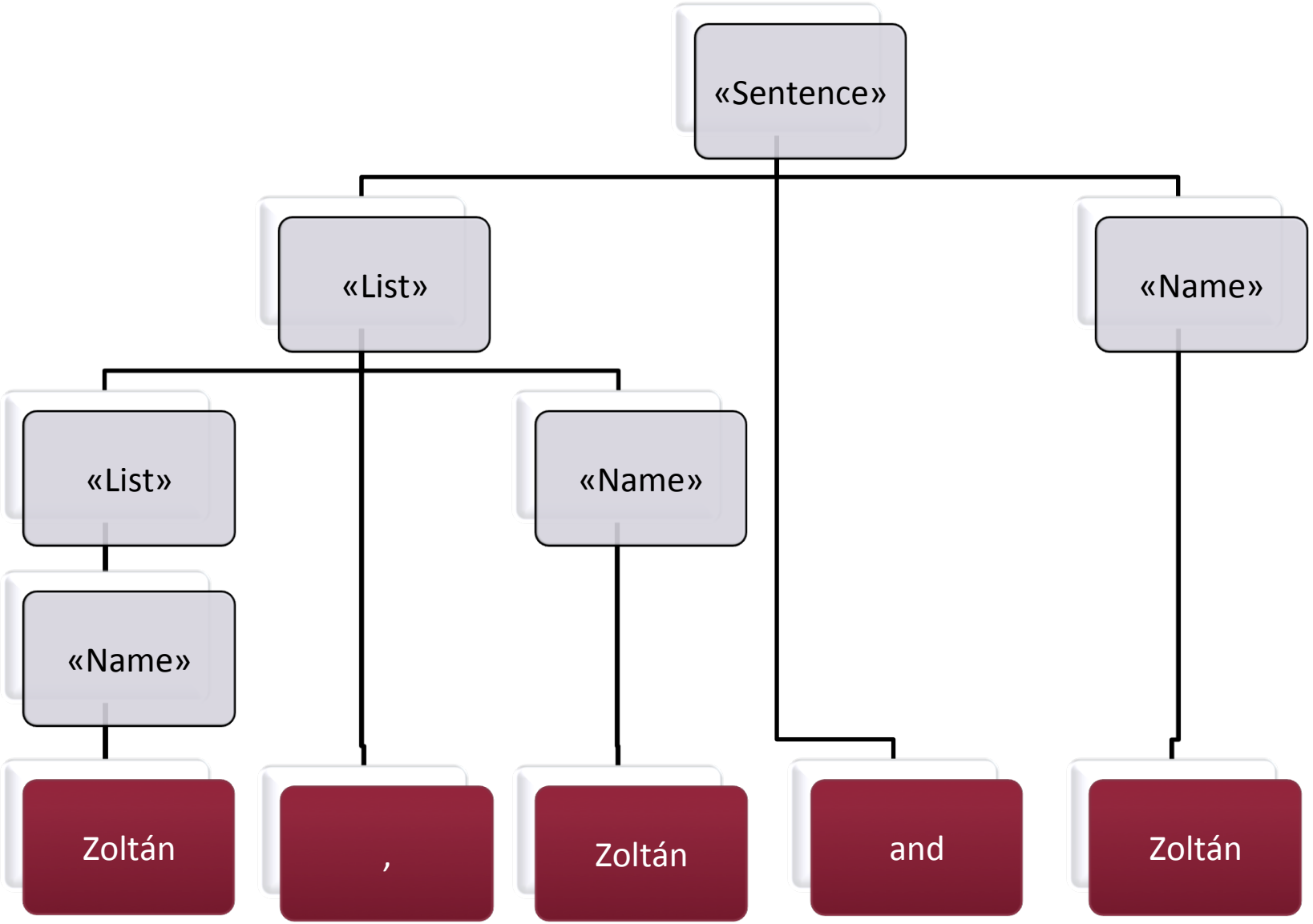
Parsers in practice

Overview

Overview

- So far
  - Grammars
  - Parsers
- What is still missing?
  - Input character stream handling
  - Variable handling
  - High-level analysis

Input



Input

Real input:

'Z' 'o' 'l' 't' 'á' 'n' ' ' ',' 'Z'
'o' 'l' 't' 'á' 'n' ' ' 'é' 's' ' '
'Z' 'o' 'l' 't' 'á' 'n'

Zoltán , Zoltán and Zoltán

Input handling

Input handling

- Input:
  - Character stream
- Parser input
  - Higher level tokens
    - «Name», ‘,’ or “and”
  - **Lexer**s connect the gap
- Why is this indirection useful?
  - Error handling
  - Performance
  - Problem decomposition

Lexing

- Goal:
  - Tokenizing character stream
  - Similar to parsing problem
    - But much simpler – typically regular expressions are enough
    - Identifying words and tokens
    - Removing comments (and possible white spaces)
  - Simplifies the task of the parser

Variable handling

```
a=3;
System.out.println(a);
```

- Variable
  - Runtime
    - Value stored/retrieved from an address
  - Editing/parsing time
    - Crossreference
    - To another part of the AST

Variable handling

- *Variable reference*
  - Usage of a variable
- *Variable declaration*
  - Definition of a variable
- Unique name (well-formedness constraints)
  - Variable declarations must be identifiable
    - Not necessary unique!
  - Extra phase after parsing

Variable handling

- Parser provides
  - Variable name is syntactically valid
- Resolver needs to check
  - The existence of a corresponding definition
    - Scoping problem
  - Uniqueness of definition

Scoping problem

```
private int value;

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

What does ‘value’ refer to?

Scoping problem

- Solution
  - Resolver defines approach
- Possible approaches
  - Most specific declaration
  - Error in case of conflicts
  - Qualified names
  - …

Parsing process