Daniel Tsioni (Section 02) and Gavriel Tsioni (Section 01)
Systems Programming 198:214

Programming Assignment 1 (Tokenizer) Readme

Solution Description

For our solution we decided to implement a Finite State Machine. We started by drawing our FSM which would be able to identify the tokens we needed. This allowed us to go character by character and come to conclusions about how the token was being constructed, and allowed us to determine if a token was malformed. We knew that a solution like this (once we implemented the basics in C) would be easy to implement entirely, and would be easily expandable, readable, and extensible.

Our tokenizer moves through the string character by character and performs the corresponding transition in the FSM (explicitly defined in the C implementation). It concatenates those characters to a current token string, to keep track of the longest unbroken token that we are building.

We decided to implement our tokenizer so that it would move as far ‘right’ in the string as possible before identifying the token. It would only stop moving to the right when it encountered something that broke a token, which we decided was either a whitespace character (given in the project outline) or a character which would cause the current token to be malformed. Doing this allowed us to settle a large amount of ambiguities. For instance, with a string “0x00ffh”, this could be interpreted by another student as a hex “0x0”, then a zero “0”, then a word “ffh”, or as any other number of interpretations. By moving as far right before we break a token, the same string can only be interpreted as a hex “0x00ff” and a word “h”.

Certain states can assign a ‘type’ to the token. This means that the FSM will keep track of what kind of token has been built up until this point. If any transition would cause the current token to be malformed (or is a whitespace or escape character), we instead break out of the FSM and return the currently built token with the currently designated type. Moving through the FSM then begins with the character that caused the FSM to end the last token.

Whitespace characters are handled through two different cases. A whitespace either occurs while the current token is not blank, meaning it breaks the current token and is skipped, or it occurs while the current token is blank, and is skipped. Any amount of consecutive whitespace characters are skipped as well, in either case.

By using an FSM, our solution became a matter of simply mapping out our hand drawn FSM into our program. We uniquely identified each state with a number in our diagram to help with the implementation.

Program Description

Our program places the entire input string into the Tokenizer struct as a char * type. It also holds an index integer, which keeps track of how far into the input stream we have tokenized, and a state integer, which keeps of what state in the FSM the program is in.

The main function ensures a correct number of arguments, then creates the tokenizer. It then calls the TKGetNextToken function on the tokenizer until it returns a null value (which indicates the end of the string was reached).

The resultOutput function takes a given token type and a token and prints them in the format “type: “token"“.

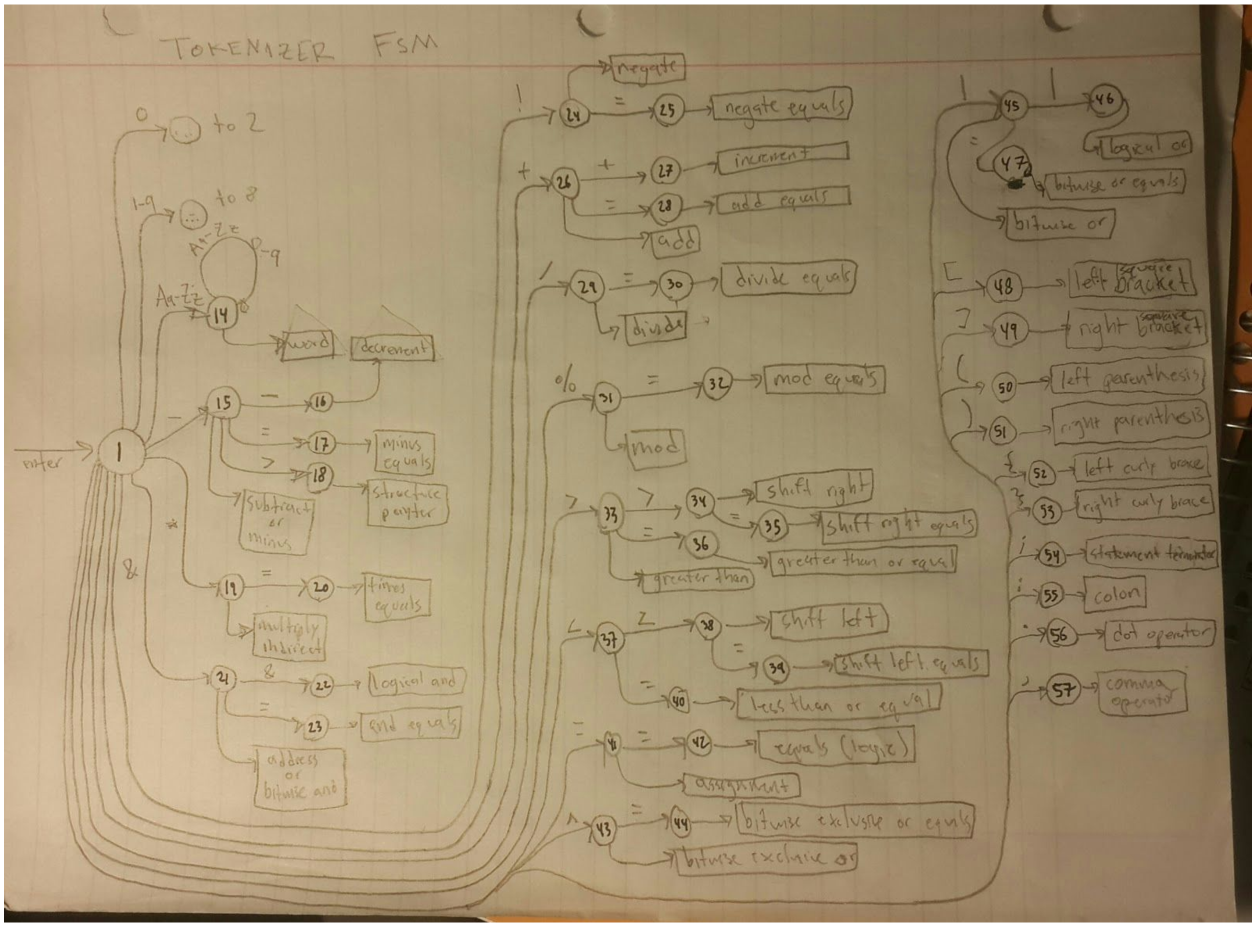
The escapeCharacterResultOutput function takes an escapeCharacter and prints it in the form “escape character: “[character in hex]"”.

The FSM function takes a Tokenizer struct and extracts the next valid token out of the string. We modeled the FSM by utilizing a ‘switch’ statement. Each case represents a uniquely numbered state in our FSM diagram, and each ‘if’ statement within a case represents the transitions which branch off of that state. After the switch statement (which represents when we are leaving a state), the current char is concatenated to the string holding the token. The while loop outside of the switch brings the program back to the top of the switch statement and enters into the state indicated by the last transition. This is repeated until the function finds a token and returns it. The tokens type is assigned in the variable ‘type’ which keeps track of what type the token will be if the FSM were to exit at that state. This removes the need for knowledge of the next character, because if the next character malforms our token we can simply return our token and print with the type, and then start again at the malforming character. The token is printed in this function too, whenever it would either become malformed, a whitespace character is found, or an escape character is found. If the ‘\0’ character is found, the function returns NULL.

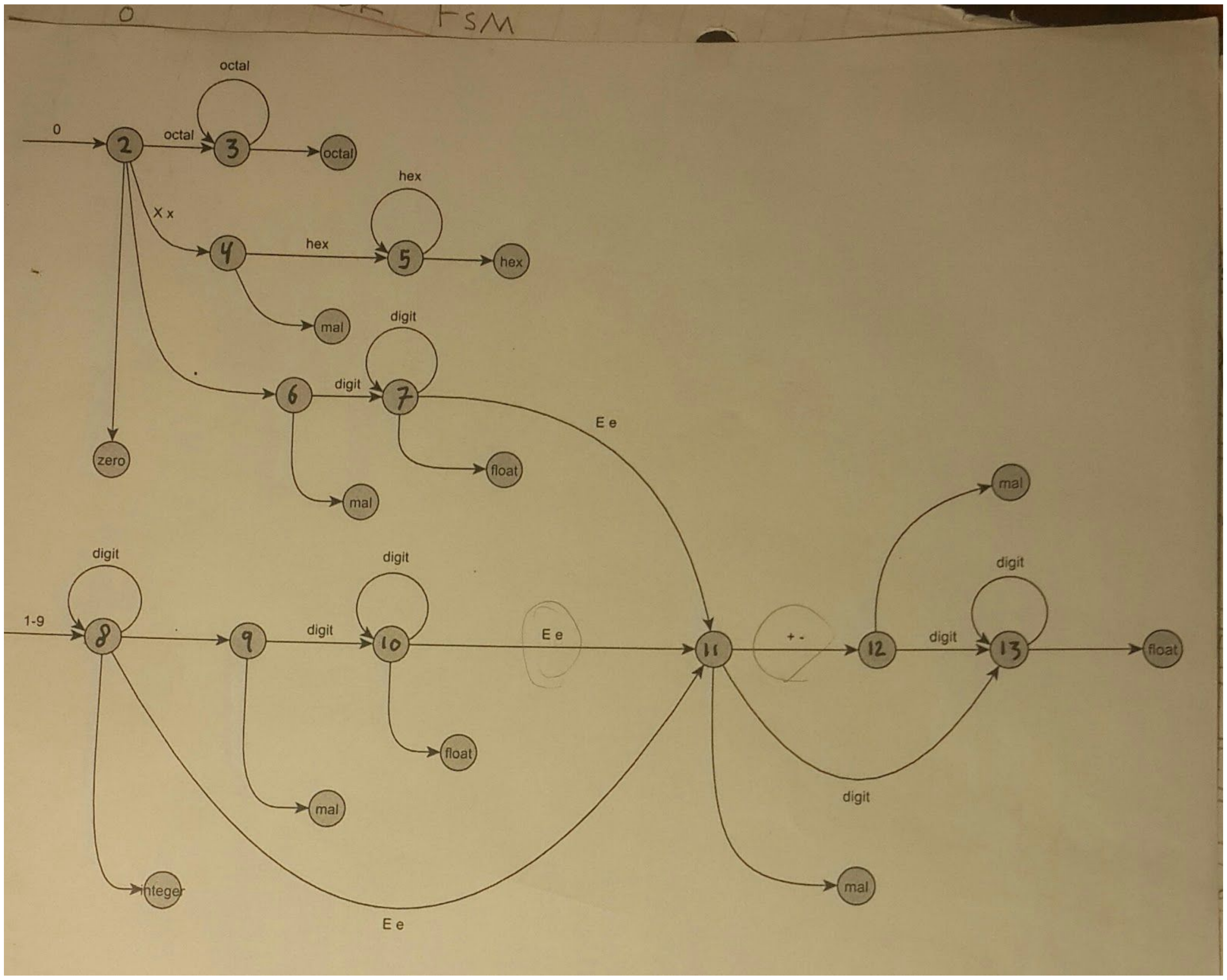
Features to Note

By implementing a Finite State Machine, we made our solution very simple to scale. New tokens can be added very quickly and very intuitively. To add new tokens, you simply need to do add your new FSM states as cases in the switch statement and add the proper transitions with type assignments. The solution is also very readable, and easy to understand.

Our FSM



We created an FSM, similar to the one provided for us, which describes the C operators given on the C reference card, as well as the following: ., ,, {, }, [,], (,). The two transitions at the top, from state 1 to 2 and 1 to 8 go to the FSM below, which was the FSM provided for us.


O
FSM
0
octal
octal
octal
X x
hex
hex
hex
mal
digit
digit
.
E e
float
mal
zero
mal
digit
digit
1-9
.
digit
E e
+ -
digit
digit
float
float
mal
digit
integer
mal
E e