Projeto de Implementação de um Compilador para a Linguagem T++

Juliano Cesar Petini

Universidade Tecnológica Federal do Paraná (UTFPR) Campo - Mourão– Brazil

[1]julianopetini@alunos.utfpr.edu.br

Abstract. *The following work aims to make a compiler for a Tpp language, starting with the lexical analysis that consists of an algorithm to check and identify the tokens in an input file, in which this input is a file of type ' Tpp 'and will generate related tokens for each predefined type. Then a syntax analyzer was implemented, which using the tokens generated by the first part together with the defined ones, to be able to generate a program tree that can be used later in the last step which is the generation of intermediate code in which it allows the machine to execute the created algorithm.*

Resumo. *O trabalho a seguir tem com o objetivo conseguir fazer um compilador para a linguagem Tpp, começando pela análise léxica que consiste em um algoritmo para verificar e identificar os tokens de um arquivo de entrada, no qual essa sua entrada é um arquivo do tipo 'Tpp' e irá gerar tokens relacionados para cada tipo predefinido. Em seguida foi implementado um analisador sintático, que utilizando os tokens gerados pela primeira parte juntamente com as regras definidas, conseguem nos gerar uma árvore do programa que podemos utilizar posteriormente na última etapa que é a geração de código intermediário no qual possibilita a máquina executar o algoritmo criado.*

1. Introdução

O Trabalho tem como objetivo de conseguir fazer um compilador começando com a análise léxica em cima de um algoritmo da linguagem *'Tpp'*, após efetuar a análise o algoritmo retorna uma lista de tokens para a entrada de um algoritmo que possa fazer uma análise sintática. para realizar o trabalho foi utilizado a linguagem python com uso do *'Ply'* em conjunto com as expressões regulares *'Regex'*. O trabalho está dividido em partes, iremos apresentar cada parte do projeto em sequência.

Na primeira parte iremos apresentar o Python *'Ply',* iremos dar uma ideia de como funciona e como está sendo utilizado no algoritmo, logo em seguida será explicado o'que é expressões regulares *'Regex'* e também como está sendo utilizado em nosso projeto. Já a seguir iremos falar sobre oque se trata a análise léxica qual a sua ideia e como ela funciona dando alguns exemplos e mostrando como nosso algoritmo tenta reproduzir essa ideia. No próximo capítulo iremos falar sobre o *'Tpp'* a linguagem que foi implementada, e o compilador para reconhecimento, juntamente com a implementação do algoritmo, como funciona os tokens, como ele identifica cada um dos token, e também o'que ele retorna após executar o programa

e por fim algumas conclusões falando sobre o desempenho do algoritmo com base em vários testes executados.

Na Segunda Parte iremos apresentar o Python 'Yacc', iremos mostrar como ele nos ajuda na parte de análise sintática do código que geramos pelo lexer do passo anterior, juntamente a isso também será mostrado a árvore que é gerada ao executar o algoritmo que nos mostra exatamente como está sendo chamado e montado a lógica do programa resultante, e também como foram feitas algumas expressões de tratamento de erro.

O Projeto teve início com a disciplina de compiladores da Universidade Tecnológica Federal do Paraná - Campo Mourão onde foi feita a primeira parte do projeto que consiste em fazer o desenvolvimento de um compilador, separando o desenvolvimento em três partes a primeira é a implementação de um analisador léxico, em seguida um analisador sintático e por fim a geração de código.

2. Python Ply

O '*Ply'* é uma implementação de ferramentas para análise léxica e sintática no python onde basicamente ele faz um parsing com ajuda do *Regex* para identificar tokens, que são palavras ou símbolos que são reservadas na linguagem. A versão original da biblioteca foi desenvolvida em 2001 por David Beazley para uso em um curso de Introdução aos Compiladores, em que os alunos a usavam para construir um compilador para uma linguagem simples do tipo Pascal. Devido ao seu uso em um ambiente instrucional, muito trabalho foi feito para fornecer uma extensa verificação de erros. Além disso, essa experiência foi usada para resolver problemas comuns de usabilidade. Desde então, uma variedade de melhorias incrementais foram feitas na biblioteca.

2.1. Expressões regulares

Expressões regulares '*Regex'* e um notação para poder identificar um padrão em uma string, serve para validar entradas de dados ou fazer busca e extração de informações em textos. Expressões regulares são escritas numa linguagem formal que pode ser interpretada por um processador de expressão regular, um programa que serve como um examinador de texto e identifica as partes que combinam com a especificação dada.

No algoritmo criado são utilizadas várias expressões regulares para identificar padrões, iremos apresentar algumas delas como autômatos, na (Figura-1) que irá mostrar um autômato que reconhece a palavra *'inteiro',* na (Figura-2) apresenta um autômato que reconhece a palavra *'repita'*, e por fim na (Figura-3) iremos apresentar um autômato que reconhece a palavra *'leia'*, esse são apenas alguns de vários outros que foram implementados no projeto.

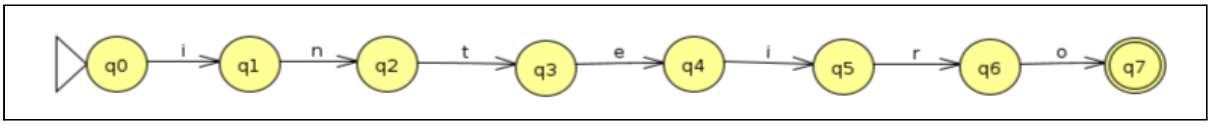

Figura-1 Apresenta o autômato que identifica o token *inteiro.*

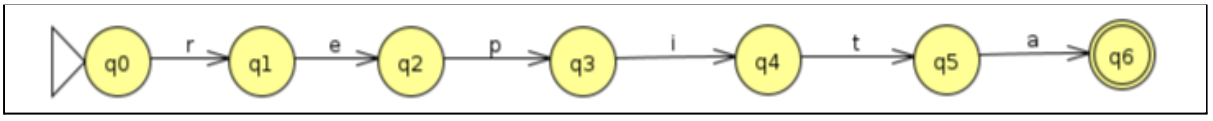

Figura-2 Apresenta o autômato que identifica o token *repita.*

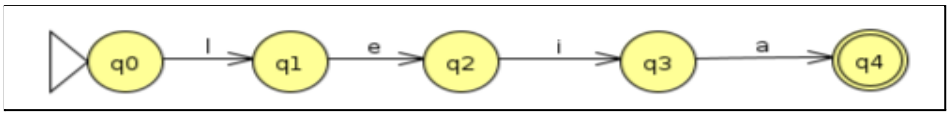

Figura-3 Apresenta o autômato que identifica o token de *leia.*

2.2. Análise Léxica

Léxico formalmente é o conjunto de palavras existente em um determinado linguagem que as pessoas têm à disposição para expressar-se, oralmente ou por escrito em seu contexto, são basicamente palavras predefinidas de uma linguagem a serem seguidas, no projeto criado a nossa linguagem léxica tem várias palavras e símbolos reservados algumas já foram apresentada aqui como a palavra *inteiro*,*leia* ou a *repita*, porém existem várias outras, cada uma dessas palavra é capturadas pelo nosso analisador léxico e caso esteja escrita correto ela vira um token de saída.

2.3. Algoritmo

Para apresentar o algoritmo iremos separar em duas partes, a primeira iremos falar um pouco da ideia do tpp e como surgiu a suas especificações, já na segunda parte iremos falar da implementação do algoritmo e como foi feita cada parte e o'que cada parte é responsável em fazer.

2.4.. Tpp

A ideia do '*Tpp'* e ser uma linguagem de fácil entendimento permitindo criar algoritmos sem dificuldades, com semelhanças de linguagem de alto nível e também ter aspectos bem intuitivos, a descrição da linguagem tem origem na disciplina de compiladores com intuito de criarmos uma nova linguagem para nosso compilador.

2.5. Implementação

A implementação do algoritmo foi feito utilizando a linguagem python com a biblioteca '*Ply'* e o '*Regex',* para começar o algoritmo precisamos primeiro definir uma lista de tokens que serão nossas palavras e símbolos reservados, na (Figura-4) apresentamos como está feito a lista de tokens no compilador.

```
def Toke():
    tokens = (
    'MAIS','MENOS','MULTIPLICACAO','DIVISAO','DOIS_PONTOS','VIRGULA','MENOR','MAIOR',
    'IGUAL','DIFERENTE','MENOR_IGUAL','MAIOR_IGUAL','E_LOGICO','OU_LOGICO','NEGACAO','ABRE_PARENTESE',
    'FECHA_PARENTESE','ABRE_COLCHETE','FECHA_COLCHETE','SE','ENTAO','SENAO','FIM','REPITA','ATE',
    'ATRIBUICAO','LEIA','ESCREVA','RETORNA','INTEIRO','FLUTUANTE','NUM_INTEIRO','NUM_PONTO_FLUTUANTE',
    'NUM_NOTACAO_CIENTIFICA','ID','COMENTARIO',
    )

    return tokens
```

Figura-4 Apresenta a lista de tokens definida na linguagem *Tpp.*

A segunda parte da implementação e a criação de expressões regulares ‘*Regex’* para ler linha a linha fazendo parsing e capturar os tokens, assim gerando uma lista de tokens no final, para cada token é preciso criar uma expressão específica que basicamente funciona como um autômato. Na Figura-5 iremos apresentar algumas expressões regulares simples para capturar palavras reservadas e gerar os tokens. Já na (Figura-6) apresenta algumas das expressões regulares complexas para identificar palavras reservadas.

```
t_FIM = r'fim'
t_SENAO = r'senão'
t_SE = r'(se)'
t_ATE = r'até'
t_LEIA = r'leia'
t_MAIS = r'\+'
t_MENOS = r'-'
```

Figura-5 Apresenta expressões regulares simples para gerar tokens de palavras reservadas.

```
def t_NUM_PONTO_FLUTUANTE(t):
    r'(-)?\d+\.\d*'
    t.value = float(t.value)
    return t

def t_NUM_INTEIRO(t):

    r'((?<=\D)[+-]\d+)|(^(?<=\D)[+-]\d+)|\d+'

    t.value = int(t.value)
    return t
```

Figura-6 Apresenta expressões regulares complexas para gerar tokens de palavras reservadas.

A entrada do programa é um arquivo do tipo tpp, que contém um código, para então a partir desse código o programa começar a fazer a análise e retirar os tokens. Na (Figura-7) iremos ilustrar um código simples na linguagem, logo em seguida na Figura-8 apresentamos a lista de tokens gerada pelo programa.

```
inteiro: n
flutuante: a[10]

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

inteiro principal()
    leia(n)
    escreva(fatorial(n))
    retorna(0)
fim
```

Figura-7 Ilustra um código Fatorial em *Tpp.*

```
INTEIRO
DOIS_PONTOS
ID
FLUTUANTE
DOIS_PONTOS
ID
ABRE_COLCHETE
NUM_INTEIRO
FECHA_COLCHETE
INTEIRO
ID
ABRE_PARENTESE
INTEIRO
DOIS_PONTOS
ID
FECHA_PARENTESE
INTEIRO
DOIS_PONTOS
ID
SE
ID
MAIOR
NUM_INTEIRO
```

Figura-8 Ilustra uma parte da lista de tokens gerada.

3. Python Yacc

O *'Yacc'* é software desenvolvido para computador no qual sua função é gerar um código de analisador Look Ahead da esquerda para a direita *LALR*, gerando um analisador *LALR* a parte de um compilador que tenta dar sentido sintático ao código-fonte com base em uma gramática formal, escrita em uma notação semelhante ao tokens, que juntamente com a parte lexa gerada na primeira parte do projeto, faz com que ao final da execução consiga nos dizer se o código que escrevemos está sintaticamente correto, caso esteja correto ele nos gera um árvore, que futuramente podemos utilizar para a geração de código que a máquina consiga interpretar.

3.2. Descrição da gramática

Para definir a gramática foi utilizado várias regras, para assim obter a identificação na árvore, como por exemplo a regra ‘lista_desclaracoes’, a partir desta regras, geramos um node pai na árvore que tem como filho vários parâmetros, que por si podem gerar mais regras e continuar gerando recursivamente ou podem ser estados terminais, como estamos em uma árvore podemos chamar esse estado de folha, a partir de uma condição da regra existem suas dependência que precisam existir para criar uma árvore sintaticamente correta, caso isso não ocorra haverá um node com a label de "ERRO" na árvore, assim possibilitando a leitura e análise do problema que ocasionou isso, para que no próximo analisador conseguirmos retornar um feedback para o usuário, em qual local ele errou, tirando essa regra que foi exemplificado existem mais 36 (trinta e seis), no qual se tem uma regra para cada caso possível existente a partir da lista de tokens, que foi gerado pelo analisador léxico, as regras e a árvore que estão sendo geradas pelo analisador sintático serão utilizados para identificar erros na próxima etapa do compilador, no qual conseguiremos a partir da árvore identificar erros e retornar o feedback para o usuário, com possíveis soluções pré-codificadas.

3.2. Definição das regras

Para o funcionamento do programa precisamos definir várias regras de como ele deve encontrar e tratar as expressões encontradas e juntamente a isso verifica se a expressão encontrada está correta ou não, as regras são como alguns padrões que ele deve identificar por exemplo a regra para identificar a expressão do *‘Repita’,* que é “ repita : REPITA corpo ATE expressao ”, para que essa expressão seja aprovada ele precisa achar o token *‘REPITA’,*em seguida o corpo dessa expressão, após isso deve encontrar um token com o nome de *‘ATE’,* com uma expressão na sequência que irá definir a condição de parada, caso tudo estiver correto então ele vai criar um nó pai na árvore chamado repita e com os filhos que são o ‘CORPO’,e o ‘ATE’, porém pelo fato de isso ser um recurso ele chamar o algoritmo novamente para o'que está no dentro do corpo para ver se o conteúdo contido lá dentro do escopo irá gerar mais nós pai ou acaba ali mesmo. Na (Figura-9) iremos ilustrar como foi definida a regra do repita no código do algoritmo.

```
def p_repita(p):
    """repita : REPITA corpo ATE expressao"""

    pai = MyNode(name='repita', type='REPITA')
    p[0] = pai

    filho1 = MyNode(name='REPITA', type='REPITA', parent=pai)
    filho_repita = MyNode(name=p[1], type='REPITA', parent=filho1)
    p[1] = filho1

    p[2].parent = pai  # corpo.

    filho3 = MyNode(name='ATE', type='ATE', parent=pai)
    filho_ate = MyNode(name=p[3], type='ATE', parent=filho3)
    p[3] = filho3

    p[4].parent = pai   # expressao.
```

Figura-9 Apresenta a regra criada no código do algoritmo de análise sintática.

3.3. Definição de tratamento de erro

Para o funcionamento das regras do programa precisamos definir vários tratamentos de erros para assim quando alguma regras não conseguir de match em todas suas parte, conseguir notificar o usuário que o código dele está com problema em algum lugar ou está faltando alguma coisa, no código implementado existe um tratador de erro para a maioria das regras, esses tratamento basicamente funciona capturando o no que foi gerado e analisa ele onde está ocorrendo o problema e notifica o usuário por exemplo ‘’repita : REPITA error ATE expressao”, nessa expressão estamos tentando capturar o erro que vai ser gerado caso o usuário esqueça ou coloque alguma coisa inválida no corpo da regra *‘Repita’,* caso isso aconteça o algoritmo vai notificar o usuário com a seguinte mensagem:’’Erro no corpo do loop: line/col:(x,y)”, assim o feedback para o usuário e instantaneamente, facilitando assim a correção do código. Na (Figura-10), iremos apresentar o código de tratamento de erro da regra *‘Repita’,* apresentando todos os casos que podem gerar erros.

```
def p_repita_error(p):
    """repita : REPITA error ATE expressao
              | REPITA corpo error expressao
    """

    error_line = p.lineno(1)

    indice = 0
    for i in p:
        indice += 1

    for i in range(indice):
        try:
            if p[i].type in tokens and i == 1:
                print("\nErro na definicao do loop: esperado 'REPITA' line/col:({},{})".format(error_line,(p[i].lexpos)))
            elif p[i].type in tokens and i == 2:
                print("\nErro no corpo do loop: line/col:({},{})".format(error_line,(p[i].lexpos)))
            else:
                print("\nErro na definicao do loop:  line/col:({},{})".format(error_line,(p[i].lexpos)))
        except:
            0

    father = MyNode(name='ERROR::{}'.format(error_line), type='ERROR')
    logging.error(
        "Syntax error parsing index rule at line {}".format(error_line))
    parser.errok()
    p[0] = father
```

Figura-10 Apresenta o tratamento de erros para a regra *‘Repita’* ,no código do algoritmo de análise sintática.

3.5. Exemplos de entrada e saída

Para a entrada do analisador sintático é utilizado a saída do analisador léxico, que contém uma lista de tokens geradas de um código que foi escrito em *tpp,* como é apresentado na (Figura-11)

Ao pegar os tokens geradas pelo lexer, juntamente com a biblioteca *Any Tree,* é gerado um árvore no qual cada nodo é gerado pelas regras da gramática,porém a árvore que é criada sai em formato de *Dot,* a partir disso usamos o auxílio de uma outra biblioteca chamada de *Graphviz,* conseguimos gerar imagens dessa árvore para facilitar a visualização, na subseção a seguir iremos explicar melhor a parte da geração na árvore .

```
('INTEIRO', 'DOIS_PONTOS', 'ID', 'INTEIRO', 'ID', 'ABRE_PARENTESE',
'INTEIRO', 'DOIS_PONTOS', 'ID', 'FECHA_PARENTESE', 'RETORNA',
 'ABRE_PARENTESE', 'ID', 'FECHA_PARENTESE', 'FIM')
```

Figura-11 Ilustra a saída do analisador léxico, que serve de entrada para o sintático.

Lembrando que todas essas tokens, são geradas a partir do analisador léxico, no qual todas as tokens são criadas a partir de regras feitas com regex, que extraem as tokens diretamente de um código de *tpp.*

3.5. Geração da árvore

Como foi mencionado acima cada regra gera um pai e filhos em um árvore, e cada filho de uma regra pode se tornar pai e gerar mais filhos, por se tratar de uma recursão o programa no final da execução do programa se tudo estiver correto, o algoritmo irá gerar um árvore com todos os e filhos, que futuramente usaremos essa árvore para a próxima análise, mas no momento podemos visualizar essa árvore utilizando uma biblioteca chamada de *'AnyTree'* juntamente com a biblioteca *'Graphviz',* no qual ela pega o arquivo gerado pelo algoritmo e cria uma imagem png dessa árvore, assim facilitando muito a leitura da do resultado na árvore, e também para verificar se possui algum tipo de erro nas regras. Na (Figura-12), apresentamos um código base para o analisador sintático gerar a árvore,e em seguida na (Figura-13), apresentamos a imagem da árvore gerada pelas bibliotecas utilizando como base a saída do algoritmo da análise sintática.

```
inteiro teste(inteiro: n)
    se n > 1 então
      retorna(0)
    fim
fim
```

Figura-12 Apresenta o código base utilizado para a geração da árvore no analisador sintático

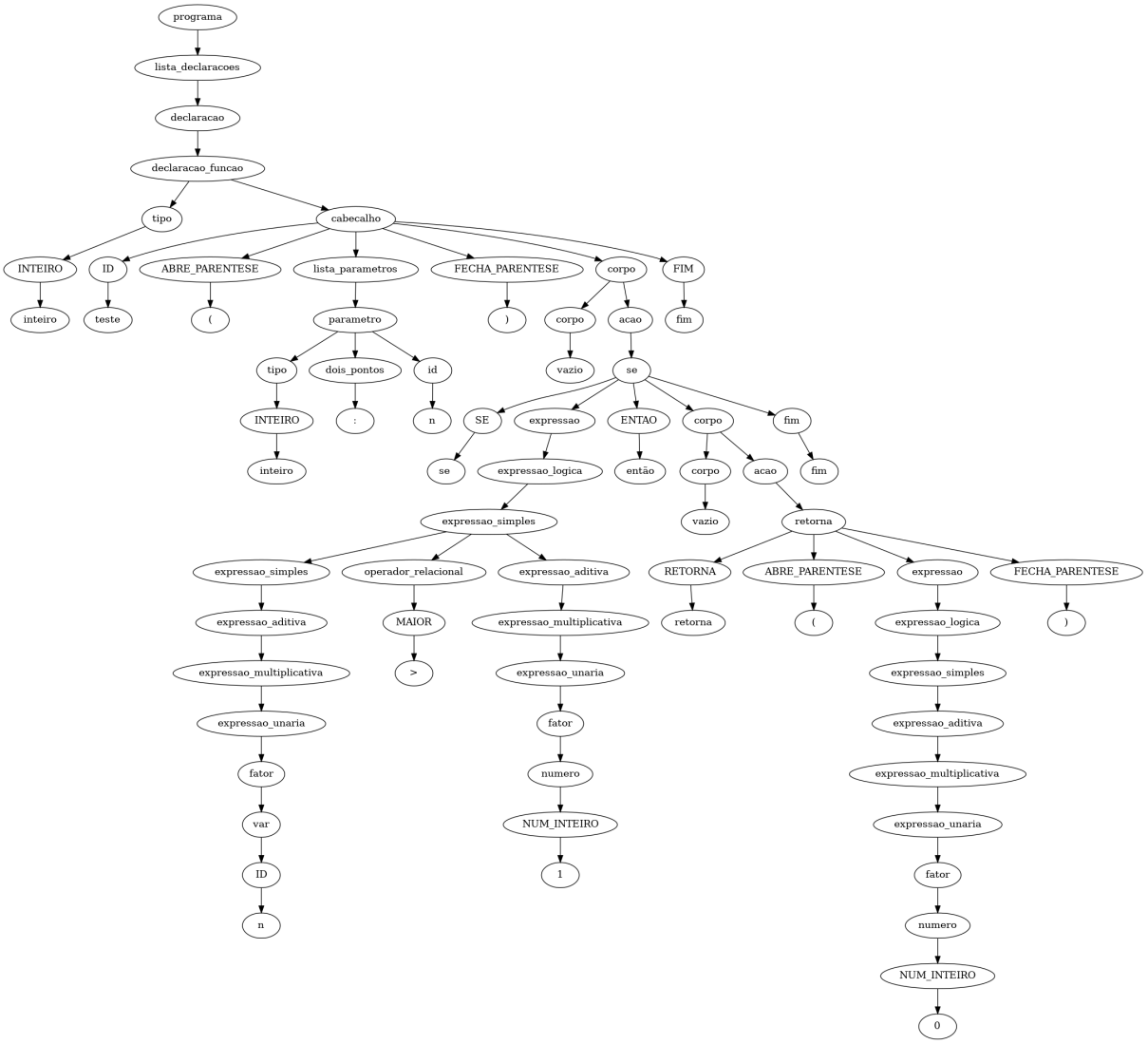

Figura-13 Apresenta a árvore gerada pela saída do analisador sintático.

5. Análise Semântica

A análise semântica e o último processo que fazemos para identificar erros em um código do compilador, nessa análise tentamos achar erros como se funções que foram declara de um tipo estão retornando valor deste tipo, ou se variáveis de um tipo estão sendo atribuídas a variáveis de outros tipos, entre várias outras verificações necessárias para assim descobrir se está um código limpo e possível de gerar o código interpretado pela máquina. A análise semântica trata a entrada sintática e transforma-a numa representação mais simples e mais adaptada à geração de código. Esta camada do compilador fica igualmente encarregada de analisar a utilização dos identificadores e de ligar cada uma delas à sua declaração. Nesta situação verificar-se se o programa respeita as regras de visibilidade e de porte dos identificadores. Além disso, é também esperado que no processo da compilação se verifique se cada expressão definida tem um tipo adequado conforme as regras próprias da linguagem.

5.1. Definição das regras

Para fazer a descrição da gramática primeiro percorremos a árvore resultante da análise sintática e adicionamos em uma lista os que são nodes país e os que são nodes terminais em outra lista, a partir disso começamos a fazer vários testes em nodes estratégicos para verificarmos se tem erros presente naquele determinado node como por exemplo se acharmos um node com o nome de ‘chamada_funcao’, quando encontramos esse node fazemos a primeiro verificação se a função chamada existe, caso não exista notificamos ao usuário, caso contrário continuamos para a próxima fase que é verificar se a quantidade de parâmetros passado é a quantidade esperada na definição da função, se a quantidade de parâmetros for maior ou menor do que a esperada, notificamos ao usuário, caso contrário novamente continuamos com a próxima verificação, que é se a função chamada está com um node de retorno, se não for encontrado um node com nome de 'retornar' na subárvore da função notificamos ao usuário novamente, caso contrário entramos na última verificação que é se a função foi declarada com um tipo específico como exemplo ‘inteiro’,e se o retorno dessa função é um valor inteiro, se não for notificamos ao usuário caso contrário acabamos as verificações desse node e o processo de análise semântica pode continuar para os demais nodes que são verificados atrás de encontrar erros semânticos.

5.2. Tabela de Símbolos

A partir do analisador semântico, conseguimos gerar um tabela de símbolos, que é construída durante a execução da verificação das regras no analisador, como quando ele verifica se funções existem ou se variáveis foram declaradas, entre alguns outros momentos, conseguimos analisar esse node e verificar se ele é eleito para ser adicionado na tabela de símbolos, para posteriormente saber quem é do mesmo escopo, ou quem é global, também conseguimos saber se o node é um variável ou um função juntamente com seu tipo, entre alguns outros detalhes que conseguimos capturar e adicionar na tabela de símbolos. Na (Figura-14) apresentamos a saída gerada no terminal, que nos mostra visualmente a tabela que foi gerada pelo analisador.

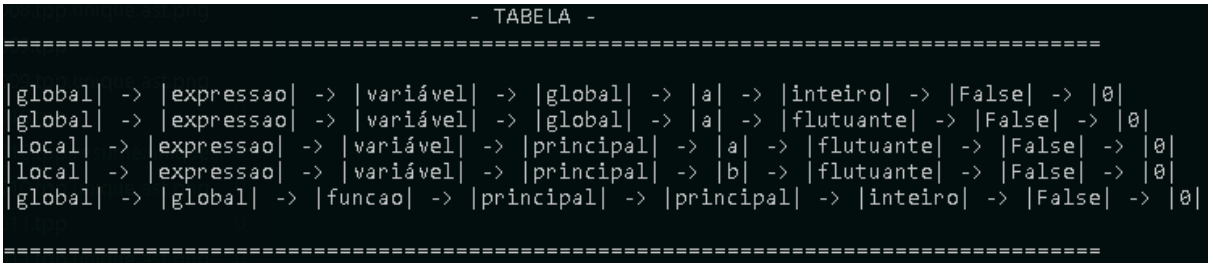

Figura-14 Apresenta a tabela de símbolos gerada pelo analisador semântico.

5.3. Árvore Abstrata

Ao concluirmos a análise semântica e gerar a tabela de símbolos apenas nos falta gerar uma árvore podada, ou seja um árvore no qual geralmente só se leva em conta os nodes terminais, gerando assim um árvore mais simples de se analisar e posteriormente utilizar para acelerar o processo de geração de código, no presente projeto não conseguimos ainda atingir a perfeição que essa árvore precisaria ter para realmente gerar um árvore limpa com tudo em seu devido lugar, mas em futuras correções nesta parte iremos refazer essa geração da árvore para assim conseguir criar um árvore mais limpa e possibilitar de usar ela no próximo passo que seria a geração de código de máquina. Na (Figura-15) Apresentamos a saída da árvore com a partir de uma linha código no caso : ‘a := 1+1’, que nos gera a seguinte árvore abstrata.

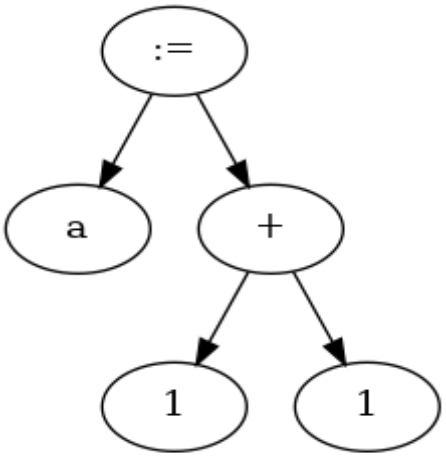

Figura-15 Apresenta a árvore abstrata gerada a partir do código: a := 1+1.

6. Geração de código

Para concluir o projeto proposto pelo curso de compiladores, só nos resta apresentar a parte de criação de código intermediário, e para entender o'que é isso, precisamos primeiro falar do llvm que é um conjunto de tecnologias de compilador e de ferramentas, que pode ser usado para desenvolver um front end para qualquer linguagem de programação e um back end para qualquer arquitetura de conjunto de instruções. O LLVM 'Low Level Virtual Machine' é projetado em torno de uma representação intermediária independente de linguagem (IR) que serve como uma linguagem assembly portátil de alto nível que pode ser otimizada com uma variedade de transformações em várias passagens, a partir disso para começar a gerar o código intermediário começamos a percorrer a árvore simplificada que foi resultado da análise semântica, nessa passagem pela árvore tentamos identificar padrões que servirão para gerar o código como, encontrar o node com o nome 'declaracao variável', no qual é gerado uma variável a partir desse node, que pode ser tanto uma variável local ou global dependendo de quem é seu pai na árvore, e assim por diante vários nodes são responsáveis por começar um bloco de instruções ou para finalizar um bloco de instruções.

6.1. Detalhes de implementação

Para conseguirmos a geração de código intermediários existe um passo a passo a ser seguido até lá no qual vamos explicar cada um deles começando pela biblioteca do *Python* chamada de llvm lite que é um binding simples e leve para escrever funcionalidades de um compilador, essa biblioteca possui várias funções prontas geração de código como variáveis, funções, *array*, condições, loop entre várias outras estruturas prontas para ser chamadas em um código em python para posteriormente criarem um código intermediário, assim em seguida é possível criar um arquivo compilado utilizando outra biblioteca chamando de *'Clang'*. Para criarmos o código intermediário precisamos instanciar a classe do *'IR',* que está dentro do llvm lite essa classe nos permite usar várias funções de criação de código intermediário como um dos principais o *'builder',* utilizando ele como construtor executamos a maioria dos códigos usando funções que viram de filhos da classe builder, ao final de toda a varredura da árvore semântica teremos um *módulo.bc* no formato de binário que utilizando a função do python de conversão para string teremos o arquivo *módulo.ll,* que já é nosso resultado final da geração de código, porém com a possibilidade de utilizando um criador de código executável como o clang para assim termos um arquivo executável para utilizar na máquina.

6.2. Exemplo de entrada

Para dar início na geração de código, primeiramente precisamos pegar a árvore simplificada que foi gerado pelo analisador semântico, pois se usarmos a árvore que o sintático gera iria ficar muito mais complicado andar pela árvore porque a árvore do sintático saio com muito ruído e poluída para vermos com clareza os nodes que realmente importa. Na Figura 17 iremos apresentar a imagem de uma árvore simples resultando de um algoritmo que será apresentado na Figura 16.

```
inteiro: a

inteiro principal()
  inteiro: b

  a := 10
  b := a

  retorna(b)
fim
```

Figura-16 Apresenta um código de exemplo para geração da árvore.

No código acima temos a declaração de uma variável ‘a’ do tipo inteiro no escopo global, logo após a uma declaração da função principal ‘main’, em seguida temos a declaração da variável ‘b’ no escopo local, para finalizar o corpo da função temos a atribuição do valor 10 a variável ‘a’ e a atribuição do ‘a’ na variável ‘b’, para acabar o bloco temos o retorna que vai gerar o escape do bloco básico para acabar o algoritmo.

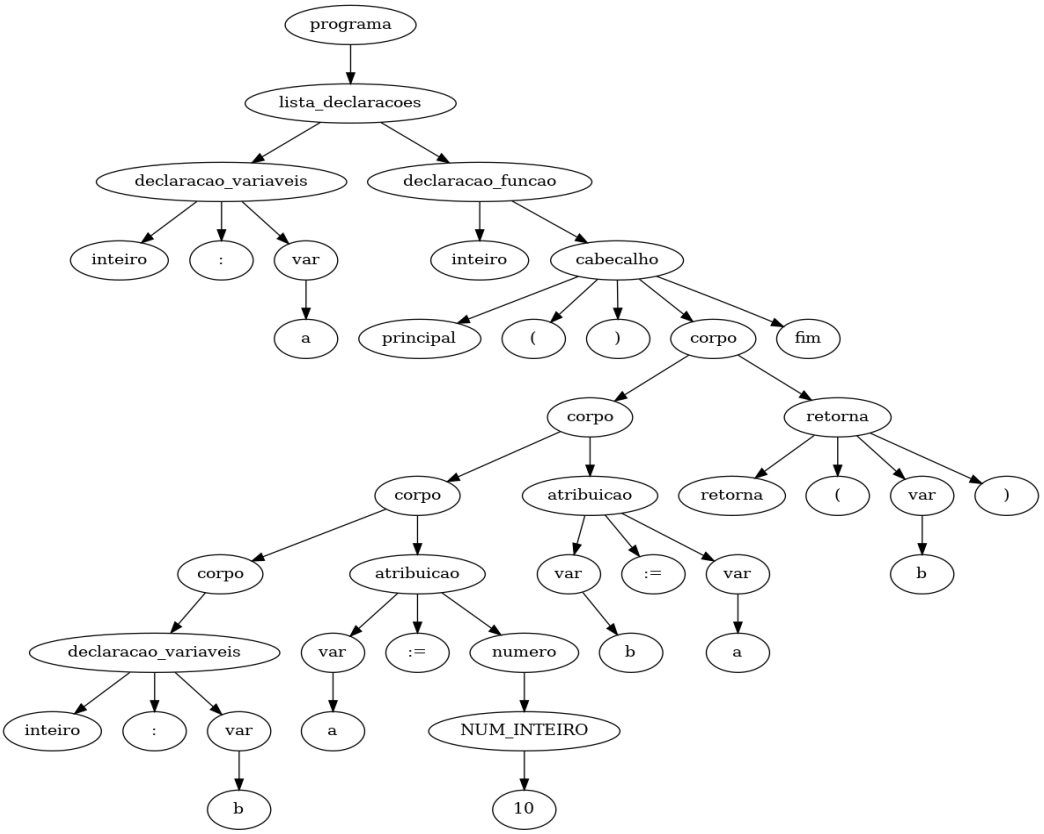

Figura-17 Apresenta a árvore abstrata gerada a partir do código da Figura 16.

Essa é a árvore que é resultante da análise semântica, no qual é feita uma poda para deixar mais clara e intuitiva, assim conseguimos gerar código a partir dela de uma maneira um pouco mais clara e simples.

6.3. Exemplo de saída

A partir de conseguirmos a árvore gerada pelo analisador semântico, começamos fazer a varredura de node a node, em busca dos padrões que precisamos, utilizando o código criado na última parte do projeto o ‘gerador_codigo.py’ para assim gerar o código intermediário, na Figura 18 iremos apresentar o resultado que foi obtido a partir da varredura da árvore usando o algoritmo junto com as regras do llvm lite.

```
; ModuleID = "module.bc"
target triple = "x86_64-pc-linux-gnu"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"

declare void @"escrevaInteiro"(i32 %".1")

declare void @"escrevaFlutuante"(float %".1")

declare i32 @"leiaInteiro"()

declare float @"leiaFlutuante"()

@"a" = common global i32 0, align 4
define i32 @"main"()
{
entry:
  %"retorno" = alloca i32, align 4
  store i32 0, i32* %"retorno"
  %"b" = alloca i32, align 4
  store i32 10, i32* @"a"
  %".4" = load i32, i32* @"a"
  store i32 %".4", i32* %"b"
  br label %"exit"
exit:
  %".7" = load i32, i32* %"b"
  store i32 %".7", i32* %"retorno"
  %"returnTemp" = load i32, i32* %"retorno", align 4
  ret i32 %"returnTemp"
}
```

Figura-18 Apresenta o código intermediário resultante da utilização ‘ gerador_código.py’ utilizando como base o código da figura 16.

O código acima é o resultado do algoritmo ‘gerador_codigo.py’, esse código resultante é gerador a partir das regras do llvm lite que cada uma delas nós era um linha apresentada acima que lembra muito um assembly porém é feito de uma maneira um pouco mais alto nível para fácil compreensão humana e também para acelerar o processo, com esse código resultante já é possível compilar ele com *‘Clang’* e assim gerar o .exe ou o arquivo executavel comumente chamado, mas como o objetivo do projeto era chegar até no passo de gerar um código intermediário, então acabamos com ele aqui.

7. Conclusão

Como conclusão, podemos afirma que tanto o analisador léxico, analisador sintático e o analisador semântico, juntamente com a última parte e mais importante a geração de códigos criados são funcional, pelo fato de funcionar como deveria em diversos teste em algoritmos diferentes, em todos os casos funcionou perfeitamente sem apresentar qualquer tipo de falha ou classificar um token incorretamente ou gerar um árvore incorretamente ou até mesmo passar por uma regra semântica sem ser detectado algum erro, então a partir disso afirmamos que foi concluído o objetivo do projeto que era a princípio criar essas quatro parte de um compilador, as três análises tanto a léxica como a sintática e semântica foram feitas com êxito e apresenta ótimos resultados, como a geração de código intermediário que também apresenta bons resultados em uma base de testes feitos.

8. References

Wikipedia Analizador Léxico ‘Lexicography’. Acesso em: 20 mar. 2020. disponível no link: <https://pt.wikipedia.org/wiki/L%C3%A9xico>
Wikipedia Analizador Semántico ‘Semantic Analysis'. Acesso em: 20 mar. 2021. disponível no link: <https://en.wikipedia.org/wiki/Compiler#Front_end>
Wikipedia Python Yacc ‘Yet Another Compiler-Compiler’. Acesso em: 20 mar 2021 disponivel no link: <https://en.wikipedia.org/wiki/Yacc>
Wikipedia LLVM ‘Low Level Virtual Machine’ . Acesso em 10 mai 2021 disponivel no link: https://en.wikipedia.org/wiki/LLVM
Dabeaz Python Py ‘Python Lex-Yacc’. Acesso em: 20 mar. 2021. disponível no link: <https://www.dabeaz.com/ply/>
Wikipedia Regex ‘Regular expression’. Acesso em: 21 mar. 2020. disponível no link: <https://en.wikipedia.org/wiki/Regular_expression>
JFLAP Autômato ‘Automaton Simulator’. Acesso em: 21 mar. 2020. disponível no link: <http://www.jflap.org/>.