An Introduction to Parsing

Dániel Varró

Model Driven Software Development

Lecture 5

Using contents from Guido Wachsmuth
(Compiler Construction at TU Delft)

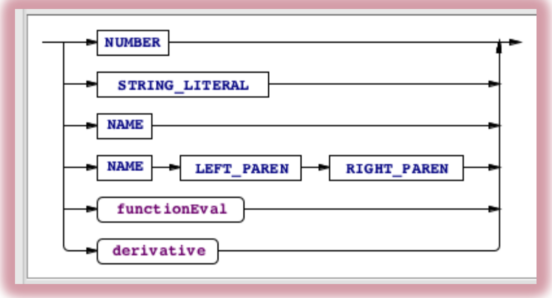
Budapest University of Technology and Economics
Department of Measurement and Information Systems

Looking Inside Advanced IDEs

From parsers to development tools

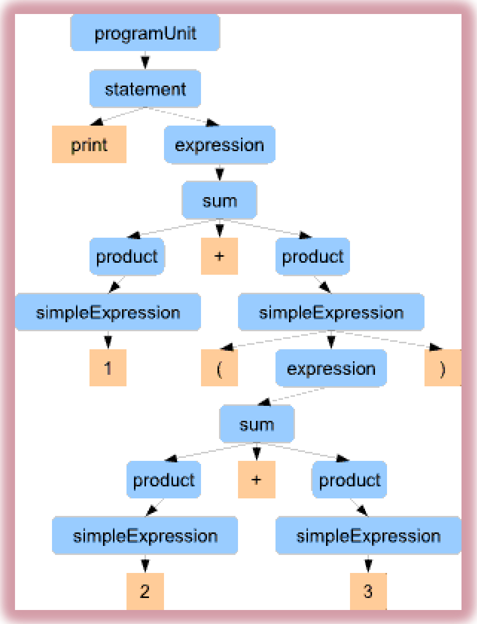
Parsers: The Traditional Setup

Grammar



Source code of program

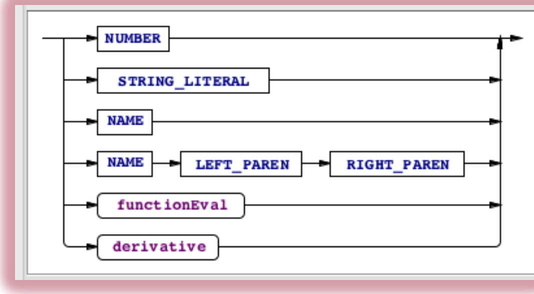
Parsing
(lexer + parser)

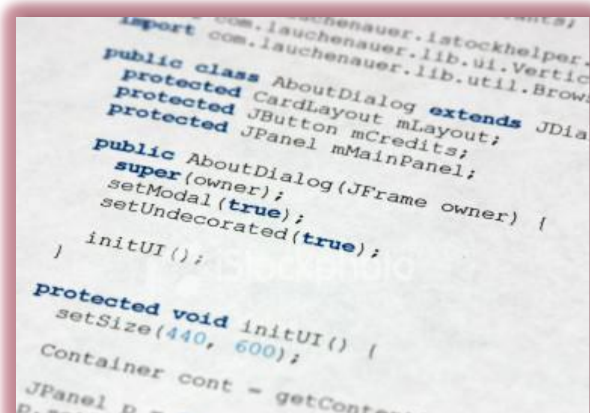


Abstract
syntax tree (AST)

Parsers in Software Engineering Practice

Grammar

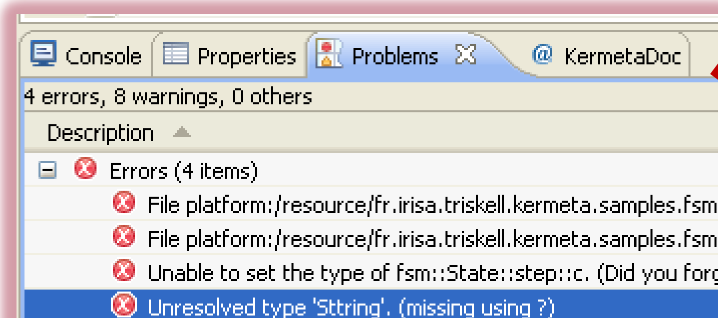


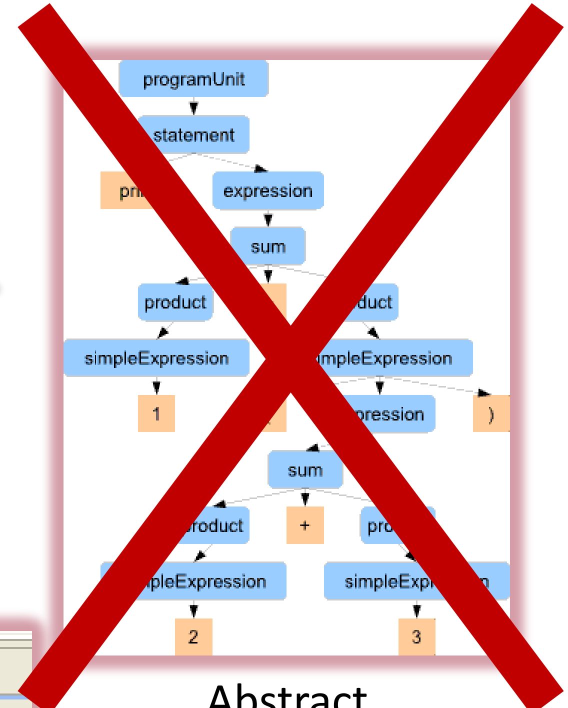


Source code of program

Parsing (lexer + parser)

Error report

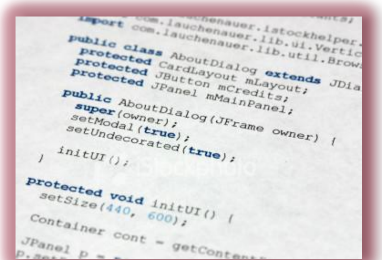




Abstract syntax tree (AST)

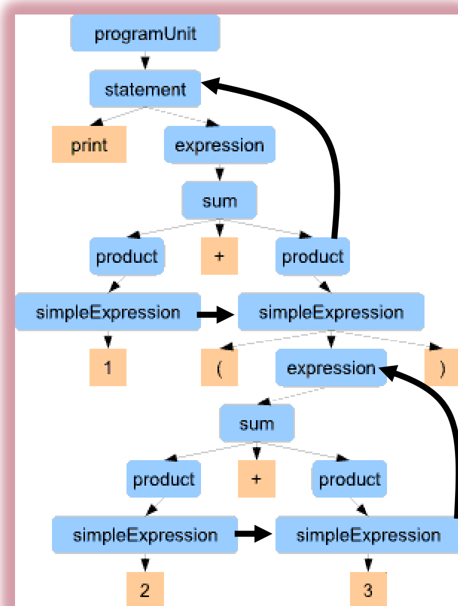
Error recovery parsing

View Generation + Program Analysis



Source code of program

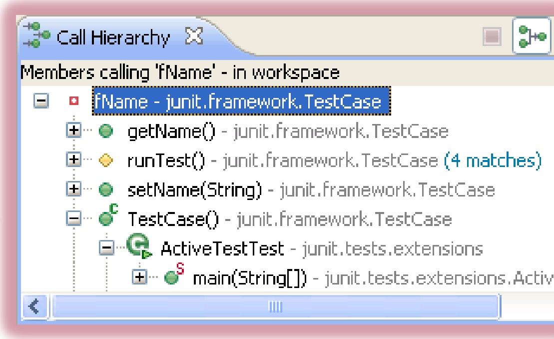
Parsing



AST: Abstract syntax tree

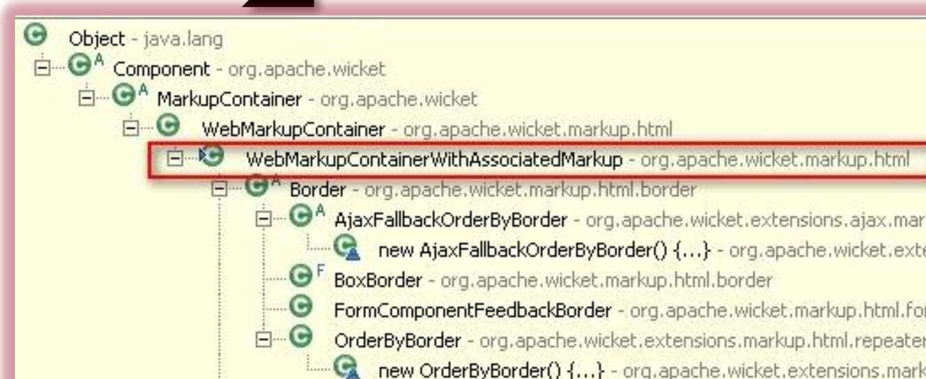
„Visitor”

Call graph



„Visitor”

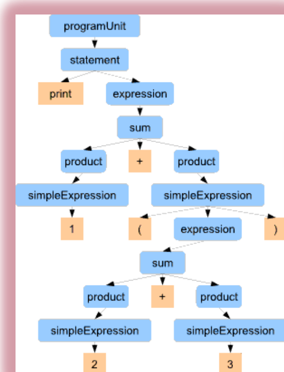
Type hierarchy



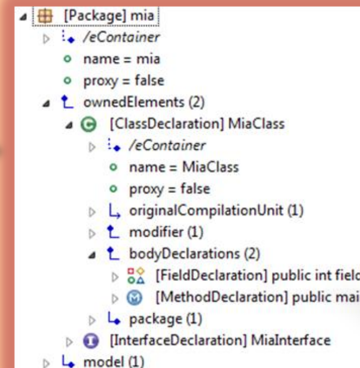
ASTs vs DOMs

Source code of program

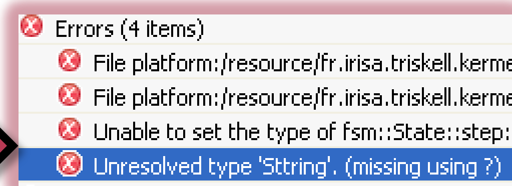
AST: Abstract syntax tree



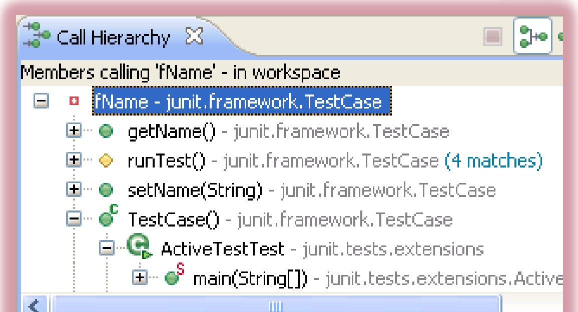
DOM: Document Object Model



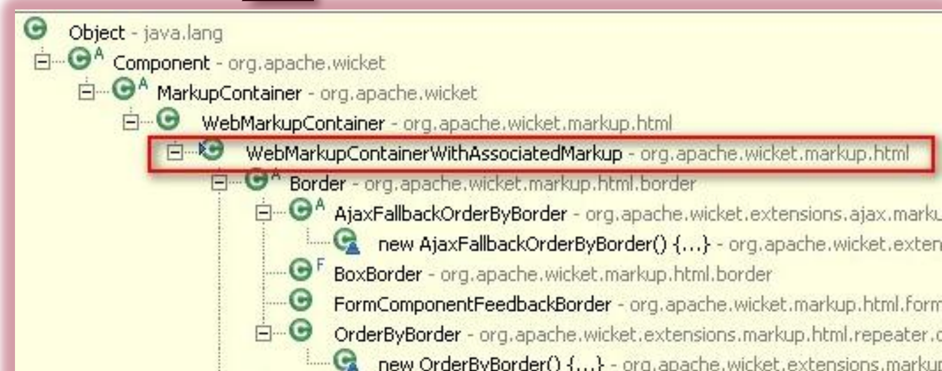
Well-formedness constraints



Defined by a metamodel



Call graph (View)



Type hierarchy (View)

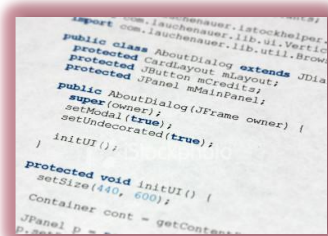
Eclipse IMP framework

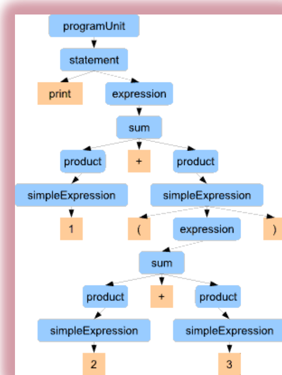
Refactoring

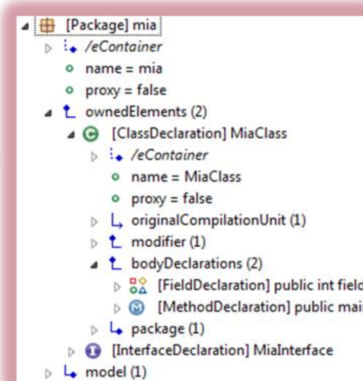
Source code $_1$

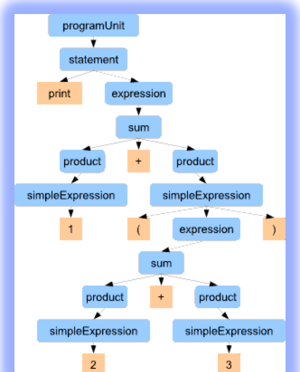
AST $_1$

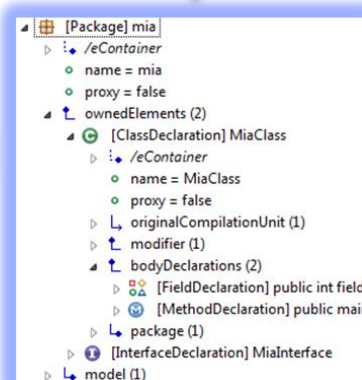
DOM$_1$









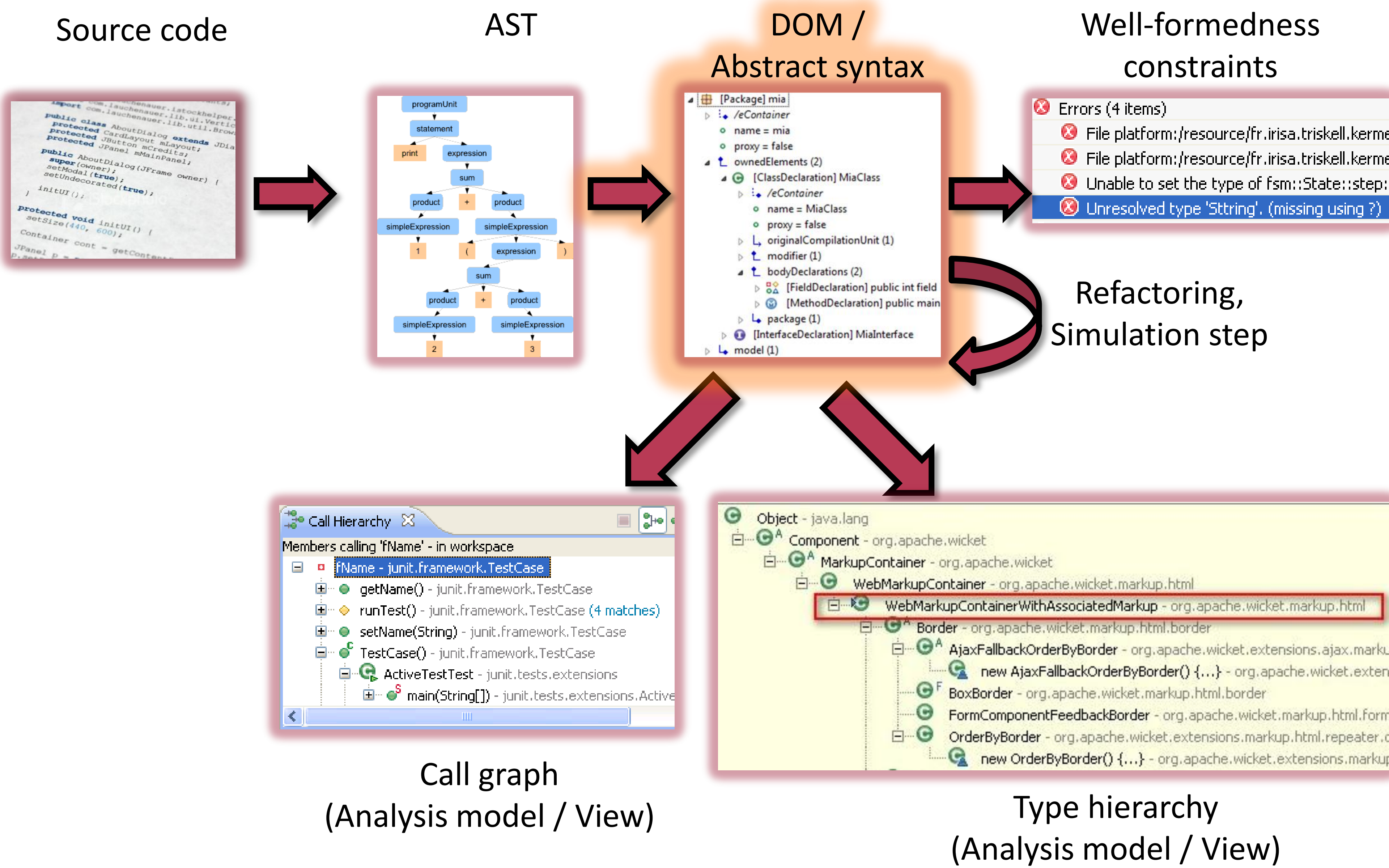


Source code $_2$

AST$_2$

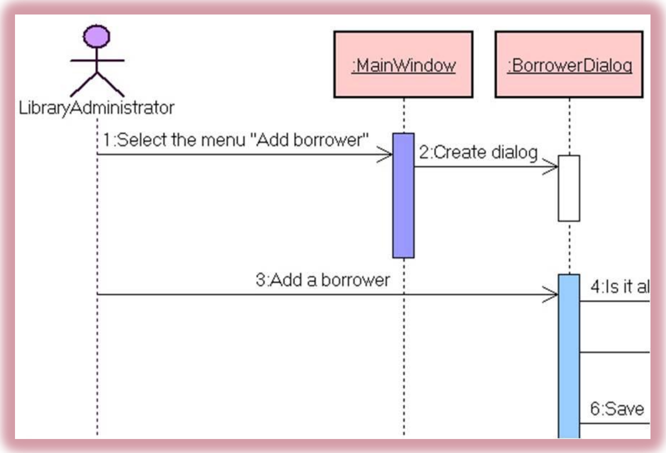
DOM$_2$

Textual DSM Languages: An Overview

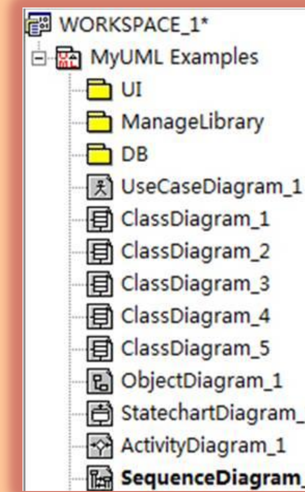


Graphical DSM Languages

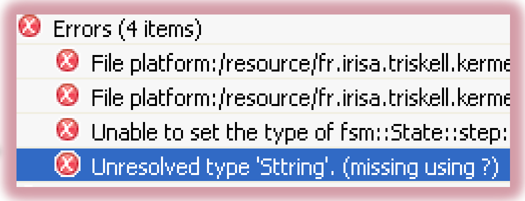
Diagram model



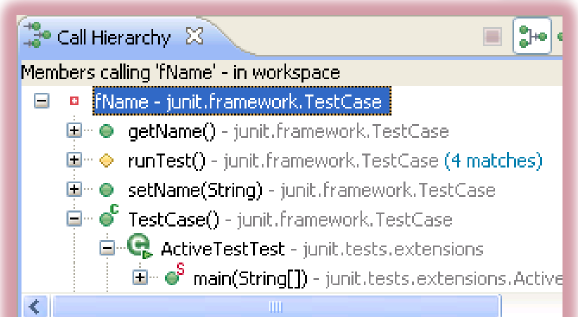
Abstract syntax



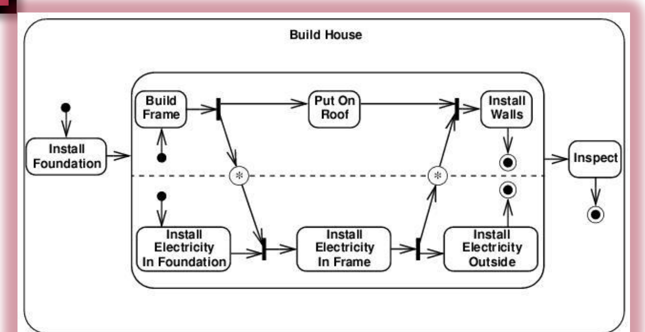
Well-formedness constraints



Refactoring, Simulation



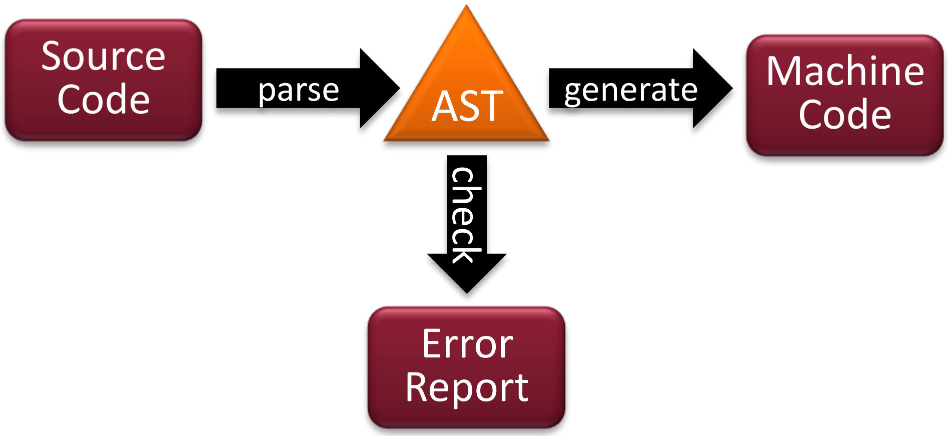
Call graph (View)



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

Source: c

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)

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)

- Num → Digit Num
- Num → Digit
- Digit → 0 | 1 | 2 … | 9

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

Empty symbol

Derivation and Language

- **Derivation step**:
  using grammar G = (T,N,P,S)
  - αAγ ➔ αβγ
  - applying production rule: A → β
  - αAγ, αβγ : sentential forms
- **Derivation over G**: S ➔* w
  where
  - S: start symbol
  - ➔* transitive closure
    (apply as long as possible)
  - w ∈ T*: sentence, i.e.
    string of terminals only
- **Language generated by G**
  - L(G) = {w ∈ T* | there exists a derivation S ➔* w of G}
  - Set of sentences derivable from S

Example derivation

Num	Num → Digit Num
Digit Num	Digit → 1
1 Num	Num → Digit Num
1 Digit Num	Digit → 9
1 9 Num	Num → Digit Num
1 9 Digit Num	Num → Digit
1 9 Digit Digit	Digit → 6
1 9 Digit 6	Digit → 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 polynomical algorithm for regular and context-free grammars

Binary Operations over Numbers

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

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

Two Deriviations:

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

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

Scanning and Parsing

Tokenizer / Lexer / Scanner

- **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)*

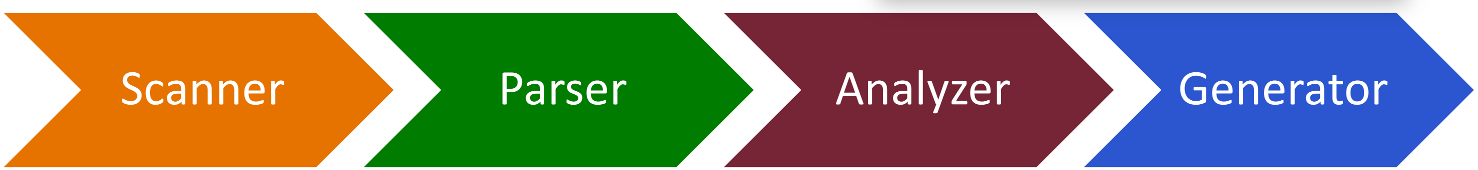
Parser

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

Grammar:

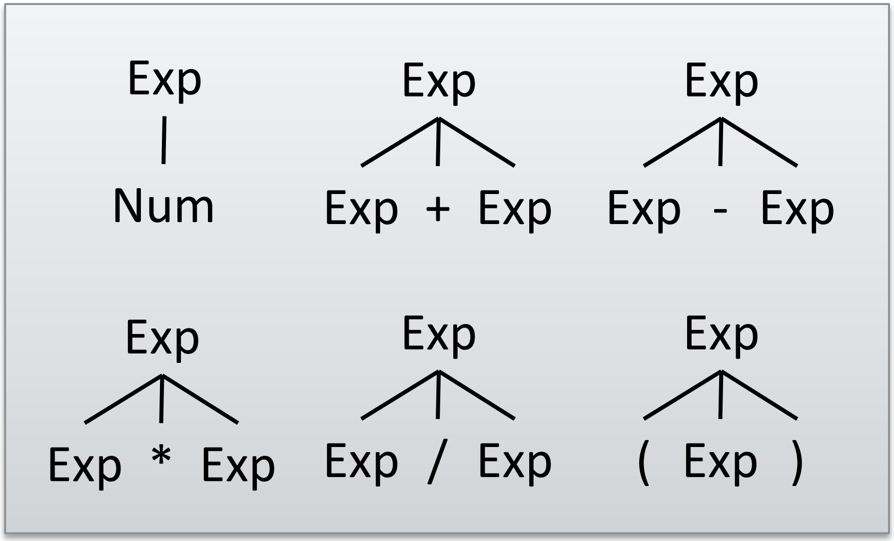
Exp → Num | Exp ”+” Exp
| Exp ”-” Exp | Exp ”*” Exp
| Exp ”/” Exp | ”(” Exp ”)”

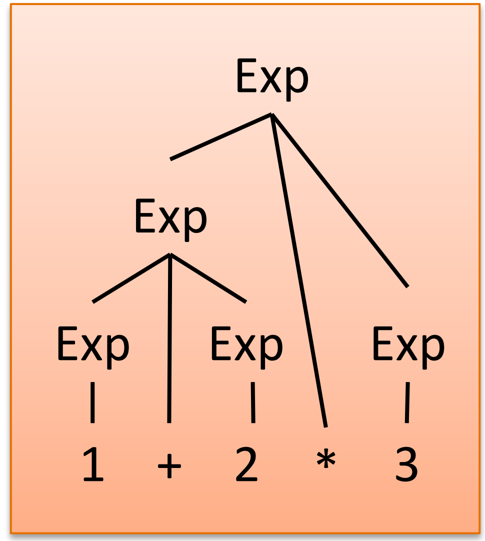
EBNF Notation

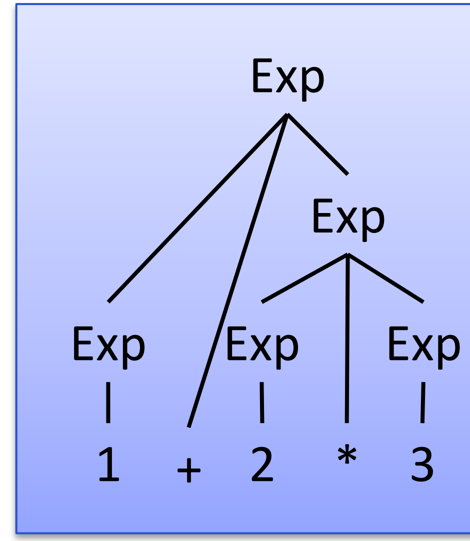


Parse Trees and Abstract Syntax Trees

Parse Tree Construction







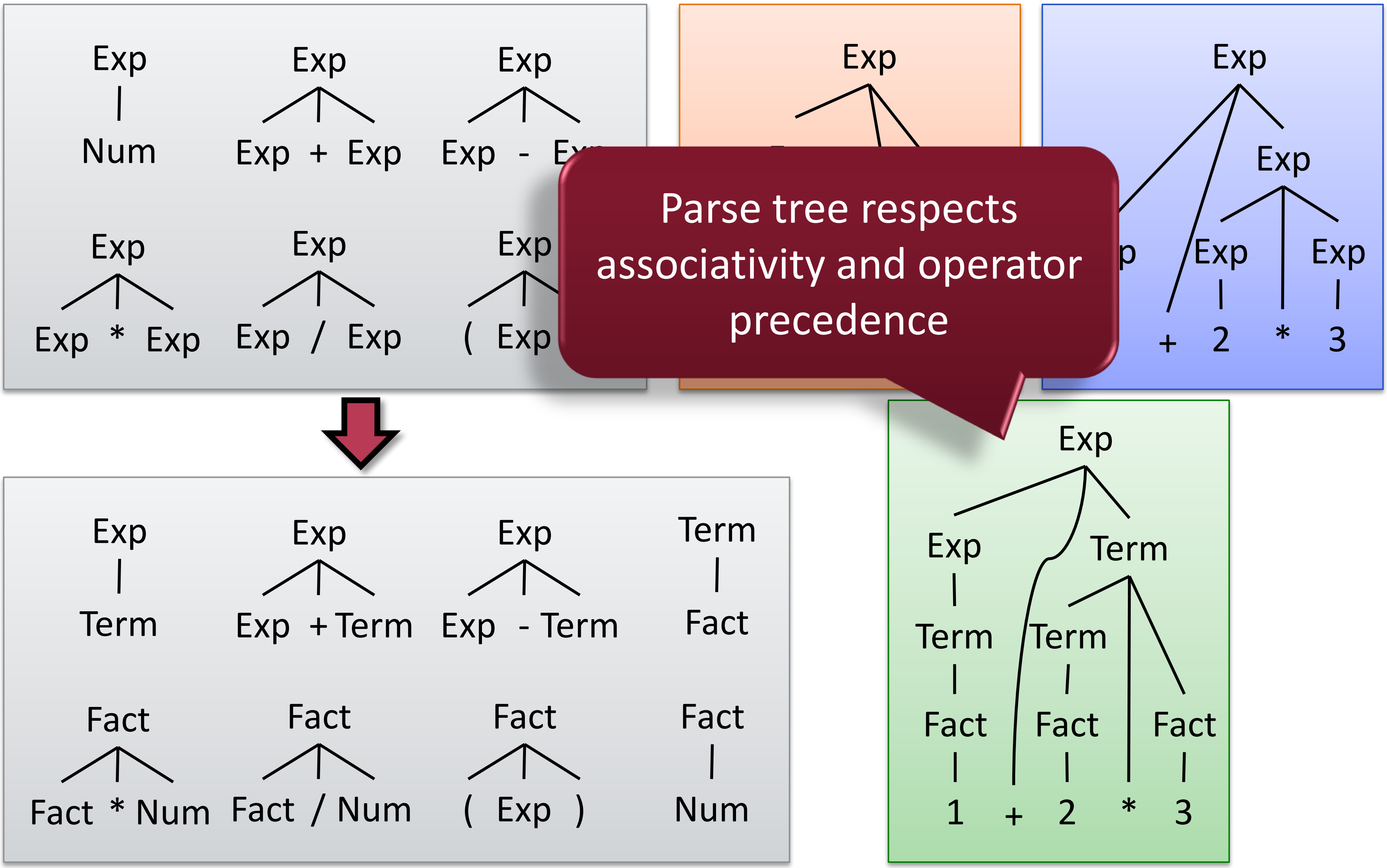
- **Parse Tree:**
  - Parent node: nonterminals
  - Child node: nonterminals/terminals
  - Built up according to productions of the grammar

Ambiguous derivation!
How to disambiguate?

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

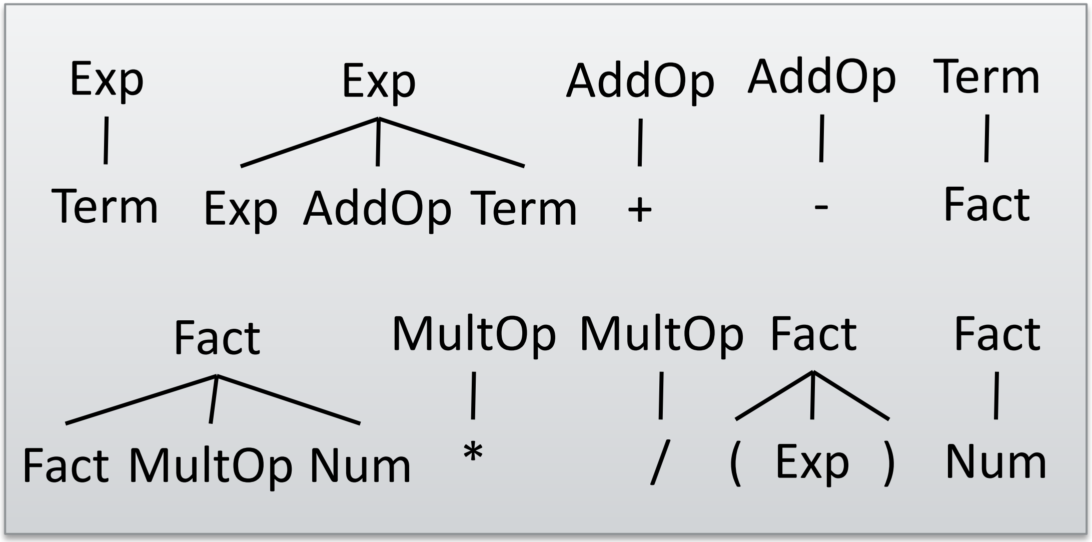
Idea: Change the grammar to force the correct parse tree!

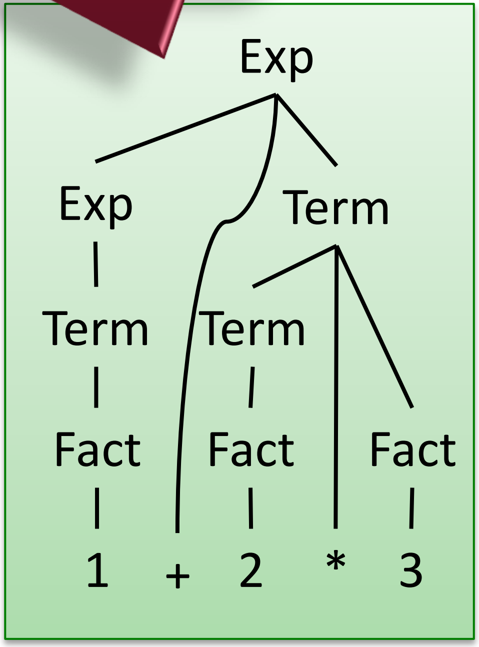
Disambiguation: Binary Expression



Simplifying Grammar: Binary Expression

Parse tree respects associativity and operator precedence





Disambiguation: If-then-else

Grammar: (bold terminals)

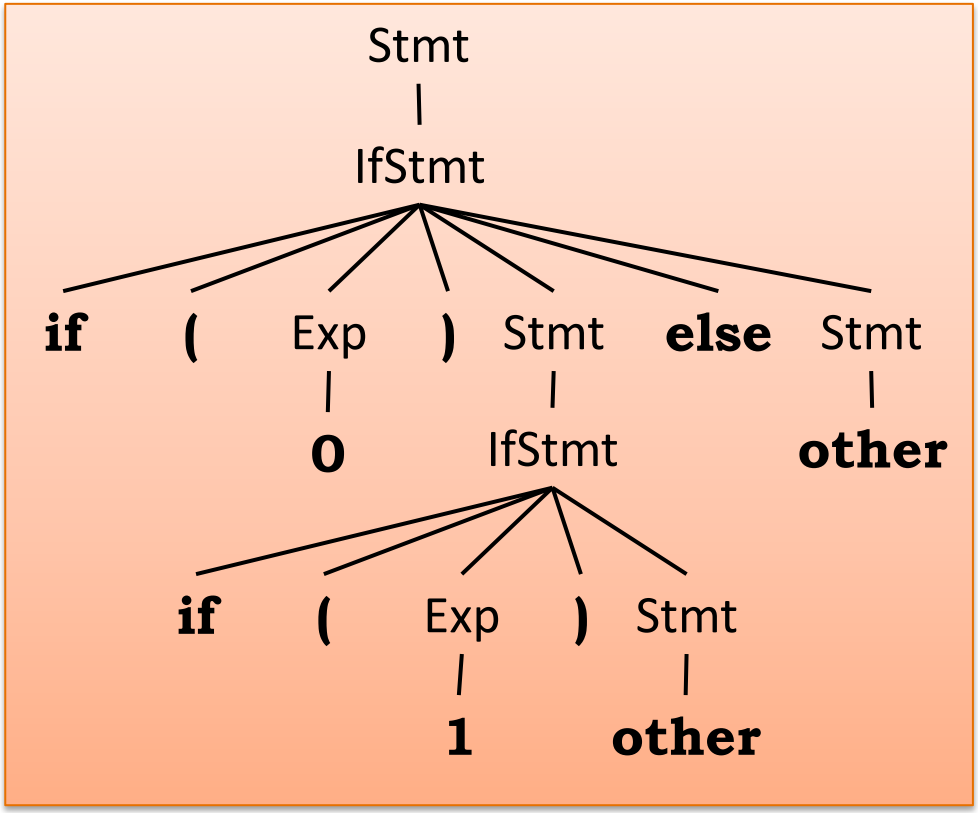
Stmt → IfStmt | **other**

IfStmt →
  if (Exp **)** Stmt |
  if (Exp **)** Stmt **else** Stmt

Exp → **0** | **1**

Ambiguous sample program:

if (0) if (1) other else other



Disambiguation: If-then-else

Grammar: (bold terminals)

Stmt → IfStmt | **other**

IfStmt →

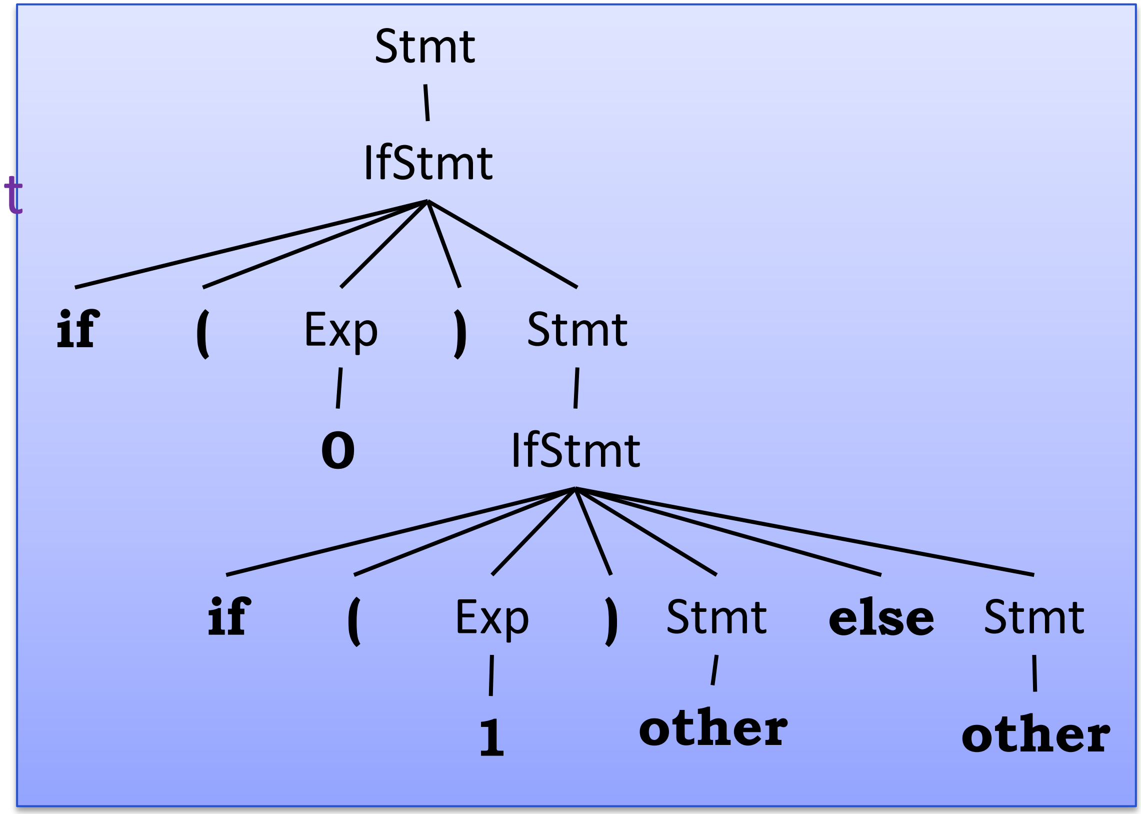
  if (Exp **)** Stmt |

  if (Exp **)** Stmt **else** Stmt

Exp → **0** | **1**

Ambiguous sample program:

if (0) if (1) other else other



Disambiguation rule:
Most closely nested else

Disambiguation: If-then-else

Grammar: (bold terminals)

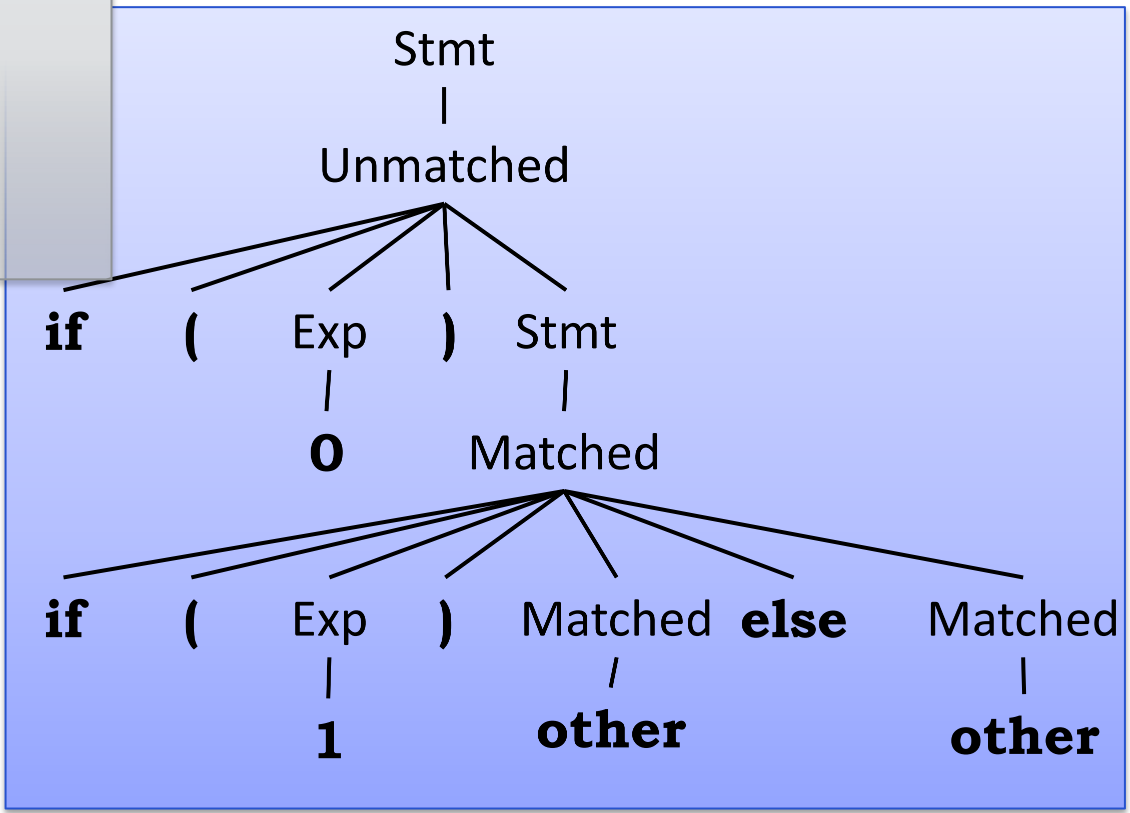
Stmt →
UnMatched | Matched

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

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

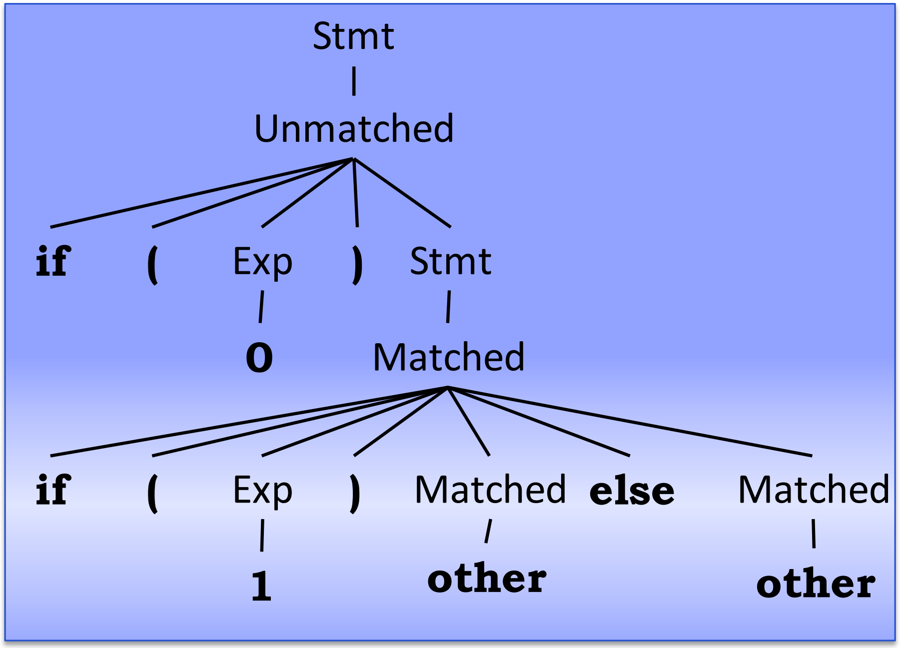
Exp → **0** | **1**

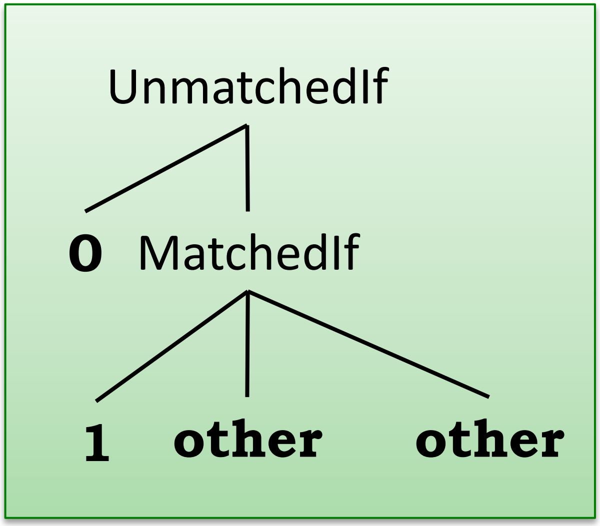
Unambiguous program:
(else matched to 2nd if construct)
if (0) if (1) other else other



Parse Tree vs. Abstract Syntax Tree

- **Parse tree (Concrete syntax)**
  - Contains more information than absolutely necessary for a compiler
- **Abstract Syntax Tree**:
  - Abstractions of source code tokens
    - Abstract over injective productions
    - Individual tokens cannot be recovered
  - Contain all information necessary for translation
  - Defined by
    - Another grammar (EBNF)
    - Auto-generated from concrete grammar





Predictive Parsing LL(k)

Recursive Descent + Look Ahead

Implementing Predictive Parsing

Exp → **while** Exp **do** Exp

Exp → **if** Exp **then** Exp **else** Exp

```
public  void parseExp() {
  switch  current () {
    case WHILE:  consume(WHILE);  parseExp (); ... ; break ;
    case IF   :  consume(IF); parseExp (); ... ; break ;
    default    : error();
  }
}
```

Parser programs are auto-generated by parser generator frameworks (JavaCC, LPG, ANTLR)

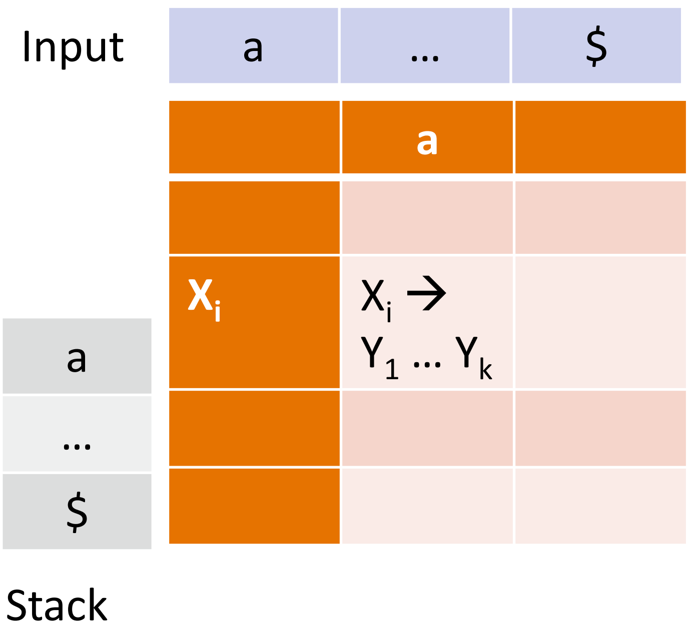
Parse Tables

- **Rows**
  - Nonterminals: N
  - Symbol to parse
- **Columns**
  - Sequnce of terminals: T^k
  - Look ahead k
- **Entries**
  - Production rules: P
  - Possible conflicts
  - Error handling ("empty" entries)

	T_1	T_2	T_3	…
N_1	$N_1 \rightarrow$…		$N_1 \rightarrow$…	
N_2		$N_2 \rightarrow$…		
N_3		$N_3 \rightarrow$…	$N_3 \rightarrow$…	
N_4	$N_4 \rightarrow$…			
N_5		$N_5 \rightarrow$…		
…				

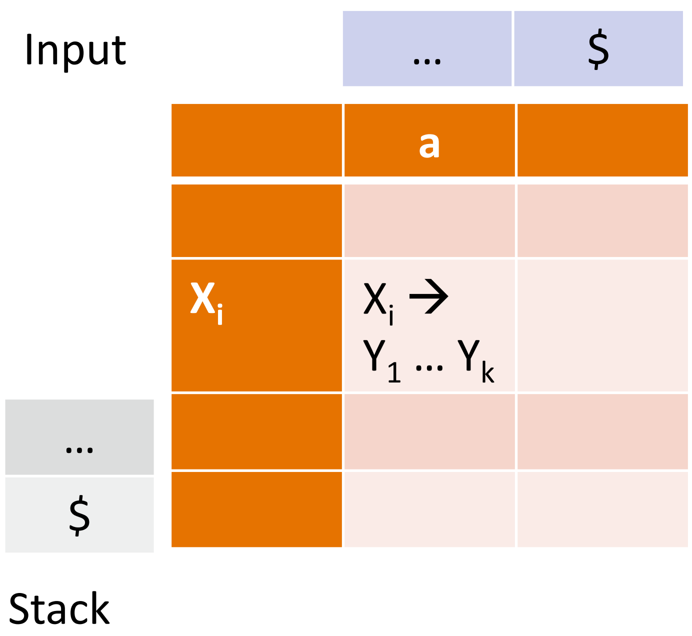
Predictive parsing

- Processing terminals



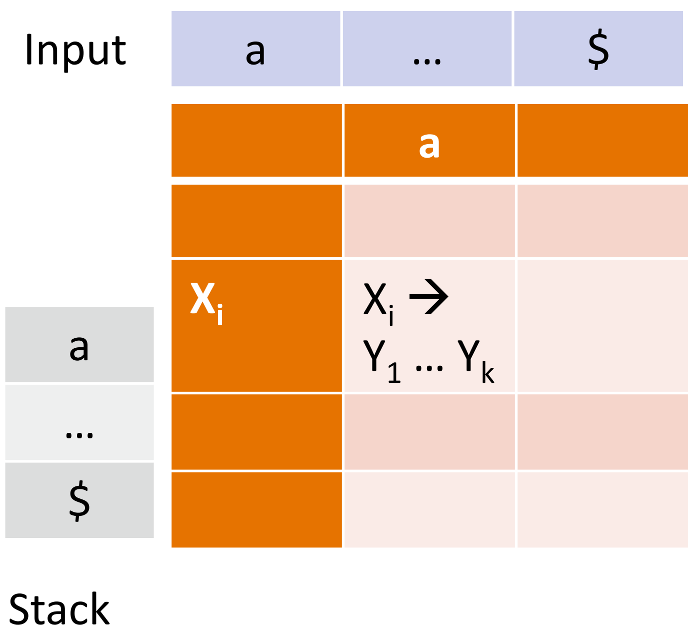
Predictive parsing

- Processing terminals

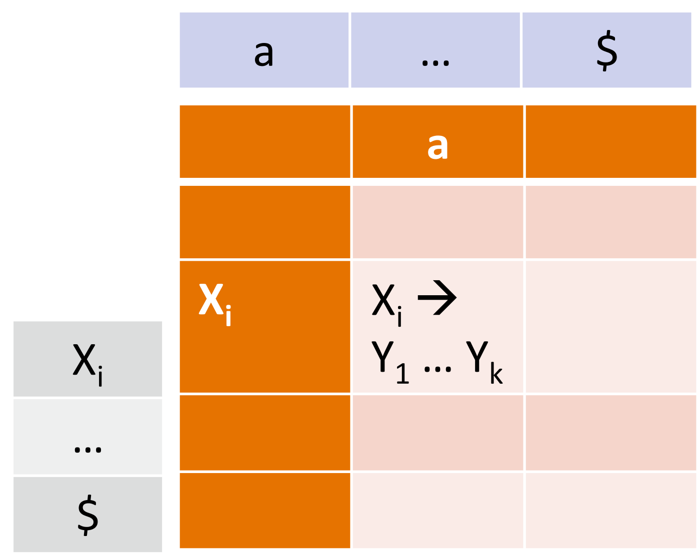


Predictive parsing

- Processing terminals

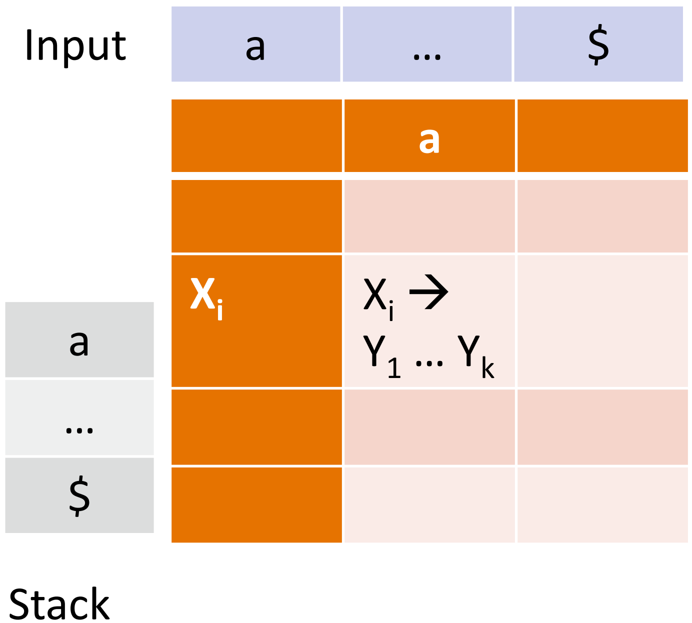


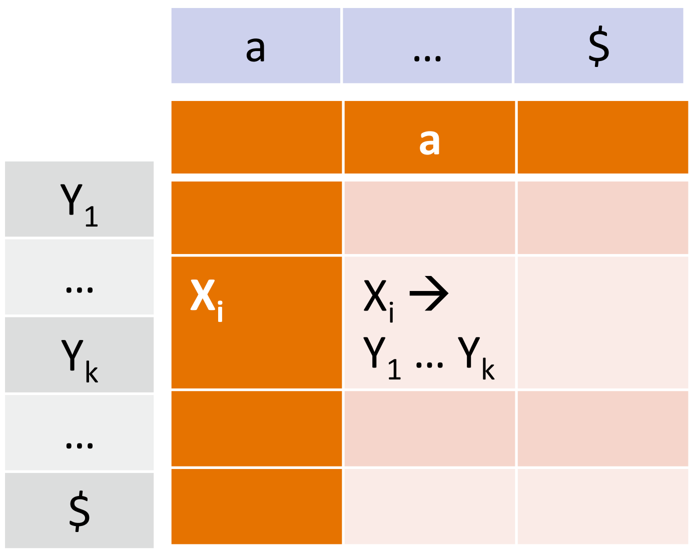
- Processing nonterminals



Predictive parsing

- Processing terminals
- Processing nonterminals





Defining Table Entries

- Entry at (X,w)
  at Row X column T
  - T ∈ FIRST(w)

    Letters that w can start with
  - nullable (w) ∧ T ∈ FOLLOW(X)

    w → * ε (can be empty)

    Letters that can follow X

- Preprocessing is needed for the grammar
  - Eliminate left recursion
  - By Factoring

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

Exp → Exp "+" Term | Term

Term → Term "*" Fact | Fact

Fact → "(" Exp ")" | Num

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

Sample Parse Table

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

	nullable	FIRST	FOLLOW
Exp	No		
Exp'	Yes		
Term	No		
Term'	Yes		
Fact	No		

Nullable?

Sample Parse Table

New grammar G’ = (N,T,P,S)

Term’ → ”*” Fact Term’ | ε

Term → Fact Term’

Exp’ → ”+” Term Exp’ | ε

Exp → Term Exp’

Fact → ”(” Exp ”)” | Num

	nullable	FIRST	FOLLOW
Exp	No	Num (	
Exp’	Yes	+	
Term	No	Num (	
Term’	Yes	*	
Fact	No	Num (	

FIRST set?

Sample Parse Table

New grammar G’ = (N,T,P,S)

Term’ → ”*” Fact Term’ | ε

Term → Fact Term’

Exp’ → ”+” Term Exp’ | ε

Exp → Term Exp’

Fact → ”(” Exp ”)” | Num

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp’	Yes	+	)
Term	No	Num (	+)
Term’	Yes	*	+)
Fact	No	Num (	* +)

FOLLOW set?

Sample LL(k) Parsing

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

1	+	2	*	3	$

Input

Exp
$

Stack

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp'	Yes	+	)
Term	No	Num (	+)
Term'	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G’ = (N,T,P,S)

Term’ → ”*” Fact Term’ | ε

Term → Fact Term’

Exp’ → ”+” Term Exp’ | ε

Exp → Term Exp’

Fact → ”(” Exp ”)” | Num

1	+	2	*	3	$

Input

Term
Exp’
$

Stack

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp’	Yes	+	)
Term	No	Num (	+)
Term’	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G’ = (N,T,P,S)

Term’ → ”*” Fact Term’ | ε

Term → Fact Term’

Exp’ → ”+” Term Exp’ | ε

Exp → Term Exp’

Fact → ”(” Exp ”)” | Num

1	+	2	*	3	$

Input

Stack
Fact
Term’
Exp’
$

Stack

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp’	Yes	+	)
Term	No	Num (	+)
Term’	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

1	+	2	*	3	$	Input

Stack
Num
Term'
Exp'
$

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp'	Yes	+	)
Term	No	Num (	+)
Term'	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

+	2	*	3	$

Input

Term'
Exp'
$

Stack

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp'	Yes	+	)
Term	No	Num (	+)
Term'	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

+	2	*	3	$

Input

Exp'
$

Stack

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp'	Yes	+	)
Term	No	Num (	+)
Term'	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

+	2	*	3	$

Input

+
Term
Exp'
$

Stack

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp'	Yes	+	)
Term	No	Num (	+)
Term'	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp'	Yes	+	)
Term	No	Num (	+)
Term'	Yes	*	+)
Fact	No	Num (	* +)

Input

2	*	3	$

Stack

Term
Exp'
$

Sample LL(k) Parsing

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

2	*	3	$

Input

Fact
Term'
Exp'
$

Stack

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp'	Yes	+	)
Term	No	Num (	+)
Term'	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G’ = (N,T,P,S)

Term’ → ”*” Fact Term’ | ε

Term → Fact Term’

Exp’ → ”+” Term Exp’ | ε

Exp → Term Exp’

Fact → ”(” Exp ”)” | Num

2	*	3	$

Input

Num
Term’
Exp’
$

Stack

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp’	Yes	+	)
Term	No	Num (	+)
Term’	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G’ = (N,T,P,S)

Term’ → ”*” Fact Term’ | ε

Term → Fact Term’

Exp’ → ”+” Term Exp’ | ε

Exp → Term Exp’

Fact → ”(” Exp ”)” | Num

*	3	$

Input

Term’
Exp’
$

Stack

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp’	Yes	+	)
Term	No	Num (	+)
Term’	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G’ = (N,T,P,S)

Term’ → ”*” Fact Term’ | ε

Term → Fact Term’

Exp’ → ”+” Term Exp’ | ε

Exp → Term Exp’

Fact → ”(” Exp ”)” | Num

*	3	$

Input

*
Fact
Term’
Exp’
$

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp’	Yes	+	)
Term	No	Num (	+)
Term’	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

3	$

Input

Fact
Term'
Exp'
$

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp'	Yes	+	)
Term	No	Num (	+)
Term'	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G’ = (N,T,P,S)

Term’ → ”*” Fact Term’ | ε

Term → Fact Term’

Exp’ → ”+” Term Exp’ | ε

Exp → Term Exp’

Fact → ”(” Exp ”)” | Num

3	$

Input

Num
Term’
Exp’
$

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp’	Yes	+	)
Term	No	Num (	+)
Term’	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G’ = (N,T,P,S)

Term’ → ”*” Fact Term’ | ε

Term → Fact Term’

Exp’ → ”+” Term Exp’ | ε

Exp → Term Exp’

Fact → ”(” Exp ”)” | Num

$

Input

Term’

Exp’

$

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp’	Yes	+	)
Term	No	Num (	+)
Term’	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G’ = (N,T,P,S)

Term’ → ”*” Fact Term’ | ε

Term → Fact Term’

Exp’ → ”+” Term Exp’ | ε

Exp → Term Exp’

Fact → ”(” Exp ”)” | Num

$

Input

Exp’
$

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp’	Yes	+	)
Term	No	Num (	+)
Term’	Yes	*	+)
Fact	No	Num (	* +)

Sample LL(k) Parsing

New grammar G' = (N,T,P,S)

Term' → "*" Fact Term' | ε

Term → Fact Term'

Exp' → "+" Term Exp' | ε

Exp → Term Exp'

Fact → "(" Exp ")" | Num

$

Input

$

Successful parsing

	nullable	FIRST	FOLLOW
Exp	No	Num (	)
Exp'	Yes	+	)
Term	No	Num (	+)
Term'	Yes	*	+)
Fact	No	Num (	* +)

Summary

Goal: Processing of textual languages

- First steps: scanning + parsing
  - Scan: character stream → token sequence
  - Parse: token sequence → parse tree → AST
- Formal basics
  - Grammar =
    Terminals + nonterminals + productions + start
  - Language generated by a grammar
  - Refactoring: Disambiguation, etc.
- Predictive parsing (LL(k))