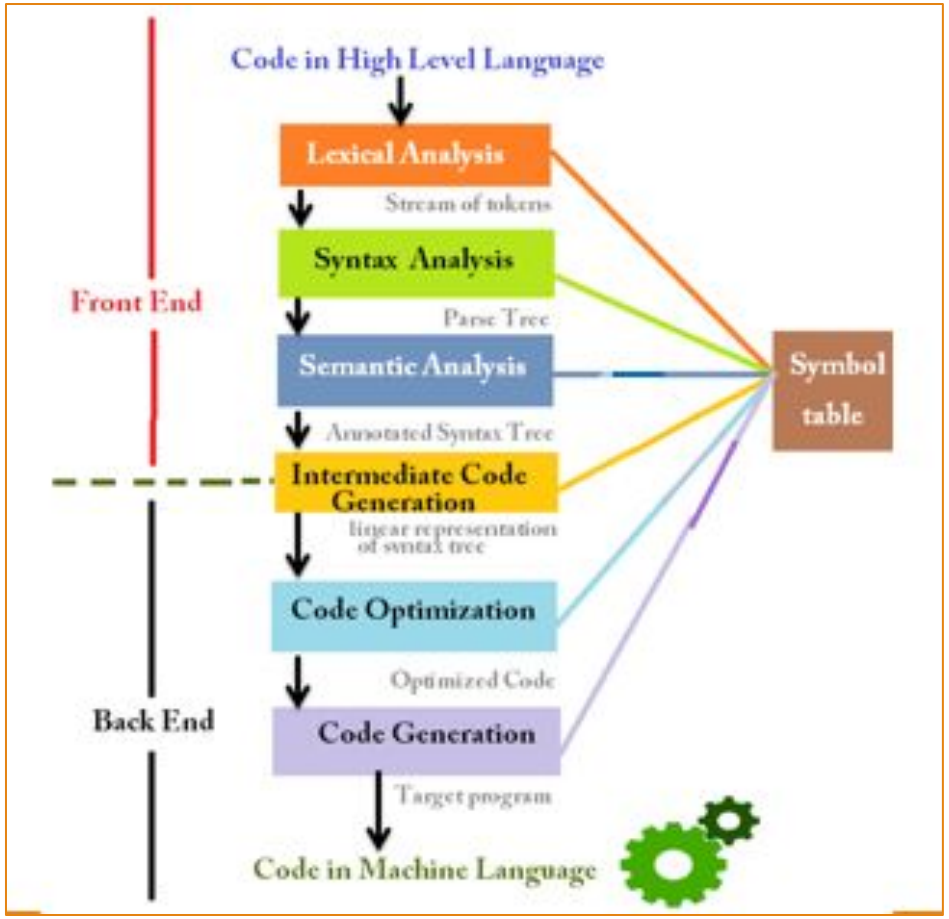
Run-Time Environment

Compiler design phases



Runtime Environment

- 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.

Activation Record

- A program is a **sequence of instructions combined into a number of procedures**.
- Instructions in a procedure are executed sequentially.
- A procedure has a start and an end delimiter and everything inside it is called the body of the procedure.
- The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.

- The execution of a procedure is called its **activation**.
- **An activation record** contains **all the necessary information** required to call a procedure.

Activation Record

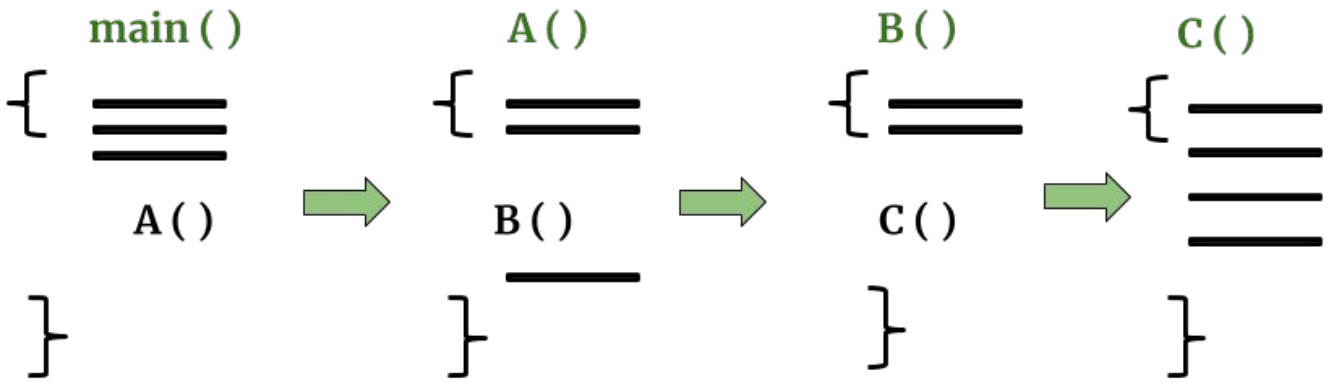
An activation record may contain the following units:
(depending upon the source language used)

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.
Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.
Return Value	Stores return values.

Activation Record

- Whenever a procedure is executed, its **activation record is stored on the stack**, also known as **control stack**
- When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the **activation record of the called procedure is stored on the stack**
- We assume that the **program control flows in a sequential manner** and when **a procedure is called, its control is transferred to the called procedure**
- When a called procedure is executed, it returns the control back to the caller.
- **A procedure is recursive if a new activation begins before an earlier activation of the same procedure has ended.**

Example of Activation Records:



Before going to A() activation record of main is created when A() is called activation record of A() is created.

Before going to B() activation record of main and A() in stack.

Before going to C() activation record of main and B() is pushed in stack.

Activation Trees

- An activation tree shows the **way control enters and leaves** activations.
- This **type of control flow makes it easier to represent a series of activations** in the form of a **tree**, known as the **activation tree**.

Properties of Activation trees are :-

✔ **Each node** represents an **activation of a procedure**.
✔ The **root** shows the activation of the **main function.**
✔ The node for procedure ‘x’ is the parent of node for procedure ‘y’ if and only if the control flows from procedure x to procedure y.

Activation Trees

Example – Consider the following program of Quicksort

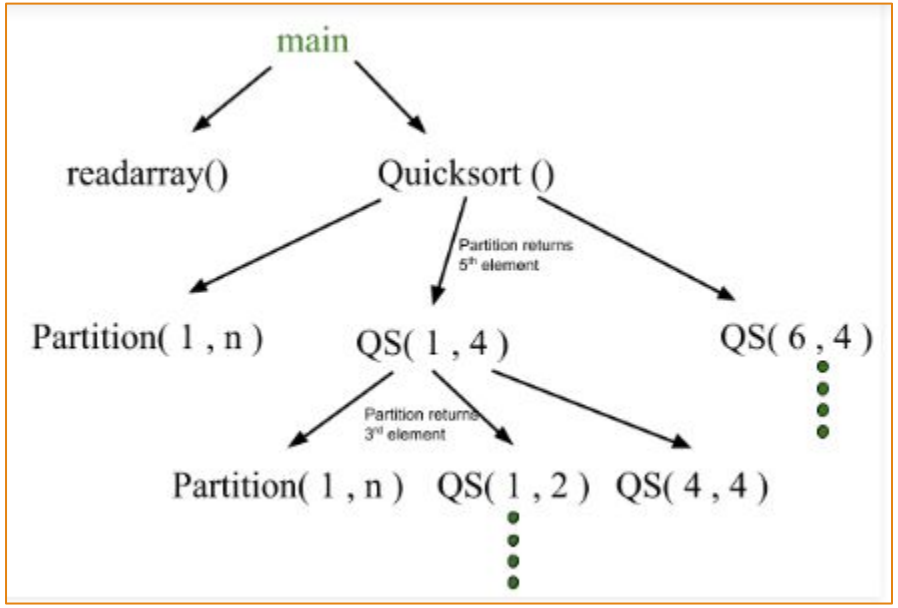
```
main() {

      Int n;
      readarray();
      quicksort(1,n);
}

quicksort(int m, int n) {

     Int i= partition(m,n);
     quicksort(m,i-1);
     quicksort(i+1,n);
}
```

The activation tree for this program will be:

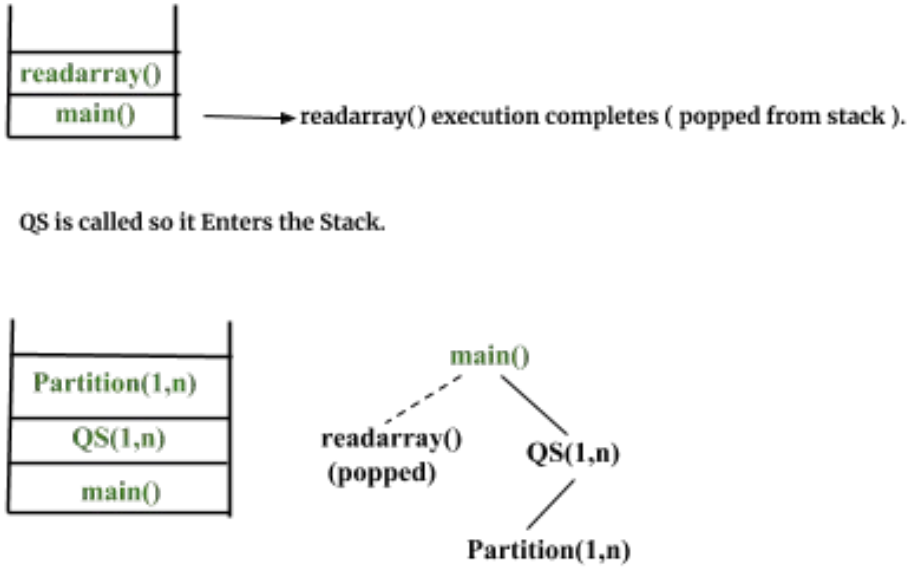


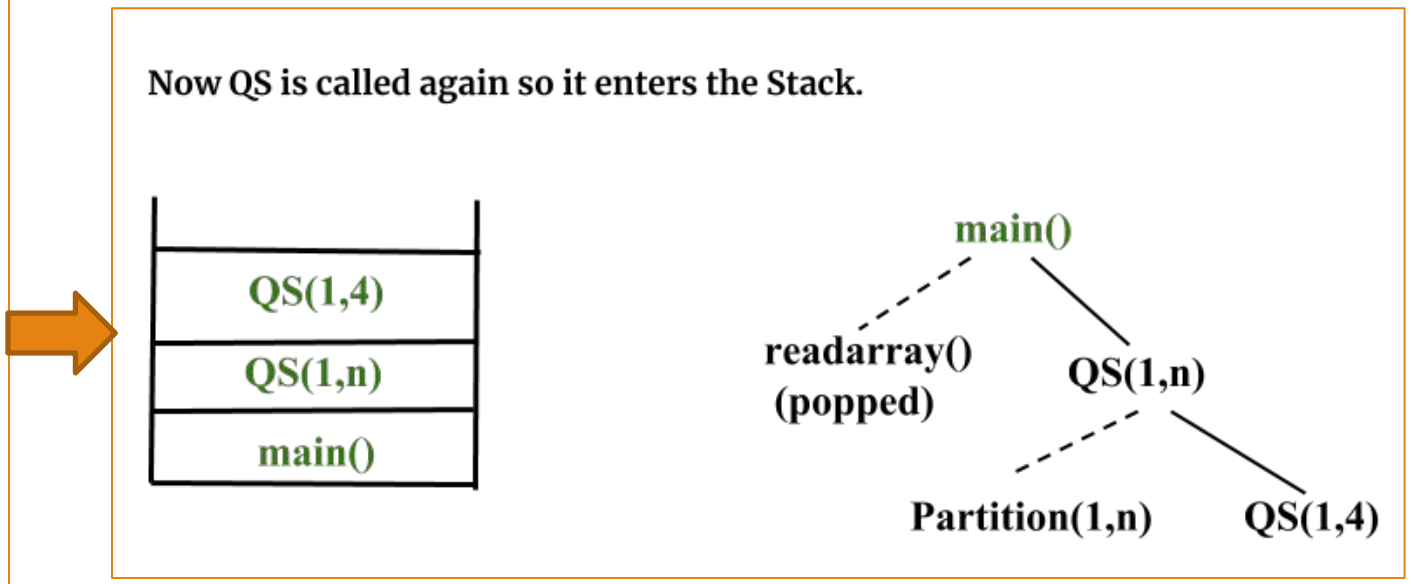
The flow of control in a program corresponds to the **depth first traversal of activation tree which starts at the root**.

CONTROL STACK AND ACTIVATION RECORDS

- **Control stack or runtime stack** is used to keep track of the live procedure activations i.e the procedures whose execution have not been completed.
- A procedure **name is pushed on to the stack when it is called** (activation begins) and **it is popped when it returns** (activation ends).
- Information needed by **a single execution of a procedure is managed using an activation record or frame.**
- When a procedure is called, **an activation record is pushed into the stack** and as soon as **the control returns to the caller function the activation record is popped.**

CONTROL STACK FOR THE QUICKSORT EXAMPLE:





SUBDIVISION OF RUNTIME MEMORY

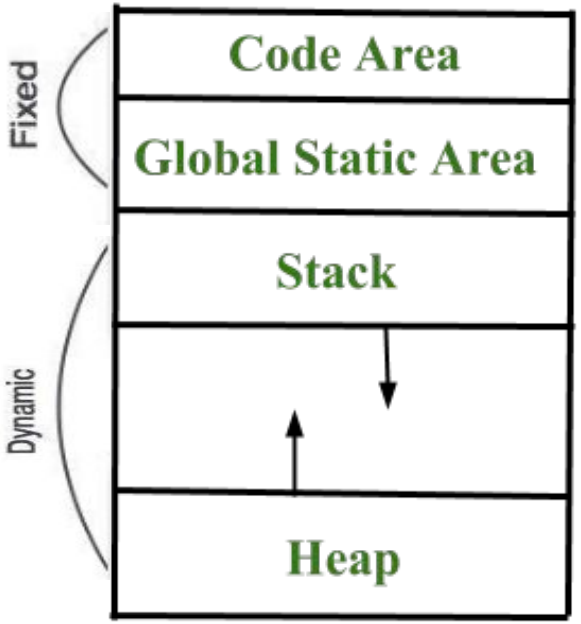
Runtime environment manages runtime memory requirements for the following entities:

Code : It is known as the text part of a program that **does not change at runtime.** Its memory requirements are **known at the compile time**.

Procedures : Their text part is **static but they are called in a random manner.** That is why, **stack storage** is used to manage procedure calls and activations.

Variables : Variables are known at the **runtime only, unless they are global or constant**. **Heap memory allocation** scheme is used for managing allocation and de-allocation of memory for **variables in runtime**.

SUBDIVISION OF RUNTIME MEMORY



The Stack grows towards Higher Memory.

Heap grows towards lower Memory.

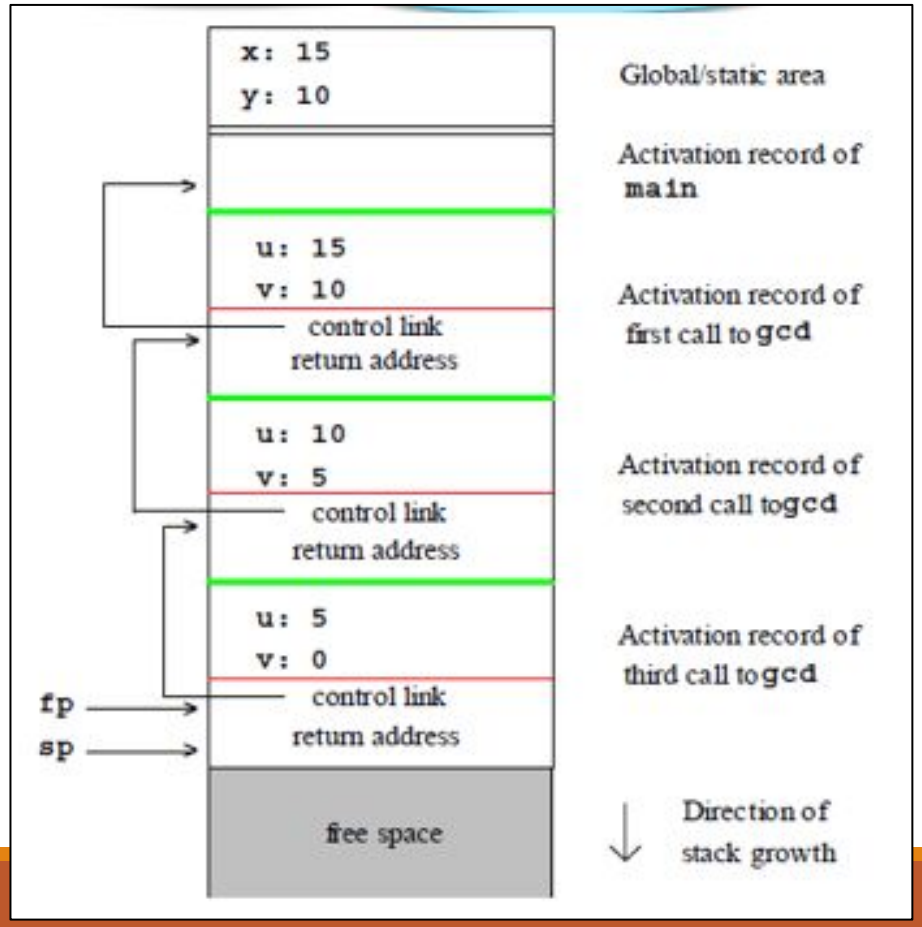
- ✔ In the image, the text part of the code is allocated a fixed amount of memory.
- ✔ Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both **shrink and grow against each other.**

EXAMPLE: SUBDIVISION OF RUNTIME MEMORY

Example Program in C

```
int x,y;
int gcd (int u, int v)
{
     if (v==0) return u;
     else return gcd(v,u%v);
}
main()
{
     scanf ("%d%d",&x,&y);
     printf ("%d\n", gcd(x,y));
return 0;
}
```

Call Structure on Input: 15,10



STORAGE ALLOCATION TECHNIQUES

I. Static Storage Allocation

- For any program **if we create memory at compile time, memory will be created in the static area and only once**
- It don't support dynamic data structure i.e memory is created at compile time and deallocated after program completion.
- **The drawback with static storage allocation is recursion is not supported.**
- **Another drawback is size of data should be known at compile time**

Eg- FORTRAN was designed to permit static storage allocation.

STORAGE ALLOCATION TECHNIQUES

II. Stack Storage Allocation

- **Storage is organized as a stack** and activation records are pushed and popped as activation begin and end respectively.
- Locals are contained in activation records so they are bound to fresh storage in each activation.
- **Recursion is supported in stack allocation**

STORAGE ALLOCATION TECHNIQUES

II. Heap Storage Allocation

- **Memory allocation and de-allocation can be done at any time** and at any place depending on the requirement of the user
- **Heap allocation is used to dynamically** allocate memory to the variables and claim it back when the variables are no more required.
- **Recursion is supported.**

Parameter Passing

- **The communication medium among procedures is known as parameter passing.**
- The values of the variables from a calling procedure are transferred to the called procedure by some mechanism

Basic terminology :

R- value: The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if its appear on **the right side of the assignment** operator. R-value can always be assigned to some other variable.

L-value: The location of the memory(address) where the expression is stored is known as the l-value of that expression. It always appears on the **left side if the assignment** operator.

Parameter Passing

For example:

```
day = 1;
week = day * 7;
month = 1;
year = month * 12;
```

From this example, we understand that constant values like 1, 7, 12, and variables like day, month all have r-values.

- **Only variables have l-values as they also represent the memory location assigned to them.**

For example: 7 = x + y;

The equation is an l-value error, as the constant 7 does not represent any memory location.

Parameters

i. Formal Parameter:

Variables that take the information passed by the caller procedure are called formal parameters. **These variables are declared in the definition of the called function.**

ii. Actual Parameter:

Variables whose values and functions are passed to the called function are called actual parameters. **These variables are specified in the function call as arguments.**

Example:

```
fun_one()
{
   int actual_parameter = 10;
   call fun_two(int actual_parameter);
}
   fun_two(int formal_parameter)
{
   print formal_parameter;
}
```

***Formal parameters hold the information of the actual parameter, depending upon the **parameter passing technique used. <u>It may be a value or an address.</u>**

Different ways of passing the parameters to the procedure

Pass by Value

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. **Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.**

Pass by Reference

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, **the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location.** Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value**.**

Different ways of passing the parameters to the procedure

Pass by Copy-restore

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.

Example:

```
int y;
calling_procedure()
{
   y = 10;
   copy_restore(y); //l-value of y is passed
   printf y; //prints 99
}
copy_restore(int x)
{
   x = 99; // y still has value 10 (unaffected)
   y = 0; // y is now 0
}
```

- When this function ends, the l-value of formal parameter x is copied to the actual parameter y.
- Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference.

Different ways of passing the parameters to the procedure

Pass by Name

- ✔Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language.
- ✔In pass by name mechanism, the name of the procedure being called is replaced by its actual body.
- ✔Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

Lecture Materials

✔https://www.tutorialspoint.com/compiler_design/compiler_design_runtime_environment.htm

✔https://www.geeksforgeeks.org/runtime-environments-in-compiler-design/?ref=lbp

✔**'Chapter 7:** Run-Time Environments' of the Text Book:

"Compilers: Principles , Techniques and Tools"

The End