Projeto de Implementação de um Compilador para a linguagem TPP: Análise Léxica

(Trabalho - Parte1)

Pedro Acácio Rodrigues[1]

[1]Universidade Tecnológica Federal do Paraná - Campo Mourão (UTFPR-CM)

[2]Departamento de Computacao (DACOM)

pedrorodrigues.2019@alunos.utfpr.edu.br

Abstract. *This article aims to present the path taken and results obtained during the lexical analysis phase of the project to implement a compiler for the TPP language*

Resumo. *Esse artigo tem o objetivo de apresentar o caminho percorrido e resultados obtidos durante a fase de análise léxica do projeto de implementação de um compilador para a linguagem TPP.*

1. Introdução

O processo de implementação de um compilador para uma linguagem pode ser dividido em 4 partes, sendo elas: análise léxica, análise sintática, análise semântica e geração de código. Cada uma possui seus métodos de resolverem os problemas propostos em fases para alcançar o objetivo final do compilador.

Com foco na análise léxica, assume-se que nela é buscado dar significado aos símbolos que o código compilado irá possuir, assim para reconhecê-los são referenciados a tokens. Tais elementos são de suma importância visto que em fases posteriores serão utilizados para propriamente definir como o compilador irá agir sintaticamente.

Para referenciar esses elementos utilizados no código a tokens é necessário primeiramente localizá-los. Nesse contexto são realizadas diversas verificações que buscam por meio de expressões regulares encontrar esses elementos léxicos para de certa forma rotulá-los com autômatos finitos definindo os tokens esperados. Assim, para que o código possa funcionar na linguagem é necessário que cada elemento seja passível de associação a algum token.

1.1. Especificação da Linguagem T++

A linguagem TPP se trata de uma evolução da Tiny que é desenvolvida no livro "Compiladores: princípios e práticas" de Kenneth C. Louden.

Sobre seu suporte, o TPP deve possuir desenvolvido tipos inteiros e flutuantes, arranjos uni e bidimensionais, laços de repetição, estruturas de tomada de decisão, variáveis locais e globais devem ter um dos tipos especificados, tipos de funções podem ser omitidos (quando omitidos viram um procedimento e um tipo void e devolvido explicitamente, linguagem quase fortemente tipificada: nem todos os erros são

especificados mas sempre deve ocorrer avisos, operadores aritméticos: +, -, * e /, operadores lógicos: e (&&), ou (||) e não (!), operador de atribuição: recebe (:=), operadores de comparação: maior (>), maior igual (>=), menor (<), menor igual (<=), igual (=).

2. Autômatos

Autômatos representam conjuntos de estados e transições, e é utilizado para descrição e análise de linguagens formais, incluindo linguagens de programação e expressões regulares. Levando em conta a definição de autómatos, para essa fase foram definidas expressões regulares utilizando Regex para localizar as palavras buscadas e rotulá-las utilizando autómatos.

2.1. Dígitos

Token	Expressão Regular
digito	**‘[0-9]’**

Chart 1. Regex para Dígitos

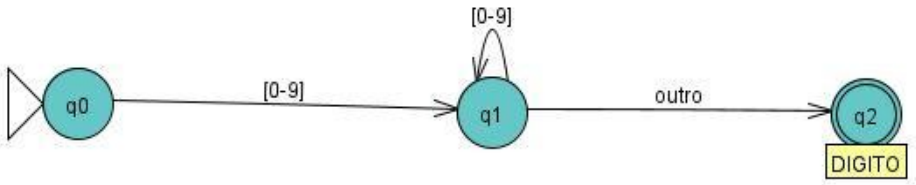

Figure 1. Autômato para Dígitos

2.2. ID

Token	Expressão Regular
ID	**‘(([a-zA-ZáÁãÃàÀéÉíÍóÓõÕ])(([0-9]) | _ | ([a-zA-ZáÁãÃàÀéÉíÍóÓõÕ]))’**

Chart 2. Regex para ID

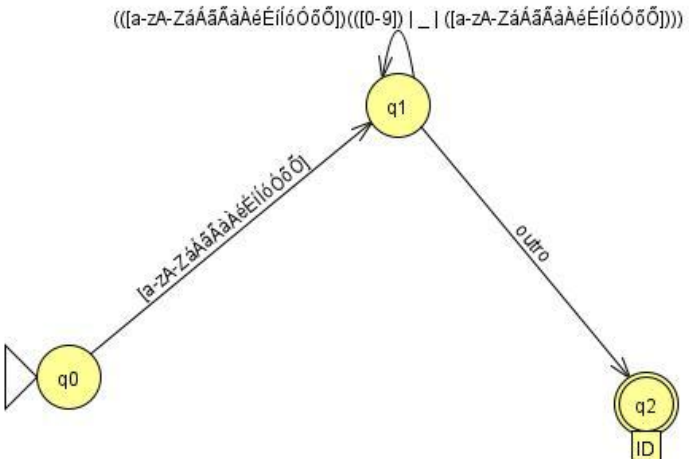

Figure 2. Autômato para ID

2.3. Ponto Flutuante

Token	Expressão Regular
NUM_PONTO_FLUTUANTE	'\d+[eE][-+]?\d+ | (\.\d+ | \d+ . \d*)([eE][-+]?\d+)?'

Chart 3. Regex para Ponto Flutuante

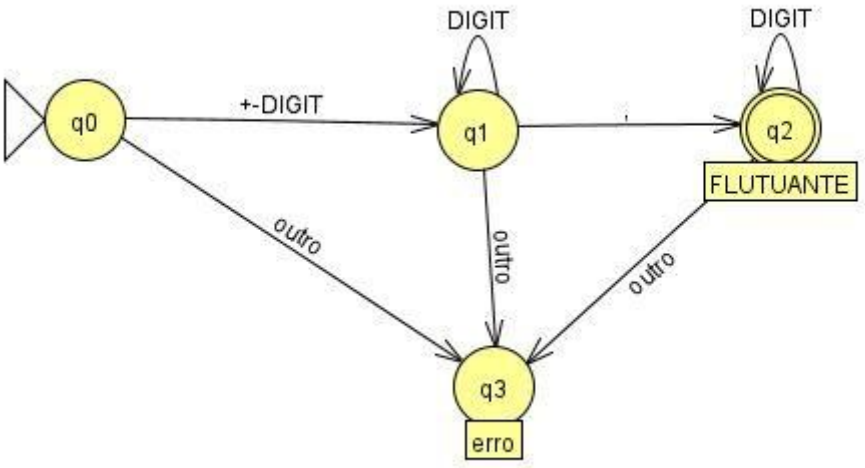

Figure 3. Autômato para Ponto Flutuante

2.4. Tipo Simples

Token	Expressão Regular
MAIS	**‘+’**
MENOS	**‘-’**
VEZES	**‘*’**
DIVIDE	**‘/’**
DOIS_PONTOS	**‘:’**
VIRGULA	**‘,’**
ABRE_PARENTESE	**‘(‘**
FECHA_PARENTESE	**‘)’**
ABRE_COLCHETE	**‘[‘**
FECHA_COLCHETE	**‘]’**
ATRIBUICAO	**‘:=’**

Chart 4. Regex para Tipos Simples

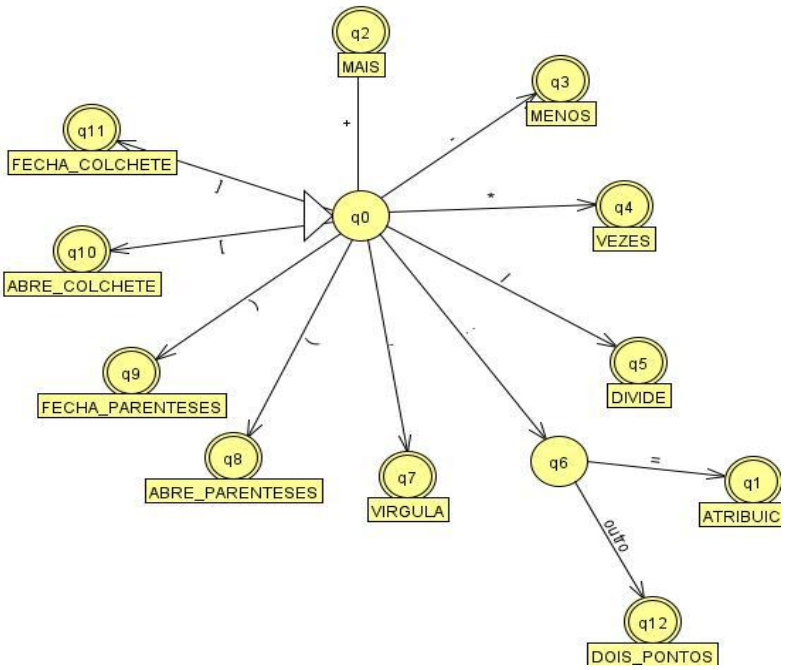

Figure 4. Autômato para Tipos Simples

2.5. Tipo Lógico

Token	Expressão Regular
E	**‘&&’**
OU	**‘||’**
NAO	**‘!’**

Chart 5. Regex para Tipo Lógico

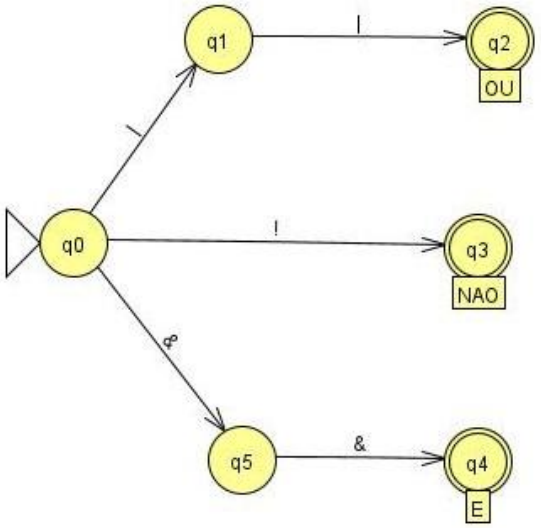

Figure 5. Autômato para Tipo Lógico

2.6. Tipo Relacional

Token	Expressão Regular
MENOR	**‘<’**
MAIOR	**‘>’**
IGUAL	**‘=’**
DIFERENTE	**‘<>’**
MENOR_IGUAL	**‘<=’**
MAIOR_IGUAL	**‘>=’**

Chart 6. Regex para Tipo Relacional

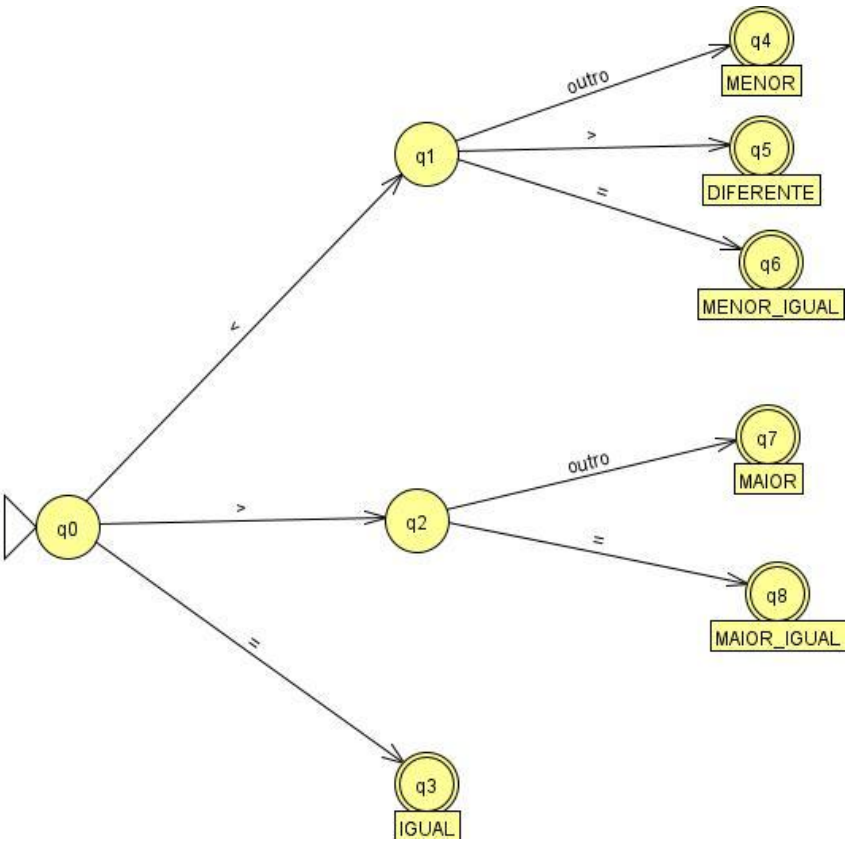

Figure 6. Autômato para Tipo Relacional

3. Implementação

3.1. PLY

Para a implementação do scan léxico foi utilizado o PLY (Python Lex-Yacc) que é uma biblioteca que possui ferramentas usadas na análise léxica. O PLY por ser muito parecido com o FLex em sua estrutura, leva seu arquivo de entrada semelhante que deve possuir as definições, regras e rotinas auxiliares

3.2. Definições

Nesse bloco do arquivo de entrada iremos definir pontos importantes a respeito da léxica a ser desenvolvida, como os tokens de linguagem e expressões regulares.

Para a definição dos mesmos, são utilizados os tokens e regex do item 2.

```
tokens = [
    "ID",
    ..........................
    "NUM_INTEIRO",
]
```

Código 1. Definição do Vetor de Tokens

```
reserved_words = {
    "se": "SE",
    "então": "ENTAO",
    ……
    "repita": "REPITA",
    "flutuante": "F”
]
```

Código 2. Definição do Vetor de Palavras Reservadas

```
digito = r"([0-9])"
letra = r"([a-zA-ZáÁãÃàÀéÉíÍóÓõÕ])"
sinal = r"([\-\+]?)"
id = (
    r"(" + letra + r"(" + digito + r"+|_|" + letra + r")*)"
)
```

Código 3. Definição do Regex

No PLY deve-se utilizar o prefixo “t_” seguido do nome exato do token para associá-lo.

```
t_MAIS = r'\+'
t_MENOS = r'-'
t_VEZES = r'\*'
t_DIVIDE = r'/'
t_ABRE_PARENTESE = r'\('
t_FECHA_PARENTESE = r'\)'
```

Código 4. Utilização do Prefixo “t_”

3.2. Regras

Nessa parte é definido como o autômato irá se comportar. No código abaixo é definido que caso a palavra não seja um item reservado, então será necessariamente um ID:

```
@TOKEN(id)
def t_ID(token):
    token.type = reserved_words.get(token.value, "ID")
    return token
```

Código 5. Rotulando elemento com “t_ID”

No próximo exemplo, vemos a principal estrutura dessa parte, pois nela associamos o token ao regex e se verificado válido a expressão regular, então o termo receberá o rótulo do token NUM_INTEIRO.

```
@TOKEN(inteiro)
def t_NUM_INTEIRO(token):
    return token
```

Código 6. Estrutura básica

4. Testes

4.1. Exemplo Utilizado

Para o teste realizado foi utilizado o programa principal que realiza operações matemáticas simples utilizando ponto flutuante.

```
inteiro principal()
  a :=+1
  c := a + b
  b:= 3 + a
  c:= +3
  a:= +3.5
  a := 3.5 + 4.5
fim
```

Código 7. Exemplo em T++

4.2. Execução

Para a rodar o teste e verificar a léxica do exemplo o comando a ser utilizado é:

```
python caminho/do/Lex.py caminho/do/teste.tpp
```

Código 8. Comando de Execução do Teste

4.3. Resultado

No caso para o comando:

```
python tpplex.py ./tests/teste-003.tpp
```

Código 9. Comando Utilizado

Foi obtido o seguinte retorno, mostrando que o código não possui erros léxicos em sua elaboração.

```
INTEIRO

ID

ABRE_PARENTESE

FECHA_PARENTESE

ID

ATRIBUICAO

MAIS

NUM_INTEIRO

ID

ATRIBUICAO

ID

MAIS

ID

ID

ATRIBUICAO

NUM_INTEIRO

MAIS

ID

ID

ATRIBUICAO

MAIS

NUM_INTEIRO
```

```
ID

ATRIBUICAO

MAIS

NUM_PONTO_FLUTUANTE

ID

ATRIBUICAO

NUM_PONTO_FLUTUANTE

MAIS

NUM_PONTO_FLUTUANTE

FIM
```

Código 10. Retorno Obtido pelo Testes

5. Conclusão

Durante os processos necessários para se conseguir analisar o código de forma léxica é possível concluir a importância dos autómatos e expressões regulares com objetivo de se entender o que antes era um arquivo de texto sem sentido, agora é passível de ser classificado e dado sentido aos elementos dele.

References

Louden, K. C. (2004). In Learning, C. editor, “Compiladores: princípios e práticas". 1th edition.

Beazley, D. (2007). “Writing parsers compilers with PLY”.

Beazley, D. (2007). “PLY (Python Lex-Yacc)”.