

# Organisation of Programming Languages CMP 401 Describing Syntax

Egena Onu, PhD.  
Computer Science Department,  
Bingham University.

# Problem of Describing Syntax

- ▶ Whether it is natural or not, a language is only a set of strings of characters from some alphabets.
- ▶ Strings of a language are combined to form sentences or statements.
- ▶ The syntax rules of a language specify which characters from the language's alphabet are in the language.
- ▶ For example, English 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 simple.



# Problem of Describing Syntax

- The formal description of the syntax of programming languages often do not include descriptions of the lowest-level syntactic units.
- These syntactic units are called **lexemes**.
- The **lexemes** of a programming language include its numeric literals, operators, and special words, among others.
- The description of lexemes can be given by a lexical specification, usually separate from the syntactic description of the language.



# Problem of Describing Syntax

- ▶ Think of programs as strings of lexemes rather than of characters.
- ▶ **Lexemes** are partitioned into groups such as:
  - ▶ variable names
  - ▶ Methods (functions)
  - ▶ Etc.
- ▶ These groups are called identifiers.
- ▶ Each group is represented by a name or **token**.



# Problem of Describing Syntax

- A **token** of a language is a category of its lexemes.
- For example, an identifier is a **token** that can have lexemes, or instances such as sum or total.
- In some cases, a token has only a single possible lexeme.

# Problem of Describing Syntax

- For example, consider the simple statement:

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

- This statement can be described in the following form:

Lexeme	token
Index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon



# Language Recognisers and Generators

- ▶ Languages are formally defined in two distinct ways:
  - ▶ Language Recognition
  - ▶ Language Generation
- ▶ Language Recogniser:
  - ▶ A recogniser is a device that reads the input strings of a language and decides whether the input strings belong to the language.
  - ▶ The recogniser only determines whether given programs are in the language.
  - ▶ For example, the syntax analyser part of a compiler is a recognition device.
  - ▶ The syntax analyser is also known as a parser.





# Language Recognisers and Generators

- ▶ Language Generator:
  - ▶ Language generator is a device that generates sentences of a language.
  - ▶ One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator.



# Formal Methods of Describing Syntax

- ▶ The formal language generation mechanisms are usually called **grammars**.
- ▶ **Grammars** are commonly used to describe the syntax of programming languages.
- ▶ Over the years, two methods of syntax description became the most widely used. These include:
  - ▶ Backus-Naur Form
  - ▶ Context Free Grammar

# Backus-Naur Form (BNF)

- Invented in 1959 by John Backus to describe ALGOL 58 syntax.
- Fundamentals:
  - **Metalinguage:**
    - A **metalanguage** is a language used to describe another language.
    - BNF is a metalanguage for programming languages.
    - In **BNF**, abstractions are used to represent classes of syntactic structures.
    - That is, they act like syntactic variables also called the **nonterminal symbols**.

# Backus-Naur Form (BNF)

## ► Fundamentals:

- For example, a simple assignment statement can be represented by the abstraction **<assign>**.

- The definition of **<assign>** is given as:

**<assign>** → **<var>** = **<expression>**

- This definition is a rule that describes the structure of an assignment statement.
- A rule has a left hand side (LHS) and the right hand side (RHS).
  - **LHS** is the abstraction being defined.
  - **RHS** consists of mixtures of tokens, lexemes and references to other abstractions.

# Backus-Naur Form (BNF)

- The abstractions in a BNF description, or grammar, are often called **nonterminal symbols** or simply, nonterminals.
- The lexemes and tokens of the rules are called **terminal symbols** or simply, terminals.
- Keep in mind that the BNF description, or grammar is simply a collection of rules.
- Nonterminal symbols can have two or more distinct definitions, representing two or more possible syntactic forms in the language.
- Multiple definitions can be written as a single rule, with the different definitions separated by the symbol | , meaning logical OR.

# Backus-Naur Form (BNF)

- For example a simple **if** statement can be described with the rules:

$\langle \text{if\_stmt} \rangle \rightarrow \text{if ( } \langle \text{logic\_expr} \rangle \text{ ) } \langle \text{stmt} \rangle$

$\langle \text{if\_stmt} \rangle \rightarrow \text{if ( } \langle \text{logic\_expr} \rangle \text{ ) } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

- or with the rule:


$\langle \text{if\_stmt} \rangle \rightarrow \text{if ( } \langle \text{logic\_expr} \rangle \text{ ) } \langle \text{stmt} \rangle$

$\quad | \text{ if ( } \langle \text{logic\_expr} \rangle \text{ ) } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

- In these rules,  $\langle \text{stmt} \rangle$  represents either a single statement or a compound statement.



# Backus-Naur Form (BNF)

- Although BNF is simple, it is sufficiently powerful to describe nearly all of the syntax of programming languages.
  - In particular, it can describe lists of similar constructs, the order in which different constructs must appear, and nested structures to any depth, and even imply operator precedence and operator associativity.
- 



# Backus-Naur Form (BNF)

## Describing List:

- variable-length lists in mathematics are often written using an ellipsis (. . .); something like: 1, 2, . . .
- BNF does not include the ellipsis, so an alternative method is required for describing lists of syntactic elements in programming languages (for example, a list of identifiers appearing on a data declaration statement).
- For BNF, the alternative is recursion.
- A rule is **recursive** if its LHS appears in its RHS. The following rules illustrate how recursion is used to describe lists:

$$\begin{aligned} \langle \text{ident\_list} \rangle &\rightarrow \text{identifier} \\ &\mid \text{identifier}, \langle \text{ident\_list} \rangle \end{aligned}$$

- This defines  $\langle \text{ident\_list} \rangle$  as either a single token (identifier) or an identifier followed by a comma and another instance of  $\langle \text{ident\_list} \rangle$ .
- Recursion is used to describe lists in many of the example grammars.

# Grammar and Derivation

- 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 the following listing is used to illustrate derivations.

# Grammar and Derivation

- Listing 1: A grammar for a small language:

$\langle \text{program} \rangle \rightarrow \mathbf{begin} \langle \text{stmt\_list} \rangle \mathbf{end}$

$\langle \text{stmt\_list} \rangle \rightarrow \langle \text{stmt} \rangle$

$\quad | \langle \text{stmt} \rangle ; \langle \text{stmt\_list} \rangle$

$\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$

$\langle \text{var} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle + \langle \text{var} \rangle$

$\quad | \langle \text{var} \rangle - \langle \text{var} \rangle$

$\quad | \langle \text{var} \rangle$

- The language described by the grammar of Listing 1 has only one statement form: assignment.



# Grammar and Derivation

- 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.

# Grammar and Derivation

- A derivation of a program in a language can therefore be as in Listing 2

- Listing 2:

```
<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
```

# Grammar and Derivation

- This derivation, like all derivations, begins with the start symbol, in this case <program>.
- The symbol  $\Rightarrow$  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**.



# Grammar and Derivation

- 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.



# Grammar and Derivation

- As a final example, consider the statement

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

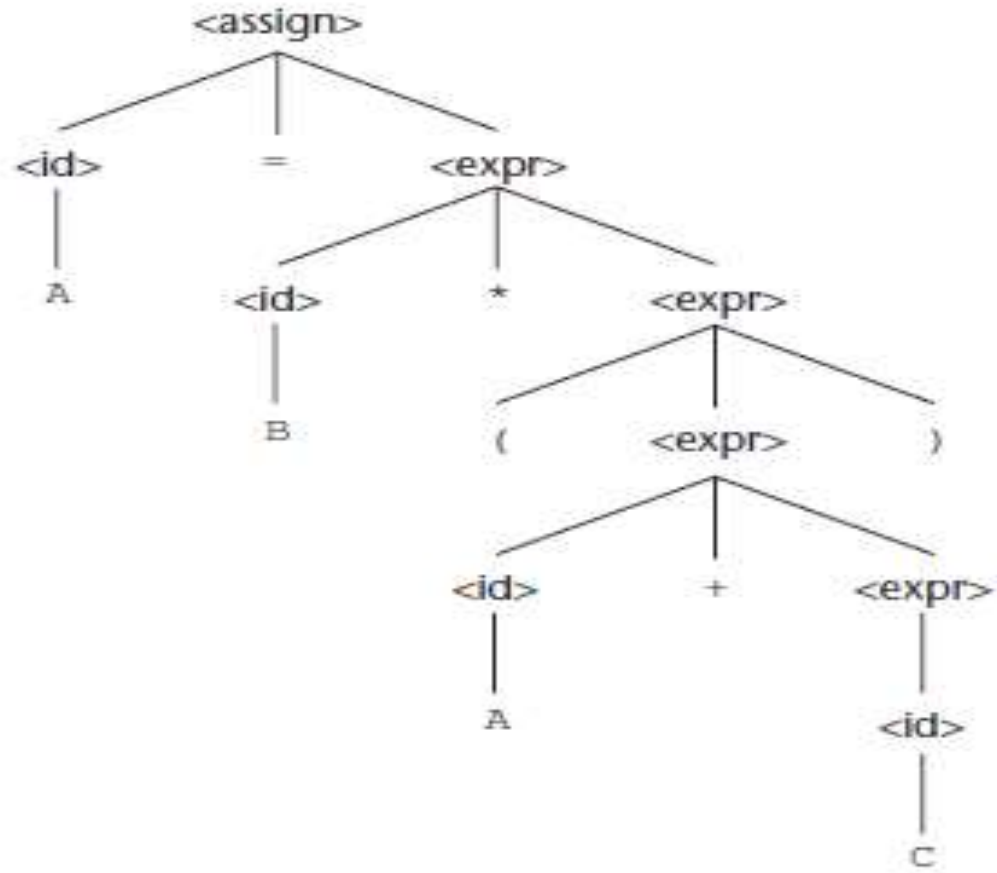
- The grammar is generated by the leftmost derivation as in Listing 3:
- Listing 3:

```
<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 )
```

# Parse Tree

- 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 the following figure shows the structure of the assignment statement derived previously in Listing 3.
- Every internal node of a parse tree is labelled with a nonterminal symbol; every leaf is labelled with a terminal symbol.
- Every subtree of a parse tree describes one instance of an abstraction in the sentence.

# Parse Tree



# Ambiguity

- A grammar is ambiguous if and only if it generates a sentential form that has two or more distinct parse trees.

- Consider Listing 1:

$\langle \text{assign} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{id} \rangle \rightarrow A \mid B \mid C$

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle$

$\mid \langle \text{expr} \rangle * \langle \text{expr} \rangle$

$\mid ( \langle \text{expr} \rangle )$

$\mid \langle \text{id} \rangle$

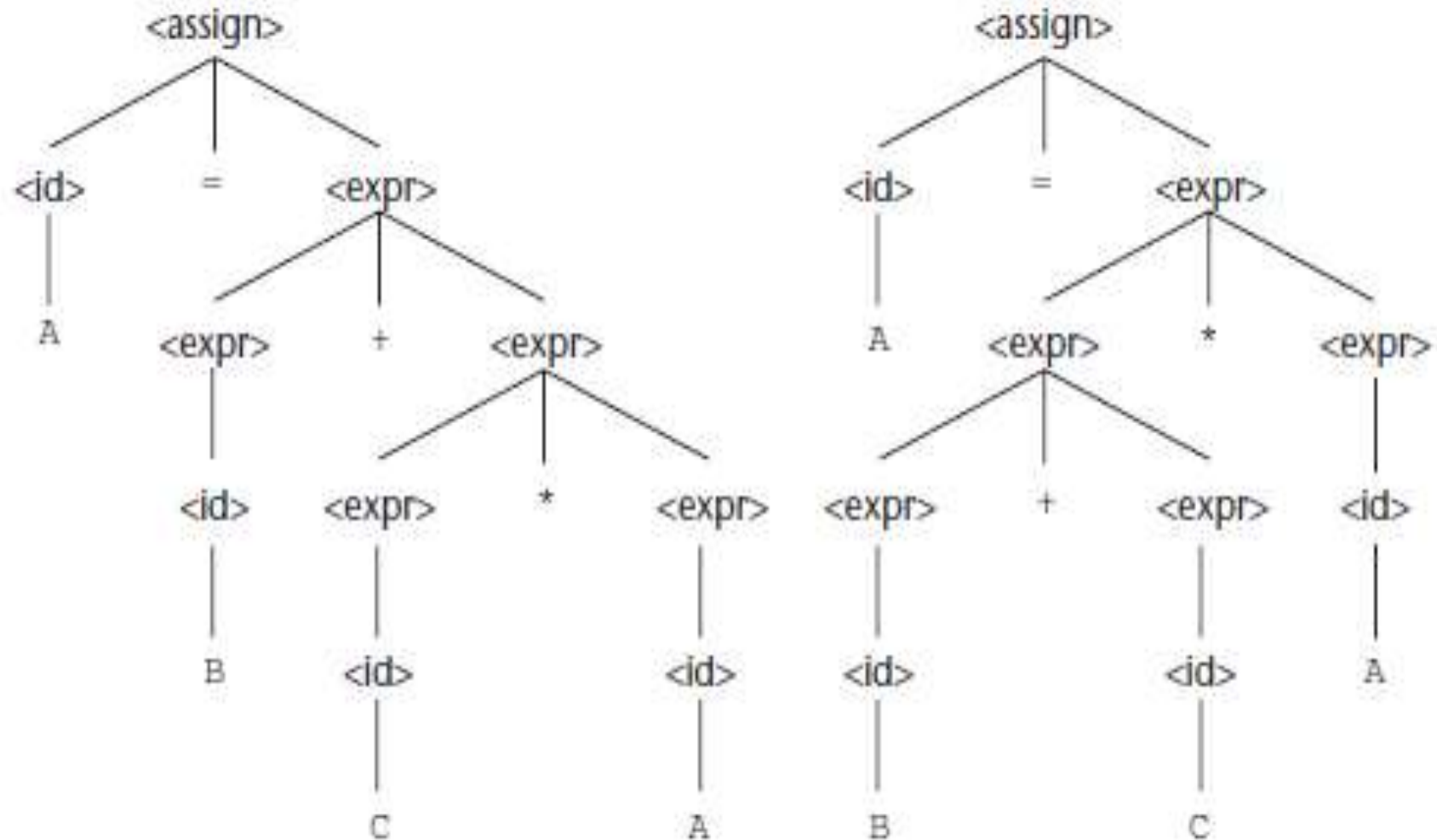
# Ambiguity

- The grammar in Listing 1 is ambiguous because, the sentence:

$$A = B + C * A$$

- Has two distinct parse tree as shown in the following figure:

# Ambiguity





# Abiguity



- Characteristically, a grammar is also ambiguous if:
  - If the grammar generates a sentence with more than one leftmost derivation and
  - If the grammar generates a sentence with more than one rightmost derivation.



# Ambiguity Problem

- 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.
- This problem is appropriately manifested in operator precedence.

# Static Semantic

- 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.

# Static Semantic

- 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.
- As an example of a syntax rule that cannot 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 cannot be specified in BNF.
- These problems exemplify the categories of language rules called static semantics rules.

# Static Semantic


- The **static semantics** of a language supposes that the semantic is not 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.

# Attribute Grammar

- Attribute grammars are a formal approach both to describing and checking the correctness of the static semantics rules of a program.
- 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 Grammar

- 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.
- 



# Concepts of Attribute Grammar

- Attribute grammars are context-free grammars to which have been added attributes, attribute computation functions, and predicate functions.
- **Attributes**, which are associated with grammar symbols (the terminal and nonterminal symbols), are similar to variables in the sense that they can have values assigned to them.
- **Attribute computation functions**, sometimes called semantic functions, are associated with grammar rules. They are used to specify how attribute values are computed.
- **. Predicate functions**, which state the static semantic rules of the language, are associated with grammar rules.



# Intrinsic Attributes

- **Intrinsic attributes** are synthesized attributes of leaf nodes whose values are determined outside the parse tree.
- For example, the type of an instance of a variable in a program could come from the symbol table, which is used to store variable names and their types.
- The contents of the symbol table are set based on earlier declaration statements.
- Initially, assuming that an unattributed parse tree has been constructed and that attribute values are needed, the only attributes with values are the intrinsic attributes of the leaf nodes.
- Given the intrinsic attribute values on a parse tree, the semantic functions can be used to compute the remaining attribute values.



Questions!!!