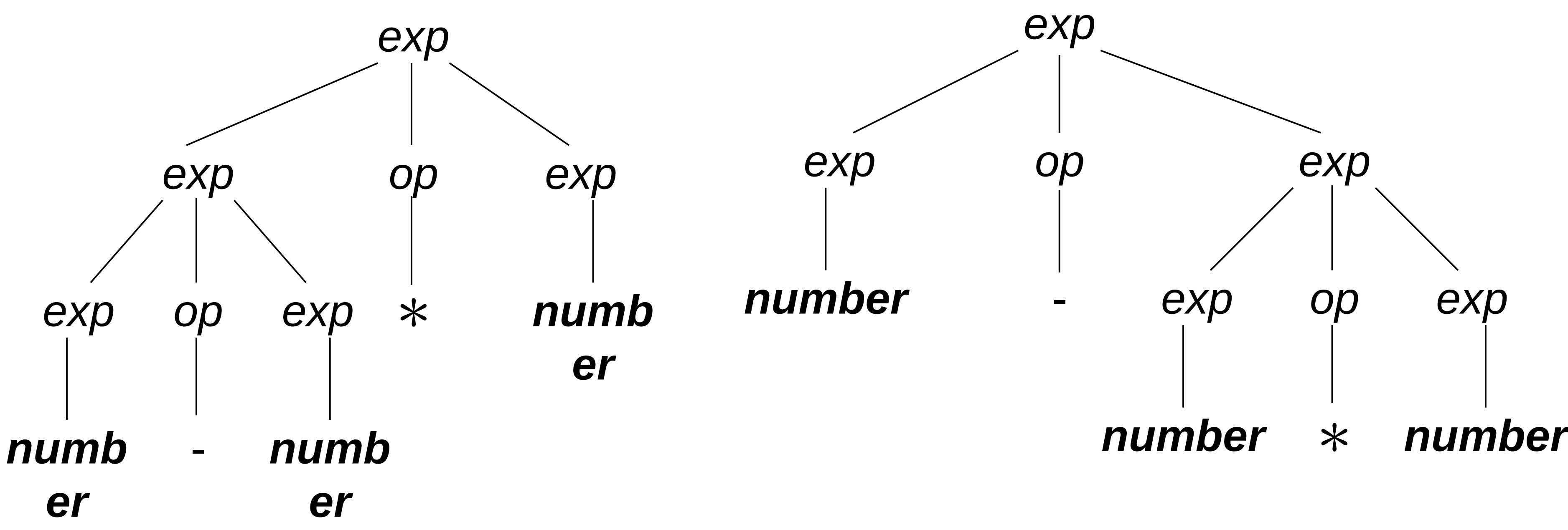
AMBIGUITY

3.4.1 Ambiguous Grammars

- Parse trees and syntax trees uniquely express the structure of syntax, as do leftmost and rightmost derivations, but not derivations in general.
- Unfortunately, it is possible for a grammar to permit a string to have more than one parse tree.
- Consider, for example, the simple integer arithmetic grammar we have been using as a standard example

- *exp* → *exp op exp* | (*exp*) | ***number***
- *op* → + | - | *
- *exp*
- and consider the string 34-3*42. This string has two different parse trees



corresponding to the two leftmost derivations

exp	⇒ *exp op exp*	[*exp* → *exp op exp*]
	⇒ *exp op exp op exp*	[*exp* → *exp op exp*]
	⇒ ***number*** *op exp op exp*	[*exp* → ***number***]
	⇒ ***number*** **–** *exp op exp*	[*op* → -]
	⇒ ***number*** **–** ***number*** *op exp*	[*exp* → ***number***]
	⇒ ***number*** **–** ***number*** * *exp*	[*op* → *]
	⇒ ***number*** **–** ***number*** * ***number***	[*exp* → ***number***]

AND

exp	⇒	*exp op exp*	[*exp* → *exp op exp*]
	⇒	***number*** *op exp*	[*exp* → ***number***]
	⇒	***number*** - *exp*	[*exp* → **-**]
	⇒	***number*** – *exp op exp*	[*op* → *exp op exp*]
	⇒	***number*** – ***number*** *op exp*	[*exp* → ***number***]
	⇒	***number*** – ***number*** * *exp*	[*op* → *]
	⇒	***number*** – ***number*** * ***number***	[*exp* → ***number***]

- Example 4.5.
- Let us again consider the arithmetic expression grammar (4.7), with which we have been dealing. The sentence **id + id * id** has the two distinct leftmost derivations :

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned} \qquad \begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

with the two corresponding parse trees shown in fig. 4.3.

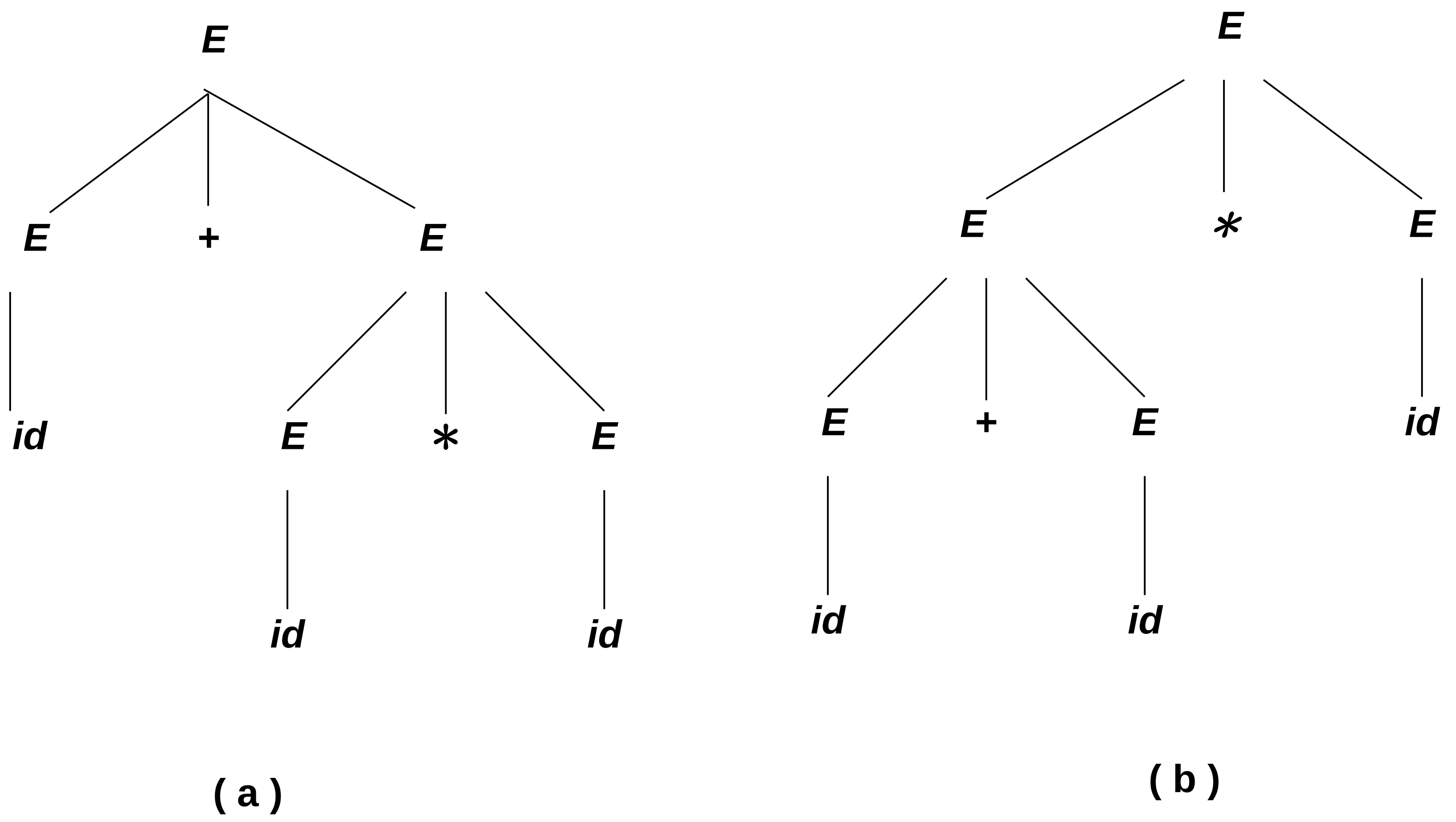


Fig.4.3. Two parse trees for **id + id * id**

- This example shows the two things we must do in order to prove that a grammar generates a language L. We must show that every sentence generated by the grammar is in L, and we must show that every string in L can be generated by the grammar.

- We have already seen a grammar for arithmetic expression. the following grammar fragment (4.11) generates conditional statements.

stat → **if** cond **then** stat

| **if** cond **then** stat **else** stat

| other-stat

(4.11)

- Thus the string

  if C_1 **then** S_1

  else if C_2 **then** S_2 **else** S_3

would have the parse tree shown in Fig. 4.4.

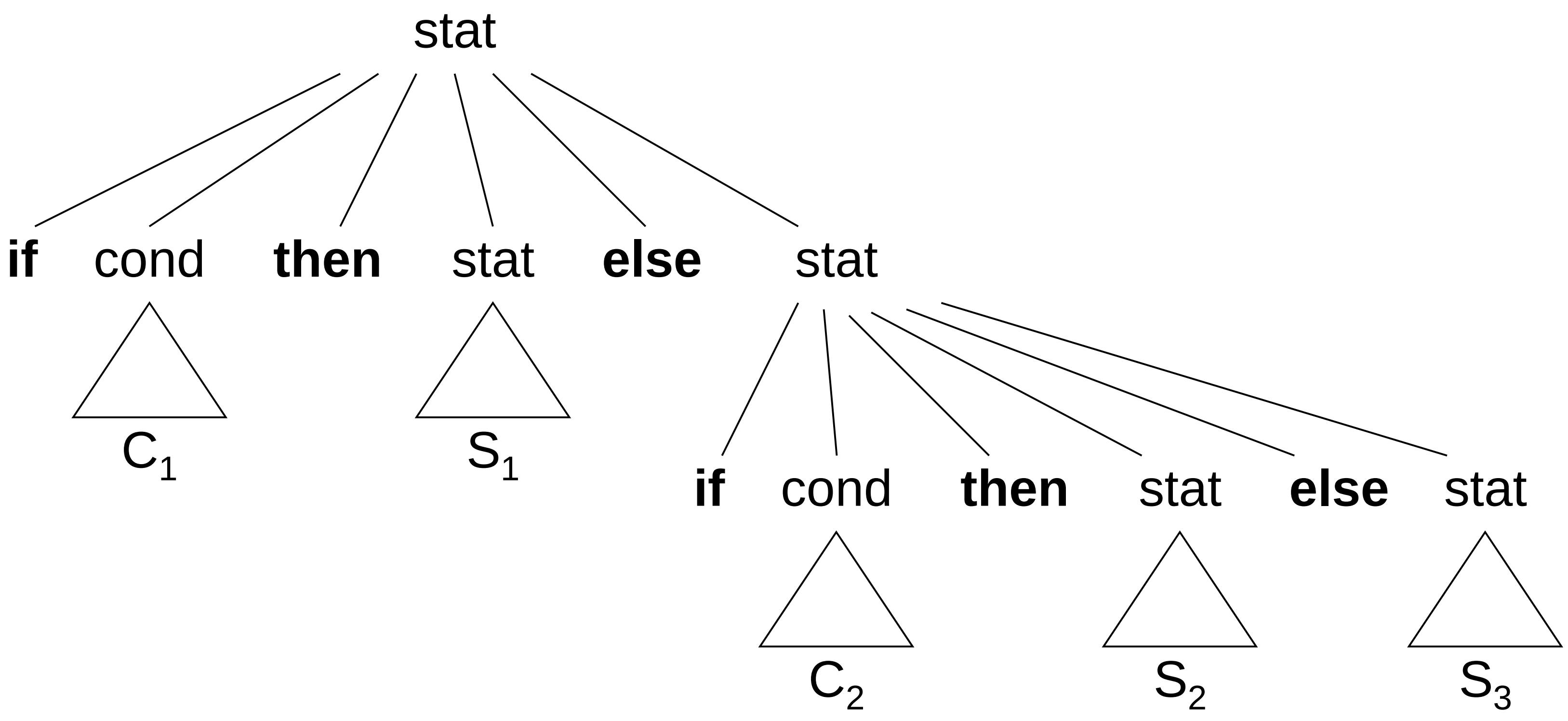


Fig. 4.4. Parse tree

- Grammar (4.11) is ambiguous, however, since the string

  if C_1 **then if** C_2 **then** S_1 **else** S_2 (4.12)

  has the two parse trees shown in Fig. 4.5.

- In all programming languages with conditional statement of this form, the first parsing is preferred. The general rule is “ Each **else** is to be matched with the closest previous unmatched **then** ”.

- We could incorporate this disambiguating rule directly into the grammar if we wish. for example, we could rewrite grammar (4.11) as the following

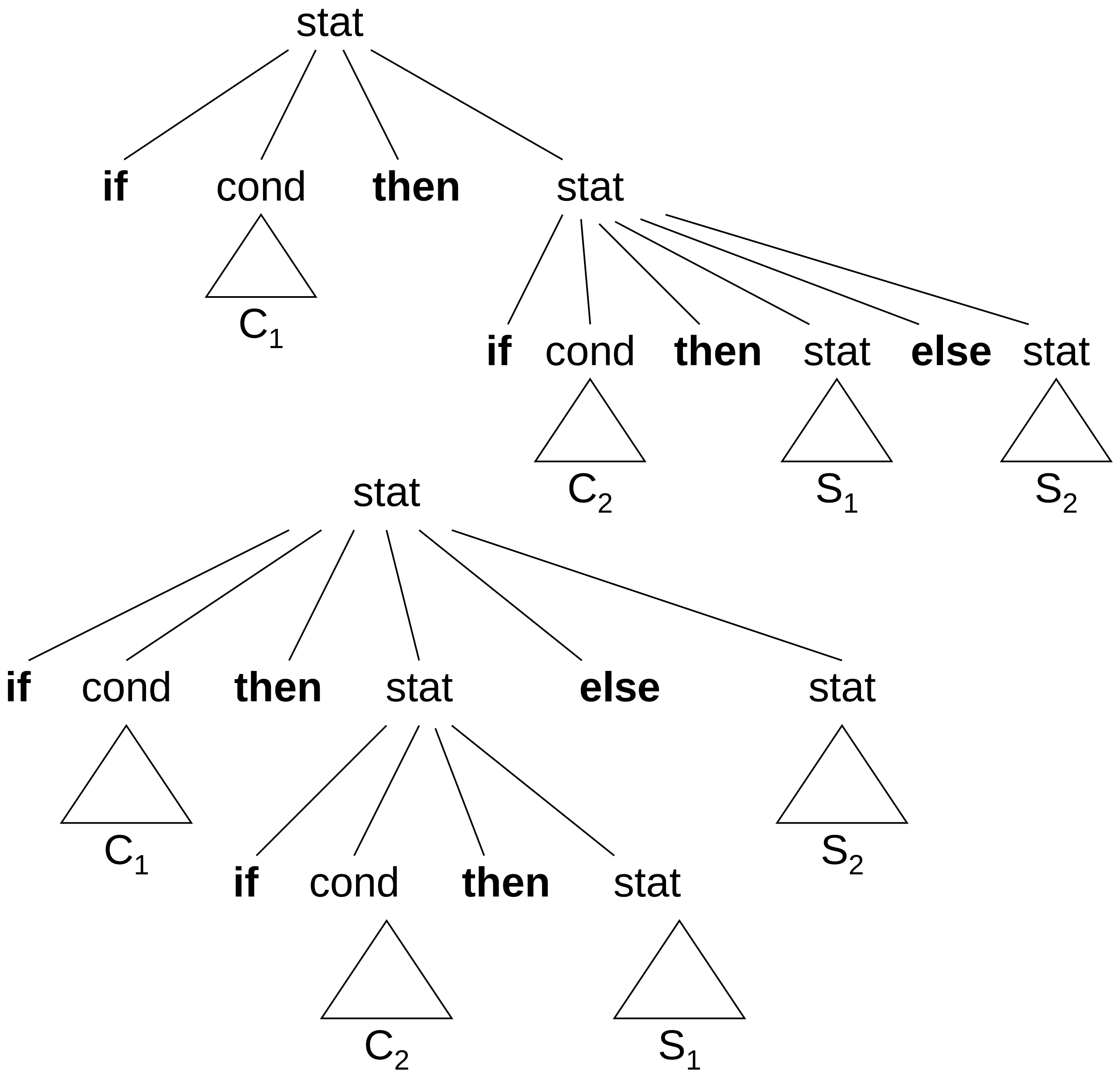


Fig.4.5. Two parse trees for ambiguous sentence.

Unambiguous Grammar

stat	→	matched-stat
		| unmatched-stat
matched-stat	→	**if** cond **then** matched-stat **else** matched-stat
		| other-stat
unmatched-stat	→	**if** cond **then** stat
		| **if** cond **then** matched-stat **else** unmatched-stat

- This grammar generates the same set of strings as (4.11), but it allows only one parsing for string (4.12), namely the one stat associates each **else** with the previous unmatched **then**.

The Dangling Else Problem

- Consider the grammar from Example 3.4 (page 103) :

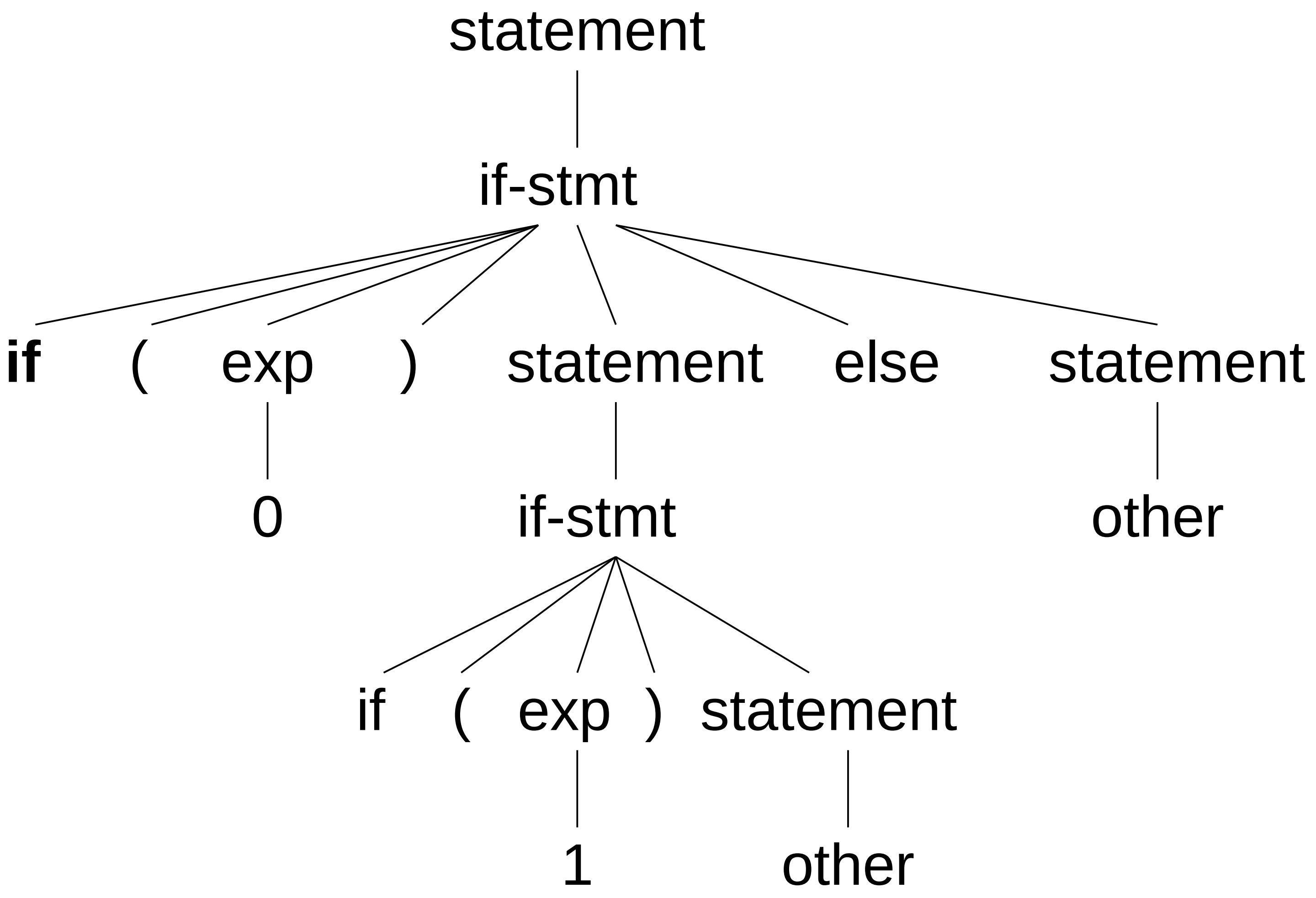
statement	→	if-stmt | other
if-stmt	→	if (exp) statement
		| if (exp) statement else statement
exp	→	0 | 1

- The grammar is ambiguous as a result of the optional else. To see this, consider the string

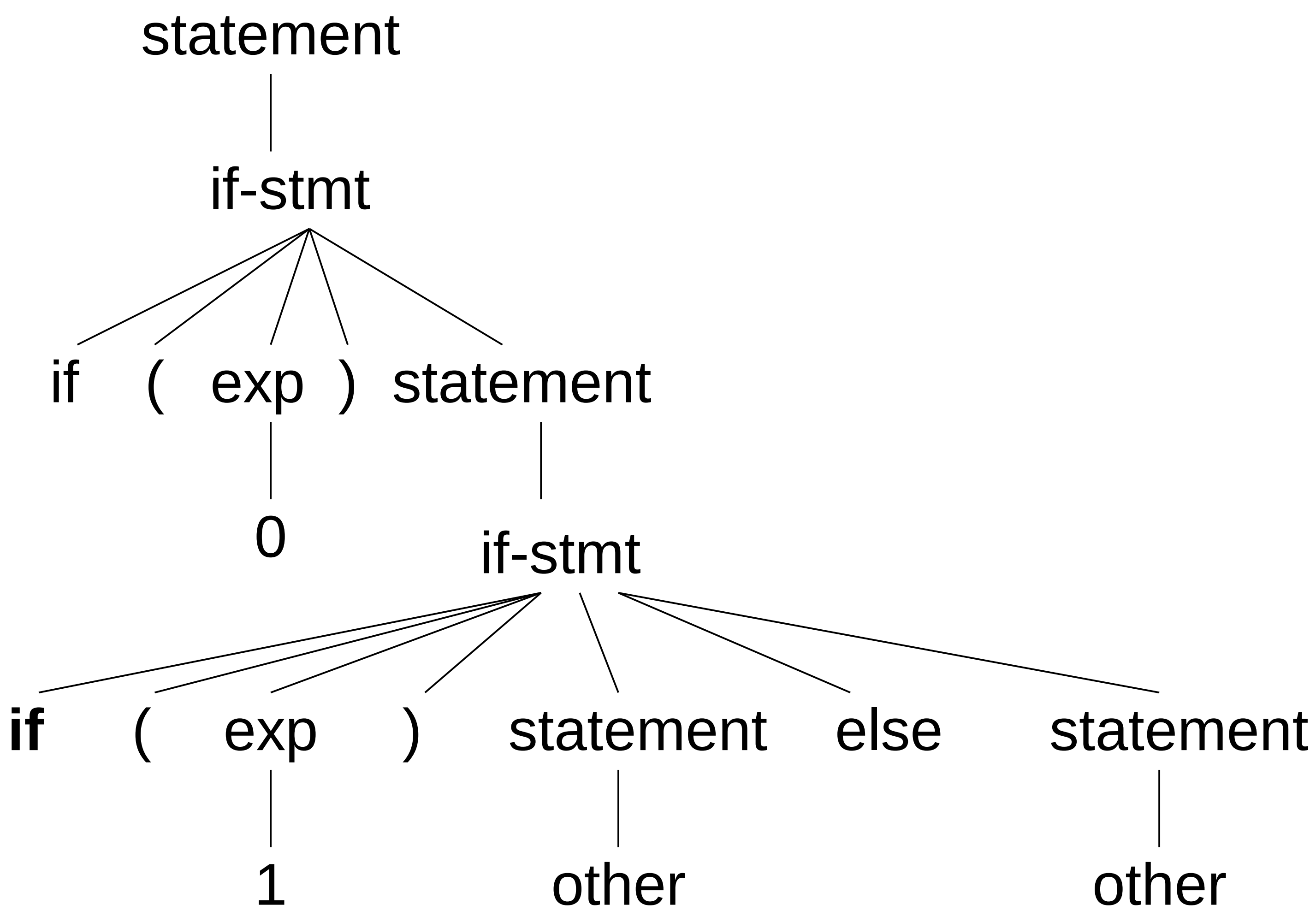
  if (0) if (1) other else other

  This string has the two parse trees :

  (show in next page)


statement
if-stmt
if
(
exp
)
statement
else
statement
0
if-stmt
other
if
(
exp
)
statement
1
other

AND


statement
if-stmt
if
(
exp
)
statement
0
if-stmt
if
(
exp
)
statement
else
statement
1
other
other

- Which one is correct depends on whether we want to associate the single else-part with the first or the second if-statement :
  - the first parse tree associates the else-part with the first if-statement
  - the second parse tree associates it with the second if-statement.
- This ambiguity is called the **dangling else problem**.
- To see which parse tree is correct, we must consider the implications for the meaning of the if-statement. To get a clearer idea of this,

consider the following piece of C code

```
if ( x != 0 )
    if ( y == 1/x ) ok = TRUE ;
  else z = 1/x ;
```

- In this code, whenever **x** is 0, a division by zero error will occur if the else-part is associated with the first if-statement.

- Thus, the implication of this code (and indeed the implication of the indentation of the else-part) is that an else-part should always be associated with the nearest if-statement that does not yet have an associated else-part.

- This disambiguating rule is called the **most closely nested rule** for the dangling else problem, and it implies that the second parse tree above is the correct one.

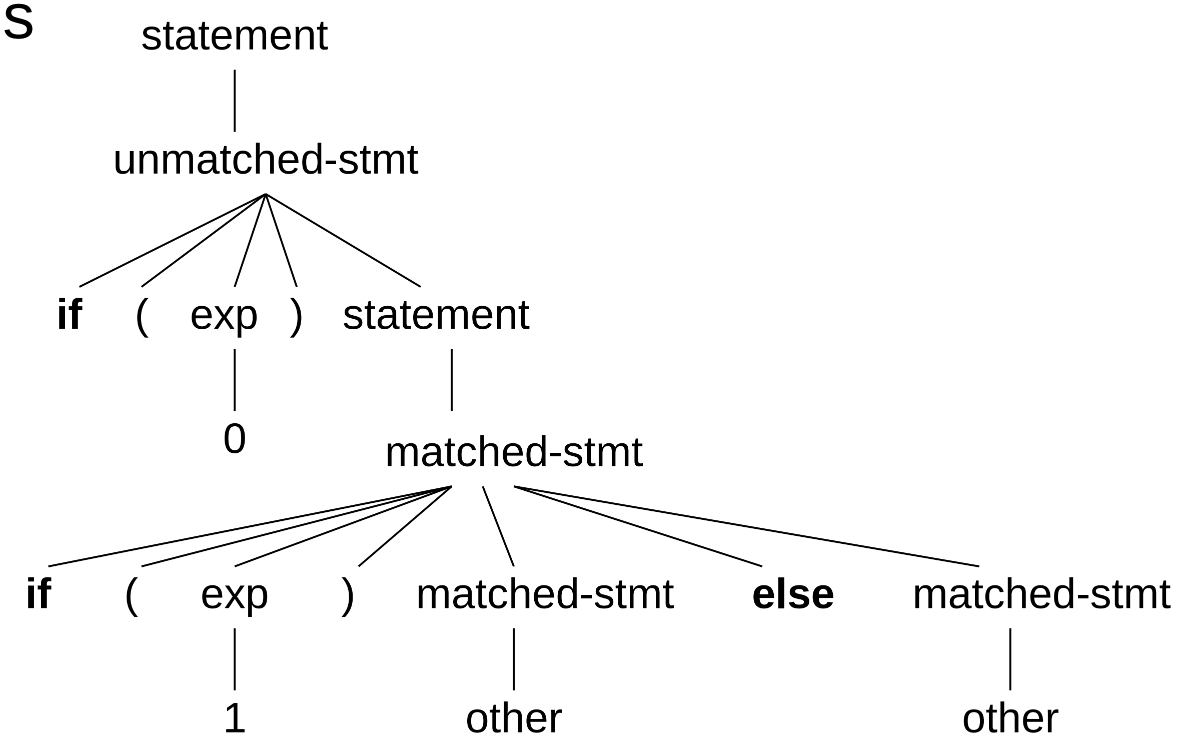
- Note that, if we wanted we *could* associate the else-part with the first if-statement by using brackets {…} in C, as in

```
if ( x != 0 )
    { if ( y== 1/x ) ok = TRUE ; }
else z = 1/x ;
```

- A solution to the dangling else ambiguity in the BNF itself is more difficult than the previous ambiguities we have seen. A solution is as follows :

statement	→	matched-stmt
		| unmatched-stmt
matched-stmt	→	**if** (exp) matched-stmt **else** matched-stmt
		| **other**
unmatched-stmt	→	**if** (exp) statement
		| **if** (exp) matched-stmt **else** unmatched-stmt
exp	→	0 | 1

- This works by permitting only a matched-stmt to come before an **else** in an if-statement, thus forcing all else-parts to be matched as soon as possible. For instance, the associated parse tree for our sample string now becomes



which indeed associates the else-part with the second if-statement.

```
if x /= 0 then
    if y = 1/x then ok := true;
    else z := 1/x;
    end if;
end if;
if x /= 0 then
    if y = 1/x then ok := true;
    end if;
else z := 1/x;
end if;
```

if-stmt → **if** condition **then** statement-sequence **end if**
| **if** condition **then** statement-sequence
else statement-sequence **end if**

Precedence and Associativity

- To handle the precedence of operations in the grammar, we must group the operators into groups of equal precedence, and for each precedence we must write a different rule.
- For example, the precedence of multiplication over addition and subtraction can be added to our simple expression grammar as follows :

exp	→	exp addop exp | term
addop	→	+ | -
term	→	term mulop term | factor
mulop	→	*
factor	→	(exp) | **number**

- In this grammar, multiplication is grouped under the *term* rule, while addition and subtraction are grouped under the *exp* rule.

- Since the base case for an *exp* is a *term*, this means that addition and subtraction will appear “higher”(that is, closer to the root) in that parse and syntax trees, and thus receive lower precedence.

- Such a grouping of operator into different precedence levels is a standard method in syntactic specification using BNF.

- We call such a grouping a **precedence cascade**.

- This last grammar for simple arithmetic expressions still does not specify the associativity of the operators and is still ambiguous.

- The reason is that the recursion on both sides of the operator allows either side to match repetitions of the operator in a derivation (and, hence, in the parse and syntax trees).

- The solution is to replace one of the recursions with the base case, forcing the repetitive matches on the side with the remaining recursion.

- Thus, replacing the rule

  exp → *exp addop exp* | *term*

  by

  exp → *exp addop term* | *term*

  makes addition and subtraction left associative,

- while writing

  exp → term addop exp | term

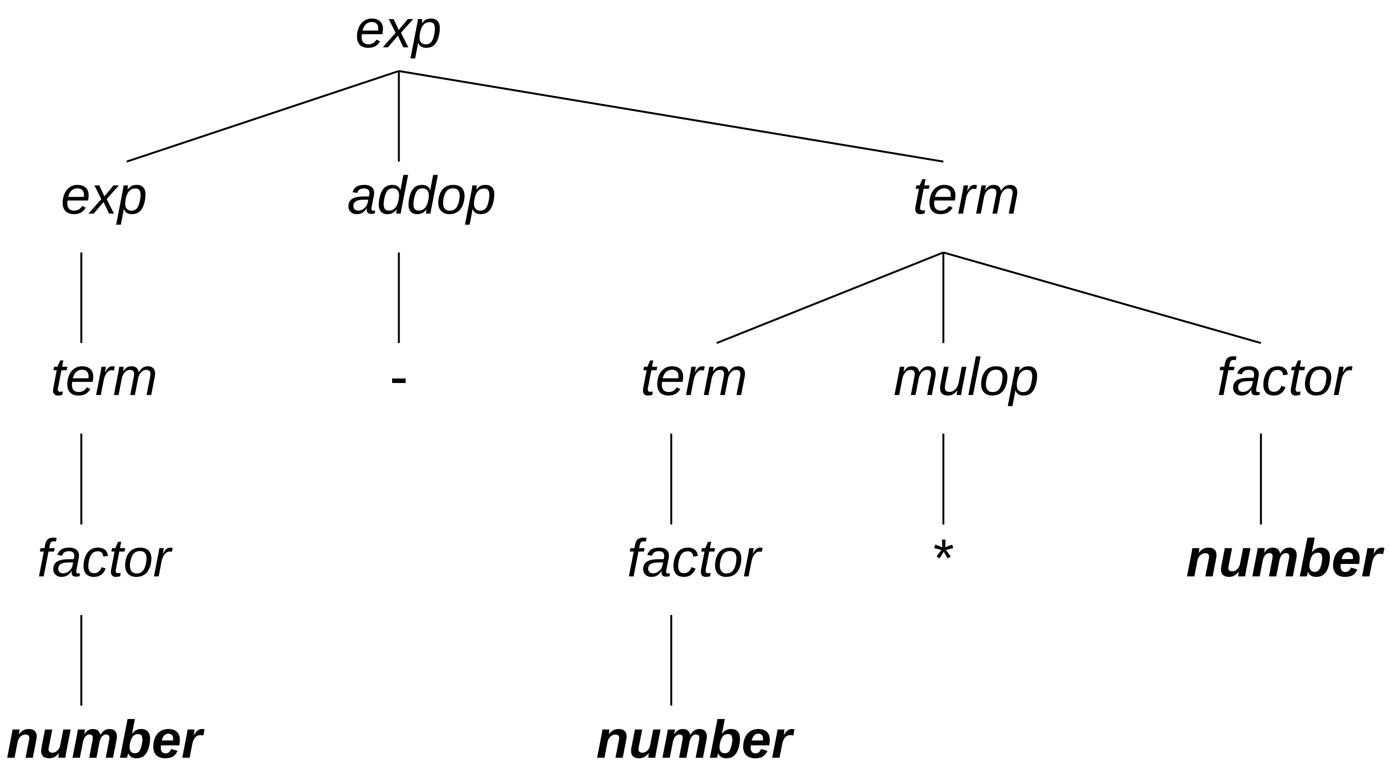
  makes them right associative.

- In other words, a left recursive rule makes its operators associate on the left, while a right recursive rule makes them associate on the right.

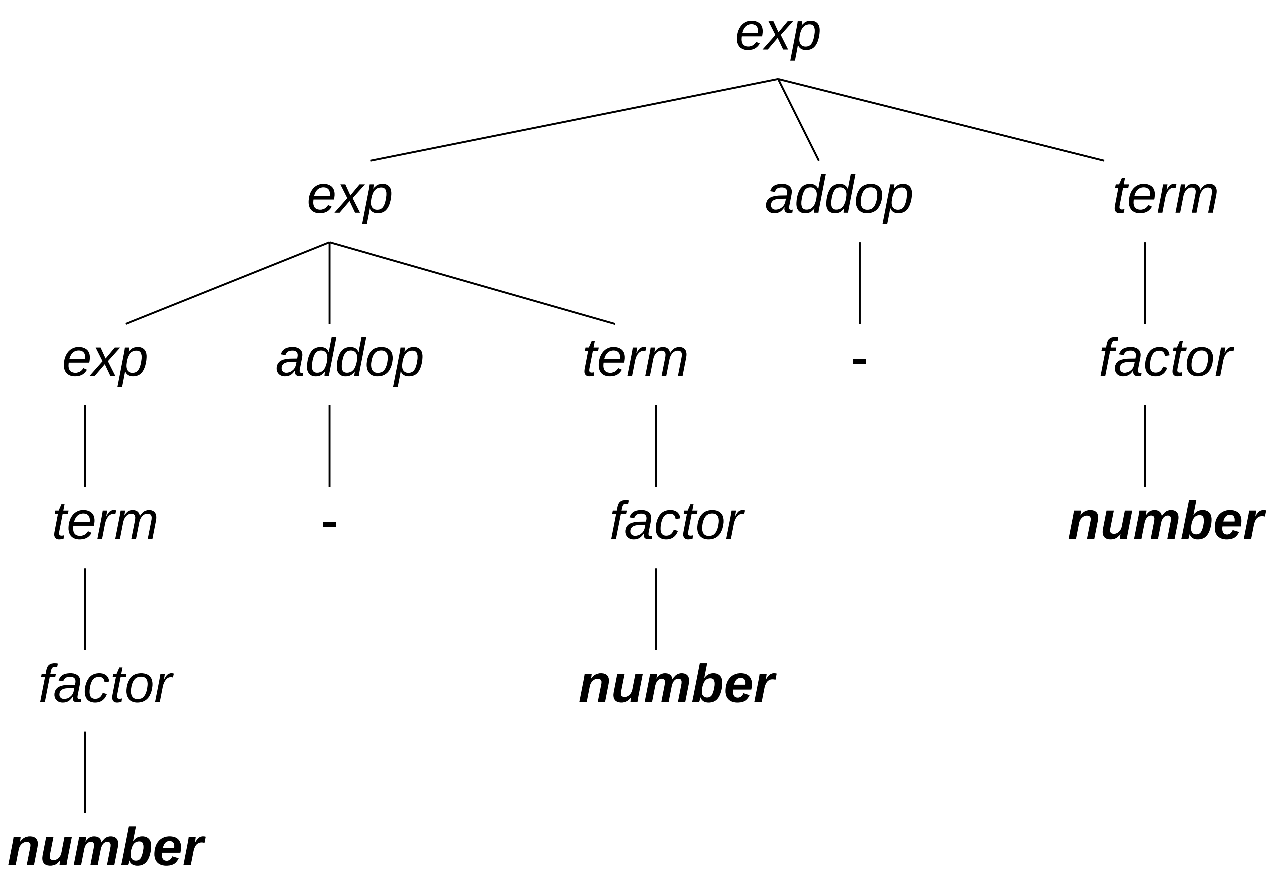
- To complete the removal of ambiguity in the BNF rules for our simple arithmetic expressions, we write the rules to make all the operations left associative :

exp	→	*exp addop term* | *term*
addop	→	+ | -
term	→	*term mulop factor* | *factor*
mulop	→	*
factor	→	(*exp*) | ***number***

- Now the parse tree for the expression 34-3*42 is



- The parse tree for the expression 34-3-42 is



- Note that the precedence cascades cause the parse trees to become much more complex.
- The syntax trees, however, are not affected.

- Example 4.6.

  Consider the following grammar for arithmetic expressions involving +, -, *, /, and $\uparrow$ (exponentiation)

$$
\begin{aligned}
E \quad \rightarrow \quad & E + E \mid E - E \\
& \mid E * E \mid E / E \\
& \mid E \uparrow E \mid (E) \\
& \mid -E \mid id
\end{aligned}
$$

(4.9)

- This grammar, like (4.7), is ambiguous. However, we can disambiguate both these grammars by specifying the associativity and precedence of the arithmetic operators.

- Suppose we wish to give the operators the following precedences in decreasing order :

  - (unary minus)

  ↑

  * /

  + -

- Suppose further we wish ↑ to be right-associative [e.g., a ↑ b ↑ c is to mean a ↑ (b ↑ c)] and the other binary operators to be left-associative [e.g., a – b – c is to mean (a – b) – c].

- These precedences and associativities are the ones customarily used in mathematics and in many, but not all, programming languages [e.g., a + - b ↑ c + d * e is interpreted as precedence of operators are sufficient to disambiguate both grammars (4.7) parse tree that groups operands of operators according to these associativity and precedence rules.

- For example, Fig. 4.3 (b) would not be a valid parse tree for **id** + **id** * **id** according to these rules because there + appears to have higher precedence than *.

- We can also rewrite a grammar to incorporate the associativity and precedence rules into the grammar itself.
- To illustrate what is involved, let us transform (4.9) into an equivalent unambiguous grammar that obeys the associativity and precedence rules given above.
- We begin by introducing one nonterminal for each precedence level.
- A subexpression that is essentially indivisible we shall call an *element*.
- An element is either a single identifier or a parenthesized expression.

- We therefore have the productions

  element → (expression) | **id**

- Next, we introduce the category of *primaries*, which are elements with zero or more of the operator of highest precedence, the unary minus. The rule for primary is :

  primary → - primary | element

- Then we construct *factors* as sequences of one or more primaries connected by exponentiation signs. That is :

$$\text{factor} \rightarrow \text{primary} \uparrow \text{factor} \mid \text{primary}$$

- Note that the choice of the right side primary $\uparrow$ factor rather than factor $\uparrow$ primary forces expressions like $a \uparrow b \uparrow c$ to group from the right as $a \uparrow (b \uparrow c)$.

- Then we introduce *terms*, which are sequences of one or more factors connected by the *multiplicative operators*, namely * and /, and finally *expressions*, which are sequences of one or more terms connected by the *additive operators*, + and binary -.

- The productions for term are

```
term  →  term * factor
         | term / factor
         | factor
```

- These productions cause terms to be grouped from the left [e.g., a * b * c means (a * b) * c]. The final, unambiguous grammar is :

expression	→	expression + term
		| expression – term
		| term
term	→	term * factor
		| term / factor
		| factor
factor	→	primary ↑ factor
		| primary
primary	→	- primary
		| element
element	→	(expression)
		| **id**

EXTENDED NOTATIONS : EBNF AND SYNTAX DIAGRAMS

3.5.1 EBNF Notation

- Repetitive and optional constructs are already common in programming languages, and thus in BNF grammar rules as well.
- Therefore, it should not be surprising that the BNF notation is sometimes extended to include special notations for these two situations.
- These extensions comprise a notation that is called extended **BNF**, or **EBNF**.

- Consider, first, the case of repetition, such as that of statement sequences. We have seen that repetition is expressed by recursion in grammar rules and that either left or right recursion might be used, indicated by the generic rules

  $A \rightarrow A\ \alpha \mid \beta$ (left recursive)

  and

  $A \rightarrow \alpha\ A \mid \beta$ (right recursive)

  where α and β are arbitrary strings of terminals and nonterminals and where in the first rule β does not begin with A and in the second β does not end with A.

- It would be possible to use the same notation for repetition that regular expressions use, namely, the asterisk * (also called Kleene closure in regular expressions). Then these two rules would be written as the nonrecursive rules

  $$A \rightarrow \beta \alpha^*$$

  and

  $$A \rightarrow \alpha^* \beta$$

- Instead, EBNF opts to use curly brackets {…} to express repetition (thus making clear the extent of the string to be repeated), and we write

  $$A \rightarrow \beta \{ \alpha \}$$

  and

  $$A \rightarrow \{ \alpha \} \beta$$

  for the rules.

- The problem with any repetition notation is that it obscures how the parse tree is to be constructed, but, as we have seen, we often do not care. Take for example, the case of statement sequences (Example 3.9). We wrote the grammar as follows, in right recursive form :

  stmt-sequence → *stmt*; *stmt-sequence* | *stmt*

  stmt → **s**

- This rule has the form A → α A | β, with A = *stmt-sequence*, α = *stmt*. In EBNF this would appear as

  stmt-sequence → { *stmt* ; } *stmt*
  (right recursive form)

- We could equally as well have used a left recursive rule and obtained the EBNF

  stmt-sequence → *stmt* { ; *stmt* }
  (left recursive form)

- In fact, the second form is the one generally used (for reasons we shall discuss in the next chapter)

- A more significant problem occurs when the associativity matters, as it does for binary operations such as subtraction and division. For example, consider the first grammar rule in the simple expression grammar of the previous subsection :

  $exp \rightarrow exp\ addop\ term \mid term$

- This has the form $A \rightarrow A\ \alpha \mid \beta$, with $A = exp$, $\alpha = addop\ term$, and $\beta = term$. Thus, we write this rule in EBNF as

  $exp \rightarrow term\ \{\ addop\ term\ \}$

- We must now also assume that this implies left associativity; although the rule itself does not explicitly state it. We might assume that a right associative rule would be implied by writing

  exp → { *term addop* } *term*

  but this is not the case.

- Instead, a right recursive rule such as

  stmt-sequence → *stmt*; *stmt-sequence* | *stmt*

  is viewed as being a *stmt* followed by an optional semicolon and *stmt-sequence*.

- Optional constructs in EBNF are indicated by surrounding them with square brackets [...].

- This is similar in spirit to the regular expression convention of putting a question mark after an optional part, but has the advantage of surrounding the optional part without requiring parentheses.

- For example, the grammar rules for if-statement with optional else-parts (Examples 3.4 and 3.6) would be written as follows in EBNF :

statement	→ *if-stmt* | other
if-stmt	→ if (*exp*) *statement* [else *statement*]
exp	→ 0 | 1

- Also, a right recursive rule such as

  stmt-sequence → *stmt* ; *stmt-sequence* | *stmt*

  is written as

  stmt-sequence → *stmt* [; *stmt-sequence*]

  (contrast this to the use of curly brackets previously to write this rule in recursive form).

- If we wished to write an arithmetic operation such as addition in right associative form, we would write

  exp → *term* [*addop exp*]

  instead of using curly brackets.

Algorithm to Transform Extended BNF Grammars into Standard Form

```
for ( each production $P = A \rightarrow \alpha\ [\ X_1 \dots X_n\ ]\ \beta$ ) {
    Create a new nonterminal, N.
    Replace production P with $P' = A \rightarrow \alpha\ N\ \beta$
    Add the productions : $N \rightarrow X_1 \dots X_n$ and $N \rightarrow \lambda$
}

for ( each production $Q = B \rightarrow \gamma\ \{Y_1 \dots Y_m\}\ \delta$ ) {
    Create a new nonterminal, M.
    Replace production Q with $Q' = B \rightarrow \gamma\ M\ \delta$
    Add the productions : $M \rightarrow Y_1 \dots Y_m\ M$ and $M \rightarrow \lambda$
}
```