C String Tokenizer

Systems Programming | Section 5

Names:
Christopher Naporlee - cmn134
Michael Nelli - mrn73

Description

C String Tokenizer takes in a single string and parses it for words, numbers, symbols, and C keywords. *Words* are denoted by a token starting with a letter followed by any sequence of alphanumeric characters (a-z, 0-9). *Numbers* are denoted by a token starting with a number followed by any sequence of numeric characters only. *Symbols* are denoted by a token starting with any non-alphanumeric. *C keywords* are words that are used to denote special reserved words in C (if, while, do, switch, case, return, etc.). All tokens given within the input string will be printed back to the user with it's determined type. For example, given input string *"if array += 3"* the user will be shown:

if keyword: "if"
word: "array"
plus equals: "+="
decimal integer: "3"

How we achieved this is described in the *Design* section.

Design

Our main design uses a linked list data structure. Each node of the list holds a reference to the next node in the list and a pointer to a singular token. A linked list was chosen for the simplicity to easily add in and remove tokens. In contrast, if an array was chosen you would need to realloc() multiple times to account for adding in and removing multiple tokens. Since the number of tokens is not known at compile-time, a mutable list is a safer bet. The way this list is created and manipulated is described below in the following function calls:

1. **int main(int argc, char **argv)**
    - Receives input string from user to be parsed.
    - Holds a pointer to the head of the linked list structure of tokens.
    - Returns 1 if no characters are found, 0 otherwise.

2. **struct input_token *create_token_list(char *arg)**
    - Receives the input string from main(), separating the string by spaces until a null terminator is found. This is done by calling helper functions which create a new token out of the substrings found between spaces. These tokens will further be parsed later to ensure that a single token doesn't contain mismatched data types.
    - Searches for C comments (// and /*) and omits them.
    - Returns a pointer to the head of the new list.

3. **void sanitize(struct input_token **list)**
    - Receives a pointer to a pointer of the first node in the list.
    - Parses each token in the list to determine whether there are "unsanitized" inputs such as "123abc" which is a number and a word combined. If so, call split_token().
    - Has no return value.

4. **void split_token(struct input_token **token_node, int toklen)**
    - Receives a pointer to a pointer of the token node to be split and an integer toklen (token length).
    - Where the token is split is determined by the value passed in by toklen.
    - Visually, OriginalNode -> "123abc" ⇒ new_node1 -> "123" and new_node2 -> "abc".
    - Has no return value.

5. **void parse_tokens(struct input_token *list)**
    - Receives a pointer to the start of the list.
    - Goes through each node in the list looking at the first character of the string to determine whether it is a word, number, or symbol.
        i. Words are checked to see if it is a C keyword.
        ii. Numbers are checked to see if it is a hex, octal, float, or decimal integer.
        iii. Symbols are checked to see if it is a C symbol.
    - Has no return value.

These functions in conjunction successfully parse any users input string and successfully return the correct type. Examples of complicated test cases are shown in testcases.txt. The runtime of this algorithm is $O(n^2)$ because the program does one pass to create a linked list of the input, and another pass to ensure the tokens are sanitized for simple printing.

Challenges

1. Choosing the most appropriate data structure.
2. Dealing with combined tokens, such as “123abc”
3. Dealing with comments.

Notable Features

The program takes note of special character sequences within the input string. The two important ones are C comment sequences and C keywords. All C keywords (if, do, while, etc.) are given special names when recognized within the string, such as while keyword: “while”. If a C comment sequence is inputted, one of two things will happen. If ‘//’ is inputted, the program will discard all following characters until a newline character or the null terminating byte is found. If ‘/*’ is inputted, the program will discard all following characters until it’s closing ‘*/’ or the null terminating byte is found. These special instances can be examined within testcases.txt.