Lab 1 (part 1) - 2019CS10399

1 `c.l` File

This is a flex file that is used to define the regular expressions used to identify each token. This file also has code to return the corresponding token, directly, or after processing the text (`yytext` variable). For the case of `comment`, input is consumed until the block comment terminates.

2 Generating `c.lex.cpp` from `c.l`

`flex` command is used to generate `c.lex.cpp` file from `c.l` file. This requires `c.tab.hpp` file to be present which is generated by the parser.

3 Understanding `c.lex.cpp`

`c.lex.cpp` file is an auto-generated file based on the contents in the `c.l` file. The main scanner function that does the actual lexing is `yylex()` (this has been aliased as `YY_DECL` in the file).

3.1 Working of `YY_DECL`

1. The function maintains the current state the DFA is in using a `yy_current_state` variable.
2. Two character pointers are also maintained `yy_cp` (which stores the current character pointer), `yy_bp` (which stores the *base* pointer: position where we started off in the current run)
3. `yy_act` integer is also maintained which stores the action to be performed on matching with a particular token.
4. If this function is being called for the first time, it initializes a few variables such as `yy_start`, `yyin`, `yyout`. It also (creates and) initializes the buffer stack for buffering the input.
5. Then the actual lexing begins, wherein the entire file is read until end-of-file is reached.

3.2 Lexing in `YY_DECL`

1. Ther are 385 states in the DFA, most of which are accepting states. This information is stored in the `yy_accept` array. Non-zero value for a state implies that it is accepting and the value corresponds to the action that has to be taken on reaching the state.

2. Inside the while loop, there are multiple labels such as `yy_match`, `yy_find_action`, `do_action`. Working of these labels is discussed below.

3.2.1 `yy_match`

This code attempts to find a valid match. It *consumes* the input as long as there are valid transitions. The transitions happen as follows:

1. `yy_ec` maintains an equivalence class among the 256 ASCII characters. This helps in reducing the size of the transition table.
2. When making transitions, if the current state is accepting, then this information is stored in `yy_last_accepting_state` and `yy_last_accepting_cpos` variables. This helps in backtrack when performing `yy_find_action` (discussed below).
3. Transitions happen at two different levels, one happens using `yy_def` and another happens using `yy_nxt`-
   i. `yy_def`
      - Input is not consumed when transitioning using this table.
      - This transition is used as long as `yy_chk` disallows `yy_nxt` transition.
      - If the current state obtained is ≥ 385 (0-indexed), the equialence class is updated using `yy_meta` (which is the meta-equivalence class).
   ii. `yy_nxt`
      - Input is consumed
      - Transition is done using the state obtained by `yy_base` with the offset of `yy_c` (the equivalence class of the input character).

3.2.2 `yy_find_action`

`yy_act` is used to store the action to be taken when at the current state. If `yy_act` is 0, it means that we are not at an accepting state and we need to backtrack to the last character position when we were at an accepting state. After this, we setup the `yytext` variable which we use to maintain the text for the token. We also update the line number if the rule that matched contains an EOL character.

3.2.3 `do_action`

Based on the value of `yy_act`, corresponding action is performed (token generation and/or text processing) as specified in the `c.l` file. This also has a case to handle (unexpected) end of buffer.