Systems Programming
Project #1 - A Real Tokenizer

David Lambropoulos
Demetrios Lambropoulos

Professor Brian Russell
February 11, 2015

Rutgers University
Department of Computer Science
School of Arts and Sciences

1 How to Use

To use simply given argument in the form of "[argument]". Anything that appears between these double quotes will be tokenized into a struct that will contain two char** arrays such that they will hold the individual tokens and the descriptions of those tokens. The first double char point will contain the tokens that are made from the arguments given to the program as the second char** will point to locations in memory containing the descriptions of the tokens that were made from the given argument. THIS PROGRAM WAS BUILT TO HANDLE ONLY ONE STATEMENT SURROUNDED BY DOUBLE QOUTES (one argument to argv). No token will be larger than 200 characters. If given a # everything after will be declared a preprocessor directive until it reaches the newline. ALL EXAMPLES OF RUNNING OCCUR IN THE TESTCASES.txt!

1.1 Binary Operators

The tokenizer will recognize binary operators.

1. * - Multiply
2. / - Divide
3. % - Modulus
4. + - Add
5. - - Subtract
6. >> - Shift Right
7. << - Shift Left
8. < - Less than
9. > - Greater than
10. <= - Less or equal
11. >= - Greater or equal
12. == - Equals
13. != - Not Equals
14. & - Bitwise and
15. | - Bitwise or
16. ˆ - Bitwise exclusive or
17. && - Logical and
18. || - Logical or

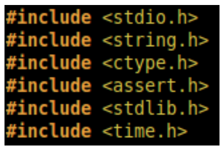
1.2 Conversion Characters

The tokenizer will recognize conversion characters.

1. %d - signed decimal integer conversion character
2. %u - unsigned decimal integer conversion character (used in printf only)
3. %x - unsigned hexidecimal integer conversion character
4. %h - unsigned short integer conversion character (used in scanf only)
5. %o - unsigned octal integer conversion character
6. %c - single character conversion character
7. %s - null terminated string conversion character
8. %f - fixed point notation for float or double conversion character
9. %e - scientific notation for float or double conversion character (printf only)
10. %g - use %e or %f, whichever is shorter conversion character (printf

2 Modularity/Functionality

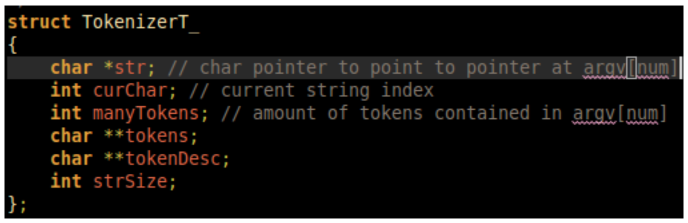
2.1 Packages Included



```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>
#include <stdlib.h>
#include <time.h>
```

1. stdio.h - used for printf
2. string.h - used for strcmp
3. ctype.h - used for isalphanum, isalpha and isdigit
4. assert.h - used to verify assumptions
5. stdlib.h - used for malloc, realloc, calloc, free, etc.
6. time.h - used to calculate execution time

2.2 Tokenizer Structure



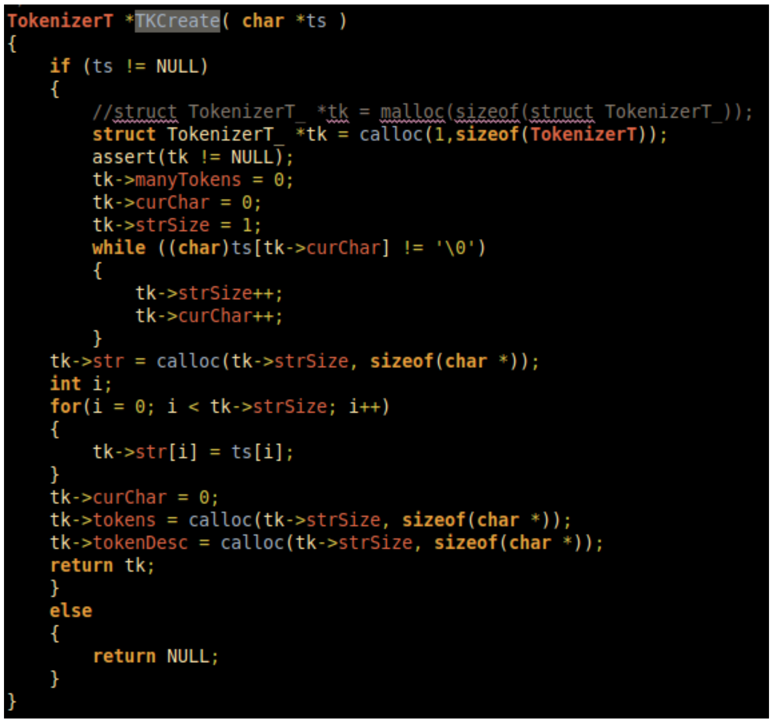
```
struct TokenizerT_
{
    char *str; // char pointer to point to pointer at argv[num]
    int curChar; // current string index
    int manyTokens; // amount of tokens contained in argv[num]
    char **tokens;
    char **tokenDesc;
    int strSize;
};
```

Internal Structure of Tokenizer

The char* str is essential a pointer that points to pointer at argv[nim], the curChar is an integer the is used as an index to walked through the string, manyTokens is an integer count of how many current tokens are in the array, char** is a pointer to pointers that contain tokens of the original string, char** tokenDesc is a pointer to pointers that contain the corresponding descriptions of the tokens, and int strSize holds an integer value of the size of the string being analyzed.

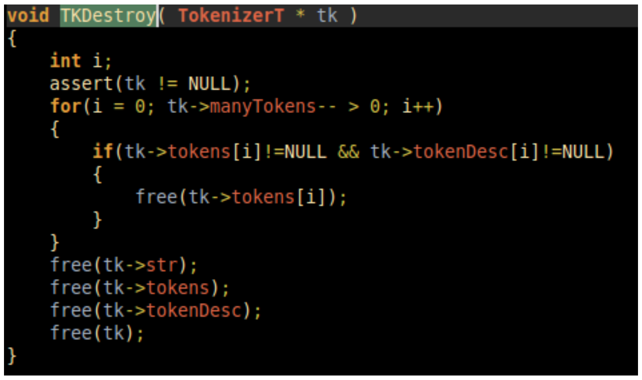
2.3 TKCreate

TKCreate creates a new TokenizerT object for a given token stream (given as a string). TKCreate should copy the arguments so that it is not dependent on them staying immutable after returning. (In the future, this may change to increase efficiency.) If the function succeeds, it returns a non-NULL TokenizerT. Else it returns NULL.



```
TokenizerT *TKCreate( char *ts )
{
    if (ts != NULL)
    {
        //struct TokenizerT_ *tk = malloc(sizeof(struct TokenizerT_));
        struct TokenizerT_ *tk = calloc(1,sizeof(TokenizerT));
        assert(tk != NULL);
        tk->manyTokens = 0;
        tk->curChar = 0;
        tk->strSize = 1;
        while ((char)ts[tk->curChar] != '\0')
        {
            tk->strSize++;
            tk->curChar++;
        }
    tk->str = calloc(tk->strSize, sizeof(char *));
    int i;
    for(i = 0; i < tk->strSize; i++)
    {
        tk->str[i] = ts[i];
    }
    tk->curChar = 0;
    tk->tokens = calloc(tk->strSize, sizeof(char *));
    tk->tokenDesc = calloc(tk->strSize, sizeof(char *));
    return tk;
    }
    else
    {
        return NULL;
    }
}
```

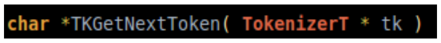
2.4 TKDestroy



```
void TKDestroy( TokenizerT * tk )
{
    int i;
    assert(tk != NULL);
    for(i = 0; tk->manyTokens-- > 0; i++)
    {
        if(tk->tokens[i]!=NULL && tk->tokenDesc[i]!=NULL)
        {
            free(tk->tokens[i]);
        }
    }
    free(tk->str);
    free(tk->tokens);
    free(tk->tokenDesc);
    free(tk);
}
```

TKDestroy destroys a TokenizerT object. It should free all dynamically allocated memory that is part of the object being destroyed.

2.5 TKGetNextToken



```
char *TKGetNextToken( TokenizerT * tk )
```

TKGetNextToken Function Header

TKGetNextToken returns the next token from the token stream as a character string. Space for the returned token should be dynamically allocated. The caller is responsible for freeing the space once it is no longer needed. If the function succeeds, it returns a C string (delimited by '/0') containing the token. Else it returns 0. It begins by calling $calloc(tk->strSize - tk->curChar, sizeof(char))$; This will assign for the first token the amount of space required to contain the entire string in the case that the first token the whole input is one token. The next time the function is used it will allocate the size of the string minus the amount of characters already surpassed. This will be followed by an if-statement that will check if $tk->str[tk->curChar]$ is a whitespace character. If a whitespace character is found the current character which contains the whitespace will be incremented over (e.g. $tk->curChar++$) so that the next time the method is called it will start after the whitespace character.

3 Extra Credit

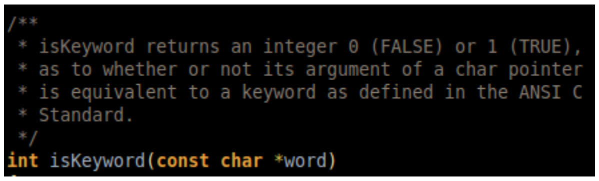
3.1 Keyword Recognition

This version of tokenizer recognizes keywords as defined in Standard ANSI C

1. auto
2. break
3. case
4. char
5. const
6. continue
7. default
8. do
9. double
10. else
11. enum
12. extern
13. float
14. for
15. goto

16. if
17. int
18. long
19. register
20. return
21. short
22. signed
23. sizeof
24. static
25. struct
26. switch
27. typedef
28. union
29. unsigned
30. void
31. volatile
32. while

This is accomplished by using the module added by us, 'isKeyword'. 'isKeyword' is a function that returns an int, either 0 (FALSE) or 1 (TRUE) as to whether or not a given word is a keyword. Upon finishing tokenizing individual words the words are then determined against a conditional block to determine whether their description is to be a word or a keyword. The function header of isKeyword is shown below.



```
/**
 * isKeyword returns an integer 0 (FALSE) or 1 (TRUE),
 * as to whether or not its argument of a char pointer
 * is equivalent to a keyword as defined in the ANSI C
 * Standard.
 */
int isKeyword(const char *word)
```

isKeyowrd function header

3.2 Single/Multi-Line Comment Pattern Matching and Removal

This version of tokenizer can determine the begining of a single line comment and it can also determine the begin and the end of a multi line comment and ignore all characters as follows in the comment. Our definition of a single line comment used in our pattern match is that a single line comment is any numbers of characters occuring after // is ignored as a comment until the end of the line. A single line comment will match '/' and increment the curChar index. Upon reading another '/' character the program will the begin to count everything that occurs after this point continue until it either see '/0' or '/n'. As for multi-line comments it checks for '/' followed by '*' and ignores until it sees a '*' followed by a '/' or until it reaches '/0'.

3.3 User-Defined Tokens (Strings)

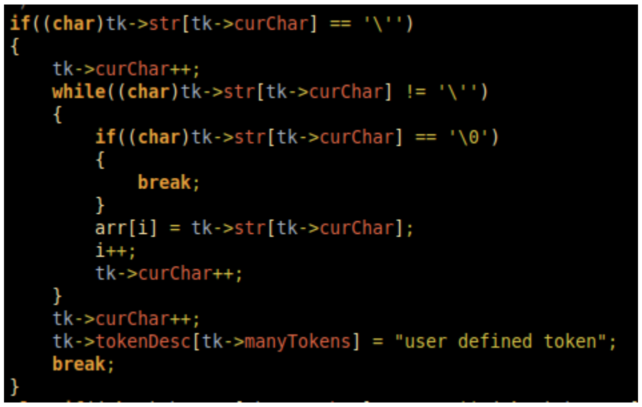
3.3.1 How To

A user can define a token by simply inclosing within single quotes (' ') within these qoutes no spaces and/or indentations will be ignored. These user defined tokens are no longer than 200 characters and are labeled as strings.

3.3.2 Methodology

Essentially if a single qoute is read, then the program enters a finite state machine that will continuesly keep walking through and appened new characters onto the token currently being worked on until another single qoute or a /0 is read in which case it will define the token as a user defined token and break from the finite state machine loop.

3.3.3 Algorithm/Implementation



```
if((char)tk->str[tk->curChar] == '\'')
{
    tk->curChar++;
    while((char)tk->str[tk->curChar] != '\'')
    {
        if((char)tk->str[tk->curChar] == '\0')
        {
            break;
        }
        arr[i] = tk->str[tk->curChar];
        i++;
        tk->curChar++;
    }
    tk->curChar++;
    tk->tokenDesc[tk->manyTokens] = "user defined token";
    break;
}
```

User-Defined Token Algorithm

4 END

Have Fun Tokenizing!! :)