

Rafael Menezes Barboza¹

¹Universidade Tecnológica Federal do Paraná (UTFPR)

Via Rosalina Maria dos Santos, 1233

CEP: 87301-899, Campo Mourão – PR – Brasil

ra29fa@gmail.com

Resumo. *O seguinte trabalho faz parte da disciplina de Compiladores do curso de ciência da computação, é a primeira parte de total de quatro etapas que ao final resultarão no estudo e implementação de compilador para a linguagem T++. Esta primeira abordagem pretende explorar o funcionamento do Analisador Léxico, um dos componentes principais para separar todo código fonte em pequenos "tokens" devidamente classificados e reconhecidos pela linguagem.*

1. Introdução

Um compilador de forma simplificada nada mais é do que um tradutor de um código fonte em uma linguagem de programação de alto nível para uma linguagem de programação de baixo nível. As linguagens de alto nível são muito utilizadas no desenvolvimento de aplicações, porém a maioria dos computadores processam instruções mais simples e elementares e o compilador tem exatamente este papel de transformar o código para algo mais próximo da linguagem da máquina.

O processo de compilação é dividido em várias etapas bem definidas, cada etapa gera uma saída de dados correspondente a entrada da próxima etapa a ser executada, se assemelhando aos famosos Pipelines. Cada etapa deve ser encarregada de tratar possíveis erros ou notificá-los antes de passar para as próximas fases.

2. Linguagem de Estudo T++

A linguagem de programação escolhida para o trabalho foi a T++ linguagem essa inventada para fins acadêmicos e não é utilizada em meios comerciais. A linguagem T++ se assemelha bastante com as principais linguagem de programação compiladas do mercado por conter uma lógica de programação muito bem estruturada e familiar porém com algumas certas limitações.

Se trata de uma linguagem considerada quase fortemente tipada pois toda e qualquer variável criada deve ser atribuída a algum tipo de dado reconhecido pela linguagem e este deve ser persistente durante toda a execução do código. Por se tratar de uma linguagem quase fortemente tipada nem todos os erros são especificados mas sempre deve ocorrer avisos. O código pode ser estruturado por meio de funções que se relacionam podendo ou não retornar algum valor, no caso de não retornar nada assumimos que a função em questão é do tipo void.

A linguagem conta com dois tipos de dados básicos sendo eles “inteiro” e “flutuante” suporta também estruturas de dados unidimensional (array) e bidimensional (matriz). Com tudo isso é possível realizar uma série de operações aritméticas e lógicas, assim com em outras linguagem como por exemplo C/C++.

A Tabela 1 apresenta os principais símbolos e palavras reservadas suportadas pela linguagem T++, nesta etapa de análise léxica iremos nos referir a cada item da tabela como Tokens. Porém ainda podem ser definidos como tokens, números com um ou mais dígitos que podem ser inteiro, flutuante ou notação científica, identificadores que começam com uma letra e precedem com N letras e números e por fim comentários cercados de chaves {...} podendo conter uma ou mais linhas. Todos estes tokens posteriormente serão devidamente relacionados gerando assim a toda a lógica do código fonte.

palavras reservadas	símbolos
se	+ soma
então	- subtração
senão	* multiplicação
fim	/ divisão
repita	= igualdade
flutuante	, vírgula
retorna	:= atribuição
até	< menor
leia	> maior
escreva	<= menor-igual
inteiro	>= maior-igual
	(abre-par
) fecha-par
	: dois-pontos
	[abre-col
] fecha-col
	&& e-logico
	ou-logico
	! negação

Tabela 1. Tokens da linguagem T++

O código fonte da figura 1 em T++ calcula o fatorial de um número passado pelo input do teclado, é possível visualizar o uso de algumas palavras reservadas e símbolos em ação no código. É notório que algumas palavras reservadas como “se, repita, senão ...” estão sempre aninhadas da palavra reservada “fim” que denota o fechamento do bloco, função está realizada pelo abre e fecha chaves na linguagem C/C++.

3. Análise Léxica

Análise léxica é a primeira fase do processo de compilação de um código fonte, nessa etapa o componente léxico realiza a varredura pelo código fonte carácter por carácter coletando as palavras e separando em conjuntos de marcas (tokens) que podem ser manipulados de forma mais fácil por um parser leitor de saída.

São reconhecidas as palavras reservadas, identificadores, constantes, símbolos entre outras palavras da linguagem em questão. Também é nessa etapa que é tratado espaços

```

1 ▼ inteiro principal()
2     inteiro: digitado
3     inteiro: i
4     i := 1
5 ▼     repita
6         flutuante: f
7         inteiro: int
8         flutuante: resultado
9         f := i/2.0
10        int := i/2
11        resultado := f - int
12
13        se resultado > 0
14            escreva (i)
15        fim
16        i := i+1
17    até i <= digitado
18 fim

```

Figura 1. Fatorial em T++

em branco, contagem das linhas e eliminação de comentários simplificando a saída para os próximos componentes do processo.

Cada conjunto de caracteres ou palavra é reconhecida por uma regra, no seguinte trabalho as regras foram elaboradas por meio de expressões regulares bem definidas que realizam a filtragem e separação de cada token com o seu devido valor.

3.1. Expressão Regular e Linguagens Formais

Para esse tipo de tarefa é de extrema importância a percepção da necessidade da utilização de linguagens formais para validar os tokens corretamente. A linguagem natural apesar de muito usual em diversos aspectos ela não é recomendada processos validativos pois pode trazer ambiguidade e uma complexidade ainda maior ao problema. Por isso a escolha de uma linguagem formal que seja simples, concisa, clara e sem ambiguidade é tão importante.

Por meio das expressões regulares é possível determinar uma cadeia de caracteres específicos para ser reconhecido, um exemplo é a expressão regular $r'senão'$ para expressão só será reconhecido palavras com a exata ordem de caracteres determinado. Para reconhecer palavras de forma menos engessada pode ser usado um range de caracteres como por exemplo $r'[A-Z]'$ que reconhece qualquer caractere do alfabeto (maiúsculo) que esteja dentro do range definido. Também é possível definir expressões que aceitam repetição de caracteres, números ou símbolos, um exemplo é a expressão regular $r'[A-Za-z]+[0-9_]*'$ que pode aceitar palavras que comece com pelo menos um carácter maiúsculo ou minúsculo e nenhum ou vários caracteres numéricos de 0 a 9 podendo haver o símbolo underscore ou não, aceitando palavras como *[rafa, RAFAEL, R, a56-, w-, ...]*.

3.2. Autômatos Finitos

Através de autômatos finitos podemos representar o funcionamento de expressões regulares visualizando os estados de aceitação e transições. Nesta sessão serão apresentados os autômatos finitos referentes às expressões regulares implementadas no analisador léxico, é possível veras com mais detalhes na seção 4 deste documento.

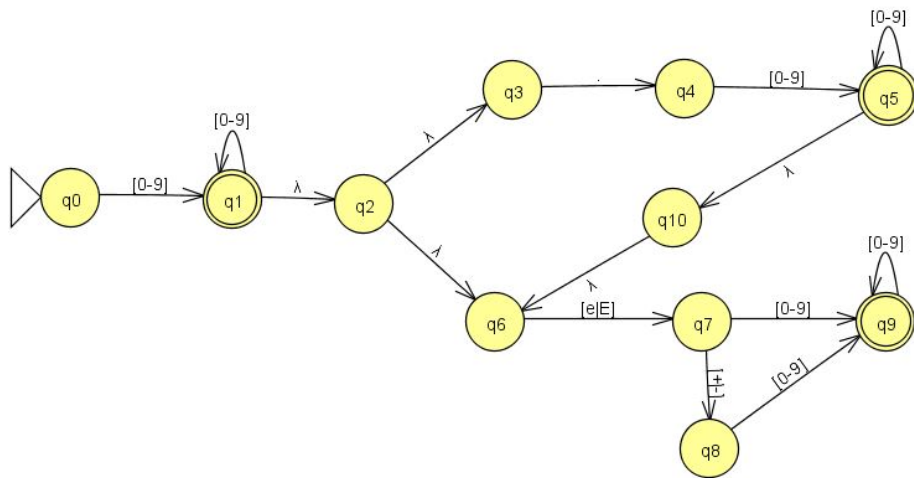


Figura 2. Reconhece números (Inteiros, Ponto Flutuante e Notação Científica)

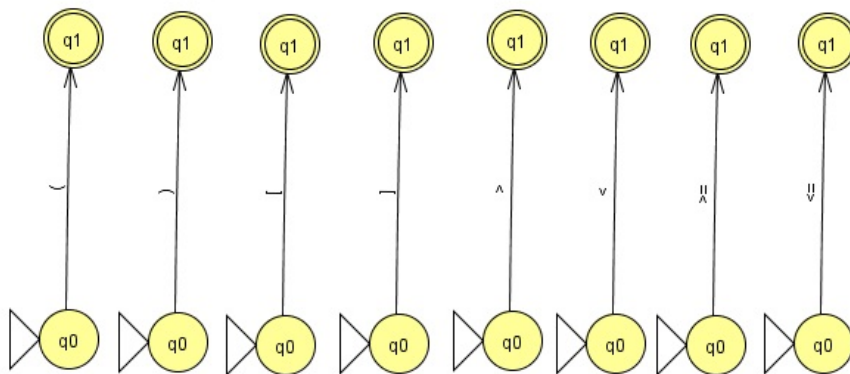


Figura 3. Reconhece operadores aritméticos

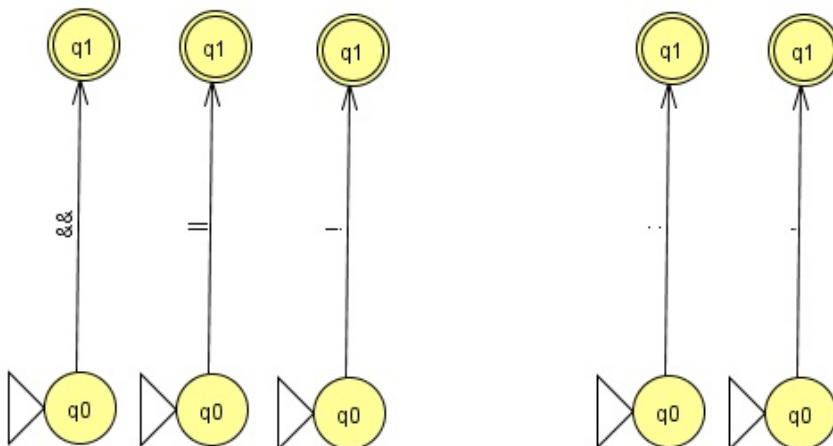


Figura 4. Reconhece símbolos e caracteres especiais da linguagem

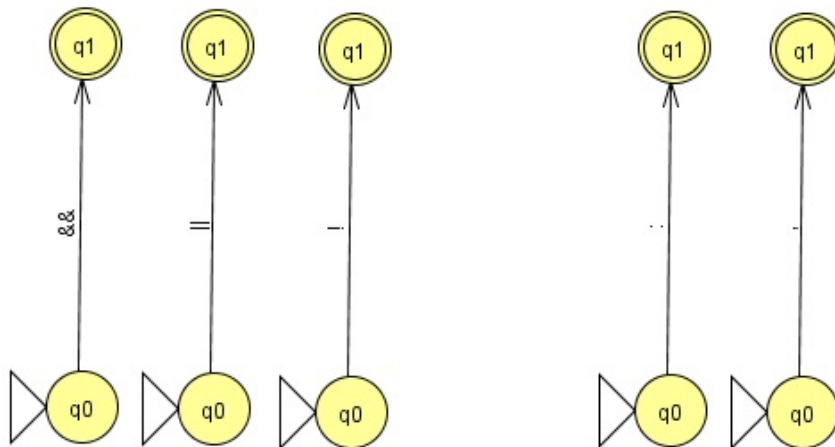


Figura 5. Reconhece Operadores lógicos e pontuações

4. Detalhes da implementação

Nesta sessão serão abordados detalhes da implementação do analisador léxico bem como as ferramentas e métodos utilizados para a realização do mesmo.

4.1. Ferramentas auxiliares

Para a implementação e realização deste trabalho algumas ferramentas auxiliares foram utilizadas. As sessões seguintes descrevem um pequeno resumo das ferramentas.

4.1.1. Python

A linguagem de programação usada para a implementação deste projeto foi a linguagem python pois oferece uma gama maior de ferramentas de fácil acesso facilitando a implementação.

4.1.2. PLY

A ferramenta PLY se trata de uma implementação do analisador léxico (lex) e (yacc), com esta ferramenta é possível junto com a linguagem de programação Python definir regras de precedências, gramáticas e etc.

4.2. Implementação

Para a implementação do analisador léxico foi necessário definir tokens e palavras reservadas da linguagem T++. Estes foram definidos em duas listas presentes no código nas linhas [3 a 49], assim que definidos foi necessário definir para cada token uma expressão regular para que fosse reconhecido, podemos verificar isto nas linhas [51 a 87]. Para as palavras reservadas do dicionário RESERVED não foi necessário definir as expressões regulares pois o próprio dicionário de palavras é capaz de mapear isto.

Como o analisador léxico é responsável por notificar possíveis erros e ignorar comentários no código, foi implementado duas funções presentes nas linhas [89 a 98] que realizam esta função respectivamente.

As linhas [100 a 108] apenas realizam a chamada da função lex(), a abertura do arquivo a ser analisado e o processo de impressão dos tokens.

```
1 import ply.lex as lex
2 import sys
3
4 RESERVED = {
5     'se': 'SE',
6     'senão': 'SENAO',
7     'então': 'ENTAO',
8     'repita': 'REPITA',
9     'escreva': 'ESCREVA',
10    'leia': 'LEIA',
11    'até': 'ATE',
12    'fim': 'FIM',
13    'inteiro': 'INTEIRO',
14    'flutuante': 'FLUTUANTE',
15    'retorna': 'RETORNA',
16 }
17 tokens = [ 'ID',
18            'DOIS_PONTOS',
19            'ATRIBUICAO',
20            'MENOR',
21            'MAIOR',
22            'MENOR_IGUAL',
23            'MAIOR_IGUAL',
24            'MULT',
25            'VIRGULA',
26            'MAIS',
27            'MENOS',
28            'DIVIDE',
29            'ABRE_PAREN',
30            'FECHA_PAREN',
31            'ABRE_COUCH',
32            'FECHA_COUCH',
33            'SE',
34            'SENAO',
35            'ENTAO',
36            'REPITA',
37            'ESCREVA',
38            'LEIA',
39            'ATE',
40            'FIM',
41            'INTEIRO',
42            'FLUTUANTE',
43            'COMPARACAO',
44            'NUMERO',
45            'RETORNA',
46            'E_LOGICO',
47            'OU_LOGICO',
48            'NEGACAO',
49            'QUEBRA_LINHA',
50        ]
51
52 t_ignore = " \t\n"
53
54 ##operadores
55 t_MAIS = r'\+'
56 t_MENOS = r'\-'
57 t_MULT = r'\*'
58 t_DIVIDE = r'\/'
59 t_COMPARACAO = r'\='
60
61 ##símbolos
62 t_ABRE_PAREN = r'\('
63 t_FECHA_PAREN = r'\)'
64 t_ABRE_COUCH = r'\['
65 t_FECHA_COUCH = r'\]'
66 t_MENOR = r'<'
```

```

67 t_MAIOR = r'>'
68 t_ATRIBUICAO = r':='
69 t_MENOR_IGUAL = r'<='
70 t_MAIOR_IGUAL = r'>='
71
72 ##operadores lógicos
73 t_E_LOGICO = r'&&'
74 t_OU_LOGICO = r'\|\|'
75 t_NEGACAO = r'!'
76
77 ##pontos
78 t_DOIS_PONTOS = r':'
79 t_VIRGULA = r','
80
81 ##números
82 t_NUMERO = r'([0-9]+)(\.[0-9]+)?([e|E][+|-]?[0-9]+)?'
83
84 ##identificadores
85 def t_ID(t):
86     r'[A-Za-z_][\w]*'
87     t.type = RESERVED.get(t.value, 'ID')
88     return t
89
90 ##comentários
91 def t_comment(t):
92     r'\{[^\}]*[^\{]*\}'
93     pass
94
95 ##tratamento de erros
96 def t_error(t):
97     print("Erro", t.value[0])
98     raise SyntaxError("ERRO", t.value[0])
99     t.lexer.skip(1)
100
101 lex.lex()
102
103 file = open(sys.argv[1], "r", encoding="utf-8")
104 lex.input(file.read())
105 while True:
106     tok = lex.token()
107     if not tok: break
108     print( tok.type, ': ', tok.value )

```

5. Exemplos de saída do Analisador Léxico

Nesta seção serão apresentados alguns exemplos de saída quando o analisador léxico é submetido a códigos testes. A função responsável por imprimir a saída foi manipulada para assumir uma estrutura mais simples e facilitar o entendimento seguindo sempre a lógica de “TOKEN : valor”.

```

1 ▼ inteiro principal()
2     inteiro: digitado
3     inteiro: i
4     i := 1
5 ▼     repita
6         flutuante: f
7         inteiro: int
8         flutuante: resultado
9         f := i/2.0
10        int := i/2
11        resultado := f - int
12
13        se resultado > 0
14            escreva (i)
15        fim
16        i := i+1
17    até i <= digitado
18 fim

```

Figura 6. Verifica se é numero primo em T++ (ENTRADA)

1 INTEIRO: inteiro	12 ATRIBUICAO ::=	23 ID: resultado	34 ID: resultado	45 ID: i	56 ID: digitado
2 ID: principal	13 NUMERO: 1	24 ID: f	35 ATRIBUICAO ::=	46 FECHA_PAREN:)	57 FIM: fim
3 ABRE_PAREN: (14 REPITA: repita	25 ATRIBUICAO ::=	36 ID: f	47 FIM: fim	
4 FECHA_PAREN:)	15 FLUTUANTE: flutuante	26 ID: i	37 MENOS: -	48 ID: i	
5 INTEIRO: inteiro	16 DOIS_PONTOS: ::	27 DIVIDE: /	38 ID: int	49 ATRIBUICAO ::=	
6 DOIS_PONTOS: ::	17 ID: f	28 NUMERO: 2.0	39 SE: se	50 ID: i	
7 ID: digitado	18 INTEIRO: inteiro	29 ID: int	40 ID: resultado	51 MAIS: +	
8 INTEIRO: inteiro	19 DOIS_PONTOS: ::	30 ATRIBUICAO ::=	41 MAIOR: >	52 NUMERO: 1	
9 DOIS_PONTOS: ::	20 ID: int	31 ID: i	42 NUMERO: 0	53 ATE: até	
10 ID: i	21 FLUTUANTE: flutuante	32 DIVIDE: /	43 ESCREVA: escreva	54 ID: i	
11 ID: i	22 DOIS_PONTOS: ::	33 NUMERO: 2	44 ABRE_PAREN: (55 MENOR_IGUAL: <=	

Figura 7. Verifica se é numero primo em T++ (SAIDA)

```

1 inteiro: vet[10]
2 inteiro: tam
3
4 tam := 10
5
6 { preenche o vetor no pior caso }
7 preencheVetor()
8     inteiro: i
9     inteiro: j
10    i := 0
11    j := tam
12    repita
13        vet[i] = j
14        i := i + 1
15        j := j - 1
16    até i < tam
17 fim
18 { implementação do bubble sort }
19 ▼ bubble_sort()
20     inteiro: i
21     i := 0
22 ▼     repita
23         inteiro: j
24         j := 0
25 ▼         repita
26             se vet[i] > v[j] então
27                 inteiro: temp
28                 temp := vet[i]
29                 vet[i] := vet[j]
30                 vet[j] := temp
31             fim
32             j := j + 1
33         até j < i
34         i := i + 1
35     até i < tam
36 fim
37 { programa principal }
38 ▼ inteiro principal()
39     preencheVetor()
40     bubble_sort()
41     retorna(0)
42 fim

```

Figura 8. Algoritmo de ordenação BubbleSort em T++ (ENTRADA)

1 INTEIRO : inteiro	24 NUMERO : 0	47 MENOR : <	70 ID : i	93 ABRE_COUCH : [116 NUMERO : 1
2 DOIS_PONTOS : ::	25 ID : j	48 ID : tam	71 FECHA_COUCH :]	94 ID : j	117 ATE : até
3 ID : vet	26 ATRIBUICAO : :=	49 FIM : fim	72 MAIOR : >	95 FECHA_COUCH :]	118 ID : i
4 ABRE_COUCH : [27 ID : tam	50 ID : bubble_sort	73 ID : v	96 ID : vet	119 MENOR : <
5 NUMERO : 10	28 REPITA : repita	51 ABRE_PAREN : (74 ABRE_COUCH : [97 ABRE_COUCH : [120 ID : tam
6 FECHA_COUCH :]	29 ID : vet	52 FECHA_PAREN :)	75 ID : j	98 ID : j	121 FIM : fim
7 INTEIRO : inteiro	30 ABRE_COUCH : [53 INTEIRO : inteiro	76 FECHA_COUCH :]	99 FECHA_COUCH :]	122 INTEIRO : inteiro
8 DOIS_PONTOS : ::	31 ID : i	54 DOIS_PONTOS : ::	77 ENTAO : então	100 ATRIBUICAO : :=	123 ID : principal
9 ID : tam	32 FECHA_COUCH :]	55 ID : i	78 INTEIRO : inteiro	101 ID : temp	124 ABRE_PAREN : (
10 ID : tam	33 COMPARACAO : =	56 ID : i	79 DOIS_PONTOS : ::	102 FIM : fim	125 FECHA_PAREN :)
11 ATRIBUICAO : :=	34 ID : j	57 ATRIBUICAO : :=	80 ID : temp	103 ID : j	126 ID : preencheVetor
12 NUMERO : 10	35 ID : i	58 NUMERO : 0	81 ID : temp	104 ATRIBUICAO : :=	127 ABRE_PAREN : (
13 ID : preencheVetor	36 ATRIBUICAO : :=	59 REPITA : repita	82 ATRIBUICAO : :=	105 ID : j	128 FECHA_PAREN :)
14 ABRE_PAREN : (37 ID : i	60 INTEIRO : inteiro	83 ID : vet	106 MAIS : +	129 ID : bubble_sort
15 FECHA_PAREN :)	38 MAIS : +	61 DOIS_PONTOS : ::	84 ABRE_COUCH : [107 NUMERO : 1	130 ABRE_PAREN : (
16 INTEIRO : inteiro	39 NUMERO : 1	62 ID : j	85 ID : i	108 ATE : até	131 FECHA_PAREN :)
17 DOIS_PONTOS : ::	40 ID : j	63 ID : j	86 FECHA_COUCH :]	109 ID : j	132 RETORNA : retorna
18 ID : i	41 ATRIBUICAO : :=	64 ATRIBUICAO : :=	87 ID : vet	110 MENOR : <	133 ABRE_PAREN : (
19 INTEIRO : inteiro	42 ID : j	65 NUMERO : 0	88 ABRE_COUCH : [111 ID : i	134 NUMERO : 0
20 DOIS_PONTOS : ::	43 MENOS : -	66 REPITA : repita	89 ID : i	112 ID : i	135 FECHA_PAREN :)
21 ID : j	44 NUMERO : 1	67 SE : se	90 FECHA_COUCH :]	113 ATRIBUICAO : :=	136 FIM : fim
22 ID : i	45 ATE : até	68 ID : vet	91 ATRIBUICAO : :=	114 ID : i	
23 ATRIBUICAO : :=	46 ID : i	69 ABRE_COUCH : [92 ID : vet	115 MAIS : +	

Figura 9. Algoritmo de ordenação BubbleSort em T++ (SAIDA)

6. Análise Sintática

O analisador sintático também conhecido em outras literaturas como parser tem como principal objetivo, verificar se determinada cadeia de tokens possuem ou não sentenças válidas ou seja se faz ou não parte das regras gramaticais da linguagem em questão podendo assim validar ou não expressões.

Trata-se da segunda etapa do processo de compilação, sua entrada de dados são tokens reconhecidos pelo processo anterior de análise léxica. A saída ao término da análise sintática será uma árvore sintática contendo nos nós a estrutura do código submetido ao processo de compilação.

É durante esta etapa que possível descobrir se há erros sintáticos no código por exemplo se em um declaração de variável o tipo não estiver presente o analisador sintático através de regras gramaticais e classes de erros previamente declaradas irá tomar a decisão de alertar o usuário com tipo do erro e linha correspondente, interrompendo a compilação e não permitindo que o erro se propague para as próximas etapas do processo.

6.1. Gramática Livre de Contexto

É possível determinar as regras gramaticais de uma linguagem de programação através de Gramáticas Livre de Contexto (GCL) que se diferencia das expressões regulares simples por suportar regras recursivas, para a linguagem de programação abordada neste trabalho foi elaborado um documento [201 2016] contendo as regras gramaticais escritas no Formalismo de Backus-Naur (BNF).

Uma Gramática Livre de Contexto pode ser representada pelos seguintes itens:

- Conjunto finito de terminais (T)
- Conjunto finito de não-terminais (N)
- Símbolo inicial da gramática (S)
- Um conjunto de produções (P)

Breve explicação sobre cada item:

- **Símbolos Terminais:** Os conjuntos terminais são basicamente os tokens reconhecidos pela a linguagem
- **Símbolos Não-Terminais:** Conjuntos não terminais são formados por símbolos terminais e outros símbolos não terminais.
- **Símbolo Inicial:** Indica o início da definição da linguagem
- **Regras de Produção:** Representa o conjunto das regras sintáticas da linguagem indicando como os símbolos terminais e não terminais podem ser combinados.

Cada símbolo deve passar por um processo de derivação começando pelo símbolo inicial (S) até chegar em símbolos terminais, ou seja enquanto houver símbolos ou expressões não-terminais, estes devem passar pelo processo de derivação aplicando as regras da gramática até chegar em apenas símbolos terminais, os símbolos terminais caracterizam o fim de um ramo da árvore sintática ou seja nós folhas, sendo assim os nós intermediários são caracterizados como não-terminais e devem ser derivados.

Existem algumas abordagens de derivações sintática, cada compilador utiliza a forma que mais lhe convém, os itens a seguir mostram as formas de derivação existentes. Como estamos utilizando a ferramenta PLY ficamos limitados a utilização da derivação que a ferramenta implementa que é a LR (Left-to-Right) porem independente do algoritmo utilizado a derivação deve produzir o mesmo resultado, ou seja, a mesma árvore de derivação, caso contrario significa que ha fortes indícios de que a gramatica esteja ambígua, nos próximos parágrafos será apresentado em maiores detalhes cada forma de derivação e seu funcionamento.

Tipos de Analise:

- **Top-Down ou Descendente:** Examina a cadeia de tokens da esquerda para a direita, forma a árvore sintática de cima para baixo. Percorre a árvore em pré-ordem ou seja da raiz para as folhas. Dentro desta categoria temos as seguintes técnicas:
 - **Analizador sintático preditivo:** Tenta prever a construção seguinte na cadeia de entrada com base nas marcas de verificação à frente.
 - **Analizador sintático com retrocesso:** Testa diferentes possibilidades de análise sintática de entrada. Em caso de falha ha um processo de retrocesso ate o nível mais estável. Por conta do retrocesso estes acabam ganhando uma característica mais lenta pois requerem tempo exponencial para a execução.
- **Bottom-Up ou Ascendente:** Examina os símbolos terminais da direita para a esquerda,forma a árvore sintática de baixo para cima. O processo de derivação é reduitiva ou seja a derivação ocorre ate atingir o simbolo não-terminal inicial.Dentro desta categoria ha diferentes técnicas de derivação como por exemplo (LL, LR, SLR)

A ferramenta PLY implementa no modulo yacc uma analise Button-Up informação esta encontrada e explicada na documentação da ferramenta "O Yacc usa uma técnica de análise conhecida como análise LR ou análise de redução de deslocamento. A análise LR é uma técnica de baixo para cima que tenta reconhecer o lado direito de várias regras gramaticais. Sempre que um lado direito válido é encontrado na entrada, o código de ação apropriado é acionado e os símbolos de gramática são substituídos pelo símbolo de gramática no lado esquerdo."[Beazley 2018]

6.2. Detalhes da implementação e Resultados

A implementação do analisador léxico faz uso da ferramenta PLY e seu módulo Yet Another Compiler Compiler (YACC), este módulo implementa um componente de análise.

Antes da declaração das regras gramaticais, foi elaborada uma estrutura de dados semelhante a uma árvore, cada nó tem um tipo, valor e um vetor de filhos que podem ou não serem nulos. O trecho de código abaixo mostra a declaração da classe da estrutura.

```
1 def __init__(self, type_node='', child=[], value=''):
2     self.type = type_node
3     self.child = child
4     self.value = value
```

Assim que declarado a estrutura de dados, foi declarado a gramática através do documento BNF Comentada que lista todas as regras e ações possíveis para a linguagem. A declaração das regras foi feita por meio de funções dentro da classe do analisador. O código a seguir mostra um exemplo de como foi declarado, vale observar que o segundo argumento da Tree são os filhos do nó em questão.

```
1 def p_programa(self, p):
2     '''programa : lista_declaracoes'''
3     p[0] = Tree('programa', [p[1]])
4
5 def p_lista_declaracoes(self, p):
6     '''lista_declaracoes : lista_declaracoes declaracao
7                             | declaracao'''
8     if len(p) == 3:
9         p[0] = Tree('lista_declaracoes', [p[1], p[2]])
10    else:
11        p[0] = Tree('lista_declaracoes', [p[1]])
```

Afim de manter a coerência e diminuir qualquer tipo de conflito ou ambiguidade que possa existir foi declarado no analisador as precedências de sinais como de multiplicação, divisão e outros, o trecho de código a seguir mostra como isto foi implementado de fato.

```
1 #Definição das precedências
2 self.precedence = (
3     ('left', 'COMPARACAO', 'MAIOR_IGUAL', 'MAIOR', 'MENOR_IGUAL',
4      'MENOR'),
5     ('left', 'MAIS', 'MENOS'),
6     ('left', 'MULT', 'DIVIDE'),
7 )
```

Para o tratamento dos erros durante a análise foram implementadas as funções de erros mais comuns, assim que ocorre a invocação de alguma definição de erro há a interrupção do processo e mostrando na saída do terminal a linha com o erro, qual é o erro e uma pequena dica para solucionar. O código a seguir mostra um exemplo de declarações de erros implementadas no analisador sintático.

```
1 def p_error(self, p):
2     if p:
```

```

3         print("Erro Sintatico: '%s' na linha '%d'" %(p.value, p.
          lineno))
4     else:
5         print("Erro Fatal!@")
6
7 def p_leia_error(self, p):
8     '''leia : LEIA ABRE_PAREN error FECHA_PAREN'''
9     print("N O FOI POSSIVEL REALIZAR A LEITURA (ARGUMENTO
        INVALIDO) ")
10    exit(1)
11
12 def p_escrava_error(self, p):
13     '''escrava : ESCRIVA ABRE_PAREN error FECHA_PAREN '''
14     print("N O FOI POSSIVEL REALIZAR A ESCRITA (ARGUMENTO
        INVALIDO) ")
15    exit(1)

```

Para imprimir a estrutura da árvore foi usado o pacote Graphviz que facilita a criação e renderização de descrições gráficas, este pacote foi de suma importância no desenvolvimento desta etapa pois consegue desenhar a estrutura da árvore de saída e suas derivação facilitando a visualização e derivações de cada expressão. Na sessão de Exemplos de Saída temos um exemplo que mostra uma árvore sintática desenhada por tal pacote. O trecho de código a seguir apresenta como é feito o processo de impressão recursiva da estrutura de árvore usando o Graphviz.

```

1 from graphviz import Digraph
2
3 def print_tree(node, dot, i="0", pai=None):
4     if node != None:
5         filho = str(node) + str(i)
6         dot.node(filho, str(node))
7         if pai: dot.edge(pai, filho)
8         j = "0"
9         if not isinstance(node, Tree): return
10        for son in node.child:
11            j+="1"
12            print_tree(son, dot, i+j, filho)

```

6.2.1. Exemplo de Entrada

Como dito anteriormente o componente analisador sintático tem como entrada a lista de tokens gerados pela análise léxica. Para testar o analisador sintático usaremos o código teste da Figura 10. Iremos passar o código teste via argumento para o analisador sintático que ira usar o analisador Léxico para gera uma lista de tokens ilustrados na Figura 11. A lista de tokens ira alimentar o analisador léxico que de acordo com as regras da gramatica definidas searão validadas as expressões.

```

1 inteiro: n
2
3 inteiro fatorial(inteiro: n)
4     inteiro: fat
5     se n > 0 então {não calcula se n > 0}
6         fat := 1
7         repita
8             fat := fat * n
9             n := n - 1
10        até n = 0
11        retorna(fat) {retorna o valor do fatorial de n}
12    senão
13        retorna(0)
14    fim
15 fim
16
17 inteiro principal()
18     leia(n)
19     escreva(fatorial(n))
20     retorna(0)
21 fim

```

Figura 10. Código Teste (ENTRADA Analisador Léxico)

1 INTEIRO : inteiro	24 ATRIBUICAO ::=	47 FIM : fim
2 DOIS_PONTOS ::	25 ID : fat	48 INTEIRO : inteiro
3 ID : n	26 MULT : *	49 ID : principal
4 INTEIRO : inteiro	27 ID : n	50 ABRE_PAREN : (
5 ID : fatorial	28 ID : n	51 FECHA_PAREN :)
6 ABRE_PAREN : (29 ATRIBUICAO ::=	52 LEIA : leia
7 INTEIRO : inteiro	30 ID : n	53 ABRE_PAREN : (
8 DOIS_PONTOS ::	31 MENOS : -	54 ID : n
9 ID : n	32 INTEIRO : 1	55 FECHA_PAREN :)
10 FECHA_PAREN :)	33 ATE : até	56 ESCREVA : escreva
11 INTEIRO : inteiro	34 ID : n	57 ABRE_PAREN : (
12 DOIS_PONTOS ::	35 COMPARACAO :=	58 ID : fatorial
13 ID : fat	36 INTEIRO : 0	59 ABRE_PAREN : (
14 SE : se	37 RETORNA : retorna	60 ID : n
15 ID : n	38 ABRE_PAREN : (61 FECHA_PAREN :)
16 MAIOR : >	39 ID : fat	62 FECHA_PAREN :)
17 INTEIRO : 0	40 FECHA_PAREN :)	63 RETORNA : retorna
18 ENTAO : então	41 SENAO : senão	64 ABRE_PAREN : (
19 ID : fat	42 RETORNA : retorna	65 INTEIRO : 0
20 ATRIBUICAO ::=	43 ABRE_PAREN : (66 FECHA_PAREN :)
21 INTEIRO : 1	44 INTEIRO : 0	67 FIM : fim
22 REPITA : repita	45 FECHA_PAREN :)	
23 ID : fat	46 FIM : fim	

Figura 11. Código Teste (ENTRADA Analisador Sintático)

6.2.2. Exemplo de Saída

A lista de tokens mostrada na Figura 11 quando submetida ao módulo da análise sintática resulta em um arquivo PDF contendo a árvore sintática do código como mostrado na Figura 13, é notório que devido as dimensões da árvore fica inviável a leitura dos nós da árvore, sendo assim disponibilizo um link de um arquivo PDF com maior resolução para eventuais análises. https://mega.nz/#!URcGVaCI!NWOS7ltDwoFlXZRQMu4PiY7hVOVKldMn_2b7PnuHz30

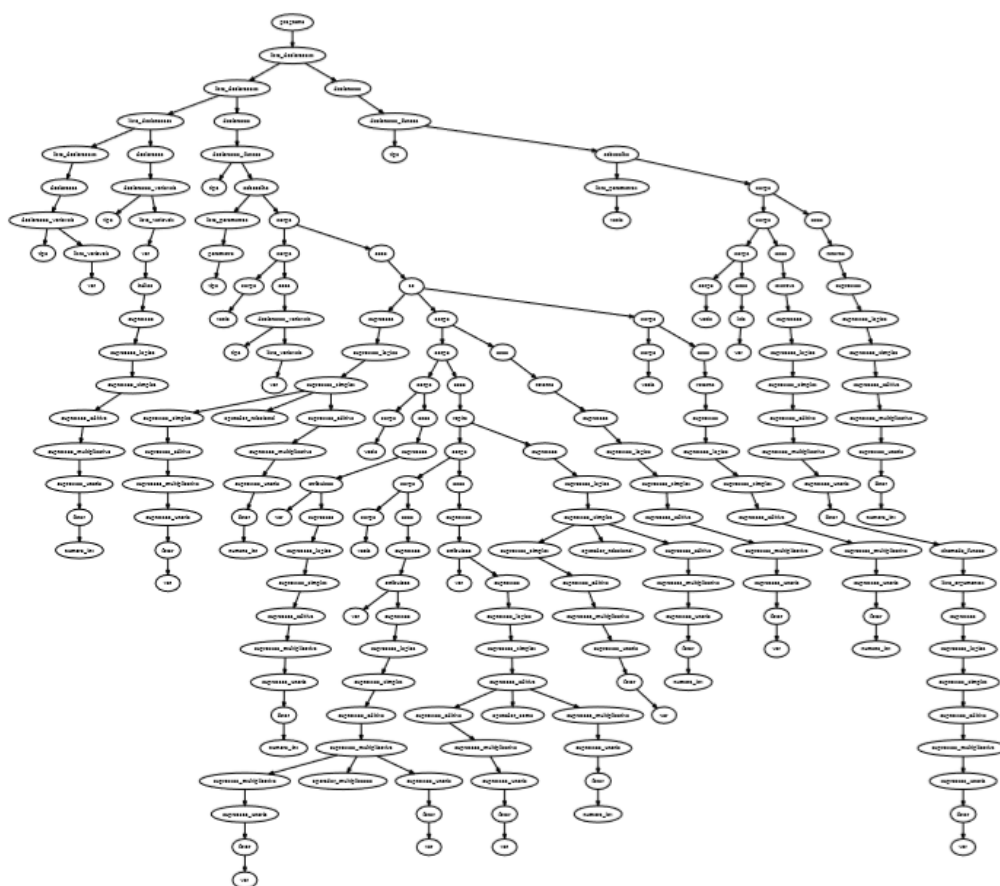


Figura 12. Árvore Sintática (SAÍDA Analisador Sintático)

7. Análise Semântica

A análise semântica representa a quarta etapa do processo de compilação, é nesta etapa que é identificado erros semânticos que possam ocorrer no código. É de responsabilidade do analisador sintático mostrar para o usuário em forma de avisos quais erros ocorreram de forma que facilite o usuário no entendimento do problema e sua possível resolução.

As classes de erros no analisador implementado pode ser separados em dois níveis, sendo o primeiro:

“Aviso:” que não interrompe o processo de compilação mas mostra para o usuário oque está acontecendo, geralmente estes avisos ocorrem quando há erros semânticos como atribuição de um valor com uma variável de tipos diferentes, multiplicação de valores ou variáveis de tipos diferentes, na maioria da vezes quando ocorrem problemas desta categoria, o analisador semântico indica que ouve alguma conversão, ou alguma tomada de providencia.

“Erro:” esta classe de erro interrompe imediatamente o processo de compilação assim que encontrado pois é entendível que erros desta categoria causam resultados prejudiciais que não devem ser propagados para a próximas etapas. Este tipo de erro acontece por exemplo quando há no código uma chamada de função que não foi declarada ou quando não está declarada a função principal. Como se trata de erros mais complexos o

analisador semântico opta por não trabalhar em uma solução ou conversão, apenas mostra a mensagem de erro correspondente para o usuário para que possa sancionar o erro.

8. Detalhes da implementação e Resultados

A entrada para o analisador semântico é árvore sintática provinda do analisador sintático. Como se trata de uma árvore onde os nós folhas quando juntos representam o código, algumas decisões foram tomadas para tornar o processo de análise semântica mais simples e lógico.

A primeira decisão tomada foi podar a árvore de entrada, tendo em vista que muitos dos nós da árvore não eram úteis para o procedimento, por isso estes foram removidos da árvore se seus nós filhos foram rearranjados novamente a fim de manter a estrutura lógica.

Assim que podada a árvore a segunda procuração posta em jogo, foi a maneira de percorrer a árvore. Então foi decidido que a leitura dos nós da árvore usaria uma abordagem da esquerda para a direita semelhante a uma busca em profundidade, isso pois queremos sempre verificar os nós folhas de cada ramo para conhecer os valores do código. A figura a seguir apresenta um exemplo da poda realizada em um código simples de demonstração contendo duas funções vazias.

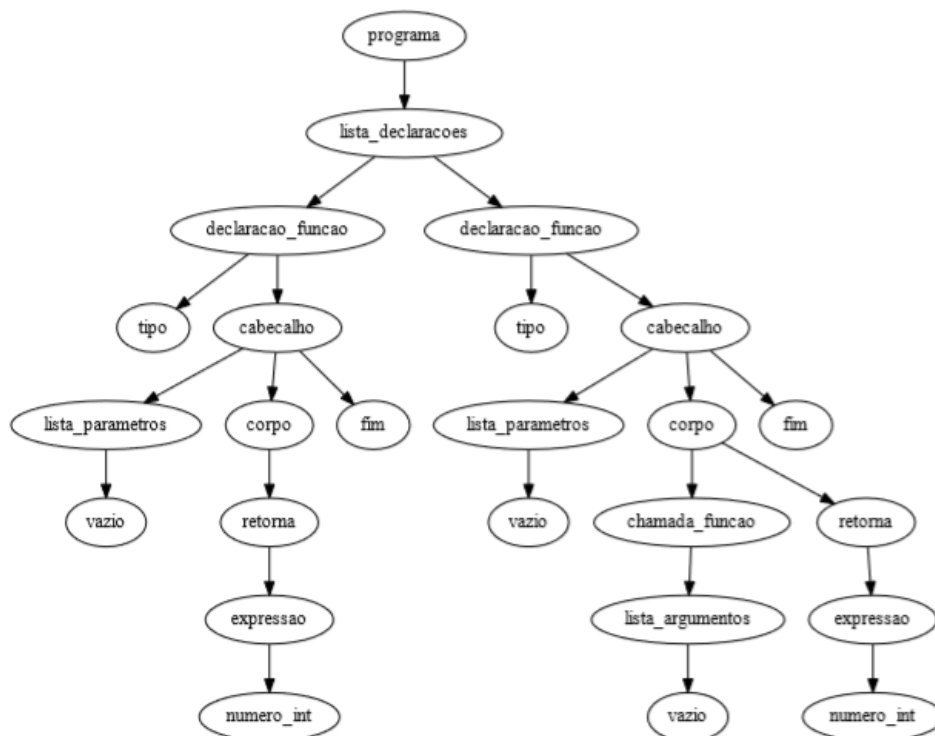


Figura 13. Exemplo simples de árvore Podada

Tendo isso definido a próxima decisão que foi tomada foi como organizar de forma lógica as variáveis e funções que forem sendo encontradas para realizar futuras verificações. Assim então foi elaborado uma estrutura de classes bem definidas com

seus respectivos métodos para atuarem nos parâmetros. A seguir é possível verificar a descrição de cada classe e um breve trecho de código para representá-las.

Classe Elemento: Esta classe representa elementos no código, que podem ser variáveis, vetores, constantes. A cada nova declaração que acontece no código é iniciado um novo elemento desta classe com seus respectivos atributos, que podem sofrer alterações durante o processo de análise.

```
1 class Elemento():
2     def __init__(self, nome, escopo, tipo, indice, used,
3         ehParametro, linha=None):
4         self.nome = nome
5         self.escopo = escopo
6         self.tipo = tipo
7         self.used = used
8         self.indice = indice
9         self.ehParametro = ehParametro
10        self.linha = linha
```

Classe Lista de Funções: Esta classe representa as funções ou qualquer tipo de estrutura que tenha um escopo próprio, esta função pode ser instanciada assim que há uma declaração de função ou até mesmo uma abertura de de uma estrutura condicional ou de laço.

```
1 class ListaFuncoes():
2     def __init__(self, nome, tipo_retorno, lista_parametros,
3         foi_retornada, escopo, used, escopo_pai, linha=None):
4         self.nome = nome
5         self.tipo_retorno = tipo_retorno
6         self.lista_parametros = lista_parametros
7         self.escopo = escopo
8         self.escopo_pai = escopo_pai
9         self.foi_retornada = foi_retornada
10        self.used = used
11        self.linha = linha
```

Classe Tabela de Símbolos: A classe de tabela de símbolos gerencia todas as outras classes e representa o as variáveis e funções do código. Os atributos dela são lista de elementos e lista de funções. Esta classe é instanciada apenas uma vez, no processo de inicialização da análise semântica, e é propagada por toda a análise. A classe em questão possui métodos para interagir com os parâmetros da classe, possibilitando que ocorra inserções e buscas de elementos e funções durante o processo.

```
1 class TabelaSimbolos():
2     TemPrincipal = False
3     def __init__(self):
4         self.lista_elementos = []
5         self.lista_funcoes = []
```

Outra decisão tomada foi de como controlar os escopos de cada função e o que é válido em um escopo específico. Fazendo uma analogia com pilhas foi possível pensar em uma solução cabível, onde quando um novo escopo é declarado ele é empilhado em uma pilha de escopos onde o topo da pilha é o escopo onde está sendo executado a análise, e os escopos anteriores são os escopos ainda abertos e válidos.

Assim que o analisador semântico encontra um nó que representa o fechamento do escopo topo da pilha, este é des-empilhado. Como o escopo global não tem nenhuma *tag* de fechamento ele sempre está presente na pilha de escopos e é declarado logo que inicia a análise. Desta forma acessar os escopos anteriores fica ainda mais intuitivo e fácil.

O analisador semântico conta com uma função recursiva chamada "andar", que quando iniciada recebe o nó raiz da árvore podada. A cada nó que é iterado, é feito algumas verificações, por exemplo se o nó tiver como nome "declaracao_funcao", "declaracao_variavel", "chamada_funcao", o nó correspondente é passado como parâmetro para a função que irá tratar as minúcias, podendo assim manipular por meio de métodos, a estrutura de dados Tabela de Símbolos.

Os erros e avisos são gerados realizando verificações, por exemplo no caso de uma atribuição de um valor de tipo flutuante em uma "variável" de tipo "inteiro" a função responsável pela atribuição verifica se tipos se correspondem, se não corresponderem, uma mensagem de aviso é impressa no console relatando que a atribuição possui tipos diferentes e foi necessário transformar o tipo para o adequado. Existe também funções de verificação de erros que são invocadas ao final da leitura do código fonte.

O trecho de código a seguir mostra um função usada para conferir se todas as variáveis na tabela de símbolos foram inicializadas e utilizadas. Caso isso não ocorra, a função imprime na tela a linha com o problema, nome da variável e a mensagem correspondente.

```
1 def conferir_variaveis_usadas(self):
2     if self.lista_elementos:
3         for elemento in self.lista_elementos:
4             if elemento.used == False:
5                 print ("Linha:[" + str(elemento.linha) + "] Aviso:
6                     Variável '" + str(elemento.nome) + "'
7                     declarada e não utilizada")
8             elif elemento.used == True and elemento.inicializado
9                 == False:
10                print ("Linha:[" + str(elemento.linha) + "] Aviso:
11                    Variável '" + str(elemento.nome) + "' não
12                    inicializada")
```

A saída desta etapa de análise semântica são mensagens de erro e avisos para o usuário e uma árvore podada que será usada na etapa de geração de código. O trecho de código a seguir contém alguns erros semânticos. A figura 14 mostra o que a saída do analisador semântico quando posto em análise o código seguinte.

```
1 inteiro: a
2
3 inteiro principal()
4     flutuante: a
5     flutuante: b
6     a:=10
7     se 1 > a então
8         inteiro: x
9         x:=2
10    fim
11    b := a + 1.0
12 fim
```

```

Linha:[4] Aviso: Variável 'a' já declarada anteriormente
Linha:[11] Aviso: A expressão com o operador '+' trabalha com tipos diferentes flutuante e inteiro houve uma coerção implícita
Erro: Função 'principal' deveria retornar 'inteiro', mas retorna vazio

### TABELA DE SIMBOLOS ###
Tipo: inteiro Nome: a Escopo: global Usada: True
Tipo: flutuante Nome: b Escopo: principal Usada: True
Tipo: inteiro Nome: x Escopo: se Usada: True

### TABELA DE FUNCOES ###
Nome: principal Lista_Parametros: [] Tipo_Retorno: inteiro Escopo: principal Escopo_PAi: global Foi_Retornada: False
| + ----> Nome: se Escopo: se Escopo_PAi: principal

### PILHA DE ESCOPOS ###
global

```

Figura 14. Exemplo Saída Analisador Semântico

Podemos analisar que Figura 14 traz como resultado os erros semânticos encontrados na análise. Trás também a tabela de símbolos, tabela de funções e pilha de escopos. Note que assim que uma variável é declarada ela é adicionada na tabela de símbolos, junto a seu escopo pai.

Podemos observar que a tabela de funções armazena todas as funções declaradas no código e além disto registra funções não convencionais como estruturas de laço e condicional colocando-as sobre uma leve indentação para indicar que este escopo esta aninhado a outro escopo.

A pilha de escopos apresenta somente o escopo global no fim da análise, isso é ótimo pois indica que todos os escopos foram empilhados e devidamente desempilhados assim que encontrado sua devida tag de fechamento.

Referências

- (2016). Bnf comentada. https://docs.google.com/document/d/1oYX-5ipzL_izj_h08s7axuo2OyA279YEhnAItgXzXzXAQ/edit.
- Aho, A., Sethi, R., and Lam, S. (2008). *Compiladores. Princípios, Técnicas e Ferramentas*. LONGMAN DO BRASIL.
- Bank, S. (2018). Graphviz 0.9. <https://pypi.org/project/graphviz/>.
- Beazley, D. M. (2018 (Acessado em 01 de Setembro, 2018)). *PLY (Python Lex-Yacc)*. <http://www.dabeaz.com/ply/ply.html>.
- Marangon, J. D. (2016). *Compiladores para humanos*. <https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/syntax-analysis.html>.
- Wikipédia (2018). *Compilador* - wikipédia, a enciclopédia livre. (Acessado em 01 de Setembro, 2018).