

## Overview of Compilers and Interpreters

In this set of notes you will learn about:

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

Note: These slides cover Chapter 1 of the textbook

## Compiling and Interpreting Programming Languages

To understand the dynamics of a particular programming language better, we need to have a look at *compilation* and *interpretation*

- The *compiler* versus *interpreter* implementation is often fuzzy
  - One can view an interpreter as a *virtual machine*
  - A *processor* is an implementation in hardware of a virtual machine for *machine code* (also called object code)
- Some languages cannot be purely compiled into machine code when the language allows programs to extend themselves at run time (by writing source code), requiring the interpreter or virtual machine to call 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
  - Variable values can be inspected and modified

## Compilation and Interpretation

- *Compilation (conceptual):*

Source Program ® **Compiler** ® Target Program

Input ® **Target Program** ® Output

- *Interpretation (conceptual):*

Source Program ® **Interpreter** ® Output

Input ® **Interpreter** ® Output

## Pure Compilation and Linking

- Adopted by the typical Fortran implementation
- Library routines are separately linked (merged) with the object code

Source Program ® **Compiler** ® Incomplete Object Code

Incomplete Object Code ® **Linker** ® Object Code

Library Routines ® **Linker** ® Object Code

## Compilation, Assembly, and Linking

- Adopted by most compilers
- Facilitates debugging of the compiler

Source Program ® **Compiler** ® Assembly  
Assembly ® **Assembler** ® Incomplete Object Code  
Incomplete Object Code ® **Linker** ® Object Code  
Library Routines ® **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 object code by a just-in-time compiler)
- Functional and logic languages are compiled, but also allow newly created source to be compiled at run time and the virtual machine invokes the compiler when necessary

Source Program ® **Translator** ® Intermediate Program  
Intermediate Program ® **Virtual Machine** ® Output  
Input ® **Virtual Machine** ® Output

## Preprocessing

- Compilers for C and C++ adopt a preprocessor for macro expansion

Source Program ® **Preprocessor** ® Modified Source Program  
Modified Source Program ® **Compiler** ® Assembly

- Early C++ compilers generated intermediate C code

Source Program ® **Preprocessor** ® Modified Source Program  
Modified Source Program ® **C++ Compiler** ® C Code  
C Code ® **C 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, Visual C++

## 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

**Scanner**



## Syntax Analysis

- Parsing organizes tokens into a hierarchy called a *parse tree*
- A grammar of a language defines the structure of the parse tree
- Example (incomplete) Pascal grammar:

```

<Program> -> program <id> ( <id> <More_ids> ) ; <Block> .
<Block> -> <Variables> begin <Stmt> <More_Stmts> end
<More_ids> -> , <id> <More_ids>
                | e
<Variables> -> var <id> <More_ids> : <Type> ; <More_Variables>
                | e
<More_Variables> -> <id> <More_ids> : <Type> ; <More_Variables>
                | e
<Stmt> -> <id> := <Exp>
                | read ( <id> <More_ids> )
                | writeln ( <Exp> <More_Exps> )
                | 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

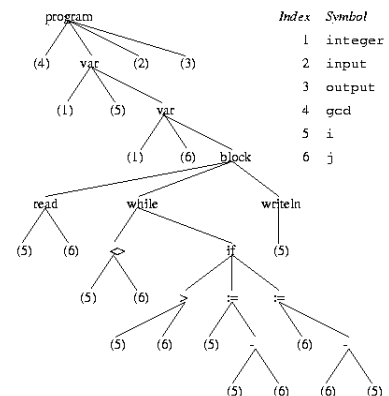
- Semantic analysis by a compiler discovers the *meaning* of a program by analyzing its parse tree (or abstract syntax tree, see later)
- *Static semantic checks* are performed at compile time
  - Type checking
  - Every variable is declared before used
  - Identifiers are used in appropriate contexts
  - 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++
- 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