



Centro Federal de Educação Tecnológica de Minas Gerais

Departamento de Computação

Curso de Engenharia de Computação

Josué Rocha Lima

Trabalho Prático: Implementação de um Analisador Léxico

---

## **KPiler: compilador para a linguagem K**

---

Relatório do trabalho prático apresentado à  
disciplina de compiladores.

Orientador: Kecia Aline Marques Ferreira

Belo Horizonte

2017

# Sumário

1	INTRODUÇÃO . . . . .	3
1.1	Proposta de trabalho prático . . . . .	3
2	METODOLOGIA . . . . .	5
2.1	Projeto do analisador léxico . . . . .	5
2.2	Implementação do analisador léxico . . . . .	7
3	INSTRUÇÕES DE UTILIZAÇÃO . . . . .	9
4	AVALIAÇÃO DOS RESULTADOS . . . . .	10
5	CONCLUSÃO . . . . .	19
	REFERÊNCIAS . . . . .	20

# 1 Introdução

Com o intuito de reduzir a complexidade da implementação de sistemas de software, foi proposta a utilização de linguagens próximas da linguagem humana, denominadas linguagens de alto nível. Entretanto, para possibilitar a implementação de software dessa maneira, é necessário a tradução para uma linguagem que possa ser executada diretamente no *hardware* (AHO; SETHI; ULLMAN, 2007).

O compilador é um sistema de *software* responsável pela tradução automática de um programa em linguagem fonte de alto nível, para um programa semanticamente equivalente em uma linguagem alvo, conforme pode-se observar na Figura 1. Deste modo, os compiladores são considerados *software* básico para a computação.

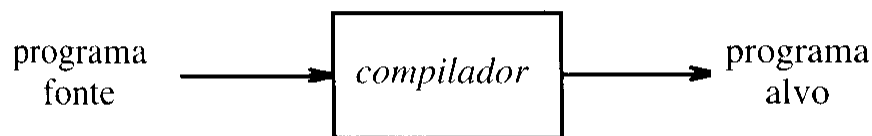


Figura 1 – Tradução do programa fonte para a linguagem alvo realizada pelo compilador (AHO; SETHI; ULLMAN, 2007).

## 1.1 Proposta de trabalho prático

Foi proposto o projeto e implementação de um compilador para uma linguagem hipotética *K* como trabalho prático e didático para a disciplina de Compiladores. Devido à complexidade do problema de compilação, o projeto foi dividido em quatro componentes com suas respectivas entregas:

1. Analisador léxico e tabela de símbolos;
2. Analisador sintático;
3. Analisador semântico;
4. Gerador de código.

O trabalho entregue anexo à este relatório corresponde ao analisador léxico e tabela de símbolos (item 1). Para essa etapa, foi especificada uma gramática para a linguagem *K* com a listagem de seus respectivos *tokens*.

O analisador léxico lê o programa fonte caractere a caractere e agrupa o código-fonte em *tokens* formados por um par  $\langle nome, valor \rangle$ , no qual valor pode ser seu lexema correspondente ou posição na tabela de símbolos.

Foi solicitado a implementação do reconhecimento dos *tokens* especificados na gramática e identificação de erros léxicos.

## 2 Metodologia

A metodologia utilizada na implementação do analisador léxico foi a combinação de vários diagramas de transição em um único diagrama. Posteriormente, seu comportamento foi codificado diretamente no código de modo à evitar o custo computacional adicional despendido em implementações dirigidas por tabela de transição.

### 2.1 Projeto do analisador léxico

Para possibilitar a implementação da metodologia proposta, os *tokens* especificados na gramática da linguagem *K* foram identificados e relacionados com suas respectivas categorias (Tabela 1):

Categoria	Nome dos tokens	Lexemas
Palavras reservadas	<program, end, int, string, if, else, then, end, do, while, scan e print	Idêntico ao nome dos tokens
Operadores	=, -, !, >, <, EQ, GE, LE, DIFFERENT	=, -, !, >, <, ==, >=, <=, !=
Identificadores	ID	Letra seguida por letras e dígitos.
Constantes	int, string	Um ou mais dígitos, caracteres cercados por aspas duplas
Outros	; ( ),	; , ( )

Tabela 1 – Relação dos tokens e suas categorias.

As palavras reservadas foram reconhecidas por meio de pré-inserção na tabela de símbolos. Posteriormente, foram gerados os autômatos para reconhecimento dos *tokens* das categorias **operador**, **identificador** e **constantes**:

- Reconhecimento de identificadores

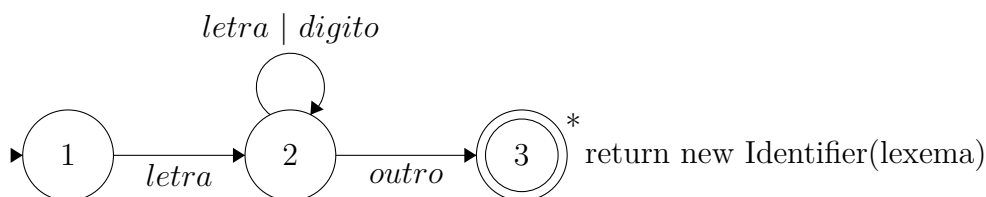


Figura 2 – Reconhecimento de identificadores por meio de um autômato finito determinístico.

### • Reconhecimento de operadores

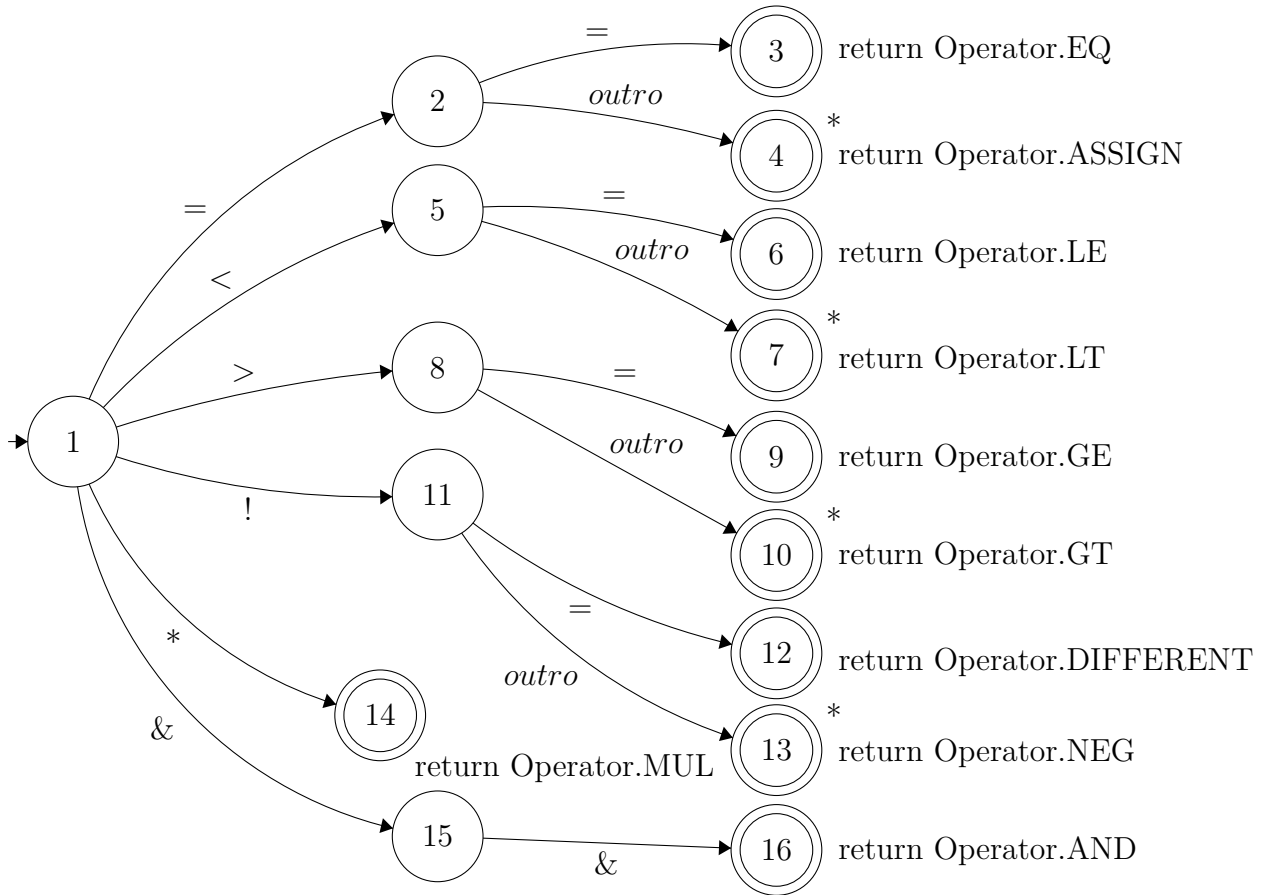


Figura 3 – Autômato finito determinístico construído para o reconhecimento de operadores.

### • Reconhecimento de constantes

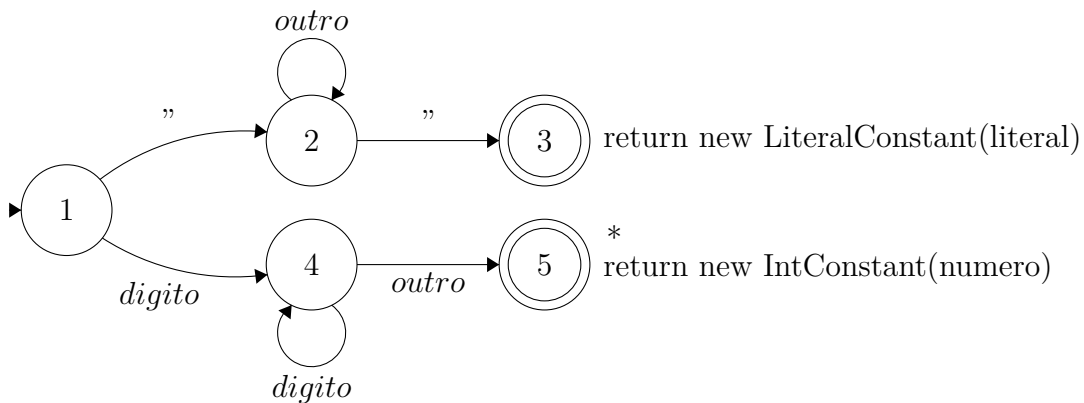


Figura 4 – Reconhecimento de constantes inteiras e literais por meio de um autômato finito determinístico.

## 2.2 Implementação do analisador léxico

A linguagem de programação Java foi escolhida para a implementação do analisador léxico e demais módulos do compilador devido aos recursos de processamento de texto e estruturas de dados disponíveis na linguagem, além de sua compilação híbrida que permite execução multiplataforma. Um diagrama de classe (Figura 2.2) foi construído para melhor entendimento do programa desenvolvido:

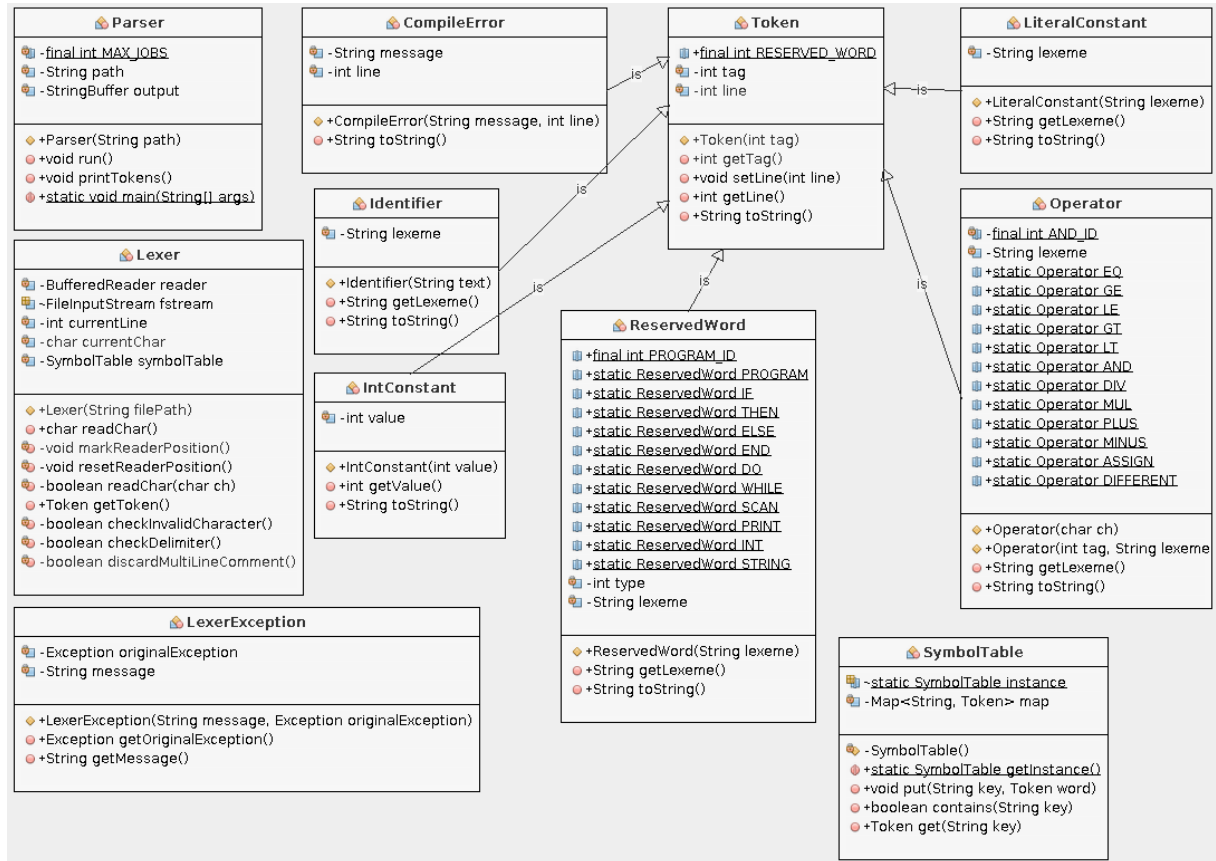


Figura 5 – Diagrama de classe do compilador, as classes utilitárias foram omitidas para facilitar o entendimento.

As classes e funções representados no diagrama de classes da estão detalhados a seguir:

- **Parser (analisador sintático):** classe que contem o ponto de entrada (método *main*) do programa. Uma requisição é feita à classe *Lexer* para obter o fluxo de tokens, de modo à implementar compilação em um passo. A classe possui implementação multi-thread possibilitando processamento de vários arquivos simultaneamente.
- **Lexer (analisador léxico):** lê o programa fonte caractere a caractere, buscando sequências de caracteres significativas (lexemas) que casem com o padrão de algum *token*. Possui estruturas para leitura do arquivo, armazenamento da linha corrente e

um ponteiro para a tabela de símbolos. O método *getToken()* contém a implementação dos autômatos para reconhecimento de *tokens*.

- **Token:** contém o dicionário de *tags* e um atributo inteiro para armazenamento do mesmo. É classe pai das classes *CompileError*, *Identifier*, *IntConstant*, *LiteralConstant*, *ReservedWord* e *Operator*. Estas classes implementam polimorfismo no método *toString()* para exibir os *tokens* em seu formato apropriado.
- **ReservedWord:** armazena palavras reservadas possibilitando diferenciação de identificadores por meio de verificação da classe instanciada. Possui uma relação de palavras reservadas pré-instanciadas para otimizar uso de memória.
- **LiteralConstant:** classe para armazenamento de constantes literais, filha de *Token*, contendo o atributo *lexeme* do tipo *String*.
- **IntConstant:** possui um atributo do tipo inteiro para armazenamento de constantes do mesmo tipo informadas diretamente no código.
- **Identifier:** armazenamento de identificadores com atributos similares à classe *LiteralConstant*, entretanto exibe os identificadores no formato de *token* apropriado utilizando polimorfismo no método *toString()*;
- **CompileError:** armazena erros de compilação, contendo métodos para exibição do erro na cor vermelho utilizando *color escape sequences* multiplataforma.
- **Operator:** armazenamento de operadores, possui construtores que possibilitam identificação por meio de código ASCII. Contém um dicionário de operadores instanciados previamente para diminuir custo de armazenamento.
- **SymbolTable:** implementa tabela de símbolos utilizando a tabela *hash*, com complexidade de busca para  $O(1)$ . Mapeia lexemas do tipo *string* para objetos da classe *Token*. Entretanto, em fases futuras a estrutura de armazenamento será alterada para conter mais dados do identificador (escopo e tipo). Além disso, foi utilizado o padrão de desenvolvimento *singleton* para permitir o acesso da tabela de símbolos por qualquer um dos módulos do compilador.



### 3 Instruções de utilização

O projeto está preparado para receber o caminho dos arquivos de código-fonte via linha de comando (terminal do Linux ou *prompt* de comandos do Windows). A passagem dos argumentos pode ser configurada no ambiente de desenvolvimento de preferência ou informado na inicialização do arquivo executável .jar ou na execução direta dos arquivos .class. Exemplo de execução:

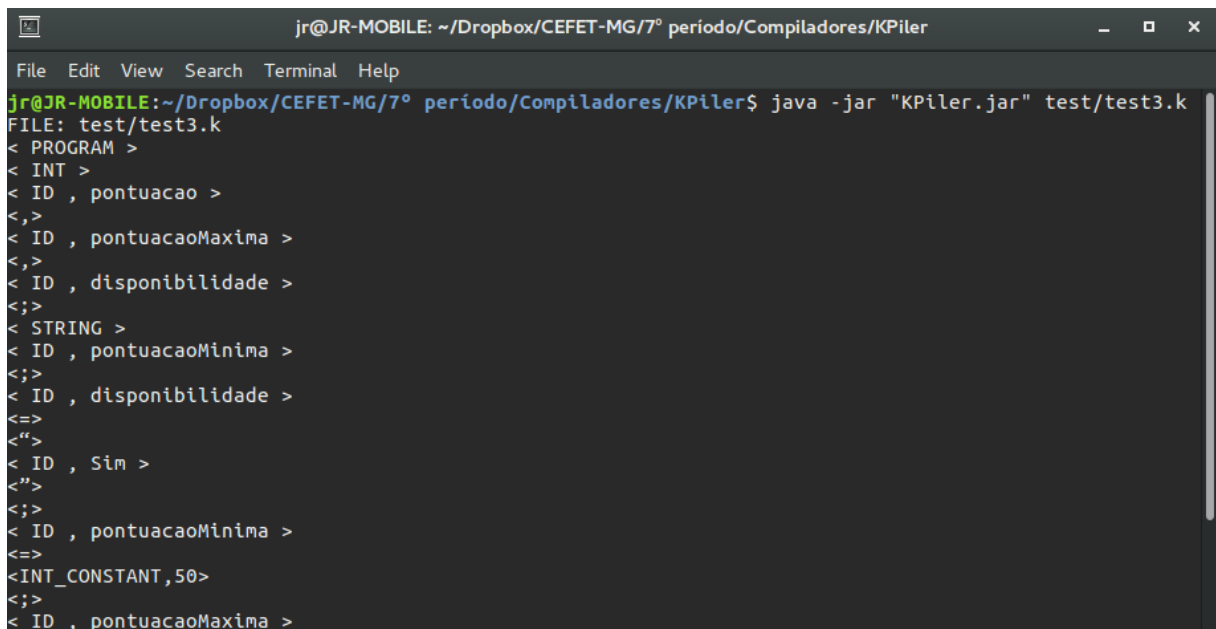
- **Execução do .jar:** o seguinte comando deve ser executado na mesma pasta do arquivo .jar caso as configurações do projeto anexado sejam mantidas:

```
java -jar KPiler.jar test/test1.k test/test1.k
```

- **Execução dos arquivos .class:** deve-se navegar para a pasta do *build* e executar os seguintes comandos para compilar e executar o código:

```
javac Parser.java
```

```
java Parser test/test1.k test/test1.k
```



```
jr@JR-MOBILE: ~/Dropbox/CEFET-MG/7º período/Compiladores/KPiler
File Edit View Search Terminal Help
jr@JR-MOBILE:~/Dropbox/CEFET-MG/7º período/Compiladores/KPiler$ java -jar "KPiler.jar" test/test3.k
FILE: test/test3.k
< PROGRAM >
< INT >
< ID , pontuacao >
< , >
< ID , pontuacaoMaxima >
< , >
< ID , disponibilidade >
< ; >
< STRING >
< ID , pontuacaoMinima >
< ; >
< ID , disponibilidade >
< == >
< " >
< ID , Sim >
< " >
< ; >
< ID , pontuacaoMinima >
< == >
< INT_CONSTANT,50>
< ; >
< ID , pontuacaoMaxima >
```

Figura 6 – Execução do programa via terminal.

## 4 Avaliação dos resultados

O programa foi submetido a seis testes de compilação. Cinco dos testes foram elaborados pela orientadora deste trabalho e um teste foi elaborado pelo autor. Os arquivos de teste estão disponibilizados em anexo no diretório *test* disponível neste projeto e em sua página do GitHub<sup>1</sup>.

Foram implementadas verificações de dois tipos de erros léxicos: *strings* não delimitadas com `"` e comentários não encerrados. A seguir se encontram os testes e o resultado dos mesmos:

- **Teste 1**

### Código-fonte:

---

```

program
  int a, b;
  int result;
  float a,x,total;

  a = 2;
  x = .1;
  scan (b);
  scan (y)
  result = (a*b ++ 1) / 2;
  print "Resultado: ";
  print (result);
  print ("Total: ");
  total = y / x;
  print ("Total: ");
  print (total);
end

```

---

### Resultado de execução: FILE: test/test1.k

---

```

< PROGRAM > < INT > < ID , a > <,> < ID , b > <;> < INT > < ID , result >
<;> < ID , float > < ID , a > <,> < ID , x > <,> < ID , total > <;> <
ID , a > <=> <INT_CONSTANT,2> <;> < ID , x > <=> <.> <INT_CONSTANT,1>
<;> < SCAN > <(> < ID , b > <)> <;> < SCAN > <(> < ID , y > <)> < ID
, result > <=> <(> < ID , a > <*> < ID , b > <+> <+> <INT_CONSTANT,1>

```

---

<sup>1</sup> <<https://github.com/josuerocha/KPiler>>

---

```

<>> </> <INT_CONSTANT,2> <;> < PRINT > <LITERAL_CONST , Resultado: >
<;> < PRINT > <(> < ID , result > <)> <;> < PRINT > <(>
<LITERAL_CONST , Total: > <)> <;> < ID , total > <=> < ID , y > </> <
ID , x > <;> < PRINT > <(> <LITERAL_CONST , Total: > <)> <;> < PRINT >
<(> < ID , total > <)> <;> < END >

```

---

O programa fonte não apresentou erros léxicos, correspondendo ao resultado esperado.

## • Teste 2

### Código-fonte:

---

```

program
  int: a, c;
  float d, _e;
  a = 0; d = 3.5
  c = d / 1.2;
  Scan (a);
  Scan (c);
  b = a * a;
  c = b + a * (1 + a*c);
  print ("Resultado: ");
  print c;
  a = b + c + d)/2;
  e = val + c + a;
  print ("E: ");
  print (e);

```

---

### Resultado de execução:

FILE: test/test2.k

```

< PROGRAM > < INT > <:> < ID , a > <,> < ID , c > <;> < ID , float > < ID
, d > <,> <_> < ID , e > <;> < ID , a > <=> <INT_CONSTANT,0> <;> < ID
, d > <=> <INT_CONSTANT,3> <.> <INT_CONSTANT,5> < ID , c > <=> < ID ,
d > </> <INT_CONSTANT,1> <.> <INT_CONSTANT,2> <;> < ID , Scan > <(> <
ID , a > <)> <;> < ID , Scan > <(> < ID , c > <)> <;> < ID , b > <=>
< ID , a > <*> < ID , a > <;> < ID , c > <=> < ID , b > <+> < ID , a
> <*> <(> <INT_CONSTANT,1> <+> < ID , a > <*> < ID , c > <)> <;> <
PRINT > <(> <LITERAL_CONST , Resultado: > <)> <;> < PRINT > < ID , c
> <;> < ID , a > <=> < ID , b > <+> < ID , c > <+> < ID , d > <)> </>
<INT_CONSTANT,2> <;> < ID , e > <=> < ID , val > <+> < ID , c > <+> <
ID , a > <;> < PRINT > <(> <LITERAL_CONST , E: > <)> <;> < PRINT >
<(> < ID , e > <)> <;>

```

---

O programa fonte correspondeu ao resultado esperado.

- Teste 3

## Código-fonte:

---

```

program
  int pontuacao, pontuacaoMaxima, disponibilidade;
  string pontuacaoMinima;
  disponibilidade = "Sim";
  pontuacaoMinima = 50;
  pontuacaoMaxima = 100;
  /* Entrada de dados
  Verifica aprovao de candidatos
  do
  print("Pontuacao Candidato: ");
    scan(pontuacao);
    print("Disponibilidade Candidato: ");
    scan(disponibilidade);

    if (( pontuao > pontuacaoMinima) and (disponibilidade=="Sim")
      then
        print("Candidato aprovado");
      else
        print("Candidato reprovado")
      end
    while ( pontuao >= 0)end
  end

```

---

## Resultado de execução:

---

```

< PROGRAM > < INT > < ID , pontuacao > <,> < ID , pontuacaoMaxima > <,> <
  ID , disponibilidade > <;> < STRING > < ID , pontuacaoMinima > <;> <
  ID , disponibilidade > <=> <LITERAL_CONST , Sim> <;> < ID ,
  pontuacaoMinima > <=> <INT_CONSTANT,50> <;> < ID , pontuacaoMaxima >
  <=> <INT_CONSTANT,100> <;>

```

ERROR: Unclosed multiple line comment on line 7

---

O compilador retornou erro de comentário não fechado na linha sete do programa fonte. Para corrigir o comentário foi fechado na última linha.

---

**Código-fonte:**

---

```
program
  int pontuacao, pontuacaoMaxima, disponibilidade;
  string pontuacaoMinima;
  disponibilidade = "Sim";
  pontuacaoMinima = 50;
  pontuacaoMaxima = 100;
  /* Entrada de dados
  Verifica aprovao de candidatos
  do
  print("Pontuacao Candidato: ");
    scan(pontuacao);
    print("Disponibilidade Candidato: ");
    scan(disponibilidade);

    if (( pontuao > pontuacaoMinima) and (disponibilidade=="Sim")
      then
        print("Candidato aprovado");
      else
        print("Candidato reprovado")
      end
    while ( pontuao >= 0)end
  end*/
```

---

**Resultado de execução:**

---

```
< PROGRAM > < INT > < ID , pontuacao > <,> < ID , pontuacaoMaxima > <,> <
  ID , disponibilidade > <;> < STRING > < ID , pontuacaoMinima > <;> <
  ID , disponibilidade > <=> <LITERAL_CONST , Sim> <;> < ID ,
  pontuacaoMinima > <=> <INT_CONSTANT,50> <;> < ID , pontuacaoMaxima >
  <=> <INT_CONSTANT,100> <;>
```

---

O resultado esperado foi alcançado e nenhum erro foi retornado pelo compilador.

- Teste 4

## Código-fonte:

---

```

int: a, aux\$, b;
string nome, sobrenome, msg;
print(Nome: );
scan (nome);
print("Sobrenome: ");
scan (sobrenome);
msg = "Ola, " + nome + " " +
sobrenome + "!";
msg = msg + 1;
print (msg);
scan (a);
scan(b);
if (a>b) then
    aux = b;
    b = a;
    a = aux;
end;
print ("Apos a troca: ");
out(a);
out(b)
end

```

---

## Resultado de execução: FILE: test/test4.k

---

```

< INT > <:> < ID , a > <,> < ID , aux > <\$> <,> < ID , b > <;> < STRING
> < ID , nome > <,> < ID , sobrenome > <,> < ID , msg > <;> < PRINT >
<(> < ID , Nome > <:> <>> <;> < SCAN > <(> < ID , nome > <>> <;> <
PRINT > <(> < LITERAL_CONST , Sobrenome: > <>> <;> < SCAN > <(> < ID ,
sobrenome > <>> <;> < ID , msg > <=> < LITERAL_CONST , Ola, > <+> < ID
, nome > <+> < LITERAL_CONST , > <+> < ID , sobrenome > <+>
< LITERAL_CONST , !> <;> < ID , msg > <=> < ID , msg > <+>
< INT_CONSTANT,1> <;> < PRINT > <(> < ID , msg > <>> <;> < SCAN > <(>
< ID , a > <>> <;> < SCAN > <(> < ID , b > <>> <;> < IF > <(> < ID ,
a > <>> < ID , b > <>> < THEN > < ID , aux > <=> < ID , b > <;> < ID
, b > <=> < ID , a > <;> < ID , a > <=> < ID , aux > <;> < END > <;>
< PRINT > <(> < LITERAL_CONST , Apos a troca: > <>> <;> < ID , out >
<(> < ID , a > <>> <;> < ID , out > <(> < ID , b > <>> < END >

```

---

O programa fonte não apresentou erros.

- Teste 5

## Código-fonte:

---

```

program
    int a, b, c, maior, outro;

    do
        print("A");
        scan(a);
        print("B");
        scan(b);
        print("C");
        scan(c);
        //Realizacao do teste
        if ( (a>b) && (a>c)
            maior = a
        )
        else
            if (b>c) then
                maior = b;
            else
                maior = c;
        end
    end
    print("Maior valor:");
    print (maior);
    print ("Outro? ");
    scan(outro);
    while (outro >= 0)
end

```

---

## Resultado de execução: FILE: test/test5.k

---

```

< PROGRAM > < INT > < ID , a > <,> < ID , b > <,> < ID , c > <,> < ID ,
maior > <,> < ID , outro > <;> < DO > < PRINT > <(> <LITERAL_CONST ,
A> <)> <;> < SCAN > <(> < ID , a > <)> <;> < PRINT > <(>
<LITERAL_CONST , B> <)> <;> < SCAN > <(> < ID , b > <)> <;> < PRINT >
<(> <LITERAL_CONST , C> <)> <;> < SCAN > <(> < ID , c > <)> <;> < IF
> <(> <(> < ID , a > <>> < ID , b > <)> <&&> <(> < ID , a > <>> < ID
, c > <)> < ID , maior > <=> < ID , a > <)> < ELSE > < IF > <(> < ID
, b > <>> < ID , c > <)> < THEN > < ID , maior > <=> < ID , b > <;> <
ELSE > < ID , maior > <=> < ID , c > <;> < END > < END > < PRINT >

```

---



```

<(> <LITERAL_CONST , Maior valor:>
ERROR: Unclosed string literal on line 22
< PRINT > <(> < ID , maior > <>> <;> < PRINT > <(> <LITERAL_CONST ,
    Outro? > <>> <;> < SCAN > <(> < ID , outro > <>> <;> < WHILE > <(> <
    ID , outro > <=> <INT_CONSTANT,0> <>> < END >

```

---

O programa fonte apresentou erros por má formação em *string* literal.

### Correção

```

program
    int a, b, c, maior, outro;

    do
        print("A");
        scan(a);
        print("B");
        scan(b);
        print("C");
        scan(c);
        //Realizacao do teste
        if ( (a>b) && (a>c)
            maior = a
        )
        else
            if (b>c) then
                maior = b;
            else
                maior = c;
            end
        end
        print("Maior valor:");
        print (maior);
        print ("Outro? ");
        scan(outro);
        while (outro >= 0)
    end

```

---

### Resultado:

```

< PROGRAM > < INT > < ID , a > <,> < ID , b > <,> < ID , c > <,> < ID ,
    maior > <,> < ID , outro > <;> < DO > < PRINT > <(> <LITERAL_CONST ,
    A> <>> <;> < SCAN > <(> < ID , a > <>> <;> < PRINT > <(>

```

---

```

<LITERAL_CONST , B> <>> <;> < SCAN > <(> < ID , b > <>> <;> < PRINT >
<(> <LITERAL_CONST , C> <>> <;> < SCAN > <(> < ID , c > <>> <;> < IF
> <(> <(> < ID , a > <>> < ID , b > <>> <&&> <(> < ID , a > <>> < ID
, c > <>> < ID , maior > <=> < ID , a > <>> < ELSE > < IF > <(> < ID
, b > <>> < ID , c > <>> < THEN > < ID , maior > <=> < ID , b > <>> <;> <
ELSE > < ID , maior > <=> < ID , c > <>> <;> < END > < END > < PRINT >
<(> <LITERAL_CONST , Maior valor:> <LITERAL_CONST , > <>> <;> < PRINT
> <(> < ID , maior > <>> <;> < PRINT > <(> <LITERAL_CONST , Outro? >
<>> <;> < SCAN > <(> < ID , outro > <>> <;> < WHILE > <(> < ID ,
outro > <=> <INT_CONSTANT,0> <>> < END >

```

---

Os erros apresentados anteriormente foram resolvidos.

## • Teste 5

### Código-fonte:

---

```

/*THIS IS A MULTIPLE LINE COMMENT
   Josu Rocha Lima */

program
  //this is an unclosed string literal
  string str = "HELLO WORLD"
  int 20 = 20;

  for i = 0 : str.size()
    print(i);
  end

end

```

---

### Resultado de execução:

---

```

FILE: test/test6.k
< PROGRAM > < STRING > < ID , str > <=> <LITERAL_CONST , HELLO WORLD> <
INT > <INT_CONSTANT,20> <=> <INT_CONSTANT,20> <;> < ID , for > < ID ,
i > <=> <INT_CONSTANT,0> <:> < ID , str > <.> < ID , size > <(> <>> <
PRINT > <(> < ID , i > <>> <;> < END > < END >

```

---

O programa fonte não apresentou erros, correspondendo aos resultados esperados.

## 5 Conclusão

O presente trabalho contribuiu para o entendimento dos fundamentos e técnicas de análise léxica, essenciais para compilação, processamento de texto, coletores *web* e robôs de resposta automatizada.

Além disso, foi possível entender o funcionamento de compiladores das linguagens de uso frequente - como Java, C++ e C - por meio da associação dos desafios encontrados neste projeto com experiências anteriores envolvendo compiladores para estas linguagens.

# Referências

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compilers: principles, techniques, and tools*. [S.l.]: Addison-wesley Reading, 2007. v. 2.