Code generation exercises

Function body

Transform the following code into java bytecode:

```
def middle(small: Int, big: Int): Int = {
  val mid = small + (big - small) / 2
  return mid
}
```

```
small: local variable 0
big: local variable 1
mid: local variable 2
```

```
iload_addr, istore_addr, iadd, idiv, isub, iconst_2
```

Function body

Transform the following code into java bytecode:

```
def middle(small: Int, big: Int): Int = {
  val mid = small + (big - small) / 2
  return mid
}
```

```
[val mid = small + (big - small) / 2; mid]
ireturn
```

Function body

Transform the following code into java bytecode:

```
def middle(small: Int, big: Int): Int = {
  val mid = small + (big - small) / 2
  return mid
}
```

```
[[val mid = small + (big - small) / 2]] l2
iload_2
ireturn
```

Function body

Transform the following code into java bytecode:

```
def middle(small: Int, big: Int): Int = {
  val mid = small + (big - small) / 2
  return mid
}
```

```
[[small]][[big]] [[small]] [[-]] [[2]] [[/]] [[+]]
istore_2
iload_2
ireturn
```

Function body

Transform the following code into java bytecode:

```
def middle(small: Int, big: Int): Int = {
  val mid = small + (big - small) / 2
  return mid
}
```

```
[[small]] [[big]] [[small]] [[-]] [[2]] [[/]]
iadd
istore_2
iload_2
ireturn
```

Function body

Transform the following code into java bytecode:

```
def middle(small: Int, big: Int): Int = {
  val mid = small + (big - small) / 2
  return mid
}
```

```
iload_0              idiv
iload_1              iadd
iload_0              istore_2
isub                 iload_2
iconst_2             ireturn
```

Function body

Transform the following code into java bytecode:

```
def middle(small: Int, big: Int): Int = {
  val mid = small + (big - small) / 2
  return mid
}
```

```
iload_0
iload_1
iload_0
isub
iconst_2
idiv
iadd
ireturn
```

binarySearch

```
def binarySearch(array: Array[Int], value: Int, left: Int,
right: Int): Int = {
  if (left > right)
      return -1
  val middle = (left + right) / 2
  if (array(middle) == value)
    return middle
  else
    if (array(middle) > value)
      return binarySearch(array, value, left, middle - 1)
    else
      return binarySearch(array, value, middle + 1, right)
}
```

binarySearch - **bit.ly/1aCuKZz**

Stack state:

```
def binarySearch(array: Array[Int], value: Int, left: Int,
right: Int): Int
```

Methods:

```
#1 : binarySearch
```

Local variables mapping:

```
0:object itself
1:array
2:value
3:left
4:right
5:mid
```

```
Use the destination passing style
```

binarySearch - **bit.ly/1aCuKZz**

```
[[if (left > right) return -1; …]]
        iload_3
        iload_4
        if_icmple goto after1:
        iconst_1
        ineg
        goto return
after1: [[…]]
return: ireturn
```

binarySearch - **bit.ly/1aCuKZz**

```
[[val middle = (left + right) / 2; …]]
                => left right + 2 /
after1: iload_3
        iload_4
        iadd
        iconst_2
        idiv
        istore_5
        [[…]]
return: ireturn
```

binarySearch - **bit.ly/1aCuKZz**

```
[[if (array(middle) == value) return middle else …]]
        aload_1
        iload_5
        iaload
        iload_2
        if_cmpne goto after2:
        iload_5
        goto return:
after2: [[…]]
return: ireturn
```

binarySearch - **bit.ly/1aCuKZz**

```
[[if (array(middle) > value)
return binarySearch(array, value, left, middle - 1)
else …]]
after2: aload_1
        iload_5
        iaload
        iload_2
        if_cmple goto after3:
        aload_0 // Object itself
        aload_1
        iload_2
        iload_3
        iload_5
        iconst_1
        isub
        invokevirtual #1
        goto return:
after3: [[…]]
return: ireturn
```

binarySearch - **bit.ly/1aCuKZz**

```
[[return binarySearch(array, value, middle+1, right)]]
after3: aload_0
        aload_1
        iload_2
        iload_5
        iconst_1
        iadd
        iload_4
        invokevirtual #1
return: ireturn
```

Branching conditions

```
Boolean b;
int f(int x, int y, int z) {
  while ((!b && (x > 1 *(y+z)) || (x < 2*y +
z)) {
      x = x + 3
  }
  return x;
}
Context: b is field 0 of object. (aload, getfield)
         x -> 1, y -> 2, z -> 3
```

Branching conditions

```
Boolean b;
int f(int x, int y, int z) {
  while ((!b && (x > 1 *(y+z)) || (x < 2*y + z)) {
     x = x + 3
  }
  return x;
}
lLoop:  branch(condition, body, lAfter)
 body:  [x=x+3]
        goto lLoop
lAfter: iload_0
        ireturn
```

Translating the body

```
[|x=x+3|]

    iload_1
    iconst_3
    iadd
    istore_1
```

Translating complex branching

```
((!b && (x > 2 *(y+z)) || (x < 2*y + z)) {
lLoop:  branch(c1 || c2, body, lAfter)
=>

lLoop:  branch(c1, ???, ???)
c1No:   branch(c2, ???, ???)
body:   [|x=x+3|]
        goto lLoop
lAfter: iload_0
        ireturn
```

Translating complex branching

```
((!b && (x > 2 *(y+z)) || (x < 2*y + z)) {
lLoop:  branch(c1 || c2, body, lAfter)
=>

lLoop:  branch(c1, body, c1No)
c1No:   branch(c2, body, lAfter)
body:   [|x=x+3|]
        goto lLoop
lAfter: iload_0
        ireturn
```

Translating complex branching

```
(!b && (x > 2 *(y+z)) {
lLoop:  branch(c11 && c12, body, c1No)
=>

lLoop:  branch(c11, ???, ???)
c11Yes: branch(c12, ???, ???)
c1No:   branch(x < 2*y + z, body, lAfter)
body:   [|x=x+3|]
        goto lLoop
lAfter: iload_0
        ireturn
```

Translating complex branching

```
(!b && (x > 2 *(y+z)) {
lLoop:  branch(c11 && c12, body, c1No)
=>

lLoop:  branch(c11,  c11Yes, c1No)
c11Yes: branch(c12,  body, c1No)
c1No:   branch(x < 2*y + z, body, lAfter)
body:   [|x=x+3|]
        goto lLoop
lAfter: iload_0
        ireturn
```

Translating complex branching

```
lLoop:  branch(!b,  c11Yes, c1No)
c11Yes: branch(x > 2 *(y+z),  body, c1No)
c1No:   branch(x < 2*y + z, body, lAfter)
body:   [|x=x+3|]
        goto lLoop
lAfter: iload_0
        ireturn
```

Translating complex branching

```
lLoop:  branch(b,  c1No, c11Yes)
c11Yes: branch(x > 2 *(y+z),  body, c1No)
c1No:   branch(x < 2*y + z, body, lAfter)
body:   [x=x+3]
        goto lLoop
lAfter: iload_0
        ireturn
```

Translating complex branching

```
lLoop:  branch(b,  c1No, c11Yes)
c11Yes: [x]
        [2 *(y+z)]
        if_cmpgt body
c1No:   [x]
        [2*y + z]
        if_cmplt body
        goto lAfter)
body:   [x=x+3]
        goto lLoop
lAfter: iload_0
        ireturn
```

Translating complex branching

```
lLoop:  [b]
        [0]
        if_cmpne c11No
c11Yes: [x]
        [2 *(y+z)]
        if_cmpgt body
c1No:   [x]
        [2*y + z]
        if_cmplt body
        goto lAfter
body:   [x=x+3]
        goto lLoop
lAfter: iload_0
        ireturn
```

Translating complex branching

```
lLoop:  aload_0
        getfield
        iconst_0
        if_cmpne c11No
c11Yes: iload_1
        [2] [y] [z] [+] [*]
        if_cmpgt body
c1No:   iload_1
        [2] [y] [*] [z] [+]
        if_cmplt body
        goto lAfter)
body:   [x=x+3]
        goto lLoop
lAfter: iload_0
        ireturn
```

Translating complex branching

```
lLoop:  aload_0
        getfield
        iconst_0
        if_cmpne c11No
c11Yes: iload_1
        iconst_2
        iload_2
        iload_3
        iadd
        imul
        if_cmpgt body
c1No:   iload_1
        iconst_2
        iload_2
        imul
        iload_3
        iadd
        if_cmplt body
        goto lAfter
body:   iload_1
        iconst_3
        iadd
        istore_1
        goto lLoop
lAfter: iload_0
        ireturn
```

Designing Code Generators

- Can we design a byte-code translation for the construct

```
repeat {
    S1
} until (condition)
```

The statement S1 should be repeatedly executed as long as the condition does not hold.

Designing Code Generators 2

- Design a byte-code translation for a switch construct defined as follows:

```
switch(i) {
    case c1 => e1
    case c2 => e2
    …
    case cn => en
}
```