UNIT 5
CODE OPTIMIZATION

5.1 INTRODUCTION

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.
- Optimizations are classified into two categories. They are
  - Machine independent optimizations:
  - Machine dependant optimizations:

Machine independent optimizations:

- Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependant optimizations:

- Machine dependant optimizations are based on register allocation and utilization of special machine- instruction sequences.

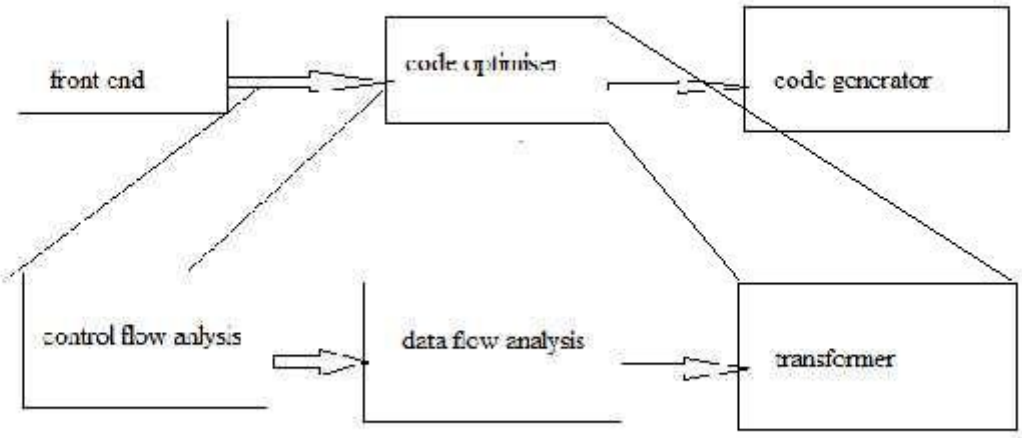
The criteria for code improvement transformations:

- Simply stated, the best program transformations are those that yield the most benefit for the least effort.

- The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

- A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "optimization" may slow down a program slightly.
- The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. "Peephole" transformations of this kind are simple enough and beneficial enough to be included in any compiler.

Organization for an Optimizing Compiler:



- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flow analysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as
  - control flow graph
  - Call graph
- Data flow analysis (DFA) is the process of ascerting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

5.2 PRINCIPAL SOURCES OF OPTIMISATION

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- The transformations
  - Common sub expression elimination,
  - Copy propagation,
  - Dead-code elimination, and
  - Constant folding

are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

Common Sub expressions elimination:

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid re-computing the expression if we can use the previously computed value.

For example

t_1: =4*i

t_2: =a [t1]

t_3: =4*j

t_4:=4*i

t_5: =n

t_6: =b [t_4] +t_5

The above code can be optimized using the common sub-expression elimination as

t_1: =4*i

t_2: =a [t_1]

t_3: =4*j

t_5: =n

t_6: =b [t_1] +t_5

The common sub expression t_4: =4*i is eliminated as its computation is already in t_1. And value of i is not been changed from definition to use.

Copy Propagation:

- Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.
- For example: x=Pi;

  ……

  A=x*r*r;

  The optimization using copy propagation can be done as follows:

  A=Pi*r*r;

  Here the variable x is eliminated

Dead-Code Eliminations:

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous

transformations. An optimization can be done byeliminating dead code.

Example:

```
i=0;
if(i=1)
{
 a=b+5;
}
```

Here, 'if'statement is dead code because this condition will never get satisfied.

Constant folding:

It is also possible to eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation.

Loop Optimizations:

Let us give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

- ✓ code motion, which moves code outside a loop;
- ✓ Induction -variable elimination, which we apply to replace variables from inner loop.
- ✓ Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion:

- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion "before the loop" assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2) /* statement does not change limit*/
```

Code motion will result in the equivalent of

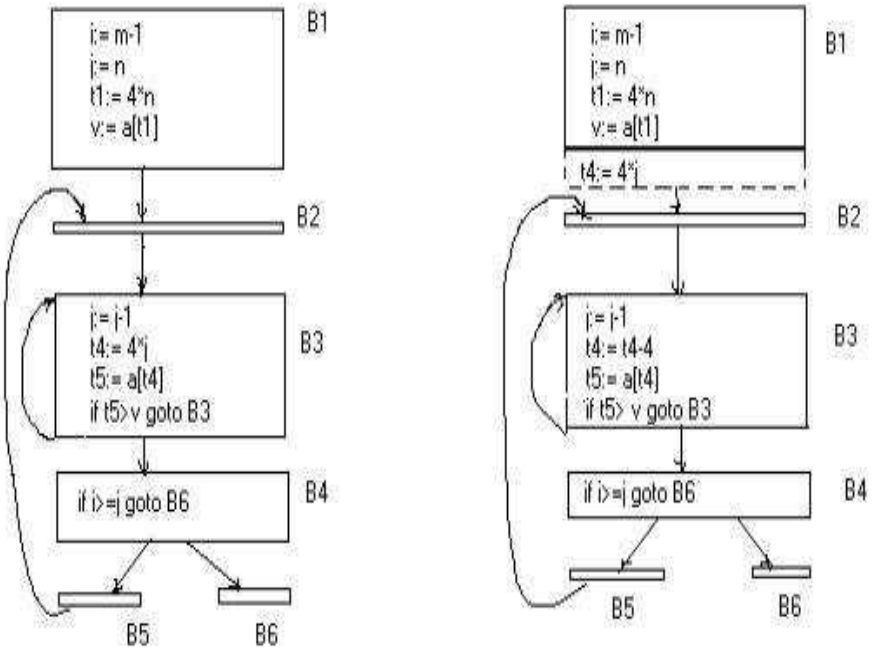
```
t= limit-2;
while (i<=t)  /* statement does not change limit or t */
```

Induction Variables :

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

  Example:

  As the relationship t_4:=4*j surely holds after such an assignment to t_4 in Fig. and t_4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j -1 the relationship t_4:= 4*j-4 must hold. We may therefore replace the assignment t_4:= 4*j by t_4:= t_4-4. The only problem is that t_4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t_4=4*j on entry to the block B3, we place an initializations of t_4 at the end of the block where j itself is



before **after**

initialized, shown by the dashed addition to block B1 in second Fig.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Reduction In Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

5.3 OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are :

- Structure -Preserving Transformations
- Algebraic Transformations

Structure- Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expressionelimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

- **Common sub-expression elimination:**

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

a: =b+c
b: =a-d
c: =b+c
d: =a-d

The 2^{nd} and 4^{th} statements compute the same expression: b+c and a-d

Basic block can be transformed to

a: =b+c
b: =a-d
c: =a
d: =b

Dead code elimination:

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error -correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

Renaming of temporary variables:

- A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.

- In this we can transform a basic block to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

Two statements

t_1:=b+c

t_2:=x+y

can be interchanged or reordered in its computation in the basic block when value of t_1 does not affect the value of t_2.

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
- The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

a :=b+c

e :=c+d+b

the following intermediate code may be generated:

a :=b+c

t :=c+d

e:=t+b

Example:

x:=x+0 can be removed

x:=y**2 can be replaced by a cheaper statement x:=y*y

- The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

5.4INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.

- A compiler could take advantage of “reaching definitions” , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.

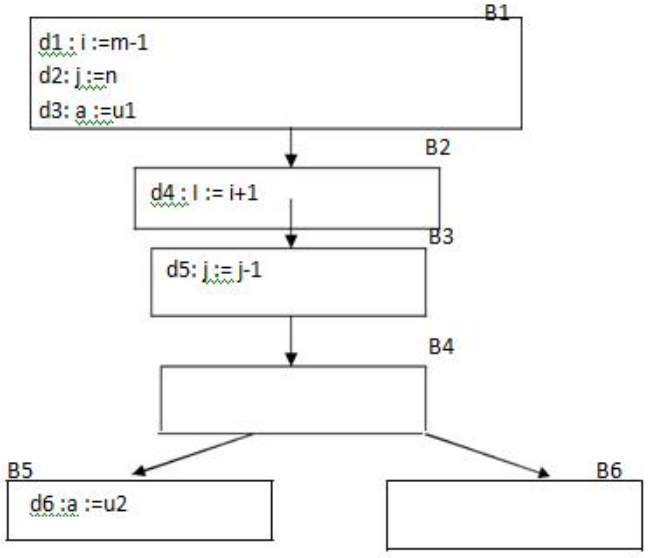
- Data- flow information can be collected by setting up and solving systems of equations of the form :

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

- This equation can be read as " the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement."
- The details of how data-flow equations are set and solved depend on three factors.
- The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].
- Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
- There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:

- Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.



B1

- Now let us take a global view and consider all the points in all the blocks. A path from p_1 to p_n is a sequence of points $p_1, p_2, \ldots, p_n$ such that for each i between 1 and n-1, either
- P_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that statement in the same block, or
- P_i is the end of some block and p_{i+1} is the beginning of a successor block.

Reaching definitions:

- A definition of variable x is a statement that assigns, or may assign, a value to x. The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x.
- These statements certainly define a value for x, and they are referred to as **unambiguous** definitions of x. There are certain kinds of statements that may define a value for x; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:
- A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.
- An assignment through a pointer that could refer to x. For example, the assignment *q: = y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.
- We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

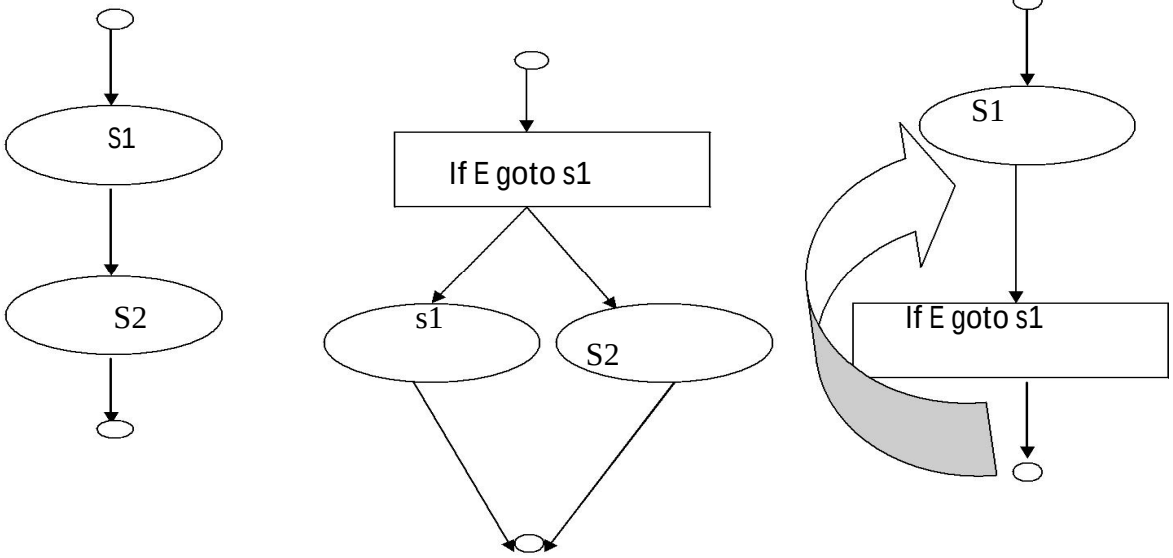
Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

  S ⟶ id: = E| S; S | if E then S else S | do S

  while E E ⟶ id + id| id

- S2Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.



S1 ; S2 IF E then S1 else S2 do S1 while E

- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.

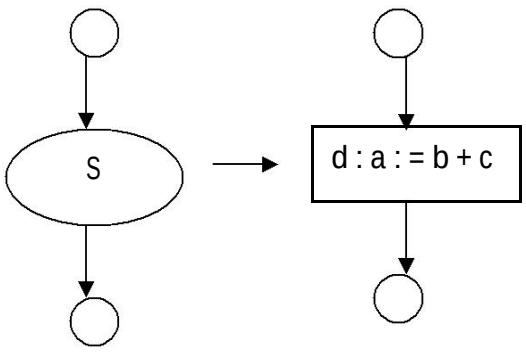
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

✓ We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.

✓ **gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.**

✓ Consider the following data-flow equations for reaching definitions :

i)



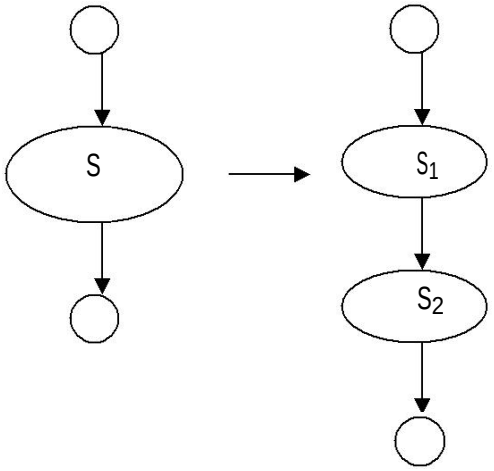
gen [S] = { d }
kill [S] = D_a – { d }
out [S] = gen [S] U (in[S] – kill[S])

- Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus
- Gen[S]={d}
- On the other hand, d "kills" all other definitions of a, so

  we write Kill[S] = D_a – {d}

Where, D_a is the set of all definitions in the program for variable a.

ii)



gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])

Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

in [S_1] = in [S] in [S_2] = out [S_1] out [S] = out [S_2]

- Under what circumstances is definition d generated by S=S_1; S_2? First of all, if it is generated by S_2, then it is surely generated by S. if d is generated by S_1, it will reach the end of S provided it is not killed by S_2. Thus, we write
- gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])
- Similar reasoning applies to the killing of a definition, so

  we have Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

Conservative estimation of data-flow information:

- There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are "uninterpreted"; that is, there exists inputs to the program that make their branches go either way.
- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.
- When we compare the computed gen with the "true" gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.
- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.
- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.
- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached.
- Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.
- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that in[S] be the set of

definitions reaching the beginning of S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

- The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S.
- Assuming we know in[S] we compute out by equation, that is
- Out[S] = gen[S] U (in[S] - kill[S])
- Considering cascade of two statements S_1; S_2, as in the second case. We start by observingin[S_1]=in[S]. Then, we recursively compute out[S_1], which gives us in[S_2], since a definition reaches the beginning of S_2 if and only if it reaches the end of S_1. Now we can compute out[S_2], and this set is equal to out[S].
- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S_1 or S_2 exactly when it reaches the beginning of S.
- In[S_1] = in[S_2] = in[S]
- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,
- Out[S]=out[S_1] U out[S_2]

Representation of sets:

- Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.
- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.
- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A .

|

Local reaching definitions:

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, re-computing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.
- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions ofa.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.
- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.
- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.
- Several approaches may be taken. The iterative method works arbitrary flow

graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods

- However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

5.5 SOURCE LANGUAGE ISSUES

Procedures:

A *procedure definition* is a declaration that associates an identifier with a statement. Theidentifier 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.

Activation trees:

An *activation tree* is used to depict the way control enters and leaves activations. In anactivation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to*b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before thelifetime of *b*.

Control stack:

A *control stack* is used to keep track of live procedure activations. The idea is to push thenode for an activation onto the control stack as the activation begins and to pop the nodewhen the activation ends. 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 control stack, the stack contains the nodes along the pathfrom *n* to the root.

The Scope of a Declaration:

A declaration is a syntactic construct that associates information with a name. Declarations may be explicit, such as:

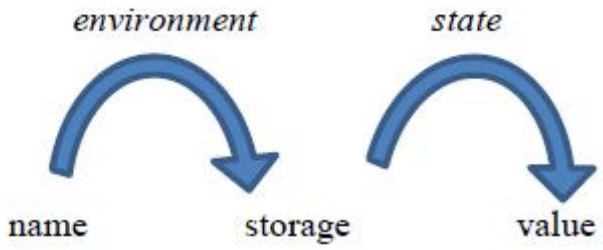
var i : integer ;

or they may be implicit. Example, any variable name starting with I is assumed to denote aninteger.The portion of the 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 differentdata objects at run time. “Data object” corresponds to a storage location that holds 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.

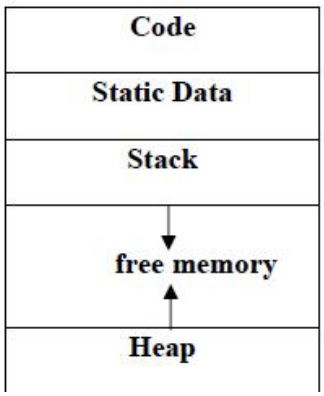


When an *environment* associates storage location *s* with a name *x*, we say that *x* is *bound*to *s*. This association is referred to as a *binding* of *x*.

5.6 STORAGE ORGANISATION

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

Typical subdivision of run-time memory:



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compile may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *controlstack.*
- Each live activation has an activation record on the control stack, with the root of theactivation 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. Thediagram below shows the contents of activation record.
- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.
- Space for the return value of the called functions, if any. Again, not all calledproceduresreturn a value, and if one does, we may prefer to place that value in a register forefficiency.
- The actual parameters used by the calling procedure. These are not placed in activationrecord but rather in registers, when possible, for greater efficiency.

5.7STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

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 deallocates storage as needed at run time from a data areaknown as heap.

STATIC ALLOCATION

- In static allocation, names are bound to storage as the program is compiled, so there is noneed for a run-time support package.
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they werewhen control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

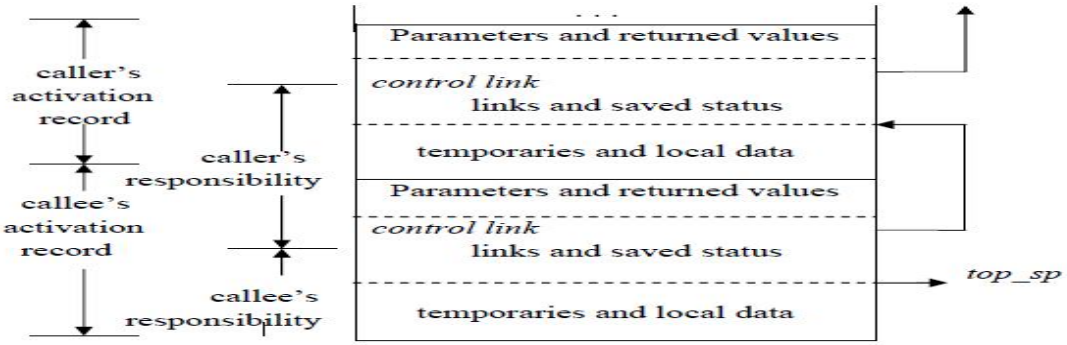
5.7.1STACK ALLOCATION OF SPACE

- All compilers for languages that use procedures, functions or methods as units of userdefinedactions manage at least part of their run-time memory as a stack.

- Each time a procedure is called , space for its local variables is pushed onto a stack, andwhen the procedure terminates, that space is popped off the stack.

Calling sequences:

- Procedures called are implemented in what is called as calling sequence, which consistsof code that allocates an activation record on the stack and enters information into itsfields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) andthe procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
- Values communicated between caller and callee are generally placed at the beginning of the callee's activation record, so they are as close as possible to thecaller's activation record.
- Fixed length items are generally placed in the middle. Such items typically includethe control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of theactivation record. The most common example is dynamically sized array, where thevalue of one of the callee's parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to haveit point to the end of fixed-length fields in the activation record. Fixed-length datacan then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.



Division of tasks between caller and callee

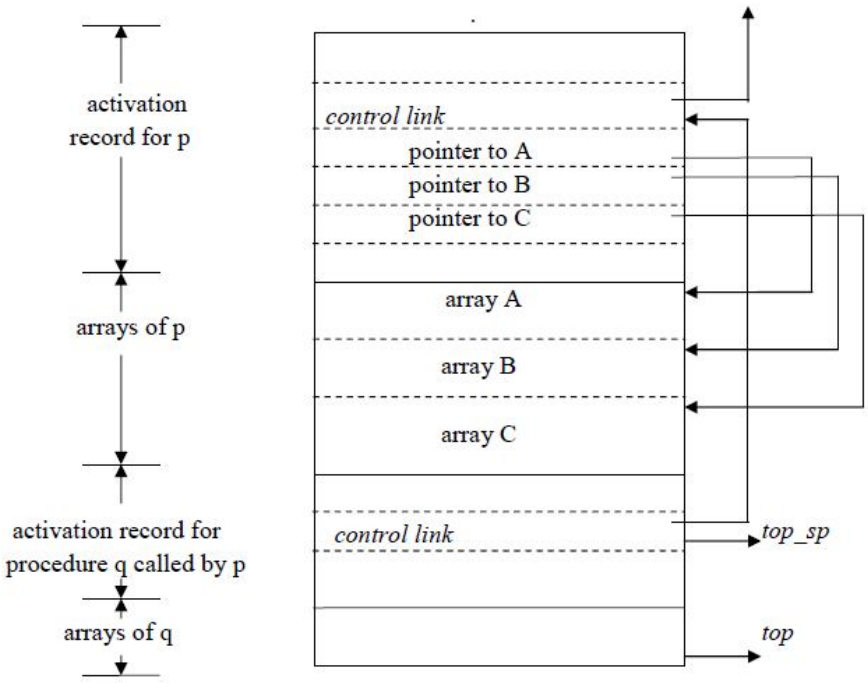
The calling sequence and its division between caller and callee are as follows.

- 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. Parameters and returned values *control link* links and saved status temporaries and local data Parameters and returned values *control link* links and saved status temporaries and local data

Variable length data on 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 the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



Access to dynamically allocated arrays

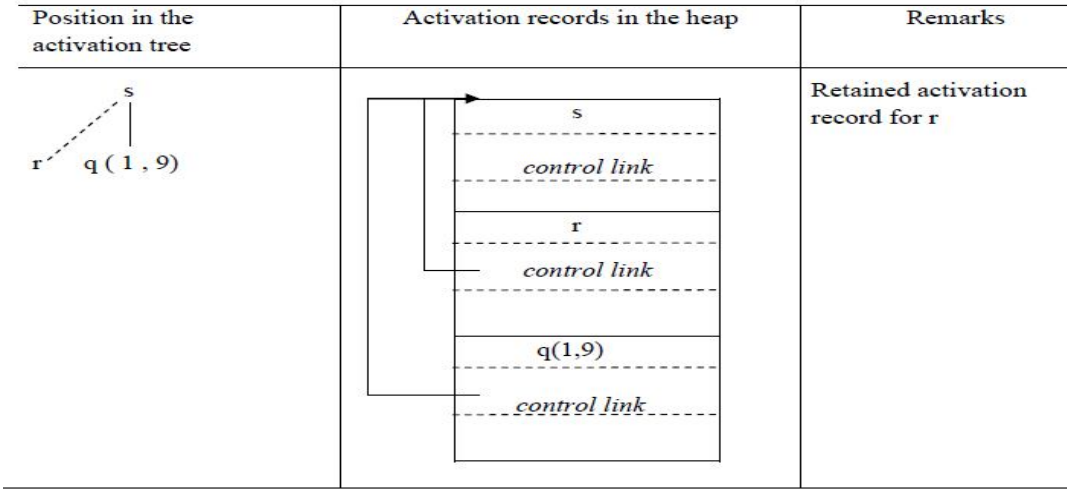
Procedure p has three local arrays, whose sizes cannot be determined at compile time.
The storage for these arrays is not part of the activation record for p.

- Access to the data is through two pointers, *top* and *top-sp*. Here the *top* marks the actualtop of stack; it points 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.
- The code to reposition *top* and *top-sp* can be generated at compile time, in terms of sizesthat will become known at run time.

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 recordsor other objects.
- Pieces may be de-allocated in any order, so over the time the heap will consist of alternateareas that are free and in use.



- 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 de-allocated, there will be free space in the heapbetween the activation records for s and q.

DANGLING REFERENCES:

Whenever storage can be de-allocated, the problem of dangling references arises.A *dangling reference* occurs when there is a reference to storage that has been de-allocated. It is alogical error to use dangling references, since the value of de-allocated storage is undefined according to the semantics of most languages. Worse, since that storage may later be allocated to another datum, mysterious bugs can appear in programs with dangling references.

5.8 ACCESS TO NON LOCAL NAMES

The storage-allocation strategies of the last section are adapted in this section to permitaccess to non-local names. Although the discussion is based on stack allocation of activationrecords, the same ideas apply to heap allocation.

The scope rules of a language determine the treatment of references to non- local names.A common rule, called the lexical or static scope rule, determines the declaration that applies to aname by examining the program text alone. Pascal, C, and Ada are among the many languagesthat use lexical scope, with an added ”most closely nested” stipulation that is discussed below.

An alternative rule, called the dynamic-scope rule determines the declaration applicableto a name a (run time, by considering the current activations. Lisp, APL, and SNOBOL are among the languages that use dynamic scope

5.8.1 BLOCKS

A block is o statement containing its own local data declarations. The concept of a blockoriginated with Algol. In C, a block has the syntax

{Declarations statements}

A characteristic of blocks is their nesting structure Delimiters mark the beginning and end of a block C uses the braces (and) as delimiters while the ALGOL tradition is to use begin and end.Delimiters ensure that one block is either independent of another or is nested inside the other that is it is not possible for two blocks B1 and B2 to overlap in such a way that first *B1* begins then *B2* but *B1* ends before B2 This nesting property is sometimes referred to as *blockstructure*

The scope of a declaration in a block-structured language is given by the most closely nested rule

1. The scope of a declaration in a block B includes *B.*
2. If a name x is not declared in a block *B,* then an occurrence of x in *B* is in the scope of a declaration of x in an enclosing block *B’* such that

i) *B’* has a declaration of x, and

ii) B’is more closely nested around *8* than any other block with a declaration of x.

By design each declaration in Fig. 7.18 is initialized the declared name to the number of the block that it appears in. The scope of the declaration of b in *B0* does not include *B1.* Becauseb is re declared in *B1.* Indicated by *B0 – B1* in the figure such a gap is called a *hole* in the scopeof the declaration.

The most closely nested scope rule is reflected in the output of the program in Fig. 7. 18 Control flows to a block from the point just before it, and flows from the block to the point just after it in the source text. The print statements are therefore executed in the order *B2 B3 B1* and *B0* the order in which control leaves the blocks The values of a and b in these blocks are:

2 1
0 3
0 1
0 0

Block structure can be implemented using stack allocation. Since the scope of a declaration doesnot extend outside the block in which it appear the space for the declared name can be allocatedwhen the block is entered and reallocated when control leaves the block This view treats a blockas a parameterized procedure, called only from the point just before the block and returning onlyto the point just after the block The non-local environment for a block can be maintained usingthe techniques for procedure later is this section Note, however, that blocks are

simpler thanprocedures because no parameters are passed and because the flow of control from and to a blockclosely follows the static program text .

```
main()
{
int a=0;
int b=0;
{
int b=1;
{
int a=2;
B2
printf("%d %d \n" ,a,b);
B0 }
B1
{
int b=3;
B3
printf("%d %d \n",a,b);
}
printf("%d %d \n",a,b);
}
printf(" %d %d \n"a,b);
}
```

Fig:7.18 Blocks in a C program.

An alternative implementation is to allocate storage for a complete procedure body at one time if there are blocks within the procedure then allowance is made for the storage needed fordeclarations within he blocks For block B, in Fig. 7. l8, we can allocate storage as in Fig. 7.19. Subscripts on locals a and b identify the blocks that the locals are declared in. Note that a2 and b3 may be assigned the same storage because they are in blocks that are not alive at the same time

In the absence of variable-length data, the maximum amount storage needed during anyexecution of a block can be determined at compile time (Variable length data can be handled using pointers as in section 7.3) by making this determination, we conservatively assume that allcontrol paths in the program can indeed be taken. That is, we assume that both the then- and elsepartsof a conditional statement can be executed, and that all statements within a while loop canbe reached.

Lexical Scope without Nested Procedures

The lexical-scope rules for C are simpler than those of Pascal discussed next, because procedure definitions cannot be nested in C. That is, a procedure definition cannot appear withinanother. As in Fig. 7.20, a C program consists of a sequence of declarations of variables andprocedures (C calls them functions).

If there is a non-local reference to a name a in somefunction, then it must be declared outside any function. The scope of a declaration outside a function consists of the function bodies that follow the declaration, with holes if the name is re-declared within a function. In Fig.

7.20, non-local occurrences of a in read array, partition, and main refer to the array declared, on line 1

```
(1) int a[11];
(2) readarray() { …..a…..}
(3) int partion(y,z) int y,z; {….a….}
(4) quicksort(m,n) int m,n; {…….}
(5) main() {….a…}
```

Fig: 7.20. C program with non-local occurrences of a.

In the absence of nested procedures, the stack-allocation strategy for local names can be used directly for a lexically scoped language like C. Storage for all names declaredoutside any procedures can he allocated statically. The position of this storage is known atcompile time, so if a name is nonloca1 in some procedure body, we simply use the staticallydetermined address. Any other name must be a local of the activation at the top of the stack,accessible through the *top* pointer. Nested procedures cause this scheme to fail because anonloca1 may then refer to data deep in the stack, as discussed below.

An important benefit of static allocation for non-locals is that declared procedures can freely hepassed as parameters and returned as result (a function is passed in C by passing a pointer to it).With lexical scope and v without nested procedures, any name non-local to one procedure is nonlocalto all procedures. Its static address can be used by all procedures, regardless of how theyare activated. Similarly, if procedures are returned as results, non-locals in the returnedprocedure refer to the storage statically allocated for them. For example, consider the Pascalprogram in Fig. 7.2l. All occurrence of name m, shown circled in Fig. 7.2l, are in the scope of thedeclaration on line 2. Since m is non local to all procedures in the program, its storage can. Beallocated statically. Whenever procedures f and g are executed, they can use

```
(1) program pass (input,output);
(2) var m :integer ;
(3) function f(n : integer ) : integer ;
(4) begin f : = m+n end {f };
(5) function g(n : integer ) : integer ;
(6) begin g := m*n end {g};
(7) procedure b (function h(n :integer ) : integer);
(8) begin write (h (2)) end {b};
(9) begin
(10) m:=0;
(11) b(f); b(g); writeln
(12) end.
```

Fig: 7.12 Pascal program with non-local occurrences of m.

The static address to access the value of m. The fact that f and g are passed as parameters onlyaffects when they are activated; it does not affect how they access the value of m.

In more detail the call b (f) on line 1l associates the function f with the formal parameter h ofthe procedure b. So when the formal h is called on line 8, in write (h (2)), the function f isactivated. The activation of f returns 2 because non-local m has value 0 and formal p has value 2Next in the execution, the call b (g) associates g with h; this time, a call of h activates g. Theoutput of the program is2 0

Lexical Scope with Nested Procedures

A non-local occurrence of a name a in a Pascal procedure is in the scope of the most closelynested declaration of a in the static program text.The nesting of procedure definitions in the Pascal program of Fig. 7.22 is indicated by thefollowing indentation:

sort
readarray
exchange
quicksort
Partition

The occurrence of a on line 15 in Fig. 7. 22 is within function partition, which is nestedwithin procedure quick sort. The most closely nested declaration of a is on line 2 in theprocedure consisting of the entire program. The most closely nested rule applies to procedure names as well.

```
(1)program sort (input , output);
(2) var a : array [0.....10] of integer;
(3) x: integer
(4) procedure readarray;
(5) var i : integer;
(6) begin ......a.......end { readarray};
(7) procedure exchange (i,j :integer);
(8)begin
(9) x: = a[i]; a[i] := a[j]; a[j] :=x
(10) end (exchange}:
(11) procedure quicksort(m,n:integer);
(12) var k,v :integer;
(13) function partion (y,z : integer ): integer;
(14) var i,j : integer;
(15) begin ...a....
(16) .....v......
(17) .....exchange(i,j);...
(18)end {partition};
(19)begin ...end {quicksort};
(20) begin .....end {sort}
```

Fig: 7.22. A Pascal with nested procedures.

The procedure exchange, called by partition on line 17, is non-local to partition.Applying the rule we first check if exchange is defined within quick sort; since it is not .We lookfor it in the main program sort.

Nesting Depth

The notion of nesting depth of a procedure is used below to implement lexical scope.Let the name of the main program be at nesting depth 1;we add 1 to the nesting depth as we go from an enclosing to an enclosed procedure. In Fig: 7.22, procedure quick sort on line 11 is at nesting depth 2, while partition on line 13 is at nesting depth 3. With each occurrence of a name,we associate the nesting depth of the procedure in which it is declared. The occurrences of a, v,and i on lines 15-17 in partition therefore have nested depths 1,2 and 3 respectively.

Access Links

A direct implementation of lexical scope for nested procedures is obtained be adding apointer called an access link to each activation record. If procedure p is nested immediatelywithin q in the source text, then the access link in an activation record for p points to the accesslink in the record for the most recent activation of q.Snapshots of run-time stack during an execution of the program in Fig: 7.22 are shown in Fig:2.23.Again, to save space in the figure, only the first letter of each procedure name is shown. Theaccess link for the activation of sort is empty, because there is no enclosing procedure.

Theaccess link for each activation of quick sort points to the record for sort. Note Fig: 7.23c that theaccess link in the activation record for partition (1,3) points to the access link in the record of themost recent activation of quick sort, namely quick sort (1,3).

Suppose procedure p at nesting depth np refers to a non-local a with nesting depth *na*<=*np*. The storage for a can be found as follows.

1. When control is in p, an activation record for p is at top of the stack. Follow *np* - *na* accesslinks from the record at the top of the stack. The value of *np* - *na* can be precomputed at compilertime. If the access link in one record point to the access link in another, then performing a singleindirection operation can follow a link.
2. After following *np* - *na* links, we reach an activation record for the procedure that a is localto. As discussed in the last section, its storage is at a fixed offset relative to a position in therecord. In particular, the offset can be relative to the access Hence, the address of non-local a in procedure p is given by the following pair computed at

compile time and stored in the symbol table:

(*np* - *na, offset* within activation record containing a)

The first component gives the number of access links to be traversed.

For example, on lines 15-16 in Fig: 7.22,the procedure partition at nesting depth 3 references non-locals a and v at nesting depths 1 and 2,respectively. The activation record containing the

storage for these non-locals are found by following 3-1=2 and 3-2=1 access links , respectively, from the record for partition.

The code to setup access links is part of the calling sequence. Suppose procedure p at nestingdepth *np* calls procedure x at nesting depth *nx*. The code for setting up the access link in thecalled procedure depends on whether or not the called procedure is nested within the caller.

1. Case *np* < *nx. Since* the called procedure x is nested more deeply than p it must be declare within p, or it would not be accessible to p. This case occurs when sort calls quick sort in Fig:7.23 (a) and when quick sort calls partition in Fig: 7.23(c) .In this case, the access link in thecalled procedure must point to the access link in the activation record of the caller just below in the stack.

2. Case *np* >= *nx*. From the scope rules, the enclosing procedures at nesting depths 1,2,3…. *nx*- 1 of the called and calling procedures must be the same, as when quick sort calls itself in Fig: 7.23(b) and when partition calls exchange in Fig: 7.23 (d). Following *np*-*nx*+1 access links fromthe caller we reach the most recent activation record of procedure that statically encloses both thecalled and calling procedures most closely. The access link reached is the one to which theaccess link in the called procedure must point. Again *np*-*nx* +1 can be computed at compile time

Procedure Parameters

Lexical scope rules apply even when nested procedure is passed as parameter. The function f onlines 6-7 of the Pascal program in Fig: 7.24.has a non-local m; all occurrences of m are showncircled. On line 8, procedure c assigns 0 to m and then passes f as a parameter to b. Note that thescope of the declaration of m on line 5 does not include the body of b on lines 2-3.

```
(1) program param(input,output);
(2)procedure b(function h(n:integer):integer);
(3)begin writeln(h(2) end {b};
(4)procedure c;
(5)var (m):integer;
(6)function f(n:integer ): integer;
(7) begin f:=(m)+n end {f};
(8)begin (m):=0; b(f) end {c};
(9)begin
(10)c
(11)end
```

Fig: 7.24.An access link must be passed with actual parameter f.

Within the body of b, the statement writeln (h (2)) activates f because the formal h refers to f.That is writeln prints the result of the call f (2).

How do we setup the access link for the activation of f. The answer is that a nested procedure that is passed as a parameter must take its access link along with it, as shown in Fig: 7.25.When aprocedure c passes f, it determines an access link for f, just as it would if it were calling f. Thatlink is passed along with f to b. Subsequently, when f is activated from within b; the link is usedto setup the access link in the activation record of f.

Displays

Faster access to non-locals than with access links can be obtained using an array *d* ofpointers to activation records, called a *display*. We maintain the display so that storage for a nonloca**l a** at nesting depth **i** is in the activation record pointed to by display element d [i].

Suppose control is in an activation of a procedure p at nesting depth j. Then, the first j-1 elements of the display point to the most recent activations of the procedures that lexicallyenclose procedure p, and d [j] points to the activation of p. Using a display is generally faster than following access link because the activation record holding a non-local is found by accessing an element of d and then following just one pointer.

A simple arrangement for maintaining the display uses access links in addition to the display. As part of the call and return sequences, the display is updated by following the chain ofaccess links. When the link to an activation record at nesting depth *n* is followed, display elementd[n] is set to point to that activation record. In effect, the display duplicates the information inthe chain of access links.

The above simple arrangement can be improved upon. The method illustrated in fig7.26 requires less work at procedure entry and exit in the usual case when procedures are notpassed as parameters. In Fig. 7.22. Again, only the first letter of each procedure name is shown.Fig. 7.26(a) shows the situation just before activation q (1,3) begins. Since *quick sort* isat nesting depth 2, display element d [2] is affected when a new activation of quick sort begins.

The effect of the activation q (1,3) on d [2] is shown in Fig 7.26(b), where d [2] now points to thenew activation record; the old value of d [2] is saved in new activation record. The

saved valuewill be needed later to restore the display to its states in Fig 7.26(a) when controls return toactivation of q (1,9).

The display changes when a new activation occurs, and it must be reset when controlreturns from new activation. The scope rules of Pascal and other lexically scoped languagesallow the display to be maintained by following steps. We discuss the only easier case in whichprocedures are not passed as parameters. When a new activation record for a procedure at nesteddepth i is set up, we

1. Save the value of d[i] in the activation record and
2. Set d [i] to the new activation record.

Just before activation ends, d [i] is reset to the saved value.

These steps are justified as follows. Suppose a procedure at nesting depth j calls aprocedure at depth i. There are two cases, depending on whether or not the called procedure isnested within the caller in the source text, as in the discussion of access links

1. Case j<i. Then i=j+1 and the called procedure is nested within the caller. The first j elementsof the display therefore do not need to be changed, and we set d [i] to the new activation record.This case is illustrated in Fig.7.26(c).

2. Case j<=i. Again, the enclosing procedures at nesting depths 1,2…i-1 of the called and callingprocedure must be the same. Here, we save the old value of d [i] in the new activation record,and make d [i] point to the new activation record. The display is maintained correctly becausethe first i-1 elements are left as is.

An example of Case 2, with i=j=2,occurs when *quick sort* is called recursively in Fig.7.26(b). A more interesting example occurs when activation p (1,3) at nesting depth 3 calls e(1,3) at depth 2, and their enclosing procedure is s at depth 1, as in Fig. 7.26(d). Note that when (1,3) is called, the value of d [3] belonging to p (1,3) is still in the display, although it cannot beaccessed while control is in e.

Should e call another procedure at depth 3, that procedure willstore d [3] and restore it on returning to e. We can thus show that each procedure sees the correctdisplay for all depths up to its own depth.

There are several places where a display can be maintained. If there are enoughregisters, then the display, pictured as an array, can be a collection of registers. Note that thecompiler can determine the maximum nesting depth of procedures in the program. Otherwise thedisplay can be kept statically allocated memory and all references to activation records begin byusing indirect addressing through the appropriate display pointer. This approach is reasonable ona machine with indirect addressing, although each indirection costs a memory cycle. Anotherpossibility is to store the display on the run time stack itself, and to create a new copy at each procedure entry.

5.8.2 DYNAMIC SCOPE

Under dynamic scope, a new activation inherits the existing bindings of non-localnames to storage. A non-local name a in the called activation refers to the same storage that it didin the calling activation. New bindings are set up for local names of the called procedure; thenames refer to storage in the new activation record.

The program in fig7.27 illustrates dynamic scope. Procedure show on lines 3-4 writesthe value of non-local r. under lexical scope in Pascal, the non-local r is in the scope of thedeclaration on line 2, so the output of the program is

0.250 0.250

0.250 0.250
However, under dynamic scope, the output is
0.250 0.125
0.250 0.125

When show is called on line 10-11 in the main program, 0.250 is written because thevariable r local to the main program is used. However, when show is called on line 7 from withinsmall, 0.125 is written because the variable r local to small is used.

```
(1) program dynamic (input, output);
(2) var r:real;
(3) procedure show;
(4) begin write(r:5:3) end;
(5) procedure small;
(6) var r : real;
(7) begin r := 0.125;show end;
(8) begin
(9) r := 0.25;
(10) show;small;writeIn;
(11) show; small; writeIn
(12) end.
```

Fig 7.27

(The output depends on whether lexical or dynamic scope is used)

The following 2 approaches to implementing dynamic scope bear some resemblance to the use ofaccess links and displays, respectively, in the implementation of lexical scope.

1. **Deep Access**. Conceptually, dynamic scope results if access links point to the same activationrecords that control links do. A simple implementation is to dispense with access links and usethe control link to search into the stack, looking for the first activation record containing storagefor the non-local name. The term deep access comes from the fact that the search may go "deep"into the stack. The depth to which the search may go depends on the input to the program andcannot be determined at compile time.

2. **Shallow access**. Here the idea is to hold the current value of each name in statically allocatedstorage. When a new activation of a procedure p occurs, a local name n in p takes over thestorage statically allocated for n. The previous value of n can be saved in the activation recordfor p and must be restored when the activation of p ends.

The trade off between the 2 approaches is that deep access takes longer to access a nonlocal but there is no overhead associated with beginning and ending activation. Shallow accesson the other hand allows non-locals to be looked up directly, but time is taken to maintain thesevalues when activation begin and end. When functions are passed as parameters and returned asresults, a more straightforward implementation is obtained with deep access.

5.9 PARAMETER PASSING

- Parameter passing
- Terminology:

• **procedure declaration:**
. parameters, formal parameters, or formals.
• **procedure call:**
. arguments, actual parameters, or actuals.

The value of a variable:
• r-value: the current value of the variable.. Right valueon the right side of assignment• l-value: the location/address of the variable.
. left value - . on the left side of assignment• Example: x := y

Four different modes for parameter passing

- call-by-value
- call-by-reference
- call-by-value-result(copy-restore)
- call-by-name

Call-by-value

Usage:
• Used by PASCAL if you use non-var parameters.
• Used by C++ if you use non-& parameters.
• The only thing used in C.
Idea:
• calling procedure copies the r-values of the arguments into the called procedure's A.R.
Effect:
• Changing a formal parameter (in the called procedure) has no effect on the corresponding actual. However, if the formal is a pointer, then changing the thing pointed to does have an effectthat can be seen in the calling procedure.

Example:

```
void f(int *p)
{ *p = 5;
p = NULL;
}
main()
{int *q = malloc(sizeof(int));
*q=0;
f(q);
}
```

• In main, q will not be affected by the call of f.
• That is, it will not be NULL after the call.
• However, the value pointed to by q will be changed from 0 to 5.

Call-by-reference

Usage:
• Used by PASCAL for var parameters.
• Used by C++ if you use & parameters.
• FORTRAN.
Idea:
• Calling procedure copies the l-values of the arguments into the called procedure's A.R. as follows:
. If an argument has an address then that is what is passed.
. If an argument is an expression that does not have an l-value (e.g.,a + 6), then evaluate the argument and store the value in a temporary address and pass that address.Effect:

• Changing a formal parameter (in the called procedure) does affect the corresponding actual.
• Side effects.

Call-by-reference

Example:

```
FORTAN quirk /* using C++ syntax */
void mistake(int & x)
{x = x+1;}
main()
{mistake(1);
cout<<1;
}
```

Call-by-value-result

Usage: FORTRAN IV and ADA.

Idea:

• Value, not address, is passed into called procedure's A.R.
• When called procedure ends, the final value is copied back into theargument's address.

Equivalent to call-by-reference except when there is aliasing.

• "Equivalent" in the sense the program produces the same results, NOTthe same code will be generated.
• Aliasing : two expressions that have the same l-value are calledaliases. That is, they access the same location from different places.
• Aliasing happens through pointer manipulation.
. call-by-reference with an argument that can also be accessed by thecalled procedure directly, e.g., global variables.
. call-by-reference with the same expression as an argument twice; e.g. test(x, y, x).

Call-by-name

Usage: Algol.

Idea: (not the way it is actually implemented.)

• Procedure body is substituted for the call in the calling procedure.
• Each occurrence of a parameter in the called procedure is replacedwith the corresponding argument, i.e., the TEXT of the parameter,not its value.
• Similar to macro substitution.
• Idea: a parameter is not evaluated unless its value is needed duringthe computation.

Call-by-name

Example:

```
void init(int x, int y)
{ for(int k = 0; k <10; k++)
{ x++; y = 0;}
}
main()
{ int j;
int A[10];
j = -1;
init(j,A[j]);
}
```

Conceptual result of substitution:

```
main()
{ int j;
int A[10];
j = -1;
for(int k = 0; k<10; k++)
{ j++; /* actual j for formal x */
A[j] = 0; /* actual A[j] for formal y */
}}
```

Call-by-name is not really implemented like macro expansion.
Recursion would be impossible, for example, using this approach.
How to implement call-by-name?
Instead of passing values or addresses as arguments, a function(or the address of a function) is passed for each argument.These functions are called thunks. , i.e., a small piece of code.
Each thunk knows how to determine the address of thecorresponding argument.
• Thunk for j: find address of j.
• Thunk for A[j]: evaluate j and index into the array A; find the address of the appropriate cell.
Each time a parameter is used, the thunk is called, then the address returned by the thunk is used.
• y = 0: use return value of thunk for y as the l-value.
• x = x + 1: use return value of thunk for x both as l-value and to getr-value.
• For the example above, call-by-reference executes A[1] = 0 ten times,while call-by-name initializes the whole array.
Note: call-by-name is generally considered a bad idea, because It is hard to know what a function is doing – it may requirelooking at all calls to figure this out.

Advantages of call-by-value

- Consider not passing pointers.
- No aliasing.
- Arguments are not changed by procedure call.
- Easier for static optimization analysis for both programmers andthe complier.
- Example:
- x = 0;
- Y(x); /* call-by-value */
- z = x+1; /* can be replaced by z=1 for optimization */
- Compared with call-by-reference, code in the called function isfaster because of no need for redirecting pointers.

Advantages of call-by-reference

- Efficiency in passing large objects.
- Only need to copy addresses.

Advantages of call-by-value-result

- If there is no aliasing, we can implement call-by-value-resultusing call-by-reference for large objects.
- No implicit side effects if pointers are not passed.

Advantages of call-by-name

- More efficient when passing parameters that are never used.

Example:

```
P(Ackerman(5),0,3)
/* Ackerman's function takes enormous time to compute */
function P(int a, int b, int c)
{ if(odd(c)){
return(a)
}else{ return(b) }
}
```