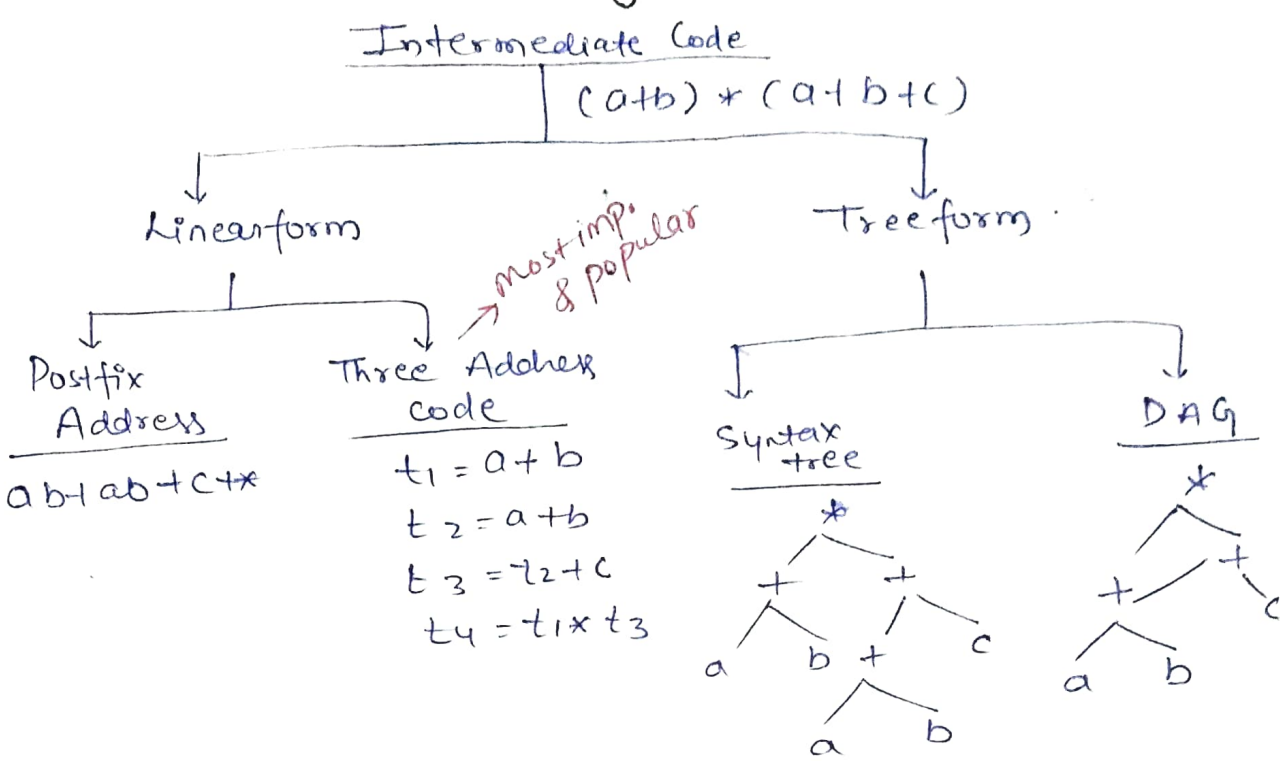
Intermediate code generation



Syntax tree → Condensed and modified version of parse tree.

DAG (Directed Acyclic Graph) → It doesn't repeat anything and use all the things that is already generated.

To remember

Types of 3 address code -

1. $x = y \ op \ z$
2. $x = op \ z$ (Unary)
3. $x = y$
4. if x (relop) y goto L
5. goto L
6. $A[i] = x$
   $y = A[i]$
7. $x = *P$
   $y = \&y$

⇒ We convert Higher level code to three address code in order to generate the machine code.

Various representation of 3-address code.

Ex → (a+b) + (c+d) + (a+b+c)

	Quadruples				Triple				Indirect Triples		
		opr	op1	op2	res		opr	op1	op2		
$t_1 = a+b$	1)	+	a	b	t_1	1)	+	a	b	(i) (1)	→ Pointer to instruction
$t_2 = -t_1$	2)	−	t_1		t_2	2)	−	(1)		(ii) (2)	→ instruction to be executed
$t_3 = c+d$	3)	+	c	d	t_3	3)	+	c	d	(iii) (3)	
$t_4 = t_2 * t_3$	4)	*	t_2	t_3	t_4	4)	*	(2)	(3)	(iv) (4)	
$t_5 = a+b$	5)	+	a	b	t_5	5)	+	a	b	(v) (5)	
$t_6 = c+t_5$	6)	+	t_5	c	t_6	6)	+	(5)	c	(vi) (6)	
$t_7 = t_4+t_6$	7)	+	t_4	t_6	t_7	7)	+	(4)	(6)	(vii) (7)	
	adv. statements can be move around				adv. space is not wasted				adv. statement can't be moved		
	dis: too much of space is wasted				dis. statement can't be moved				dis. two memory access.		

Order can be changed bcoz all statements are independent

Instrn order can't be change bcoz statement are dependent

Back patching and conversations to 3 address code

if (a<b) then t=1
else t=0

] four address code but we can implement it using 3 address code.

(i): if a<b goto (i+3) — ① Blank
(i+1) : t=0
(i+2) : goto (i+4) — ② Blank
(i+3) : t≠1
(i+4) :

leaving the tables as empty and filling them later called as back patching.

Explaination → if a<b then goto instruction at (i+3) else (i+1) automatically get executed because it comes just next. Either (i) or (i+1) will get executed and after that pointer comes at (i+2) then exit the code.

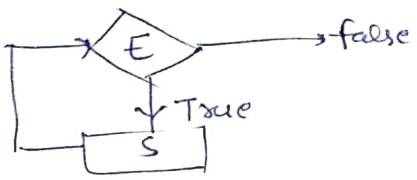
If a<b and c<d or e<f

100) if a<b goto 103
101) t1 = 0
102) goto 104
103) t1 = 1
104) if (c<d) goto 107
105) t2 = 0
106) goto 108
107) t2 = 1
108) if (e<f) goto 111
109) t3 = 0
110) goto 112
111) t3 = 1
112) t4 = t1 and t2
113) t5 = t4 or t3

3 address code using Back patching

while loop → If is not in 3 address code

while E do S



① L: if (E == 0) goto L1 (T → means condn is false)
S
goto L
L1:
] 3 address code

② L: if E goto L1] → If E is true goto execute L1 else
goto Last] → move to last

L1: S
goto L

Last: Code other than while loop in program.

while loop can be execute in 3 address code using various other methods too.

```
while (a<b) do
x = y+z
```

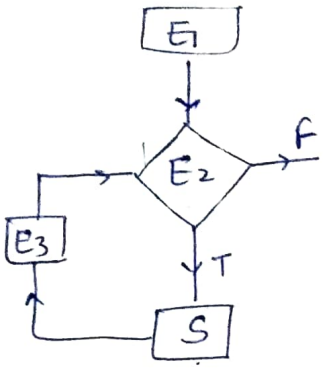
3-Add. code

It may fall in infite loop but our intention is just to understand the concept

L: if a<b goto L1
goto Last
L1: t = y+z
x = t
goto L

Last: Code other than while loop in programme.

for loop: for (E_1 ; E_2 ; E_3)
S



During execution of while loop copiler doesn't know we are executing while but compiler executes its prewritten 3-address code.

```
for (i=0; i<10; i++)
    a = b+c
```

```
    i = 0
L : if (i<10) goto L1
    goto Last
L1: t1 = b+c
    a = t1
    t = i+1
    i = t
    goto L
Last : Remaining code
```

Switch case:

```
Switch (i+j)
{
  case (i) : a = b+c;
             break;
  case (ii) : P = Q+R;
              break;
  default : x = y+z;
            break;
}
```

```
    t = i+j
    goto test
L1: t1 = b+c
    a = t1
    goto Last
L2 : t2 = Q+R
     P = t2
     goto Last
L3: t3 = y+z
    x = t3
    goto Last
test : if (t==1) goto L1
       if (t==2) goto L2
       goto L3
Last : Remain codes
```

2D array to 3-address code

A: 10 X 20 and we want to access

$x = A[y,z]$

$t_1 = y * 20$] → to compute position.

$t_2 = t_1 + z$

$t_3 = t_2 * (4)$] to compute base address of element

t_4 = base address of A

$x = t_4[t_3]$

This size (4 byte) is compiler dependent

Contn. next page

Explaination A[4][4]

	0	1	2	3
0	00	01	02	03
1	10	11	12	13
2	20	21	22	23
3	30	31	32	33

→ 2D array can be stored in m/m using two types

① Row major order

00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33

② Column major order

00 10 20 30 01 11 21 31 02 12 22 32 03 13 23 33

$x = A[2,3]$ (i, j) colum = 3 row = 2 but in memory the location somewhere else. So the code that converts Hight level code position to m/m location is 3-address code. we consider row major order.

$A[2,3] = (2 * 4 + 3) + 1 \rightarrow$ 12 position

(cross 11 element to get element at)

$A[i,j] = (i \times (\text{no. of col.}) + j) + 1$

→ Array of any dimension will get store in one dimension in memory.

Gate: 2007

In a simplified computer - - - - - - - - - - for this basic

① ADD R_1, R_2
$R_1 + R_2 \rightarrow R_2$

② ADD R_1, m
$R_1 + m \rightarrow m$

MOV	a, R_1
ADD	b, R_1
MOV	c, R_2
ADD	d, R_2
SUB	e, R_2
SUB	R_1, R_2
MOV	R_2, m

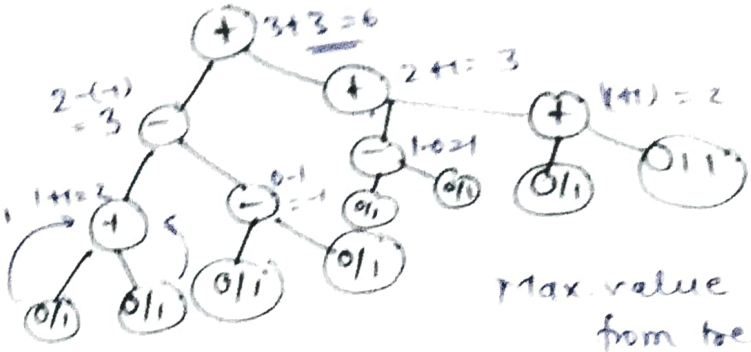
⇒ Total 3 move operations

Question Gate PYQ Book में 2 लगी है.

Gate-2004

Consider the grammar rule $E \rightarrow E_1 - E_2$ - - -

Given in videos must do on your own

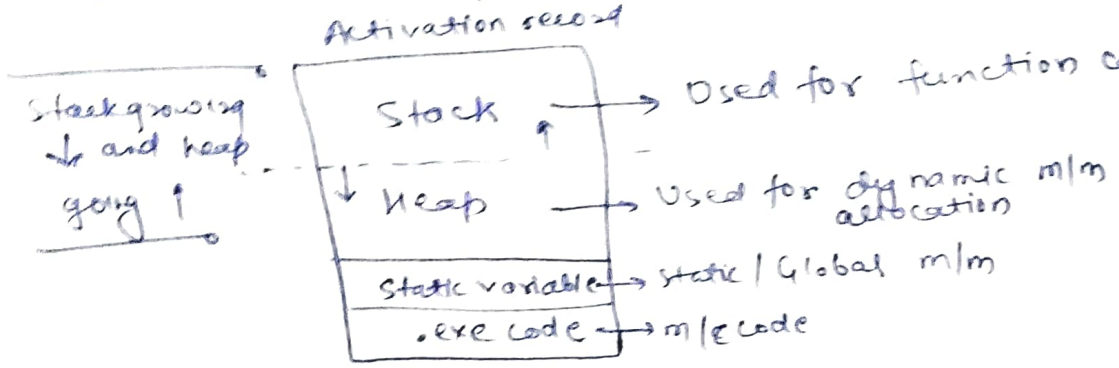


→ Remember the trick to solve this question.

Max value that can be obtain from tree = 6

Runtime - Environments

RTE means on the time of loading of programme to OS what kind of support you needed from OS.



Storage Allocation Strategies

1) Static : ① Allocation done at compile time.
② Binding do not change at run time.
③ One activation record for procedure.

V.table can be alive through out the program

Disadvantage : ① Recursion is not supported.
② Size of data object must be know to compile time.
③ Data structure can't be created / supported dynamically.

2) Stack : → whenever a new activation begin, activation record pushed on stack, and when activation ended it popped off.

→ Local variable are bound to fresh storage.

→ Dis : local variable can't return once the activation record popped off.

3) Heap : - Allocation and dellocation can be in any order.

Dis → Heap management is overhead. We can deal with overhead using first fit in OS.

Summary → Activation / function call can have —

i) Permanent lifetime in case of static allocation.

(ii) Nested lifetime in case of stack allocation.

(iii) Arbitrary lifetime in case of heap allocation.

Code optimization / Code generation (Not for Gate Exam)

→ Read it 'must' for the sake of knowledge

Introduction

Optimization (use to reduce no. of lines in code)

Machine Independent

1) loop optimization
   a) code motion or frequency reduction
   b) loop unrolling
   c) loop jamming
2) folding
   - constant propagation
3) Redendancy elimination
4) strength Reduction

Machine dependent

1) Register allocation
2) use of addressing modes
3) Peephole optimization
   a) Redundant load / store
   b) flow of control optimization
   c) strength reduction
   d) use of m/c idioms.

Loop optimisation and basic blocks

① To apply loop optimization, we must first detect the loops.

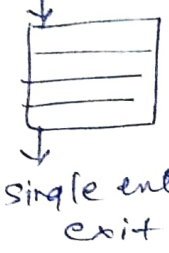
② For detecting loops we use control flow analysis (CFA) using program flow graph (PFG).

③ To find PFG we need to find basic blocks.

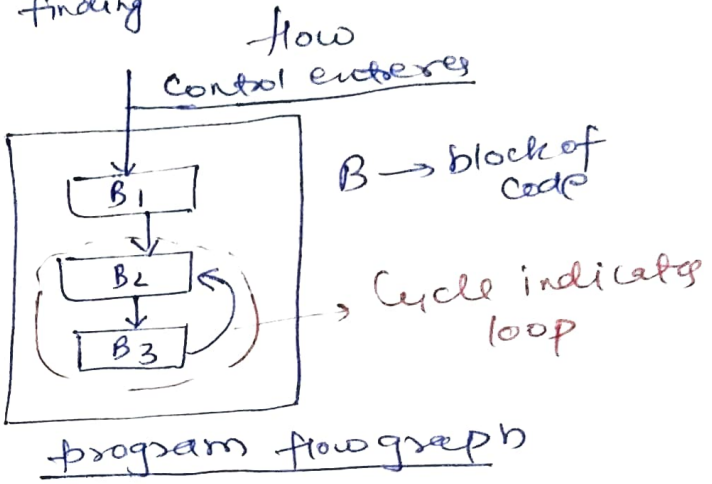
A block is sequence of 3 address statements where control enters at the beggining and leaves only at the end without any jump or halts.

HLL have (for loop, while loop) but when its concern to 3 address code then there is no visibility of loops and then it comes to process under code optimisation.

Basic Blocks → CFA (PFG) → loops finding → optimization applied



Single entry and exit point



program flowgraph

Algorithm to find the basic block.

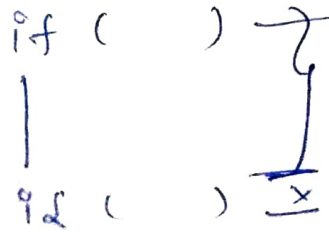
In order to find basic block, we need to find the leaders in program. Then a basic block will start from one leader to the next leader but not including next leader.

Identifying leaders in the block

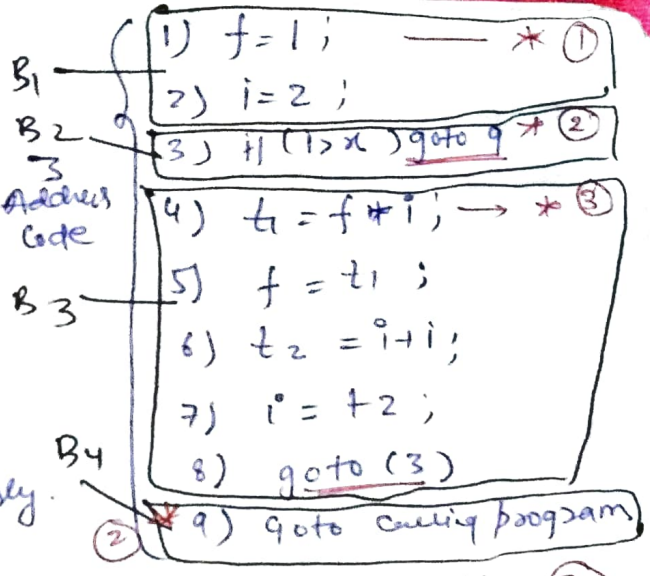
1) I statement is a leader
2) Statement that is target of conditional or unconditional statement is a leader.
3) Statement that follows immediately a conditional or unconditional goto statement is a leader.

(3) conditions

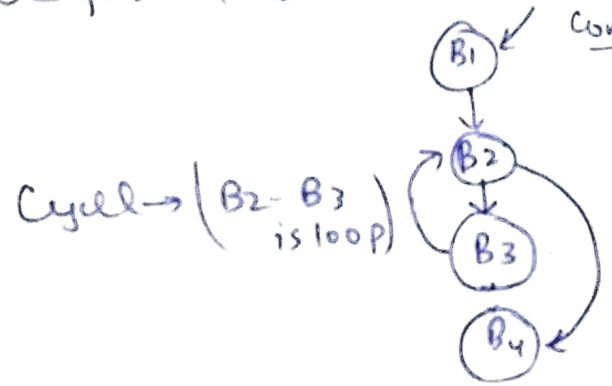
→ Block is a code between two leaders includes first leader statement only



Ex
```
fact(n)
{
  int f = 1
  for(i=2; i<=n; i++)
      f = f*i;
  return f;
}
```



We have 3 conditions to find leader as discussed previously.
So,
* ③ → means statement is leader due to condition ③
We get 4 basic blocks due to four leader.



→ This way we can identify the loop using (PFG)

Types of loop optimization

Loops can be optimize in two ways -
① either remove the loops
② reduce the lines in loop

① **Frequency Reduction**: Step after finding loops.
Move the code from high frequency region to low frequency region called code motion

ex-
```
i=0
while(i<5000)
{  A = sin(x) * i;
   i++;
}
```
⇒
```
t = sinx/cosx
while(i<5000)
   A = t * i
```

→ sinx/cosx eventime will have same value so calculate it once and save the calculation time

loop unrolling: → Reduce the no. of times we enters in loop & reduce the no. of comparison.

```
while(i<10)                while(i<10)
{ x[i]=0          ⇒        { x[i]=0
  i++                        i++
}                            x[i]=0
                             i++;
                           }
```

work on array that have size divisible by 2.

loop jamming:- Combine two loops together or reduce no. of loops.

```
for(i=0; i<10; i++)          for(i=0; i<10; i++)
  for(j=0; j<10; j++)        { for(j=0; j<10; j++)
    x[i,j]=0                   {
                                 x[i,j]=0
for(i=0; i<10; i++)              x[i,i]=0
  x[i,i]=0                     }
                             }
```

Machine Independent Optimization

folding: Replacing an expression that can be computed at compile time by its values.

Ex: $2+3+C+B = \underline{5+C+B}$.

Redundancy elimination: (DAG) [Use already evaluate values]

$A = B+C$

$D = 2+B+3+C$

$D = 2+3+A = \underline{5+A}$

Strength Reduction: Replace a costly operation by cheaper 1.

→ * and / are costly operations.

Ex $A = A*2$

$A = A<<1$

Algebric simplification

$A = A + 0$ } eliminate such statement bcoz they doesn't effect anything.

$x = x * 1$

Machine dependent optimization

① Register allocation
- local allocation
- global allocation

] Depends on how many register you have.

② Use of addressing modes

③ Peephole Optimisation

(a) Redundant load and store elimination

$x = y + z$

```
mov y, R0
add z, R0
mov R0, x
```

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

```
mov b, R0
add c, R0
mov R0, a
mov a, R0
add e, R0
mov R0, d
```

→ Load and store problem. So we can remove it.

(b) Flow of control optimization

- Avoid jumps on jumps
- eliminate dead code

L_1: jump L_4

L_2: jump L_3

L_3: jump L_4

```
#define x 0
if(x)
{
    ...
}
```

Dead code

→ bcoz x is never going to be 1.

never get execute

(c) use of machine idioms

$i = i + 1$

```
mov R0, i
add R0, 1
mov i, R0
```

⇒ inc i

Points to write / add additionally

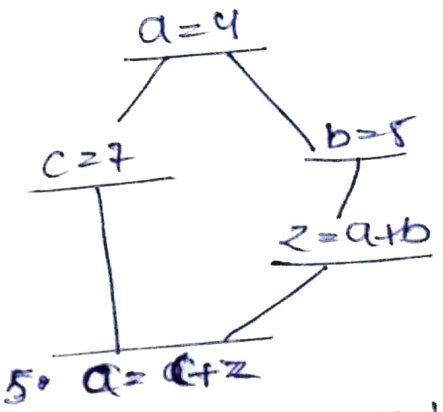
Liveness Analysis

→ Also called as live variable analysis.
→ Data-flow analysis to calculate the variables that are live at each point in a program

Adv. → Used in dead code elimination
Used for register allocation

① b = 3
↓ → b is live
c = 5
↓ → b & c are live here.
a = b + c

②



a → killed at 5 but we redifed at

Compiler Design

HLL → Preprocessor → Compiler → Assembler → Loader/Linker

Pure HLL — Assembly Language — Machine Code

HLL → (Character stream) Lexical Analysis → (tokens stream) Syntax Analyzer → ICG → TCG

Semantic Analysis

3 address code → CO (optional)

HLL → ICG : frontend

CO → TCG : Backend (System dependent)

Symbol table collect info. in analysis phase and use in synthesis phase.

Line of usage → Lines of code in which variable is used.

—— declaration → info. of line in which variable is declared.

Operations on symbol table depends on structure.

① Non block structured : Single instance of variable

② Block structured : multiple instance of variable

→ Complete string assumes as a single token.

Grammar → Left Recursive or right recursive

There is no direct method to verify ambiguity of grammar other than parse tree.

Ambiguity problem is undecidable. | Power is always right recursive.

Error recovery operations in lexical analysis: ① Delete ② Insert ③ Replace ④ Transpose

Ambiguity is generally due to —

① Associativity → Recursion ② Precedence. → level of operators.

Remove left recursion b/o2 Top-down parser doesn't support.

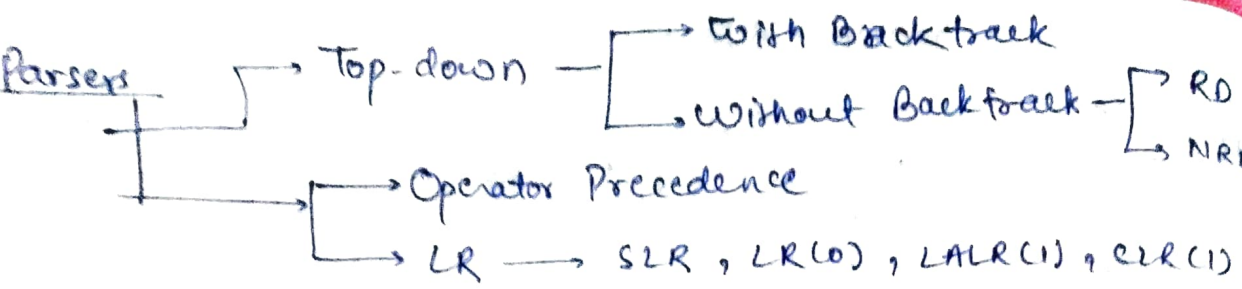
$A \rightarrow A\alpha \mid \beta$ then,

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

Non-deterministic grammar generates common prefix problem, to remove it we use left-factoring method.

Even after left-factoring we can't say ambiguity is removed.



RD → Recursive descent , NRD → Non-Recursive Descent

TDP without backtracking can be done by removing left recursion & non-determinism.

TDP → Left most derivation

Bottom-up → right most derivation in reverse

LL(1) → no look ahead

L → left to right

L → Leftmost derivation

First() → It can contain ∈.

follow() → It doesn't contain ∈ but contain $ in follow() of starting symbol.

For LL(1) grammar must not be left recursive.

A → ∈ kind of productions kept in follow() of variable and other in first().

It can't contains multiple entries in a single column.

→ No ambiguous grammar is LL(1).

RDP:- We make recursive fn for every variable.

To solve recursive fn we use Stack and concept of activation records.

Operator precedence grammar:

In this grammar, no two operands come together.

We need an operator precedence table.

No relation among id & id or $ & $.

→ It takes $O(n^2)$ size. So overhead

To avoid we use operator function tables.

→ If graph has no cycle then construct fn tables.

It takes $O(n)$ size.

But disadvantage, it converts blank entries to valid entries.

→ Low error detecting capabilities but we use it anyhow.

→ Right recursive → right associative

→ Left recursive → left associative

→ Some times we can convert normal grammar to operator precedence grammar.

→ Operator precedence grammar can't handle (-)ve sign and it is for small grammar only.

LR(0) & SLR(1) • simple LR.

In LR(0) make the grammar augmented first then perform closure and goto operations on grammar to for a canonical collections of LR(0) items.

finally form a LR(0) passing table will all states and Action and Goto parts

→ In follow():

If follow(X) = ∈ or Null then add follow(V), where V is variable present as LHS of production in which we are searching for follow of (V).

→ In LL(1) first(S), S → any symbol/variable doesn't contain any same terminal twice

→ It should not be left-recursive (LL(1)).

for LR(0), there should be no SR/RR conflicts

→ If grammar is LR(0) then it must be SLR(1) otherwise it can be SLR(1) only.

→ In SLR verify that next traced symbol should present inside the follow (LHS) of previous reduced production.

→ If augment state or accepting state have conflict then we didn't accept the conflict

→ In CLR(1) keep the look aheads with the state too and keep the reduced move in table corresponds to look aheads.

→ In LALR(1) we merge the same entries of table together.

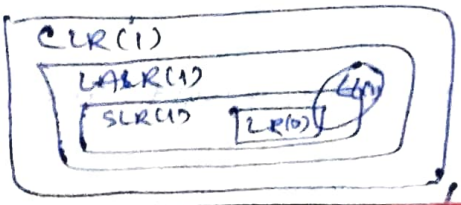
→ If a state has two reduce moves then it will work here if both have different look aheads.

If not CLR(1) then must not be LALR(1). If it is CLR(1) then it is not necessarily LALR(1).

No. of states:

CLR ≥ LR(0) = SLR(1) = LALR(1)

CLR(1) is most powerful among all.



→ SLR(1) detects the error earlier than LR(0).

→ In SR/RR conflict if both occur on same terminal or variable then neither SLR(1) nor LR(0). If both on different then it will be SLR(1) not LR(0)

→ If I_5 is reduced as r_1 and have production as $X \to y$ then we kept r_1 in follow(x). It avoids the conflicts a lot of times because in LR(0) we kept reduced moves blindly. It increases the chance of conflicts.

→ In CLR(1), it arises problem if there is a conflict at some state and both reduction moves have same look ahead.

In CLR(1): we get conflicts if.

$A \to \alpha . a / G$	a
$B \to r . / a, \$$	

Conflict SR on (a)

$A \to \alpha ., a$
$B \to \alpha ., a$

RR conflict on a

~~But try to remove this~~

→ In range from LR(0) → SLR(1) we try to reduce to reduce conflict and increase in error and blank space detecting capabilities.

In SLR(1) & LALR(1)

1. Goto and shift moves are always identicals
2. Reduce moves and error entries might get changed
3. Both have same no. of states.

If $S \to SS / a / \epsilon$ then I_0 will contains a SR conflict as

$S' \to .S$	S
$S \to .SS / .a / .\epsilon$	a

$S \to .\epsilon$ creates a reduced move

There is no shift move on $A \to .\epsilon$

→ YACC → O/p LALR(1). It resolves both SR & RR conflicts. SR resolved by shift move & RR resolved by first reduce move.

Syntax Direct Translation: Grammar + Semantic Rules.

→ Same grammar can behave different according to SDT

→ SDT gives same output on top down and bottom up both type of parsing.

→ Symbol that occur at lower level has high precedence.

→ Right recursive grammar → Right associativity of symbol and same for left recursion.

→ No. of reductions = no. of non-leaf in a tree.

→ SDT → Concrete Syntax Tree (Parse Tree)
→ Abstract Syntax Tree (Expression tree)

Types of attributes in SDT

Synthesise: $A \to BCD$: A.attr. = f(B.att, C.att, D.att).
Parent takes att. from their child.

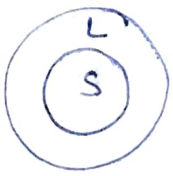
Inherit: $A \to BCD$: C.att = f(A.attr, B.att, D.att)
Child can take attribute from parents and any of its seiblings.

S attributed SDT

① Only Synthesized attr.

② Semantics can placed at last to
$A \to BC\{\}$

③ bottom up pasing



L-attributed SDT

① Both but inherit only from left & also synthesize.

② semantics can be placed anywhere
$A \to \{\}BC$, $A \to B\{\}C$,
$A \to BC\{\}$

⇒ Top-down, left → right

ICG: Types of 3 address Code

$x = y \text{ op } z$, $x = \text{op } z$ (unary) , $x = y$, if x (relop) y goto L,
goto L , $A[i] = x$ or $y = A[i]$, $x = \&p$, $y = *x$.

IC → Linear form → Postfix Addl. / 3 Addl. Code
→ Tree form → Syntax tree / DAG

Storage Allocation Strategies

① **Static**: Compile time allocation, no change at run time in binding, one activation record per procedure.

→ No recursion, size should be known at compile time, no dynamic data structure.

② **Stack**: Support recursion, local variable are fresh, local variable can't retain after popped off.

③ **Heap**: No order of allocation & deallocation, heap management is overhead, only support first fit in OS.

Activation record / fn call: Permanent lifetime in static, nested lifetime in stack, arbitrary in case of heap.

Code optimization or code generation

① **Machine Independent**: ① Loop Optimization [frequency reduction, loop unrolling, loop jamming], folding [Constant propagation], redundancy elimination, Strength reduction. $X=4, Y=2$ $Z=X+Y$, $T=Z+X+Y$, $T=\frac{Z+1}{2}$, algebraic simplification

② **Machine Dependent**: Register allocation, peephole optimization, [Redundant load/store, flow of control, use of m/c idioms].

Liveness Analysis / Constant propagation:

→ Lexical Analysis → it uses regular expression and finite automata

→ Type checking done at semantic analysis.

→ In L attribute SDT if all semantics rule are at end then we can use bottom up parsing.

→ Keywords of language are recognized during lexical phase

→ Code optimization (DAG), Abelian group (PDA), code generation (syntax tree).

→ $P \to \epsilon$ is also not an operator grammars

→ Control flow graph / flow graph both are same

→ character stream is input to lexical and tokens stream is input to syntactic phase.

→ CFG can be used to specify both lexical and syntax rules

→ In static single assignment form we use different variable register of same name at the time of assignment

→ In SR parsing stack contains only a set of viable prefix. Handle is actually on the top of the stack.
Viable prefix is prefix to handle so can never extend to right of handle (top of stack).

→ Lexical: DFA minimization, Parsing: Production tree, Register Allocation: Graph Coloring, exp. evaluation: postorder

→ SLR is very easy to implement.

(31)

→ Register spill: If all registers are full then transfer the content from register to memory.

→ In single-static we can't use same variable on LHS in expressions.

→ In CFG each variable (not registers) declare in separate state / block.

→ 3-address Code generation is a part of frontend. So no relation with backend. It is system independent.

→ In Grammar $S \to Sa \mid a$ [$S \to a$ is not a unit prodn]
but $S \to A$; $A \to a$ [$S \to A$ is unit production.]

→ Goto() always occur on non-terminals.

→ Two states in LALR gets merged if they have different look-aheads but same set of productions.

→ Some code-optimizations carried out on the intermediate code because it enhances the portability of compiler for other target processes.

→ LR(1) means CLR. LALR(1) will have SR conflict if only if CLR(1) have S-R conflicts

→ Every regular set has atleast one grammar which is unambiguous and LR(1).

→ By doing some modification in grammar, its language doesn't get changed.

→ Yacc gives o/p as LALR. It identified S-R conflict and resolve the conflict in favour of shift over reduce.

→ Handle is a substring that present in the RHS of substring. $S \rightarrow aB \mid a$, $B \rightarrow Bb \mid \epsilon$ [Only in Bottom up parsing]

String = abba
↳ a is handle in the stack

→ Viable Prefix : In shift/reduce parsing.
If string is not appearing on the stack then it will not be viable prefix.

Ex $S \rightarrow CC$; $C \rightarrow ac \mid b$

Str1 = ab

Stack	Prefix	
$	ab $	shift a
$a	b $	shift b
$ab	$	

$ab → viable

Str2 = bb

Stack	Prefix	
$	bb	shift b
$ b	b $	Reduce (→b)
$ C	b $	shift b
$ Cb	$	

$Cb → not bb

→ Global Variables are not present in the activation record of a procedure (C Acts / Control both linkage proc

→ Jumps are not allowed in the middle of basic block

→ DAG can be used to eliminate common sub expressions

→ White spaces removed in lexical phase, so not tokens

→ Leaf node of a tree is synthesized attributes.

→ In $B \rightarrow Bb \mid \epsilon$, first(B) = {b, ε}

→ In constant folding we did run time computations at compile time [Part of common subexpression]

→ In Chomsky Normal formal no. of nodes in parse tree for string of length $n = 2n-1$

→ In merging CLR → LALR, it might introduce R/R conflicts

→ In a string of length 'n' — no of subsequence (2^n), no. of prefix / suffix ($n+1$), [Including ∈] — no of proper prefix ($n-1$). [Excluding ∈ and string]

→ Every regular lang. doesn't follow LL(1).

→ CFL is closed under regular intersection.

→ Regular languages are closed under difference ($L_1 - L_2$) ($L_1 \cap \bar{L}_2$), reversal (DFA(L) $\xrightarrow{\text{reverse}}$ DFA(LR)), homomorphism, inverse homomorphism, quotient operation, substitution & infinite union.

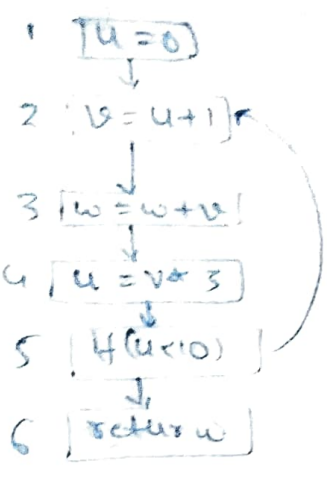
Rule 1 → $X = X + 3$; X is alive here.
$X = Y + 1$; X is dead here only Y is alive.

Rule 2 → In a line if variable is present at RHS, it is alive present at LHS follow Rule 1.

→ If you can reach 'nth' Instn in which variable without any definition then variable is alive.

	X	Y	Z	U	V	W
1	D	D	D	D	D	D
2	D	D	D	D	L	D
3	D	D	L	D	L	D
4	L	D	L	D	D	D
5	L	L	L	D	D	D
6	D	L	L	D	D	L
7	D	L	D	L	D	L
8	D	D	D	L	L	D

1. V = 1
2. Z = V + 1
3. X = Z * V
4. Y = X * 2
5. W = X + Z * Y
6. U = Z + 2
7. V = U + Y + W
8. return (V * U)



	u	v	w
1	D	D	L
2	L	D	L
3	D	L	L
4	D	L	L
5	L	D	L
6	D	D	L

→ flow of control remain same in both SSA and non-SSA form.

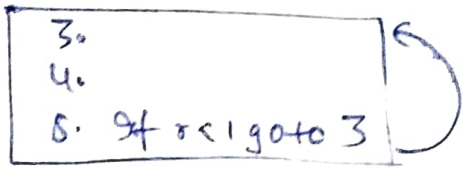
→ Let 'L' be a CFL. If every CFG with language $L = L(G)$ is ambiguous then L is said to be inherently ambiguous language.

→ Semantic errors occur at compile time.

→ CFG (parser) used in syntactic analysis. So it can capture matching curly braces, nested parantheses and syntax of if-else, but equality of datatype in RHS & LHS is work of semantic analysis.

→ RG (Lexical analysis): DFA, CFG (Syntax analysis) CSG (Semantic analysis).

→ We can move to same block at if as,



→ Also consider Entry and exit node inside CFG.

→ Automatic garbage collection is not essential to implement recursion.

→ The identification of common sub-expression and replacement of run-time computation by compile time computation is constant folding.

→ In DAG (a+a) can be represented as