Lecture 26: Grammars

CMPS 258 – Programming Languages

Course Organization

Programming Language Design (7 weeks)			Programming Language Implementation (4 weeks)		
✓ Functional Programming (4 weeks)	✓ Object-Oriented Programming (2 weeks)	✓ Type Systems (1 week)	✓ **Lexing** (1 week)	Parsing (1.5 weeks)	Semantic Analysis (1.5 weeks)

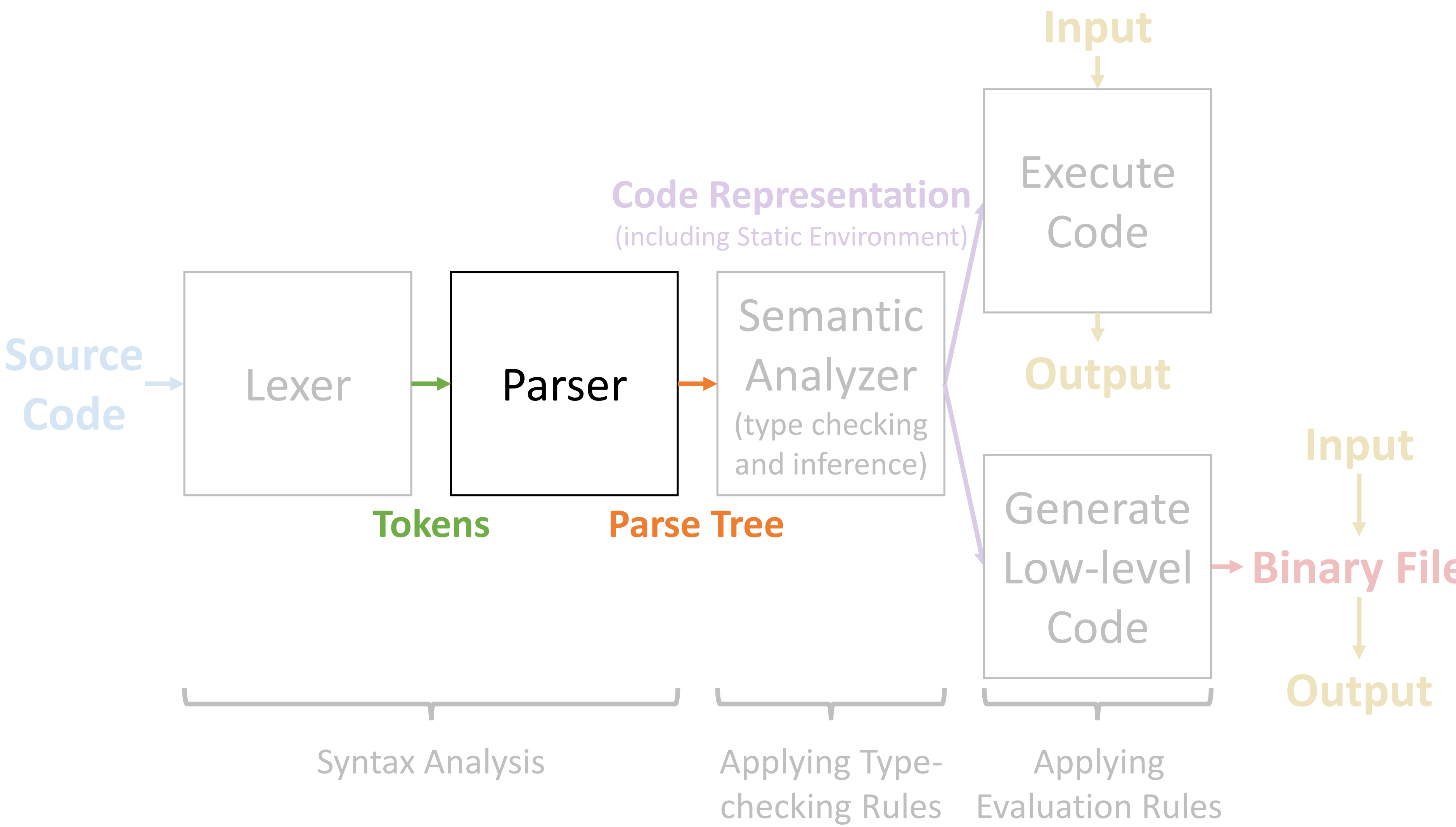
Introduction to the first stages of implementing a programming language

Course Organization

Programming Language Design (7 weeks)			Programming Language Implementation (4 weeks)		
Functional Programming (4 weeks) ✓	Object-Oriented Programming (2 weeks) ✓	Type Systems (1 week) ✓	Lexing (1 week) ✓	**Parsing (1.5 weeks)**	Semantic Analysis (1.5 weeks)

Introduction to the first stages of implementing a programming language

Parsers



Input
Execute Code
Output
Code Representation
(including Static Environment)
Source Code
Lexer
Parser
Semantic Analyzer
(type checking and inference)
Input
Generate Low-level Code
Binary File
Output
Tokens
Parse Tree
Syntax Analysis
Applying Type-checking Rules
Applying Evaluation Rules

Designing Language Syntax

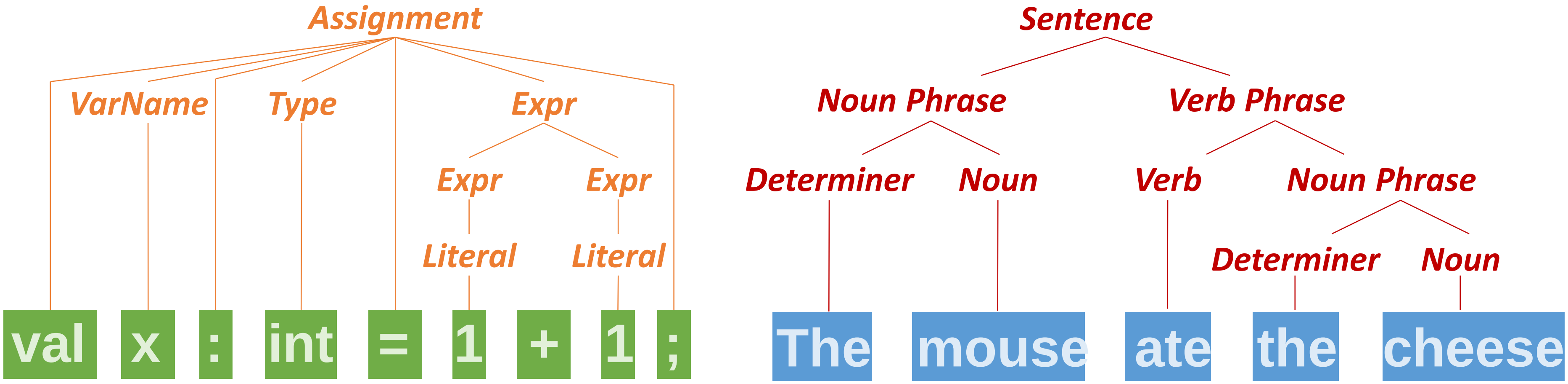
- Recall: One aspect of designing the syntax of a language is specifying the set of rules for what tokens are valid in that language (lexical syntax)
  - Analogous to *spelling* in natural languages (valid ways to write words in the language)

val x : int = 1 + 1 ;

The mouse ate the cheese

Designing Language Syntax

- Another aspect of designing the syntax of a language is specifying a set of rules for how tokens can be assembled together in a meaningful way
  - Analogous to *grammar* in natural languages (valid ways to combine words to form meaningful sentences)



Context Free Grammars

- A **context free grammar** is a set of rules for describing all possible strings in a language

- It is context free because the rules can be applied regardless of context

Context Free Grammars

- A context free grammar is made up of four parts:
  - Terminals: Σ
    - Words/tokens
  - Non-terminals: N
    - Phrases

$\Sigma \cup N = V$ *(vocabulary)*

$\Sigma \cap N = \Phi$

  - Start symbol: S
    - Sentence/program, $S \in N$
  - Production rules: P
    - Relation from N to V^*, notation commonly used for describing production rules is called **Backus-Naur form** (BNF)

Context Free Grammar Example

- Example grammar for arithmetic expressions
  - Terminals: Σ = { id , + , * , (,) }
  - Non-terminals: N = { Expr }
  - Start symbol: S = { Expr }
  - Production rules: P = {

```
Expr ::= Expr + Expr
       | Expr * Expr
       | ( Expr )
       | id
```

}

Derivations

- A **derivation** is a sequence of production rule applications to derive a string from the start symbol
  - Serves as a proof that the string is valid in the grammar

- Example derivation: id + id * id

Expr	=> Expr + Expr	Expr	=> Expr + Expr
	=> id + Expr		=> Expr + Expr * Expr
	=> id + Expr * Expr		=> Expr + Expr * id
	=> id + id * Expr		=> Expr + id * id
	=> id + id * id		=> id + id * id

Expr ::= Expr + Expr
| Expr * Expr
| (Expr)
| id

Could have applied the same rules on the same non-terminal in a different order and gotten a different derivation as above

Canonical Forms of Derivation

- A canonical form of derivation deterministically chooses which non-terminal to rewrite
  - **Leftmost derivation:** replace leftmost non-terminal first
  - **Rightmost derivation:** replace rightmost non-terminal first

Expr	=> **Expr** + Expr	Expr	=> Expr + **Expr**
	=> id + **Expr**		=> Expr + Expr * **Expr**
	=> id + **Expr** * Expr		=> Expr + **Expr** * id
	=> id + id * **Expr**		=> **Expr** + id * id
	=> id + id * id		=> id + id * id
	Leftmost derivation		**Rightmost derivation**

Expr ::= Expr + Expr
| Expr * Expr
| (Expr)
| id

Parse Trees

- A **parse tree** is a graphical representation of a derivation
  - Leaves correspond to terminals
  - Inner nodes correspond to non-terminals
  - Root node corresponds to start symbol

Parse Tree Example

- Expression: id + id * id

```
Expr   => Expr + Expr
       => id + Expr
       => id + Expr * Expr
       => id + id * Expr
       => id + id * id
```

Leftmost derivation

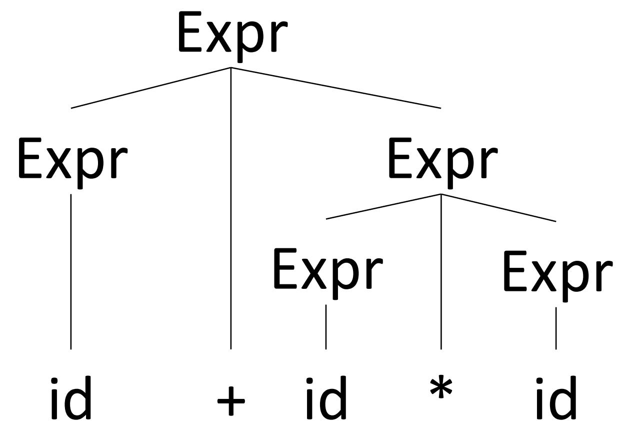
```
Expr   => Expr + Expr
       => Expr + Expr * Expr
       => Expr + Expr * id
       => Expr + id * id
       => id + id * id
```

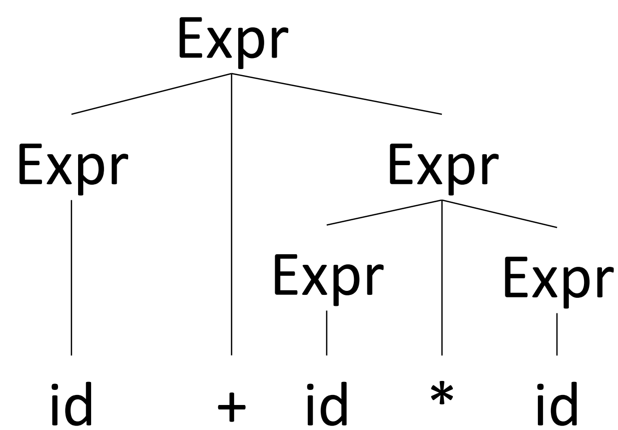
Rightmost derivation

```
Expr ::= Expr + Expr
       | Expr * Expr
       | ( Expr )
       | id
```

Parse tree

(same parse tree for both derivations)



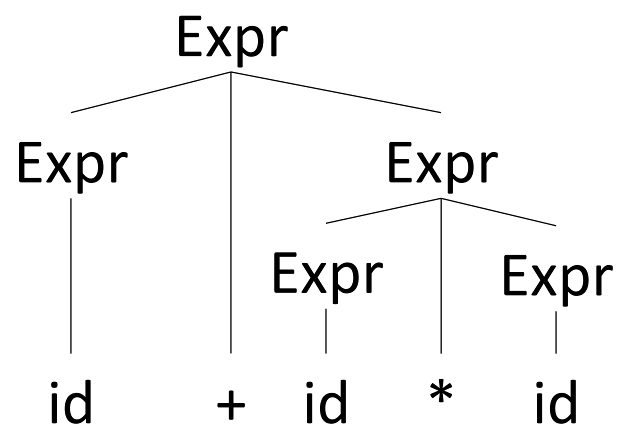


Grammar Ambiguity

- Expression: id + id * id

Expr => **Expr** + Expr
=> id + **Expr**
=> id + **Expr** * Expr
=> id + id * **Expr**
=> id + id * id

Leftmost derivation

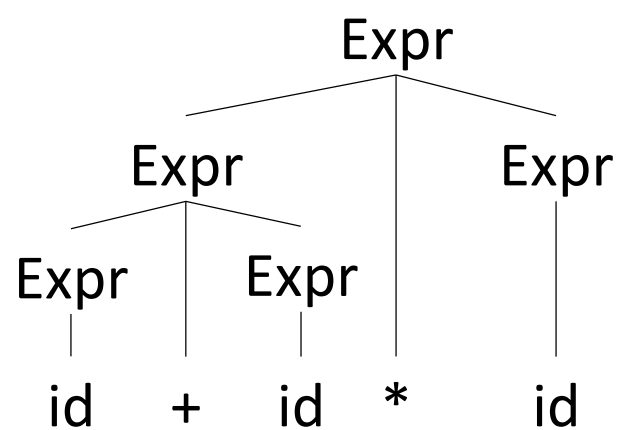


This grammar is ambiguous!

Two different leftmost derivations (and parse trees) can be validly constructed for the same string.

Expr => **Expr** * Expr
=> **Expr** + Expr * Expr
=> id + **Expr** * Expr
=> id + id * **Expr**
=> id + id * id

Leftmost derivation



Expr ::= Expr + Expr
| Expr * Expr
| (Expr)
| id

Disambiguating Grammars

- A grammar is ambiguous if there are multiple derivations <u>of the same canonical form</u> for the same string
- Identifying whether an arbitrary grammar is ambiguous is an undecidable problem
  - There is no mechanical algorithm for disambiguating a grammar
- Two major sources of ambiguity:
  - Undetermined operator **precedence**
  - Undetermined operator **associativity**
  - Not the only sources

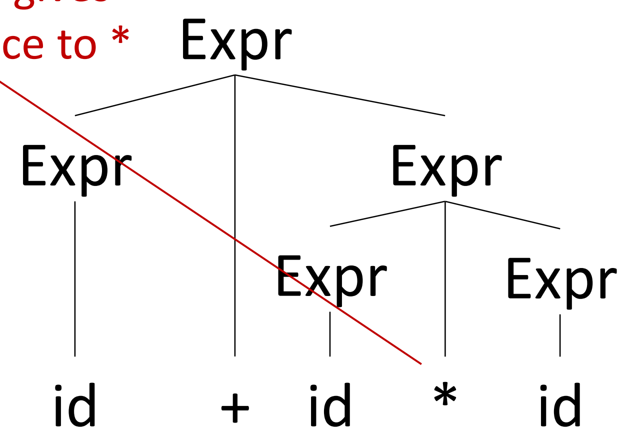
Undetermined Precedence

- Expression: id + id * id

Expr => **Expr** + Expr
=> id + **Expr**
=> id + **Expr** * Expr
=> id + id * **Expr**
=> id + id * id

Leftmost derivation

The tree gives precedence to *



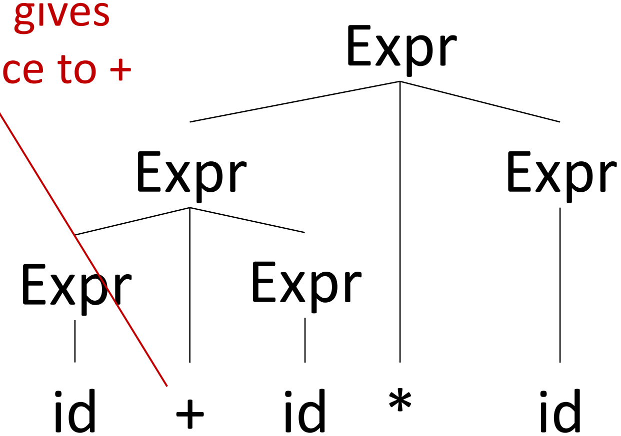
The grammar does not distinguish between the + and * operators

Expr ::= Expr + Expr
| Expr * Expr
| (Expr)
| id

Expr => **Expr** * Expr
=> **Expr** + Expr * Expr
=> id + **Expr** * Expr
=> id + id * **Expr**
=> id + id * id

Leftmost derivation

The tree gives precedence to +



Enforcing Precedence

- Expression: id + id * id

Expr ::= Expr + Expr
| Expr * Expr
| (Expr)
| id

Expr ::= Expr + Expr
| Term
Term ::= Term * Term
| (Expr)
| id

Impossible to have expressions inside terms without parenthesis

Expr => **Expr** + Expr
=> **Term** + Expr
=> id + **Expr**
=> id + **Term**
=> id + **Term** * Term
=> id + id * **Term**
=> id + id * id