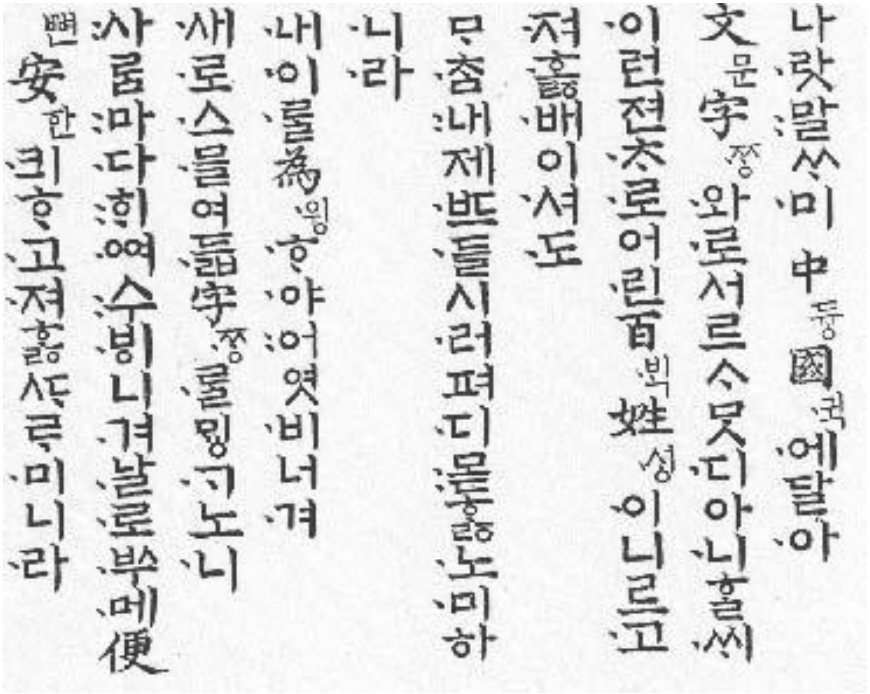
Chapter 3b: Syntax

CMSC 124, 1st Semester, AY 2009-10

Chapter 3b: Syntax

Describing Syntax

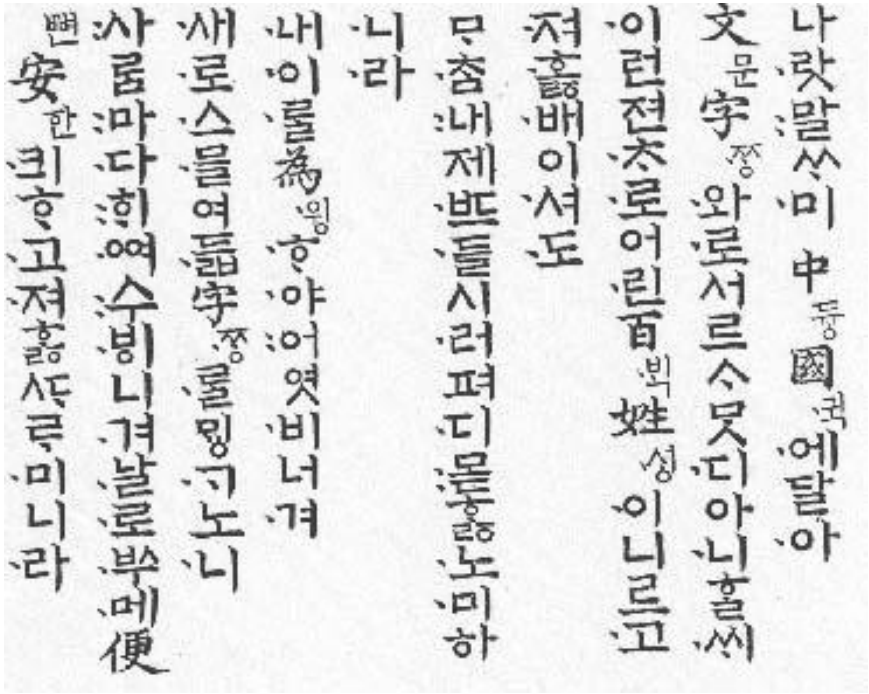
- **Language** consists of a set of strings (syntactically correct programs) of characters from some alphabet of symbols.
- Strings of a language are called **sentences** or **statements**.
- **Syntax rules** specify which strings of characters from the language's alphabet are in the language.



Chapter 3b: Syntax

Describing Syntax

- **Grammar**
  - Formal definition of the syntax of the language.
  - It naturally defines the hierarchical structure of many PL's.



Chapter 3b: Syntax

Grammar

Definition: A **grammar < ∑, N, P, S>** consists of four parts:

1. A finite set ∑ of **terminal symbols** of tokens.
2. A finite set **N** of **non-terminal symbols** or syntactic categories.
3. A finite set **P** of **productions** or **rules** that describe how each non-terminal is defined in terms of terminal symbols and non-terminals.
4. A distinguished non-terminal **S**, the **start symbol**, that specifies the category being defined.

Chapter 3b: Syntax

Sample English Grammar

- **Terminals**
  - the, boy, ran, ate, cake, a, an
- **Non-Terminals**
  - <sentence>, <subject>, <predicate>, <verb>, <article>, <noun>
- **Start Symbol**
  - One of the non-terminals.
- **Rules/Productions** (A finite set of replacement rules)

  <sentence> ::= <subject><predicate>
  <subject> ::= <article><noun>
  <predicate> ::= <verb><article><noun>
  <verb> ::= ran | ate
  <article> ::= the | a | an
  <noun> ::= boy | girl | cake

Try Later!

1. The boy ate the cake.
2. A cake ate the boy.

Chapter 3b: Syntax

Backus-Naur Form

- A grammar used to express the rules/production.
- Originally developed for the syntactic definition of Algol-60
- The BNF grammar is a set of rules or productions of the form:

```
left-side ::= right-side
```

where

✓ `left-side` is a non-terminal.

✓ `right-side` is a string of non-terminals and terminals.

John Backus

Chapter 3b: Syntax

Backus-Naur Form

- The BNF grammar is a set of rules or productions of the form:

```
left-side ::= right-side
```

- A **terminal** represents the atomic symbols in the language.
- A **non-terminal** represents for other symbols as defined to the right of the symbol **"::="**.
- The operator **"::="** is read as **"produces"**.
- The form above is read **"the non-terminal left-side produces right-side"**.
- **"|"** is interpreted as alternative.
- **"{}"** denotes possible repetition of the enclosed symbols 0 or more times.

Eg:

A ::= B | {C} | 9

"A produces B"
"A produces a string of 0 or more C's"
"A produces 9"

Something to Ponder

Given the grammar, how do determine if a particular string is a member of the lang.?

Something to Ponder

Chapter 3b: Syntax

Derivations

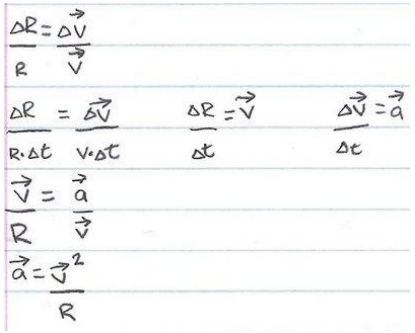
- Use derivations to determine. (a valid syntax in PL's)
- Derivation is a sequence of sentential forms starting from the start symbol.
- Either leftmost or rightmost derivation.
- Replace any non-terminal by a right hand side value using any rule.
- Symbol used: "=>"

Eg:

<sentence> ::= <subject><predicate>
<subject> ::= <article><noun>
<predicate> ::= <verb><article><noun>
<verb> ::= ran | ate
<article> ::= the | a | an
<noun> ::= boy | girl | cake

Try Now!

1. The boy ate the cake.
2. A cake ate the boy.



Chapter 3b: Syntax

Derivations

- **Derivation for #1:**

<sentence> => <subject><predicate>
=> <article> <noun> <predicate>
=> the <noun> <predicate>
=> the boy <predicate>
=> the boy <verb> <article> <noun>
=> the boy ate <article> <noun>
=> the boy ate the <noun>
=> the boy ate the cake

- Also from <sentence>, the statement **"a cake ate the boy"** can also be derived. What does that mean?
- Syntax **does not imply** correct semantics.

Chapter 3b: Syntax

Derivations: Another Example

- Show that 010 is a member of the following grammar:
  - ✓ **<B> ::= 0<B> | 1<B> | 0 | 1**

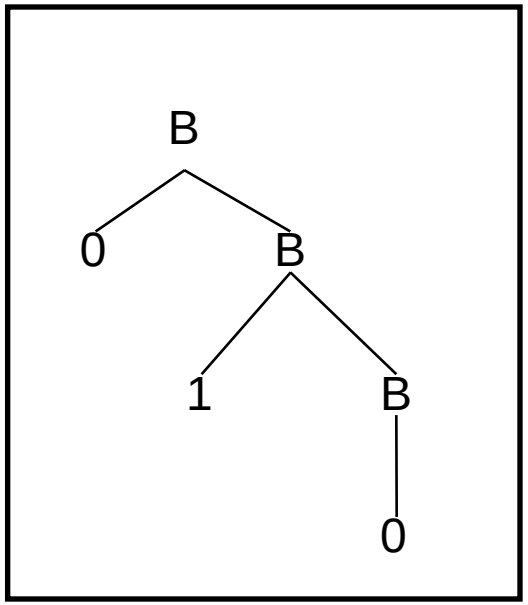
- Rightmost or Leftmost Derivation
  - ✓ **<B> => 0<B>**
    => 01<B>
    => 010

Chapter 3b: Syntax

Derivation/Parse Tree

- Graphically shows how the start symbol of a grammar derives a string in the language.

- Using a parse tree, show that 010 is a member of the grammar:
  - **B -> 0B | 1B | 0 | 1**

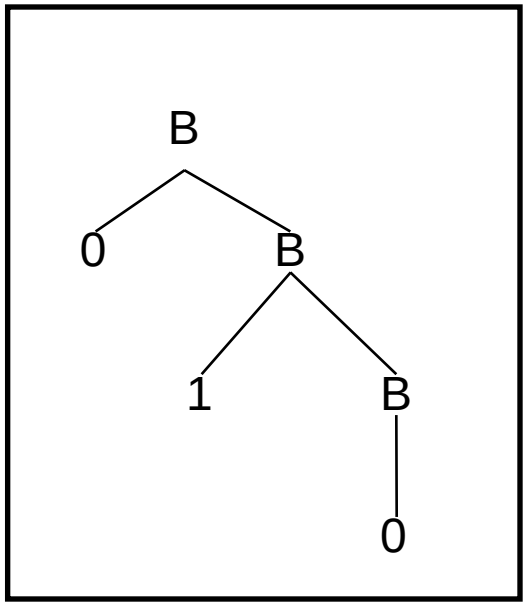


Chapter 3b: Syntax

Derivation/Parse Tree

Properties of a Parse Tree

- The **root** is labeled by the start symbol.
- Each **leaf** is labeled by a token or by **ε**.
- Each **interior node** is labeled by a non-terminal.
- If A is the non-terminal labeling some interior node and x1, x2,...xn are the children of that node from left to right then **A -> x1, x2,...xn** is a **production**.



Chapter 3b: Syntax

Derivation Tree: Sample PL Grammar

<expression> ::= <term> | <expression><addoperator><term>
<term> ::= <factor> | <term><multoperator><factor>
<factor> ::= <identifier> | <literal> | (<expression>)
<identifier> ::= a | b | c | ... | z
<literal> ::= 0 | 1 | 2 | ... | 9
<addoperator> ::= + | - | or
<multoperator> ::= * | / | div | mod | and

Generate a parse tree for the string a + b * c.

Chapter 3b: Syntax

Leftmost and Rightmost Derivations

Leftmost Derivation
- In each step, the leftmost non-terminal is replaced.

Rightmost Derivation
- In each step, the rightmost non-terminal is replaced.

Eg.
Consider the grammar:
G = ({S, A}, {a, b}, P, S)
P = { S -> aAS | a, A -> SbA | SS | ba }

Derive the string **aabbaa.**

Something to Ponder

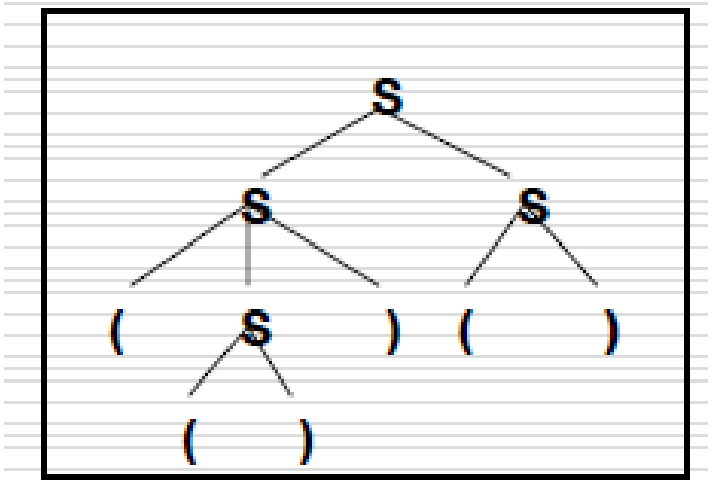
A certain statement can be generated by
2 or more distinct LM/RM derivations.
What's the implication?

Something to Ponder

Chapter 3b: Syntax

Different Derivations, Same Parse Tree

- Derivations may not be **unique.**
  - **Grammar:** SS -> SS | (S) | ( )
  - **Sentence:** (( )) ( )
  - **Derivations:**
    - S => SS => (S)S => (( ))S => (( ))( )
    - S => SS => S( ) => (S)( ) => (( ))( )
  - Different derivations but same parse tree.



Something to Ponder

Another statement can be generated by 2 or more distinct parse trees. What's the implication?

Something to Ponder

Chapter 3b: Syntax

Ambiguity

• A grammar that generates a sentence for which there are 2 or more distinct parse trees is said to be ambiguous.

Eg:
Consider the following grammar:

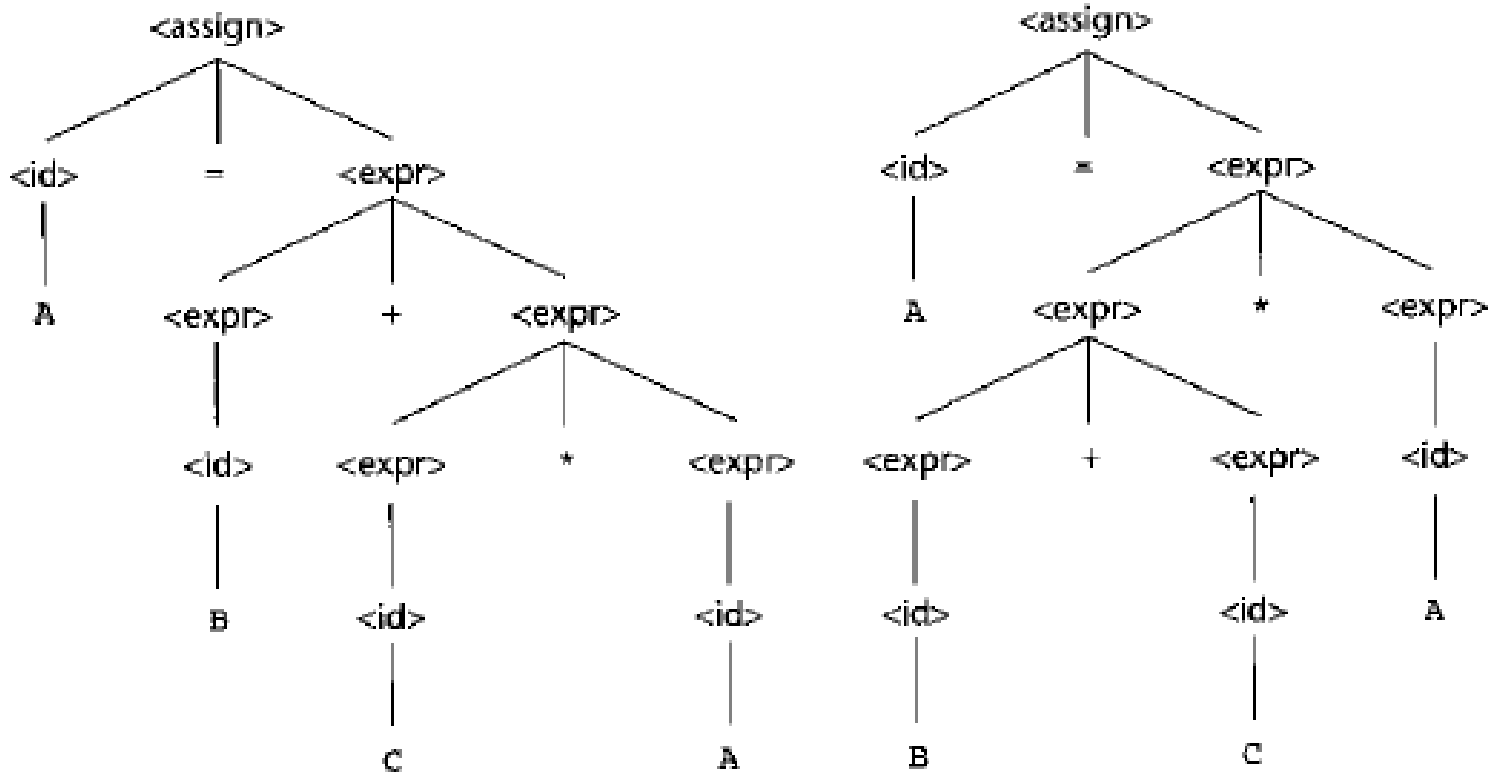
<assign> -> <id> = <expr>
<id> -> A | B | C
<expr> -> <expr> + <expr> | <expr> * <expr> | (<expr>) | <id>

Show the derivation for the sentence:

A = B + C * A

Chapter 3b: Syntax

Ambiguity



Two distinct parse trees for **A = B + C * A**

Chapter 3b: Syntax

The Problem of Ambiguity

• Syntactic ambiguity of language structures is a problem for compilers often base the semantics of those structures on their **syntactic form.**

• The compiler decides what code to generate for a stmt by examining its parse tree.

• Meaning of the structure cannot be determined **uniquely** if there are >1 parse tree.

Chapter 3b: Syntax

Ambiguity Issue #1: Operator Precedence

- A grammar can describe a certain syntactic structure so that part of the structure's meaning can follow its parse tree.
- **A fact:**

"If an operator in an arithmetic expression is generated **lower** in the parse tree, it indicates that it has **higher** precedence over an operator produced higher up in the tree."

- A grammar can be rewritten to separate addition and multiplication operators.
- Rewriting would require additional non-terminals and some new rules.

Chapter 3b: Syntax

Ambiguity Issue #1: Operator Precedence

ORIGINAL GRAMMAR

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

MODIFIED GRAMMAR

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

NOW!
Generate a parse tree for **A = B + C * A**

Chapter 3b: Syntax

Ambiguity Issue #2: Operator Associativity

- Another interesting question is whether operator associativity is also **correctly described**.
  - Expressions with 2 or more adjacent occurrences of operators with equal precedence have those occurrences in proper hierarchical order?

Chapter 3b: Syntax

Ambiguity Issue #2: Operator Associativity

- Consider the following statements:
  - **A = B + C + A**
  - How do we specify the associativity?

Left Associativity

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

Left Recursive

A BNF rule has its left hand side (LHS) appear at the beginning of its RHS.

Chapter 3b: Syntax

Ambiguity Issue #2: Operator Associativity

- Consider the following statements:
  - **A = B + C + A**
  - How do we specify the associativity?

Right Associativity

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

Right Recursive
A BNF rule has its left hand side (LHS) appear at the right end of its RHS.

Chapter 3b: Syntax

Ambiguity Issue #2: Operator Associativity

- In addition and multiplication, it does not matter what kind of associativity to be used. **Why?**
- But in other operations, like **exponentiation**, the kind of associativity should be **defined.**
- In most PL's, the exponentiation operator is right associative.
    - A right recursive grammar rule is provided.

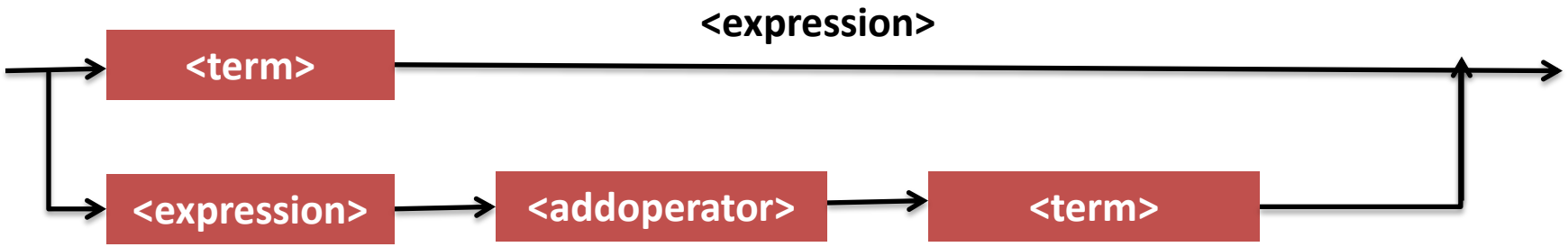
Eg:

<factor> -> <expr> ** <factor> | <expr>
<expr> -> (<expr>) | <id>

Chapter 3b: Syntax

Syntax Diagrams

- A **graphical way** used to represent BNF rules.
- For each grammar rule, an equivalent syntax diagram can be drawn.
- This was popularized in the design of Pascal.
- **Symbols:**
  - Rectangle nodes for non-terminals.
  - Circles for terminals.



Sample syntax diagram for
<expression>::=<term> | <expression><addoperator><term>