CPSC 323 Documentation

Names: Burhan Taskin, George Hanna, Christopher Contreras, Brandon Wu

1. Problem Statement:

Our project aims to create a lexical and syntax analyzer for a simplified version of the Rat24F programming language. The program will input a small Rat24F script, such as one that converts Fahrenheit to Celsius. It will break the script down into tokens, which are the basic building blocks of the code. After that, the program will check if the script follows the correct syntax rules for the language. Finally, it will simulate running the script to show how it would work in practice.

2. How to use your program:

The use of a C++ compiler like G++ is necessary to compile our code and then run the program. The program includes an example script written in Rat24F, which will go through several steps:

A. It will break down the code into tokens.
B. It will check the syntax to ensure everything is correct.
C. It will run a mock version of the program.

Once this process is complete, the program will display the tokens and corresponding lexemes, followed by a simulated execution of the script.

3. Design of your program:

This part of our program goes through the input code one character at a time and breaks it down into smaller parts called tokens. These tokens include things like keywords, names of variables or identifiers, operators, and symbols. It uses patterns to recognize different things like variable names, numbers, operators, and separators in our code.

Data Structures:

- enum TokenType: Enum for token types (KEYWORD, IDENTIFIER, INTEGER, etc.).
- struct Token: Structure containing a token's type and its value.
- vector<Token>: A vector to store the list of tokens.

Key Algorithms:

- Regular expressions to identify token types.
- Iterative parsing to break the input into tokens.

Syntax Analyzer:
The syntax analyzer is a simple parser that reads through the tokens one by one and shows both the type of each token and its value. It is just a basic setup for more advanced syntax analysis that will later check if expressions and statements are correct.

Execution Simulator:
A simple execution phase where tokens are printed sequentially to simulate the program's behavior.

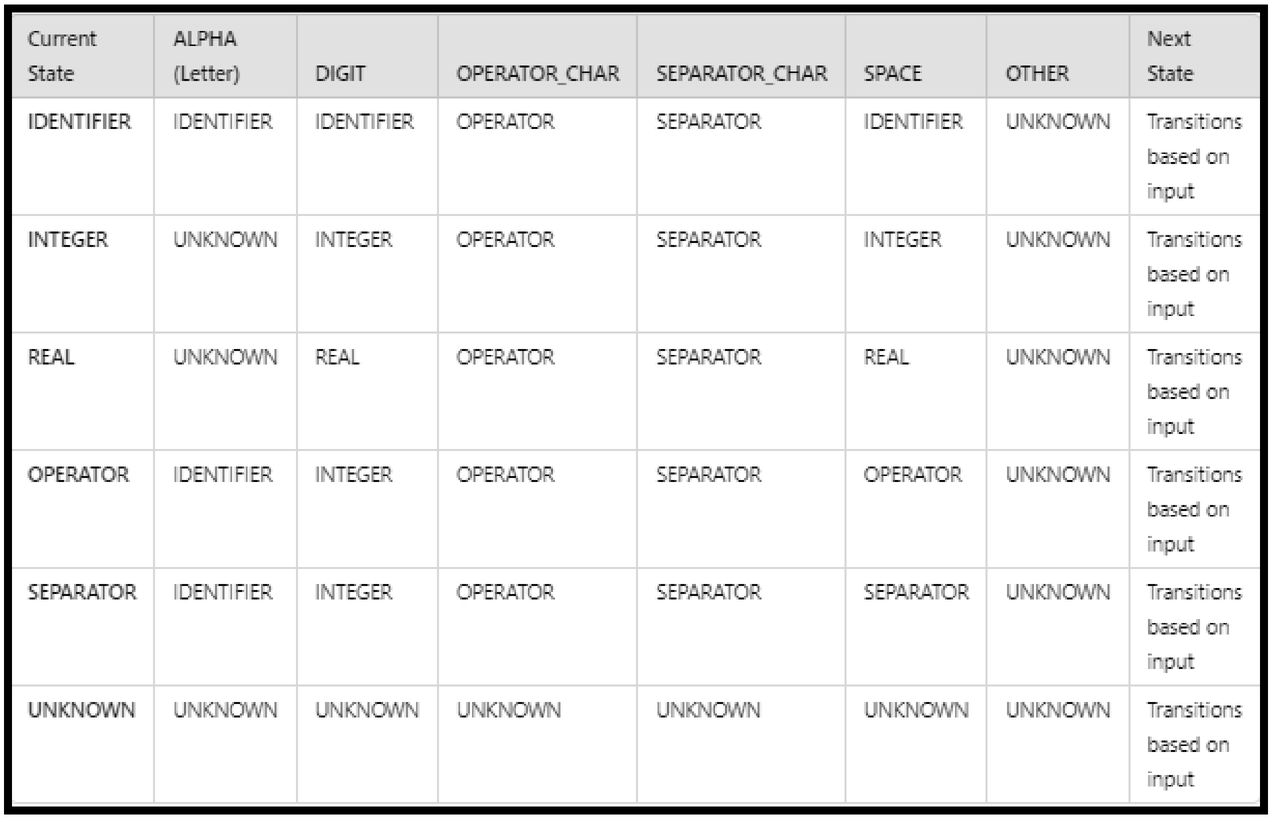
4. **Any Limitation:**

None.

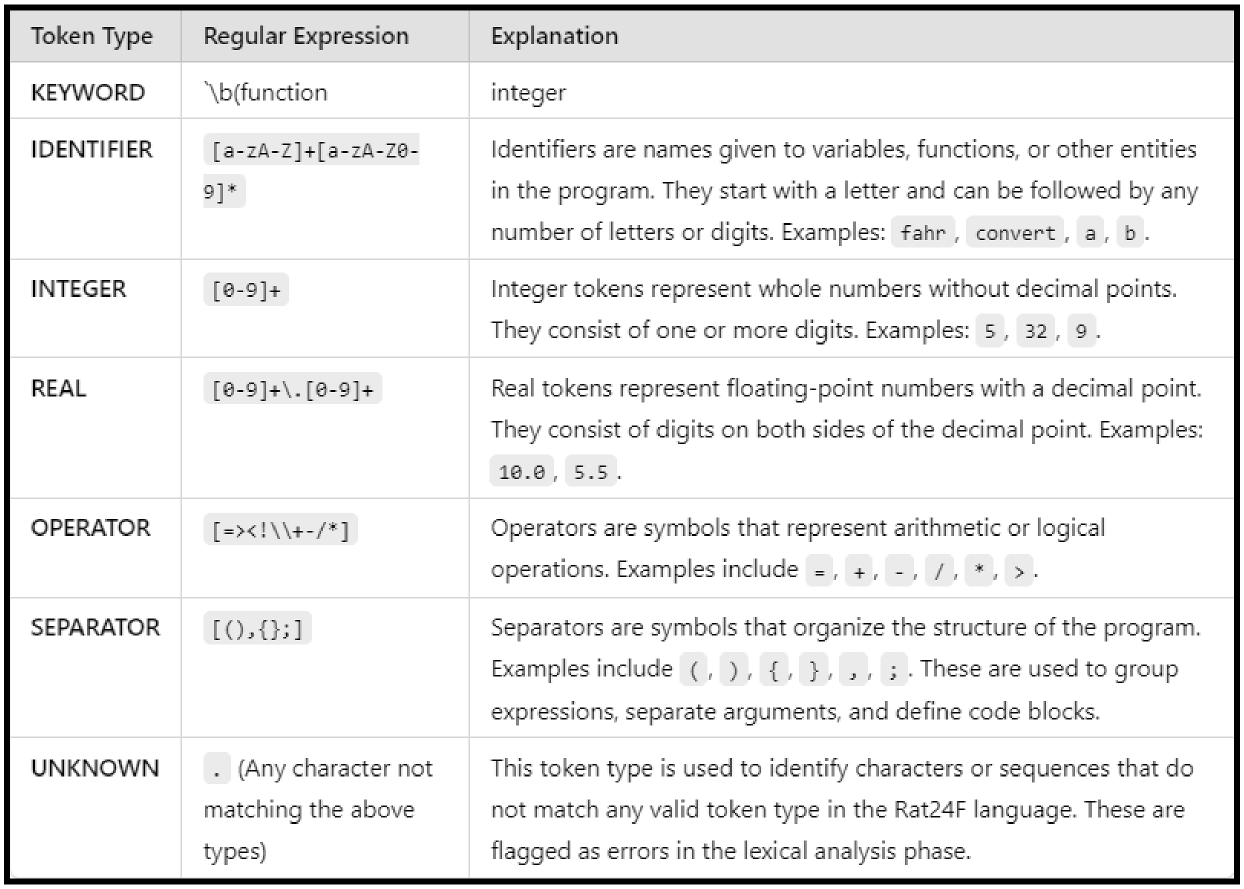
5. **Any shortcomings:**

None.

The Chart:



Current State	ALPHA (Letter)	DIGIT	OPERATOR_CHAR	SEPARATOR_CHAR	SPACE	OTHER	Next State
IDENTIFIER	IDENTIFIER	IDENTIFIER	OPERATOR	SEPARATOR	IDENTIFIER	UNKNOWN	Transitions based on input
INTEGER	UNKNOWN	INTEGER	OPERATOR	SEPARATOR	INTEGER	UNKNOWN	Transitions based on input
REAL	UNKNOWN	REAL	OPERATOR	SEPARATOR	REAL	UNKNOWN	Transitions based on input
OPERATOR	IDENTIFIER	INTEGER	OPERATOR	SEPARATOR	OPERATOR	UNKNOWN	Transitions based on input
SEPARATOR	IDENTIFIER	INTEGER	OPERATOR	SEPARATOR	SEPARATOR	UNKNOWN	Transitions based on input
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	Transitions based on input



Token Type	Regular Expression	Explanation
KEYWORD	`\b(function	integer
IDENTIFIER	`[a-zA-Z]+[a-zA-Z0-9]*`	Identifiers are names given to variables, functions, or other entities in the program. They start with a letter and can be followed by any number of letters or digits. Examples: `fahr`, `convert`, `a`, `b`.
INTEGER	`[0-9]+`	Integer tokens represent whole numbers without decimal points. They consist of one or more digits. Examples: `5`, `32`, `9`.
REAL	`[0-9]+\.[0-9]+`	Real tokens represent floating-point numbers with a decimal point. They consist of digits on both sides of the decimal point. Examples: `10.0`, `5.5`.
OPERATOR	`[=><!\\+-/*]`	Operators are symbols that represent arithmetic or logical operations. Examples include `=`, `+`, `-`, `/`, `*`, `>`.
SEPARATOR	`[(),{};]`	Separators are symbols that organize the structure of the program. Examples include `(`, `)`, `{`, `}`, `,`, `;`. These are used to group expressions, separate arguments, and define code blocks.
UNKNOWN	`.` (Any character not matching the above types)	This token type is used to identify characters or sequences that do not match any valid token type in the Rat24F language. These are flagged as errors in the lexical analysis phase.

[S] -- L --> (ID) -- L/I --> (ID) -- ε --> [Z]   [Identifiers: red]

[S] -- I --> (INT) -- I --> (INT) -- ε --> [Z]   [Integers: blue]

[S] -- I --> (REAL) -- I --> (.) -- I --> (REAL) -- ε --> [Z]   [Reals: yellow]

(ID) -- ε --> [Z]
(INT) -- ε --> [Z]
(REAL) -- ε --> [Z]

Legend:
L: Any alphabetic letter [a-zA-Z]
I: Any digit [0-9]
A: Any character [a-zA-Z0-9!@#$%^&*()_+= etc.]
ε: Represents an epsilon (ε) transition (transition without consuming any input)