Compiler
– 1. Compiler Overview –

Jieung Kim
jieungkim@yonsei.ac.kr

YONSEI UNIVERSITY

Contents

- Compiler definitions
- Structure of compilers
- Compiler construction tools
- Programs in compilers
- Applications of compiler techniques
- Programming language theories and compilers

Intro

- Execute this

```
position = …
initial = …
rate = …

…

position = initial + rate * 60
```

- How?

Intro

- Computing

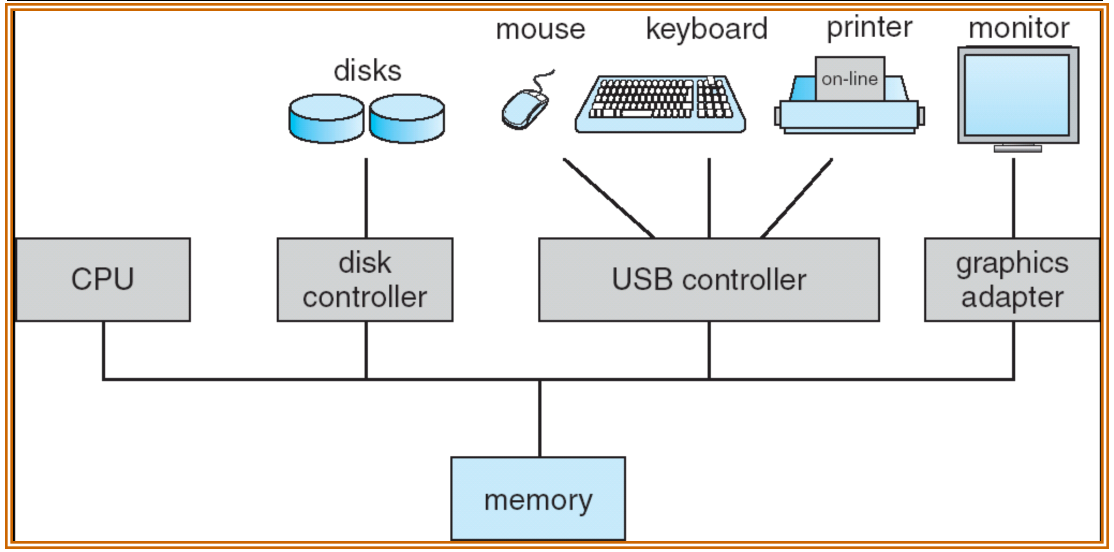
Written in HLL

SW app ………… SW app

Translate SW into Executables

OS as a resource Manager

Resources



Interpreters & compilers

- Interpreter
  - A program that reads a source program and produces the results of executing that program.
- Compiler
  - A program that translates a program from one language (the source) to another (the target.)

YONSEI UNIVERSITY

Common issues

- Common issues
  - Compilers and interpreters both must read the input – a stream of characters – and "understand" it; *analysis.*

```
position = …
initial = …
rate = …

…

position = initial + rate * 60
```

```
<id,1>  <=> <id,2> <+>  …….. <60>
```

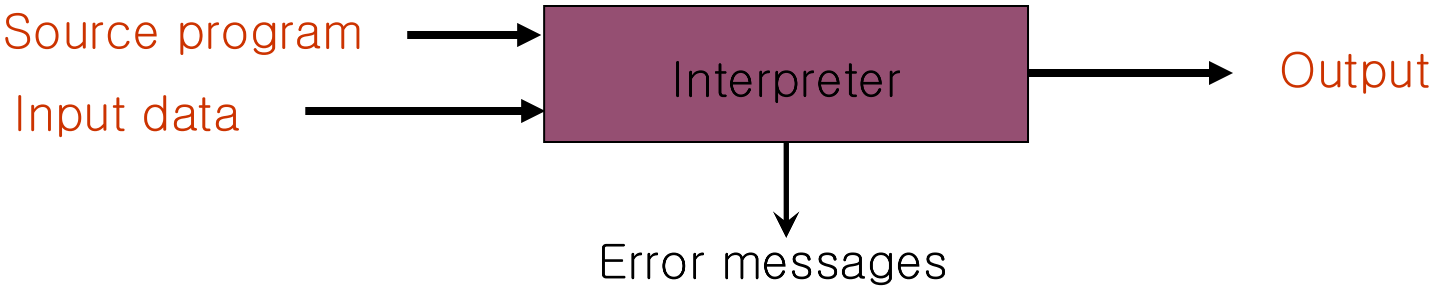
YONSEI UNIVERSITY

Interpreters

- Interpreters
  - Execution engine.
  - Program execution interleaved with analysis.

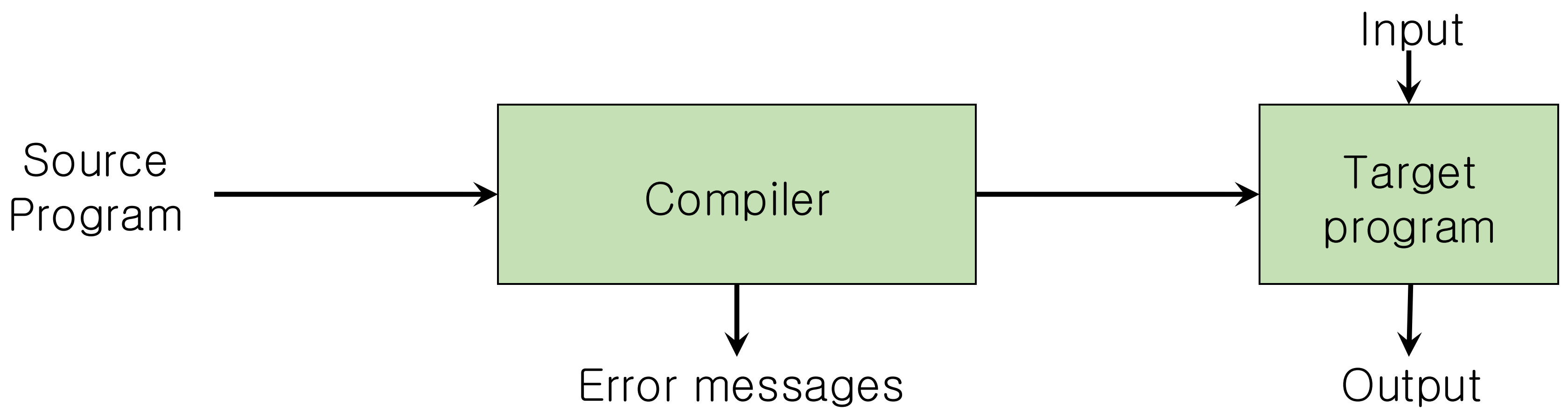
```
running = true;
while (running) {
    analyze next statement;
    execute that statement;
}
```

  - Usually need repeated analysis of statements (particularly in loops, functions.)
  - But: immediate execution, good debugging & interaction.



Compilers

- Compilers
  - Read and analyze entire program.
  - Translate to semantically equivalent program in another language.
    - Presumably easier to execute or more efficient.
    - Should “improve” the program in some fashion.
  - Offline process.
    - Tradeoff: compile time overhead (preprocessing step) vs execution performance.



Typical implementations

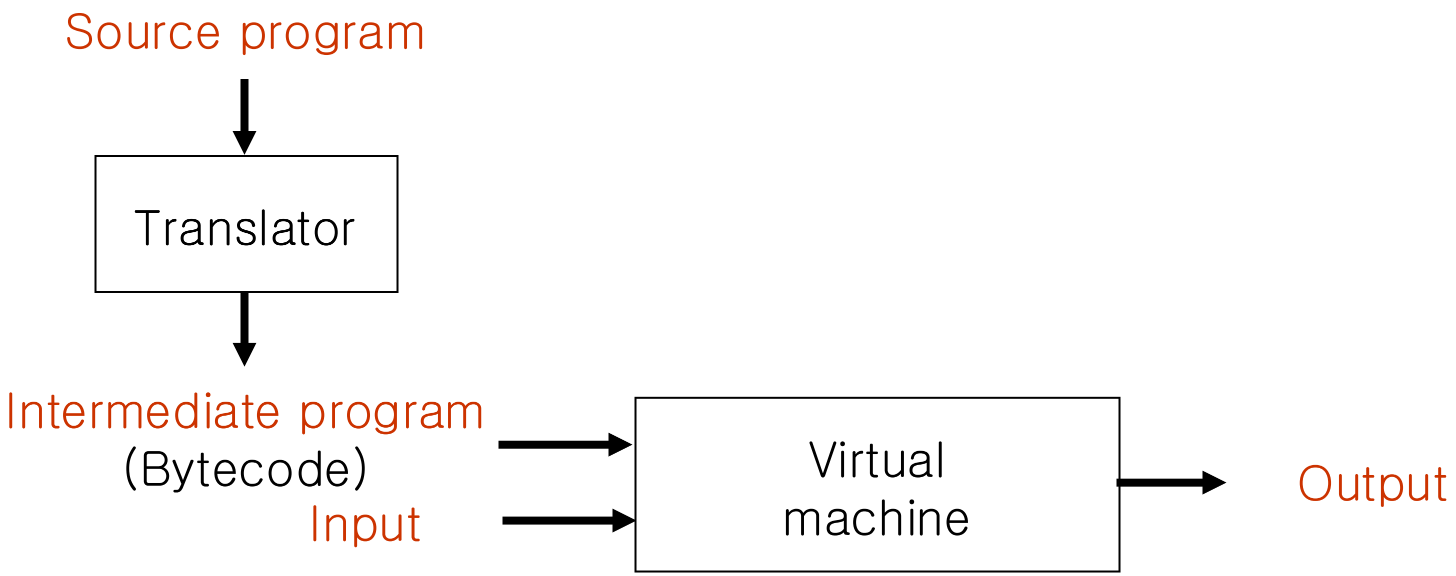
- Interpreters
  - PERL, Python, Ruby, awk, sed, shells, Scheme/Lisp/ML, postscript/pdf, Java VM.
  - Particularly effective if interpreter overhead is low relative to execution cost of individual statements.
- Compilers
  - FORTRAN, C, C++, Java, COBOL, etc. etc.
  - Strong need for optimization in many cases.

Hybrid approaches

- **Well-known example: Java**
  - Compile Java source to byte codes – Java Virtual Machine language (.class files).
  - Execution is either of the following two options.
    - Interpret byte codes directly.
    - Compile some or all byte codes to native code.
      - Just-In-Time compiler (JIT) – detect hot spots & compile on the fly to native code – standard these days.
- **Variation: .NET**
  - Compilers generate MSIL.
  - All IL compiled to native code before execution.

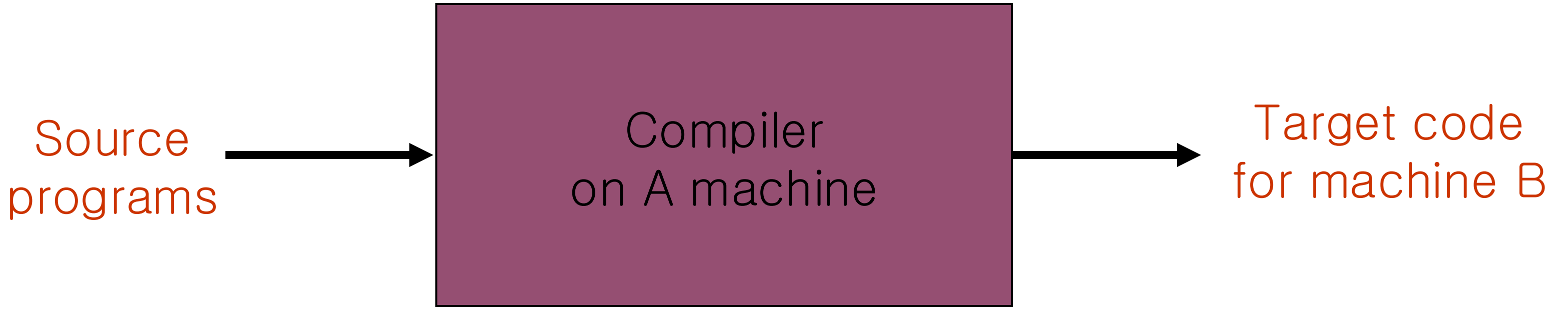
YONSEI UNIVERSITY

Hybrid approaches (e.g., Java)



Cross compiler

- Cross compiler
  - A cross-compiler is a program which is to run on machine A and produce target code for another machine B.
  - Execution of the result: *down-loading* or interpretation.



Some history

- First, there was nothing.
- Then, there was machine code.
- Then, there were assembly languages.
- Programming expensive; 50% and/or more of costs for machines went into programming.

Some history

- High-level languages

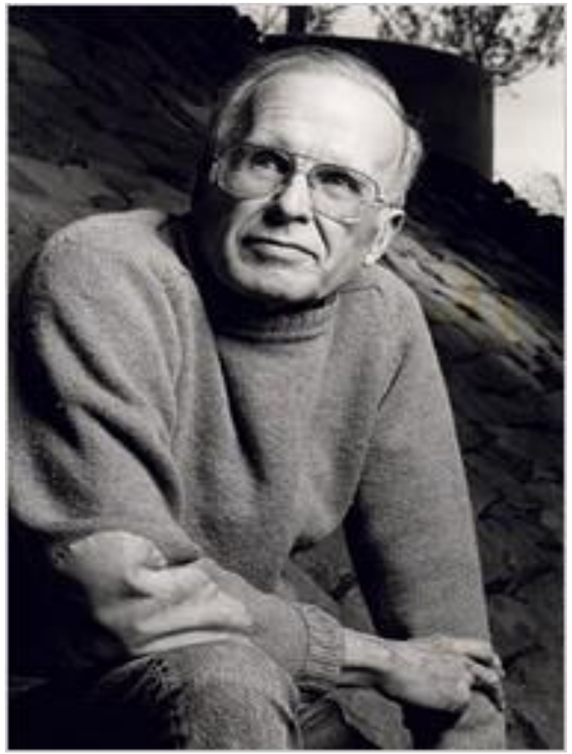
Rear Admiral **Grace Hopper**, inventor of A-0, COBOL, and the term “**compiler.**”

http://upload.wikimedia.org/wikipedia/commons/thumb/5/55/Grace_Hopper.jpg/300px-Grace_Hopper.jpg

http://www.nytimes.com/2007/03/20/business/20backus.html

Some history

- FORTRAN compiler (1957)
  - First commercially available compiler (18 person-years to create)



John Backus, team leader on FORTRAN

http://upload.wikimedia.org/wikipedia/commons/thumb/5/55/Grace_Hopper.jpg/300px-Grace_Hopper.jpg

http://www.nytimes.com/2007/03/20/business/20backus.html

Some history

- 1950's
  - FORTRAN compiler (1957) – first commercially available compiler (18 person-years to create).
- 1960's
  - New languages: ALGOL, LISP, COBOL, SIMULA.
  - Formal notations for syntax, especially BNF.
  - Fundamental implementation techniques (Stack frames, recursive procedures, etc.)
- 1970's
  - Syntax: formal methods for producing compiler front-ends; many theorems.
- Late 1970's, 1980's
  - New languages (functional; Smalltalk & object-oriented.)
  - New architectures (RISC machines, parallel machines, memory hierarchy issues.)
  - More attention to back-end issues.

Some history

- 1990s and beyond
  - Compilation techniques appearing in many new places
    - Just-in-time compilers (JITs)
    - Software analysis, verification, security
  - Phased compilation – blurring the lines between "compile time" and "runtime"
    - Using machine learning techniques to control optimizations(!)
  - Compiler technology critical to effective use of new hardware (RISC, Itanium, complex memory hierarchies)
  - The new gorilla – multicore

Why do we need compilers?

- Traditional view
  - Translation: high-level language (HLL) programs to low-level machine code
  - Optimization: reduce number of arithmetic operations by optimizations
- Modern view
  - Collection of automatic techniques for extracting meaning from and transforming programs
  - Useful for debugging, optimization, verification, detecting malware, translation, …..
  - Optimization:
    - Restructure (reorganize) computation to optimize locality and parallelism
    - Reducing amount of computation is useful but not critical
    - Optimizing data structure accesses is critical
    - Reduce the size of code and power efficiency

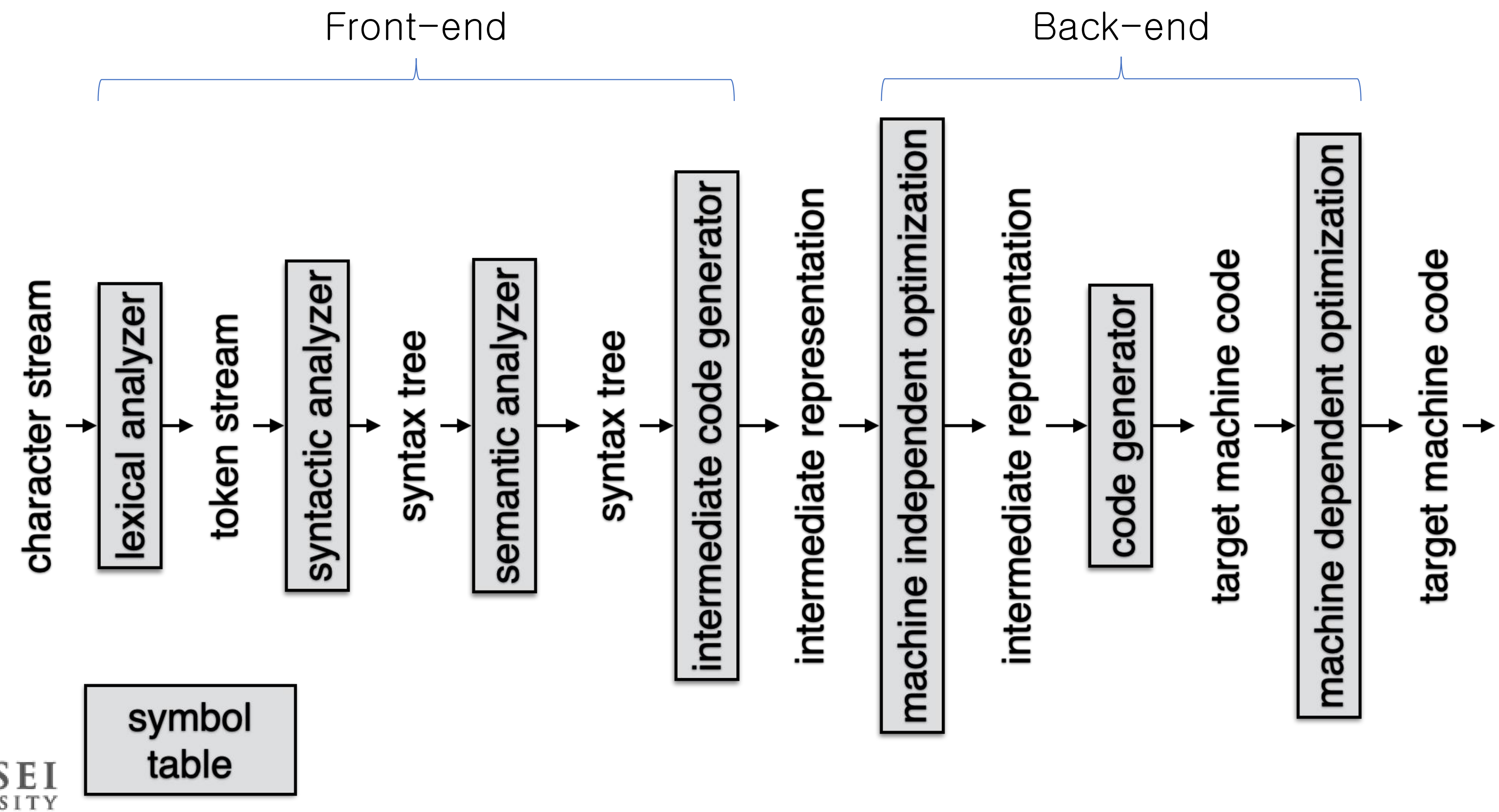
Why do we need compilers?

- Bridge the "semantic gap"
  - Programmers prefer to write programs at a high level of abstraction
  - Modern architectures are very complex, so to get good performance, we have to worry about a lot of low-level details
  - Compilers let programmers write high-level programs and still get good performance on complex machine architectures
- Application portability
  - When a new ISA or architecture comes out, you only need to reimplement the compiler on that machine
  - Application programs should run without (substantial) modification
  - Saves programming effort

Summary: performance + portability of HLL programs

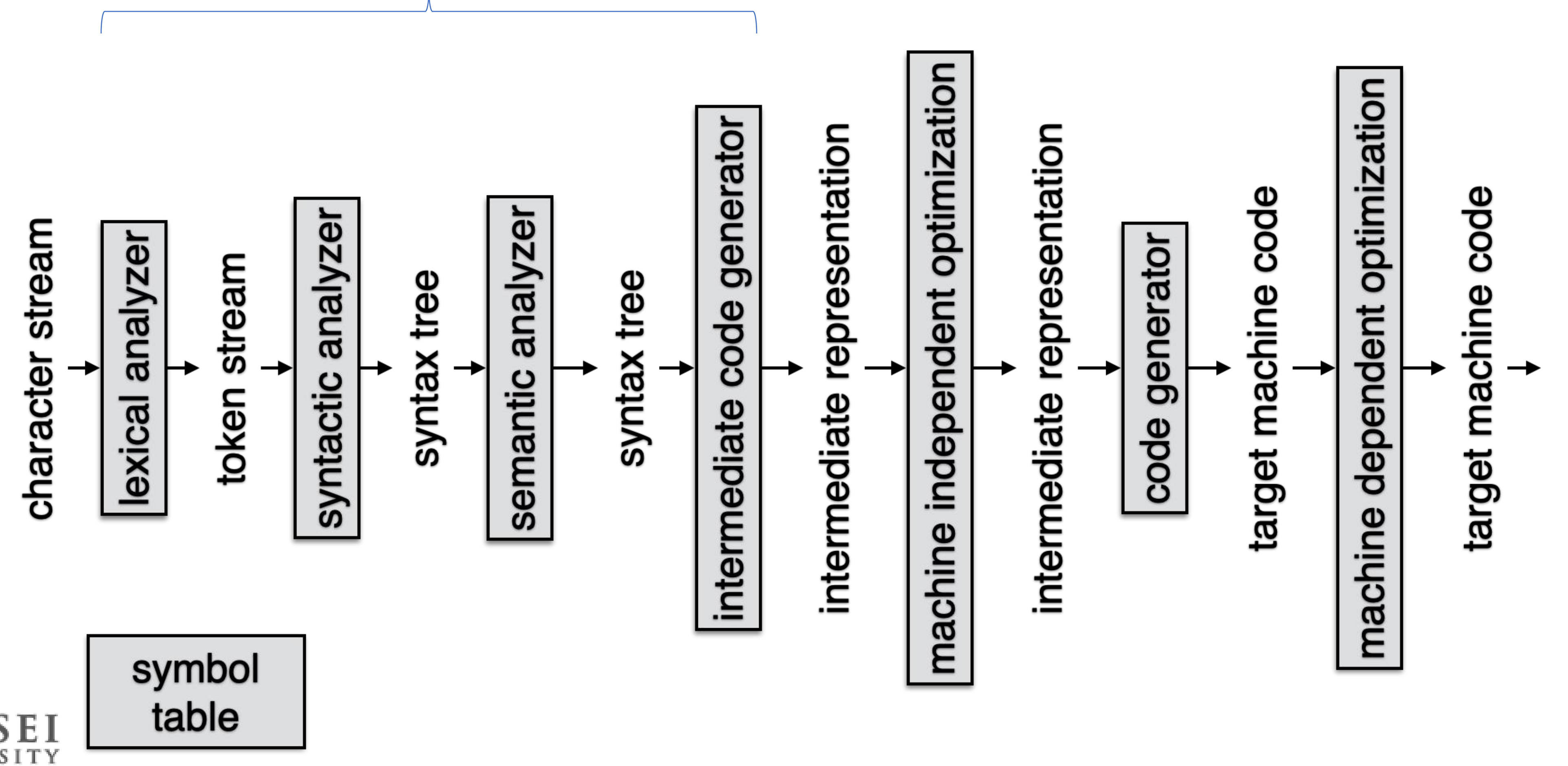
YONSEI UNIVERSITY

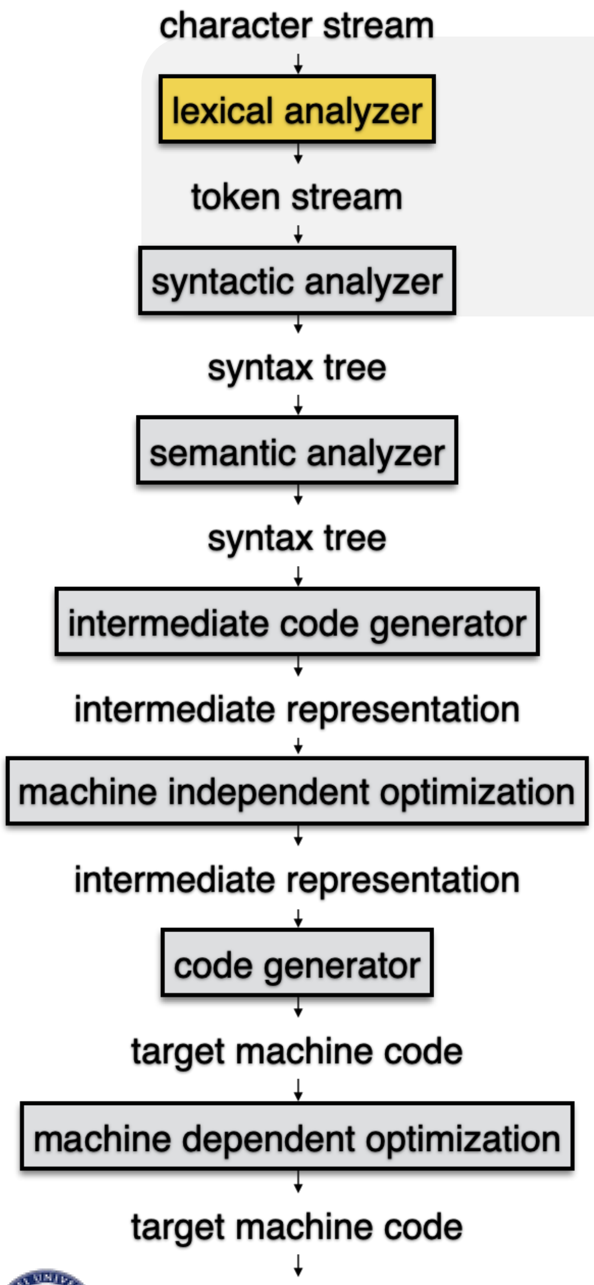
Structure of compilers



Structure of compilers

Scope of our assignments





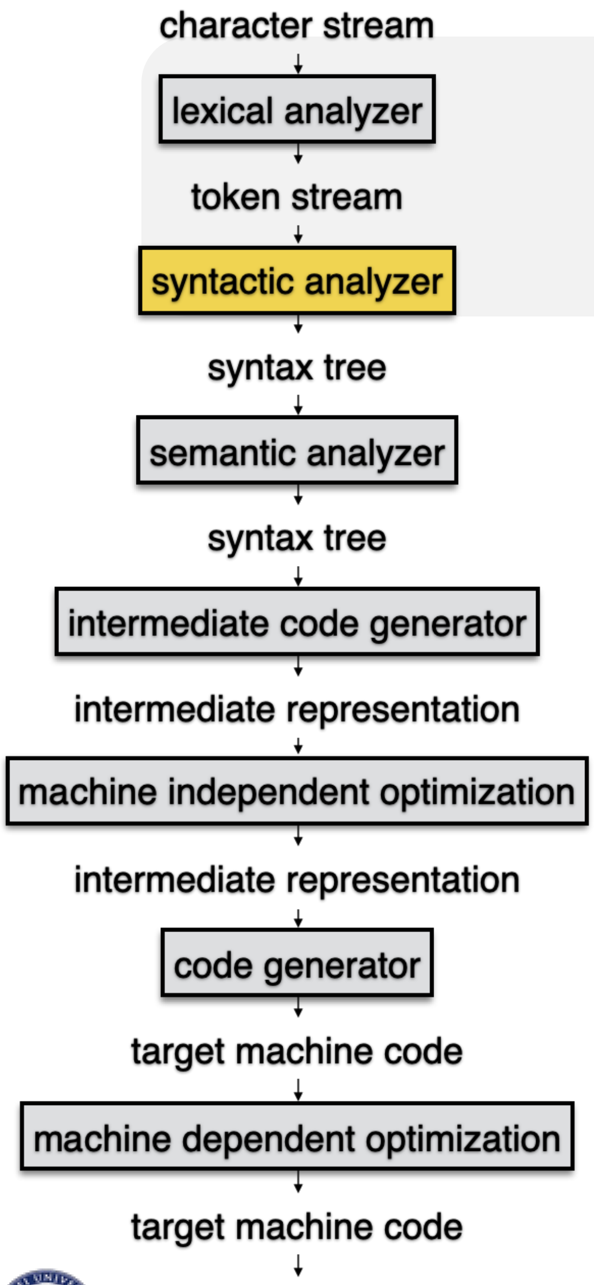
Lexical analysis Scanning

- Lexical analyzer
  - It converts the High-level input program into a sequence of Tokens (컴파일러 내부에서 효율적이며 다루기 쉬운 token으로 바꾸어 줌).
  - Lexical Analysis can be implemented with the Deterministic finite Automata.

```
position = initial + rate * 60

lexemes   : position = initial  +  rate *  60
                 ↓      ↓      ↓     ↓    ↓    ↓    ↓
Token     : <id,1>  <=> <id,2> <+>  ……..  <60>
```

Symbol table entry   Abstract symbols

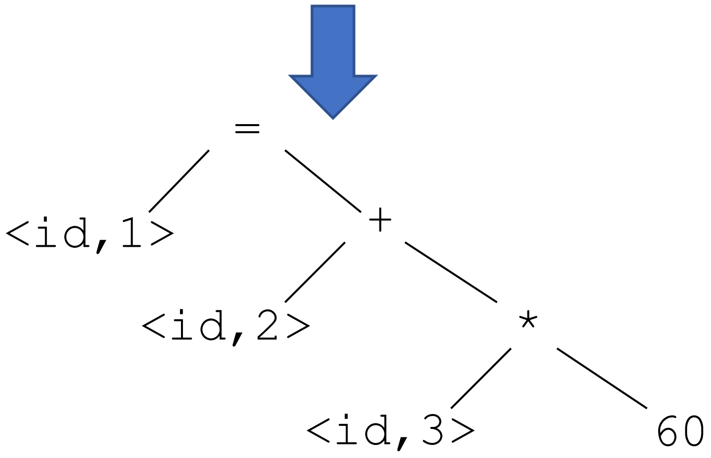
idx	name	token	type	…
1	positio n	id	float	…
2	Initial	id	float	…
3	rate	id	float	…

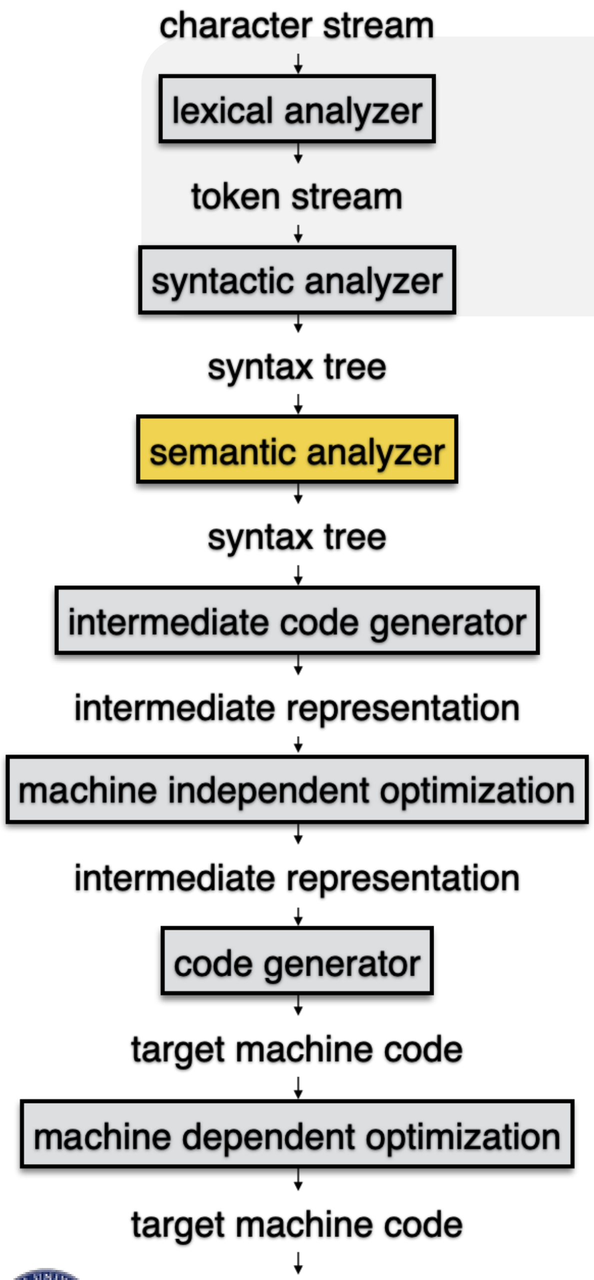


Syntax analysis Parsing

- Syntax analyzer
  - Functionalities
    - Syntax checking, Tree generation.
    - build grammatical structure
  - Result
    - Print program structure (=> with tree structure)

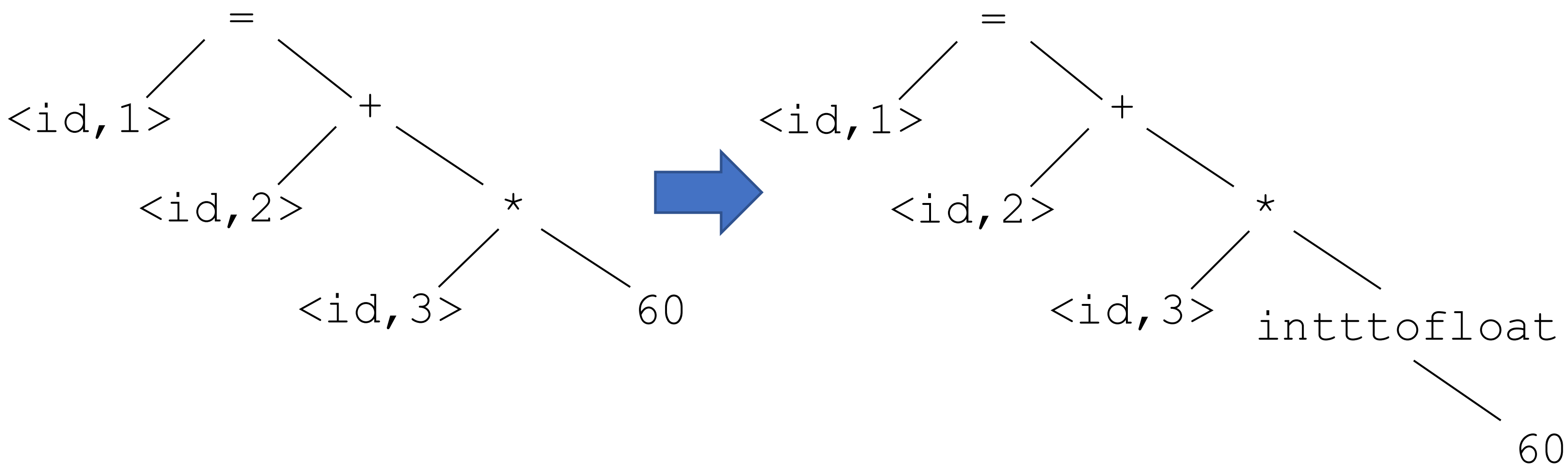
<id,1> <=> <id,2> <+> ……. . <60>



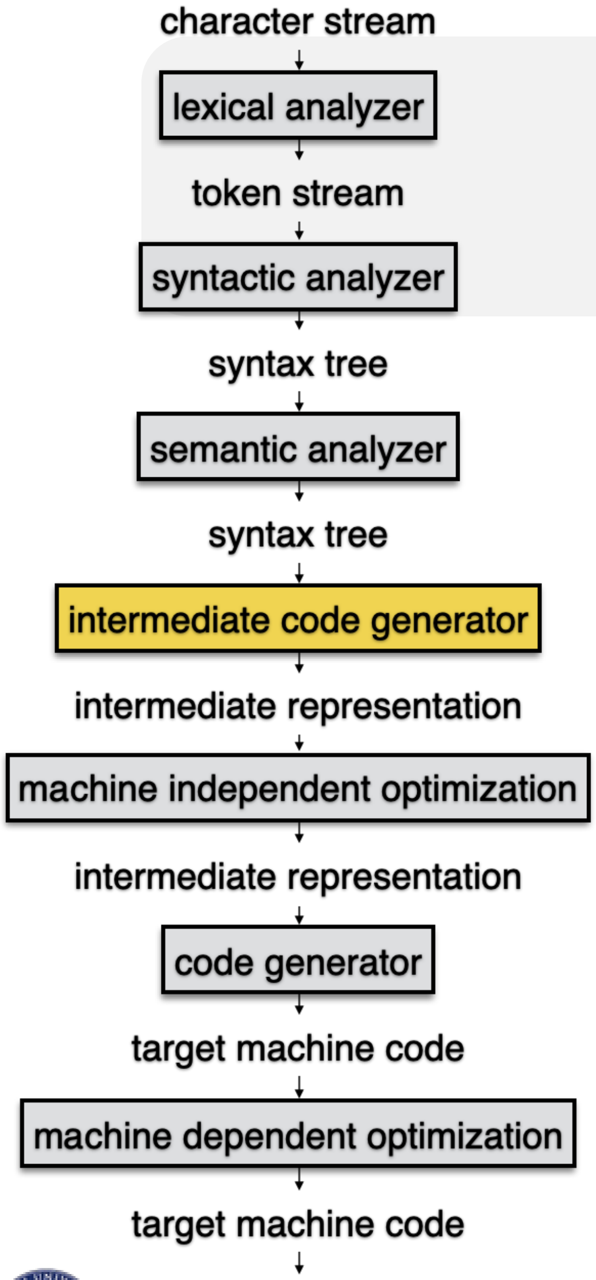


Semantic analysis

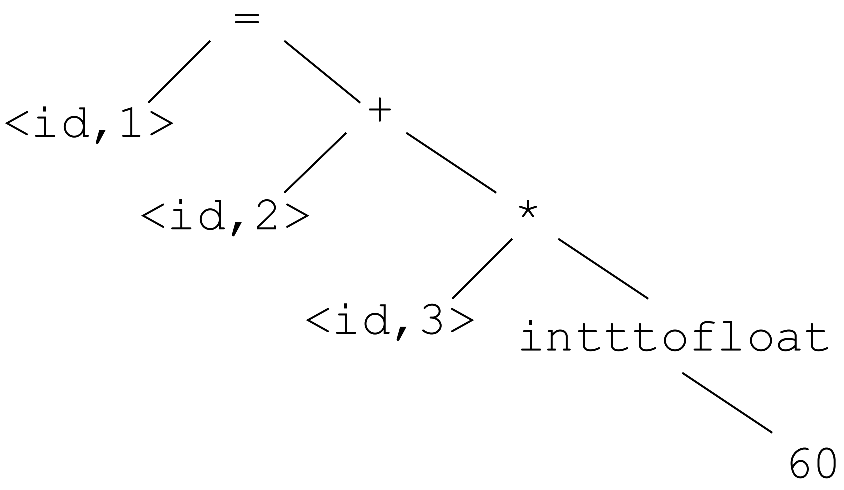
- Semantic analyzer
  - Check the consistency of Semantic tree
  - Refer Symbol table
  - Examples
    - `IF A > 10 THEN  A := 1.0` ➔ Semantics error when A is an integer number



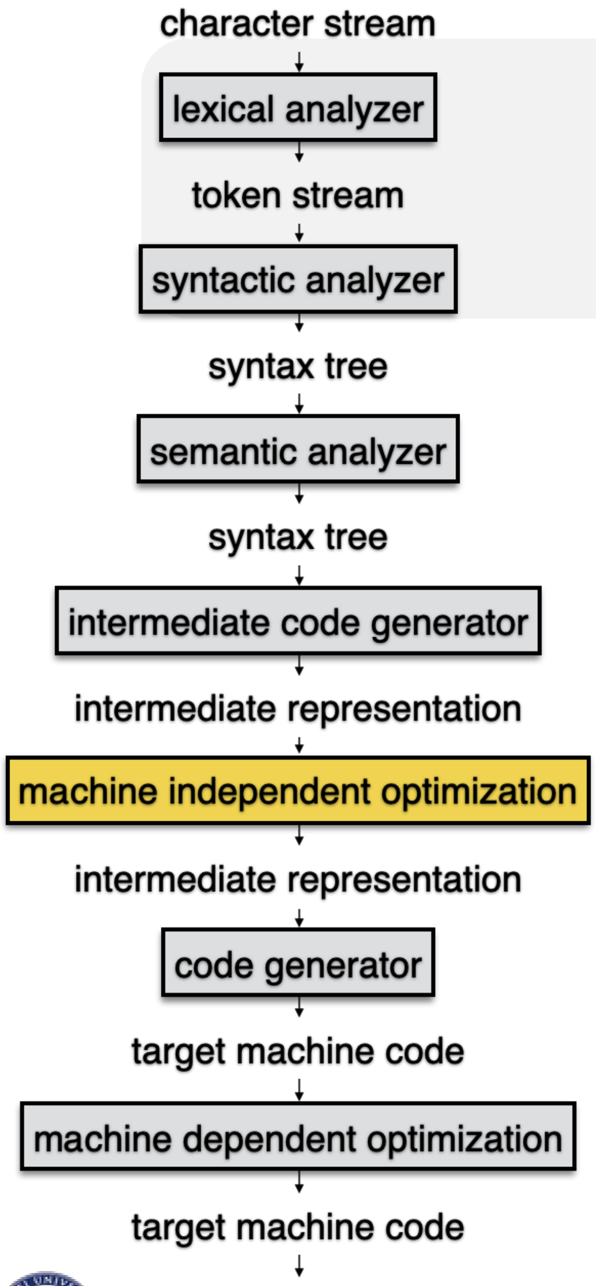
Intermediate code generation



- Intermediate code generator
  - Generate Intermediate Code from syntax tree
  - Machine like, low-level intermediate representation




```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
t4 = t3
```



Machine independent

Code optimization

- Intermediate code generator
  - Generate Intermediate Code from syntax tree
  - Machine like, low-level intermediate representation

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
t4 = t3
```

```
t2 = id3 * 60.0
t4 = id2 + t2
```

Code optimization

- Code optimization
  - Optional phase
    - Provide a more efficient code by modifying inefficient parts in the original code
    - improve running time, reduce code size
    - Should generate same results!

      e.g.,) `t2 = id3`

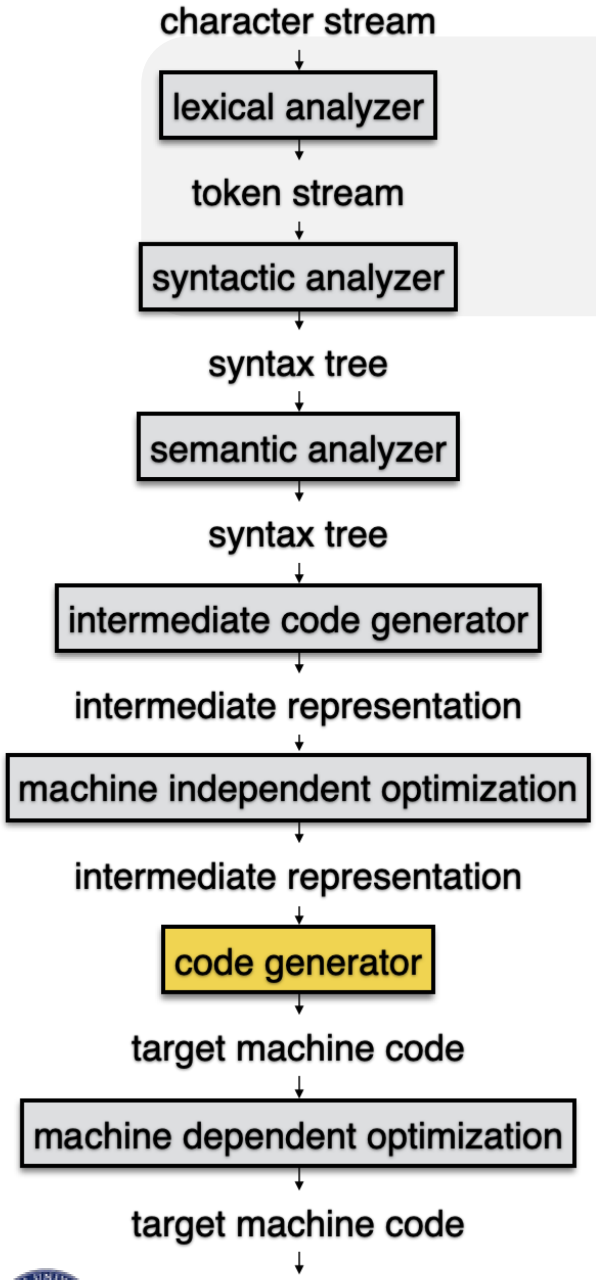
      `t2 = id3` (x)
  - Criteria for optimization
    - <u>Preserve the program meanings</u>
    - Speed up on average
    - Be worth the effort

YONSEI UNIVERSITY

Code optimization

- Local optimization
  - Find out inefficient codes via local inspection and change those parts into more efficient ones
  - Constant folding
  - Eliminating redundant load, store instructions
  - Algebraic simplification
  - Strength reduction
- Global optimization
  - Use flow analysis technique
  - Common sub expression elimination
  - Moving loop invariants
  - Removing unreachable codes

YONSEI UNIVERSITY



Code generation

- Code generation
  - Generate machine instruction from intermediate representations
  - Code generator tasks
    - Instruction selection & generation
    - Register management
    - Storage allocation

```
t2 = id3 * 60.0
t4 = id2 + t2
```

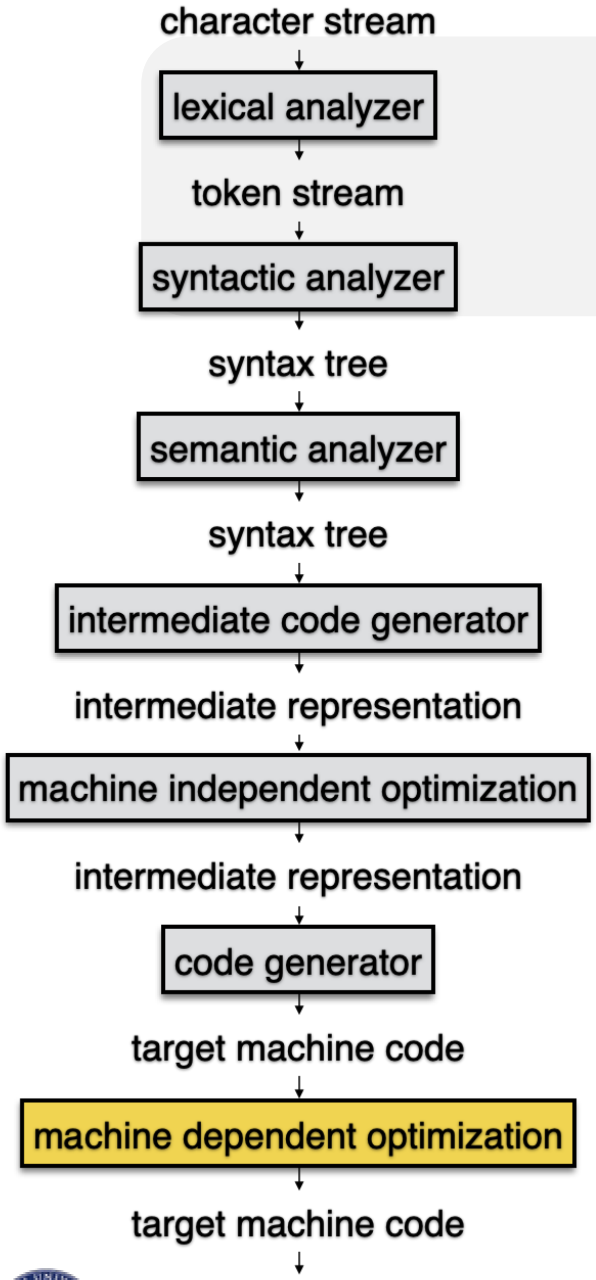
→

```
LDF   R2,  id3
MULF  R2,  R2,  #60.0
LDF   R1,  id2
ADDF  R1,  R1, R2
STR   id1,  R1
```

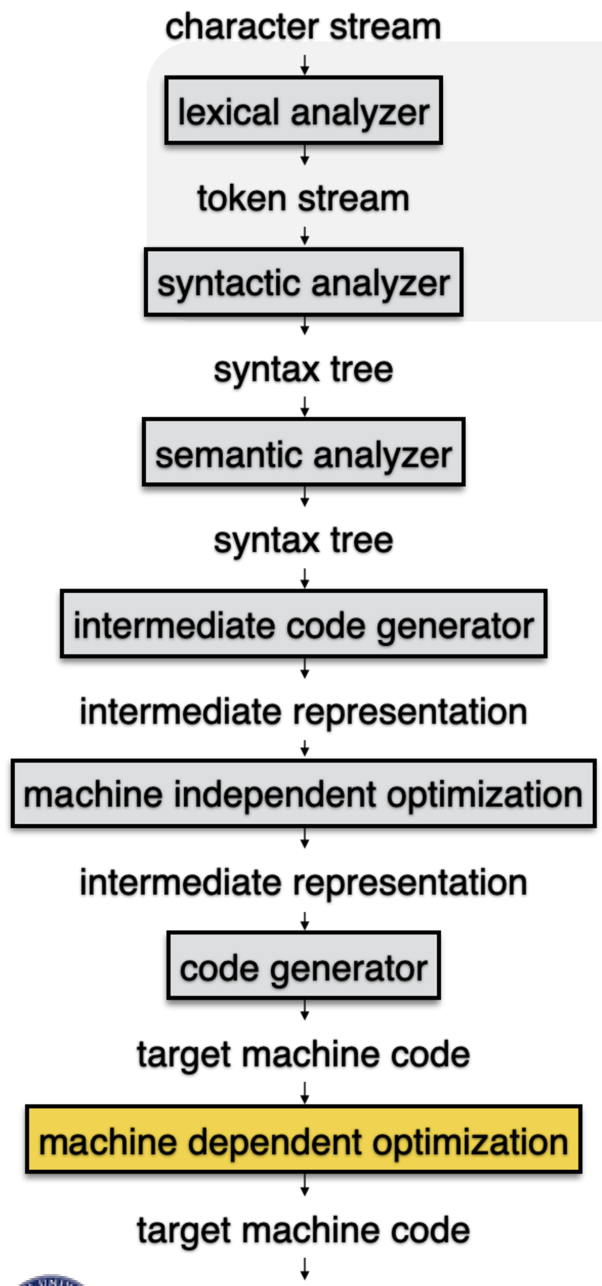
YONSEI UNIVERSITY

Machine dependent

Code optimization



- Optimizations for new architecture
  - Parallelism
    - Support Instruction level parallelism
      - Super Scalar Unit, VLIW, SIMD (vector machines), Multi-Core CPUs, TPU, GPU
      - Sol 1. User (programmer) write parallel code
      - Sol 2. HW detects the opportunity of parallelism
      - Sol 3. Compiler automatically generate parallel code from normal code
    - ➔ Combine all together!!!



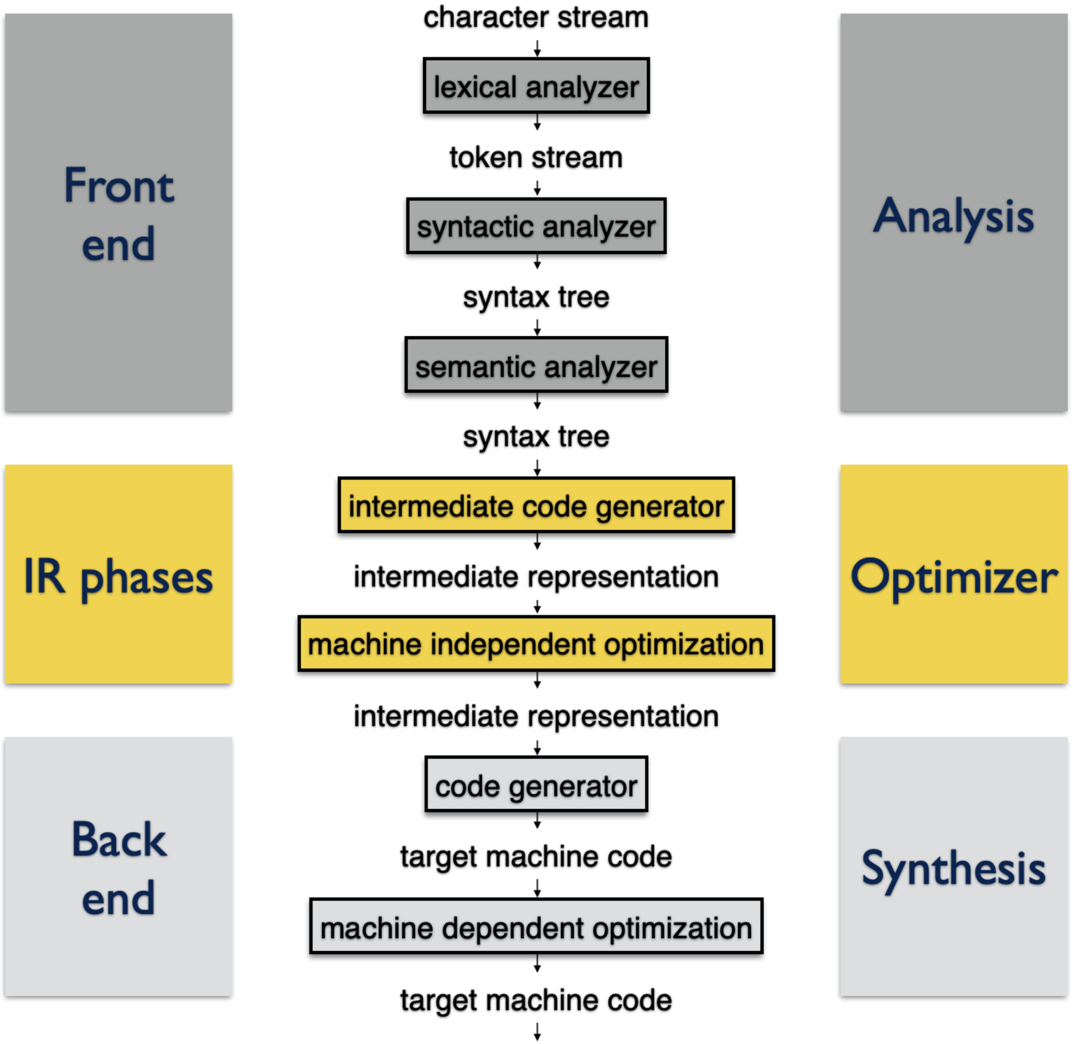
Machine dependent

Code optimization

- **Optimizations for new architecture**
  - Memory Hierarchy
    - Memory has a hierarchy in all computing machines
      - Trade off between speed and capacity
  - Designing of architecture and compiler are closely related
    - Traditional model : Architecture design -> Compiler
    - RISC : Compiler technique drive the architecture design

YONSEI UNIVERSITY

Parts of compilers



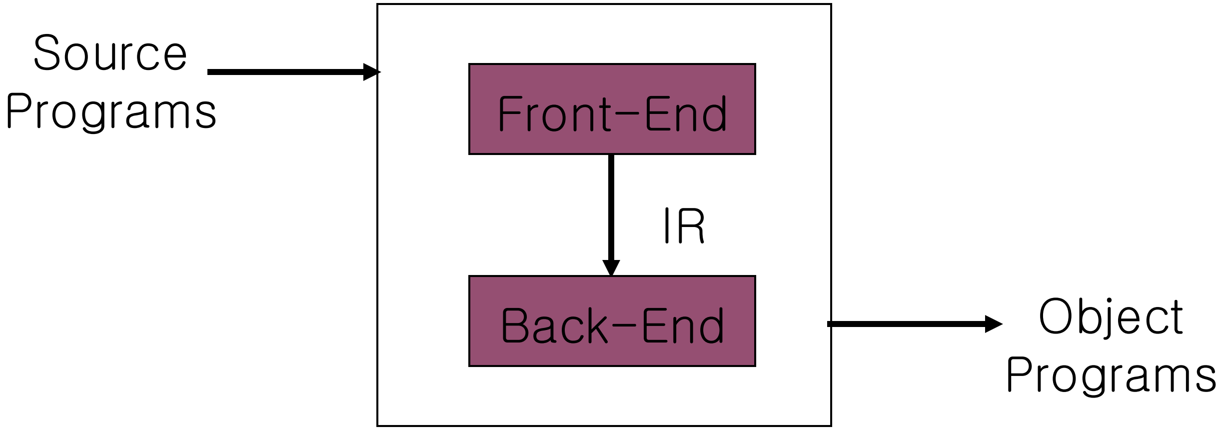
YONSEI UNIVERSITY

Multi-pass compiler

- Passes
  - Pass: Read an input file and writes output file.
  - One-pass compiler
    - Read input source code and generate output target code at once.
  - Multi-pass compiler
    - Create IR and perform input read/output write multiple times
    - Pros: we can expect better performance
    - Cons: it has more overhead due to several I/Os

Multi-pass compiler

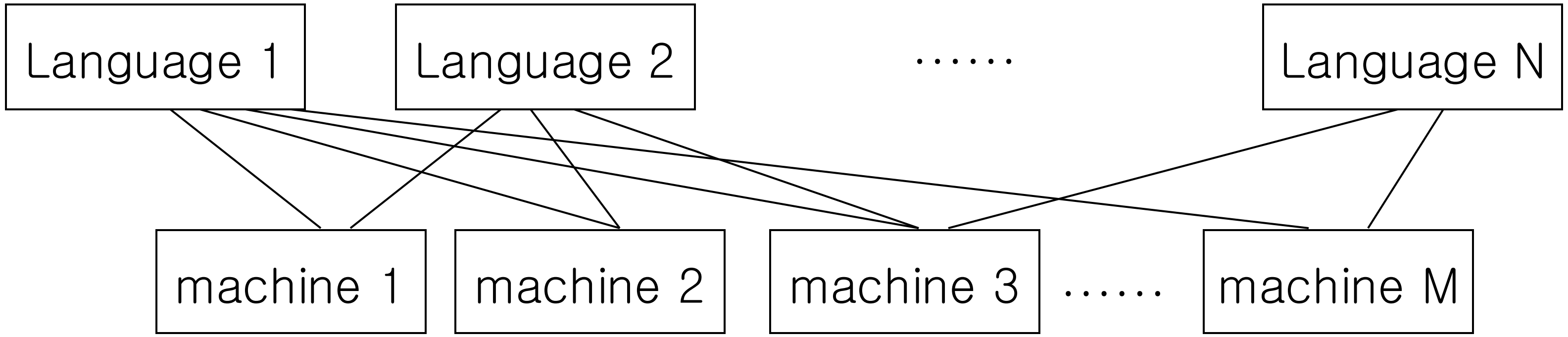
- Two-pass compiler
  - Front-end
    - Multiple passes →Better code
    - *O(n) or O(nlogn)*
    - Legality check (error report)
    - Produce IR
    - Preliminary storage map
    - Main topic of this course
  - Back-end
    - NP complete

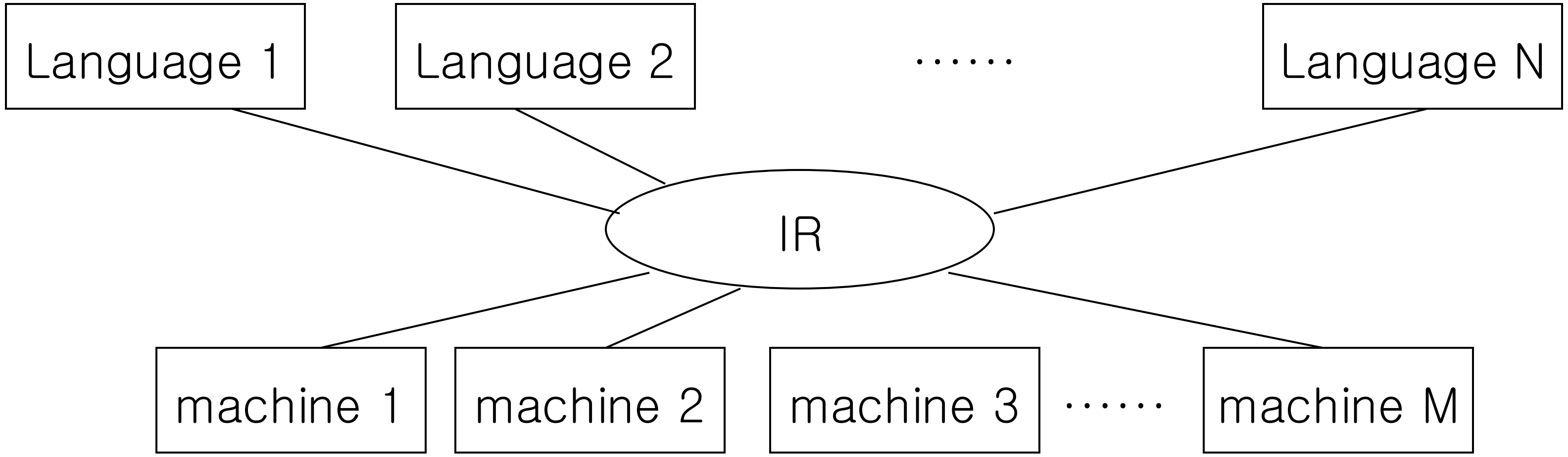


Front-End : language dependent part
Back-End : machine dependent part
IR : intermediate representation

Two-pass compiler

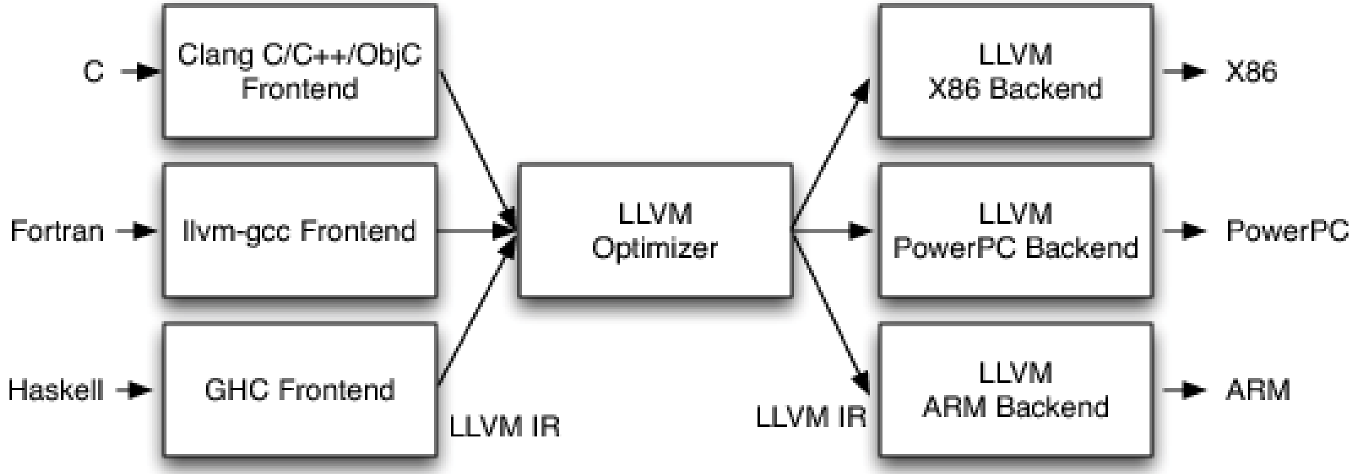
- Two-pass compiler (benefits) – How many compilers do we need?





LLVM

- LLVM
  - Is a set of compiler and toolchain technologies that can be used to develop a front end for any programming language and a back end for any instruction set architecture.
  - Is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes.
  - Has been an integral part of Apple's XCode development tools for macOS and iOS since XCode 4.



LLVM structure

Compiler construction tools

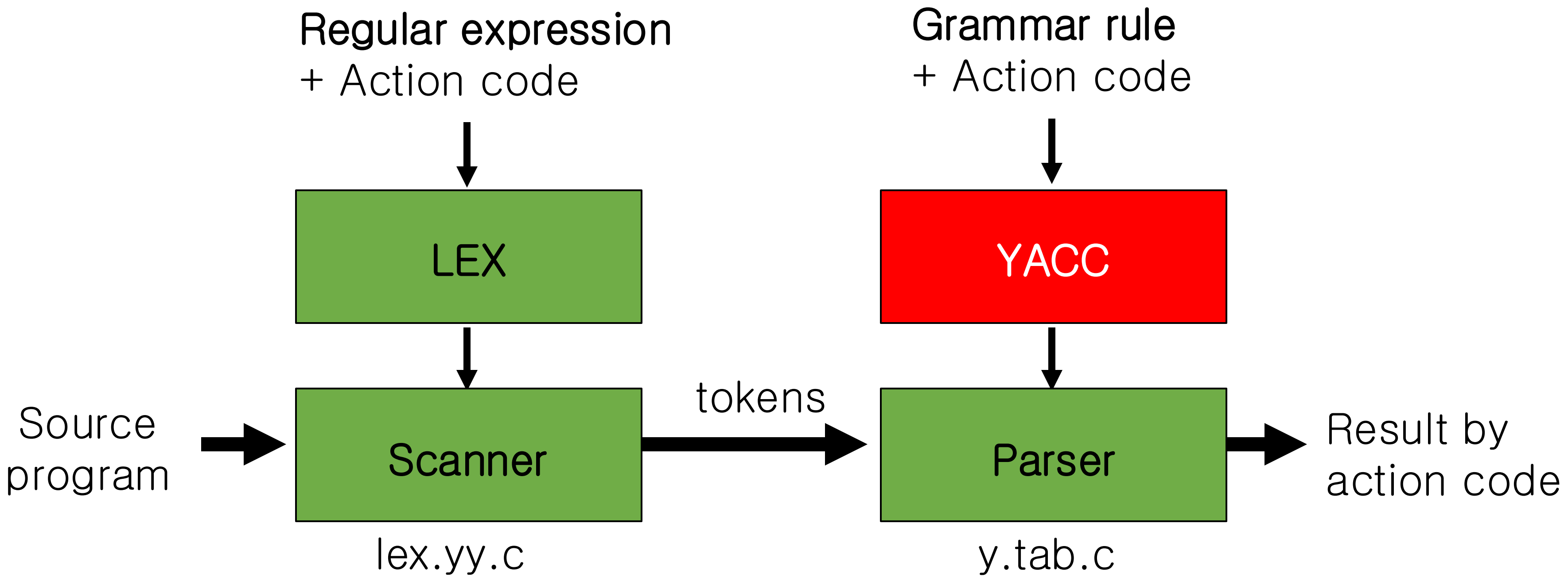
Tools	Characteristics
Parser generator	Automatically generate syntax analyzer from grammatical description
Scanner generator	Lexical analyzer from a regular-expression description of the token
Syntax-directed translation engines	Walking routines of a parse tree for code generator
Code-generator generator	Generate code generator from rules of translation
Data-flow analysis engines	Facilitate the gathering of information from one part to another
Compiler-construction toolkits	Compiler-compiler, Translator writing system

http://www.compilertools.net/

Lexical analyzer generator & parser generator

- Lex
  - Lexical analyzer generator
  - *M.E.Lesk* invented in 1975
  - It is useful tool to write programs that finds out tokens written in regular expressions from the input stream
- YACC (Yet Another Compiler Compiler)
  - Parser generator Run on top of UNIX (GNU Bison)
  - Written in C language

Lexical analyzer generator & parser generator



Applications of compiler techniques

- Implementation of high-level programming languages
- Optimizations for computer architectures
- Design of new computer architectures
- Program translators
  - SQL, HDL, compiled simulation
- SW productivity Tools
  - Memory-management tools (purify)
- Text Editor
  - Syntax checking, syntax highlighting
  - Auto code completion, keyword recommendation

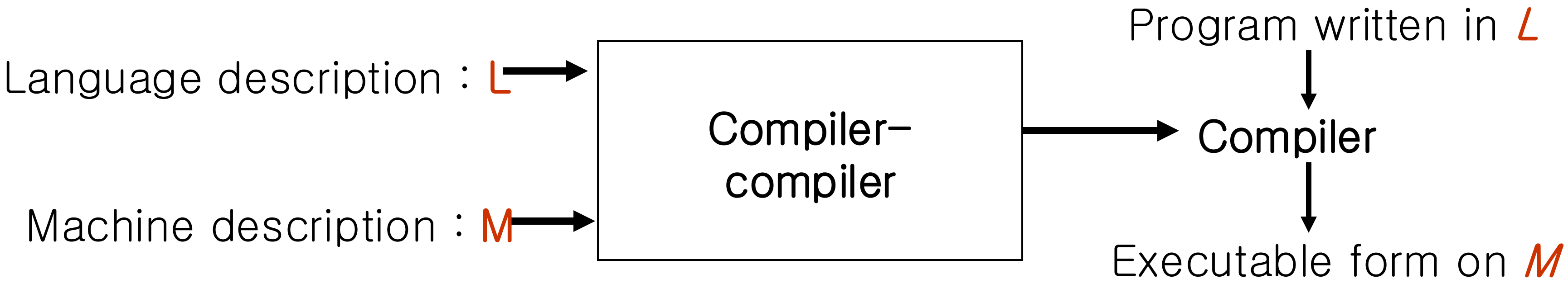
Applications of compiler techniques

- Pretty printer
  - Auto indentation
- Text formatter
  - Tex, LaTeX
- Internet browser
  - HTML : rendering web pages
- XML parser
  - use of XML format documents

Compiler generator (compiler-compiler)

- Compiler generator
  - More and more languages and machines has been invented, more compilers are required
  - New languages are required due to the new area of computer science and engineering
  - With N language and M machines, we requires N * M compilers
    - Two languages: C, Java
    - Three machines: IBM, SPARC, Pentium
    - C-to-IBM, C-to-SPARC, C-to-Pentium, Java-to-IBM, Java-to-SPARC, Java-to-Pentium
  - Language description uses grammar theory, but machine description does not have any formalized methods yet
    - HDL : Hardware Description Language, and it used in designing computer architecture
    - Automatic compiler generation is in research due to the improvement in machine architecture and programming language

Compiler generator (compiler-compiler)



Programs in compilers

- Preprocessor

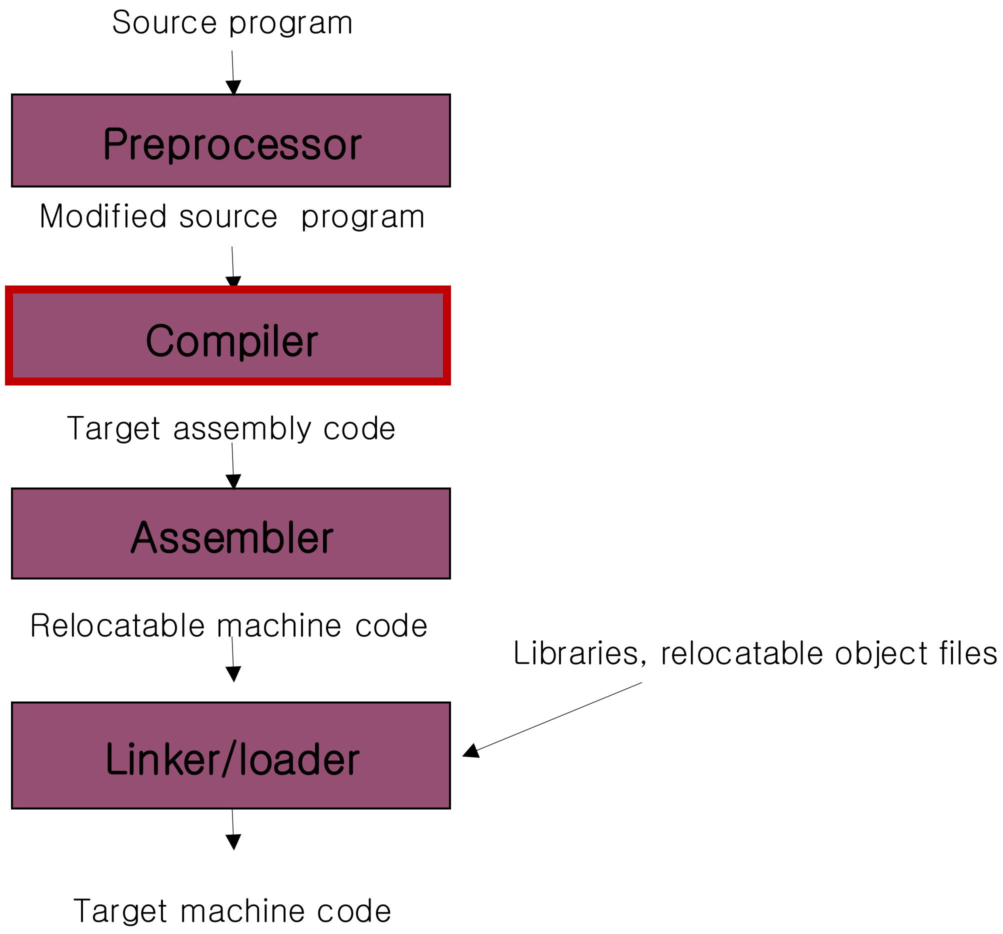
```
#include <stdlib.h> // malloc, free
#ifdef _WIN32
#include <malloc.h>
#endif
...

# if defined __GNUC__ || defined __clang__
#  define UA_alloca(size) __builtin_alloca (size)
# elif defined(_WIN32)
...
```

Programs in compilers

- Assembler
  - Translate assembler to machine code
- Loader/linker
  - Linker
    - Intakes object codes(generated by assembler)
    - Combine them to generate executable module (ELF)
  - Loader
    - Loads the executable module to the main memory for execution
- Library routines
  - Sub module and programs

YONSEI UNIVERSITY

Programs in compilers



Programming language theories

- **Background language theory affects compilers**
  - Decision Making (policy) about (source) Program
    - Static (compile time).vs. dynamic(run-time)
    - Ex: Scope of Declaration
      - Static(lexical) scope .vs. dynamic scope
  - Environments and states
    - Most binding of names to location is dynamic
    - Binding of locations to values are generally dynamic

$$\text{names} \xrightarrow{\text{environments}} \begin{array}{c}\text{locations}\\ \text{(variables)}\end{array} \xrightarrow{\text{states}} \text{values}$$

Programming language theories

- Background language theory affects compilers
  - Scope
    - The region where a certain name is valid
    - Most languages use static scope rules
      - Exceptions? (dynamic scope)
    - Block groups the scope
  - Explicit access control
    - Public, private, protected
  - Parameter passing
    - Call by value, call by reference, call by name

Questions?