	a	d	$	S	C
0	S36	S47		1	2
1			ACC		
2	S36	S47			5
36	S36	S47			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

SIVANI

01/04/2024 UNIT-3 (Part-1)

* Syntax Directed Definition (SDD)

→ Semantic analysis deals with having a proper meaning for any statement in a code.

→ To avoid errors related to type of the data used; type conversions are done. This is done with the help of Symbol table where the data is updated as and when new symbol ~~are~~ is seen in lexical analysis phase.

→ Functionalities of Semantic analysis

- → Checks if the O/P of syntax analysis is a meaningful parse tree or not.
- → type checking
- → type conversion

→ When we ~~attach~~ assign attributes to the context free grammar symbols; the generated grammar is called SDD.

→ attribute : Semantic rule

* $E \rightarrow E+T$, {$$ = $1 + $3} → Syntax Directed Translation (SDT)

$E \rightarrow E+T$, {E.type = E.type '+' T.type} → SDD

→ If we attach programming statements ~~in~~ to CFG; we get SDT.

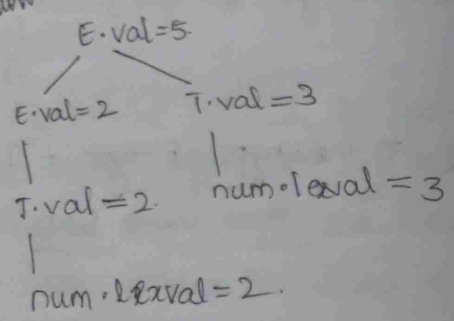
* **Types of attributes**
  → Synthesized attribute
  → Inherited attribute

* **Synthesized attribute** (S-attributed)
  for N.T. A at a parse tree node N is defined by a semantic rule associated with production at N.
  * attribute value at the children of N & at N itself.

Ex: E → E+T {E.val = E.val + T.val}
E → T {E.val = T.val}
T → id {T.val = id.lexval}

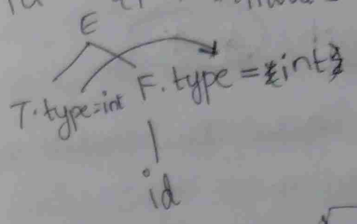
order of evaluation

bottom to top

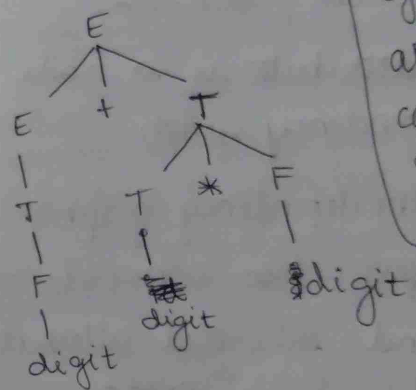


* **Inherited attribute:**
  attribute values at N's parent, N itself, and N's siblings.

Ex: int a;
<int>, <id> <;>
E → TF {T.type = F.type}
F → id {T → int|float}



* 3+4*5
<id> + <id> * <id>



* SDD that involves only Synthesized attributes is called S-attributed grammar.

→ If all the attributes are S-attributed then by default order of evaluation is from bottom to top.
→ While constructing annotated parse tree, we must avoid cyclic/circular dependency.

→ If circular dependency is present; such parse tree does not have any order of evaluation.

* Dependency graph: **

- → to ~~predict~~ give the order of evaluation.
- → depicts the flow of info. among the attribute instances in a particular parse tree.
- → Each attribute is a node in dependency graph.

S I V A N I

* → Evaluating the dependency graph order of where the parse tree consists of both synthesized and inherited attributes is same as topological sort.

* Write SDD to compute prefix expression which consists of addition & multiplication operators.

* L-attributed SDD

- → dependency graph edges can go from left to right; but not right to left.
- → if the inherited attributes are evaluated from parent (or) left siblings only; such SDD is L-attributed.
- → Each attribute must be either synthesized or inherited.

* SDD with side effects:

C-compiler allows only 1-D array where any other dimension is converted to 1D

* Applications of SDD:

→ Construction of Syntax trees.

→ Structure of a type

15/05/2024

* CODE OPTIMIZATION:

→ Optimizing the code deals with modifying the code ; so as the CPU utilization is more efficient.

S I V A N I

→ Levels of Optimization:

* Design level:
  - → to make best use of available resources.
  - → implementation of design will benefit ~~the~~ from the use of suitable algorithms.
  - → choice of algorithm effects efficiency more than anything.

* Compile level:
  - → optimizing compiler tends to ensure that the executable program is optimized atleast as much as compiler can predict.

* Assembly level:
  - → Writing a code using assembly language designed for a particular hardware platform will normally produce most efficient code.

* Run-time:
  - → run time compiler i.e. Just in time compiler is able to perform runtime optimization.

→ Criteria for optimization:

* Optimization must preserve the meaning of a program:
  - → cannot change O/P produced for any I/P.
  - → cannot introduce an error.

* Optimization ; on an average should speedup the program.

* Transformation should be worth the effort.

→ Improvements can be made at various phases:

* Source Code:
  - → Algorithm transformations
  - → Profiling can be helpful to focus.

* Intermediate Code:
  - → improve loops, procedure calls & address calculations
  - → optimizing compilers.

* Target Code:
  - compilers can use registers
  - Peephole transformation can be applied.

→ Types of Code optimization:
  - Common sub expression removal
  - Dead code optimization
  - Loop optimization.

SIVANI

*→ Common Sub-expression removal:
  - Searches for instances of identical expressions & analyses if it is worthwhile replacing with a single variable.

a = b*c+g ⟹ temp = b*c
d = b*c*d   a = temp+g
             d = temp*d.

*→ Dead code optimization:
  - removing a code that does not affect a program.
  - removing such code; reduces the size of the program & avoids execution of irrelevant code.

  - 2 types:
    - Unreachable Code
    - Redundant Statement.

→ Unreachable Code is the code that exists in the source code of program but can never be executed.

Ex:

```
if (a>b)                       if (a>b)
   m=a                            m=a
else if (a<b)        ⟹        else if (a<b)
   m=b                            m=b
else if (a==b)                 else
   m=0                            m=0.
else
   m=-1
```

→ Redundant Code is the code that is executed but no effect on the O/P of the program.

Ex:

```
main()
{ int a,b,c,r;
  a=5; b=6;
  c=a+b;
Redundant { r=2;
          { r++;   printf("%d", c); }
```

*→ Loop Optimization: (Independent)

→ Improve performance ~~using~~ by reducing overheads associated with loops.

→ can be done by removing

→ Loop invariant

→ Induction variables.

S I V A N I

→ Loop invariant:

Ex:

i = 1		i = 1
s = 0		s = 0
do{	$\Longrightarrow$	a = 5
s = s + i	Code motion	do
a = 5		{
i = i + 1		s = s + i
}		i = i + 1
while (i <= n)		}.
		while (i <= n)

→ Induction variables:

* One variable is used to prove other variable.

* Control Flow Graph:

→ Nodes represent basic blocks.

→ Initial node is the block whose leader is the first statement.

→ There exists an edge from B_i to B_j if:

* There is a conditional/unconditional jump from the last statement in B_i to the first statement in B_j.

* B_j immediately follows B_i in the order of the program and B_i does not end in an unconditional jump.

* Copy propagation:

→ Assignments of the form f := g are called copy statements.

→ Copy propagation often turns the copy statement into dead code.

* Loop Optimizations:

→ Running time of a program:

→ ↓se the no. of statements in an inner loop.

→ ↑se the no. of statements in the outer loop.

→ Code motion moves code outside of a loop.

→ Induction variables: variables that remain in lock-step.

→ Reduction in strength replaces a more expensive operation with a less expensive one.

→ Methods exist to recognise induction variables & apply appropriate transformations automatically.

S I V A N I

*→ Platform dependent techniques:

- → Peephole optimization
- → Instruction level parallelism
- → Cache optimization
- → Redundant resources.
- → Data level optimization

*→ Platform independent techniques:

- → Loop optimization
- → Constant folding
- → Constant propagation
- → Common sub-expression elimination

* 3-address code:

$$x = (a*b) + (c*d)$$

$$t_1 = a*b$$

$$t_2 = c*d$$

$$x = t_1 + t_2$$

*Peephole optimization: (Machine dependent)

↳ We apply this optimization only after converting complete source code to target code.

→ It is applied on small piece of code.

→ Repeatedly applied on the code.

1) Redundant load and store:

Ex: $a = b+c$

$d = a+e$

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

mov R_0, a ⇒ Redundant load & store.

mov a, R_0 → even if this is removed; code runs properly.

2) Strength reduction:

↳ certain operations ~~per~~ take a lot of time & memory ; such operations undergo reduction.

*Ex:

→ $x^2 \Rightarrow x*x$

→ $x*2 \Rightarrow$ left shift

→ $x/2 \Rightarrow$ right shift

3) Simplify the algebraic expressions:

$$\left.\begin{array}{l} a := a+0 \\ a := a*1 \\ a := a/1 \\ a := a-0 \end{array}\right\} \Rightarrow$$

these give ~~sim~~ the same o/p even if we remove this.

4) Replace slower transactions with faster:

Add #1,R ⇒ INC R

Sub #1,R ⇒ DEC R.

• Java byte code:

```
a load x      a load x
a load x  ⇒   dup
Mul           Mul
```

5) Dead code elimination:

→ unwanted code that does not effect the o/p of the code.

Ex:

```
int dead (void)
{ int a=10;
  int b=20;
  int c;
  c=a*10;
  return c;
  b=30;
  b=b*10;
  return 0; }
```

$\left.\begin{array}{l} b=30; \\ b=b*10; \\ return\ 0; \end{array}\right\}$ dead code.

* Loop optimization:

↳ Code motion: frequency reduction

where any program statement that executes a no. of times; can be minimized to less no. of times.

Ex:

```
a=100;
while(a>0)
{ x=y+z;
  if(a%x==0):
  { printf("%d",a); a--; } }
```

∴ the values remain same ⇒ the no. of executions of the statement can be reduced

↳ Loop fusion | jamming:

* combining 2 similar loops without changing the meaning.

Ex: int i, a[100], b[100]
```
for(i=0; i<100; i++)
    a[i]=1;
for(i=0; i<100; i++)
    b[i]=2;
```
→ combining these 2 loops

↳ Loop unrolling:

* avoiding the loop if it is not required.

Ex: for(i=0; i<5; i++)
```
    printf("Varun");
```
instead of loop give 5 printf statements.

* Principle Sources of Optimization:

→ Causes of Redundancy
→ Semantics preserving transformations
→ Global common sub expressions
→ Copy propagation
→ Dead-code elimination
→ Code Motion
→ Induction variables & reduction in strength.

* Introduction to data-flow analysis:

↳ Data flow analysis refers to a body of techniques that derive information about the flow of data along program execution paths.

→ Data flow abstraction:

* Execution of a program can be viewed as a series of transformations of the program state.

* Each execution of an intermediate code statement transforms an I/P state to a new O/P state.

* We must consider all possible sequences of program points through a flow graph that the program execution can take.

* Control flow graph:

→ How to find leader in basic block?
- → first statement is always leader.
- → Address of conditional, unconditional goto are leaders. goto(a) $\boxed{a \text{ is } L}$
- → Next line of conditional & unconditional

7. goto(a)
8.

goto are leaders.

→ First line of basic block is leader.

* Basic block is a set of instructions with a single entry & exit point

control enter
↓
Basic block
↓
control exit

* Basic blocks are fundamental units of control flow analysis & optimization.

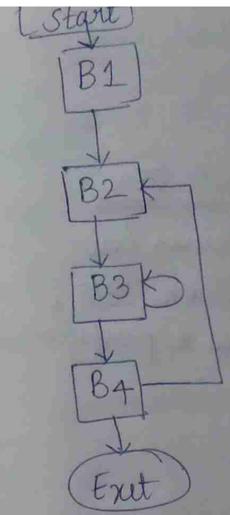
→ Consider the intermediate code:

L 1. i = 1 - B1
L 2. j = 1 - B2
L 3. $t_1 = 5 * i$ - B3
4. $t_2 = t_1 + j$
5. $t_3 = 4 * t_2$
6. $t_4 = t_3$
7. $a[t_4] = -1$
8. $j = j + 1$
9. if $j <= 5$ goto (3)
L 10. $i = i + 1$ - B4.
11. if $i < 5$ goto (2).

* Control graph is used to detect loop in the generated 3-address intermediate code.

B1 → 1
B2 → 2
B3 → 3 to 9
B4 → 10, 11

→ find the no. of nodes & edges in the control flow graph



⇒ 4 nodes.
5 edges.

⇒ if we add start & exit

⇒ 6 nodes, 7 edges.

→ Scope of optimization:

* Local
→ within basic block
→ Local common sub-expression elimination

* Global
→ between basic blocks
→ extended to entire control flow graph.

* Register allocation and assignment:

→ Assigning variables to processor registers to minimize memory accesses & improve performance.

→ Moving variables b/w registers & memory when there are not enough registers available to hold.

* CODE GENERATION:

*→ ISSUES:

→ Target Machine architecture:

↳ different machines have different instruction sets, addressing modes & performance characteristics.

→ Instruction Selection:

↳ deciding instructions to use from target machine's instruction set to implement high level language constructs efficiently.

→ Register Allocation:

↳ allocating registers to variables and managing spills to memory when the no. of variables exceeds the available registers.

→ Instruction Scheduling:

↳ Reordering instructions to minimize pipeline stalls & maximize instruction-level parallelism.

→ Error Handling: detecting & handling syntax errors (or) unsupported language constructs.

*→ Target Language:

→ Instruction set Architecture:

↳ set of instructions that target machine can execute.

↳ including their formats, addressing codes.

→ Assembly Language:

↳ human readable representation of machine code.

→ Abstraction level:

↳ balancing performance with readability & maintainability.

*→ Addresses in the target code:

→ Direct v/s Indirect addressing

⇓ memory addresses

⇓ pointers/indices.

→ Address Calculation:

↳ computing with the help base addresses, offsets & data sizes.

* RUNTIME ENVIRONMENTS:

→ A runtime environment is a set of data structures maintained at run-time to implement high level structures.

→ Runtime deals with the layout, allocation & deallocation of storage locations, linkages between procedures and passing parameters among other concerns.

*→ Storage Organization:

→ Target program runs in its logical space.

→ Size of generated code is usually fixed at compile time.

→ Compiler can place the executable at fixed addresses.

→ Runtime Storage:
- → Target Code.
- → Static data objects such as global constants.
- → Stack to keep track of procedure activations and local data.
- → Heap to keep all other information like dynamic data.

* Strategies for Storage Allocation:

- → Static allocation
  - ↳ memory allocation at compile time.
- → Dynamic allocation
  - ↳ Storage allocation is done at the time of program running.
  - → Stack allocation
  - → Heap allocation

→ STACK	→ HEAP
* allocation / deallocation is automatic	* allocation/ deallocation is explicit.
* less expensive	* more expensive
* Space for allocation is limited	* challenge is fragmentation.

→ STATIC	→ DYNAMIC
* Variable access is fast.	* Variable access is slow.
* addresses are known at compile time	* Redirection through stack/heap pointer.
* Cannot Support recursion	* supports recursion

*PROCEDURE CALLS:

→ Procedure defⁿ is a declaration that associates an identifier with a statement
- → formal parameters appear in declaration
- → actual parameters when a procedure is called.

→ Defines critical interfaces among large parts of a software

→ Each procedure has its own private named storage (or) name space.

→ Abstractions provided by procedures:
- → Control abstraction
- → Name space
- → external interface.

*Control Abstraction:

→ Language has rules to
- → invoke a procedure
- → Map a set of arguments from caller's name space to the callee's name space.
- → Return control to caller & continue execution

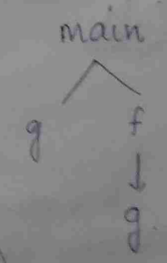
→ Each execution of a procedure P is an activation of the procedure P.

* The lifetime of an activation P is all the steps to execute P including all the steps in procedures that P calls.

* Activation tree: (follows DFS)

→ Depicts the way control enters & leaves activations.
- * Root : activation of main
- * Each node represents activation of a procedure
- * Node a is the parent of b if control flows from a to b.
- * Node a is to the left of b if lifetime of a occurs before b

Ex:
```
int g(){ return 42;}
int f(){ return g();}
main()
{ g();
  f();}
```



* CONTROL STACK:

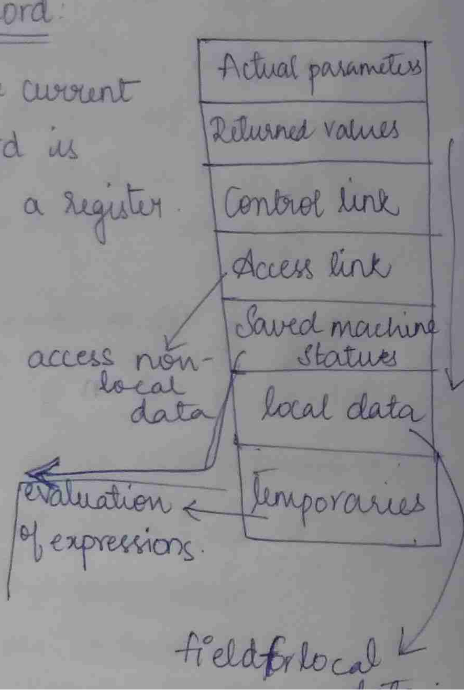
- → Manages procedure calls & returns.
- → Run-time stack.
- → Each live activation has an activation record on stack and also called a frame.
- → frame is pushed when activation begins and popped when activation ends.

→ Every activation record has either finished executing or is an ancestor of the current activation record.

→ Activation record:

* A pointer to the current activation record is maintained in a register.

*Fields:



* If a procedure F calls G; then G's activation record contains information about both F and G.
  - → G's activation record contains
    - → G's return value (needed by F)
    - → Actual parameters to G (supplied by F)
    - → Space for G's local variables.

→ Division of tasks between caller and callee:

* Call sequence:

- *→ Caller evaluates the actual parameters.
- *→ Caller stores a return address & the old value of top-stack into the callee's activation record.
- *→ Caller then ↑ments top-stack
- *→ Callee saves the register values & other status information

→ Communication b/w procedures:

* Types of mapping conventions
  - → Pass by value
  - → Pass by reference
  - → Pass by name.

Call by Value	Call by Reference	Call by Name
→ Caller evaluates the actual parameters & passes their r-values to the callee.	→ Convention where the compiler passes an address for the formal parameter to callee	→ Reference to a formal parameter behaves as if actual parameter had been textually substituted in its place.
→ Formal parameter in the callee is treated like a local name.	→ Redefinition of a reference formal parameter is reflected in corresponding actual parameter	→ Actual parameters are evaluated inside the called function
→ Any modification of a value parameter in the callee is not visible in caller.	→ formal parameter requires an extra indirection.	

* LEXICAL & DYNAMIC SCOPING:

→ A variable that a procedure refers to and that is declared outside the procedure's own scope is called a free variable.

* Lexical Scoping: a free variable is bound to the declaration for its name that is lexically closest to use.

* Dynamic Scoping: a free variable is bound to the variable most recently created at runtime.

→ Allocating Activation Records:

* Stack Allocation:
  - → activation records follow LIFO ordering
  - → Eg: Pascal, C, Java.

* Heap Allocation:
  - → Needed when a procedure can outlive its caller.
  - → Garbage collection support eases complexity.
  - → Eg: Implementations of Scheme and ML

* Static Allocation:
  - → Procedure P cannot have multiple active invocations if it does not call other procedures.

→ Variable length data on the stack:

* Possible to allocate variable-sized local data on the stack.
* Data may be local to a procedure

but the size may not be known at compile time.

* Access to non-local data in Nested procedures:

→ 2 common strategies:

* Access Links:

→ Suppose procedure p is nested immediately within procedure q.

→ Access link in any activation of p points to the most recent activation of q

*→ Static link: A pointer to the activation record of the lexically enclosing function.

* Displays:

* An array of pointers where each entry points to the activation record of a corresponding lexical level.

* Heap Management:

→ Heap is used for allocating space for objects created at runtime.

→ Interface to the heap:

- → allocate(size) & free(addr)
- → ~~malloc/fr~~
- → malloc() | free() in C

(or) new | delete in C++

* allocation & deallocation

- → manual (C/C++)
- → semi-automatic (Java)
- → fully automatic (Lisp)

→ Goals of Heap Management:

- → Space efficiency: minimize fragmentation
- → Program efficiency: take advantage of locality of objects in memory & make the program run faster
- → Low overhead: allocation & deallocation must be efficient.

* First fit allocation
  - allocate(k) & free(addr)

* Reducing fragmentation :
  - Merge free blocks
  - other variants :
    - best-fit : smallest possible chunk.
    - next fit : fit the object in the recent split

* Garbage collection is support for implicit deallocation of objects that reside on the runtime heap.

---

* SYMBOL TABLE :
  - data structures that is created & maintained by compilers for information storing regarding the occurrence of various entities.

→ used by both analysis & synthesis parts of compiler.

* Purpose :
  → to store the names of all entities in a structured form.
  → to verify if variable has been declared.
  → to determine the scope of a name.

* Information stored in symbol table :
  → name
  → attribute : reserved word, variable name, type name, procedure name & constant name.
  → data type
  → the block level
  → scope
  → offset from base pointer.

* Symbol table can be implemented as:
  - Unordered list
  - Linear list
  - Binary Search tree
  - Hash table

* Entry format :
  - <symbol name, type, attribute>

* Operations :
  - insert() :
    - used by analysis phase
    - adds information
    - format depends on compiler we use.
    - Ex: int a;
      → insert(a, int)

→ lookup():

- → Search a name in the symbol list.
- → lookup(symbol).
- → returns 'zero' if symbol does not exist in the table.
- if symbol exists in the table; it returns the attributes stored.

* → Scope Management:

→ Multiple block levels of symbol tables.

- → Level 0: a null hash table.
- → level 1: keyword in hash table.
- → level 2: global symbol table; that can be accessed by all procedures.
- → level ~~4~~ 3: Scope symbol tables created for each scope.

→ Hierarchical structure:

Global Symbol table.

- Pro_one symbol table
- Pro_two ST

→ Level-3 → Level-2 → Level-1 → Level-0

- Level-3 (Hash table of locals)
- Level-2 (Hash table of globals)
- Level-1 (Hash table of keywords)
- Level-0 (Null)

* → Hash Tables

- → is a list which each member is accessed through a key.
- → function that produces a location from the key is called hash fn.
- → capacity of table = size of bucket * no. of buckets.

→ too many buckets : Space inefficiency

→ too few buckets : time inefficiency.

* SYNTAX DIRECTED TRANSLATION

→ A syntax directed definition (SDD) specifies the values of attributes by associating semantic rules with the grammar productions.

→ Production:

$E \longrightarrow E_1 + T$

Grammar + Semantic Rule

⇓

SDT

Semantic Rule:

$E.code = E_1.code \,||\, T.code \,||\, '+'$

→ A syntax directed translation scheme embeds program fragments called semantic actions within production bodies

$E \longrightarrow E_1 + T$ {print '+'}

→ 2 types of syntax directed translations

- → L-attributed
- → S-attributed.

* → Syntax Directed Definitions:

→ A SDD is a context free grammar together with attributes & rules.

→ attributes: grammar symbols
rules: productions.

→ Applications of SDT:

* Executing arithmetic expressions.
* Conversion from infix to postfix & prefix.
* Conversion from binary to decimal
* Counting no. of reductions
* Creating syntax tree.
* Generating intermediate code.
* Type checking
* Storing type info into symbol table.

* Example:

S → S # A | A {S.val = S.val * A.val}

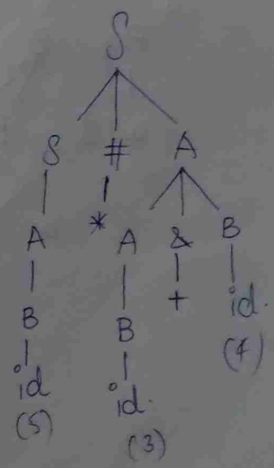
A → A & B | B. {A.val = A.val + B.val}

B → id {B.val = id.lval}

→ 5#3&4

= 5 * 3 + 4

= 19



* Parsing SDT:

Ex: $S \rightarrow AS \{pf(1)\}$

$S \rightarrow AB \{pf(2)\}$

$A \rightarrow a \{pf(3)\}$

$B \rightarrow bC \{pf(4)\}$

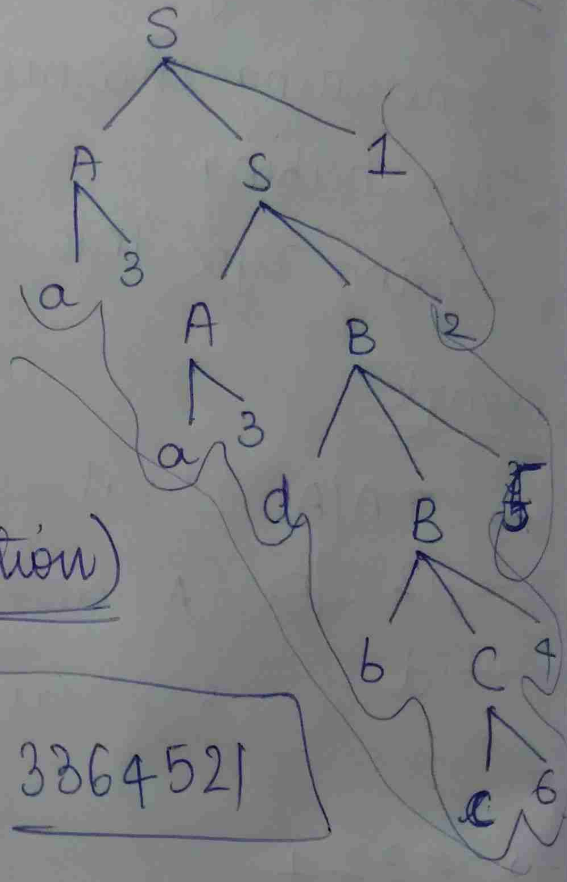
$B \rightarrow dB \{pf(5)\}$

$C \rightarrow c \{pf(6)\}$

I/P: aadbc

O/P: ?

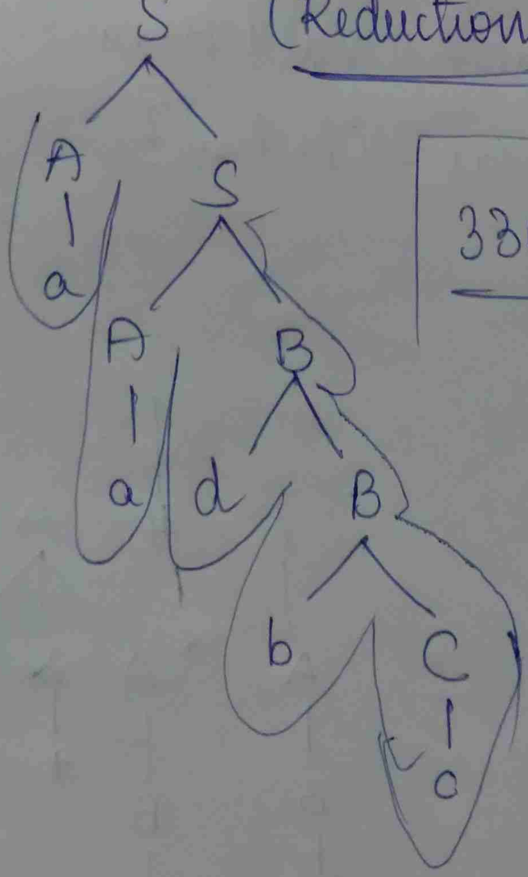
top-down: bottom-up



O/P: 3364521

Bottom-Up

(Reduction)



$\boxed{3364521}$