GUDLAVALLERU ENGINEERING COLLEGE

(An Autonomous Institute with Permanent Affiliation to JNTUK, Kakinada)

Seshadri Rao Knowledge Village, Gudlavalleru – 521 356.

Department of Computer Science and Engineering

HANDOUT

on

COMPILER DESIGN

Vision

To be a Centre of Excellence in computer science and engineering education and training to meet the challenging needs of the industry and society.

Mission

- To impart quality education through well-designed curriculum in tune with the growing software needs of the industry.
- To serve our students by inculcating in them problem solving, leadership, teamwork skills and the value of commitment to quality, ethical behaviour & respect for others.
- To foster industry-academia relationship for mutual benefit and growth.

Program Educational Objectives

PEO1: Identify, analyze, formulate and solve Computer Science and Engineering problems both independently and in a team environment by using the appropriate modern tools.

PEO2: Manage software projects with significant technical, legal, ethical, social, environmental and economic considerations.

PEO3: Demonstrate commitment and progress in lifelong learning, professional development, leadership and Communicate effectively with professional clients and the public.

HANDOUT ON COMPILER DESIGN

Class & Sem : III B.Tech-I Semester Year: 2019-20

Branch : CSE Credits: 3

1. Brief history and scope of the subject

The first compiler was written by Grace Hopper, in 1952, for the A-0 System language. The term compiler was coined by Hopper. The A-0 functioned more as a loader or linker than the modern notion of a compiler. The first auto code and its compiler were developed by Alick Glennie in 1952 for the Mark 1 computer at the Manchester and is considered by some to be the first compiled programming language. The FORTRAN team led by John W. Backus at IBM is generally credited as having introduced the first complete compiler, in 1957.

The first ALGOL 58 compiler was completed by the end of 1958 by Friedrich L. Bauer, Hermann Bottenbruch, Heinz Rutishauser, and Klaus Samelson for the Z22 computer. By 1960, an extended Fortran compiler, ALTAC, was available on the Philco 2000, so it is probable that a Fortran program was compiled for both IBM and Philco computer architectures in mid-1960. The first known demonstrated cross-platform high-level language was COBOL. In a demonstration in December 1960, a COBOL program was compiled and executed on both the UNIVAC II and the RCA 501.

The COBOL compiler for the UNIVAC II was probably the first to be written in a high-level language, namely FLOW-MATIC, by a team led by Grace Hopper.

2. Pre-requisites

Student should be familiar with the subject Formal Languages and Automata Theory.

3. Course Objectives

- To familiarize with lexical analyzer and different parsers.
- To introduce various storage allocation strategies, code generation and code optimization techniques.

4. Course Outcomes

Upon successful completion of the course, the students will be able to

- list out compilation process steps of a language.
- use regular languages to identify the tokens of a programming language.
- design a parser to verify the syntax of a programming language.
- compare top down parser with bottom up parser
- create symbol table to access identifier information
- apply code optimization techniques to enhance the efficiency of the intermediate code.
- write a program for the execution of DAG to generate the code.

5. Program Outcomes:

Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis

and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

6. Mapping of Course Outcomes with Program Outcomes:

	1	2	3	4	5	6	7	8	9	10	11	12
CO1		3										
CO2	3	3										
CO3			3									
CO4		3										
CO5		3										
CO6	3				2							
CO7		3										

7. Prescribed Text Books

1. Alfred V Aho, Monical S Lam, Ravi Sethi, Jeffrey D. Ullman, "Compilers, Principles Techniques and Tools", 2nd edition, Pearson.
2. V. Raghavan, Principles of compiler design, 2nd edition, TMH.

8. Reference Text Books

1. Kenneth C Louden, "Compiler construction, Principles and Practice", 1st edition, Cengage.
2. Yunlinsu, "Implementations of Compiler, A new approach to Compilers including the algebraic methods", Springer.
3. Jean-Paul Trembly, Paul G. Sorenson, "The theory and practice of Compiler writing", 1st edition, McGraw-Hill.
4. Nandini Prasad, "Principles of compiler design", 2nd edition, Elsevier.

9. URLs and Other E-Learning Resources

- http://www.nptel.iitm.ac.in/downloads/106108052/
- http://www.cs.ualberta.ca/~amaral/courses/680
- http://www.learnerstv.com

10. Digital Learning Materials:

- http://jntuk-coeerd.in
- http://ocw.mit.edu
- www.diku.dk/~torbenm/Basics/basics_lulu2.pdf

11. Lecture Schedule / Lesson Plan

Topic	No. of Periods	
	Theory	Tutorial
UNIT –I: Lexical Analysis		
Introduction, overview of language processing	1	1
Preprocessors, compiler, assembler, interpreters, linkers and loaders	1	
Structure of a compiler-Analysis-Synthesis model of a compiler	1	
phases of a compiler	1	1
Lexical Analysis – Role of Lexical Analysis	1	
Token, patterns and Lexemes	1	
Regular expressions for reserved words, identifiers, operators	1	
Transition diagram for recognition of tokens, reserved words and identifiers	1	
Lexical analyzer program	1	
Total	9+2(T)	
UNIT – II: Top-down Parsing		
Syntax analysis, role of a parser, classification of parsing techniques	2	1
Brute force approach, Recursive descent parsing	1	
Elimination of ambiguity, elimination of left factoring, elimination of left recursion	2	
First and Follow, LL (1) grammars	1	1
Model of predictive parsing, pre-processing steps for predictive parsing	1	

Construction of predictive parsing table	2	
Non-recursive predictive parsing program	1	
Total	10+2(T)	
UNIT – III: Bottom-up Parsing		
Bottom-up parsing approach, Types of Bottom-up parsers, shift- reduce Parsing	1	1
LR parsers: LR(0) Grammar, LR(0) parser, LR(0) items, SLR parsing table and string checking	3	
LR(1) items, CLR parsing Table and string checking	3	1
LALR Parsing table and string checking	1	
Operator precedence parser	1	
Dangling ELSE Ambiguity, Comparison of all bottom-up parsers with top-down parsers	1	
Total	10+2(T)	
UNIT – IV: Semantic Analysis		
Role of Semantic analysis, SDT	1	1
Evaluation of semantic rules	1	
Symbol Tables- use of symbol tables, contents of symbol table, operations on symbol table	1	
Block structured symbol table	2	
Non block structured symbol table	2	1
Runtime environment-Storage organization-static allocation	1	
Stack allocation	1	
Heap management, Differences between static and dynamic storage organization, heap and stack	1	

allocation		
Total	10+2(T)	
UNIT – V: Intermediate Code Generation		
Intermediate code- Three address code- quadruples and triples	2	1
Abstract syntax trees	1	
Partition into basic blocks	1	
Flow Graph Construction	1	
DAG construction and its applications	1	
Machine independent code optimization: Common sub expression elimination ,Constant folding, copy propagation, dead code elimination	2	
Loop optimization- strength reduction, code motion	1	1
Total	9+2(T)	
UNIT – VI: Code Generation		
Code generation: issues in code generation	1	1
Generic code generation	1	
Code generation from DAG	2	
Machine dependent code optimization : Peephole optimization	1	1
Register allocation and assignment	1	
Total	6+2(T)	
Total No. of Periods:	**54**	**12(T)**

12. Seminar Topics: Nil

COMPILER DESIGN

UNIT-I

Lexical Analysis

Objective:

To identify tokens with lexical analyzer.

Syllabus:

Overview of language processing, preprocessors, compiler, assembler, interpreters, linkers and loaders, phases of a compiler. Lexical Analysis- role of lexical analysis, token, patterns and lexemes, transition diagram for recognition of tokens, reserved words and identifiers.

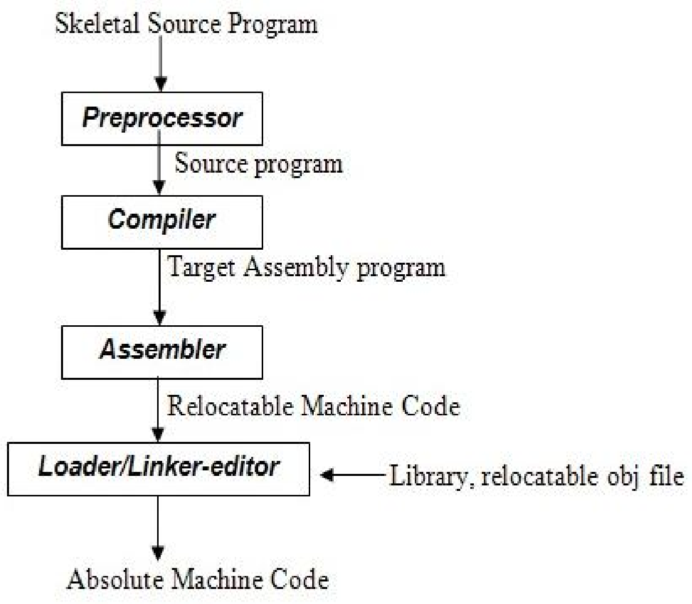
Learning Outcomes:

Students will able to

- explain language processing system.
- identify the differences between compiler and interpreter.
- identify tokens for the programming language constructs.

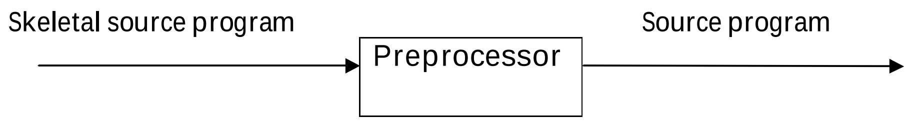
Learning Material

Overview of language-processing system:



Pre-processor:

A pre-processor is a program that processes its input data to produce output that is used as input to another program.



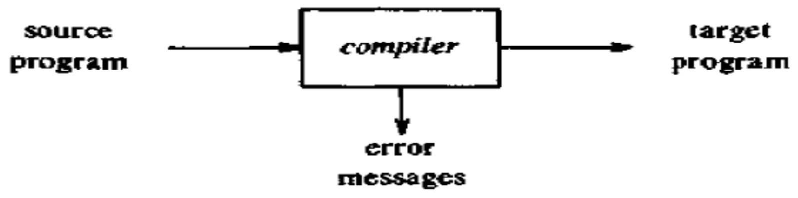
Functions of pre-processor:

1. **Macro processing**: A pre-processor may allow a user to define macros that are short hands for longer constructs.
2. **File inclusion**: A pre-processor may include header files into the program text.
3. **Language Extensions**: These pre-processor attempts to add capabilities to the language by certain amounts to built-in macro.

Compiler:

- A compiler is a computer program that reads a program written in one language-the source language-and translates it into an equivalent program in another language.

- An important part of a compiler is it presents error information to the user.



- Target language may be another programming language ,or the machine language of any any computer between microprocessor or a super computer.
- Compilers are sometimes classified as the following depending on how they have been constructed or on what function they are supposed to perform.

  single-pass
  multi-pass
  load-and-go
  debugging
  optimizing

Assembler:

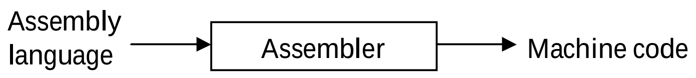
- An assembler is a translator that converts assembly code to machine code.
- Assembly code is a mnemonic version of machine code, in which names are used instead of binary codes for operations, and names are also given to memory addresses.
- A typical sequence of assembly instructions might be

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

- Typically, assemblers make two passes over the assembly file

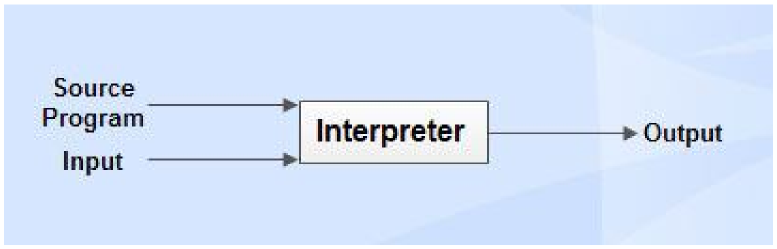
- First pass: reads each line and records labels in a symbol table.

- Second pass: use info in symbol table to produce actual machine code for each line



Interpreter:

- An interpreter is another common kind of language processor, instead of producing a target program as a translation; it appears directly to execute the operations specified in the source program on inputs supplied by the user.
- It provides better debugging environment.
- An interpreter can usually give better error diagnostic than a compiler.



- Java language processors combine compilation and interpretation.
- A java source program may first be compiled into an intermediate code called byte codes.
- These byte codes are then interpreted by a virtual machine.
- Some interpreted languages are

  BASIC, LISP, Python etc.,

Linker:

- A linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.
- The Linker resolves external memory addresses, where the code in one file may refer to code in another file.
- Link editors are commonly known as linkers. The compiler automatically invokes the linker as the last step in compiling a

program. The linker inserts code (or maps in shared libraries) to resolve program library references, and/or combines object modules into an executable image suitable for loading into memory.

- Static linking is the result of the linker copying all library routines used in the program into the executable image. This may require more disk space and memory, but is both faster and more portable, since it does not require the presence of the library on the system where it is run.
- Dynamic linking is accomplished by placing the name of a sharable library in the executable image. Actual linking with the library routines does not occur until the image is run, when both the executable and the library are placed in memory. An advantage of dynamic linking is that multiple programs can share a single copy of the library.
- If linker does not find a library of a function then it informs to compiler and then compiler generates an error.
- Usually a longer program is divided into smaller subprograms called modules. And these modules must be combined to execute the program. The process of combining the modules is done by the linker.
- Linker can convert machine understandable format into Operating system understandable format.

Loader:

- The loader puts together the entire executable object files into memory for execution.
- Some operating systems need relocating loaders, which adjust addresses (pointers).
- The operating systems that need relocating loaders are those in which a program is not always loaded into the same location in the

address space and in which pointers are absolute addresses rather than offsets from the program's base address.

- Linking and loading provides 4 functions
  1. Allocation
  2. Linking
  3. Relocation
  4. Loading

Differences between a Compiler and Interpreter:

Compiler	**Interpreter**
Compiler Scans the entire program first and then translates it into an equivalent machine code.	Interpreter scans the program line by line.
Compiled programs take more memory because the entire program has to reside in memory.	Interpreted programs take less memory because at a time a line of code will reside in memory.
A compiled language is more difficult to debug	Debugging is easy because interpreter stops and reports errors as it encounter them.
Execution time is less	Execution time is more.
Code optimization is possible	Code optimization is not possible
Examples: C,C++	Examples: LISP, Python

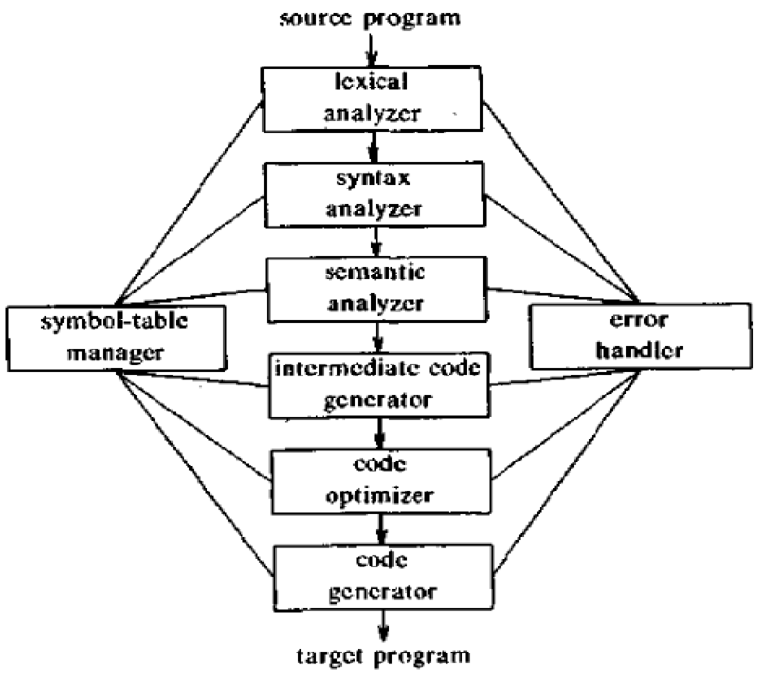
Analysis and Synthesis model of Compiler:

- The Analysis part breaks up the source program into constituent pieces and creates an intermediate representation of source program.
- In compiling ,analysis consists of three phases:

i. Linear analysis or Lexical analysis, in which the stream of characters making up the source program is read from left to right and grouped into tokens that are sequence of characters having a collective meaning.
ii. Hierarchical analysis, in which characters or tokens are grouped hierarchically into nested collections with collective meaning.
iii. Semantic analysis, in which certain checks are performed to ensure that the components of a program fit together meaningfully.

- The Synthesis part constructs the desired target program from the intermediate representation.
- In compiling ,synthesis consists of 2 phases:
  i. Code optimization
  ii. Code Generation
- The analysis part is often called the front of compiler and synthesis part is called back end of compiler.

Phases of a compiler:

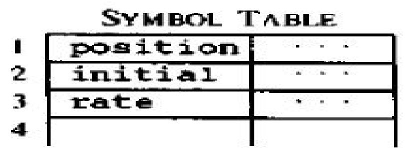


Symbol Table Management:

- An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier.
- Symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.

- Various attributes of each identifier are
  - ✓ its type, (by semantic and intermediate code)
  - ✓ its scope, (by semantic and intermediate code)
  - ✓ storage allocated for an identifier, (by code generation)
  - ✓ in case of procedure names such as number of arguments and its type for procedure, the type returned
- For example for the statement below , the symbol table entries are shown below

position := initial + rate * 60


SYMBOL TABLE

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

Error handler:

- Each phase encounters errors.
- After detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- Lexical analysis phase can detect errors that do not form any token of the language.
- Syntax analysis phase can detect the token stream that violates the (structure (or) syntax rules of the language.
- Semantic analysis phase detects the constructs that have no meaning to the operation involved.

Lexical analysis:

- Lexical analysis is the first phase of a compiler.
- Lexical analyzer is also called Scanner.
- The lexical analysis phase reads the characters from the source program and group them into stream of tokens in which each token represents a logically cohesive sequence of characters, such as an identifier, a keyword (if,while,etc.,),a punctuation character etc.,
- For example in the statement position := initial + rate * 60 would be grouped into the following tokens:

identifier 1	: position.
assignment symbol	: =.
identifier 2	: initial.
plus sign.	: +
identifier 3	: rate.
multiplication sign.	: *
number	: 60

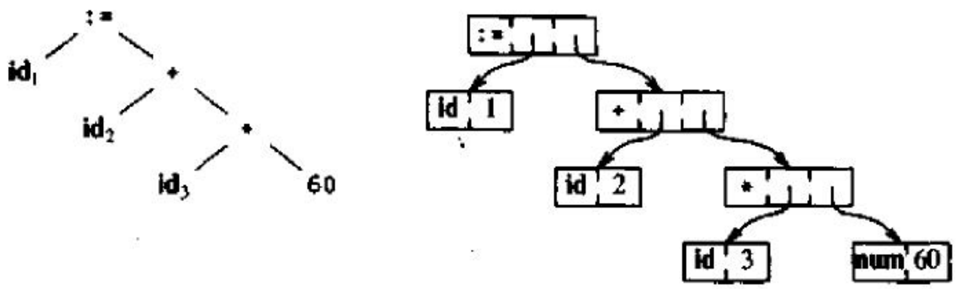
$id_1 = id_2 + id_3 * 60$

- The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

Syntax Analysis Phase:

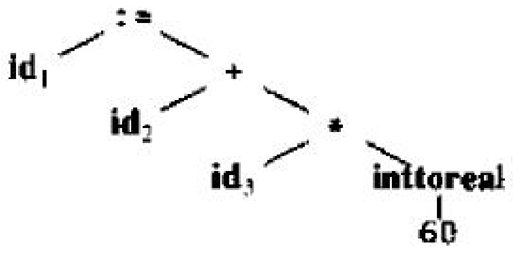
- Syntax analysis imposes a hierarchical structure on the token stream. This hierarchical structure is called syntax tree.
- Syntax analyzer is also called Parser.
- The syntax analyzer basically checks the syntax of the language.
- A syntax analyzer takes the token from the lexical analyzer and groups them in such a way that some programming structure can be recognized.
- A syntax tree has an interior node is a record with a field for the operator and two fields containing pointers to the records for the left and right children.
- A leaf is a record with two or more fields, one to identify the token at the leaf, and the other to record information about the token.

- Syntax tree for the example statement is



Semantic analysis:

- This phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
- It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements
- An important component of semantic analysis is type checking.
- Syntax trees after semantic analysis phase for the example statement is



Intermediate code generation:

- Some compilers generate a explicit intermediate representation of the source program after the syntax and semantic analysis.
- The intermediate representation is a program for an abstract machine
- The intermediate representation should have two important properties: It should be easy to produce, and easy to translate into target program.

- Intermediate representation can have a variety of forms.
- One of the intermediate form is: three address code; which is like the assembly language for a machine in which every location can act like a register.
- Three address code consists of a sequence of instructions, each of which has at most three operands.
- Three address code after intermediate code generation phase for the example statement is

  temp1 := inttoreal (60)

  temp2 := id3 * temp1

  temp3 := id2 + temp2

  id1 := temp3

Code optimization:

- Code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result.
- Optimized Three address code after Code Optimization phase for the example statement is

  temp1 := id3 * 60.0

  id1 := id2 + temp1

Code generation:

- The final phase of the compiler is the generation of target code, consisting of relocatable machine code or assembly code.
- Memory locations are selected for each of the variables used by the program.
- Then, each intermediate instruction is translated into a sequence of machine instructions that perform the same task.
- A crucial aspect is the assignment of variables to registers.

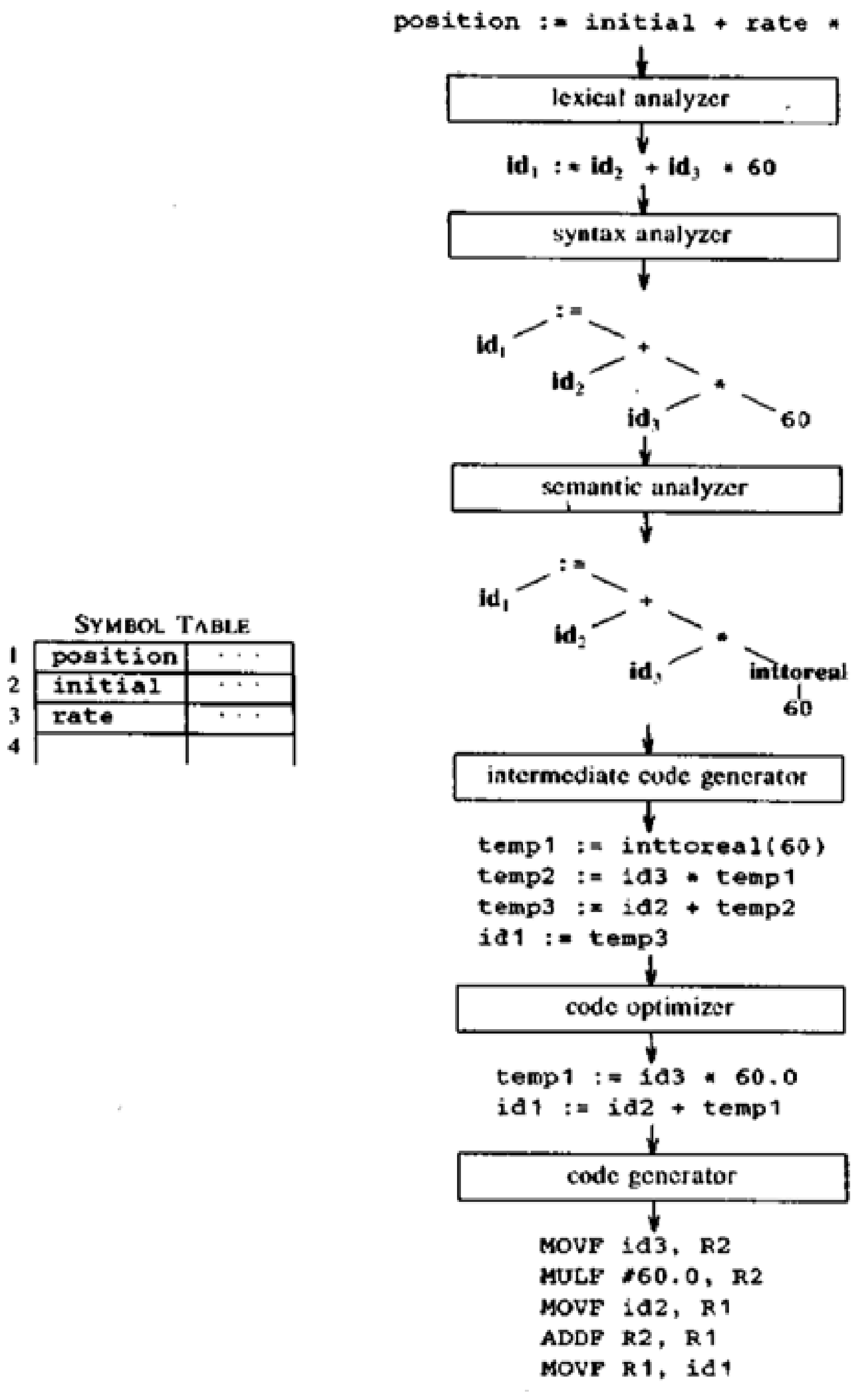
  MOVF id3, R2

  MULF #60.0, R2

MOVF id2, R1

ADDF R2, R1

MOVF R1, id1



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

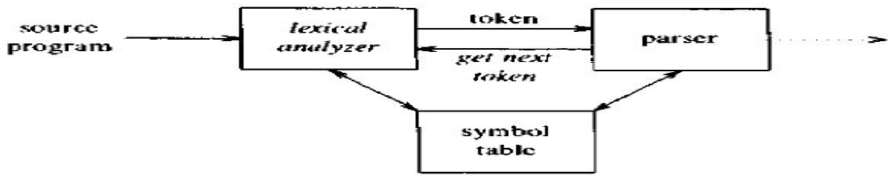
```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

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

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

Role of Lexical Analysis:

- The lexical analyzer is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.



- Another task of lexical analyzer is stripping out from the source program comments and white space in the form of blank and tab and newline characters.
- Correlating error messages from the compiler with the source program.
- The lexical analyzer may keep track of the number of newline characters seen, so that line number can be associated with an error message.
- In some compilers, the lexical analyzer is in charge of making a copy of the source program with the error messages marked in it.
- If the lexical analyzer finds a token invalid, it generates an error.
- The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.
- The lexical analyzer collects information about tokens into their associated attributes.

Token:

- Token is a sequence of characters that can be treated as a single logical entity.
- Typical tokens are identifiers, keywords, operators, special symbols, constants.

Attributes for Tokens:

- A token has only a single attribute – a pointer to the symbol-table entry in which the information about the token is kept.
- The token names and associated attribute values for the statement
- E = M * C + 2 are written below as a sequence of pairs.

    <id, pointer to symbol-table entry for E>
    <assign_op>
    <id, pointer to symbol-table entry for M>
    <mult_op>
    <id, pointer to symbol-table entry for C>
    <add_op>
    <number, integer value 2>

Recognition of Tokens:

- In many programming languages, the following classes cover most or all of the tokens:
    1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
    2. Tokens for the operators, either individually or in classes such as the token comparison.
    3. One token representing all identifiers.
    4. One or more tokens representing constants, such as numbers and literal
    5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Example:

```
stmt → if expr then stmt
        | if expr then stmt else stmt
        | ε
expr → term relop term
        | trem
term → id
```

| num

The terminals if, then, else, relop, id, num generates set of strings given by the following regular definitions

if → if

then → then

else → else

relop → < | <= | = | <> | > | >=

id → letter(letter|digit)*

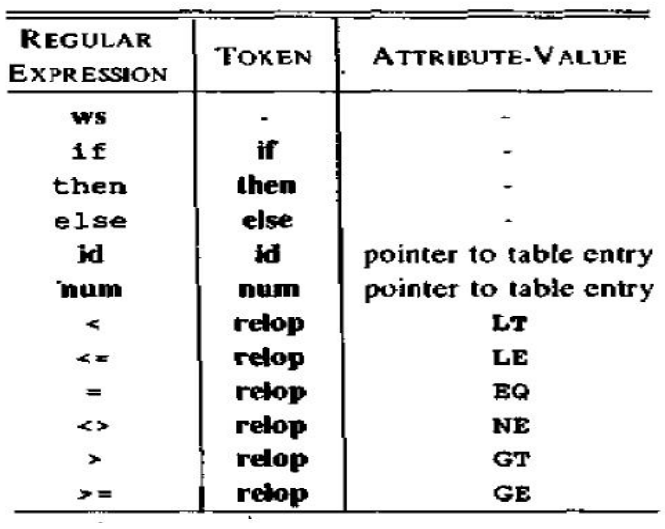
num → $\text{digit}^{+}(.\text{digit}^{+})?(E(+|-)?\text{digit}^{+})?$

Lexeme:

- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.
- Each lexeme corresponds to a token.

Pattern:

- A rule that describes the set of strings associated to a token.
- Expressed as a regular expression and describing how a particular token can be formed. For example, [A-Za-z][A-Za-z_0-9] *



Regular Expression	Token	Attribute-Value
ws	-	-
if	if	-
then	then	-
else	else	-
id	id	pointer to table entry
num	num	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Regular-expression Patterns for Tokens

Regular expressions:

- The languages accepted by finite automata are easily described by simple expressions called Regular Expressions.

- Let Σ be an alphabet. The regular expressions over Σ and the sets that they denote are defined recursively as follows.
  - i. Ø is a regular expression and denotes the empty set.
  - ii. ε is a regular expression and denotes the set { ε }.
  - iii. For each a in Σ, a is a regular expression and denotes the set {a}.
  - iv. If r and s are regular expressions denoting the languages R and S, respectively, then (r + s), (rs), and (r*) are regular expressions that denote the sets R U S, RS, and R*, respectively.
- Example: Regular expression for pascal identifier

  letter (letter |digit)*

Regular Definition:

- If Σ is a n alphabet of basic symbols ,then a regular definition is a sequence of definitions of the form

  d_1 -> r_1

  d_2 -> r_2

  ...

  dn-> rn

  where each di is a distinct name and each ri is a regular expression over the symbols in Σ ∪ {d1,d2,.... dn},
- Regular expressions are an important notation for specifying lexeme patterns.

Examples:

i. Regular expression for identifiers in PASCAL.

letter → A | B | ... | Z | a | b | ... | z

digit → 0 |1 | ... | 9

id → letter(letter|digit)*

ii. Regular expression for white space in PASCAL or in C.

delim → blank | tab | newline

ws → delim$^+$

iii. Regular expression for unsigned numbers in PASCAL or in C such as 5280, 39.37, 6.336E4 or 1.894E-4

digit → 0 |1 | … | 9

digits → digit digit*

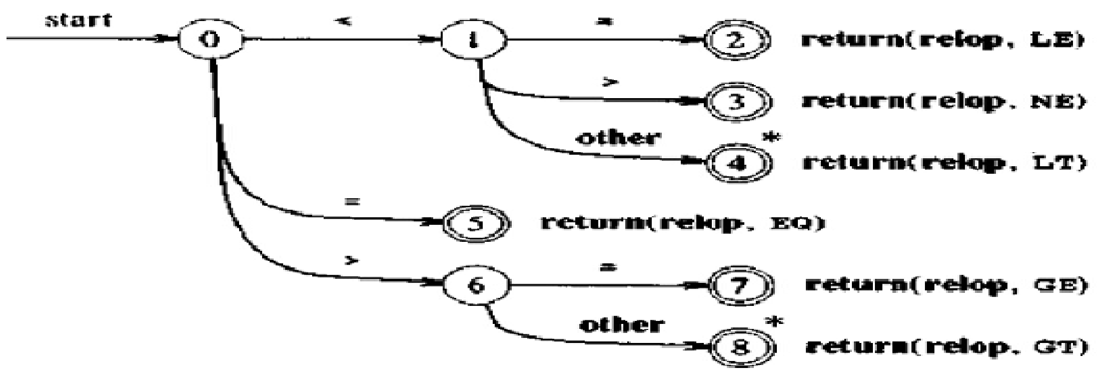
optional_fraction → .digits | ε

optional_exponent → E(+ | - | ε) digits) | ε

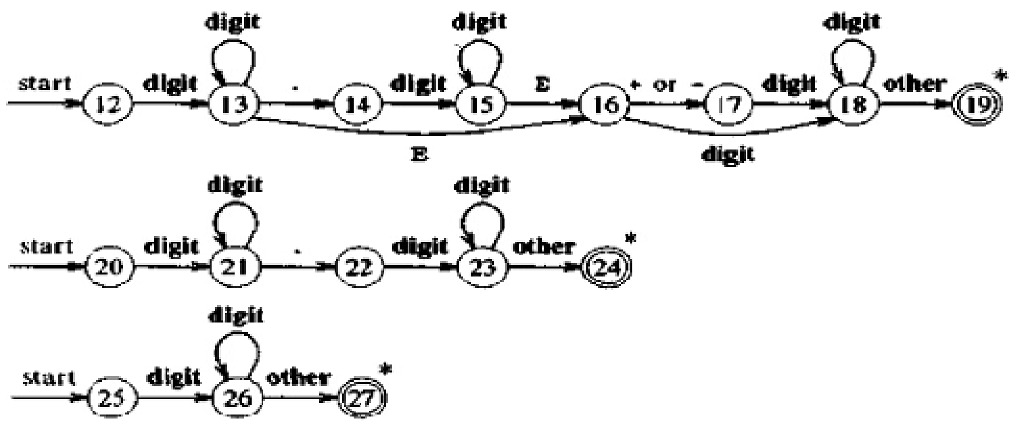
num → digits optional_fraction optional_exponent

Transition Diagrams:

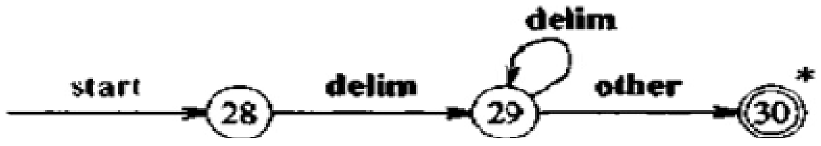
i. Transition diagrams for Relational operators in PASCAL or in C



ii. Transition diagrams for unsigned numbers in PASCAL or in C



iii. Transition diagrams for white space in PASCAL or in C



UNIT-I

Assignment-Cum-Tutorial Questions

SECTION-A

Objective Questions

1. The output of a pre-processor is [ ]
   a) absolute machine language program
   b) relocatable machine language program
   c) assembly language program
   d) a high level language program

2. A compiler running on computers with small memory would normally be [ ]
   a) a multi-pass compiler b) single pass compiler
   c) a compiler with less number of phases d) none of these

3. In a compiler, grouping of characters into tokens is done by __________.

4. A computer program that translates a program statement by statement into machine language is called a_____________.

5. Front end of compiler does not include the phase [ ]
a) semantic analysis b) intermediate code generation
c) code optimization d) lexical analysis

6. Back end of compiler includes those phases that depend on [ ]
a) target machine b) source language
c) both a and b d) None of the above

7. Assembly language_________ [ ]
a) is usually the primary user interface
b) requires fixed format commands

c) is a mnemonic form of machine language

d) is quite different from the SCL interpreter

8. Relocating loaders perform four functions in which order? [ ]

a) Allocation, linking, relocation, loading

b) Loading, linking, relocation, allocation

c) Allocation, loading, relocation, linking

d) None of the above

9. ____________is a sequence of characters in the source program that is matched to some pattern for a token.

10. r+ represents ____________.

11. Which of the following phase of compilation process is an optional phase? [ ]

a) lexical analysis phase b) Syntax analysis phase

c) Code optimization d) Code generation

12. In some programming languages, an identifier is permitted to be a letter followed by a number of letter or digits. If L and D denote the set of letters and digits respectively, which of the following expression denotes an identifier? [ ]

a) (L U D)* b) L(L U D)* c) [L U D]* d) L[L U D]*

13. Which of the following is the name of the data structure in a compiler that is responsible for managing information about variable and their attributes?

a) Symbol table b) Attribute grammar [ ]

c) Stack d) syntax tree

14. Match the following [ ]

LIST-1	LIST-2
A. pre-processor	1) Resolving external reference
B. Assembler	2) loading the program
C. Loader	3) producing relocatable machine code
D. Linker	4) allow user to define shorthand for longer construct

	A	B	C	D
a)	4	3	2	1
b)	3	4	1	2
c)	4	3	1	2
d)	4	2	3	1

SECTION-B

SUBJECTIVE QUESTIONS

1. What is the role of Lexical analyzer in a compiler?
2. Differentiate between Compiler and Interpreter.
3. Construct the transition diagram for relational operators and identifiers in 'C'.
4. Explain the various phases of a compiler. Show the translations for an assignment statement position=initial+rate *60, clearly indicate the output of each phase.
5. Define Regular expression with notation.
6. Identify the lexemes that make up the tokens in the following program segment.Indicate corresponding token and pattern.

```
void swap(int i, int j)
{
int t;
t=i;
i=j;
j=t;
}
```

7. Define compiler? List out its functions?
8. What is the role of Lexical analyzer in a compiler?
9. Explain the reasons why lexical analysis is separated from syntax analysis.
10. Define lexeme, token, pattern.
11. Draw a block diagram of phases of a compiler and indicate the main functions of each phase.
12. What is LEX? Give LEX specification to identify identifiers & keywords of C language.
13. Discuss about functions of pre-processor.
14. Consider the following C statement and determine the type of compiler error:

int *p, *a[][3]; float 34var;

i) Syntax error ii) lexical error iii) semantic error iv) linker error

SECTION-C

GATE QUESTIONS

1. Which one of the following statements is FALSE? **[GATE CS 2018]**

a) Context-free grammar can be used to specify both lexical and syntax rules
b) Type checking is done before parsing.
c) High-level language programs can be translated to different Intermediate Representations.
d) Arguments to a function can be passed using the program stack.

2. In a compiler, keywords of a language are recognized during **[GATE CS 2011]**

a) parsing of the program b) The code generation
c) the lexical analysis of the program d) dataflow analysis

3. The lexical analysis for a modern computer language such as Java needs the power of which one of the following machine models in a necessary and sufficient sense? **[GATE CS 2011]**

a) finite state automata b) deterministic pushdown automata

c) non- deterministic PDA d) Turing machine

4. The number of tokens in the following C statement is

printf("i=%d, &i=%x", i, &i); **[GATE 2000]**

a) 3 b) 26 c) 10 d) 21