An Example Grammar

```
<program> -> <stmts>

<stmts>   -> <stmt>
           |  <stmt> ; <stmts>

<stmt>    -> <var> = <expr>

<var>     -> a | b | c | d

<expr>    -> <term> + <term>
           |  <term> - <term>

<term>    -> <var>
           |  const
```


An Exemplar Derivation

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

← **sentence**

Sentential Forms

- Every string of symbols in the derivation is a **sentential form**
- A **sentence** is a sentential form that has only terminal symbols
- A **leftmost derivation** is one in which the leftmost non-terminal in each sentential form is the one that is expanded next in the derivation

3

Sentential Forms

- A **left-sentential form** is a sentential form that occurs in the leftmost derivation
- A **rightmost derivation** works right to left instead
- A **right-sentential form** is a sentential form that occurs in the rightmost derivation
- Some derivations are neither leftmost nor rightmost

4

Why BNF?

- Provides a clear and concise syntax description
- The parse tree can be generated from BNF
- Parsers can be based on BNF and are easy to maintain

Context-Free Grammars

- The syntax of simple arithmetic expression

```
expr -> id | number | -expr |(expr)
       |expr op expr
  op -> + | - | * | /
```

- What are the terminal symbols and nonterminal symbols?
- What is the start symbol?

One Possible Derivation

```
expr => expr op expr
     => …
     => id + number
```


Another Example

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

- G = {T, N, S, P}
- What are the terminals?
- What are the nonterminals?
- What is the start symbol?
- Possible strings?

Parse Tree

- A **parse tree** is
  - a hierarchical representation of a derivation
  - to represent the structure of the derivation of a terminal string from some non-terminal
  - to describe the hierarchical syntactic structure of programs for any language

9

An Example

- Given the simple assignment statement syntax

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

- With leftmost derivation, how is A = B * (A + C) generated?

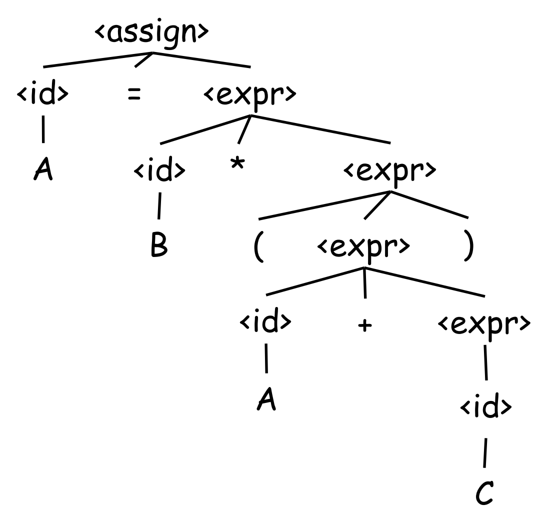
10

Derivation for A = B * (A + C)

<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)

11

The Parse Tree for A = B * (A + C)



12

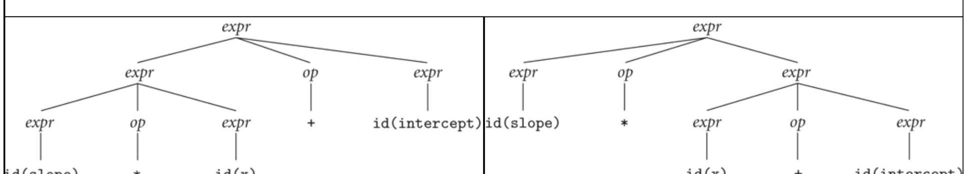
Parse Tree

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

An Ambiguous Grammar

```
expr -> id | number | -expr |(expr)
        | expr op expr
  op -> + | - | * | /
```

- Parse trees for “slope * x + intercept”:



What goes wrong?

- The production rules do not capture the **associativity** and **precedence** of various operators
  - **Associativity** tells whether the operators group left to right or right to left
    - Is 10 – 4 – 3 equal to (10 - 4) – 3 or 10 – (4 – 3) ?
  - **Precedence** tells some operators group more tightly than the others?
    - Is slope * x + intercept equal to (slope * x) + intercept or slope * (x + intercept)?

15

Operator Associativity

- Single recursion in production rules

```
<expr> -> <expr> - <expr> | const
```

✗ **Ambiguous**

```
<expr> -> <expr> - const | const
```

✓ **Unambiguous**

```
<expr> -> const - <expr> | const
```

✓ **Unambiguous (less desirable)**

16

Operator Precedence

- Use stratification in production rules
  - Intentionally put operators at different levels of parse trees

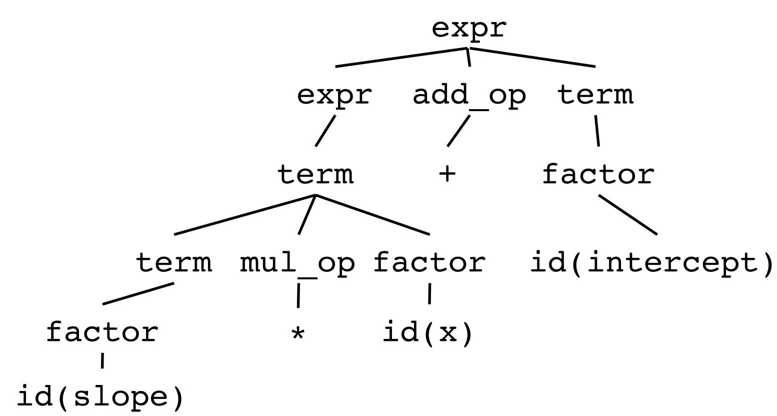
```
<expr> -> <expr> - <term> | <term>
<term> -> <term> / const  | const
```


Improved Unambiguous Context-Free Grammar

```
1. expr -> expr add_op term
         | term
2. term -> term mul_op factor | factor
3. factor -> id | number | -factor
                          | (expr)
3. add_op -> + | -
4. mul_op -> * | /
```

Revisit "slope * x + intercept"

- Parse Tree



19

Extended BNF (EBNF)

- There are extensions of BNF to simplify representation
  - Kleene star ***** or **{}** to represent repetition (0 or more)
  - **()** to represent alternative parts
  - **[]** to represent optional parts
    - id_list -> id **(**, id**)***
    - proc_call -> id'('**[**expr_list**]**')'

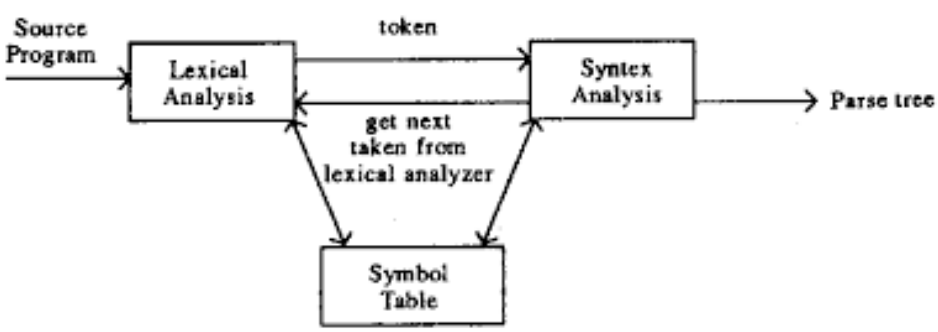
20

Lexical and Syntactic Analysis

- Two steps to discover the syntactic structure of a program
  - Lexical analysis (Scanner): to read the input characters and output a sequence of tokens
  - Syntactic analysis (Parser): to read the tokens and output a parse tree and report syntax errors if any

21

Interaction between lexical analysis and syntactic analysis



22

Scanner

- Pattern matcher for character strings
  - If a character sequence matches a pattern, it is identified as a token
- Responsibilities
  - Tokenize source, report lexical errors if any, remove comments and whitespace, save text of interesting tokens, save source locations, (optional) expand macros and implement preprocessor functions

23

Tokenizing Source

- Given a program, identify all lexemes and their categories (tokens)

24