

COMPILER CONSTRUCTION

Overview

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Spring 2017



Chapter 1

Overview

Introduction

- Compilers act as *translators*
  - Transforming human-oriented **programming languages** into computer-oriented **machine languages**

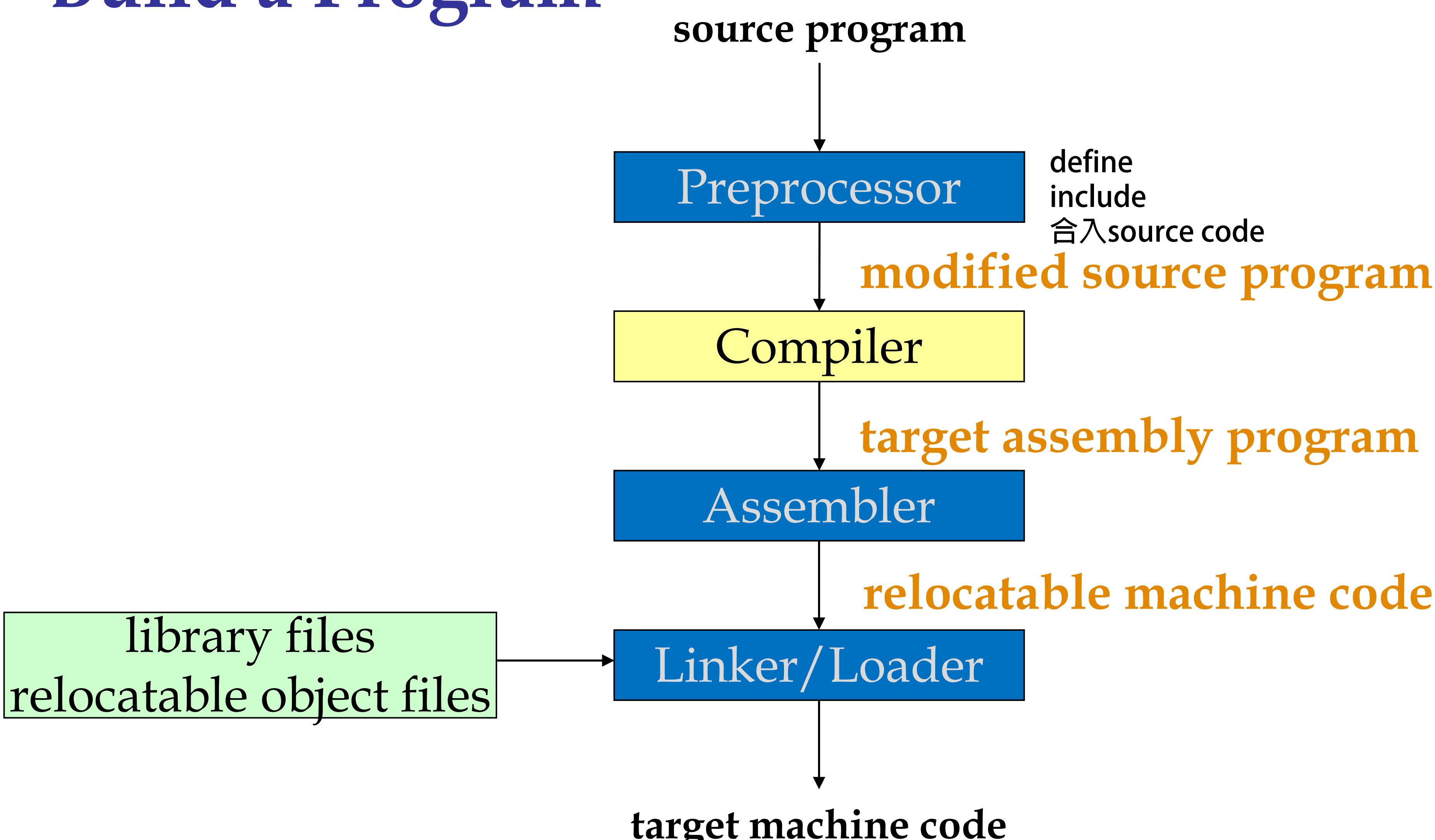
Figure 1.1: A user's view of a compiler.



Introduction (Cont'd)

- A **compiler** is a program that
  - accepts, as input, a program text in a certain **programming language** (*source* language), and
  - produces, as output, a program text in an **assembly language** (*target* language),
  - which will later be assembled by the **assembler** into machine language
- Example:
  - You **build** the C/C++ programs on your laptop using the Microsoft Visual Studio
  - The built binary is executed on the Intel CPU on the machine
  - Input: C/C++ programs; output: x86 machine code

Build a Program



Static vs. Dynamic Build

- Try for example:

  gcc foo.c

- By default, the command generates dynamic built binary
  - Link to the shared libraries (e.g., .dll or .so) at runtime
  - Has smaller file size of the built binary
  - Check it with **file** command
- Statically build:

  gcc –static foo.c

  - Include the static libraries (e.g., .a) during **linking**
  - Has larger file size of the built binary



Machine Code Generated by Compilers

- While the issue of the accepted source language is indeed simple, there are many alternatives in describing <u>*the **output** of a compiler*</u>
  - By the **type** (kind) of machine code they generate
    1. Pure Machine Code
    2. Augmented Machine Code
    3. Virtual Machine Code
  - By the **format** of the target code they generate



Three Types of Generated Code (1/4)

- **Pure machine code**
  - Compiler may generate code for **a particular machine's instruction set**
  - without assuming the existence of any operating system or library routines

- Pure machine code is used in compilers for **system implementation languages**
  - which are for implementing operating systems or embedded applications (e.g., bare-metal programs)
- This form of target code can execute on bare hardware without dependence on any other software



Three Types of Generated Code (2/4)

- **Augmented machine code**
  - Compilers generate code for a machine architecture that is **augmented** with:
  1. operating system routines and
  2. runtime language support routines
  - It involves something related to **ABI**

- The execution of a program generated by such a compiler requires:
  - a particular operating system be present on the target machine and
  - a collection of language-specific runtime support routines be available to the program
  - E.g., I/O, storage allocation, mathematical functions, etc.

Three Types of Generated Code (3/4)

- **Virtual machine code**
  - Compilers generate virtual machine code that is composed entirely of **virtual machine instructions**
  - Adopted in the programming HWs of our course

- **Portability** is achieved by writing just one **virtual machine (VM)** interpreter for all the target architectures
  - That code can run on any architecture for which a VM interpreter is available
    - For example, the VM for Java, Java virtual machine (JVM), has a JVM interpreter

Three Types of Generated Code (4/4)

In summary

- Most compilers generate code that interfaces with:
  - runtime libraries, operating system utilities, and other software components
- VMs can enhance:
  - compiler portability and
  - increase consistency of program execution across diverse target architectures



Machine Code Generated by Compilers (Cont'd)

- While the issue of the accepted source language is indeed simple, there are many alternatives in describing the **output** of a compiler
  - By the **type** (kind) of machine code they generate
  - By the **format** of the target code they generate
    1. Assembly or other source formats
    2. Relocatable binary
    3. Absolute binary



Three Formats of Generated Code (1/3)

- **Assembly language (source) format**
  - Simplify and modularize translation
  - Is relatively easy to scrutinize
    - For students to learn and for system designers to inspect the code
- Example:
  gcc –S foo.c
  - **-S** flag asks the **gcc** to stop after the stage of compilation proper; do not assemble
    - The output is in the form of an assembler code file for each non-assembler input file specified
    - By default, the assembler file name for a source file is made by replacing the suffix `.c', `.i', etc., with `**.s**'



Three Formats of Generated Code (2/3)

- **Relocatable binary format**
  - which is essentially the form of code that most assemblers generate; as can be done by compiler
  - External **references**, local instruction **addresses**, and data addresses are not yet bound
  - Instead, addresses are assigned relative either to the beginning of the module or to some symbolically named locations
  - The latter alternative makes it easy to group together code sequences or data areas
  - A **linkage step** is required to incorporate any support libraries as well as other separately compiled routines referenced from within a compiled program
  - The result is an **absolute binary format** that is executable



Three Formats of Generated Code (2/3)

- **Relocatable binary format**
  - Both relocatable binary and assembly language formats allow **modular compilation**
    - the decomposition of a large program into separately compiled pieces
  - They also allow **cross-language support**
    - incorporation of assembler code and code written and compiled in other high-level languages
    - Such code can include I/O, storage allocation, and math libraries that supply functionality regarded as part of the language's definition



Three Formats of Generated Code (3/3)

- **Absolute Binary Format**
  - The binary can be directly executed when the compiler is finished
  - This process is usually **faster** than the other approaches
    - However, the ability to **interface with other code** may be limited
  - Is useful for student exercises and prototyping use,
    - where frequent changes are the rule and compilation costs far exceed execution costs

Interpreter

- To an interpreter, **a program is merely *input*** that can be arbitrarily manipulated, just like any other data
- The focus of control during execution resides in the interpreter, not in the user program
  - i.e., the user program is passive rather than active

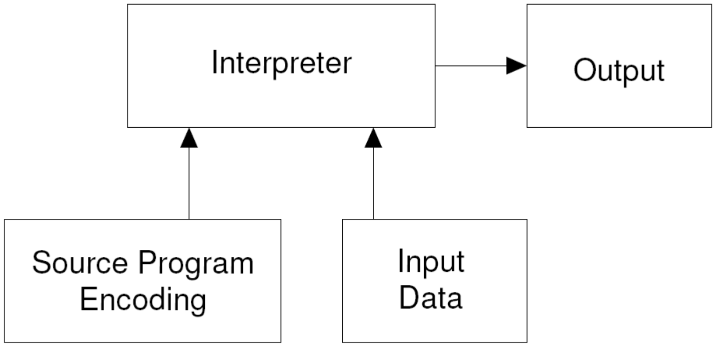
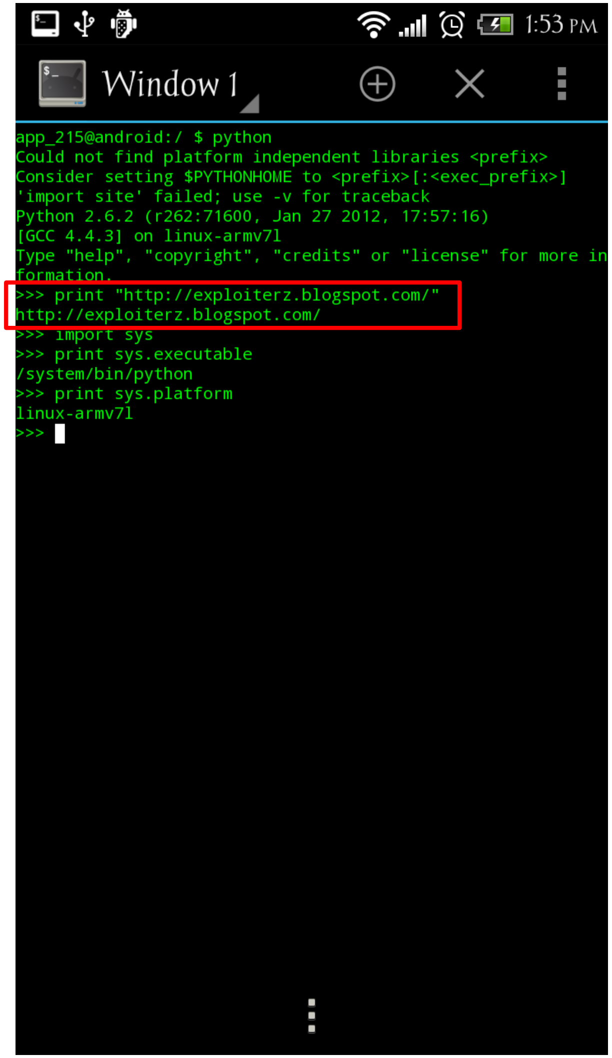


Figure 1.3: An interpreter.



Example: Python Interpreter for Android

- Interpreter directly **interprets** (executes) the **source program** that reads inputs and writes outputs
- Example:
  - You type >>>print "URL"
  - It prints the "URL"



Courtesy of http://exploiterz.blogspot.tw/2013/08/how-to-provide-python-support-in.html



Compiler vs. Interpreter

- Interpreters differ from compilers in that
  - they **execute** programs without explicitly performing much **translation**

- Using compiler involves two phases:
  1. The **compilation phase** generates target program from source program
  2. The **execution phase** executes the target program



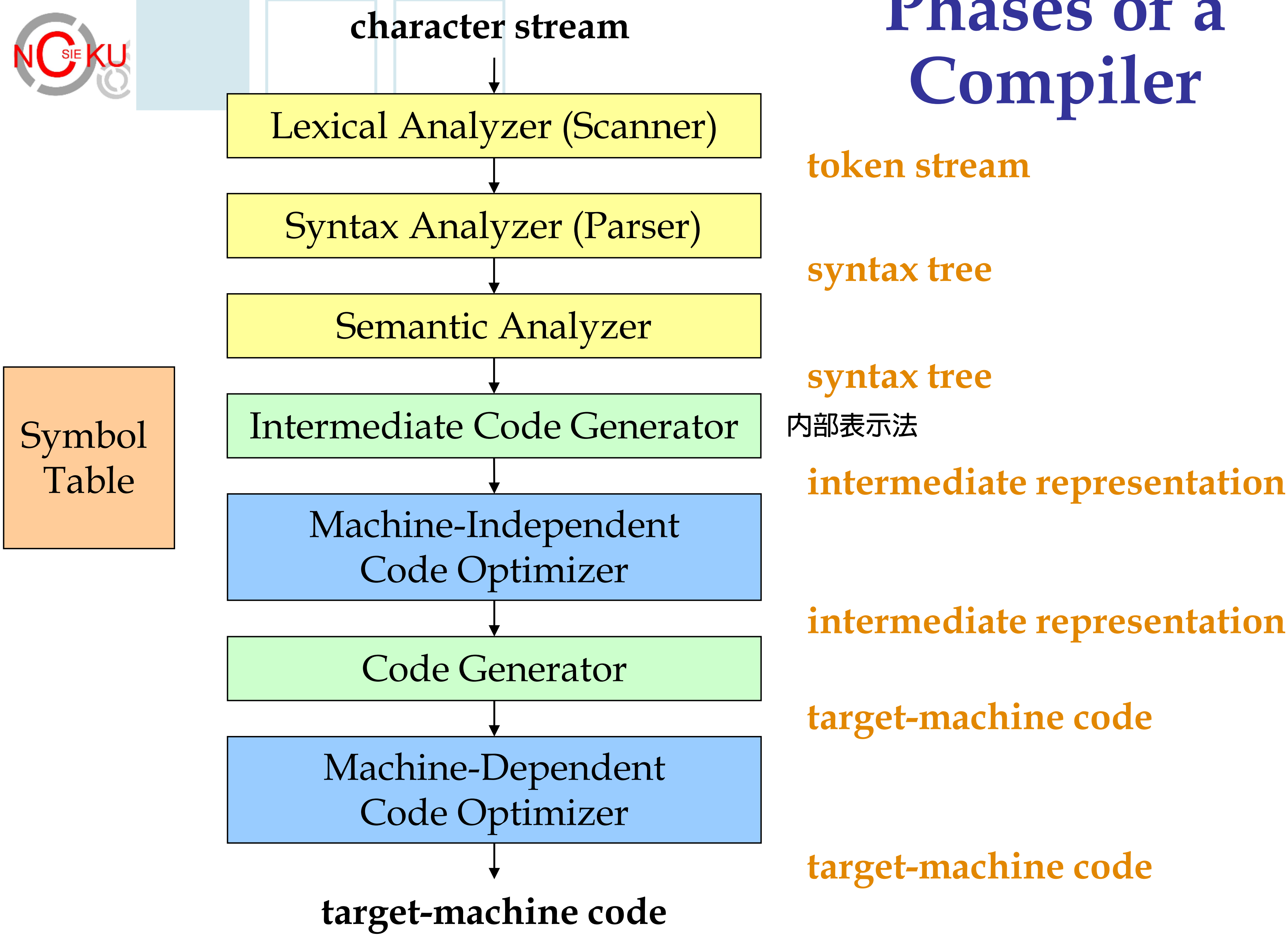
Organization of a Compiler

- Compilers generally perform the following tasks:

  1. **Analysis** of source program, such as scanning and parsing

  2. **Synthesis** of target program, such as code generation

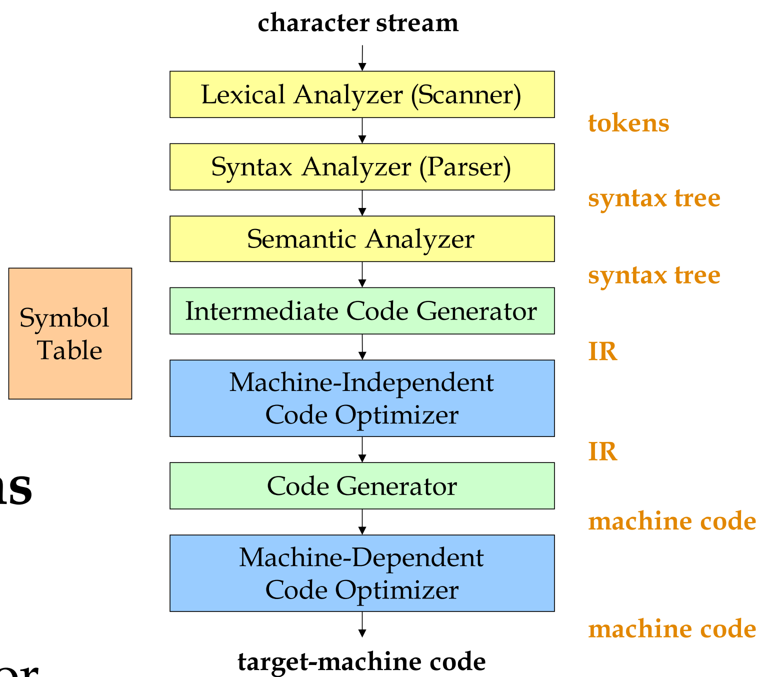
Phases of a Compiler





The Phases of a Compiler

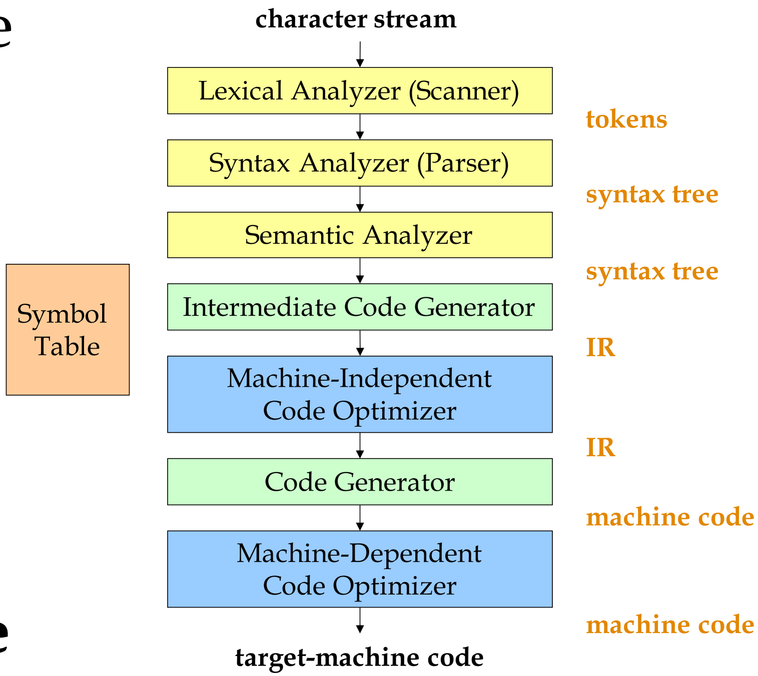
- The compilation process is driven by the **syntactic structure** of the source program, as recognized by the parser
  - Almost all modern compilers are **syntax-directed**
- Most compilers distill the source program's structure into an **abstract syntax tree** (AST)
  - that omits unnecessary syntactic detail
- The parser builds the AST out of **tokens**
  - which is the elementary symbols used to define a programming language syntax
  - Recognition of syntactic structure is a major part of the **syntax analysis** task





The Phases of a Compiler (Cont'd)

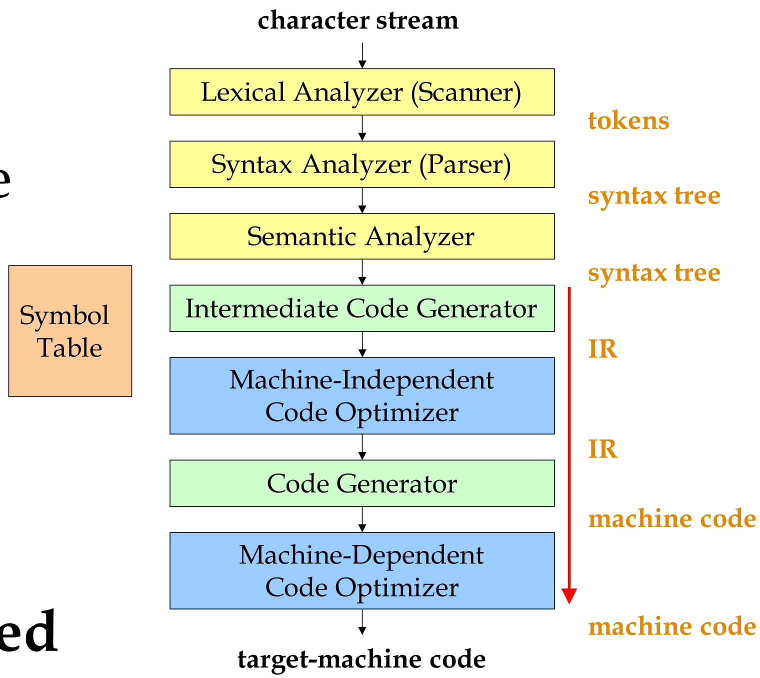
- **Semantic analysis**
  - examines the meaning (semantics) of the program on the basis of its syntactic structure
- It plays a dual role:
  - It finishes the analysis task by performing a variety of correctness checks
    - for example, enforcing type and scope rules
  - It also begins the **synthesis phase**





The Phases of a Compiler (Cont'd)

- In the synthesis phase
  - Source language constructs are translated into an **intermediate representation** (IR) of the program
  - Some compilers generate target code directly
- IR serves as input to a code generator component
  - which actually produces the desired machine-language program
  - The IR may optionally be **transformed** by an optimizer so that a more efficient program may be generated



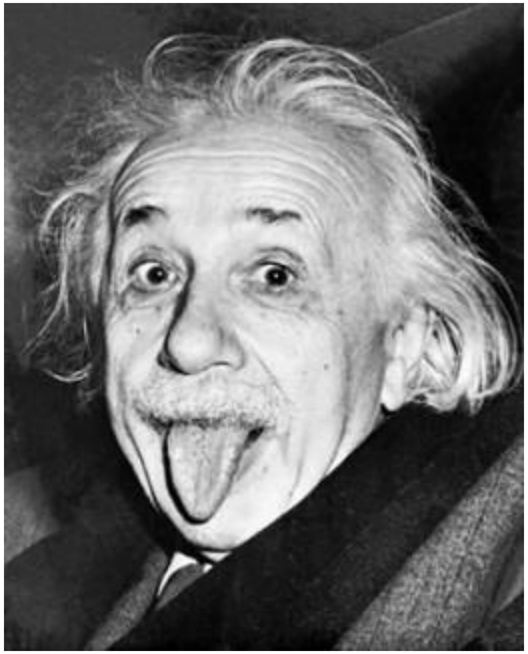


The Grouping of Phases

- Compiler front and back ends:
  - Frontend: analysis (machine independent)
  - **Backend**: synthesis (machine dependent)

- Compiler passes:
  - A collection of phases is done only once (single pass) or multiple times (multi pass)
  - Single pass: usually requires everything to be defined before being used in source program
  - Multi pass: compiler may have to keep entire program representation in memory





In fact, what we will learn in this course is the basis of today's compiler-based tools!!!

Courtesy of paulocoelhoblog.com



Variants of Modern Compilers

- Language support
  - High-level scripting languages
  - E.g., JavaScript, Python, Java (e.g., Android systems) etc.
- Profiling
  - Performance analysis for generated program
  - E.g., Gprof
- Debugging
  - Examine the errors in programs, such as memory leaks
  - E.g., Valgrind, Sanitizer
- Program analysis
  - Characterize the program behaviors, such as control flow
  - E.g., Pin, Contech
- Optimizing compilers
  - Generate faster program binaries for computers, including multicore, heterogeneous multicore platforms
  - E.g., LLVM, OmpSs
- Retarget able compilers
  - E.g., LLVM



QUESTIONS?