PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA E DE ARTES
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INTERPRETADOR DBASE EM C - DOCUMENTAÇÃO

LEONARDO DE MOURA ALVES

GOIÂNIA
2024

LEONARDO DE MOURA ALVES

INTERPRETADOR DBASE EM C - DOCUMENTAÇÃO

Trabalho apresentado à disciplina CMP1076 – Compiladores, do curso de Bacharel em Ciência da Computação, da Escola Politécnica e de Artes da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para aprovação nesta disciplina.
Orientador:
Prof. Me. Claudio Martins Garcia

GOIÂNIA

2024

1 INTRODUÇÃO

Foi realizado neste trabalho o desenvolvimento de um interpretador de DBASE em linguagem C, com a finalidade de ler e executar comandos.

Esse interpretador processa um arquivo de texto contendo instruções em DBASE e é capaz de interpretar e executar os comandos CREATE TABLE, INSERT INTO e SELECT * FROM.

Além da execução dos comandos, também foi implementado uma verificação de sintaxe para garantir a integridade dos dados e prevenir possíveis erros na execução das operações.

2 CÓDIGO COMPLETO

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

// Definição dos tamanhos máximos de campos e registros
#define MAX_FIELDS 10
#define MAX_RECORDS 100
#define MAX_FIELD_LENGTH 50

// Estrutura para definir um campo de tabela (nome e tipo)
typedef struct {
    char field_name[MAX_FIELD_LENGTH];  // Nome do campo
    char field_type;                    // Tipo do campo (C ou N)
} Field;

// Estrutura para definir uma tabela com campos e registros
typedef struct {
    char table_name[MAX_FIELD_LENGTH];  // Nome da tabela
    Field fields[MAX_FIELDS];           // Array de campos da tabela
```

```
    char records[MAX_RECORDS][MAX_FIELDS][MAX_FIELD_LENGTH]; // Array de registros
    int field_count;                // Número de campos na tabela
    int record_count;               // Número de registros na tabela
} Table;

Table tables[10];                   // Array de tabelas
int table_count = 0;                // Contador de tabelas

void show_table(Table *table);          // Função para exibir dados de uma tabela
int execute_command(const char *input); // Função para executar comandos
Table* get_table_by_name(const char *name); // Função para buscar uma tabela por nome

// Função que exibe a tabela e seus registros
void show_table(Table *table) {
    // Imprime o nome de cada campo
    for (int i = 0; i < table->field_count; i++) {
        printf("%s\t", table->fields[i].field_name);
    }
    printf("\n");

    // Imprime cada registro
    for (int i = 0; i < table->record_count; i++) {
        for (int j = 0; j < table->field_count; j++) {
            printf("%s\t", table->records[i][j]);
        }
        printf("\n");
    }
}

// Função para buscar uma tabela pelo nome
Table* get_table_by_name(const char *name) {
```

```
    for (int i = 0; i < table_count; i++) {
        if (strcmp(tables[i].table_name, name) == 0) {
            return &tables[i];  // Retorna o ponteiro para a tabela
        }
    }
    return NULL; // Retorna NULL se a tabela não for encontrada
}

// Definição de tipos de tokens para análise léxica
typedef enum {
    TOKEN_EOF,
    TOKEN_CREATE,
    TOKEN_TABLE,
    TOKEN_INSERT,
    TOKEN_INTO,
    TOKEN_SELECT,
    TOKEN_FROM,
    TOKEN_STRING,
    TOKEN_NUMBER,
    TOKEN_COMMA,
    TOKEN_LPAREN,  // (
    TOKEN_RPAREN,  // )
    TOKEN_SEMICOLON,
    TOKEN_ASTERISK // *
} TokenType;

// Estrutura para definir um token (tipo e valor)
typedef struct {
    TokenType type;
    char value[MAX_FIELD_LENGTH];
} Token;

// Função que extrai o próximo token da entrada
```

```
Token get_token(const char **input) {
    Token token;
    token.type = TOKEN_EOF;
    token.value[0] = '\0';

    // Ignora espaços, tabulações e novas linhas
    while (**input == ' ' || **input == '\n' || **input == '\t' || **input == '\r') {
        (*input)++;
    }

    // Verifica palavras-chave e símbolos e atribui o tipo de token
    if (strncmp(*input, "CREATE", 6) == 0) {
        (*input) += 6;
        token.type = TOKEN_CREATE;
    } else if (strncmp(*input, "TABLE", 5) == 0) {
        (*input) += 5;
        token.type = TOKEN_TABLE;
    } else if (strncmp(*input, "INSERT", 6) == 0) {
        (*input) += 6;
        token.type = TOKEN_INSERT;
    } else if (strncmp(*input, "INTO", 4) == 0) {
        (*input) += 4;
        token.type = TOKEN_INTO;
    } else if (strncmp(*input, "SELECT", 6) == 0) {
        (*input) += 6;
        token.type = TOKEN_SELECT;
    } else if (strncmp(*input, "FROM", 4) == 0) {
        (*input) += 4;
        token.type = TOKEN_FROM;
    } else if (**input == ',') {
        (*input)++;
        token.type = TOKEN_COMMA;
    } else if (**input == '(') {
```

```
        (*input)++;
        token.type = TOKEN_LPAREN;
    } else if (**input == ')') {
        (*input)++;
        token.type = TOKEN_RPAREN;
    } else if (**input == ';') {
        (*input)++;
        token.type = TOKEN_SEMICOLON;
    } else if (**input == '*') {
        (*input)++;
        token.type = TOKEN_ASTERISK;
    } else if (isdigit(**input)) {
        int i = 0;
        while (isdigit(**input)) {
            token.value[i++] = *(*input)++;
        }
        token.value[i] = '\0';
        token.type = TOKEN_NUMBER;
    } else if (isalpha(**input)) {
        int i = 0;
        while (isalnum(**input) || **input == '_') {
            token.value[i++] = *(*input)++;
        }
        token.value[i] = '\0';
        token.type = TOKEN_STRING;
    }
    return token;
}

// Função para validar o tipo de dado de um valor inserido
int validate_field_value(const char *value, char field_type) {
    if (field_type == 'N') { // Campo numérico
        for (int i = 0; value[i] != '\0'; i++) {
```

```
            if (!isdigit(value[i])) {
                return 0;  // valor inválido
            }
        }
    } else if (field_type == 'C') { // Campo de texto
        for (int i = 0; value[i] != '\0'; i++) {
            if (!isalpha(value[i])) {  // Aceita apenas letras
                return 0;  // valor inválido
            }
        }
    }
    return 1; // valor válido
}

// Função para processar o comando CREATE TABLE
int parse_create_table(const char **input) {
    Token token = get_token(input);
    if (token.type == TOKEN_STRING) {
        Table *table = &tables[table_count++];
        strcpy(table->table_name, token.value);
        table->field_count = 0;
        token = get_token(input);
        if (token.type == TOKEN_LPAREN) {
            int field_index = 0;
            do {
                token = get_token(input);
                if (token.type == TOKEN_STRING) {
                    strcpy(table->fields[field_index].field_name, token.value);
                    token = get_token(input);
                    if (strcmp(token.value, "C") == 0) {
                        table->fields[field_index].field_type = 'C';
                    } else if (strcmp(token.value, "N") == 0) {
                        table->fields[field_index].field_type = 'N';
```

```
                } else {
                    printf("Erro de sintaxe: Tipo de campo inválido.\n");
                    return 1;
                }
                field_index++;
                table->field_count++;
            }
            token = get_token(input);
        } while (token.type == TOKEN_COMMA);
        if (token.type != TOKEN_RPAREN) {
            printf("Erro de sintaxe: Esperado ')'.\n");
            return 1;
        }
    } else {
        printf("Erro de sintaxe: Esperado '('.\n");
        return 1;
    }
} else {
    printf("Erro de sintaxe: Nome da tabela esperado.\n");
    return 1;
}
return 0;
}

// Função para processar o comando INSERT INTO, incluindo validação de tipo
int parse_insert_into(const char **input) {
    Token token = get_token(input);
    if (token.type == TOKEN_STRING) {
        Table *table = get_table_by_name(token.value);
        if (!table) {
            printf("Erro: Tabela '%s' não encontrada.\n", token.value);
            return 1;
        }
```

```
        token = get_token(input);
        if (token.type == TOKEN_LPAREN) {
            int record_index = table->record_count;
            int field_index = 0;
            do {
                token = get_token(input);
                if (token.type == TOKEN_STRING || token.type == TOKEN_NUMBER) {
                    // Valida o valor conforme o tipo do campo
                    if (field_index < table->field_count &&
                        !validate_field_value(token.value, table->fields[field_index].field_type))
{
                        printf("Erro de sintaxe: Tipo de valor inválido para o campo '%s'.\n",
                               table->fields[field_index].field_name);
                        return 1;
                    }
                    if (field_index < table->field_count) {
                        strcpy(table->records[record_index][field_index], token.value);
                        field_index++;
                    }
                }
                token = get_token(input);
            } while (token.type == TOKEN_COMMA);

            table->record_count++;
            printf("Registro inserido com sucesso! (Registro %d)\n", record_index + 1);
        } else {
            printf("Erro de sintaxe: Esperado '('.\n");
            return 1;
        }
    } else {
        printf("Erro de sintaxe: Nome da tabela esperado.\n");
        return 1;
    }
```

```
    return 0;
}

// Função para processar o comando SELECT
int parse_select_from(const char **input) {
    Token token = get_token(input);
    if (token.type == TOKEN_ASTERISK) {
        token = get_token(input);
        if (token.type == TOKEN_FROM) {
            token = get_token(input);
            if (token.type == TOKEN_STRING) {
                Table *table = get_table_by_name(token.value);
                if (table) {
                    show_table(table);
                    token = get_token(input);  // Verifica se há um ';' após SELECT
                    if (token.type != TOKEN_SEMICOLON && token.type != TOKEN_EOF) {
                        printf("Erro de sintaxe: Comando inesperado após SELECT.\n");
                        return 1;
                    }
                    return 0;
                } else {
                    printf("Erro: Tabela '%s' não encontrada.\n", token.value);
                    return 1;
                }
            } else {
                printf("Erro de sintaxe: Nome da tabela esperado após FROM.\n");
                return 1;
            }
        } else {
            printf("Erro de sintaxe: Esperado 'FROM'.\n");
            return 1;
        }
    } else {
```

```
        printf("Erro de sintaxe: Esperado '*'.\n");
        return 1;
    }
}

// Função que executa o comando baseado na entrada
int execute_command(const char *input) {
    const char *ptr = input;
    Token token = get_token(&ptr);
    if (token.type == TOKEN_CREATE) {
        token = get_token(&ptr);
        if (token.type == TOKEN_TABLE) {
            if (parse_create_table(&ptr) != 0) return 1;
            printf("Tabela criada com sucesso!\n");
        }
    } else if (token.type == TOKEN_INSERT) {
        token = get_token(&ptr);
        if (token.type == TOKEN_INTO) {
            if (parse_insert_into(&ptr) != 0) return 1;
        }
    } else if (token.type == TOKEN_SELECT) {
        if (parse_select_from(&ptr) != 0) return 1;
    } else {
        printf("Erro de sintaxe: Comando desconhecido.\n");
        return 1;
    }
    return 0;
}

// Função principal que lê o arquivo e executa os comandos
int main() {
    table_count = 0;
```

```
char filename[100];

printf("\nInterpretador DBASE\n");
printf("CMP1076 - COMPILADORES\n");
printf("Aluno: Leonardo de Moura Alves\n\n");

printf("Informe o nome do arquivo .txt com os comandos: ");
scanf("%99s", filename);

FILE *file = fopen(filename, "r");
if (!file) {
    printf("Erro ao abrir o arquivo.\n");
    return 1;
}

char line[256];
while (fgets(line, sizeof(line), file)) {
    // Remove novas linhas e espaços ao final
    line[strcspn(line, "\r\n")] = 0;

    // Ignora linhas vazias
    if (strlen(line) > 0) {
        if (execute_command(line) != 0) {
            printf("Interrompendo execução devido a erro.\n");
            fclose(file);
            return 1;
        }
    }
}

fclose(file);

printf("\nFinalizado\n");
```

```
    return 0;
}
```

3 EXPLICAÇÕES SOBRE CÓDIGO

Iremos detalhar cada parte do código para explicar suas funcionalidades.

Utilizamos 4 bibliotecas da linguagem C.

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

Essas bibliotecas oferecem funções essenciais para o desenvolvimento em C, stdlib.h permite alocação de memória e conversões, stdio.h fornece entrada e saída de dados, string.h facilita a manipulação de strings, e ctype.h permite análise e conversão de caracteres, como verificar se são letras ou números.

Após realizamos a definição de variáveis constantes que determinarão um limite para construção das tabelas e campos das tabelas.

```
#define MAX_FIELDS 10
#define MAX_RECORDS 100
#define MAX_FIELD_LENGTH 50
```

Onde MAX_FIELDS é úmero máximo de campos/colunas pode ter, MAX_RECORDS é número máximo de registros/linhas que uma tabela pode armazenar e MAX_FIELD_LENGTH sendo o comprimento máximo de um nome armazenado em um campo.

Logo após iremos definir a estrutura de dados das tabelas e campos de tabelas que serão lidos.

```
typedef struct {
    char field_name[MAX_FIELD_LENGTH];  // Nome do campo
```

```
    char field_type;                // Tipo do campo (C ou N)
} Field;

typedef struct {
    char table_name[MAX_FIELD_LENGTH];  // Nome da tabela
    Field fields[MAX_FIELDS];           // Nomes das colunas e tipos
    char   records[MAX_RECORDS][MAX_FIELDS][MAX_FIELD_LENGTH];   // Dados inseridos
    int field_count;                // Quantidade de colunas
    int record_count;               // Quantidade de registros
} Table;
```

Sendo Field a função que definira um campo na tabela e armazenara o tipo de variável, no caso C para texto e N para numérico.

É Table que tem a função de definir uma tabela, como, nome da tabela, um array de campos, um array de registros, e contadores para campos e registros.

Em seguida instanciamos duas variáveis globais.

```
Table tables[10];                   // Array de tabelas
int table_count = 0;                // Contador de tabelas
```

Sendo uma variável para armazenar no máximo 10 tabelas e um contador para rastrear quantas tabelas foram criadas no total.

Em seguida foi construído os Tokens, foi criado a definição dos tokens que o interpretador reconhecerá.

```
typedef enum {
    TOKEN_EOF,
    TOKEN_CREATE,
    TOKEN_TABLE,
    TOKEN_INSERT,
    TOKEN_INTO,
    TOKEN_SELECT,
```

```
    TOKEN_FROM,
    TOKEN_STRING,
    TOKEN_NUMBER,
    TOKEN_COMMA,
    TOKEN_LPAREN,  // (
    TOKEN_RPAREN,  // )
    TOKEN_SEMICOLON,
    TOKEN_ASTERISK // *
} TokenType;
```

A função struct tem como objetivo armazena o tipo de token e seu valor.

```
typedef struct {
    TokenType type;
    char value[MAX_FIELD_LENGTH];
} Token;
```

Objetivo da função get_token é ler uma sequência e definir as entradas dos tokens.

```
Token get_token(const char **input) {
    Token token;
    token.type = TOKEN_EOF;
    token.value[0] = '\0';

    // Ignorar espaços e novas linhas
    while (**input == ' ' || **input == '\n' || **input == '\t' || **input == '\r') {
        (*input)++;
    }

    if (strncmp(*input, "CREATE", 6) == 0) {
        (*input) += 6;
        token.type = TOKEN_CREATE;
```

```
} else if (strncmp(*input, "TABLE", 5) == 0) {
    (*input) += 5;
    token.type = TOKEN_TABLE;
} else if (strncmp(*input, "INSERT", 6) == 0) {
    (*input) += 6;
    token.type = TOKEN_INSERT;
} else if (strncmp(*input, "INTO", 4) == 0) {
    (*input) += 4;
    token.type = TOKEN_INTO;
} else if (strncmp(*input, "SELECT", 6) == 0) {
    (*input) += 6;
    token.type = TOKEN_SELECT;
} else if (strncmp(*input, "FROM", 4) == 0) {
    (*input) += 4;
    token.type = TOKEN_FROM;
} else if (**input == ',') {
    (*input)++;
    token.type = TOKEN_COMMA;
} else if (**input == '(') {
    (*input)++;
    token.type = TOKEN_LPAREN;
} else if (**input == ')') {
    (*input)++;
    token.type = TOKEN_RPAREN;
} else if (**input == ';') {
    (*input)++;
    token.type = TOKEN_SEMICOLON;
} else if (**input == '*') {
    (*input)++;
    token.type = TOKEN_ASTERISK;
} else if (isdigit(**input)) {
    int i = 0;
    while (isdigit(**input)) {
```

```
            token.value[i++] = *(*input)++;
        }
        token.value[i] = '\0';
        token.type = TOKEN_NUMBER;
    } else if (isalpha(**input)) {
        int i = 0;
        while (isalnum(**input) || **input == '_') {
            token.value[i++] = *(*input)++;
        }
        token.value[i] = '\0';
        token.type = TOKEN_STRING;
    }
    return token;
}
```

Agora será informado as funções que realizam a manipulação direta nas tabelas utilizando os tokens criados.

Começando com a função show_table que tem como objetivo imprimir na tela os valores que estão armazenados dentro de uma tabela.

```
void show_table(Table *table) {
    for (int i = 0; i < table->field_count; i++) {
        printf("%s\t", table->fields[i].field_name);
    }
    printf("\n");

    for (int i = 0; i < table->record_count; i++) {
        for (int j = 0; j < table->field_count; j++) {
            printf("%s\t", table->records[i][j]);
        }
        printf("\n");
    }
}
```

A função get_table_by_name realiza a busca por uma tabela e retorna via ponteiro a tabela ou NULL informando que não foi encontrado.

```
Table* get_table_by_name(const char *name) {
    for (int i = 0; i < table_count; i++) {
        if (strcmp(tables[i].table_name, name) == 0) {
            return &tables[i];
        }
    }
    return NULL;
}
```

A função validate_field_value tem o objetivo de validar se o valor inserido corresponde ao tipo de campo definido (numérico ou texto).

```
int validate_field_value(const char *value, char field_type) {
    if (field_type == 'N') { // campo numérico
        for (int i = 0; value[i] != '\0'; i++) {
            if (!isdigit(value[i])) {
                return 0;  // valor inválido
            }
        }
    } else if (field_type == 'C') { // campo de texto
        for (int i = 0; value[i] != '\0'; i++) {
            if (!isalpha(value[i])) {  // Aceita apenas letras para simplicidade
                return 0;  // valor inválido
            }
        }
    }
    return 1; // valor válido
}
```

Foi criado 3 funções são as principais para o funcionamento do código, sendo elas, parse_create_table que é responsável em analisa as instruções e criar a tabela de acordo com os parâmetros do código, parse_insert_into que tem o objetivo de armazenar valores dentro de uma tabela já criada e parse_select_from que exibe os registros da tabela solicitada.

```
// Função para criação da tabela
int parse_create_table(const char **input) {
    Token token = get_token(input);
    if (token.type == TOKEN_STRING) {
        Table *table = &tables[table_count++];
        strcpy(table->table_name, token.value);
        table->field_count = 0;
        token = get_token(input);
        if (token.type == TOKEN_LPAREN) {
            int field_index = 0;
            do {
                token = get_token(input);
                if (token.type == TOKEN_STRING) {
                    strcpy(table->fields[field_index].field_name, token.value);
                    token = get_token(input);
                    if (strcmp(token.value, "C") == 0) {
                        table->fields[field_index].field_type = 'C';
                    } else if (strcmp(token.value, "N") == 0) {
                        table->fields[field_index].field_type = 'N';
                    } else {
                        printf("Erro de sintaxe: Tipo de campo inválido.\n");
                        return 1;
                    }
                    field_index++;
                    table->field_count++;
                }
                token = get_token(input);
```

```
            } while (token.type == TOKEN_COMMA);
            if (token.type != TOKEN_RPAREN) {
                printf("Erro de sintaxe: Esperado ')'.\n");
                return 1;
            }
        } else {
            printf("Erro de sintaxe: Esperado '('.\n");
            return 1;
        }
    } else {
        printf("Erro de sintaxe: Nome da tabela esperado.\n");
        return 1;
    }
    return 0;
}

// Atualização na função parse_insert_into para validar tipos de dados
int parse_insert_into(const char **input) {
    Token token = get_token(input);
    if (token.type == TOKEN_STRING) {
        Table *table = get_table_by_name(token.value);
        if (!table) {
            printf("Erro: Tabela '%s' não encontrada.\n", token.value);
            return 1;
        }
        token = get_token(input);
        if (token.type == TOKEN_LPAREN) {
            int record_index = table->record_count;
            int field_index = 0;
            do {
                token = get_token(input);
                if (token.type == TOKEN_STRING || token.type ==
TOKEN_NUMBER) {
```

```
                    // Validar tipo do valor com o tipo de campo
                    if (field_index < table->field_count &&
                        !validate_field_value(token.value, table->fields[field_index].field_type)) {
                        printf("Erro de sintaxe: Tipo de valor inválido para o campo '%s'.\n",
                               table->fields[field_index].field_name);
                        return 1;
                    }
                    if (field_index < table->field_count) {
                        strcpy(table->records[record_index][field_index], token.value);
                        field_index++;
                    }
                }
                token = get_token(input);
            } while (token.type == TOKEN_COMMA);

            table->record_count++;
            printf("Registro inserido com sucesso! (Registro %d)\n", record_index + 1);
        } else {
            printf("Erro de sintaxe: Esperado '('.\n");
            return 1;
        }
    } else {
        printf("Erro de sintaxe: Nome da tabela esperado.\n");
        return 1;
    }
    return 0;
}

// Função para selecionar e imprimir dados da tabela
int parse_select_from(const char **input) {
```

```
            Token token = get_token(input);
            if (token.type == TOKEN_ASTERISK) {
                token = get_token(input);
                if (token.type == TOKEN_FROM) {
                    token = get_token(input);
                    if (token.type == TOKEN_STRING) {
                        Table *table = get_table_by_name(token.value);
                        if (table) {
                            show_table(table);
                            token = get_token(input);  // Verificar se há um ponto e vírgula após o comando
                            if (token.type != TOKEN_SEMICOLON && token.type != TOKEN_EOF) {
                                printf("Erro de sintaxe: Comando inesperado após SELECT.\n");
                                return 1;
                            }
                            return 0;
                        } else {
                            printf("Erro: Tabela '%s' não encontrada.\n", token.value);
                            return 1;
                        }
                    } else {
                        printf("Erro de sintaxe: Nome da tabela esperado após FROM.\n");
                        return 1;
                    }
                } else {
                    printf("Erro de sintaxe: Esperado 'FROM'.\n");
                    return 1;
                }
            } else {
                printf("Erro de sintaxe: Esperado '*'.\n");
                return 1;
            }
```

```
}
```

A função execute_command foi criada para determinar qual comando dos 3 deve ser executado e após selecionado ele chama as funções parse.

```
int execute_command(const char *input) {
    const char *ptr = input;
    Token token = get_token(&ptr);
    if (token.type == TOKEN_CREATE) {
        token = get_token(&ptr);
        if (token.type == TOKEN_TABLE) {
            if (parse_create_table(&ptr) != 0) return 1;
            printf("Tabela criada com sucesso!\n");
        }
    } else if (token.type == TOKEN_INSERT) {
        token = get_token(&ptr);
        if (token.type == TOKEN_INTO) {
            if (parse_insert_into(&ptr) != 0) return 1;
        }
    } else if (token.type == TOKEN_SELECT) {
        if (parse_select_from(&ptr) != 0) return 1;
    } else {
        printf("Erro de sintaxe: Comando desconhecido.\n");
        return 1;
    }
    return 0;
}
```

Por fim a função Main, que tem o objetivo de realizar a leitura do documento TXT e ler cada linha do documento para executar o interpretador de forma correta.

```
int main() {
   table_count = 0;
```

```
char filename[100];

printf("\nInterpretador DBASE\n");
printf("CMP1076 - COMPILADORES\n");
printf("Aluno: Leonardo de Moura Alves\n\n");

printf("Informe o nome do arquivo .txt com os comandos: ");
scanf("%99s", filename);

FILE *file = fopen(filename, "r");
if (!file) {
    printf("Erro ao abrir o arquivo.\n");
    return 1;
}

char line[256];
while (fgets(line, sizeof(line), file)) {
    // Remover novas linhas e espaços no final da linha
    line[strcspn(line, "\r\n")] = 0;

    // Ignorar linhas vazias
    if (strlen(line) > 0) {
        if (execute_command(line) != 0) {
            printf("Interrompendo execução devido a erro.\n");
            fclose(file);
            return 1;
        }
    }
}

fclose(file);
```

```
    printf("\nFinalizado\n");
    return 0;
}
```

4 CONCLUSÃO

O código construído é um interpretador básico para DBASE, permitindo criar tabelas, inserir registros e selecionar e imprmir dados. Cada componente é cuidadosamente projetado para lidar com as operações fundamentais do DBASE, enquanto valida entradas e gerencia tabelas e registros em memória.