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

---

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
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
  \[ expr \rightarrow id \mid number \mid -expr \mid (expr) \mid expr \ op \ expr \]
  \[ op \rightarrow + \mid - \mid * \mid / \]
• 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

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?
Derivation for $A = B \times (A + C)$

\begin{align*}
\langle \text{assign} \rangle & \Rightarrow \langle \text{id} \rangle = \langle \text{expr} \rangle \\
& \Rightarrow A = \langle \text{expr} \rangle \\
& \Rightarrow A = \langle \text{id} \rangle \times \langle \text{expr} \rangle \\
& \Rightarrow A = B \times \langle \text{expr} \rangle \\
& \Rightarrow A = B \times (\langle \text{expr} \rangle) \\
& \Rightarrow A = B \times (\langle \text{id} \rangle + \langle \text{expr} \rangle) \\
& \Rightarrow A = B \times (A + \langle \text{expr} \rangle) \\
& \Rightarrow A = B \times (A + \langle \text{id} \rangle) \\
& \Rightarrow A = B \times (A + C)
\end{align*}

The Parse Tree for $A = B \times (A + C)$
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”:

```
expr expr
    op
  expr  id(intercept)
        * id(slope)
expr expr
    op
  expr  id(slope)
        * expr
        op
      expr
expr expr
    op
  expr  id(slope)
        * expr
        op
      expr
expr expr
    op
  expr  id(slope)
        * id(slope)
        op
      expr
expr expr
    op
  expr  id(slope)
        * expr
        op
      expr
expr expr
    op
  expr  id(slope)
        * id(slope)
        op
      expr
expr expr
    op
  expr  id(slope)
        * id(slope)
        op
      expr
```
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)?

Operator Associativity

• Single recursion in production rules

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

  ✗ Ambiguous

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

  ✓ Unambiguous

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

  ✓ Unambiguous (less desirable)
Operator Precedence

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

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr} + \text{term} \\
& \quad \text{term} \\
\text{term} & \rightarrow \text{term} \times \text{factor} \\
& \quad \text{factor} \\
\text{factor} & \rightarrow \text{id} \mid \text{number} \mid -\text{factor} \\
& \quad \text{(expr)} \\
\text{add_op} & \rightarrow + \mid - \\
\text{mul_op} & \rightarrow * \mid / 
\end{align*}
\]
Revisit “slope * x + intercept”

- Parse Tree

```
expr
  /\       /\      /\    /\           /\   
expr  add_op term   term + factor   term mul_op factor id(intercept)
  /\    /\          /\     /\                      /\               /\ 
term  +  factor    id(slope)    id(x)               id(slope)        id(x)
```

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]')
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

Interaction between lexical analysis and syntactic analysis
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

Tokenizing Source

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