Introduction to Syntax Analysis

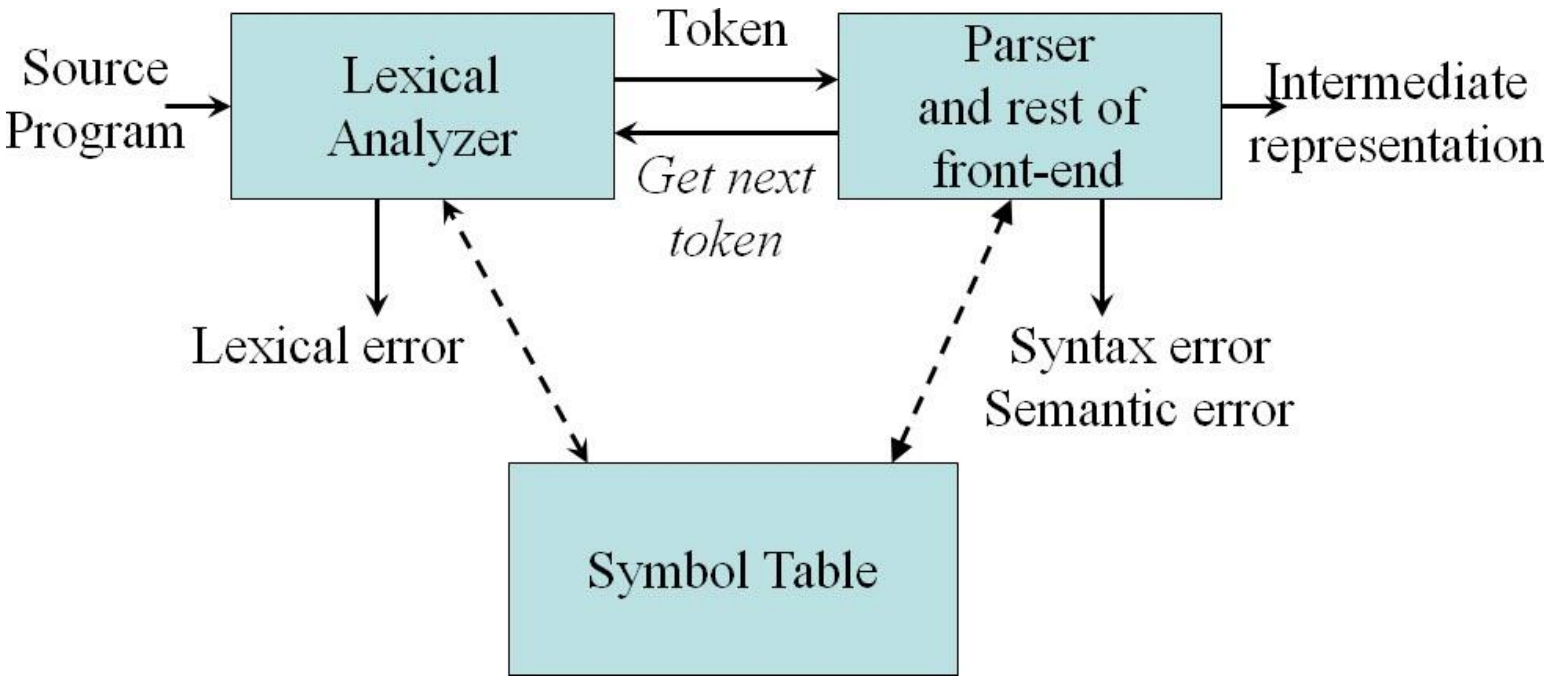
Syntax Analysis

Syntax Analysis

The parser (syntax analyzer) receives the source code in the form of tokens from the lexical analyzer and performs **syntax analysis**, which create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

- Syntax analysis is also called *parsing*.
- A typical representation is a **abstract syntax tree** in which
  - each interior node represents an operation
  - the children of the node represent the arguments of the operation

Position of Syntax Analyzer



Role of Parser

Parser

- Checks the stream of words and their parts of speech (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax (static semantics checking)
- Builds an IR representation of the code

Study of Parsing

Parser

The parser

- Needs the syntax of programming language constructs, which can be specified by context-free grammars or BNF (Backus-Naur Form)
- Need an algorithm for testing membership in the language of the grammar.

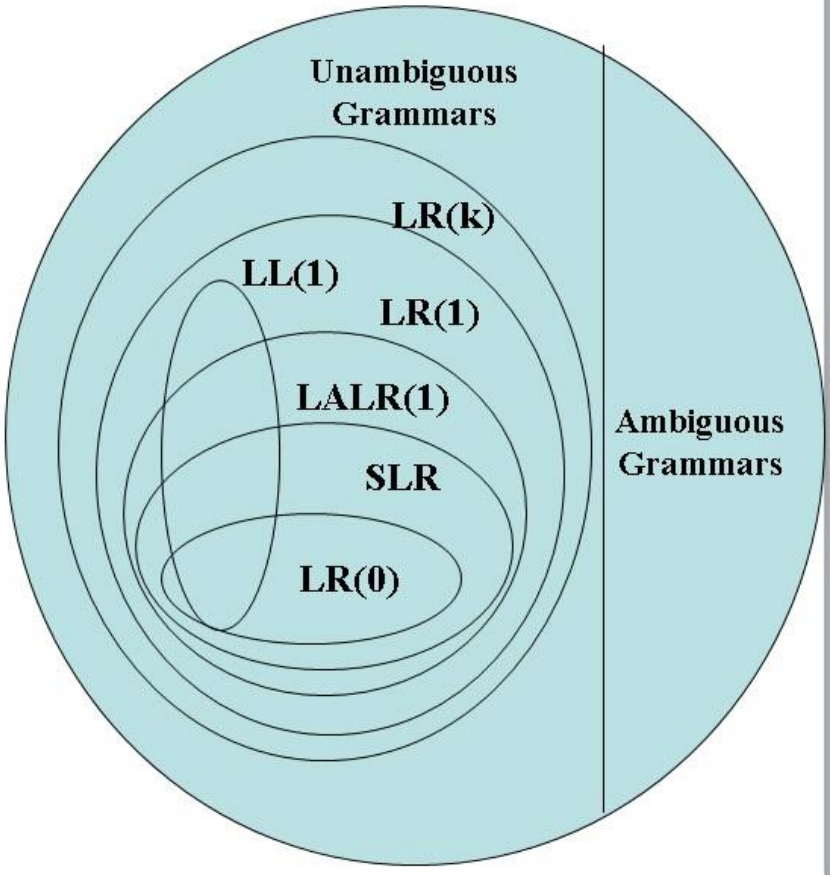
Roadmap

The roadmap for study of parsing

- Context-free grammars and derivations
- Top-down parsing
  - Recursive descent (predictive parsing)
  - LL (Left-to-right, Leftmost derivation) methods
- Bottom-up parsing
  - Operator precedence parsing
  - LR (Left-to-right, Rightmost derivation) methods
    - SLR, canonical LR, LALR

Expressive Power of Different Parsing Techniques

- LR(0) is of academic interest only.
- Simple LR (SLR) can parse some interesting languages.
- LR(1) is a very powerful parsing technology.
  - The implementation of LR(1) can get unwieldy.
  - Engineers try rewrite their grammars into LALR(1).
- Most real computer languages can be expressed as LALR(1) grammars.
  - The YACC, CUP, CSTools, and sableCC parser generators requires this class of grammars.



Benefits Offered by Grammar

Grammars offer significant benefits for both language designers and compiler writers:

- A grammar gives a precise, yet easy-to-understand syntactic specification to a programming language.
- Parsers can automatically be constructed for certain classes of grammars.
  - The parser-construction process can reveal syntactic ambiguities and trouble spots.
- A grammar imparts structure to a language.
  - The structure is useful for translating source programs into correct object code and for detecting errors.
- A grammar allows a language to be evolved.
  - New constructs can be integrated more easily into an implementation that follows the grammatical structure of the language.

Why Not Use RE/DFA?

Advantages of RE/DFA

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Limits of RE/DFA

- Finite automata cannot count, which means a finite automaton cannot accept a language like $\{a^n b^n | n \geq 1\}$ that would require it to keep count of the number of a's before it sees the b's.
- Therefore, RE cannot check the balance of parenthesis, brackets, begin-end pairs.

CFG vs. RE

- Grammars are a more powerful notation than regular expressions.
  - Every construct that can be described by a regular expression can be described by a grammar, but not vice-versa.
  - Every regular language is a context-free language, but not vice-versa.

Context-Free Grammar

Definition

A context-free grammar (CFG) has four components:

- A set of *terminal symbols*, sometimes referred to as "tokens."
- A set of *nonterminal* symbols. sometimes called "syntactic variables."
- One nonterminal is distinguished as the *start symbol*.
- A set of *productions* in the form: *LHS* → *RHS*
  where
  - LHS (called head, or left side) is a single nonterminal symbol
  - RHS (called body, or right side) consists of zero or more terminals and nonterminals.

- The terminals are the elementary symbols of the language defined by the grammar.
- Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
- Conventionally, the productions for the start symbol are listed first.
- The productions specify the manner in which the terminals and nonterminals can be combined to form strings.

CFG Example

A CFG Grammar

1. Expr → Expr Op
2. Expr → **number**
3. Expr → **id**
4. Op → +
5. Op → -
6. Op → *
7. Op → /

where

- Expr and Op are nonterminals
- number, id, +, -, *, and / are terminals
- Expr is the start symbol

CFG Example

Productions with the same head can be grouped. Therefore, the previous CFG grammar is equivalent to the one below.

Equivalent CFG Grammar

1. Expr → Expr Op Expr | number | id
2. Op → + | - | * | /

Another CFG Example

Grammar for simple arithmetic expressions

1. *expr* → *expr* + *term* | *expr* - *term* | *term*
2. *term* → *term* * *factor* | *term* / *factor* | *factor*
3. *factor* → (*expr*) | **id**

where

- expr, term, and factor are nonterminals
- id, +, -, *, /, (, and) are terminals
- expr is the start symbol

Notational Conventions

To avoid confusion between terminals and nonterminals, the following notational conventions for grammar will be used.

terminal symbols

- lowercase letters like a, b, c.
- digits, operator and punctuation symbols, such as +, *, (,), 0, 1, ..., 9.
- Boldface strings such as **id**, or **if**. Each of which represents a single terminal symbol.

nonterminal symbols

- uppercase letters early in the alphabet like A, B, C.
- lowercase italic names such as *expr*, or *stmt*.
- Specific symbols begin with a uppercase letter such as Expr, Stmt.
- Unless stated otherwise, the head of the first production is the start symbol.

Notational Conventions (cont)

To avoid confusion between terminals and nonterminals, the following notational conventions for grammar will be used.

Grammar symbols (i.e. either terminal or nonterminal)

- uppercase letters late in the alphabet, such as X, Y, Z.
- lowercase Greek letters, α, β, γ for example, represent strings of grammar symbols. Thus, a production can be written as: $A \rightarrow \alpha$.

Derivations

Derivations

A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal. This sequence of replacements is called **derivation**.

Derivation Example

Given the grammar:

1. *exp* → *exp op exp* | (*exp*) | **number**
2. *op* → + | - | *

The following is a derivation for an expression. At each step the grammar rule choice used for the replacement is given on the right.

(1)	*exp* ⇒ *exp op exp*	[*exp* → *exp op exp*]
(2)	⇒ *exp op* **number**	[*exp* → **number**]
(3)	⇒ *exp* * **number**	[*op* → *]
(4)	⇒ (*exp*) * **number**	[*exp* → (*exp*)]
(5)	⇒ (*exp op exp*) * **number**	[*exp* → *exp op exp*]
(6)	⇒ (*exp op* **number**) * **number**	[*exp* → **number**]
(7)	⇒ (*exp* - **number**) * **number**	[*op* → -]
(8)	⇒ (**number** - **number**) * **number**	[*exp* → **number**]

Context-Free Language

New Notations: $\stackrel{*}{\Rightarrow}$ and $\stackrel{+}{\Rightarrow}$

$\alpha_1 \stackrel{*}{\Rightarrow} \alpha_n$ means α_1 derives α_n in zero or more steps.

$\alpha_1 \stackrel{+}{\Rightarrow} \alpha_n$ means α_1 derives α_n in one or more steps.

Definition

- If $S \stackrel{*}{\Rightarrow} \alpha$, where S is the start symbol of grammar G, then α is called a *sentential form* of G. A sentential form may contain both terminals and nonterminals.
- A *sentence* of G is a sentential form with **no** nonterminals.
- The *language generated* by a grammar G is its set of sentences, denoted as L(G).
- A language that can be generated by a context-free grammar is said to be a *context-free language*.
- If two grammars generate the same language, the grammars are said to be *equivalent*.
- Process of discovering a derivation is called **parsing**.

Leftmost and Rightmost Derivations

- The point of parsing is to construct a derivation.
  - At each step, we choose a nonterminal to replace.
  - Different choices can lead to different derivations
- Two derivations are of interest
  - *Leftmost* derivation - replace leftmost nonterminal at each step, denoted as: $\Rightarrow_{lm}$.
  - *Rightmost* derivation - replace rightmost nonterminal at each step, denoted as: $\Rightarrow_{rm}$.
- Leftmost and rightmost are the two systematic derivations. We don't care about randomly-ordered derivations!

Leftmost and Rightmost Derivations

Leftmost Derivation of (**number** - **number**)*number

(1)	*exp* ⇒ *exp op exp*	[*exp* → *exp op exp*]
(2)	⇒ **(***exp***)** *op exp*	[*exp* → **(** *exp* **)**]
(3)	⇒ **(***exp op exp***)** *op exp*	[*exp* → *exp op exp*]
(4)	⇒ **(number** *op exp***)** *op exp*	[*exp* → **number**]
(5)	⇒ **(number** − *exp***)** *op exp*	[*op* → **-**]
(6)	⇒ **(number - number)** *op exp*	[*exp* → **number**]
(7)	⇒ **(number - number) *** *exp*	[*op* → *****]
(8)	⇒ **(number - number) * number**	[*exp* → **number**]

Rightmost Derivation of (**number** - **number**)*number

(1)	*exp* ⇒ *exp op exp*	[*exp* → *exp op exp*]
(2)	⇒ *exp op* **number**	[*exp* → **number**]
(3)	⇒ *exp* * **number**	[*op* → *****]
(4)	⇒ **(** *exp* **) * number**	[*exp* → **(** *exp* **)**]
(5)	⇒ **(** *exp op exp* **) * number**	[*exp* → *exp op exp*]
(6)	⇒ **(***exp op* **number) * number**	[*exp* → **number**]
(7)	⇒ **(***exp* **- number) * number**	[*op* → **-**]
(8)	⇒ **(number - number) *number**	[*exp* → **number**]

Parse Trees

Definition

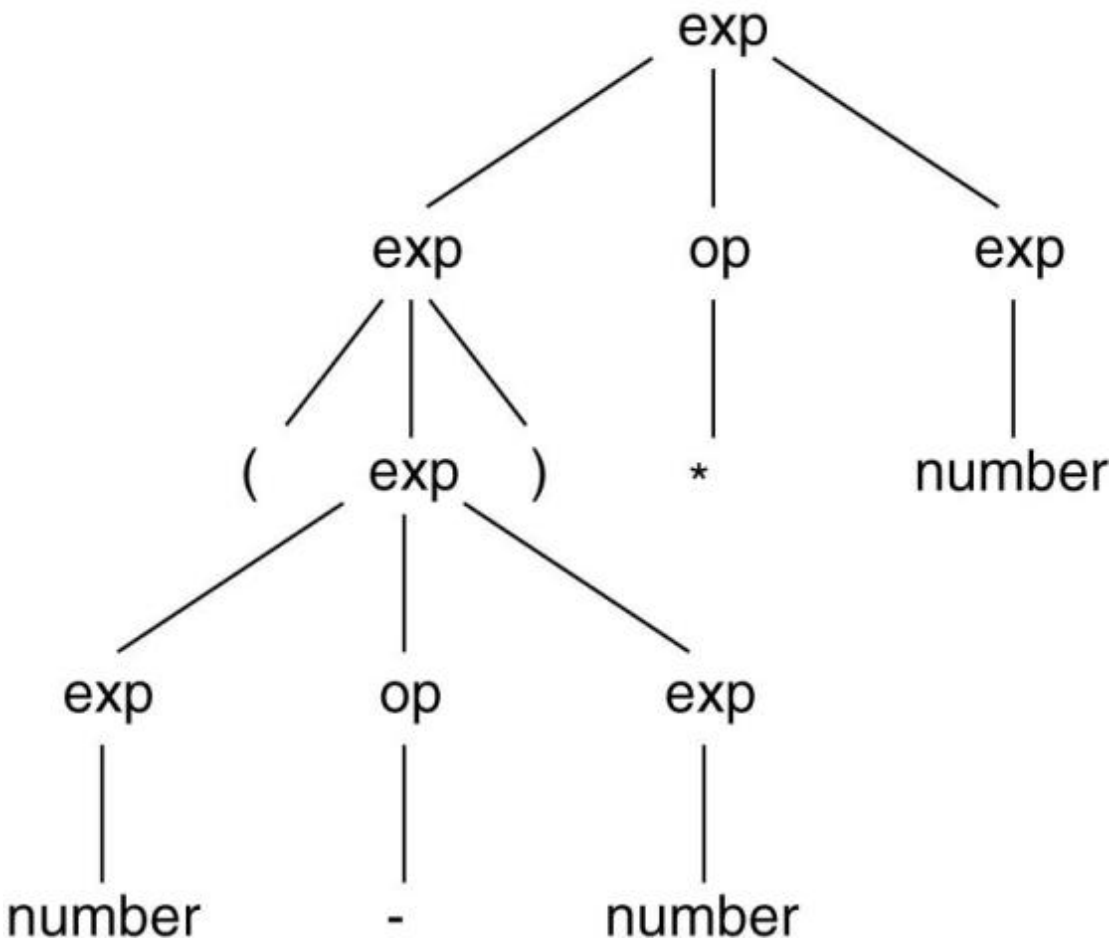
A **parse Tree** is a labeled tree representation of a derivation that filters out the order in which productions are applied to replace nonterminals.

- The interior nodes are labeled by nonterminals
- The leaf nodes are labeled by terminals
- The children of each internal node A are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.

- Since a parse tree ignores variations in the order in which symbols in sentential forms are replaced, there is a many-to-one relationship between derivations and parse tree.

Leftmost and Rightmost Derivations

The following is a parse tree for these two derivations discussedhere.



Ambiguous Grammars

Definition

A grammar that produces more than one parse tree for some sentence is said to be *ambiguous*. Such a grammar is called *ambiguous grammar*.

Put another way,

- If a grammar has more than one leftmost derivation for a single sentential form, the grammar is ambiguous.
- If a grammar has more than one rightmost derivation for a single sentential form, the grammar is ambiguous

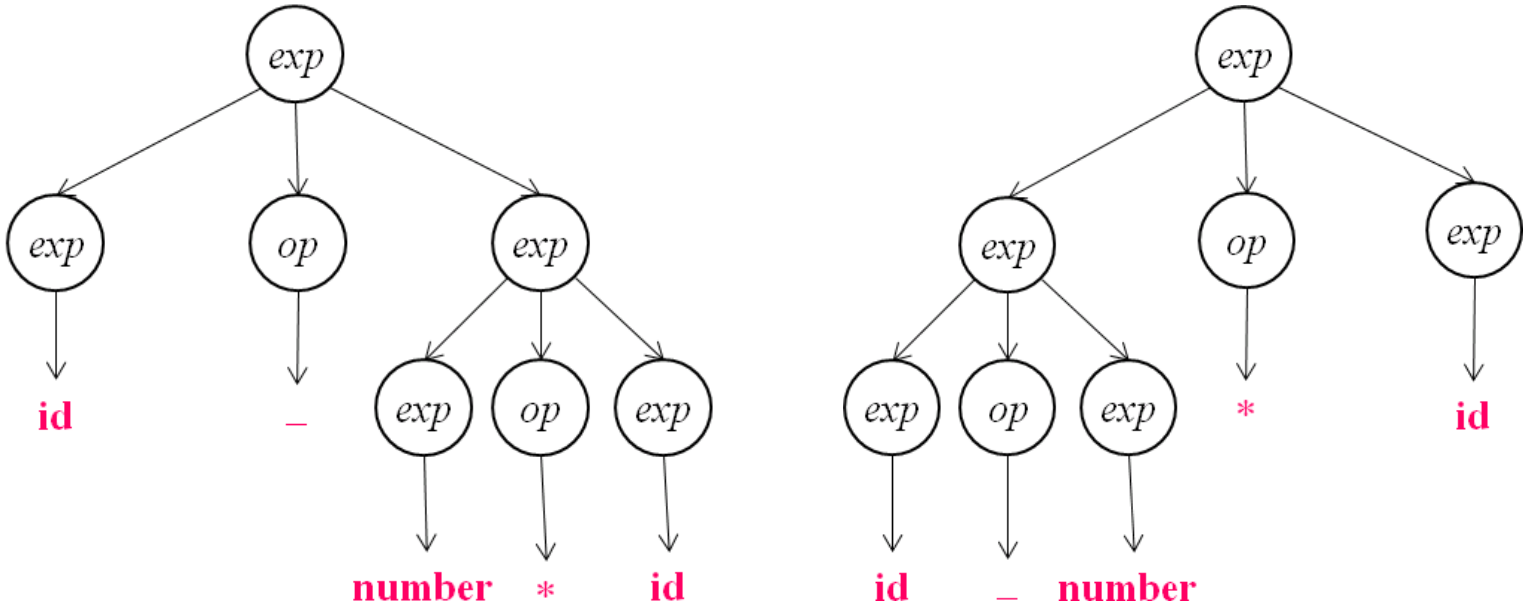
Ambiguous Grammars

The grammar:

1 *exp* → *exp op exp* | **id** | **id**

2 *op* → + | - | * | /

are ambiguous because there are two different parse trees for sentence: **id** - **number*id**



Solving Ambiguity

There are two basic methods to deal with ambiguities.

Approach 1: Disambiguating Rule

State a rule that specifies in each ambiguous case which of the parse trees is the correct one. Such a rule is called a **disambiguating rule**.

- Advantage: No need to change the grammar itself
- Disadvantage: the syntactic structure of the language is no longer given by the grammar alone.

Approach 2: Rewriting Grammar

Change the grammar into a form that forces the construction of the correct parse tree, thus removing the ambiguity.

Precedence and Associativity

Ambiguous Grammar

1 $exp \rightarrow exp\ op\ exp \mid \mathbf{id} \mid \mathbf{id}$

2 $op \rightarrow + \mid - \mid * \mid /$

To use Approach 1 to remove ambiguity from the above ambiguous grammar, the following disambiguating rules are defined:

- all operators (+, -, *, /) are left associative.
- + and - have the same precedence
- * and / have the same precedence
- * and / have higher precedence than + and -.

Based on these rules, which parse tree inthis slideis correct?

Precedence and Associativity

Ambiguous Grammar

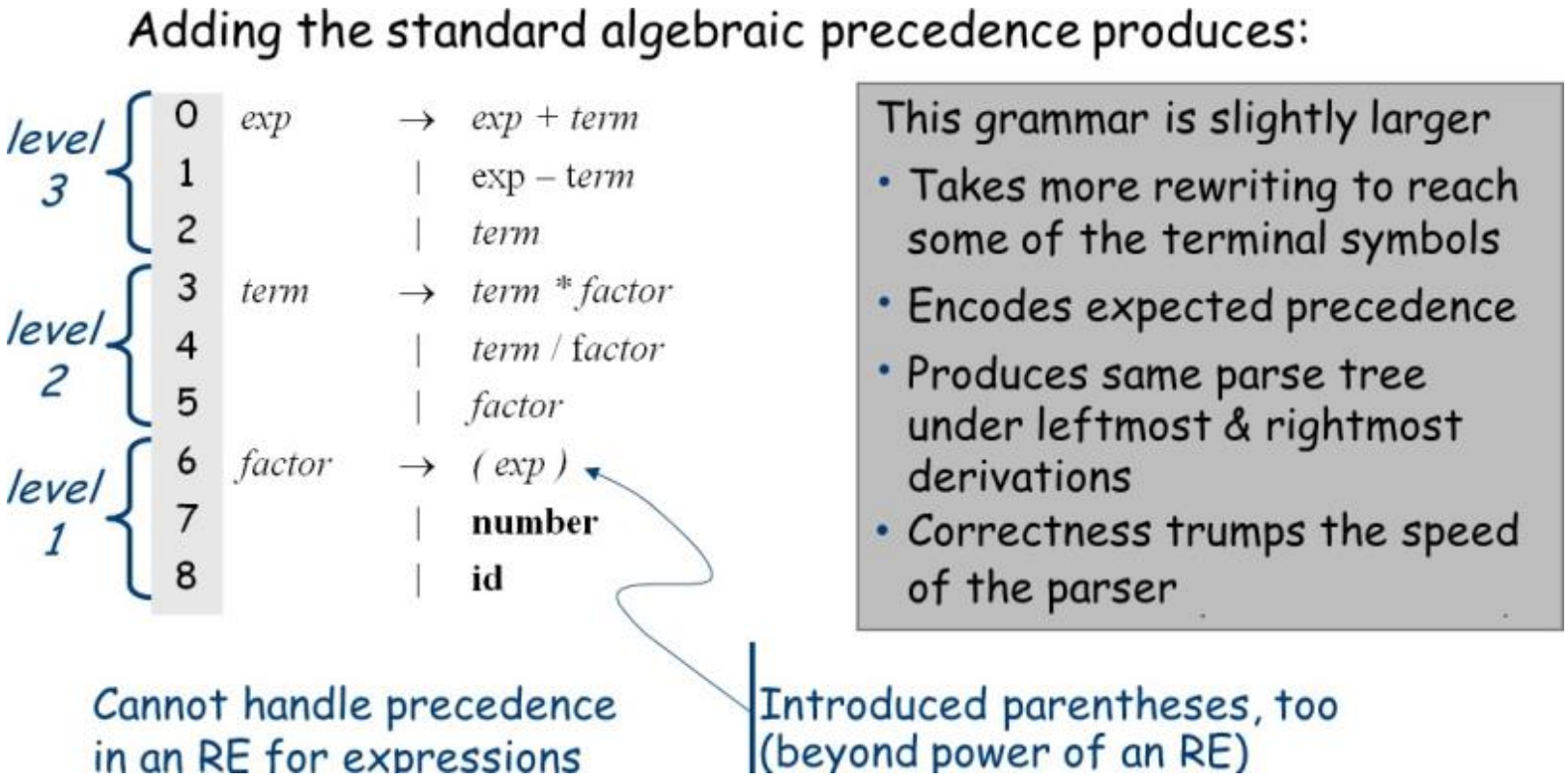
1. $exp \rightarrow exp\ op\ exp \mid \mathbf{id} \mid \mathbf{id}$
2. $op \rightarrow + \mid - \mid * \mid /$

We can add precedence to the above ambiguous grammar to remove ambiguity. To add precedence:

- Group Operators into Precedence Levels
- Create a nonterminal for each level of precedence
- Make operators left, right, or none associative. Position of the recursion relative to the operator dictates the associativity
  - Left (right) recursion → left (right) associativity
  - None: Don't be recursive, simply reference next higher precedence non-terminal on both sides of operator
- Isolate the corresponding levels of the grammar
- Force the parser to recognize high precedence subexpressions first

Precedence and Associativity

The figure below demonstrates how to add precedence to a grammar.

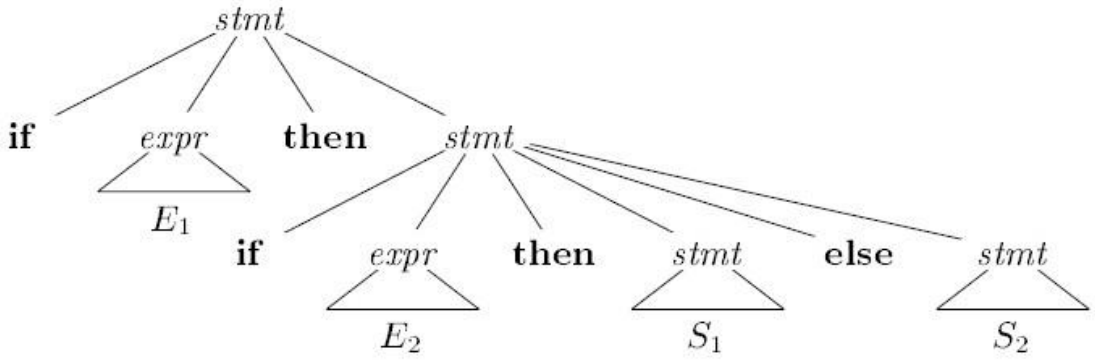


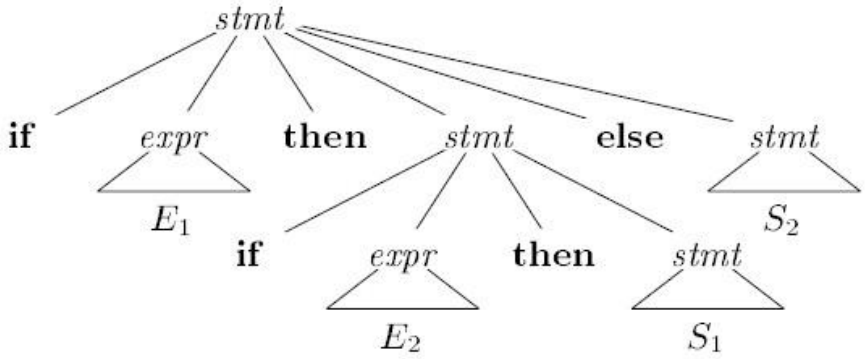
Dangling Else Problem

Dangling Else Grammar

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	**other**

The above grammar is ambiguous since the string
if E_1 **then if** E_2 **then** S_1 **else** S_2 has the two parse trees shown below.





Dangling Else Problem

Two ways to solve the dangling else problem

Approach 1: Create the following disambiguating rule

- Match each **else** with the closest unmatched **then**.

Approach 2: Rewriting the grammar so that the disambiguating rule can be incorporated directly into the grammar.

Dangling Else Problem

The following explain the idea to rewrite the dangling-else grammar to remove the ambiguity.

- A statement appearing between a **then** and an **else** must be ”matched”; that is, the interior statement must not end with an unmatched or open **then**.
- A matched statement is either an **if-then-else** statement containing no open statements or it is any other kind of unconditional statement.

Rewritten Grammar without Ambiguity

stmt	→	*matched_stmt*
	|	*open_stmt*
matched_stmt	→	**if** *expr* **then** *matched_stmt* **else** *matched_stmt*
	|	**other**
open_stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *matched_stmt* **else** *open_stmt*

Dangling Else Problem

With regarding to the dangling else problem

- Rewrite the grammar is usually not taken. Instead, the disambiguating rule is preferred.
  - The principal reason is that parsing methods are easy to configure in such a way that the most closely nested rule is obeyed.
  - Another reason is the added complexity of the new grammar.

Class Problem

Ambiguous Grammar

S	$\rightarrow$	S + S
	|	S - S
	|	S * S
	|	S / S
	|	(S)
	|	-S
	|	S ^S
	|	**number**

Precedence (high to low)

(), unary -
^
*, /
+, -

Associativity

ˆ is right-associative
rest are left-associative


THANK YOU