



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 sintático

---

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

---

Relatório do trabalho prático apresentado à  
disciplina de compiladores.

Orientador: Kecia Aline Marques Ferreira

Belo Horizonte

2017

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>3</b>
<b>1.1</b>	<b>Proposta de trabalho prático</b>	<b>4</b>
<b>2</b>	<b>METODOLOGIA</b>	<b>6</b>
<b>2.1</b>	<b>Projeto do analisador sintático</b>	<b>6</b>
<b>2.2</b>	<b>Implementação do analisador sintático</b>	<b>13</b>
2.2.1	Arquitetura da implementação	13
2.2.2	Recuperação de erro	14
2.2.3	Recursos adicionais	14
<b>3</b>	<b>INSTRUÇÕES DE UTILIZAÇÃO</b>	<b>15</b>
<b>4</b>	<b>AVALIAÇÃO DOS RESULTADOS</b>	<b>16</b>
<b>4.1</b>	<b>Teste 1</b>	<b>16</b>
<b>4.2</b>	<b>Teste 2</b>	<b>18</b>
<b>4.3</b>	<b>Teste 3</b>	<b>21</b>
<b>4.4</b>	<b>Teste 4</b>	<b>22</b>
<b>4.5</b>	<b>Teste 5</b>	<b>27</b>
<b>4.6</b>	<b>Teste 6</b>	<b>29</b>
<b>5</b>	<b>CONCLUSÃO</b>	<b>32</b>
	<b>REFERÊNCIAS</b>	<b>33</b>

# 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.

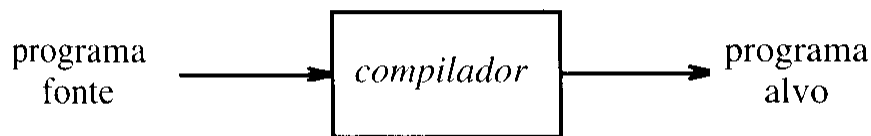


Figura 1 – Tradução do programa fonte para a linguagem alvo realizada pelo compilador (AHO; SETHI; ULLMAN, 2007).

O compilador é comumente dividido em módulos para reduzir a complexidade do seu processo de implementação. Essa divisão é frequentemente compreendida pelos seguintes módulos:

- **Front-end (Análise)**
  - **Analisador léxico:** lê o programa fonte caractere a caractere e agrupa o programa em sequências significativas denominadas lexemas. Para cada um dos lexemas é gerado um *token* no formato <nome , valor>. O analisador léxico também é responsável por reconhecer e ignorar comentários.
  - **Analisador sintático:** recebe o fluxo de *tokens* gerado pelo analisador sintático e verifica se a sequência está de acordo com a estrutura sintática da linguagem. É o "coração" do compilador.
  - **Analisador semântico:** Utiliza a tabela de símbolos e árvore sintática para verificar se o programa-fonte atende às regras semânticas da linguagem, como regras de compatibilidade de tipos.
  - **Gerador de código intermediário:** Código em representação intermediária em baixo nível é gerado para auxiliar a geração de código e otimização.

- **Back-end (Síntese)**

- **Gerador de código:** mapeia a linguagem intermediária para a linguagem objeto.

A Figura 1 ilustra o fluxo entre os módulos do compilador:

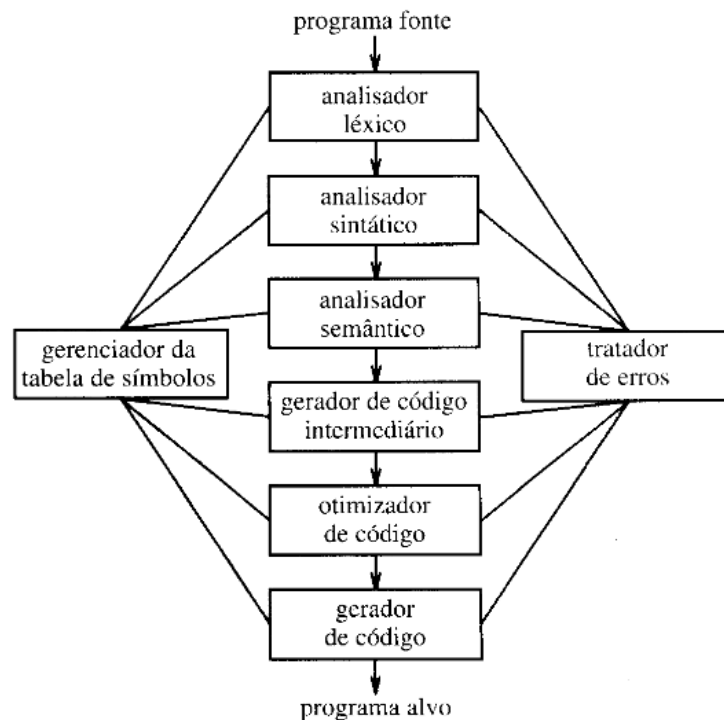


Figura 2 – Divisão em módulos do compilador (AHO; SETHI; ULLMAN, 2007).

## 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 sintático (2). Linguagens de programação possuem regras precisas que descrevem sentenças aceitas na construção de um programa bem formado (AHO; SETHI; ULLMAN, 2007). O analisador sintático verifica se o programa-fonte submetido ao compilador está sintaticamente correto, isto é, se as ordens dos *tokens* casam com as regras especificadas para a linguagem.

A organização e disposição sintática dos elementos de uma linguagem de programação é especificada por gramáticas livres de contexto. Como especificação para a linguagem  $K$ , foi fornecida uma gramática no formato Formalismo de Backus-Naur Extendido (EBNF, do inglês *Extended Backus-Naur Form*).

Deste modo, o analisador sintático recebe o fluxo de *tokens* gerado pelo analisador léxico e verifica se sua ordem casa com as regras especificadas na gramática. Foi solicitado a implementação do reconhecimento das regras sintáticas especificados na gramática e identificação de erros sintáticos.

Foi proposta implementação de uma analisador sintático descendente, cujo nome se origina do processo de derivação da entrada. A árvore sintática é construída de cima para baixo, produzindo uma derivação mais à esquerda para a cadeia de entrada.

O *parser* recursivo descendente utiliza uma série de procedimentos recursivos. São gerados procedimentos para cada símbolo não terminal da gramática que contêm pontos de decisão de acordo com cada elemento que o respectivo símbolo não-terminal possa produzir.

Entretanto, para que seja possível implementar este *parser*, é necessário que a gramática que especifica as regras da linguagem seja classificada como LL(1). O nome indica que a cadeia é lida da esquerda para a direita (primeiro L), é realizada uma derivação mais à esquerda (segundo L) e só é necessário conhecer um único símbolo a frente para a tomada de decisões.

Para que seja classificada como LL(1), é necessário que dada gramática não contenha recursão à esquerda e produções com prefixo comum. Pode-se verificar se é possível implementar um *parser* recursivo descendente por meio da geração de um projeto que tem como produto final a tabela do *parser*. Caso não existam entradas múltiplas na tabela do *parser*, a gramática é denominada LL(1). A geração da tabela do *parser* é realizada baseado nos conjuntos *first* e *follow* de cada não-terminal.

A metodologia e principais decisões tomadas nesse trabalho se encontra detalhada no Capítulo 2.

## 2 Metodologia

Para verificação da conformidade do programa-fonte com as regras sintáticas da linguagem  $K$ , foi desenvolvido um analisador sintático descendente. Foi construído um projeto para o *parser* com o intuito de facilitar o desenvolvimento e evitar descoberta de peculiaridades da implementação em uma fase tardia da implementação.

### 2.1 Projeto do analisador sintático

Inicialmente, foi realizado um estudo da gramática da linguagem  $K$ , com o intuito de identificar produções que não possuem características LL(1), isto é, recursão à esquerda e prefixos comuns. Após identificação destas características, as produções que continham as mesmas foram adaptadas para atender aos requisitos de uma gramática LL(1). As alterações realizadas consistiram nos seguintes aspectos:

- Adição de um ponto de partida contendo o símbolo inicial original seguido de \$ (fim de arquivo);
- Decomposição de regras com recursão à esquerda em duas regras para transformar em recursão à direita;
- Adiamiento da tomada de decisão em produções com prefixo comum por meio da criação de uma regra específica para o ponto de decisão.

A gramática obtida após as adaptações pode ser observada a seguir:

$$\langle \text{program\_prime} \rangle ::= \langle \text{Program} \rangle \text{'\$'} \quad (1)$$

$$\langle \text{Program} \rangle ::= \text{'program'} \langle \text{decllist} \rangle \langle \text{stmt-list} \rangle \text{'end'} \quad (2)$$

$$\langle \text{decllist} \rangle ::= \langle \text{decl-list} \rangle \quad (3)$$

$$| \text{'\lambda'} \quad (4)$$

$$\langle \text{decl-list} \rangle ::= \langle \text{decl} \rangle \langle \text{decllist} \rangle \quad (5)$$

$$\langle \text{decl} \rangle ::= \langle \text{type} \rangle \langle \text{identifier-list} \rangle ; \quad (6)$$

$$\langle \text{identifier-list} \rangle ::= \langle \text{identifier} \rangle \langle \text{possible-ident} \rangle \quad (7)$$

$$\langle \text{possible-ident} \rangle ::= \text{' ,' } \langle \text{identifier} \rangle \langle \text{possible-ident} \rangle \quad (8)$$

$$| \text{'\lambda'} \quad (9)$$

$$\langle type \rangle ::= \langle int \rangle \quad (10)$$

$$| \langle string \rangle \quad (11)$$

$$\langle stmt-list \rangle ::= \langle stmt \rangle \langle stmtlist \rangle \quad (12)$$

$$\langle stmtlist \rangle ::= \langle stmt-list \rangle \quad (13)$$

$$| \text{'}\lambda\text{' } \quad (14)$$

$$\langle stmt \rangle ::= \langle assign-stmt \rangle ; \quad (15)$$

$$| \langle if-stmt \rangle \quad (16)$$

$$| \langle while-stmt \rangle \quad (17)$$

$$| \langle read-stmt \rangle \text{'};' \quad (18)$$

$$| \langle write-stmt \rangle \text{'};' \quad (19)$$

$$\langle assign-stmt \rangle ::= \langle identifier \rangle \text{'=' } \langle simple-expr \rangle \quad (20)$$

$$\langle if-stmt \rangle ::= \text{'if' } \langle condition \rangle \text{'then' } \langle stmt-list \rangle \langle if-stmt-prime \rangle \quad (21)$$

$$\langle if-stmt-prime \rangle ::= \text{'end' } \quad (22)$$

$$| \text{'else' } \langle stmt-list \rangle \langle end \rangle \quad (23)$$

$$\langle condition \rangle ::= \langle expression \rangle \quad (24)$$

$$\langle while-stmt \rangle ::= \text{'do' } \langle stmt-list \rangle \langle stmt-suffix \rangle \quad (25)$$

$$\langle stmt-suffix \rangle ::= \text{'while' } \langle condition \rangle \text{'end' } \quad (26)$$

$$\langle read-stmt \rangle ::= \text{'scan' } \text{'(' } \langle identifier \rangle \text{' )' } \quad (27)$$

$$\langle write-stmt \rangle ::= \langle print \rangle \text{'(' } \langle writable \rangle \text{' )' } \quad (28)$$

$$\langle writable \rangle ::= \langle simple-expr \rangle \quad (29)$$

$$| \langle literal \rangle \quad (30)$$

$$\langle expression \rangle ::= \langle simple-expr \rangle \langle expression-prime \rangle \quad (31)$$

$$\langle expression-prime \rangle ::= \langle relop \rangle \langle simple-expr \rangle \quad (32)$$

$$| \text{'}\lambda\text{' } \quad (33)$$

$$\langle simple-expr \rangle ::= \langle term \rangle \langle simple-expr-prime \rangle \quad (34)$$

$$\langle simple-expr-prime \rangle ::= \langle addop \rangle \langle term \rangle \langle simple-expr-prime \rangle \quad (35)$$

$$| \text{'}\lambda\text{' } \quad (36)$$

$$\langle term \rangle ::= \langle factor-a \rangle \langle term-prime \rangle \quad (37)$$

$$\langle term-prime \rangle ::= \langle mulop \rangle \langle factor-a \rangle \langle term-prime \rangle \quad (38)$$

$$| \text{'}\lambda\text{' } \quad (39)$$

$$\langle factor-a \rangle ::= \langle factor \rangle \quad (40)$$

$$| \quad '!' \langle factor \rangle \quad (41)$$

$$| \quad '-' \langle factor \rangle \quad (42)$$

$$\langle factor \rangle ::= \langle identifier \rangle \quad (43)$$

$$| \quad \langle constant \rangle \quad (44)$$

$$| \quad '(' \langle expression \rangle ') \quad (45)$$

$$\langle relop \rangle ::= '=' \quad (46)$$

$$| \quad '>' \quad (47)$$

$$| \quad '>=' \quad (48)$$

$$| \quad '<' \quad (49)$$

$$| \quad '<=' \quad (50)$$

$$| \quad '!=' \quad (51)$$

$$\langle addop \rangle ::= '+' \quad (52)$$

$$| \quad '-' \quad (53)$$

$$| \quad '||' \quad (54)$$

$$\langle mulop \rangle ::= '*' \quad (55)$$

$$| \quad '/' \quad (56)$$

$$| \quad '&&' \quad (57)$$

Para possibilitar a implementação da metodologia proposta, os *tokens* especificados na gramática da linguagem *K* foram identificados e seus conjuntos *FIRST* e *FOLLOW* foram calculados (Tabela 1):

Token	Conjunto <i>FIRST</i>	Conjunto <i>FOLLOW</i>
Program	program	\$
decllist	$\lambda, int, string$	identifier, do, print, if, scan
decl-list	int, string	identifier, do, print, if, scan
decl	int, string	int, string, identifier, do, print, if, scan
identifier-list	identifier	;
possible-ident	",", $\lambda$	;
type	int, string	identifier
stmtlist	$\lambda, identifier,$ do, print, if, scan	while, end, else
assign-stmt	identifier	;
if-stmt	if	identifier, do, print, if, scan, while, end, else
if-stmt-prime	end, else	identifier, do, print, if, scan, while, end, else



while-stmt	do	identifier, do, print, if, scan, while, end, else
stmt-suffix	while	identifier, do, print, if, scan, while, end, else
read-stmt	scan	;
write-stmt	print	;
writable	literal, !, -, identifier, (, integer-const	)
simple-expr-prime	$\lambda$ , +, -,	==, >, >=, <, <=, !=, ) , end, then, ;
term-prime	$\lambda$ , *, /, &&	+, -,   , ==, >, >=, <, <=, !=, ) , end, then, ;
factor-a	!, -, identifier, ( integer_const, literal	*, /, &&, +, -, or, ==, >, >=, <, <=, !=, ) , end, then, ;
factor	identifier, (, integer_const, literal	*, /, &&, +, -, or, ==, >, >=, <, <=, !=, ) , end, then, ;
relop	==, >, >=, <, <=, !=	!, -, identifier, ( integer_const, literal
addop	+, -, or	!, -, identifier, ( integer_const, literal
mulop	*, /, &&	!, -, identifier, ( integer_const, literal
constant	integer_const, literal	*, /, &&, +, -, or, ==, >, >=, <, <=, !=, ) , end, then, ;
stmt	identifier, do, print, if, scan	identifier, do, print, if, scan, while, end, else
term	!, -, identifier, (, integer-const, literal	+, -,   , ==, >, >=, <, <=, !=, ) , end, then, ;
stmt-list	identifier, do, print, if, scan	while, end, else
simple-expr	!, -, identifier, ( integer_const, literal	==, >, >=, <, <=, !=, ) , end, then, ;

expression	!, -, identifier, ( integer-const, literal	), end, then
expression-prime	$\lambda$ , ==, >, >=, <, <=, !=	), end, then
condition	!, -, identifier, ( integer_const, literal	end, then

Tabela 1 – Produções e seus conjuntos *FIRST* e *FOLLOW*.

Produção	\$	program	end	identifier	,	int	string	;	=	if	then	else	do	while	scan	(	)	print
Program		2																
decllist				3						4			4		4			4
decl-list				5														
decl				6														
identifier-list				7														
possible-ident					8					8			8		8			9
type						10	11								8			
stmtlist			14	13						13		14	12	14	13			13
stmt-list				12						12			12		12			12
stmt				15						16			17		18			19
assign-stmt				20														
assign-stmt				20														
if-stmt										21								
if-stmt-prime			22									23						
condition				24												24		
while-stmt													25					
stmt-suffix														26				
read-stmt															27			
write-stmt																		28
write-stmt																		28
writable				29												29		
expression				31												31		
expression-prime			33								32						32	
simple-expr				34												34		

Tabela 2 – Primeira parte da tabela do parser LL(1)

literal	!	-	==	>	>=	<	<=	!=	+	or	*	/	&&	integer-const	literal
Program															
decllist															
decl-list															
decl															
possible-ident															
type															
stmtlist															
stmt-list															
stmt															
assign-stmt															
assign-stmt															
if-stmt															
if-stmt-prime															
condition	24	24												24	24
while-stmt															
stmt-sufix															
read-stmt															
write-stmt															
write-stmt															
writable	29	29												29	30
expression	31	31												31	31
expression-prime			32	32	32	32	32	32							
simple-expr	34	34												34	34

Tabela 3 – Segunda parte da tabela do parser LL(1)

## 2.2 Implementação do analisador sintático

A linguagem de programação Java foi escolhida para a implementação do analisador sintático do compilador em continuação a implementação prévia do primeiro trabalho prático e devido aos recursos de processamento de texto e estruturas de dados disponíveis na linguagem, além de sua compilação híbrida que permite execução multiplataforma.

### 2.2.1 Arquitetura da implementação

O diagrama de classe gerado no trabalho anterior (Figura 2.2.1) foi atualizado para compreender as alterações e avanços realizados no projeto:

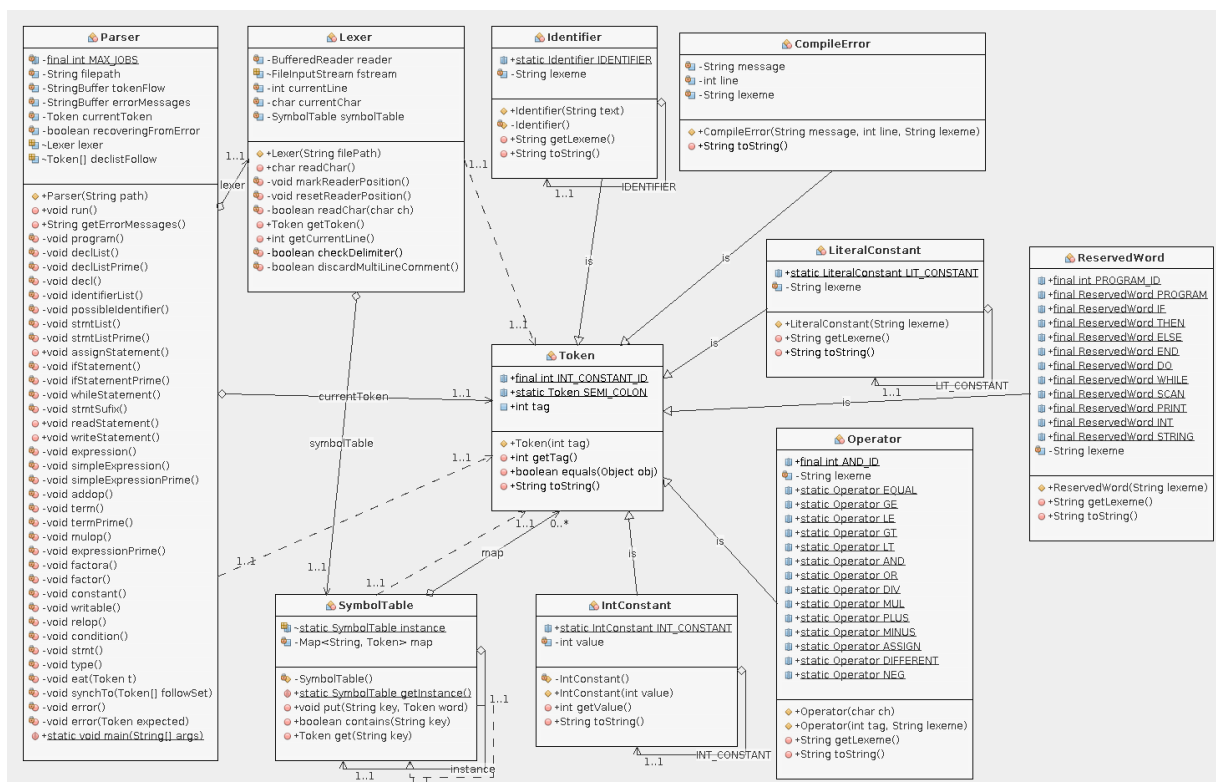


Figura 3 – Diagrama de classe do compilador, as classes utilitárias foram omitidas para facilitar o entendimento.

Nesta etapa do projeto, somente as classes *Token*, *Lexer* e *Parser* foram alteradas significativamente. Estas classes e funções estão detalhados a seguir:

- **Parser (analisador sintático):** classe que contém o ponto de entrada (método *main*) do programa. Uma requisição é feita à classe *Lexer* para obter o fluxo de *tokens*, de modo a implementar compilação em um passo. A classe possui implementação multi-thread possibilitando processamento de vários arquivos simultaneamente. Para cada símbolo não-terminal da gramática foi implementado um método contendo as

respectivas cláusulas da gramática. Os métodos *error()*, *synchTo(Token[] followSet)* e a **flag** booleana **recoveringFromError** foram utilizadas na implementação de erro em modo pânico.

- **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. Foi incorporado o método *equal(Token t)* à classe *Token* para facilitar as comparações de igualdade.

### 2.2.2 Recuperação de erro

Com o intuito de exibir a maior quantidade de erros possível ao usuário, recuperação de erro em modo pânico foi implementada. Após ocorrência de um erro, *tokens* são consumidos até que se encontre um *token* de um conjunto especificado. Utilizou-se o conjunto *follow* de cada não-terminal como heurística para recuperação de erro.

Com este recurso, foi possível exibir maior quantidade de erros em uma única compilação ao usuário. Os resultados da implementação podem ser observados no Capítulo 4 (Avaliação dos resultados).

### 2.2.3 Recursos adicionais

### 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:

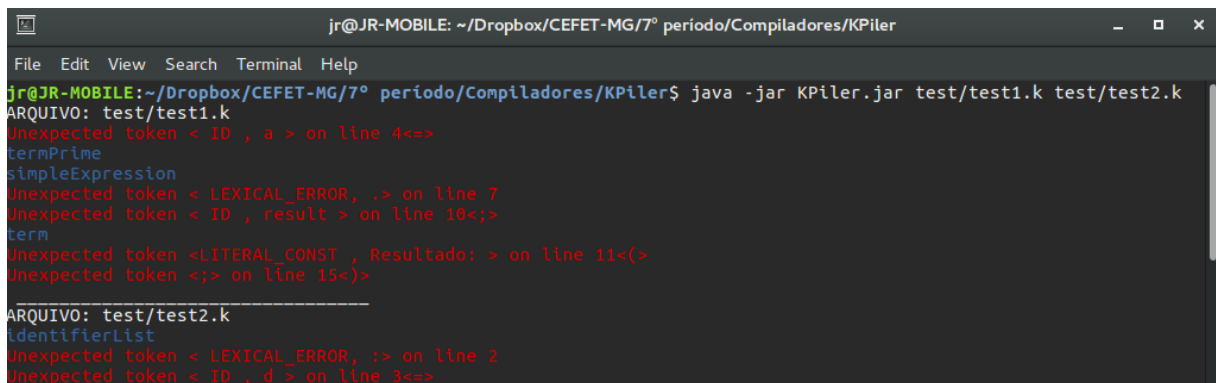
- **Execução do .jar:** o seguinte comando deve ser executado na mesma pasta do arquivo .jar caso as configurações do projeto anexado sejam mantidas:

```
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/test1.k test/test2.k
ARQUIVO: test/test1.k
Unexpected token < ID , a > on line 4<=>
termPrime
simpleExpression
Unexpected token < LEXICAL_ERROR, .> on line 7
Unexpected token < ID , result > on line 10<;>
term
Unexpected token < LITERAL_CONST , Resultado: > on line 11<(>
Unexpected token <;> on line 15<)>
-----
ARQUIVO: test/test2.k
IdentifierList
Unexpected token < LEXICAL_ERROR, !> on line 2
Unexpected token < ID , d > on line 3<=>
```

Figura 4 – 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>.

Os testes foram submetidos ao compilador de maneira incremental, isto é, os erros reportados em na fase inicial foram corrigidos e o código submetido novamente de maneira repetida até que não houvessem mais erros.

### 4.1 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 da execução 1:

---

```
Syntax error: unexpected token < ID , a > on line 4
Lexical error: Invalid token . on line 7
Syntax error: unexpected token < ID , result > on line 10
Syntax error: unexpected token <LITERAL_CONST , Resultado: > on line 11
```

---

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



---

Syntax error: unexpected token <> on line 15

---

Os erros apresentados foram corrigidos com as seguintes modificações:

- Substituição da palavra chave *float* por *int* na linha 4;
- Remoção do . na linha 7;
- Adição do ; ao final da linha 9;
- Envolvimento do literal "Resultado" com () na linha 11;
- Adição de ) após o literal "Total: " na linha 15.

O código-fonte submetido ao compilador novamente:

**Código-fonte corrigido:**

---

```
program
  int a, b;
  int result;
  int 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 da execução 2:**

---

Syntax error: unexpected token <+> on line 10

---

O símbolo + adicional foi removido e o código submetido novamente.

---

```
program
  int a, b;
  int result;
```

```
int 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
```

---

Nenhum outro erro foi retornado pelo compilador, atendendo aos resultados esperados.

## 4.2 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:

---

Lexical error: Invalid token : on line 2

Syntax error: unexpected token < ID , d > on line 3

Lexical error: Invalid token . on line 4  
Syntax error: unexpected token <(> on line 6  
Syntax error: unexpected token <(> on line 7  
Syntax error: unexpected token < ID , c > on line 11  
Syntax error: unexpected token <)> on line 12

---

O programa fonte foi corrigido da seguinte maneira:

- O caractere : foi removido da linha 2;
- A palavra *float* foi substituída por *int* na linha 3;
- O caractere . foi removido na linha 4;
- A palavra *Scan* foi substituída por *scan* nas linhas 6 e 7;
- Foi acrescentado o ( na linha 12.

**Código-fonte corrigido:**

---

```
program
    int a, c;
    int d, _e;
    a = 0; d = 35
    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);
```

---

O código-fonte foi submetido novamente ao compilador:

---

Lexical error: Invalid token \_ on line 3  
Syntax error: unexpected token < ID , e > on line 3  
Syntax error: unexpected token < ID , c > on line 5  
Syntax error: unexpected token < ID , c > on line 11  
Syntax error: unexpected token <EOF> on line 16

---

Correções:

- Remoção do caractere `_` na linha 3;
- Adição de `;` ao final da linha 4;
- Adição de `()` ao redor do identificador `c` na linha 11;
- Adição da palavra reservada `end` ao final do programa.

**Código-fonte corrigido:**

---

```
program
  int a, c;
  int d, e;
  a = 0; d = 35;
  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);
end
```

---

Após submissão, o compilador retornou um erro:

---

Lexical error: Invalid token . on line 5

---

O caractere `.` na linha 5 foi removido e submetido novamente:

**Código-fonte corrigido:**

---

```
program
  int a, c;
  int d, e;
  a = 0; d = 35;
  c = d / 12;
  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);  
end
```

---

Nenhum erro foi retornado, atendendo aos resultados esperados.

## 4.3 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:

---

Lexical 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 linha 21.

#### 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
```

---

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

## 4.4 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

---

Syntax error: unexpected token < INT > on line 1

---

O programa fonte apresentou erro na linha 1, pois a palavra chave program não foi introduzida ao início do programa. O programa foi corrigido e submetido novamente.

### Código-fonte corrigido:

---

```
program
    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:

---

Lexical error: Invalid token : on line 2  
Lexical error: Invalid token : on line 4  
Syntax error: unexpected token <;> on line 18

---

O código-fonte foi corrigido de acordo com os seguintes procedimentos:

- O caractere : foi removido na linha 2;
- O caractere : foi removido na linha 4.
- o caractere ; foi removido da linha 18

---

### Código-fonte corrigido:

---

```
program
  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 da execução:

---

Lexical error: Invalid token \\$ on line 2  
Syntax error: unexpected token < ID , b > on line 2  
Syntax error: unexpected token < STRING > on line 3

---



O token inválido \$ prejudicou a sincronia do analisador sintático e causou os dois erros subsequentes. O token \$ foi removido e submetido novamente:

#### Código-fonte corrigido:

---

```
program
  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 da execução:

---

Syntax error: unexpected token <(> on line 20

Syntax error: unexpected token <(> on line 21

---

Foram detectados erros nas linhas 20 e 21 devido à utilização de um comando (out) inválido que foi detectado como identificador. A palavra *out* foi substituída pela palavra chave *print*:

#### Código-fonte corrigido:

---

```
program
  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: ");
print(a);
print(b)
end
```

---

O programa-fonte foi testado novamente:

---

Syntax error: unexpected token < END > on line 23

---

Este erro foi corrigido adicionando-se o ; ao final da linha 23.

---

```
program
    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
end
```

```
print ("Apos a troca: ");  
print(a);  
print(b);  
end
```

---

A execução atendeu aos resultados esperados não retornando outros erros.

## 4.5 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:

---

Syntax error: unexpected token < ID , maior > on line 13

Syntax error: unexpected token < ELSE > on line 16

Syntax error: unexpected token <> on line 16

Syntax error: unexpected token < END > on line 22

---

O programa fonte foi corrigido de acordo com as seguintes alterações:

- Adição da palavra reservada *then* e *)* ao fim da linha 12;
- Remoção do *)* na linha 14 (correção do segundo erro);
- Adição da palavra reservada *end* na linha 25 após a palavra chave *while*.

### 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)) then
      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
end
```

---

**Resultado:**

---

Lexical error: Unclosed string literal on line 21

---

A aspas na linha 21 foram removidas.

---

```
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)) then
      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
end
```

---

Os erros apresentados anteriormente foram resolvidos.

## 4.6 Teste 6

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:

---

Syntax error: unexpected token <=> on line 6  
Syntax error: unexpected token < ID , for > on line 9  
Syntax error: unexpected token <>> on line 9

---

Correção dos erros:

- Separação da declaração e atribuição de valor às variáveis;
  - Troca do for por do e outras adaptações necessárias;
  - Substituição de str.size() por um identificador.
- 

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

```
program
    //this is an unclosed string literal
    string str;
    int i;
    int maxtam;

    maxtam = 20;
    str = "HELLO WORLD";

    do
        print(i);
        i = i + 1;
    while i < maxtam end

end
```

---

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

## 5 Conclusão

O trabalho em questão contribuiu para o entendimento dos fundamentos e técnicas de análise sintática por meio do desenvolvimento do software compilador *KPiler*. Percebeu-se a importância do analisador sintático para o compilador como núcleo do mesmo.

Experiências durante esta implementação facilitaram o entendimento do funcionamento dos analisadores sintáticos presentes em compiladores comerciais, bem como suas mensagens de erro

Todos os resultados de testes aplicados ao *KPiler* atenderam aos resultados esperados, quanto à coerência da informação reportada e quanto ao tempo de execução.



## Referências

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