COEN 175

Lecture 1: Overview and Lexical Analysis

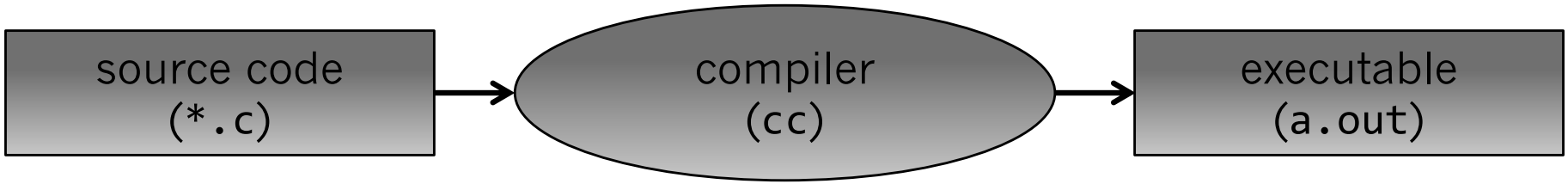
Read: Chapters 1, 2, 3.1–3.3

What is a Compiler?

- A **translator** is a program that takes as input a program written in one language, called the **source**, and produces as output an equivalent program in another language, called the **target**.
- If the source language is **high-level** and the target language is **low-level,** then we call the translator a **compiler**.
  - High-level languages include C, C++, Java, and Fortran.
  - Low-level languages include assembly, binary, and bytecode.

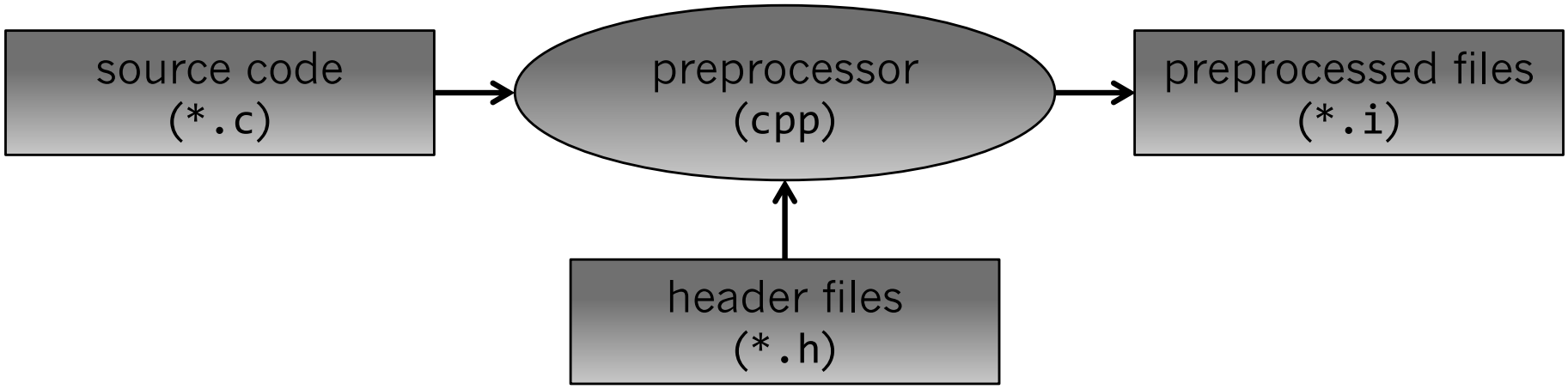
Anatomy of a Compiler

- From the simplest viewpoint, the compiler takes source code and produces an executable.
  - You may have used separate compilation before. But, in reality, there's still a lot more going on behind the scenes.



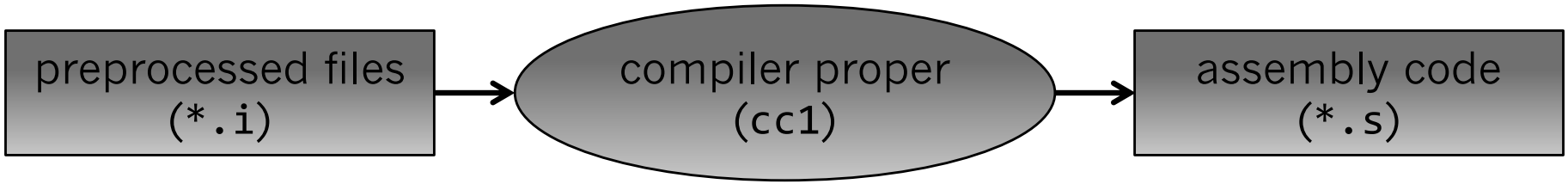
Anatomy of a Compiler

- Actually, the input files need to be preprocessed to expand macros and include files.
  - The actual C language does not contain `#define` or `#include` directives.



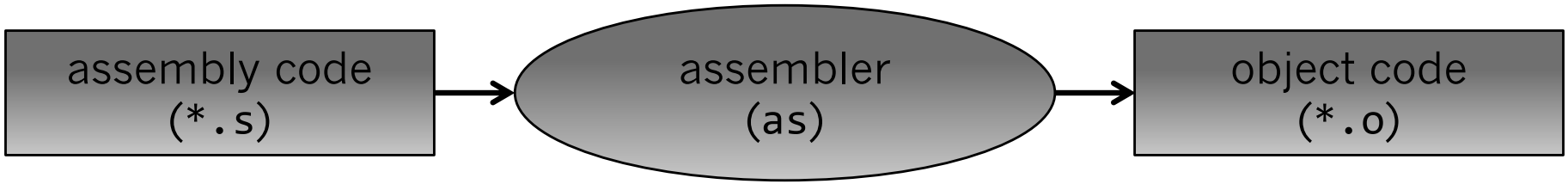
Anatomy of a Compiler

- The preprocessed files can now be compiled by the compiler proper.
  - The compiler proper does not produce binary files, only assembly code.



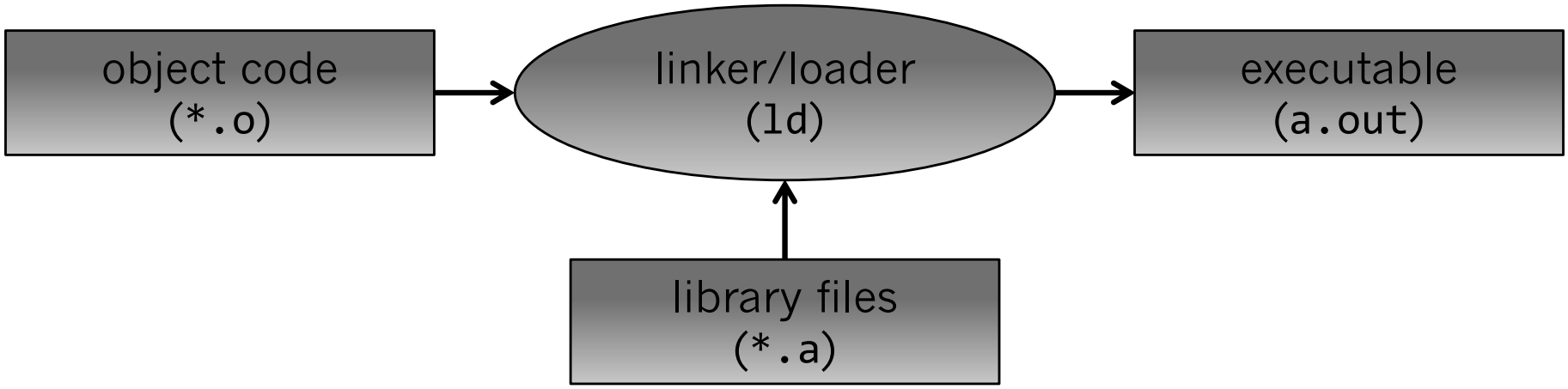
Anatomy of a Compiler

- The assembler produces binary object code, which is still not executable.
  - Many assemblers also let you define macros, but these are taken care of by the assembler itself.



Anatomy of a Compiler

- Finally, the linker/loader produces an executable using library files, which are packaged object files.
  - In dynamic linking, the actual library object files are not placed in the executable. Only a reference is placed there.



Phases of Compilation

- The compiler itself is broken into several phases:
  - Lexical analysis (scanning),
  - Syntax analysis (parsing),
  - Semantic analysis (static checking),
  - Optimization (optional),
  - Code generation (with storage allocation),
  - Instruction scheduling (optional).
- The first three phases are part of the **front end**, which is specific to the source language.
- The last two phases are part of the **back end**, which is specific to the target language.

Retargetable Compilers

- A **retargetable compiler** separates the front end and the back end.
- An **intermediate representation** (IR) connects the front and back ends.
  - An IR is at a higher level than assembly, but not so high as the source language.
  - GCC uses at least three intermediate representations: two tree structures and RTL (register transfer language).
- The optimizer often works on the IR (which means it is independent of the source and target) and is therefore “middle end.”

Lexical Analysis

- Lexical analysis is the process of reading the input program and grouping the characters into meaningful sequences called **tokens**.
- You are already experts in lexical analysis.
- Identify the tokens in the following program:

```
int main(void) {
    printf("Hello, world.\n");
}
```

Lexical Structure

- Specifying lexical structure in words is ambiguous.
- Example: an identifier consists of letters or an underscore followed by digits. Which are legal?
  - `a` ✗
  - `a5` ✗
  - `_` ✗
  - `_56` ✓
- Oops, I thought an identifier consists of **letters**, **or** an underscore followed by **digits**.
- Clearly, a more formal mechanism is needed.

What is a Language?

- A **language** is a set of strings.
  - A string is a sequence of symbols from an alphabet.
  - {0,1} is the binary alphabet.
  - ASCII and Unicode are also alphabets.
  - We let ϵ denote the empty string.
- If L_1 and L_2 are languages, then:
  - $L_1 \cup L_2$ is the set of all strings from L_1 or L_2.
  - $L_1 \cdot L_2$ is the set of all strings from L_1 followed by (**concatenated** with) any string from L_2.
  - L_1^* is the set of all strings from L_1 concatenated zero or more times.

Regular Expressions

- A **regular expression** denotes a set of strings.
- We have rules for building arithmetic expressions:
  - An integer or variable is an expression.
  - The sum or product of two expressions is an expression.
- We also have rules for building regular expressions. The base cases for expressions are:
  - ϵ denotes $\{\epsilon\}$ (the set consisting of the empty string).
  - a denotes $\{a\}$ (the set consisting of the symbol a).

Constructed Expressions

- If r_1 and r_2 are regular expressions for languages L_1 and L_2, respectively, then:
  - $r_1 | r_2$ denotes $L_1 \cup L_2$ (**alternation**).
  - $r_1 r_2$ denotes $L_1 \cdot L_2$ (**concatenation**).
  - r_1^* denotes L_1^* (**closure**).
- These are in order of increasing precedence:
  - *foo*|*bar* denotes (*foo*)|(*bar*) and not *fo* (*o*|*b*) *ar*.
  - *foo** denotes *fo* (*o**) and not (*foo*)*.
- We also introduce some shorthands:
  - r^+ denotes rr^* (one or more).
  - $r^?$ denotes $r \mid \epsilon$ (zero or one).
  - r^n denotes $rr \cdots r$ (exactly n times).

Character Classes

- A **character class** is another useful shorthand:
  - [*aeiou*] denotes *a*|*e*|*i*|*o*|*u*.
  - [*x*–*z*] denotes *x*|*y*|*z*.
  - [*abcx*–*z*] denotes *a*|*b*|*c*|*x*|*y*|*z*.
- A character class can also be **negated**:
  - [^*a*] denotes any symbol other than *a*.
  - [^0–9] denotes any symbol other than a digit.
- A character class can also be used with operators:
  - [*abc*]* denotes (*a*|*b*|*c*)*.

Example Expressions

- The keyword **for** in the C language.
  - Answer: *for*.
- The keyword **do** in Pascal, which is case-insensitive.
  - An answer: *do*|*Do*|*dO*|*DO*.
  - A better answer: (*d*|*D*)(*o*|*O*).
  - An even better answer: [*dD*][*oO*].
- An identifier in the C language.
  - Answer: [*a–zA–Z*_][*a–zA–Z*_0–9]*.
- A floating-point literal in the Pascal language.
  - Partial answer: [0–9]+.[0–9]+([*eE*][+–]?[0–9]+)?.