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. To read and write the abstract syntax of different languages, we will use s-expressions, as described in Lecture 1b Slides 29-31. Like any other language, we can specify both a concrete and abstract syntax for s-expressions. Here is the concrete syntax for s-expressions used in the lab assignments:

```
sexpr ::= '(' slist ')' | symbol | number | string
slist ::= sexpr | sexpr slist
```

where the terminals (each carrying the obvious attribute) are as follows:

- **number** is an (possibly signed) decimal integer literal
- **string** is a string literal delimited by double quotes ("")
- **symbol** is a string of one or more printing characters excluding (,), and " that is not a number

In addition, comments enclosed in matching curly braces ({})) are permitted anywhere; this is not shown in the grammar.

Convince yourself that this grammar is unambiguous. It is also so simple that it can almost serve directly as the abstract syntax for s-expressions—but it is still not quite ideal. Here is the tree grammar for a better possible abstract syntax:

```
SList :  SExpr  →  SExpr₁ . . . SExprₙ
SSym :  SExpr  →  (string)
SNum :  SExpr  →  (int)
SString :  SExpr  →  (string)
```

Draw a parse tree and an AST based on these grammars for a simple example at the bottom of slide 30.

2. Consider compiling the following case statement, where \( e \) is an integer-valued expression and the \( sᵢ \) are arbitrary statements.

```
case e of
  4 : s4
  17 : s17
  13 : s13
  99 : s99
  2 : s2
  88 : s88
  12 : s12
  default: sd
end
```
Lecture 2a Slide 16 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:

```java
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”.

```java
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 such an instruction; this led to FORTRAN’s having a similar operator 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 Lecture 2a Slide 23, 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):

```scala
val start: Int   // initial value of sequence
val end: Int     // final value of sequence
```
Sketch how to implement an iterator object for a Range that can be used as shown at the bottom of Slide 23. Describe the data fields of the iterator, how it is constructed given a particular 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 parameter-less method, so it must be called without parentheses; see https://www.artima.com/pinsled/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.)