MODULE 4

Syntax directed translation:

Syntax directed definitions, Bottom- up evaluation of S-attributed definitions, L- attributed definitions, Top-down translation, Bottom-up evaluation of inherited attributes.

Type Checking:

Type systems, Specification of a simple type checker.

Need for Semantic Analysis

- Semantic analysis is a phase by a compiler that adds semantic information to the parse tree and performs certain checks based on this information.
- It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which (intermediate/target) code is generated. (In a compiler implementation, it may be possible to fold different phases into one pass.)
- Typical examples of semantic information that is added and checked is typing information (type checking) and the binding of variables and function names to their definitions (object binding).
- Sometimes also some early code optimization is done in this phase. For this phase the compiler usually maintains *symbol tables* in which it stores what each symbol (variable names, function names, etc.) refers to.

Following things are done in Semantic Analysis:

1. **Disambiguate Overloaded operators** : If an operator is overloaded, one would like to specify the meaning of that particular operator because from one will go into code generation phase next.

2. **Type checking** : The process of verifying and enforcing the constraints of types is called type checking.

- This may occur either at compile-time (a static check) or run-time (` dynamic check).
- Static type checking is a primary task of the semantic analysis carried out by a compiler.
- If type rules are enforced strongly (that is, generally allowing only those automatic type conversions which do not lose information), the process is called strongly typed, if not, weakly typed.

3. **Uniqueness checking** : Whether a variable name is unique or not, in the its scope.

4. **Type coercion** : If some kind of mixing of types is allowed. Done in languages which are not strongly typed. This can be done dynamically as well as statically.

5. **Name Checks** : Check whether any variable has a name which is not allowed. Ex. Name is same as an identifier(Ex. int in java).

- A parser has its own limitations in catching program errors related to semantics, something that is deeper than syntax analysis.
- Typical features of semantic analysis cannot be modeled using context free grammar formalism.
- If one tries to incorporate those features in the definition of a language then that language doesn't remain context free anymore**.**
- These are a couple of examples which tell us that typically what a compiler has to do beyond syntax analysis.
- An identifier **x** can be declared in two separate functions in the program, once of the type int and then of the type char. Hence the same identifier will have to be bound to these two different properties in the two different contexts.

Semantic Errors

We have mentioned some of the semantics errors that the semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Syntax Directed Translation

The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree. By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.

– **We associate Attributes to the grammar symbols representing the language constructs.**

– **Values for attributes are computed by Semantic Rules associated with grammar productions**.

Evaluation of Semantic Rules may:

– Generate Code;

– Insert information into the Symbol Table;

– Perform Semantic Check;

– Issue error messages;

– etc.

- There are two ways to represent the semantic rules associated with grammar symbols.

1. Syntax-Directed Definitions (SDD)
2. Syntax-Directed Translation Schemes (SDT)

Syntax Directed Definitions

Syntax Directed Definitions are a generalization of context-free grammars in which:

1. Grammar symbols have an associated set of Attributes;

2. Productions are associated with Semantic Rules for computing the values of attributes.

• Such formalism generates Annotated Parse-Trees where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).

- The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

ATTRIBUTE GRAMMAR

- Attributes are properties associated with grammar symbols. Attributes can be numbers, strings, memory locations, datatypes, etc.

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.

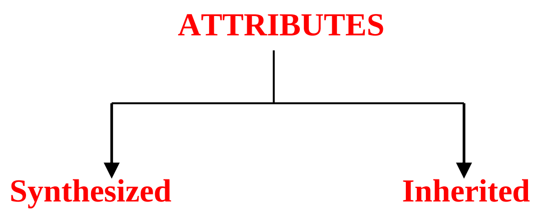
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

```
Ex:   E → E + T { E.value = E.value + T.value }
```

- The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E.

- Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions.

- Based on the way the attributes get their values, they can be broadly divided into two categories : synthesized attributes and inherited attributes



1. **Synthesized Attributes:** These are those attributes which get their values from their children nodes i.e. value of synthesized attribute at node is computed from the values of attributes at children nodes in parse tree.

- To illustrate, assume the following production:

EXAMPLE : S -> ABC

S.a= A.a,B.a,C.a

If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

Computation of Synthesized Attributes

- Write the SDD using appppropriate semantic rules for each production in given grammar.
- The annotated parse tree is generated and attribute values are computed in bottom up manner.
- The value obtained at root node is the final output.

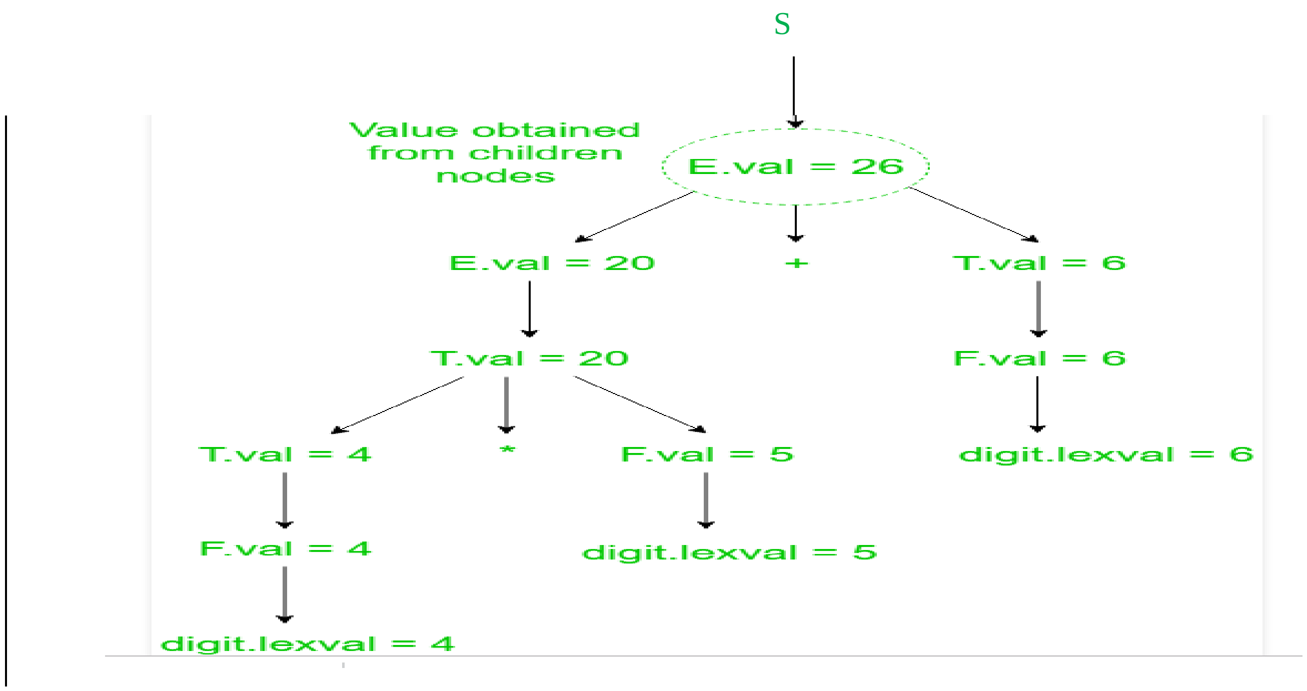
Consider the following grammar:

```
S --> E
E --> E₁ + T
E --> T
T --> T₁ * F
T --> F
F --> digit
```

The SDD for the above grammar can be written as follow

PRODUCTIONS	**SEMANTIC RULES**
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is



- For computation of attributes we start from leftmost bottom node. The rule F –> digit is used to reduce digit to F and the value of digit is obtained from lexical analyzer which becomes value of F i.e. from semantic action F.val = digit.lexval.
- Hence, F.val = 4 and since T is parent node of F so, we get T.val = 4 from semantic action T.val = F.val.
- Then, for T –> T_1 * F production, the corresponding semantic action is T.val = T_1.val * F.val . Hence, T.val = 4 * 5 = 20
- Similarly, combination of E_1.val + T.val becomes E.val i.e. E.val = E_1.val + T.val = 26. Then, the production S –> E is applied to reduce E.val = 26 and semantic action associated with it prints the result E.val . Hence, the output will be 26.

Annotated parse tree

- The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

2. **Inherited Attributes**: These are the attributes which inherit their values from their parent or sibling nodes. i.e. value of inherited attributes are computed by value of parent or sibling nodes.

EXAMPLE:

```
A --> BCD   { C.in = A.in, C.type = B.type }
```

B can get values from A, C and D. C can take values from A, B, and D. Likewise, D can take values from A, B, and C.

Computation of Inherited Attributes

- Construct the SDD using semantic actions.
- The annotated parse tree is generated and attribute values are computed in top down manner.

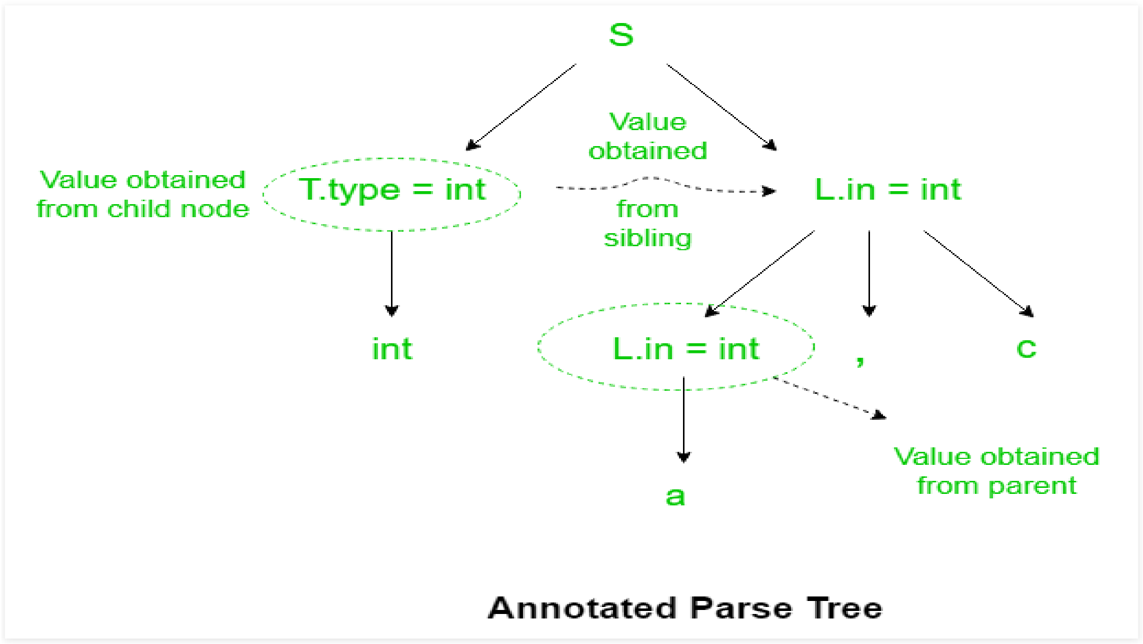
Consider the following grammar:

```
D --> T L
T --> int
T --> float
T --> double
L --> L₁, id
L --> id
```

The SDD for the above grammar can be written as follow

PRODUCTIONS	SEMANTIC RULES
D→TL	**L.in==T.type**
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

➢ Let us assume an input string **int a, c** for computing inherited attributes. The annotated parse tree for the input string is


S
Value obtained from child node
T.type = int
Value obtained from sibling
L.in = int
int
L.in = int
,
c
a
Value obtained from parent

Annotated Parse Tree

➢ The value of L nodes is obtained from T.type (sibling) which is basically lexical value obtained as int, float or double.

➢ Then L node gives type of identifiers a and c. The computation of type is done in top down manner or preorder traversal.

➢ Using function Enter_type the type of identifiers a and c is inserted in symbol table at corresponding id.entry.

Implementing Syntax Directed Definitions

1. **Dependency Graphs**

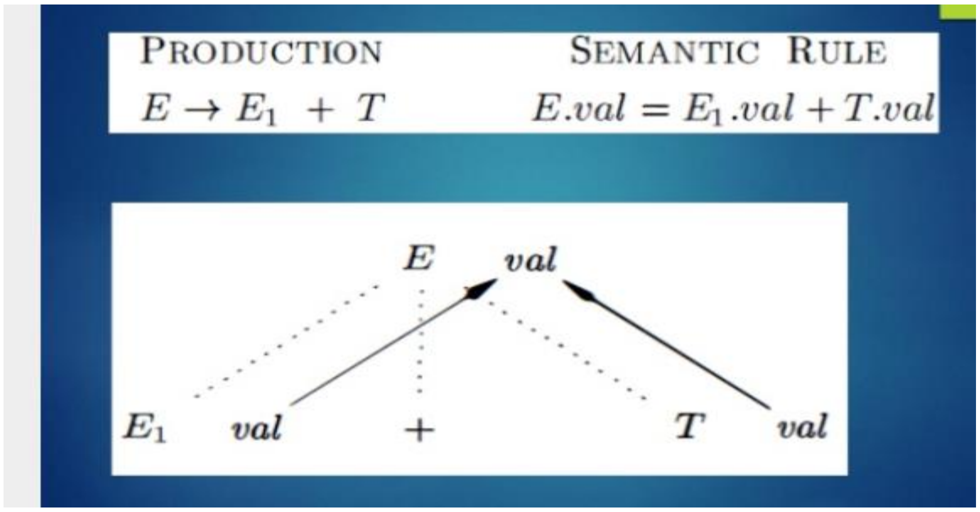
➢ Implementing a Syntax Directed Definition consists primarily in finding an order for the evaluation of attributes

- Each attribute value must be available when a computation is performed.

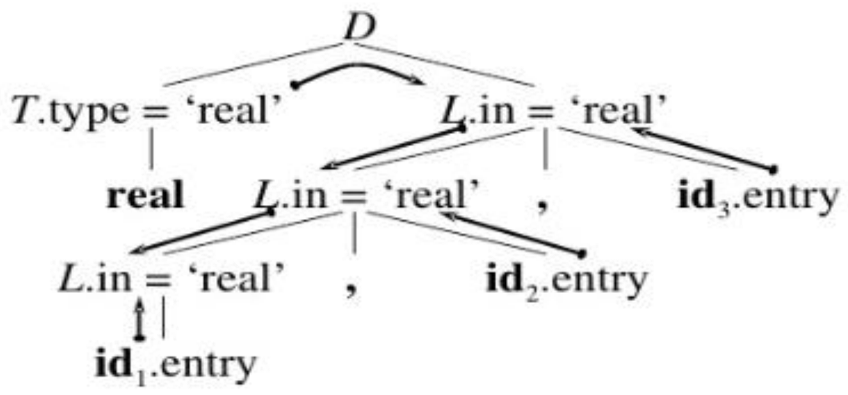
➢ Dependency Graphs are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.

➢ Annotated parse tree shows the values of attributes, dependency graph helps to determine how those values are computed

- The interdependencies among the attributes of the various nodes of a parse-tree can be depicted by a directed graph called a dependency graph.
    - **There is a node for each attribute;**
    - **If attribute b depends on an attribute c there is a link from the node for c to the node for b (b ← c).**
- Dependency Rule: If an attribute b depends from an attribute c, then we need to find the semantic rule for c first and then the semantic rule for b.



PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$



Dependency graph for declaration statements.

2. Evaluation order

- A dependency graph characterizes the possible order in which we can calculate the attributes at various nodes of the parse tree.
- If there is an edge from node M to N, then the attribute corresponding to M first be evaluated before evaluating N.
- Thus the allowable orders of evaluation are N1,N2…..,Nk such that if there is an edge from Ni toNj then i<j
- Such an ordering embeds a directed graph into a linear order, and is called a **topological sort** of the graph.
- If there is any cycle in the graph ,then there is no topologicalsort.ie, there is no way to evaluate SDD on this parse tree.

TYPES OF SDT'S

1. **S –attributed definition**
2. **L –attributed definition**

S-attributed definition

- S stands for synthesized
- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.

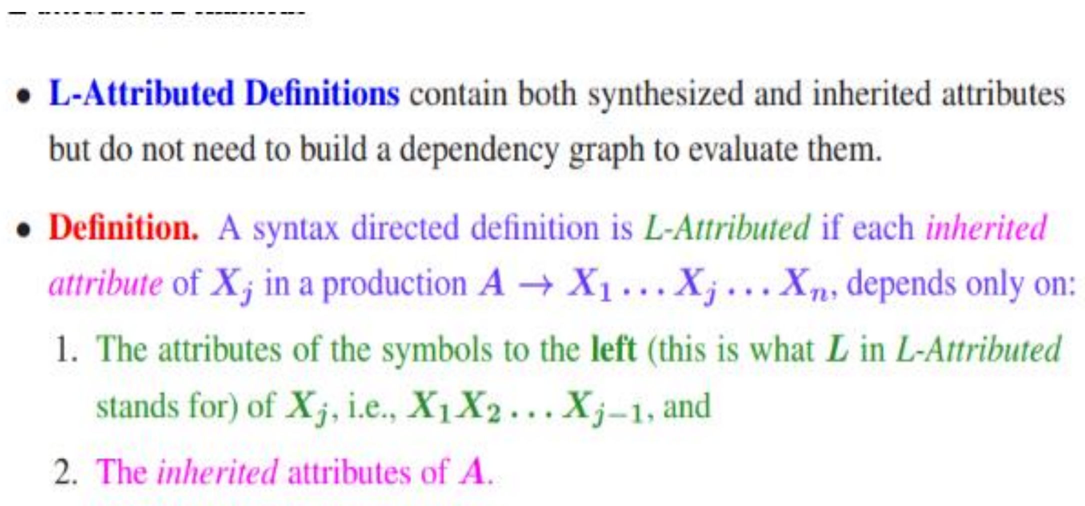
  EXAMPLE:

  A→BC { A.a=B.a,C.a}
- S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

  A→BCD{ }.
- **Note:** (Also write SDD for desk calculator as example).

L –attributed definition

- L stands for one parse from left to right.



- **L-Attributed Definitions** contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.
- **Definition.** A syntax directed definition is *L-Attributed* if each *inherited attribute* of X_j in a production $A \to X_1 \ldots X_j \ldots X_n$, depends only on:
  1. The attributes of the symbols to the **left** (this is what L in *L-Attributed* stands for) of X_j, i.e., $X_1 X_2 \ldots X_{j-1}$, and
  2. The *inherited* attributes of A.

- Ie, If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from parent and left siblings only, it is called as L-attributed SDT.

  EXAMPLE:

  A →BCD {B.a=A.a, C.a=B.a}

  C.a=D.a → This is not possible

- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.

  A→{ }BC

  B{ }C

  BC{ }

- **Note:** (Also write SDD for declaration statement as example)

If an attribute is S attributed , it is also L attributed.

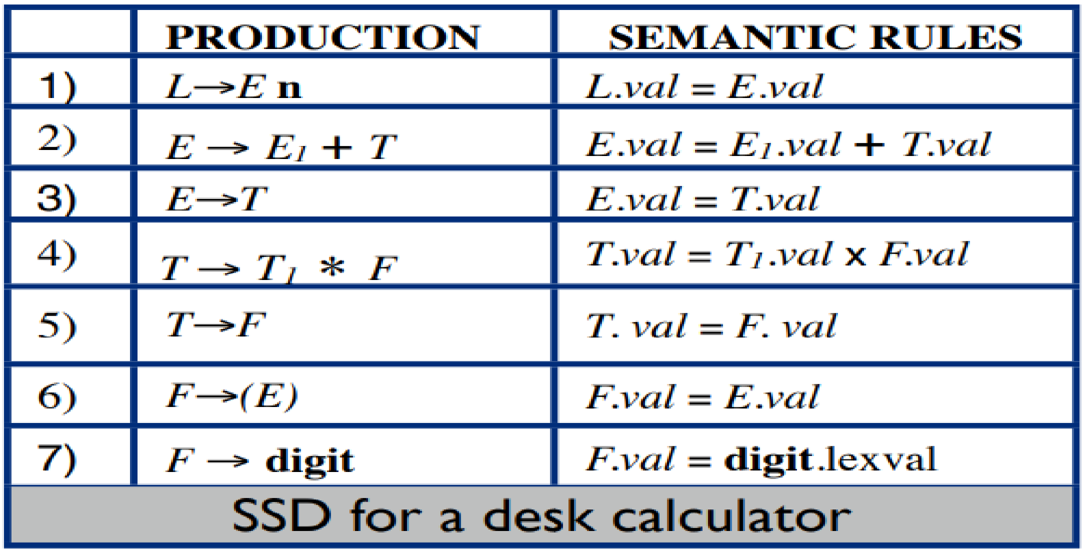
Evaluation of L-attributed SDD

- **L-Attributed Definitions are a class of syntax directed definitions whose attributes can always be evaluated by single traversal of the parse-tree.**
- The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

Algorithm: L-Eval(n: Node)
Input: Node of an annotated parse-tree.
Output: Attribute evaluation.

```
Begin
    For each child $m$ of $n$, from left-to-right Do
    Begin
        Evaluate inherited attributes of $m$;
        L-Eval(m)
    End;
    Evaluate synthesized attributes of $n$
End.
```

SDD for desk calculator/SDD for evaluation of expressions



	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

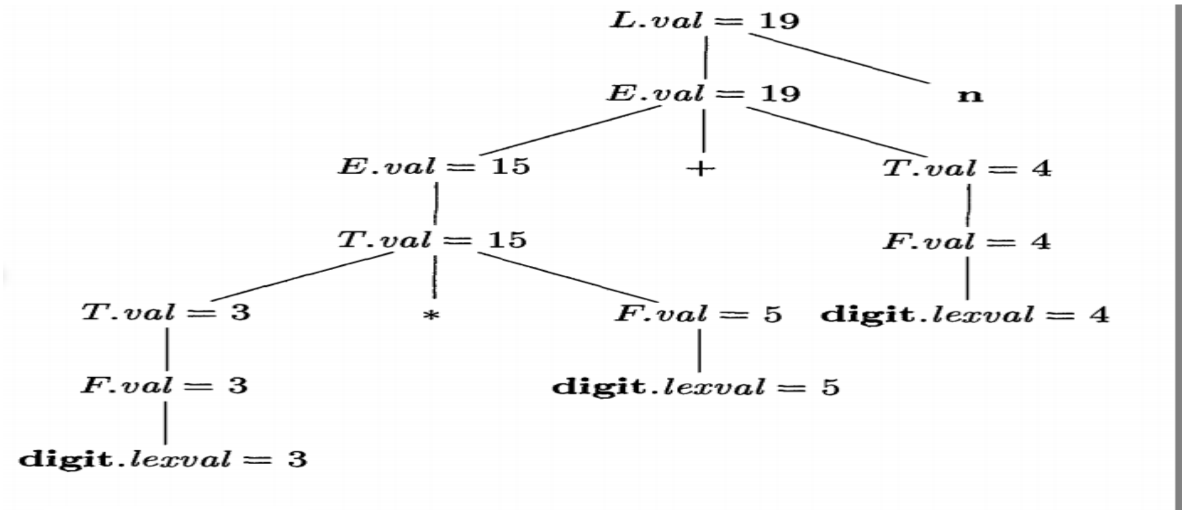
SSD for a desk calculator

- **Evaluate the expression 3*5+4n using the above SDD both in bottom up and top down approach**

Solution: Bottom up evaluation for this expression is shown below

- In both case first we need to draw the parse tree.

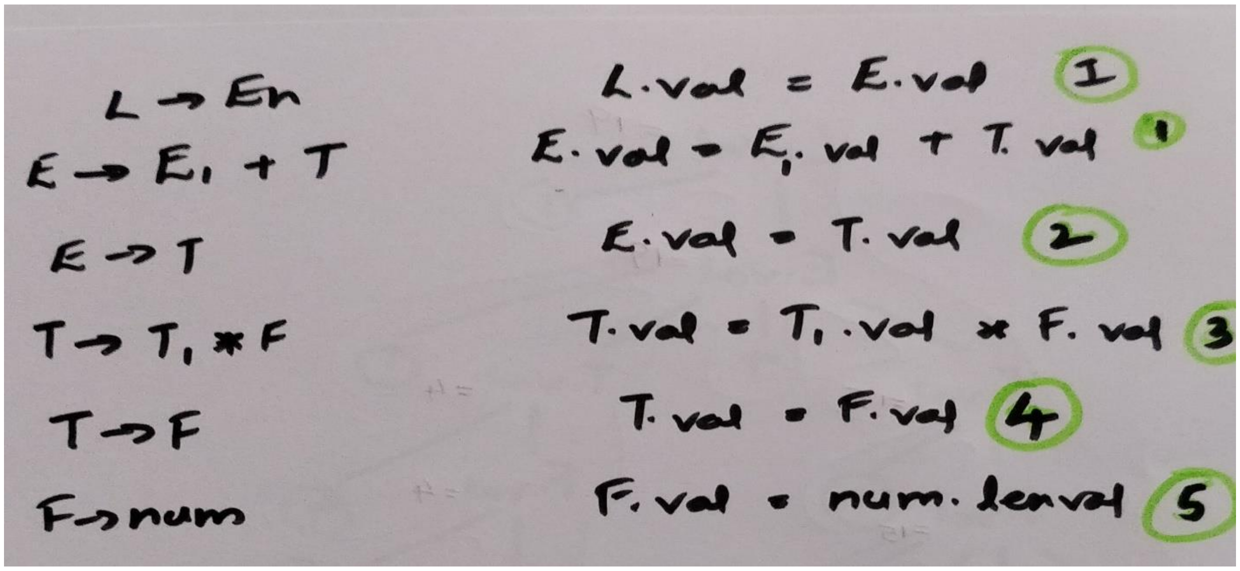
- Then traverse from top to down left to right
- In bottom up approach, whenever there is a reduction ,go to the production and carry out the action



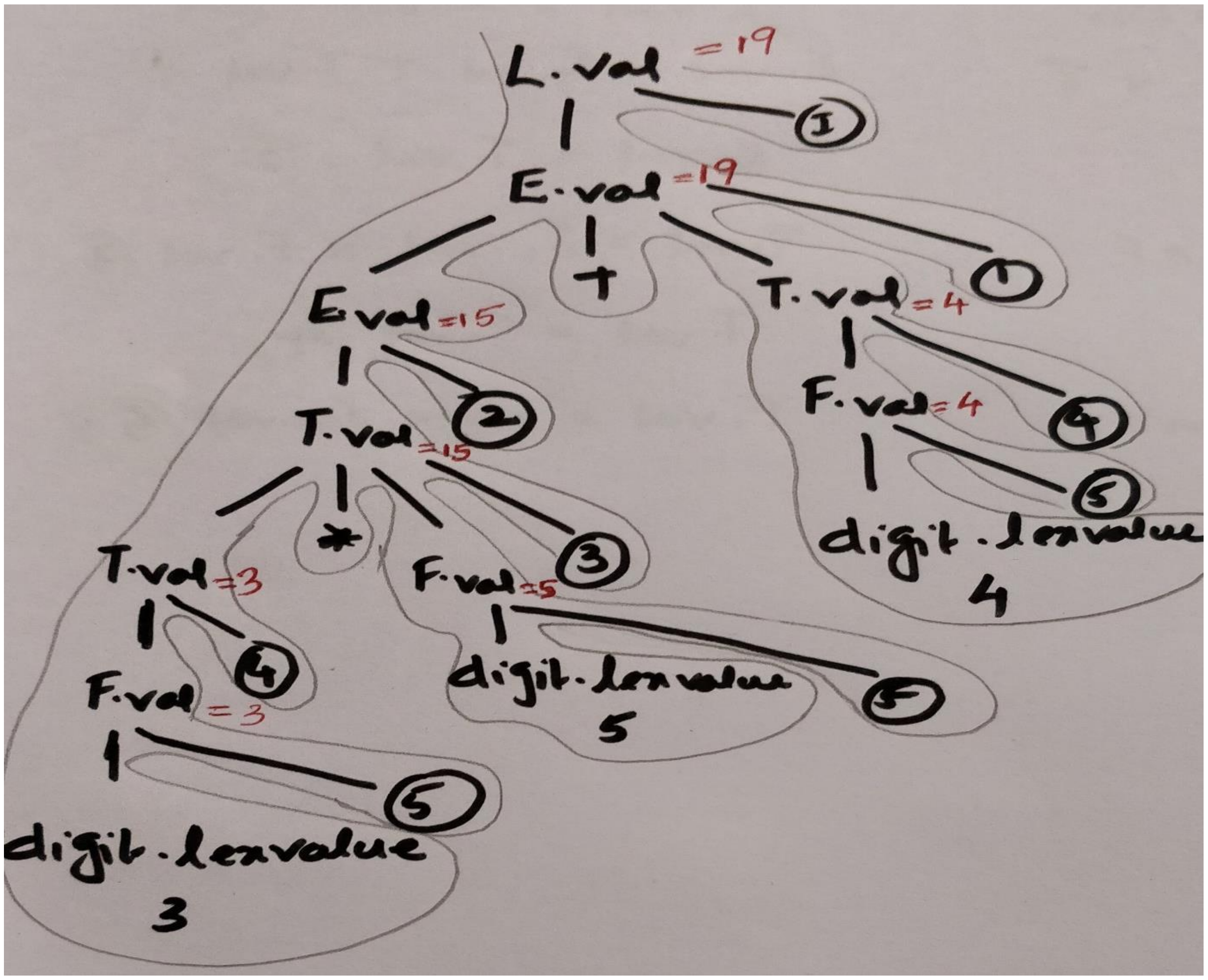
Annotated parse tree for $3 * 5 + 4\ \mathbf{n}$

Solution: Top down approach:

- First we need to draw the parse tree.
- Then traverse from top to down left to right
- In Top down up approach, whenever we come across a semantic action, it is performed.



$L \rightarrow En$	$L.val = E.val$	1
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$	1
$E \rightarrow T$	$E.val = T.val$	2
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$	3
$T \rightarrow F$	$T.val = F.val$	4
$F \rightarrow num$	$F.val = num.lexval$	5



Workout problems

1. Evaluate the expression 4-6/2
2. Evaluate the expression 2+ (3*4)

Traversal

```
Procedure  visit (n:node)
  begin
 for each child m of n, from left to right do
visit(m)
```

evaluate semantic rule at node n

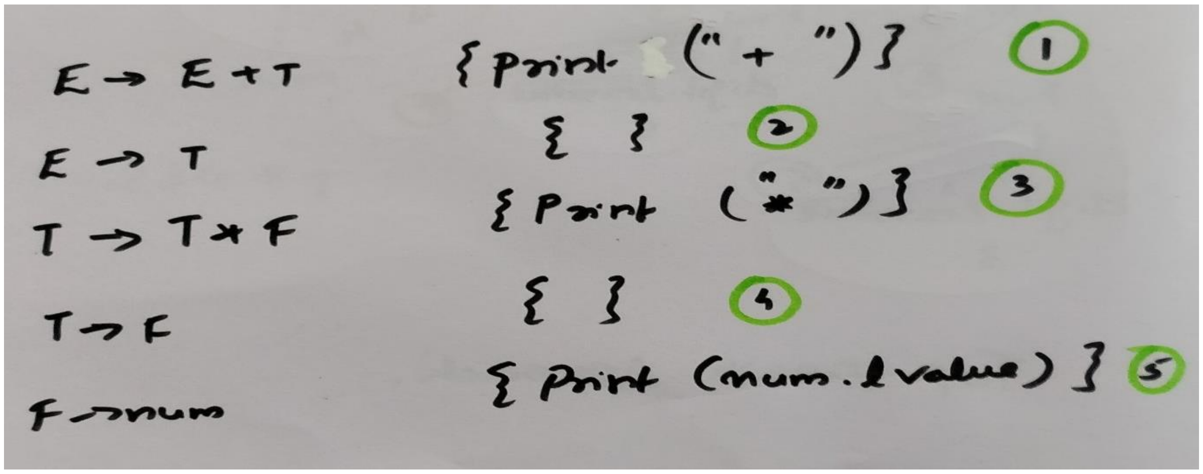
end

Syntax Directed Translation Schemes (SDT'S)

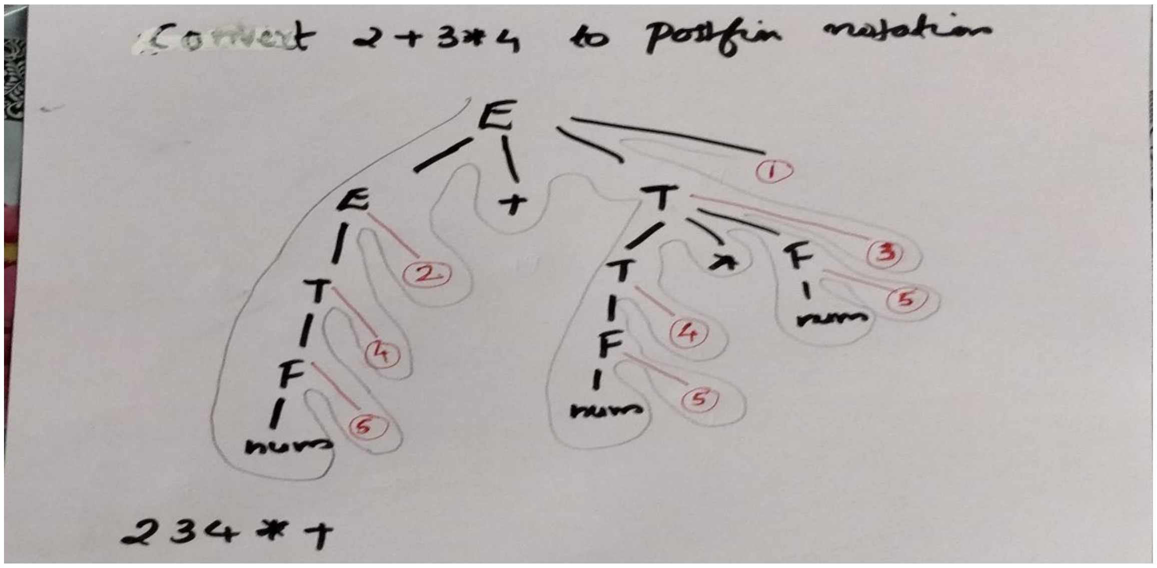
- It is a CFG with program fragments embedded with the production body.
- The program fragments are called semantic actions

SDD for infix to postfix translation

GRAMMAR **SEMANTIC ACTIONS**



$E \rightarrow E + T$	{print("+")}	①
$E \rightarrow T$	{ }	②
$T \rightarrow T * F$	{Print("*")}	③
$T \rightarrow F$	{ }	④
$F \rightarrow num$	{Print(num.lvalue)}	⑤



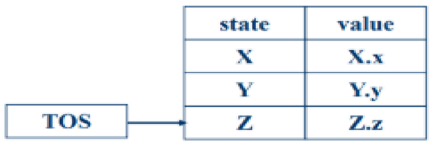
Convert 2+3*4 to postfix notation

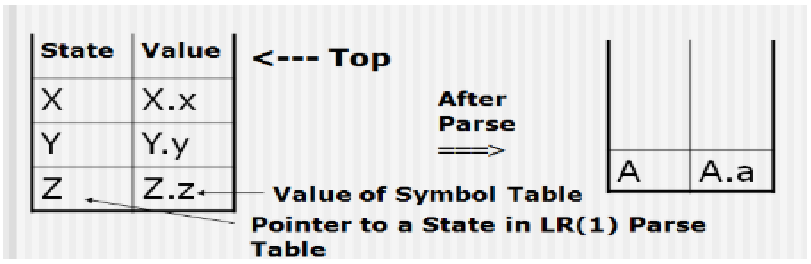
234*+

Bottom up evaluation of S-attributed definition

- Syntax-directed definitions with only synthesized attributes(S- attribute) can be evaluated by a bottom up parser (BUP) as input is parsed
- In this approach, the parser will keep the values of synthesized attributes associated with the grammar symbol on its stack.
- The stack is implemented as a pair of state and value.
- When a reduction is made ,the values of the synthesized attributes are computed from the attribute appearing on the stack for the grammar symbols
- implementation is by using an LR parser (e.g. YACC)

• **e.g. A => XYZ and A.a := f (X.x, Y.y, Z.z)**

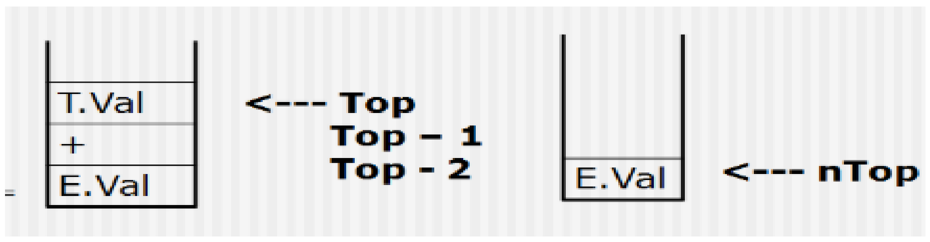




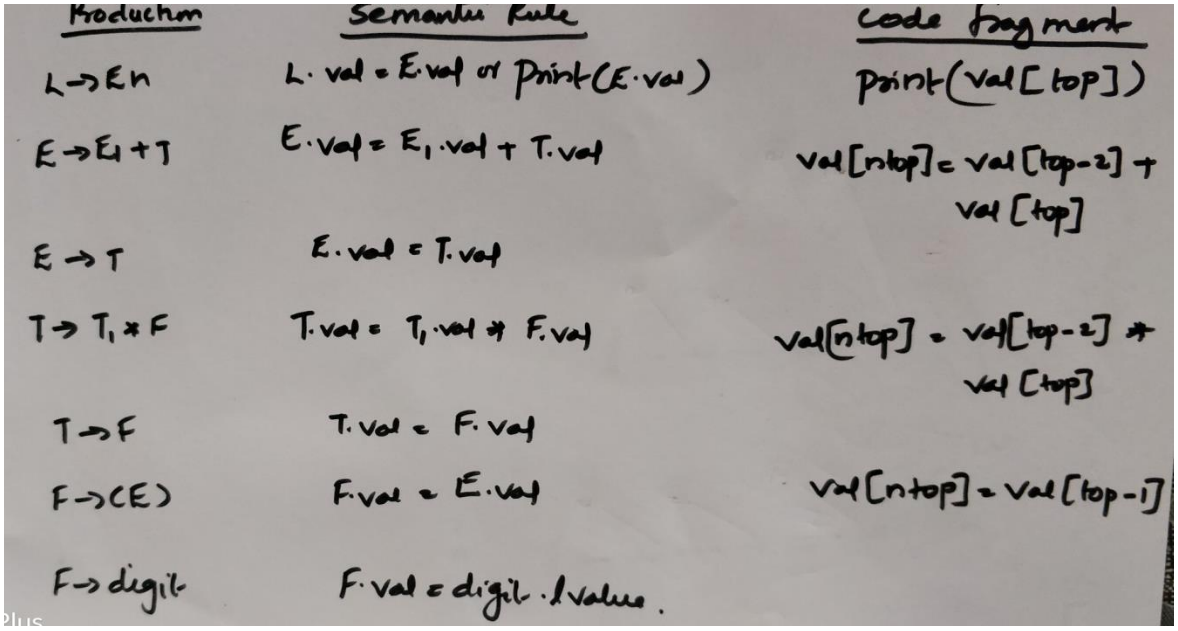
Consider again the syntax-directed definition of the desk calculator.

PRODUCTION	CODE FRAGMENT
$L \rightarrow E$ **n**	*print* (*val* [*top*])
$E \rightarrow E_1 + T$	*val* [*ntop*] := *val* [*top* − 2] + *val* [*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val* [*ntop*] := *val* [*top* − 2] × *val* [*top*]
$T \rightarrow F$	
$F \rightarrow (E)$	*val* [*ntop*] := *val* [*top* − 1]
$F \rightarrow$ **digit**	

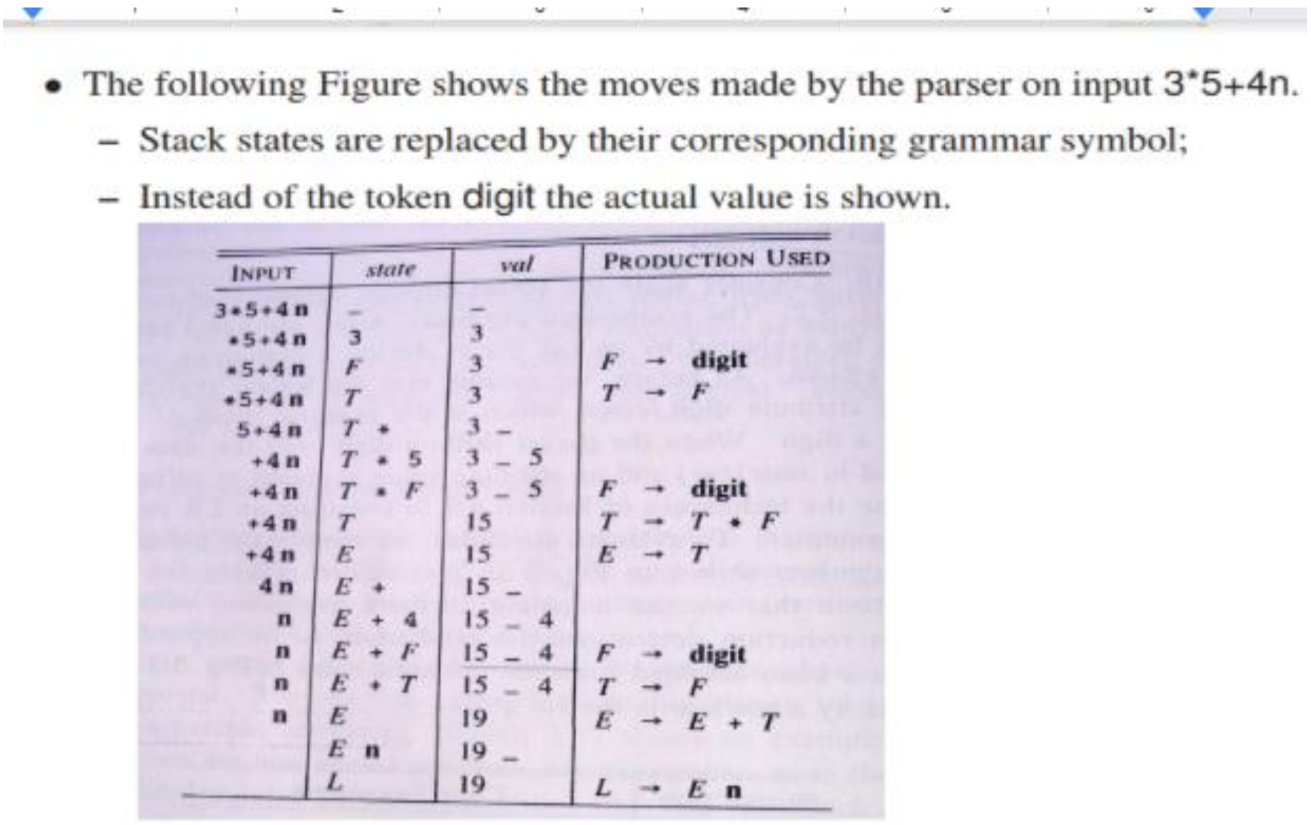
Fig. 5.16. Implementation of a desk calculator with an LR parser.



Example :- SDD and code fragment using S attributed definition for the input 3*5+4n is as follows:



Production	Semantic Rule	code fragment
L->En	L.val = E.val or Print(E.val)	print(val[top])
E->E1+T	E.val = E1.val + T.val	val[ntop] = val[top-2] + val[top]
E->T	E.val = T.val	
T->T1*F	T.val = T1.val * F.val	val[ntop] = val[top-2] * val[top]
T->F	T.val = F.val	
F->(E)	F.val = E.val	val[ntop] = val[top-1]
F->digit	F.val = digit.lvalue.	



- The following Figure shows the moves made by the parser on input 3*5+4n.
  - Stack states are replaced by their corresponding grammar symbol;
  - Instead of the token digit the actual value is shown.

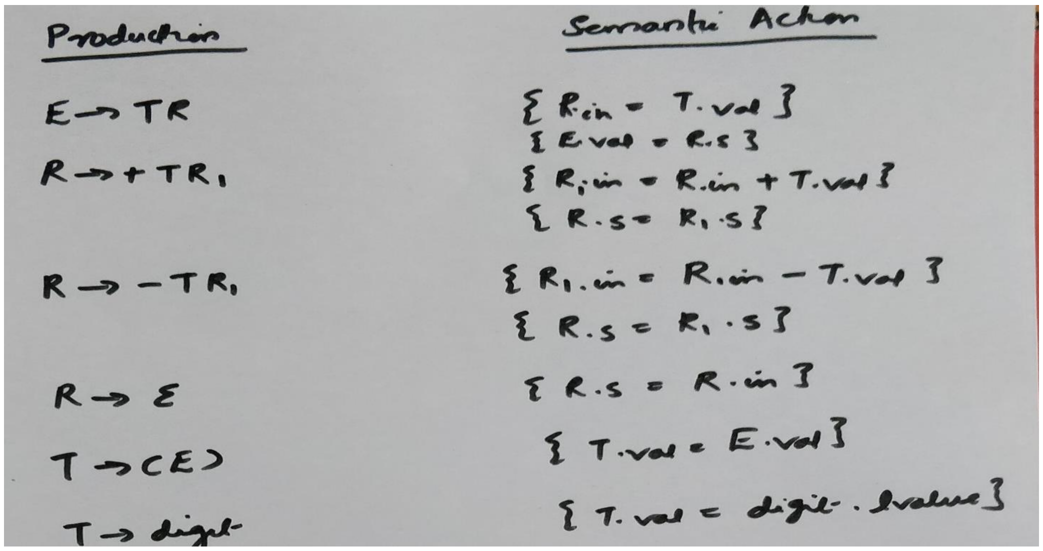
INPUT	state	val	PRODUCTION USED
3*5+4n	–	–	
*5+4n	3	3	
*5+4n	F	3	F → digit
*5+4n	T	3	T → F
5+4n	T *	3 _	
+4n	T * 5	3 _ 5	
+4n	T * F	3 _ 5	F → digit
+4n	T	15	T → T * F
+4n	E	15	E → T
4n	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	F → digit
n	E + T	15 _ 4	T → F
n	E	19	E → E + T
	E n	19 _	
	L	19	L → E n

Top Down Translation

- implementation of L-attribute definitions during predictive parsing using left recursion grammars and left recursion elimination algorithms
- e.g. translation schema with left recursive grammar

E	=>	E_1 + T	{ E.val := E_1.val + T.val }
E	=>	E_1 - T	{ E.val := E_1.val - T.val }
E	=>	T	{ E.val := T.val }
T	=>	(E)	{ T.val := E.val }
T	=>	num	{ T.val := num.val }

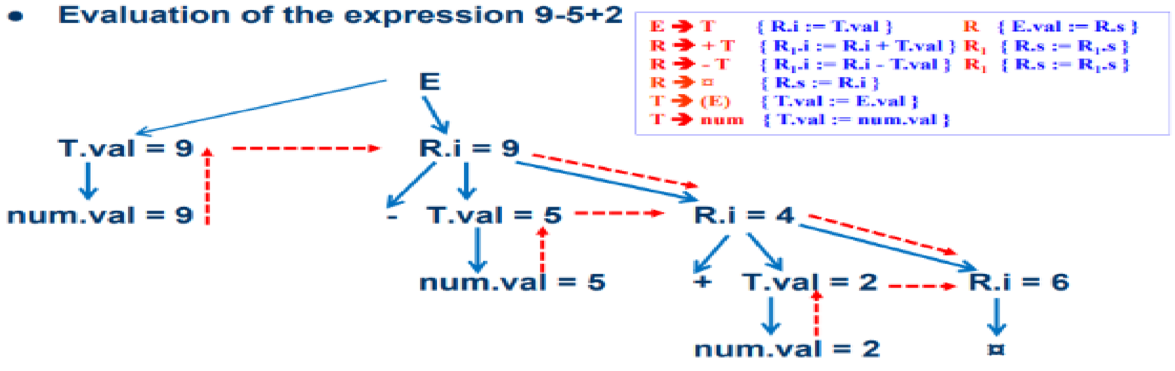
Do left recursion for this grammar



Production	Semantic Action
E→TR	{ R.in = T.val } { E.val = R.s }
R→+TR$_1$	{ R$_1$.in = R.in + T.val } { R.s = R$_1$.s }
R→ −TR$_1$	{ R$_1$.in = R.in − T.val } { R.s = R$_1$.s }
R→ ε	{ R.s = R.in }
T→(E)	{ T.val = E.val }
T→ digit	{ T. val = digit . lvalue }

Here R represents E', i for inheritance, s for synthesised attributes.

- Evaluation of the expression 9-5+2



E → T	{ R.i := T.val }	R	{ E.val := R.s }
R → + T	{ R$_1$.i := R.i + T.val }	R$_1$	{ R.s := R$_1$.s }
R → - T	{ R$_1$.i := R.i - T.val }	R$_1$	{ R.s := R$_1$.s }
R → ¤	{ R.s := R.i }		
T → (E)	{ T.val := E.val }		
T → num	{ T.val := num.val }		

5.1 RUN-TIME ENVIRONMENTS

A translation needs to relate the static source text of a program to the dynamic actions that must occur at runtime to implement the program. The program consists of names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.

Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.

5.1.1 SOURCE LANGUAGE ISSUES

Procedure

A ***procedure*** definition is a declaration that associates an identifier with a statement. The identifier is ***procedure*** name, and statement is the ***procedure*** body.

For example, the following definition of procedure named ***readarray***

```
procedure readarray;
var i : integer;
begin
    for i : = 1 to 9 do read(a[i])
end;
```

When a procedure name appears with in an executable statement, the procedure is said to be ***called*** at that point.

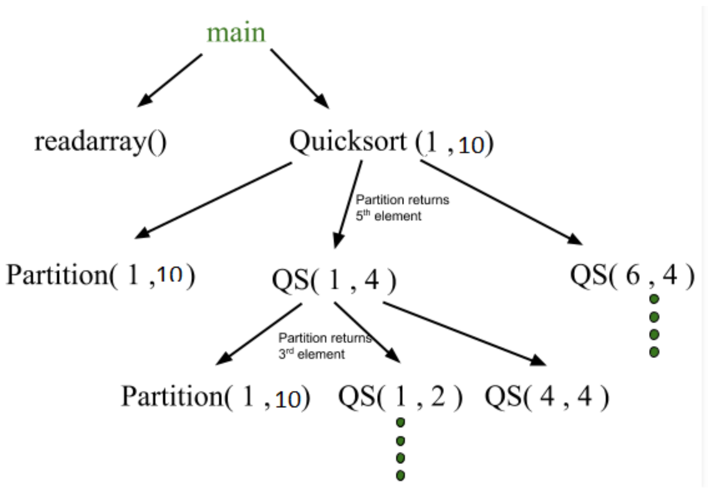
Activation Tree

- Each execution of procedure is referred to as an activation of the procedure. Lifetime of an activation is the sequence of steps present in the execution of the procedure.
- If 'a' and 'b' be two procedures, then their activations will be non-overlapping (when one is called after other) or nested (nested procedures).
- A procedure is recursive if a new activation begins before an earlier activation of the same procedure has ended. An activation tree shows the way control enters and leaves, activations.
- Properties of activation trees are :-
    - Each node represents an activation of a procedure.
    - The root shows the activation of the main function.
    - The node for procedure 'x' is the parent of node for procedure 'y' if and only if the control flows from procedure x to procedure y.

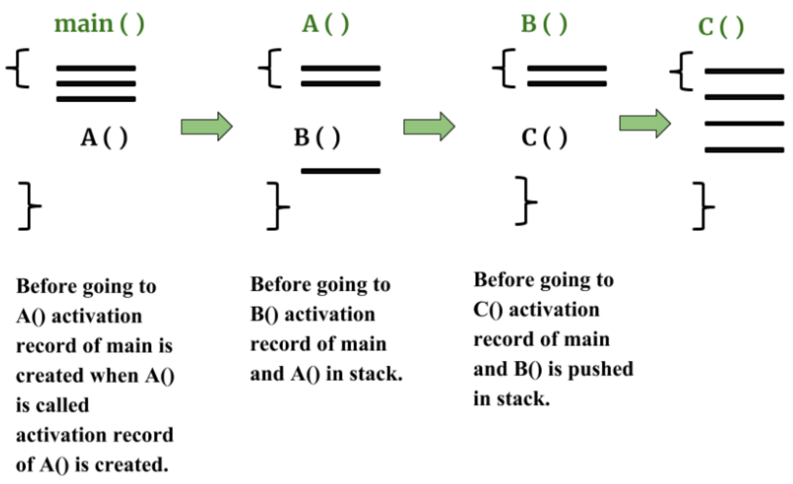
EXAMPLE

Consider the following program of quicksort

```
main()
{
        readarray();
        quicksort(1,10);
}
quicksort(int m, int n)
{
        int i= partition(m,n);
        quicksort(m,i-1);
        quicksort(i+1,n);
}
```



- First main function as root then main calls readarray and quicksort.
- Quicksort in turn calls partition and quicksort again. The flow of control in a program corresponds to the depth first traversal of activation tree which starts at the root.


main ( )
A ( )
A ( )
B ( )
B ( )
C ( )
C ( )
Before going to A() activation record of main is created when A() is called activation record of A() is created.
Before going to B() activation record of main and A() in stack.
Before going to C() activation record of main and B() is pushed in stack.

Control Stack

- Control stack or runtime stack is used to keep track of the live procedure activations i.e the procedures whose execution have not been completed.
- A procedure name is pushed on to the stack when it is called (activation begins) and it is popped when it returns (activation ends).
- Information needed by a single execution of a procedure is managed using an activation record.
- When a procedure is called, an activation record is pushed into the stack and as soon as the control returns to the caller function the activation record is popped on as the control turns to the caller function the activation record is popped.
- Then the contents of the control stack are related to paths to the root of the activation tree. When node n is at the top of the control stack, the stack contains the nodes along the path from n to the root.
- Consider the above activation tree, when quicksort(4,4) gets executed, the contents of control stack were main() quicksort(1,10) quicksort(1,4), quicksort(4,4)

quicksort(4,4)
Quicksort(1,4)
Quicksort(1.10)
Main()

The Scope Of Declaration

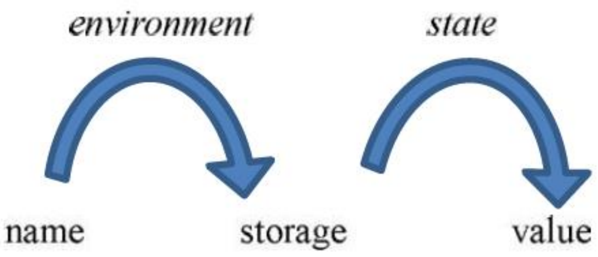
- A declaration is a syntactic construct that associates information with a name. Declaration may be explicit such as

  var i : integer;

  or may be explicit. The portion of program to which a declaration applies is called the **scope** of that declaration.

Binding Of Names

- Even if each name is declared once in a program, the same name may denote different data object at run time. “Data objects” corresponds to a storage location that hold values.
- The term ***environment*** refers to a function that maps a name to a storage location.
- The term ***state*** refers to a function that maps a storage location to the value held there.

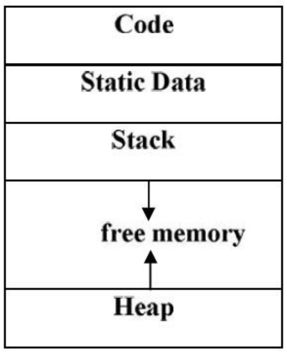


- When an environment associates storage location ***s*** with a name ***x,*** we say that ***x*** is bounds to ***s.*** This association is referred to as a binding of ***x.***

5.1.2 STORAGE ORGANIZATION

- The executing target program runs in its own logical address space in which each program value has a location
- The management and organization of this logical address space is shared between the compiler, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread through memory.

Typical subdivision of run time memory.



- **Code area**: used to store the generated executable instructions, memory locations for the code are determined at compile time
- **Static Data Area**: Is the locations of data that can be determined at compile time
- **Stack Area**: Used to store the data object allocated at runtime. eg. Activation records
- **Heap**: Used to store other dynamically allocated data objects at runtime (for ex: malloac)
- This runtime storage can be subdivided to hold the different components of an existing system

    1. Generated executable code
    2. Static data objects
    3. Dynamic data objects-heap
    4. Automatic data objects-stack

Activation Records

- It is LIFO structure used to hold information about each instantiation.
- Procedure calls and returns are usually managed by a run time stack called control stack.
- Each live activation has an activation record on control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack
- The contents of the activation record vary with the language being implemented.
- The diagram below shows the contents of an activation record.
- The purpose of the fields of an activation record is as follows, starting from the field for temporaries.

    1. Temporary values, such as those arising in the evaluation of expressions, are stored in the field for temporaries.
    2. The field for local data holds data that is local to an execution of a procedure.
    3. The field for saved machine status holds information about the state of the machine just before the procedure is called. This information includes the values of the program counter and machine registers that have to be restored when control returns from the procedure.
    4. The optional access link is used to refer to nonlocal data held in other activation records.
    5. The optional control /ink paints to the activation record of the caller
    6. The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.
    7. The field for the returned value is used by the called procedure to return a value to the calling procedure, Again, in practice this value is often returned in a register for greater efficiency.

Returned value
Actual parameters
Optional control link
Optional access link
Saved machine status
Local data
temporaries

General Activation Record

5.1.3 STORAGE ALLOCATION STRATEGIES

- The different storage allocation strategies are:

  Static allocation - lays out storage for all data objects at compile time

  Stack allocation - manages the run-time storage as a stack.

  Heap allocation - allocates and deallocates storage as needed at run time from a data area known as heap.

Static Allocation

- In static allocation, names bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at runtime, every time a procedure activated, its run-time, names bounded to the same storage location.
- Therefore, values of local names retained across activations of a procedure. That is when control returns to a procedure the value of the local are the same as they were when control left the last time.
- From the type of a name, the compiler decides amount of storage for the name and decides where the activation records go. At compile time, we can fill in the address at which the target code can find the data it operates on.

Stack Allocation

- All compilers for languages that use procedures, functions or methods as units of user functions define actions manage at least part of their runtime memory as a stack run-time stack.
- Each time a procedure called, space for its local variables is pushed onto a stack, and when the procedure terminates, space popped off from the stack

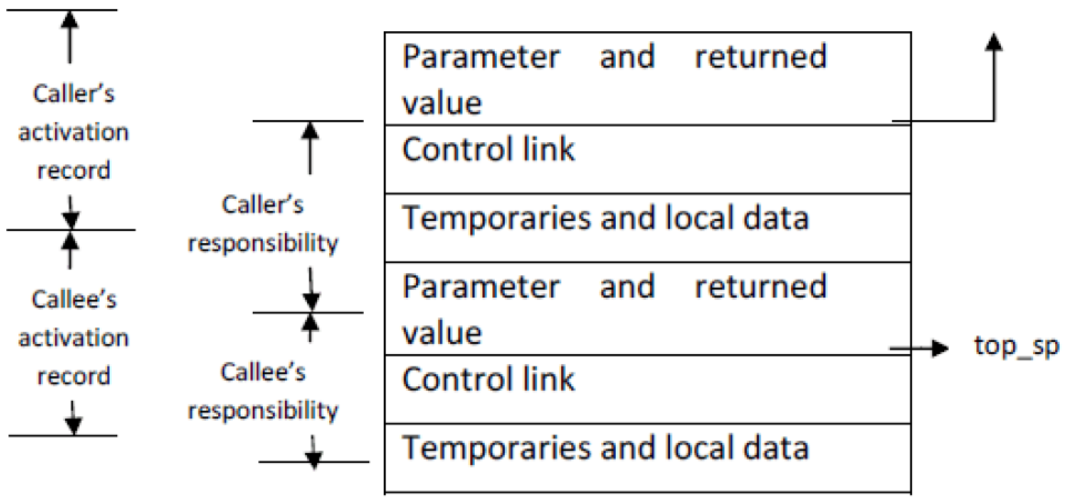
Calling Sequences

- Procedures called implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of a machine so the calling procedure can continue its execution after the call.
- The code is calling sequence of often divided between the calling procedure (caller) and a procedure is calls (callee)(callee).
- When designing calling sequences and the layout of activation record, the following principles are helpful:

1. Value communicated between caller and callee generally placed at the caller beginning of the callee's activation record, so they as close as possible to the caller's activation record.
2. Fixed length items generally placed in the middle. Such items typically include the control link, the access link, and the machine status field.
3. Items whose size may not be known early enough placed at the end of the activation record.
4. We must locate the top of the stack pointer judiciously. A common approach is to have it point to the end of fixed length fields in the activation is to have it point to fix the end of fixed length fields in the activation record. Fixed length data can then be accessed by fixed offsets, known to the **intermediate code generator**, relative to the top of the stack pointer.

The calling sequence and its division between caller and callee are as follows:

1. The caller evaluates the actual parameters.
2. The caller stores a return address and the old value of top_sp into the callee's activation record. The caller then increments the top_sp to the respective positions.
3. The callee-saves the register values and other status information.
4. The callee initializes its local data and begins execution.

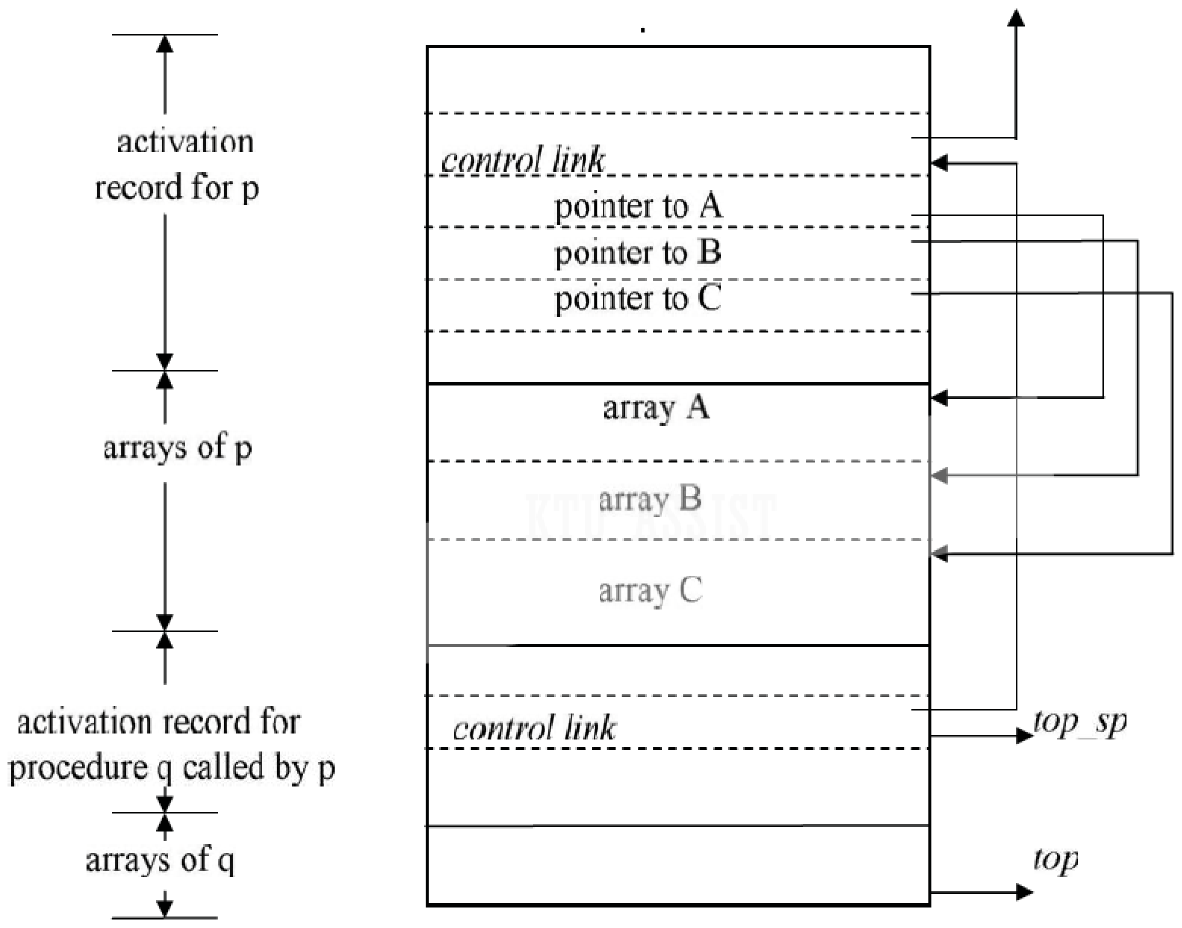

Caller's activation record
Callee's activation record
Caller's responsibility
Callee's responsibility
Parameter and returned value
Control link
Temporaries and local data
Parameter and returned value
Control link
Temporaries and local data
top_sp

A suitable, corresponding return sequence is:

1. The callee places the return value next to the parameters.
2. Using the information in the machine status field, the callee restores top_sp and other registers, and then branches to the return address that the caller placed in the status field.
3. Although top_sp has been decremented, the caller knows where the return value is, relative to the current value of top_sp; the caller, therefore, may use that value.

Variable length data on the stack

- The run-time memory-management system must deal frequently with the allocation of space for **objects the sizes**of which **are not known** at compile time, but which are local to a procedure and thus **may be allocated on the stack**.
- In modern languages, objects whose **size cannot be determined** at **compile time** are **allocated** space **in the heap**
- **However**, it is also **possible to allocate objects, arrays, or other structures of unknown size on the stack**.
- We **avoid** the **expense of garbage collecting** their space. Note that the stack can be used only for an object if it is local to a procedure and **becomes inaccessible** when **the procedure returns**.
- A common strategy for allocating variable-length arrays is shown in following figure



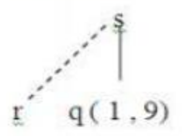
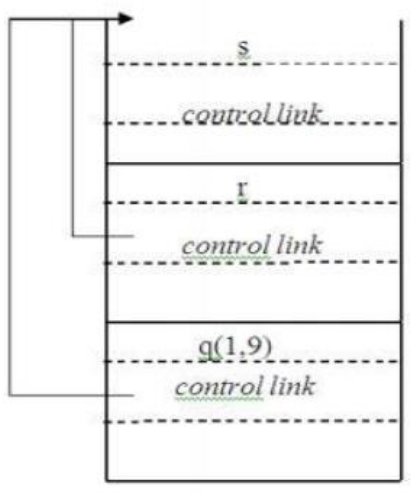
Access to dynamically allocated arrays

Dangling Reference (Storage allocation strategies)

- Whenever storage allocated, the problem of dangling reference arises. The dangling reference occurs when there is a reference to storage that has been allocated.
- It is a logical error to u dangling reference, since, the value of de use de-allocated storage is undefined according to the semantics of most languages.
- Whenever storage allocated, the problem of dangling reference arises. The dangling reference occurs when there is a reference to storage that has been allocated.

Heap Allocation

- Stack allocation strategy cannot be used if either of the following is possible :
    1. The values of local names must be retained when an activation ends.
    2. A called activation outlives the caller.
- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the activation tree	Activation records in the heap	Remarks
	 s control link r control link q(1,9) control link	Retained activation record for r

Records for live activations need not be adjacent in heap

- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

5.2 INTERMEDIATE CODE GENERATION (ICG)

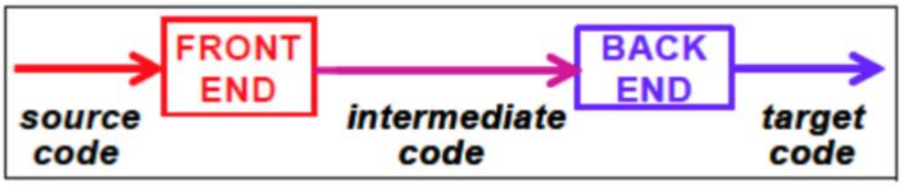
In compiler, the front-end translates a source program into an intermediate representation from which the back end generates target code.

Need For ICG

1. If a compiler translates the source language to its target machine language without generating IC, then for each new machine, a full native compiler is required.
2. IC eliminates the need of a new full compiler for every machine by keeping the analysis portion for all the compilers.
3. Synthesis part of back end depends on the target machine.

2 important things:

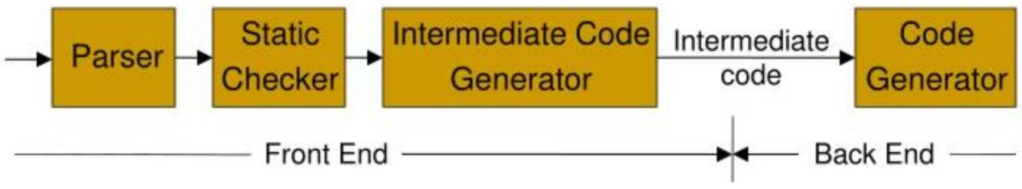
- IC Generation process should not be very complex
- It shouldn't be difficult to produce the target program from the intermediate code.



A source program can be translated directly into the target language, but some benefits of using intermediate form are:

- Retargeting is facilitated: a compiler for a different machine can be created by attaching a Back-end(which generate Target Code) for the new machine to an existing Front-end (which generate Intermediate Code).
- A machine Independent Code-Optimizer can be applied to the Intermediate Representation.

Logical Structure Of A Compiler Front End



5.2.1 INTERMEDIATE LANGUAGES

The most commonly used intermediate representations were :-

- **Syntax Tree**
- **DAG (Direct Acyclic Graph)**
- **Postfix Notation**
- **3 Address Code**

5.2.1.1 GRAPHICAL REPRESENTATION

Includes both

- **Syntax Tree**
- **DAG (Direct Acyclic Graph)**

Syntax Tree Or Abstract Syntax Tree(AST)

- Graphical Intermediate Representation
- Syntax Tree depicts the hierarchical structure of a source program.
- Syntax tree (AST) is a condensed form of parse tree useful for representing language constructs.

EXAMPLE

Parse tree and syntax tree for **3 * 5 + 4** as follows.

Grammar

E → E + T

E → E - T

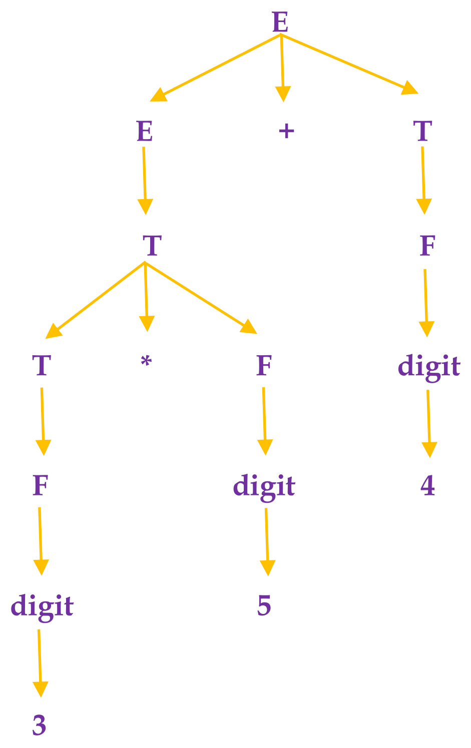
E→ T

T→T * F

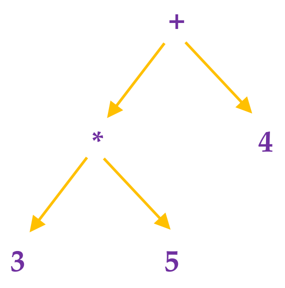
T→F

F→ digit

Parse Tree



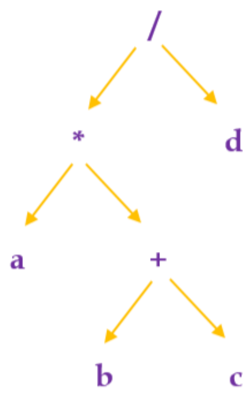
Syntax Tree



Parse Tree VS Syntax Tree

Parse Tree	Syntax Tree
A parse tree is a graphical representation of a replacement process in a derivation	A syntax tree (AST) is a condensed form of parse tree
Each interior node represents a grammar rule	Each interior node represents an operator
Each leaf node represents a terminal	Each leaf node represents an operand
Parse tree represent every detail from the real syntax	Syntax tree does not represent every detail from the real syntax Eg : No parenthesis

Syntax tree for a * (b + c) /d



Constructing Syntax Tree For Expression

- Each node in a syntax tree can be implemented in arecord with several fields.
- In the node of an operator, one field contains operator and remaining field contains pointer to the nodes for the operands.
- When used for translation, the nodes in a syntax tree may contain addition of fields to hold the values of attributes attached to the node.
- Following functions are used to create syntax tree
  1. **mknode(op,left,right)**: creates an operator node with label op and two fields containing pointers to left and right.
  2. **mkleaf(id,entry)**: creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for identifier
  3. **mkleaf(num,val)**: creates a number node with label num and a field containing val, the value of the number.
- Such functions return a pointer to a newly created node.

EXAMPLE

a – 4 + c

The tree is constructed bottom up

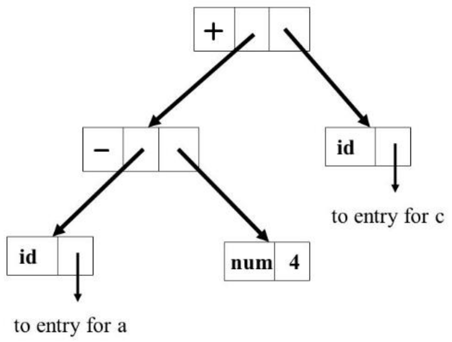
P_1 = **mkleaf(id,entry a)**

P_2 = **mkleaf(num, 4)**

P_3 = **mknode(-, P_1, P_2)**

P_4 = **mkleaf(id,entry c)**

P_5 = **mknode(+, P_3, P_4)**



Syntax Tree

Syntax directed definition

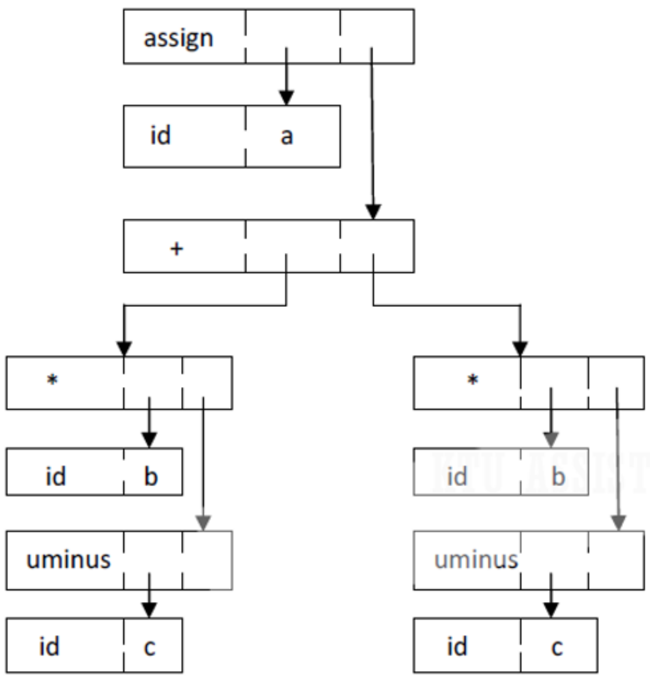
- Syntax trees for assignment statements are produced by the syntax-directed definition.
- Non terminal S generates an assignment statement.
- The two binary operators + and * are examples of the full operator set in a typical language. Operator associates and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a:=b* -c + b* -c

PRODUCTION	SEMANTIC RULE
$S \rightarrow id := E$	S.nptr : = mknode('assign',mkleaf(id, id.place), E.nptr)
$E \rightarrow E_1 + E_2$	E.nptr : = mknode('+', E_1.nptr, E_2.nptr)
$E \rightarrow E_1 * E_2$	E.nptr : = mknode('*', E_1.nptr, E_2.nptr)
$E \rightarrow - E_1$	E.nptr : = mknode('uminus', E_1.nptr)
$E \rightarrow (E_1)$	E.nptr : = E_1.nptr
$E \rightarrow id$	E.nptr : = mkleaf(id, id.place)

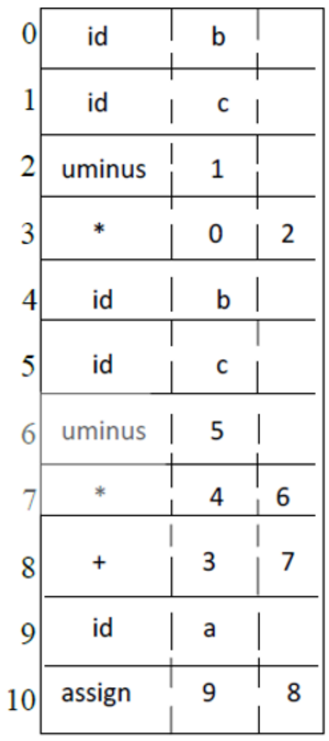
Syntax-directed definition to produce syntax trees for assignment statements

- The token id has an attribute *place* that points to the symbol-table entry for the identifier.
- A symbol-table entry can be found from an attribute **id.***name*, representing the lexeme associated with that occurrence of id.

- If the lexical analyser holds all lexemes in a single array of characters, then attribute name might be the index of the first character of the lexeme.
- Two representations of the syntax tree are as follows.



(a)



0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8

(b)

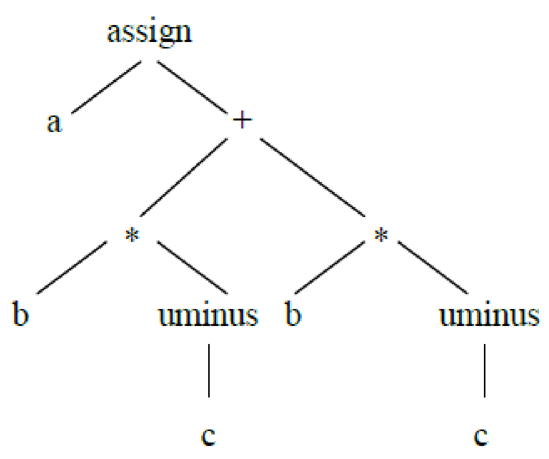
- In (a), each node is represented as a record with a field for its operator and additional fields for pointers to its children.
- In Fig (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node.
- All the nodes in the syntax tree can be visited by following winters, starting from the root at position 10.

Direct Acyclic Graph (DAG)

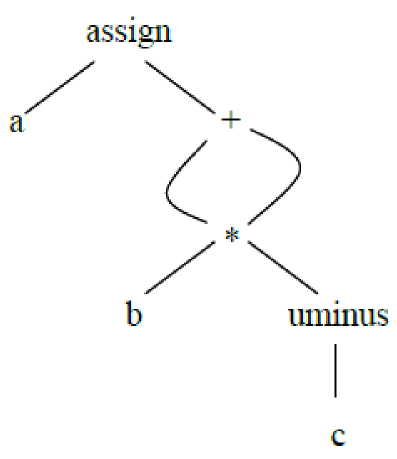
- Graphical Intermediate Representation
- Dag also gives the hierarchical structure of source program but in a more compact way because common sub expressions are identified.

EXAMPLE

a=b*-c + b*-c



(a) Syntax tree



(b) Dag

Postfix Notation

- Linearized representation of syntax tree
- In postfix notation, each operator appears immediately after its last operand.
- Operators can be evaluated in the order in which they appear in the string

EXAMPLE

Source String : a := b * -c + b * -c

Postfix String: a b c uminus * b c uminus * + assign

Postfix Rules

1. If E is a variable or constant, then the postfix notation for E is E itself.
2. If E is an expression of the form E1 op E2 then postfix notation for E is E1′ E2′ op, here E1′ and E2′ are the postfix notations for E1and E2, respectively
3. If E is an expression of the form (E), then the postfix notation for E is the same as the postfix notation for E.
4. For unary operation –E the postfix is E-

- Ex: postfix notation for 9- (5+2) is 952+-
- Postfix notation of an infix expression can be obtained using stack

5.2.1.2 THREE-ADDRESS CODE

- In Three address statement, at most 3 addresses are used to represent any statement.
- The reason for the term "three address code" is that each statement contains 3 addresses at most. Two for the operands and one for the result.

General Form Of 3 Address Code

$$a = b \text{ op } c$$

where,

a, b, c are the operands that can be names, constants or compiler generated temporaries.

op represents operator, such as fixed or floating point arithmetic operator or a logical operator on Boolean valued data. Thus a source language expression like **x + y * z** might be translated into a sequence

t_1 := y*z

t_2 := x+t_1 where, **t_1** and **t_2** are compiler generated temporary names.

Advantages Of Three Address Code

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.
- The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged - unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph.

Three Address Code corresponding to the syntax tree and DAG given above (page no:)

$t_1 := -c$	$t_1 := -c$
$t_2 := b * t_1$	$t_2 := b * t_1$
$t_3 := -c$	$t_5 := t_2 + t_2$
$t_4 := b * t_3$	$a := t_5$
$t_5 := t_2 + t_4$	
$a := t_5$	
(a) Code for the syntax tree	**(b) Code for the dag**

Types of Three-Address Statements

1. **Assignment statements**

   x := y op z, where op is a binary arithmetic or logical operation.

2. **Assignment instructions**

   x : = ***op*** **y**, where op is a unary operation . Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that for example, convert a fixed-point number to a floating-point number.

3. **Copy statements**

   x : = y where the value of y is assigned to x**.**

4. **Unconditional jump**

   goto L The three-address statement with label L is the next to be executed

5. **Conditional jump**

   if x relop y goto L This instruction applies a relational operator **(<, =, =, etc,)** to **x** and y, and executes the statement with label **L** next if x stands in relation **relop** to **y.** If not, the three-address statement following if **x relop y goto L** is executed next, as in the usual sequence.

6. **Procedural call and return**

   param x and **call p, n** for procedure calls and **return y**, where **y** representing a returned value is optional. Their typical use is as the sequence of three-address statements

   param x_1
   param x_2
   ……….
   param x_n
   call p,n

   generated as part of the call procedure **p(x_1 , x_2, . . . , x_n)** . The integer **n** indicating the number of actual-parameters in **"call p , *n*"** is not redundant because calls can be nested.

7. **Indexed Assignments**

   Indexed assignments of the form **x = y[i] or x[i] = y**

8. **Address and pointer assignments**

   Address and pointer operator of the form **x := &y**, **x := *y** and ***x := y**

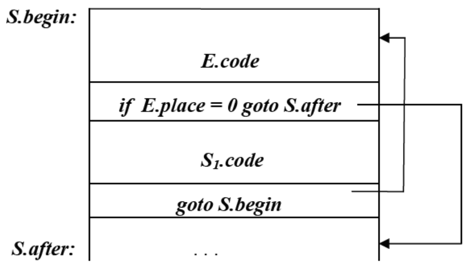
Syntax-Directed Translation Into Three-Address Code

- When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. for example **id** : = **E** consists of code to evaluate E into some temporary **t**, followed by the assignment **id**.*place* : = **t**.
- Given input **a:= b * - c + b + - c**, it produces the three address code in given above (page no:) The synthesized attribute **S.code** represents the three address code for the assignment S. The nonterminal E has two attributes:
    1. **E.***place* the name that will hold the value of E, and
    2. **E.**code. the sequence of three-address statements evaluating E.

Syntax-directed definition to produce three-address code for assignments.

PRODUCTION	SEMANTIC RULES
$S \rightarrow id := E$	*S.code : = E.code || gen(id.place ':=' E.place)*
$E \rightarrow E_1 + E_2$	*E.place := 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.place := newtemp;* *E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)*
$E \rightarrow - E_1$	*E.place := newtemp;* *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 : = ' '*

Semantic rule generating code for a while statement



PRODUCTION	SEMANTIC RULES
S → while *E* do S_1	*S.begin := newlabel;* *S.after := newlabel;* *S.code := gen(S.begin ':') ||* *E.code ||* *gen ('if' E.place '=' '0' 'goto' S.after)||* *S_1.code ||* *gen ('goto' S.begin) ||* *gen (S.after ':')*

- The function ***newtemp*** returns a sequence of distinct names **t_1, t_2**,……… in respose of successive calls. Notation *gen*(x ':= 'y '+' z is used to represent the three address statement **x := y + z.**
- Expressions appearing instead of variables like **x, y** and **z** are evaluated when passed to ***gen,*** and quoted operators or operand, like '**+**' are taken literally.
- Flow of control statements can be added to the language of assignments. The code for **S→ while E do S_1** is generated using new attributes ***S.begin*** and ***S.after*** to mark the first statement in the code for E and the statement following the code for **S**, respectively.
- The function ***newlabel*** returns a new label every time is called. We assume that a nonzero expression represents true; that is when the value of ***E*** becomes zero, control laves the while statement

Implementation Of Three-Address Statements

A three address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such, representations are

- **Quadruples**
- **Triples**
- **Indirect triples**

5.2.1.3 QUADRUPLES

- A quadruple is a record structure with four fields, which are ***op, ag1, arg2*** and ***result***
- The op field contains an internal code for the operator. The three address statement **x:= y op z** is represented by placing **y** in ***arg1***, **z** in ***arg2*** and **x** in ***result***.
- The contents of ***arg1, arg2***, and ***result*** are normally pointers to the symbol table entries for the names represented by these fields. If so temporary names must be entered into the symbol table as they are created.

<u>EXAMPLE 1</u>

Translate the following expression to quadruple triple and indirect triple

a + b * c | e ^ f + b * a

For the first construct the three address code for the expression

t1 = e ^ f
t2 = b * c
t3 = t2 / t1
t4 = b * a
t5 = a + t3
t6 = t5 + t4

Location	OP	arg1	arg2	Result
(0)	^	e	f	t1
(1)	*	b	c	t2
(2)	/	t2	t1	t3
(3)	*	b	a	t4
(4)	+	a	t3	t5
(5)	+	t3	t4	t6

Exceptions

- ⇨ The statement **x := op y**, where op is a unary operator is represented by placing **op** in the operator field, **y** in the argument field & n in the result field. The ***arg2*** is not used
- ⇨ A statement like **param t1** is represented by placing **param** in the operator field and t1 in the arg1 field. Neither ***arg2*** not result field is used
- ⇨ Unconditional & Conditional jump statements are represented by placing the target in the result field.

5.2.1.4 TRIPLES

- In triples representation, the use of temporary variables is avoided & instead reference to instructions are made
- So three address statements can be represented by records with only there fields OP, arg1 & arg2.
- Since, there fields are used this intermediated code formal is known as triples

Advantages

- ❖ No need to use temporary variable which saves memory as well as time

Disadvantages

- ❖ Triple representation is difficult to use for optimizing compilers
- ❖ Because for optimization statements need to be suffled.
- ❖ for e.g. statement 1 can be come down or statement 2 can go up ect.
- ❖ So the reference we used in their representation will change.

<u>**EXAMPLE 1**</u>

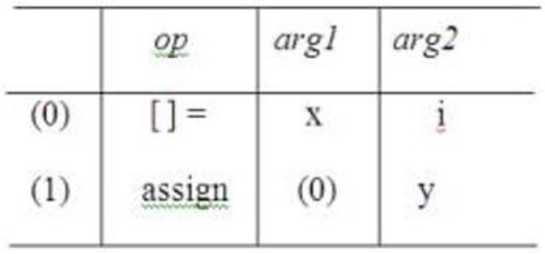
a + b * c | e ^ f + b * a

t1 = e ^ f
t2 = b * c
t3 = t2 / t1
t4 = b * a
t5 = a + t3
t6 = t5 + t4

Location	OP	arg1	arg2
(0)	^	e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

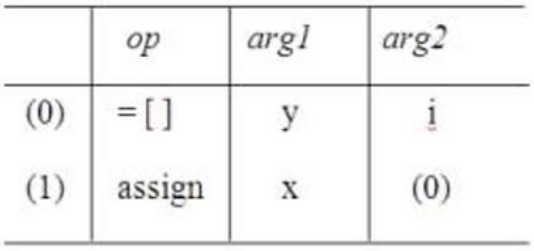
EXAMPLE 2

A ternary operation like x[i] : = y requires two entries in the triple structure while x : = y[i] is naturally represented as two operations.



	op	arg1	arg2
(0)	[ ] =	x	i
(1)	assign	(0)	y

x[i] := y



	op	arg1	arg2
(0)	= [ ]	y	i
(1)	assign	x	(0)

x := y[i]

INDIRECT TRIPLES

- This representation is an enhancement over triple representation.
- It uses an additional instruction array to led the pointer to the triples in the desired order.
- Since, it uses pointers instead of position to stage reposition the expression to produce an optimized code.

EXAMPLE 1

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

Location	op	arg1	arg2
(0)	^	E	f
(1)	*	B	c
(2)	/	(1)	(0)
(3)	*	B	a
(4)	+	A	(2)
(5)	+	(4)	(3)

Comparison

- When we ultimately produce the target code each temporary and programmer defined name will assign runtime memory location
- This location will be entered into symbol table entry of that data.
- Using the quadruple notation, a three address statement containing a temporary can immediately access the location for that temporary via symbol table.
- But this is not possible with triples notation.
- With quadruple notation, statements can often move around which makes optimization easier.
- This is achieved because using quadruple notation the symbol table interposes high degree of indirection between computation of a value and its use.
- With quadruple notation, if we move a statement computing **x**, the statement using **x** requires no change.
- But with triples, moving a statement that defines a temporary value requires us to change all references to that statement in arg1 and arg2 arrays. This makes triples difficult to use in optimizing compiler
- With indirect triples also, there is no such problem.
- A statement can be moved by reordering the statement list.

Space Utilization

- Quadruples and indirect triples requires same amount of space for storage (normal case).
- But if same temporary value is used more than once indirect triples can save some space. This is bcz, 2 or more entries in statement array can point to the same line of op-arg1-arg2 structure.
- Triples requires less space for storage compared to above 2.
- **Quadruples**
    - direct access of the location for temporaries
    - easier for optimization

  Triples
    - space efficiency

  Indirect Triples
    - easier for optimization
    - space efficiency

PROBLEM 1

Translate the following expression to quadruple tuples & indirect tuples

a = b * - c + b * - c

Sol : - Three address code for given expression is

TAC

t1 = uniminus c
t2 = b* t1
t3 = uniminus c
t4 = b* t3
t5 = t2 + t4
Q = t5

QUADRUPLES

Location	OP	arg1	arg2	result
(0)	uniminus	c		t1
(1)	*	b	t1	t2
(3)	uniminus	c		t3
(4)	*	b	t3	t4
(5)	+	t2	t4	t5
(6)	=	t5		a

TRIPLES

Location	OP	arg1	arg2
(1)	uniminus	c	
(2)	*	b	(1)
(3)	uniminus	c	
(4)	*	b	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

INDIRECT TRIPLES

	Statements
35	(1)
36	(2)
37	(3)
38	(4)
39	(5)
40	(6)

Location	OP	arg1	arg2
(1)	uniminus	C	
(2)	*	B	(1)
(3)	uniminus	C	
(4)	*	B	(3)
(5)	+	(2)	(4)
(6)	=	A	(5)

<u>EXAMPLE</u> : Annotated Parse Tree For Generation Of TAC For Assignment Statements

$x = a*b+c+d$

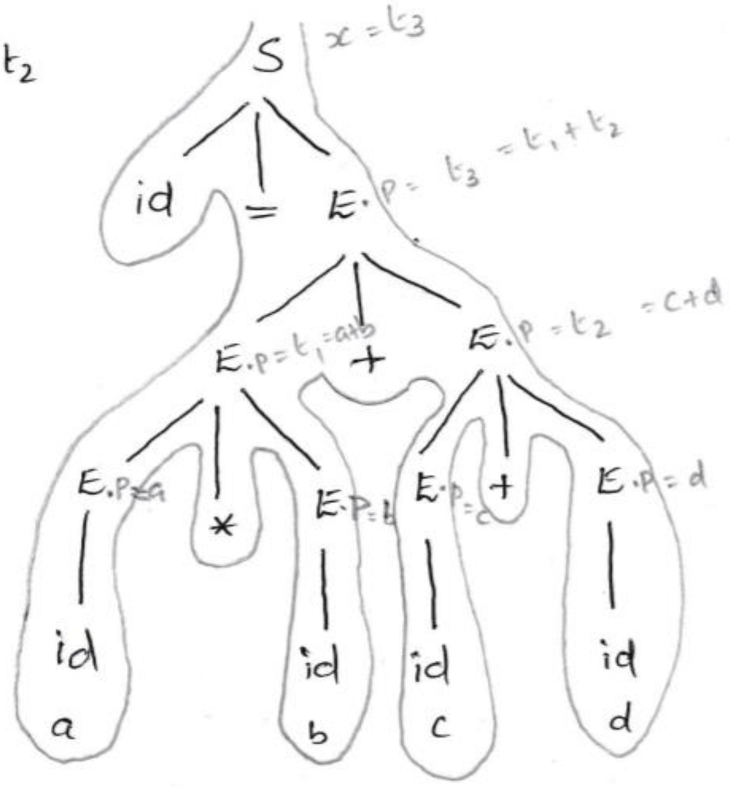
<u>TAC</u>

$t_1 = a * b$

$t_2 = c+d$

$t_3 = t_1+t_2$

$x = t_3$



Syntax-directed definition to produce three-address code for assignments.

PRODUCTION	SEMANTIC RULES
S → id : = E	*S.code : = E.code || gen(id.place ':=' E.place)*
E → E_1 + E_2	*E.place := newtemp;* *E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '+' E_2.place)*
*E → E_1 * E_2*	*E.place := newtemp;* *E.code := E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)*
E → - E_1	*E.place := newtemp;* *E.code := E_1.code || gen(E.place ':=' 'uminus' E_1.place)*
E → (E_1)	*E.place : = E_1.place;* *E.code : = E_1.code*
E → id	*E.place : = id.place;* *E.code : = ' '*