# 3. Compilers and Interpreters

## Overview
- Common compiler and interpreter configurations
- Virtual machines
- Integrated programming environments
- Compiler phases
  - Lexical analysis
  - Syntax analysis
  - Semantic analysis
  - Code generation

## Compiling and Interpreting Programming Languages
- The **compiler** versus **interpreter** implementation is often fuzzy
  - One can view an interpreter as a **virtual machine**
  - A **processor (CPU)** is an implementation in hardware of a virtual machine for **machine code**
- Some languages cannot be purely compiled into machine code when the language allows a program to rewrite its own code, requiring the interpreter or virtual machine to invoke the compiler
- In general, compilers try to be as smart as possible to fix decisions that can be taken at compile time to avoid to generate code that makes a decision at run time
- Compilation leads to better performance in general
  - Allocation of variables without variable lookup at run time
  - Aggressive code optimization to exploit hardware features
- Interpretation leads to better diagnostics of a programming problem
  - Procedures can be invoked from command line by a user
  - Variable values can be inspected and modified by a user

## Compilation and Interpretation

**Compilation (conceptual):**
- Source Program \( \rightarrow \) **Compiler** \( \rightarrow \) Target Program
- Input \( \rightarrow \) Target Program \( \rightarrow \) Output

**Interpretation (conceptual):**
- Source Program \( \rightarrow \) **Interpreter** \( \rightarrow \) Output

## Pure Compilation and Linking
- Adopted by the typical Fortran implementation
- Library routines are separately linked \( \rightarrow \) (merged) with the object code of the program

- Source Program \( \rightarrow \) **Compiler** \( \rightarrow \) Incomplete Object Code
- Incomplete Object Code \( \rightarrow \) **Linker** \( \rightarrow \) Object Code
Compilation, Assembly, and Linking

- Adopted by most compilers
- Facilitates debugging of the compiler

Source Program $\xrightarrow{\text{Compiler}}$ Assembly
Assembly $\xrightarrow{\text{Assembler}}$ Incomplete Object Code
Incomplete Object Code $\xrightarrow{\text{Linker}}$ Object Code

Mixed Compilation and Interpretation

- Adopted by Pascal, Java, functional and logic languages, and most scripting languages
- Pascal compilers generate $P$-code that can be interpreted or compiled into object code
- Java compilers generate byte code that is interpreted by the Java virtual machine (or translated into machine code by a just-in-time (JIT) compiler)
- Functional and logic languages are compiled, but also allow dynamically created code to be compiled at run time for which the virtual machine invokes the compiler

Source Program $\xrightarrow{\text{Translator}}$ Intermediate Program
Intermediate Program $\xrightarrow{\text{Input}}$ Virtual Machine $\xrightarrow{\text{Output}}$

Preprocessing

- Compilers for C and C++ adopt a preprocessor
- Early C++ compilers generated intermediate C code

Source Program $\xrightarrow{\text{Preprocessor}}$ Modified Source Program
Modified Source Program $\xrightarrow{\text{Compiler}}$ Assembly

Integrated Programming Environments

- Programming tools (editors, compilers, interpreters, debuggers) function together in concert
- Trace facilities to monitor execution of the program
- Upon run time error in compiled code the editor is invoked with cursor at source line
- Fundamental to Smalltalk-80
- Java Studio, VisualStudio

Overview of Compilation

- Compilation of a program proceeds through a series of phases, where subsequent phases use information found in an earlier phase or uses a form of the program produced by an earlier phase
- Each phase may consist of a number of passes over the program representation

    Character Stream $\xrightarrow{\text{Scanner}}$
Lexical Analysis
- Lexical analysis breaks up a program (e.g. in Pascal)

```
program gcd (input, output);
var i, j : integer;
begin
  read (i, j);
  while i <> j do
    if i > j then i := i - j else j := j - i;
  writeln (i);
end.
```

into a stream of tokens

```
program gcd { input, output } ;
var i, j : integer ;
begin
read ( i, j ) ;
while i <> j do
  if i > j then i := i - j else j := j - i;
writeln ( i ) ;
end .
```

- This is also known as scanning performed by a scanner

Note: a scanner in Java

Syntax Analysis
- Parsing organizes tokens into a hierarchy called a parse tree
- A grammar of a language with the token stream defines the structure of the parse tree
- Syntax analysis is applied by a compiler to check the syntax of a program by constructing a parse tree of the program

Example (incomplete) Pascal grammar:

```
<Program> -> program <id> { <id> <More_ids> } ; <Block> .
<Block> -> <Variables> <begin> <Stmt> <More_Stmts> <end>
<More_ids> -> , <id> <More_ids>
| ε
<Variables> -> var <id> <More_ids> ; <Type> ; <More_Variables>
| ε
<More_Variables> -> <id> <More_ids> ; <Type> ; <More_Variables>
| ε
<Stmt> -> <id> := <Exp>
  [ read { <id> <More_id> } ]
  writeln { <Exp> <More_Exp> } |
  if <Exp> then <Stmt> else <Stmt>
  while <Exp> do <Stmt>
  begin <Stmt> <More_Stmts> <end
```

Note: An interactive parser demo demonstrates the parsing of the gcd Pascal example program into a parse tree (see also textbook pp. 20-21)

Context-Free Grammars
- A context-free grammar defines the syntax of a programming language
- The grammar defines syntactic categories
  - Statements
  - Expressions
  - Declarations
- Categories are subdivided into more detailed categories
  - Loop-statement
  - If-statement
  - Logical-expression
  - ...
- Most programming language manuals include language grammar:

```
<statement> -> <loop-statement>
<statement> -> <if-statement>
<loop-statement> -> for {(expression); <expression>}; <expression>
<expression> -> <logical-expression>
...```

Note: a scanner in Java

Syntax Analysis
- Parsing organizes tokens into a hierarchy called a parse tree
- A grammar of a language with the token stream defines the structure of the parse tree
- Syntax analysis is applied by a compiler to check the syntax of a program by constructing a parse tree of the program

Example (incomplete) Pascal grammar:

```
<Program> -> program <id> { <id> <More_ids> } ; <Block> .
<Block> -> <Variables> <begin> <Stmt> <More_Stmts> <end>
<More_ids> -> , <id> <More_ids>
| ε
<Variables> -> var <id> <More_ids> ; <Type> ; <More_Variables>
| ε
<More_Variables> -> <id> <More_ids> ; <Type> ; <More_Variables>
| ε
<Stmt> -> <id> := <Exp>
  [ read { <id> <More_id> } ]
  writeln { <Exp> <More_Exp> } |
  if <Exp> then <Stmt> else <Stmt>
  while <Exp> do <Stmt>
  begin <Stmt> <More_Stmts> <end
```

Note: An interactive parser demo demonstrates the parsing of the gcd Pascal example program into a parse tree (see also textbook pp. 20-21)
Semantic Analysis

- Semantic analysis is applied by a compiler to discover the meaning of a program by analyzing its parse tree or abstract syntax tree (see later)

- Static semantic checks are performed at compile time
  - Type checking
  - Every variable is declared before used
  - Identifiers are used in appropriate contexts
  - Check subroutine call arguments
  - Check labels

- Dynamic semantic checks are performed at run time, and the compiler produces code that performs these checks
  - Array subscript values are within bounds
  - Arithmetic errors, e.g. division by zero
  - Pointers are not dereferenced unless pointing to valid object
  - A variable is used but hasn’t been initialized
  - When a check fails at run time, an exception is raised

Strong Typing

- A language is strongly typed “if (type) errors are always detected”
- Such errors are listed on previous slide
- Errors are either detected at compile time or at run time
- Strong typing makes language safe and easier to use, but slower because of dynamic semantic checks
- Languages that are strongly typed are
  - Ada
  - Java
  - ML, Haskell

- Languages that are not strongly typed are
  - Fortran, Pascal, C
  - Lisp, C++, Prolog

- In some languages, most (type) errors are detected late at run time which is detrimental to reliability (e.g. early Basic, Lisp, Prolog, some script languages)

Intermediate Code Generation

- A typical intermediate form of code produced by the semantic analyzer is an abstract syntax tree (AST)
- The AST is annotated with useful information such as pointers to the symbol table entry of identifiers
- Example AST for the gcd Pascal program:

Target Code Generation and Optimization

- The AST with the annotated information is traversed by the compiler to generate a low-level intermediate form of code, close to assembly
- This machine-independent intermediate form is optimized
- From the machine-independent form assembly or object code is generated by the compiler
- This machine-specific code is optimized to exploit specific hardware features