Abstract

Introduction : In this section, we discuss what is a lexical analyzer and how it works. We also discuss here the typical use cases of this lexical analyzer.

Methodology: In this section, we discuss how we developed our lexer or lexical analyzer. We talk about Deterministic Finite Automata or DFA. And we used DFA to implement our lexical analyzer.

Results: In this section, we talk about for a input file what are the results for the program and how analyze our result.

Conclusion: In this section, we talk about what are the next steps we can do and how this analyzer could be extended.

Introduction:

In computer science, lexical analysis, lexing or tokenization is the process of converting a sequence of characters into a sequence of tokens. A program that performs lexical analysis may be termed a lexer, tokenizer, or scanner, though scanner is also a term for the first stage of a lexer.

Lexers are most often used for compilers, but can be used for other computer language tools, such as **prettyprinters** or **linters**. Lexing can be divided into two stages: the *scanning*, which segments the input string into syntactic units called *lexemes* and categorizes these into token classes; and the *evaluating*, which converts lexemes into processed values.

Lexers are generally quite simple, with most of the complexity deferred to the parser or semantic analysis phases, and can often be generated by a lexer generator, notably lex or derivatives. However, lexers can sometimes include some complexity, such as phrase structure processing to make input easier and simplify the parser, and may be written partly or fully by hand, either to support more features or for performance.

In this assessment we are assigned to make a Lexical Analyzer for Mk Programming Language. This particular lexical analyzer should be developed using C Programming Language.

Aims and Objectives:

1. Understand about Lexical Analysis and Lexer.

2. Understand about Deterministic Finite Automata.

3. How Deterministic Finite Automata and Lexer co operates together.

4. Vector Class Implementation in C Programming Language to store dynamic string arrays.

5. Reading Files using C Programming Language.

Methodology:

A *lexical token* or simply *token* is a string with an assigned and thus identified meaning. It is structured as a pair consisting of a *token name* and an optional *token value*. The token name is a category of lexical unit. Common token names are

* identifier: names the programmer chooses;
* keyword: names already in the programming language;
* separator (also known as punctuators): punctuation characters and paired-delimiters;
* operator: symbols that operate on arguments and produce results;
* literal: numeric, logical, textual, reference literals;
* comment: line, block.

In this assessment we are required to implement following tokens:

**1. TOK\_IDENT** for variables, functions and constants names.

**2. TOK\_INT\_LIT** for integer literals.

**3. TOK\_LET** for keyword let .

Special characters -

**1. TOK\_SQ\_BKT\_L** for [ .

**2. TOK\_SQ\_BKT\_R** for ] .

**3. TOK\_COLON** for : .

**4. TOK\_SEMICOLON** for ; .

**5. TOK\_COMMA** for , .

**6. TOK\_EQ** for = .

**7. TOK\_EOF** to denote the end of a file.

**8. TOK\_INVALID** to denote a token not known to the lexer.

To implement a lexer , first thing we need to do is analyze string. For this, we need DFA.

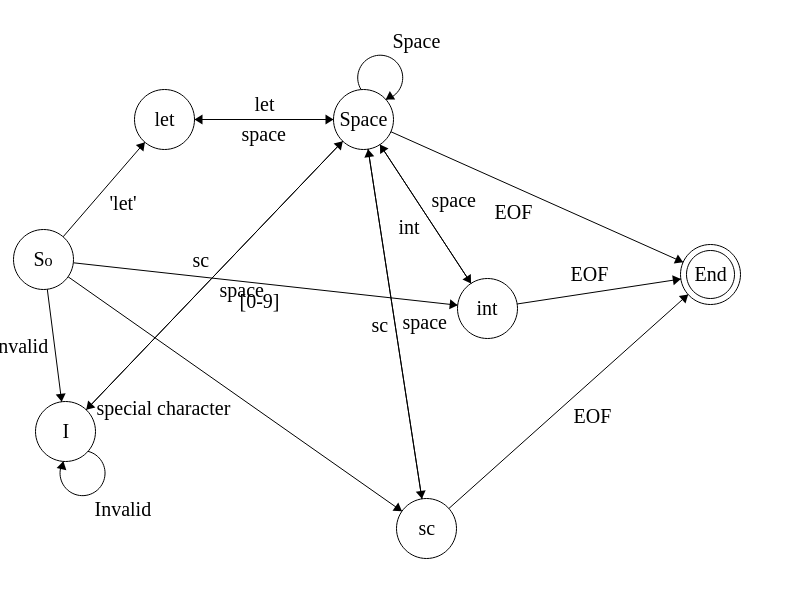
DFA or deterministic Finite Automata is a finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string. *Deterministic* refers to the uniqueness of the computation run. In search of the simplest models to capture finite-state machines.

A deterministic finite automaton Mconsists of

* a finite set of states Q
* a finite set of input symbols called the alphabet \Sigma 
* a transition function {\displaystyle \delta :Q\times \Sigma \rightarrow Q}
* an initial or start state q_{0}\in Q
* a set of accept states

Deterministic finite automata are always *complete*: they define a transition for each state and each input symbol.

DFA for this assessment is given below:



This DFA contains one start state and one end state and 5 intermediate state. Input of start state is a line of string. For each character certain comparisons would be performed and we will move to start state increment our string to to that point and continue from that state.

At each state we would generate a token unless we reach the end of life and generate EOF token.

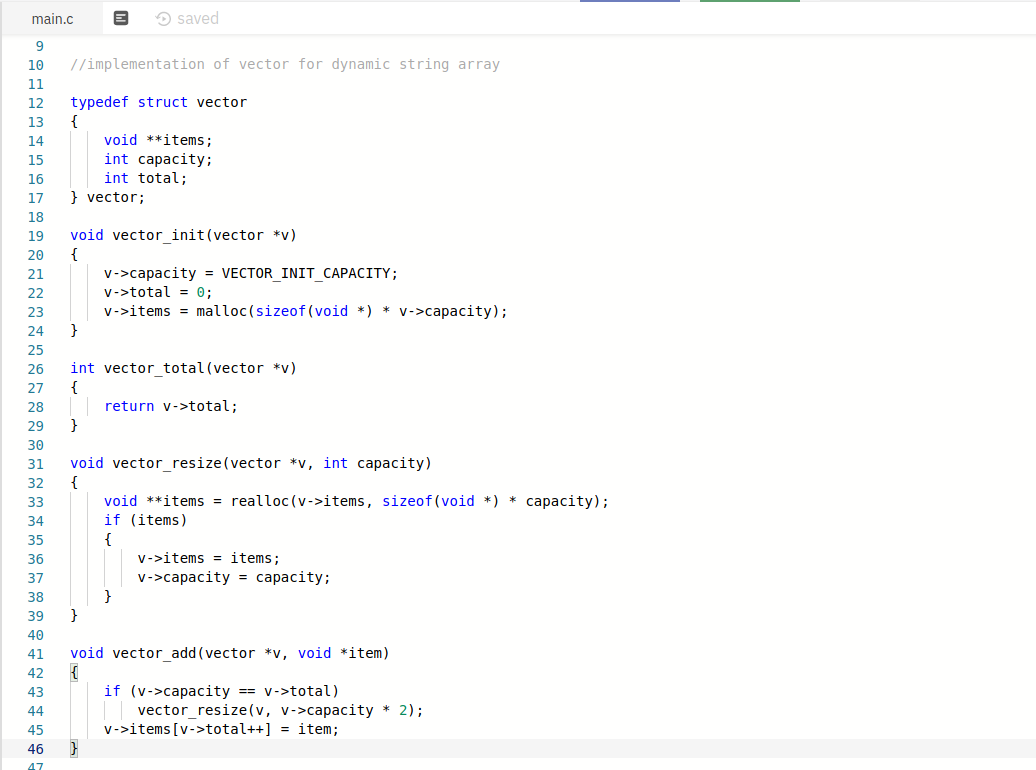
The process is simple. We will read the input file line by line. For each line EOF is ‘\n’ character. So as we do not know how many tokens each statement would contain we would require a dynamic array data structure which is known as vector in many programming languages.

A **vector** is a sequence container **data structure** that implements dynamic array, means size automatically changes when appending elements. A **vector** stores the elements in contiguous memory locations and allocates the memory as needed at run time.

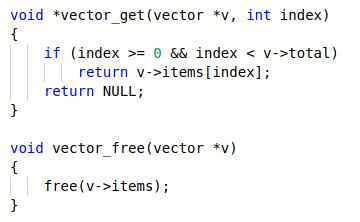
Before we move to implement the dfa we have to first implement the vector class data structure. We would only require add, allocate, reallocate feature of the vector as there are no token deletion.

Vector Implementation:

1. First we will need to define a struct. A struct in the C programming language is a composite data type declaration that defines a physically grouped list of variables under one name in a block of memory, allowing the different variables to be accessed via a single pointer or by the struct declared name which returns the same address.



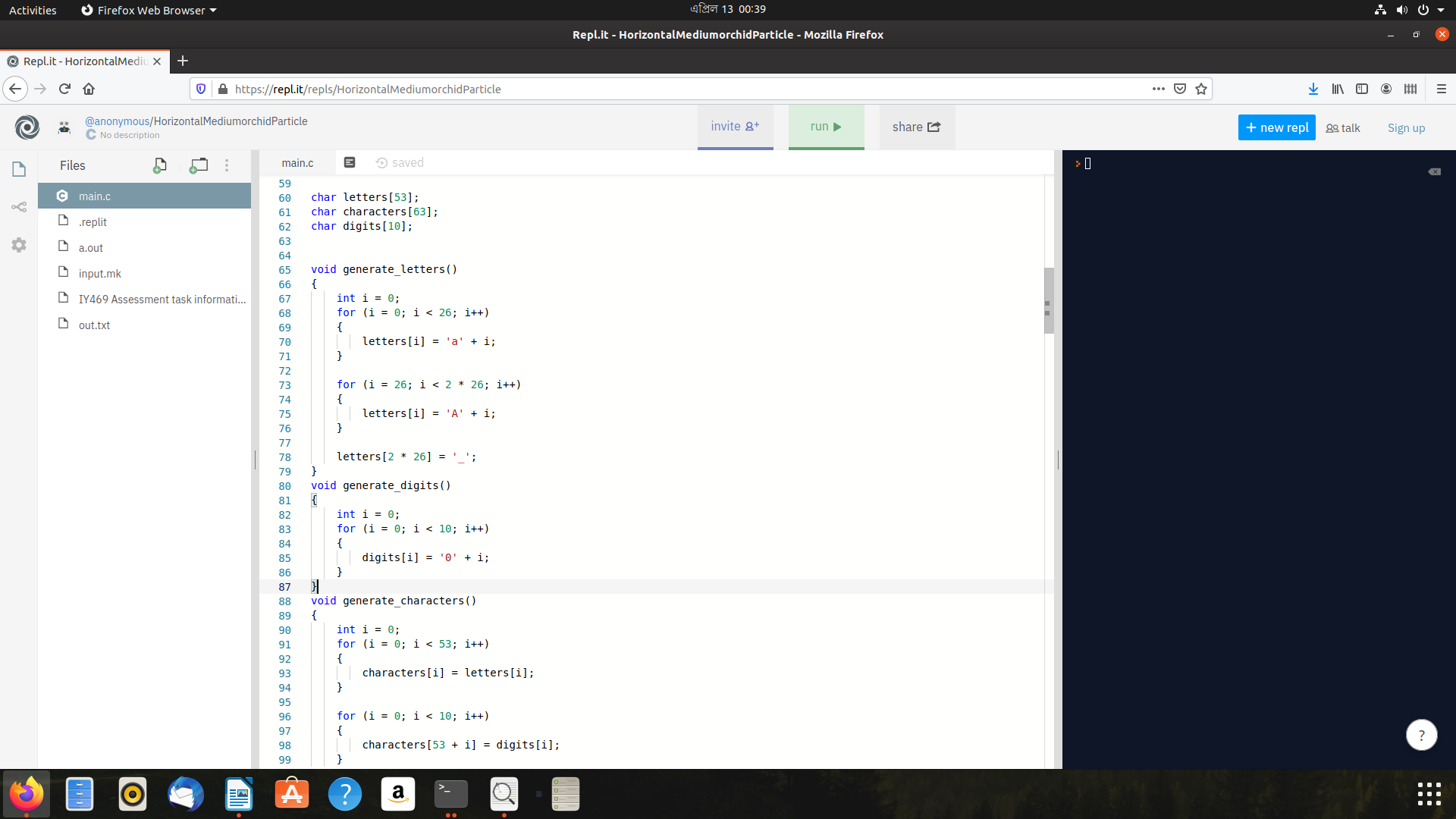
The functions mentioned in this images implements vector initialization, vector resize, vector add and total vector size.



These two functions gets items in the vector at particular point and after the use of the vector it frees the allocated memory

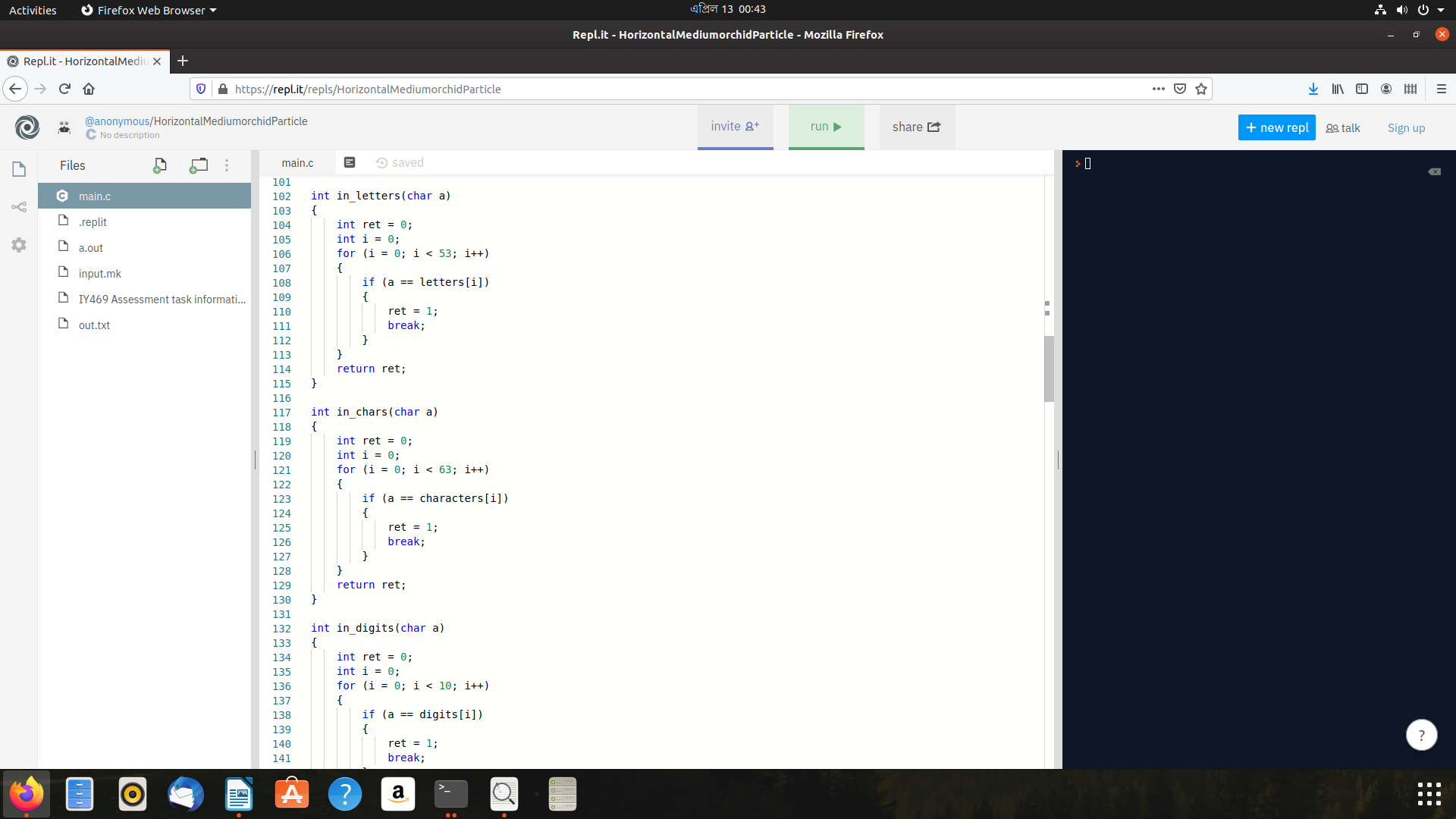
As now we have completed implementing the vector class, now we move to our main implementation

1. First we will need letter, digit and accepted character set for identifiers.

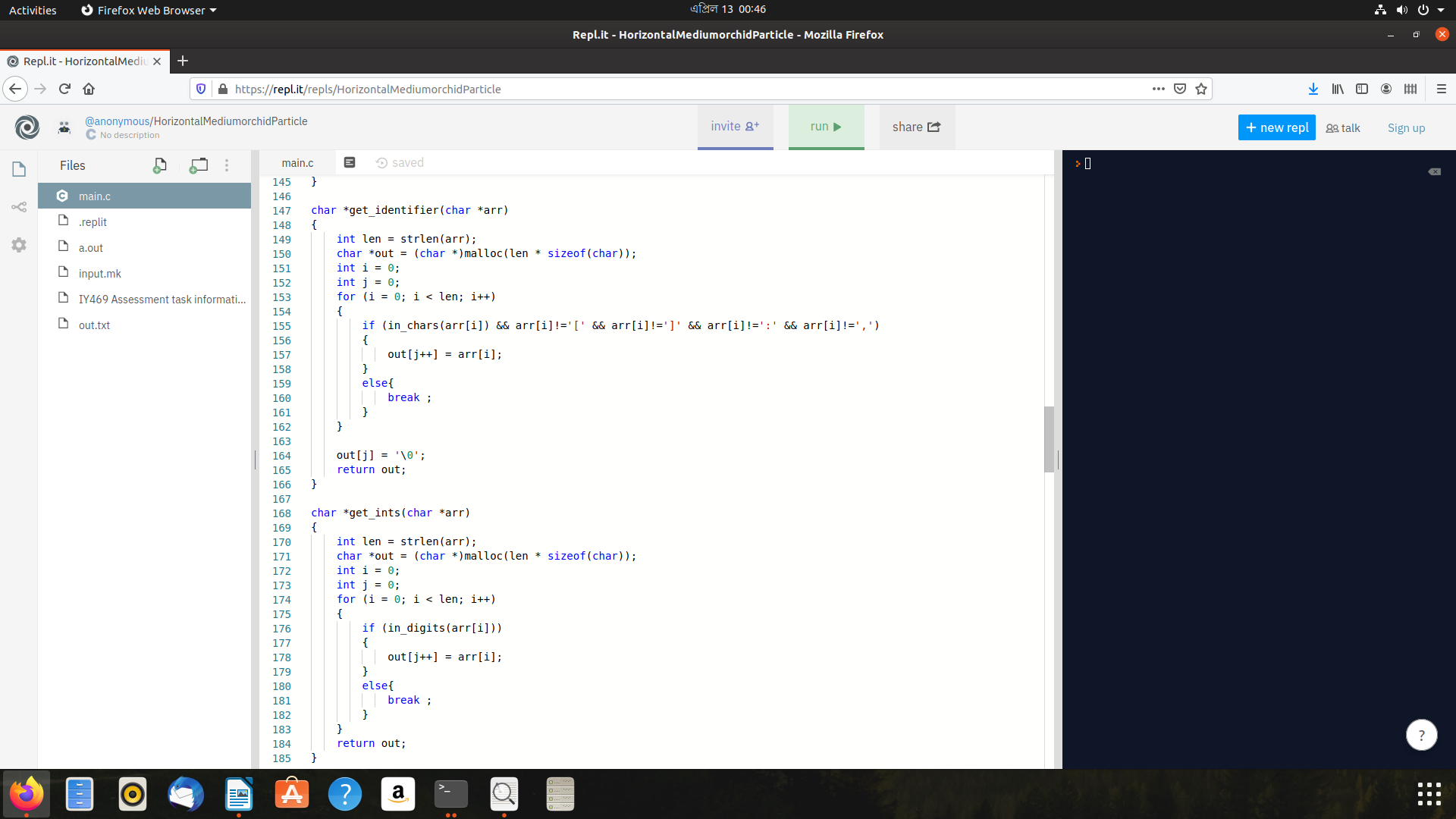


We declared three global variable arrays for storing digits, letters and characters and these functions sets the values to the appropriate variable.

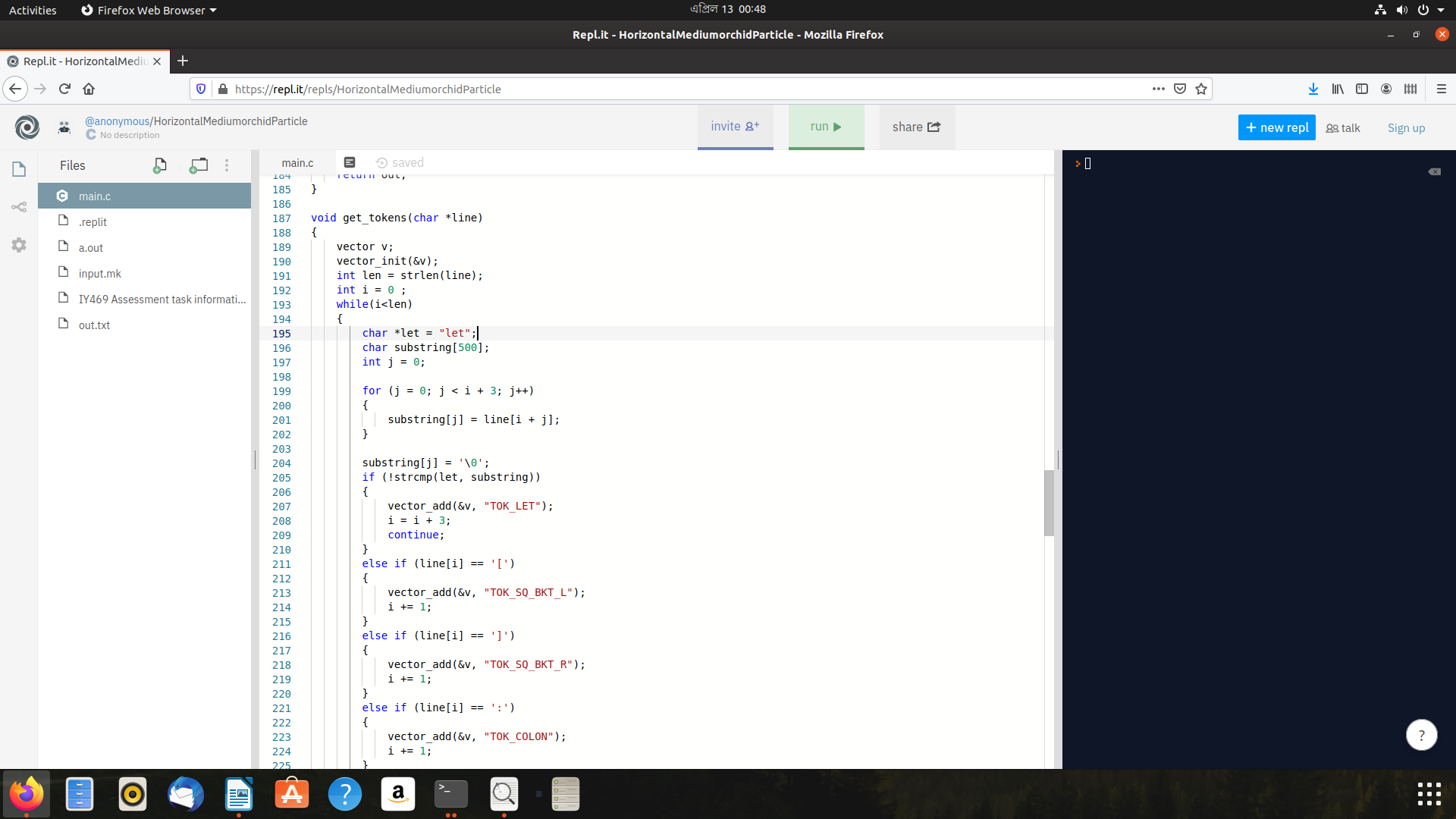
We will also require 3 helper function to check whether a character is a letter or digits or in accepted character set. These functions are in\_digit, in\_letter, and in\_chars

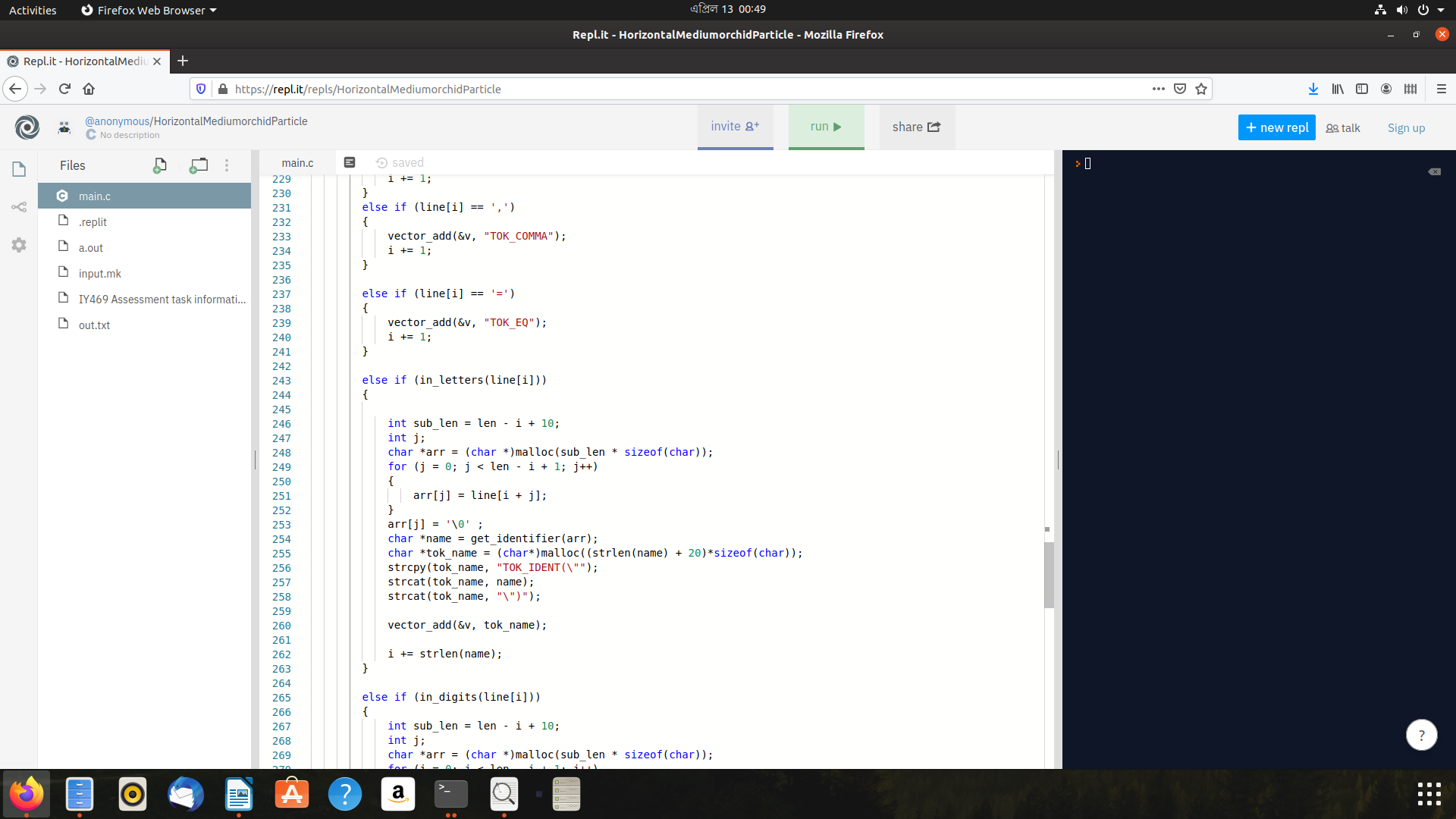
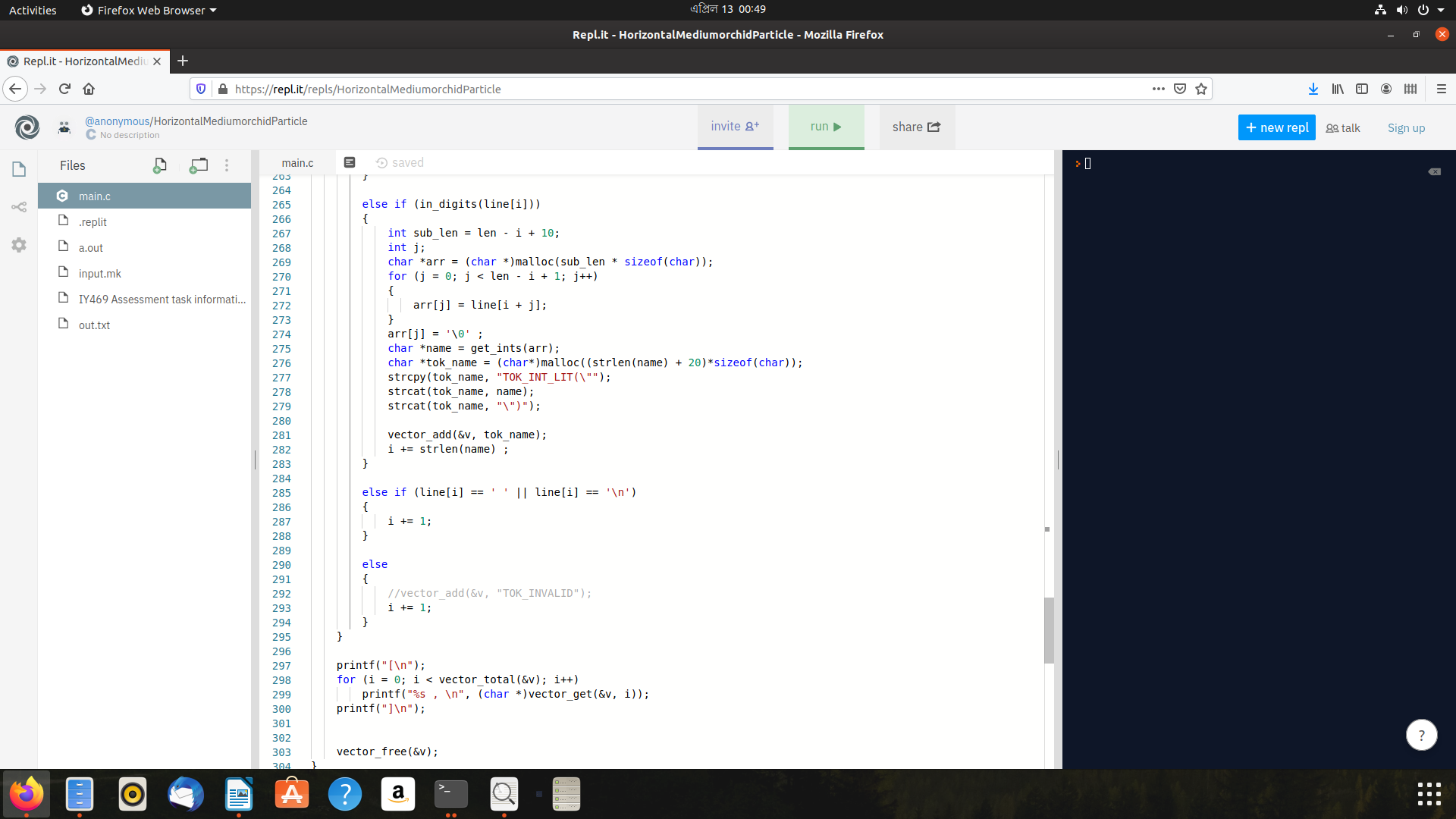


We will also require 2 additional functions which will generate integer and identifier from the given string.

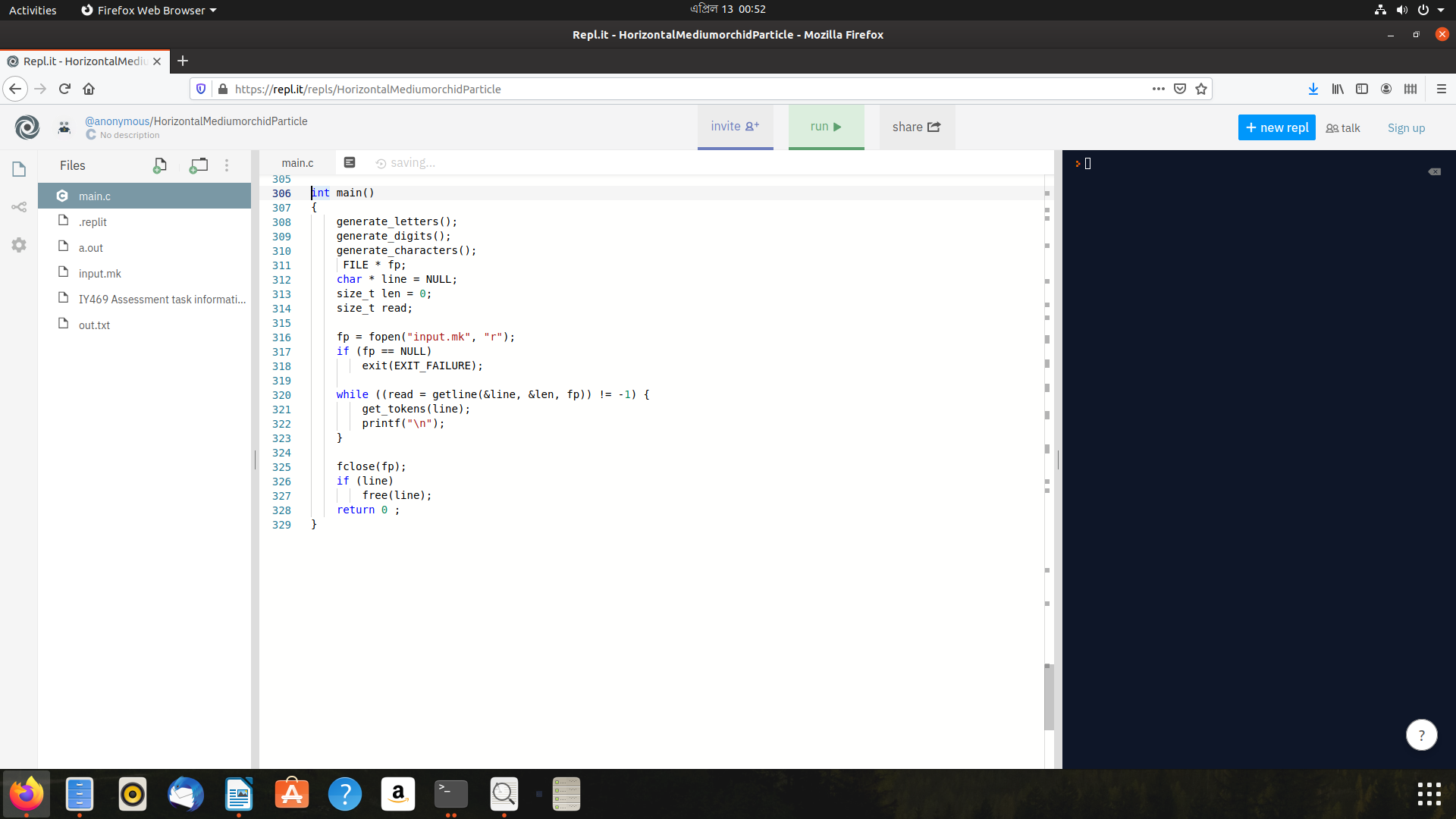


Now, as we have implemented all required function implementation we will switch to our DFA Implementation. get\_tokens function is the DFA. It takes a line as a input and prints all tokens in the line.





And finally the main function where we will call the generator functions and read line from input file and pass each line to the get\_tokens(function)



Results :

The input is stored in input.txt file and output is written in out.txt file. No other specific code is written for output code generation. A simple linux command is used to generate this output file.

**./a.out >> out.txt**

Input for this program is:

let alpha\_1 = 54;

let vec = 0:2:6;

let m = [1,2,3; 4,5,6];

let s = m[:,2];

m[:,2] = m[:,1];

And the output generated from the program is:

[

TOK\_LET ,

TOK\_IDENT("alpha\_1") ,

TOK\_EQ ,

TOK\_INT\_LIT("54") ,

TOK\_SEMICOLON ,

]

[

TOK\_LET ,

TOK\_IDENT("vec") ,

TOK\_EQ ,

TOK\_INT\_LIT("0") ,

TOK\_COLON ,

TOK\_INT\_LIT("2") ,

TOK\_COLON ,

TOK\_INT\_LIT("6") ,

TOK\_SEMICOLON ,

]

[

TOK\_LET ,

TOK\_IDENT("m") ,

TOK\_EQ ,

TOK\_SQ\_BKT\_L ,

TOK\_INT\_LIT("1") ,

TOK\_COMMA ,

TOK\_INT\_LIT("2") ,

TOK\_COMMA ,

TOK\_INT\_LIT("3") ,

TOK\_SEMICOLON ,

TOK\_INT\_LIT("4") ,

TOK\_COMMA ,

TOK\_INT\_LIT("5") ,

TOK\_COMMA ,

TOK\_INT\_LIT("6") ,

TOK\_SQ\_BKT\_R ,

TOK\_SEMICOLON ,

]

[

TOK\_LET ,

TOK\_IDENT("s") ,

TOK\_EQ ,

TOK\_IDENT("m") ,

TOK\_SQ\_BKT\_L ,

TOK\_COLON ,

TOK\_COMMA ,

TOK\_INT\_LIT("2") ,

TOK\_SQ\_BKT\_R ,

TOK\_SEMICOLON ,

]

[

TOK\_IDENT("m") ,

TOK\_SQ\_BKT\_L ,

TOK\_COLON ,

TOK\_COMMA ,

TOK\_INT\_LIT("2") ,

TOK\_SQ\_BKT\_R ,

TOK\_EQ ,

TOK\_IDENT("m") ,

TOK\_SQ\_BKT\_L ,

TOK\_COLON ,

TOK\_COMMA ,

TOK\_INT\_LIT("1") ,

TOK\_SQ\_BKT\_R ,

TOK\_SEMICOLON ,

]

Discussion:

From the output we can see that, the program is definitely working and correctly printed the tokens. The only limitation in this program is not able to recognize floating point number (12.46) and other set of character which are used in other programming languages but no applicable in this language.

In a typical lexical analyzer the token set range is way higher than this assessment like recognize floating points, function declaration and definition. We can say that as we have introduced all helper and necessary functions we can easily extend it to recognize more different type of tokens.

Conclusion:

So, after completing this assessment, the outcome is indeed an inspirational. While it was difficult task to complete it required us to study and acquire knowledge of different sectors of computer science and merging all of these to one assignment. Although the tasks or features of this program is limited due to potential increase of complexity level for each extra tokens, we can think that this theory or techniques that are used in implementing this program would be beneficial in developing or implementing other types of lexical analyzer for different language and knowledge of dfa can also be used in other sectors in computer programming.

References:

1. https://en.wikipedia.org/wiki/Deterministic\_finite\_automaton

2. https://eddmann.com/posts/implementing-a-dynamic-vector-array-in-c/

3. https://en.wikipedia.org/wiki/Lexical\_analysis

4. https://www.programiz.com/c-programming/examples/read-file