Return of the tokenizer

This review project will show you how we may leverage what we have learned since the tokenizer program previously developed in order to modify it. We will start by rewriting parts of the sources without adding or removing features in order to improve the quality / flexibility / clarity. This is referred to as refactoring. Then we will start adding features.

Step #1: Refactoring – Adding a Tokens data structure

You probably noticed that, when calling a function meant to work on our array of tokens, it was fairly annoying to have to constantly specify the maximal number of tokens (the size) along with the array of pointers itself.

Let's define a simple structure *struct tokens* (and a typedef for it to be used as *Tokens*) in the *tokenizer.h* file which will encapsulate both the dynamical array of pointers (*tokens*) and its size (*maxnbtokens*).

```
struct tokens {
  char** tokens;
  int maxnbtokens;
};
typedef struct tokens Tokens;
```

The function *tokens_allocate* will now return a pointer on a newly allocated Tokens structure. All the other functions will now work on a pointer on such a structure instead of a char** parameter for the tokens and an int for the maximal number of tokens. We also adapted our *tests* to these alterations.

```
void        tokenize(Tokens* tokens, char * str, int length);
Tokens*     tokens_allocate(int maxnbtokens);
void        tokens_display(Tokens* tokens);
void        tokens_deallocate(Tokens* tokens);
```

Also, it's kind of silly to have to call *tokenize* with a user-provided string and its size, we will just let *tokenize* figure out the size itself with *strlen*.

All these modifications are in the source provided as step 1. It is a reafactored version of our previous mini tokinizer; i.e. no features have been added but the source quality has been improved. When you run it, you will re-apply the same tests we used while developing the first version. This process of re-applying a test-harness to ensure that we didn't degrade the correctness of the program is referred to as regression testing.

Step #2: Refactoring – Adding a Token data structure

Before we add a feature to this tokenizer, we are going to prepare the source by adding a new data structure. By making sure that our program is still working with this new data structure before to start adding features, we make it easy to improve our tokenizer in small, controlled, increments. Using regression testing often after small, well-defined, modifications is key to control the difficulty of a program.

So far, each token is simply a string which address is stored in the tokens array field in a Tokens data structure. We want to go one step further and be able to not only store the string representing the token but also an enumerate which will tell us what type of data the token represents. Here are the new data types definitions we will be using;

```
enum token_type {INTEGER, FLOATING, OPERATOR, TEXT, NOTSET};
typedef enum token_type TokenType;

struct token {
  char*         str;
  TokenType     type;
};
typedef struct token Token;

struct tokens {
  Token** tokens;
  int maxnbtokens;
};
typedef struct tokens Tokens;
```

The source provided to you as step 2 will show how the functions were rewritten to integrate the above modifications. The *TokenType* field of each *Token* data structure should be set initially to NOTSET value. We will be using the same tests than before to perform regression testing. However, we will also display the *TokenType* information to make sure all tokens are appropriately set.

Step #3: Feature – Let's start using this enumerate field

Let's now revisit our program so that we actually make use of this *TokenType* field. After the *tokenize* function is done identifying all the tokens in the user input, we are going to call a function *tokens_identify* which will look at each token and update the *TokenType* field with a value indicating what kind of data the string representing the token contains. You will do so by invoking a *token_identify* function which works on a single *Token*.

- *void tokens_identify (Tokens* p);*
- *void token_identity (Token* p);*

To determine whether that string contains an INTEGER, we use the function from the character handling library to ensure that each character of the string representing the token is a digit. If the string contains all digits but a single '.' then the type will be FLOATING. If the string contains only alphabetic characters, then the type will be TEXT. If the string has only one character which is '+', '-', '/' or '*' then it is an OPERATOR. Otherwise, the type will be UNKNOWN.

When we display all tokens, we make sure to check the value of the *TokenType* field and display what the identified type of the token is.

New tests are added so we may see our function identify the right *TokenTypes*. New features added means more tests needed.

Step #4: Feature –Numerical tokens into values?

Let's add to the *Token* data structure a field *value* which is a union of two fields; *integer* and *floating* of respective type *int* and *double*.

```
union token_value {
  int       integer;
  double floating ;

};
typedef union token_value TokenValue;

struct token {
  char*           str;
  TokenType     type;
  TokenValue    value;
};
typedef struct token Token;
```

These fields will be used in every token which identified type is respectively INTEGER or FLOATING. When the *token_identify* function determines the type of a given token's string to be either INTEGER or FLOATING, it will proceed to convert the contents of that string into a value of type *int* or *double* and store it inside the *value* union field's appropriate subfield (i.e. *integer* or *floating*).To make these conversions, use the sscanf function (check its man page).

Apply regression testing to ensure your program is still working appropriately and make sure that the display function now also displays the value of each token.

Step #5: Feature – Evaluating post-fix expressions

Let's add a function *double tokens_evaluate (Tokens* p);* which evaluates the expression typed by the user into a double value. This is assuming that the expression typed by the user is only made of tokens of the type OPERATOR, INTEGER or FLOATING and that it respects the syntax of postfix arithmetic expressions.

Before we go any further, let's explain what this means. Postfix notations allow you to write arithmetic expression by putting the operands first and then the operator. In our program, we will only handle binary operators (+, -, / and *) so this means that the following expressions should be written as indicated in the right column.

Infix notation		Postfix notation
3 + 6		3 6 +
3 + 6 / 9		3 6 9 / + 6 9 / 3 +

To keep this version easy, we will not worry about priorities. We will assume the user typed the expression in a way which allows us to simply evaluate it right to left. The evaluation will be done by the *tokens_evaluate* function with the following algorithm;

- Verify that the list of tokens only contains OPERATOR, INTEGER or FLOATING tokens (use functions *tokens_verify* and *token_verify* to do so).
- Start with the last token, if this token is a value, then you just got the value of your whole expression (ignore the other tokens)
- If that last token is an operator, then we will start evaluating. In order to do so we need to look at the next token. If it is a value, then you have one of your operand. You then look to the token before with the same logic.
- If at any point, one of your operands is not a value but an operator, then you need to apply this operator on its own operands and use the resulting value

This suggests that we can write a recursive algorithm to work on this evaluation. It also suggests that we are using our dynamical array of *Token** almost as a stack.

```
double tokens_evaluate ( Tokens* p, int* index )
if      index < 0
            no tokens left
            raise error: not enough tokens in expression

if      p->tokens[index] is a value
            return this value as a double

if      p->tokens[index] is an operator
            we look for 1st operand
                index --
                op1 = tokens_evaluate (p, &index);
            we look for 2nd operand
                index --
                op2 = tokens_evaluate ( p, & index);
            Apply the operator designated by p->tokens[index]
            to op1 and op2, return the resulting value as a double
```