**1. Define an EBNF and abstract syntax for adding record structures to Clite. The EBNF for a structure reference should use the 'dot' notation discussed in Chapter 5. The concrete and abstract syntax of Declaration, Expression, and Assignment should be modified.**

Concrete syntax changes:
declaration ::=  --ordinary declaration stuff-- | structDeclaration  “;”
structDeclaration ::= "struct" "{" declarations "}" identifier { “,” identifier }
type ::= "int" | "float" | "char" | "bool" | struct
fieldident ::= identifier "." identifier
identifier ::= letter { letter | digit } | fieldident

Abstract syntax changes (here :myvar represents a variable myvar of the preceding type):
Struct = String :typename, declarations :fields
Type = "int" | "float" | "char" | "bool" | String :structName
FieldRef = VariableRef :struct, String :fieldId
VariableRef = Variable | ArrayRef | FieldRef

**2. Expand the type system to determine when a Record reference is well-typed. Make sure to consider both an occurrence on the left hand side and occurrences as expressions.**

A record reference is a special sort of variable reference, essentially. So the addition to the type rules would be essentially:
A Record Reference is valid if the variable is valid and the field appears in the struct’s typemap.

**3. Using your definition from the previous problem, design an algorithm that determines whether two records are type equivalent. Note that you should consider name equivalence (ie two records are only equal if they have the same type name) vs structural equivalence (two records are the same if they have the same structure.)**

Struct a is never equal to struct b if they have a different number of fields.

Under name equivalence:
sort the fields of structs a and b by name.
for i from 1 to the number of fields, if field i in a does not have the same name and type as field i in b the two are not equal.

Under structural equivalence:
sort the fields of structs a and b by the name of that field’s type.
for i from 1 to the number of fields, if field i in a does not have the same type as field i in b the two are not equal.

**4. Expand Type Rule 6.2 for Declarations so that it defines the requirement that the type of each variable be taken from a small set of available types.**

Rule 6.2.1: All declared variables must have a name and a type. The name must be unique, and the type must be int, bool, char, float.

**5. In the above problem, consider how Type Rule 6.2 must be expanded in order to include your Record type.**

Rule 6.2.2: All declared variables must have a name and a type. The name must be unique, and the type must be int, bool, char, float, or a previously defined struct type.

**6. Add a put() statement to Clite, where put() takes an expression as an argument and writes the value of the expression to the standard out (stdout). For this new statement:**
**(a) Define the EBNF and the abstract syntax.**
**(b) Modify Type Rule 6.4 to define the necessary validity checks**
**(c) Modify the function V for statements to implement the type rules.**

putStatement ::= “put(“ expression “);”

a put statement is valid if it’s expression is valid and has type Int, Float, Bool, or Char

```
public static void V( Statement s, TypeMap ™){

…

        if( s instanceof Put ){
                check( typeOf(s.expression).equals(Type.INT) ||
                        typeOf(s.expression).equals(Type.FLOAT) ||
                        typeOf(s.expression).equals(Type.CHAR) ||
                        typeOf(s.expression).equals(Type.BOOL) ,
                        “Cannot put() expression: invalid type));
        }
```

**7. Consider how to allow type coercion between two record definitions. You will have to decide the rules that allow type coercions between record definitions that follow the same philosophy as widening and narrowing for numeric coercions. Formalize your rules and give an example of both a type correct use and a type error.**

Structs should only be coercible to struct types which are name-equivalent. This prevents the problem of deciding what values go in what field, and prevents loss of data from fields that are lost in a generalizing or narrowing (same thing) conversion. It might be reasonable to allow assignment to structs whose fields are equivalent to a strict subset of the fields in the left-hand type.

**8. (a) How does Java define the numerical idea of infinity? (You should look at the Java language specification by Gosling et al, 2012 online.) (Here is a link to the latest: [http://docs.oracle.com/javase/specs/jls/se7/html/index.html](http://docs.oracle.com/javase/specs/jls/se7/html/index.html)).**

positive and negative infinity are values of each of the floating point types. They compare > and < all other floating point values, and equal to 1.0/0.0 and 1.0/-0.0, respectively.

**(b) Looking at the specification in the Java Language Definition (Arnold and Gosling, 1998), can you explain in plain English the meaning of the statement i = 3=j for all possible values of j including 0? (Here is a link to the pdf: docs.oracle.com/javase/specs/jls/se5.0/jls3.pdf)**

It doesn't mean anything. Assignment operators in Java are right-associative, meaning that the above expression is essentially the same as i = (3 = j). There are two problems with this. The first expression to be evaluated is (3 = j), which is syntactically and semantically invalid in Java--it actually just doesn't mean anything. It also doesn't have a value, which is the second problem; trying to assign the result of an assignment statement is also meaningless, because assignment statements don't have a value.