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

Note: Study Chapter 1 Sections 1.4 to 1.6 of the textbook.

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) can be viewed as an implementation in hardware of a virtual machine
- Some languages cannot be purely compiled into machine code
  - Some languages allow programs to rewrite/add code
- 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 ® Compiler ® Target Program
  - Input ® Target Program ® Output
- Interpretation (conceptual):
  - Source Program ® Interpreter ® Output

Pure Compilation and Linking
- Adopted by the typical Fortran implementation
- Library routines are separately linked (merged) with the object code of the program
  - Source Program ® Compiler ® Incomplete Object Code
  - Incomplete Object Code ® 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

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 ® Translator® Intermediate Program
Intermediate Program ® Virtual Machine® Output

Preprocessing

- Compilers for C and C++ adopt a preprocessor

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 (IDEs)

- Programming tools (editors, compilers/interpreters, debuggers, preprocessors, assemblers, linkers) function together in concert
- Editors can help formatting and cross referencing
- 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, Borland

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
Lexical Analysis

- Lexical analysis breaks up a program (e.g. in Pascal)

```plaintext
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

```plaintext
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
- A lexical error is produced when an unrecognized character is encountered

Note: Download a scanner application in Java

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
  - ...
- Some programming language manuals include language grammar

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
- A syntax error is produced by a compiler when the parse tree cannot be constructed for a program (fragment)
- Example (incomplete) Pascal grammar:

```plaintext
program
  var id : Type ;
  begin
    <stmt> -> <if-statement> <else-statement> <for-statement> <while-statement> <expression> <logical-expression> ...
```

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++
- 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
<table>
<thead>
<tr>
<th>Exercise 1</th>
<th>Exercise 2</th>
<th>Exercise 3</th>
<th>Exercise 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name two languages in which a program can rewrite new pieces of itself. Hint: which languages are said to be suitable for symbolic and logic processing?</td>
<td>Which IDEs do you regularly use? If not, explain the tools you use for programming projects.</td>
<td>Describe the six tools that are commonly used with a compiler within an IDE.</td>
<td>Using your favorite compiled programming language, give an example of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- a lexical error detected by the scanner</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- a syntax error detected by the parser</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- a static semantic error detected by semantic analysis</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- a dynamic semantic error detected at runtime by the code generated by the compiler</td>
</tr>
</tbody>
</table>