Lab 1, Part 1

Problem Statement

> [Part 1]: Study the generated lexer in file c.lex.cpp, and write a short report (1-5 pages) on the logic of the code produced through the lexical analyser generator. In particular, we are interested in the logic of the function represented by YY_DECL (representing the yylex() function). Keep your report as brief as possible, but do not miss out any interesting fact about the generated code or logic. Your report should reflect your understanding of the logic of the generated lexical analyzer.

Background

For each token class in our language we define a *regular expression* that matches all the strings for that token and an *action* that is called when a token is found in the stream of input chars.

The flex tool reads the regular expressions in a `.l` file and creates a DFA which recognizes all the token classes.

This can be done by converting the `Regex` into `NFA` using `McNaughton–Yamada–Thompson algorithm` and then to a `DFA` using `subset construction algorithm`.

However in practice more advanced techniques like `Gerard Berry and Ravi Sethi algorithm` to convert `regex` directly into `DFA` is used.

Next, the number of DFA states maybe minimized using `Hopcroft's algorithm`.

The input character stream is run on the DFA until a `dead state`(A state from which no accepting state can be reached) is reached. Then we backtrack to find an accepting state.
At this point we have found the *longest prefix* of the input that matches with a token.

The longest prefix that is found may match to more than one regular expressions. In this case we output the token corresponding to the regex specified first in the file and call the specified action.

Understanding the code

DFA and actions

The code inside the label `yy_match` runs the DFA.

The array `yy_ec` maps each char to a character class.

Instead of storing a `2-D transition table`, the transition table is compressed and stored in a `1-D` array `yy_nxt`.
To get the `NextState(state, char)` we index the array using `yy_base[state] + yy_ec[char]`, here `yy_base` points to the start of the transitions for a state in the `1-D table`. Note that many states have the same `yy_base[state]`.

Actions for accepting states are identified using ids, which are stored in `yy_accept`.
If a state is non-accepting then `yy_accept[state] = 0`.

While running the `DFA` we may reach into some `accepting` states. We do not stop there and keep recording the `last_accepting_state` and `last_accepting_char_pos` as we move on. This is because we want to find the longest prefix that matches with a token.

The `DFA` is run until we reach the `base state = 699`.
After we are out of the DFA loop (`yy_match`), we go into the `yy_find_action` label.

The `base state 699` seems to occur in 2 cases:

1. When we have reached a dead state in the DFA i.e. there is no possibility of the current prefix matching with any token. (For ex: a whitespace after "int" `int `).
2. When we have found an accepting state an there is no possibility of a longer prefix being an accepted state. For ex: when we are not inside a quotes and have found a semi-colon `;`.

In the Case[1], `yy_accept[current_state] = 0` and we "roll back" to the last found accepted state and get the corresponding action. In C language, most of the time this is just the previous character.

In Case[2], `yy_accept[current_state] != 0` (which means we are in an accepting state) and we directly get the action for the current state.

Finally we perform the action in `do_action` label using a `switch-case` block.

> The longest prefix that is found may match to more than one regular expressions. In this case we output the token corresponding to the regex specified first in the file and call the specified action.
>
> This is not checked at runtime in the code, instead the highest priority action is hard coded in the `DFA` and `yy_accept`.

Buffering Logic

The input file is read into memory using double buffering. While running the DFA we might run out of buffer space, in this case we switch to the next buffer and the old buffer is filled with the next block of the input file.