Compilers HW3

David Meyer

1 Labeled Break Statements:

```
L1:
...
L2:
...
L3:
goto, AFTER_L2, ,
assign, t0, a,
breq, a, 0, L3
AFTER_L3:
assign, t1, b,
breq, t1, 0, L2
AFTER_L2:
assign, t2, c,
breq, t2, 0, L1
AFTER_L1:
```

Essentially the algorithm we follow for instruction creation is similar to that of a while do loop, but flipped a little. So first we create a label for the beginning of the loop, then generate the instructions that are to be repeated. After this we can do our check for whether to loop back to the top or not. Lastly, in order to account for break statements that can break out of different nested loops, we add an endlabel.

Since I understand my own implementation of IR code generation a lot better than AST trees, I will explain how I would implement this with my own code. The way my implementation works is that I have a list of IR objects. Each IR object contains an opcode and a number of parameters. Essentially each specific opcode has it's own constructor. The one for break is very simple because it contains no parameters. I can easily add another constructor that takes in a label parameter.This would allow be to generate the goto, <Label>,, as seen above. So what I would do is as I parse through if I find "break" I will check if the next token is a semicolon or a label. If it is a label then I will create the correct kind of IR.

Another caveat is that in order for this grammar given to work we would need to allow a label or id to start a do-while loop. So the labels L1, L2, L3 would be in the tiger code already. Knowing this, when I generate code I will keep it as a standard to take the ID for the loop and generate the after label to always be "AFTER_<ID>".

This way I can guarantee when I see a "break <ID>;" I know I can in
fact generate the coee "goto, AFTER<ID>,," and it will
work because that label will automatically be generated below the loop block.

2 Liveness

Below is the divided blocks of code. I divided them basically by
logically deciding which blocks will be repeated. Because of
this some blocks will inevitably fall through
to other blocks.

2.1 BLOCK1

```
main:
assign, a, 1.0, // a = 1.0
assign, b, 1.0,
assign, c, 1.0,
assign, i, 0,
assign, t1, 5,
assign, t0, 0,
```

2.2 BLOCK2

```
label0:
brgt, t0, t1, label1 // if (t0 > t1) goto label 1, else fall through
```

2.3 BLOCK3

```
assign, i, t0,
add, t0, 1, t0 // t0 = t0 + 1
mult, a, 1.5, t2 // t2 = a * 1.5
assign, b, t2,
add, a, b, t3
assign, b, t3,
mult, 2.0, a, t4
assign, c, t4,
mult, a, c, t5
assign, a, t5,
mult, 1.5, a, t6
assign, b, t6,
add, a, b, t7
assign, c, t7,
brneq, i, 4, label2 // if i <> 4) go to label 2, else fall through
```

2.4 BLOCK4

```
add, a, 1.0, t8
assign, c, t8,
goto, label3, ,
```

2.5 BLOCK5

```
label2:
add, a, c, t9
assign, c, t9,
```

2.6 BLOCK6

```
label3:
brgeq, j, 5, label4 // if ( j >= 5), go to label 4 else fall through
```

2.7 BLOCK7

```
brgeq, i, 3, label5
```

2.8 BLOCK8

```
add, j, 1, t10
assign, j, t10,
```

2.9 BLOCK9

```
label5:
goto, label3, , // unconditional branch go to label 3
```

2.10 BLOCK10

```
label4:
goto, label0, ,
```

2.11 BLOCK11

```
label1:
add, 2.0, c, t11
assign, a, t11,
mult, a, a, t12
assign, c, t12,
brleq, c, 4.0, label6 // if (c < = 4) go to label 6, else fall through
```

2.12 BLOCK12

```
add, a, c, t13
assign, a, t13,
goto, label7, ,
```

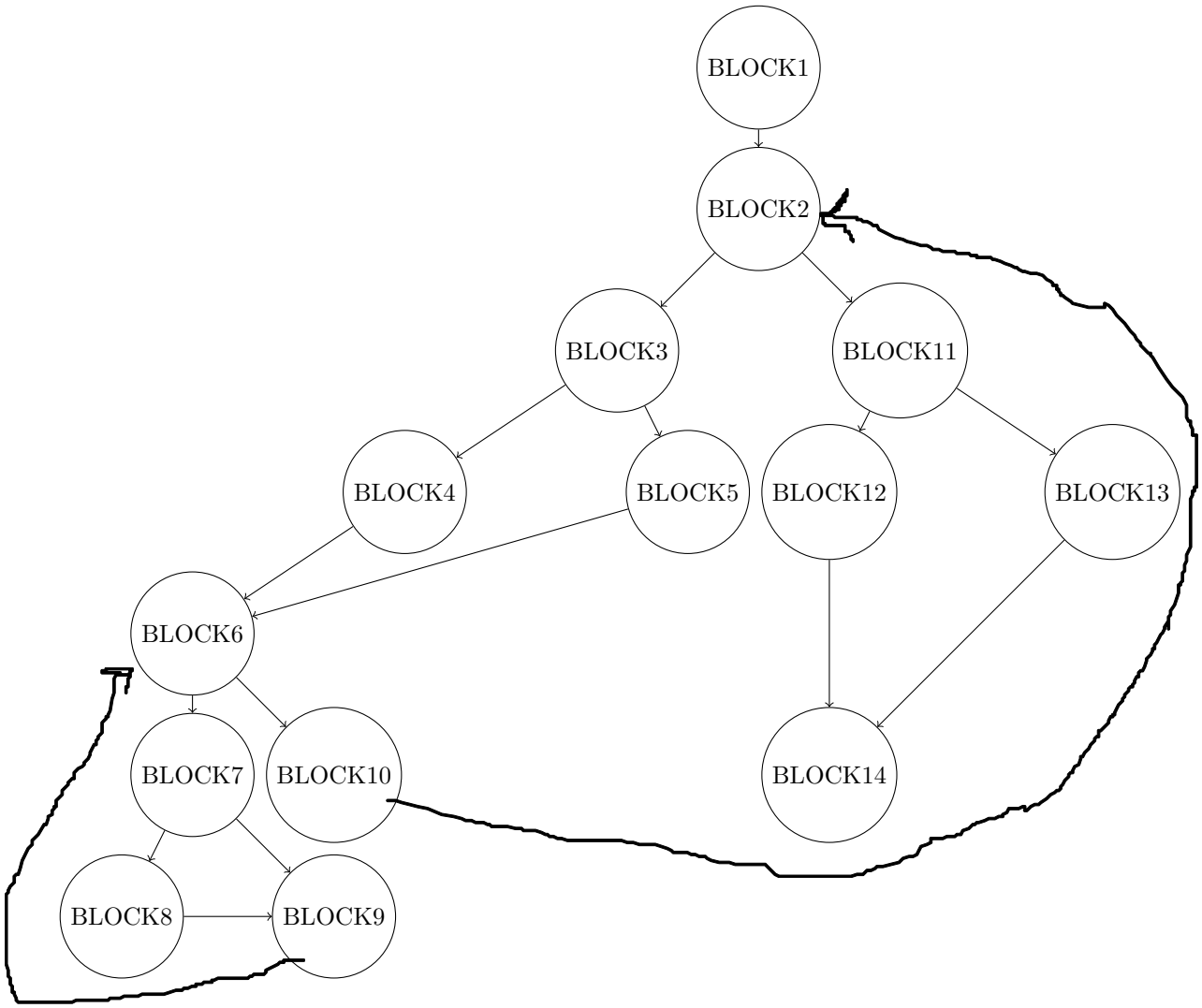
2.13 BLOCK13

```
label6:
mult, a, c, t14
assign, c, t14,
```

2.14 BLOCK14

```
label7:
return, , , // the function returns to caller
```

Below is the CFG for the above blocks. Each node will be labeled "BLOCK<#>"
It is not perfectly how the book has it, but I think it gets the main point across.
In the book it lists the blocks as actual blocks where I have circles.
The book also has the blocks listed sequentially in terms of the y-axis,
where mine are graphically more level ordered like a tree. I didn't think
this would make a big difference graphically, since the visual is just an
aesthetic way to view the concept.



Below is the live analysis for each variable.

Below is the computation for UEVars and VarKills sets for each block in the code. This will allow us to compute liveness through the algorithm:
LiveOut(n) = UEVar(m)|| (LiveOut(m) && ~VarKill(m)).

- BLOCK1: UEVar: VARKill: a,b,c,i,t1,t0
- BLOCK2: UEVar: t0, t1 VARKill:
- BLOCK3: UEVar: t0,a,t2,b,t3,i VARKill: i,t0,t2,b,t4,t5,t6,t7,c
- BLOCK4:UEVar: a VARKill: t8,c

- BLOCK5:UEVar: a,c VARKill: t9,c
- BLOCK6:UEVar: j VARKill:
- BLOCK7:UEVar: i VARKill:
- BLOCK8:UEVar: j VARKill: t10,j
- BLOCK9:UEVar: VARKill:
- BLOCK10:UEVar: VARKill:
- BLOCK11:UEVar: c,a VARKill: t11,t12
- BLOCK12:UEVar: a,c VARKill: t13
- BLOCK13:UEVar: a,c VARKill: t14
- BLOCK14:UEVar: VARKill:

`Using the algorithm: LiveOut(n) = UEVar(m)|| (LiveOut(m) && ~VarKill(m)), below is the LiveOut sets for each block.`

- BLOCK1: t0,t1,t2,t3,j
- BLOCK2: a,b,i,t0,t2,t3,j
- BLOCK3: a,j
- BLOCK4: a,c,j,i
- BLOCK5: i,j,a
- BLOCK6: i,j,a,c
- BLOCK7: j,a,c
- BLOCK8: a,c
- BLOCK9: a,c
- BLOCK10: a,c
- BLOCK11: a,c
- BLOCK12: a,c
- BLOCK13:
- BLOCK14:

`Finally, using these sets we can see how long each variable is alive and in which blocks`

- a: [B1.2-B1.7,B2,B3.3-B3.13,B4.1-B4.3,B5.2-B5.3,B6,B7,B8,B9,B10,B11.3-B11.4,B12.1-B12.2,B13.2-B13.3]

- b: [B1.3-B1.7,B2,B3.4-B3.13]
- c: [B1.4-B1.7, B2,B3.8-B3.14,B4.2-B4.3,B5.2-B5.3,B6,B7,B8,B9,B10,B11.2-B11.6,B12,B13.2-B13.3]
- t0: [B1.7, B2.2, B3.1-B3.2]
- t1: [B1.6-B1.7, B2.2]
- t2: [B3.3-B3.4]
- t3: [B3.5-B3.6]
- t4: [B3.7-B3.8]
- t5: [B3.9-B3.10]
- t6: [B3.11-B3.12]
- t7: [B3.13-B14]
- t8: [B4.1-B4.2]
- t9: [B5.2-B5.3]
- t10:[B8.1-B8.2]
- t11: [B11.2-B11.3]
- t12: [B11.4-B11.5]
- t13: [B12.1-B12.2]
- t14: [B13.2-B13.3]
- j: [B6.2,B8.1-B8.2]
- i: [B1.5-B1.7,B2,B3.1-B3.15,B4,B5,B6,B7.1]