21-03-2025

Syllabus – "Weightage 4 to 6 marks"

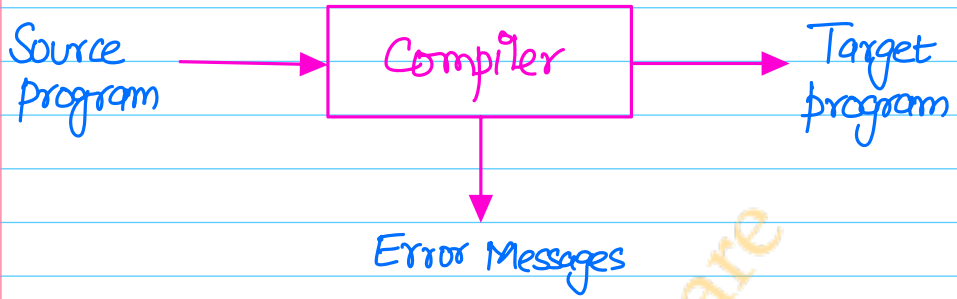
1. Introduction to Compiler
2. Lexical Analysis
3. Syntax Analysis (Parser) } Most important
4. Syntax Directed translation }
5. Intermediate Code Generation – Very important for GATE 2026
6. Run time environments – Theoretical topic
7. Code optimisation – Live variable analysis is very important

Book – O. G Kakde

"Solve all practice questions given in the Book"

— Introduction to Compiler —

A Compiler is a program that reads a program written in one language the source language and translates into an equivalent program in another language - the target language. The translation process should also report the presence of errors in the source program.



Read again -

1. "A Compiler is a translator which translates source language into target language. The target language could be -
   a) Low-level assembly / machine code
   b) Intermediate representation (Bytecode)
   c) Another high-level language."

2. A Compiler does not always translate a program directly into low-level machine code. Instead, its target code may be another programming which must then be further Compiled or interpreted to produce machine code

3. A Compiler doesn't necessarily have to generate machine code directly. Its target code could be -
   a) Machine Code. C-Compiler GCC, Clang directly generates assembly/machine code for x86, ARM etc.

   b) Java Compiler (Javac) translates Java into Java bytecodes not machine. It is an intermediate code. •NET languages translated into CIL (common Intermediate Language)

"These are later executed by virtual machine or JIT Compiler"

Not for GATE –

1) Emscripten – Compiles C/C++ to JavaScript/webassembly
2) Typescript Compiler compiles typescript to JavaScript
3) Kotlin/Scala Compilers Compiles into Java bytecode
4) GCJ (GNU Compiler for Java) Compiles Java source into either native code or C (now discontinued).

Interpreter – An Interpreter is a program that executes other programs. The interpreter and the underlying machine are combined into virtual machine and Simulate the execution of the input program on the virtual machine.

OR

An Interpreter is a program that –
Directly executes instructions written in a programming language, line by line or statement by statement without producing a separate machine code executable file.

"The source code itself is read, analyzed and executed immediately.

Python Interpreter executes .py Scripts without generating .exe files.

a) It reads source code line by line
b) Checks for errors
c) Executes the instructions immediately
d) moves to the next line.

"Errors are reported as soon as they are encountered".

High level & low-level programming languages

Low-level Language - Directly deals with hardware, registers, memory addresses. It is fast and efficient

" In low level languages (Machine/Assembly), each statement corresponds to at most one operation. It may involve upto 2-3 operands depends on instruction set."

ex- Machine Language
Assembly language

High level language - closer to human thinking. Provide abstraction from hardware means we don't care about registers/memory addresses. They are easy to read, write and debug.

ex- C, fortran, Java, Python, C++, Ruby etc.

" Each statement may have any number of operands
a = a*b+c/d^e;".

Generations of programming Languages -

First generation - Machine language. Extremely hard to program

Second generation - Assembly Language. Machine dependent and uses Mnemonics (MOV, ADD, JMP)

Third generation - High level languages. English language like Syntax C, C++, Java, Pascal, fortran, Python, Ruby, Lisp. Requires Compilers or Inter

- prefers. It is portable accross platforms.

Fourth generation - Very high level. focus on problem solving rather than hardware details. Used in databases, report generation, query languages.

ex - SQL, MATLAB

Fifth-generation - Artificial Intelligence languages Computers solve problems using logic, constraints, AI techniques.

ex - Prolog, LISP, Mercury etc.

Can we convert high generation programs into low generation? yes possible.

Program and programming language—

Program– A program is essentially a finite sequence of instructions that tell a computer what to do.

OR

A program is a specification of what data computer is required to process, by using what operations, and by going in what sequence.

Programming language— A programming language is a way to talk to a computer and tell it what to do.

OR

A programming language is a set of rules and symbols used to write instructions that a computer can understand and execute.

" Every computer is designed to understand one language which is called as Machine language of that Comp-uter and it is a language whose alphabet contains only two symbols 0, 1.

Why do we need Compiler— We need compilers because they translate human readable code into machine-executable instruc-tions, optimize it for performance, and check for errors.

1. Bridge between humans and machines
2. optimization
3. portability – C code can be compiled on Windows, Linux or Mac into their respective machine code.
4. Error checking
5. productivity – Without compilers the whole process of

programming will be slow and error prone.

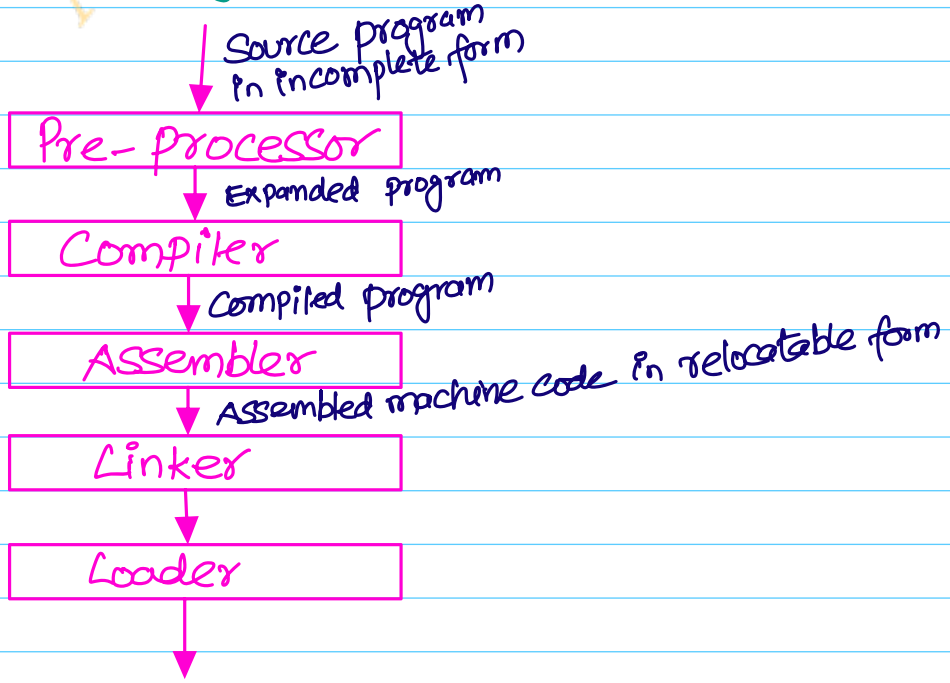
"We can work without Compilers but it would be very difficult and impractical in most cases."

Why do humans write programs in High-Level Languages —

Humans prefer high-level languages for —

1. Readability & Simplicity. They use words and symbols close to human logic ex- if, while, for, printf
2. Programs are shorter and quicker to write. Printing "Hello world" in python is one line but in assembly language it takes 10-15 lines.
3. Error reduction & portability

— Steps in Program execution —



Pre-processor → The preprocessor run before the actual Compilation. It processes all preprocessor directives i.e lines beginning with # in the program/code

#include <stdio.h>, preprocessor directive #include includes the contents of the specified file i.e stdio.h. The pre-processor copies the entire content of stdio.h into your source file. Replaces all macros with their values.

Compiler - It converts preprocessed program of C into assembly code for our specific processor architecture like x86, ARM etc. Compiler works in six phases to convert source code into assembly code.

Six phases of Compiler are —
Lexical analyzer, Syntax analyzer, Semantic analyzer, Intermediate code, Code optimisation & Code generation.

" The Compiler produces an assembly language file with .S extension. The file is then passed to the assembler in the next step."

Can we use a compiler which can directly compile our code into machine code without using assembler?
Yes, we can use a compiler that directly translates code into machine code without the explicit intermediate "assembly" step.

OR

Some compilers can skip generating human-readable assem-bly and instead directly generate machine code .obj files.

.Gcc/clang are commonly used for c/c++ by default they compile directly into object code

At its core, the compiler is a translator it doesn't just translate blindly. It goes through multiple phases. It also find errors. Error reporting is integral part of the compiler. The compiler cannot generate correct code unless the input program is syntactically and semantically correct. Without error handling, the compiler would just crash or generate garbage machine code.

" The compiler must also provide error recovery, so it doesn't stop at the first error but continues checking for others.

Assembler → After the compiler translates a high-level program into assembly language mnemonics like MOV, ADD etc. The assembler takes over now. Its job is to convert assembly language into machine code (object code).

There are two types of Assemblers –

a) One-pass assembler
b) Two-pass assembler } Not for GATE

" It converts human readable assembly code into binary machine code (object code). The output is an object file ready for the linker."

Linker →

Compiler → generates assembly code
Assembler → Converts assembly to object file (.obj)
Linker → It takes one or more object files + library files, and combines them into a single executable file

The main job of linker is to combine all object files from different modules i.e main program + functions + libraries, resolve symbols, produce a final executable file

Important –

1. scanf and printf are not C keywords
2. Keywords are reserved words defined by the C language itself like if, for, while, int etc.
3. scanf and printf are functions provided by the C standard library, not part of the Core C language.
4. Library functions in stdio.h – They are standard library functions declared in the header file. These are just declarations not actual code. Header file only provides function declaration. They do not contain compiled code. They tell the compiler about the function signatures, so that type checking can happen during compilation
5. Where the actual code resides? The compiled code of printf & scanf is stored in a precompiled library file. During linking, the linker connects your function calls with the corresponding compiled code.
6. The compiler only checks – " Does a function called printf exist with the given signature?

   yes, found in stdio.h

   The linker then ensures " where is the actual machine code for printf " found in standard library. The linker fills the hole by attaching the library code into final executable

Summary –

scanf and printf are not C keywords. They are standard library functions in stdio.h. The header file only contains their declarations. The actual compiled code resides in the C standard library, which is linked by the linker during to produce the final executable file.

Loader – Loads the executable into memory and prepares it for execution. Loader comes after the linker, just before the CPU starts running the program.

Cross-Compiler –

A cross compiler is a compiler that runs on one machine but generates executable code for a different machine with a different architecture or operating system.

Host System – where the compiler itself runs

Target System – where the compiled program will run.

We need cross-compiler because devices like micro-controllers, routers, IOT devices often cannot run a full compiler. There are many other usages also.

Analysis & Synthesis model of Compiler –

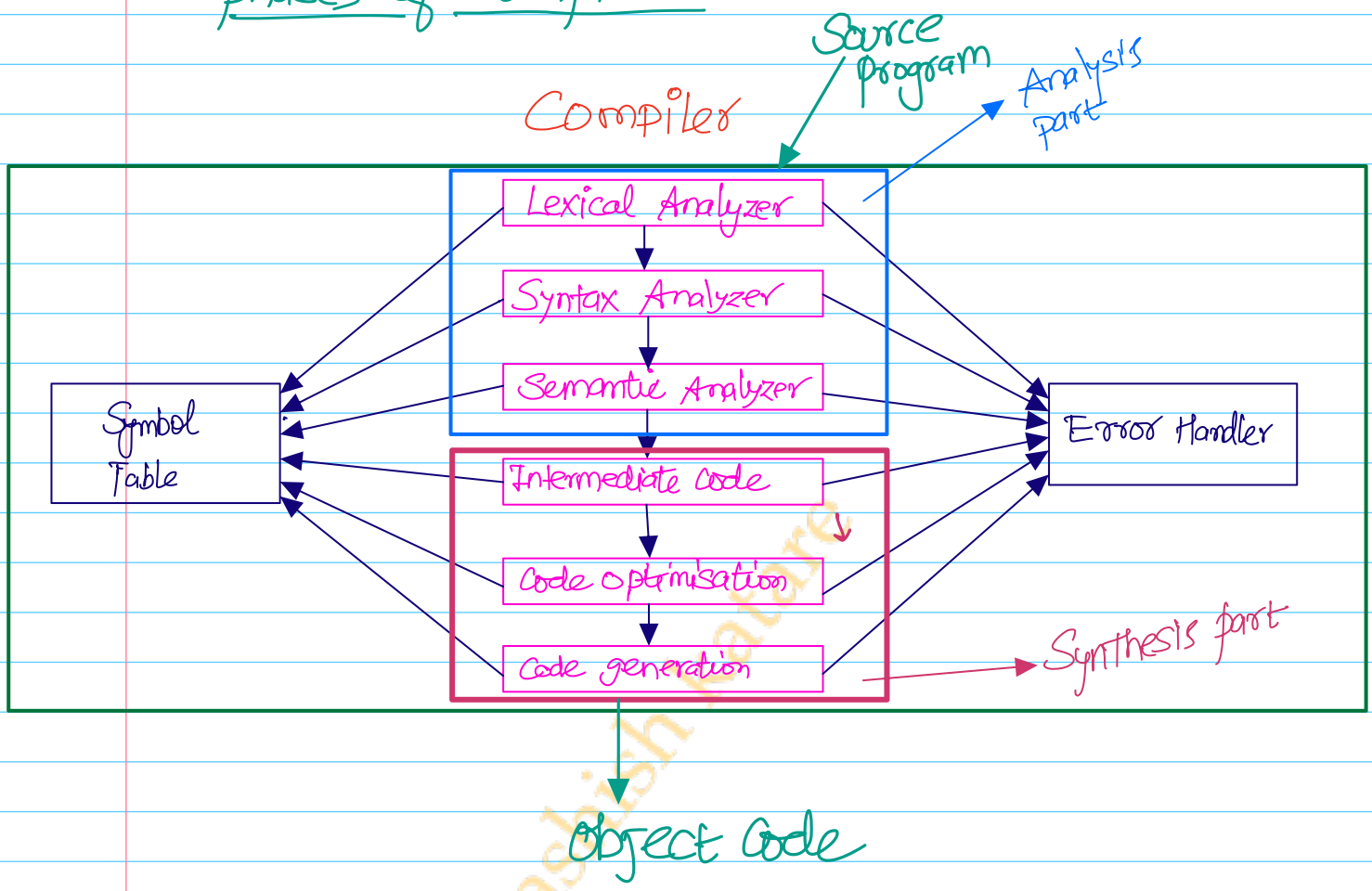
Analysis → Breaks the code and understand the source program.
Analysis part is also known as front end
Output of analysis part is Intermediate code

Synthesis – Generate target code (assembly or machine code)
Synthesis means "to combine"
Synthesis part is known as back end

"Intermediate code is produced after analysis but before final target code generation. It serves as a bridge between front-end and back end."

" We Consider Intermediate code of Under Synthesis back end phase".

Phases of Compiler –



Example – $x = a+b+c*d-e/f\uparrow g$

We will go through working of all the phases of Compiler

Lexical analysis – The lexical analysis phase reads the characters in the source program and groups them into streams of tokens, each token represents keywords, identifiers, Constants, operators.

" Lexical analyzer is also known as Scanner "

Goal → Convert code into tokens

Output → Stream of tokens

" The code is scanned left to right and it breaks into tokens"

In our example $x = a+b+c*d-e/f\uparrow g$ there are 15 tokens.

" We will study Lexical analyzer in detail in coming topics"

Syntax analysis- The second phase of the Compiler is Syntax analysis or parsing. The parser uses the first components of the token produced by the Lexical analyzer to create tree like structure that depicts grammatical structure of the token stream.

" Parser is used in Syntax analysis phase for parsing".

" Lexical analyzer identifies the words in a sentence, Syntax analysis ensures those words form a grammatically Correct sentence".

" Without Syntax analysis Compiler Cannot know program structure".

A Compiler is a program that uses a formal grammar usually a context free grammar to describe the syntactic structure of a programming language. During syntax analysis phase, the Compiler parses the entire source code to check whether it conforms to this grammar. If the parsing succeeds, the program is syntactically Correct, otherwise Compiler reports Syntax errors.

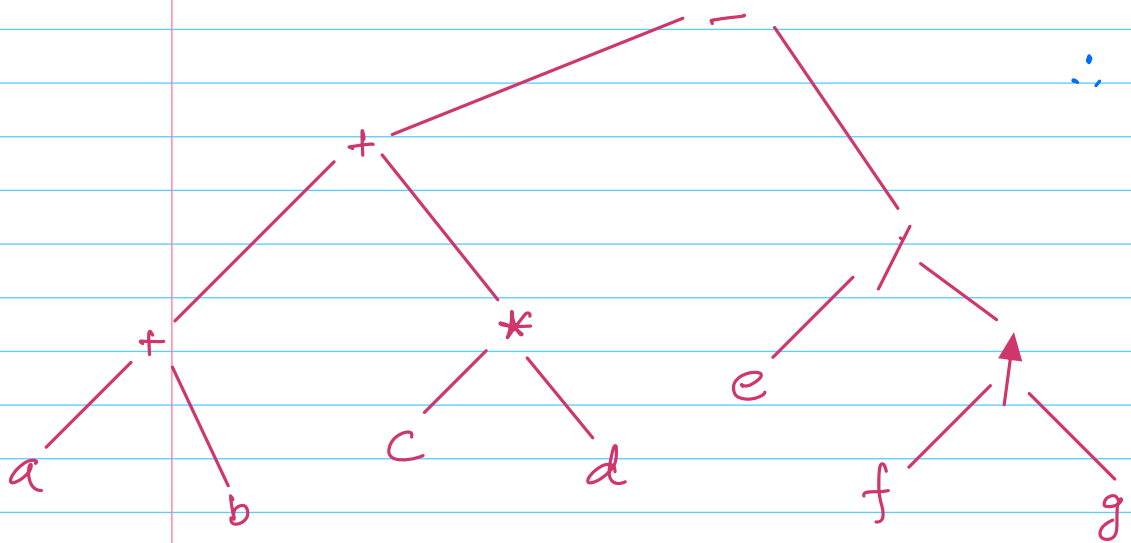
Further phases, Semantic analysis type checking may still detect additional errors beyond Syntax.

Parser's role – Parser takes the entire token stream and tries to match it against the grammar of the language (CFG). It checks whether whole program can be derived from the start symbol of the grammar.

The Compiler is based on one formal grammar i.e a CFG written in BNF that describes the entire syntax of the C- language. Every valid C- program is a string that belongs to the language defined by this grammar. The Compiler's parser check this membership. Semantic rules like type checking and scope are enforced separately after parsing.

Example – $x = a+b+c*d-e/f\uparrow g$

$$x = \Big(\big((a+b)+(c*d)\big)-\big(e/(f\uparrow g)\big)\Big)$$



∴ a/b a is divided by b

Semantic analysis - The Semantic analyzer uses syntax tree and the information in Symbol table to check the source program for Semantic Consistency with the language definition.

"Syntax analysis ensures the program follows the grammar of the language, Semantic analysis ensures the program makes sense according to the language's meaning.

Syntax analysis → Is the sentence grammatically Correct?

Semantic Analysis → Does the sentence actually make sense?

Main tasks of Semantic analysis -

(1) Type checking
(2) Declaration checking
(3) Scope resolution
(4) Consistency checking
(5) Building annotated Syntax tree

4. Intermediate Code generation - In the process of translating a source program into target code Compiler may Construct one or more intermediate representations, which can have a variety of forms.

"Syntax trees are a form of intermediate representation, they are commonly used during Syntax & Semantic analysis".

"After Syntax and Semantic analysis of the source program many compilers generate an explicit low level or machine like intermediate code".

"Intermediate code must be easy to produce and it should be

easy to translate into the target machine language.

Is intermediate code representation compulsory in all compilers? No. Compilers can directly translate source code into target code but modern compilers use it. It makes the design simpler, modular, portable and easier to optimize.

The main reasons compilers use IR—

a) Portability
b) Simplifies compiler design
c) Optimization
d) Machine independence

Types of IR— These are three broad categories of IR, depending on how "close" they are to the source or target language.

a) Graph based
   - Syntax tree
   - DAG

b) Linear
   - 3-address code (SSA)
   - postfix

3-address code—

$$x = (a+b)+(c*d)-(e/(f \uparrow g))$$

$t_1 = f \uparrow g$
$t_2 = e / t_1$
$t_3 = c * d$
$t_4 = a + b$
$t_5 = t_4 + t_3$
$t_6 = t_5 - t_2$
$x = t_6$

Code optimisation – The code optimisation phase of a compiler improves the intermediate code to make the final target code.

" To make the code time & space efficient".

There are two categories of optimizations

a) Machine independent → Applied on IR before knowing the exact target machine.

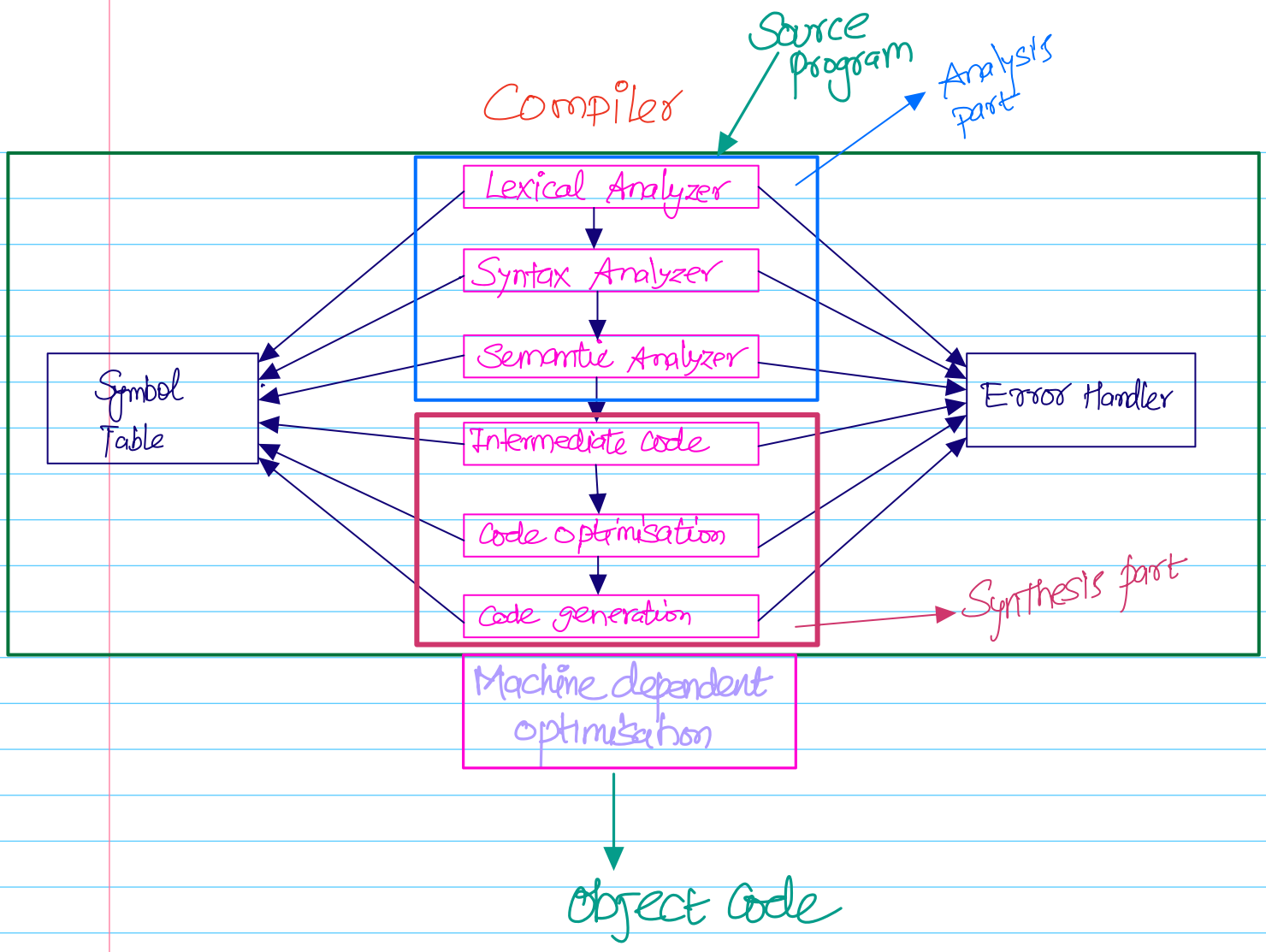
Its focus is on improving algorithmic efficiency and reducing redundancy. These are general purpose applied to all machines.

ex – Constant folding
Dead code elimination
strength reduction
Common subexpression elimination
Loop optimisation.

b) Machine dependent – It is done during target code generation. It is done based on specific target machine. It improve execution speed and memory usage on that machine.

a) Register allocation
b) Cache optimisation
c) Peephole optimisation

Some of the above mentioned optimisation techniques belong to both machine dependent and independent optimisation.



Symbol table management – An essential function of a Compiler is to record the variable names used in the source program and collect information about various attributes of each name.

" It is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly".

"Symbol table is a central repository of semantic information needed during Compilation".

It is like a directory or hash table that maps identifiers, to their attributes such as –

Type (eg :- int, float)
Scope (eg :- global, local, class-level)
Memory location (address or offset)
Size
Value (optional, for constants)
function parameters
Line numbers (where it is declared/used)

" Implementation techniques – Linear list, Hash table, AVL trees ".

Error Handler – It is the process by which a compiler detects, reports, and recovers from errors in the source program so that compilation can continue as much as possible.

(i) Detects error
(ii) provide clear message to programmer
(iii) Attempt recovery, so the compiler does not stop after the first error.

Some examples –

(I) Lexical errors
(II) Syntax errors
(III) Semantic errors – Correct Syntax but wrong meaning ex– Type mismatch, wrong function arguments

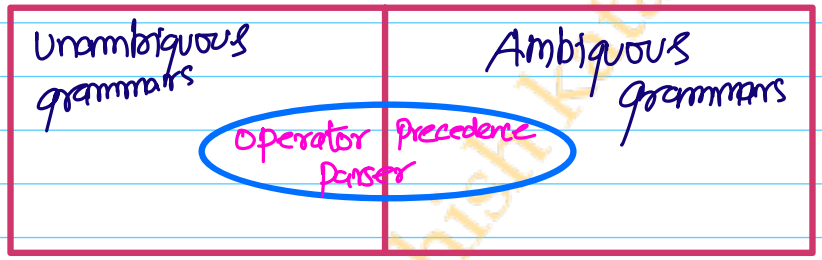
(IV) Logical errors not caught by Compilers, but by programmers.

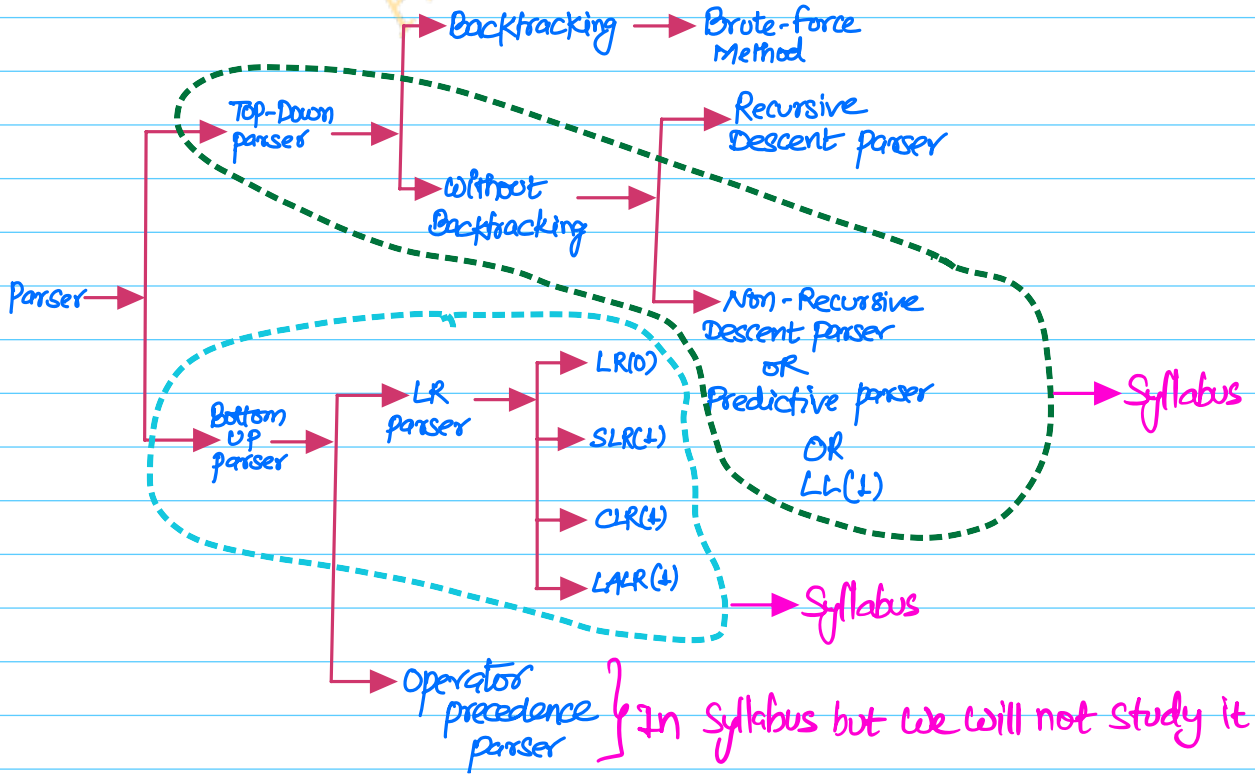
(v) Runtime errors – Detected during execution, not Compilation.

ex- Divide by zero, Null pointer deference, Out of memory.

Syntax Analysis – 1. Takes tokens from lexical analyzer as input

2. Produces parse tree as output
3. Checks grammatical structure of source program according to the grammar of the language.
4. Uses context free grammar for specification
5. It reports syntax errors.





Note:- We are focussing only on LL(1). Below points are based on LL(1)

Top-Down parser— A top down parser is a type of parser that constructs the parse tree from the root (start variable) down to the leaves. It tries to expand the start symbol into the input string using grammar productions.

(1) Parsing starts from the start Symbol of the grammar
(2) It uses Left most derivation strategy.
(3) The parser predicts productions and matches input Step by Step.
(4) LL(1) parser is a special type of non-recursive descent parser.
(5) It is a table driven parser. It uses parsing table
(6) LL(1) parser is possible only for grammars that are free from left recursion & left factor.
(7) Not possible for ambiguous grammar.
(8) It is very fast, needs linear time $O(n)$ but less powerful than LR parsers.

Note:- If CFG is given then we can always remove left recursion from the given useful grammar

Remove Left recursion – Revise from TOC notes

1. $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid a$

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

2. $S \to aBDh$
$B \to Bb \mid h$
$D \to Ef$
$E \to g \mid \epsilon$
$F \to f \mid \epsilon$

$S \to aBDh$
$B \to hB'$
$B' \to bB' \mid \epsilon$
$D \to EF$
$E \to g \mid \epsilon$
$F \to f \mid \epsilon$

3. $S \to (L) \mid a$
$L \to L, S \mid S$

$S \to (L) \mid a$
$L \to SL'$
$L' \to , SL' \mid \epsilon$

4. $S \to A$
$A \to Ad \mid Ae \mid aB \mid aC$
$B \to bBe \mid f$

$S \to A$
$A \to aBA' \mid aCA'$
$A' \to dA' \mid eA' \mid \epsilon$
$B \to bBc \mid f$

5. $A \to b \mid Bd$
$B \to Bc \mid Ac$

$A \to b \mid Bcd \mid Acd$
$B \to Bc \mid bc \mid Bdc$

↓

$A \to bA' \mid BcdA'$
$A' \to cdA' \mid \epsilon$
$B \to bcB'$
$B' \to cB' \mid dcB' \mid \epsilon$

6. $S \to aAcBe$
$A \to Ab \mid b$
$B \to d$

$S \to aAcBe$
$A \to bA'$
$A' \to bA' \mid \epsilon$
$B \to$

7. $S \to L \cdot L \mid L$
$L \to LB \mid B$
$B \to 0 \mid 1$

$S \to L \cdot L \mid L$
$L \to BL'$
$L' \to BL' \mid \epsilon$
$B \to 0 \mid 1$

Left factor the grammar — Also known productions with Common prefixes

Left factor is the Common prefix of two or more productions for the same non-terminal.

EX- $A \to abA \mid aA \mid b$ → Here left factor is 'a'

Left factoring is the process of removing such Common prefixes to make grammar suitable for predictive parsing

"Common prefix = Left factor"

"Left factoring means removal of left factor"

What is the problem? Why we remove left factor?

When a grammar has Common prefixes (left factors) in productions of non-terminal, a predictive parser Cannot decide which production to use just by looking at the next input symbol.

Will there be any difference in language after removing left factor? No, grammar will remain same semantically.

"Yes, we Can remove left factor from every CFG".

Remove left factors / Common prefixes / Left factoring the grammar —

1. $A \to abA \mid aA \mid b$

$A \to aX \mid b$
$X \to bA \mid A$

↓

$A \to aX \mid b$
$X \to bA \mid aX \mid b$

↓

$A \to aX \mid b$
$X \to bY \mid aX$
$Y \to A \mid \epsilon$

2. $A \to abb \mid abA \mid ab$

$A \to abX$
$X \to b \mid A \mid \epsilon$

3. $A \to a \mid aA \mid aAb \mid ab$

$A \to aX$
$X \to A \mid Ab \mid b \mid \epsilon$ → $A \to aX$, $X \to AY \mid b \mid \epsilon$, $Y \to b \mid \epsilon$

How to find first and follow set –

First Set → The first set of a grammar symbol (variables) is the set of all terminals that can appear as the first symbol in some string derived from that symbol.

Follow Set →

1. If the production is of type –

   $S \rightarrow \alpha A \beta$ where α, β are strings of grammar symbols and variables

   follow(A) = First(β) Copy first(β) into follow(A)

2. If the production is of type –

   $S \rightarrow \alpha A \beta$ where α, β are strings of grammar symbols/variables and $\beta \xrightarrow{*} \epsilon$

   follow(A) = First(β) Copy first(β) into follow(A)

   AND

   follow(A) = follow(S) Copy follow(S) into follow(A)

3. If the production is of type –

   $S \rightarrow \alpha A$ where α is string of grammar symbols and variables

   follow(A) = follow(S) Copy follow(S) into follow(A)

4. Include '$' in follow of start variable

Important points –

1. First & follow are sets hence order of elements don't matter.
2. First set & follow sets always contain terminals. They can never contain variables.
3. First set can contain 'ϵ' but can never contain $.
4. Follow set can never contain 'ϵ' but it can have $.
5. Intersection of first and follow set of a variable may be ϕ but not always.
6. We find first set & follow set only for variables.
7. Suppose grammar G has left recursion and G' is the grammar after removing left recursion. First & follow sets for variables of G and G' will be different.

First and follow sets –

1. $S \rightarrow AB$
   $A \rightarrow aA \mid b \mid \epsilon$
   $B \rightarrow bB \mid c \mid d \mid \epsilon$

	First	Follow
S	a, b, c, d, ϵ	$
A	a, b, ϵ	b, c, d, $
B	b, c, d, ϵ	$

2. $S \rightarrow ASB \mid c$
   $A \rightarrow aA \mid aAb \mid a$
   $B \rightarrow bB \mid d \mid \epsilon \mid aBS$

	First	Follow
S	a, c	$, b, d, a, c
A	a	a, c, b
B	b, d, a, ϵ	$, b, d, a, c

3. $S \to ACB \mid CbB \mid Ba$
$A \to da \mid BC$
$B \to g \mid \epsilon$
$C \to h \mid \epsilon$

	First	Follow
S	d, g, h, ϵ, b, a	$
A	d, g, h, ϵ	g, h, $
B	g, ϵ	$, a, h, g
C	h, ϵ	g, $, b, h

4. $S \to AaAb \mid BbBa$
$A \to c \mid \epsilon$
$B \to d \mid \epsilon$

	First	Follow
S	a, b, c, d	$
A	c, ϵ	a, b
B	d, ϵ	b, a

5. $S \to aABb$
$A \to c \mid \epsilon$
$B \to d \mid \epsilon$

	First	Follow
S	a	$
A	c, ϵ	d, b
B	d, ϵ	b

6. $S \to L=R \mid R$
$L \to *R \mid id$
$R \to L$

	First	Follow
S	*, id	$
L	*, id	=, $
R	*, id	$, =

7. $S \to (L) \mid a$
$L \to L,S \mid s$

	First	Follow
S	(, a	$,), ,
L	(, a	, ,)

8. $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

	First	Follow
E	(, id	$, +,)
T	(, id	$, +,), *
F	(, id	$, +,), *

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

	First	Follow
E	(, id	$,)
E'	+, ϵ	$,)
T	(, id	+, $,)
T'	*, ϵ	+, $,)
F	(, id	*, +, $,)

10. $S \to AB$
$A \to aA \mid \epsilon$
$B \to bB \mid \epsilon$

	First	Follow
S	a, b, ϵ	$
A	a, ϵ	b, $
B	b, ϵ	$

Rules to Construct LL(1) parser –

1. Remove left recursion, if reqd
2. Left factor the grammar, if reqd
3. Find first and follow set.
4. Construct the table.

} follow the order

Q Construct LL(1) parser for the given grammar –

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

Remove left recursion →

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

} No need to left factor

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

	First	Follow
E	(, id	$,)
E'	+, ϵ	$,)
T	(, id	+, $,)
T'	*, ϵ	+, $,)
F	(, id	*, +, $,)

Parsing table of LL(1) –

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

E→TE'
E'→+TE'|ϵ
T→FT'
T'→*FT'|ϵ
F→(E)|id

ω = id+id*id$

Stack	Input
E$	id+id*id$
TE'$	id+id*id$
FT'E'$	id+id*id$
idT'E'$	id+id*id$
TE'$	+id*id$
E'$	+id*id$
+TE'$	+id*id$
TE'$	id*id$
FT'E'$	id*id$
idT'E'$	id*id$
T'E'$	*id$
*FT'E'$	*id$
FT'E'$	id$
idT'E'$	id$
T'E'$	$
E'$	$
$	$

2. Construct LL(1) parsing table for the given grammar –

S→iEtSS'
S'→ϵ|eS
E→b

} No left recursion
} No left factor

	First	Follow
S	i	$,e
S'	e,ϵ	$,e
E	b	t

	i	t	e	b	$
S	S→iEtSS'	Error	Error	Error	Error
S'	Error	Error	S'→eS, S'→ϵ	Error	S'→ϵ
E	Error	Error	Error	E→b	Error

Conflict
Multiple entries

Above grammar is not LL(1)
Above grammar is ambiguous on "ibtibtaea"

3. Construct LL(1) parsing table for the given grammar –

S→ L=R | R
L→ *R | id
R→ L

} No left recursion
Left factor is there

S→ L=R | L
L→ *R | id
R→ L

Put R→L

Left factoring the grammar –

S→ L=R | L
L→ *R | id
R→ L

→

S→ LX
X→ =R | ϵ
L→ *R | id
R→ L

} No left factor

	=	*	id	$
S	Error	S→LX	S→LX	Error
X	X→=R	Error	Error	X→ϵ
L	Error	L→*R	L→id	Error
R	Error	R→L	R→L	Error

	First	Follow
S	*, id	$
X	=, ϵ	$
L	*, id	=, $
R	*, id	$, =

S→ L=R | R
L→ *R | id
R→ L

} Try without removing left factor

	First	Follow
S	*, id	$
L	*, id	=, $
R	*, id	$, =

	=	*	id	$
S	Error	S→L=R, S→R	S→L=R, S→R	Error
L	Error	L→*R	L→id	Error
R	Error	R→L	R→L	Error

multiple entry

Grammar is not LL(1)

4. Construct LL(1) parsing table for the given grammar—

$S \to aSa \mid bSb \mid \epsilon$ No left factor, No left recursion, Not ambiguous, Not DCFL

↳ $L = \{ww^R / w \in (a,b)^*\}$

	First	follow
S	a, b, ϵ	$, a, b

	a	b	$
S	S→aSa, S→ϵ	S→bSb, S→ϵ	S→ϵ

multiple entry

Above grammar is not LL(1)

5. Construct LL(1) parsing table for the given grammar—

$S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$

No Left Recursion, No left factor, not Ambiguous

	First	follow
S	a, b	$
A	ϵ	a, b
B	ϵ	a, b

	a	b	$
S	S→AaAb	S→BbBa	
A	A→ϵ	A→ϵ	
B	B→ϵ	B→ϵ	

Above grammar is LL(1)

Alternate method to check whether grammar is LL(1) or Not - (Short cut)

1. If the production of the grammar —

$A \to \alpha_1 | \alpha_2 | \alpha_3$ then -

$First(\alpha_1) \cap First(\alpha_2) = \phi$
$First(\alpha_1) \cap First(\alpha_3) = \phi$
$First(\alpha_2) \cap First(\alpha_3) = \phi$

2. If the production of the grammar —

$A \to \alpha_1 | \alpha_2 | \alpha_3 | \epsilon$ then
$First(\alpha_1) \cap First(\alpha_2) = \phi$, $Follow(A) \cap First(\alpha_1) = \phi$
$First(\alpha_1) \cap First(\alpha_3) = \phi$, $Follow(A) \cap First(\alpha_2) = \phi$
$First(\alpha_2) \cap First(\alpha_3) = \phi$, $Follow(A) \cap First(\alpha_3) = \phi$

Example -

1. $E \to \overset{\alpha_1}{E+T} | \overset{\alpha_2}{T}$ $First(E+T) \cap First(T) = \{(, id\}$ is not ϕ
violation of property hence Grammar is not LL(1)

$T \to T*F | F$ $First(T*F) \cap First(F) = \{(, id\}$ is not ϕ
violation of property hence Grammar is not LL(1)

$F \to (E) | id$ $First((E)) \cap First(id) = \phi$

Not LL(1)

2. $E \to TE'$
$E' \to +TE' | \epsilon$ $first(+TE') \cap follow(E') = \phi$
$T \to FT'$
$T' \to *FT' | \epsilon$ $First(*FT') \cap follow(T') = \phi$
$F \to (E) | id$ $First((E)) \cap first(id) = \phi$

satisfies the property

It is LL(1)

3. $S \rightarrow aSa \mid bSb \mid \epsilon$

First(aSa) ∩ first(bSb) = ∅
First(aSa) ∩ follow(S) = {a}
First(bSb) ∩ follow(S) = {b}
} violation of property

Above grammar is not LL(1)

Important points –

1. A grammar can be LL(1) only if it is not ambiguous, not left recursive and it must not contain left factor.

2. If the grammar is unambiguous, not left recursive and there is no left factor still it may not be LL(1).

3. Regular languages can be LL(1) because RL are always unambiguous hence unambiguous grammars are possible without left recursion and left factor.

"Regular grammars are not always LL(1)"

4. Removing left recursion & left factors don't make grammar LL(1).

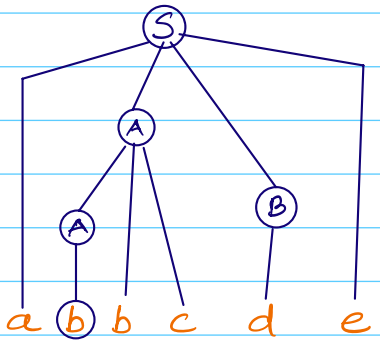
Bottom Up parser — It uses RMD in reverse "Reverse RMD".

1. Bottom up parsers reduce the input string ω to the start symbol of a grammar by tracing out the right most derivation of ω in reverse.

"It is equivalent to constructing a parse tree for input string ω by starting with leaves and proceeding toward the root that is, attempting to construct the parse tree from the bottom up

2. The reason why bottom-up parsing tries to trace out the right most derivations of an input string ω in reverse and not the leftmost derivations is because the parser scans the input string ω from left to right, one symbol/token at a time. To trace out right most derivations of an input string ω in reverse, the tokens of ω must be made available in left to right order.

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$
$\omega = abbcde$



$S \rightarrow a\,A\,B\,e$

$S \rightarrow a\,A\,d\,e$

$S \rightarrow a\,A\,b\,c\,d\,e$

$S \rightarrow a\,b\,b\,c\,d\,e$

Steps to Construct Parsing table of Bottom-up parsers -

1. Expand the grammar
2. Number all the productions of the grammar
3. Augment the grammar
4. Find first & follow sets
5. Find LR(0) Canonical item sets. They are also known as LR(0) DFA
6. Construct parsing table of respective parser.

SLR(1) :- Simple LR parser

1. Construct SLR(1) parsing table for given grammar

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

Step-1 – Expand the grammar

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

Step-2 – Number all the productions

1. $E \to E+T$
2. $E \to T$
3. $T \to T*F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

Step-3 – Augment the grammar

$E' \to E$ (Augmented production)

1. $E \to E+T$
2. $E \to T$
3. $T \to T*F$
4. $T \to F$
5. $F \to (E)$
6. $F \to id$

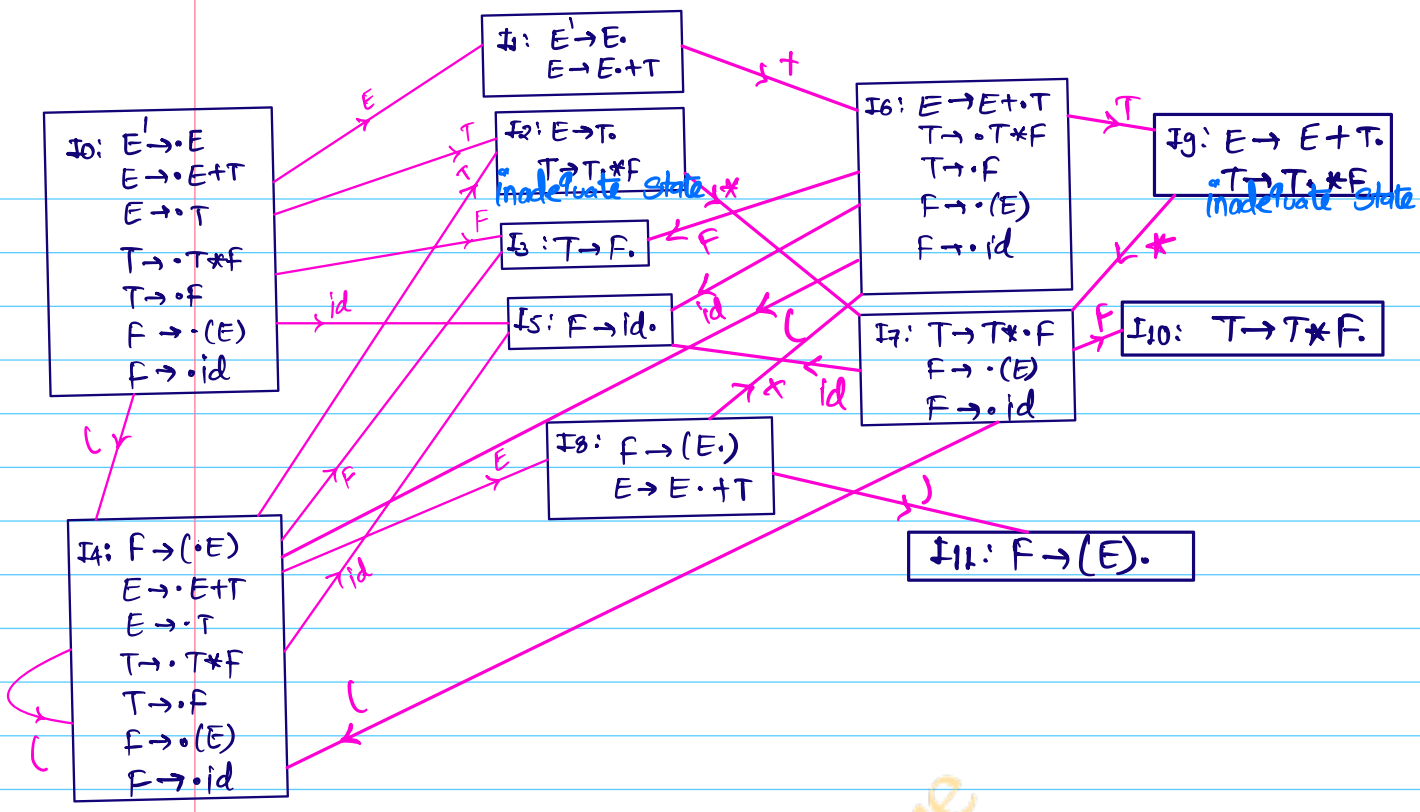
Grammar is augmented when we use augmented production in it.

Step-4 – Find First & Follow Set

	First	Follow
E	(, id	$, +,)
T	(, id	$, +,), *
F	(, id	$, +,), *

Step-5 – Find LR(0) Canonical Item set. It is known as LR(0) DFA

I_0
$E' \to \cdot E$
$E \to \cdot E+T$
$E \to \cdot T$
$T \to \cdot T*F$
$T \to \cdot F$
$F \to \cdot (E)$
$F \to \cdot id$

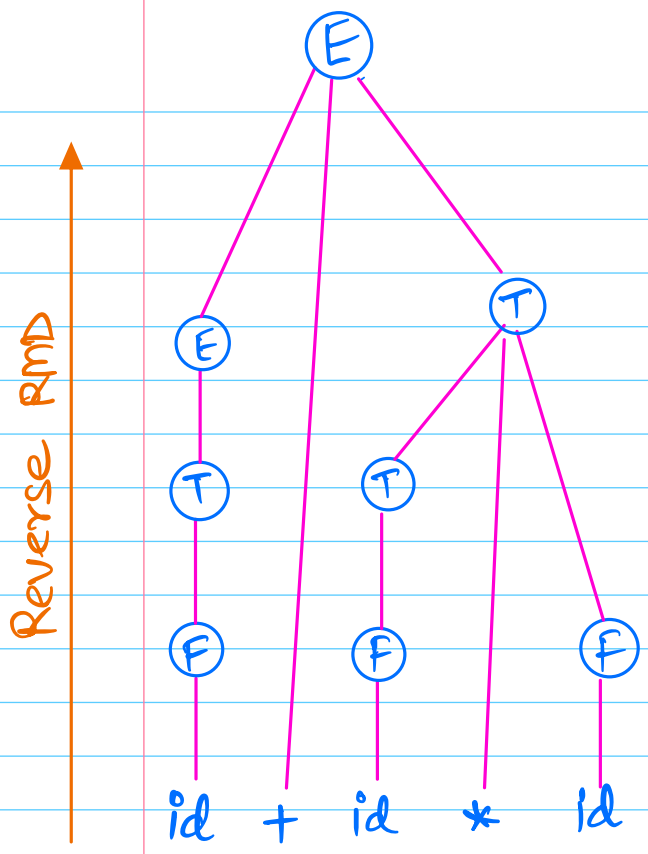


	+	*	(	)	id	$	E	T	F
I_0	error	error	S_4	error	S_5	error	1	2	3
I_1	S_6	error	error	error	error	Accepted	error	error	error
I_2	r_2	S_7	error	r_2	error	r_2	error	error	error
I_3	r_4	r_4	error	r_4	error	r_4	error	error	error
I_4	error	error	S_4	error	S_5	error	8	2	3
I_5	r_6	r_6	error	r_6	error	r_6	error	error	error
I_6	error	error	S_4	error	S_5	error	error	9	3
I_7	error	error	S_4	error	S_5	error	error	error	10
I_8	S_6	error	error	S_{11}	error	error	error	error	error
I_9	r_1	S_7	error	r_1	error	r_1	error	error	error
I_{10}	r_3	r_3	error	r_3	error	r_3	error	error	error
I_{11}	r_5	r_5	error	r_5	error	r_5	error	error	error

← Action table → * → goto →

Inadequate State means where we have a possibility of some Conflict. Inadequate State doesn't mean that it has 100% chance of Conflict. It always say "possibility".

In our example I_2 & I_9 are inadequate State but have no Conflict.



Stack	Input
$0	id+id*id$
$0id5	+id*id$
$0F3	+id*id$
$0T2	+id*id$
$0E1	+id*id$
$0E1+6	id*id$
$0E1+6id5	*id$
$0E1+6F3	*id$
$0E1+6T9	*id$
$0E1+6T9*7	id$
$0E1+6T9*7id5	$
$0E1+6T9*7F10	$
$0E1+6T9	$
$0E1	$

Q. What is right sentential form?

A right sentential form is a concept from derivations. A sentential form is any string of grammar symbols (both terminals and non-terminals) that can appear at some step during a derivation from the start symbol.

$S \rightarrow AB$
$A \rightarrow aA/a$
$B \rightarrow bB/b$

$S \rightarrow AB$
$S \rightarrow aAB$
$S \rightarrow aaAB$
$S \rightarrow aaaAb$ Sentencial form

Right sentencial form is a sentential form that results from rightmost derivation.

$S \rightarrow AB$
$A \rightarrow aA/a$
$B \rightarrow bB/b$

$S \rightarrow AB$
$S \rightarrow AbB$
$S \rightarrow AbbB$
$S \rightarrow AbbbB$ Right sentential form

Q. What is handle?

A handle is the exact piece of the string that the parser can 'reduce' at that point to move one step closer back to the start symbol.

It is the valid construct to be replaced during shift reduce parsing step.

In other words —

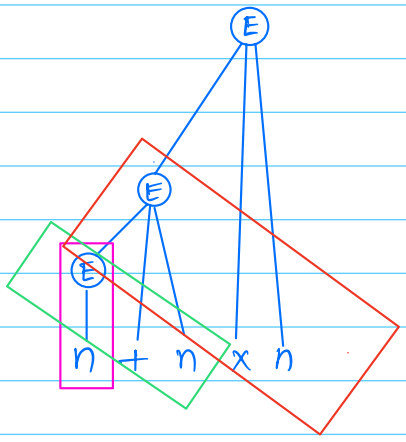
A handle of a right-sentential form is a substring of the sentential form that matches the body of some production rule and whose reduction to the LHS of the production represents one step of the reverse of a rightmost derivation.

Important - Not every production rule in the grammar is the handle at every step. A handle is always context dependent. It depends on which string we are looking at in derivation

Ex- $E \rightarrow E+n \mid Exn \mid n$ for the string $n+n \times n$, the handles are —

$E \rightarrow \boxed{Exn}$
$E \rightarrow \boxed{E}+n \times n$
$E \rightarrow \boxed{n}+n \times n$

$E \rightarrow n$
$E \rightarrow E+n$
$E \rightarrow Exn$
} Handles



$E \rightarrow Exn$
$E \rightarrow n \times n$

$E \rightarrow n$
$E \rightarrow Exn$
} Handles

Handle pruning? The process of repeatedly finding a handle in a right sentential form and reducing it until the start symbol is reached.

Q. What is Viable prefixes ? Handle in the making

A Viable prefix is any prefix of a right Sentential form that does not extend beyond the handle of that sentential form.

In other words, it is a prefix of the input that can appear on the stack of S-R parser without causing an error.

ex- E+T is handle

E
E+
E+T } Viable prefixes

" Prefixes of a valid handle".

Every viable prefix is a prefix of some right Sentential form.

It never goes beyond the handle boundary. If you cross the handle, the parser would be in an invalid state.

The set of viable prefixes of a grammar is a regular language.

S L R (1)

S → Simple
L → Left to right Scanning of input
R → using reverse RMD
(1) → Lookahead symbol is 1

SLR(1) uses LR(0) DFA

Parsing table of SLR(1) is combination of two table i.e Action table & Go to table.

Conflicts → Shift-Reduce
                → Reduce-Reduce

"Multiple entries in a cell of a table is known as Conflict"

a) Shift-Shift Conflict is not possible because we make LR(0) DFA. A cell of a table can never contain S/S entry.

b) A cell may have more than one value like —

S/R , R/R , R/R/R/R....n , S/R/R/R/R....n

S/S Not possible

In the previous example —

I_2: $E \to T.$ → Reduction Produced by E
$T \to T. * F$ → Shift against '*'

exactly one Shift and one or more reduction in a cell make it "Inadequate State" where there is a possibility of Conflict.

No Shift & more than one reduction also make a state inadequate.

I_7: $A \to b.$
$B \to c.$
} Inadequate one more reductions in a state

Reduction means • has reached at the end of production no more shift possible.

I_2: $E \to T.$
$T \to T. * F$

follow(E) ∩ Shift Symbol of $I_2 \neq \phi$ then Conflict

I_2 has Shift against '*' and one reduction. If you find '*' in follow of E then there will be multiple entries in a cell of '*' in row of I_2.

2. Construct SLR(1) parsing table for given grammar

$S \to L=R \mid R$
$L \to *R \mid id$
$R \to L$

Number all the productions

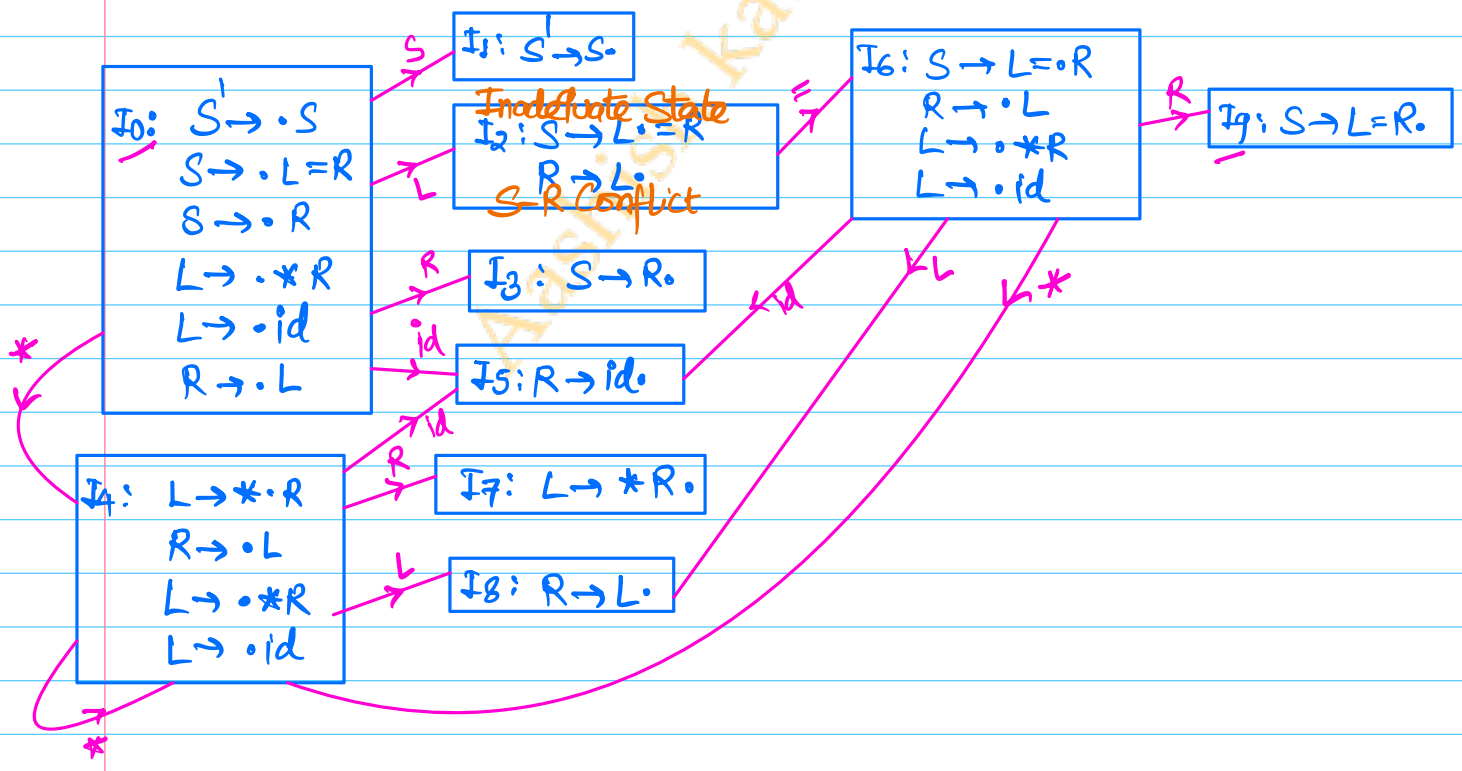
1. $S \to L=R$
2. $S \to R$
3. $L \to *R$
4. $L \to id$
5. $R \to L$

Augment the grammar

$S' \to S$

1. $S \to L=R$
2. $S \to R$
3. $L \to *R$
4. $L \to id$
5. $R \to L$

	First	Follow
S	*, id	$
L	*, id	=, $
R	*, id	$, =



	=	*	id	$	S	L	R
I_2	S6/r5			r5			

Multiple entries hence Conflict

Above grammar is not SLR(1)
not LR(0)
not LL(1)

3. Construct SLR(1) parsing table for given grammar

$S \to AaAb \mid BbBa$

$A \to \epsilon$

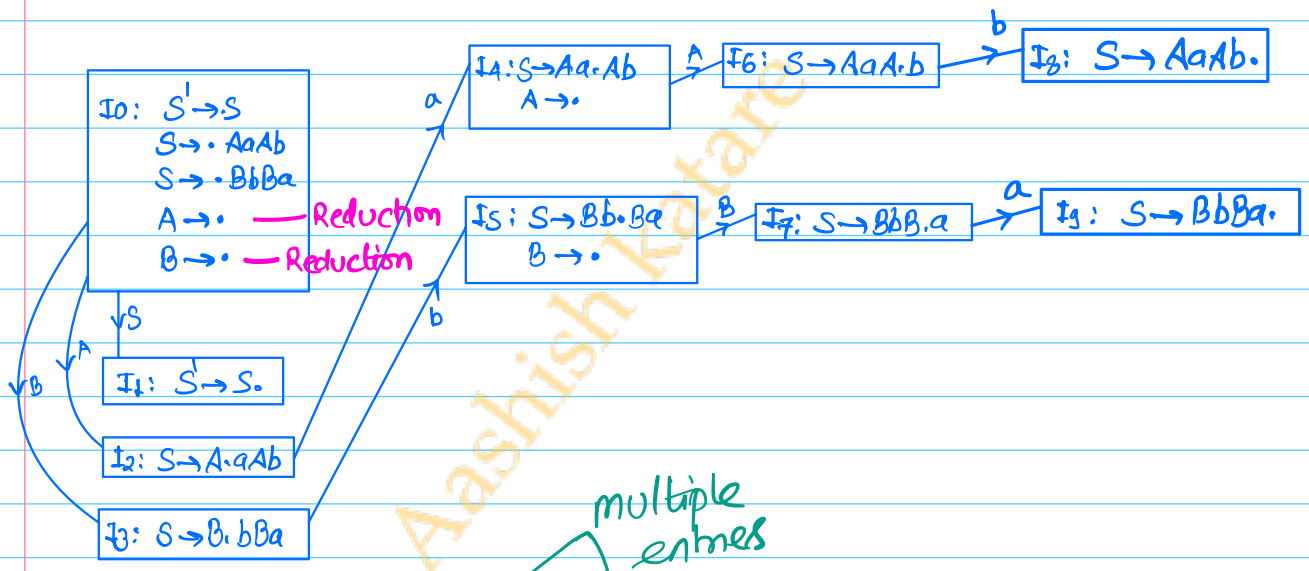
$B \to \epsilon$

1. $S \to AaAb$
2. $S \to BbBa$
3. $A \to \epsilon$
4. $B \to c$

$S' \to S$

1. $S \to AaAb$
2. $S \to BbBa$
3. $A \to \epsilon$
4. $B \to c$

	First	Follow
S	a, b	$
A	ϵ	a, b
B	ϵ	a, b



multiple entries

	a	b	$	S	A	B
I0	r3 / r4	r3 / r4		1	2	3
I4	r3	r3			6	
I5	r4	r4				7

Above grammar is — SLR(1) ✗
LR(0) ✗
LL(1) ✓

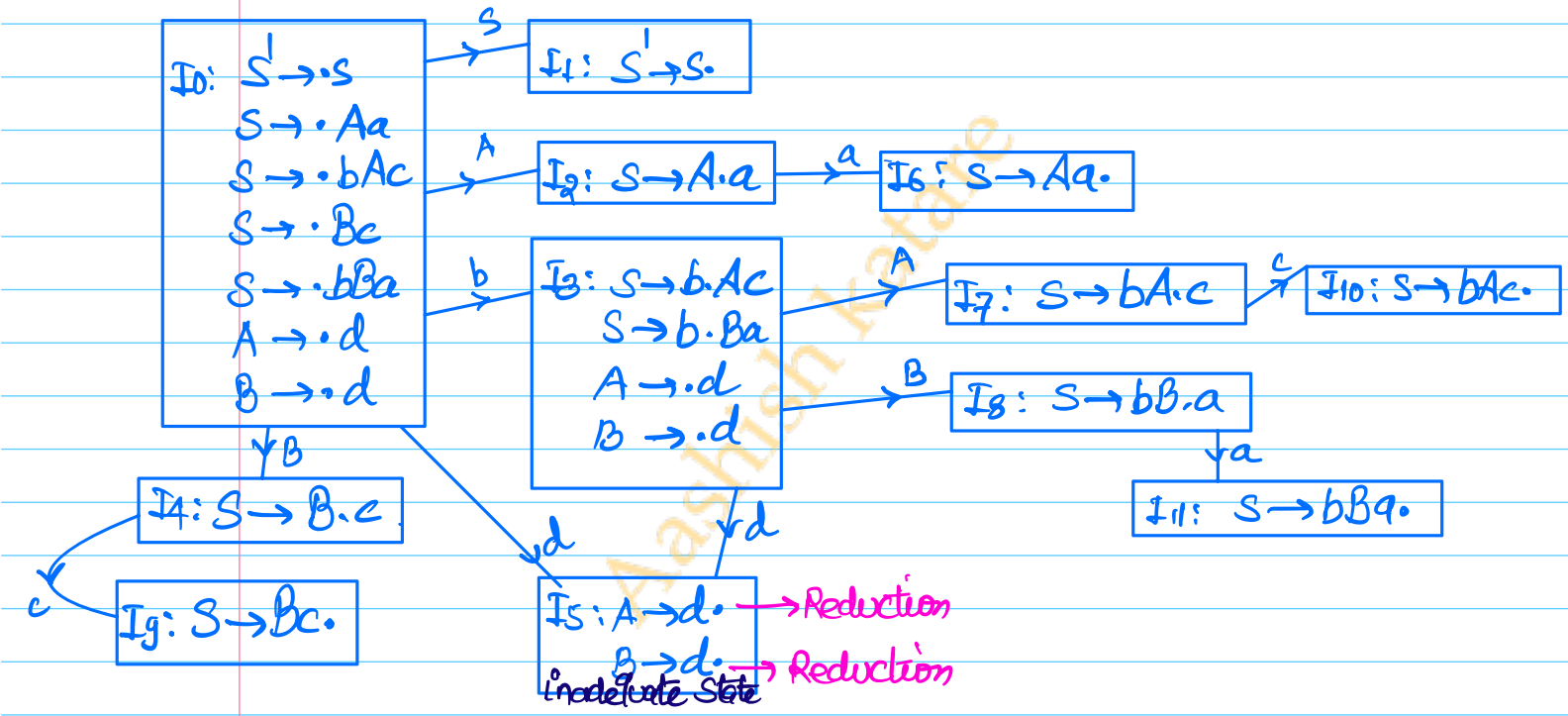
4. Construct SLR(1) parsing table for given grammar

$S \to Aa \mid bAc \mid Bc \mid bBa$
$A \to d$
$B \to d$

1. $S \to Aa$
2. $S \to bAc$
3. $S \to Bc$
4. $S \to bBa$
5. $A \to d$
6. $B \to d$

$\dot{S} \to S$
1. $S \to Aa$
2. $S \to bAc$
3. $S \to Bc$
4. $S \to bBa$
5. $A \to d$
6. $B \to d$

	First	follow
S	b, d	$
A	d	a, c
B	d	a, c



	a	b	c	d	$	S	A	B
I_5	r_5/r_6		r_5/r_6					

multiple entry "R-R conflict"

multiple entry "R-R conflict"

Above grammar is Not SLR(1), not LR(0) Not LL(1)

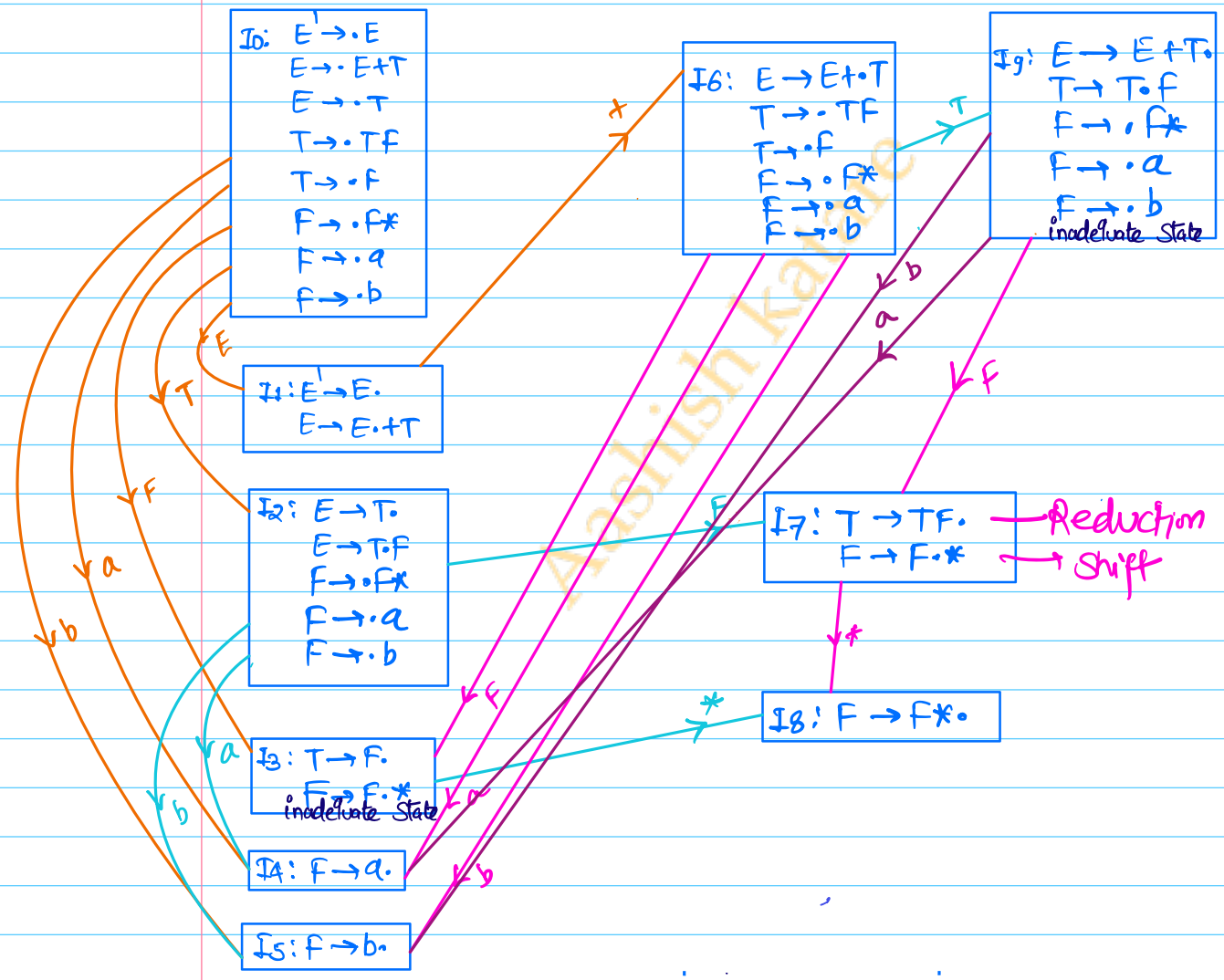
5. Construct SLR(1) parsing table for given grammar

$E \to E+T \mid T$
$T \to TF \mid F$
$F \to F* \mid a \mid b$

	First	follow
E	a, b	$, +
T	a, b	$, +, a, b
F	a, b	$, +, a, b, *

1. $E \to E+T$
2. $E \to T$
3. $T \to TF$
4. $T \to F$
5. $F \to F*$
6. $F \to a$
7. $F \to b$

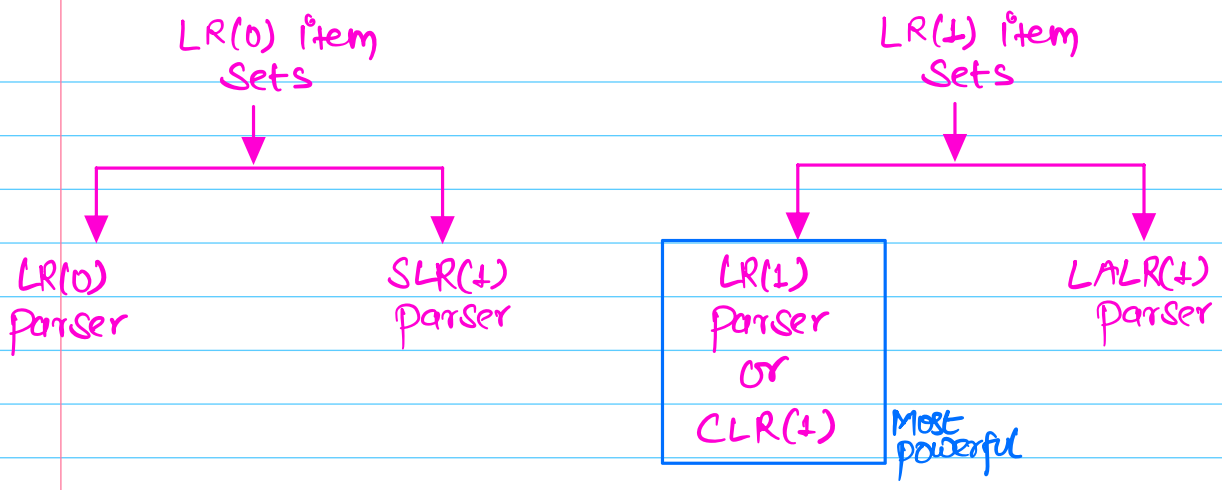
$E' \to E$
1. $E \to E+T$
2. $E \to T$
3. $T \to TF$
4. $T \to F$
5. $F \to F*$
6. $F \to a$
7. $F \to b$



	+	*	a	b	$	E	T	F
I_3	r4	S8	r4	r4	r4			
I_9	r1		S4	S5	r1			

Above grammar is SLR(1)
not LR(0), not LL(1)

LR(1) DFA –



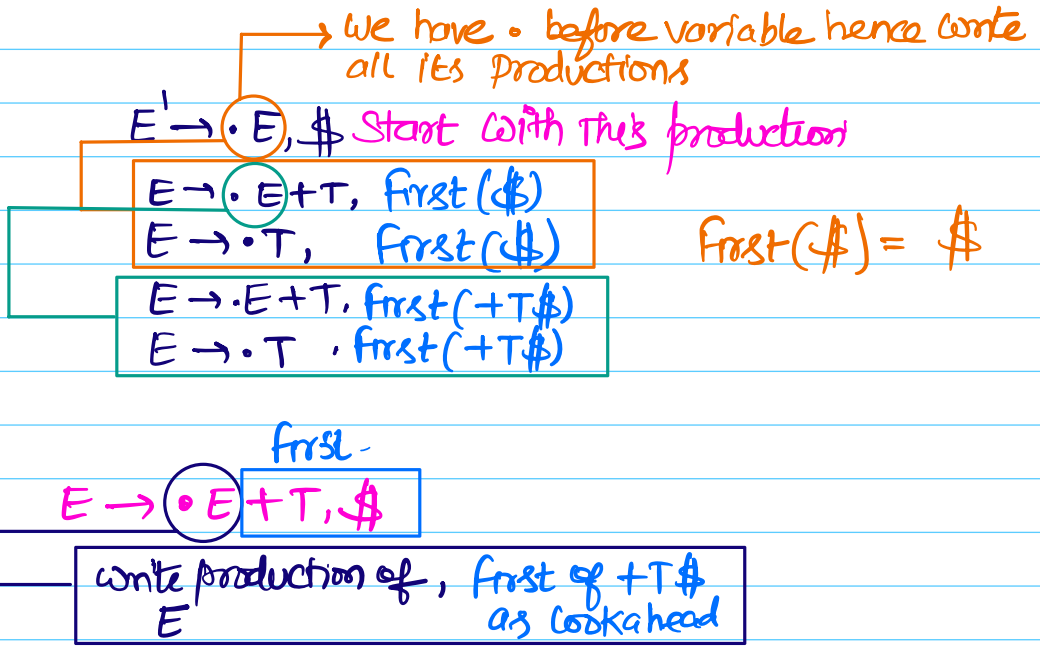
1. $S \to \cdot a$ LR(0) bcoz no look aheads are used as $S \to \cdot a, \$$ → look ahead (LR(1))
2. $S \to a\cdot$ LR(0)
3. $S \to a\cdot Ab$ LR(0)
4. $S \to \cdot aAb$ LR(0)
5. $S \to a\cdot A$ LR(0)
6. $S \to \cdot aA$ LR(0)
7. $S \to aA\cdot$ LR(0)
8. $S \to \cdot a, \$$ LR(1) → Look ahead
9. $S \to a\cdot, \$$ LR(1) } All are different
10. $S \to \cdot a, \#$ LR(1) }
11. $S \to a\cdot A, \$$ LR(1) } both are different
12. $S \to a\cdot A, \#$ LR(1) }
13. $S \to \cdot aA, \$$ LR(1)
14. $S \to \cdot a, \$/\#$ LR(1)
15. $S \to \cdot a, \$/ab/123$ LR(3) → $S \to \cdot a, \$$; $S \to \cdot a, \#$; $S \to \cdot a, 123$ OR $S \to \cdot a, \$/\#/123$
16. $S \to A\cdot b, \$/12/abc$ LR(3)
17. $S \to Ab\cdot, \$$ LR(1) } both are different
18. $S \to Ab\cdot, \#/\$$ LR(1) }
19. $S \to \cdot, \$$ LR(1)
20. $S \to \cdot, \#$ LR(1)

→ We can write it like – $S \to Ab\cdot, \$/\#$

How to calculate look aheads in CLR(1) –

Example –

$E \to E+T$
$E \to T$
$T \to TF$
$T \to F$
$F \to F*$
$F \to a$
$F \to b$



Conclusion –

1. $S \to \cdot \alpha \beta, \$ / \#$
   $\alpha \to \cdot x,$ First(β$) ∪ First(β#)
   $\alpha \to \cdot y,$ First(β$) ∪ First(β#)

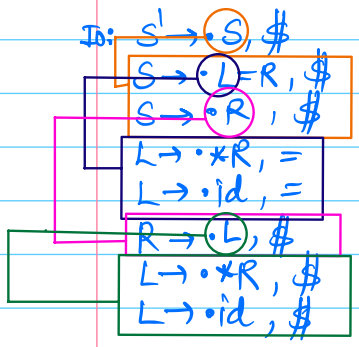
2. $S \to \cdot \alpha \beta CD, \$ / \#$
   $\alpha \to \cdot x,$ First(βCD$) ∪ First(βCD#)
   $\alpha \to \cdot y,$ First(βCD$) ∪ First(βCD#)

1. Construct CLR(1) parsing table for given grammar

$S \to L=R \mid R$
$L \to *R \mid id$
$R \to L$

$S' \to S$

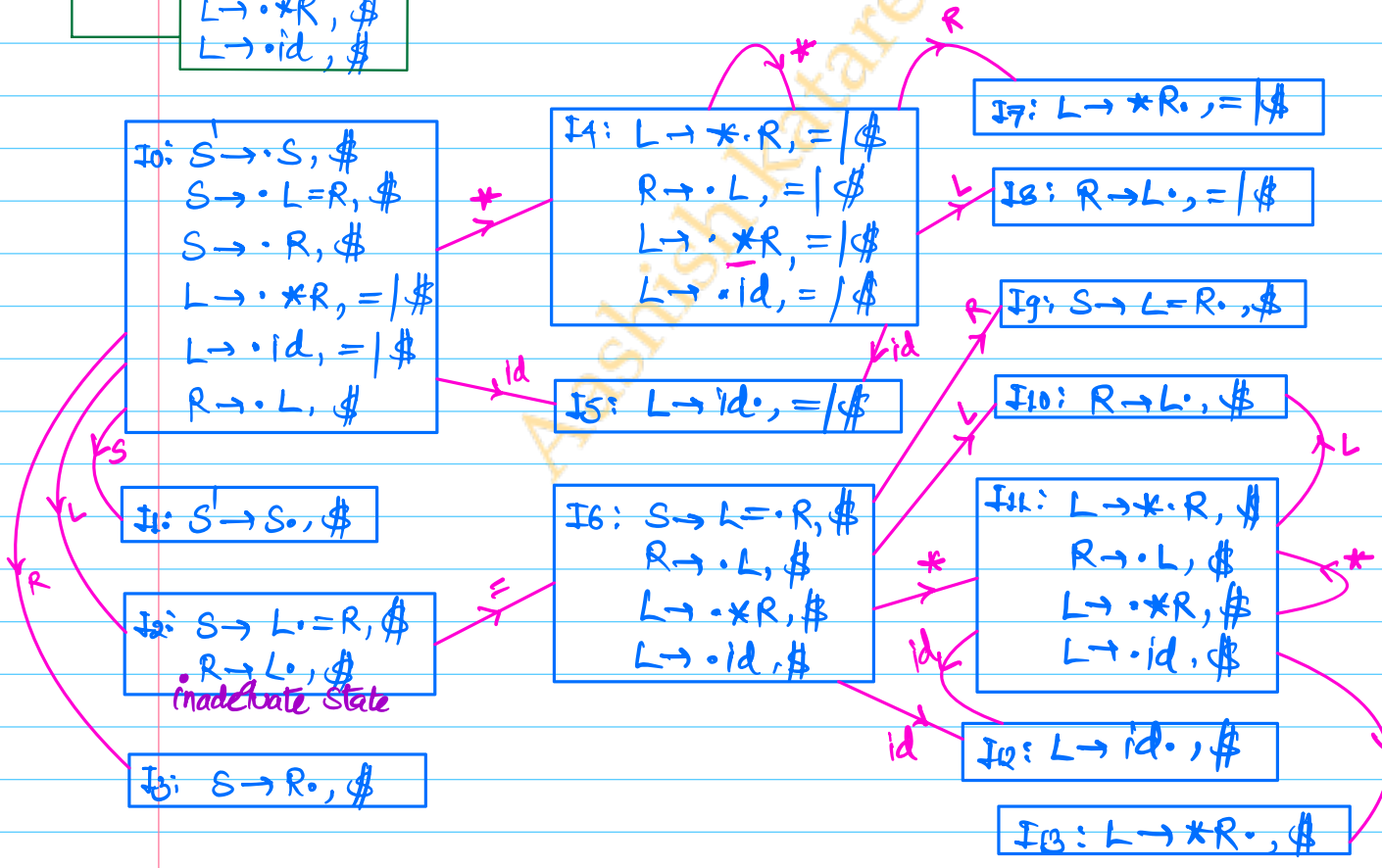
1. $S \to L=R$
2. $S \to R$
3. $L \to *R$
4. $L \to id$
5. $R \to L$



OR

I0: $S' \to \cdot S, \$$
$S \to \cdot L=R, \$$
$S \to \cdot R, \$$
$L \to \cdot *R, = / \$$
$L \to \cdot id, = / \$$
$R \to \cdot L, \$$

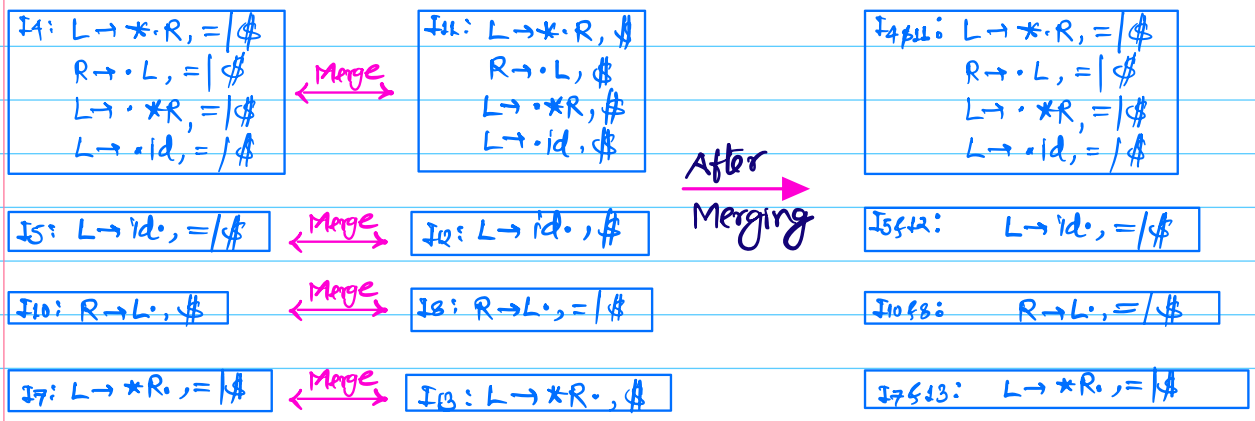
} minimised form



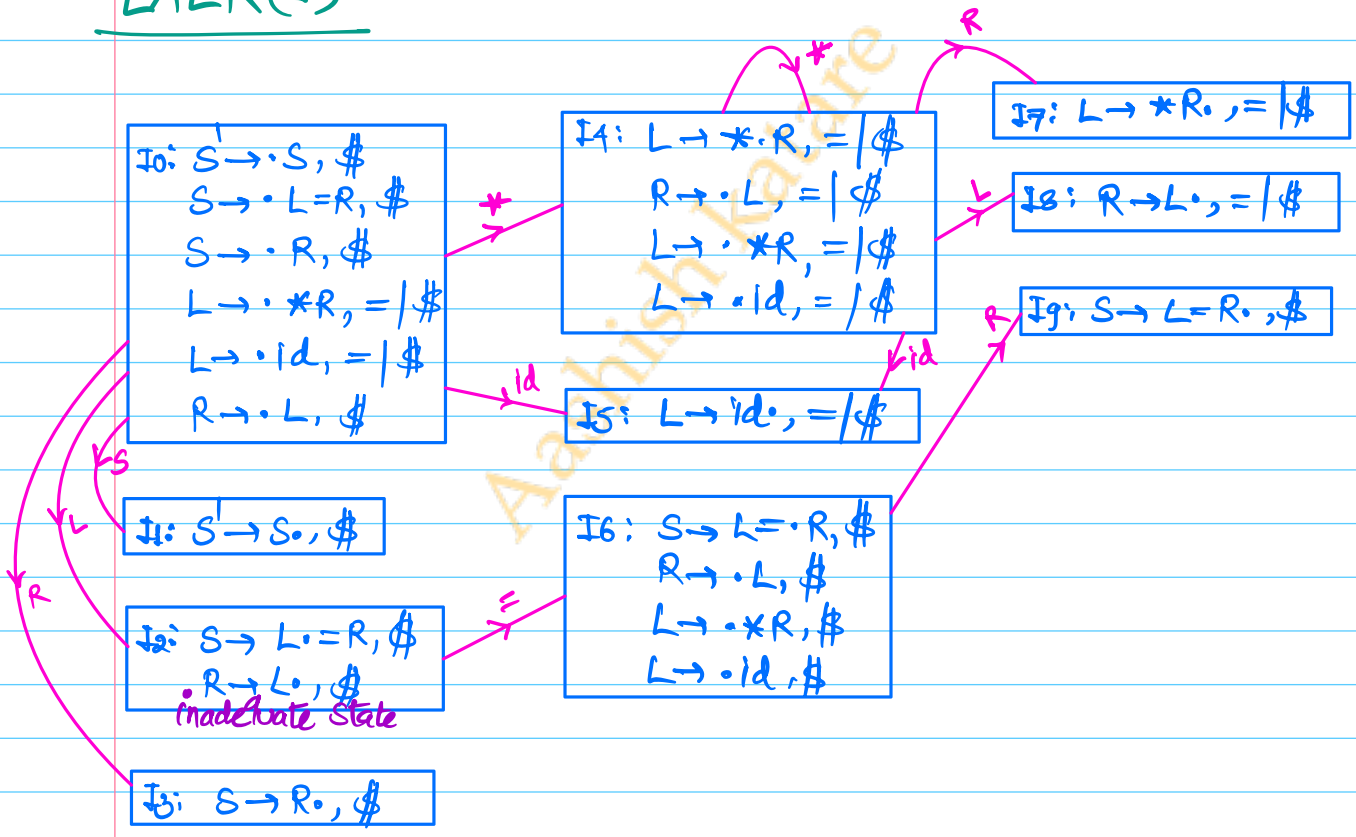
	*	=	id	$	S	L	R
I2		S6		r5			

Above grammar is CLR(1), No multiple values in any cell.

Look at these Canonical item sets. If you ignore lookahead symbols then I_4 and I_{11} are same, similarly $I_5 = I_{12}$, $I_{10} = I_8$ and $I_7 = I_{13}$. Merge them by taking union of lookaheads.



LALR(1) —



Important – After merging of states still there is no change in inadequate state, there was no conflict before and after merging.

"Above grammar is LALR(1) also"

Important — 1. The number of Canonical item sets i.e states in SLR(1), LALR(1), CRL(1) are n_1, n_2 & n_3 respectively for a given grammar G.

SLR(1) ⇓ n_1 LALR(1) ⇓ n_2 CLR(1) ⇓ n_3

$$n_1 = n_2 \leq n_3$$

Revise again —

2. a) $S \to \cdot\alpha$, \$
   $\alpha \to \cdot xy$, First(\$)
   $\alpha \to \cdot pq$, first(\$)

   b) $S \to \cdot \alpha BCD$, \$
   $\alpha \to \cdot x$, First(BCD\$)
   $\alpha \to \cdot y$, First(BCD\$)

   c) $S \to \cdot \alpha BCD$, \$/#
   $\alpha \to \cdot x$, First(BCD\$) ∪ First(BCD#)
   $\alpha \to \cdot y$, First(BCD\$) ∪ First(BCD#)

   d) $S \to \cdot \alpha BCD$, \$
   $\alpha \to \cdot x$, first(BCD\$) ∪ first(CD\$) } if $B \xrightarrow{*} \epsilon$
   $\alpha \to \cdot y$, first(BCD\$) ∪ first(CD\$) }

Similarly, if $C \to \epsilon$ or $D \to \epsilon$ you have do it accordingly.

3. $A \to \cdot\epsilon$ or $A \to \epsilon\cdot$ } $= A \to \cdot$

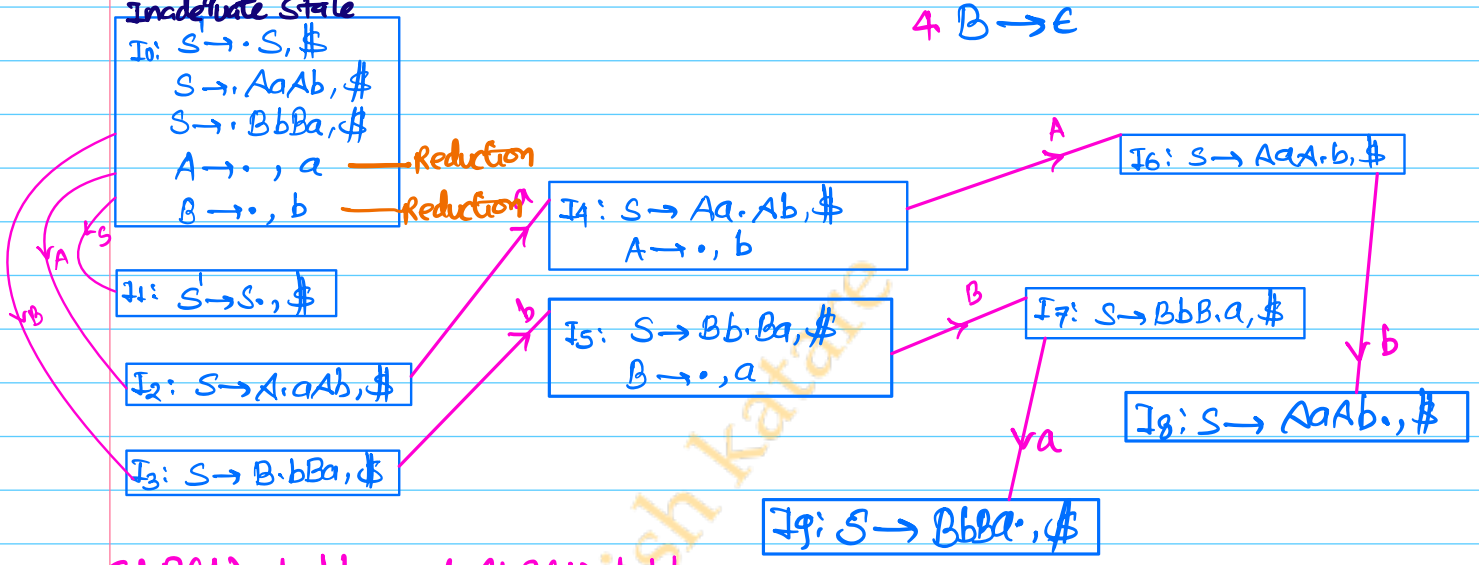
4. We prefer LALR(1) over CLR(1) because LALR(1) has parsing tables almost same size of SLR(1), yet they are much more powerful than SLR(1). CLR(1) generates huge parsing tables that are impractical while LALR(1) achieves almost the same parsing power with for fewer states making it efficient and sufficient for most programming languages.

2. Construct CLR(1) & LALR(1) parser for the given grammar -

$S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$

1. $S \to AaAb$
2. $S \to BbBa$
3. $A \to \epsilon$
4. $B \to \epsilon$

$S' \to S$
1. $S \to AaAb$
2. $S \to BbBa$
3. $A \to \epsilon$
4. $B \to \epsilon$



CLR(1) table, LALR(1) table

	a	b	$	S	A	B
I0	r_3	r_4		1	2	3
I4		r_3			6	
I5	r_4					7

Grammar is CLR(1) & LALR(1)

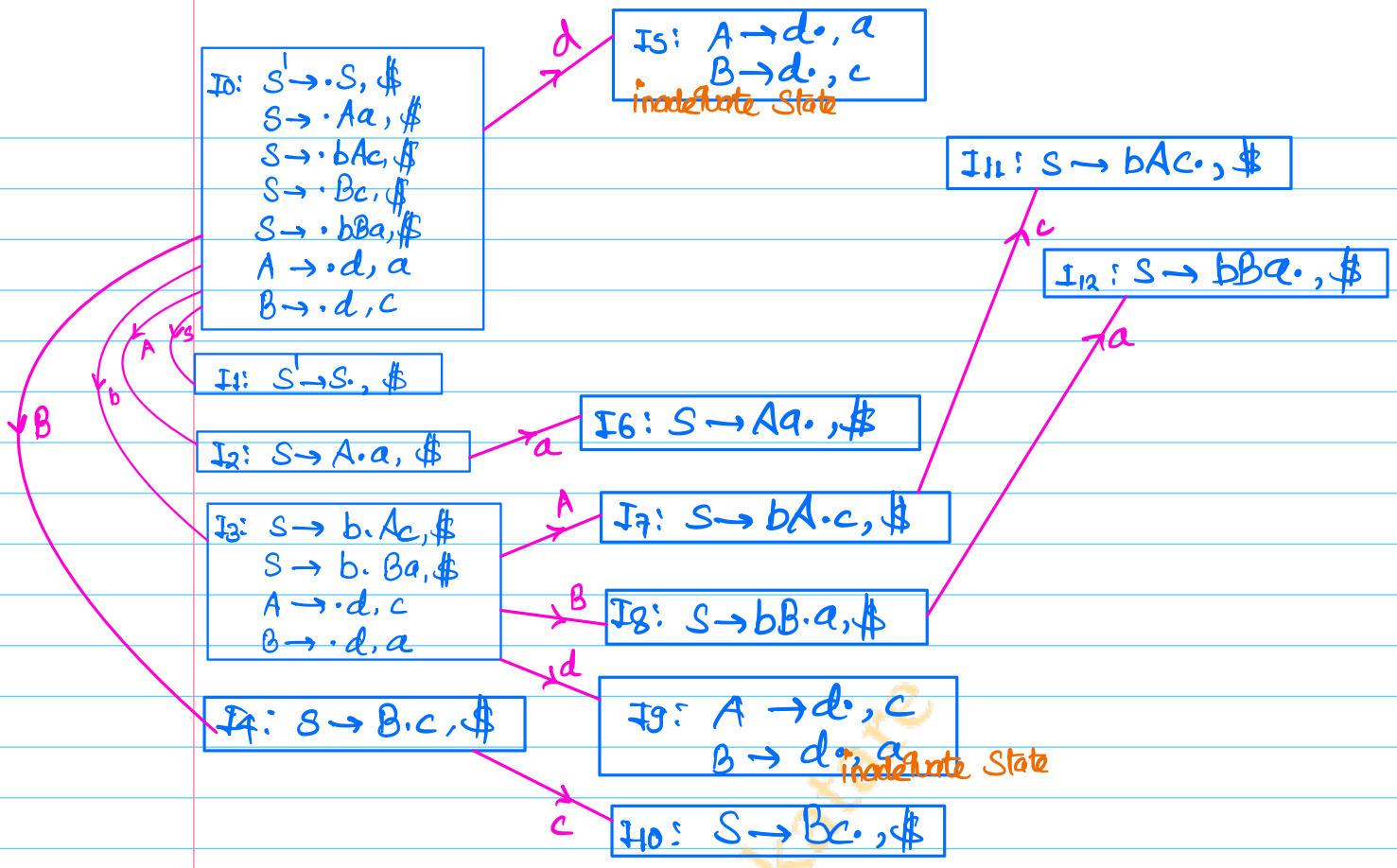
I0 is the only inadequate state but no conflict hence CLR(1)
No scope of merging. It is by default merged and LALR(1)

3. Construct CLR(1) & LALR(1) parser for the given grammar -

$S \to Aa \mid bAc \mid Bc \mid bBa$
$A \to d$
$B \to d$

1. $S \to Aa$
2. $S \to bAc$
3. $S \to Bc$
4. $S \to bBa$
5. $A \to d$
6. $B \to d$

$S' \to S$
1. $S \to Aa$
2. $S \to bAc$
3. $S \to Bc$
4. $S \to bBa$
5. $A \to d$
6. $B \to d$



CLR(1)

	a	b	c	d	$	S	A	B
I_5	r_5		r_6					
I_9	r_6		r_5					

We Can merge I_5 & I_9, $I_{5=9}$ A→d., a/c
B→d., a/c

LALR(1)

	a	b	c	d	$	S	A	B
$I_{5=9}$	r_5/r_6		r_6/r_5					

Multiple entries in a cell
R-R Conflict

Above grammar is CLR(1) but not LALR(1)

Note :-

1. 

Bottom up parser

2. If CLR(1) has no S-R & R-R Conflict then after merging only R-R Conflict Can occur.

3. If CLR(1) has no S-R Conflict then after merging S-R will never occur in LALR(1).

4. If CLR(1) itself has some Conflict then obviously it will reflect in LALR(1) also after merging.

4. (i) I_J: $S \to \cdot a\beta, \$/\#$ } No conflict
$T \to x\cdot, i/J$ }

I_K: $S \to \cdot a\beta, p/q$ } No conflict
$T \to x\cdot, d/b$ }

Can we merge I_J & I_K ? yes we Can.

$S \to \cdot a\beta, \$/\#/p/q$ } No conflict
$T \to x\cdot, i/J/d/b$ }

(ii) I_J: $S \to \cdot a\beta, \$/\#$ } S-R conflict
$T \to x\cdot, i/J/a$ }

I_K: $S \to \cdot a\beta, p/q$ } No conflict
$T \to x\cdot, d/b$ }

Can we merge I_J & I_K ? yes we Can.

$S \to \cdot a\beta, \$/\#/p/q$ } S-R conflict
$T \to x\cdot, i/J/d/b/a$ }

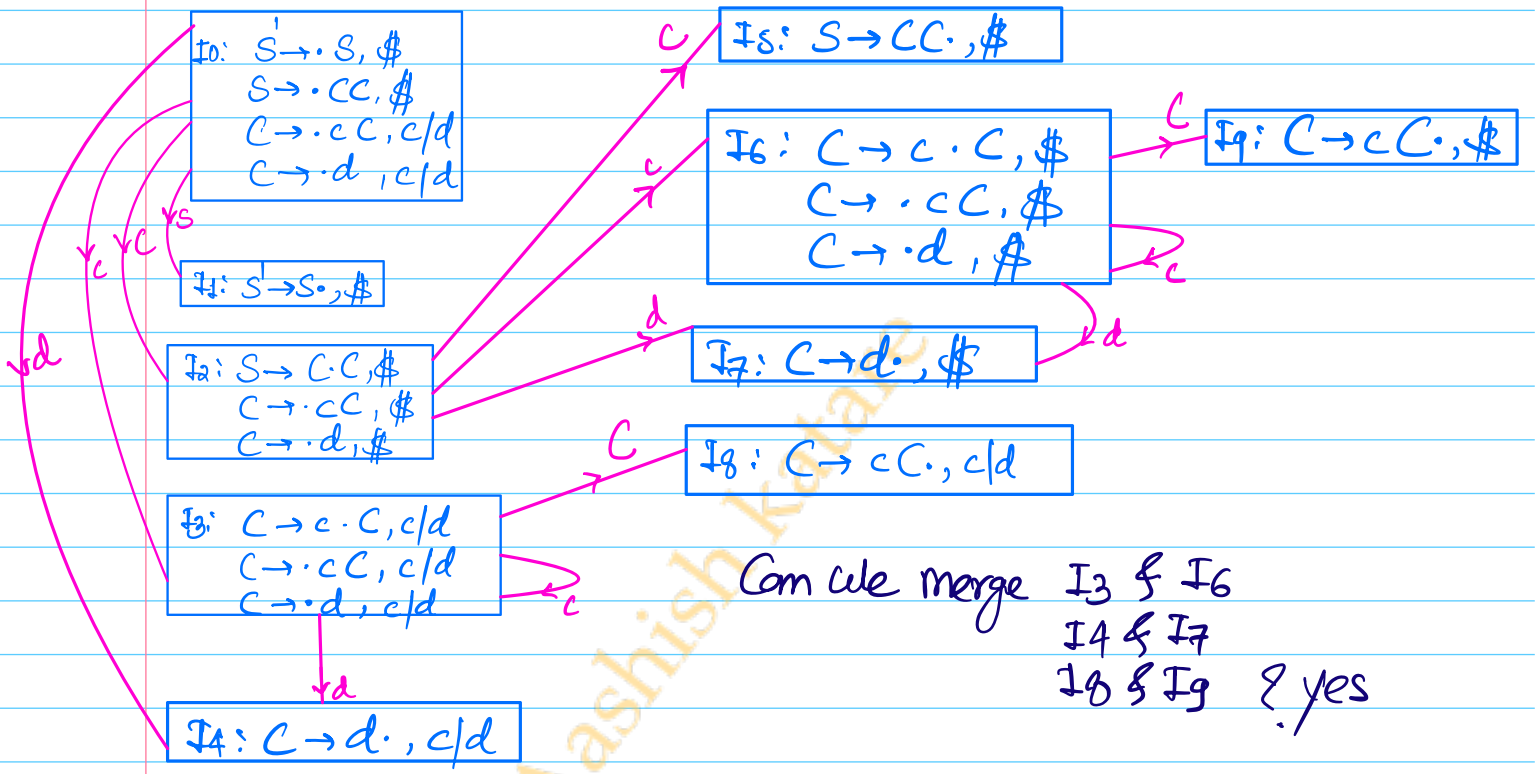
" S-R Conflict Can never occur after merging of two or more states". ✓

4. Construct CLR(1) & LALR(1) parser for the given grammar -

$S \to CC$
$C \to cC$
$C \to d$

1 $S \to CC$
2 $C \to cC$
3 $C \to d$

$S' \to S$
1 $S \to CC$
2 $C \to cC$
3 $C \to d$



Aashish Katare

Conclude merge I3 & I6
I4 & I7
I8 & I9 ? yes

I3: C→c·C, c/d
C→·cC, c/d
C→·d, c/d

I6: C→c·C, $
C→·cC, $
C→·d, $

= I3=6
C→c·C, c/d/$
C→·cC, c/d/$
C→·d, c/d/$

I4: C→d·, c/d

I7: C→d·, $

= I4=7
C→d·, c/d/$

I8: C→cC·, c/d

I9: C→cC·, $

= I8=9
C→cC·, c/d/$

In above grammar there is no inadequate state hence no conflict. It is CLR(1) as well as LALR(1)

Important point -

The difference between SLR(1) and CLR(1) lies in how the reduction entries are filled in the Action table.

a) If: $E \to T.$
$T \to T.*F$
then SLR(1) do reduction entries in follow(E).

What if follow(E) has '*'? then there will be S-R Conflict. Why? bcoz I_j will shift against '*'.

b) If: $E \to T., \$ | a | b$
$T \to T.*F, \$$
then CLR(1) says do reduction entries in lookahead symbols of reduced production i.e $\$, a, b$ not in the follow of E.

Correct way to do reduction entries in CLR(1) is -

follow(E) $\cap$ {look ahead symbols} = {x, y, z}

{x, y, z} = {look ahead Symbols}

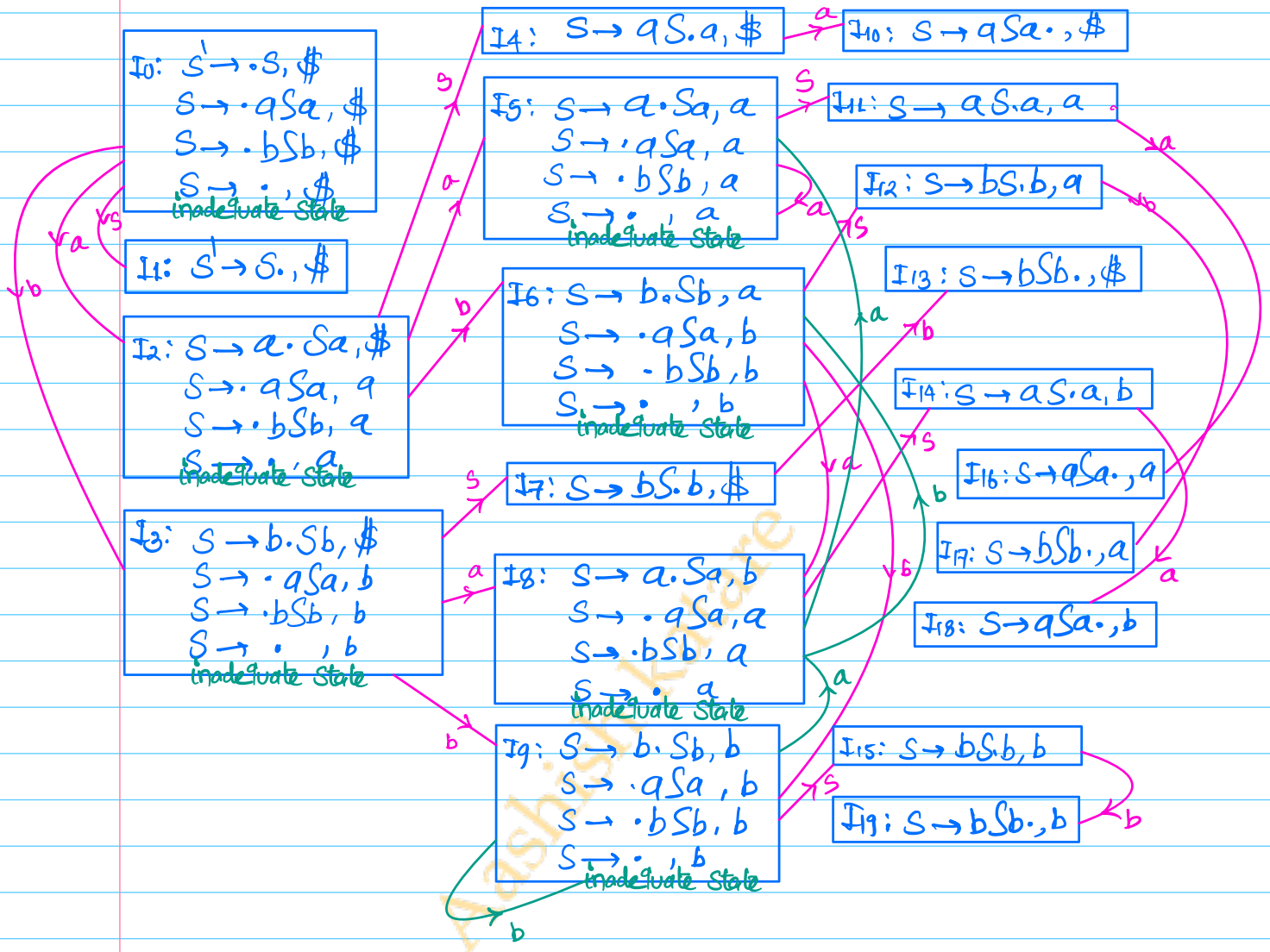
5. Construct CLR(1) & LALR(1) parser for the given grammar -

$S \to aSa \mid bSb \mid \epsilon$

1. $S \to aSa$
2. $S \to bSb$
3. $S \to \epsilon$

$S' \to S$
1. $S \to aSa$
2. $S \to bSb$
3. $S \to \epsilon$

Language of above grammar is not DCFL. It is WCFL hence No, deterministic parser is possible.



	a	b	$	S
I0				
I2				
I3				
I5				
I6	S8	r3/S9		
I8	r3/S5	S6		
I9	S8	S9/r3		

Fill the in-complete table

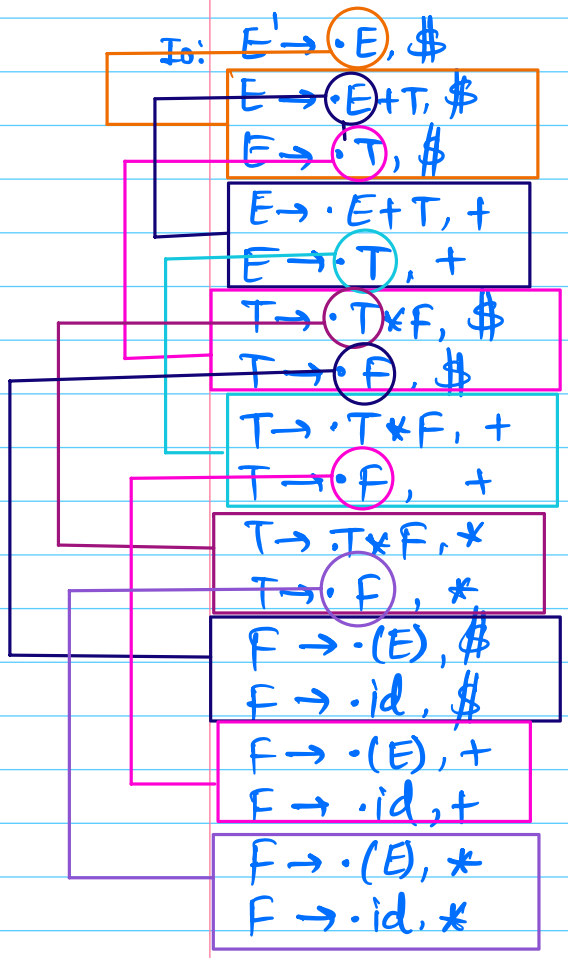
Above grammar is not even CLR(1)

6. Construct CLR(1) Parser for the given grammar

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

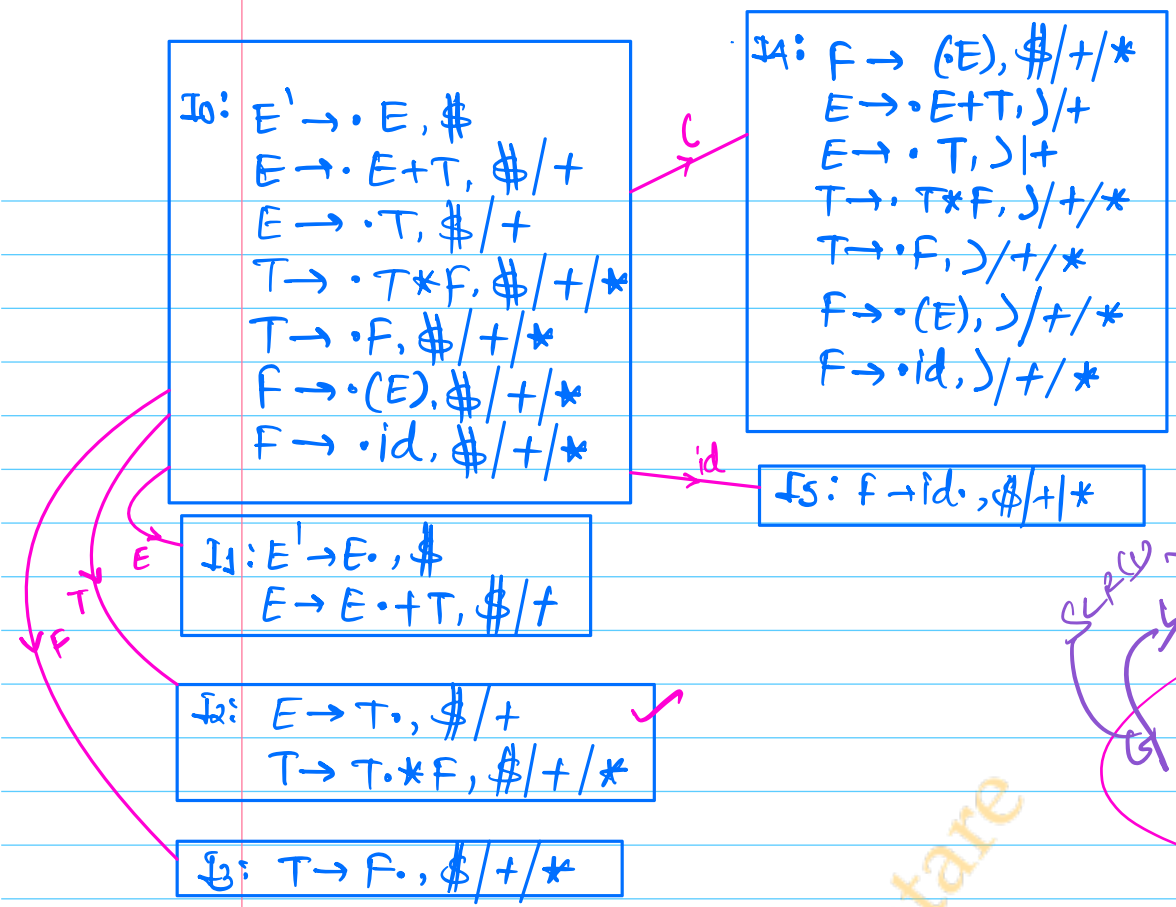
1 $E \to E+T$
2 $E \to T$
3 $T \to T*F$
4 $T \to F$
5 $F \to (E)$
6 $F \to id$

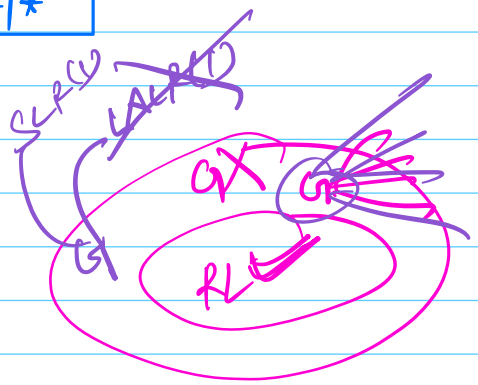
$E' \to E$
1 $E \to E+T$
2 $E \to T$
3 $T \to T*F$
4 $T \to F$
5 $F \to (E)$
6 $F \to id$



OR

$E' \to \cdot E, \$$
$E \to \cdot E+T, \$/+/$
$E \to \cdot T, \$/+/$
$T \to \cdot T*F, \$/+/*$
$T \to \cdot F, \$/+/*$
$F \to \cdot(E), \$/+/*$
$F \to \cdot id, \$/+/*$





Complete the question
"you have basic idea"

Construct CLR(1) parser for the given grammar

$E \to E+T \mid T$
$T \to TF \mid F$
$F \to F* \mid a \mid b$

Home work question
Make only I0

Important - All are deterministic parsers

1. a) L L C 1 S

L → Left to right Scanning of Input
L → using LMD
C 1 S → Lookahead Symbol is 1

b) S L R (1)

S → Simple
L → Left to right Scanning of Input
R → using Reverse RMD
(1) → Lookahead Symbol is 1

c) L A L R (1)

L → Look
A → Ahead
L → Left to right Scanning of Input
R → using Reverse RMD
(1) → Lookahead Symbol is 1

d) C L R (1)

C → Cannonical
L → Left to right Scanning of Input
R → using Reverse RMD
(1) → Lookahead Symbol is 1

e) L R (0)

L → Left to right Scanning of Input
R → using Reverse RMD
(0) → Lookahead Symbol is 0

2. Grammar is LL(1) or LR(0) or SLR(1) or CLR(1) or LALR(1) is different from the question whether given language is LL(1) or LR(0) or SLR(1) or CLR(1) or LALR(1).

3.
$$\begin{array}{l} E \to E+T \mid T \\ T \to T*F \mid F \\ F \to (E) \mid id \end{array} \quad \equiv \quad \begin{array}{l} E \to TE' \\ E' \to +TE' \mid \epsilon \\ T \to FT' \\ T' \to *FT' \mid \epsilon \\ F \to (E) \mid id \end{array}$$

G_1 $\equiv$ G_2

Language of the above grammars is same but G_1 is not LL(1) while G_2 is.

My very Innocent Question- Is every regular language LL(1)?

Can we write a grammar for every regular language in such a way that it becomes LL(1)?

or

for any regular language L, can we always find an LL(1) grammar that generates L?

Yes, every regular language is LL(1) in terms of existence of some LL(1) grammar for it. An arbitrary grammar for a regular language may not be LL(1). You may need to transform it.

4. Similarly every regular language is SLR(1), LALR(1) & LR(1) Not every arbitrary CFG for a regular language is SLR(1) OR LALR(1) OR CLR(1), but one always exist.

5. for any DCFL L, can we always find an LL(1) grammar that generates L? NO, not every DCFL has an equivalent LL(1) grammar.

$LL(1) \subseteq DCFL$ "$LL(k) \subset DCFL$"

6. For any DCFL L, Can we always find an SLR(1) grammar that generates L? NO, there exist DCFLs that are not SLR(1).

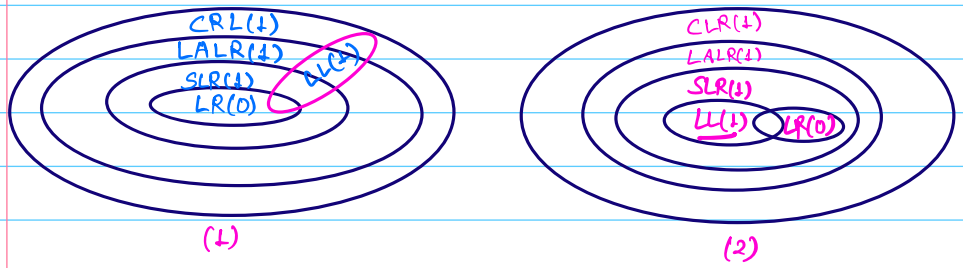
7. For any DCFL L, Can we always find an LALR(1) grammar that generates L? NO, there exist DCFLs that are not LALR(1).

8. For any DCFL L, Can we always find an CLR(1) grammar that generates L?
CLR(1). yes. Not every deterministic CFG is CLR(1)

LR(1) language = DCFL = LR(K)
CLR(1) is the maximal deterministic parsing class. NO DCFL lies outside it.

6. The language of the grammar for which any deterministic parser exist is always DCFL.
for the grammars whose language is not DCFL we can't construct deterministic parsers.
Deterministic parsers? LL(1), LR(0), SLR(1), LALR(1), CLR(1)
Non-deterministic parser? CYK algorithm.

Venn diagram –



In terms of relationship between LL(1) & LALR(1) both Venn diagrams are opposite/contradictory. Venn diagram no.1 is correct and it is based on

grammars.

Venn diagram no. 2 is correct for languages. There is no language in Universe for which LL(1) exist but LALR(1) doesn't.

7. $LL(1) \subset SLR(1) \subset LALR(1) \subset CLR(1) = DCFL$ (valid for languages)

8. $LL(1) \subset SLR(1)$ Not true if you consider grammars
   $LL(1) \subset LALR(1)$ Not true if you consider grammars

9. $LL(1) \subset LL(2) \subset LL(3) \subset \ldots\ldots LL(k) \subset LR(1) = LR(k)$, $k \geq 1$

10. Every DCFL has an equivalent LR(1) grammar. According to Knuth "If L is DCFL then there is LR(1) grammar for it. A language generated by LR(k) is DCFL.

$$LR(1) \equiv LR(k)$$

11. $LR(k) \geq LL(k)$ for any fixed 'k'.

Syntax Directed Translation –

Without going into details first learn through a question –

Q. Consider the grammar with the following translation rules and E as the start symbol.

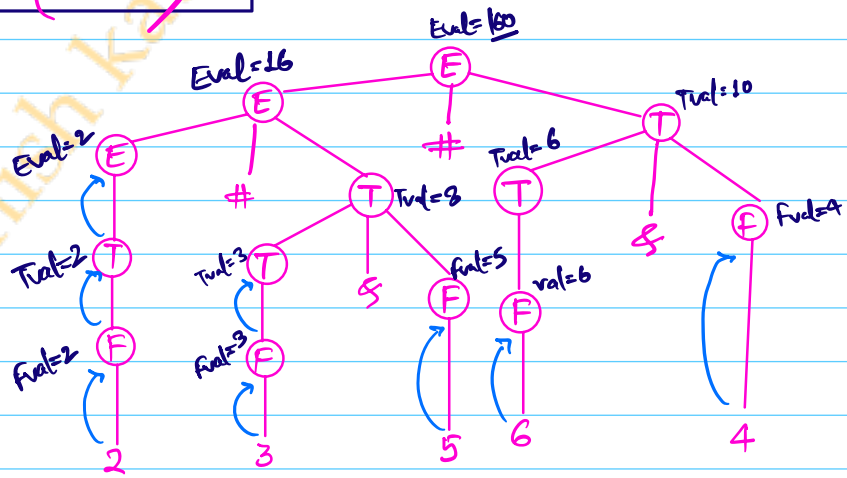
$E \to E \# T$ { E.value = E_1.value * T.value }
$E \to T$ { E.value = T.value }
$T \to T \& F$ { T.value = T_1.value + F.value }
$T \to F$ { T.value = F.value }
$F \to num$ { F.value = num.value }

Compute E.value for the root of the parse tree for the expression

$$2\#(3\&5)\#(6\&4)$$

RMD

$E \to E\#T$
$E \to E\#T\&F$
$E \to E\#T\&4$
$E \to E\#F\&4$
$E \to E\#6\&4$
$E \to E\#T\#6\&4$
$E \to E\#T\&F\#6\&4$
$E \to E\#T\&5\#6\&4$
$E \to E\#F\&5\#6\&4$
$E \to E\#3\&5\#6\&4$
$E \to T\#3\&5\#6\&4$
$E \to F\#3\&5\#6\&4$
$E \to 2\#3\&5\#6\&4$



When we write a program in a high-level language the Computer cannot directly understand it. So the Compiler translates it into machine code.

The translation process does not happen randomly it follows the syntax rules of the language. Along with these rules, we attach small instructions

Called Semantic rules that help in translation.

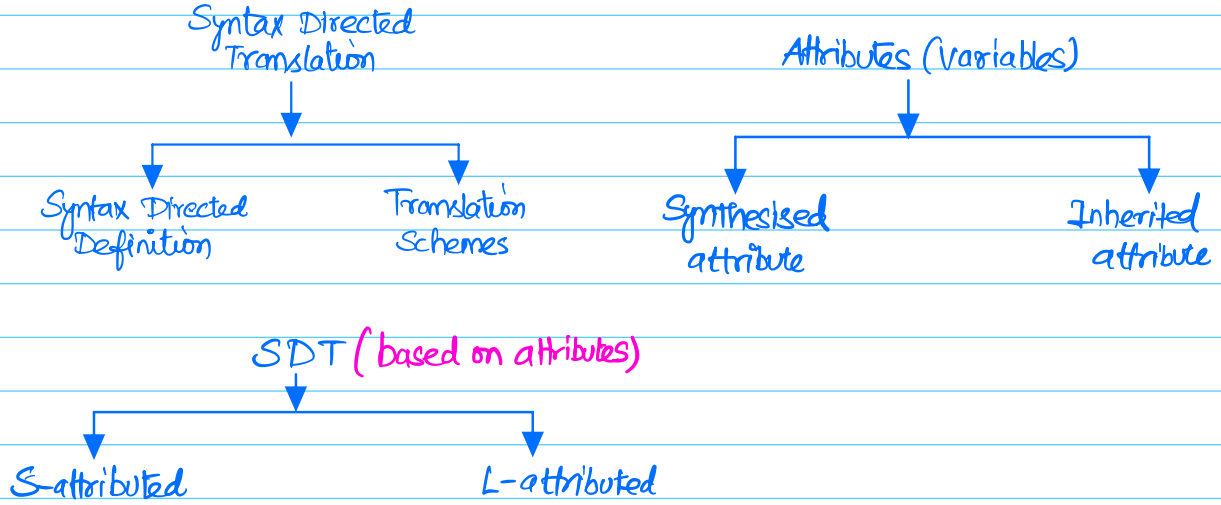
SDT = Grammar rules + Attached actions (Semantic rules)

"Syntax directed translation is a method where we attach small pieces of work to grammar rules so that while parsing, translation happens automatically.

"Grammar is syntax rules, it defines the structure of the language and Semantic rules tell what to do when that structure appears.

With Syntax directed translation, we do all the extra jobs beyond parsing-

(i) Evaluate values
(ii) Check types
(iii) Build Symbol tables
(iv) Generate intermediate code
(v) Handle flow control
(vi) Detect errors.



What is Syntax directed definition (SDD) & Translation Schemes →

Syntax Directed Translation – SDT means attaching semantic rules or actions to grammar productions.

SDT has two well known notations
(i) Syntax Directed definitions
(ii) Translation Schemes

Syntax directed definition – An SDD is a Context free grammar where each grammar symbol has attributes (Synthesised or inherited) and each production may have semantic rules that define how to compute these attributes.

↳ Semantic rules are written Separately from the grammar productions".

$E \rightarrow E \# T$ { E.value = E_1.value * T.value }
$E \rightarrow T$ { E.value = T.value }
$T \rightarrow T \& F$ { T.value = T_1.value + F.value }
$T \rightarrow F$ { T.value = f.value }
$F \rightarrow num$ { F.value = num.value }

Important –

It describes what attributes need to be Computed. It does not define the exact order of execution. Instead, evaluation order is determined using dependency graph and topological sorting.

In SDD we can use both Synthesised & inherited attributes.

" SDD can be S-attributed & L-attributed".

Translation Schemes –

A translation scheme is a CFG with semantic actions embedded directly in the RHS of productions. Actions are enclosed in { }.

example – 1) $S \rightarrow T\ R$
$R \rightarrow +\ T\ \{Print('+')\}\ R / \epsilon$
$T \rightarrow num\ \{Print(num.val)\}$

2) $S \rightarrow E\ R$
$R \rightarrow *\ E\ \{Print('*')\}\ R / \epsilon$
$E \rightarrow F + E\ \{Print('+')\} / F$
$F \rightarrow (S) / id\ \{Print(id.value)\}$

Important – It specifies both what and when to perform action. Translation schemes are easier for implementation. Translation scheme can use both synthesized & inherited attributes.

Difference between SDD & TS –

Syntax Directed definition	Translation Scheme
1. Rules are written separately	1. Rules are written inside productions.
2. Execution order is not fixed. It is determined by dependency graph	2. Execution order is explicitly given within production.

"Every SDD can be implemented as a translation scheme"

$S \rightarrow x\ y\ z\uparrow$ – SDD
$S \rightarrow \uparrow x \uparrow y \uparrow z \uparrow$ – Translation Scheme

Translation Scheme
SDD

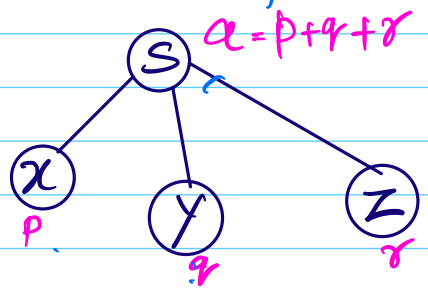
What is attribute?

An attribute is a value associated with a grammar symbol in a syntax directed Translation. These values are used to store semantic information about the program's construct

Types of attribute –

a) **Synthesised attribute** – An attribute is synthesised when computed from the attributes of the children in the parse tree.

ex-

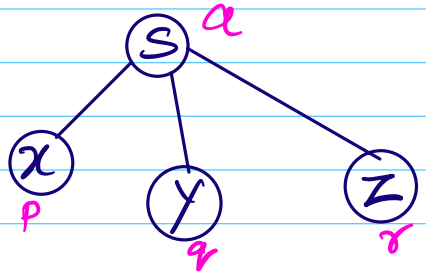


'a' attribute of S
'p' attribute of X
'q' attribute of Y
'r' attribute of Z

value of 'a' depends/computed from its children.

b) **Inherited attribute** → we use restricted/limited inheritance

An attribute is inherited when computed from the attributes of the parent or siblings to the left.



"An attribute can inherit value from its left sibling or from parent or from both."

'q' can't inherit value from Z
'p' can't inherit value from q, r
'r' can inherit value from p, q, a

Q

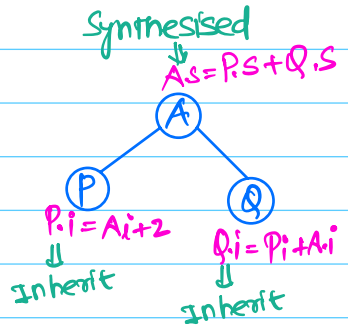
3.21 Consider the productions $A \rightarrow PQ$ and $A \rightarrow XY$. Each of the five non-terminals A, P, Q, X and Y has two attributes: s is a synthesized attribute, and i is an inherited attribute. Consider the following rules.

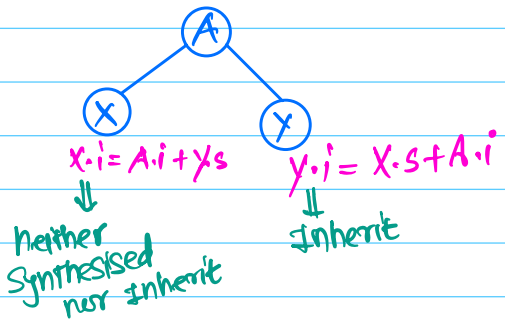
Rule 1: $P.i = A.i + 2$, $Q.i = P.i + A.i$ and $A.s = P.s + Q.s$

Rule 2: $X.i = A.i + Y.s$ and $Y.i = X.s + A.i$

Which one of the following is TRUE?

(a) Neither Rule 1 nor Rule 2 is L-attributed.
(b) Both Rule 1 and Rule 2 are L-attributed.
(c) Only Rule 1 is L-attributed.
(d) Only Rule 2 is L-attributed. **[2020 : 2 M]**





What is S-attributed & L-attributed SDT?

S-attributed →

An SDD is S-attributed iff every attribute is Synthesised means no Inherited attributes.

Because all attributes flow up the parse tree, you can evaluate in any bottom up or postorder traversal.

" Bottom up parsing Corresponds to postorder"

All attributes are Synthesised so a standard LR parser can compute them by executing the rule on each reduce.

L-attributed →

An SDD is L-attributed if every attribute is either
(i) Synthesised
(ii) Inherited

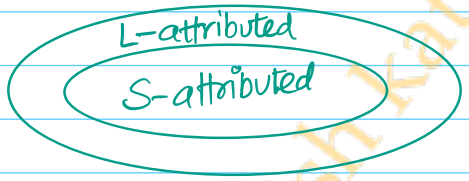
There is a natural left to right evaluation along the production body often realised as preorder.

" Top-Down parser is a great fit"

Top-Down parsers Can handle both S-attributed & L-attributed. Bottom up parser are best fit for S-attributed but Can also handle L-attributed SDD.

Why S-attributed is Subset of L-attributed?

In S-attributes SDD's all attributes are Synthesised. The L-attributed definition permits Synthesised attributes therefore, any S-attributed SDD trivially Satisfies the L-attributed Constraints, it has no inherited attributes to restrict, so every S-attributed SDD is L-attributed.



Q

3.4 In a bottom-up evaluation of a syntax directed definition, inherited attributes can
(a) always be evaluated
(b) be evaluated only if the definition is L-attributed
(c) be evaluated only if the definition has synthesized attributes
(d) never be evaluated
[2003 : 1 M]

In the question they have given, In bottom-up evaluation of syntax directed definition, inherited attributes Can —

" We have restricted inheritance".

a) Always be evaluated, No bcoz they have not mentioned above L-attributes or restricted inheritance

b) Correct. As we know bottom up parsers Can handle both S-attributed & L-attributed SDT. S-attributed is Subset of L-attributed. Bottom up is fit for S-attributed but Can also handle L-attributed

c) Wrong.

d) Wrong

Read Again –

S-attributed	L-attributed
1. All the attributes is of synthesised	1. It can have both synthesised and inherited but we have restricted inheritance.
2. Attributes are evaluated in bottom up evaluation bcoz attribute value depends on children.	2. Attributes are evaluated in depth first and left to right evaluation.
3. Semantic actions can be placed at the right most end of RHS of the production $A \rightarrow xyz$ { }	3. Semantic actions can be placed anywhere on the RHS of the production. $A \rightarrow \uparrow x \uparrow y \uparrow z \uparrow$ Note: every S-attributed L is attributed

Q. Are there Syntax Directed translations that can only be evaluated using bottom-up parsers? Can any SDT be parsed by any parser, or does it depend on the grammar and the placement of semantic rules?

Any SDT can be implemented with any parser, provided the grammar is parsable by that parser. Semantic rules are evaluated later, parser just build parse tree. Above explanation is valid only if we use two-pass approach i.e parser builds tree first, then semantics are evaluated.

"In one pass approach parser type and action placement are linked".

Imp " By default parsers use one pass approach".

Q Compiler can use only one parser then how L-attributed SDT will be handled if my compiler has CLR(1) only?

OR

If a compiler uses only one parsing strategy i.e CLR(1) then how can it still support L-attributed definitions, which are usually evaluated naturally in top-down parsing?

Parsing means recognizing the structure of the input it is done by grammar + parser.

Attribute evaluation = executing semantic rules attached to grammar

" parsing order (top down or bottom up) and attribute evaluation order are two separate but related issues".

L-attributed definitions allow inherited attributes This evaluation is most natural is Top-Down because when we expand a non-terminal, we can pass inherited attributes immediately to its children.

A bottom up parser builds the parse tree from leaves to root so inherited attributes look tricky because the parent is discovered later not earlier.

How bottom up parser will handle L-attributed SDT?

It will build parse tree first. Traverse the tree in a left to right manner to evaluate inherited attributes. This separates parsing from attributes evaluation, ensuring any L-attributed SDT can be handled.

Quick recap –

S-attributed SDD → only Synthesised attributes (Bottom UP)
L-attributed SDD → both Synthesised + Inherited attributes (Left to right top down)

Top-Down parsers –

Naturally support L-attributed definitions
We know that S-attributed is a subset of L-attributed they can also handle S-attributed easily.

Bottom-UP parsers –

Naturally support S-attributed definitions. They can handle L-attributed too.

" S-attributed or L-attributed doesn't matter it can be handled by both type of parsers i.e Top down & Bottom up".

Attribute Grammar – An attribute grammar is a formal way to define the semantics (meaning) of a programming language by attaching attributes to the symbols of a grammar and specifying rules for computing those attributes.

" Context-free-grammar + attributes + rules to compute them, with the restriction that semantic rules are pure functions (NO side effects)".

An attribute grammar has three main parts
(i) Grammar
(ii) Attributes – → Synthesised
→ Inherited
(iii) Semantic rules – functions to compute attribute values.

What is no side effects?

(i) Attribute grammar rules are pure functions. They only compute attribute values.

(ii) They cannot print, update global state or do I/O

(iii) This makes them predictable, analyzable and useful for formal compiler theory.

Can we have SDD with side effects?

Yes, we can have SDD with side effect
ex- $F \rightarrow num$ { print(num.val) }.
Above example does not compute an attribute. It produces output i.e side effect.

"Above example is not attribute grammar".

Important - Must read

1. Bottom up parsing & L-attributed grammars -

a) L-attributed grammars use both synthesised attributes & inherited attributes.

b) One pass bottom-up parsing can always handle S-attributed grammars.

c) Not all L-attributed grammars can be evaluated this way, because inherited attributes may not be available at the right moment during reduction.

d) Two pass bottom up parsing, first build the parse tree, then evaluate attributes in second traversal. This can evaluate any L-attributed grammar.

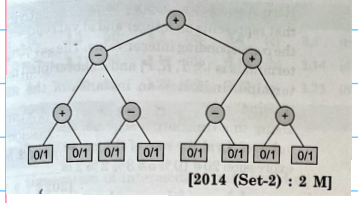
"In exams like GATE, assume one-pass parsing by default"

2. Top-Down parsing & L-attributed - This is perfect fit for L-attributed SDT since inherited attributes flow naturally from parent to child and left to right. Since S-attributed are subset of L-attributed hence they can be evaluated by top down parsers without any issue.

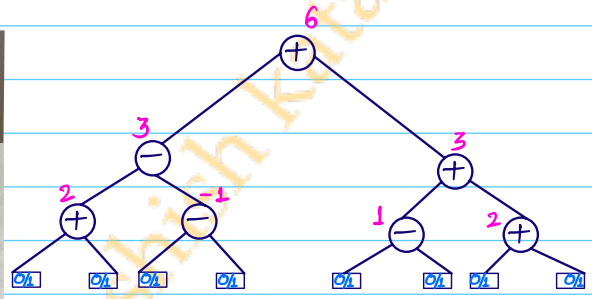
"S-attributed SDT can always be evaluated in single pass using top-down parsing".

Q

3.14 Consider the expression tree shown. Each leaf represents a numerical value, which can either be 0 or 1. Over all possible choices of the values at the leaves, the maximum possible value of the expression represented by the tree is ___6___.



[2014 (Set-2) : 2 M]



Q

3.5 Consider the translation scheme shown below:
S → T R
R → + T {print ('+');} R | ε
T → num {print (num.val);}
Here num is a token that represents an integer and num.val represents the corresponding integer value. For an input string '9 + 5 + 2', this translation scheme will print
(a) 9 + 5 + 2 (b) 9 5 + 2 +
(c) 9 5 2 + + (d) + + 9 5 2

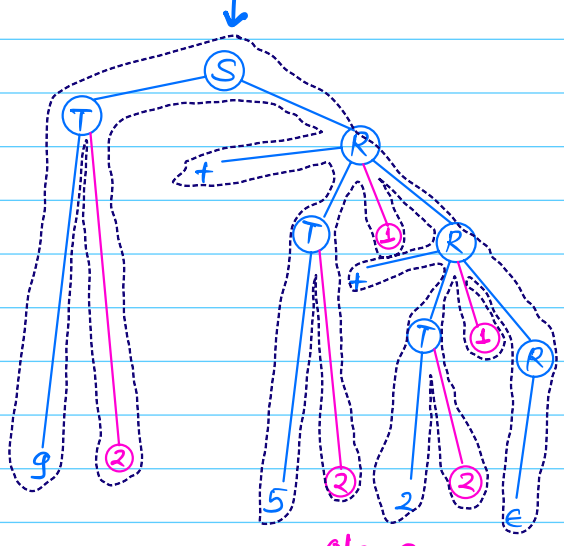
[2003 : 2 M]

S → TR
S → T+TR
S → T+T+TR
S → T+T+T+ε
S → T+T+2
S → T+5+2
S → 9+5+2

S → TR
R → +T {Print('+')} R/ε
T → num {Print(num.val)}

S → TR
R → +T1R/ε
T → num 2

"Left to right in depth"



O/P → 95+2+

Q

3.12 Consider the following translation scheme:

$S \rightarrow ER$
$R \rightarrow *E$ {print('*'); $R \mid \varepsilon$
$E \rightarrow F + E$ {print ('+'); | F
$F \rightarrow (S) \mid id$ {print (id. value);}

Here id is a token that represents an integer and id.value represents the corresponding integer value. For an input '2 * 3 + 4', this translation scheme prints

(a) 2 * 3 + 4 (b) 2 * + 3 4
(c) 2 3 * 4 + (d) 2 3 4 + *

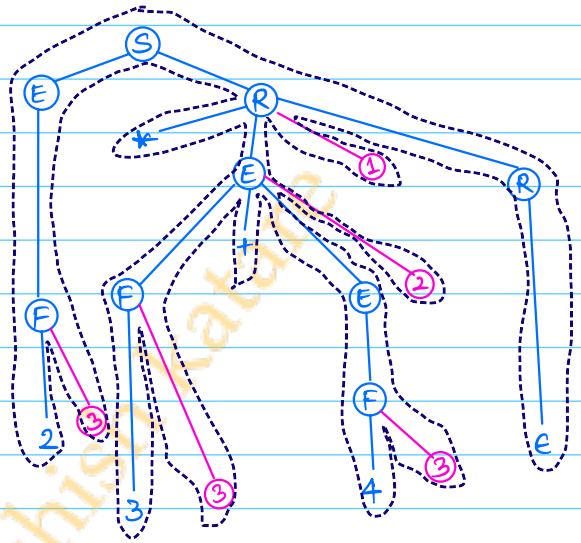
[2006 : 2 M]

$S \rightarrow ER$
$R \rightarrow *E \{Print(1)('*')\} R / \epsilon$
$E \rightarrow F+E \{Print(2)('+')\} / F$
$F \rightarrow (S) / id \{Print(3)(id.value)\}$

OR

$S \rightarrow ER$
$R \rightarrow *E1R \mid \epsilon$
$E \rightarrow F+E2 \mid F$
$F \rightarrow (S) \mid id3$

$S \rightarrow ER$
$S \rightarrow E*ER$
$S \rightarrow E*E\epsilon$
$S \rightarrow E*F+E$
$S \rightarrow E*F+F$
$S \rightarrow E*F+4$
$S \rightarrow E*3+4$
$S \rightarrow F*3+4$
$S \rightarrow 2*3+4$



234 + *

Q

3.17 Consider the following Syntax Directed Translation Scheme (SDTS), with non-terminals {*S*, *A*} and terminals {*a*, *b*}.

$S \rightarrow aA$ {print 1}
$S \rightarrow a$ {print 2}
$A \rightarrow Sb$ {print 3}

Using the above SDTS, the output printed by a bottom-up parser, for the input *aab* is:

(a) 1 3 2 (b) 2 2 3
(c) 2 3 1 (d) Syntax error

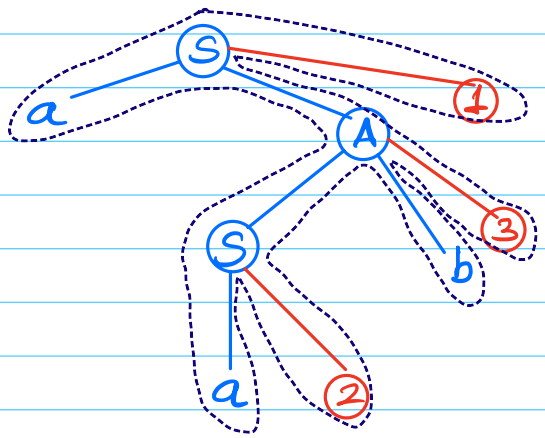
[2016 (Set-1) : 2 M]

$S \rightarrow aA \{Print\ 1\}$
$S \rightarrow a \{Print\ 2\}$
$A \rightarrow Sb \{Print\ 3\}$

OR

$S \rightarrow aA1$
$S \rightarrow a2$
$S \rightarrow Sb3$

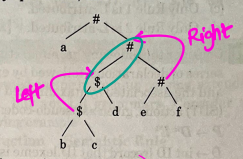
$S \rightarrow aA$
$S \rightarrow aSb$
$S \rightarrow aab$



O/P → 231

Q

3.19 Consider the following parse tree for the expression a # b $ c $ d # e # f, involving two binary operators $ and #.



Which one of the following is correct for the given parse tree?

(a) $ has higher precedence and is left associative; # is right associative

(b) # has higher precedence and is left associative; $ is right associative

(c) $ has higher precedence and is left associative; # is left associative

(d) # has higher precedence and is right associative; $ is left associative

[2018 : 2 M]

Q

3.6 Consider the syntax directed definition shown below:

$S \rightarrow id := E$ {gen (id.place = E.place;);}

$E \rightarrow E_1 + E_2$ {t = newtemp ( );
gen (t = E_1. place + E_2. place;);
E.place = t}

$E \rightarrow$ id {E.place = id.place;}

Here, gen is a function that generates the output code, and newtemp is a function that returns the name of a new temporary variable on every call. Assume that t_i's are the temporary variable names generated by newtemp.

For the statement 'X: = Y + Z', the 3-address code sequence generated by this definition is

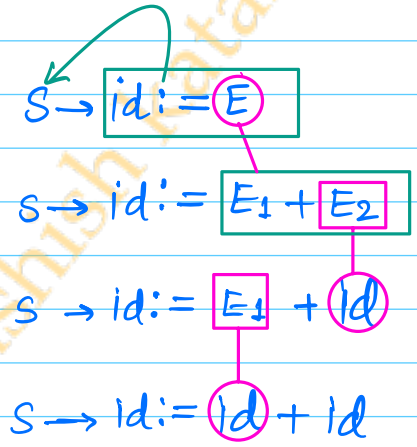
(a) X = Y + Z

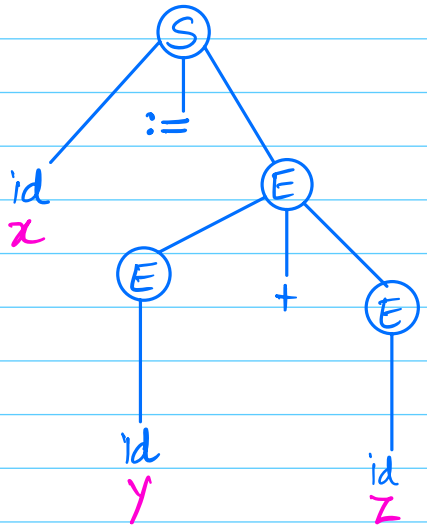
(b) t_1 = Y + Z; X = t_1

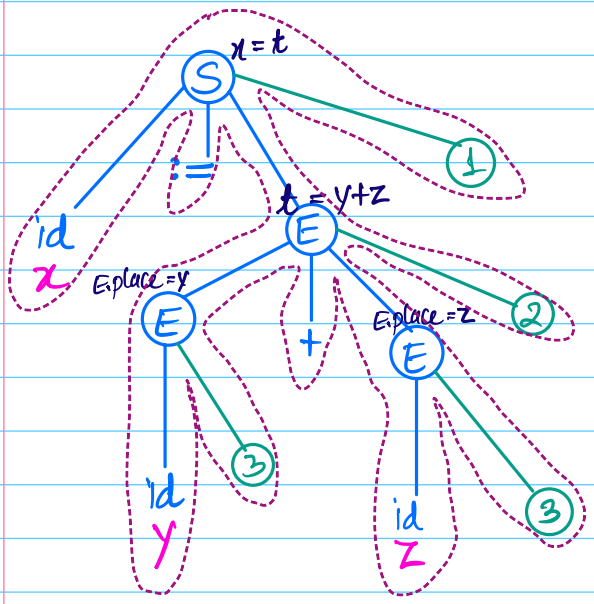
(c) t_1 = Y; t_2 = t_1 + Z; X = t_2

(d) t_1 = Y; t_2 = Z; t_3 = t_1 + t_2; X = t_3

[2003 : 2 M]







3.23 Consider the following grammar along with translation rules.

$S \rightarrow S_1 \# T$	{S.val = S_1.val * T.val}
$S \rightarrow T$	{S.val = T.val}
$T \rightarrow T_1 \% R$	{T.val = T_1.val ÷ R.val}
$T \rightarrow R$	{T.val = R.val}
$R \rightarrow$ id	{R.val = id.val}

Here # and % are operators and id is a token that represents an integer and id.val represents the corresponding integer value. The set of non-terminals is {S, T, R, P} and a subscripted non-terminal indicates an instance of the non-terminal.
Using this translation scheme, the computed value of S.val for root of the parse tree for the expression 20#10%5#8%2%2 is 80.
[2022 : 2 M]

% has higher precedence over #
% is left associative
# is left associative

% denotes division 8%2 means 8 is divided by 2.

# denotes multiplication 20#10 means 20 * 10

Note :- You can solve it by making tree also.

20 # (10 % 5) # ((8 % 2) % 2)

20 # 2 # 2
20 * 2 * 2
80

Q Which of the following statements is/are false?

a) The attributes in L-attributed definition cannot always be evaluated in a depth first order.

b) An attribute grammar is a Syntax-directed definition (SDD) in which the functions in the Semantic rules have no side effects

c) All L-attributed definitions based on LR(1) grammar can be evaluated using a bottom up parsing strategy.

d) Synthesised attributes can be evaluated by a bottom up parser as the input is parsed.

Doubts –

check→ In SDD rules are written at then 'end' & in translation schemes rules are written 'anywhere'

In SDD, rules are specified per production often written after it, but execution order is determined by attribute dependencies not the textual placement. In translation schemes, the position of each action in the production controls when it runs during parsing.

Dependency graph –

In Syntax Directed definition or translation schemes, attributes may depend on other attributes.

" The dependency graph shows these relationship"

Dependency graph is used to figure out a valid evaluation order for attributes. Attribute evaluation means topological sort of the dependency graph. If the graph has a cycle, the SDD is not well-defined and cannot be evaluated.

Do we always need a dependency graph?

In general SDDs, yes we may need it. The dependency graph tells us exactly when and in what order to evaluate attributes.

" In special classes like S-attributed & L-attributed, we don't need to build the graph explicitly."

The structure of S-attributed & L-attributed guarantees that simple traversals like postorder for S-attributed, left to right depth first for L-attributed already give a correct order.

Here in S-attributed & L-attributed dependency graph is implicit. We know evaluation order directly from the grammar's properties.

Conclusion -

SDTs can be evaluated without explicitly constructing a dependency graph but only if the SDT is S-attributed or L-attributed.

Intermediate Code -

Intermediate Code is an internal representation of a program generated by Compiler after the source Code is parsed and before the final machine Code is produced.

" Bridge between high level source code & low level assembly/machine code".

Why intermediate code is used ?

1) Portability - Compiler translates Source to intermed -iate code and then intermediate Code to target machine.
ex - Java bytecodes

2) Optimization - Optimisations are easier to perform on Intermediate code than on raw source or machine code.

Some Cons of IR (Intermediate Representation) -

" It needs extra Compilation step."
" Consumes memory because IR must be stored and processed".
" Not directly executable".
" Complexity of design".

Conclusion - Intermediate Code is used because it provides portability and optimization opportunities. Its main strength is simplifying Compiler design by acting as a bridge, but the trade off is extra steps and Complexity.

Revisit IR again —

Intermediate Code —

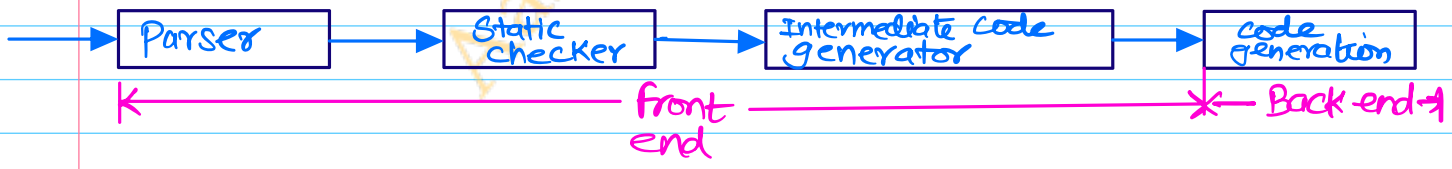
Intermediate Code is closer to the target machine than the source language, and hence easier to generate target code from. Unlike machine language, Intermediate Code is more or less machine independent. This makes it easier to retarget the Compiler.

" It allows a lot of optimizations to be performed in a machine independent way ".

Generally, Intermediate Code generation can be implemented via Syntax directed translation.

In the analysis - Synthesis model of a Compiler, the front-end analyses the source code and creates an intermediate representation, from which the backend generate target code.

" Intermediate representation does not depend on the target machine ".



" The choice on design of an Intermediate representation varies from compiler to compiler ".

There are two categories of Intermediate representation

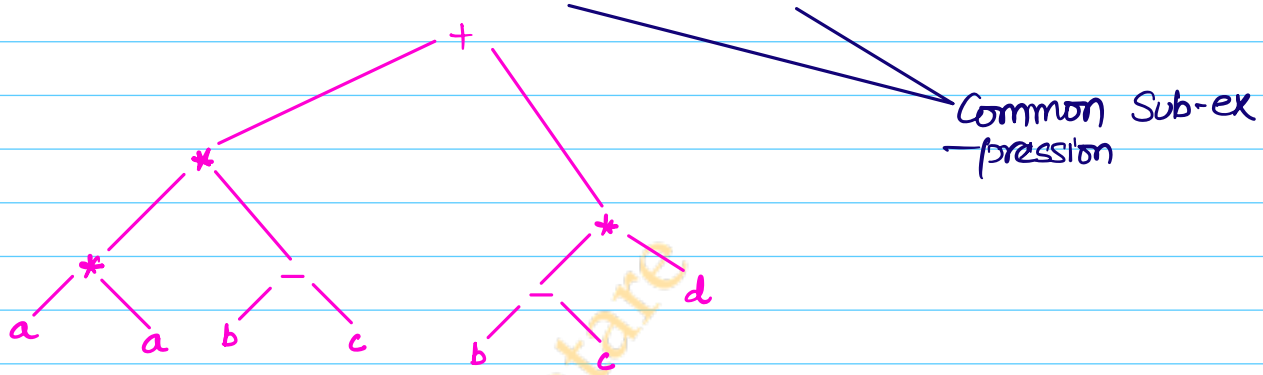
(i) Linear representation → Three address code, Postfix

(ii) Tree representation → Syntax tree, Directed Acyclic Graph (DAG)

Tree representation -

Syntax tree - Also known as Abstract Syntax tree or just Syntax tree. It is tree representation of the syntactic structure of source code written in a programming language.

ex- $a * a * (b - c) + (b - c) * d$



DAG - It is a special kind of abstract syntax tree. DAG for an expression identifies the common sub expressions of the expression.

"Sub-expressions that occur more than once"

" DAG is an intermediate code as well as code Optimi-sation technique".

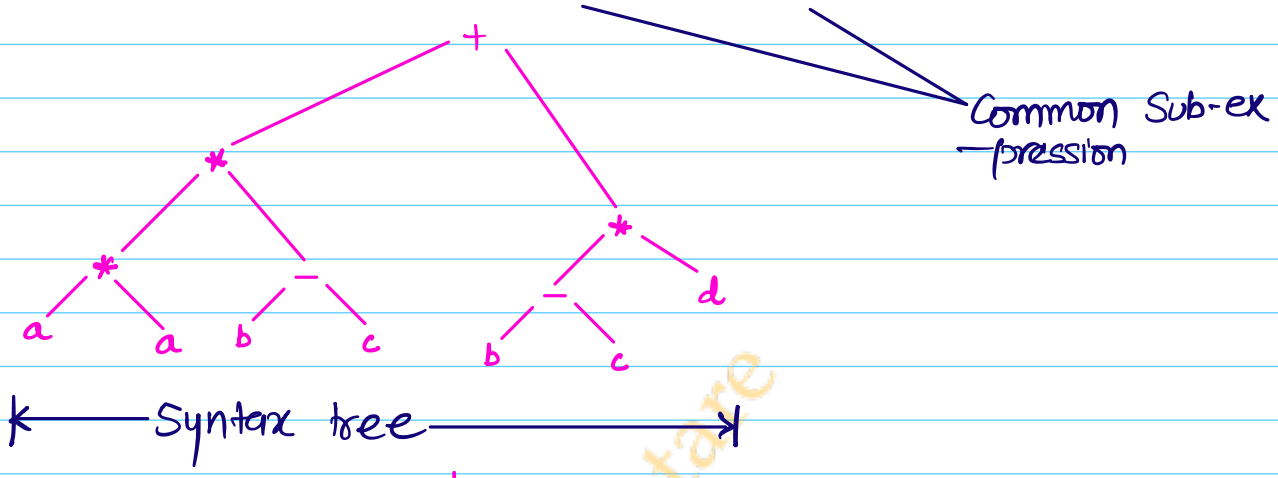
How to construct DAG -

1. Interior nodes always represent the operators
2. Exterior nodes (leaf nodes) always represent the names identifiers or constants.
3. Always check if there exists any node with the same value.
4. A new node is created only when there does not exist any node with same value.
5. This action helps in detecting the common sub-expressions

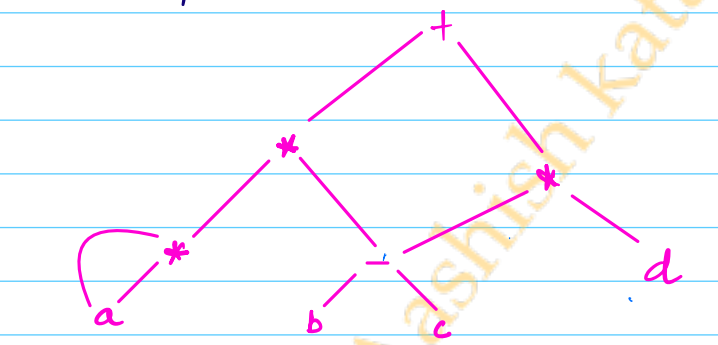
and avoiding the recomputation of the same.

6. Assignment instructions of the form X := Y are not performed unless they are necessary.

ex- $a * a * (b - c) + (b - c) * d$

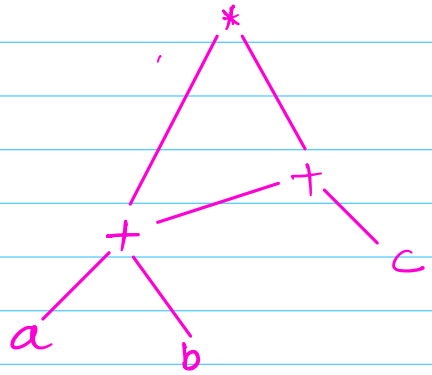


← Syntax tree →

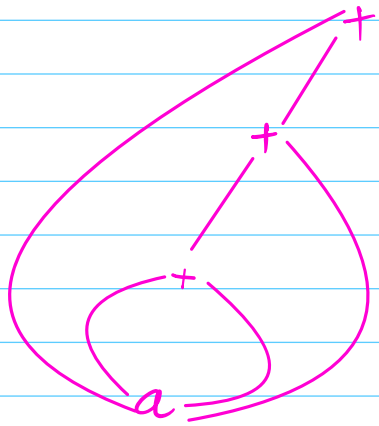


← DAG →

Q $(a+b) * (a+b+c)$



Q $a + a + a + a$



Linear representation-

Three-Address Code → In three address code, there is atmost one operator on the right side of an instruction, that is no built-up arithmatic expressions are permitted.

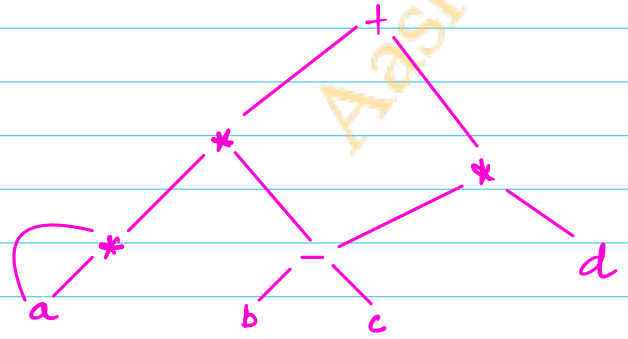
ex- $x + y * z$ — High level language statement

$t_1 = y * z$
$t_2 = x + t_1$ } 3-address code

" t_1 & t_2 are Compiler generated temporary names"

Three-address code is a linearized representation of a Syntax tree or a DAG in which explicit names corresponding to the interior nodes of the graph.

ex- $a * a * (b - c) + (b - c) * d$



DAG

$t_1 = b - c$
$t_2 = t_1 * d$
$t_3 = a * a$
$t_4 = t_3 * t_1$
$t_5 = t_2 + t_4$

3-address code

Note :- "In three address Code is a type of intermediate represen-tation used in compilers. Each instruction typically has at most three addresses.
(i) Two source operands (the values to be used)
(ii) One destination operand (where result is Stored)

Note :- $t = a + b$, we don't Count = Separately. The whole $t = a + b$ is treated as Single 3-address Code meaning perform addi

-tion and Store it.

Note :- t = a + b is one instruction performing one operation addition. = is just notation, not an extra operation

Note :- If you need both addition and assignment as separate steps, 3-address code breaks it further into multiple instructions.

Note :- Single static assignment form is a property of three address code where each variable is assigned exactly once.

"No value ever gets a "new value i.e reassignment".

"Every time a new value is Computed, it gets a new version of the variable".

This ensures that every variable name corresponds to exactly one definition in the program.

Postfix – In postfix notation, the operator follows the operand. For example, in the expression

(a-b) * (c+d) + (a-b), the postfix representation is –

ab - cd + * ab - +

Common three-address code instruction forms –

1. Assignment instructions of the form $x = y$ op z, where 'op' is a binary arithmetic or logical operation, and x, y, z are addresses.

2. Assignments of the form $x =$ op y, where 'op' is a unary operation.

3. Copy instructions of the form $x = y$, where x is assigned the value of y.

4. An unconditional jump goto L. The three-address instructions with label L is the next to be executed.

5. Conditional jumps of the form if x goto L and if false x goto L. These instructions execute the instruction with label L next if x is true and false respectively. Otherwise, the following three address instruction in sequence is executed next as usual.

6. Address and pointer assignments of the form $x = \&y$, $x = *y$, and $*x = y$. The instructions $x = \&y$ sets the r-value of x to the location of y.

7. Index Copy instruction of the form $x = y[i]$ and $x[i] = y$. The instruction $x = y[i]$ sets x to the value in the location i units beyond location y. The instruction $x[i] = y$ sets the contents of the location i units beyond x to the value y.

1. Ex- $a < b$ and $c > d$

1. if $a < b$ goto 4
2. $t_1 = 0$
3. goto 5

2. $a < b$ and $c > d$

1. if $a < b$ goto 4
2. $t_1 = 0$
3. goto 10

4. $t_1 = 1$
5. if $c > d$ goto 8
6. $t_2 = 0$
7. goto 8
8. $t_2 = 1$
9. $t_3 = t_1$ and t_2

4. $t_1 = 1$
5. if $c > d$ goto 8
6. $t_2 = 0$
7. goto 10
8. $t_2 = 1$
9. $t_3 = t_1$ and t_2
10. end of program

3. if $a < b$ then $x = y + z$

1. if $a < b$ goto 3
2. goto 5
3. $t_1 = y + z$
4. $x = t_1$
5. Rest of the program

4. While $a < b$ do $x = y + z$

1. if $a < b$ goto 3
2. goto 6
3. $t_1 = y + z$
4. $x = t_1$
5. goto 1
6. Rest of the program

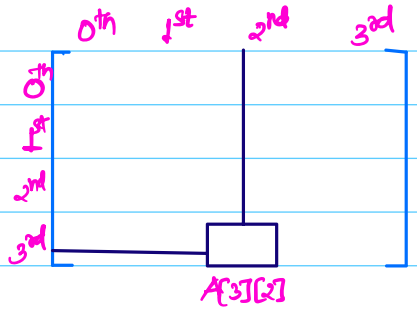
5. do $x = y + z$ while $(a < b)$

1. $t_1 = y + z$
2. $x = t_1$
3. if $(a < b)$ goto 1
4. Rest of the program

6. For(i=0, i<=20; i++)
$x = y + z$;

1. $i = 0$
2. if $(i <= 20)$ goto 4
3. goto 9
4. $t_1 = y + z$
5. $x = t_1$
6. $t_2 = i + 1$
7. $i = t_2$
8. goto 2
9. Rest of the program

7. A[4][4] given array
A[3][2] we want to access this element, hence
We want to convert this into 3-address code

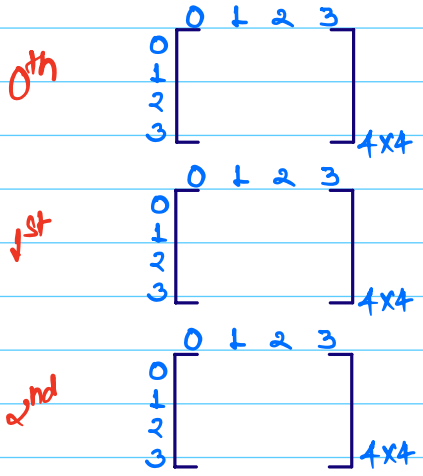


1. $t_1 = 3 * 4$
2. $t_2 = t_1 + 2$
3. $t_3 = t_1 + t_2$
4. $t_4 = t_3 * 2$ bytes
5. $t_5 = A[t_4]$

A[M][N] – given Size of an array
A[i][J] – An element

1. $t_1 = i * N$
2. $t_2 = t_1 + J$
3. $t_3 = t_1 + t_2$
4. $t_4 = t_3 *$ Size of int
5. $t_5 = A[t_4]$

8.



A[M][N][P]
A[i][J][K]

1. $t_0 = N \times P$
2. $t_1 = i * t_0$
3. $t_2 = J * P$
4. $t_3 = t_2 + K$
5. $t_4 = t_1 + t_3$
6. $t_5 = t_4 *$ Size of int
7. $t_6 = A[t_5]$

A[3][4][4] → Given
A[2][3][2] – Given element

1. $t_0 = 4 \times 4$
2. $t_1 = 2 * t_0$
3. $t_2 = 3 * 4$
4. $t_3 = t_2 + 2$
5. $t_4 = t_1 + t_3$
6. $t_5 = t_4 * 2$ bytes
7. $t_6 = A[t_5]$

Q. For a C-program accessing X[i][j][k], the following intermediate code is generated by a compiler. Assume that the size of an integer is 32 bits and the size of the character is 8 bits.

$t_0 = i * 1024 \Rightarrow t_0 = i * 32 * 8 * 4$ bytes $\Rightarrow t_0 = i * 1024$
$t_1 = j * 32 \Rightarrow t_1 = j * 8 * 4$ bytes $\Rightarrow t_1 = j * 32$
$t_2 = k * 4 \quad t_2 = k * 4$
$t_3 = t_1 + t_0$
$t_4 = t_3 + t_2$
$t_5 = X[t_4]$

Which of the following statements about the source code for the C-program is correct?

a) X is declared as int X[32][32][8]

Single Static Assignment forms -

Static Single assignment form is an intermediate representation that facilitates certain code optimizations Two distinctive aspects distinguish SSA from three address code.

" The first is that all assignments in SSA are to variables with distinct names, hence the term static Single assignment is used".

Why do we use SSA ?

Compilers can easily spot unused values, Common Sub-expressions or dead code. Data flow analysis, Liveness becomes simpler.

Ex -

$p = a + b$
$q = p - c$
$p = q * d$
$p = e - p$
$q = p + q$

3-address code

$p_1 = a_0 + b_0$
$q_1 = p_1 - c_0$
$p_2 = q_1 * d_0$
$p_3 = e_0 - p_2$
$q_2 = p_3 - q_1$

$p_1, q_1, p_2, p_3, q_2, a_0, b_0, c_0, d_0, e_0$

variables used

Ex - $a + (p * q) - c + (d * 5) - e + (u * v)/w$

$t_1 = p * q$
$t_2 = d * 5$
$t_3 = u * v$
$t_4 = t_3 / w$
$t_5 = a + t_1$
$t_6 = t_5 - c$
$t_7 = t_6 + t_2$
$t_8 = t_7 - e$
$t_9 = t_8 + t_4$

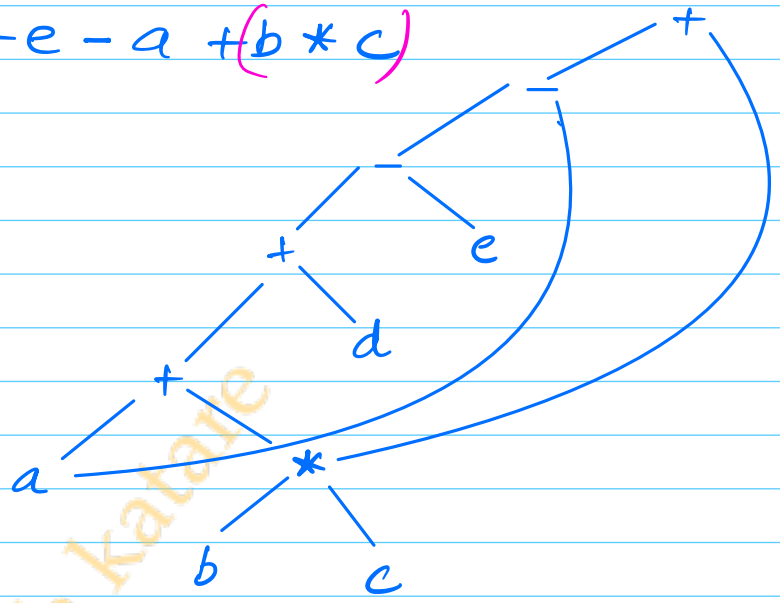
Above is the 3-address code as well as written in single static form.

" SSA is a property of an intermediate code which requires that each variable is assigned exactly once, and every variable is defined before it is used. "

Ex- $a+(b*c)+d-e-a+(b*c)$

$t_1 = b*c$
$t_2 = a+t_1$
$t_3 = t_2+d$
$t_4 = t_3-e$
$t_5 = t_4-a$
$t_6 = t_5+t_1$

SSA



$t_1 = b*c$
$t_2 = a+t_1$
$t_2 = t_2+d$
$t_2 = t_2-e$
$t_2 = t_2-a$
$t_2 = t_2+t_1$

Minimum number of temporary variables used without SSA

Q

Consider the following code segment:

$x = u - t;$
$y = x * v;$
$x = y + w;$
$y = t - z;$
$y = x * y;$

The minimum number of *total* variables required to convert the above code segment to *static single assignment* form is 10.

[2016 (Set-1) : 1 M]

$x_1 = u_0 - t_0$
$y_1 = x_1 * v_0$
$x_2 = y_1 + w_0$
$y_2 = t_0 - z_0$
$y_3 = x_2 * y_2$

x_1 x_2 y_1 y_2 y_3 u_0 v_0 t_0 w_0 z_0

Q

4.14 The least number of temporary variables required to create a three-address code in static single assignment form for the expression q+r/3+s−t * 5+u * v/w is ___8___.

[2015 (Set-1) : 2 M]

Static Single assignment form

$q + (r/3) + s - (t * 5) + (u * v/w)$

$t_1 = r/3$
$t_2 = t * 5$
$t_3 = u * v$
$t_4 = t_3/w$
$t_5 = q + t_1$
$t_6 = t_5 + s$
$t_7 = t_6 - t_2$
$t_8 = t_7 + t_4$

Q

4.19 Consider the following intermediate program in three address code

$p = a - b$
$q = p * c$
$p = u * v$
$q = p + q$

Which one of the following corresponds to a static single assignment form of the above code?

(a) $p_1 = a - b$
$q_1 = p_1 * c$
$p_1 = u * v$
$q_1 = p_1 + q_1$

(b) $p_3 = a - b$
$q_4 = p_3 * c$
$p_4 = u * v$
$q_5 = p_4 + q_4$

(c) $p_1 = a - b$
$q_1 = p_2 * c$
$p_3 = u * v$
$q_2 = p_4 + q_3$

(d) $p_1 = a - b$
$q_1 = p * c$
$q_2 = u * v$
$q_2 = p + q$

[2017 (Set-1) : 1 M]

Single static assignment form—

$p_1 = a_0 - b_0$
$q_1 = p_1 * c_0$
$p_2 = u_0 * v_0$
$q_2 = p_2 + q_1$

a) In this option p_1 is assigned more than one time hence false.

b) Correct option
$p_3 = a_0 - b_0$
$q_4 = p_3 * c_0$
$p_4 = u_0 * v_0$
$q_5 = p_4 + q_4$

c) $p_1 = a - b$
$q_1 = p_2 * c$
$p_3 = u * v$
$q_2 = p_4 + q_3$

d) $p_1 = a - b$
$q_1 = p * c$
$q_2 = u * v$
$q_2 = p + q$

Q

3.1 Generation of intermediate code based on an abstract machine model is useful in compilers because

(a) it makes implementation of lexical analysis and syntax analysis easier

(b) syntax-directed translations can be written for intermediate code generation

(c) it enhances the portability of the front end of the complier

(d) it is not possible to generate code for real machines directly from high level language programs

[1994 : 1 M]

Correct option is "it enhances the portability of the front end of the compiler".

Representation of 3-address Code -

(I) Quadruple (II) Triple (III) Indirect triple

The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions. Can be implemented as objects or as records with fields for the operator and the operands.

Example -

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

$t_1 = a+b$
$t_2 = -t_1$
$t_3 = t_1+c$
$t_4 = c+d$
$t_5 = t_2 * t_4$
$t_6 = t_5 + t_3$

Quadruple (only 4 columns)

Pointer to Quad	Operator	Operand 1	Operand 2	Result
(1)	+	a	b	t_1
(2)	−	t_1		t_2
(3)	+	t_1	c	t_3
(4)	+	c	d	t_4
(5)	*	t_2	t_4	t_5
(6)	+	t_5	t_3	t_6

not a part of Quadruple

Disadvantages of Quadruple –

1. Contain lot of temporary variables
2. Temporary variables creation increases time and space complexity.

Note :- for readability, we used actual identifiers like a, b and c in the fields operand1, operand2, result in above quadruple representation. But, actually they are pointers to their symbol table entries.

Temporary names are just names.

Triple – A triple has only three fields, which we call operator, operand1, operand2. Note that the result field in quadruple is used primarily for temporary names. Using triples, we prefer to the result of an operation x OP y but its position (1). Parenthesised numbers represent pointers into triple structure itself.

Triple

Pointer to Qual	Operator	Operand1	Operand2
(1)	+	a	b
(2)	−	(1)	
(3)	+	(1)	c
(4)	+	c	d
(5)	*	(2)	(4)
(6)	+	(5)	(3)

Advantage – (1) Easy to rearrange code for global optimization
(2) One can quickly access value of temporary variables using symbol table.

Disadvantage – (1) Temporaries are implicit and difficult to re-arrange code.

(2) It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved any other triple referring to it must be updated also. With the help of pointer one can directly access symbol table entry. The problem does not occur with indirect

triples, which we consider next.

Indirect triples – Indirect triple makes use of pointer to the listing of all references to computations which is made separately and stored. It is similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

SQ.NO	pointer to Triple

Triple

Pointer to Quad	Operator	Operand 1	Operand 2
(1)	+	a	b
(2)	−	(1)	
(3)	+	(1)	c
(4)	+	c	d
(5)	*	(2)	(4)
(6)	+	(5)	(3)

Basic Blocks and flow graphs -

The graphical representation of intermediate code that is helpful for discussing code generation even if the graph is not constructed explicitly by a code generation.

Control flow graph is Constructed as follows -

1. Partition the intermediate code into basic blocks, which are maximal sequences of consecutive three-address instructions with the properties that -

a) The flow of control can only enter the basic block through the first instruction in the block. That is there are no jumps into the middle of the block.

b) Control will leave the block without halting or branching except possibly at the last instruction in the block.

2. The basic block become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.

Basic blocks - Our first job is to partition a sequence of three address instructions into basic blocks. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction.

" In the absence of jumps and labels, control proceeds sequentially from one instruction to the next ".

" To find basic blocks we need to find leaders ".

First, we determine those instructions in the Interme-diate code that are leaders, that is, The first instruction in some basic block.

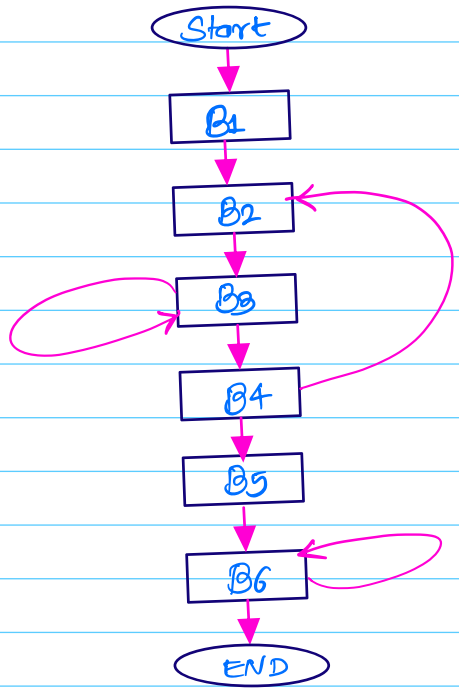
Rules to find Leader –

1. The first three-address instruction in the interm-ediate code is a leader.
2. Any instruction that is target of a conditional or un-conditional jump is a leader.
3. Any instruction that immediately follow a Conditional or un-conditional jump is a leader.

" for each leader, its basic block consists of itself and all instructions upto but not including the next leader or the end of the intermediate program is a leader ".

Q Construct CFG for given 3-address codes.

```
1.  i = 1          B1 → L
2.  J = 1          B2 → L
3.  t1 = 10 * i    → L
4.  t2 = t1 + J
5.  t3 = 8 * t2                 B3
6.  t4 = t3 - 88
7.  a[t4] = 0.0
8.  J = J + 1
9.  if J <= 10 goto (3)
10. i = i + 1      → L
11. if i <= 10 goto (2)         B4
12. i = 1          → L B5
13. t5 = i - 1     → L
14. t6 = 88 * t5                B6
15. a[t6] = 1.0
16. i = i + 1
17. if i <= 10 goto (13)
```



Number of nodes = 8

Number of edges = 10

" Always count Start & entry as node ".

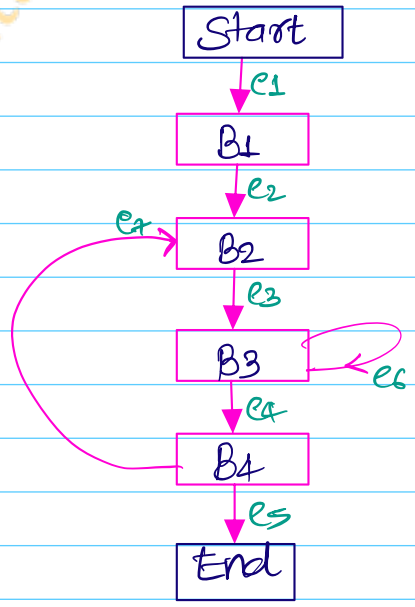
"Once an intermediate code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the graph are the basic blocks. There is an edge from block B to C if and only if it is possible for the first instruction in block C to immediately follow the last instruction in block B".

Note :- We often add two nodes called entry and exit, that do not correspond to executable intermediate instructions.

Q Consider the intermediate code given below –

```
1.  i = 1                    B1
2.  j = 1                    B2
3.  t1 = 5 * i               B3
4.  t2 = t1 + j
5.  t3 = 4 * t2
6.  t4 = t3
7.  a[t4] = -1
8.  j = j + 1
9.  if j <= 5 goto (3)
10. i = i + 1                B4
11. if i < 5 goto (2)
```

Leader (1) → B1
Leader (2) → B2
Leader (3) → B3
Leader (10) → B4



The number of nodes and edges in the control flow graph constructed for the above code respectively are –

a) 5 and 7 b) 6 and 7 c) 5 and 5 d) 7 and 8

Imp Count start & end as node of CFG

Q Consider the following pseudo code-

```
1.  t1 = -1                          → Leader   B1
2.  t2 = 0
3.  t3 = 0                           → Leader   B2
4.  t4 = 4 * t3                      → Leader   B3
5.  t5 = 4 * t2
6.  t6 = t5 * M
7.  t7 = t4 + t6
8.  t8 = a[t7]
9.  if t8 <= Max goto 11
10. t1 = t8                          → Leader   B4
11. t3 = t3 + 1                      → Leader   B5
12. if t3 < M goto 4
13. t2 = t2 + 1                      → Leader   B6
14. if t2 < N goto 3
15. max = t1                         → Leader   B7
```

(4–9: 6 statements)

Which of the following options correctly specifies the number of basic blocks and the number of instructions in the largest basic blocks-

a) 6 & 6 b) ~~7 & 6~~ c) 6 & 7 d) 7 & 7

Q Consider the following 3-address code

```
t1 = t + e
t2 = g + a
t3 = t1 * t2
t4 = t2 + t2
t5 = t4 + t3
```

There are five temporary variables in above code. find minimum number of temporary variables can be used in the equivalent optimised 3-address code of above code.

In the question they have not mentioned Static Single assignment hence we can reuse the temporary variable.

```
t1 = t + e
t2 = g + a
t1 = t1 * t2
t2 = t2 + t2
t2 = t2 + t1
```

There are many ways to do this and here 2 is the answer.

Q. What will be the optimized code when the expression $p = q * -r + q + -r$ is represented in DAG?

a)
$t_1 = -r$
$t_2 = q * t_1$
$t_3 = t_1 + t_2$
$t_4 = t_2 + t_3$
$p = t_3$

b)
$t_1 = -r$
$t_2 = q$
$t_3 = t_1 * t_2$
$t_4 = t_3 + t_3$
$p = t_4$

c)
$t_1 = -r$
$t_2 = q * t_1$
$t_3 = q * t_1$
$t_4 = t_2 + t_3$
$p = t_4$

d) ✓
$t_1 = -r$
$t_2 = q * t_1$
$t_3 = q + t_1$
$t_4 = t_2 + t_3$
$p = t_4$

"DAG doesn't mean SSA"

Q

4.12 Consider the basic block given below:

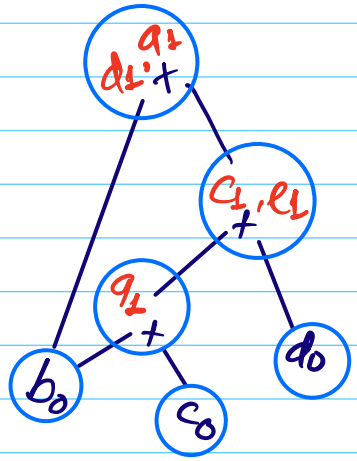
$a = b + c, c = a + d, d = b + c$
$e = d - b, a = e + b$

The minimum number of nodes and edges present in the DAG representation of the above basic block respectively are

(a) 6 and 6 ✓ (b) 8 and 10
(c) 9 and 12 (d) 4 and 4

[2014 (Set-3) : 2 M]

$a = b + c$
$c = a + d$
$d = b + c$
$e = d - b$
$a = e + b$



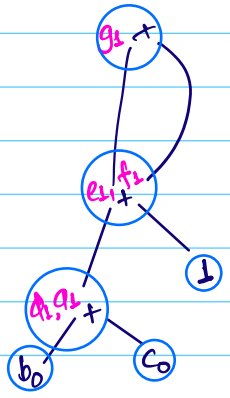
Q

4.22 Consider the following C code segment:

$a = b + c;$
$e = a + 1;$
$d = b + c;$
$f = d + 1;$
$g = e + f;$

In a compiler, this code segment is represented internally as a directed acyclic graph (DAG). The number of nodes in the DAG is ___6___.

[2021 (Set-1) : 2 M]



Live variable Analysis → A variable is live at a program point if its value will be used along the same path in the future before being redefined.

" If we throw away the current value of that variable, will the program's behaviour change later?
If yes, the variable is live
If No, the variable is dead ".

What is the meaning of "throw away the current value"?
Throw away means overwrite or delete the current value of a variable and the program never uses that value again in the future then program's behaviour will not change. The variable is said to be dead at that point. If the current value will be read/used later then discarding it will change the program's behaviour. In this case the variable is live.

Local Liveness analysis –

1. Done inside a single basic block
2. Local analysis is quick and limited view
3. We only analyse the variable inside the block without considering other blocks or control flow.

Global Liveness analysis –

1. To find which variables must be kept in registers at each program point.
2. To eliminate dead code
3. To help register allocation. If two variables are not live

at the same time, they can share the same register.

In other words

Live variable Analysis - Assume an instance where we have 500 variables in a program but only 200 registers. As we know we can't accomodate all variables into registers then we have to do live variable analysis. Those variable which are live we can put them into register.

EX- $R_1 = a$ 'a' is live hence it is placed in register
now assume 'a' is dead then placing variable back to the secondry memory is known as memory spilling.

" A variable is live then it doesn't mean that we should put it in a register. We can do it if reqiured we allocate register only to live variables."

A variable is said to be live at any statement if it is used in that statement or anywhere in the subsequent program before it is defined.

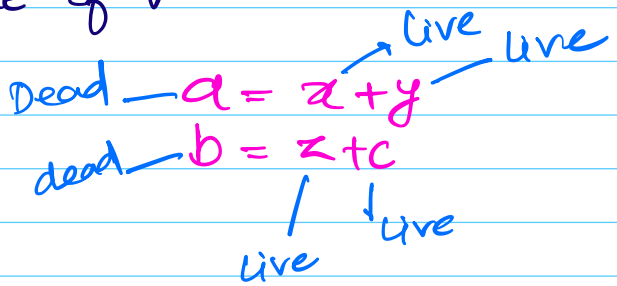
$a = b + c$
↓
'a' is dead

$x = a + b$
↑
'a' is used hence live, x is dead, b is live

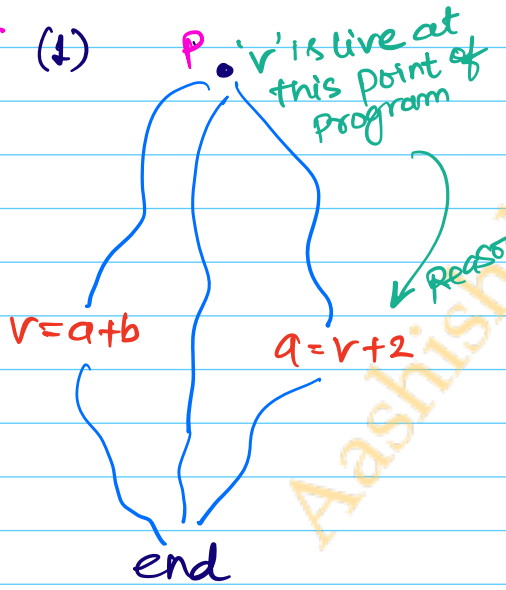
When you find a variable in RHS means live and if you find it on LHS then dead but what if $a = a + b$?

live↓
$a = a + b$
↓Dead

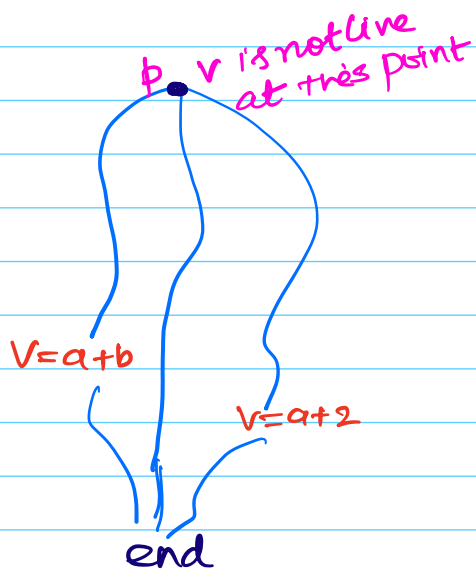
A variable 'v' is live at a program point 'p' if some path from p to program exit contains a read value occurrence of 'v' which is not preceeded by written value occurrence of 'v'.



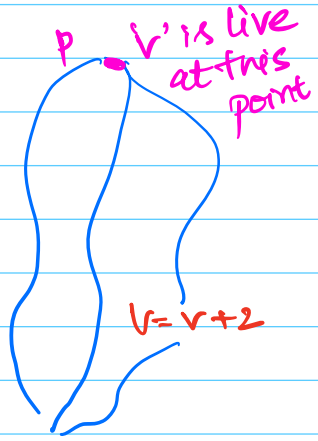
ex- (1)



(2)

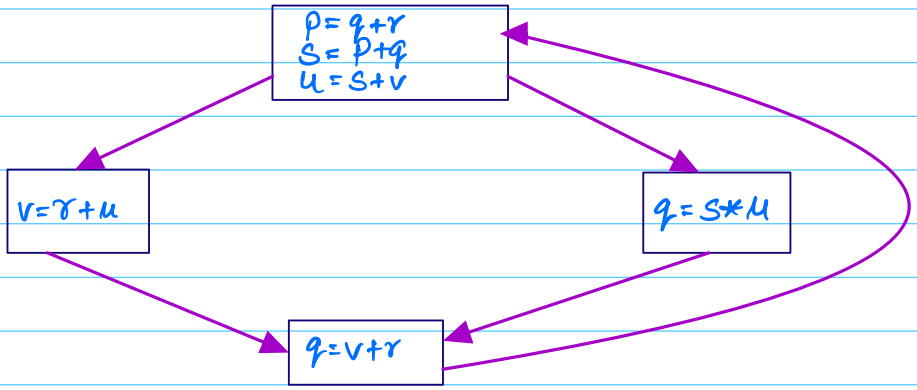


3)



In example (1) & (2), we said variable 'v' is live because the variable is useful later so we wont to preserve it hence live.

Example - find out the variables which are live at B_2 & B_3 both



IN - Set of variables live at the beginning of the block
" Live at the entry "
These variable must already hold correct values before block starts otherwise the program will be incorrect.

OUT - Set of variables live at just after the block (exit)
" These are variables whose values are needed either in successor blocks or later in the program ".

USE/GEN - Set of variables that are used in the block before any assignment i.e the variables in RHS but not in LHS of preceeded statement in that block.
" These variables must already have a value when the block starts ".

DEF/KILL - Set of variables that are defined (variables in LHS)
" These variables get a new value in the block so their previous values are killed ".

$$IN = USE \cup (OUT - KILL)$$

$$OUT = \cup (IN \text{ of all Successor Nodes})$$

			← 1st Iteration		→ 2nd Iteration ←	
Node	USE	KILL	IN	OUT	IN	OUT
1	q, r, v	p, s, u	q, r, v	r, s, u	q, r, v	r, s, u, v
2	r, u	v	r, u	v, r	r, u	r, v
3	s, u	q	s, u	v, r	r, s, u, v	r, v
4	v, r	q	v, r	q, r, v	r, v	q, r, v

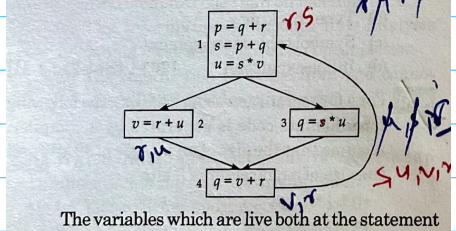
Variables r, u are live at B_2 & B_3

Note :- Continue the algorithm till thom you find two iterations same.

The frost iteration gives you liveness of variables locally. It is more like "local + whatever OUT had, more or less it equivalent to liveness locally. The global picture only emerges after propagation in subsequent iterations.

4.13 A variable x is said to be live at a statement S_i in a program if the following three conditions hold simultaneously:

1. There exists a statement S_j that uses x
2. There is a path from S_i to S_j in the flow graph corresponding to the program
3. The path has no intervening assignment to x including at S_i and S_j



The variables which are live both at the statement in basic block 2 and at the statement in basic block 3 of the above control flow graph are

(a) p, s, u (b) r, s, u
(c) r, u (d) q, v

[2015 (Set-1) : 2 M]

→ Answer to this question is (C) r, u if you find liveness of variable globally. The above question has been solved by using algorithm and algorithm checks liveness globally.

Above question & this GATE question both are same.

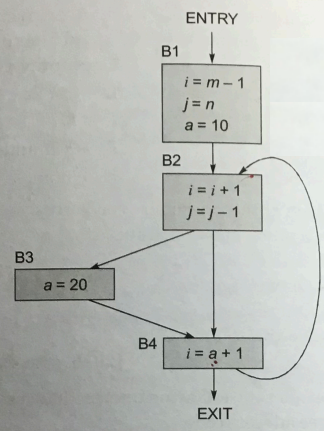
" If someone says in this question they have asked liveness locally in Block 2 and 3 thon answer will be different". The answer in that case will be 'u'.

" In exam Consider global liveness and use algorithm".

			1st Iteration		2nd Iteration	
Node	USE	KILL	IN	OUT	IN	OUT
1	q, r, v	p, s, u	q, r, v	r, s, u	q, r, v	r, s, u, v
2	r, u	v	r, u	v, r	r, u	r, v
3	s, u	q	s, u	v, r	r, s, u, v	r, v
4	v, r	q	v, r	q, r, v	r, v	q, r, v

→ gives liveness locally

4.26 Consider the control flow graph shown:



Which one of the following choices correctly lists the set of live variables at the exit point of each basic block?

(a) B1: { }, B2: {a}, B3: {a}, B4: {a}
(b) B1: {i, j}, B2: {a}, B3: {a}, B4: {i}
(c) B1: {a, i, j}, B2: {a, i, j}, B3: {a, i}, B4: {a}
(d) B1: {a, i, j}, B2: {a, j}, B3: {a, j}, B4: {a, i, j}

[2023 : 2 M]

1st iteration → 2nd → 3rd →

	USE	DEF	IN	OUT	IN	OUT	IN	OUT
B_1	m, n	i, j, a	m, n	i, j	m, n	a, i, j	m, n	a, i, j
B_2	i, j	i, j	i, j	a	a, i, j	a, j	a, i, j	a, j
B_3	ϕ	a	ϕ	a	ϕ	a, j	j	a, j
B_4	a	i	a	i, j	a, j	a, i, j	a, j	a, i, j

Continue finding iteration till the point you find two iterations same.

4.24 For a statement S in a program, in the context of liveness analysis, the following sets are defined:

$USE(S)$: The set of variables used in S
$IN(S)$: The set of variables that are live at the entry of S
$OUT(S)$: The set of variables that are live at the exit of S

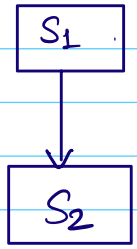
Consider a basic block that consists of two statements, S_1 followed by S_2.

Which one of the following statements is correct?

(a) $OUT(S_1) = USE(S_1) \cup IN(S_2)$
(b) $OUT(S_1) = IN(S_2) \cup OUT(S_2)$
(c) $OUT(S_1) = IN(S_2)$
(d) $OUT(S_1) = IN(S_1) \cup USE(S_1)$

[2021 (Set-2) : 2 M]

We know all the definitions written in the question
Give basic block S_1 is followed by S_2



IN = Use ∪ (OUT − DEF)

OUT = Union of IN of all immediate successor

Hence $out(S_1) = IN(S_2)$

Semantic Analysis – After Syntax analysis parsing ensures that a program follows the grammar of the language, Semantic analysis checks whether the program makes sense according to the rules of the language.

ex- `int x = true;` No syntax error but semantically wrong.

" Meaningful correctness".

What Semantics does it check ?

Semantic analysis deals with static semantics means things that can be checked without running the program.

a) Type checking → Operations must apply to compatible types
b) Scope checking → variable must be declared before use
c) Identifier checking → No multiple declaration in the same scope
d) Function / procedure checking → must match to the type of parameters
e) Control-flow Semantics → Breaks must appear inside
f) Array and pointer Semantics → Indexes, dereferences
g) Consistency rules → functions must have return.

Why Semantic Analysis is Important-

(a) Prevent non-sense programs
(b) Ensures type safety and language rules enforcement
(c) Provides meaningful Compiler errors to the programmar
(d) prepares for intermediate Code generation

" Semantic analysis is performed by Semantic analyzer using AST + attribute grammar + Symbol table".

"Parser + SDT"

Symbol table — One of the most important data structures inside a compiler.

"A Symbol table is a data structure used by the compiler to store information about program identifiers i.e. variables, functions, classes, objects, constants etc."

Each entry in the table contains attributes of the identifier such as —

Name
Type (int, float, array, struct, function - return type)
Scope
Storage

"It is Compiler's dictionary and it is used by all the phases".

The Symbol table must support fast lookup, insertion, and deletion because the Compiler queries it very frequently.

It is implemented by —

(1) Hash table (Most Common)
O(1) lookup & insertion

(2) Tree based structures
Binary search tree, AVL trees
O(log n)

(3) Linear list (Simple but slow)
Array or Linked list
O(n)