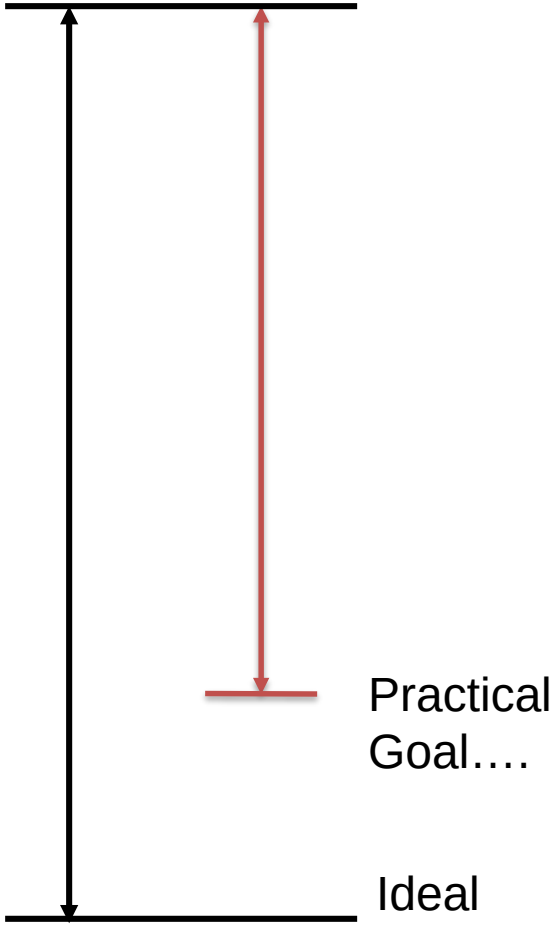
Ch 1 Compiler Overview

Yongjun Park
Hanyang University

Course Schedule (subject to change)

Week	Subject	Reading (Aho)	Exams
1	Introduction	Ch 1	
2	Lexical analysis	Ch 2-3	
3	Syntax analysis	Ch 4-5	
4	Syntax analysis	Ch 4-5	
5	Syntax, Semantic analysis	Ch 4-5 Ch 6	
6	Semantic analysis	Ch 6	
7	Semantic analysis	Ch 6	
8	Exam review, Exam 1		E1
9	Intermediate code	Ch 7-8	
10	Control flow	Ch 9 –10	
11	Dataflow and Opti	Ch 9-10	
12	Dataflow and Opti	Ch 9-10	
13	Code generation	Ch 9-10	
14	Code gen, Exam review	Ch 9-10	
15	Advanced topics		
16	Exam2		E2



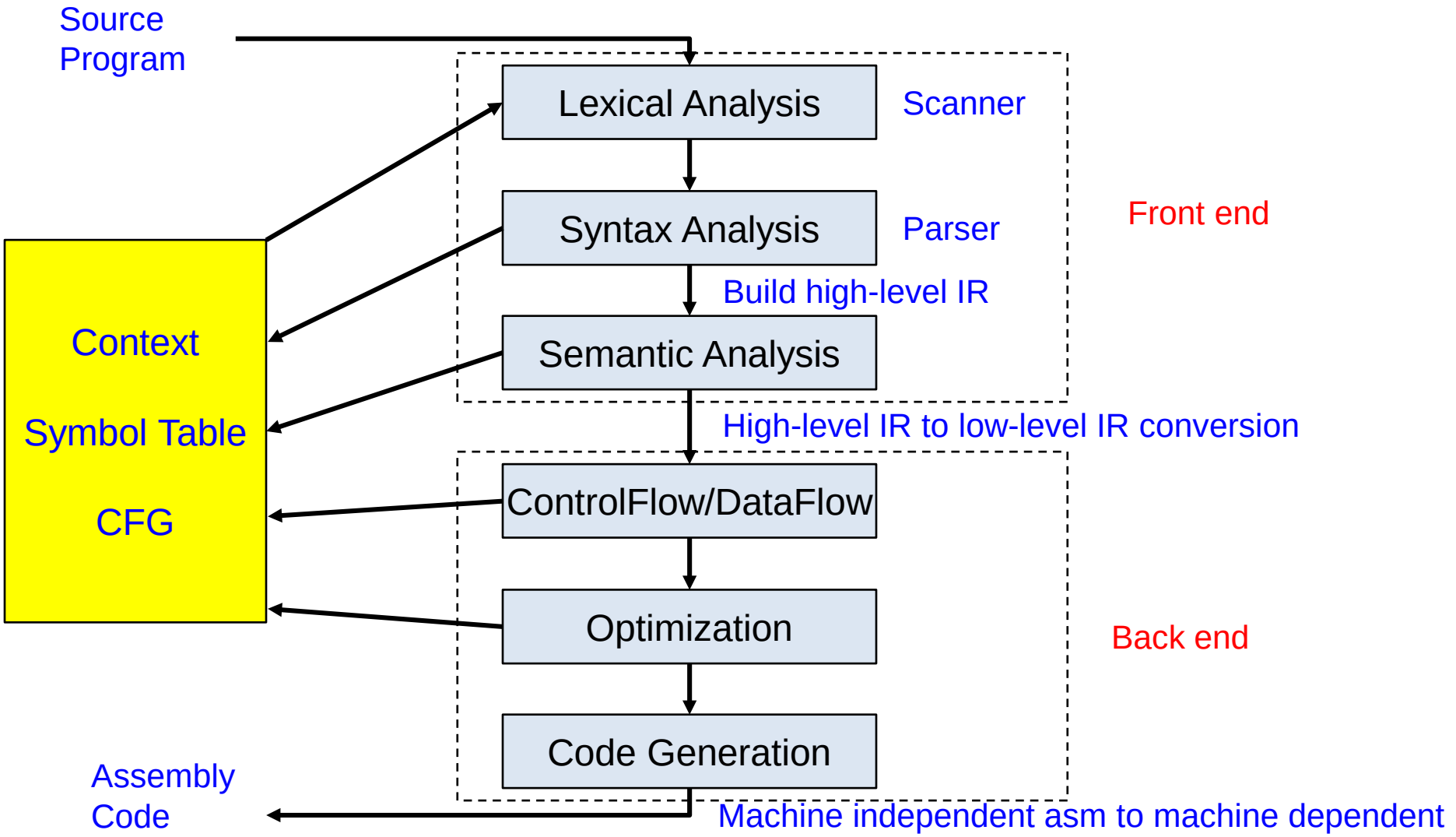
Why Compilers?

- **Compiler**
  - A program that translates from 1 language to another
  - It must preserve semantics of the source
  - It should create an efficient version of the target language
- **In the beginning, there was machine language**
  - Ugly – writing code, debugging
  - Then came textual assembly – still used on DSPs
  - High-level languages – Fortran, Pascal, C, C++
  - Machine structures became too complex and software management too difficult to continue with low-level languages

Compiler Structure

- **Source language**
  - Fortran, Pascal, C, C++
  - Verilog, VHDL, Tex, Html
- **Target language**
  - Machine code, assembly
  - High-level languages, simply actions
- **Compile time vs run time**
  - Compile time or statically – Positioning of variables
  - Run time or dynamically – SP values, heap allocation

General Structure of a Modern Compiler



Lexical Analysis (Scanner)

- **Extracts and identifies lowest level lexical elements from a source stream**
  - Reserved words: for, if, switch
  - Identifiers: “i”, “j”, “table”
  - Constants: 3.14159, 17, “%d\n”
  - Punctuation symbols: “(“,”)”, “,”, “+”
- **Removes non-grammatical elements from the stream – i.e. spaces, comments**
- **Implemented with a Finite State Automata (FSA)**
  - Set of states – partial inputs
  - Transition functions to move between states

Lex/Flex

- **Automatic generation of scanners**
  - Hand-coded ones are faster
  - But tedious to write, and error prone!

- **Lex/Flex**
  - Given a specification of regular expressions
  - Generate a table driven FSA
  - Output is a C program that you compile to produce your scanner

Parser

- **Check input stream for syntactic correctness**
  - Framework for subsequent semantic processing
  - Implemented as a push down automaton (PDA)
- **Lots of variations**
  - Hand coded, recursive descent?
  - Table driven (top-down or bottom-up)
  - For any non-trivial language, writing a correct parser is a challenge
- **Yacc (yet another compiler compiler)/bison**
  - Given a context free grammar
  - Generate a parser for that language (again a C program)

Static Semantic Analysis

- **Several distinct actions to perform**
  - Check definition of identifiers, ascertain that the usage is correct
  - Disambiguate overloaded operators
  - Translate from source to IR (intermediate representation)
- **Standard formalism used to define the application of semantic rules is the Attribute Grammar (AG)**
  - Graph that provides for the migration of information around the parse tree
  - Functions to apply to each node in the tree

Backend

- **Frontend -**
  - Statements, loops, etc.
  - These broken down into multiple assembly statements
- **Machine independent assembly code**
  - 3-address code, RTL
  - Infinite virtual registers, infinite resources
  - “Standard” opcode repertoire
    - Load/store architecture
- **Goals**
  - Optimize code quality
  - Map application to real hardware

Dataflow and Control Flow Analysis

- **Provide the necessary information about variable usage and execution behavior to determine when a transformation is legal/illegal**
- **Dataflow analysis**
  - Identify when variables contain “interesting” values
  - Which instructions created values or consume values
  - DEF, USE, GEN, KILL
- **Control flow analysis**
  - Execution behavior caused by control statements
  - If’s, for/while loops, goto’s
  - Control flow graph

Optimization (I like this!)

- **How to make the code go faster**
- **Classical optimizations**
  - Dead code elimination – remove useless code
  - Common subexpression elimination – remove recomputing the same thing multiple times
- **Machine independent (classical)**
  - Focus of this class
  - Useful for almost all architectures
- **Machine dependent**
  - Depends on processor architecture
  - Memory system, branches, dependences

Code Generation

- **Mapping machine independent assembly code to the target architecture**
- **Virtual to physical binding**
  - Instruction selection – best machine opcodes to implement generic opcodes
  - Register allocation - infinite virtual registers to N physical registers
  - Scheduling – binding to resources (i.e. adder1)
  - Assembly emission
- **Machine assembly is our output, assembler, linker take over to create binary**

Why are Compilers Important?

- **Computer architecture**
  - Build processors that software can be automatically mapped to efficiently
  - Exploiting hardware features
- **CAD tools**
  - Behavioral synthesis / C-to-gates tools are hardware compilers
  - Use program analysis/optimization to generate cheaper hardware
- **Software developers**
  - How do I create a compiler?
  - How does it map my code to the hardware