

UNIT IV RUNTIME ENVIRONMENT AND CODE GENERATION

Runtime Environment

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. 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.

1. SOURCE LANGUAGE ISSUES:

1.1 Procedures

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

For example, the following is the 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 within an executable statement, the procedure is said to be called at that point.

1.2 Activation Tree

A program consists of procedures, Variable declaration etc.,

Each execution of the 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.

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 for activation tree

```
Execution begins...
enter readarray
leave readarray
enter quicksort(1,9)
enter partition(1,9)
leave partition(1,9)
enter quicksort(1,3)
...
leave quicksort(1,3)
enter quicksort(5,9)
...
leave quicksort(5,9)
leave quicksort(1,9)
Execution terminated.
```

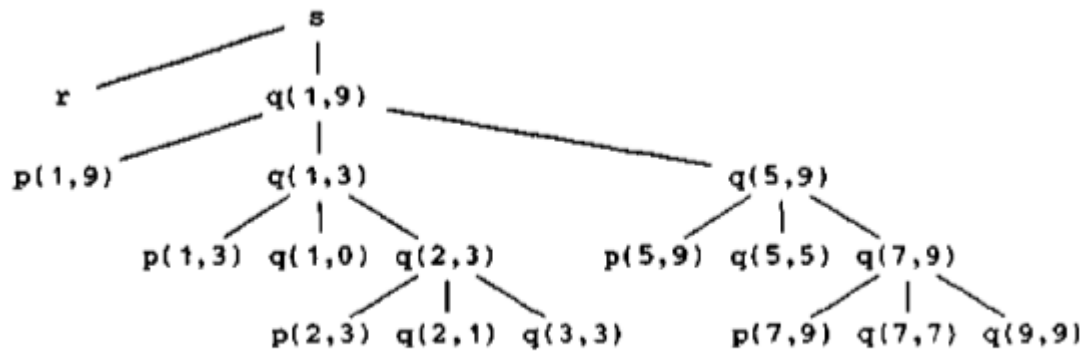


Fig. 7.3. An activation tree corresponding to the output in Fig. 7.2.

1.3 Control Stacks and Activation Record

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 or frame.

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.

Return value
Actual Parameters
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries

Activation Record

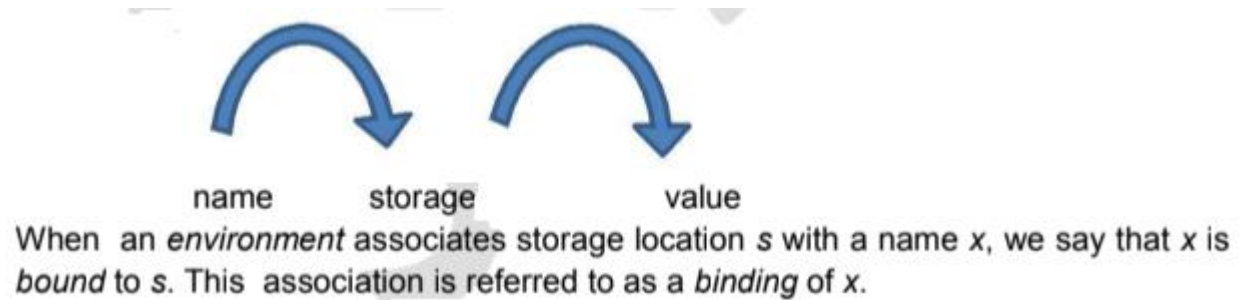
A general activation record consists of the following things:

- **Local variables:** hold the data that is local to the execution of the procedure.
- **Temporary values:** stores the values that arise in the evaluation of an expression.
- **Machine status:** holds the information about the status of the machine just before the function call.
- **Access link** (optional): refers to non-local data held in other activation records.
- **Control link** (optional): points to activation record of caller.
- **Return value:** used by the called procedure to return a value to calling procedure
- **Actual parameters**-actual value we are passing through function calling

1.4 Scope of Declaration

The portion of the program to which a declaration applies is called the scope of that declaration.

1.5 Binding of Names



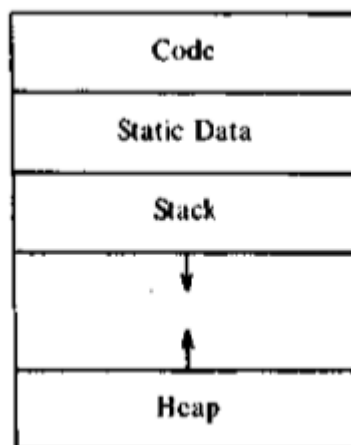
2.STORAGE ORGANIZATION

- 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 throughout memory.

1.2 Sub Division of Run Time Memory

Runtime storage divided into

- 1.General Target Code
- 2.Data Objects
- 3.Control Stack to keep track of procedure activation.



2.2 Activation Record

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 or frame.

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.

Each live activation has an activation record on the 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.

Return value
Actual Parameters
Control Link
Access Link
Saved Machine Status
Local Data
Temporaries

Activation Record

A general activation record consists of the following things:

- **Local variables:** hold the data that is local to the execution of the procedure.
- **Temporary values:** stores the values that arise in the evaluation of an expression.
- **Machine status:** holds the information about the status of the machine just before the function call.
- **Access link** (optional): refers to non-local data held in other activation records.
- **Control link** (optional): points to activation record of caller.
- **Return value:** used by the called procedure to return a value to calling procedure
- **Actual parameters**-actual value we are passing through function calling

2.3 Compile Time Layout of Local Data

The field for local data is laid out as the declarations in a procedure are examined at compile time. Variable-length data is kept outside this field. We keep a count of the memory locations that have been allocated for previous declarations. From the count we determine a *relative* address of the storage for a local with respect to some position such as the beginning of the activation record. The relative address, or *offset*, is the difference between the addresses of the position and the data object.

3.STORAGE ALLOCATION STRATEGIES

The different ways to allocate memory are:

- Static storage allocation
- Stack storage allocation
- Heap storage allocation

3.1 Static Storage Allocation

- In static allocation, names are bound to storage locations.

- If memory is created at compile time, then the memory will be created in static area and only once.
- Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
- The **drawback** with static storage allocation is that the size and position of data objects should be known at compile time.
- Another **drawback** is restriction of the recursion procedure.
- Another **drawback** is the data structures cannot be created dynamically.

3.2 Stack Storage Allocation

- In static storage allocation, storage is organized as a stack.
- An activation record is pushed into the stack when activation begins and it is popped when the activation ends.
- Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
- It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

Stack vs Heap Allocation

Stack	Heap
<ul style="list-style-type: none"> • Allocation/deallocation is automatic • Less expensive • Space for allocation is limited 	<ul style="list-style-type: none"> • Allocation/deallocation is explicit • More expensive • Challenge is fragmentation

Static vs Dynamic Allocation

Static	Dynamic
<ul style="list-style-type: none"> • Variable access is fast <ul style="list-style-type: none"> • Addresses are known at compile time • Cannot support recursion 	<ul style="list-style-type: none"> • Variable access is slow <ul style="list-style-type: none"> • Accesses need redirection through stack/heap pointer • Supports recursion


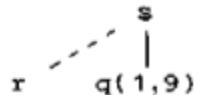
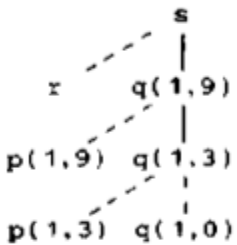
POSITION IN ACTIVATION TREE	ACTIVATION RECORDS ON THE STACK	REMARKS
s	<div style="border: 1px solid black; padding: 5px; margin: 5px;"> s <hr style="border-top: 1px dashed black;"/> a : array </div>	Frame for s
	<div style="border: 1px solid black; padding: 5px; margin: 5px;"> s <hr style="border-top: 1px dashed black;"/> a : array <hr style="border-top: 1px dashed black;"/> r <hr style="border-top: 1px dashed black;"/> i : integer </div>	r is activated
	<div style="border: 1px solid black; padding: 5px; margin: 5px;"> s <hr style="border-top: 1px dashed black;"/> a : array <hr style="border-top: 1px dashed black;"/> q(1,9) <hr style="border-top: 1px dashed black;"/> i : integer </div>	Frame for r has been popped and q(1,9) pushed
	<div style="border: 1px solid black; padding: 5px; margin: 5px;"> s <hr style="border-top: 1px dashed black;"/> a : array <hr style="border-top: 1px dashed black;"/> q(1,9) <hr style="border-top: 1px dashed black;"/> i : integer <hr style="border-top: 1px dashed black;"/> q(1,3) <hr style="border-top: 1px dashed black;"/> i : integer </div>	Control has just returned to q(1,3)

Fig. 7.13. Downward-growing stack allocation of activation records.

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

- The caller evaluates the actual parameters.
- 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.
- The callee saves the register values and other status information. The callee initializes its local data and begins execution.

A suitable, corresponding return sequence is:

- The callee places the return value next to the parameters.

- 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.
- 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.

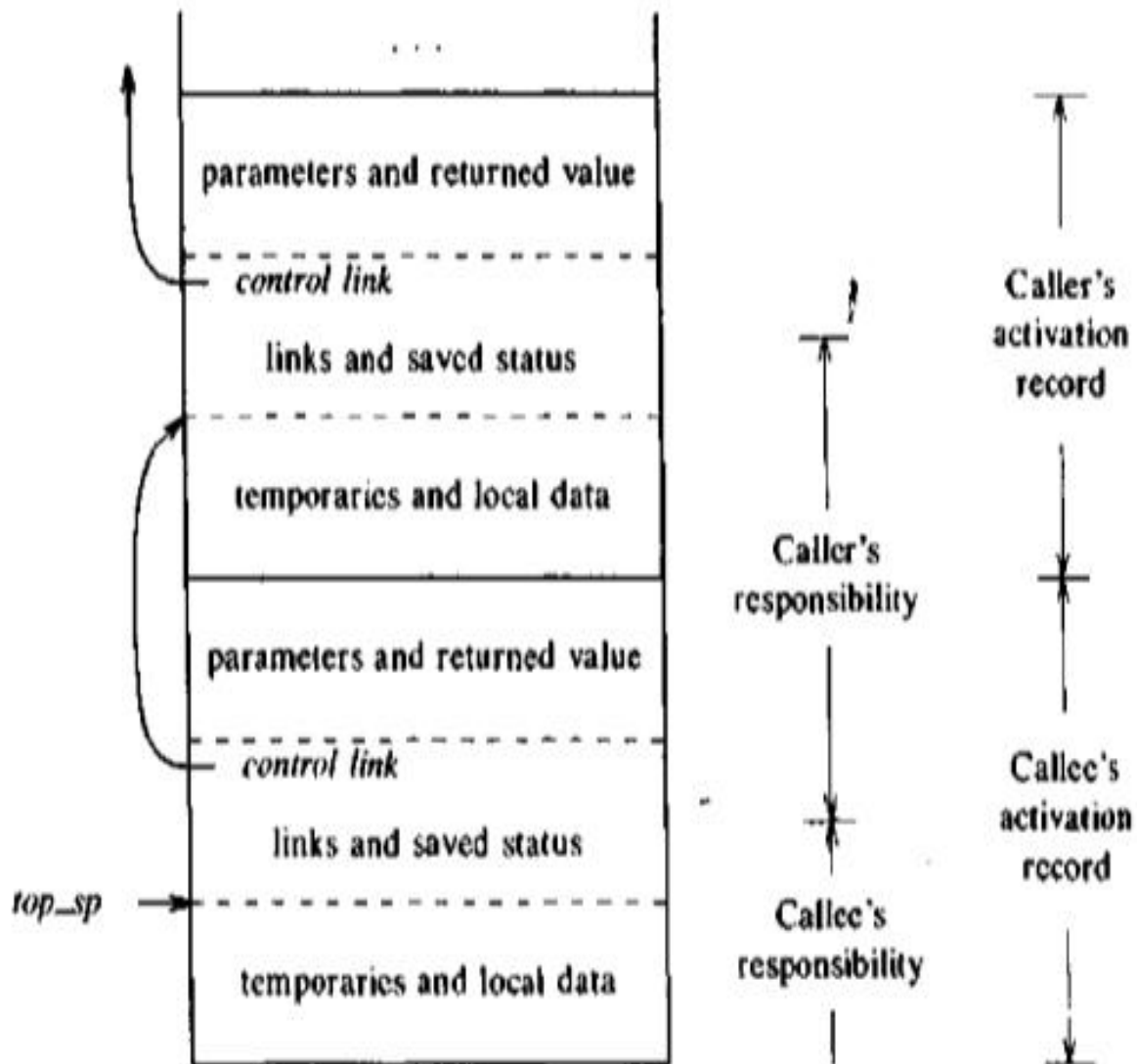


Fig. 7.14. Division of tasks between caller and callee.

Variable-Length Data

A common strategy for handling variable-length data is suggested in Fig. 7.15, where procedure *p* has three local arrays. The storage for these arrays is not part of the activation record for *p*; only a pointer to the beginning of each array appears in the activation record. The relative addresses of these pointers are known at compile time, so the target code can access array elements through the pointers.

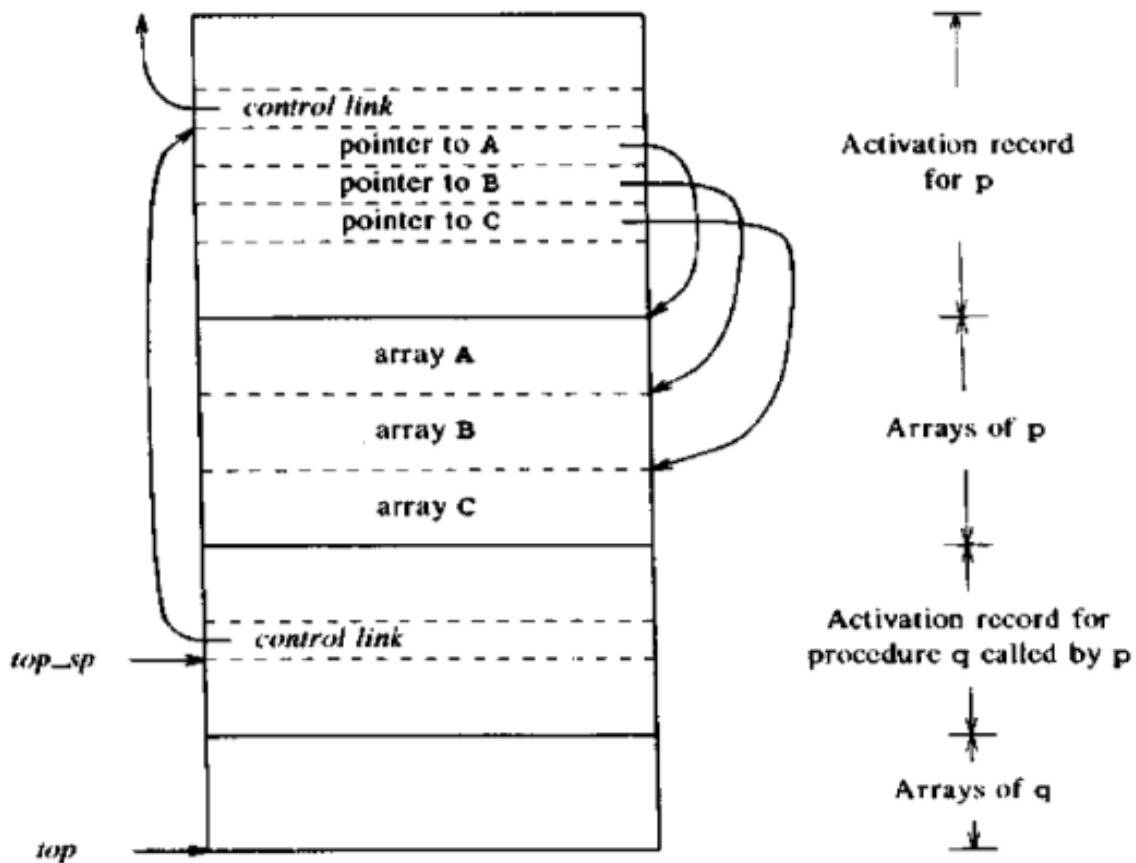


Fig. 7.15. Access to dynamically allocated arrays.

3.3 Heap Storage Allocation

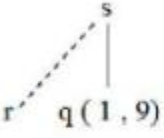
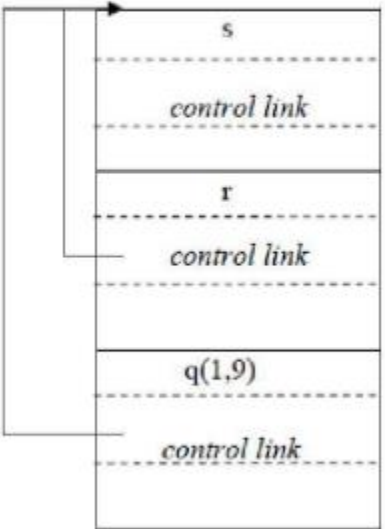
Stack allocation strategy cannot be used if either of the following is possible:

1. The values of local names must be retained when activation ends.
2. A called activation outlives the caller.

That time we are using Heap Allocation

- Heap allocation is the most flexible allocation scheme.
- Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
- Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.

- Heap storage allocation supports the recursion process.

Position in the activation tree	Activation records in the heap	Remarks
		Retained activation record for r

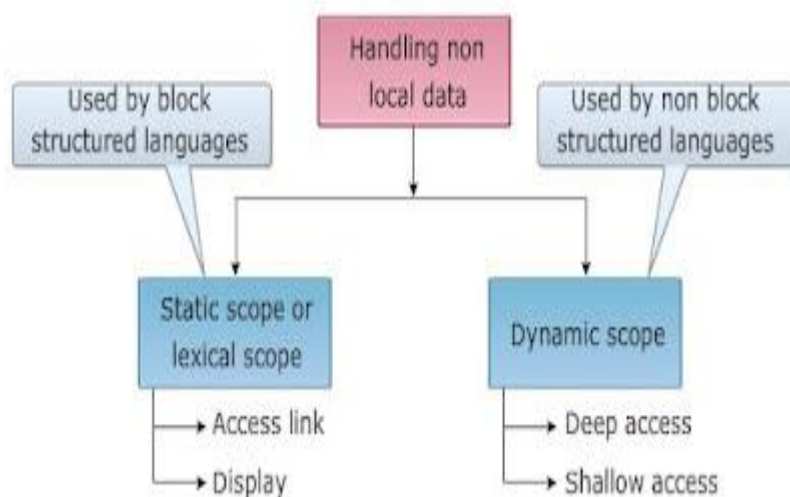
Records for Live Activation need not to be adjacent in a Heap

4. ACCESS TO NON LOCAL NAMES

- In some cases, when a procedure refer to variables that are not local to it, then such variables are called nonlocal variables
- There are two types of scope rules, for the non-local names. They are

Static scope

Dynamic scope



4.1 Static Scope or Lexical Scope

- Lexical scope is also called static scope. In this type of scope, the scope is verified by examining the text of the program.
- Examples: PASCAL, C and ADA are the languages that use the static scope rule.
- These languages are also called block structured languages

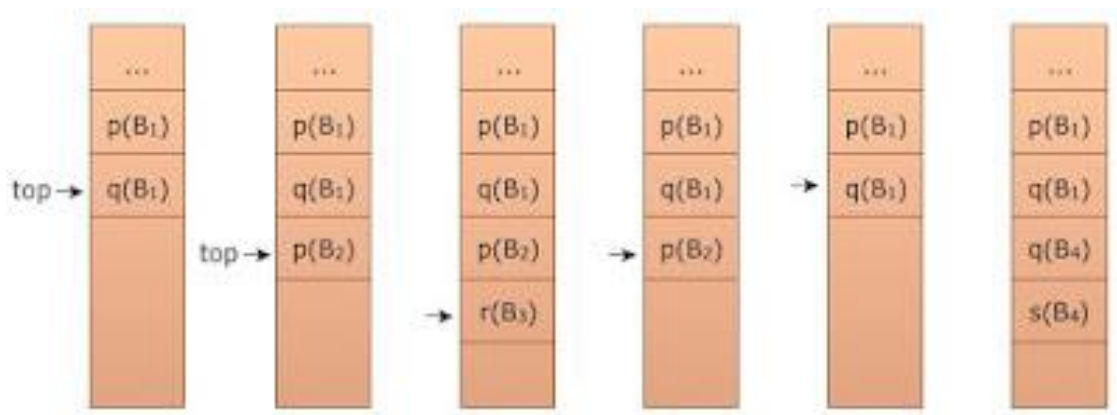
4.2 Block

- A block defines a new scope with a sequence of statements that contains the local data declarations. It is enclosed within the delimiters.

Example:

```
{  
  Declaration statements  
  .....  
}
```

- The beginning and end of the block are specified by the delimiter. The blocks can be in nesting fashion that means block B2 completely can be inside the block B1
- In a block structured language, scope declaration is given by static rule or most closely nested loop
- At a program point, declarations are visible
 - The declarations that are made inside the procedure
 - The names of all enclosing procedures
 - The declarations of names made immediately within such procedures
- The displayed image on the screen shows the storage for the names corresponding to particular block
- Thus, block structure storage allocation can be done by stack



4.3 Lexical Scope for Nested Procedure

- If a procedure is declared inside another procedure then that procedure is known as nested procedure
- A procedure pi, can call any procedure, i.e., its direct ancestor or older siblings of its direct ancestor

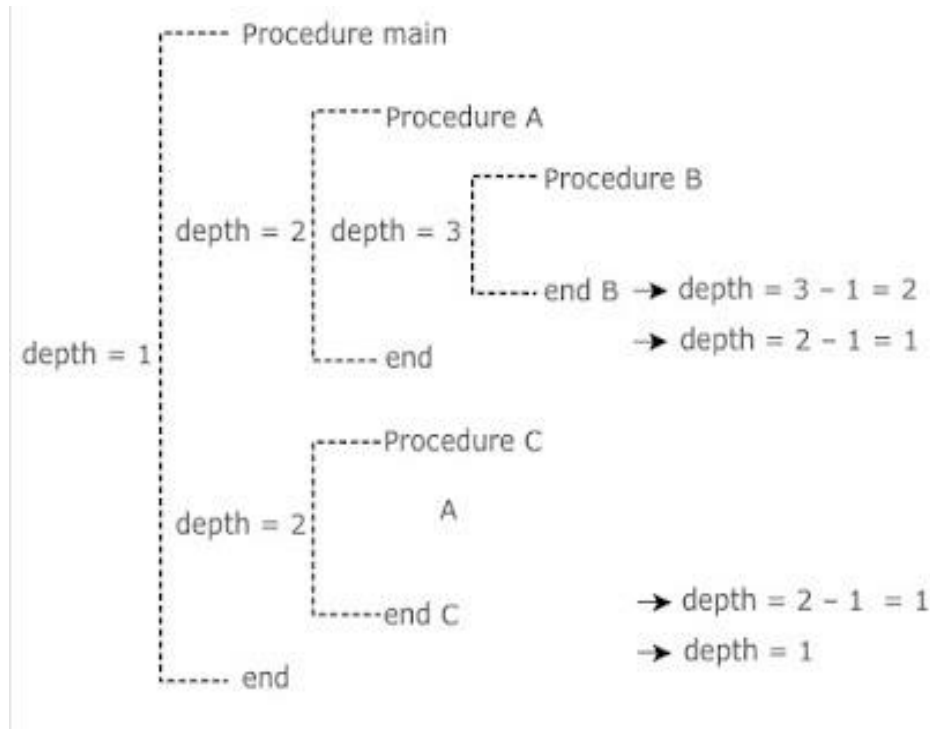
Procedure main

Procedure P1

Procedure P2

Procedure P3

Procedure P4



4.4 Nesting Depth:

- Lexical scope can be implemented by using nesting depth of a procedure. The procedure of calculating nesting depth is as follows:

The main programs nesting depth is '1'

When a new procedure begins, add '1' to nesting depth each time

When you exit from a nested procedure, subtract '1' from depth each time

The variable declared in specific procedure is associated with nesting depth

4.5 Static Scope or Lexical Scope

- The lexical scope can be implemented using access link and displays.

Dynamic Scope

- The dynamic scope allocation rules are used for non-block structured languages
- By considering the current activation, it determines the scope of declaration of the names at runtime

4.6 Access Link:

- Access links are the pointers used in the implementation of lexical scope which is obtained by using pointer to each activation record
- If procedure p is nested within a procedure q then access link of p points to access link or most recent activation record of procedure q

Example: Consider the following piece of code and the runtime stack during execution of the program

program test;

var a: int;

procedure A;

```

var d: int;
{
    a := 1,
}
procedure B(i: int);
var b : int;
procedure C;
var k : int;
{
    A;
}
{
    if(i<>0) then B(i-1)
    else C;
}
{
    B(1);
}

```

Deep Access:

- The idea is to keep a stack of active variables. Use control links instead of access links and to find a variable, search the stack from top to bottom looking for most recent activation record that contains the space for desired variables
- This method of accessing nonlocal variables is called deep access. Since search is made “deep” in the stack, hence the method is called deep access. In this method, a symbol table should be used at runtime

Shallow Access

- The idea to keep a central storage and allot one slot for every variable name
- If the names are not created at runtime, then the storage layout can be fixed at compile time. Otherwise, when new activation procedure occurs, then that procedure changes the storage entries for its local at entry and exit (i.e., while entering in the procedure and while leaving the procedure)

Comparison of Deep and Shallow Access

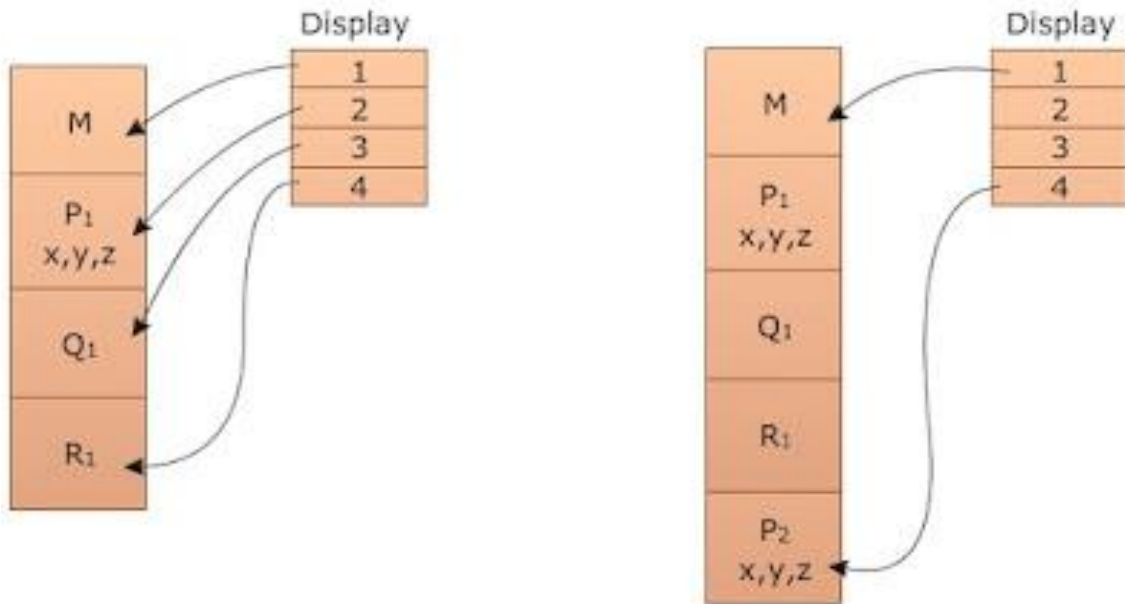
Deep Access	Shallow Access
<ul style="list-style-type: none"> • Deep access takes longer time to access the nonlocals 	<ul style="list-style-type: none"> • Shallow access allows fast access
<ul style="list-style-type: none"> • It needs a symbol table at runtime 	<ul style="list-style-type: none"> • It has a overhead of handling procedure entry and exit

4.7 Displays:

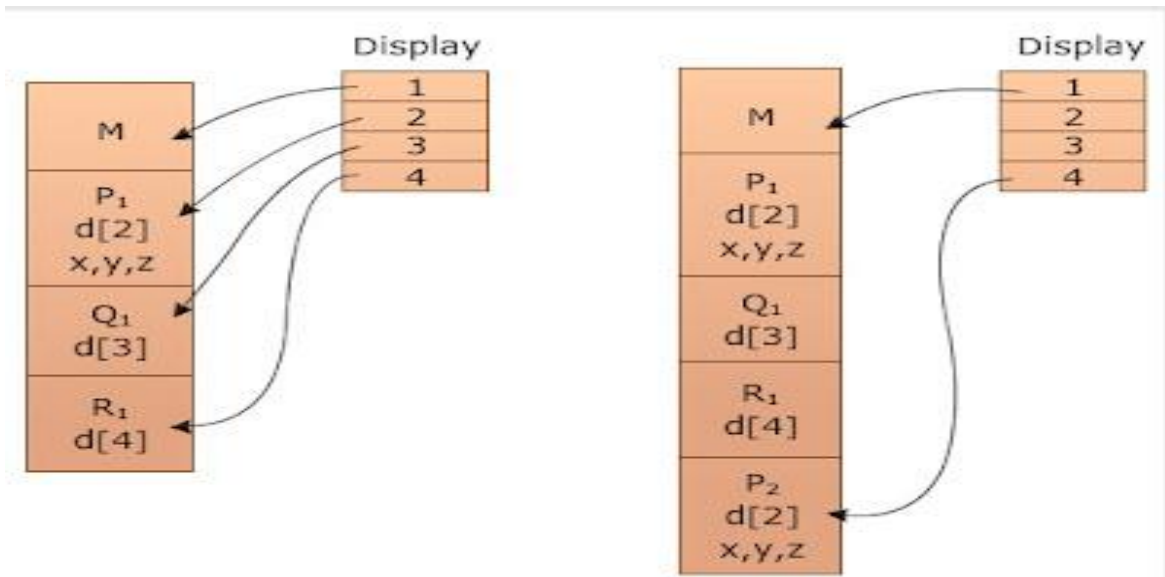
- If access links are used in the search, then the search can be slow
- So, optimization is used to access an activation record from the direct location of the variable without any search
- Display is a global array d of pointers to activation records, indexed by lexical nesting depth. The number of display elements can be known at compiler time

- $d[i]$ is an array element which points to the most recent activation of the block at nesting depth (or lexical level)
- A nonlocal X is found in the following manner:
- Use one array access to find the activation record containing X . if the most-closely nested declaration of X is at nesting depth I , the $d[i]$ points to the activation record containing the location for X
- Use relative address within the activation record

Example:



How to maintain display information?



- When a procedure is called, a procedure 'p' at nesting depth 'i' is setup:
 - Save value of $d[i]$ in activation record for 'p'
 - 'I' set $d[i]$ to point to new activation record
- When a 'p' returns:
 - Reset $d[i]$ to display value stored

5.PARAMETER PASSING

Parameter Passing

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism.

R- value

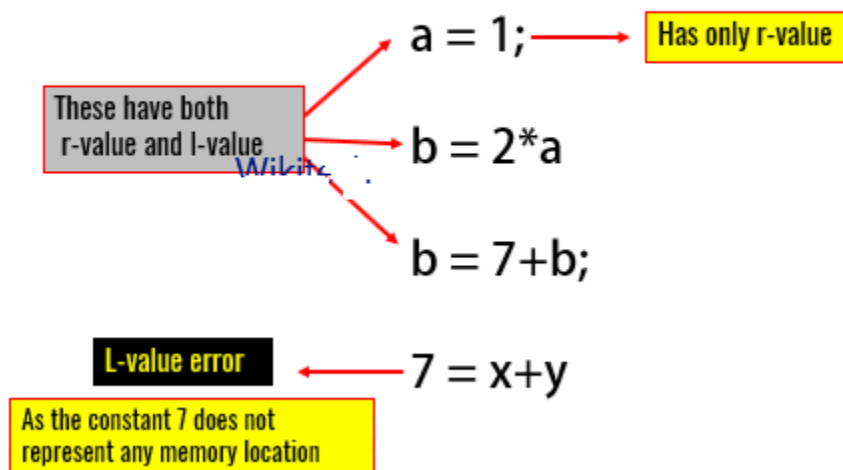
The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if its appear on the right side of the assignment operator.

R-value can always be assigned to some other variable.

L-value

The location of the memory(address) where the expression is stored is known as the l-value of that expression.

It always appears on the left side if the assignment operator.



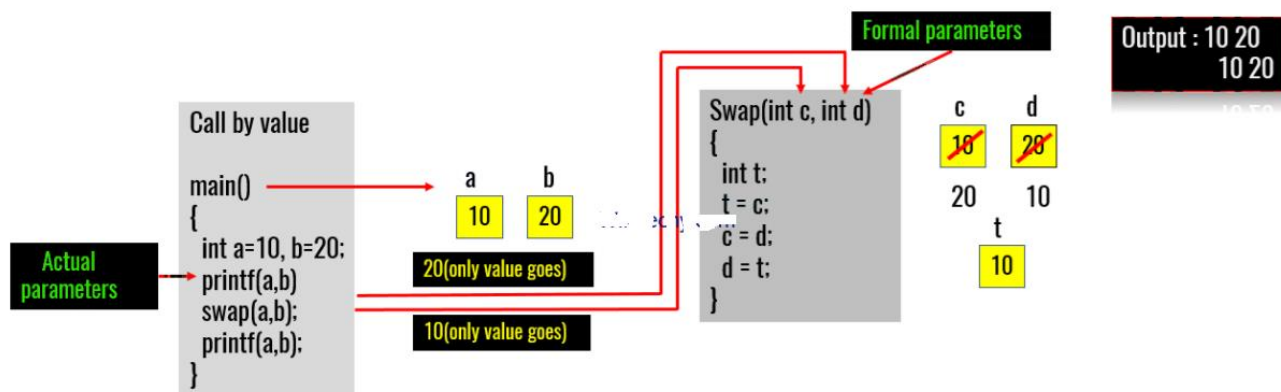
Different ways of passing the parameters to the procedure

- Call by Value
- Call by reference
- Copy restore
- Call by name

5.1 Call by Value

In call by value the calling procedure pass the r-value of the actual parameters and the compiler puts that into called procedure's activation record.

Formal parameters hold the values passed by the calling procedure, thus any changes made in the formal parameters does not affect the actual parameters.



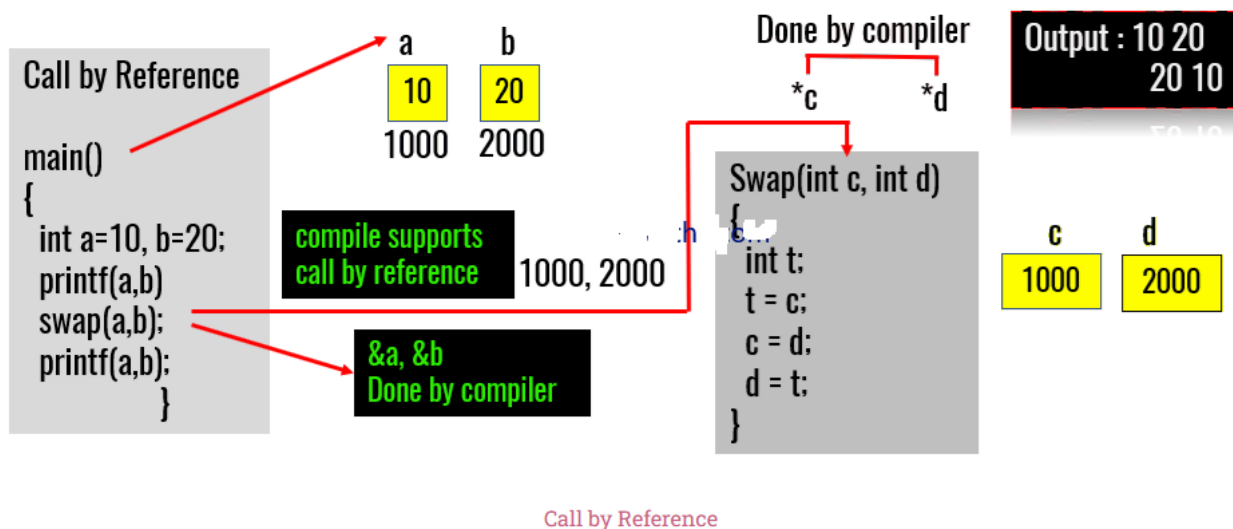
5.2 Call by Reference

In call by reference the formal and actual parameters refers to same memory location.

The l-value of actual parameters is copied to the activation record of the called function. Thus the called function has the address of the actual parameters.

If the actual parameters does not have a l-value (eg- i+3) then it is evaluated in a new temporary location and the address of the location is passed.

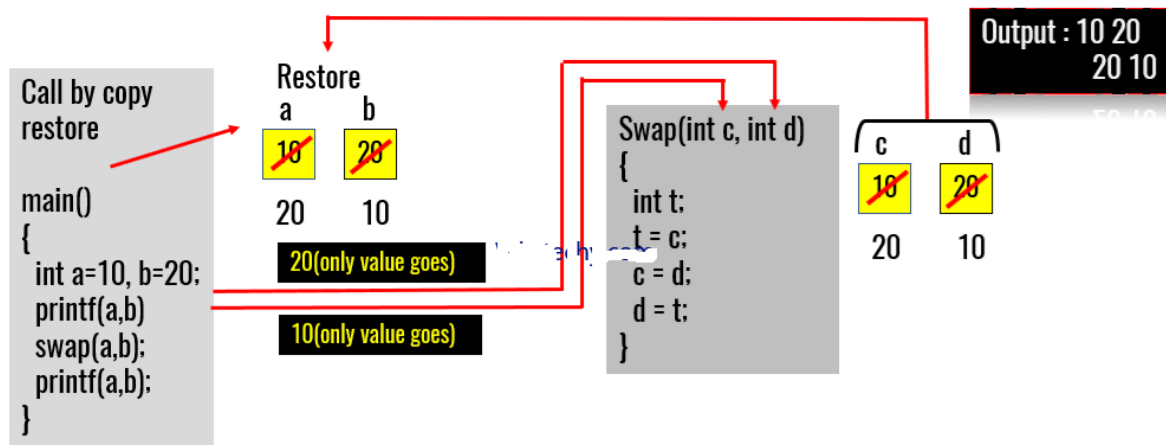
Any changes made in the formal parameter is reflected in the actual parameters (because changes are made at the address).



5.3 Call by Copy Restore

In call by copy restore compiler copies the value in formal parameters when the procedure is called and copy them back in actual parameters when control returns to the called function.

The r-values are passed and on return r-value of formals are copied into l-value of actuals.

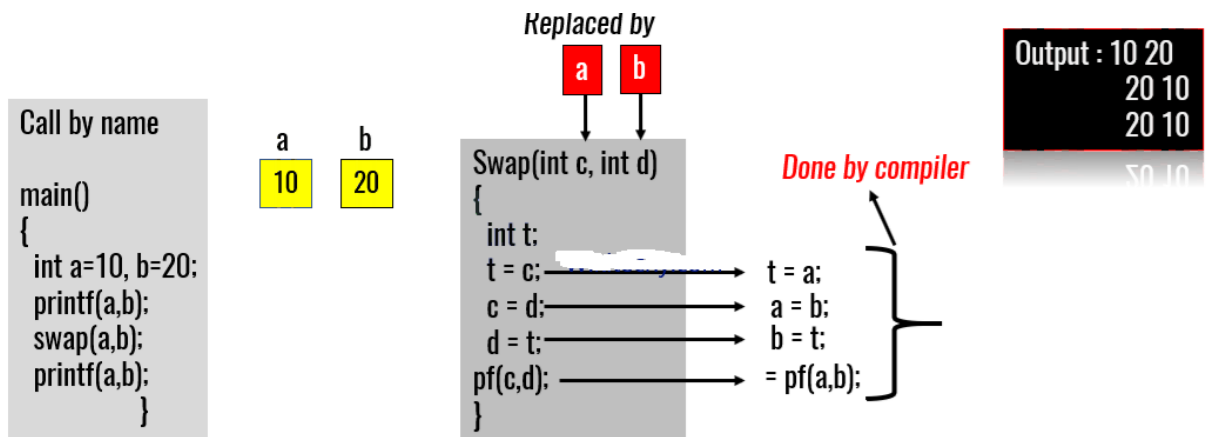


Call by Copy

5.4 Call by Name

In call by name the actual parameters are substituted for formals in all the places formals occur in the procedure.

It is also referred as lazy evaluation because evaluation is done on parameters only when needed.



Call by Name

6. ISSUES IN THE DESIGN OF CODE GENERATOR

The following issues arise during the code generation phase :

- **Input to code generator**
- **Target program**
- **Memory management**
- **Instruction selection**
- **Register allocation**
- **Evaluation order**

6.1 Input to code generator

- The input to the code generation consists of the intermediate representation of the source program produced by front end together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be :

- **Linear representation such as postfix notation**
- **Three address representation such as quadruples**
- **Virtual machine representation such as stack machine code**
- **Graphical representations such as syntax trees and DAGS.**

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

6.2 Target program:

The output of the code generator is the target program. The output may be :

1. Absolute machine language

It can be placed in a fixed memory location and can be executed immediately.

2. Relocatable machine language

It allows subprograms to be compiled separately.

3. Assembly language

Code generation is made easier.

6.3 Memory management:

- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions. For example,

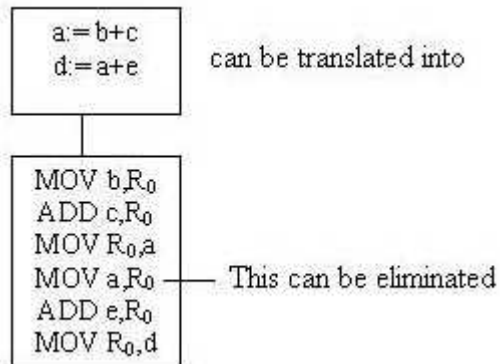
j: goto i generates jump instruction as follows :

1) if $i < j$, a backward jump instruction with target address equal to location of code for quadruple i is generated.

2) If $i > j$, the jump is forward. We must store on a list for quadruple i th location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

6.4 Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.
- The former statement can be translated into the latter statement as shown below:



In the above example there are two move operations contains the same values in the register and we can eliminate one instruction from that in order to increase the size and speed.

6.5 Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two sub problems :

Register allocation– the set of variables that will reside in registers at a point in the program is selected.

Register assignment– the specific register that a variable will reside in is picked.

- Certain machine requires even-odd register pairs for some operands and results. For example , consider the division instruction of the form :

D x, y

where, x – dividend even register in even/odd register pair y – divisor even register holds the remainder odd register holds the quotient

Example:

t = a + b

Assembly language for the above three address code statement is(Allocating Registers and we are picking the value from the register.

MOV R1,a

```
MOV R2,b
ADD R1,R2
STR t,R1
```

6.6 Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code.

Some computation orders require fewer registers to hold intermediate results than others.

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

The three-address code, the corresponding code and its reordered instruction are given below:

Three-address code	Code	Reordered three-address code	Code	Inference
$t1 := a+b$ $t2 := c+d$ $t3 := e*t2$ $t4 := t1-t3$	<pre>MOV a,R0 ADD b,R0 MOV R0,t1 MOV c,R1 ADD d,R1 MOV e,R0 MUL R1,R0 MOV t1,R1 SUB R0,R1 MOV R1,t4</pre>	$t2 := c+d$ $t3 := e*t2$ $t1 := a+b$ $t4 := t1-t3$	<pre>MOV c,R0 ADD d,R0 MOV e,R1 MUL R0,R1 MOV a,R0 ADD b,R0 SUB R1,R0 MOV R0,t4</pre>	The reordered instructions reduced the number of final code by 2 and thus saved in cost. The three-address code is reordered so that t1 is computed after computing t2 and t3. This reordering has saved in the instruction cost.

7. DESIGN OF SIMPLE CODE GENERATION ALGORITHM

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

Properties in Code Generation:

- Absolute quality code
- Efficient Use of Resources
- Quick Code

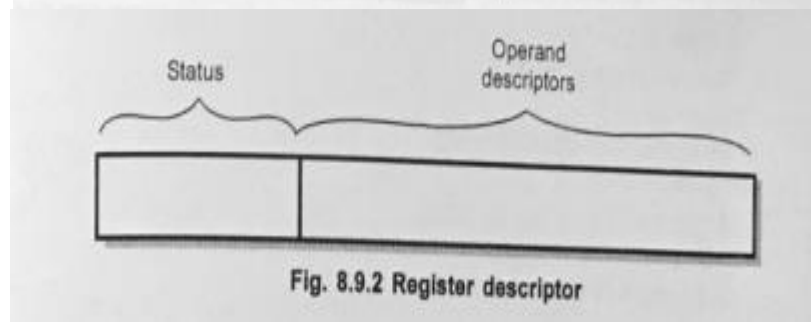
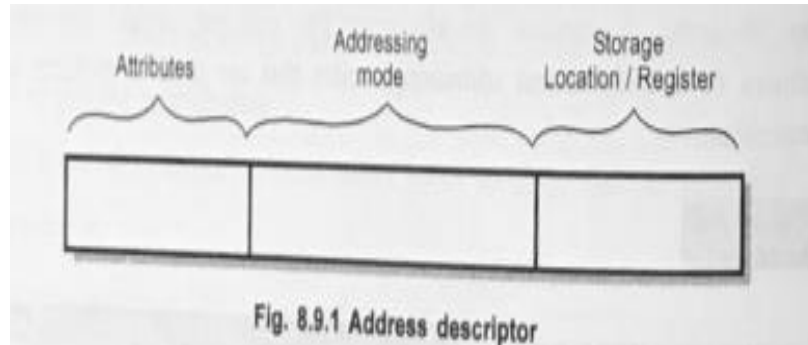
Example:

Consider the three address statement $x := y + z$. It can have the following sequence of codes:

```
MOV x, R0
ADD y, R0
```

7.1 Register and Address Descriptors:

- A register descriptor contains the track of what is currently in each register. The register descriptors show that all the registers are initially empty.
- An address descriptor is used to store the location where current value of the name can be found at run time.



7.2 A code-generation algorithm:

The code generation algorithm is the core of the compiler. It sets up register and address descriptors, then generates machine instructions that give you CPU-level control over your program.

The algorithm is split into four parts:

- Register descriptor set-up
- Basic block generation
- Instruction generation for operations on registers (e.g., addition), and ending the basic block with a jump statement or return command.
- Instruction Scheduling

Function **getReg**:

The algorithm takes a sequence of three-address statements as input. For each three address statement of the form **x := y op z** perform the various actions. These are as follows:

1. Invoke a function **getreg** to find out the location **L** where the result of computation **y op z** should be stored.

2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y'. If the value of y is not already in L then generate the instruction **MOV y'** , L to place a copy of y in L.
3. Generate the instruction **OP z'** , L where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptor.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of **x : = y op z** those register will no longer contain y or z.

Program:

```

Gen_Code(operator,operand1,operand2)
{
    if (operand1.addressmode = 'R')
    {
        if (operator = '+')
            Generate('ADD operand2,R0');
        else if(operator = '-')
            Generate('SUB operand2,R0');
        else if(operator = '*')
            Generate('MUL operand2,R0');
        else if(operator = '/')
            Generate('DIV operand2,R0');
    }
    else if (operand2.addressmode = 'R')
    {
        if (operator = '+')
            Generate('ADD operand1,R0');
        else if(operator = '-')
            Generate('SUB operand1,R0');
        else if(operator = '*')
            Generate('MUL operand1,R0');
        else if(operator = '/')
    }
}

```

```

        Generate('DIV operand1,R0');
    }
    else
    {
        Generate('MOV operand2,R0');
        if (operator == '+')
            Generate('ADD operand2,R0');
        else if(operator == '-')
            Generate('SUB operand2,R0');
        else if(operator == '*')
            Generate('MUL operand2,R0');
        else if(operator == '/')
            Generate('DIV operand2,R0');
    }
}

```

Generating Code for Assignment Statements:

The assignment statement **d := (a-b) + (a-c) + (a-c)** can be translated into the following sequence of three address code:

t := a-b

u := a-c

v := t +u

d := v+u

Code sequence for the example is as follows:

Statements	Code Generated	Register descriptor Register empty	Address descriptor
t := a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u := a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d := v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Operand Descriptor

t	R(Addressing Mode)	R0
u	R	R1
v	R	R0
d	R	R0

Generating code for Indexed Assignments

Statements	Code Generated	Cost
$a := b[i]$	MOV b(Ri), R	2
$a[i] := b$	MOV b, a(Ri)	3

Generating code for Pointer Assignments

Statements	Code Generated	Cost
$a := *p$	MOV *Rp, a	2
$*p := a$	MOV a, *Rp	2

Generating code for Conditional Statements

Consider the following example:

$x := y + z$ if $x < 0$ goto z

The following would be the target code

MOV y, R0

ADD z, R0

MOV R0, x //x is the condition code CJ < z

8. PROGRAM AND INSTRUCTION COST:

- The instruction LD RO, R1 copies the contents of register R1 into register RO. This instruction has a cost of one because no additional memory words are required.
- The instruction LD RO, M loads the contents of memory location M into register RO. The cost is two since the address of memory location M is in the word following the instruction.
- The instruction LD R1, *100(R2) loads into register R1 the value given by contents(contents(100 + contents(K2))). The cost is three because the constant 100 is stored in the word following the instruction.

For example: consider the three-address statement $a := b + c$. It can have the following sequence of codes:

ADD Rj, Ri Cost = 1

(or)

ADD c, Ri Cost = 2

(or)

MOV c, Rj Cost = 3

ADD Rj, Ri

Instruction costs : Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.

Address modes involving registers have cost zero. Address modes involving memory location or literal have cost one. Instruction length should be minimized if space is important.

Doing so also minimizes the time taken to fetch and perform the instruction.

For example :

1. The instruction MOV R0, R1 copies the contents of register R0 into R1. It has cost 7 one, since it occupies only one word of memory.

2. The (store) instruction MOV R5,M copies the contents of register R5 into memory location M. This instruction has cost two, since the address of memory location M is in the word following the instruction.

3. The instruction ADD #1,R3 adds the constant 1 to the contents of register 3, and has cost two, since the constant 1 must appear in the next word following the instruction.

4. The instruction SUB 4(R0),*12(R1) stores the value contents(contents(12+contents(R1)))-contents(contents(4+R0)) into the destination *12(R1). Cost of this instruction is three, since the constant 4 and 12 are stored in the next two words following the instruction.

For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.

The three-address statement $a := b + c$ can be implemented by many different instruction sequences :

MOV b, R0

ADD c, R0

MOV R0, a

cost = 6

MOV b, a

ADD c, a

cost = 6

Assuming R0, R1 and R2 contain the addresses of a, b, and c :

MOV *R1, *R0

ADD *R2, *R0

cost = 2

In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.