CSE 304/504 Compiler Design Overview

YOUNGMIN KWON / TONY MIONE

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
- Symbol management
- Runtime environments
- MIPS & Intel assembly languages

Become familiar with useful tools and improve development skills

- Lexical scanner
- Parser generators
- Debugging skills…

Course Materials

Textbook:

- "Compilers Principles, Techniques, and Tools" 2nd edition by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman

Recommended Texts:

Lexical scanner and Parser generator tools:

- "lex & yacc" by John R. Levine, Tony Mason, and Doug Brown

Additional Compiler Information:

- Engineering a Compiler", Cooper, Keith, D., Torczon, Linda, Elsevier, 2012, ISBN 978-0-12-088478-0.

Course Organization

Learn overall compiling steps using the tools

- Build a simple compiler for an abstract stack machine
- Become familiar with Lex and Yacc tools (Lexical scanner and Parser generator)

Lexical analysis

- Regular expressions
- Nondeterministic Finite Automata (NFA), Deterministic Finite Automata (DFA)

Symbol Management

- Scope
- Lifetime

Parsing

- Context-free grammars
- Top-Down parsing, Bottom-Up parsing

Course Organization (continued)

Semantic analysis
- Syntax directed translation
- Type checking

MIPS assembly code generation (without optimization)
- Runtime environment
- MIPS assembly language
- Translation to MIPS assembly language

Intermediate code generation

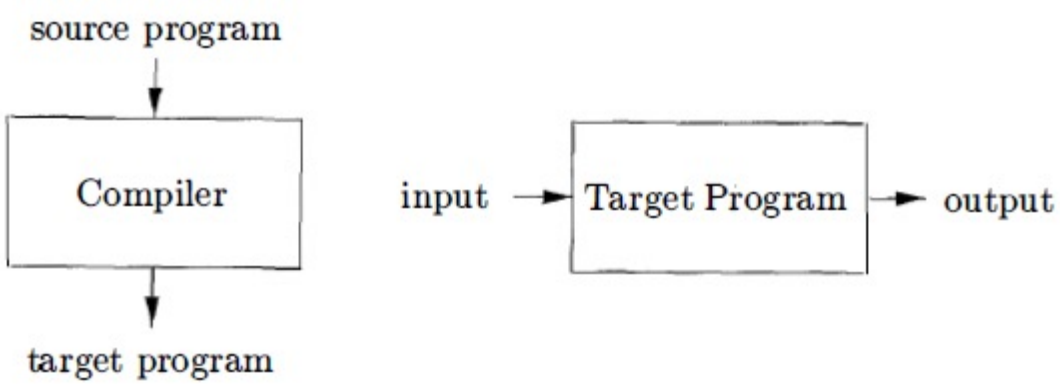
Code generation
- Register allocation and assignment

Code optimization
- Global code optimization

Language Processors

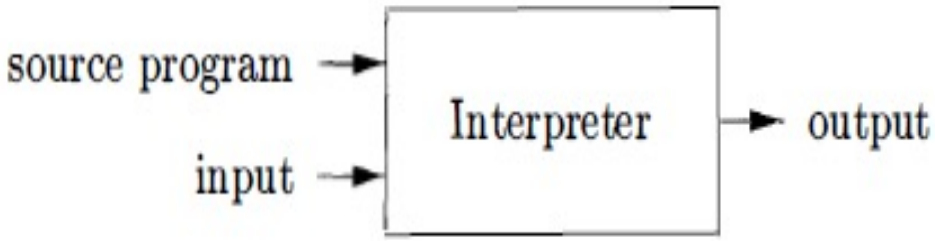
Compiler: a program that reads a program in one language (the source language) and translates it into an equivalent program in another language (the target language).

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



Language Processors

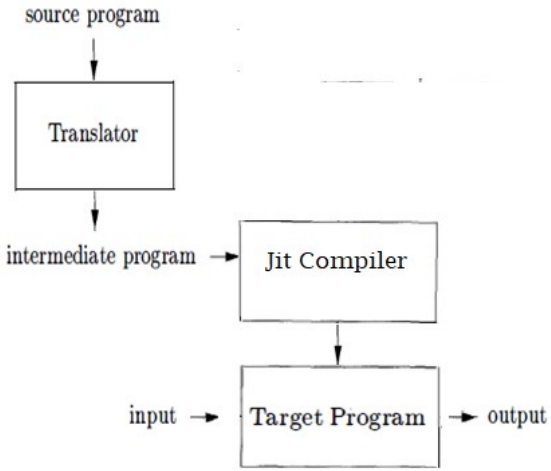
Interpreter: without producing a target program, an interpreter executes the source program on user's input.



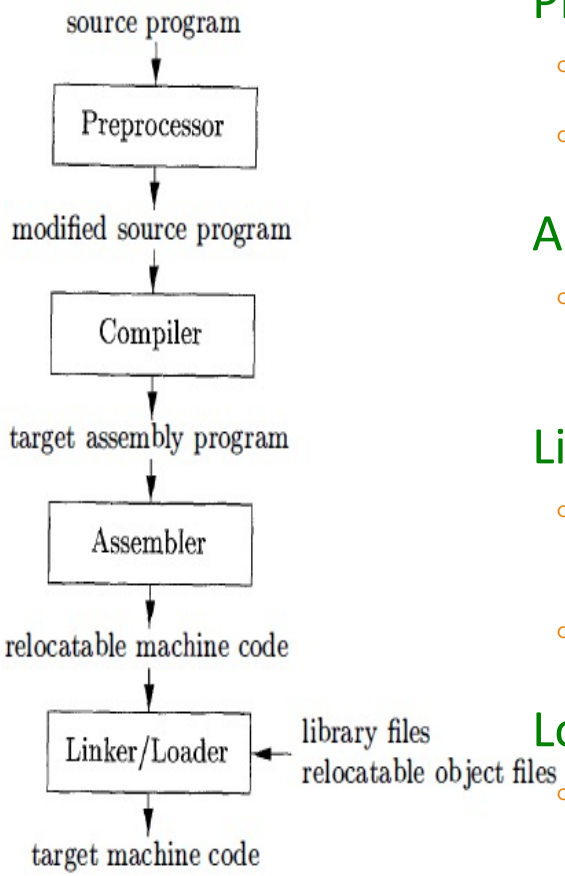
Language Processors

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

In newer Java implementations (last decade or so) Just-In-Time (JIT) compiler is used to translate the bytecodes into native machine language immediately before they run the program. The native code can be run directly on the processor and is much faster than interpreting bytecodes.



Language Processors



Preprocessor (some languages):

- Collects the source code stored in separate files
- Handles macro 'expansion'.

Assembler:

- Translates assembly language to a relocatable machine code.

Linker:

- Combines relocatable object files and libraries so that code in one file can refer to locations in another file.
- Builds a single executable with all addresses resolved.

Loader:

- Loads all executable files into memory for execution.

The Structure of a Compiler

Front end (analysis part)

- Breaks up source program into pieces **[Lexical analyzer]**
- Imposes grammatical structure on them **[Parser]**
- Syntactic and Semantic checking **[Semantic Analyzer]**
- Produces intermediate representation of the source program **[Intermediate code generator]**

Back end (synthesis part)

- Optimizes the intermediate representation **[Machine independent code optimizer]**
- Produces the target program **[Code Generator / Optimizer]**

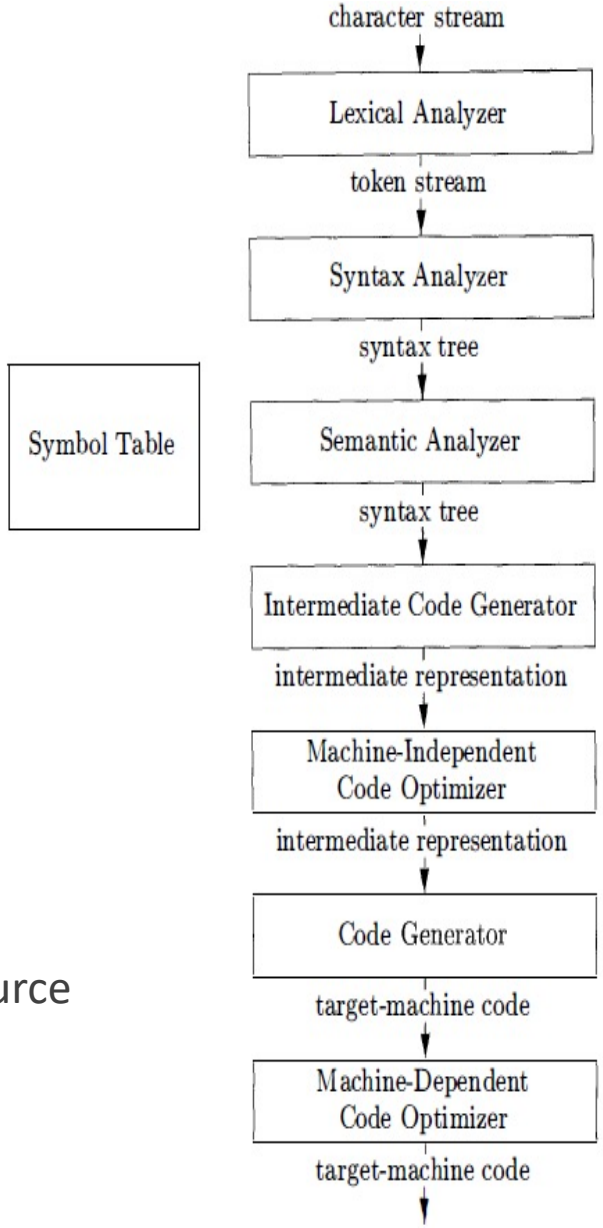
Phases of a Compiler

Lexical analysis
- source text -> tokens

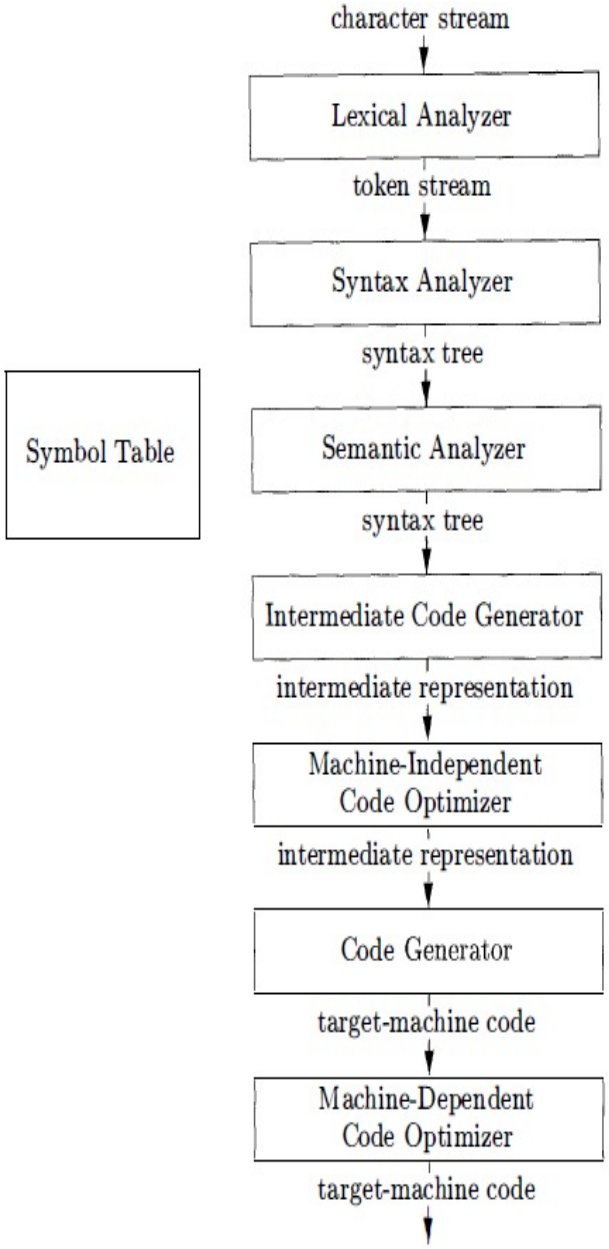
Syntax analysis
- tokens -> parse tree

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



Phases of a Compiler (continued)



Code optimization (Machine-independent)

- optimization (fast, short, less power) on the intermediate code
- General mathematical transformations

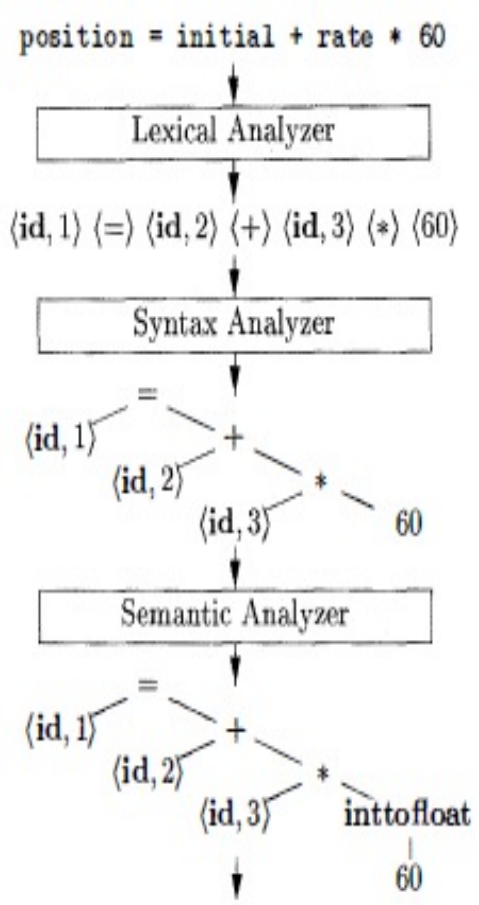
Code generation

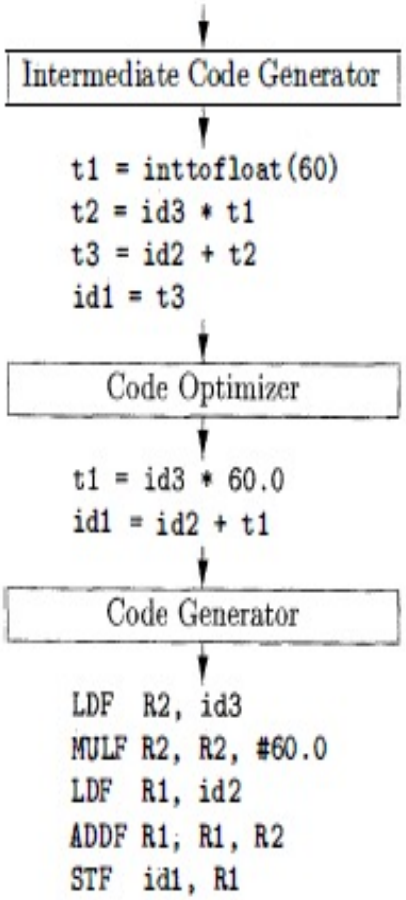
- intermediate code -> target machine code

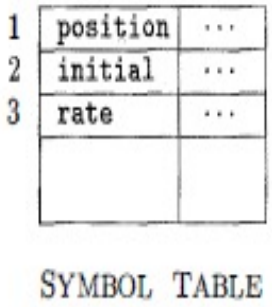
Code optimization (Machine-dependent)

- Leverages knowledge of CPU idiosynchrocies to generate more efficient code.

Phases of a Compiler (continued)







Grouping Phases into passes

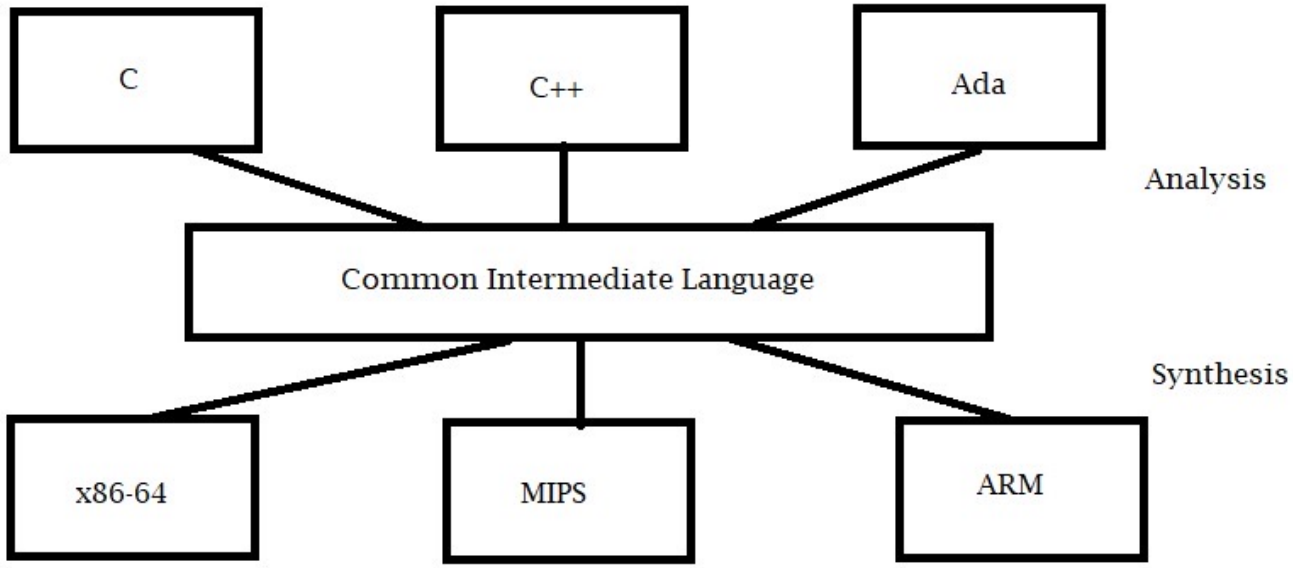
In an implementation, several phases are grouped into passes

- 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.

Modern Compile System Architecture



- Instead of building 9 compilers (3 languages for each of 3 architectures)
  - Build 3 front ends and 3 back ends.
  - To add a new language, just build a single front end!
  - To support all languages on a new CPU, build a single back end!

Programming Language Basics

Static policy: allows the compiler to decide an issue.

Dynamic policy: allows the decision to be made at 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

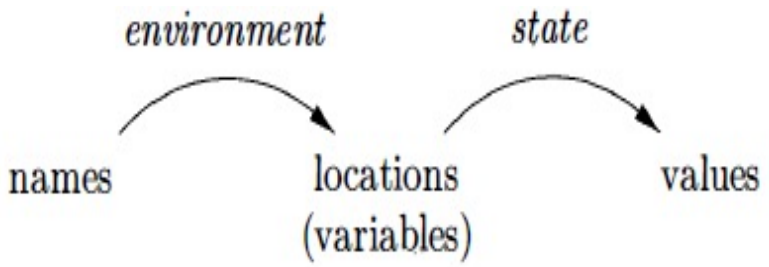
```
main() {
    int a = 1;                          B1
    int b = 1;
    {
        int b = 2;                      B2
        {
            int a = 3;                  B3
            cout << a << b;
        }
        {
            int b = 4;                  B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

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 r-values)



Parameter Passing Mechanism

Actual parameter (arguments): 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 for 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 (but… **Structures in C are passed by value**).

Questions?