Syntax Directed Definition (SDD):-

→ Semantic Analysis phase & ICG phase uses SDD's.

→ SDD is Context Free Grammar with Semantic Rules and attributes

$\boxed{\text{SDD} = \text{CFG} + \text{Semantic Rules}}$

→ SDD is also called as "attribute grammar"

→ Attributes are associated with grammar symbols and semantic Rules are associated with productions.

→ If 'X' is a symbol and 'a' is one of attribute then X.a denotes value at node 'x'.

→ Attributes may be numbers, string, data types & references etc

Eg

Production	Semantic Rules
$E \to E+T$	$E.val = E.val + T.val$ (Here E → is grammar symbol, val → is attribute of E)
$E \to T$	$E.val = T.val$
$D \to TL$	$L.inh = T.type$

↳ Semantic Rules have two Notations. (i) SDD (ii) SDT

Types of attribute:-

(i) Synthesized Attribute:- If a node takes value from its children node then it is called as synthesized Attribute.

Eg:- $A \to BCD$ $\quad A.S = B.S$, $A.S = C.S$, $A.S = D.S$

A → is synthesized attribute.
parent node takes value from child nodes
A → be a parent node
B, C, D → are children node.

(ii) Inherited Attributes:- If a node takes value from its sibling or Parent then it is called as inherited attributes.

$D \to TL$ eg

production | Semantic Rules

Eg:- $A \to BCD$ → C - is inherited attribute

$C.i = A.i$ → Here C is taking value from its parent A.

$C.i = B.i$ → C is taking value from its sibling B.

$C.i = D.i$ → C is " " " " D

Eg:-

Production	Semantic Rules
$L \to En$	$E.val = E.val$
$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to T_1 * F$	$T.val = T_1.val * F.val$
$T \to F$	$T.val = F.val$
$F \to (E)$	$F.val = E.val$
$F \to digit$	$F.val = digit.lexval$

Fig: SDD for a simple desk calculator

Evaluating an SDD at Nodes of parse Tree:-

Annotated parse tree:-

→ A parse tree which contain values at each node is known as Annotated parse tree.

→ A parse tree is constructed inorder to evaluate the attribute value at each node of parse tree.

→ If an attributed is synthesized.

- 1st evaluate the val attribute at all the children node.
- evaluate the value attribute at parent node.
- synthesized attributes, attributes are evaluated in bottom-up manner.

Production	Semantic Rules
$A \to B$	$A.S = B.i$
	$B.i = A.S + 1$

These rules are circular, it is impossible to evaluate A.S without first evaluating B.i at some node.

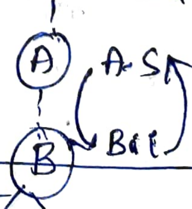


Fig: The Circular dependency of A.S and B.i on one another

Eg1: Construct an SDD for simple desk calculator Grammar and construct parse tree for 3*5+4n.

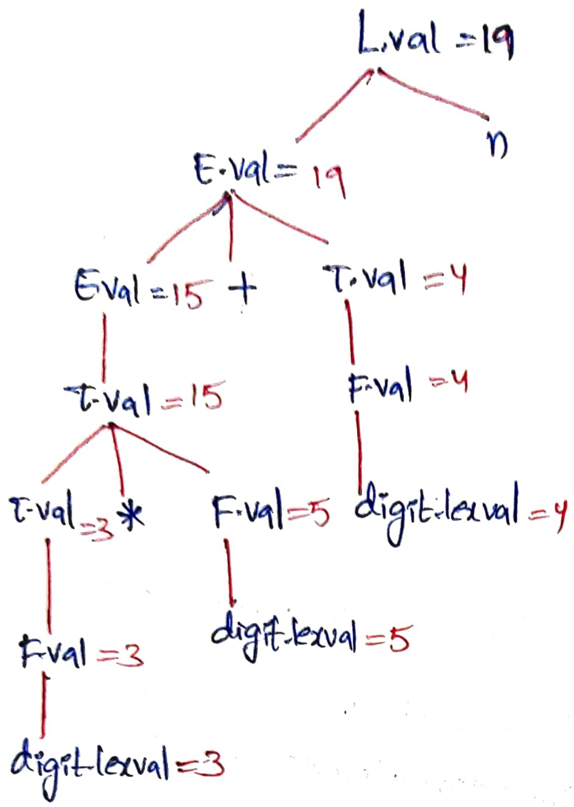


Fig: Annotated parsetree for 3*5+4n

Grammar

production	semantic Rule
L→En	L.val = E.val
E→E+T	E.val = E.val + T.val
E→T	E.val = T.val
T→T₁*F	T.val = T.val * F.val
T→F	T.val = F.val
F→(E)	F.val = E.val
F→digit	F.val = digit.lexval

Fig: SDD for simple desk calculator

Eg2: construct Annoted parse tree for 2+3*4.

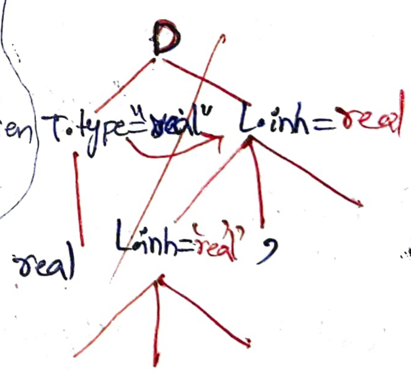
Eg: ↳ Annotated parse tree contains values at each node.

↳ To construct Annotated parse tree, we have to perform top-down left to Right traversing, if there is reduction execute corresponding action.

Eg2:

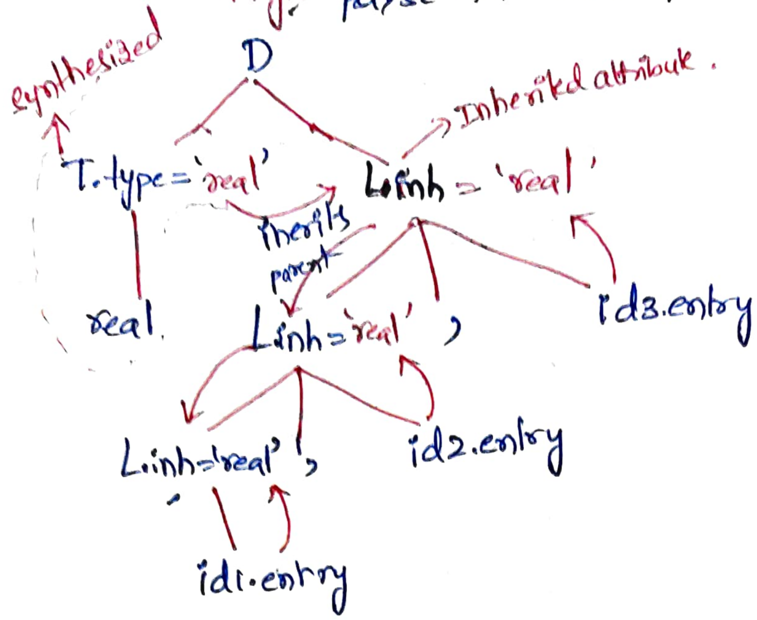
production	semantic Rules
D→TL	L.inh = T.type → list inherits type T.
T→int	T.type = "integer"
T→real	T.type = "real"
L→L₁,id	L₁.inh = L.inh → here id needs to be given T.type.
	addtype(id.entry, L.in)
L→id	addtype(id.entry, L.in)

Parse tree:



↳ In dependency graph we have ~~an edges~~ directed edges

Fig:- ~~parse~~-Annotated parse tree for id_1, id_2, id_3



② Evaluating order SDD's :-

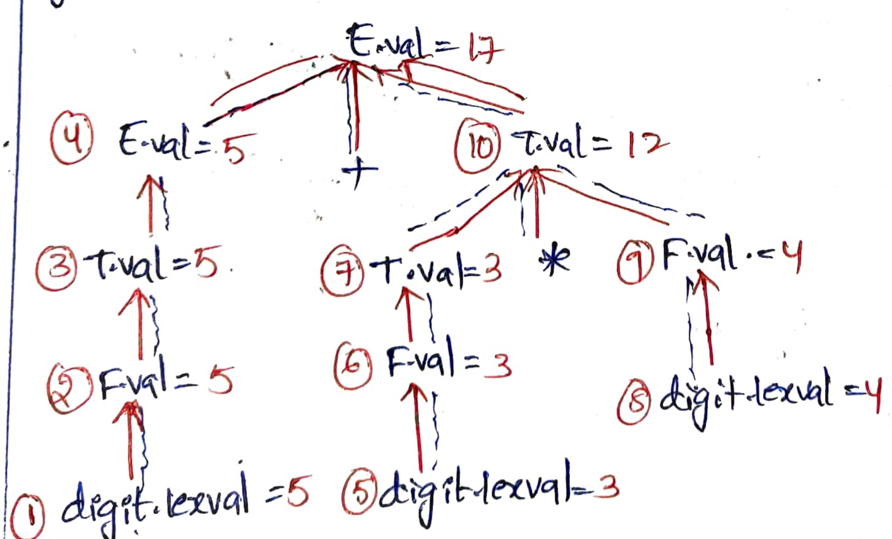
→ Dependency graph is used to determine an evaluation order for the attribute given

Dependency Graph :- → A directed graph that represents the interdependency b/w synthesized and inherited attribute at node in the parse tree

→ Dependency Graph represents the flow of information among the attributes in parse tree.

→ used to determine the evaluation order for attribute in a parse tree. (which semantic action should execute first)

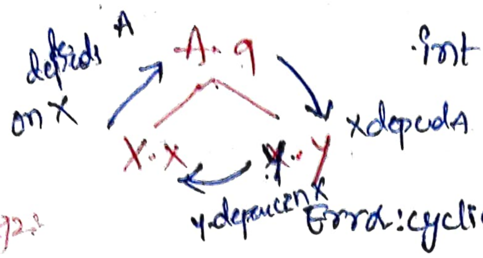
→ An Annotated parse tree shows the value of attributes, a dependency graph determines how those values can be computed.



production	semantic Rules
E → E + T	E.val = E.val + T.val
E → T	E.val = T.val
T → T * F	T.val = T.val * F.val
T → F	T.val = F.val
F → digit	F.val = digit.lexval

Fig:- Dependency graph for Annotated parse tree for 5 + 3 * 4

→ Edges in dependency graph show the interdependency b/w synthesized & inherited attributes at nodes in its parse tree.



int a
D→TL

D→ int a

→ Dependency graph cannot be cyclic

Error: cyclic dependence

→ I DG there is a edge ~~towards~~ the dependent node to originating node.

Eg:-

Production	Semantic Rules
$T \rightarrow FT'$	T'.inh = F.val T.val = T'.syn
$T' \rightarrow *FT'$	T'.inh = T'.inh * F.val T'.syn = T'.syn
$T' \rightarrow \varepsilon$	T'.syn = T'.inh
F → digit	F.val = digit.lexval

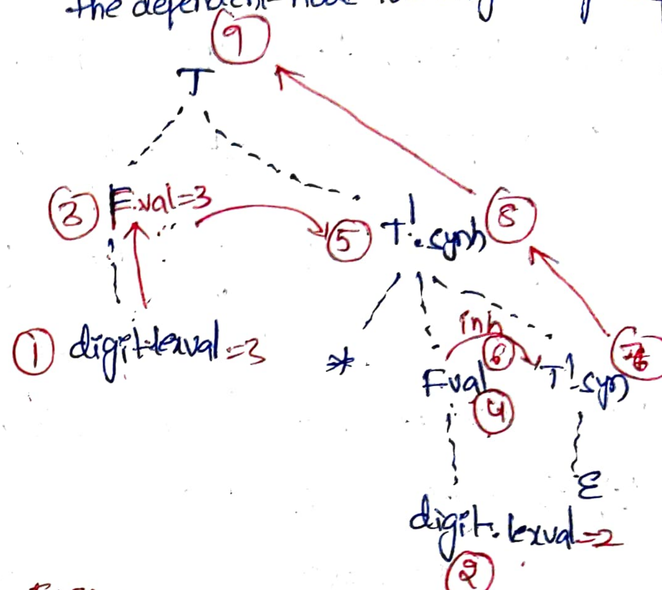
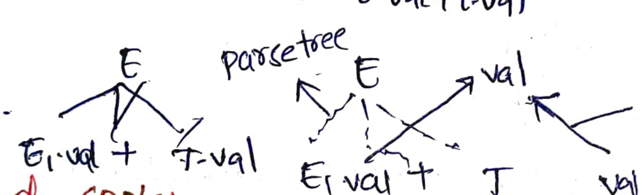


Fig:- Dependency graph for 3*5

Production	Semantic Rule
E→E₁+T	E.val = E₁.val + T.val



solid line → Dependency
--- line → parse tree

Fig:- E.val is synthesized from E₁.val and T.val.

Types of SDD's:-

S-attributed Definition:-

* A SDD that use only synthesized attributes is called as S-attributed SDD.

Eg A → BCD

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

Eg:-

Production	Semantic Rules
L → En	L.val = E.val
E → E+T	E.val = E.val + T.val
E → T	E.val = T.val
T → T*F	T.val = T.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → digit	F.val = digit.lexval

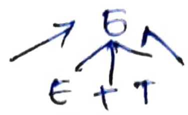
Eg:- SDD for S-attributed Definition

* semantic actions are placed at the end of the production.

	production	semantic action
Eg:-	E→E+T	{ E.val = E.val + T.val }

→ It is also called as postfix "SDD"

→ Attributes are evaluated with Bottom-up parsing



Bottomup manner

L-Attributed SDD:-

* A SDD that use both synthesis attributes & Inherited attributes is called as L-Attributed SDD.

* In L-attributed SDD, Inherited attributed is restricted to inherit from parent & left sibling only.

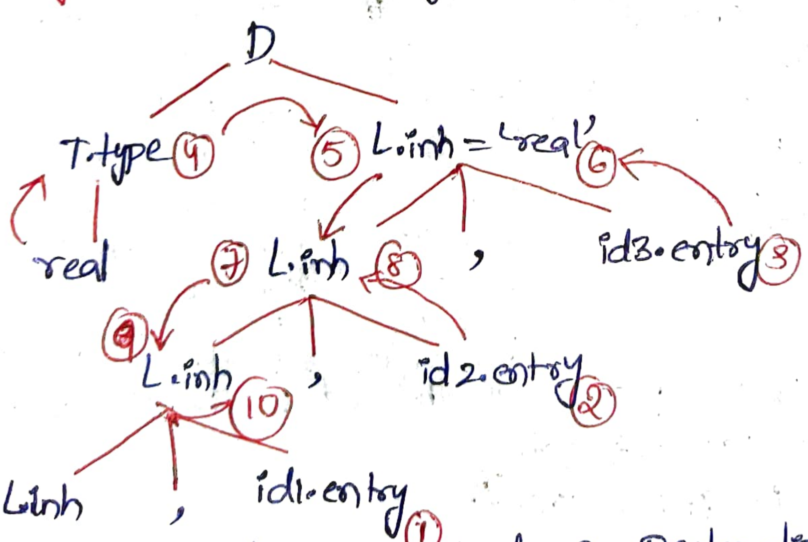
Eg:- $A \to XYZ \{ y.S = A.S,\ y.S = X.S,\ y.S = Z.X \}$ (x) production semantic each

* semantic actions are placed anywhere on R.H.S. eg= $E \to E+T \{ \}$
$E \to \{ \} E+T$

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

Production	Semantic Rules
$D \to TL$	$L.inh = T.type$
$T \to int$	$T.type = integer$
$T \to float$	$T.type = float$
$L \to L_1, id$	$L_1.inh = L.inh$. addtype(id.entry, L.inh)
$L \to id$	L.inh = addtype(id.entry, L.inh)

Eg:- SDD for simple type Declaration for L-Attributed SDD



Eg:- Dependency graph for a Declaration float id_1, id_2, id_3

(3) Application of Syntax Directed Translation:-

Syntax Directed Translation (SDT):-

- ↳ SDT is a CFG together with semantic actions.
- ↳ Semantic actions are enclouse in '{', '}' Eg: A→ABC → ABC{ }, → A{ }BC, → { }ABC
- ↳ Semantic actions are placed at any where on RHS of productions.
- ↳ semantic actions specifies in which order the Expression is Executed.

Eg:- production — semantic Action

A→B+C — {printf ('+');}

Eg 2:-

production	semantic Action	
E→E+T	{print ('+');}	(1)
E→T	{ }	(2)
E→T*F	{printf ('*');}	(3)
T→F	{ }	(4)
F→num	{printf(num.val);}	(5)

Fig:- SDT for evalution of Expression.

Eg 2:-

production	Semantic Action
E→E+T /	{E.val=E.val+T.val} (1)
T	{E.val=T.val}
T→T*F /	{T.val=T.val*F.val}
F	{T.val=F.val}
T→num	{E.val=num.lvalue}

2+3*4

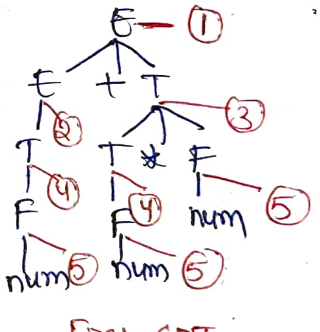


Fig:- SDT

Applications of SDT:-

Construction of syntax Tree:-

- → syntax tree is an intermediate representation
- → The nodes in syntax tree is implemented by objects with suitable no. of fields
- → Each field will have an op-field that is the lable of the node.

we can construct the syntaxtree by using the following functions.

(1) Mknode (op, left, Right)

(2) mkleaf (id, entry to symbol table)

(3) mkleaf (num, value)

→ If the node is leaf, an additional field hold the lexical value for the leaf

leaf (op, val) – create leaf object

→ If Node is an operator, create an object with first fied. op and k additional for the k children $c_1 \ldots c_2$

Eg:- construct the syntax tree for the following grammar for the expression $x*y-5+3$.

Step1:- construct SDD for the given grammar

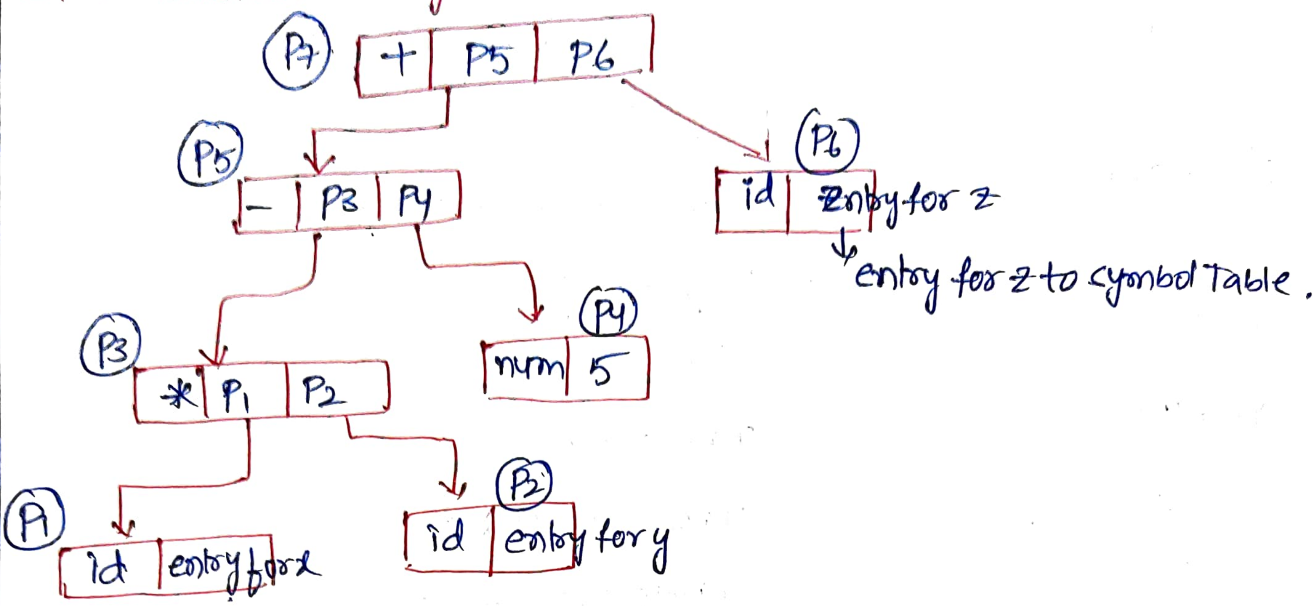
production	Semantic Rule
$E \to E_1 + T$	E.node = mknode ('+', E_1.node, T.node) (or) E.node = newleaf ('+', E_1.node, T.node)
$E \to E_1 - T$	E.node = mknode ('-', E_1.node, T.node)
$E \to T$	E.node = T.node
$T \to (E)$	T.node = E.node
$T \to id$	T.node = mkleaf (id, id.entry) (or) T.node = newleaf (id, id.entry)
$T \to num$	T.node = mkleaf (num, num.val)

Step2:-

symbol	operation
x	P_1 = mkleaf (id, entry_a)
y	P_2 = mkleaf (id, entry_y)
*	P_3 = mknode ('*', P_1, P_2)
5	P_4 = mkleaf (num, 5)
-	P_5 = mknode ('-', P_3, P_4)

z	P_6 = mkleaf (id, entry-z)
+	P_7 = mknode ('+', P_5, P_6)

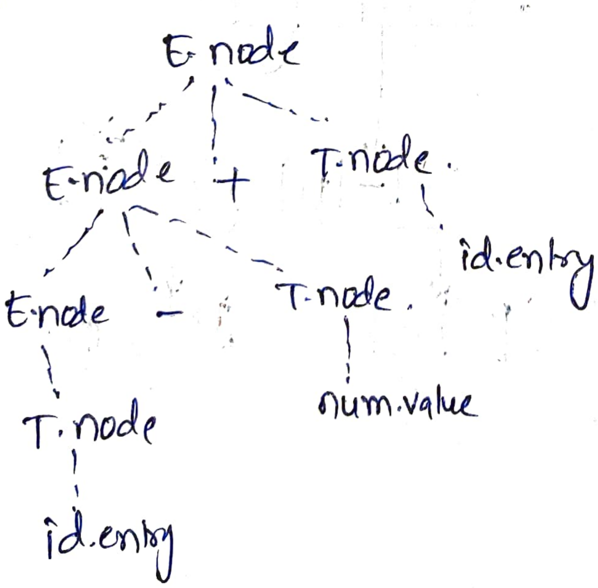
Step3:- construct the syntax tree.

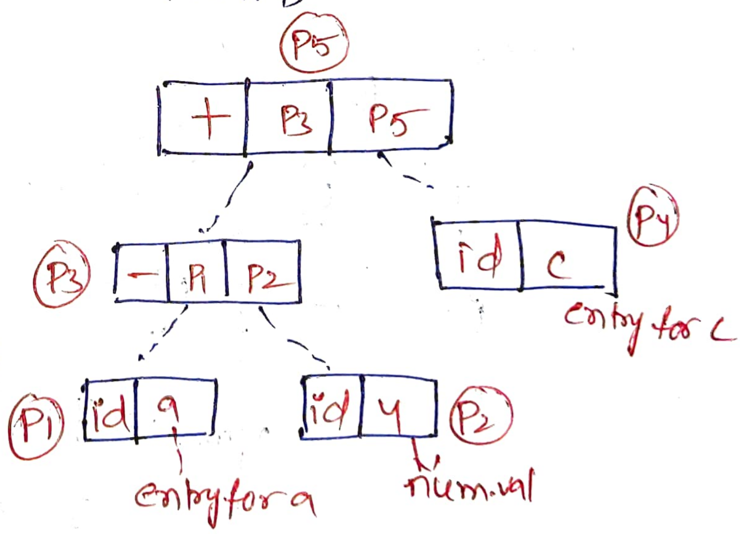


Eg2:- a–4+c

symbol	operation
a	P_1=mkleaf (id, entry-a)
4	P_2=mkleaf (num, 4)
–	P_3 = mknode ('–', P_1, P_2)
c	P_4 = mkleaf (id, entry-c)
+	P_5 = mknode ('+', P_3, P_4)

Constructing syntax tree:-





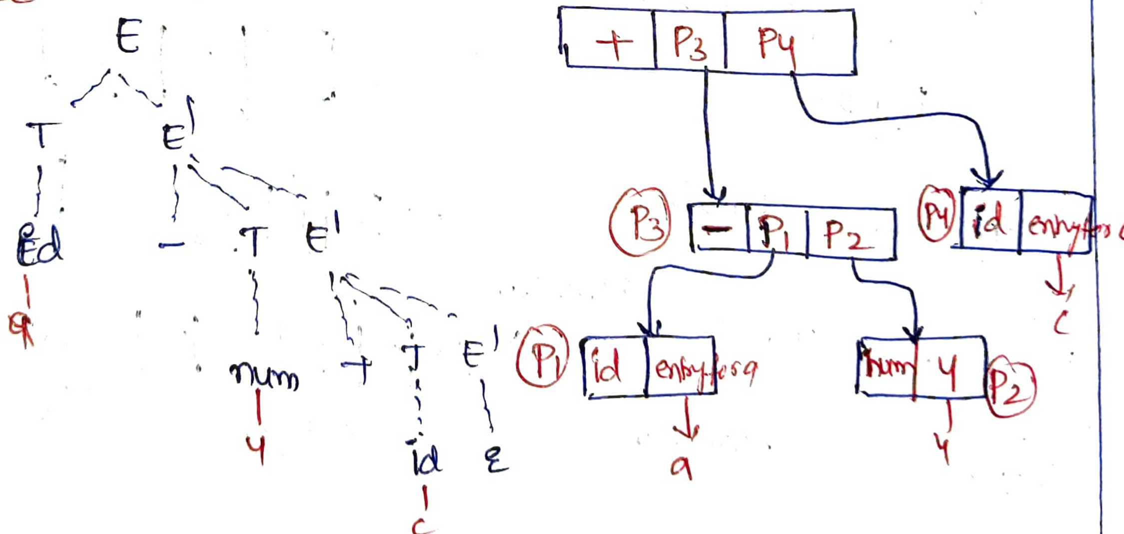
Eg3:- Production | Semantic Rules

Production	Semantic Rules
$E \to TE'$	$E.node = E'.syn$ $E'.inh = T.node$
$E' \to +TE_1'$	$E_1'.inh = newNode('+', E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \to -TE_1'$	$E_1'.inh = newnode('-', E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \to \varepsilon$	$E'.syn = E'.inh$
$T \to (E)$	$T.node = E.node$
$T \to id$	$T.node = newleaf(id, id.entry)$
$T \to num$	$T.node = newleaf(num, num.val)$

Eg:- $a - 4 + c$

symbol	operation
a	$P_1 = mkleaf(id, entry\text{-}a)$
4	$P_2 = mkleaf(num, 4)$
-	$P_3 = mknode('-', P_1, P_2)$
c	$P_4 = mkleaf(id, entry\text{-}a)$
+	$P_5 = mknode('+', P_3, P_4)$

Syntax tree:



(5) Syntax-Directed Translation schemas:-

→ A SDT is a context free Grammar combined with semantic actions.

→ Semantic actions are also called as program fragments.

→ semantic actions are embedded with production body.

→ Any SDT can be implemented by constructing parse tree and then performing the actions in a left-to-Right depth-first order.

SDT's are used to implement two important classes of SDD's

1) The underlying grammar is LR-parsable, & the SDD is S-attributed

2) The underlying grammar is LL-parsable, and the SDD is L-attributed.

Postfix Translation schemes:-

→ It is used to convert infix expression to postfix expression

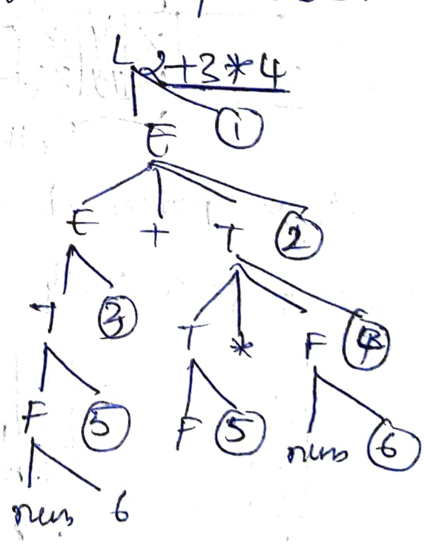
→ Infix expression - operator appears b/w operands

postfix expression - operator appears after the operands.

Eg:-

Production	Semantic actions
L → En	{ print(E.val); } (1)
E → E+T	{ print('+'); } (2)
E → T	{ } (3)
T → T*F	{ print('*'); } (4)
T → F	{ } (5)
F → num	{ print num.val; } (6)

fig:- SDT for desk calculator



2 3 4 * +

2

IInd method To convert the ~~Postfi~~ infix Expression to postfix Expression

Eg:- $E \to E+T \mid T$ for $= 1+2+3$

$T \to num$

= (here we are having only operation, so that we need check the left recursion in the grammar

→ if Grammar contains left recursion we we need to ~~convert the th~~ eliminate left recursion from the grammar

production	SDT
$E \to E+T$	{printf('+');}
$E \to T$	{ }
$T \to num$	{print num-value}

SDT for given grammar

so, the given Grammar contains left recursion

$\underbrace{E}_{A} \to \cancel{\underbrace{E}_{A}+T} \mid T$ $\quad \underbrace{E}_{A} \to \underbrace{E}_{A}\underbrace{+T\{printf('+');\}}_{\alpha} \mid \underbrace{T}_{\beta}$

$T \to num$

$A \to A\alpha \mid \beta$

$\boxed{\begin{aligned} A &\to \beta A' \\ A' &\to \alpha A' \mid \epsilon \end{aligned}}$

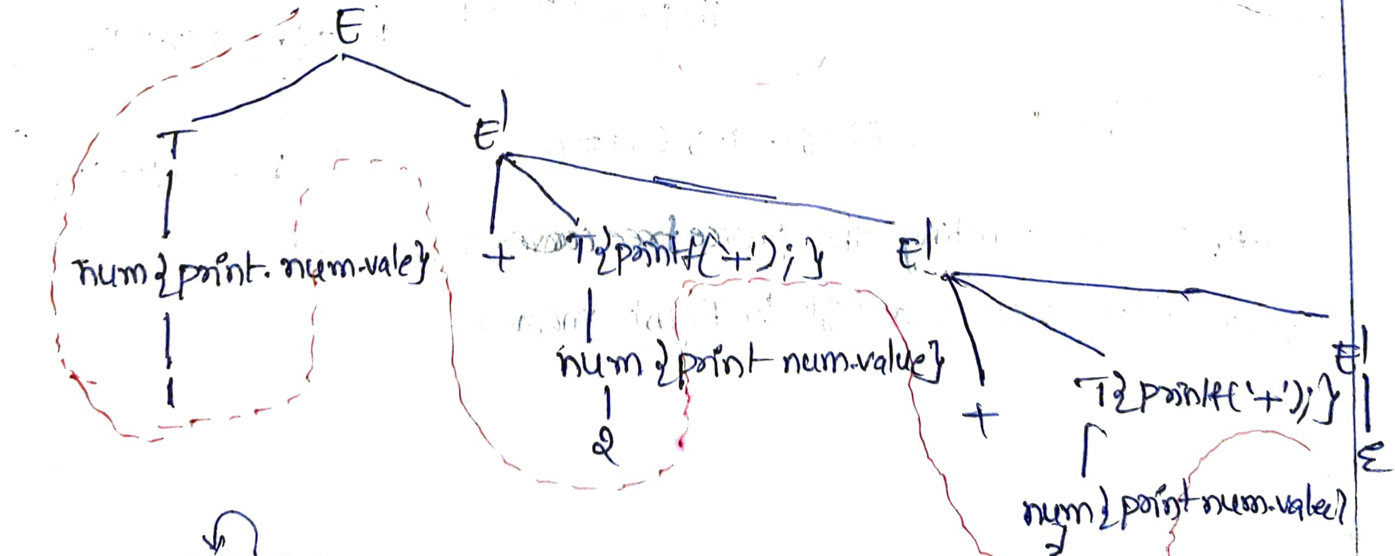
The grammar after eliminating the left recursion:

$E \to TE'$

$E' \to +T\{printf('+');\}E' \mid$

$E' \to \epsilon$

$T \to num\{print\ num.value\}$



1 2 + 3 +

$(1+2)+3 \Rightarrow 6$

Eg:- Give Translation scheme that convert infix expr to postfix expression for the following grammar and also generate Annotated parse for inputstring 2+6+1

Grammar → $E \to E+T \{print('+')\} \mid T$

$T \to 0 \mid 1 \mid 2 \mid \ldots 9$ (it contains left recursion)

$A \to A\alpha \mid \beta$

$\boxed{\begin{array}{l} A \to \beta A' \\ A' \to \alpha A' \mid \varepsilon \end{array}}$

$E \to TE'$
$E' \to +T \{print('+');\} E'$
$E' \to \varepsilon$
$\Bigg\}$ The Grammar after eliminating left recursion

$T \to 0 \{print('0');\}$

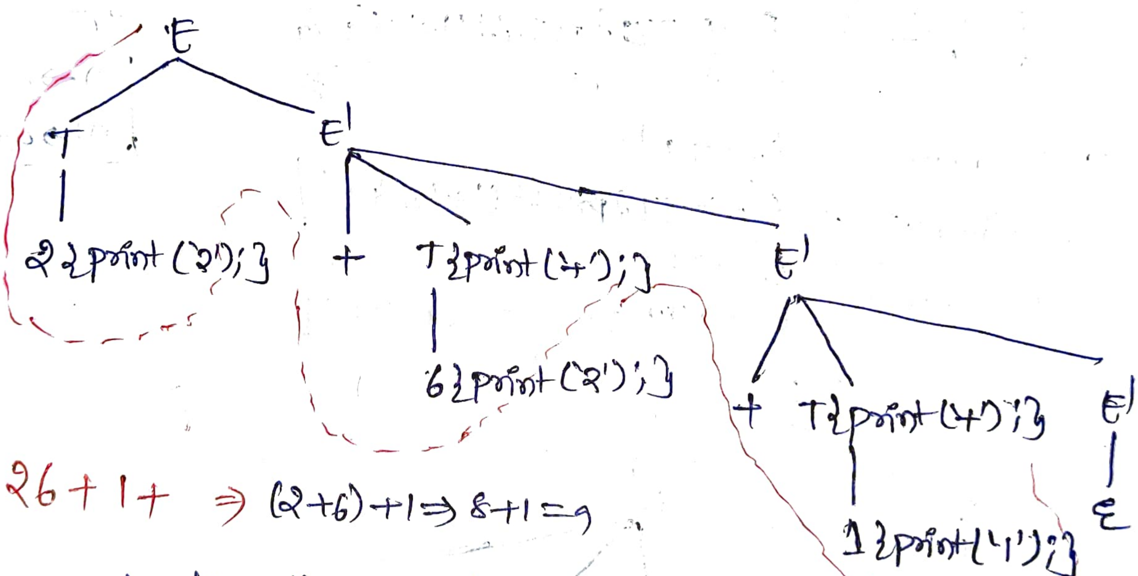
$T \to 1 \{print('1');\}$

⋮

$T \to 9 \{print('9');\}$

Fig: SDT to convert infix to postfix

Annotated Parsetree:- The given i/p string 2+6+1 ⇒ 26+1+ (infix ⇒ postfix)



26+1+ ⇒ (2+6)+1 ⇒ 8+1 = 9

→ After constructing the parse tree, traverse the parse from top-down and left to Right manner

$L \rightarrow En$ {print (E.val);}

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

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

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

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

$F \rightarrow (E)$ {F.val = E.val}

$F \rightarrow digit$ {F.val = digit.lexval}

Fig:- postfix SDT implementing the desc calculator

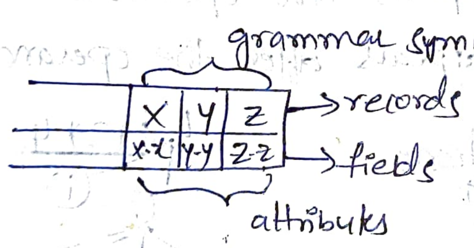
parser - stack implementation of postfix SDT's :=

→ postfix SDT's can be implemented during LR parsing by executing the actions when the reductions occur.

→ ~~The grammar symbols of each grammar s~~

→ The attributes of each grammar symbols can be placed in stack during the parsing.

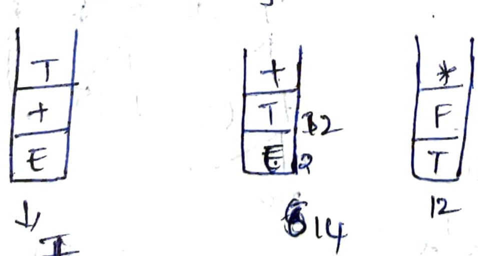
→ The parser stack contain records with fields for a grammar symbol.



if production $A \rightarrow XYZ$

fig:- parser stack with a field for synthesized attributes

eg $E \rightarrow E + T$ {print('+');} or $E \rightarrow E + T$ · 2+3*4

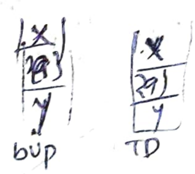


production	Actions.
L → En	{ print (stack[top-1], val); top=top-1
E → E+T	{ stack [top-2]. val= stack [top-2]. val + stack [top].val; top=top-2;}
E → T	
T → T₁*F	{stack [top-2]. val = stack[top-2]. val x stack[top].val; top= top-2;}
T → F	
F → (E)	{ stack[top-2]. val = stack [top-1] .val; top=top-2;}
F → digit	

Eg:- Implementing the desd calculator on bottomup-parsing stack.

SDT's with Actions Inside productions:-

→ ~~an~~ actions maybe placed at any position ~~Right~~ within the body of production.



eg:- B → x{a} y.

The action "a" is executes after recognizing the "x"

- If the parse is bottom-up then we perform action "a" when "x" is appears on the top of the stack.
- If the parse is top-down, we perform a just before expanding the "y"

Any SDT can be implemented as follows.

1) Ignoring the actions, parse the i/p & produce a parse tree as a result.

2) examine each node and add additional node for corresponding action

3) perform the preorder traversal of the tree, and as soon as a node is labeled by action is visited, perform that action.

Eg :- parse tree for expression 3*5+4 with actions inserted ~~we get~~ if we visit the nodes in preorder, we get the prefix form of expression +*354

production	semantic action
L → En	{ }
E → E+T	{print('+');} ~~E+T~~
T → T*F	{print('*');} ~~T*F~~
E → T	{ }
T → F	{ }
F → digit	{print(4);}

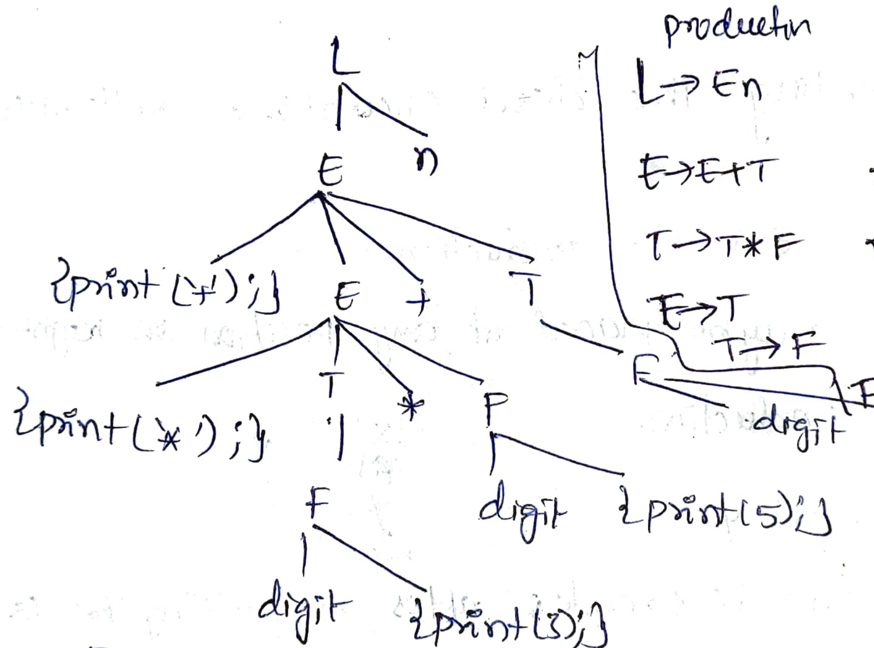


Fig: Parse tree with actions embedded.

Intermediate code generation:-

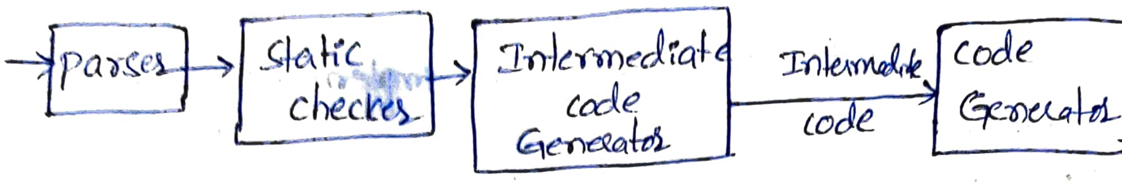


Fig:- logical structure of a compiler Front end.

→ Intermediate code is used to translate the sourcecode into the machine-code.

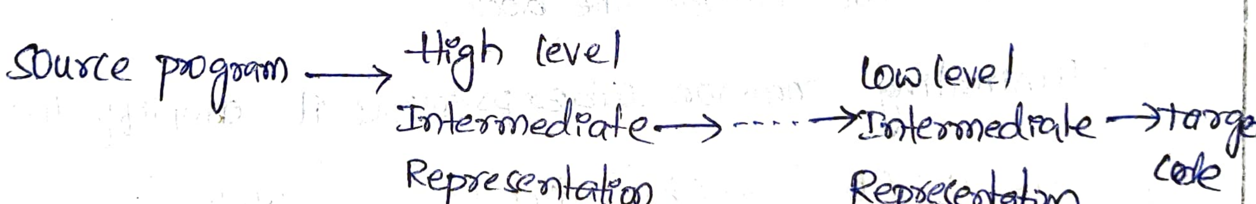
→ In the above figure parsing, static checking and Intermediate-code generation are done sequentially.

→ Static type checking includes type checking, which ensures that operators are applied to compatible operand.

→ ICG receives from its predecessor phase & semantic analyzer phase

→ It takes i/p in the form of an annotated syntax tree.

→ In process of translating a source program into target code compiler may construct a sequence of intermediate representation



→ Syntax trees are high level representation

→ A lower representation is suitable for machine-dependent tasks like register allocation and instruction selection.

7. Variants of syntax trees:-

Directed Acyclic Graph (DAG) for Expressions:-

→ DAG is a datastructure used for implementing transformations on basic blocks.

→ ~~DAG~~ Nodes in ~~↓~~
→ DAG represent the structure of a basic block.

- In DAG internal nodes represents operators and leaf nodes represent identifiers, constants.

result

$\oplus$

a b

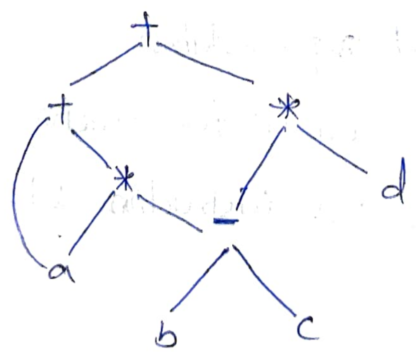
- Internal nodes represent the result of expression

→ The only difference b/w syntax tree and DAG is, In DAG a node has more than one parent.

Applicatione of DAG:-

* Determining the common subexpression.
* Determining which ~~the~~ names are used inside the block and computed outside, the blocks.
- Determining which statement of the block could have their computed value outside the block.
- By Eliminating common subexpressions it simplify the code.

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



Syntax tree

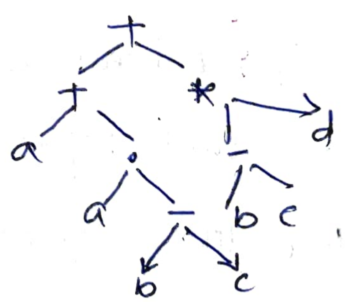


Fig:- ~~Dag~~ DAG for expression $a+a*(b-c)+(b-c)*d$

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

Fig:- SDD ~~for~~ to produce syntax tree & DAG's

1) P_1 = leaf (id, entry-a)

P_2 = leaf (id, entry-a) = P_1

P_3 = leaf (id, entry-b)

P_4 = leaf (id, entry-c)

P_5 = Node ('-', P_3, P_4)

P_6 = Node ('*', P_1, P_5)

P_7 = Node ('+', P_1, P_6)

P_8 = leaf (id, entry-b) = P_3

P_9 = leaf (id, entry-c) = P_4

P_{10} = Node ('-', P_3, P_4) = P_5

P_{11} = leaf (id, entry-d)

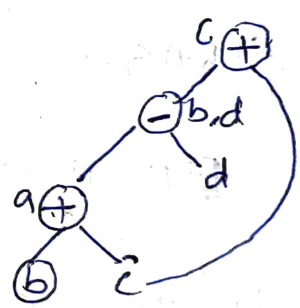
P_{12} = Node ('*', P_5, P_{11})

P_{13} = Node ('+', P_7, P_{12})

Eg 2:- construct DAG for

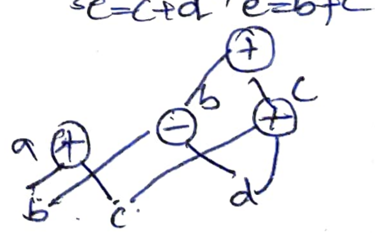
1 a=b+c 3 c=b+c

2 b=a-d 4 d=a-b



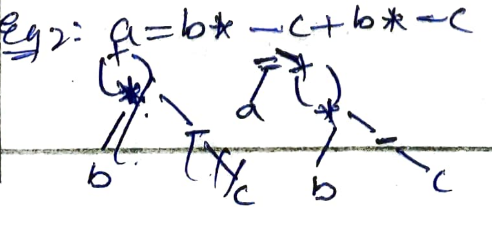
Eg 3:- 1 a=b+c 2 b=b-d

3 c=c+d 4 e=b+c

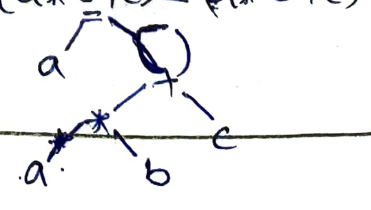


The value number Fig: steps for constructing the DAG.

Eg 2: a=b*-c+b*-c



eg 3: a=(a*b+c)-(a*b+c)



II The Value Numbers method for constructing DAG's:-

→ Nodes of syntax tree or DAG are stored in array of records.

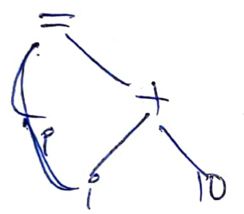
→ Each row of array represent one record.(node)

→ In each record first field is operation code, indicating the label of the node.

→ leaves has ~~the leaves~~ one additional field which holds the lexical value.

→ Interior nodes have two additional field indicating left and right children.

$i = i + 10$



(a) DAG for $i=i+10$

leaves

record

1	id		→ to entry for i
2	num	10	
3	+	1	2
4	=	1	3
5			

(b) Array

Fig:- Nodes of a DAG for $i = i + 10$ allocated in an array.

→ The array index is used for reference a node rather than a pointer

→ Initially the array is empty.

→ First it searches for <id, lexval>, if it is not there, we will make new record and so on.

→ if already, record is present, it is just used for further records

→ In the array, we refer to node by giving integer index of the record, for that node within the array this integer is called 'Value Number'. <op, value-num of left, value num of right>
↓
is also called as signature of node

Drawbacks

Search options: It takes time for every New/old record to search.

→ To overcome this we are using hash-functions, in which the nodes are put into "buckets" (hash table)

→ These buckets have only few nodes.

→ hashtable is a data structure that support the dictionaries.

→ Dictionaries are used to insert & delete elements of a set.

→ Dictionaries are used to determine whether a given element is currently in the set. ~~this is don~~

→ It search the elements in less time and independent of the size of set.

TO construct a has table for node of a DAG, ~~we~~ hashfunction "h" is used that computes the index of bucket.

→ ~~h(op,l,r)~~ The bucket index h<op, l,r>

→ bucket can be implemented as linked list.

→ ~~An~~ Array indexed by hashvalue, hold the bucket headers, each of which point to first cell of list

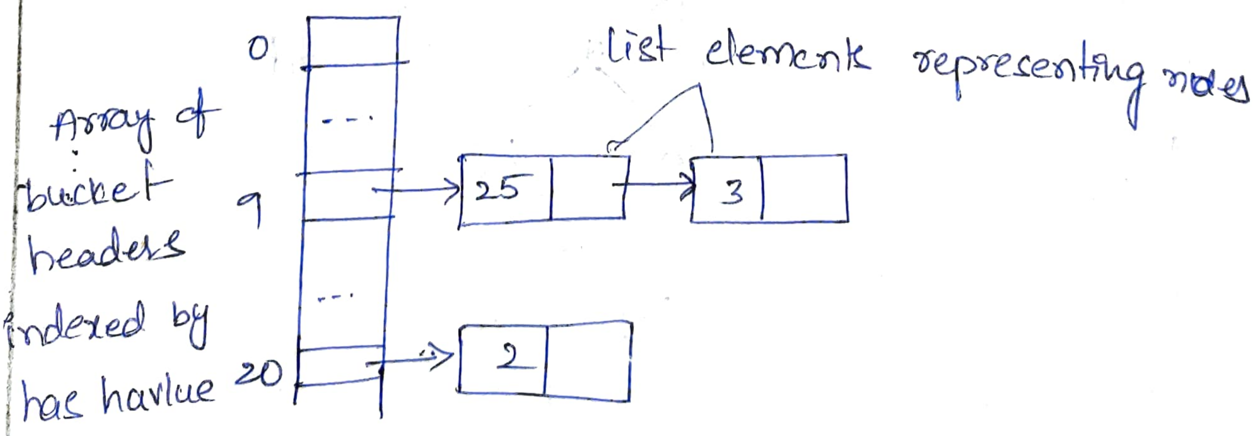


Fig: Data structure for searching buckets

(8) Three Address code:-

Intermediate code is three types
1. syntax tree Representation
2. postfix Notation
3. Three Address code.

In three-address code
1. Each Instruction should contain atmost ~~2~~ 3 addresses
2. Each Instruction should contain 1 operator on R.H.S.

eg:- source language expression $x+y*z$ is converted into sequence of 3-address instructions.

~~$t_1 = $~~ $t_1 = y*z$
$t_2 = x+t_1$

$t_1, t_2 \rightarrow$ compiler generated temporary variables.

→ 3-address code is linearized representation of syntax tree & DAG

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

$t_1 = b-c$
$t_2 = a*t_1$
$t_3 = a+t_2$
$t_4 = t_1*d$
$t_5 = t_3+t_4$

Fig:(a) Three address code

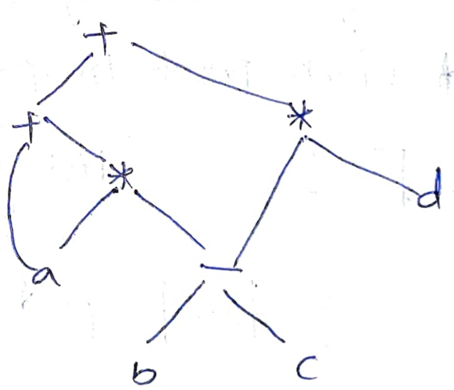


Fig:(b):- DAG

→ Three-address code is represented in 3-ways
(a) Quadruple (2) Triple (3) Indirect Triples

~~Types of 3-address code:~~

Addresses and Instructions:-

→ 3-address code can be implemented by using records with fields for the addresses

→ records are called quadruples and triples.

The address can be one of the following:

- A name → (source program names are addresses in 3-address code and names are replaced by pointer to its symbol table entry)
- A constant –
- compiler generated temporary variables.

List of the common three-address instruction forms:-

(i) Assignment instruction → $x = y$ op z, where x, y, z - addresses

→ $x =$ op y where op - is unary operation $(-, \sim)$

(ii) copy instruction → $x = y$

(iii) unconditional jump → goto L

(iv) conditional jump → if x goto L

→ if false x goto L.

→ if x relop y goto L.

(v) procedure call → y = call p, n

return y

p → is the address of starting of line of procedure p

n → argument address.

$y \xrightarrow{\text{takes}}$ return value

(vi) Indexed copy instruction → $x = y[i]$, $x[i] = y$ } x, y, i are the variables.

(vii) Address and pointer assignment $x = \&y$

$x = *y$

$*x = y$

Three-address code

Quadruples :-

3-address code is implemented as objects (or) records with fields for the operator and operand.

→ Quadruple has 4-fields (i) op (ii) arg1 (iii) arg2 (iv) result.

Eg :- * instruction like $x=y$ & $x=-y$ do not use arg2

* operator like param use neither arg2 nor result
* conditional & unconditional jumps put the target lable in result.

Eg :- $a = b * -c + b * -c$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$ $\Big\}$ $t_3 = b * t_1$, $t_4 = t_2 + t_3$
$a = t_5$

(a) Three address code

	op	arg1	arg2	result
(0)	−	c		t_1
(1)	*	b	t_1	t_2
(2)	−	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

(b) quadruples

→ The disadvantage of Quadruples is too many temporary variables are needed, it require more amount of memory.

→ Inorder to overcome we are using Triples.

triples :- Triples has only three fields ① op ② arg1 ③ arg2

Eg:- a = b* -c + b* -c

	op	Arg1	Arg2
(0)	−	c	
(1)	*	b	(0)
(2)	−	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
5	=	a	(4)

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$

(b) Triples representation of a = b * − c + b * − c;

→ using triples we refer results of an operation x op y by its position, rather than by an explicit temporary variable.

Indirect Triples :-

Indirect triples consist of listing of pointers to triples, rather than a listing of triples themselves.

→ with triples the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result., This problem doesnot occure with indirect triples.

instruction

35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	op	arg1	arg2
0	−	c	
1	*	b	(0)
2	−	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Fig: Indirect triple representation of three address code

Static single Assignment Form:- (SSA)

→ SSA is an special case of 3-address code. SSA is an intermediate representation that facilitates certain code optimizations. → In SSA each assignment to a variable should be specified with distinct names.

→ Eg:-

(a) Three-address code	(b) Static single assignment form.
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - d$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

Eg:- Intermediate program in three-address code SSA

→ The least no. of temporary variables required to create 3-address code in SSA.

* A variable can only be initialized one in L-H-S
* A variable which is initialized in L-H-S could only used R-H-S

9

Control Flow:-

Simple if, if-else, else-if, switch, for, while, do-while.

The translation of statements suchas if-else-statement and while-statement is tied with translation of Boolean Expressions.

→ Boolean Expressions are used to.

i) change the flow of control → Boolean expr are used as conditional expr in stmts that alter the flow of control.
for eg:- in if (E) S,

ii) compute the logical values → A Boolean Expression can represent true or false as value.

Boolean Expressions:-

Boolean expressions are composed of boolean operators & &&, ||, and !

→ Boolean Expressions are generated by the following grammar

$$B \rightarrow B || B \mid B \&\& B \mid !B \mid (B) \mid E \text{ rel } E \mid \text{true} \mid \text{false}$$

→ AND (or) OR are left associative

→ "NOT" has higher precedence than AND & or

Short circuit code:- (Jumping code)

In short-circuit code, the boolean operators &&, || and ! are translate into jumps.

→ In short-circuit code the 2nd (expression) argument is evaluated only if 1st argument does not suffice to determine the value of Expressions.

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

In this translation the BE is true if control reaches label L_2. If the expression is false, control immediately to L_1, skipping L_2 and the assignment x=0.

if x<100 goto L_2

if false x>200 goto L_1

ifFalse x!=y goto L_1

L_2: x=0

L_1:

fig: Jumping code

Eg2: (x==y || y==z) . . .

Flow of control statements:

→ Translation of ~~Boolean Expressions~~ control-statements into three-address code.

Grammar $S \rightarrow if\ (B)\ S_1$ (or) $S \rightarrow if\ (B)\ then\ S_1$

↳ condition (or) Boolean Expression

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

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

∴ Grammar for simple if, if-else, while statements

(1) $S \rightarrow$ if (B) then S_1 (B) is evaluated 1st

Code for simple if:	Semantic Rule:=
B.code — B.true, B.false B.true: S_1.code B.false: S.next	B.true = newlabel( ) B.false = S_1.next = S.next Intermediate code:= S.code = B.code || label (B.true) || S_1.code

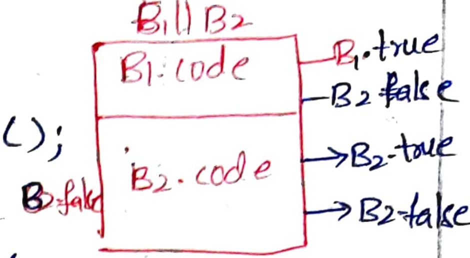
Fig:= SDD for simple if-statement

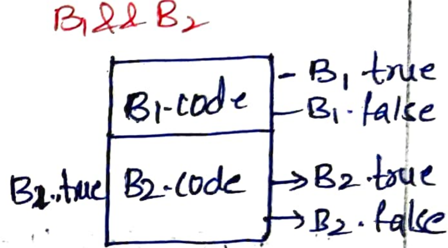
if ( )
{
}
S

↳ Here, newlabel( ) function produce, three address code for B.true.

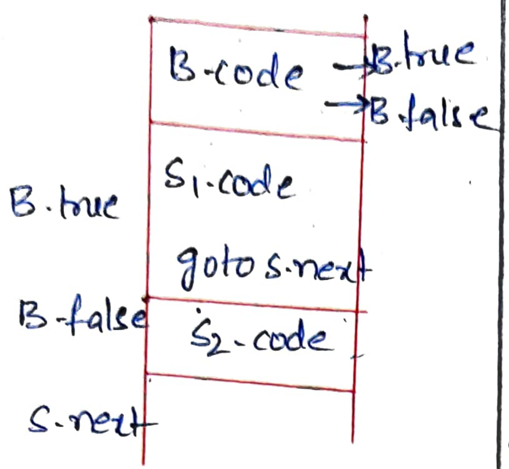
Control flow Translation of Boolean expression: (or)
Three address code for Boolean Expression (or) SDD (or) SDT for Boolean expr

production	Semantic Rules
$B \rightarrow B_1 \| \| B_2$	{ B_1.true = B.true; B_1.false = newlabel(); B_2.true = B.true; B_2.false = B.false; Intermediate code: B.code = B_1.code || label(B_1.false) || B_2.code
$B \rightarrow B_1 \&\& B_2$	{ B_1.true = newlabel(); B_1.false = B.false; B_2.true = B.true; B_2.false = B.false; B.code = B_1.code || label(B_1.true) || 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); }
$B \rightarrow E_1$ relop E_2 Eg if $\boxed{a<b}$ if a<b goto E.true goto E.false	{ B.code = E_1.code || E_2.code || gen('if' E_1 relop E_2 'goto' B.true || ~~gen('if' E_1 relop E_2~~ goto || gen('goto' E.false
$E \rightarrow (E_1)$	{ E_1.true = E.true; E_1.false = E.false; } E.code = E_1.code; }

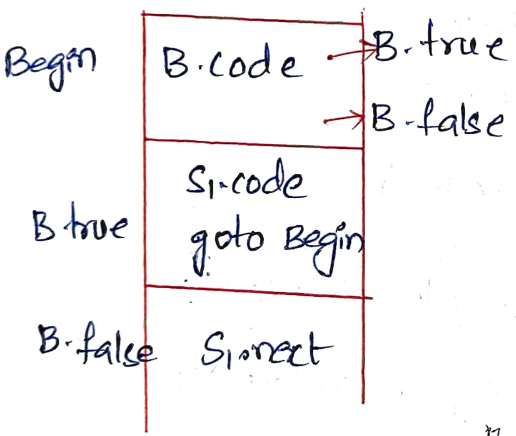




$S \rightarrow$ if (B) then S_1 else S_2

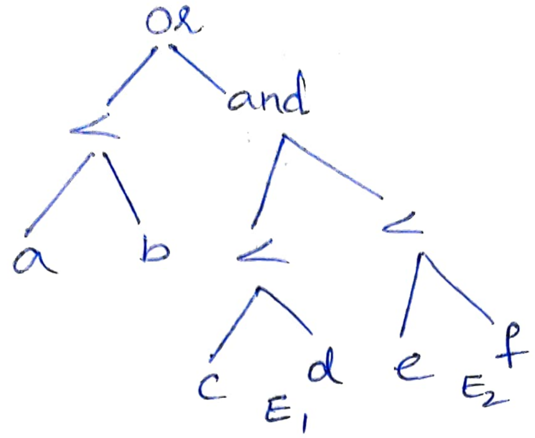
code for if-else:	Semantic rules for if-else stmt:
	B.true = newlable() B.false = newlable() S_1.next = S.next S_2.next = S.next S.code ~~Three address code~~ Intermediate code:- S.code = B.code || label (B.true) || S_1.code || gen ('goto' S.next) || label (B.false || S_2.code

(iii) while (B) then S_1

code for while	Semantic Rules
	Begin = newlabel() B.true = newlabel() S_1.next = begin B.false = S.next Intermediate code S.code = label (Begin) || B.code || label (B.true) || S_1.code || gen ('goto' begin)

production	Semantic Rules
$P \rightarrow S$	S.next = newlabel () P.code = S.code || label (S.next)
$S \rightarrow$ assign	S.code = assign-code
$S \rightarrow S_1 S_2$	S_1.next = newlabel() ~~S_2.next = S.next~~ S.code = S_1.code || label (S_1.next) || S_2.code

eg: ① a<b or c<d and e<f



if a<b goto E.true

goto E_1

E_1: if c<d goto E_2

goto E.false

E_2: if e<f E.true

goto E.false

eg:- ② if (x<100 || x>200 && x!=y) x=0

if x<100 goto L_2

goto L_3

L_3: if x>200 goto L_4

goto L_1

L_4: if x!=y goto L_2

goto L_1

L_2: x=0

L_1:

Types and Declarations:-

*Type checking uses logical rules to decide about the behaviour of program at runtime.

* It also ensures that types operand match type expected by the operators

Eg:- "&&" operation Java expect its two operands to be boolean
int * float → type error

→ Determine the storage needed

Translation Application:

compiler Translate a type of name into storage

compiler also determines the amount of storage required to store the type name at run time.

Type Expression:-

Type Expression is either a basictype or formed by applying an operator called type constructor to a type Expression.

→ T.E are used represent the structure of type,

→ T.E are premitive datatypes.

→ Type name:- is a Type Expression Eg:- ~~int~~ typedef abc int.

type Expressions are of two types.

(i) Basic type:- Basic type for language are int, real, boolean, char float, and void. A special type, type-error is used to indicate type error

Eg:- int x;   Eg:- type abc int;
int a;   is $\boxed{a=b;}$ → depend on language
abc b;

(ii) type constructor (or) Type Name:-

↳ Type constructor applied to list of type expressions.

↳ Types are formed applying an operator called type constructor to type expression.

(I) **Arrays**:- Arrays are specified as array (I, T) where I → Int (or) range of integers, T → Type.

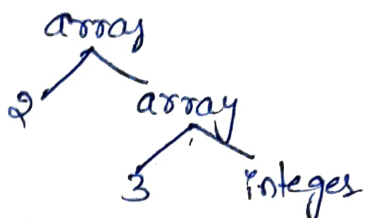
Eg:- ~~Ex~~ In "C" Declaration "int a[100]" identifies type of "a" to be array(100, integer)

↓ int [100] a;

Eg2:- T.E for int [2][3], "2 array with 3-integers".

obj name → array (2, array(3, integer)) → how many object

↓ Type Expression



Type Expression for int[2][3]

Eg3:- int a[100], b[50];

$\boxed{a=b}$ × type error.

→ A type Expression can be formed by applying array constructor to a number & type expression.

(II) **Record**:- A record is a data structure with name fields.

→ A type Expression can be formed by applying a record type constructor to the field name and their types.

Eg:- Struct st
{
int s;
float f;
};
Struct st s1;
record ((1, (int, float))
record (S1, integer)
record (S2, float)

Eg2:- Named records are product with named elements for record structure with 2 named fields.

length (an integer) and word (of type array (10, char))

the record is of type.

record ((length × Int) × (word × array(10, char)))

struct
{ int length
char word[10]
}

→ Type Expression may contain variable whose values are T.E eg: int a=5

Product:- s and t are 2 T.E then their cartesian product s×t is a type expression eg:- int * int.

Function:- Function maps a collection of types to another represented by D→R, where D is domain R is range of function.

eg:- int f1 (int x, char y, float z)
{
; return m;

Domain = (int × char × float)

Range -

o/p: int

→ T.E "int × int → char" represent a function that takes 2-integers & returns a char value

Type equivalence:-

→ Two type are said to be equivalent if and only if an operand of one type in an expression is substituted for one of the other type, without type conversion.

Type equivalence are of two types.

i) **Name equivalence**:-

The two type expression are said to be name equivalence if they they have same name & label.

Eg:- typedef int value
typedef int total
:
value var1, var2;
total var3, var4;

Eg 2: type def struct Node
{
int x;
} node;
node * first & second;
struct node *last1, *last2;

→ in the above eg1; var1 and var2 are name equivalence because their types are same.

→ var3 & var4 also Name equavalence.

→ but var1 & var3 are not name equivalent because their types are different

ii) **Structural equivalence**:-

→ If two expression are the basic type (a)

→ Formed by applying the same constructor to structurally types equivalent types then those expression are called structuraly equivalent

(i) It checks the structure of type

(ii) Determines equivalence by whether they have same constructors applied to structurally equivalent types

eg:= type array (I_1, T_1) and array (I_2, T_2) structurally equivalent if $I_1 = I_2$ & $T_1 = T_2$

$I_1 \rightarrow$ Index of array

$T_1 \rightarrow$ Type of

array a[100], b[50]

array a[100], b[100]

↓

structurally equivalent

eg1: type def int value<x

typedef int number<y

x : array (50, int)

y : array (100, int)

eg2:

S_1	S_2	Equivalence	~~Reason~~ Reason
char	char	S_1 is equivalent to S_2	similar basic type
pointer (char)	pointer (char)	S_1 is equivalent to S_2	similar constructs pointer to the char type.

Declarations :=

$D \rightarrow T$ id ; D | ε

$T \rightarrow B\ c$ | record '{D}'

$B \rightarrow$ int / float

$C \rightarrow ε$ / [num] c

D → Sequence of declarations.

T → basic ~~a~~ or array and record types

BC → 'component' - generates zero or more integers within the brackets.

- Array type consists of basic type specified by B, followed by array component C.

Eg:- int [10][11]

- Record type is sequence of declaration for field of the record all surrounded by curly braces

record {int, a}

Storage layout for local Names :=

→ compiler converts the typenames into the storage.

→ and determines the amount of storage needed to store the typename at runtime.

→ at compile time we can use these amount to assign a type name to relative address.

relative address = offset + program counter.

→ Relative & types are saved in symbol table entry for type name.

→ Data of varying length such as string or whose size cannot determined until runtime such as dynamic arrays.

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

SDT computes types and their widths for basic and array types.

T → B { t = B.type ; w = B.width; }
    C { T.type = C.type ; T.width = C.width; }

B → int { B.type = integer; B.width = 4; }

B → float { B.type = float; B.width = 8; }

C → ε { C.type = t ; C.width = w; }

C → [num] C₁ { C.type = array (num.value, C₁.type)
    C.width = num.value × C₁.width; }

Fig:- SDT for computing their types & widths

→ These declaration are represented with DAG & parsetree

Eg:- ~~int~~ parse tree for int[2] [3]

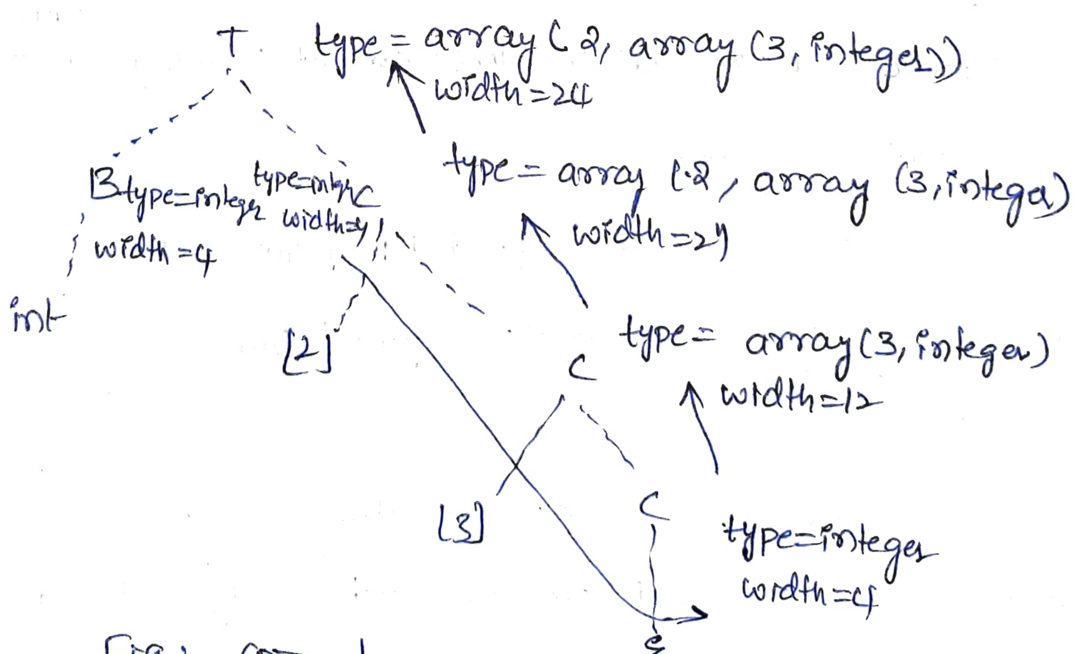


Fig: SDT of array types

Sequence of declarations:-

→ In procedures all the declaration are passed at a time.

→ all declarations in single procedure to be pas a group.

$P \rightarrow \{offset = 0;\}$ D

$D \rightarrow T\ id;\ \{top.put(id.lexeme, T.type, offset);$
(a) id T; $offset = offset + T.width;\}$
D ; D

$D \rightarrow \epsilon$

offset — is variable to keep track of the next available relative address.

$D \rightarrow T\ id\ ;\ D$ — creates a symbol table entry for executing top.put (id.lexeme, T.type, offset)

top — The current symbol table

top.put → creates a symbol table entry for id.lexeme with type & relative address.

Field in Records & classes:-

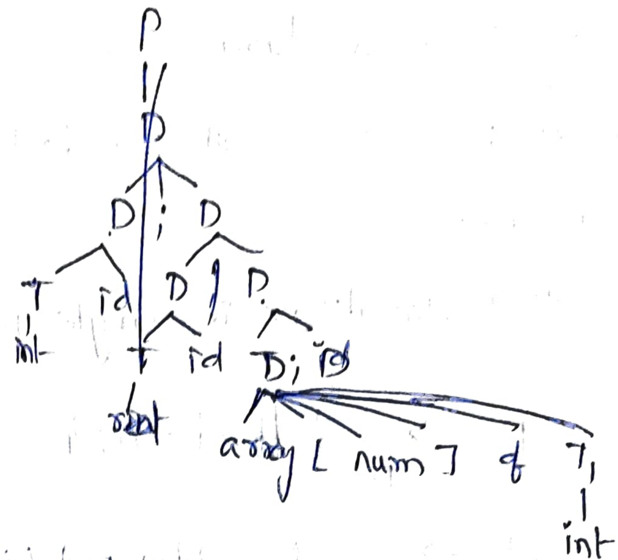
$T \rightarrow record\ '\{' D '\}'$

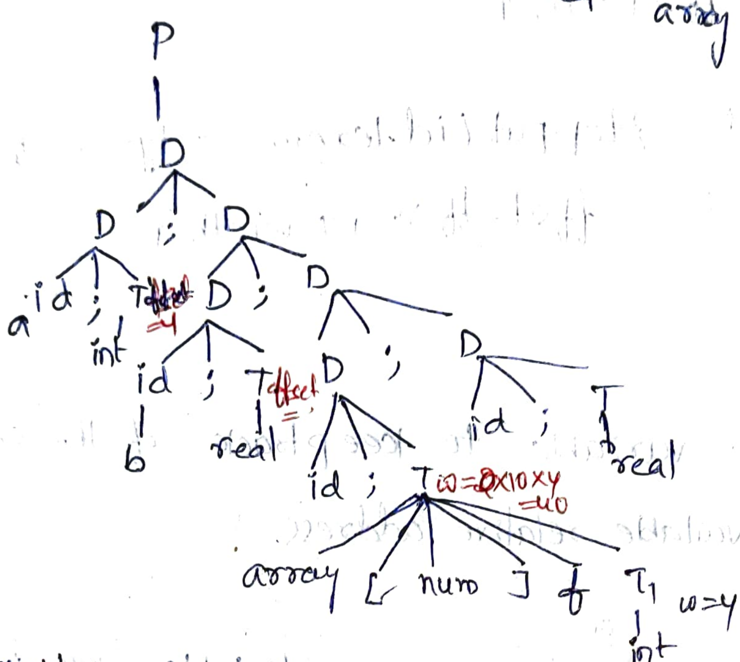
The field in this record is type specified by the sequence of declaration grouped by D.

- The field names with record must be distinct
- The offset & relative address for fieldname is relative to the data area for that record

eg:-

{
a : int;
b : real;
c : array[10] of integer;
d : ↑real;
}





Symbol table:-

Name	Type	offset	
a	integer	0 [0-3]	offset=0+4=4
b	real	4	offset=~~0~~+4+8=12
c	array(10,int)	12	offset= 12+40=52
d	real	52	

Type checking :-

→ Compiler checks whether the program is following type rules or not.

→ information about data types is maintained & Computed by compiler.

→ Type checker is a module of a compiler devoted to typechecking tasks.

→ To do typechecking a compiler needs to assign a TE to each Component of source program.

→ Compiler determines TE conform to collection of logical rules that is called type system for the source language.

→ Typechecking catch the Errors in program.

→ Assign types to values.

→ Simple situation :- check types of objects & report a type error in case of a violation.

→ more Complex :- Incorrect types may be corrected (type Coercing).

Static	Dynamic
→ Type checking done at compile time.	→ perform during program Execution.
→ properties can be verified before program run.	→ permits programmer to be less concern with c, pascal strongly.
→ can catch ~~many~~ Common Errors.	→ Mandatory in some situations such as array, bounds check.
→ Desirable when faster Execution importance	→ more robust and clearer Code

Eg: pascal, c-type

→ Type checking have been used to ~~gener~~ improve the Security of System.

Rules for Type checking:-

Type checking has two forms

i) Synthesis

ii) Interference

i) Type Synthesis :-

→ It derives the expressions from the types of its subexpressions.

→ It must be declared before they are used.

Ex:- The type of $E_1 + E_2$ is defined in terms of the types of E_1 and E_2.

> If f has type $s \to t$ and x has type s, then expression f(x) has type t

Ex:-
```
add(int a, float b)
{

}
fn1(float c, int d) → t
{
    add(2, 2.5)
}
```

$\boxed{\text{add(float, int)}}$

[∵ int changes to float, float changes to int]

→ Here f and x denote expression, $s \rightarrow t$ denote a function from s to t.

→ This rule for functions with one argument carries over to functions with several arguments.

ii) Type ~~inference~~ interference :-

→ It generally determines the type of language construct from the way it is used.

→ Ex:- $E_1 + E_2$ i,e., $\underset{int}{2} + \underset{int}{5}$ = Datatype will be int

$\underset{(string)}{abc} + \underset{(string)}{abc}$ = Datatype will be string

→ There is no need to declare variables.

→ Type inference are used in meta languages.

If $f(x)$ is an expression then for some α and β, f has a type $\alpha \rightarrow \beta$ and x has type α

Type Conversion or type Casting:-

→ A type cast is basically a Conversion from one type to another.

→ There are two types of Conversions

1) Implicit type Conversion
2) Explicit type Conversion

1) Implicit type conversion:- (smaller to bigger)

→ If a compiler Converts one data type into another type of data automatically.

→ There is no data loss

Ex:- short a = 20;
int b = a; // Implicit Conversion

Assign:- bool → char → short int → int → long → float

2) Explicit type Conversion:-

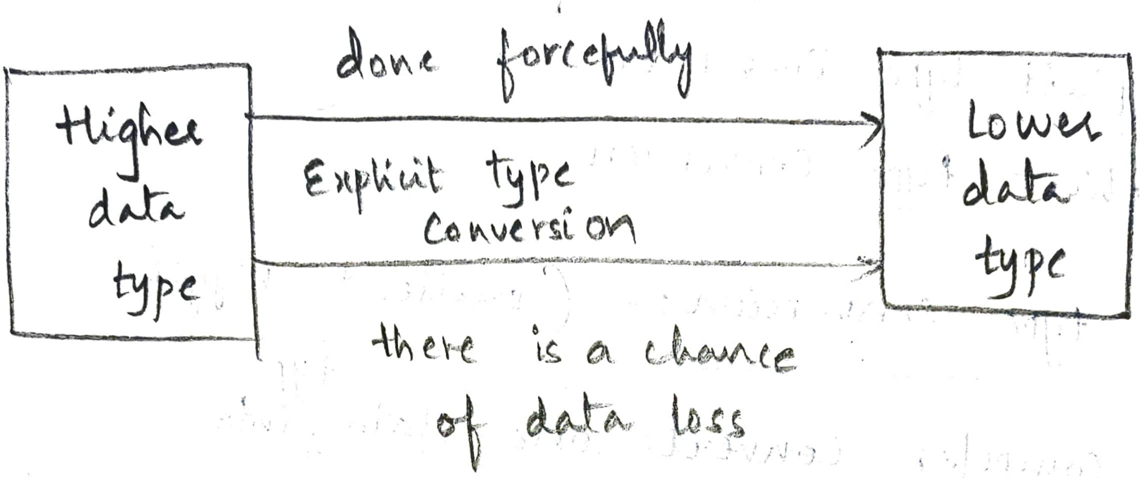
→ when data of one type is Converted explicitly to another type with the help of predefined functions.

→ There is a data loss.

→ Conversion done forcefully.

→ Some Conversions cannot be made Implicitly.

int to short ($\because$ int range is more than short so there is a chance of data loss)



Ex:- $t_1 = (\text{float})\, 2$

$t_2 = t_1 * 3.14$

Ex:- if (E_1 . type = integer and E_2 . type = Integer)

E. type = integer;

else if (E_1 . type = float and E_2 . type = Integer) ---

E. type = float;

— — — —

→ two Conversions
i) Widening Conversions
ii) Narrowing Conversions.

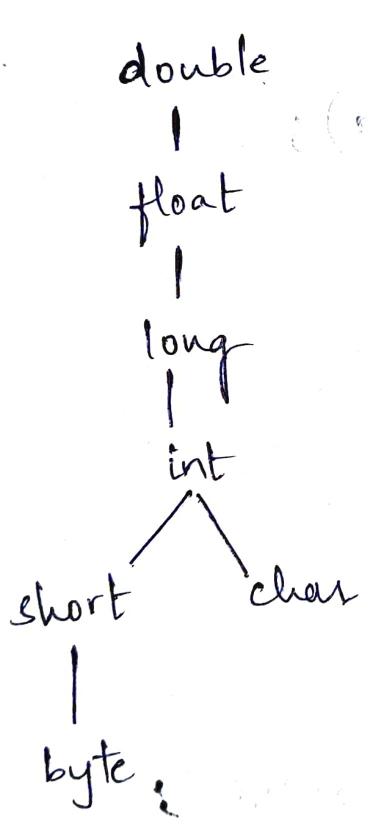
→ widening conversions generally preserve information

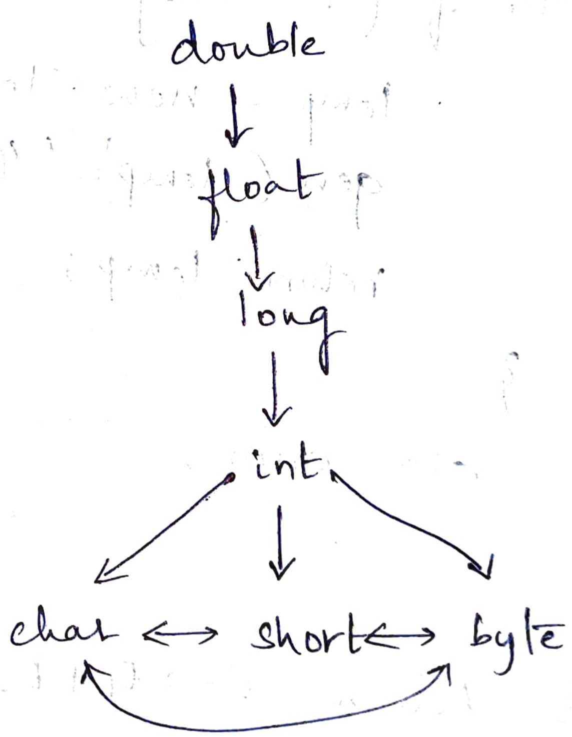
→ narrowing conversions generally lose information

→ widening rules – any lower can be widened to higher type.

→ a char can be widened to int or float but char cannot be widened to short.

→ narrowing rules – a type s can be narrowed to type t if there is a path from t.





1) max(t_1, t_2) takes two types t_1 and t_2 and returns the maximum of two types in widening hierarchy.

2) widen (a, t, w) generates type conversions if needed to widen the contents of an address a of type t into a value of type w.

```
Addr{ widen( Addr a, Type t, Type w)
    if (t=w) return a;
  else if (t = integer and w= float) {
        temp = new Temp();
        gen ( temp'=' '(float)' a);
        return temp;
  }
    else error;
}
```

→ Semantic Action $E \rightarrow E_1 + E_2$

```
E→E1+E2 { E.type = max (E1.type, E2.type);
          a1 = widen(E1.addr, E1.type, E.type);
          a2 = widen (E2.addr, E2.type, E.type);
        E.addr = new Temp( );
        gen( E.addr'=' a1'+'a2); }
```

Overloading of Functions and operators:-

An overloaded Symbol has different meanings depending on its context. overloading is resolved when a unique meaning is determined for each occurence of a name.

Ex:- The + operator in Java denotes either string concatenation or addition, depending on the type of its operands.

```
void err() { - - - }
void err(String s) { - - }
```

Intermediate code for switch statements (a) Three Address code

Translation of switch statement:

Switch statement syntax:

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 statement:

Code to Evaluate E into t
goto test
L_1 : code for s_1
goto next
L_2 : code for s_2
goto next
- - -
- - -
L_{n-1} : code for s_{n-1}
goto next
L_n : code for s_n
goto next
test : if $t = v_1$ goto L_1
if $t = v_2$ goto L_2
- - -
if $t = v_{n-1}$ goto L_{n-1}, goto L_n
next:

Eg := switch (x+y)
{ case 1 : a=a+2;
break;
case 4 : b=b*5;
break;
case 6 : c=c/2;
break;
default : d=d-2;
break;
}

Three Address code:=

if $t = v_{n-1}$ goto L_{n-1}
goto L_n
next:

1. $t_1 = x+y$
2. If $(t_1 = 1)$ goto 8
3. If $(t_1 = 4)$ goto 11
4. If $(t_1 = 6)$ goto 14
5. $t_2 = d-2$
6. $d = t_2$
7. goto Next
8. $t_3 = a+2$
9. $a = t_3$
10. goto Next
11. $t_4 = b*5$
12. $b = t_4$
13. goto Next
14. $t_5 = c/2$
15. $c = t_5$
16. goto Next
17. Next

if $q = V_{n-1}$ goto L_{n-1}

goto Ln

next:

Intermediate code for ~~product~~ procedures:- (a) Three Address code

D → define T id (F) {S}

F → ε | T id, F

S → return E;

E → id (A);

A → ε | E, A

→ S → adds stmts that returns the value of an expression.

→ E → adds function calls, with actual parameters A.

Non-terminals D and T generates declarations and types.

→ Function definition generated by D consists of keyword define, a return type, the function name, Formal parameters in paranthesis and function body consisting of statements.

→ Non-terminal F generates zero or more Formal parameters.

where Formal parameters consists of a type followed by identifies

→ Non-terminal S & E generate statements and expressions.

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

→ In three-address code, a function call is unraveled into the evaluation of parameters in preparation for a call, followed by call itself. and the parameters are passed by value.

Eg:- If the given function is in the form of

P(A1, A2, A3, --- An)

param A1

param A2

⋮

param An

call P, n

P → is function name.

n → no. of arguments.

Eg2:- n = f(a[i]);

Translated into three-address code as follows

1) $t_1 = i * 4$
2) $t_2 = a[t_1]$
3) param t_2
4) t_3 = call f, 1
5) $n = t_3$

→ The first 2 lines compute the value of expression a[i] into temporary t_2,

→ line 3 makes t_2 an actual parameter for the call on line 4 of f with one parameter

→ line 5 assign the value returned by the function call to t_3.

Functions Types:-

→ The type of function must encode the return type and types of the return type and the types of the formal parameters.

→ Let "void" be a special type that represent no parameter or no return type.

→ whenever the function is called the function name is name entered into the symbol table for use in the rest of the program.

→ The Formal Parameters are stored in the Activation Record. For storing Formal parameters the Activation Records are used.

Eg2:- void main()

```
{
int x, y
 - - -
 - - -
swap(&x, &y);
}
void swap(int *a, int *b)
{
int i;
i=*b;
*b=*a;
*a=i;
```

Three address code

1. call main
2. param &x
3. param &y
4. call swap, 2

Eg3:- float add() or float add(int a)

Formal parameters

```
float add(int a, float b)
{
return add() or return add(x),
    return add (x,y);
}
```

Actual parameters