CW

16/2/24

Compiler design

↓

application of formal lang. (this is the diff b/w HLL → Compiler → LLL (C.d. & f.L.)

Prog.c → Compiler → Prog.exe

Compiler → System S/W

↓

translator

Cousins of ~~~~~ compiler

PYQ 23 Prog. c (high level lang.)

↓ Pre-Processing → macro expansion; removing preprocessor directives

Prog. I (Extended high l.l)

↓ Compiler

Prog.asm (Assembly level code)

↓ Assembler

Prog. obj (low level language)

+

library fn. obj (linker) → (a computer prog that combines 1 or more obj files into a single executable file, library file or another obj file)

↓ loader (a component of an OS that loads progs & libraries)

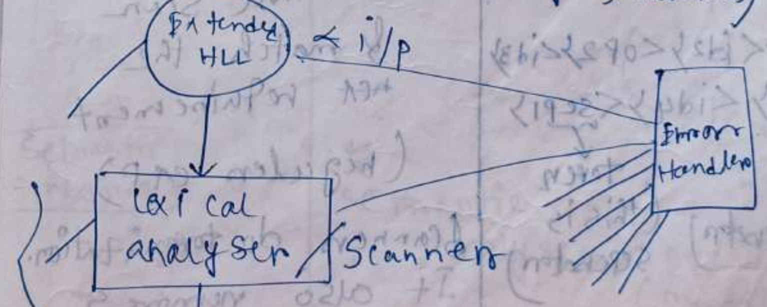
Prog. exe

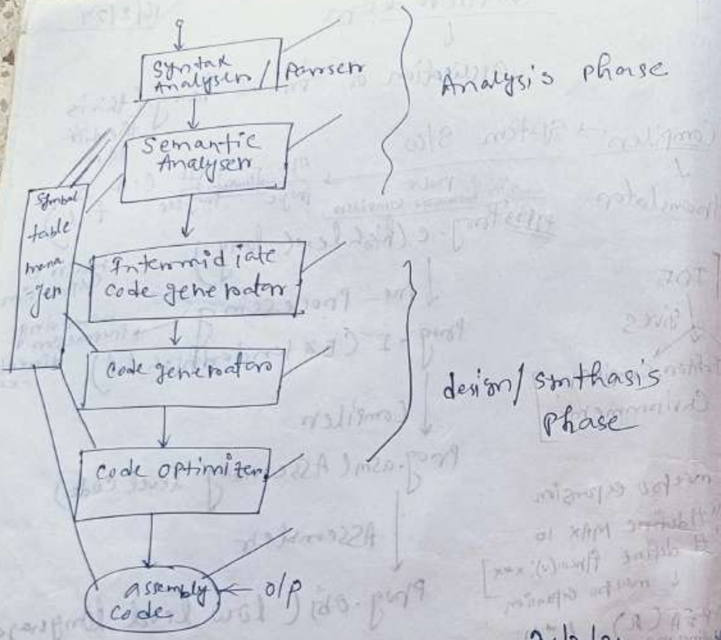
[IDE gives Editor, Compiler, Environment]

[macro expansion
#define MAX 10
define AREA(x) x*x]
↓ macro expansion

[AREA(n) n*n
#define MAX 20
main()
{
if (a[i] > 20)
P = n*n
:
}]

Compiler





20/2/24

CN (E.H.L.L) → (source code)

a = b + c + d ; → 1 → n f t → d f

↓ Scanner

Tokens → o/P (meaningful words)

1 chart at a time will be scanned by scanner/lexical analyser. It will scan & match the requirement (regular exp)

in CS, variable is called identifier

→ <id1><OP1><id2><OP2><id3> <OP3><id4><sep1>

token (identifier), token (operator), token (this is seperator)

this is strings of tokens

Scanner do tokenization. It also removes the comment lines.

ST M (Symbol table manager) this will maintain all symbols (identifier, seperator, operator)

It is a datastructure

[symbol table → database]

eg

① itn n ; → This is not error by scanner, i.e scanner can't find errors in this.
id1 id2 seperator (sep1)

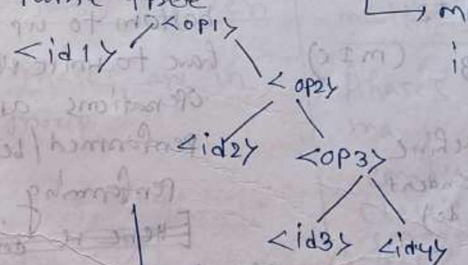
② int 10x ; → Lexical error. This isn't match with the reg ex.
keyword const id sep

<id1><OP1><id2> . . . . . <sep1>

↓ Parser/ syntax analyser (: it is (FA))

└→ main role of this is derivation. It generates parse tree / syntax tree. If parse tree doesn't match then it will show syntax error

Parse tree

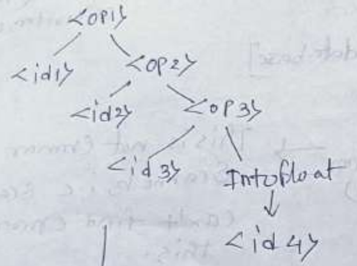


[Grammers of this E → E+E | E*E | id]

Semantic = meaning

semantic analyser / SDT (Syntax directed translation)

Anotated syntax tree



SA can find the errors named 'type casting'.

[if a, b, c → float, d → int then we can't do mul of int & float then we need to correct 'type casting' errors & redraw the correct tree]

Assembly lang. always targets some machine

↓ ICG (machine independent code)

t_1 = inttofloat (id4) (3rd variable)

t_2 = id3 + t_1

t_3 = id2 + t_2

id1 = t_3 (MIC)

→ we have to read the tree from bottom to up & have to write which operations are performed/being performing

~~Here we don't~~

the o/p of ICG is machine independent code. (as here we don't need to depend upon machine)

- ICG generates machine independ code

↓ CG (machine dependent code)

```
LD  R1 id4
LD  R2 id3    → (2 level instruction format) → (Assembly level lang)
MUL R1 R2
LD  R3 id2
ADD R1 R3
mov id1 R1    (MDC)
```

↓ CO

```
LD  R1 id4
LD  R2 id3
MUL R1 R2
LD  R2 id2   → As R2 is free now we are using this resource rather than taking an extra resource (optimization).
ADD R1 R2
mov id1 R1
```

(optimited Assembly level code)

- CG generates machine dependent code

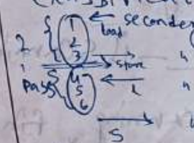
[IP 1 level ins → we will use AC]

[(1) for(i=0; i<2; i++) P("Allah"); } we need div here
(2) P("Allah"); P("Allah"); } more optimized]

Compiler construction ... 21/2/24 (theory needed)

Qn group of phases based name

One-pass compiler	multipass compilers
1) All phases are in 1 group.	1) Phases are divided into many groups. (2-pass compiler - ours 1st classification) eg- Java
2) Execution is faster than M.P.C.	2) Execution is slower than O.P.C
3) Doesn't have multipurpose usability.	3) Has multipurpose usability.



i) called narrow compiler	i) called a wide compiler.
ii) Has a limited scope.	ii) Has a great scope.
iii) There is no code optimization.	iii) There is code optimization.
iv) There is no intermediate code generation.	iv) There is i, d, gi
v) takes a minimum time to compile.	v) takes some time to compile.
vi) memory consumption is lower.	vi) memory consumption is higher.

Compiler Construction tools (short note comp) Pg23

H can handle semantics also

1) Lex/flex → LA (rule; tokenization)
2) YACC (Yet another compiler compiler) → rule; to check syntax.
3) SDT engine (syntax directed translation engine) → rule; Same as semantic analyser.
4) Code generator → rule; generates code
5) flow engine → rule; optimization

Lexical analyser

Roles of LA — i) Scanning (1 → a, 1 → n, 1 char at a time)

ii) After scanning, finding / trying to terms / trying to match with some pattern ie reg ex.

iii) Tokenization

iv) comment line removal.

v) white space removal. [Int x; → Int x;] [In compiler multiline codes are analyzed & to compiler it is a single line code]

Reg Ex of lang starting with 'ab', Σ = {a, b}

Reg ex; ab(a+b)* → ab(a|b)* (in compilers (Lex flex)) "ab"[ab]* (targeting the lex code)

'1' as 't' isn't a int value.

Integers constant

① Reg ex; (0+1+2+3+4+5+6+7+8+9)+ [(0|1-9)+ Not correct]

in compilers → (0|1|2|3|4|5|6|7|8|9)+ → [0123456789]+

In flex /compilers, [abc] → [ ] → class format]

means either a' or 'b' or 'c'

without class → a|b|c

In b/w class we can define range ∴ → [0-9]+ → the reg ex of int. const.]

② [0 1 2 5 6 7 8] → As we can't write → [0-2 5-8] [0 1 2 . 5 6 7 8] space is a char in reg ex]

③ [^ab]* → this can be anything except 'ab'. [not 'a' not 'b']

[^abcd]* → this ... except 'a', 'b', 'c', 'd'

④ [a_z] → either 'a' or '_' (underscore) or 'z'

alphabet (reg ex)

① reg ex: [A-Z a-z]+

variable or Identifiers

Reg ex: P.E. of identifiers ← pattern (not reg ex)

↓

[alphabet] [alphabet + number + _]*

reg ex ↓

[A-Z a-z] [A-Z a-z 0-9 _]* → [Here 'or' means nothing to write]

int _x;

↳ From above this is matched with variable, but it isn't a variable. So, that we have to define the reg ex. of the keywords first.

Reg ex of keywords

- " " → Exact matching (literals)
- keywords have no reg ex., it is exact matching.
- v ("if"|"else") → we can't define the reg ex of keywords in a class.

reg ex of keywords

- We have to write the reg ex of keywords at the first of prog then the reg ex of variables / identifiers.
- ["if" "else"] → not correct, as too many 'or's as every char is treated as single char.

- "." 59 → If there's a single chars only then we can write it in the class. [Y.]

(bt chars it is conts. it define some range)

Floating constants

Reg ex - [0-9]+ "." [0-9]+ → [In C Prog we can't write '.09' for this reg ex] 23|2|24

It gives the

Che LA does → lexemes the name tokens (tokenization)

Int x ; [Every i/p is lexeme]

x → lexeme, which LA gives name identifier

Pg 23	RE
integer constant	[0-9]+
floating "	[0-9]+ "." [0-9]+
identifiers	[a-z A-Z] [a-z A-Z 0-9 _]*
keywords	("int" | "if" | "else")
operators	"++" | ">=" | "<=" | "=="
	[=+\-/* ><]

[as a·t = a] [But normally t is not a string (i/p) [we have to write all keywords like this]

Reg ex [eg- ("++" | "+") → longest one will be at 1st → then the end ones

If the prefix is same then we have to see the longest reg ex. look for

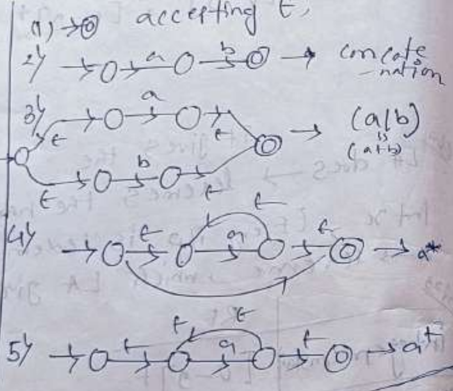
Seperators | [, ;) ({ } tab \n] space → It is itself a symbol so we can keep it into / write it into class.

[" " → literals]

• The defination of keywords should be above the variables.

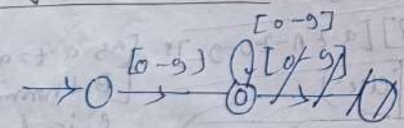
① if (p>0)
q++;
else
q=q-1;
(imp for exam)

1) accepting ε,
2) concatenation
3) (a|b) (a+b)
4) a*
5) a+

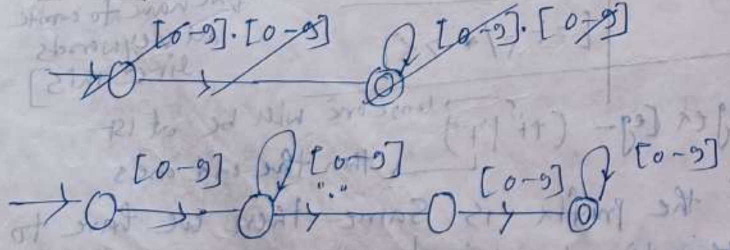


F.A

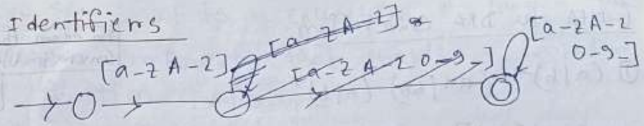
1) Integer constant



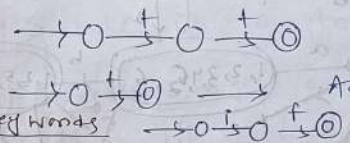
2) Floating constant



3) Identifiers

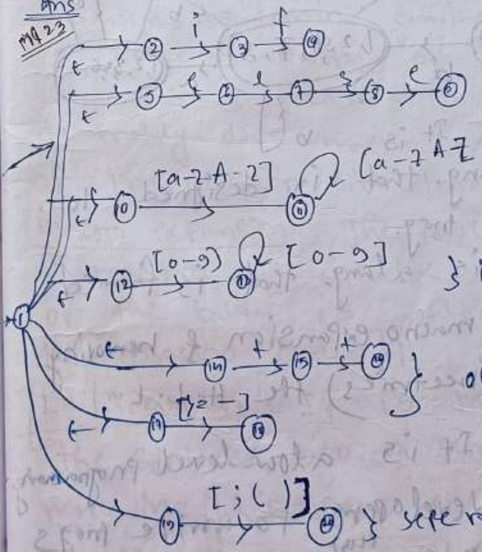


4) Operators



5) keywords

Ans
MQ23



[1st keywords]

[for identifiers we need to write generic exp]

[can ignore space, tab newline(\n)]

the finite automata of Ans ①

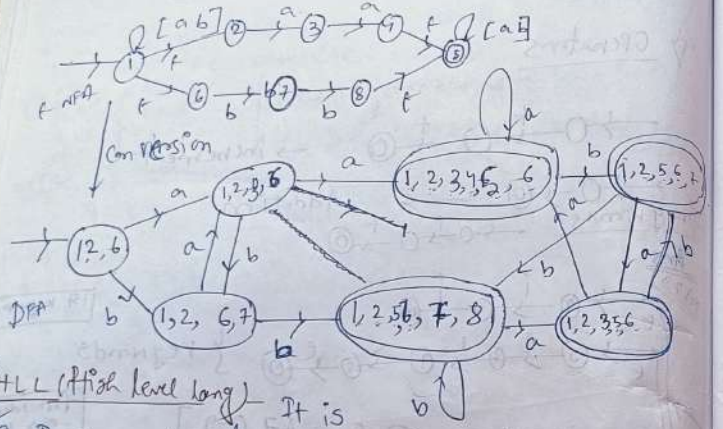
• operators can also act as seperators as (int we acting as sep)

∴ sep = [\t\n ;(){}=><]

~~Final~~ ε-NFA to DFA conversion

For exam this conversion should not be done

① $(a|b)^*(aa|bb)(a|b)^*$



HLL (High level lang) - It is a Programming lang. that is designed to simplify Computer progs.

E.H.L.L - F.H.L.L is a lang. that is formed by PreProcessing (macro expansion & removing Preprocessor directives) the H.L.L.

Assembly level code - It is a low-level programming lang. that allows developers to write progs that run directly on a computer's CPU.

Low L.L - It is a programming lang that provides little or no abstraction from a computer's instruction set architecture. It deals with a computer's hardware components & constraints.

System S/w - It is a collection of progs & software components that enable a computer to function properly.

Compiler - It is a software prog that translates source code into machine code, bytecode or another programming lang. It's role is to translate H.L.L into L.L.L.

Pre Processor - It is a prog that process-es it's i/p data to produce o/p that is used as i/p in another prog. It is a process where a preprocessor needs symbols & macros to substitute. It do mainly does macro expansion & removal of preprocessor directives.

macro expansion - The process of replacing a macro call with the processed copy of the body is called expansion of the macro call.]||

[LA / Scanner - It is responsible for breaking the syntaxes into a series of tokens, by removing whitespace in the source code. It is also known as scanner. It takes source code as i/p & scans it & give a series of tokens as o/p. Token is a predefined sequence of chars that represents a unit.

Syntax analyzer - It is a process that checks if source code follows the grammatical rules of a programming lang. It is also known as parsers. It generates parse tree.

Semantic analyzer - It is a process that checks a program's semantic consistency with the language definition. It makes sure that declaration & statements of prog are semantically correct. It uses a syntax tree & symbol table to gather type info & check for semantic errors. It is also called SDT if it generates annotated syntax tree.

Intermediate Code generator - It is the process of creating a representation of source code that is closer to the target machine lang. (ICG generation), The machine which generates it is called ICG. It generates machine independent code.

code generator - It is a compiler that translates the intermediate representation of the source prog. into target prog. In other words, a code generator translates an abstract syntax tree into machine-dependent executable code. It generates machine dependent code.

Code optimizer - Code optimization is the process of improving the performance & efficiency of a prog. by changing its code. This is done by code optimizer. It's main objectives are - i) reduce resource consumption.
ii) Decrease execution in time.
iii) Ensure that the prog uses minimal memory.
iv) Lessens CPU time
v) Delivers optimum speed.]||

Symbol table - It is a data structure that stores info about every symbol in a prog. [It is used by a lang. translators, like compilers | interpreters, to help the compiler determine the semantics of a source prog.]

Parse tree - It is a hierarchical representation of terminals & non-terminals. On the other words, it is an ordered rooted tree that represents the syntactic structure of a string according to some context-free grammar.

Annotated syntax tree - It is a parse tree that shows the values of the attributes at each node. It is generated after the rectification in parse tree & redrawn by the rectified tree.

Single/one pass compilers - A type of compilers that passes through the parts of each compilation unit only once, immediately translating each code section into it's final machine code.

multipass compiler - A type of compiler that processes the source code or abstract syntax tree of a prog several times.] ||

[Lex/flex - It is a computer prog that generates lexical analyzers. LA, also called tokenizers, identify lexical patterns in i/p progs & convert i/p text into a sequence of tokens. It is commonly used with the yacc parsers (lex) generators. Lex works by - i) Reading an i/p stream that specifies the lexical analyzer.
ii) Writing source code in the C programming lang. that implements the lexical analyzer.
iii) Converting the stream of characters into tokens.
iv) Dividing the i/o into meaningful units.
v) Discovering the relationships among the units.
Lex is the standard lexical analyzer generators on many Unix & Unix-like systems.

The lex tool itself is a compiler.

YACC - It stands for Yet Another Compilers Compiler. It provides a tool to produce a parser for a given grammar. [YACC is a prog. designed to compile a LALR(1) grammar. It is used to produce the source code of the syntactic analyzer of the lang. produced by LALR(1) grammar. The i/p of YACC is the rule of grammar & the o/p is a C prog.] ||
It is a grammar parser & parser generators that reads a grammar specification & generates code that can organize i/p tokens in a syntactic tree.
It helps in converting lang. rules into code that can understand & interpret complex lang. structure in langs like C, C++. It is a prog designed to compile a LALR(1) grammar. It's specification consists of 3 parts - i) YACC declarations, ii) Translation rules, iii) User-defined auxiliary Procedures.] ||

Error handeler - Error detection + error report + error recovery. In C.d, error handeling is the process of detecting & managing errors [or exceptional conditions during compilation. This process is done by error handeler.] ||

SDT engine – It is a technique of compilers execution, where the source code translation is totally conducted by the parser, is known as SDT. The parsers primarily uses a CFG to check the i/p sequence & deliver o/p for the compiler's next stage. It can also convert infix to postfix expression.

Code generator – It is the final phase of a compiler model. It takes an intermediate representation of the source code as i/p & produces an equivalent target prog as o/p. Code generation is part of a compiler's process chain. It converts the source code's intermediate representation into a form that the target system can execute. Sophisticated compilers often go through multiple passes over various intermediate forms. The i/p to the code generator is usually an abstract syntax tree or a parse tree, this tree is then converted into an intermediate lang. like three address code.

Flow engine – Data flow analysis in compiler design is a technique used in C.d. to analyze how data flows through a prog. It helps in identifying the variables that hold values at different points in the prog. & how these values change over time. It does optimization of the code.

Lexeme – It is a sequence of chars in the source text that matches a pattern for a token. They are part of the input stream from which tokens are identified.

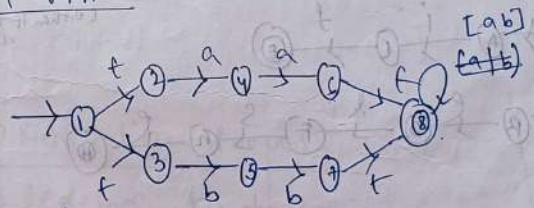
Conversion from ε-NFA to DFA

24/2/24

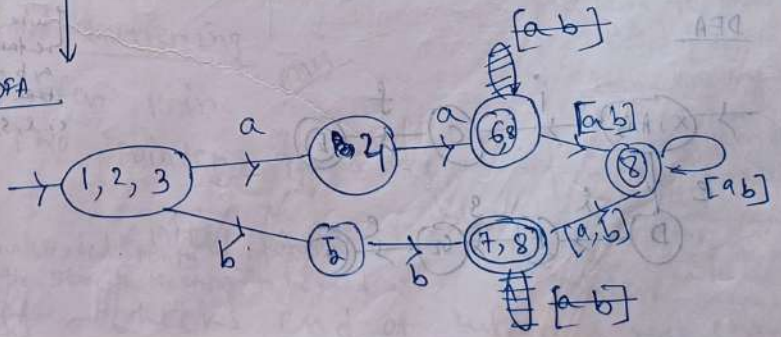
(EATON by me)

(aa|bb) (a|b)*

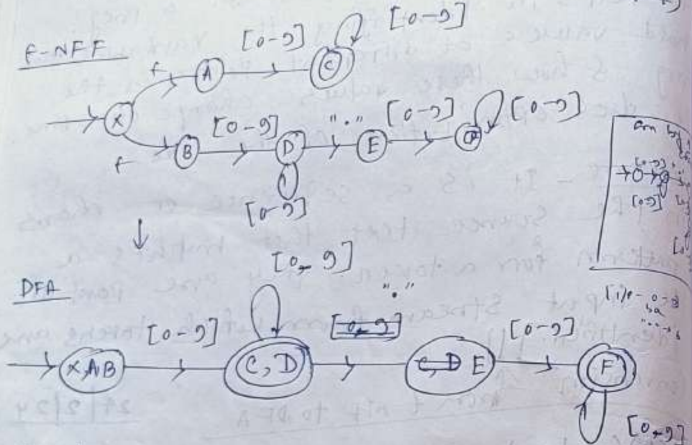
ε-NFA



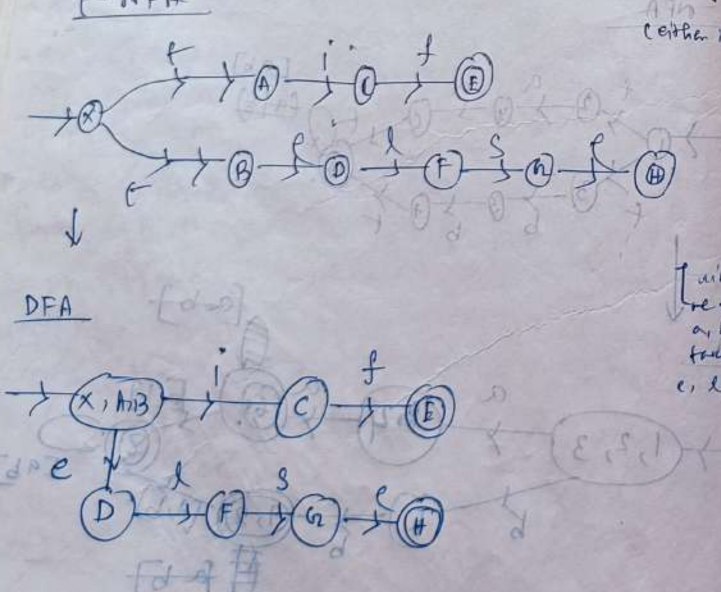
DFA



(2) Construct a DFA to identify integer constants & floating constant. (either cons. / float)

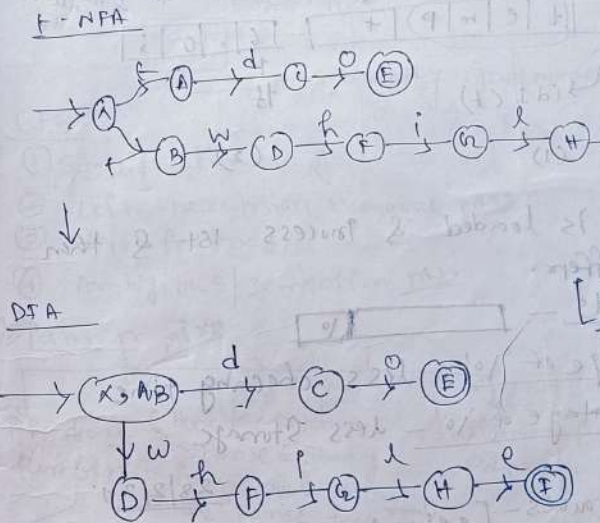


(3) Construct DFA to accept if-else keywords. (either if or else)



[since we take a, b - then take i, f, e, l, s, e]

(4) Construct DFA to accept do-while keywords.



[Every letters / alphabets are i/p here but we know from ε-NFA or can see from ε-NFA that for which i/p the state is giving o/p state]
[So, no need to think of other i/ps except the i/ps on the state for which we have to show o/ps] [If there are any common b/w 2 keywords then the rule won't applicable]

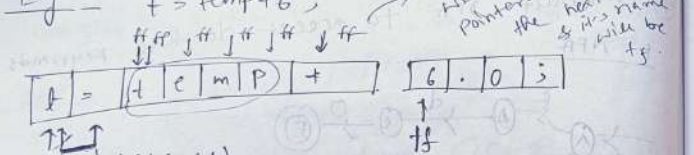
Tokenization process

↳ lexemes to tokens

Input buffering

- Buffer pair
- Two pointers f_p & f_f
  - f_p → initial pointer — a pointer that points to the start of the string to be read
  - f_f → forward pointer → a ptr that moves ahead to search for the end of a lexeme.
- ff - checks end of buffers & next char.

Eg - t = temp + 6 ;



* Buffer is loaded & process 1st & then 2nd buffer.

Sentinels - 10

Advantage of '\0' - less checking time

Disadvantage of '\0' - less Storage

28/2/24

- Regex of vowels - [aeiouAEIOU]
- Regex of consonants - [^aeiouAEIOU]* [b-df-hj-np-v-tB-DF-HJ-NP-TV-Z]

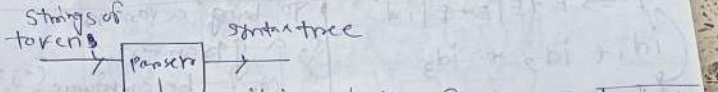
Input buffering - It is a technique that allows a compiler to read i/p in larger chunks. It improves the performance & reduce the overhead.

(will give us only 1 char that can be anything except these.)

Buffer pair - Buffer pairs are a key part of i/p buffering. They are typically implemented as 2 buffers: one for reading characters & other for buffering the lexemes. (1)

1/3/24

Parser / Syntax analysers



- in the syllabus
1. Grammer
2. Ambiguity
3. Simplification

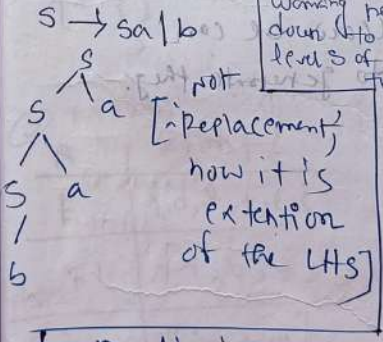
CFG

1. Simplification of g.
2. Left recursion removal
3. Left-factored
4. Ambigious / Derivation

Parser

Top-down Parsers (Here, the sentence is parsed starting from root of the parse tree, moving recursively down to the levels of the tree (terminals).

S → Sa | b



Bottom-up Parser (Here, it is an attempt to reduce the i/p string w to start symbol of g. by tracing out the rightmost derivations of w in reverse)

→ Predictive parser (LL(1) parser) or LL(k) parser

(It is a kind of top-down parser that predicts which p.r. to use to replace an i/p string.)

It is known as this as it builds a table for the string to be parsed

Algo/ ① First , ② Follow ⟶ to determine which parser can be grammar.
function (to generate the g.)

(eg) $E \to E+E \mid E*E \mid id$

$id_1 + id_2 * id_3$

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to id$
} Ambiguity removal

• To being a parser, a grammar can't be ambiguous (e.g.)

• To be LL(1) grammar a parser can't have left recursion

(eg)
$A \to A\alpha \mid B$
$A \to BA'$
$A' \to \alpha A' \mid \epsilon$
} left recursion removal

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

(this is suitable for LL(1) g)

• To do first & follow we can start from anywhere to generate the g.

• First for NT of the g.

• Follow all the NT ...

First

$A \to a(N \cup T)^*$

$A \to (N \cup T)^*$

① First(A) = {a} [first of the NT 'A' is a i.e. the 1st terminal of combination of N & T]

As 1st of any NT terminal will be always Tabulation of Ts.

First(F) = {id} [follow of starting symbol of a gram. will always have $]

②

N	First	Follow
E	{id}	{$}
T	{id}	{+,$}
F	{id}	{*,+,$}
E'	{+, ε}	{$}
T'	{*, ε}	{+,$}

First(E')
= First(+TE') ∪ First(ε)
= {+, ε}

First(T')
= F(*FT') ∪ First(ε)
= {*, ε}

$E \to TE'$ (first of T; this will be considered like this as ②)
$E' \to +TE' \mid \epsilon$ — not beneficial
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to id$

[If the first of F is the 1st T without any 'ε' then the first of F is 1st of T no need to consider the next NT]

→ If terminal only no 'ε' then applicable only

[ε is also treated as {i}, First(ε) = ε]

③ For practice only?

N	First
E	{id, *, +, ε}
E'	{+, ε}
T	{id, *, ε}
T'	{*, ε}
F	{id, ε}

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

F(E) = F(T)
F(T) = F(F)
F(F) = {id, ε}

If First (A) contains 'ε' then,

e.g, First (T) = { First (F) − ε } ∪ First (T')

= {id} ∪ {*} ∪ {ε}

= {id, *, ε}

First (E) = {id, *, +} ∪ {ε}

(or)

{id, *, +, ε} → As it is the last symbol & so getting ε

[F → ABCD
If F(A), (B), (C) X (D) contains 'ε' then F when not contain
↓
if ε is at middle (like in A, not in B then F will not contain ε)
if 'ε' is at end i.e, no 'ε' then at last by default we will get ε
Eg not 'ε' in A, B, C, D so F will contain ε]

③ S → AaB | bBC | ABb

A → (BC | dAB)

B → e | ε

C → g | ε

F(B) {e, ε}

F(C) = {g, ε}

F(A) = F(BC) ∪ F(dAB)

F(S) = F(AaB) ∪ F(bBC) ∪ F(ABb)

N ↓	First ↓
S	{b, d, e, g, a} ≈ {a, b, d, e, g} → [this g doesn't generate ε]
A	{d, e, g, ε}
B	{e, ε}
C	{g, ε}

[As we are getting 'a' hence (AaB) we need to proceed in 3rd Prod rule]

First

if: A → α(N∪T)* & α is a terminal then.

First (A) = {α}

else: A → N(N∪T)*

First (A) = First (N), if First (N) doesn't have ε.

else

{First (N) − ε} ∪ First (N∪T)*

Follow

if: A → αDβ

[α = could be NT & T]

[D = NT]

[β, & also β can contain ε too]

~~If First(B)~~

Follow(D) = First (β), if First (β) doesn't have 'ε'.

else

(i) Follow (D) = First (β) − ε

(ii) Everything of Follow (A) should be copying Follow (D), i.e Follow(D) will contain Follow(A), Follow(D) = Follow(A)

- E' → +TE'

E → TE'

} → T is upgraded

④ as, F(E) will be in Follow (E'), F(ε) = ε

i.e nothing is going to be inputted. & as ε then have to remove ε (again there will not be

anything) & samely rule (11) of else part will not give anything, otherwise we will not get anything. That is why.

~~Sentinels~~ Sentinels – They are used to making a check, each time when the forward pointer is converted, a check is completed; to provide the 1 half of the buffer has not converted off. If it is completed, then the other half should be reloaded.

5|3|24

LL(1) / LL(k) Parser

LL(k) Parser → LMD; L → left to right scan; k → look-ahead value, k is an int.

LL(1) → 1 char look ahead

eg – grammer eg ②

eg – S → saAb | b lmd ~~control~~

b a a — l to r scanning, 1 char look ahead (at a time only 1 char will be scanned)

When a grammer will be LL(1) parser

① There will be no ambiguity
② Left recursion should be removed
③ ~~tef~~ should be left factored

(min requirements / necessary requirements)

[As grammer ② satisfies those 3 conditions, so we can read it as LL(1) parser]

LL(1) parsing table

(terminal: +, *, id; $ special char)

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

[from which p.r we got the first(N) we have to write that p.r under the first of that NT]

[If there is no ε in first of NT, no need to calculate follow]

[∵ the first (E) = first (T) = first (F) = {id}]

[When there will be ε at first of any NT then we have to write the p.r for which we got 'ε' in the entry of it's all follows]

Here, there is no more than 1 elements in any entry. So we can call it LL(1) parser. i.e, this grammar is LL(1) parser.

④ In any entry there should n't be more than 1 prod. rules. (main requirement)

• If there is left recursion we will not use that grammer for this (↑) eg ② grammer as LL(1).

① E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → id

Parsing (by the help of the i/p of g.eg ② we are trying to generate the i/p°)

Stack	input (randomly taken)	Action
E$	id+id+id$	E→TE' (replace E by TE')
TE'$	id+id+id$	T→FT'
FT'E'$	id+id*id$	F→id
idT'E'$	id+id*id$	pop
T'E'$	+id*id$	T'→ε
E'$	+id*id$	E'→+TE'
+TE'$	+id*id$	pop
TE'$	id*id$	T→FT'
FT'E'$	id*id$	F→id
idT'E'$	id*id$	pop
T'E'$	*id$	T'→*FT'
*FT'E'$	*id$	pop
FT'E'$	id$	F→id
idT'E'$	id$	pop
T'E'$	$	T'→ε

[i/p not in LL(1) table then stop and the string can not be generated by this grammar]

[If both side terminal then pop]

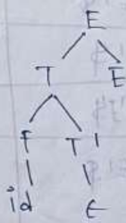
Stack	input	Action
E'$	$	E'→ε
$	$	

↓ Accepting condition

Parsing: with the help of parse table we'll see that a randomly generated i/p string is accepted or not. If it's accepted, then it can be generated by that gr.

Yes, this string can be generated by this grammar (eg ②)

Action (from a tree by seeing (scaning) it from top-bottom) →



→ LMD

• i/p = id++id

Stack	i/p	Action
E$	id++id$	E→TE'
TE'$	id++id$	T→FT'
FT'E'$	id++id$	F→id
idT'E'$	id++id$	pop
T'E'$	++id$	T'→ε
E'$	++id$	E'→+TE'
+TE'$	++id$	pop
TE'$	+id$	

No, this string can't be generated by this grammar.

• i/p - ~~id+id~~ id+*id

Stack	i/p	Action
E$	id+*id $	E→TE'
TE'$	id+*id $	T→FT'
FT'E'$	id+*id$	F→id
idT'E'$	id+*id $	Pop
T'E'$	+*id $	T'→*FT'
*FT'E'$	*+id $	Pop
FT'E'$	*id $	
no, ----		

LL(1) / LL(k) Parser - An LL(1) Parser is a top-down Parser [that uses one-token lookahead is called an LL(1) parser] if the first L indicates that the i/p is read from left to right. The 2nd L says that it produces a left-to-right derivation. [An LL(k) Parser is called an LL(k) parser if it uses k tokens of lookahead when parsing a sentence] 1 means 1 char at a time (1 is lookahead value)

Q4) First & follow calculation 12/3/24

S → ADB | DbB | Ba
A → da | BD → Another i/p
B → g | ε
D → h | ε

for all p/ps we have to get first.
In exam, formula,
first(S) = first(ADB) ∪ first(DbB) ∪ first(Ba)

	First	Follow	
S	{a,b,d,g,h,ε}	{$}	✓LL(1)
A	{d,g,h,ε}	{g, h, $}	× LL(1)
B	{g, ε}	{h, a, $, g}	
D	{h, ε}	{g, $, b, h}	

first(A) = {d} ∪ first(BD) → as f(B) has 'ε' so we have to take f(BD)
= {d} ∪ {g, h, ε}
= {d, g, h, ε}

first(S) = first(ADB) ∪ first(DbB) ∪ first(Ba)
= {f(A) - ε} ∪ f(BD) ∪ {f(D) - ε} ∪ f(bB) ∪ {f(B) - ε} ∪ f(a)
(not containing ε part) (containing ε part)
= {d, g, h} ∪ {g, h, ε} ∪ {h} ∪ {b} ∪ {g} ∪ {a}

• another approach: ABD — Rest part contains 'ε' so instead of f(A) we have to calculate f(ABD) [first(A) has 'ε' so need this]

$= \{a, b, d, g, h, t\}$

- If there is no N.T then, follow format isn't applicable, like

$S \to ADB \mid DbB \mid Ba$
$B \to g \mid t$
$D \to h \mid t$
$A \to da \mid BD$

$A \to \underset{\alpha}{} \underset{B}{B} \underset{\beta}{D}$ (ii)

$A \to \underset{\alpha}{B} \underset{B}{D} \underset{\beta}{}$ (iii) — first $(\epsilon) = \epsilon$ → But as it is 'ε' if we include it in D, it will be removed later on, so no benifit.

already done here

~~So (this) is the case should be followed~~

$S \to \underset{\alpha}{} \underset{B}{B} \underset{\beta}{a}$ no 'ε' so, no (ii) [pending task

$S \to \underset{\alpha}{A} \underset{B}{D} \underset{\beta}{B}$ (iii) | $S \to \underset{\alpha}{A} \underset{B}{B} \underset{\beta}{D}$ (ii) | $S \to \underset{\alpha}{D} \underset{B}{A} \underset{\beta}{B}$ (ii) | $S \to \underset{\alpha}{B} \underset{B}{A} \underset{\beta}{D}$ (ii)

$S \to \underset{\alpha}{B} \underset{B}{D} \underset{\beta}{A}$ (ii) | $S \to \underset{\alpha}{D} \underset{B}{B} \underset{\beta}{A}$ (ii) | $S \to \underset{\alpha}{AD} \underset{B}{B} \underset{\beta}{}$ (iv) | $S \to ADB$ (iv)

already done

$S \to \underset{\alpha}{D} \underset{B}{bB} \underset{\beta}{}$ as f(bB) has no 'ε'.

$S \to \underset{\alpha}{Db} \underset{B}{B} \underset{\beta}{}$ (this case has already done upper) So this has no benifits.

$f(bB) = \{b\} \cup \{\}$
$= \{b, g\}$

- It isn't LL(1) parsers, as in 1 cell there is more than 1 entry [trick →

(5) $S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$

will it form LL(1) parsing table?

S will never generate 'ε'.

	first	follow
S	{a, b}	{$}
A	{ε}	{a, b}
B	{ε}	{b, a}

[If it contains 'ε' then check intersection of first & follow, if no 'ε' no need to check it is LL(1) parser.]

$S \to \underset{\alpha}{} \underset{B}{A} \underset{\beta}{aAb}$ | $S \to \underset{\alpha}{Aa} \underset{B}{A} \underset{\beta}{b}$

$S \to \underset{\alpha}{} \underset{B}{B} \underset{\beta}{bBa}$ | $S \to \underset{\alpha}{Bb} \underset{B}{B} \underset{\beta}{a}$

(6) $A \to xB \mid Dy$
$B \to AzD$
$D \to zB \mid t$

$B \to \underset{\alpha}{} \underset{B}{A} \underset{\beta}{zD}$

$B \to \underset{\alpha}{Az} \underset{B}{D} \underset{\beta}{}$ (ii)

$D \to \underset{\alpha}{z} \underset{B}{B} \underset{\beta}{}$ (iii)

$A \to \underset{\alpha}{x} \underset{B}{B} \underset{\beta}{}$ (iv) | $A \to \underset{\alpha}{} \underset{B}{D} \underset{\beta}{y}$

	fi	fo	
A	{x, z, t}	{$, z}	✓ LL(1)
B	{x, z, t}	{$, z, y}	✓ this can't be LL(1) parser
D	{z, t}	{y, $, z}	✗ [∩ = z]

First (α) ∩ follow (α) ≠ ϕ then LL(1) parsers (if ε occurs) — same → do first(α) ∩ follow(α) if there is ε in first

If there is no ε in first(α) it will be definately LL(1) parsers but if ε then Do this

HW Draw Parsing table for (5) & (6)

(4) Parsing table,

	a	b	d	g	h	$
S	S→Ba	S→DbB	S→ABD	S→ABD	S→ABD	S→ABD
A			A→da	A→BD A→BD	A→BD A→BD	A→BD
B						
D						

more than 1 entry in cell → so no need to proceed, it can't be LL(1) parser

(5) Parsing table,

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

(the grammar) → it is a LL(1) parser

(6) Parsing table,

	x	y	z	$
A	A→xB	A→Dy	A→Dy	
B	B→AzD	B→AzD	B→AzD	
D		D→ε	D→zB D→ε	D→ε

→ this grammar isn't a LL(1) parser — more than 1 entry

Cw 15/3/24

Bottom-up parsers

- Operator Precedence Parser
- Shift-reduce parser (LR(k) parser)
  - → LR(0)
  - → SLR (simple LR) / SLR(0)
  - → CLR(1) canonical
  - → LALR(1) LR) (look ahead LR)

[Parsers = syntax analyzer]

• LR(k): L — left to right scan; R — RMD; k — look-ahead value

Closure of items

Imp Necessary Conditions for LR(1) [atleast what are needed] (but it means not that only that is needed)

1) There will be no ambiguity

(1) E → E+T|T
(3) T → T*F|F
(5) F → id

← (Augmented rule) [For the starting symbol we are using an extra symbol, it will just change the start symbol]

I_0 = E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .id

[place dot infront of NT & write a new its all p.r. & give infront them & repeat]

LR(0) item sets Imp

• goto( ) ⟶ It is a func that will help to transmit all prod. rules of I_0. It's func is to decide when to shift & when to reduce.

I_1 = goto(I_0, E) ⟶ means in I_0 whenever there is a '.' before that T/NT in goto that '.' will be shifted / moved after that T/NT & we have to write that new p.p.

E' → E.
E → E.+T

[If there is any '.' before NT here, so we have to do like 1st (write all p.p. of that NT & give inform of that NT]

I_2 = goto(I_0, T)

E → T.
T → T.*F

I_3 = goto(I_0, F)

T → F.

[+ calculate for all NTs before that there is '.']

all items of I_0 are calculated

I_4 = goto(I_0, id)

F → id.

It will continue until the pos of '.'s when not reach to the end of all p.p.

I_5 = goto(I_1, +)

E → E+.T
F → .id
T → .T*F
T → .F

I_6 = goto(I_2, *)

T → T*.F
F → .id

I_7 = goto(I_5, T) = E → E+T.
T → T.*F

I_3 = goto(I_5, F) ⟶ [T → F.]

I_4 = goto(I_5, id) ⟶ [F → id.]

I_8 = goto(I_6, F)

T → T*F.

I_4 = goto(I_6, .id) → [F → id.]

I_6 = goto(I_7, *) → [T → T*.F, F → .id]

It generates total 8 elements.

If '.' before + giving '.' information of all NTs in that step is already finished then stop

If a new item should write that ... as already exists so giving the existing name otherwise it will continue infinitely

It is end, & we can understand that it is end as it is generating no new elements.

	First	Follow
E	{id}	{$, +}
T	{id}	{*, $, +}
F	{id}	{*, $}

It is sticking on it's own value F(T) = F(T) we don't know

↓ To overcome this

Consider T → T*F | F

[Operator precedence parser] - It is a bottom-up parser that interprets an operator-precedence grammer. [It is a shift reduce parser that can be easily constructed by hand.] The grammers class of from which it can be constructed is called operator grammer. It is used to convert from human-readable infix notation, which relies on order of operations, to a format that is optimized for evaluation, such as reverse polish notation.] ||

shift-reduce parser - It is a table-driven, bottom-up parsing method for computer languages & other notations defined by a grammer. It is also known as bottom up parsing.] ||

LR(1)/LR(k)/LR(0) parser - It is a type of bottom-up parser that uses left-to-right scanning, rightmost derivation (in reverse) & the no. of unconsumed "lookaheads" i/p symbols to make parsing decisions.

[SLR(1) parser - It stands for simple LR parser. It is a bottom-up parser that can be used to parse large classes of CFG. It is same as LR(0) parsing. The only difference is in parsing table.] || To construct SLR(1) parsing table, we use canonical collection of LR(0) item.] ||

(CLR(1) parsing parser - CLR refers to canonical LR. [CLR parsing use the canonical collection of LR(1) items to build the CLR(1) parsing table. [It CLR(1) parsing table produces the more no. of states as compare to the SLR(1) parsing.] ||

LALR(1) parser - It stands for lookahead LR. to construct the LALR(1) parsing table, we use the canonical collection of LR(1) items. It is a bottom-up parser used in compiler design.] ||

10/3/24

LR(1) Parsing table formation

SLR(1) table

[ACTION & GOTO should be written in col & in row there will be no. of items (LR(1) P.T)]

Items↓	+	*	id	$	E	T	F
0			S4		1	2	3
1	S5			accept			
2	r2	S6		r2			
3	r4	r4		r4			
4	r5	r5		r5			
5			S4			7	3
6			S4				8
7	r1	S6		r1			
8	r3	r3		r3			

[S4 → shift 4 i/p. then have to add S before no.]

[r → reduce]

I_{p_2} = goto(I_n, a)
Entry ↓ now ↘ cal
to the

if NT = only no.
if T = S no.

} → shift

Eg, I_8 = goto(I_9, T)
$E \to T.$
$T \to T. * F$
} → reduce

row no. ← (No. of this Prod. rule ②)

∴ p2 will be written in the fallow of E

∵ Accept = in item 1, P.P has · at the last of P.P
∵ $E \to T.$ (in ②)
∴ row = 2
cal = E's fallow (Always in LHS's NT's fallow)
& as the no. of P.P is ② so r2

It is LR(1) [or more clearly SLR(1)] parsers as no cell is containing more than 1 entries. (As here there is no conflicts, so it is LR(1) parser.) [more clearly it is SLR(1) parser] 27/3/24

Conflicts
shift - shift ×
s - r - can possible
r-r - " in LR(1) pars. ta.

- Proper reg ex of identifier =
[-\t\n, ; ( ) { } = > <]* [a-z A-Z 0-9 _]+ [\t \n , ; ( ) { } = > <]+

But we don't do so (as we can't define seperators perfectly) [If there is any not expected thing in o/p, then this is the cause, & we have to accept it like this]

- LR(0) → parsing table, we can construct, but that is ambiguous.

Parsing (for LR(1))
SLR(1) parsing

(shift = 1 step
Reduce = 2 step (Replacement pop tos & now which is tos check the relation (actions again))

Stack (State)	Stack	i/p	
\$0 ← tos	\$	id + id * id \$	S4 (shift 4)
\$04 ← tos	\$ id (shift)	+ id * id \$	Reduce $F \to id$ [∴ $F \to id$ (5)]
\$03 (tos deleted)	\$F (Replace RHS with LHS)	+ id * id \$	Reduce $T \to F$
\$02	\$T	+ id * id \$	Reduce $E \to T$
\$01	\$E	+ id * id \$	S5 (shift 5)
\$015	\$E +	id * id \$	shift 4
\$0154	\$E + id	* id \$	Reduce $F \to id$
\$0153	\$E + F	* id \$	Reduce $T \to F$
\$0157	\$E + T	* id \$	shift 6
\$01576	\$E + T *	id \$	shift 4
\$015764	\$E + T * id	\$	reduce $F \to id$ (1 item to 1 item remove)
\$015768	\$E + T * F	\$	reduce $T \to T * F$ (3 items reduction so from tos 3 items will be removed)
\$0157	\$E + T	\$	Reduce $E \to E + T$
\$01	\$E	\$	accept or ACCEPT

- If got stuck at any point, then Rejected.

- It is forming Rmb tree in reverse [LR → Rmb in reverse]

i/p: id++id $ i/p = id++id

- We can't construct SLR(1) table from LR(1) table.
- By the items LR(0) → LR(0) / SLR(1) parsing table
- By the items LR(1) → CLR(1) / LALR(1) parsing table

→ i/p = id++id

SLR(1) Parsing

Stack(State)	Stack	i/p	Actions
$0	$	id++id$	shift 4
$04	$id	++id$	reduce F → id
$03	$F	++id$	reduce T → F
$02	$T	++id$	shift 6
$026	$T+	+id$	not accepted

∴ This string can't be generated by this grammer.

i/p = id++id

SLR(1) Parsing

Stack(State)	Stack	i/p	Actions
$0	$	id++id$	shift 4
$04	$id	++id$	reduce F → id
$03	$F	++id$	reduce T → F
$02	$T	++id$	shift 6
$026	$T+	+id$	not accepted
$01	$E	++id$	shift 5
$015	$E+	+id$	not accepted

∴ not stucked, so this i/p is rejected i.e this string can't be generated by this grammer

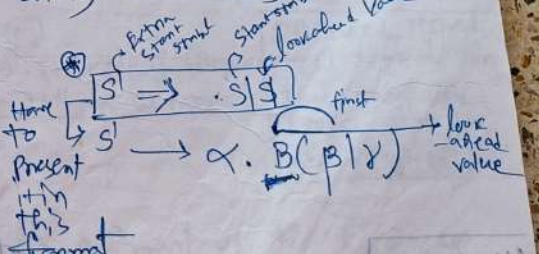
16/4/24

CLR(1)

It will form from LR(1) item sets.

LR(1) item sets

(1) F → F++
E → T
T → T*F
T → F
F → (E)
F → id



'I' → OR

Augmented rule

I_0: E' → .E | $
E → .E+T | $
E → .T | $
T → .T * F | $
T → .F | $
F → .id | $

(as we are writing this first for the 1st one (i.e.) as there $ is lookahead so in here we have to give it there too.) (*) (Clear reasons)

I_1 = goto(I_0, E)
E' → E. | $
E → E.+T | $

It is just shifting, no lookahead calculation is here ($ = prev lookahead)

It generates total 8 elements (like previous one).

new pr.s must have to write (don't think that same pr.s are there in another item) [Eg. I_5, I_6, I_7] & have to do cal. for them too. Although they will give same elems
eg - (2) PTO

I_2 = goto(I_0, T)
E' → T. | $
T → T.*F | $

I_3 = goto(I_0, F)
T → F. | $

I_4 = goto(I_0, id)
F → id. | $

I_5 = goto(I_1, +)
E → E+.T | $
T → .T*F | $
T → .F | $
F → .id | $

I_6 = goto(I_2, *)
T → T*.F | $
F → .id | $

(*) B> nothing first of this $ so we are writing $ (correct it)

I_8 = goto(I_6, F)
T → T*F. | $

I_4 = goto(I_6, id)

I_7 = goto(I_5, T)
E → E+T. | $
T → T.*F | $

I_4 = goto(I_5, id)

I_3 = goto(I_5, F)

I_6 = goto(I_7, *)

If you calc lookahead here, then the lookah. of I_0 will be matched with it (so you can directly take look. from I_0/ conical here)
But better is always calc. here
(crosscheck)

the same of (I_1,+) exists so 'i' E → E.+T | $ & the further steps are for ∵ infront NT

After ∵ more than 2 things (NT/T) then calc Lh (may be $, or may be something else) (*) this format

In general (E → .E+T | $) is shifting so, both looka. will be same.

If need to cal. new looka., in that case the looka. will be new (eg - (2)) PTO

Clear reasons
(*) For 1st, E' → .E | $ ← This lookahead value is default. Here, as '.' is infront of 'E' so we have to write all the prs of E. In there the lookahead will be depended on the lookahead of that pr for which we had to write all prs of E (i.e. E' → .E | $) & like that the further process will continue.

Assignment

(1) S → AA — (1)
A → aA — (2)
A → b — (3)

If nothing in first → $
if anything = no need of $

I0:
S' → .S | $
S → .AA | $
A → .aA | a, b
A → .b | a, b

lookahead value

I1 = goto(I0, S) → S' → S. | $
I2 = goto(I0, A) → S → A.A | $
A → .aA | $
A → .b | $
I3 = goto(I0, a) → A → a.A | a, b
A → .aA | a, b
A → .b | a, b
I4 = goto(I0, b) → A → b. | a, b

I3 = goto(I3, a) → A → a.A | a, b
A → .aA | a, b
A → .b | a, b
I4 = goto(I3, b) → A → b. | a, b
I5 = goto(I2, A) → S → AA. | $
I6 = goto(I2, a) → A → a.A | $
A → .aA | $
A → .b | $
I7 = goto(I2, b) → A → b. | $
I6 = goto(I3, A) → A → aA. | a, b
I9 = goto(I6, A) → A → aA. | $
I6 = goto(I6, a) → A → a.A | $
A → .aA | $
A → .b | $
I7 = goto(I6, b) → A → b. | $

It generates 10 items / items total

(this is recursive, the 'dot' will not reach the end)

CLR(1) parsing table

Items	a	b	$	S	A
0	S3	S4		1	2
1			accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

- LR(0) = no lh value
- LR(1) = lh values
- no. of items SLR(1) < CLR(1) [Because of lookahead value] not for all time, but it can happen.

↳ this is (CLR(1) Parser (no conflict in cells)

(no fits for val cols here, write below here, from A → b. | a, b (I4))

LALR(1) → somehow an extension of CLR(1)
Same prods, just lh val different → merge them (in LALR)

I3 & I6 ⇒ I3,
A → a.A | a, b, $
A → .aA | a, b, $
A → .b | a, b, $

I_4 & $I_7 \Rightarrow I_4$ (A → b. ,b/$)

I_8 & $I_5 \Rightarrow I_8$, (A → aA. /a,b,$)

(more than 2 elements items, merging is also possible)

- No of states
  SLR(1) = LALR(1)
- SLR(1) & LALR(1) acts samely, just in LALR(1) there is no need to calc. fi & fo.
- CLR ≥ (SLR = LALR) (from the point of 'no. of states')

viva
- S–S conflict → doesn't exist
- S–R conflict
- R–R conflict

LALR(1) Parsing table

I	a	b	$	S	A
0	s3	s4		1	2
			accept		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

It generally shown in LALR, but it can also consist in CLR & SLR

It is common, it can come in LALR, CLR, SLR (all generally)

Parsers

Top down
S → input string

Predictive parsers
LL(1) / LL(k) parsers

Bottom up
input string → S

Operator Precedence parsers

shift reduce parsers
- LR(0) parsers (from this LR(0))
- SLR(1) s
- CLR(1) s
- LALR(1) s

(exam)

reduction is written below T always because T are replaced to get a NT (reducted)

- For LR(0) parsing table, if A → b. , then write the rule no. (for reduction) below every T.s, So, ~~there is more~~ (chances of happening conflicts are more here. So it isn't used / less imp.

Operators Precedence parsers

Operators grammer

Consecutive NTs are not possible here. there must have to be a T b/w NTs, this is operators grammers.

SS → ABC ✗
B → +|-|+
S → A+C | A-C | A+C (operators grammers)

pg 23

O.P. P table / operation relationship table

LHS \ RHS	+	-	*	/	id	$
+	·> (from the associativity)	·>	<· (less than)	<·	<·	·>
-	·>	·>	<·	<·	<·	·>
*	·>	·>	·>	·>	<·	·>
/	·>	·>	·>	<·	<·	·>
id	·>	·>	·>	·>	—	·>
$	<·	<·	<·	<·	<·	Accept

LHS is compared with RHS

LHS 1st then RHS

the higher precedence here old lower => $

p.o.v => point of view

+, -, /, *
L → R
associativity
/ = p → L

pg 23

(1) Grammar

E → E+E | E*E | id

Can 'id+id' be generated from this g.?

(i/p)

O.P. parsing

Stack	I/P	Action
$	id+id$	S
$id >	+id$	~~shift~~ R
$E <	+id$	~~Reduction~~ S
$E+ <	id$	~~shift~~ S
$E+id >	$	~~shift~~ R
$E+E >	$	Reduction
$E	$	Accept

[if LHS < RHS ⇒ shift else Reduce]

23/4/24

(3) S → L=R —(1)

S → R —(2)

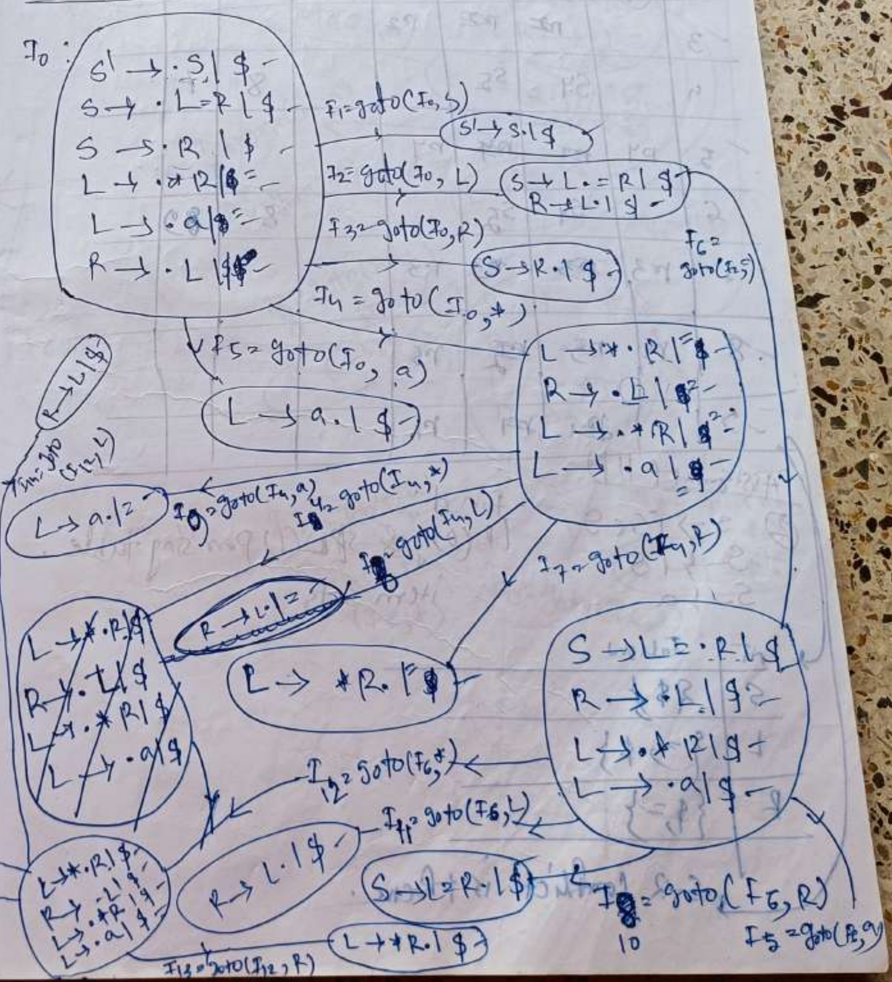
L → *R —(3)

L → a —(4)

R → L —(5)

Not this → PTO

. LR(1) item sets



SLR(1) parsing table

items	=	*	a	$	S	L	R
0		s4	s5		1	2	3
1				acpt			
2	s6 r5	r5	r5	r5			
3		r2	r2	r2			
4		s4	s5			8	7
5	r4	r4	r4	r4			
6		s4	s5			8	89
7	r3	r3	r3	r3			
8	r5	r5	r5	r5			
9		r1	r1				

Assignment (4+5)

(2) S → i S e S
S → i S
S → a

Find LR(0) & SLR(1) parsing table.
item sets

NT	Follow 2
S	{$}
L	{=, $}
R	{$, =}

S-R conflict is there.

LR(0) item sets

I_0: S' → .S
S → .L = R
S → .R
L → .* R
L → .a
R → .L

I_1 = goto(I_0, S) : S' → S.

I_2 = goto(I_0, L) : S → L.= R, R → L.

I_3 = goto(I_0, R) : S → R.

I_4 = goto(I_0, *) : L → *.R, R → .L, L → .* R, L → .a

I_5 = goto(I_0, a) : L → a.

I_6 = goto(I_2, =) : S → L =.R, R → .L, L → .* R, L → .a

I_7 = goto(I_4, R) : L → *R.

I_8 = goto(I_4, L) : R → L.

I_9 = goto(I_6, R) : S → L = R.

I_5 = goto(I_6, a)

I_4 = goto(I_6, *)

I_8 = goto(I_6, L)

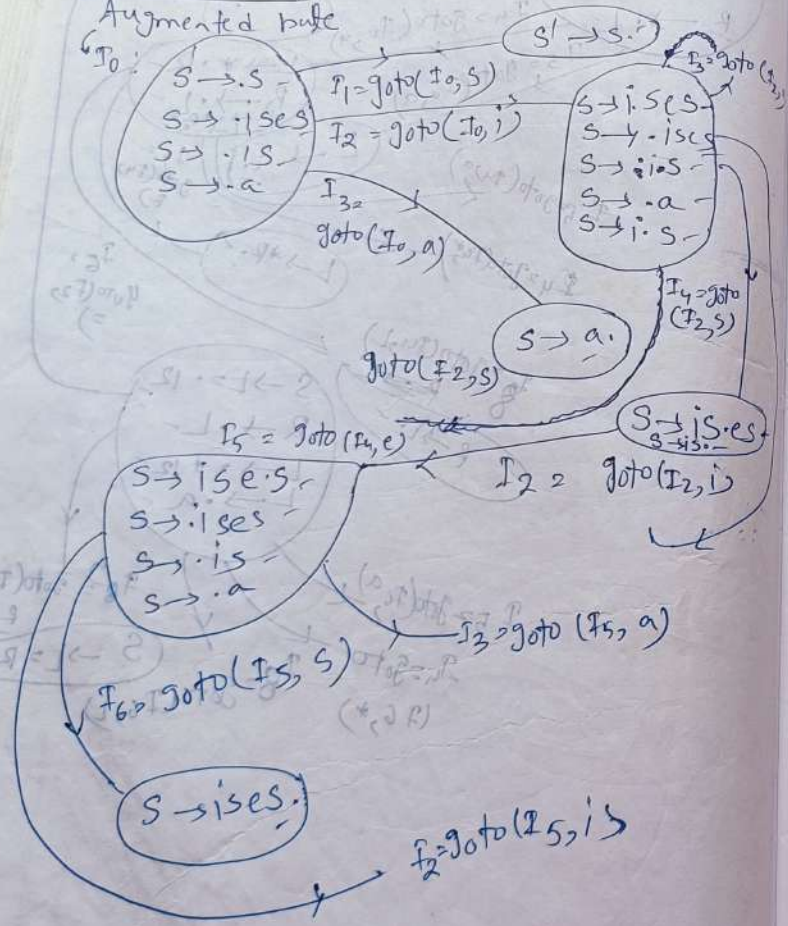
Assignment

(2) S → i ses — (1)
S → ies — (2)
S → a — (3)

Find: LR(0) & SLR(1) Parsing table.

LR(0) item sets — item sets

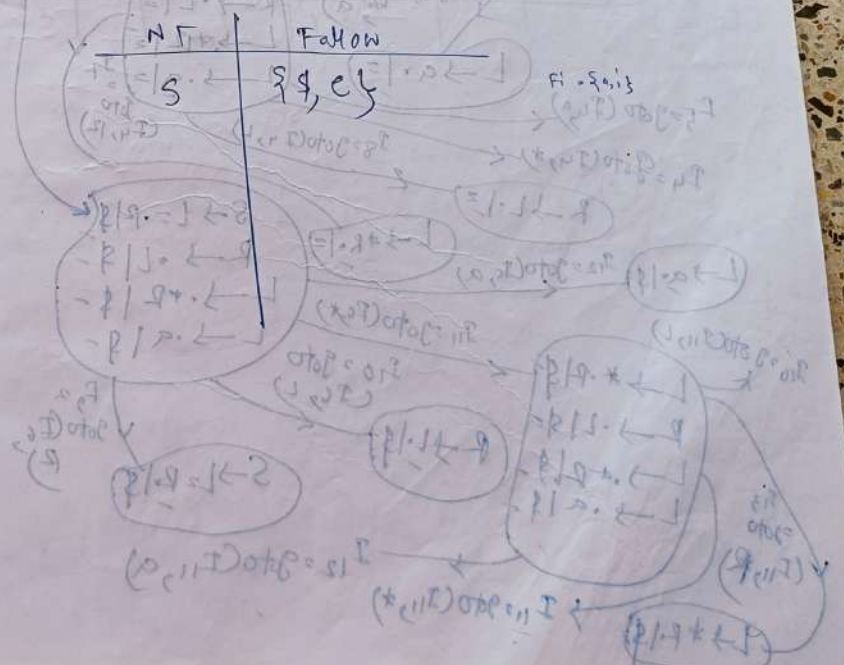
Augmented rule



SLR(1) Parsing table

item	i	e	a	$	S
0	s2		s3		1
1				Accept	
2	s2		s3		4
3		r3		r3	
4		s5 / r2		r2	
5	s2		s3		6
6		r1		r1	

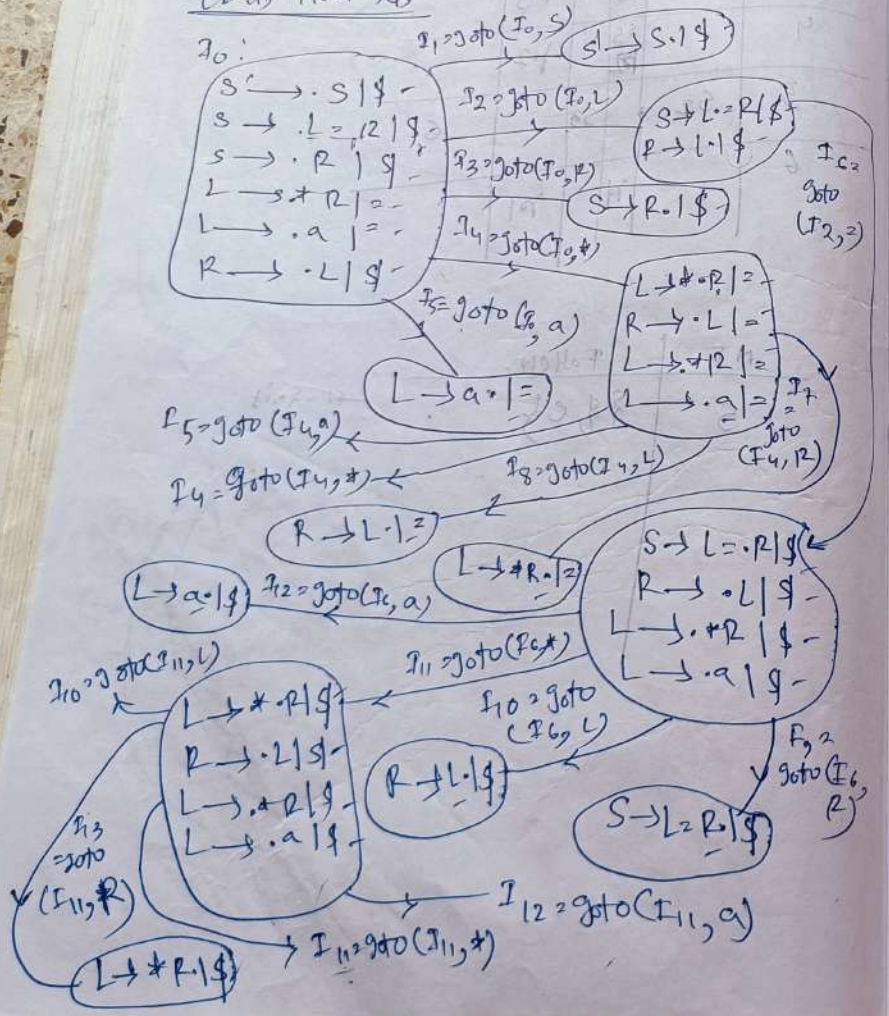
NT	Follow
S	{$, e}



③ LR(1) item sets & SLR CLR (1) parsing table

$S \rightarrow L = R$ — ①
$S \rightarrow R$ — ②
$L \rightarrow *R$ — ③
$L \rightarrow a$ — ④
$R \rightarrow L$ — ⑤

LR(1) item sets



CLR(1) parsing table

State	=	*	a	$	S	L	R
0		S4	S5		1	2	3
1				accept			
2	S6			r5			
3				r2			
4		S4	S5			8	7
5	r4						
6		S11	S12			10	9
7	r3						
8	r5						
9				r1			
10				r5			
11		S11	S12			10	13
12				r4			
13				r3			

24/4/24

CD

Semantic analysis

tin x; ⟶ syntax error
identifier
(at 1st stage no errors
L.A ⟶ generates meaning for words (so no error in 1st stage), it will be in syntax error)

int ion; x

Parser

tokens → [Parser] → syntax tree / derivation tree

I are waiting ⟶ no syntax errors
But it is semantic error (error in meaning).

- SDD & SDT.
  - SDD ↓ syntax directed defination
  - SDT ⟶ syntax directed translation

SDD ⇓
(CFG + Defination/meaning → (A CFG with attributes & rules)
(grammer) (meaning) ⟶ attributes

eg

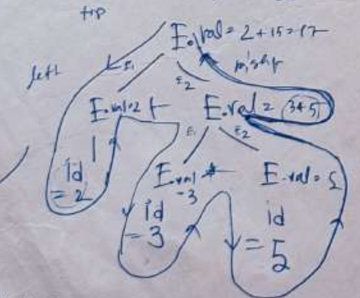
Defination (MR23)

$E \rightarrow E_1 + E_2$ { E_1.val = E_1.val + E_2.val; }
$E \rightarrow E_1 * E_2$ { E.val = E_1.val * E_2.val; }
$E \rightarrow id$ { E.val = id.lexval; }

(It gives meaning to a grammer)
SDD (context free grammer with it's defi (it's semantic) (m) (collection of P.R.S with some meaning)

If you don't wanna give defination then just give { } otherwise it's not SDD (formatis it following)

2 + 3 * 5

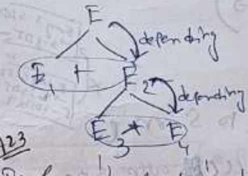


Starting from left bottom & then up

- Attributes – Attributes are characteristics that determine the value of a grammatical symbol. Eg = type, value etc.

SDD ⟶ synthesised attribute } diff (MR23)
⟶ Inherited attribute

(MR23) Synthesised attribute – If any attribute is depending upon it's self or it's children, then it is called synthesised attribute.



E → E_1 + E_2
dependent ⟶ right things upon

(MR23) Inherited attribute – ... itself / it's parents / siblings ... inherited attribute.

E ⟶ E_1 + E_2
← depends on left things in it's prod. rules

(MR23) Eg

① A ⟶ BDC { D.type = A.type }
inherited (D's type depends upon it's left)

A ⟶ BDC { A.type = D.type }
synthesised (A's type depends upon it's right)

② float a, b;
depends (inherited)

SDT = It is Collection of SDD

$A \rightarrow BDC\{A.type = D.type;\}$ synthesised SDT

2 types

→ S-Attributed SDT / defination

→ L-attributed SDT / defination

S-Attributed SDT – only synthesised SDDS are written here / the concept where only synthesised SDDS are written there, it is S-attributed SDT. (It uses only synthesised attributes)

Eg

E → E+E

E → E*E

E → id

L-attributed SDT / defination – Either S or L, but only left-side inheritance. (All S-attri. SDTs are L-attri. SDT, but reverse isn't correct)

$A \rightarrow BDC\{D.type = C.type;\}$ → not L-attri. SDT

inherited but not left-side. (It uses either syn or i. but only L inheritance)

Eg

$E \rightarrow E + T \mid T$ (left recursion removal)

$T \rightarrow T * F \mid F$

$F \rightarrow id$

$E \rightarrow T A \{A.inh = T.val; B.val = T.val;\}$

$A \rightarrow + T A_1 \{A_1.inh = T.val + A.inh; A.val = A_1.val;\}$

$A \rightarrow \epsilon \{A.val = A.inh;\}$

$T \rightarrow F B \{B.inh = F.val; T.val = B.val;\}$

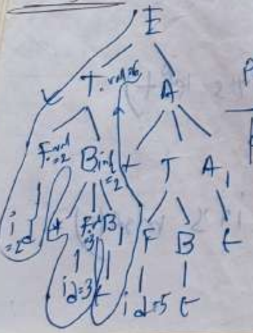
$B \rightarrow * F B_1 \{B_1.inh = F.val * B.val; B.val = B_1.val;\}$

$B \rightarrow \epsilon \{B.inh = B.val;\}$

$F \rightarrow id \{F.val = id.lexval;\}$

(L-inherited defination)

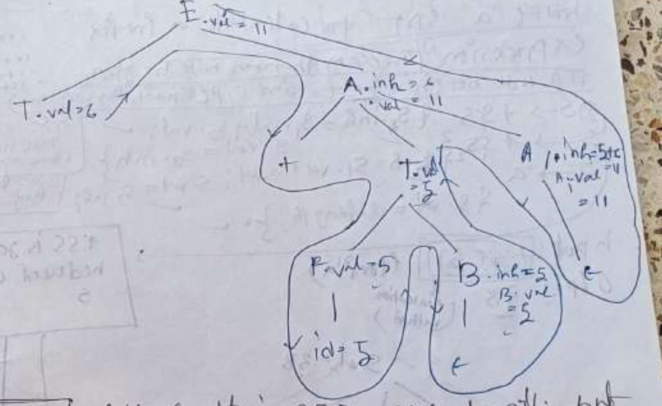
① 2+3*5



$B_1 = 3*4$ $B.inh = 3*2 = 6$

B.inh = 2

B.val = 6



Shortcut

2 + 3 + 5

+ (+ 2 3) 5

⇓

11 (cross check)

• All S-attri. SDTs are L-att., but the rev isn't correct – An L-att. SDT can be either S or I with the restriction on that the inherited attri. can only inherit vals from L siblings. But S-att. can have only S-SDDS. So, the statement is true. 26/4/24

C/W

① $A \rightarrow BCD\ \{B.i = A.i,\ D.i = C.i\}$ (inh, inh) → × (R-hand inherited)

not L / S-attributed

② $A \rightarrow BCD\ \{B.i = A.i,\ A.type = C.type\}$ (inh, syn)

↓

L-attributed

Application 5 of SDT

SDT

① Arithmetic expression calculation: 2+3*5=17

② Calculating result of Prefix expression = +3*57

③ — " — — — — Postfix " - 357*+

Write a SDT to calculate a Prefix expression (in exam the grammar will be given i/p will be given, just find o/p (through parsing tree))

① $S \to +SS$ { $S.inh = S_1.val + S_2.val$; $S.val = S_2.inh$ }

$S \to *SS$ { $S.inh = S_1.val * S_2.val$; $S.val = S_2.inh$ }

$S \to a$ { $S.val = a.length$; }

input: +3*57 (prefix)

o/p: 38 (evaluation method)

(Annotated Parse tree)

③ Write a SDT for Post fix exp (do same)

i/p = 357*+ (Post fix)

o/p = 38

(evaluation method)

not this

(fresh PTO)

④ Type checking

$S \to AB;$ { $B.inh = A.type$; } (inh but L-handed inh)

$A \to int$ { $A.type = int$; }

$A \to float$ { $A.type = float$; }

$B \to id$ { addtype(id.entry, B.inh); }

$B \to B_1, id$ { $B_1.inh = B.inh$; addtype(id.entry, $B_1.inh$); }

(like int x, y, z;)

B_1.inh will be same val, and that val's type will be B_1's type

(not this)

for postfix exp code.

i/p = 3 5 7 + +

o/p = 38

SDT

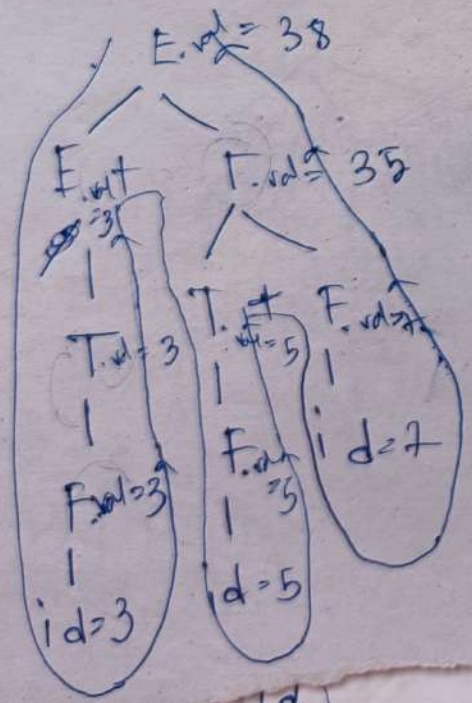
$E \rightarrow E_1 + T \quad \{E.val = E_1.val + T.val;\}$

$E \rightarrow T \quad \{E.val = T.val;\}$

$T \rightarrow T_1 * F \quad \{T.val = T_1.val * F.val;\}$

$T \rightarrow F \quad \{T.val = F.val;\}$

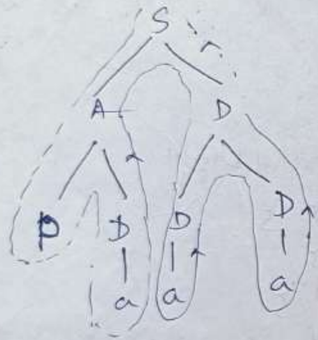
$F \rightarrow id \quad \{F.val = id.lexval;\}$



Imp (exam) pt 2

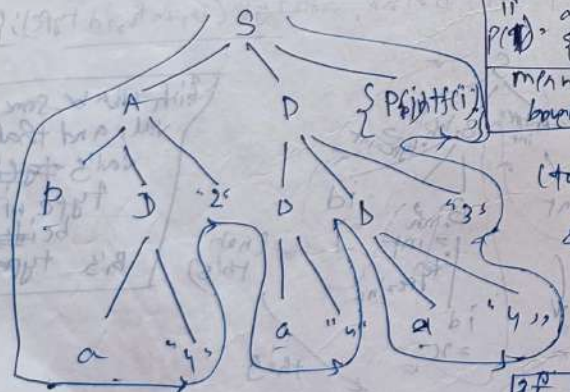
(5) S → A D { Print ("1"); }
A → P D { Printf ("2"); }
D → D D { Printf ("3"); }
D → a { Printf ("4"); }

i/P = Paaa
O/P = ? (what will be the o/p using this SDT)



o/p = 424431

(Bottom up approach)



"1" P = Printf
'1' actual syntax
P(1) = { Printf("1"); }
meaning is a branch too

(top down approach & L → R)

O/P = 424431

if L-inherited then topdown approach is best as inh. val. needs the result of parents & topdown is starting from parents

- Syn. → bottom up parsing (best)
- Inh. → top down " (best)

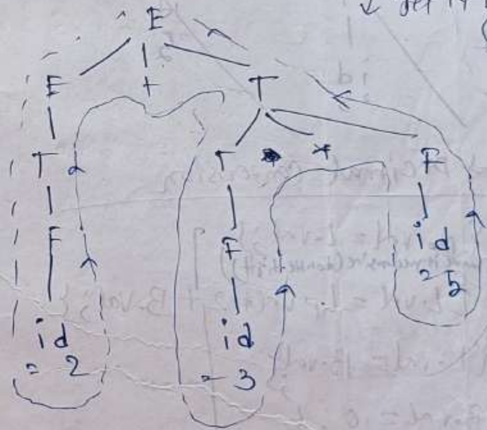
(5) Infix to Postfix conversion
input: 2+3*5
o/p: (2+35* →) 235*+ (expected)

E → E+T { Printf ("+"); }
E → T { }
T → T*F { Printf ("*"); }
T → F { }
F → id { printf (id.lexval); }

{ } = do nothing

Bottom up approach

get it by parsing



O/P = 235*+

(6) infix to Prefix (↑) [do the same] conversion
input: 2+(3*5)
O/P: 2+(*35) → +2*35 (expected)

E → { Printf ("+"); } E + T
E → { } T

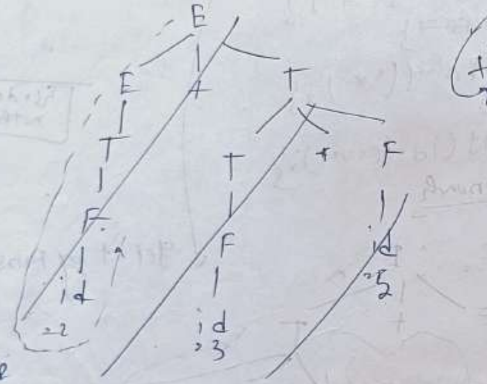
Doing this to make the grammar a prefix from this
Conv CG. will be given in exam

T → { Printf('+'); } T F

T → { } F

F → { Printf(" id.lexval) } id

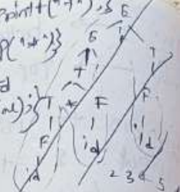
~~Bottom up~~ Topdown approach



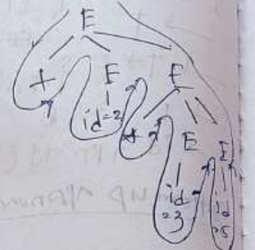
E → E₁ F₂ { Printf("+"); }

F → id { P(id.lexval); }

Efresh (exx)



o/p = +2*35



⑥ Binary → Decimal conversion

N → L { N.val = L.val; }
(to make it recursive (double digit))

L → L₁B { L.val = L₁.val*2 + B.val; }

L → B { L.val = B.val; }

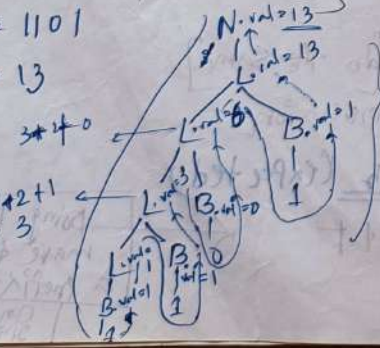
B → 0 { B.val = 0; }

B → 1 { B.val = 1; }

(for all need to read)

i/p = 1101

o/p = 13



(understanding part)

1101
(((1*2+1)*2+0)*2+1)
(1+2)+1 = 3
3*2+0 = 6
6*2+1 = 13 (13)

HW Prob: Count binary digits

i) count the calc. the length of binary val^n (same g.)

ii) no. of 1's & 0's calculate^n (same g.)

① N → L { N.count = L.count; }

L → L₁B { L.count = L₁.count + B.count; }

L → B { L.count = B.count; }

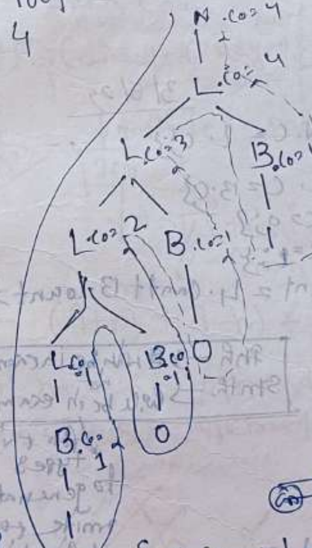
B → 0 { B.count = 1; }

B → 1 { B.count = 1; }

(given) (written)

i/p = 1001

o/p = 4



{ N.ct_1 = L.ct_1; N.ct_0 = L.ct_0; }

Prob

⑪ N → L { Print(L.count_1, L.count_0); }

L → L₁B { L.count_1 = L₁.count_1 + B.count_1; L.count_0 = L₁.count_0 + B.count_0; }

L → B { L.count_1 = B.count_1; L.count_0 = B.count_0; }

B → 0 { B.count_1 = 0; B.count_0 = 1; }

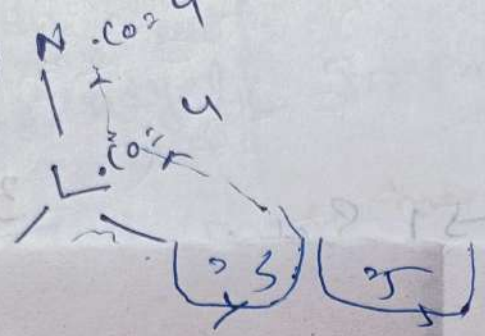
B → 1 { B.count_1 = 1; B.count_0 = 0; }

For in → POST → Evaluation, Conversion } for Both grammers same, defi diff. as both parsing is bottom up parsing

For in → Pre → Evaluation, Conversion } for both grammers & def. both are diff. as evaluation is bottom up parsing & conversion is top down parsing

i/p = 1001

o/p = 4



$S \rightarrow \{Printf('+')\}\ S_1\ S_2$

$S \rightarrow \{printf('*')\}\ S_1\ S_2$

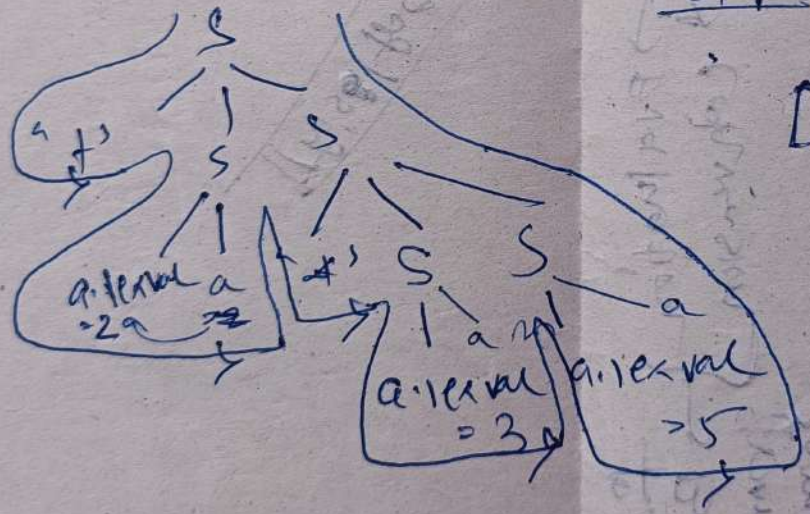
$S \rightarrow \{printf(a.lexval);\}\ a$

i/p → 2+3

o/p → + 2 * 3 5

['+' → P(+)
'*' → P(*)]

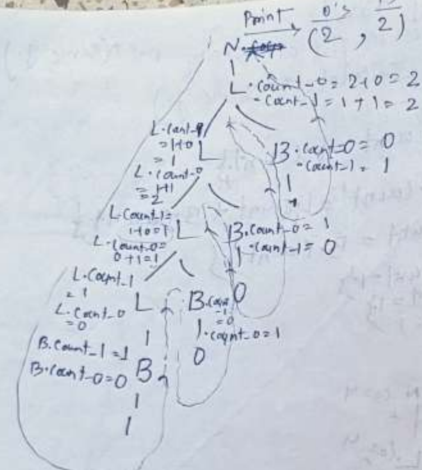
Both top-down



$L \rightarrow$

$B \rightarrow 0$

$B \rightarrow 1\ \{B.count = 1$

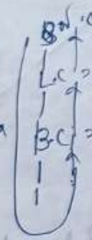


3/5/24

Q: 1101 ⇒ 13

N → L {N.C = L.C;}
L → B {L.C = B.C;}
B → 0 {B.C = 0;}
B → 1 {B.C = 1;}
L → LB {L.count = L.count + B.count;}

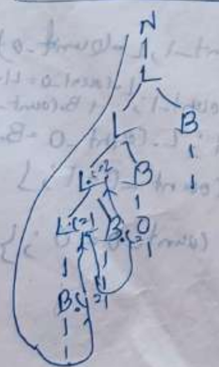
single digit



Inh → will be in exam
Synth. → will not be in exam

↓ for this types to generate / write the definations

more than single digit



(Anotated parse tree / Syntax tree)
↓ no need, same as prev)

---

(4) Intermediate Code generator – m/c independent Code

(5) Code generator – m/c dependent

(6) Code optimization

(4) Intermediate co. g.

The code is of assembly type.

Way to write the code

(1) AST (Abstract syntax tree) → will be generated from grammer semantic rules

(2) 3-address code

(3) DAG (directed acyclic graph)

(1) AST P123

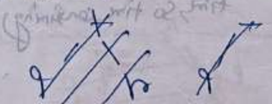
↓ Anotated parse tree
ICG
↓ AST (this is also a syntax tree, generated from G + S. R. but it's abstract)
↓ DAG

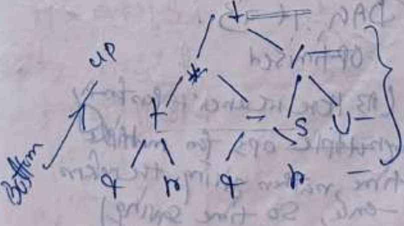
AST is a data structure used in c.g to rep. the structure of a prog. / code snipet.

eg – (a+b) * (a−b) + s/u ← Build AST

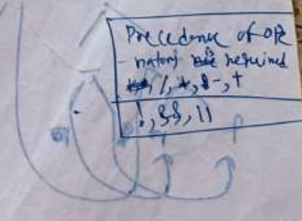
leaf nodes = operands
child nodes = operators

AST is a tree rep. of a prog's source code that helps the compiler to understand it's structure.





AST

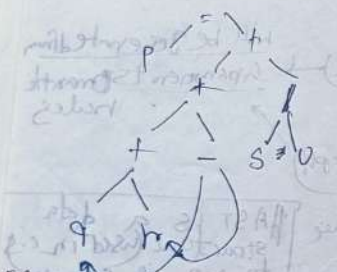


③ DAG pg 123

Common operations will not be performed multiple times, in this place we'll calculate the reference / give the references to the directions

DAG is a data structure used in C.d. to visualize the flow of values b/w basic blocks, rep. the structure of basic blocks & provide optimization techniques.

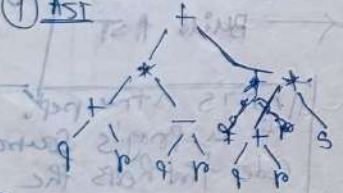
Eg ① P = (q+r) * (q-r) + s/u



Eg ② (q+r) * (q-r) + (q+r) * s

① AST? ② DAG? ③ DAG2?

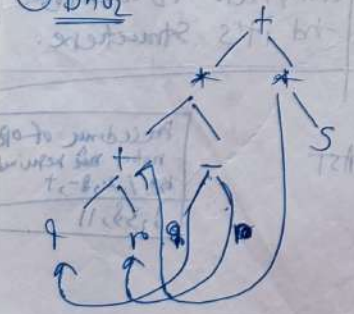
① AST



pg 123

It is not optimized. (As here we are performing same ops for multiple time, so time consuming)

② DAG



③ DAG2

pg 123

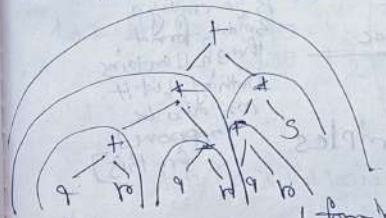
DAG it is optimised (As Here we can't perform same multiple ops for multiple time, rather giving the reference, so time saving)

2nd pg

② 3-address code for it → at least 3 things will be there

pg 124

1 operator (3 instructions)



① 2 address will be must/for

② In 1 exp there will not be more than 1 operators

(As LHS there will be an operand & at RHS there will be 1 or 2 operands & 1 operator)

forming from AST, so it's not optimistic, if it was DAG then we can use t_1 again as reference

$t_1 = q + r$;
$t_2 = q - r$; (2 add.)
$t_3 = t_1 * t_2$;
$t_4 = q + r$;
$t_5 = t_4 * s$;
$t_6 = t_3 + t_5$

It is a type of intermediate code, that makes use of at most 3 add. & 1 operator to rep. an expression & the value computed at each instruction is stored in temp variable generated by compiler

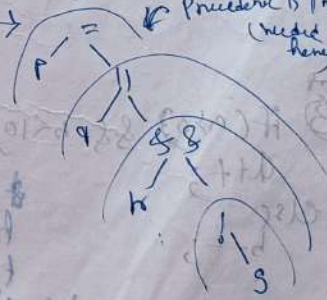
Write 3-address code for it

Eg ① if (a > 0)
    a++;
else
    b--;

② P = q || r && (!s) write 3-address code for it

AST

Precedence is imp (needed here)



$t_1 = !s$
$t_2 = r \&\& t_1$
$t_3 = q || t_2$
$p = t_3$

• $t_1 = a+1$; → this is also a 3 address as '++' is a single operator. [if the [] ++ [] this is generally a 3add. format though [] contains nothing, but it can also be approved for this]

Intermediate rep. of code

- ~~Quad~~ Quadruples
- → ~~A~~ triples
- → ~~Indeed~~ Indirect triples

① if (a>0) a++; else b--;

3-address code

$t_1 = a>0$
if t_1 goto L_1
goto L_2
L_1: $t_2 = a+1$;
$a = t_2$;
goto L_3 (+)
L_2: $t_3 = b-1$;
$b = t_3$;
goto L_3 → (if not written then also correct)
L_3: ..... (After the upper code what is the next code is rep. by '.....')

③ if (a>0 && b<10) a++; else b--;

3 add. C.

$t_1 = a>0$
$t_2 = b<10$
if t_1 goto L_1
goto L_2

triples

if a>0 goto t_0
goto L_2

if t_1 goto L_0
goto L_2
L_0: if t_2 goto L_1
goto L_2
L_1: $t_2 = a+1$;
$a = t_2$;
goto L_3 (not too need, but it's a good practice)
L_2: $t_3 = b-1$; $b = t_3$;
goto L_3 (+)
L_3: .....

L_0: if b<0 goto L_1
goto L_2
L_1: $t_2 = a+1$;
$a = t_2$;
goto L_3
L_2: $t_2 = b-1$;
$b_2 = t_2$
goto L_3
L_3: .....

① if (a>0 || b<10) a++; else b--;

$t_1 = a>0$
$t_2 = b<10$
if t_1 goto L_1
goto L_0
if t_2 goto L_1
goto L_2
L_0: if t_2 goto L_1
goto L_2
L_1: $t_2 = a+1$;
$a = t_2$;
goto L_3;
L_2: $t_3 = b-1$; $b = t_3$;
goto L_3
L_3: ...

3-address code

(5) while(n>0)
{ n = n%10;
n = n+n;
}

L: if t_1 goto L_1
goto L_2

L_1: t_2 = n%10;
n = t_2;
t_3 = n==t_3;
goto L0

L_2: ——

Intermediate rep. of code used (during various stages of compilation process)

① Quadruples (3-add. code) – rep. of statements in a prog. with 4 fields, typically consisting of an operator & 3 operands.

② Triples – similar to quadruples, but with 3 fields, typically & consisting of an operator & 2 operands.

③ Indirect triples – Triples that involve indirect addressing or references, usually for Compiler exp.s on memory operations.

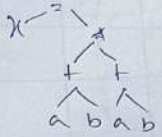
10/5/24

CW

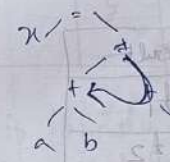
• DAG is also a kind of AST (but with no same task performed)

Eg

① n = (a+b) * (a+b)



→ AST



→ DAG
(Directed, as it has direction & acyclic as no cycle)

Eg

⑥ 3-address codes rep.

if (a>0 && b<10)
i++;
else
b--;

Triples

Add.	operation

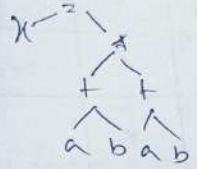
operation	op1	op2	result	
Boolean (greater than)	a	0	t_1	No need
Boolean (less than)	b	0	t_2	goto
Boolean (greater than)	a	0	goto L_0	
Boolean	–	–	goto L_2	
Boolean (less than	b	0	goto L_1	
–	–	–	goto L_2	

	Operation	op1	op2	Result
	Greater than	a	0	t_1
	less than	b	10	t_2
L_0 ⇒	Boolean	t_1	–	goto L_0
	–	–	–	goto L_2
L_0 ⇒	Boolean	t_2	–	goto L_1
	–	–	–	goto L_2
L_1 ⇒	Add	a	1	t_2
	Assignment	t_2	–	a
	–	–	–	goto L_3
L_2 ⇒	Sub	b	1	t_3
	Assignment	t_3	–	b
	–	–	–	goto L_3
L_3 ⇒	....	....	....	....

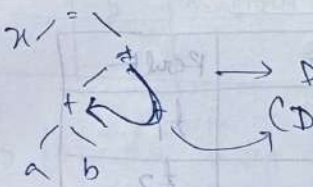
↳ quadruples by 1st way

• DAG is also a kind of ask perform)

→ $x = (a+b) * (a+b)$



→ AST



(6) 3-address codes for

if (a>0 && b<10)

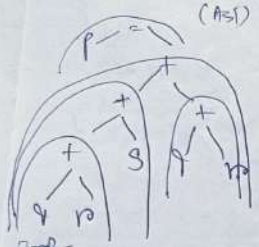
a++;

else

b--;

Operation	op1	op
Boolean (greater than)	a	
Boolean (less than)	b	
Boolean (greater than)	a	
Boolean	–	
Boolean (less than	b	
–	–	

(i) (ii) p = (q+r)*s + (q+r)



$t_1 = q + r$
$t_2 = t_1 * s$
$t_3 = q + r$
$t_4 = t_2 + t_3$
$p = t_4$

Quadruples (as 4 cols so quadruple)

operation	op1	op2	Result
Addition	q	r	t_1
multiplication	t_1	s	t_2
Addition	q	r	t_3
Addition	t_2	t_3	t_4
Moving Assignment	t_4	-	p

Triples (3 cols, so triple)

operation	op1	op2 / Result	Address
Addition	q	r	1 (As we can't get space to write next, so write it)
mul	(1)	s	2
Add	q	r	3
Add	(2)	(3)	4
Assignment	(4)	P (result)	5

Indirect triples (As we use address of address, so indirect (like indirect + addressing mode))

Address of Address	Address	operation	op1	op2 / Result
A	1	Add	q	r
B	2	mul	((A))	s
C	3	Add	q	r
D	4	Add	((B))	((C))
E	5	Assignment	((D))	P

↓

It is logically most useful. But it is not used in generally. It is useful as it know in computer always contigious memory allocation doesn't happen, paging happens etc.

(i) (ii)

L_1:	Add	a	1	t_1 (annonced)
	Assignment	t_1	-	a
	-	-	-	goto L_3
L_2:	Subtraction	b	1	t_2
	Assignment	t_2	-	a
	-	-	-	goto L_3
L_3:	.....	----	----	---

(try others may (the 1st way))

Triples

Address	operation	op1	result / op2
~~100~~	~~Boolean (greater than)~~	a	0
2	~~Boolean~~	(1)	goto L_0
	—	—	goto L_2
1	greater than	a	0
2	Less than	b	10
3	Boolean	(1)	goto (5)
4	—	—	goto (70)
5	Boolean	(2)	goto (7)
6	—	—	goto (70)
7	Addition	a	1
8	Assignment	(7)	a
9	—	—	goto (13)
10	Sub	b	1
11	Assignment	(70)	b
12	—	—	goto (13)
13	—	—	—

⑦ for(i=0; i<n; i++)
a=a+5;

3-add. Code

i=0 } assignment

L_2: $t_1 = i < n$ }

if t_1 then goto L_0

goto L_1

L_0: $t_2 = a + 5$
$a = t_2$
$t_3 = i + 1$
$i = t_3$
goto L_2

L_1: ---

⑧ do
{
$h = n \% 10$;
$n = n / 10$;
} while(n>0)

3-add. Code

L_0: $t_1 = n \% 10$;
$n = t_1$
$t_2 = n / 10$;
$h = t_2$
$t_3 = n > 0$
if t_3 then goto L_0
goto L_1

L_1: ---

⑨ while (n>0)
{ n=n/10;
}
3 add. code

L_2: t_1 = n>0
if t_1 goto L_0
goto L_1

L_0: t_2 = n/10;
n = t_2;
goto L_2
L_1: ....

⑩ switch(ch){
{case 1: a=a+b;
break;
case 2: a=a*b;
break;
} ~~case 3~~

if ch==1 goto L_1
if ch==2 goto L_2
goto L_3

L_1: t_1=a+b } (As targeting AST so this format otherwise it is already in 3-add. format)
a = t_1
goto L_3

L_2: t_2 = a*b
a = t_2
goto L_3

L_3: ....

If no 'break;' then either write goto L_2 instead of goto L_3, or, it has no need to write as it will automatically go to L_2

2nd way
if ch==1 goto L_1
goto L_2 (if not L_1 or L_2 then automatically in L_4)

L_2: if ch==2 goto L_3
goto L_4

L_1: t_1=a+b
a = t_1
goto L_4

L_3: t_2=a*b
a = t_2
goto L_4

L_4: ....

Assignment-3
H/W ⑪ while (n>10)
{
if (a>5 && b<c || c==10){
p++;
else
{ q++;
n---
}
write 3-add. codes, quadruples, triples for it.

L_0: t_1 = n>10
if t_1 goto L_1
goto L'

L_1: if t_2 goto L_2
goto L_4

L_2: if t_3 goto L_3
goto L_4

L_3: t_5 = p+1
p = t_5
goto L_6

L_4: if t_4 goto L_3
goto L_5

L_5: t_6 = q+1
q = t_6
goto L_6

L_6: $t_7 = n - 1$
$n = t_7$
goto L_0
L': - - - - - - -

Quadruples

operation	op1	op2	Result
Greater than	a	5	t_2
less than	b	6	t_3
Equal to	c	10	t_4
Greater than	n	10	t_1
Boolean	t_1	-	goto L_1
-	-	-	goto L'
Boolean	t_2	-	goto L_2
-	-	-	goto L_4
Boolean	t_3	-	goto L_3
-	-	-	goto L_4
Add	P	1	t_5
Assignment	t_5	-	P
-	-	-	goto L_6
Boolean	t_4	-	goto L_3
-	-	-	goto L_5
Add	q	1	t_6
Assignment	t_6	-	q
-	-	-	goto L_6
sub	n	1	t_7
Assignment	t_7	-	n
-	-	-	goto L_0
L'	- - -	- - - -	- - - - - - -

Triples

Address	operation	op1	op2 / Result
1	Greater than	a	5
2	less than	b	6
3	Equal to	c	10
4	Greater than	n	10
5	Boolean	(4)	goto (7)
6	-	-	goto (22)
7	Boolean	(1)	goto (9)
8	-	-	goto (14)
9	Boolean	(2)	goto (11)
10	-	-	goto (14)
11	Add	P	1
12	Assignment	(11)	P
13	-	-	goto (19)
14	Boolean	(3)	goto (11)
15	-	-	goto (16)
16	Add	q	1
17	Assignment	(16)	q
18	-	-	goto (19)
19	Sub	n	1
20	Assignment	(19)	n
21	-	-	goto (4)
22	- - -	- - - -	- - - - - - -

13/5/24

(12) for(i=0; i<n; i++)
{
if (x>10 || y<10)
p++;
else
q++;
}

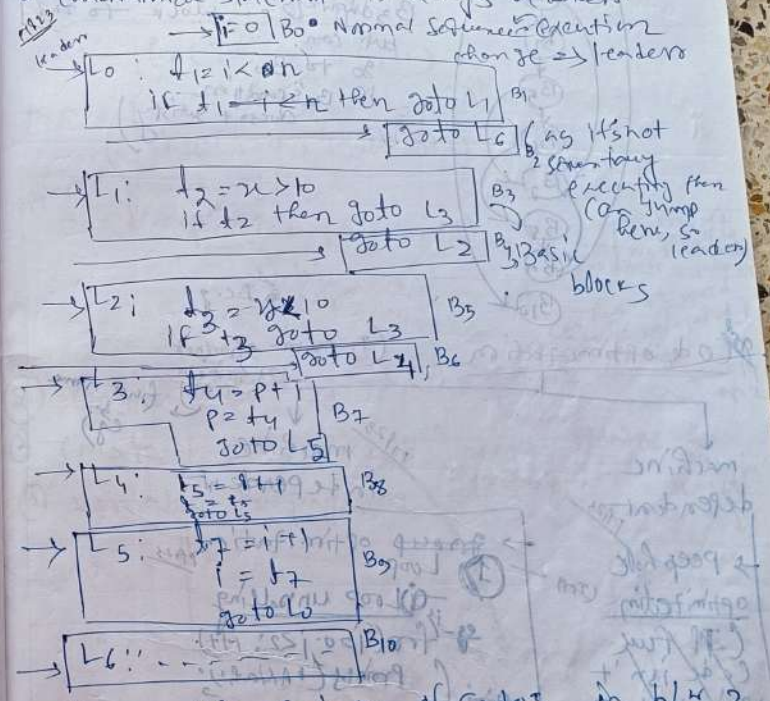
3-add. Code

i=0
L_0: $t_1 = i < n$
if t_1 then goto L_1
goto L_6
L_1: $t_2 = x > 10$
if t_2 goto L_3
goto L_2
L_2: $t_3 = y < 10$
if t_3 goto L_3
goto L_4
L_3: $t_4 = p + 1$
p = t_4
goto L_5
L_4: $t_5 = q + 1$
q = t_5
goto L_5
L_5: $t_6 = i + 1$
i = t_6
goto L_0
L_6: ——

Control flow graph (CFG) – A graphical rep. of the paths that a prog may take during execution.

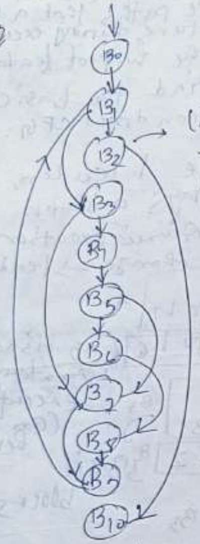
We'll do that with the help of leaders. From leaders we can find the basic blocks. From the blocks we can draw CFG.

- Start statement could be 1 leader.
- Conditional statement is always a leader.



- Block — The no. of codes in b/w 2 leaders, is called a block.
- A leader is that (statement) where the sequence of execution changes.

P.123



(implementating from B2 we can't go to B3 always, but can go to B10 if B2's condition doesn't satisfy)

[B/w 2 blocks → no path → means can't go from 1st block to 2nd & if path → can go from 1st block to 2nd]

I.C.g

Code optimization

(In exam = short note on optimization) (first name & eg)

- machine dependent (P.123)
  - → peephole optimization (imp)
    (If fully code isn't optimized it is a technique to optimize the code by optimizing slides of code the)
- machine independent (P.123)
  - → optimization (P.123)

1) Loop

(i) Loop unrolling

eg -
```
for(i=0; i<2; i++)
    Printf("Ahalya");
```
optimized code
```
Printf("Ahalya");
Printf("Alich");
```

(ii) Code movement / code motion / freq reduction

eg
```
for(i=0; i<100; i++)
{ x=y+z;
  if(i==5)
    p++;
}
```
optimized
```
x=y+z;
for(i=0; i<100; i++)
{ if(i==5)
    p++;
}
```
→ (it is called freq reduction as, if it would be inside the code, it's freq (the no of times it's executing) = 100, but here freq = 1)

(iii) Loop jamming / fusion

eg -
```
for(i=0; i<100; i++)
    n++;
for(j=0; j<100; j++)
    y++;
```
optimized
```
for(i=0; i<100; i++)
{ n++;
  y++;
}
```

2) Constant folding → eg $x = \frac{22}{7} * p2$ (as here $\frac{22}{7}$ need to be calculated) optimized → $x = 3.14 * p2$ (we know 2 values, if the req is well many write it, rather we...)

3) Constant propagation → eg
```
x=12;
y=x/3+4;
```
optimized →
```
x=12;
x=16;
```
(where we know, it will not be changed later (y will not be changed later, y is fixed))

4) Variable propagation

eg
```
a=x+y
b=x+y
c=b+d
```
optimized
```
a=x+y
c=a+d
```
(value isn't changing, if we will change then not this)

5) Common subexpression elimination (like variable propagation)

eg
```
c=a+b
x=c
d=x+y
```
optimized ⟷
```
c=a+b
d=c+y
```

(6) ① Peephole optimization

i) Strength reduction

eg $x^2 \rightarrow x * x$

$x * 2 = x + x$
$= x << 1$ [as '<<' is comparatively easy to compiler]

ii) unreachable / dead code elemination

eg switch(ch)
{
printf("Alink:", x → dead code (as it will never execute) → optimized
case 1: a = a+b
break;
case 2: a = a-b
break;
}

iii) Replace slower instruction with faster one

ADD R, #1 → slow ins.
optimized ↳ INR R → fast "

iv) Redundant load store removal

② Parallel execution of operation (most fast imp)

i) pipelining optimization
③ Distributed concept
④ Cache optimization

Pg 23

Ques 4-A

a) Determine the strings generated by this regular expression: a(bc)?d

ad
abcd

4-B

5) calculate LR(1) item sets of the grammar given below -

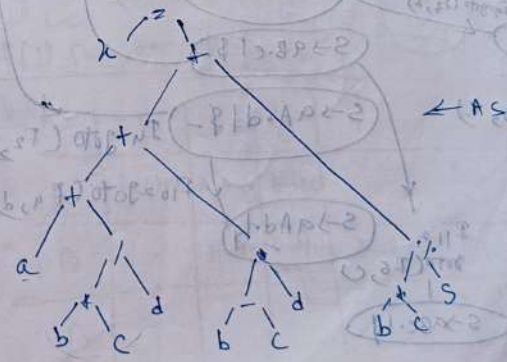
S → aAd | bBd | aBe | bAe
A → c
B → c

Here, N = {S, A, B} & S is starting symbol.

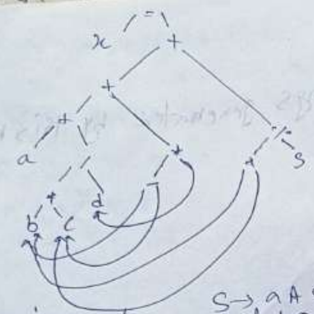
7) From AST & DAG for following statement:

x = a + (b*c) / d + (b-c) * d + (b*c) / s

x = ((a + (b*c) / d) + (b-c) * d) + (b*c) / s) * / / (+ -)



← AST

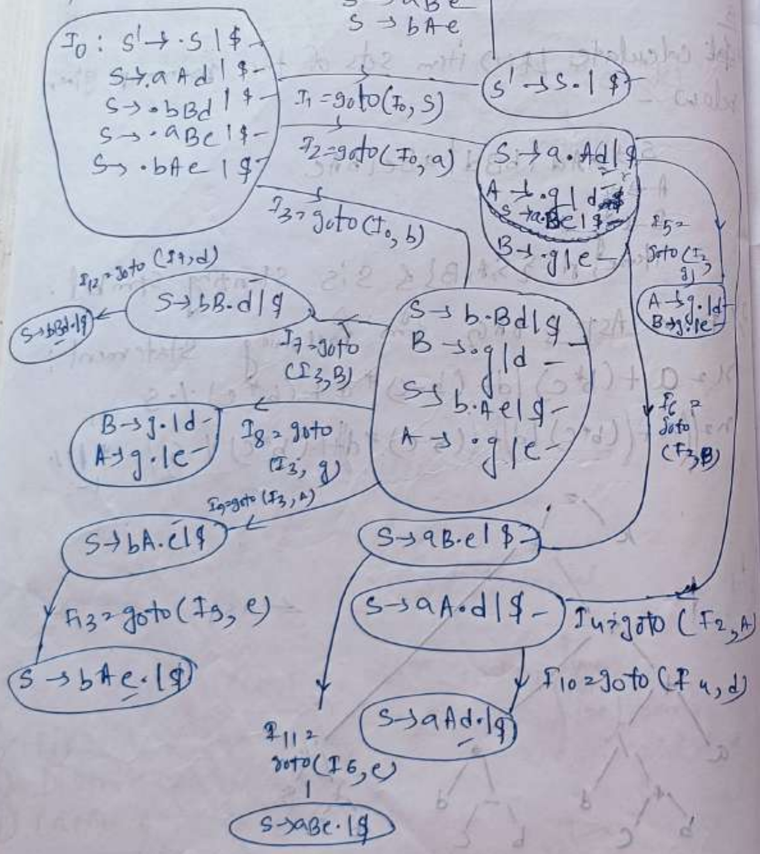


← DAG

c) LR(1) item sets

S → aAd
S → bBd
S → aBe
S → bAe

A → g
B → g



5-c

Q) Conclude whether the following grammar could be considered as LL(1) Parser or not.

A → rB | DS
B → At D
D → tB | ε.

Here N = {A, B, D} & A is Start symbol

NT	Fi	Fo
A	{r, t, s}	{$, t}
D	{t, ε}	{s, $}
B	{r, t, s}	{$, s}

A → rB
(i)

A → DS
B → At D , B → At D
(ii)

D → tB
(iii)

LL(1) parsing table

	r	s	t	$
A	A → rB	A → DS	A → DS	
D		D → ε	D → tB D → ε	D → ε
B				

more than 1 entry elements in this entry. so, it can't be LL(1) parser.

11) b) Consider the syntax directed definations given below. Using the SDTs, what will be the o/p printed by a bottom up parser for the i/p : 2*3+5?

$S \to ER$ $\$$

$R \to * E$ { printf("*"); }

$R \to \epsilon$ $\$$

$E \to F + E$ { printf("+"); }

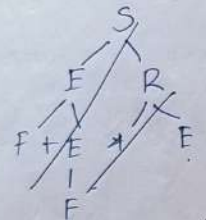
$E \to F$ $\$$

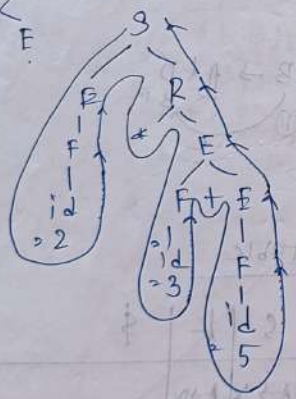
$F \to (S)$ $\$$

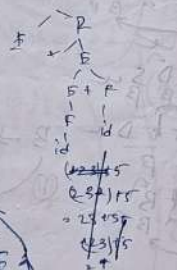
$F \to id$ $\$$ { Printf("%d", id.lexval); }

Here N = { S, E, F, R } & S is Start symbol.

i/p = 2*3+5







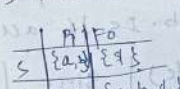
o/p = 235+*

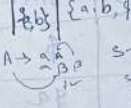
12) a) Find all LR(0) item sets for the following grammar.

$S \to AA$ — (1)

$A \to aA$ — (2)

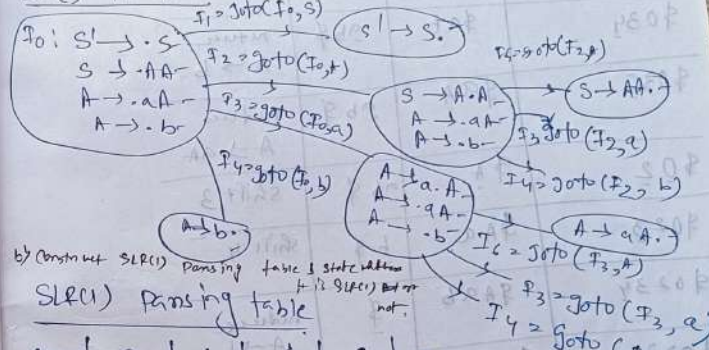
$A \to b$ — (3)





Here N = { S, A } & S is Start Symbol.

LR(0) item sets



b) Construct SLR(1) parsing table & state whether it is SLR(1) or not.

SLR(1) parsing table

States	a	b	$	S	A
0	S3	S4		1	2
1			accept		
2	S3	S4			5
3	S3	S4			6
✓ 4	r3	r3	r3		
✓ 5			r1		
✓ 6	r2	r2	r2		

It is SLR(1) ~~Asp~~ parsen.

5 (i) Explain full parsing mechanism for an i/p : abab. Is it accepted or not?

Stack (state)	Stack	i/p	Actions
$0	$	abab$	shift 3
$03	$a	bab$	shift 4
$034	$ab	ab$	return A → b
$036	$aA	ab$	return A → aA
$02	$A	ab$	shift 3
$023	$Aa	b$	shift 4
$0234	$Aab	$	reduce A → b
$0236	$AaA	$	reduce A → aA
$025	$AA	$	reduce S → AA
$01	$S	$	accept

Yes, it is accepted.