Date 10th Jan-13 Thursday

• Compiler Design.

(TSR) Coding for viruses.

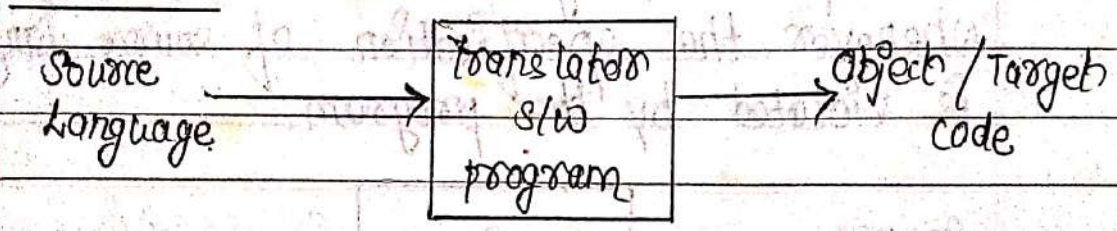
• UNIT-I

• INTRODUCTION TO COMPILER

• TRANSLATOR

A translator is defined as a software program that takes as a input a program in source language and produces as output a program in another language.

→ The concept of translating a program from source to object or target is shown as:



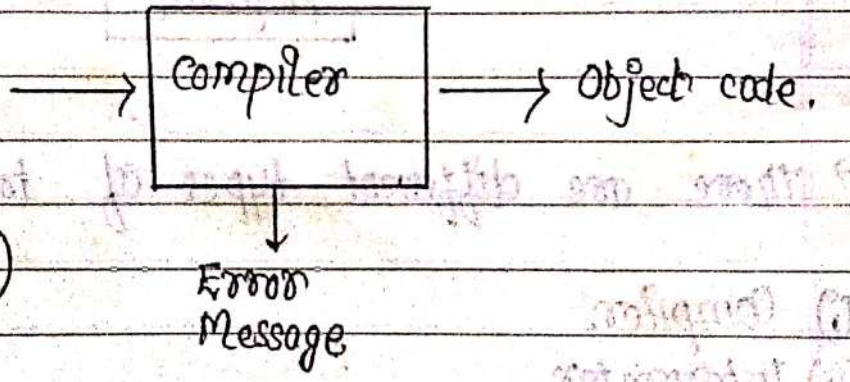
→ There are different types of translators.

(i) Compiler.
(ii) Interpreter.
(iii) Assembler.
(iv) Macro assembler.
(v) Preprocessor.
(vi) High level translator.
(vii) De-compiler and de-assembler.

• COMPILER

→ A compiler is a translator (a s/w program) that translates a HLL program into a functionally equivalent Low level or Machine language program which can be executed directly.

→ A compiler also generates error message whenever the specification of source language is violated by the program.



• ADVANTAGES OF COMPILER

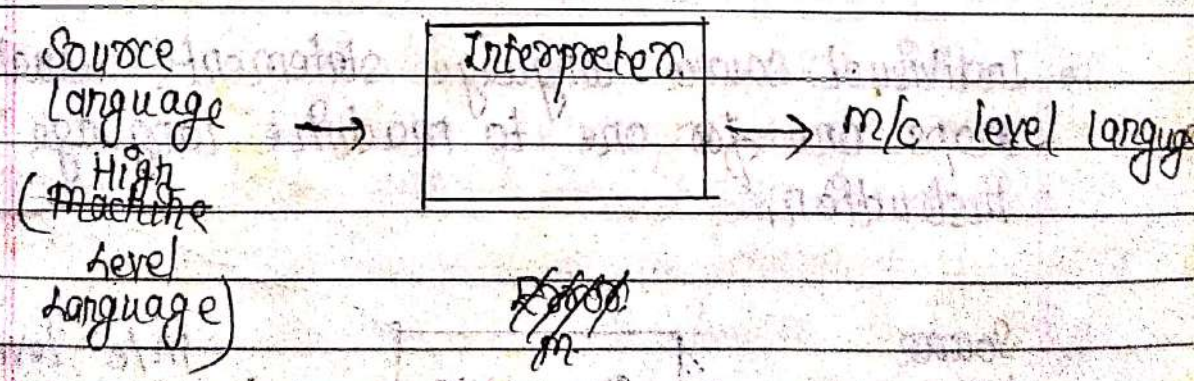
(i) It consumes less time.
(ii) A compiler translates a program in single run.
(iii) CPU utilisation is more.
(iv) Both syntactic and semantic error can be checked at the same time.
(v) →

- **DISADVANTAGES OF COMPILER**

(i) It is not flexible.
(ii) It consumes more space.
(iii) Error localisation is difficult.
(iv) The source program has to be compiled for every modification.
(v) An object program produced much larger than the source program that produced it.

- **INTERPRETER**

→ An interpreter is a translator that translates a high level language program into functionally equivalent low level machine code, but it does it at the moment the program run.



- **ADVANTAGES**

(i) An interpreter translates a program line by line.
(ii) Interpreter are often smaller in size.
(iii) It is flexible.
(iv) Error localisation is easier.
(v) An interpreter facilitates the implementation of

complex programming language constructs.
(vi) It is easily supported by languages like Java, HTML.

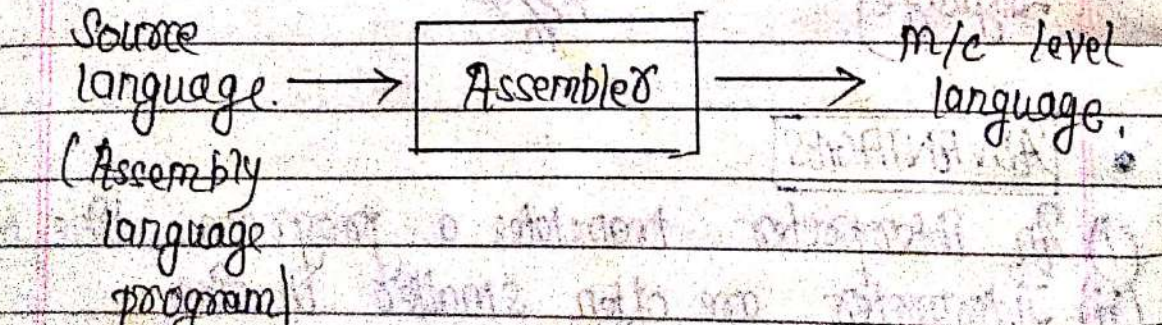
- **DISADVANTAGES**

(i) It consumes more time.
(ii) Only syntactic error can be checked.
(iii) CPU utilization is less.
(iv) Interpreter are less efficient as compared to compiler.

- **ASSEMBLER**

An assembler is a translator that translates a assembly language program into low level machine language program which can then be executed directly.

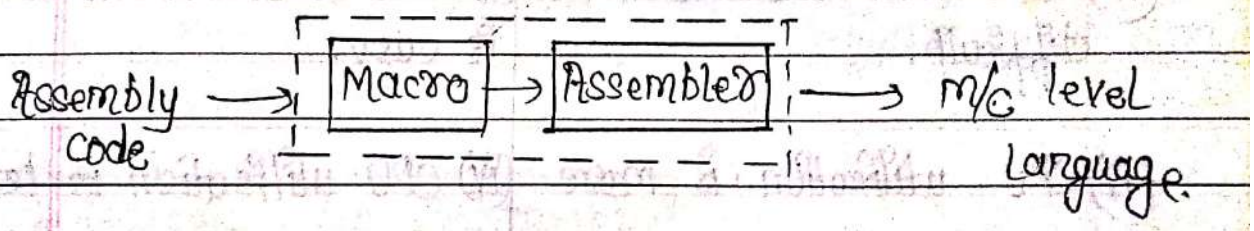
→ Individual source language statements usually maps one for one to machine language instruction.



- **MACRO ASSEMBLER**

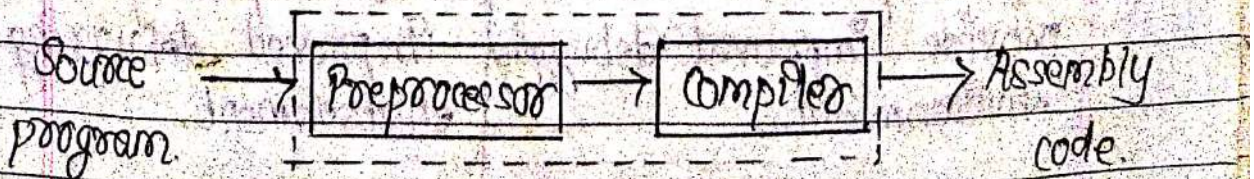
A macro assembler is a translator that translates assembly level language instructions into machine code and is a variation on assembler.

Most source level language statement maps one for one into their target language equivalent, but some macro statement maps into a sequence of machine level instructions effectively.



- **PRE-PROCESSOR**

A pre-processor is a translator that translates a super set of high level language into the original high level language or that performs simple text substitution before the translation takes place, this enables the statement of source program to be processed before it is being inputed to main compiler.



• DIFFERENCE BETWEEN COMPILER AND INTERPRETER

COMPILER	INTERPRETER
(i) A compiler translates complete source program in a single run.	(i) An interpreter translates source program line by line.
(ii) It consumes less time.	(ii) It consumes more time.
(iii) The error localisation is difficult.	(iii) The error localisation is easy.
(iv) CPU utilisation is more	(iv) CPU utilisation is less.
(v) Both syntactic and semantic errors can be checked at the same time.	(v) Only syntactic error can be checked at a time
(vi) Presence of an error may cause whole program to be reorganised.	(vi) A presence of an error causes only a part of the program to be reorganised.
(vii) A compiler does not provide improved debugging environment	(vii) An ~~compiler~~ interpreter provides improved debugging environment.

COMPILER	INTERPRETER
(viii) It is faster	(viii) It is slower
(ix) It is more efficient.	(ix) It is less efficient.
(x) It is not flexible	(x) It is flexible.
(xi) Compiler are larger in size.	(xi) Interpreter are often smaller in size.
(xii) Compiler used by languages C, C++ etc.	(xii) An interpreter is used by languages like Java, HTML.

- **PHASES OF COMPILER**

Compilation refers to compiler's process of translating a high level language program into low level language program.

This process is very complex; hence, from the logical as well as implementation point of view it is customary to partition the compilation process into several phases, which are nothing more than logically cohesive operations that input one representation of source program and output another representation.

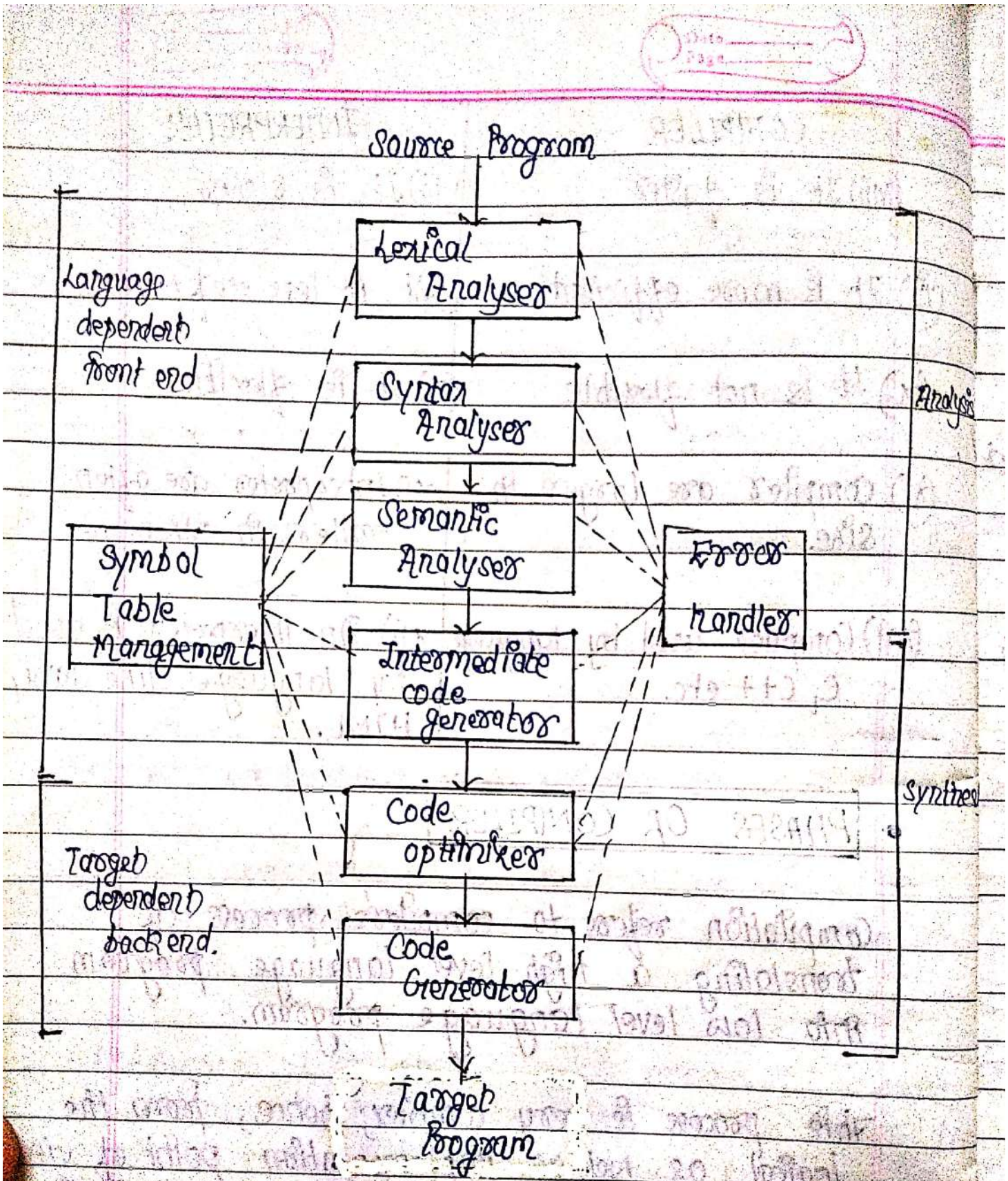


Fig: Phases of Compiler

LEXICAL ANALYSIS

→ It is an interaction between source program and compiler. This phase is also called scanner.

→ This phase performs the linear analysis on the source program, it reads character of source program from left to right and separate character into groups that logically belongs together called tokens i.e keywords, Identifiers, operators, constants, delimiters and other puntuation symbols.

→ And it also removes multiple spaces, comments and any other character not relevant to the later stages of analysis.

→ The recognition of token is done by DFA.

Example = Position := initial + rate * 60.

→ The lexical analyser will compute this as follows:

Symbol	Tokens	Attributes
Position	Identifier	#1 (id1)
:=	operator	Assignment (1)
Initial	Identifier	#2 (id2)
+	operator	Arithmetic (1)
rate	Identifier	#3 (id3)
*	Operator	Arithmetic (2)
60	Constant	#4.

11-1-13 Friday

SYNTAX ANALYSIS

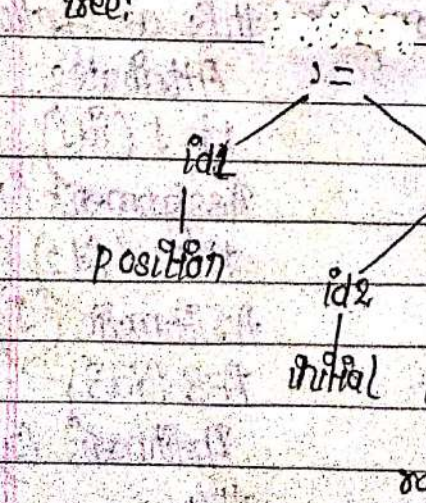
→ The second phase of compiler is syntax analyser or parsing.

→ The parser uses the first components of the token produced by lexical analyser to create tree-like intermediate representation that depicts the grammatical structure of token string.

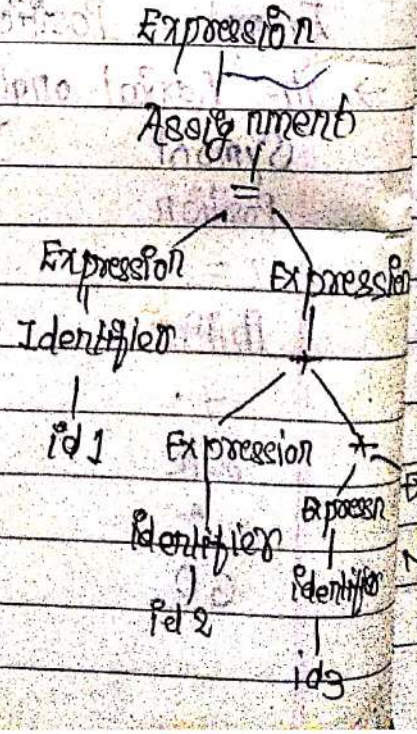
→ A typical representation is a syntax tree in which each interior node represents an operator and the children of the node represents the arguments of operator i.e operand.

Ex: A syntax tree for
position := initial + rate * 60

Syntax tree:



Parse tree:



SEMANTIC ANALYSIS

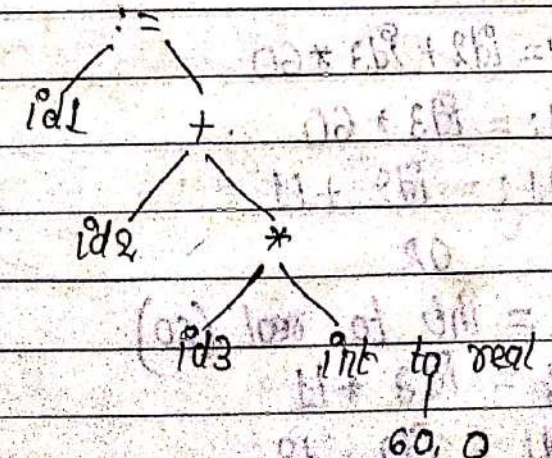
→ A semantic analyser uses the syntax tree and the info in the symbol table to check the source program for semantic consistency with the language definition.

→ It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate code generation

→ An important part of semantic analysis is type checking where the compiler check that each operator has matching operands.

for ex: Many programming language definition require an array index to be an integer; the compiler must report an error if a floating point number is used to index an array.

a[s]



Expression Number 60

Annotated tree or Semantic tree.

INTERMEDIATE CODE GENERATOR

After syntax and semantic analysis, compiler generates an explicit intermediate representation of source program.

This intermediate representation have two imp properties:

(i) It should be easy to produce.
(ii) Easy to translate into the target program.

The intermediate representation can have variety of forms.

(i) Syntax directed translation.
(ii) 3 address code (TAC) 3AC
(iii) Postfix notation.

(1) id1 := id2 + id3 * 60

3AC
t1 := id3 * 60
id1 := id2 + t1

OR

t1 = int to real (60)
t2 = id3 * t1
id1 = id2 + t2

② $a = b * c + d / e$

$t1 = b * c$

$t2 = d / e$

$t3 = t1 + t2$

$a = t3$

or $a = t_1 + t_2$

- **CODE OPTIMISATION**

This phase attempts to improve the execution efficiency of the program.

The main aim of this phase is to improve the intermediate code to generate a code that runs faster and/or occupies less space.

Ex: $id1 = id2 + id3 * 60$

$t1 = id3 * 60.0$

$id1 = id2 + t1$

- **CODE GENERATION**

Final phase of compiler is the generation of target code.

The main aim of this phase is to allocate storage and generates a relocatable m/c / assembling code, memory location and registers are allocated for variables.

The instructions in intermediate code format are converted into machine instructions.

Ex: id1 = id2 + id3 * 60

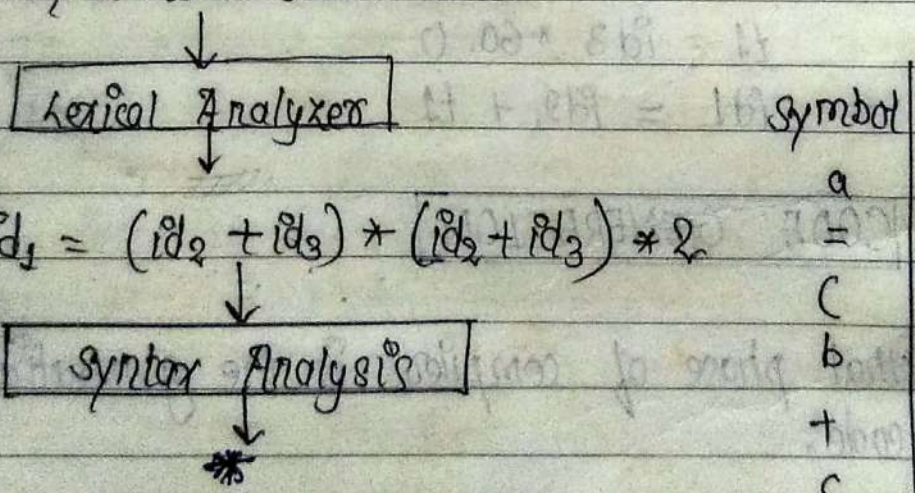
```
MOV  R1, id3
MUL  R1, #60.0
MOV  R2, id2
ADD  R2, R1
MOV  id1, R2
```

Date: 16-Jan-18 Wednesday

CD

Ques: a = (b+c) * (b+c) * 2

Solve: Since, it is a source code.

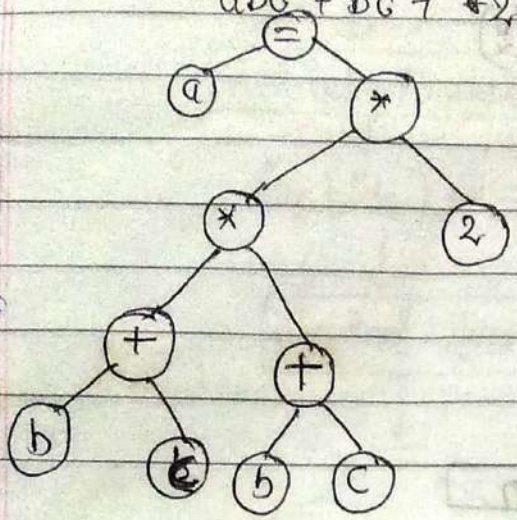


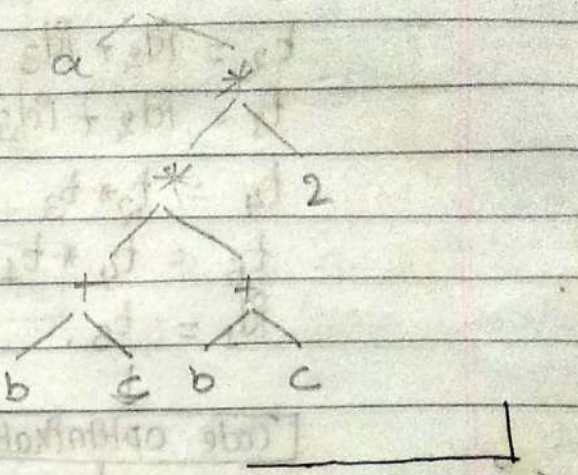
Symbol	Tokens
a	id1
=	Assignment operator
(	Opening brace
b	id2
+	Arithmetic operator
c	id3
)	closing brace
*	Arithmetic operator
2	Constant

abc+ bc+*2*=

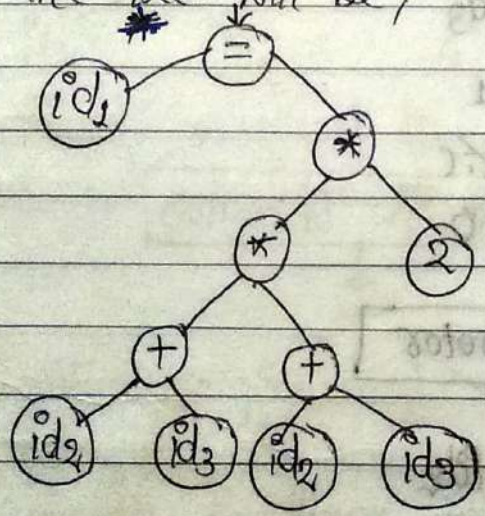
Now, a = (b+c) * (b+c) *2

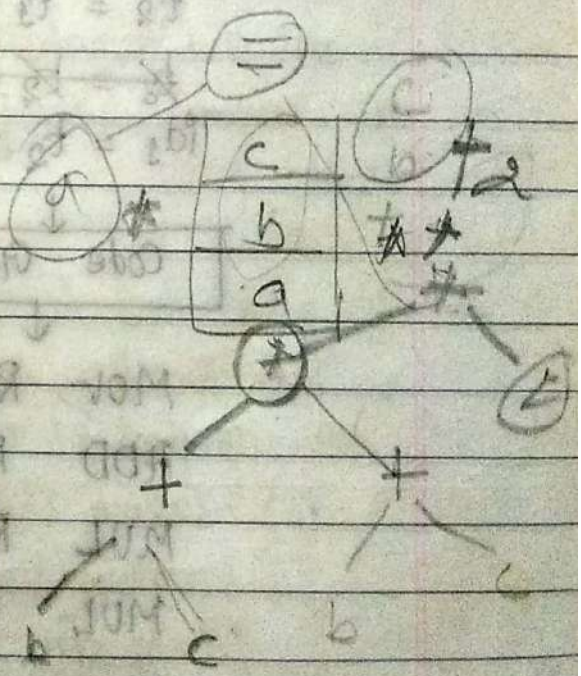
abc + bc + *2, * =



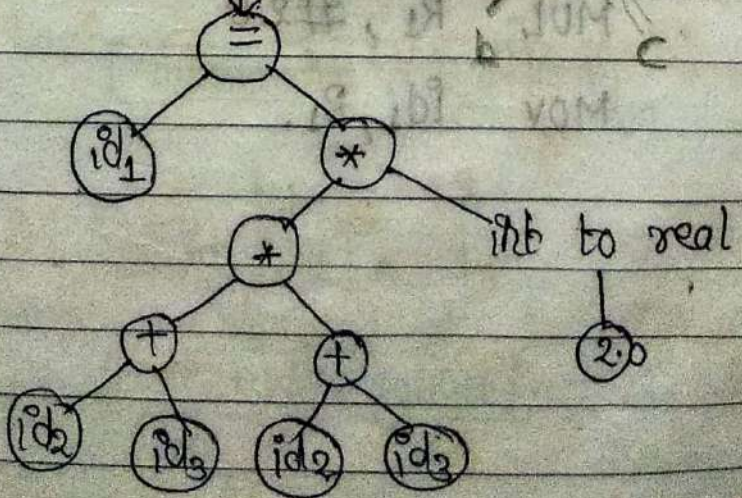


∴ The tree will be,





Semantic Analysis



*

↓

Intermediate code generation

$t_1 = \text{int to real}(2)$

$t_2 = id_2 + id_3$

$t_3 = id_2 + id_3$

$t_4 = t_2 * t_3$

$t_5 = t_4 * t_1$

$id_1 = t_5$.

↓

Code optimization

$t_1 = id_2 + id_3$

$t_2 = t_1 * t_1$

~~$t_3 = t_2 * 2.0$~~

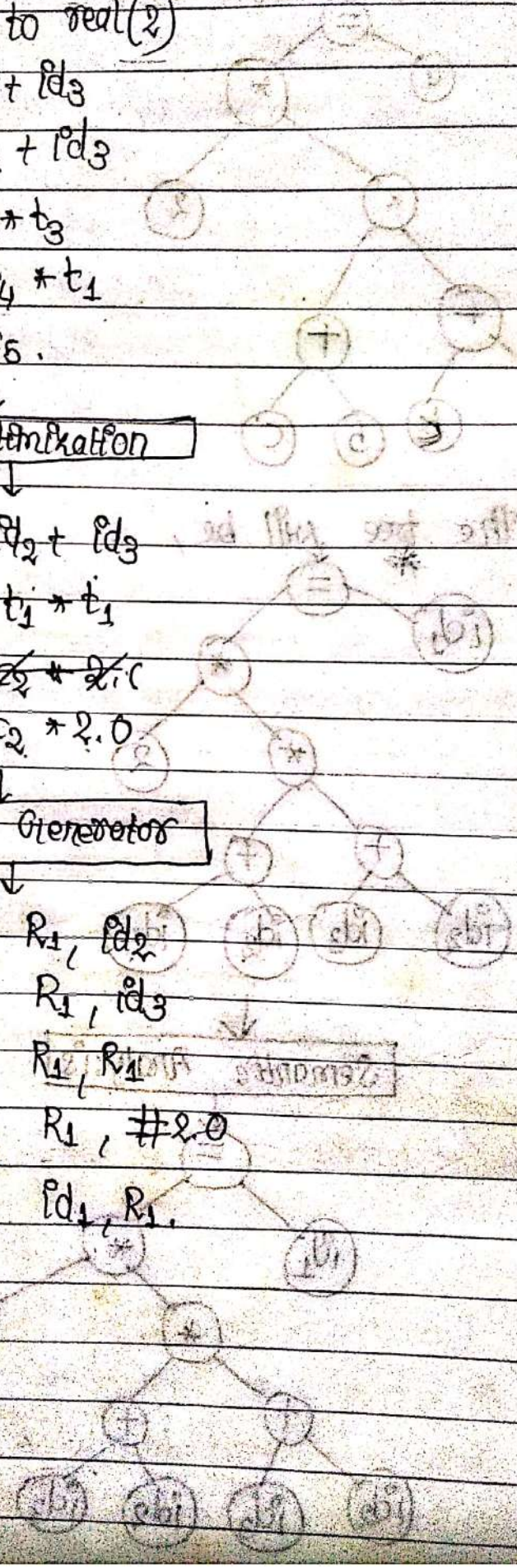
$id_1 = t_2 * 2.0$

↓

Code Generator

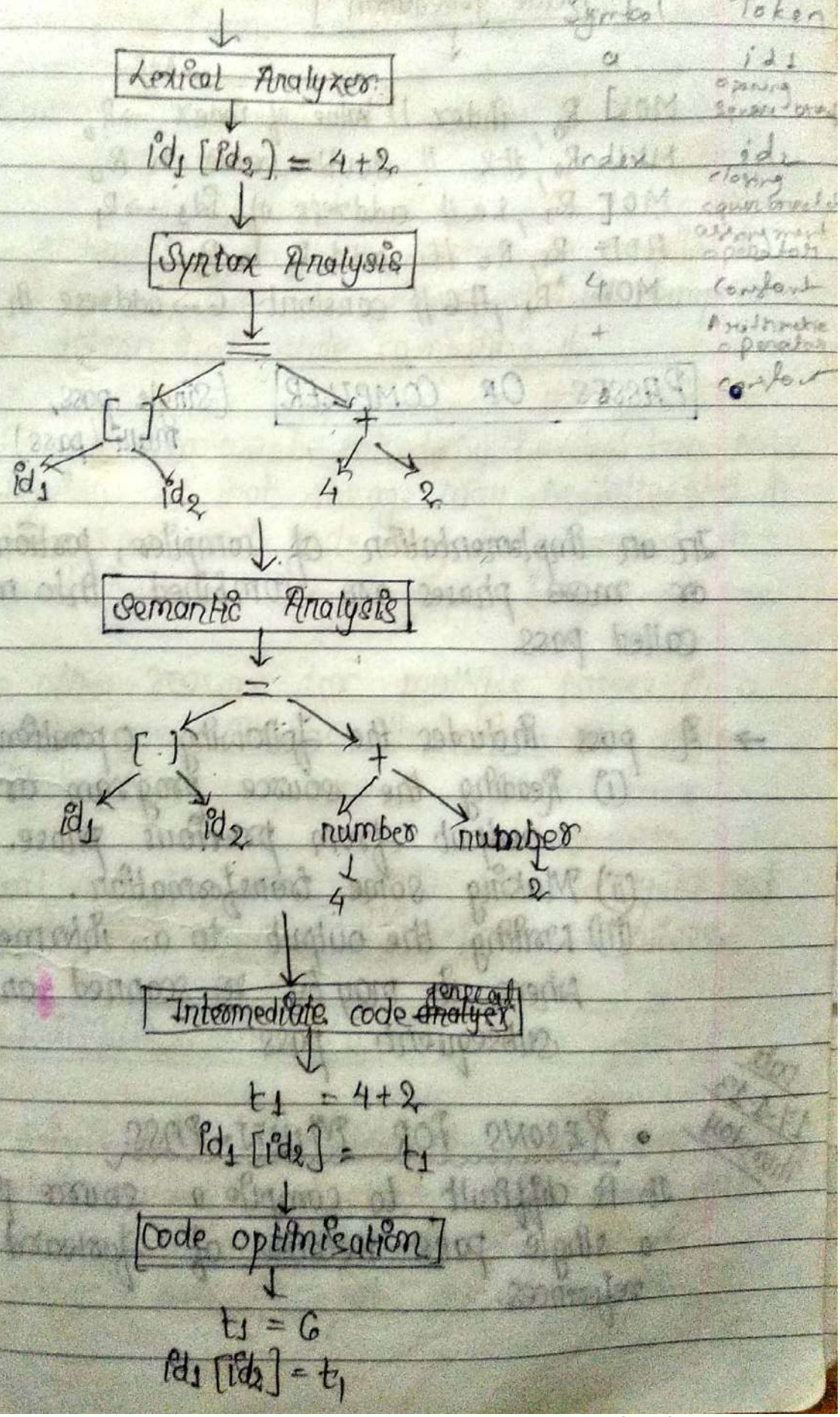
↓

```
MOV  R1, id2
ADD  R1, id3
MUL  R1, R1
MUL  R1, #2.0
MOV  id1, R1.
```



Ques: a [index] = 4+2.

Solve:



↓

[Code generation]

↓

= 1+2.

```
MOV R0, index  // value of index → R0
MUL R0, #2     // double value in R0
MOV R1, &a     // address of id1 → R1
ADD R1, R0     // add R0 to R1
MOV *R1, #6    // constant 6 → address in R1.
```

- **PASSES OF COMPILER** (Single pass, Multi pass)

In an implementation of compiler, portion of one or more phases are combined into modules called pass.

→ A pass includes the following operations:
(i) Reading the source program or output from previous phase.
(ii) Making some transformation.
(iii) Writing the output to an intermediate file where it may be re-scanned for a subsequent pass.

Date 17-1-13 Thursday

- REASONS FOR MULTI-PASS:

It is difficult to compile a source prog in a single pass because of forward references.

```
a := y
goto jump;
    .
    .
    .
jump z=a;
```

The statement "goto jump" cannot be compiled in one pass since the address of label jump cannot be determined while compiling it.

This problem can be solved by having two-pass compiler, so that storage may be allocated in first pass and code may be generated in second pass.

→ The other reasons for multiple passes in a compiler are. Storage limitation and code optimisation.

→ Pascal and C uses single pass compiler and modulo-2 requires two pass compilers.

- **SINGLE PASS COMPILER**

In single pass compiler all the phases (lexical, syntax analysis, semantic, intermediate code generator, code optimization and code generator) are grouped into one pass.

• MULTI-PASS COMPILER

→ A multi pass compiler consists of several different passes where each pase perform a well defined function of a compiler.

→ For ex: The first pass can read the input source & scan and extract the token and store the result in an intermediate file.

→ The second pass can read the file which was produced in the first and do the syntactical analysis.

→ The output of the second pass is a syntax tree.

→ The third pass can read the output file produced by second pass and perform the optimization.

• DIFFERENCE BETWEEN SINGLE PASS AND MULTI PASS

SINGLE PASS	MULTI PASS
(i) In a single pass compiler all the phases of compiler are grouped in one pass.	(i) In a multi-pass compiler the different phases of compiler are grouped into multiple passes.

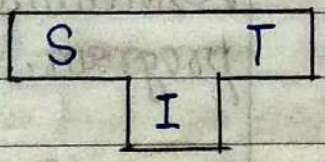
SINGLE PASS	MULTI PASS
(i) It is faster.	(ii) It is slower.
(iii) It consumes more space	It consumes less space.
(iv) Less overheads are involved.	More overheads are involved.
(v) No need to keep track of several intermediate files, as intermediate file is written only once.	It involves unnecessary reading and writing of several intermediate files, as intermediate files are updated after each pass.
(vi) It imposes certain restrictions on the program.	It does not imposes any restrictions on the program.
(vii) A single pass compiler may not allow for better provision of code optimisation, error reporting and error handling.	A multi pass compiler may allow for better provision of code optimisation, error reporting & handling.
(viii) Pascal & C requires one-pass.	Modula-2 requires two-passes

CROSS COMPILER

- ~~T diagram:~~
- T-diagram:

A compiler is characterised by three languages:
(i) Source language (S) that it compiles.
(ii) Target language (T) that it generates code, for its own machine or other machine.
(iii) Implementation language (I) that it is written in.

→ We represent the three languages by using the diagram called T-diagram.



$C_I^{S\ T}$

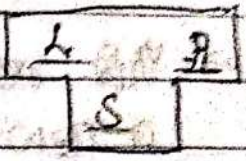
Cross Compiler is a compiler that runs on one machine and produces code for another machine i.e $C_x^{\ y}$; $x \neq y$.

→ The cross compiler is used to implement the compiler which is characterised by three languages:
(i) Source Language.
(ii) Object Language.
(iii) The language in which it is written.

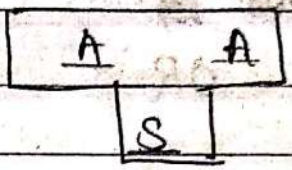
Cross compilers are widely used in mini and micro computers.

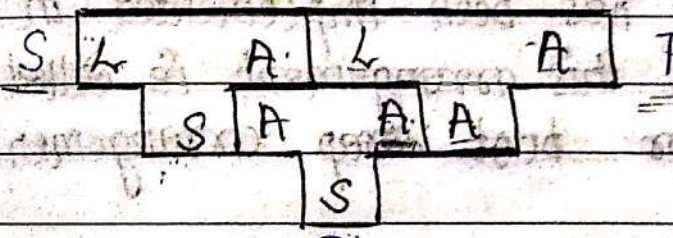
Ques: Create a compiler with the help of cross compiler $C^{L}_{S}{}^{A}$ using previously known language $C^{A}_{S}{}^{A}$

Solve: We have T-diagram for $C^{L}_{S}{}^{A}$



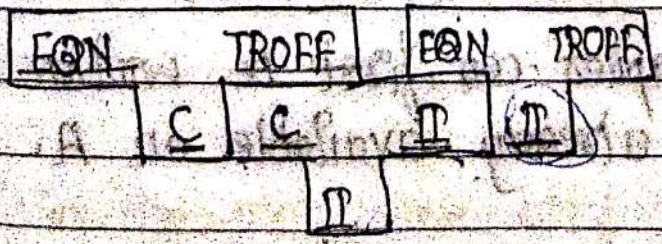
and $C^{A}_{S}{}^{A}$





Ques: Create a cross-compiler for EQN. Using C-compiler written in & PDP-II, producing code in PDP-II and a EQN language producing code for text formatter, TROFF written in C.

Solve:



- **BOOT STRAPPING**

Boot strapping is a concept of obtaining a compiler for a new language by using compiler for less powerful version that is subset of the same language.

→ This reduces the amount of work in implementing a new compiler

OR

If a compiler has been implemented in its own language then this arrangement is called a Boot strap or boot strap arrangement.

→ Suppose we have a new language L for which we want to write a compiler which runs on machine A. As a first step we might write for machine A a small compiler $C^{S}_{A}{}^{A}$, that translates a subset S of language L into the machine or assembly code of A.

→ This compiler can first be written in a language that is already available on A.

→ We then write a compiler $C^{L}_{S}{}^{A}$ in the simple language S.

This program when run through $C_A^{S\,A}$, becomes $C_A^{L\,A}$, the compiler for the complete language L running on machine A and producing object code for A.

→ The process is as follows:

$$C_S^{L\,A} \longrightarrow C_A^{S\,A} \longrightarrow C_A^{L\,A}$$

• COMPILER CONSTRUCTION TOOLS

Some general tools have been created for the automatic design of specific compiler design components.

→ These tools use specialised languages for specifying and implementing the component, and many use algorithm that are quite sophisticated.

→ The most successful tools are those that hide the details of generation algorithms and produce components that can be easily integrated into the remainder of a compiler.

→ The following are some useful compiler construction tools:

(1)

(1) PARSER GENERATORS:

→ This produces syntax analyser normally from input that is based on a context free grammer.

→ Initially, in compilers syntax analysis consumes not only a large fraction of running time of a compiler but also a large fraction of the intellectual effort of writing a compiler.

→ This phase is now considered one of the easiest to implement.

(2) SCANNER GENERATORS:

→ This automatically generates lexical analyzers normally ~~based~~ from a specification based on regular expressions.

(3) SYNTAX DIRECTED TRANSLATION ENGINES:

→ This produce collection of routines which generate intermediate code through parse tree.

(4) AUTOMATIC CODE GENERATOR:

→ Such tools takes a collection of rules that define the translation of each operation

of intermediate language into the machine language for the target machines.

(5) DATA FLOW ENGINES:

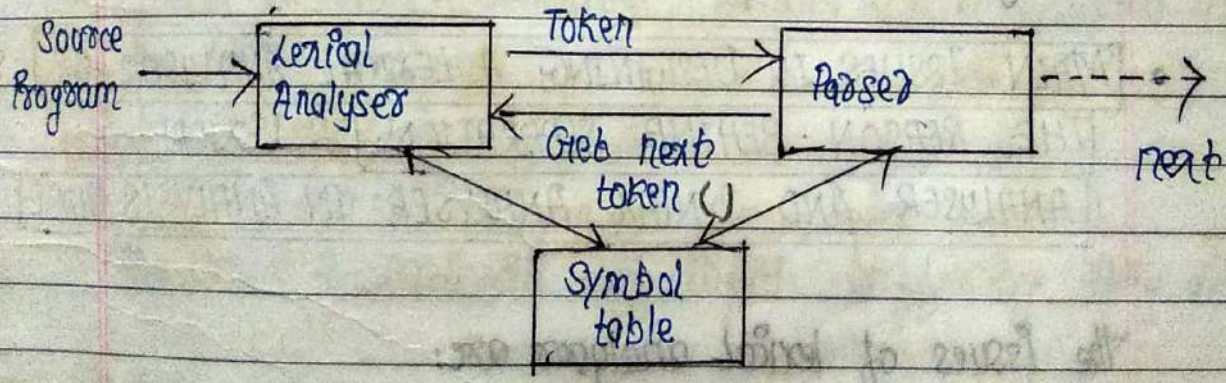
→ Much of the information needed to perform good code optimisation involves "data flow analysis", the gathering of information about how values are transmitted from one part of a program to each other part.

Date 18-Jan-13 Friday

CD

• THE ROLE AND FUNCTION OF LEXICAL ANALYSER

The lexical analyser is the first phase of a compiler. Its main task is to read the input characters and produce output a sequence of tokens that the passer uses for syntax analysis. This interaction is shown as follows:



upon recieving a "get next token" command from the passer, the lexical analyser reads input characters until it can identify the next token. Since the lexical analyser is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface these are:-

(i) Stripping out from the source program comments and white space in the form of blank, tab and new line characters, etc.

(ii) Correlating error messages from the compiler with the source program.

For ex: The lexical analyser may keep track of the no. of new line characters seen, so that a line no can be associated with an error message.

The main functions of lexical analyser are,

(a) Removal of white space.
(b) Removal of comments.
(c) Handling constants.
(d) Handling operators.
(e) Recognisation of identifiers and keywords.

• MAIN ISSUES IN DESIGNING A LEXICAL ANALYSER OR THE REASON BEHIND SEPARATION OF LEXICAL ANALYSER AND SYNTAX ANALYSER IN ANALYSIS PHASE

The issues of lexical analyser are,

(i) Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax analysis often allow us to simplify one or the other of this phases.

For ex:- A parser combining the conventions for comments and white space is significantly more than complex

than one that can assume comments and white space have already been removed by a lexical analyser.

(ii) Compiler efficiency is improved. A separate lexical analyser allows us to construct a specialised and potentially efficient processor for the task. A large amount of time is spent reading the source program and partitioning it into tokens. Specialised buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler.

(iii) Compiler portability is enhanced Input alphabet peculiarities and other device specific anomalies can be restricted to the lexical analyser. The representation of special or non-standard symbols can be isolated in the lexical analyser.

- **INPUT BUFFERING**

There are 2 methods of input buffering. They are:

1) Buffer pairs.
2) Sentinels.

P.T.O →

1) BUFFER PAIRS:

In this scheme, we use a buffer divided into 2 N character halves as shown below:

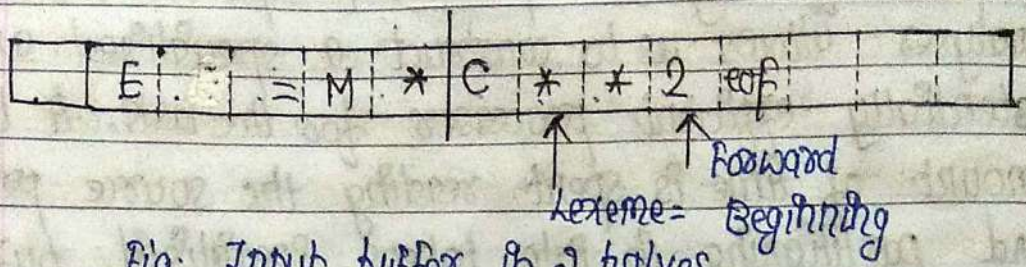


Fig: Input buffer in 2 halves.

Typically, N is the number of characters from one disk i.e 1024 or 4096.

→ We read N input characters into each half of the buffer with one system read command rather than invoking a read command for each input character.

→ If fewer than N characters remain in the input, then a special character 'eof' is coded into the buffer after the input characters, i.e eof marks the end of the source file and is different from any input character. 2 pointers to the input buffer are maintained.

i) Pointer lexeme begin, marks the beginning of the current lexeme.

(ii) Pointer forward scans a head until a pattern match is found. The string of characters betwee

the two pointers to the current lexeme. Initially both pointers points to the first character of the next lexeme to be found.

↳ forward pointer scans a head until a match for a pattern is found. Once the next lexeme is determined, the forward pointer is set to the character at its right end. After the lexeme is processed, both pointers are set to the character immediately after the lexeme.

- **ALGORITHM FOR ADVANCING FORWARD POINTER**

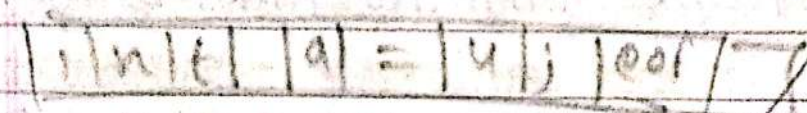
```
If forward at the end of first half then begin
        reload second half;
        forward := forward + 1;

    end
    else if forward at the end of second half
                                then begin
        reload first half;
        move forward to beginning of first half

    end
        else forward := forward + 1;
```

→ The disadvantage of these buffering scheme is amount of look ahead is limited, and this limited look ahead may make it impossible to recognise tokens in situation where the distance the forward pointer must travel and this time is more than the length of the buffer.

int a = 4;



SOME IMPORTANT DEFINITIONS

(1) **LEXME:**
These are the smallest logical units (words) of a program, such as A, B, 1.0, 7, TRUE, +, <= and etc.

(2) **TOKENS:**
They are classes of similar lexmes such as identifier, constants, operators. Hence, tokens are the category to which a lexme belongs to.

(3) **PATTERN:**
It gives a informal or formal description of a token. For ex: An identifier is a string in which the first character is an alphabet and the successive characters are either digits or alphabets.

(4) **ATTRIBUTE:**
When more than one lexme can match a pattern, the lexical analyzer must provide the subsequent compiler phases, additional information about the particular lexme that match.

For ex: E = M * C * * 2

<id, Pointer to symbol table entry for E>
<assign_op>
<id, ... M>

<multi_op>
<id, ... c>
<Exp_op>
<number, integer value 2>

- **SPECIFICATIONS OF TOKENS**

Regular expression notation can be used for specification of tokens because tokens constitutes a regular set.

→ It is compact, precise and contains a deterministic finite automata that accepts the language specified by regular expresn.

→ The DFA is used to recognise the language specified by regular expression.

→ • Regular expression notations:

(1) A string is a finite sequence of symbols. We can denote the string by 's'. If 's' is the string then the length of the string is denoted as |s|.

For ex: S = xyz.
|S| = 3.

→ If $|s| = 0$, then the string is called an empty string and we use ϵ to denote the empty string.

(2) Prefix:

→ A string's prefix is the string formed by taking any no. of leading symbols of strings.

→ For ex: If $S = ABC$, then prefix $S = \epsilon$

$S = A$

$S = AB$

$S = ABC$

→ Any prefix of a string other than the string itself is called proper prefix of the string.

(3) Suffix:

→ A string's suffix is formed by taking any number of trailing symbols of a string.

→ For ex: If $S = ABC$, then suffix, $S = \epsilon$

$S = C$

$S = BC$

$S = ABC$.

→ Similar to prefix, any suffix of the string other than the string itself is called proper suffix of the string.

(4) Concatenation:

→ If S1 and S2 are two strings then the concatenation of S1 & S2 is denoted by S1.S2.

→ Simply, a string obtained by writing S1 followed by S2 without any space between them.

→ Ex: If S1 = xyz , S2 = abc
then S1.S2 = xyzabc.

(5) Alphabet:

→ An alphabet is a finite set of symbols denoted by Σ.

(6) Language:

→ A language is a set of strings formed by using the symbol belonging to alphabet

→ Ex: Σ = {0,1} , then language L = {0,1}*

(7) Set:

→ A set is a collection of object denoted by following method:

(a) We can enumerate the members by placing them within { }.

For ex: $A = \{0, 1, 2, 3, 4, 5\}$

(b) We can use a predetermined notation in which the set is denoted as $A = \{x : p(x)\}$

This means that A is a set of all those elements x for which the predicate $p(x)$ be true.

For ex: A set of all integers divisible by 3 will be denoted as $A = \{x : x \in I \text{ and } x \text{ is divisible by } 3\}$

(8) Set Operation:

(i) $A \cup B$.
(ii) $A \cap B$.
(iii) $A - B$.
(iv) $B - A$.

21-Jan
Monday

FINITE AUTOMATA:

A recongniser for language is a program that takes a string x as an input and answers "yes" if x is a sentence of the language and "No" otherwise.

→ One can compile any regular expression into a recongniser by constructing a generalised transition diagram called finite automata.

→ A finite automata can be deterministic or non-deterministic.

→ Deterministic finite automata means not more than one transition out from a state on same i/p symbol.

→ Non-deterministic means more than one transition out of a state may be possible on a same i/p symbol.

→ Both automata are capable of what regular expressh can denote.

→ There are various components of FA:

(i) Input tape.

→ The i/p tape is divided into squares, each containing a single symbol between input end marker, ¢ (c-cut) at the left end and $ at right end.

(ii) Read Head:

→ The head examines only one square at a time and can move one square either to left or to the right.

(iii) Finite control:

→ The input to the finite control is symbol under the read head and present state of machine that gives following output.

(a) The motion of R-head.
(b) The next state of finite state machine.

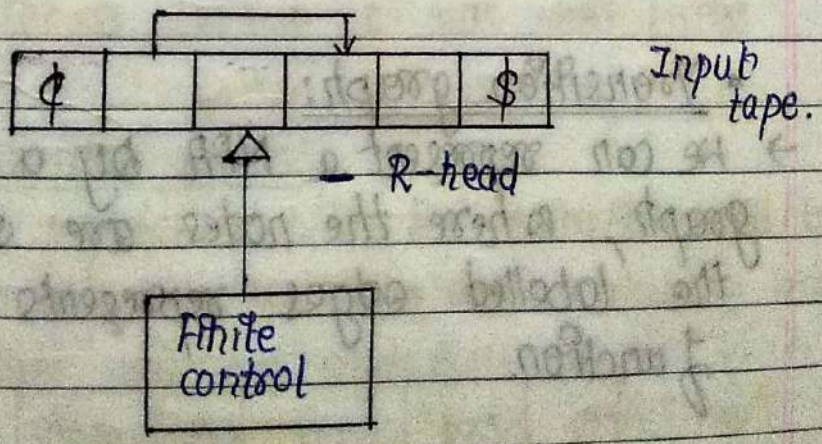


Fig: Block diagram of a finite automata.

• NON-DETERMINISTIC FINITE AUTOMATA [NFA]

→ A NFA consists of:

(i) A finite set of states S.
(ii) A set of input symbol Σ, the input alphabet. We assume that ϵ, which stands for empty string, is never a member of Σ.
(iii) A state s_0 from S, that is distinguished as the start state (initial)
(iv) A set of state 'F', a subset of S that is distinguished as the accepting state (final state)

→ We can represent a NFA by:
(a) Transition graph.
(b) Transition table.

• Transition graph:

→ We can represent a NFA by a transition graph, where the nodes are states and the labelled edges represents the transition function.

→ There is an edge labelled "a" from state 's' to 't' iff t is one of the state

from state s with input 'a'.

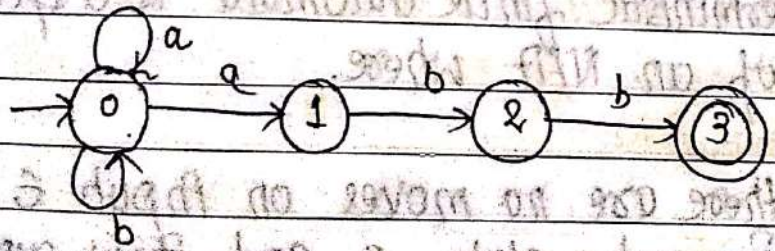
→ The transition graph for NFA has following properties:

(i) The same symbol can be labelled edges from one state to several different states.

(ii) The edge may labelled by ε, the empty string instead of or in addition to symbols from the i/p alphabet.

for ex: The transition graph for NFA recognising the language of regular expression.

(a/b) *abb



→ • Transition table:

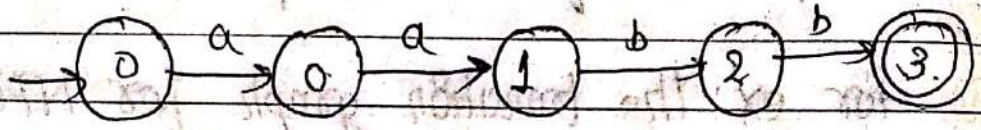
→ We can represent NFA by transition table whose rows corresponds to states and columns corresponds to i/p symbol and ε.

State / Σ	a	b	ε
0	{0,1}	{0}	ϕ
1	ϕ	{2}	ϕ
2	ϕ	{3}	ϕ
3	ϕ	ϕ	ϕ

- • Acceptance of string by Automata:

→ An NFA, accepts a input string x, iff there is any path in transition graph from start state to one of the accepting state such that the symbols along the path spell out 'x'.

→ ~~The~~ SI for ex: the string aabb accepted by above NFA is



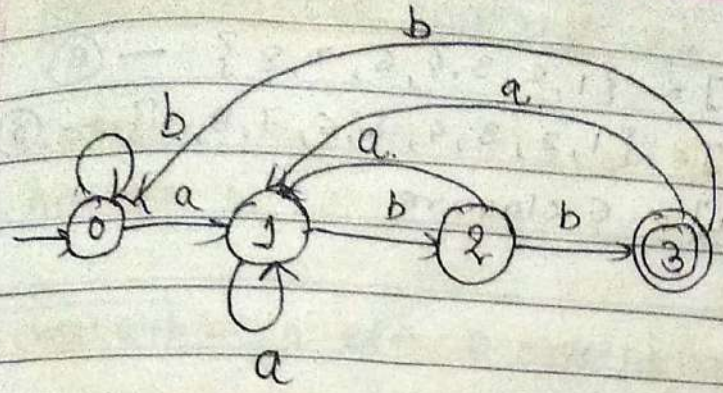
• DETERMINISTIC AUTOMATA (DFA)

→ A deterministic finite automata is a special case of an NFA where.

(i) there are no moves on input ∈.
(ii) for each state s and input symbol 'a' there is exactly one edge out of s labelled 'a'.

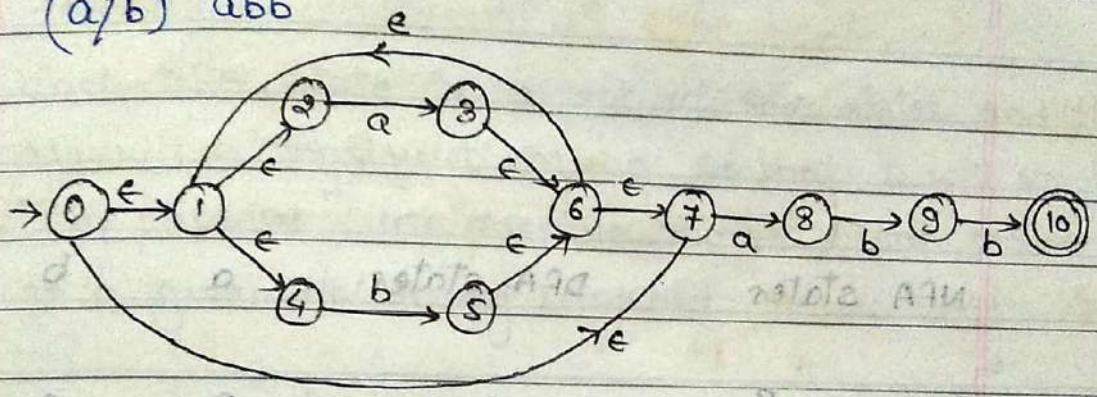
→ If we are using a transition table to represent a DFA then each entry is a single state.

→ Transition graph for '(a/b)*abb will be,



CONVERSION OF REGULAR EXPRESSION TO DFA

Que: $(a/b)^*$ abb



0 is the start state, first we find the ε for start state 0

ε closure (0) = {0, 1, 2, 4, 7} — (A)

According to algo mark A

Dtran [A, a] = ε closure (move (A, a))
= ε closure (3, 8)
= {1, 2, 3, 4, 6, 7, 8} — (B)

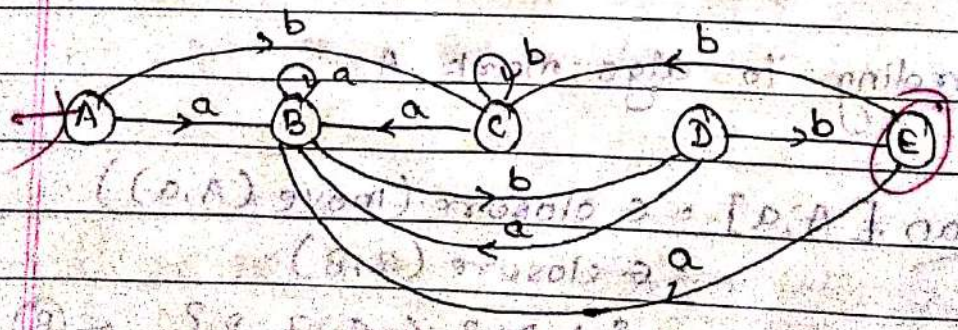
Dtran [A, b] = ε closure (move (A, b))
= ε closure (5) = {1, 2, 4, 5, 6, 7} — (C)

Dtran [B, a] = {1, 2, 3, 4, 6, 7, 8} — (B)

Dtran [B, b] = {1, 2, 3, 4, 5, 6, 7, 8, 9} — (D)

Dtran [C, a] = ∈ closure

NFA states	DFA states	a	b
{0, 1, 2, 4, 7}	A	B	C
{1, 2, 3, 4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 4, 5, 6, 7, 10}	(E)	B	C



Que: (i) ((0/1)*00)/0

(ii) (ab/(a/b))a

Date
24-JAN-13
Thursday

ALGORITHM FOR CONSTRUCTING FA FROM NDFA

INPUT

An NFA N.

OUTPUT - A DFA D accepting the same language.

METHOD

This algorithm constructs a transition table Dtran for D.

Each DFA state is a set of NFA states and this algorithm construct Dtran, so that D will simulate "in parallel" all possible moves N can make on a given i/p string.

OPERATION	DESCRIPTION
1) ϵ closure (s)	Set of NFA states reachable from NFA state S on ϵ transition.
2) ϵ closure (T)	Set of NFA states reachable from some NFA state S in T on ϵ transitions.
3) Move (T, a)	Set of NFA states to which there is a transition on i/p symbol 'a' from some NFA states S in T

ALGORITHM FOR THE SUBSET CONSTRUCTION

1) Initially, ϵ closure S_0 is the only D state and it is unmarked.

2) While (there is an unmarked state T in D states)
{
mark T ;
for (each i/p symbol a)
{
U = ϵ closure (move (T, a)) ;
if (U is not in D states)
ADD U as an unmarked state to D states
Dtran [T, a] = U ;
}
}

ALGORITHM FOR COMPUTING ϵ-CLOSURE (T)

Push all states of T onto stack
Initialise ϵ closure (T) to T
while (stack is not empty)
{
pop t, the top element, off stack ;
for (each state u with an edge from t to u labelled ϵ)
{
if (u is not in ϵ closure T)
{

ADD u to ε closure T ;
push u into stack ;
}
}
}

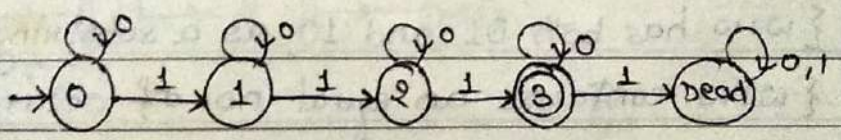
Que: Construct a DFA for each of the following RL. In all cases the alphabet is (0,1)

(a) The set of strings that has exactly three 1

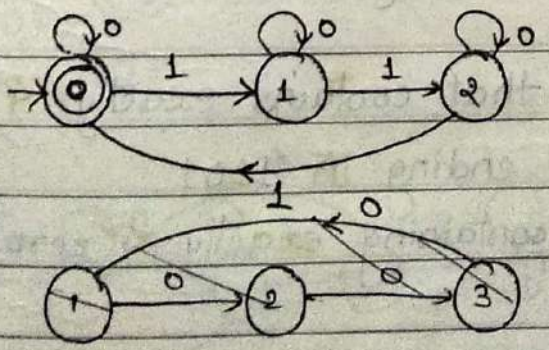
(b) The set of strings where the no. of 1 is a multiple of 3

(c) The set of strings where the difference between no. of 0's and no. of 1's is multiple of 3.

(a) $0^*10^*10^*10^*$

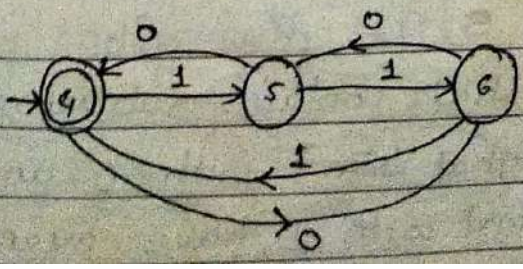


(b)



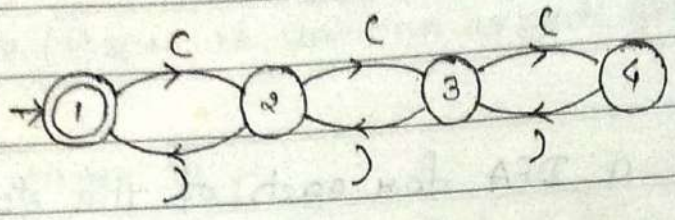
$0^*10^*10^*$

(c)



Que: Construct a DFA that recognises balanced sequence of parenthesis with a maximum nesting depth of 3.

((())) or ()()() or (())()



Que: Construct a DFA for the following languages:

(a) {w : w doesnot contain the substring 110}

(b) {w : w is any string 'a' except 11 and 111}

(c) {w : w contains atleast 2 0's and atmost one 1}

(d) {w : each zero in w is immediately preceeded and immediately followed by one 1}

(e) {w : w has neither 00 nor 11 as a substring}

(f) {w : w has both 01 and 10 as a substring}

(g) {w : w contains an equal no. of occurances of the substring 01 and 10}

(h) All strings that contains exactly 4 zeros.

(i) All strings ending in 1101.

(j) All string containing exactly 4 zeros and atleast 2 1's

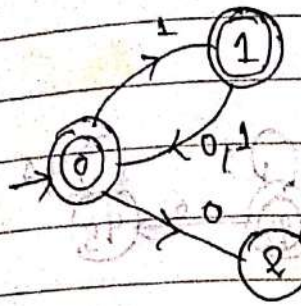
(k) All strings whose binary representation is divisible by 5.

(l) All strings that contain substring 0101.

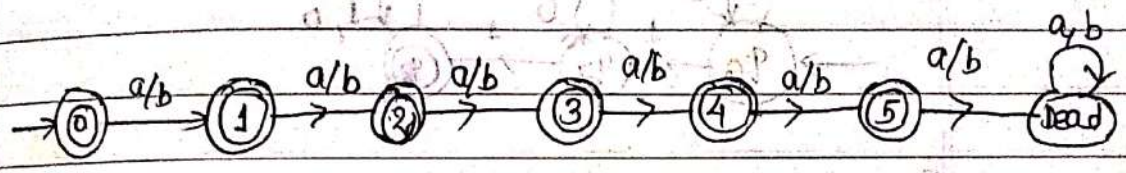
(m) All strings that start with 0 and have odd length. or start with 1, have even length.

(n) All string that donot contain the substring 110
(o) All strings of length atmost 5.
(p) All the strings where every odd position is a 1.

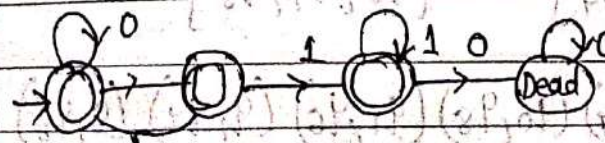
(P) 1 is must at odd position, but it is possible that 1 is at even position.



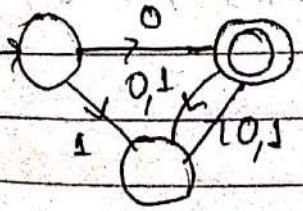
(o)



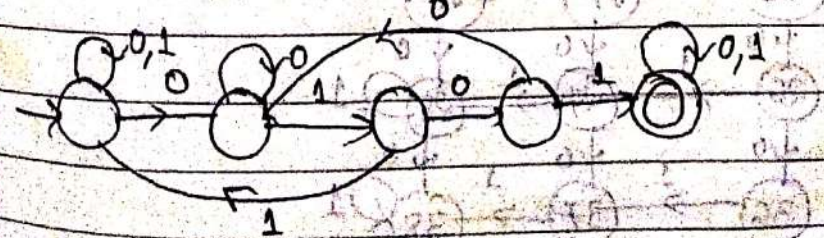
(n)



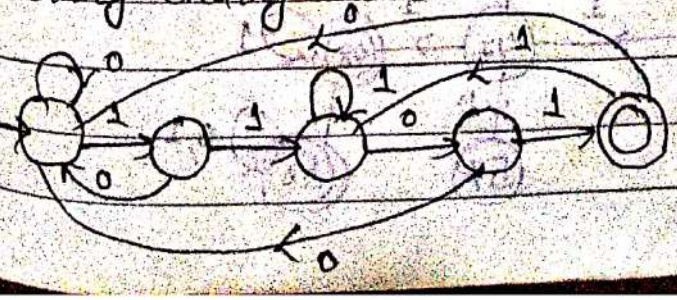
(m)



(l)



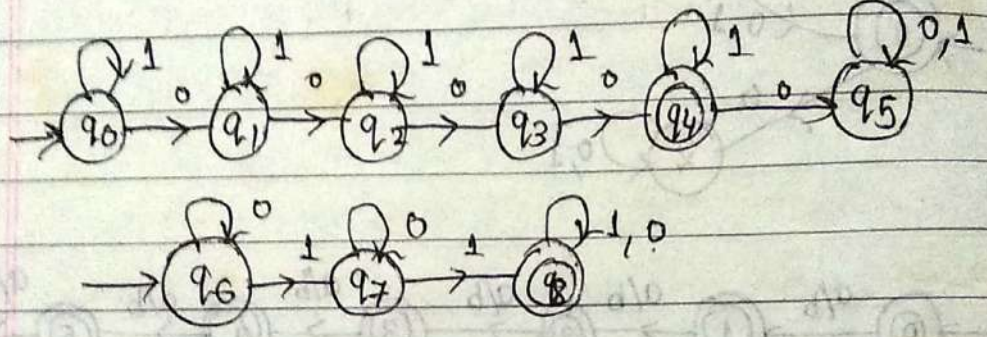
(i) All string ending with 1101



(k)

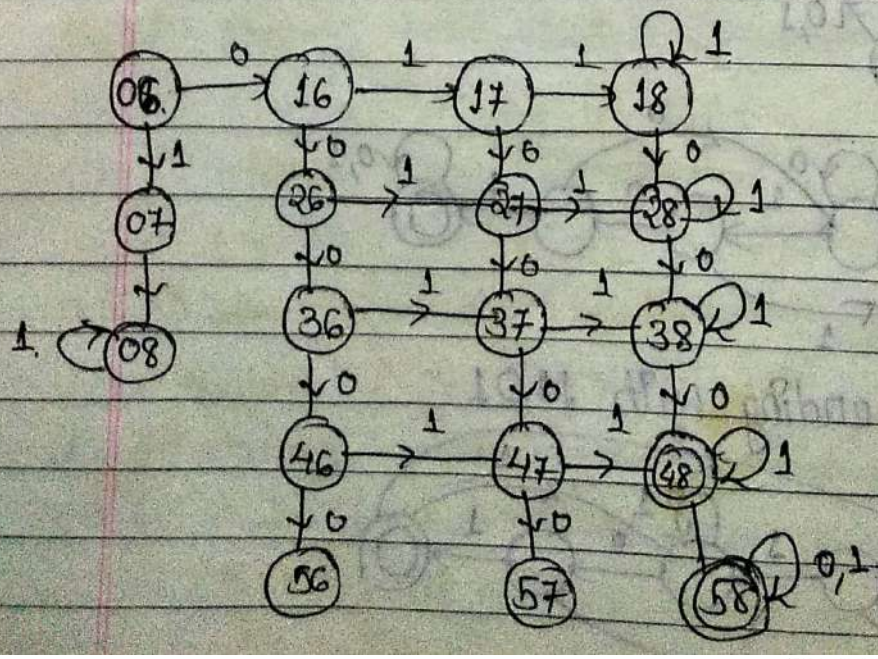
	0	1
0	0	1
1	0	1
2	1	3
3	2	4
4	3	1
	4	3	2
		4

(j)

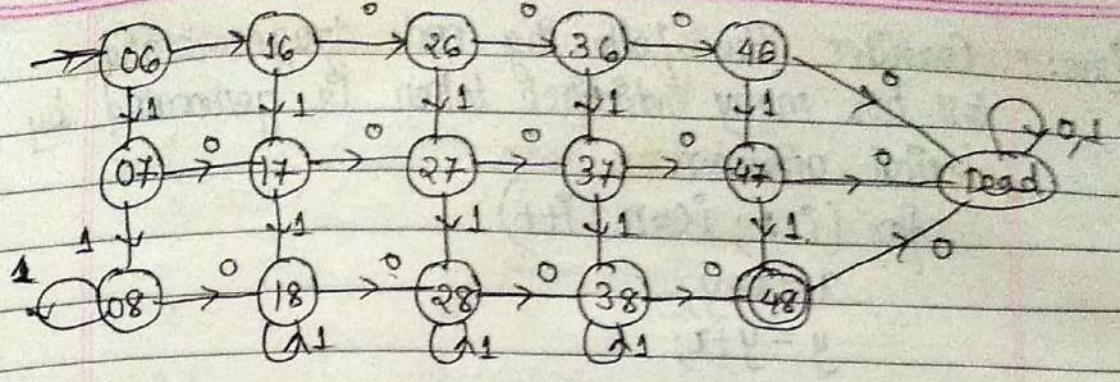


$\{q_0, q_1, q_2, q_3, q_4\} \times \{q_6, q_7, q_8\}$

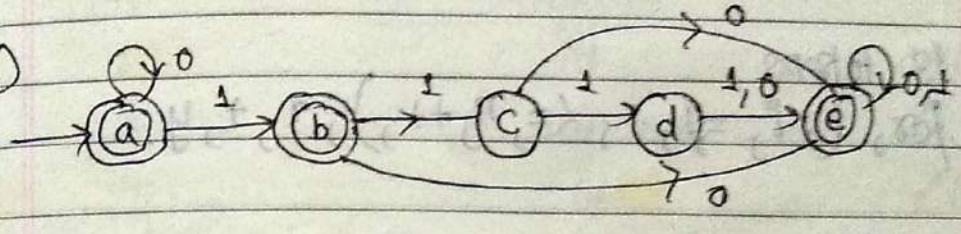
$= \{(q_0 q_6), (q_0, q_7) (q_0, q_8) (q_1, q_6) (q_1, q_7) (q_1, q_8)$
$(q_2 q_6) (q_2, 7) (2,8) (3,6) (3,7) (3,8) (4,6)$
$(4,7) (4,8)\} \{q_5\}$



OR



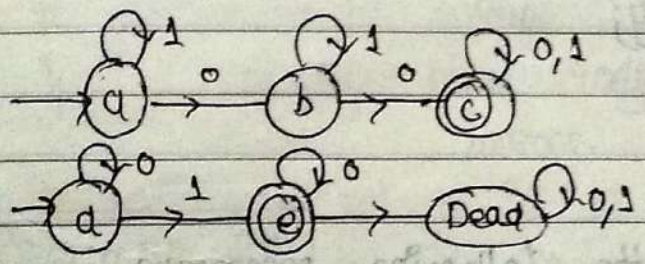
(b)



•CD

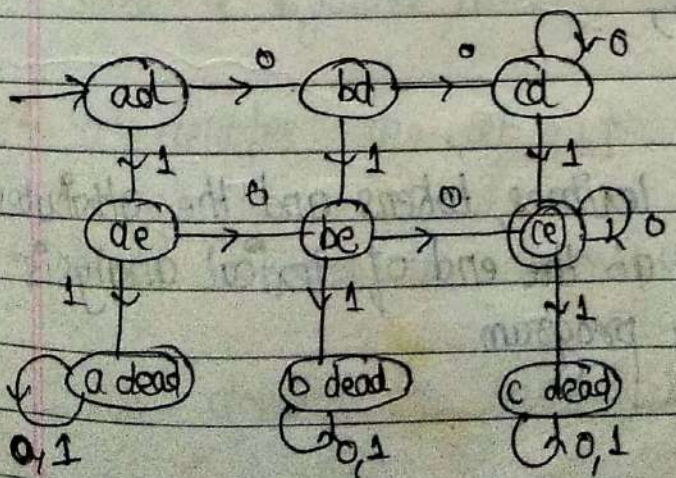
Date
30-Jan-13
Wednesday

(c) Atleast 2 zero and atmost 1 one



{a, b, c} {d, e}

{ad, ae, bd, be, cd, ce}



Ques: Consider the following code fragment and tell how many distinct token is generated by lexical analyser.

```
for (i=1; i<=n; i++)
    x = x+y;
    y = y+x;
```

13 tokens

for, (, i, =, 1, ;, i, <=, n, ++,), x, +, y.

```
main()
{
  int x, y, z;
  z = x+y;
}
```

13

Ques: Consider the following program

```
main()
{
  int x, y, z;
  z = x+y;
}
```

List down the lexemes, tokens and the attributes of the tokens at the end of lexical analysis of the above program.

LEXIMES	TOKEN	ATTRIBUTE
main	Keyword	
(	open parenthesis	
)	close	
{	open curly braces	
int	Keyword	
x	id	<id, 1>
,	comma	
y	id	<id, 2>
z	id	<id, 3>
;	semicolon.	
=	Assignment	
+	sum	
}	close curly braces.	

- **REGULAR SET**

Any set represented by regular expression is called a regular set.

- 'a' denotes the set {a}
- 'a+b' denotes the set {a, b}
- ab denotes the set {ab}
- a^* denotes the set $\{\epsilon, a, aa, aaa \ldots\}$

- $(a+b)^*$ denotes the set $\{a,b\}^*$

A regular set is a set of string for which there exist some finite automata that accepts the set.

- **REGULAR EXPRESSION**

They are useful for representing regular set.

We give a formal recursive definition of regular expression over Σ as follows:

(i) Any terminal symbol, an element of Σ, ϵ and ϕ are regular expression.

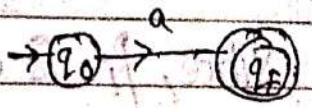
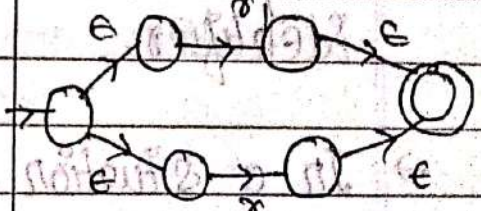
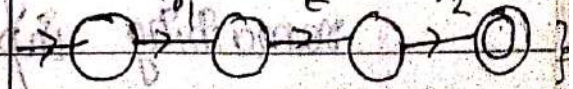
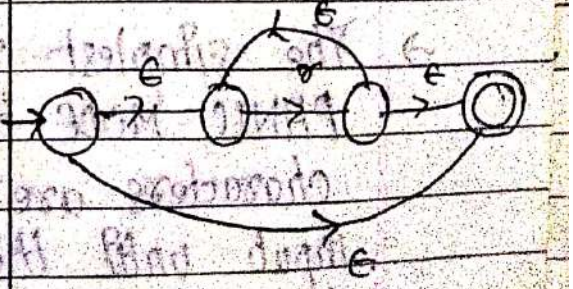
(ii) The union of 2 regular expressions R_1 and R_2 written as R_1+R_2 is also a regular expression.

(iii) The concatenation of regular expression R_1 and R_2 written as R_1R_2 is also a regular expression.

(iv) The iteration (or closure) of a regular expression R written as R^*, is also a regular expression.

(v) If R is regular expression, then (R) is also a regular expression.

Regular expression is a notation to specify a regular set. Hence, for every regular expression there exists a FA that accepts the language specified by the regular expression.

Regular Expression	Regular set	Finite Automata.
ϕ	$\{\}$	
ϵ	$\{\epsilon\}$	
a	$\{a\}$	
$r_1 + r_2$	$r_1 \cup r_2$	
$r_1 \cdot r_2$	$\{r_1 r_2\}$	
r^+	r^+	

• LEXICAL ERROR & Recovery

A lexical analyser has a localised view of a source program. If the string fi is encountered in a C-program,

```
fi (a == f(x))
```

→ A lexical analyser cannot tell whether fi is a missed spelling of the keyword 'if' or an undeclared function identifier.

→ Since, fi is a valid identifier, the lexical analyser must return token for an identifier.

→ In a situation when the lexical analyser enable to recieve as because none of the token matching a prefix of the remaining input.

→ The simplest recovery technique is PANIC MODE RECOVERY where successive characters are deleted from remaining input until the lexical analyser can find a well formed token.

→ This recovery technique may occasionally confuse the parser, but in an interactive computer environment it may be quite sufficient.

Other possible error recovery actions are:

(1) Deleting an extraneous character.

(2) Inserting a missing character.

(3) Replacing an incorrect character by correct character.

(4) Transposing two adjacent characters.

ERROR HANDLING

Programs are written by programmer and hence contain many errors. It is the function of compiler to detect or report error in source program.

→ The error handling is performed by a program called error handler.

→ Whenever a phase of the compiler discovers an error, it must report an error to the error handler, which issues an appropriate message.

→ The various error encountered by different phases of compiler can be:

(i) The lexical analysis may detect a illegal or unrecognised character or mis-spelled token.

(ii) The parser may detect a syntax error. for ex: Missing parenthesis.

(iii) The semantic analyser may detect a syntatic error. for ex: Type mis-match.

(iv) The intermediate code generator may detect an operator with incompatible types.

(v) The code optimiser may detect certain unreachable statements.

(vi) The code generator may detect a too large constant to fit in a word of the target machine.

• UNIT-2

SYNTAX ANALYSIS

• CFG (Context free Grammer)

It specify a context free language that consists of terminal, non-terminals, starting symbol and production rule.

→ The terminals are nothing more than token of the language used to form the language construct.

→ Non-terminals are the variables that denotes a set of strings.

→ The context free grammer is denoted by 4 tupples:

$$G_f = \{V_N, P, S, \Sigma\}$$

where, V_N is a finite set of symbols called as non-terminals or variable.

Σ is a set of symbol called as terminal.

P is a set of production rules
S is a member of V_N called as start symbol.

For ex: $G = (\{S\}, \{a, b\}, P, S)$

Where P consists of

$S \rightarrow aSa$

$S \rightarrow bSb$

$S \rightarrow \epsilon$

- **DERIVATION**

Derivation refers to replacing an instance of a given strings non-terminal by the RHS of the production rule, whose LHS contains the non-terminal to be replaced.

→ Derivation produces a new string, therefore, derivation can be used repeatedly to obtain a new string from a given string.

→ If the string obtained as a result of the derivation contains only terminal symbols, then no further derivations are possible.

→ There are two types of derivations:
(i) Left most Derivation
(ii) Right " "

(i) Left-most derivation:

→ In this derivation, left-most non-terminal is considered first for derivation at every stage in the derivation process.

→ Consider the following grammer:

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id.$

LMD for string id+id*id.

$E \rightarrow E+E \rightarrow id+E$

$\rightarrow id+E*E$

$\rightarrow id+id*E$

$\rightarrow id+id*id.$

(ii) Right-most derivation:

→ In this derivation the rightmost non-terminal is considered first.

for ex: RMD for string id+id*id

$E \rightarrow E*E$

$\rightarrow E*id$

$\rightarrow E+E*id$

$\rightarrow E+id*id$

$\rightarrow id+id*id.$

• AMBIGOUS GRAMMAR

If more than one parse tree exists for same string in grammer, then grammer is said to be an ambiguous grammer.

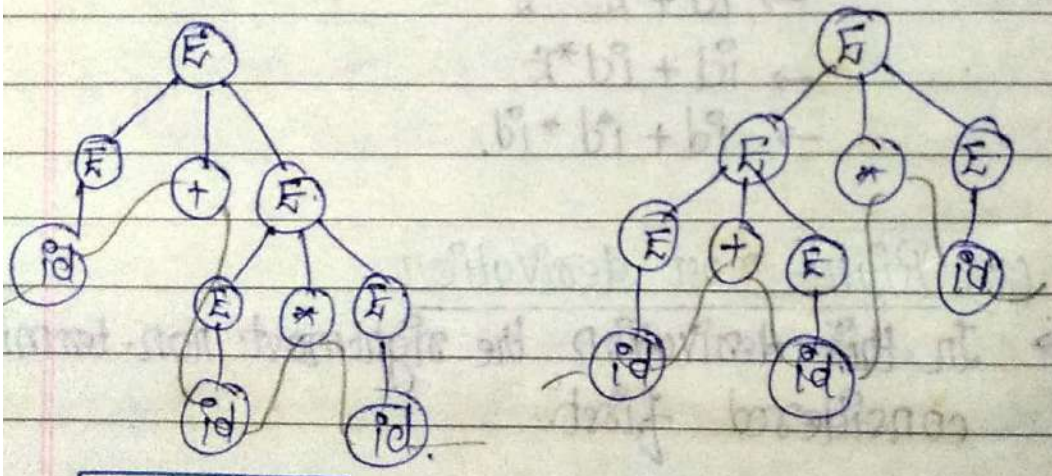
→ Consider the following grammer whose productions are:

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow id$

is ambigous because there exists more than one parse tree for a string id + id * id.



• PARSING

Parsing is the process of obtaining a string of tokens from the lexical analyser and verifies whether or not the string is valid construct of source language i.e whether or not it can be generated by the grammer for the source language and for this the parser either attempts to derive

the string of tokens w from start symbol S, or it attempts to reduce w to start symbol of the grammer. by tracing the derivation of w in reverse.

→ Parsing has two major techniques:

(i) TOP-DOWN PARSING:

The parse tree is constructed from root to leaf using a left-most derivation.

(ii) BOTTOM-UP PARSING:

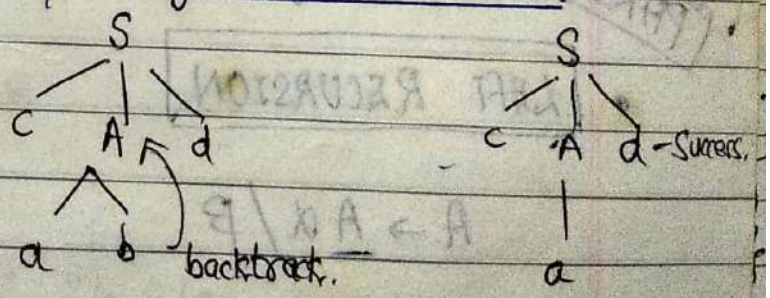
Parse tree is constructed from leaf to root using reduction process employing handle pruning technique.

•CD

Date
2-Feb-13
Saturday

- Recursive descent parsing with backtrack.

$S \rightarrow cAd$
$A \rightarrow ab/a$
$w = cad.$



Backtracking

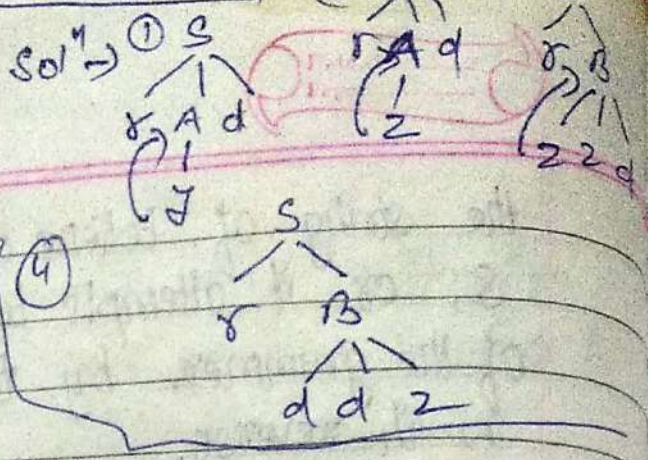
$S \rightarrow rAd / rB$

$A \rightarrow \gamma / z$

$B \rightarrow zzd / ddz$

$w = rdd[z]$

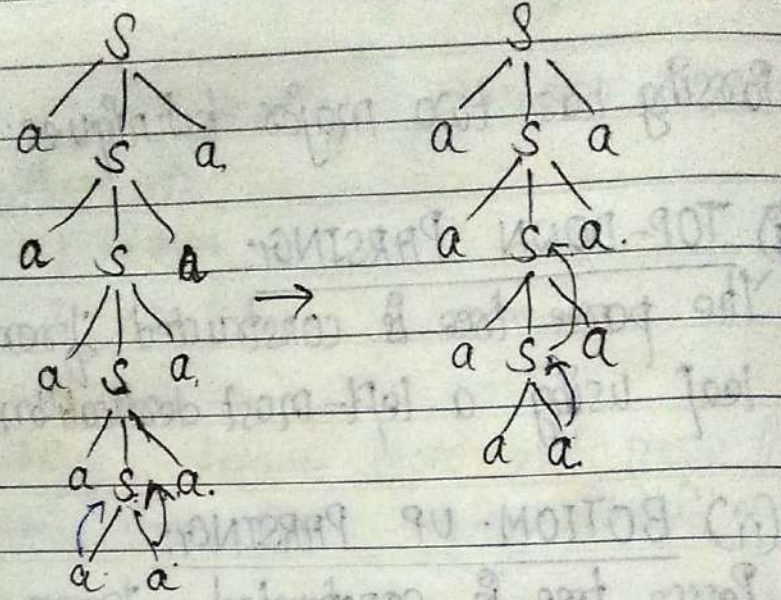
Sol^n → ① ② ③ ④

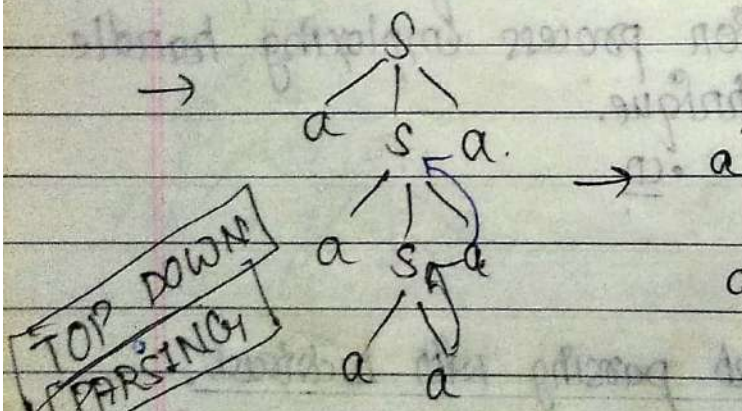


Ex. $S \rightarrow aSa / aa$

$w = aaaa.$

Solve:





TOP DOWN PARSING

- **LEFT RECURSION.**

$A \rightarrow \underline{A}\alpha / \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \epsilon$

Formula.

Ques: 1. $E \to E+T/T$
$T \to T*F/F$
$F \to (E)/id.$

Solve: $E \to TE'$
$E' \to +TE'/\epsilon$

$T \to FT'$
$T' \to *FT'/\epsilon$
$F \to (E)/id.$

• CD

Date 4-2-13 Monday

Q. 2. $S \to aBDh$
$B \to Bb/c$
$D \to EF$
$E \to g/\epsilon$
$F \to f/\epsilon$

Solve: $S \to aBDh$
$B \to cB'$
$B' \to bB'/\epsilon$
$D \to EF$
$E \to g/\epsilon$
$F \to f/\epsilon$
$A' \to$

- **LEFT FACTORING**

When the starting symbols of all alternate expression is same.

$A \to a\alpha / a\beta$

then, $A \to aA'$
$A' \to \alpha / \beta.$

$A \to a\alpha / a\beta / \phi$
$A \to aA' / \gamma$
$A' \to \alpha / \beta /$

Ques: $S \to abBc / abD\gamma$
$B \to b / \epsilon$
$D \to d / \epsilon$

Solve: $S \to abS'$
$S' \to Bc / D\gamma$
$B \to b / \epsilon$
$D \to d / \epsilon$

Parser
- Top Down (root leaf) — Backtracking / Non backtracking
- Bottom up (leaf-root)

Recursive descent parsing (implements by predictive parsing)

for implementing predictive parsing we know
1. Left Recursion
2. Left factoring
3. first
4. follow.

- **CONSTRUCTION OF FIRST( )**

Ques: 1 $S \to aA / bB$
$A \to iC / mC$
$B \to pP / eC$
$P \to g / f$
$C \to c.$

Solve: $first(S) = first(aA) \cup first(bB)$
$= first(a) \cup first(b)$
$= \{a\} \cup \{b\} = \{a, b\}$

first(A) = first(ic) ∪ first(mC)
= first(i) ∪ first(m)
= {i} ∪ {m} = {i, m}

first(B) = first(pP) ∪ first(eC)
= first(p) ∪ first(e)
= {p} ∪ {e} = {p, e}

first(P) = {g, f}
first(C) = {c}.

Ques: 2 S → aABC
A → x
B → y
C → z.

Solve: first(S) = first(aABC) = first(a) = {a}.
first(A) = first(x) = {x}
first(B) = first(y) = {y}
first(C) = first(z) = {z}

Ques: 3 S → ABC
A → x
B → y
C → z

Solve: first(S) = first(ABC) = first(A) = {x}
first(A) = first(x) = {x}
first(B) = first(y) = {y}

Ques:4 $S \to ABC$

$A \to x / \epsilon$

$B \to y$

$C \to z$

Solve: $First(C) = First(z) = \{z\}$

$First(B) = First(y) = \{y\}$

$First(A) = First(x) \cup First(\epsilon) = \{x, \epsilon\}$

$First(S) = First(ABC)$

$= First(A) - \{\epsilon\} \cup First(B)$

$= \{x, \epsilon\} - \{\epsilon\} \cup \{y\}$

$= \{x, y\}$

Ques: 5 $S \to ABC$

$A \to x / \epsilon$

$B \to y / \epsilon$

$C \to z$

Solve: $First(C) = First(z) = \{z\}$

$First(B) = First(y) \cup First(\epsilon) = \{y, \epsilon\}$

$First(A) = First(x) \cup First(\epsilon) = \{x, \epsilon\}$

$First(S) = First(ABC)$

$= First(A) - \{\epsilon\} \cup First(B) - \{\epsilon\} \cup First(C)$

$= \{x, \epsilon\} - \{\epsilon\} \cup \{y, \epsilon\} - \{\epsilon\} \cup \{z\}$

$= \{x, y, z\}$

Ques: 6 $S \to ABC$

$A \to x/\epsilon$

$B \to y/\epsilon$

$C \to z/\epsilon$

Solve: $First(C) = First(z) \cup First(\epsilon) = \{z, \epsilon\}$

$First(B) = First(y) \cup First(\epsilon) = \{y, \epsilon\}$

$First(A) = First(x) \cup First(\epsilon) = \{x, \epsilon\}$

$First(S) = First(ABC)$

$= First(A) - \{\epsilon\} \cup First(B) - \{\epsilon\} \cup First(C)$

$= \{x, \epsilon\} - \{\epsilon\} \cup \{y, \epsilon\} - \{\epsilon\} \cup \{z, \epsilon\}$

$= \{x, y, z, \epsilon\}$

Ques: 7 $S \to ABCD$

$A \to x/\epsilon$

$B \to y/\epsilon$

$C \to z/\epsilon$

$D \to d.$

Solve: $First(S) = First(ABCD)$

$= First(A) - \{\epsilon\} \cup First(B) - \{\epsilon\} \cup First(C) - \{\epsilon\} \cup First(D)$

$= \{x, \epsilon\} - \{\epsilon\} \cup \{y, \epsilon\} - \{\epsilon\} \cup \{z, \epsilon\} - \{\epsilon\} \cup \{d\}$

$= \{x, y, z, d\}$

* Digits are terminals

Ques: 8 $S \to ABCh$

$A \to x / \epsilon$

$B \to y / \epsilon$

$C \to z / \epsilon$.

Solve: $first(S) = first(ABCh)$

$= first(A) - \{\epsilon\} \cup first(B) - \{\epsilon\} \cup first(C) \cup first(h)$

$= \{x, y, z, h\}$

Ques: 9 $E \to 10 * T / 5 + T$

$T \to PS$

$S \to QP / \epsilon$

$Q \to + / *$

$P \to a / b / c$.

Solve: $first(P) = \{a, b, c\}$

$first(Q) = \{+, *\}$

$first(S) = \{A, +, a, b, c\}$

$= first(QP) \cup first(\epsilon)$

$= \{+, *, \epsilon\}$

$first(T) = first(PS) = first(P)$

$= \{a, b, c\}$

$first(E) = first(10) \cup first(*) \cup first(T) \cup$

$= \{10, 5\}$

Ques: 10 S → AbC / ad
A → eS / ∝ / ϵ
C → f / p / ϵ

Solve: first(C) = {f, p, ϵ}
first(A) = first(eS) ∪ first(∝) ∪ first(ϵ)
= {e} ∪ first(C) − {ϵ} ∪ first(∝) ∪ first(ϵ)
= {e, f, p, ∝, ϵ}
first(S) = first(A) − {ϵ} ∪ first(b) ~~∪ first(C)~~
∪ first(a)
= {e, f, p, ∝, b, a}

Date
7-Feb-13
Thursday

• CD

• CONSTRUCTION OF FOLLOW( )

Ques: S' → 1S#
S → qABC
A → a / bbD
B → a / ϵ
C → b / ϵ
D → c / ϵ
→ First Function:

Solve: First(D) = {c, ϵ}
First(C) = {b, ϵ}
First(B) = {a, ϵ}
First(A) = {a, b}
First(S) = {q}
First(S') = {1}

→ Dollar ($) is included always for starting symbol.

Follow(S') = {$}

Follow(S) = First(#) → {#}

Follow(A) = First(B) - {ε} ∪ First(C) - {ε} ∪ Follow(S)
= {a, ε} - {ε} ∪ {b, ε} - {ε} ∪ {#}
= {a, b, #}

Follow(B) = First(C) - {ε} ∪ Follow(S)
= {b, ε} - {ε} ∪ {#}
= {b, #}

Follow(C) = Follow(S) = {#}
Follow(D) = Follow(A) = {a, b, #}

Solve:1
Follow(S) = {$}
Follow(A) = Follow(S) = {$}
Follow(B) = Follow(S) = {$}
Follow(P) = Follow(S) = {$}
Follow(C) = Follow(S) = {$}

Solve:2
Follow(S) = {$}
Follow(A) = First(B) = {y}
Follow(B) = First(C) = {z}
Follow(C) = Follow(S) = {$}

Solve:3
Follow(S) = {$}
Follow(A) = First(B) = {y}
Follow(B) = First(C) = {z}
Follow(C) = Follow(S) = {$}

solve: 4 follow(S) = {$}
follow(A) = first(B) = {y}
follow(B) = first(C) = {x}
follow(C) = follow(S) = {$}

solve: 5 follow(S) = {$}
follow(A) = first(B) - {ε} ∪ first(C)
= {y, ε} - {ε} ∪ {x}
= {y, x}
follow(B) = first(C) = {x}
follow(C) = follow(S) = {$}

solve: 6 follow(S) = {$}
follow(A) = first(B) - {ε} ∪ first(C) - {ε} ∪ follow(S)
= {y, x, $}

follow(B) = first(C) - {ε} ∪ follow(S)
= {x, $}
follow(C) = follow(S) = {$}

Solve: 7 follow(S) = {$}
follow(A) = first(B) - {ε} ∪ first(C) - {ε} ∪ first(D)
= {y, x, d}
follow(B) = {x, d}
follow(C) = {d}
follow(D) = follow(S) = {$}

Solve:8 : follow(S) = {$}
follow(A) = {y, z, h}
follow(B) = {z, h}
follow(C) = {h}

Solve:10 : follow(S) = {$} ∪ follow(A) = {$, b}
follow(A) = {b}
follow(B) = first(γ) ∪ follow(S) = {a, $, b}

Solve:9 : follow(E) = {$}
follow(T) = follow(E) = {$}
follow(S) = follow(T) = {$}
follow(Q) = follow(P)
follow(P) = first(S) - {ε} ∪ follow(T) ∪ follow(S)
= {+, *} ∪ {$} ∪ {$}
= {+, *, $}
follow(Q) = follow(P) first(P)
= {a, b, c}

** Steps:

(1) Left factoring
(2) Left Recursion
(3) first()
(4) follow().

Date 8-2-13 Friday.

Ques: Modify the following CFG so as to make it suitable for top-down passing. Construct LL(1) parser for modified CFG. Show moves made by this LL(1) parser on input id + id * id.

$E \rightarrow E+T/T$

$T \rightarrow T*F/F$

$F \rightarrow (E)/id.$

Solve: • Elimination of Left Recursion:

$A \rightarrow A\alpha/\beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A'/\epsilon$

$\Rightarrow E \rightarrow E+T/T$

$E \rightarrow TE'$

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

$\Rightarrow T \rightarrow T*F/F$

$T \rightarrow FT'$

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

After eliminating the left recursion, the grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E)/id.$

- Construction of First()

~~First(E) → First(E)~~

~~First(E) = First(T)~~

First(E) = { (, id }

First(E') = { +, ε }

First(T) = { (, id }

First(T') = { *, ε }

First(F) = { (, id }

- Construction of Follow()

Follow(E) = First( ) U $ = {), $ }

Follow(E') = ~~First(~~ Follow(E) U Follow(E')

= {), $ }

Follow(T) = First(E') - {ε} U Follow(E)

= { + } U {), $ }

= { +,), $ }

Follow(T') = Follow(T) U Follow(T')

= { +,), $ }

Follow(F) = ~~Follow(~~ First(T') - {ε} U Follow(T)

= { *, ε } - { ε } U { +,), $ }

= { *, +,), $ }

	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$E' \to +TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F			$F \to (E)$		$F \to id$	

Table does not contains multiple entries. Thus, grammer is LL(1).

Let the string be id + id * id.

Stack	Input string	Moves
$E	id + id * id $	
$E'T	id + id * id $	$E \to TE'$
$E'T'F	id + id * id $	$T \to FT'$
$E'T'id	id + id * id $	$F \to id$
$E'	+ id * id $	$T' \to \epsilon$
$E'T+	+ id * id $	$E' \to +TE'$
$E'T	id * id $	
$E'T'F	id * id $	$T \to FT'$
$E'T'id	id * id $	$F \to id$
$E'T'	* id $	$T' \to *FT'$
$E'T'F*	* id $	
$E'T'	id $	$F \to id$
$E'T'id	id $	
$E'	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$

Ques: $S \to aABb$ $\omega = ab$.
$A \to c/\epsilon$
$B \to d/\epsilon$

Solve: Construction of First().
First(S) = First(a) = {a}
First(A) = First(c) ∪ First(ϵ) = {c, ϵ}
First(B) = {d, ϵ}

Construction of Follow()
Follow(S) = {$}
Follow(A) = First(B) − {ϵ} ∪ ~~Follow(S)~~ First(b)
= {d, $, b}
Follow(B) = First(b) = {b}

	a	b	c	d	$
S	S→aABb				
A		A→ϵ	A→c	A→ϵ	
B		B→ϵ		B→d	

Stack	Input string	moves
$S	ab$	–
~~$aABb~~	~~ab$~~	S→aABb
$bB	b$	A→ϵ
$b	b$	B→ϵ
$	$	–

Ques: $S \to a / \uparrow / (T)$
$T \to T, S / S$
$\omega = (a, (\uparrow), a)$

Solve: • Elimination of left Recursion.

$A \to A\alpha / \beta$

$A \to \beta A'$, $A' \to \alpha A' / \epsilon$.

$\Rightarrow$ $T \to T, S / S$

$T \to ST'$

$T' \to , ST' / \epsilon$.

After eliminating the left recursion, the grammar is

$S \to a / \uparrow / (T)$

$T \to ST'$

$T' \to , ST' / \epsilon$.

• Construction of First()

$First(S) = First(a) \cup First(T) \cup First( )$

$= \{a, \uparrow, (\}$

$First(T) = First(S) = \{ a, \uparrow, (\}$

$First(T') = First(T') = \{ , , \epsilon \}$

• Construction of Follow()

$Follow(S) = First(T') - \{\epsilon\} \cup Follow(T) \cup follow($

$= \{ , , \$,) \}$

$follow(T) = First( ) = \{) \}$

$follow(T') = follow(T) \cup follow(T') = \{) \}$

	a	(	↑	)	,	$
S	S→a	S→(T)	S→↑			
T	T→ST'	T→ST'	T→ST'			
T'				T'→ϵ	T'→,ST'	

Stack	Input string	moves
$S	(a,(↑),a)$	-
$)T(	(a,(↑),a)$	S→(T)
$)T'S	a,(↑),a)$	T→ST'
$)T'a	a,(↑),a)$	S→a
$)T'Sx	x(↑),a)$	T'→,ST'
$)T')T'(	(↑),a)$	S→(T)
$)T')TS	↑),a)$	T→ST'
$)T')T'↑	↑),a)$	S→↑
$)T')T'	),a)$	T'→ϵ
$)T'y	),a)$	T'→,ST'
$)T'Sy	a)$	S→a
$)T'a	a)$	T'→ϵ
$)y	)$	-
$	$	

Capital

Small

Ques: $S \to A$

$A \to aB / aC / aD / AC$

$B \to bBC / f$

$C \to g / \epsilon$

$D \to d / \epsilon$

Solve: Elimination of Left Factoring

$A \to a\alpha / a\beta / a\gamma / \delta$

$A \to aA' / \delta$

$A' \to \alpha / \beta / \gamma$

$A \to aA' / AC$

$A' \to B / C / D$

∴ After removing left factoring, the grammar is

$S \to A$

$A \to aA' / AC$

$A' \to B / C / D$

$B \to bBC / f$

$C \to g / \epsilon$

$D \to d / \epsilon$

Elimination of left Recursion.

$A \to aA'A''$

$A'' \to cA'' / \epsilon.$

- Construction of First()

After removing left recursion, the grammer is.

$S \to A$

$A \to aA'A''$, $A' \to B / C / D$.

$A'' \to c\,A'' / \epsilon$

$B \to bBf / f$

$C \to g / \epsilon$

$D \to d / \epsilon$

- Construction of First()

First(S) = {a}

First(A) = {a}

First(A') = {b, g, d, f, ε}

First(A'') = {g, ε}

First(B) = {b, f}

First(C) = {g, ε}

First(D) = {d, ε}

- Construction of follow()

Follow(S) = {$}

Follow(A) = Follow(S) = {$}

Follow(A') = First(A'') − {ε} ∪ Follow(A)

= {c, $}

Follow(A'') = Follow(A) ∪ Follow(A'')

= {$}

follow(B) = first(C) − {ε} ∪ follow(B) ∪ follow(A')
= {g, c, $}

follow(C) = follow(B) ∪ follow(A')
= {g, c, $}

follow(D) = follow(A') = {c, $}

	a	b	c	d	f	g	$
S	S→A						
A	A→aA'A''						
A'		A'→bBC	A'→ε	A'→D	A'→B	A'→C	A'→ε
A''			A''→cA''				A''→ε
B		B→bBC			B→f		
C			C→ε	ε		C→g C→ε	C→ε
D			D→ε	D→d			D→ε

Table contains multiple entries. Therefore, grammar is not LL(1).

BOTTOM-UP PARSING

Shift-Reduce Parsing:

Ques: Construct the shift-Reduce parser for the following grammer and show the moves made by the parser for the input string

(i) cdcd

(ii) dddc

and the grammer is

$S \rightarrow CC$

$C \rightarrow cC$

$C \rightarrow d$

Solve: The stack implementation of shift-Reduce parser for input string cdcd.

S.No	Stack	Input string	Action
1.	$	cdcd $	–
2.	$c	dcd $	shift c
3.	$cd	cd $	shift d
4.	$cC	cd $	Reduced by C→d
5.	$C	cd $	Reduced by C→cC
6.	$Cc	d$	shift c
7.	$Ccd	$	shift d
8.	$CcC	$	Reduced by C→d
9.	$CC	$	Reduced by C→cC
10.	$S	$	Reduced by S→CC
			(Accepted)

The input string is succesfully passed.

→ The stack implementation of shift-Reduce parser for input string dddc is.

S.No	Stack	Input string	Action
1.	$	dddc$	shift d
2.	$d	ddc$	Reduced by C→d
3.	$C	ddc$	shift d
4.	$Cd	dc$	Reduced by C→d
5.	$CC	dc$	Reduced by S→CC
6.	$S	dc$	shift d
7.	$Sd	c$	Reduced by C→d
8.	$SC	c$	shift c
9.	$SCc	$	String cannot be further reduced.

Date
9-Feb-13
Saturday

• CD

• OPERATOR PRECEDENCE GRAMMER (PARSER)

This parser is a parser which parses a special class of grammers called operator grammer.

It is a type of shift-Reduce parser.

• OPERATOR GRAMMER

If the grammer satisfies the following conditions, then it can be called as operator grammer.

(i) No production rule should have ε on its right

$S \rightarrow \in X$

S → ABA

(ii) No production rule should have two adjacent non-terminals.

- **COMPUTATION OF LEADING**

1. 'a' is in ~~leading~~ LEADING(A) if:

(i) A → aY, Here Y is a grammer symbol

OR

(ii) A → Xa Y, Here X is a single non-terminal.

2. If there is a production, A → Bα and 'a' is in LEADING(B), then 'a' will be in LEADING(A)

- **COMPUTATION OF TRAILING**

1. 'a' is in TRAILING(A), if:

(i) A → Xa, Here X is a grammer symbol,

(ii) A → XaY, Here Y is a single non-terminal.

2. If there is a production A → qB, then TRAILING(B) will be in TRAILING(A), where q is a grammer symbol.

• COMPUTATION OF PRECEDENCE RELATION

1. Set $ <· a for all a in LEADING(S) and set b ·> $ for all b in TRAILING(S), where S is the start symbol of grammer G.

2. If A → xy and if x, y are terminals then set x ≐ y

3. If A → xBy and if x, y are terminals and B is a non-terminal then set x ≐ y

4. If A → qB and if q is a terminal then for all 'a' in LEADING(B), set q <· a.

5. If A → Bq and if q is a terminal then for all 'b' in TRAILING(B) set b ·> q.

• STEPS FOR OPERATOR PRECEDENCE PARSING

1. If top of the stack is $ and current input is also $ then ACCEPT.

2. If top of the stack is (let say a), a <· c or a ≐ c (c is current input symbol) then shift c on to the stack.

3. If a ·> c then reduce i.e pop all the symbols until the backward scan reaches <·,

from the stack (until first <· is reached).

4. Error, if undefined entry encountered.

Ques: $E \to E+T$

$E \to T$

$T \to T * F$

$T \to F$

$F \to (E)$

$F \to id$

Solve: Computation of LEADING(A):

Leading(E) = {+} ∪ {*, (, id} = {+, *, (, id}

Leading(T) = {*} ∪ {(, id} = {*, (, id}

Leading(F) = {(, id}

Computation of Trailing(A):

Trailing(E) = {+, *,), id}

Trailing(T) = {*,), id}

Trailing(F) = {), id}

(1) (a) $ <· leading(E)

$ <· +

$ <· (

$ <· id

$ <· *

(6) Trailing (S) :> $

+ :> $

* :> $

) :> $

id :> $

(2) & (3) (≐) $A \to aB$, aBb $a \doteq b$

(4) $A \to qB$

$q <\cdot$ leading (B)

→ +T :

+ <· leading (T)

⇒ + <· *

+ <· (

+ <· id

→ *E

* <· leading (E)

⇒ * <· (

* <· id

→ (E

(<· leading (E)

⇒ (<· +

(<· *

(<· (

(<· id

(5) $A \to Bq$

Trailing (B) :> q

E+

Trailing (E) :> +

+ :> +

* :> +

) :> +

id :> +

E)

→ Trailing (E) :>)

+ :>)

* :>)

) :>)

id :>)

T*

* :> *

) :> *

id :> *

$<\cdot$, $\doteq$ → shift in stack
$\cdot>$ → Reduce till $<\cdot$

Precedence

- ~~Parsing~~ Table.

	+	*	(	)	id	$
+	$\cdot>$	$<\cdot$	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$\cdot>$	$\cdot>$	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
(	$<\cdot$	$<\cdot$	$<\cdot$	$\doteq$	$<\cdot$	
)	$\cdot>$	$\cdot>$		$\cdot>$		$\cdot>$
id	$\cdot>$	$\cdot>$		$\cdot>$		$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$		$<\cdot$	

Date 11-Feb-13 Monday

CD

id + id * id.

Stack	Input string	Action
$	id + id * id $	Shift
$ <· id	+ id * id $	reduce
$	+ id * id $	Shift
$ <· +	id * id $	Shift
$ <· + <· id	* id $	reduce
$ <· +	* id $	Shift
$ <· + <· *	id $	Shift
$ <· + <· * <· id	$	reduce
$ <· + <· *	$	reduce
$ <· +	$	reduce
$	$	accept

Ques: $S \to L = R$ $\quad w = *id = id$

$S \to R$

$L \to *R$

$L \to id$

$R \to L$

Solve: Computation of Leading (A)

Leading (S) = $\{=\} \cup \{*, id\} = \{=, *, id\}$

Leading (L) = $\{*, id\}$

Leading (R) = $\{id, *\}$

Computation of Trailing (A):

Trailing (S) = $\{*, id, =\}$

Trailing (L) = $\{*, id\}$

Trailing (R) = $\{id, *\}$

(1) (a) $\$ \lessdot$ leading (S)

$\$ \lessdot =$

$\$ \lessdot *$

$\$ \lessdot id$

(b) Trailing (S) $\gtrdot \$$

$= \gtrdot \$$

$* \gtrdot \$$

$id \gtrdot \$$

(2) & (3)

(4) $A \to qB$ then $q \lessdot$ leading (B)

(a) $= R.$

$= \lessdot$ leading (R)

$= \lessdot *$

$= \lessdot$ id

(b) $*R$

$* \lessdot$ leading (R)

$* \lessdot *$

$* \lessdot$ id

(5) $A \to Bq$, then Trailing $(B) \gtrdot q$

(a) $L =$

Trailing $(L) \gtrdot =$

$* \gtrdot =$

id $\gtrdot =$

- Precedence table.

	=	*	id	$
=		$\lessdot$	$\lessdot$	
*	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$
id	$\gtrdot$			$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	

Ques

Solve

$w = *id = id$

Stack	Input string	Action
$	*id = id $	Shift
$*	id = id $	Shift
$*id	= id $	Reduce
$<· *	= id $	Reduce
$	= id $	Shift
$<· =	id $	Shift
$<· = <· id	$	Reduce
$<· =	$	Reduce
$	$	Accepted

Ques: $S \to a$ $\quad w = (a,a)$

$S \to \uparrow$

$S \to (T)$

$T \to T, S$

$T \to S$

Solve: Computation of Leading(A)

Leading(S) = {a, ↑, (}

Leading(T) = {, , a, ↑, (}

Computation of Trailing(A)

Trailing(S) = {a, ↑,)}

Trailing(T) = {, , a, ↑,)}

(1) (a) $\$ <\cdot$ leading(S)

$\$ <\cdot a$

$\$ <\cdot \uparrow$

$\$ <\cdot ($

(b) Trailing(S) :> $

Q :> $

↑ :> $

) :> $

(a) & (3) (≐)

(4) A → αB then α <· leading(B)

(a) (T

(<· leading(T)

(<·)

(<· a

(<· ↑

(<· (

(b) ,S

,<· leading(S)

,<· a

,<· ↑

,<· (

(5) A → Bα then Trailing(B) :> α

(a) T)

% · Trailing(T) :>)

, :>)

↑ :>)

a :>)

) :>)

(b) T,

trailing (T) .> ,

, .> ,

↑ .> ,

a .> ,

) .> ,

• Precedence Table.

	a	↑	(	)	,	$
a				.>	.>	.>
↑			<.	.>	.>	.>
(	<.	<.	<.	≐	<.	
)				.>	.>	.>
,	<.	<.	<.	.>	.>	
$	<.	<.	<.			

Stack	Input	Move.
$	(a, a) $	shift
$<. (	a, a) $	shift
$<. (<. a	, a) $	Reduce
$<. (	, a) $	shift
$<. (<. ,	a) $	shift
$<. (<. , <. a	) $	Reduce
$<. (<.	$	

• (dot) represents that symbol is ready for parsing.

- **LR PARSER**

Ques: Consider the following grammer:

$S \to AS/b$

$A \to SA/a$

Construct the SLR parser table.

Solve: 1. Augmented Grammer:

$S' \to S$ — (0)

$S \to AS$ — (1)

$S \to b$ — (2)

$A \to SA$ — (3)

$A \to a$ — (4)

2. LR(0) collection of items:

$I_0 = closure(S' \to .S)$

$S' \to .S$

$S \to .AS$

$S \to .b$

$A \to .SA$

$A \to .a$

$I_1 = Goto(I_0, S)$

$S' \to S.$

$A \to S.A$

$A \to .SA$

$A \to .a$

(wherever the starting symbol S is parsed place 1, $ = accept)

$S \to \cdot AS$

$S \to \cdot b$

$I_2 = \text{Goto}(I_0, A)$

$S \to A \cdot S$

$S \to \cdot AS$

$S \to \cdot b$

$A \to \cdot SA$

$A \to \cdot a$

$I_3 = \text{Goto}(I_0, b)$

$S \to b \cdot$

$I_4 = \text{Goto}(I_0, a)$

$A \to a \cdot$

$I_5 = \text{Goto}(I_1, A)$

$A \to SA \cdot$

$S \to A \cdot S$

$S \to \cdot AS$

$S \to \cdot b$

$A \to \cdot SA$

$A \to \cdot a$

$I_6 = \text{Goto}(I_1, S)$

$A \to S \cdot A$ $S \to \cdot AS$

$A \to \cdot SA$ $S \to \cdot b$

$A \to \cdot a$

Goto$(I_1, a) = I_4$
Goto$(I_1, b) = I_3$

I_4 = Goto (I_2, S)
$S \to AS.$
$A \to S.A$
$A \to .SA$
$A \to .a$
$S \to .AS$
$S \to .b$

Goto $(I_2, A) = I_2$
Goto $(I_2, a) = I_4$
Goto $(I_2, b) = I_3$

Goto $(I_5, S) = I_7$
Goto $(I_5, A) = I_2$
Goto $(I_5, a) = I_4$
Goto $(I_5, b) = I_3$

Goto $(I_6, A) = I_5$
Goto $(I_6, S) = I_6$
Goto $(I_6, a) = I_4$
Goto $(I_6, b) = I_3$

Goto $(I_7, A) = I_5$
Goto $(I_7, S) = I_6$
Goto $(I_7, a) = I_4$
Goto $(I_7, b) = I_3$

S_3 → represents shift (6 is a me S_4)

I_1 = Goto(I_0, S) 0 is me 1.

SLR(1) Table

State	Action a	Action b	Action $	Goto S	Goto A
I_0	S_4	S_3		1	2
I_1	S_4	S_3	accept	6	5
I_2	S_4	S_3		7	2
I_3	r_2	r_2	r_2		
I_4	r_4	r_4			
I_5	S_4, r_3	S_3, r_3		7	2
I_6	S_4	S_3		6	5
I_7	S_4, r_1	S_3, r_1	r_1	6	5

first(S) = {a, b}
first(A) = {a, b}
follow(S) = {$, a, b}
follow(A) = {a, b}

→ Search for the productions that have been completely parsed.

» $I_3 = S \to b.$
Action = [3, follow(S)] [2 is subscript of r ∵ it is eg-(2)]
[3, {$, a, b}] ($r_2$)

» $I_4 = A \to a.$
Action : [4, follow(A)] = r_4
[4, (a, b)] = r_4

$I_5 = A \to SA.$
Action = [5, follow(A)] = r_3
5, {a, b} : r_3

$I_7 = S \to AS.$
Action = [7, follow(S)] = r_1
[7, {$, a, b}] = r_1

The grammer is not LR(0), as there are multiple entries in the parse table. There is shift-reduce conflict in ~~action~~.
Action[5,a], Action[5,b], Action[7,c]

Ques: $S \to 1S1/0S0/10$.

Solve: (1) Augmented grammer will be,

$S' \to S$ — (0)
$S \to 1S1$ — (1)
$S \to 0S0$ — (2)
$S \to 10$ — (3)

(2) LR(0) collection of items:

$I_0 =$ closure $(S' \to .S)$
$S' \to .S$
$S \to .1S1$
$S \to .0S0$
$S \to .10$

$I_1 = \text{Goto}(I_0, S)$

$S' \to S.$

$I_2 = \text{Goto}(I_0, 1)$

$S \to 1.S1$ $S \to 1.0$

$S \to .1S1$

$S \to .0S0$

~~$S \to .10$~~ $S \to .10$

$I_3 = \text{Goto}(I_0, 0)$

$S \to 0.S0$

$S \to .1S1$

$S \to .0S0$

$S \to .10$

$I_4 = \text{Goto}(I_2, S)$

$S \to 1S.1$

$I_5 = \text{Goto}(I_2, 1)$

$S \to 1.S1$ $S \to 1.0$

$S \to .1S1$

$S \to .0S0$

$S \to .10$

$I_5 = \text{Goto}(I_2, 0)$

$S \to 10.$

$S \to 0.S0$

$S \to .1S1$

$S \to .0S0$

$S \to .10$

$I_6 = \text{Goto}(I_3, S)$
$S \to 0S.0$

$I_2 = \text{Goto}(I_3, 1)$
$S \to 1.S1$
$S \to 1.0$
$S \to .1S1$
$S \to .0S0$
$S \to .10$

$I_3 = \text{Goto}(I_3, 0)$
$S \to 0.S0$
$S \to .1S1$
$S \to .0S0$
$S \to .10$

$I_7 = \text{Goto}(I_4, 1)$
$S \to 1S1.$

$I_6 = \text{Goto}(I_5, S)$
$S \to 0S.0$

$I_4 = \text{Goto}(I_5, 1)$
$S \to 1.S1$
$S \to 1.0$

$I_8 = \text{Goto}(I_5, 0)$

$I_8 = \text{Goto}(I_6, 0)$
$S \to 0S0.$

	Action			Goto
State	0	1	$	S
I_0	S_3	S_2		1
I_1			accept	
I_2	S_5	S_2		4
I_3	S_3	S_2		6
I_4		S_7		
I_5	S_3, r_3	S_2, r_3	r_3	6
I_6	S_8			
I_7	r_1	r_1	r_1	
I_8	r_2	r_2	r_2	

first(S) = {1, 0}
follow(S) = {$, 1, 0}

$I_5 = S \to 10.$ — (3)
$I_7 = S \to 1S1.$ — (1)
$I_8 = S \to 0S0.$ — (2)

(1) $S \to 10$
Action = [5, follow(S)] = [5 {(1, 0, $)}
(2) $S \to 1S1$
Action [7, follow(S)]
(3) $S \to 0S0$
Action [8, follow(S)].

There is shift- reduce conflict.

* We do not remove left recursion or factoring in SLR

Ques: 3 $S \to CC$
$C \to xC / d$

Date 12-2-13 Tuesday

• CD

EXAMINATION QUESTIONS SLR

4) $S \to 0S0 / 1S1 / 10$

5) $S \to L = R / R$ → CLR also.
$L \to *R / id$
$R \to L$

6) $S \to AaAb$ =
$S \to BaBb$
$A \to \epsilon$
$B \to \epsilon$

7) $E \to E+T / T$
$T \to T*F / F$
$F \to (E) / id$

8) $S \to CC$
$C \to C.C$
$C \to d.$

• CLR(1) PARSER

Ques: 1 $S \to AaBb / BbAa$

$A \to \epsilon$

$B \to \epsilon$.

Sol: 1. Augmented Grammar:

$S' \to S$ — ⓪

$S \to AaBb$ — ①

$S \to BbAa$ — ②

$A \to \epsilon$ — ③

$B \to \epsilon$ — ④

2. LR(1) Collection of items:

I_0 = closure $(S' \to .S)$

$S' \to .S, \$$

$S \to .AaBb, \$$ → First $(aBb\$)$ → one after .A

$S \to .BbAa, \$$

$A \to . \ , a$

$B \to . \ , b$

I_1 = Goto (I_0, S)

$S' \to S. , \$$

I_2 = Goto (I_0, A)

$S \to A.aBb, \$$

I_3 = Goto (I_0, B)

$S \to B.bAa, \$$

$I_4 = \text{Goto}(I_2, a)$
$S \to Aa \cdot Bb, \$$
$B \to \cdot, b$

$I_5 = \text{Goto}(I_3, b)$
$S \to Bb.Aa, \$$
$A \to \cdot, a$

$I_6 = \text{Goto}(I_4, B)$
$S \to AaB.b, \$$

$I_7 = \text{Goto}(I_5, A)$
$S \to BbA.a, \$$

$I_8 = \text{Goto}(I_6, b)$
$S \to AaBb., \$$

$I_9 = \text{Goto}(I_7, a)$
$S \to BbAa., \$$

State | Action | | | Goto | |
| a | b | $ | S | A | B |

State	Action			Goto		
	a	b	$	S	A	B
I_0	r_3	r_4		1	2	3
I_1			accept			
I_2	S_4					
I_3		S_5				
I_4		r_4			7	6
I_5	r_3				7	
I_6		S_8				
I_7	S_9					
I_8			r_1			
I_9			r_2			

Ques: 2 $S \to CC$

$C \to cC/d$

Solve: 1. Augmented Grammer:

$S' \to S$ — (0)

$S \to CC$ — (1)

$C \to cC$ — (2)

$C \to d$ — (3)

2. LR(1) collection of items:

$I_0 = \text{closure}(S' \to .S)$

$S' \to .S, \$$

$S \to .CC, \$$

$C \to .cC, c/d$

$C \to .d, c/d$

I_1 = Goto (I_0, S)
$S' \to S.$, $

I_2 = Goto (I_0, C)
$S \to C.C$, $
$C \to .cC$, c/d , $
$C \to .d$, $

I_3 = Goto (I_0, c)
$C \to c.C$, c/d
$C \to .cC$, c/$
$C \to .d$, c/d

I_4 = Goto (I_0, d)
$C \to d.$, c/d

I_5 = Goto (I_2, C)
$S \to CC.$, $

I_6 = Goto (I_2, C)
$C \to c.C$, $
$C \to .cC$, $
$C \to .d$, $

I_7 = Goto (I_2, d)
$C \to d.$, $

I_8 = Goto (I_3, C)
$C \to cC.$, c/d

I_9 = Goto (I_3, C) = I_3
$C \to c.C$, c/d

I_4 = Goto (I_3, d)
$C \to d.$

I_5 = Goto (I_6, C)
$C \to cC.$, $

I_{10} = Goto (I_6, C)
$C \to c.C$, $
$C \to .cC$, $
$C \to .d$, $

I_6 = Goto (I_6, c)

I_7 = Goto (I_6, d)

I_{11} = Goto (I_6, d)

I_9 = Goto (I_{10}, C)

I_{10} = Goto (I_{10}, c)
$C \to c.C$, $

I_{11} = Goto (I_{10}, d)

State	Action c	Action d	Action $	Goto S	Goto C
I_0	S_3	S_4		1	2
I_1			accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	S_3	S_4			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	S_2	r_2			
I_9			r_2		

Date 13-2-23

•CD

Ques:

Ques: 3 $S \to L = R / R$

Solve: $L \to *R / id$

$R \to L$

Solve: 1. Augmented Grammer:

$S' \to S$ — (0)

$S \to L = R$ — (1)

$S \to R$ — (2)

$L \to *R$ — (3)

$L \to id$ — (4)

$R \to L$ — (5)

2. LR(1) collection of items:

I_0 = closure $(S' \to .S)$

$S' \to .S, \$$

$S \to .L = R, \$$

$S \to .R, \$$

$L \to .*R, =$

$L \to .id, =$

$R \to .L, \$$

I_1 = Goto (I_0, S)

$S' \to S., \$$

I_2 = Goto (I_0, L)

$S \to L. = R, \$$

$R \to L., \$$

I_3 = Goto (I_0, R)

$S \to R., \$$

I_4 = Goto $(I_0, *)$

$L \to *.R, =$ $\quad L \to .*R, =$

$R \to .L, =$ $\quad L \to .id, =$

I_5 = Goto (I_0, id)

$L \to id., =$

I_6 = Goto $(I_2, =)$

$S \to L = .R, \$$ $\quad L \to .*R, \$$

$R \to .L, \$$ $\quad L \to .id, \$$

I_7 = Goto (I_2, R)
$L \to *R.$, =

I_8 = Goto (I_4, L)
$R \to L.$, =

I_9 = Goto (I_4, *)
$L \to *.R$, =
$R \to .L$, =
$L \to .*R$, =
$L \to .id$, =

I_5 = Goto (I_4, id)
$L \to id.$, =

I_9 = Goto (I_6, R)
$S \to L=R.$, $

I_{10} = Goto (I_6, L)
$R \to L.$, $

I_{11} = Goto (I_6, *)
$L \to *.R$, $ $L \to .*R$, $
$R \to .L$, $ $L \to .id$, $

I_{12} = Goto (I_6, id)
$L \to id.$, $

State	=
I_0	
I_1	
I_2	
I_3	
I_4	
I_5	
I_6	
I_7	
I_8	
I_9	
I_{10}	
I_{11}	
I_{12}	

for reduce → $I_{10} = R \to L. \$$ (I_{10} R $ me, R→L.$ production ka eqn-no.)

$I_3 = \text{Goto}(I_{11}, R)$
$L \to *R., \$$

$I_{10} = \text{Goto}(I_{11}, L)$
$R \to L., \$$

$I_{11} = \text{Goto}(I_{11}, *)$
$L \to *.R, \$$
$R \to .L, \$$

$I_{12} = \text{Goto}(I_{11}, id)$
$L \to id., \$$

~~$I_{15} = \text{Goto}(I_{10}, R)$~~

State	Action				Goto		
	=	*	id	$	S	L	R
I_0		S_4	S_5		1	2	3
I_1				accept			
I_2	S_6			γ_5			
I_3				γ_2			
I_4		S_4	S_5			8	7
I_5	γ_4,						
I_6		S_{11}	S_{12}			10	9
I_7	γ_3						
I_8	γ_5						
I_9				γ_1			
I_{10}				γ_5			
I_{11}		S_{11}	S_{12}			10	13
I_{12}				γ_4			
I_{13}				γ_3			

The productions are,

$S \to L = R$

$\to id = L$

$\to id = id$

Stack	Input string	Action
0	id = id $	S_5 → shift id
0 id 5	= id $	r_4 → reduce id
0 L 2	= id $	
0 L = 6	id $	
0 L = 6 id 12	$	
0 L = 6 L 10	$	
0 L = 6 R 9	$	
0 S 1	$	
	accept.	

Date: 20-Feb-13 Wednesday

• CP

Ques: 4 $S \to AS$

$S \to b$

$A \to SA$

$A \to a$

First(S) = {a, b}

First(A) = {a, b}

Solve: (1) Augmented Grammer:

$S \to S$ — (0)

$S \to AS$ — (1)

$S \to b$ — (2)

$A \to SA$ — (3)

$A \to a$ — (4)

⇒ I_0 = closure $(S' \to .S, \$)$

$S' \to .S, \$$

$S \to .AS, \$$ }

$S \to .b, \$$ }

$A \to .SA, a/b$

$A \to .a, a/b$

$S \to .AS, a/b$ }

$S \to .b, a/b$ }

This can be written as,

I_0 = closure $\{S' \to .S, \$\}$

$S' \to .S, \$$

$S \to .AS, \$ / a/b$

$S \to .b, \$ / a/b$

$A \to .SA, a/b$

$A \to .a, a/b$

⇒ I_1 = Goto (I_0, S)

$S' \to S., \$$

$A \to S.A, a/b$

$A \to .SA, a/b$

$A \to .a, a/b$

$S \to .AS / a/b$

$S \to .b, a/b$

I_2 = Goto(I_0, A)

$S \to A.S$, $/a/b$

$S \to .AS$, $/a/b$

$S \to .b$, $/a/b$

$A \to .SA$, $/a/b$

$A \to .a$, $/a/b$

I_3 = Goto(I_0, b)

$S \to b.$, $/a/b$

I_4 = Goto(I_0, a)

$A \to a.$, a/b

I_5 = Goto(I_1, A)

$A \to SA.$, a/b

$S \to A.S$, a/b

$S \to .AS$, a/b

$S \to .b$, a/b

$A \to .SA$, a/b

$A \to .a$, a/b

I_6 = Goto(I_4, S)

$A \to S.A$, a/b

$A \to .SA$, a/b

$S \to .AS$, a/b

$S \to .b$, a/b

$A \to .a$, a/b

I_4 = Goto (I_1, a)
$A \to a.$, $a/b.$

I_5 = Goto (I_1, b)
$S \to b.$, $a/b.$

I_6 = Goto (I_2, S)
$S \to AS.$, $\$/a/b.$
$A \to S.A$, $\$/a/b.$
$A \to .SA$, $\$/a/b.$
$A \to .a$, $\$/a/b.$
$S \to .AS$, $\$/a/b.$
$S \to .b$, $\$/a/b.$

I_7 = Goto (I_2, A)
$S \to A.S$, $\$/a/b.$
$S \to .AS$, $\$/a/b.$

I_8 = Goto (I_2, b)
$S \to b.$, $\$/a/b.$

I_9 = Goto (I_2, a)
$A \to a.$, $a/b.$

I_9 = Goto (I_5, S)
$S \to AS.$, $a/b.$
$A \to S.A$, $a/b.$
$A \to .SA$, $a/b.$
$A \to .a$, $a/b.$

S → .AS , a/b
S → .b , a/b.

I_{10} = Goto (I_5, A)
S → A.S , a/b.
S → .AS , a/b.
S → .b , a/b.
A → .SA , a/b
A → .a , a/b.

I_{11} = Goto (I_5, b)
S → b. , a/b.

I_{12} = Goto (I_5, a)
A → a. , a/b.

I_{13} = Goto (I_6, A)
A → SA. , a/b.
S → A.S , a/b.
I_{14} = ~~Goto~~

I_{16} = Goto (I_6, S)
A → S.A , a/b.

I_{17} = Goto (I_6, b)
S → b. , a/b

$I_8 = \text{Goto}(I_8, A)$
$A \to SA.$, $a/b.$

$I_6 = \text{Goto}(I_8, S)$
$A \to S.A$, $a/b.$

$I_4 = \text{Goto}(I_8, a)$
$A \to a.$, $a/b.$

$I_7 = \text{Goto}(I_8, b)$
$S \to b.$, $a/b.$

$I_5 = \text{Goto}(I_9, A)$
$A \to SA.$, $a/b.$

$I_6 = \text{Goto}(I_9, S)$
$A \to S.A$, $a/b.$

$I_4 = \text{Goto}(I_9, a)$

$I_7 = \text{Goto}(I_9, b)$

$I_9 = \text{Goto}(I_{10}, S)$
$S \to AS.$, $a/b.$

$I_{10} = \text{Goto}(I_{10}, A)$
$S \to A.S$, $a/b.$

$I_7 = \text{Goto}(I_{10}, b)$

$I_4 = \text{Goto}(I_{10}, a)$

State	Action			Goto	
	a	b	$	S	A
I_0	S_4	S_3		1	2
I_1	S_4	S_7	accept.	6	5
I_2	S_4	S_3		8	2
I_3	r_2	r_2	r_2		
I_4	r_4	r_4			
I_5	S_4/r_3	S_7/r_3		9	10
I_6	S_4	S_7		6	5
I_7	r_2	r_2			
I_8	S_4/r_1	S_7/r_1	r_1	6	5
I_9	S_4/r_1	S_7/r_1		6	5
I_{10}	S_4	S_7		9	10

→ This is not CLR grammar.

H.W

Ques: 5 $E \to E+T/T$

$T \to T*F/F$

$F \to (E)/id.$

Solve: Augmented Grammer:

$E' \to .E$ — (0)

$E \to .E+T$ — (1)

$E \to .T$ — (2)

$T \to .T*F$ — (3)

$T \to .F$ — (4)

$F \to .(E)$ — (5)

$F \to .id$ — (6)

⇒ I_0 = closure($E' \to .E, \$$)

$E' \to .E, \$$

$E \to .E+T, \$/+$

$E \to .T, \$/+$

$T \to .T*F, \$/+/*$

$T \to .F, \$/+/*$

$F \to .(E), \$/+/*$

$F \to .id, \$/+/*$

I_1 = Goto(E.

- **LALR** (Look ahead LR Parser)

Ques: 1. $S \to CC$

$C \to cC$

$C \to d.$

Sol: 1. Augmented Grammer:

$S' \to .S$ — (0)

$S \to .CC$ — (1)

$C \to .cC$ — (2)

$C \to .d$ — (3)

2. LR(1) collection of items:

I_0 = closure($S' \to .S$)

$S' \to .S, \$$

$S \to .CC, \$$

$C \to .cC, c/d$

$C \to .d, c/d.$

I_1 = Goto closure (I_0, S)
$S' \to S.$, $

I_2 = Goto closure (I_0, C)
$S \to C.C$, $
$C \to .cC$, $
$C \to .d$, $

I_3 = Goto closure (I_0, c)
$C \to c.C$, c/d
$C \to .cC$, c/d
$C \to .d$, c/d.

I_4 = Goto closure (I_0, d)
$C \to d.$, c/d.

I_5 = Goto closure (I_2, C)
$S \to CC.$, $

I_6 = Goto closure (I_2, c)
$C \to c.C$, $
$C \to .cC$, $
$C \to .d$, $

I_7 = Goto closure (I_2, d)
$C \to d.$, $

State
I_0
I_1
I_2
I_3
I_4
I_5
I_6
I_7
I_8
I_9

→ Co

I_8 = Goto (I_3, C)
$C \to cC.$, c/d
I_3 = Goto (I_3, c)
I_4 = Goto (I_3, d)
I_9 = Goto (I_6, C)
$C \to cC.$, $
I_6 = Goto (I_6, c)
I_7 = Goto (I_6, d)

State	Action			Goto	
	c	d	$	S	C
I_0	S_3	S_4		1	2
I_1			accept		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	r_3	r_3			
I_5			r_1		
I_6	S_6	S_7			9
I_7			r_3		
I_8	r_2	r_2			
I_9			r_2		

→ Consider the states I_4 and I_7,
I_4 = Goto (I_0, d)
$C \to d.$, c/d.

and I_7 = Goto (I_2, d)
$C \to d.$, $

I_4 and I_7 differs only in the look ahead they have the same production. Hence, this states are combined to form a single state called as I_{47}.

$\therefore I_{47} = C \to d. \;,\; c/d/\$$

Similarly,

$I_3 = Goto(I_0, C)$

$C \to c.C \;,\; c/d$

$C \to .cC \;,\; c/d$

$C \to .d \;,\; c/d$

and $I_6 = Goto(I_2, C)$

$C \to c.C \;,\; \$$

$C \to .cC \;,\; \$$

$C \to .d \;,\; \$$

$\therefore I_{36} = C \to c.C \;,\; c/d/\$$

$C \to .cC \;,\; c/d/\$$

$C \to .d \;,\; c/d/\$$

and

$I_{89} = C \to cC. \;,\; c/d/\$$

State	C
I_0	S_{36}
I_1	
I_2	S_{36}
I_{36}	S_{36}
I_{47}	r_3
I_{89}	r_2
I_5	

Date: 21-Feb-13 Thursday

Ques:

Solve: ① Augr

2. LF

State	Action			Goto	
	c	d	$	S	L
I_0	S_{36}	S_{47}		1	2
1			accept		
2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	r_3	r_3	r_3		
I_{89}	r_2	r_2	r_2		
I_5			r_1		

Date: 27-02-13 Tuesday

Ques: $S \to L = R / R$

$L \to *R / id$

$R \to L$

Give: ① Augmented Grammer.

$S' \to .S$ — ⓪

$S \to .L = R$ — ①

$S \to .R$ — ②

$L \to .*R$ — ③

$L \to .id$ — ④

$R \to .L$ — ⑤

2. LR(1) collection of items:

I_0 = closure ($S' \to .S$)

$S' \to .S$, $

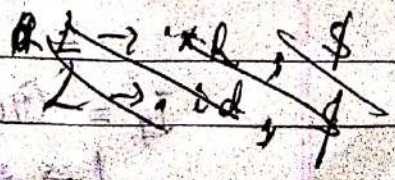
$S \to .L = R$, $

$S \to .R$, $

$L \to .*R$, =/$

$L \to .id$, =/$

$R \to .L$, $



$I_1 = \text{Goto}(I_0, S)$

$S' \to S.\ , \$$

$I_2 = \text{Goto}(I_0, L)$

$S \to L. = R, \$$

$R \to L.\ , \$$

$I_3 = \text{Goto}(I_0, R)$

$S \to R.\ , \$$

$I_4 = \text{Goto}(I_0, *)$

$L \to *.R\ , = / \$$

$R \to .L\ , = / \$$

$L \to .*R\ , = / \$$

$L \to .id\ , = / \$$

$I_5 = \text{Goto}(I_0, id)$

$L \to id.\ , = / \$$

$I_6 = \text{Goto}(I_2, =)$

$S \to L = .R\ , \$$

$R \to .L\ , \$$

$L \to .*R\ , \$$

$L \to .id\ , \$$

$I_7 = \text{Goto}(I_4, R)$

$L \to *R.\ , = / \$$

I_8 = Goto (I_4, L)
$R \to L.$, =/$

I_4 = Goto $(I_4, *)$
$L \to *.R$, =/$
$R \to .L$, =/$
$L \to .*R$, =/$
$L \to .id$, =/$

I_5 = Goto (I_4, id)
$L \to id.$, =/$

I_9 = Goto (I_6, R)
$S \to L=R.$, $

I_{10} = Goto (I_6, L)
$R \to L.$, $

Goto $(I_6, *) = I_{11}$
$L \to *.R$, $
$R \to .L$, $
$L \to .*R$, $
$L \to .id$, $

I_{12} = Goto $(I_6, id) = I_{12}$
$L \to id.$, $

~~I_7~~ = Goto ~~(I_9, R)~~
~~$L \to *R.$~~ , =/$

I_8 = Goto (I_3, L)
R → L. , $

I_4 = Goto (I_3, *)
L → *. R , $

I_5 = Goto (I_3, id)
L → id. , $

I_{13} = Goto (I_4, R)
L → *R. , $

I_{10} = Goto (I_{11}, L)
R → L. , $

I_{11} = Goto (I_4, *)
I_{12} = Goto (I_4, id)

P.T.O →

State	Action				Goto		
	=	*	id	$	S	L	R
I_0		S_4	S_5		I	2	3
I_1				accept			
I_2	S_6			r_5			
I_3				r_2			
I_4		S_4	S_5			8	7
I_5	r_4			r_4			
I_6		S_{11}	S_{12}			10	9
I_7	r_3			r_3			
I_8	r_5			r_5			
I_9				r_1			
I_{10}				r_5			
I_{11}		S_{11}	S_{12}			10	13
I_{12}				r_4			
I_{13}				r_3			

I_4 and I_{11} = I_{411}

$L \rightarrow *.R$, =/$

$R \rightarrow .L$, =/$

$L \rightarrow .*R$, =/$

$L \rightarrow .id$, =/$

I_5 and I_{12} = I_{512}

$L \rightarrow id.$, =/$

I_8 and I_{10} = I_{810}

$R \rightarrow L.$, =/$.

I_7 and I_{13} = I_{713}

$L \rightarrow *R.$ =/$

State	Action				Goto		
	=	*	id	$	S	L	R
					1	2	3
I_0		S_{411}	S_{512}				
I_1				r_5			
I_2	S_6			r_2			
I_3						8,10	7,13
I_{411}		S_{411}	S_{512}				
I_{512}	r_4			r_4			
I_6		S_{411}	S_{512}			8,10	9
I_{713}	r_3			r_3			
I_{810}	r_5			r_5			
I_9				r_1			

Date
22-2-13
Friday

Ques: 3 $S \to a / \wedge / (R)$

$T \to S, T / S$

$R \to T$

Solve: I_0 = closure $(S' \to .S)$

$S' \to .S, \$$ — (0)

$S \to .a, \$$ — (1)

$S \to .\wedge, \$$ — (2)

$S \to .(R), \$$ — (3)

I_1 = Goto (I_0, S)

$S' \to S., \$$

I_2 = Goto (I_0, a)

$S \to a\cdot, \$$

I_3 = Goto $(I_0, \wedge)$

$S \to \wedge\cdot, \$$

I_4 = Goto $(I_0, ($

$S \to (\cdot R), \$$

$R \to \cdot I_3$)

$T \to \cdot S, I$)

$T \to \cdot S$,)

$S \to \cdot a$;) ,)

$S \to \cdot \wedge$,) ,)

$S \to \cdot (R)$;) ,)

I_5 = Goto (I_4, R)

$S \to (R\cdot)$; $\$$

I_6 = Goto (I_4, I)

$R \to I\cdot$;)

I_7 = Goto (I_4, S)

$T \to S\cdot, I$;)

$T \to S\cdot$;)

I_8 = Goto (I_4, a)

$S \to a\cdot$;)

I_9 = Goto $(I_4, \wedge)$

$S \to \wedge\cdot$;) ,)

I_{10} = Goto $(I_4, ()$

$S \to (.R)$,) , ,

$R \to .T$,)

$T \to .S ; T$,)

$T \to .S$;)

$S \to .a$;) , ,

$S \to .\wedge$;) , ,

$S \to .(R)$;) ,

I_{11} = Goto $(I_5,)$

$S \to (R)\cdot$, $

I_6 = Goto (I_{10}, T)

$R \to T.$,)

I_7 = Goto (I_{10}, S)

$T \to S.$, T ;)

$T \to S.$;)

I_8 = Goto (I_{10}, a)

$S \to a.$,)

I_9 = Goto $(I_{10}, \wedge)$

$S \to \wedge.$,)

I_{10} = Goto $(I_{10}, ()$

$S \to (.R)$,)

$I_{12} = \text{Goto}(I_7, ,)$

$T \to S, \cdot T,)$

$T \to \cdot S, T,)$

$T \to \cdot S,)$

$S \to \cdot a, ,)/,$

$S \to \cdot \wedge, ,)/,$

$S \to \cdot (R), ,)/,$

$I_{13} = \text{Goto}(I_{10}, R)$

$S \to (R \cdot), ,)/,$

$\text{Goto}(I_{10}, T) = I_6$

$\text{Goto}(I_{10}, S) = I_7$

$\text{Goto}(I_{10}, a) = I_8$

$\text{Goto}(I_{10}, \wedge) = I_9$

$\text{Goto}(I_{10}, ( ) = I_{10}$

$I_{14} = (I_{12}, T)$

$T \to S, T \cdot,)$

$\text{Goto}(I_{12}, S) = I_7$

$(I_{12}, a) = I_8$

$(I_{12}, \wedge) = I_9 \quad , (I_{12}, ( ) = I_{10}$

$I_{15} = (I_{13},))$

$S \to (R) \cdot ,)/,$

State	Action						Goto		
	a	∧	(	)	,	$	R	S	T
I_0	S_2	S_3	S_4					1	
I_1						accept			
I_2						r_1			
I_3						r_2			
I_4	S_8	S_9	S_{10}				5	6	7
I_5				S_{11}					
I_6				r_6					
I_7				r_5	S_{12}				
I_8				r_1	r_1				
I_9				r_2	r_2				
I_{10}	S_8	S_9	S_{10}				13	7	6
I_{11}						r_3			
I_{12}	S_8	S_9	S_{10}					7	14
I_{13}				S_{15}					
I_{14}				r_4					
I_{15}				r_3	r_3				

$I_5 = I_{13}$

$S \to (R.)$, $\$ /) / ,$

$I_{11} = I_5$

$S \to (R) . ,) / , / \$$

$I_4 = I_{10}$

$S \to (.R)$, $) / , / \$$

$R \to .T$,)

$T \to .S$, T ,)

$T \to .S$,)

$S \to .a$, $) / ,$

$S \to .\wedge$, $) / ,$

$S \to .(R)$, $) / ,$

State

I_0

I_{28}

I_{39}

I_{410}

I_{513}

I_6

I_7

I_8

I_{115}

I_{12}

I_{14}

I_1

Ques 1:

$I_3 = I_9$

$S \to \wedge .\ ,\ \$ /) / ,$

$I_2 = I_8$

$S \to a .\ ,\ \$ /) / ,$

State	Action						Goto		
	a	∧	(	)	,	$	R	S	T
I_0	S_{28}	S_{39}	S_{410}					1	
I_{28}						r_1			
I_{39}				r_2	r_2	r_2			
I_{410}	S_{28}	S_{39}	S_{104}				513	7	6
I_{513}				S_{115}					
I_6				r_6					
I_7				r_5	S_{12}				
I_8				r_1	r_1				
I_{115}				r_3	r_3	r_3			
I_{12}	S_{28}	S_{39}	S_{410}					7	14
I_{14}				r_4					
I_1						Accept			

Ques 4. $A \to BaC / CaBa$ (12-state) CLR
$C \to \omega$ (10 state) LALR
$B \to q$

5. $S \to AA$ (9 state) CLR
$A \to aA / b$ (6 state) LALR

6. $S \to Aa / bAc / Bc / bBa$ (12 states) CLR.
$A \to d$
$B \to d$

OPERATOR PRECEEDENCE FUNCTION

- Compilers using operator precedence parser need not store the table of precedence relation.

In most cases, the table can be encoded by two precedence functions f and g, which map terminal symbols to integers.

We attempt to select f and g so that, for symbols 'a' and 'b'.

1) $f(a) < g(b)$
   whenever $a \lessdot b$.

2) $f(a) > g(b)$
   whenever $a \gtrdot b$

3) $f(a) = g(b)$
   whenever $a \doteq b$

ALGORITHM FOR CONSTRUCTING THE PRECEDENCE FUNCTION.

Input: Operator precedence table.
Output: Precedence function; if such function exist

[METHOD:]

1. Create function $f(a)$ and $g(b)$ for each terminal including $\$$.
2. Partition the symbol in group so that $f(a)$ and $g(b)$ are in the same group; if $(a = b)$
3. Create a directed graph whose nodes are the group identified in step-2
   For any symbol 'a' and 'b' do
   (i) if $a < b$;
   place an edge from the group $g(b)$ to the group of $f(a)$.

   (ii) if $a > b$,
   place an edge from the group $f(a)$ to $g(b)$.
4. If the constructed graph has a cycle then no precedence function exist.
5. If the constructed graph has no cycle then collect the length of longest path from the group of $f(a)$ and $g(b)$ respectively.

P.T.O.

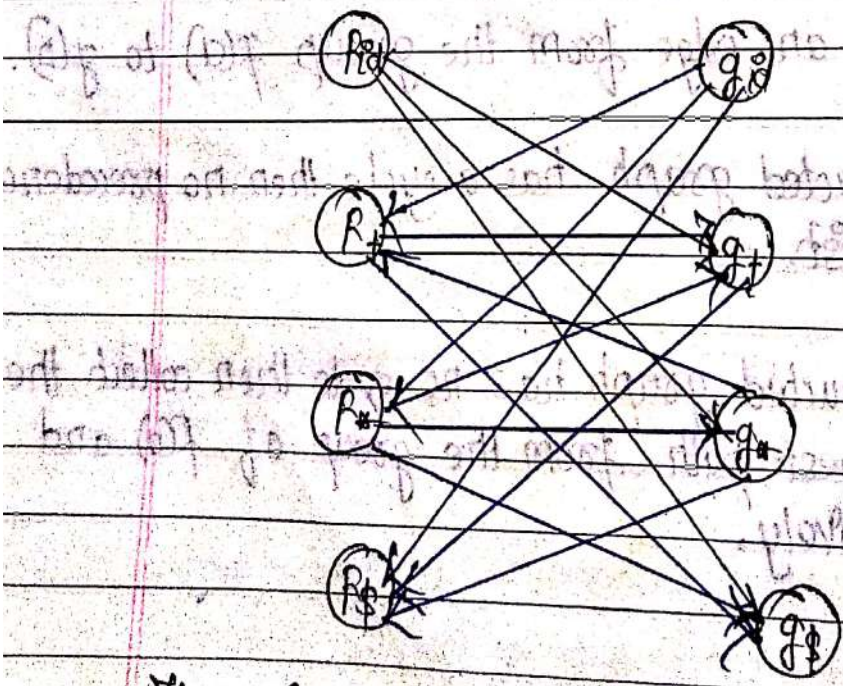
Ques/ Construct the operator precedence function of the given following precedence table.

f \ g	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

There are no equal to (≐) relation, so each symbol is in a group by itself.

Groups are f_{id}, f_+, f_*, $f_\$$

g_{id}, g_+, g_*, $g_\$$



There is no cycle, so precedence function exist

function \ terminal	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Given

$S \to L = R$

$S \to R$

$L \to *R$

$L \to id$

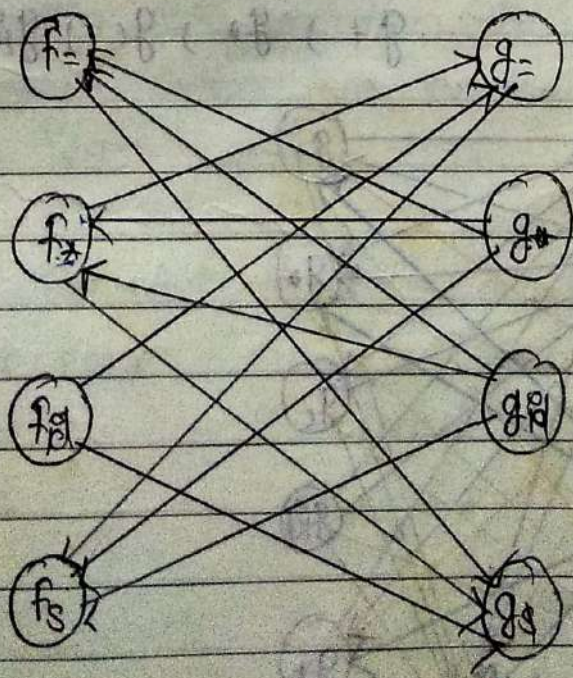
$R \to L$

f \ g	=	*	id	$
=		<	<	>
*	>	<	<	>
id	>			>
$	<	<	<	

There are no. $\doteq$ relation, so each symbol is in group by itself.

Groups are $f_= \; f_* \; f_{id} \; f_\$$

$g_= \; g_* \; g_{id} \; g_\$$



① $f_* \to g_= = 1$

② $f_< \to g_= = 1 \to g_* = 2$

$f_< \to f_* \to g_* = 2$

$f_< \to f_* = f_* = f_{id} = 3$

	=	*	id	$
f	1	2	2	0
g	1	3	3	0

Ques. $E \to E+T / T$

$T \to T*F / F$

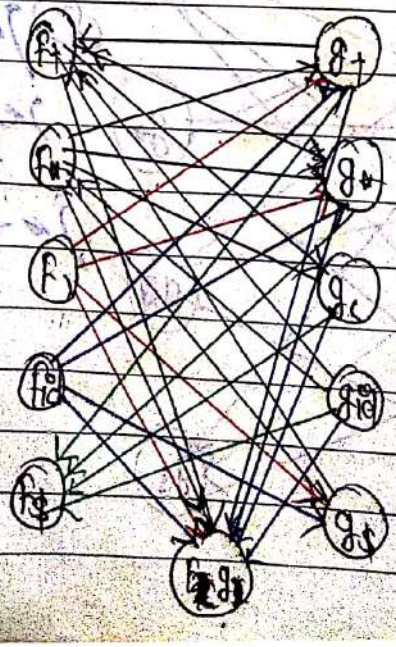
$F \to (E) / id$

F/g	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
id	>	>		>		>
$	<	<	<		<	

There is $\doteq$ relation

Groups are $f_+, g_+, f_*, f_(, f_), f_{id}, f_\$$

$g_+, g_*, g_(, g_{id}, g_\$$



	+	*	(	)	id	$
f	2	4	6	6	4	0
g	1	3	5	0	5	0

V.V Imp.

SHIFT REDUCE PARSING:

Shift Reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

→ As we shall see, the handle always appears at the top of the stack just before it is identified as the handle.

→ We use $ to mark the bottom of the stack and also the right end of the input.

→ Conventionally, Here, we show the top of the stack on the right, rather than on the left as we did for top-down parsing.

→ Initially, the stack is empty and the string w is on the input.

STACK	INPUT
$	w$

During a left to right scan of the i/p string, the parser shifts 0 or more i/p symbols onto the stack, until it is ready to reduce a string β of grammer symbols on top of the stack.

→ It then reduces β to the head of appropriate production. The parser repeat this cycle until it has detected an error or until the stack contains the start symbol and input is empty.

STACK	INPUT
$S	- $

→ Upon entering above configuration, the parser halts and announces successful completion.

→ The primary operations of shift Reduce Parser are Shift & Reduce, there are actually four possible actions a shift Reduce parser can make.

(1) SHIFT:
Shift the next i/p symbol onto the top of the stack.

(2) REDUCE:
The right end of the string to be reduced must be at the top of stack. It locate

the left end of the string within the stack & decide what with what non-terminal to replace the string.

(iii) ACCEPT:
Announce succesful completion of parsing.

(iv) ERROR:-
Discover a syntax error and call an recovery routine.

- **VIABLE PREFIXES:**

In shift-Reduce parsing, the stack contents are always a viable prefix - that is, a prefix of some right sentencial form that ends no further write then the end of handle of that right sentential form.

Q: Consider the following grammer:

$E \rightarrow E+T / T$
$T \rightarrow T*F / F$
$F \rightarrow id$

Show the shift Reduce parsing for a string id+id*id and show the handle and viable prefix:

P.T.O.

Solve:

Stack	Input	Action
$	id + id * id $	Shift
$id	+ id * id $	reduce
$F	+ id * id $	reduce
$T	+ id * id $	reduce
$E	+ id * id $	Shift
$E +	id * id $	Shift
$E + id	* id $	reduce
$E + F	* id $	reduce
$E + T	* id $	Shift
$E + T *	id $	reduce
$E + T * id	$	reduce
$E + (T * F)	$	reduce
$E + T	$	reduce
$E	$	Accept.

Handle	Viable Prefix
id	id
F	F
T	T
id	E + id
F	E + F
id	E + T * id
T * F	E + T * F
E + T	E + T

HANDLE

A handle of a string is the string that matches the right string of production and whose reduction to the non-terminal to left hand side of production represents one step along reverse of a rightmost derivation.

In many cases, the leftmost substring β that matches the right side of some production. $A \to \beta$ is not a handle, because a reduction by the production $A \to \beta$ yields a string that cannot be reduced to start symbol.

Formally, a handle of a right sentencial form, is a production $A \to \beta$ and a position of γ where string β may be found and replaced by A. to produce the previous right sentential form in a rightmost derivation of γ. i.e if $S \Rightarrow \alpha A w \Rightarrow \alpha \beta w$, then $A \to \beta$ in the position following α is a handle of ~~$\alpha A w$~~ $\alpha \beta w$.

Date 7-3-13 Thursday

UNIT-3

SYNTAX DIRECTED TRANSLATION & INTERMEDIATE CODE GENERATION

INTERMEDIATE CODE GENERATION

- While translating a source program into a functionally equivalent object code representation a parser may first generate an intermediate representation.

→ This makes retargetting of the code possible and allows some optimizations to be carried out, that would otherwise not be possible.

→ The following are commonly used intermediate representations:

(i) Postfix Notations.
(ii) Syntax tree.
(iii) Three address code (tac).

(i) Three address code (tac):

→ Three address code is a sequence of statements, of the form A := B OP C where A, B, and C are either program defined names, constants or compiler generated temporary names.

→ OP stands for any operator, such as fixed or floating point ~~number~~ arithmetic operator or a logical operator on boolean

* / %.
+ -

value data.

The reason for the name "three address code" is that each statement usually contains three addresses, 2 for the operands and 1 for the result.

For ex: The three address code for the expression,

$s = a + b * c + d$

$t_1 = b * c$

$t_2 = a + t_1$

$t_3 = t_2 + d$

$s = t_3$

Q: $x = -(a+b) * (c + d/e - f)$

$\Rightarrow$ $t_1 = a + b$

$t_2 = -t_1$

$t_3 = d/e$

$t_4 = c + t_3$

$t_5 = t_4 - f$

$t_6 = t_2 * t_5$

$x = t_6$

Date 8-3-23 Friday

• REPRESENTATION OF 3-ADDRESS CODE

(i) Quadruple Representation.
(ii) Triple Representation.
(iii) Indirect Triple Representation.

(i) Quadruple Representation:

→ Using this representation, the three address statement A := B OP C is represented by placing OP in the operator fields, B in the operand1 field, C in the operand2 field and A in the result field.

→ The statement A := OP B where OP is a unary operator is represented by placing OP in the operator field, B in the operand1 field and A in the result field; The operand2 field is not used.

For ex: a = b * c + d

$t_1 := b * c$

$t_2 := t_1 + d$

$a := t_2$

	Operator	Operand1	Operand2	result
1	*	b	c	t_1
2	+	t_1	d	t_2
3	=	t_2		a

(ii) Triple Representation:

→ To avoid entering temporary names into the symbol table, one can allow the statement computing a temporary value to represent that value

→ If we do so, three address statement can be representable by a structure with only three fields OP, operand1, & operand2, where operand1 & operand2, the arguments of OP.

→ Since, three fields are used, this intermediate code format is known as triples.

for ex: → We use parenthesed numbers to represent pointers into the triple structure.

	operator	operand 1	operand 2
(1)	*	b	c
(2)	+	(1)	d
(3)	:=	a	(2)

(iii) Indirect Triple Representation.

→ In direct triple represtn uses an additional array to list pointers to the triples, in the desired order.

for ex:

Statement
(1) (14)
(2) (15)
(3) (16)

S.H	Operator	Operand1	Operand2
(14)	*	b	c
(15)	+	(14)	d
(16)	:=	a	(15)

Ques Write the Quadruple, Triples and Indirect triples for the expression $-(a*b)+(c+d)-(a+b+c+d)$

Sol^n: Nao, $t_1 = a*b$

$t_2 = -t_1$

$t_3 = c+d$

$t_4 = a+b$

$t_5 = t_4 + t_3$

$t_6 = t_2 + t_3$

$t_7 = t_6 - t_5$

(1) Quadruple Representation:

	operator	operand 1	operand 2	result
1	*	a	b	t_1
2	-	t_1	-	t_2
3.	+	c	d	t_3
4	+	a	b	t_4
5	+	t_4	t_3	t_5
6	+	t_2	t_3	t_6
7	-	t_6	t_5	t_7

(2) Triple Representation:

P.T.O.

	Operator	Operand1	Operand2
1	*	a	b
2	-	(1)	-
3	+	c	d
4	+	a	b
5	+	(4)	(3)
6	+	(2)	(3)
7	-	(6)	(5)

(3) Indirect Triple Representation.

	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)
(6)	(19)
(7)	(20)

	Operator	Operand1	Operand2
(14)	*	a	b
(15)	-	(14)	-
(16)	+	c	d
(17)	+	a	b
(18)	+	(17)	(16)
(19)	+	(15)	(16)
(20)	-	(19)	(18)

Ques: $A = -B * (C+D)$

$t_1 := -B$ $\quad t_3 := t_1 * t_2$

$t_2 := C+D$ $\quad A := t_3$

Date
9-3-13
Saturday

• CD

Ques: $(a+b+c) * (c+d*e)$

Sore: Now,

$t_1 := a+b$

$t_2 := t_1 + c$

$t_3 := d * e$

$t_4 := c + t_3$

$t_5 := t_2 * t_4$

(1) Quadruple Representation:

	Operator	Operand 1	Operand 2	result
1	+	a	b	t_1
2	+	t_1	c	t_2
3	*	d	e	t_3
4	+	c	t_3	t_4
5	*	t_2	t_4	t_5

(2) Triple Representation.

	Operator	Operand 1	Operand 2
1	+	a	b
2	+	(1)	c
3	*	d	e
4	+	c	(3)
5	*	(2)	(4)

3) Indirect triple representation

	Statement
(1)	(10)
(2)	(11)
(3)	(12)
(4)	(13)
(5)	(14)

	Operator	Operand 1	Operand 2
(10)	+	a	b
(11)	+	(10)	c
(12)	*	d	e
(13)	+	c	(12)
(14)	*	(11)	(13)

ues: $S = a + b * c / f * g - d * e + h$

ves Now, $t_1 = b * c$

$t_2 = f * g$

$t_3 = d * e$

$t_2 = t_1 / f$

$t_3 = t_2 * g$

$t_4 = d * e$

$t_5 = a + t_3$

$t_6 = t_5 - t_4$

$t_7 = t_6 + h$

$S = t_7$

(i) Quadruple Representation:

S.No	Operator	Operand1	Operand2	Result
1	*	b	c	t_1
2	/	t_1	f	t_2
3	*	t_2	g	t_3
4	*	d	e	t_4
5	+	a	t_3	t_5
6	-	t_5	t_4	t_6
7	+	t_6	h	t_7
8	:=	t_7	-	S

(ii) Triple Representation:

S.No.	Operator	Operand1	Operand2
1	*	b	c
2	/	(1)	f
3	*	(2)	g
4	*	d	e
5	+	a	(3)
6	-	(5)	(4)
7	+	(6)	h
8	:=	S	(7)

(iii) Indirect Triple Representation

	Statement
(1)	(10)
(2)	(11)
(3)	(12)
(4)	(13)
(5)	(14)
(6)	(15)
(7)	(16)
(8)	(17)

No.	Operator	Operand1	Operand2
(10)	*	b	c
(11)	/	(10)	f
(12)	&	(11)	g
(13)	*	d	e
(14)	+	a	(12)
(15)	−	(14)	(13)
(16)	+	(15)	h
(17)	:=	~~(16)~~ s	(16)

Ques: $-(a+b) * (c+d) + (a+b+c)$

Ques: If (a<b and c<d) Generate tac.
then
s = s+1;
else
s = s-1;

Solve:
```
0   if a<b goto 2
1   goto 7
2   if c<d goto 4
3   goto 7
4   t1 = s+1
5   s = t1
6   goto 9
7   t2 = s-1
8   s = t2
```

Ques: if (a<b or c<d)
then
S=S+1;
else
S=S-1;

Solve:
0 if a<b goto 4
1 goto 2
2 if c<d goto 4
3 goto 7
4 $t_1 = S+1$
5 $S = t_1$
6 goto 9
7 $t_2 = S-1$
8 $S = t_2$

Ques: if (a<b or c<d and
then
S=S+1;
else
S=S-1;

Solve:
0 if c<d goto 2
1 goto
2 if e<f goto 6
3 goto
4 if a<b goto
5 goto 9

6 $t_1 = s + 1$

7 $s = t_1$

8 goto 11.

9 $t_2 = s - 1$

10 $s = t_2$