Chapter 3
Context-Free Grammars and Parsing

2022 Spring&Summer

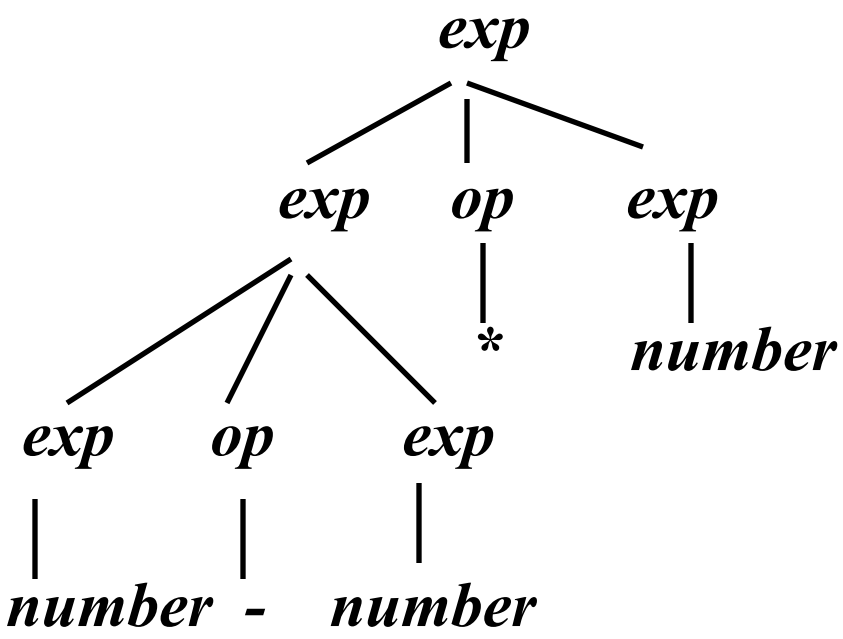
Outline

- Context-Free Grammars (CFGs)
- Derivations
- Parse trees and abstract syntax
- Ambiguous grammars

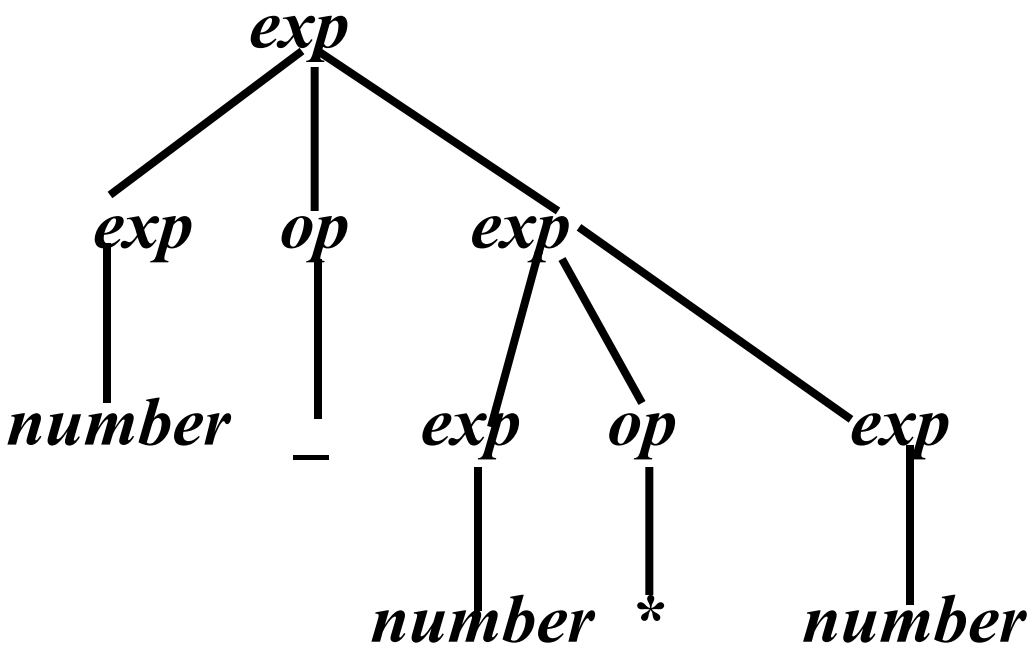
3.4 Ambiguity

exp $\rightarrow$ ***exp op exp*** | **(** ***exp*** **)** | ***number***

op $\rightarrow$ **+** | **-** | *****



3.4.1 Ambiguity grammars



3.4.1 Ambiguity grammars

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***

and

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

3.4.1 Ambiguity grammars

- **An ambiguous grammar**: a grammar that generates a string with two distinct parse trees is called an ambiguous grammar.
- Such a grammar represents a serious problem for a parser.
- It does not specify precisely the syntactic structure of a program.

3.4.1 Ambiguity grammars

Two basic methods :

1. A rule: that specifies in each ambiguous case which of the parse trees (or syntax trees) is the correct one.

   Such a rule is 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.

2. Change the grammar into a form that forces the construction of the correct parse tree.

In either method we must first decide which of the trees in an ambiguous case is the correct one.

3.4.1 Ambiguity grammars

To remove the given ambiguity in our simple expression grammar:

- State *a disambiguating rule* that establishes the relative precedences of the three operations represented.
- The *associativity* of each of the operations of addition, subtraction, and multiplication.

3.4.1 Ambiguity grammars

Specify that an operation is *nonassociative*, in that a sequence of more than one operator in an expression is not allowed.

$exp \rightarrow factor\ op\ factor \mid factor$

$factor \rightarrow (\ exp\ ) \mid number$

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

fully parenthesized expressions

3.4.2 Precedence and associativity

- 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:

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

$$addop \rightarrow + \mid -$$

$$term \rightarrow term\ mulop\ term \mid factor$$

$$mulop \rightarrow *$$

$$factor \rightarrow (\ exp\ ) \mid \textbf{number}$$

3.4.2 Precedence and associativity

- addition and subtraction will appear "higher" (that is, closer to the root) in the parse and syntax trees, and thus receive lower precedence.
- A precedence cascade: a grouping of operators into different precedence levels is a standard method in syntactic specification using BNF.

3.4.2 Precedence and associativity

- A *left recursive* rule makes its operators associate on the left
- A *right recursive* rule makes them associate on the right.

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

  replacing the rule by

  $exp \rightarrow exp\ addop\ term \mid term$ *(left associative)*

  $exp \rightarrow term\ addop\ exp \mid term$ *(right associative)*

3.4.2 Precedence and associativity

- Complete the removal of ambiguity in the BNF rules for our simple arithmetic expressions:

  exp → *exp addop term* | *term*

  addop → **+** | **-**

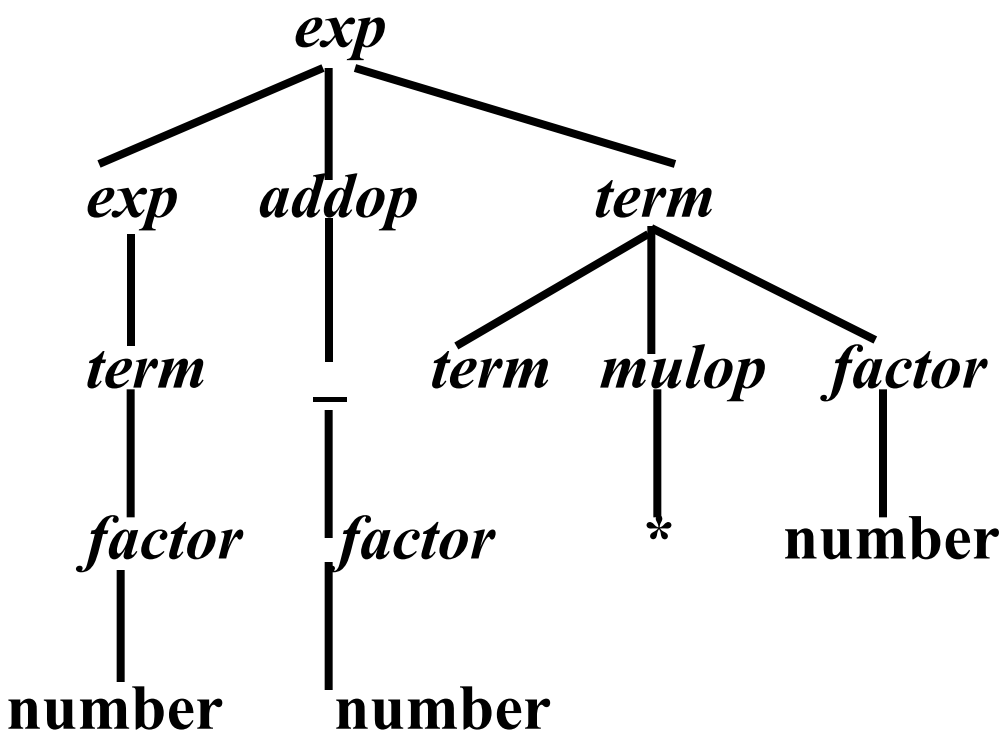
  term → *term mulop factor* | *factor*

  mulop → *

  factor → ***(*** *exp* ***)*** | ***number***

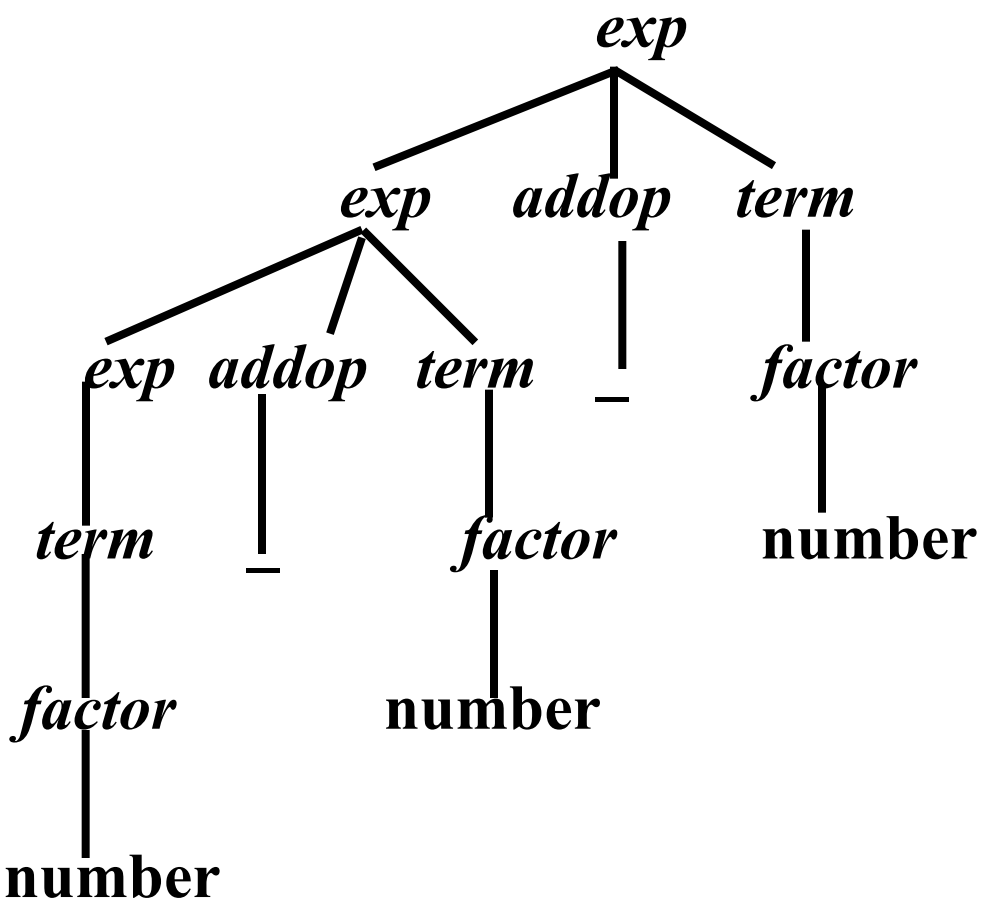
3.4.2 Precedence and associativity

The parse tree for the expression 34-3*42



3.4.2 Precedence and associativity

The parse tree for the expression 34-3-42



3.4.2 Precedence and associativity

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

The grammar from Example 3.4:

statement → *if-stmt* | **other**

if-stmt → **if (** *exp* **)** *statement*

| **if (** *exp* **)** *statement* **else** *statement*

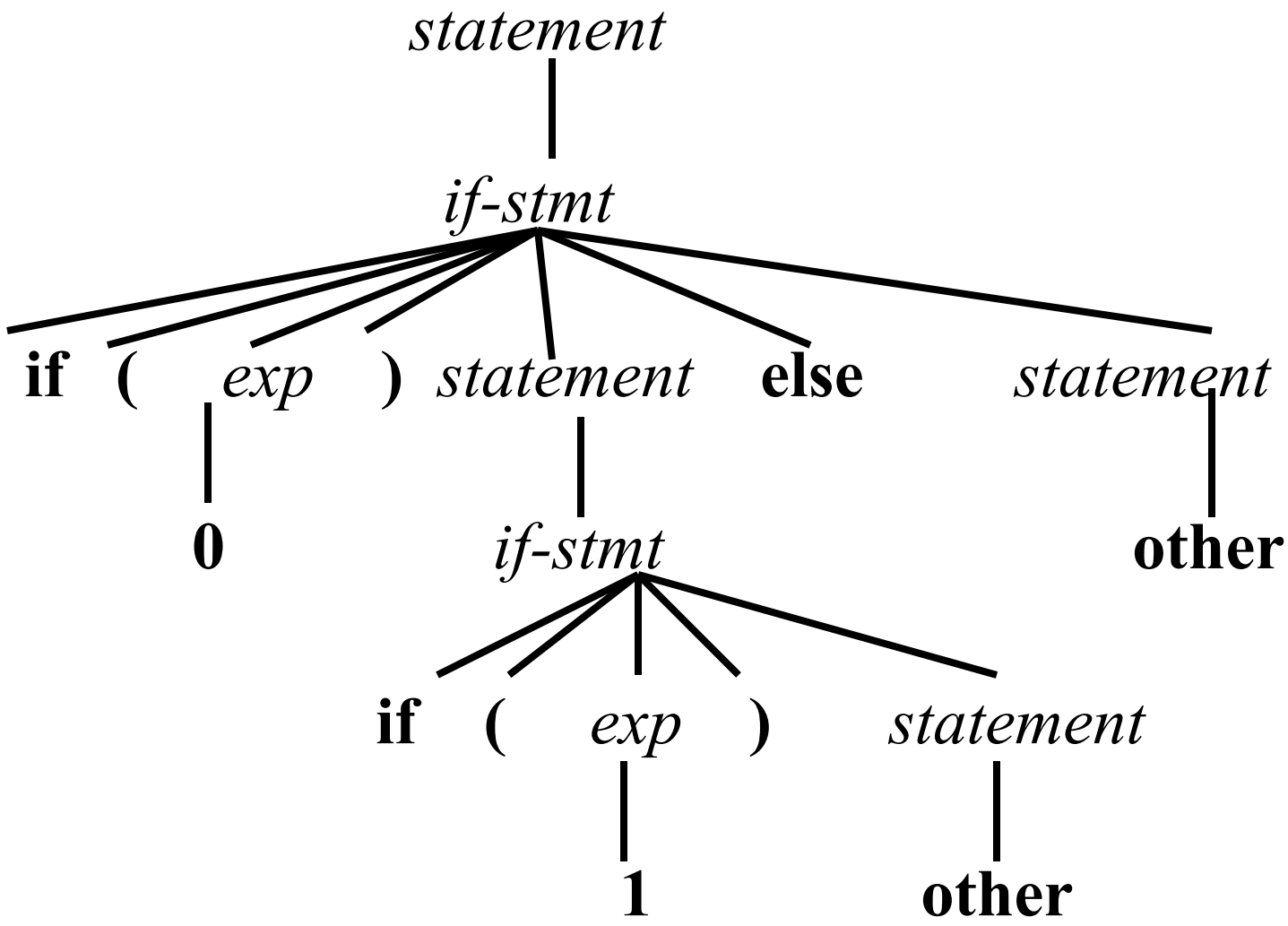
exp → **0 | 1**

This grammar is ambiguous as a result of the optional else.

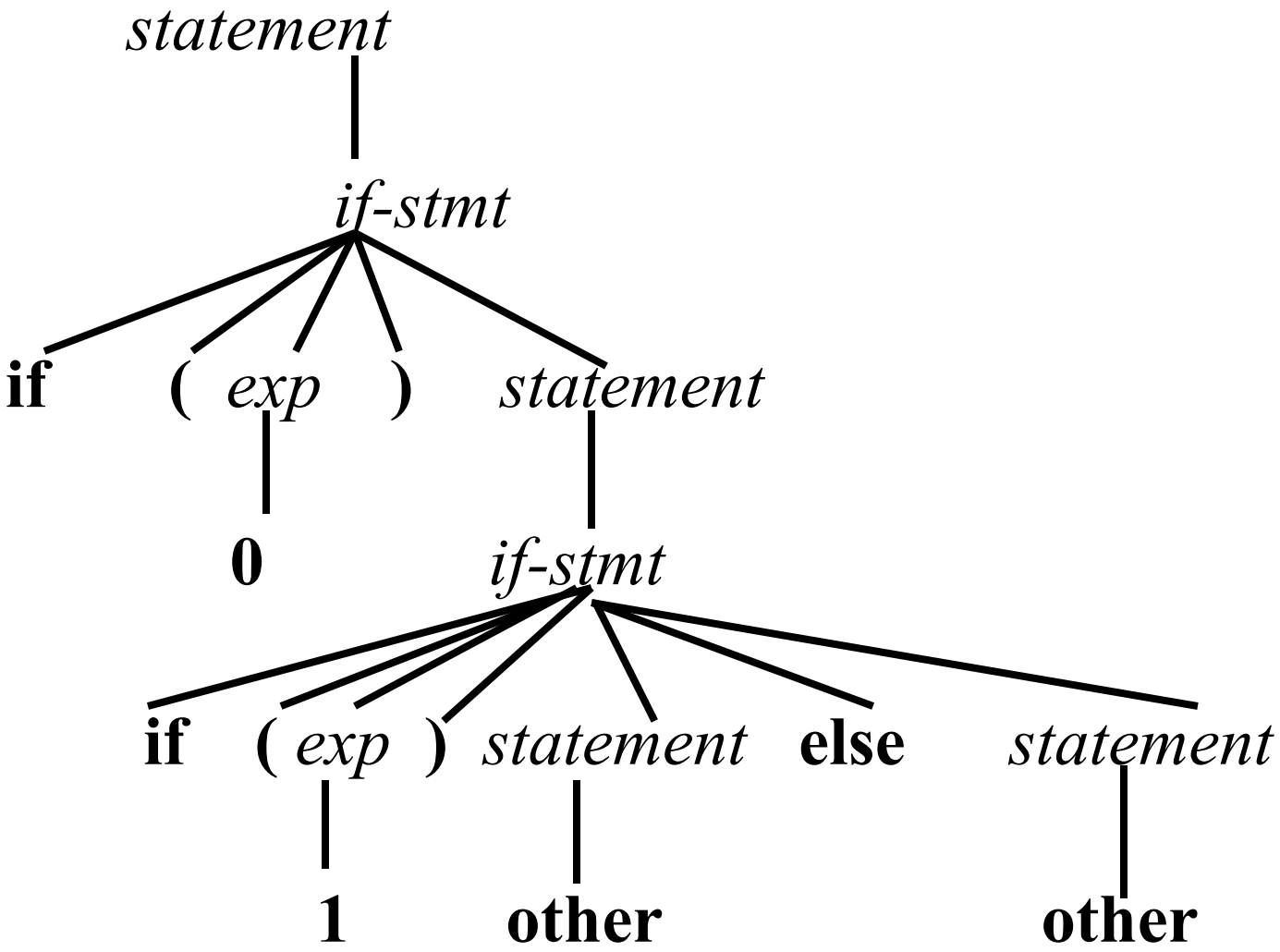
if (0) if (1) other else other

This string has the two parse trees.

3.4.3 The dangling else problem



3.4.3 The dangling else problem



3.4.3 The dangling else problem

- 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.*

An else-part should always be associated with the nearest if-statement that does not yet have an associated else-part.

3.4.3 The dangling else problem

- This disambiguating rule is called the *most closely nested rule* for the dangling else problem.
- Can 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;
```

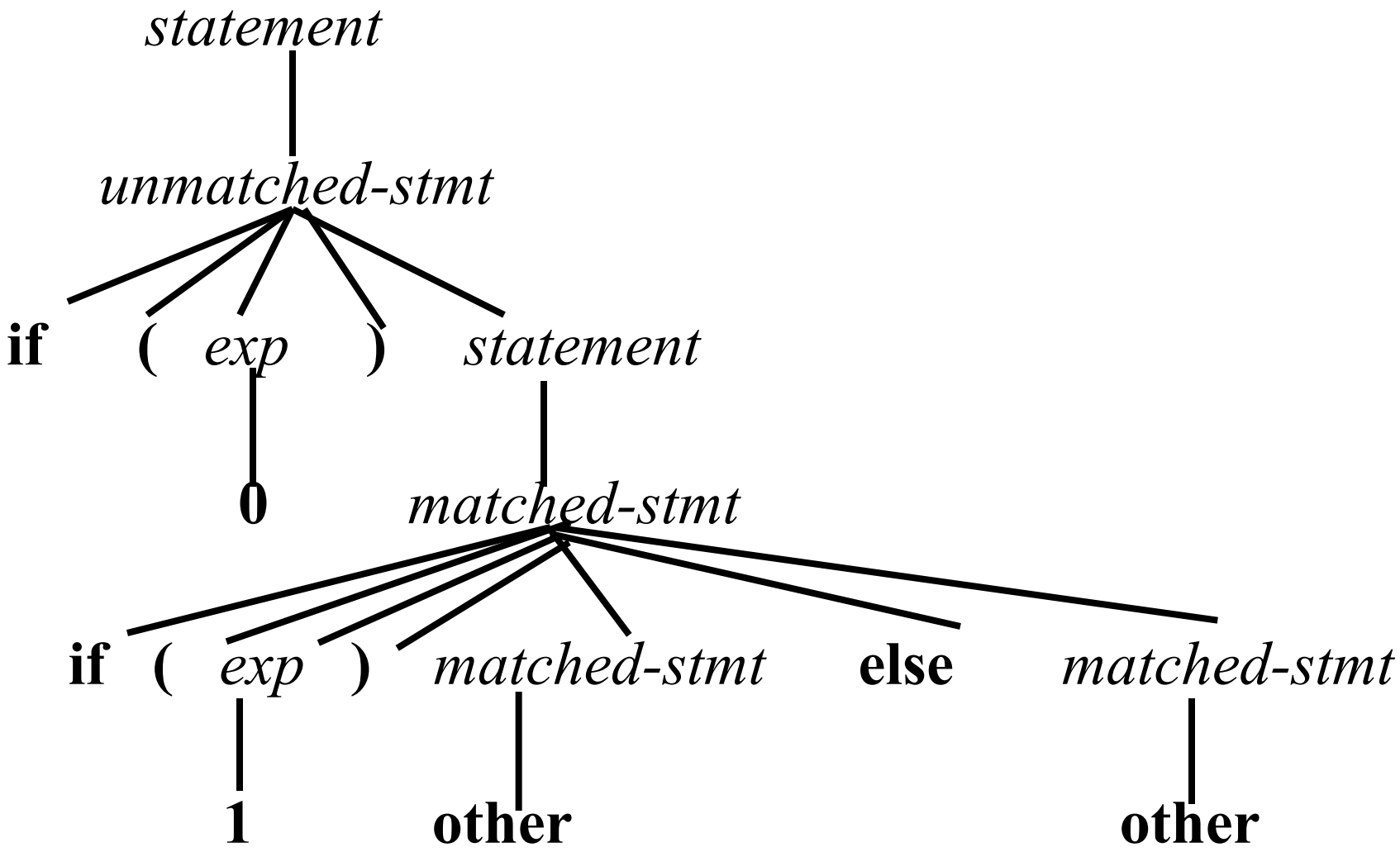
3.4.3 The dangling else problem

- 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**

- 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.

3.4.3 The dangling else problem



3.4.3 The dangling else problem

- The dangling else problem has its origins in the syntax of Algol60.
- To design the syntax that the dangling else problem does not appear.

1、Require the presence of the else-part, and this method has been used in LISP and other functional languages (where a value must also be returned).

2、Use a *bracketing keyword* for the if-statement languages that use this solution include Algol68 and Ada.

3.4.3 The dangling else problem

- In Ada, for example, the programmer writes

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

- To associate the else-part with the second if-statement. So the programmer writes

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

3.4.3 The dangling else problem

- BNF in Ada (somewhat simplified)

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

3.4.4 Inessential ambiguity

- Sometimes a grammar may be ambiguous and yet always produce unique *abstract syntax trees*.

  the grammar ambiguously as

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

  stmt → **s**

- Either a *right recursive* or *left recursive* grammar rule would still result in the same syntax tree structure.

3.4.4 Inessential ambiguity

- **Inessential ambiguity:** *the associated semantics do not depend on what disambiguating rule is used.*
- Arithmetic *addition* or string *concatenation*.

  that represent **associative operations**

  (a binary operator • is associative if (a • *b)* • *c* = *a* • *(b* • *c)* for all values *a, b,* and c).
- The syntax trees are still distinct, semantic value are the same.

3.5 extended notations: EBNF and syntax diagrams

- BNF notation is sometimes extended to include special notations for *repetitive* and *optional* constructs.
- Extended BNF, or EBNF.

Repetition

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

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

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

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

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

3.5 extended notations: EBNF and syntax diagrams

- **Example**:

  stmt-Sequence → *stmt* ; *stmt-Sequence* | *stmt*

  stmt → s

- **In EBNF this would appear as**

  stmt-sequence → **{** ***stmt*** **;** **}** ***stmt***

  (right recursive form)

  stmt-sequence → ***stmt*** ***{*** **;** ***stmt}***

  (left recursive form)

3.5 extended notations: EBNF and syntax diagrams

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

  statement → *if-stmt* | other

  if-stmt → if (*exp*) *statement* [else *statement*]

  exp → 0 | 1

3.6 Formal properties of context-free language

- Definition : A context-free grammar consists of the following:
  1. A set T of **terminals.**
  2. A set *N* of **nonterminals** (disjoint from *T).*
  3. A set *P* of **productions**, or **grammar rules**, of the form $A \rightarrow a$, where *A is an* element of *N* and *a* is an element of $(T \cup N)^*$ (a possibly empty sequence of terminals and nonterminals).
  4. A **start symbol** *S* from the set *N.*

3.6.1 A formal definition of context-free language

- Let G be a grammar as defined above, $G = (T, N, P, S)$.
- A derivation step over G is of the form $a\ A\ \gamma => a\ \beta\ \gamma$,
  where a and γ are elements of $(T \cup N)^*$ and $A \rightarrow \beta$ is in P.
- The set of symbols: The union $T \cup N$ of the sets of *terminals* and *nonterminals*
- A sentential form: a string a in $(T \cup N)^*$.

3.6.1 A formal definition of context-free language

- $\alpha \Rightarrow^* \beta$ if and only if there is a sequence of 0 or more derivation steps (n >= 0)

  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n$ such that $\alpha = \alpha_1$, and $\beta = \alpha_n$ (If $n = 0$, then $\alpha = \beta$)

- A derivation over the grammar G is of the form $S \Rightarrow^* w$, where $w \in T^*$

  w is a string of terminals only, called a sentence.

  S is the start symbol of G.

3.6.1 A formal definition of context-free language

- The language generated by G, written $L(G)$, is defined :

  $L(G) = \{w \in T^* \mid$ there exists a derivation $S \Rightarrow^* w$ of G$\}$.

- A *leftmost derivation* $S \Rightarrow^*_{lm} w$ is a derivation

each derivation step a $\alpha A \gamma \Rightarrow \alpha \beta \gamma$, is such that $\alpha \in T^*$; that is, a consists only of terminals.

- A *rightmost derivation* is that each derivation step

  $\alpha A \gamma \Rightarrow \alpha \beta \gamma$ has the property that $\gamma \in T^*$

3.6.1 A formal definition of context-free language

- A parse tree over the grammar G is a rooted labeled tree with the following properties:
  1. Each node is labeled with a terminal or a nonterminal or ε.
  2. The root node is labeled with the start symbol *S*.
  3. Each leaf node is labeled with a terminal or with ε.
  4. Each nonleaf node is labeled with a nonterminal.
  5. If a node with label $A \in N$ has n children with labels $X_1, X_2, \ldots, X_n$ (which may be terminals or nonterminals), then $A \rightarrow X_1 X_2 \ldots X_n \in P$ (a production of the grammar).

3.6.1 A formal definition of context-free language

- Each derivation gives rise to a parse tree.
- In general, many derivations may give rise to the same parse tree.
- Each parse tree, however, has a unique leftmost and rightmost derivation that give rise to it.
- The leftmost derivation corresponds to a *preorder traversal* of the parse tree,.
- The rightmost derivation corresponds to the reverse of a *postorder traversal* of the parse tree.

3.6.1 A formal definition of context-free language

- A grammar G is *ambiguous* if there exists a string $w \in L(G)$ such that w has two distinct parse trees (or *leftmost* or *rightmost* derivations).
- A set of strings L is said to be a context-free language if there is context-free grammar G such that $L = L$ (G).

3.7 Syntax of the TINY language

Figure 3.6: Grammar of the TINY language in BNF

program → *stmt-sequence*
stmt-sequence → *stmt-sequence* **;** *statement* | *statement*
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **if** *exp* **then** *stmt-sequence* **end**
    | **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**
repeat-stmt → **repeat** *stmt-sequence* **until** *exp*
assign-stmt → **identifier :=** *exp*
read-stmt → **read identifier**
write-stmt → **write** *exp*
exp → *simple-exp comparison-op simple-exp* | *simple-exp*
comparison-op → **<** | **=**
simple-exp → *simple-exp addop term* | *term*
addop → **+**|**-**
term → *term mulop factor factor* | *factor*
mulop → ***|/**
factor → **(***exp***)** |**number** |**identifier**

Homework of Chapter 3

- 3.2 Given the grammar A -> AA | (A) | ε ,
- 1). Describe the language it generates,
- 2). Show that it is ambiguous.
- 3.3 Given the grammar
- exp -> exp addop term | term
- addop -> + | -
- term -> term mulop factor | factor
- mulop -> *
- factor -> (exp) | number
- write down leftmost derivations, parse trees, and abstract syntax trees for the following expressions:
- a. 3+4*5-6 b. 3*(4-5+6) c. 3-(4+5*6)

Homework of Chapter 3

- 3.4 The following grammar generates all regular expressions over the alphabet of letters (we have used quotes to surround operators,since the vertical bar is an operator as well as a metasymbol):
- rexp -> rexp “ | ” rexp
- | rexp rexp
- | rexp “*”
- | “(“ rexp ”)”
- | letter
- a.Give a derivation for the regular expression *(ab|b)** using this grammar.
- b.Show that this grammar is ambiguous.
- c.Rewrite this grammar to establish the correct precedences for the operators.
- d.What associativity does your answer in part (c) give to the binary operators? Why?