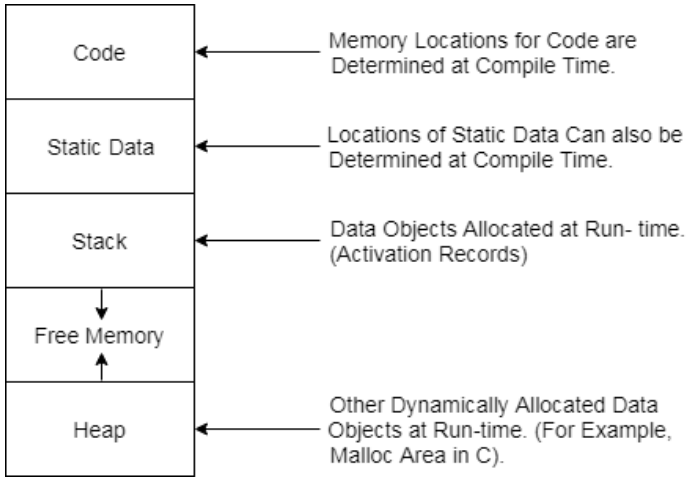
UNIT V

Run Time Environments: Storage Organization, Run Time Storage Allocation, Activation Records, Procedure Calls, Displays

Code Generation: Issues in the Design of a Code Generator, Object Code Forms, Code Generation Algorithm, Register Allocation and Assignment

Storage Organization

The compiler demands for a block of memory to operating system. The compiler utilizes this block of memory for executing the compiled program. This block of memory is called run time storage.



The runtime storage is sub divided into :

1. Code Area.
2. Static data Area.
3. Stack Area.
4. Heap Area.

Code Area: holds the instructions to be executed, which is static & size is determined at compile time.

Heap Area: allocated memory dynamically on demand for various purposes, dynamically allocated like malloc()

Stack Area: store the return address, dynamic variables & local variables declared inside the function, size is determined only during the runtime

Static data: holds the global data and any other data that could be determined at compile time.

Run Time Storage Allocation

There are three different storage allocation strategies based on this division of runtime storage. The strategies are:

1. Static allocation: the static allocation is for all the data objects at compile time.
2. Stack allocation: in the stack is used to manage the runtime storage.
3. Heap allocation: in heap allocation the heap is used to manage the dynamic memory allocation.

Static Allocation

- The size of data objects is known at 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 time.
- The binding of name with the amount of memory allocated do not change at runtime. Hence the name of this allocation is static allocation.
- In static the compiler determine the storage required by each data object, then compiler find the addresses of these data in the activation record.
- At compile time it fills the addresses at which the target code can find the data operates on it.
- FORTRAN uses the static allocation strategy.

Stack Allocation

- Stack allocation strategy is in which the storage is organized as stack. This stack is also called control stack.
- When Activation begins the activation records are pushed on to the stack when it completes activation records are popped off from the stack.
- The local data is stored in the activation record. The data structure is can be created dynamically for stack allocation.

Heap Allocation

The heap allocation allocates the continuous block of memory when required for storage of the activation records or other data objects. This allocated memory can be de-allocated when activation ends. This de-allocation space can be further re-used by heap manager.

Activation Record

Activation Record is a memory block used for information management for single execution of a procedure.

The following is the activation record (Read from bottom to top)

Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries

Temporaries – It hold temporary values, such as the result of mathematical calculation, a buffer value or so on.

Local data – It belongs to the procedure where the control is currently located.

Saved machine status- It provides information about the state of a machine just before the point where the procedure is called.

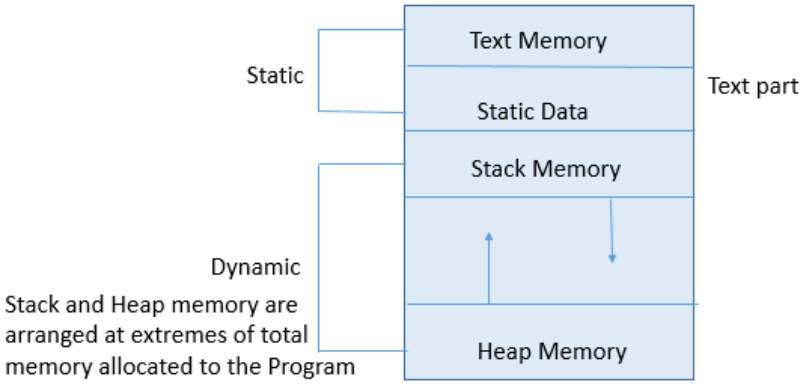
Access link – It is used to locate remotely available data. This field is optional. Control link – It points to the activation record of the procedure which called it, ie the caller. This field is optional. This link is also known as dynamic link.

Return value – It holds any valued returned by the called procedure. These values can also be placed in a register depending on the requirements.

Actual parameters – These are used by the calling procedures.

Heap Management

- Heap is the unused memory space available for allocation dynamically.
- It is used for data that lives indefinitely, and is freed only explicitly.
- The existence of such data is independent of the procedure that created it.
- Heap storage allocation supports the recursion process



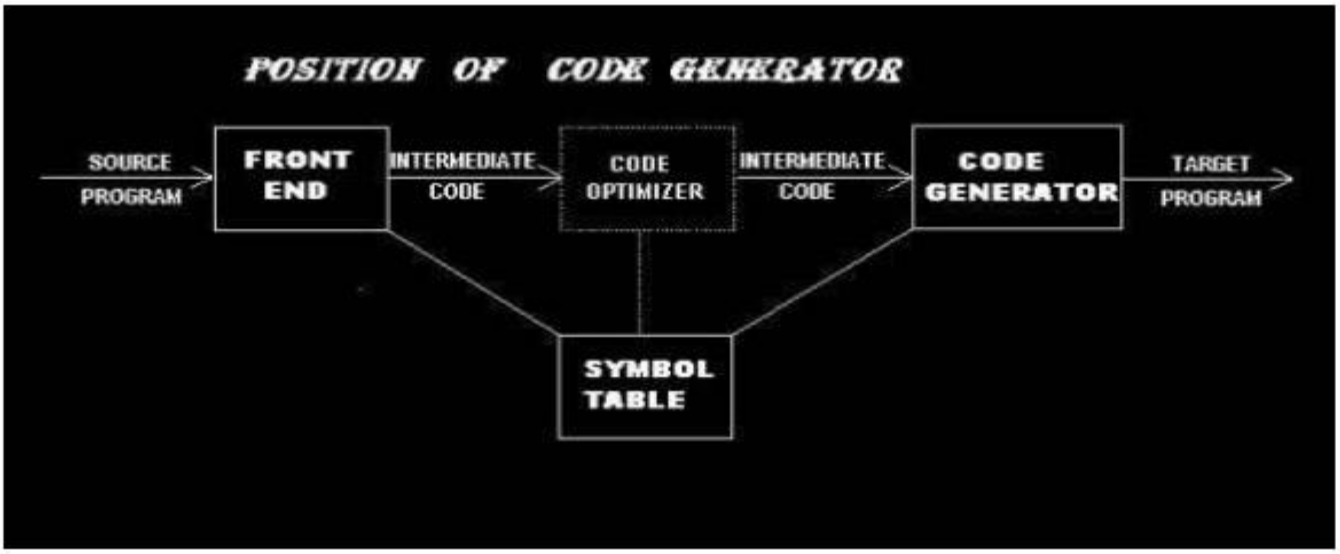
Memory manager:

Memory manager is used to keep account of the free space available in the heap area. Its functions include,

1. Allocation: The procedure allocated in the storage during the execution.
2. De-allocation: The procedure leaves the storage after completion of the execution.

Code Generation

Issues in the Design of a Code Generator



The issues in the code generator are:

1. Input to the code generation
2. Target code
3. Instruction selection
4. Register Allocation

Input to the code generation: The input to the code generation are intermediate code representation of source code and the information in the symbol table.

- The intermediate code generated by the code generator can be represented in several ways. Mainly the representation is done by three address code (TAC).
- The type checking can be done and the input is error free, so that code generation phase can proceed for desired output.

Instruction selection

- The job of instruction selector is to do a good job overall choosing which instructions to implement which operator in the low level intermediate representation.

- Issues here are:
  - level of the IR
  - nature of the instruction set architecture
  - desired quality of the generated code □Target Machine:
  - n general purpose registers
  - instructions: load, store, compute, jump, conditional jump
  - various addressing mode
  - indexed address, integer indexed by a register, indirect addressing and immediate constant.
- For example, X = Y + Z we can generate,
  LD R0, Y
  ADD R0, R0, Z
  ST X, R0

Register allocation

- Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of registers is particularly important in generating good code.
- The most frequently used variables should be kept in process register for faster access.
- The use of registers if often subdivided into two sub problems:
  - During "register allocation", we select the set of variables that will reside in register at a point in the program.
  - During "register assignment", we pick the specific register that a variables will reside in.

  For example, consider t1 = a * b
  t2 = t1 + c
  t3 = t2 / d

The optimal machine code sequence is,

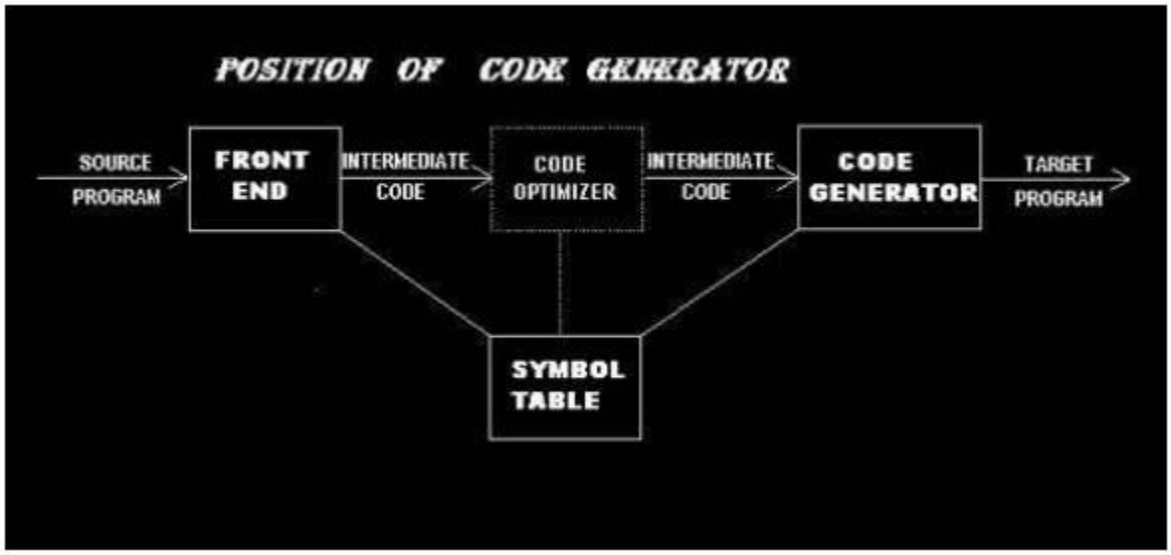
```
L     R1, a
M     R1, b
A     R2, c
D     R2, d
ST    R1, t3
```

Machine dependent code generation

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references.

Object code forms

The final phase in our compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.
The requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.



After compilation of the source code, the object code is generated, which not only contains machine level instructions but also information about hardware registers, memory address of some segment of the run-time memory (RAM), information about system resources, read-write permissions ..etc.

- The output of Code Generation (CG) is the target language.
- The output may take different forms like absolute machine language, re-locatable machine language or assembly language.

The operations performed in the target code are:
The possible kinds of operations are listed below:

1. Load operation (LD)

- General format is LD dst, addr.
- LD loads the value in location 'addr' into location 'dst'. Here dst is destination.
- For example, LD R1, x → loads the value in location x into register R1.

2. Store operation (ST)

- General format is ST dst, src.

- For example, ST x, R1 → stores the value in register R1 into the location x.

3. Computation operation

- The general form is OP dst, src1, src2 where OP is a operator like ADD, SUB.
- For example, ADD R1, R2, R3 → adds R2 and R3 values and stores into R1.

4. Unconditional jump (BR)

The general form is BR L where BR is branch. This causes control to branch to the machine instruction with label L.

5. Conditional jump

- The general form is Bcond R, L where R is a register, L is label and cond stands for any of the common tests on values in register R.

For example, BGTZ R2, L → This instruction causes a jump to label L if the value in register R2 is greater than zero and allows control to pass to the next machine instruction if not.

The below example performs target code a=b+c; d=e+f;

MACHINE CODE			REGISTER TRANSFER
MOV	R1	b	R1 ← m[b]
ADD	R1	c	R1 ← R1+m[c]
MOV	a	R1	M[a] ← R1
MOV	R2	e	R2 ← m[e]
ADD	R2	f	R2 ← R2+m[f]
MOV	d	R2	m[d] ← R2

Addressing modes in the Target code:

1. Direct addressing
2. Indirect addressing
3. Immediate addressing
4. Indexed addressing

Direct addressing

MOV 1000, R0 ------move the content of data at the location 1000 into R0

Indirect addressing

MOV (R1), R0------- move the content of the data at the location addressed by R1 into R0

Immediate addressing

MOV #1000, R0------move the immediate data 1000 into R0

Indexed addressing

(a). MOV 1000(R1), R0 ---------move the content of data at(R1+1000)into R0

(b). MOV * 1000(R1), R0-------- move the content of the data at a location whose address is at (R1+1000) into R0

Generic code generation algorithm

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form **x : = y op z**, perform the following actions:
Generating Code for Assignment Statements:
The assignment **d : = (a-b) + (a-c) + (a-c)** might be translated into the following three- address code sequence:

t : = a – b
u : = a – c
v : = t + u
d : = v + u

with d live at the end.
Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a – b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a – c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0

d : = v + u	ADD R_1, R_0
	
	MOV R_0, d
		R
		0 contains d
			d in R_0
			d in R_0 and memory

Register allocation and assignment

Instructions involving register operands are usually shorter and faster than those involving operands in memory.
Therefore, efficient utilization of register is particularly important in generating good code. The use of registers isoften subdivided into two sub problems:

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

For example, consider t1 = a * b

t2 = t1 + c

t3 = t2 / d

The optimal machine code sequence is,

```
L     R1, a
M     R1, b
A     R2, c
D     R2, d
ST    R1, t3
```