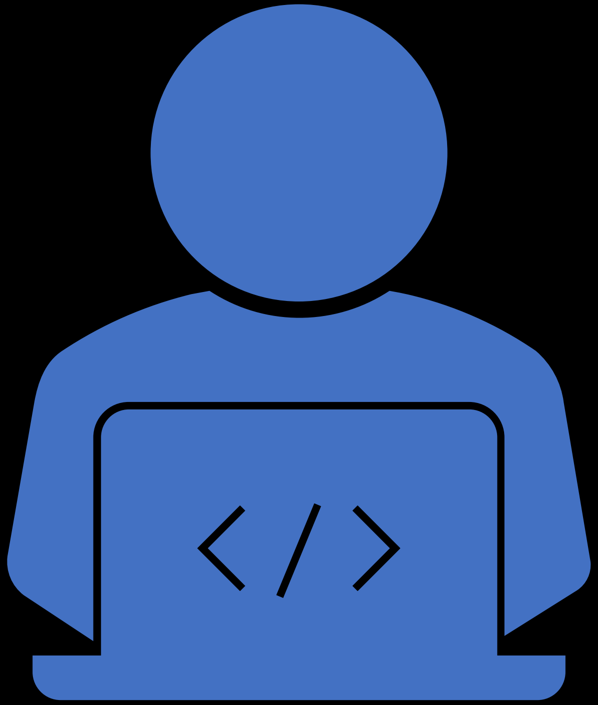


CENG204 - Programming Languages Concepts

Asst. Prof. Dr. Emre ŞATIR

Lecture 7

Describing Syntax and Semantics

Lecture 7 Topics

- Introduction
- The General Problem of Describing Syntax
  - Terminology, Recognizers, Generators
- Formal Methods of Describing Syntax
  - Backus-Naur Form (BNF)
  - Grammars and Derivations
  - Parse Trees
  - Ambiguity / Operator Precedence / Associativity of Operators / Ambiguity Issues with if-else
  - Extended BNF
- Attribute Grammars
- Describing the Meanings of Programs: Dynamic Semantics

Introduction

Introduction

- The study of programming languages, like the study of natural languages, can be divided into examinations of **syntax** and **semantics**.
  - **Syntax**: the form or structure of the expressions, statements, and program units.
  - **Semantics:** the meaning of those expressions, statements, and program units.
- Syntax and semantics provide a language's **definition**.

Introduction

- For example, the **syntax** of a Java while statement is

```
while (boolean_expr) statement
```

- The **semantics** of this statement form is that "when the current value of the boolean expression is <u>true</u>, the embedded statement is <u>executed</u>. Then control implicitly returns to the boolean expression to repeat the process. If the boolean expression is <u>false</u>, control transfers to the statement <u>following the while construct</u>".

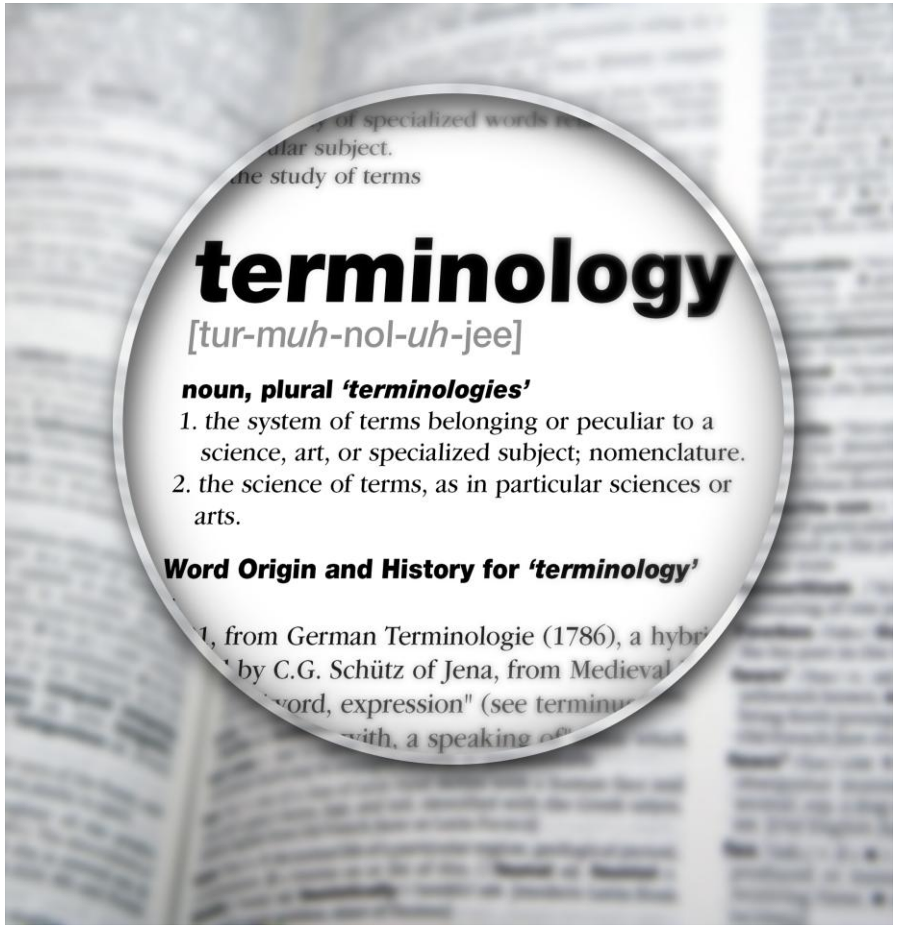
Introduction

- Although they are often separated for discussion purposes, syntax and semantics are closely related.
- In a well-designed programming language, semantics should follow directly from syntax; that is, the appearance of a statement should strongly suggest what the statement is meant to accomplish.
- “Describing syntax” is easier than “describing semantics”, partly because a concise and universally accepted notation is available for syntax description, but none has yet been developed for semantics.

The General Problem of Describing Syntax

Terminology

- A **language**, whether natural (such as Turkish or English) or artificial (such as C or Java), is a set of strings of characters from some alphabet.
- The strings of a language are called **sentences** or **statements**.
- So,
- A **sentence** (or a **statement**) is a string of characters over some alphabet.
- A **language** is a set of **sentences** (or **statements**).



Terminology

- The **syntax rules** of a language specify which strings of characters from the language's alphabet are in the language.
- English, for example, has a large and complex collection of rules for specifying the syntax of its sentences.
- By comparison, even the largest and most complex programming languages are syntactically very simple according to a natural language.

Terminology

- A **lexeme** is the "lowest level syntactic unit" of a language.
  - The lexemes of a programming language include its numeric literals, operators, and special words, among others (e.g., `*`, `sum`, `begin`). One can think of programs as strings of lexemes rather than of characters.
- Lexemes are partitioned into groups—for example, the names of variables, methods, classes, and so forth in a programming language form a group called "**identifiers**".
- Each lexeme group is represented by a name, or **token**. So, a **token** of a language is a category of its lexemes.

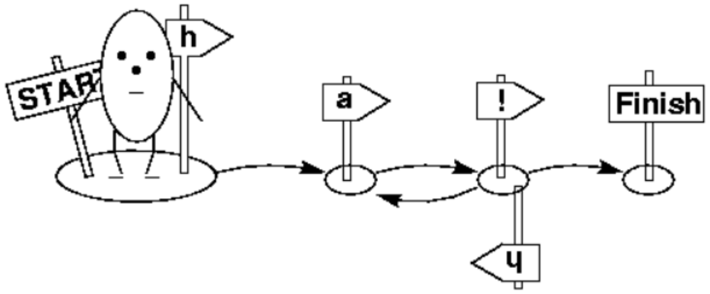
Terminology - Example

Consider the following Java statement:

```
index = 2 * count + 17;
```

Lexemes	**Tokens**
`index`	identifier
`=`	equal_sign
`2`	int_literal
`*`	mult_op
`count`	identifier
`+`	plus_op
`17`	int_literal
`;`	semicolon

Recognizers



- In general, languages can be formally defined in two distinct ways: by **recognition** and by **generation**.
- A **recognition device** (a **language recognizer**) reads input strings over the alphabet of the language and decides whether the input strings belong to the language.
- The "syntax analysis part" of a compiler is a recognizer for the language the compiler translates. In this role, the recognizer need to determine whether given programs are in the language.
- In effect then, the syntax analyzer determines whether the given programs are syntactically correct. **Syntax analyzers** also known as "**parsers**".

Language Generators

Generators

- A **language generator** is a device that can be used to generate the sentences of a language.
- To determine the correct syntax of a particular statement using a compiler, the programmer can submit it to compiler and note whether the compiler accepts it.
- On the other hand, it is often possible to determine whether the syntax of a particular statement is correct by comparing it with the structure of the generator.

An Example of Token Checking

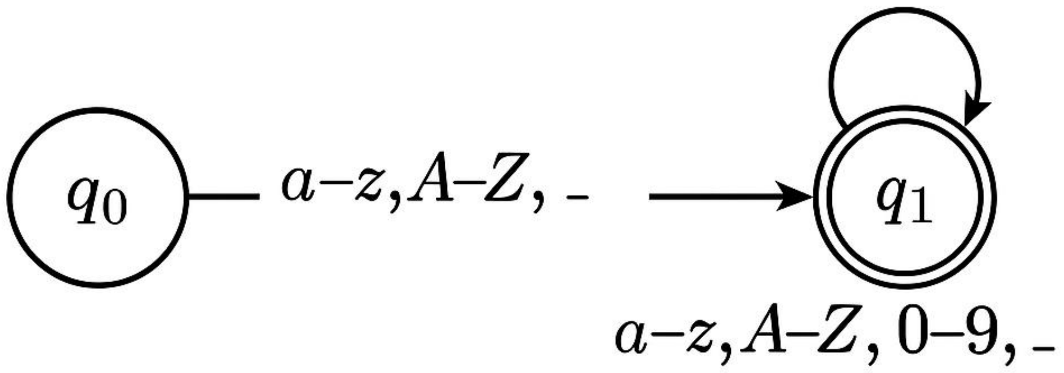
- In a programming language, <u>variable names</u> are usually subject to specific rules. For example, a variable name should follow these rules:
  - It may contain only letters (a–z, A–Z), digits (0–9), and the underscore (_).
  - The first character must be a letter or an underscore.
  - It cannot start with a digit.

- The following “regular expression” represents the rules mentioned above:

  `[a-zA-Z_][a-zA-Z0-9_]*`

  - `[a-zA-Z_]`: The first character must be a letter or an underscore.
  - `[a-zA-Z0-9_]*`: The rest can contain zero or more letters, digits, or underscores.

An Example of Token Checking



Recognizers and Generators

- People prefer certain forms of generators over recognizers because they can more easily read and understand them.
- The syntax-checking portion of a compiler (a language recognizer / parser) is not as useful a language description for a programmer because it can be used only in trial-and-error mode.
- There is a close connection between formal generation and recognition devices for the same language.
- This was one of the seminal discoveries in computer science, and it led to much of what is now known about formal languages and compiler design theory.

Formal Methods of Describing Syntax

Formal Methods of Describing Syntax

- This section discusses the formal language-generation mechanisms, usually called **grammars**, that are commonly used to describe the syntax of programming languages.
- In the middle to late 1950s, two men, Noam Chomsky and John Backus, in unrelated research efforts, developed the same syntax description formalism (CFG and BNF), which subsequently became the most widely used method for programming language syntax.

Context-Free Grammars (CFGs)

- In the mid-1950s, Noam Chomsky, a noted linguist (among other things), described four classes of generative devices or grammars that define four classes of languages (natural languages).
- Two of these grammar classes, named “**regular**” and “**context-free**”, turned out to be useful for describing the syntax of programming languages.
- The forms of the tokens of programming languages can be described by regular grammars (Regular Expressions).
- The syntax of whole programming languages can be described by context-free grammars.

Grammars

Noam Chomsky gave a Mathematical model of Grammar which is effective for writing computer languages

The four types of Grammar according to Noam Chomsky are:

Type	Language Class	Automaton	Grammar Type	Example
Type-3	Regular Languages	Finite Automata	Regular Grammar	a^*b^*
Type-2	Context-Free Languages (CFL)	Pushdown Automata	Context-Free Grammar (CFG)	a^nb^n
Type-1				
Type-0				

Backus-Naur Form (BNF)

- 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 in 1959.
- This paper introduced a new formal notation for specifying programming language syntax.
- The new notation was later modified slightly by **Peter Naur** for the description of ALGOL 60.
- This revised method of syntax description became known as **Backus-Naur Form**, or simply **BNF**.

Backus-Naur Form (BNF)

- BNF is a notation for describing syntax.
- Although the use of BNF in the ALGOL 60 report was not immediately accepted by computer users, it soon became and is still the most popular method of concisely describing programming language syntax.
- It is remarkable that BNF is nearly identical to Chomsky's generative devices for context-free languages, called context-free grammars (CFGs) (i.e., BNF is equivalent to context-free grammars).
- In the remainder of the chapter, we refer to context-free grammars simply as grammars. Furthermore, the terms BNF and grammar are used interchangeably.

BNF Fundamentals

- A **metalanguage** is a language that is used to describe another language. BNF is a “metalanguage for programming languages”.
- BNF uses abstractions for syntactic structures. A simple Java assignment statement, for example, might be represented by the abstraction <assign> (pointed brackets are often used to delimit names of abstractions). The actual definition of <assign> can be given by

```
<assign> → <var> = <expression>
```

- The text on the left side of the arrow, which is aptly called the left-hand side (LHS), is the abstraction being defined. (which is always a nonterminal and act like syntactic variables).
- The text to the right of the arrow is the definition of the LHS. It is called the right-hand side (RHS) and consists of some mixture of tokens, lexemes, and references to other abstractions.
- Altogether, the definition is called a **rule** or **production**.

BNF Fundamentals

- This particular rule specifies that the abstraction <assign> is defined as an instance of the abstraction "<var>", followed by the lexeme "=", and followed by an instance of the abstraction "<expression>".
- One example sentence whose syntactic structure is described by the rule might be:

```
total = subtotal1 + subtotal2 (<var> = <expression>)
```

- The abstractions in a BNF description, or grammar, are often called nonterminal symbols, or simply **nonterminals**, and the lexemes and tokens of the rules are called terminal symbols, or simply **terminals**.
- A BNF description, or grammar, is a collection of rules.

A Grammar for Simple Assignment Statements

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

For example, the statement

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

is generated by the 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)
```

BNF Fundamentals

- Example for “if” statements:

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

- Describing Lists:

```
<ident_list> → identifier
             | identifier, <ident_list>
```

- A rule is recursive if its LHS appears in its RHS.
- This example defines <ident_list> as either a single token (identifier) or an identifier followed by a comma and another instance of <ident_list>.

Grammars and Derivations

- A **grammar** is a generative device for defining languages. (It is a finite non-empty set of rules).
- 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. A **derivation** is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols).
- 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 below is used to illustrate derivations:

A Grammar for a Small Language

```
<program> → <stmts>
  <stmts> → <stmt> | <stmt> ; <stmts>
   <stmt> → <var> = <expr>
   <var>  → a | b | c | d
   <expr> → <term> + <term> | <term> - <term>
   <term> → <var> | const
```

A Derivation of a Program in Example Language Follows

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

Another Example Grammar for a 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>

A Derivation of a Program in Example 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 + C ; <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
```

Derivations

- Every string of symbols in a derivation is a **sentential form** (Each of the strings in the derivation, including <program>, is called a sentential form).
- A **sentence** is a sentential form that has only terminal symbols.

  `a = b + const`

  and

  `begin` `A = B + C ; B = C` **`end`**

  are sentences for the previous examples.

Parse Trees

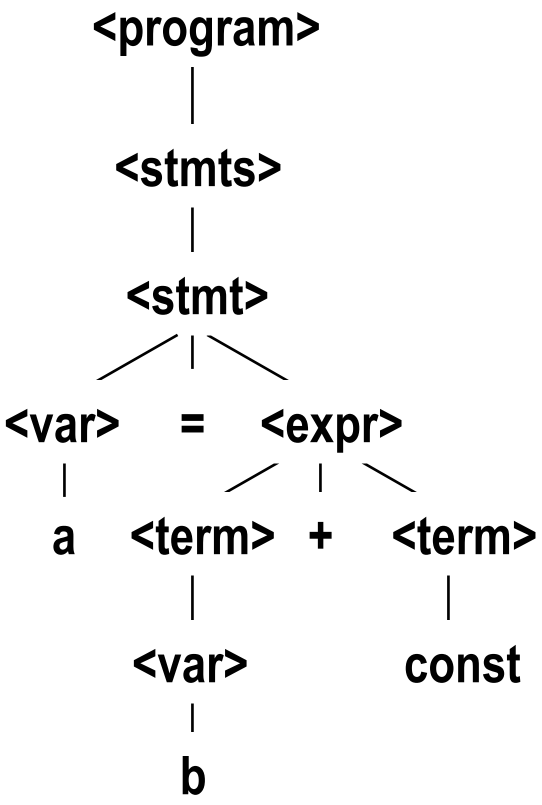
- One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the languages they define.
- These hierarchical structures are called **parse trees**.
- Every internal node of a parse tree is labeled with a nonterminal symbol; every leaf is labeled with a terminal symbol.

Parse Trees – Example

(a = b + const)

```
<program> → <stmts>
  <stmts> → <stmt> | <stmt> ; <stmts>
   <stmt> → <var> = <expr>
   <var>  → a | b | c | d
   <expr> → <term> + <term> | <term> - <term>
   <term> → <var> | const
```

```
<program> => <stmts>
          => <stmt>
          => <var> = <expr>
          => a = <expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

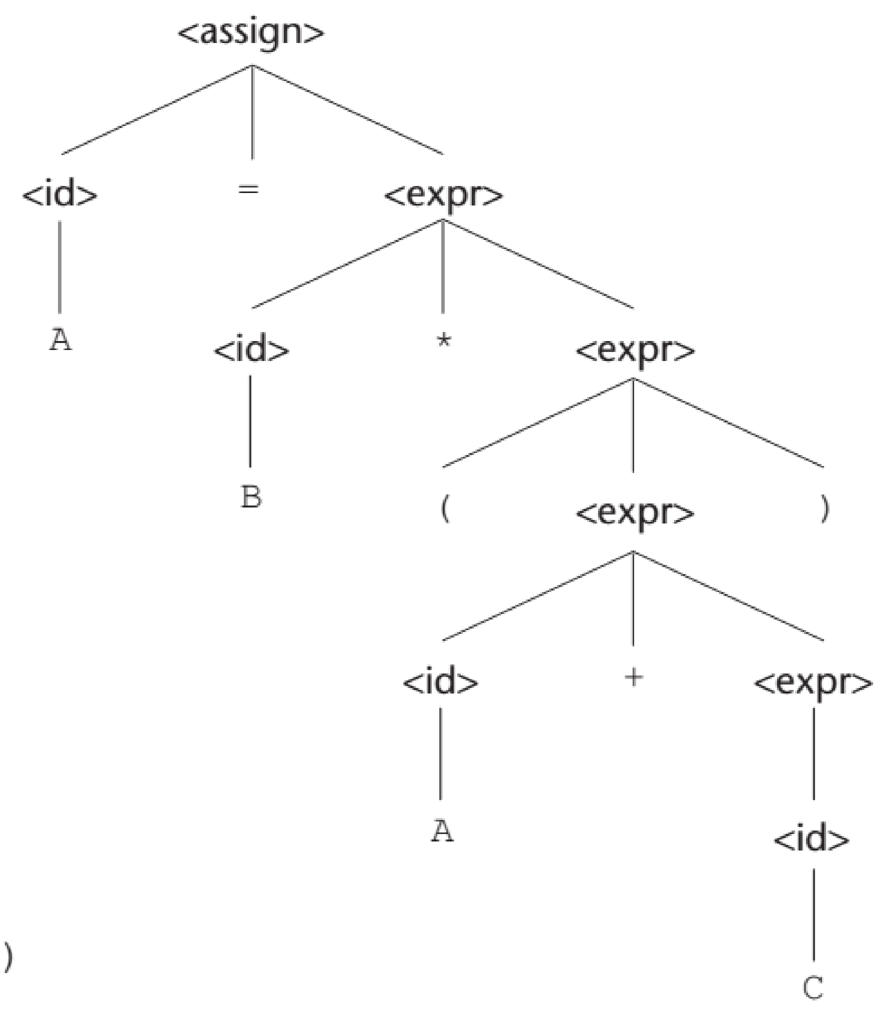


Parse Trees – Another Example

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

<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)

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



Ambiguity

- A grammar that generates a sentence (statement) for which there are two or more distinct parse trees is said to be **ambiguous** (If a grammar generates the same string in several different ways).
- Consider the grammar shown below which is a minor variation of the grammar shown before.

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

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

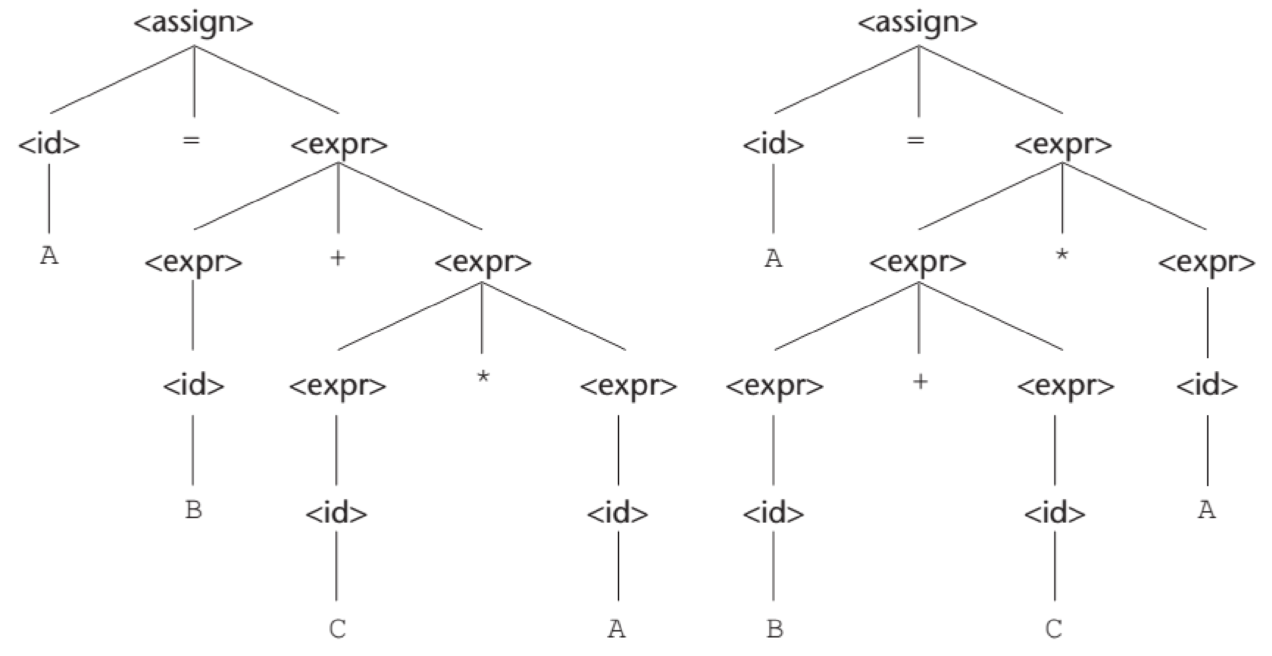
Ambiguity

- The grammar above (on the right) is <u>ambiguous</u> because the sentence

  `A = B + C * A`

has two distinct parse trees, as shown in the figure.

- *** Rather than allowing the parse tree of an expression to grow <u>only on the right</u>, this grammar allows growth on <u>both the left and the right</u>.



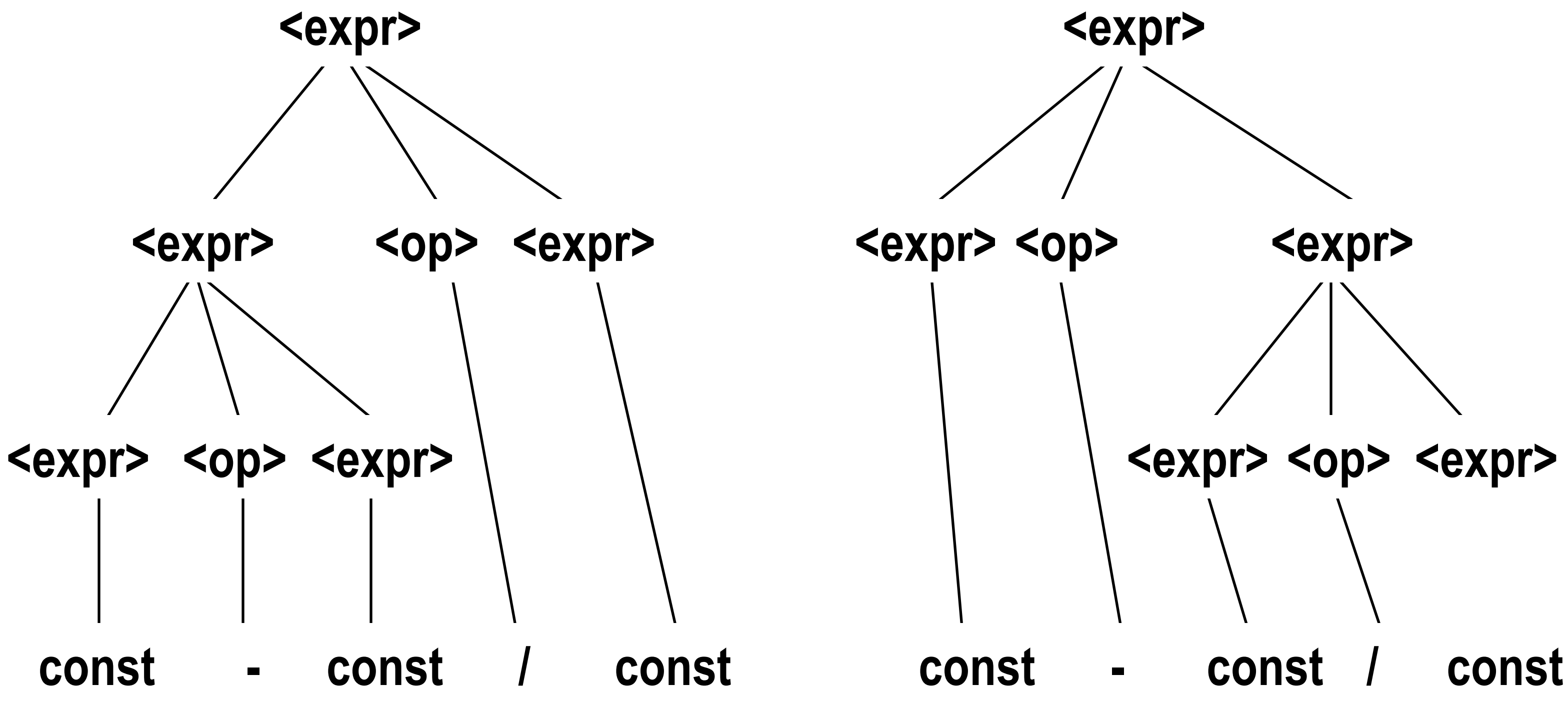
Ambiguity

- Syntactic ambiguity of language structures is a problem because compilers often base the semantics of those structures on their syntactic form.
- If a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely.
- In many cases, an ambiguous grammar can be rewritten to be unambiguous but still generate the desired language.

- *** Note that it is mathematically impossible to determine whether an arbitrary grammar is ambiguous.

Another Ambiguous Grammar Example

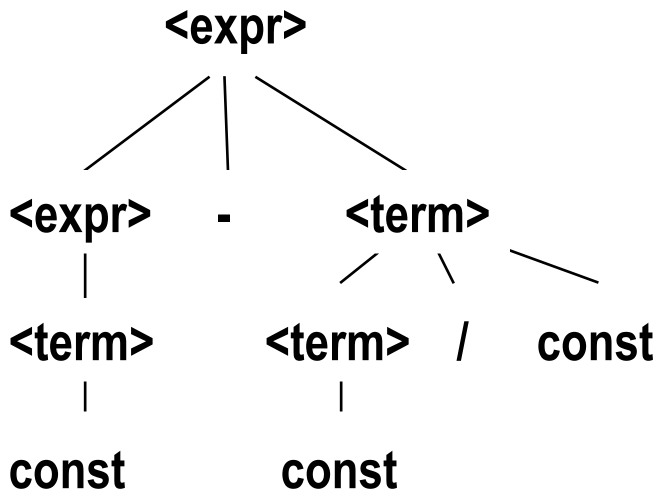
```
<expr> → <expr> <op> <expr>  |  const
<op> → /  |  -
```



Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity.

```
<expr> → <expr> - <term>  | <term>
<term> → <term> / const   | const
```



Ambiguity

- We’ve said that if a language structure <u>has more than one parse tree</u>, then the <u>meaning</u> of the structure <u>cannot be determined uniquely</u>.
- This problem is discussed in <u>two specific examples</u> in the following subsections:
  - Operator Precedence
  - Associativity of Operators

Operator Precedence

- 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.

Operator's Precedence in Java

Operators	Precedence
!, +, - (unary Operators)	First (Highest)
* , / , %	Second
+ , -	Third
< , <= , >=, >	Fourth
== , !=	Fifth
&&	Sixth
||	Seventh
= (assignment Operator)	Lowest

Operator Precedence

An Unambiguous Grammar for Expressions

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

- In this example grammar, * will always be lower in the parse tree, simply because it is farther from the start symbol than + in every derivation.

Operator Precedence

- The grammar 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 this grammar.

```
<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
```

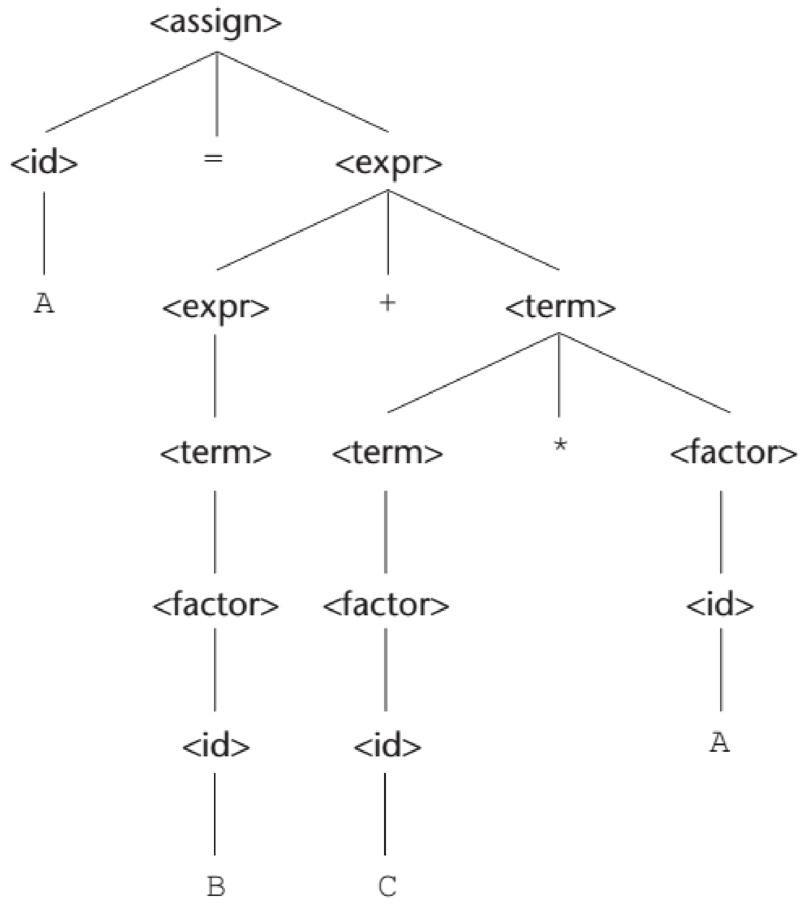
Operator Precedence

- Note that, every derivation with an unambiguous grammar has a unique parse tree, although that tree can be represented by different derivations.
- For example, we can give different derivations of the sentence

  A = B + C * A

- from given grammar. But, all of these derivations, however, are represented by the same parse tree.

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



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**.
- An expression with two occurrences of the same operator has the same issue; for example, A * B * C.
- 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
```

Associativity of Operators

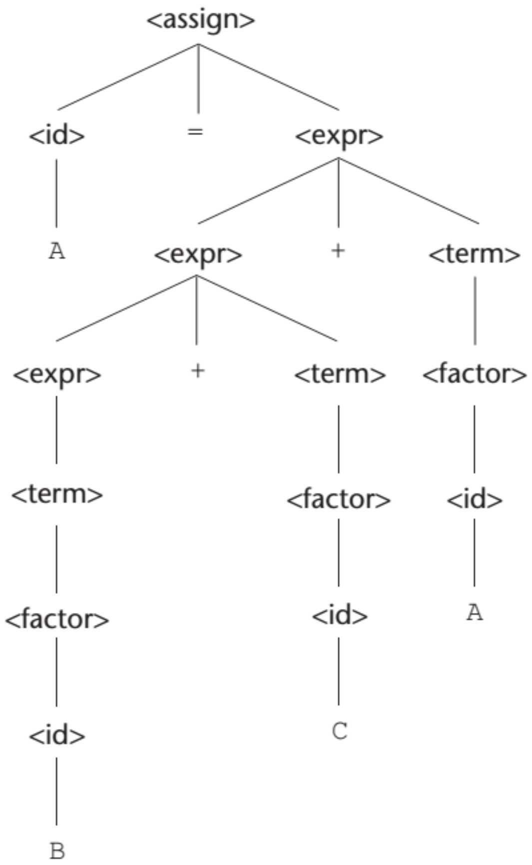
The parse tree for this sentence, as defined with the given example before, is shown below.

The parse tree 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.

An Unambiguous Grammar for Expressions

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

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



Associativity of Operators

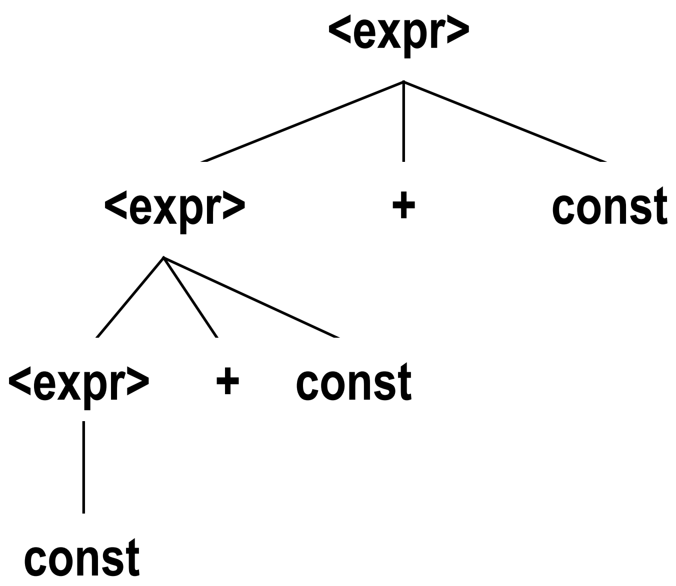
- In mathematics, addition and multiplication is associative, which means that left and right associative orders of evaluation mean the same thing. That is,

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

- Floating-point addition and multiplication in a computer, however, is not necessarily associative (because of digits of accuracy).
- 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.

Associativity of Operators - Example

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Ambiguity Issues with **`if-else`**

- The BNF rules for a Java **if-else** statement are as follows:

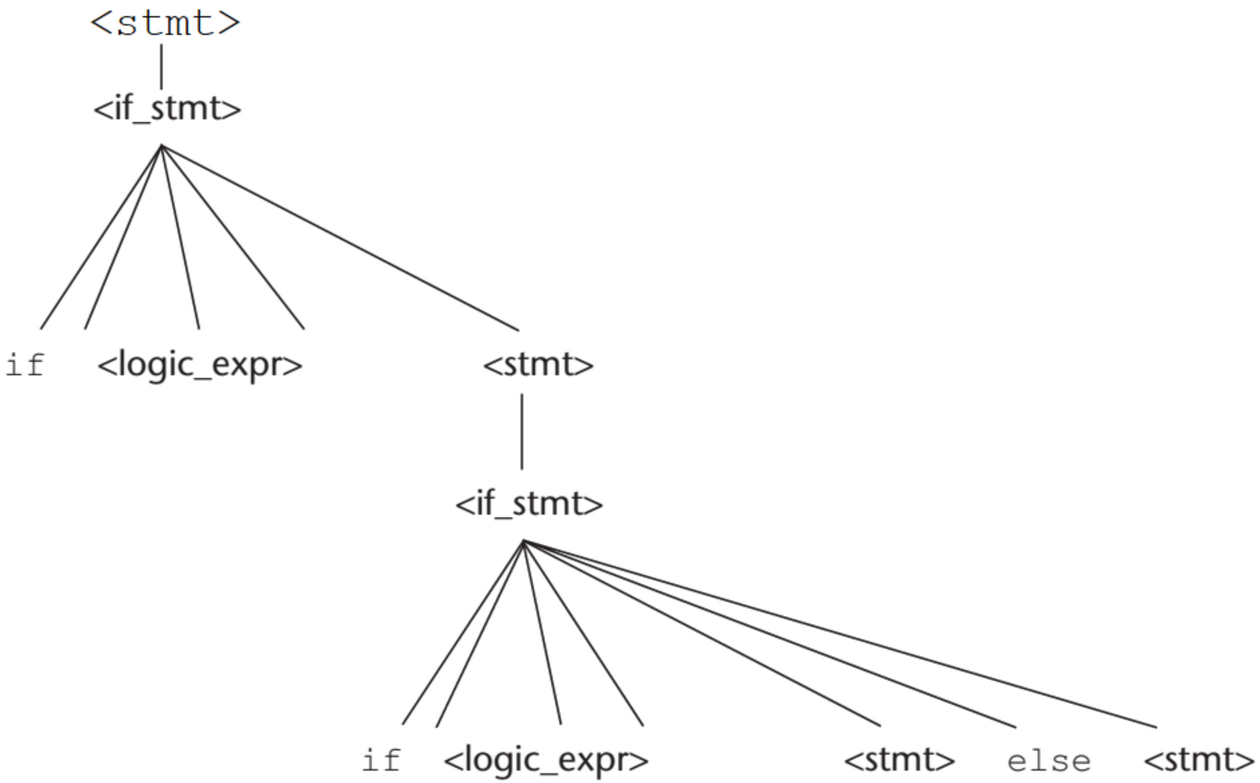
```
<stmt> → <if_stmt>
<if_stmt> → if (<logic_expr>) <stmt>
          | if (<logic_expr>) <stmt> else <stmt>
```

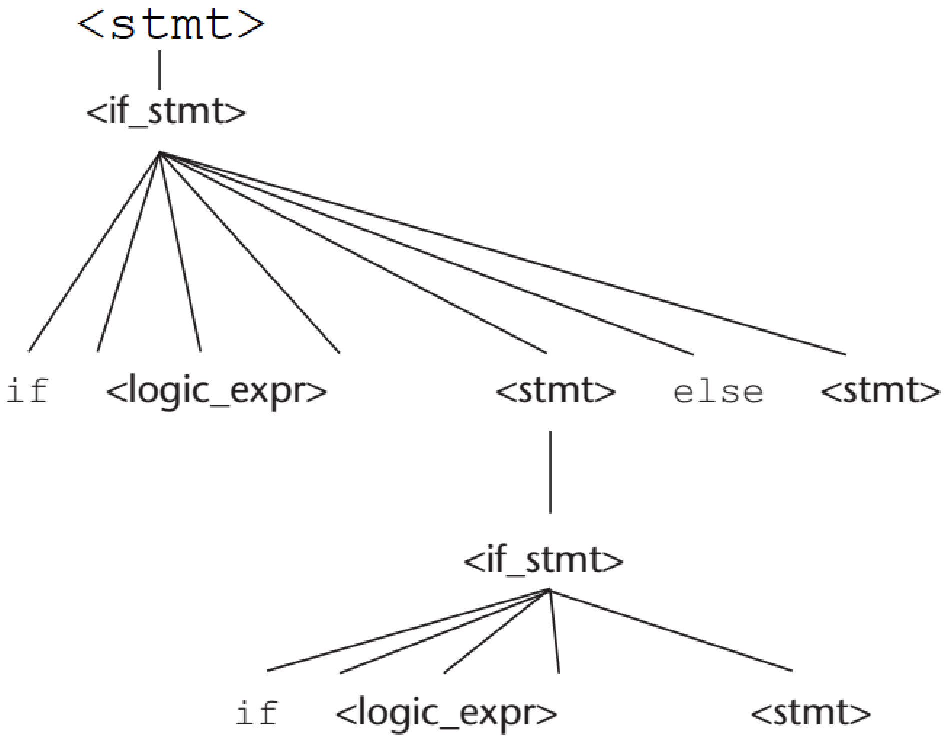
- This grammar is ambiguous. The simplest sentential form that illustrates this ambiguity is

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

- The two parse trees in figure below show the ambiguity of this sentential form.

Two distinct parse trees for the same sentential form





Ambiguity Issues with **`if-else`**

- Consider the following example of this construct ("**Dangling else problem**"):

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

- The problem is that if the right parse tree in figure above 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.

Ambiguity Issues with **`if-else`**

- We will now develop an <u>unambiguous</u> grammar that describes this if statement.
- The rule for if constructs in many languages is that an else clause, when present, is matched with the <u>nearest previous unmatched</u> if clause.
- The problem with the earlier grammar is that it treats all statements as if they had equal syntactic significance.
- To reflect the different categories of statements, <u>different abstractions or nonterminals, must be used</u>.

An Unambiguous Grammar for **`if-else`**

- The unambiguous grammar based on these ideas follows:

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

- There is <u>just one</u> possible parse tree, using this grammar, for the following sentential form:

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

Extended BNF

- Because of a few minor inconveniences in BNF, it has been extended in several ways.
- Most extended versions of BNF are called **Extended BNF**, or simply **EBNF**, even though they are not all exactly the same.
- The extensions do not enhance the descriptive power of BNF; they only increase its readability and writability.

Extended BNF

- Optional parts are placed in brackets [ ]

```
<proc_call> -> ident [(<expr_list>)]
```

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

```
<term> → <term> (+|-) const
```

- Repetitions (0 or more) are placed inside braces { }

```
<ident> → letter {letter|digit}
```

EXAMPLE

BNF and EBNF Versions of an Expression Grammar

BNF:

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
<factor> → <exp> ** <factor>
           <exp>
<exp> → (<expr>)
      | id
```

EBNF:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
      | id
```

Grammars and Recognizers

- Earlier in this chapter, we suggested that there is a close relationship between generation and recognition devices for a given language.
- In fact, given a context-free grammar, a recognizer for the language generated by the grammar can be algorithmically constructed.
- A number of software systems have been developed that perform this construction.
- Such systems allow the quick creation of the syntax analysis part of a compiler for a new language and are therefore quite valuable.

Yet Another Compiler-Compiler

One of the first of these syntax analyzer generators is named **yacc** (**y**et **a**nother **c**ompiler **c**ompiler).

Yacc is a computer program for the Unix operating system developed by Stephen C. Johnson.

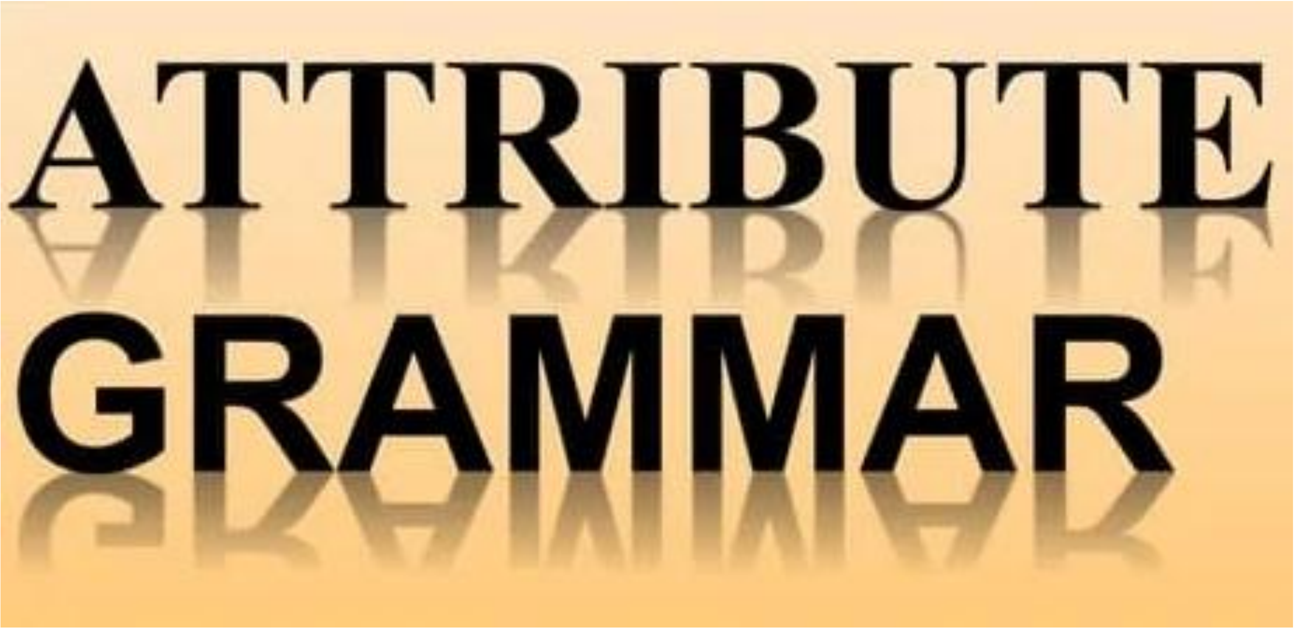
It is a parser generator, based on a formal grammar, written in a notation similar to Backus–Naur Form (BNF).

There are now many such systems available.

Attribute Grammars

Attribute Grammars

- An **attribute grammar** is a device used to describe more of the structure of a programming language than can be described with a context-free grammar.
- An attribute grammar is an extension to a context-free grammar.
- The extension allows certain language rules to be conveniently described, such as type compatibility.



Attribute Grammars

- There are some characteristics of programming languages that are difficult to describe with BNF, and some that are impossible.
- As an example of a syntax rule that is difficult to specify with BNF, consider type compatibility rules.
- In Java, for example, a floating-point value cannot be assigned to an integer type variable, although the opposite is legal.
- Although this restriction can be specified in BNF, it requires additional nonterminal symbols and rules.
- If all of the typing rules of Java were specified in BNF, the grammar would become too large to be useful, because the size of the grammar determines the size of the syntax analyzer.

Attribute Grammars

- As an example of a syntax rule that <u>cannot</u> be specified in BNF, consider the common rule that “all variables must be declared before they are referenced”.
- It has been proven that this rule <u>cannot</u> be specified in BNF.
- These problems exemplify the categories of language rules called **static semantics rules**.

Static Semantics

- The **static semantics** of a language is only indirectly related to the meaning of programs during execution; rather, it has to do with the legal forms of programs (syntax rather than semantics).
- Many static semantic rules of a language state its type constraints.
- Static semantics is so named because the analysis required to check these specifications can be done at compile time.
- Because of the problems of describing static semantics with BNF, a variety of more powerful mechanisms has been devised for that task.
- One such mechanism, **attribute grammars**, was designed by Donald Knuth to describe both the syntax and the static semantics of programs.

Attribute Grammars

- **Attribute grammars (AGs)** are a formal approach both to describing and checking the correctness of the static semantics rules of a program (i.e., AG is a descriptive formalism that can describe both the syntax and static semantics of a language).
- Although they are not always used in a formal way in compiler design, the basic concepts of attribute grammars are at least informally used in every compiler.
- Attribute grammars have additions to CFGs to carry some semantic info on parse tree nodes.
- Actually, an attribute grammar is a context-free grammar with some additions to define some semantic information.

An Attribute Grammar for Simple Assignment Statements

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

Describing the Meanings of Programs: Dynamic Semantics

Dynamic Semantics

- We now turn to the difficult task of describing the **dynamic semantics**, or "**meaning**", of the expressions, statements, and program units of a programming language.
- Because of the power and naturalness of the available notation, describing syntax is a relatively simple matter.
- On the other hand, no universally accepted notation or approach has been devised for dynamic semantics.
- There are several methods that have been developed.
- For the remainder of this section, when we use the term **semantics**, we mean dynamic semantics.

Semantics

- There are several different reasons underlying the need for a methodology and notation for describing semantics.
- Programmers obviously need to know precisely what the statements of a language do before they can use them effectively in their programs. (Programmers need to know what statements mean)
- Compiler writers must know exactly what language constructs mean to design implementations for them correctly. (Compiler writers must know exactly what language constructs do).
- However, software developers and compiler designers typically determine the semantics of programming languages by reading English explanations in language manuals.

Example Revisited

- For example, the **syntax** of a Java while statement is

```
while (boolean_expr) statement
```

- The **semantics** of this statement form is that "when the current value of the boolean expression is <u>true</u>, the embedded statement is <u>executed</u>. Then control implicitly returns to the boolean expression to repeat the process. If the boolean expression is <u>false</u>, control transfers to the statement <u>following the while construct</u>".

Semantics

- Some approaches that are suitable for imperative languages for describing formal semantics.
  - Operational Semantics
  - Denotational Semantics
  - Axiomatic Semantics