Análise Léxica da linguagem de programação T++

Jorge L. F. Rossi[1]

[1]Departamento de Computação – Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 071 – 87.301-899 – Campo Mourão – PR – Brasil

jorgerossi@alunos.utfpr.edu.br

Abstract. *In the implementation of a compiler, the first step to be performed is Lexical Analysis, where the source code is separated into a set of tokens, which will be used in the future as input to the Syntactic Analysis process. This paper describes a lexical parser for the T++ programming language, using Python in conjunction with the PLY[Beazley]. library.*

Resumo. *Na implementação de um compilador, a primeira etapa a ser realizada é a Análise Léxica, onde o código fonte é separado em um conjunto de tokens, os quais serão utilizados futuramente como entrada no processo da Análise Sintática. Este trabalho descreve um analisador léxico para a linguagem de programação T++, utilizando Python em conjunto com a biblioteca PLY.*

1. Introdução

Este trabalho descreve a especificação da linguagem de programação T++, utilizada para estudo durante a disciplina. Trata-se de uma linguagem simples contendo comandos em português. Tem suporte a dois tipos básicos de dados (inteiro e flutuante), funções e procedimentos, operadores aritméticos, operadores lógicos, vetores unidimensionais e um laço de repetição. Serão apresentados em detalhes cada classe de *token* disponível na linguagem, bem como os respectivos autômatos. Além disso, será descrito o funcionamento de uma implementação da análise léxica utilizando a linguagem *Python* e a biblioteca *PLY*, juntamente com exemplos de saída do sistema de varredura.

2. Especificação da linguagem

A linguagem T++ se baseia em comandos em português, tendo suporte a dois tipos de dados (inteiro e flutuante). É possível criar funções e procedimentos (funções sem retorno) além da função principal. Permite o uso de operadores lógicos e aritméticos básicos, além de ter suporte a vetores unidimensionais e a um laço de repetição.

2.1. Lista de palavras reservadas

`se, então, senão, repita, até, fim, retorna, leia, escreva, flutuante, inteiro`

2.2. Lista de operadores

`+ (soma), - (subtração), * (multiplicação), / (divisão), = (igual), <> (diferente), < (menor), > (maior) <= (menor ou igual) >= (maior ou igual), := (atribuição), && (e), || (ou), !  (não)`

2.3. Exemplo de código

(Bubble Sort)

```
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
```

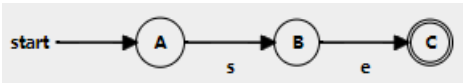
3. Lista de *tokens*

Os tokens foram separados em categorias, sendo estas: palavras reservadas, operadores aritméticos, operadores lógicos, operadores de atribuição, constantes, identificadores, símbolos e comentários.

3.1. Palavras reservadas

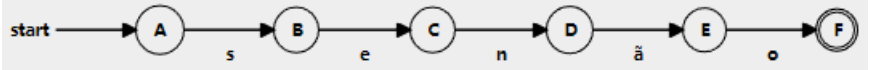
As palavras reservadas (*keywords*) são palavras que controlam a estrutura do programa, não sendo possível utilizá-las como nome de variáveis ou funções.

- *Token:* SE
- Expressão regular: `se`
- Autômato:

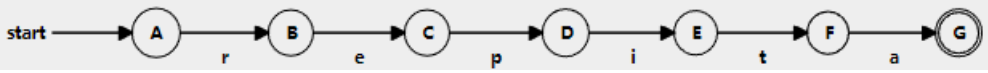


- *Token:* ENTAO
- Expressão regular: `então`
- Autômato:

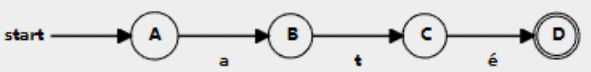
- *Token:* SENAO
- Expressão regular: `senão`
- Autômato:



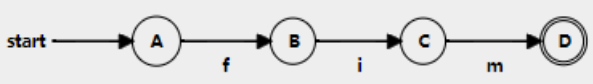
- *Token:* REPITA
- Expressão regular: `repita`
- Autômato:



- *Token:* ATE
- Expressão regular: `até`
- Autômato:

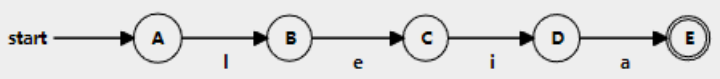


- *Token:* FIM
- Expressão regular: `fim`
- Autômato:



- *Token:* RETORNA
- Expressão regular: `retorna`
- Autômato:

- *Token:* LEIA
- Expressão regular: `leia`
- Autômato:



- *Token:* ESCREVA
- Expressão regular: `escreva`
- Autômato:

- *Token:* FLUTUANTE
- Expressão regular: `flutuante`
- Autômato:

- *Token:* INTEIRO

- Expressão regular: `inteiro`
- Autômato:

3.2. Operadores Aritméticos

Operações matemáticas básicas

- *Token:* SOMA
- Expressão regular:
- Autômato:

- *Token:* SUBTRACAO
- Expressão regular:
- Autômato:

- *Token:* MULTIPLICACAO
- Expressão regular:
- Autômato:

- *Token:* DIVISAO
- Expressão regular:
- Autômato:

3.3. Operadores Lógicos

Operações lógicos básicos

- *Token:* IGUALDADE
- Expressão regular: =
- Autômato:

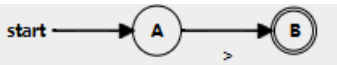
- *Token:* DESIGUALDADE
- Expressão regular: <>

- Autômato:

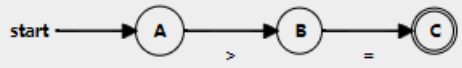
- *Token:* MENOR
- Expressão regular: <
- Autômato:

- *Token:* MENOR_IGUAL
- Expressão regular: <=
- Autômato:

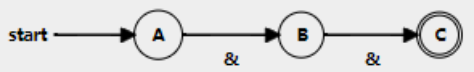
- *Token:* MAIOR
- Expressão regular: >
- Autômato:



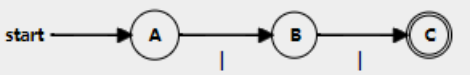
- *Token:* MAIOR_IGUAL
- Expressão regular: >=
- Autômato:



- *Token:* E_LOGICO
- Expressão regular: \&\&
- Autômato:



- *Token:* OU_LOGICO
- Expressão regular: \|\|
- Autômato:



- *Token:* NEGACAO
- Expressão regular: !
- Autômato:

3.4. Operadores de Atribuição

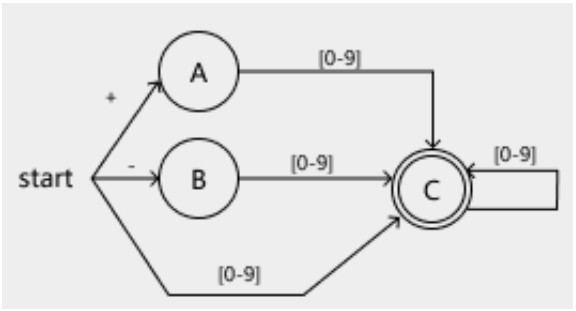
Somente um operador básico de atribuição

- *Token:* ATRIBUICAO
- Expressão regular: `:=`
- Autômato:

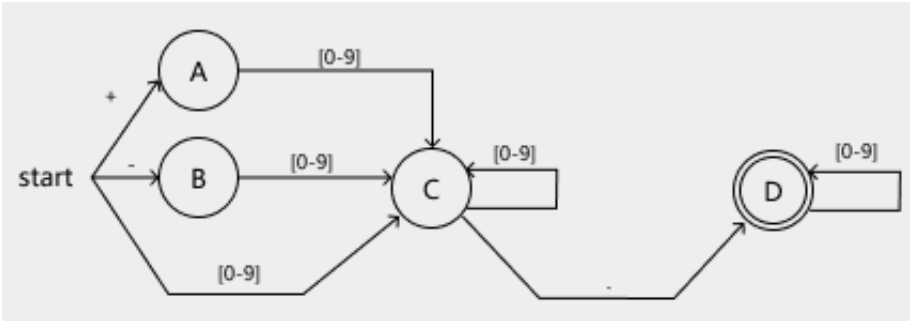
3.5. Constantes

Números inteiros e de ponto flutuante.

- *Token:* NUM_INTEIRO
- Expressão regular: `(-|+)?[0-9]+`
- Autômato:



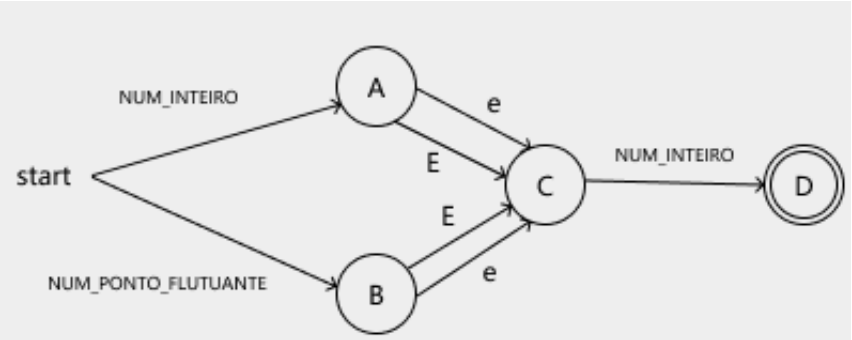
- *Token:* NUM_PONTO_FLUTUANTE
- Expressão regular: `(-|+)?[0-9]+.[0-9]*`
- Autômato:



- *Token:* NUM_NOTACAO_CIENTIFICA
- Expressão regular:

```
(((-|+)?[0-9]+.[0-9]*)|((-|+)?[0-9]+))(e|E)(-|+)?[0-9]+
```

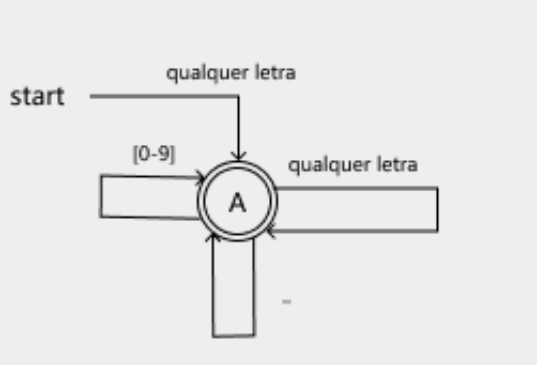
- Autômato:



3.6. Identificadores

Variáveis e funções.

- *Token:* IDENTIFICADOR
- Expressão regular:
  `[A-Za-záàâãéèêíïóôõöúçñÁÀÂÃÉÈÍÏÓÔÕÖÚÇÑ]`
  `[0-9A-Za-záàâãéèêíïóôõöúçñÁÀÂÃÉÈÍÏÓÔÕÖÚÇÑ_]*`
- Autômato:



3.7. Símbolos

Símbolos diversos.

- *Token:* VIRGULA
- Expressão regular: `,`
- Autômato:

- *Token:* DOIS_PONTOS
- Expressão regular: `:`
- Autômato:

- *Token:* ABRE_PAR
- Expressão regular: (

- Autômato:

- *Token:* FECHA_PAR
- Expressão regular:)
- Autômato:

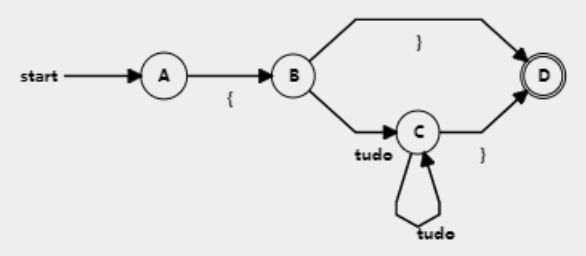
- *Token:* ABRE_COL
- Expressão regular: [
- Autômato:

- *Token:* FECHA_COL
- Expressão regular:]
- Autômato:

3.8. Comentário

Comentário de bloco

- *Token:* COMENTARIO
- Expressão regular: $\{[\hat{}\}]*[\hat{}\{]*\}$
- Autômato:



4. Detalhes da implementação

O analisador léxico da linguagem T++ foi desenvolvido utilizando a linguagem de programação Python, em conjunto com a biblioteca PLY (Python Lex-Yacc), uma implementação das ferramentas *lex* e *yacc*. Foi utilizada uma abordagem orientada a objetos na construção do analisador.

Primeiramente, foram definidas as palavras reservadas, bem como quais *tokens* as mesmas geram. O PLY requer que seja especificado uma lista com todos os *tokens* disponíveis na linguagem a ser desenvolvida, juntamente com as expressões regulares relacionadas. Alguns tipos de *tokens* necessitam de um tratamento especial, por exemplo:

- Ao encontrar um comentário, pular a linha.
- Ao encontrar um identificador, verificar se é uma palavra reservada.
- Ao encontrar um número, converte-lo para o tipo adequado.

Para os demais tipos de *tokens* não é necessário nenhum tratamento especial, bastando definir as expressões regulares que os geram.

5. Exemplo de execução

O programa a seguir lê um número a partir do teclado, calcula seu fatorial e mostra o resultado na tela.

```
inteiro: n
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
```

Após a execução da Análise Léxica, o programa resultou na seguinte lista de *tokens* como saída. O nome entre colchetes se refere ao tipo de *token* identificado pelo analisador léxico. O segundo valor é o conjunto de caracteres que resultou no *token*.

```
[INTEIRO] inteiro
[DOIS_PONTOS] :
[ID] n
[INTEIRO] inteiro
[ID] n
[ATRIBUICAO] :=
[ID] n
[SUBTRACAO] -
[NUM_INTEIRO] 1
[ATE] até
[ID] n
[IGUALDADE] =
[NUM_INTEIRO] 0
[RETORNA] retorna
[ABRE_PAR] (
[ID] fat
[FECHA_PAR] )
[SENAO] senão
[RETORNA] retorna
[ABRE_PAR] (
[NUM_INTEIRO] 0
[FECHA_PAR] )
[FIM] fim
[FIM] fim
[INTEIRO] inteiro
[ID] principal
[ABRE_PAR] (
[FECHA_PAR] )
[LEIA] leia
[ABRE_PAR] (
[ID] n
[FECHA_PAR] )
[ESCREVA] escreva
[ABRE_PAR] (
[ID] fatorial
[ABRE_PAR] (
[ID] n
[FECHA_PAR] )
[FECHA_PAR] )
[RETORNA] retorna
[ABRE_PAR] (
[NUM_INTEIRO] 0
[FECHA_PAR] )
[FIM] fim
```

Referências

Beazley, D. M. Ply (python lex-yacc).