Chapter 3: CONTEXT-FREE GRAMMARS AND PARSING –Part2
3.3 Parse Trees and Abstract Syntax Trees

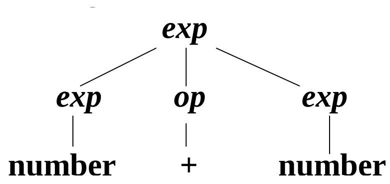
3.3.1 Parse trees
1. Derivation V.S. Structure

- **Derivations do not uniquely represent the structure of the strings**
  - **There are many derivations for the same string.**
- **The string of tokens:**
  - **(*number - number*) * number**
- **There exist two different derivations for above string**

(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*]
----
(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]

2. Parsing Tree

- **A parse tree corresponding to a derivation is a labeled tree.**
  - **The interior nodes are labeled by non-terminals, the leaf nodes are labeled by terminals;**
  - **And the children of each internal node represent the replacement of the associated non-terminal in one step of the derivation.**
- **The example:**
  - ***exp*** **=>** ***exp op exp*** **=> number** ***op exp*** **=> number +** ***exp*** **=> number + number**
- **The example:**
  - ***exp*** **=>** ***exp op exp*** **=> number** ***op exp*** **=> number +** ***exp*** **=> number + number**
- **Corresponding to the parse tree:**



- **The above parse tree is corresponds to the three derivations:**
- **<u>Left most derivation</u>**

(1) ***exp*** **=>** ***exp op exp***

(2) **=>** ***number op exp***

(3) **=>** ***number*** **+** ***exp***

(4) **=>** ***number*** **+** ***number***

- **<u>Right most derivation</u>**

(1) ***exp*** **=>** ***exp op exp***
(2) **=>** ***exp op number***
(3) **=>** ***exp*** **+** ***number***
(4) **=>** ***number*** **+** ***number***

- **<u>Neither leftmost nor rightmost derivation</u>**

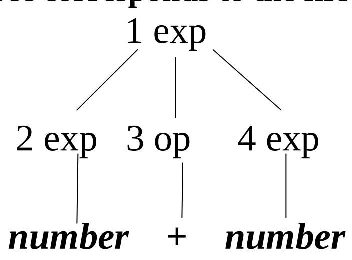
(1) ***exp*** **=>** ***exp op exp***

(2) **=>** ***exp*** **+** ***exp***

(3) **=>** ***number*** **+** ***exp***

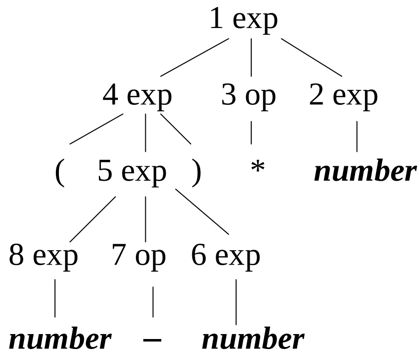
(4) **=>** ***number*** **+** ***number***

- **Generally, a parse tree corresponds to many derivations**
  - **represent the same basic structure for the parsed string of terminals.**
- **It is possible to distinguish particular derivations that are uniquely associated with the parse tree.**
- **<u>A leftmost derivation:</u>**
  - **A derivation in which the leftmost non-terminal is replaced at each step in the derivation.**
  - **Corresponds to the *preorder* numbering of the internal nodes of its associated parse tree.**
- **<u>A rightmost derivation:</u>**
  - **A derivation in which the rightmost non-terminal is replaced at each step in the derivation.**
  - **Corresponds to the *postorder* numbering of the internal nodes of its associated parse tree.**
- **The parse tree corresponds to the first derivation.**



Example: The expression (34-3)*42

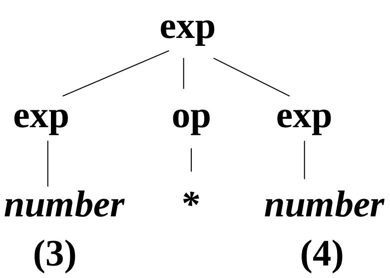
- **The parse tree for the above arithmetic expression**



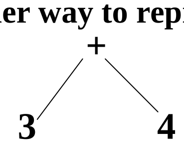
3.3.2 Abstract syntax trees

<u>1. Why Abstract Syntax-Tree</u>

- **The parse tree contains more information than is absolutely necessary for a compiler**
- **For the example: 3*4**

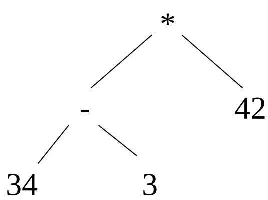


- **<u>The principle of syntax-directed translation</u>**
  - **The meaning, or semantics, of the string 3+4 should be directly related to its syntactic structure as represented by the parse tree.**
- **In this case, the parse tree should imply that the value 3 and the value 4 are to be added.**
- **A much simpler way to represent this same information, namely, as the tree**



Tree for expression (34-3)*42

- **The expression (34-3)*42 whose parse tree can be represented more simply by the tree:**



- **The parentheses tokens have actually disappeared**
  - **still represents precisely the semantic content of subtracting 3 from 34, and then multiplying by 42.**

2. Abstract Syntax Trees or Syntax Trees

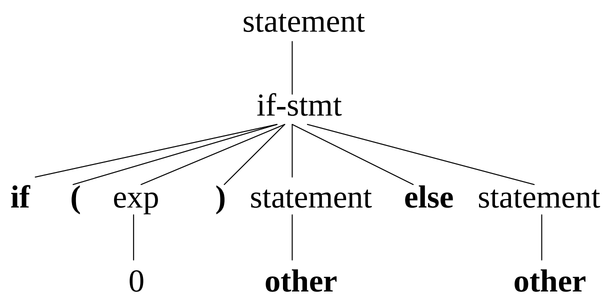
- **Syntax trees represent abstractions of the actual source code token sequences,**
  - **The token sequences cannot be recovered from them (unlike parse trees).**
  - **Nevertheless they contain all the information needed for translation, in a more efficient form than parse trees.**

Examples

- **Example 3.8:**
  - **The grammar for simplified if-statements**

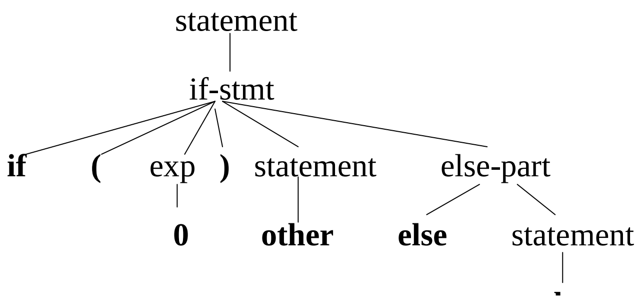
statement* → *if-stmt* | *other
if-stmt* → if (*exp*) *statement* | if (*exp) statement* else s*tatement
***exp* → 0 | 1**

- The parse tree for the string:
  - **if (0) other else other**

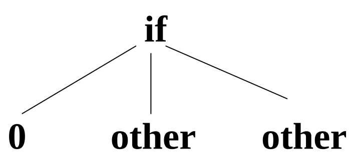


- **Using the grammar of Example 3.6**
  statement* → *if-stmt* | *other
  if-stmt* → if (*exp*) *statement else-part
  ***else-part* → else *statement* | ε**
  exp → 0 | 1

- This same string has the following parse tree:
  - **if (0) other else other**



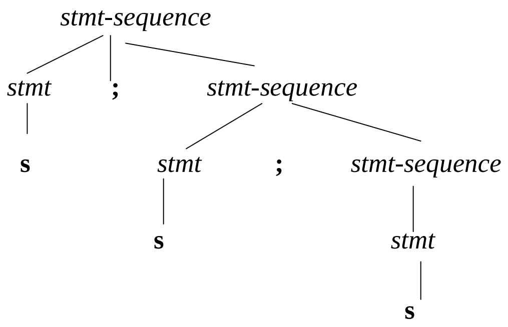
- A syntax tree for the previous string (using either the grammar of Example 3.4 or 3.6) would be:
  - **if (0) other else other**



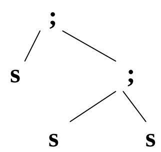
- **Example 3.9:**
  - **The grammar of a sequence of statements separated by semicolons from Example 3.7:**

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

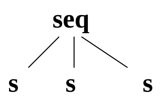
- The string ***s; s; s*** has the following *parse tree* with respect to this grammar:



- A possible syntax tree for this same string is:



- Bind all the statement nodes in a sequence together with just one node, so that the previous syntax tree would become



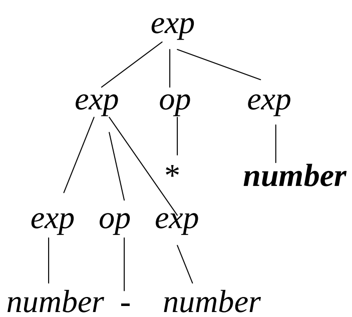
3.4 Ambiguity

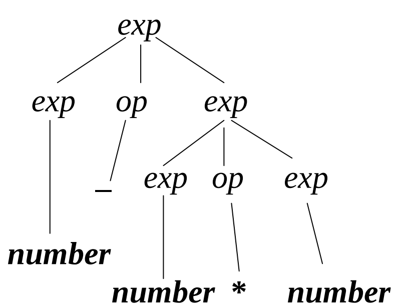
<u>1. What is Ambiguity</u>

- **Parse trees and syntax trees uniquely express the structure of syntax**
- **But it is possible for a grammar to permit a string to have more than one parse tree**
- **For example, the simple integer arithmetic grammar:**

  exp* → *exp op exp*|(*exp*) | *number
  ***op* → + | - | ***
  The string: 34-3*42

  This string has two different parse trees.



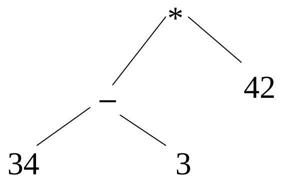


Corresponding to the two leftmost derivations

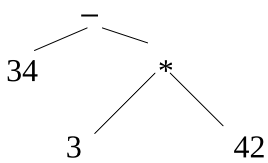
exp => *exp op exp*
=> *exp op exp op exp ,*
=> ***number*** *op exp op exp*
=>***number -*** *exp op exp*
=> ***number - number*** op exp
=> ***number - number*** * *exp*
=> ***number - number* *
*number***

exp=> exp *op exp*
=>***number*** *op exp*
=>***number -*** *exp*
=>***number -*** *exp* op *exp*
=>***number - number*** op *exp*
=>***number - number*** * *exp*
=> ***number - number* *
*number***

The associated syntax trees are



AND



<u>2. An Ambiguous Grammar</u>

- **A grammar that generates a string with *two distinct parse trees***
- **Such a grammar represents a serious problem for a parser**
  - **Not specify precisely the syntactic structure of a program**
- **In some sense, an ambiguous grammar is *like a non-deterministic automaton***
  - **Two separate paths can accept the same string**
- **Ambiguity in grammars *cannot be removed nearly as easily as non-determinism in finite automata***
  - **No algorithm for doing so, unlike the situation in the case of automata**
- ***Ambiguous grammars always fail the tests that we introduce later for the standard parsing algorithms***
  - **A body of standard techniques have been developed to deal with typical ambiguities that come up in programming languages.**

<u>3. Two Basic Methods dealing with Ambiguity</u>

- **One is to state a rule that *specifies in each ambiguous case which of the parse trees (or syntax trees) is the correct one,* called a disambiguating rule.**
  - **The advantage: it corrects the ambiguity without changing (and possibly complicating) the grammar.**
  - **The disadvantage: the syntactic structure of the language is no longer given by the grammar alone.**
- **The alternative is to Change the grammar into a form that forces the construction of the correct parse tree, thus removing the ambiguity.**
- **Of course, in either method we must first decide which of the trees in an ambiguous case is the correct one.**

4. Remove The Ambiguity in Simple Expression Grammar

- **Simply *state a disambiguating rule that establishes the relative precedence of the three operations* represented.**
  - **The standard solution is to give addition and subtraction the same precedence, and to give multiplication a higher precedence.**
- **A further disambiguating rule is the associativity of each of the operations of addition, subtraction, and multiplication.**
  - ***Specify that all three of these operations are left associative***
- **Specify that an operation is nonassociative**
  - **A sequence of more than one operator in an expression is not allowed.**
- **For instance, writing simple expression grammar in the following form: fully parenthesized expressions**

exp* → *factor op factor* | *factor
factor* → (*exp*) | *number
***op* → + |- | ***

- **Strings such as 34-3-42 and even 34-3*42 are now illegal, and must instead be written with parentheses**
  - **such as (34-3) -42 and 34- (3*42).**
- ***Not only changed the grammar, also changed the language being recognized.***

3.4.2 Precedence and Associativity

1. Group of Equal Precedence

- **The precedence 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

- **Addition and subtraction will appear "higher" (that is, closer to the root) in the parse and syntax trees**
  - **Receive lower precedence.**

2. Precedence Cascade

- **Grouping operators into different precedence levels.**
  - **Cascade is a standard method in syntactic specification using BNF.**
- **Replacing the rule**
  - ***exp* → *exp addop exp* | *term***
  - **by *exp* → *exp addop term* |*term***
  - **or *exp* → *term addop exp* |*term***
  - **A left recursive rule makes operators associate on the left**
  - **A right recursive rule makes them associate on the right**

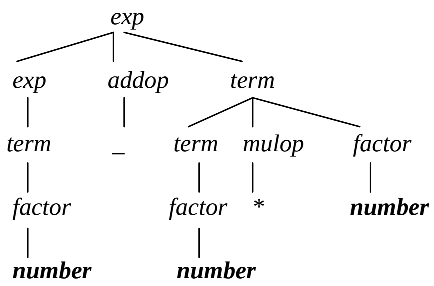
<u>3. Removal of Ambiguity</u>

- **Removal of ambiguity in the BNF rules for simple arithmetic expressions**
  - **write the rules to make all the operations left associative**

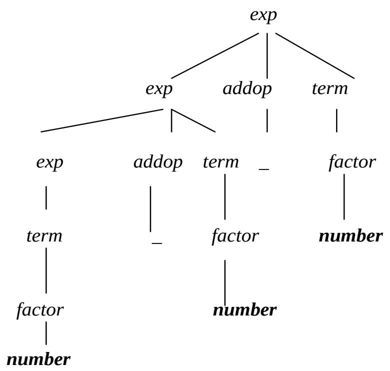
    exp* → *exp addop term* |*term
    ***addop*→ + | -**
    term* → *term mulop factor* | *factor
    ***mulop* → ***
    factor* → (*exp*) | *number

New Parse Tree

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



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



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

3.4.3 The dangling else problem

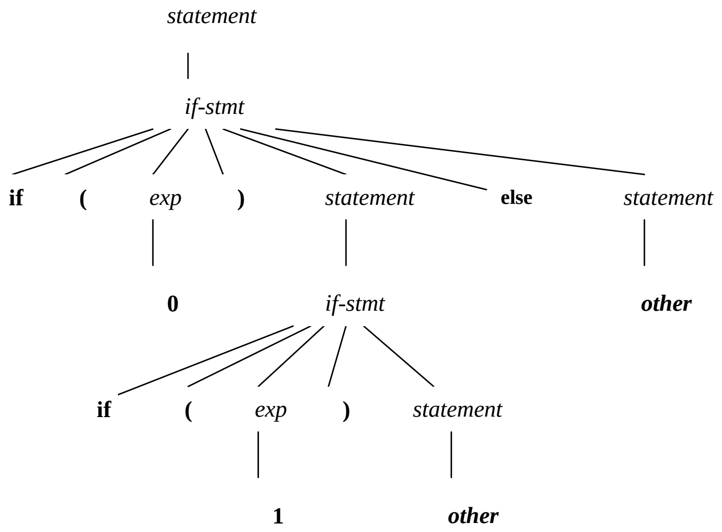
<u>1. An Ambiguity Grammar</u>

- **Consider the grammar from:**

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

- **This grammar is ambiguous as a result of the optional else. Consider the string if (0) if (1) other else other**

This string has two parse trees:



2. Dangling else problem

- **Which tree is correct depends on associating the single else-part with the first or the second if-statement.**
    - **The first associates the else-part with the first if-statement;**
    - **The second associates it with the second if-statement.**
- **This ambiguity called dangling else problem**
- **This disambiguating rule is the most closely nested rule**
    - **implies that the second parse tree above is the correct one.**

An Example

- **For example:**

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

- **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;
```

3. A Solution to the dangling else ambiguity in the BNF

statement* → *matched-stmt* | *unmatched-stmt

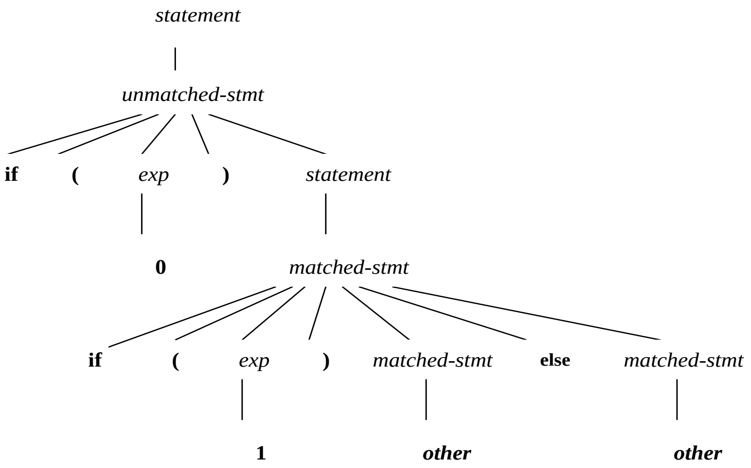
***matched-stmt* → if (*exp*) *matched-stmt* else *matched-stmt* |**
other

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

exp → 0 | 1

- **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.**

The associated parse tree for our sample string now becomes



Which indeed associates the else part with the second if-statement

3.5 Extended Notations: EBNF and Syntax Diagrams

3.5.1 EBNF Notation

1. Special Notations for *Repetitive Constructs*

- **Repetition**
  - $A \to A\ \alpha \mid \beta$ **(left recursive), and**
  - $A \to \alpha A \mid \beta$ **(right recursive)**
    - **where α and β are arbitrary strings of terminals and non-terminals, and**
  - **In the first rule β does not begin with A and**
  - **In the second β does not end with A**
- **Notation for repetition as regular expressions use, the asterisk * .**

$A \to \beta\ \alpha^*$ **, and**

$A \to \alpha^*\ \beta$

- **EBNF opts to use curly brackets {. . .} to express repetition**

$A \to \beta\{\alpha\}$ **, and**

$A \to \{\alpha\}\ \beta$

- **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.**

Examples

- **Example: The case of statement sequences**
- **The grammar as follows, in right recursive form:**

stmt-Sequence $\to$ ***stmt* ; *stmt-Sequence* | *stmt***

stmt $\to$ **s**

- **In EBNF this would appear as**

stmt-sequence $\to$ **{ *stmt* ; } *stmt* (right recursive form)**

stmt-sequence $\to$ ***stmt* { ; *stmt*} (left recursive form)**

- **A more significant problem occurs when the associativity matters**

exp $\to$ ***exp addop term* | *term***

exp $\to$ ***term* { *addop term* }**

(imply left associativity)

exp $\to$ **{*term addop* } *term***

(imply right associativity)

2. Special Notations for *Optional Constructs*

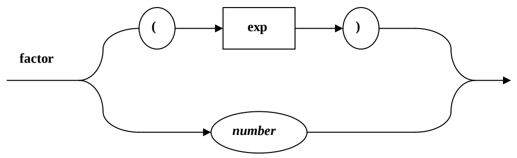
- **Optional construct are indicated by surrounding them with square brackets [...].**
- **The grammar rules for if-statements with optional else-parts would be written as follows in EBNF:**

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

- ***stmt-sequence* → *stmt*; *stmt-sequence* | *stmt* is written as**
- **stmt-sequence → stmt [; stmt-sequence]**

3.5.2 Syntax Diagrams

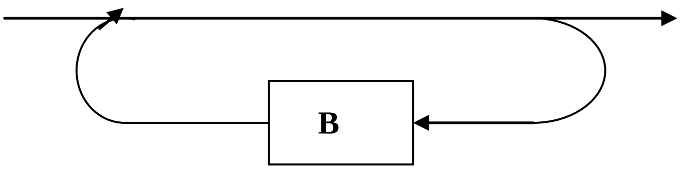
- **Syntax Diagrams:**
  - **Graphical representations for visually representing EBNF rules.**
- **An example: consider the grammar rule**
  ***factor*→(*exp*) | number**
- **The syntax diagram:**



- **Boxes representing terminals and non-terminals.**
- **Arrowed lines representing sequencing and choices.**
- **Non-terminal labels for each diagram representing the grammar rule defining that Non-terminal.**
- **A round or oval box is used to indicate terminals in a diagram.**
- **A square or rectangular box is used to indicate non-terminals.**

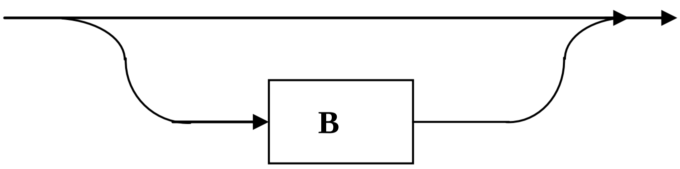
- **A repetition : A → {B}**

A



- **An optional : *A* → *[B]***

A



Examples

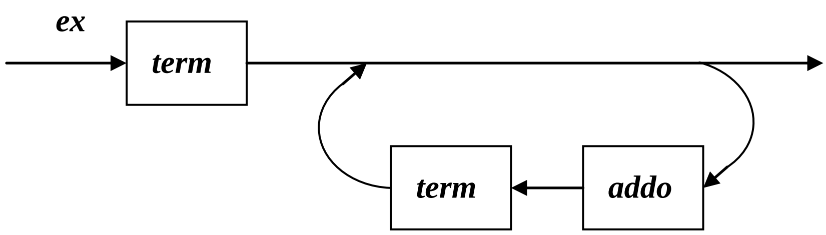
- **Example: Consider the example of simple arithmetic expressions.**

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

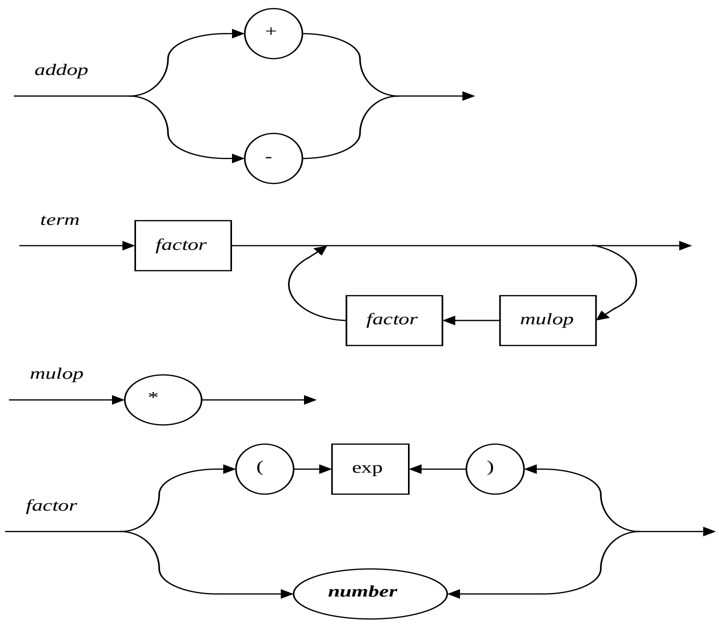
- **This BNF includes associativity and precedence**
- **The corresponding EBNF is**

exp **→ *term { addop term }***
addop **→ + | -**
term **→ *factor { mulop factor }***
mulop **→ ***
factor **→ (*exp)* | *numberr* ,**

- **The corresponding syntax diagrams are given as follows:**



Examples



- **Example: Consider the grammar of simplified if-statements, the BNF**

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

- **and the EBNF**

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

The corresponding syntax diagrams are given in following figure.