Introduction of Lecturer

Designation, Office Details

Md Masum Billah
Lecturer
Department of Computer Science
Faculty of Science & Technology
Office: D Building
Email: billah.masumcu@aiub.edu
Website: masum-billah.net
Mobile: 01646677086
American International University-Bangladesh (AIUB)

Introduction To Compiler

Course Code: CSC 3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:		**Week No:**	**1**	**Semester:**	**Summer**
Lecturer:	*Md Masum Billah; billah.masumcu@aiub.edu* *https://masum-billah.net*				

Introduction of Lecturer

Educational Background

- **M.Sc. in Computer Engineering**

Major: Embedded Systems

University of Duisburg-Essen, Germany

- **B.Sc. in Computer Science & Engineering**

University of Chittagong

Introduction of Lecturer

Research Interests

- IoT
- Machine Learning
- Data Science
- Deep Learning

Lecture Outline

1. Vision, Mission, Quality Policy, and Goals of AIUB
2. Vision and Mission of Computer Science Department
3. Course Evaluation
4. Objective of this Course
5. Basic things of a Compiler
6. Language Processors

Vision

AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB) envisions promoting professionals and excellent leadership catering to the technological progress and development needs of the country.

Mission

AMERICAN INTERNATIONAL UNIVERSITY-BANGLADESH (AIUB) is committed to provide quality and excellent computer-based academic programs responsive to the emerging challenges of the time. It is dedicated to nurture and produce competent world class professional imbued with strong sense of ethical values ready to face the competitive world of arts, business, science, social science and technology.

Quality Policy

"Quality shall be adhered to in conformity with the prescribed national and international standards of quality and excellence including those provided by the professional bodies and organizations. The American International University- Bangladesh is committed to translate into actions the programs, projects and activities related to the sustainable delivery of quality management operation system. The students being the valued customers are the central focus of the university shall be provided with utmost care and attention to meet their primordial needs and future career success. In view of this commitment, the university shall exert best efforts to harmonize its action through collaboration, cooperation and consultation with every unit and components of the university."

Goals

- Sustain development and progress of the university
- Continue to upgrade educational services and facilities responsive of the demands for change and needs of the society
- Inculcate professional culture among management, faculty and personnel in the attainment of the institution's vision, mission and goals
- Enhance research consciousness in discovering new dimensions for curriculum development and enrichment
- Implement meaningful and relevant community outreach programs reflective of the available resources and expertise of the university
- Establish strong networking of programs, sharing of resources and expertise with local and international educational institutions and organizations
- Accelerate the participation of alumni, students and professionals in the implementation of educational programs and development of projects designed to expand and improve global academic standards

Vision of Computer Science Department

Our vision is to be the preeminent Department of Computer Science through creating recognized professionals who will provide innovative solutions by leveraging contemporary research methods and development techniques of computing that is in line with the national and global context.

Mission of Computer Science Department

The mission of the Department of Computer Science of AIUB is to educate students in a student-centric dynamic learning environment; to provide advanced facilities for conducting innovative research and development to meet the challenges of the modern era of computing, and to motivate them towards a life-long learning process.

Course Evaluation

Mid-Term Evaluation

- Attendance : 10%
- Quizzes : 30%
- Lab Performance : 20%
- Term Exam : 40%

Course Evaluation

Final Term Evaluation

- Attendance : 10%
- Quizzes : 30%
- Lab performance : 20%
- Final Term : 40%
- **Grand Final = 40% of Midterm + 60% of Final Term**

Objectives and Outcomes

Objectives:

- Understand the objective of this Course
- Understand the basic concept of a compiler

Outcomes:

- Students should be able to understand the importance of a compiler
- Students will analyze the language Processors.

Basic things of a Compiler

1. What is a Compiler?
2. Why do we need a compiler?
3. Why study compilers?

Basic things of a Compiler

Compilers Construction touches many topics in Computer Science

1. Theory
   - Finite State Automata, Grammars and Parsing, data-flow
2. Algorithms
   - Graph manipulation, dynamic programming
3. Data structures
   - Symbol tables, abstract syntax trees
4. Software Engineering
   - Software development environments, debugging
5. Artificial Intelligence
   - Heuristic based search

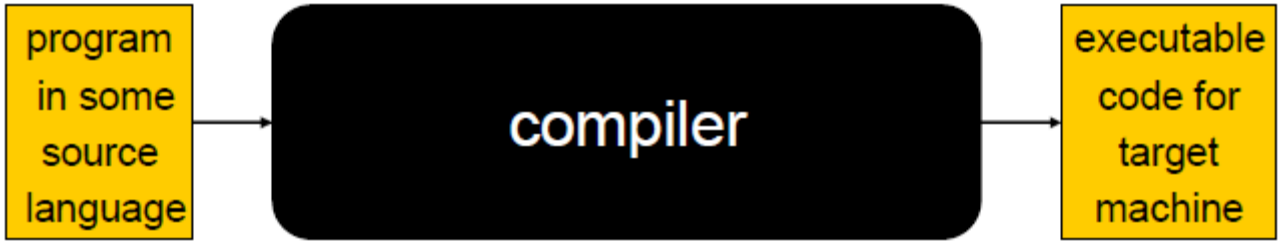
Language processors

Some common language processors are:

1. Compiler
2. Interpreter
3. Preprocessor
4. Assembler

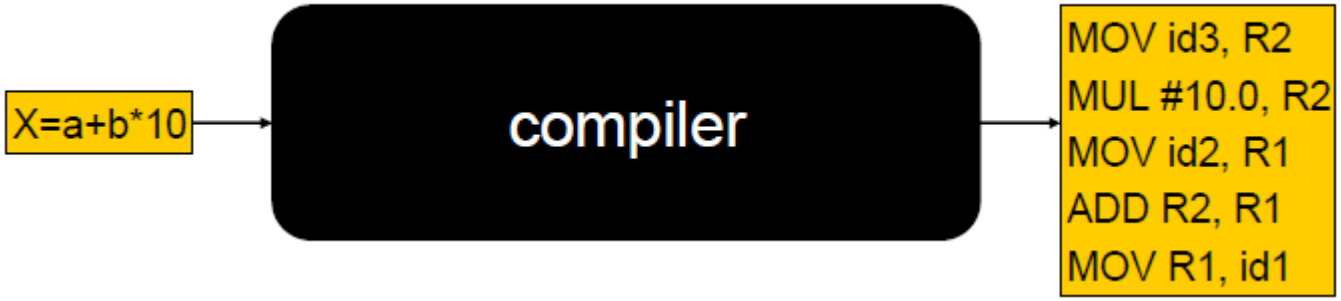
Compiler

A compiler is a program that reads a program written in one language and translates it into another language.



Traditionally, compilers go from high-level languages to low-level languages.

Example


X=a+b*10
compiler
MOV id3, R2
MUL #10.0, R2
MOV id2, R1
ADD R2, R1
MOV R1, id1

Interpreter

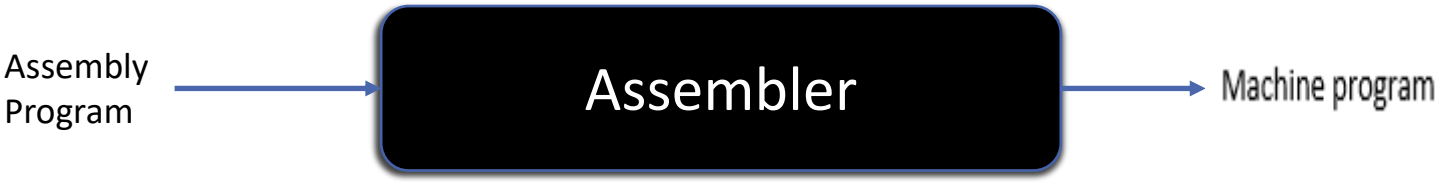
An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Preprocessor

Preprocessing performs (usually simple) operations on the source file(s) prior to compilation.

Assembler

An Assembler is a translator that translates Assembly language to machine code. So, an assembler is a type of a compiler and the source code is written in Assembly language.



Differences between Compiler and Interpreter

Compiler	Interpreter
Compiler takes whole program as input.	Interpreter takes single instruction as input.
Intermediate Object code is generated.	No Intermediate Object Code is Generated
Memory Requirement: More	Memory Requirement is Less
Program need not be compiled every time .	Every time higher level program Is converted into lower level program.

Lecture References

A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools*** *(*The Dragon Book*)*, [Second Edition]

References

1. A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools(***The Dragon Book*)*, [Second Edition]

2. **Principles of Compiler Design** (2nd Revised Edition 2009) A. A. Puntambekar

3. Basics of Compiler Design Torben Mogensen

Introduction To Compiler

Course Code: CSC 3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**2**	**Week No:**	**2**	**Semester:**	**Summer**
Lecturer:	*Md Masum Billah; billah.masumcu@aiub.edu* *www.masum-billah.net*				

Lecture Outline

1. Language Processing System
2. Different Phases of a Compiler

Objectives and Outcomes

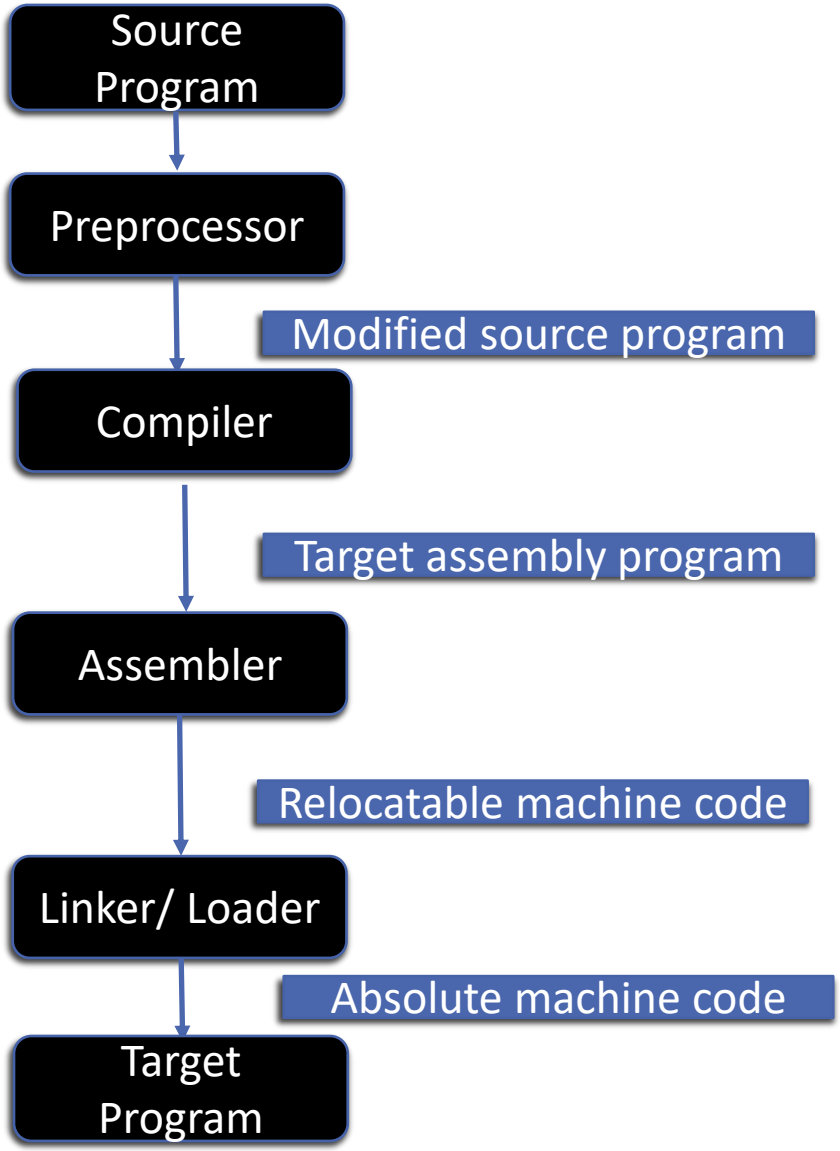
Objectives:

- Understand the structure of a language processing system.
- Understand the structure of a compiler.
- Understand the tools involved(Scanner generator, Parser generator, etc)

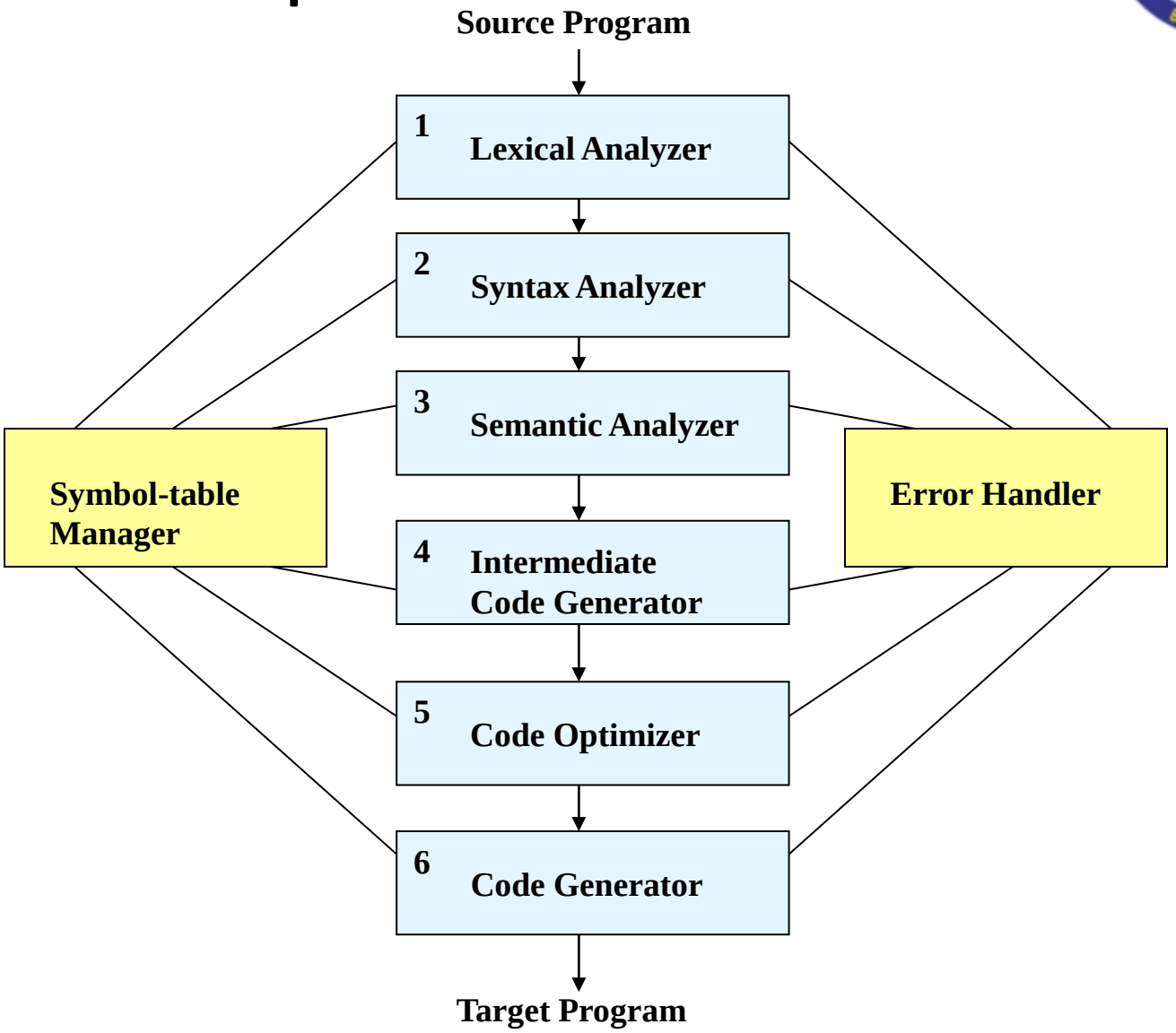
Outcomes:

- Students should be able to understand how a language is processed step by step.
- Students will analyze the phases of a compiler.

Language Processing System



The Phases of a Compiler



Two main Phases of a Compiler

1. **Analysis Phase**: Breaks up a source program into constituent pieces and produces an internal representation of it called intermediate code.
    I. Lexical Analyzer
    II. Syntax Analyzer
    III. Semantic Analyzer
    IV. Intermediate code generator

2. **Synthesis Phase**: Translates the intermediate code into the target program.
    V. Code optimizer
    VI. Code generator

The Phases of a Compiler

Lexical Analyzer: The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form *(token-name, attribute-value).* For example, suppose a source program contains the assignment statement

p o s i t i o n = i n i t i a l + r a t e * 60

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. p o s i t i o n is a lexeme that would be mapped into a token **(id,** 1), where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol table entry for p o s i t i o n . The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

The Phases of a Compiler

2. The assignment symbol = is a lexeme that is mapped into the token (=).Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. Initial is a lexeme that is mapped into the token **(id,** 2), where 2 points to the symbol-table entry for i n i t i a l .

4. + is a lexeme that is mapped into the token (+).

5. r a t e is a lexeme that is mapped into the token **(id,** 3), where 3 points to the symbol-table entry for r a t e .

6. * is a lexeme that is mapped into the token (*).

7. 60 is a lexeme that is mapped into the token (60).

The Phases of a Compiler

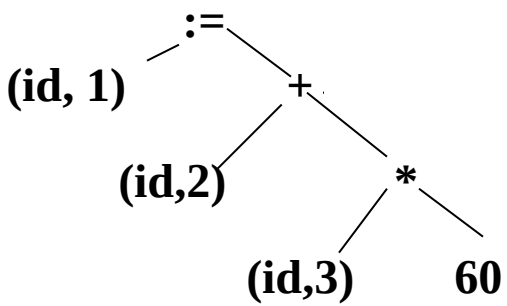
After lexical analyzer compiled the above expression the output of lexical analyzer would be

<i d , l > <=> <id, 2> <+> **<id,** 3> <*> <60>

Lexical Errors: A lexical error is a mistake in a lexeme, for examples, typing **tehn** instead of **then**, or missing off one of the quotes in a literal.

The Phases of a Compiler

Syntax Analyzer: The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation. A syntax tree for the token stream (obtain from lexical analyzer) is shown below as the output of this phase.



The Phases of a Compiler

The tree has an interior node labeled * with **(id, 3)** as its left child and the integer **60** as its right child. The node **(id, 3)** represents the identifier **rate.** The node labeled * makes it explicit that we must first multiply the value of **r a t e** by **60.** The node labeled + indicates that we must add the result of this multiplication to the value of i n i t i a l . The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier p o s i t i o n . This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

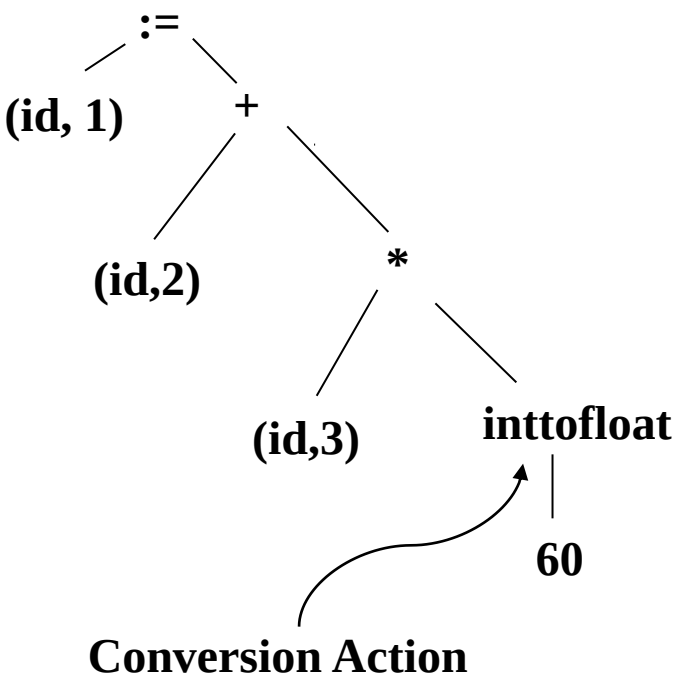
Syntax Error: A grammatical error is a one that violates the (grammatical) rules of the language, for example if x = 7 y := 4 (missing **then**).

The Phases of a Compiler

Semantic Analyzer: The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. In practice semantic analyzers are mainly concerned with type checking and type coercion based on type rules. The semantic analyzer produces an annotated syntax tree as an output.

The Phases of a Compiler



The Phases of a Compiler

Semantic Errors: During compilation Semantic analyzer will recognize the following semantic errors.

- Datatype mismatch
- Undeclared variable
- Multiple declaration of a variable in a scope
- Actual and formal parameter mismatch

Lecture References

A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools*** *(*The Dragon Book*)*, [Second Edition]

References

1. A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools(***The Dragon Book*)*, [Second Edition]

2. **Principles of Compiler Design** (2nd Revised Edition 2009) A. A. Puntambekar

3. Basics of Compiler Design Torben Mogensen

Introduction To Compiler

Course Code: CSC 3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**3**	**Week No:**	**3**	**Semester:**	**Summer**
Lecturer:	*Md Masum Billah; billah.masumcu@aiub.edu* *Masum-billah.net*				

Lecture Outline

1. Phases of a Compiler
2. Practice on Different Input Expressions
3. Linker and Loader
4. Front end and Back end of a compiler
5. Symbol Table Management
6. Error Handler

Objectives and Outcomes

Objectives:

- Understand the Structure of a compiler
- Understand the tools involved(Scanner generator, Parser generator, etc)

Outcome:

- Students will be able to represent the simulation of all phases of a compiler for inputs.

The Phases of a Compiler

Intermediate Code generator: After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties:

- Easy to Produce and
- Easy to translate into target program

The intermediate representation can have a variety of forms. In this course we consider an intermediate form called " **three address code**".

The Phases of a Compiler

We need to follow some steps to generate three address code.

- Each three address instruction has at most one operator on the right side.
- The compiler must generate a temporary name to hold the value computed by each instruction.
- Some three address instructions have fewer than three operands.

So the output of the intermediate code generator will be

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

The Phases of a Compiler

Code Optimizer: The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

- Find More Efficient Ways to Execute Code
- Replace Code With More Optimal Statements
- Significantly improve the running time of the target program

So this phase optimized the code and produced the output as follows

temp1 := id3 * 60.0

id1 := id2 + temp1

The Phases of a Compiler

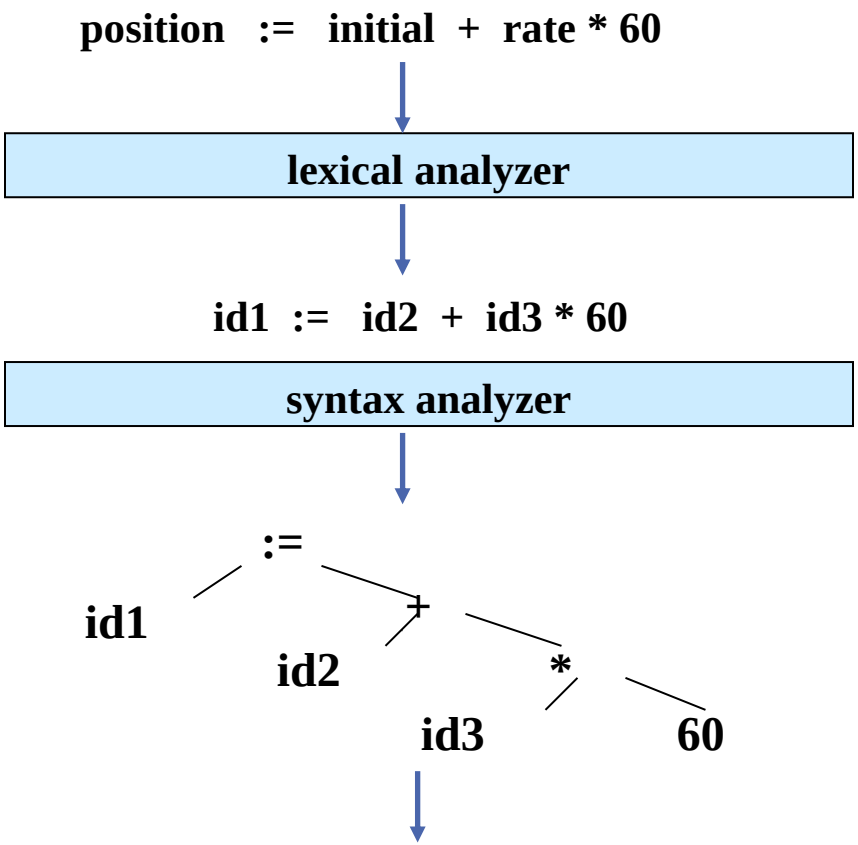
Code Generator: The final phase of the compiler is to generate code for a specific machine. In this phase we consider:

- memory management
- register assignment

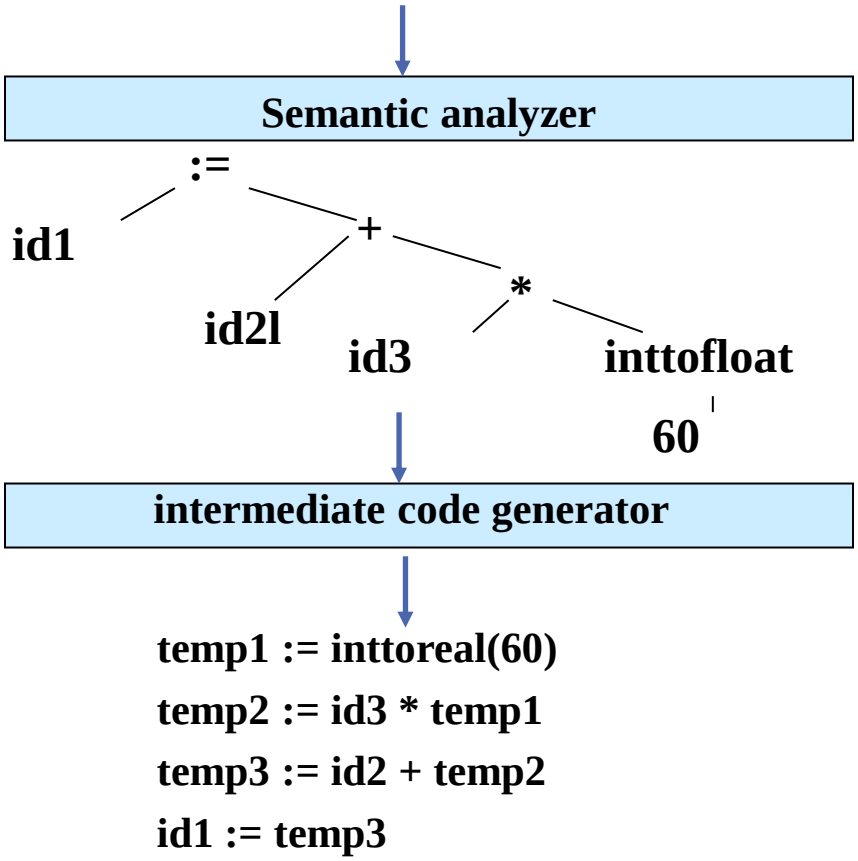
The output from this phase is usually assembly language or relocatable machine code.

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

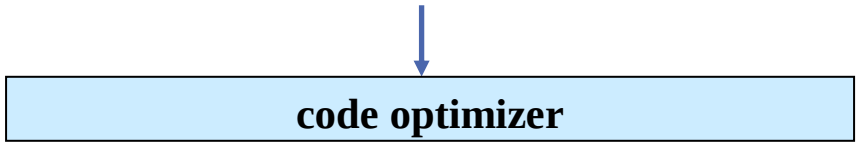
Reviewing the Entire Process



Reviewing the Entire Process



Reviewing the Entire Process



temp1 := id3 * 60.0

id1 := id2 + temp1

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

Exercises

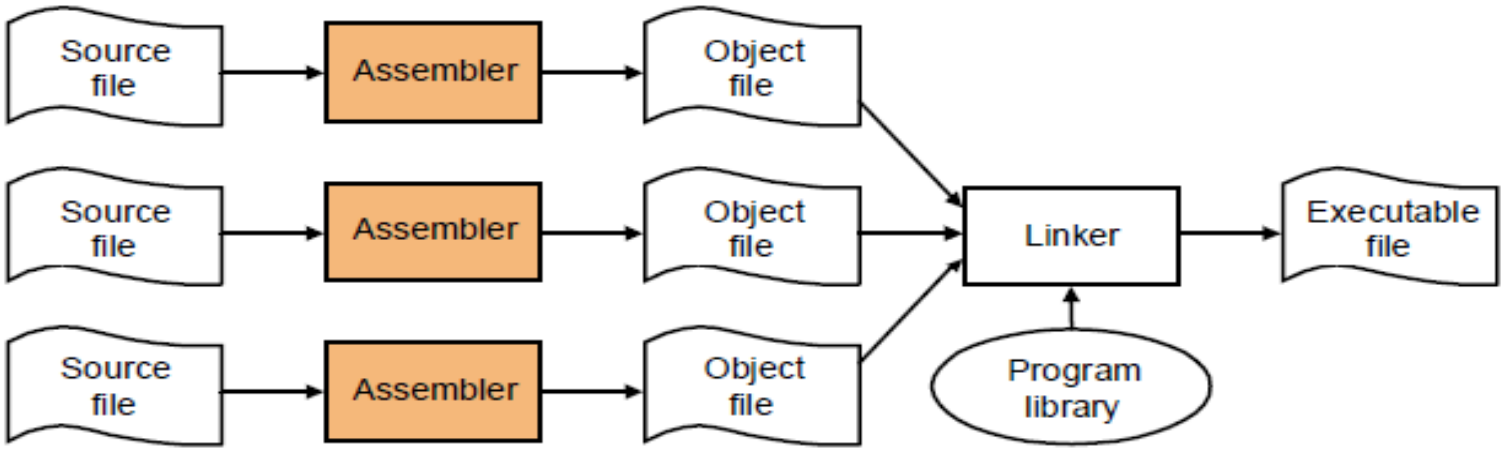
Find the output for the following expressions

1. a=a +b *c *2
2. Y= b+c-d+20

Linker and Loader

Linker: A linker, also called link editor or binder, is a program that combines the object modules to form an executable program. In general , in case of a large program, programmers prefer to break the code in to smaller modules, as this simplifies the programming task. Eventually, when the source code of all the modules has been converted in to object code, all the modules need to be put together, which is done by the linker

Process for producing an executable file



Loader

A loader is a special type of a program that copies programs from a storage device to the main memory, where they can be executed.

Front end and Back end of a Compiler

Front end:

I. Lexical Analyzer
II. Syntax Analyzer
III. Semantic Analyzer
IV. Intermediate Code Generator

Back end :

V. Code Optimizer
VI. Code Generator

Advantages of Using Front-end and Back- end

Retargeting: Build a compiler for a new machine by attaching a new code generator to an existing front-end

Optimization: Reuse intermediate code optimizers in compilers for different languages and different machines.

Symbol Table Management

A symbol table is a data structure containing all the identifiers (i.e. names of variables, procedures etc.) of a source program together with all the attributes of each identifier.

For variables, typical attributes include:

- its type,
- how much memory it occupies,
- its scope.

For procedures and functions, typical attributes include:

- the number and type of each argument (if any),
- the method of passing each argument, and
- the type of value returned (if any).

Symbol Table Management

The purpose of the symbol table is to provide quick and uniform access to identifier attributes throughout the compilation process. Information is usually put into the symbol table throughout the analysis phase and used for the synthesis phase.

Error Handler

Each of the six phases (but mainly the analysis phases) of a compiler can encounter errors. On detecting an error the compiler must:

- report the error in a helpful way,
- correct the error if possible, and
- continue processing (if possible) after the error to look for further errors.

Lecture References

A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools*** *(*The Dragon Book*)*, [Second Edition]

References

1. A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools(***The Dragon Book*)*, [Second Edition]

2. **Principles of Compiler Design** (2nd Revised Edition 2009) A. A. Puntambekar

3. Basics of Compiler Design Torben Mogensen