

```
/**
 * Arquivo: scannerTAD.c
 * Versão : 1.0
 * Data   : 2024-10-18 09:14
 * -----
 * Este arquivo implementa a interface scannerTAD.h
 *
 * Baseado em: Programming Abstractions in C, de Eric S. Roberts.
 *             Capítulo 8: Abstract Data Types (pg. 347-358).
 *
 * Prof.: Abrantes Araújo Silva Filho (Computação Raiz)
 *        www.computacaoraiz.com.br
 *        www.youtube.com.br/computacaoraiz
 *        github.com/computacaoraiz
 *        twitter.com/ComputacaoRaiz
 *        www.linkedin.com/company/computacaoraiz
 *        www.abrantes.pro.br
 *        github.com/abrantesasf
 */

/** Includes */

#include <ctype.h>
#include "genlib.h"
#include "scannerTAD.h"
#include <stdio.h>
#include <stdlib.h>
#include "strlib.h"

/** Declarações de Subprogramas Privados */

static void pular_espacos (scannerTAD scanner);
static int achar_final_do_token (scannerTAD scanner);

/** Tipos de Dados */

/**
 * TIPO: scannerTCD
 * -----
 * Esta struct é a representação concreta do tipo abstrato scannerTAD, exportado
 * pela interface. O objetivo deste tipo de dado é manter o estado do scanner
 * entre as chamadas ao scanner. Os membros da struct são:
 *
 *      str          -- cópia da string passada ao scanner
 *      tam          -- tamanho da string
 *      pa           -- posição do caractere atual na string
 *      token_salvo  -- token salvo pelo cliente (NULL se não houver)
 *      opcao_de_espacos -- opção para ignorar/considerar espaços como tokens
 */

struct scannerTCD
{
    string str;
    int tam;
    int pa;
    string token_salvo;
    int opcao_de_espacos;
};

/** Definições de Subprogramas Exportados */
```

```
/**
 * FUNÇÃO: criar_scanner
 * Uso: scanner = criar_scanner( );
 * -----
 * Cria uma instância do scanner. Retorna NULL se erro.
 */

scannerTAD criar_scanner (void)
{
    scannerTAD S = malloc(sizeof(struct scannerTCD));
    if (S == NULL)
    {
        fprintf(stderr, "Erro: impossível criar o scanner.\n");
        return NULL;
    }
    S->str = NULL;
    S->tam = 0;
    S->pa = 0;
    S->token_salvo = NULL;
    S->opcao_de_espacos = 0;
    return S;
}

/**
 * PROCEDIMENTO: remover_scanner
 * Uso: remover_scanner(&scanner);
 * -----
 * Recebe um PONTEIRO para um scannerTAD (um ponteiro para ponteiro para
 * scannerTCD) e libera todas as estruturas de memória alocadas.
 */

void remover_scanner(scannerTAD *scanner)
{
    if (*scanner != NULL)
    {
        free((*scanner)->str);
        free((*scanner)->token_salvo);
        free(*scanner);
        *scanner = NULL;
    }
}

/**
 * PROCEDIMENTO: ler_string
 * Uso: ler_string(scanner, str);
 * -----
 * Aloca a string str para o scannerTAD scanner, fazendo uso das funções da
 * biblioteca CSLIB, e "zerando" a condição inicial do scanner (exceto a
 * opção de tratamento de espaços).
 *
 * A alocação da string passada pelo usuário é feita através da função
 * CopyString, para preservar a separação entre o cliente e a implementação.
 * Como a implementação não tem nenhum controle sobre o quê o usuário passará
 * como string, para evitar receber um ponteiro para str (que pode ser alterado
 * pelo usuário a qualquer momento) fazemos uma cópia da string. Nesse caso o
 * usuário pode alterar a string original, sem causar efeito no scanner (nesse
 * caso, se o usuário alterar a string original, será forçado a passar a nova
 * string para o scanner).
 */
```

```
void ler_string (scannerTAD scanner, string str)
{
    if (scanner == NULL)
    {
        fprintf(stderr, "Erro: impossível atribuir string a scanner NULL.\n");
        exit(1);
    }

    if (scanner->str != NULL)
    {
        free(scanner->str);
        if (scanner->token_salvo != NULL)
            free(scanner->token_salvo);
    }

    scanner->str = CopyString(str);
    scanner->tam = StringLength(str);
    scanner->pa = 0;
    scanner->token_salvo = NULL;
}

/**
 * FUNÇÃO: obter_token
 * Uso: token = obter_token(scanner);
 * -----
 * Retorna o próximo token da string armazenada no scannerTAD. Se já houver um
 * token salvo, retorna o token salvo. Se não houver mais nenhum token a ser
 * lido, retorna a string vazia.
 *
 * No caso da função retornar a string vazia, também é utilizada a função
 * CopyString. Por quê? Por dois motivos principais:
 *
 * 1. Proteger a implementação de clientes descuidados ou maliciosos.
 *    Retornar um ponteiro para memória alocada em seu próprio domínio
 *    é uma falha de segurança pois permite ao cliente sobrescrever o
 *    valor apontado por esse ponteiro.
 *
 * 2. Para permitir que o cliente passar utilizar o procedimento
 *    liberar_token em qualquer token retornado por esta função. Note que
 *    esta função utilizar a SubString para retornar um token, e essa função
 *    sempre aloca um novo espaço de armazenamento (fora do domínio desta
 *    implementação). Alocando um espaço também para a string vazia
 *    garantimos que TODOS os tokens que podem ser retornados por esta
 *    função estão alocados da mesma maneira e podem ser liberados da mesma
 *    maneira (com a função liberar_token).
 */

string obter_token (scannerTAD scanner)
{
    char c;
    string token;
    int inicio, fim;

    if (scanner == NULL)
    {
        fprintf(stderr, "Erro: lendo token de scanner NULL.\n");
        exit(1);
    }
}
```

```
else if (scanner->str == NULL)
{
    fprintf(stderr, "Erro: scanner não recebeu linha.\n");
    exit(1);
}
else if (scanner->token_salvo != NULL)
{
    token = scanner->token_salvo;
    scanner->token_salvo = NULL;
    return token;
}

if (scanner->opcao_de_espacos == 1)
    pular_espacos(scanner);

inicio = fim = scanner->pa;
if (inicio >= scanner->tam)
    return CopyString("");

c = scanner->str[scanner->pa];
if (isalnum(c))
    fim = achar_final_do_token(scanner);
else
    scanner->pa++;

return SubString(scanner->str, inicio, fim);
}

/**
 * PROCEDIMENTO: liberar_token
 * Uso: liberar_token(&token);
 * -----
 * Este procedimento recebe um PONTEIRO para um token (um ponteiro para um
 * ponteiro para char) e libera a memória alocada para o token.
 */

void liberar_token (string *token)
{
    if (*token != NULL)
    {
        free(*token);
        *token = NULL;
    }
}

/**
 * PREDICADO: existe_outro_token
 * Uso: if (existe_outro_token( )) . . .
 * -----
 * Retorna TRUE se ainda houver algum token para ser retornado.
 */

bool existe_outro_token (scannerTAD scanner)
{
    if (scanner == NULL)
    {
        fprintf(stderr, "Erro: scanner NULL.\n");
        exit(1);
    }
    else if (scanner->str == NULL)
```

```
{
    fprintf(stderr, "Erro: scanner não recebeu linha.\n");
    exit(1);
}

if (scanner->token_salvo != NULL)
    return (!StringEqual(scanner->token_salvo, ""));

if (scanner->opcao_de_espacos == 1)
    pular_espacos(scanner);

return (scanner->pa < scanner->tam);
}

/**
 * PROCEDIMENTO: salvar_token
 * Uso: salvar_token(scanner, token);
 * -----
 */

void salvar_token (scannerTAD scanner, string token)
{
    if (scanner == NULL)
    {
        fprintf(stderr, "Erro: scanner NULL.\n");
        exit(1);
    }
    else if (scanner->str == NULL)
    {
        fprintf(stderr, "Erro: scanner não recebeu linha.\n");
        exit(1);
    }
    else if (scanner->token_salvo != NULL)
    {
        fprintf(stderr, "Erro: já existe um token salvo.\n");
        exit(1);
    }

    scanner->token_salvo = token;
}

/**
 * PROCEDIMENTO: configurar_tratamento_de_espacos
 * Uso: configurar_tratamento_de_espacos(scanner, opcao);
 * -----
 * As opções para o controle do tratamento de espaços são representadas por
 * números inteiros:
 *
 * 0 = considerar os espaços em branco como tokens
 * 1 = ignorar os espaços em branco
 *
 * Atenção: qualquer valor diferente de 0 ou 1 é entendido como inválido e,
 * nesse caso, o scanner volta para seu comportamento padrão (considerar os
 * espaços como tokens válidos).
 */

void configurar_tratamento_de_espacos (scannerTAD scanner, int opcao)
{
    if (scanner == NULL)
    {
```

```
        fprintf(stderr, "Erro: scanner NULL.\n");
        exit(1);
    }

    if (opcao == 1)
        scanner->opcao_de_espacos = 1;
    else
        scanner->opcao_de_espacos = 0;
}

/**
 * FUNÇÃO: obter_tratamento_de_espacos
 * Uso: opcao = obter_tratamento_de_espacos(scanner);
 * -----
 */

int obter_tratamento_de_espacos (scannerTAD scanner)
{
    if (scanner == NULL)
    {
        fprintf(stderr, "Erro: scanner NULL.\n");
        exit(1);
    }

    return scanner->opcao_de_espacos;
}

/**/
Definições de Subprogramas Privados
/**/

/**
 * PROCEDIMENTO: pular_espacos
 * Uso: pular_espacos(scanner);
 * -----
 * Este procedimento "pula" os espaços em branco em uma string, avançando a
 * posição atual até que ela não esteja sob um caractere branco.
 */

static void pular_espacos (scannerTAD scanner)
{
    while (isspace(scanner->str[scanner->pa]))
        scanner->pa++;
}

/**
 * FUNÇÃO: achar_final_do_token
 * Uso: final = achar_final_do_token(scanner);
 * -----
 * Esta função avança a posição atual do scanner até que ela alcance o final de
 * uma sequência de letras ou dígitos que forma um identificador (token). O
 * valor de retorno é a posição (índice) do último caractere no identificador;
 * o valor da posição atual que ficará no scanner será o primeiro caractere
 * posterior ao retorno.
 */

static int achar_final_do_token (scannerTAD scanner)
{
    while (isalnum(scanner->str[scanner->pa]))
        scanner->pa++;

    return (scanner->pa - 1);
}
```

}