CST-402(T): Language Processors

- Course Outcomes:
- On successful completion of the course, students will be able to:

1. Exhibit role of various phases of compilation, with understanding of types of grammars and design complexity of compiler.
2. Design various types of parses and perform operations like string parsing and error handling.
3. Demonstrate syntax directed translation schemes, their implementation for different programming language constructs.
4. Implement different code optimization and code generation techniques using standard data structures.

UNIT – I: Introduction [CO1]

Outcomes:

1. To understand the design complexity of language processor.
2. To understand the functions of various phases of compilation.
3. To understand allied concepts like cross compilation, bootstrapping etc.

Motivation

- **Early days software were written in assembly language. The software was machine specific.**
- **No portability.**
- **Separate module for separate task [Assembler, Linker, Loader].**
- **Software cost for operation increased.**
- **First complier FORTRAN – IN 1950**
- **Total 18 person-years to build.**

Typical Compilation Process

Source program with macros

↓

Preprocessor

↓

Source program

↓

Compiler

↓

Target assembly program

↓

assembler

↓

Relocatable machine code

↓

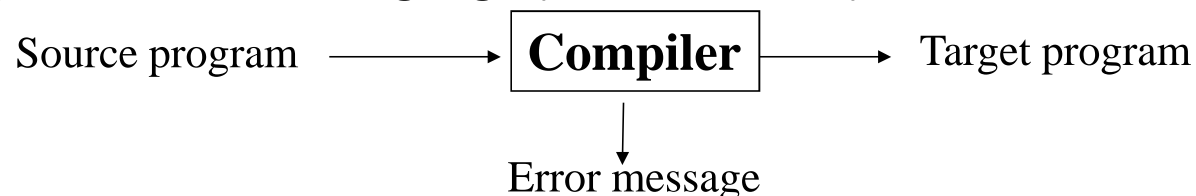
linker

↓

Absolute machine code

Compiler

- A compiler acts as a translator, transforming human-oriented programming languages into computer-oriented machine languages.
- Ignore machine-dependent details for programmer
- A program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language).
  - Two components
    - Understand the program (make sure it is correct)
    - Rewrite the program in the target language.
  - Traditionally, the source language is a high level language and the target language is a low level language (machine code).

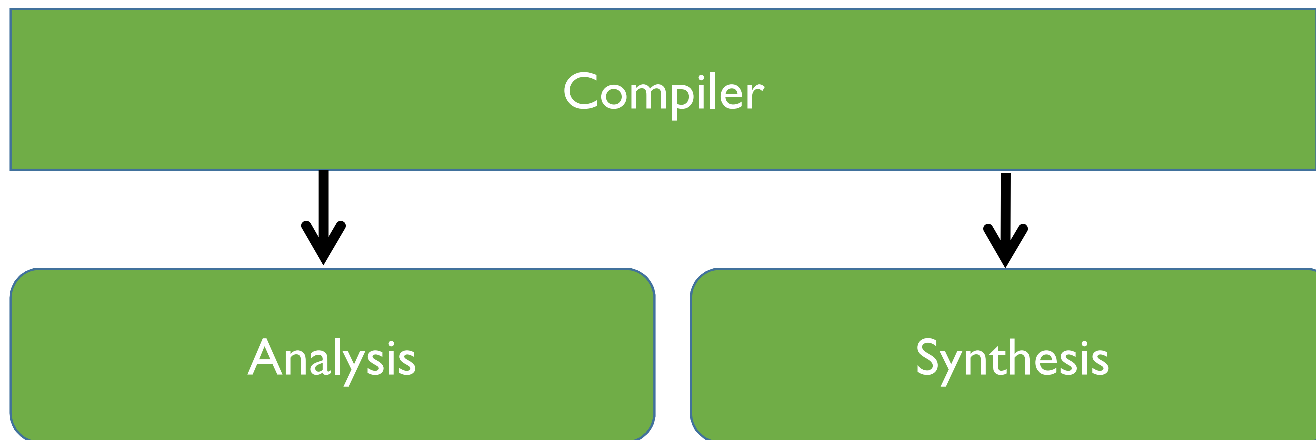


Compilation process

- Compilation of a program proceeds through a fixed series of phases
  - Each phase use an (intermediate) form of the program produced by an earlier phase. **[Cascading effect]**
  - Subsequent phases operate on lower-level code representations. [Close to system]
- Each phase may consist of a number of passes over the program representation
  - Pascal, FORTRAN, C languages designed for one-pass compilation, which explains the need for function prototypes
  - Single-pass compilers need less memory to operate
  - Java, C++ and ADA are multi-pass

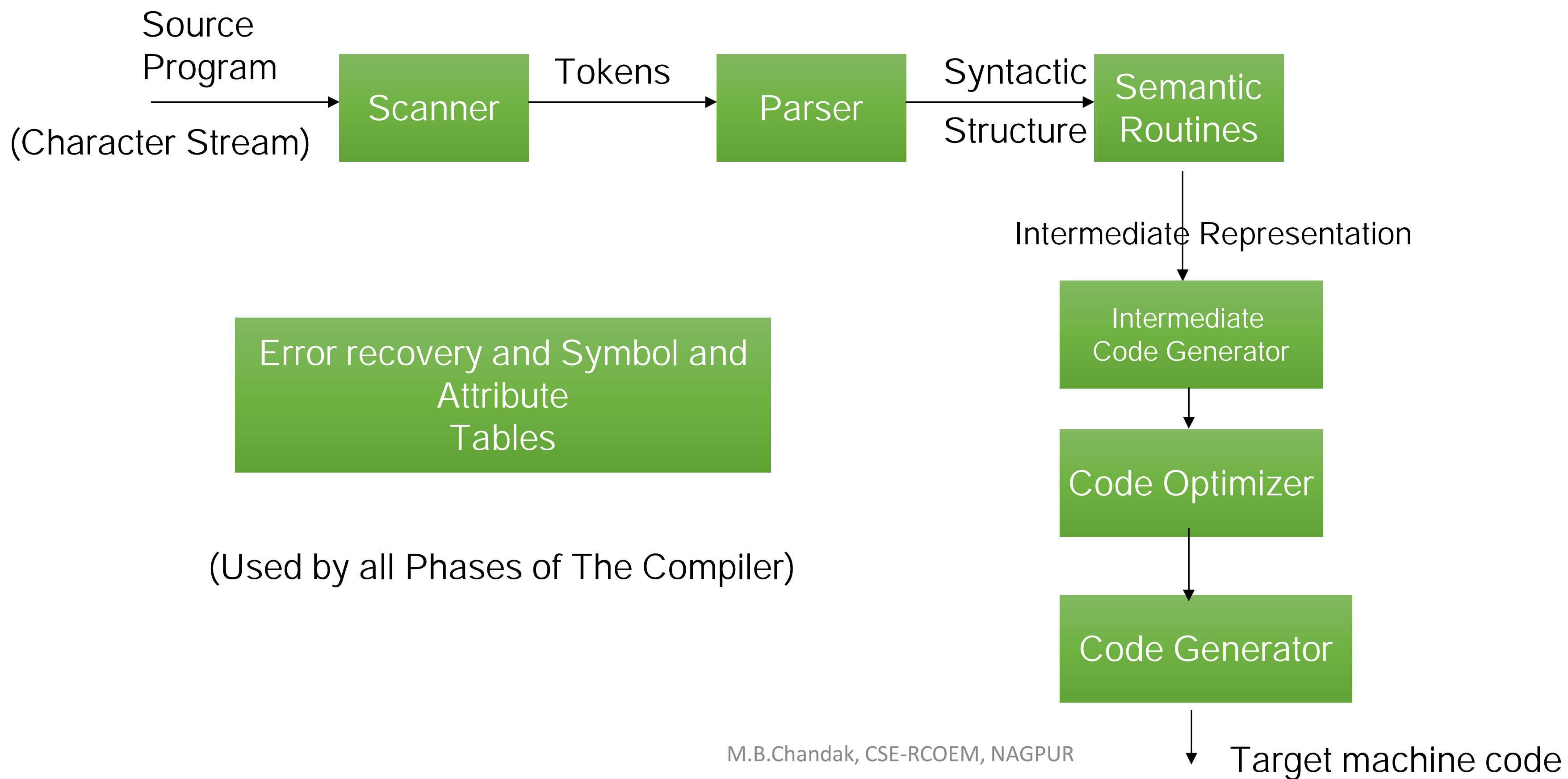
Two major operations

- Any compiler must perform two major tasks

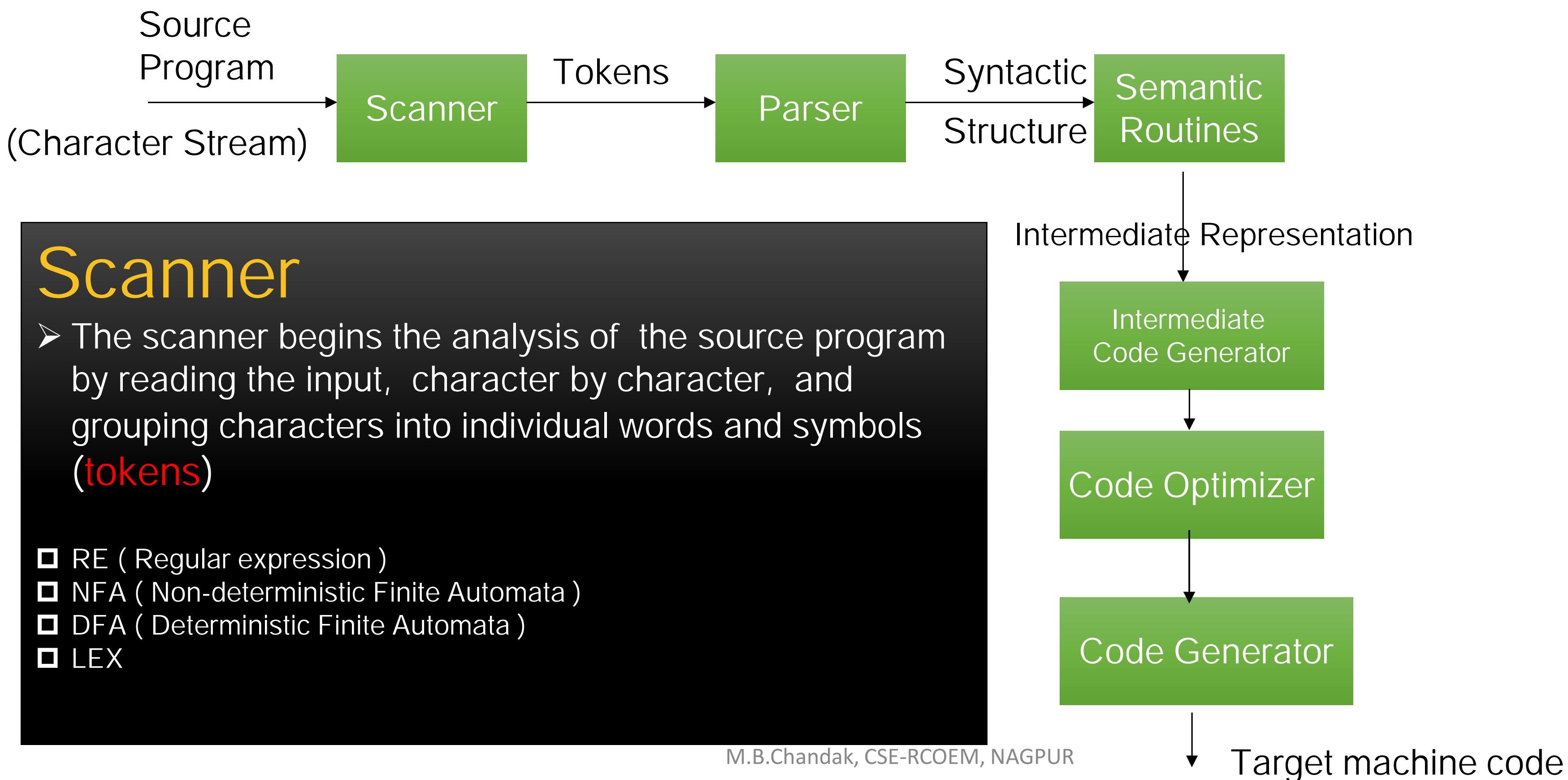


  - Analysis of the source program
  - Synthesis of a machine-language program

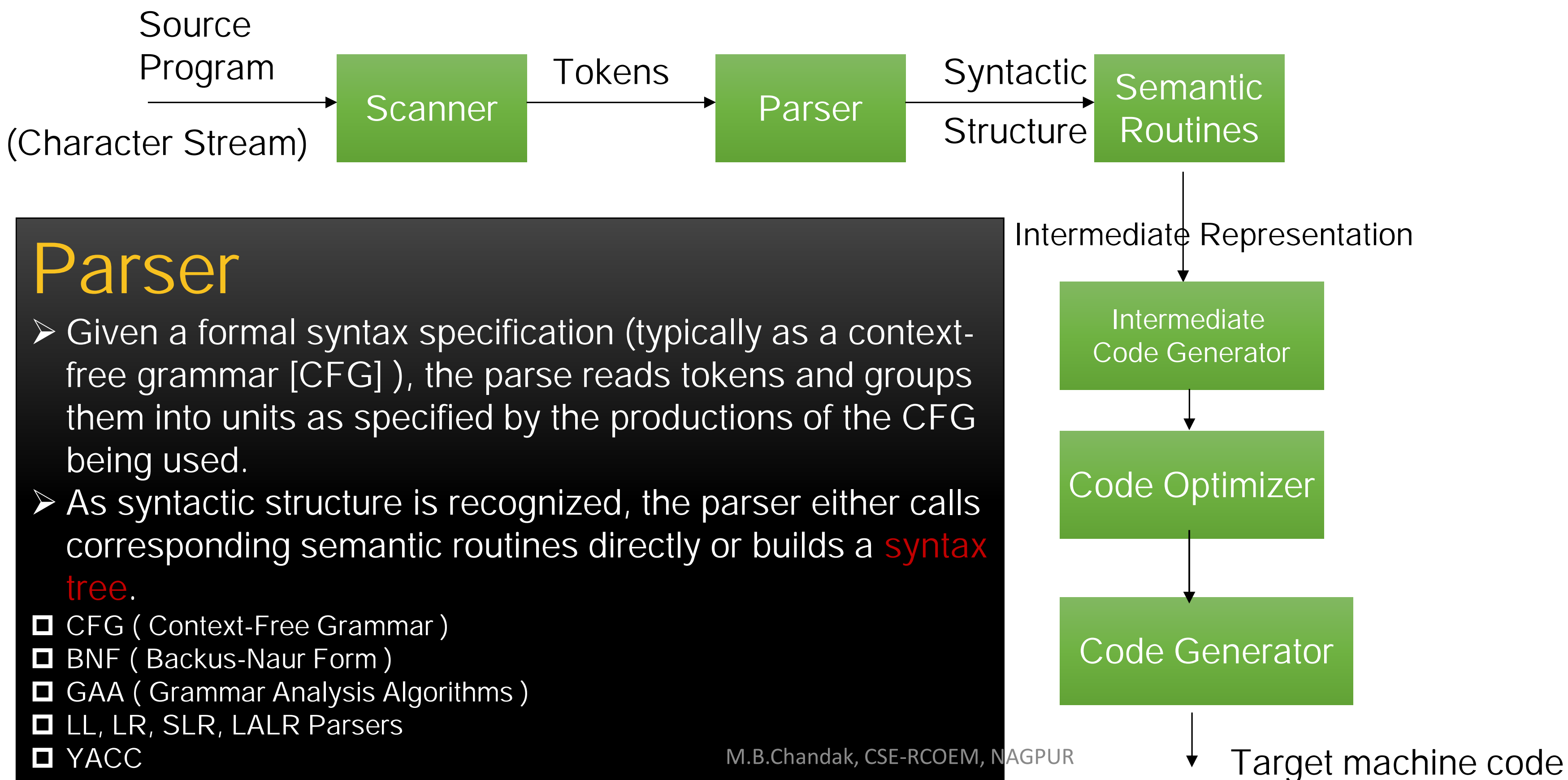
Block Schematic: Modern Compilers



Block Schematic



Block Schematic

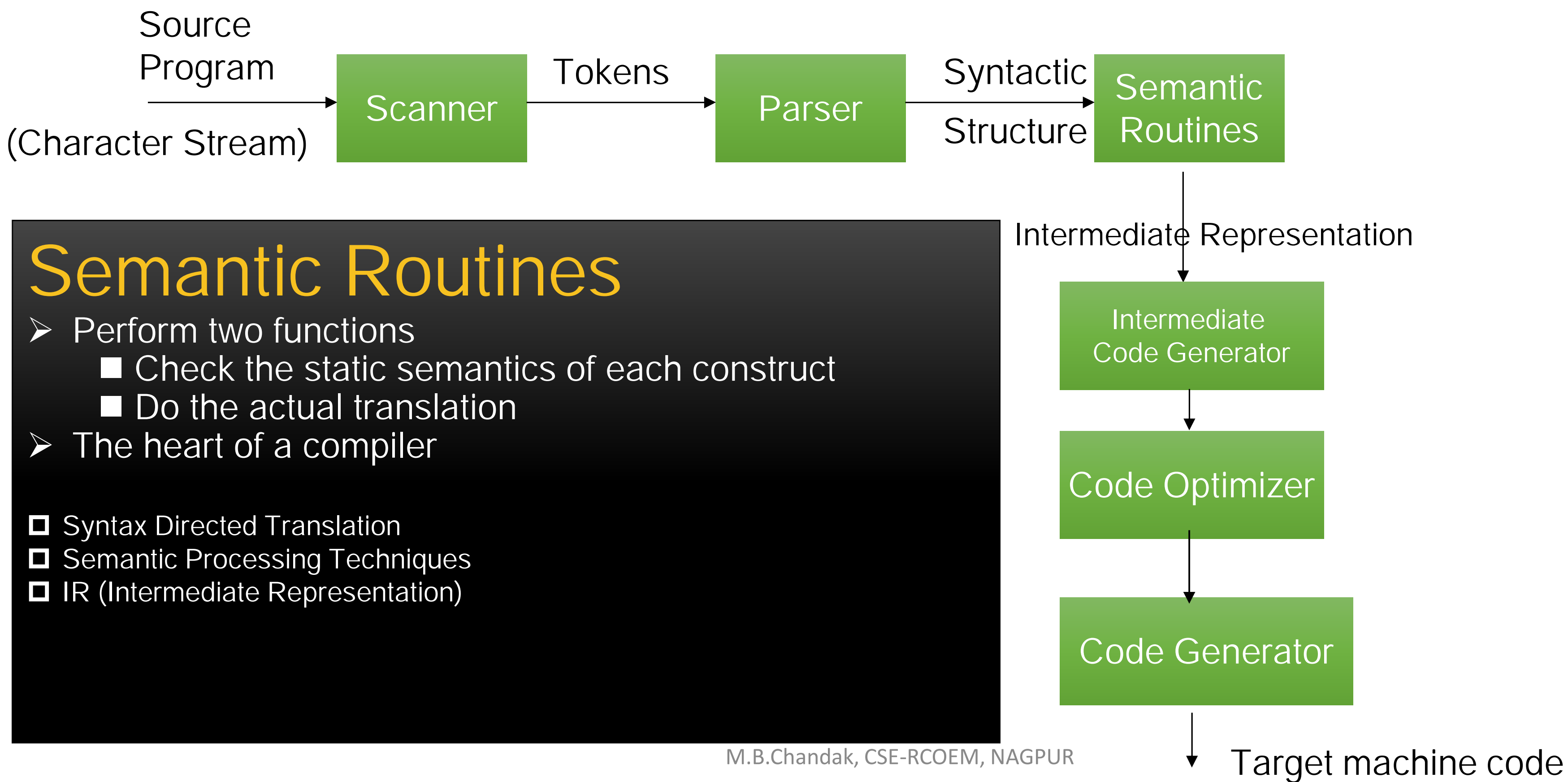


Parser

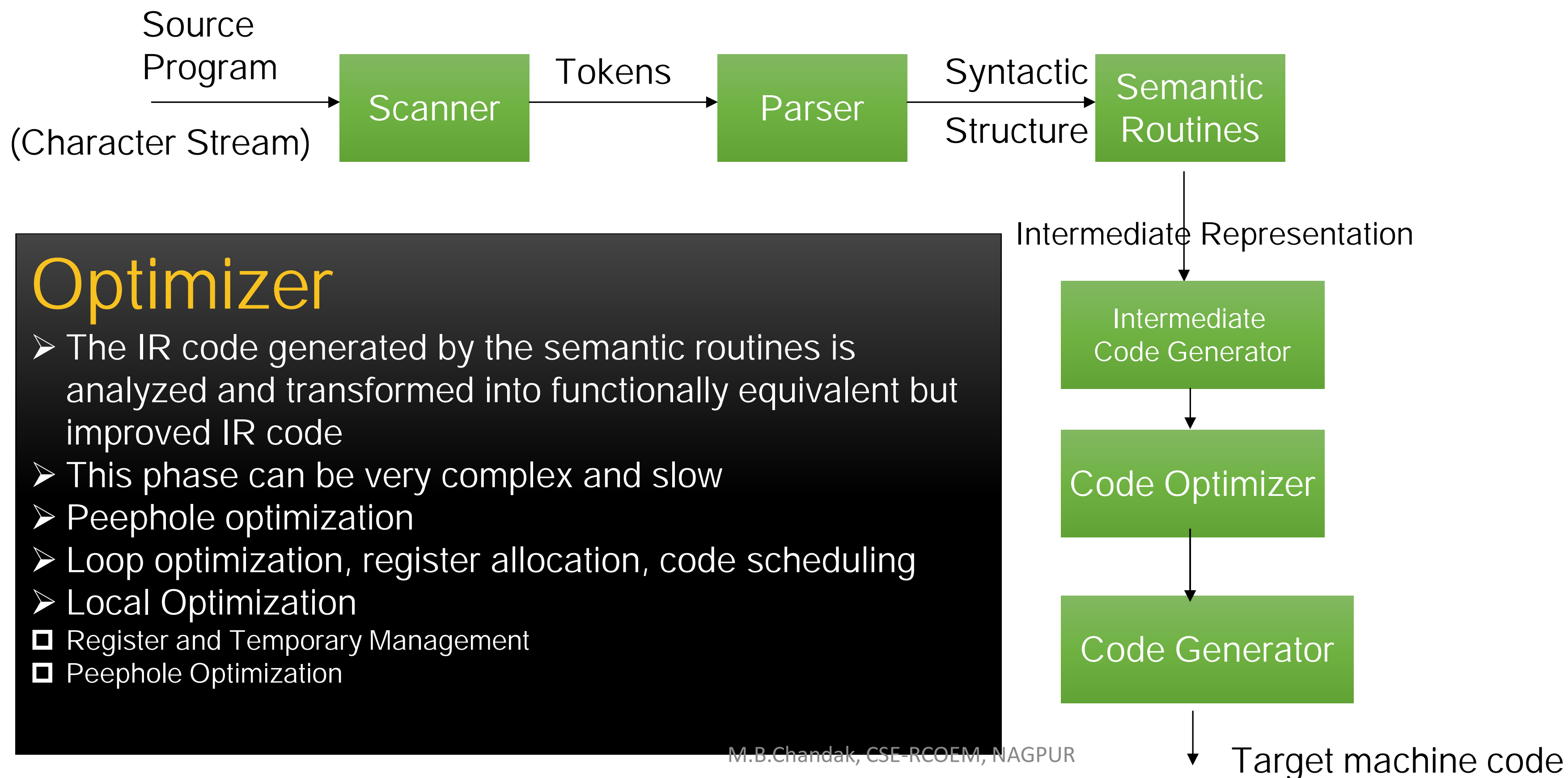
- Given a formal syntax specification (typically as a context-free grammar [CFG]), the parse reads tokens and groups them into units as specified by the productions of the CFG being used.
- As syntactic structure is recognized, the parser either calls corresponding semantic routines directly or builds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

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

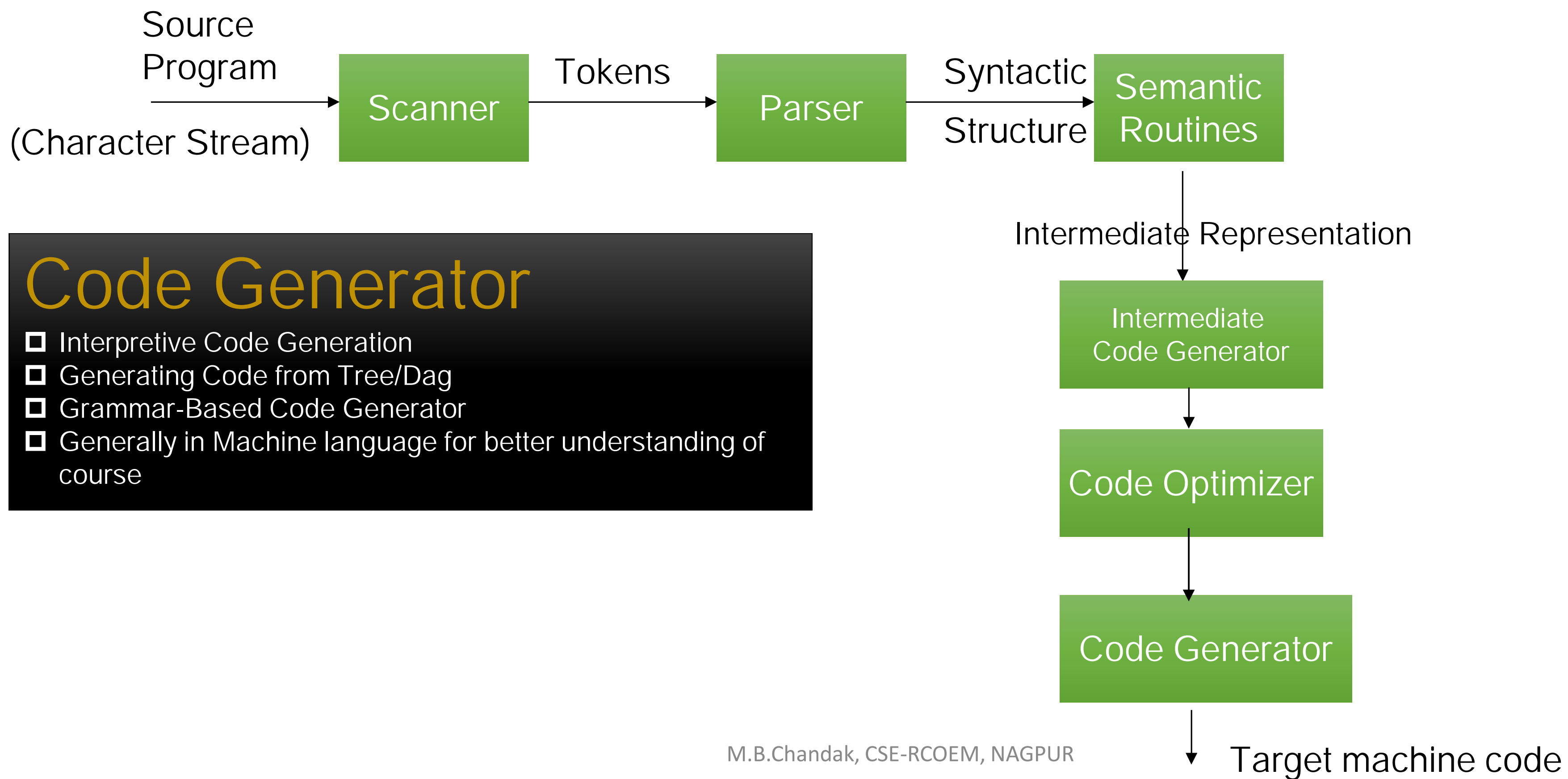
Block Schematic



Block Schematic



Block Schematic



Example:1

SYMBOL TABLE

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

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

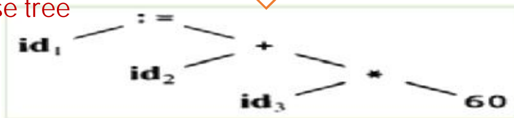
Scanner
[Lexical Analyzer]

Tokens

id_1 := id_2 + id_3 * 60

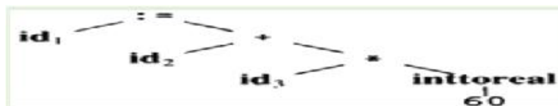
Parser
[Syntax Analyzer]

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

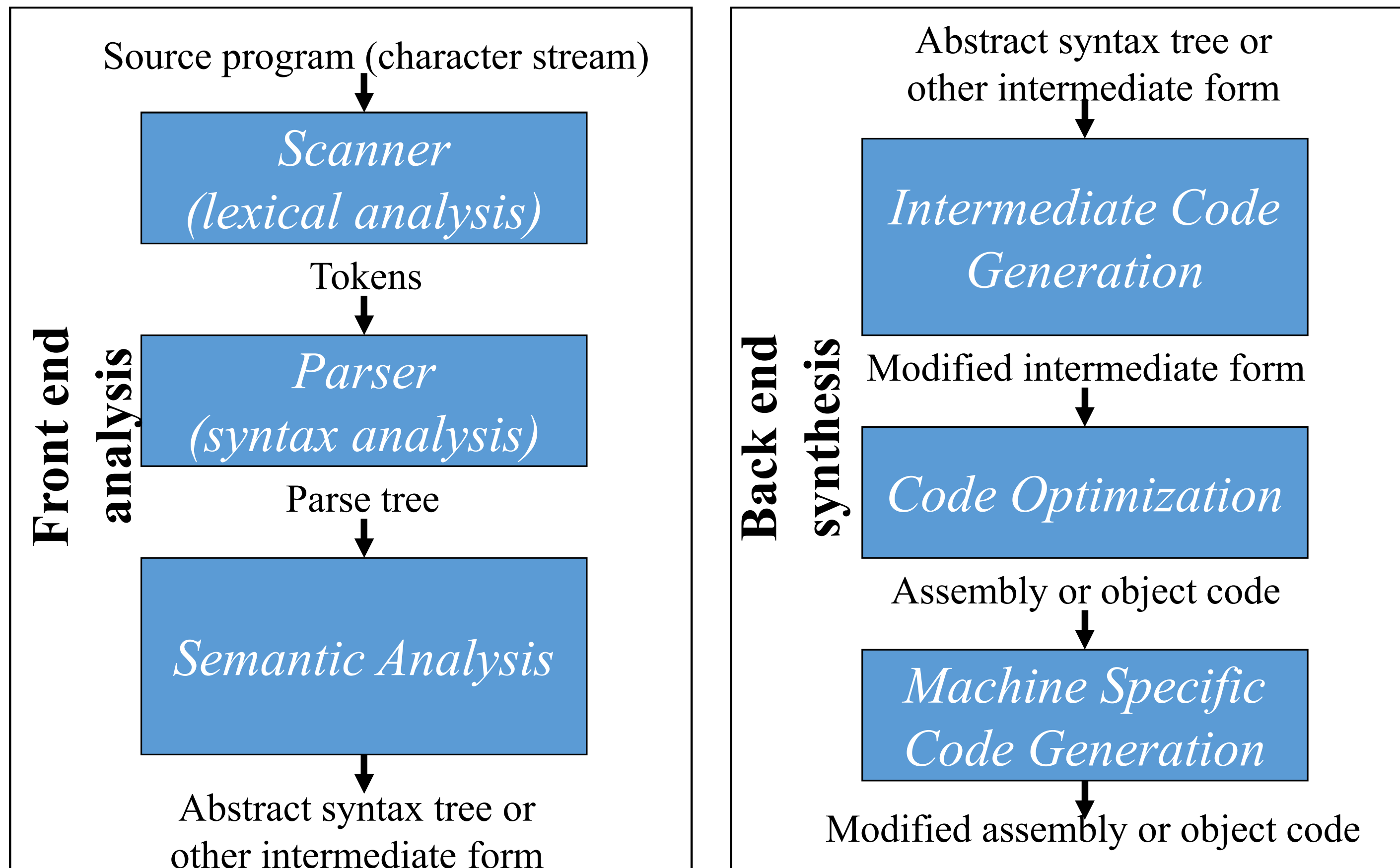
```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

Code Generation

Target machine code

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

Compiler Front-end / Back-end



Phases Functionalities

Lexical Analyzer

- Lexical analysis breaks up a program into tokens/lexicon
  - Grouping characters into non- separable units (tokens)
  - Changing a stream to characters to a stream of tokens

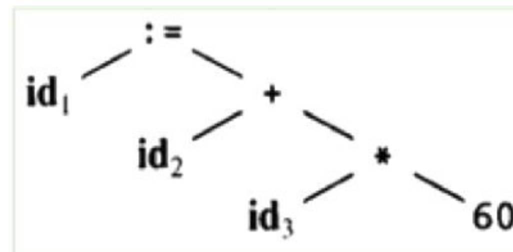
```
program gcd (input, output);
var i, j : integer;
begin
  read (i, j);
  while i <> j do
    if i > j then i := i - j else j := j - i;
  writeln (i)
end.
```

Comment on kinds of errors reported by lexical analyzer

```
program   gcd    (     input    ,
output     )       ;
var         i      ,     j        :
integer    ;       begin
read       (      i      ,        j     )
;        while
i           <>    j     do       if    i
>        j
then        i     :=     i        -     j
else     j
:=          i     -      i
;     writeln     (       i
)           end    .
```

Syntax Analyzer

- Grammatical check of tokens.
  - A syntax error is produced by the compiler when the program does not meet the grammatical specification.
  - For grammatically correct program, this phase generates an internal representation that is easy to manipulate in later phases
    - Typically a syntax tree (also called a parse tree).
- A grammar of a programming language is typically described by a context free grammar, which also defines the structure of the parse tree.



Syntax Analyzer: Parser: Parse Tree

- The **syntax defines the syntactic categories** for language constructs
  - Statements
  - Expressions
  - Declarations
- Categories are subdivided into more detailed categories
  - A Statement is a
    - **For-statement**
    - **If-statement**
    - **Assignment**

```
<statement>      ::= <for-statement> | <if-statement> | <assignment>
<for-statement>  ::= for ( <expression> ; <expression> ; <expression> ) <statement>
<assignment>     ::= <identifier> := <expression>
```

Semantic Analysis/SDTS

- Semantic analysis is applied by a compiler to discover the meaning of a program by analyzing its parse tree or abstract syntax tree.
- A program without grammatical errors may not always be correct program.
    - *pos = init + rate * 60*
    - **What if *pos* is a char while *init* and *rate* are integers?**
    - This kind of errors cannot be found by the parser
    - Semantic analysis finds this type of error and ensure that the program has a meaning.

    - *C++: Semantically strong language?*

Types of Semantic Checks

- **Static semantic checks (done by the compiler) are performed at compile time**
  - **Type checking**
  - **Every variable is declared before used**
  - **Identifiers are used in appropriate contexts**
  - **Check subroutine call arguments**
  - **Check labels**
- **Dynamic semantic checks are performed at run time, and the compiler produces code that performs these checks**
  - **Array subscript values are within bounds**
  - **Arithmetic errors, e.g. division by zero**
  - **Pointers are not dereferenced unless pointing to valid object**
  - **A variable is used but hasn't been initialized**
  - **When a check fails at run time, an exception is raised**

Semantic Analysis

- A language is “strongly typed” if (type) errors are always detected.
  - Errors are either detected at compile time or at run time
  - Languages that are strongly typed are Ada, Java, ML, Haskell
  - Languages that are not strongly typed are Fortran, Pascal, C/C++, Lisp
- Strong typing makes language safe and easier to use, but potentially slower because of dynamic semantic checks
- In some languages, most (type) errors are detected late at run time which is detrimental to reliability e.g. early Basic, Lisp, Prolog, some script languages

Intermediate Code Generator

- Conversion of parse tree into intermediate code.
- Various forms of intermediate code: Quadruple, Triplet, Indirect Triplet etc.
- Temporary storage is used in representation.
- Proper use of data structures is key factor.

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

Code Optimization

- Purpose:
  - To improve efficiency of code.
  - To reduce time required for execution.
- Types
  - Local Optimization
  - Loop Optimization
  - Peep-hole Optimization
  - Role of data structures and their memory implementation is important [Trees/Graphs]
- Optimization:
  - Machine independent
  - Machine dependent

Code Generation

- Purpose:
  - To convert optimized code into machine code.
  - Depends upon machine architecture.
  - For learning purpose assembly language code will be used.
- Example:

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

Summary

- Compiler front-end: lexical analysis, syntax analysis, semantic analysis
  - Tasks: understanding the source code, making sure the source code is written correctly
- Compiler back-end: Intermediate code generation/improvement, and Machine code generation/improvement.
  - Tasks: translating the program to a semantically the same program (in a different language).

Questions

- Explain the various phases of compilation.
- Open source tools: for various phases of compilation.
- File name in which details of keywords of “C” language are stored. Its locations and structure.
- C++ is semantically strong language? Justify
- Any five rules to design lexical analyzer. For example: “?” “!” symbols are not considered as valid tokens.
- Advantages of Late and Early binding approaches.
- How to decide the complier is one pass or two pass. Any two rules.
- How to classify the front end and back end component of any software product.

Course Curriculum

UNIT-I

Introduction to Compilers- Compilers and translators, Phases of compiler design, cross compiler, Bootstrapping, Design of Lexical analyser, LEX.

UNIT-II

Syntax Analysis- Specification of syntax of programming languages using CFG, Top-down parser, design of LL(1) parser, bottom up parsing technique, LR parsing, Design of SLR, CLR, LALR parsers, YACC.

UNIT-III

Syntax directed translation- Study of syntax directed definitions & syntax directed translation schemes, implementation of SDTS, intermediate notations- postfix, syntax tree, TAC, translation of expressions, controls structures, declarations, procedure calls, Array reference.

UNIT-IV

Storage allocation & Error Handling- Run time storage administration stack allocation, symbol table management, Error detection and recovery- lexical, syntactic and semantic.

UNIT-V

Code optimization- Important code optimization techniques, loop optimization, control flow analysis, data flow analysis, Loop invariant computation, Induction variable removal, Elimination of Common sub expression.

UNIT-VI

Code generation – Problems in code generation, Simple code generator, Register allocation and assignment, Code generation from DAG, Peephole optimization.

Web resource: www.mbchandak.com

TEXTBOOKS

Aho, Sethi, and Ullman; Compilers Principles Techniques and Tools; Second Edition, Pearson education, 2008.
Alfred V. Aho and Jeffery D. Ullman; Principles of Compiler Design; Narosa Pub. House, 1977.
Vinu V. Das; Compiler Design using Flex and Yacc; PHI Publication, 2008.