Universidade do Estado do Rio de Janeiro
Instituto de Matemática e Estatística
Departamento de Ciência da Computação
Compiladores

Relatório Técnico do Analisador Léxico

Aluno:
Marcio Sirimarco de Souza Junior - 201710076811
Mateus Henrique Freitas Maciel - 202220460111

Professor(a):
Lis Custódio

21 de setembro de 2025

Conteúdo

1 Descrição Teórica do Programa

O programa `analisador_lexico.c` implementa a primeira fase de um compilador, conhecida como **Análise Léxica**. O objetivo fundamental de um analisador léxico é ler o código-fonte, que é uma sequência de caracteres, e convertê-lo em uma sequência de **tokens**. Cada token representa uma palavra-chave, um identificador, um número ou um operador. No nosso caso implementamos um token para erro **TOKEN_ERRO** que indica qual erro que ocorreu no léxico, também adicionamos um **TOKEN_EOF** que, assim como token de erro, facilita a programação do analisador léxico.

Teoricamente, o analisador se baseia no modelo de um **Autômato Finito Determinístico (AFD)**. Este modelo matemático define um conjunto de estados e transições entre eles com base nos caracteres de entrada. A implementação prática deste AFD no código é realizada através de uma estrutura `switch-case` dentro de um laço `while`, onde cada `case` representa um estado do autômato e as condições (`if`, `else if`) determinam as transições para os próximos estados.

Além do reconhecimento de tokens, esta implementação também gerencia uma **Tabela de Símbolos**, uma estrutura de dados para armazenar e referenciar de forma única os identificadores definidos.

2 Descrição da Estrutura do Programa

O programa é estruturado em torno de um conjunto de estruturas de dados globais que mantêm o estado da compilação e funções que operam sobre esses dados para produzir os tokens.

2.1 Estruturas de Dados Globais

Antes de detalhar as funções, é importante entender as estruturas de dados globais que elas compartilham:

- `AnalisadorLexico g_lexer`: Uma estrutura global que contém o estado atual da leitura do arquivo-fonte. Ela armazena um ponteiro para o buffer com todo o código (`fonte`), a `posicao` de leitura atual e o `atual` caractere sendo analisado. Essa abordagem centraliza o controle do buffer de entrada.
- **Tabela de Símbolos Dinâmica**:
  - `Simbolo* tabela_de_simbolos`: Um ponteiro que aponta para uma área de memória alocada dinamicamente para armazenar os identificadores. Começa como `NULL`.
  - `int tamanho_tabela`: Um contador que indica quantos símbolos estão atualmente armazenados na tabela.
  - `int capacidade_tabela`: Indica o tamanho total do bloco de memória alocado, ou seja, quantos símbolos a tabela pode conter antes de precisar de mais espaço.

  Essa abordagem permite que a tabela cresça conforme a necessidade, utilizando `realloc`, tornando o compilador robusto para programas de qualquer tamanho.
- `PalavraChave palavras_chave[]`: Um array global e estático que funciona como um dicionário. Ele mapeia as strings das palavras reservadas da linguagem (como "if", "while", "int") para seus respectivos códigos de token.

2.2 Estruturas de Dados Principais

- `Token`: Representa a unidade de saída do analisador. Contém `nome_token` (um inteiro que identifica o tipo do token) e um `union` chamado `atributo`. O uso de `union` é uma otimização de memória:
  - Se o token for um `IDENTIFICADOR`, o atributo armazena `pos_simbolo`, o índice único na tabela de símbolos.
  - Para `NUMERO` ou `STRING`, armazena o `valor_literal` em um array de caracteres.

```
typedef struct
{
    int nome_token;
    union
    {
        int pos_simbolo;
        char valor_literal[1024];
    } atributo;
} Token;
```

- `Simbolo`: Define a estrutura de uma entrada na tabela de símbolos, contendo apenas o `lexema` (a string do identificador) com um tamanho máximo definido por `MAX_ID_LEN`. A Tabela de símbolos será um ponteiro de símbolo

```
typedef struct
{
    char lexema[MAX_ID_LEN + 1];
} Simbolo;
```

3 Descrição de Funcionamento

O fluxo de execução do programa é orquestrado pela função `main`, que delega a tarefa de reconhecimento de tokens para a função `get_token`.

1. **Inicialização (`main`)**: A `main` primeiramente lê todo o conteúdo do arquivo-fonte para a memória. Em seguida, inicializa a estrutura global `g_lexer` para apontar para o início deste conteúdo.
2. **Loop Principal (`main`)**: A `main` entra em um laço infinito que chama repetidamente a função `get_token()` para obter o próximo token do código-fonte. Após cada chamada, o token recebido é passado para `imprimir_token()`. O laço é interrompido apenas quando um `TOKEN_EOF` (fim do arquivo) ou `TOKEN_ERRO` é recebido.
3. **O Motor de Análise (`get_token`)**: Esta é a função central. Seu laço `while(1)` e `switch(estado)` implementam o AFD. O `STATE_INICIAL` atua como um "distribuidor", decidindo para qual estado transitar com base no caractere atual. Estados como `STATE_EM_IDENTIFICADOR` são "gulosos", consumindo múltiplos caracteres para formar um lexema.
4. **Classificação e Instalação**: Ao finalizar um identificador, o código primeiro verifica se o lexema corresponde a uma palavra-chave. Se não corresponder, ele chama a função `instalar_id()` para registrar o identificador na tabela de símbolos e armazena o índice retornado no atributo do token.
5. **Tratamento de Erros**: Os estados para comentários e strings possuem lógica para detectar se não foram fechados antes do fim do arquivo, retornando um `TOKEN_ERRO` específico para auxiliar o programador.

6. **Gerenciamento da Tabela de Símbolos (`instalar_id`)**: Esta função garante que cada identificador seja armazenado apenas uma vez, expandindo a tabela dinamicamente com `realloc` quando necessário.

7. **Finalização (`main`)**: Após o término do laço de análise, a `main` imprime o conteúdo final da tabela de símbolos e libera toda a memória alocada dinamicamente.

4 Descrição dos Testes Realizados e Saídas Obtidas

4.1 Teste 1: `teste1.txt`

Objetivo: Validar a tokenização básica, incluindo o reconhecimento das palavras-chave `inicio` e `fim`, a declaração de um tipo (`int`), a instalação de um identificador na tabela de símbolos (`a`), sua reutilização em uma atribuição e em uma chamada de função (`print`), e o reconhecimento de um número e operadores de pontuação.

Código-Fonte de Entrada:

```
inicio
  int a;
  a = 10;
  print(a);
fim
```

Saída Gerada:

```
<INICIO, >
<TIPO_INT, >
<ID, 0>
<PONTO_VIRGULA, >
<ID, 0>
<ATRIBUICAO, >
<NUM, 10>
<PONTO_VIRGULA, >
<PRINT, >
<ABRE_PARENTESES, >
<ID, 0>
<FECHA_PARENTESES, >
<PONTO_VIRGULA, >
<FIM, >
<EOF, >


--- Tabela de S mbolos ---
 ndice    | Lexema
----------------------------
0       | a
----------------------------
```

Análise: A saída foi exatamente a esperada. O identificador `a` foi instalado na tabela de símbolos no índice 0 na sua primeira aparição. Nas utilizações seguintes, o token para `a` corretamente referencia o mesmo índice 0, demonstrando que a busca na tabela de símbolos funciona. Todas as palavras-chave e operadores foram tokenizados corretamente.

4.2 Teste 2: `teste2.txt`

Objetivo: Testar o reconhecimento de múltiplos identificadores distintos, o tipo `float`, e números de ponto flutuante.

Código-Fonte de Entrada:

```
inicio
  float salario;
  float bonus;
  salario = 2500.50;
  bonus = salario * 0.1;
  print(salario + bonus);
fim
```

Saída Gerada:

```
<INICIO, >
<TIPO_FLOAT, >
<ID, 0>
<PONTO_VIRGULA, >
<TIPO_FLOAT, >
<ID, 1>
<PONTO_VIRGULA, >
<ID, 0>
<ATRIBUICAO, >
<NUM, 2500.50>
<PONTO_VIRGULA, >
<ID, 1>
<ATRIBUICAO, >
<ID, 0>
<OP_MULT, >
<NUM, 0.1>
<PONTO_VIRGULA, >
<PRINT, >
<ABRE_PARENTESES, >
<ID, 0>
<OP_SOMA, >
<ID, 1>
<FECHA_PARENTESES, >
<PONTO_VIRGULA, >
<FIM, >
<EOF, >


--- Tabela de S mbolos ---
 ndice    | Lexema
----------------------------
0       | salario
1       | bonus
----------------------------
```

Análise: O teste foi bem-sucedido. O analisador léxico instalou corretamente `salario` no índice 0 e `bonus` no índice 1, provando a capacidade da tabela de símbolos de crescer. O estado `STATE_EM_NUMERO` demonstrou ser capaz de ler e armazenar literais numéricos contendo o caractere `.` (ponto decimal).

4.3 Teste 3: `teste3.txt`

Objetivo: Validar o reconhecimento do tipo `string` e de literais de string.
Código-Fonte de Entrada:

```
inicio
  string nome;
  string saudacao;
  nome = "mundo";
  saudacao = "Ola, ";
  print(saudacao);
  print(nome);
fim
```

Saída Gerada:

```
<INICIO, >
<TIPO_STRING, >
<ID, 0>
<PONTO_VIRGULA, >
<TIPO_STRING, >
<ID, 1>
<PONTO_VIRGULA, >
<ID, 0>
<ATRIBUICAO, >
<STRING, mundo>
<PONTO_VIRGULA, >
<ID, 1>
<ATRIBUICAO, >
<STRING, Ola, >
<PONTO_VIRGULA, >
<PRINT, >
<ABRE_PARENTESES, >
<ID, 1>
<FECHA_PARENTESES, >
<PONTO_VIRGULA, >
<PRINT, >
<ABRE_PARENTESES, >
<ID, 0>
<FECHA_PARENTESES, >
<PONTO_VIRGULA, >
<FIM, >
```

```
<EOF, >


--- Tabela de S mbolos ---
 ndice    | Lexema
---------------------------
0        | nome
1        | saudacao
---------------------------
```

Análise: O estado `STATE_EM_STRING` funcionou como esperado, capturando corretamente os conteúdos "mundo"e "Ola, "como o atributo literal dos tokens `STRING`.

4.4 Teste 4: `teste4.txt`

Objetivo: Testar a robustez com identificadores longos, comentários de linha única, e caracteres especiais dentro de uma string.

Código-Fonte de Entrada:

```
inicio
  int produto_id;
  produto_id = 404;

  -- O simbolo @ e invalido
  string email;
  email = "usuario@exemplo.com";

  print(produto_id);
fim
```

Saída Gerada:

```
<INICIO, >
<TIPO_INT, >
<ID, 0>
<PONTO_VIRGULA, >
<ID, 0>
<ATRIBUICAO, >
<NUM, 404>
<PONTO_VIRGULA, >
<TIPO_STRING, >
<ID, 1>
<PONTO_VIRGULA, >
<ID, 1>
<ATRIBUICAO, >
<STRING, usuario@exemplo.com>
<PONTO_VIRGULA, >
<PRINT, >
<ABRE_PARENTESES, >
<ID, 0>
<FECHA_PARENTESES, >
<PONTO_VIRGULA, >
<FIM, >
<EOF, >


--- Tabela de S mbolos ---
 ndice    | Lexema
---------------------------
0        | produto_id
1        | email
---------------------------
```

Análise: O teste demonstrou três pontos importantes: 1) O identificador longo foi aceito. 2) O comentário iniciado com `--` foi completamente ignorado. 3) O caractere `@` dentro da string foi corretamente processado como parte do literal.

4.5 Teste 5: `teste5_erro.txt`

Objetivo: Validar o tratamento de erro para um comentário de bloco (longo) não fechado.
Código-Fonte de Entrada:

```
inicio
    int a;
    --abcd
fim
--[[
```

Saída Gerada:

```
<INICIO, >
<TIPO_INT, >
<ID, 0>
<PONTO_VIRGULA, >
<FIM, >
<ERRO, Caractere 'Comentario longo nao finalizado antes do fim do arquivo' inv lido>


--- Tabela de S mbolos ---
 ndice    | Lexema
----------------------------
0       | a
----------------------------
```

Análise: O analisador léxico se comportou como o esperado. Ao encontrar `--[[` e não encontrar um `]]` correspondente antes do fim do arquivo, a lógica de erro no estado `STATE_EM_COMENTARIO` foi ativada, retornando um `TOKEN_ERRO`.

4.6 Teste 6: `teste6_erro.txt`

Objetivo: Validar o tratamento de erro para um literal de string não fechado.
Código-Fonte de Entrada:

```
inicio
    a = 0;
    b = "abc;
    a = b * 0.1;
fim
```

Saída Gerada:

```
<INICIO, >
<ID, 0>
<ATRIBUICAO, >
<NUM, 0>
<PONTO_VIRGULA, >
<ID, 1>
<ATRIBUICAO, >
<ERRO, Caractere 'String nao finalizada antes do fim do arquivo: "abc;
    a = b * 0.1;
fim' inv lido>


--- Tabela de S mbolos ---
 ndice    | Lexema
----------------------------
0       | a
1       | b
----------------------------
```

Análise: A saída confirma que a lógica de erro no estado `STATE_EM_STRING` está funcionando corretamente, retornando um `TOKEN_ERRO` informativo quando o fim do arquivo é alcançado dentro de uma string.

A Referências

1 - COOPER, Keith D.; TORCZON, Linda. Construindo Compiladores. Elsevier, 2011.

2 - PRICE, Ana Maria de Alencar; TOSCANI, Simão Sirineo. Implementação de Linguagens de Programação - Compiladores. Bookman, 2009.

3 - AHO, Alfred V. et al. Compiladores: Princípios, Técnicas e Ferramentas. Pearson Education, 2007.

4 - UNIVERSIDADE DO ESTADO DO RIO DE JANEIRO. Material didático de Compiladores - Especificação da Linguagem LIS. UERJ, 2024.