Getting Started on the Compiler Project

Your first step is to pick an implementation language. Here are the modules you will write:

- A *scanner*, which reads a text file containing a BPL program. It reads this file character by character. The scanner has a variable that holds the current *token* (one symbol of the program, such as a variable name, a keyword such as *while*, a grammatical symbol such as a semicolon, a token representing the end of the input file, etc.). The scanner also has a function getNextToken( ) which reads the file until it can update the token variable.

- A *parser*, which uses the grammar and the getNextToken() function to build a tree that represents the program.

- A *type-checker*, which makes 2 passes through the parse tree. A top-down pass assigns to every identifier a pointer back to the place in the parse tree where that identifier was declared. A bottom-up pass then assigns a type to every expression.

- A code generator, which makes a pass through the parse tree and builds up a file of assembly language code representing the program.

The language you choose for implementing your compiler needs to allow you to read a text file character by character, output a text file, and work with tree-like structures and pointers.

If you write your compiler in Java and include only enough documentation to remind you of what the code does, the whole project is about 3000 lines (roughly 60 pages) of code. I suggest using the implementation language you are most comfortable with. I do not suggest using this project to learn a new language, though that has been done successfully in the past.

Reasonable choices for implementation languages include Java, Python, C, and C++.

You are welcome to use other languages, but if you choose to do so, I want to know about it THIS WEEK.

Each of the assignments describes the module you need to write for the next stage of the compiler, and also an application program that will allow me to see that your module works.

I would prefer that you set this up so that I type

```
bpl foo.bpl
```

(where foo.bpl is a text file with bpl code) and the appropriate application program is run. With the final module

```
bpl foo.bpl
```

should produce an assembly-language file foo.s which I can assemble and run with

```
as foo.s
a.out
```