Syntax directed translation are augmented rules to the grammar that facilitate the semantic analysis.

SDT = Grammar + Semantic rules

SDT is a method used to practically implement a compiler where semantic rules are associated along with the grammar.

Generalization of CFG where each grammar symbol has an associated set of attributes.
Value of attributes are computed by semantic rules

- Two notations for associating semantic rules with the productions
  - → SDD (Syntax Directed Definition)
  - → SDT schemes (Syntax Directed Translation schemes).

Semantic rules and attributes:

Semantic rules specify how the grammar should be interpreted

Example:-

$E \rightarrow E+T \quad \{ E.val = E.val + T.val \}$

Here, the values of non-terminals T and E are added together and the result is passed to the non-terminal E.

Attributes:

Attributes are the variables to which the values are assigned Every attribute variable is related with one or greater number of non-terminal or terminal of the grammar For example. $E \rightarrow E+T \{ E.val = E.val + T.val \}$

Here attribute 'val' i.e value is related to non-terminal E and T.

SDD: Syntax Directed Definition:

A SDD is a CFG with attributes and rules. Grammar symbols are associated with attributes and productions are associated with rules.

Example $\underbrace{E_1 \rightarrow E_2 + T}_{\text{production}}$ $\underbrace{\{ E_1.code \quad E_2.code \,||\, T.code \,||\, '+' \}}_{\text{semantic rule}}$

Syntax Directed Definition:

- → pure high level specification
- → hides implementation details
- → explicit order of evaluation is not specified.
- → easy to read.

Syntax directed translation schemes:

SDT schemes embeds program fragments (also called semantic actions) within the production bodies.

The position of the action defines the order in which the action is executed. (in the middle or end).

Example $E \rightarrow E + T \; \{ \text{print } '+' \}$

in this rule action is written at the end of production.

SDT schemes:

- indicate order in which semantic rules are to be evaluated
- allow some implementation details to be shown.
- More efficient.
- → easy to implement.

Example. infix to postfix

SDT Schemes	SDD
$E \to E+T$ {print '+'}	$E \to E+T$ ~~{E.code = E.code + T.code}~~ E.code = E.code || T.code || '+'
$E \to T$ { }	E.code = T.code
$T \to T*F$ { print '*' }	T.code = T.code || F.code || '*'
$T \to F$ { }	T.code = F.code
$F \to id$ { print id.lexval }	F.code = id.lexval

Evaluation of an expression using SDT:

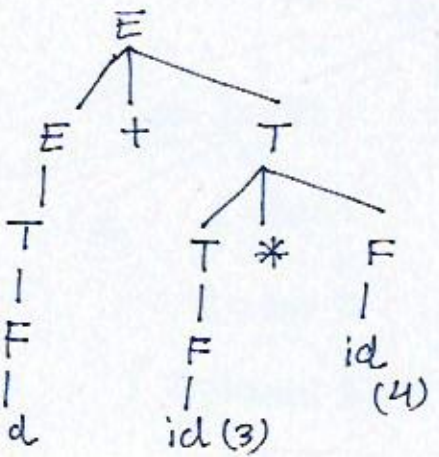
Example How to evaluate an expression '2+3*4'

→ write grammar and semantic rule to generate as well evaluate this expression.

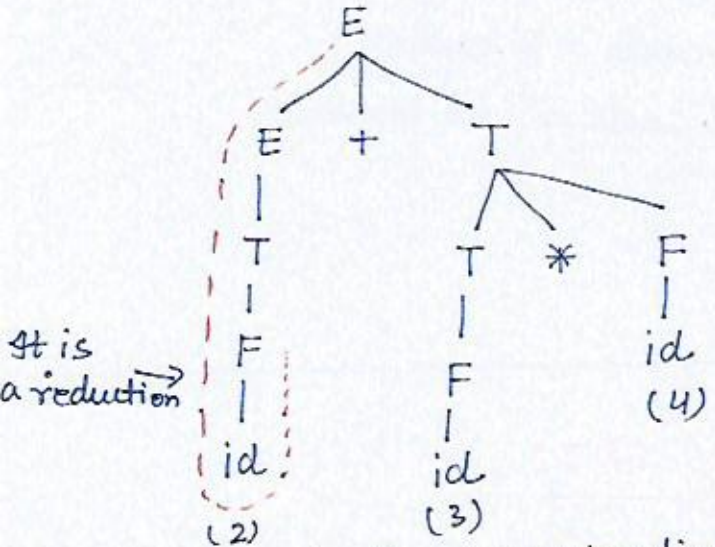
productions	Semantic rules
$E \to E+T$	E.val = E.val + T.val
$E \to T$	E.val = T.val
$T \to T*F$	T.val = T.val * F.val
$T \to F$	T.val = F.val
$F \to id$.	F.val = id.lexval

Expression: 2+3*4

Parse tree:



After generating this parse tree, we are going to traverse it from left to right and whenever there is a reduction, we go to the production and carryout the corresponding action.



$F \rightarrow id$ and the corresponding action to this production is

$F.val = id.lexval$

$F.val = 2$

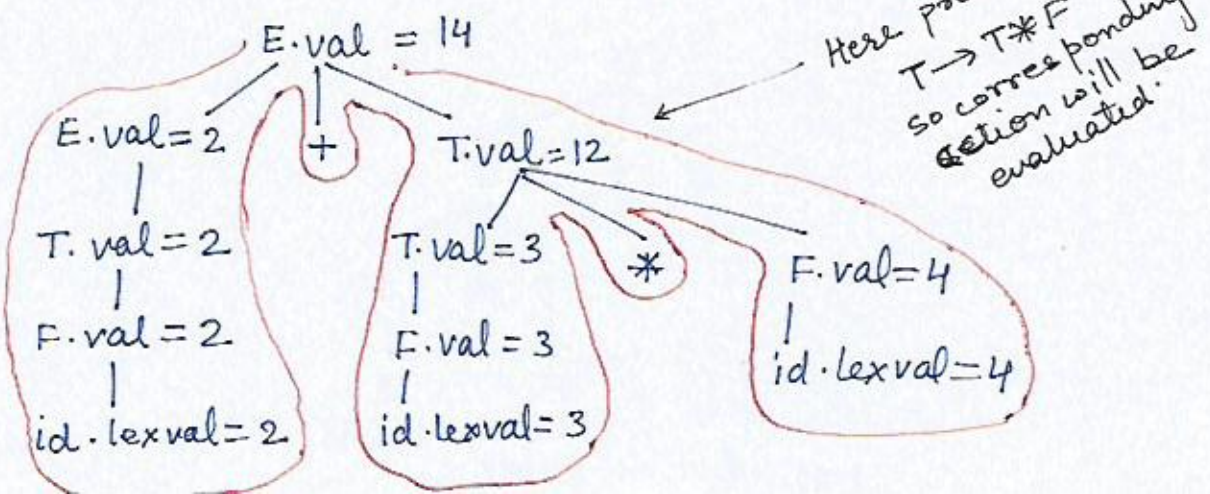
lexval ⇒ which is provided by the lexical analyzer.

After this, there is again a reduction i.e $T \rightarrow F$ (F is reduced to T)

Then action is taken $T.val = F.val$

$T.val = 2$

similarly at final step we have,



Annotated Parse tree:

A parse tree showing the values of attributes at each node is called an annotated parse tree.

The process of computing the attribute values at the nodes is called annotating or decorating the parse tree.

Postfix conversion using SDT:

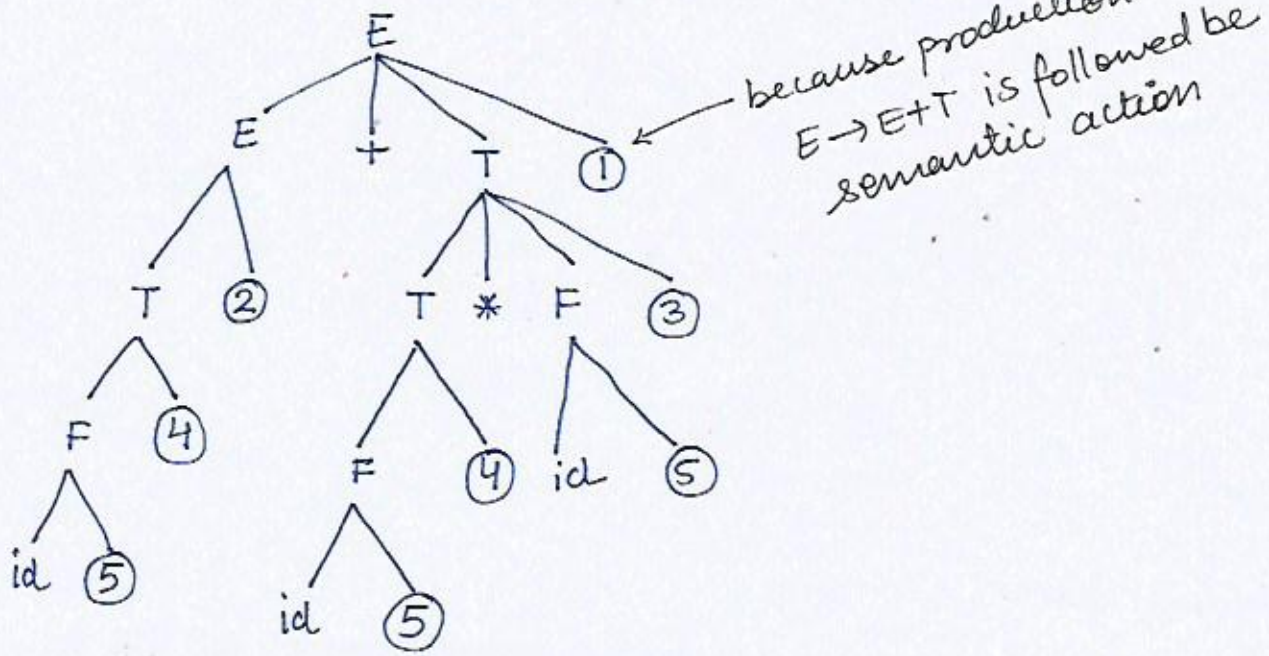
SDT scheme for converting the expression into postfix:

$E \rightarrow E+T$ { print '+' } —— (1)
$E \rightarrow T$ { } —— (2)
$T \rightarrow T*F$ { print '*' } —— (3)
$T \rightarrow F$ { } —— (4)
$F \rightarrow id$ { print id.lexval } —— (5)

⇒ Number the semantic actions.

Now suppose $E \rightarrow E+T$ { print '+' } is complete production.

Now make parse tree with actions for expression 2+3*4



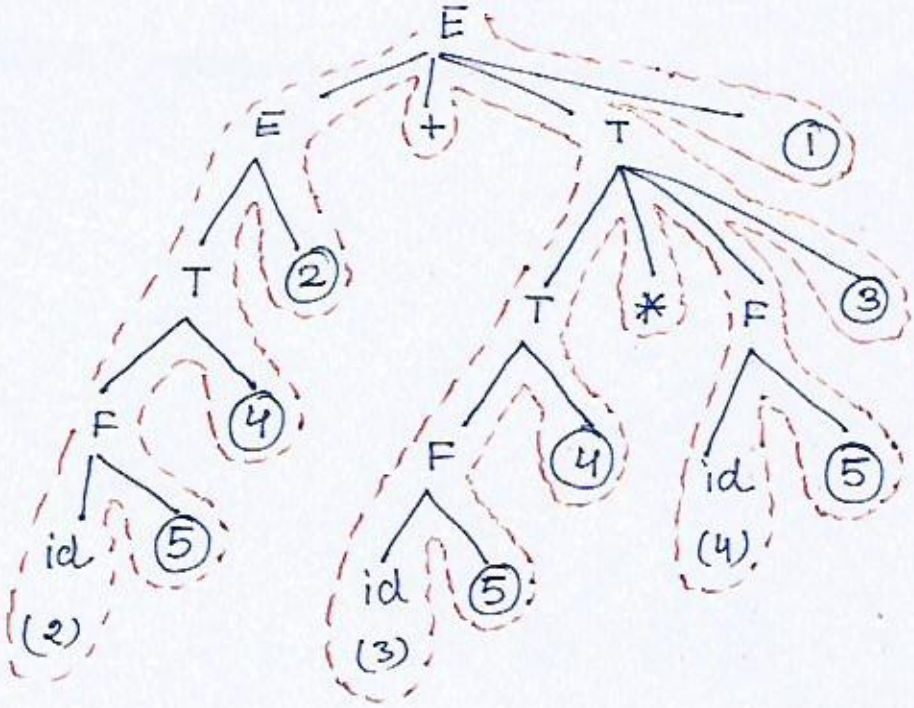
Now how does the top down parser work for this.

we are going to traverse the parse tree top down left to right and wherever we come across a semantic action we need to perform that semantic action.

So when we traverse the parse tree top down and left to right then the sequence of actions will be.

→ action ⑤	print 2
→ then action ④	Do nothing
→ then action ②	Do nothing
→ then action ⑤	print 3
→ then action ④	Do nothing
→ then action ⑤	print 4
→ then action ③	print *
→ then action ①	print +

Output: 2 3 4 * +



Finally we have

expression 2+3*4 postfix expression = 234*+

Types of attributes:

attributes fall into two classes:

→ Synthesized attributes

→ Inherited attributes

→ the value of synthesized attribute is computed from the values of its children nodes.

→ the value of inherited attributes is computed from the sibling and parent nodes.

Each grammar production $A \rightarrow a$ has associated with it a set of semantic rules of the form:

$$b = f(c_1, c_2 \text{---} c_k)$$

where f is a function and either

- b is a synthesized attribute of A

OR

- b is an inherited attribute of one of the grammer symbol on the right.

Example

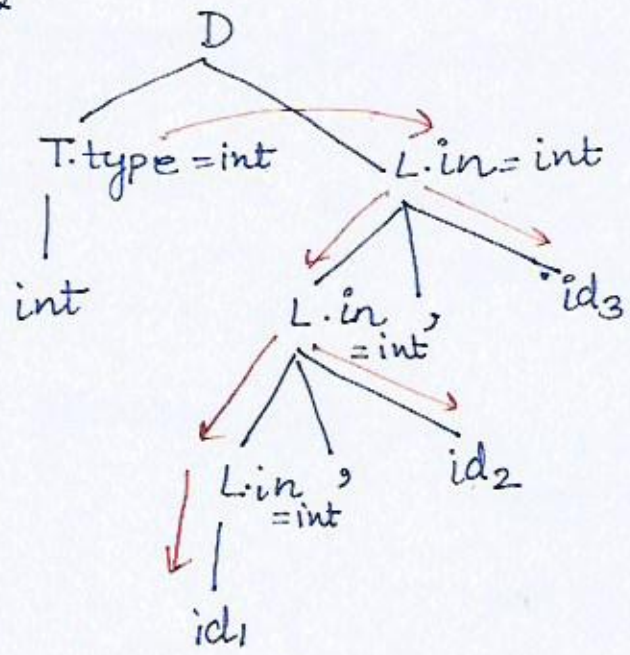
Productions	Semantic rule
$D \rightarrow TL$	L.in = T.type
$T \rightarrow$ int	T.type = integer
$T \rightarrow$ float	T.type = float
$L \rightarrow L_1$, id	L_1.in = L.in addtype(id.entry, L.in)
$L \rightarrow$ id	addtype(id.entry, L.in)

Here addtype is a procedure to add the type of each identifier to its entry in the symbol table (pointed to by the attribute entry).

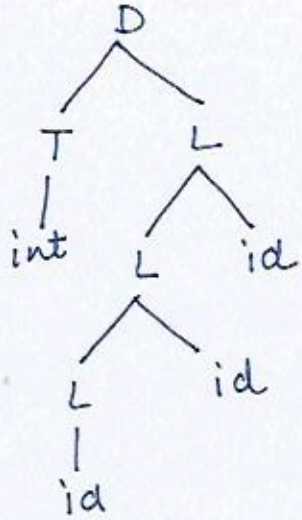
Here inherited attribute : in
synthesized attributes : type, entry.

Expression int id_1, id_2, id_3.

Annotated Parse tree ⟹



Parse tree



$D \rightarrow T\ L$

This L depends on T for type information
So L.in is an inherited attribute

$E \rightarrow E+T$ { $E.nptr = mknode(E_1.nptr, '+', T.nptr)$ }

$E \rightarrow T$ { $E.nptr = T.nptr$ }

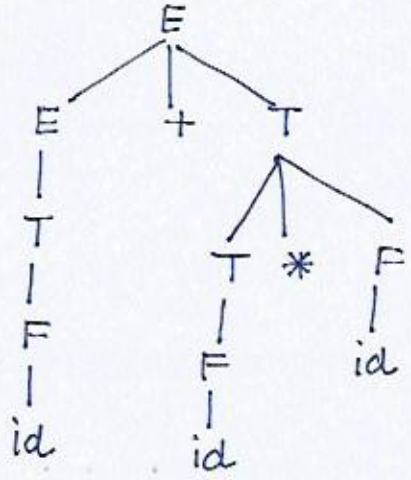
$T \rightarrow T*F$ { $T.nptr = mknode(T_1.nptr, '*', F.nptr)$ }

$T \rightarrow F$ { $T.nptr = F.nptr$ }

$F \rightarrow id$ { $F.nptr = mknode(null, id.name, null)$ }

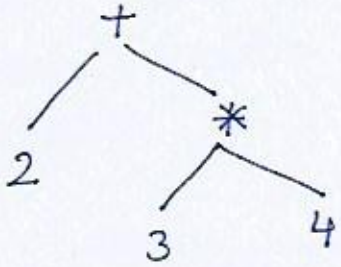
Expression:- 2+3*4

Parse Tree



This parse tree is known as concrete parse tree because it represents more details

Syntax tree



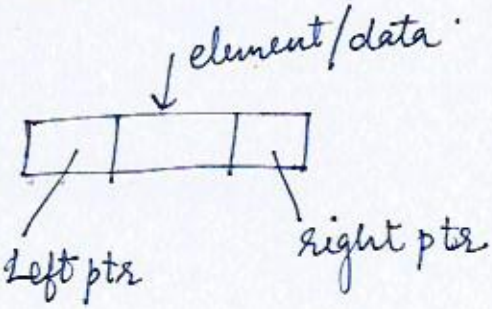
This tree is known as abstract tree because detail is not given here

Above SDT is given to create abstract parse tree. Here.

nptr = node pointer

mknode = mknode is a function that is used to make a node and it has 3 values

1. Left pointer
2. Id Value
3. Right pointer



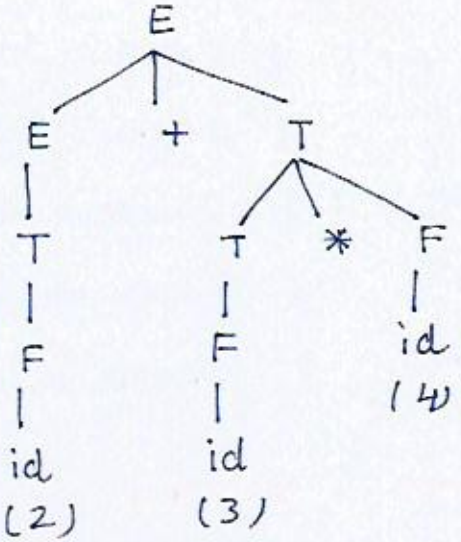
step ① This is a reduction $F \rightarrow id$

semantic action:

mknode(null, id.name, null)

| null | 2 | null |

suppose address of this node = 100



Step ②

Reduction $T \rightarrow F$

action $T.nptr = F.nptr$

$T.nptr = 100$

step ③ Reduction $E \rightarrow T$

action $E.nptr = T.nptr$

$E.nptr = 100$

step ④ Then again Reduction $F \rightarrow id$

action $F.nptr =$ mknode(null, id.name, null)

| null | 3 | null | 200 |

step ⑤ Reduction $T \rightarrow F$

$T.nptr = 200$

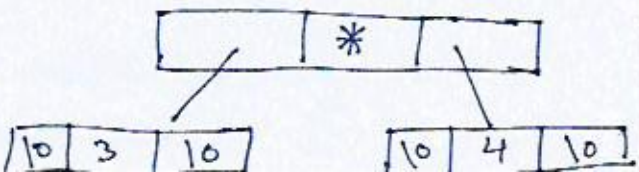
step ⑥ Reduction $F \rightarrow id$

| null | 4 | null | 300 |

step ⑦ Reduction $T \rightarrow T*F$

action $T.nptr =$ mknode(T.nptr, *, F.nptr)

\0 ⇒ means null



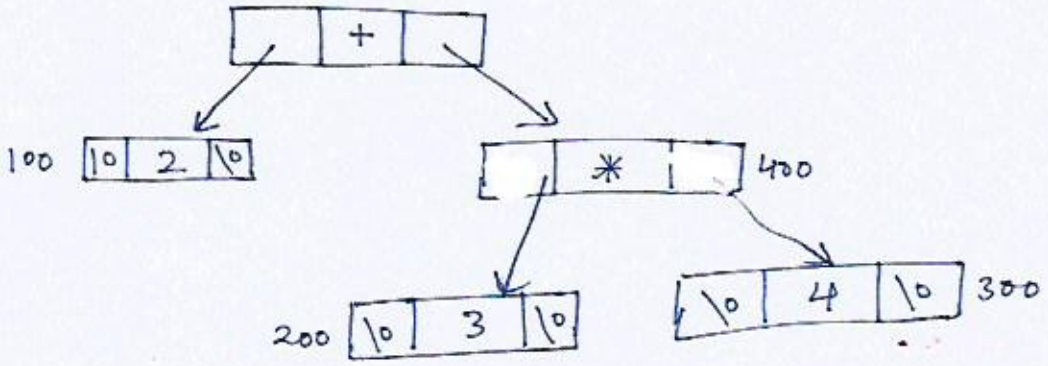
step (8) Now suppose address of new created node is 400 then

$T.nptr = 400$

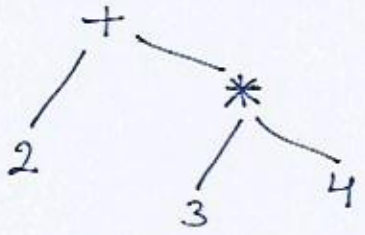
Now $E \rightarrow E+T$ (reduction)

Action : { E.nptr = mknode(E_1.nptr, '+', T.nptr) }

Thus



This syntax tree is similar as



synthesized Attribute

Production $A \rightarrow BCD$

$A.S = f(B.S, C.S, D.S)$

Here A is taking value from its children. Such an attribute is known as synthesized attribute

Inherited Attributes

Production $A \rightarrow BCD$

if C is taking value from its parent i.e A and sibling like B & D. Then such an attribute is known as inherited attribute.

In $A \rightarrow BCD$

$C.i = A.i$ { inherited }
$C.i = B.i$ { inherited }
$C.i = D.i$ { inherited }

Classification of SDT:

→ S- attributed SDT
→ L- attributed SDT

S- attributed SDT

① Uses only synthesized attribute

② Semantic actions are placed at right end of production

$A \rightarrow BCD$ { }

③ Attributes are evaluated during Bottom up parsing

L- attributed SDT

① Uses both inherited and synthesized attributes. Each inherited attribute is restricted to inherit either from parent or left sibling only.

$A \rightarrow xyz$

$y.s = A.s$
$y.s = x.s$

② Semantic actions are placed anywhere on R.H.S

③ Attributes are evaluated by traversing parse tree depth first left to right.

Dependency graph:

If an attribute b at a node in parse tree depends on an attribute, then this dependence can be shown as

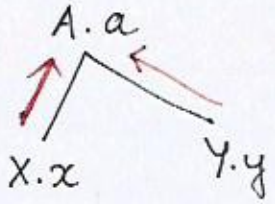
$$c \longrightarrow b$$

i.e the semantic rule for b at node must be evaluated after the semantic rule that defines c.

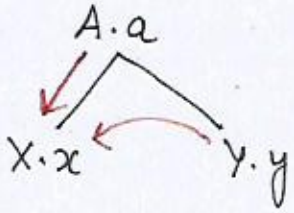
→ The interdependence among attributes can be depicted by a graph called dependency graph.

→ if there is a production $A \longrightarrow XY$

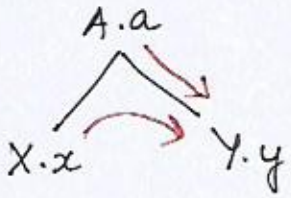
① if $A.a = f(X.x, Y.y)$ then



② if $X.x = f(A.a, Y.y)$ then ⟹



③ if $Y.y = f(A.a, X.x)$ then



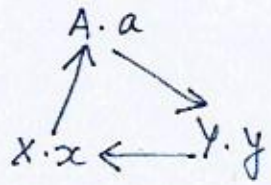
→ (arrow) show the value dependence., i.e the dependence of one attribute on other.

So it is called dependence graph. Edges in the graph shows the evaluation order of attribute value.

Dependency graph with cycle.

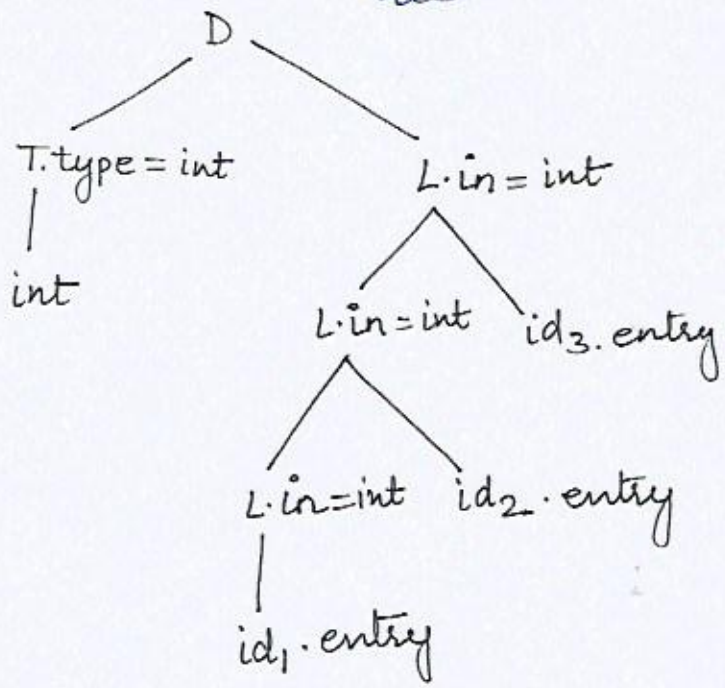
→ it is an error because we can not evaluate any particular attribute

$$A.a = f(X.x)$$
$$X.x = f(Y.y)$$
$$Y.y = f(A.a)$$

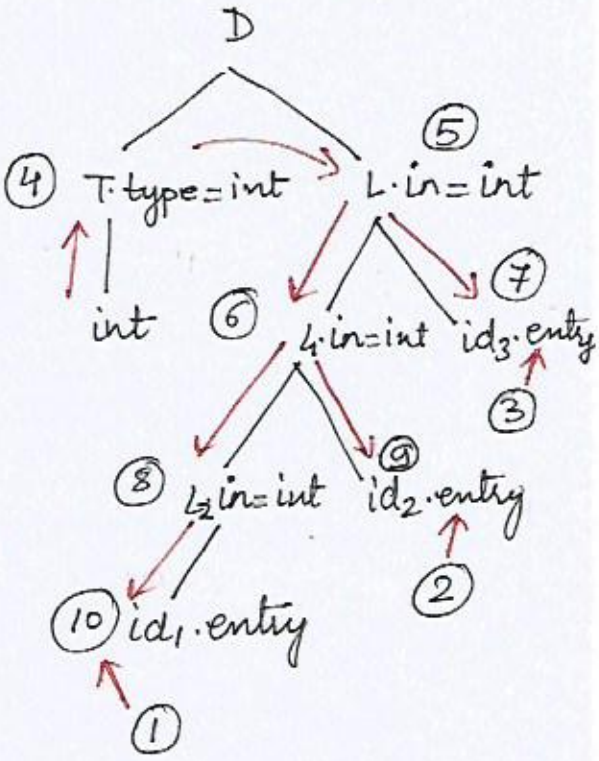


Example:

Annotated parse tree



Annotated parsetree with Dependency graph.



After constructing dependency graph what is the order of evaluation

→ Topological Sort.

A topological sort of a directed graph (directed acyclic graph) is any ordering $m_1, m_2 \cdots m_n$ of the nodes of the graph such that if $m_i \rightarrow m_j$ is an edge then m_i appears before m_j.

A topological sort is a sorting algorithm where sorting is not done on the bases of decreasing or increasing order of input rather it depends upon which node should occur which node.

Annotated graph with dependence graph: numbering as-

(1) Make entry id_1 into symbol table, get id_1.entry
(2) Make entry id_2 into symbol table, get id_2.entry.
(3) Make entry id_3 into symbol table, get id_3 entry.
(4) T.type = int
(5) L.in = T.type
(6) L_1.in = L.in
(7) addtype (id_3.entry, L_1.in)
(8) L_2.in = L_1.in
(9) addtype (id_2.entry, L_2.in)
(10) addtype (id_1.entry, L_2.in)

INTERMEDIATE CODE

In many compilers the source code is translated into a language which is intermediate in complexity between a (high level) programming language and machine code.

Such a language is therefore called intermediate code or intermediate text.

Intermediate codes are considered as machine instruction and these are machine independent.

Intermediate code can be written in different languages, and the designer, who designs the compiler decides this intermediate language.

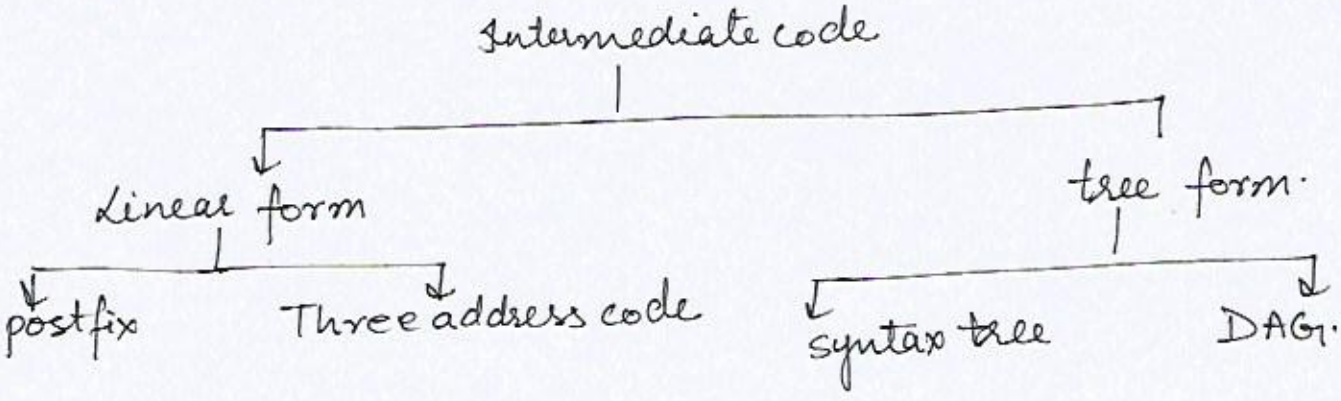
Benifits of using a machine independent intermediate form:

- → Retargeting is possible.
- → Machine independent code optimization is possible.

Intermediate code Representation:

There are three ways to represent intermediate code.

- → Abstract syntax tree
- → postfix notation
- → three address code.



(1) Postfix Notation:

→ Linearized representation of syntax tree

→ Postfix notation is used to represent intermediate code in form of expressions.

→ A postfix notation is also known as Reverse Polish Notation

→ In postfix notation operator is placed after operands instead of between them

→ There are two reasons for using postfix notation
  → There is only one interpretation
  → no need of parenthesis

Example: $(p+q) * (r+s)$ infix notation

$\Rightarrow (pq+) * (r+s)$

$\Rightarrow (pq+) * (rs+)$

$\Rightarrow pq+\ rs+\ *$ (postfix notation)

Postfix notation is used in stack based machine.

Example

infix	postfix
$a = b* -c + b* -c$	$abc- * bc- * + =$

Three address code:

Three address code is something which is close to machine architecture.

- Three address code is most popular in all 4 intermediate representation.
- It is a sequence of statements of the general form

  $X = Y \text{ op } Z$ where X, Y, or Z are names, constants, or compiler generated temporaries.

- In three address code, there can be at most one operator and two operands on the right side of an instruction.
- 'at most' means we can have also fewer.
- Source expression like $x + y * z$ might be translated into three address code

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

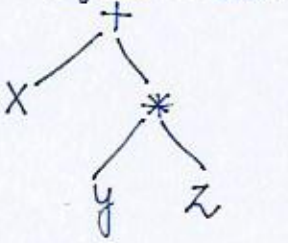
  where t_1 and t_2 are compiler generated temporaries names.

Ex ⇒ $x + y * z$

Three address code

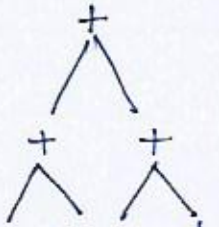
$t_1 = y * z$
$t_2 = x + t_1$

Syntax tree



$a + b + c + d$

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



Three address code is a linearized representation of a syntax tree where explicit names corresponds to the internal nodes of graph.

Three address code instructions

Assignment: we can have set of assignments, assigments can look something like:

$x = y \text{ op } z$ (in case of binary operator)

$x = \text{op } z$ (unary operator)

$x = y$ (straight assignment) or copy statement

Jump:

- goto L (unconditional goto) Here L is label (address)
- if x relop y goto L (conditional jump)

Indexed assignment

$x = Y[i]$

$x[i] = Y$

Function: Argument passing

- param x
- call p, n
- return y

pointers:

$x = \&y$

$x = *y$

$*x = y$

you can see that with this kind of intermediate representation we can generate code form intermediate code form for a large class of language.

Three Address Code representation methods:

Three address code can be represented in various forms:

1. Quadruples
2. Triples
3. Indirect triples

Quadruples: →

we may use a record structure with four fields which we call op, op1, op2 result.

op ⇒ operator
op1 → operand 1 or Arg1 (argument 1)
op2 → operand 2 or Arg2 (argument 2)

Example expression: $-B * (C+D)$

Three address code:

$T_1 = -B$
$T_2 = C+D$
$T_3 = T_1 * T_2$
$A = T_3$

These statements are represented by quadruples like:

	op	op1	op2	Result
(0)	uminus	B	—	T_1
(1)	+	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	=	T_3	—	A

contents of fields op1, op2, Result are normally pointer to symbol ~~table~~ entries

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

Three address code

$t_1 = a+b$
$t_2 = -t_1$
$t_3 = c+d$
$t_4 = t_2 * t_3$
$t_5 = a+b$
$t_6 = t_5+c$
$t_7 = t_4+t_6$

Quadruples

	op	op1	op2	Result
(0)	+	a	b	t_1
(1)	−	t_1		t_2
(2)	+	c	d	t_3
(3)	*	t_2	t_3	t_4
(4)	+	a	b	t_5
(5)	+	t_5	c	t_6
(6)	+	t_4	t_6	t_7

Triples :

→ Avoids entering temporary names into the symbol table
→ allows three address statements to be representable with only three fields op, op1, op2.
→ Since three fields are used, this intermediate format is known as triples.
→ we use parenthesized numbers to represent pointers in to the triple structure.

Example $A = -B * (C+D)$

Triples

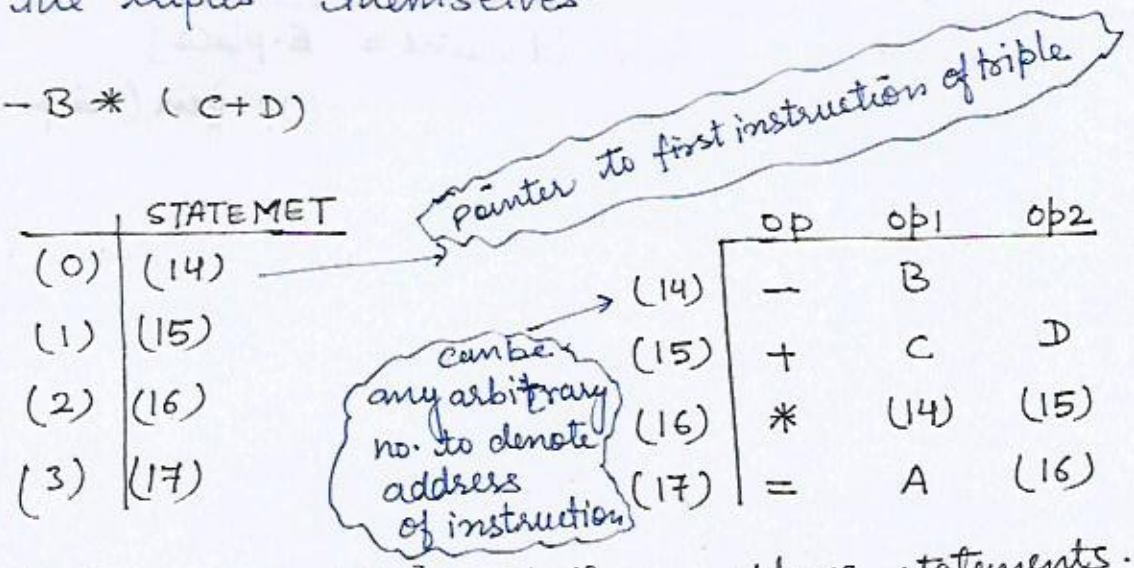
	op	op1	op2
(0)	−	B	
(1)	+	C	D
(2)	*	(0)	(1)
(3)	=	A	(2)

This (1) is representable result of statement (1)

Indirect triples :-

This method includes listing pointers to the triples rather than listing the triples themselves.

Example: $A = -B * (C+D)$



Indirect triples representation of three address statements.

For indirect triples, instructions are stored on memory location and then for particular instruction we are using pointer to that location on which that instruction is stored.

Advantages and disadvantages :

Quadruples :

Advantages :- statements can be moved.
Disadvantage- Too much space is wasted.

Triples :

Advantage :- space is not wasted because result is not being saved
Disadvantage- statements can not be moved because if we change the order then result of instruction will be changed.

Indirect triples :

Advantage :- statements can be moved.
Disadvantage : Requires two memory access
- One to get address of the instruction/statement
- Second to get the statement itself.

TRANSLATION OF ASSIGNMENT STATEMENTS:

Grammar:	Translation scheme.
$S \rightarrow id = E$	{ gen(id.name = E.place }
$E \rightarrow E + T$	{ E.place = newtemp(); gen(E.place = E_1.place + T.place) }
$E \rightarrow T$	{ E.place = T.place }
$T \rightarrow T * F$	{ T.place = newtemp(); gen(T.place = T_1.place * F.place)
$T \rightarrow F$	{ T.place = F.place }
$F \rightarrow id$	{ F.place = id.name }

This translation scheme translate assignment statements into intermediate code.

Here.

newtemp() ⇒ function is used to create new temporary names.

gen( ) ⇒ to produce the three address statement.

place ⇒ attribute is the name that holds the value of expression

Example expression: $x = a + b * c$

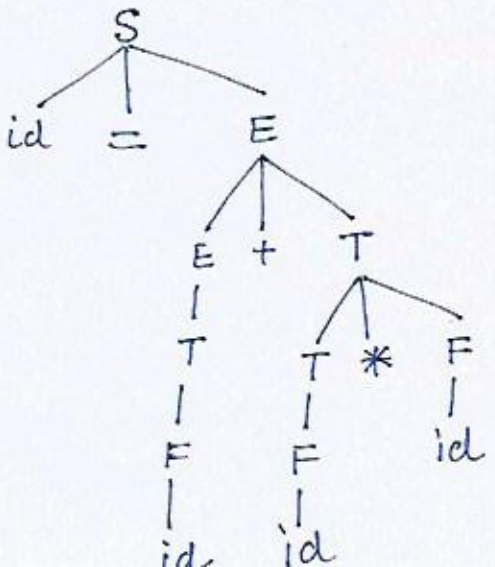
Three address code:

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

First construct a parse tree.



step ① $F \rightarrow id$

$F.place = id.name = a$

step ② $T \rightarrow F$

$T.place = F.place$

$T.place = a$

step ③ $E \rightarrow T$

$E.place = T.place = a$

step ④ $F \rightarrow id$

$F.place = id.name = b$

step ⑤ $T \rightarrow F$

$T.place = F.place = b$

step ⑥ $F \rightarrow id$

$F.place = id.name = c$

step ⑦ $T \rightarrow T * F$

$T.place = newtemp() = t_1$ (suppose)

$gen(T.place = T_1.place * F.place)$

$t_1 = b * c$

step ⑧ $E \rightarrow E + T$

$E.place = newtemp() = t_2$

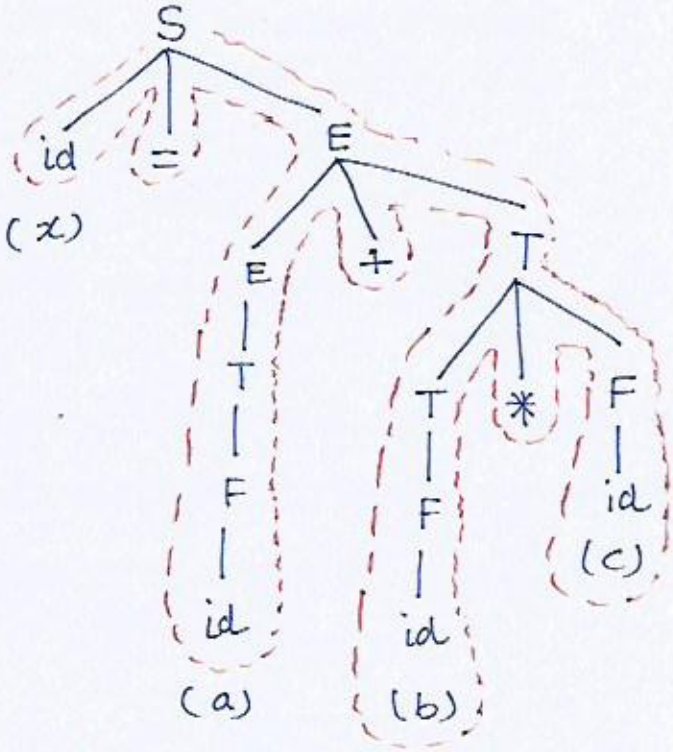
$gen(E.place = E_1.place + T.place)$

$t_2 = a + t_1$

step ⑨ $S \rightarrow id = E$

$gen(id.name = E.place)$

$x = t_2$



Generated three address code:

$t_1 = b * c$

$t_2 = a + t_1$

$x = t_2$

BOOLEAN EXPRESSIONS:

(1) Boolean expressions are used as conditional expressions in statements that alter the flow of control such as
   - → if-else statements
   - → while statements etc.

(2) Boolean expressions are also used to compute logical values.

→ Boolean operators are and, not, or.

→ Boolean operators can also be combined with relational operators

$<, >, <=, >=, ==, !=$.

Translation scheme for producing three address code for boolean expression.

Production	
$E \rightarrow E$ OR E	E.place = newtemp(); gen(E.place = E1.place 'or' E2.place)
$E \rightarrow E$ and E	E.place = newtemp() gen(E.place = E1.place 'and' E2.place)
$E \rightarrow$ not E_1	E.place = newtmp() gen(E.place = 'not' E1.place)
$E \rightarrow (E1)$	E.place = E1.place
$E \rightarrow$ id1 relop id2	E.place = newtemp() gen(if id1.place relop id2.place, goto nextstat+3) gen (E.place = 0) ← denotes 'false' gen (goto nextstat + 2) gen (E.place = 1) ← denotes 'true'
$E \rightarrow$ true	E.place = newtemp() gen. (E.place = 1)
$E \rightarrow$ false	E.place = newtemp(); gen(E.place = 0)

Example:

a<b or c<d and e<f

Generate three address code for this boolean expression.

solution

100: if a < b goto 103
101: $t_1 = 0$
102: goto 104
103: $t_1 = 1$
104: if c<d goto 107
105: $t_2 = 0$
106: goto 108
107: $t_2 = 1$
108: if e<f goto 111
109: $t_3 = 0$
110: goto 112
111: $t_3 = 1$
112: $t_4 = t_2$ and t_3
113: $t_5 = t_1$ OR t_4

This method is known as numerical representation method where true is denoted by value 1 and false is denoted by 0.

Ques: generate three address code for the following statements

① P<Q and R<S OR T<U

② P<Q OR R<S and T<U

Statements that alter the flow of control:

→ if-else statements
→ while statements

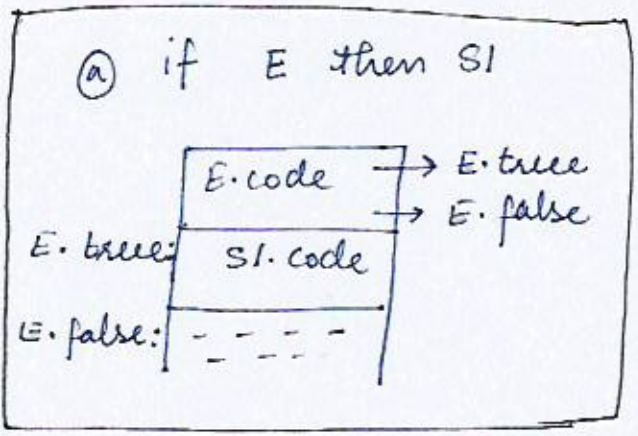
→ Boolean expressions are typically used along with if-then, if-then-else, while-do statements.

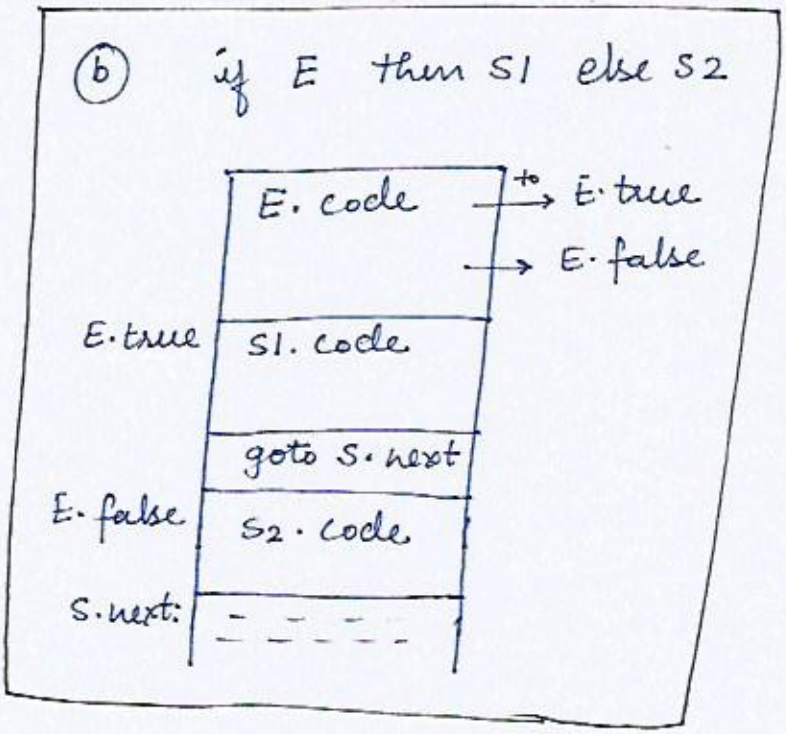
→ Grammar: S → if E then S1 | if E then S1 else S2 |
while E do S1 | do S1 while E

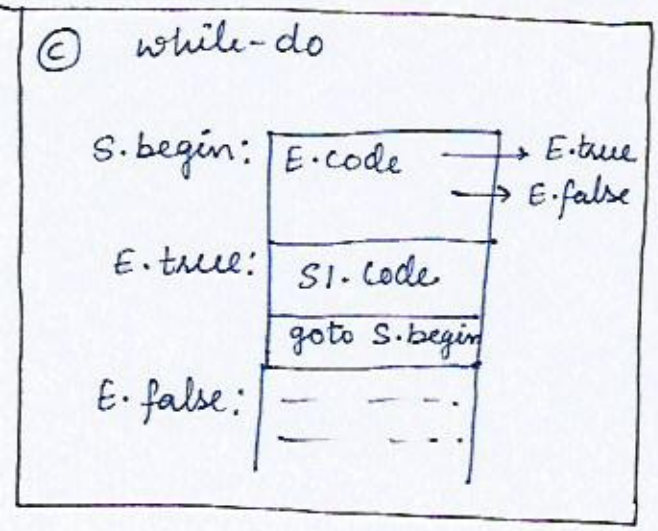
All these statements 'E' corresponds to a boolean expression evaluation

→ This expression E should be converted to three address code as already discussed in 'translation of boolean expressions'

→ With a boolean expression E, we associate two labels:
E.true ⇒ the label to which control flows if E is true
E.false ⇒ the label to which control flows if E is false.







SDT for flow of control statements:

Production	Semantic rules
$S \rightarrow$ if E then S1	E.true = newlabel E.false = S.next S_1.next = S.next S.code = E.code || gen(E.true:) || S_1.code
$S \rightarrow$ if E then S1 else S2	E.true = newlabel E.false = newlabel 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
$S \rightarrow$ while E do S1	S.begin := newlabel E.true = newlabel E.false = S.next S_1.next = S.begin S.code = gen(S.begin':') || E.code || gen(E.true':') || S_1.code || gen('goto' S.begin)

Ques: if a<b
x = y+z
else
p = q+r

Threeaddress code:

100: if a<b goto L1
101: goto L2

102: L1: t_1 = y+z
103: x = t_1
104: goto L3:
105: L2: t_2 = q+r
106: p = t_2
107: goto L3
108: L3:

Control flow translation of boolean expressions:

→ Label method

→ Example:

a<b OR c<d AND e<f

① if a<b goto LTrue
② goto L2
③ L1: if c<d goto L2
④ goto Lfalse ⟹ for sure it will execute false cond.
⑤ L2: if e<f goto LTrue
⑥ goto Lfalse
⑦ LTrue:
⑧ LFalse:

Ques: write three address code for the following statements

```
while (i<10)
{
  x=0
  i=i+1;
}
```

① LBegin: if (i<10) goto LTrue
② goto Lnext
③ LTrue: x=0
④ $t_1 = i+1$
⑤ $i = t_1$
⑥ goto LBegin
⑦ Lnext:

Case statements translation:

Syntax

```
switch E
{
 case V1 : S1
 case V2 : S2
 case V3 : S3
      :
 case Vn : Sn
 default : Sn+1
}
```

SDT for case statement translation

```
    code to evaluate condition E into T
    goto Test
L1 :  S1.code
      goto Snext
L2 :  S2.code
      goto Snext
      :
Ln :  Sn.code
      goto Snext
Ln+1: Sn+1.code goto Snext
Test: if T == V1 goto L1
      if T == V2 goto L2
         :
      if T == Vn goto Ln
         goto Ln+1
Snext:
```

Example

```
switch (ch)
{
 case 1:  c = a+b ; break;
 case 2:  c = a-b , break;
}
```

Three address code:

```
    T = ch
    goto Test
L1: t1 = a+b
    c = t1
    goto snext
L2: t2 = a-b
    c = t2
    goto snext
Test: if T == 1 goto L1
      if T == 2 goto L2
```

Procedure Call: code generation for procedure calls:

→ generate three address code needed to evaluate arguments which are expressions.

→ generate a list of param three address statements.

→ store arguments in a list.

SDT for a simple procedure call

S → call id (Elist) { for each item p on queue do
gen(~~pop~~ param p)
gen(call id.place) }

Elist → Elist, E { append E.place to the end of queue }

Elist → E { initialize queue to contain E.place }

Example ① call fun(a, b, c)

Three address code:

param a
param b
param c
call fun

Example ② call f(x+y, a+b, c)

Three address code

→ first evaluate arguments

$t_1 = x + y$
$t_2 = a + b$

→ generate list of param

param t_1
param t_2
param c

→ generate call.

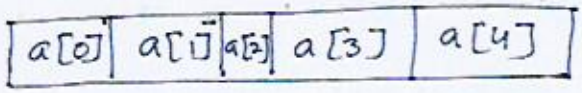
call f.

Addressing of Arrays:

Arrays: Array is a collection of elements.

→ Array has certain properties. one property is that all elements in array will be stored in contiguous memory location.

Example `int a[5]`



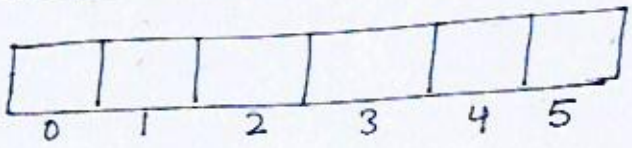
→ Because of contiguous memory, only thing we need to know the base address of array to access the elements.

→ Now if we want to compute address of i^{th} element of array a ⇒

address of a[i] = base + (i - low) × w

where low ⇒ lower bound of array.
w ⇒ width of each element (it is known from type info).
base ⇒ base address of array.

Example



address of A[4] = base + (i - low) × w where base = 100 (Suppose. w = 1 !.

= 100 + (4 - 0) × 1

= 104

---

expression ⇒ base + (i - low) × w

⇒ base + i × w - low × w

⇒ (base - low × w) + i × w

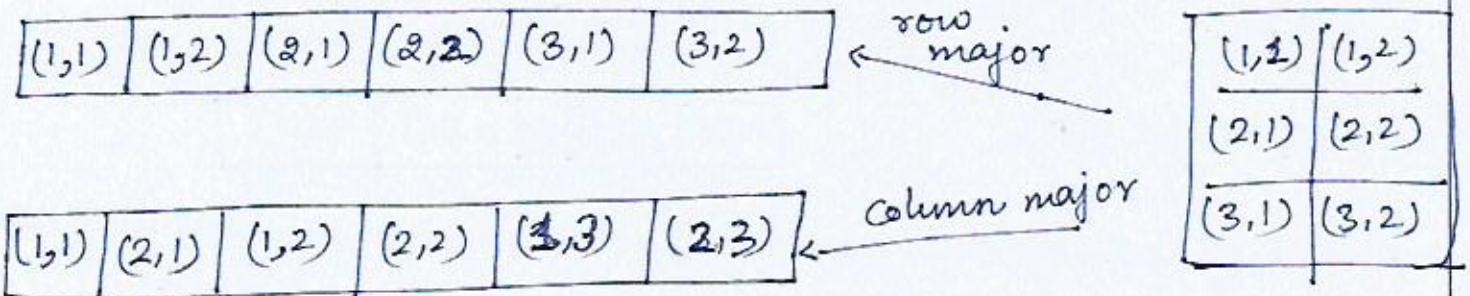
⇒ i × w + const

⇒ base - low × w is known at compile time and evaluated once, stored in symbol table.

2D - arrays:

Arrays has two types of representations.

1. row - representation
2. column - representation.

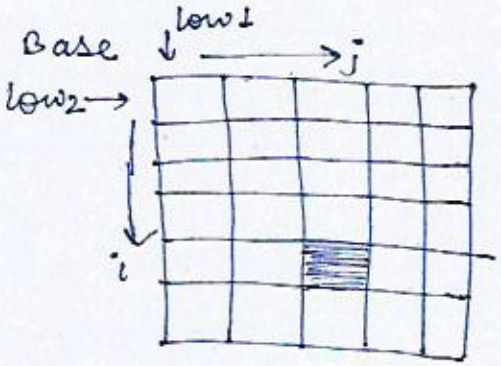


To find A[i,j] ⇒ i = row number
j = column number.

Row major representation:

$$\text{Address of } A[i,j] = \text{base} + ((\underbrace{(i - low_1) \times n_2}_{\text{no. of rows before element}} + \overbrace{j - low_2}^{\text{no. of cols before element}}) \times w$$

$n_2 \rightarrow$ no. of columns
$w \rightarrow$ width of each element
$low_1 \rightarrow$ low index of rows
$low_2 \rightarrow$ low index of columns



$$n_2 = high_2 - low_2 + 1.$$

Rearranged

$$\text{Address of } A[i,j] = (i \times n_2 + j) \times w + (\text{base} - ((low_1 * n_2) + low_2) \times w)$$

Example: Three address code generation for array.

Let A be a 10X20 array. Then write three address code to access A[i, j] and assume w=4.

A[10][20] Therefore: $n_1 = 10$ $w = 4$ $low_2 = 1$ (Assume)
$n_2 = 20$ $low_1 = 1$

⇒ we have

Address of A[i, j] ⇒ $(i \times n_2 + j) \times w + \underbrace{(base - ((low_1 \times n_2) + low_2) \times w)}_{constant}$

Three address code:

① $t_1 = i * n_2 = i \times 20$

② $t_2 = t_1 + j$

③ $t_3 = t_2 \times 4$

④ $t_4 =$ constant or C(A)
or A − 84

⑤ $t_5 = t_3 + t_4$
or
$t_4[t_3]$

* Array name is treated as base address.

$(low_1 \times n_2 + low_2) \times w$
⇒ $(1 \times 20 + 1) \times 4$
$= 21 \times 4$
$= 84$

→ $base - ((low_1 \times n_2 + low_2) \times w)$ ⇒ this value will be computed at compile time.

→ $((i \times n_2 + j) \times w)$ computed at run time.

Example find three address code of the following :-

```
for( i=0; i<10; i++)
{
  b[i] = i * i;
}
```

Three address code

```
(1)  i=0
(2)  if i<10 goto (4)
(3)  goto (11)
(4)  t1 = i*i
(5)  t2 = i*w
(6)  t3 = b+t2
(7)  *t3 = t1
(8)  t4 = i+1
(9)  i = t4
(10) goto (2)
(11)
```

Declarative statements :

Data items along with their data types.

SDT to translate declarative statements

(1) S → D { offset = 0 }

(2) D → id : T { enter_table (id.name, T.type, offset); offset = offset + T.width }

(3) T → integer { T.type = integer, T.width = ____ }

(4) T → real { T.type = real, T.width = ____ }

⑤ $T \rightarrow array[num]\ of\ T_1$ { T.type = array(num.val, T_1.type
T.width = num.val × T_1.width }

⑥ $T \rightarrow *T_1$ { T.type = pointer(T_1.type)
T.width = ____ }

→ In declaration statements the data items along with their data types are declared.

→ Type attribute ⇒ synthesized attribute. indicates the data ty

width " ⇒ " " indicates the memor
units occupied by an identifier.

→ enter-table function is used to make entry of identifier in symbol table.

→ width of an array ⇒ number of elements in array × widt
of each element in array.