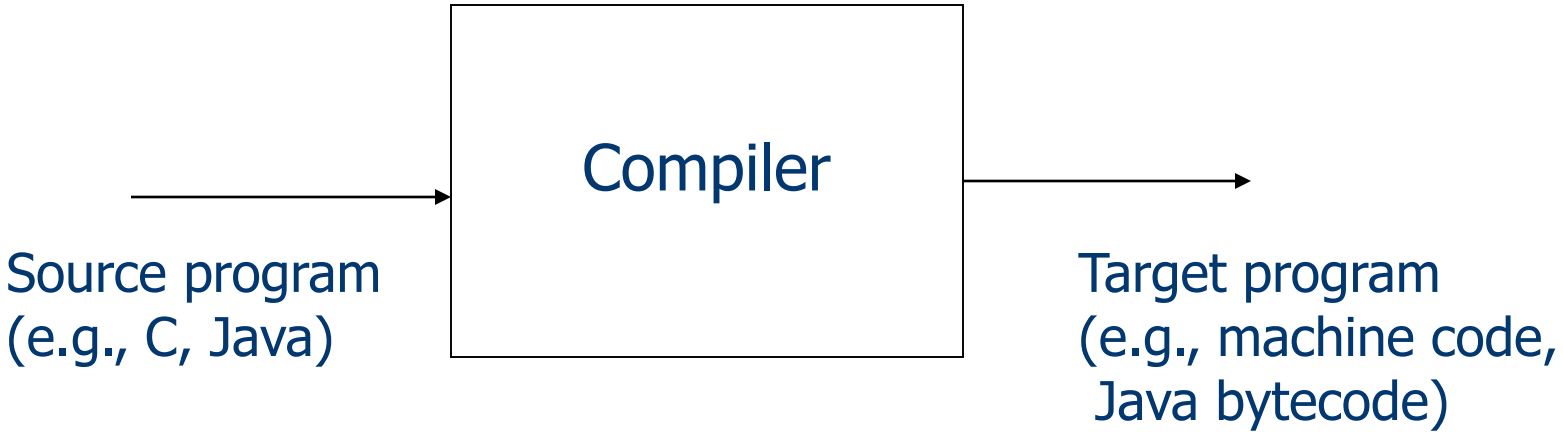
Overview

CIS*4650 (Winter 2020)

What is a compiler?

- A translator: also a validator and optimizer



- Allows a programmer to ignore machine-specific details.

History

- 1940's: stored program computer with direct machine code
- 1950's: assembly language with symbolic instructions
- mid-1950's: first compiled language Fortran (officially released in April 1957)
- 1950's-1960's: Chomsky language hierarchy with classes of languages and grammars (e.g., regular expressions, CFG's)
- 1960's-1970's: scanning and parsing tools for recognizing regular expressions and CFG's

Advantages of Compiled Languages

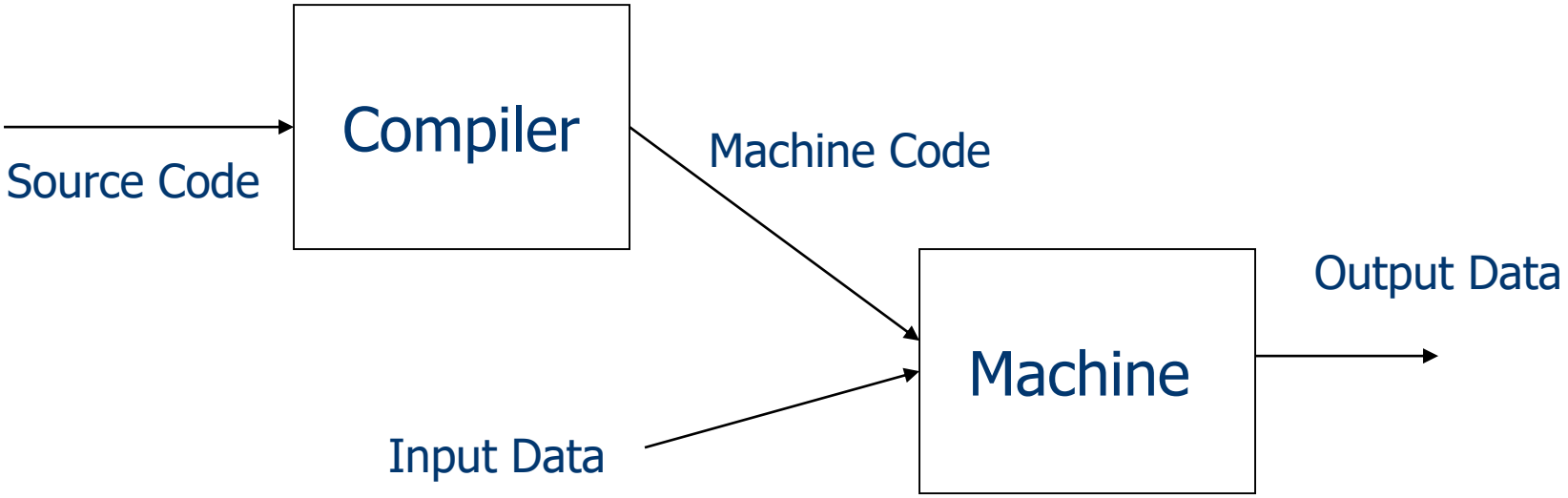
- Efficiency: producing efficient object code (optimization)
- Convenience: reducing low-level complexity
- Complexity: increasingly more complex with supports for object-oriented programming such as encapsulations, inheritance, and polymorphism
- Retargetability: single source language to multiple target languages

Programming Paradigms

- Imperative languages: sequential and explicit execution (e.g., C and Pascal)
- Functional languages: functions calling other functions (e.g., Lisp)
- Logical languages: rules in no specific order executed with the built-in backtracking mechanism (e.g., Prolog)
- Object-oriented languages: typically extended from imperative languages (e.g., C++, Java)

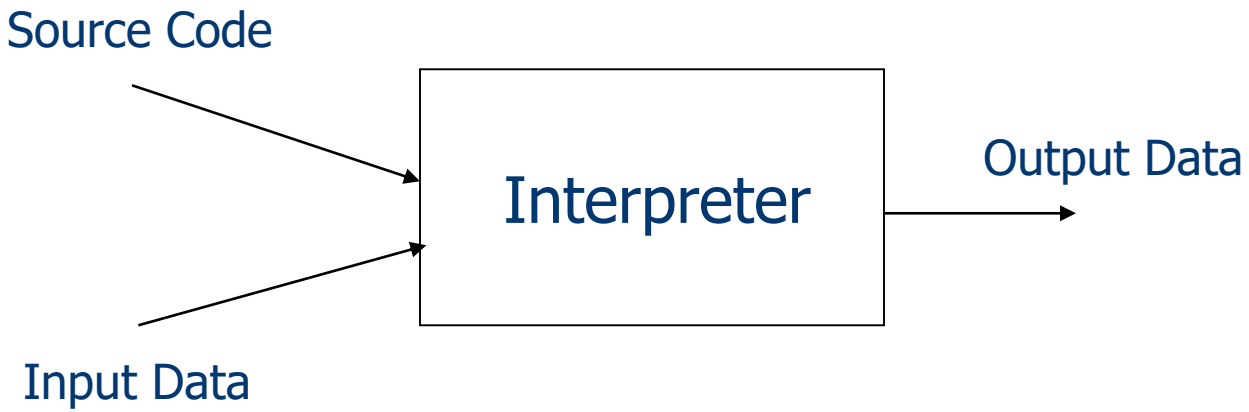
Implementation Methods

- Compilation (e.g., C/C++)
  - Slow translation from source code to machine code
  - Fast execution (one compilation and multiple executions)
  - Additional effort in porting source code



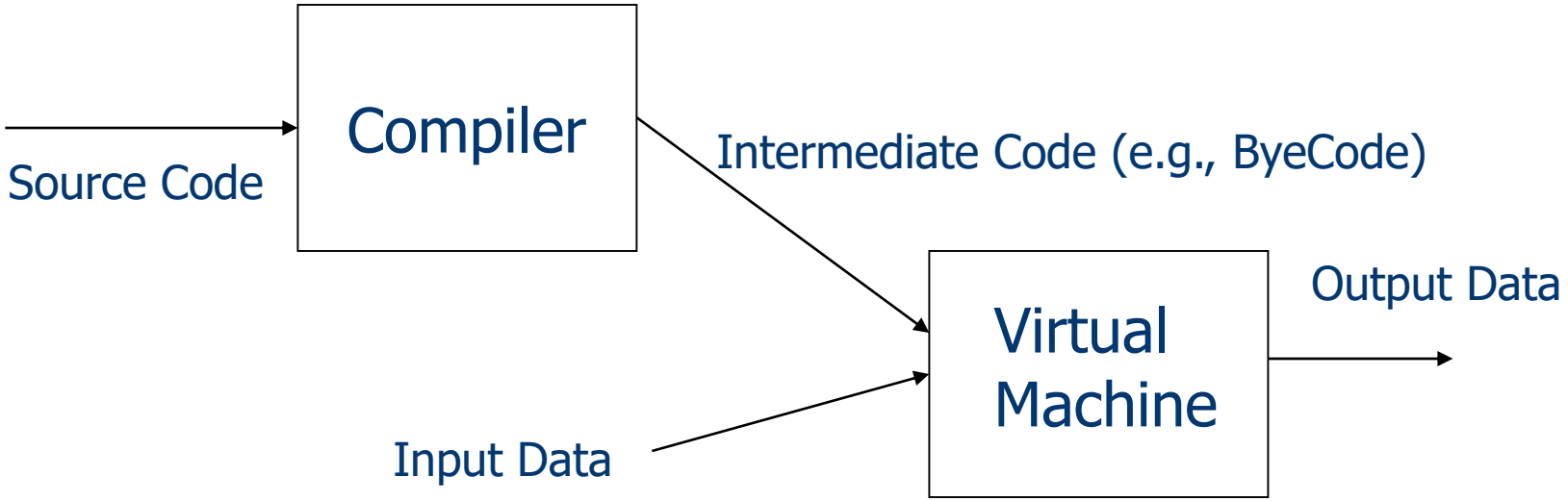
Implementation Methods

- Interpretation (e.g., Lisp, Prolog)
  - Small translation cost
  - Slow execution
  - Highly portable code

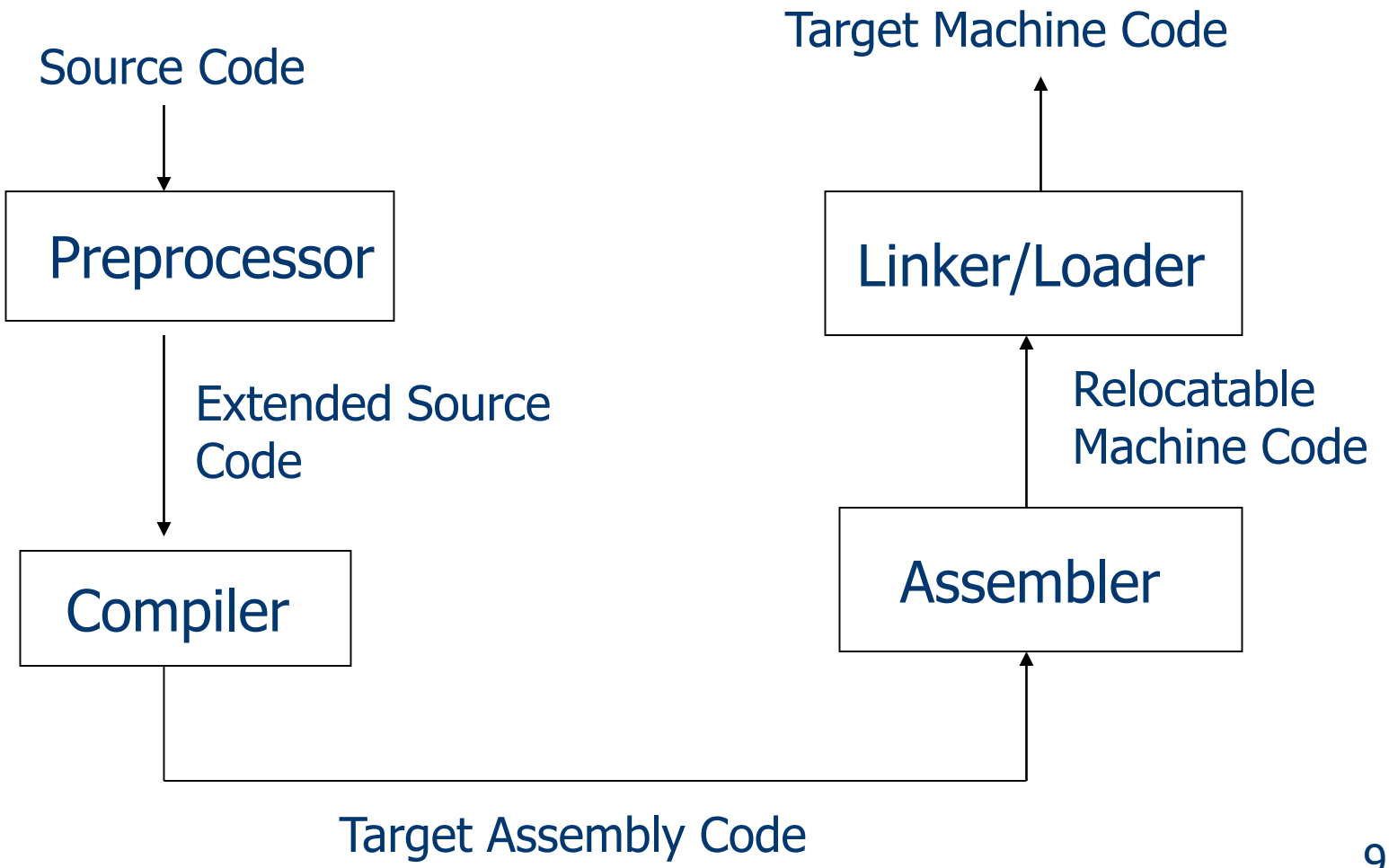


Implementation Methods

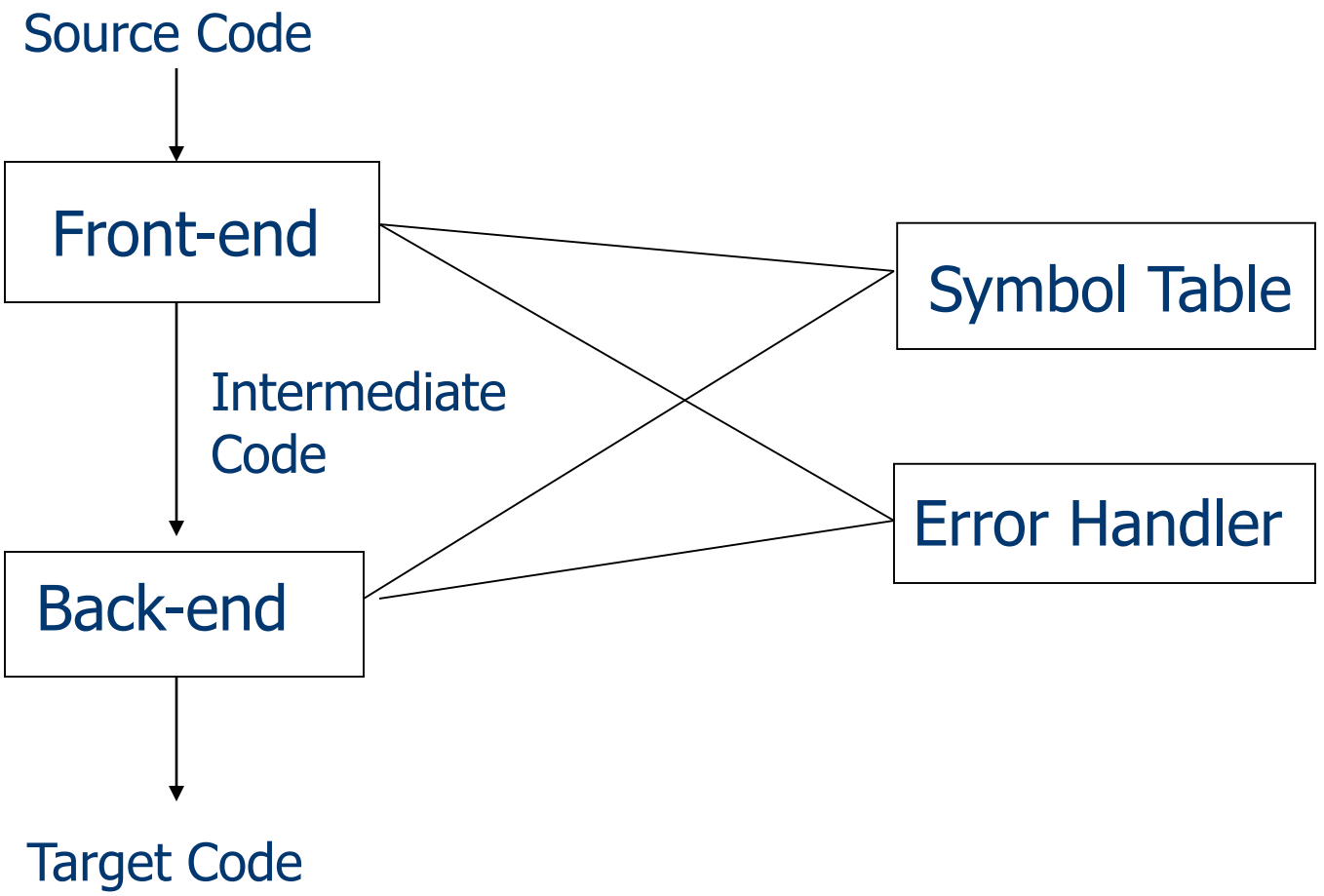
- Hybrid systems (e.g., Java)
  - Medium translation cost
  - Medium execution speed
  - Highly portable code



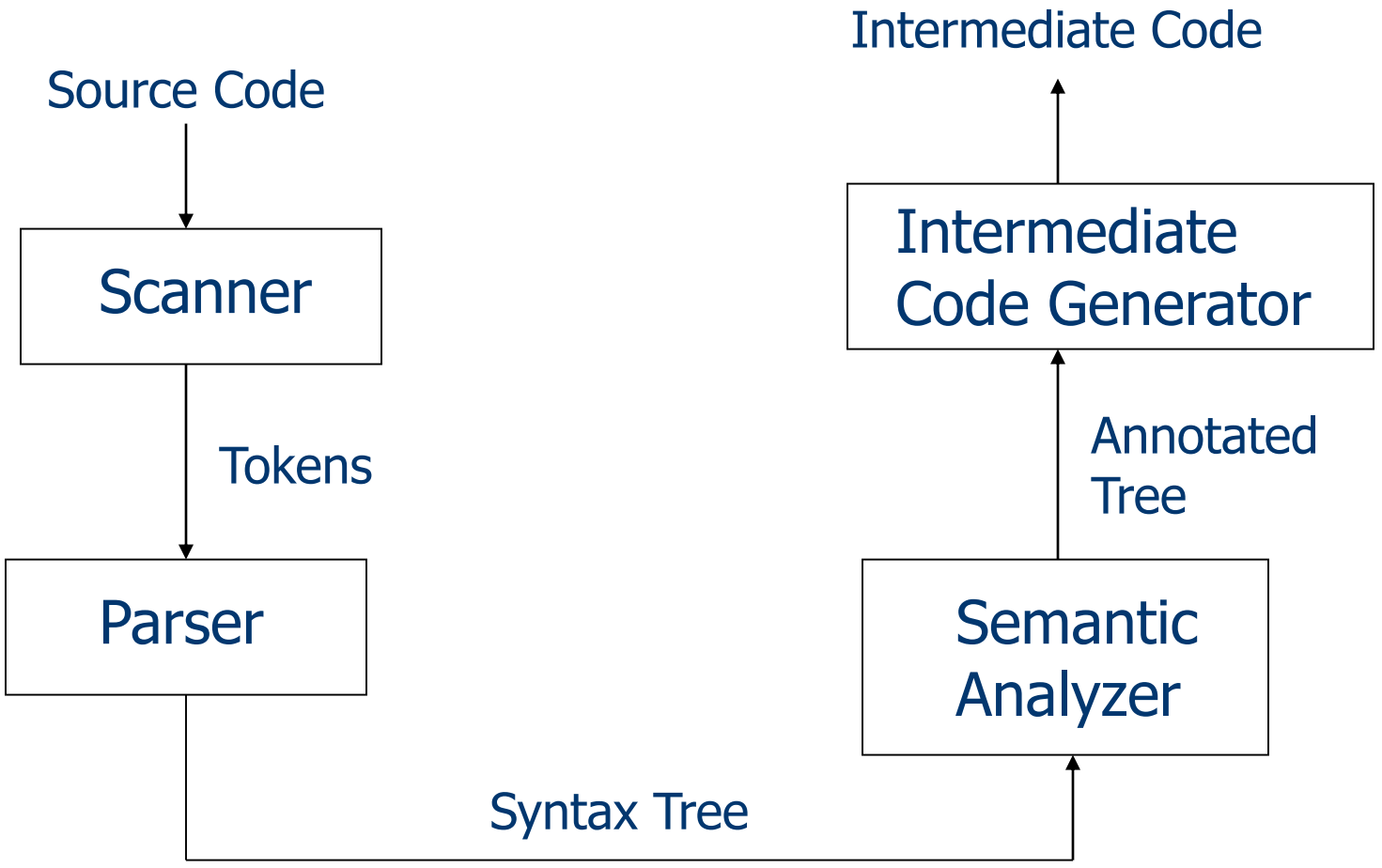
A Complete System



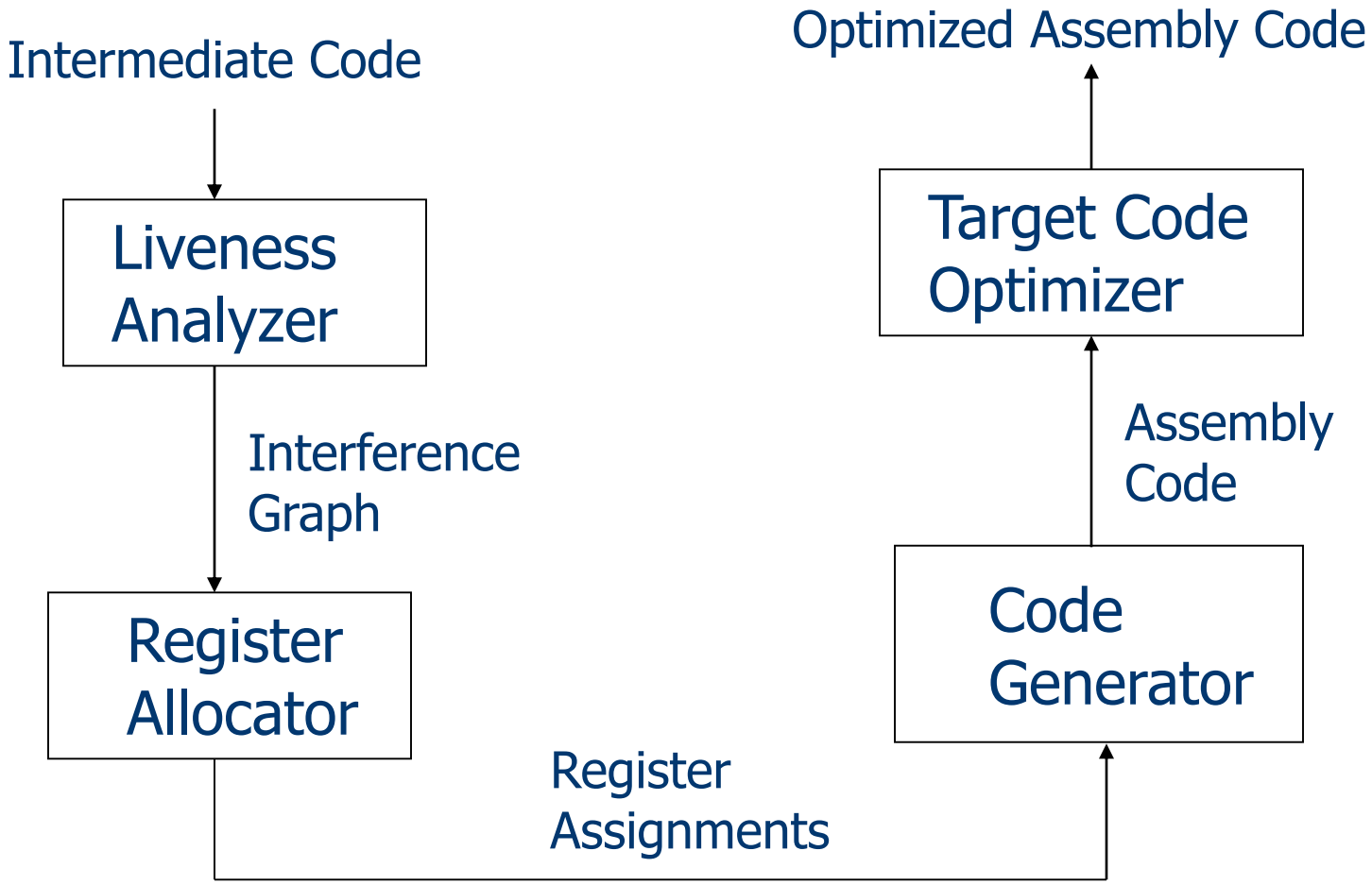
Phases of a Compiler



Front-End Analysis



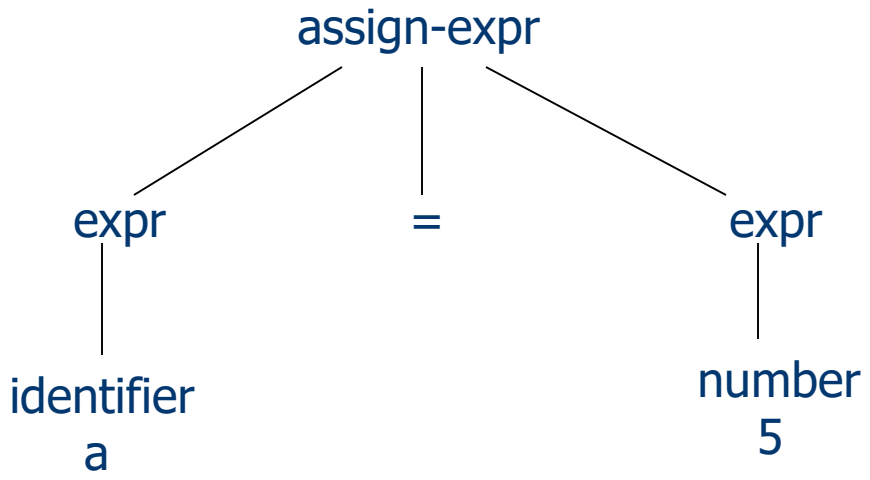
Back-End Synthesis



Scanning and Parsing

- Scanning/Lexical Analysis: break up source program into tokens (or words)
- Parsing/Syntactic Analysis: analyze ordering of tokens (phrase structure or syntax tree)

e.g.: a = 5

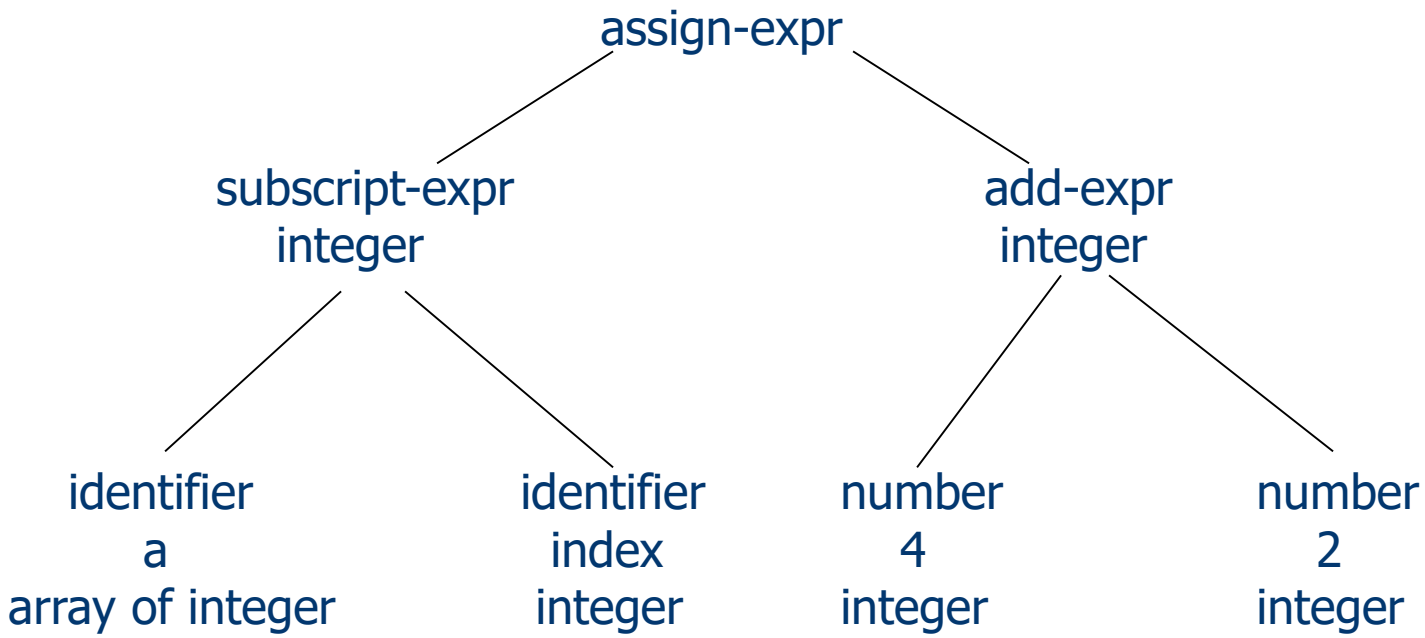


Semantic Analysis

- Attach meanings to phrases
- Relate symbols to their definitions
- Type checking of expressions

Semantic Analysis

- Annotated Tree: e.g., a[index] = 4 + 2

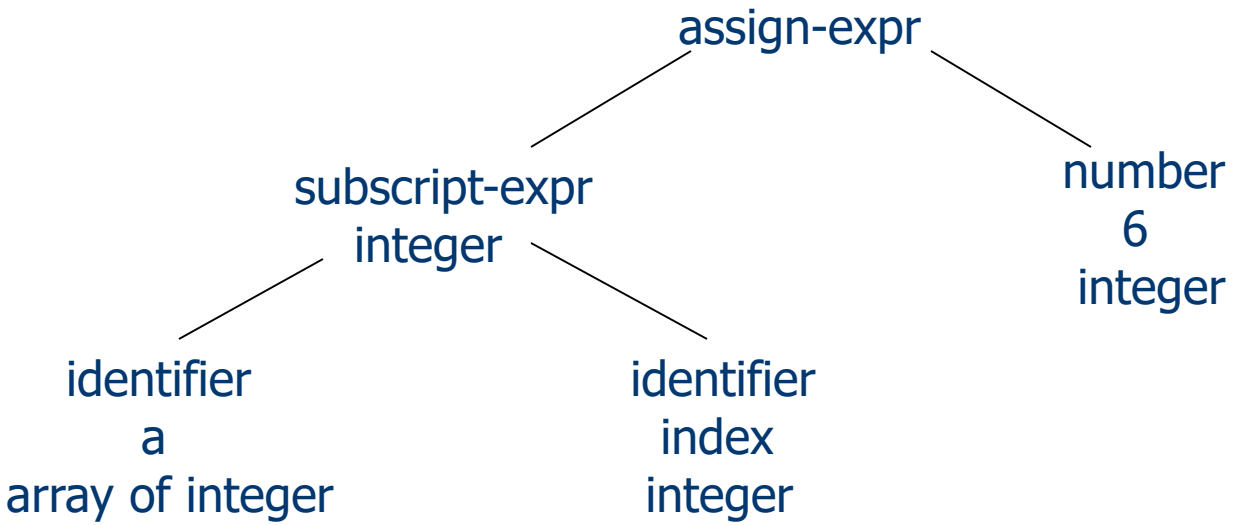


Intermediate Code Generation

- Layout stack frames
  - Variables and parameters in different scopes or activation records
- Produce intermediate code
  - Abstract syntax tree may still be complex
  - Need a linear representation to accommodate code generation
- Intermediate code optimization

Intermediate Code Optimizer

- Optimized tree:



- Three-address code:

```
t = 4 + 2                t = 6                a[index] = 6
a[index] = t    ⟶        a[index] = t    ⟶
```

Code Generator

- Convert intermediate code into target assembly/machine code:

```
MOV   R0, index      ;; value of index -> R0
MUL   R0, 2          ;; double value in R0
MOV   R1, &a         ;; address of a -> R1
ADD   R1, R0         ;; add R0 to R1
MOV   *R1, 6         ;; constant 6 -> address in R1
```

- Further optimized code:

```
MOV   R0,  index     ;; value of index -> R0
SHL   R0             ;; double value in R0
MOV   &a[R0], 6      ;; constant 6 -> address a + R0
```