Ben Reichert
CS322 HW#3

Question 1.

The For implementation was similar to a while loop, just with checks for null variables in the formal parameters. Here is the source code of the compile method in For.java:

```
public boolean compile(Assembly a, Frame f) {
     // Remember to allow for the possibility that
     // init, test, and step could each be null ...

     String lab1 = a.newLabel(); //we make a label for the top of the loop
     String lab2 = a.newLabel(); //
     if(init != null){
       init.compileExpr(a,f);
      //if init is not null we compile it, otherwise it is left out.
     }
     a.emit("jmp", lab2);
     a.emitLabel(lab1);
     if(step != null){
       step.compileExpr(a, f);
     }
     body.compile(a,f);
     a.emitLabel(lab2);   //label 2 is after the body but before the test
     if(test != null){    //if test is not null, compile the test branch condition
       test.branchTrue(a, f, lab1);
     }
     else {        //if it is null, we need to properly loop infinitely hehe
     /*infinite loop ftw*/
       a.emit("jmp", lab1);
     }
     return true;
 }
```

For testing For I wrote a series of for loops with either a missing init, test, or step. I also moved the step into the body instead of the formals and tested that to see if it worked (which it did). I verified that the for loop printed out the i variable properly each iteration and it worked. Infinite loops also worked :-) Here is my basic test of the For loop:

```
//for.prog
int x=1;
int y=2;
```

```
void main() {
  int x = 1;
  int y = 2;
  int i = 0;
  for (i = 0; i < 100; i = i+1){
    print i;
  }

  for (i = 0; i < 10;){
    //step is null
    print i;
    i = i+1;
  }

  //init is null
  i = 0;
  for(; i<5; i = i+1){
    print i;
  }

  //test is null, infinite loop
  //for(i = 0; ; i = i+1){ print i;}


  //all null, infinite loop
 // for(;;){print i;}
}
```

Question 2.

Most of the XinitGlobals method declaration took place in Defn.java:

ast/Defn.java:

```
 public static void compile(String name, Defn[] defns) {
        LocEnv   globals = null;
        LocEnv   env1 = null;
        Assembly a       = Assembly.assembleToFile(name);

        a.emit();
        a.emit(".data");


       //declare global variables storage (previous code)
        for (int i=0; i<defns.length; i++) {
            globals = defns[i].declareGlobals(a, globals);
```

```
        }

        a.emit();     //we want to make it print nice, so adda  newline
        a.emit(".text");     //we're in the text section for writing functions

        //shove initGlobals in here
        a.emit(".globl",a.name("initGlobals"));    //make the global statement so the
compiler recognizes the function den
        a.emitLabel(a.name("initGlobals")); //emit the function label

        a.emitPrologue();    //emit the prologue of the function

       //the secret sauce
       //the pasta with extra meatballs
        for(int i = 0; i < defns.length; i++){
         //we want to loop through the definitions like in the global declarations
         //each defns[i] is of type Globals, so we are calling Global.compileGlobals method
          defns[i].compileGlobals(a, new FunctionFrame(new Formal[0], globals));
         //since the very last compileExpr requires a Frame, we need to make an empty one
here and pass pass pass the Frame merrily merrily merrily merrily just pass the Frame.
        }

        a.emitEpilogue();    //emit the epilogue of the function
        a.emit();            //print a new line for awesomeness


       //compile functions (previous code)
        for (int i=0; i<defns.length; i++) {
            defns[i].compileFunction(a, globals);
        }
        a.close();
    }
```

Diving into ast/Globals.java:
I wrote a compileGlobals method in this file to call another compileGlobals method and another and another and another…until we get a down to the InitVarIntro level and we can do a compileExpr. Here is the change with comments:

```
void compileGlobals(Assembly a, Frame f){
       for(int i = 0; i < vars.length; i++){
         //vars is of parent class VarIntro, but of type InitVarIntro
         //so we loop through the vars and call the compileGlobals method in InitVarIntro
          vars[i].compileGlobals(a,f); //call initvarintro compileGlobals
       }
    }
```

Since InitVarIntro is an extension of VarIntro, we need to make a class declaration in ast/VarIntro.java:

```
public void compileGlobals(Assembly a, Frame f){}
```

Here is ast/InitVarIntro (this does most of the work). I basically borrowed the format from the compile method, but without the f.allocLocal(a,name,f.free64()); since I didn't want it to for one, allocate a local which pushed it onto the stack. Since we don't push the variable onto the stack, we have to explicitly move the return of the compileExpr into the proper global variable declaration. If we don't do this, the value is assigned, but never put back into the global scope. Here is the ast/InitVarIntro.java file:

```
public void compileGlobals(Assembly a, Frame f){
      exp.compileExpr(a,f); //compile the expression so we can evaluate things like x =
x*y*z; in the global namespace
      a.emit("movl","%eax","X"+name); //I guess this could also be a.name(name), but who
cares? I like it my way ;)
    }
```

So that's it! We now have all the code used to properly emit a XinitGlobals method and initialize the globals to their proper values. For testing, I used Mark's provide implicit.prog to ensure my global variables were initialized, the XinitGlobals method was produced in the ASM, and the globals were printed with the expected results in the body of Xmain.

Question 3.

Source code of solution.prog:

```
//BEN REICHERT
//CS322 HW#3
//FIBONACCI YEAH

void main(){
  int arg1 = 1;
  int arg2 = 2;
  int arg3 = 2;

  while(arg1 < 10){
    int arg4 = 0;
    while(arg4 < arg1){
```

```
      arg2 = f(arg2, arg4);
      print h(g(arg4));
      arg4 = arg4 + 1;
    }
    print arg2;
    arg2 = arg2 * arg3;
    arg1 = arg1 + 1;
  }

  print arg1;
}

int f(int arg1, int arg2){
  int local = (arg1 - arg2);
  local = local * g(local);
  return local;

}

int g(int arg){
  return ((arg * 2) - arg);
}

int h(int n){
  if(n < 2){
    return 1;
  }
    return (h(n - 1) + h(n - 2));
}
```

Please see attached 3 pages of my scribbly-scrobbylies all over the printed out ASM code to see my “process”.

There are a few ways that come to mind when thinking about how the source program might be different than the original source program. Off the top of my head I can think that variable names are lost in the compilation to assembly code, as are the formal parameter names that are passed in as arguments to a function. These things are stripped away as they are not needed during the compilation process. Therefore, my source program might have different argument variables names, and variable names that differ. I picked things like arg* for variables, and have one mention of an int local variable. These are the only things that should differ between my source code and the sample.prog source code.

Optimization 1:

```
Xf:
   …
   movl    %edi, %eax
   movl    %esi, %ecx
   subl    %ecx, %eax
   pushq   %rax
   …
```

can be optimized to:

```
Xf:
   …
   movl    %edi, %eax
   subl    %esi, %eax  #reduce one instruction move
   pushq   %rax
   …
```

Optimization 2:

```
Xf:
   …
   movl    %eax, -8(%rbp)
   movl    -8(%rbp), %eax
   …
```

can be optimized to:

```
Xf:
   …
   #remove both of those moves, they are not necessary. since one saves the variable onto the stack, and the other pulls it back down back into eax, there is no need since the value didn't change, it just moved around on the stack. And since the stack is slow compared to registers we should reduce these accesses :)
   …
```

Optimization 3:

```
Xg:
   …
   movl    %edi, %esi
   imull   %esi, %eax
   movl    %edi, %esi
   subl    %esi, %eax
   …
```

can be optimized to:

```
Xg:
   …
   imull   %edi, %eax
   subl    %edi, %eax
   …
```

Include all optimizations in code and compile and run it. I took the stdout of the sample.s and saved that in a file, as well as the output of the optimized code to another file. I then diffed the output to ensure the program continued to work as it should (and it did!).

```
breic2@ada:~/homework/322/hw3$ ./optimized  > op
breic2@ada:~/homework/322/hw3$ gcc -o sample sample.s runtime.c
breic2@ada:~/homework/322/hw3$ ./sample > samp
breic2@ada:~/homework/322/hw3$ diff op samp
breic2@ada:~/homework/322/hw3$
```

I had a great time with the reverse engineering section of the homework. Thank you for making fun exercises to do that actually inspire learning!

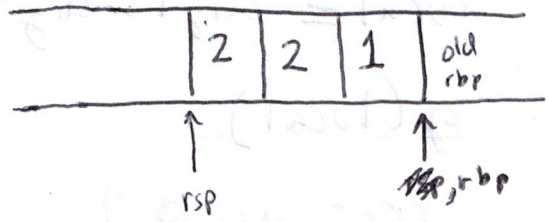
#1

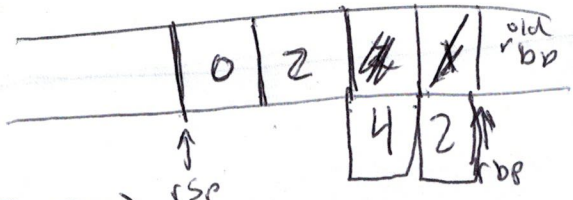
```
        .file   "out.s"

        .data

        .text
        .globl  Xmain
Xmain:
        pushq   %rbp        # Save old base ptr
        movq    %rsp, %rbp  # Move stack ptr to base ptr
        movl    $1, %eax    # eax=1
        pushq   %rax        # push onto stack
        movl    $2, %eax    # eax=2
        pushq   %rax        # push rax
        movl    $2, %eax
        pushq   %rax
        jmp     l1
l0:
        movl    $0, %eax
        pushq   %rax        # return 0 push onto stack
        jmp     l3
l2:
        movl    -16(%rbp), %edi   arg2
        movl    -32(%rbp), %esi   arg4
        call    Xf
        movl    %eax, -16(%rbp)
        movl    -32(%rbp), %edi
        call    Xg
        movq    %rax, %rdi        rdi
        call    Xh
        movq    %rax, %rdi
        call    Xprint
        movl    $1, %eax
        movl    -32(%rbp), %edi
        addl    %edi, %eax
        movl    %eax, -32(%rbp)
l3:
        movl    -8(%rbp), %eax
        movl    -32(%rbp), %edi
        cmpl    %eax, %edi
        jl      l2
        movl    -16(%rbp), %edi
        call    Xprint
        movl    -16(%rbp), %eax
        movl    -24(%rbp), %edi
        imull   %edi, %eax
        movl    %eax, -16(%rbp)
        movl    -8(%rbp), %eax
        movl    $1, %edi
        addl    %edi, %eax
        movl    %eax, -8(%rbp)
```

Fibonacci?





arg2 = xf(arg2, arg4)

xg(arg4)

FallThru

#eax = 1
edi = 0
if (edi < eax) {
  goto L2

#edi = 2
print 2
eax = 2
edi = 2
eax = 2*2 // edi * eax
eax = 1
// edi = 1
eax = 1+1

Main is off

If is off

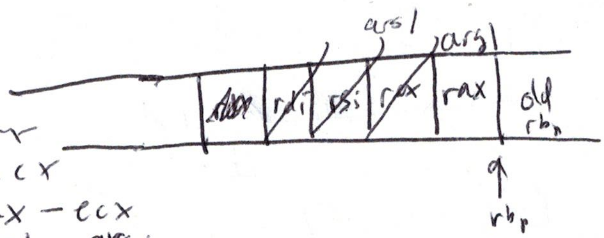
#2

```
    addq    $8, %rsp
l1:
    movl    $10, %eax
    movl    -8(%rbp), %edi
    cmpl    %eax, %edi
    jl      l0
    movl    -8(%rbp), %edi
    call    Xprint
    movq    %rbp, %rsp
    popq    %rbp
    ret

    .globl  Xf
Xf:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, %eax
    movl    %esi, %ecx
    subl    %ecx, %eax
    pushq   %rax
    movl    -8(%rbp), %eax
    pushq   %rax
    pushq   %rsi
    pushq   %rdi
    movl    -8(%rbp), %edi
    call    Xg
    movq    %rax, %rcx
    popq    %rdi
    popq    %rsi
    popq    %rax
    imull   %ecx, %eax
    movl    %eax, -8(%rbp)
    movl    -8(%rbp), %eax
    movq    %rbp, %rsp
    popq    %rbp
    ret

    .globl  Xg
Xg:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $2, %eax
    movl    %edi, %esi
    imull   %esi, %eax
    movl    %edi, %esi
    subl    %esi, %eax
    movq    %rbp, %rsp
    popq    %rbp
    ret
```

- Stack adjustment; Fall thru; While?
- # edi = 1
- # if (edi < 10) { goto l0 } else { # edi = 1 print 1 }
- redundant
- epilogue; remove base pointer
- prologue
- local vars from formals
- arg1 → eax; arg2 → ecx; eax = eax − ecx; arg1 = arg1 − arg2
- Optimize
- Save local stack
- across global stack
- move; g(−8)
- Why?
- restore local stack; g(arg1)
- arg1 o g(arg1) ret
- x = g(−8)
- x ·
- eax = (−8) · ecx
- eax = (−8) · X
- eax = (−8) · g(−8)
- Pointless
- prologue; eax = 2; arg1 → esi; arg1 ★ 2; arg1 → esi
- return (arg1 ★ 2) − arg1



#3

```
        .globl  Xh
Xh:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $2, %eax
        movl    %edi, %esi
        cmpl    %eax, %esi
        jnl     l4
        movl    $1, %eax
        movq    %rbp, %rsp
        popq    %rbp
        ret
l4:
        pushq   %rdi
        movl    -8(%rbp), %edi
        movl    $1, %esi
        subl    %esi, %edi
        call    Xh
        popq    %rdi
        pushq   %rax
        pushq   %rdi
        movl    -16(%rbp), %edi
        movl    $2, %esi
        subl    %esi, %edi
        call    Xh
        movq    %rax, %rsi
        popq    %rdi
        popq    %rax
        addl    %esi, %eax
        movq    %rbp, %rsp
        popq    %rbp
        ret
```

WHY NO INLINE???

xh

eax = 2

arg → esi

if (arg ≥ 2) {

} L4

return 1

local

Global

Save old rdi, which is arg1 ?

Stack1 → edi

esi = 1

Stack1 = Stack1 − 1

h(Stack1)

restore rdi

local Stack2 → edi

local − 2

h(local − 2)

2

local 2

rdi	rax	rdi	old rbp

rbp

rsp

return h(local − 2) + eax

arg = edi

h(local − 2) + 2