

Compilador T++

Thiago Alexandre Nakao França

1

Abstract. *The goal of this paper is to detail the development process of a T++ compiler; the processes was split into 4 main phases: lexical, syntactic and semantic analyzers, followed by the generation of intermediate code, at this moment the lexic, syntatic and semantic analyzers are complete, we used the PLY framework to identify the tokens from the input programs and to build the syntactic tree.*

Resumo. *O objetivo deste artigo é o detalhamento do processo de desenvolvimento de um compilador na linguagem T++, tal processo se dá em quatro fases distintas, sendo elas a análise léxica, análise sintática, análise semântica e geração de código intermediário, no dado momento somente a análise léxica, sintática e semântica foram implementadas, o processo de tokenização e construção da árvore sintática foram feitos com o auxílio da biblioteca PLY.*

1. Introdução

Compiladores é uma disciplina clássica na área da computação, é nela onde a maioria dos acadêmicos passa a entender como as instruções de uma linguagem de alto nível são convertidas para um código intermediário(semelhante ao assembly), compreender o funcionamento desse processo ainda pode ser útil para o desenvolvimento de linguagens interpretadas, que desfrutam de um processo deveras semelhante. Neste artigo, desenvolvemos e documentamos um compilador para a linguagem fictícia T++ em python, para tal o processo de leitura do arquivo de entrada até a geração de código intermediário se dá em etapas clássicas, que são:

- Analise léxica.
- Analise sintática.
- Analise semântica.
- Geração de código intermediário.

2. A linguagem T++

A linguagem escolhida é uma linguagem de programação em alto nível, suas palavras reservadas são em português com direito a acentuação, possui tipos numéricos, operadores lógicos, operadores matemáticos, declaração de funções e variáveis, chamada de funções, um operador condicional e um operador de loop, por ser uma linguagem imperativa a forma de raciocínio para se programar nela se assemelha a linguagens tradicionais como C, Java e Python, todas essas especificações encontram-se nesta seção.

2.1. Tipos de variáveis

A linguagem tem dois tipos de variáveis, sendo eles inteiro e flutuante.

2.2. Operadores

Existem os operadores lógicos AND e OR, representados da mesma forma que na linguagem C por && e ||, respectivamente, além disso, a linguagem conta com operadores matemáticos de soma, subtração, multiplicação e divisão, sendo +, -, * e /, respectivamente. É importante mencionar a existência dos operadores de comparação, usados para verificar igualdade, desigualdade e maioridade, sendo eles =, <>, >, <, >= e <=.

Expressões unárias também são válidas.

2.3. Declarações e atribuições de variáveis

As variáveis obrigatoriamente precisam ter o tipo a elas anexado em sua declaração, passado esse momento, dentro do escopo em questão elas podem ser invocadas sem a necessidade do seu tipo, por exemplo:

- inteiro: a,b
- a:=2
- b:=3

Para a separação da identificação do tipo de uma variável e a sua label, utiliza-se o caracter ':', além disso, mais de uma variável pode ser declarada na mesma linha, contanto que possuam o mesmo tipo. A atribuição de variáveis se dá pelo operador ':='.

2.4. Declarações de funções

Na linguagem t++ é possível a criação de funções, toda a função precisa explicitar qual o tipo de saída esperada e quais os parâmetros de entrada, segue um exemplo:

- inteiro principal(inteiro:n,flutuante:p)
- código
- fim

Observe que se faz necessária a declaração do tipo da função, bem como a declaração do tipo de todos os parâmetros presentes na lista de entrada, a vírgula é usada como separador entre parâmetros, e a palavra reservada fim para denotar o final do escopo da função.

2.5. Laço de repetição e condicionais

A linguagem fornece uma forma de aplicação de estruturas de condicional, na forma de SE condição ENTÃO ação SENÃO ação FIM, com o bloco senão sendo opcional.

Além disso, é possível a utilização de laços de repetição, sendo que utiliza as palavras reservadas REPITA e ATE na forma REPITA ação ATE condição.

2.6. Leitura e escrita

Na linguagem é possível realizar a leitura de variáveis passadas pelo usuário com o comando LEIA(expressão), e pode-se escrever no console padrão com o comando ESCREVA(expressão).

2.7. Comentários

Os comentários são todos os caracteres contidos entre os caracteres { e }, nessa ordem específica.

3. Análise léxica

O objetivo desta etapa é ler o programa de entrada, e agrupar os conjuntos de caracteres de interesse de uma forma clara e sintetizada, formando as entidades que chamamos de tokens.

3.1. Tokens

Um token nada mais é que uma estrutura de dados que armazena duas informações, um identificador de tipo e outro de valor, visualmente podem ser representados por <tipo, valor>, usaremos essa estrutura para identificar blocos de interesse no código de entrada.

Por exemplo, seria interessante se agrupássemos todas as palavras reservadas da linguagens contidas em um código na forma

<PalavraReservada, RETORNA>

<PalavraReservada, SE>

<PalavraReservada, SENA0>

E separar os identificadores de entidades em

<identificador, var1> e <identificador, var2>

Montando essas estrutura podemos separar quais palavras e/ou simbolos nos interessam do texto de entrada agrupados pela forma que nos for mais conveniente, isso facilita e muito o trabalho nas próximas etapas.

Então, na realidade o objetivo final da análise léxica é transformar o código de entrada em uma lista de tokens, e a partir desse momento o arquivo de entrada não será mais utilizado no decorrer do processo, dado que estamos lidando com a linguagem T++, precisaremos de agrupamentos para identificadores, palavras reservadas, símbolos, comentários, números inteiros e reais.

3.2. Planejamento

Agora que sabemos que devemos gerar uma lista de tokens, a próxima questão é como faremos isso, na literatura existem duas abordagens clássicas adequadas para implementar o tokenizador, sendo pela utilização de automatos ou fazendo uso de expressões regulares. Neste trabalho optamos pelo uso de expressões regulares, mas iremos exibir os autômatos de cada agrupamento de token de interesse de forma visual.

Primeiramente, daremos nomes as categorias de tokens que utilizaremos e o que elas podem armazenar, segue:

- OID = Identificador de variáveis e demais objetos
- RWO = Palavras reservadas(repita,ate,se,senao,entao,fim,inteiro,real,leia,escreva,retorna)
- SYB = Símbolos(, + - * / := < > <= >= = () && || } :)
- INT = Numero inteiro
- FLT = Numero flutuante
- CMT = Comentários(Texto dentro dos caracteres { }, o token { estará incluso, mas o de fechamento excluido do valor dos tokens de match)

3.3. Materiais e métodos

Para a implementação do tokenizador, foi utilizado o framework Ply(python3.4), pois o método de implementação das expressões regulares é prático e de fácil entendimento utilizando essa ferramenta, para a utilizar basta realizar a instalação e depois importar a biblioteca ply.lex para o seu código python.

No ply.lex, a ideia é que para cada categoria de token seja definida uma expressão regular que será responsável por identificar os valores a ela pertencidos no texto de entrada, todos os grupos de tokens são definidos em um array que contém seus nomes.

Para cada nome de token, deve haver uma variável com o nome `t_{nomeGrupo}`, um exemplo no nosso cenário seria `t_SYB`, essa variável deve receber uma string indicando qual a expressão regular a ser utilizada para encontrar todos os tokens do grupo em questão.

```
tokens = [ 'CMT' , 'SYB' , 'FLT' , 'INT' , 'OID' ]+ list ( reservedList . values ( ) )
```

No exemplo acima, vemos que além da lista, existe uma referência a uma estrutura chamada `reservedList`, se trata de uma estrutura de dados que contém todas as palavras reservadas da linguagem T++, como segue:

```
reservedList = {  
    'se'          : 'RWO' ,  
    'SE'          : 'RWO' ,  
    'entao'       : 'RWO' ,  
    'ENTAO'       : 'RWO' ,  
    'senao'       : 'RWO' ,  
    'SENAO'       : 'RWO' ,  
    'fim'         : 'RWO' ,  
    'FIM'         : 'RWO' ,  
    'repita'      : 'RWO' ,  
    'REPITA'      : 'RWO' ,  
    'ate'         : 'RWO' ,  
    'ATE'         : 'RWO' ,  
    'inteiro'     : 'RWO' ,  
    'INTEIRO'     : 'RWO' ,  
    'flutuante'   : 'RWO' ,  
    'FLUTUANTE'   : 'RWO' ,  
    'leia'        : 'RWO' ,  
    'LEIA'        : 'RWO' ,  
    'escreva'     : 'RWO' ,  
    'ESCREVA'     : 'RWO' ,  
    'retorna'     : 'RWO' ,  
    'RETORNA'     : 'RWO' ,  
}
```

Isso porque é complicado separar essas palavras por uma expressão regular comum, para tal tarefa utilizamos uma função 'coringa', é muito fácil dos tokens da classe OID se confundirem com os de palavra reservada, por isso ao invés de criarmos uma expressão para OID e outra para RWO, iremos codificar ambas em na função do agrupamento OID.

Ah sim, eu esqueci de mencionar mais cedo mas além de strings, os tokens podem ser representados por funções que contenham uma expressão regular, bastando com que o nome da função seja igual a `t_nomeTokenCategoria`.

De volta para a função coringa, usaremos a `t_OID` para identificar os tokens de palavras reservadas e os identificadores.

```
def t_OID(t):
    r'_|([a-zA-Z])\w*_\w*'
    t.type = reservedList.get(t.value, 'OID')
    return t
```

Perceba que existe uma expressão regular dentro dessa função, o que ocorre é que essa expressão será utilizada para identificar todos os identificadores de variável (pois obrigatoriamente devem começar com letra ou underscore, depois podem conter qualquer sequência de letras e/ou números e underscores), no entanto, se o texto do token for IDENTICO a algum texto presente na reserved List, ele terá o valor respectivo de tipo da reservedList, que no nosso caso sempre será RWO.

Mas agora que você já viu do pior, vamos voltar para as declarações das expressões regulares simples, as que usam variáveis com para representar os grupos de tokens.

```
t_CMT = r'{[^\]}*'
t_SYB = r'\+|-|=|>|<|<>|:|\\*|<|>|}|=|\\(|\\)|,|\\[|\\]|&&|\\\\|'
t_FLT = r'\\d+\\.\\d+'
t_INT = r'\\d+'
t_ignore = ' \\t\\r\\n'
```

Aqui temos variáveis com os nomes das categorias dos nossos tokens que restaram, dentro de cada `r''` está a expressão regular de cada token group.

A `t_CMT` contém uma expressão que ao encontrar um abre chaves, só para de casar quando encontrar o fecha chaves, por isso conseguimos pegar todos os comentários com o abre chaves, mas token fecha chaves vai ser da categoria dos símbolos.

A `t_SYB` é mediocrementemente simples, se trata apenas de um ou entre os possíveis símbolos da linguagem, nada mais. Além de que `t_FLT` e `t_INT` também são deveras modestas, utilizando dígito.dígito e somente dígito para casar.

Repare que existe o `t_ignore`, que não é uma categoria de token, como o nome sugere, são caracteres que devem ser ignorados pelo lexer.

Por fim, basta instanciar o lexer (entidade que leva em conta as definições acima descritas), informar a string de entrada, e chamar a função `token()` até que não haja mais um token a ser lido. No nosso caso, estaremos escrevendo os tokens no arquivo `output.txt`.

```
lexer = lex.lex()
f = open('input.txt', 'r')
input = f.read()
f.close()
lexer.input(input)

file = open('output.txt', 'w');
```

```

while True:
    tok = lexer.token()
    if not tok:
        break      # No more input
    file.write("<" + tok.type + "," + tok.value + ">\n")
file.close()

```

E assim temos a essência do tokenizador, que mais adiante será testado, também abordaremos os autômatos referentes a cada expressão regular aqui exposta mais adiante, o software usado para desenhar os autômatos foi o JFLAP.

3.4. Testes

Dado o seguinte código:

```

inteiro: A[20]
inteiro busca(inteiro: n)
    inteiro: retorno
    inteiro: i
    retorno := 0{Comentario
formoso}
    i := 0
    repita
        se A[i] = n
            retorno := 1
        fim
        i := i + 1
    at i = 20
    retorna(retorno)
fim

```

Após rodar o tokenizador, obtivemos a saída:

```

<RWO, inteiro> <SYB,:> <OID,A> <SYB,[> <INT,20>
<SYB,]> <RWO, inteiro> <OID, busca> <SYB,(> <RWO, inteiro>
<SYB,:> <OID,n> <SYB,)> <RWO, inteiro> <SYB,:> <OID, retorno>
<RWO, inteiro> <SYB,:> <OID,i> <OID, retorno> <SYB,:=> <INT,0>
<CMT,{ Comentario
formoso>
<SYB,}>
<OID,i> <SYB,:=> <INT,0> <RWO, repita> <RWO, se> <OID,A> <SYB,[>
<OID,i> <SYB,]> <SYB,=> <OID,n> <OID, retorno> <SYB,:=> <INT,1>
<RWO, fim> <OID,i> <SYB,:=> <OID,i> <SYB,+> <INT,1> <RWO, at >
<OID,i> <SYB,=> <INT,20> <RWO, retorna> <SYB,(> <OID, retorno>
<SYB,)> <RWO, fim> <RWO, inteiro> <OID, principal><SYB,(>
<SYB,)> <RWO, inteiro> <SYB,:> <OID,i> <OID,i> <SYB,:=> <INT,0>
<RWO, repita> <OID,A> <SYB,[> <OID,i> <SYB,]> <SYB,:=> <OID,i>
<OID,i> <SYB,:=> <OID,i> <SYB,+> <INT,1> <RWO, at > <OID,i>
<SYB,=> <INT,20> <RWO, leia> <SYB,(> <OID,n> <SYB,)> <RWO, escreva>

```

<SYB,(> <OID, busca > <SYB,(> <OID, n > <SYB,) > <SYB,) > <OID, retorno >
 <SYB,(> <INT, 0 > <SYB,) > <RWO, fim >

Acompanhando token a token, fica visível que obtivemos o resultado esperado, o comentário mesmo foi identificado ainda que houvesse em mais que uma linha.

3.5. Autômatos das expressões regulares

Abaixo, seguem os autômatos das expressões regulares mencionadas neste documento, para compactação de espaço visual, utilizamos colchetes para sinalizar a presença do operador lógico OR(). Acima estão os autômatos das expressões OID, RWO, CMT, SYB

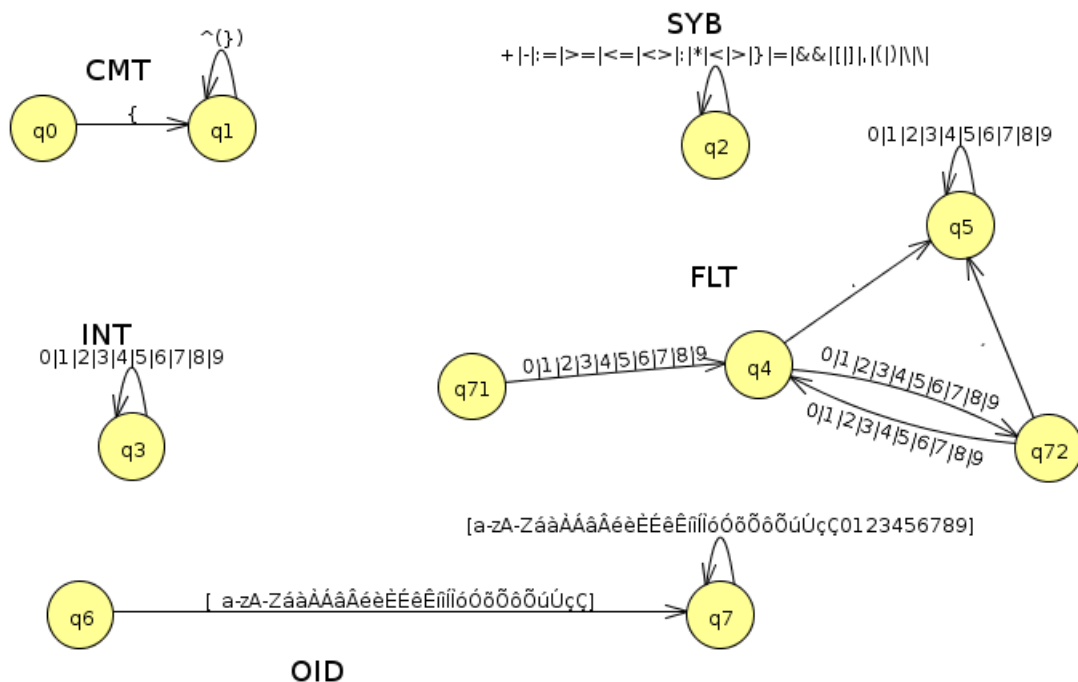


Figura 1. Autômatos das expressões OID, RWO, CMT, SYB, INT, FLT

e INT, todos finitos e determinísticos.

Seguindo, temos os autômatos das palavras reservadas. A aqui temos os autômatos finitos determinísticos das palavras reservadas do lexer.

4. Análise sintática

O objetivo dessa etapa é o processamento dos tokens da análise léxica, que resulta em uma validação deles e em uma árvore sintática abstrata.

4.1. Gramáticas livres de contexto e a BNF do T++

Nesse momento, nos deparamos em uma situação onde não é mais possível a utilização de somente expressões regulares, isso porquê não seria possível o controle de sequenciamento e recursão somente dessa maneira, por essa razão foi preciso a utilização de

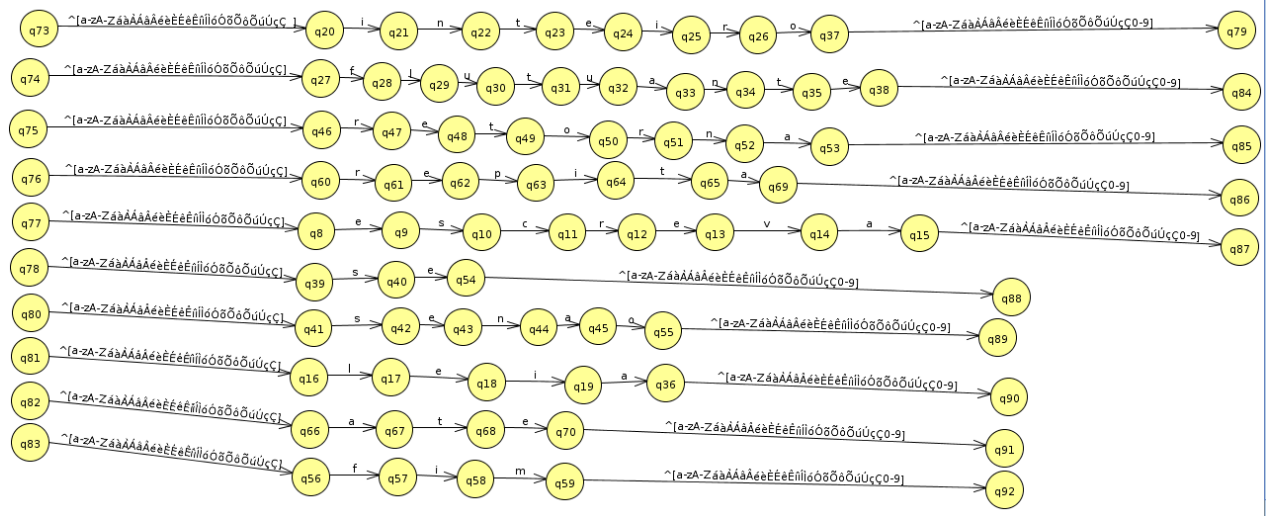


Figura 2. Autômatos das palavras reservadas

gramáticas livres de contexto para a construção da árvore sintática. A gramática utilizada está presente logo abaixo.

`programa : lista_declaracoes`

`lista_declaracoes : lista_declaracoes declaracao
| declaracao`

`declaracao : declaracao_variaveis
| inicializacao_variaveis
| declaracao_funcao`

`declaracao_variaveis : tipo DOISPONTOS lista_variaveis`

`inicializacao_variaveis : atribuicao`

`lista_variaveis : lista_variaveis VIRGULA var
| var`

`var : ID
| ID indice`

`indice : indice LCOLCHETE expressao RCOLCHETE
| LCOLCHETE expressao RCOLCHETE`

`tipo : INTEIRO
| FLUTUANTE`

`declaracao_funcao : tipo cabecalho`


```

        | cabecalho

cabecalho : ID LPARENTHESESYS lista_parametros RPARENTHESESYS corpo FIM

lista_parametros : lista_parametros VIRGULA parametro
                  | parametro
                  | vazio

parametro : tipo DOISPONTOS ID
           | parametro LCOLCHETE RCOLCHETE

corpo : corpo acao
       | vazio

acao : expressao
      | declaracao_variaveis
      | se
      | repita
      | leia
      | escreva
      | retorna
      | error

se : SE expressao ENTAO corpo FIM
   | SE expressao ENTAO corpo SENAO corpo FIM

repita : REPITA corpo ATE expressao

atribuicao : var ATRIB expressao

leia : LEIA LPARENTHESESYS var RPARENTHESESYS

escreva : ESCREVA LPARENTHESESYS expressao RPARENTHESESYS

retorna : RETORNA LPARENTHESESYS expressao RPARENTHESESYS

expressao : expressao_logica
           | atribuicao

expressao_logica : expressao_simples
                  | expressao_logica operador_logico expressao_simples

expressao_simples : expressao_aditiva
                  | expressao_simples operador_relacional expressao_aditiva

expressao_aditiva : expressao_multiplicativa

```

```

| expressao_aditiva operador_soma expressao_multiplicativa

expressao_multiplicativa : expressao_unaria
| expressao_multiplicativa operador_multiplicacao expressao_unaria

expressao_unaria : fator
                  | operador_soma fator
                  | operador_negacao fator

operador_relacional : LESSER
                    | GREATER
                    | EQUAL
                    | DIFF
                    | LESSEREQ
                    | GREATEREQ

operador_soma : PLUS
              | MINUS

operador_negacao : NEGATION

operador_logico : AND
                | OR

operador_multiplicacao : MULT
                       | DIV

fator : LPARENTHESES expressao RPARENTHESES
      | var
      | chamada_funcao
      | numero

numero : INT
       | FLOAT

chamada_funcao : ID LPARENTHESES lista_argumentos RPARENTHESES

lista_argumentos : lista_argumentos VIRGULA expressao
                 | expressao
                 | vazio

vazio :

```

4.2. O formato da Análise sintática realizado pelo PLY

O PLY.yacc utiliza uma técnica de parsing conhecida como LR-parsing ou shift-reduce parsing. Se trata de uma técnica de baixo para cima que tenta reconhecer o "lado direito

da mão"de várias regras gramaticais. Toda vez que o lado direito de uma regra é encontrado na entrada, a ação apropriada é invocada e os símbolos da gramática são substituídos pelo símbolo gramatical do lado esquerdo da declaração.

O LR-parsing geralmente é implementado utilizando enviando símbolos gramaticais para uma pilha, adicionalmente olhando para a pilha e para o próximo token de entrada para ver se casou com alguma das regras gramaticais, a seção abaixo exemplifica como se dá o processo dado a gramática hipotética.

#Regras gramaticais – exemplo

Gramática	Ação
$\begin{aligned} \text{exp0} &: \text{exp1} + \text{term} \\ & \text{exp1} - \text{term} \\ & \text{term} \end{aligned}$	$\begin{aligned} \text{exp0.val} &= \text{exp1.val} + \text{term.val} \\ \text{exp0.val} &= \text{exp1.val} - \text{term.val} \\ \text{exp0.val} &= \text{term.val} \end{aligned}$
$\begin{aligned} \text{term0} &: \text{term1} * \text{factor} \\ & \text{term1} / \text{factor} \\ & \text{factor} \end{aligned}$	$\begin{aligned} \text{term0.val} &= \text{term1.val} * \text{factor.val} \\ \text{term0.val} &= \text{term1.val} / \text{factor.val} \\ \text{term0.val} &= \text{factor.val} \end{aligned}$
$\begin{aligned} \text{factor} &: \text{NUMBER} \\ & (\text{exp}) \end{aligned}$	$\begin{aligned} \text{factor.val} &= \text{int}(\text{NUMBER.lexval}) \\ \text{factor.val} &= \text{exp.val} \end{aligned}$

#Processamento do parsing para a entrada '3 + 5 * (10 - 20)'

#OBS: Sh = shift / Rd = Reduce

St Symbol Stack	Input Tokens	Action
1	3 + 5 * (10 - 20)\$	sh 3
2 3	+ 5 * (10 - 20)\$	
Rd factor : NUMBER		
3 factor	+ 5 * (10 - 20)\$	Rd term
: factor		
4 term	+ 5 * (10 - 20)\$	Rd exp : term
5 exp	+ 5 * (10 - 20)\$	sh +
6 exp +	5 * (10 - 20)\$	sh 5
7 exp + 5	* (10 - 20)\$	
Rd factor : NUMBER		
8 exp + factor	* (10 - 20)\$	Rd term
: factor		
9 exp + term	* (10 - 20)\$	sh *
10 exp + term *	(10 - 20)\$	sh (
11 exp + term * (10 - 20)\$	sh 10
12 exp + term * (10	- 20)\$	
Rd factor : NUMBER		
13 exp + term * (factor	- 20)\$	
Rd term : factor		

```

14 exp + term * ( term          - 20 )$      Rd exp : term
15 exp + term * ( exp          - 20 )$      sh -
16 exp + term * ( exp -        20 )$      sh 20
17 exp + term * ( exp - 20          )$
Rd factor : NUMBER
18 exp + term * ( exp - factor          )$
Rd term : factor
19 exp + term * ( exp - term          )$
Rd exp : exp - term
20 exp + term * ( exp          )$      sh )
21 exp + term * ( exp )          $
Rd factor : (exp)
22 exp + term * factor          $
Rd term : term * factor
23 exp + term          $
Rd exp : exp + term
24 exp          $      Rd exp
25          $      Success !

```

No parsing de uma expressão, um estado de máquina e o token corrente determinam qual será a próxima expressão, se o próximo token parecer com parte de uma regra gramatical válida, ele é geralmente enviado para a pilha, se o topo da pilha contém um valor válido de "lado direito" de uma regra gramatical, ele é reduzido e os símbolos são substituídos pelo símbolo do lado esquerdo, quando essa redução ocorre, a ação apropriada é disparada (se definida). Se o token de entrada não puder ir para a pilha e o topo não condiz com nenhuma das regras gramaticais, um erro sintático ocorreu e o parser deve partir para um passo de recuperação, o parse só é bem sucedido quando alcançar o estado onde a pilha de símbolos está vazia e não existem mais tokens de entrada.

O PLY também utiliza uma técnica chamada LALR (Look-ahead LR parser), se trata de uma versão simplificada do LR parser tradicional.

4.3. Implementação e o PLY.yacc

A primeira tarefa da análise sintática foi a escrita da gramática de acordo com as regras do PLY.yacc, cada gramática deve estar dentro de um bloco de comentários, e cada bloco deve estar dentro de uma função, cada função tem acesso ao objeto do PLY responsável pela construção da árvore, e em cada chamada deve-se decidir que "nó" será pendurado na árvore quando o caso de redução esperado ocorrer, como segue:

```

def p_operador_soma(self, p):
    """
        operador_soma : PLUS
                       | MINUS
    """
    p[0] = NkTree([p[1]], 'operador_soma')

```

Por enquanto, ignore a existência da classe NkTree, apenas leve em conta que no PLY.yacc, toda vez que ocorrer uma redução de acordo com o lado direito da regra, o que p[0] receber no corpo da função será o "nó" pendurado na árvore sintática.

A documentação do PLY instrui a utilização de uma estrutura de dados auxiliar para a montagem da árvore, nossa implementação a chamou de NkTree, se trata de uma árvore simples, o código da sua estrutura está logo abaixo:

```
class NkTree:

    def __init__(self, children, type, leaf=None):
        self.type = type
        self.children = children
        self.leaf = leaf
        self.label = ""

    def __str__(self):
        return self.type

    def getNextCount(self):
        self.count += 1
        return str(self.count)

    def setLabelId(self, value):
        self.label = self.type + str(value)
```

Toda vez que uma redução for ocorrer, umas das funções que contém as gramáticas será chamada, e sempre a saída irá ser uma NkTree, que recebe como parametros as próximas regras a serem executadas em cadeia e o nome da regra em questão.

Agora você deve estar se perguntando, mas que valores são armazenados no child de cada NkTree em cada chamada de função? A resposta é simples, são passados todas as entidades do lado direito das regras gramaticais como filhos de cada nó NkTree, para todas as regras definidas nas funções, seguem alguns exemplos.

```
def p_declaracao(self, p):
    """
        declaracao : declaracao_variaveis
                   | inicializacao_variaveis
                   | declaracao_funcao
    """
    p[0] = NkTree([p[1]], 'declaracao')

def p_declaracao_variaveis(self, p):
    """
        declaracao_variaveis : tipo DOISPONTOS lista_variaveis
    """
    p[0] = NkTree([p[1], p[2], p[3]], 'declaracao_variaveis')

def p_inicializacao_variaveis(self, p):
    """
        inicializacao_variaveis : atribuicao
    """
```

```

p[0] = NkTree([p[1]], 'inicializacao_variaveis ')

def p_lista_variaveis(self, p):
    ,,,
    lista_variaveis : lista_variaveis VIRGULA var
                    | var
    ,,,
    if len(p)==4:
        p[0] = NkTree([p[1],p[2],p[3]], 'lista_variaveis ')
    elif len(p)==2:
        p[0] = NkTree([p[1]], 'lista_variaveis ')

```

Note que o atributo children do NkTree é um simples array com os statements do lado direito das regras, e que algumas condicionais sobre atributos do objeto em questão foram usadas para definir quais parametros deveriam ser passados como filhos, dependendo da situação.

Outro detalhe importante é que a precedência entre os tokens do lexer teve que ser configurada, quanto mais próximo do topo(sequencialmente falando), menor a precedência, e elementos no mesmo grupo compartilham da mesma precedência. Como multiplicação e divisão tem prioridade sobre soma e subtração, eles estão em um conjunto posterior ao das somas.

```

self.precedence = (
('left', 'EQUAL', 'GREATEREQ', 'GREATER', 'LESSEREQ', 'LESSER'),
('left', 'PLUS', 'MINUS'),
('left', 'MULT', 'DIV')
)

```

4.4. A árvore sintática

Por fim, uma vez que todas as funções das regras gramaticais foram definidas, e as atribuições referentes as instâncias da classe NkTree foram atribuídas a cada nó da árvore sintática, obtivemos a árvore sintática no PLY, no entanto ainda se fazia necessária uma forma de representação gráfica da mesma, para tal optamos pela utilização da biblioteca treelib.

A biblioteca requer um objeto Tree() para funcionar, então tudo o que tivemos que fazer foi uma função que monta esse objeto como um espelho do NkTree populado pelo PLY, e depois chamar o método show da biblioteca, segue o código.

```

def mountTree(self, nktree):
    actuals = []
    nexts = nktree.children
    nktree.setLabelId(self.getNextCount())
    self.tree.create_node(nktree.type, nktree.label)
    actuals.append(nktree)

    while len(nexts)!=0:
        nexts = []

```

```

parentName=""
for node in actuals:
    if (type(node) is NkTree):
        for next in node.children:
            if (type(next) is NkTree):
                next.setLabelId(self.getNextCount())
                self.tree.create_node(next.type, next.label, node.label)
            else:
                if (next is None):
nexts.append(next)
                self.tree.create_node(next, count, node.label)
            nexts.append(next)
actuals = nexts

def print(self, **kwargs):
    self.mountTree(self.parsed)
    self.tree.show(**kwargs)

```

Segue um exemplo de impressão da Árvore Sintática abstrata gerada, invocando a função print acima descrita.

```

Input:
inteiro soma(inteiro: a, inteiro :b)
    retorna (a+b)
fim

```

A figura 3 mostra a árvore gerada para essa entrada trivial.

5. Análise semântica

O objetivo da análise semântica é a verificação de erros de contexto, um tipo de erro muito comum que é pego nessa fase é a atribuição de um tipo inadequado para uma variável, ou ainda um retorno de tipo inesperado de uma função. Na análise semântica navega-se entre a árvore gerada na análise sintática, o objetivo é criar uma estrutura de dados que facilite o processo de identificação de possíveis erros, essa estrutura é chamada de tabela de símbolos. Uma vez que a tabela de símbolos é carregada, o programa deve mais uma vez percorrer a árvore sintática, para identificar declarações, atribuições ou invocações que estejam com algum problema lógico, pois sabendo-se o escopo de todas as funções e variáveis declaradas, é possível identificar cenários onde problemas de tipo/contexto relacionados a variáveis ocorrem. O analisador semântico da linguagem T++ identifica erros e emite warnings em certos eventos, sendo eles:

- A quantidade de parâmetros reais de uma chamada de procedimento deve ser igual a quantidade de parâmetros formais da sua definição.
- Warnings deverão ser mostrados quando uma variável for declarada mais de uma vez.
- Warnings deverão ser mostrados quando uma variável for declarada mas nunca utilizada.
- Warnings deverão ser mostrados quando ocorrer uma coerção implícita de tipos.

- Uma variável não declarada.
- Uma variável não inicializada.
- Verificar a existência de uma função principal que inicializa a execução do código.
- Verificar a chamada recursiva da função principal.
- Todas as regras acima descritas devem se aplicar para os escopos global, função e de blocos SE e REPITA adequadamente.

5.1. Estratégia utilizada para a geração da tabela de símbolos

Sabendo que a forma mais fácil de realizar a análise semântica é pela criação de uma tabela de símbolos, notei que uma abordagem interessante seria a utilização de um dicionário, no qual, cada chave é representado por um contexto do programa, no código a seguir é apresentado um programa simples, nele existem três contextos, sendo o global, plus e principal, isso porquê cada função possui o seu próprio contexto, e os contextos internos tem acesso as variáveis dos externos sem a necessidade de passagem por parâmetro.

```
a := 5
```

```
interno plus( inteiro :a)
  retorna a+1
```

```
inteiro principal()
  flutuante : p
  p := 12
  escreva(p)
  se p > 15 ent o
    inteiro :k
    k := 4
  fim
  retorna 0
fim
```

No exemplo acima ilustrado, a estrutura de dados ideal seria:

```
{
  "global": [a, plus, principal],
  "plus": [a],
  "principal": [p]
  "principal.sel": [k]
}
```

A essência da estrutura é essa, cada contexto é chave de uma lista de símbolos, que podem ser declarações de variáveis ou funções, neste projeto cada símbolo é uma classe que possui informações do nome, tipo, quantidade de parâmetros, quantidade de dimensões em x e em y, isso porquê a ideia é utilizar uma mesma estrutura de dados para armazenar os símbolos de variáveis e funções, sendo que o fator diferenciador será o preenchimento de campos específicos. É importante notar que para escopos internos dos blocos SE e REPITA nas funções, o nome do escopo é composto pelo nome da função de origem,

acrescentado por um caracter '.' e então a palavra se ou rp, contendo um número a sua frente indicando a sequencia da estrutura na função(Como ilustrado para o bloco SE na função principal, o resultado é principal.sel:[listaVariaveis]), se for ventura dentro de um bloco interno hajam outros blocos, o nome do escopo irá incrementalmente crescer seguindo a mesma regra.

Então, para lidar com os contextos, utilizaremos duas estruturas de dados, uma classe chamada SymbolsController, que contém um mapa onde cada chave é um escopo, e o valor é uma lista da símbolos.

A classe símbolo pode ser usada para representar uma variável unidimensional, bidimensional e funções, diferenciando as instâncias pelos atributos utilizados.

```
class SymbolsController:
```

```
    def __init__(self):
        self.scopeHash = {}
        self.declarationErrors = []
```

```
class Symbol:
```

```
    def __init__(self, name, type, isArray, qtLines, qtColumns, scope, params):
        self.name=name
        self.type=type #se inteiro, flutuante
        self.isArray=isArray
        self.dX=qtLines
        self.dY=qtColumns
        self.scope=scope
        self.params=params
        if(self.params is None):
            self.isFunction=0
        else:
            self.isFunction=1
```

Uma vez que a estrutura foi definida, o próximo passo foi pensar em como percorrer a árvore, o problema é que o produto da análise sintática é uma árvore que contém muitos galhos redundantes, isso porquê para a representação de listas de parametros, listas de variáveis, cada novo elemento foi adicionado de uma forma ruim para se percorrer, como segue.

Como pode ser visto na figura 4, uma declaração de variáveis com três variáveis fez uso de vários nós do tipo listaVariaveis, por razões de simplificação, todos os galhos que possuíssem esse mesmo padrão foram simplificados, a ponto que esses nós intermediários entre os parametros fossem removidos, a figura 5 demonstra o resultado de um processo de redução. Esse processo foi realizado para diversos nós, sendo eles parametro, operadorRelacional, operadorLogico, operadorMultiplicacao, operadorSoma, tipo, listaArgumentos, expressaoUnaria, expressaoMultiplicativa, expressaoAditiva, expressaoSimples, expressaoLogica, indice, corpo, listaVariaveis, listaParametros e listaDeclaracoes.

Com a árvore reduzida, o próximo passo foi realizar a criação da tabela de símbolos, para tal, percorreu-se a árvore sintática a partir de sua raiz, tomando por escopo inicial o global, toda vez que um nó de `declaracaoVariavel` ou `declaracaoFuncao` era encontrado, esse registro era registrado na lista de símbolos do escopo corrente(que inicialmente é o global), o escopo só mudar ao expandir o nó de uma declaração de função, nele o escopo é alterado para o nome da função, e as declarações de variáveis são acrescentadas na lista de símbolos do escopo em questão. Abaixo segue o esqueleto da função responsável por essa funcionalidade, para mais detalhes olhar o código fonte.

```
def mountSymbolsTables(self, sourceNode):
    if sourceNode is None:
        return -1
    scope = "global"
    types = ["declaracao", "lista_declaracoes"]
    functionNodes = []
    actuals = []
    nexts = sourceNode.children
    actuals.append(sourceNode)

    #O código abaixo percorre os nós da
    #árvore sintática em busca de
    #declarações de funções e
    #variáveis no escopo global.
    while len(nexts) != 0:
        nexts = []
        for node in actuals:
            for next in node.children:
                if next.type == "declaracao_variaveis":
                    #Se achou uma declaração de variáveis,
                    #adicione a tabela de símbolos no escopo
                    #global
                    symbols = self.getVarList(next, "global")
                    for symbol in symbols:
                        self.put("global", symbol)
                elif next.type == "declaracao_funcao":
                    #Mesma coisa pra função
                    tmpTipo = ""
                    if len(next.children) == 2:
                        tmpTipo = next.children[0].type
                        cabecalho = next.children[1]
                    else:
                        tmpTipo = None
                        cabecalho = next.children[0]
                    funcName = cabecalho.children[0]
                    params = self.getParamsSymbolsList(
                        cabecalho.children[2], funcName)
                    funcSymbol = Symbol
```

```

        (funcName, tmpTipo, 0, 0, 0, scope, params)
        self.put("global", funcSymbol)
        functionNodes.insert(0, next)
        functionNodes.insert(0, next)
    elif next.type in types:
        nexts.append(next)
actuals = nexts

#Agora que os nos de fucao foram encontrados,
#procura por declara es destas funcoes
for function in functionNodes:
    tmpTipo = ""
    if len(function.children)==2:
        tmpTipo=function.children[0].type
        cabecalho = function.children[1]
    else:
        tmpTipo=None
        cabecalho = function.children[0]
        funcName = cabecalho.children[0]
        corpo = cabecalho.children[4]
        types = ["declaracao", "lista_declaracoes", "acao"]
        functionNodes = []
        actuals = []
        nexts = corpo.children
        actuals.append(corpo)
        seCount=1 #Contador dos se's
        repitaCount=1 #Contador dos para
        while len(nexts)!=0:
            nexts=[]
            for node in actuals:
                for next in node.children:
                    if next.type=="declaracao_variaveis":
                        symbols = self.getVarList(next, funcName)
                        for symbol in symbols:
                            self.put(funcName, symbol)
                    elif next.type=="se":
                        next.sequenceId = seCount
                        self.mountSeTable
                        (next, funcName+".se"+str(seCount))
                        seCount+=1
                    elif next.type=="repita":
                        next.sequenceId = repitaCount
                        self.mountRepitaTable
                        (next, funcName+".rp"+str(repitaCount))
                        repitaCount+=1
                    elif next.type in types:

```

```

            nexts.append(next)
        actuals = nexts
    self.insertParamVariables()
    return ""

```

Agora você deve estar se perguntando, como que a tabela de símbolos dos escopos RE-PITA e SE é preenchida? Veja que para isso, existem duas funções, a mountRepitaTable e a mountSeTable, essas funções recebem como entrada no nó de seu respectivo tipo(repita ou se), e realiza o mesmo procedimento de cadastro de variável na tabela ao encontrar uma declaração de variáveis, veja o seu código abaixo.

```

def mountSeTable(self, seNode, scope):
    corpo1 = seNode.children[3]
    corpo2 = None
    expressionNode = seNode.children[1]
    corposToExplore = [corpo1]
    if len(seNode.children)>5:
        corpo2 = seNode.children[5]
        corposToExplore.append(corpo2)
    for corpo in corposToExplore:
        self.processCorpo(corpo, scope)

def mountRepitaTable(self, repitaNode, scope):
    expressionNode = repitaNode.children[3]
    corpo = repitaNode.children[1]
    self.processCorpo(corpo, scope)

def processCorpo(self, corpo, scope):
    types = ["declaracao", "lista_declaracoes", "acao"]
    functionNodes = []
    actuals = []
    nexts = corpo.children
    actuals.append(corpo)
    seCount=1 #Contador dos se's
    repitaCount=1 #Contador dos para
    while len(nexts)!=0:
        nexts=[]
        for node in actuals:
            for next in node.children:
                if next.type=="declaracao_variaveis":
                    symbols = self.getVarList(next, scope)
                    for symbol in symbols:
                        self.put(scope, symbol)
                elif next.type=="se":
                    next.sequenceId=seCount
                    self.mountSeTable(next, scope+".se"+str(seCount))
                    seCount+=1
                elif next.type=="repita":

```

```

        next.sequenceId=repitaCount
        self.mountRepitaTable
        (next, scope+".rp"+str(repitaCount))
        repitaCount+=1
    elif next.type in types:
        nexts.append(next)
actuals = nexts

```

Como é visível no código acima, as funções de montagem dos blocos SE e REPITA invocam a função `processCorpo`, que registra as variáveis em seus devidos escopos e marca os nós SE e REPITA na árvore sintática para que a busca de erros seja mais simplificada(`next.sequenceId=seCount`), quando a `processCorpo` encontre um escopo interno, ela faz uma chamada recursiva ao `mountRepitaTable` e `mountSeTable`, passando o escopo interno adequado como parâmetro.

Uma vez que o procedimento foi realizado, a tabela de símbolos é completa para todos os nós. A idéia central dessa etapa foi buscar pelas declarações de variáveis e as declarações de funções na árvore, para então os registrar no dicionário de símbolos.

Estratégias utilizadas para a realização da análise sensível ao contexto Uma vez que a "tabela/dicionário" foi elaborada, o próximo passo foi percorrer a árvore reduzida em busca dos erros semânticos, a estratégia foi a seguinte:

Analisando as regras da gramática, ficou claro que o primeiro nível de busca, do escopo global deveria navegar apenas nos nós `declaracao`, `listaDeclaracoes` e `declaracaoFuncao`, assumindo esses nós como "pais" e expandindo os filhos, que são os nós que nos interessam.

Especificamente estamos interessados nos nós de `inicializacaoVariaveis`, `declaracaoFuncao`, `expressaoLogica`, `retorna`, `leia`, `escreva`, `se e repita`, para o escopo global e para os escopos das funções internas, isso pois todos esses estão diretamente envolvidos com a utilização das variáveis previamente inseridas no dicionário de símbolos.

A parte mais difícil desse trabalho foi pensar em uma forma de descobrir se o tipo atribuído a uma variável era correto, isso porquê um a atribuição é uma expressão, que pode conter muitas variáveis atreladas, até mesmo chamadas de função, para resolver tal questão, eu criei uma função chamada `getAllVarsAndNumbers(self,sourceNode,scope)`, passando um nó de origem(que sempre é uma expressão ou converge a uma expressão), essa função retorna todos os nós que sejam variáveis, números ou funções, dessa forma foi fácil decifrar qual o tipo em questão de uma variável que recebe uma operação envolvendo todas essas possibilidades, pois se um inteiro realiza operações somente com números inteiros, não há problema nem necessidade de gerar warnings, o mesmo vale para o tipo flutuante.

Outro ponto importante, para permitir saber que variáveis já foram utilizadas em cada escopo, toda vez que ela sofre uma atribuição ou é envolvida por uma operação de um terceiro, adicioná-la a uma lista de variáveis já utilizadas, para que depois do processamento geral da função de busca, seja fácil saber quais variáveis foram iniciadas para comparar com as declaradas, gerando assim os warnings almejados pela análise semântica.

Então, o código da chamada que realiza a busca de erros na árvore sintática, de forma análoga ao preenchimento da tabela de símbolos, percorre-se os nós iniciais da árvore, expandindo a princípio as inicializações e declarações de funções pertencentes ao escopo global, uma vez que os nós das funções são encontrados, cada um deles sofre o seguinte processamento:

```
def findFunctionSemanticErrors( self , declFuncNode , tabelaSimbolos ):
    if len(declFuncNode.children)==2:
        cabecalho = declFuncNode.children[1]
    else:
        cabecalho = declFuncNode.children[0]
    scope = cabecalho.children[0]
    corpo = cabecalho.children[4]
    checouRetorno=ReturnChecker(0)
    self.processCorpo(corpo , scope , tabelaSimbolos , checouRetorno)
    if( checouRetorno.value==0):
        syb = tabelaSimbolos.getVar(scope , " global ")
        self.addException("funcao "+scope+" sem retorno ")
    return ""

def processCorpo(self , corpo , scope , tabelaSimbolos , checouRetorno ):
    types = ["acao" ," expressao "]
    actuals = []
    nexts = corpo.children
    actuals.append(corpo)
    while len(nexts)!=0:
        nexts=[]
        for node in actuals:
            for next in node.children:
                if next.type=="atribuicao ":
                    self.processarAtribuicao
                    (next , scope , tabelaSimbolos )
                elif next.type=="expressao_logica ":
                    expVarList = self.getAllVarsAndNumbers
                    (next , scope , tabelaSimbolos )
                    self.checkFunctionParams
                    (expVarList , tabelaSimbolos , scope )
                elif next.type=="retorna ":
                    self.checkFunctionReturn(next , scope , scope , tabelaSimbolos )
                    checouRetorno.value=1
                elif next.type=="leia ":
                    var = next.children[2]
                    varName = var.children[0]
                    exists = tabelaSimbolos.checkVarExists
                    (varName , scope )
                    if exists==0:
                        self.addException ( "... ")
```

```

else:
    simbolo = tabelaSimbolos.getVar(varName, scope)
    simbolo.isUsed=1
    simbolo.isInitialized=1
elif next.type=="escreva":
    expressao = next.children[2]
    self.checkIfVarListExists
    (expressao, scope, tabelaSimbolos, "ESCREVA")
elif next.type=="se":
    expression = next.children[1]
    corpo1 = next.children[3]
    innerScope = scope+".se"+str(next.sequenceId)
    self.checkIfVarListExists
    (expression, scope, tabelaSimbolos, "SE")
    self.processCorpo
    (corpo1, innerScope, tabelaSimbolos, checouRetorno)
    self.checkIfVarListExists
    (corpo1, innerScope, tabelaSimbolos, "SE")
    if len(next.children)>5:
        corpo2 = next.children[5]
        self.processCorpo
        (corpo2, innerScope, tabelaSimbolos, checouRetorno)
        self.checkIfVarListExists
        (corpo2, innerScope, tabelaSimbolos, "SE")
elif next.type=="repita":
    corpo = next.children[1]
    expressao = next.children[3]
    innerScope = scope+".rp"+str(next.sequenceId)
    self.checkIfVarListExists
    (expressao, scope, tabelaSimbolos, "REPITA")
    self.checkIfVarListExists
    (corpo, innerScope, tabelaSimbolos, "REPITA")
    self.processCorpo
    (corpo, innerScope, tabelaSimbolos, checouRetorno)
elif next.type in types:
    nexts.append(next)
actuals = nexts

```

Como pode-se ver, a função processCorpo lida com os warnings e erros das chamadas de atribuição, e ao encontrar um bloco SE ou REPITA, ela faz uma chamada recursiva ao processCorpo, acrescentando ao escopo o valor que foi anotado no nó da árvore sintática pela geração da tabela de símbolos, para que ele consiga encontrar o escopo adequado na busca do símbolo em questão.

O ponto principal para realizar a validação de tipos foi criar uma maneira de obter as variáveis de uma dada expressão, e é exatamente isso que a função getAllVarsAndNumbers faz, ela percorre os nós de uma expressão procurando por qualquer nó que seja

do tipo var, numero ou chamada de funcao, pois com esses nós, e o escopo em questão que é passado como parametro para a função, fica fácil de determinar se uma variável está "ok", bastando realizar uma busca na tabela de símbolos e verificar se ela existe no escopo em questão.

A função processarAtribuicao, também realiza uma chamada para a getAllVarsAndNumbers, no nó da expressão referente a atribuição, tudo que ela faz é verificar se o valor obtido condiz com o tipo esperado pela tabela de símbolos.

Seguem alguns exemplos da saída do programa:

```
entrada :
inteiro : a
flutuante : b
```

```
saida :
WARNING: a declarado mas nunca usado no escopo global
WARNING: b declarado mas nunca usado no escopo global
SEMANTIC ERROR: Funcao principal nao declarada
```

```
entrada :
flutuante : c[2.5][5]
inteiro : n
```

```
inteiro principal()
  n := 5
  se n>0 ent o
    inteiro: k
    k:=3
  fim
  se n<2 ent o
    inteiro: p
    inteiro: j
    j := 2.3
  fim
  retorna 0
fim
```

```
saida :
WARNING: c    declarado mas
nunca usado no escopo global
WARNING: p    declarado mas
nunca inicializado no
escopo principal.se2
WARNING: j e do tipo inteiro mas
recebe um valor de outro tipo
no escopo principal.se2
```


SEMANTIC ERROR: Erro de indice ,
indice 2.5 de tipo float
em dimensao na declaracao do array c

6. Referências

Seguem as referências utilizadas nesse trabalho:

[1]LOUDEN, Kenneth C. Compiladores: princípios e práticas. São Paulo, SP: Thomson, c2004. xiv, 569 p. ISBN 8522104220

Referências

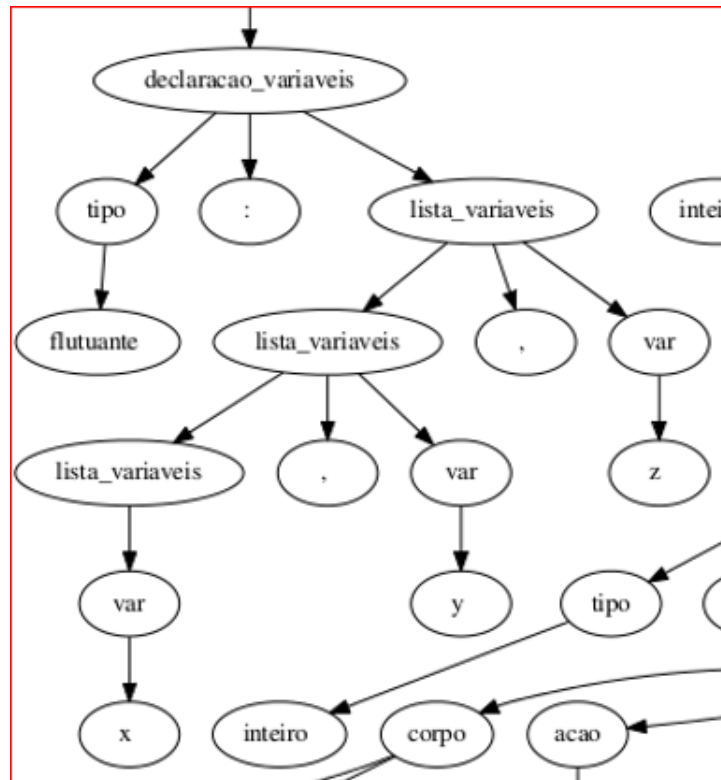


Figura 4. Árvore sintática

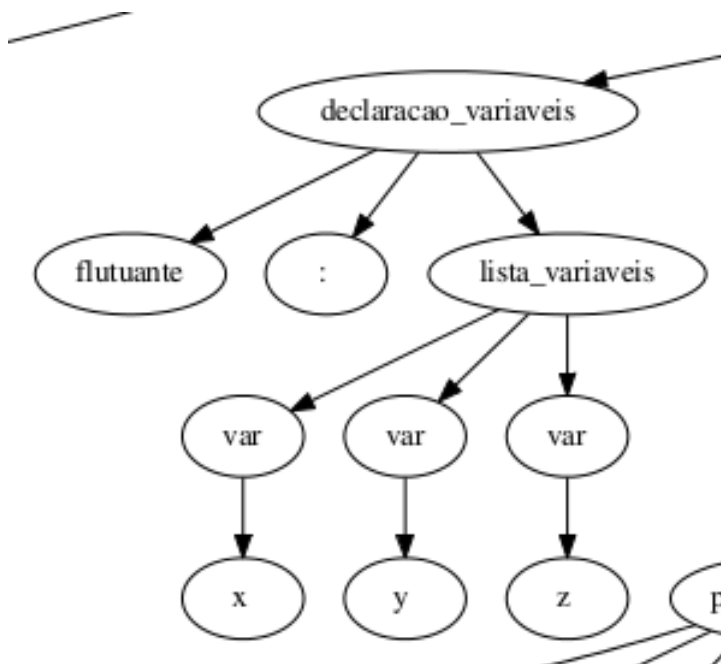


Figura 5. Árvore sintática reduzida