UNIT-IV

RUNTIME ENVIRONMENT & CODE GENERATOR

This section examines the relationship between names and data objects.

The allocation and de-allocation of data objects is managed by the runtime support package which has routines loaded with generated target code. Each execution of procedure is called an aeration of the procedure.

SOURCE LANGUAGE ISSUES

(1) This section differentiates between source text of a procedure and its activation at runtime.

Procedures:

A procedure definition is a declaration that in its simplest form of association an identifier with a statement.

(i.e.) Identifier is Procedure Name.

Statement is procedure body.

Procedures that return values are called functions in many languages. A complete program can also be treated as a procedure.

When the procedure name appears with in an executable statement, we can say that the procedure is called at that point. The procedure calls executes the procedure body. Identifiers appearing in the procedures are called as formal parameters and a identifiers that are passed to the procedure are called as actual parameters. They are substituted for the formals in the body.

Activation Trees:

Assumptions about the flow of control among procedures during the execution are,

1) Control flow is sequential.
2) Each execution of a procedure starts at the beginning of the procedure body and eventually return control to the point immediately following statement where procedure was called.

This can be represented using Trees.

Each execution of the procedure body is called as an activation of the procedure.

Life time of the activation is the steps between the first and last steps in the procedure body.

A procedure is recursive if a new activation can begin before an earlier activation of the same procedure has ended.

A recursive procedure need not call itself directly. Procedure P calls procedure q, q in turn calls P.

This can be represented using a tree called activation tree to represent the way the control enters and leaves the activation.

Control Stack

The flow of control in a program corresponds to a Depth First Traversal of the activation tree.

We can use a stack called control stack to keep track of live procedure activations. Push the node for an activation on to the control stack, as the activation begins and to pop the node when the activation ends. The contents of the control stack are the paths to the root of the activation tree.

4.1 STORAGE ORGANIZATION

The organization of run time storage is discussed in this section.

The runtime storage can be subdivided to hold

(1) The generated target code.

(2) Data objects.

(3) Control stack to keep track of procedure activations.

The size of the generated target code is fixed at compile time, hence the compiler can place t, in a statically determined area, in the low-end of the memory.

When the size of the data objects (i.e) array may be known at compile time, can be placed in the statically determined area.

Static allocation is preferable because the addresses of the objects can be placed in the target code.

A separate area of realtime memory called heap, holds all the information. Pascal allows data to be allocated under program control, for which the storage is taken from heap. In the languages where activation cannot be represented by an activation record can be represented by the heap. The data allocation and de-allocation is cheap in a stack when compared to heap.

The sizes of the stack and the heap can change as the program executes. Hence it is represented at the opposite end of memory. Pascal and C need both runtime stack and heap.

By convention stack grows down. Top of the stack is towards the bottom of the page. Memory addresses also increases as one move down the page. It is called as down ward growing.

Stack addresses can be represented as an offset from the top.

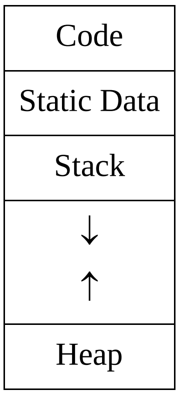


Figure 4.1 Typical subdivision of run-time memory into code and data area

Activation Records:

Information needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record or frame. The fields are shown in the Fig. 4.3. All the fields are not used at all times. In languages like Pascal and C, it is customary to push the activation record onto the stack, when the procedure is called and pop out when the procedure is over.

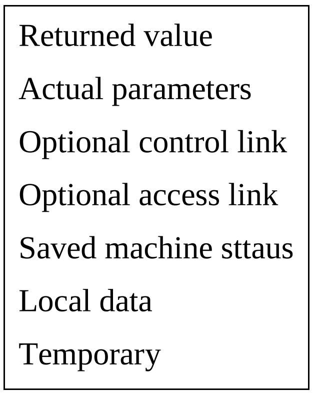


Figure 4.2 Activation Record

The purpose of the fields are:

Temporaries:

Here the temporary intermediate values arising out of an expression evaluation are stored.

Local Data:

This field is used to store the data that is local to the execution of the procedure.

Saved Machine Status:

This field holds the information about the state of the machine just before the procedure is called. The information are program counter value, Machine Registers that needs to be restored are saved in this field.

Optional Access Link:

This is used to refer to non-local data held in other activation records. (eg.) In Fortran Access Links are not-needed because non-local data are kept in a fixed place. In Pascal access links are needed.

Optimal control ink points to the activation record of the caller.

Actual Parameters:

Here the actual parameter values are stored. In reality the parameters are passed in machine register for greater efficiency.

Returned Value:

This field is used to store the return value which is used by the called procedure to return a value to the calling procedure.

Compile-Time Layout of Local Data:

If Run time storage comes in blocks of contiguous bytes where byte is the smallest unit of addressable memory. Multi-byte objects are stored in consecutive byte and given the address of the first byte.

The amount of storage needed for a name is determined by its type. Storage for an aggregate must be large enough to hold all its components, storage for aggregate is allocated in contiguous block.

Local data declaration is examined at compile time. Variable length data is kept out side. Relative addressing mechanism is used to calculate the address.

The storage layout for data object is strongly influenced by the addressing constraints of the target machine.

Example: Add two integers.

The integer is expected to be aligned (i.e.,) placed in a certain position (i.e.,) which is divisibly by 4.

Space left unused due to alignment are called as padding. When there is demand for the space to the padding in left.

4.2 STORAGE ALLOCATION STRATEGIES

Different allocation strategy is used in three areas of organization.

(i)	**Static allocation**	˜	Storage all data object at compile time.
(ii)	**Stack allocation**	˜	Run time storage as a stack.
(iii)	**Heap allocation**	˜	Run time storage as a heap.

Static Allocation:

Names are bound to storage during compilation time. Hence there is no need for runtime support. Bindings do not change during runtime. When a procedure is activated, its names are bound to the same storage location. When the control returns to a procedure the values of the local are the same as when the control left last time.

The compiler must decide where the activation records go, relative to the target code. After this decision is made, the position of each activation record, storage for each name in the record is fixed.

During compilation the address of the data on which the target code operates can be fixed.

Limitation:

1) Size of the data object should be known at compile time.
2) Restriction of Recursive procedure because all activations use the same bindings for local names.
3) Data structure cannot be created dynamically.

   Fortran permit stack storage allocation.

STACK ALLOCATION IN FORTRAN

Stack allocation is based on the idea of control stack. Storage is organized as stack. Activation records are pushed and popped as activations begin and end respectively. Storage for the local in each call of the procedure is contained in the activation record for that call.

$\therefore$ Local data objects are bound to get fresh stack storage in each activation because new activation record is pushed on to the stack when the call is made.

The values of the local are deleted when the activation ends. The stack allocation can be discussed under two different environment.

Environment 1: Size of all activation records are known at compile time.

Register top marks the top of the stack. At realtime an activation record can be allocated and de-allocated by incrementing or decrementing top by the size of the record.

$$\text{Procedure} \rightarrow Q$$

Activation Record Size *a*.

Top is incremented by '*a*' before target code of Q is executed. When control return from Q top is decremented by '*a*'.

Calling Sequences:

A call sequence allocates an activation record and enters information into its fields. A return sequence re-stores the state of the machine so that the calling procedure can continue its execution.

Calling sequences and activation records differ for implementation of the same language. Caller and Callee terminologies are used.

Caller: The calling procedure.

Callee: The procedure is called.

But there is no exact division of runtime tasks between the caller and callee.

Each call has its own parameters, the caller evaluates the actual parameter and communicates to the activation record of the callee. In the run time stack the activation record of the caller is below to that of callee.

This method is advantageous because the caller can access these fields using offsets from its end of activation record without knowing the callee. The caller need not know the local data or temporary data of the callee.

This is advantageous in case of variable number of arguments to a procedure length.

In case of Pascal, locals to a procedure to be determined at compile time. The size depends upon the value of the parameter passed.

The size of all data local to the procedure cannot be determined until the procedure called is the Division of Tasks between Caller and Callee.

The call sequence is explained with the help of above Fig. 4.9.

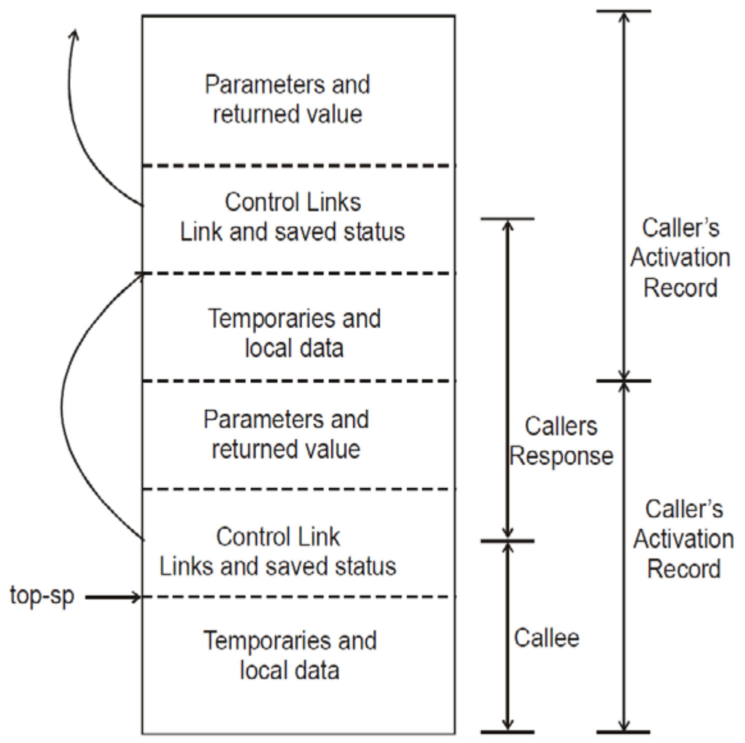

Parameters and returned value
Control Links Link and saved status
Temporaries and local data
Parameters and returned value
Control Link Links and saved status
Temporaries and local data
top-sp
Callers Response
Callee
Caller's Activation Record
Caller's Activation Record

Figure 4.3 Division of task between caller and callee

Register Top-sp $\rightarrow$ Points to the end of machine status field in the activation record.

- Position is known to the caller.
- Hence caller can be made responsible for setting top-sp before control flows to the called procedure. The code for the callee can access is temporary and local data using offsets from top-sp.

The call sequence is

1) The caller evaluates the actual.
2) The caller stores a return address and the old value of top-SP in the callee's activation record. The caller increments top-sp. The top-sp is moved past the caller's local data, temporaries the callee's parameter and status fields.
3) Callee saves register values and status formation.
4) Callee initializes it's local data and begins execution.

For Return the Sequence is

(1) Callee places return value next to the activation record of the caller.

(2) Callee restores top-sp and other registers to a return address in the caller's code. The return address in the caller's code.

(3) Top-sp to be determined. Caller copies the returned value to its activation record and use the same for expression evaluation.

The target code of the caller knows the number of arguments it supplies to the callee. Hence the target code of the callee must be prepared to handle other calls also.

The above is used in printf( ) C language.

4.2.1 ACCESS TO NON-LOCAL DATA ON THE STACK

Strategy to handle variable length data is given in Fig. 4.10.

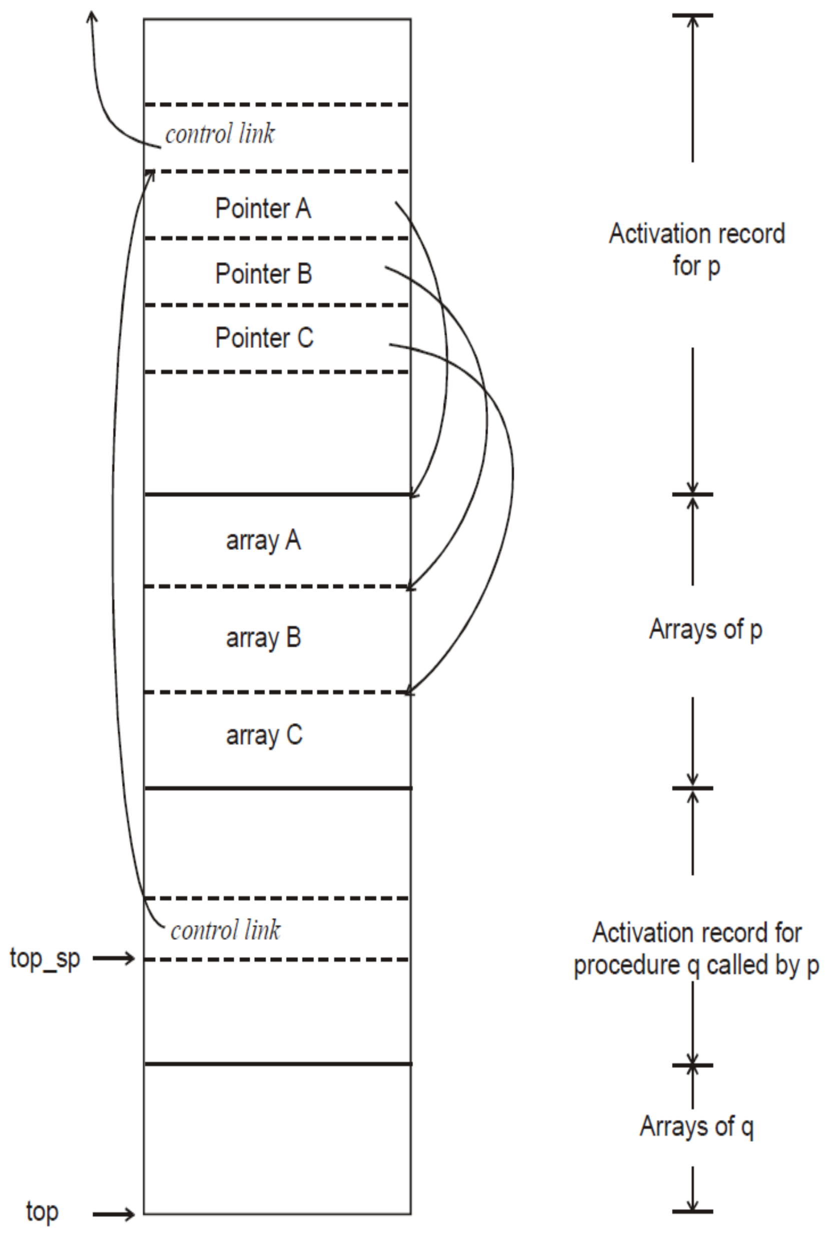


Figure 4.4

The storage for these arrays are not part of the activation record.

A pointer to the beginning of each array appears in the activation record. Relative address of pointers is known at compile time hence the target code can access the array elements via. the pointers.

Procedure p, q:

Procedure *q* is called by *p*. The activation record for q begin after the arrays of *p*, and variable length arrays of *q*.

Access to data on stack is via. two pointers top and top-sp.

Top: actual top of the stack.

- points to the position at which the next action time record begin.

Top-SP: used to find the local data.

Dangling References:

A dangling reference occurs when there is a reference to storage that has been de-allocated. It is a logical error to use dangling references. Since the value of the de-allocated storage is undefined according to the semantics of the language.

Draw Backs:

Stack allocation cannot be used under the following:

(i) The value of the local name must be retained when the activation ends.

(ii) A called activation outlines the caller.

Not possible in languages which user the activation trees.

4.3 HEAP ALLOCATION

Heap allocation overcomes the drawback of stack allocation.

In heap pieces of configures storage are allotted when needed.

The difference between heap and stack can be shown in Fig. 4.5.

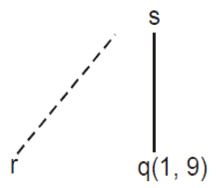
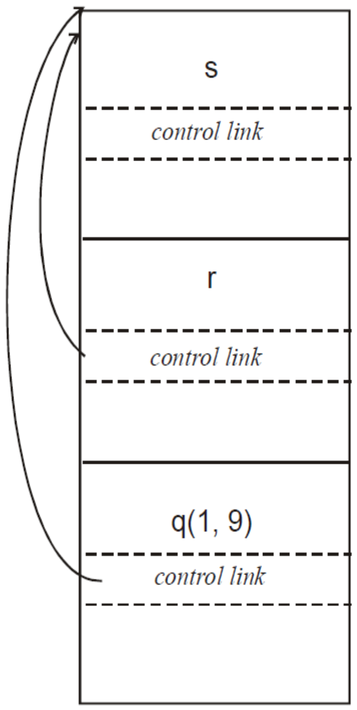
Position in the Activation Tree	*Activation Records in the Heap*	*Remarks*
	 s control link r control link q(1, 9) control link	Retained activation record for *x*

Figure 4.5

In heap the record for an activation of a procedure *r* is retained when the activation ends. If this retained activation record is deallocated there will be space in the heap between the activa- tion records.

The drawback of heap is that of time and space over head.

It is indeed helpful to handle small activation records or records of a predictable size.

(i) For each size of interest a linked list of free blocks is maintained.

(ii) Fill a request for size S with a block of size S’. Where S’ is smallest size greater or equal to S.

(iii)For large blocks use heap manager.

Advantage:

1) Fast allocation and deallocation of small amount of storage.
2) For large amount of storage, the computation takes time which is negligible when compared to the time taken for computation.

4.4 CODE GENERATION

The code generation takes as input an intermediate representation of the source program and produces as output an equivalent target program.

Position of a Code Generator

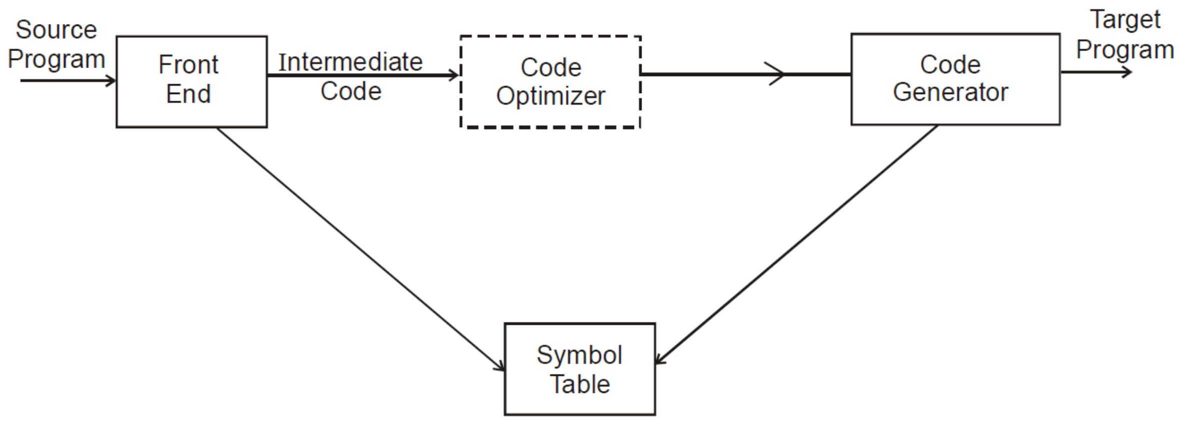


Figure 4.6

Requirements imposed on a code generation are:

- Output must be correct and of high quality meaning that, it should make efficient use of the resources of the target machine.
- The code generator should run efficiently.

<u>4.4.1 ISSUES IN THE DESIGN OF A CODE GENERATOR</u>

Input to the Code Generators

The input to the code generator consists of the intermediate representation of the source program produced by thefront end, together with information in the symbol table.

There are several choices for the intermediate language including linear representation such as postfix notation, three address representations such as quadruples, triples, etc. graphical representations such as syntax directed trees and dags, etc.

We assume that prior to code generation the front end has scanned, parsed and translated the source program into a reasonably detailed intermediate representation. The code generation phase, can therefore are proceed on the assumption that its input is free of errors.

Target Program:

The output of the code generator is the target program.

This output may take on a variety of forms.

(1) absolute machine language

(2) relocatable machine language

(3) assembly language.

Advantage of absolute machine language program as output:

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

Relocatable Machine Language:

- Producing a relocatable machine language program as output allows subprograms to be compiled separately.
- A set of relocatable object modules can be linked together and loaded for execution by a linking loader.

Assembly languge program:

- Producing an assembly language program as output makes the process of code generation somewhat easier.
- This choise is an alternative for a machine with a small memory.

Memory Management:

Mapping names in a source program to addresses of data objects in runtime memory is done comparatively by the front end and the code generator.

A name in a three address statement refers to a symbol table entry for the name. From the symbol table information the relative address can be determined for the name.

If machine code is being generated, labels in three address statements have to be converted to address of instructions.

If a reference such as:

j: goto i is encountered and ‘i’ is less j, the current quadruple number, we may simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple “i”.

If however the jump is forward. So ‘i’ exceeds ‘j’, we must store on a list for quadruple i the location of the first machine instruction of j. Then when we process i, we fill in the proper machine location for all instructions that are forward jumps to i.

Example: If “i” is > 100 and ‘j’ is 100.

We have 100 : goto i.

The list for ‘i’ will now have:

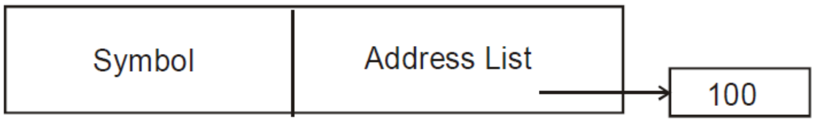


Figure 4.7

‘We go to the location 100 in the list and assign it as [goto 200].

Instruction Selection:

The nature of the instruction set of the target machine determines the difficulty of instruction selection. Improtant factor which influence instruction selection.

- The uniformity and completeness of the instruction set.
- Instruction speeds and machine idioms.

Every three address statement of the form x := y + z can be translated into the code sequence.

MOV y, R_0

ADD z, R_0

MOV R_0 , X.

This kind of statement-by-statement, code generation often produces poor code for example the sequence of statements

a := b + c

d := a + e

would be translated into:

MOV b, R_0

ADD C, R_0

MOV R_0, a

ADD e, R_0

Here the third and fourth statement is redundant if a is not subsequently used.

The quality of the generated code is determined by its speed and size. Atarget machine with a such instruction set may provide several ways of implementing a given operation. For example, if the target machine has an increment instruction, then the three address statement a := a + 1 may be implemented more efficiently by the single instruction INC a rather than a sequence that loads ‘a’ into a register adds one to the register and then stores the result back in a,

```
MOV  a, R0
ADD  #1, R0
MOV  R0 , a
```

Instruction speeds are also needed to design good code sequences but accurate timing information is often difficult to obtain.

Register Allocation

Instructions involving register operands are usually shorter and faster than those involving operands is memory. Therefore efficient utilization of registers is particularly important in generating good code.

The use of registers is often

Subdivided into two problems.

1) During register allocation we select the set of variables that will reside in register at a point in the program.
2) During a subsequent register assignment phase we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult consider the following three address codes which follows:

```
t := a + b
t := t + c
```

Three address code sequence.

```
LOAD     R0, a
ADD      R0, b        Optimal machine code sequences.
ADD      R0, c
STORE    R0, t
```

In this case the optimal choice for the register into which ‘a’ is to be loaded depends on what will ultimately happen to ‘t’.

Choice of Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. But picking a best order is difficult.

Approaches to Code Generations

The most important criterion for a code generator is that it produce correct code.

Code generator has to be designed so that it can be easily implemented, tested and maintained. The code generation algorithm which will be discussed uses information about subsequent uses of an operand to generate code for a register machine. It considers each statement in turn keeping operands in registers as long as possible.

4.4.2 INTERMEDIATE CODE GENERATOR

The intermediate code generator receives the source program that is parsed, syntatically checked and represents them in any one of the following intermediate codes,

(a) Syntax Tree (b) Postfix Expression (or) Notation (c) Three Address Code.

Syntax Tree

Syntax tree depicts the natural hierarchical structure of source program.

A Directed Acyclic Graph (DAG) also gives the same information of a syntax tree but it identifies common sub-expressions. A syntax tree and DAG for the assignment statement *a* = *b* * *c* + *b* * *c* appear in the following Figure.

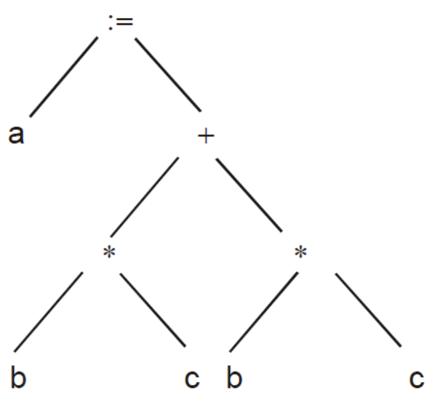


Figure 4.8 Syntax tree

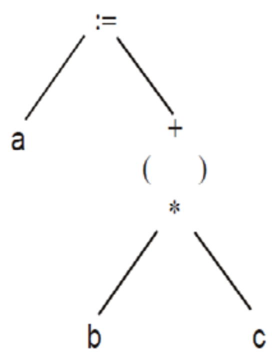


Figure 4.9 DAG

Two Representations of the Syntax Tree:

Each node is represented as a record with a field for its operator and additional fields for pointers to its children.

a = *b* * *c* + *b* * *c*

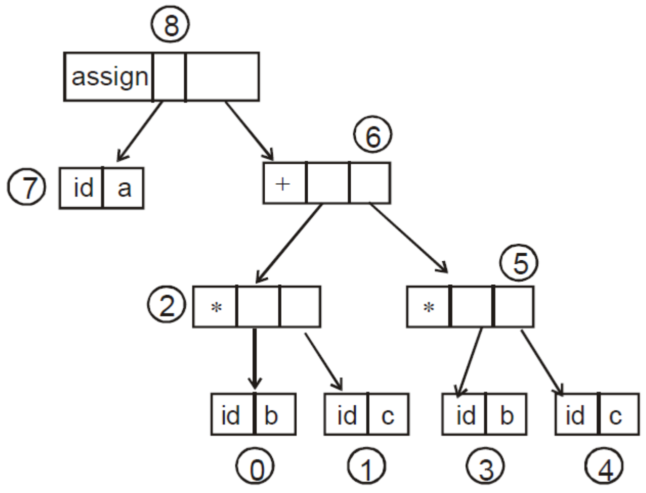


Figure 4.10

In another method nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node as shown below.

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

POSTFIX NOTATION

It 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 preceding syntax tree is *bc* * *bc* * + *a* =.

4.5 A SIMPLE CODE GENERATOR

The code generation strategy generates target code for a sequence of three address statement. It considers each statement in turn, remembering if any of the operands of the statement are currently in register and taking advantage of that fact if possible. This is done because registers can be accessed much more faster than memory locations. Computed results can be left in registers as long as possible storing then only.

- if their register is needed for another ocmputation (or)
- just before a procedure call, jump or labeled statement.

  We can produce reasonable code for a 3 address statement $a := b + c$ as ADD R_j , R_i if R_j contains C , R_i constants b and b' is not live after the statement. (The result will be stored in the register which had the values b).

- If R_i contains b, but 'c' is in a memory location. We can generate the sequence:

```
ADD   C1Rj Cost = 2
```

Provided 'b' is not subsequently live, and 'c' is not subsequently used.

(or)

```
MOV       C, Rj
ADD       Rj , Ri       Cost = 3
```

if the value of 'c' is subsequently used, and 'b' is not live.

A Code-Generation Algorithm

1) Invoke a function *getreg* to determine the location L where the result of the computation *y op z* should be stored. L will usually be a register, but it could also be a memory location.
2) Consult the address descriptor for y to determine *y′*, (one of) the current location(s) of *y*.
3) Generate the instruction OP *z′*, L where *z′* is a current location of *z*.
4) If the current values of *y* and/or *z* have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of *x*:=*y op z*, those registers no longer will contain *y* and/or *z*, respectively.

The Function getreg

The function *getreg* returns the location L to hold the value of *x* for the assignment *x* := *y op z*.

1) If the name *y* is in a register that holds the value of no other names and *y* is not live and has no next use after execution of *x* := *y op z*, then return the register of *y* for L. Update the address descriptor of y to indicate the y is no longer in L.

2) If one fails return empty register for L.
3) If two fails and if *x* has next to use that requires a register find a occupied register R.
4) If *x* is not used in the block or register can be found for *x* then store *x* in the memory.