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 the 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 parser was implemented, which, using the tokens generated by the first part together with the defined rules, manage to generate a program tree that we can use later to generate code that the computer can understand.*

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 para a geração de código que o computador possa entender.*

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.

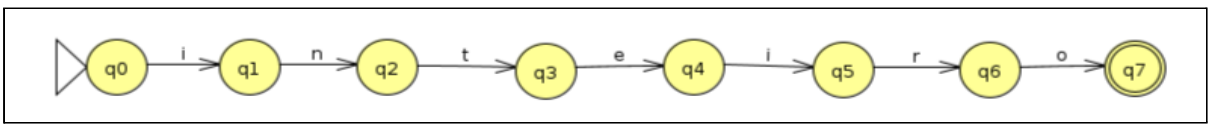

Figura-1 Apresenta o autômato que identifica o token *inteiro.*

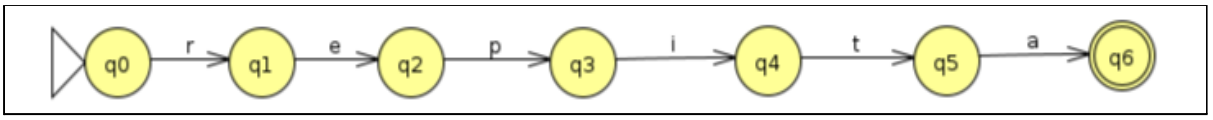

Figura-2 Apresenta o autômato que identifica o token *repita.*

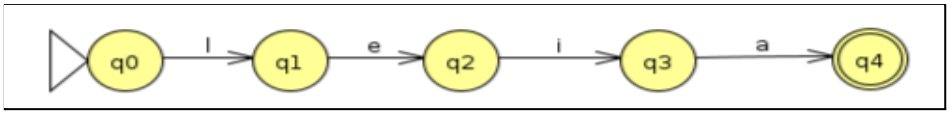

Figura-3 Apresenta o autômato que identifica o token de *leia.*

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.

2.3.1 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.

2.3.2 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.

3.1 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.

3.2 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.

3.3 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 11, apresentamos um código base para o analisador sintático gerar a árvore,e em seguida na figura 12, 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-11 Apresenta o código base utilizado para a geração da árvore no analisador sintático

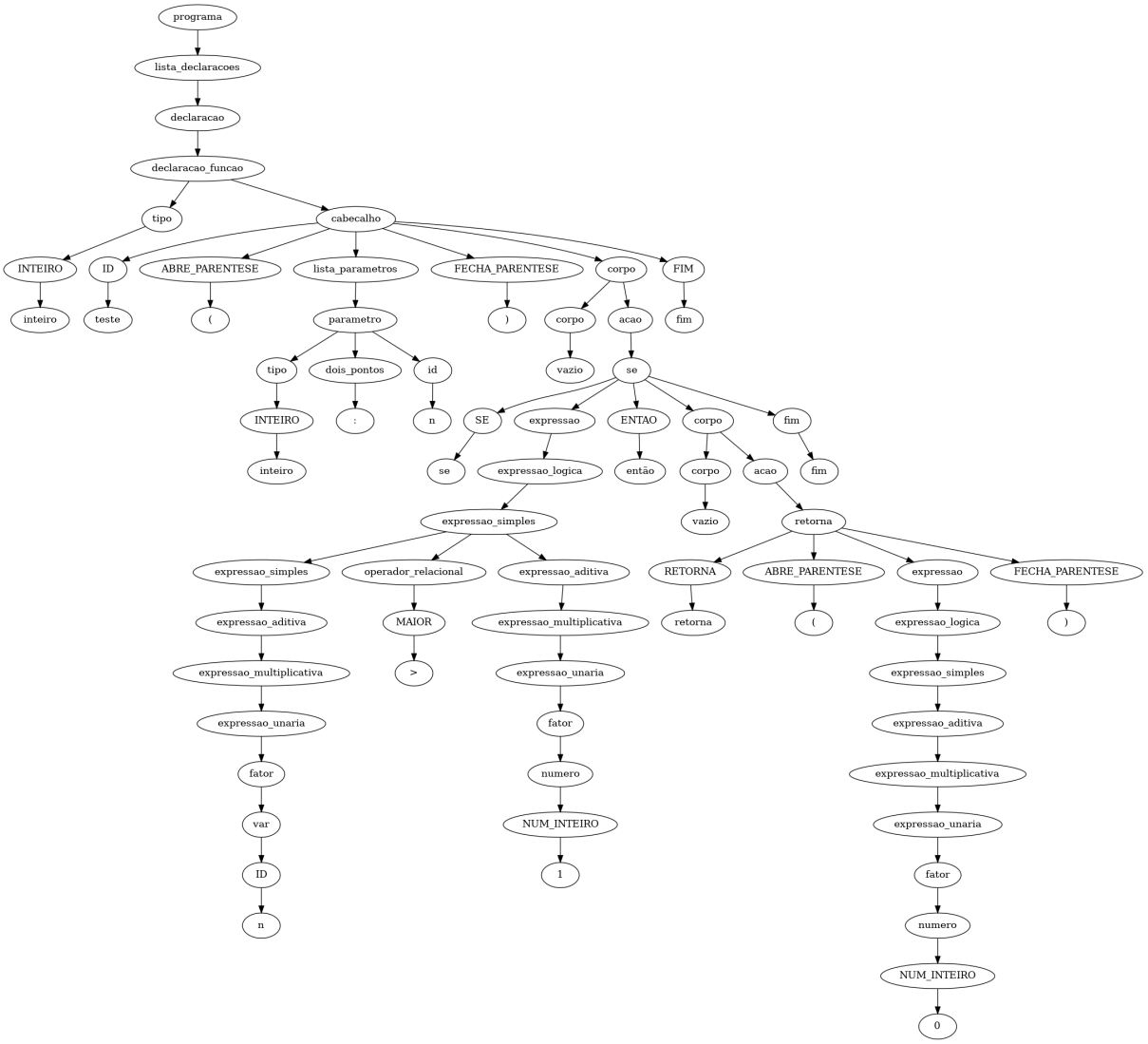

Figura-10 Apresenta a árvore gerada pela saída do analisador sintático.

4. Conclusão

Como conclusão, podemos afirma que tanto o analisador léxico e o analisador sintático criados até o momento são funcional pelo fato de funciona como deveria em diversos teste e algoritmos diferentes, em todos os casos funcionou perfeitamente sem apresentar qualquer tipo de falha ou classificar um token incorretamente ou gerar um árvore incorretamente, então a partir disso afirmamos que foi concluído o objetivo do projeto que era a princípio criar uma essas duas parte de um compilador, as duas análises tanto a léxica como a sintática foram feitas com êxito e apresenta ótimos resultados.

5. References

Wikipedia Léxico Disponível em: <https://pt.wikipedia.org/wiki/L%C3%A9xico>.Acesso em: 20 mar. 2020.
Wikipedia Python Yacc Disponível em: <https://en.wikipedia.org/wiki/Yacc> .Acesso em: 28 mar 2021.
Dabeaz PythonPly Disponível em: <https://www.dabeaz.com/ply/>.Acesso em: 20 mar. 2020.
Wikipedia Regex Disponível em: <https://en.wikipedia.org/wiki/Regular_expression>.Acesso em: 20 mar. 2020.
JFLAP Autômato Disponível em: <http://www.jflap.org/>.Acesso em: 20 mar. 2020.