1/

```
def visit Dowhile (self,ctx,o):
    o.frame.enterLoop()
    breakLabel = o.frame.getBreakLabel()
    contLabel = o.frame.getContLabel()
    startLabel = o.frame.getStartLabel()
    self.emit.printout(self.emit.emitLABEL(startLabel))
    listStatements = []
    for statements in ctx.body:
        listStatements += statements.accept(self,o)

    op = list(map(lambda statements: statements.accept(self,o),ctx.body))

    self.emit.printout(self.emit.emitLABEL(contLabel))
    condCode, condType = ctx.expr.accept(self,Access(o.frame,o.sym,False)

    self.emit.printout(condCode)
    self.emit.printout(self.emit.emitIFTRUE(startLabel,o.frame))
    self.emit.printout(self.emit.emitLABEL(breakLable))
    o.frame.exitLoop()

def visitContinue(self,ctx,o):
    self.emit.printout(self.emit.emit.GOTO(o.frame.getContLabel),o.frame))
```

Explanation: the frame in o.frame passed to a continue node in a AST deep inside a Dowhile statements keep the information about the continue label and break label of the current loop (the Dowhile loop), so when we enterLoop(), these two label is saved on the o.frame and passed down into the Dowhile’s body statements until it visit an Continue node. Assuming there’s no other inner loop get in the way, when visitContinue use the frame method getContLabel(), the continue label create by the original Dowhile is return, allowing the generation of GOTO code to jump to that label. In our visitDowhile, we mark contLabel at the point just before we evaluate expr so that a conitnue statements will jump to exactly before the Dowhile’s condition execute

2/

```
def visitBinaryOp(self,ctx,o):
    if ctx.op == “+” or ctx.op == “-”:
        lhsCode, lhsType = ctx.left.accept(self,o)
        rhsCode, rhsType = ctx.right.accept(self,o)
        if type(lhsType) is IntType and type(rhsType) is IntType:
            return lhsCode + rhsCode + self.emit.emitADDOP(ctx.o, IntType(), o.frame),IntType()
        if type(lhsType) is IntType:
            lhsCode += self.emit.emitI2F(o.frame)
        if type (rhsType) is IntType:
            rhsCode += self.emit.emitI2F(o.frame)
        return lhsCode + rhsCode + self.emit.emitADDOP(ctx.o, FloatType(), o.frame),FloatType()
    else:
        jCode = “”
        labelF =o.frame.getNewLabel()
        labelO = o.frame.getNewLabel()
        jCode = ctx.left.accept(self,o)[0]
        jCode += self.emit.emitIFFALSE (labelF,o.frame)
        jCode += ctx.right.accept(self,o)[0]
        jCode += self.emit.emitIFFALSE(labelF, o.frame)
        jCode += self.emit.emitPUSHICONST(1, o.frame)
        jCode += self.emit.emitGOTO(labelO, o.frame)
        jCode += self.emit.emitLABEL(labelF, o.frame)
        jCode += self.emit.emitPUSHICONST(0, o.frame)
        jCode += self.emit.emitLABEL(labelO, o.frame)
        return jCode, BoolType()
```

3/

Pointer Type:
Refer to an address in memory:
Ex:
```
        int x;
        int *ptrx = &x;
```

+ Can be null
+Don’t need to be initialized

+The address that the pointer point to CAN BE CHANGE
+ Need to dereference to access the object

Reference Type
Refer to an existing object or value;
Ex:
```
int x;
int    &refx = x;
```

+ Cannot be null
+ Must be initialized (or in case of reference type parameter passed must not be optional)
+ Cannot be change to refer to other object

+Don’t need dereference to access the object

Alias: Pointer and Reference type can cause alias because they allow an object to be able to be bound to a different name. In our example, object X is bound to other names: *ptrx(pointer type) and refx(reference type)

4/
Basic features of Simple Call Return are:
+ No recursive call
+ Explicit Call site
+ Single Entry Point
+ Immediate Control Passing
+ Single Execution

Difference:

+ Recursive Call:	Allow recursion both directly and in directly
+ Explicit Handler:	May have no Explicit Call Site
+ Coroutine:	The subprogram execution can have Multi Entry Point
+ Tasks:	The subprogram is able to execute concurrently with other tasks on multiple processor or using time sharing on single processor
+ Schedule:	The execute is not immediately, but is schedule to run by specific time or priority, control by a scheduler

5/
Tombstone: Every time an object is allocated on the heap, the address of that object is also allocated into a extra memory, region called the “tombstone”. (The Tombstone is also allocated when a stack-referencing pointer is allocated. ie. When the “&” operator is used). The pointer to another object on the heap in this case will store the address of the tombstone that store the address of that object. Deallocation or pointer to stack reference become invalid, due to popped activation record the Tombstone is set to be null (or a special value) to indicate the data is dead. When dereferencing the pointer, it first access the Tombstone to retrieve the address of the actual object. If the

Tombstone store a value is null (or a special value), the program will detect that this is a illegal operation due to dangling reference.

Lock-and-key: Every time an object is allocated on the heap, aside from its data, another random integer value called (lock) also stored with that object. Any pointer to this object is a pair composed of the address of the object and the "key" storing the corresponding value of the object "lock". Every access to the pointer, first it check if the "lock" and "key" are matched before allowing the value of the object pointed to our pointer to be accessed. Every time a object is deallocated, the value of its "lock" is set to an given value (for example zero) to indicate the object is dead. So, if we tried to access an dead object through the dangling pointer. The machine will detect the "lock" and "key" are no longer matched.

6/
```
H(x,f,h) {
        if (f(x)) return h(h(x));
        else return f(x);
}
```

- H is a function with 3 parameter so its type is T_1 x T_2 x T_3 -> T_0 (1a)

Where: + T_1 is the type of x (1b)
+ T_2 is the type of f (1c)
+ T_3 is the type of h(1d)

- In the body, from the condition expression of the If statements if(f(x)) we have the type of f(x) is T_4 -> T_5 (2a); x is passed to the function f's call, so from 1b we have $T_4 = T_1$ (2b); also the value of its expression must be boolean type so T_5 = boolean (2c)

- From the expression h(x), h is a function with 1 parameter so the type of its function h is: T_6 -> T_7 (3a), x is passed to the function h, so from (1b) we have $T_1 = T_6$ (3b)

- From the expression h(h(x)), the return value is passed into h again so both its return type and parameter must be the same, and from (3a) we have $T_6 = T_7$ (4)

- From the first return statement, we see that the return value of h is also the return of the function H, so from (1a) and (3a) we have $T_0 = T_7$ (5)

- In the second return statement of the else branch, the return of function f is also used for the next return value of H, so from (1a) and (2a) we have $T_0 = T_5$ (6)

- From (1a), (2a), (3a) the type of H is: T_1 X (T_4->T_5) X (T_6->T_7) -> T_0 (7)
- From (2b),(2c),(3b),(4),(5),(6) we have: $T_1 = T_4 = T_5 = T_6 = T_7 = T_0$ = boolean (8)
- From (7) and (8), we have the type of function H is: bool x (bool -> bool) x (bool -> bool) -> bool

7/
a/

Function	Referencing Environment						
Main	a//1	b//1	c//1	sub1			
Sub1	a//2	b//1	c//1	sub1	sub2	sub3	
Sub2	a//3	b//1	c//3	f//3	sub1	sub2	sub3
Sub3	a//2	b//4	c//1	sub1	sub2	sub3	

b/

	Main	->	sub1	->	sub2	->	sub3
a	0		3		1		
b	1				2		1
c	4						
				f	sub3	return value	1

- The return value 1 from sub3 is the resulting value of the call expression f(a) in the body of sub2

- f(c) is called with c = 2, and therefor sub3(2) is called. In the body sub3, with b = 2 (actual parameter passed from sub2), c = 4 and a = 3 it will return the value 2*4-3 = 5 to sub2.

- Next sub2 will return (f(a) - f(c) * 2 = (1 - 5) * 2 = -8 to sub1 at the line code b = sub2 (1,2,sub3)
- With this, the value of the b//1 in the scope of main is set to -8
- Lastly, the sub1 return and control transfer back to main function just after the line where it's call, and so line //5 is execute and print -8 to the screen.

8/
a/ Pass by value-result:
i = 0
    s = s + a = 0 + 4 = 4
    A[j] = A[j] - 1 => A[0] = 3
i = 1
    s = s + a = 4 + 4 = 8
    A[j] = A[j] - 1 => A[0] = 2
i = 2
    s = s + a = 8 + 4 = 12
    A[j] = A[j] - 1 => A[0] = 2

At the end:
    s = 12
    a = 4 => A[0] = 4
Result: 124614

b/ Pass by reference:
i = j = 0
    s = s + a = 0 + 4 = 4
    A[j] = A[j] - 1 => A[0] = 3
i = j = 1
    s = s + a = 4 + 3 = 7 (a = A[0])
    A[j] = A[j] - 1 => A[1] = 5
i = j = 2
    s = s + a = 7 + 3= 10 (a = A[0])
    A[j] = A[j] - 1 => A[2] = 13
Result: 103513

c/ Pass by name:
i = j = 0
    s = s + a = 0 + 4 = 4 (a = A[j] = A[0] = 4)
    A[j] = A[j] - 1 => A[0] = 3
i = j = 1
    s = s + a = 4 + 6 = 10 (a = A[j] = A[1] = 6)
    A[j] = A[j] - 1 => A[1] = 5
i = j = 2
    s = s + a = 10 + 14= 24 (a = A[j] = A[2] = 14)
    A[j] = A[j] - 1 => A[2] = 13
Result: 243513