# CS 335: An Overview of Compilation

Swarnendu Biswas

Semester 2022-2023-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

# A Bit of History

- In the early 1950s, most programming was with assembly language
  - Led to low programmer productivity and high cost of software development
- In 1954, John Backus proposed a program that translated high level expressions into native machine code for IBM 704 mainframe
- Fortran (Formula Translator) I project (1954-1957): The first compiler was released

# Impact of Fortran

- Programmers were initially reluctant to use a high-level programming language for fear of lack of performance
- Fortran I compiler was the first optimizing compiler
- The Fortran compiler has had a huge impact on the field of programming languages and computer science
  - Many advances in compilers were motivated by the need to generate efficient Fortran code
  - Modern compilers preserve the basic structure of the Fortran I compiler!

# Executing Programs

- Programming languages are an abstraction for describing computations
  - For e.g., control flow constructs and data abstraction
  - Advantages of high-level programming language abstractions
    - Improved productivity, fast prototyping, improved readability, maintainability, and debugging

- The abstraction needs to be transferred to machine-executable form to be executed

# What is a Compiler?

- A compiler is a **system software** that **translates** a program in a source language to an **equivalent** program in a target language
  - System software (e.g., OS and compilers) helps application software to run

- Typical "source" languages might be C, C++, or Java
- The "target" language is **usually** the instruction set of some processor

# Important Features of a Compiler

- Generate **correct** code
- **Improve** the code according to some metric
- Provide **feedback** to the user, point out errors and potential mistakes in the program

# Source-Source Translators

- Produce a target program in another programming language rather than the assembly language of some processor
  - Also known as **transcompilers** or **transpilers**
  - TypeScript transpiles to JavaScript, and many research compilers generate C programs
- The output programs require further translation before they can be executed
- A typesetting program that produces PostScript can be considered a compiler
  - Typesetting LaTeX to generate PDF is compilation

# Transpiler vs Compiler

| Transpiler | Compiler |
|---|---|
| • Converts between programming languages at **approximately** the same level of abstraction | • A "**traditional**" compiler translates a higher level programming language to a lower level language |

# Interpreter

- An interpreter takes as input an executable specification and produces as output the result of executing the specification

- Scripting languages are often interpreted (e.g., Bash)

# Compilers vs Interpreters

| Compilers | Interpreters |
|---|---|
| • Translates the whole program at once | • Executes the program one line at a time<br>  • Compilation and execution happens at the same time |
| • Memory requirement during compilation is more | • Memory requirement is less, since there is less state to maintain |
| • Error reports are congregated | • Error reports are per line |
| • On an error, compilers try to fix the error and proceed past | • Stops translation on an error |
| • Examples: C, C++, and Java | • Examples: Bash and Python |

# More about Interpreters and Compilers

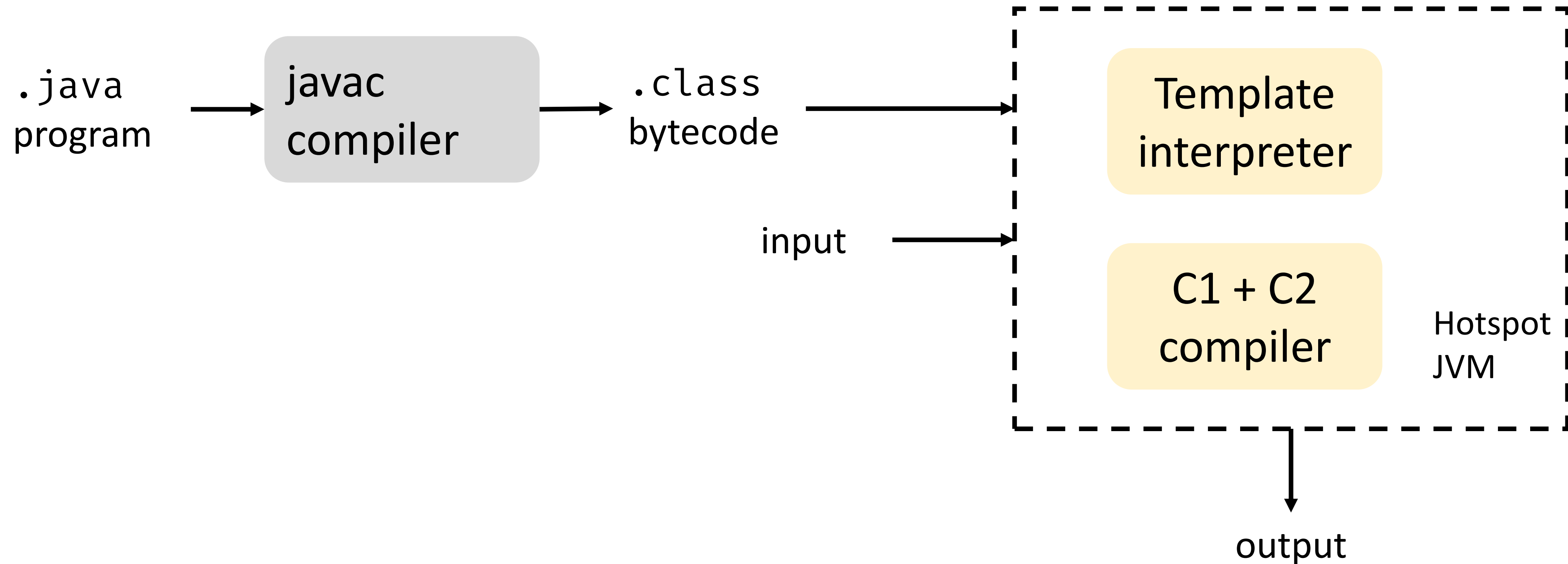
- Whether a language is interpreted or compiled is an **implementation-level detail**
  - If all implementations are interpreters, we say the language is interpreted
- Python is compiled to bytecode, and the bytecode is interpreted (CPython is the reference implementation)
  - Interpreting the bytecode is faster than interpreting a higher-level representation
  - PyPy both interprets and just-in-time (JIT) compiles the bytecode to optimized machine code at runtime

https://stackoverflow.com/questions/6889747/is-python-interpreted-or-compiled-or-both

# Hybrid Translation Schemes

- Translation process for a few languages include both compilation and interpretation (e.g., Lisp)
- Java is compiled from source code into a form called bytecode (`.class` files)
- Java virtual machines (JVMs) start execution by interpreting the bytecode
- JVMs usually also include a just-in-time compiler that compiles frequently-used bytecode sequences into native code
    - JIT compilation happens at runtime and is driven by profiling

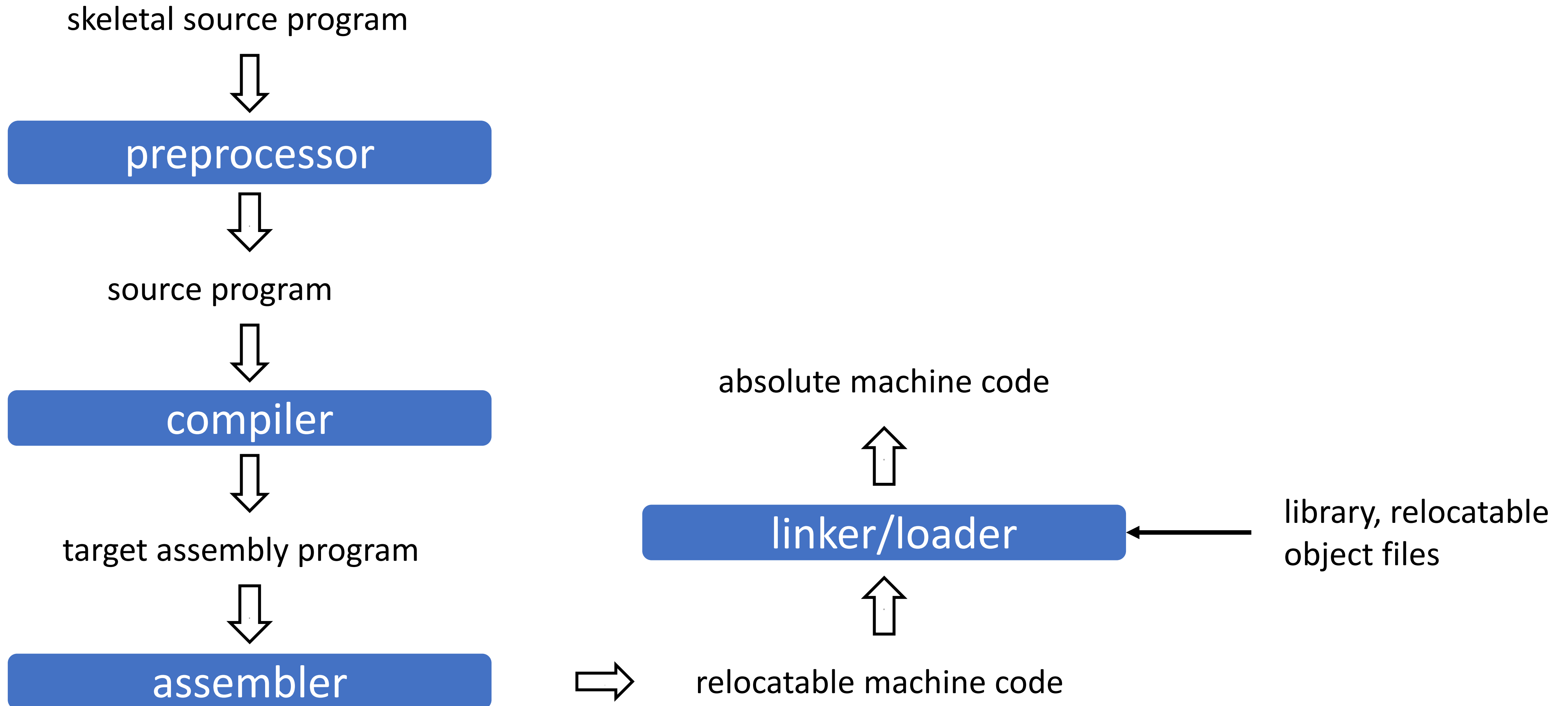
# Compilation Flow in Java with Hotspot JVM



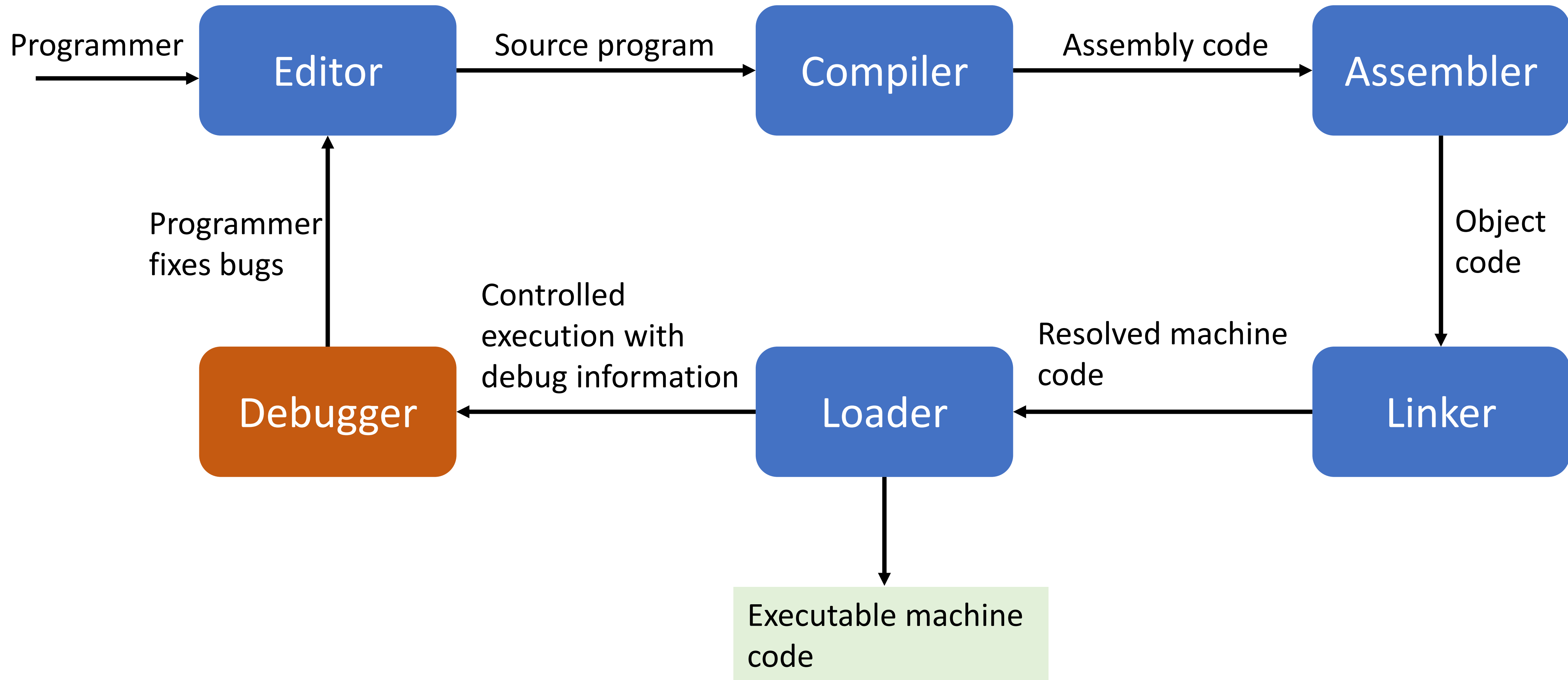
# Language Processing

- Language processing is an important component of programming
- A large number of system software and application programs require structured input
  - Command line interface in Operating Systems
  - Query language processing in Databases
  - Type setting systems like Latex

# A Language-Processing System



# Development Toolchain



# Goals of a Compiler

- A compiler must **preserve** the meaning of the program being compiled
  - Proving a compiler correct is a challenging problem and an active area of research
- A compiler must **improve** the input program in some discernible way
- Compilation time and space required must be reasonable
- The engineering effort in building a compiler should be manageable

# Applications of a Compiler

```
DO I = 1, N
  DO J = 1, M
    A(I,J+1) = A(I,J) + B
  ENDDO
ENDDO
```

# Applications of a Compiler

- Perform loop transformations to help with parallelization

```
DO I = 1, N
  DO J = 1, M
    A(I,J+1) = A(I,J) + B
  ENDDO
ENDDO
```

```
DO J = 1, M
  DO I = 1, N
    A(I,J+1) = A(I,J) + B
  ENDDO
ENDDO
```

# Programming Language vs Natural Language

- Natural languages
  - Interpretation of words or phrases evolve over time
    - E.g., “awful” meant worthy of awe and “bachelor” meant an young knight
  - Allow ambiguous interpretations
    - “I saw someone on the hill with a telescope.” or “I went to the bank.”
    - “Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo”

- Programming languages have well-defined structures and interpretations, and disallow ambiguity

---

https://en.wikipedia.org/wiki/Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo

# Constructing a Compiler

- A compiler is one of the most intricate software systems
  - General-purpose compilers often involve more than a hundred thousand LoC
- Very practical demonstration of integration of theory and engineering

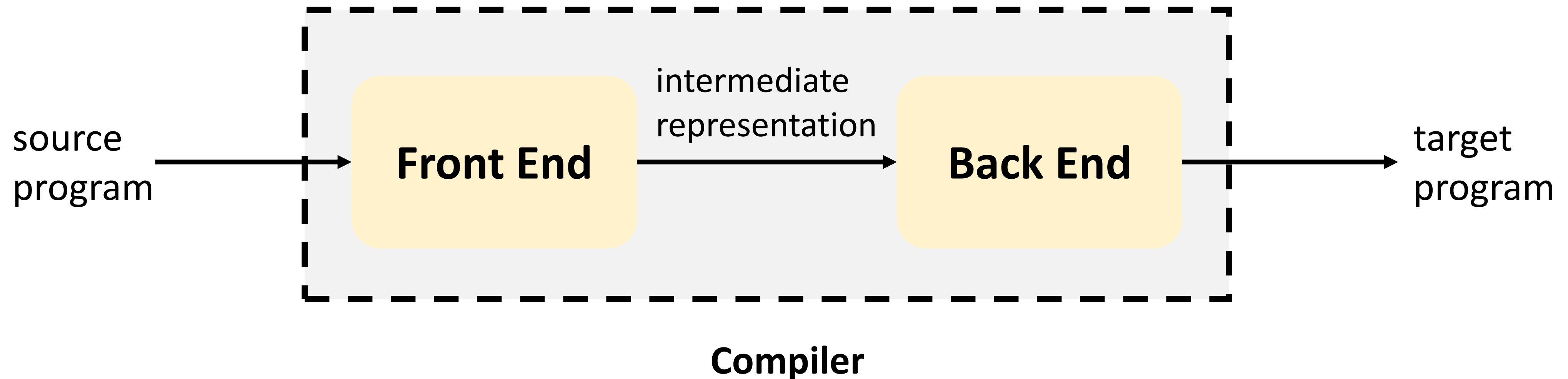
| Idea | Implementation |
| --- | --- |
| Finite and push-down automata | Lexical and syntax analysis |
| Greedy algorithms | Register allocation |
| Fixed-point algorithms | Dataflow analysis |
| … | … |

- Other practical issues such as concurrency and synchronization, and optimizations for the memory hierarchy and target processor

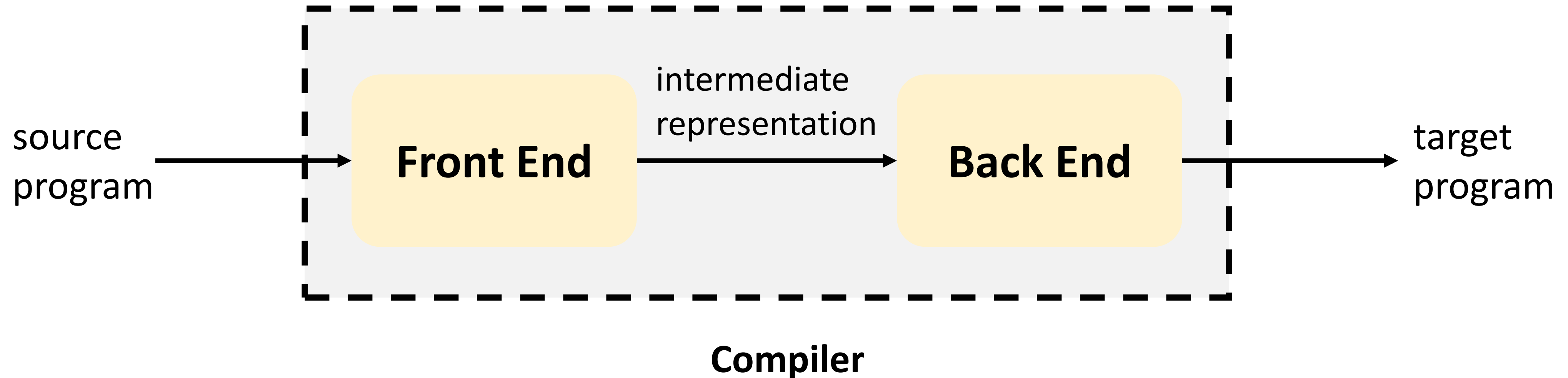
# Structure of a Compiler

# Compiler Structure

- A compiler interfaces with both the source language and the target architecture



# Compiler Structure




- Front end is responsible for **understanding the input program** in a source language
- Back end is responsible for **translating the input program** to the target architecture

# Intermediate Representation

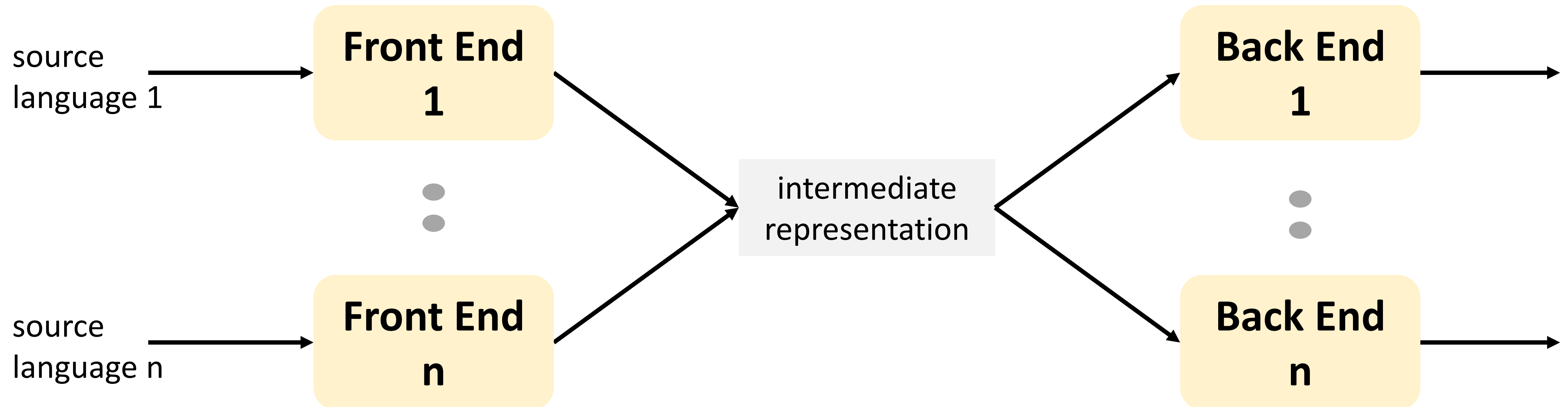
- An intermediate representation (IR) is a data structure to encode information about the input program
  - E.g., graphs, three address code, LLVM IR
- Different IRs may be used during different phases of compilation

```
int f(int a, int b) {
  return a + 2*b;
}

int main() {
  return f(10, 20);
}
```

```
define i32 @f(i32 %a, i32 %b) {
; <label>:0
    %1 = mul i32 2, %b
    %2 = add i32 %a, %1
    ret i32 %2
}

define i32 @main() {
; <label>:0
    %1 = call i32 @f(i32 10, i32 20)
    ret i32 %1
}
```

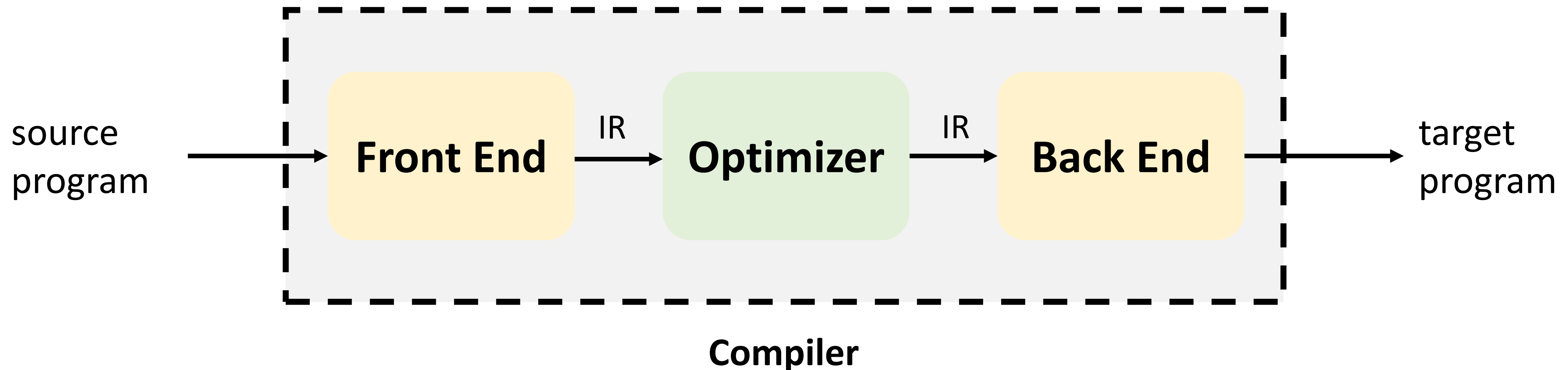
# Advantages of Two-Phased Compiler Structure

- Simplifies the process of writing or **retargeting** a compiler
  - Retargeting is the task of adapting the compiler to generate code for a new processor



# Three-Phased View of a Compiler

- IR makes it possible to add more phases to compilation

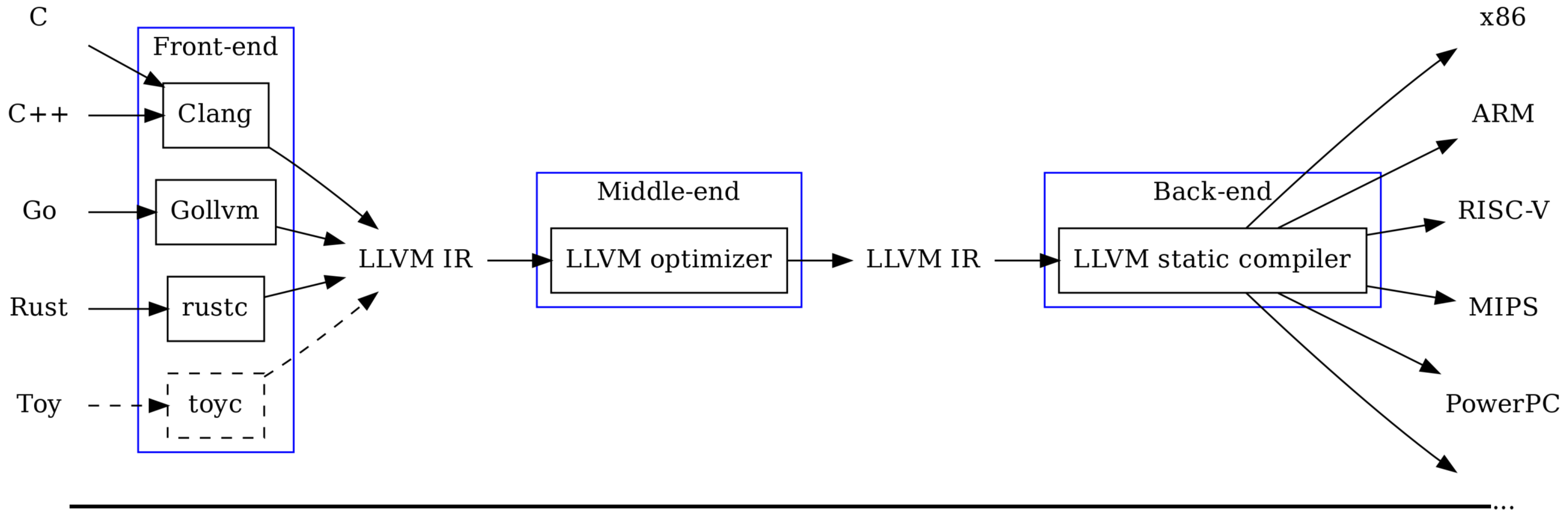



- Optimizer is an IR→IR transformer that tries to improve the IR program in some way

# Three-Phased View of a Compiler

- Front end consists of two or three passes that handle the details of the input source-language program
- Optimization phase contains many passes to perform different optimizations
  - The IR is generated by the front end
  - The number and purpose of these passes vary across compiler implementations
- The back end passes lower the IR representation closer to the target machine's instruction set

# Visualizing the LLVM Compiler System




https://blog.gopheracademy.com/advent-2018/llvm-ir-and-go/

# Implementation Choices

| Monolithic structure | Multipass structure |
| --- | --- |
| • Can potentially be more efficient, but is less flexible | • Less complex and easier to debug<br>• Can incur compile time performance penalties |

# Phases in a Compiler

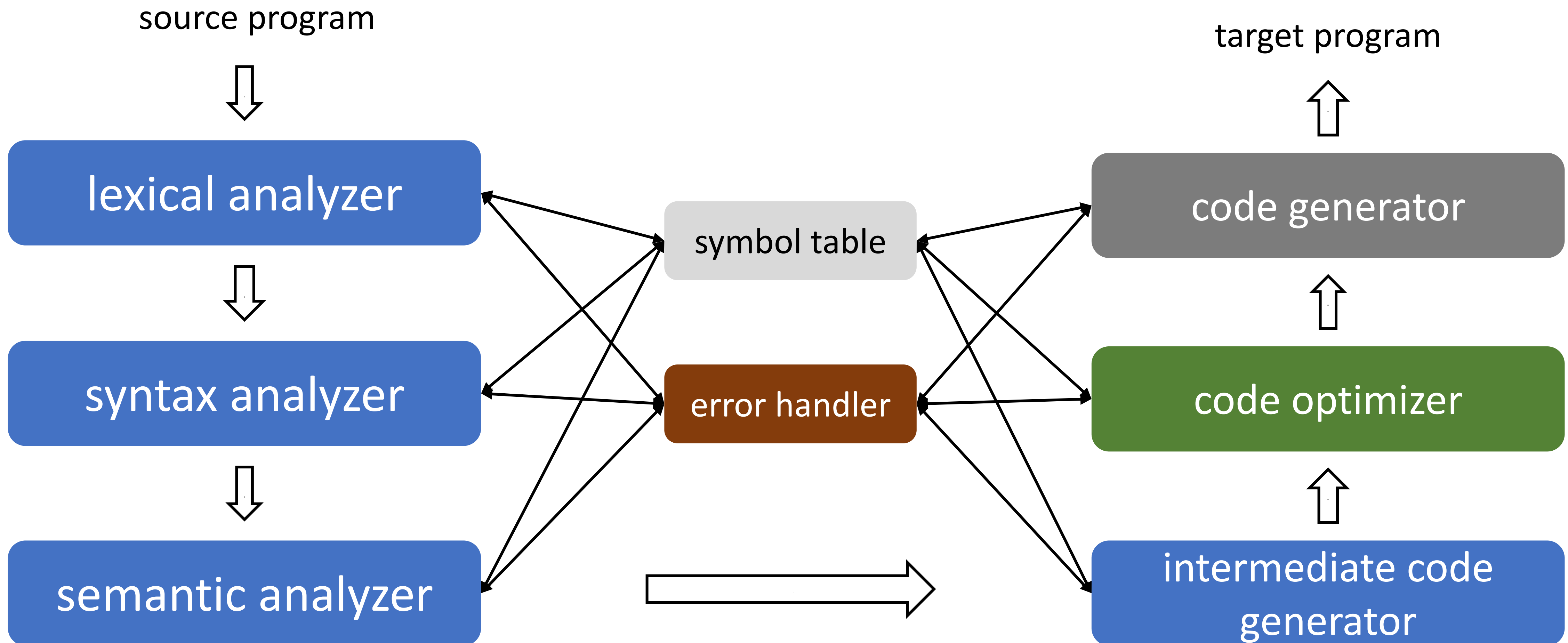
# Translation in a Compiler

- Direct translation from a high-level language to machine code is difficult
  - Mismatch in the abstraction level between source code and machine code
    - Compare abstract data types and variables vs memory locations and registers
    - Control flow constructs vs jump and returns
  - Some languages are farther from machine code than others
    - For example, languages supporting object-oriented paradigm

# Translation in a Compiler

- Translate in small steps, where each step handles a reasonably simple, logical, and well defined task
- Design a series of IRs to encode information across steps
  - IR should be amenable to program manipulation of various kinds (e.g., type checking, optimization, and code generation)
- IR becomes more machine specific and less language specific as translation proceeds

# Different Phases in a Compiler



# Front End

- First step in translation is to compare the input program structure with the language definition
  - Requires a formal definition of the language, in the form of regular expressions and context-free grammar
  - Two separate passes in the front end, often called the **scanner** and the **parser**, determine whether or not the input code is a valid program defined by the grammar

# Lexical Analysis

- Reads characters in the source program and groups them into a stream of **tokens** (or words)
  - Tokens represent a syntactic category
  - Character sequence forming a token is called a **lexeme**
  - Tokens can be augmented with the lexical value

```
position = initial + rate * 60
```

- Tokens are ID, “=”, ID, “+”, ID, “*”, CONSTANT

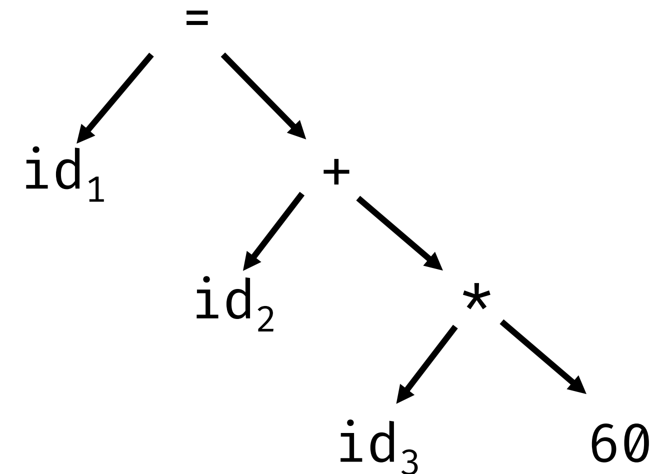
# Challenges in Lexical Analysis

- Identify word separators
  - The language must define rules for breaking a sentence into a sequence of words
  - Normally white spaces and punctuations are word separators in languages
  - In programming languages, a character from a different class may also be treated as a word separator

# Syntax Analysis

- Once words are formed, the next logical step is to understand the structure of the sentence
  - This is called syntax analysis or parsing
- Syntax analysis imposes a hierarchical structure on the token stream

```
position = initial + rate * 60
```



# Semantic Analysis

- Once the sentence is constructed, we need to interpret the meaning of the sentence

X saw someone on the hill with a telescope

JJ said JJ left JJ's assignment at home

- This is a very challenging task for a compiler
  - Programming languages define very strict rules to avoid ambiguities
  - For e.g., scope of variable named JJ

# Semantic Analysis

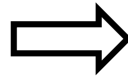
- Compiler performs other checks like type checking and matching formal and actual arguments

```
position = initial + “rate” * 60
```

# Intermediate Representation

- Once all checks pass, the front end generates an IR form of the code
  - IR is a program for an abstract machine

$id_1 = id_2 + id_3 * 60$



$t_1$ = inttofloat(60)
$t_2 = id_3 * t_1$
$t_3 = t_2 + id_2$
$id_1 = t_3$

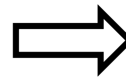
# Code Optimization

- Attempts to improve the IR code according to some metric
  - E.g., reduce the execution time, code size, or resource usage
- "Optimizing" compilers spend a significant amount of compilation time in this phase
- Most optimizations consist of an **analysis** and a **transformation**
  - Analysis determines where the compiler can safely and profitably apply the technique
    - Data flow analysis tries to statically trace the flow of values at run-time
    - Dependence analysis tries to estimate the possible values of array subscript expressions

# Code Optimization

- Some common optimizations
  - Common sub-expression elimination, copy propagation, dead code elimination, loop invariant code motion, strength reduction, and constant folding

$$
\begin{aligned}
t_1 &= \texttt{inttofloat(60)} \\
t_2 &= id_3 * t_1 \\
t_3 &= t_2 + id_2 \\
id_1 &= t_3
\end{aligned}
$$



$$
\begin{aligned}
t_1 &= id_3 * 60.0 \\
id_1 &= t_1 + id_2
\end{aligned}
$$

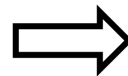
# Challenges with Code Optimization

- All strategies may not work for all applications
- Compiler may need to adapt its strategies to fit specific programs
  - Choice and order of optimizations
  - Parameters that control decisions & transformations
- Active research on “autotuning” or “adaptive runtime”
  - Compiler writer cannot predict a single answer for all possible programs
  - Use learning, models, or search to find good strategies

# Steps in Code Generation

- Back end traverses the IR code and emits code for the target machine
- First stage is instruction selection
  - Translate IR operations into target machine instructions
  - Can take advantage of the feature set of the target machine
  - Assumes infinite number of registers via virtual registers

```
t1 = id3 * 60.0
id1 = t1 + id2
```



```
MOVF id3 -> r2
MULF #60.0, r2 -> r2
MOVF id2 -> r1
ADDF r2, r1 -> r1
MOVF r1 -> id1
```

# Steps in Code Generation

- Register allocation
  - Decide which values should occupy the limited set of architectural registers
- Instruction scheduling
  - Reorder instructions to maximize utilization of hardware resources and minimize cycles

# Naïve Instruction Scheduling

LOAD/STORE take 3 cycles, MUL takes 2 cycles, and ADD takes 1 cycle.

```
LOAD  @ADDR1, @OFF1 -> R1
ADD   R1, R1 -> R1
LOAD  @ADDR2, @OFF2 -> R2
MUL   R1, R2 -> R1
LOAD  @ADDR3, @OFF3 -> R2
MUL   R1, R2 -> R1
STORE R1 -> @ADDR1, @OFF1
```

# Improved Instruction Schedule

LOAD/STORE take 3 cycles, MUL takes 2 cycles, and ADD takes 1 cycle.

```
LOAD  @ADDR1, @OFF1 -> R1
ADD   R1, R1 -> R1
LOAD  @ADDR2, @OFF2 -> R2
MUL   R1, R2 -> R1
LOAD  @ADDR3, @OFF3 -> R2
MUL   R1, R2 -> R1
STORE R1 -> @ADDR1, @OFF1
```

```
LOAD  @ADDR1, @OFF1 -> R1
LOAD  @ADDR2, @OFF2 -> R2
LOAD  @ADDR3, @OFF3 -> R3
ADD   R1, R1 -> R1
MUL   R1, R2 -> R1
MUL   R1, R3 -> R1
STORE R1 -> @ADDR1, @OFF1
```

# References


- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition.