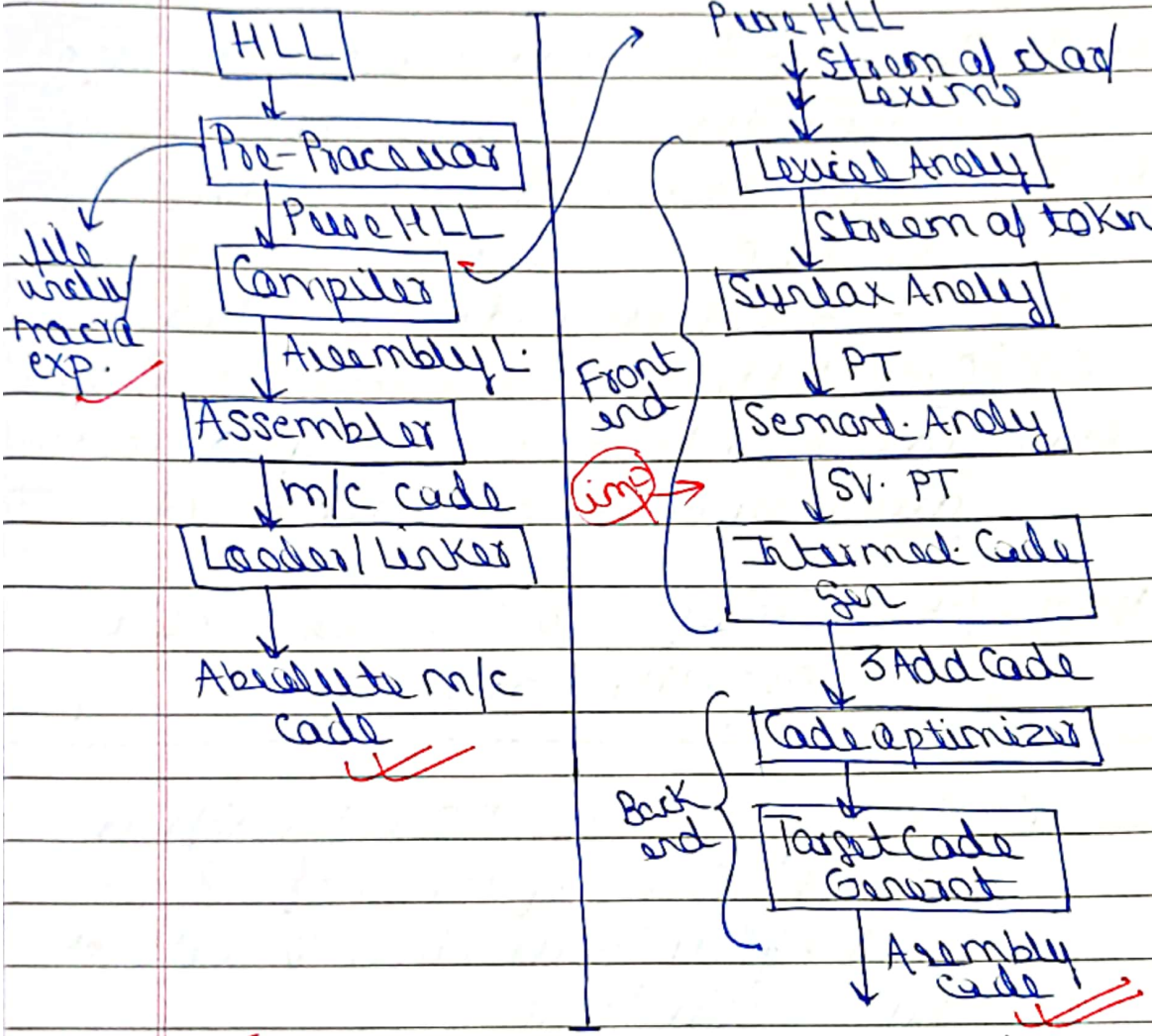
COMPILER DESIGN

1. Intro to Various Phases of Compiler



→ Phases of compiler is also associated with Symbol T & error Handler.

eg:
$x = a + b * c$: Stream of lexins.

(i) Lexical Analysis: lexins → Stream of tokens

→ Remove white space/comments.
→ Identifies using RE (patterns)
∴ id = id + id * id. [Token]

token

(ii) Syntax Analysis — SA ← Grammar

- Builds PT = i/p to SA.
- Known as Parser.

↓ Parse Tree.

(iii) Semantic Analysis : type checking & generates SV. Parse tree

(iv) Intermediate Code Generator :
$t_1 = b * c$; $t_2 = a + t_1$; $x = t_2$.
⇒ 3 Address code generated.

(v) Code Optimizer : reduces no of line

(vi) Target Code Generator : produces assembly code eg: Mul r1, r2

NOTE: change only CO & TCG, to make a new compiler (BE)

* SYMBOL TABLE : DS where info is stored / used by compiler (imp)
- used by Synthesis Phase & called by Analysis Phase.

⇒ Attributes in Symbol Table :
[Name, Size, dimension, type, line of dec, line of usage, Address].

* Operations of ST depends upon lang.
Block Structured & Non-Block struct
[insert, lookup, Set, reset] [insert, lookup]

* Implementations of Symbol Table (imp) [Will be useful in data Structures Qs]

(1) Array → Ordered [Sorted] — LT: O(n), LT: O(log n)
Unordered — LT: O(n), LT: O(n)

(2) LinkedList → Ordered — LT: O(n), LT: O(n)
Unordered — LT: O(1), LT: O(n)

(3) Self organising list : LT: O(1), LT: O(n)

(4) Search tree : LT / LT : O(log n)

(5) Hash Table : LT / LT : O(1)

① Lexical Analyser
→ reads char by char, uses FA (imp)
→ ids as identifiers & keywords
→ removes white Space & comments.
Stream of lexim → Stream of tokens
eg → Pg 12 [No. of tokens] (Gate)

② Syntax Analyser — SA ← G, i/p, ↓ PT.
→ Parser / SA doesn't allow Ambiguous Grammar, to avoid confusion. (exc. operator Prec Parser).
more than 1 LMD / PT.
eg → Pg 14 [Ambiguous]

* Errors in LA :

- Number literals too long / ill formed no. literals : int a = $123.

(ii) Ambiguous & Unambiguous Gramm.

imp

(I) + → Left Associative → Left recursive
E → E + id | id (LR)

(II) Highest Prec. operator in last level

E → E + T | T
T → T * F | F
F → id

* > prec +

∴ Take care of associativity : recursion
take care of precedence : levels

Hence, removes ambiguity.

eg → Pg 20 / Pg 22 [removes ambiguity] [through CNF/PNF]

∴ Ambiguous → Unambiguous is an undecidable algorithm

(iii) Elimination of Left Recursion

A → Aα | β : LRG. (causes ∞ loop)
A → αA | β : RRG.

∴ Top Down Parsers cant work with Left recursive Gramm.

A → Aα | β → A → βA'
imp → A' → αA' | ∈ .

eg → Pg 25 [2 examples]

(iii) Deterministic & Non-deterministic Gramm

A → αβ1 | αβ2 | αβ3 → NDG
∴ for String : αβ3 → lot of backtracking
Common Prefix problem

imp

∴ TDP cant work with NDG
NDG → DG (Left Factoring)

eg → Pg 28 [S → bSSaaS]

(2) * PARSERS

- Top Down Parser [NDG X, AG X, LR X]
  - TDP with full backtracking
    - Recursive descent
    - Non-recursive descent LL(1)
  - TDP without full backtracking
- Bottom up Parser
  - operator precedence Parser [AG ✓]
  - LR Parser
    - LR(0)
    - SLR(1)
    - CLR
    - LALR

LR(0) items ⊂ SLR(1) ⊂ LR(1) items ⊂ CLR ⊂ LALR
powerful

eg → Pg 31 [TDP | BUP]

imp

TDP follows left most derivative
BUP follows RMD (reverse)

1) LL(1) PARSER (Non-recursive descent)

→ L: Scan left to right.
L: LMD. [1 look Ahead]
→ LL(1) Parsing Table & a Stack.
→ Top Down Parser.
→ functions → first() & follow()

imp

eg → Pg 33 (Q2) ii) S → ABCDE
follow(E) = follow(S) = $
follow(A) = first(B)

eg → Pg 34/35 [E → TE', S → aBDh]

* Construction of LL(1) Parsing Table

→ rows → variables / Non-terminals
→ columns → terminals.

eg → Pg 13 [Rev copy]

imp

T → FT' will be in the:
column no: first(F)
row no: T

F → id | (E):
{F → id : col no: id
F → (E) : col no: (
} row no: F

* If multiple entries in 1 cell → LL(1) X

If LL(1): (Gate)
eg: A → α1 | α2 | α3 | ..... αn
first of α1, first of α2 ... must be mutually exclusive.
eg: A → α | ∈
first of α, follow of A must be mutually exclusive.

eg → Pg 38/39 [S → aABb / Q2, Q3]

2) Recursive Descent Parser [TDP]

imp

→ 1 function for every variable.
eg → Pg 40/41
→ Recursion Stack (OS) is used.

* Bottom Up Parsers

1) Operator Precedence Parser: define Math operations for compiler.
→ AG allowed.

imp

* Operator Grammar: No 2 Vn adjacent
No ∈ productions.
S → SAS | a
A → bSb | b
Not oper. Gramm

S → SbSbS | SbS | a
A → bSb | b
Oper. Grammar

(ii) Using Operator Precedence Table

$E \to E+E \mid E*E \mid id$ →

	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	-

$: Least Prec
id: Highest Prec

[Oper. Precedence Table]

If n operator: Size = $O(n^2)$.

→ Drawback of OPT ↗

(iii) Using Operator Function Table

- First construct oper. Precedence Graph.
- Find longest path starting from node (f id, g id).

eg → Pg 46

- Operator func^n Table →

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

R | OPT → $O(n^2)$ | OFT → $O(2n)$ | imp

→ Drawback of OFT: less error detecting capability (more Blank entries)

2) LR PARSER → BUP.

LR(0) < SLR(1) < LALR(1) < CLR(1)

LR(0) items — LR(1) items

powerful

→ They use LR Parsing Table & Stack.

(a) LR(0) PARSER

→ LR(0) items used to construct LR(0) Parsing Table. [Any Production with •]

eg: $S \to A \cdot A$ (seen A), $S \to AA \cdot$ (seen everything)

eg → Pg 53 [Canonical collection of LR(0) items using goto() & closure().

Goto move → on Variables
Action/Shift move → on Terminals.
reduce move → Final items

eg → Pg 54 [Parsing Table] imp

State	Action			Goto	
	id	+	$	E	T
0	s5			1	2
1		s3	acc		
2		r2	r2		
3	s5				4
4		r1	r1		
5		r3	r3		

(b) SLR(1) PARSER

- Shift & Goto Moves are same as LR(0)

LR(0): place r_i in entire row [Acc]
SLR(1): place r_i in the follow of LHS

imp

SLR(1) > LR(0)

- powerful
- detects more no. of error due to Blank entries

ii) 2 reduce moves in 1 state : RR conflict

(ii) Conflicts in LR(0)

SR conflict

	a	b
5	S_6/r_1	r_1

RR conflict

	a	b	c
5	r_1 r_2	r_1 r_2	r_1 r_2

imp → 2 final items
$I_5 = A \to \alpha.$
$B \to \beta.$

(iii) Conflicts in SLR(1)

SR conflict

$I_5 \Rightarrow S \to .aA$ ①
$A \to \alpha.$ ②

	a	b
5	S_6	

ii) follow(A) ≠ a, NP.
∴ r_2 placed in follow(A)

RR conflict

$I_5 \Rightarrow A \to \alpha.$ ①
$B \to \beta.$ ②

r_1 will be placed in follow(A), r_2 in follow(B).
∴ ii) follow(A) ∩ follow(B) ≠ ϕ ∴ RR

NOTE: If Gramm is LR(0), definitely SLR(1)

eg → Pg 60

eg → Pg 65 [Q → call]
[Check the final items, & look for conflicts in them].

eg → Pg 66 [Call]

imp

(iv) CLR(1) AND LALR(1) Parsers

→ Canonical coll. of LR(1) items used.
LR(0) items + LookAheads

$S \to AA$
$A \to aA/b$

→ $S' \to .S\$, \$$
$S \to .AA, \$$
$A \to .aA, a/b$
$/b, a/b$

imp → [LA = first(A) ; ∵ A is after Acclaim comma]

→ while applying transition, dont change LA, while closure(), change LA.

eg → Pg 71 [Canonical coll. of LR(1)]

eg → Pg 72/73 [CLR & LALR Parsing Table]

NOTE: In CLR(1), reduce moves are only present in LAS. [Blank space ↑, error det. ↑]

* States → imp → Gate

$N(LALR(1)) = N(LR(0)) = N(SLR(1)) \leq N(CLR(1))$

→ Merge those States with same LR(0) items, but diff LA.

* Conflicts in LR(1) items

i) SR Conflict

$A \to \alpha \cdot a\beta, c/d$
$B \to \alpha \cdot, a/\$$: SR.

a → shift
γ in $a/\$$ → reduce

ii) RR : $A \to \alpha \cdot, a$, $B \to \alpha \cdot, a$ — γ_1, γ_2 both under a

* if CLR(1), its not necessarily LALR(1), maybe after merge, conflict can occur.

eg → Pg 79/80 (imp)

eg → Pg 29 [Rev Copy] (Case 1)

$A \to \cdot AB, \$/a/b$
$/\cdot, \$/a/b$

* CLR(1) ⊃ LALR(1) ⊃ SLR(1) ⊃ LR(0), LL(1) ⊂ CLR(1)

→ Every LL(1) must be CLR(1) but may/may not be LALR(1)

→ Between SLR(1) | LALR(1)
- reduce move in follow(LHS) | - reduce move in LAS
- Goto/Shift exactly identical

(imp)

eg → Pg 85

eg → Pg 86 [Handle → Goto Q] (Gate)
$E \to E+n \cdot$ (all)

eg → Pg 87+ [No. of states]

NOTE: If 2 States have same LR(0) items, but different lookAheads
(imp) → n(CLR(1)) > n(LALR(1))

* Advantages of AG: No. of prodⁿ less, meaningful, flexibility ↑.
[But we need to manage conflicts meaningfully, if we go with AG]

* YACC generates LALR(1) PT.
if, SR conflict → Shift.
RR conflict → 1st Ka reduce
(Gate) (imp)

(3) SYNTAX DIRECTED TRANSLATION

→ Grammar + Semantic rules = SDT.
eg: $E \to E+T$ {E.V = E.V + T.V}

Eg → Pg 91 [BUP technique]

* SDT is carried out along with Parsing either: → BUP [traverse top-down, left-right & perform action while reducing] (imp)
→ TDP [carry semantic actions while parsing]

eg → Pg 93 [Infix → Postfix → TDP]

* Compute value, without creating Parse tree.

eg → Pg 28 [Rev Copy] (Use associativity & precedence, derived from Gramm rules)

imp

NOTE: No. of reductions = No of non-leaf

(1) SDT to create Abstract Syntax tree [using Linked Rep]
(2) SDT for type checking
(3) SDT to translate Binary no. to decimal.

imp
- 01 ⇒ $1/2^2 = 0.25$
- 11 ⇒ $3/2^2 = 0.75$

(2 → no. of bits after binary value after)

eg → Pg 30 [Rev Copy]

(4) SDT to generate 3 Address code.

eg → Pg 106 [Prev Years]

* S-Attributed & L-Attributed

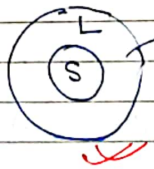
* 2 types of Attributes
  - Synthesized attributes:
    A → BCD
    A.S = f(B.S, C.S, D.S)
    [A takes value from children]
  - Inherited Attributes:
    A → BCD
    C.i = A.i / B.i
    [child takes value from parent/sibling to its left]

imp

* S Attributed SDT = use only Synthesized attributes.
  L-Attributed SDT = use both inherited & Synthesized attrib.

S-Attributed	L-Attributed
- Semantic actions placed at RE of prodn. A → BC { }	- Semantic actions can be placed anywhere in prodn. A → { } BC
- Attributes evaluated during BUP [reductions]	- Attributes evaluated by traversing DFS, left to right [Top Down]

imp



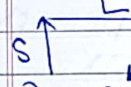
If SDT is S-attributed, it is also L-attributed.

Gate

eg → Pg 32 [L-attributed SDT]

info flows in L attributed

info flows in S attributed



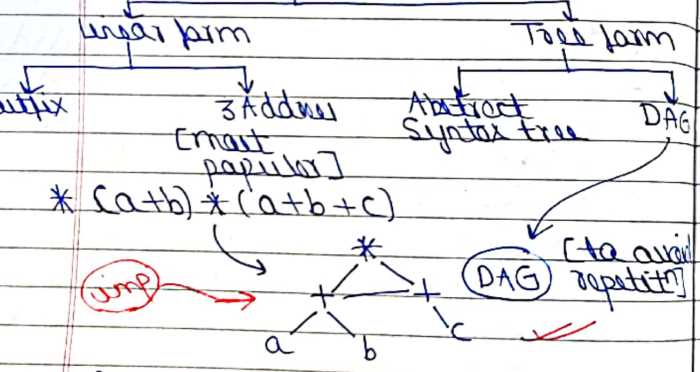
eg → Pg 112 [S-attributed Grammar]

4 Intermediate Code Generation

→ ICG is same for all m/cs

→ Lex & phase changed to make a new compiler.

Intermediate code



eg → Pg 114 [Types of 3 Address code]

eg → Pg 115 [Representations of 3 Add code]

imp [Quadruples, Triple, indir Triple]
(mostly used) (not much used) (mostly used)

* BACK PATCHING & Conversions to 3 Address Code

↳ leaving labels empty & fill them later.

eg → Pg 119 [3 Add code for For Loop]

NOTE: ∵ No braces in Intermediate code, unlike HLL, difficult to identify loops.

eg → Pg 120 [3 Add code for Switch]

↳ Qs on back patching can be asked

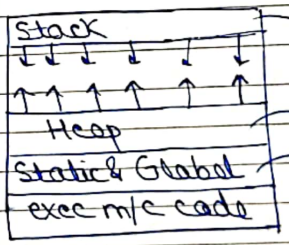
* Goto Qs:

imp

eg → Pg 123 [Goto Qs]

eg → Pg 126 [Goto Qs]

5 RUNTIME ENVIRONMENT



→ fun^n call / AR pushed

→ dynamic memalloc

→ perm storage [text is static]

[Stack & heap both grow dynamically in opp dir^n to avoid memory wastage].

* 3 Storage Allocation Strategies:

1) **Static**: (imp)
   - Allocⁿ done at compile time
   - 1 AR per procedure
   - Recursion X
   - Size of data obj. must be known at compile time
2) **Stack**:
   - Local Var. can be retained once AR vanishes / popped.
3) **Heap**:
   - Allocation/ deallocⁿ in any order (at run time).

NOTE: ∴ we use Hybrid of all three.

(6) CODE OPTIMIZATION

- reduce no. of lines in program, mainly loop (∵ major time of prog spent in loop).

CO → m/c independent
   → m/c dependent

(i) m/c independent → (imp)
- Loop optimizⁿ
- folding
- Redundancy elimⁿ
- Strength reductⁿ.

1) * Loop optimization

→ Find leaders, then find Basic Block (BB) then CFA or PFG; then identify loops; optimize it.

[ ] BB [Seq of 3 addcode]

eg → Pg 133 [Any cycle in PFG, indicates loops]

* Algo to find leaders.

1) If, L_1 = leader, L_{10} = leader then $L_1 - L_9$: BB (imp)
2) 1st Stat = leader
3) Target of cond/ uncondⁿ Stat = leader
4) Stat that follows cond^n / uncondⁿ Statement = leader.

eg → Pg 135/136 (imp)

∴ n leaders = n Basic Blocks

* Types of loop optimization.

a) code motion / loop reduction: reduce no. of operations inside loop eg: sin x / cos x. (imp)

b) loop unrolling: reduce no. of times a loop executes.

(c) Loop Jamming : reduce no. of loops [combine 2 loops together]

2) Folding : 2+3+C+B ⇒ 5+C+B

3) Redundancy elimination

eg: A = B+C ; D = 2+B+3+C

D = 2+3+A (DAG)

D = 5+A (old)

4) Strength reduction : replacing costly operation by cheaper one.

eg: B = A*2 → B = A<<1 (DS core)

* PREVIOUS YEAR QS.

1) Advantage of DLL over SLL. (imp)

→ Smaller size of executible, ∵ DLL are seperated from exe.

→ DLL shared among executibles, ∴ Page fault ↓

→ Prog need not be relinked to newer version, automatic linking occurs.

* Referencing at linking time

* L-attributed SDT can be evaluated in BUP, there exist parsing algo. for same lang, whose complexity is less than $\Theta(n^3)$ (imp)

* Semantic Analysis – Type check. Runtime environment – AR.

eg → Pg 45 [Diag]

* Relocation : process of assigning load addresses to various parts of prog & adjusting the code & data in prog to reflect assigned addresses.

* Handle : production p. that will be used for reduction in the next step along with pos^n in the sentential form where, RHS of prod^n may be found. (imp)

* LALR(1) can have SR conflict, if & only if, CLR(1) has SR conflict [~~not RR~~] [not RR]

* Stack contains only a Set of Viable Prefixes.

Lexical Analysis — DFA minimiz^n

Parsing → Production tree

Reg Alloc → Graph colouring. (imp)

Exp. Evaluation → Post order traversal.