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 analyzes, followed by the generation of intermediate code, we used the PLY framework to identify the tokens from the input programs and to build the syntactic tree, then we generated an noted three and a symbolic tables at the semantic step, finally, we used the library llvmlite to generate the llvmcode from our three.*

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 intermiário, o processo de tokenização e construção da árvore sintática foram feitos com o auxílio da biblioteca PLY. Então foi gerada uma árvore anotada e uma tabela de símbolos na análise semântica, finalmente, utilizamos a biblioteca llvmlite para gerar o codigo llvm da nossa árvore.*

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)
- codigo
- fim

Observe que se faz necessária a declaração do tipo da função, bem como a declaração do tipo de todos os paramêtros presentes na lista de entrada, a virgula 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 especifica.

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, SENAO>

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 analise 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 carácteres { }, o token { estará incluso, mas o de fechamento excluso 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 ve 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 porquê é complicado separar essas palavras por uma expressão regular comum, para tal tarefa utilizamos uma função 'coringa', é muito facil dos tokens da classe OID se confundirem com os de palavra reservada, por isso ao invez 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 sequencia de letras e/ou numeros 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 simbolos.

A t_SYB é mediocremente simples, se trata apenas de um ou entre os possíveis simbolos 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 comentario mesmo foi identificado ainda que houvesse em mais que uma linha.

3.5. Automatos 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

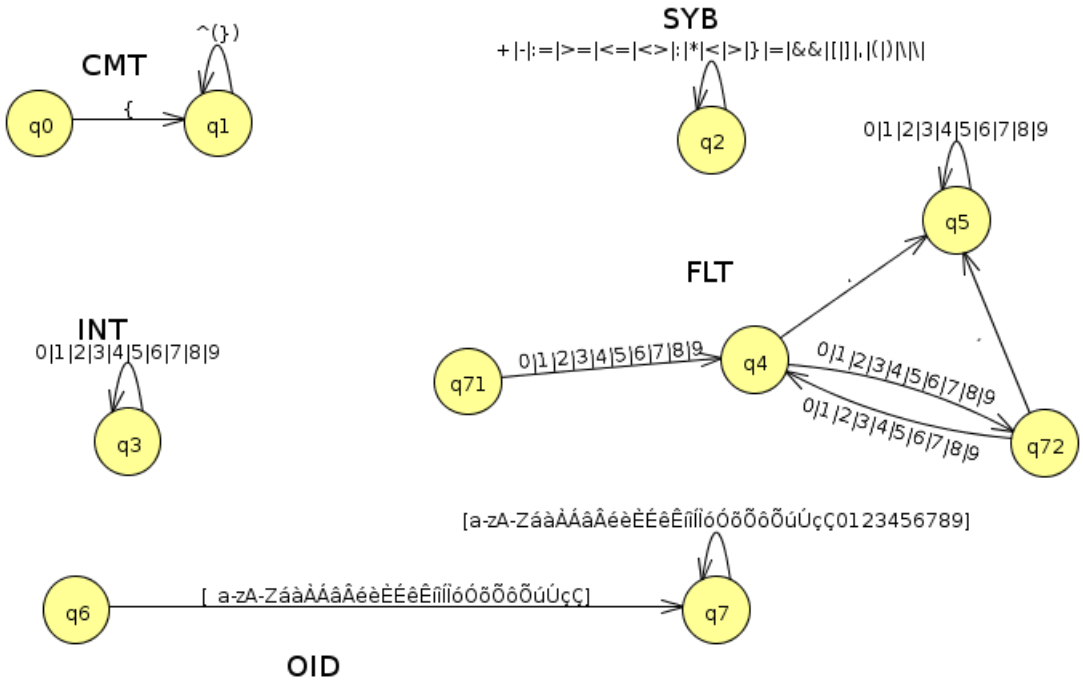

Figura 1. Autômatos das expressões OID, RWO, CMT, SYB, INT, FLT

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. Analise 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

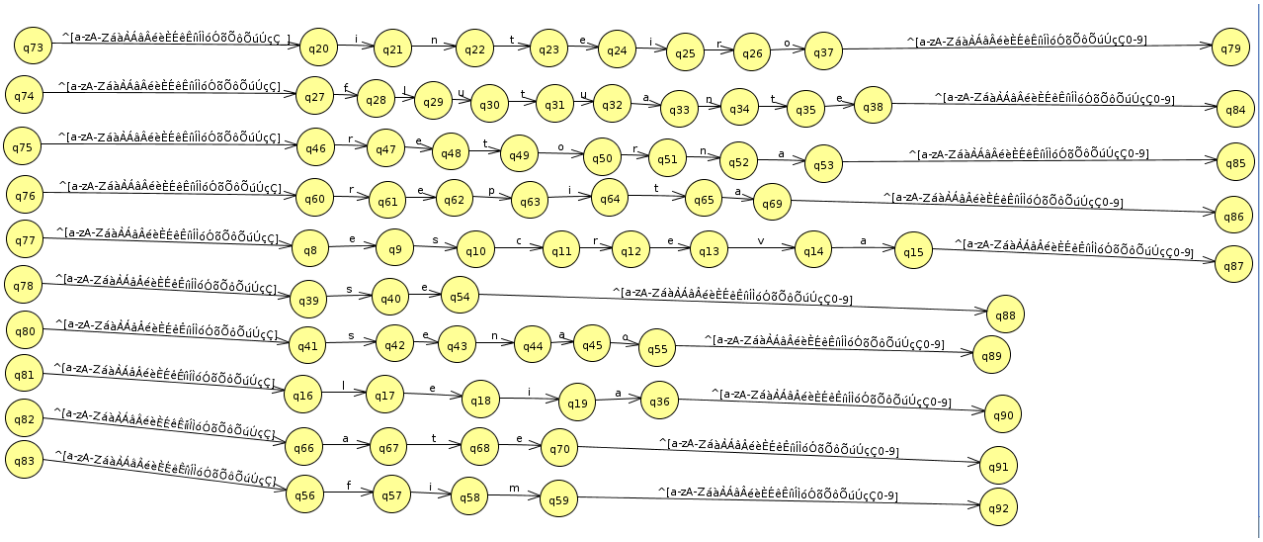

Figura 2. Autômatos das palavras reservadas

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 LPARENTHESYS lista_parametros RPARENTHESYS 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 LPARENTHESYS var RPARENTHESYS

escreva : ESCREVA LPARENTHESYS expressao RPARENTHESYS

retorna : RETORNA LPARENTHESYS expressao RPARENTHESYS

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 : LPARENTHESYS expressao RPARENTHESYS
      | var
      | chamada_funcao
      | numero

numero : INT
       | FLOAT

chamada_funcao : ID LPARENTHESYS lista_argumentos RPARENTHESYS

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 trada 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 simbolos da gramática são substituidos pelo simbolo gramatical do lado esquerdo da declação.

O LR-parsing geralmente é implementado utilizando enviando simbolos gramaticas para uma pilha, adicionalmente olhando para a pilha e para o proximo 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
Gram tica                    A   o
-------------------------    ------------------------------------------

exp0 : exp1 + term           exp0.val = exp1.val + term.val
     | exp1 - term           exp0.val = exp1.val - term.val
     | term                  exp0.val = term.val

term0 : term1 * factor       term0.val = term1.val * factor.val
      | term1 / factor       term0.val = term1.val / factor.val
      | factor               term0.val = factor.val

factor : NUMBER              factor.val = int(NUMBER.lexval)
       | ( exp )             factor.val = exp.val


#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 espressão, 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 simbolos são substituidos pelo simbolo 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 quanado alcançar o estado onde a pilha de simbolos está vazio 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 gramatica 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 dp 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 gramaticas 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 alguams 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 atribuidas 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,funçõe 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.se1:[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 simbolos 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.se1:[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 varíaveis fez uso de vários nós do tipo listaVariaveis, por razões de simplificação, todos os galhos que possuissem 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 simbolos 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 codigo abaixo percorre os nos da
    #arvore sintatica em busca de
    #declaracoes de funcoes e
    #variaveis 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 declaracao de variaveis,
                #adicione a tabela de simbolos no escopo
                #global
                    symbols = self.getVarList(next,"global")
                        for symbol in symbols:
                            self.put("global",symbol)
                elif next.type=="declaracao_funcao":
                    #Mesma coisa pra funcao
                    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 REPITA 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 simbolos 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 atribuido 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 variaveis, numeros ou funções, dessa forma foi fácil decifrar qual o tipo em questão de uma variavel 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 facil saber quais variaveis 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 simbolos, percorre-se os nós iniciais da árvore, expandindo a principio 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. Geração de código intermediário

Finalmente chegamos a parte final desse trabalho, que é gerar o código intermediário que é efetivamente utilizado para compilar os programas escritos em T++, neste momento novamente iremos percorrer a árvore anotada e utilizaremos a tabela de símbolos que foi gerada na etapa anterior, então iremos utilizar estruturas da biblioteca llvmlite para gerar o código no formato llvm, que pode ser compilado no compilador GCC(utilizando clang).

6.1. A LLVM

LLVM(*L*ow Level Virtual Machine) é uma infraestrutura de compilador escrita em C++, criada para a compilação, ligação e execução de programas escritos em diversas linguagens de programação.

O LLVM pode prover camadas intermediárias de compilador, no sentido que ele realiza otimizações, bem como gerar código binário otimizado em tempo de execução.

Possui uma arquitetura independente de linguagem, pois cada instrução é definida de uma forma padronizada, a logo pode ser vista na 6.

6.2. LLVMlite

LLVMlite é uma biblioteca do python que é compatível como muitas funcionlidades de llvm original, em verdade ela faz uma ligação para com ele.

O llvmlite tem um foco na criação de compiladores JIT(*J*ust in time), e é por isso que ela é focada em duas tarefas, que são a construção de um modulo, função por função, instrução por instrução e compilação/otimização do modulo em código fonte de máquina.

6.3. Estruturas comuns do *llvmlite*

O *llvmlite* provê vários métodos para gerar códigos llvm para variáveis, funções, constantes, atribuições, operações lógicas, operações matemáticas e muito mais, nesa seção estarão listados as estruturas utilizadas nesse projeto bem como uma breve explicação de cada uma.

1. *IntType(32)* : Tipo usado para inteiros.
2. *FloatType()* : Tipo usado para flutuantes
3. *Module* : Módulo onde o programa será carregado, é utilizado apenas um módulo para todo o código.
4. *FunctionType* : Tipo de função, recebe como parâmetro um tipo comum e uma lista de tipos de parâmetros, ex: FunctionType(IntType(32),[])
5. *Function* : Objeto de uma funcao, requer um FunctionType, nome e modulo.
6. *Block*: As operações de funções só podem ocorrer dentro de blocos, geralmente cada escopo tem um bloco inicial e um bloco de "despache"para quando terminar seu procedimento.
7. *funcObj.appendbasicbloc(name)* : Adiciona um irBlock a uma função.

8. *IRBuilder* : Builder de um bloco de função, esse objeto é quem realiza operações de gravação, chamadas e obtenções.
9. *builder.ret(A)* : Retorno de uma função.
10. *builder.call* : Realiza a chamada de uma funcao e guarda o resultado
11. *branch* : Troca para um novo bloco, encerrando o atual.
12. *builder.position*$_a t_e nd$: $Moveocursorparaofinaldobloco$.*builder.store*$(var, LLVMObject)$: $armazenaumconteudodevariavelemumobjeto$.
13. *builder.load* Retorna o conteudo(llvm) de uma variavel alocada.
14. *builder.icmp*$_s igned$: $Usadoparamontarexpressesdecondio$.*builder.cbrach* : $Mudaparauma intervaloespecificodeblocos, sendoentredois$.
15. *builder.alloca*(ir.IntType(32), name) : Função para alocar uma variavel, retorna seu objeto.
16. ir.ArrayType : Tipo especifico para arrays.
17. *GlobalVariable* : Tipo especifico para variaveis globais, necessario pois essas geralmenta nao estão em um bloco de função.

Essas são as estruturas recorrentes no projeto, é interessante

6.4. Estratégias utilizadas para a geração de código

Como já mencionado, utilizaremos muito do que já foi produzido na **Análise Semântica**, mais especificamente, novamente iremos percorrer a árvore simplificada bem como utilizaremos a tabela de símbolos.

Na tabela de símbolos, já possuimos informações de tipo e escopo para todas as variáveis e funções do código análisado, no entanto, agora desejamos expandir as informações para cada símbolo da tabela, como segue:

- *codeObject* = Código de objeto LLVM, usado em variaveis e funções.
- *builder* = Builder de um modulo LLVM, usado em funções.
- *module* = Modulo LLVM
- *entryBlock*=Bloco de entrada de escopo, usado para funções.
- *endBlock* = Bloco de termino de escopo, usado em funções.

Todos esses campos serão carregados com instâncias de objetos do *llvmlite*, o campo de maior importância é o *codeObject*, pois ele armazena desde valores diretos como *ir.Constant* até referências a objetos de variáveis, funções e arrays, que podem ser carregados por loads e calls.

Mas vamos logo à intuição do algoritmo, novamente partimos do topo na nossa árvore, os primeiros alvos são os nós de declarações de funções e variáveis, que são os possíveis artefatos de interesse do escopo global, para ilustrar, a figura 8 nos mostra a estrutura da árvore em questão.

O caso inicial mais simples é o de declaração de variáveis, nesse caso, para todos os filhos de declaracaoVariaveis, pegamos os nós que sejam do tipo var, e para cada um coletamos a sua instancia *Symbol* da tabela de símbolos.

```
if next.type=="declaracao_variaveis":
  vars = self.getVarList(next,scope)
  for var in vars:
    reference = tabelaSimbolos.getVar
```

```
(var.name, scope)
self.addGlobalVarDeclaration
(reference, tabelaSimbolos)
```

Como o codigo exibe, tendo a referencia do objeto da tabela de simbolos, realizamos uma chamada a uma função que gera o código llvm para uma variável global, que é a função *self.addGlobalVarDeclaration*, um trecho interessente da função seria...

```
if varSymbol.type=="inteiro":
    tmpType=ir.ArrayType(ir.IntType(32), fatorMultiplicador)
    arrayA = ir.GlobalVariable(self.globalModule,
    tmpType, varSymbol.name)
```

Nesse trecho, caso o objeto(passado por parametro) da tabela de símbolos seja inteiro, criamos um objeto llvm no escopo global para essa variável.

O caso para variáveis flutuantes também foi implementado, de forma bastante semelhante, como o código é análogo ao exibido, mudando apenas os tipos, não o mostraremos aqui.

Bem, mas ainda faltou o código que trata das declarações de funções, quando encontramos um nó de declaração, o repassamos para uma função chamada *genFunctionCode*, essa função é responsável por registrar a função, seu corpo e tudo que nele existe, ou seja, atribuições, chamadas de funções e blocos se e repita.

O código mostra que, para funções realiza-se uma busca de seu simbolo na tabela de simbolos, entao armazenamos seu modulo, objeto de função, bloco de entrada e builder ao *Symbol*, bem como aos parametros, isso é muito importante pois mais tarde a chamada dessa função se dará por meio do seu *codeObject*.

```
funSymbol = tabelaSimbolos.getVar(scope,"global")
funSymbol.module = self.globalModule
funSymbol.codeObject = ir.Function
(funSymbol.module, functionType, name=funcName)
funSymbol.entryBlock=funSymbol.codeObject.append_basic_block
(name=(funcName+"Entry"))
funSymbol.endBlock = None
funSymbol.builder = ir.IRBuilder(funSymbol.entryBlock)
tabelaSimbolos.updateVar("global",funSymbol)
for i in range (0,len(funSymbol.params)):
  funSymbol.params[i].codeObject=funSymbol.codeObject.args[0]
```

Uma vez que a "registramos", precisamos realizar tudo que há no corpo dessa função, para tal, chamamos a função *generateCorpoCode*.

O processamento do corpo de um dado escopo é o coração deste sistema, pois é aqui que se encontram as atribuições, blocos de se/repita, funções de leitura e funções de escrita, se esse trecho funcionar como deve, basicamente cada função irá o invocar, e irá gerar o código llvm ideal para si e para o seu corpo.

Como podemos ver na figura 7, todos os filhos do corpo são nós do tipo ação, no entanto, estamos interessados apenas nos filhos efetivos dessas ações, ou seja, os casos

de nó de **atribuição, declaração de variável, retorna, leia, escreva, repita e se**. Mas antes de falar delas, falaremos sobre uma função que processa uma expressão não lógica, ou seja, do tipo que pode ser var, aditiva, multiplicativa ou chamada de função, pois ela é muito usada em diversos casos.

6.4.1. genExpressionNode

Essa é a função responsável por lidar com numeros, variáveis, chamadas de funções e expressões de matemática básica para quem quer que a chame, no caso de número, ela simplesmente gera uma nova constante inteira, se for var, ele simplesmente retorna o que estiver carregado no codeObject da sua correspondente na tabela de símbolos, em caso de expressões matemáticas, infelizmente foi-se implementado somente para operações com numeros inteiros, caso seja uma chamada de função, ele realiza um call para o seu objeto llvm, e retorna o que nele vier.

6.4.2. Atribuição

No caso da atribuição, o nó é repassado para uma função *genCodeAtribuicao*, que por sua vez, pega da tabela de símbolos a varíavel ao lado esquerdo da expressão, bem como o nó de operação(chamada de função, expressao aditiva/multiplicativa, ou caso nenhuma desses exista ele pega o nó var ou o numero) ao lado direito da atribuição, tendo o nó que representa a expressão ao lado direito e o símbolo que representa o esquerdo, ele invoca uma função que toma ambos como parâmetro para gerar o seu llvm, essa função é a *genAtributionExpressionNode*.

Essa *genAtributionNode* delega o nó da expressão(lado direito) a função *genExpressionNode*, que retorna o gera o llvm da expressão(seja ela uma operação matematica, constante, variavel ou chamada de função) e retorna o objeto llvm correspondente. Então, a *genAtributionNode* coloca no *codeObject* da variável do lado direito da expressão o que veio do resultado da *genExpressionNode*.

6.4.3. Declaração de variável

Esse caso é bem simples, e se assemelha bastante com o caso da função main, basicamente pega-se a variável existente na tabela de símbolos, e então a passa para a função *genCodeVarDeclaration*, que checa os tipos do varSymbol em questão e declara a variável adequadamente.

6.4.4. Retorna

A primeira coisa que fazemos é pegar o simbolo da função "dona"o retorna, depois, mudamos o bloco da função para o bloco de finalização, a partir dai, alocamos uma nova variável, que pode ser do tipo inteiro ou flutuante, dependedo do tipo da função.

Então, assim como no caso da atribuição, invocamos a *self.genAtributionExpressionNode*, para processar a expressão de retorno ao lado

direito(que também gera o llvm), realizamos um load no resultado dessa função, e com o comando

```
scopeSymbol.builder.ret(loadVarSymbol)
```

definimos o retorno da função, no llvm.

6.4.5. Leia

A função leia é um caso mais complicado, para a executar, foi definida uma variável de função llvm global que é chamada toda vez que leia é invocada, como segue:

```
self.leiaFunTypeInt = ir.FunctionType(ir.IntType(32),[])
self.leiaFunObjectInt =ir.Function
(self.globalModule, self.leiaFunTypeInt, 'LeiaInteiro')
self.leiaFunEntryBlock=self.leiaFunObjectInt.append_basic_block
(name=("LeiaEntry"))
#funSymbol.endBlock = None
self.leiaBuilder = ir.IRBuilder(self.leiaFunEntryBlock)
self.leiaBuilder.ret(ir.Constant(ir.IntType(32),5))
```

Tendo essa definição global para função, basta checar o seu tipo antes da invocação, e realizar a chamada do call, guardar o seu resultado, e o armazenar no codeObject da variável que recebeu o leia.

```
if varSymbol.type=="inteiro":
  res = scopeSymbol.builder.call(self.leiaFunObjectInt,[])
  scopeSymbol.builder.store(res, varSymbol.codeObject)
elif varSymbol.type=="flutuante":
 res = scopeSymbol.builder.call(self.leiaFunObjectFloat,[])
  scopeSymbol.builder.store(res, varSymbol.codeObject)
```

6.4.6. Escreva

Assim como no caso da função leia, tivemos que definir uma função global para o escreva.

```
self.escrevaTypeInt = ir.FunctionType
(ir.IntType(32), [ir.IntType(32)])
self.escrevaFunInt = ir.Function
(self.globalModule, self.escrevaTypeInt, 'EscrevaInteiro')
self.escrevaFunEntryBlock=self.escrevaFunInt.append_basic_block
(name=("EscrevaEntry"))
self.escrevaBuilder = ir.IRBuilder(self.escrevaFunEntryBlock)
self.escrevaVar = self.escrevaBuilder.alloca
(ir.IntType(32),name="escrevaBuffer")
self.escrevaVar.align=4
self.escrevaBuilder.ret(ir.Constant(ir.IntType(32),1))
```

Então, voltando a função que gera o llvm para o corpo.

```
expressao = next.children[2]
result = self.genExpressionNode
(expressao,scopeSymbol,tabelaSimbolos)
if "i32" in str(result):
  scopeSymbol.builder.call(self.escrevaFunInt,[result])
else:
  scopeSymbol.builder.call(self.escrevaFunFloat,[result])
```

Como pode-se ver, apenas realizamos o call para o que foi retornado da expressão que faz parte do escreva, e novamente utilizamos o self.genExpressionNode.

6.4.7. genSimpleLogicExpression

Nos casos do repita e do se, existem nós de expressão que nós sabemos que vão vair em casos de expressões simples, por isso, essa função recebe um nó de expressão lógica, encontra uma expressão simples(do tipo a>b, 1<3, etc...), monta o objeto llvm da expressão e o retorna, a montagem é como no exemplo:

```
logicExp = scopeSymbol.builder.icmp_signed
(str(operation), result0, result1 )
```

6.4.8. Repita

No caso do repita, a primeira coisa que fazemos é usar a *genSimpleLogicExpression* para pegar o objeto llvm da comparação, então, nós criamos um novo bloco, utilizando o builder da função, depois, definimos o escopo desse bloco, acrescentendo a ele um '.rp' e o sequenceId que existe no seu símbolo(isso foi feita na etapa semantica, para cada simbolo da tabela).

Também se faz necessario criar um bloco de finalização para o novo escopo, depois de os criar, mudamos para o novo escopo, e recursivamente chamamos a funcao *generateCorpoCode*, passando o corpo do repita e o novo escopo.

```
innerScope = scope+".rp"+str(next.sequenceId)
newScopeBlock=scopeSymbol.codeObject.append_basic_block
(name=innerScope)
newEndBlock = scopeSymbol.codeObject.append_basic_block
(name=innerScope+"Fim")
scopeSymbol.builder.cbranch(expGr,newScopeBlock,newEndBlock)
scopeSymbol.builder.position_at_end(newScopeBlock)
self.generateCorpoCode
(corpo,scopeSymbol.name,scopeSymbol,tabelaSimbolos)
```

6.4.9. Se

Assim como no caso do enquanto, no se é preciso passar descobrir a expressao logica com o *genSimpleLogicExpression*, e tambem criamos novos blocos de inicio e fim para o se,

e recursivamente chamamos a generateCorpoCode, mas além disso, caso se venha acompanhado do senão, é criado um novo bloco para o senão, e mais uma chamada recursiva ocorre, passando o corpo do senão como parâmetro.

6.5. Considerações sobre a implementação

Fizemos um *tour* sobre como o código gera o llvm para a árvore anotada da etapa anterior, vimos que ele precisa processar as declarações de variáveis e funções do escopo global, e depois se aprofundar dos corpos das funções, isso resume toda a implementação da geração de código desse compilador em T++.

6.6. Exemplos de execução

Bom, como um velho ditado diz, a melhor prova de um pudim é comê-lo, no nosso caso não temos um pudim, mas sim um compilador, então, para testâ-lo, você deve o baixar, ele estará disponível em breve no repositório:

- https://github.com/nakaosensei/

Procure por um projeto de Compilador T++, ele estará lá.

Para executar o projeto, entre na pasta da implementação, digite python3 nkParser.py exemplos/geracao-codigo-tests/gencode-001.tpp Isso ira imprimir o arquivo llvm gerado. Para descobrir se deu tudo certo, você pode o compilar, para tal:

```
llc -3.9 gerado.ll
gcc -c gerado.s
gcc -o saidaGeracao gerado.o
./saidaGeracao
```

6.6.1. Exemplo 1

Bem, vamos rodar alguns testes, o primeiro teste será com um codigo bem simples.

```
inteiro: a
inteiro principal()
        inteiro: b
        a := 10
        b := a
    retorna(0)
fim
```

A saida é o código llvm, que é escrito em um arquivo.ll, que para esse caso foi:

```
; ModuleID = "nkModuloGlobal.bc"
target datalayout = ""
define i32 @"LeiaInteiro"()
{
LeiaEntry:
  ret i32 5
}
```

```
define i32 @"EscrevaInteiro"(i32 %".1")
{
EscrevaEntry:
  %"escrevaBuffer" = alloca i32, align 4
  ret i32 1
}
@"a" = common global i32 0, align 4
define i32 @"main"()
{
mainEntry:
  %"b" = alloca i32, align 4
  store i32 10, i32* @"a"
  %".3" = load i32, i32* @"a"
  store i32 %".3", i32* %"b"
  br label %"principalEnd"
principalEnd:
  %"retornNkprincipal" = alloca i32, align 4
  store i32 0, i32* %"retornNkprincipal"
  %".7" = load i32, i32* %"retornNkprincipal"
  ret i32 %".7"
}
```

Executamos os comandos do llc e gcc, geramos o executavel, e ele executou sem problemas.

6.6.2. Exemplo 2

Entrada:

```
inteiro: a
inteiro principal()
        inteiro: ret
        a := 10
        se a > 5 ent o
        ret := 1
    sen o
        ret := 0
    fim
    retorna(ret)
fim
```

Saída:

```
; ModuleID = "nkModuloGlobal.bc"
target datalayout = ""
define i32 @"LeiaInteiro"()
```

```
{
LeiaEntry:
  ret i32 5
}
define i32 @"EscrevaInteiro"(i32 %".1")
{
EscrevaEntry:
  %"escrevaBuffer" = alloca i32, align 4
  ret i32 1
}
@"a" = common global i32 0, align 4
define i32 @"main"()
{
mainEntry:
  %"ret" = alloca i32, align 4
  store i32 10, i32* @"a"
  %".3" = load i32, i32* @"a"
  %".4" = icmp sgt i32 %".3", 5
  br i1 %".4", label %"principal.se1", label %"principalEnd"
principal.se1:
  store i32 1, i32* %"ret"
  br i1 %".4", label %"principal.se1Else", label %"principalEnd"
principalEnd:
  %"retornNkprincipal" = alloca i32, align 4
  %".10" = load i32, i32* %"ret"
  store i32 %".10", i32* %"retornNkprincipal"
  %".12" = load i32, i32* %"retornNkprincipal"
  ret i32 %".12"
principal.se1Else:
  store i32 0, i32* %"ret"
  br label %"principalEnd"
}
```

Após a execução, foram executados so comandos do llc e gcc, e todos foram bem sucedidos.

```
llc -3.9 gerado.ll
gcc -c gerado.s
gcc -o saidaGeracao gerado.o
```

7. Conclusão

Implementar um compilador para uma linguagem qualquer, ainda que tendo o auxilio de ferramentas fantásticas como a llvm, expressões regulares e gramáticas livres de contexto, não é uma tarefa trivial. Mas apesar disso, o entendimento de como se dá o processo do início ao fim foi uma experiência importantíssima para efetivamente atuar na área dependendo menos de "caixas pretas", foi um trabalho interessante e divertido de implementar, apesar de quê infelizmente a geração de código não ficou perfeita.

8. 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

Figura 3. Árvore Gerada

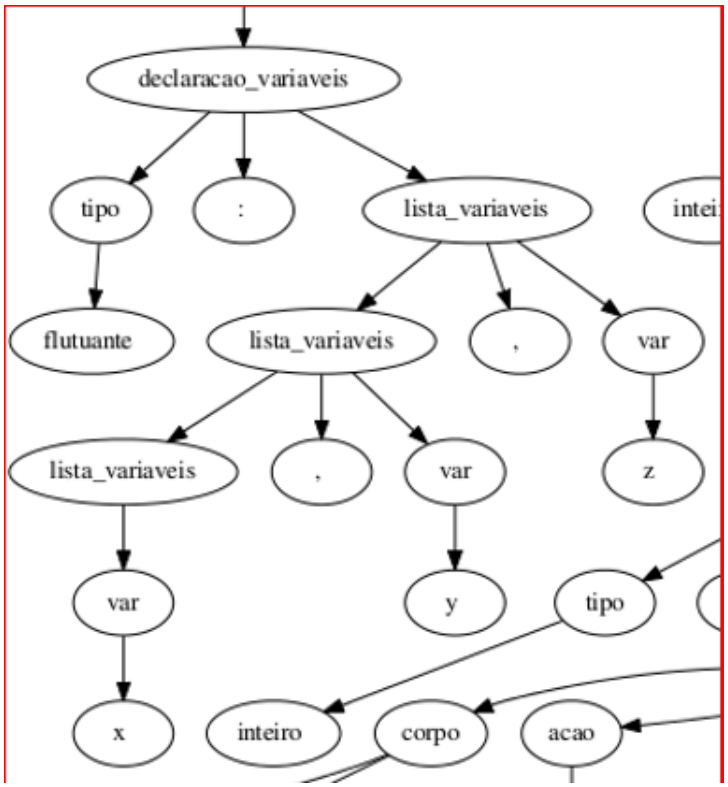

Figura 4. Árvore sintática

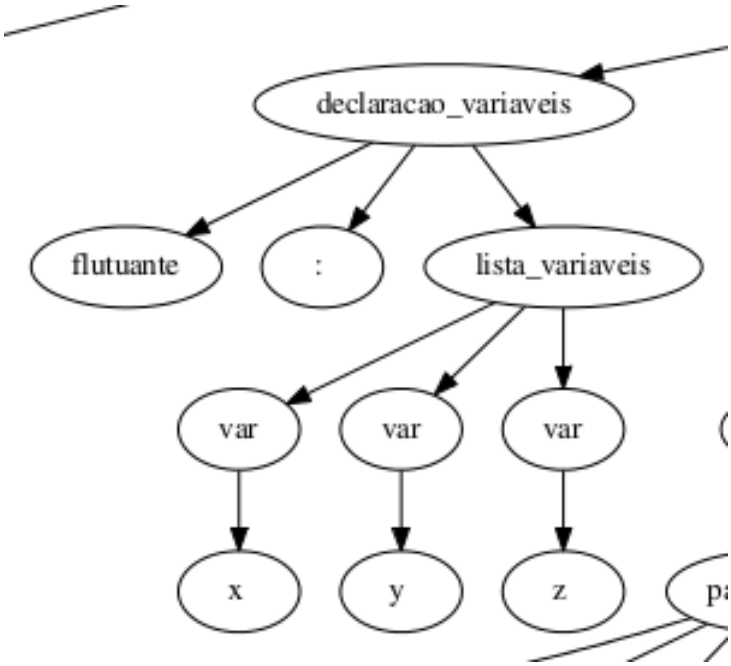

Figura 5. Árvore sintática reduzida

Figura 6. Logo LLVM

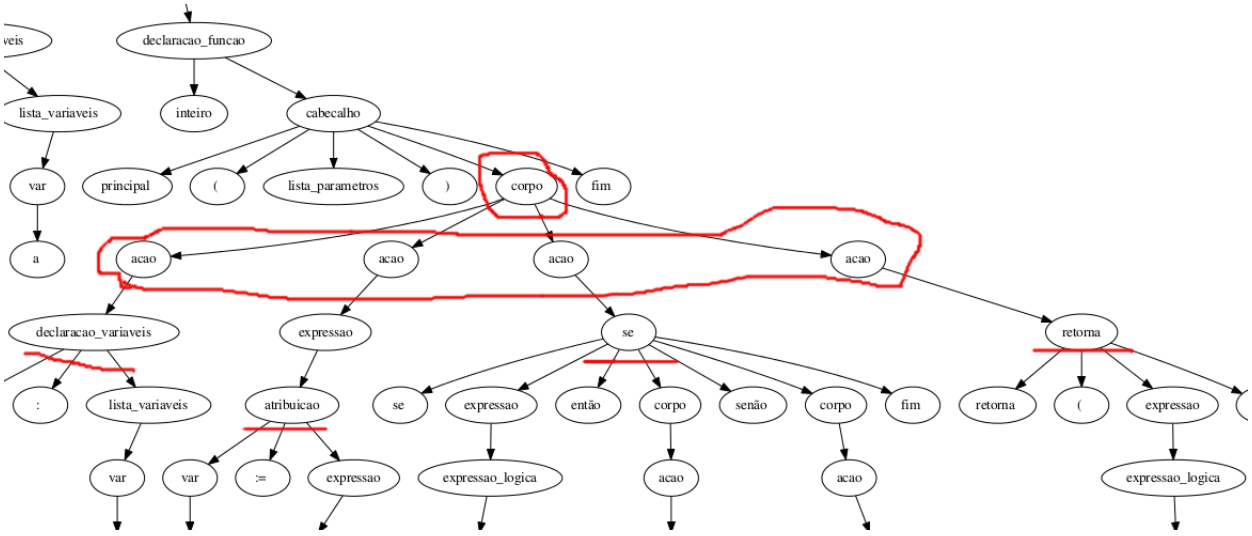

Figura 7. Nós filhos do corpo

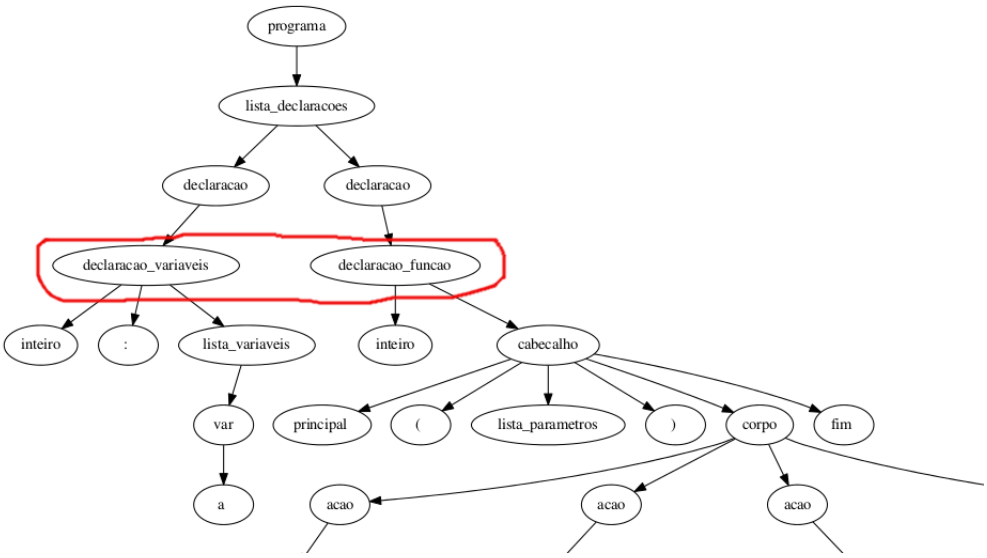

Figura 8. Nós iniciais