COMPILER CONSTRUCTION

Principles and Practice

Kenneth C. Louden

3. Context-Free Grammars and Parsing

PART ONE

Contents

PART ONE

PART TWO

Introduction

- Parsing is the task of **Syntax Analysis**
  - Determining the syntax, or structure, of a program.
- The syntax is defined by the **grammar rules of a Context-Free Grammar**
  - 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.

- *Sequence of tokens* $\xrightarrow{\text{Parser}}$ *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 as it is needed 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

- The parser incorporate all the other phases of a compiler *in a single-pass compiler*
  - No explicit syntax tree needs to be constructed
  - The parser steps themselves will represent the syntax tree implicitly by a call **Parse ( )**

Issues of the Parsing

- In Multi-Pass, the *further passes will use the syntax tree* as their input
  - The structure of the syntax tree is heavily dependent on the particular syntactic structure of the language
  - This tree is usually defined as a dynamic data structure
  - Each node consists of a record whose fields include the attributes needed for the remainder of the compilation process (i.e., not just those computed by the parser).

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.

Issues of the Parsing

- 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)
- A parser may perform **error repair**
  - Error recovery is the reporting of meaningful error messages and the resumption of parsing as close to the actual error as possible

Back

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:

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

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

3.2.1 Comparison to Regular Expression Notation

Comparing an Example

- The context-free grammar:

  exp → *exp op exp* | (*exp*) | ***number***

  op → + | − | *

- The regular expression:

  *number = digit digit**

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

Basic Regular Expression Rules

- Three operations:
  - Choice, concatenation, repetition
- Equal sign represents the definition of a name for a regular expression;
- Name was written in italics to distinguish it from a sequence of actual characters.

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)

Grammars Rules

- Grammar rules use regular expressions as components
- The notation was developed by John Backus and adapted by Peter Naur for the Algol60 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

Symbols of Grammar Rules

- Grammar rules are defined over *an alphabet*, or set of symbols.
  - The symbols are tokens representing strings of characters
- Using the regular expressions to represent the tokens
  - A token is a fixed symbol, as in the reserved word while or the special symbols such as + or :=, write the string itself in the code font used in Chapter 2
  - Tokens such as identifiers and numbers, representing more than one string, use code font in italics, just as though the token is a name for a regular expression.

Symbols in TINY

- Represent the alphabet of tokens for the TINY language:

  {if. then, else, end, repeat, until, read, write, *identifier, number,* +, -, *, /, =, <, (,), ; , := }

- Instead of the set of tokens (as defined in the TINY scanner)

  {IF,THEN,ELSE,END,REPEAT,UNTIL,READ,WRITE,ID,NUM, PLUS,MINUS,TIMES, OVER,EQ, LT, LPAREN,RPAREN, SEMI, ASSIGN }

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"→".
  - 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 metasymbol "| ".

Construction of a CFG rule

- A grammar rule in BNF 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 sequences of symbols and structure names within each choice defines the layout of the structure
  - For example:
    - *exp* → *exp op exp* | *(exp)* | ***number***
    - *op* → + | – | *

More about the Conventions

- The meta-symbols and conventions used here are in wide use but there is no universal standard for these conventions
  - Common alternatives for the arrow metasymbol '→' include "=" (the equal sign), ":" (the colon), and "::=" ("double-colon-equals")
- In normal text files, replacing the use of italics, by surrounding structure names with angle brackets <...>
- and by writing italicized token names in uppercase
- Example:
  - **<exp> ::= <exp> <op> <exp> | (<exp>) | NUMBER**
  - **<op> ::= + | - | ***

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

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 sides 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 → *exp op exp* / *(exp)* / ***number***

  op → + / – / *

- A 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 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* | *exp* =>* *s* }

   G represents the expression grammar

   s represents *an arbitrary string of token symbols*

      (sometimes called a sentence)

   The symbols =>* stand for a derivation con-sisting of a sequence of replacements as described earlier.

      (The asterisk is used to indicate a sequence of steps, much as it indicates repetition in regular expressions.)

   Grammar rules are sometimes called productions

      Because they "produce" the strings in *L(G)* via derivations

Grammar for a Programming Language

- The grammar for a programming language often defines a structure called ***program***
- The language of this structure is the set of all syntactically legal programs of the programming language.
  - For example: a BNF for Pascal

    program → *program-heading; program-block*

    program-heading → *….*

    program-block → *…..*
- The first rule says that a program consists of a program heading, followed by a semicolon, followed by a program block, followed by a period.

Symbols in rules

- **Start symbol**
  - The most general structure is listed first in the grammar rules.
- **Non-terminals**
  - Structure names are also called non-terminals, since they always must be replaced further on in a derivation.
- **Terminals**
  - Symbols in the alphabet are called terminals, since 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

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

  $\{ (^n\ a\ )^n \mid n$ an integer >= 0 }

- Derivation for ((a))

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

Examples

Example 3.2:

- The grammar G with the single grammar rule $E \rightarrow (E)$
- 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

  $E \rightarrow E + a \mid a$

- 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 → *a*

Prove the Example 3.3

(1) Every string $a + a + \ldots + a$ is in $L(G)$ by induction on the number of a's.

- The derivation $E => a$ shows that a is in $L(G);$
- Assume now that $s = a + a + \ldots + a,$ with $n-1$ $a's,$ is in L(G).
- Thus, there is a derivation $E =>^* s:$ Now the derivation E $=>$ E + a $=>^*$ $s + a$ shows that the string $s + a,$ with $n +$ a's, is in L(G).

Prove the example 3.3

(2) Any strings from *L(G)* must be of the form:

$$a + a + \ldots . + a$$

- If the derivation has length 1, then it is of the form E => *a,* and so *s* is of the correct form.
- Now, assume the truth of the hypothesis for all strings with derivations of length $n - 1$;
- And let $E \Rightarrow^* s$ be a derivation of length $n > 1$. This derivation must begin with the replacement of E by E + a, and so is of the form $E \Rightarrow E + a \Rightarrow^* s' + a = s$. Then, s' has a derivation of length n - 1, and so is of the form $a + a + \ldots + a$. Hence, *s* itself must have this same form.

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 lan-guage 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

- **The grammar rule:**
  - $A \rightarrow A\, a \mid a$ **or** $A \rightarrow a\, A \mid a$
  - **Generates the language {an | n an integer >=1 }**

    (the set of all strings of one or more a's)
  - **The same language as that generated by the regular expression a+**
- **The string *aaaa* can be generated by the first grammar rule with the derivation**
  - $A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa \Rightarrow aaaa$

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:** $A \rightarrow A\alpha / \beta$
  - **where** α **and** β **represent arbitrary strings and** β **does not begin with** A**.**
- **This rule generates all strings of the form**
  - $\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha$, **...**
  - **(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** $A \rightarrow \alpha\, A$ / β
  - **(where** β **does not end in** A**)**
  - **generates all strings** $\beta, \alpha\beta, \alpha\alpha\beta, \alpha\alpha\alpha\beta$**, ...**

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
  - *empty* → 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* → *A a* | ε *or* *A* → *a A* | ε
- Both grammars generate the language
  - { an | n an integer >= 0} = *L*(a*).

Examples

- **Example 3.5:**
  - *A* → *(A) A* | ε
  - **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.**

Examples

- Example 3.7: Consider the following grammar *G* for a sequence of statements:

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

  stmt → s

- This grammar generates sequences of one or more statements separated by semicolons
  - (statements have been abstracted into the single terminal s*):*
- *L*(*G*)= { s*,* s; s, s; s; s,...)

Examples

- If allow statement sequences to be empty, write the following grammar *G'*:
  - *stmt-sequence* → *stmt* ; *stmt-sequence* | ε
  - *stmt* → s
  - semicolon is a terminator rather than a separator:
- *L*(*G'*)= { ε, s;, s;s;, s;s;s;,... }
- If allow statement sequences to be empty, but retain the semicolon as a separator, write the grammar as follows:
  - *stmt-sequence* → *nonempty-stmt-sequence* |ε
  - *nonempty-stmt-sequence* → *stmt*; *nonempty-stmt-sequence* / *stmt*
    stmt→ s
- *L*(*G*)= {ε, s, s; s, s; s; s,...)

Back

3.3 Parse trees and abstract syntax trees

3.3.1 Parse trees

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

Derivation V.S. Structure

(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 V.S. Structure

(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_→ *]

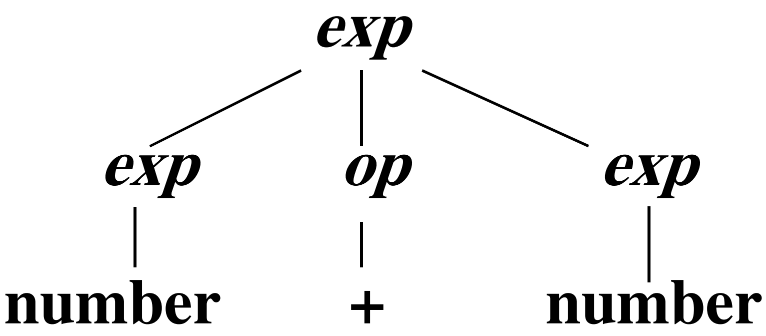
(8) =>(_number - number)_ * _number_ [_exp_ → number]

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;
  - 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

Parsing Tree

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



exp

exp ***op*** ***exp***

number **+** **number**

- **The above parse tree is 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***

Parsing Tree

- Neither leftmost nor rightmost derivation

  （1） *exp* => *exp op exp*

  （2） => *exp* + *exp*

  （3） => *number* + *exp*

  （4） => *number* + *number*

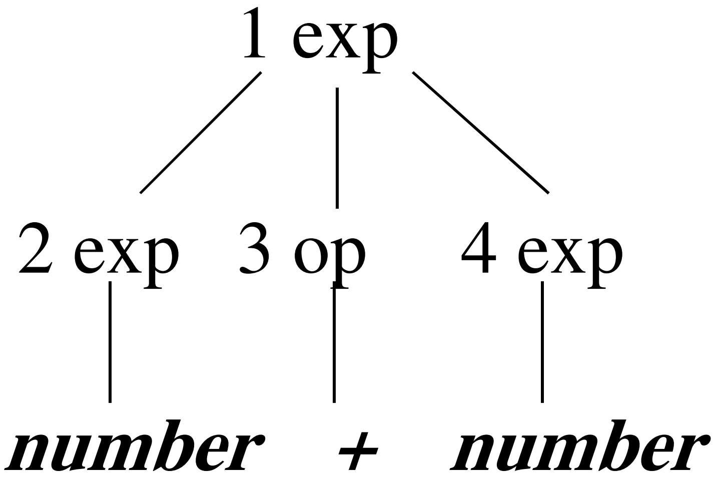
- Generally, a parse tree corresponds to many derivations
  - represent the same basic structure for the parsed string of terminals.
- It is possible to distinguish particular derivations that are uniquely associated with the parse tree.

Parsing Tree

- **A left-most derivation:**
  - **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.**
- **A rightmost derivation:**
  - **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.**

Parsing Tree

- The parse tree corresponds to the first derivation.



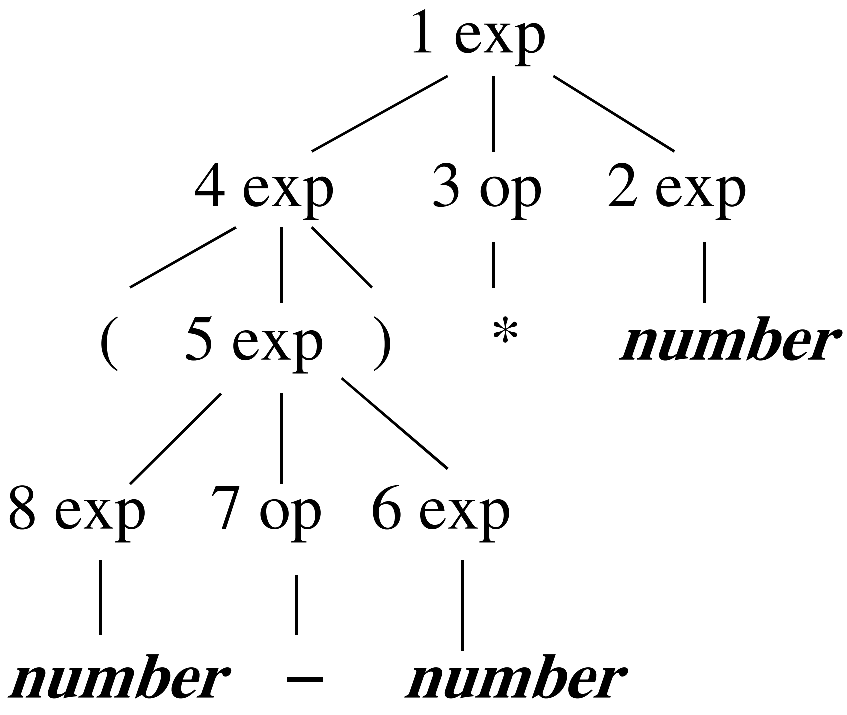
1 exp

2 exp 3 op 4 exp

number* + *number

Example: The expression (34-3)*42

- The parse tree for the above arithmetic expression



1 exp

4 exp 3 op 2 exp

(5 exp) * ***number***

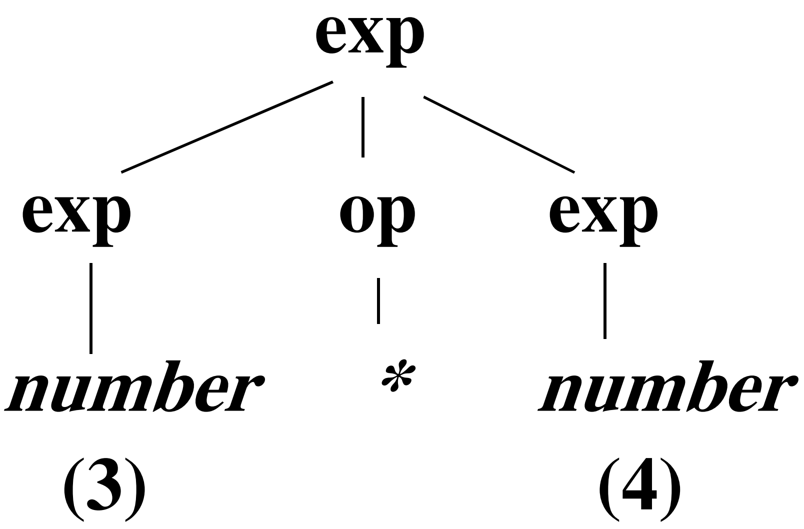
8 exp 7 op 6 exp

number – ***number***

3.3.2 Abstract syntax trees

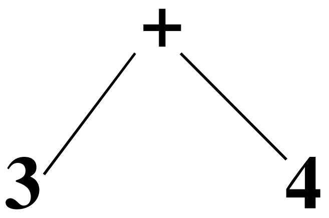
Way Abstract Syntax-Tree

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



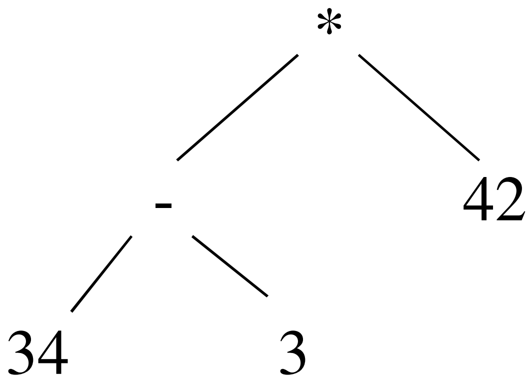
Why Abstract Syntax-Tree

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

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.

Abstract Syntax Trees or Syntax Trees

- A parse tree is a representation for the structure of ordinary called **concrete syntax** when comparing it to abstract syntax.
- Abstract syntax can be given a formal definition using a BNF-like notation, just like concrete syntax.
- The BNF-like rules for the abstract syntax of the simple arithmetic expression:

$$exp \rightarrow OpExp(op,exp,exp) \mid ConstExp(integer)$$
$$op \rightarrow Plus \mid Minus \mid Times$$

Abstract Syntax Trees or Syntax Trees

- Data type declaration.：the C data type declarations.

```
typedef enum {Plus,Minus,Times} OpKind;
typedef enum {OpK.ConstK} ExpKind;
typedef struct streenode
      { ExpKind kind;
        OpKind op;
        struct streenode *lchild,*rchild;
        int val;
         } STreeNode;
 typedef STreeNode *SyntaxTree;
```

Examples

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

statement → *if-stmt* | ***other***

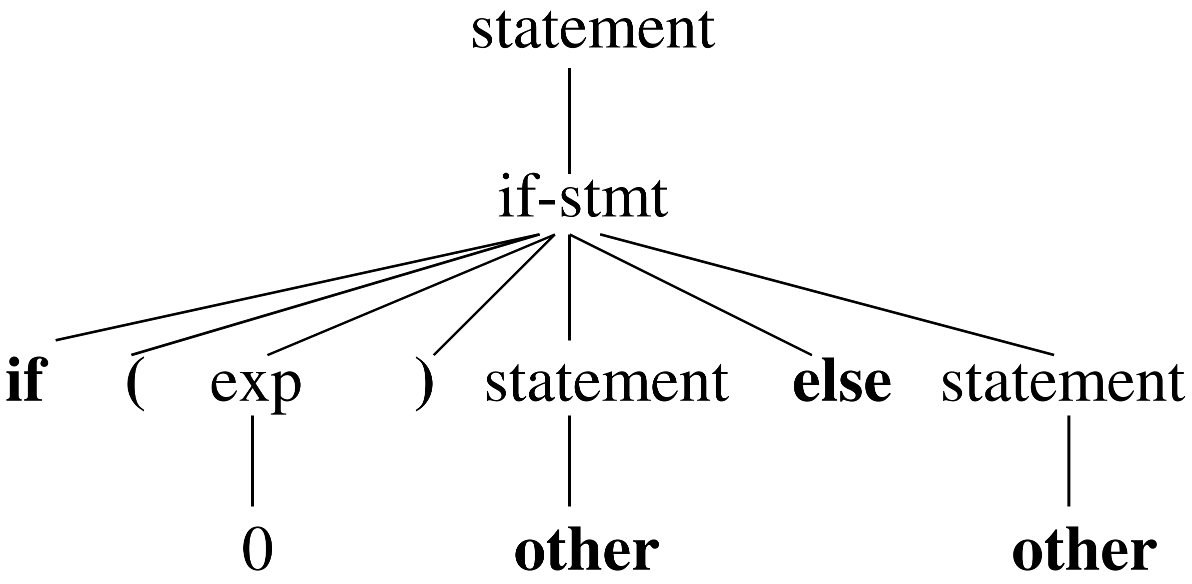
if-stmt → **if** (*exp*) *statement*

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

exp → **0 | 1**

Examples

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



statement

if-stmt

if (exp) statement **else** statement

0 **other** **other**

Examples

- Using the grammar of Example 3.6

statement → *if-stmt* | ***other***

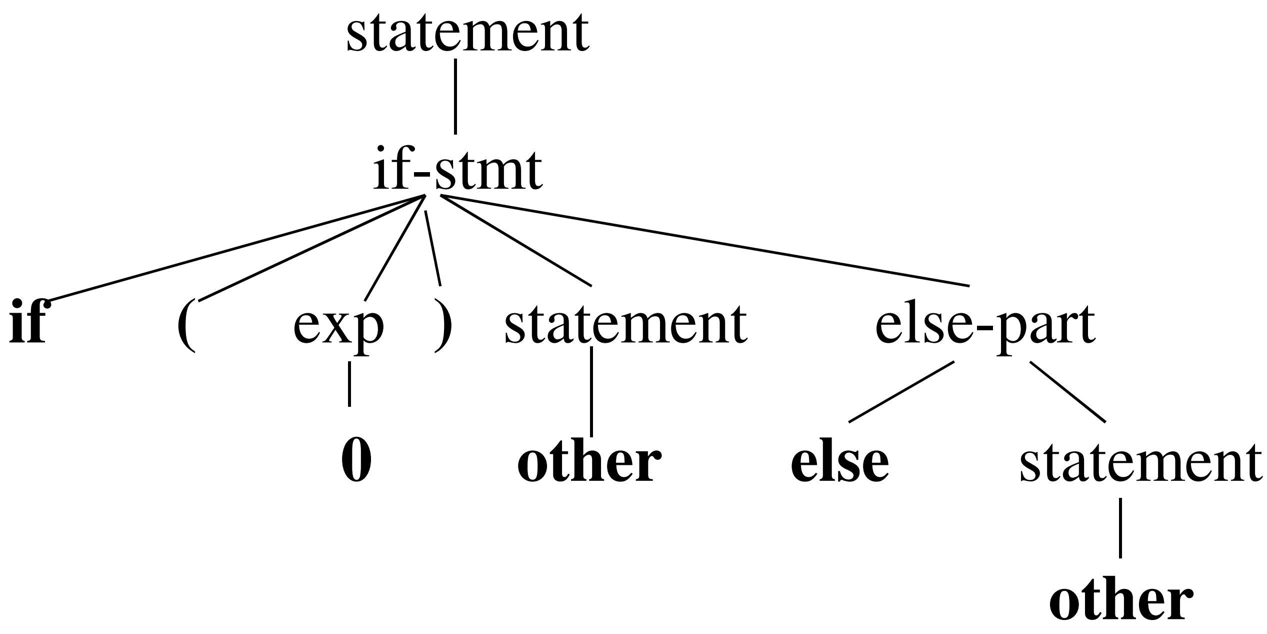
if-stmt → **if (** *exp* **)** *statement else-part*

else-part → **else** *statement* | ε

exp → **0 | 1**

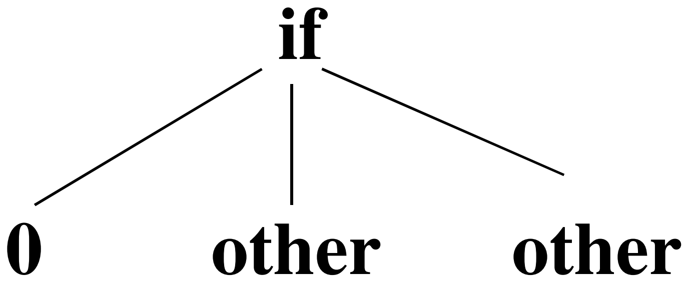
Examples

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



Examples

- 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**



Examples

- A set of C declarations that would be appropriate for the structure of the statements and expressions in this example' is as follows:

```
typedef enum  {ExpK, StmtK) NodeKind;
typedef enum  {Zero, One} ExpKind;
typedef enum {IfK, OtherK) StmtKind;
typedef struct streenode
{ NodeKind kind;
  ExpKind ekind;  .
  StmtKind skind;
  struct streenode
    *test,*thenpart,*elsepart;
  } STreeNode;
typedef STreeNode * SyntaxTree;
```

Examples

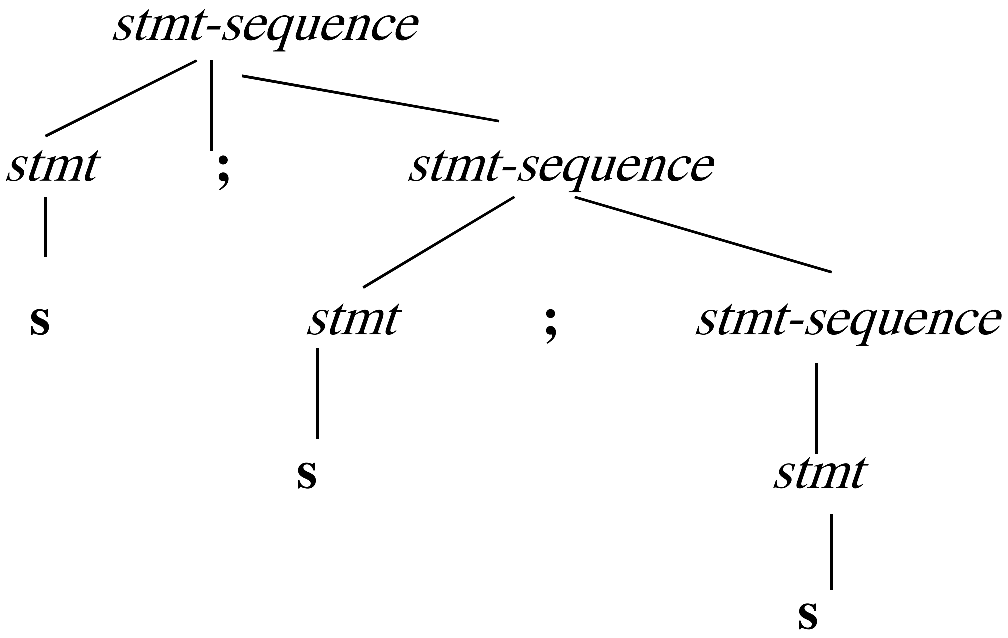
- Example 3.9:
  - The grammar of a sequence of statements separated by semicolons from Example 3.7:

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

    stmt → **s**

Examples

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



stmt-sequence

stmt **;** *stmt-sequence*

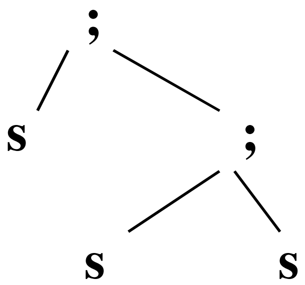
s *stmt* **;** *stmt-sequence*

s *stmt*

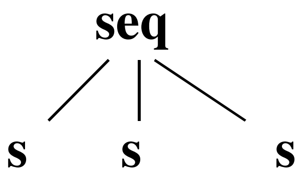
s

Examples

- 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

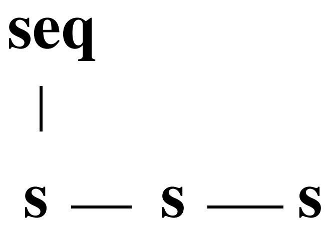


Problem & Solution

- The solution: use the standard **leftmost-child right-sibling** representation for a tree (presented in most data structures texts) to deal with arbitrary number of children
  - The only physical link from the parent to its children is to the leftmost child.
  - The children are then linked together from left to right in a standard linked list, which are called **sibling** links to distinguish them from parent-child links.

Problem & Solution

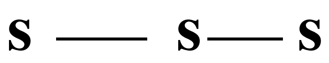
- The previous tree now becomes, in the leftmost-child right-sibling arrange-ment:



seq

s — s — s

- With this arrangement, we can also do away with the connecting **seq** node, and the syntax tree then becomes simply:



s — s — s

Back

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:

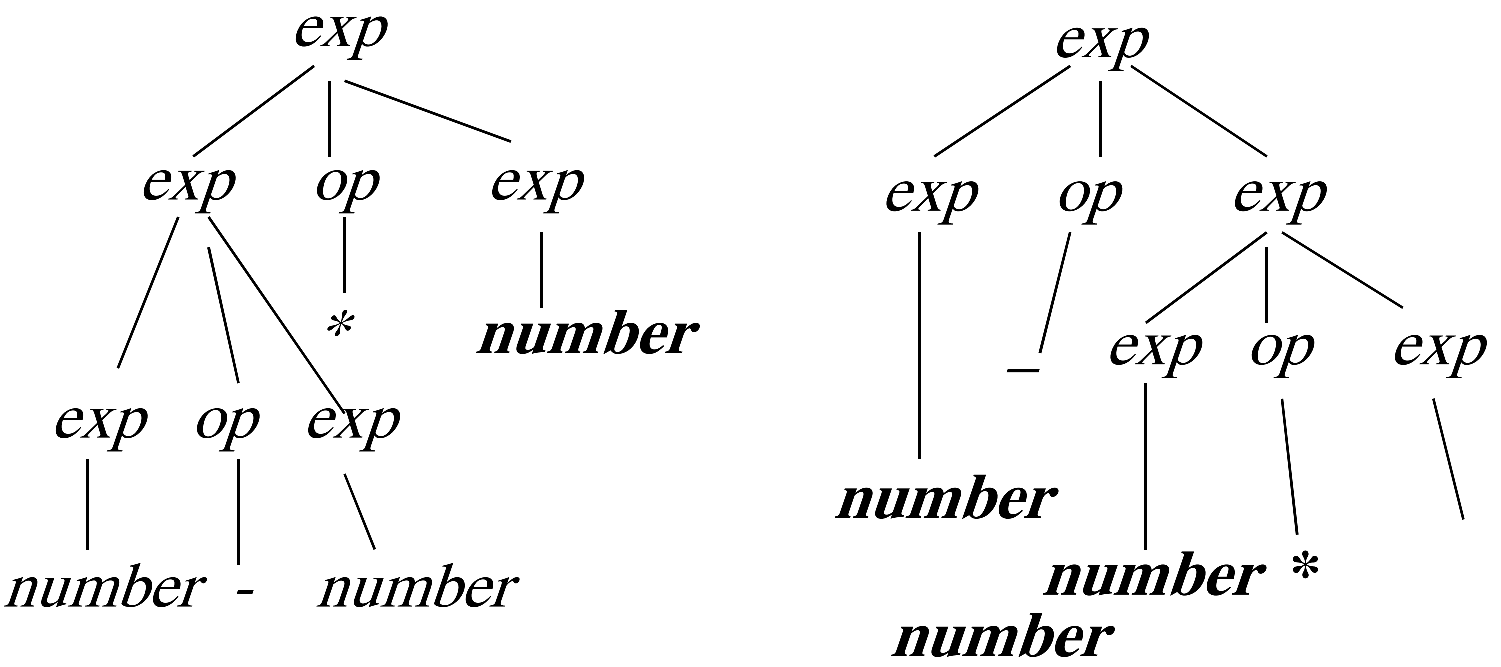
exp → *exp op exp* | (*exp*) | ***number***

op → **+ | - | ***

The string: 34-3*42

What is Ambiguity

This string has two different parse trees.



What is Ambiguity

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*

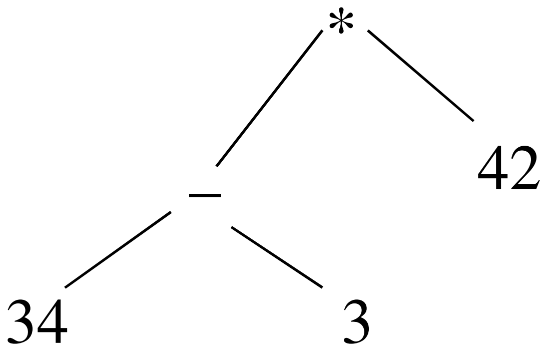
=>***number - number*** op *exp*

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

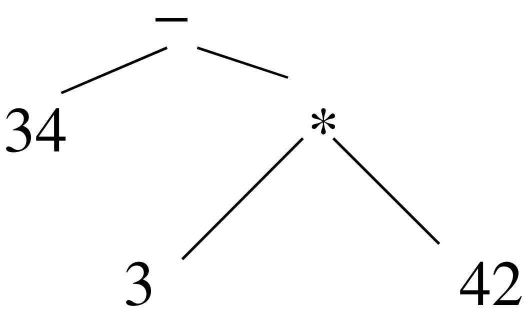
=> ***number - number*** * ***number***

What is Ambiguity

The associated syntax trees are



AND



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

An Ambiguous Grammar

- 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.

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

- 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.

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.
  - S***pecify that all three of these operations are left associative***

Remove the Ambiguity in simple Expression Grammar

- 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→ ***+*** **/-** **/** *

Remove the Ambiguity in simple Expression Grammar

- 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

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.

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

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→ **+ | -**

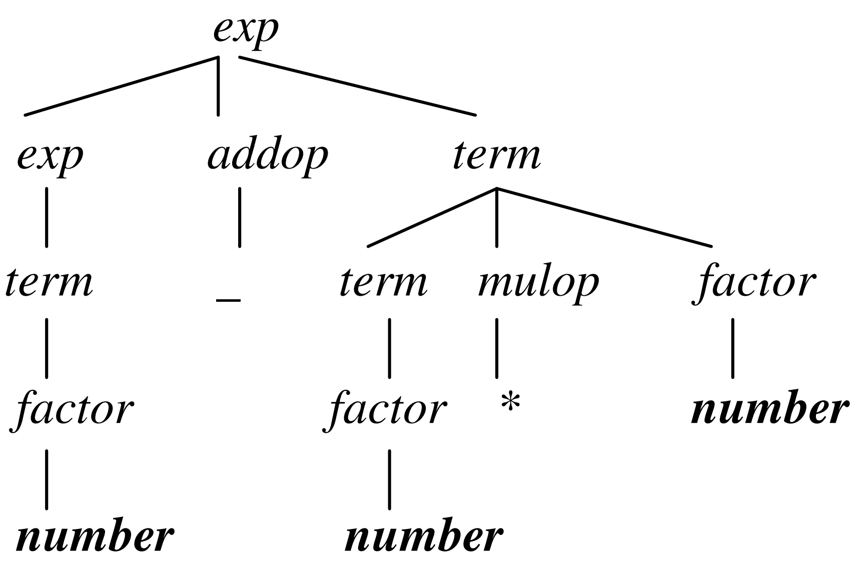
term → *term mulop factor* | *factor*

mulop → *

factor → *(exp)* | ***number***

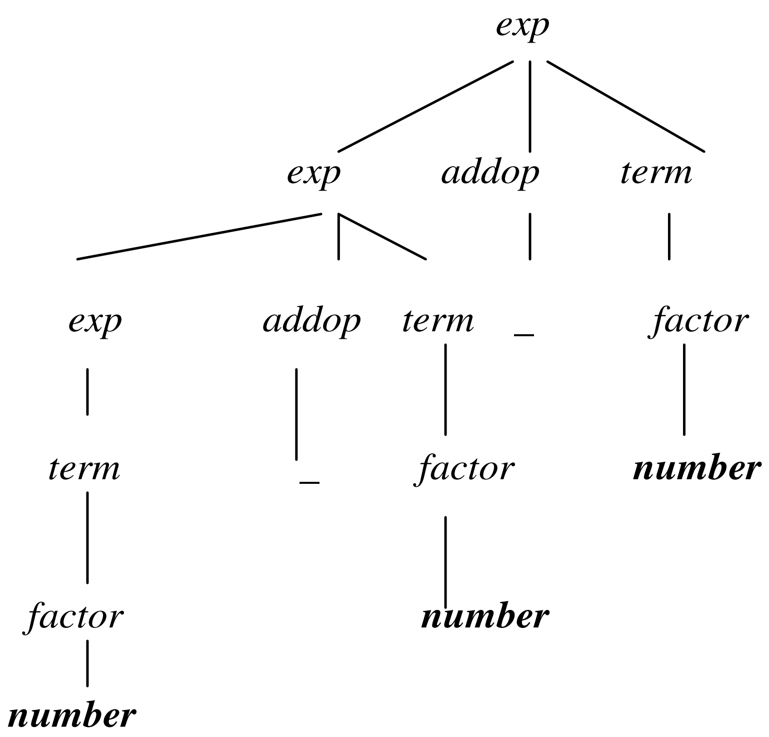
New Parse Tree

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



New Parse Tree

- 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

An Ambiguity Grammar

- Consider the grammar from:

  statement → *if-stmt* | ***other***

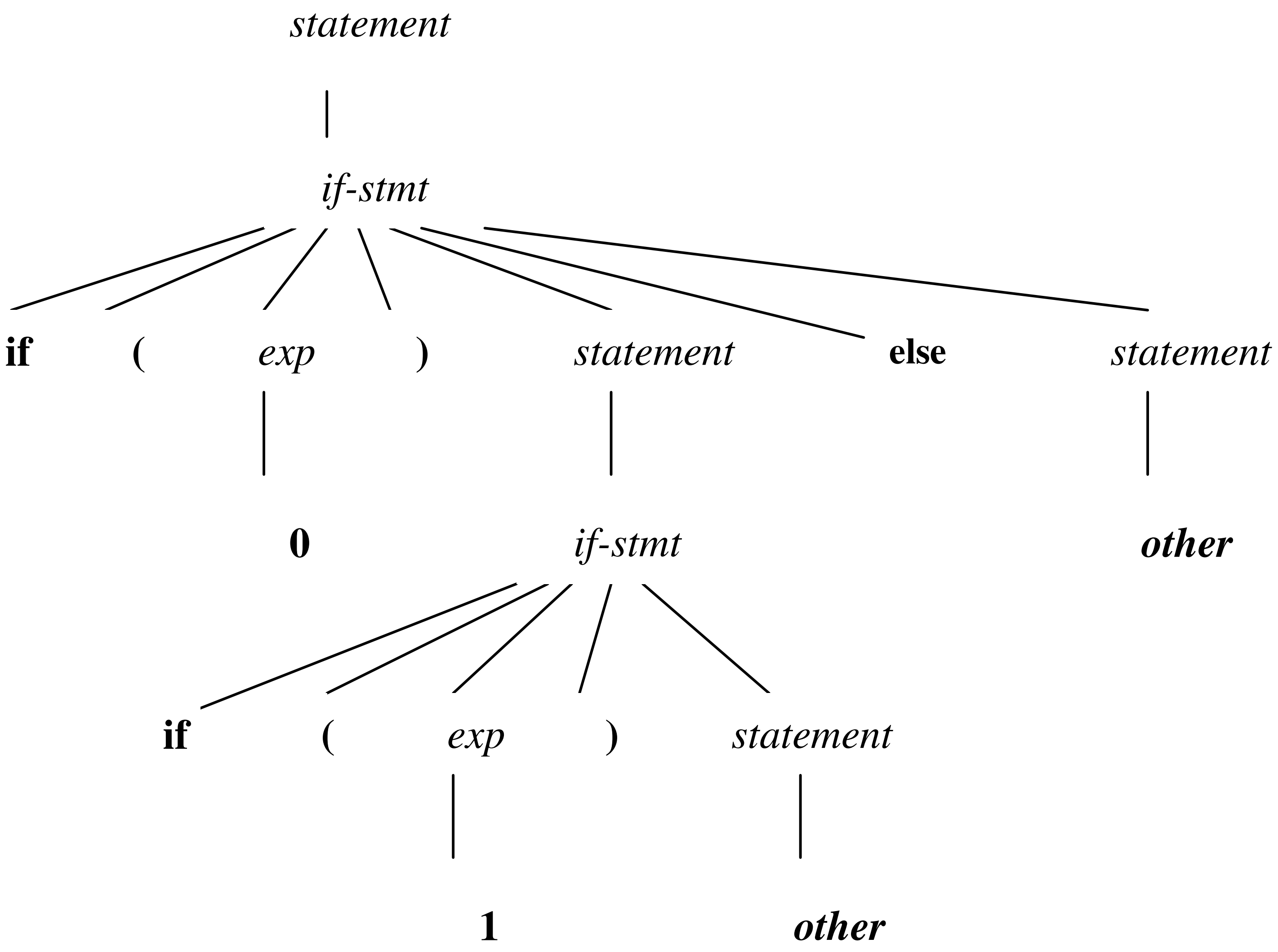
  if-stmt → **if (** *exp* **)** *statement*

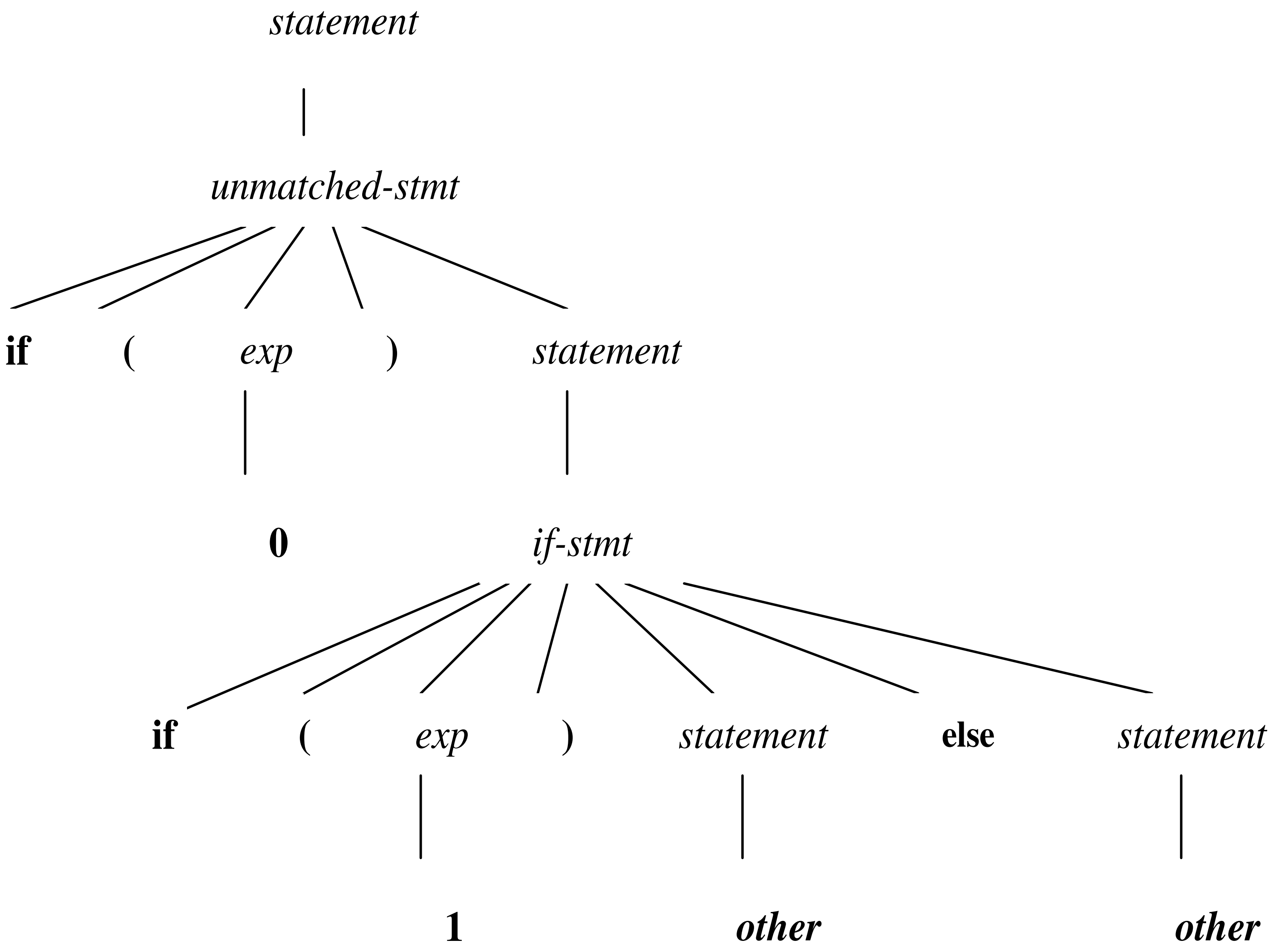
  | **if (** *exp* **)** *statement* **else** *statement*

  exp→ **0 | 1**

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

  if (0) if (1) other else other


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.
  - 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;
```

A Solution to the dangling else ambiguity in the BNF

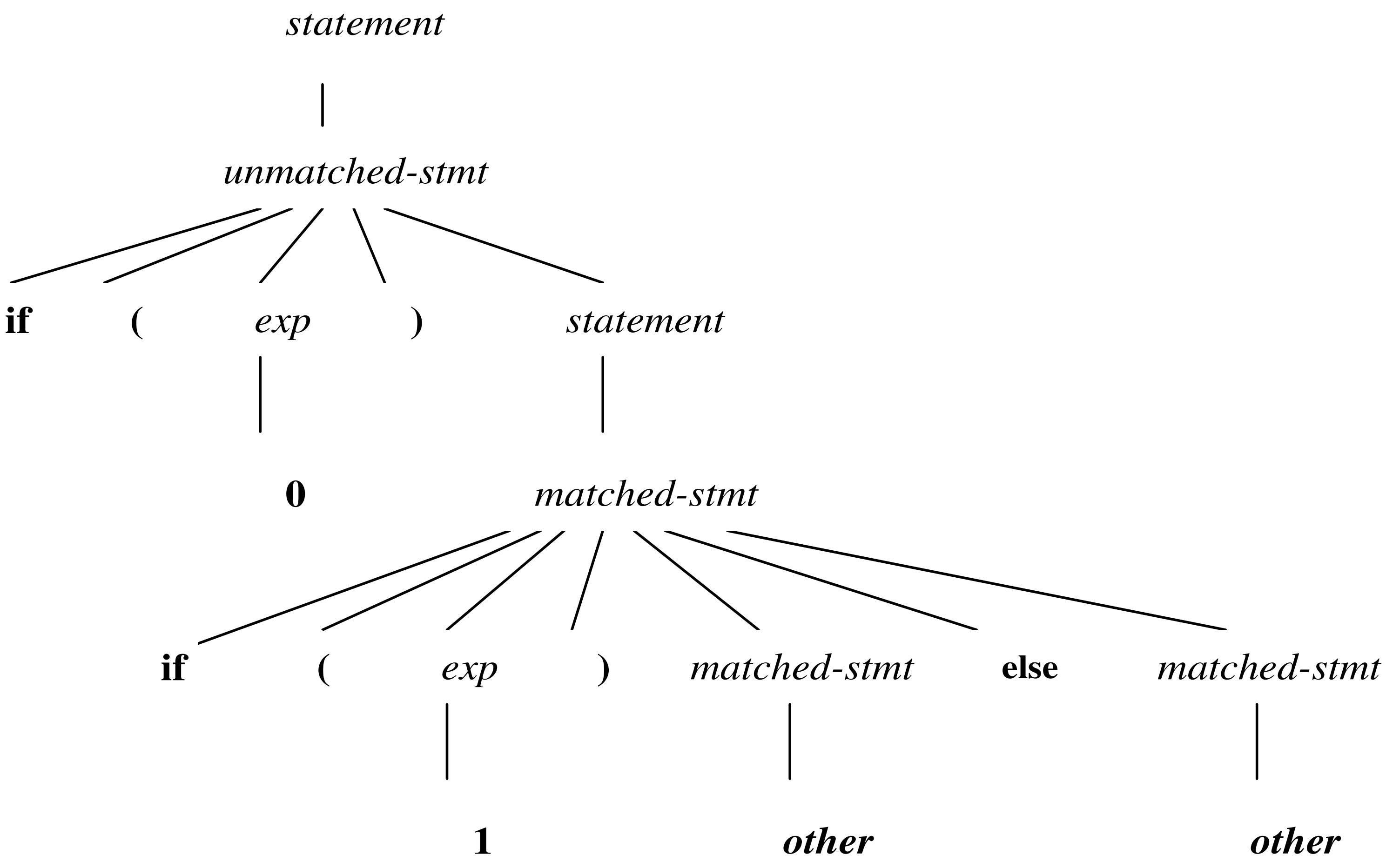
statement → *matched-stmt* | *unmatched-stmt*

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
  - forcing all else-parts to be matched as soon as possible.


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

More about dangling else

- The dangling else problem has its origins in the syntax of Algol60.
- It is possible to design the syntax in such a way that the dangling else problem does not appear.
  - 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).
  - Use a **bracketing keyword** for the if-statement languages that use this solution include Algol68 and Ada.

More About Dangling else

For example, in Ada, the programmer writes

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

Associate the else-part with the second if-statement, the programmer writes

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

More about dangling else

- BNF in Ada (somewhat simplified) is

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

3.4.4 Inessential ambiguity

Why Inessential

- 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,
- Such an ambiguity could be called an **inessential ambiguity**

Why Inessential

- **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 asso-ciative if (a • *b)* • *c = a* • *(b* • *c)* for all values *a, b,* and c).
  - – In this case the syntax trees are still distinct, but represent the same semantic value, and we may not care which one we use.
- Nevertheless, a parsing algorithm will need to apply some disambiguating rule that the compiler writer may need to supply

End of Part One

THANKS