Unstudied-

# Syntax Directed Translation

I

## A. Syntax- Directed Derivation (SDD)

A SDD is a kind of abstract specification. It is a combination of CFG and semantic rules.

SDD = CFG + Semantic rules.

Attributes are associated with grammar symbols and semantic rules are associated with productions.

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

Attributes may be number, string, references, datatypes etc.,

| Production | Semantic rule |
|---|---|
| $E \rightarrow E+T$ | { E.val = E.val + T.val } |
| $E \rightarrow T$ | { E.val = T.val } |

Semantic rules are fragments of code which are embedded usually at the end of production and enclosed in curly braces.

Types of attributes:

1. Synthesized Attributes: These are those attributes which derive their values from their children nodes ie., value of Synthesized attribute at node is computed from the values of attributes at children node.

Ex: $A \longrightarrow BCD$

$A$ — Parent; $BCD$ — children.

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

} Parent node of A taking value from its children B, C, D

2. Inherited Attribute: These are the attributes which derive their values from their parent (or) sibling nodes. ie., value of inherited attributes are computed by value of parent (or) sibling nodes.

Ex: $A \longrightarrow BCD$

$C.i = A.i$ (or) $C.i = B.i$ (or) $C.i = D.i$

Types of SDD:

1. S-Attributed SDD (or) S-Attributed Defination (or) S-Attribute grammar

1. It uses only Synthesized attributes is called S-Attributed SDD.
2. Semantic Actions are always placed at right end of the production.
3. Attributes are evaluated with Bottom up parsing.
4. It is also called "post-fix SDD"

L-Attributed SDD:

1. ASDD that uses both Synthesized (or) Inherited Attributes is called as L-Attributed SDD.
2. In L-Attributed SDD each Inherited attributes is restricted to inherit from parent (or) left sibling only.
3. Semantic Actions are placed anywhere (or) R.H.S
4. Attributes are evaluated by depth first Top-down parsing.

Ex: $A \rightarrow XYZ$

{ A.S  Y.S = A.S,  Y.S = X.S,  Y.Z = Z.S }

## B. Evaluation Orders of SDD :—

### Dependency Graph :

A Dependency graph provides information about the order of evaluation of attributes with the help of edges. It is used to determine the order of evaluation of attributes according to the semantic rules of production. Evaluation Order of SDD includes how SDD is evaluated with the help of attributes, dependency graphs, semantic rules, and S and L attributes.

→ Dependency Graphs represent the flow of information among one Attributes in a parse tree.

→ Dependency Graphs are useful for determining Evaluation orders for Attributes in a parse tree.

→ While Annotated parse tree shows the values of attributes, a DG determines how those values can be computed.

Ex: $E \to E+T$ — $E.val = E_1.val + T.val$

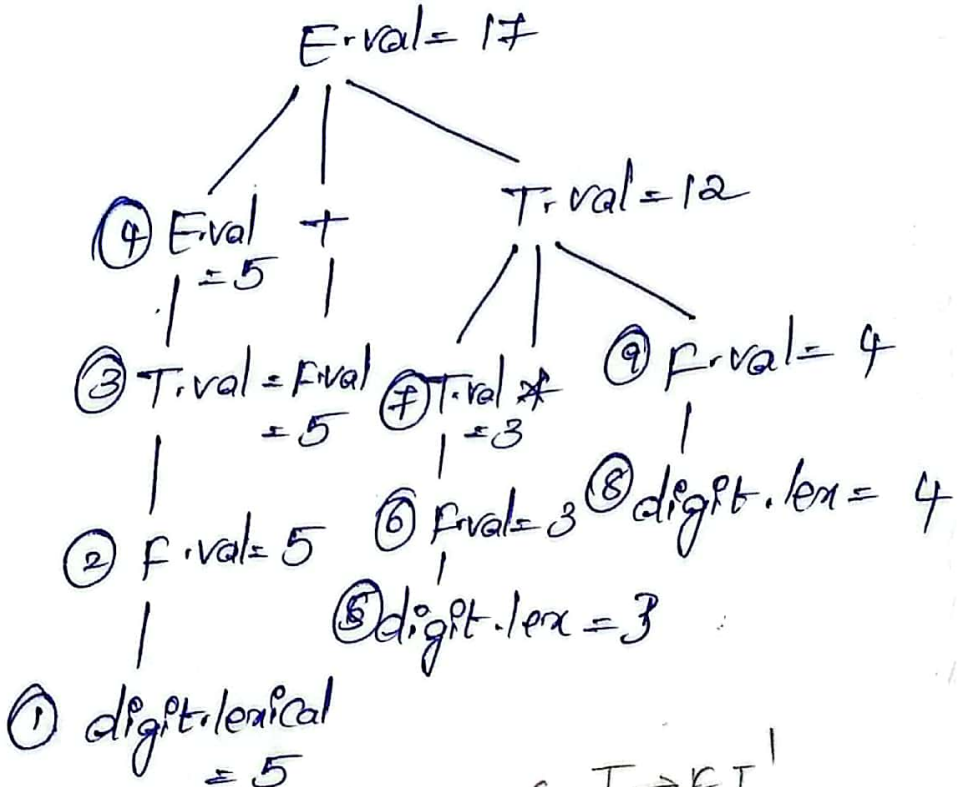
$E \to T$ — $E.val = T.val$

$T \to T*F$ — $T.val = T.val * F.val$

$T \to F$ — $T.val = F.val$

$F \to digit$ — $F.val = digit.lexval$

$5+3*4$




$T \to FT'$

$T' \to *FT'$

$T' \to \epsilon$

$F \to digit$

$T'.inh = F.val$

$T.val = T'.syn$

$T'.inh = T'in * F.val$

$T'.Syn = T_1'Syn$

$T'.syn = T'.inh$

$F.val = digit.lexval$

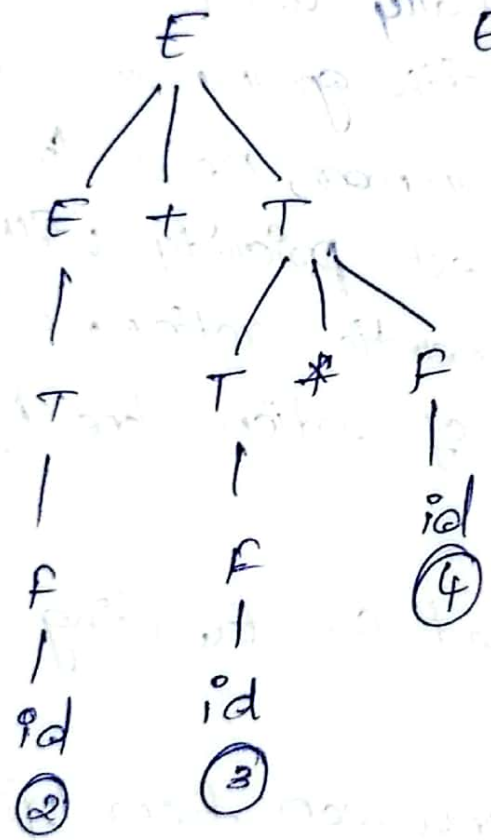
c) Applications of Syntax Directed Translation:

It is used for Semantic analysis and SDT is basically used to construct the parse tree with grammars and semantic action. In Grammars, need to decide who has the highest priority will be done first and in semantic actions, will decide what type of action done by grammars.

1. SDT is used for Executing Arithmetic Expression.
2. In the Conversion from Infix to postfix expression.
3. It is also used for Binary to Decimal Conversion.
4. In Creating Syntax tree.
5. SDT is used to generate intermediate code.
6. SDT is Commonly used for typechecking also.

Input : 2+3*4

O/p : 14




E.val = Eval + T.val
= 2+12
= 14

T.val = T.val * F.val
= 3*4
= 12

Semantic Action :

$E \rightarrow E+T$ { Eival = Eival + T.val then print E.val }

| T { Eival = T.val }

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

| F { T.val = F.val }

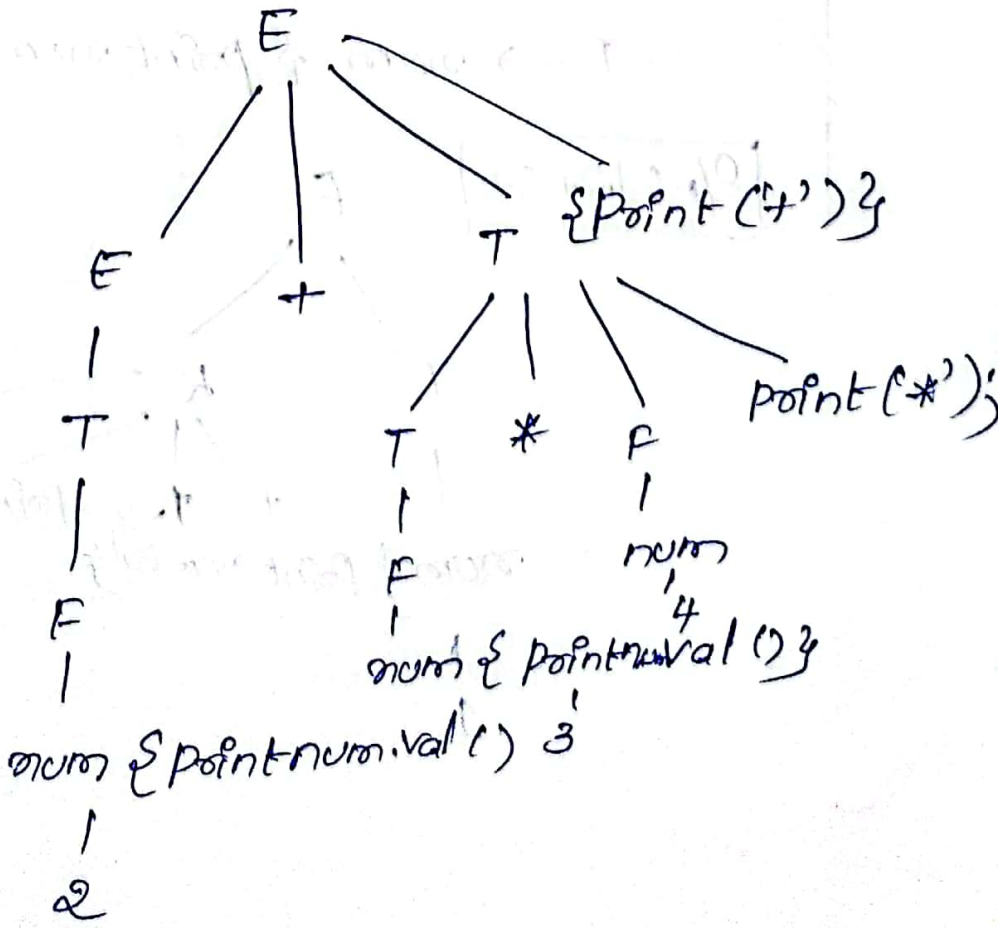
$F \rightarrow id$ { F.val = id }

Translation Scheme to Convert infix to post fix Expression.

Method1 : 2+3*4

| Production | Semantic Action |
|---|---|
| $E \rightarrow E+T$ | {print('+');} |
| $E \rightarrow T$ | |
| $T \rightarrow T*F$ | {print('*');} |
| $T \rightarrow F$ | |
| $F \rightarrow num$ | {print(num.val);} |

O/p : 234*+



Ex: 1+2+3

$E \rightarrow \underbrace{E+T}_{A}\ \{ \text{point}('+'); \}\ /\ \underset{\beta}{T}$

$T \rightarrow \underset{\alpha}{\text{num}}\ \{ \text{point num.val} \}$

$A \rightarrow A\alpha / \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \epsilon$

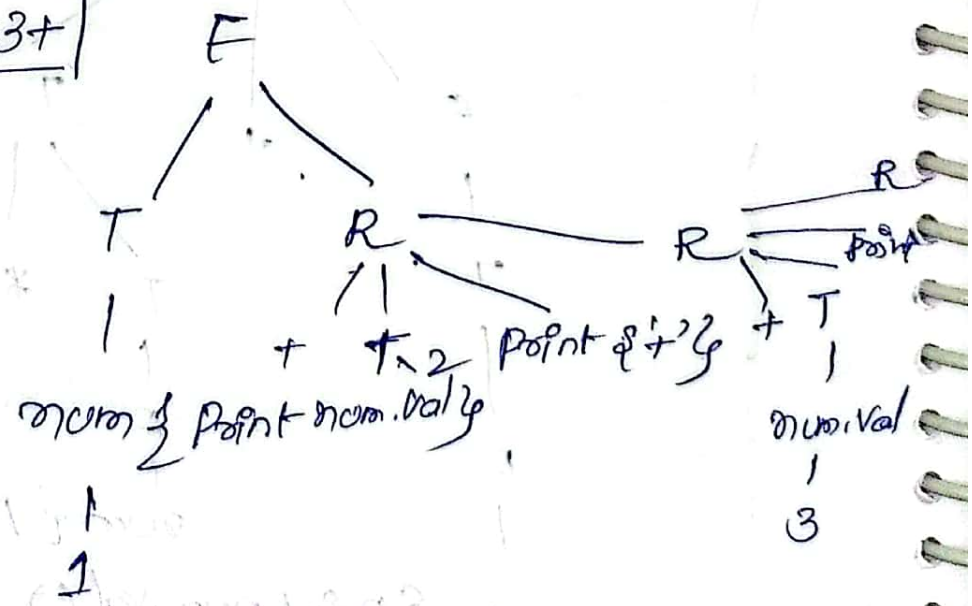
$E \rightarrow TR$

$R \rightarrow +T\ \{ \text{point}('+'); \}\ R$

$R \rightarrow \epsilon$

$T \rightarrow \text{num}\ \{ \text{point num.val} \}$

**O/p: 12+3+**



D) Syntax Directed Translation Schemes:

Syntax Directed Translation is a set of productions that have semantic rules embedded inside it. The SDT helps in the semantic analysis phase in the compiler. The types of SDT schemes are

→ Postfix Translation Schemes.
→ Parser Stack Implementation of postfix SDTs:
→ SDT with action inside the production
→ Eliminating left Recursion from SDT.

1. Post Translation Schemes:

* The Syntax directed translation which has its semantic actions at the end of the production is called the Postfix Translation Scheme.

* This type of translation of SDT has its corresponding semantics at the last in the RHS of production.

Ex: $S \rightarrow A * B$ { S.val = A.val * B.val }
$A \rightarrow B + 1$ { A.val = B.val + 1 }
$B \rightarrow num$ { B.val = num.lexval }

2. Parser Stack Implementation of Post-fix SDTs:

Post-fix SDT are implemented when the semantic actions are at the right end of production and with bottom up parser with non terminals having Synthesized attributes.

→ The parser stack contains the record for the nonterminals in the grammar and their corresponding attributes.

→ The non terminal symbols of the production are pushed onto the parser stack.

→ If the attributes are Synthesized and semantic actions are at right hand then attributes of nonterminals are evaluated for the symbol of in the top of the stack.

→ When the reduction occurs at the top of stack the attributes are available in the stack and after the action occurs these attributes are replaced by corresponding LHS non terminal and its attribute.

Ex: $A \rightarrow BC$ { A.str = B.str . C.str }

Initially parser stack

| | Nonterminals |
|---|---|
| B | C |
| B.str | C.str |

Synthesized attributes

↑

Top of Stack

After reduction occurs $A \rightarrow BC$

| | Nonterminals |
|---|---|
| A | |
| A.str | |

Synthesized attributes

↑

Top of Stack

3. SDT with action inside the production:

When the semantic actions are present anywhere on the rightside of production then it is SDT with action inside the production.

It is evaluated and actions are performed immediately after the left-non-terminal is processed.

This type of SDT includes S and L attributed SDTs.

If the SDT is parsed into bottom up parser then actions are performed immediately after the occurence of non-terminal at the top of parser stack.

If the SDT is parsed into top down parser then actions are before the expansion of non. terminal

Ex: $S \rightarrow A + \{print '+'\} B$

$A \rightarrow \{print num\} B$

4. Elimination of LR from SDT:

The grammar with LR cannot be parsed by topdown parser. So left recursion should be eliminated and the grammar can be transformed by eliminating it.

| Grammar with LR | Grammar without LR |
|---|---|
| $P \rightarrow P\alpha / q$ | $P \rightarrow qA$ <br> $A \rightarrow \alpha A / \epsilon$ |

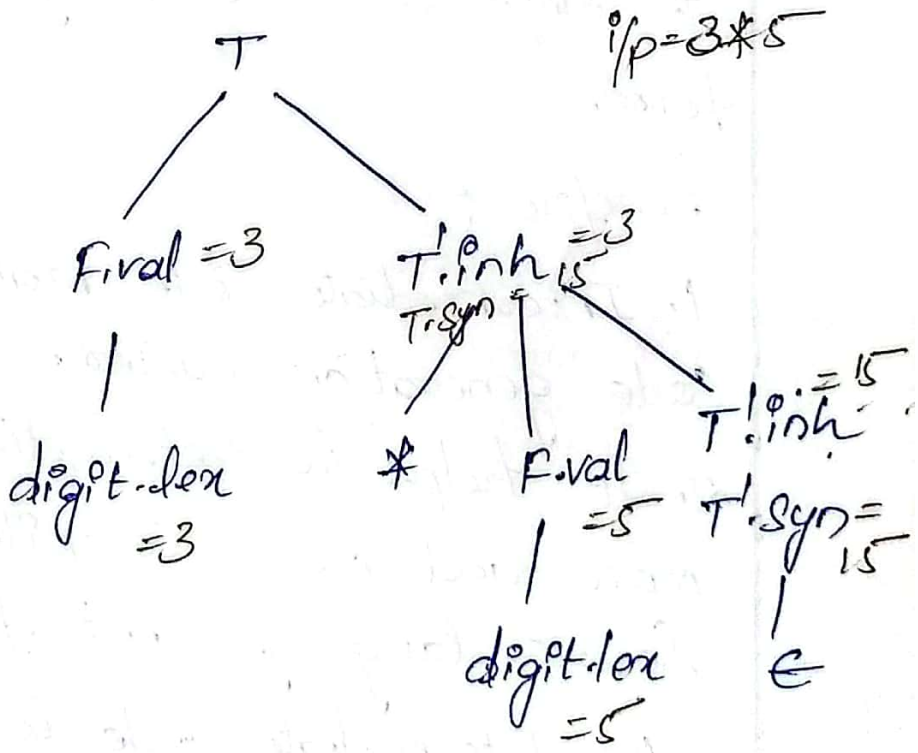
## ⇒ Implementation of L-Attributed SDD's:

Ex: $T \rightarrow FT'$ { T.val = T'.syn
T'.inh = F.val }

$T' \rightarrow *FT'$ { T'.inh = T'.inh * F.val
T'.syn = T'.syn }

$T \rightarrow \epsilon$ { T'.syn = T'.inh }

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



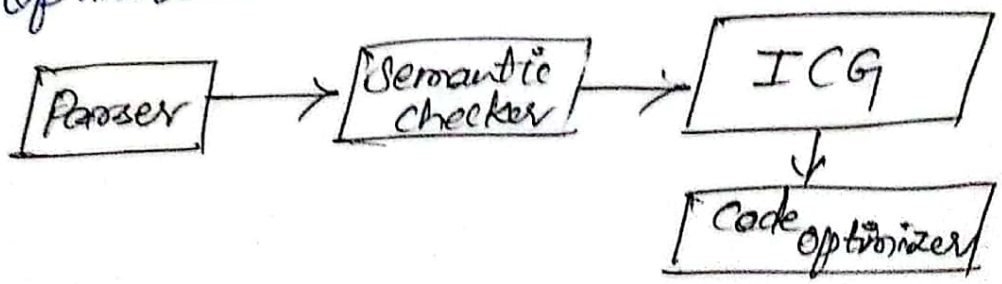
II Intermediate Code Generation :-

A) Variants of Syntax Trees :

The Intermediate code is useful representation when compilers are designed as two pass system i.e., as front end and back end. The Intermediate Code can be generated by modifying the Syntax directed Translation rules to represent the program in intermediate form.

Benefits :

1. Intermediate code makes the target code generation easier.
2. It helps in retargeting, creating more and more compilers for the same source language but for different machines
3. As Intermediate code is machine independent, it helps in machine Independent Code optimization.



A Syntax tree is a graphical representation of source program. Each internal node represents operators and leaf node operands.

1. mknode(op, left, right)
2. mkleaf(id, entry to Symboltable)
3. mkleaf(Num, value)

1. It uses for create a operator, left child and right child address.
2. mkleaf is used to create node for identifiers are stored in Symbol table/ pointer (or) address.
3. mkleaf is used to create num it represent a number, value represents value of number.

Ex: Construct Syntax tree for

$x*y-5+z$.

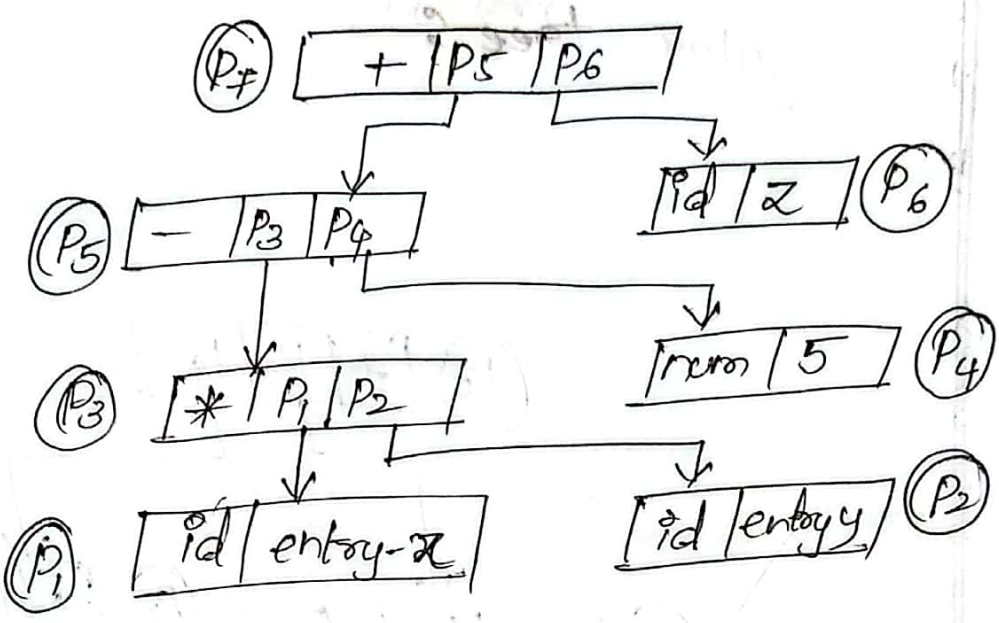
States to construction of Syntax tree

$x*y-5+z$.

| Symbol | Operation |
|---|---|
| x | $p_1 = mk\_leaf(id, entry\ to\ x)$ |
| y | $p_2 = mk\_leaf(id, entry\ to\ y)$ |
| * | $p_3 = mk\_node(*, p_1, p_2)$ |
| 5 | $p_4 = mk\_leaf(num, 5)$ |
| − | $p_5 = mk\_node(-, p_3, p_4)$ |
| z | $p_6 = mk\_leaf(id, entry\ to\ z)$ |
| + | $p_7 = mk\_node(+, p_5, p_6)$ |

SDD for $x*y-5+z$

| | |
|---|---|
| $E \to E_1 + T$ | $E.node = mk\_node(+, E_1.node, T.node)$ |
| $E \to E_1 - T$ | $E.node = mk\_node(-, E_1.node, T.node)$ |
| $E \to E_1 * T$ | $E.node = mk\_node(*, E_1.node, T.node)$ |
| $E \to T$ | $E.node = T.node$ |
| $T \to id$ | $T.node = mk\_leaf(id, id.entry)$ |
| $id\ T \to num$ | $T.node = mk\_leaf(num, num.val)$ |




Ex: Construct a syntax tree for

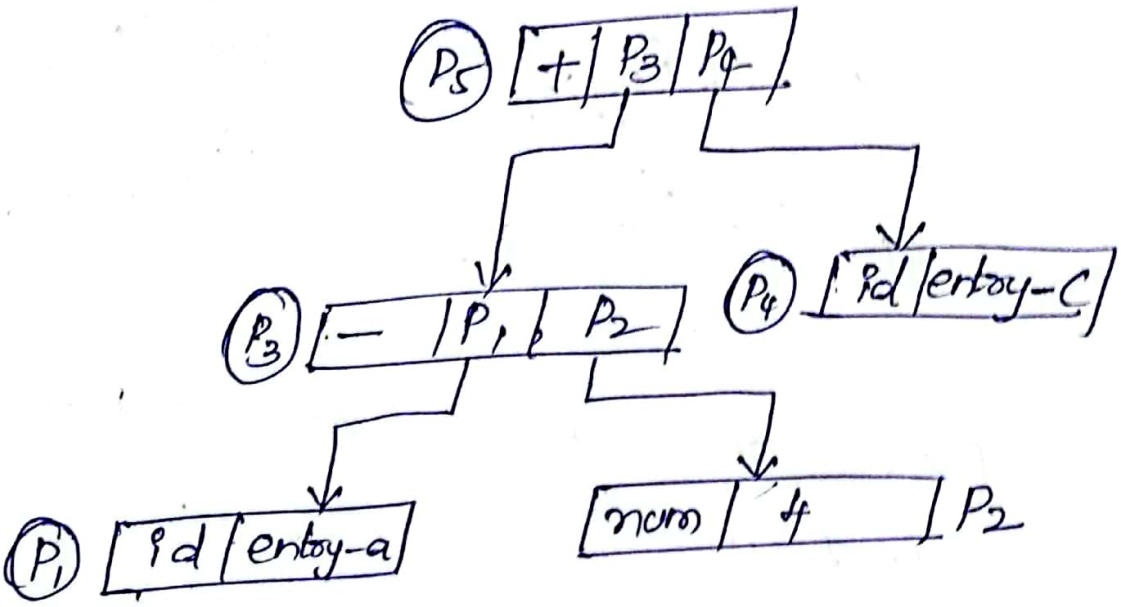
$a - 4 + c$

| | |
|---|---|
| a | $P_1$ = mknleaf (id, entry of a) |
| 4 | $P_2$ = mknleaf (num, 4) |
| − | $P_3$ = mknnode (−, $P_1$, $P_2$) |
| c | $P_4$ = mknleaf (id, entry of c) |
| + | $P_5$ = mknnode (+, $P_3$, $P_4$) |

SDD:

| | |
|---|---|
| $E \to E + T$ | { E.node = mknnode (+, $E_1$.node, T.node) |
| $E \to E - T$ | E.node = mknnode (−, E.node, T.node) |
| $E \to T$ | E.node = mknleaf (T.node) |
| $T \to id$ / $T \to num$ | T.node = mknleaf (num, num.val) |

Syntax tree :-




2. Directed Acyclic Graph (DAG)

1. In a DAG, internal node represents operators (Parent node)
2. Leaf node represents identifiers, Constants.
3. Internal node also represents result of expressions.

DAG represents the structure of basic block.

Applications :

→ Determining the common sub expressions.

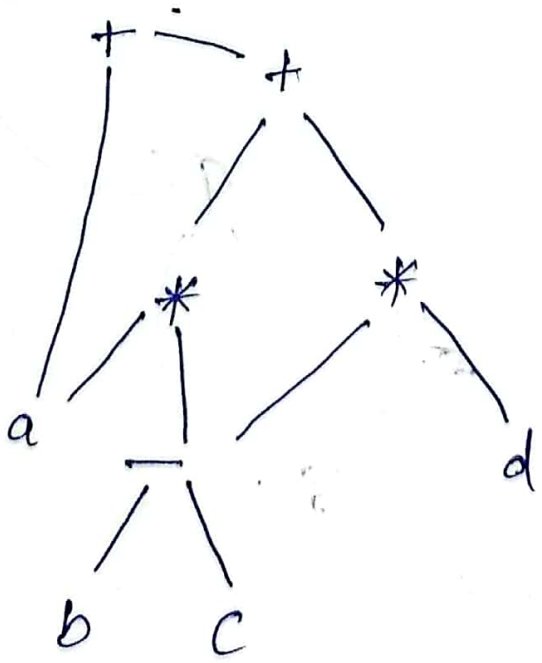
→ Determining which names are used inside the block and computed outside the block.

→ Determining which statements of the block could have their computed value outside the block.

→ Simplifying the list of quadruples by eliminate common subexpressions

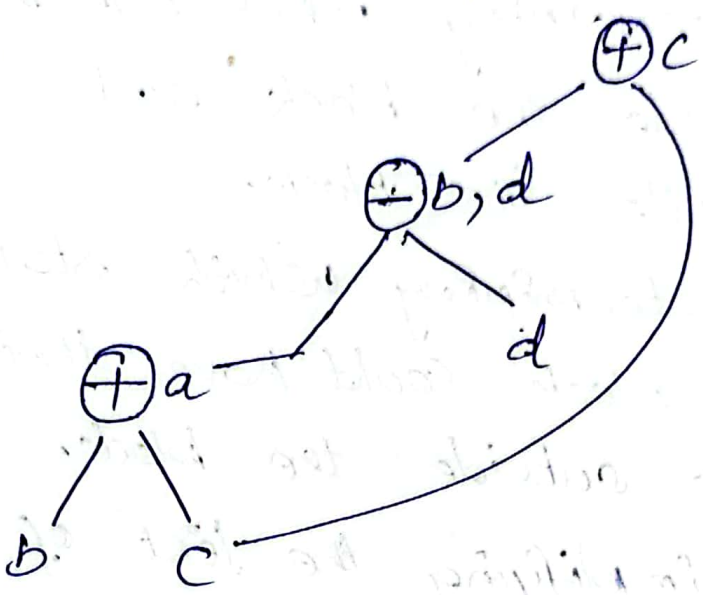
Ex: a + a * ( b - c ) + ( b - c ) * d

Paranthesis has highest priority.



Ex: 1. $a = b + c$ 3. $c = b + c$

2. $b = a - d$ 4. $d = a - b$




Ex: $a = b + c$

$b = b - d$

$c = c + d$

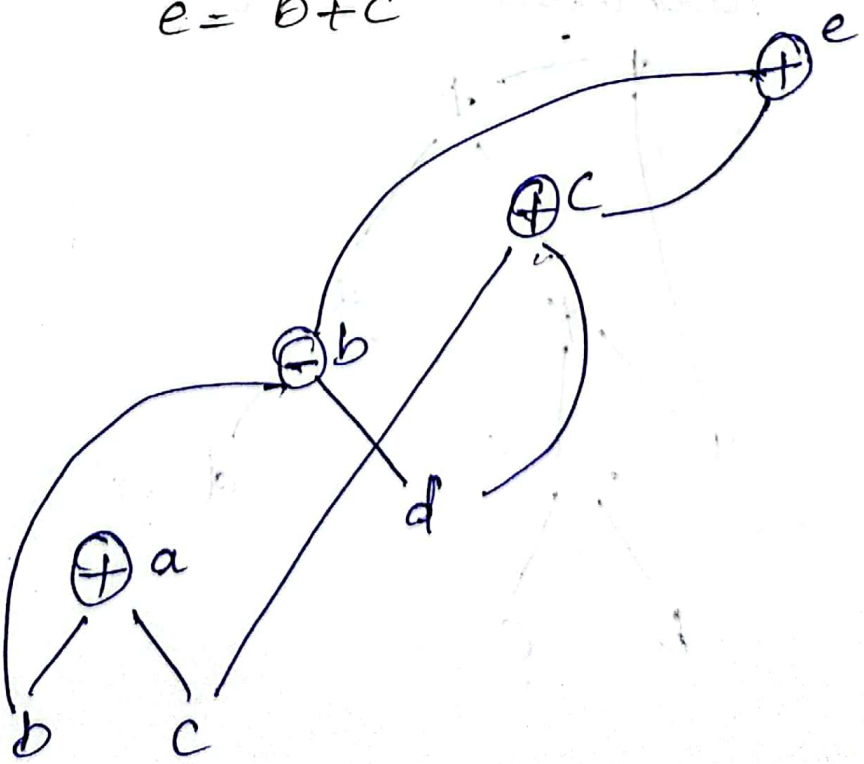
$e = b + c$

Ex: $d = b * c$

$e = a + b$

$b = b * c$

$a = e - d$



By 3. Three Address Code Representation:

→ Each Instruction contain atmost three address.

→ Atmost 1 Operator on R.H.S

1. Quadruple — Operator, arg 1, arg 2, result } 4 fields.
2. Triple
3. Indirect triple

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

unary minus has highest priority

$t_1 = -c$

$t_2 = b * t_1$

$t_3 = -c$

$t_4 = b * t_3$

$t_5 = t_2 + t_4$

(or)

$t_3 = b * t_1$

$t_4 = t_2 + t_3$

| | Op | arg 1 | arg 2 | result |
|---|---|---|---|---|
| (0) | − | c | | $t_1$ |
| (1) | * | b | $t_1$ | $t_2$ |

| | | | | |
|---|---|---|---|---|
| (2) | − | c | $t_3$ | Disadv: |
| (3) | * | b | $t_4$ | → too many temporary variables. |
| (4) | + | $t_2$ | $t_4$ | $t_5$ |

Taiple:

| | op | arg1 | arg 2 |
|---|---|---|---|
| (0) | − | c | |
| (1) | * | b | t0) |
| (2) | − | c | |
| (3) | * | b | $t_3$ (2) |
| (4) | + | c) | (3) |

Indirect Taiple: Pointers

| | |
|---|---|
| 100 | (0) |
| 101 | (1) |
| 102 | (2) |
| 103 | (3) |
| 104 | (4) |

1. Assignment Instructions of the form
$x = y$ op $z$ where op is binary operator

ex: $x = y + z$ , $x = y > z$, $x = y \&\& z$

2. Assignment of the form $x =$ op $y$
where op is unary operator.
Such as Binary-, incre, decre, logicalnot

ex: $x = -y$

$x = !\, y$

3. Copy Instructions of the form $x = y$
where value of $y$ is assigned to $x$.

ex: $x = y$ ;

$y = 10$ ;

4. UnConditional Jump goto L.

ex: goto L ;

L : stmt be executed.

5. Conditional Jump of the form
if $x$ relop $y$ goto L.

ex: if $x > y$ got L

6. Procedure calls and returns are implemented using following

Parameter x

y = call P, x

return y;

7. Address and pointers assignments of the form

x = &y

x = *y

*x = y

8. Indexed copy Instruction

a = y[i]

x[i] = y

Ex: Construct Quadraples, triples, Indirect triples for (a+b) * (c+d) - (a+b+c)

$t_1 = a + b$

$t_2 = c + d$

$t_3 = t_1 * t_2$

$t_4 = t_1 + c$

$t_5 = t_3 - t_4$

| | op | arg1 | arg2 | result |
|---|---|---|---|---|
| (0) | + | a | b | $t_1$ |
| (1) | + | c | d | $t_2$ |
| (2) | * | $t_1$ | $t_2$ | $t_3$ |
| (3) | + | $t_1$ | c | $t_4$ |
| (4) | - | $t_3$ | $t_4$ | $t_5$ |

| | op | arg1 | arg2 |
|---|---|---|---|
| (0) | + | a | b |
| (1) | + | c | d |
| (2) | * | (0) | (1) |
| (3) | + | (0) | c |
| (4) | - | (2) | (3) |

| Pointers | |
|---|---|
| 100 | (0) |
| 101 | (1) |
| 102 | (2) |
| 103 | (3) |
| 104 | (4) |

c) Types and Declarations :—

Typical basic types and declarations for a language contain boolean, char, integer, float and void. A type name is a type expression we can form a type expression by applying the array type constructor to a number and a type expression.

Declaration involves allocating space in memory and entering type and name in the symbol table. Memory allocation is consecutive, and it assigns names to memory in the sequence they are declared in pgm.

Applications of Types and Declarations:

* Type checking: It uses a logical rules to reason regarding the behaviour of a program at runtime. It ensures that the operands match the type that an operation expects.

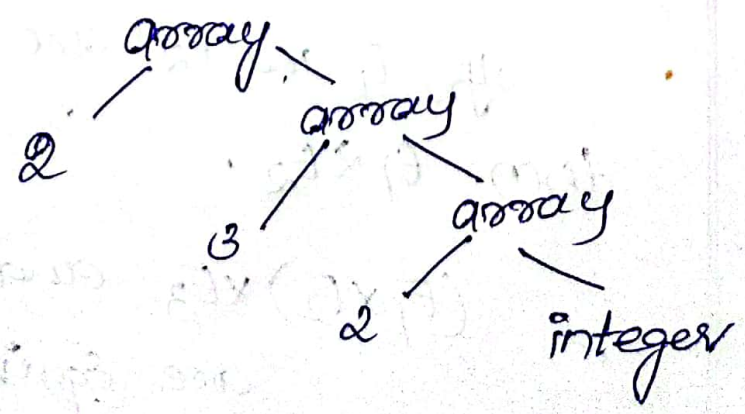
* Translation Application: A compiler can determine the storage that we will need for that name at runtime from the type of name.

Type Expressions:

A primary type is a type of expression. Typical basic types for a language contain boolean, char, integer, float, void (implies the value is absence)

We form a type expression by applying array type constructor to a number and a type expressions.

Ex:



The array type int[2][3][2] can be read as "array of 2 arrays each of 3 arrays of 2 integer each"

type expression array(2, array(3, array(2, integer))).

Basic types:

char, int, double, float are type expressions.

Type names:

It is convenient to consider that names of types are types expressions.

Arrays

If E is an type Expression and n is an int, then

array(n, E)

product:

If $E_1$ and $E_2$ are Type Expressions then $E_1 \times E_2$.

$(E_1 \times E_2) \times E_3$ and $E_1 \times E_2 * E_3$ are Equivalent.

Records:

The only difference b/w a record and product is that the fields of record have names.

If abc and xyz are type names if $E_1$ and $E_2$ are type expressions then

record (abc. $E_1$ , xyz. $E_2$)

pointers

Let E is a type expression then pointer(E) is a type expression denoting that type pointer to an object of type T.

function types:

If $E_1$ and $E_2$ are Type expressions then $E_1 \rightarrow E_2$ is a type expression from $E_1$ with an element from $E_2$.

Declarations :-

When we encounter declarations, we need to layout storage for declared variables. For every local name in a procedure we create a symbol table entry including

1. The type of name.
2. How much storage the name requires

Grammar:

$D \rightarrow$ real, id

$D \rightarrow$ Integer, id

$D \rightarrow D_1$, id

We use Enter to enter the Symbol table and ATTR to trace the datatype.

| Production rule | Semantic Action |
|---|---|
| $D \rightarrow$ real, id | Enter (id.PLACE, real)<br>D.ATTR = real |
| $D \rightarrow$ Integer, id | ENTER (id.PLACE, Integer)<br>D.ATTR = Integer) |
| $D \rightarrow D_1$, id | Enter (id.PLACE, $D_1$.ATTR)<br>D.ATTR = $D_1$.ATTR, |

D) Type checking in ICG:

Type checking is the process of verifying and enforcing constraints of types in values. A compiler must check that the source program should follow the syntactic and semantic conventions of the source language. It allows the programmer to limit what types may be used in certain circumstances and assign types to values.

Conversion:

Conversion from one type to another type automatically. is known as implicit / coercion.

Ex: Real is not converted to Int but Int converted to Real.

Conversion is said to be Explicit if the programmer writes some thing to do conversion.

Typechecking Types:

1. Static Type checking
2. Dynamic Type checking

## Static type checking:

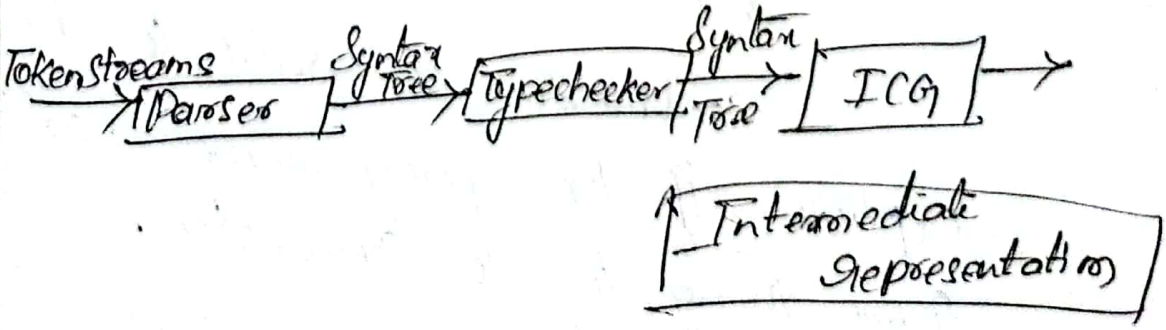
It is defined as type checking performed at compile time. It checks the type variables at compile time.

Benefits:

1. Runtime Error Protection.
2. It Catches Syntactic errors.
3. It Catches wrong names.
4. Detects incorrect argument types.

## Dynamic type checking:

It is defined as the type checking being done at runtime. In Dynamic type checking types are associated with values not variables.



D) Control Statements :— Flow :

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

Code for Simple if

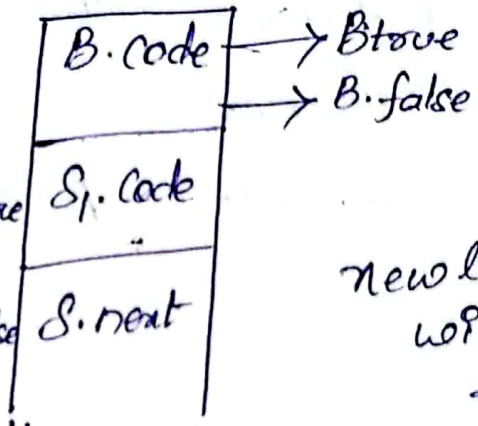
Semantic Rule :

B.true = newlabel()

$S_1$.next = S.next

B.false = S.next

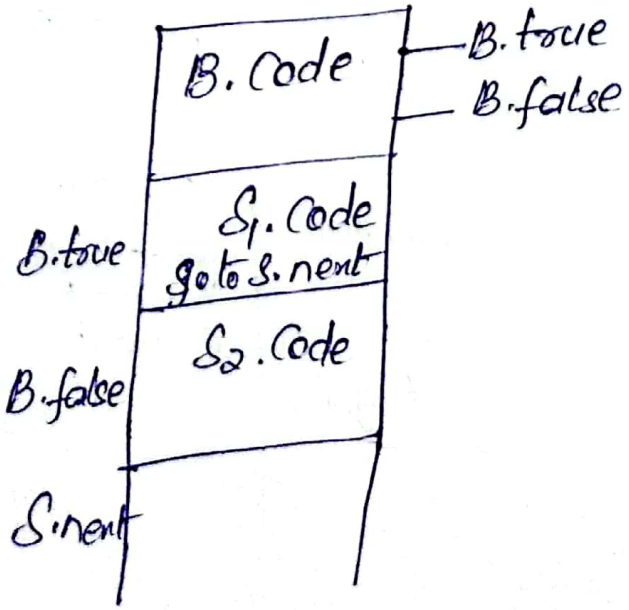
S.code = B.code || label(B.true || $S_1$.code.




newlabel() will generate three address code.

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

Code for if else.



Semantic rules :

B.true = newlabel()

$S_1$.next = S.next

B.false = newlabel()

$S_2$.next = S.next

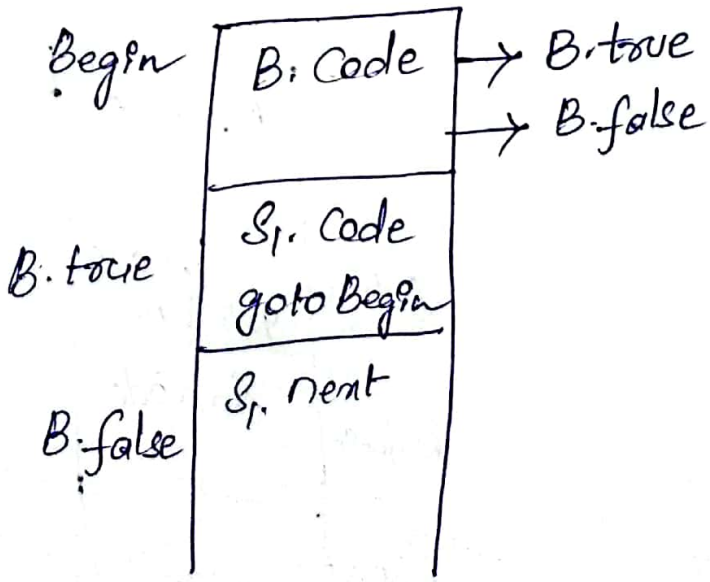
S.code = B.code || Label(B.true) || $S_1$.code || generate('goto' S.next) || label(B.false) || $S_2$.code

Concatenation

Production :

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

Code for while



Semantic Rules :

Begin = new label()

B.true = new label()

$S_1$.next = Begin

B.false = S.next

S.code = label(Begin) || B.code || (B.true) label

|| $S_1$.code || 'goto' begin. || label(false)

Generate

||S.code

E) Switch Statements :

Syntax : Switch (E)

{

Case $V_1$ : $S_1$

Case $V_2$ : $S_2$

:

Case $V_{n-1}$ : $S_{n-1}$

default : $S_n$

}

Statement block : $S_1, S_2, S_3 \ldots S_n$

# Translation of Switch Statement

T stores the result of Expression.

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 next :

f) Intermediate Code for Procedures :

$D \rightarrow$ define $T$ id $(F)\{S\}$

$F \rightarrow \epsilon / T$ id, $F$

$S \rightarrow$ return $E$ ;

$E \rightarrow$ id$(A)$ ;

$A \rightarrow \epsilon / E$ , $A$ ;

$F \rightarrow$ Formal parameters

$T \rightarrow$ type name