Implementation Methods

- **Compilation**
  - Translate high-level program to machine code
  - Slow translation
  - Fast execution

![Compilation Diagram]

Compilation Process
Implementation Methods

- Pure interpretation
  - No translation
  - Slow execution
  - Becoming rare

![Pure interpretation diagram](image)

Implementation Methods

- Hybrid implementation systems
  - Small translation cost
  - Medium execution speed

![Hybrid implementation diagram](image)
Programming Environments

- The collection of tools used in software development
- UNIX
  - An older operating system and tool collection
- Borland JBuilder
  - An integrated development environment for Java
- Microsoft Visual Studio.NET
  - A large, complex visual environment
  - Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++

Describing Syntax

- Lexemes: lowest-level syntactic units
- Tokens: categories of lexemes

\[ \text{sum} = x + 2 - 3 \]

Lexemes: \( \text{sum}, =, x, +, 2, -, 3 \)

Tokens: identifier, equal_sign, plus_op, integer_literal, minus_op

Formal Method for Describing Syntax

- Backus-Naur form (BNF)
  - Also equivalent to context-free grammars, developed by Noam Chomsky (a linguist)
  - BNF is a meta-language
  - a language used to describe another language
  - Consists of a collection of rules (or productions)
  - Example of a rule:
    \[ <\text{assign}> : \text{var} \rightarrow <\text{expression}> \]
    - LHS: the abstraction being defined
    - RHS: contains a mixture of tokens, lexemes, and references to other abstractions
  - Abstractions are called non-terminal symbols
  - Lexemes and tokens are called terminal symbols
  - Also contains a special non-terminal symbol called the start symbol
Example of a grammar in BNF

\[
\text{<program>} \ ? \begin{align*}
& \begin{align*}
& \text{<stmt_list>} \end{align*} \\
& \text{<stmt>} | \text{<stmt>;} \text{<stmt_list>}
\end{align*} \\
\text{<stmt>} \ ? \ \text{<var>} = \text{<expression>}
\text{<var>} \ ? \ A | B | C | D
\text{<expression>} \ ? \ \text{<var>} + \text{<var>} \mid \text{<var>} - \text{<var>} \mid \text{<var>}
\]

Derivation

The process of generating a sentence

\[
\begin{align*}
& \text{begin} \ \text{A} = \text{B} - \text{C} \text{ end}
\end{align*}
\]

Derivation:  \text{<program>} \ (\text{start symbol})
\rightarrow \text{begin} \text{<stmt_list>} \text{ end}
\rightarrow \text{begin} \text{<stmt>} \text{ end}
\rightarrow \text{begin} \text{<var>} = \text{<expression>} \text{ end}
\rightarrow \text{begin} \text{A} = \text{<expression>} \text{ end}
\rightarrow \text{begin} \text{A} = \text{<var>} - \text{<var>} \text{ end}
\rightarrow \text{begin} \text{A} = \text{B} - \text{<var>} \text{ end}
\rightarrow \text{begin} \text{A} = \text{B} - \text{C} \text{ end}
\]

BNF

- Leftmost derivation:
  - the replaced non-terminal is always the leftmost non-terminal
- Rightmost derivation
  - the replaced non-terminal is always the rightmost non-terminal
- Sentential forms
  - Each string in the derivation, including
  \text{<program>}
Derivation

\begin{align*}
\text{begin } A = B + C; B = C \text{ end}
\end{align*}

Rightmost:  
\begin{align*}
&<\text{program}> \\
&\Rightarrow \begin{align*}
&\text{begin} \\
&<\text{stmt_list}> \\
&\text{end} \\
&\Rightarrow \begin{align*}
&\text{begin} \\
&<\text{stmt}> <\text{stmt_list}> \\
&\text{end} \\
&\Rightarrow \begin{align*}
&\text{begin} \\
&<\text{stmt}> <\text{var}> = <\text{expression}> \\
&\text{end} \\
&\Rightarrow \begin{align*}
&\text{begin} \\
&<\text{stmt}> <\text{var}> = C \\
&\text{end} \\
&\Rightarrow \begin{align*}
&\text{begin} \\
&B = C \\
&\text{end} \\
&\Rightarrow \begin{align*}
&A = B + C; B = C \\
&\text{end}
\end{align*}
\end{align*}
\end{align*}

Parse Tree

A hierarchical structure that shows the derivation process

Example:
\begin{align*}
A = B \ast (A + C)
\end{align*}

\begin{align*}
&<\text{assign}> \\
&\Rightarrow \begin{align*}
&A = <\text{expr}> \\
&\Rightarrow \begin{align*}
&A = <\text{id}> * <\text{expr}> \\
&\Rightarrow \begin{align*}
&A = B * <\text{expr}> \\
\end{align*}
\end{align*}
\end{align*}

\begin{align*}
&<\text{expr}> \\
&\Rightarrow \begin{align*}
&A + <\text{expr}> \\
&A = B + ( <\text{expr}> ) \\
&A = B + ( <\text{id}> + <\text{expr}> ) \\
&A = B + ( <\text{id}> * <\text{expr}> ) \\
&A = B + ( A + <\text{id}> ) \\
&A = B + ( A + C ) \\
\end{align*}
\end{align*}
A grammar that generates a sentence for which there are two or more distinct parse trees is said to be ambiguous.

Example:

\[
\begin{align*}
\langle \text{assign} \rangle & \rightarrow \langle \text{id} \rangle + \langle \text{expr} \rangle \\
\langle \text{id} \rangle & \rightarrow \text{A} \mid \text{B} \mid \text{C} \mid \text{D} \\
\langle \text{expr} \rangle & \rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{expr} \rangle \ast \langle \text{expr} \rangle \mid ( \langle \text{expr} \rangle ) \mid \langle \text{id} \rangle
\end{align*}
\]

Draw two different parse trees for

\[A = B + C \ast A\]

Is the following grammar ambiguous?

\[
\begin{align*}
\langle \text{if_stmt} \rangle & \rightarrow \text{if} \langle \text{logic_expr} \rangle \text{then} \langle \text{stmt} \rangle \\
& \quad \mid \text{if} \langle \text{logic_expr} \rangle \text{then} \langle \text{stmt} \rangle \text{else} \langle \text{stmt} \rangle
\end{align*}
\]
Operator Precedence

\[ A = B + C \times A \]

- How to force \( \times \) to have higher precedence over \(+\)?
- Answer: add more non-terminal symbols
- Observe that higher precedent operators reside at “deeper” levels of the trees

A = B + C * A

Before:
- Assign
- \( <id> + <expr> \)
- \( <id> \)
- A | B | C | D
- \( <expr> \)
  - \( <expr> + <expr> \)
  - \( ( <expr> ) \)
  - \( <id> \)

After:
- Assign
- \( <id> + <expr> \)
- \( <id> \)
- A | B | C | D
- \( <expr> + <term> \)
- \( <term> \)
  - \( <term> * <factor> \)
  - \( <factor> \)
  - \( ( <expr> ) \)
  - \( <id> \)
**Associativity of Operators**

A = B + C – D * F / G

- **Left-associative**
  - Operators of the same precedence evaluated from left to right
  - C++/Java: +, -, *, /, %

- **Right-associative**
  - Operators of the same precedence evaluated from right to left
  - C++/Java: unary -, unary +, ! (logical negation), ~ (bitwise complement)

- How to enforce operator associativity using BNF?

---

**Associative of Operators**

- Left-recursive rule
  - \(<assign> \ ? \ <id> = <expr> \)
  - \(<id> \ ? \ A | B | C | D \)
  - \(<exp> \ ? \ <expr> + <term> \)
    - \(<term> \ ? \ <term> * <factor> \)
    - \(<factor> \ ? \ (<expr>) | <id> \)

- Left-associative

---

**Associativity of Operators**

- Right-recursive rule
  - \(<assign> \ ? \ <id> = <factor> \)
  - \(<factor> \ ? \ <exp> ^ <factor> \)

- Exercise: Draw the parse tree for A = B^C^D
  (use leftmost derivation)
Extended BNF

- BNF rules may grow unwieldy for complex languages
- Extended BNF
  - Provide extensions to “abbreviate” the rules into much simpler forms
  - Does not enhance descriptive power of BNF
  - Increase readability and writability

1. Optional parts are placed in brackets ([ ])
   \[ <select_stmt> ? \] = ( <expr> ) <stmt> \[ else <stmt> \]

2. Put alternative parts of RHSs in parentheses and separate them with vertical bars
   \[ <term> ? <term> (+ | -) const \]

3. Put repetitions (0 or more) in braces ({})
   \[ <id_list> ? <id> { , <id> } \]

Extended BNF (Example)

<table>
<thead>
<tr>
<th>BNF:</th>
<th>EBNF:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ &lt;expr&gt; ?</td>
<td>[ &lt;expr&gt; ? &lt;term&gt;</td>
</tr>
<tr>
<td>] + &lt;term&gt; ]</td>
<td>] ((+</td>
</tr>
<tr>
<td>] ^ &lt;factor&gt; ]</td>
<td>] (^[])/&lt;factor&gt; ]</td>
</tr>
<tr>
<td>] * &lt;factor&gt; ]</td>
<td>] ^ &lt;factor&gt; ]</td>
</tr>
<tr>
<td>] / &lt;factor&gt; ]</td>
<td>] ^ &lt;factor&gt; ]</td>
</tr>
<tr>
<td>] [ &lt;exp&gt; ]</td>
<td>] ^ &lt;factor&gt; ]</td>
</tr>
<tr>
<td>] [ &lt;expr&gt; ]</td>
<td>] ^ &lt;factor&gt; ]</td>
</tr>
<tr>
<td>] [ &lt;id&gt; ]</td>
<td>] ^ &lt;factor&gt; ]</td>
</tr>
</tbody>
</table>

BNF:

- Extended BNF
  - Provide extensions to “abbreviate” the rules into much simpler forms
  - Does not enhance descriptive power of BNF
  - Increase readability and writability
Lexical Analyzer

- A pattern matcher for character strings
- The “front-end” for the parser
- Identifies substrings of the source program that belong together => lexemes
  - Lexemes match a character pattern, which is associated with a lexical category called a token
  
  **Example:**
  
  \[
  \text{sum} = B - 5;
  \]

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>ID</td>
</tr>
<tr>
<td>=</td>
<td>ASSIGN_OP</td>
</tr>
<tr>
<td>B</td>
<td>ID</td>
</tr>
<tr>
<td>-</td>
<td>SUBTRACT_OP</td>
</tr>
<tr>
<td>5</td>
<td>INT_LIT</td>
</tr>
<tr>
<td>;</td>
<td>SEMICOLON</td>
</tr>
</tbody>
</table>

- Functions:
  1. Extract lexemes from a given input string and produce the corresponding tokens, while skipping comments and blanks
  2. Insert lexemes for user-defined names into symbol table, which is used by later phases of the compiler
  3. Detect syntactic errors in tokens and report such errors to user

- How to build a lexical analyzer?
  - Create a state transition diagram first
    - A state diagram is a directed graph
    - Nodes are labeled with state names
    - One of the nodes is designated as the start node
    - Arrows are labeled with input characters that cause the transitions
Lexical Analyzer

- Need to distinguish reserved words from identifiers
- E.g., reserved words: `main` and `int`
- Identifiers: `sum` and `B`
- Use a table lookup to determine whether a possible identifier is in fact a reserved word

Lexical Analyzer

- Useful subprograms in the lexical analyzer:
  1. `lookup`
     - Determines whether the string in lexeme is a reserved word (returns a code)
  2. `getChar`
     - Reads the next character of input string, puts it in a global variable called `nextChar`, determines its character class (letter, digit, etc.) and puts the class in `charClass`
  3. `addChar`
     - Appends `nextChar` to the current lexeme
Lexical Analyzer

```c
int lex() {
    switch (charClass) {
    case LETTER:
        addChar();
        getChar();
        while (charClass == LETTER || charClass == DIGIT) {
            addChar();
            getChar();
        }
        return lookup(lexeme);
        break;
    case DIGIT:
        addChar();
        getChar();
        while (charClass == DIGIT) {
            addChar();
            getChar();
        }
        return INT_LIT;
        break;
    } /* End of switch */
} /* End of function lex */
```

Parsers (Syntax Analyzers)

- Goals of a parser:
  - Find all syntax errors
  - Produce parse trees for input program
- Two categories of parsers:
  - **Top down**
    - produces the parse tree, beginning at the root
    - Uses leftmost derivation
  - **Bottom up**
    - produces the parse tree, beginning at the leaves
    - Uses the reverse of a rightmost derivation

Recursive Descent Parser

- A top-down parser implementation
- Consists of a collection of subprograms
  - A recursive descent parser has a subprogram for each non-terminal symbol
- If there are multiple RHS for a given nonterminal, parser must make a decision which RHS to apply first
  - $A \Rightarrow x \ldots | y \ldots | z \ldots \ldots$
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
### Recursive Descent Parser

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

#### <term>
- `term();`
- `while (nextToken == PLUS_CODE ||
  nextToken == MINUS_CODE ) {
  lex();
  term();
}

#### <factor>
- `/* Determine which RHS */
- if (nextToken == ID_CODE)
  lex();
- else if (nextToken == LEFT_PAREN_CODE) {
  lex();
  expr();
  if (nextToken == RIGHT_PAREN_CODE)
    lex();
  else
    error();
}
- else
  error(); /* Neither RHS matches */

1. lex() is the lexical analyzer function. It gets the next lexeme and puts its token code in the global variable `nextToken`.
2. All subprograms are written with the convention that each one leaves the next token of input in `nextToken`.
3. Parser uses leftmost derivation.

---

Problem with left recursion
- A ↛ A + B (direct left recursion)
- A ↛ B c D (indirect left recursion)
- B ↛ A b

A grammar can be modified to remove left recursion
- Inability to determine the correct RHS on the basis of one token of lookahead
  - Example: A ↛ aC | Bd
    - B ↛ ac
    - C ↛ c
LR Parsing

- LR Parsers are almost always table-driven
- Uses a big loop to repeatedly inspect 2-dimen
  table to find out what action to take
- Table is indexed by current input token and
  current state
- Stack contains record of what has been seen SO
  FAR (not what is expected/predicted to see in
  future)
- PDA: Push down automata:
  - State diagram looks just like a DFA state diagram
  - Arcs labeled with <input symbol, top-of-stack
    symbol>

PDAs

- LR PDA: is a recognizer
- Builds a parse tree bottom up
- States keep track of which productions we
  “might” be in the middle of.

Example

```plaintext
<pgm>     ->  <stmt list> $$
<stmt list> ->  <stmt list> <stmt>
<stmt>    ->  id := <expr> | read id | write <expr>
<expr>    ->  <term> | <expr> <add op> <term>
<term>    ->  <factor> | <term> <mult op> 
<factor>  ->  ( <expr> ) | id | literal
<add op>  ->  + | - 
<mult op> ->  * | /
```

1. read A
2. read B
3. sum := A + B
4. write sum
5. write sum / 2

See handout for trace of parsing.
Static Semantics

- BNF cannot describe all of the syntax of PLs
- Examples:
  - All variables must be declared before they are referenced
  - The end of an ADA subprogram is followed by a name, that name must match the name of the subprogram
    
    ```
    Procedure Proc_example (P: in Object) is
    begin
    ...end Proc_example
    ```

- Static semantics
  - Rules that further constrain syntactically correct programs
  - In most cases, related to the type constraints of a language
  - Static semantics are verified before program execution (unlike dynamic semantics, which describes the effect of executing the program)

- BNF cannot describe static semantics

Attribute Grammars (Knuth, 1968)

- A BNF grammar with the following additions:
  1. For each symbol x there is a set of attribute values, A(x)
     - A(X) = S(X) ? I(X)
     - S(X): synthesized attributes
       - used to pass semantic information up a parse tree
     - I(X): inherited attributes
       - used to pass semantic information down a parse tree
  2. Each grammar rule has a set of functions that define certain attributes of the nonterminals in the rule
     - Rule: X₁ : X₂ ;... Xₙ
       - S(Xₙ) = f (A(X₁), ..., A(Xₙ))
       - I(Xₙ) = f (A(X₁), ..., A(Xₙ))
  3. A (possibly empty) set of predicate functions to check whether static semantics are violated
     - Example: S(X) = f (X) ?
Procedure Proc_example (P: in Object) is
begin
…
end Proc_example
Syntax rule:
<Proc_def> : Procedure <proc_name>[1]
<proc_body> end <proc_name>[2]
Semantic rule:
<proc_names>[1].string = <proc_names>[2].string

Expressions of the form <var> + <var>
var's can be either int_type or real_type
If both var's are int, result of expr is int
If at least one of the var's is real, result of expr is real
BNF
<assign> = <var> = <expr> (Rule 1)
<expr> = <var> + <var> (Rule 2)
<var> = <var> (Rule 3)
<var> = A | B | C (Rule 4)
Attributes for non-terminal symbols <var> and <expr>
actual_type - synthesized attribute for <var> and <expr>
expected_type - inherited attribute for <expr>
1. Syntax rule: <assign> = <var> = <expr>
   Semantic rule: <expr>.expected_type = <var>.actual_type
   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 = lookup(<var>-string)
Note: Lookup function looks up a given variable name in the symbol table
and returns the variable's type
Parse Trees for Attribute Grammars

A = A + B

How are attribute values computed?
1. If all attributes were inherited, the tree could be decorated in top-down order.
2. If all attributes were synthesized, the tree could be decorated in bottom-up order.
3. If both kinds of attributes are present, some combination of top-down and bottom-up must be used.
Attribute Grammar Implementation

- Determining attribute evaluation order is a complex problem, requiring the construction of a dependency graph to show all attribute dependencies.

- Difficulties in implementation:
  - The large number of attributes and semantic rules required make such grammars difficult to write and read.
  - Attribute values for large parse trees are costly to evaluate.
  - Less formal attribute grammars are used by compiler writers to check static semantic rules.

Describing (Dynamic) Semantics

- `<for_stmt>` ? `for (<expr1>; <expr2>; <expr3>)`
- `<assign_stmt>` ? `<var> = <expr>;`

What is the meaning of each statement?

- dynamic semantics

How do we formally describe the dynamic semantics?

Describing (Dynamic) Semantics

- There is no single widely acceptable notation or formalism for describing dynamic semantics.

- Three formal methods:
  1. Operational Semantics
  2. Axiomatic Semantics
  3. Denotational Semantics
Operational Semantics

- Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement.

  ![Diagram](image)

  Initial State: \((i_1, v_1), (i_2, v_2), \ldots\)
  Final State: \((i_1', v_1'), (i_2', v_2'), \ldots\)

Operational Semantics

- To use operational semantics for a high-level language, a virtual machine is needed.
  - A hardware pure interpreter would be too expensive.
  - A software pure interpreter also has problems:
    1. The detailed characteristics of the particular computer would make actions difficult to understand.
    2. Such a semantic definition would be machine-dependent.

Operational Semantics

- Approach: use a complete computer simulation
  1. Build a translator (translates source code to the machine code of an idealized computer)
  2. Build a simulator for the idealized computer

- Example:

  C Statement:
  ```c
  for (expr1; expr2; expr3) {
    expr1;
    loop: if expr2 == 0 goto out
    expr3;
    goto loop
  }
  ...
  out: ...
  ```

  Operational Semantics:
  ```c
  ...
  ```
Operational Semantics

- Valid statements for the idealized computer:
  - `iden = var`
  - `iden = iden + 1`
  - `iden = iden - 1`
  - `goto label`
  - `if var relop var goto label`

- Evaluation of Operational Semantics:
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g., VDL)

Axiomatic Semantics

- Based on formal logic (first order predicate calculus)

- Approach:
  - Each statement is preceded and followed by a logical expression that specifies constraints on program variables
  - The logical expressions are called predicates or assertions
  - Define axioms or inference rules for each statement type in the language
    - to allow transformations of expressions to other expressions

Axiomatic Semantics

\[
\{P\} \quad A = B + 1 \quad \{Q\}
\]

where

- \(P\): precondition
- \(Q\): postcondition

- Precondition: an assertion before a statement that states the relationships and constraints among variables that are true at that point in execution
- Postcondition: an assertion following a statement
- A weakest precondition is the least restrictive precondition that will guarantee the postcondition

- Example: \(A = B + 1\) \(\{A > 1\}\)
  - Postcondition: \(A > 1\)
  - One possible precondition: \(B > 10\)
  - Weakest precondition: \(B > 0\)

- Evaluation of Axiomatic Semantics:
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g., VDL)
Axiomatic Semantics

- Program proof process:
  - The postcondition for the whole program is the desired results.
  - Work back through the program to the first statement.
  - If the precondition on the first statement is the same as the program spec, the program is correct.
- An axiom for assignment statements

\[ \text{Axiom: } P = Q_x, \quad x = E \quad (\text{P is computed with all instances of } x \text{ replaced by } E) \]

- Example: \( a = b / 2 - 1 \) \( (a < 10) \)
- Weakest precondition: \( b/2 - 1 < 10 \implies b < 22 \)
- Axiomatic Semantics for assignment:

\[ \{Q_x, \_ \} \quad x = E \quad \{Q\} \]

Inference rule for Sequences:

\[
\begin{align*}
\{P1\} & \quad S_1 \quad \{P2\}, \\
\{P2\} & \quad S_2 \quad \{P3\}
\end{align*}
\]

\[ \{P1\} \quad S_1; \quad S_2 \quad \{P3\} \]

- Example:

\( Y = 3 \times X + 1; \ X = Y + 3; \ (X < 10) \)
- Precondition for second statement: \( \{Y < 7\} \)
- Precondition for first statement: \( \{X < 2\} \)

\[ \{X < 2\} \quad Y = 3 \times X + 1; \ X = Y + 3; \quad \{X < 10\} \]

Denotational Semantics

- Based on recursive function theory

- The meaning of language constructs are defined by the values of the program's variables

- The process of building a denotational specification for a language:
  1. Define a mathematical object for each language entity
  2. Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
Denotational Semantics

- Decimal Numbers
  - The following denotational semantics description maps decimal numbers as strings of symbols into numeric values.
  - Syntax rule:
    \[
    \langle \text{dec\_num}\rangle \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    \| \langle \text{dec\_num}\rangle | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    \]
  - Denotational Semantics:
    \[
    M_{\text{dec}}('0') = 0, \quad M_{\text{dec}}('1') = 1, \ldots, \quad M_{\text{dec}}('9') = 9
    \]
    \[
    M_{\text{dec}}(\langle \text{dec\_num}\rangle '0') = 10 \cdot M_{\text{dec}}(\langle \text{dec\_num}\rangle)
    \]
    \[
    M_{\text{dec}}(\langle \text{dec\_num}\rangle '1') = 10 \cdot M_{\text{dec}}(\langle \text{dec\_num}\rangle) + 1
    \]
    \[
    \vdots
    \]
    \[
    M_{\text{dec}}(\langle \text{dec\_num}\rangle '9') = 10 \cdot M_{\text{dec}}(\langle \text{dec\_num}\rangle) + 9
    \]
  - Note: \( M_{\text{dec}} \) is a semantic function that maps syntactic objects to a set of non-negative decimal integer values.