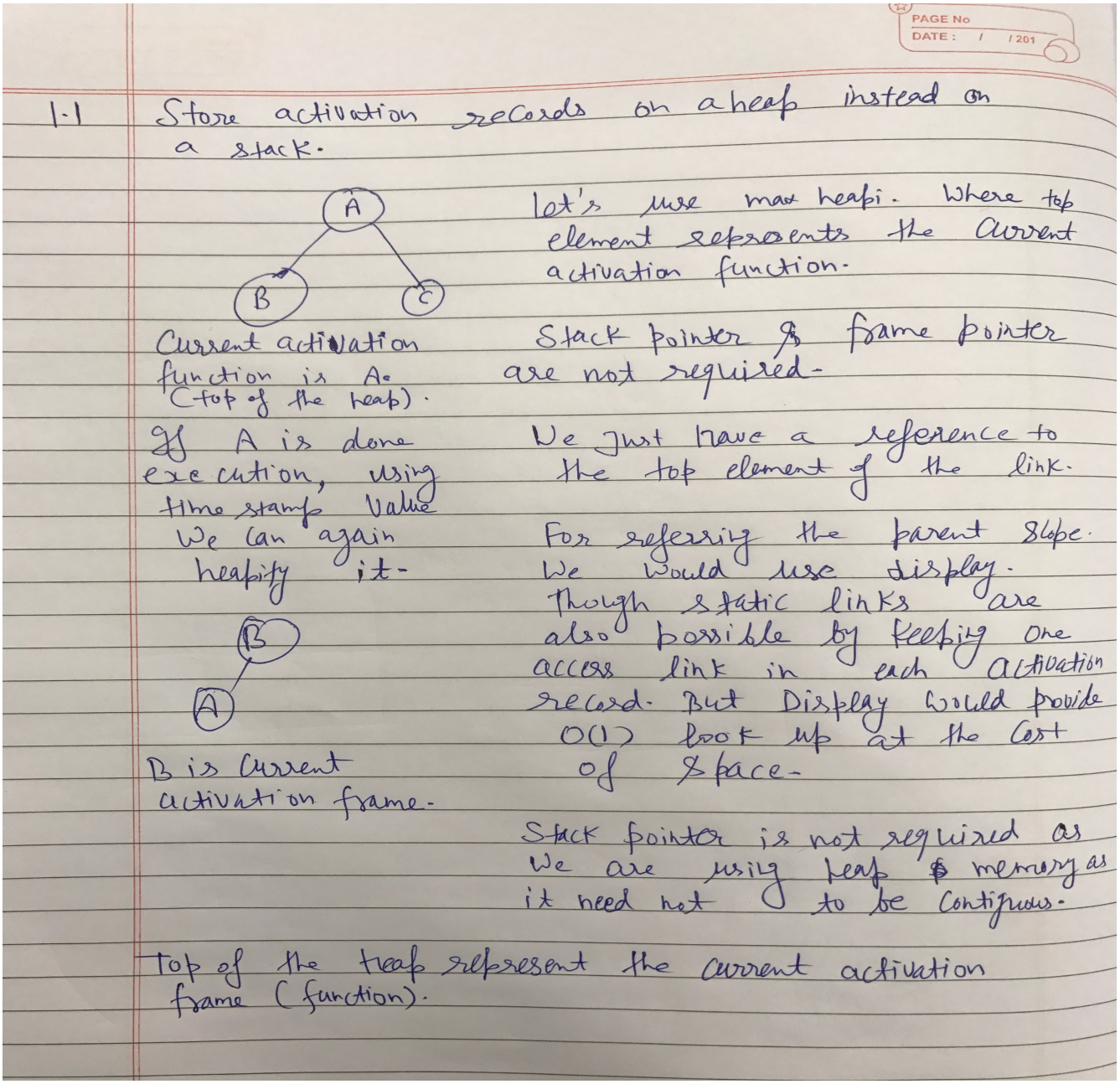
Programming Language, Assignment- 2

Tulsi Jain

1. Runtime Stack

1.1



Stack and pointer are not required because memory is in heap. But we are using the heap pointer, which points to the current activation, Top element. Heap is max heapify based of time-stamp of function calling. When a activation is complete it would take to time to max heapify the node. Node Structure is ActivationFrame, LeftChild, RightChild, Static_ointer

Second Approach for Part a. In this approach, Two pointer are used to traverse forward and

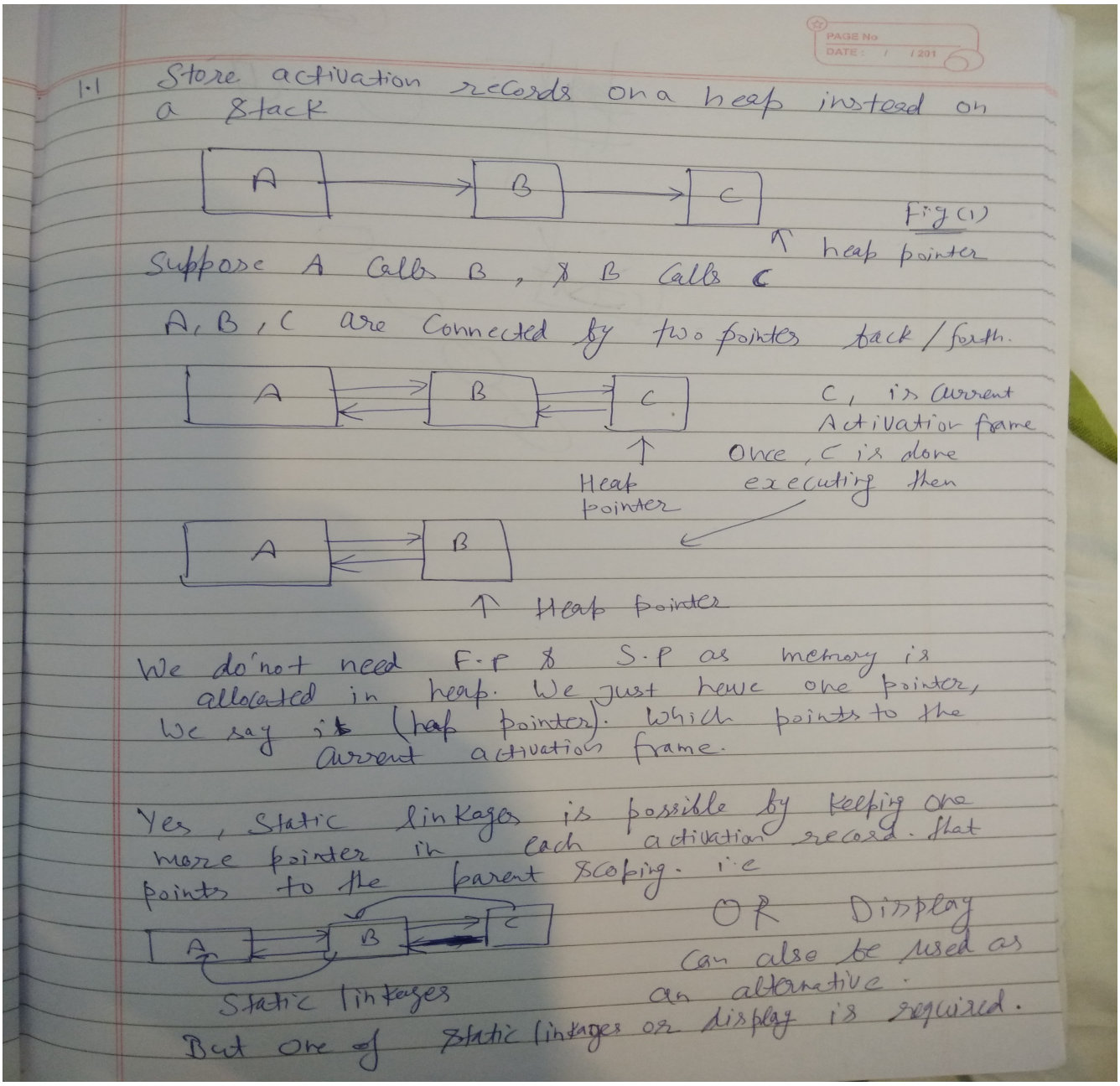
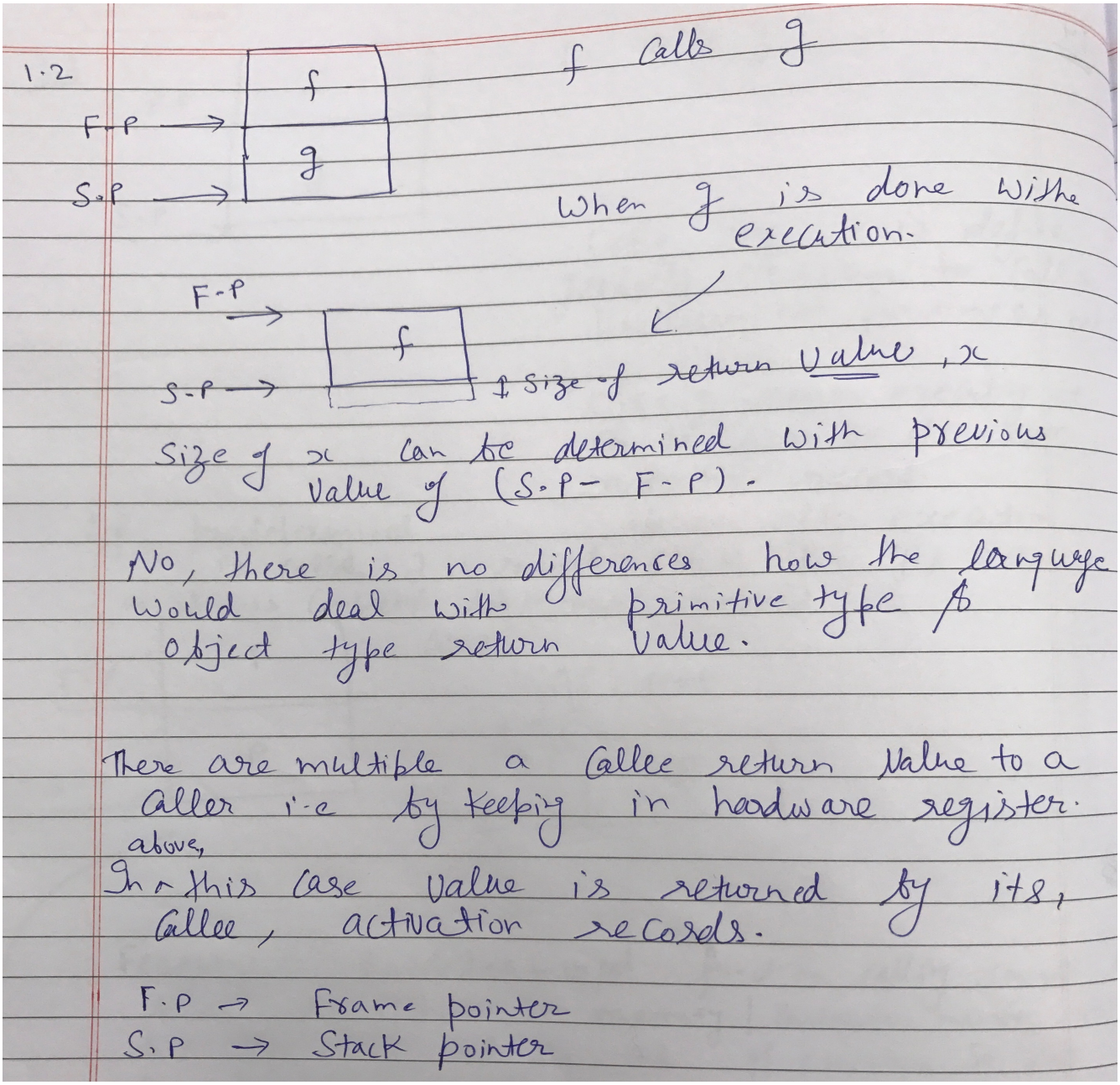


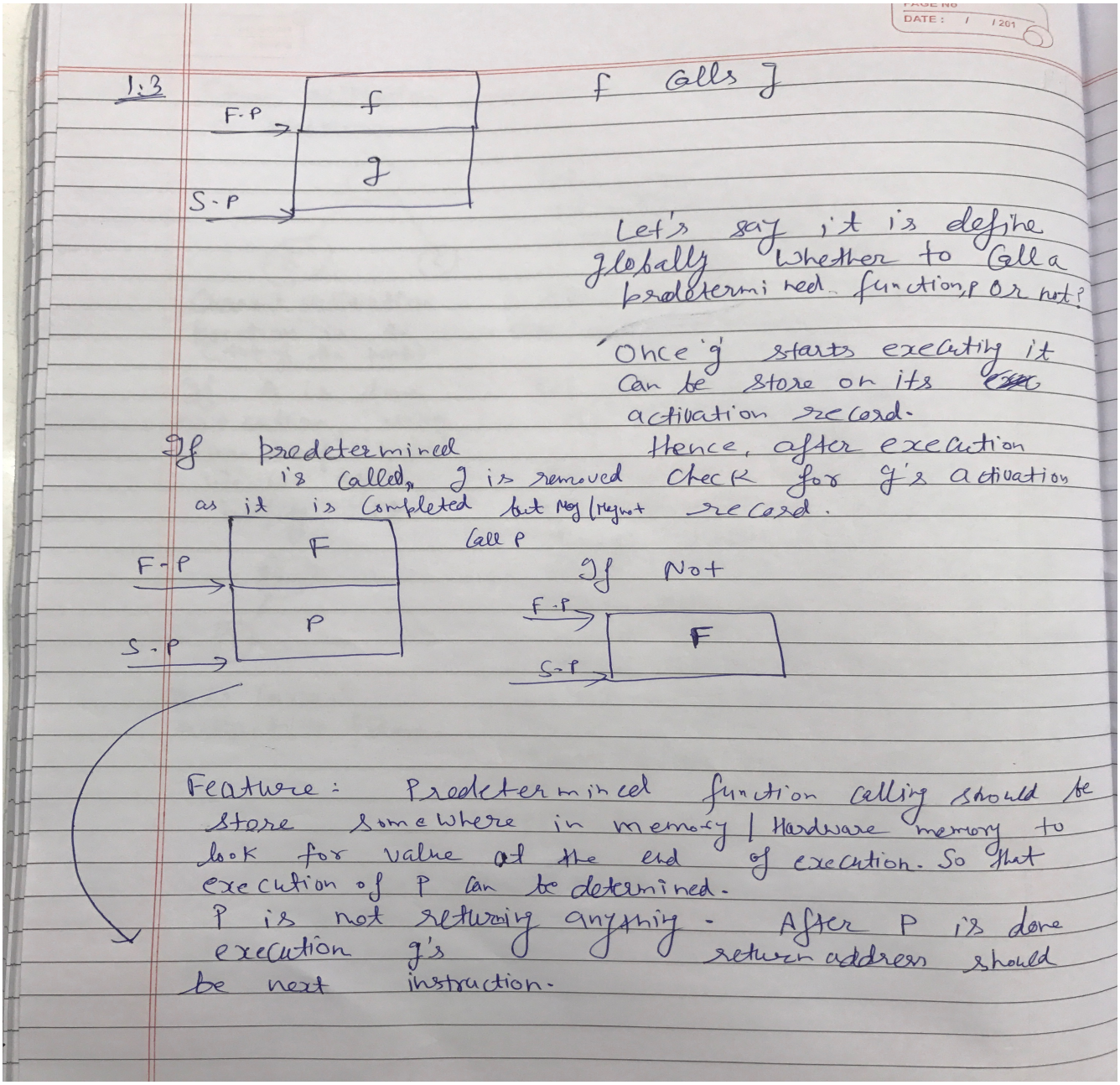
Figure 1.1: Second Approach

backward each. Activation removal time is $O(1)$. Linked-list node points to the current activation nodes. For static linkages every node has a pointer. Node Structure is ActivationFrame, Forward, Backward, StaticPointer

1.2

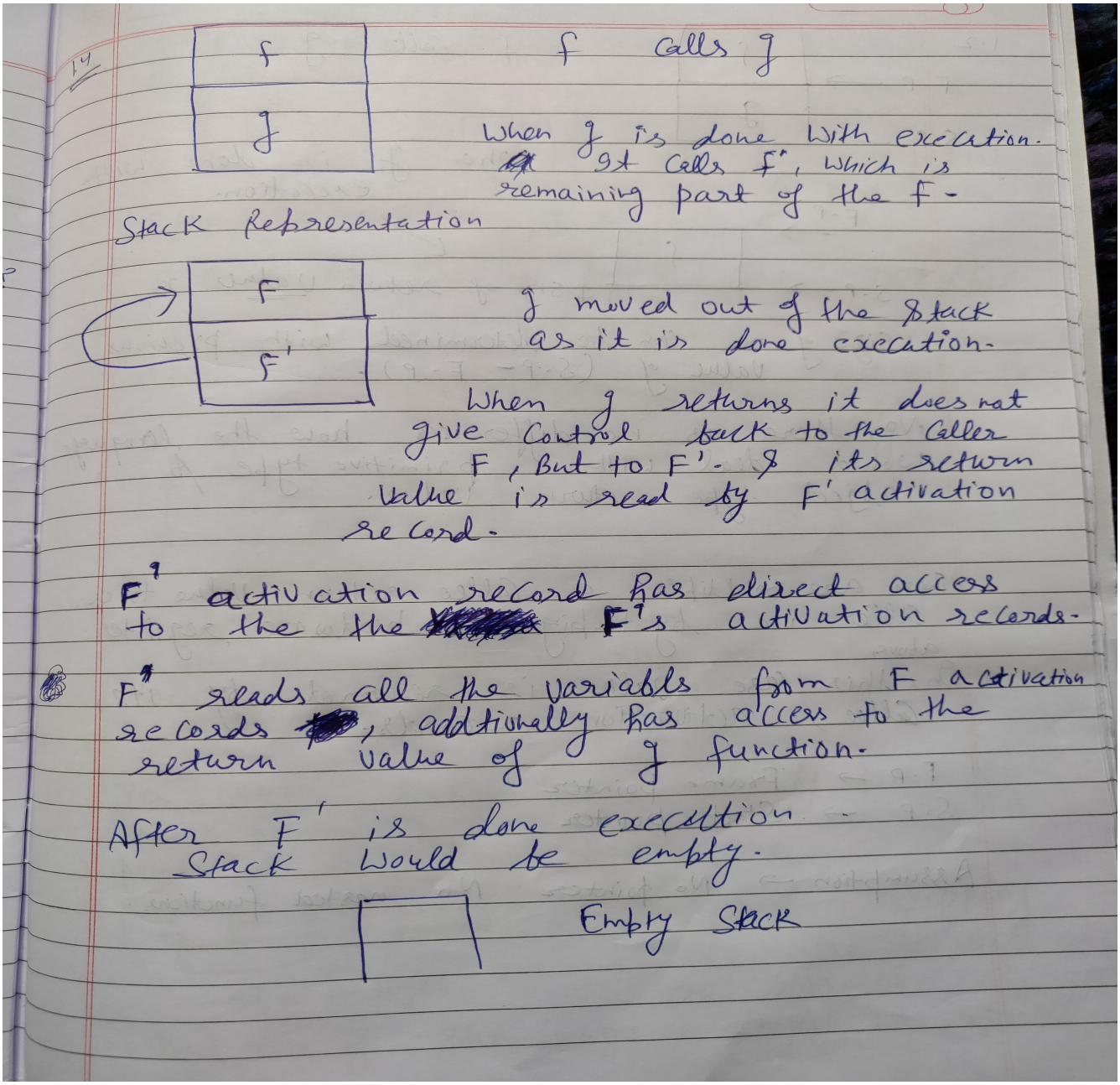

1.2
f
g
F.P
S.P
f calls g
When g is done with execution.
F.P
S.P
f
↕ Size of return Value, x
Size of x can be determined with previous Value of (S.P − F.P).
No, there is no differences how the language would deal with primitive type & object type return value.
There are multiple a Callee return Value to a Caller i.e by keeping in hardware register. above,
In this case Value is returned by its, Callee, activation records.
F.P → Frame pointer
S.P → Stack pointer

1.3



Annotation can be used as one of optimized way to determine whether predetermined function call should be made or not like @Override annotation in java. By default any value can be set let's say true and a programmer can change it accordingly.

1.4



f' has access to f activation record and g return value.

2. Nested Procedures

2.1

Outer → Inner → Innermost → Innermost → Innermost
The same Outer function would again call these in functions because for loop is running twice in mentioned order
→ Inner → Innermost → Innermost → Innermost

2.2

No, Order is same in this example because *inner* is getting out of scope when first iteration is complete. hence order is
Outer → Inner → Innermost → Innermost → Innermost
The same Outer function would again call these in functions because for loop is running twice in mentioned order
→ Inner → Innermost → Innermost → Innermost

2.3

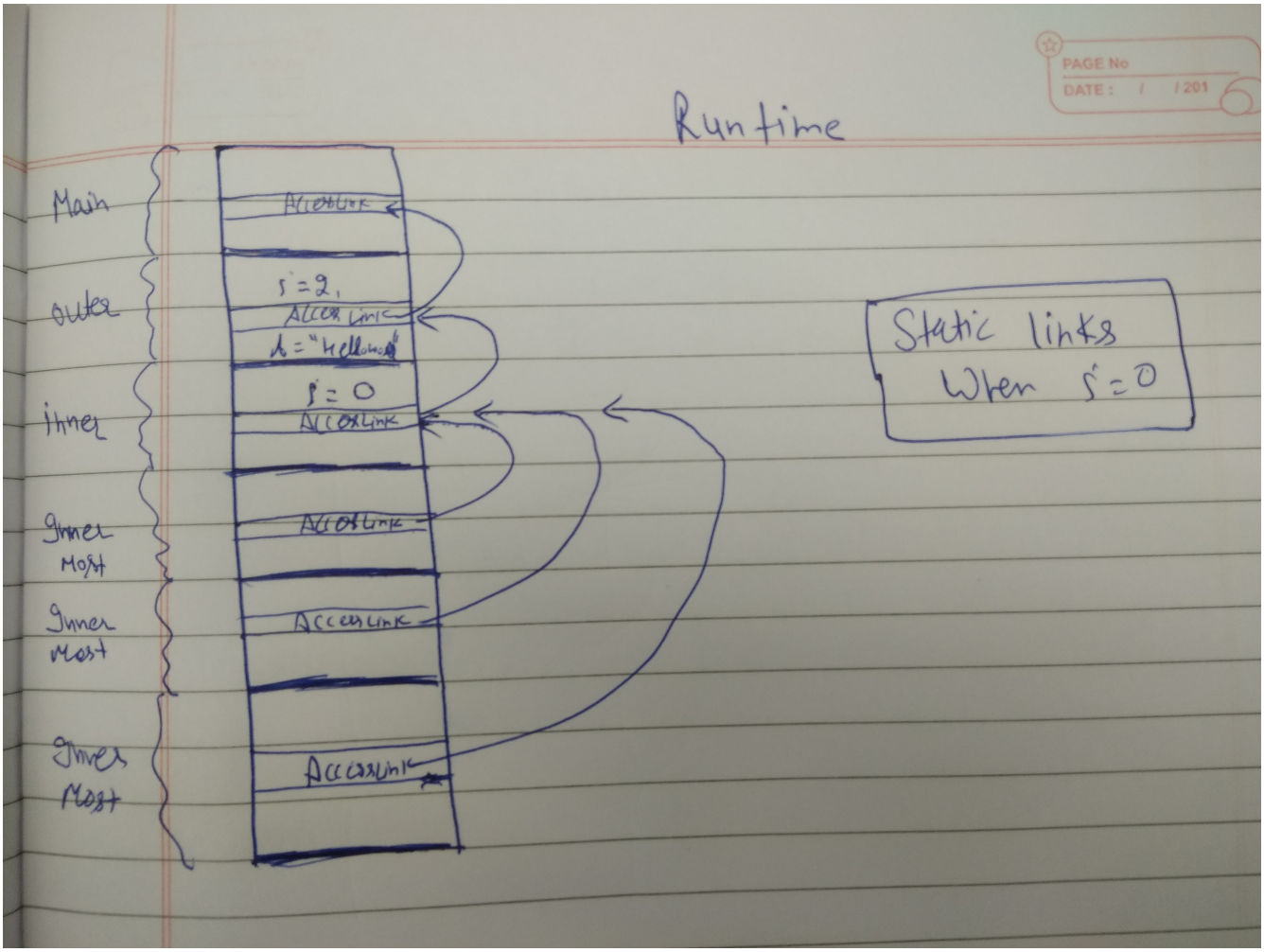


Figure 2.1: Run time stack

2.4

Outer can not call innermost function because it is not visible. Innermost can call Outer as because of its parent scope.
Reference: Chapter 9 - Programming Language Pragmatics FOURTH EDITION

3. Parameter Passing

3.1 Iteration

$a_1 = i, a_2 = \frac{i}{2}, a_3 = j$ (Every time)
In beginning, $a_1 = 0, a_2 = 0, a_3 = 0$ means $i = 0, j = 0$
After first iteration $a_1 = 2, a_2 = 0, a_3 = 0$ means $i = 2, j = 0$
After second iteration $a_1 = 4, a_2 = 1, a_3 = 1$ means $i = 4, j = 1$
After third iteration $a_1 = 6, a_2 = 2, a_3 = 3$ means $i = 6, j = 3$
After fourth iteration $a_1 = 8, a_2 = 3, a_3 = 6$ means $i = 8, j = 6$
After fivth iteration $a_1 = 10, a_2 = 4, a_3 = 10$ means $i = 10, j = 10$

3.2 Call-by-name

$a_1 = i, a_2 = \frac{i}{2}, a_3 = j$ (Every time)
In beginning, $a_1 = 0, a_2 = 0, a_3 = 0$ means $i = 0, j = 0$
After first iteration $a_2 = 2, a_2 = 0, a_3 = 0$ means $i = 2, j = 0$
After second iteration $a_1 = 4, a_2 = 1, a_3 = 1$ means $i = 4, j = 1$
After third iteration $a_1 = 6, a_2 = 2, a_3 = 3$ means $i = 6, j = 3$
After fourth iteration $a_1 = 8, a_2 = 3, a_3 = 6$ means $i = 8, j = 6$
After fivth iteration $a_1 = 10, a_2 = 4, a_3 = 10$ means $i = 10, j = 10$
hence print statement would be
the value of j is: 10

3.3 Call-by-need

$a_1 = i, a_2 = \frac{i}{2}, a_3 = j$ (Only first time)
In beginning, $a_1 = 0, a_2 = 0, a_3 = 0$ means $i = 0, j = 0$
After first iteration $a_1 = 2, a_2 = 0, a_3 = 0$ means $j = 0$
After second iteration $a_1 = 4, a_2 = 0, a_3 = 0$ means $j = 0$
After third iteration $a_1 = 6, a_2 = 0, a_3 = 0$ means $j = 0$
After fourth iteration $a_1 = 8, a_2 = 0, a_3 = 0$ means $j = 0$
After fivth iteration $a_1 = 10, a_2 = 0, a_3 = 0$ means $j = 0$
the value of j is: 0

3.4 Call-by-value

In beginning, $a_1 = 0$, $a_2 = 0$, $a_3 = 0$
After first iteration $a_1 = 2, a_2 = 0, a_3 = 0$
After second iteration $a_1 = 4, a_2 = 0, a_3 = 0$
After third iteration $a_1 = 6, a_2 = 0, a_3 = 0$
After fourth iteration $a_1 = 8, a_2 = 0, a_3 = 0$

After fivth iteration $a_1 = 10, a_2 = 0, a_3 = 0$
i and j are always 0
This is call be value it means value of j would not change hence print statement would be
the value of j is: 0

3.5 Call-by-reference for a_1 and a_3 and a_2 is by value

In beginning, $a_1 = 0, a_2 = 0, a_3 = 0$ means $i = 0, j = 0$
After first iteration $a_1 = 2, a_2 = 0, a_3 = 0$ means $i = 2, j = 0$
After first iteration $a_1 = 4, a_2 = 0, a_3 = 0$ means $i = 4, j = 0$
After third iteration $a_1 = 6, a_2 = 0, a_3 = 0$ means $i = 6, j = 0$
After fourth iteration $a_1 = 8, a_2 = 0, a_3 = 0$ means $i = 8, j = 0$
After fivth iteration $a_1 = 10, a_2 = 0, a_3 = 0$ means $i = 10, j = 0$
hence print statement would be
the value of j is: 0

4. Lambda Calculus

4.1

4.1.1

λ s.s z λq.s q
λ s.(s z (λq.s q))
z is free
q is bound
both **s** are bound

4.1.2

(λ s.s z)λ q.q λ w.w q z s
(λ s.s z)(λ q.(q (λ w.w q z s)))
Alpha Conversion(s $\longrightarrow$ y, λ s.s z)
(λ y.y z)(λ q.(q (λ w.(w q z s))
both **z** are free
s is free
y is bound
w is bound
both **q** are bound

4.1.3

(λs.s)(λq.q s)
(λs.s)(λq.q s)
Alpha conversion (s $\longrightarrow$ z, λs.s)
(λz.z)(λq.q s)
s is free
z is bound
q is bound

4.1.4

λz.((((λs.sq)(λq.qz))λz.(zz))
λz.((((λs.sq)(λq.qz))(λz.(zz)))
Alpha conversion (q $\longrightarrow$ y, λq.qz)
λz.((((λs.sq)(λy.yz))(λz.zz))
q is free
All three **z** are bound
y is bound
s is bound

4.2

4.2.1 $(\lambda z.z)(\lambda z.z\ z)(\lambda z.zq)$

$(\lambda z.z)(\lambda z.zz)(\lambda z.zq)$
Alpha Conversion $(z \rightarrow y, (\lambda z.z))$
$(\lambda y.y)(\lambda z.z\ z)(\lambda z.zq)$
$((\lambda y.y)(\lambda z.z\ z))(\lambda z.zq)$
Beta Reduction $[y \hookrightarrow \lambda z.zz]$ $(\lambda y.\mathbf{y})$
$(\lambda z.zz)(\lambda z.zq)$
Alpha Conversion $(z \rightarrow y, (\lambda z.zz))$
$(\lambda y.yy)(\lambda z.zq)$
Beta Reduction $[y \hookrightarrow \lambda z.zq]$ $(\lambda y.\mathbf{y}\ \mathbf{y})$
$(\lambda z.zq)\ (\lambda z.zq)$
Alpha Conversion $(z \rightarrow y, (\lambda z.zq)\ )$
$(\lambda y.yq)\ (\lambda z.zq)$
Beta Reduction $[y \hookrightarrow \lambda z.zq]$ $(\lambda y.\mathbf{y}q)$
$\lambda z.zqq$
qq

4.2.2 $(\lambda s.\lambda q.sqq)(\lambda a.a)\ b$

$(\lambda s.\lambda q.sqq)(\lambda a.a)\ b$
Beta Reduction $[s \hookrightarrow \lambda a.a]$ $(\lambda \mathbf{s}.\lambda q.\mathbf{s}\ qq)$
$(\lambda q.(\lambda a.a)\ qq)\ b$
Beta Reduction $[q \hookrightarrow b]$ $(\lambda q.(\lambda a.a)\ \mathbf{qq})$
$(\lambda a.a)\ bb$
Beta Reduction $[a \hookrightarrow bb](\lambda a.\mathbf{a})$
bb

4.2.3 $(\lambda s.\lambda q.s\ q\ q)(\lambda q.q)\ q$

Alpha Conversion $[q \hookrightarrow y, (\lambda q.\mathbf{q})\ ]$
$(\lambda s.\lambda q.sqq)(\lambda y.y)\ q$
Beta Reduction $[s \hookrightarrow (\lambda y.y)](\lambda s.\lambda q.\mathbf{s}qq)$
$(\lambda q.((\lambda y.y)qq))\ q$
Beta Reduction $[y \hookrightarrow qq](\lambda y.\mathbf{y})$
$(\lambda q.(qq))\ q$
Alpha Conversion $[q \hookrightarrow y, (\lambda q.(\mathbf{qq}))\ ]$
$(\lambda y.(yy))\ q$
Beta Reduction $[y \hookrightarrow q](\lambda y.(\mathbf{y}\ \mathbf{y}))$
qq

4.2.4 ((λs.s s)(λq.q))(λq.q)

((λs.s s)(λq.q))(λq.q)
Beta Reduction [s ↪ (λq.q)] (λs.**ss**)
((λq.q)(λq.q))(λq.q)
Alpha Conversion [q ↪ y, (λq.**q**)]
((λy.y)(λq.q))(λq.q)
Beta Reduction [y ↪ (λq.q)] (λy.**y**))
(λq.q)(λq.q)
Alpha Conversion [q ↪ y, (λq.**q**)]
(λy.y)(λq.q)
Beta Reduction [y ↪ (λq.q)] (λy.**y**)
λq.q

4.2.5 Reduce PLUS [0] [2]

Reduce PLUS [0] [2]
λmnfx.mf(nfx) (λfx.x) (λfx.f(fx))
Beta Reduction λnfx.(λfx.x)f(nfx)λfx.f(fx)
Beta Reduction λnfx.(λx.x)(nfx)λfx.f(fx)
Beta Reduction λnfx.(nfx)λfx.f(fx)
Beta Reduction λfx.(λfx.f(fx))fx
Beta Reduction λfx.(λx.f(fx))x
Beta Reduction λfx.(f(fx))
≡ [2]

4.2.6 Reduce SUCC [1]

Reduce SUCC [1]
λnfx.f(nfx)λfx.fx
Beta Reduction λfx.f((λfx.fx) fx)
Beta Reduction λfx.f(λx.fx x)
Beta Reduction λfx.f(fx)
≡ [2]

5. Scheme

All the coding answers and comments are in .rkt file.