Problem 6.1

Theorem 1 *The algorithm is correct.*

Proof.

Basis: The start state is trivially reachable from the start state.

Induction: Assume that we are currently some state A that is reachable from the start state. Consider a possible production on A, namely $A\alpha$. This production. This production would lead us to a set of symbols. We can label all of these symbols reachable as they are reachable from A which is reachable from the start state by the inductive hypthesis. Furthermore, since all symbols that we can reach from A are reachable from the start state, no state that we can visit will be erroneously marked as unreachable. Thus, if A is reachable from the start state, the symbols in $A \to \alpha$ are also reachable.

Because no reachable state will be erroneously marked as unreachable and because each reachable state will be marked as reachable, this algorithm is correct.
■

Theorem 2 *The algorithm is complete.*

Proof.

Basis: Assume that the start state is the only state. This state is trivially reachable and will not be removed, satisfying our basis.

Induction: Assume that we are at some state A that is reachable from the start state. Because this state is reachable, it will not be removed. All productions from A, namely $A \to \alpha$, lead to symbols which may or may not have been visited before. In the case where they have been visited before, we do nothing. In the case where they haven't been visited, we mark that state as reachable from the start state. We can conclude that this because it was immediately reachable from A, which is reachable from the start state by the inductive hypthesis.

Because every state that we were able to reach from the start state has been marked as reachable, we know that those states that are not marked as reachable were states that we did not encounter when taking productions beginning at the start state. Thus, we can conclude that the algorithm is correct and will only remove unreachable states and will not remove reachable states.
■

Problem 6.2

```
import java.io.*;

class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void E() throws IOException {
        System.out.println("E -> TR");
        T();

        for(;;) {

            if( lookahead == '+' ) {
                System.out.println("R -> +TR");
                match('+');
                T();
                continue;
            } else if( lookahead == '-' ) {
                System.out.println("R -> -TR");
                match('-');
                T();
                continue;
            }

            break;
        }

        System.out.println("T -> epsilon");
    }

    void T() throws IOException {
        if( Character.isDigit((char)lookahead) ) {
            System.out.printf("T -> %c\n", (char)lookahead);
            match(lookahead);
        } else throw new Error("syntax error");
    }

    void match(int t) throws IOException {
        if( lookahead == t )
        lookahead = System.in.read();
        else throw new Error("syntax error");
    }
}
```

```
public class Descent {
    public static void main(String[] args) throws IOException {
        Parser parse = new Parser();
        parse.E();
    }
}
```

Example output

Problem 6.3

First, some example output:

9-3*2

```
E -> TR
T -> 9
R -> A
R -> -TR
T -> 3
R -> *TR
T -> 2
R -> A
A -> epsilon
A -> epsilon
```

```
import java.io.*;

class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void E() throws IOException {

        if( lookahead == '(' ) {
            System.out.println("E -> (E)");
            match('(');
            E();
            match(')');
        } else {
            System.out.println("E -> TR");
            T();
```

```
            for(;;) {

                if( lookahead == '*' ) {
                    System.out.println("R -> *TR");
                    match('*');
                    T();
                    continue;
                } else if( lookahead == '/' ) {
                    System.out.println("R -> /TR");
                    match('/');
                    T();
                    continue;
                }

                break;
            }

            System.out.println("R -> A");
            A();
        }
    }

    void A() throws IOException {
        for(;;) {
            if( lookahead == '+' ) {
                System.out.println("R -> +TR");
                match('+');
                T();
                continue;
            } else if( lookahead == '-' ) {
                System.out.println("R -> -TR");
                match('-');
                T();
                continue;
            }

            break;
        }

        System.out.println("A -> epsilon" );
    }

    void T() throws IOException {
        if( Character.isDigit((char)lookahead) ) {
            System.out.printf("T -> %c\n", (char)lookahead);
```

```
            match(lookahead);
        } else if (lookahead == '(' ) {
            match('(');
            E();
            match(')');
        } else {
            throw new Error("syntax error");
        }
    }

    void match(int t) throws IOException {
        if( lookahead == t )
            lookahead = System.in.read();
        else throw new Error("syntax error");
    }
}


public class Descent {
    public static void main(String[] args) throws IOException {
        Parser parse = new Parser();
        parse.E();
    }
}
```