SPCC University paper May-18

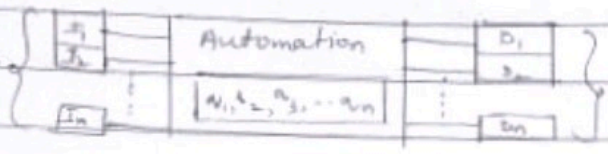
Q.1)

a) Differentiate between System Software and Application Software.

System Software	Application Software
1) System software is any computer program that manages and controls computer hardware so that application software can perform a task	1) Application software are programs that enable the end-user to perform specific productive tasks, such as word processing or image manipulation
2) The function of system software is to manage the resources of computer, automate its operation and facilitate program development. It is generally provided by the computer manufacturer or a specialized programming firm	2) Application software are designed to perform specific data processing or computational task for the user. These programs are specifically designed to meet end-user requirements
3) System software is the operating system which handles devices etc	3) Application software is third party software that seeks help from the system software.
4) System software or operating system is the software that allows the computer to boot.	4) Application Software are programs that run inside of or on top of the Operating System and allows you to do things like email or word processing
5) Example: operating systems such as Microsoft Windows, Mac OS X or Linux, compilers, linkers, etc.	5) Example: Microsoft word, Oracle, Paint brush, Spread sheets, anti-virus tools etc.

b) Q 2) Explain the role of finite automata in compiler theory?

1) An automaton is a self operating machine. The word is sometimes used to describe a robot, more specifically an autonomous robot. Automaton is more often used to describe non-electronic moving machines, especially those that have been made to resemble human being or animal actions.



2) Characteristics of Automata

1) Input - At each of the discrete instants of time input values $I_1, I_2 \cdots I_n$ are applied to the input side of the model.

2) Output - $O_1, O_2 \cdots O_m$ are the output of the model.

3) States - At any time automata is anyone of the state q_0, q_1, q_2, q_3

4) Transition relation - The next state of automata is determined by present state of the automata.

5) output relation - output of automata is depend only on state or both input and state

3) Finite automata is a machine with a finite number of states in which machine can perform. There is a distinctive start state, from which the machine starts. Finite Automata is a recognized for a language that takes as input a string x and answer "yes" if x is a sentence of the language and 'no' otherwise. We compile a regular expression into a recogniser by constructing a generalized transition diagram called finite automata

c) Explain the functions of a Loader?

The fundamental task of a loader is to
a) bring an object program into memory and
b) start its execution.

An object program contains translated instructions and data from the source program. It also specifies addresses in memory where these items are to be loaded.

To execute a program a loader performs the following four functions
1) Allocation: It is used to allocate space in memory for the object program. Translators cannot allocate space since overlap may occur or large wastage of memory takes place.
2) Linking - It combines two or more separate object programs and resolve symbolic references between object decks. It also supplies the information needed to allow to reference between them.
3) Relocation: It modifies the object program so that it can be loaded at an address different from the location originally specified and adjusts all address dependent location.
4) Loading: Physically it places the machine instructions and data into the memory for the execution.

d) Compare Compilers and interpreters?

Compiler	Interpreters
1) It compiles the whole program at one time directly into machine	1) It translates the high level language into intermediate form line-by-line.
2) Compiled program are faster than interpreted program	2) Interpreters are slower in comparison to compiler
3) More Compact compiled	3) less Compact
4) Interpreted code is good for application	4) Interpreted code is good for simple applications
5) The disadvantage are that we cannot change the program going back to the original source code	5) The disadvantage is that every line has to be translated every time, it is executed.
6) Example of compilers visual Basic C, C++, C#, COBOL etc	6) Example of interpreters are Java Script, LISP, FORTH, BASIC

Q.2] a) With reference to assembler, explain the following tables with suitable example.

i) POT

1. POT is fixed length table i.e the contents of these tables are not filled in or altered during the assembly process.
2. Example:-

POT (Pseudo-Op-Table) for pass 1.

Pseudo-Opcode (8-bytes) (character)	Address of routine to process Pseudo-Opcode (3-bytes = 24 bit address)
"DROPbb"	PIDROP
"ENDbb"	PIEND
"EQUbb"	PIEQU
"START"	PISTART
"USING"	PIUSING

↳ These are presumably lables of routines in pass 1, the table will actually contains the physical addresss.

ii) MOT

1. MOT (Machine Opcode table) is a fixed length table i.e the contents of these tables are not filled in or altered during the assembly process.

2. Example:-

MOT for pass 1 and pass 2

← 6-bytes per entry →

Mnemonic op-code (4-bytes) (characters)	Binary op-code (1-byte) (hexadecimal)	Instruction length (2-bits) (binary)	Instruction format (3-bits) (binary)	Not used in this design (3-bits)
"Abbb"	5A	10	001	
"AHbb"	4A	10	001	
"Albb"	5E	10	001	
"ALRb"	1E	01	000	
"ARbb"	1A	01	000	
...	...	...	...	
"MVCb"	D2	11	100	

b~ represents the character "blank"

3) ST

1. ST (Symbol Table) includes for each entry:
   i) Name of the symbol
   ii) Symbols assembly time value
   iii) Length (in bytes) and
   iv) Relative location indicator (R/A)
2. Operand can be either immediate (typically one byte values coded in the instruction itself) or the address of the data located elsewhere.

Symbol (8-byte) (characters)	Value (4-bytes) (hexadecimal)	length (1-byte) (hexadecimal)	Relocation (1-byte) (character)
"JOHNbbbb"	0000	01	"R"
"FOURbbbb"	000C	04	"R"
"FIVEbbbb"	0010	04	"R"
"TEMPbbbb"	0014	04	"R"

4. LT

1. LT (Litral table) includes for each entry
   i) Name of litral
   ii) Litral assembly time value
   iii) Length (in bytes) and
   iv) Relative location Indicator (RIA)
2. Litral is equivalent
2. Example:-

Name of Litral (1-byte)	Value of litral (4-byte)	Length of litral (1-byte)	Relative location Indicator (1-byte)

Q.2b) Explain different code optimization techniques in compiler design.

Ans 1. Important code optimization techniqhis and their types are explained below as:

1. Function Preserving Transformation
It is majorly of four type of transformation. They are:
a) Compile Time Evaluation
The two possible ways are:
i) Constant Folding
Evaluation of an expression with constant operands to replace the expression with single value.
ex:- pi = 22/7 is replaced as pi = 3.142
ii) Constant Propagation

Replace a variable with constant which has been assigned to it earlier.
Ex:-
area = pi*r**2 → area = 3.142*r**2.

b) Common Sub-Expression Elimination
The expression that has been already computed before and appears again in the code for computation is called as Common Sub-Expression Elimination.

c) Dead Code Elimination
Dead code is portion of program which will not be executed in any path of program. If an instruction's result is never used in the program, the instruction is considered to be "dead" and can be removed from the instruction stream.

d) Copy Propogation
This optimization is similar to constant propogation, but generalized to non-constant values. Copy propogation is the process of removing them and replacing the copies generate by the compilers in an intermediate form. It often reveals dead-code.
In the optimized version of the code, we eliminated the unnecessary copies and propogated the original variable

2. Loop Optimization.
The various types are as follows:
1) Code Motion
In this, the code moves from one part of the program to other without modifying the algorithm
The two advantages of using it are as
1. It reduce the size of the program
2. It reduces the execution frequency of the code subjected to the movement.
2) Strength Reduction
Operator strength reduction replaces an operator by a "less expensive one.
Typically strength reduction occurs in address calculation of array references
3) ~~Induction variable strength Reduction~~
3) Loop Fission
Loop Fission tries to break a loop into multiple loops over the same index range but each taking only a part of the loop's body
It improves the locality of reference.
4) Loop Unrolling
It duplicates the body of the loop multiple times, in order to decrease the number of times the loop conditions needs to be tested.

3. Reducible Flow Graph
A flow graph is reducible when the edges can be partitioned into forward edges and back edges. With the help of duplication code irreducible flow graph can always be converted into reducible flow graph

4. Global Optimizations.

In order to perform the global optimization, the program is to be represented in the form of program flow graph. Two types are A) Control flow analysis B) Data flow analysis.

A) Control Flow

It determines information concerning the structure of the program like the presence of loops, nodes visited before the execution reaches a specific node.

B) Data Flow.

It determines information regarding the data flow in a program like how data items are assigned and referenced in a program, what are the values which are available when program execution reaches a specific statement of the program.

Q3. a) Explain the different issues in code generation?

① Input data for code generator
Below mentioned are the following formats used as input for code generator
i) Three address code (quadruple, triples, indirect triples)
ii) Virtual machine representations (byte-code, stack machine)
iii) Linear representation (postfix, infix, prefix)
iv) Graphical representation (syntax tree, DAG, parse tree, syntax directed translation)

② Target Program
- Knowledge of machine architecture and instruction set are the pre-requisite for the design of good code generator.
- Proper selection of machine architecture helps in producing absolute machine language program and re-locatable machine-language program.

③ Instruction selection
The complexity of mapping IR program into code sequence depends on:
- Level of intermediate representation (High-level or low-level)
- Type & nature of instruction set

④ Register Allocation & Assignment
Selection of set of variables that will reside in register at each point and choosing specific register helps in faster execution of

data.

(4) Evaluation Order
Listing the order in which computations are performed affect the efficiency of larger code.

Q3. b Explain working of direct linking loader with example, showing entries in different databases built by DLL.

→ Direct linking loader uses four types of records in the object file they are as ESD, TXT, RLD, END.

(1) External Symbol Directory (ESD)
It contains information about all symbols that are mentioned in the program but that may be referenced elsewhere

(2) Text record (TXT)
It contains the information about the actual object code translated version of the source program.

(3) Relocation & Linkage Directory (RLD)
RLD cards are used to store those location and addresses on which the program content is dependent.

(4) END Record

It specifies the end of the object file & starting address for execution if the executed routine is in the main program.

→ Example program

Some	Relative		Sample program
1	0	LOOP1	START
2			ENTRY LOSTART0, LOSTART1
3			EXTRN LISTART0, LISTART1
4	8	LOSTART0	
5	12	LOSTART1	
6	16		DC B (LOSTART0)
7	20		DC B(LOSTART1 + 10)
8	24		DC B(LOSTART1 - LOSTART1)
9	28		DC B(LISTART1)
10	32		DC B(LISTART1 + LISTART1)
11			END
12	0	LISTART1	START
13			ENTRY LISTART0
14			EXTRN LOSTART0, LOSTART1
15	4	LISTART0	
16	14	DC	A (LOSTART0)
17	8	DC	A (LOSTART1 + 10)
18	22	DC	A (LOSTART1 - LISTART1)
19	END		

* SAMPLE PROGRAM FOR LOOP1 & LISTART1

25 8

— 2 Addrs required for DLL

① Pass 1

- Object files
- Initial Program Load Addrs (IPLA)
- Program Load Addrs (PLA)
- GEST
- Output is used as input in pass 2
- Load map for external symbols

② Pass 2

- Output of pass 1 as input
- IPLA from pass 1
- Program load addrs words
- GEST
- ESD & RLD cards relation given by local External Symbol array

ESD Records

Source card Reference	Variable Name	ESD Type	ESD ID	Relative address	Length
1	LOOP1	SD	01	0	28 (0-31)
2	LOSTART0	LD	02	08	
2	LOSTART1	LD	02	12	
3	LISTART1	ER	03	-	
3	LISTART0	ER	03	-	

* Object deck program for LOOP1

where
SD - Segment Definition (01)
LD - Local Definition (02)
ER - External Reference (02)

TXT Records

Source card Reference	Relative Address	Content	What records do
1	16-19	08	
7	20-23	32	12+10=22
8	24-27	02	12-08-02=02
9	28-31	0	NOW moves to LOOP1
10	32-35	75	(5) LISTART0, LISTART1 NOT KNOWN TO ASSE

RLD Words

Variable Name	Source Card Reference	ESD ID	Length (bytes)	Arithmetic Operation	Relative address
LOOPBACK	6	02	4	+	16
LOSTART2	7	02	4	+	20
LISTART2	9	03	4	+	24
LISTART1	10	03	4	+	28
LISTART0	11	03	4	+	28

Global External Symbol Table (GESYT)

It is used to store the external symbol & their corresponding core addresses in table which are defined by SD or LD entry on an ESD card.

← 12 byte per entry →

External symbol	Assigned core address
"LOOP1bb"	100
"LOSTART0bb"	124
"LOSTART1bb"	130
"LISTART1bb"	134
"LISTART0b"	138

Local External Symbol Table (LESYT)

		Assigned core address of corresponding symbol
	1	100
Ordinal	2	120
No.	3	130
identifier	4	-

Q.4

a) Construct predictive parsing table for following grammar.

$E \rightarrow TE'$

$E' \rightarrow +TE' / \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \epsilon$

$F \rightarrow (E) / id$

Production	First	Follow
$E \rightarrow TE'$	{(, id}	{), $}
$E' \rightarrow +TE' / \epsilon$	{+, ε}	{), $}
$T \rightarrow FT'$	{(, id}	{+,), $}
$T' \rightarrow *FT' / \epsilon$	{*, ε}	{+,), $}
$F \rightarrow (E) / id$	{(, id}	{*, +,), $}

First (E) = First (T) - ①

First (T) = First (F) - ②

First (F) = {(, id} - ③

from ①, ② & ③

First (E) = {(, id}

First (T) = {(, id}

First (E') = {+, ε}

First (T') = {*, ε}

Follow (E) = {), $}

Follow (E') = Follow (E) = {), $}

Follow (T) = {

b) Explain different error recovery technique?

The program errors are detected and reported by parser. The parser handles the errors encountered and the rest of the input is parsed. The errors may be encountered at various stages of the compilation process. At various stages, the following kinds of errors occur.

a) lexical - name of some identifier typed incorrectly
b) Syntactical - missing semicolon or unbalanced parantheses
c) Semantical - Incompatible value assignment.
d) Logical - code not reachable, infinite loop.

In order to deal with the errors in the code, the following are the four common error recovery strategies.

1) Panic Mode

When an error is encountered anywhere in the statement the rest of the statement is ignored by not processing the input from the erroneous input to delimiter such as semi-colon. This mode prevents the parser from developing infinite loops and is considered as the easiest way for recovery of the errors.

2) Statement mode

when an error is encountered by the parser corrective measure are taken which facilitate the parser to parse the rest of the inputs of the statements. For example inserting a missing semicolon, replace comma with a semicolon etc. Here more attention is required as one wrong correction may lead to infinite loop.

follow(E) = {), $}

follow(E') = follow(E) [From production E → TE']

= follow(E') = {), $}

follow(T) = First of E' [From productions E' → +TE' | ∈
E → TE']

∴ follow(T) = {+,), $}

follow(T') = Follow(T) [From productions T → FT'
T' → *FT' | ∈]

follow(T') = {+,), $}

follow of(F) = First(T') [From productions T → FT'
T' → *FT' | ∈]

follow of(F) = { *, +,), $}

Parsing Table:

	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ∈	E' → ∈
T	T → FT'			T → FT'		
T'		T → ∈	T' → *FT		T' → ∈	T' → ∈
F → id	F → id			F → (E)		

3) Error productions

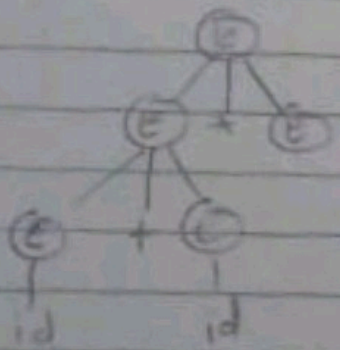
The compiler designers sometimes know that certain errors may occur in the code. In such instances, augmented grammar is created by the designers as productions which generate constructs during the time of occurance of errors.

4) Global correction

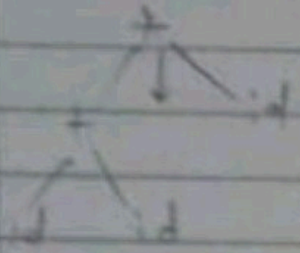
The program in hand is considered as a whole and the intended program is figured out and the closest match for the same is matched, which is error-free. When a erroneous input x is fed, it creates a parse tree for some closest error-free statement y. This calculates the person is making minimal changes to the source code.

c) Abstract Syntax Tree

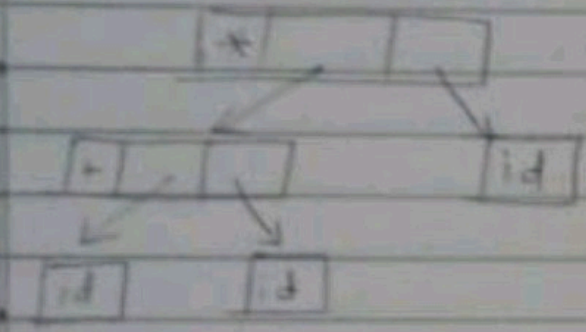
The representation of the parse tree is not easily parsed by the compiler as they contain more details. For instance, the following parse tree is considered as an example



It is observed that the parent nodes have a child leaf node with. Reading it to the next phase, this information can be eliminated. The hiding of extra information results in a tree as shown below



Abstract tree can be represented as

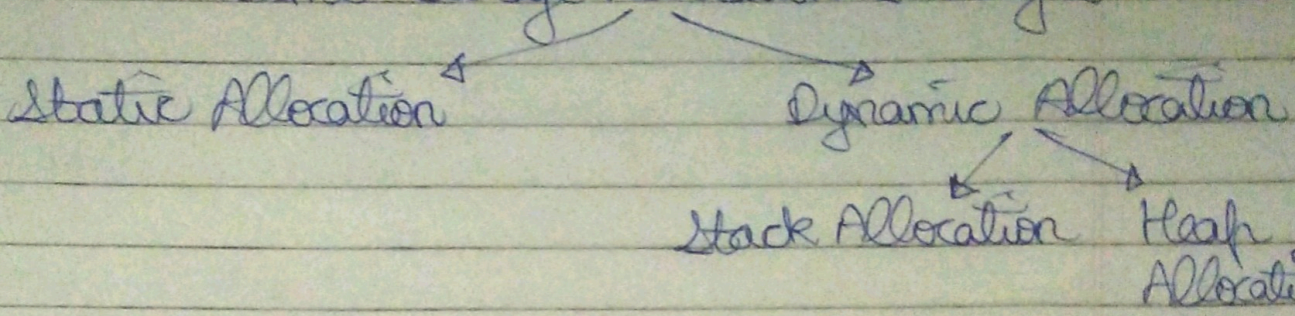


The important data structures in a compiler with minimum redundancy important information are ASTs. ASTs are easily used by the compilers as they are more compact than a parse tree.

Page No.
Date

Q5) a) Explain the different storage allocation strategies in detail.

→ Runtime Storage Allocation Strategies



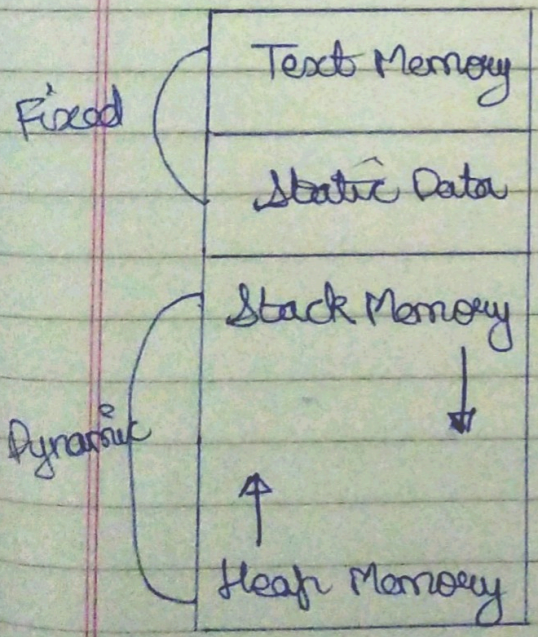
- **Storage Allocation:**
  Runtime environment manages runtime memory requirements for the following entities:
  - Code: It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
  - Procedures: Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
  - Variables: Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

- **Static Allocation:**
  In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-location is not required

- **Stack Allocation:**
Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

- **Heap Allocation:**
Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

Except statically allocated memory area, both stack and heap grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.



The text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

Q.5](b) Differentiate Top-down and Bottom-up parsing techniques. Explain shift reduce parser in detail

→

Top-Down Parsing	Bottom-Up Parsing
i) It is a parsing strategy that first looks at the highest level of the parse tree and works down the parse tree by using the rules of grammer.	It is a parsing strategy that first looks at the lowest level of the parse tree and works up the parse tree by using the rules of grammer.
ii) Top-down parsing attempts to find the most derivations of an input string.	Bottom-up parsing can be defined as an attempt to reduce the input string to start symbol of a grammer.
iii) In this parsing technique we start parsing from top (start symbol of parse tree) to down (the leaf node of parse tree) in top-down manner	In this parsing technique we start parsing from bottom (leaf node of parse tree) to up (the start symbol of parse tree) in bottom-up manner.
iv) This parsing technique uses Left Most Derivation	This parsing technique uses Right Most Derivation.
v) Its main decision is to select what production rule to use in order to construct the string.	Its main decision is to select when to use a production rule to reduce the string to get the starting symbol.

- Shift-Reduce Parser:

i) Shift reduce parsers are use the principle of bottom up parsing.

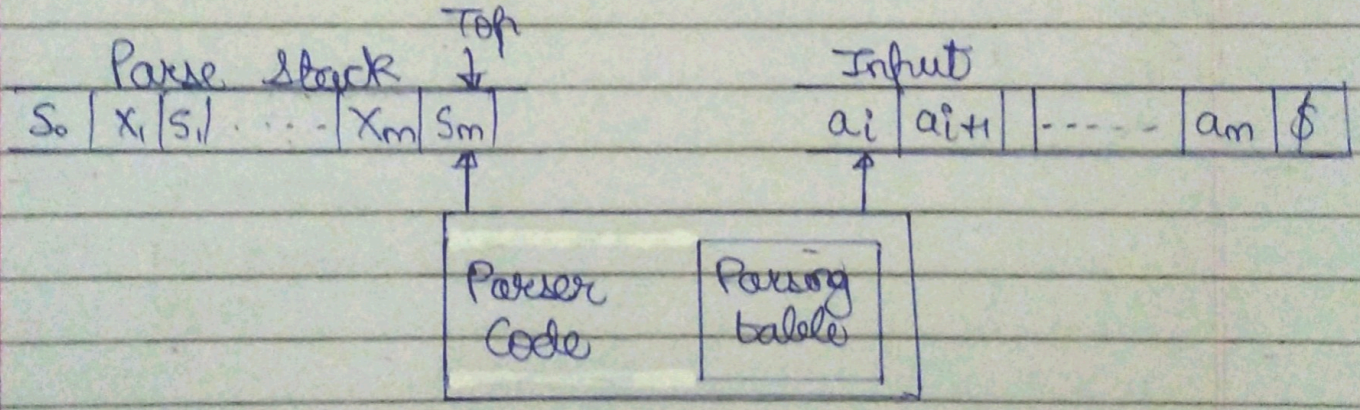
ii) Shift reduce parsers shifts input symbols until a handle is found. Then, reduce the substring to the non-terminal on the left hand side of the corresponding production.

iii) The various data structures required for a shift-reduce parsers are as follows:

(a) A stack to hold grammer symbols and to detect handles

(b) An input buffer to hold the input string to be parsed.

(c) A data structure for storing and accessing the left hand side and right hand side of the production rules.



The shift reduce parser performs the following four actions:

(i) Shift - next input symbol is shifted onto the stack

(ii) Reduce - If handle is at top of the stack

(iii) Accept - Stop parsing and report success

(iv) Error - Call error reporting/recovery routine.

Algorithm:

It uses a stack and an input buffer.

1. Initialize stack with $
2. Repeat until the top of the stack is the goal symbol and the input token is eof
   (a) find the handle
   if we don't have a handle on top of the stack, shift an input symbol onto the stack.
   (b) prune the handle.
   if we have a handle (A ::= β, k) on top of the stack, reduce
   (i) pop |β| symbols off the stack
   (ii) push A onto the stack.

Page No.
Date

Q6. a. Explain different phases of compiler. Illustrate all these phases for the following

$$a = b + c * 5$$

① Lexical Analysis
- It is also known as linear analysis or scanning. It works as interface between source program's syntax analysis.
- It determines the lexical constituents in a source string by reading the stream of characters from left to right and grouping them into tokens.

② Syntax Analysis
- It is also known as hierarchical or parsing analysis. It is a process of analyzing a sequence of tokens in order to determine its grammatical structure with respect to a given formal grammar.
- The output of syntax analysis is AST, a tree with a finite, labelled, where internal nodes are labelled by operators & leaf represent operands.
- This can be done using
  1. Top down parsing
  2. Bottom up parsing

③ Semantic Analysis
- It is used to describe the meaning of the source code. It determines the meaning of the source string and ensure that the components of

program fit together meaningfully
- It perform type checking and object binding.

(5) Intermediate code generation
- It should have two properties, it should be easy-to translate into the target program.
- The intermediate code is represented in a form called "three address code".
- Each TAC can be defined as a quad or double or triple

(3) Code optimization
- It tries to improve the intermediate code to achieve faster running machine code
- Code optimization uses various techniques as common in them are invariant, avoid redundancy. Use code straight line code avoid unary access & loop optimization

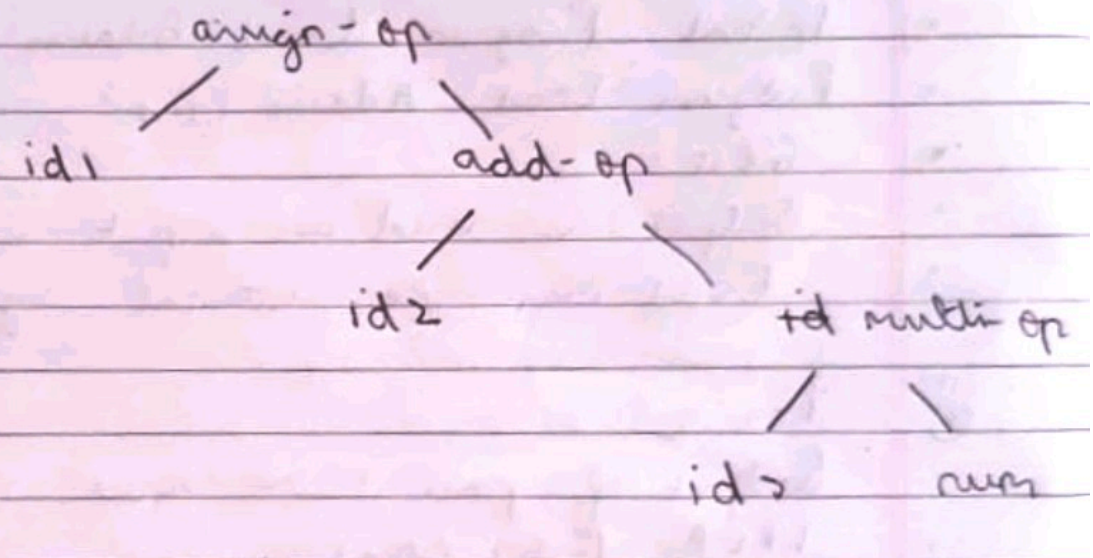
(6) Code Generation
- Code generation is the process in which a code generator accepts some internal representation of source code into machine code that can be readily executed by a machine
- The output of a code generator will be a machine code, assembly code & code for an abstract machine.

$a = b + c * 5$

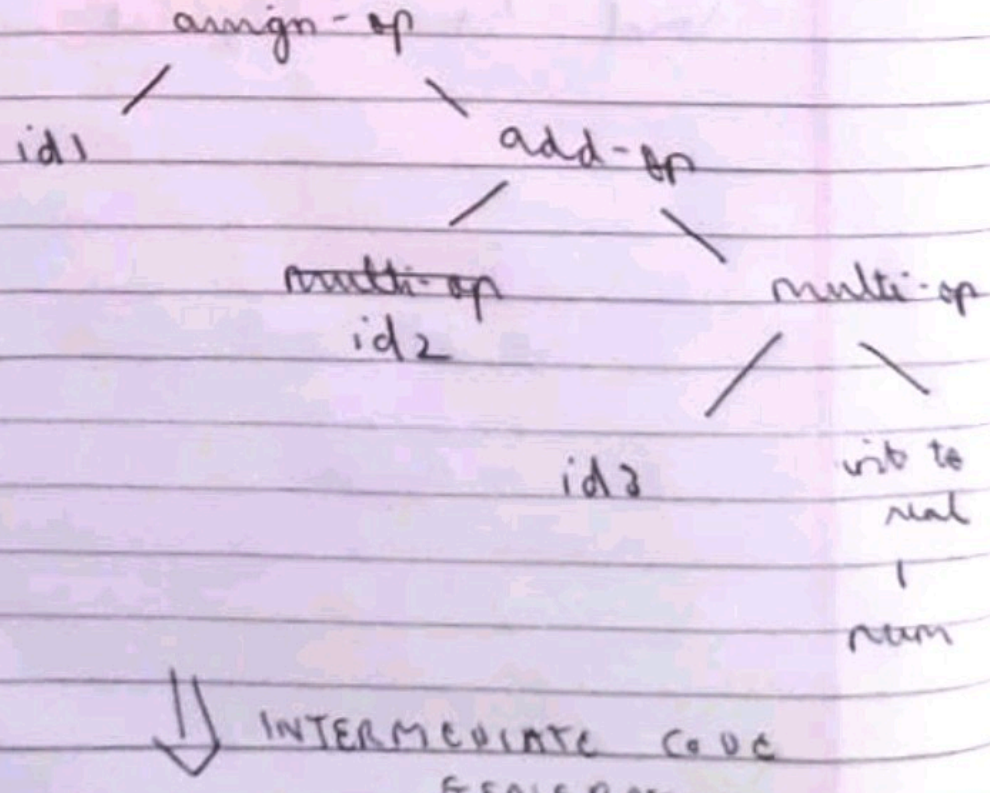
⇓ LEXICAL ANALYSIS

id1 assign-op id2 add-op id3 multi-op num

⇓ SYNTAX ANALYSIS



⇓ SEMANTIC ANALYSIS



⇓ INTERMEDIATE CODE GENERATION

```
temp1 = int to real (num)
temp2 = id3 * temp1
temp3 = temp2 + id2
id1 = temp3
```

⇓ CODE OPTIMISATION

```
t1 = id3 * rnum
id1 = t1 + id2
```

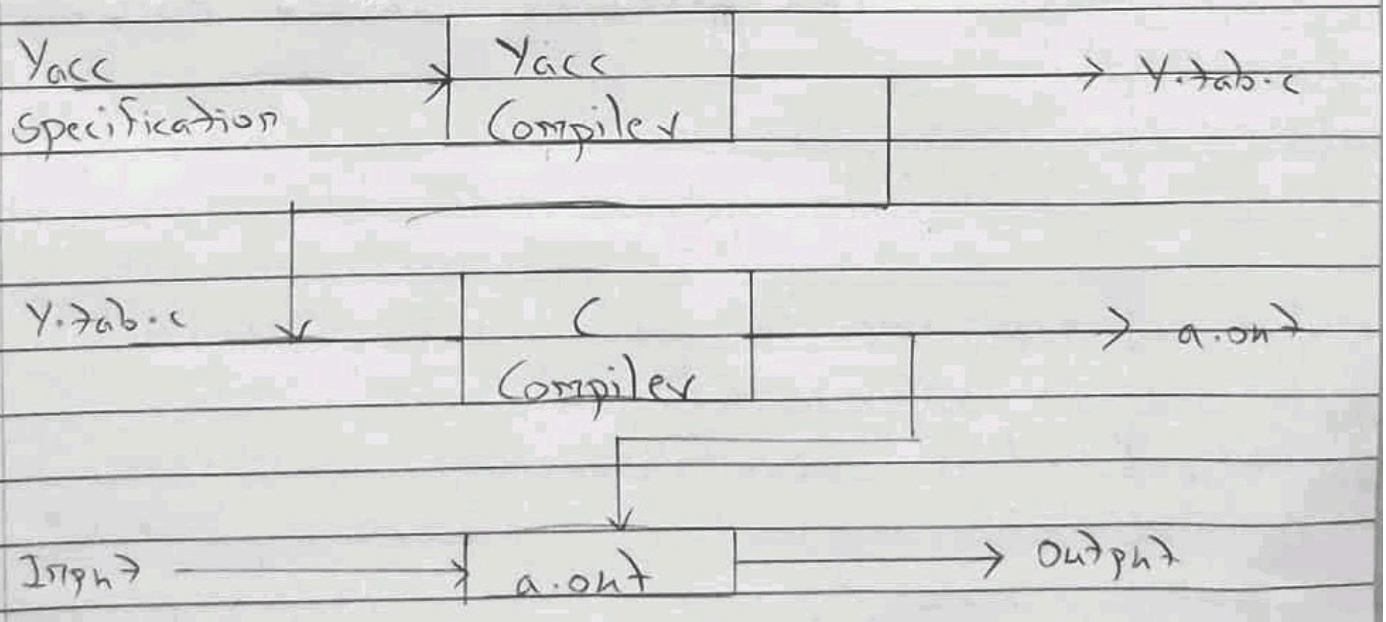
⇓ CODE GENERATION

```
movf    id3 , r2
mulf    rnum , r2
movf    id2 , r1
addf    r2 , r1
movf    r1 , id1
```

f - floating point arithmetic
r_1, r_2 two registers to store values.

Q.6 b] YACC

1. YACC is officially known as a "parser". It's job is to analyse the structure of the input stream, and operate of the "big picture".
2. In the course of it's normal work, the parser also verifies that the input is syntactically sound.
3. Consider again the example of a C-Compiler. In the C-language, a word can be a function name or a variable, depending on whether it is followed by a (or a =. There should be exactly one } for each { in the program
4. YACC stands for "Yet another Compiler Compiler". This is because this kind of analysis of text files is normally associated with writing compilers.



Working

1. It is designed for use with C code and generates a parser written in C
2. The parser is configured for use in conjunction with lex generated scanner and relies on

standard stdio features and calls the function yylex as a scanner coroutine.

3. You provide a grammar specification file and then invoke yacc on it which creates y.tab.h and y.tab.c files containing a thousand or so lines of intense C code that implements an efficient LALR(1) parser for your grammar, including code for actions you specified.

4. File provides an extern function yyparse.y that will attempt to successfully parse a valid sentence.

5. Compile that C file normally, link with the rest of your code. By default, the parser reads from stdin and writes to stdout just like a lex-generated scanner does.

Q.6] b) i) Parameterized Macros:

i) A parameterized macro is a macro that is able to insert given objects into its expression.

ii) This gives the macro some of the power of a function.

iii) Parameterized macros are useful source-level mechanism for performing in-line expansion.

iv) There are generally three ways of specifying arguments to a macro call,

(a) Positional arguments:
Arguments are matched with the dummy arguments according to the order in which they appear.

(b) Keyword arguments:
Dummy variables can also be referenced using name. This is useful when a macro has arguments that are not always used.

(c) Default arguments:
It is an argument that a programmer is not required to specify.