1) Develop a formal definition of the scanner algorithm from the lexical structure for our Itty Bitty Teaching Language (IBTL).
Step 1) Accept input string of undefined Length
Step 3) Move through the string one character at a time.
Step 4) If the character is a (or a) remove it.
Step 5) Check the input character against all Token Types.
Step 6) Keep track of accepted Token Types in a List
Step 7) Repeat process until no Automatons accept it.
Step 8) Order the list of accepted Token Types by key and value
Step 9) Select the most appropriate key.
Step 10) Store the token in a list of all tokens and go back to step 1)
    and repeat process until the input string is all read in.

2) Develop a scanner algorithm from a formal definition.

```
        static void Main(string[] args)
                {
                        Scanner scanner = new Scanner();
                        List<Token> tokenList = new List<Token>();

                        string fileContents = GetFileContents("test.txt");
                        while(fileContents.Any()) {
                                var token = scanner.ParseToken(ref fileContents);
                                tokenList.Add(token);
                                scanner.RemoveTokenFromBeginning(ref fileContents, token);
                        }

                        Console.WriteLine(
                                string.Join("",
                                        tokenList.Select(item => string.Format("{0}\t:\t{1}\n", item.Key,
item.Type))));
                        Console.ReadLine();
                }

                public Token ParseToken(ref string s)
                {
                        var listOfPossibleTypes = new List<KeyValuePair<TokenType, int>>(10);

                        var sb = new StringBuilder(s.First().ToString());
                        int posInString = 0;
                        int tokenLength = 0;
                        bool automatonAcceptFlag = false;

                        do {

                                if (sb[0] == '(') {
                                        automatonAcceptFlag = true;
                                        RemoveBeginningParens(ref s, ref sb, posInString);
                                        continue;
                                }
                                else if (sb[sb.Length - 1] == ')') {
                                        automatonAcceptFlag = true;
                                        RemoveEndingParens(ref s, ref sb, posInString);
                                        continue;
                                }

                                automatonAcceptFlag = false;

                                if (OperatorAutomaton.Parse(sb.ToString())) {
                                        listOfPossibleTypes.Add(new KeyValuePair<TokenType,
int>(TokenType.Operator, tokenLength));
                                        automatonAcceptFlag = true;
                                }
                                if (KeywordAutomaton.Parse(sb.ToString())) {
                                        listOfPossibleTypes.Add(new KeyValuePair<TokenType,
int>(TokenType.Keyword, tokenLength));
                                        automatonAcceptFlag = true;
                                }
                                if (BooleanAutomaton.Parse(sb.ToString())) {
                                        listOfPossibleTypes.Add(new KeyValuePair<TokenType,
int>(TokenType.Boolean, tokenLength));
                                        automatonAcceptFlag = true;
                                }
                                if (RealAutomaton.Parse(sb.ToString())) {
                                        listOfPossibleTypes.Add(new KeyValuePair<TokenType,
int>(TokenType.Real, tokenLength));
                                        automatonAcceptFlag = true;
                                }
                                if (IntegerAutomaton.Parse(sb.ToString())) {
                                        listOfPossibleTypes.Add(new KeyValuePair<TokenType,
int>(TokenType.Integer, tokenLength));
                                        automatonAcceptFlag = true;
                                }
                                if (StringAutomaton.Parse(sb.ToString())) {
                                        listOfPossibleTypes.Add(new KeyValuePair<TokenType,
int>(TokenType.String, tokenLength));
                                        automatonAcceptFlag = true;
                                }
                                if (IdentifierAutomaton.Parse(sb.ToString())) {
                                        listOfPossibleTypes.Add(new KeyValuePair<TokenType,
int>(TokenType.Identifier, tokenLength));
                                        automatonAcceptFlag = true;
                                }

                                if (!automatonAcceptFlag &&
StringAutomaton.ParsePartialString(sb.ToString())) {

                                        automatonAcceptFlag = true;
```

```
                        }

                        if (automatonAcceptFlag) {

                                tokenLength++;
                                posInString++;
               if (s.Length != posInString)
               {
                  sb.Append(s.ElementAt(posInString));
               }
               else
               {
                  sb.Append(" ");
               }
                        }

                    } while (automatonAcceptFlag);



                    // order the possible types by the key
                    var orderedByKey = listOfPossibleTypes
                        .OrderBy(x => x.Key);

                    // grab the longest-length value with the highest priority type
                    var result = orderedByKey
                        .Where(x => x.Key == orderedByKey.First().Key)
                        .OrderByDescending(x => x.Value)
                        .First();

                    // construct the token with result
                    return new Token(
                        sb.ToString().Substring(0, result.Value + 1),
                        result.Key);

                }
```

3) Develop a data structure for the tokens of the language.

The data structure for the tokens of the language will be a list of Tokens. Tokens will be a key, value pair with the key being the input string and the value being of TokenType where TokenType is one of our types of Tokens.

TokenTypes = Operator, Keyword, Real, Integer, Boolean, String, Identifier

4) Develop a test driver for the developed program .

Tests have been written in a seperate project called CompilerTests. They tests the Parser for the various TokenTypes. Tests have also been written for the Scanner. Strings are fed into the scanner and compared against what the tokens key and type should be.

5) Test the resulting program for correctness based on the formal definition.

We decided upon waiting to develop our symbol table until we start work on our parser.