Analisador Léxico - Compiladores I

Membros:

- Alexandre Saura
- Fábio Thomaz Vieira Junior
- Ricardo Freitas
- Rodrigo Suarez

Nosso projeto foi dividido em seis arquivos de implementação (.c) e cinco arquivos de headers diferentes, no qual o arquivo principal pela execução de todo o projeto é realizado através do arquivo "main.c".

A seguir serão descritas as principais funções de cada arquivo do projeto:

O arquivo "lexical_analyzer.c" realiza a análise léxica e verifica se o tipo de notação utilizada está no formato pós-fixa.

- Primeira função contida no "lexical_analyzer.", responsável por analisar os comandos enviados pelo usuário.

```
bool analyse(const char * expression) {
    Stack * stack;
    int type;
    char aux[ (int)(strlen(expression) / sizeof(char)) ];
    char symbol[ (int)(strlen(expression) / sizeof(char)) ];
    strcpy(aux, expression);
    strcpy(symbol, "");

    do {
        // printf(">>> %s\n", aux);
        if (!get_symbol(aux, symbol)) return true;

        type = check_type(aux, stack, symbol);
        printf("%10s ", symbol);

        switch (type) {
            case SYM_INTEGER:
                printf("=> número inteiro\n");
                break;
            case SYM_FLOAT:
                printf("=> ponto flutuante\n");
                break;
            case SYM_UNARY_OPERATOR:
                printf("=> operador unário\n");
                break;
            case SYM_BINARY_OPERATOR:
                printf("=> operador binário\n");
                break;
            case SYM_COMMAND:
                printf("=> comando\n");
                break;
            default:
                printf("=> erro\n");
                return false;
        }
        // printf("aux: %s\n", aux);
        // printf("l: %d\n", (int)(strlen(aux)/sizeof(char)));
    } while((int)(strlen(aux)/sizeof(char)) > 0);
```

```
    return true;
}
```

- A função acima dentro do comando de repetição "do while", chama em primeiro lugar a função "check_type", que recebe como parâmetros uma variável auxiliar, uma pilha e um símbolo e retorna um número inteiro que será utilizado dentro do switch case para definir qual foi o tipo do token lido. A seguir temos a função "check_type":

```
int check_type(char * expression, Stack * stack, char symbol[]) {
    char copy[ (int)(strlen(expression) / sizeof(char)) ];
    strcpy(copy, expression);

    int type = is_sym_number(copy, stack, symbol);
    if (type == -1) type = is_sym_operator(copy, stack, symbol);
    if (type == -1) type = is_sym_command(copy, stack, symbol);

    strcpy(expression, copy);

    return type;
}
```

- A função "check_type", irá chamar a função "is_sym_number" que será responsável por verificar se o valor inserido é um numero, e caso seja qual o tipo do número (inteiro, flutuante, etc.).

```
int is_sym_number(char expression[], Stack * stack, char symbol[]) {
    char original[ (int)(strlen(expression) / sizeof(char)) ];
    char number[ (int)(strlen(expression) / sizeof(char)) ];
    bool is_float = false; // Indica se é um ponto flutuante
    int counter = 1;       // Conta a quantidade de tokens no número
    strcpy(original, expression);
    strcpy(number, symbol);

    // Verifica se é um número com sinal
    if (strcmp(symbol, "+") == 0 || strcmp(symbol, "-") == 0) {
        get_symbol(expression, symbol);
        strcat(number, symbol);
        counter++;
    }

    // Verifica se há um número no início ou após o sinal
    if (!is_number2(symbol)) {
        strcpy(expression, original);
        strcpy(symbol, number);
        return -1;
    }

    // Enquanto for um número com menos de 9 dígitos, continua
    while (is_number2(symbol) && counter < 9 && (int)(strlen(expression)/sizeof(char)) > 0) {
        get_symbol(expression, symbol);
        strcat(number, symbol);
        counter++;
    }

    // Verifica se após o número há um ponto
    if (strcmp(symbol, ".") == 0) {
        is_float = true;
        get_symbol(expression, symbol);
        strcat(number, symbol);
        counter++;

        if (!is_number2(symbol)) {
            strcpy(expression, "");
```

```
            strcpy(symbol, number);
            return -1;
        }
    }

    // Enquanto for um número com menos de 9 dígitos, continua
    while (is_number2(symbol) && counter < 9 && (int)(strlen(expression)/sizeof(char)) > 0) {
        get_symbol(expression, symbol);
        strcat(number, symbol);
        counter++;
    }

    // Verifica se após o número há uma exponenciação de 10
    if (strcasecmp(symbol, "e") == 0) {
        get_symbol(expression, symbol);
        strcat(number, symbol);
        counter++;

        if (strcmp(symbol, "-") == 0 || strcmp(symbol, "+") == 0) {
            get_symbol(expression, symbol);
            strcat(number, symbol);
            counter++;
        }

        // Verifica se há um número após o sinal
        if (!is_number2(symbol)) {
            strcpy(expression, original);
            strcpy(symbol, number);
            return -1;
        }

        // Enquanto for um número com menos de 9 dígitos, continua
        while (is_number2(symbol) && counter < 9 && (int)(strlen(expression)/sizeof(char)) > 0) {
            get_symbol(expression, symbol);
            strcat(number, symbol);
            counter++;
        }
    }

    if (counter > 9) {
        printf("Erro! Muitos tokens em um mesmo número. Limite: 8.\n");
        strcpy(expression, "");
        strcpy(symbol, number);
        return -1;
    }

    // Reinserção do símbolo removido
    if (!is_number2(symbol)) {
        strcpy(original, "<");
        strcat(original, symbol);
        strcat(original, ">");
        strcat(original, expression);
        strcpy(expression, original);

        // Remoção do símbolo adicional
        int idx = strlen(number) - strlen(symbol);
        strncpy(symbol, number, idx);
        symbol[idx] = '\0';
    } else {
        strcpy(symbol, number);
    }

    return is_float ? SYM_FLOAT : SYM_INTEGER;
}
```

- A função "check_type", irá chamar a função "is_sym_operator" que será responsável por verificar se o valor inserido é um operador, e caso seja qual o tipo do operador.

```
int is_sym_operator(/*char expression[], Stack * stack,*/ char symbol[]) {
    char aux[((int) (strlen(symbol) / sizeof(char))) + 2];
    strcpy(aux, "");
    strcat(aux, "<");
    strcat(aux, symbol);
    strcat(aux, ">");

    if (!is_operator(aux)) return -1;
    if (strcasecmp(symbol, "sen") == 0 || strcasecmp(symbol, "cos") == 0) return SYM_UNARY_OPERATOR;
    return SYM_BINARY_OPERATOR;
}
```

- A função "check_type", irá chamar a função "is_sym_command" que será responsável por verificar se o valor inserido é um comando, e caso seja qual o tipo do comando.

```
int is_sym_command(/*char expression[], Stack * stack,*/ char symbol[]) {
    char aux[((int) (strlen(symbol) / sizeof(char))) + 2];
    strcpy(aux, "");
    strcat(aux, "<");
    strcat(aux, symbol);
    strcat(aux, ">");

    if (!is_command(aux)) return -1;
    return SYM_COMMAND;
}
```

- Dentro de cada umas das funções descritas acima, três funções complementares são utilizadas.
  - A função "get_symbol" é responsável por buscar pelo token da string inserida.

```
bool get_symbol(char expression[], char symbol[]) {
    char * startAtPtr = strstr(expression, "<"); // Ponteiro para o primeiro '<'
    char * endAtPtr = strstr(expression, ">");   // Ponteiro para o primeiro '>'
    int startAt = (startAtPtr == NULL ? -1 : startAtPtr - expression); // Índice para o primeiro '<'
    int endAt = (endAtPtr == NULL ? -1 : endAtPtr - expression);       // Índice para o primeiro '>'

    if (startAt != 0) return false;
    if (endAt == -1 || startAt == -1) return false;
    if (startAt > endAt) return false;

    memset(symbol, 0, strlen(symbol));
    strncpy(symbol, expression + 1, ++endAt - 2);
    symbol[endAt - 2] = '\0';

    if ((int)(strlen(expression) / sizeof(char) - endAt) == 0) {
        strcpy(expression, "");
    } else {
        char aux[(int)(strlen(expression)/sizeof(char)) - endAt];
        strncpy(aux, expression + endAt, (int)(strlen(expression)/sizeof(char)));
        strcpy(expression, aux);
    }

    return true;
}
```

  - A função "is_number2", verifica se é um símbolo:

```
bool is_number2(const char symbol[]) {
    return strstr("0123456789", symbol) != NULL;
```

```
}
```

- A função "put_token" coloca o token na string passada como parâmetro da string:

```
void put_token(char expression[], const char token[]) {
    char aux[(int)((strlen(expression)+strlen(token)+2)/sizeof(char))];

    memset(aux, 0, strlen(aux));

    strcat(aux, "<");
    strncat(aux, token, (int)(strlen(token)/sizeof(char)));
    strcat(aux, ">");

    strncat(aux, expression, (int)(strlen(expression)/sizeof(char)));

    memset(expression, 0, strlen(expression));

    strcpy(expression, aux);
}
```

O arquivo "queue.c" é responsável pela implementação de diversas funções relacionadas manipulação da fila.

```
//queue.c
#include "queue.h"
#include <stdlib.h> // malloc
#include <string.h> // strcpy

// Cria uma fila
Queue * create_queue() {
    Queue * queue = (Queue *)malloc(sizeof(Queue));
    queue->top = 0;
    queue->end = -1;
    queue->nTokens = 0;
    queue->limit = MAX_TOKENS;

    return queue;
}

// Insere um novo elemento na fila
void enqueue(Queue * queue, char token[]) {
    if(queue->end == queue->limit - 1)
        queue->end = -1;

    queue->end++;
    strcpy(queue->tokens[queue->end], token);
    queue->nTokens++;
}
```

O arquivo "stack.c" é responsável pela implementação de diversas funções relacionadas a pilha: criação da pilha e inserir uma elemento na pilha, remover um elemento, verificar se a pilha está cheia ou vazia e exibir a pilha.

```
//stack.h
#include "stack.h"
#include <stdio.h>  // printf, sizeof
#include <string.h> // strcpy
#include <stdlib.h> // malloc
```

```
// Cria uma pilha
Stack * create_stack() {
    Stack * stack = (Stack *)malloc(sizeof(Stack));
    stack->top = -1;
    stack->limit = MAX_TOKENS;

    return stack;
}

// Verifica se a pilha está vazia
bool is_empty_stack(Stack * stack) {
    return stack->top == -1;
}

// Verifica se a pilha está cheia
bool is_full_stack(Stack * stack) {
    return stack->top == stack->limit - 1;
}

// Insere um novo elemento na pilha
void push(Stack * stack, char * token) {
    if(!is_full_stack(stack)) {
        strcpy(stack->tokens[++stack->top], token);
    }
}

// Remove um elemento da pilha
char * pop(Stack * stack) {
    if(is_empty_stack(stack)) {
        printf("Pilha vazia.\n");
        return "";
    }

    return stack->tokens[stack->top--];
}

void print_stack(Stack * stack) {
    while(!is_empty_stack(stack)) {
        printf("desimpilha: %s\n", pop(stack));
    }
}
```

O arquivo "token.c" é responsável por implementar diversas funções responsáveis por:

- Varrer a string e quebra-la em tokens no formato "<informação>";
- Buscar pelo primeiro token da string, caso seja um token válido, retorna "true";
- Verificar se o token é válido;
- Verificar se o token é um número ou um dígito que compõe um número;
- Verificar se o token é um operador;
- Verifica se o token é um comando.

```
//token.c

#include <stdio.h>   // printf,
#include <string.h>  // strlen, strcpy, strstr, memset, strncpy, strchr
#include <strings.h> // strncasecmp

#include "token.h"

// Varre a string, quebrando-a em tokens, por exemplo, no formato <number>
bool check_tokens(const char * expression) {
    char aux[(int)(strlen(expression)/sizeof(char))];
```

```
    char token[(int)(strlen(expression)/sizeof(char))];
    strcpy(aux, expression);

    do {
        if (!get_token(aux, token)){
            printf("Insira tokens válidos!\n");
            return false;
        }

        if (!is_valid_token(token)) {
            printf("O token '%s' é inválido!\n", token);
            return false;
        }
    } while(( (int)(strlen(aux)/sizeof(char)) ) > 0);
    return true;
}

// Busca pelo primeiro token da string, caso seja um token válido, retorna true
bool get_token(char expression[], char token[]) {
    char * startAtPtr = strstr(expression, "<"); // Ponteiro para o primeiro '<'
    char * endAtPtr = strstr(expression, ">");   // Ponteiro para o primeiro '>'
    int startAt = (startAtPtr == NULL ? -1 : startAtPtr - expression); // Índice para o primeiro '<'
    int endAt = (endAtPtr == NULL ? -1 : endAtPtr - expression);       // Índice para o primeiro '>'

    if (startAt != 0) return false;
    if (endAt == -1 || startAt == -1) return false;
    if (startAt > endAt) return false;

    memset(token, 0, strlen(token));     // Limpa o conteúdo do token
    strncpy(token, expression, ++endAt); // Copia o token da expressão para a variável
    token[endAt] = '\0';                 // Força o fim da string

    // Verifica se sobraram caractéres na string (expressão)
    if ((int)(strlen(expression) / sizeof(char) - endAt) == 0) {
        strcpy(expression, "");
    } else {
        // Remove o token da string (expressão)
        char aux[(int)(strlen(expression)/sizeof(char)) - endAt];
        strncpy(aux, expression + endAt, (int)(strlen(expression)/sizeof(char)));
        strcpy(expression, aux);
    }

    return true;
}

// Verifica se o token é válido
bool is_valid_token(const char token[]) {
    if(is_number(token)) return true;
    if(is_operator(token)) return true;
    if(is_command(token)) return true;

    return false;
}

// Verifica se o token é um número ou um dígito que compõe um número
bool is_number(const char token[]) {
    if( ((int)(strlen(token) / sizeof(char))) > 3) return false;

    char digit = (char)token[1];
    return strchr("0123456789Ee.", digit) != NULL;
}

// Verifica se o token é um operador
bool is_operator(const char token[]) {
    if( ((int)(strlen(token) / sizeof(char))) > 5) return false;

    char operators[8][4] = {"+", "-", "*", "/", "^", "log", "sen", "cos"};
    int i;

    for(i = 0; i < 8; i++){
        if(strlen(token) - 2 != strlen(operators[i])) continue;
        if(strncasecmp(token + 1, operators[i], strlen(operators[i])) == 0) return true;
    }
```

```
    return false;
}

// Verifica se o token é um comando
bool is_command(const char token[]) {
    if( ((int)(strlen(token) / sizeof(char))) > 7) return false;

    char commands[1][6] = {"enter"};
    int i;

    for(i = 0; i < 1; i++){
        if(strlen(token) - 2 != strlen(commands[i])) continue;
        if(strncasecmp(token + 1, commands[i], strlen(commands[i])) == 0) return true;
    }

    return false;
}
```

O arquivo "util.c" é responsável por implantar uma função responsável por ler uma sequência de carácteres do teclado e criar uma string.

```
//util.c
#include "util.h"
#include <stdlib.h> // size_t, realloc
#include <ctype.h>  // isspace, tolower
#include <stdio.h>  // getchar

char * read_string() {
    char * str = NULL, ch;
    size_t size = 0;
    int i = 0;

    // Lê os caractéres enquanto não encontrar um ENTER
    while((ch = getchar()) != '\n') {
        if(isspace(ch)) continue;
        str = realloc(str, size);
        size += sizeof(char);
        str[i++] = tolower(ch);
    }

    str[i] = '\0';

    return str;
}
```

O arquivo "main.c" é o arquivo que será executado no nosso projeto e este irá chamar as devidas implementações para a execução das funcionalidades propostas pela atividade.

```
//main.c
#include <stdlib.h> // free
#include <stdio.h>  // printf

#include "util.h"             // read_string
#include "token.h"            // check_tokens
#include "lexical_analyzer.h" // analyse

int main() {
    printf("Exemplo de expressão: <4><.><2><*><7><+><log><8>\n");
    printf("Insira sua expressão: ");

    char * expression;
```

```
    expression = read_string();

    printf("\n");

    if(!check_tokens(expression)) return 1;
    if(!analyse(expression)) {
        printf("\nEntrada inválida!\n");
        return 1;
    }

    free(expression);
    return 0;
}
```