RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Compiler Design**

Subject Code: **IT-603**

Semester: **6th**

Department of Information Technology
Subject Notes
IT603 (A) – Compiler Design
B.Tech, IT-6th Semester

Unit IV

Syllabus: Intermediate Code Generation: Three address code, quadruple & triples, translation of assignment statements, Boolean expression and control structures, Backpatching, Run Time Memory Management: Static and Dynamic storage allocation, stack based memory allocation schemes, Symbol Table management.

Unit Outcome: Describe techniques for intermediate code and machine code optimization.

Unit 4: Intermediate Code Generation

Intermediate Code:

The task of compiler is to convert the source program into machine program. This activity can be done directly, but it is not always possible to generate such a machine code directly in one pass. Then, typically compilers generates an easy to represent form of source language which is called intermediate language. The generation of an intermediate code leads to efficient code generation.

Benefits of Intermediate Code Generation:

1. A compiler for different machines can be created by attaching different backend to the existing front ends of each machine.
2. A compiler for different source languages (on the same machine) can be created by proving different front ends for corresponding source languages to existing back end.
3. A machine independent code optimizer can be applied to intermediate code in order to optimize the code generation.

Role of intermediate code generator in compiler:

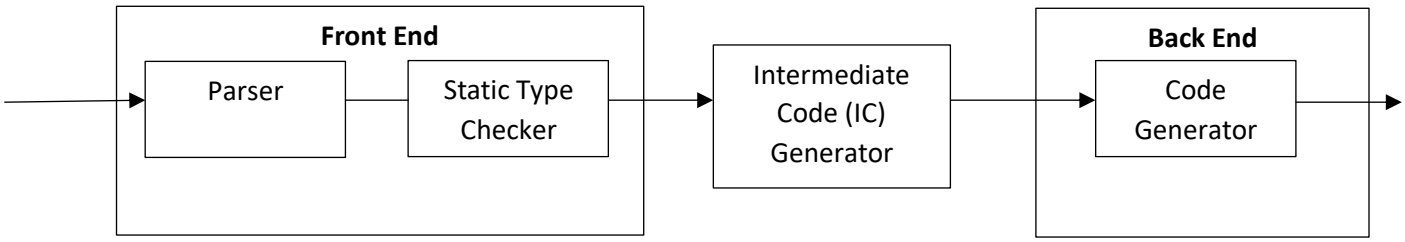

Figure 4.1: Intermediate Code Generator in Compiler

Forms of Intermediate Code:

Mainly three types of Intermediate Code representation are there:

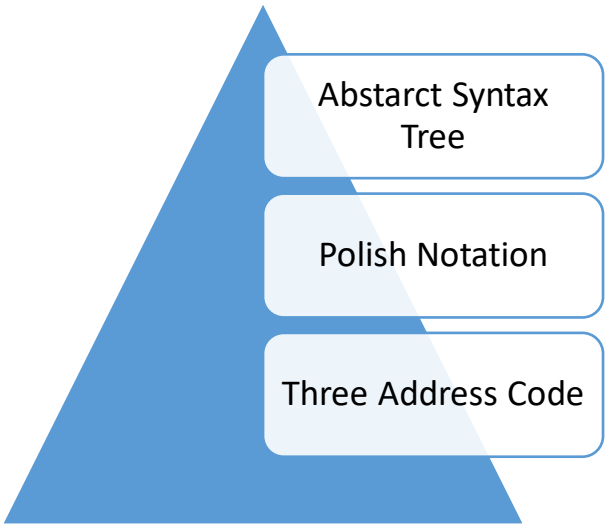

Figure 4.2: Forms of Intermediate Code

Abstract Syntax Tree:

The natural hierarchical structure is represented by syntax trees. The code being generated as intermediate should be such that the remaining processing of the subsequent phases should be easy.

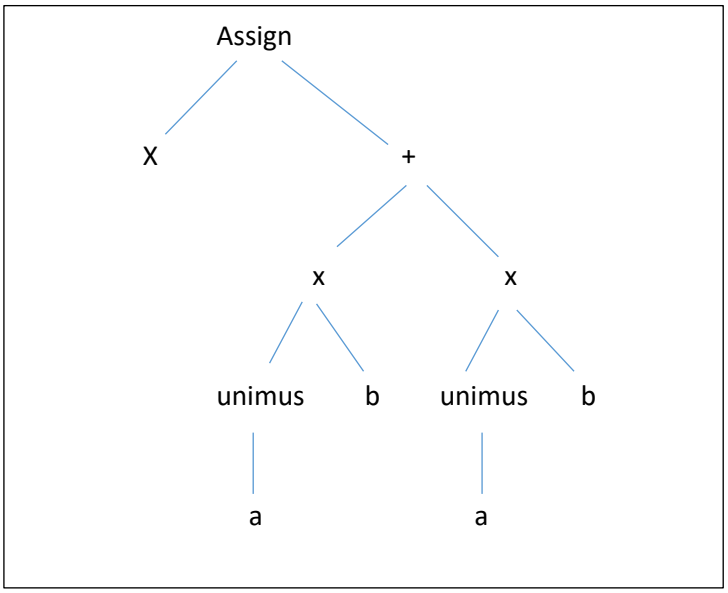

Figure 4.3: Abstract Syntax Tree

Polish Notation:

Basically, the linearization of syntax tree is polish notation. In such representation, the operator can be easily associated with the corresponding operands. This is the most natural way of representation in expression evaluation.

The polish notation is also called as Prefix notation in which the operator occurs first and then operands are arranged.

Example:

(a+b)*(c-d) can be written as: +ab-cd

Three Address Code:

In three address code form at the most three addresses are used to represent any statement. The general form of three address code representation is:

a := b op c

where a,b or c are the operands that can be names, constants, compiler generated temporaries and op represents the operator. The operator can be fixed or floating point arithmetic operator or logical operator on Boolean valued data. Only single operation at right side of the expression is allowed at a time.

For the expression a= b+c+d the three address code will be

t_1 := b+c

t_2 := t_1+d

a := t_2

Here t_1 and t_2 are the temporary names generated by the compiler. There are at the most three addresses allowed (two for operands and one for result). Hence, the name of this representation is three-address code.

Implementation of Three Address Code:

Three address code is an abstract form of intermediate code that can be implemented as a record with the address fields. There are three representations used for three address code such as quadruples, triples and indirect triples.

Quadruple Representation:

The quadruple is a structure with at the most four fields such as op, arg1, arg2, result. The op field is used to represent the internal code for operator, the arg1 and arg2 represent the two operands used and result field is used to store the result of an expression.

Example: Consider the input statement x := - a * b + - a * b

The three address code is

t_1 := uminus a

t_2 := t_1-b

t_3 := -a

t_4 := t_3 * b

t_5 := t_2 + t_4

x := t_5

	Op	Arg1	Arg2	Result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	A		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		x

Figure 4.4: Three Address Code and Quadruple Representation

Triples Representation:

In the triples representation the use of temporary variables is avoided by referring the pointers in the symbol table.

Example: x := - a * b + - a * b

Number	Op	Arg1	Arg2
(0)	uminus	A	
(1)	*	(0)	B
(2)	uminus	A	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	X	(4)

Table 4.1: Triples Representation

Indirect Triples:

In this representation the listing of triples is been done and listing pointers are used instead of using statements.

Number	Op	Arg1	Arg2			Statement
(0)	uminus	a			(0)	(11)
(1)	*	(11)	b		(1)	(12)
(2)	uminus	a			(2)	(13)
(3)	*	(13)	b		(3)	(14)
(4)	+	(12)	(14)		(4)	(15)
(5)	:=	x	(15)		(5)	(16)

Table 4.2: Indirect Triples

Translation of Assignment Statements:

The assignment statement mainly deals with the expressions. The expressions can be of type integer, real, array and record.

Example: Obtain the translation scheme for obtaining the three address code for the grammar-

S → id:= E

E → E1 + E2

E → E1 * E2

E → -E1

E → (E1)

E → id

Solution: The translation scheme which is used to generate the three address code

Production Rule	Semantic Rule

S → id:= E	{ id_entry:=look_up(id.name);
	if id_entry != nil then
	append (if=id_entry ':=' E.place)
	else error; /* id not declared*/
	}

E → E1 + E2	{ E.place:= newtemp(); append(E.place ':=' E1.place '+' E2.place) }
E → E1 * E2	{ E.place:= newtemp(); append(E.place ':=' E1.place '*' E2.place) }
E → -E1	{ E.place:= newtemp(); append(E.place ':=' 'uminus' E1.place) }
E → (E1)	{E.place:=E1.place}

E → id	(id_entry:=look_up(id.name);
	If id_entry != nil then
	append(id_entry ':=' E.place)
	else error; /*id not declared*/
	}

Table 4.3: Translation of Assignment Statement

- The look_up returns the entry for id.name in the symbol table if exists there.
- The function append is for appending the three address code to the output file. Otherwise an error will be reported.
- newtemp() is the function for generating new temporary variables.
- E.place is used to hold the value of E.

Boolean Expression:

Normally there are two types of Boolean expressions used:

1. For computing the logical values.
2. In conditional expression using if-then-else or while-do

Consider the Boolean expression generated by following grammar:

E → E OR E

E → E AND E

E → E NOT E

E → (E)

E → id relop id

E → TRUE

E → FALSE

Here the relop is denoted by <=, >=, !=, <, >. The OR & AND are left associate. The highest precedence is to NOT then AND & lastly OR.

Numerical Representation of Boolean Expression:

The translation scheme for Boolean expression having numerical representation is as given below:

Production Rule	Semantic Rule

E → E1 OR E2	{
	E.place:=newtemp()
	append(E.place ':=' E1.place 'OR' E2.place)
	}

E → E1 AND E2	{
	E.place:=newtemp()
	append(E.place ':=' E1.place 'AND' E2.place)
	}

E → NOT E1	{
	E.place:=newtemp()
	append(E.place ':=' 'NOT' E1.place)
	}

E → (E1)	{
	E.place := E1.place
	}

E → id, relop id2	{
	E.place := newtemp()
	append('if' id1.place relop.op id2.place 'goto' next_state +3);
	append(E.place ':=' '0');
	append('goto' next_state +2);
	append(E.place := '1')
	}

E → TRUE	{
	E.place := newtemp();
	append(E.place ':=' '1')
	}

E → FALSE	{
	E.place := newtemp()
	append(E.place ':=' '0')
	}

Table 4.4: Numerical Representation of Boolean Expression

Flow of Control Statement in Boolean Expression:

The control statements are if-then-else and while-do. The grammar for such statements is as shown below:

S → if E then S1

| if E then S1 else S2

| while E do S1

While generating three address code:

- To generate new symbolic label the function new_label() is used.
- With the expression E.true and E.false are the labels associated.
- S.code and E.code is for generating three address code.

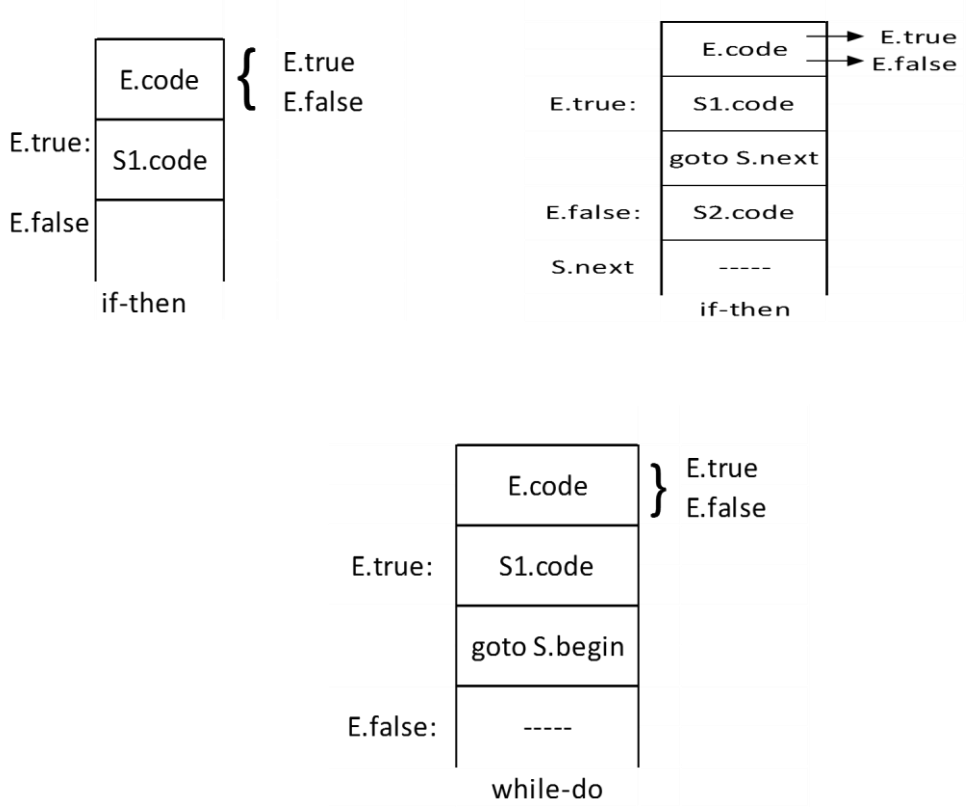

Figure 4.5: Flow of Control Statement in Boolean Expression

S → if E then S1

E.true := new_label()

E.false := S.next

S1.next := S.next

S.code := E.code || gen_code(E.true':') || S.code

In the above translation scheme || is used to concatenate the strings. The function gen_code is used to evaluate the non-quoted arguments passed to it and to concatenate complete string. The S.code is the important rule which ultimately generates the three address code.

Consider the statement is if a<b then a = a+5 else a = a+7

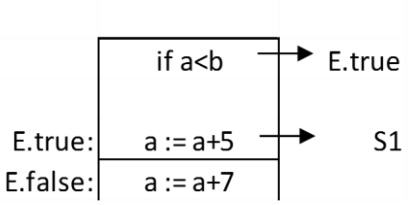

Figure 4.6

The three address code for if-then is:

100: if a<b goto L1

101: goto 103

102: L1: a = a +5 /*E.ture*/

103: a = a+7

Here E.code is "if a<b" L1 denotes E.true and E.false is shown by jumping to line 103 (that is S.next)

Back patching:

Implementation of syntax directed definition using two passes is the most convenient method. If user decide to generate the three address code for given syntax directed definition using single pass only, then the main problem that occurs is the decision of address of the labels. The jump(goto) statements refer these label statements and in one pass it becomes difficult to know the locations of these label statements. If user use two passes instead of one pass then in one pass user can leave these addresses unspecified and in the second pass this incomplete information can be filled up. To overcome the problem of processing the incomplete information in one pass the back patching technique is used.

Back patching is the activity of filling up unspecified information of labels using appropriate semantic actions in during the code generation process.

To generate code using back patching, in the semantic actions following functions are used:

- *mklist(i)* creates the new list. The index i is passed as an argument to this function where i is an index to the array of quadruple.
- *merge_list(p1,p2)* this function concatenates two lists pointed by p1 and p2. It returns the pointer to the concatenated list.
- *backpatch(p,i)* inserts i as target label for the statement pointed by pointer p.

Run Time Memory Management:

A program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.

By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system. Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

There are three different storage allocation strategies based on run time storage:

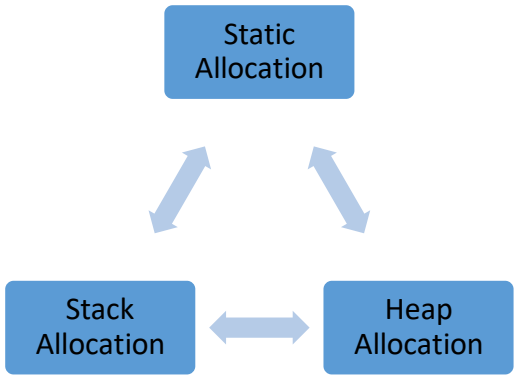

Figure 4.7: Storage Allocation Strategies

Static Allocation:

- The size of data objects is known as compile time. The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.
- The binding of name with the amount of storage allocated do not change at run time. Hence the name of this allocation is static allocation.
- In static allocation the compiler can determine the amount of storage required by each data object. And thereof it becomes easy for a compiler to find the addresses of these data in the activation record.
- At compile time, compiler can fill the address at which the target code can find the data it operates on.
- FORTRAN uses the static allocation strategy.

Stack Allocation:

- Stack allocation strategy is a strategy in which the storage is organized as stack. This stack is also called control stack.
- As activation begins the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be popped.
- The locals are stored in the each activation record. Hence local are bound to corresponding activation record on each fresh activation.
- The data structures can be created dynamically for stack allocation.

Dynamic Storage Allocation:

The data under program control can be allocated dynamically. This allocation is done from heap memory. There are two approaches used to allocate data - the explicit allocation and implicit allocation.

Explicit Allocation: It is the kind of memory allocation that can be done with the help of some procedure.

For example: In PASCAL the memory allocation is done using new function and de-allocation is done using dispose.

Implicit Allocation: It is a kind of allocation that can be done automatically for storing the results of expression evaluation. Various languages that handle implicit allocation are:

- LISP allocates a memory for the cell of a list when cons is used.

- SNOBOL allows a string of varing length. The space required for varied length of the string is managed at run time. This space is taken from the heap memory.

Garbage is a substantial amount of memory which gets allocated dynamically but is unreachable memory. If this memory remains unreachable then it becomes a wastage of memory. Hence a new technique called garbage collection is introduced. Java and LISP are those language which perform garbage collection.

Dangling Reference is a kind of complication that occurs due to explicit de-allocation of memory. The dangling reference is a kind of reference that gets de-allocated which is referred to.

Explicit Allocation:

The explicit allocation can be done for fixed size and variable size blocks.

Explicit Allocation for Fixed Size Blocks:

- This is the simplest technique of explicit allocation in which the size of the block for which memory is allocated is fixed.
- In this technique a free list is used. Free list is a set of free blocks. This list is observed when we want to allocate memory. If some memory is de-allocated then the free list gets appended.
- The blocks are linked to each other in a list structure. The memory allocation can be done by pointing previous node to the newly allocated block. Similarly memory de-allocation can be done by de referencing the previous link.
- The pointer which points to first block of memory is called Available.
- This memory allocation and de-allocation is done using heap memory. The explicit allocation consists of taking a block off the list and de-allocation consist of putting the block on the list.
- The advantage of this technique is that there is no space overhead.

Explicit Allocation of Variable Sized Blocks:

Due to frequent memory allocation and de-allocation the heap memory becomes fragmented. That means heap may consist of some blocks that are free and some that are allocated.

For allocating variable sized blocks some strategies such as first fit, worst fit and best fit are used. When a block of size S is allocated and if we search for the free block in the heap memory and the first available free block whose size > S is used then that technique is called first fit method.

Implicit Allocation:

The implicit allocation is performed using user program and runtime package. The run time package is required to know when the storage block is not in use.

Block Size (optional)
Reference Count (optional)
Mark (optional)
Pointers to blocks
User Data

Table 4.5: Block Format of Implicit Allocation

There are two problems that are faced in implicit memory allocation- First problem is recognizing block boundaries. If the block size is fixed then user data can be easily accessible. Second problem is to determine which block is in use.

There are two approaches used for implicit allocation:

Reference Count:

Reference count is a special counter used during implicit memory allocation. If any block is referred by some another block then its reference count incremented by one. That also means if the reference count of particular block drops down to 0 then, that means that block is not referenced one and hence it can be de-allocated. Reference count are best used when pointers between blocks never appear in cycle.

Marking Techniques:

This is an alternative approach to determine whether the block is in sue or not. In this method, the user program is suspended temporarily and frozen pointers are used to mark the blocks that are in use. Sometime bitmaps are used. Bitmaps are used to mark the blocks which are in use. A bit-table stores all the bits. This table indicates the blocks which are in use currently.

There is one more technique called compaction in which all the used blocks are moved at the one end of heap memory, so that all the free blocks are available in one large free blocks.

Symbol Table:

Symbol table is a data structure used by compiler to keep track of semantics of variable. That means symbol table stores the information about scope and binding information about names.

Symbol table is built in lexical and syntax analysis phases.

The symbol table is used by various phases as follows:- Semantic analysis phase refers symbol table for type conflict issue. Code generation refers symbol table knowing how much run-time space is allowed? What type of run-time space is allocated?

Stack Based Memory Allocation:

Stack based memory allocation is very simple and typically faster than heap-based memory allocation (also known as dynamic memory allocation). Memory on the stack is automatically, and very efficiently, reclaimed when the function exits, which can be convenient for the programmer if the data is no longer required. If however, the data needs to be kept in some form, then it must be copied from the stack before the function exits. Therefore, stack based allocation is suitable for temporary data or data which is no longer required after the creating function exits.

A thread's assigned stack size can be as small as only a few bytes on some small CPUs. Allocating more memory on the stack than is available can result in a crash due to stack overflow.

Symbol Table Management:

Requirements for symbol table management

- For quick insertion of identifier and related information.
- For quick searching of identifier.

List data structure for symbol table:

- Linear list is a simplest kind of mechanism to implement the symbol table.
- In this method an array is used to store names and associated information.
- New names can be added in the order as they arrive. The pointer 'available' is maintained at the end of all stored records.
- To retrieve the information about some name we start from beginning of array and go on searching up to available pointer. If we reach at pointer available without finding a name we get an error "use of undeclared name".
- While inserting a new name we should ensure that it should not be already there. If it is there another error occurs "Multiple defined Name".
- The advantage of list organization is that it takes minimum amount of space.

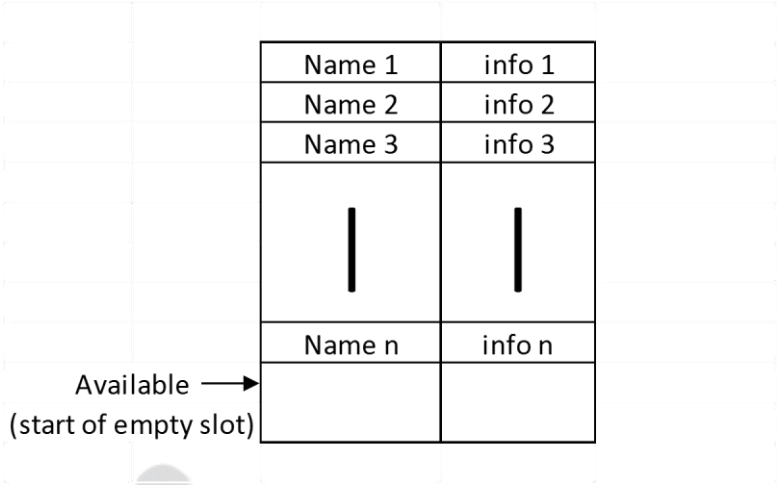

Figure 4.8: List Data Structure using Array

Self-organizing list:

This symbol table implementation is using linked list. A link field is added to each record.

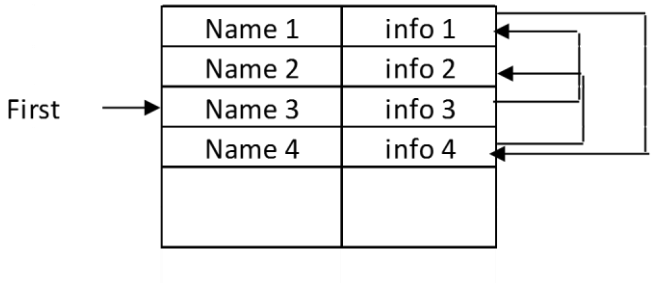

Figure 4.9: Self-organizing List

- We search the records in the order pointed by the link of link field.
- A pointer "First" is maintained to point to first record of the symbol table.
- When the name is referenced or created it is moved to the front of the list.
- The most frequently referred names will tend to be front of the list. Hence access time to most frequently referred names will be the least.

Hash Table:

- Hashing is an important technique used to search the records of symbol table. This method is superior to list organization.

- In hashing scheme two tables are maintained: hast table and symbol table.
- The hash table consists of k entries from 0,1 to k-1. These entries are basically pointers to symbol table pointing to the names of symbol table.
- To determine whether the 'Name' is in symbol table, we use a hash function 'h' such that h(name) will result any integer between 0 to k-1. We can search any name by position = h(name)
- Using this position we can obtain the exact location of name in symbol table.
- The hash function should result in uniform distribution of names in symbol table.
- The hash function should be such that there will be minimum number of collision. Collision is such a situation where hash function results in same location for storing the names.
- The advantage of hashing is quick search is possible and the disadvantage is that hashing is complicated to implement. Some extra space is required obtaining scope of variables is very difficult.

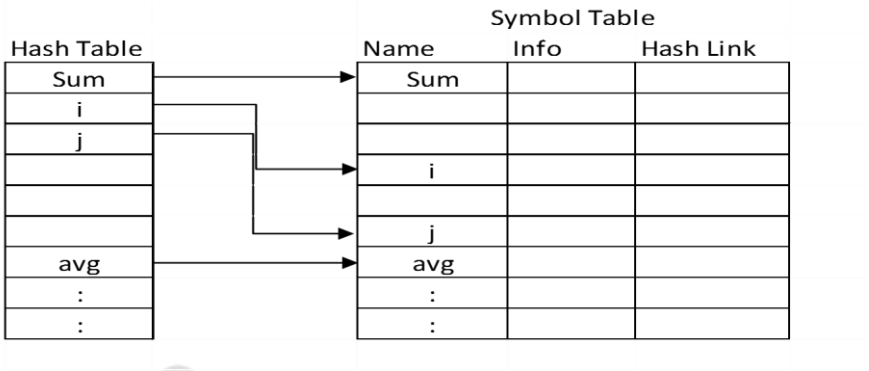

Figure 4.10: Hashing for Symbol Table

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in