Formal Languages and Compilers

Introduction

Alessandro Barenghi & Michele Scandale

January 8, 2020

Topics

In these 5 lessons we will see:

- how theoretical concepts (re, automata, bottom-up parsing, attribute grammars) are applied in compiler construction
- the internal organization and workflow of a compiler
- how to modify and extend a simple compiler

These concepts can be applied also in everyday work.

Exam

The lab is $\frac{1}{5}$ of the exam score:

- you need to pass the lab exam in order to pass the whole exam
- the minimum score to pass the lab test is $\frac{15}{30}$

The lab exam is usually held before the theory exam.
You can consult anything you want during the exam:

- except your classmates and your laptops/phones

Requirements & Assumptions

You are expected to meet or exceed the following requirements:

- have a good command of the C language
- know how a C construct is translated into assembly
- be able to use a compilation toolchain (e.g. `gcc`, `make`, ...)
- to employ all the above is a thoughtful way (*your brain must be turned on*)

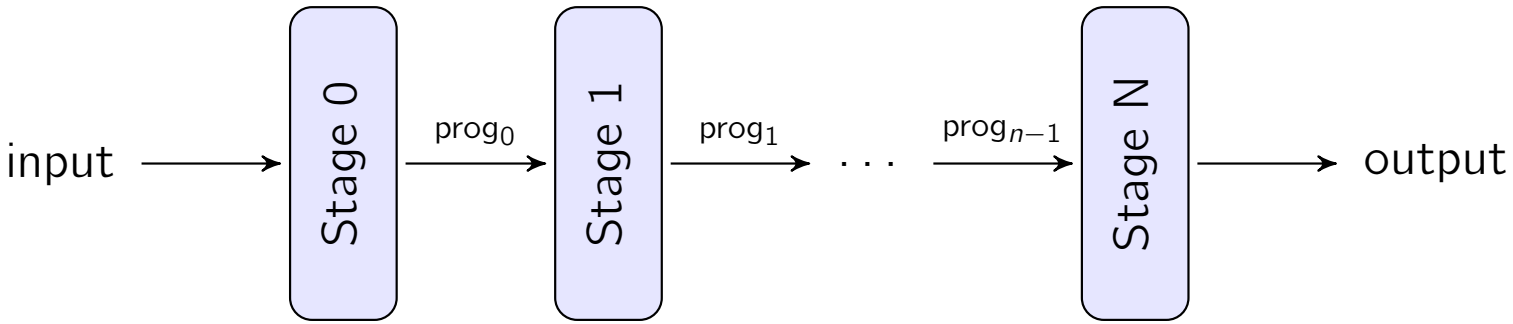
What a does a compiler do?

The purpose of a compiler is:

- it translates a program written in a language L_0 into a semantically equivalent program expressed in language L_1.

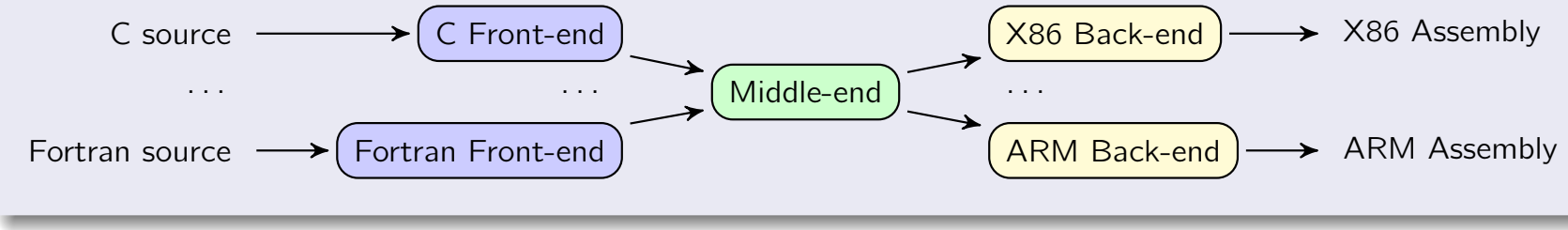
A compiler is organized as a pipeline:

- each stage applies a transformation to the input program producing an output program



Compiler pipeline

Generic compiler structure



Each stage has its own purpose:

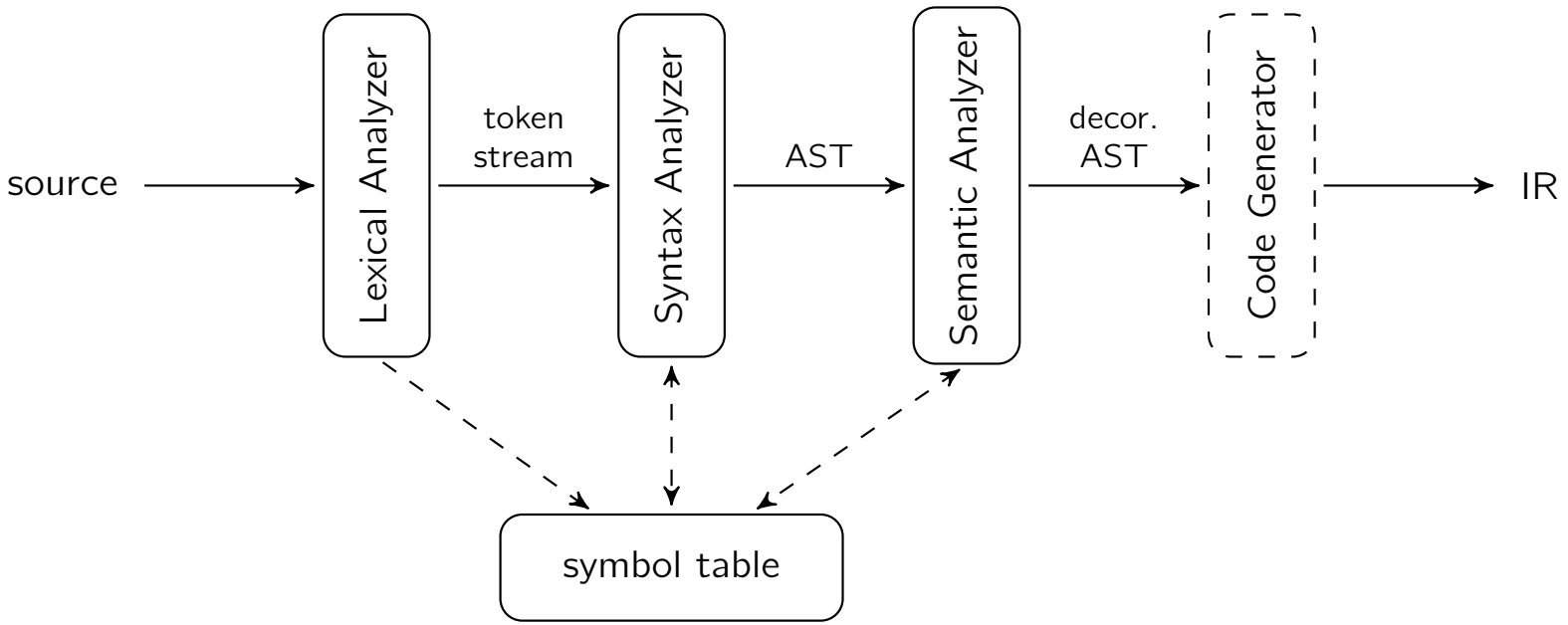
front-end converts from source language into an intermediate form

middle-end applies transformations and optimizations on the intermediate form

back-end convert from the intermediate form into target machine language

Front-end structure

The front-end purpose is to translate a program from a source language into an intermediate form.



Main tasks:

- recognize language constructs
- verify syntactical correctness
- verify semantical correctness

A real world example: GCC

Many frontends:

- various languages (C, C++, Objective C, Java, Fortran, Ada, Go)
- most of them target the *GENERIC* language

Common lowering to the intermediate representation:

- *GIMPLE* and *GIMPLE-SSA* languages

At last:

- translation to *RTL* language
- back-ends emit native intructions

Think First

We will see a couple of concepts:

- tokens
- statements
- control structures
- ...

You already know how to *use* them:

- you only need to understand how to *recognize* and *compile* them

Many statements are just a variation of a common idiom:

- *syntactic sugar* around a concept

UNIX is your friend

Every UNIX-derived OS contains a lot of compiler-related tools:

- to automate compilers development
- to automate tedious tasks

Only a few will work on compilers, almost all, sooner or later, will find a tedious task:

- count the occurrences of a pattern
- substitute a parametric sentence with another
- ...

Tools (`grep`, `sed`, `awk`) can automate your work!