2CS701 Compiler Construction

Evaluation Method

Assessment scheme	CE			LPW		SEE
Component weightage	0.4			0.2		0.4
	Quiz 0.35	Sessional 0.35	Project 0.30	Continuous Evaluation 75%	Viva Voce 25%	100 Marks

Objective of the course:

- To make the students understand the structure of compiler for translating end to end languages

- To understand the phases of compiler design in detail. It covers Lexical Analysis, Syntax Analysis, Semantic analysis, Code optimization, and Code Generation phases.

Course Learning Outcomes(CLO)

- After successful completion of the course, the students will be able to:

  - **summarize the functionalities of various phases of compiler**
  - **apply language theory concepts to various phases of compiler design**
  - **identify appropriate optimization technique for compilation process**
  - **develop a miniature compiler using appropriate compiler design tool**

Preliminaries Required

- Basic knowledge of programming languages.
- Basic knowledge of Finite State Machine and Context Free Grammar.
- Knowledge of a higher level programming language for the programming assignments.

Textbook:

- Aho, Lam, Ullman, Sethi, Compilers, Principles, Techniques and Tools, Pearson
- Keith D Cooper & Linda Torczon, Engineering a Compiler, Elsevier
- Jean Paul Trembly & Paul G Sorenson, The theory and Practice of Compiler writing, McGraw Hill

Lesson Planning

Topic	Hours	Mapped CLO
Course overview and significance Translator , Compilers, Interpreter	1	1
Analysis-Synthesis Compiler Model	1	1
Single Pass vs Multi-pass Cousins of Compiler Types of compiler and applications Symbol Table,	1	1
Lexical Analysis		
Role of lexical analyzer Token, Lexeme Use of Regular Expression for token Look-ahead operator	1	1
Input Buffering, and Significance Sentinel	1	1
Finite Automata, Optimization of DFA based Pattern Matching Lexical analyzer generator	1	1,2
Regular expression to DFA	1	1,2

Lesson Planning continue

Topic	Hours	Mapped CLO
Syntax Analysis		
Role of Parser Introduction to Error recovery Context Free Grammar Comparison of CFG and Regular Expressions	1	1
Context Free Grammar for programming constructs	1	1,2
Introduction to Top Down Parsing LL(1) Parsing		
Recursive-Descent Parser	1	1,2
Need of Left Recursion elimination Left Recursion elimination : Direct, Indirect	1	1,2
Predictive parse table generation LL(1), LL(k) Grammar	1	1
Error recovery strategies , Error recovery at LL(1) parsers	1	1,2

Lesson Planning continue

Topic	Hours	Mapped CLO
Bottom-up Parsing Shift Reduce Parsing Shift/Reduce conflict, Reduce/Reduce conflict	1	1
LR Parsing Model	1	1
LR(0) Parse table Generation	1	1,2
SLR Parse table Generation	1	1,2,3
LR(1) Parse table Generation	1	1,2
LALR(1) Parse table Generation, Error recovery at LR Parsers	1	1,2
Comparison of LR Parsers and LL Parsers Dealing with ambiguous grammar Parsing Generator Tools	1	1,2
Operator Precedence Parsing Error recovery at operator precedence parsing	1	1,2

Lesson Planning continue

Topic	Hours	Mapped CLO
Syntax Directed Translation		
Syntax Directed Definition (SDD), Annotated Tree, S-attributed and L-attributed SDD	1	1
Type checking SDD	1	1,2,4
Bottom – Up Evaluation of S – Attributed Definitions		
Bottom – Up Evaluation of L – Attributed Definitions	1	1,2
Translation scheme	1	1
Top – Down Translation	1	1,2
Recursive evaluators	1	1,2

Lesson Planning continue

Topic	Hours	Mapped CLO
Run-time Environments		
Static Memory Allocation , Stack Memory Allocation	1	1
Symbol Table Management	1	1
Intermediate code generation		
Intermediate code representations	1	1`
Types of three address statements	1	1
Three address code for declaration, assignment 1,2 statements, Boolean expressions	1	1,2
Three address code for control(if..else) construct	1	1,2
Three address code for Loop construct	1	1,2
Back-patching	1	1,2

Lesson Planning continue

Topic	Hours	Mapped CLO
Code Generation		
Functionalities of Code Generation phase Issues in design of a code generation	1	1
Basic code generation	1	1,2
Basic blocks and flow graph, Control flow		1
A simple code generator	1	1,2
Register allocation and assignment	1	1,3
Code Optimization	1	1,2
Local and Global Code optimization	1	3
Machine independent optimization techniques	1	3
Peephole Optimization	1	3

Course Outline

- Introduction to Compiling
- Lexical Analysis
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Semantic Analysis, Type Checking
- Run-Time Organization
- Intermediate Code Generation
- Code Optimization
- Code Generator

TRANSLATOR

- A translator is a program that takes as input a program written in one language and produces as output a program in another language.
- The translator performs another very important role, the error-detection. Any violation of HLL specification would be detected and reported to the programmers. Important role of translator are:

  – 1 Translating the HLL program input into an equivalent machine language program.
  – 2 Providing diagnostic messages wherever the programmer violates specification of the HLL.

TYPE OF TRANSLATORS:-

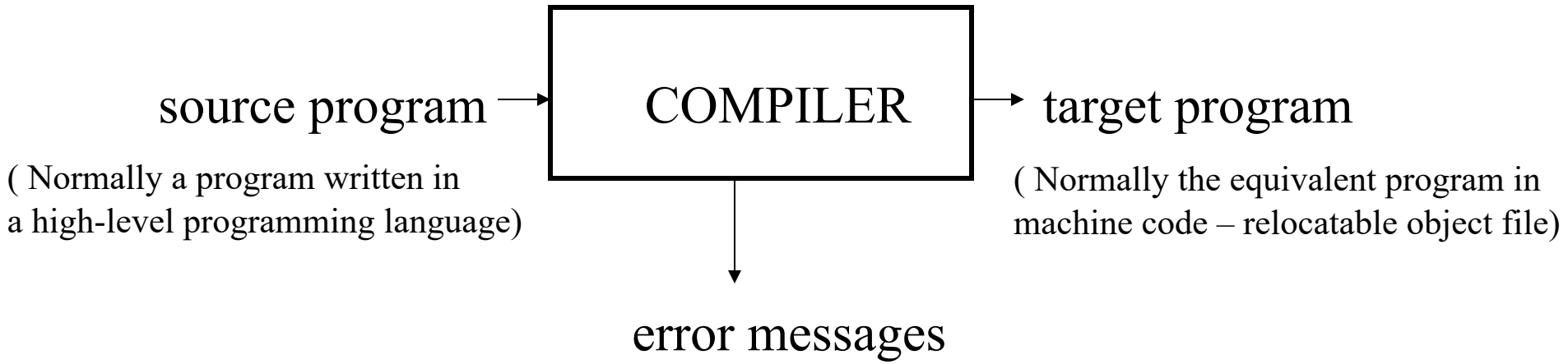
a. Compiler

b. Interpreter

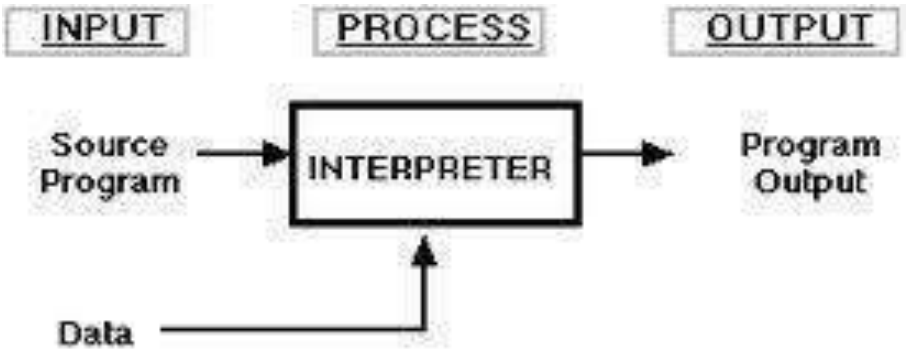
c. Preprocessor

COMPILERS

- Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



- Executing a program written in HLL programming language is basically of two parts. The source program must first be compiled and translated into a object program. Then the resulting object program is loaded into a memory executed.
- **Interpreter*:*** An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

Lexical analysis
Synatx analysis
Semantic analysis
Direct Execution

Interpreter

Advantages:

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes a various may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages:

- The execution of the program is *slower*. *Memory* consumption is more.

OVERVIEW OF LANGUAGE PROCESSING SYSTEM

Preprocessor

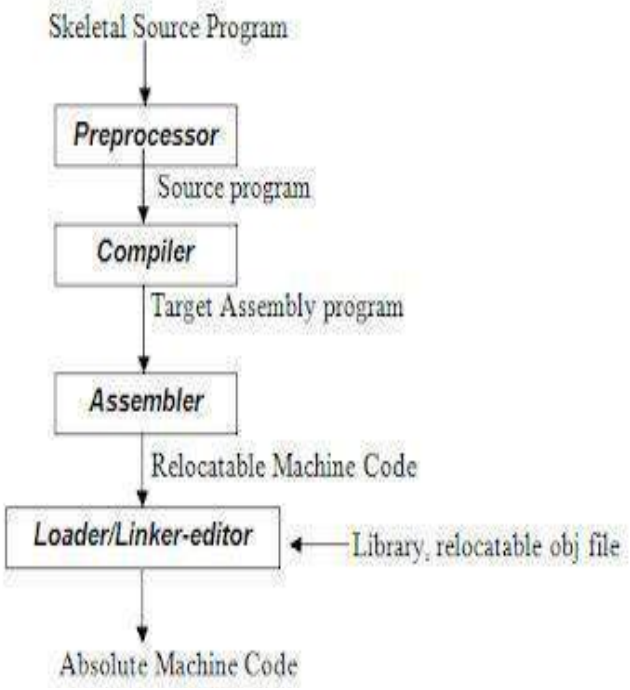
- Produce input to compilers. They may perform the following functions.

  1. Macro processing: may allow a user to define macros that are short hands for longer constructs.

  2. File inclusion: may include header files into the program text.

  3. Rational preprocessor: augment older languages with more modern flow- of-control and data structuring facilities.

  4. Language Extensions: attempts to add capabilities to the language by certain amounts to build-in macro



Assembler

- Programmers found it difficult to write or read programs in machine language.
- Mnemonic (symbols) for each machine instruction is used, which subsequently translate into machine language.
- Such a mnemonic machine language is now called an assembly language.
- Programs known as assembler were written to automate the translation of assembly language into machine language.
- The input to an assembler program is called source program, the output is a machine translation (Object Program)

Loader and Linker

- Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user‟s program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome this problems of wasted translation time and memory. System programmers developed another component called loader
- “A loader is a program that places programs into memory and prepares them for execution.” It would be more efficient if subroutines could be translated into object form the loader could”relocate” directly behind the user‟s program. The task of adjusting programs othey may be placed in arbitrary core locations is called relocation.

Other Applications

- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL.
  - Many software having a complex front-end may need techniques used in compiler design.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

Major Parts of Compilers

- There are two major parts of a compiler: **Analysis** and **Synthesis**

- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Phases of A Compiler

- Each phase transforms the source program from one representation into another representation.
- They communicate with error handlers.
- They communicate with the symbol table.

Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

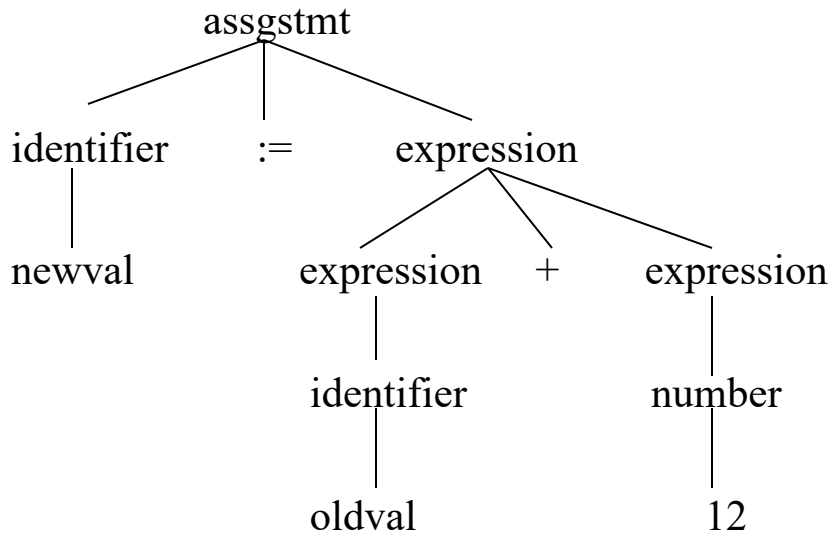
  Ex: newval := oldval + 12 => tokens:

newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

Syntax Analyzer (CFG)

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus Naur Form) to specify a CFG

  assgstmt -> identifier := expression
  expression -> identifier
  expression -> number
  expression -> expression + expression

Syntax Analyzer versus Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
  - The syntax analyzer deals with recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - ***Top-Down Parsing,***
  - ***Bottom-Up Parsing***
- **Top-Down Parsing:**
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).
- **Bottom-Up Parsing:**
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-Precedence Parsing – simple, restrictive, easy to implement
  - LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars
- Ex:

  newval := oldval + 12

  - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Intermediate Code Generation

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes.
- Ex:

```
newval  :=  oldval * fact + 1
            ↓
id1  :=  id2 * id3 + 1
            ↓
MULT     id2,id3,temp1
ADD      temp1,#1,temp2
MOV      temp2,,id1
```

Intermediates Codes (Quadraples)

Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

- Ex:

```
MULT    id2,id3,temp1
ADD     temp1,#1,id1
```

Code Generator

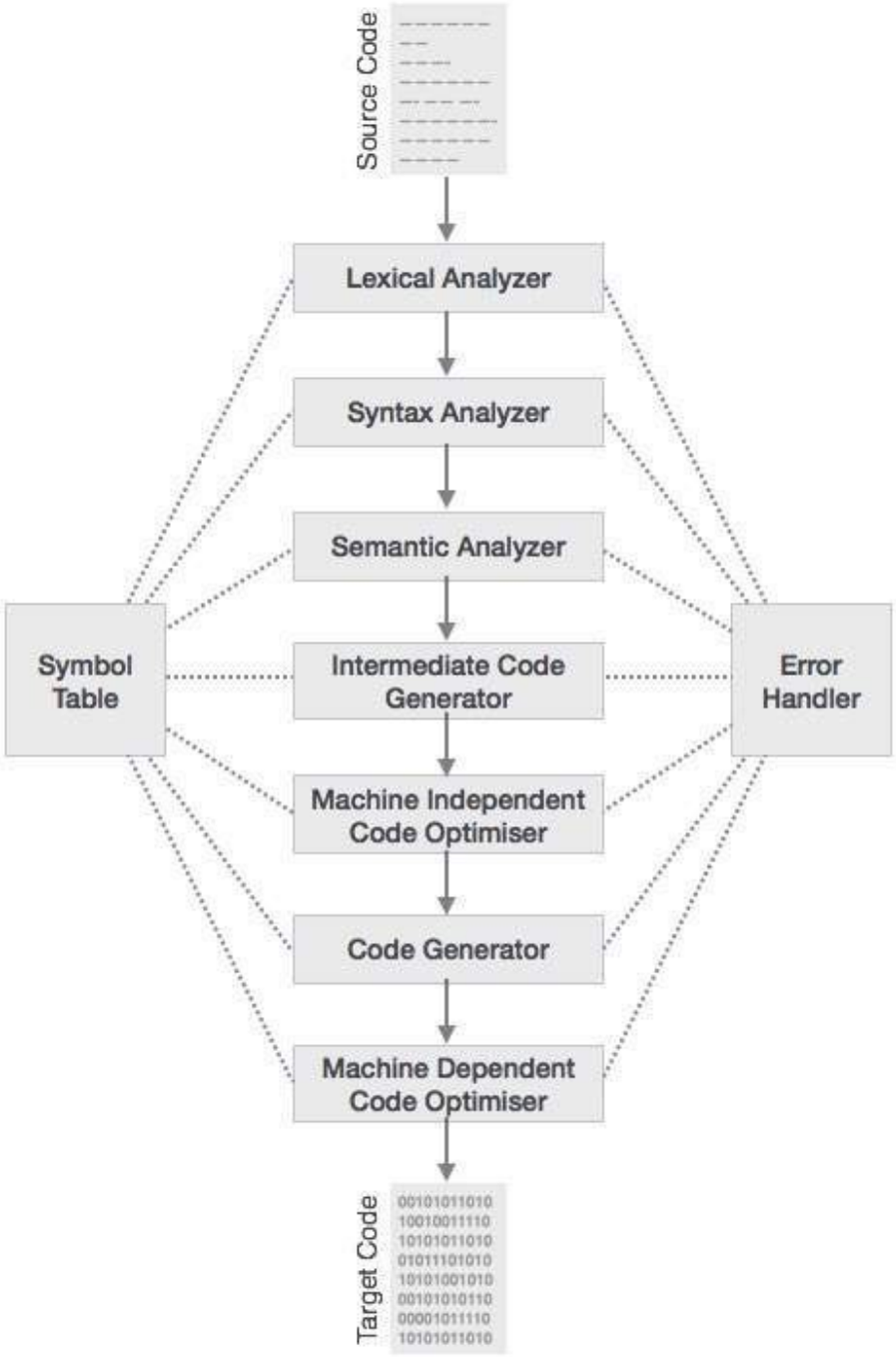
- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

- Ex:

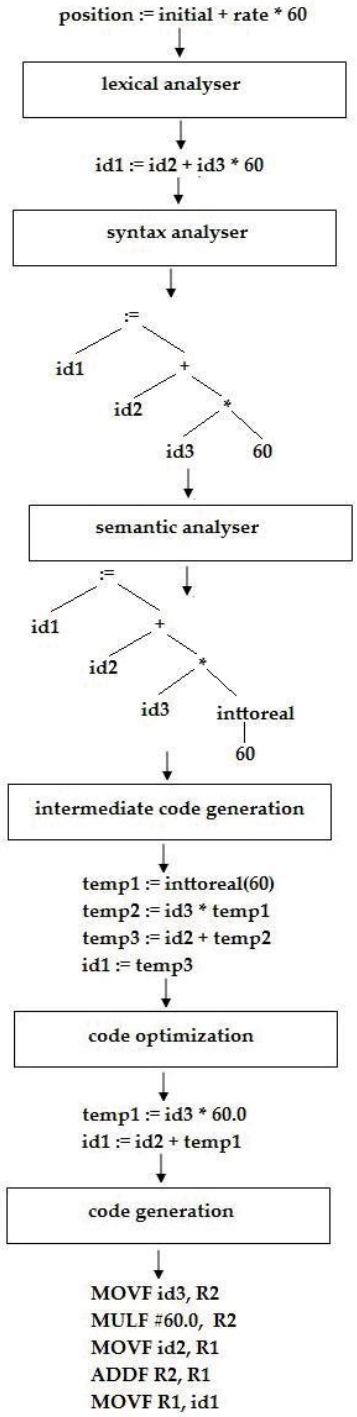
  (assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
MOVE    id2,R1
MULT    id3,R1
ADD     #1,R1
MOVE    R1,id1
```

Phases of Compiler



Example


position := initial + rate * 60
lexical analyser
id1 := id2 + id3 * 60
syntax analyser
:=
id1
+
id2
*
id3
60
semantic analyser
:=
id1
+
id2
*
id3
inttoreal
60
intermediate code generation
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
code optimization
temp1 := id3 * 60.0
id1 := id2 + temp1
code generation
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1