Course: Compiler Construction

Chapter 1 Introduction

Muhammad Haggag, Ph.D.

Computer Science Department
Faculty of Computers and Information
Mansoura University

- https://tinyurl.com/COMCON2020

Outlines

Overview and History (1)

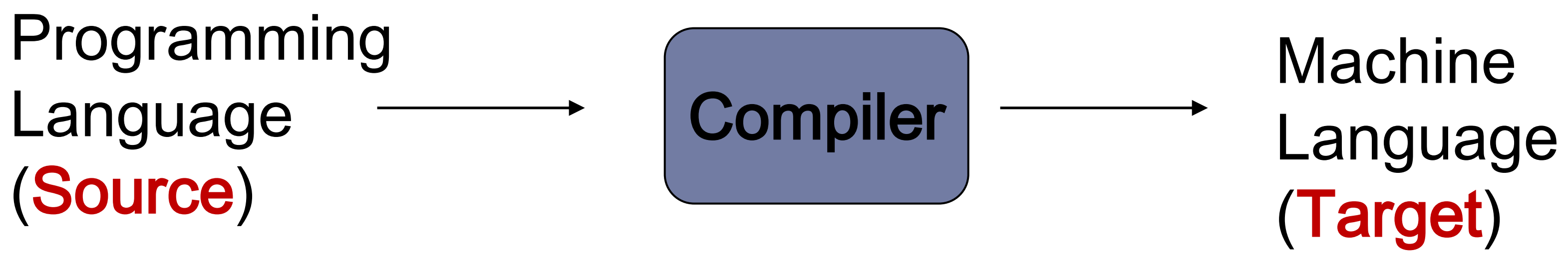
- Cause
  - Software for early computers was written in assembly language
  - The benefits of reusing software on different CPUs started to become significantly greater than the cost of writing a compiler
  - Each different CPU has own Assembly language
- The first real compiler
  - FORTRAN compilers of the late 1950s

Overview and History (2)

- Compiler technology
  - is more broadly applicable and has been employed in rather unexpected areas.
    - Text-formatting languages,
      Silicon compiler for the creation of VLSI circuits
    - Command languages of OS
    - Query languages of Database systems

What Do Compilers Do (1)

- A compiler acts as a translator, transforming human-oriented programming languages into computer-oriented machine languages.
- Ignore machine-dependent details for programmer



What Do Compilers Do (2)

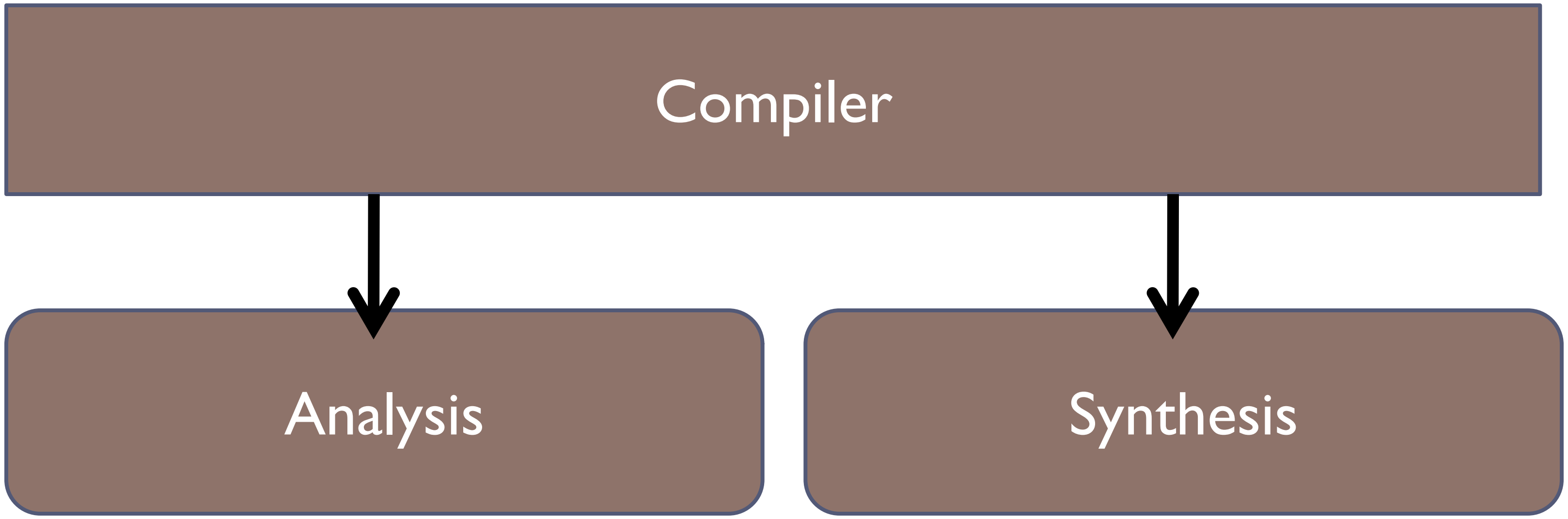
- Compilers may generate three types of code:
  - Pure Machine Code
    - Machine instruction set without assuming the existence of any operating system or library.
    - Mostly being OS or embedded applications.
  - Augmented Machine Code
    - Code with OS routines and runtime support routines.
    - More often
  - Virtual Machine Code
    - Virtual instructions, can be run on any architecture with a virtual machine interpreter or a just-in-time compiler
    - Ex. Java, C#

What Do Compilers Do (3)

- Another way that compilers differ from one another is in the <u>format of the target</u> machine code they generate:
  - Assembly or other source format
  - Relocatable binary (.obj, .dll)
    - Relative address
    - A linkage step is required
  - Absolute binary (.exe)
    - Absolute address
    - Can be executed directly

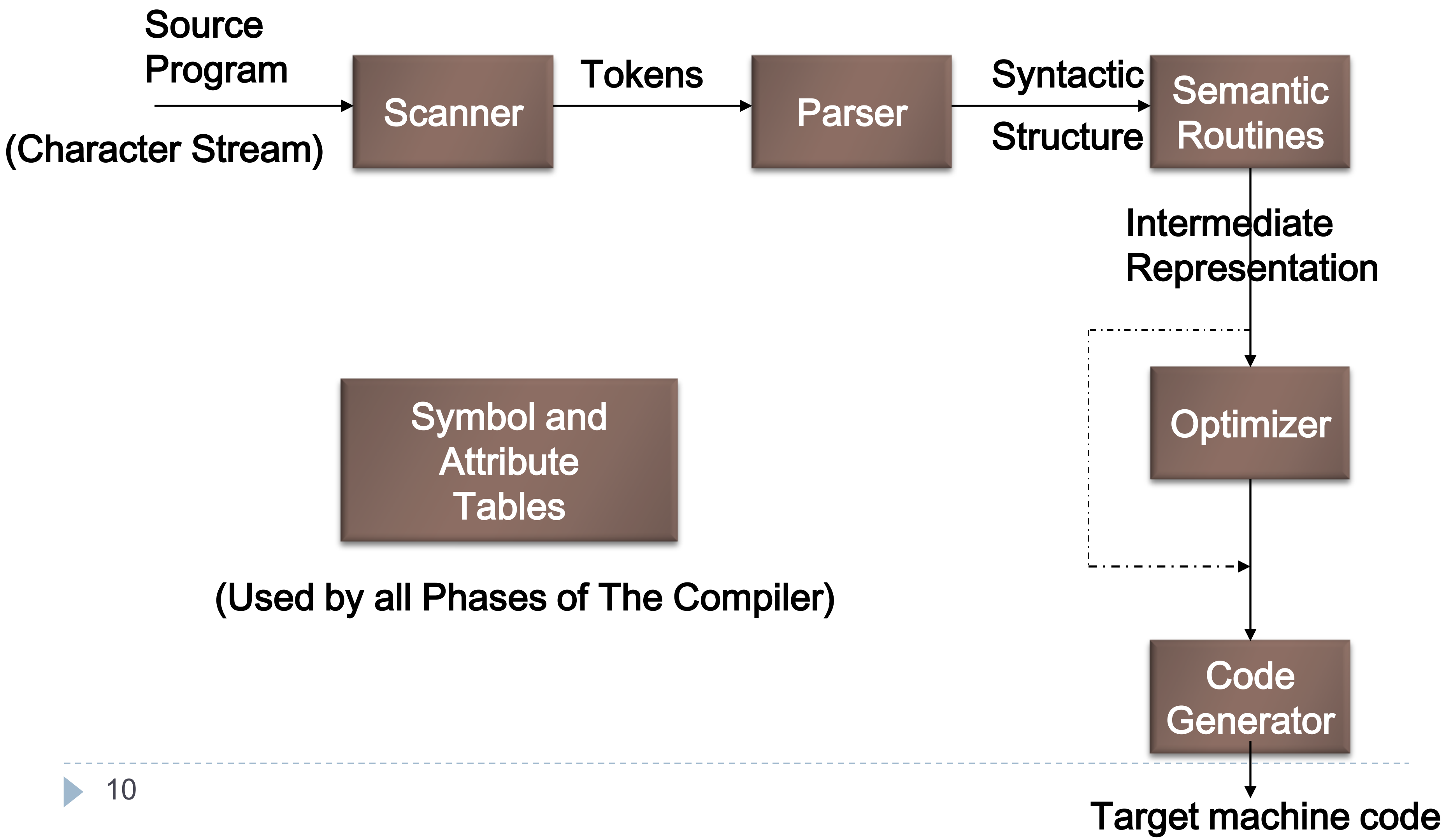
The Structure of a Compiler (1)

- Any compiler must perform two major tasks

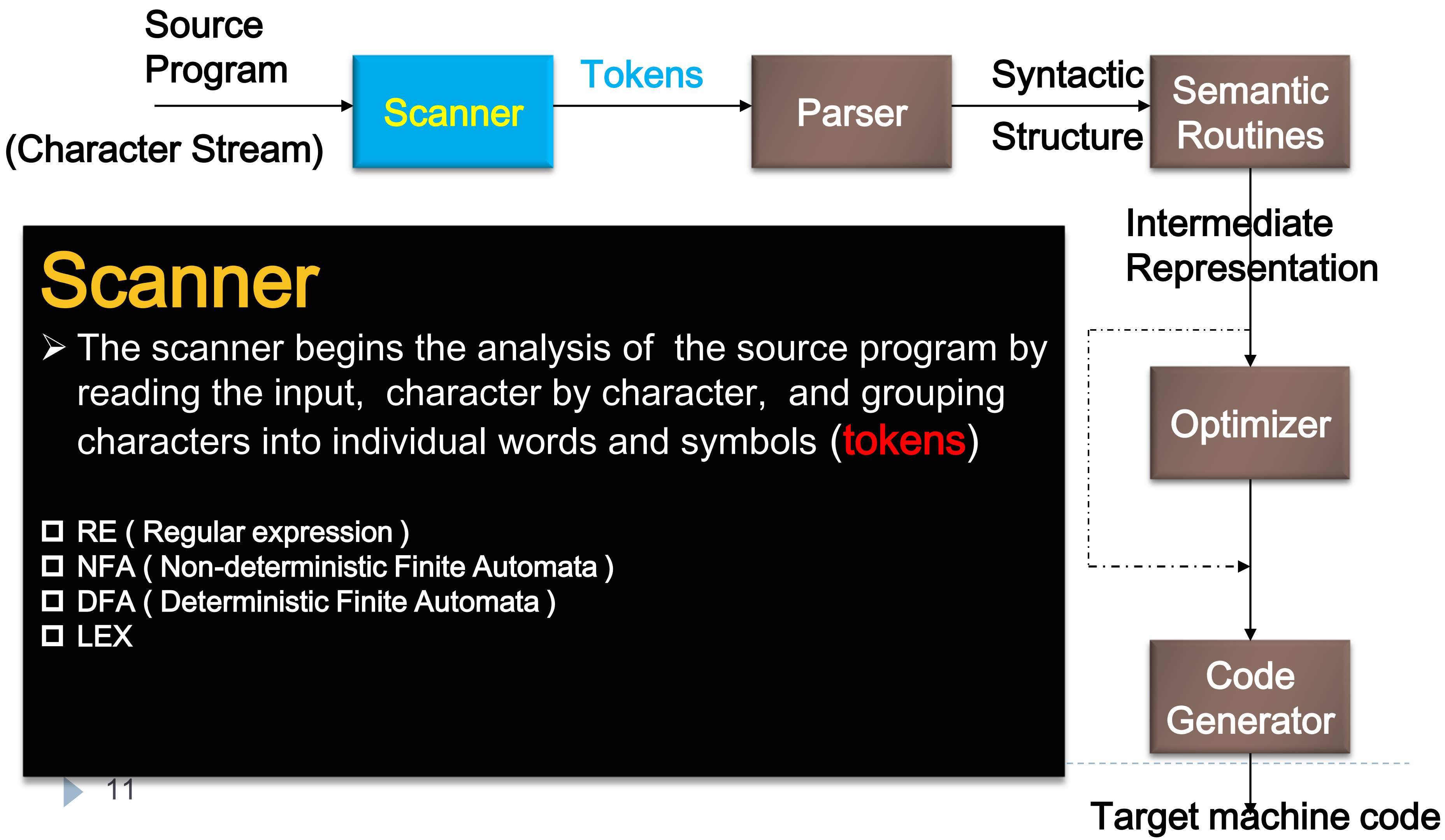


  - *Analysis* of the source program
  - *Synthesis* of a machine-language program

The Structure of a Compiler (2)



The Structure of a Compiler (3)

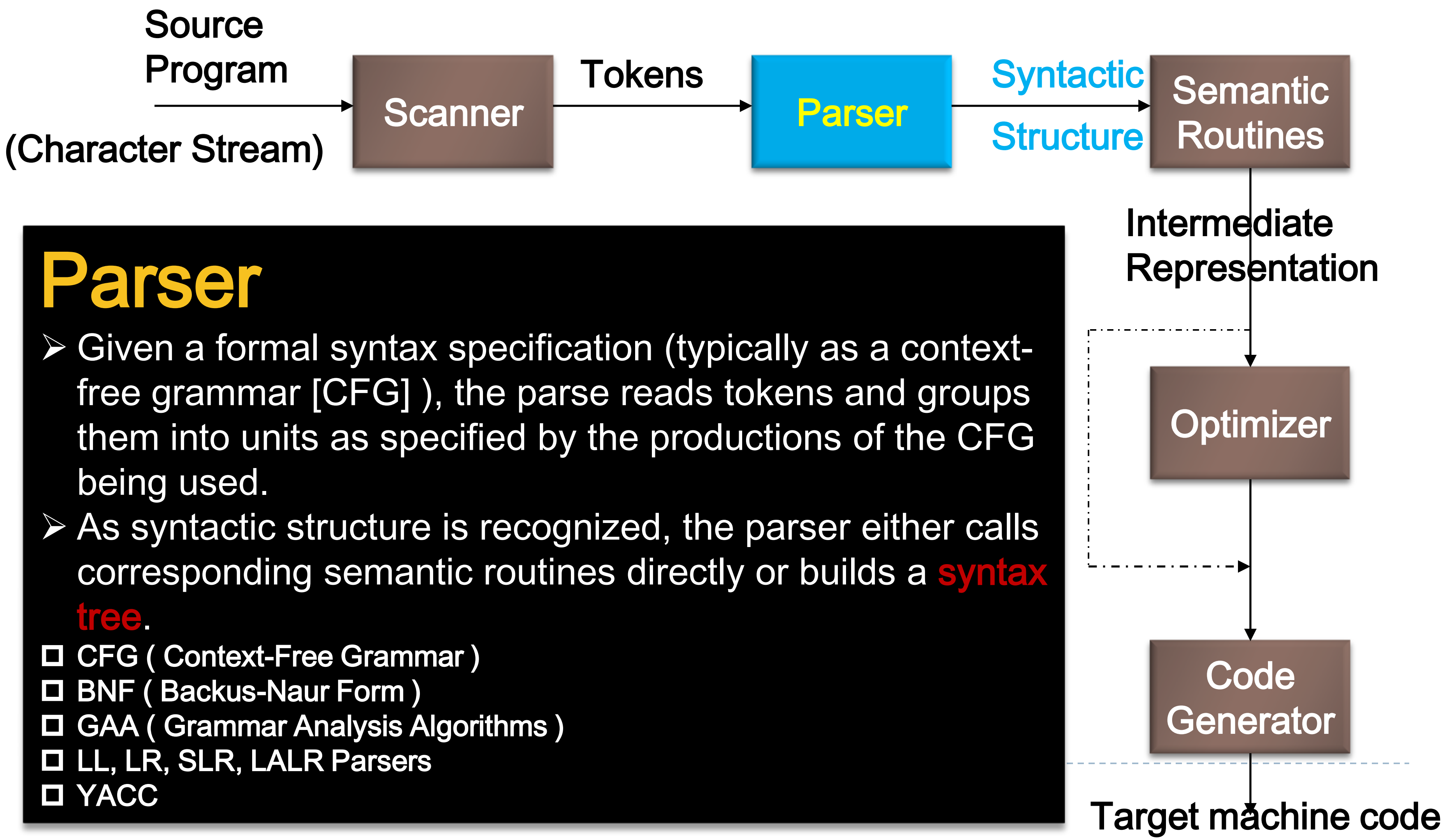


Scanner

- The scanner begins the analysis of the source program by reading the input, character by character, and grouping characters into individual words and symbols (tokens)

- RE (Regular expression)
- NFA (Non-deterministic Finite Automata)
- DFA (Deterministic Finite Automata)
- LEX

The Structure of a Compiler (4)

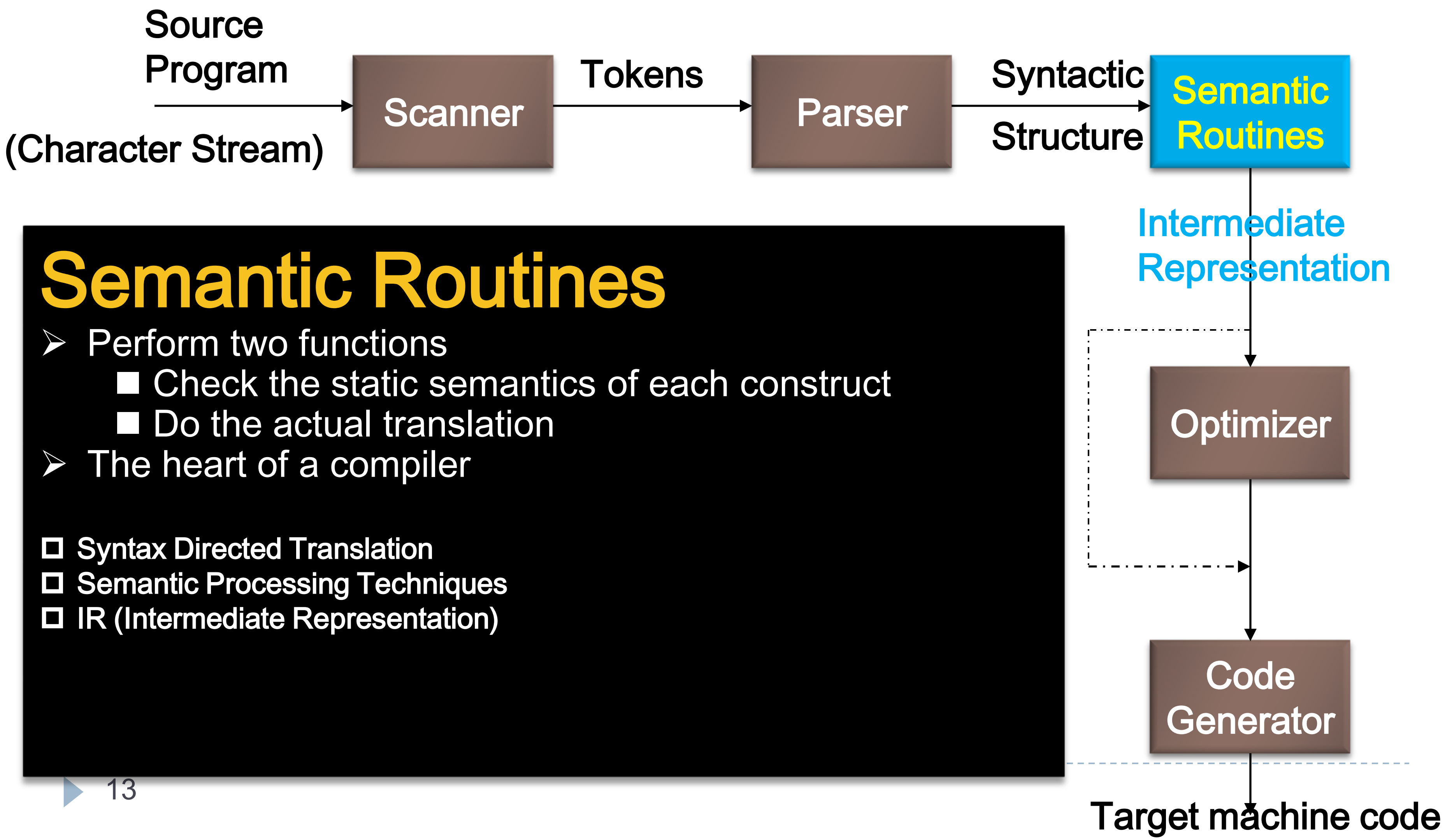


Parser

- Given a formal syntax specification (typically as a context-free grammar [CFG]), the parse reads tokens and groups them into units as specified by the productions of the CFG being used.
- As syntactic structure is recognized, the parser either calls corresponding semantic routines directly or builds a syntax tree.

- CFG (Context-Free Grammar)
- BNF (Backus-Naur Form)
- GAA (Grammar Analysis Algorithms)
- LL, LR, SLR, LALR Parsers
- YACC

The Structure of a Compiler (5)

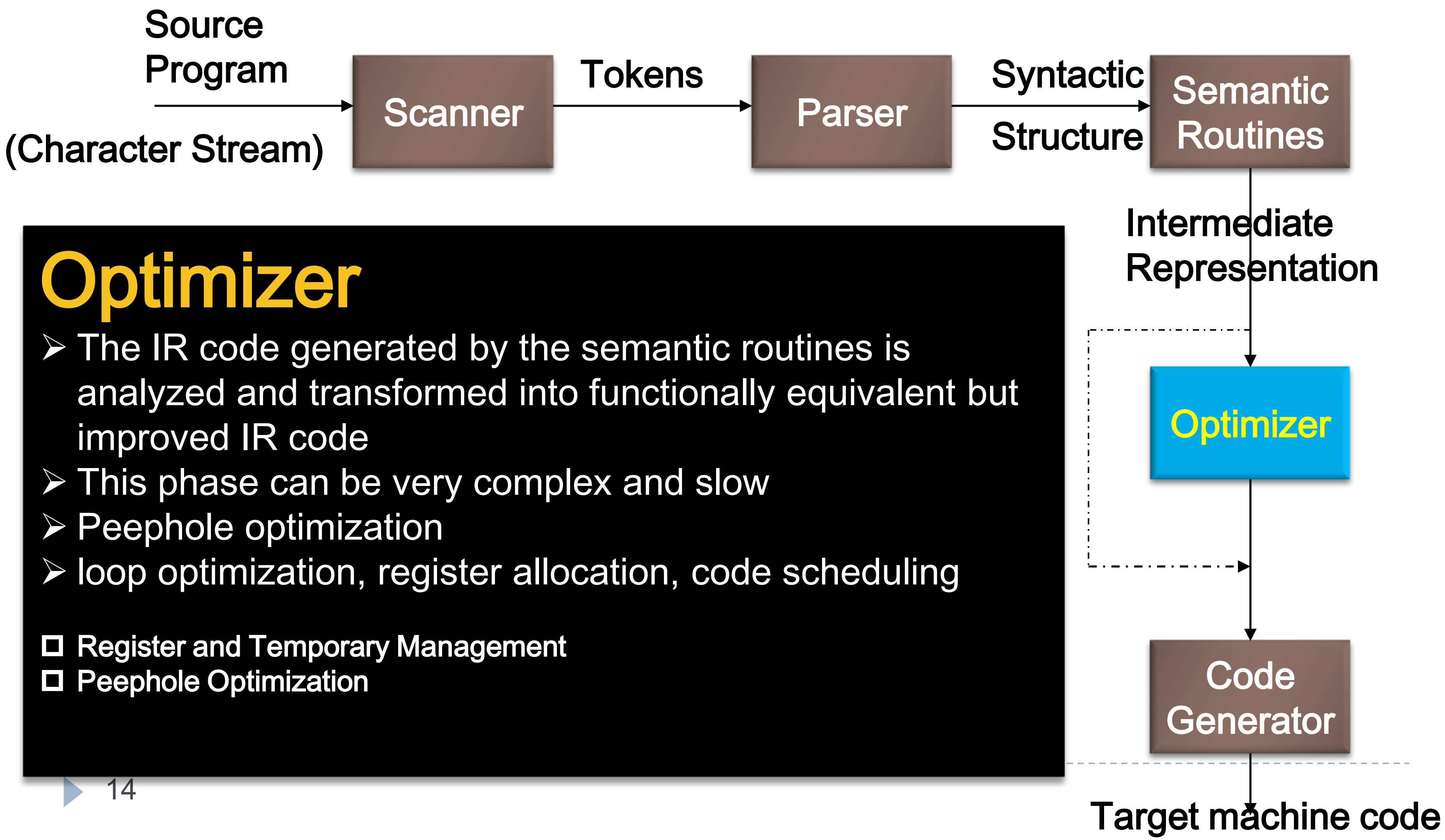


Semantic Routines

- Perform two functions
  - Check the static semantics of each construct
  - Do the actual translation
- The heart of a compiler

- Syntax Directed Translation
- Semantic Processing Techniques
- IR (Intermediate Representation)

The Structure of a Compiler (6)

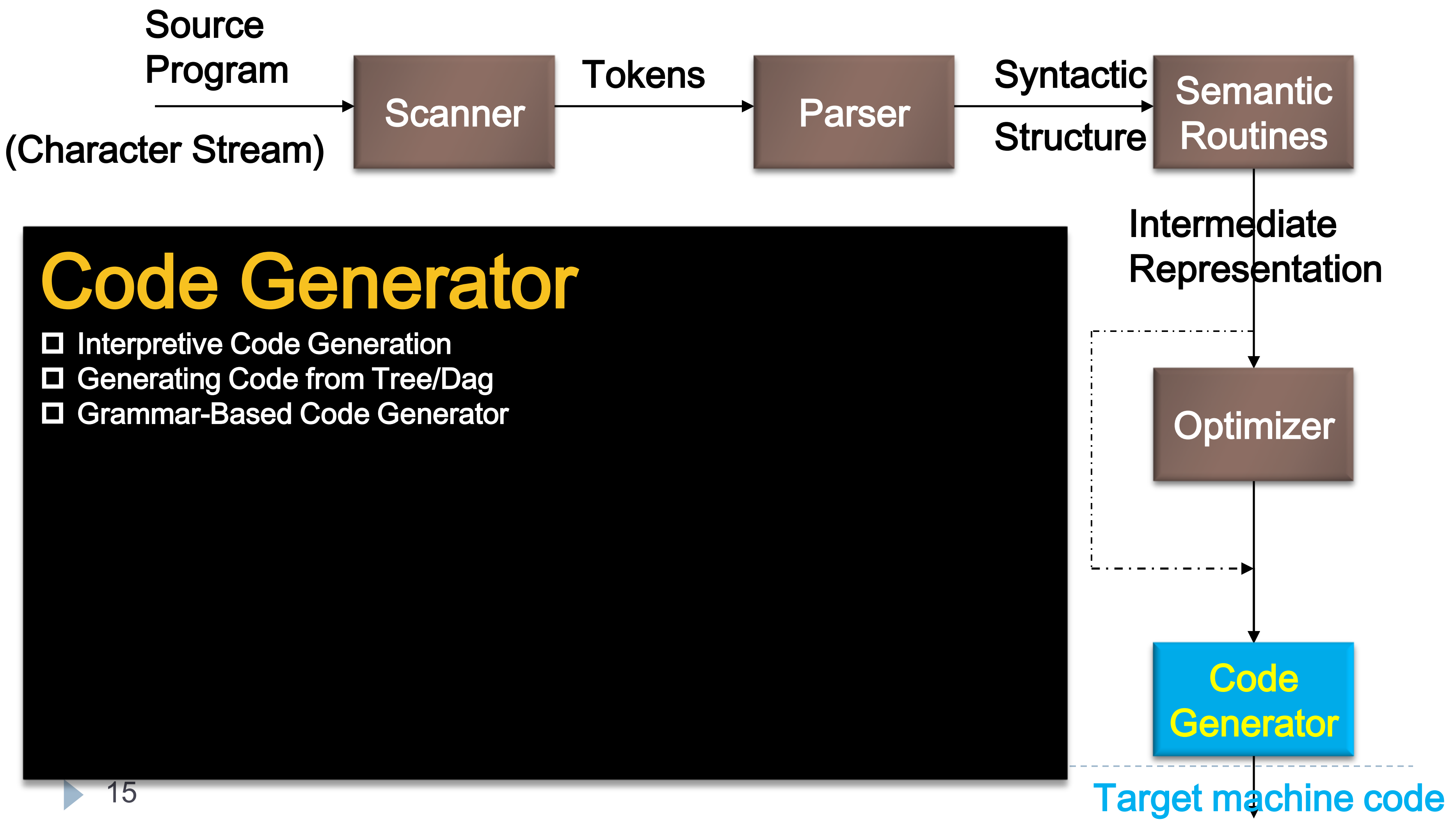


Optimizer

- The IR code generated by the semantic routines is analyzed and transformed into functionally equivalent but improved IR code
- This phase can be very complex and slow
- Peephole optimization
- loop optimization, register allocation, code scheduling

- Register and Temporary Management
- Peephole Optimization

The Structure of a Compiler (7)



Code Generator

- Interpretive Code Generation
- Generating Code from Tree/Dag
- Grammar-Based Code Generator

The Structure of a Compiler (8)

SYMBOL TABLE

1	position	...
2	initial	...
3	rate	...
4		

```
position := initial + rate * 60
```

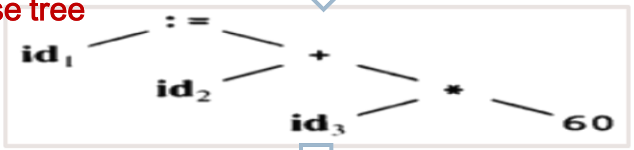
Scanner
[Lexical Analyzer]

Tokens

id_1 := id_2 + id_3 * 60

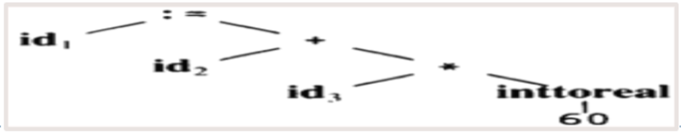
Parser
[Syntax Analyzer]

Parse tree



Semantic Process
[Semantic analyzer]

Abstract Syntax Tree w/ Attributes



Code Generator
[Intermediate Code Generator]

Non-optimized Intermediate Code

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Code Optimizer

Optimized Intermediate Code

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Code Optimizer

Target machine code

```
MOVF  id3, R2
MULF  #60.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

The Structure of a Compiler (9)

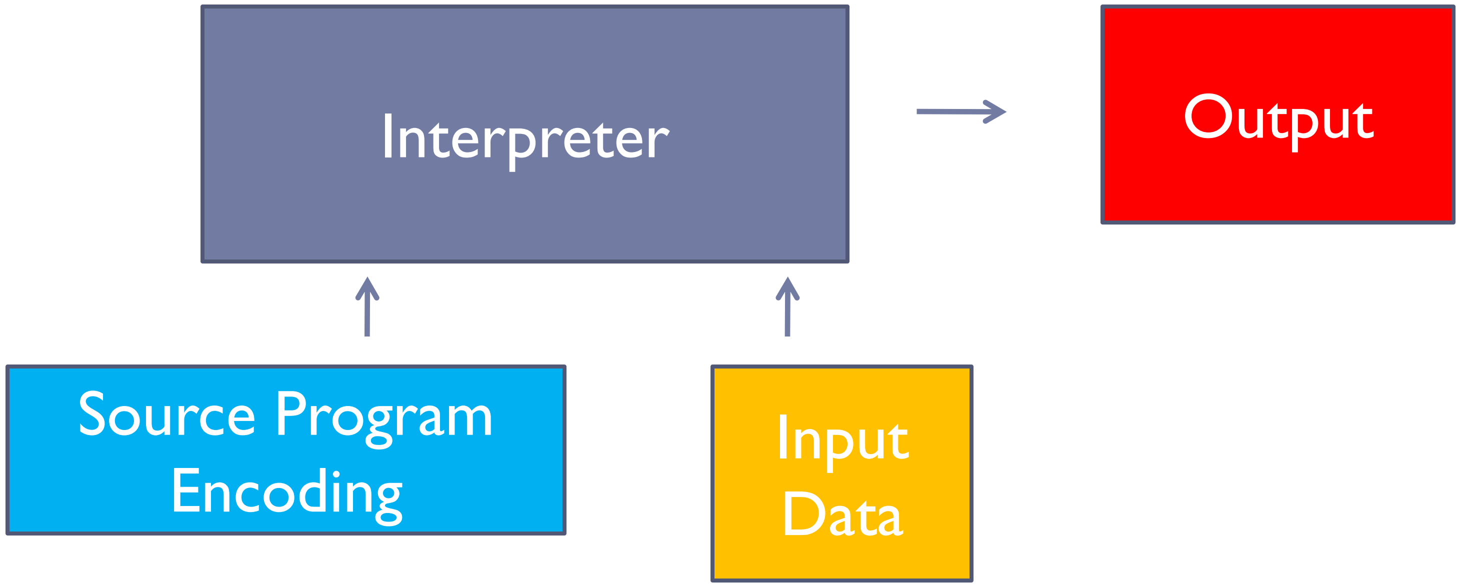
- Compiler writing tools
  - Compiler generators or compiler-compilers
    - E.g. scanner and parser generators
    - Examples : Yacc, Lex

Compiler Design Considerations

- Debugging Compilers
  - Designed to aid in the development and debugging of programs.
- Optimizing Compilers
  - Designed to produce efficient target code
- Retargetable Compilers
  - A compiler whose target architecture can be changed without its machine-independent components having to be rewritten.

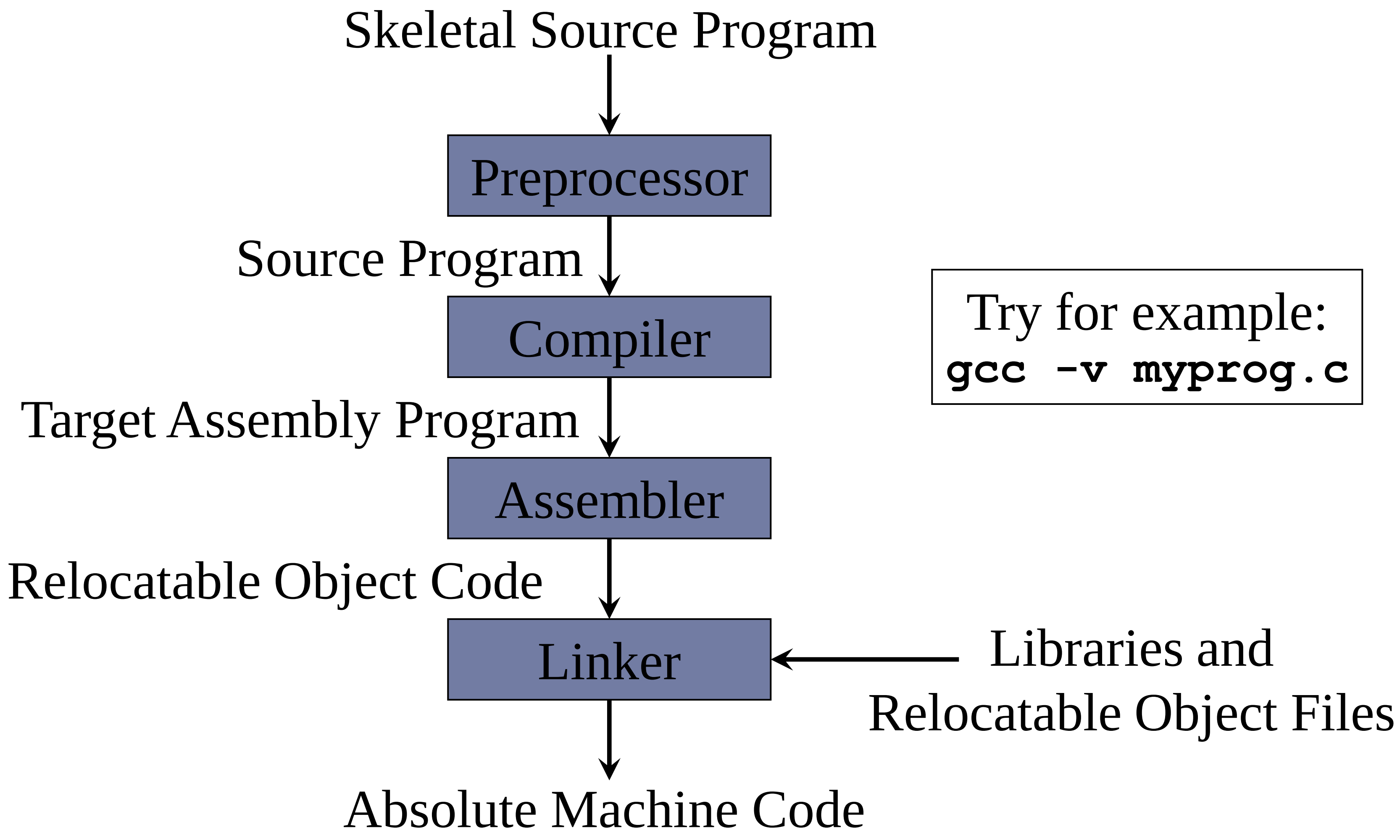
Interpreters (1)

- Performing the operations implied by the source program



- Machine-independent
- Significant overhead

Preprocessors, Compilers, Assemblers, and Linkers



Try for example:

```
gcc -v myprog.c
```

The Grouping of Phases

- Compiler *front* and *back ends*:
  - Front end: *analysis* (*machine independent*)
  - Back end: *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