# Question 1

This question was relatively straight forward. In order to implement “For” loops in this version of mini I began by completing the compilation schemes in the various initializers, **NoInit.java**, **VarDeclInit.java**, and **ExprInit.java**. These modifications were made as follows

**NoInit.java**

public void compile**(**Assembly a**,** int pushed**)** **{**

**return;** // no init so nothing to do.

**}**

**VarDeclInit.java**

public void compile**(**Assembly a**,** int pushed**)** **{**

**return;** // no init so nothing to do.

**}**

**ExprInit.java**

public void compile**(**Assembly a**,** int pushed**)** **{**

**return;** // no init so nothing to do.

**}**

The final piece needed to complete this question was to complete the compilation scheme in **For.java.** My solution followed the method described in class and in the while loop. The only difference being that an extra step was needed to increment the step counter. My final solution is illustrated below:

**For.java**

public void compile**(**Assembly a**,** int pushed**)** **{**

// TODO: Provide an implementation for this method

String Loop **=** a**.**newLabel**();**

String Test **=** a**.**newLabel**();**

init**.**compile**(**a**,** pushed**);**

a**.**emit**(**"jmp"**,** Test**);**

a**.**emitLabel**(**Loop**);**

body**.**compile**(**a**,** pushed**);**

**if** **(**step **!=** **null)**

step**.**compileExpr**(**a**,** pushed**,** 0**);**

a**.**emitLabel**(**Test**);**

**if** **(**test **!=** **null)**

test**.**branchTrue**(**a**,** pushed**,** 0**,** Loop**);**

**else**

a**.**emit**(**"jmp"**,** Loop**);**

**}**

**Testing**

**After crafting such a fine compilation scheme I had to make sure it work so I created some for loops in for.mini being sure to test each type of initializer and took my new code generator out for a spin…**

**for.mini**

void mini\_main**()** **{**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int total **=** 0**;**

**for** **(**int i**=**0**;** i**<**10**;** i**=**i**+**1**)** **{**

// break;

print i**;**

total **=** total **+** i**;**

// continue;

**}**

print total**;**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int total **=** 0**;**

int i **=** 0**;**

**for** **(;** i**<**10**;** i**=**i**+**1**)** **{**

// break;

print i**;**

total **=** total **+** i**;**

// continue;

**}**

print total**;**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int total **=** 0**;**

int i **=** 0**;**

**for** **(**5**\***5**;** i**<**10**;** i**=**i**+**1**)** **{**

// break;

print i**;**

total **=** total **+** i**;**

// continue;

**}**

print total**;**

**}**

**for.out**

Total failures found**:** 0

print**:** 0

print**:** 1

print**:** 2

print**:** 3

print**:** 4

print**:** 5

print**:** 6

print**:** 7

print**:** 8

print**:** 9

print**:** 45

print**:** 0

print**:** 1

print**:** 2

print**:** 3

print**:** 4

print**:** 5

print**:** 6

print**:** 7

print**:** 8

print**:** 9

print**:** 45

print**:** 0

print**:** 1

print**:** 2

print**:** 3

print**:** 4

print**:** 5

print**:** 6

print**:** 7

print**:** 8

print**:** 9

print**:** 45

# **Question 2**

This required us to implement functionality for break and continue statements. The implementation of this seemed to be straight forward at first but ended up becoming a little tricky as I tried to implement the the code generation for them.

First of all it seemed to me that bread and continue statements at the core were the same beast with subtle differences. Break statements essentially just skip to the end of the loop and quit, whereas continue statements skip to the beginning of the loop and continue on. So the solution seemed simple enough, all I had to do was just emit break and jump labels in the appropriate places.

As I went to write code for this, however I soon realized the trick was knowing which label to jump to since the **break.java** class and the **continue.java** classes did not know which labels had been printed out. To solve this problem I decided to keep track of lables with static stacks in the **break.java** class and the **continue.java** class.

Since break and continue were compiled in the body *before* the Lables they would be required to jump to I decided to create new lables their respective compile()methods emit them and then push them onto their stacks. Then when For.compile() resumed it would check each stack and pop off a label if there was one there. This way I could jump to the proper place even in nested loops.

I have included the relevant additions below:

**break.java**

class Break **extends** Stmt **{**

static Stack**<**String**>** stack **=** **new** Stack**<**String**>();**

**...**

/\*\* Generate code for executing this statement.

\*/

public void compile**(**Assembly a**,** int pushed**)** **{**

String breakTo **=** a**.**newLabel**();**

a**.**emit**(**"jmp"**,** breakTo**);**

stack**.**push**(**breakTo**);**

**}**

**}**

**continue.java**

class Continue **extends** Stmt **{**

static Stack**<**String**>** stack **=** **new** Stack**<**String**>();**

**...**

/\*\* Generate code for executing this statement.

\*/

public void compile**(**Assembly a**,** int pushed**)** **{**

String continueTo **=** a**.**newLabel**();**

a**.**emit**(**"jmp"**,** continueTo**);**

stack**.**push**(**continueTo**);**

**}**

**}**

**For.compile()**

public void compile**(**Assembly a**,** int pushed**)** **{**

// TODO: Provide an implementation for this method

String Loop **=** a**.**newLabel**();**

String Test **=** a**.**newLabel**();**

init**.**compile**(**a**,** pushed**);**

a**.**emit**(**"jmp"**,** Test**);**

a**.**emitLabel**(**Loop**);**

body**.**compile**(**a**,** pushed**);**

// If we have encountered continues in the compilation of the body

// we need to emit the continue label to jump to here

**if** **(!**Continue**.**stack**.**isEmpty**())**

a**.**emitLabel**(**Continue**.**stack**.**pop**());**

**if** **(**step **!=** **null)**

step**.**compileExpr**(**a**,** pushed**,** 0**);**

a**.**emitLabel**(**Test**);**

**if** **(**test **!=** **null)**

test**.**branchTrue**(**a**,** pushed**,** 0**,** Loop**);**

**else**

a**.**emit**(**"jmp"**,** Loop**);**

// If we hae encountered in breaks in the compilation of the body

// we need to emit the break label to jump to here

**if** **(!**Break**.**stack**.**isEmpty**())**

a**.**emitLabel**(**Break**.**stack**.**pop**());**

**}**

**}**

**Testing**

**Of course adequate testing was required, so the same for loops were run again, this time with breaks and continues added. I also added some loops that would not be able to terminate without as well as some nested loops to ensure the labels were being printed in the correct order.**

**for.mini**

void mini\_main**()** **{**

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int total **=** 0**;**

**for** **(**int i**=**0**;** i**<**10**;** i**=**i**+**1**)** **{**

**break;**

print i**;**

total **=** total **+** i**;**

**continue;**

**}**

print total**;**

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int total **=** 0**;**

int i **=** 0**;**

**for** **(;** i**<**10**;** i**=**i**+**1**)** **{**

**break;**

print i**;**

total **=** total **+** i**;**

**continue;**

**}**

print total**;**

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int total **=** 0**;**

int i **=** 0**;**

**for** **(**5**\***5**;** i**<**10**;** i**=**i**+**1**)** **{**

**break;**

print i**;**

total **=** total **+** i**;**

**continue;**

**}**

print total**;**

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**for** **(**int i**=**0**;** i**<**5**;** i**=**i**+**1**){**

**for** **(** **;** **;** **)** **{**

int k **=** 42**;**

print k**;**

**break;**

**}**

**continue;**

print i**;**

**}**

int i **=** 0**;**

print i**;**

**}**

**for.out**

Total failures found**:** 0

print**:** 0

print**:** 0

print**:** 0

print**:** 42

print**:** 42

print**:** 42

print**:** 42

print**:** 42

print**:** 0

# **Question 3**

This question was a bear. The description mentioned that it might be tricky, but I found that it started at “tricky” took a detour through “mind bending” and landed me right in the middle of seg fault city where I camped for about a week. After many attempts and a lot of wrong turns were made I finally came up with a solution that seems to work rather well and produces correct output for the tests provided.

I started by filling in Id.compileExpr()and Assign.compileExpr(). I figured these would be the easiest functions to write so I got them out of the way. I implemented them exactly as described in the assignment description. For Id.compileExpr()the idea was to load an address from the stack into a register than to use that address to load the proper value from that location in memory as follows:

**Id.compileExpr()**

public void compileExpr**(**Assembly a**,** int pushed**,** int free**)** **{**

**if** **(this.**getVe**().**isByRef**()){**

a**.**emit**(**"# ID-----------------------------------------------"**);**

// load addy from stack to register

a**.**emit**(**"movl"**,** **this.**fromStackFrame**(**a**),** a**.**reg**(**free**));**

// load value at that address into register.

a**.**emit**(**"movl"**,** a**.**indirect**(**0**,**a**.**reg**(**free**)),** a**.**reg**(**free**));**

a**.**emit**(**"# -------------------------------------------------"**);**

**}**

**else**

a**.**emit**(**"movl"**,** **this.**fromStackFrame**(**a**),** a**.**reg**(**free**));**

**}**

Assign.compileExpr()followed suit in much the same way

**Assign.compileExpr()**

public void compileExpr**(**Assembly a**,** int pushed**,** int free**)** **{**

rhs**.**compileExpr**(**a**,** pushed**,** free**);**

**if** **(**lhs**.**getVe**().**isByRef**()){**

a**.**emit**(**"# ASSIGN REF---------------------------------------"**);**

// move address lhs reffers to into reg

a**.**emit**(**"movl"**,** lhs**.**fromStackFrame**(**a**),** a**.**reg**(**free **+** 1**));**

// load rhs into that address

a**.**emit**(**"movl"**,** a**.**reg**(**free**),** a**.**indirect**(**0**,**a**.**reg**(**free**+**1**)));**

a**.**emit**(**"#--------------------------------------------------"**);**

**}**

**else** **{**

a**.**emit**(**"# ASSIGN VAL---------------------------------------"**);**

a**.**emit**(**"movl"**,** a**.**reg**(**free**),** lhs**.**fromStackFrame**(**a**));**

a**.**emit**(**"#--------------------------------------------------"**);**

**}**

**}**

My next task was to modify the **Call.compile()**method to scan the argument list and add the appropriate space to the stack. In implementing this I decided to take the approach laid out in the assignment. To be safe I added space for every argument that was passed by reference whether or not it would need extra space on top of the arg list or not and ended up with a stack layout that resembled the figure below where v1...vn are the spaces created for new values which correspond to their respective argument:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ... | a1 | ... | an | v1 | ... | vn | ...padding... | ri | ... | rj | ...scratch... | old | ret | | ... |
| esp | |  |  | | | |  |  |  |  |  | ebp | | S | |

So once I had created this space I needed to use it to store new values which I did in Args.java as follow

**Args.java**

static void compileArgs**(**Assembly a**,** int pushed**,** Args args**)** **{**

int valueSpace **=** argBytes**(**args**);**

**for** **(**int offset**=**0**;** args**!=null;** args**=**args**.**rest**)** **{**

**if** **(**args**.**isByRef**()){**

a**.**emit**(**"#COMPILE ARGS---------------------------------"**);**

**if** **(**args**.**getArg**()** **instanceof** Id**)**

//compile basic variables straight to stack

args**.**arg**.**compileRefToStack**(**a**,** pushed**,** 0**,** offset**,**

**((**Id**)** args**.**getArg**()).**getVe**().**getOffset**());**

**else{**

// compile this argument, and store new value on stack

args**.**arg**.**compileToStack**(**a**,** pushed**,** 0**,** offset **+** valueSpace**);**

// a.reg(0) now holds mem addy of esp

a**.**emit**(**"movl"**,** "%esp"**,** a**.**reg**(**0**));**

// add offset of new value location

a**.**emit**(**"addl"**,** a**.**immed**(**offset **+** valueSpace**),** a**.**reg**(**0**));**

// now we want to push this new addy on the stack.

a**.**emit**(**"movl"**,** a**.**reg**(**0**),** a**.**indirect**(**offset**,** "%esp"**));**

**}**

a**.**emit**(**"#---------------------------------------------"**);**

**}**

**else**

// compile this argument, writing final value on the stack

args**.**arg**.**compileToStack**(**a**,** pushed**,** 0**,** offset**);**

// compute the offset for the next argument

offset **+=** Assembly**.**WORDSIZE**;**

**}**

**}**

You’ll notice in the code above that I made calls to some new methods in Expr.java. The purpose of this function was just to take the address which stored the value of the specific Id and push it on to the argument list of the new stack. In retrospect there really was no need to create that function and the code could have been kept in Args.java, but I think it may have kept the code a little cleaner and for the sake of completeness I have included that method below:

**Expr.CompileReftoStack()**

public void compileRefToStack**(**Assembly a**,** int pushed**,** int free**,** int offset**,** int env**)** **{**

a**.**emit**(**"# COMPILE REF TO STACK-------------------------------"**);**

// a.reg(free) now holds mem addy of ebp

a**.**emit**(**"movl"**,** "%ebp"**,** a**.**reg**(**free**));**

// add offset of arg location

a**.**emit**(**"addl"**,** a**.**immed**(**env**),** a**.**reg**(**free**));**

// now we want to push the addy on the stack.

a**.**emit**(**"movl"**,** a**.**reg**(**free**),** a**.**indirect**(**offset**,** "%esp"**));**

a**.**emit**(**"# ---------------------------------------------------"**);**

**}**

**I** ran this code against the byrefs.mini test code and everything seemed to work fine except when I tried to call a byref function inside another by ref function passing the initial value that was passed by reference. I think this is because the second function received the address of the address of the original value rather that the address of the original value, but I was unable to rectify this problem. Below is this assembly code generated for byref.mini as well as the test cases and the output.

**Byref.mini**

void mini\_main**()** **{**

int x **=** 0**;**

byref**(**x**);**

print x**;**

byval**(**x**);**

print x**;**

byref**(**6**\***7**);**

byval**(**6**\***7**);**

**}**

void byref**(**int**&** y**)** **{**

y **=** y **+** 1**;**

print y**;**

**}**

void byval**(**int**&** y**)** **{**

y **=** y **+** 1**;**

print y**;**

**}**

**byref.s**

|  |
| --- |
| .file "byref.s"  .globl mini\_main  byval**:**  # **PROLOGUE-------------------------------------------**  pushl %ebp  movl %esp**,%ebp**  #**----------------------------------------------------**  movl $1**,%eax**  # ID**-------------------------------------------------**  movl 8**(%ebp),%ecx**  movl **(%ecx),%ecx**  # **---------------------------------------------------**  addl %ecx**,%eax**  # ASSIGN REF**---------------------------------------**  movl 8**(%ebp),%ecx**  movl %eax**,(%ecx)**  #**--------------------------------------------------**  # ID**-------------------------------------------------**  movl 8**(%ebp),%eax**  movl **(%eax),%eax**  # **---------------------------------------------------**  pushl %eax  **call** print  # **EPILOGUE-------------------------------------------**  movl %ebp**,%esp**  popl %ebp  **ret**  #**----------------------------------------------------**  byref**:**  # **PROLOGUE-------------------------------------------**  pushl %ebp  movl %esp**,%ebp**  #**----------------------------------------------------**  movl $1**,%eax**  # ID**-------------------------------------------------**  movl 8**(%ebp),%ecx**  movl **(%ecx),%ecx**  # **---------------------------------------------------**  addl %ecx**,%eax**  # ASSIGN REF**---------------------------------------**  movl 8**(%ebp),%ecx**  movl %eax**,(%ecx)**  #**--------------------------------------------------**  # ID**-------------------------------------------------**  movl 8**(%ebp),%eax**  movl **(%eax),%eax**  # **---------------------------------------------------**  pushl %eax  **call** print  # **EPILOGUE-------------------------------------------**  movl %ebp**,%esp**  popl %ebp  **ret**  #**----------------------------------------------------**  mini\_main**:**  # **PROLOGUE-------------------------------------------**  pushl %ebp  movl %esp**,%ebp**  #**----------------------------------------------------**  subl $4**,%esp**  movl $0**,%eax**  movl %eax**,-**4**(%ebp)**  subl $12**,%esp**  #COMPILE ARGS**--------------------------------------**  # COMPILE REF TO STACK**-------------------------------**  movl %ebp**,%eax**  addl **$-**4**,%eax**  movl %eax**,(%esp)**  # **---------------------------------------------------**  #**--------------------------------------------------**  **call** byref  addl $12**,%esp**  movl **-**4**(%ebp),%eax**  pushl %eax  **call** print  subl $8**,%esp**  #COMPILE ARGS**--------------------------------------**  # COMPILE REF TO STACK**-------------------------------**  movl %ebp**,%eax**  addl **$-**4**,%eax**  movl %eax**,(%esp)**  # **---------------------------------------------------**  #**--------------------------------------------------**  **call** byval  addl $12**,%esp**  movl **-**4**(%ebp),%eax**  pushl %eax  **call** print  subl $8**,%esp**  #COMPILE ARGS**--------------------------------------**  # COMPILE TO STACK**-----------------------------------**  movl $7**,%eax**  movl $6**,%ecx**  imull %ecx**,%eax**  movl %eax**,**4**(%esp)**  # **---------------------------------------------------**  movl %esp**,%eax**  addl $4**,%eax**  movl %eax**,(%esp)**  #**--------------------------------------------------**  **call** byref  #COMPILE ARGS**--------------------------------------**  # COMPILE TO STACK**-----------------------------------**  movl $7**,%eax**  movl $6**,%ecx**  imull %ecx**,%eax**  movl %eax**,**4**(%esp)**  # **---------------------------------------------------**  movl %esp**,%eax**  addl $4**,%eax**  movl %eax**,(%esp)**  #**--------------------------------------------------**  **call** byval  # **EPILOGUE-------------------------------------------**  movl %ebp**,%esp**  popl %ebp  **ret**  #**----------------------------------------------------** |

**byref.out**

Total failures found: 0

print: 1

print: 1

print: 2

print: 1

print: 43

print: 43

# Conclusion

In conclusion this was a nerve racking experience to be sure. However, I now feel I have a solid understanding of function calls and stack manipulation at a very low level so I feel good about that. I am still perturbed by the fact that I cannot seem to nest “by reference” function calls. ☹