ECE 449, Fall 2014
Project 1: Lexical Analysis

Initial Release Due: 09/12 (Fri.), by the end of the day
Final Release Due: 09/19 (Fri.), by the end of the day

1 Overview

While it is quite natural for human designers to understand a circuit specified in the EasyVL language in a textual file, the file on the disk contains no more than a stream of characters. Therefore, for a computer program to interpret this file as a circuit, it must be able to "understand" the EasyVL language by identifying the various parts, e.g. gates and wires, of the circuit from the file. This process is formally known as *compiling*. Though learning the general theory of compiling may take at least one semester, the EasyVL language is designed in a way such that we can extract components and interconnects from the textual file in two projects: lexical analysis followed by syntactic analysis.

2 Lexical Analysis

The purpose of lexical analysis is to convert a sequence of characters into a sequence of *tokens*. A tokens in a computer language is the counterpart of a word in a human languages. It is the minimal element in the language that has meanings. For example, let's look at the following EasyVL file **test.evl**, which describes a simple circuit consisting of a flip-flop (`evl_dff`) and a `not` gate.

```
// a comment
module top;

   wire s0, s1, clk;

   evl_clock(clk);

   evl_dff(s0, s1, clk);

   not(s1, s0);
```

```
endmodule
```

In this example, both `wire` and `s0` are examples of tokens since the former starts a list of wires and the later is the name of a wire. On the other hand, neither `0` nor `1` is a token since they don't mean anything in the context – they are part of the wire names.

There are three types of tokens defined in the EasyVL language as follows.

- **NAME**: tokens of this type should start with a letter or '`_`', and any following character should be among those characters or '`$`' or digits.
- **NUMBER**: tokens of this type is a sequence of digits.
- **SINGLE**: any character among '`(`', '`)`', '`[`', '`]`', '`:`', '`,`', and '`;`'.

Note that first, the longest match rule applies here so a token should contain as much character as possible, e.g. `s0` should be interpreted as one **NAME** token instead of a **NAME** token followed by a **NUMBER** token; second, comments (`//` to the end of the line) and spaces are removed during the process since they don't mean anything to the simulator.

The file **lex.cpp** provided in the initial project package is able to handle **NAME** and **SINGLE** tokens. We recommend that you don't change this file for the initial release of Project 1. The support for **NUMBER** tokens is then added at the final release.

3 Required Program Output

To allow simulating an arbitrary EasyVL file with your program, your program should take the name of the EasyVL file as the first command-line argument. After the file is processed to obtain tokens, the tokens should be stored into an output file so we can validate them by using our golden simulator. The name of the output file should be that of the EasyVL file concatenated with **.tokens**, e.g. the output file for **test.evl** should be **test.evl.tokens** . In the output file, tokens should appear in the same order as they do in the EasyVL file. Each token should occupy one line including its type followed by a space and then the token itself.

For example, the content of **test.evl.tokens** should be.

```
NAME module
NAME top
SINGLE ;
NAME wire
NAME s0
SINGLE ,
NAME s1
SINGLE ,
```

```
NAME clk
SINGLE ;
NAME evl_clock
SINGLE (
NAME clk
SINGLE )
SINGLE ;
NAME evl_dff
SINGLE (
NAME s0
SINGLE ,
NAME s1
SINGLE ,
NAME clk
SINGLE )
SINGLE ;
NAME not
SINGLE (
NAME s1
SINGLE ,
NAME s0
SINGLE )
SINGLE ;
NAME endmodule
```