Assignment 4

Tue Mar 1 (125 points)

For this assignment, we will use ANTLR to create a configuration file parser. Early in the semester, I asked you to run a command-line tool to tell your Git installation who you are:

```
git config --global user.name "YOUR NAME"
git config --global user.email "YOUR.ADDRESS@EXAMPLE.COM"
```

That information is stored with other configuration settings in a file `~/.gitconfig`, which looks like this:

```
[user]
    name = YOUR NAME
    email = YOUR.ADDRESS@EXAMPLE.COM
[color]
    diff = auto
[core]
    autocrlf = input
    mergeoptions = --no-edit
[push]
    default = matching
```

This is a fairly common configuration file format, sometimes called an "INI file." The format isn't really standardized, so different tools have slightly different capabilities and requirements.

We want to create a program that can read such files and make their settings available as a `HashMap` (dictionary) data structure. In the map, the section heading and variable name are joined with a dot, so that the key `push.default` is set to the value `matching`:

```
    HashMap<String,String> conf = readConf(".gitconfig");
    conf.get("push.default");  // returns "matching"
    conf.get("color.diff");    // returns "auto"
```

The basic structure of this `readConf` function is provided for you, because it relies on the ANTLR grammar and a visitor to do the heavy lifting:

```
    HashMap<String,String> readConf(String filename) throws IOException {
        // Instantiate the lexer and parser generated by Config.g4
        ANTLRInputStream input = new ANTLRInputStream(new FileReader(filename));
        ConfigLexer lexer = new ConfigLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ConfigParser parser = new ConfigParser(tokens);
        // Call our syntaxError method when there are parse errors
        parser.addErrorListener(this);
        // Instantiate the visitor
        MapBuilder builder = new MapBuilder();
        // Do the parse and then visit the tree with the MapBuilder.
        parser.top().accept(builder);
        // Return the HashMap containing the configuration data.
        return builder.map;
    }
```

So it will be your job to write the grammar and lexer rules in `Config.g4`, and to code the `MapBuilder` as a subclass of the ANTLR-generated `ConfigBaseVisitor`.

Syntax and examples

A configuration file consists of one or more **sections,** each of which contains one or more **variable bindings.** Sections begin with a name in brackets, on a line by itself:

```
[server-data2]
```

Both section and variable names are composed of one or more alphabetic characters, digits, the hyphen ‘-’, or the underscore ‘_’.

White-space on either side of an equal sign ‘=’ in a variable binding should be ignored. That’s why `user.name` in the `.gitconfig` example:

```
    name = YOUR NAME
```

has the value `YOUR␣NAME` (where we use ‘␣’ to represent a space character) and **not** `␣YOUR␣NAME`. Section and variable bindings may also be indented any number of spaces:

```
[user]
  name=Carly
    id=22419
  [workspace]
    geometry  =  145x92+8+10
```

Importantly, spaces **within** a value are preserved, so in this file:

```
[message]
sig = Goodbye,   world   !
```

The value of `message.sig` should be `Goodbye,␣␣␣world␣␣␣!`.

Configuration files can have **comments,** which begin with a pound sign '#' and run to the end of the line. They appear, however, only on lines by themselves (possibly indented). They cannot be used on lines that introduce sections or contain variable bindings. Thus, the following file:

```
# My account info
[user]
name=ralph
password=secret#123
    # The end!
```

defines two variables, `user.name` and `user.password`. The value of `user.password` is `secret#123`. In other words, the `#123` should be interpreted as part of the value of the variable, **not** as an ignored comment.

Getting started

I provided a starting point in the assn4 directory in the cs664pub repository. A mostly-empty grammar file is in `src/main/antlr/Config.g4`, and there are three Java classes in `src/main/java`. To build it, open the `Config.g4` and use **Tools » Generate ANTLR Recognizer.** It will generate code into the gen/ directory. Then you should be able to build using **Build » Make Project.**

The `assn4` project contains a bunch of test cases in the `test/` directory. There are two types of files in there:

- `.conf` files are sample configuration files that your parser should be able to interpret.
- `.dat` files are binary data files that contain the correct map data for the corresponding configuration file.

The `ConfigTest` class contains a `main` method that invokes `runAllTests`. It looks at each configuration file in the `test/` directory, runs it through your parser and `MapBuilder`, and tests your result against the expected one. At first, the output of `ConfigTest` will be:

```
========= #1: 00-easy
line 1:0 token recognition error at: '['
line 1:1 token recognition error at: 'e'
line 1:2 token recognition error at: 'a'
[...]
  actual: {}
expected: {easy.name=Alice}
    FAIL
```

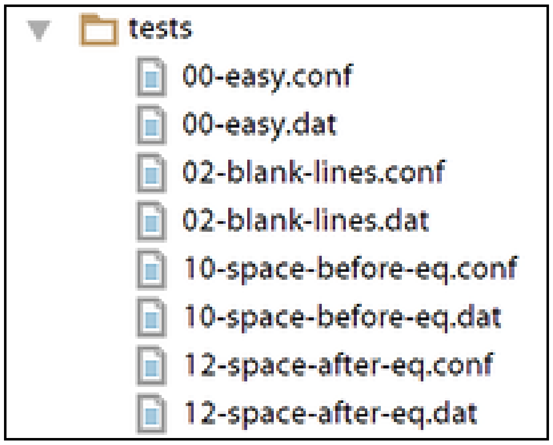

Figure 1: Test files

```
[...]
!!!!!!!!! 22 failures out of 22 tests
```

It's full of "token recognition" errors because we haven't given ANTLR any lexical rules that would allow it to recognize any characters in the files.

You can begin working on a grammar and testing it with the built-in ANTLR Preview panel. Once you get something partly working, regenerate the recognizers and rebuild. Some of the syntax errors should disappear, but your `MapBuilder` is still always returning the empty map.

Once you have eliminated all (or almost all) of the syntax errors, then work on implementing the couple of methods in `MapBuilder` that will populate the `map` field in response to visiting a variable binding. The syntax for adding a binding to the map is:

```
    map.put(key, value);
```

If you're not using IntelliJ, I provided a `build.gradle` file that will run ANTLR to generate the lexer, parser, and visitor code, and then build it with java. You can build and run the `ConfigTest` class just by typing:

```
gradle run
```

in the project directory.