CSE 504 Compiler Design Overview

YoungMin Kwon

Course Objective

- Learn how compilers are designed and implemented
  - How to write grammars
  - How to parse and translate grammars
  - Theory behind them
- Learn details of programming languages
  - How programming language elements are implemented
  - Runtime environments
  - x86 assembly language
- Get used to useful tools and improve development skills
  - Lexical scanner
  - Parser generator
  - Debugging skills…

Course Materials

- Textbook:
  - “Compilers Principles, Techniques, and Tools” 2nd edition by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman
- Lexical scanner and Parser generator tools:
  - “lex & yacc” by John R. Levine, Tony Mason, and Doug Brown

Course Organization

- Learn overall compiling steps using the tools
  - Build a simple compiler for an abstract stack machine
  - Get used to Lex and Yacc tools (Lexical scanner and Parser generator)
- Lexical analysis
  - Regular expressions
  - Nondeterministic Finite Automata (NFA), Deterministic Finite Automata (DFA)
- Parsing
  - Context-free grammars
  - Top-Down parsing, Bottom-Up parsing
- Semantic analysis
  - Syntax directed translation
  - Type checking

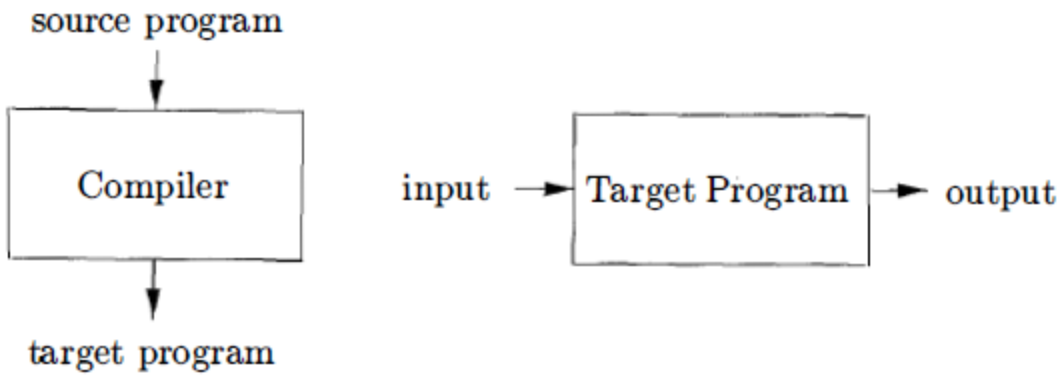
Course Organization (continued)

- x86 assembly code generation (without optimization)
  - Runtime environment
  - Get used to x86 assembly language
  - Translation to x86 assembly language
- Intermediate code generation
- Code generation
  - Register allocation and assignment
- Code optimization
  - Global code optimization

Language Processors

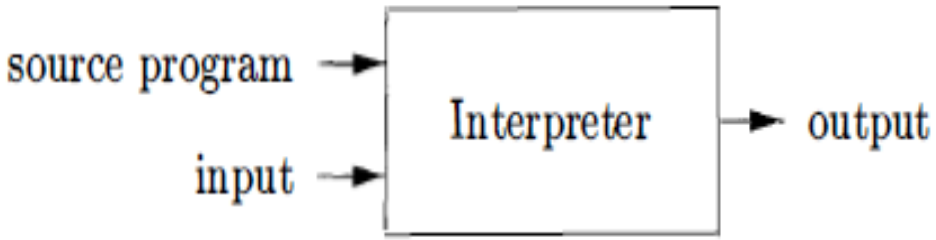
- Compiler: a program that reads a program in one language (the source language) and translates it into an equivalent program in another language (the target language).

- The target program is a self-sufficient program that can handle user's input and produce output.



Language Processors

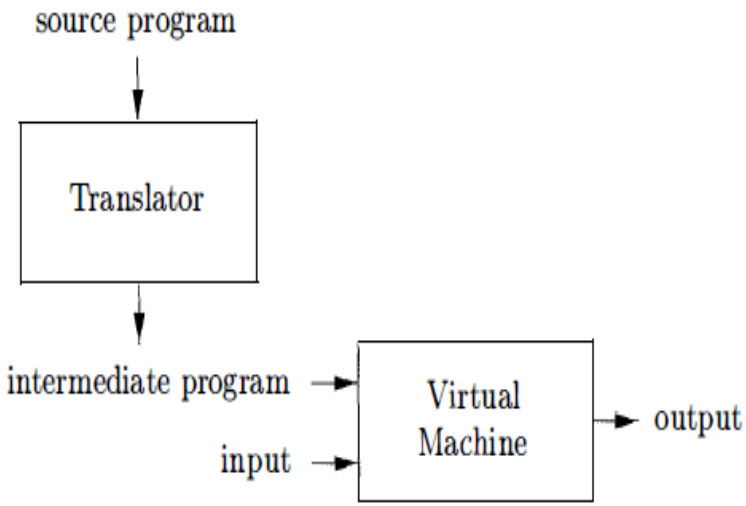
- Interpreter: without producing a target program, an interpreter executes the source program on user's input.



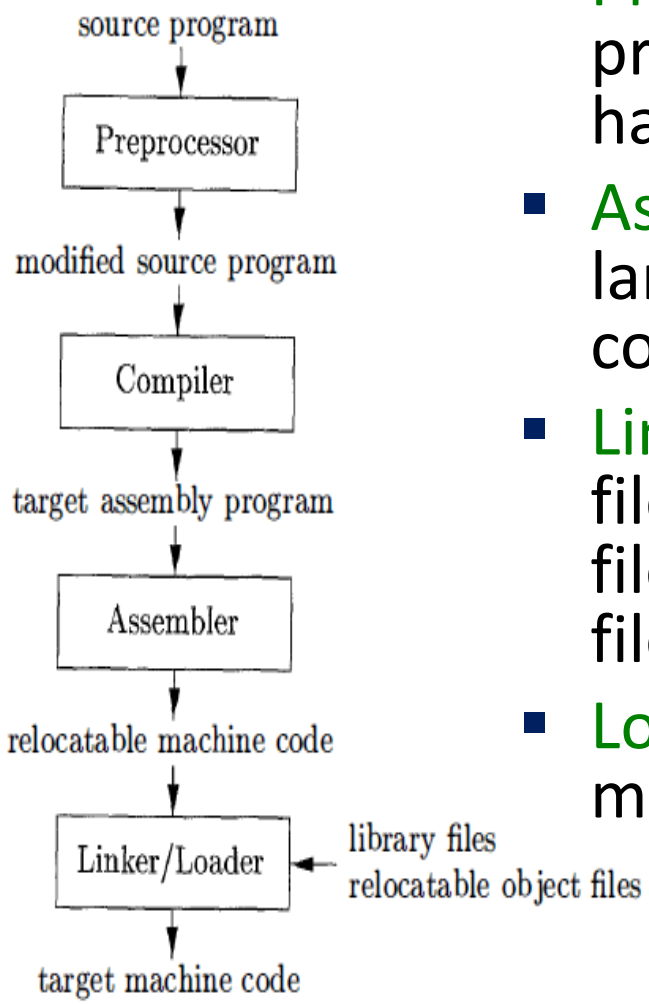
Language Processors

- Hybrid model: Java source code is compiled bytecodes and the bytecodes are interpreted by a virtual machine

- Just-In-Time (JIT) compilers: translate the bytecodes into the machine language immediately before they run the intermediate program



Language Processors



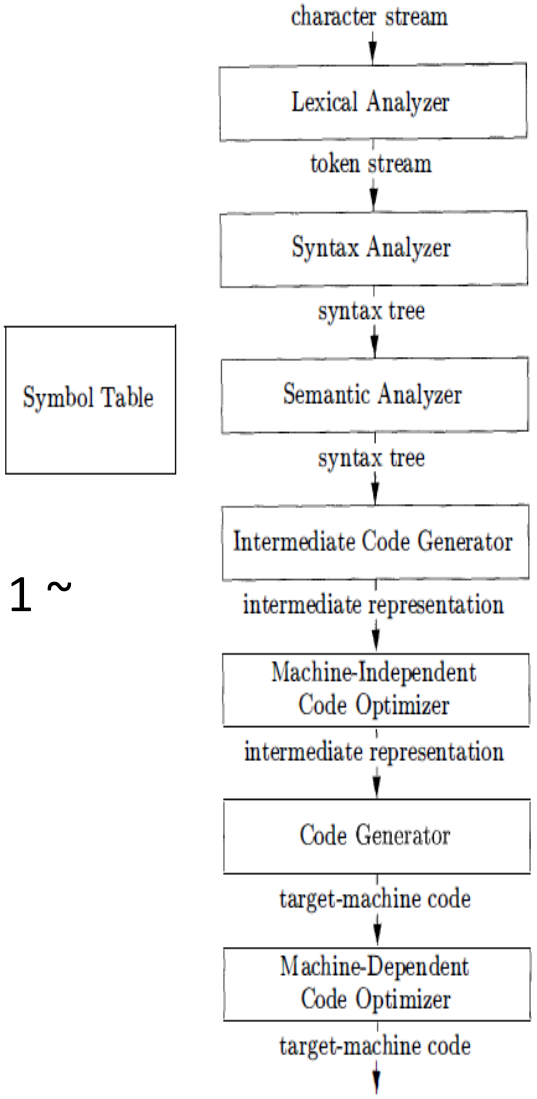
- Preprocessor: collecting the source program stored in separate files, handling macros.
- Assembler: translate assembly language to a relocatable machine code.
- Linker: combines relocatable object files and libraries so that a code in one file can refer to a location in another file.
- Loader: loads all executable files into memory for execution.

The Structure of a Compiler

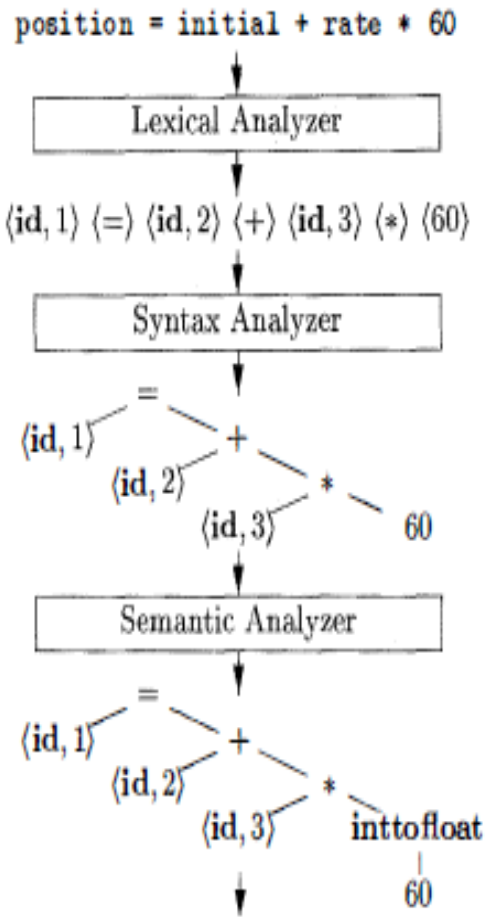
- Front end (analysis part)
  - Breaks up source program into pieces
  - Imposes grammatical structure on them
  - Syntactic and Semantic checking
  - Produces intermediate representation of the source program
- Back end (synthesis part)
  - Optimizes the intermediate representation
  - Produces the target program

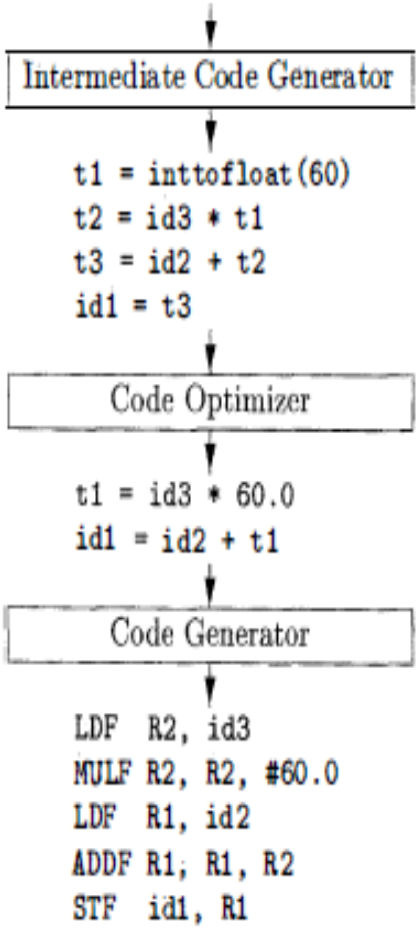
Phases of a Compiler

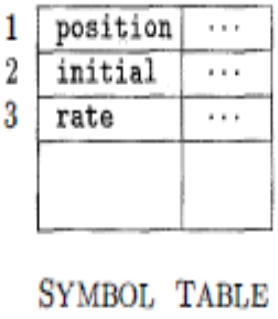
- Lexical analysis
  - source text -> tokens
- Syntax analysis
  - tokes -> parse tree
- Semantic analysis
  - parse tree -> syntax tree (type checking)
- Intermediate code generation
  - syntax tree -> machine independent code
  - e.g. three address code: 1 target address and 1 ~ 2 source addresses; at most 1 operator for 1 instruction
- Code optimization
  - optimization (fast, short, less power) on the intermediate code
- Code generation
  - intermediate code -> target machine code



Phases of a Compiler (continued)







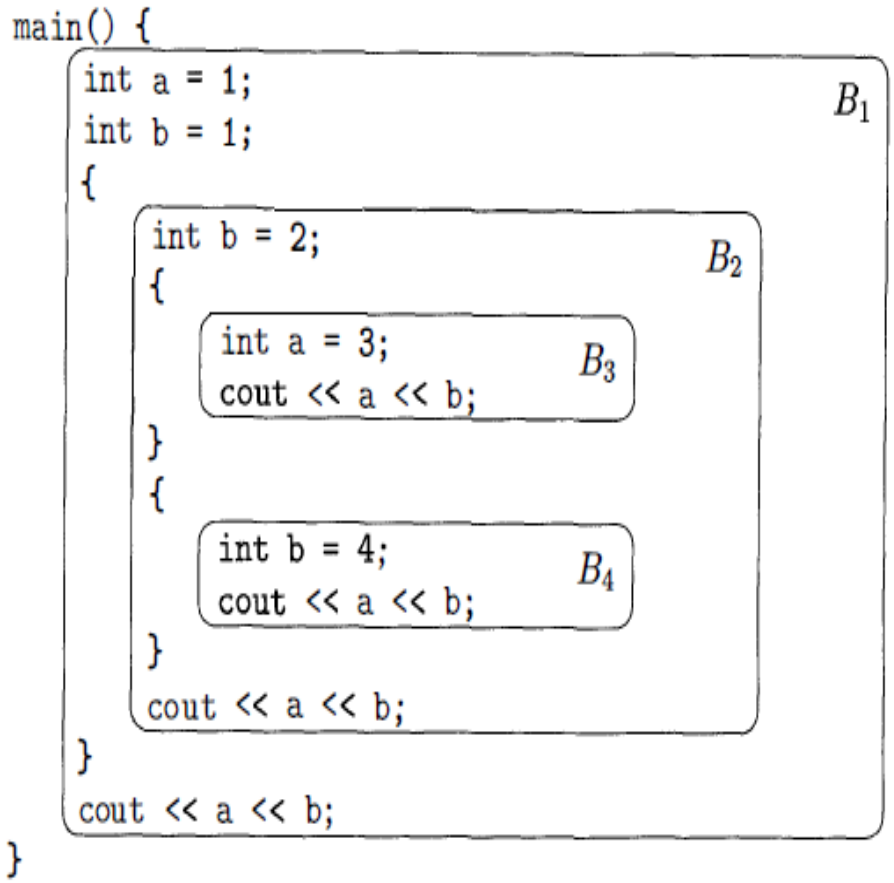
Grouping Phases into passes

- In an implementation, several phases are grouped into passes
  - e.g.) front-end phases are grouped into one pass, optimization is left as an optional pass, and the code generation makes another pass.
    - Lexical analysis, Syntax analysis, Semantic analysis, Intermediate code generation
    - Code optimization
    - Code generation
- Some compilers have several front-ends and one back-end to handle multiple programming languages for a single target machine
- Some compilers have a front-end and multiple back-ends to handle multiple different target machines.

Programming Language Basics

- Static policy: allows the compiler to decide an issue.
- Dynamic policy: allows the decision to be made on runtime.
- e.g.) Scoping rule
  - Scope of a declaration x is the region of the program where x refers to this declaration.
  - Static scope or lexical scope: if it is possible to determine the scope of a declaration by looking only at the program.
  - Dynamic scope: the same x can refer to different declarations of x as the program runs

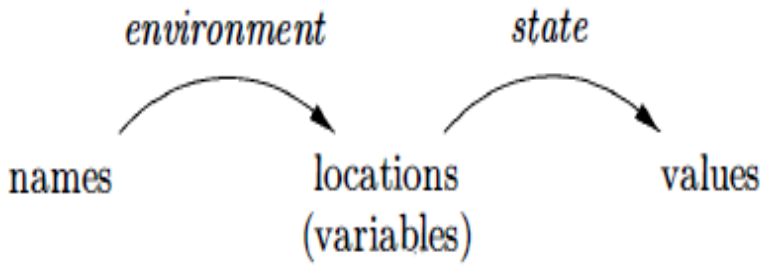
Static Scoping and Block Structure



DECLARATION	SCOPE
int a = 1;	$B_1 - B_3$
int b = 1;	$B_1 - B_2$
int b = 2;	$B_2 - B_4$
int a = 3;	B_3
int b = 4;	B_4

Environments and States

- Environment: mapping from names to locations in the store (l-values)
- State: mapping from locations in store to their values (mapping from l-values to their r-values)



Parameter Passing Mechanism

- Actual parameter: the parameters used in the call of a procedure.
- Formal parameter: the parameters in the procedure definition.
- Call-by-Value: actual parameter is evaluated and placed in the location corresponding to the formal parameter of the callee.
- Call-by-Reference: the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.
- Call-by-Name: callee runs as if the actual parameters were substituted with the formal parameters in the code.
- In C, C++, Java, the basic type parameters are passed by Call-by-Value mechanism. Composition types like objects and arrays are passed by their addresses (Structures in C are passed by value).