CMP4060 Languages and Compilers

Ayman AboElHassan, PhD
Assistant Professor
ayman.abo.elmaaty@eng.cu.edu.eg

Course Outline

Introduces the fundamentals of compilers and programming languages

- Introduction to Compilers
- Lexical Analysis: Regular Grammars
- Lexical Implementation: Finite Automata
- Syntax Analysis: Context-Free Grammars
- Parser Implementation: Top-Down Parsers
- Parser Implementation: Bottom-Up Parsers
- Semantic Analysis
- Code Generation
- Code Optimization

Course Structure

The course has theoretical & practical aspects

- Need both in programming languages

Focusing on theory

- Lectures
- Assignments

Focusing on practice

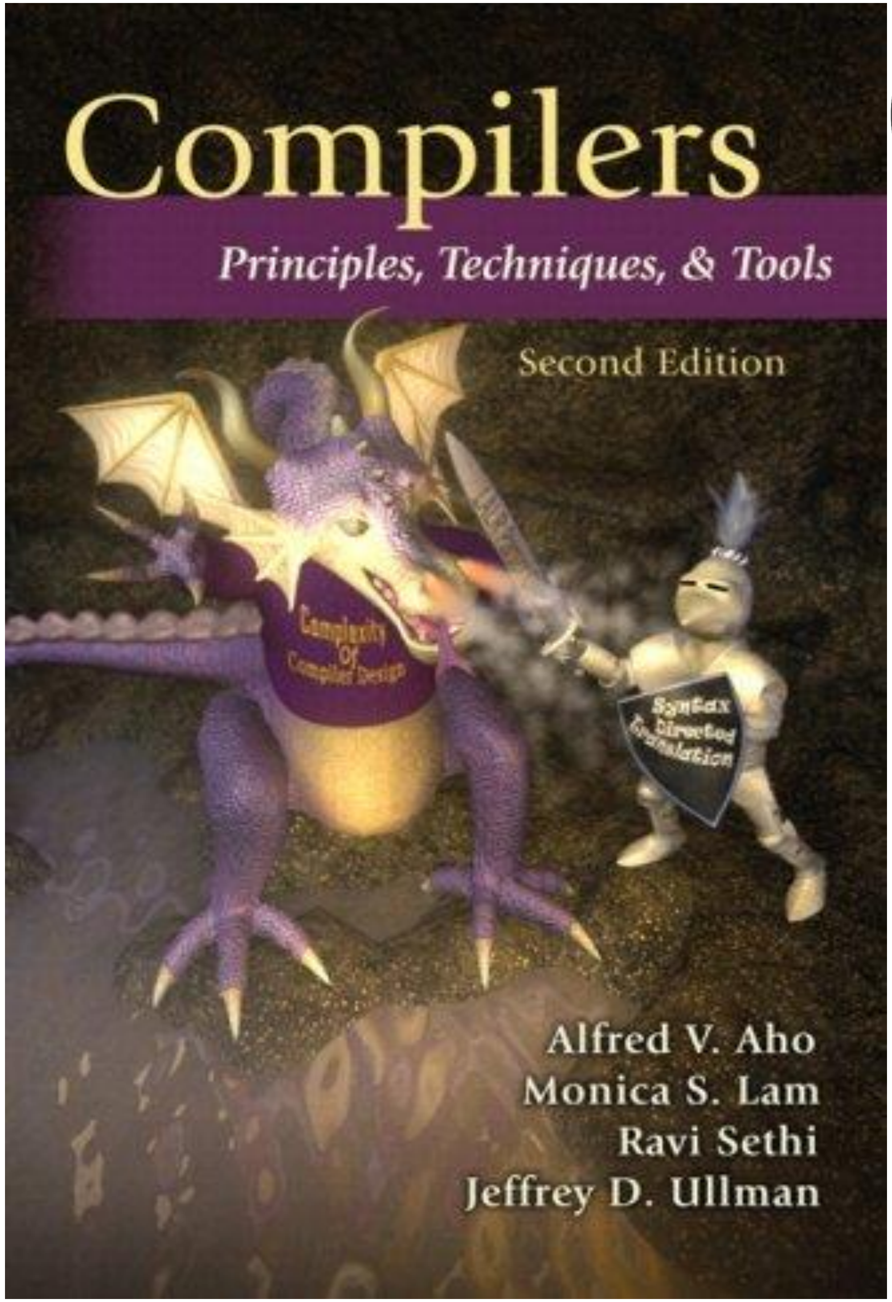
- Labs
- Project

References

Lecture Slides

Textbook

- The Purple Dragon Book "**Compilers: Principles, Techniques & Tools**"
  - Alfred V. Aho,
  - Monica S. Lam
  - Ravi Sethi
  - Jeffrey D. Ullman



Draft Grading (Subject to Change)

Final	60%
Midterm	15%
Project	15%
Labs	5%
Quizzes & Assignments	5%

Draft Schedule (Subject to Change)

No.	Date	Lecture
1	14 Feb	Introduction
3	28 Feb	Lexical Analysis
5	13 Mar	Top-Down Parsing
7	27 Mar	Bottom-Up Parsing
9	**10 Apr**	**Off due to Eid ElFitr**
11	24 Apr	Semantic Analysis
13	8 May	Code Generation
15	**22 May**	**Off**

No.	Date	Lecture
2	21 Feb	Lexical Analysis
4	6 Mar	Syntax Analysis CFG
6	20 Mar	Top-Down Parsing
8	**3 Apr**	**Off due to Midterms**
10	17 Apr	Semantic Analysis
12	1 May	Code Generation
14	15 May	Code Optimization

Office hours:

- Mon ➔ 1pm to 2pm
- Wed ➔ 10am to 12pm

Google Classroom (Gmail)

Class Code

3mim4aa

Historic Background

A Long Time Ago

1940

- First electronic computers were programmed in machine language
  - Example code: 0110 0001 0000 0110
- Very low-level operations
  - Move data from one location to another
  - Add values of 2 registers
  - Compare values of 2 registers
- Cons
  - Programming was slow/painful/error-prone
  - Hard to understand & modify

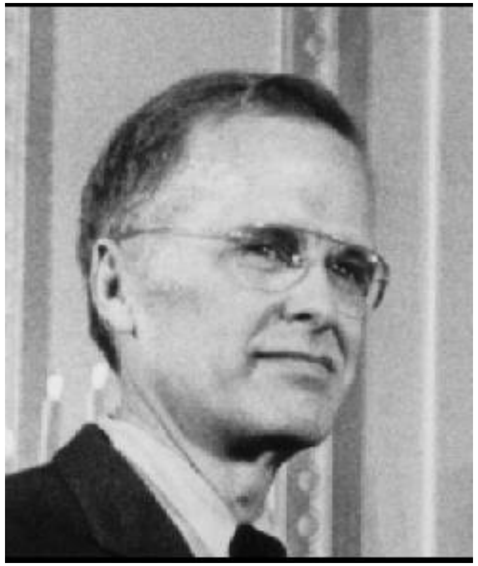
A Few Years Later

1954

- IBM developed Mainframe
  - Programming is done in assembly
- Assembly
  - Higher level than machine code
  - mnemonic representations of machine instructions
    - Example code: Add Reg1 6

      0110 0001 0000 0110
  - Later macro instructions used
- Cons
  - Software cost exceeds hardware → Sol: Speed coding “Interpreter”

A Few Moments Later



John Backus

- Developed the first compiler for FORTRAN 1
- Translate high-level code to assembly
- 1954-1957 → FORTRAN 1 Project
- 1958 → 50% of all software were in FORTAN
  - **Development time halved**

Modern compilers preserve the outlines of FORTRAN 1

Other High-Level Languages

FORTRAN → Scientific programming

COBOL → Business data processing

LISP → Symbolic Computation

Concept

- Create high-level notations
- Easy to write/understand/modify

Programming Languages

Generations

- 1st ➔ Machine Language
- 2nd ➔ Assembly
- 3rd ➔ High-Level Languages
- 4th ➔ Application-specific languages (i.e.: SQL, Postscript)
- 5th ➔ Graphical development interface (i.e.: Node-RED, Scratch)
- LLM-based Copilots?

Why do we have so many languages?

Why do we have so many languages?

Application domains have distinctive and conflicting needs

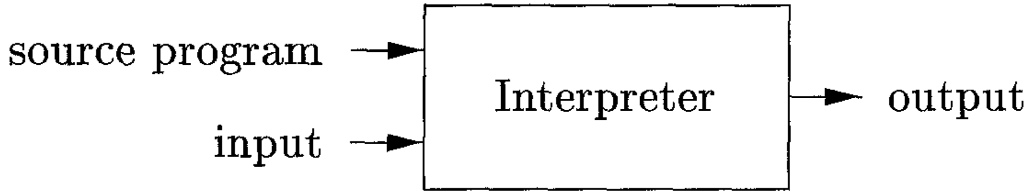
Examples

- Scientific Computing → High Performance
- Business → Report Generation
- Artificial Intelligence → Symbolic Computation
- Embedded Systems → Low-level Access
- Hardware Design → Physical Attributes
- Special Purpose Languages

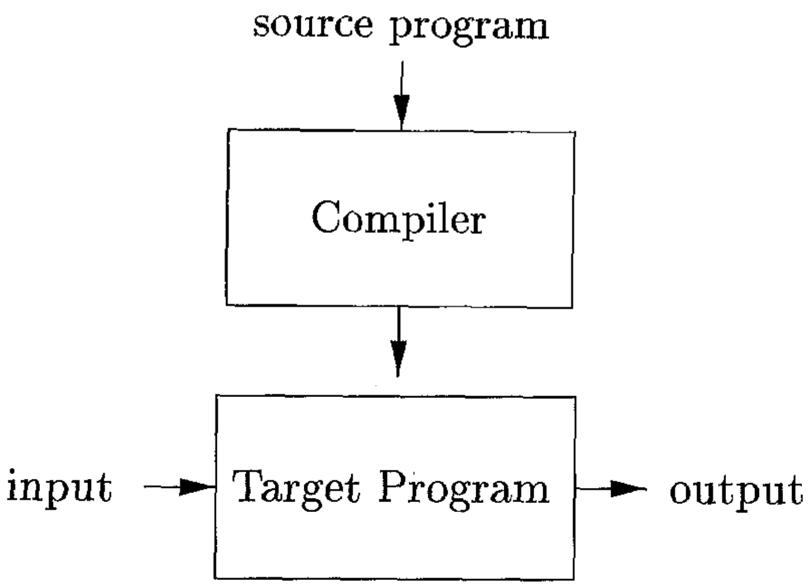
Interpreters VS Compilers

Language Processors

Interpreters



Compilers



Language Processors

Interpreters

Run programs “as is”

Example:

- Assembler (Old concept)
- Python/R (Renewed concept)

Compilers

Translates a program from the *source language* to an <u>equivalent</u> program in the *target language*

Example

- C/C++

Language Processors

Interpreters

Good error diagnosis (Usually)

- Executes the source program statement by statement

Compilers

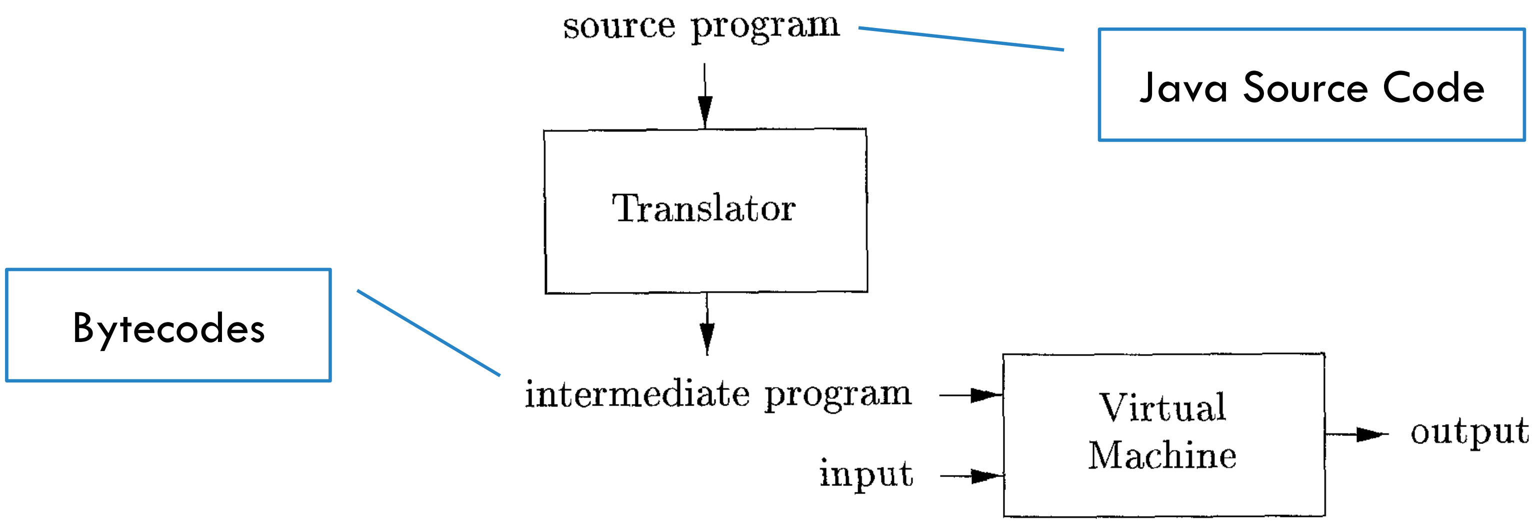
Much faster than interpreters

- *Machine-language* target program maps inputs to outputs directly

Language Processors

Hybrid Compilers

- Java language processors combine compilation and interpretation



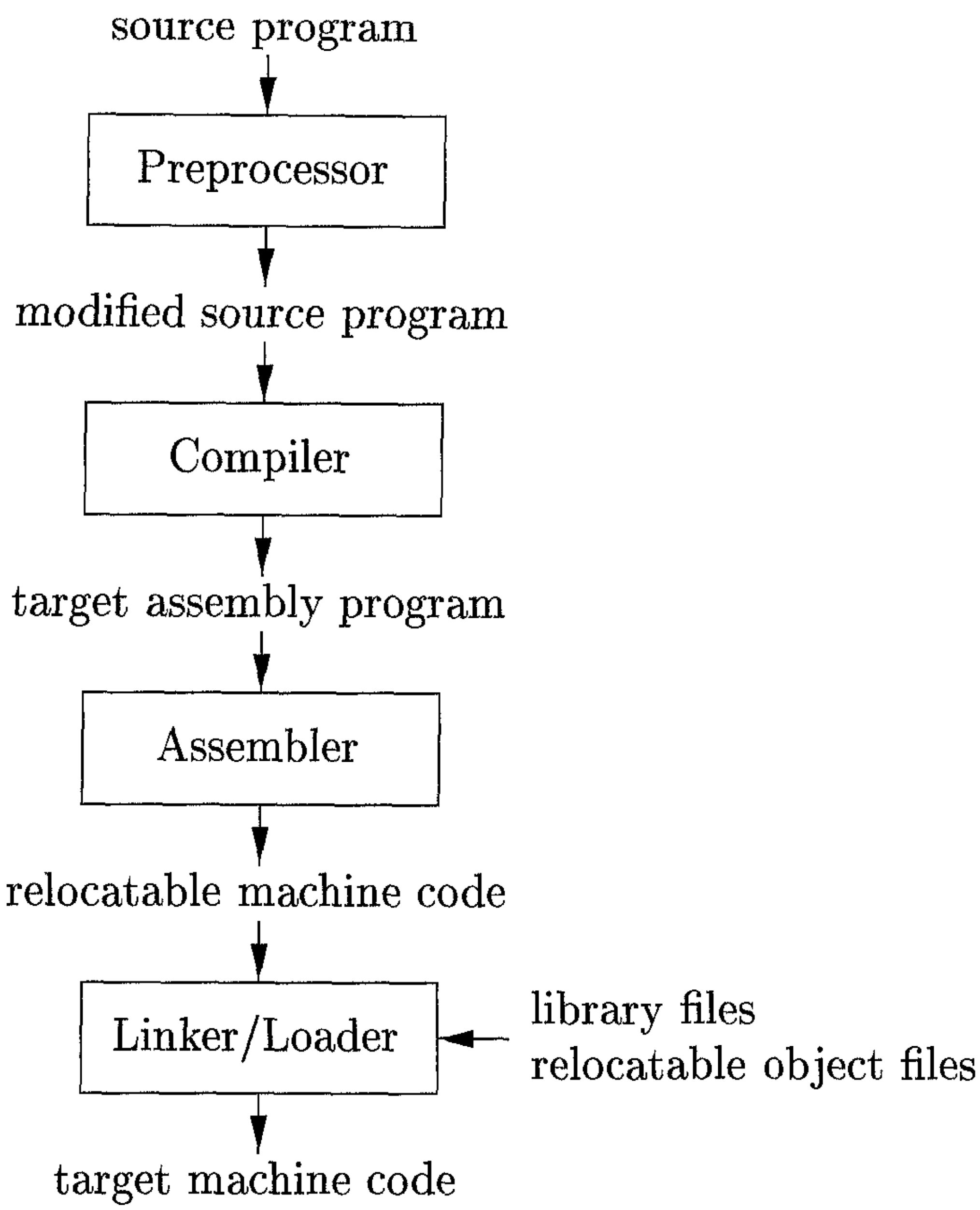
- Bytecodes compiled on one machine can be interpreted on another machine (could be done across a network)

Language Processors

Multiple supporting modules are used to reach the executable version

1. Preprocessor

- Collect source program modules stored in separate files
  - Build the actual program from all its subfiles
- Expand shorthand “macros” into the source program

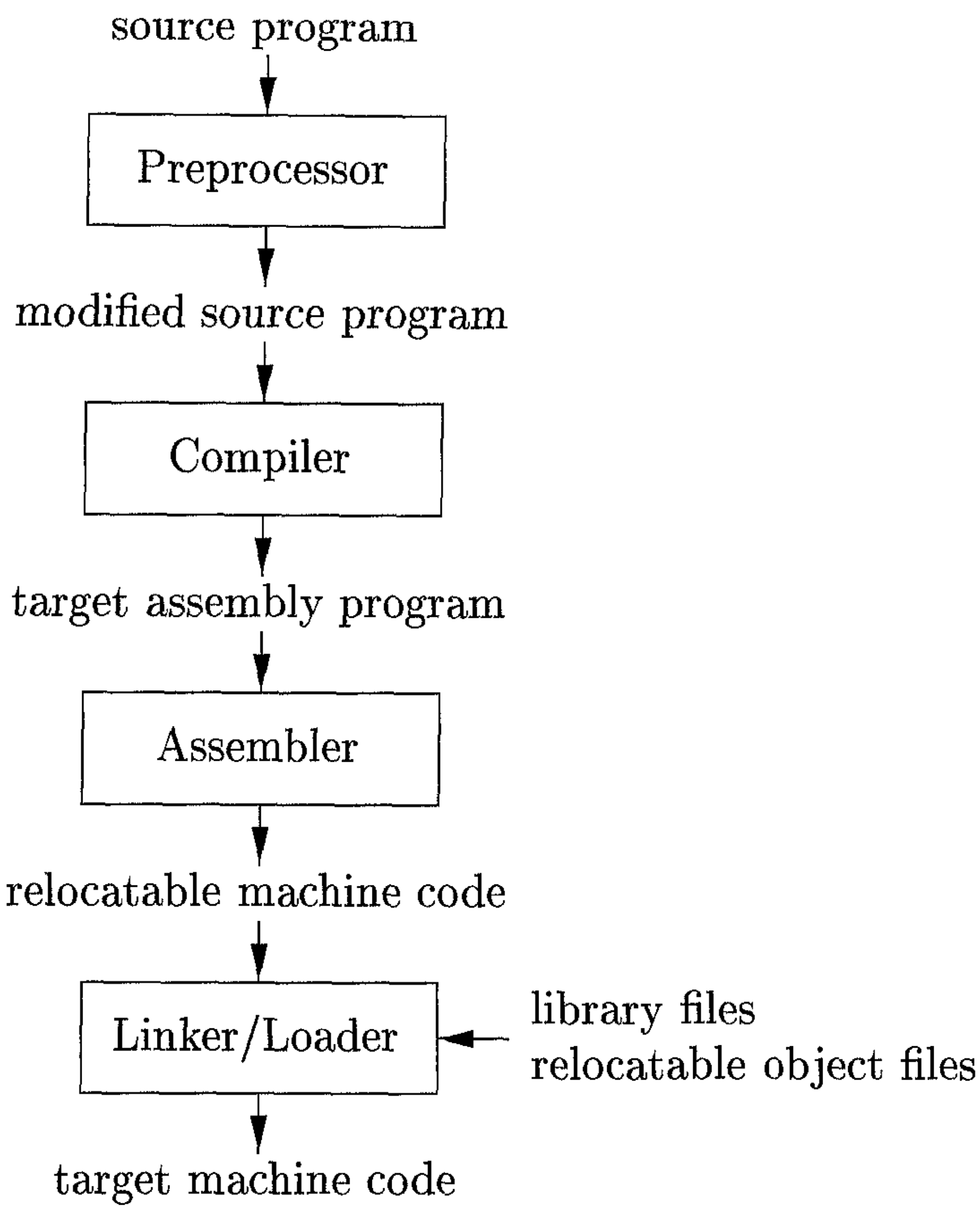


Language Processors

Multiple supporting modules are used to reach the executable version

2. Compiler
   - Translates the source program into assembly language

3. Assembler
   - Produces re-locatable machine code

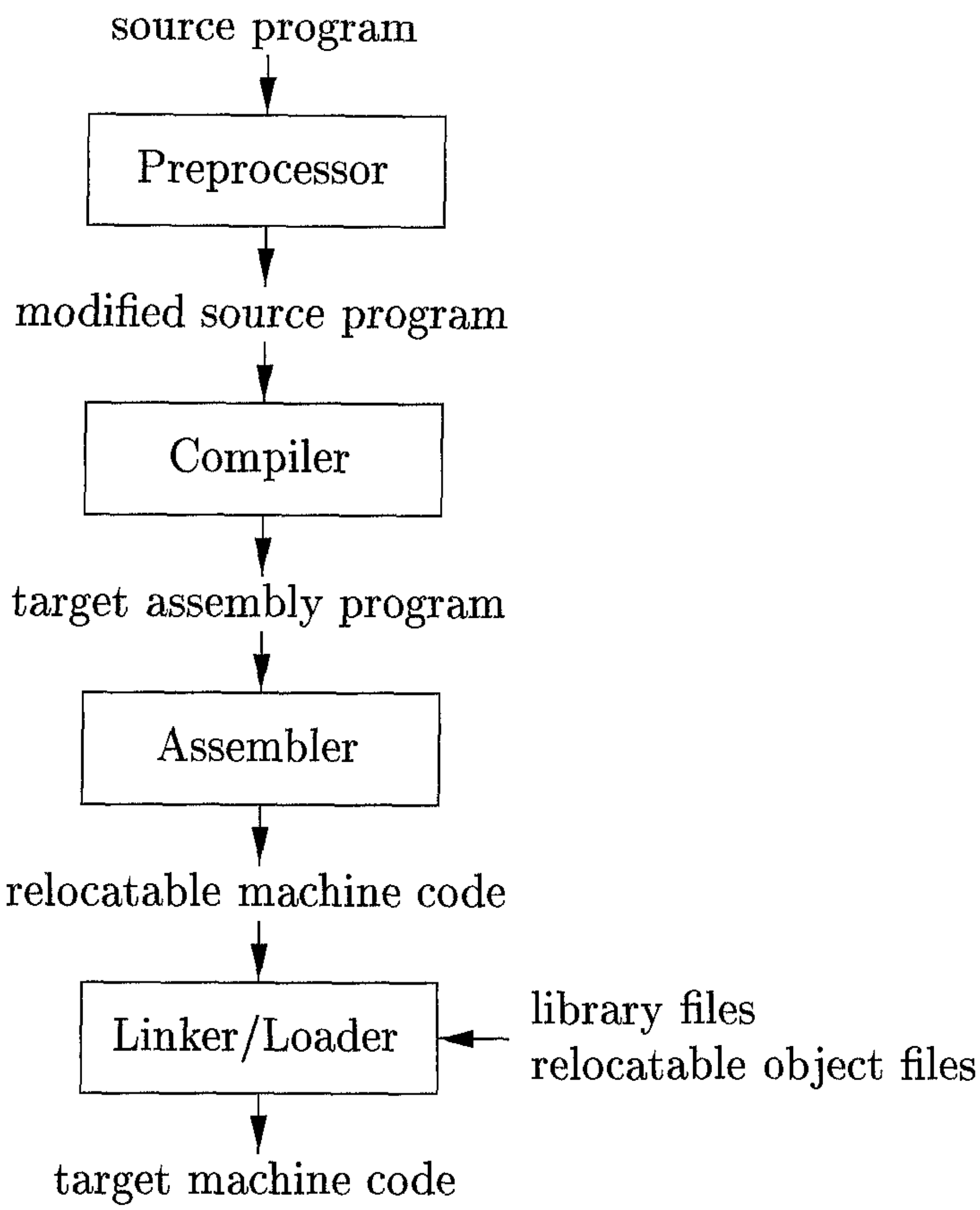


Language Processors

Multiple supporting modules are used to reach the executable version

4. Linker
- Resolves external memory addresses
  - Code in one file may refer to a location in another file (.dll files)

5. Loader
- Combine all the executable object files into memory for execution



Compilers Structure

Structure of a Compiler

Start/End of a **Word**?

Start/End of a **Sentence**?

Full-Sentence vs Clause?

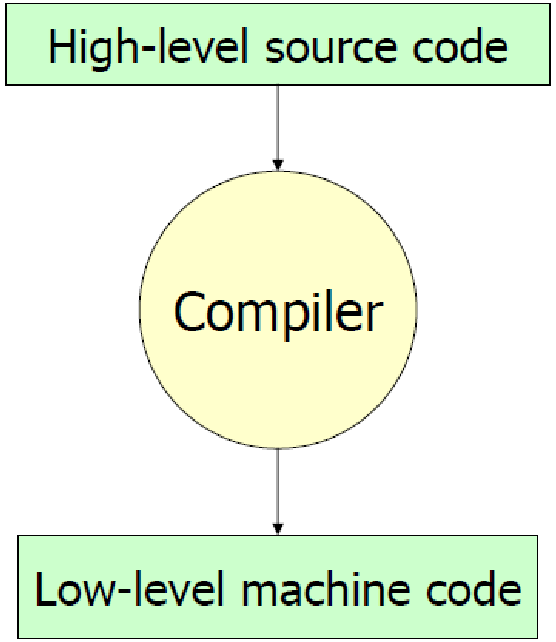
Name/Verb?

Sequence?

Grammer?

Meaning?

Structure of a Compiler



Analysis (**Front End**)

- Breaks up the source into pieces
- Imposes a grammatical structure on them
- Creates an **intermediate representation**
- Detects if the source is syntactically ill-formed or semantically unsound,
- Provides informative messages
- Collects info about the source and stores it in a **symbol table**

Synthesis (**Back End**)

- Constructs the desired **target program** from the intermediate representation and the symbol table.

Structure of a Compiler

1. Lexical Analysis → Scanning
2. Syntax Analysis → Parsing
3. Semantic Analysis
4. Code Generation
5. Optimization

The first 3 steps can be understood by analogy to how humans comprehend language

1. Lexical Analysis

<u>Goal:</u> Recognize words (divide program text into tokens)

- Smallest unit above letters

How:

- Determine the start of a sentence (start of sentence symbol)
- Separate whole text into words (word separator)
- Determine the end of a sentence (end of sentence symbol)

Example: This is a sentence.

1. Lexical Analysis

<u>Goal:</u> Recognize words (divide program text into tokens)

- Smallest unit above letters

How:

- Determine the start of a sentence (start of sentence symbol)
- Separate whole text into words (word separator)
- Determine the end of a sentence (end of sentence symbol)

Example: This is a sentence.

Start Separator End

1. Lexical Analysis

<u>Goal:</u> Recognize words (divide program text into tokens)

- Smallest unit above letters

How:

- Determine the start of a sentence (start of sentence symbol)
- Separate whole text into words (word separator)
- Determine the end of a sentence (end of sentence symbol)

Example:

if x == y then z = 1; else z = 2;

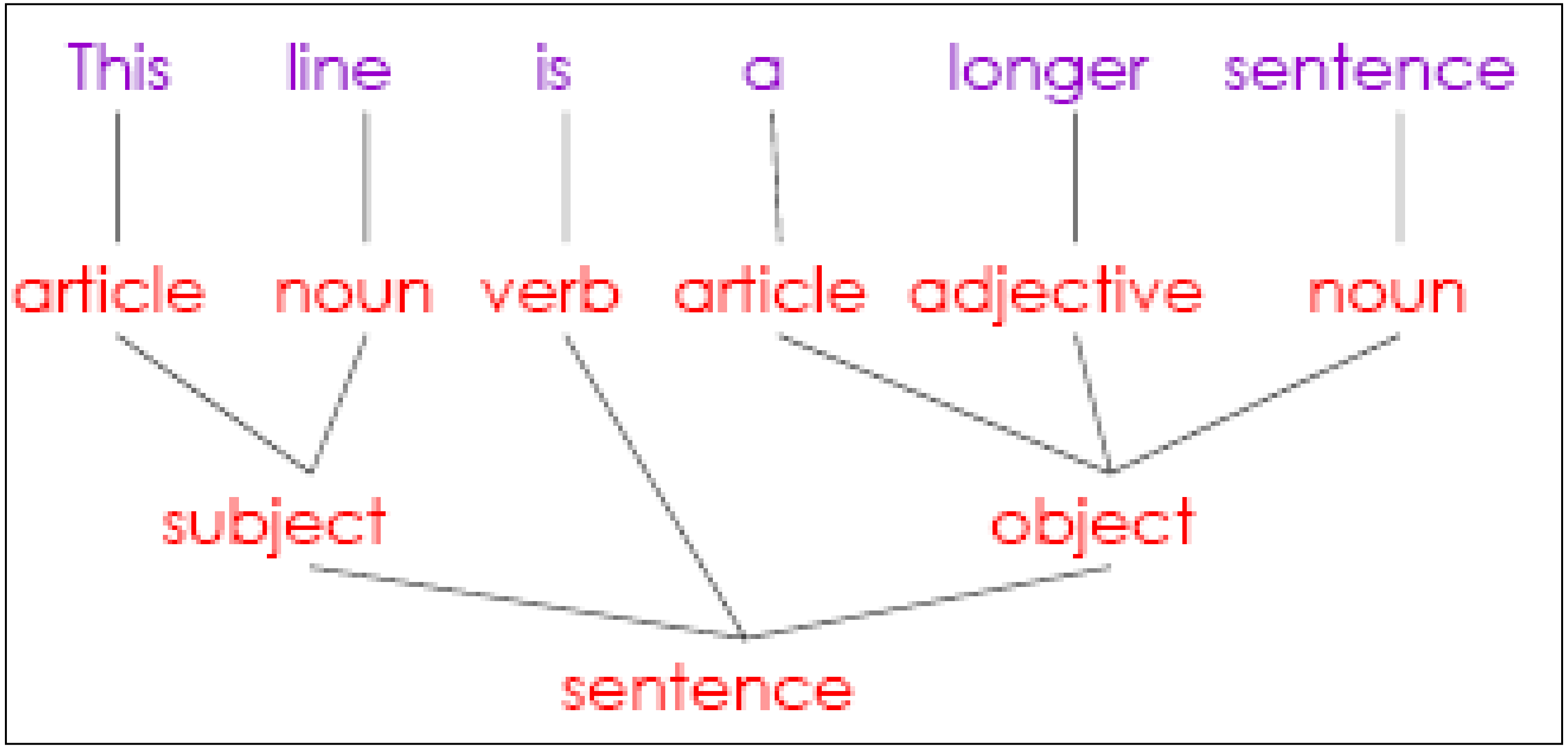
1. Lexical Analysis

Performs

- Collates multiple line statements
- Ignore un-important input characters
  - Comments, Blanks
- Recognizes basic entities of the language
  - Constants, identifiers, operators
- Stores recognized items in the **symbol table**

2. Syntax Analysis

<u>Goal:</u> Understand sentence structure



2. Syntax Analysis

<u>Goal:</u> Understand sentence structure

- Parsing program expressions is the same as diagraming a sentence

Example:

- Code → if x == y then z = 1; else z = 2;
- Diagramed →

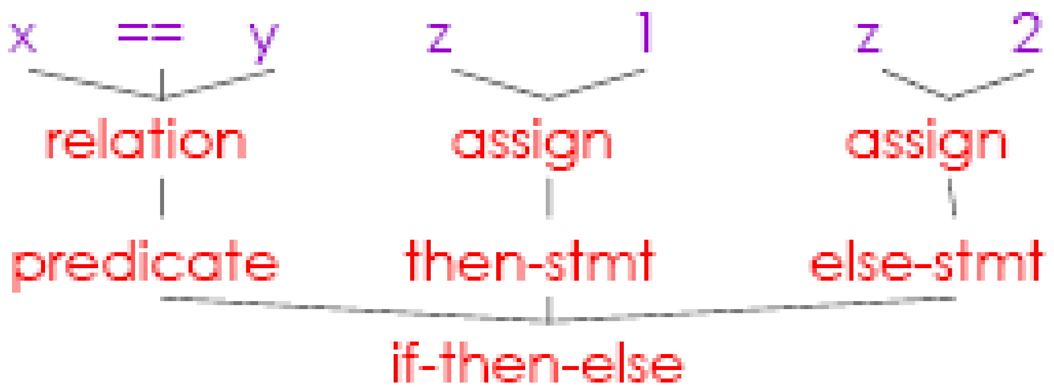
2. Syntax Analysis

Goal: Understand sentence structure

- Parsing program expressions is the same as diagraming a sentence

Example:

- Code ➔ if x == y then z = 1; else z = 2;
- Diagramed ➔



3. Semantic Analysis

<u>Goal:</u> Understand “meaning” and catch inconsistencies

- Meaning is too hard for compilers (limited analysis is done)

Example in English:

- Jack said Jerry left his assignment at home
  - Who is referred to by “his”? Jack or Jerry?
- Jack said Jack left his assignment at home
  - How many Jacks are there?
  - Which Jack left his assignment?

3. Semantic Analysis

<u>Goal:</u> Understand “meaning” and catch inconsistencies

- Meaning is too hard for compilers (limited analysis is done)

Example in Programming:

- What is the output?

```
{
        int Jack = 3;
        {
                int Jack = 4;
                cout << Jack;
        }
        cout << Jack;
}
```

3. Semantic Analysis

Goal: Understand “meaning” and catch inconsistencies

- Meaning is too hard for compilers (limited analysis is done)
- Define strict rules to avoid ambiguities

Compilers perform many semantic checks

- Variables declared before use
- Type checks
- Scope checks

4. Code Generation

Compiler could construct one or more intermediate representations (usually assembly), which can have a variety of forms

- Syntax Tree
  - A form of intermediate representation
  - Used during syntax & semantic analysis
- Intermediate Language
  - Explicit low-level intermediate representation (for an abstract machine)
  - Should be easy to produce
  - Should be easy to translate into the target machine

Some compilers may translate into another language

5. Optimization

Automatically modify programs to

- Run faster
- Use less memory
- Generally, conserve used resources

Intermediate code optimization

- Get rid of unused variables
- Eliminate multiplication by 1 & addition by 0
- Loop optimization ➔ removes statements not modified in the loop
- Common sub-expression elimination

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

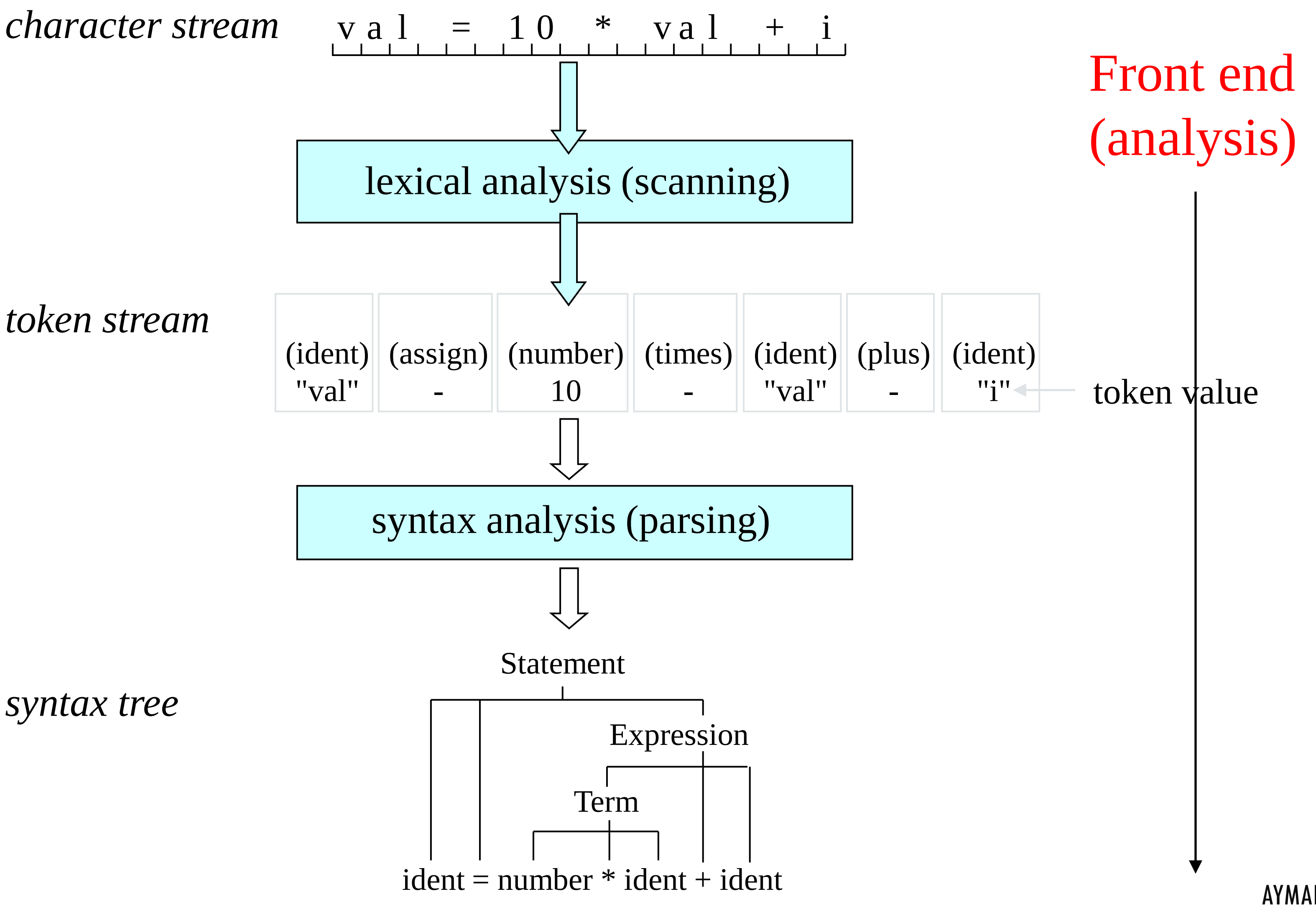
```
t1 = id3 * 60.0
id1 = id2 + t1
```

5. Optimization

Object code optimizations

- It is possible to have another code optimization phase that transforms the object code into a more efficient object code
- These optimizations use features of the hardware itself to make efficient use of processors and registers
  - Specialized instructions
  - Pipelining

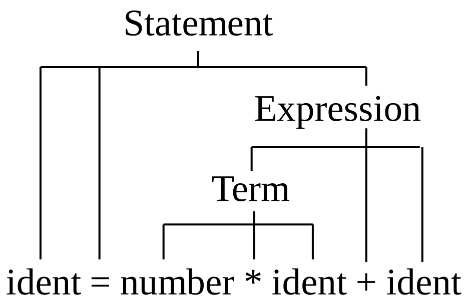
Recap


character stream
v a l = 1 0 * v a l + i
lexical analysis (scanning)
token stream
(ident) "val"
(assign) -
(number) 10
(times) -
(ident) "val"
(plus) -
(ident) "i"
token value
syntax analysis (parsing)
syntax tree
Statement
Expression
Term
ident = number * ident + ident
Front end (analysis)

Recap

Front end

syntax tree



semantic analysis (type checking, ...)

intermediate representation

syntax tree, symbol table, or three address code (TAC) ...

Back end (synthesis)

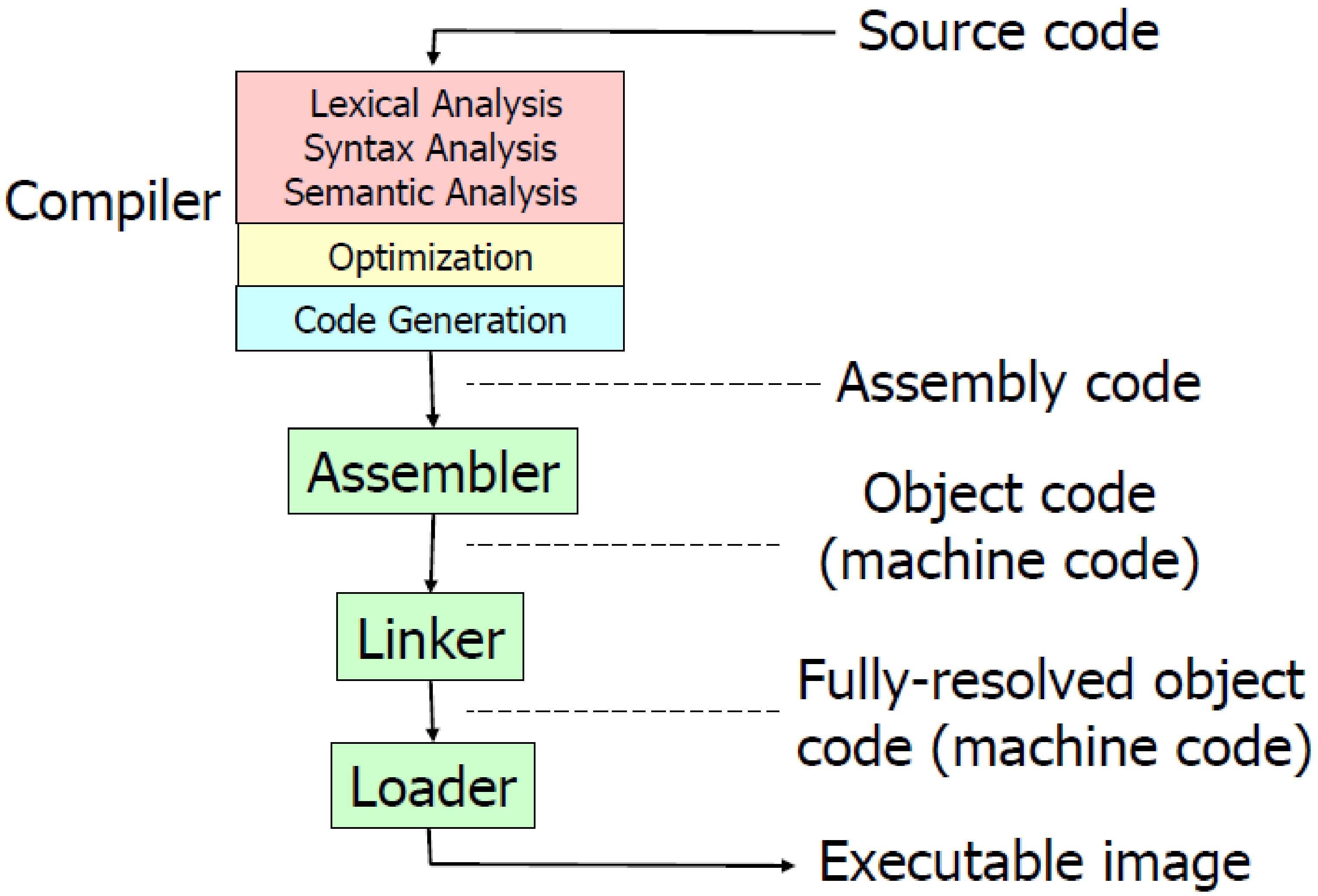
optimization

code generation

machine code

const 10
ld
mul
...

Big Picture



Source code
Compiler
Lexical Analysis
Syntax Analysis
Semantic Analysis
Optimization
Code Generation
Assembly code
Assembler
Object code
(machine code)
Linker
Fully-resolved object
code (machine code)
Loader
Executable image

Multi-Phase Operations

Symbol Table

A part of the compiler that interacts with several phases

- Identifiers are found in **lexical** analysis and placed in the symbol table
- Type and scope information is added during **syntactical** and **semantical** analysis
- Type information is used to determine what instructions to use during **code generation**

Error Handling

Occurs across multiple phases

Lexical Analyzer

- Reports invalid character sequences
- Example:

3app = 18..23 + val#ue

Error Handling

Occurs across multiple phases

Lexical Analyzer

- Reports invalid character sequences
- Example:

3app = 18..23 + val#ue

Variable names can not start with a number

Not a number

Variable names can not have '#' character

Error Handling

Occurs across multiple phases

Syntax Analyzer

- Reports invalid token sequences
- Example:

```
int* foo (i, m, k))
   int i;
   int m;
{
   for (i=0; i m) {
   fi (i > m)
      return m;
}
```

Error Handling

Occurs across multiple phases

Syntax Analyzer

- Reports invalid token sequences
- Example:

```
int* foo (i, m, k))
   int i;
   int m;
{
   for (i=0; i m) {
   fi (i > m)
      return m;
}
```

Extra Parentheses ‘)’

Not an expression

Not a keyword or an identifier

Missing Braces ‘}’

Error Handling

Occurs across multiple phases

Symantec Analyzer

- Reports types and scope errors
- Example:

```
int* foo (i, m, k)
   int i;
   int m;
{
   int x;
   x = x + m + N;
   return m;
}
```

Error Handling

Occurs across multiple phases

Symantec Analyzer

- Reports types and scope errors
- Example:

```
int* foo (i, m, k)
   int i;
   int m;
{
   int x;
   x = x + m + N;
   return m;
}
```

Type not declared

Uninitialized variable used

Undeclared variable

Mismatch return type

Compiler may be able to continue with some errors, but other errors may stop the process

Grouping Phases into a Pass

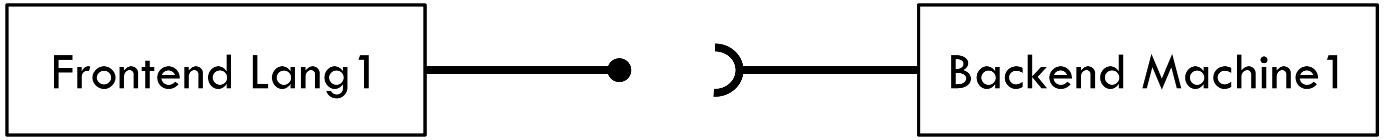
Example:

- Frontend phases (lexical analysis, syntax analysis, semantic analysis, and intermediate code generation) can be grouped into one pass.
- Code optimization might be an optional pass.
- A backend pass may consist of code generation for a particular target machine.

Frontend-Backend Interface

Some compiler collections have been created around carefully designed intermediate representations

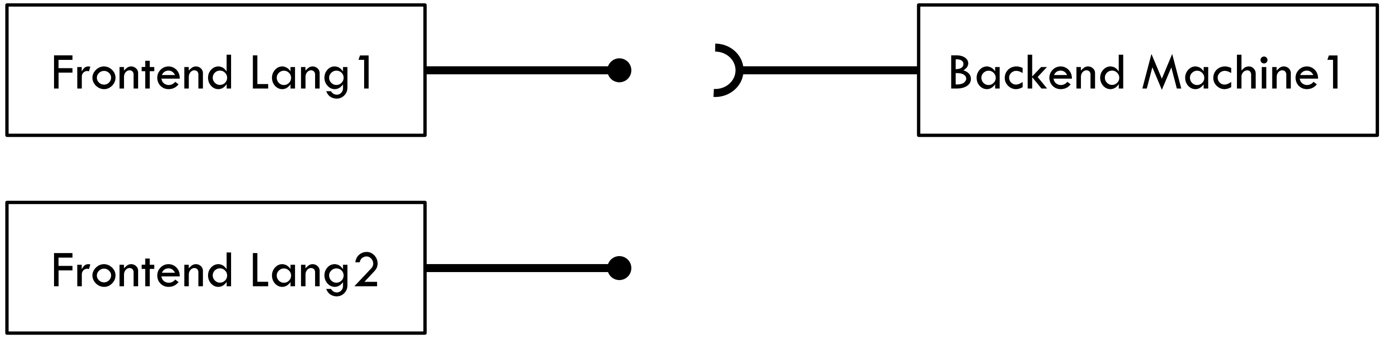
- Frontend for a particular language interfaces with the backend for a certain target machine



Frontend-Backend Interface

Some compiler collections have been created around carefully designed intermediate representations

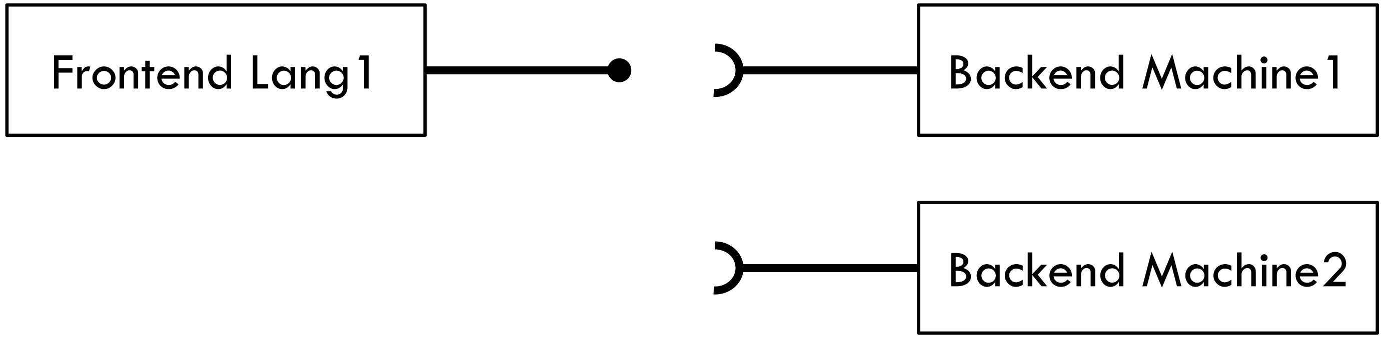
- Frontend for a particular language interfaces with the backend for a certain target machine
- Multiple frontends interact with the backend of a certain target machine



Frontend-Backend Interface

Some compiler collections have been created around carefully designed intermediate representations

- Frontend for a particular language interfaces with the backend for a certain target machine
- A frontend interacts with multiple backends for different target machines



Thank you