CD UNIT-3

Syntax-Directed Translations:

- In syntax directed translation, along with the grammar we associate some informal notations & these notations are called as semantic rules.

> Grammar + semantic actions = SDT (syntax directed translation)

- In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by semantic rules associated with the production rule.
- In the semantic rule, attribute is VAL & an attribute may hold anything like a string, a number, a memory location & a complex record.
- In syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Eg: $E \Rightarrow E + T$ { production → print "+" in SDT

$E.val := E.val + T.val$ { semantic Rule.

(1) Syntax-Directed Definitions (SDD's):

- Syntax directed definition is a kind of abstract application.

SDD = CFG + Semantic Rules

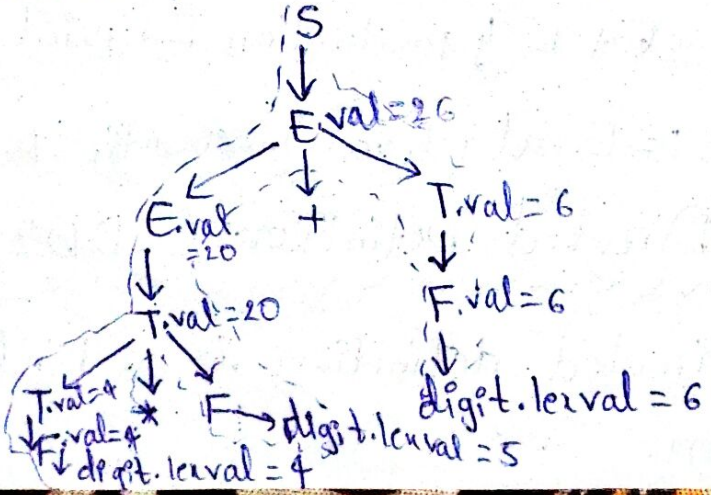
- A SDD is a content free grammar together with semantic rules.
- Attributes are associated with grammar symbols & semantic rules are associated with productions.
- If 'X' is a symbol & 'a' is one of its attribute then, X.a denotes value at node 'X'.
- Attributes may be numbers, strings, references, datatypes etc.,

Production	Semantic Rule
$S \to E$	S.val = E.val
$E \to E_1 + T$	E.val := E_1.val + T.val
$E \to T$	E.val = T.val
$T \to T_1 * F$	T.val = T_1.val * F.val
$T \to F$	T.val = F.val
$F \to$ digit	F.val = digit.lexval

Eg: Input string 4*5+6 - Write the annotated parse tree.

Annotated parse tree: The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

→ Now Annotated parse tree for input string 4*5+6 is:



Types of attributes:-

i) Synthesized attributes: If a node takes value from its children then it is synthesized attribute.

Eg: $A \rightarrow BCD$, A be a parent node
B, C, D are children nodes.

$A.S = B.S$
$A.S = C.S$
$A.S = D.S$
} parent node A taking value from its children B, C, D. S - attribute (synthesized)

ii) Inherited Attributes: If a node takes value from its parent or siblings.

Eg: $A \rightarrow BCD$

$C.i = A.i$ → 'A' is parent node; 'i' inherited attribute
$C.i = B.i$ → 'B' is sibling
$C.i = D.i$ → 'D' is sibling

Types of SDD:

i) S-Attributed SDD:

A SDD that uses only synthesized attributes is called as S-Attributed SDD

Eg: $A \rightarrow BCD$

$A.S = B.S$
$A.S = C.S$
$A.S = D.S$

Semantic actions are always plays at right end of the production. It is also called as "positive SDD".
Attributes are evaluated with bottom-up parsing.

ii) L-Attributed SDD:

A SDD that uses both synthesized & inherited attributes is called as L-Attributed SDD but each inherited attribute is restricted to inherits from parent or left sibling only.

Eg: $A \to XYZ$ { $Y.s = A.s$ ✓, $Y.s = X.s$ ✓, $Y.s = Z.s$ ✗ }

. Semantic actions are placed anywhere on R.H.S

. Attributes are evaluated by traversing parse tree depth first, left to right order.

(2) Evaluation Orders for SDD's:

. The evaluation order to find attribute values in a parse tree using semantic rules can be easily obtained with the help of dependency graph.

. While annotated parse tree shows the values of attributes, a dependency graph helps us to determine how those values can be computed.

Dependency Graphs(i) The approaches are:

Def: A graph that shows the flow of information which helps in computation of various attribute values in a particular parse tree is called dependency graph.

. An edge from one attribute instance to another attribute instance indicates that the attribute value of the first is needed to compute value of the second.

	Production	semantic Rule
Eg:	$E \to E_1 + T$	$E.val = E_1.val + T.val$

. The attribute value of E is obtained from its children E_1 and T.

Dependency graph is:

partial dependency graph.

. In the above figure, the dotted lines along with nodes connected to them represent the parse tree. The shaded nodes represented as "val" with solid arrows originating from one node & ends in another node is the dependency graph.

complete dependency graph:

Eg:- $T \rightarrow T * F$

$F \rightarrow$ digit

After removing left recursion.

$T \rightarrow FT'$

$T' \rightarrow * FT_1'$

$T' \rightarrow \epsilon$

$F \rightarrow$ digit

(ii) Ordering the evaluation of attributes:

. The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various needs of a parse tree.

. If the dependency graph has and edge from node M to node N, then attribute corresponding to M must be valuated before the attribute of N.

. Thus, the allowable orders of evaluation of the sequences of nodes is: $N_1, N_2, \ldots, N_k$ such that if there is an edge of the dependency graph from N_i to N_j then $i < j$.

. Such an ordering a directed graph into a linear order, & is called a topological sort of the graph.

- If there is any cycle in the graph, then there are no topological sorts i.e there is no way to evaluate the SDD on this parse tree.
- If there are no cycles, however, then there is always at least one topological sort.
- To see why, since there are no cycles, we can surely find a node with no edge entering.
- For if there were no such node, we could proceed from predecessor to predecessor until we came back to some node we had already seen, yeilding a cycle.
- Make this node the first in topological order, remove it from dependency graph & repeat the process on the remaining nodes.

Eg: Topological sort for above graph 1 to 9 numbered.

1st possibility: 1, 2, 3, 4, 5, 6, 7, 8, 9
(1: leaf, 2: F.val, 3,4: F.inh, 5,6: T.syn, 7: final T.val)

2nd - 1, 3, 5, 2, 4, 6, 7, 8, 9
(1: left, 3,5: left, 2,4,6: right, 7,8: syn, 9: final)

iii) S-Attributed definitions:

- It is very hard to tell whether the dependency graphs have cycles. So, it can be implemented using classes of SDD's that guarantee an evaluation order, since they do not permit dependency graphs with cycles:
  a) S-attributed definition
  b) L-attributed definition
  } two classes of SDD's

a) S-attributed definition: 1st class of SDD.

. In this, every attribute is synthesized.

. In this SDD, each semantic rule computes the attribute value of a non-terminal that occurs on LHS of the production that represent head of production with the help of attribute values of non-terminals on RHS of production that represent body of production.

Eg:- $S \to E_n$ $S.val = E.val$ — s-attributed (synthesized)

$E \to E_1 + T$ $E.val = E_1.val + T.val$ - s-attributed

$E \to T$ $E.val = T.val$ - s-attributed.

∴ All these as s-attributed, so SDD is known as s-attributed.

. s-attributed definitions can be implemented during bottom-up parsing, since a bottom-up passer corresponds to a postorder traversal.

b) L-attributed definition: 2nd class of SDD.

. The idea behind this class is that, b/w the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left. (Hence 'L-attributed')

. An L-attributed has synthesized or inherited attributes with following any one of the rules: ($A \to X_1 X_2 \ldots$ (X.a))

i) Inherited attributes associated with the head A.

ii) Either inherited or synthesized attributes, located to left of X_i

iii) Inherited or synthesized associated with this occurrence of X_i itself, but with no cycles in graph.

Eg:- $T \rightarrow FT'$ $T'.inh = F.val$ - Inherited

$T.val = T'.syn$ - synthesized.

4th approach

iv) Semantic Rules with Controlled side Effects:

- In practice, translations involve side effects:
- With SDD's, we strike a balance b/w attribute grammars & translation schemes.
- Attribute grammars have no side effects & allow any evaluation order consistent with the dependency graph.
- Translation schemes impose left-to-right evaluation & allow semantic actions to any program fragment.

Attribute grammar:

- An SDD without any side effects is called attribute grammar.
- The semantic rules in this define the value of an attribute purely in terms of values of other attributes & constants.

→ Side effects in SDD's can be controlled in one of the following ways:

- Permitting side effects when attribute evaluation based on any topological sort of the dependency graph produces a correct translation.
- Impose constraints in the evaluation order so that same translation is produced for any

allocable order.

Eg: $E \to En$ $\quad S.val = E.val$

Let us modify to print a result. Instead of rule $s.val = E.val$, which saves the result in synthesized attribute s.val.

Now $S \to En$ $\quad$ print (E.val)

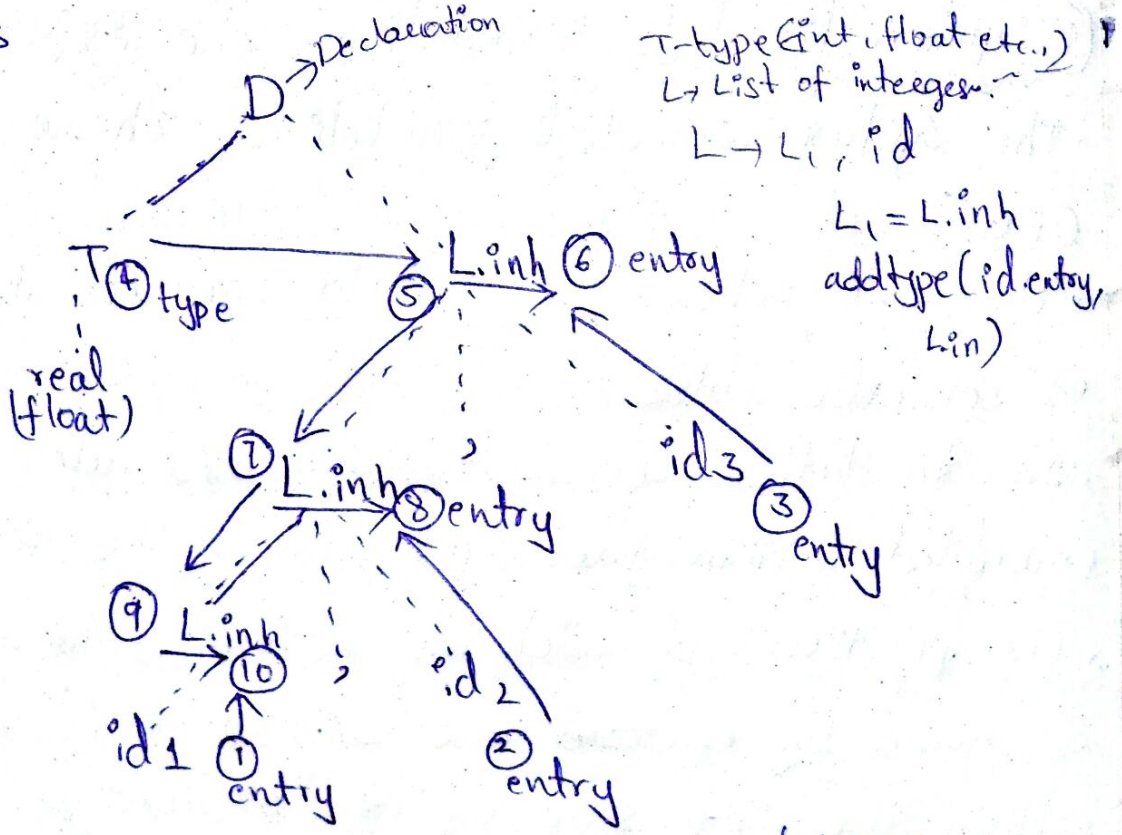
. Semantic rules that are executed for their side effects, such as print (E.val), this will be treated as dummy synthesized attributes.

. The print statement is executed at the end, after the result is computed into E.val.

In exam; Eg:

A dependency graph for input string - float id_1, id_2, id_3

sol



∴ O/p: float id_1, id_2, id_3

(6, 8, 10 dummy entries.

(3) Applications of syntax-directed translation (SDT)

. SDT is used for executing arithmetic Expression

Eg:- 4*3+5 etc.,

. In the conversion from infix to postfix expression.

. In the conversion from infix to prefix expression.

. It is also used for binary to decimal conversion

. In counting no. of reduction.

. In creating a syntax tree

. SDT is used to generate intermediate code.

. In storing information into symbol table.

. SDT is commonly used for type checking also.

(4) Syntax-directed translation schemes:-

. The syntax directed translation scheme is a CFG.

. The SDT scheme is used to evaluate the order of semantic rules.

. In translation scheme, the semantic rules are embedded within the right side of the productions.

. The position at which an action is to be executed is shown by enclosed b/w braces. It is written within the right side of the production.

Eg:-

Production	Semantic Rules {
S → E$	{ print E.val }
E → E+E	{ E.val := E.val + E.val }

(SDT)
ssion

$E \rightarrow E * E$	$\{E.val := E.val * E.val\}$
$E \rightarrow (E)$	$\{E.val := E.val\}$
$E \rightarrow I$	$\{E.val := I.val\}$
$I \rightarrow I$ digit	$\{I.val := 10 * I.val + Le.val\}$
$I \rightarrow$ digit	$\{I.val := le.val\}$

ession.

Implementation of SDT:

ession

. Syntax directed translation is implemented by construction a parse tree & performing the actions in a left to right depth first order.

. SDT is implementing by parse the input & produces a parse tree as a result.

lso.

Eg: Input string $4 * 5 + 6$ as before we implemented (Refer Topic-1)

a

(5) Implementing L-Attributed SDD's:

rder

. The following methods do translation by traversing a parse tree:

e

1) → Build the parse tree and annotate. This method works any nonciculal SDD.

tions.

→ Build the parse tree, add actions, and execute the actions in preorder. This approach works for any L-attributed definition.

ecuted

ten

→ Use a recursive-descent parser (top-down) with one func^n for each non-terminal. The func^n for non-terminal 'A' receives the inherited attributes of 'A' as arguments & returns the synthesized attributes of 'A'.

l }

→ Generate code on the fly, using a recursive-descent parser.

→ Implement an SDT in conjuction with an LL-parser. The attributes are kept on the parsing stack, & the rules fetch the needed attributes from known locations on the stack.

→ Implementing an SDT in conjuction with an LR-parser.

i) Translation During Recursive-Descent Parsing: 3rd pt of above arrow heads.

. In the body of func^n A, we need to both parse & handle attributes:

→ Decide upon the production used to expand A.

→ check that each terminal appears on the input when it is required. We shall assume that no back-tracking is needed, but the extension to recursive-descent parsing with backtracking can be done by restoring the input pos^n upon failure.

→ Preserve, in local variables, the values of all attributes needed to compute inherited attributes for non-terminals in the body or synthesized attributes for the head non-terminal.

→ Call func^{n}s corresponding to non-terminals in body of selected production, providing them with the proper arguments. Since the underlying SDD is L-attributed, we have already computed these attributes & stored them in local variables.

Eg: $S \rightarrow$ while (C) S_1

Here, S is non-terminal that generates all kinds of statements, including if-statements, assignment statements, & others. In this example, 'C' stands for a conditional expression - a boolean expression that evaluates to true or false.

SDD for while statements:

Productions	Semantic rules / actions
$S \rightarrow$ while (C) S_1	{ L_1 = new(); L_2 = new(); S_1.next = L_1; C.false = S.next; C.true = L_2; } S.code = label || L_1 || C.code || label || L_2 || S_1.code

SDT for while statements:

Productions	Actions
$S \rightarrow$ while (C) S_1	{ L_1 = new(); L_2 = new(); C.false = S.next; C.true = L_2; } { S_1.next = L_1; } { S.code = label || L_1 || C.code || label || L_2 || S_1.code; }

Implementing while-statements with a recursive-descent Parser:

```
string s(label next) {
    string scode, ccode; /* local variables
                            holding code fragments */
```

```
label L1, L2; /* local labels */
if (current input == token while)
    { advance input;
      check '(' is next on the input, and
                                  advance;
      scode = S(L1);
      return("label" || L1 || Ccode || "label" || L2 ||
                                  scode);
    }
else /* other statement types */
```

ii) On-the-Fly Code Generation:-

. The construction of long strings of code that are attribute values, is undesirable for several reasons, including the time it could take to copy or move long strings.

. In common cases such as our running code generation example, we can instead incrementally generate pieces of code into an array or output file by executing actions in an SDT.

Eg:- On-the-fly recursive-descent code generation for while-statements:

. We can modify func^{n} of while-statements to emit elements of the main translation 's.code' instead of saving them for concatenation into a return value of 's.code'. The revised func^{n} 's' shown below:

```
void S(label next)
  { label L1, L2; /* local labels */
```

```
if(current input == token while)
  { advance input;
    check '(' is next on the input, & advance;
    L1 = new();
    L2 = new();
    print("label", L1)
    c(next, L2);
    check ')' is next on the input, and advance;
    print("label", L2)
    c(next, L2);
    s(L1);
    }else /* other statement types */
  }
```

SDT: SDT for on-the-fly code generation for while statements.

Productions	Actions
$S \rightarrow$ while (	{ L_1 = new(); L_2 = new(); C.false = S.next; C.true = L_2; print("label", L_1); }
C)	{ S_1.next = L_1; print("label", L_2); }
S_1	

iii) L-Attributed SDD's and LL Parsing:-

- We can perform the translation during 'LL' parsing by extending the parser stack to hold actions & certain data items needed for attribute evaluation. Typically, the data items are copies of attributes.

• In addition to records representing terminals & non-terminals, the parser stack will hold action-records representing actions to be executed & synthesize-records to hold synthesized attributes for non-terminals.

• We use the following 2 principles to manage attributes on the stack:

→ The inherited attributes of a non-terminal 'A' are placed in stack record the represents the non-terminal. The code to evaluate these attributes will usually be represented by an action-record immediately above the stack record for 'A'. The conversion of L-attributed SDD's to SDT's ensure that the action-record will be immediately above 'A'.

→ The synthesized attributes for a non-terminal 'A' are placed in separate synthesize-record that is immediately below the record for 'A' on the stack.

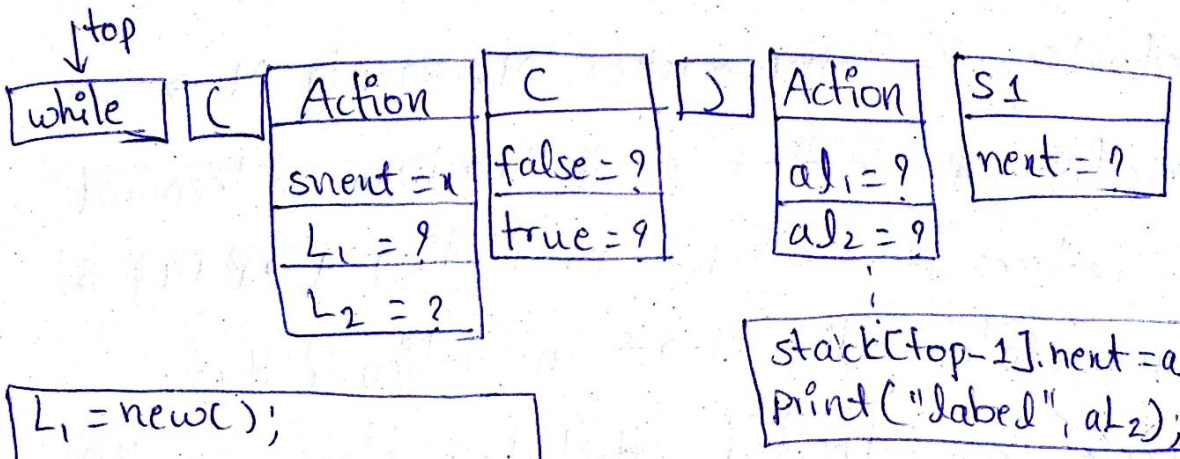
• This strategy places records of several types on the parsing stack, trusting that these variant record types can be maneged properly as sub-classes of a "stack-record" class.

• Actions-records contain pointers to code to be executed. This may also appear in synthesize-records These actions typically place copies of synthesized attribute in other records further down the stack, where the value of attribute will be needed after

the synthesize-record & its attributes are popped off the stack.

Eg: Expansion of 'S' according to the while statement production.

$S \rightarrow$ while $(C) S_1$



iv) Bottom-up parsing of L-attributed SDD's:-

. We can do bottom-up every translation that we can do top-down. More precisely, given an L-attributed SDD on an LL grammar, we can adapt the grammar to compute the same SDD on the new grammar during an LR parse. The "trick" has three parts:

→ start with the SDT constructed earlier, which places embedded actions before each non-terminal to compute its inherited attributes & an action

at the end of the production to compute synthesized attributes.

→ Introduce into the grammar a marker non-terminal in place of each embedded action. Each such place gets a distinct marker, & there is one production for any marker M, namely M → ∈.

→ Modify the action 'd' if marker non-terminal 'M' replaces it in some production A → α {a} B, & associate with M → ∈ an action 'a' that.

a) Copies, as inherited attributes of M, any attributes of 'A' or symbols of that action 'a' needs.

b) Computes attributes in the same way as 'a', but makes those attributes be synthesized attributes of 'M'.

Eg: A → B C in LL grammar & the inherited attribute 'B.i' is computed from inherited attribute 'A.i' by some formula B.i = f(A.i).

i.e, the fragment of an SDT we care about is A → { B.i = f(A.i); } B C

we introduce marker M with inherited attribute M.i and synthesized attribute M.s. The former will be copy of A.i & the latter will be B.i.

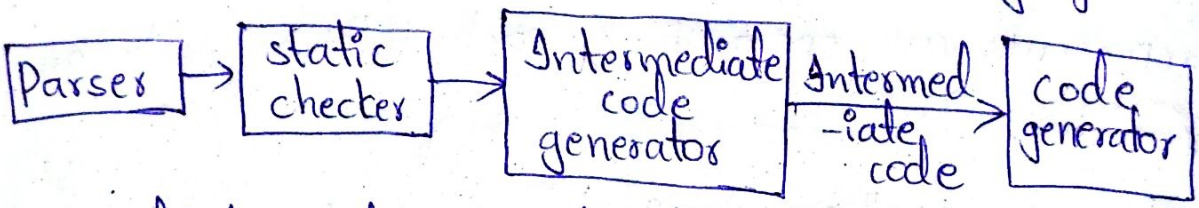
The SDT will be written

A → M B C

M → {M.i = A.i; M.s = f(M.i); }

Intermediate code Generation:

. Intermediate code is used to translate the source code into the machine code. Intermediate code lies blw the high-level language & machine language.



. Intermediate code generator receives input from its predecessor phase & semantic analyzer phase. It takes input in the form of an annotated syntax tree.

. Using the intermediate code, the second phase of the compiler synthesis phase is changed according to the target machine.

. Intermediate code can be represented in two ways:

i) High-level intermediate code:

. It can be represented as source cod. To enhance performance of source code, we can easily apply code modification. But to optimize the target machine, it is less preferred.

ii) Low-level intermediate code:

. Low-level intermediate code is close to the target machine, which makes it suitable for register and memory allocation etc, It is used for machine-dependent optimizations.

① Variants of Syntax trees:

. A variant of a parse tree (Intermediate language representation for a source program) is what is called an syntax tree, a tree in which each leaf represents an operand & each interior node an operator.

. There are two variants of syntax trees:

i) Directed Acyclic Graphs (DAG) for expressions:

. Nodes of syntax tree represents the constructs of the source program; children represents the meaning-ful components.

DAG - It identifies the common subexpressions.

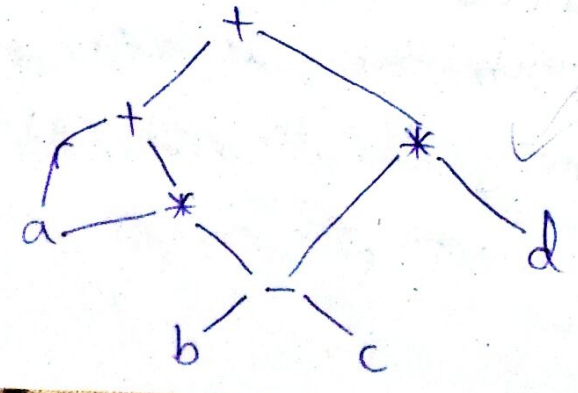
In DAG, leaves represents the operands and the interior nodes represents the operators.

DAG	difference	syntax tree
. More than one parent if 'N' has common subexpressions		. The tree would be replicated as many times as the subexpression appears.

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

Here a & b-c are common subexpressions

DAG:



production	Semantic Rules.
$E \rightarrow E_1 + T$	E.node = newNode('+', E_1.node, T.node)
$E \rightarrow E_1 - T$	E.node = new Node('-', E_1.node, T.node)
$E \rightarrow T$	E.node = T.node
$T \rightarrow (E)$	T.node = E.node
$T \rightarrow id$	T.node = new leaf(id, id.entry)
$T \rightarrow num$	T.node = new leaf(num, num.val)

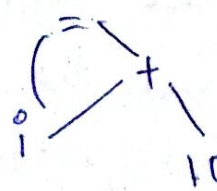
sequence of steps to construct DAG:

i) P_1 = leaf(id, entry-a)
ii) P_2 = leaf(id, entry-a) = P_1
iii) P_3 = leaf(id, entry-b)
iv) P_4 = leaf(id, entry-b) = P_3
v) P_5 = Node('-', P_3, P_4)
vi) P_6 = Node('*', P_1, P_5)
vii) P_7 = Node('+', P_1, P_6)
viii) P_8 = leaf(id, entry-b) = P_3
ix) P_9 = leaf(id, entry-c) = P_4
x) P_{10} = Node('-', P_3, P_4) = P_5
xi) P_{11} = leaf(id, entry-d)
xii) P_{12} = Node('*', P_5, P_{11})
xiii) P_{13} = Node('+', P_7, P_{12})

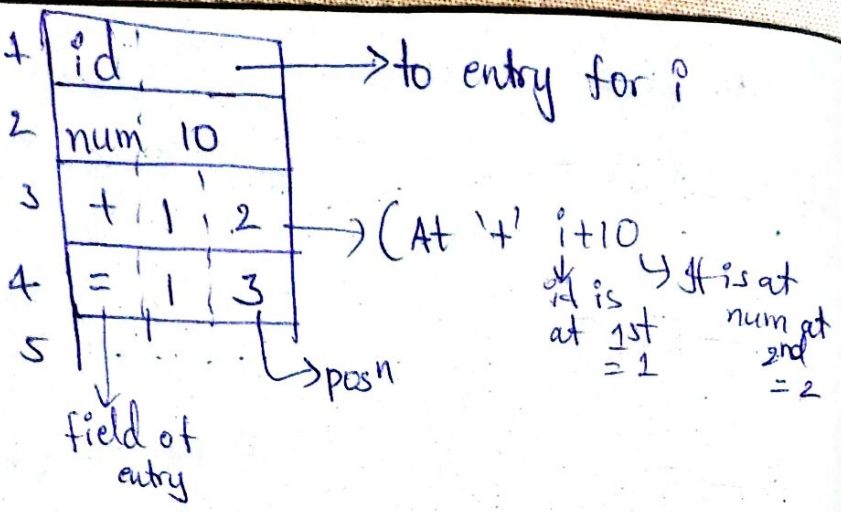
iv) The -value Number method for constructing DAG's:

- Nodes are stored in an array of records.
- Each row of array represents a record (one node).

Eg: DAG for i = i + 10



Array:



(2) Three-Address Code (TAC)

. Three-address code is an intermediate code. It is used by the optimizing compilers.

. In TAC, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.

. Each TAC instruction has at most three operands. It is a combination of assignment & binary operator.

Eg: $x+y*z$

TAC: $t_1 := y * z$
$t_2 := x + t_1$

t_1 & t_2 are compiler generated temporary names.

. In TAC, there is atmost one operaton on the right side of an instruction.

→ There are two ways to represent the TAC:

i) Quadruples:

. It has 4 fields to implement the TAC. The field of Quadruples contains the name of the operator, the first source operand, the second source operand & the result respectively.

Eg: a := -b * c + d

TAC → $t_1 := -b$ —(0)

$t_2 := c + d$ —(1)

$t_3 := t_1 * t_2$ —(2)

$a := t_3$ —(3)

. These statements are represented by quadruples as follows;

	Operator	source 1	source 2	Result
Production(0)	u-minus	b	-	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3	-	a

ii) Triples:

. Triples has 3 fields to implement the TAC. The field of triples contains the name of the operator, the first source operand & the second source operand.

. In triples, the results of respective sub-expressions are denoted by posⁿ of expression. Triple is equivalent to DAG while representing expressions.

Eg: a := -b * c + d

TAC → $t_1 := -b$ $t_2 := c + d$

$t_3 := t_1 * t_2$

$a := t_3$

·These statements are represented by triples as follows:

	operator	source1	source2
(0)	uminus	b	-
(1)	+	c	d
(2)	*	(0) (sub-expression is placed with its posn)	(1)
(3)	:=	(2)	-

(3) Types and Declarations:

· The applications of types can be grouped under checking & translation.

Type checking uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator.

Translation Applications: From the type of a name, a compiler can determine the storage that will be needed for that name at run time.

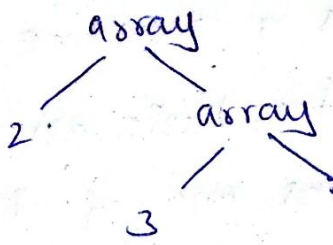
i) Type expressions:

· A type expression is either a basic type or is formed by applying an operator called a type constructor to a type expression.

· The sets of basic types & constructors depend on the language to be checked.

Eg: The array type int [2][3] can be read as "array

of 2 arrays of 3 integers each" & written as a type expression array(2, array(3, integer)).



. This type is represented by the tree.

. The operator array takes 2 parameters, a number & a type.

. A basic type is a type expression. Typical basic types for a language include boolean, char, integer, float & void.

ii) Type equivalence:-

. When type expressions are represented by graphs, two types are structurally equivalent if & only if one of the following conditions is true:

- They are the same basic type.
- They are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

Eg:- int a, b

Here 'a' & 'b' are structurally equivalent. (both 'a' & 'b' are int)

iii) Declarations :-

. Types & declarations using a simplified grammar that declares just one name at a time; declarations with lists of names can also be handled.

The grammar is: $D \rightarrow T\ id\ ;\ D \mid \varepsilon$

$T \rightarrow B\ C \mid record\ '\{'\ D\ '\}'$

$B \rightarrow$ int | float

$C \rightarrow \varepsilon \mid [num]\,C$

- Non-terminal 'D' generates a sequence of declaration
- Non-terminal 'T' generates basic, array, or record type
- Non-terminal 'B' generates one of the basic types int and float.
- Non-terminal C, for "component", generates strings of zero or more integers, each integer surrounded by brackets.

→ An array type consists of a basic type specified by B, followed by array components specified by non-terminal C.

→ A record type (second production for T) is a sequence of declarations for the fields of record, all surrounded by curly braces.

iv) Storage layout for local names:

- From the type of a name, we can determine the amount of storage that will be needed for the name at run time.
- At compile time, we can use these amounts to assign each name a relative address. The type & relative address are saved in the symbol-table entry for name.
- Data of varying length, such as strings, or data whose size cannot be determined until run time, such as dynamic arrays, is handled by reserving a known fixed amount of storage for a pointer to data.

. The width of a type is no. of storage units needed for objects of that type.

. A basic type, such as character, integer or float, requires an integral no. of bytes.

Eg: $T \rightarrow B \quad \{ t = B.type; w = B.width; \}$ - SDT

t, w — synthesized attributes.

. The body of T-production consists of nonterminal B, an action.

If, $B \rightarrow$ int then B.type is set to integer & B.width is 4, width of int.

If, $B \rightarrow$ float then B.type is float & B.width is 8, the width of a float.

(4) Type checking:

. Type checking uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator.

. To do type checking a compiler needs to assign a type expression to each component of source program.

. Type checking has the potential for catching errors in programs.

i) Rules for type checking:

. Type checking can take on two forms: Synthesis & Inference.

Type synthesis:

. Type synthesis builds up the type of an expression

from the types of its subexpressions.
. It requires names to be declared before they are used.
. The type of $E_1 + E_2$ is defined in terms of the types of E_1 & E_2.
. A typical rule for type synthesis has the form:
if f has type $s \to t$ & x has type s,
then expression f(x) has type t.
Here, f & x denotes expressions, $s \to t$: function from s to t
. This rule for func^{n}s with one argument carries over to func^{n}s with several arguments.

Type inference:
. It determines the type of a language construct from the way it is used.
. A typical rule for type inference has the form
if f(x) is an expression,
then for some α and β, f has type $\alpha \to \beta$ &
x has type α.

ii) Type Conversions:
. Conversion from one type to another is said to be implicit if it is done automatically by compiler. ,also called coercions.
. Conversion is said to be explicit if the programmer must write something to cause the conversion. These are also called casts.

. Introducing type conversions into expression evaluation:

$E \rightarrow E_1 + E_2$ { E.type = max(E1.type, E2.type);
a_1 = widen(E1.addr, E1.type, E.type);
a_2 = widen(E2.addr, E2.type, E.type);
E.addr = new Temp();
gen(E.addr '=' a_1 '+' a_2); }

iii) Overloading of functions & Operators:-

. Overloading is resolved when a unique meaning is determined for each occurrence of a name.
. We restrict attention to overloading that can be resolved by looking at the arguments of a func".

iv) Type inference & Polymorphic functions:

. Type inference ensures that names are used consistently.
. The term "polymorphic" refers to any code fragment that can be executed with arguments of different types.
. For each occurrence of a polymorphic func", replace the bound variables in its type by distinct fresh variables & remove the ∀ quantifiers. The resulting type expression is the inferred type of this occurrence.

v) An algorithm for Unification:-

. Unification is the problem of determining whether two expressions 's' and 't' can be made identical by substituting expressions for variables in

s and t.

Algorithm: Unification algorithm:

```
boolean unify(Node m, Node n)
{
    s = find(m); t = find(n);
    if (s = t) return true;
    else if (nodes s & t represent the same
                                basic type)
        return true;
    else if (s is an op-node with children s1
        and s2 and t is an op-node with
        children t1 & t2)
            {
              union(s,t);
              return unify(s1, t1) & unify(s2, t2);
            }
        else if (s or t represents a variable)
            {
              union(s, t);
              return true;
            }
        else return false;
}
```

(5) Control flow:

. In programming languages, boolean expressions are often used to:

Alter the flow of control: Boolean expressions are used as conditional expressions in statements that alter the flow of control.

Compute logical values: A boolean expression can represent true or false as values.

i) Boolean expressions:

. These are composed of boolean operators (&&, ||, ! - AND, OR, NOT) applied to elements that are boolean variables or relational expressions.

. Relational expressions are of the form E_1 rel E_2, where E_1 & E_2 are arithmetic expressions.

ii) Short-circuited code:

. In short-circuit (or jumping) code, the boolean operators &&, ||, & ! translate into jumps.

. The operators themselves do not appear in the code. instead, the value of a boolean expression is represented by a posn in code sequence.

Eg:- if (x < 100 || x > 200 && x != y) x = 0;

might be translated into code shown below:

```
    if x < 100 goto L2
    if False x > 200 goto L1
    if False x != y goto L1
L2: x = 0
L1: .
```

It is true if control reaches label L2, false if it goes to L1.

iii) Flow-of-control statements:

. $S \rightarrow$ if (B) S1

$S \rightarrow$ if (B) S_1 else S_2

$S \rightarrow$ while (B) S_1

. In these productions, non-terminal 'B' represents a boolean expression and non-terminal 'S' represents a statement.

iv) Contro-flow translation of boolean expressions:

. The semantic rules for boolean expressions.

Production	semantic rule
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow !B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow true$	$B.code = gen('goto' B.true)$
$B \rightarrow false$	$B.code = gen('goto' B.false)$

v) Avoiding Redundant Gotos:

. fall through technique is used to avoid the redundant gotos:

$B \rightarrow E_1 \ rel \ E_2$

semantic rules: test $= E_1.addr$ rel.op $E_2.addr$

s = if B.true ≠ fall and B.false ≠ fall then
gen('if' test 'goto' B.true) || gen('goto' B.false)
else if B.true ≠ fall then gen('if' test 'goto' B.true)
else if B.false ≠ fall then gen('iffalse' test 'goto' B.false)
else ''

B.code = E_1.code || E_2.code || s

vi) Boolean values and jumping code:

. To handle the both roles(alter the flow of

control & compute logical values) of boolean expressions is to first build a syntax tree for expressions, using either of following approaches:

→ Use two passes: Construct a complete syntax tree for input, & then walk the tree in depth-first order, computing the translations specified by semantic rules.

→ Use one pass for statements, but two passes for expressions. With this approach, we would translate E in while(E) S_1 before S_1 examined. The translation of E, however, would be done by building its syntax tree & then walking the tree.

⑥ Switch-statements:-

. Type of selection control statement that allows the value of a variable to change the control flow of program execution.

Eg:- switch(E){

case V_1 : S_1

case V_2 : S_2

.

.

.

case V_{n-1} : S_{n-1}

default : S_n

}

Translation of switch-statements:

i) Evaluate the expression E

ii) Find the value V_j in the list of cases.

Implemented as,

- Sequence of conditional jumps
  - ↳ create table of pairs (value, label)
  - Hash table for the values.

iii) Execute the statement S_j.

Syntax-directed translation of switch-statement

code to evaluate E into t

if t != V_1 goto L_1

code for S_1

goto next

L_1 : if t != V_2 goto L_2

code for S_2

goto next

L_2 : . . . .

L_{n-1} : code for S_n

next :

(1) Intermediate code for Procedures:-

- Procedure means a function.

let the grammar be:

D → define T id (F) { S } (T → return type, id → func^n)

F → ε | T id , F (id → variable)

S → return E ;

E → id (A) ;

A → ε | E , A

where D stands for func^n procedure definition

T – return type/data type

id – variable (identifier)

label)

n-statement

F - Formal parameters (in called fun^n)

S - statements

E - Expressions, A - Actual parameters. (in main func^n called)

Intermediate code for procedure:

```
float add( )
        (int a)
{       (int a, float b)
return add(a)
          ( )
}
```

F → ϵ | T id, F

formal paraunts

actual parameters // {S}

```
main( )
{
 add( ); // It is called as it defined before
}
add( )
{
 }
```