Rafael Menezes Barboza[1]

[1]Universidade 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 logica do codigo 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. Analise 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

```
inteiro principal()
    inteiro: digitado
    inteiro: i
    i := 1
    repita
        flutuante: f
        inteiro: int
        flutuante: resultado
        f := i/2.0
        int := i/2
        resultado := f - int

        se  resultado > 0
            escreva (i)
        fim
        i := i+1
    até i <= digitado
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 minusculo 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 velas com mais detalhes na seção 4 deste documento.

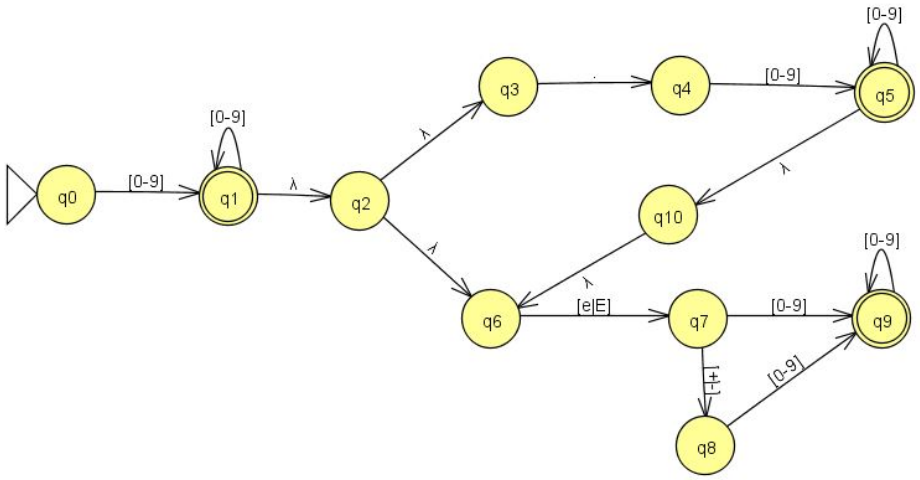

Figura 2. Reconhece números (Inteiros, Ponto Flutuante e Notação Cientifica)

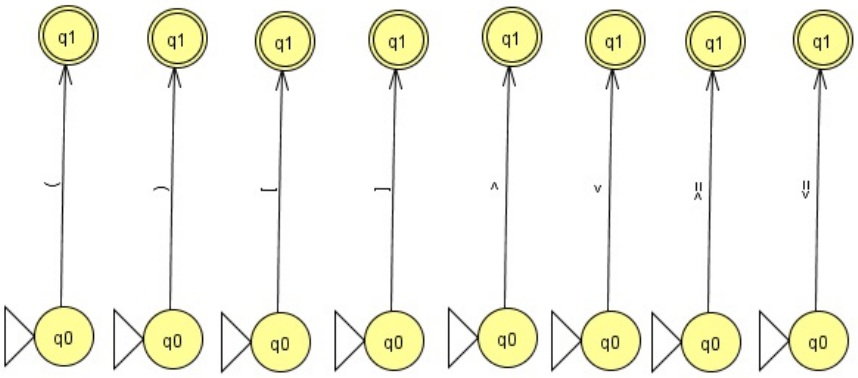

Figura 3. Reconhece operadores aritméticos

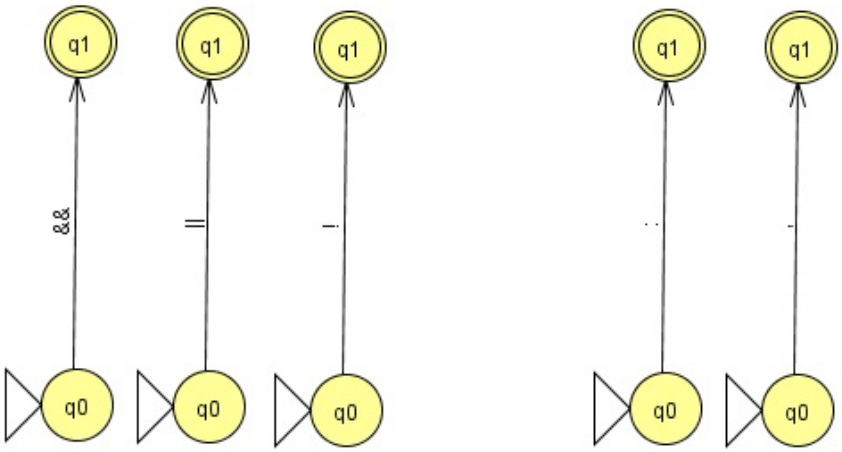

Figura 4. Reconhece símbolos e caracteres especiais da linguagem

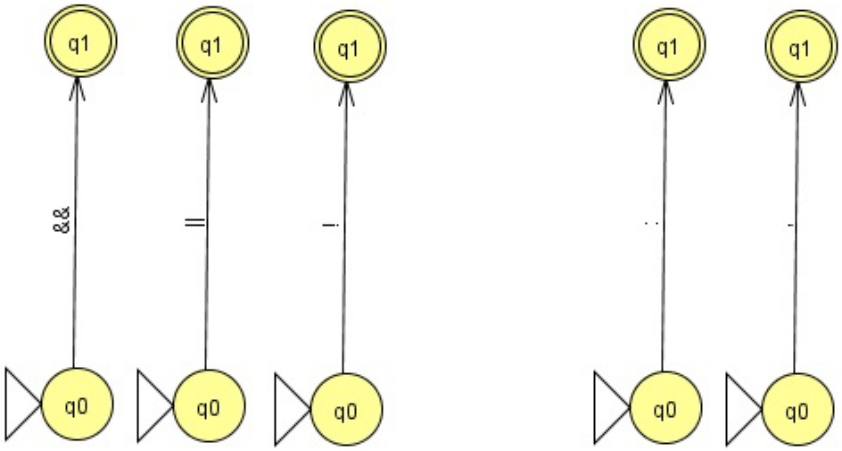

Figura 5. Reconhece Operadores lógicos e pontuações

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.

```
import ply.lex as lex
import sys

RESERVED = {
            'se': 'SE',
            'senão':'SENAO',
            'então':'ENTAO',
            'repita':'REPITA',
            'escreva':'ESCREVA',
            'leia':'LEIA',
            'até':'ATE',
            'fim':'FIM',
            'inteiro' : 'INTEIRO',
            'flutuante' : 'FLUTUANTE',
            'retorna' : 'RETORNA',
}
tokens = ['ID',
            'DOIS_PONTOS',
            'ATRIBUICAO',
            'MENOR',
            'MAIOR',
            'MENOR_IGUAL',
            'MAIOR_IGUAL',
            'MULT',
            'VIRGULA',
            'MAIS',
            'MENOS',
            'DIVIDE',
            'ABRE_PAREN',
            'FECHA_PAREN',
            'ABRE_COUCH',
            'FECHA_COUCH',
            'SE',
            'SENAO',
            'ENTAO',
            'REPITA',
            'ESCREVA',
            'LEIA',
            'ATE',
            'FIM',
            'INTEIRO',
            'FLUTUANTE',
            'COMPARACAO',
            'NUMERO',
            'RETORNA',
            'E_LOGICO',
            'OU_LOGICO',
            'NEGACAO',
            'QUEBRA_LINHA',
            ]

t_ignore = " \t\n"

##operadores
t_MAIS = r'\+'
t_MENOS = r'-'
t_MULT = r'\*'
t_DIVIDE = r'/'
t_COMPARACAO = r'='

##símbolos
t_ABRE_PAREN = r'\('
t_FECHA_PAREN = r'\)'
t_ABRE_COUCH = r'\['
t_FECHA_COUCH = r'\]'
t_MENOR = r'<'
```

```
t_MAIOR = r'>'
t_ATRIBUICAO = r':='
t_MENOR_IGUAL = r'<='
t_MAIOR_IGUAL = r'>='

##operadores lógicos
t_E_LOGICO = r'&&'
t_OU_LOGICO = r'\|\|'
t_NEGACAO = r'!'

##pontos
t_DOIS_PONTOS = r':'
t_VIRGULA = r','

##números
t_NUMERO = r'([0-9]+)(\.[0-9]+)?([e|E][+|-]?[0-9]+)?'

##identificadores
def t_ID(t):
    r'[A-Za-z_][\w]*'
    t.type = RESERVED.get(t.value, 'ID')
    return t

##comentários
def t_comment(t):
    r'\{[^}]*[^{]*\}'
    pass

##tratamento de erros
def t_error(t):
    print("Erro", t.value[0])
    raise SyntaxError("ERRO", t.value[0])
    t.lexer.skip(1)

lex.lex()

file = open(sys.argv[1], "r", encoding="utf-8")
lex.input(file.read())
while True:
    tok = lex.token()
    if not tok: break
    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 ”.

```
inteiro principal()
    inteiro: digitado
    inteiro: i
    i := 1
    repita
        flutuante: f
        inteiro: int
        flutuante: resultado
        f := i/2.0
        int := i/2
        resultado := f - int

        se  resultado > 0
            escreva (i)
        fim
        i := i+1
    até i <= digitado
fim
```

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

```
INTEIRO : inteiro
ID : principal
ABRE_PAREN : (
FECHA_PAREN : )
INTEIRO : inteiro
DOIS_PONTOS : :
ID : digitado
INTEIRO : inteiro
DOIS_PONTOS : :
ID : i
ID : i
ATRIBUICAO : :=
NUMERO : 1
REPITA : repita
FLUTUANTE : flutuante
DOIS_PONTOS : :
ID : f
INTEIRO : inteiro
DOIS_PONTOS : :
ID : int
FLUTUANTE : flutuante
DOIS_PONTOS : :
ID : resultado
ID : f
ATRIBUICAO : :=
ID : i
DIVIDE : /
NUMERO : 2.0
ID : int
ATRIBUICAO : :=
ID : i
DIVIDE : /
NUMERO : 2
ID : resultado
ATRIBUICAO : :=
ID : f
MENOS : -
ID : int
SE : se
ID : resultado
MAIOR : >
NUMERO : 0
ESCREVA : escreva
ABRE_PAREN : (
ID : i
FECHA_PAREN : )
FIM : fim
ID : i
ATRIBUICAO : :=
ID : i
MAIS : +
NUMERO : 1
ATE : até
ID : i
MENOR_IGUAL : <=
ID : digitado
FIM : fim
```

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

```
inteiro: vet[10]
inteiro: tam

tam := 10

{ preenche o vetor no pior caso }
preencheVetor()
  inteiro: i
  inteiro: j
  i := 0
  j := tam
  repita
    vet[i] = j
    i := i + 1
    j := j - 1
  até i < tam
fim
```

```
{ implementação do bubble sort }
bubble_sort()
  inteiro: i
  i := 0
  repita
    inteiro: j
    j := 0
    repita
      se vet[i] > v[j] então
        inteiro: temp
        temp := vet[i]
        vet[i] := vet[j]
        vet[j] := temp
      fim
      j := j + 1
    até j < i
    i := i + 1
  até i < tam
fim
```

```
{ programa principal }
inteiro principal()
  preencheVetor()
  bubble_sort()
  retorna(0)
fim
```

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

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

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 caracterizaram 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:
  - Analisador sintático preditivo: Tenta prever a construção seguinte na cadeia de entrada com base nas marcas de verificação à frente.
  - Analisador 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 redutiva ou seja a derivação ocorre ate atingir o simbolo não-terminal inicial.Dentro desta categoria ha diferentes técnicas de derivação como por exempo (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 modulo Yet Another Compiler Compiler (YACC), este modulo implementa um componente de analise.

Antes da declaração das regras gramaticais, foi elaborado 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.

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

Assim que declarado a estrutura de dado, foi declarado a gramatica 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.

```
def p_programa(self, p):
        '''programa : lista_declaracoes'''
        p[0] = Tree('programa', [p[1]])

def p_lista_declaracoes(self, p):
        '''lista_declaracoes : lista_declaracoes declaracao
                             | declaracao'''
        if len(p) == 3:
            p[0] = Tree('lista_declaracoes', [p[1], p[2]])
        else:
            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.

```
#Defini ão das precedencias
self.precedence = (
    ('left', 'COMPARACAO', 'MAIOR_IGUAL', 'MAIOR', 'MENOR_IGUAL'
        , 'MENOR'),
    ('left', 'MAIS', 'MENOS'),
    ('left', 'MULT', 'DIVIDE'),
)
```

Para o tratamento dos erros durante a analise 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.

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

```
        print("Erro Sintatico: '%s' na linha '%d'" %(p.value, p.
            lineno))
    else:
        print("Erro Fatal!@")

def p_leia_error(self, p):
    '''leia : LEIA ABRE_PAREN error FECHA_PAREN'''
    print("N O FOI POSSIVEL REALIZAR A LEITURA (ARGUMENTO
        INVALIDO)")
    exit(1)

def p_escreva_error(self, p):
    '''escreva : ESCREVA ABRE_PAREN error FECHA_PAREN '''
    print("N O FOI POSSIVEL REALIZAR A ESCRITA (ARGUMENTO
        INVALIDO)")
    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.

```
from graphviz import Digraph

def print_tree(node, dot, i="0", pai=None):
    if node != None:
        filho = str(node) + str(i)
        dot.node(filho, str(node))
        if pai: dot.edge(pai, filho)
        j = "0"
        if not isinstance(node, Tree): return
        for son in node.child:
            j+="1"
            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 analise 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.

```
inteiro: n

inteiro fatorial(inteiro: n)
    inteiro: fat
    se n > 0 então {não calcula se n > 0}
        fat := 1
        repita
            fat := fat * n
            n := n - 1
        até n = 0
        retorna(fat) {retorna o valor do fatorial de n}
    senão
        retorna(0)
    fim
fim

inteiro principal()
    leia(n)
    escreva(fatorial(n))
    retorna(0)
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 modulo da analise sintático 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 nos da árvore, sendo assim disponibilizo um link de um arquivo PDF com maior resolução para eventuais analises. `https://mega.nz/#!URcGVaCI!NWOS7ltDwoFlXZRQMu4PiY7hVOVKldMn_2b7PnuHz30`

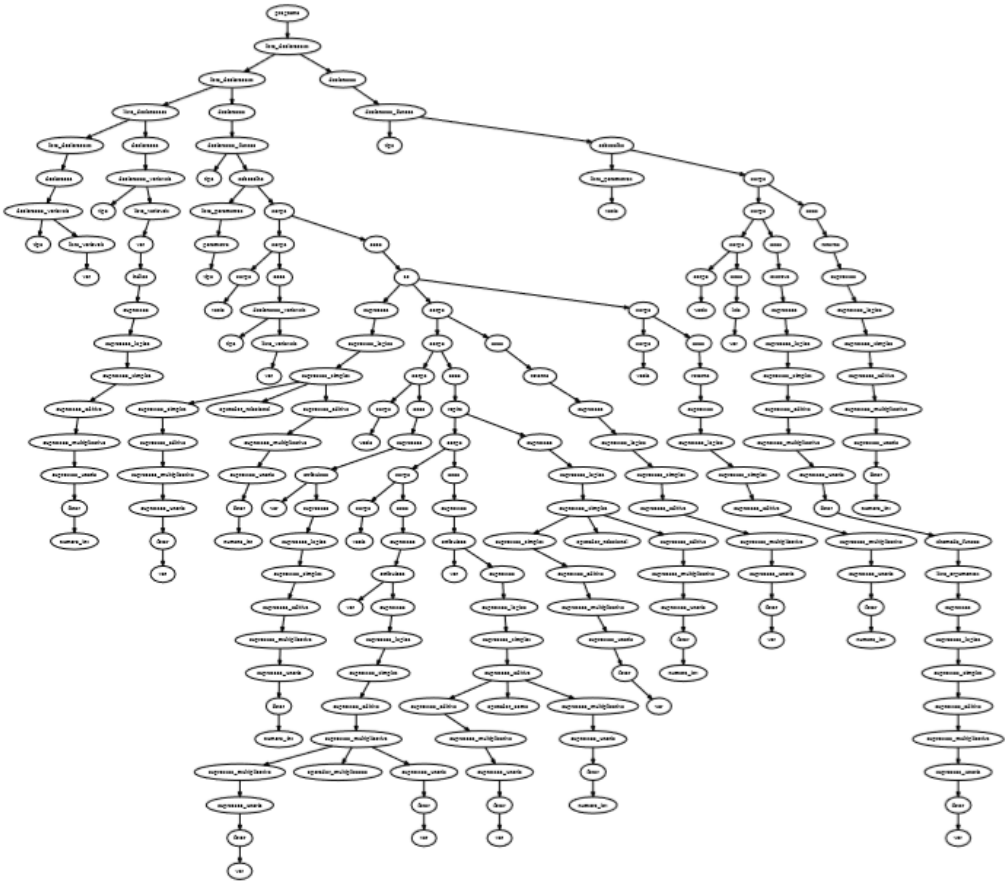

Figura 12. Árvore Sintática (SAÍDA Analisador Sintático)

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.

7.1. 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 uteis para o procedimento, por isso estes foram removidos da árvore se seus nós filhos foram rearranjados novamente a fim de manter a estrutura logica.

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 demostração contendo duas funções vazias.

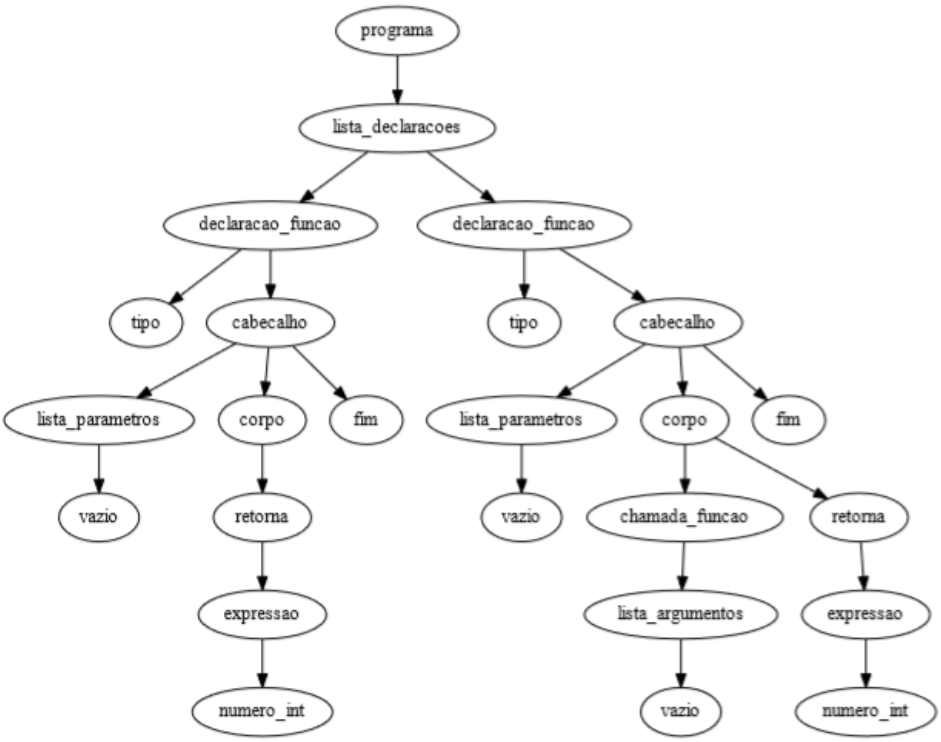

Figura 13. Exemplo simples de árvore Podada

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.

```
class Elemento():
    def __init__(self, nome, escopo, tipo, indice, used,
       ehParametro, linha=None):
        self.nome = nome
        self.escopo = escopo
        self.tipo = tipo
        self.used = used
        self.indice = indice
        self.ehParametro = ehParametro
        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.

```
class ListaFuncoes():
    def __init__(self, nome, tipo_retorno, lista_paramentros,
       foi_retornada, escopo, used, escopo_pai, linha=None):
        self.nome = nome
        self.tipo_retorno = tipo_retorno
        self.lista_paramentros = lista_paramentros
        self.escopo = escopo
        self.escopo_pai = escopo_pai
        self.foi_retornada = foi_retornada
        self.used = used
        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. Está 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.

```
class TabelaSimbolos():
    TemPrincipal = False
    def __init__(self):
        self.lista_elementos = []
        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 inciada recebe o no raiz da arvore 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 no correspondente é passado como parâmetro para a função que ira tratar as minucias, 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 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:
                       Variável '" + str(elemento.nome) +"'
                       declarada e não utilizada")
6              elif elemento.used == True and elemento.inicializado
                   == False:
7                  print ("Linha:["+str(elemento.linha)+"] Aviso:
                       Variável '" + str(elemento.nome) +"' não
                       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 contem alguns erros semânticos. A figura 14 mostra oque a saída do analisador semântico quando posto em analise 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 analise. 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 analise, isso é ótimo pois indica que todos os escopos foram empilhados e devidamente desempilhados assim que encontrado sua devida tag de fechamento.

8. Geração de Código

A etapa de geração de código pode ser considerada a última etapa do processo de compilação, também pode ser associado a essa etapa procedimentos de otimização de código para diminuir o número de instruções que o processador precisa trabalhar deixando um código mais limpo e eficiente.

O processo de geração de código em linhas gerais trabalha traduzindo um código em uma linguagem de alto nível, como C, C++, T++ para a linguagem de máquina de baixo nível mais conhecida como assembly. Porém existe uma limitação em desenvolver um compilador para traduzir da linguagem de alto nível para a linguagem de máquina, isso pode ser explicado pois nem todas as arquiteturas dos processadores implementam todas as instruções assembly da mesma forma tornando o compilador limitado a uma certa arquitetura pré definida.

O problema de produzir compiladores limitados a arquiteturas específica pode ser resolvido facilmente com a geração de um código intermediário muito semelhante ao assembly porém genérico. Neste trabalho foi utilizado a ferramenta LLVMLite que pode ser definida e explicada como uma biblioteca para criar programaticamente código nativo de máquina. É usada para gerar instruções em um formato chamado de representação intermediária ou IR que pode ser convertido e vinculado a um código de linguagem de montagem dependente da máquina para uma plataforma de destino. [Yegulalp 2018]

8.1. Detalhes da implementação

Toda a implementação da geração de código está implementada na classe “Gerador_TOP”. Esta classe recebe como entrada o nó raiz da árvore podada, provinda da análise semântica, é possível verificar um exemplo de como é a entrada através da Figura 15

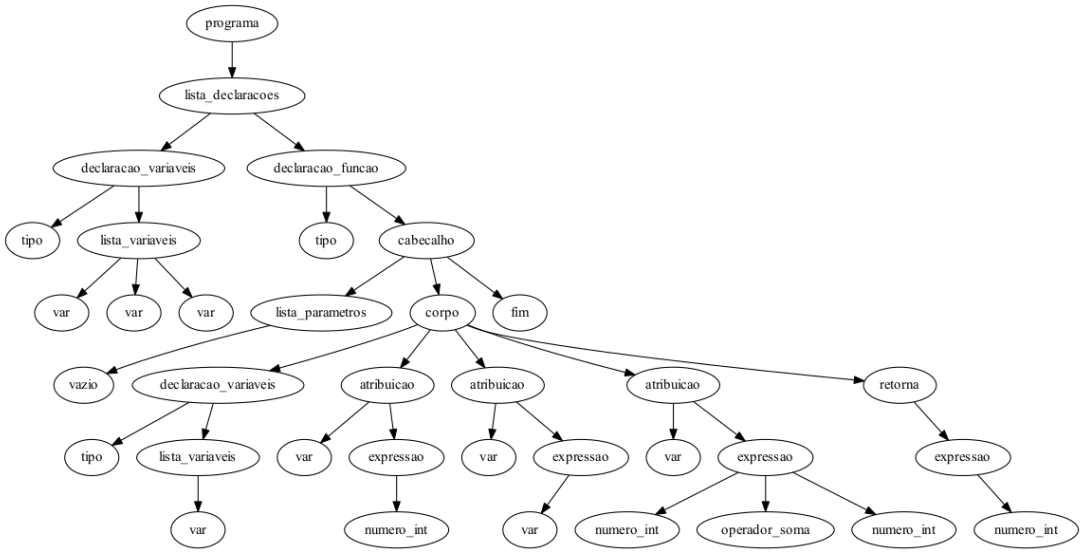

Figura 15. Exemplo de Entrada Geração de Código

A classe Gerador_TOP possui alguns métodos que julgo como essenciais para realizar as verificações em nós e tratar expressões. Serão apresentados abaixo os principais métodos para a geração de código e seu funcionamento.

O método recursivo ”andar”é invocado na função main do gerador, recebe como entrada o nó raiz da arvore podada e um modulo a cara iteração a função recebe uma nova raiz. O modulo declarado através de instruções do LLVMLite desta forma: ”modulo = ir.Module(NomeProg)”onde nome do programa é o nome que vem por argumento na execução do gerador.

O método abaixo realiza declaração de variáveis globais, e declaração de funções que podem ser a ”principal”ou não. Quando identificado um no de tipo declaração de variável ou declaração de função, ha uma chamada para um outro método que ira tratar cada caso de forma mais detalhada. Estes métodos específicos utilizam a biblioteca do LLVMLite para realizar alocação de variáveis, criação de funções e etc.

```
1  def andar(self, raiz, modulo):
2      if raiz:
3          for filho in raiz.child:
4              if filho.type == "declaracao_variaveis":
5                  self.llvm_declaracao_variavel_global(raiz, filho
                       , modulo)
6              if filho.type == "declaracao_funcao":
7                  self.declaracao_funcao(raiz, filho, modulo)
8              if not isinstance(filho, Tree): return
9              self.andar(filho, modulo)
10     else:
11         return
```

O código a baixo é um exemplo de como é utilizado a biblioteca do LLVMLite para realizar uma declaração de função. Primeiramente o método precisa saber qual é o tipo de retorno da função. Assim que descoberto este é especificado o tipo a função a ser criada através. Junto com a declaração de variável também é alocado um espaço para a variável de retorno, o tamanho desse espaço varia de a cordo com o tipo de função declarada

A fim de tornar a tarefa de geração de código mais flexível e organizada, a cada nova função ou variável criada, esta adicionada a uma lista de ponteiros onde a tarefa de buscar funções variáveis já alocada fica mais fácil e intuitiva.

```
def llvm_declaracao_funcao(self, modulo, filho, nome,
    tipo_de_retorno):
    if nome == "principal":
        nome = "main"
    if tipo_de_retorno == "inteiro":
        tipo_de_retorno = ir.IntType(32)
    elif tipo_de_retorno == "flutuante":
        tipo_de_retorno = ir.FloatType()

    tipo_da_funcao = ir.FunctionType(tipo_de_retorno, [])
    funcao = ir.Function(modulo, tipo_da_funcao, name=nome)
    self.lista_ponteiros_funcoes.append(funcao)

    bloco_de_entrada = funcao.append_basic_block('%s.start' %
        nome)

    self.builder = ir.IRBuilder(bloco_de_entrada)

    retorna = self.builder.alloca(tipo_de_retorno, name='return'
        )
    self.lista_ponteiros_variaveis.append(retorna)

    corpo = filho.child[1].child[1]
    self.resolve_corpo(corpo, modulo, self.builder)
```

Outro método importante para o processo de geração é o ”resolve_corpo”. É notório que o método ”andar”sozinho não trata as mi nuncias do código fonte, por isso foi criado o método abaixo. A cada função declarada ou escopo como por exemplo o escopo de laço e condicional possuem nós de tipo copro. Então quando é encontrado um nó deste tipo, este é passado para o método ”resolve_corpo”que ira identifica e tratar a existência de declaração de variáveis locais, laços de repetição, estruturas condicionais entre outras.

```
def resolve_corpo(self, raiz, modulo, builder):
    if raiz:
        for filho in raiz.child:
            if filho.type == "declaracao_variaveis":
                self.llvm_declaracao_variavel_local(filho,
                    builder)
            elif filho.type == "atribuicao":
                self.atribuicao(filho, modulo, builder)
            elif filho.type == "se":
                self.se(filho, modulo, builder)
            elif filho.type == "repita":
                self.repita(filho, modulo, builder)
            elif filho.type == "leia":
```

```
                self.leia_func(filho, modulo, builder)
            elif filho.type == "escreva":
                self.escreva(filho, modulo, builder)
            elif filho.type == "retorna":
                self.retorna(filho, modulo, builder)
            elif not isinstance(filho, Tree): return
            self.resolve_corpo(filho, modulo, builder)
    else:
        return
```

8.1.1. Exemplos de Entrada e Saída

Nesta sessão serão apresentados os resultados bem como as entradas e saídas submetidas submetidas ao gerador de código. O código fonte de entrada é esta na linguagem de alto nível chamada T++, o código de saída do gerador pode ser chamado de código intermediário (IR) e possui sua extensão ".ll"pois se trata de um arquivo gerado pela biblioteca do LLVMLite.

Para os testes foi utilizado o compilador Clang para compilar o código intermediário para a arquitetura do computador alvo. Isso pode ser feito através do dos comandos abaixo.

$ python3 gerador.py CODIGO-FONTE.tpp

$ clang CODIGO-FONTE.tpp.ll -o exe

$ ./exe

Código Fonte 01 Entrada

```
{Declara ão de variáveis}

inteiro: a

inteiro principal()
  inteiro: b

  a := 10

  b := a

    retorna(0)
fim
```

Código Fonte 01 Saída

```
; ModuleID = "gencode-001.tpp"
target triple = "unknown-unknown-unknown"
target datalayout = ""

@"a" = common global i32 0, align 4
define i32 @"main"()
{
main.start:
  %"return" = alloca i32
  %"b" = alloca i32, align 4
  store i32 10, i32* @"a"
  %"varTemp" = load i32, i32* @"a"
```

```
  store i32 %"varTemp", i32* %"b"
  %"retorna" = load i32, i32* %"return", align 4
  br label %"main.end"
main.end:
  store i32 0, i32* %"return"
  %"ret" = load i32, i32* %"return"
  ret i32 %"ret"
}
```

O código fonte 2 abaixo esta escrito na linguagem de programação de alto nível T++ realiza uma estrutura condicional, o código fonte de saída é um código intermediário que pode ser traduzido para a arquitetura do computador alvo.

Código Fonte 02 Entrada

```
{Condicional}
inteiro: a

inteiro principal()
  inteiro: ret

  a := 10
  se a > 5 então
        ret := 1
    senão
        ret := 0
    fim

    retorna(ret)
fim
```

Código Fonte 02 Saída

```
; ModuleID = "gencode-002.tpp"
target triple = "unknown-unknown-unknown"
target datalayout = ""

@"a" = common global i32 0, align 4
define i32 @"main"()
{
main.start:
  %"return" = alloca i32
  %"ret" = alloca i32, align 4
  store i32 10, i32* @"a"
  %"varTempLeft" = load i32, i32* @"a"
  %"if_0" = icmp sgt i32 %"varTempLeft", 5
  br i1 %"if_0", label %"iftrue_0", label %"iffalse_0"
iftrue_0:
  store i32 1, i32* %"ret"
  br label %"ifend_0"
iffalse_0:
  store i32 0, i32* %"ret"
  br label %"ifend_0"
ifend_0:
  store i32 1, i32* %"ret"
  store i32 0, i32* %"ret"
  %"retorna" = load i32, i32* %"return", align 4
  br label %"main.end"
main.end:
  %"varTemp" = load i32, i32* %"ret"
  store i32 %"varTemp", i32* %"return"
  %"ret.1" = load i32, i32* %"return"
```

```
30    ret i32 %"ret.1"
31  }
```

Referências

(2016). Bnf comentada. https://docs.google.com/document/d/1oYX-5ipzL_izj_hO8s7axuo2OyA279YEhnAItgXzXAQ/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).

Yegulalp, S. (2018). What is llvm? the power behind swift, rust, clang, and more.