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 AS of Declaration, Expression, and Assignment should be modified.

```
Struct employeeType {
    int id;
    char name[26];
    int age;
    float salary;
    char debt;
};
Struct employeeType employee;
```

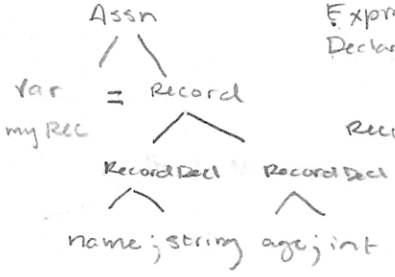
```
dot -> employee `dot` age = 45;

dot :: Structure -> Field -> a -> Structure
dot str fld n
    | not (elem fld str) = error "field not in"
                           ++ str ++ "."
    | otherwise = Structure {fld = n}
```

```
type field     = a
data structure = {[field]}
Record      -> {RecordDecl}+
Assignment  -> Identifier = Expression |
               Identifier = Record
Expression  -> VarRef | Value | Bin | Un | Rec
Declaration -> VariableDecl | ArrayDecl
               RecordDecl
RecordDecl  -> Variable V; Type t
```

Here my tree would produce

- Assignment
  - -> VariableRef
    - -> Variable
      - -> myRecord
        - -> Declaration
          - -> VariableDecl
            - v1
            - int
        - -> Declaration, etc.



Here I altered Assignment to compensate for a different kind of expression. Identifier {[RecordDecl]}. I'd have to modify/add a keyword 'Record' to distinguish from a normal expression.

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.

```
public static void V (Program p) {
    V(p.decpart);
    V(p.body, typing(p.decpart)); }
```

ex.

```
Record myRec {
    int one;
    int one;
}
```
} error, duplicate field names

```
Record myRec{
    String Kurt;
    String Medley;
}
```
-> error, duplicate record declarations

Type Rule K_1: all records must have unique names.
Type Rule K_2: All record's declared variables must have unique names
Type Rule K_3: field assignment values must match their declared variable's types according to the type map.

```
Record myRec {
    int one = 1;
    int two = 2.0;
}
```
→ No coercion defined by the type system error

```
Record myRec {
    int one;
}
myRec.one = 1.5;
```
→ error, this assignment produces an error as the type system does not allow type coercion

3. I will design an algorithm that determines structural equivalence

1. A record is valid if there is no other record with the same name.
2. Two records are equivalent when the following are true:
   (a) The length of RecordDecl are equal.
   (b) The types of R1 match R2 based on the sorted enumeration of [int, float, char, bool] being 0..4.
   (R1 types [0, 2, 0, 0] ≠ R2 types [1, 2, 0, 0])
   (c) The values of each field in R1 must match the values of each field in R2

```
Begin: R1; R2
    if (R1.getName() == R2.getName())
        error "Records cannot have the same name."
```
This is unnecessary as it would be caught at compile time because of the type system.

```
    if (R1.RecordDecl.length() == R2.RecordDecl.length())
        for (int i=0; i < R1.RecordDecl.length()-1; i++)
            for (int j=0; j < R2.RecordDecl.length()-1; j++)
                t1 = R1.getType();
                t2 = R2.getType();
                check (t1 = t2);
    END: Equivalent
END: Not Equivalent
```

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.

pseudo code

```
public static void V (Declarations d) {
  for (int i=0; i<d.size()-1 ; i++)
    for (int j=i+1; j<d.size()-1 ; j++) {
      Declaration di = d.get(i); Type type1 = di.getType();
      Declaration dj = d.get(j); Type type2 = dj.getType();
      if ( type1 == Type.FLOAT | Type.INT | Type.CHAR | Type.BOOL)
        if (type2 " " ...)
        check( !(di.v.equals(dj.v)),
            "duplicate declaration:" + dj.v);
      else
      S.O.S." Type not supported";
```

5. Consider how Type Rule 6.2 must be expanded to include the Record type
The type rule for duplicate names would be type rules K_1 and K_2 concatted onto the existing rule definition.
++ All records must have unique names
++ A particular record's fields must have unique names.
If a variable is not a reference to a Record, it may have the same name, as it is considered a different element.

6. Add a put() statement to C lite, where put() takes an expression as an argument and writes the value of the expression to the standard out (stdout)

CS
Statement → ; | Block | Assignment | IfStatement | WhileStatement | Put
AS
Statement = Skip | Block | Assignment | IfStatement | whileStatement | Put
Put = Expression

Following the definitions of Type Rule 6.4
A Statement is valid with respect to the program's type map if it satisfies the following constraints
{ 1..5 } ++ 6. A Put statement is valid if all of its expressions are valid.

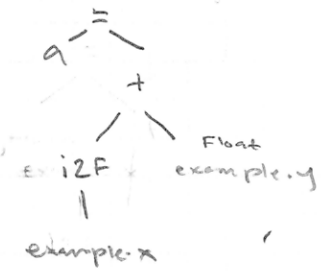
```
public static void V(Statement s, TypeMap tm)
    if (s instanceof Put)
        Put e = (Expression) e;
        V(e, TypeMap tm);
    ... Rest of def pg. 142
```

```
void put (Expression e) {
    V(e, TypeMap tm)
    SOS(e);
```

sample
```
int main() {
  char c;
  for(c='A'; c<='Z'; c++)
    putchar(c);
  return 0;
```

7. Type coercion between record field definitions would work following the Transformer for expressions.

```
Recod myRec {
    int x = 5;
    float y = 4.0;
}
```



This would widen my int and save information loss.

```
Record myRec example;
a = example.X + example.y
```

If a record field has the type char, int, it may be widened to Float
" " char, it may be widened to int

Type error
char a = exm.x + 'c' → the explicit declaration of char a makes this a type error
Type correct
float a = exm.y + 'c' → The unary ops c2i and i2f will allow this to be type correct

8. It would appear that Java has a sort of unuseful representation of Infinity under Double and Float types

```
S.O.S(Double.POSITIVE_INFINITY);
"                ", NEGATIVE_INFINITY);  which print

> Infinity
> -Infinity
```

9. Integer division rounds toward 0. The quotient produced for operands n and d that are integers after binary numeric promotion is an int value q whose magnitude is as large as possible while satisfying $|d \cdot q| \leq |n|$

If either operand is 0, the result is NaN.
" " Infinity, NaN