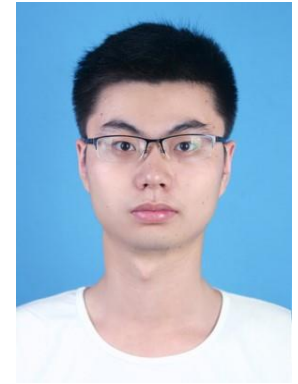


# CS131: Programming Languages and Compilers

Fall 2019

# Course Information

- Instructor: Fu Song  
Office: Room 1A-504C, SIST Building  
Email: [songfu@shanghaitech.edu.cn](mailto:songfu@shanghaitech.edu.cn)
- Class Hours : Tuesday and Thursday, 15:00--16:45  
Room 1D-106, SIST Building
- TA: Luwei Cai, Qi Qin
- Office Hours: TBD
- Discussion: PIAZZA  
<https://piazza.com/shanghaitech.edu.cn/fall2019/CS131>
- Writing Assignment: Gradescope, tex/pdf  
<https://www.gradescope.com>



Luwei Cai



Qi Qin

# Preliminaries Required

## Preliminaries

- data structures and algorithms
- programming languages (C/C++)
- optional (MIPS ASM)

## Textbook

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman  
“*Compilers: Principles, Techniques, and Tools*” Second Edition  
Pearson Addison-Wesley, 2007, ISBN 0-321-48681-1

## Reference book

1. Kenneth C. Loudon, *Compiler Construction Principles and Practice*, 2002

# Course Structure = Goals

- Theoretical Part

Basic knowledge of theoretical techniques

- Practical experience

Design and program an actual compiler using tools, such as LEX/Flex, YACC/Bison

# Grading

- Homework : 20% (4)
- Projects : 25% (4)
- Quiz : 10% (random)
- Midterm : 20%
- Final : 25%
- Bonus : 10%

Optional projects (2-3 students per project)

solution code + report + presentation + demo

Ark compiler: [www.openarkcompiler.cn](http://www.openarkcompiler.cn) (discussion) ?

# Integrity

You must **NOT**

- use work from uncited sources
- read or possess solutions written by others
- allow other people to read or possess your solution

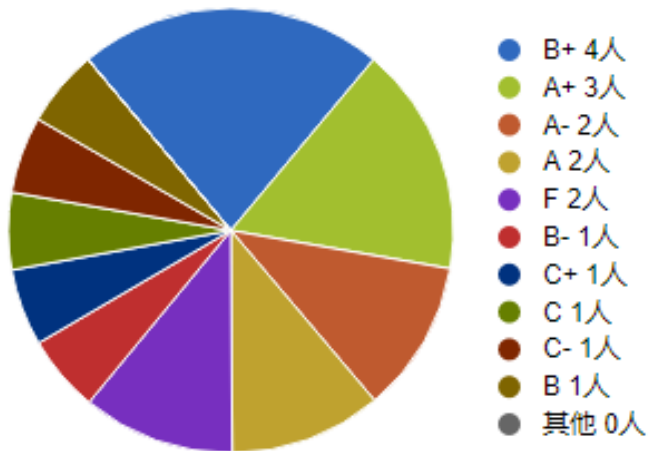
Ask **before** action when in doubt

# Course Outline

- Introduction to Compiling
- Lexical Analysis
  - Regular expressions, regular definitions
  - NFA, DFA, DFA minimization,
  - Subset construction, Thompson's construction
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Intermediate Representations
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Intermediate Code Generation
- Semantic Analysis, Type Checking
- Operational Semantics
- Run-Time Environment
- Code Generation
- Garbage Collection
- Optimization, Dataflow Analysis, Control Hijacking Attack

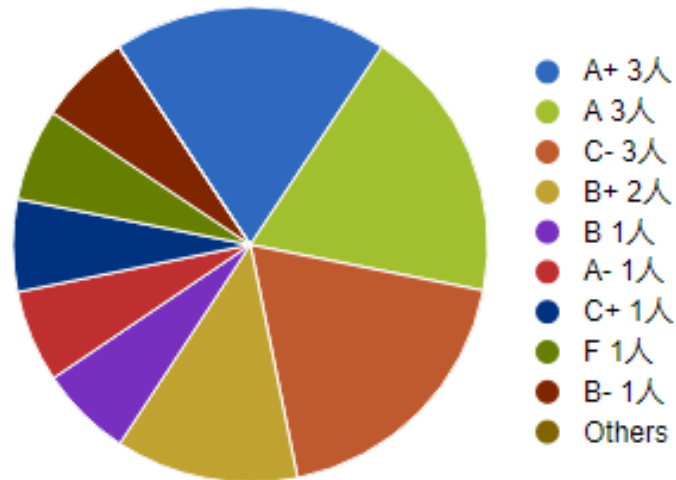
# Quit or Stay

Spring 2017



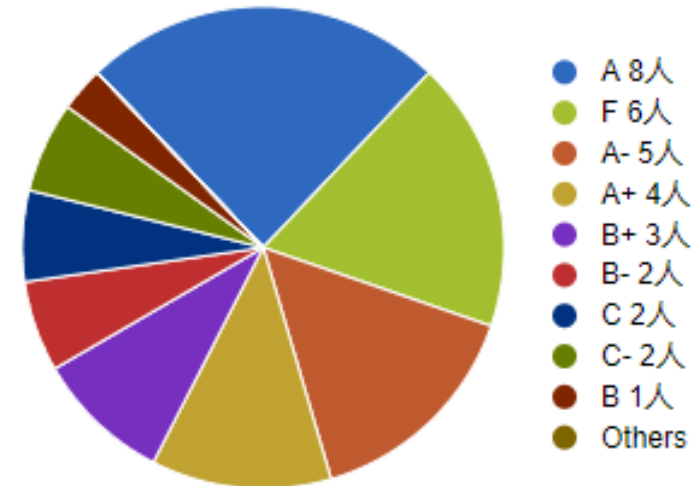
31 → 18 Students

Spring 2018



33 → 16 Students

Fall 2018



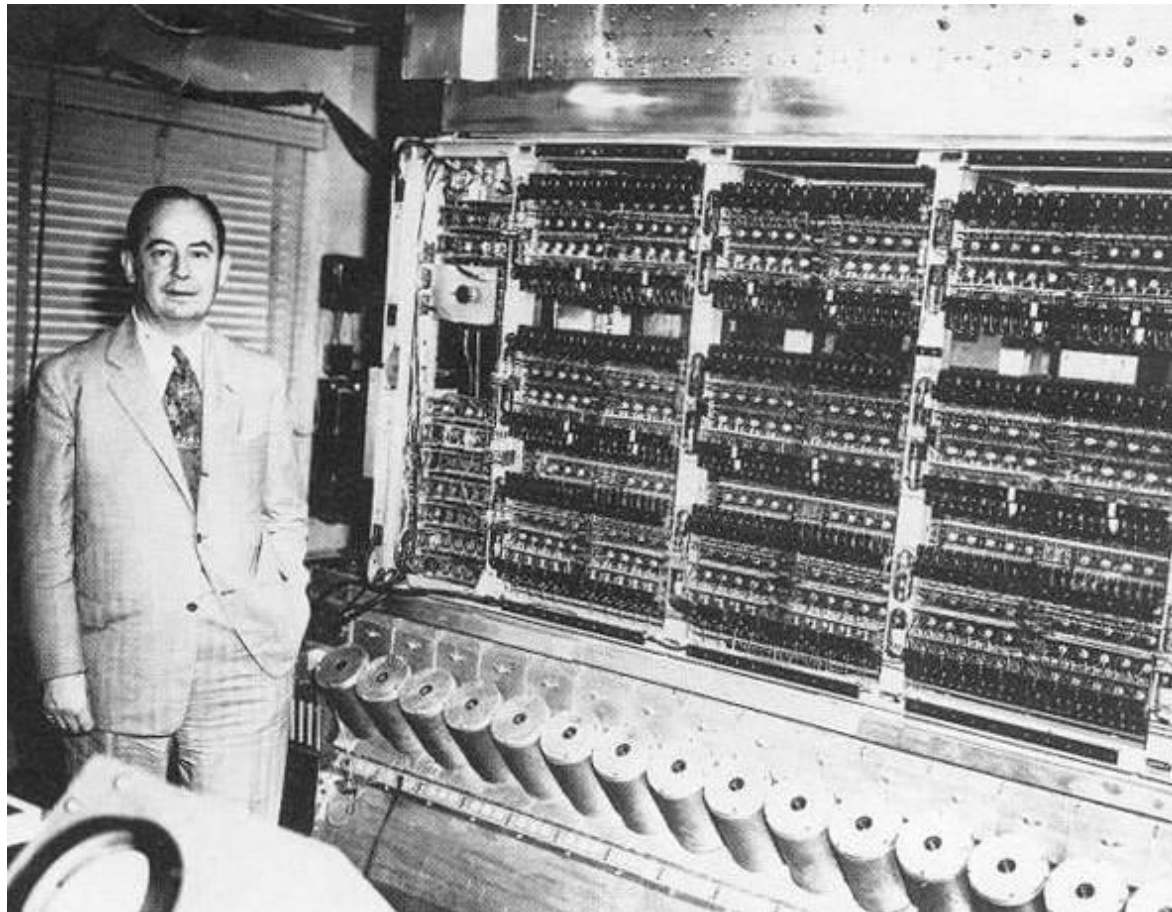
54 → 33 Students

57 → 54 → ? Students



# A brief history of compiler

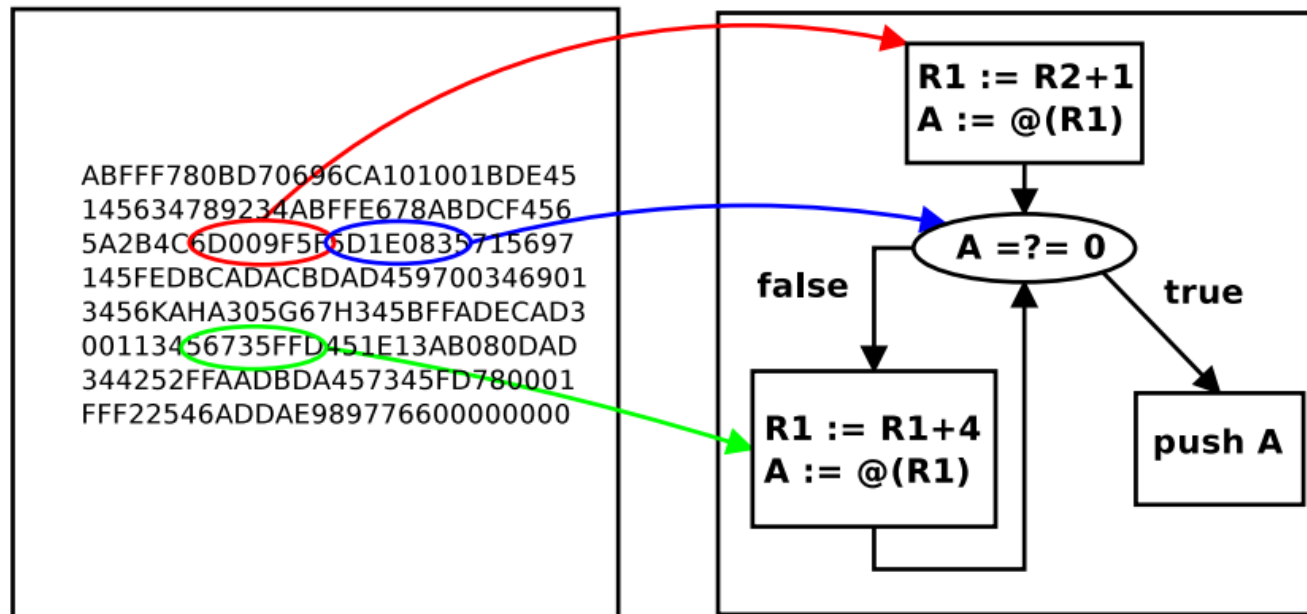
- In the late 1940s, the **stored-program computer** invented by John von Neumann, programs were written in machine language,  $\{0,1\}^+$



# A brief history of compiler (cont.)

- **Mnemonic assembly language**: in 1949  
numeric codes were replaced symbolic forms `Mov R, 2 (R=2)`
- **Assembler**: translate the symbolic codes and memory location of **assembly language** into the corresponding **numeric codes**.

Macro instructions + assembly language



# A brief history of compiler (cont.)

- **FORTRAN language and its compiler**: between 1954 and 1957 , developed by the team at IBM , John Backus.
  - (1) The structure of natural language studied by **Noam Chomsky**,
  - (2) The classification of languages according to the complexity of their grammars and the power of the algorithms needed to recognize them.
  - (3) Four levels of grammars: type 0 、 type 1、 type 2 and type 3 grammars
    - Type 0: Turing machine/Recursive enumerable language
    - Type 1: context-sensitive grammar/language
    - Type 2: context-free grammar/language, the most useful of programming language
    - Type 3: right-linear grammar/language, regular expressions

# A brief history of compiler (cont.)

- (4) parsing problem : studied in 1960s and 1970s
- (5) Code improvement techniques (optimization techniques): improve compilers efficiency.
- (6) Compiler-compilers (parser generator): only in one part of the compiler process.

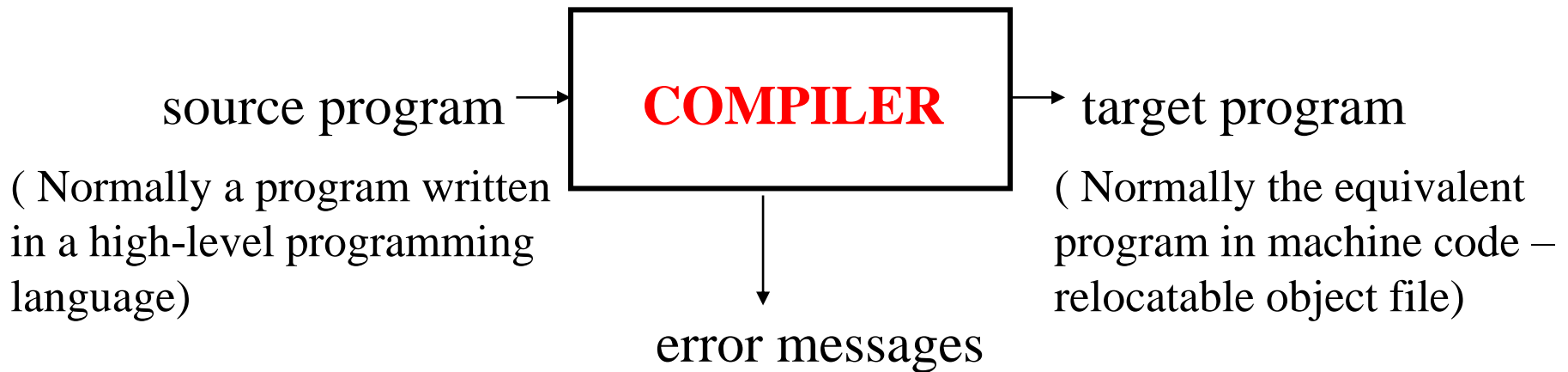
YACC: written in 1975 by Steve Johnson for the UNIX system.

LEX: written in 1975 by Mike Lest.

- (7) recent advances in compiler design:
  - (a) application of more sophisticated algorithms for inferring and /or simplifying the information contained in a program (with the development of more sophisticated programming languages that allow this kind of analysis), e.g. memory safe language Rust
  - (b) development of standard windowing environments ( interactive development environment IDE)
  - (c) just-in-time compiler

# Compilers

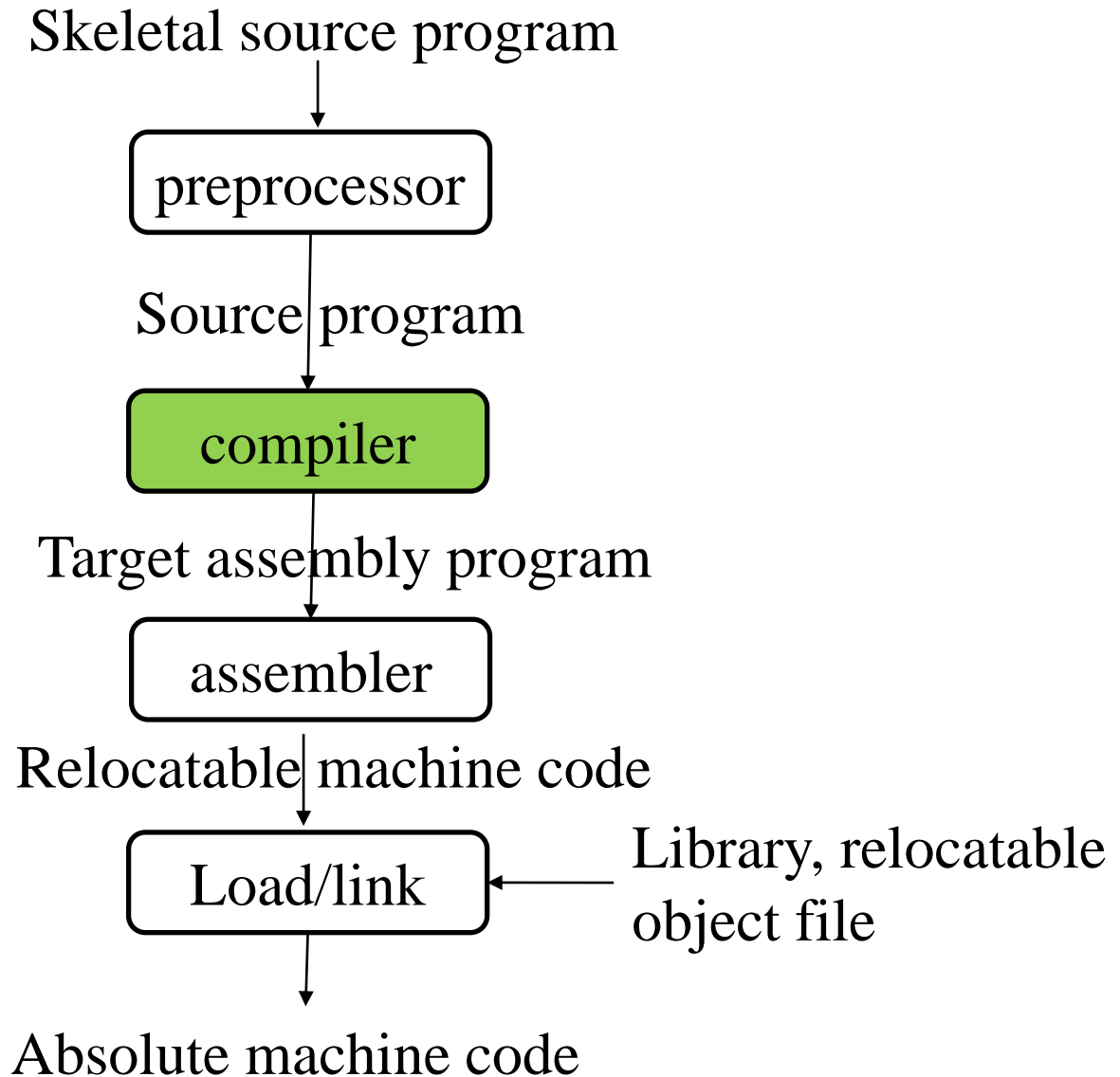
- A **compiler** is a program that takes a program written in a source language and translates it into an equivalent program in a target language.



# Other Applications

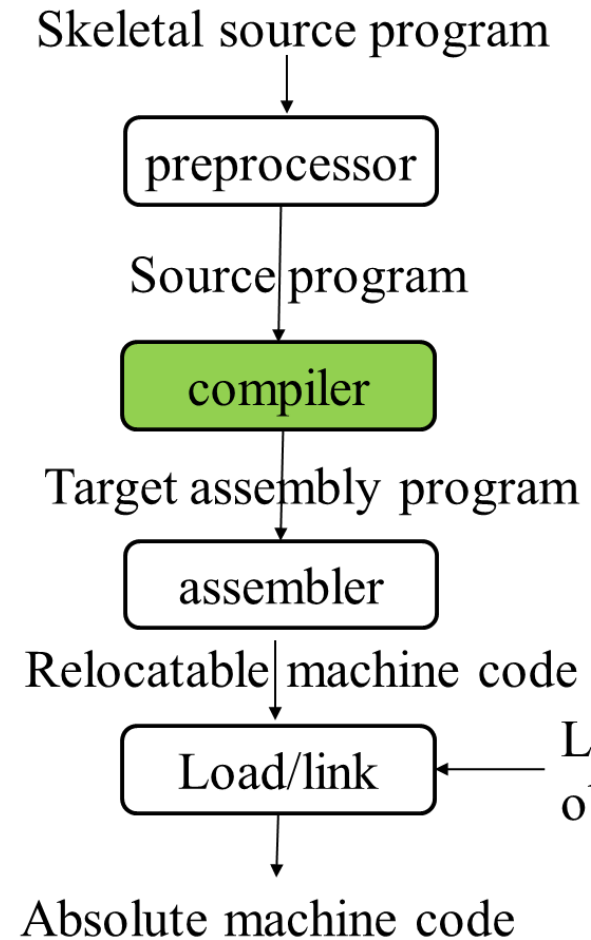
- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL (Structured Query Language).
  - Many software having a complex front-end may need techniques used in compiler design.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

# Compiling system



# Cousins of the compiler

- **Preprocessors:** delete comments, include other files, perform macro substitutions.
- **Compilers:** a language translator. It executes the source program immediately. Depending on the language in use and the situation
- **Assemblers:** a translator translates assembly language into object code
- **Linkers:**
  1. Collects code separately compiled or assembled in different object files into a file.
  2. Connects the code for standard library functions.
  3. Connects resources supplied by the operating system of the computer.





# Cousins of the compiler (cont.)

- **Loaders**

Relocatable : the code is not completely fixed .

Loaders resolve all relocatable address relative to the starting address.

- **Editors**

Produce a standard file ( structure based editors)

- **Debuggers**

Determine execution errors in a compiled program.

# Major Parts of Compilers

- There are two major parts of a compiler:

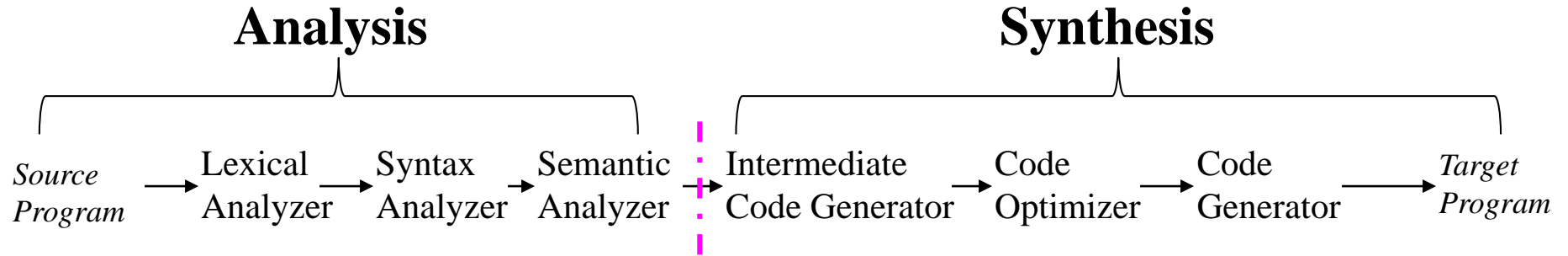
**Analysis and Synthesis**



**Front-end and Back-end**

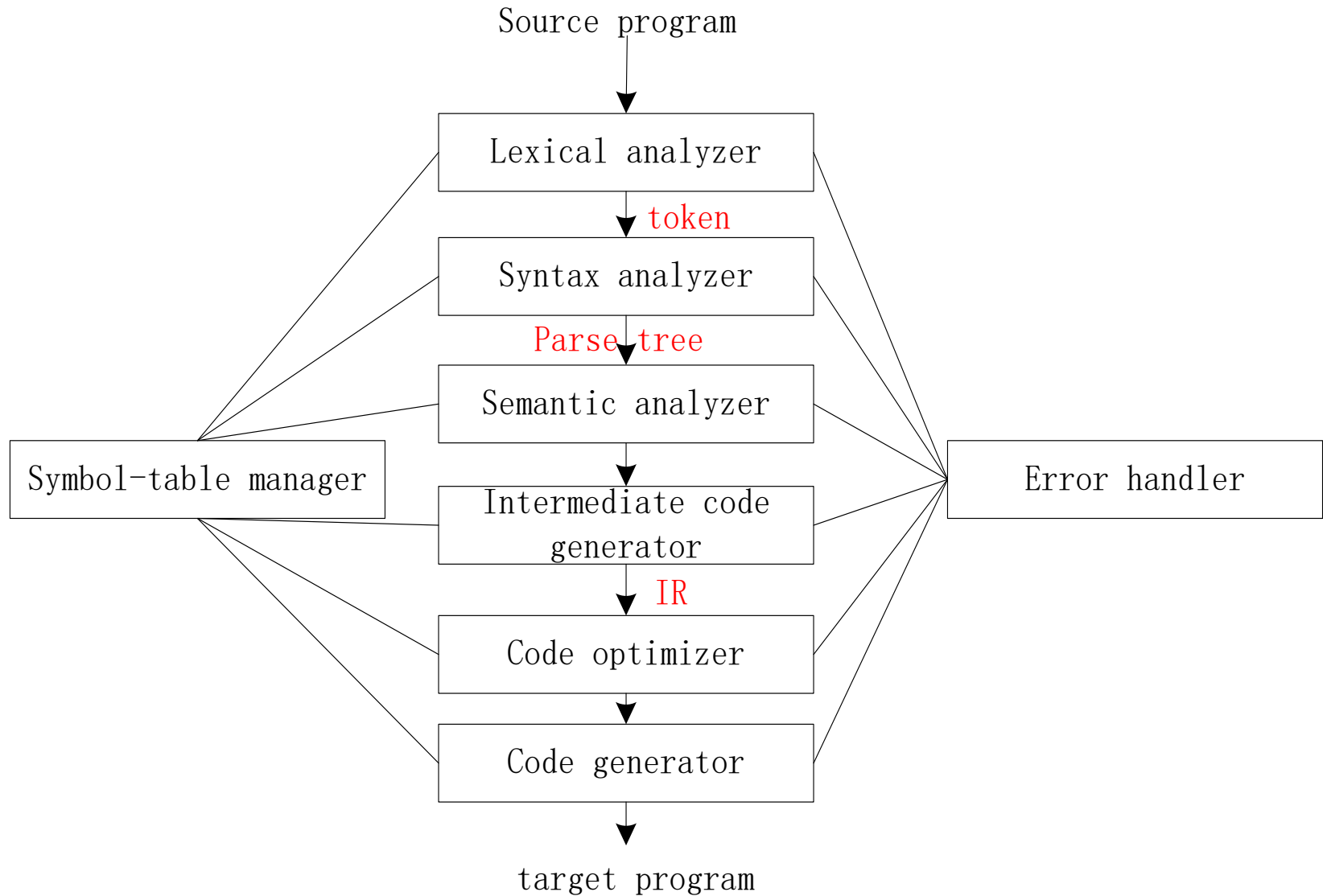
- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

# The Phases of a Compiler



- Each phase transforms the source program from one representation into another representation.
- They communicate with the symbol table
- They communicate with error handlers.

# The Phases of a Compiler (cont.)



# Lexical Analyzer

- First step: recognize words.
  - Smallest unit above letters

This is a sentence.

- Lexical analysis is not trivial.  
Consider:

ist his ase nte nce

# Lexical Analyzer (cont.)

- **Lexical Analyzer:** reads the source program character by character and returns the *tokens* of the source program.
- **Token** describes a pattern of characters having some meaning in the source program (such as identifiers, operators, keywords, numbers, delimiters and so on)

Ex: `x = y + 12 ;` => tokens: `x` and `y` are identifiers  
                                  `=` assignment operator  
                                  `+` add operator  
                                  `12` a number  
                                  `;` delimiter

Ex: `if x == y then z = 1; else z = 2;` => tokens: `x`, `y` and `z` are identifiers  
  `=` assignment operator  
  `==` compare operator  
  `1` and `2` are number  
  `;` delimiter

## Lexical Analyzer (cont.)

- Puts information about identifiers into the symbol table.
- **Regular expressions** are used to describe tokens (lexical constructs).
- A (Deterministic) **Finite State Automaton** can be used in the implementation of a lexical analyzer.

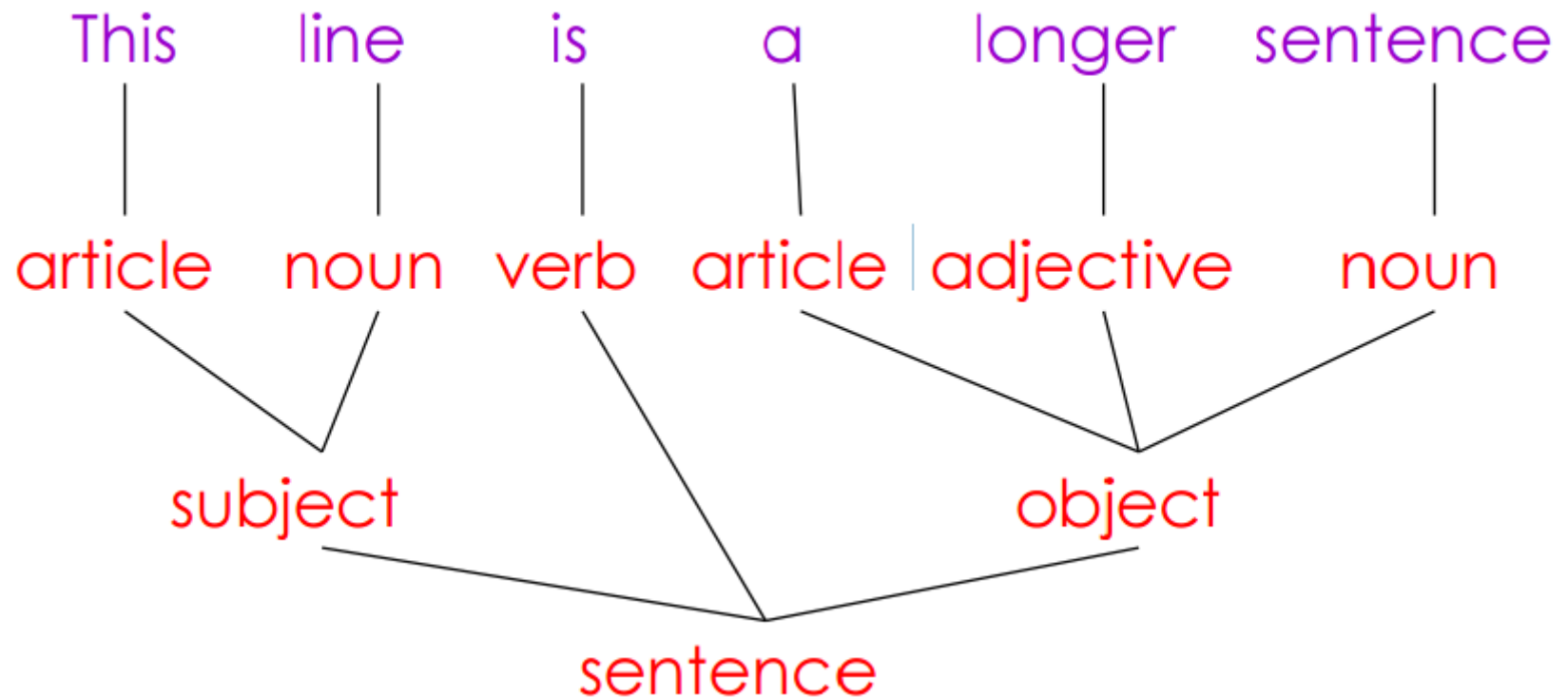
Eg. Identifier:  $[a-zA-Z][a-zA-Z0-9]^*$

Good Id: x, y, x1

Bad Id: 1x

# Syntax Analyzer

- Lexical Analyzer provides recognized words
- The next step is to understand sentence structure

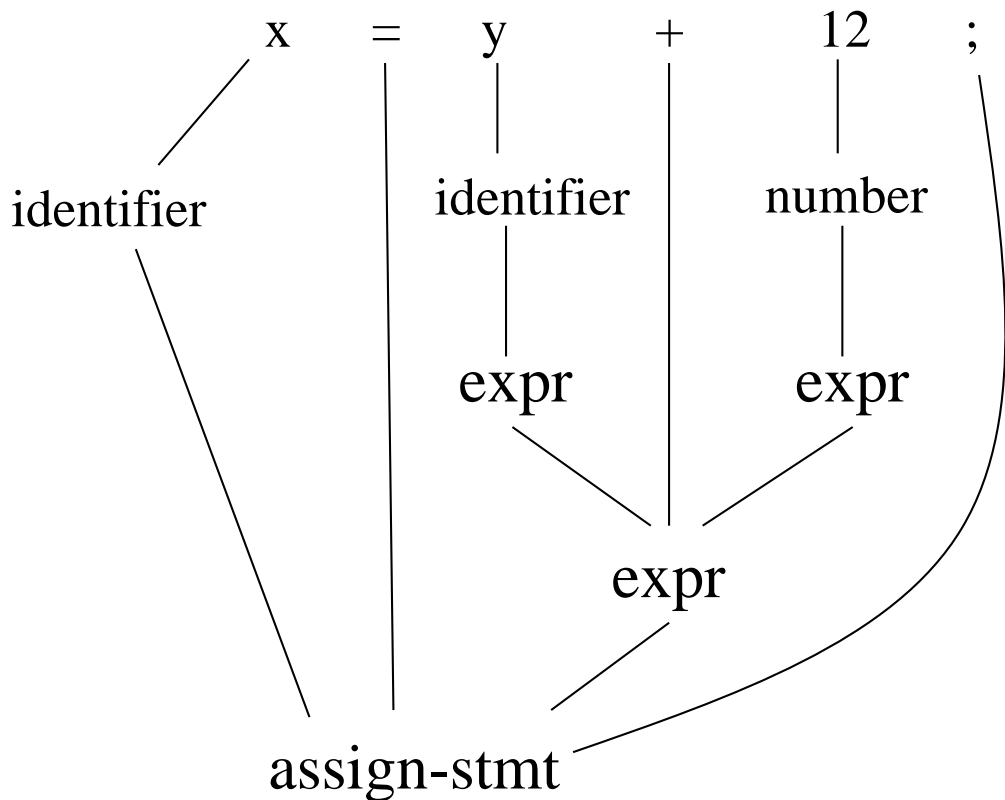




# Syntax Analyzer (cont.)

- A **Syntax Analyzer** (also known as **parser**) creates the syntactic structure (generally a **parse tree**) of the given program
- A **parse tree** describes a syntactic structure.

Ex:

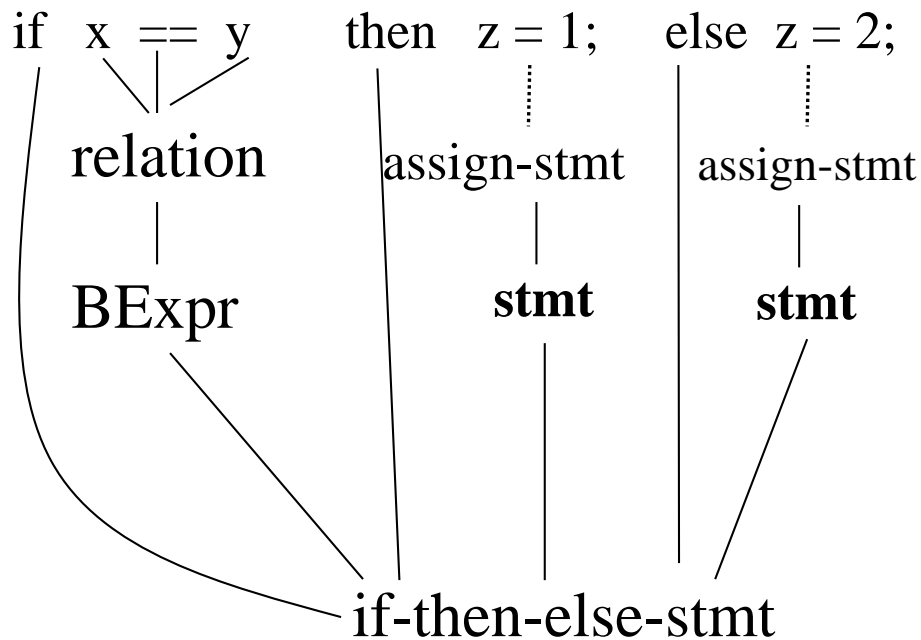


- In a parse tree, all **terminals** are at leaves.
- All inner nodes are **non-terminals** in a context free grammar.

# Syntax Analyzer (cont.)

- A **Syntax Analyzer** (also known as **parser**) creates the syntactic structure (generally a **parse tree**) of the given program
- A **parse tree** describes a syntactic structure.

Ex:



- In a parse tree, all **terminals** are at leaves.
- All inner nodes are **non-terminals** in a context free grammar.

# Syntax Analyzer (cont.)

- The syntax of a language is specified by **a context free grammar (CFG)**
- The rules in a CFG are mostly **recursive**.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.
- Ex: We use EBNF (Extended Backus-Naur Form) to specify a CFG
  - assign-stmt  $\rightarrow$  identifier = expr ;
  - expr  $\rightarrow$  identifier
  - expr  $\rightarrow$  number
  - expr  $\rightarrow$  expr BOP expr
  - BOP  $\rightarrow$  + | - | \* | /

Good:  $x = y + 1$ ; Bad:  $x = y + 1$ ,  $x = + y 1$

# Syntax Analyzer vs. Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with simple **non-recursive** constructs of the language.
  - The syntax analyzer deals with **recursive** constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (**tokens**) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures (**sentences**) in our programming language.

**Why not put them together in one analyzer?**

# Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - *Top-Down Parsing*
  - *Bottom-Up Parsing*

# Parsing Techniques (cont.)

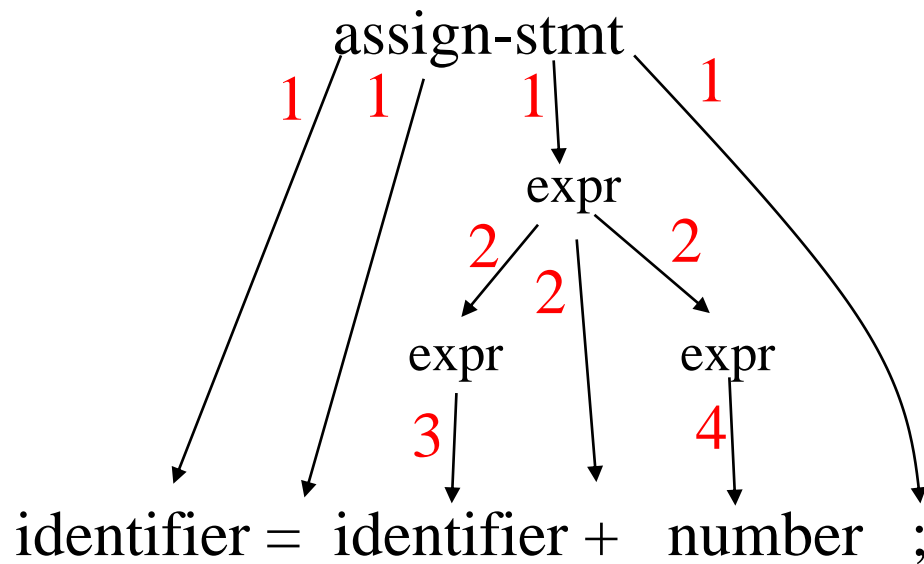
- Top-Down Parsing:
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing,
  - Non-Recursive Predictive Parsing (**LL Parsing**).  
(**L**-left to right; **L**-leftmost derivation)

# Parsing Techniques (cont.)

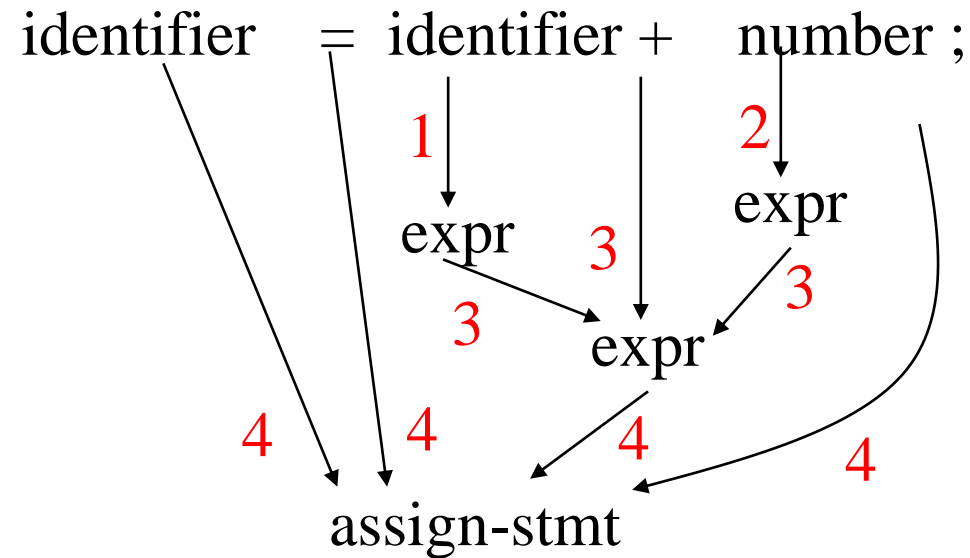
- Bottom-Up Parsing:
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as **shift-reduce parsing**
  - **LR Parsing** – much general form of shift-reduce parsing:  
**LR, SLR, LALR** (**L**-left to right; **R**-rightmost derivation)

# Top-down vs. Bottom-up

**x = y + 12 ;**



Top-down (preorder )



Bottom-up (postorder)



# Semantic Analyzer

- Once sentence structure is understood, we can try to understand “meaning”
  - But meaning is too hard for compilers
- Compilers perform limited analysis to catch inconsistencies

Eg.

Jack said Jerry left his assignment at home.

What does “his” refer to? Jack or Jerry?

Even worse:

Jack said Jack left his assignment at home.

How many Jacks are there? Which one left the assignment?

# Semantic Analyzer

- Programming languages define strict rules to avoid such ambiguities

```
Jack =1
def f():
    Jack =2
    print(Jack)
print(Jack)
f()
```

Output ?

>> 1

>> 2

# Semantic Analyzer (cont.)

- A **semantic analyzer** checks the source program for semantic errors and collects the type information for the code generation.
- **Type-checking** is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.

## Semantic Analyzer (cont.)

- Context-free grammars used in the syntax analysis are integrated with attributes (**semantic rules**)
  - the result is **a syntax-directed translation**
  - **Attribute grammars**

- Ex:

$$x = y + 12$$

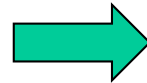
- The type of the identifier  $x$  must match with type of the expression  $(y+12)$

# Optimization

- No strong counterpart in English, but akin to editing
- Automatically modify programs so that they
  - Run faster
  - Use less recourse, e.g. power, memory
  - Small size

Eg.  $x = y + 0 \Rightarrow x = y$

$x = y + 1; z = x \Rightarrow z = y + 1$	
for (x=1; x++; x<=10)	for (x=1; x++; x<=10)
{ z=10; y = y + x *z }	{ y = y + x *10 }



The project has no optimization component

# Intermediate Code Generation

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture) independent (except GCC). But the level of intermediate codes is close to the level of machine codes.

# Intermediate Code Generation (example)

- Ex:

$x = y * fact + 1$



$id1 = id2 * id3 + 1$



MULT temp1, id2, id3     ;*Intermediate Codes*  
(*Quadraples or three address code*)

ADD temp2, temp1, #1

MOV id1, temp2

# Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.
- Ex:  $id1 = id2 * id3 + 1$   
( assume that we have an architecture with instructions whose at least one of its operands is a machine register)

ARM:

~~MOV~~ R1, R2, [id2]

~~MULT~~ R1, R3, [id3]

~~MULT~~ R1, R3, R1, R2

~~MOV~~ id1, R1, R3, #1

STR R1, [id1]

MULT temp1, id2, id3

ADD id1, temp1, #1





# Symbol-table management

- A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier
- In lexical analyzer the identifier is entered into the symbol table
- The attributes of an identifier cannot normally be determined during lexical analysis
- Updated and used by syntax analyzer, semantic analyzer, others

# The grouping of phases

- analysis and synthesis

**analysis:** lexical analysis , syntax analysis, semantic analysis  
(optimization)

**synthesis:** code generation (optimization)

- front end and back end

separation depend on the source language or the target language

the **front end**: the scanner, parser, semantic analyzer, intermediate code synthesis

the **back end**: the code generator, some optimization

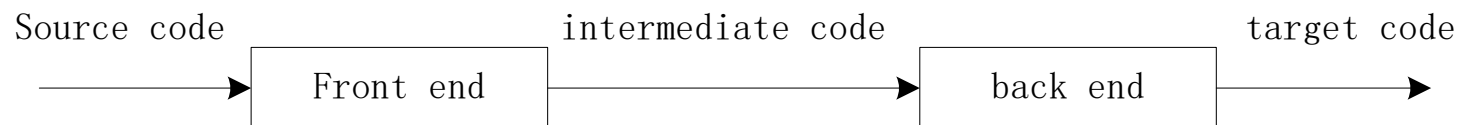


Fig. a. 1. 1

Lexical analyzer

syntax analyzer

<b>position</b>	...
<b>initial</b>	...
<b>rate</b>	...

semantic analyzer

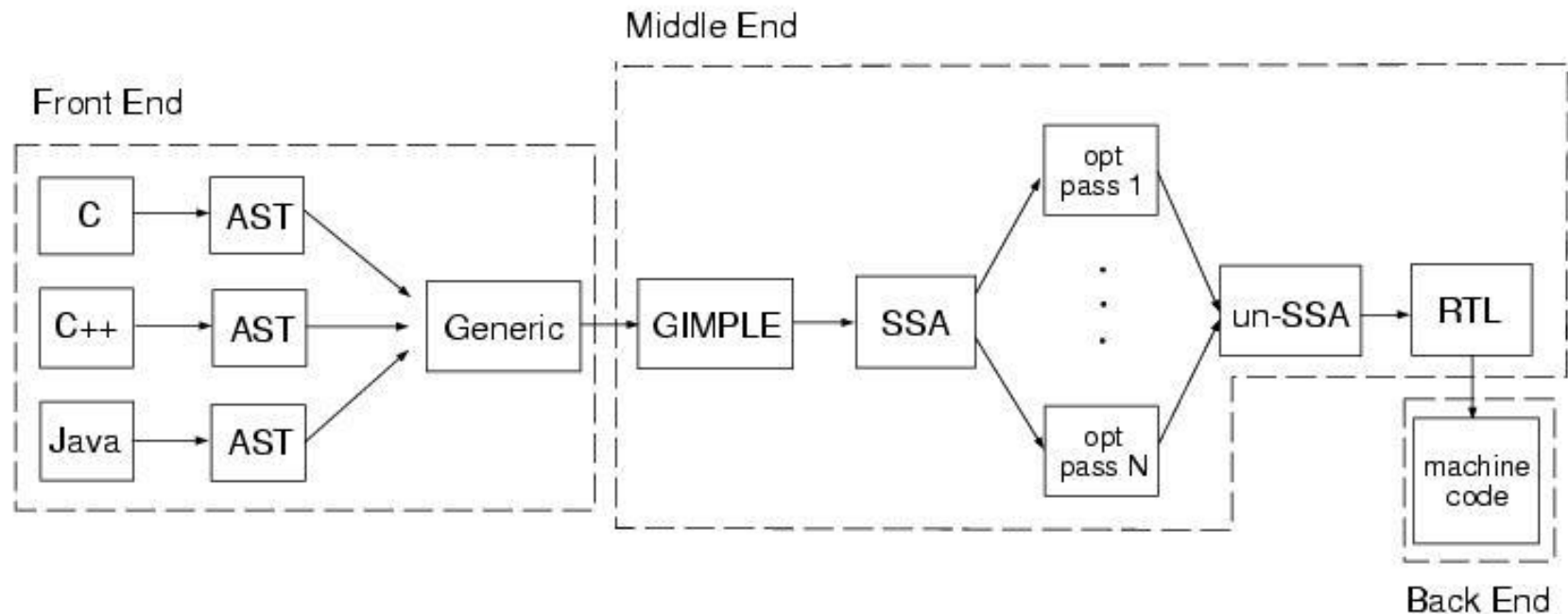
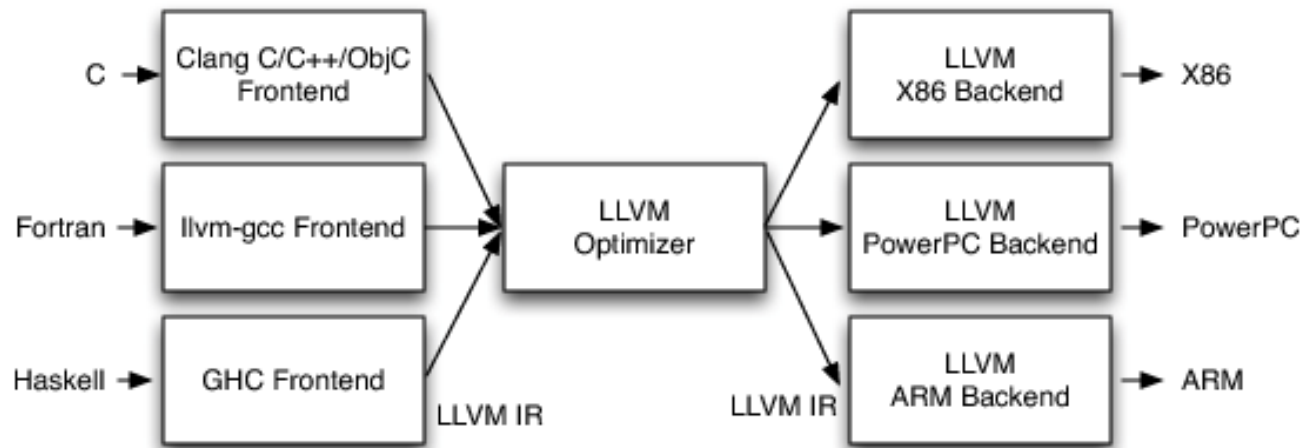
## Intermediate code generator

Code optimizer

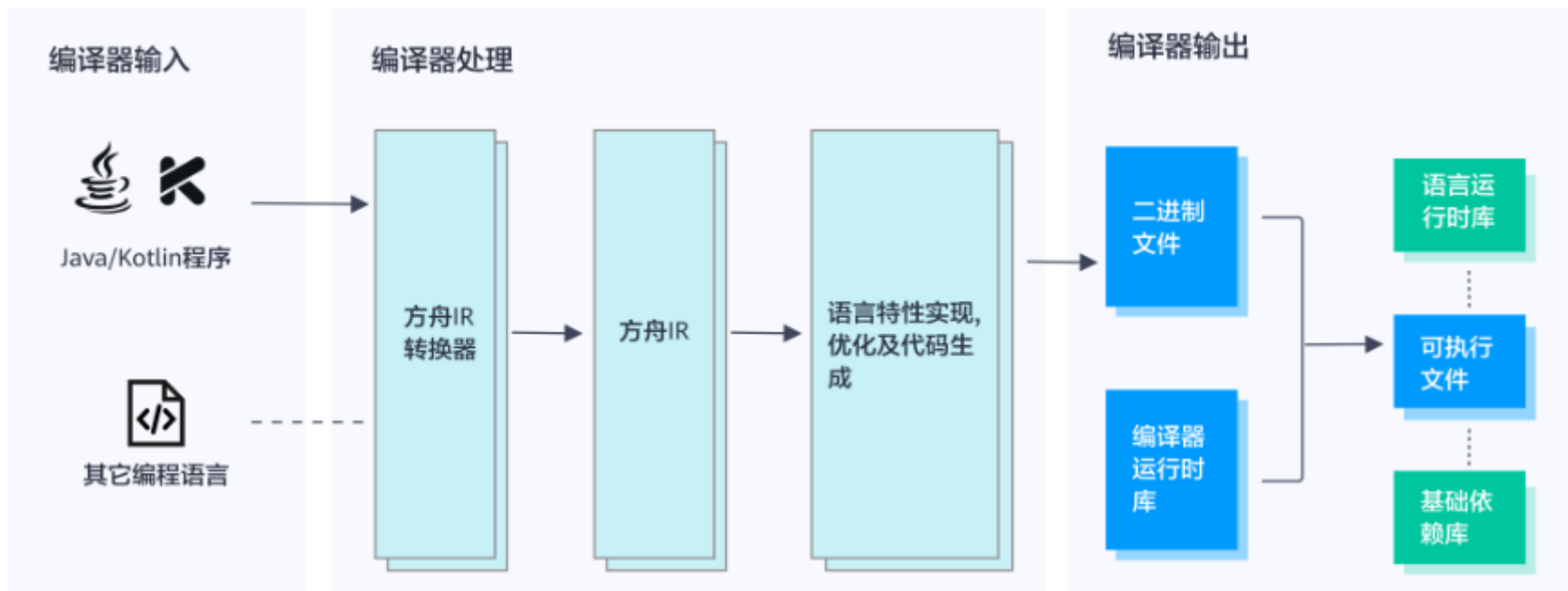
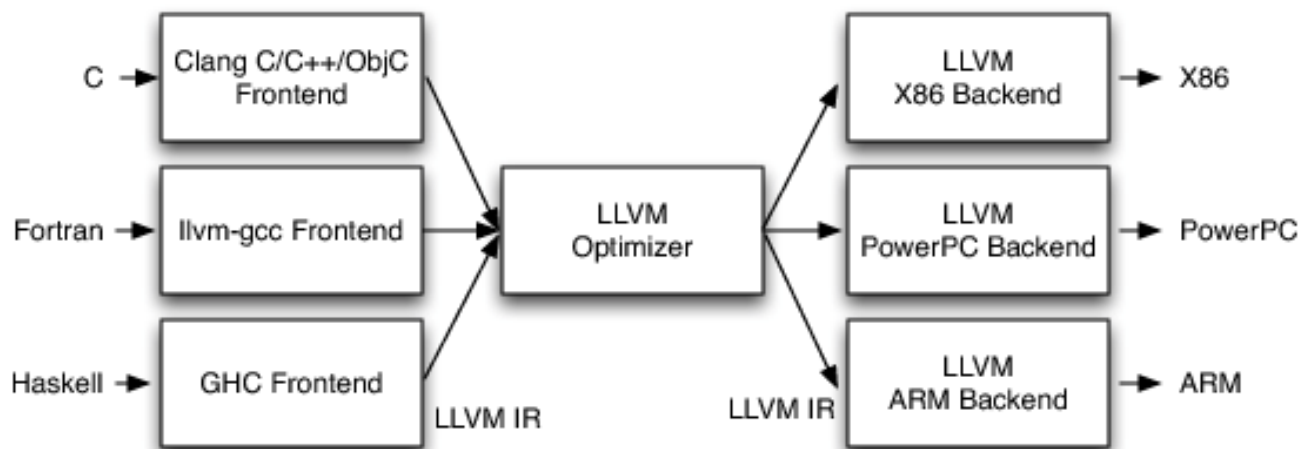
Code generator

```
for MIPS:
lw R2, id3
mult R2, R2, #60.0
lw R1, id2
add R1, R1, R2
sw R1, id1
```

# LLVM and GCC



# LLVM and Ark



# Issues

- Compiling is almost this simple, but there are many pitfalls.
- Each phase can encounter errors
- A phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- It is very difficult to proceed to analyze program or to correct errors.
- Language design has big impact on compiler
  - Determines what is easy and hard to compile
  - Course theme: many trade-offs in language design

# Summary

- the constitute of compiler and compiling system
- the concept of lexical analyzer, syntax analyzer, semantic analyzer, code generation, symbol table, intermediate representation, parse tree, token and so on.
- Regular expression and regular definition
- Context free grammar (CFG)