Projeto de Implementação de um Compilador para a Linguagem T++
Análise Léxica (Trabalho – 1ª parte)

Rafael Menezes Barboza[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Via Rosalina Maria dos Santos, 1233
CEP: 87301-899, Campo Mourão – PR – Brasil

ra29fa@gmail.com

Resumo. *O seguinte trabalho faz parte da disciplina de Compiladores do curso de ciência da computação, é a primeira parte de total de quatro etapas que ao final resultarão no estudo e implementação de compilador para a linguagem T++. Esta primeira abordagem pretende explorar o funcionamento do Analisador Léxico, um dos componentes principais para separar todo código fonte em pequenos "tokens"devidamente classificados e reconhecidos pela linguagem.*

1. Introdução

Um compilador de forma simplificada nada mais é do que um tradutor de um código fonte em uma linguagem de programação de alto nível para uma linguagem de programação de baixo nível. As linguagens de alto nível são muito utilizadas no desenvolvimento de aplicações, porém a maioria dos computadores processam instruções mais simples e elementares e o compilador tem exatamente este papel de transformar o código para algo mais próximo da linguagem da máquina.

O processo de compilação é dividido em várias etapas bem definidas, cada etapa gera uma saída de dados correspondente a entrada da próxima etapa a ser executada, se assemelhando aos famosos Pipelines. Cada etapa deve ser encarregada de tratar possíveis erros ou notificá-los antes de passar para as próximas fases.

2. Linguagem de Estudo T++

A linguagem de programação escolhida para o trabalho foi a T++ linguagem essa inventada para fins acadêmicos e não é utilizada em meios comerciais. A linguagem T++ se assemelha bastante com as principais linguagem de programação compiladas do mercado por conter uma lógica de programação muito bem estruturada e familiar porém com algumas certas limitações.

Se trata de uma linguagem considerada quase fortemente tipada pois toda e qualquer variável criada deve ser atribuída a algum tipo de dado reconhecido pela linguagem e este deve ser persistente durante toda a execução do código. Por se tratar de uma linguagem quase fortemente tipada nem todos os erros são especificados mas sempre deve ocorrer avisos. O código pode ser estruturado por meio de funções que se relacionam podendo ou não retornar algum valor, no caso de não retornar nada assumimos que a função em questão é do tipo void.

A linguagem conta com dois tipos de dados básicos sendo eles “inteiro” e “flutuante” suporta também estruturas de dados unidimensional (array) e bidimensional (matriz). Com tudo isso é possível realizar uma série de operações aritméticas e lógicas, assim com em outras linguagem como por exemplo C/C++.

A Tabela 1 apresenta os principais símbolos e palavras reservadas suportadas pela linguagem T++, nesta etapa de análise léxica iremos nos referir a cada item da tabela como Tokens. Porém ainda podem ser definidos como tokens, números com um ou mais dígitos que podem ser inteiro, flutuante ou notação científica, identificadores que começam com uma letra e precedem com N letras e números e por fim comentários cercados de chaves {...} podendo conter uma ou mais linhas. Todos estes tokens posteriormente serão devidamente relacionados gerando assim a toda a logica do codigo fonte.

palavras reservadas	símbolos
`se`	+ soma
`então`	- subtração
`senão`	* multiplicação
`fim`	/ divisão
`repita`	= igualdade
`flutuante`	, vírgula
`retorna`	:= atribuição
`até`	< menor
`leia`	> maior
`escreva`	<= menor-igual
`inteiro`	>= maior-igual
	(abre-par
	(fecha-par
	: dois-pontos
	[abre-col
	] fecha-col
	&& e-logico
	|| ou-logico
	! negação

Tabela 1. Tokens da linguagem T++

O código fonte da figura 1 em T++ calcula o fatorial de um número passado pelo input do teclado, é possível visualizar o uso de algumas palavras reservadas e símbolos em ação no código. É notório que algumas palavras reservadas como “se, repita, senão ...” estão sempre aninhadas da palavra reservada “fim” que denota o fechamento do bloco, função está realizada pelo abre e fecha chaves na linguagem C/C++.

3. Analise Léxica

Análise léxica é a primeira fase do processo de compilação de um código fonte, nessa etapa o componente léxico realiza a varredura pelo código fonte carácter por carácter

```
inteiro principal()
    inteiro: digitado
    inteiro: i
    i := 1
    repita
        flutuante: f
        inteiro: int
        flutuante: resultado
        f := i/2.0
        int := i/2
        resultado := f - int

        se  resultado > 0
            escreva (i)
        fim
        i := i+1
    até i <= digitado
fim
```

Figura 1. Fatorial em T++

coletando as palavras e separando em conjuntos de marcas (tokens) que podem ser manipulados de forma mais fácil por um parser leitor de saída.

São reconhecidas as palavras reservadas, identificadores, constantes, símbolos entre outras palavras da linguagem em questão. Também é nessa etapa que é tratado espaços em branco, contagem das linhas e eliminação de comentários simplificando a saída para os próximos componentes do processo.

Cada conjunto de caracteres ou palavra é reconhecida por uma regra, no seguinte trabalho as regras foram elaboradas por meio de expressões regulares bem definidas que realizam a filtragem e separação de cada token com o seu devido valor.

3.1. Expressão Regular e Linguagens Formais

Para esse tipo de tarefa é de extrema importância a percepção da necessidade da utilização de linguagens formais para validar os tokens corretamente. A linguagem natural apesar de muito usual em diversos aspectos ela não é recomendada processos validativos pois pode trazer ambiguidade e uma complexidade ainda maior ao problema. Por isso a escolha de uma linguagem formal que seja simples, concisa, clara e sem ambiguidade é tão importante.

Por meio das expressões regulares é possível determinar uma cadeia de caracteres específicos para ser reconhecido, um exemplo é a expressão regular *r'senão'* para expressão só será reconhecido palavras com a exata ordem de caracteres determinado. Para reconhecer palavras de forma menos engessada pode ser usado um range de caracteres como por exemplo *r'[A-Z]'* que reconhece qualquer caractere do alfabeto (maiúsculo) que esteja dentro do range definido. Também é possível definir expressões que aceitam repetição de caracteres, números ou símbolos, um exemplo é a expressão regular *r'[A-Za-z]+[0-9_]*'* que pode aceitar palavras que comece com pelo menos um carácter maiúsculo ou minusculo e nenhum ou vários caracteres numéricos de 0 a 9 podendo haver o símbolo underscore ou não, aceitando palavras como *[rafa, RAFAEL, R, a56_, w_, ...]*.

3.2. Autômatos Finitos

Através de autômatos finitos podemos representar o funcionamento de expressões regulares visualizando os estados de aceitação e transições. Nesta sessão serão apresentados os autômatos finitos referentes às expressões regulares implementadas no analisador léxico, é possível velas com mais detalhes na seção 4 deste documento.

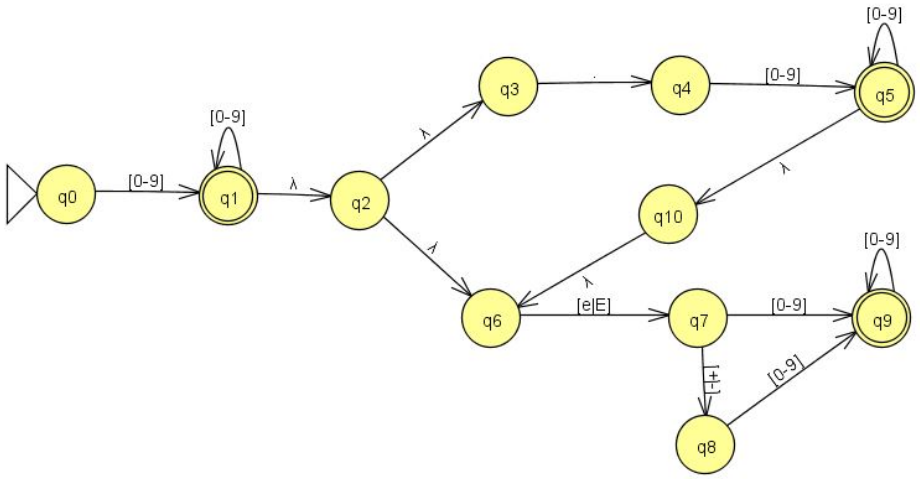

Figura 2. Reconhece números (Inteiros, Ponto Flutuante e Notação Cientifica)

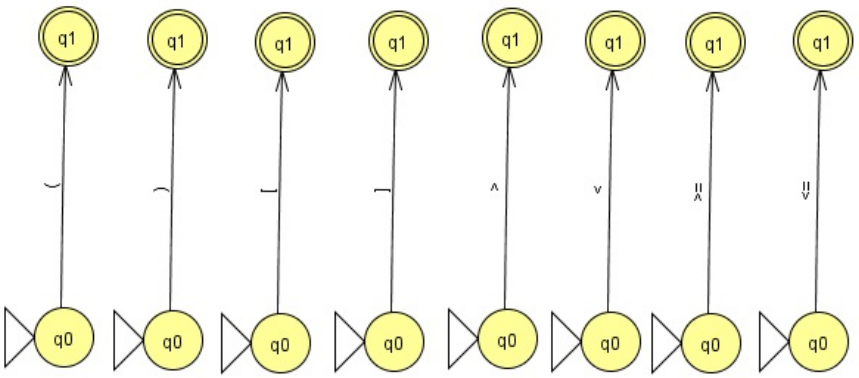

Figura 3. Reconhece operadores aritméticos

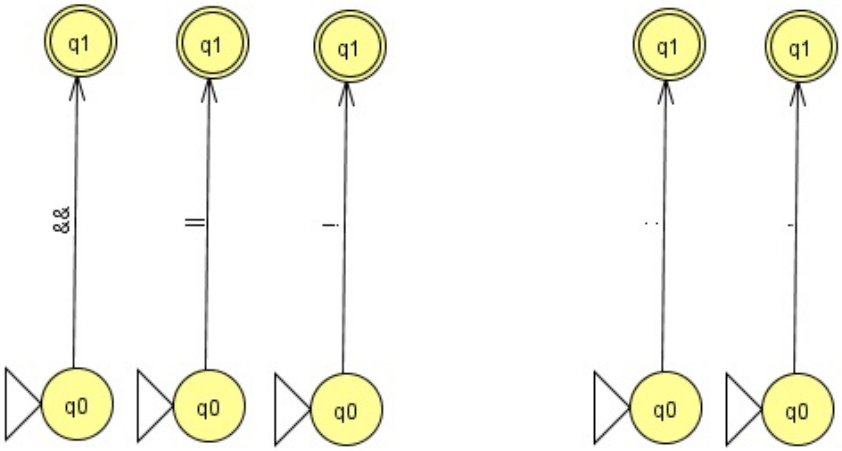

Figura 4. Reconhece símbolos e caracteres especiais da linguagem

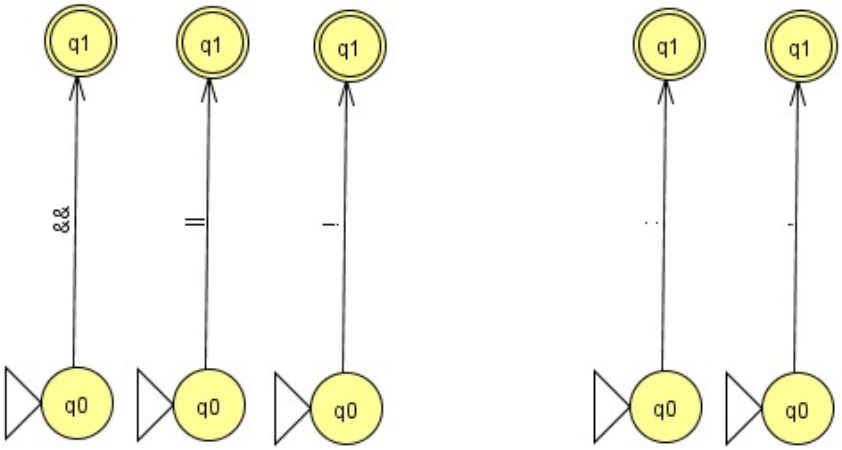

Figura 5. Reconhece Operadores lógicos e pontuações

4. Detalhes da implementação

4.1. Ferramentas auxiliares

Para a implementação e realização deste trabalho algumas ferramentas auxiliares foram utilizadas. As sessões seguintes descrevem um pequeno resumo das ferramentas.

4.1.1. Python

A linguagem de programação usada para a implementação deste projeto foi a linguagem python pois oferece uma gama maior de ferramentas de fácil acesso facilitando a implementação.

4.1.2. PLY

A ferramenta PLY se trata de uma implementação do analisador léxico (lex) e (yacc), com esta ferramenta é possível junto com a linguagem de programação Python definir regras de precedências, gramáticas e etc.

4.2. Implementação

Para a implementação do analisador léxico foi necessário definir tokens e palavras reservadas da linguagem T++. Estes foram definidos em duas listas presentes no código nas linhas [3 a 49], assim que definidos foi necessário definir para cada token uma expressão regular para que fosse reconhecido, podemos verificar isto nas linhas [51 a 87]. Para as palavras reservadas do dicionário RESERVED não foi necessário definir as expressões regulares pois o próprio dicionário de palavras é capaz de mapear isto.

Como o analisador léxico é responsável por notificar possíveis erros e ignorar comentários no código, foi implementado duas funções presentes nas linhas [89 a 98] que realizam esta função respectivamente.

As linhas [100 a 108] apenas realizam a chamada da função lex(), a abertura do arquivo a ser analisado e o processo de impressão dos tokens.

```
1  import ply.lex as lex
2  import sys
3
4  RESERVED = {
5             'se': 'SE',
6             'senão':'SENAO',
7             'então':'ENTAO',
8             'repita':'REPITA',
9             'escreva':'ESCREVA',
10            'leia':'LEIA',
11            'até':'ATE',
12            'fim':'FIM',
13            'inteiro' : 'INTEIRO',
14            'flutuante' : 'FLUTUANTE',
15            'retorna' : 'RETORNA',
16 }
17 tokens = ['ID',
18            'DOIS_PONTOS',
19            'ATRIBUICAO',
20            'MENOR',
21            'MAIOR',
22            'MENOR_IGUAL',
23            'MAIOR_IGUAL',
24            'MULT',
25            'VIRGULA',
26            'MAIS',
27            'MENOS',
28            'DIVIDE',
29            'ABRE_PAREN',
30            'FECHA_PAREN',
31            'ABRE_COUCH',
32            'FECHA_COUCH',
33            'SE',
34            'SENAO',
35            'ENTAO',
36            'REPITA',
37            'ESCREVA',
38            'LEIA',
39            'ATE',
40            'FIM',
41            'INTEIRO',
```

```
            'FLUTUANTE',
            'COMPARACAO',
            'NUMERO',
            'RETORNA',
            'E_LOGICO',
            'OU_LOGICO',
            'NEGACAO',
            'QUEBRA_LINHA',
            ]

t_ignore = " \t\n"

##operadores
t_MAIS = r'\+'
t_MENOS = r'-'
t_MULT = r'\*'
t_DIVIDE = r'/'
t_COMPARACAO = r'='

##símbolos
t_ABRE_PAREN = r'\('
t_FECHA_PAREN = r'\)'
t_ABRE_COUCH = r'\['
t_FECHA_COUCH = r'\]'
t_MENOR = r'<'
t_MAIOR = r'>'
t_ATRIBUICAO = r':='
t_MENOR_IGUAL = r'<='
t_MAIOR_IGUAL = r'>='

##operadores lógicos
t_E_LOGICO = r'&&'
t_OU_LOGICO = r'\|\|'
t_NEGACAO = r'!'

##pontos
t_DOIS_PONTOS = r':'
t_VIRGULA = r','

##números
t_NUMERO = r'([0-9]+)(\.[0-9]+)?([e|E][+|-]?[0-9]+)?'

##identificadores
def t_ID(t):
    r'[A-Za-z_][\w]*'
    t.type = RESERVED.get(t.value, 'ID')
    return t

##comentários
def t_comment(t):
    r'\{[^}]*[^{]*\}'
    pass

##tratamento de erros
def t_error(t):
    print("Erro", t.value[0])
    raise SyntaxError("ERRO", t.value[0])
    t.lexer.skip(1)

lex.lex()

file = open(sys.argv[1], "r", encoding="utf-8")
lex.input(file.read())
while True:
    tok = lex.token()
    if not tok: break
    print( tok.type, ':', tok.value )
```

5. Exemplos de saída do Analisador Léxico

Nesta seção serão apresentados alguns exemplos de saída quando o analisador léxico é submetido a códigos testes. A função responsável por imprimir a saída foi manipulada para assumir uma estrutura mais simples e facilitar o entendimento seguindo sempre a lógica de " TOKEN : valor ".

```
inteiro principal()
    inteiro: digitado
    inteiro: i
    i := 1
    repita
        flutuante: f
        inteiro: int
        flutuante: resultado
        f := i/2.0
        int := i/2
        resultado := f - int

        se  resultado > 0
            escreva (i)
        fim
        i := i+1
    até i <= digitado
fim
```

Figura 6. Verifica se é numero primo em T++ (ENTRADA)

```
1  INTEIRO : inteiro
2  ID : principal
3  ABRE_PAREN : (
4  FECHA_PAREN : )
5  INTEIRO : inteiro
6  DOIS_PONTOS : :
7  ID : digitado
8  INTEIRO : inteiro
9  DOIS_PONTOS : :
10 ID : i
11 ID : i
12 ATRIBUICAO : :=
13 NUMERO : 1
14 REPITA : repita
15 FLUTUANTE : flutuante
16 DOIS_PONTOS : :
17 ID : f
18 INTEIRO : inteiro
19 DOIS_PONTOS : :
20 ID : int
21 FLUTUANTE : flutuante
22 DOIS_PONTOS : :
23 ID : resultado
24 ID : f
25 ATRIBUICAO : :=
26 ID : i
27 DIVIDE : /
28 NUMERO : 2.0
29 ID : int
30 ATRIBUICAO : :=
31 ID : i
32 DIVIDE : /
33 NUMERO : 2
34 ID : resultado
35 ATRIBUICAO : :=
36 ID : f
37 MENOS : -
38 ID : int
39 SE : se
40 ID : resultado
41 MAIOR : >
42 NUMERO : 0
43 ESCREVA : escreva
44 ABRE_PAREN : (
45 ID : i
46 FECHA_PAREN : )
47 FIM : fim
48 ID : i
49 ATRIBUICAO : :=
50 ID : i
51 MAIS : +
52 NUMERO : 1
53 ATE : até
54 ID : i
55 MENOR_IGUAL : <=
56 ID : digitado
57 FIM : fim
```

Figura 7. Verifica se é numero primo em T++ (SAIDA)

```
inteiro: vet[10]
inteiro: tam

tam := 10

{ preenche o vetor no pior caso }
preencheVetor()
  inteiro: i
  inteiro: j
  i := 0
  j := tam
  repita
    vet[i] = j
    i := i + 1
    j := j - 1
  até i < tam
fim

{ implementação do bubble sort }
bubble_sort()
  inteiro: i
  i := 0
  repita
    inteiro: j
    j := 0
    repita
      se vet[i] > v[j] então
        inteiro: temp
        temp := vet[i]
        vet[i] := vet[j]
        vet[j] := temp
      fim
      j := j + 1
    até j < i
    i := i + 1
  até i < tam
fim

{ programa principal }
inteiro principal()
  preencheVetor()
  bubble_sort()
  retorna(0)
fim
```

Figura 8. Algoritmo de ordenação BubbleSort em T++ (ENTRADA)

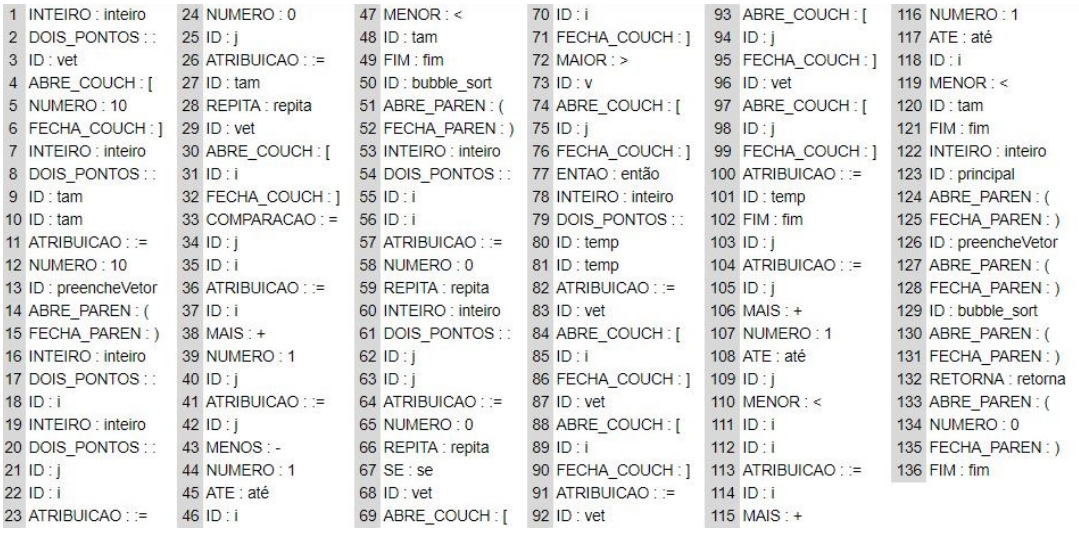

Figura 9. Algoritmo de ordenação BubbleSort em T++ (SAIDA)

Referências

Aho, A., Sethi, R., and Lam, S. (2008). *Compiladores. Princípios, Técnicas e Ferramentas*. LONGMAN DO BRASIL.

Beazley, D. M. (2018 (Acessado em 01 de Setembro, 2018)). *PLY (Python Lex-Yacc)*. `http://www.dabeaz.com/ply/ply.html`.

Wikipédia (2018). Compilador - wikipédia, a enciclopédia livre. (Acessado em 01 de Setembro, 2018).