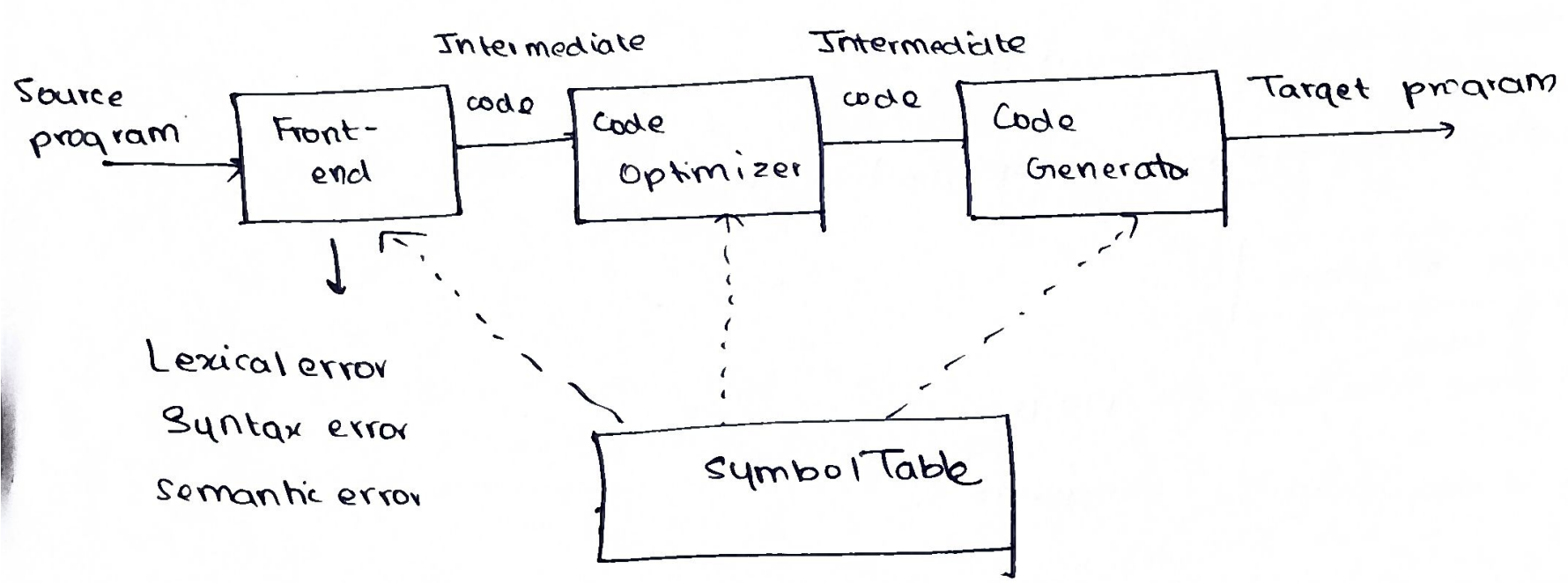
UCS702 COMPILER DESIGN

Unit - 4

Runtime Environments and Code Generation

Source Language Issues - Storage organization - Storage allocation strategies: Static, Stack and Heap - Implementation of symbol table - Issues in code generation - Design of a simple code generator

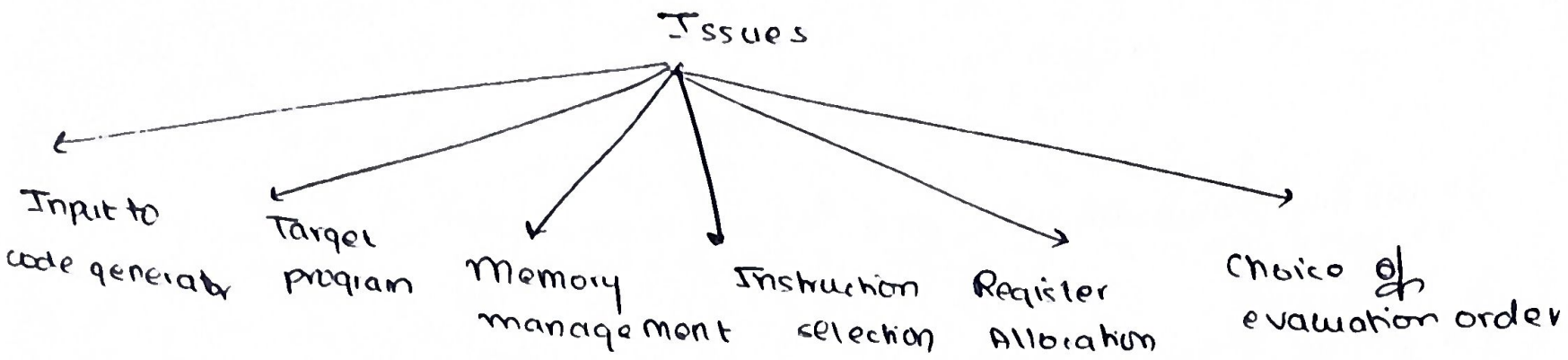
* Position of a Code Generator in the Compiler Model



* Key aspects of code generation

1. Code produced by the compiler must be correct - source to target program transforming should be semantics preserving
2. Code produced by the compiler should be of high quality.
3. Code generator must run efficiently.

* Issues in the Design of a Code Generator



A. Input to Code Generator

Input to the code generator in an intermediate may be of several forms :

(i) Linear representation - postfix notation

(ii) Three-address representation - quadruples

(iii) Virtual machine representation - stack machine code

(iv) Graphical representation - syntax tree, DAG

B. Target Program

can be

1. Absolute machine language
2. Relocatable machine language
3. Assembly language

C. Memory Management

→ Both the front end and code generator perform the task of mapping the names in the source program to addresses to the data objects in runtime memory.

D. Instruction Selection

→ Instruction selection is important to obtain efficient code

Suppose the TAC $x := 4+2$ is translated as:

```
mov 4, R0
ADD 2, R0
mov R0, x
```

→ In the same manner, if we translate $a := b+c$ and $d := a+e$

then

```
mov  c, R0
ADD  b, R0
mov  R0, a
mov  a, R0
ADD  e, R0
mov  R0, d
```

} These lines become redundant

→ Another example is for the TAC $a := a+1$

→ It can be written as

```
mov a, R0
ADD #1, R0
mov R0, a
```

or more efficiently

```
INC A
```

E. Register Allocation

→ Efficient utilization of the limited set of registers is important to generate good code.

→ Registers are assigned by:

(i) Register allocation to select the set of variables that will reside in registers at a point in the code

(ii) Register assignment to pick the specific register that a variable will reside in

→ Finding an optimal register assignment is NP-complete.

Consider the following TAC

```
t: = a * b
t: = t + a
t: = t / d
```

To make the runtime more efficient, use a single register R1.

```
MOV  a, R1
MUL  b, R1
ADD  a, R1
DIV  d, R1
mov  R1, t
```

(No other temp vars or registers are used)

F. Choice of Evaluation Order

→ When instructions are independent, their evaluation order can be changed.

→ This may lead to fewer instructions.

Consider the following example:

$a+b-(c+d)*e$ => $t_1 := a+b$ => mov a, R0
$t_2 := c+d$ ADD b, R0
$t_3 := e*t_2$ mov R0, t1
$t_4 := t_1 - t_3$ mov c, R1
ADD d, R1
mov e, R0
MUL R1, R0
mov t1, R1
SUB R0, R1
mov R1, t4

reorder independent instructions

$t_2 := c+d$ mov c, R0
$t_3 := e*t_2$ ADD d, R0
$t_1 := a+b$ ⇒ mov e, R1
$t_4 := t_1 - t_3$ MUL R0, R1
mov a, R0
ADD b, R0
SUB R1, R0
mov R0, t4

→ much shorter instruction set

* Target Machine

The key components involved in developing a target machine are:

- Nature of Target Program Code
- Architecture of Target Machine
- Op-codes and Address Modes
- Instruction Costs

A. Nature of Target Program Code

→ The back-end code generator of a compiler may generate different forms of code, based on the requirements:

- Absolute machine code (executable code)
- Relocatable machine code (object files for linker)
- Assembly language (facilitates debugging)
- Byte code forms for interpreters (JVM)

B. Architecture of the Target Machine

→ Implementing code generation requires thorough understanding of the target machine architecture and its instruction set

→ A hypothetical machine may have the following architecture:

- Byte-addressable (word = 4 bytes)
- Has n general purpose registers $R_0, R_1 \ldots R_{n-1}$
- Two address instructions of the form

op source, destination

C. Op-codes and Address Modes

→ Opcodes can be:

MOV (move content of source to dest)

ADD (add content of source to dest)

SUB (sub content of source from dest)

Address Modes

Mode	Form	Added Cost
Absolute	M	1
Register	R	0
Indexed	c(R)	1
Indirect Regista	*R	0
Indirect Indexed	*c(R)	1
Literal	# c	1

→ The cost of an instruction

= 1 + cost (source-mode) + cost (destination mode)

Examples - Calculate the total cost of each of the following set of instructions:

① mov R0, R1 → 0 + 0 + 1 = 1

mov R0, M → 0 + 1 + 1 = 2

mov M, R0 → 1 + 0 + 1 = 2

mov 4(R0), M → 1 + 1 + 1 = 3

mov *4(R0) M → 1 + 1 + 1 = 3

mov #1, R0 → 1 + 0 + 1 = 2

ADD 4 (R0), *12(R1) → 1 + 1 + 1 = 1

Total cost = 14

* Code Generation Steps

- Optimize use of registers using next-use information
- code generation algorithm using register and address descriptor

A. **Next-use Information**

→ When writing a program with some variables, some variables will be needed in future instructions, while others won't

→ Next use information helps free temporary allocations

i.e which variables should stay in register &
which variables can be removed from registers to make space for others

Key Information

① All temporaries are dead on exit

(i.e if you have $t_1, t_2 \dots t_n$ on the LHS, mark as dead)

② All user variables are live on exit.

(i.e, consider all variables on the RHS as live in that step of the iteration)

Algorithm
1. Analyze the code backward
2. Look at a statement $X := Y \text{ op } Z$

X is assigned a value (defined)
Y and Z are used (referenced)

3. Update the symbol table

X → mark as dead

Y & Z → mark as live, and record their next use

4. Repeat for all statements, moving backward.

(X) Example : Apply the next-use algorithm on the following code block.

1. $t_1 = a * a$
2. $t_2 = a * b$
3. $t_3 = 2 * t_2$
4. $t_4 = t_1 + t_3$
5. $t_5 = b * b$
6. $t_6 = t_4 + t_5$
7. $X = t_6$

Ans Step1 $X = t_6$

X = dead

t_6 = live

Variable	Liveliness	Next Use
t_6	~~dead~~ live	7

Step2 $t_6 = t_4 + t_5$

t_6 = dead

t_4, t_5 = live

Variable	liveliness	Next Use
t4	live	6
t5	live	6
t6	Dead	7

Step3 $t5 = b * b$

b = live

t5 = dead

Variable	liveliness	Next Use
t4	live	6
t5	Dead	6
t6	Dead	7

Step4 $t4 = t1 + t3$

t4 = dead

t1, t3 = live

next use for t1, t3 = Line 4

Variable	liveliness	Next Use
t1	live	4
t3	live	4
t4	Dead	6
t5	Dead	6
t6	Dead	7

Step5 $t_3 = 2 * t_2$

t_3 = dead

t_2 = live - next use = 3

Variable	Liveliness	Next Use
t_1	live	4
t_2	live	3
t_3	Dead	4
t_4	Dead	6
t_5	Dead	6
t_6	Dead	7

Step6 : $t_2 = a * b$

a, b = live

t_2 = dead

Variable	Liveliness	Next Use
t_1	Live	4
t_2	Dead	3
t_3	Dead	4
t_4	Dead	6
t_5	Dead	6
t_6	Dead	7

Step 7 : $t1 = a * a$

$t1 = dead$

Variable	liveliness	Next Use.
$t1$	Dead	4
$t2$	Dead	3
$t3$	Dead	4
$t4$	Dead	6
$t5$	Dead	6
$t6$	Dead	7

The original code can be rewritten as:

$t1 = a * a$

$t2 = a * b$

$t3 = 2 * t2$

$t4 = t1 + t3$

$t5 = b * b$

$t6 = t4 + t5$

$X = t6$

replace based on next use

$\Rightarrow$

$t1 = a * a$

$t2 = a * b$

$t2 = 2 * t2$

$t1 = t1 + t2$

$t2 = b * b$

$t1 = t1 + t2$

$X = t1$

* Code Generation Algorithm

→ The code-generation uses descriptors to keep track of register contents and addresses for names.

Register Descriptor

→ Keep track of what is currently in each register
→ Initially, all the registers are empty.

Address Descriptor

→ Keeps track of where the current value of a variable is stored
→ This can be a register, memory or both

Algorithm

For each X := Y op Z do

1. Invoke a function getreg to determine the location L where X is stored. Usually, L is a register.

2. Consult the address descriptor for Y. If Y is not in the desired register, move it to a register, using the following operation

   mov Y', L

   first move to a register

   generate OP Z, L

   perform the operation op Z

3. Update the register and address descriptors to reflect the current state after the operation

4. Optimize dead variables, If Y & Z are no longer needed free the registers holding them.

Function getreg

→ This function decides the register L where a value will be stored.

1. If y is in a register and is not live, use that register

2. Otherwise, find an empty register

3. If no empty register exists, spill (save) a value to memory and reuse the register

4. If all else fails, store the value directly in the memory

Example 1 : Generate TAC and target code for the following

Also compute the total cost.

$d := (a-b) + (a-c) + (a-c)$

This is rewritten in TAC as

$t_1 := a - b$

$t_2 := a - c$

$t_3 := t_1 + t_2$

$d := t_3 + t_2$

Statements	Code Generated	Reg Descriptor	Address Descriptor	Cost
$t_1 := a - b$	MOV a, R0 SUB b, R0	Initially empty R0 contains t_1	t_1 in R0	2 2
$t_2 := a - c$	MOV a, R1 SUB c, R1	R0 contains t_1 R1 contains t_2	t_1 in R0 t_2 in R1	2 2
$t_3 := t_1 + t_2$	ADD R1, R0	R0 contains t_3 R1 contains t_2	t_2 in R1 t_3 in R0	1
$d := t_3 + t_2$	ADD R1, R0 mov R0, d	R0 contains d R1 contains t_2	d in R0 & memory t_2 in R1	1 2
		Total Cost	12	

Example2 : Generate TAC and target code for the following. Also compute the total cost.

$d = -(a*b) + (c+d) - (a+b+c+d)$

Ans TAC

$t1 := a * b$

$t2 := -t1$

$t3 := c + d$

$t4 := a + b$

$t5 := t4 + c$

$t6 := t5 + d$

$t7 := t2 + t3$

$t8 := t7 - t6$

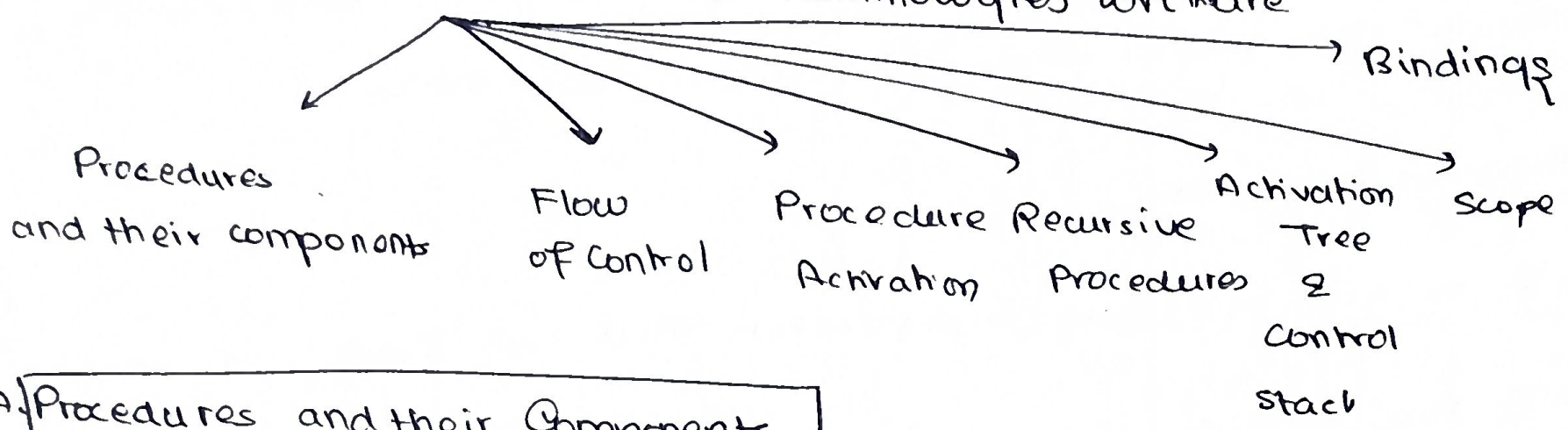
make sure not to get rid of registers being used again

Statement	Code Generated	Register descriptor	Address descriptor	Cost
$t1 = a * b$	MOV a, R0 MUL b, R0	R0 : t1	t1 : R0	4
$t2 = -t1$	NEG R0	R0 : t2	t2 : R0	1
$t3 = c + d$	MOV c, R1 ADD d, R1	R0 : t2 R1 : t3	t3 : R1 t2 : R0	4
$t4 = a + b$	MOV a, R2 ADD b, R2	R0 : t2 R2 : t4 R1 : t3	t4 : R2 t3 : R1 t2 : R0	4
$t5 := t4 + c$	ADD c, R2	R2 : t5 R1 : t3 R0 : t2	t5 : R2 t3 : R1 t2 : R0	2
$t6 : t5 + d$	ADD d, R2	R2 : t6 R1 : t3 R0 : t2	t3 : R1 t6 : R2 t2 : R0	2
$t7 = t2 + t3$	ADD R1, R0	R0 : t7 R1 : t3 R2 : t6	t6 : R2 t3 : R1 t7 : R0	1
$t8 : t7 - t6$	SUB R2, R0 MOV R0, t8	R0 : t8 R1 : t3 R2 : t6	t8 : R0 t3 : R1 t6 : R2	3
			Total	21

* Runtime Environment

→ A runtime environment must keep track of what functions are being executed, and which one to return to after execution finishes.

→ Some key components and terminologies we'll here are



A. Procedures and their Components

→ A procedure is a block of code that performs a specific task. It is called by its name in the program.

Components

1. Procedure Definition : where the procedure is written, specifying its name
2. Procedure call → Invokes the procedure to execute
3. Formal Parameters → Variables defined in the procedure's signature
4. Actual Parameters → Values passed to the procedure when called.

* Procedure Activation → An execution of a procedure body
* Lifetime of Procedure Activation → Sequence of steps between the first & last statements in the execution of the procedure body

including the time spent calling other procedures

* Flow of Control

→ Control flows sequentially

→ Execution of a procedure starts at the beginning of the procedure body, and eventually returns control to the point immediately following the place where the procedure was called.

* Recursive Procedures

→ A recursive procedure is a procedure that calls itself.

→ A new activation begins before an earlier activation of the same procedure has ended.

→ Several activations for the procedure p exist at time t.

* Activation Tree

→ An activation tree can be used to depict the way control enters and leaves activations.

Building an Activation Tree

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for a is the parent iff control flows from activation a to b
4. The node for a is to the left of the node for b iff the lifetime(a) occurs before lifetime(b)

→ The activation tree is traversed in a depth-first manner.

→ control stack

→ A stack is used to keep track of the live procedure activations.

1. Push a node for an activation onto the stack as the activation begins

2. Pop node for an activation off the stack when the activation ends

→ For a node n at the top of the stack, the stack contains the nodes along the path from n to the root.

Example: Consider the following procedure.

```
PROCEDURE quicksort(m,n):
    BEGIN
    if (n>m) BEGIN
        i = PARTITION (m,n);
        quicksort (m, i-1);
        quicksort (i+1,n);
    END
END;

/* main */
readarray;
quicksort(1,9);
```

Build an activation tree for this program

Use

s → main

r → readarray

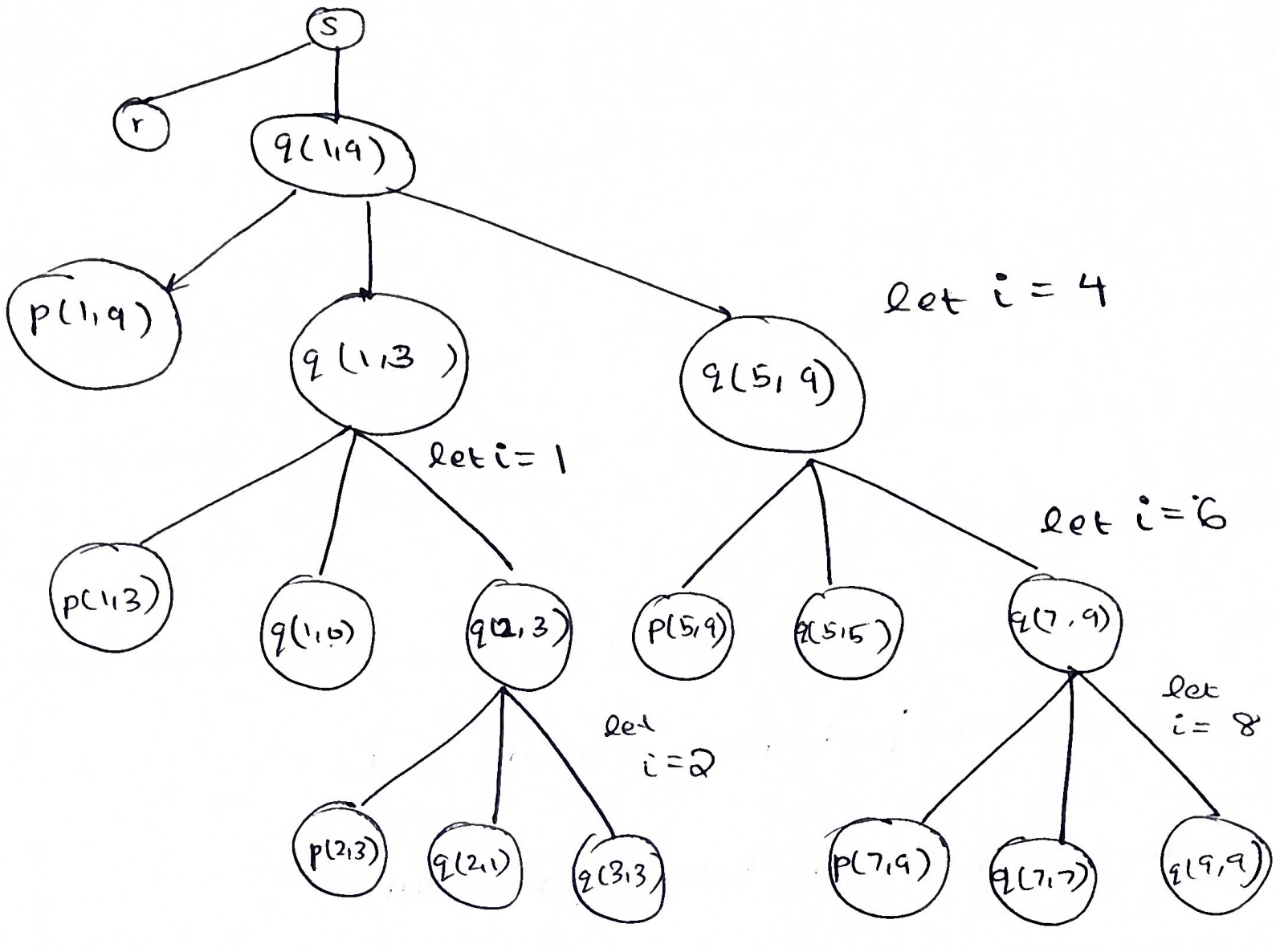
p → partition (x,y)

q → quicksort (x,y)

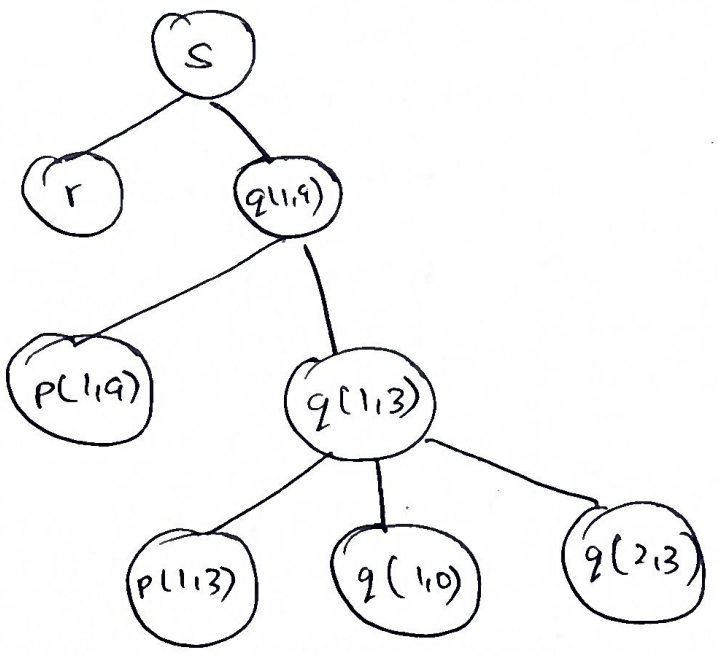
The initial pair is (1,9).

~~Assume that the pivot is~~

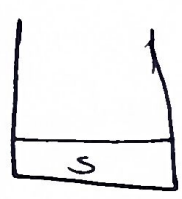
Ans



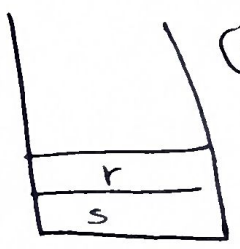
Example2 : Show the stack trace for the following activation tree



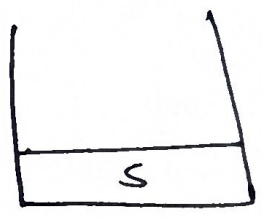
Ans (i) Activate s



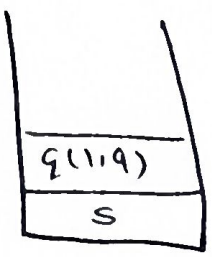
(ii) Activate r



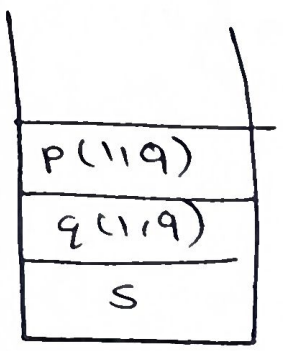
(iii) Execute & return



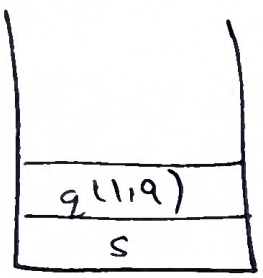
(iv) Activate q(1,9).



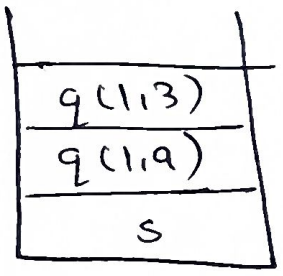
(v) Active p(1,9)



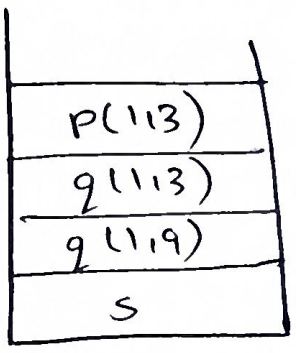
(vi) Execute & pop



(vii) Activate q(1,3)

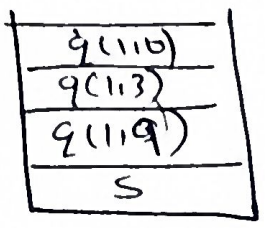


(viii) Activate p(1,3)

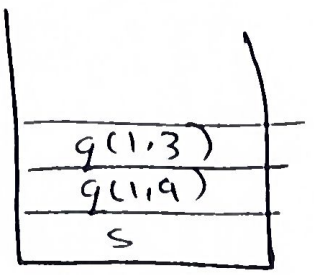


(ix) Execute and popout p(1,3)

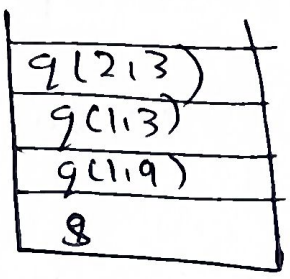
(x) Activate q(1,0)



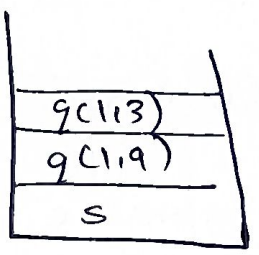
(xi) ~~Activate~~ Execute and popout



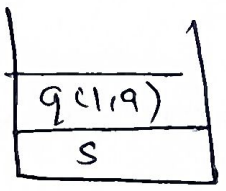
(xii) Activate q(2,3)



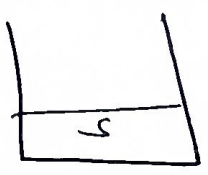
(xiii) Execute & popout



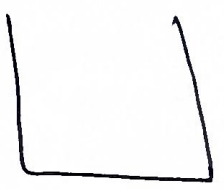
(xiv) Execute & popout

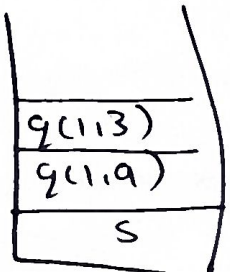


(xv) Exec & pop



(xvi) Exec & pop





* Variable Scope → can be local or global, even if the variable name is the same.

* Binaings of Names

① Environment → A function that maps a name to a storage location

$$f(\text{name}) \rightarrow \text{storage location}$$

② State → A mapping between storage locations and their values

$$g(\text{storage location}) \rightarrow \text{value}$$

→ To get the value of a variable, use the environment to map the variable name to its storage location and then use the state to find the value stored at that location

$$g(f(\text{name})) = \text{value}$$

For Example

1. Var x is mapped to storage location 100 by f
2. Storage location 100 holds the value 0 according to g

$$\therefore g(f(x)) = 0$$

3. If there is an assignment statement

$$x = 10$$

x is still bound to the storage location 100

i.e The environment does not change after an assignment, but the state does.

→ A binding is the dynamic counterpart of a declaration

→ When a variable is declared, it is bound to a storage location during execution.

Static Notion	Dynamic Counterpart
Definition of Procedure	Activations of the procedure
Declaration of name	Bindings of the name
Scope of Declaration	Lifetime of the binding

For example, in the case of Pascal local variables - a local variable in a procedure gets a new storage location for each procedure activation. This ensures that variables do not interfere between activations.

* Source Language Issues

→ Before generating machine code from your program, the compiler needs to handle the following issues.

(1) 1. Relationship between Name & Data Objects

→ Every variable, constant or function is associated with a name.

→ At runtime, these names need to be mapped to specific data stored in the memory

② Allocation and Deallocation

→ When variables are created, they are allocated memory

→ Once they are no longer needed, that memory can be released (deallocated)

→ The compiler adds routines managed by the runtime support package to manage this.

③ Handle Runtime Environment Types

→ Based on the language, the runtime environment can be:

1. **Fully Static**
   - memory fixed at compile time

2. **Stack-based**
   - use a stack to manage memory dynamically during function calls (C, Pascal)

3. **Dynamic**
   - use both a stack and a heap for flexible memory management (Python, Java)

* Storage Organization → Components of Runtime Storage

- Runtime Memory
- Activation Record
- Local Data

→ When a program is compiled, the operating system (OS) provides memory to execute it. This memory is organized into different areas.

(1) Runtime Storage components

a. Generated Target Code : The compiled instructions the computer executes

b. Data Objects : Variables and structures used during program execution

c. Control Stack : Tracks which function is currently running and allows the program to return to the correct place after completing a function.

(2) Runtime Memory

Memory Layout — The memory is divided into 4 main parts:

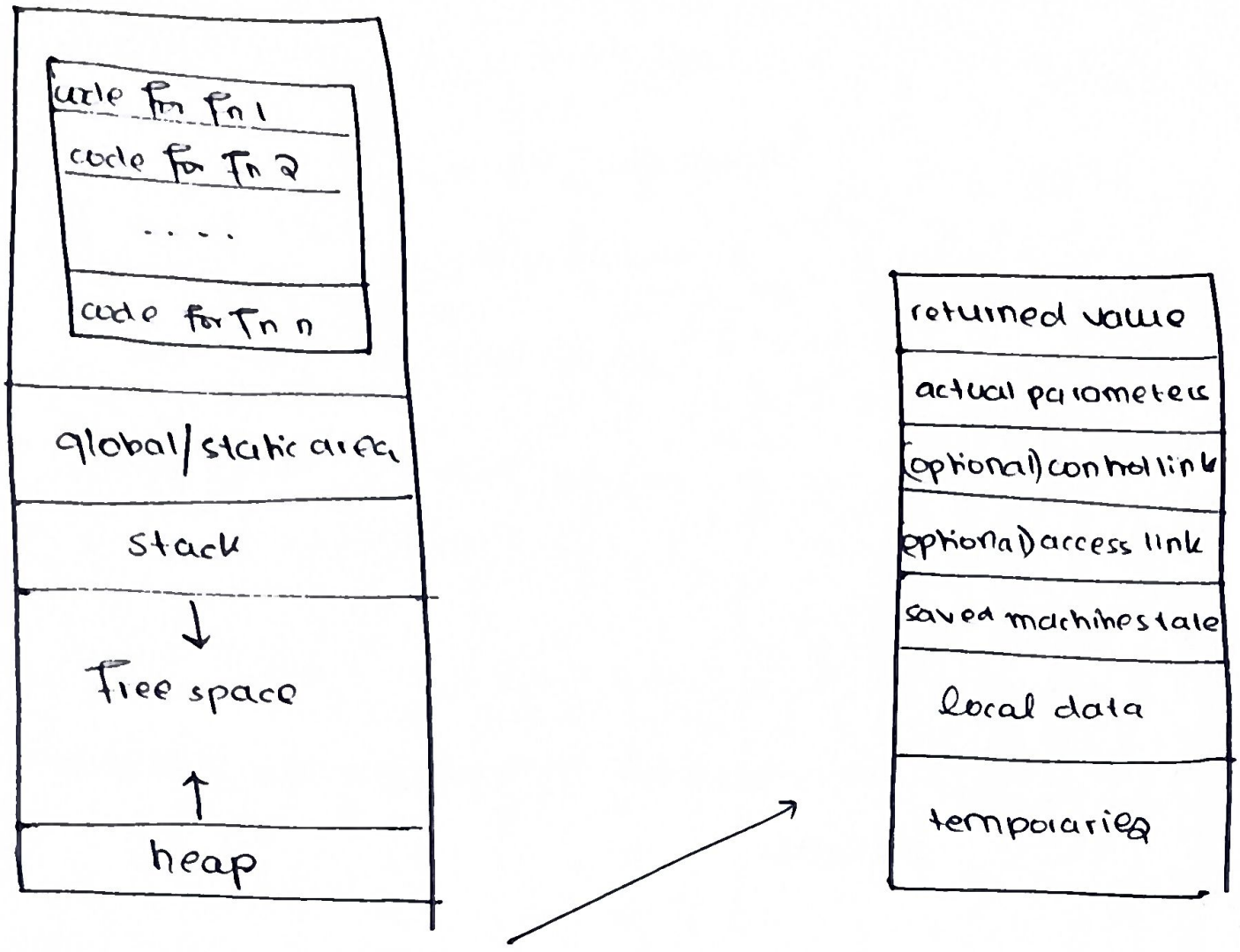
(a) Code - stores the program's instructions & functions

(b) Global / Static Area - Stores global and static variables, which remain throughout the program's execution

(c) Stack - Temporary memory for function calls and local variables

(d) Heap : memory for dynamically allocating objects (eg. w/ malloc)

Behavior of the Stack - The stack grows downward as new functions are called, and memory is allocated for them - once functions finish the memory is released.



(3.) Activation Record

→ When a function is called, its details (parameters, variables, etc.) are stored in an activation record, which is placed on the stack.
(also called frame)

components

a. ~~Temp~~ Returned value – stores result of function

b. actual parameters – input values passed to the function

c. control link – points to the caller's activation record

d. access link – helps access non-local variables

e. saved machine status – remembers the state of the system
(eg. program counter, registers)

f. local data – variables declared inside the function

g. Temporaries – temporary values generated during computation

④ Local Data

Storage - Memory for local variables is allocated during compile time

- Variable-length data is not stored here because it requires dynamic memory allocation

- Each variable gets an offset (relative pos. in memory) for efficient access

- Data that spans multiple bytes (multibyte objects) is stored sequentially in memory.

* Storage Allocation Strategies

There are 3 strategies
- Static allocation
- Stack-based allocation
- Dynamic (heap-based) allocation

① Static Allocation

Characteristics
- memory size is fixed at compile time
- no recursion is allowed since each function has a single activation record
- no dynamic memory allocation

Working → The compiler allocates memory for all variables and functions during compile time

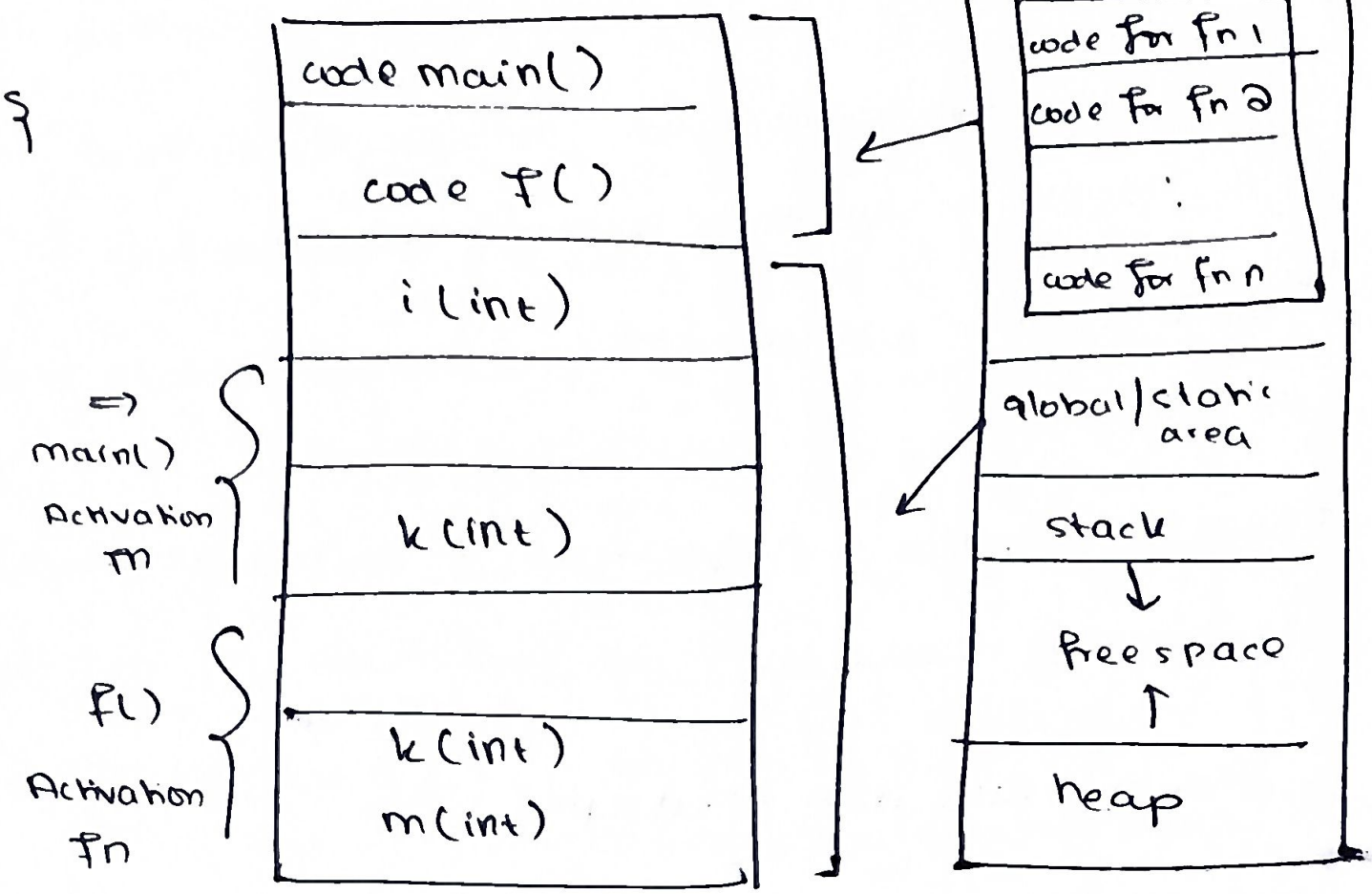
→ Bindings (associations between variables & memory locations) do not change at runtime

→ Every time a procedure is called, it uses the same pre-allocated memory

Example : Consider the following code block

```
int i=10;
int f(int i) {
    int k;
    int m;
    :
}

main() {
    int k;
    f(k);
}
```



② Stack-based Allocation

Characteristics - previous restrictions of static allocation are removed

- allows recursive function (by creating a new activation record for each call)

- allows dynamic memory allocation

Working - For each fn call - a new activation record is pushed onto the stack

- After execution, the activation record is popped from the stack

- The stack grows & shrinks dynamically

Example - same as qsort example - include runtime variables in stack too.

(3) Dynamic (Heap-Based) Allocation

→ also called a free-store

→ Memory is allocated at runtime from the heap, allowing flexibility for objects with unknown sizes at compile time

→ Requires manual deallocation / automatic garbage collection, making it slower but more flexible than stack-based allocation

→ Heap management is challenging

* Calling Sequences

→ callee outlives caller
→ can return pointers to local fns

→ A calling sequence manages how the caller (function making the call) interacts with the callee (function being called). It ensures

1. The caller's state is saved so it can resume after the callee finishes
2. Parameters and return values are passed efficiently.

A. Call Sequence

When a function is called:

The caller

1. Saves its state (program counter, registers etc.)
2. Allocates memory for the callee's activation record (stores params, return addresses, local vars)
3. Updates stack pointer (top-sp)

The callee

1. saves its own state
2. Initializes local data and starts executing

B Return Sequence

When the function completes:

The callee:

1. Stores the return value near the caller for easy access
2. Restores the saved state of the caller
3. Transfers control back to the caller

The caller:

1. reads the return value & resumes execution

* Offset Calculation

→ A special register (top-sp)

points to the boundary between caller & callee data.

→ The callee accessed local vars & temp data using this pointer.

Why parameters & return values are nearby?

→ keep close to caller

→ fast access w/ offsets

→ no need for caller to understand callee's internal details

* Dangling References

→ A dangling reference occurs when a program holds a reference (eg. ptr / var) to a memory location that has already been deallocated.

→ This leads to logical errors where the program attempts to access invalid or non-existent memory, causing unpredictable bugs.

How Dangling References Arise

→ A common scenario involves local variables in functions.

→ When a fn. is called, local vars are allocated in the fn's activation record (on the stack)

→ When the fn. returns, its activation record is popped from the stack, and the memory for those variables is deallocated.

→ If the fn. returns a reference to one of its local variables, this reference becomes dangling because the variable no longer exists

Example Consider the following program

```
int *danale() {
    int i=23;
    return &i;
}

int main() {
    int *p;
    p = danale();
}
```

→ When danale() returns, memory for i is deallocated

→ p points to an invalid memory location because i no longer exists

→ pointer p is a dangling reference, will lead to undefined behavior.