# Compilador T++

**Thiago Alexandre Nakao França**

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 271 – 87301-899 – Campo Mourão – PR – Brazil

***Abstract.*** *The goal of this paper is to detail the development process of a T++ compiler, the processes was split into 4 main phases: lexic, syntatic and semantic analyzes, followed by the generation of intermediate code, at this moment only the lexic analyze is complete, we used the PLY framework to identify the tokens from the input programs.*

***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, no dado momento somente a análise léxica foi completada, fazendo uso do framework PLY para identificar os tokens provenientes dos programas de entrada.*

## 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

### 3.1. Objetivo

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.

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

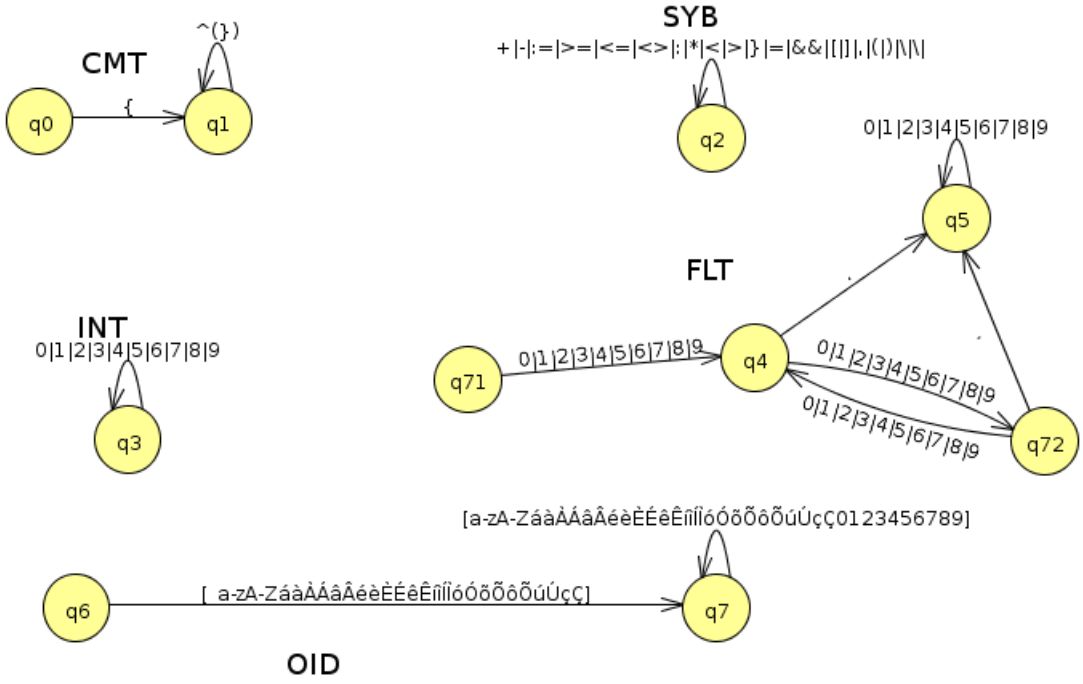


**Figura 1. Autômatos das expressões OID, RWO, CMT, SYB, INT, FLT**

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

Not yet.

## 5. Analise Semântica

Not yet.

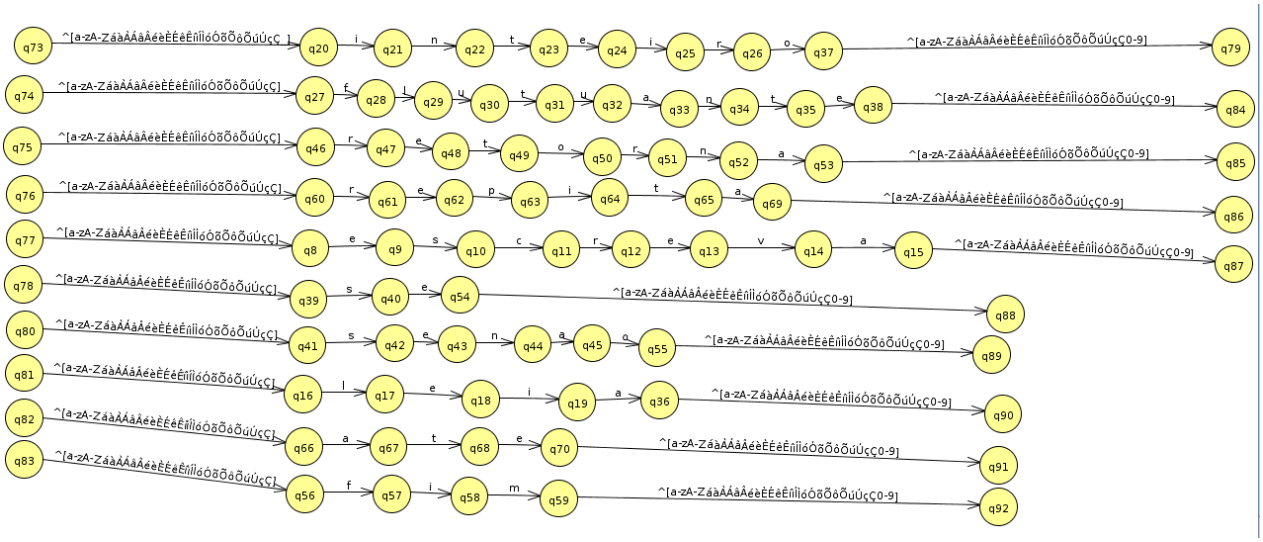


**Figura 2. Autômatos das palavras reservadas**

## 6. Geração de código intermediário

Not yet.

## 7. Conclusão

O processo de análise léxica é fundamental para o sucesso no desenvolvimento de um compilador, essa atividade também é importante pois elucida caminhos de como realizar um processo de tokenização de qualquer tipo de entrada de texto, para fins que até mesmo podem ser distintos dos da área de compiladores. Além de fixar os conhecimentos em expressões regulares.

PLY é um framework que fornece funcionalidades do lex e do yacc, embora até o momento somente o lex tenha sido explorado.

## 8. Referências

[1]LOUDEN, Kenneth C. Compiladores: princípios e práticas. São Paulo, SP: Thomson, c2004. xiv, 569 p. ISBN 8522104220

[2]LOUDEN, Kenneth C. Compiladores: princípios e práticas. São Paulo, SP: Thomson, c2004. xiv, 569 p. ISBN 8522104220.

## Referências