Code Report: A table-driven lexer for VC

Table of Contents

Introduction

This is a report for the project *A table-driven lexer for VC*.

Purpose

As the teacher's slide demand, we need to write a table-driven lexer for VC. The lexer should be able to scan the input file and output the tokens.

Design

The lexer was written in Python, and it has 3 parts:

1. The transition table
2. The utility functions
3. The tokenizer

Transition Table

This is the reason why we missed the deadline, as there's still bugs in the transition table to be found.

First, as the teacher's slide mentioned, we define the state transition graph (which is painstakingly, with the help of the DOT language). The graph is based on the VC Language Definition.

After that, we wrote the transition table in the form of a csv file (which is also painstakingly written by hand, using Excel). Then we change its name to `transition_table.dat` and put it in the `core` folder.

The format of the table is as follows:

```
state,input1,input2,input3...
s0,sA,sB,sC...
s1,sX,sY,sZ...
...
```

We treat the first row (from the input1) as the input alphabet, and the first column (from the s0) as the state set. The rest of the table is the transition function (i.e. from s0, with input1, the next state is sA).

Utility functions

The utility functions are written in the `core/utils.py` file. It is a utility class that provides methods for reading the source code, getting state and key maps, parsing the transition table, and writing the output files.

`__init__(self, path: str)`: Initializes the instance with the transition table, state and key maps, source code, and other necessary variables. It could also create new directories for the output files.

`get_source_code(self, path: str)`: Reads the source code from the provided file path.

`get_maps(self, data: list)`: Retrieves the maps for states and keys. The maps are used for accessing the transition table with the state and key names (i.e. `s0` and `input1`, in our case: `0` and `+`).

`get_next_state(self, state, key)`: Determines the next state from the current state and input key.

`get_error_line(self, line, column)`: Gets the context of an error at a given line and column. Example:

```
Error at line 2, column 30.

int main() {

    putString("Hello, world!);
------------------------------^ Error here.

}
```

`write_tokens(self, tokens: list)`: Writes the tokens to an output file.

`write_verbose(self, tokens: list)`: Writes the verbose information of tokens to an output file.

`find_type(self, state)`: Finds the type of a token given its state.

Tokenizer

The tokenizer is written in the `core/lexer.py` file. It is the main class that use the utility functions to tokenize the source code and write the output files.

Here is the tokenizer's workflow:

1. Check if the required command-line argument (source code file path) is provided.

2. Initialize a `Utils` instance with the provided file path.
3. Tokenize the source code using the transition table, state and key maps, and various utility methods.
4. Write the tokens and their verbose information to two separate output files.

Detailed workflow

- First, we check if the required command-line argument (source code file path) is provided. If not, we print the usage `Usage: python lexer.py <path>` and exit.
- Next, we initialize a `Utils` instance with the provided file path. This will read the source code, get the state and key maps, and parse the transition table.
- Some required variables are initialized. Note that source code is considered as a string, and the line `source_code = utils.source_code + " "` is used to add a space at the end of the source code. This is due to the lexer's design, which won't be able to identify the last token correctly.

  The tokens is initialized as an array containing a list of tokenized tokens. Every token indicates four information of a tokenized character or keyword:

  - The first information returns a string type and indicates its content.
  - The second information returns an interger type and indicates its ending state.
  - The third information is a two-dimensional array type [a, b] including a row and two collumns. It indicates a starting point of a given character by the index of a specific row and collumn in the source code, starting from index 1.
  - The fourth information is a two-dimensional array [a, b] including a row and two collumns. It indicates the ending point of a given character by the index of a specific row and collumn in the source code, starting from index 1. For example, considering the following keyword "int" in the following source code:

    ```
    int main() {
        putString("Hello, world!");
    }
    ```

  As can be seen clearly from the source code, the keyword can be extracted into the following information:

  - Content: int (the name of the keyword is int).
  - Ending state: 33 (The ending state of the keyword is 33 according to the transition table)

  - Starting points: row(1) collumn(1) (The character starts at row 1, collumn 1 in the source code).
  - Ending points: row(1) collumn(3). (The character ends row 1, collumn 3 in the source code).

- Start looping through the source code. The loop will stop when the current index is greater than the length of the source code.
  - If the next state is an error state, the script prints an error message and exits.
  - If the next state is empty or None, it indicates the end of the current token, and the token is added to the tokens list. The state machine is reset to the initial state.
  - If the next state is a comment state, the script ignores the comment and moves on to the next character.
  - If the next state is a special character state, the script handles escape sequences and adds the corresponding character to the current token.
  - In other cases, the script updates the current state and appends the next character to the current token.
- At the end of the file, an end token (`$`) is added to the tokens list.
- The script filters out any empty tokens that were created from whitespaces or newlines, then writes the token information to two output files:
  - "tokens.vctok": Contains the list of tokens extracted from the source code.
  - "token.verbose.vctok": Contains verbose information of the tokens, such as state, token type, spelling, and position.

Conclusion

The lexer is able to tokenize the source code correctly, but there are still some bugs in the transition table that need to be fixed.

Some noted problem/bugs:

- Misidentify the ' character. In example: `"I don't know"` is misunderstood as an error.
- Weird state-handling for some characters (e.g. `?`, `~`, `:`, ect.).
- Unable to parse empty strings (e.g. `""`).

However, we also found some interesting things:

- Making a transition table is a very tedious task, and it is very easy to make mistakes.
- The transition table is very sensitive to the order of the states and keys. If the order is wrong, the lexer will not work correctly.