Programming Language Design

8.1, 8.4, 8.7, 8.16, 8.21

8.1

(a)

The type checker would need a type rule under the Unary operators to check that the target has type float, int, or char. The type transformer would treat transformations in the same way that it would treat x = x+1. You could also parse it as syntactic sugar for an assignment statement, and it would not add anything new to the AST, but it could not return a special value this way.

(b)

M : PreIncrement x State → Value x State
M: PostIncrement x State → Value x State
M(PreIncrement p, State state) = <p.target, state `onion` {<p.target, M(p.source, state)}>
M(PostIncrement p, State state) = <M(p.source, state), state `onion` {<p.target, M(p.source, state>}>
Similarly for pre and post decrement.

(c)

```
// I assume this is just for concept, there are probably syntax errors
(Value, State) M (PreIncrement p, State s) {
        StaticTypeCheck.check(p.target == type.float || p.target == type.int || p.target == type.char,
                                        “cannot increment this value”);
        return (p.target, state.onion(p.target, M(p.source, state));
}
```

Similarly for the others

(d)

Because of side effects? All assignments would have to return Value x State. Seems like no biggie to me.

8.4

(a)

It would simply be added to MulOp:
MulOp → * | / | %

(b)

Add it to:
ArithmeticOp → + | - | / | * | %

(c)

Add a Modulus token type and the term() function in the parser would not need to be changed. Add the MODULUS operator field to the operator type class at the bottom of the AST file.

(d)

Add the following check in applyBinary:

```
if (op.val.equals(Operator.MODULUS))
        return new IntValue(v1.intvalue() % v2.intvalue());
```

8.7

- It seems to me that line and column information would have to begin with the lexer. The next function could increment a line counter when it encounters '\n' and it would increment the column counter for all other characters.
- Each token object would then have a line and column field within it.
- The parser could then use those fields to return parse errors with lines and columns. It would also use that data to return an Abstract Syntax Tree where each node object also contains line/column fields. Each AST constructor could have line/column parameters that could be extracted from the fields in the respective tokens.
- Now that the AST contains line and column fields for each of its nodes, we can simply extract those fields when we call StaticTypeCheck

8.16

I am not sure what the author means by “show how it is derived”
(a) Rule 8.7. The expression is a binary. term1 is a binary, whose own terms are values. term2 is a value. It returns a value using rule 8.8 which is -231
(b) Rule 8.7. The expression is various binaries embedded in one another. Using rule 8.8 on each term, the expression returns the value -1
(c) Rule 8.7.1, it simply returns the value itself.

8.21

I did this already for exercise 1. I actually find denotational semantics more clear and readable than the author's english language rules.