CS558 Programming Languages – Fall 2023 – Study Questions Lecture 2a

These questions are intended for self-study, to help review and deepen your understanding of the lecture. Sample answers are available. There is nothing to hand in.

1. (a) The schematic machine code for `while` loops shown on slide 19 is slightly inefficient because it executes two branch instructions (one conditional and one unconditional) on *each* iteration of the loop body. Write down a better schematic machine code sequence that executes just one branch per iteration of the body. (Hint: Your sequence can start with an unconditional branch that is executed just once, no matter how many times the loop iterates.)

(b) Slide 19 shows how a `repeat` statement can be translated into a `while` statement, but the translation duplicates the code for the loop body s. Write down a schematic machine code sequence for `repeat` that does not have this duplication.

2. Consider compiling the following case statement, where `e` is an integer-valued expression and the `s`i are arbitrary statements.

```
case e of
  4  : s4
  17 : s17
  13 : s13
  99 : s99
  2  : s2
  88 : s88
  12 : s12
  default: sd
end
```

Slide 17 showed how this can be compiled into a linear chain of `if-then-else` statements. For n case labels, evaluating this code requires in the worst case $O(n)$ equality comparisons—seven in this example.

Suppose instead that we want to compile the `case` statement into code that does a *binary search* on the possible values to dispatch to the correct sub-statement. This should be possible using only $O(\log n)$ comparisons. Assume that the target language of our compiler has (only) a three-way comparison primitive that compares a value against zero and branches to one of three sub-statements based on whether the result is negative, zero, or positive:

```
ifsign v
  <: s1    // execute this if v is negative
  =: s2    // execute this if v is zero
  >: s3    // execute this if v is positive
```

For example, the following code prints "`is positive`".

```
x = 10+20
```

```
ifsign x
  <: print "is negative"
  =: print "is zero"
  >: print "is positive"
```

(The IBM 704, an early processor that was the original compilation target for FORTRAN, had a similar instruction, which inspired a similar operator in FORTRAN called the "arithmetic `IF`." More modern processors don't have a three-way comparison instruction like this, but it is easily synthesized from a pair of ordinary binary comparisons without affecting the asymptotic complexity of the code. We use it here to make the exponential improvement given by binary search more obvious.)

Show pseudo-code (in the style of the slide) for the result of compiling this example into a decision tree of nested `ifsign` statements. Executing your code should require evaluating only three `ifsign` tests in the worst case. Note that multiple leaves of your tree will need to execute the statement `sd`; an ideal solution will avoid duplicating that statement (which could in general be a large block), but this is not the crucial point here.

3. Recall that in Scala, we can write a loop that iterates over consecutive numbers $n, n+1, n+2, \ldots, m$ as `for (i <- n to m)`. As suggested in slide 21, we can also write `for (i <- c)` to iterate over any collection `c` that has an `iterator` method. In fact, the first of these is just a special case of the second: the Scala class `Range` is just a particular kind of collection that represents a set of consecutive integers. This is a neat economy in the Scala language design.

Assume that a `Range` value carries these two pieces of data (this is a slight simplification of the real definition in Scala):

```
val start: Int  // initial value of sequence
val end: Int    // final value of sequence
```

Sketch how to implement an iterator class for a `Range` that can be used as shown at the bottom of slide 21. Describe the data fields of the iterator, how it is constructed given a particulary `Range r`, and the implementation of the `hasNext()` and `next()` methods. Don't worry about using legal Scala syntax; pseudo-code is fine. (Note: In real Scala, the `hasNext` method is defined as a *parameterless method*, so it must be called without parentheses; see `https://www.artima.com/pins1ed/composition-and-inheritance.html#10.3`. So if you do want to try writing real Scala code here, remember to omit the parentheses on this call when testing, or you'll get a syntax error.)