CSC 318 – COMPILER CONSTRUCTION I (2-UNITS)

Week Two:

Structure of Compilers and Phases of Compilation Process

The CSC318 Team [2020|2021]
femi.odusote@covenantuniversity.edu.ng

Recommended Texts for this Course

1. Watson, Des (2017). **A Practical Approach to Compiler Construction**, Springer. [MainText]

2. Campbell, Bill; Iyer, Swami; and Akbal-Delibas, Bahar (2013). **Introduction to Compiler Construction in a Java World**, Taylor & Francis Group.

3. Seidl, Helmut; Wilhelm, Reinhard; and Hack, Sebastian (2012). **Compiler Design: Analysis and Transformation**, Springer

4. Grune, Dick; Reeuwijk, Kees van; Bal, Henri E; Jacobs, Ceriel J.H; and Langendoen, Koen (2012). **Modern Compiler Design**, Second Edition, Springer.

5. Reis, Anthony Dos (2011). **Compiler Construction Using Java, JavaCC, and Yacc**, Wiley

Lecture Topics

1. What is a Compiler?
2. Structure of a Compiler
3. Phases/Steps of Compilation
   - *Lexical Analysis (Scanning)*
   - *Syntax Analysis (Parsing)*
   - *Semantic Analysis*
   - *Intermediate Code Generation (ICG)*
   - *Code Optimization*
   - *Code Generation (Target Machine Code)*
4. Issues Driving Compiler Design.
   - *Correctness;*
   - *Speed (Runtime & Compile Time);*
   - *Space*
   - *Feedbacks of Users*
   - *Debugging*

1. What is a Compiler?

- A **Compiler** is a computer program that translates a program in a source language into an equivalent program in a target (machine) language.
  - Translation from programs written in high- level languages (HLL) to low- level object code and machine code (LLL).

- A **source program/code** is a program/code written in the source language, which is usually a high-level language.

- A **target program/code** is a program/code written in the target language, which often is a machine language or an intermediate code.

2. Structure of a Compiler

- At an abstract level, a typical **Compiler** is represented as follows;

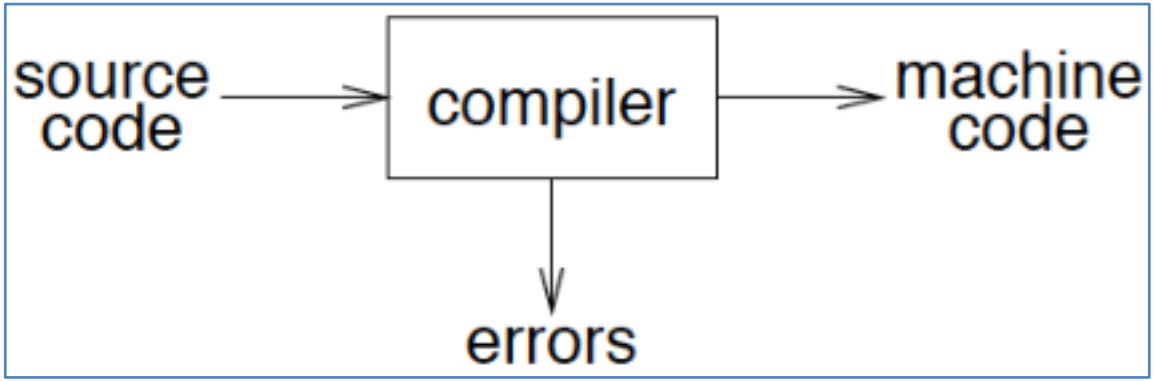
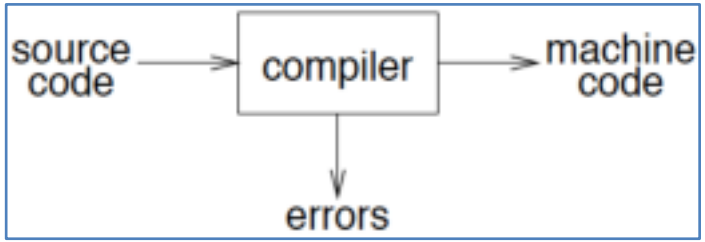


Figure 1: An Abstract Structure of a Compiler

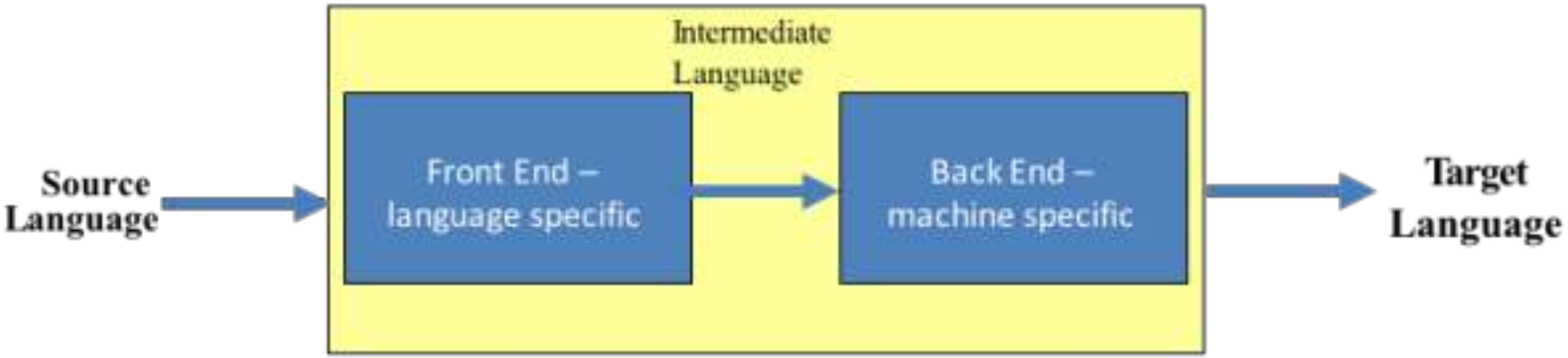
2. Structure of a Compiler (1)



- The structure of a compiler, at an abstract level:
  - By implication, a compiler:
    1. recognizes legal (and illegal) programs,
    2. generates correct code,
    3. manages storage of all variables and code,
    4. agreement on format for object (or assembly) code,
    5. Generates Error Messages for the programmer's attention.

2. Structure of a Compiler (2)

In more detail:



- Separation of Concerns
- Retargeting

Figure 2: The Structure of a Compiler

2. Structure of a Compiler (3)

- The Figure 2 above shows the structure of a two-pass compiler.
  - FRONT End
  - BACK End

- The **Front End** maps legal source code into an intermediate representation (IR).

- The **Back End** maps IR into target machine code. An immediate advantage of this scheme is that it admits multiple front ends and multiple passes.

2. Structure of a Compiler (4)

- A **Compiler** is usually designed as a phased tool that executes the translation process in phases. Basically the compiler is structured into two phases;

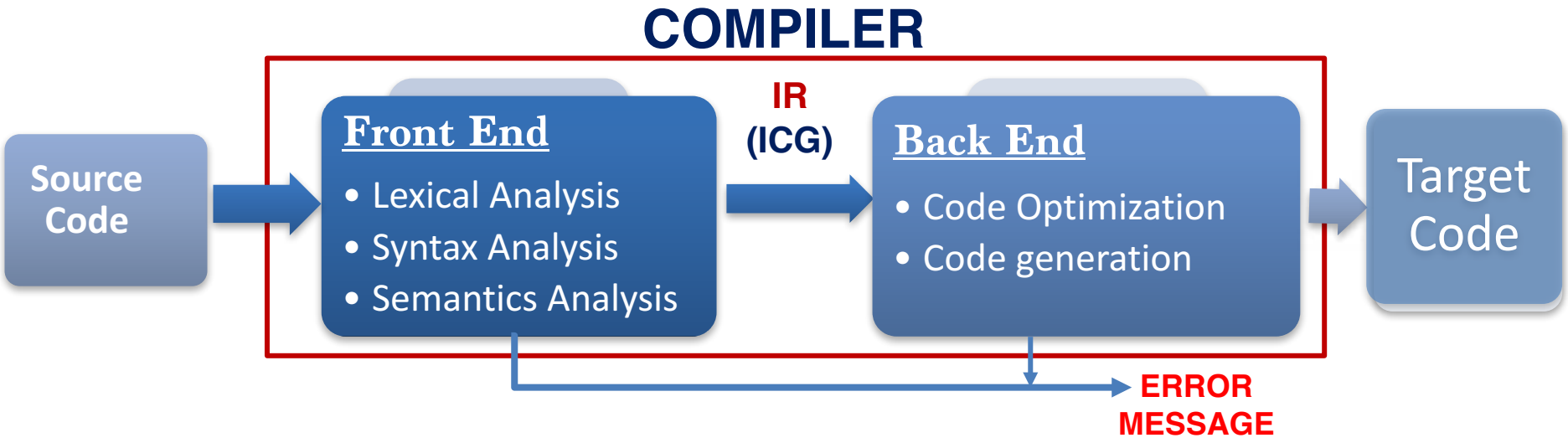


Figure 3: The Structure/Phases of a Compiler

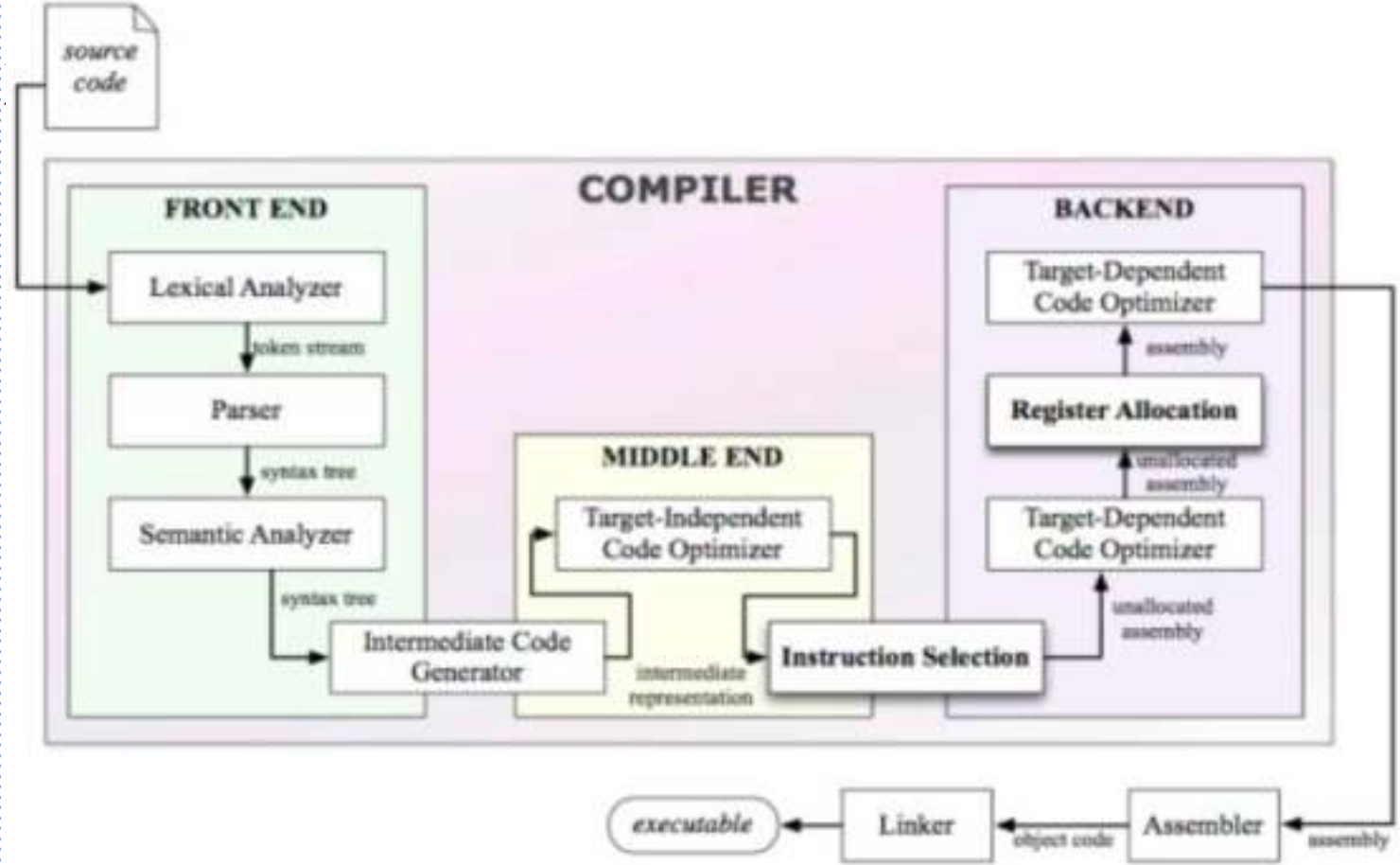


Figure 4: The Internal Structure of a Compiler

2. Structure of a Compiler (4)

- The implications of a two-phases compiler are as follows:
  - Intermediate Representation (IR)
  - Front End maps legal source code into IR
  - Back End maps IR onto Target Machine Code
  - Simplifies retargeting
  - Allows multiple Front Ends
  - multiple passes => better code

3. Phases of Compilation

- A compiler operates in phases.
  - A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation

- The Phase of a Compiler otherwise referred to as 'Phases/Steps/Process of Compilation'.

- There are two phases of compilation:
  1. Analysis (Machine Independent/Language Dependent)
  2. Synthesis(Machine Dependent/Language independent)

3. Phases of Compilation

- Compilation process is partitioned into no-of-sub processes called **'phases'**.

- The Phases/Steps/Process of Compilation' are as follows:
  - **Lexical Analysis** (Scanning) **–** receives 'Streams of Characters' to produce 'Tokens'
  - **Syntax Analysis** (Parsing) **–** receives 'Tokens' to produce 'Parse' (Syntax Tree)
  - **Semantic Analysis –** annotates the 'Syntax Tree' to produce 'Abstract Syntax Tree'
  - **Intermediate Representation –** generates 'Intermediate Code' & Optimizer
  - **Code Optimization**
  - **Code Generation –** Target 'Machine' Code

Process of Compilation

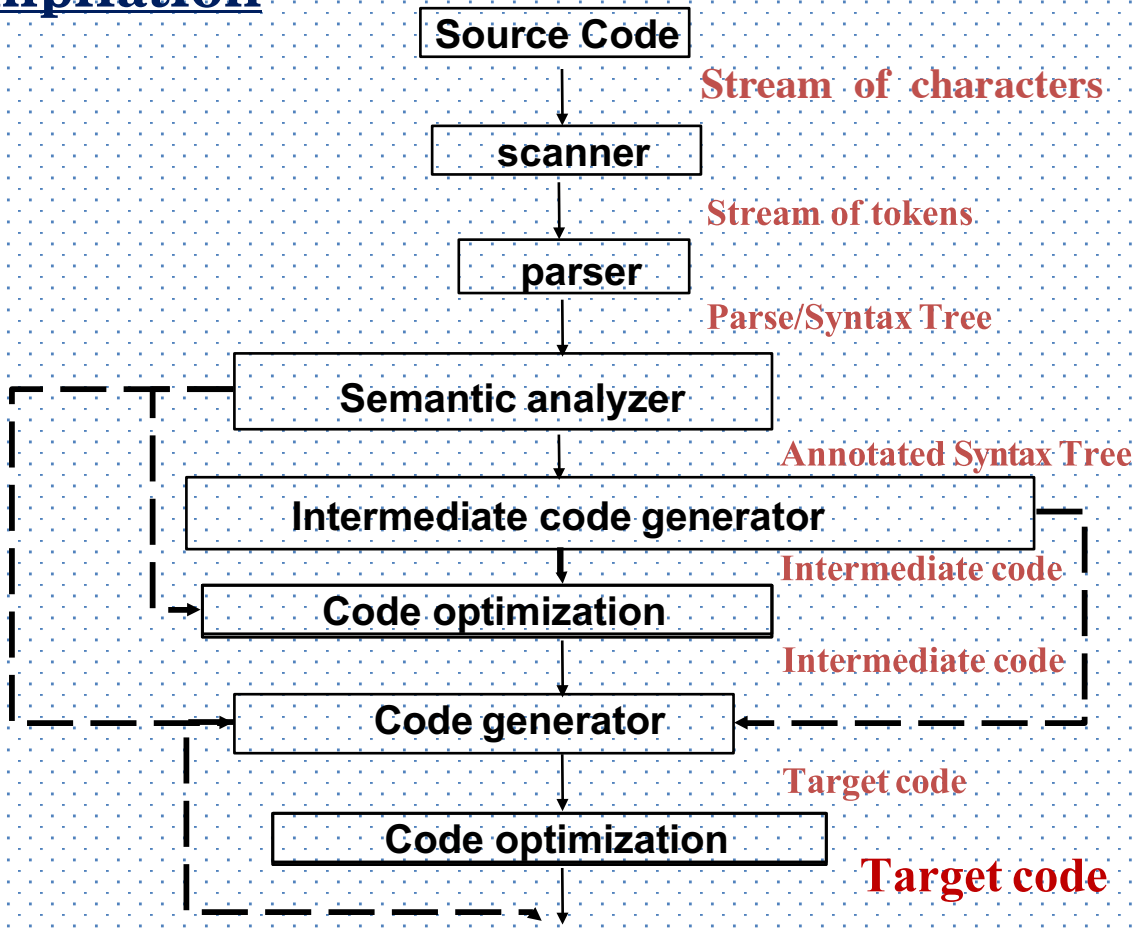


Figure 5: The Process of Compilation

The rest of the process

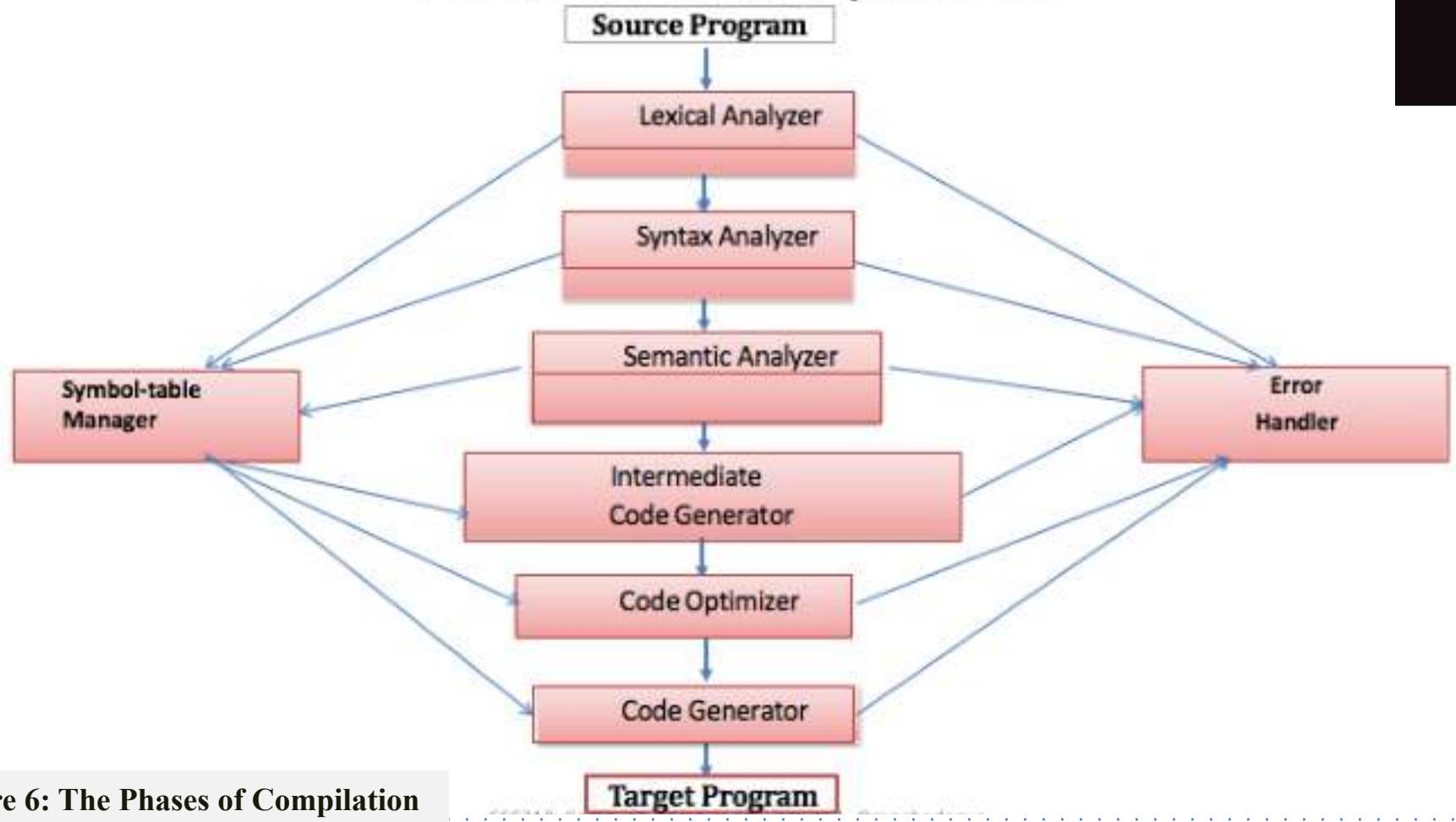


Figure 6: The Phases of Compilation

❖ ***Lexical Analysis:***

- LA or Scanner reads the source program, one character after the other, at a time, carving the program into a sequence of atomic units called '**Tokens**'.

❖ ***Syntax Analysis:***

- In this (2nd) phase/step, program expressions, statements, declarations, etc… are identified by using the results of LA. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

❖ ***Semantic Analysis:***

- This (3rd) phase, (but the last of the FRONT End phase) has to do with the task of ensuring that program expressions, statements, declarations, etc. are ***semantically*** correct, i.e. that their meanings are clear and consistent with the way in which the control structures and data types are suppose to be used.

❖ *Intermediate Code generation:*

- An intermediate representation of the final machine language code is produced.
- This phase bridges the analysis and synthesis phases of translation.

❖ *Code Optimization:*

- This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

❖ *Code Generation:*

- The last phase of translation is code generation. A number of optimizations to reduce the length of machine language program are carried out during this phase.
- The output of the code generator is the '***Machine Language Program***' of the specified computer.

❖ ***Symbol Table Manager (STM):***

- This STM keeps the names of key entities, identifiers, names, keywords, etc. used by the program and records essential information about each. The data structure used to record this information called a '***Symbol Table***'.
- Identifiers are **names** of variables, constants, functions, datatypes, etc.
- ST Stores information associated with these identifiers;
  - Information associated with different types of identifiers can be different;
    - Information associated with variables are name,type, address,size (for array),etc.
    - Information associated with functions are name, type of return value, parameters, address, etc.

❖ ***Symbol Table Manager (STM):***

- Accessed in every phase of compilers
  - The scanner, parser, and semantic analyzer put names of identifiers in symbol table.
  - The semantic analyzer stores more information (e.g. datatypes, labels) in the table.
  - The intermediate code generator, code optimizer and code generator use information in symbol table to generate appropriate code.
- Hash Table is mostly used for efficiency.

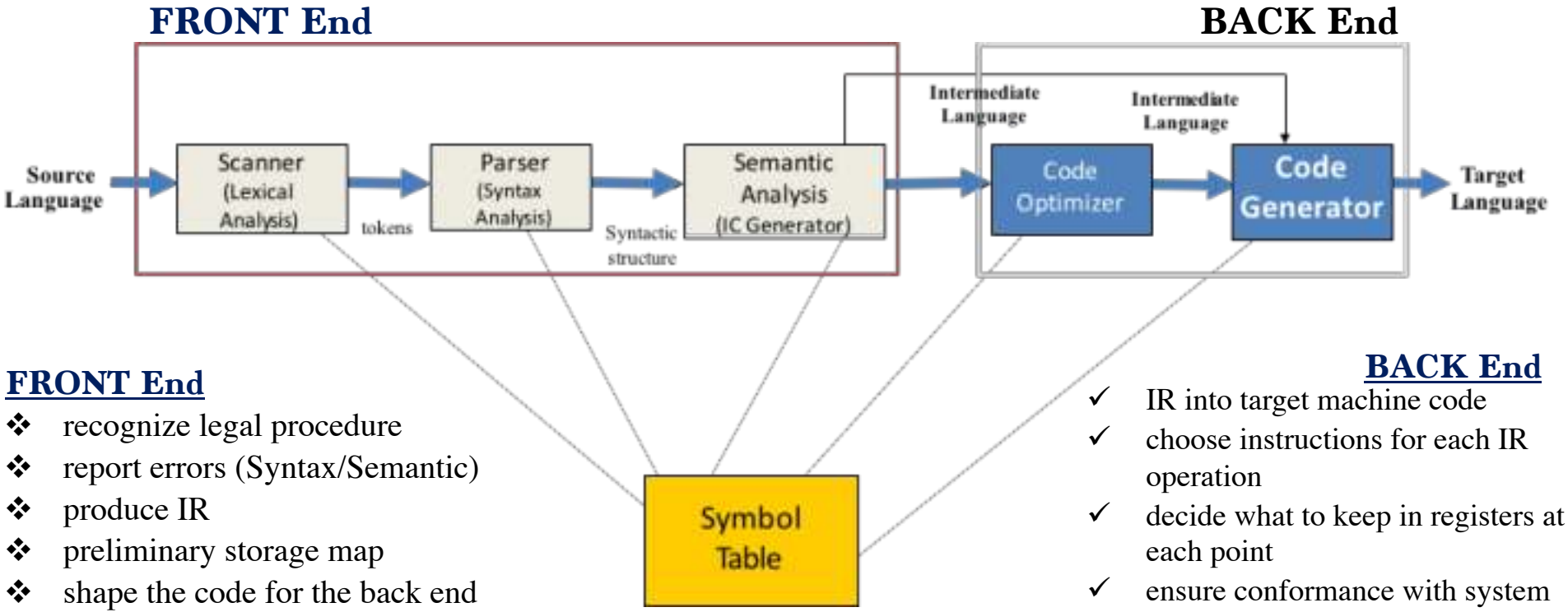
❖ ***Literals Table:***

- Store constants and strings used in program
  - reduce the memory size by reusing constants and strings
- Can be combined with symbol table

❖ *Error Handling*

- Error can be found in every phase of compilation.
  - Errors found during compilation are called *static* (or *compile-time*) *errors*.
  - Errors found during execution are called *dynamic* (or *run- time*) *errors*
- Compilers need to detect, report, and recover from error found in source programs
- Error handlers are different in different phases of compiler.

❖ *The Structure – Phases of a Compiler*



FRONT End

- ❖ recognize legal procedure
- ❖ report errors (Syntax/Semantic)
- ❖ produce IR
- ❖ preliminary storage map
- ❖ shape the code for the back end

BACK End

- ✓ IR into target machine code
- ✓ choose instructions for each IR operation
- ✓ decide what to keep in registers at each point
- ✓ ensure conformance with system interfaces

Figure 7: The Structure-Phases of a Compiler

1. Scanning

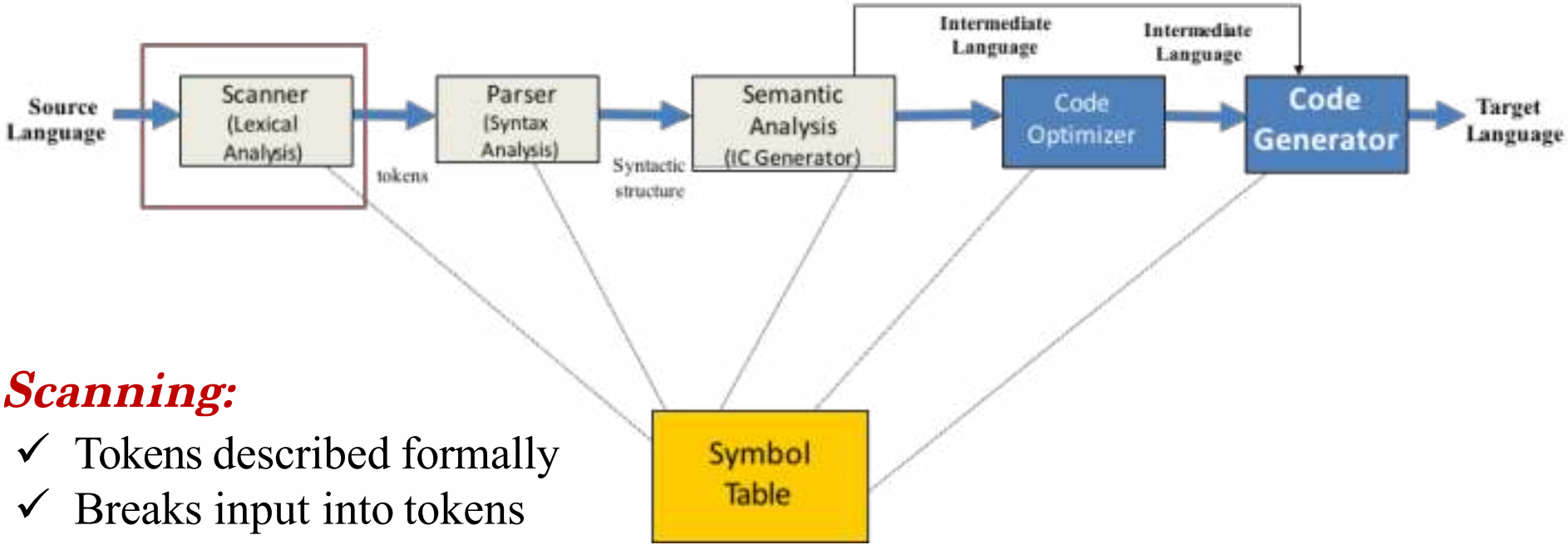
- A Scanner reads a stream of characters and puts them together into some meaningful atomic units (with respect to the source language) called '**Tokens**'.
  - It produces a stream of tokens for the next phase of compiler.

- The scanner takes a program as input and maps the character stream into "words" that are the basic unit of syntax. It produces pairs – a **word** and its '**part of speech**'.
  For example, the input: **`x = x + y;`** is scanned as '***Lexemes***':

  `<id, x> <assign, = > <id, x> <op, +> <id, y> ;`

- We call the pair "`<token type, word>`" a '**Tokens**'.
- Typical tokens are: **number**, **identifier**, +, **-**, /, *, **new**, **while**, **if**, …
- eliminates white space (tabs, blanks, comments).

Lexical Analysis - Scanning



Scanning:

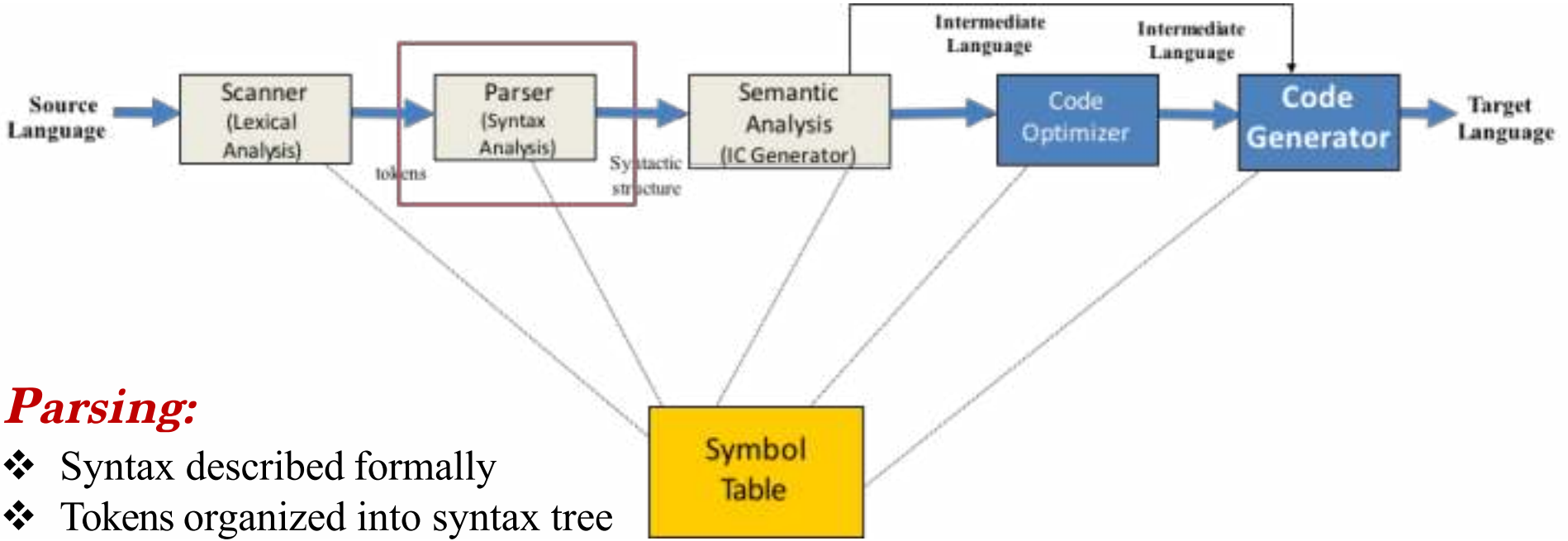
- ✓ Tokens described formally
- ✓ Breaks input into tokens
- ✓ Remove white space

Figure 8: The Lexical Analysis Phase of a Compiler

2. Parsing

- A Parser takes in the stream of tokens, and formally combine them into meaningful syntactic structure.
  - The tokens are combined and organized into a Syntax Tree that describes the structure.
  - It recognizes context-free syntax.
  - The syntax of most programming languages is specified using **Context-Free Grammars (CFG)**.
  - It generates meaningful syntax error messages
  - It attempts to correct the errors
  - It guides context-sensitive ("semantic") analysis for tasks like type checking.
  - The parser builds IR for source program.
- The Parser has two functions.
  - It checks if the tokens from lexical analyzer, occur in pattern that are permitted by the specification for the source language.
  - It also imposes on tokens a tree-like structure that is used by the sub-sequent phases of the compiler.

Syntax Analysis - Parsing



Parsing:

- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking (syntax)

Figure 9: The Syntax Analysis Phase of a Compiler

2. *Parsing - Example*

Example, if a program contains the expression **A+/B** after lexical analysis this expression might appear to the syntax analyzer as the token sequence **id+/id.** On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formulations rule of an expression. Syntax analysis is to make explicit the hierarchical structure of the incoming token stream by **identifying which parts of the token stream should be grouped**.

Example, (A/B*C has two possible interpretations.)
1, divide A by B and then multiply by C or
2, multiply B by C and then use the result to divide A.
each of these two interpretations can be represented in terms of a parse tree.

2. *Parsing*

- A Parser takes in the stream of tokens, and formally combines and organizes them into meaningful syntactic structure i.e. a Syntax Tree that describes the structure.

- For example, the input: **`r = a + b * c / d;`** is scanned as '***Lexemes***':

```
<id, r> <assign, = > <id, a> <op, +> <id, b> <op, *> <id, c> <op, /> <id, d>;
```

  - The Parser combines the lexemes and organizes them into.a syntactic structure i.e. Syntax Tree.

  - The pictorial representation of the Syntaxt Tree alongside the Grammar Rules that guides it are shown in the next slide.

...the input source code: `r = a + b * c / d;` is scanned as '***Lexemes***':

```
<id, r> <assign, = > <id, a> <op, +> <id, b> <op, *> <id, c> <op, /> <id, d>;
```

```
Exp    ::= Exp'+'Exp
         |  Exp'-'Exp
         |  Exp'*'Exp
         |  Exp'/'Exp
         |  ID
Assign ::=  ID'='Exp
```

Syntax Tree

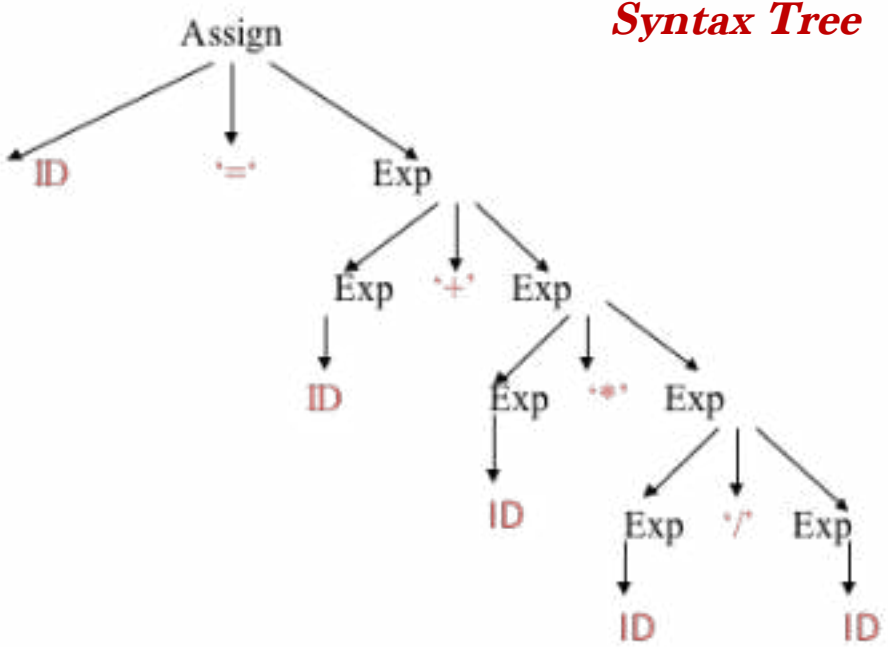
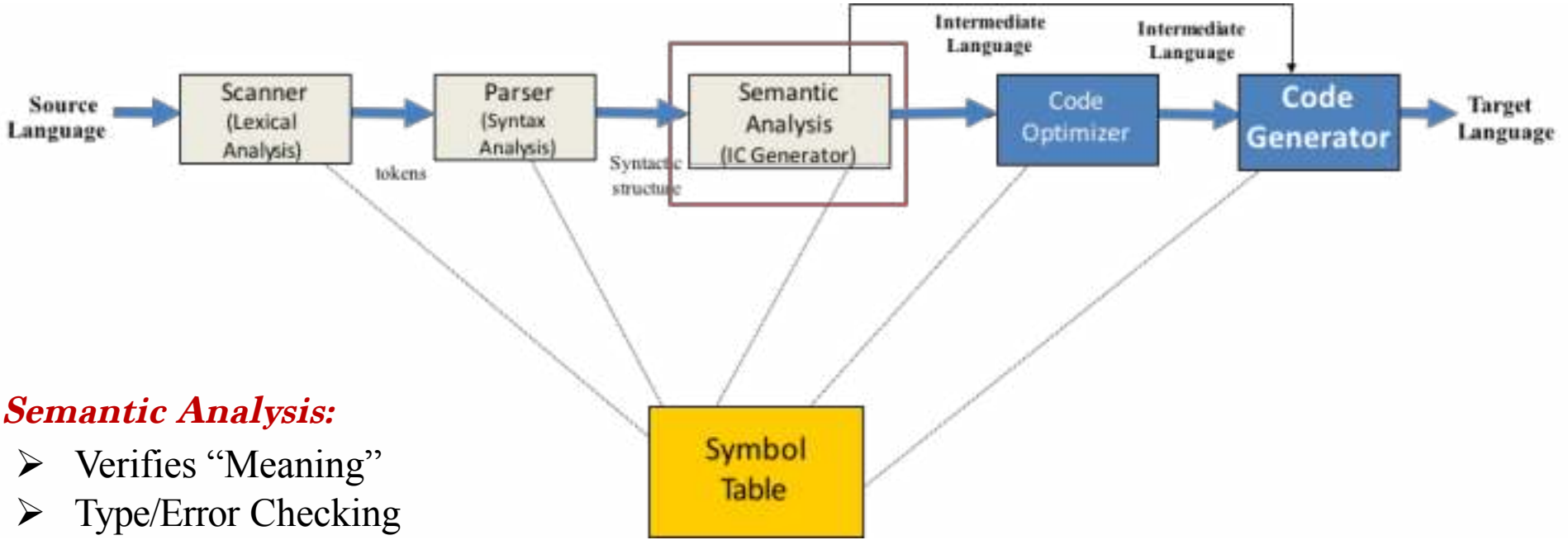


Figure 10: A Typical Syntax Tree

Semantic Analysis



Semantic Analysis:

- Verifies "Meaning"
- Type/Error Checking
- Intermediate Code Generation
  - Abstract Machine

Figure 11: The Semantic Analysis Phase of a Compiler

3. Semantic Analysis

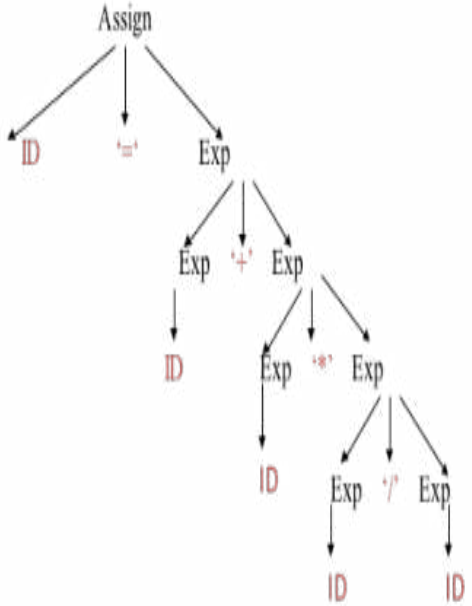
- It gets the syntax tree from the Parser together with information about some syntactic elements. It determines if the semantics or meaning of the program is correct.
- This part deals with *static semantic*.
  - semantic of programs that can be checked by reading off from the program only.
  - *syntax of the language which cannot be described in context-free grammar*.

- Most of the time, a Semantic Analyzer does Type-checking, Flow Ctrl checks,...
- It modifies the syntax tree in order to get that (static) semantically correct code; generating an Abstract Syntax Tree (AST) or Annotated Parse Tree

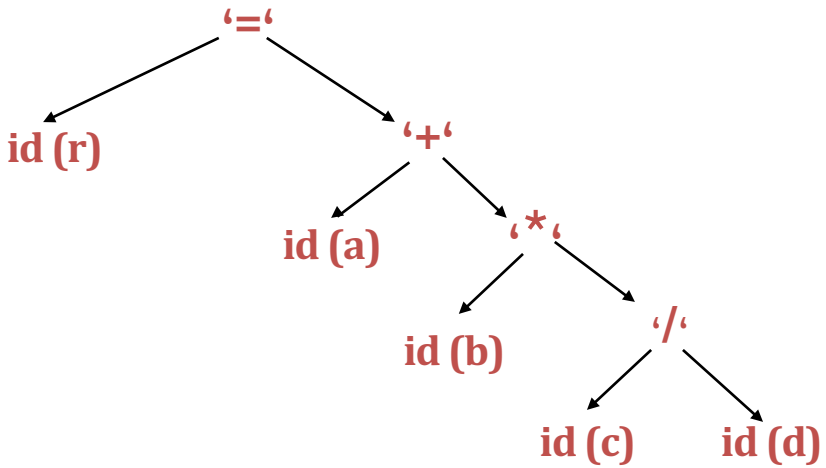
...the input source code: `r = a + b * c / d;` is scanned as '***Lexemes***':

```
<id, r> <assign, = > <id, a> <op, +> <id, b> <op, *> <id, c> <op, /> <id, d>;
```

```
Exp    ::= Exp'+'Exp
         | Exp'-'Exp
         | Exp'*'Exp
         | Exp'/'Exp
         | ID
Assign ::= ID'='Exp
```



Abstract Syntax Tree



4. Intermediate Code Generation (ICG)

- The ICG takes a syntax tree from the semantic analyzer and generates a program in the intermediate language.
  - In some compilers, a source program is translated into an intermediate code first and then the intermediate code is translated into the target language.
  - In other compilers, a source program is translated directly into the target language.

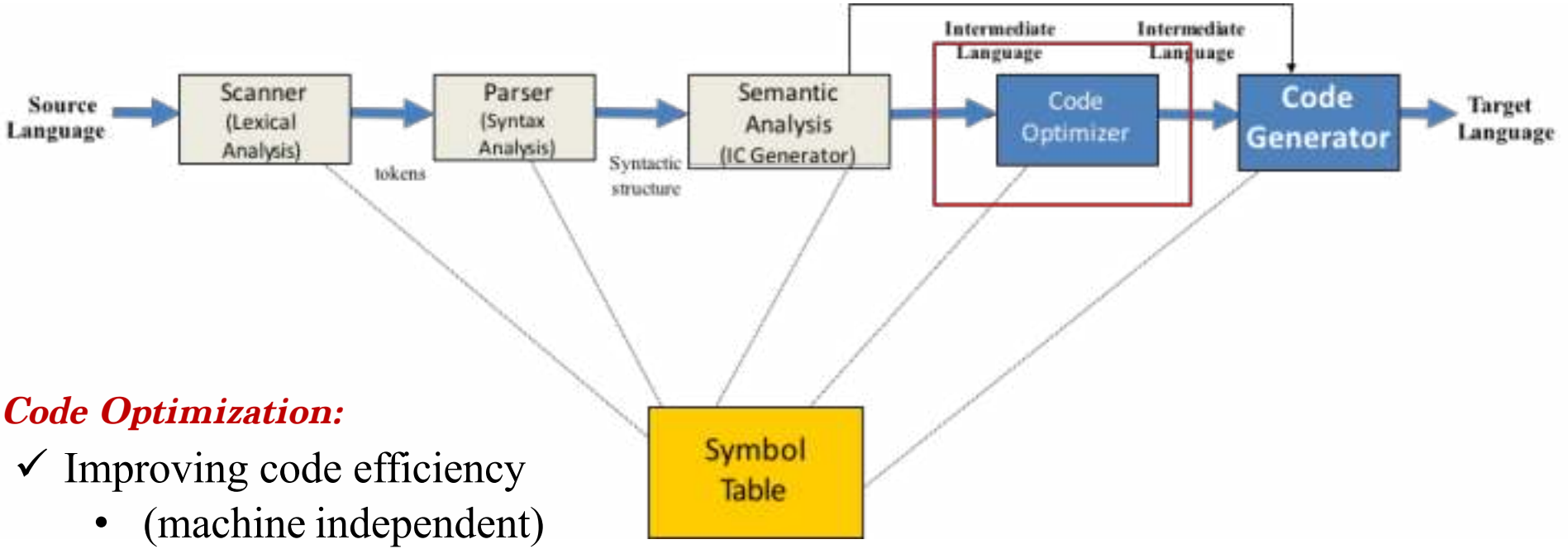
- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. However,
  - **An Intermediate Code should be easy to produce.**
  - **It should be easy to translate the required target machine language.**

4. *Intermediate Code Generation (2)*

- Using intermediate code is beneficial when compilers which translates a single source language to many target languages are required.

  - The front-end of a compiler – *scanner to intermediate code generator* – can be used for every compilers.
  - Different back-ends – *code optimizer and code generator*–is required for each target language.

Code Optimization



Code Optimization:

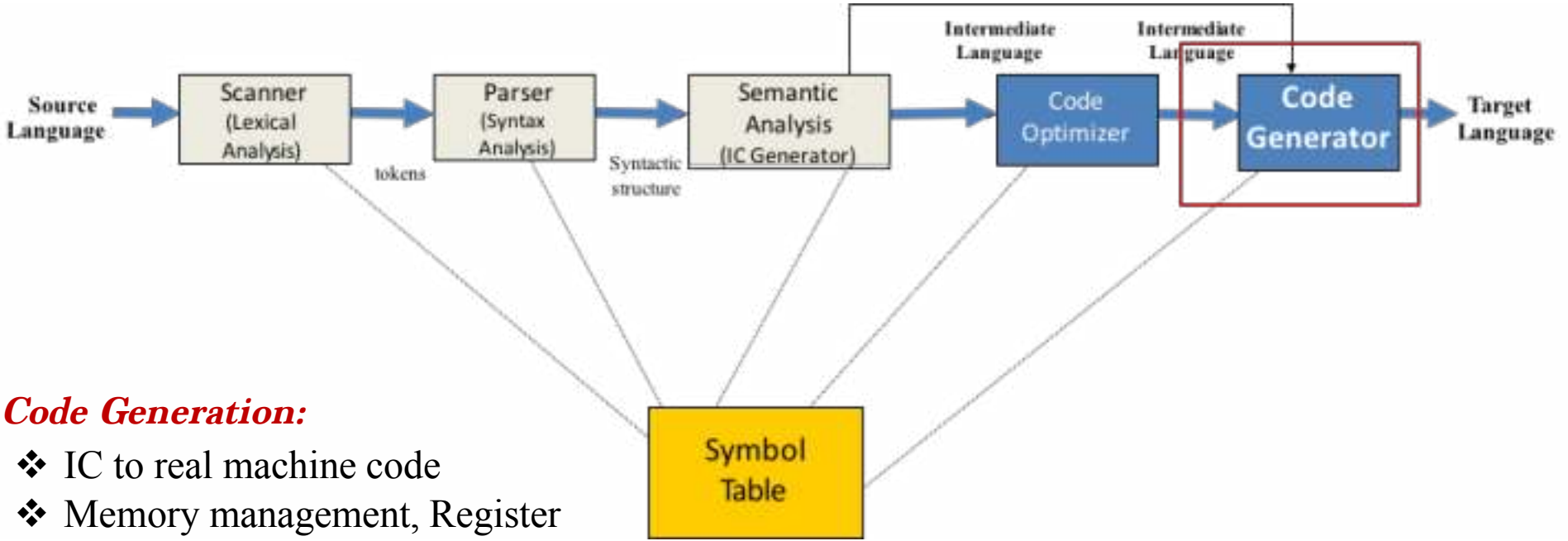
- ✓ Improving code efficiency
  - (machine independent)
- ✓ Finding optimal code is NP

Figure 12: The Code Optimization Phase of a Compiler

5. Code Optimization

- Replacing an inefficient sequence of instructions with a better sequence of instructions. It is sometimes called '***Code Improvement***'.
- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. That is:
  - ✓ Faster codes
  - ✓ Shorter codes
  - ✓ Consumes less resources (energy/power, memory, …)
  - ✓ Target codes
- Code optimization can be done:
  - after semantic analyzing has been performed on a Syntax/Parse Tree
  - or after ICG

Code Generation



Code Generation:

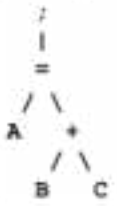
- IC to real machine code
- Memory management, Register allocation, Instruction selection, Instruction scheduling, …

Figure 13: The Semantic Analysis Phases of a Compiler

6. Code Generation

- The code generator takes as input an Intermediate Representation (IR/ICG) of the source program and maps it into the target machine language.
- A code generator:
  - takes either an Intermediate Code Generated (ICG) or a Parse Tree
  - produces a target program.
- The Code Generation phase is the last step of the BACK End, and it enatils:
  - ✓ Memory Management
  - ✓ Assignment/Allocation of Registers to hold variables.
  - ✓ Instruction Selection
  - ✓ Instruction Scheduling

The Phases of a Compiler

Phase	Output	Sample
Programmer (source code producer)	Source string	`A=B+C;`
Scanner (performs *lexical analysis*)	Token string	`'A','=','B','+','C',';'` And *symbol table* with names
Parser (performs *syntax analysis* based on the grammar of the programming language)	Parse tree or abstract syntax tree	
Semantic analyzer (type checking, etc)	Annotated parse tree or abstract syntax tree	

Intermediate code generator	Three-address code, quads, or RTL	`int2fp B t1`
		`+ t1 C t2`
		`:= t2 A`

Optimizer	Three-address code, quads, or RTL	`int2fp B t1` `+ t1 #2.3 A`

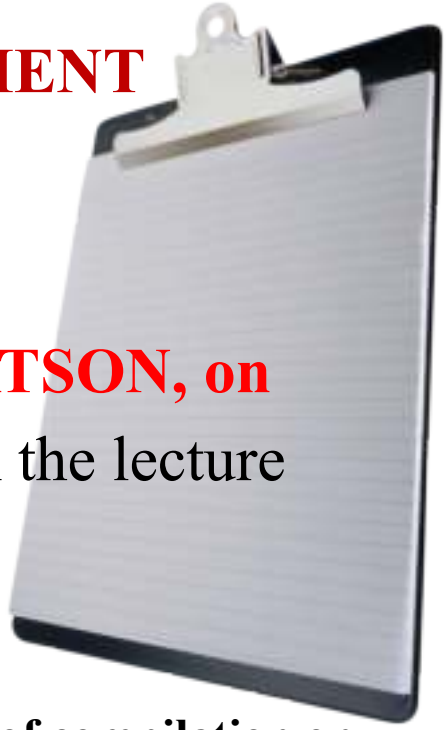
Code generator	Assembly code	`MOVF #2.3,r1`
		`ADDF2 r1,r2`
		`MOVF r2,A`

Peephole optimizer	Assembly code	`ADDF2 #2.3,r2` `MOVF r2,A`

Issues Driving Compiler Design

- Correctness
- Speed (runtime and compile time)
  - Degrees of optimization
  - Multiple passes
- Space
- Feedback to user
- Debugging

LECTURE 2: FURTHER READING & ASSIGNMENT



1. Kindly study the Chapter2 in the Textbook, **DES WATSON, on page 28-34**, for further background understanding on the lecture topic.

2. **Following the lecture, carry out the step in the FRONT End phase of compilation on the following program statement; `c = 2 * w + y / x + z;` as discussed in the lecture. (*Handwritten to be submitted next class).**