**UNIT-4**

**RUNTIME ENVIORNMENT**

**SOURCE LANGUAGE ISSUES**

**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.

**Activation trees:**

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

activation tree,

**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 an

integer.

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

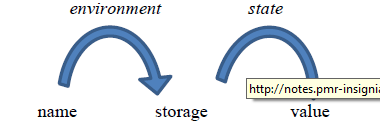
**Binding of names:**

Even if each name is declared once in a program, the same name may denote different

data 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*.

**STORAGE ORGANISATION**

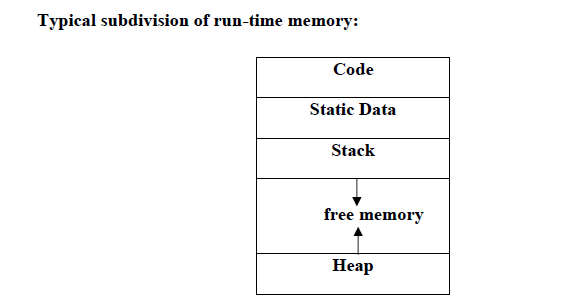
\* The executing target program runs in its own logical address space in which each

program value has a location.

\* The management and organization of this logical address space is shared between the

complier, operating system and target machine. The operating system maps the logical

address into physical addresses, which are usually spread throughout 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 compiler

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 *control*

*stack.*

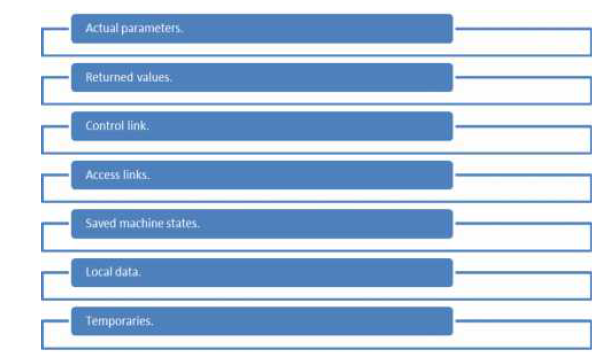
\* Each live activation has an activation record on the control stack, with the root of the

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. The

diagram 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 called procedures

return a value, and if one does, we may prefer to place that value in a register for

efficiency.

\* The actual parameters used by the calling procedure. These are not placed in activation

record but rather in registers, when possible, for greater efficiency.

**STORAGE 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 area

known as heap.

**STATIC ALLOCATION**

\* In static allocation, names are bound to storage as the program is compiled, so there is no

need for a run-time support package.

\* Since the bindings do not change at 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 were

when 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.

**STACK ALLOCATION OF SPACE**

\* All compilers for languages that use procedures, functions or methods as units of userdefined

actions 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, and

when the procedure terminates, that space is popped off the stack.

**Calling sequences:**

\* Procedures called are implemented in what is called as calling sequence, which consists

of code that allocates an activation record on the stack and enters information into its

fields.

\* A return sequence is similar to code to restore the state of 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) and

the 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 the

caller’s activation record.

\* Fixed length items are generally placed in the middle. Such items typically include

the 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 the

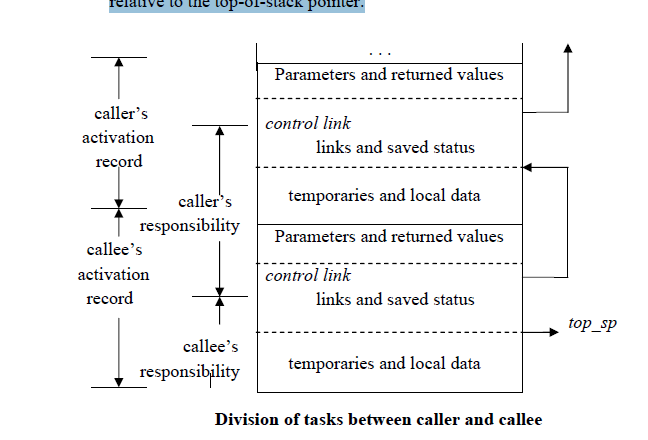
activation record. The most common example is dynamically sized array, where the

value 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 have

it point to the end of fixed-length fields in the activation record. Fixed-length data

can then be accessed by fixed offsets, known to the intermediate-code generator,

relative to the top-of-stack pointer.

\* 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.

**Variable length data on stack:**

\* The run-time memory management system must deal frequently with the allocation of

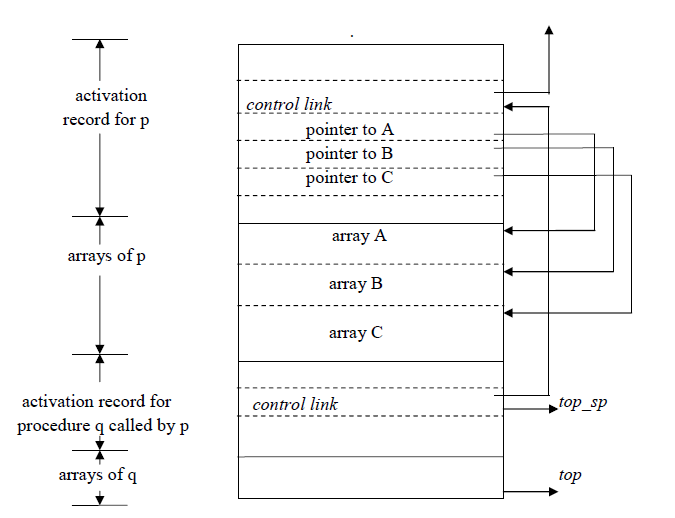
space for objects, the sizes of which are not known at the compile time, but which are

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.

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 actual

top 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 sizes

that 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.

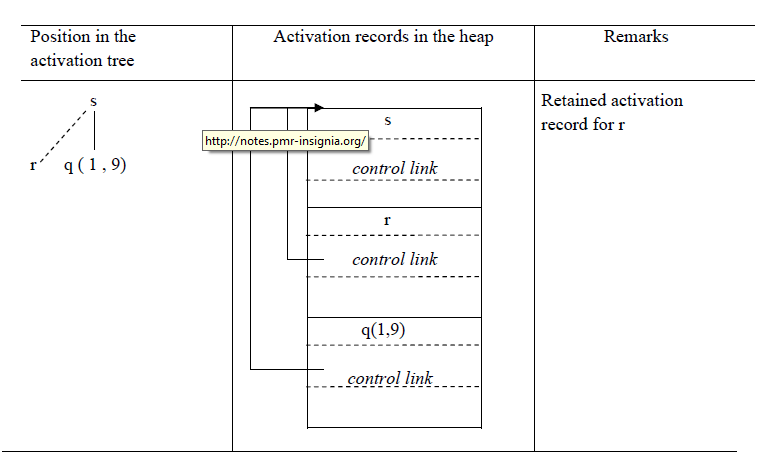
2. A called activation outlives the caller.

\* Heap allocation parcels out pieces of contiguous storage, as needed for activation records

or other objects.

\* Pieces may be deallocated in any order, so over the time the heap will consist of alternate

areas that are free and in use.



The record for an activation of procedure r is retained when the activation ends.

\* Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.

\* If the retained activation record for r is deallocated, there will be free space in the heap

between the activation records for s and q.

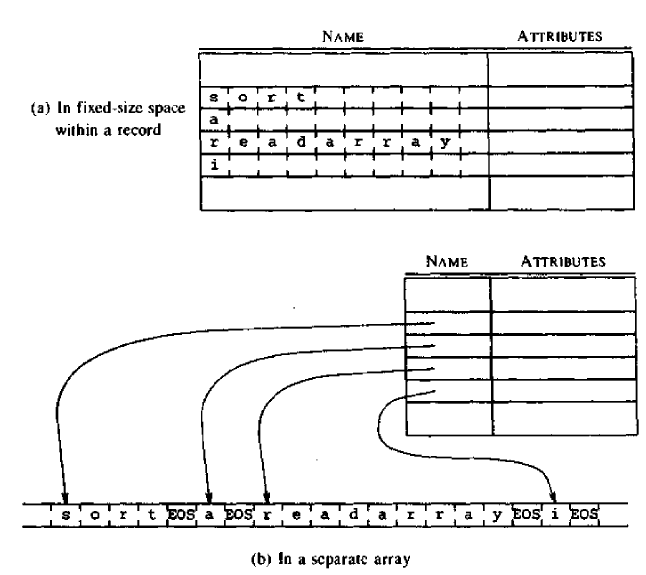
A compiler uses a symbol table to keep track of scope and binding information about names The symbol table is searched every time a name is encountered in the source text.

Changes to the table occur if a new name or Information about an existing name i s discovered. A compiler uses a symbol table to keep track of scope and binding information about names The symbol table is searched every time a name is encountered in the source text. Changes to the table occur if a new name or Information about an existing name i s discovered. The two symbol-table mechanims presented in this section are linear lists and hash tables.A linear list is the simplest to implement. but its performance is poor. Hashing schemes provide better performance for somewhat greater programming effort and space overhead.

Symbol-Table Entries

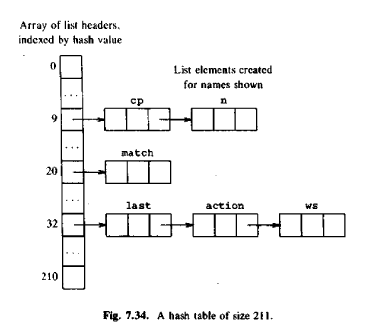
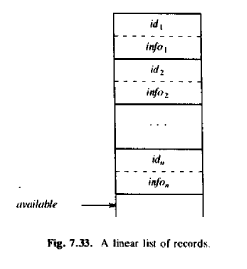
Each entry in the symbol table is for the declaration of a name A common representat ion of a name is a pointer to a symbol-table entry for it.If there is a modest upper bound on the length of a name, then the characters in the name can be stored in the symbol-table entry, as in Fig. 7.32(a). If there is no limit On the length of a name, or if the limit is rarely reached, the indirect scheme of Fig. 7.32(b) can be used

Rather than allocating in each symbol-table entry the maximum possible amount of space to hold a lexeme,we can utilize space more efficiently if there is only space for a pointer in a symbol-table entry



**Storage Allocation Information**

The List Data Structure for Symbol Tables

The simplest and easiest to implement data structure for a symbol table is a linear list of records We use a single array, or equivalently several arrays. to store names and their associated information 

The position of the end of the array is marked by the pointer available, pointing to where the next symbol-table entry will go If the symbol table contains *n names. the work necessary to insert a new* name is constant if we do the insertion without checking to see if the name is already in the table. if multiple entries for names are not allowed.

Hash Tables

Here we consider a rather simple variant knownas *open hashing, where “open" refers to the property that there need be no* limit on the number of entries that can be made in the table.,

There are two parts of data structure:

data structure: A *hashing e consisting of a fixed array of m pointers to table entries.*

Table entries organized into *m separate linked lists, called buckers {some*

buckets may be empty). Each record in the symbol table appears on exactly one of these lists. Storage for the records may be drawn from an5

array of records, as discussed in the next section. Alternatively, the dynamic storage allocation facilities of the implementation language can be used to obtain space for the records, often at some loss of efficiency.

Representing Scope Information

The entries in the symbol table are for declarations of names. When an occurrence of a name in the source text is looked up in the symbol table, the entry for the appropriate declaration of that name must be returned the symbol table for a procedure or scope *is the compile time* equivalent of an activation record. Information for the non locals of a procedure is found by scanning the symbol tables for the enclosing procedures

following the scope rules of the language We keep track of the local names of a procedure by giving each procedure a unique number. Blocks must alsobe numbered if the language is block-structured.Most closely nested scope rules can be implemented in terms of the

following operations on a name:

*Look up : fin d the most recently created entry*

insert : make a new entry

delete: remove the most recently created entry

We keep track of the local names of a procedure by giving each procedure a unique number. Blocks must also be numbered if the language is block-structured.

Most closely nested scope rules can be implemented in terms of the following operations on a name:

*Look up : fin d the most recently created entry*

insert : make a new entry

delete: remove the most recently created entry

We keep track of the local names of a procedure by giving each procedure a unique number. Blocks must also be numbered if the language is block-structured.

**LANGUAGE FACILITCES FOR DYNAMIC STORAGE ALLOCATION**

facilities provided by some languages forthe dynamic allocation of storage for data, under program control. Storage for such data is usually taken from a heap. Allocated data is often retained until it is explicit1y deallocated.The allocation itself is implicit or explicit.

Garbage Dynamically allocated storage can become unreachable. Storage that a programallocates but cannot refer to is called garbage

Dangling References

An additional complication can arise with explicit deallocation dangling references

can occur.

**DYNAMIC STORAGE ALLOCATION TECHNIQUES**

The techniques needed to implement dynamic storage allocation depend on how storage is de allocated. If de allocation is implicit, then the run-time support package is responsible for determining when a storage block is no longer needed. There *is less a compiler has lo do if de allocation is done explicitly by* the programmer.

Explicit Allocation of FixedSized Biocks

The simplest form of dynamic allocation involves blocks OF a fixed size. Bylinking the blocks in a list, as in Fig, 7.41, allocation and de allocation can be done quickly with little or no storage overhead. pointer *available points to the first block Allocation consists* of taking a block off the list and deallocation consisting of putting the block back on the list.

EXPLICIT Allocation of Variable sized Blocks

When blocks are allocated and de allocated, storage can become fragmented;that is, the heap may consist of alternate blocks that are free and in use block larger than any one of the free blocks, even though the space is available in principle.

One method for allocating variable-sized blocks is called the *first-fit method*

When a block of size s is allocated, *we search for the first free block that is of* size for s. This block is then subdivided into a used block of size s,

When a block is de allocated, we check to see if it is next to a free block. If possible, the de allocated block is combined with a free block next to it to create a larger free block

Implicit Deallocation

Implicit de allocation requires cooperation between the user program and the run-time package, because the latter needs to know when a storage block is no longer in use.

The first problem is that of recognizing block boundaries. If the size of blocks is fixed, then position information can be used,

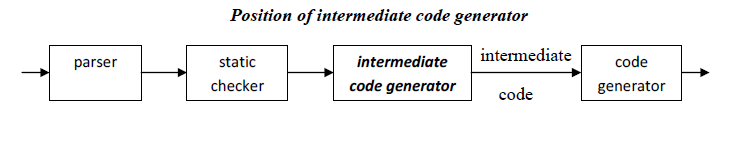
Two approaches can be used for implicit de allocation

Reference counts: We keep track of the number of blocks that point directty to the present block This approach require all the pointers into the heap to be known With variable-size blocks, we have the additional possibility of moving used storage blocks from their current positions This process Called compaction moves all used blocks to one end of the heap. So that all the free storage can be collected into one large free block.

**PART-2**

**INTERMEDIATE CODE GENERATION**

The front end translates a source program into an intermediate representation from which the back end generates target code Benefits of using a machine-independent intermediate form are: Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.



A machine-independent code optimizer can be applied to the intermediate representation

INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

· Syntax tree

· Postfix notation

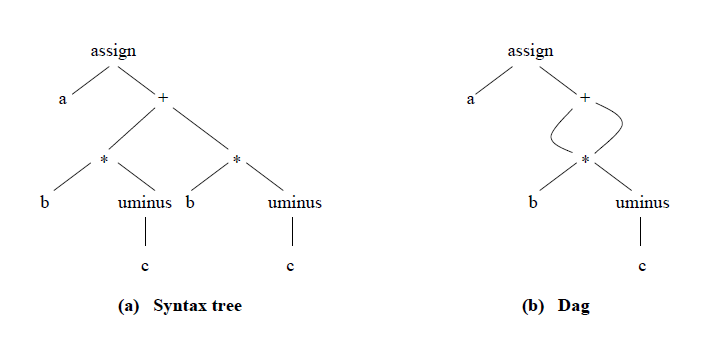
· Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

Graphical Representations

A syntax tree depicts the natural hierarchical structure of a source program.

A dag (Directed Acyclic Graph) gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statement a : =b \* - c + b \* - c are as follows:



Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of

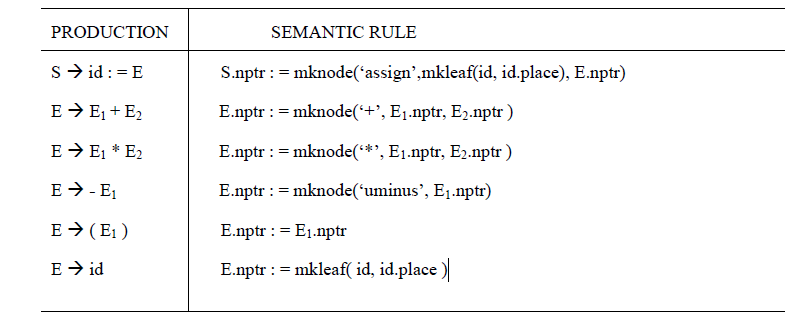
the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

a b c uminus \* b c uminus \* + assign

Syntax-directed definition:

Syntax trees for assignment statements are produced by the syntax-directed definition.

Non-terminal S generates an assignment statement.

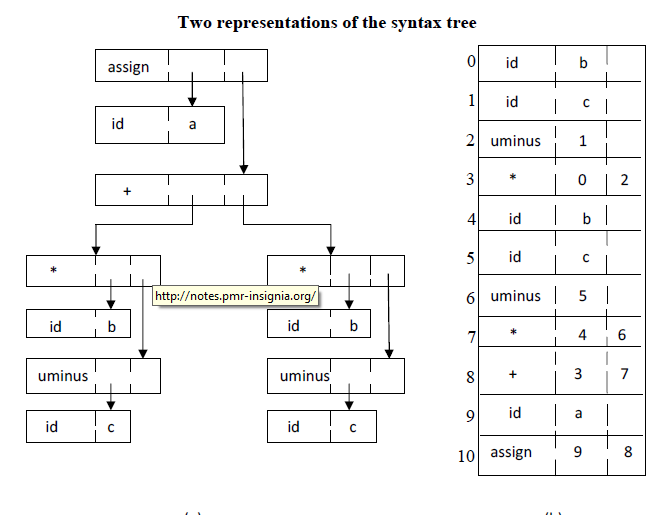


The token id has an attribute place that points to the symbol-table entry for the identifier.

A symbol-table entry can be found from an attribute id.name, representing the lexeme associated with that occurrence of id.

Two representations of the syntax tree are as follows. In (a) each node is represented as a

record with a field for its operator and additional fields for pointers to its children. In (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node



Three-Address Code:

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

x : = y *op z*

where x, y and z are names, constants, or compiler-generated temporaries; *op stands for any* operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on booleanvalued data.

Thus a source language expression like x+ y\*z might be translated into a sequence

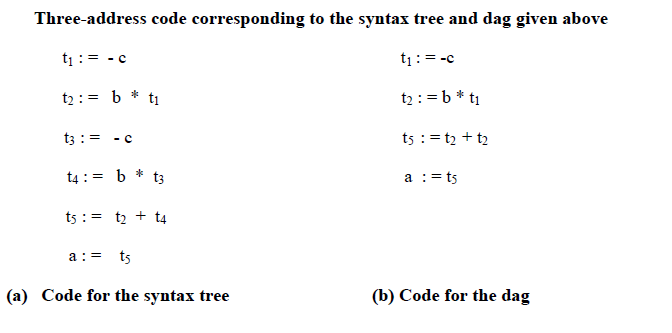
t1 : = y \* z

t2 : = x + t1

where t1 and t2 are compiler-generated temporary names.

Advantages of three-address code:

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



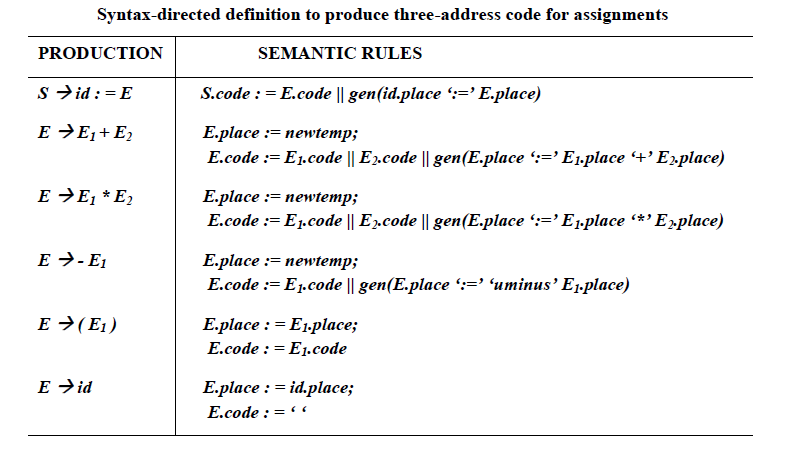
Syntax-Directed Translation into Three-Address Code:

When three-address code is generated, temporary names are made up for the interior

nodes of a syntax tree. For example, id : = *E consists of code to evaluate E into some temporary* t, followed by the assignment id.*place : = t.*

1. *E.place, the name that will hold the value of E , and*

2. *E.code, the sequence of three-address statements evaluating E.*



Implementation of Three-Address Statements:

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

Three such representations are:

Quadruples

Ø Triples

Ø Indirect triples

*Quadruples:*

Ø A quadruple is a record structure with four fields, which are, *op, arg1, arg2 and result.*

Ø The *op field contains an internal code for the operator. The three-address statement x : =*y op z is represented by lacing *y in arg1, z in arg2 and x in result.*

Ø The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

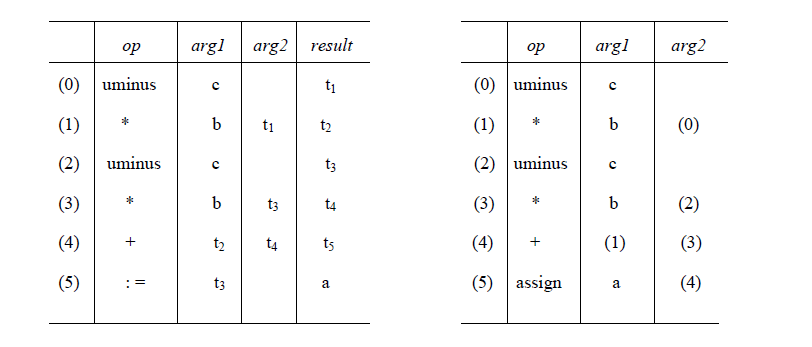
*Triples:*

Ø To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.

Ø If we do so, three-address statements can be represented by records with only three fields: *op, arg1 and arg2.*

Ø The fields *arg1 and arg2, for the arguments of op, are either pointers to the symbol table* or pointers into the triple structure ( for temporary values ).

Ø Since three fields are used, this intermediate code format is known as *triples.*



As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name

we create a symbol-table entry with information like the type and the relative address of the storage for the name

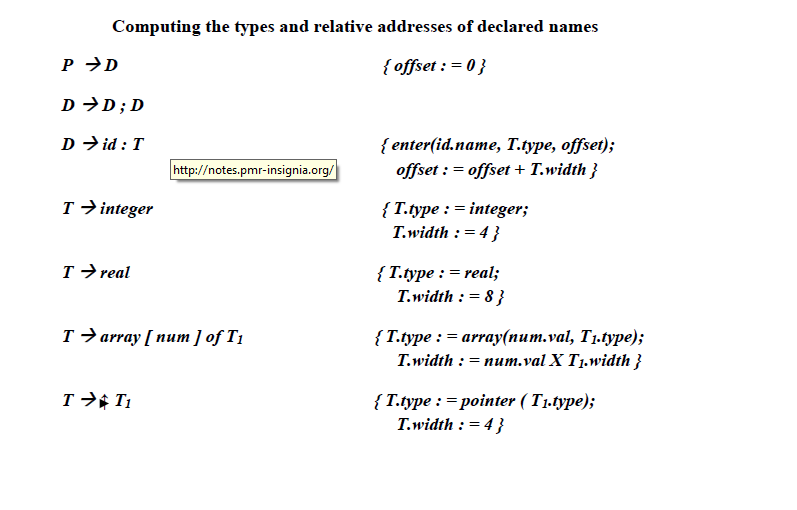
**Declarations in a Procedure:**

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a

single procedure to be processed as a group. In this case, a global variable, say *offset, can keep* track of the next available relative address.Before the first declaration is considered, *offset is set to 0. As each new name is seen ,*that name is entered in the symbol table with offset equal to the current value of *offset,*and *offset is incremented by the width of the data object denoted by that name* The procedure *enter( name, type, offset ) creates a symbol-table entry for name, gives its* type *type and relative address offset in its data area.*

The width of an array is obtained by multiplying the width of each element by the

number of elements in the array. The width of each pointer is assumed to be 4.

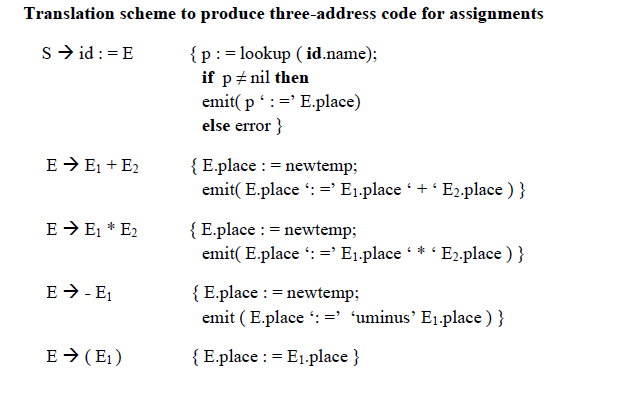
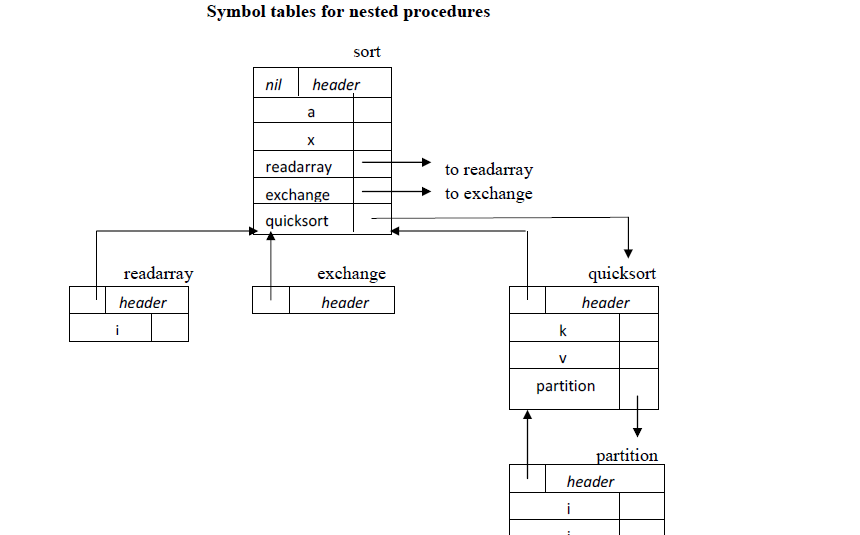


When a nested procedure is seen, processing of declarations in the enclosing procedure is

temporarily suspended.

One possible implementation of a symbol table is a linked list of entries for names

A new symbol table is created when a procedure declaration *D* and entries for the declarations in D1 are created in the new table. The new table points back to the symbol table of the enclosing procedure;

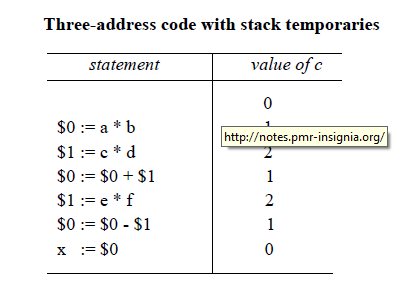
The temporaries used to hold intermediate values in expression calculations tend to clutter up the symbol table, and space has to be allocated to hold their valuesTemporaries can be reused by changing *newtemp. The code generated by the rules for E*

à E1 + E2 has the general form:

evaluate E1 into t1

evaluate E2 into t2

t : = t1 + t2



Addressing Array Elements:

Elements of an array can be accessed quickly if the elements are stored in a block ofconsecutive locations. If the width of each array element is *w, then the ith element of array A*

begins in location

*base + ( i – low ) x w*

where low is the lower bound on the subscript and base is the relative address of the storage allocated for the array The expression can be partially evaluated at compile time if

it is rewritten

*i x w + ( base – low x w)*