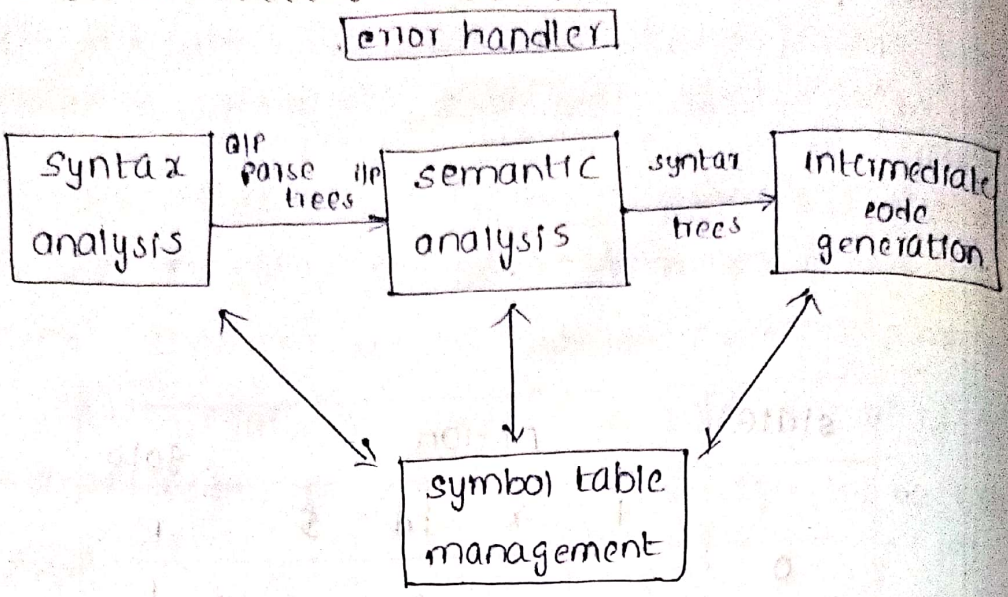
7/3/24

unit-3

* Role of semantic analyzer :-



* The input the semantic analysiser is the third phase of the compiler which is the last stage of the fornterd It takes the input as the syntax trees and make some modifications, finally it produces the output has syntax trees (or) parse trees

* In semantic analysis compiler analysis the meaning of the program and it is carried out by scanning for input starting text which is static in nature.

The properties that can't be are not captured by the complexity CFG grammar are (or) captured by the semantic roots. This process parsers should be named main scope, analysis., Type checking and Type conversion

The need for the semantic analysis is to build a Symbol table that keep each declarations. the identifier type of names in

The datatype of the identifier is obtained and type checking of the whole expression and statements is done in order to follow the type rules of the language

operators precedences parsing :-

A grammar G is called operator precedence grammar it means the following 2 conditions are :-

1) there is no exist of 'ε' Rule in the rightside in production
2) there exist no production rule which contains 2 nonterminals adjacent to eachother on its rightside.

* A parser that reads and understand operator precedence grammar (or) parser is called operator precedence parser.

1) For example, the grammar,

E → EAE | (E) | -E | id.

A → + | - | * | / | ↑

sol: convert the above grammar into an operator precedences grammar.

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id.

operator precedence can only establish between the terminals of the grammar. it ignores the non-terminals

parsing actions :-

1) Both end of the given input string add '$' symbol.
2) scan the input string from left to right until the '>' symbol is encountered
3) scan the words left overall the equal precedences until the first left most is '<' is encountered
4) Every thing between left most '<' and right most '>' is handled
5) Dollar ($) on dollar ($) means parsing is successfull

* They are three operators precedences relation.

1) a > b — terminal a has high precedence than b
2) a < b — terminal b has high precedence than a.
3) a = b — both terminals a and b has same precedences

Rules for the precedence table

1) id, a, b, c have highest priority
2) '$' has lowest priority
3) whenever same precedence occur always we have to use the greater than symbol.
4) for id on id is not equal
5) for $ on $ is accepted

* consider the grammar and construct the operator precedence grammar.

E → EAE|id
A → +|*

step ① : check operator precedence or not
step ② : construct the precedence relation table
step ③ : parse the string {t, x, id, $}
step ④ : generate the parse Tree.

step ①:
convert to the operator precedence.

E → E+E| E*E|id

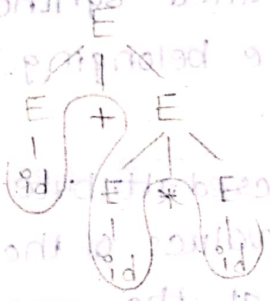
step ② : construct the precedence relation table.

	+	*	id	$
+	>	<	<	>
*	>	>	<	>
id	>	>	x	>
$	<	<	<	Accept

step ③ :
parse the string id+id*id

stack	relation	input	action
$	~~< id+id*id$~~	id+id*id$	shift
$ id	>	+id*id$	reduce E→id
$ E	<	+id*id$	shift
$ E+	<	id*id$	shift
$ E+id	>	*id$	reduce
$ E+E	<	*id$	reduce.
$ E	<	*id$	shift
$ E*	<	id$	shift
$ E*id	>	$	reduce.
$ E*E	>	$	reduce
$ E	accept	$	accept

step(4): generate the parse tree.



* SDT- syntax directed Translation (or) directed Definition

* In SDT every non-terminal can get zero or more attributes depending on the type of the attributes
* In semantic rule the attributes can hold string or a number or a memory location. It is in the form of 'val'
* In syntax directed translation along with the gramm -ar the associate some informal notations which are called as semantic rules.

* In semantic evaluation at the nodes of the syntax tree values of attributes are de on the given input string. such a parse tree containing the values of attributes at each node is called an annototed (or) decorated parse tree.

* syntax directly definitation is a generalization of context free grammar in which each production $F \rightarrow \alpha$ is associated with a set of semantic rules

Attribute:-

* The attribute can be a string a number, a type, a memory location or anything else.
They are two types of attributes
1) synthesized attribute
2) inherited attribute.

1) synthesized:-

The attribute 'a' is called synthesized attribute of 'x' and $b_1, b_2 \cdots b_k$ are belonging to the production symbols.

* The value of synthesized attribute at a node is computed from the values of the attributes at the children of the node at the parse trees

2) inherited Attribute:-

The attribute 'a' is called inherited attribute or some of the grammars in the right side of the production. and $b_1, b_2 \cdots b_k$ are belonging to the either 'x' (or) 'α'

* the inherited attributes can be computed. from the values of the attributes at the siblings and the parent of that node.

* construct the grammar, construct the syntax tree, parse tree and annotated parse tree for the expression.

2+3*4.

E -> E+T | T

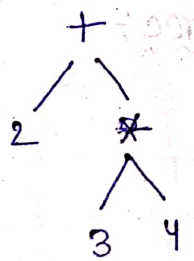
T -> T*F | F

F -> id.

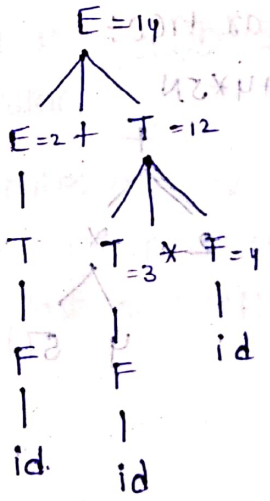
sol:

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 -> id.	F.val = id.val digit.lexvalue

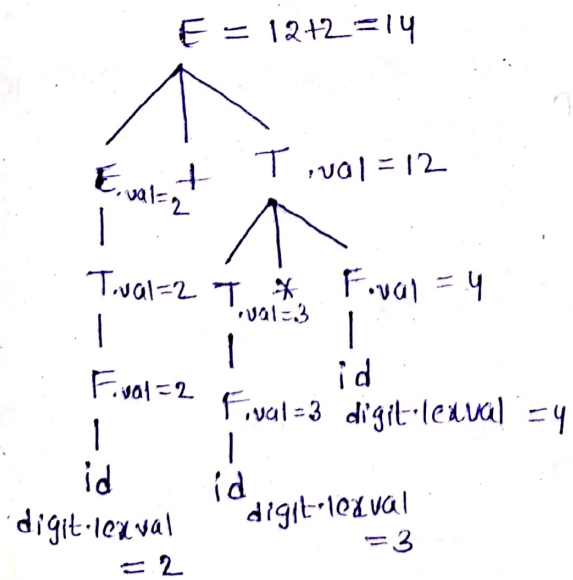
syntax tree :



parse tree :



annotated parse tree :



consider the grammar, $S \rightarrow EN$ for the expression

$2+4*5N$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id.$

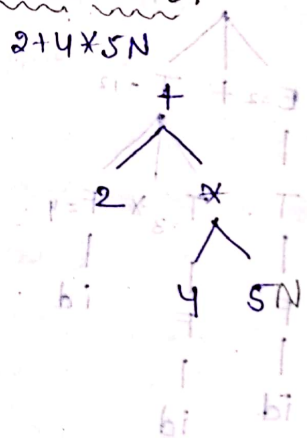
∴ N — Newline. (;)

soln

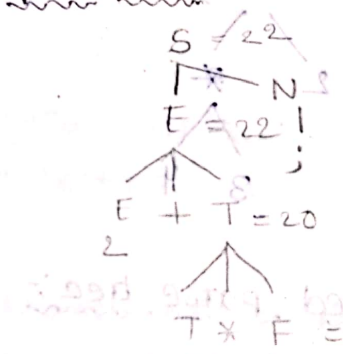
production	semantic rules
$S \rightarrow EN$	print (E.val)
$E \rightarrow E+T$	E.val = E.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T*F$	T.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow id.$	F.val = digit.lexval

syntax tree:

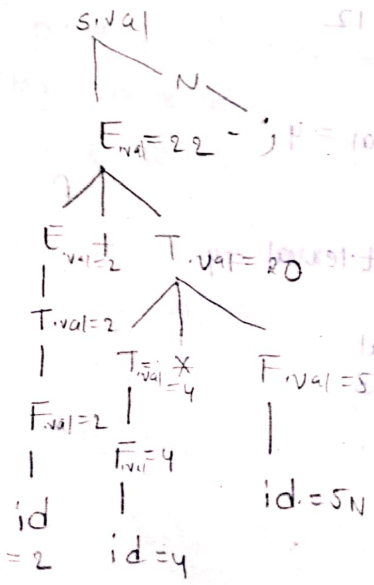
$2+4*5N$



parse tree:



annotated parse tree.



Types of syntax directed translation.

* They are two ways of syntax directed translation.

1) S-attributed SDT (synthesized)
2) L-attributed SDT (left)

S-attributed SDT	L-attributed SDT
1) It is a uses synthesized Attributes	1) It is a both uses in synthes and inherited attribute In inherited attributes can inherit values from parents (or) left siblings of the node so it is L-attributed SDT
2) semantic actions are placed at the rightend of the production.	2) semantic attributes are placed at any where on right hand side.
3) Attributes are evaluated during bottom -up parsing.	3) Attributes are evaluated by traversing parse tree has depth first and left to right.

* L-attributed SDT

consider the grammar D → TL
T → int
T → real
L → L, id
L → id.

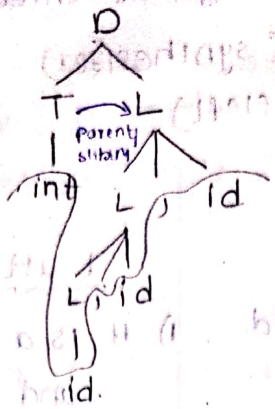
construt the elaborated

parse string id_1, id_2, id_3

production	semantic rules
D → TL	L.in = T.type
T → int	T.type = integer
T → real	T.type = real
L → L_1	L.in = L.in addtype(id.entry, L.in)
L → id	addtype(id.entry, L.in)

Soln

annotated parse tree:



* Evaluation order for syntax directed Translation:

1) Dependency graph:-

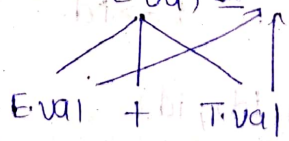
* It is used to represent the flow of information among the attributes in a parse string.
* It helps to determine the evaluation order for the attributes in a parse tree.
* The compiler has to check for various types of dependency between the statements in order to prevent them from being executed in the incorrect sequence.
* while annotated parse tree shows the values of attributes but dependancy graph shows how values can be computed.

Ex:-

1) $E \rightarrow E+T$

semantic rule - E.val = E.val + T.val

annotated parse tree. —



for '+' the associativity is left to right.

2) construct the grammar, for expression 2+3*5

$E \rightarrow E+T \mid T$

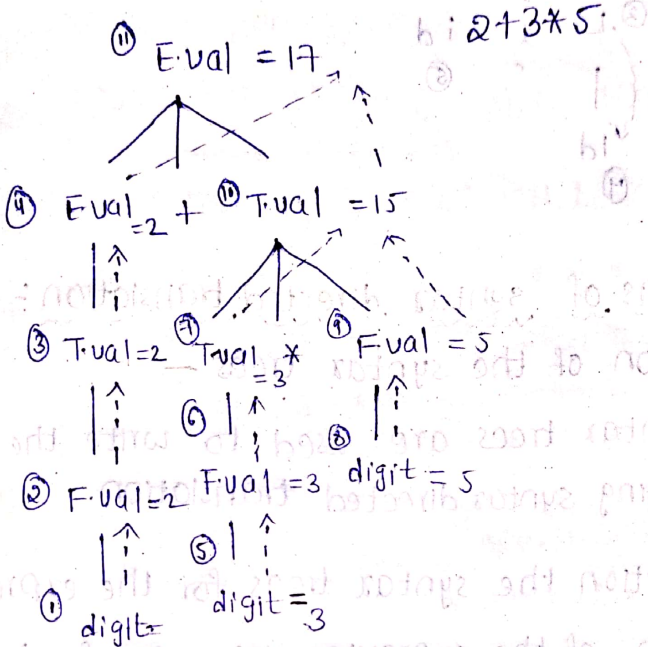
$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow digit$

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.

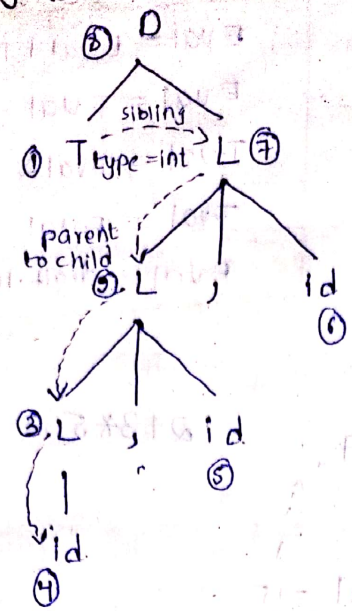
dependency graph:-



3) D → TL
T → int
T → real
L → L, id
L → id

production	semantic rules
D → TL	L.in = T.type
T → int	T.type = integer
T → real	T.type = real
L → L, id	L.in = L.in addtype(id.entry, L.in)
L → id	addtype(id.entry, L.in)

Dependency graph :-



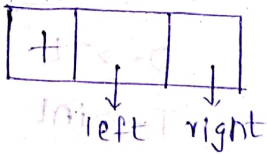
* Applications of syntax directed translation :-

1) construction of the syntax trees —

The syntax trees are used to write the transmission routines using syntax directed translation.

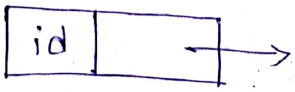
* construction the syntax trees for the expression means Translation of the expression into post fix form.
* The nodes for each operator and operand is created so each node can be implemented as a record with multiple fields.
* The following syntaxes for the nodes are :-

1) mknode (op, left, right) — This function creates a node with the field operator having a label with two pointers to left and right.



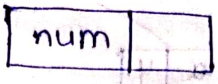
2) mkleaf (id, entry) —

This function creates an identifier node with labele -d id and pointed to the symbol table is given by the entry



3) mkleaf(num, val) –

This function creates a node for a number with label num for value of that number.



* construst the syntax tree for the Expression x-y*5+z for the grammar E → E+T | E-T | E*T T→ id/num

Sol:

step①: convert the expression from prefix to postfix.

step②: Make use of the functions like mknode, mkleaf, mkleaf(num, val).

step③: write the sequence of call based on the Expression

step①: convert the expression into postfix form.

xy * 5 - z +

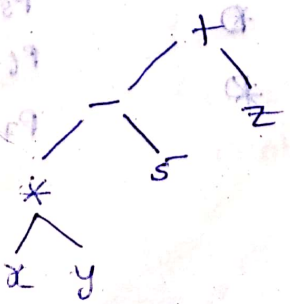
step②:

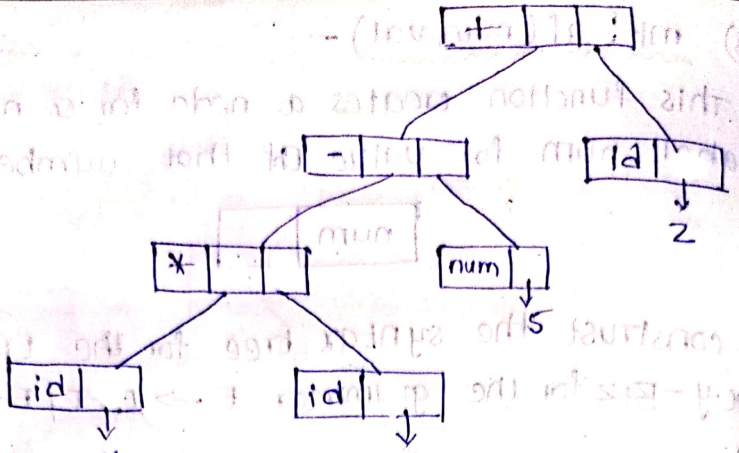
symbols	operations
x	P_1 = mkleaf(id, entry x)
y	P_2 = mkleaf(id, entry y)
*	P_3 = mknode(*, P_1, P_2)
5	P_4 = mkleaf(num, 5)
-	P_5 = mknode(-, P_3, P_4)
z	P_6 = mkleaf(id, entry z)
+	P_7 = mknode(+, P_5, P_6)

step③:

x * y - 5 + z

xy * 5 - z + – postfix form





The syntax directed translation in the above grammar.

production rule	semantic operations
E → E+T	E.nptr = mknode('+', E.nptr, T.nptr)
E → E-T	E.nptr = mknode('-', E.nptr, T.nptr)
E → E*T	E.nptr = mknode('*', E.nptr, T.nptr)
E → T	E.nptr = T.nptr.
T → id	T.nptr = mkleaf(id, id.entry)
T → num	T.nptr = mkleaf(num, num.val)

*construct the syntax tree by using the expression.

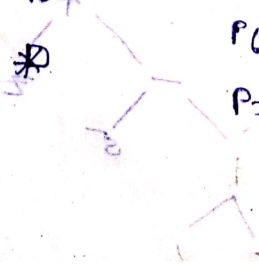
A*B+C*D. a*b+c*d

Soln

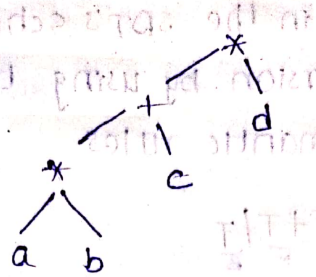
Step1 :- AB*C+D* (or) ab*c+d*

Step2 :-

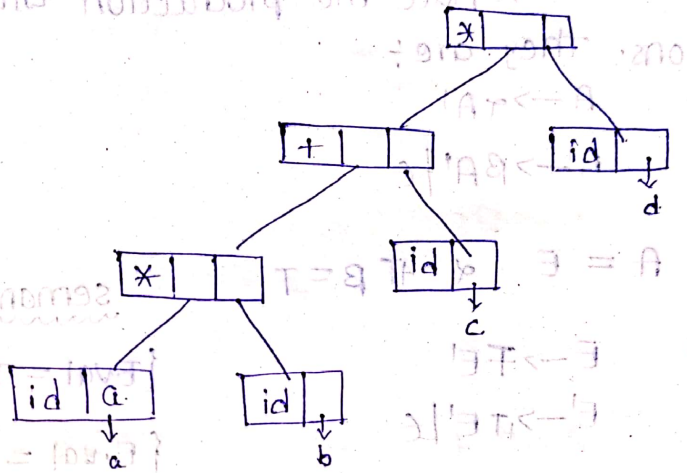
symbols	operations
A	P_1 = mkleaf (id, entry A)
B	P_2 = mkleaf (id, entry b)
*	P_3 = mknode (*, P_1, P_2)
C	P_4 = mkleaf (id, entry c)
+	P_5 = mknode (+, P_3, P_4)
D	P_6 = mkleaf (id, entry D)
*	P_7 = mknode (*, P_5, P_6)



step3:- ab*c+d* syntax tree.



step4:-



Syntax directed translation schemes:-

* It is a context Free grammar is used to evaluate the order of semantic rules.

* In translation scheme the semantic rules are embedded with the right side of the production.

* the position at which an action is to be executed and it show can enclosed between the curly braces it is written within in the right side of the production

Postfix translation schem

Ex:-

$S \rightarrow E$	{$S.val = {print E.val}}
$E \rightarrow E+T$	{print E.val = Eval+T.val}
$E \rightarrow E*E$	{print E.val = E.val * E.val}
$E \rightarrow I$	{print E.val = I.val}
$I \rightarrow$ digit	{print I.val = digit.lexval}

2) Removal of left recursion:

Ex: consider the grammar if the grammar consists of left recursion in the SDT's scheme then we can remove the left recursion by using the productions and later using the semantic rules.

Ex: $E \rightarrow E+T \mid T$

The above production is of the form as follows.

$A \rightarrow A\alpha \mid \beta$

we can replace the production with two other productions. They are:

$A \rightarrow \alpha A'$

$A' \rightarrow \beta A' \mid \epsilon$

$A = E \quad \alpha = +T \quad \beta = T$

semantic rules

1) $E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

{E.val = T.val, E'.val}

E'.val}

2) $T \rightarrow T*F \mid F$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

{E'.val = null}

{T.val = F.val, T'.val}

{print("*"), F.val, T'val}

{T'.val = null}

3) A parser stack implementation of postfix SDTS:-

This is implemented during the LR parser where at the top of the stack containing the symbols along with the synthesized attributes.

productions:

$S \rightarrow N$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

* Implementing L-attributed definitions:

The syntax directed definition can be defined as the L-attribute for the production rule $A \rightarrow X_1 X_2 X_3 \ldots X_n$ where the inherited attributed x_k is such that $1 \le k \le n$,

* The production $A \rightarrow X_1 X_2 \ldots Xn$ is depends upon on the attributes of the symbol and also depends on inherited attribute 'A'

* so every S-attributed definition is also L-attributed definition

Example: check whether the given SDT is L-attributed or not?

sol:

production	semantic action	clause of attribute
$A \rightarrow PQ$	$P.in = A.in$	L-attributed.
	$Q.in = P.sy$	L-attributed
	$A.sy = Q.sy$	L-attributed.
$A \rightarrow XY$	$X.in = A.in$	L-attributed
	$X.in = Y.sy$	L-attributed-not
	$A.sy = X.sy$	L-attributed

* write S-attributed grammar to convert given grammar with infix operators to the prefix operators

productions	semantic rules
$L \rightarrow E$	L.val = E.val
$E \rightarrow E+T$	E.val = +, E.val, T.val
$E \rightarrow E-T$	E.val = -, E.val, T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T*F$	T.val = *, T.val, F.val
$T \rightarrow T/E$	T.val = /, T.val, E.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow F^P$	F.val = ^, F.val, P.val

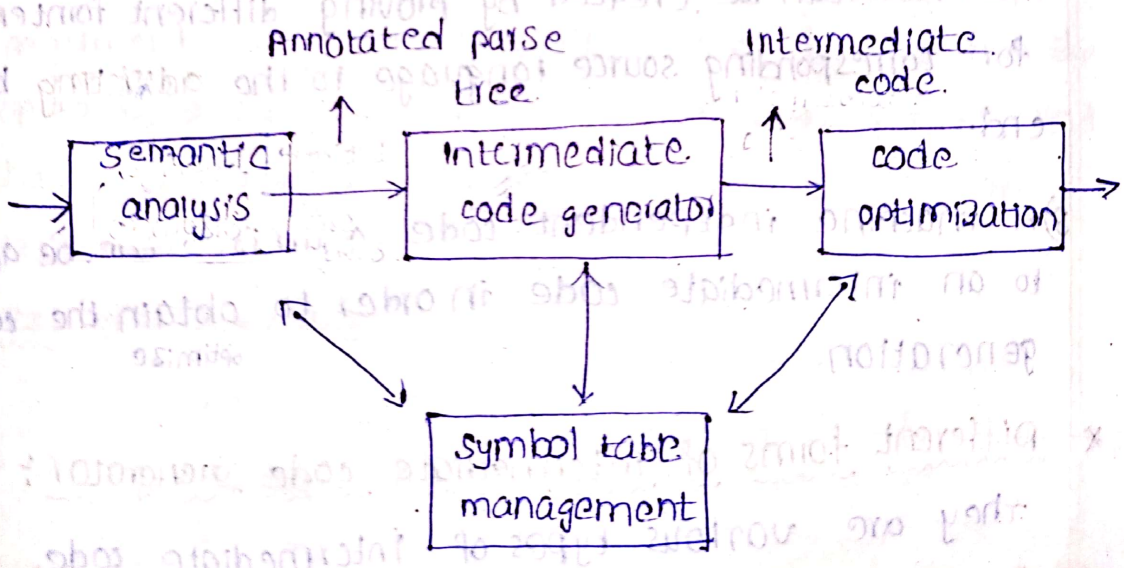
F → P	F.val = P.val
P → (E)	P.val = (), E.val
P → id	P.val = digit.lexval

write the syntax tree functionalities in the above grammar.

productions	syntax tree functionalities
L → E	L.nptr = E.nptr
E → E+T	E.nptr = mknode(+, E.nptr, T.nptr)
E → E−T	E.nptr = mknode(−, E.nptr, T.nptr)
E → T	E.nptr = T.nptr
T → T*F	T.nptr = mknode(*, T.nptr, F.nptr)
T → T/F	T.nptr = mknode(/, T.nptr, F.nptr)
T → F	T.nptr = F.nptr
F → F^P	F.nptr = mknode(^, F.nptr, P.nptr)
F → P	F.nptr = P.nptr
P → (E)	P.nptr = mknode((), E.nptr)
P → id	P.nptr = mkleaf(id, idEntry)

* Intermediate code generator :- It is the 4th stage of compiler design.

role of intermediate code generator :-



* Intermediate code generator is the 4th phase of the compiler. sometimes Intermediate code generators acts as a fornt end and also backend.

* The input is the Intermediate code generator is the annotated parse trees and output will be Intermediate code.

* The Intermediate code generator effects some part of the source language and also affected some part of the target language.

* It is not always possible to generate machine code directly in one pass. so compiler generate an easy to represent form of the source language which is called Intermediate code language.

* In this generation of an intermediate langu leads to efficient code generation.
-age

* Advantages of intermediate code language:

1) A compiler for different machines can be created by attaching different backends to existing fointends of each machine.

2) A compiler for different source languages (same machines can be created by proving different fointends for corresponding source language to the existing back end.

3) A machine independent code optimizer can be applied to an intermediate code in order to obtain the code generation.
optimize

* Different forms of intermediate code Generator:
They are various types of intermediate code.

1) 3-address code
2) quatrables
3) Triples
4) Indirect triples

5) abstract syntax trees
6) polish notations

1) 3-address code:-

In 3-address code form at the most pre 3- addresses are used to represent any statement. The general form of cerators code is $\boxed{a = b \text{ op } c}$

ex:- In 3-address code for the expression x+y*z is

$t_1 = y * z$
$t_2 = x + t_1$ } - at most 2 variables

* Represent the expression a=b+c+d in 3- address code format

$t_1 = b + c$
$t_2 = t_1 + d$
$a = t_2$

Implementation of 3 -address code.

1) Quatrables
2) Triples
3) Indirect Triples

1) Quatrables

It is a structure with atmost 4 fields such as op, argument1, argument2, result. The op field is used for operator, argument1, arg2 represent two oprands. and the result field is used to store the result of an expression.

ex:- a=-b*c+d

$t_1 = -b$
$t_2 = t_1 + d$ } - 3 address code
$t_3 = t_2 * t_2$
$a = t_3$

Now, write the quatrables form.

operator	argument1	arg 2	result
uminus	b	—	t_1
+	c	d	t_2
*	t_1	t_2	t_3
=	t_3	—	a

* construct Quatrables for the expression a*-(b+c)

$t_1 = b+c$

$t_2 = a^*$

$t_3 = t_2 - (t_1)$ (b+c)

$t_4 = t_1 * t_2$

operator	arg1	arg 2	result
=	a	—	t_1
+	b	c	t_2
−	t_2	—	t_3
*	t_1	t_3	t_4

construct quatrables $-(a*b)+(c+d)-(a+b+c+d)$

$t_1 = a*b$

$t_2 = -(t_1)$

$t_3 = c+d$

$t_4 = t_2 + t_3$

$t_5 = a+b+t_3$

$t_6 = t_5(t_5)$

$t_7 = t_4 + t_6$

operator	arg1	arg2	result
*	a	b	t1
-	t1	-	t2
+	c	d	t3
+	t2	t3	t4
+	a	b	t5
+	t3	t5	t6
-	t4	t6	t7

Triples :-

The use of the temporary variables is avoided by references pointer in the symbol table.

$a = -b * c + d$

$t_1 = -b$ $t_3 = t_1 * t_2$

$t_2 = c + d$ $a = t_3$

operator	arg1	arg2
uminus	b	-
+	c	d
*	t1	t2
=	t3	-

→ Triples

	operator	arg1	arg2
[0]	uminus	b	-
[1]	+	c	d
[2]	*	[0]	[1]
[3]	=	[2]	-

2) $t_1 = a$

$t_2 = b + c$

$t_3 = -t_2$

$t_4 = t_1 * t_3$

	operator	arg_1	arg_2
[0]	=	a	—
[1]	+	b	c
[2]	uminus	[1]	—
[3]	*	[0]	[2]

3) Indirect Triples :

In the indirect triples representation the listing of the Triples is being done and listing the pointers are used instead of using the statements.

ex: a = - b * c + d construct the indirect triples for the expression.

Indirect triples

	operator	arg_1	arg_2
[0] (11)	uminus	b	—
[1] (12)	+	c	d
[2] (13)	*	(11)	(12)
[3] (14)	=	(13)	—

(0)	(1)
(1)	(2)
(2)	(3)

ex: a * - (b + c)

	operator	arg_1	arg_2
(0) (11)	=	a	—
(1) (12)	+	b	c
(2) (13)	uminus	(12)	—
(3) (14)	*	(11)	(13)

3) $-(a*b)+(c+d)-(a+b+c+d)$

Triples

	operator	arg_1	arg_2
[0]	*	a	b
[1]	uminus	(0)	—
[2]	+	c	d
[3]	+	(1)	(2)
[4]	+	a	b
[5]	+	(4)	(2)
[6]	−	(3)	(5)

Indirect Triples

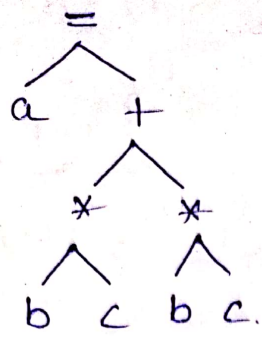
	operators	arg_1	arg_2
(0) (11)	*	a	b
(1) (12)	uminus	(11)	—
(2) (13)	+	c	d
(3) (14)	+	(15)	(13)
(4) (15)	+	a	b
(5) (16)	+	(15)	(13)
(6) (17)	−	(14)	(16)
(7)			

DAG:- Directed Acyclic graph - abstract syntax tree

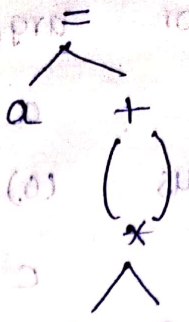
It is similar to the syntax tree, but in the directed Acyclic graph we have to identified the common subexpressions to represent the syntax tree

Ex: $a=b*c+b*c$

syntax tree



DAG graph.



* Abstract syntax tree :-

A hierarichal structure is represented like a syntax trees. Directed Acyclic graph (or) DAG is very similar to the syntax trees but they are in more compact form

Algorithm for construction of DAG :-

We Assume the 3-address code statements of following size of types :-

case1 :- x op y

case2 :- x = op - y

case3 :- x = y

* The DAG is constructed for the following steps :-

1) if y is undefined then create node y then similarly if z is undefined then create node z.

2) i) for case '1' create a node operator whose left child is node 'y' and node 'z' will be right child and also check for the common subexpression.

ii) for case '2' determined wheather a node labelled operator such a node we have a child node.

iii) In case 3 node will be a node 'y'.

3) delete 'x' from the list of the identifiers from node 'x', append 'x' to the list of attached identifiers. for node found in case 2.

Applications of DAG:-

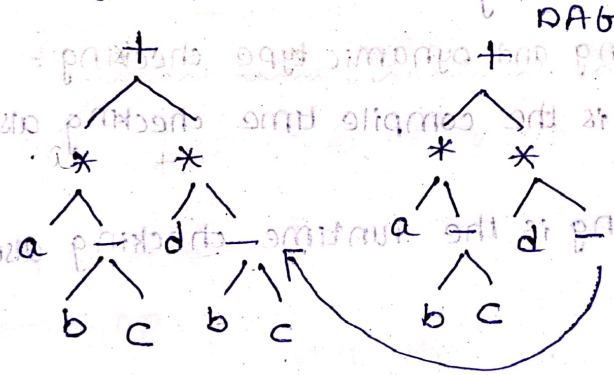
1) determining the common subexpressions
2) determining the which names are used inside a block and computed outside the block.
3) determining the which statements of the block could have a computed value outside the block.
4) simplified the list of quadrupules by eliminating the common subexpressions and not performing the assignment of the form. $x=y$ and unless x is a must and until

1) $a*(b-c)+d*(b-c)$

2) $a+(b-c)*(b-c)+d$

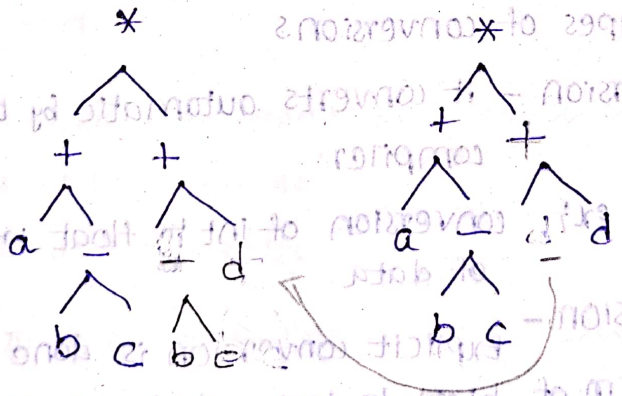
3) $-a+b*c-d+e-b*c$

1) Syntax tree

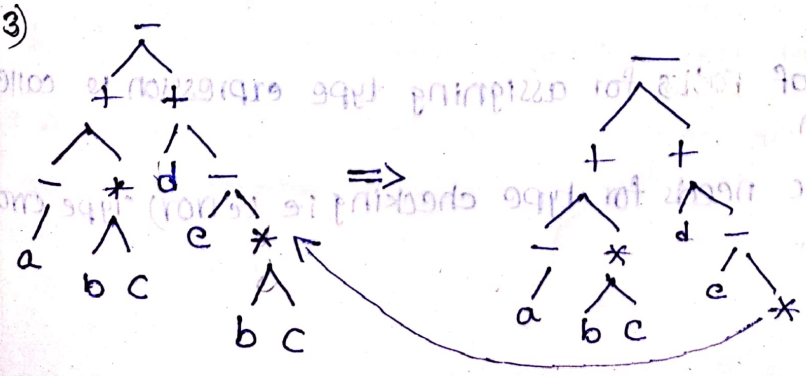


DAG

2)



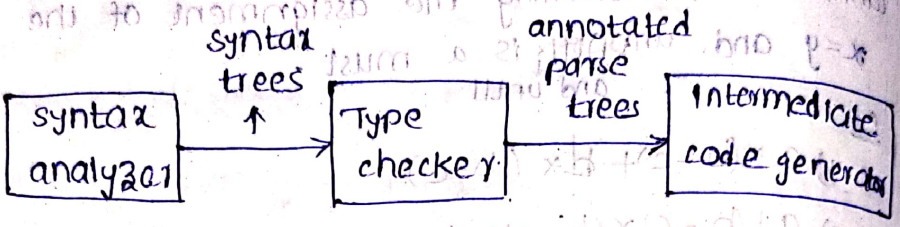
3)



* Type checking :

This is the important activity in semantic analysis. the main aim of the type checking is to detect errors in the expressions, the errors may occur due to incompatible types or operands. (not balancing)

Role of Type checker :



*they are two types of type checking :

1) static type checking - com
2) dynamic type checking

* static type checking and dynamic type checking :

* static checking is the compile time checking also called as early binding
* dynamic checking is the runtime checking also called as late binding.

Type conversions

conversion of from one datatype to another.

they are two types of conversions

1) Implicit conversion - it converts automatic by the compiler

ex: conversion of int to float without loss of data.

2) Explicit conversion - Explicit conversion is done by the users which is form of high to low which causes loss of data.

* collection of rules for assigning type expression is called type system.
* Two basic needs for type checking ie (error) type error and viod.

* Type error reports errors and void indicates no datatype.
* Type construction is also a type expression.

Polish notation

* Another name for polish notation is postfix notation, suffix notation, reverse polish notation.
* postfix notation is the linear representation of syntax trees
* i) For example the expression can x+y can be written as xy+
ii) $z=(a-b*(c-d)+(e+(f/g)))$
$= ab-*cd-+(E+fg/) = ab-cd-*efg/++$
iii) $(a+b)*c = (ab+)c* = ab+c*$
iv) $a*(b+c) = a*(bc+) = abc+*$
v) $(a+b)*(c+d) = (ab+)*(cd+) = ab+cd+*$
vi) productions
$E \rightarrow E_1 OP E_2$
$E \rightarrow (E)$
$E \rightarrow id$

write the semantic rules in the postfix notation.

sol: $E.code = E_1.code \,||\, E_2.code \,||\, op$
$E.code = E.code$
$E.code = id.entry$

* Types and declarations:-

* the languages such as c, pascal, FORTRAN allows all declaration in a single procedure to be processed as a group
* A global variable offset can be track of the next available relative address

production	semantic rules
$P \rightarrow D$	{offset = 0}
$D \rightarrow id:T$	{enter(id.name, T.type, offset); offset = offset + T.width}
$T \rightarrow integer$	{T.type = integer; T.width = 4}
$T \rightarrow real$	{T.type = real; T.width = 8}
$T \rightarrow array[n]\ of\ T$	{T.type = array(num.val, T.type), T.width = num.val × T.width}
$T \rightarrow \uparrow T_1$	{T.type = pointer(T1.type), T.width = 4}

* The non-terminal 'P' generates the sequence of declaration of the form id:T initially the offset is set to 0.

* The name is entered to the symbol table with offset = current value of the offset and increment by the width of the data object denoted by the name.

* The function enter(name, type, object) creates a symbol table entry for name, type and object in its data area.

* we use synthesised attributes type and width for T. Type represents the basic type of expression like integer and real. By applying type constructors, pointers, array.

* The width of the array is obtained by multiplying the width of each element by number of elements in the array. The width of each pointer is assumed to be '4'.

Translation of expression:

production	semantic Actions
$S \rightarrow id = E$	S.code = E.code ‖ gen(id.place = E.place)
$E \rightarrow E_1 + E_2$	E.code = newtemp E.code = E_1.code ‖ E_2.code ‖ gen(E.place = E_1.place + E_2.place)

$E \rightarrow E_1 * E_2$ E.code = new temp
E.code = E_1.code || E_2.code || gen (Eplace = E_1.place * E_2.place.

$E \rightarrow -E_1$ E.place = new temp
E.code = E_1.code || gen (E.place = uminus E_1.place)

$E \rightarrow (E_1)$ E.place = E_1.place
E.code = E_1.code

$E \rightarrow id$ E.place = id.place
E.code = id.place

SDD for three Address code for Assignments

Translation of the boolean expression

* Boolean expression have two primary purposes.
* They are used to compute logical values, but also they are used as conditional expressions in statements that alter the folow of control such as if then, if then else, while do statments.
* Boolean expressions are composed as the Boolean operators like, and, or, not the Boolean variables.

production	semantic Rules
$E \rightarrow E_1$ or E_2	{E.place = new temp; emit(Eplace = E_1.place 'or' E_2.place)}
$E \rightarrow E_1$ and E_2	{E.place = new temp; emit(E.place = E_1.place 'and' E_2.place)}
$E \rightarrow$ not E_1	{E.place = new temp; emit(E.place = 'not' E_1.place)}
$E \rightarrow (E_1)$	{E.place = E_1.place}
$E \rightarrow id_1$ relop id_2 ↓ relational operator.	{E.place = new temp; emit (if id_1.place relop id_2.place goto nextstat+3) emit (E.place = '0'); emit (goto nextstat+2). emit (E.place = '1')}

E → True { E.place = new temp;
emit (E.place = 1) }

E → false { E.place = new temp;
emit (E.place = 0) }

A translation scheme for producing 3 address for code Boolean expressions statements.

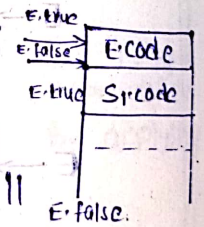
we assume that emit places 3 address the statements into an output file. in the right format; next stack gives the index of the next 3 address in the output sequence.

— and the emit increments the next stack after producing each 3-address statement.

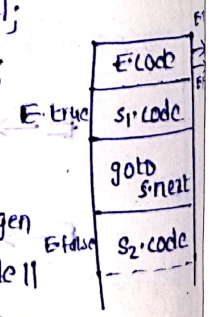
* Control flow :-

* some programming language uses the control flow like if, if-else and while statements. the intermediate code generator produces syntax direct definitions in the form of 3-address code for the control flow statements.

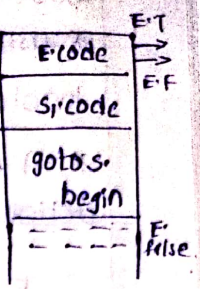
production	semantic Rules
$S \rightarrow$ If E then S_1	{ E true = new label E.false = s.next; S_1.next = s.next; S.code = E.code || gen (E.true ':') || S_1.code }



$S \rightarrow$ If E then S_1 else S_2 { E.true = new label;
E.false = new label;
S_1.next = s.next;
S_2.next = s.next;
S.code = E.code || gen (E.true ':') || S_1.code ||
gen (goto s.next) || gen
(E.false ':') || S_2.code }



S → while E do S | {S.begin = newlabel;
E.true = newlabel;
E.false = new S.next;
S1.next = S.begin;
S.code = gen (S.begin':')||
E.code || gen (E.true':') ||
S1.code || gen (goto S.begin)}



Back patching :-

* Back patching is the process of fulfilling unspecified information "incomplete transformations and information. It is basically uses the appropriate semantic actions during the process of code generation.
* It may indicate the address of the label in goto statement while producing the 3-address code for given expression.
* This is mainly used for 2 purposes

1) Boolean expressions :- These are the statements whose results can be either true or false.

2) Flow of control statements :- It needs to be controlled during the execution of the statements in a program.

* The most elementary programming language for changing the flow of control is label and goto statements
* if the label has already appeared then the symbol table will have a entry for 3-address instruction associated with the label 'L'. we use 3 functions to modify the list of pumps.

1) Makelist (i) :- create a new list containing only 'i' which is the index of quadraples.

2) Merge (P_1, P_2) :- concatenated the lists pointed by P_1 and P_2 and returns a point.

3) Backpatching (P_i) :- inserts i as a target label for each of the instructions pointed by P.

* Backpatching for a boolean expression :-

* using a translation scheme we can create a code for boolean expressions during bottom-up parsing

* A non-teminal M is a marker creates a semantic actions that mix-up the index of the next instruction.

$E \rightarrow E_1$ or E_2 backpatch (E_1.falselist, M.quad)

E.trulist = merge (E_1.truelist, E_2.truelist)

E.falselist = E_2.falselist

In the above example, the OR expression says that E_1 is false it should jump to the M-quad. this is how an OR expression logically works. The Truelist are merged if any of the arguments to the expression is true we can jump the OR expression.

* Boolean expressions are the conditional statements that change the flow of statements.

* Control statements are those that alter the order of statements executed.

* Applications of Backpatching :-

1) It is used to translate flow of control statements in a single pass

2) It is used for producing quadruples for boolean expressions

3) It is the activity of filling up information of the labels

* Intermediate code for procedure :-

* The procedures are more important and frequently used in the programming.

* The translation of call includes the calling sequence, a sequence of actions takes an entry and exit for each procedure.

* The state of the calling procedure must be same so it can resume the execution after the call. finally a jump to the begining of code for the called

procedure must be generated.

example: 3-address code.

n = f(a[i])

three address code statements

$t_1 = i * 4$

$t_2 = a[t_1]$

param t_2

t_3 = call f, 1 ← number of parameter.

$n = t_3$