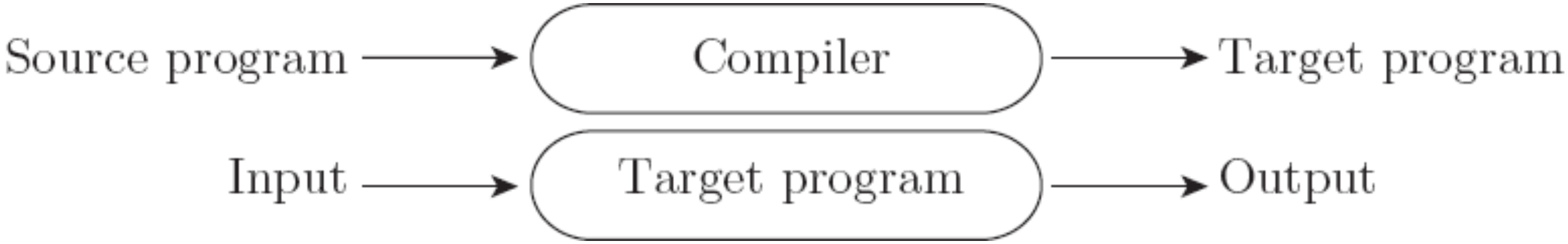
The Design and Implementation of Programming Languages

In Text: Chapter 1

Language Implementation Methods

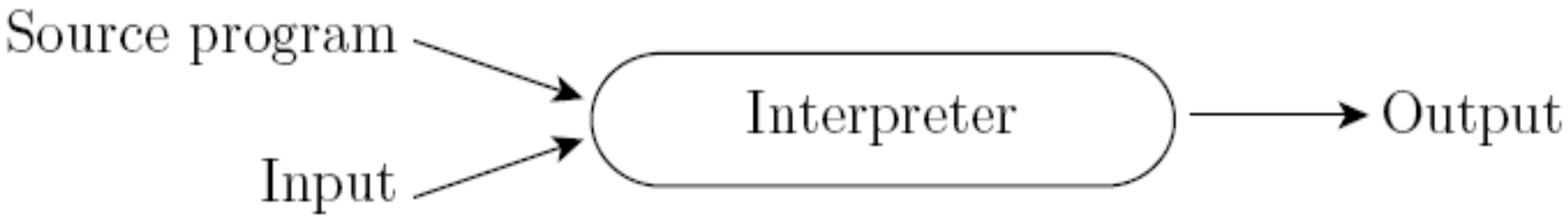
- Compilation
- Interpretation
- Hybrid

Compilation



- Translate high-level programs to machine code
- Slow translation
- Fast execution

Interpretation

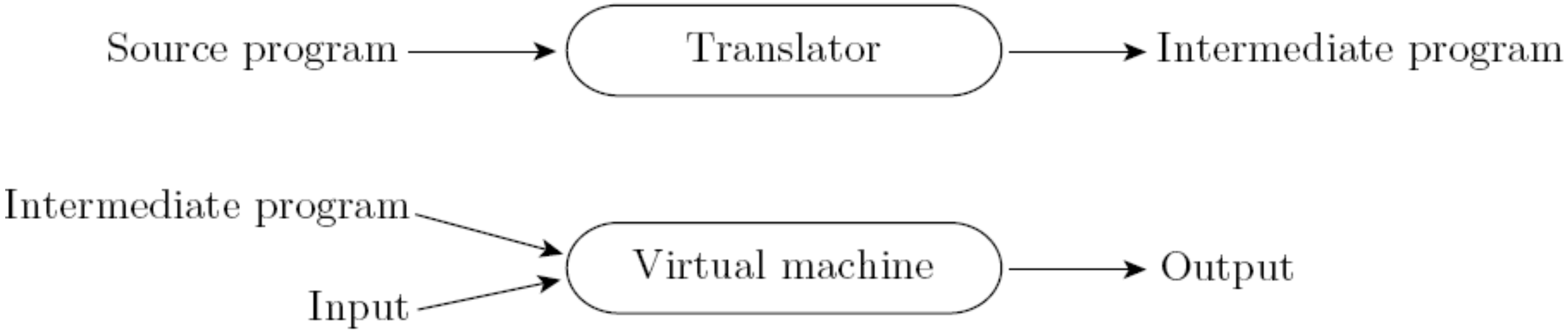


- Interpret one statement and then execute it on a virtual machine
- No translation
- Slow execution
- E.g., Basic

Compilation vs. Interpretation

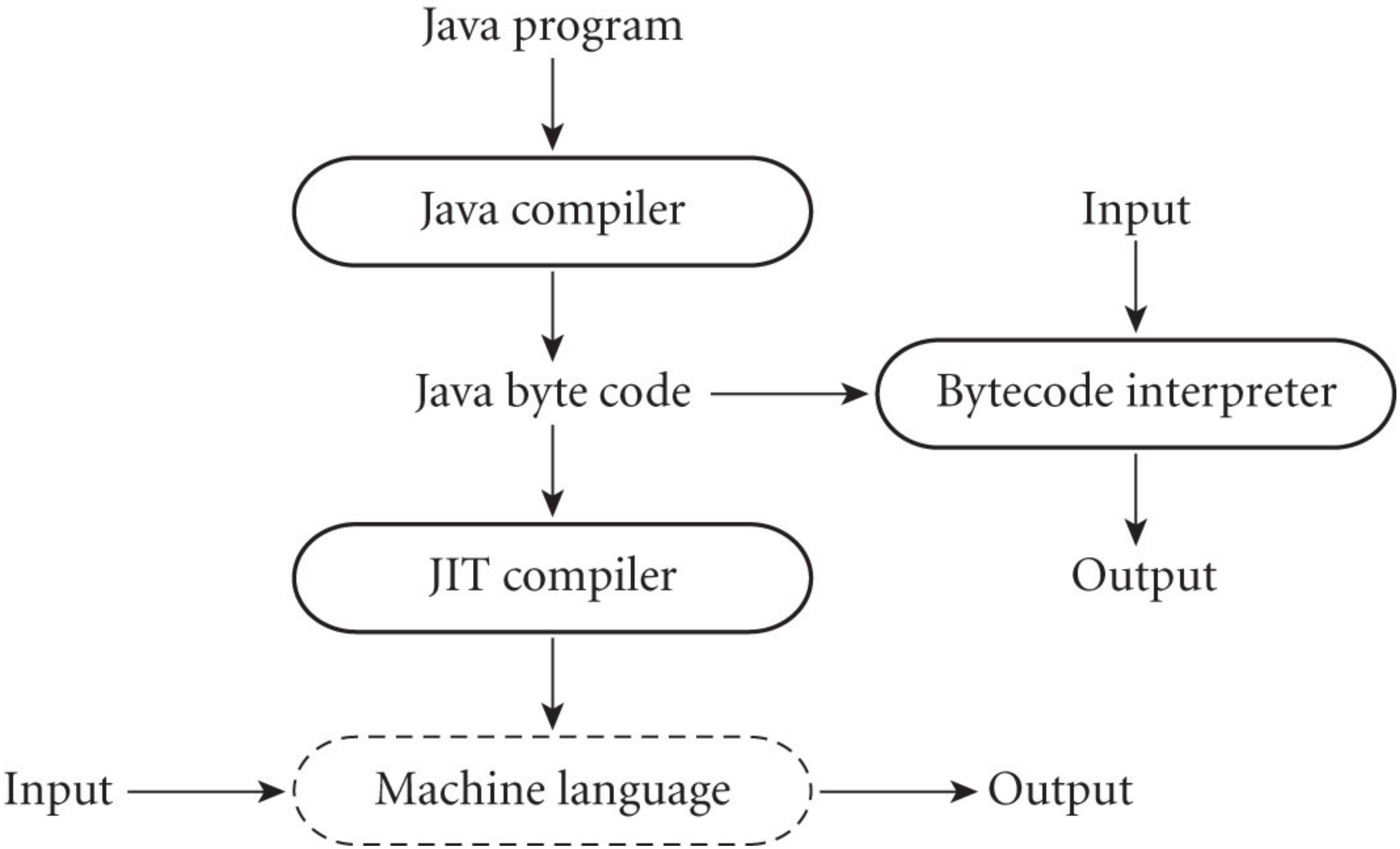
- Compilation
  - Better performance
    - No runtime cost for interpretation
    - Program optimization
- Interpretation
  - Better diagnosis (with excellent source-level debugger)
  - Earlier diagnosis (execute erroneous program)

Hybrid Implementation



- Quick start in “Interpretation” mode
- Compile code on hot paths to speed up
  - E.g., Just-in-Time (JIT) compiler in Java Virtual Machine (JVM)

Hybrid Implementation (Java)



Implementation Strategies in Practice

- Preprocessing
- Library routines and linking
- Post-compilation assembly
- Source-to-source translation
- Bootstrapping

Preprocessing (Basic)

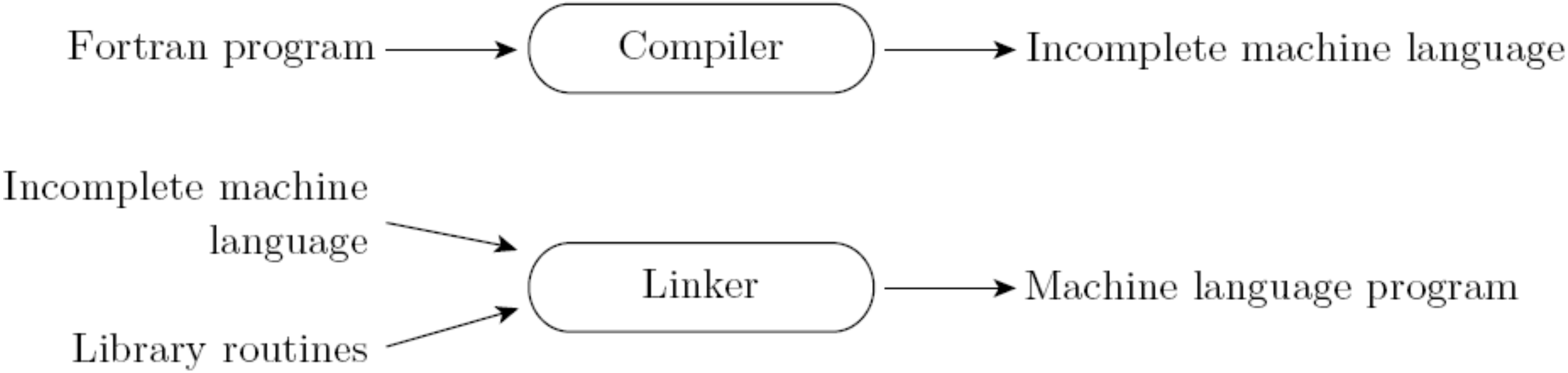
- An initial translator
  - to remove comments and white spaces,
  - to group characters together into tokens such as keywords, identifiers, numbers, and symbols,
  - to expand abbreviations in the style of a macro assembler, and
  - to identify higher-level syntactic structures, such as loops and subroutines
- Goal
  - To provide an intermediate form that mirrors the structure of the source, but can be interpreted more efficiently

Preprocessing (C)

- Conditional compilation
  - Delete portions of code to allow several versions of a program to be built from the same source

Library routines and linking (Fortran)

- The compilation of source code counts on the existence of a library of subroutines invoked by the program



Fortran program → Compiler → Incomplete machine language

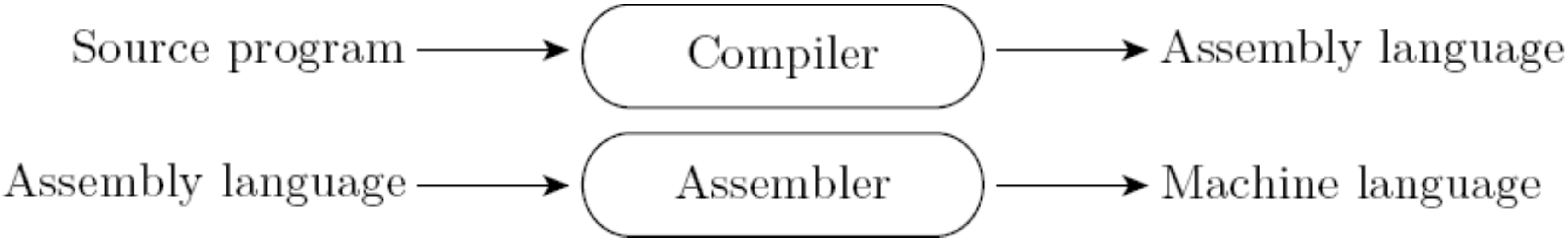
Incomplete machine language

Library routines

→ Linker → Machine language program

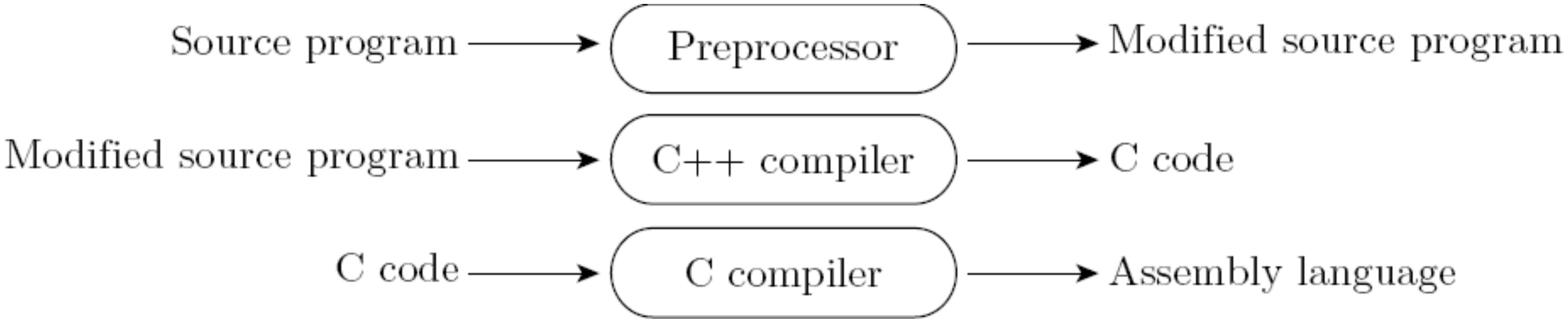
Post-compilation assembly (gcc)

- Source code is first compiled to assembly code, and then the assembler translates it to machine code
  - To facilitate debugging (assembly code is easier to read)
  - To isolate the compiler from changes in the format of machine language files (only the commonly shared assembler must be changed)



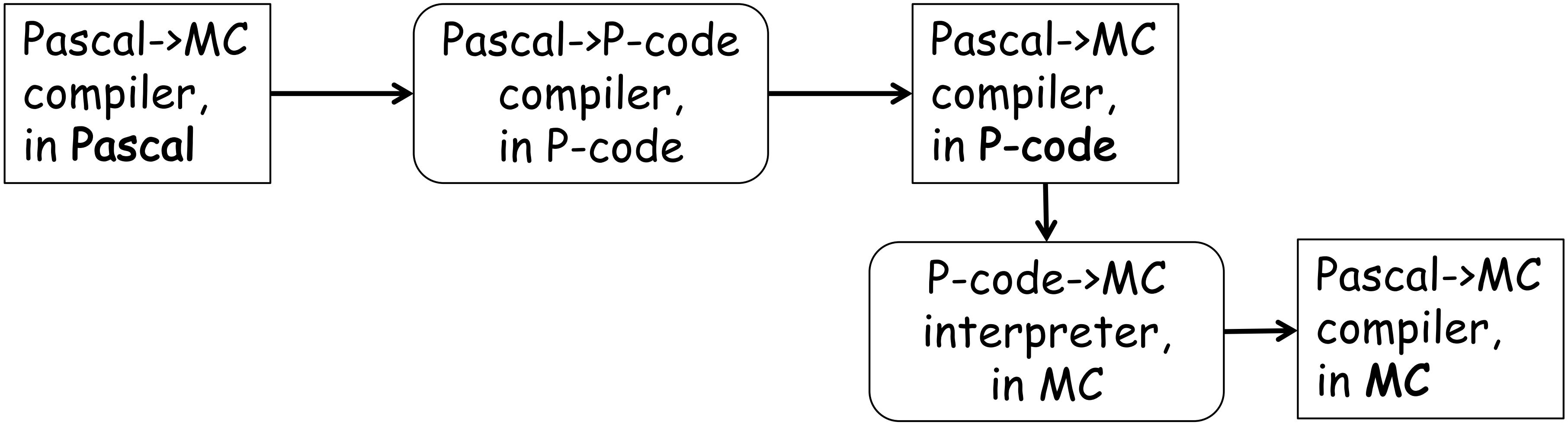
Source-to-Source Translation

- AT&T C++ compiler
  - To translate C++ programs to C programs
  - To facilitate reuse of compilers or language support


Source program
Preprocessor
Modified source program
Modified source program
C++ compiler
C code
C code
C compiler
Assembly language

Bootstrapping

- Many compilers are self-hosting:
  - They are written in the language they compile
  - Bootstrapping is used to compile the compiler in the first place



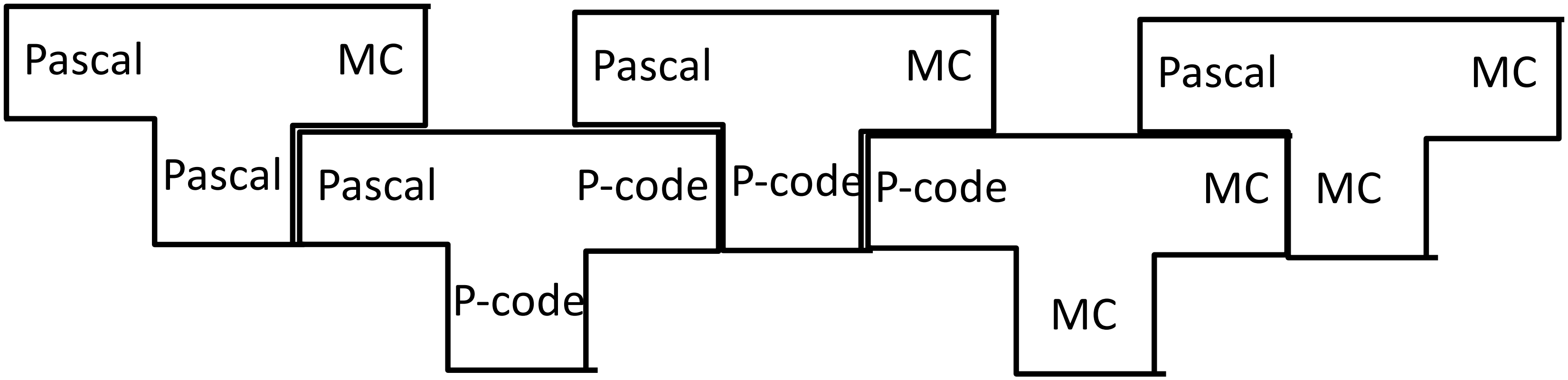
Pascal->MC compiler, in **Pascal**

Pascal->P-code compiler, in P-code

Pascal->MC compiler, in **P-code**

P-code->MC interpreter, in MC

Pascal->MC compiler, in **MC**



Pascal MC
Pascal

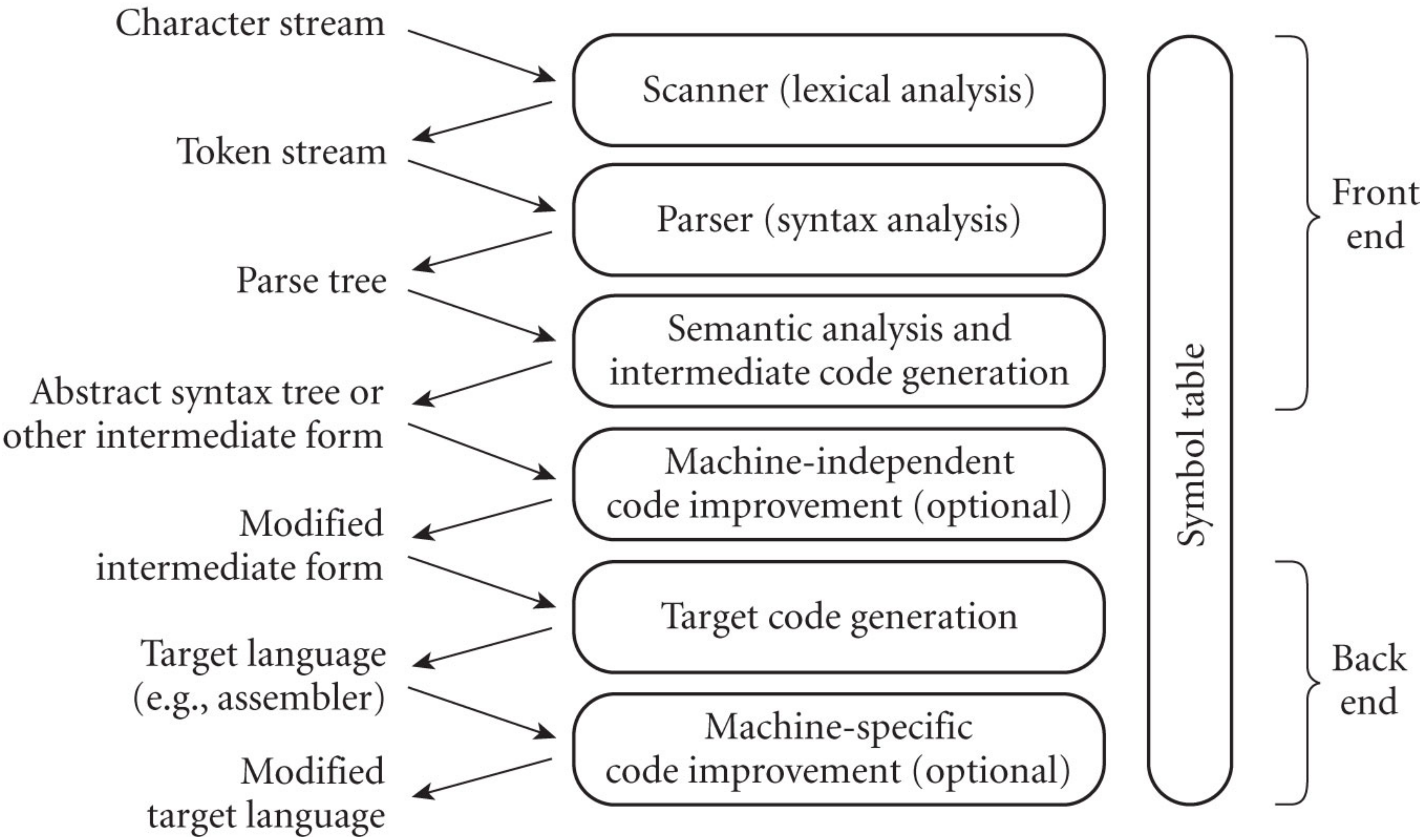
Pascal P-code
P-code

Pascal MC
P-code

P-code MC
MC

Pascal MC
MC

Overview of Compilation


Character stream

Scanner (lexical analysis)

Token stream

Parser (syntax analysis)

Parse tree

Semantic analysis and intermediate code generation

Abstract syntax tree or other intermediate form

Machine-independent code improvement (optional)

Modified intermediate form

Target code generation

Target language (e.g., assembler)

Machine-specific code improvement (optional)

Modified target language

Symbol table

Front end

Back end

Front end & back end

- Front end
  - To analyze the source code in order to build an internal representation (IR) of the program
  - It includes: lexical analysis, syntactic analysis, and semantic analysis
- Back end
  - To gather and analyze program information from IR, to optimize the code, and to generate machine code
  - It includes: optimization and code generation

Scanning (Lexical Analysis)

- Break the program into “tokens”—the smallest meaningful units
  - This can save time, since character-by-character processing is slow
- We can tune the scanner better
  - E.g., remove spaces & comments
- A scanner uses a Deterministic Finite Automaton (DFA) to recognize tokens

A running example: Greatest Common Divisor (GCD)

```
int main() {
    int i = getint(),
        j = getint();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```

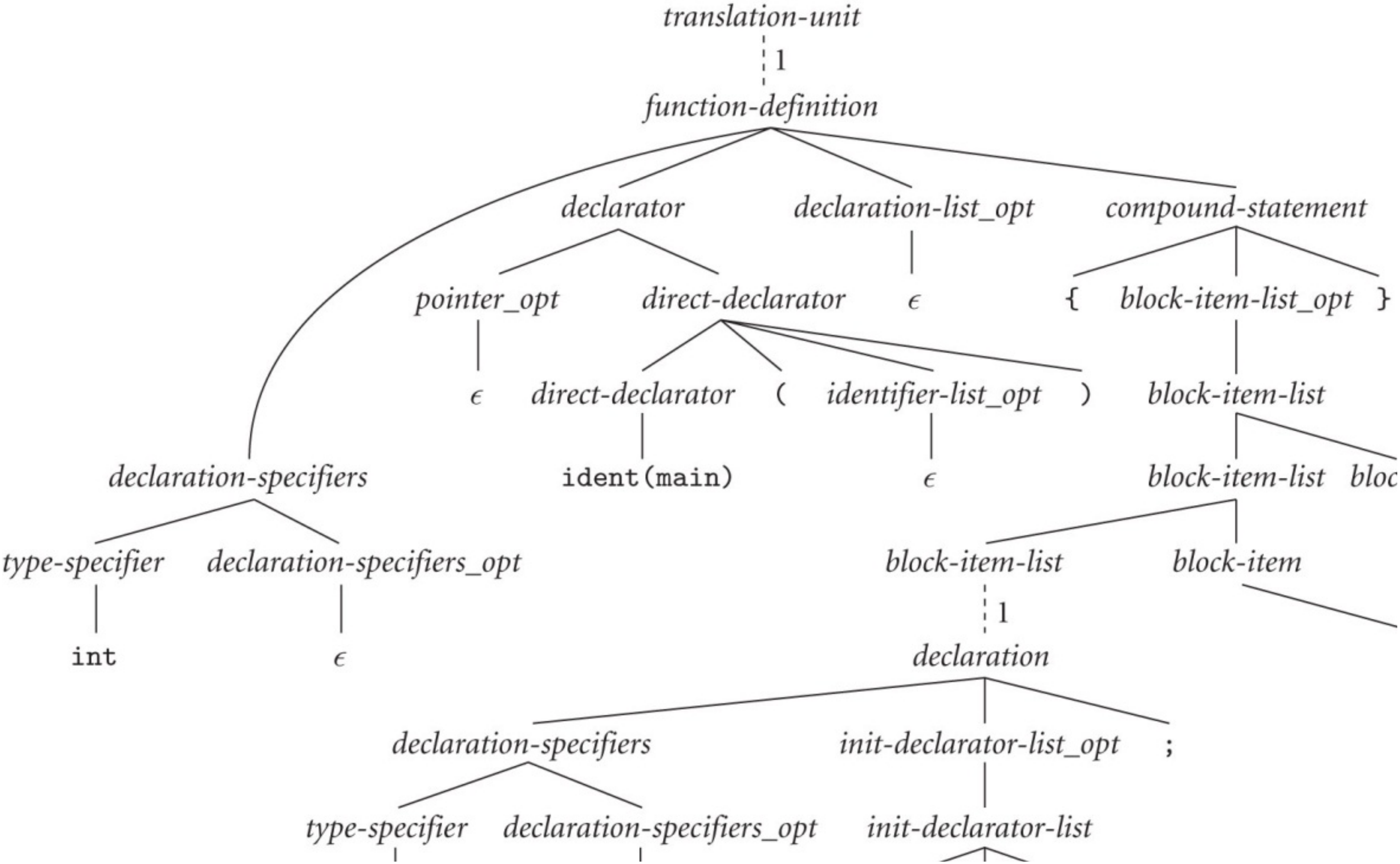
Token sequence:

```
int     main    (     )     {
int     i       =     getint
(       )       ,     j     =
getint  (       )     ;     while
(       i       !=    j     )
{       if      (     i     >
j       )       i     =     i
-       j       ;     else  j
=       j       -     i     ;
}       putint  (     i     )
;       }
```

Parsing

- Organize tokens into a parse tree that represents higher-level constructs (statements, expressions, subroutines)
  - Each construct is a node in the tree
  - Each construct's constituents are its children

GCD Parsing Tree



Semantic Analysis

- Determine the meaning of a program
- A semantic analyzer builds and maintains a symbol table data structure that maps each identifier to the information known about it, such as the identifier's type, internal structure, and scope

Semantic Analysis

- With the symbol table, the semantic analyzer can enforce a large variety of rules to check for errors
- Sample rules:
  - Each identifier is declared before it is used
  - Any function with a non-void return type returns a value explicitly
  - Subroutine calls provide the correct number and types of arguments

Semantic Analysis

- Static semantics
  - Rules that can be checked at compile time
- Dynamic semantics
  - Rules that must be checked at run time, such as
    - Variables should never be used in an expression unless they have been given a value
    - Pointers should never be dereferenced unless they refer to a valid object

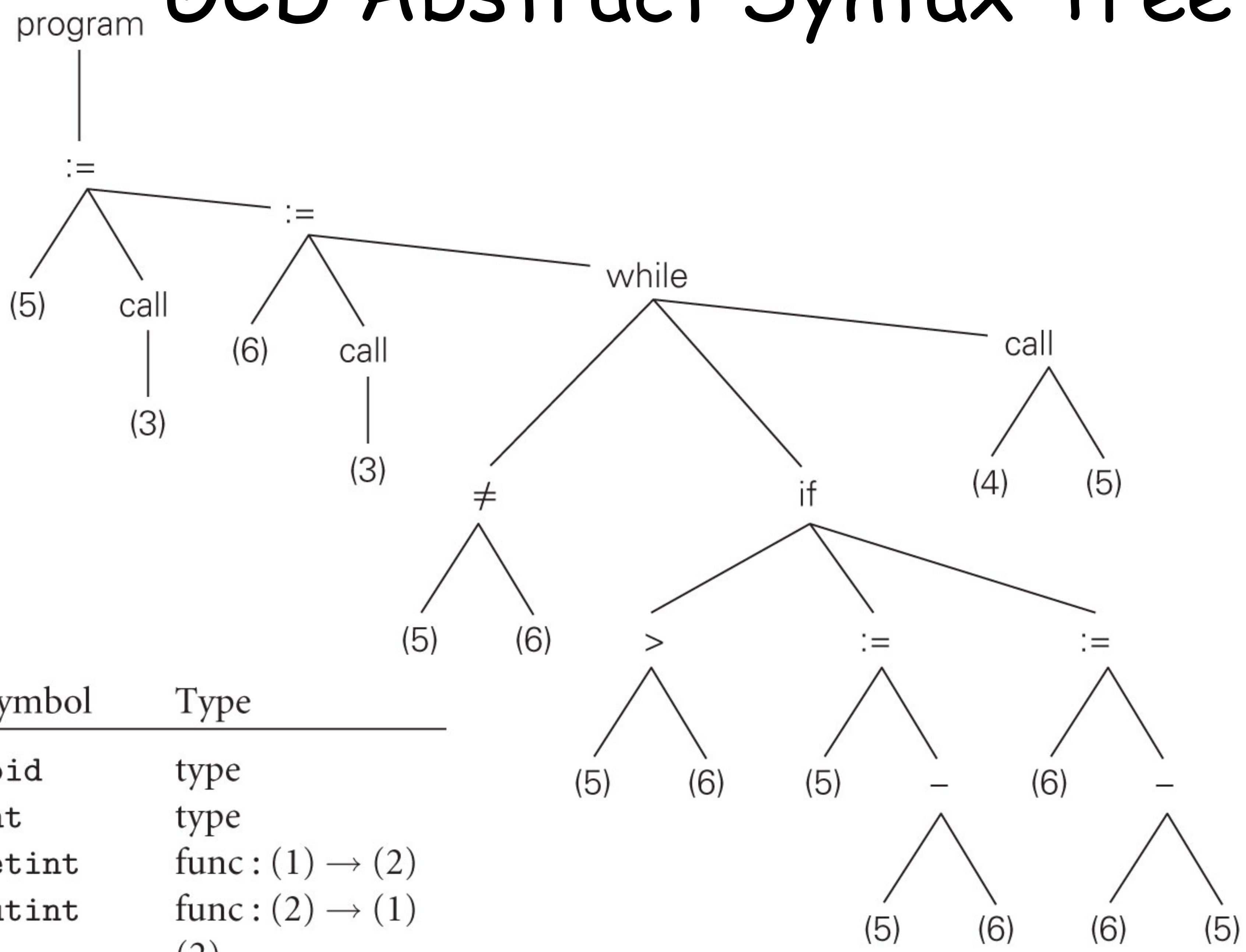
Syntax Tree

- A parse tree is known as a **concrete syntax tree**
  - It demonstrates concretely, how a particular sequence of tokens can be derived under the rule of the context-free grammar
- However, much of the information in a concrete syntax tree is irrelevant
  - E.g., ε under some branches

Syntax Tree

- In the process of checking static semantic rules, a semantic analyzer transforms the parse tree into an **abstract syntax tree (AST, or syntax tree)** by
  - removing “unimportant” nodes, and
  - annotating remaining nodes with information like pointers from identifiers to their symbol table entries

GCD Abstract Syntax Tree



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Intermediate Form (IF)

- Generated after semantic analysis
  - In many compilers, an **AST** is passed as IF from the front end to the back end
  - In other compilers, a **control flow graph** is passed as IF

Optimization [1]

- High-level optimization
  - Goal: perform high-level analysis and optimization of programs
  - Input: AST + symbol table
  - Output: low-level program representation, such as 3-address code(TAC)
  - Tasks:
    - Procedure/method inlining
    - Array/pointer dependence analysis
    - Loop transformations: unrolling, permutation, …

Optimization [1]

- Low-level optimization
  - Goal: perform low-level analysis and optimizations
  - Input: low-level representation of programs, such as 3-address code
  - Output: optimized low-level representation, and additional information, such as def-use chains
  - Tasks:
    - Dataflow analysis: live variables, reaching definitions, …
    - Scalar optimizations: constant propagation, partial redundancy elimination, …

Code Generator [1]

- Goal: produce assembly/machine code from optimized low-level representation of programs
- Tasks:
  - Register allocation
  - Instruction selection

Reference

[1] Keshav Pingali, Advanced Topics in Compilers, https://www.cs.utexas.edu/~pingali/CS380C/2013/lectures/intro.pdf