MODULE - IV

SYNTAX DIRECTED TRANSLATION AND INTERMEDIATE CODE GENERATION

SYLLABUS:
Syntax directed translation - Syntax directed definitions, S-attributed definitions, L-attributed definitions, Bottom-up evaluation of S-attributed definitions. Run-Time Environments - Source Language issues, Storage organization, Storage-allocation strategies. Intermediate Code Generation - Intermediate languages, Graphical representations, Three-Address code, Quadruples, Triples.

INTRODUCTION

We associate information with a language construct by attaching attributes to the grammar symbols representing the construct.

There are two notations for the productions:

- Syntax Directed Definitions (SDD)
- Translation Schemes

Syntax directed definitions hide many implementation details and free the user from having to specify explicitly the order in which translations takes place. Translation schemes indicate the order in which semantic rules are to be evaluated, so they allow some implementation details to be shown. In both the methods, we parse the input token stream, build the parse tree, and then traverse the tree as needed to evaluate the semantic rules at the parse tree nodes. Evaluation of the semantic rules may generate code, save information in a symbol table, issue error messages, or perform any other activities.

A Syntax Directed Definition (SDD) specifies the values of attributes by associating semantic rules with the grammar productions. For example,

$$E \rightarrow E + T \ \{ E.value = E.value + T.value \}$$

A Syntax Directed Translation (SDT) scheme embeds program fragments called semantic actions within production bodies. For example,

$$E \rightarrow E_1 + T \ \{ \text{print } '+' \}$$

Syntax Directed Definition (SDD)

Syntax directed definition is a CFG together with attributes and rules. Attributes are associated with the grammar symbols and the rules are associated with the productions. If X is a symbol and '*a*' is one of its attributes, then we write X.a to denote the value of *a* at a particular parse tree node labelled X.

Attributes are properties associated with grammar symbols. Attributes can be numbers, strings, memory locations, data types, etc. Based on the way the attributes get their values, they can be broadly divided into two categories:

1. Inherited Attributes
2. Synthesized Attributes

Synthesized Attributes: 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.

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.

Inherited Attributes: These are attributes that inherit their values from their 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 Synthesized Attributes:

- Write the SDD using apppropriate 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 and its corresponding semantic rules:

	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

Fig: SDD of a **simple desk calculator**

The above grammar evaluates expressions terminated by an endmarker n. The rule for production 1, L → En , sets L.val to E.val. Production 2, E → E1 + T, also has one rule, which computes the val attribute for the head E as the sum of the values at E1 and T. At any parse tree node N labeled E, the value of val for E is the sum of the values of val at the children of node N labeled E and T. Production 3, E→T, has a single rule that defines the value of val for E to be the same as the value of val at the child for T. Production 4 is similar to the second production; its rule multiplies the values at the children instead of adding them. The rules for productions 5 and 6 copy values at a child, like that for the third production. Production 7 gives F.val the value of a digit, that is, the numerical value of the token digit that the lexical analyzer returned.

A parse tree showing the values of its attributes is called an **annotated parse tree**. For the above defined grammar, let's construct an annotated parse tree for the input 3 * 5 + 4 n

For computation of attributes, we start from the leftmost bottom node. All the attributes in the below annotated parse tree are synthesized attributes. That is, L.val, E.val, F.val and digit.lexval are synthesized attributes. Also note that 'lexval' attribute will be always a synthesized attribute.

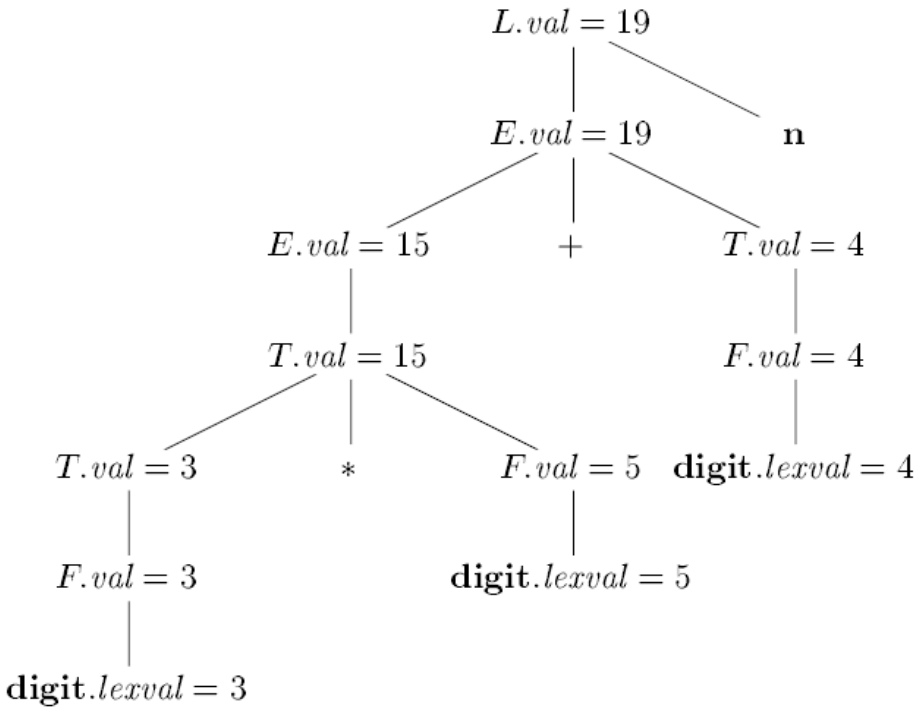


Fig: Annotated parse tree for 3 * 5 + 4 n

Computation of Inherited Attributes

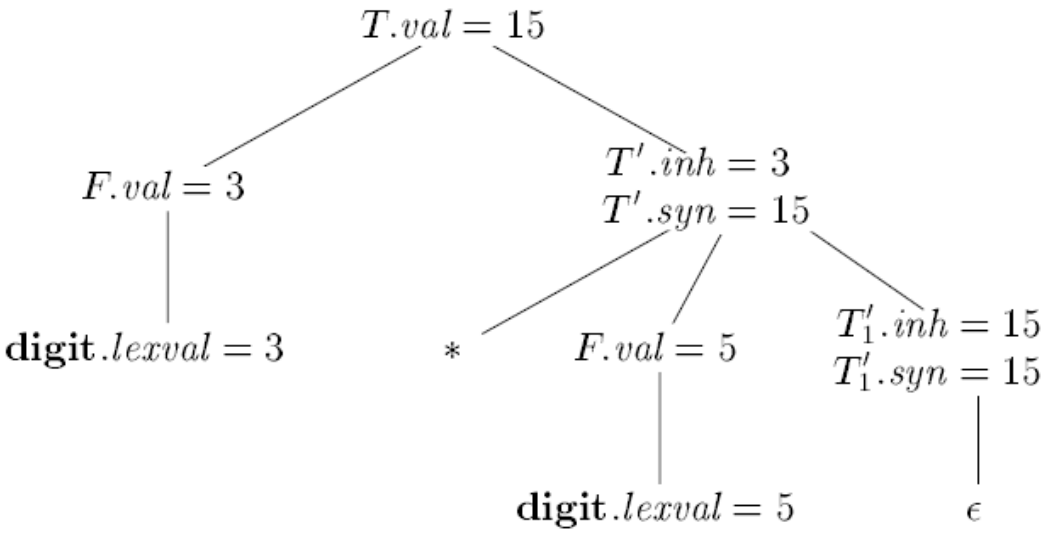
- Construct the SDD using semantic actions.
- The annotated parse tree is generated and attribute values are computed in top down manner.

Consider the following grammar:

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Each of the non-terminals T and F has a synthesized attribute *val;* the terminal **digit** has a synthesized attribute *lexval*. The nonterminal T' has two attributes: an inherited attribute *inh* and a synthesized attribute *syn*.

For the above grammar, consider the annotated parse tree for 3 * 5.



The leftmost leaf in the parse tree, labelled digit, has attribute value lexval = 3, where the 3 is supplied by the lexical analyzer. Its parent is for production 4, F → digit. The only semantic rule associated with this production defines F.val = digit.lexval, which equals 3.

At the second child of the root, the inherited attribute T'.inh is defined by the semantic rule T'.inh = F.val associated with production 1. Thus, the value of T'.inh will be 3.

The production at the node for T' is T' → * F T'_1. (We retain the subscript 1 in the annotated parse tree to distinguish between the two nodes for T'. The inherited attribute T'_1.inh is defined by the semantic rule T'_1.inh = T'.inh * F.val associated with production 2.

With T'.inh = 3 and F.val = 5, we get T'_1.inh = 15. At the lower node for T'_1, the production is T'→ ε. The semantic rule T'.syn = T'.inh defines T'_1.syn = 15. The syn attributes at the nodes for T' pass the value 15 up the tree to the node for T, where T.val = 15.

Evaluation orders for SDD's

- 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

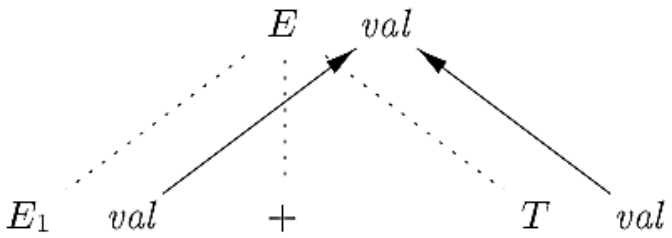
Dependency Graphs:

- A dependency graph depicts the flow of information among the attribute instances in a particular parse tree.
- An edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.
- 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).

Consider the following production rule.

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

The corresponding dependency graph will be,



Here, E.val is synthesized from E_1.val and T.val. We use dotted lines to show the actual parse tree and the solid lines to depict the edges of the dependency graph.

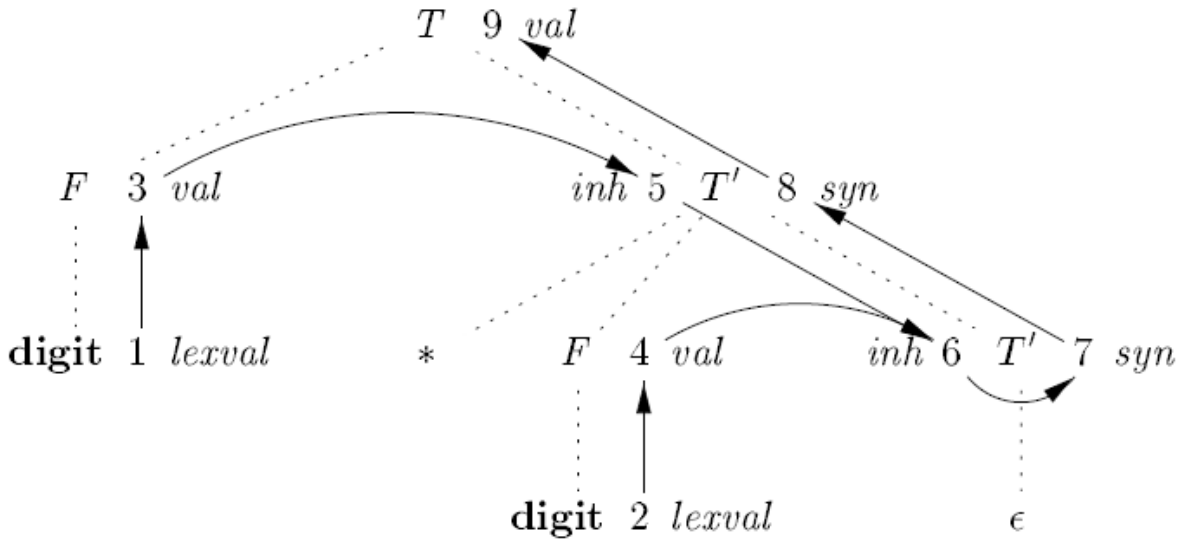
- The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of a parse tree.

- If the dependency graph has an edge from node M to N, then the attribute corresponding to M must be evaluated before the attribute of N. Thus, the only allowable orders of evaluation are those sequences of nodes N_1 , N_2, …., N_k such that if there is an edge of the dependency graph from N_i to N_j , then i < j .
- Such an ordering embeds a directed graph into a linear order, and is known as the ***topological sort*** of the graph.
- If there is any cycle in the graph, then there are no topological sorts. That is, there is no way to evaluate the SDD on this parse tree. If there is no cycle, then there will be at least one topological sort.

Consider the below grammar (which we have already seen in inherited attributes portion).

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

The complete dependency graph for the above grammar is shown below.



S-attributed and L-attributed SDD

There are two kinds of SDDs.

1. S-attributed
2. L-attributed

S-attributed SDD:

An SDD that involves only synthesized attributes is known as **S-attributed SDD.** When an SDD is S-attributed, we can evaluate its attributes in any bottom-up order of the nodes of the parse tree.

Example of S-attributed SDD: Consider our previous example on **Simple Desk Calculator.**

	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

This is a S-attributed SDD, since all the attributes are synthesized attributes only.

L-attributed SDD:

This is the second class of SDD which can contain synthesized as well as inherited attributes. Also we have to note that, between the attributes associated with a production body, dependency graph edges can go from left-to-right, but not from right-to-left (and hence the name "L-attributed"). A L-attributed SDD will be evaluated from top-to-down and left-to-right.

Example of L-attributed SDD: Consider the below SDD for S**imple Type Declaration**

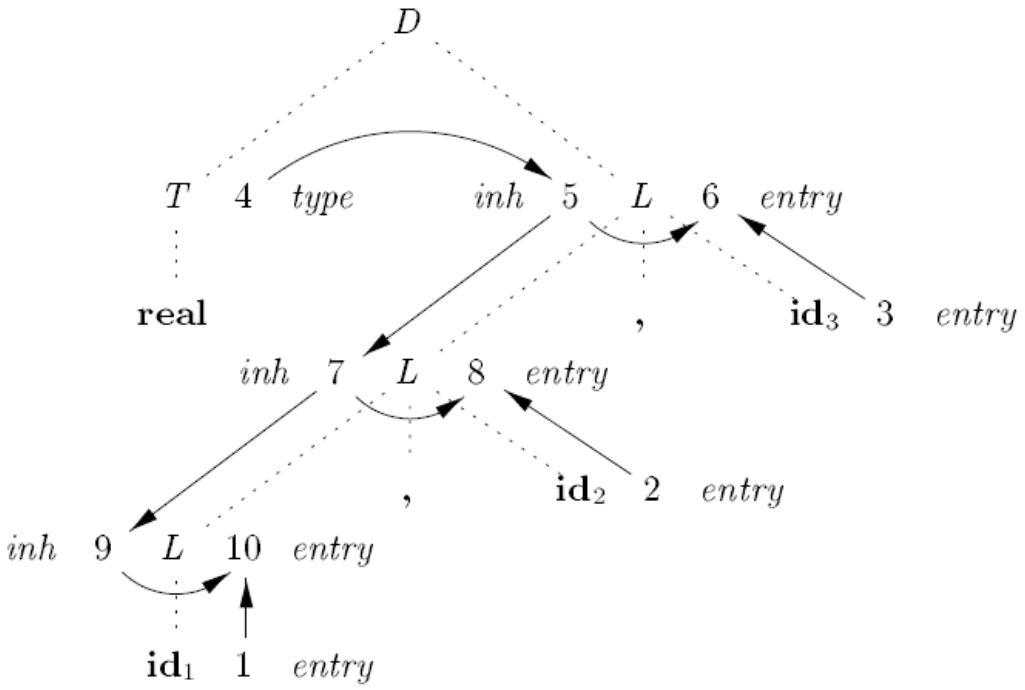
	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \mathbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \mathbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

In the above grammar, the non-terminal D represents a declaration, which consists of a type T followed by a list L of identifiers.

Productions 4 and 5 have a rule in which a function addType is called with two arguments:

1. id.entry, a lexical value that points to a symbol table object.
2. L.inh, the type being assigned to every identifier on the list.

A dependency graph for the input **float** $\boldsymbol{id_1, id_2, id_3}$ appears in the below figure. Numbers 1 to 10 represents the nodes of the dependency graph.



Syntax Directed Translation (SDT)

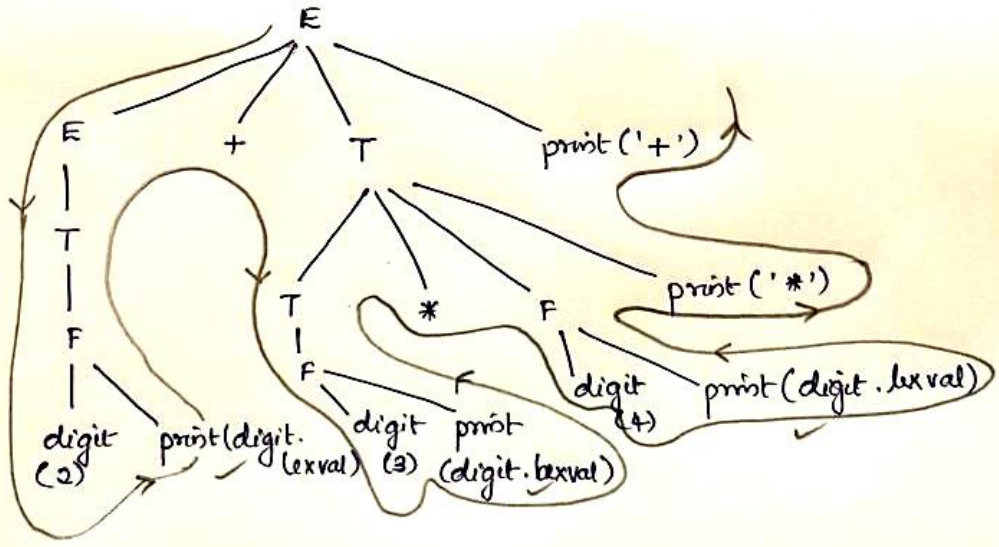
It is a CFG with program fragments embedded with the production body. The program fragments are known as the semantic actions.

Example: The syntax directed translation scheme for **infix-to-postfix conversion**

Infix-to-Postfix SDT :-

GRAMMAR	SEMANTIC ACTION
E → E+T	{ print ('+')}
E → T	{ }
T → T * F	{ print ('*')}
T → F	{ }
F → digit	{ print (digit.lexval)}

- lets now convert the infix string 2 + 3 * 4 to postfix.



o/p :- 2 3 4 * +

Bottom up evaluation of S-attributed definition

Consider the example of Simple Desk Calculator for the input string 4 * 5 + 3.

$L \rightarrow E$ **n**

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow$ **digit**

(Note : n ⇒ endmarker)

STACK		INPUT	REDUCE ACTION
STATE	VALUE		
		4 * 5 + 3 n	
digit	4	* 5 + 3n	
F	4	* 5 + 3n	F → digit
T	4	* 5 + 3 n	T → F
T *	4	5 + 3n	
T * digit	4, 5	+ 3 n	
T * F	4, 5	+ 3n	F → digit
T	20	+ 3n	T → T * F
E	20	+ 3n	E → T
E +	20	3 n	
E + digit	20, 3	n	
E + F	20, 3	n	F → digit
E + T	20, 3	n	T → F
E n	23	n	E → E + T
L	23	•	L → E n.

∴ o/p = 23

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.

Source Language Issues

Suppose that the source program is made up of procedures (as in Pascal). This section distinguishes between the source text of a procedure and its activations at the run time. In this section we deal with:

- Procedures
- Activation Trees
- Control Stacks
- The Scope of a Declaration
- Binding of Names

Procedures:

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;
```

A complete program may also be treated as a procedure.

When a procedure name appears with in an executable statement, the procedure is said to be ***called*** at that point. The identifiers appearing in a procedure definition are known as *formal parameters* of the procedure. Arguments that are passed to a called procedure are known as the *actual parameters.*

Activation Trees:

- Each execution of procedure is referred to as an activation of the procedure. The lifetime of an activation is the sequence of steps present in the execution of that 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.

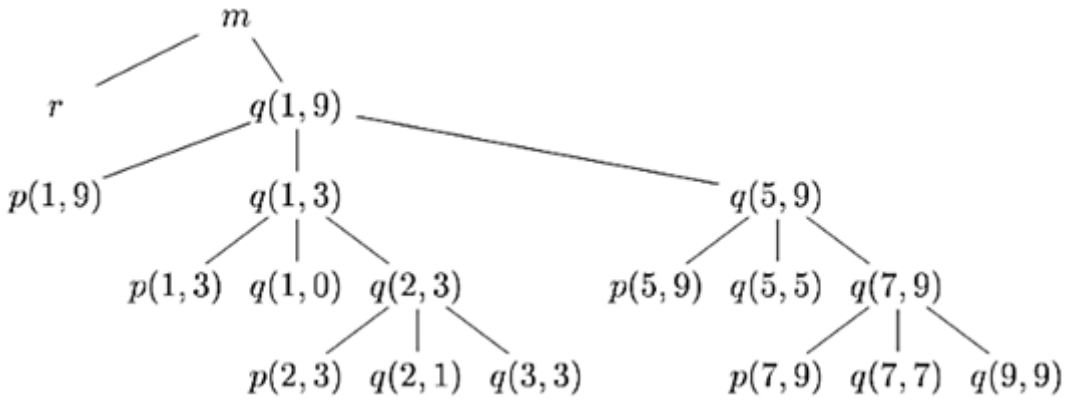
- Properties of activation trees are:
    - Each node represents an activation of a procedure.
    - The root shows the activation of the main function.
    - The node for procedure 'x' is the parent of node for procedure 'y' if and only if the control flows from procedure 'x' to procedure 'y'.
    - The node for 'x' is to the left of the node for 'y' if and only if the lifetime of 'x' occurs before the lifetime of 'y'.

Example:

Consider the following program of quicksort

```
main()
{
        readarray();
        quicksort(1, 9 );
}

quicksort(int m, int n)

{

        int i= partition(m,n);

        quicksort(m,i-1);

        quicksort(i+1,n);

}
```

Suppose that the above procedure partitioned the array with i=4 during the first call. Then the activation tree will be as given below.



Control Stack:

- The flow of control in a program corresponds to a depth-first traversal of the activation tree that starts at the root, visits a node before its children, and recursively visits children at each node in a left-to-right order.
- 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).
- 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.

For example, consider the above activation tree, when quicksort(2,3) gets executed. The contents of the control stack will be:

quicksort(2,3)
quicksort(1,3)
quicksort(1,9)
main()

The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a name.

Eg: var i : integer ;

There may be independent declarations of the same name in different parts of a program. The *scope* rules of a language determine which declaration of a name applies when the name appears in the text of a program.

The portion of the program to which a declaration applies is called the scope of that declaration. An occurrence of a name in a procedure is said to be *local* to the procedure if it is in the scope of a declaration within the procedure; otherwise, the occurrence is said to be *nonlocal.*

Bindings of Names:

The term *environment* refers to a function that maps a name to a storage location, and the term *state* refers to a function that maps a storage location to the value held there (as in figure given below). That is, an environment maps a name to an l-value, and a state maps the l-value to an r-value.

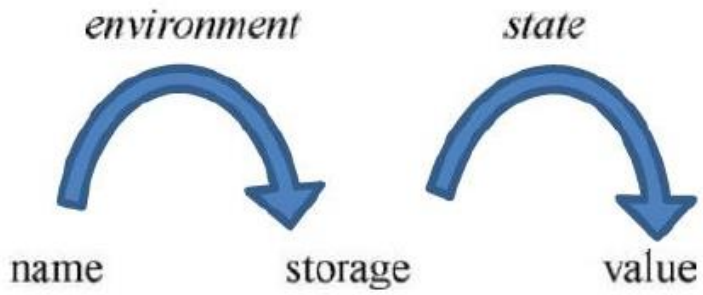


Fig: two-stage mapping from names to values

Environments and states are different. An assignment changes the state but not the environment. For example, suppose that storage address 100, associated with the variable pi, holds the value 0. After the assignment pi = 3.14, the same storage is associated with pi, but the value held there is 3.14.

When an environment associates storage location s with a name x, we say that x is *bound* to s. A binding is the dynamic counterpart of a declaration.

STATIC NOTATION	DYNAMIC NOTATION
Definition of a procedure	Activations of the procedure
Declaration of a name	Bindings of the name
Scope of a declaration	Lifetime of a binding

Fig: Corresponding static and dynamic notations

Storage Organization

Suppose that the compiler obtains a block of storage from the operating system for the compiled program to run. This run-time storage might be subdivided to hold: the generated target code (that is the executable code), data objects and a counterpart of the control stack to keep track of procedure activations.

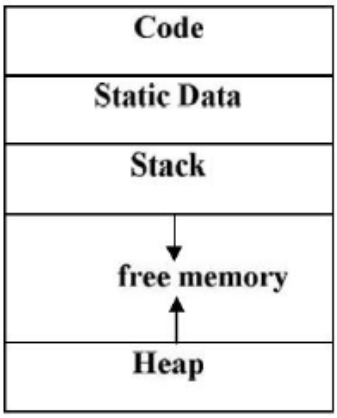


Fig: Subdivision of run-time memory

The size of the generated target code is fixed at compile time, so the compiler can place it in a statically determined area. Similarly, the size of some of the data objects are also known at the compile time and these too can be placed in a statically determined area.

The control stacks are used to manage the activation of procedures. When a call occurs, execution of an activation is interrupted and information about the status of the machine, such as the value of the program counter and machine registers, is saved on the stack. When control returns from the call, this activation can be restarted after restoring the values of relevant registers and setting the program counter to the point immediately after the call. Data objects whose lifetimes are contained in that of an activation can be allocated on the stack, along with other information associated with the activation.

A separate area of run-time memory, called a heap, holds all other information. Implementation of languages in which the lifetimes of activations cannot be depicted by an activation tree might use the heap to keep information about activations.

The sizes of stack and heap can change as the program executes. They grow from opposite sides. By convention, stack grows down and the heap grows up.

Inshort we can say that,

- **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)

Activation Records:

Information needed by a single execution of a procedure is managed using a contiguous block of storage called an *activation record* or *frame*, which consists of the following fields.

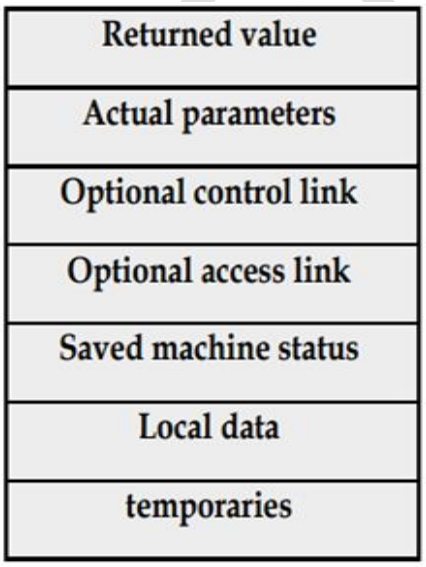


Fig: A general activation record

The purpose of the fields of an activation record is as follows:

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 link points 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.

The sizes of these fields can be determined at the time a procedure is called.

Storage Allocation Strategies

A different storage allocation strategy is used in each of the three data areas of storage organization.

1. **Static allocation** lays out storage for all data objects at compile time
2. **Stack allocation** manages the run-time storage as a stack
3. **Heap allocation** allocates and de-allocates storage as needed at run time

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 is activated, its names bounded to the same storage location.
- Therefore, values of local names are retained across activations of a procedure. That is, when control returns to a procedure the value of the locals 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 that 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.

The limitations of static allocation are:

1. The size of a data object and constraints on its position in memory must be known at compile time.
2. Recursive procedures are restricted, because all activations of a procedure use the same bindings for local names.
3. Data structures cannot be created dynamically, since there is no mechanism for storage allocation at run time.

Stack Allocation:

Stack allocation is based on the idea of a control stack; storage is organized as a stack and activation records are pushed and popped as activations begin and end respectively. Storage for the locals in each call of a procedure is contained in the activation record for that call. Thus locals are bound to fresh storage in each activation, because a new activation record is pushed onto the stack when a call is made. Also, the values of locals are deleted when the activation ends.

Suppose that the register *top* marks the top of the stack. At run time, an activation record can be allocated and de-allocated by incrementing and decrementing the top of the stack by the size of the record. If procedure *q* has a record of size *a,* then the top is incremented by *a* just before the target code of *q* is executed. When control returns from *q*, *top* is decremented by *a.*

Calling Sequences:

- Procedure calls are implemented by generating ***calling sequences***.
- A call sequence allocates an activation record and enters information into its fields.
- A ***return sequence*** restores the state of the machine so the calling procedure can continue execution.
- The code in a calling sequence is divided between the calling procedure (the caller) and the procedure it calls (the callee).
- The principles that aids the design of calling sequences and activation records are:
    - Fields whose sizes are fixed early are placed in the middle. i.e., the control link, access link, and machine status fields appear in the middle.
    - Values communicated between the caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to the caller's activation record.
    - Items whose size may not be known early enough are placed at the end of the activation record.
    - We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of the fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets relative to the top-of-stack pointer.

An example of how caller and callee might cooperate in managing the stack is suggested in the below figure.

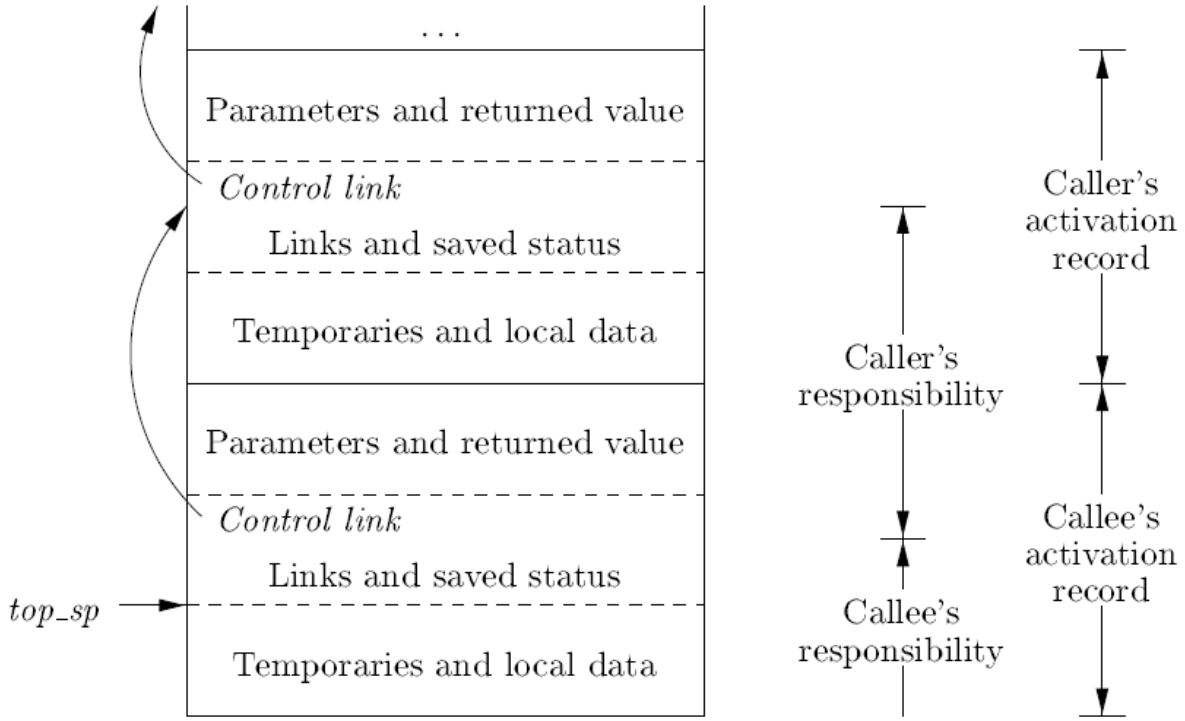


Fig: Division of task between the caller and the callee

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.

The corresponding return sequence will be:

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 (i.e., arrays whose size depends on the value of one or more parameters of the called procedure) is shown in following figure. The same scheme works for the objects of any type.

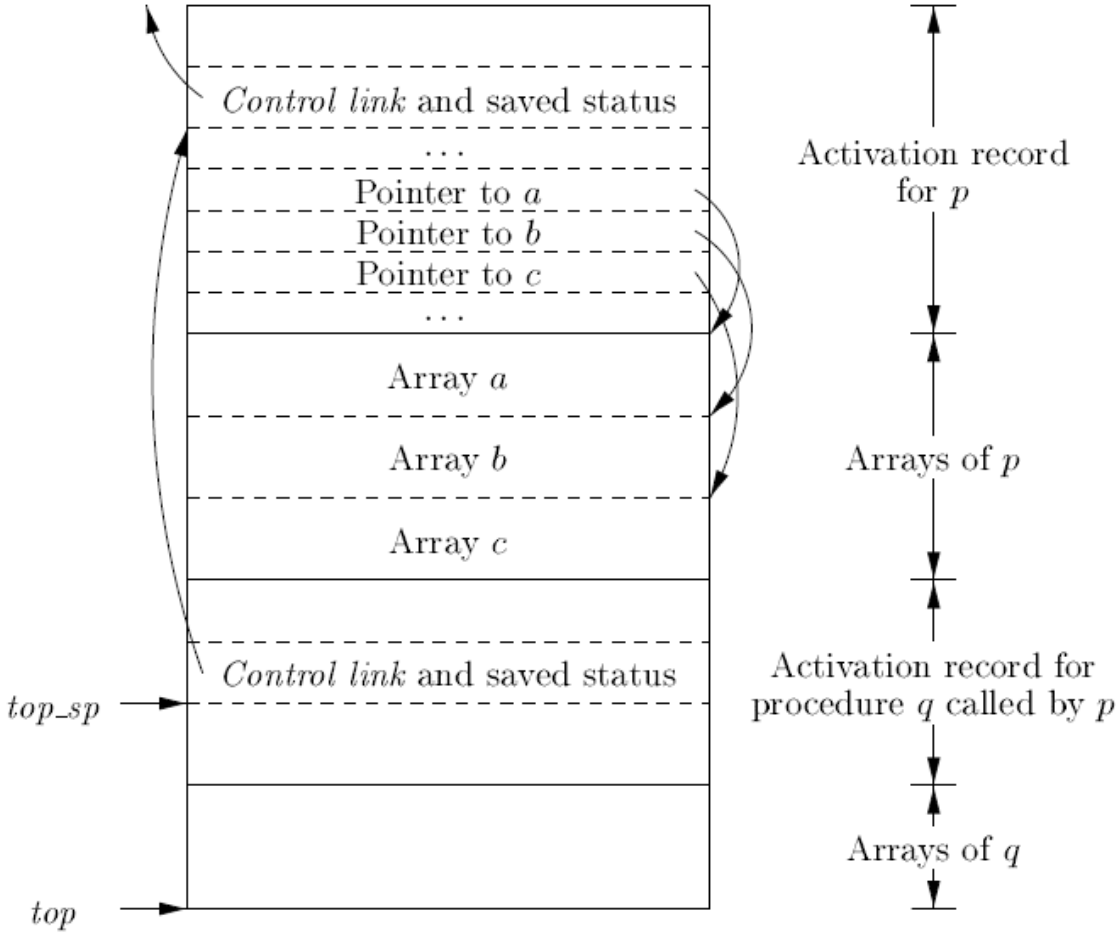


Fig: Access to dynamically allocated arrays

In the above figure, procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p, although it appears on the stack. Only a pointer to the beginning of each array appears in the activation record itself. Thus, when p is executing, these pointers are at known offsets from the top-of-stack pointer, so the target code can access array elements through these pointers.

Also, in the figure, the activation record for a procedure q, called by p is shown. The activation record for q begins after the arrays of p, and any variable-length arrays of q are located beyond that.

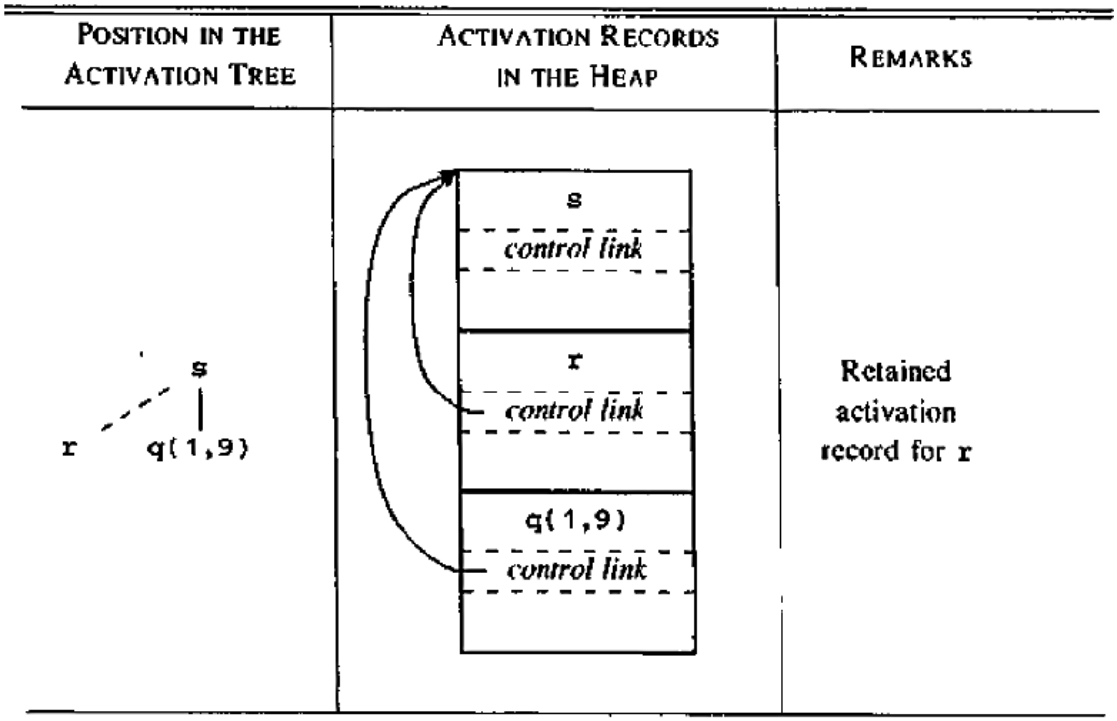
Access to the data on the stack is through two pointers, *top* and *top_sp*. Here, *top* marks the actual top of stack; it points to the position at which the next activation record will begin. The second, *top_sp* is used to find local, fixed-length fields of the top activation record.

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

In each of the above cases, the deallocation of activation records need not occur in a last-in first-out fashion, so storage cannot be organized as a stack. 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 time the heap will consists of alternate areas that are free and in use. Heap allocation for activation records is shown in the below figure.



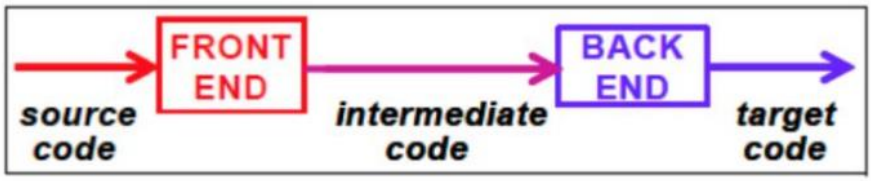
In the above figure, the record for an activation of procedure x is retained when the activation ends. The record for the new activation of procedure q(1,9) therefore cannot follow that for s physically, as it did in stack allocation. Now if the retained activation record for x is deallocated, there will be free space in the heap between the activation records for s and q(1,9). It is left to the heap manager to make use of this space.

INTERMEDIATE CODE GENERATION

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

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.



Benefits of using 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.

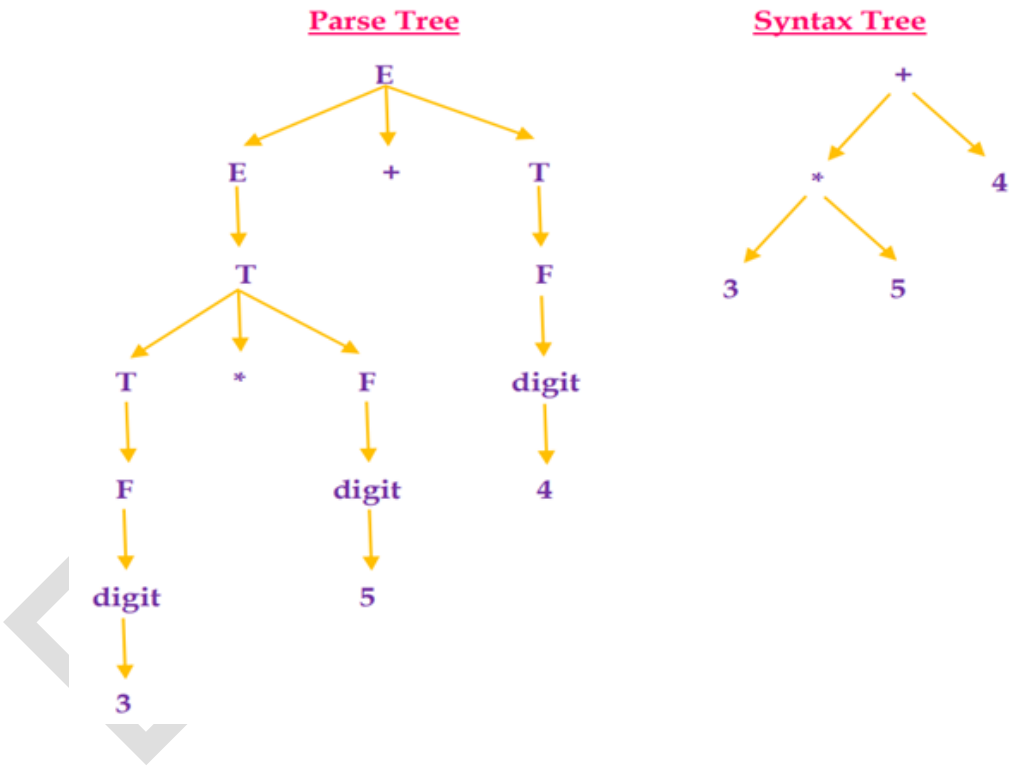
Intermediate Languages or the most commonly used **intermediate representations** are:

- Syntax Tree
- Direct Acyclic Graph (DAG)
- Postfix Notation
- Three-address Code

Graphical Representations:

We use **Syntax trees** and **DAG** to represent the intermediate code graphically.

1. **Syntax Trees (or Abstract Syntax Tree –AST):**
   - Graphical intermediate representation
   - Syntax tree represents the hierarchical structure of a source program
   - Syntax trees are condensed form of parse trees
   - Example: draw the syntax and parse trees for 3 * 5 + 4
     The grammar is:
     E → E + T
     E → E - T
     E → T
     T → T * F
     T → F
     F → digit



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

Constructing Syntax Trees for expressions:

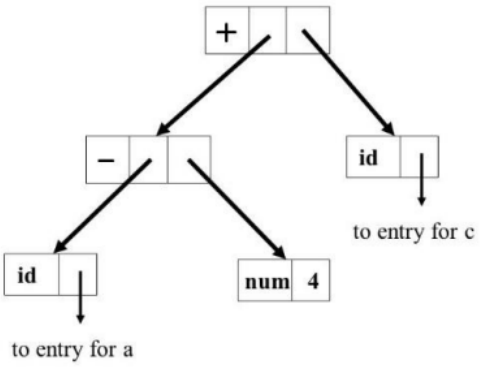
- Each node in a syntax tree can be implemented as a record with several fields
- In the node of an operator, one field contains the operator and the remaining fields contains pointer to the nodes of the operands
- Following functions are used to create syntax tree:
  1. **mknode(op,left,right)**: creates 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.
- These functions return a pointer to the 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)**

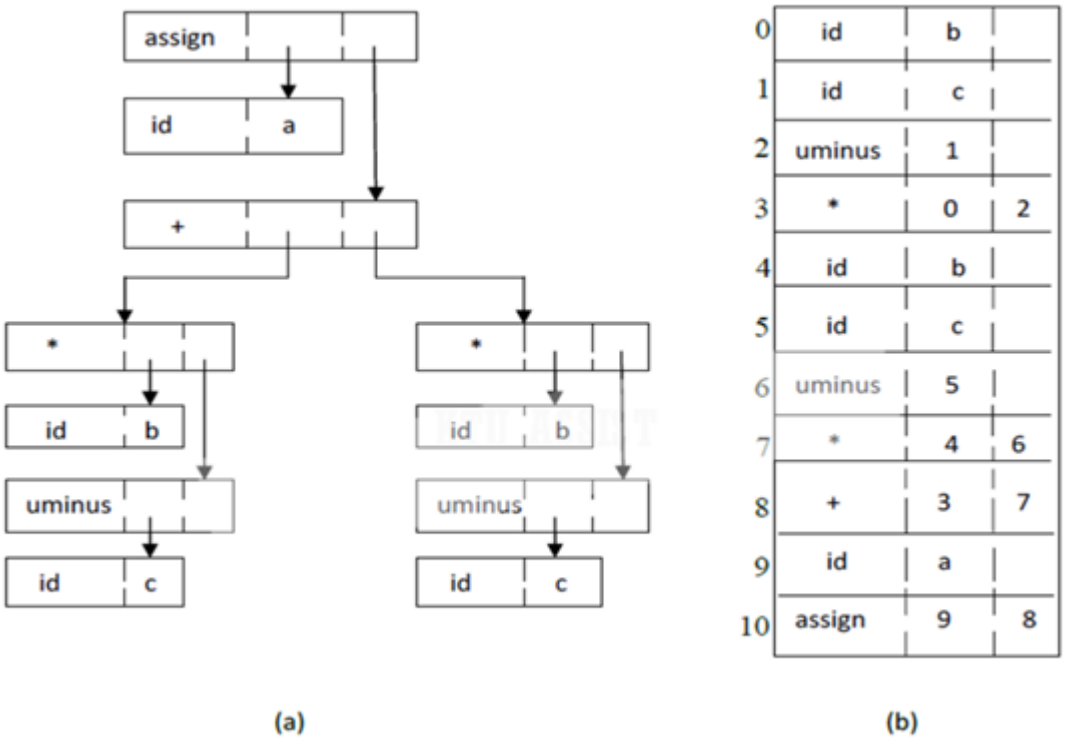


Syntax Tree

Syntax trees can be represented using 2 methods:

a) Using nodes
b) Using array

For the expression, a = b * - c + b * - c, the two representations are:



(a) (b)

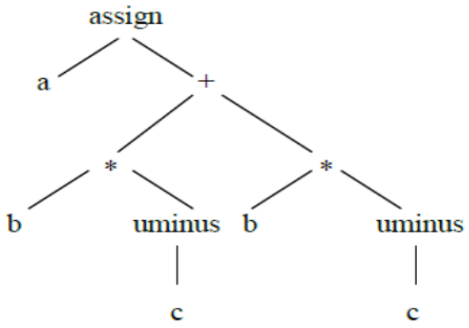
In (a), each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited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

2. Direct Acyclic Graph (DAG)

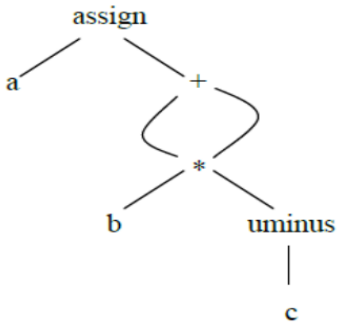
- Graphical intermediate representation
- DAG also gives the hierarchical structure of source program, but in a more compact way because the common sub expressions are identified.

EXAMPLE

a=b*-c + b*-c



(a) Syntax tree



(b) Dag

3. 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
- Postfix notation of an infix expression can be obtained using a stack

EXAMPLE

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

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

Postfix Rules:

- If E is a variable or constant, then the postfix notation for E is E itself
- If E is an expression of the form E1 op E2, then the postfix notation for E is E1 E2 op.
- If E is an expression of the form (E), then the postfix notation for E is the same as the postfix notation E.
- For unary operation –E, the postfix notation is E-
- Example, the postfix notation for the expression 9 – (5 + 2) is 9 5 2 + -

4. Three-Address Code

Three-address code is a sequence of statements having the general form:

x := y op z

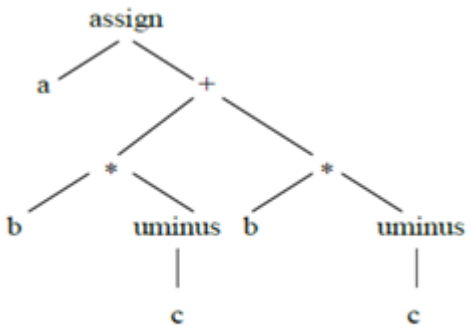
where, x, y and z are names, constants or compiler-generated temporaries and op stands for any operator. Three-address code is a linearized representation of a syntax tree or a DAG in which the explicit names corresponds to the interior nodes of the graph.

Thus a source language expression like x + y * z can be translated into a sequence of statements given below:
t1 : = y * z
t2 := x + t1
where t1 and t2 are compiler-generated temporaries.

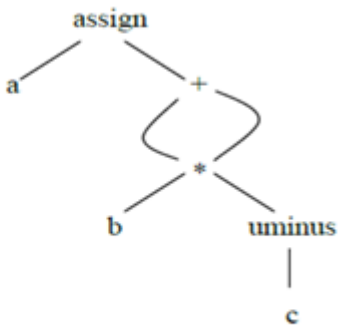
Advantages of three-address code:

- Unravelling 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, (i.e., the usage of temporary variables), allows three-address code to be easily rearranged, unlike postfix notation.

The three-address code corresponding to the syntax tree and DAG of the expression a := b * - c + b * - c are:



(a) Syntax tree



(b) Dag

$t_1 := - c$

$t_2 := b * t_1$

$t_3 := - c$

$t_4 := b * t_3$

$t_5 := t_2 + t_4$

$a := t_5$

(a) Code for the syntax tree

$t_1 := -c$

$t_2 := b * t_1$

$t_5 := t_2 + t_2$

$a := t_5$

(b) Code for the dag

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three-Address Statements:

1. **Assignment Statements**
   Assignment statements of the form **x := y op z**, where op is a binary arithmetic or logical operation

2. **Assignment Instructions**
   Assignment instructions of the form **x := op y**, where op is a unary operation.

3. **Copy Statements** of the form **x := y**, where the value of y is assigned to x

4. **Unconditional Jump**
   The unconditional jumps, **goto L** . the three-address statement with label L is executed next.

5. **Conditional Jumps** such as **if x relop y goto L**, which apply a relational operator (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if the condition is satisfied. If not, the three-address instruction following if x relop y goto L is executed next, in sequence.

6. **Conditional jumps** of the form **if x goto L** and **if False x goto L**. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

7. **Procedure calls and returns**
   Procedure calls and returns are implemented using the following instructions: param x for parameters; call p, n and y = call p, n for procedure and function calls, respectively; and return y, where y, representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

```
param x1
param x2
...
param xn
call p, n
```

   generated as part of a call of the procedure p(x1, x2, ... , xn).

8. **Indexed copy instructions** of the form **x = y[i]** and **x[i] = y**. The instruction x=y[i] sets x to the value in the location i memory units beyond location y. The instruction x[i]=y sets the contents of the location i units beyond x to the value of y.

9. **Address and pointer assignments** of the form **x = &y**, **x = *y**, and ***** **x = y**. The instruction x = &y sets the r-value of x to be the location (l-value) of y. In the instruction x = * y, y is a pointer or a temporary whose r-value is a location. The r-value of x is made equal to the contents of that location. Finally, * x = y sets the r-value of the object pointed to by x to the r-value of y.

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

1. Quadruples
2. Triples
3. Indirect Triples

Quadruples:

A *quadruple* (or just "*quad*") has four fields, which we call *op*, arg_1, arg_2, and *result*. The *op* field contains an internal code for the operator. For instance, the three-address instruction $x = y + z$ is represented by placing + in *op*, y in arg_1, z in arg_2, and x in *result*. The following are some exceptions to this rule:

1. Instructions with unary operators like x = `minus` y or $x = y$ do not use arg_2. Note that for a copy statement like $x = y$, *op* is =, while for most other operations, the assignment operator is implied.
2. Operators like `param` use neither arg_2 nor *result*.
3. Conditional and unconditional jumps put the target label in *result*.

For example, the three-address code for the statement a := b * - c + b * - c and its corresponding quadruple implementation is given below. Here we use the term "minus" to distinguish the unary minus operator from the binary minus operator (-).

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
	...			

(b) Quadruples

Triples:

A triple has only three fields: op, arg1 and arg2. Note that the result field in quadruple is used primarily for temporary names. Using triples, we refer to the result of an operation x op y by its position, rather than by explicit temporary names. Thus instead of the temporary t1 in the quadruple of our previous example, a triple representation would refer to position (0). Parenthesized numbers represent pointers to the triple structure itself. The positions or pointers to positions are called value numbers. The triple implementation of the above three address code is given below.

	op	*arg 1*	*arg 2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

A ternary operation like x[i] = y requires two entries in the triple structure; for example, we can put x and i in one triple and y in the next. Similarly, x = y[i] can implemented by treating it as if it were the two instructions t = y[i] and x = t, where t is a compiler-generated temporary. Note that the temporary t does not actually appear in a triple, since temporary values are referred to by their position in the triple structure.

	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]

A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around. With quadruples, if we move an instruction that computes a temporary t, then the instructions that use t require no change. With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result.