

## Workshop 3

### Exercise 1: configuration and parsing

NOTE: there are many different ways to express lexical analysis and parsing. So for clarity, lexical analysis can also be expressed as a "Tokenizer" while a parser can also be expressed as a "syntax analyzer". In this exercise however, lexical analysis and syntax analysis will be the terms used.

Creating a file to configure the system in the previous workshops would usually, in any language, require parsing.

To read from and configure the system a form of syntax would be useful, for this JSON(JavaScript Object Notation) is good, as one of the advantages of json is that it is both easily readable by humans while being easy for computer programs to parse and generate.

The configuration file for the system could look like the json code in listing 1:

---

```
1 {  
2     "bitrate":int,  
3     "sampling_frequency":int  
4     "example_object":  
5     {  
6         "example_key1":"placeholder",  
7         "example_key2":"placeholder"  
8     }  
9 }
```

---

**Listing 1:** JSON syntax

Where a JSON library function in a language with OOP(Object Oriented Programming) would read and save the data in a JSON object to be read from.

parsing JSON is typically broken into two stages, lexical analysis and syntactic analysis:

#### Lexical analysis

The language is built around brackets, colons and commas. The brackets indicate the beginning of an array-like structure, where there are multiple entries, these can be expressed as data types as follows:

**BEGIN\_OBJECT({)** indicates the beginning of a JSON object, these can also be used inside another JSON object as shown in listing 1

**END\_OBJECT(})** indicates the ending of a JSON object

**BEGIN\_ARRAY([)** indicates the beginning of an array

**END\_ARRAY(])** indicates the ending of an array

**NULL** indicates a null value, this usually appears whenever a key without a value is found, this will throw an exception and usually end the parsing of JSON

**NUMBER** indicates a value of either integer or floating value

**STRING** indicates a string of characters, the name of a device for example

**BOOLEAN(true/false)** indicates a type which can either be true or false

**COMMA\_SEPERATOR(,)** seperates different key/value pairs from each other, the absense of one of these is always followed by an **END\_object**

**COLON\_SEPERATOR(:)** seperates a key from its value

When parsed, json in listing 1 goes through a lexical analyzer which reads the JSON file for each character of the above datatypes and structures them as a set of tokens. The set of tokens would resemble listing 2:

```
1 { , bitrate , : , int , , , sampling_frequency , : , int , , , example_object , : , { ,  
   example_key1 , : , placeholder , , , example_key2 , : , placeholder , } , }
```

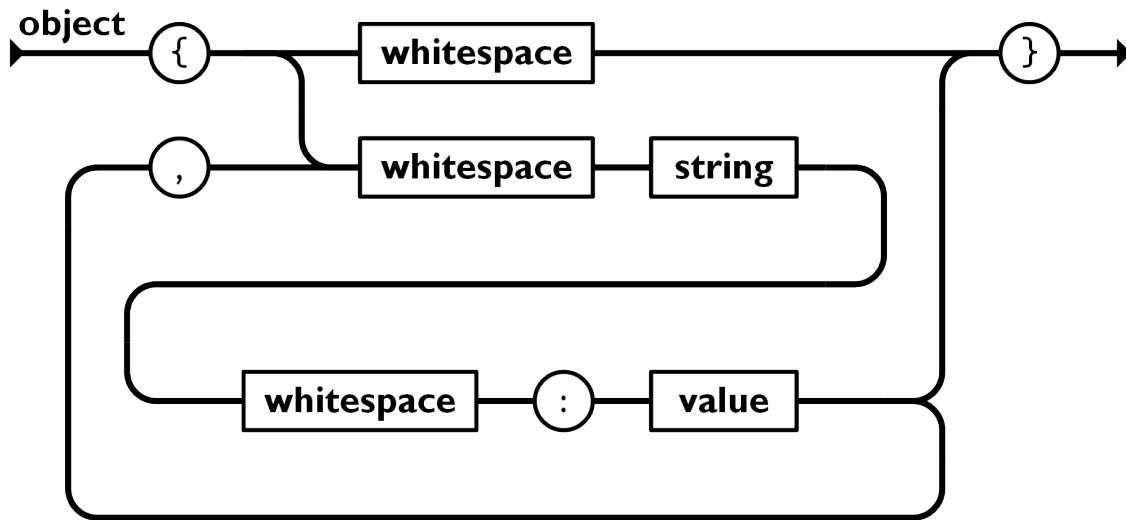
**Listing 2:** JSON syntax

Note that the lexical analyser only checks if the tokens are within one of the datatypes, the syntax analyser is responsible for checking the correct placement of the tokens. An example of this would be if the JSON file contained a key/value pair defined as: "STRING,NULL,;,NUMBER,," it would pass this pair on to the syntax analyser, in this case throwing an exception as described below.

Once the lexical analysis has finished tokenizing the json object it will continue on to the syntax analyser.

## Syntax analysis

Parsing JSON has to not be context-free, as it is very important that it throws an exception if a value is NULL for example. The purpose of passing the set of tokens through the syntax analyzer would be to match the tokens with grammar valid to the language.



**Figure 1:** JSON Object syntax, (source: <https://www.json.org/json-en.html>)

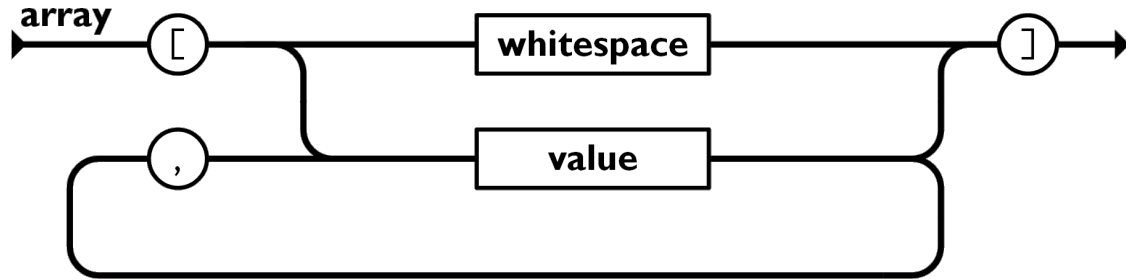
The JSON syntax analyser would essentially begin from the left-most token in the set of tokens and check if it matches the grammar of JSON, revealing that the only valid token to begin the JSON file with is the BEGIN\_OBJECT token. Then the syntax analyser would expect one of two types of tokens, either a STRING (the key of a key/value pair) token or an END\_OBJECT token, any other token would throw an exception. This syntax is illustrated in figure 1, note that for json to be valid there does not have to be whitespace in the file, it is simply just common to insert this for readability. This whitespace is ignored completely by the lexical analysis.

There are two different scenarios where the syntax analyser would have to parse a JSON object, when a JSON object is to be parsed, and a JSON array would have to be parsed:

### JSON object

The Syntax analyser will iterate through the set of tokens in the object and check if the datatype is correct beginning with the STRING (key) followed by the COLON\_SEPERATOR and a value. The value can be of data types BOOLEAN, STRING, NUMBER, JSON\_OBJECT or JSON\_ARRAY. These key/value pairs are separated by COMMA\_SEPERATORS except the last pair. After the last pair an END\_OBJECT token is always found, once the syntax analyzer is finished it should find a END\_OBJECT token.

## JSON array



**Figure 2:** JSON array type syntax (source: <https://www.json.org/json-en.html>)

When the syntax analyser encounters an array it will iterate through the set from the initial `BEGIN_ARRAY` token until it finds an `END_ARRAY` token, checking whether every odd token is of data type `BOOLEAN`, `STRING`, `NUMBER`, `JSON_OBJECT` or `JSON_ARRAY` and whether every even token is of type `COMMA_SEPERATOR`.

In both the case of a JSON object and a JSON array, if the space between the begin and end tokens is empty, the syntax analyser would simply just continue to the next key/value pair. Once the syntax analyser is finished the JSON object should have been created in the programming language it is parsed to, as the process of parsing json has finished.

## Database Configuration

JSON can also be interpreted as a database, as mentioned several times before it uses these key/value pairs, which is the basis of the key-value NOSQL database type. The configuration could also be saved in an SQL database as shown in table 1:

Config	Bitrate	Sampling_frequency	Setting #3	Setting #4
Config #1	some integer	some integer	etc	etc
Config #2	some integer	some integer	etc	etc

**Table 1:** SQL config

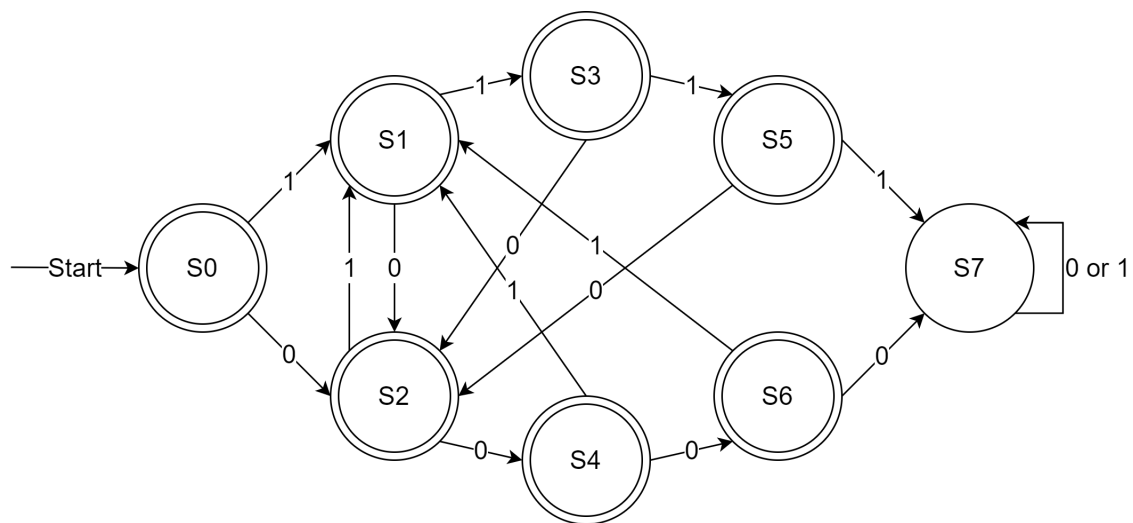
In SQL multiple configurations could be defined in the same table, which is practical for some applications where the configuration would change often or you have multiple identical systems, you could set them up to access a server with all the configuration stored in SQL. This would also enable you to configure each system remotely. The system in this exercise is not that complicated and configurable so setting up a SQL database would most likely take a lot more time than setting up a JSON file.

## Exercise 2: Regular Grammar

In this exercise it is desired to create a Finite State Automata (FSA) and a regular grammar for the language L which contains all sequences of zero and one that do not contain more than three consecutive zeroes or ones.

An example of a string of this language is "010011000111", and an example of a string outside the language is "0101111010". Furthermore, the empty string is also a member of the language L, since it fulfills its requirements.

### Language Representation



**Figure 3:** Finite state automata for accepting strings in the language L

Figure 3 shows a FSA that would only land on a final state if the string given to it is part of the language L. Otherwise, the automata will end in state 7, which is not a final state, meaning that the input is not part of the language L.

$S_0 \longrightarrow$	$aS_1$	$ $	$bS_2$	$ $	$\epsilon$
$S_1 \longrightarrow$	$aS_3$	$ $	$bS_2$	$ $	$\epsilon$
$S_2 \longrightarrow$	$aS_1$	$ $	$bS_4$	$ $	$\epsilon$
$S_3 \longrightarrow$	$aS_5$	$ $	$bS_2$	$ $	$\epsilon$
$S_4 \longrightarrow$	$aS_1$	$ $	$bS_6$	$ $	$\epsilon$
$S_5 \longrightarrow$			$bS_2$	$ $	$\epsilon$
$S_6 \longrightarrow$	$aS_1$			$ $	$\epsilon$
$a \longrightarrow$	1				
$b \longrightarrow$	0				

**Listing 3:** Regular Grammar for the FSA in figure 3

Listing 3 shows the regular grammar that represents the FSA in figure 3 and also represents the language L. This grammar is a right regular grammar, where a and b are terminal symbols representing 1 and 0 respectively, and  $S_0$  to  $S_6$  are non

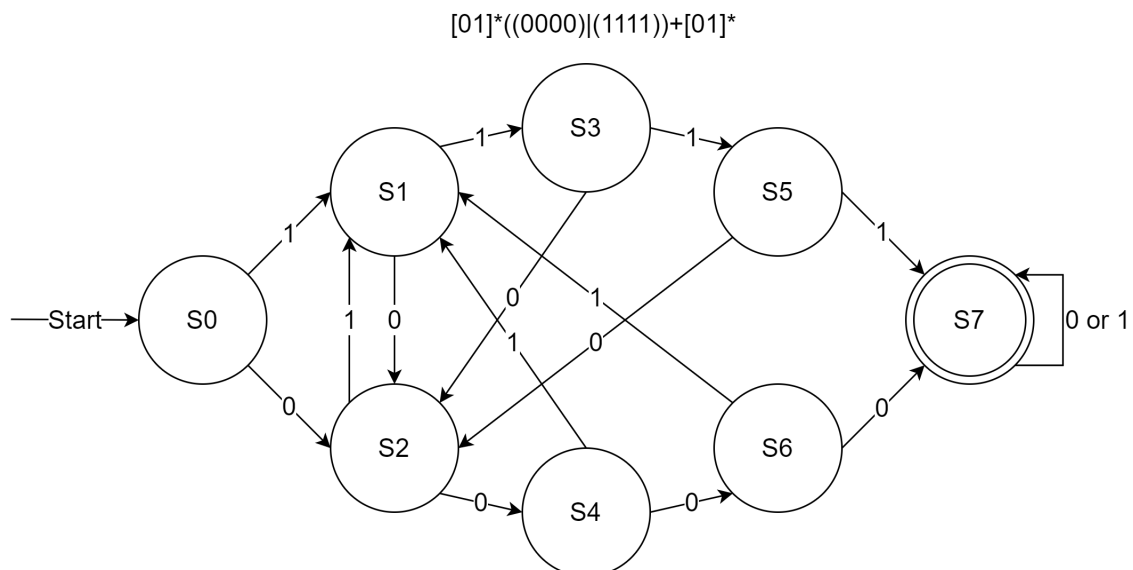
terminal symbols representing the states in the FSA in figure 3.

As with the FSA in figure 3 in the grammar in listing 3 each non terminal symbol can contain the terminal symbol 1 or 0, and then the non terminal symbol representing the state it goes to next.

However, in the regular grammar, each non terminal symbol can also contain the empty terminal symbol. This is because the strings in this language are not infinite, and will have to end. That is achieved by ending it with the empty string as a terminal symbol.

## Language FLEX code

Fast Lexical Analyzer Generator (FLEX) is a lexical analyzer that can be used scan strings or characters from user inputs or files, and perform code according to the strings or characters provided. It uses regular expressions to detect the desired input and run the code corresponding to it.



**Figure 4:** An FSA represent the compliment of the language in figure 3

In order to detect language L in FLEX a regular expression is needed. Building a regular expression using the regular grammar in listing 3 is hard, because it is corresponding FSA has many final states. On the other hand it would be easier to build a regular expression for the FSA corresponding to the compliment of the language L within the set of all strings built using "0" and "1".

Figure 4 shows that FSA. Which is the exact FSA as the one in figure 3, but with the final state being S7 instead. Above the FSA the regular expression representing it can be seen. This regular expression only accepts strings of "0" and "1" where there is at least one or more instances of "0000" or "1111".

```

1 %{
2 #include <stdio.h>
3 %}
4
5 %%
6
7 [01]*((0000)|(1111))+[01]*
8     {printf("Value: %s is not valid binary number\n",yytext);}
9 [01]*
10    {printf("Value: %s is a valid binary number\n", yytext);}
11 .
12    {printf("Value: %s is not a binary number\n", yytext);}
13 %%
14
15 main()
16 {
17     yylex();
18 }

```

**Listing 4:** FLEX Code for detecting strings in language L

The code in listing 4 is a typical FLEX code. It is separated by using `%%` into three sections.

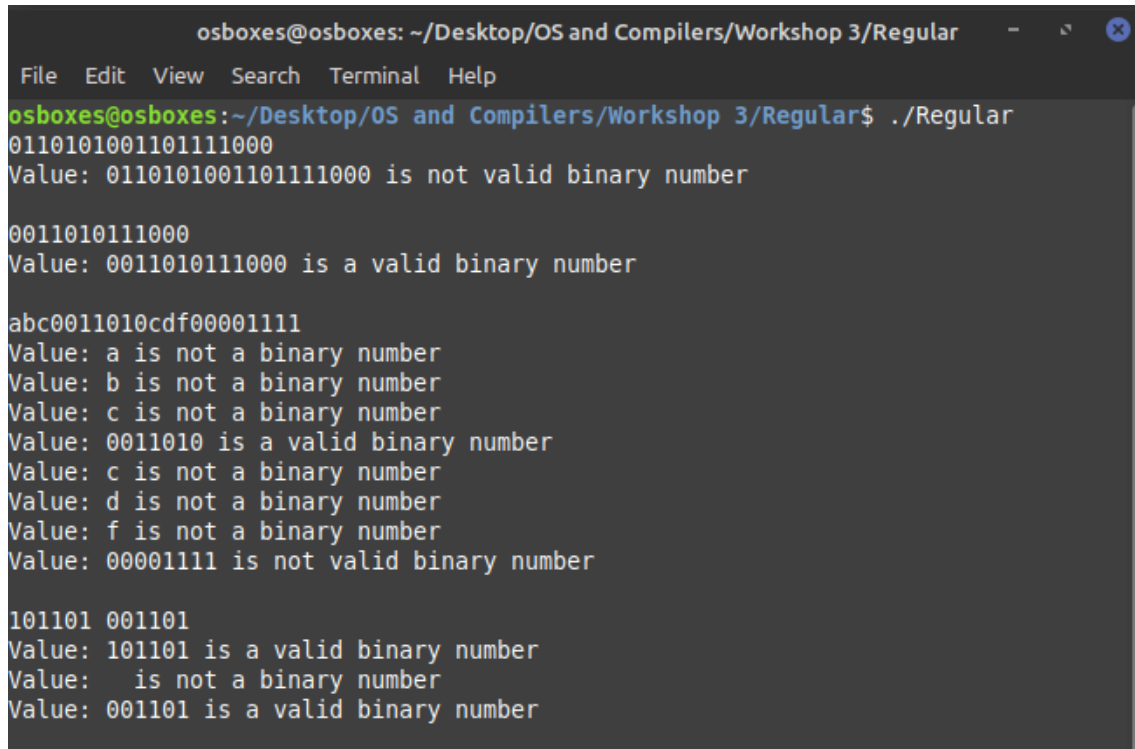
The first is the definitions section, where any constants are declared. The declarations inside the `%{ %}` are C code, and what comes after is assigning names to rules.

The second is the rules section, where the rules for the lexical analyzer are set. The rules are set using regular expressions, and then the code to be run if the rule is matched is set between two curly brackets.

The third section is the subroutines section, where any C functions are declared. The functions in here can be called in the rules section too. It is also in this section that the lex functions get called, such as `yylex()`; which is responsible for running the rules section.

The program in listing 4 prints a statement depending on the string inputted by the user. There are three cases specified in the rules section:

- First case is a binary string which is not in language L. In this case the program prints that it is not valid.
- The second case specifies any binary string. This corresponds to strings in the language L, as any binary string outside L would have already been caught by the first rule. In this case it is printed that it is a valid string.
- The third case, represented by a dot meaning any character, is if any character other than "0" and "1" are detected, since "0" and "1" would already be picked up by the second rule. In this case the program prints that it is not a valid binary number.



```
osboxes@osboxes: ~/Desktop/OS and Compilers/Workshop 3/Regular
File Edit View Search Terminal Help
osboxes@osboxes:~/Desktop/OS and Compilers/Workshop 3/Regular$ ./Regular
0110101001101111000
Value: 0110101001101111000 is not valid binary number

0011010111000
Value: 0011010111000 is a valid binary number

abc0011010cdf00001111
Value: a is not a binary number
Value: b is not a binary number
Value: c is not a binary number
Value: 0011010 is a valid binary number
Value: c is not a binary number
Value: d is not a binary number
Value: f is not a binary number
Value: 00001111 is not valid binary number

101101 001101
Value: 101101 is a valid binary number
Value:   is not a binary number
Value: 001101 is a valid binary number
```

**Figure 5:** Running the code in listing 4

Figure 5 shows some strings being inputted into the program. The first string is invalid because it contains four consecutive "1"s. The second string is a valid string. The third and fourth strings show an interesting thing. It is possible to trigger all three cases using one user input. The string "0000201" this will be treated as three separate strings of "0000", "2" and "01". This is because any character other than "0" and "1" are not part of the FSA making the first or second rule, so the program stops when it encounters one, and treats it alone.



## Exercise 3: Context Free Grammar

In this exercise it is desired to represent the language L, which includes all sequences of zeroes and ones, where the number of zeroes and ones are equal, using a context free grammar.

A	→	1A0
A	→	0A1
A	→	AA
A	→	ε

The grammar

**Listing 5:** Context free grammar for language L

shown in listing 5 is a context free grammar that represents all the strings in language L. The first two statements make sure that there is always an equal number of the terminal symbols 0 and 1. However, they are not enough, as they cannot represent strings such as "0110" or "10010011" that is why the third statement is needed. The last statement is to ensure that the string is not infinite.

The FLEX code to detect this language is shown in listing 6. In this code 2 integers are declared in the declaration section. Count is used to count the difference between zeros and ones. A negative count means there are more zeros and a positive count means there are more ones. A string is part of the language L only if the count is 0. Valid is used to check that only zeros or ones are entered. If any other character is entered, then the string is seen as invalid, and not part of L.

In this code the rules section is able to return a value. It will return a 1 to signal that the string is done, and a 0 to signal that the string is not done yet and more characters are yet to be lexed. It has the following six rules:

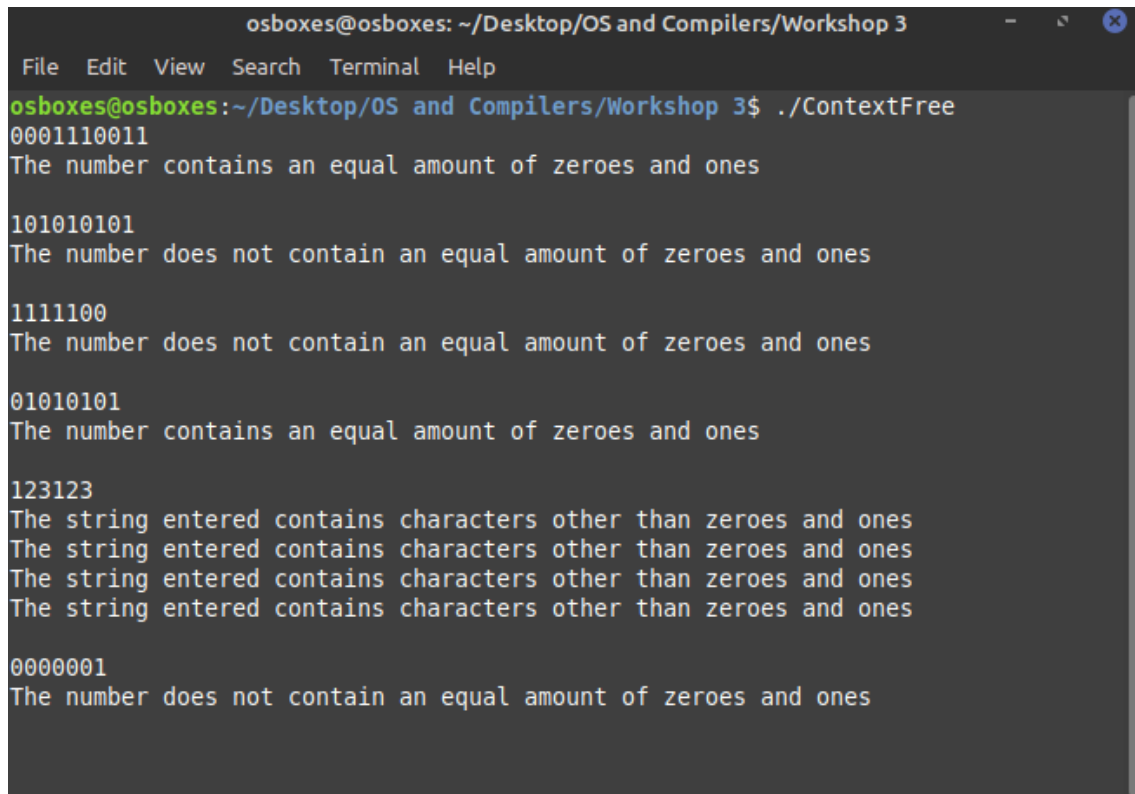
- end will return the value of END\_T, which is used to exit the while loop and exit the program
- 0\$ This will only trigger if a 0 comes at the end of a string. It will decrease the count by one and return a 1.
- 1\$ This will only trigger if a 1 comes at the end of a string. It will increase the count by one and return a 1.
- 0 This will detect a 0. It will decrease the count by one, and return a 0, as it was not at the end of a string.
- 1 This will detect a 1 and return a 0. It will increase the count by one, as it was not at the end of a string.
- . This will trigger on any other character. It will set the Valid value to 0 and return 1.

```
1 %{
2 #include <stdio.h>
3 #define END_T      258
4
5 int Count = 0;
6 int Valid = 1;
7 %}
8
9 %%
10
11 end      {return END_T;}
12 0$      {Count--; return 1;}
13 1$      {Count++; return 1;}
14 0       {Count--; return 0;}
15 1       {Count++; return 0;}
16 .       {Valid = 0; return 1;}
17
18 %%
19
20 int yywrap() {return 1;}
21
22 int
23 main()
24 {
25     int token;
26     while ((token = yylex()) != END_T)
27     {
28         if(token)
29         {
30             if(Valid)
31             {
32                 if(Count)
33                 {
34                     printf("Binary string not in L\n");
35                 }
36                 else
37                 {
38                     printf("Binary string in L\n");
39                 }
40             }
41             else
42             {
43                 printf("Not binary string\n");
44             }
45             Count = 0;
46             Valid = 1;
47         }
48     }
49 }
```

**Listing 6:** FLEX code for detecting strings in language L

The submodule section at the end will initialize first make the `yywrap()` function always return 1. This is to make sure that the program does not stop after reading the first user input. Then it defines the main function to be run.

In the main function a token is declared which takes the return value of the rules section. Then a while loop is run as long as the token is not `END_T`. Inside the while loop a statement will be printed depending on the values of `Count` and `Valid`, and then at the end the values of `Count` and `Valid` will be reset for the next string.



```
osboxes@osboxes: ~/Desktop/OS and Compilers/Workshop 3
File Edit View Search Terminal Help
osboxes@osboxes:~/Desktop/OS and Compilers/Workshop 3$ ./ContextFree
0001110011
The number contains an equal amount of zeroes and ones

101010101
The number does not contain an equal amount of zeroes and ones

1111100
The number does not contain an equal amount of zeroes and ones

01010101
The number contains an equal amount of zeroes and ones

123123
The string entered contains characters other than zeroes and ones
The string entered contains characters other than zeroes and ones
The string entered contains characters other than zeroes and ones
The string entered contains characters other than zeroes and ones

0000001
The number does not contain an equal amount of zeroes and ones
```

**Figure 6:** Running the code in listing 6

Figure 6 shows how the code handles the different strings. As can be seen in the third example whenever it encounters an invalid character it will print out a statement declaring it, and then try scanning the rest of the string alone.