Lecture Overview

1. Administrative
2. Remarks
3. Definition of Compiler
4. Typical Construction of a Compiler
5. Environment of a Compiler

Administrative matters

This course is to be conducted face-to-face. The lectures will not be recorded.

The TAs for this course are Hemang Bhanushali (hba42@sfu.ca), and Abhishek Nair (asn4@sfu.ca).

The course website is https://canvas.sfu.ca/courses/70682/.
Please make use of the discussion board there.

Marking will be on the basis of the following:

Milestones (project)	3 x 20%
Midterm	15%
Final	25%

Milestone marks are graded by running your code on a suite of tests; different tests may have different weights.

The midterm will be conducted during scheduled class time on October 26. The final will be at the time that the registrar's office schedules. Registrar's schedules the exams about a month or so into the term. The exams cover theoretical concepts, **not** project details.

Academic dishonesty is a serious matter and I often catch students who engage in it. Please don't try your luck. Read the section on Academic Honesty on the course website.

Be aware that questions asked via email or in office hours may end up (anonymized) on the course website, if I feel that everyone should know the answer.

Official communications about the course will appear via email, via Canvas announcement, and in lecture.

Remarks

This course is a heavy load; it involves a **lot** of programming. I would advise against attempting it early in your 3rd year. Get some other 300-level courses done before this course.

The project in this course is done in **java**. If you do not know java:

It is similar to C++ but has some crucial differences. Learn it right away, not next week. Pay particular attention to the object model (how to create objects and what object-type variables are), instanceof, enums, interfaces, and polymorphism in general.

Did I mention that this is a heavy course?

The project in this course is to take a small compiler that I provide you with and turn it into a bigger compiler. This is done in increments, with each milestone increasing the features of the language that the compiler handles. Milestones are marked by running test cases – hundreds of them - on your compiler and the code your compiler produces. You will not be given the test cases beforehand; it is your job to interpret the specification correctly.

Do not fall behind on the project! The milestones naturally build on one another. Under no circumstances will I provide code for a milestone, so if you fall behind you still will have to complete a milestone before going on to another. This simulates programming in real life, where versions of a program build upon another, rather than being the simple throwaway programs found in most of our courses. This is the way I develop compilers: iterative deepening, not phase-by-phase. Specs often change, and rarely does one have a complete specification at the beginning of a large project.

Developing in java can be sped up if you get a good refactoring editor and learn how to use it (particularly what the refactorings do). Eclipse (at eclipse.org) is a refactoring editor. IntelliJ Idea (at https://www.jetbrains.com/idea/) is another.

This course requires a lot of programming. I've seen students (with similar milestones to yours) produce 10,000 or more lines of code in a semester. It can probably be done in half that, though.

The first milestone eases you into the provided compiler, basically taking you on a tour of it where you get to imitate some of its code to implement the new features.

Lectures in this course are driven by the project. There are mainly-project lectures and mainly-"theory" lectures. In the beginning of the course, I will introduce project material in the order in which you need it, rather than the phase-by-phase ordering typical of books on the topic. As the course progresses, I will introduce more and more theoretical material which does correspond directly to the book. Use the textbook as a reference when you need a deeper understanding of a topic than I go into during lecture. It's a good text.

Courses vary in the amount of work they require. This is a heavy course: budget at least two courses worth of work. Think of it as getting more for your tuition dollars.

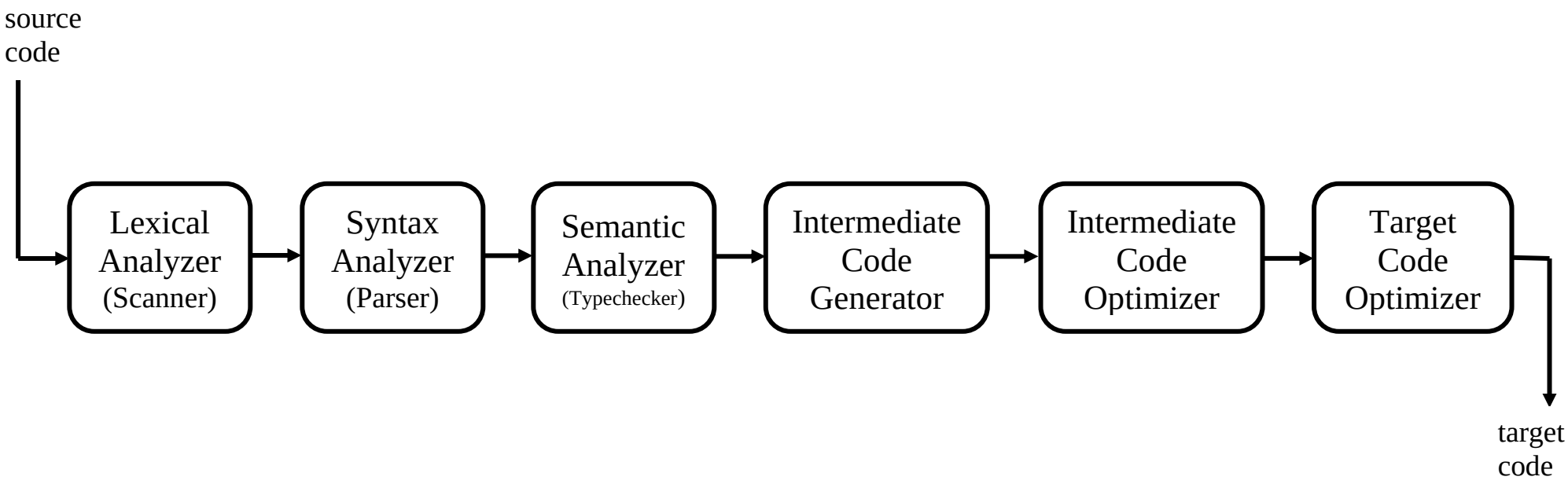
Compiler: a definition or two

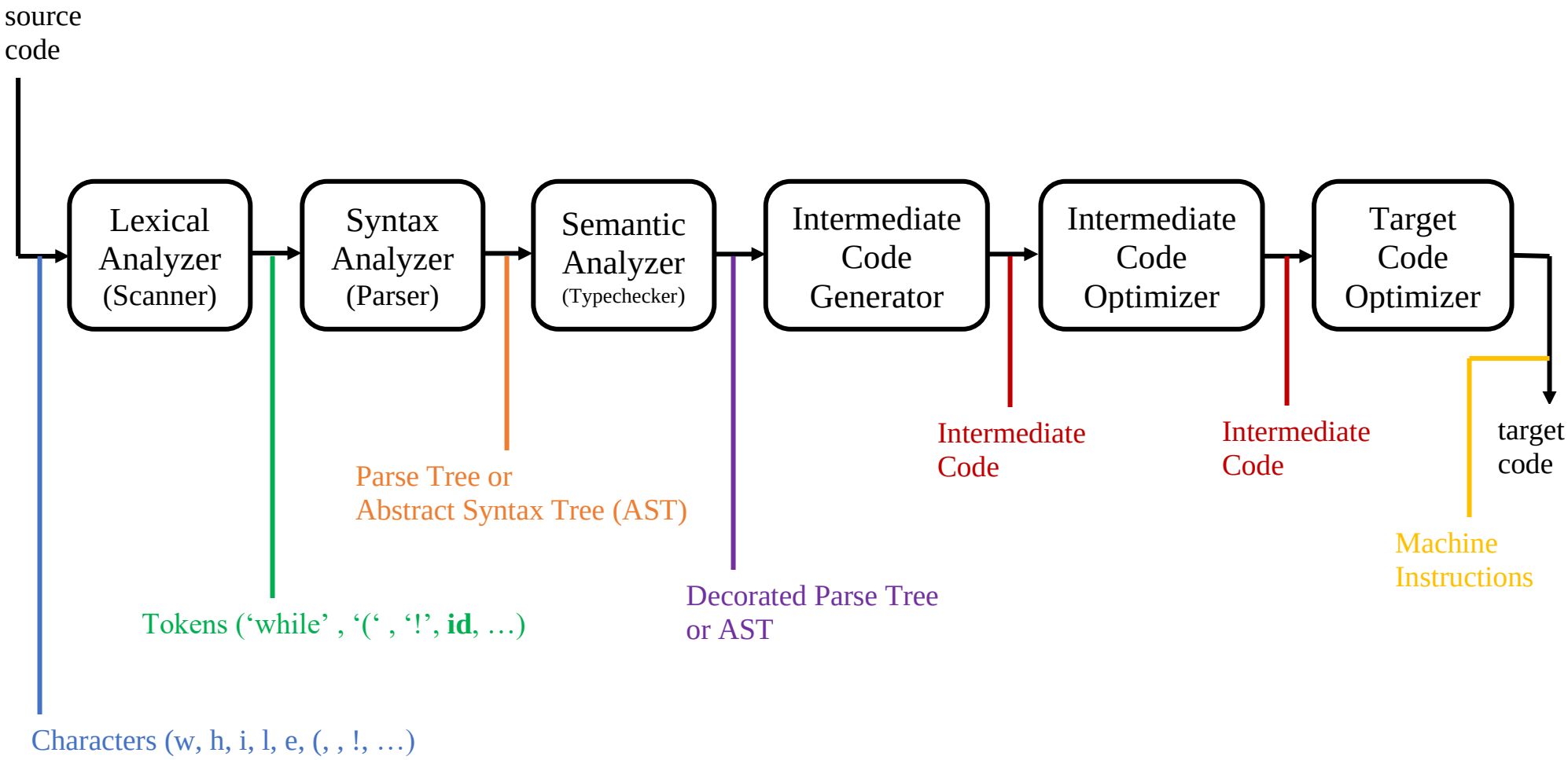
In general, a **compiler** is a program that converts one language to another. This is quite a broad definition. It includes natural-language translation, for instance, or a program to convert java to C++.

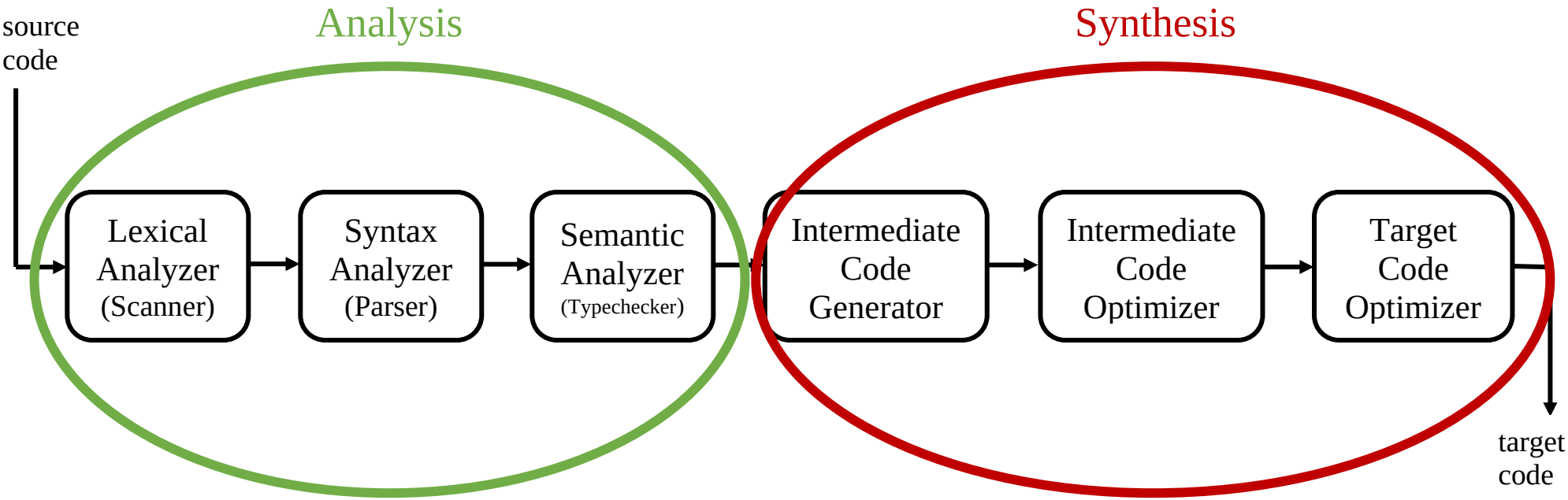
In formal language theory, a **language** is a set of strings. These strings are the valid sentences (say) in a natural language, or valid programs in a computer language.

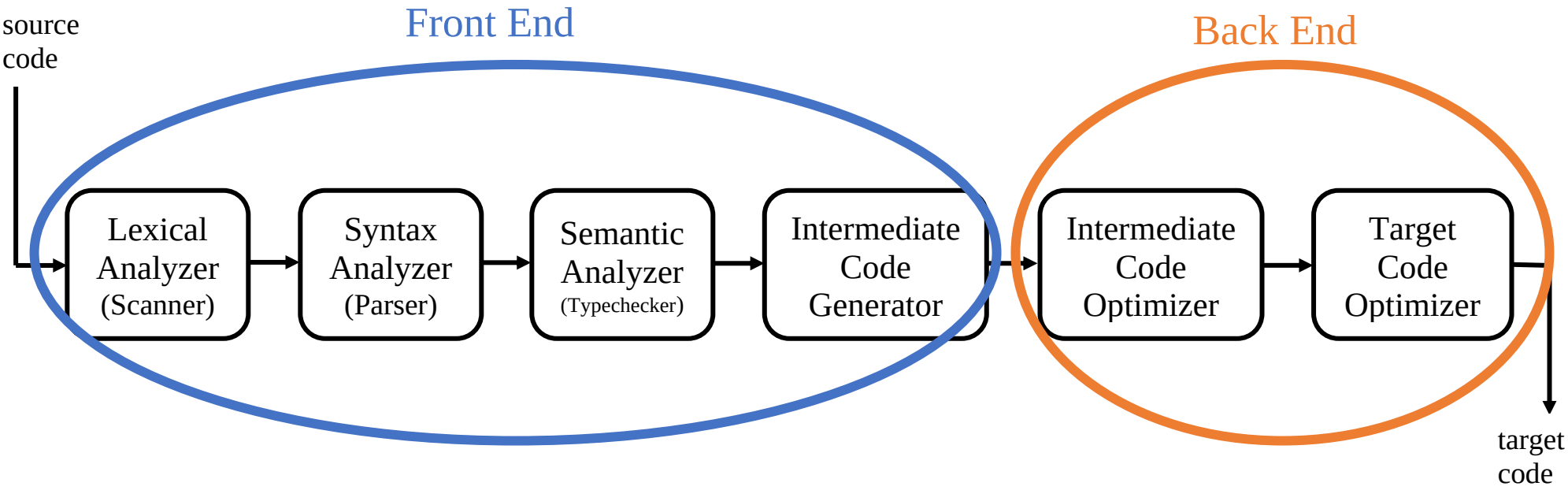
A much more specific definition of compiler, and **the one we will use in this course**, is a program that converts from (a) a high-level general-purpose programming language to (b) a low-level machine or intermediate language.

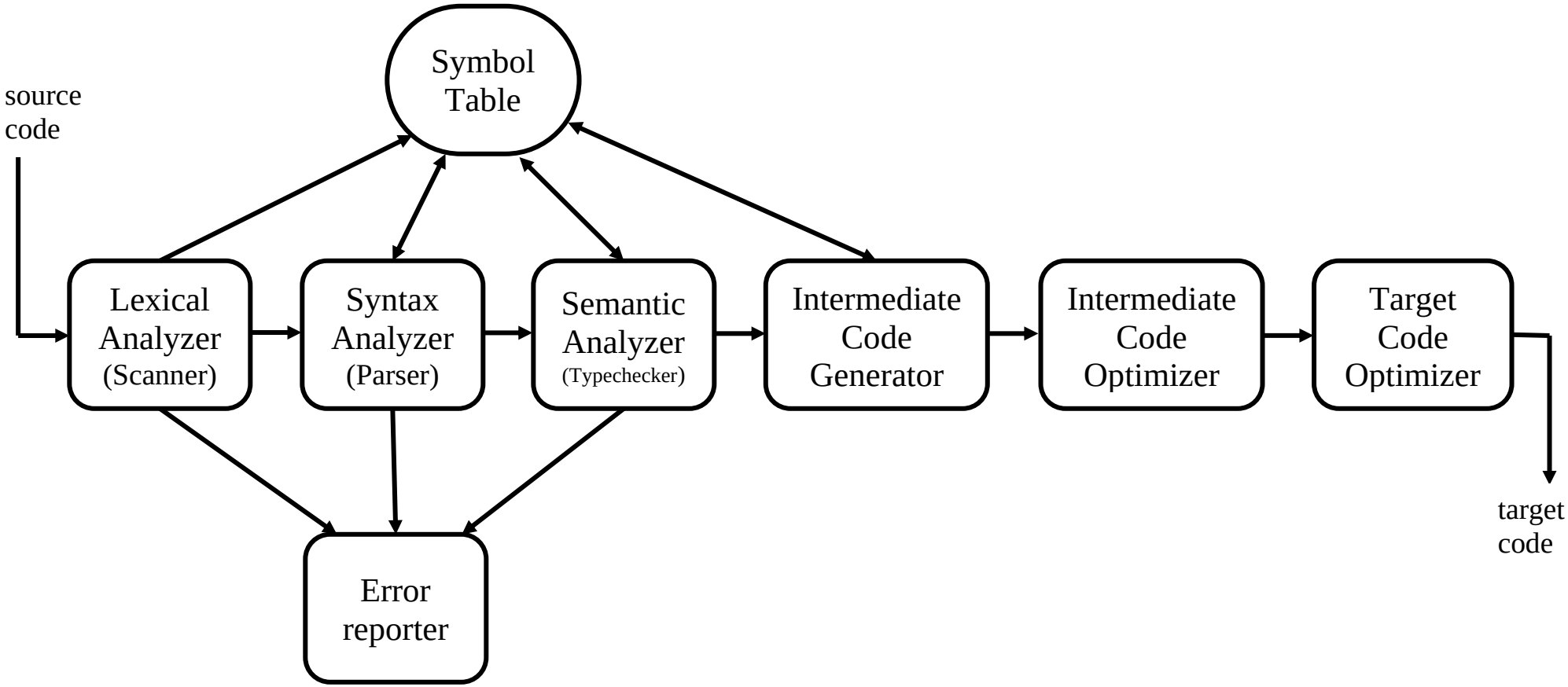
A typical compiler architecture




source code
Lexical Analyzer (Scanner)
Syntax Analyzer (Parser)
Semantic Analyzer (Typechecker)
Intermediate Code Generator
Intermediate Code Optimizer
Target Code Optimizer
target code
Characters (w, h, i, l, e, (, , !, …)
Tokens (‘while’ , ‘(‘ , ‘!’, id, …)
Parse Tree or Abstract Syntax Tree (AST)
Decorated Parse Tree or AST
Intermediate Code
Intermediate Code
Machine Instructions


source
code
Analysis
Synthesis
Lexical
Analyzer
(Scanner)
Syntax
Analyzer
(Parser)
Semantic
Analyzer
(Typechecker)
Intermediate
Code
Generator
Intermediate
Code
Optimizer
Target
Code
Optimizer
target
code


source
code
Front End
Back End
Lexical
Analyzer
(Scanner)
Syntax
Analyzer
(Parser)
Semantic
Analyzer
(Typechecker)
Intermediate
Code
Generator
Intermediate
Code
Optimizer
Target
Code
Optimizer
target
code


Symbol Table
source code
Lexical Analyzer (Scanner)
Syntax Analyzer (Parser)
Semantic Analyzer (Typechecker)
Intermediate Code Generator
Intermediate Code Optimizer
Target Code Optimizer
target code
Error reporter

The environment of a compiler

An **interpreter** is like a compiler, except that it executes the actions specified by the source program rather than creating target code. Interpreted programs tend to be slower than compiled ones.

An **assembler** is a (broad-sense) compiler that translates from an assembly language to machine language.

A **linker** is a program that takes a target-language portion of a program and combines it with other portions of the program that were compiled separately. These separately-compiled portions include **libraries**.

A **loader** is a program that takes a target-language program out of a file and places it into the memory of a computer, then starts it running.

An **integrated development environment (IDE)** is typically a compiler, linker, loader, and an editor combined and working with each other.

When something happens while the compiler is running, we call it compile-time or **static**.

When something happens while the program the compiler produced is running, we call it run-time or **dynamic**.

There are -time terms for linking and loading, too: link-time and load-time.

For example, we may link a program together with all its parts and libraries right after the compiler finishes (often as part of the compiler) and this is called **static linking.**

If we link the program together when loading it, it's **load-time linking.**

If we link parts of the program together while the program is running, it's **dynamic linking.** This is accompanied by **dynamic loading**, which pulls in some parts of the program off of a long-term storage medium while the program is running.