What is a Compiler?

- Is a program that translates one language to another
- Takes as input a source program typically written in a high-level language
- Produces an equivalent target program typically in assembly or machine language
- Reports error messages as part of the translation process

First computers of late 1940s were programmed in machine language
- Machine language was soon replaced by assembly language
  - Instructions and memory locations are given symbolic names
  - An assembler translates the symbolic assembly code into equivalent machine code
  - Assembly language improved programming, but is still machine dependent
Brief History

- The term “compiler” was coined in the early 1950s by Grace Murray Hopper
  - Translation was then viewed as the “compilation” of a sequence of routines selected from a library
- The first compiler of the high-level language FORTRAN was developed between 1954 and 1957 at IBM by a group led by John Backus
  - Proved the viability of high-level and thus less machine dependent languages
- The study of the scanning and parsing problems were pursued in the 1960s and 1970s and led fairly to a complete solution
  - Became standard part of compiler theory
  - Resulted in scanner and parser generators that automate part of compiler development
- The development of methods for generating efficient target code, known as optimization techniques, is still an ongoing research
- Compiler technology was also applied in rather unexpected areas:
  - Text-formatting languages
  - Hardware description languages for the automatic creation of VLSI circuits
The Translation Process

- A compiler performs two major tasks:
  - Analysis of the source program
  - Synthesis of the target-language instructions

- Phases of a compiler:
  - Scanning
  - Parsing
  - Semantic Analysis
  - Intermediate Code Generation
  - Intermediate Code Optimizer
  - Target Code Generator
  - Target Code Optimizer
The Translation Process – Cont'd

- Three auxiliary components interact with some or all phases:
  - Literal Table
  - Symbol Table
  - Error Handler

```
<table>
<thead>
<tr>
<th>Source Code</th>
<th>Intermediate Code Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokens</td>
<td>Intermediate Code Optimizer</td>
</tr>
<tr>
<td>Syntax Tree</td>
<td>Target Code Generator</td>
</tr>
<tr>
<td>Annotated Tree</td>
<td>Target Code Optimizer</td>
</tr>
</tbody>
</table>
```

```
| Literal Table |
| Symbol Table  |
| Error Handler |
```
Scanner

- The scanner begins the analysis of the source program by:
  - Reading file character by character
  - Grouping characters into tokens
  - Eliminating unneeded information (comments and white space)
  - Entering preliminary information into literal or symbol tables
  - Processing compiler directives by setting flags

- Tokens represent basic program entities such as:
  - Identifiers, Literals, Reserved Words, Operators, Delimiters, etc.

- Example: \( a := x + y \ast 2.5 \); is scanned as
  - \( a \) identifier
  - \(:=\) assignment operator
  - \( x \) identifier
  - \( y \) identifier
  - \( \ast \) multiplication operator
  - \( 2.5 \) real literal
  - \(+\) plus operator
  - \( ;\) semicolon
Parser

- Receives tokens from the scanner
- Recognizes the structure of the program as a **parse tree**
  - Parse tree is recognized according to a context-free grammar
  - Syntax errors are reported if the program is syntactically incorrect
- A parse tree is inefficient to represent the structure of a program
- A **syntax tree** is a more condensed version of the parse tree
- A syntax tree is usually generated as output by the parser

```
a := x + y * 2.5 ;
```

**Parse Tree**
```
assign-stmt

  id a := expr ;

  expr + expr

  id x expr * expr

  id id y literal 2.5
```

**Syntax Tree**
```
:=

  id a +

  id x *

  id y literal 2.5
```
Semantic Analyzer

- The semantics of a program are its **meaning** as opposed to syntax or structure.
- The semantics consist of:
  - **Runtime semantics** – behavior of program at runtime
  - **Static semantics** – checked by the compiler
- Static semantics include:
  - Declarations of variables and constants before use
  - Calling functions that exist (predefined in a library or defined by the user)
  - Passing parameters properly
  - Type checking.
- Static semantics are difficult to check by the parser.
- The semantic analyzer does the following:
  - Checks the static semantics of the language
  - Annotates the syntax tree with type information
Intermediate Code Generator

- Comes after syntax and semantic analysis
- Separates the compiler front end from its backend
- Intermediate representation should have 2 important properties:
  - Should be easy to produce
  - Should be easy to translate into the target program
- Intermediate representation can have a variety of forms:
  - Three-address code, P-code for an abstract machine, Tree or DAG representation

![Annotated Syntax Tree]

Three-address code

temp1 := int2real(y)
temp2 := temp1 \ real\* 2.5
temp3 := x \ real\+ temp2
a := temp3
### Code Generator

- Generates code for the target machine, typically:
  - Assembly code, or
  - Relocatable machine code

- Properties of the target machine become a major factor
- Code generator selects appropriate machine instructions
- Allocates memory locations for variables
- Allocates registers for intermediate computations

#### Three-address code

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>temp1 := int2real(y)</code></td>
<td>R1 ← y</td>
</tr>
<tr>
<td><code>temp2 := temp1 * 2.5</code></td>
<td>F1 ← int2real(R1)</td>
</tr>
<tr>
<td><code>temp3 := x + temp2</code></td>
<td>F2 ← F1 * 2.5</td>
</tr>
<tr>
<td><code>a := temp3</code></td>
<td>F3 ← x</td>
</tr>
<tr>
<td><code>a := temp3</code></td>
<td>F4 ← F3 + F2</td>
</tr>
<tr>
<td><code>STORF a, F4</code></td>
<td>a ← F4</td>
</tr>
</tbody>
</table>

#### Assembly code (Hypothetical)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOADI R1, y</td>
<td>R1 ← y</td>
</tr>
<tr>
<td>MOVF F1, R1</td>
<td>F1 ← int2real(R1)</td>
</tr>
<tr>
<td>MULF F2, F1, 2.5</td>
<td>F2 ← F1 * 2.5</td>
</tr>
<tr>
<td>LOADF F3, x</td>
<td>F3 ← x</td>
</tr>
<tr>
<td>ADDF F4, F3, F2</td>
<td>F4 ← F3 + F2</td>
</tr>
<tr>
<td>STORF a, F4</td>
<td>a ← F4</td>
</tr>
</tbody>
</table>
Code Improvement

- Code improvement techniques can be applied to:
  - Intermediate code – independent of the target machine
  - Target code – dependent on the target machine

- Intermediate code improvement include:
  - Constant folding
  - Elimination of common sub-expressions
  - Identification and elimination of unreachable code (called dead code)
  - Improving loops
  - Improving function calls

- Target code improvement include:
  - Allocation and use of registers
  - Selection of better (faster) instructions and addressing modes
Interpreter

- Is a program that reads a source program and executes it
- Works by analyzing and executing the source program commands *one at a time*
- Does not translate the source program into object code
- Interpretation is sensible when:
  - Programmer is working in interactive mode and needs to view and update variables
  - Running speed is not important
  - Commands have simple formats, and thus can be quickly analyzed and executed
  - Modification or addition to user programs is required as execution proceeds
- Well-known examples of interpreters:
  - Basic interpreter, Lisp interpreter, UNIX shell command interpreter, SQL interpreter
- In principle, any programming language can be either interpreted or compiled
  - Some languages are designed to be interpreted, others are designed to be compiled
- Interpreters involve large overheads
  - Execution speed degradation can vary from 10:1 to 100:1
  - Substantial space overhead may be involved
Programs Related to Compilers

- **Preprocessor**
  - Produces input to a compiler
  - Performs the following:
    - Macro processing (substitutions)
    - File inclusion

- **Assembler**
  - Translator for the assembly language
  - Two-Pass Assembly:
    - All variables are allocated storage locations
    - Assembler code is translated into machine code
  - Output is *relocatable machine code*.

- **Linkers**
  - Links object files separately compiled or assembled
  - Links object files to standard library functions
  - Generates a file that can be loaded and executed

- **Debuggers**

- **Editors**
Major Data and Structures in a Compiler

- **Token**
  - Represented by an integer value or an enumeration literal
  - Sometimes, it is necessary to preserve the string of characters that was scanned
    - For example, name of an identifiers or value of a literal

- **Syntax Tree**
  - Constructed as a pointer-based structure
  - Dynamically allocated as parsing proceeds
  - Nodes have fields containing information collected by the parser and semantic analyzer

- **Symbol Table**
  - Keeps information associated with all kinds of identifiers:
    - Constants, variables, functions, parameters, types, fields, etc.
  - Identifiers are entered by the scanner, parser, or semantic analyzer
  - Semantic analyzer adds type information and other attributes
  - Code generation and optimization phases use the information in the symbol table
  - Insertion, deletion, and search operations need to efficient because they are frequent
  - Hash table with constant-time operations is usually the preferred choice
  - More than one symbol table may be used
Major Data and Structures in a Compiler

❖ Literal Table
  ★ Stores constant values and string literals in a program.
  ★ One literal table applies globally to the entire program.
  ★ Used by the code generator to:
    ♦ Assign addresses for literals.
    ♦ Enter data definitions in the target code file.
  ★ Avoids the replication of constants and strings.
  ★ Quick insertion and lookup are essential. Deletion is not necessary.

❖ Temporary Files
  ★ Used historically by old compilers due to memory constraints
  ★ Hold the data of various stages