# UNIT- III

**Syntax Directed Translation**

## Syntax directed definitions:

* Syntax directed definition (SDD) is context free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules associated with productions.
* Attributes may be of any kind: numbers, types, string or memory location.
* If X is symbol and a is attribute then we write X.a to denote value of a at particular node labelled X in parse tree
* We shall deal with two kinds of attributes
  1. Synthesized attribute
  2. Inherited attribute

### Synthesized attribute:

* Synthesized attribute for non terminal A at parse tree node N is defined by semantic rules associated with production at N and A is head.
* Synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself.
* An SDD that involves only synthesized attribute is called S-attributed definition. Example: consider grammar for desk calculator

L → En

E → E1+T

E → T

T → T1\*F T → F

F → (E)

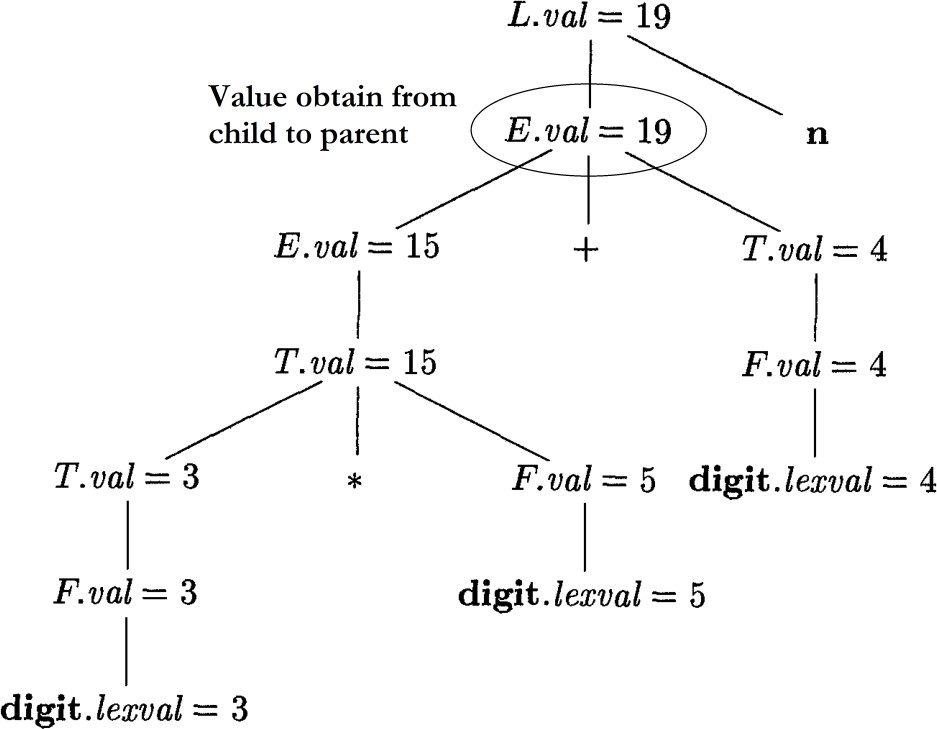
F → digit

Syntax directed definition for disk calculator is given in below table with semantic actions for each production

|  |  |
| --- | --- |
| Production | Semantic rules |
| L → En  E → E1+T  E → T  T → T1\*F T → F  F → (E)  F → digit | L.val=E.val  E.val= E1.val + T.val E.val = T.val  T.val =T1.val \* F.val T.val=F.val F.val=E.val  F.val=digit.lexval |

* Token digit has synthesized attribute lexval whose value can be obtained from lexical analyser.
* All terminals has synthesized attribute, but not inherited attribute.We visualize translation specified by SDD with the help of parse tree
* Parse tree showing the value of its attribute is called annotated parse tree.

Example: construct annotated parse tree for string 3 \* 5 + 4n using desk calculator grammar.



We start form left most child element, it is reduce to F with F.val=digit.lexval then F reduce to T with T.val=F.val finally T.val=5.

Likewise, T → T1 \* F means T.val = T1val \* F.val T.val = T1 .val \* F.val

= 3 \* 5

= 15

Similarly, E → E1 + T means E.val = E1.val + T.val E.val = E1.val + T.val

= 15 + 4

= 19

Finally L is a parent for E means L.val = E.val L.val = 19 n

### Inherited Attribute:

* Inherited attribute for non terminal B at parse tree node N is defined by semantic rule associated with production at parent N and B is in body of production.
* Inherited attributes at node N is defined only in terms of attribute values at N's parent, N itself and N's siblings.
* An SDD is called L-attribute definition if it follow below rules with production A → X1X2 Xn for computing Xi.a

1. Inherited Attributes associated with head A.Inherited Attributes associated with occurrences of symbols X1, X2 Xi-1 located to left

of Xi.

* L-attributes definition consists of both S-attribute and L-attribute.

**Example:** Consider Grammar D → T L

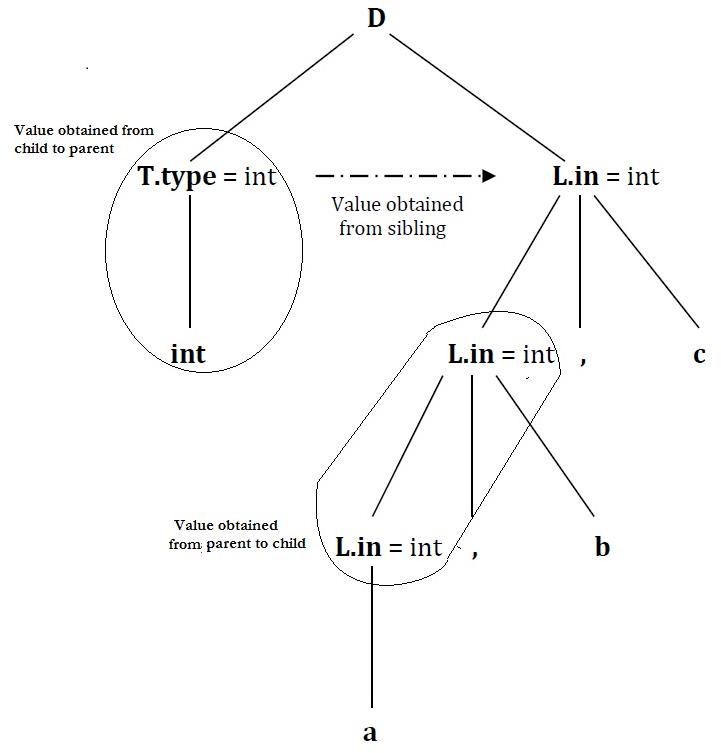
T → int T → float

L → L1, id L → id

Syntax directed definition for simple type declaration is shown in below table

|  |  |
| --- | --- |
| Production | Semantic rules |
| D → T L  T → int T → float  L → L1, id  L → id | L.inh = T.type T.type = integer T.type = float L1.inh = L.inh  addType(id.entry, L.inh)  addType(id.entry, L.inh) |

* Non terminal D represents a declaration, production 1 consists of type T followed by List L of identifiers.
* T has one attribute T.type which is type in declaration. L has one attribute, inh that is inherited attribute. It is used to pass down type to list of identifiers and also it adds to appropriate symbol table entry.

**Example:** Construct annotated parse tree for string int a, b, c using simple type declaration grammar.

Value of L obtained from T.type (sibling). T.type is lexical value obtained as int or float then L nodes give type of identifiers a, b, c. Using addtype the type of identifiers a, b, c is inserted in symbol table.

### Evaluation Orders for SDD's:

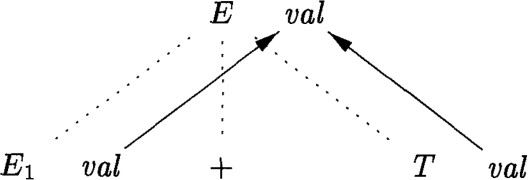
* Dependency graphs are useful tool for determining evaluation order for attributes instances in given Parse tree.
* While annotated Parse tree shows values of attributes, a dependency graph helps us to determine how these values can be computed.

### Dependency Graphs:

* Dependency graph shows the flow of information among attributes instances in particular parse tree.
* Edge from one attribute instance to other means value of first is needed to compute second.
* Edges express constraints suggest by semantic rules.

1. For each parse tree node labelled by grammar symbol X, dependency graph has node for each attribute associated with X.
2. Suppose that semantic rules associated with production P defines the value of synthesized attribute A.b in terms of value of X.c. Then dependency graph has edge from Xc to A.b. Where A is head and X is in body of production.
3. Suppose that semantic rules associated with production P defines value of inherited attribute B.c in terms of X.a. Then dependency graph has edge from X.a to B.c. Where B is in body of production and X be sibling or head of production.

**Example**: Draw dependency graph for the below production Production Semantic Rules

E → E1 + T E.val = E1.val + T.val Graph is as follows

Here dotted lines indicate Parse tree

Solid lines indicate dependency graph

**Example:** Draw dependency graph for simple type declaration shown below with string int a, b, c

S → T L

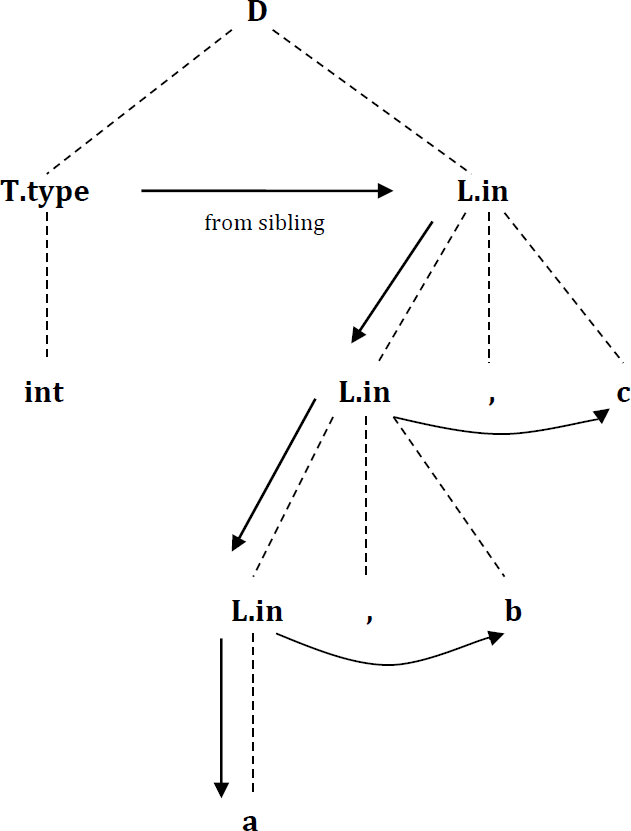
T → int

T → float L → L1, id L → id

Semantic rules of the above grammar

|  |  |
| --- | --- |
| Production | Semantic Rule |
| S → T L  T → int T → float  L → L1, id  L → id | L.in = T.type T.type = int T.type = float L1.inh = L.inh  addType(id.entry, L.inh) addType(id.entry, L.inh) |

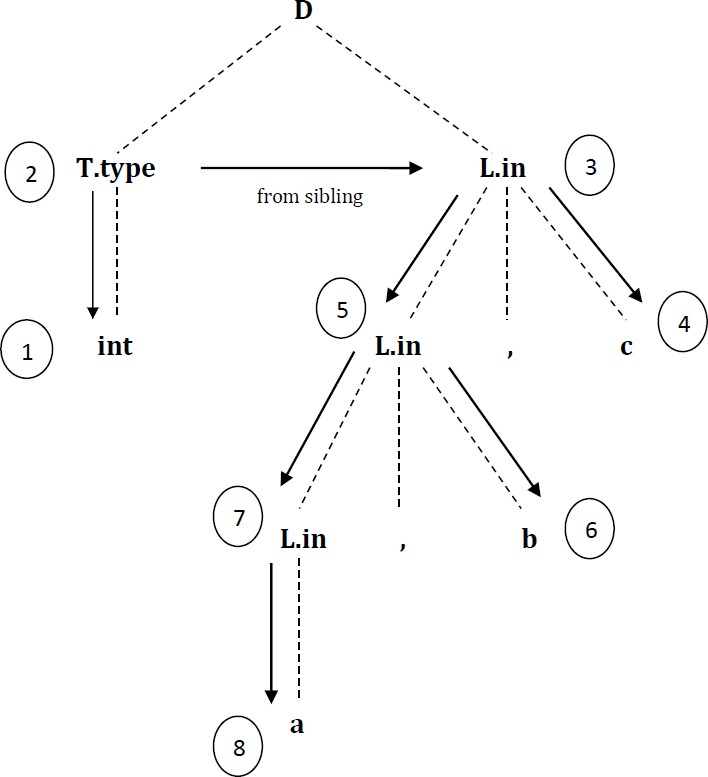
Dependency graph is



### Ordering the Evaluation of Attributes:

* Dependency graph characterizes the possible orders in which we can evaluate attributes at various nodes of parse tree.
* If dependency graph has an edge node M to node N, then attribute corresponding to M must be evaluated before attribute of N.
* Ordering attached to dependency graph is called Topological sort of graph.
* If there is any cycle in the graph, then there are no topological sorts i.e., no way to evaluate SDD on this parse tree.
* If there is no cycle, then there is at least one topological sort.

**Example:** Evaluation order for simple type declaration shown below with string int a, b, c



Evaluation order can be decided as follows:

1. Type int is obtained from lexical analyzer by input token.
2. L.in is assigned the type int from sibling T.type.
3. entry in the symbol table for identifier c gets associated with type int. Hence variable c becomes integer.
4. L.in is assigned type int from parent L.in.
5. Entry in symbol table for identifier b gets associated with type int. Hence variable b becomes integer.
6. L.in is assigned type int from parent L.in.
7. Entry in symbol table for identifier a gets associated with type int. Hence variable c becomes integer.

### Applications of Syntax Directed Translation:

* Main application in syntax directed translation is construction of Syntax trees.
* Some compilers use syntax tree as intermediate representation. Commonly SDD turns its input to tree.
* We consider two SDD's for constructing syntax tree for expressions:
  1. S - Attributed Definition: is useful during Bottom up Parsing.
  2. L - Attributed: is useful during Top down Parsing.

### Construction of syntax trees:

* We implement nodes of syntax tree by objects with suitable number of fields.
* Each object will have an op field that is label as node, object will have additional fields as follows

1. If node is leaf, additional field holds lexical value for leaf. Function Leaf(op, val) creates leaf object.
2. If node is interior node, there are many additional fields as the node has children in syntax tree. Function Node(op, c1, c2,…. ck) creates object with the first field op and k additional fields for k children.

**Example:** Construct syntax tree for given S-attribute definition grammar given below E → E1 - T

E → E1 + T

E → T

T → ( E )

T → num

Every time first production E → E+T is used, its rule creates node with ‘+’ for op and 2 children, E1.node and T.node. Second production has same rule.

Third production, E → T, no node is created since E.node is same as T.node. Similarly, no node is created for production 4, T → ( E ).

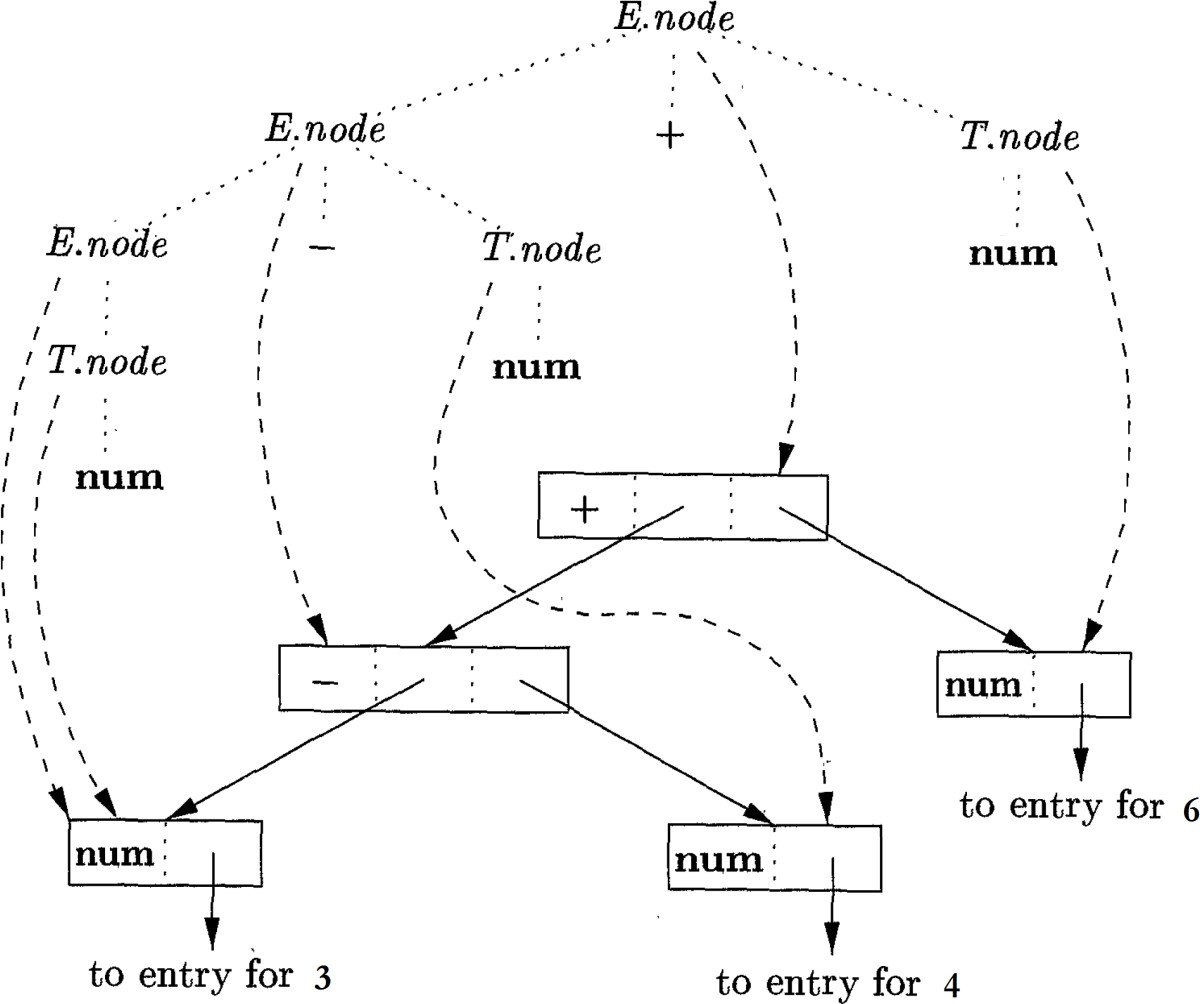
Last T productions have single terminal on right. We use leaf function to create node, which become value of T.node.

|  |  |
| --- | --- |
| Production | Semantic rule |
| E → E1 + T E → E1 - T  E → T  T → ( E )  T → num | E.node = newnode(‘+’, E1.node, T.node) E.node = newnode(‘-‘, E1.node, T.node) E.node = T.node  T.node = E.node  T.node = new leaf(num, num.val) |

Syntax tree for string 3 – 4 + 6 is

1. P1=new Leaf(num, 3)
2. P2=new Leaf(num, 4) 3. P3=new Node(‘-‘, P1, P2)
3. P4=new Leaf(num, 6)

5. P5=new Node(‘+’, P3, P4)



### Syntax directed Translation Schemes:

* Syntax directed translation schemes are complementary notation to syntax directed definition.
* All applications of syntax directed definitions can be implemented using syntax directed translation schemes.
* Syntax directed translation schemes (SDT) is a Context free grammar with program fragments embedded with production bodies.
* We place curly braces around actions; if braces are needed then we quote them.
* SDT can be implemented by first building Parse tree then forming actions in left to right depth first order.
* We focus on the use of SDT’s to implement 2 important classes of SDD’s
  1. Underlying grammar is LR Parsable and SDD’s S-attributed.
  2. Underlying grammar is LL Parsable and SDD’s L-attributed.

### Postfix Translation Schemes:

* SDT’s with all actions at the right ends of production bodies are called postfix SDT’s.
* We construct an SDT in which action is placed at the end of production and executed along with reduction of body to head of production.

**Example:** Postfix SDT’s for desk calculator are shown below L → En {print(E.val);}

E → E1 + T {E.val = E1.val + T.val;} E → T {E.val = T.val;}

T → T1\*F {T.val = T1.val \* F.val;} T → F {T.val = F.val;}

F → (E) {F.val = E.val;}

F → digit {F.val = digit.lexval;}

The action for first production prints a value it is the only change from SDD.

### Parser-Stack Implementation of Postfix SDT’s:

* Postfix SDT’s can be implemented during LR parsing by executing actions when reductions occur.
* Attribute of each grammar symbol can be put on the stack where they can found reduction.
* 3 grammar symbols XYZ are on top of stack and they are reduced according to production like A → XYZ.
* Here X.x as attribute of x amd so on. Initially X.x, Y.y and Z.z on stack before reduction.

|  |  |  |
| --- | --- | --- |
| Z.z | Top |  |
| Y.y |
| X.x | A.a | Top |

* After reduction 3 attributes of stack replaced by A.a based on A → XYZ.

**Example:** Rewrite actions of desk calculator SDT by manipulate Parser stack explicitly. These are done automatically by Parser.

Production Actions

L → En {print(stack[Top-1].val); top=top-1;}

E → E1 + T {stack[top-2].val = stack[top-2].val + stack[top].val; top=top-2;} E → T

T → T1 \* F {stack[top-2].val = stack[top-2].val \* stack[top].val; top=top-2;} T → F

F → (E) {stack[top-2].val = stack[top-1].val; top=top-2)} F → digit

* In second production, E → E1 + T , we go two position below top to get E1 value and T value at top, result sum E is place on top of stack after reduction.
* In third production, E → T, no action is necessary because length of stack not change and value of T is on top of stack same observation need for T → F and F → digit. Production F → (E) is slightly different

### SDT’s with actions inside productions:

* Action may be placed at any position within the body of production, it is performed immediately after all symbols to its left are processed.
* Production B → X {a} Y, action a done after recognizing X for all terminals derived from X.
  + 1. If parse is bottom up then perform action a as soon as occurrence of X appears on top as stack.
    2. If parse is top down, we perform a just before expansion of occurrence of Y. **Example:** We turn our desk calculator into SDT that prints the prefix form of expression. The productions and actions are given below.

1. L → En

2. E → {print(‘+’);} E1 + T

3. E → T

4. T → {print(‘\*’);} T1 \* F

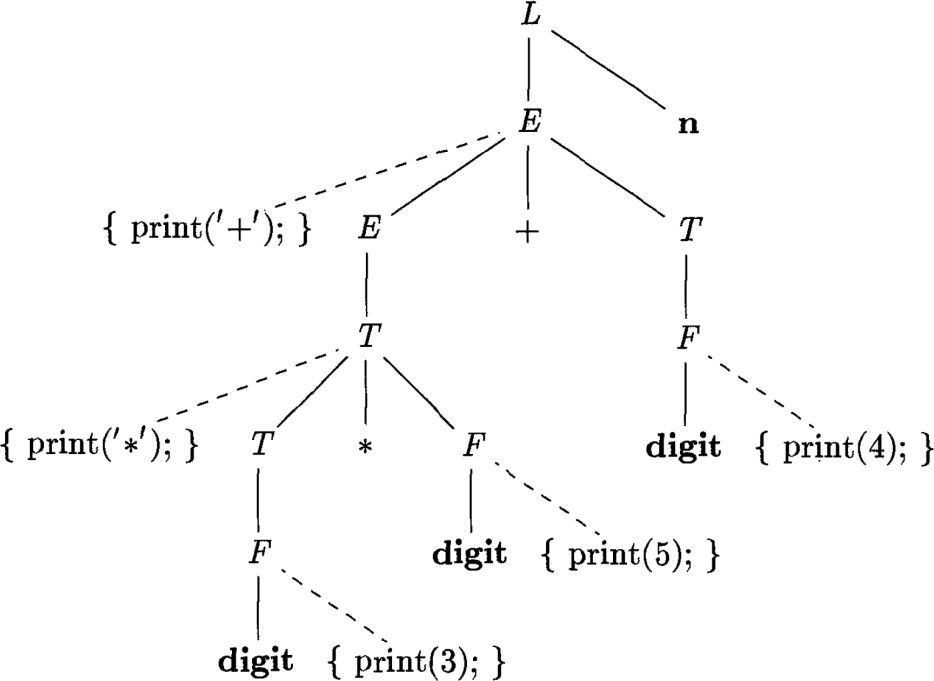
5. T → F

6. F → (E)

7. F → digit {print(digit.lexval);}

* Any SDT can be implemented as follows:

1. Ignore actions, parse input and produce parse tree as result.
2. Each interior node N, say one production A → 𝘢. Add additional children to N for actions in 𝘢, children of N have same symbols and actions of 𝘢
3. Perform preorder traversal of tree , as soon as node labeled by actions is visited, perform that action

* For instance below figure shows the parse tree for the expression 3 \* 5 + 4 with actions inserted. If we visit the nodes in preorder, we get prefix form of the expression: + \* 3 5 4

### Distinguish synthesized and inherited attributes:

|  |  |
| --- | --- |
| **Synthesized attribute** | **Inherited attribute** |
| 1. Attribute value computed from attribute values of children 2. Semantic actions for attributes evaluated after they expanded 3. They are used to pass information up the parse tree 4. All child elements are synthesized | 1. Attribute value computed form attribute values of parents or siblings 2. Semantic actions for attributes evaluated before they expanded 3. They are used to pass information down the parse tree 4. Except child elements, all are either   inherited or synthesized |

**Difference between S attributed and L attributed language**

|  |  |
| --- | --- |
| **S- Attributed** | **L- Attributed** |
| 1. A SDD is said to be S-attribute lf each grammar symbol have synthesized attributes 2. S-attributed definitions, synthesized attributes can be evaluated during bottom up traversal 3. Evaluation of attributes for grammar can be implemented during LR parse 4. They are general classes of attribute grammars | 1. A SDD is said to be L- attributed if each symbol has synthesized or inherited attribute 2. L- attributed definitions, attributes can always be evaluated during top down traversal 3. Evaluation of attribute for grammar can be implemented during LL parse 4. These are proper subset of S-attributed grammar |

.

# UNIT-IV

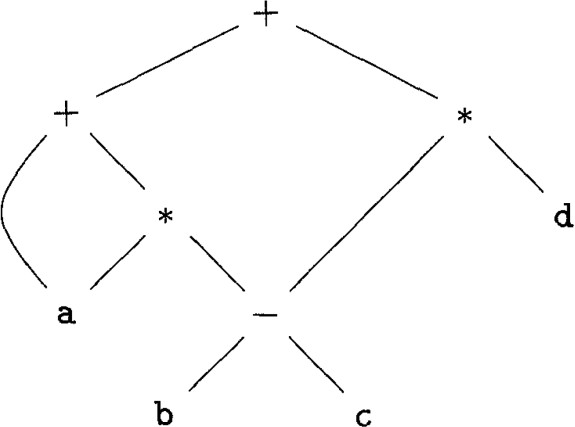
# Intermediate Code Generation

## Directed acyclic graph for expressions:

* Like syntax, DAG has leaves corresponding has leaves corresponding to atomic operands and interior nodes corresponding to operators.
* Difference is that a node N in dag has more than one parent if N represents common sub- expression.
* DAG gives important clues to compiler regarding generation of efficient code.

**Example:** Construct DAG for given expression with sequence of steps.

a + a \* (b - c ) + ( b – c ) \* d

Leaf for a has 2 parents, because a appears twice in expression. Two occurrences of common sub expressions b - c are representing by one node labelled -. Node has 2 parents, representing its 2 uses in sub expressions a \* ( b – c ) and ( b – c ) \* d.

P1=Leaf(id,entry-a)

P2= Leaf (id,entry-a)=P1 P3= Leaf (id,entry-b) P4= Leaf (id,entry-c) P5=Node(‘-‘, P3, P4)

P6=Node(‘\*’, P1, P5)

P7=Node(‘+’,P1,P6)

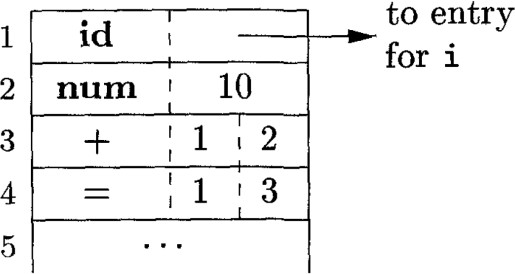
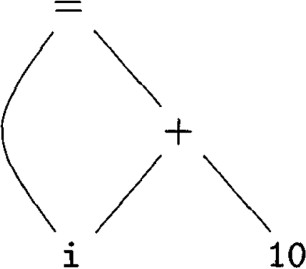
P8=Leaf(id, entry-b)=P3 P9=Leaf(id, entry-c)=P4 P10= Node(‘-‘, P3, P4)=P5

P11=Leaf(id, entry-d) P12=Node(‘\*’, P5, P11) P13=Node(‘+’, P2, P12)

### Value Number method for constructing DAG’s:

* Here node of syntax tree or DAG are stored in an array of records, each row of array represents one record(node).
* Each record, first field is an operation code indicating label of node leaves has additional field, which holds lexical value.
* Interior nodes have 2 additional fields indicating left and right children.

**Example:** DAG stored in array of records for i = i + 10



DAG Array

Here i should be entered by the user while running the program.

## Three address code:

* Here, there is at most one operator on right side of instruction.
* Source language expression like X+Y\*Z might be translated into sequence of three address code instructions.

t1 = Y \* Z t2 = X + t1

* Here t1 and t2 are compiler-generated temporary names.

### Instructions:

* List of common three address instruction forms:
  1. Assignment instruction of form x=y op z, where op is binary arithmetic or logical operation, x, y and z are addresses.
  2. Assignment instruction of form x=y, where x= op z, where op is unary operation.
  3. Copy instructions of form x=y , where x assign to y
  4. Unconditional Jump goto L, three address instructions with label L is next to executed.
  5. Conditional Jump of form if x goto L, these instruction with label L is next executable based on x value.
  6. Conditional Jump such as if x relop y goto L, these instruction with label L is next to execute based on relational operator between x and y.
  7. Procedure calls and returns are implemented using following instructions. Parm x for parameters

Call p, n for function call Return y for return value

* 1. Indexed copy instruction of form x=y[i] and x[i] = y, x = y[i] sets x to value in location i memory units in location y.
  2. Address and pointer assignments of form x=&y, x=\*y, x=&y means x store location y.

### Implementation of three address code:

* Three address code is an abstract form of intermediate code. It can be implemented as record with address fields.
* There are three types of representations are used to implement three address code
  1. Quadruple
  2. Triple
  3. Indirect triple

### Quadruple:

* Quadruple has four fields, which we call op, arg1, arg2 and result.
* Op field is used to represent internal code for operator, arg1 and arg2 represent 2 operands and result is used to store result of expression.

**Example:** Three address code for assignment a=b\*-c+b\*-c using quadruple representation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | op | arg1 | arg2 | result |
| 0 | minus | c |  | t1 |
| 1 | \* | b | t1 | t2 |
| 2 | minus | c |  | t3 |
| 3 | \* | b | t3 | t4 |
| 4 | + | t2 | t4 | t5 |
| 5 | = | t5 |  | a |

t1 = minus c t2 = b \* t1

t3 = minus c t4 = b \* t3

t5 = t2 + t4 a = t5

### Triple:

* Triple has only three fields, which we call op, arg1 and arg2
* In triple representation, instead of temporary variables like t1, t2, ...... we prefer referring pointers in symbol table.

**Example:** Three address code for assignment a=b\*-c +b\*-c using triple representation.

|  |  |  |  |
| --- | --- | --- | --- |
|  | op | arg1 | arg2 |
| 0 | minus | c |  |
| 1 | \* | b | (0) |
| 2 | minus | c |  |
| 3 | \* | b | (2) |
| 4 | + | (1) | (3) |
| 5 | = | a | (4) |

### Indirect triple:

* Indirect triple consists of listing pointer to tuples rather than listing of tuples.
* Let us an array instruction to list pointers to triples in desired order.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Instruction | |  |  | op | arg1 | arg2 |
| 20 | (0) | 0 | minus | c |  |
| 21 | (1) | 1 | \* | b | (20) |
| 22 | (2) | 2 | minus | c |  |
| 23 | (3) | 3 | \* | b | (22) |
| 24 | (4) | 4 | + | (21) | (23) |
| 25 | (5) | 5 | = | a | (24) |

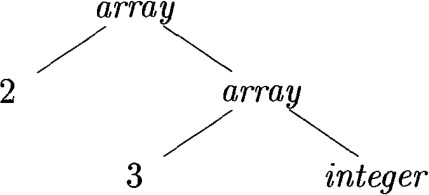
## Types and Declarations:

* Applications of types can be grouped under checking and translations:

1. Type checking:
   * It uses logical rules to explain about behaviour of program at runtime
   * It checks that types of operands match type expected by operator. Example: && operator in Java expected the 2 operands to be a Boolean
2. Translation Application:
   * From type of a name, compiler can determine storage that will need that name at runtime.
   * Type information is needed to calculate address denoted by array reference to insert explicit type conversion.

### Type Expression:

* + Type have structure, it will represent using type expression.
  + Type expression is either basic type or its formed by applying operator called type constructor to type expression.

Example: array type int [2][3] can read as ”array of two arrays of 3 integers each” and written type expression array(2, array(3, integer)”. It has 2 parameters; number, type.

* + The following definition of type expression

1. basic type is type expression .basic types for language include Boolean, char, integer, float and void(absence of value)
2. Type name is type expression.
3. Type expression can be formed by using the array type constructor to number and type expression.
4. Record is a data structure with naming fields. Type expression can be formed by applying record type constructor and their types.
5. Type expression can be formed by using the type constructor → for function types, we write s → t for function from type s to type t.
6. If s and t are type expression, then their Cartesian product, s x t is type expression.

### Type Equivalence:

* + If two type expressions are equal then return certain type else return certain type else error is rule for type checking.
  + If both type expressions are equal then those are type equivalence.
  + Two types are structurally equivalent if and only if one of the following condition is true.

1. They are same basic type.
2. They are formed by applying same construction to structurally equivalent types.
3. One type name denotes other.
   * If type name are type expression then 1st two conditions in the above lead to name equivalence of type expression.

### Declaration:

* + We study types and declarations using simplified grammar that declares just one name at a time. The grammar is

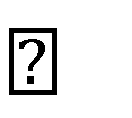
D → T id ; D | E

T → BC | record ‘{‘ D ’}’ B → int | float

C → E | [num]C

1. Non terminal D generates sequence of declarations
2. Non terminal T generates basic, array or record types
3. Non Terminal C generates string of zero or more integer each surrounded by brackets.
4. Array type consists of basic type B followed by C.
5. Record type is sequence of declaration of fields of record.

### Storage Layout For Local Name :

* + From type of name, we can determine amount of storage that will be needed for name at run time
  + At compile time we use space to each name as relative address.
  + Type and relative address are saved in symbol table entry for name.
  + Data of varying length, such as strings or data whose size can’t be determined until run time such as dynamic arrays, reserve known fixed amount storage for pointer to data,  Width of type is number of storage units needed for objects of that type.
  + Basic types like integer, character or float requires integral number of bytes.

## Type checking:

* + Type checker checks whether the correct type is used in source program. It will catch errors in the program.
  + If a language is strongly typed, compiler guarantees for these to run without type errors
  + Type checking improves the security to systems that allow software module to be imported and executed.

### Rules for type checking:

* + Type checking can take on 2 forms

1. Synthesis
2. Inference
   * Type synthesis builds up type of expression from types of its sub expression.
   * Rule for type synthesis has the form:

if f has type S → t and x has type s then expression f(x) has type t.

* + Type inference determines type of the expression from the way it is used.
  + Rule for type inference has the form :

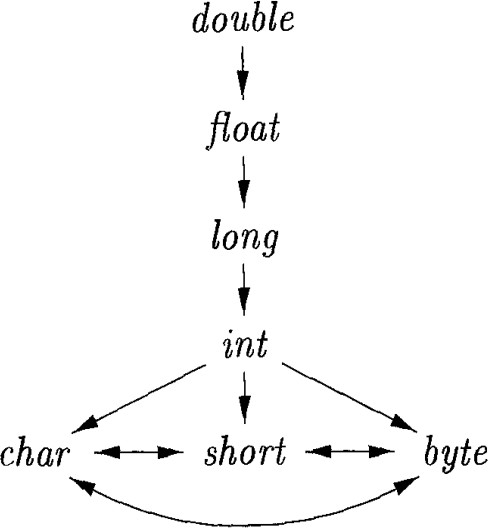
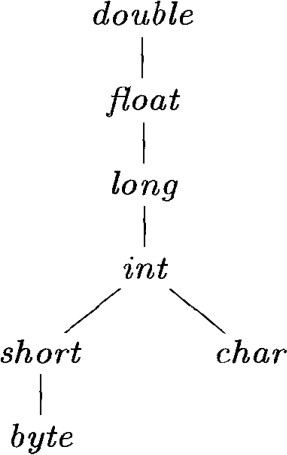
if f(x) is expression

then for some α → β, f has α → β and x has type α

### Type conversions:

* + Type conversion is a way of changing an entity of one data type into another. Example**:** integer 2 is converted to float in code for expression 2 \* 3.14

t1 = (float) 2 t2 = t1 \* 3.14

* + Type conversion are different from one language to another .conversion in Java are categorized into widening conversion and narrowing conversions.

Widening Conversion Narrowing conversion

* + Conversion from one type to another is said to be implicit if it is done automatically by compiler.
  + Implicit type conversions also called coercions. These are limited in many languages to widening conversions.
  + Conversions is said to be explicit if programmer must write something to cause conversion.
  + Explicit conversions are also called casts. Addr widen (Addr a, type t, type w)

{

if (t=w) { return a;

}

else if (t=integer and w=float) { temp = new temp();

gen (temp ‘=’ ‘(float)’a); return temp;

}

else error;

}

### Algorithm for unification:

* + Unification is problem of determining whether 2 expressions s and t can be identical by substituting expressions for variable in s and t.
  + Testing equality of expressions is special case of unification. If s and t are constants then s and t unify if and only if they are identical.
  + In unification types are represented by graphs, type variables are represented by leaves and type constructors are represented by interior nodes.
  + Nodes are grouped into equivalence class if two nodes are in same equivalence class then type expressions they represent must unify.
  + All interior nodes in same class must be for same type constructor and their children must be equivalent.
  + Unification algorithm shown below,

boolean unify(node m, node n)

{

S= find (m); t= find (n); if (s=t) {

return true;

}

else if (nodes s and t represent same basic type) { return true;

}

else if (s is an op-code with children s1 and s2 and t is an op-code with children t1 and t2)

{

union(s, t);

return unify(s1, t1) and unify(s2, t2);

}

else if (s or t represents a variable)

{

union(s, t); return true;

}

else return false;

}

* + Find (n) returns representative node of equivalence class currently containing node n.
  + Union (m, n) merges equivalence classes containing nodes m and n.