COMPILER CONSTRUCTION

Zhihua Jiang

3. Context-Free Grammars and Parsing

Content

3. Context-Free Grammars and Parsing

PART ONE

Introduction

- Parsing is the task of **Syntax Analysis**
    - Determine the syntax, or structure, of a program.
- The syntax is defined by the **grammar rules of a Context-Free Grammar (CFG)**
    - The rules of a context-free grammar are **recursive**
- The basic data structure of Syntax Analysis is **parse tree** or **syntax tree**
    - The syntactic structure of a language must also be recursive

3.1 The Parsing Process

Function of a Parser

- Takes the sequence of tokens produced by the scanner as its input and produces the syntax tree as its output.

$$\textit{Sequence of tokens} \xrightarrow{\text{Parser}} \textit{Syntax-Tree}$$

Issues of the Parsing

- The sequence of tokens is *not an explicit input parameter*
  - The parser calls a scanner procedure **getToken** to fetch the next token from the input during the parsing process
  - The parsing step of the compiler reduces to a call to the parser as follows: **SyntaxTree = parse( )**

Issues of the Parsing

- What is more difficult for the parser than the scanner is the treatment of errors.
- Error in the scanner
  - Generate an error token and consume the offending character
- Error in the parser
  - The parser must not only report an error message
  - **but it must recover from the error and continue parsing** (to find as many errors as possible)

3.2 Context-Free Grammars

Basic Concept

- A context-free grammar is *a specification for the syntactic structure of a programming language*
  - Similar to the specification of the lexical structure of a language using regular expressions
  - Except involving recursive rules
- For example:

$$exp \rightarrow exp\ op\ exp \mid (exp) \mid \textbf{number}$$
$$op \rightarrow + \mid - \mid *$$

3.2.1 Comparison to Regular Expression Notation

Comparing an Example

- The context-free grammar:

  $exp \rightarrow exp\ op\ exp \mid (exp) \mid \textbf{\textit{number}}$

  $op \rightarrow + \mid - \mid \boxed{*}$

- The regular expression:

  $number = digit\ digit\boxed{*}$

  $digit = 0|1|2|3|4|5|6|7|8|9$

Grammars Rules

- Vertical bar appears as meta-symbol for choice.
- Concatenation is used as a standard operation.
- No meta-symbol for repetition

  (like the * of regular expressions)
- Use the arrow symbol → instead of equality to express the definitions of names
- Names are written in italic (in a different font)
- Ex. *exp* → *exp op exp* | (*exp*) | **number**

Grammar Rules

- Grammar rules use regular expressions as components
- The notation was developed by John Backus and adapted by Peter Naur for the Algol 60 report
- Grammar rules in this form are usually said to be in Backus-Naur form, or **BNF**

3.2.2 Specification of Context-Free Grammar Rules

Construction of a CFG rule

- Given an alphabet, a context-free grammar rule in BNF consists of a string of symbols.
  - The first symbol is a name for a structure.
  - The second symbol is the meta-symbol "$\rightarrow$".
  - This symbol is followed by a string of symbols, each of which is either a symbol from the alphabet, a name for a structure, or the meta symbol "|".

Construction of a CFG rule

- A grammar rule is interpreted as follows
  - The rule defines the structure whose name is to the left of the arrow
  - The structure is defined to consist of one of the choices on the right-hand side separated by the vertical bars
  - The sequence of symbols and structure names within each choice defines the layout of the structure

3.2.3 Derivations & Language Defined by a Grammar

How Grammar Determine a Language

- Context-free grammar rules determine the set of syntactically legal strings of token symbols for the structures defined by the rules.
  - For example, the arithmetic expression
    - (34-3)*42
  - Corresponds to the legal string of seven tokens
    - **(number - number) * number**
  - While (34-3*42 is not a legal expression
- There is a left parenthesis that is not matched by a right parenthesis and the second choice in the grammar rule for an *exp* requires that parentheses be generated in pairs
  - Example: $exp \rightarrow exp\ op\ exp \mid (exp) \mid \textbf{number}$

Derivations

- Grammar rules determine the legal strings of token symbols by means of derivations
- A **derivation** is a sequence of replacements of structure names by choices on the right-hand side of grammar rules
- A derivation begins with a single structure name and ends with a string of token symbols
- At each step in a derivation, a single replacement is made using one choice from a grammar rule

Derivations

- The example

  $exp \rightarrow exp\ op\ exp \mid (exp) \mid \textbf{\textit{number}}$

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

- A derivation

(1) $exp \Rightarrow exp\ op\ exp$	$[exp \rightarrow exp\ op\ exp]$
(2) $\Rightarrow exp\ op\ \textbf{\textit{number}}$	$[exp \rightarrow \textbf{\textit{number}}]$
(3) $\Rightarrow exp\ *\ \textbf{\textit{number}}$	$[op \rightarrow *\ ]$
(4) $\Rightarrow (exp)\ *\ \textbf{\textit{number}}$	$[exp \rightarrow (\ exp\ )\ ]$
(5) $\Rightarrow (exp\ op\ exp\ )\ *\ \textbf{\textit{number}}$	$[exp \rightarrow exp\ op\ exp\}$
(6) $\Rightarrow (exp\ op\ \textbf{\textit{number}})\ *\ \textbf{\textit{number}}$	$[exp \rightarrow \textbf{\textit{number}}]$
(7) $\Rightarrow (exp\ -\ \textbf{\textit{number}})\ *\ \textbf{\textit{number}}$	$[op \rightarrow\ -\ ]$
(8) $\Rightarrow (\textbf{\textit{number}}\ -\ \textbf{\textit{number}})\ *\ \textbf{\textit{number}}$	$[exp \rightarrow \textbf{\textit{number}}]$

- Derivation steps use a different arrow from the arrow meta-symbol in the grammar rules.

Language Defined by a Grammar

The set of all strings of token symbols obtained by derivations from the *exp* symbol is the **language defined by the grammar** of expressions

- $L(G) = \{ s \mid exp \overset{*}{\Rightarrow} s \}$

  G represents the expression grammar

  s represents *an arbitrary string of token symbols*

  (sometimes called a sentence)

  The symbol $\overset{*}{\Rightarrow}$ stands for a derivation consisting of a sequence of replacements

  (The asterisk is used to indicate a sequence of steps)

  Grammar rules are sometimes called productions, because they "produce" the strings in $L(G)$ via derivations

Symbols in rules

- **Start symbol**
  - The most general structure is listed first in the grammar rules.
- **Non-terminals**
  - Structure names, they always must be replaced further in a derivation.
- **Terminals**
  - Symbols in the alphabet, they terminate a derivation.
  - Terminals are usually tokens in compiler applications.

Examples

Example 3.1:

- The grammar G with the single grammar rule

  $$E \rightarrow (E) \mid a$$

- This grammar generates the language

  $$L(G) = \{\ a,(a),((a)),(((a))),\ldots\} = \{\ (^n a)^n \mid n \geq 0\}$$

- Derivation for $((a))$

  $$E \Rightarrow (E) \Rightarrow ((E)) \Rightarrow ((a))$$

Examples

Example 3.2:

- The grammar G with the single grammar rule $E \to (E)$
- This grammar generates no strings at all, there is no way we can derive a string consisting only of terminals.

Examples

Example 3.3:

- Consider the grammar G with the single grammar rule

$$E \to E + a \mid a$$

- This grammar generates all strings consisting of a's separated by +'s:

$$L(G) = \{a, a + a, a + a + a, a + a + a + a, \ldots\}$$

- Derivation:

$$E \Rightarrow E+a \Rightarrow E+a+a \Rightarrow E+a+a+a \Rightarrow \ldots \ldots$$

finally replace the E on the left using the base $E \to a$

Example

Example 3.4

- Consider the following extremely simplified grammar of statements:
  - *statement* → *if-stmt* | *other*
  - *if-stmt* → if (*exp*) *statement* | if (*exp*) *statement* else *statement*
  - *exp* → 0 | 1

- Examples of strings in this language are

other

if (0) other

if (1) other

if (0) other else other

if (1) other else other

if (0) if (0) other

if (0) if (1) other else other

if (1) other else if (0) other else other

Recursion

- **left recursive:**
  - **The non-terminal *A* appears as the first symbol on the right-hand side of the rule defining *A***
  - $A \rightarrow A\,a \mid a$
- **right recursive:**
  - **The non-terminal *A* appears as the last symbol on the right-hand side of the rule defining *A***
  - $A \rightarrow a\,A \mid a$

Examples of Recursion

- Consider a rule of the form: $A \to A\alpha \mid \beta$
  - where α and β represent arbitrary strings and β does not begin with *A*.
- This rule generates all strings of the form
  - $\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots$
  - (all strings beginning with a β, followed by 0 or more α's).
- This grammar rule is equivalent in its effect to the regular expression $\beta\alpha^*$.
- Similarly, the right recursive grammar rule $A \to \alpha A \mid \beta$
  - *(where β does not end in A)*
  - generates all strings $\beta, \alpha\beta, \alpha\alpha\beta, \alpha\alpha\alpha\beta, \ldots.$

Examples of Recursion

- To generate the same language as the regular expression a* we must have a notation for a grammar rule that generates the empty string
  - use the epsilon meta-symbol for the empty string
  - $\rightarrow$ ε, called an ε-production (an "epsilon production").
- A grammar that generates a language containing the empty string must have at least one ε-production.
- A grammar equivalent to the regular expression a*
  - $A \rightarrow A\,a \mid \varepsilon$ *or* $A \rightarrow a\,A \mid \varepsilon$
- Both grammars generate the language
  - $\{a^n \mid n \geq 0\} = L(\mathrm{a}^*)$.

Examples

- **Example 3.5:**
  - $A \rightarrow (A)\, A \mid \varepsilon$
  - **generates the strings of all "balanced parentheses".**
- **For example, the string (( ) (( ))) ( )**
  - **generated by the following derivation**
  - **(the ε-production is used to make *A* disappear as needed):**
  - **A => (A) A => (A)(A)A => (A)(A) =>(A)( ) => ((A)A)( ) =>(( )A)( ) => (( ) (A)A) ( ) => (( )(A))( ) => (( )((A)A))( ) => (( )(( )A))( ) => (( )(( )))( )**

Examples

- **Example 3.6:**
  - **The statement grammar of Example 3.4 can be written in the following alternative way using an ε-production:**

    statement **→ *if-stmt* | *other***

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

    else-part **→ else *statement* | ε**

    exp → 0 | 1

- **The ε-production indicates that the structure *else-part* is optional.**

3.3 Parse trees and abstract syntax trees

3.3.1 Parse trees

Derivation vs. Structure

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

Derivation vs. Structure

Right-most derivation

(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*]**

Derivation vs. Structure

Left-most derivation

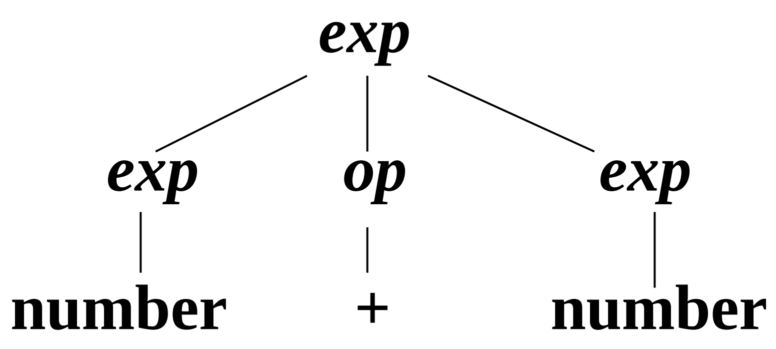
(1)	***exp* => *exp op exp***	**[*exp* → *exp op exp*]**
(2)	**=> (*exp*) *op exp***	**[*exp*→ (*exp*)]**
(3)	**=> (*exp op exp*) *op exp***	**[*exp* → *exp op exp*]**
(4)	**=> (*number op exp*) *op exp***	**[*exp*→ *number*]**
(5)	**=>(*number* - *exp*) *op exp***	**[*op* → -]**
(6)	**=> (*number* -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

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;
  - The children of each internal node represent the replacement of the associated non-terminal in one step of the derivation.

Parsing Tree

- **The example:**
  - ***exp* => *exp op exp* => number *op exp* => number + *exp* => number + number**
- **Corresponding to the parse tree:**



- **The above parse tree corresponds to the three derivations:**

Parsing Tree

- **Left most derivation**

  （1）*exp* => *exp op exp*

  （2） => *number op exp*

  （3） => *number* + *exp*

  （4） => *number* + *number*

- **Right most derivation**

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

  (2) => *exp op number*

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

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

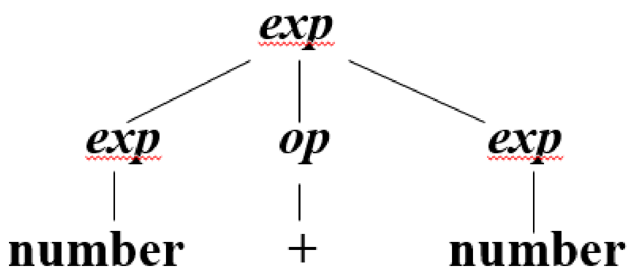
- **Neither leftmost nor rightmost derivation**

  （1） *exp* => *exp op exp*

  （2） => *exp* + *exp*

  （3） => *number* + *exp*

  （4） => *number* + *number*

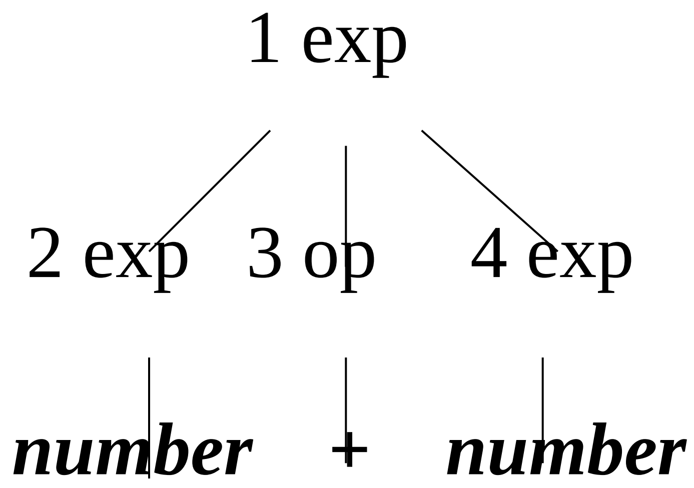


Parsing Tree

- **A leftmost derivation:**
  - **A derivation in which the leftmost non-terminal is replaced at each step in the derivation.**
  - **Corresponds to the *from-left-to-right* numbering of the internal nodes of its associated parse tree.**
- **A rightmost derivation:**
  - **A derivation in which the rightmost non-terminal is replaced at each step in the derivation.**
  - **Corresponds to the *from-right-to-left* numbering of the internal nodes of its associated parse tree.**

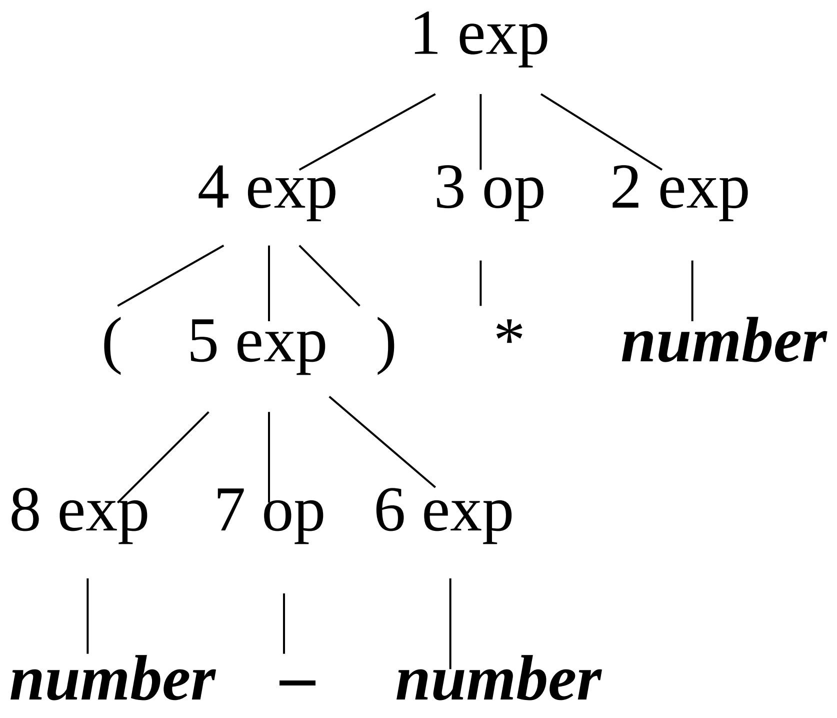
Parsing Tree

- The parse tree corresponds to the leftmost derivation.



Example: The expression (34-3)*42

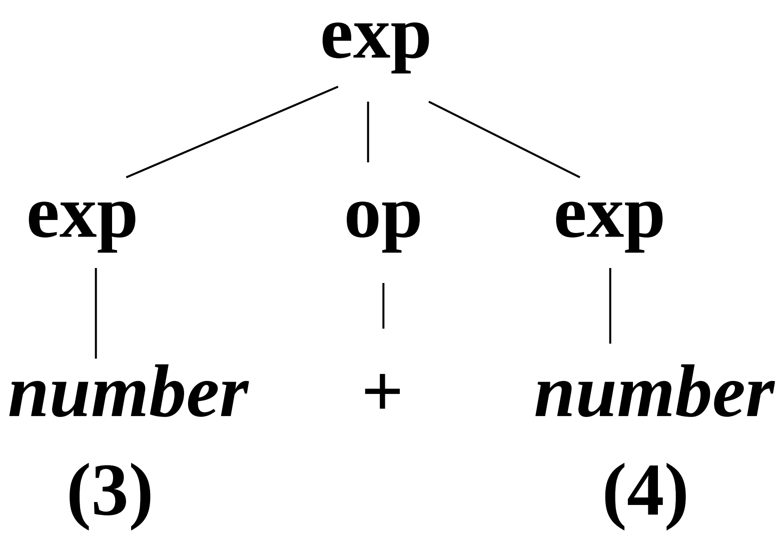
- The parse tree corresponds to the rightmost derivation



3.3.2 Abstract syntax trees

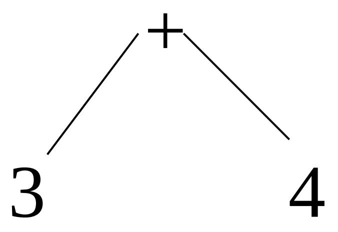
Why Abstract Syntax Tree

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



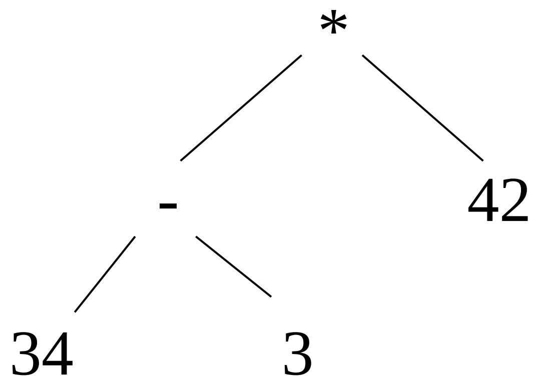
Why Abstract Syntax-Tree

- The principle of syntax-directed translation
  - The meaning, or semantics, of the string 3+4 should be directly related to its syntactic structure.
- 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.

Abstract 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**

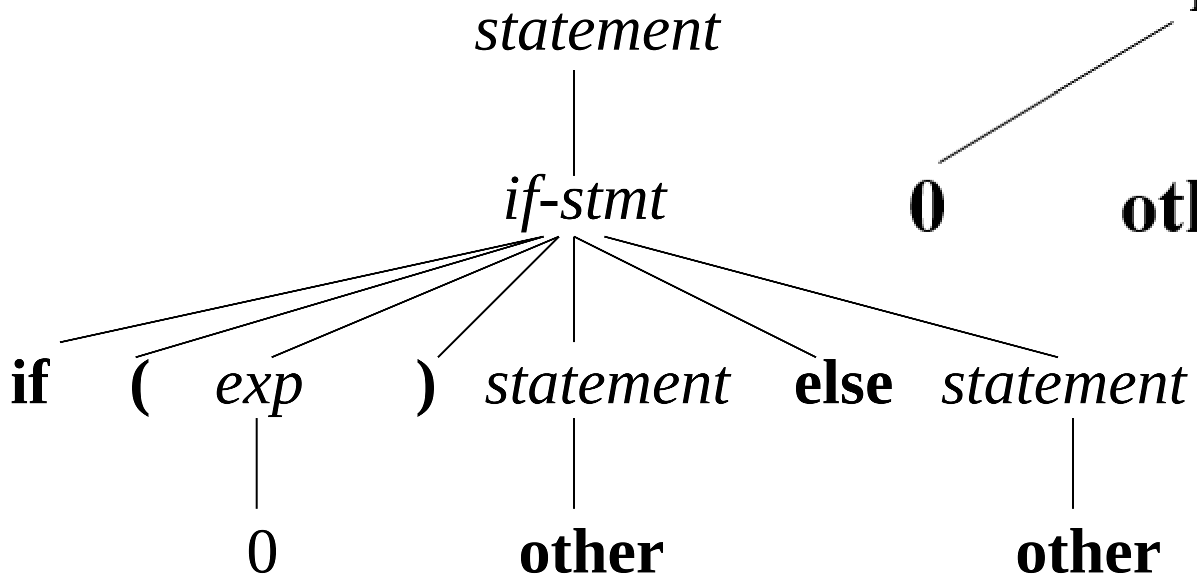
if-stmt → **if** (*exp*) *statement*

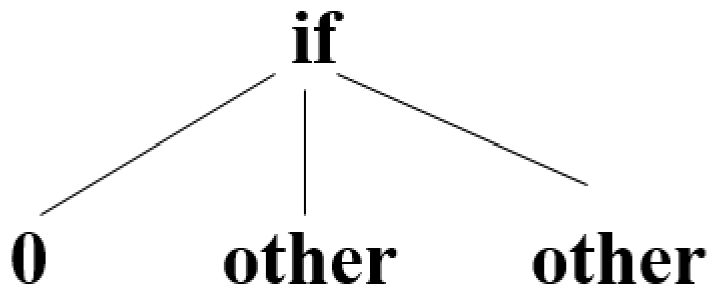
| **if (** *exp* **)** *statement* **else** *statement*

exp → **0 | 1**

Examples

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





3.4 Ambiguity

What is Ambiguity

- 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:

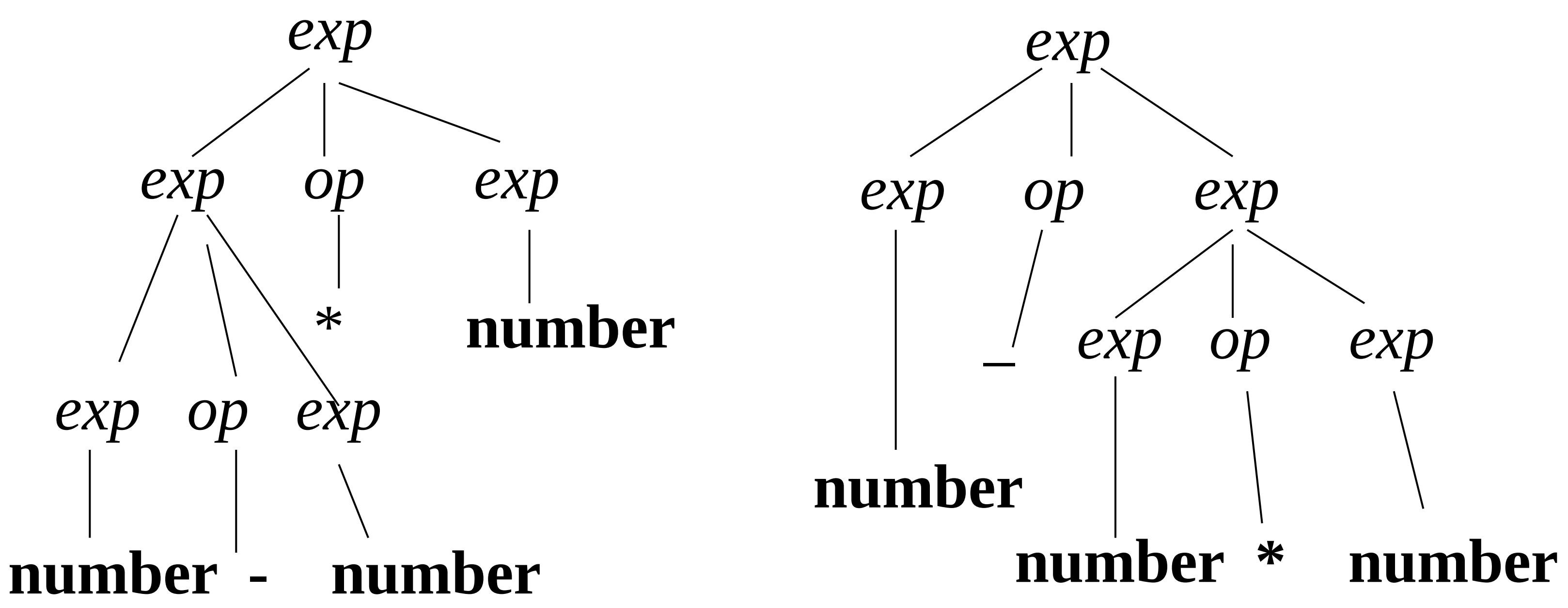
$$exp \rightarrow exp\ op\ exp \mid \mathbf{(}\ exp\ \mathbf{)} \mid \mathbf{number}$$

$$op \rightarrow \mathbf{+} \mid \mathbf{-} \mid \mathbf{*}$$

The string: 34-3*42

What is Ambiguity

This string 34-3*42 has two different parse trees.



An Ambiguous Grammar

- 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

Two Basic Methods dealing with Ambiguity

- 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.

Two Basic Methods dealing with Ambiguity

- The other 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.

3.4.2 Precedence and Associativity

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 or syntax trees
  - Receive lower precedence.

Removal of Ambiguity

- 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→ **+ | -**

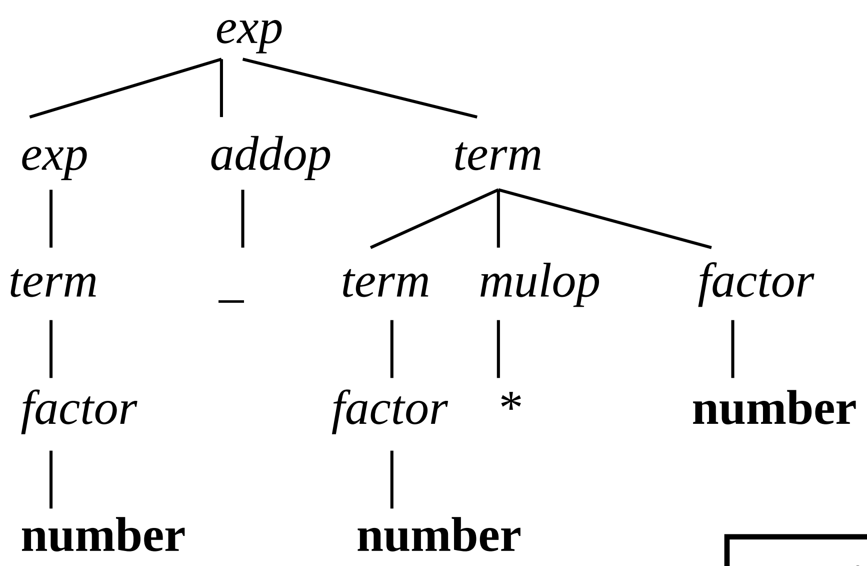
    term → *term mulop factor* | *factor*

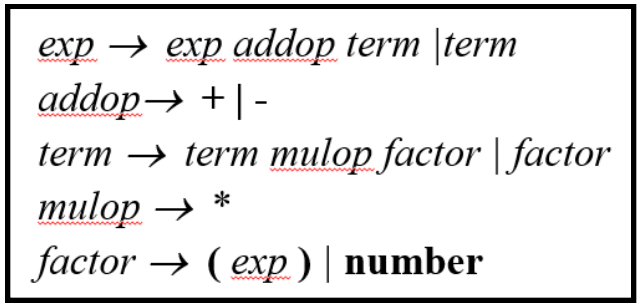
    mulop → *****

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

New Parse Tree

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

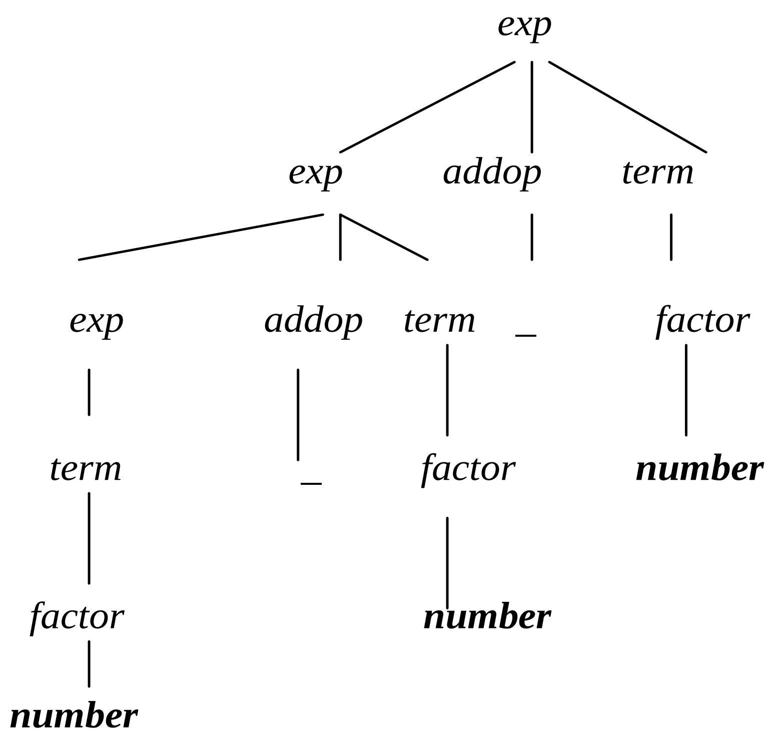


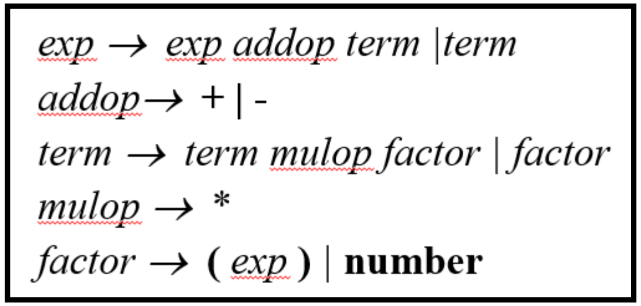


exp → *exp addop term* | *term*
addop → + | -
term → *term mulop factor* | *factor*
mulop → *
factor → **(** *exp* **)** | **number**

New Parse Tree

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





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

3.4.3 The dangling else problem

An Ambiguous Grammar

- Consider the grammar:

  statement → *if-stmt* | ***other***

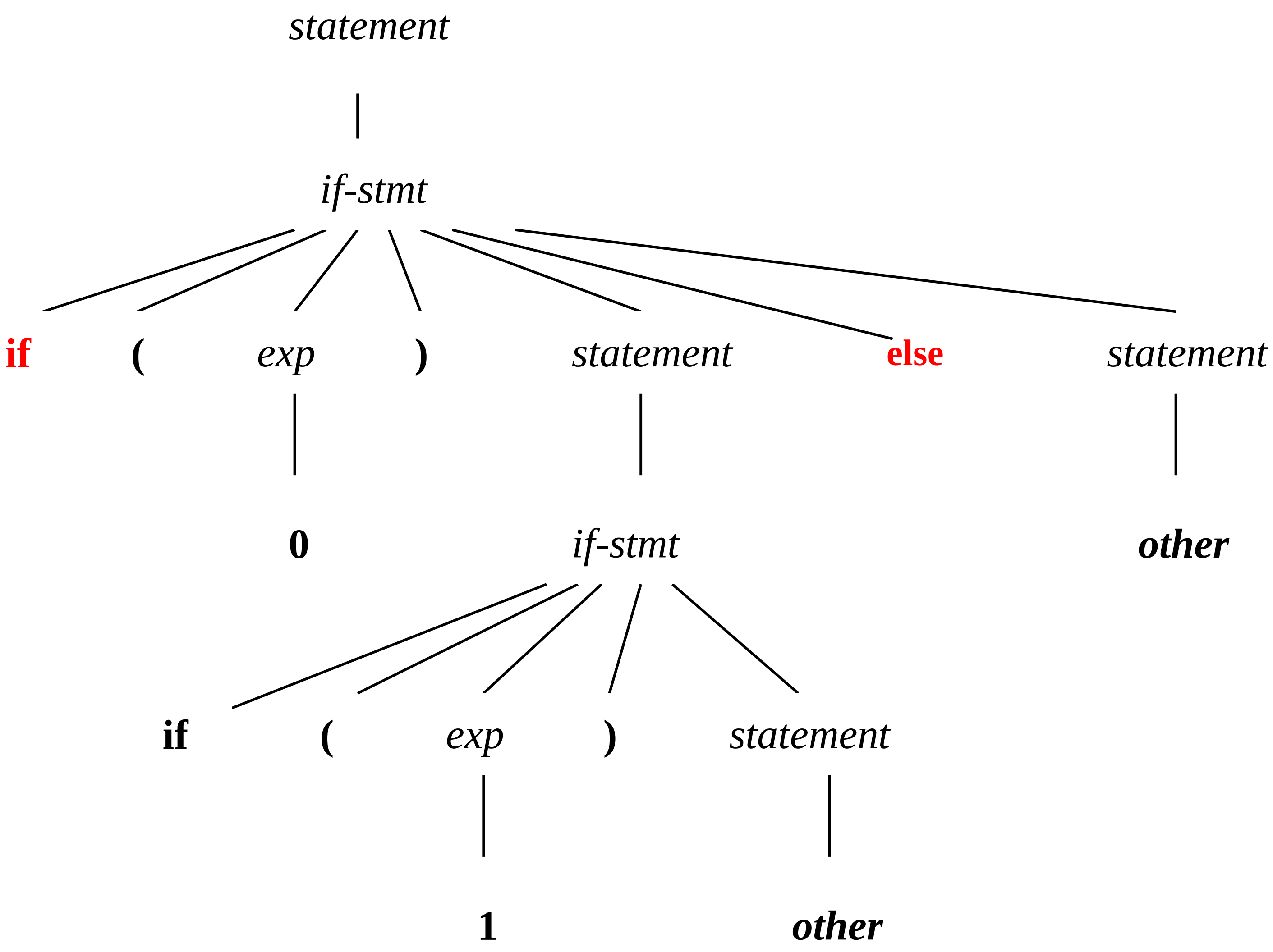
  if-stmt → **if (** *exp* **)** *statement*

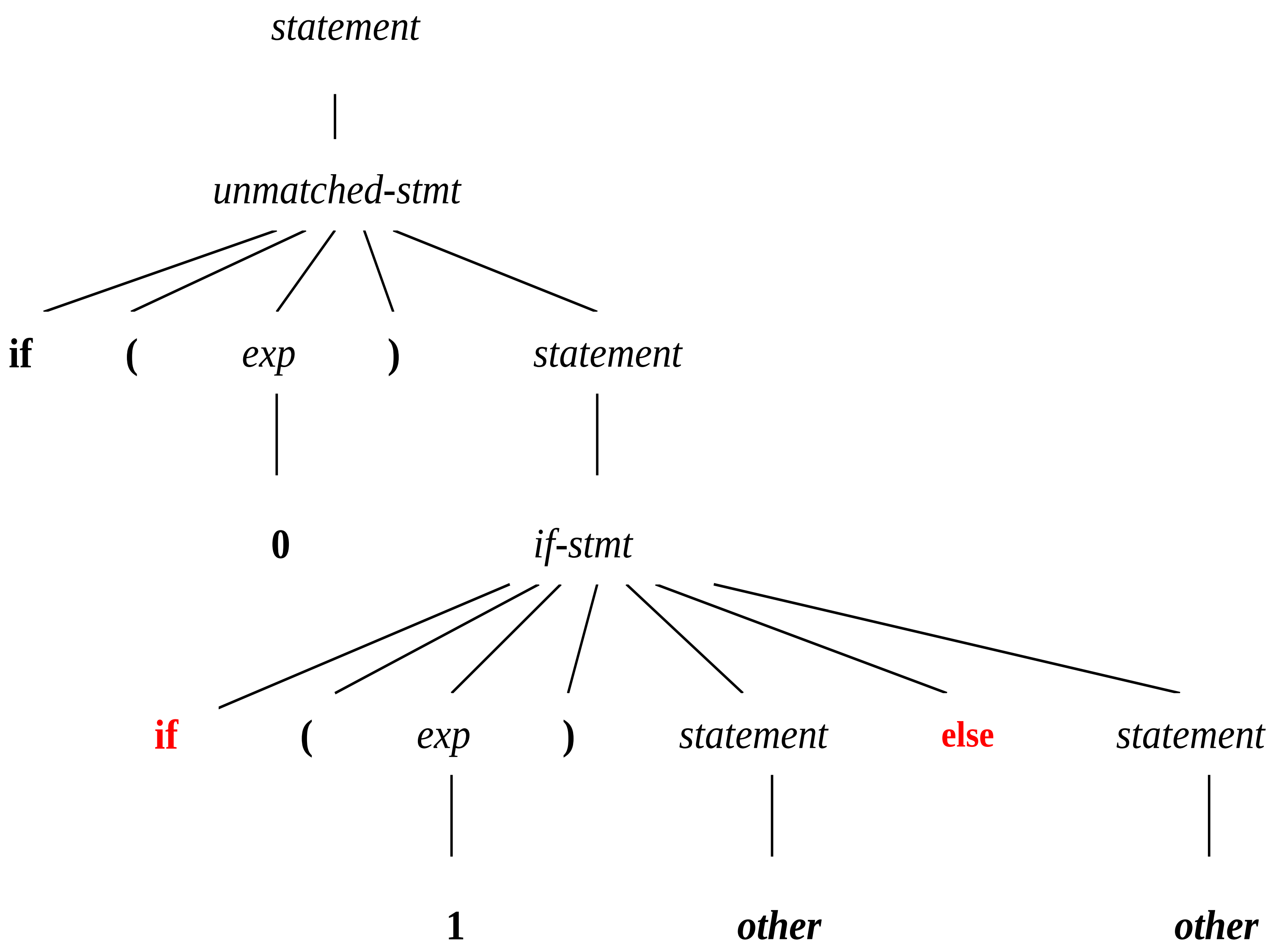
  | **if (** *exp* **)** *statement* **else** *statement*

  exp→ **0 | 1**

- This grammar is ambiguous as a result of the optional else.
- Example: consider the string

  if (0) if (1) other else other


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


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

Dangling else problem

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

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**

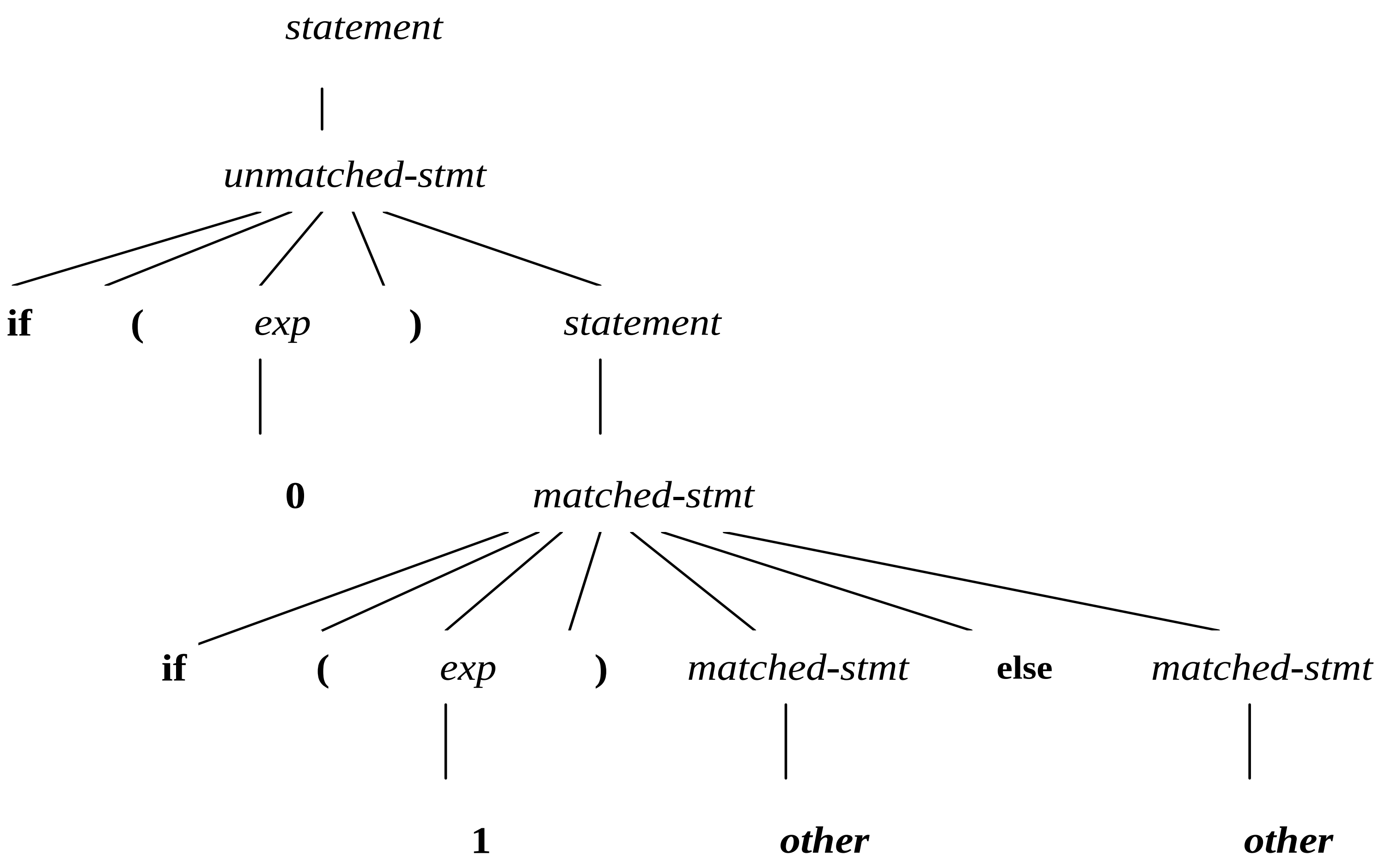
- Permit only a *matched-stmt* to come before an else in an if-statement
  - force all else-parts to be matched as soon as possible.

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**



3. Context-Free Grammars and Parsing

PART TWO

3.5 Extended Notations: EBNF

3.5.1 EBNF Notation

Special Notations for *Repetitive Constructs*

- **Notation for repetition as regular expressions use the asterisk ***

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

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

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

Examples

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

exp* → *exp addop term* | *term

***exp* → *term* { *addop term* }**

exp* → {*term addop* } *term

Special Notations for *Optional Constructs*

- **Optional constructs 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 $\rightarrow$ ***if-stmt* | other**

  if-stmt $\rightarrow$ **if (*exp*) *statement* [else *statement*]**

  exp $\rightarrow$ **0 | 1**

- ***stmt-sequence*** $\rightarrow$ ***stmt*; *stmt-sequence* | *stmt* is written as**

  stmt-sequence $\rightarrow$ ***stmt* [; *stmt-sequence*]**

3.6 Formal Properties of Context-Free Language

3.6.1 A Formal Definition of Context-Free Language

Definition

- Definition: A context-free grammar consists of the following:

  1. A set T of terminals
  2. A set N of non-terminals (disjoint from T)
  3. A set P of productions, or grammar rules, of the form $A \to \alpha$, where A *is an* element of N and α is an element of $(T \cup N)^*$ (a possibly empty sequence of terminals and non-terminals)
  4. A start symbol S from the set N

Definition

- **Let *G* be a grammar as defined above,**

  G = (T, N, P, S)
- **A derivation step over *G* is of the form**

  $\alpha A \gamma => \alpha \beta \gamma$

  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 non-terminals**
- **A sentential form:**
  - **a string α in ($T \cup N$)***

Definition

- **The relation $\alpha \overset{*}{=>} \beta$ is defined to be the transitive closure of the derivation step relation =>;**
  - **$\alpha \overset{*}{=>} \beta$ if and only if there is a sequence of zero or more derivation steps ($n \geq 0$)**
    - **$\alpha_1 => \alpha_2 => \ldots => \alpha_{n\text{-}1} => \alpha_n$**
    - **such that $\alpha = \alpha_1$, and $\beta = \alpha_n$**

  (If n = 0, then $\alpha = \beta$)

Definition

- A derivation over the grammar *G* is of the form
  - $S \overset{*}{\Rightarrow} w$
  - where $w \in T^*$ (i.e., *w* is a string of terminals only, called a **sentence**), and *S* is the start symbol of *G*
- The **language generated by *G***, written *L*(*G*), is defined as the set
  - $L(G) = \{w \in T^* \mid \text{there exists a derivation } S \overset{*}{\Rightarrow} w \text{ of } G\}$.
  - *L*(*G*) is the set of sentences derivable from *S*.

Definition

- A l**eftmost derivation** $S \overset{*}{=>}_{\text{lm}} w$
  - is a derivation in which each derivation step
  - $\alpha A \gamma => \alpha \beta \gamma$
  - is such that $\alpha \in T^*$; that is, α consists only of terminals.
- A **rightmost derivation** is one in which each derivation step
  - $\alpha A \gamma => \alpha \beta \gamma$
  - has the property that $\gamma \in T^*$.

Parse Tree over Grammar G

A rooted labeled tree with the following properties:

1. Each node is labeled with a terminal or a non-terminal 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 non-leaf node is labeled with a non-terminal.
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 non-terminals),
   then $A \rightarrow X_1\ X_2 \ldots X_n \in P$ (a production of the grammar).

Role of Derivation

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

CFG & Ambiguous

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

3.6.3 Chomsky Hierarchy and Limits of Context-Free Syntax

Context Rules

- **Free of context rule**:
  - Non-terminals appear by themselves to the left of the arrow in context-free rules.
  - A rule says that *A* may be replaced regardless of where *A* occurs.
- **Context-sensitive grammar rule**:
  - A rule would apply only if β occurs before and γ occurs after the non-terminal.
  - We would write this as
    - $\beta A \gamma \Rightarrow \beta a \gamma, (a \neq \varepsilon)$
- Context-sensitive grammars are more powerful than context-free grammars
  - but are also much more difficult to use as the basis for a parser.

Requirement of a Context-Sensitive Grammar Rule

- The C rule requires declaration before use
  - For each name, we would have to write a rule establishing its declaration prior to a potential use.
- In many languages, the length of an identifier is unrestricted,
  - The number of possible identifiers is infinite.
- Even if names are allowed to be only two charac-ters long
  - The potential for hundreds of new grammar rules. Clearly, this is an impossible situation.

Unrestricted Grammars

- More general than the context-sensitive grammars.
  - It has grammar rules of the form

    $\alpha \rightarrow \beta$
- where there are no restrictions on the form of the strings α and β (except that α cannot be β)

Types of Grammars

- The language classes are referred to as the **Chomsky hierarchy**

  type 0： unrestricted grammar, equivalent to Turing machines

  type 1： context sensitive grammar

  type 2： context free grammar, equivalent to pushdown automaton

  type 3： regular grammar , equivalent to finite automata

- These grammars represent distinct levels of computational power.

End of Part Two

THANKS