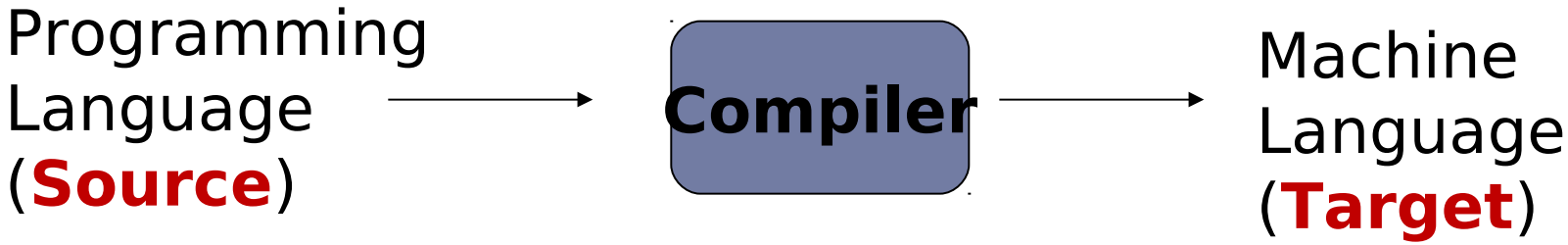
# Compiler

CSEJU

# 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)

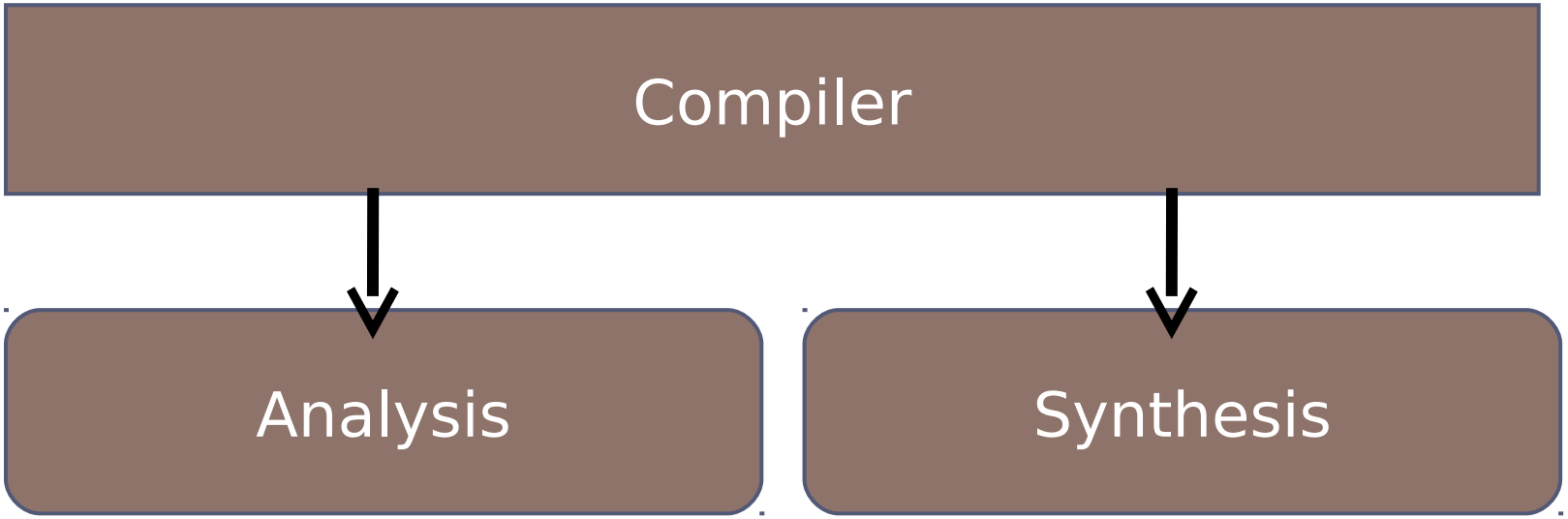
- Compilers may generate three types of code:
  - Pure Machine Code
    - Machine instruction set without assuming the existence of any operating system or library.
    - Mostly being OS or embedded applications.
  - Augmented Machine Code
    - Code with OS routines and runtime support routines.
    - 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

# What Do Compilers Do (3)

- Another way that compilers differ from one another is in the format of the target machine code they generate:
  - Assembly or other source format
  - Relocatable binary
    - Relative address
    - A linkage step is required
  - Absolute binary
    - Absolute address
    - Can be executed directly

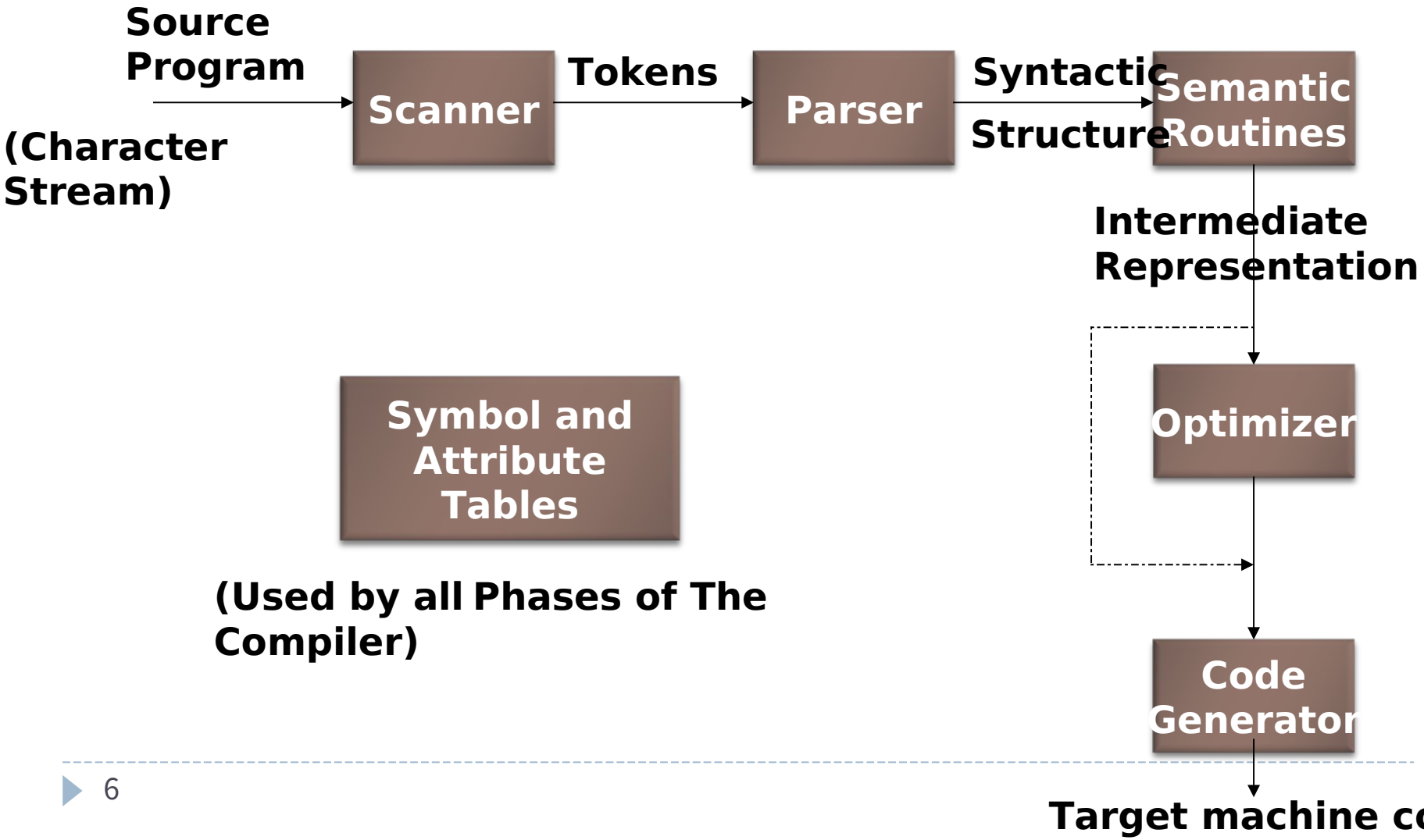
# The Structure of a Compiler (1)

- Any compiler must perform two major tasks

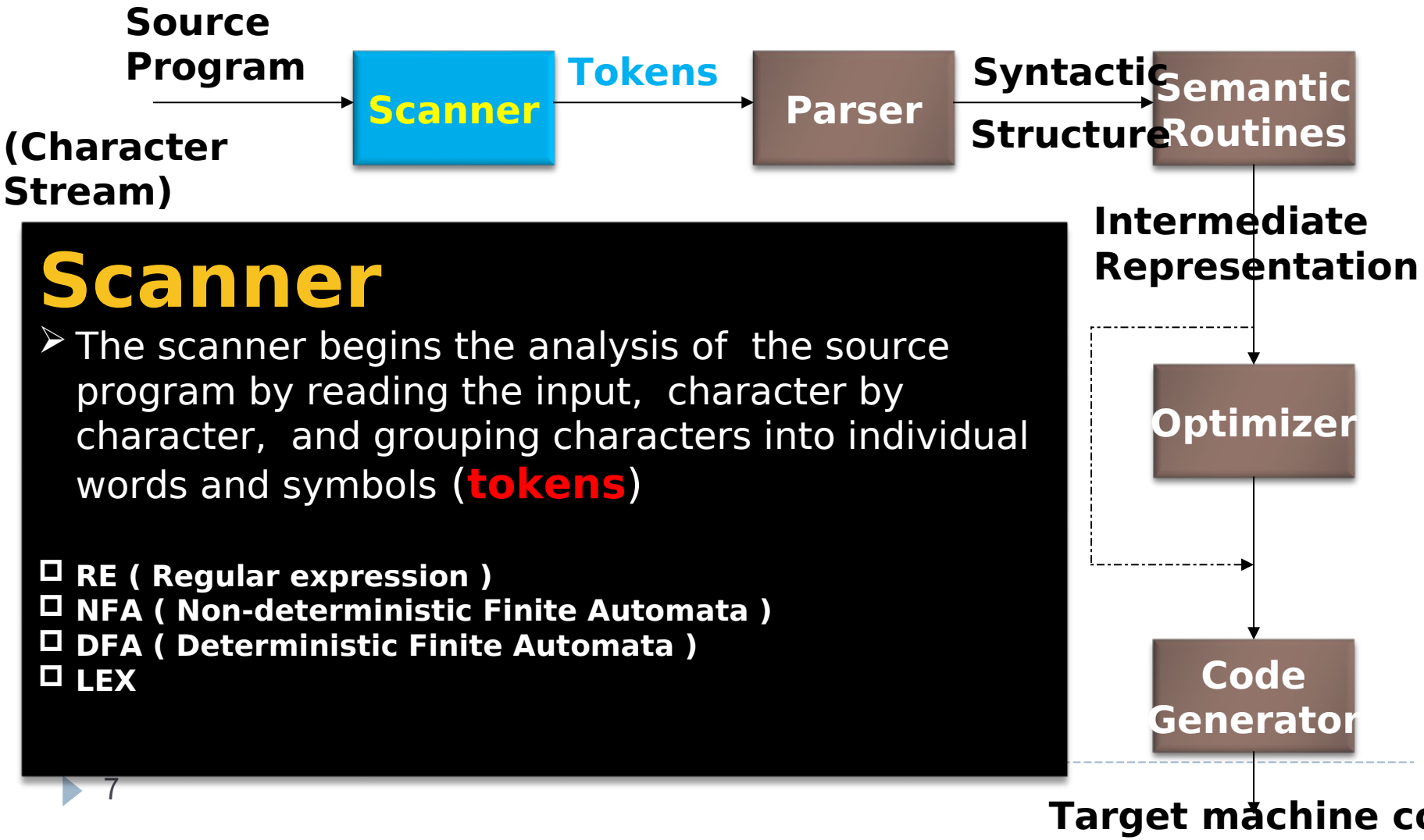



  - ***Analysis*** of the source program
  - ***Synthesis*** of a machine-language program

# The Structure of a Compiler (2)



# The Structure of a Compiler (3)

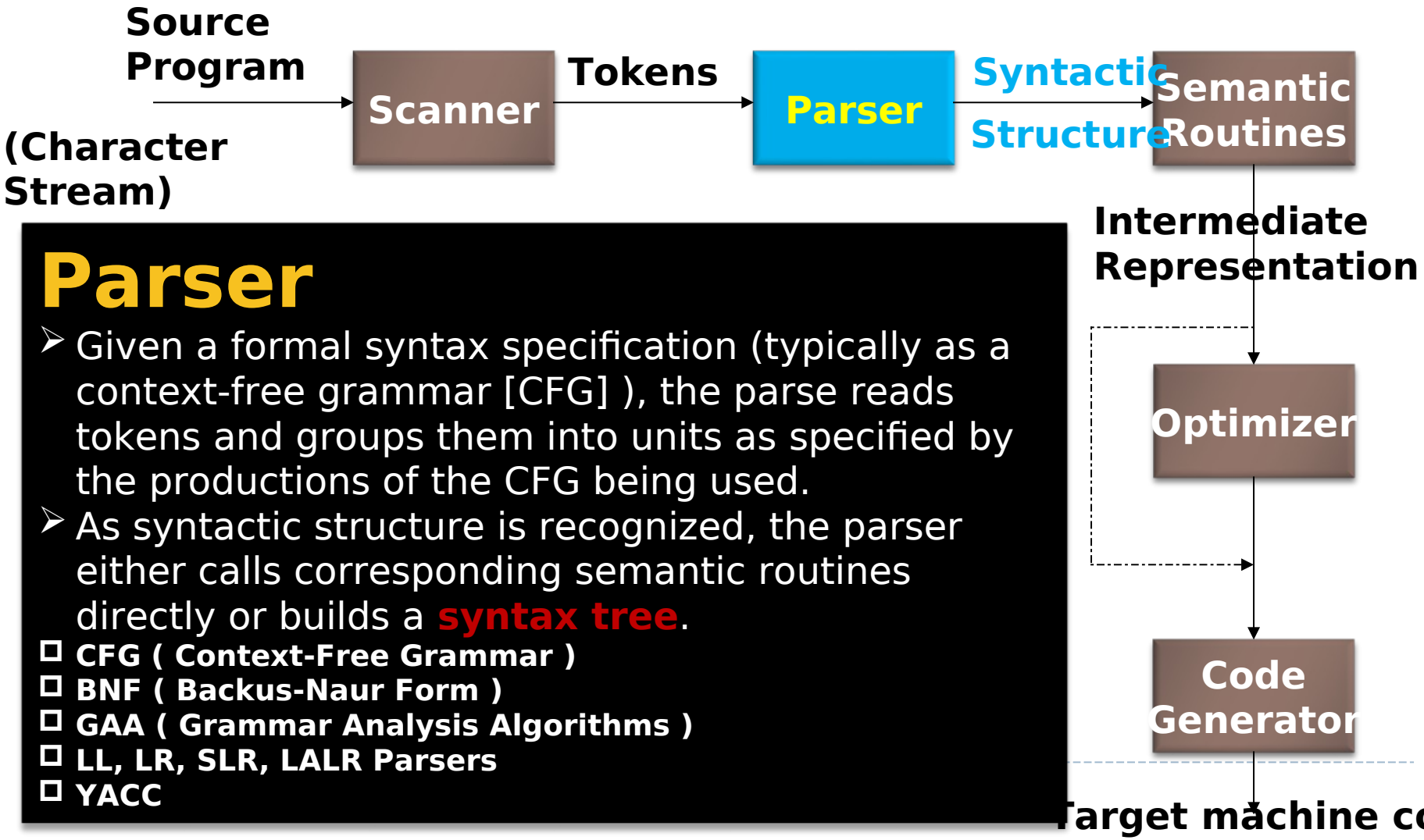



## Scanner

- 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)

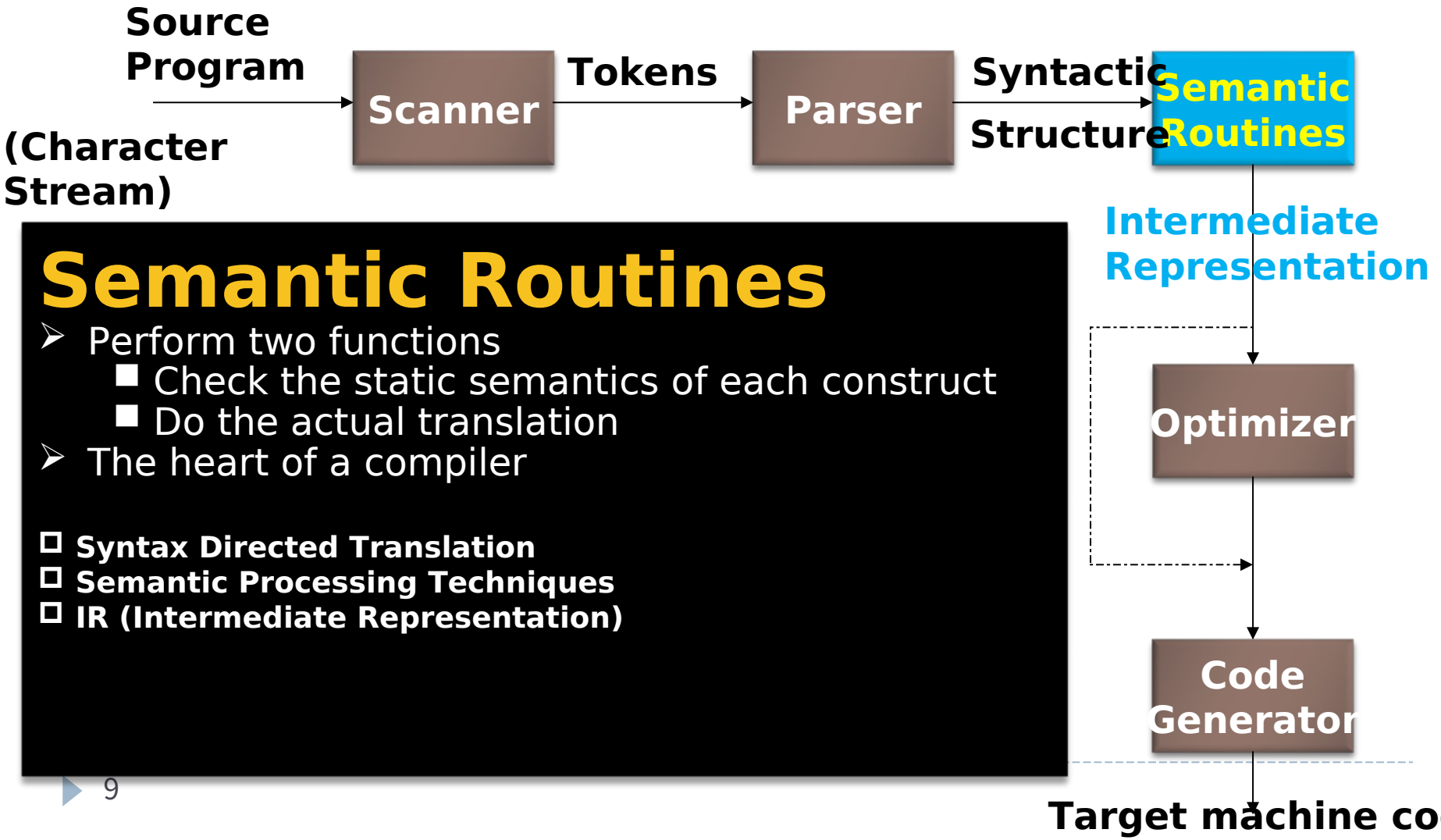



## Parser

- 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)

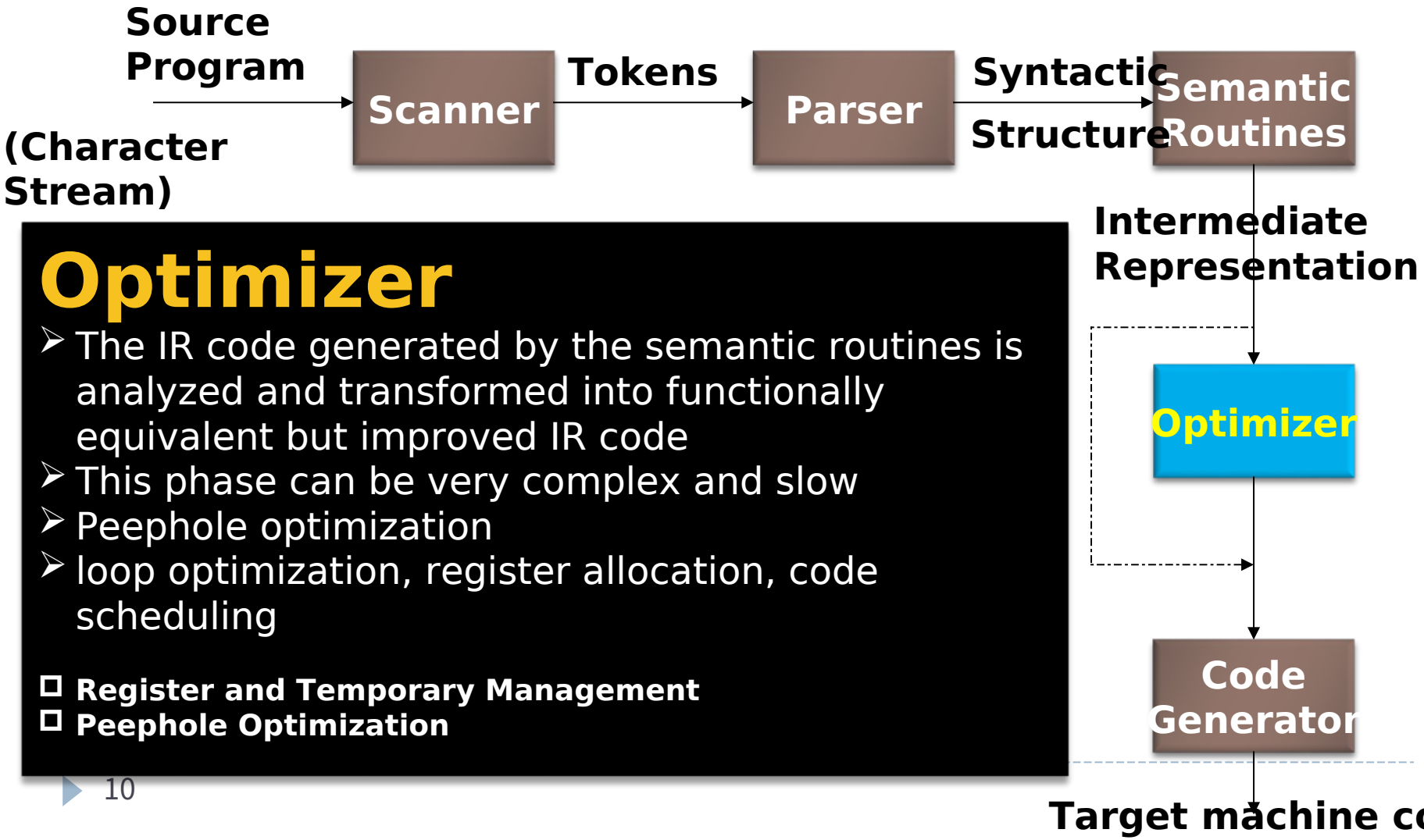



## Semantic Routines

- 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)

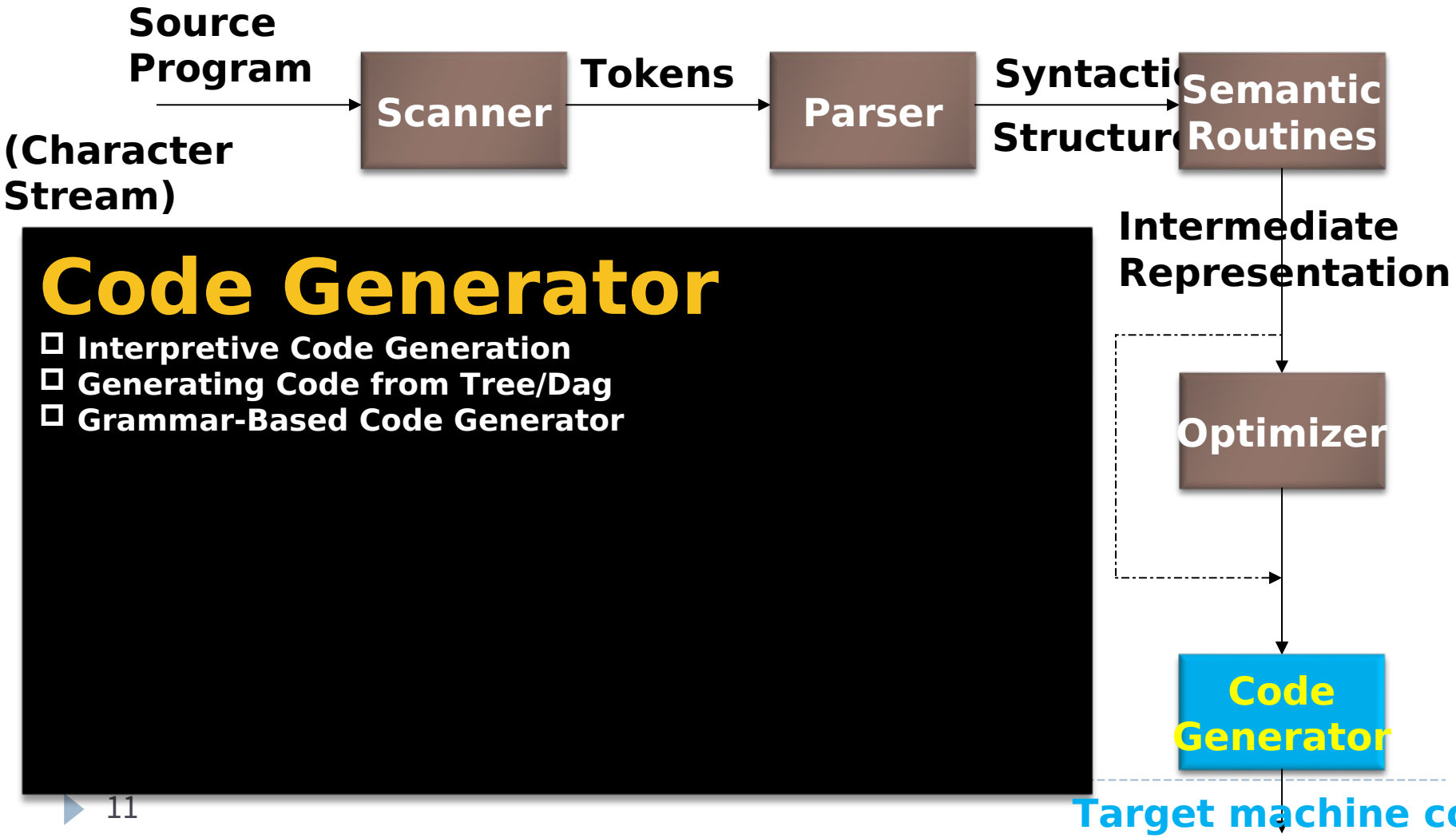



## Optimizer

- 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**

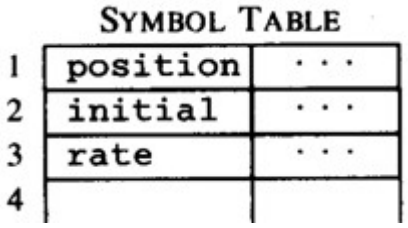
# The Structure of a Compiler (7)




## Code Generator

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

# The Structure of a Compiler (8)




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

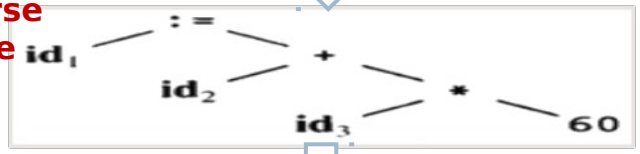
**Scanner**
**[Lexical Analyzer]**

**Tokens**

$id_1 := id_2 + id_3 * 60$

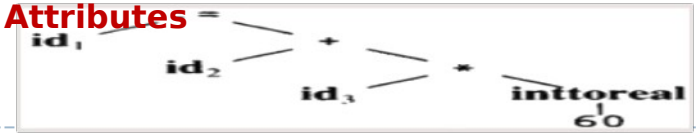
**Parser**
**[Syntax Analyzer]**

**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 Co**

```
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

# The Syntax and Semantics of Programming Language (1)

- A programming language must include the specification of syntax (structure) and semantics (meaning).
- Syntax typically means the context-free syntax because of the almost universal use of context-free-grammar (CFGs)
- Ex.
  - a = b + c is syntactically legal
  - b + c = a is illegal

# The Syntax and Semantics of Programming Language (2)

- The semantics of a programming language are commonly divided into two classes:
  - Static semantics
    - Semantics rules that can be checked at compiled time.
    - Ex.  The type and number of a function's arguments
  - Runtime semantics
    - Semantics rules that can be checked only at run time

# Compiler Design and Programming Language Design (1)

- An interesting aspect is how programming language design and compiler design influence one another.
- Programming languages that are easy to compile have many advantages

# Compiler Design and Programming Language Design(2)

- Languages such as Snobol and APL are usually considered noncompilable

- What attributes must be found in a programming language to allow compilation?
  - Can the scope and binding of each identifier reference be determined before execution begins?
  - Can the type of object be determined before execution begins?
  - Can existing program text be changed or added to during execution?

# Computer Architecture and Compiler Design

- Compilers should exploit the hardware-specific feature and computing capability to optimize code.
- The problems encountered in modern computing platforms:
  - Instruction sets for some popular architectures are highly nonuniform.
  - High-level programming language operations are not always easy to support.
    - Ex. exceptions, threads, dynamic heap access …
  - Exploiting architectural features such as cache, distributed processors and memory
  - Effective use of a large number of processors

# 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.