ASSIGNMENT - 2

Date Page 9

Ques 1. Explain input buffering in detail.

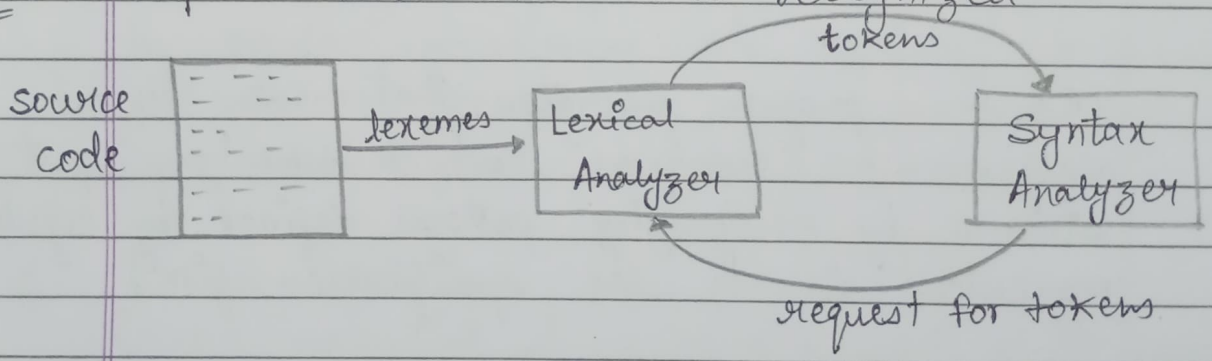
→ Input buffering is a technique where the compiler reads inputs in blocks (chunks) into a buffer instead of character by character from secondary storage. The lexical analyzer then processes characters from this buffer, which significantly reduces the number of system calls and improve performance.

→ The basic idea behind input buffering is to read a block of input from the source code into a buffer, then processes that buffer before reading the next block.

To achieve this, it uses two pointers:

- Begin pointer (bp): Marks the beginning of the current lexeme.
- Forward pointer (fp): Moves ahead to detect the end of the lexeme.

Ques. 2. Explain how tokens are recognized.



Tokens are recognized by a lexer (an lexical analyzer) in a compiler through a two-stage

process of scanning and evaluating. First, the lexer scans the source code, breaking it into meaningful character sequences called lexemes. Second, the lexer evaluates these lexemes against predefined patterns, often expressed as regular expressions, to identify and categorize them as specific tokens, such as keywords, identifiers, operators or numbers.

Example: for the input string int a = 10;

1. Scanning: The lexer produces the lexemes: int, a, =, 10, ;.
2. Evaluation:
   - int matches the pattern for the keyword int.
   - a matches the pattern for an identifier
   - = " " " " operator
   - 10 " " " a number
   - ; " " " delimiter.

Ques 3. What is LEX? Define auxiliary definitions and transitions rules for LEX with suitable examples.

→ LEX is a program generator that creates lexical analyzers, or "scanners", which tokenize input streams by recognizing patterns defined by regular expressions in a Lex specification file.

Auxiliary Definitions

This section, located at the beginning of the .l file,

defines macros and global variables that are used in the rules section.

Examples:

- Defining macros : DIGIT [0-9]
- Defining character classes - this can be used to specify ranges of characters

LETTER [a-zA-Z]

Transition Rules

This middle section contains pairs of regular expressions and their corresponding actions in C code. When the lexer encounters a sequence of characters that matches a regular expression, the associated C code is executed.

Examples: Tokenizing numbers, Recognizing keywords.

Que 4. Write a short note on

- Regular Grammar

Regular grammar is a type of formal grammar, specifically Type-3 in the Chomsky Hierarchy. It generates regular languages, which can be recognized by finite automata. Productions in regular grammars are restricted to forms like $X \to a$ or $X \to aY$, where X and Y are non-terminal and a is a terminal.

- Error Reporting

Error reporting is a crucial part of the compilation process, involving the detection, diagnosis and reporting of errors in the source code. When

an error is encountered in any phase of the compiler, the error handling routine is invoked. This routine typically generates informative error messages, often including the line number and a description of the error, to assist the programmer in debugging.

- Role of Lexical Analysis

Lexical Analysis, also known as scanning, is the first phase of a compiler. Its primary role is to read the source code character by character, grouping these characters into meaningful units called lexemes and produce a stream of token as output. It removes whitespace and comments, correlates error messages with the source code locations, and interact with the symbol table to store information about identifiers.

* Token: A token is a sequence of characters, that represents a single logical unit in the source code, such as identifier, keyword, operator or constant. It carries a specific meaning and is the smallest unit of information processed by the parser.

- Pattern: A pattern is a rule or description, often defined using regular expressions, that specifies the structure of a set of strings for which the same token will be

produced. It defines how a particular type of token should be recoguized in the input.

- Lexeme : A lexeme is the actual sequence of characters in the source code that matches a specific pattern for a token. It is the concrete instance of a token found in the input program. for ex : total = sum + 10;
total, sum and 10 are lexemes matching patterns for identifier, and number tokens respectively.

ASSIGNMENT - 3

Ques 1. Compute FIRST and FOLLOW function of the following grammar :

$S \rightarrow ABBa \mid bCA$
$A \rightarrow \epsilon BCDE$
$B \rightarrow CDA \mid cd$
$C \rightarrow eC \mid \epsilon$
$D \rightarrow bSF \mid a$

=> $FIRST(D) = FIRST(bSF) \cup FIRST(a)$
$= \{b, a\}$

$FIRST(C) = FIRST(eC) \cup FIRST(\epsilon)$
$= \{e, \epsilon\}$

$FIRST(A) = FIRST(\epsilon BCDE) = \{\epsilon\}$

$FIRST(B) = FIRST(CDA) + FIRST(cd)$
$= FIRST(C) + FIRST(DA) + FIRST(cd)$
$= \{e, b, a, c\}$

$FIRST(S) = FIRST(ABBa) \cup FIRST(bCA)$
$= \{e, b, a, c\}$

$FOLLOW(S) = \{\$, F\}$

$FOLLOW(A) = FIRST(B) + FOLLOW(S) + FOLLOW(B)$
$= \{a, b, c, e\} + \{\$, F\} + \{a, b, c, e\}$
$= \{a, b, c, e, \$, F\}$

$FOLLOW(B) = FIRST(B) + FIRST(C)$
$= \{a, b, c, e\} + \{e\} + FIRST(D)$
$= \{a, b, c, e\}$

$FOLLOW(C) = FIRST(D) + FIRST(A)$ $\because FIRST(A) = \{\epsilon\}$
$= \{a, b\} + FOLLOW(S)$

FOLLOW (C) = { a, b, $, F }

FOLLOW (D) = FIRST (A) ~~FIRST (E)~~
= FOLLOW (B) ~~&~~ ~~{E}~~
= { a, b, c, e, ~~E~~ }

Que 2. Construct Left to Right parsing table for the following grammar:

S → aAC / bB
A → eD
B → f / g
C → h / i
D → bE / ϵ
E → eD / dD

→

FIRST (S) = { a, b }	FOLLOW (S) = { $ }
FIRST (A) = { e }	" (A) = FIRST (C) = { h, i }
FIRST (B) = { f, g }	" (B) = ~~FOLLOW~~ (S) = { $ }
" (C) = { h, i }	" (C) = FOLLOW (S) = { $ }
" (D) = { b, ϵ }	FOLLOW (D) = FOLLOW (A) = { h, i }
" (E) = { e, d }	FOLLOW (E) = { h, i }

	a	b	d	e	f	g	h	i	$
S	S→aAC	S→bB							
A				A→eD					
B					B→f	B→g			
C							C→h	C→i	
D		D→bE					D→ϵ	D→ϵ	
E			E→dD	E→eD					

Ques 3. Construct Left to eliminate left recursion of the given grammar.

$S \to ABC$
$A \to Aa \mid b$
$B \to Bb \mid e$
$C \to Cc \mid f$

General:
$A \to A\alpha \mid \beta$
↳ $A \to \beta A'$
$A' \to \alpha A' \mid \epsilon$

→
$S \to ABC$
$A \to bA_1$
$A_1 \to aA_1 \mid \epsilon$
$B \to eB_1$
$B_1 \to bB_1 \mid \epsilon$
$C \to fC_1$
$C_1 \to cC_1 \mid \epsilon$

Ques 4. Construct LR(0) parsing table for following grammar:
$S \to CA \mid ccB$
$A \to CA \mid a$
$B \to CCB \mid b$

→ 1. Augmented Grammar
$S' \to S$
$S \to CA \mid ccB$
$A \to CA \mid a$
$B \to CCB \mid b$

2. $I_0 = \text{closure}(\{S' \to .S\}) = \{ S' \to .S$
$S \to .CA \mid .ccB$
~~$A \to .CA \mid .a$~~
~~$B \to .CCB \mid .b$~~

goto (I_0, S) = closure $(\{S' \to S.\})$ = $S' \to S.$ = I_1 *

goto (I_0, C) = closure $(\{S \to C.A\})$ = $S \to C.A$
$A \to .cA$
$A \to .a$
} I_2

goto (I_0, c) = closure $(\{S \to c.cB\})$ = $S \to c.cB$ = I_3

goto (I_2, A) = closure $(\{S \to CA.\})$ = $S \to CA.$ = I_4 *

goto (I_2, C) = closure $(\{A \to c.A\})$ = $A \to c.A$
$A \to .cA$
$A \to .a$
} I_5

goto (I_2, a) = closure $(\{A \to a.\})$ = $A \to a.$ = I_6 *

goto (I_3, c) = " $(\{S \to cc.B\})$ = $S \to cc.B$
$B \to .ccB$
$B \to .b$
} I_7

goto (I_5, A) = closure $(\{A \to cA.\})$ = $A \to cA.$ = I_8 *

goto (I_5, c) = " $(\{A \to c.A\})$ = State I_5 repeated

goto (I_5, a) = " $(\{A \to a.\})$ = " I_6 repeated

goto (I_7, B) = closure $(\{S \to ccB.\})$ = $S \to ccB.$ = I_9 *

goto (I_7, c) = " $(\{B \to c.cB\})$ = $B \to c.cB$ = I_{10}

goto (I_7, b) = " $(\{B \to b.\})$ = $B \to b.$ = I_{11} *

goto (I_{10}, c) = closure $(\{B \to cc.B\})$ = $B \to cc.B$
$B \to .ccB$
$B \to .b$
} I_{12}

goto (I_{12}, B) = closure $(\{B \to ccB.\})$ = $B \to ccB.$ = I_{13} *

goto (I_{12}, c) = " $(\{B \to c.cB\})$ = state I_{10} repeated

goto (I_{12}, b) = " $(\{B \to b.\})$ = state I_{11} repeated

State	Actions a	b	c	$	Goto S	A	B	C
I_0					1			2
I_1				accept				
I_2	S_6		S_5			4		
I_3			S_7					
I_4	R_1	R_1	R_1	R_1				
I_5	S_6		S_5			8		
I_6	R_4	R_4	R_4	R_4				
I_7		S_{11}	S_{10}				9	
I_8	R_3	R_3	R_3	R_3				
I_9	R_2	R_2	R_2	R_2				
I_{10}			S_{12}					
I_{11}	R_6	R_6	R_6	R_6				
I_{12}		S_{11}	S_{12}				13	
I_{13}	R_5	R_5	R_5	R_5				

Que 5. Construct LR(1) parsing table for following grammar:
$S \rightarrow aSbS \mid bSaS \mid \epsilon$

→ 1. Augmented Grammar
$S' \rightarrow S$
$S \rightarrow aSbS$
$S \rightarrow bSaS$
$S \rightarrow \epsilon$

2. I_0: $S' \rightarrow .S, \$$
$S \rightarrow .aSbS, \$$
$S \rightarrow .bSaS, \$$
$S \rightarrow .\epsilon, \$$

I_1 : $S' \rightarrow S.$, $

I_2 : $S \rightarrow a.SbS$, $
$S \rightarrow .aSbS$, b
$S \rightarrow .bSaS$, b
$S \rightarrow .\epsilon$, b

I_3 : $S \rightarrow b.SaS$, $
$S \rightarrow .aSbS$, a
$S \rightarrow .bSaS$, a
$S \rightarrow .\epsilon$, a

I_4 : Goto(I_2, S)
$S \rightarrow aS.bS$, $

I_5 : Goto(I_2, a)
$S \rightarrow a.SbS$, b
$S \rightarrow .aSbS$, b
$S \rightarrow .bSaS$, b
$S \rightarrow .\epsilon$, b

I_6 : Goto(I_2, b)
$S \rightarrow b.SaS$, b
$S \rightarrow .aSbS$, a
$S \rightarrow .bSaS$, a
$S \rightarrow .\epsilon$, a

I_7 : Goto(I_3, S)
$S \rightarrow bS.aS$, $

I_8 : Goto(I_3, a)
$S \rightarrow a.SbS$, a
$S \rightarrow .aSbS$, b
$S \rightarrow .bSaS$, b
$S \rightarrow .\epsilon$, b

I_9: Goto (I_3, b)

$S \to b.SaS, a$

$S \to .aSbS, a$

$S \to .bSaS, a$

$S \to .\epsilon, a$

I_{10}: Goto (I_4, b)

$\{ [S \to aSb.S, \$], [S \to .aSbS, \$], [S \to .bSaS, \$], [S \to .\epsilon, \$] \}$

I_{11}: Goto (I_5, S)

$S \to aS.bS, b$

for Goto (I_5, a) = State I_5 repeated

" " (I_5, b) = State I_6 "

I_{12}: Goto (I_6, S)

$S \to bS.aS, b$

I_{13}: Goto (I_7, a)

$\{ [S \to bSa.S, \$], [S \to .aSbS, \$], [S \to .bSaS, \$], [S \to .\epsilon, \$] \}$

I_{14}: Goto (I_8, S)

$S \to aS.bS, a$

I_{15}: Goto (I_9, S)

$S \to bS.aS, a$

I_{16}: Goto (I_{10}, S)

$S \to aSbS., \$$ [Eq. ① get completed]

I_{17}: Goto (I_{11}, b)

$S \to aSb.S, b$

$S \to .aSbS, b$

$S \to .bSaS, b$

$S \to .\epsilon, b$

I_{18}: Goto (I_{12}, a)

$S \to bSa.S, b$

$S \to .aSbS, b$

$S \to .bSaS, b$

$S \to .\epsilon, b$

I_{19} : Goto (I_{13}, S)
$S \to bSaS.$, $ } { for Goto(I_{13}, a) = I_2 repeated
" " (I_{13}, b) = I_3 repeated

I_{20} : Goto (I_{14}, b)
{ [$S \to aSb.S$, a], [$S \to .aSbS$, a],
[$S \to .bSaS$, a] , [$S \to .\epsilon$, a] }

I_{21} : Goto (I_{15}, a)
{ [$S \to bSa.S$, a], [$S \to .aSbS$, a], [$S \to .bSaS$, a]
[$S \to .\epsilon$, a] }

I_{22} : Goto (I_{17}, S) $\Rightarrow$ { [$S \to aSbS.$, b] }
I_{23} : Goto (I_{18}, S) $\Rightarrow$ { [$S \to bSaS.$, b] }
I_{24} : Goto (I_{20}, S) $\Rightarrow$ { [$S \to aSbS.$, a] }
I_{25} : Goto (I_{21}, S) $\Rightarrow$ { [$S \to bSaS.$, a] }

	Action			Goto
State	a	b	$	S
I_0	S_2	S_3	R_3	1
I_1			accept	
I_2	S_5	S_6/R_3		4
I_3	S_8/R_3	S_9		7
I_4		S_{10}		
I_5	S_5	S_6/R_3		11
I_6	S_8/R_3	S_9		12
I_7	S_{13}			
I_8	S_5	S_6/R_3		14
I_9	S_8/R_3	S_9		15
I_{10}	S_2	S_3	R_3	16
I_{11}		S_{17}		
I_{12}	S_{18}			
I_{13}	S_2	S_3	R_3	19
I_{14}		S_{20}		

	Action			Goto
State	a	b	$	S
I_{15}	S_{21}			
I_{16}			R_1	
I_{17}	S_5	S_6/R_3		22
I_{18}	S_5	S_6/R_3		23
I_{19}			R_2	
I_{20}	S_8/R_3	S_9		24
I_{21}	S_8/R_3	S_9		25
I_{22}		R_1		
I_{23}		R_2		
I_{24}	R_1			
I_{25}	R_2			

There is a Shift/Reduce Conflict, therefore grammar $S \to aSbS / bSaS / \epsilon$ is not an LR(1) grammar.

Ques 6. Construct LALR(1) parsing table for following grammar:

$S \to \overset{①}{A}a / \overset{②}{a}Ac / \overset{③}{B}c / \overset{④}{b}Ba$

$A \to d$ ⑤

$B \to d$ ⑥

→ 1. Augmented Grammar

$S' \to S$

$S \to Aa / aAc / Bc / bBa$

$A \to d$

$B \to d$

#. Canonical collection of LR(1) items

I_0 : $S' \to .S$, $

$S \to .Aa$, $

$S \to .aAc$, $

$S \to .Bc$, $

$S \to .bBa$, $

$A \to .d$, a

$B \to .d$, c

→ Goto $(I_0, S) = I_1$

Goto $(I_0, A) = S \to A.a, \$ = I_2$

Goto $(I_0, B) = S \to B.c, \$ = I_3$

Goto $(I_0, a) = S \to a.Ac, \$$
$A \to .d, c$ } I_4

Goto $(I_0, b) = S \to b.Ba, \$$
$B \to .d, a$ } I_5

Goto $(I_0, d) = A \to d., a$
$B \to d., c$ } I_6

Goto $(I_2, a) = S \to Aa., \$ = I_7$
Goto $(I_3, c) = S \to Bc., \$ = I_8$
Goto $(I_4, A) = S \to aA.c, \$ = I_9$
Goto $(I_4, d) = A \to d., c = I_{10}$
Goto $(I_5, B) = S \to bB.a, \$ = I_{11}$
Goto $(I_5, d) = B \to d., a = I_{12}$
Goto $(I_9, c) = S \to aAc., \$ = I_{13}$
Goto $(I_{11}, a) = S \to bBa., \$ = I_{14}$

Step 3: Combine states for LALR(1)

Core $I_0 : \{S' \to .S, S \to .Aa, S \to .aAc, S \to .Bc, S \to .bBa, A \to .d, B \to .d\}$
" $I_1 : \{S' \to S.\}$
" $I_2 : \{S \to A.a\}$
" $I_3 : \{S \to B.c\}$
" $I_4 : \{S \to a.Ac, A \to .d\}$
" $I_5 : \{S \to b.Ba, B \to .d\}$
" $I_6 : \{A \to d., B \to d.\}$
" $I_7 : \{S \to Aa.\}$
" $I_8 : \{S \to Bc.\}$
" $I_9 : \{S \to aA.c\}$
" $I_{10} : \{A \to d.\}$
" $I_{11} : \{S \to bB.a\}$
" $I_{12} : \{B \to d.\}$
" $I_{13} : \{S \to aAc.\}$
" $I_{14} : \{S \to bBa.\}$

Upon inspection, no two states share the same core.

State	Action					Goto		
	a	b	c	d	$	S	A	B
I_0	S_4	S_5		S_6		1	2	3
I_1					accept			
I_2	S_7							
I_3			S_8					
I_4				S_{10}			9	
I_5				S_{12}				11
I_6	R_5	R_5	R_6					
I_7					R_1			
I_8					R_3			
I_9			S_{13}					
I_{10}			R_5					
I_{11}	S_{14}							
I_{12}	R_6							
I_{13}					R_2			
I_{14}					R_4			

ASSIGNMENT - 4

Ques. 1. Explain various storage allocation techniques on symbol table.

1. Static Allocation :
Memory for variables and other data is allocated at compile time. The size and location of these allocations are fixed and donot change during program execution.
Characteristics :
- Data is laid out in memory before the program runs.
- Memory is allocated once and remains allocated for the entire program's execution.

2. Stack Allocation :
Memory is managed using a Last-In-First-Out (LIFO) data structure, the stack.
Characteristics :
Efficient for managing local variables and function calls, as memory is automatically allocated when a function is entered and deallocated when it exits.
Symbol Table Usage : The symbol table contains entries for local variables, and stack allocation manages their placement in memory frames on the runtime stack.

3. Heap Allocation :
Memory is allocated at runtime from a

free-storage pool called the heap.
Characteristics:
Highly flexible, allowing for dynamic memory allocation for variables with unpredictable lifetimes or those that need to persist beyond a single function call. However, it requires manual deallocation (using functions like free() or delete in languages like C++) to prevent memory leaks.

→ The symbol table holds pointers or references to the memory locations allocated on the heap for dynamic variables.

Ques 2. Explain in brief various data structures that can be used in symbol table.

→ Various data structures can be used for its implementation, each with different performance characteristics:

- Linear Lists (Arrays / Linked List):
  Symbols are stored sequentially. In an array, they are in contiguous memory; In a Linked List, each element points to the next.
  ↳ Simple to implement.
  ↳ Search time is O(N) in the worst case, making it inefficient for large symbol tables.

- Binary Search Trees (BSTs):
  Symbols are organized hierarchically, allowing for efficient searching. Left children have

keys smaller than the parent. and right children have keys larger than the parent.
- Average search, insertion & deletion time is O(log N)
- Can degenerate to a linked list (O(N) performance) in worst case if not balanced.

- Hash Tables
Uses a hash function to map symbol names (key) to indices in an array, providing direct access to the symbol's information.
- Average search, insertion & deletion time is O(1)
- Worst case performance can be O(N) if many collisions occur or hash function is poor. Requires careful design of hash function and collision resolution.

Ques 3. Differentiate Static & Dynamic binding.

Feature	Static Binding	Dynamic Binding
Time	Compile-time	Run-time
Resolution	Based on reference type	Based on actual object type.
Flexibility	Low	High
Performance	Faster	Slower
Examples	Method overloading, final/ private/ static methods.	Method Overriding (virtual functions).

Ques 4. Write short notes on :

- Activation record

An activation record (AR) also known as a stack frame, is a data structure created at runtime when a procedure or function is called. It contains all the necessary information for the execution of that procedure, including :

→ Local data : Variables declared within procedure's scope

→ Machine Status : Saved register values, program counter etc, to restore the caller's state upon return.

→ Control Link : A pointer to the activation record of the caller procedure.

→ Actual Parameters : The values or addresses of arguments passed to the procedure.

→ Return Value : Space to store the value returned by the procedure

- Syntax and Semantic Errors

→ Syntax Errors : These are violations of the grammatical rules of a programming language. They are detected during the syntax analysis phase of a compiler. Examples include missing semicolons, unmatched paranthesses or incorrect keyword usage.

→ Semantic Errors : These are the violations of the meaning or logic rules of a programming language. They are detected during the semantic analysis phase. While the code might be syntactically correct, it might not make sense

logically or adhere to type-checking rules. Examples include type mismatches in assignments or operators, using undeclared variables.

- Parameter Passing

It refers to the mechanism by which arguments are transfered from a calling procedure to a called procedure. Common methods include:

- Call by Value:

A copy of the actual parameter's value is passed to the formal parameter. Changes to the formal parameter within the called procedure do not affect the original actual parameter.

- Call by Reference:

The memory address of the actual parameter is passed to the formal parameter. This means both formal and actual parameters refer to the same memory location and changes to the formal parameter do affect the original actual parameter.

- Intermediate Code Form

Intermediate code is an abstract, machine-independent representation of the source code, generated by the front-end of a compiler before target code generation and optimization. It serves as a bridge b/w the source language and the target machine language, simplifying the optimization process and allowing for easier retargeting to different architectures.

Common intermediate forms include :

→ Three - Address code

→ Postfix Notation

→ Syntax Trees / Directed Acyclic Graphs (DAGs)

Ques 5. Translate the following expression to quadruple and indirect triple :

$$-(x+y) \times (z+c) - (x+y+z)$$

Step

→ 1. We break down the expression into a sequence of operations, each with at most one operator on the right hand side. We will use temporary variables t_1, t_2, t_3 -- to store intermediate results.

1. $t_1 = x + y$
2. $t_2 = z + c$
3. $t_3 = t_1 \times t_2$
4. $t_4 = -t_3$ (unary minus)
5. $t_5 = t_1 + z$ (Reusing t_1 for the second $x+y$)
6. $t_6 = t_4 - t_5$

(#) Quadruple representation -

A quadruple is a representation with 4 fields: (operator, argument 1, argument 2, result).

#	Operator	Arg 1	Arg 2	Result
0	+	x	y	t_1
1	+	z	c	t_2
2	x	t_1	t_2	t_3
3	uminus	t_3	-	t_4
4	+	t_1	z	t_5
5	-	t_4	t_5	t_6

(#) Indirect Triple Representation

It uses two parts

1. Triple Table : A list of instructions, each with three fields : (operator, argument 1, argument 2).

2. Instruction List : An array of pointers that indicates the order of execution of the triples.

1. Triples Table :

#	Operator	Arg 1	Arg 2
0	+	x	y
1	+	z	c
2	x	(0)	(1)
3	uminus	(2)	-
4	+	(0)	z
5	-	(3)	(4)

2. Instruction List (Pointer Array)

	Pointer
0	(0)
1	(1)
2	(2)
3	(3)
4	(4)
5	(5)