Grammars and Derivations

Objectives

+ Discuss the most common method of describing syntax - context-free grammars (also known as Backus-Naur Form). This includes a discussion of derivations, parse trees, ambiguity, descriptions of operator precedence and associativity, and extended Backus-Naur Form.

1. Grammars and Derivations

– A grammar is a generative device for defining languages.
– The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the start symbol.

*This sequence of rule applications is called **a derivation***.

– In a grammar for a complete programming language, the start symbol represents a complete program and is often named <program>.
– The simple grammar shown in Example 1 is used to illustrate derivations.

Example 1 A Grammar for a Small Language

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt>
            | <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
             | <var> - <var>
             | <var>
```

– The language described by the grammar of Example 1 has only one statement form: assignment.
– A program consists of the special word **begin**, followed by a list of statements separated by semicolons, followed by the special word **end**.
– An expression is either a single variable or two variables separated by either a + or - operator.
– The only variable names in this language are A, B, and C.
– A derivation of a program in this language follows:

```
<program> => begin <stmt_list> end
          => begin <stmt> ; <stmt_list> end
          => begin <var> = <expression> ; <stmt_list> end
          => begin A = <expression> ; <stmt_list> end
          => begin A = <var> + <var> ; <stmt_list> end
          => begin A = B + <var> ; <stmt_list> end
          => begin A = B + C ; <stmt_list> end
          => begin A = B + ; <stmt> end
          => begin A = B + C ; <var> = <expression> end
          => begin A = B + C ; B = <expression> end
          => begin A = B + C ; B = <var> end
          => begin A = B + C ; B = C end
```

–

– This derivation, like all derivations, begins with the start symbol, in this case <program>. The symbol => is read “derives.”
– Each successive string in the sequence is derived from the previous string by replacing one of the nonterminals with one of that nonterminal’s definitions.
– Each of the strings in the derivation, including <program>, is called a **sentential form**.
– In this derivation, the replaced nonterminal is always the leftmost nonterminal in the previous sentential form. Derivations that use this order of replacement are called **leftmost derivations**.
– The derivation continues until the sentential form contains no nonterminals. That sentential form, consisting of only terminals, or lexemes, is the generated sentence.
– In addition to leftmost, a derivation may be rightmost or in an order that is neither leftmost nor rightmost. Derivation order has no effect on the language generated by a grammar.
– By choosing alternative RHSs of rules with which to replace nonterminals in the derivation, different sentences in the language can be generated. By exhaustively choosing all combinations of choices, the entire language can be generated.
– This language, like most others, is infinite, so one cannot generate all the sentences in the language in finite time.
– Example 2 is another example of a grammar for part of a typical programming language.

Example 2 **A Grammar for Simple Assignment Statements**

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
       | <id> * <expr>
       | ( <expr>)
       | <id>
```

– Shortly after Chomsky’s work on language classes, the ACM-GAMM group began designing ALGOL 58. A landmark paper describing ALGOL 58 was presented by John Backus, a prominent member of the ACM-GAMM group, at an international conference in 1959.
– The grammar of Example 2 describes assignment statements whose right sides are arithmetic expressions with multiplication and addition operators and parentheses.
– For example, the statement

```
A = B * ( A + C )
```

is generated by the leftmost derivation:

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <id> * <expr>
         => A = B * <expr>
         => A = B * ( <expr>)
         => A = B * ( <id> + <expr>)
         => A = B * ( A + <expr>)
         => A = B * ( A + <id>)
         => A = B * ( A + C )
```

2 Parse Trees

– One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the languages they define.
– These hierarchical structures are called parse trees.

– For example, the parse tree in Figure 1 shows the structure of the assignment statement derived previously.

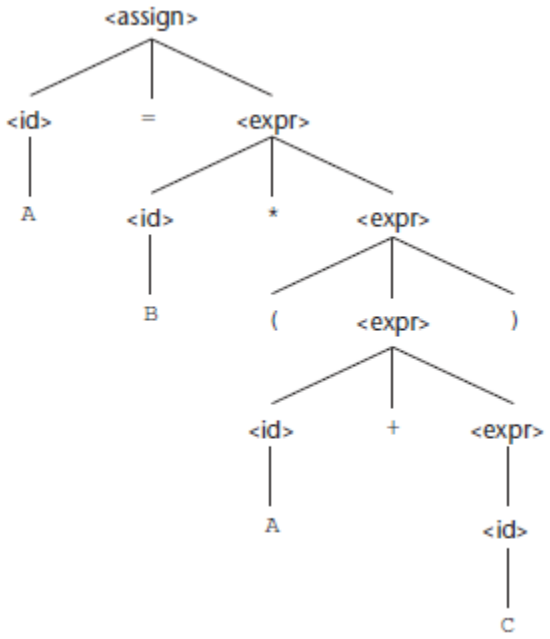

Figure 1 A parse tree for the simple statement A = B * (A + C)

– Every internal node of a parse tree is labeled with a nonterminal symbol; every leaf is labeled with a terminal symbol.
– Every subtree of a parse tree describes one instance of an abstraction in the sentence.

3 Ambiguity

– A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be ambiguous.
– Consider the grammar shown in Example 3, which is a minor variation of the grammar shown in Example 2.

Example 3 An Ambiguous Grammar for Simple Assignment Statements

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
       | <expr> * <expr>
       | ( <expr> )
       | <id>
```

The grammar of Example 3 is ambiguous because the sentence

```
A = B + C * A
```

has two distinct parse trees, as shown in Figure 2.

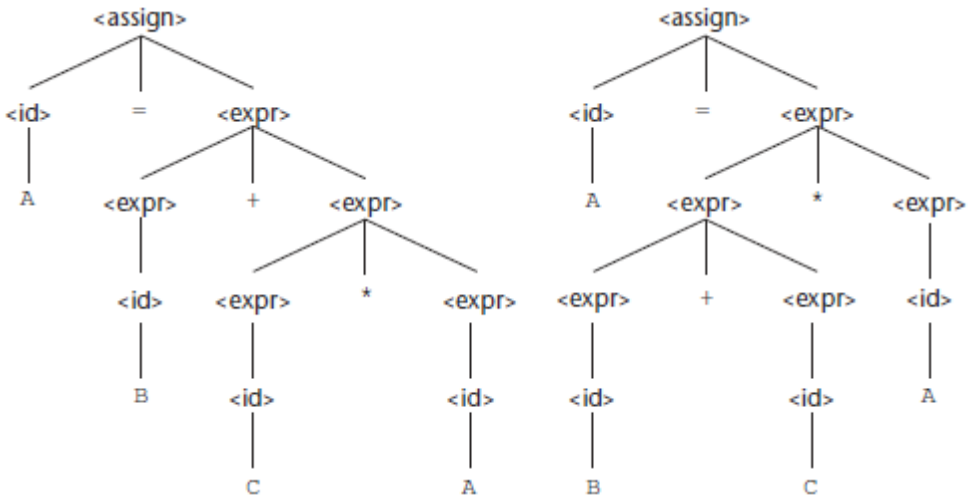

Figure 2 Two distinct parse trees for the same sentence, A = B + C * A

- The ambiguity occurs because the grammar specifies slightly less syntactic structure than does the grammar of Example 2.
- Rather than allowing the parse tree of an expression to grow only on the right, this grammar allows growth on both the left and the right.
- Syntactic ambiguity of language structures is a problem because compilers often base the semantics of those structures on their syntactic form.
- Specifically, the compiler chooses the code to be generated for a statement by examining its parse tree.
- If a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely.
- There are several other characteristics of a grammar that are sometimes useful in determining whether a grammar is ambiguous. They include the following:

  (i) if the grammar generates a sentence with more than one leftmost derivation and
  (ii) if the grammar generates a sentence with more than one rightmost derivation.

- Some parsing algorithms can be based on ambiguous grammars. When such a parser encounters an ambiguous construct, it uses nongrammatical information provided by the designer to construct the correct parse tree.
- In many cases, an ambiguous grammar can be rewritten to be unambiguous but still generate the desired language.

3 Operator Precedence

- When an expression includes two different operators, for example, `x + y * z`, one obvious semantic issue is the order of evaluation of the two operators (for example, in this expression is it add and then multiply, or vice versa?).
- This semantic question can be answered by assigning different precedence levels to operators.
- For example, if * has been assigned higher precedence than + (by the language designer), multiplication will be done first, regardless of the order of appearance of the two operators in the expression.

- As stated previously, a grammar can describe a certain syntactic structure so that part of the meaning of the structure can be determined from its parse tree.

– In particular, the fact that an operator in an arithmetic expression is generated lower in the parse tree (and therefore must be evaluated first) can be used to indicate that it has precedence over an operator produced higher up in the tree.
– In the first parse tree of Figure 2, for example, the multiplication operator is generated lower in the tree, which could indicate that it has precedence over the addition operator in the expression.
– The second parse tree, however, indicates just the opposite. It appears, therefore, that the two parse trees indicate conflicting precedence information.

– Notice that although the grammar of Example 2 is not ambiguous, the precedence order of its operators is not the usual one. In this grammar, a parse tree of a sentence with multiple operators, regardless of the particular operators involved, has the rightmost operator in the expression at the lowest point in the parse tree, with the other operators in the tree moving progressively higher as one moves to the left in the expression.
– For example, in the expression A + B * C, * is the lowest in the tree, indicating it is to be done first. However, in the expression A * B + C, + is the lowest, indicating it is to be done first.
– A grammar can be written for the simple expressions we have been discussing that is both unambiguous and specifies a consistent precedence of the + and * operators, regardless of the order in which the operators appear in an expression.

– The correct ordering is specified by using separate nonterminal symbols to represent the operands of the operators that have different precedence. This requires additional nonterminals and some new rules.
– Instead of using <expr> for both operands of both + and *, we could use three nonterminals to represent operands, which allows the grammar to force different operators to different levels in the parse tree.
– If <expr> is the root symbol for expressions, + can be forced to the top of the parse tree by having <expr> directly generate only + operators, using the new nonterminal, <term>, as the right operand of +. Next, we can define <term> to generate * operators, using <term> as the left operand and a new nonterminal, <factor>, as its right operand.
– Now, * will always be lower in the parse tree, simply because it is farther from the start symbol than + in every derivation. The grammar of Example 4 is such a grammar.

Example 4 An Unambiguous Grammar for Expressions

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
       | <term>
<term> → <term> * <factor>
       | <factor>
<factor> → ( <expr> )
         | <id>
```

– The grammar in Example 4 generates the same language as the grammars of Examples 2 and 3, but it is unambiguous and it specifies the usual precedence order of multiplication and addition operators. The following derivation of the sentence `A = B + C * A` uses the grammar of Example 4:

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <expr> + <term>
         => A = <term> + <term>
         => A = <factor> + <term>
```

```
=> A = <id> + <term>
=> A = B + <term>
=> A = B + <term> * <factor>
=> A = B + <factor> * <factor>
=> A = B + <id> * <factor>
=> A = B + C * <factor>
=> A = B + C * <id>
=> A = B + C * A
```

– The unique parse tree for this sentence, using the grammar of Example 4, is shown in Figure 3.

– The connection between parse trees and derivations is very close: Either can easily be constructed from the other. Every derivation with an unambiguous grammar has a unique parse tree, although that tree can be represented by different derivations.
– For example, the following derivation of the sentence A = B + C * A is different from the derivation of the same sentence given previously.
– This is a rightmost derivation, whereas the previous one is leftmost. Both of these derivations, however, are represented by the same parse tree.

```
<assign> => <id> = <expr>
         => <id> = <expr> + <term>
         => <id> = <expr> + <term> * <factor>
         => <id> = <expr> + <term> * <id>
         => <id> = <expr> + <term> * A
         => <id> = <expr> + <factor> * A
         => <id> = <expr> + <id> * A
         => <id> = <expr> + C * A
         => <id> = <term> + C * A
         => <id> = <factor> + C * A
         => <id> = <id> + C * A
         => <id> = B + C * A
         => A = B + C * A
```

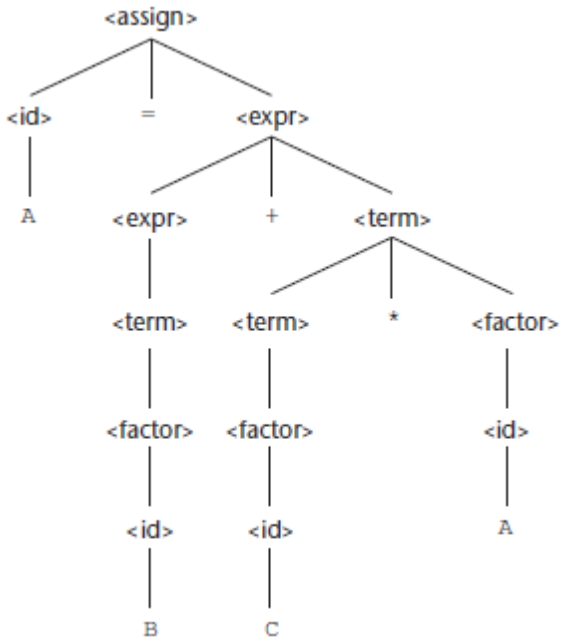

Figure 3 The unique parse tree for A = B + C * A using an unambiguous grammar

4 Associativity of Operators

– When an expression includes two operators that have the same precedence (as * and / usually have)—for example, A / B * C — a semantic rule is required to specify which should have precedence. This rule is named *associativity*.
– As was the case with precedence, a grammar for expressions may correctly imply operator associativity. Consider the following example of an assignment statement:

```
A = B + C + A
```

– The parse tree for this sentence, as defined with the grammar of Example 4, is shown in Figure 4.

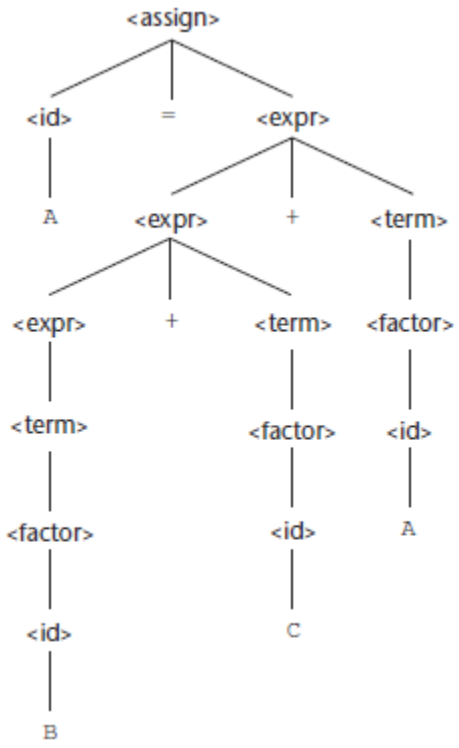

Figure 4 A parse tree for A = B + C + A illustrating the associativity of addition

– The parse tree of Figure 4 shows the left addition operator lower than the right addition operator.
– This is the correct order if addition is meant to be left associative, which is typical. In most cases, the associativity of addition in a computer is irrelevant. In mathematics, addition is associative, which means that left and right associative orders of evaluation mean the same thing.
– That is, (A + B) + C = A + (B + C). Floating-point addition in a computer, however, is not necessarily associative. For example, suppose floating-point values store seven digits of accuracy. Consider the problem of adding 11 numbers together, where one of the numbers is 10^7 and the other ten are 1.
– If the small numbers (the 1's) are each added to the large number, one at a time, there is no effect on that number, because the small numbers occur in the eighth digit of the large number.
– However, if the small numbers are first added together and the result is added to the large number, the result in seven-digit accuracy is $1.000001 * 10^7$. Subtraction and division are not associative, whether in mathematics or in a computer. Therefore, correct associativity may be essential for an expression that contains either of them.
– When a grammar rule has its LHS also appearing at the beginning of its RHS, the rule is said to be **left recursive**. This left recursion specifies left associativity.

– For example, the left recursion of the rules of the grammar of Example 4 causes it to make both addition and multiplication left associative. Unfortunately, left recursion disallows the use of some important syntax analysis algorithms.
– When one of these algorithms is to be used, the grammar must be modified to remove the left recursion.
– This, in turn, disallows the grammar from precisely specifying that certain operators are left associative. Fortunately, left associativity can be enforced by the compiler, even though the grammar does not dictate it.
– In most languages that provide it, the exponentiation operator is right associative. To indicate right associativity, right recursion can be used. A grammar rule is **right recursive** if the LHS appears at the right end of the RHS. Rules such as

```
<factor> → <exp> ** <factor>
         |<exp>
<exp> → ( <expr> )
      |id
```

could be used to describe exponentiation as a right-associative operator.

5 An Unambiguous Grammar for `if - else`

– The BNF rules for a Java if-else construct are as follows:

```
<if_stmt> → if (<logic_expr>) <stmt>
            if (<logic_expr>) <stmt> else <stmt>
```

– If we also have <stmt> S <if_stmt>, this grammar is ambiguous. The simplest sentential form that illustrates this ambiguity is

```
if (<logic_expr>) if (<logic_expr>) <stmt> else <stmt>
```

– The two parse trees in Figure 5 show the ambiguity of this sentential form. Consider the following example of this construct:

```
if (done == true)
if (denom == 0)
    quotient = 0;
    else quotient = num / denom;
```

– The problem is that if the upper parse tree in Figure 5 is used as the basis for translation, the else clause would be executed when done is not true, which probably is not what was intended by the author of the construct.
– We will now develop an unambiguous grammar that describes this **if** construct.
– The rule for if constructs in many languages is that an else clause, when present, is matched with the nearest previous unmatched then clause.
– Therefore, there cannot be an **if** construct without an **else** between a then clause and its matching **else**. So, for this situation, statements must be distinguished between those that are matched and those that are unmatched, where unmatched statements are **else**-less **if**s and all other statements are matched.

– The problem with the earlier grammar is that it treats all statements as if they had equal syntactic significance - that is, as if they were all matched.
– To reflect the different categories of statements, different abstractions, or nonterminals, must be used. The unambiguous grammar based on these ideas follows:

```
<stmt> → <matched> | <unmatched>
<matched> → if (<logic_expr>) <matched> else <matched>
          |any non-if statement
<unmatched> → if (<logic_expr>) <stmt>
            |if (<logic_expr>) <matched> else <unmatched>
```

– There is just one possible parse tree, using this grammar, for the following sentential form:

```
if (<logic_expr>) if (<logic_expr>) <stmt> else <stmt>
```

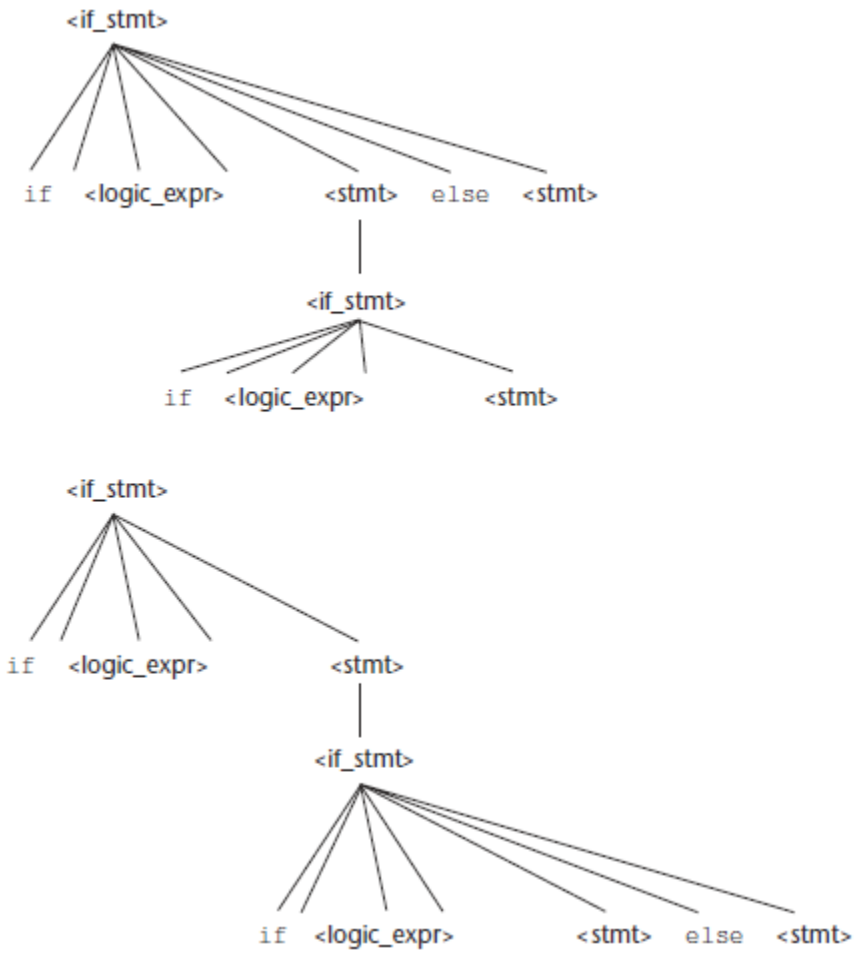

Figure 5 Two distinct parse trees for the same sentential form