

Documentação de Implementação do Analisador Léxico da Linguagem T++

Sávio de Oliveira Camacam

29 de Março de 2017

1 Introdução

No âmbito da disciplina de Compiladores - BCC36B - do curso de Ciência da Computação da UTFPR em Campo Mourão é pedido a implementação de um compilador para a Linguagem T++. Dividido em quatro partes, o projeto deve ser concluído até o final do semestre sobre pena de reprovação. Nesse documento é feito o relato da primeira fase de implementação que também representa a primeira etapa do processo de compilação: a análise léxica. A análise léxica seria basicamente, dada a especificação das características de uma linguagem, a T++ no caso, extrair de um arquivo fonte, os valores que definem essa linguagem, entendidos por tokens. Sendo um token, um par <classe, valor>, esse relato descreve o processo de implementação da ferramenta, as tecnologias usadas, a formação dos autômatos que descrevem a linguagem e os pontos fortes e fracos dessa implementação.

2 Especificação da Linguagem T++

A linguagem T++ objeto desse estudo é relativamente simples nos termos de suas funções, mas tem a complexidade inerente das questões de implementação de qualquer linguagem. Abaixo estão discriminados os símbolos, descritores, operações e opções suportadas pela linguagem.

2.1 Tipos de Dados

Por se tratar uma linguagem experimental para estudo de compiladores, T++ apresenta apenas os tipos de dados primitivos:

- Inteiro;
- Flutuante.

Entretanto, a linguagem permite a criação de Tipos de Vetores e Matrizes a partir desses tipos primitivos, na seguinte sintaxe, além do reconhecimento de números em notação científica:

```
inteiro : var1
flutuante : var1
var1 := 1.54334E-34
inteiro : arr1[indice1]
inteiro : arr2[indice1][indice2]
```

2.2 Palavras Reservadas

A linguagem atente ao parâmetro de uma linguagem discorrida no Português, o que significa o suporte à caracteres especiais dessa língua, como acentuação e "ç".

Suas palavras reservadas são: **se**, **então**, **senão**, **fim**, **repita**, **flutuante**, **retorna**, **até**, **leia**, **escreve**, e **inteiro**. A seguir, um exemplo de código em T++:

```
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
```

Figura 1: Código de exemplo em T++.

2.3 Símbolos e Operadores

Além do conjunto de palavras reservadas, essa linguagem também tem uma série de operadores mapeados para operações específicas:

- "*****": operador de *multiplicação*;
- "**-**": operador de *subtração*;
- "**+**": operador de soma *soma*;
- "**:=**": operador de *atribuição*;
- "**:**": símbolo *dois-pontos*;
- "**«**": operador de comparação *menor-quê*;
- "**»**": operador de comparação *maior-quê*;
- "**=**": operador de *igualdade*;
- "**«=**": operador de comparação *menor ou igual a*;
- "**»=**": operador de comparação *maior ou igual a*;
- "**[**": símbolo abre *colchetes*;
- "**]**": símbolo fecha *colchetes*;
- "**(**": símbolo abre *parênteses*;
- "**)**": símbolo fecha *parênteses*;

3 Sistema de Varredura e Autômatos

O sistema de varredura foi implementado com base em expressões regulares, descritas como se segue, gerando as marcas para identificação dos lexemas da linguagem:

- Palavra Reservada **PR** = (se|então|senão|fim|repita|flutuante|retorna|até|leia|escreva|inteiro)
Essa regra espera como entrada uma sequência precisa de caracteres para retornar uma marca de palavra reservada;
- Dígito **DIGITO** = [0-9]
Regra interna usada como definição de expressão;
- Letra **LETRA** = [a-zA-Zãõáéíóúâêôüç_]
Regra interna usada como definição de expressão para qualquer letra, que possa ser acentuada, especial ou *underscore*;
- Identificador **ID** = **LETRA**(**LETRA**|**DIGITO**)*
Regra que combina as definições anteriores de Dígito e Letra para identificar marcadores de identificadores da linguagem;
- Número **NUMERO** = **DIGITO***?**DIGITO**+(**[eE]****[+]****DIGITO**+)?
Regra genérica para captura de símbolos numéricos, dentre naturais e flutuantes em notação comum ou científica;
- Branco **BRANCO** = [**\n** | **\t** | **\r**]+
Regra para captura de símbolos de espaço e outros não imprimíveis que não fazem parte do programa;
- Símbolo **SB** = (**:****=**|**:****>**|**<****(****)**|*****|**-**|**+**|**/**|**<=**|**>=**|**[****]**)
Regra que busca o casamento de um símbolo ou uma sequência deles para uso como operador na linguagem;
- Comentário **CT** = {**[]**}*
Regra que casa abertura e fechamento de comentários reconhecido pela linguagem;
- Erro de comentário **ERROR_CT** = {**[]**}
Regra de exceção para abertura ou fechamento sem sequência de comentários;
- Erro genérico **ERRO** = [**\x20** - **\x7F**]
Regra de levantamento de erro genérico para entrada de caractere em sequência incompleta, incorreta da linguagem ou caracteres não suportados.

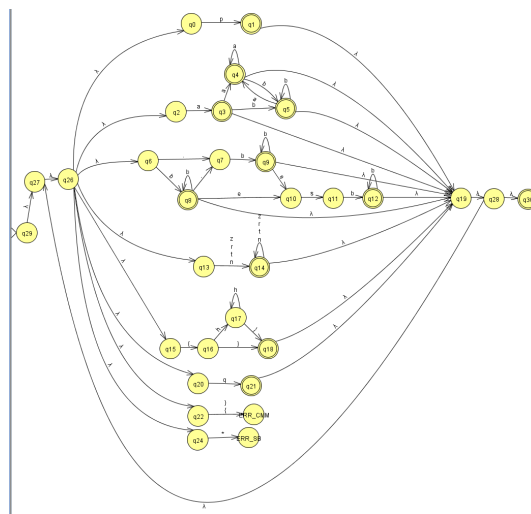


Figura 2: Autômato Finito Não-determinístico da união das partes do analisador.

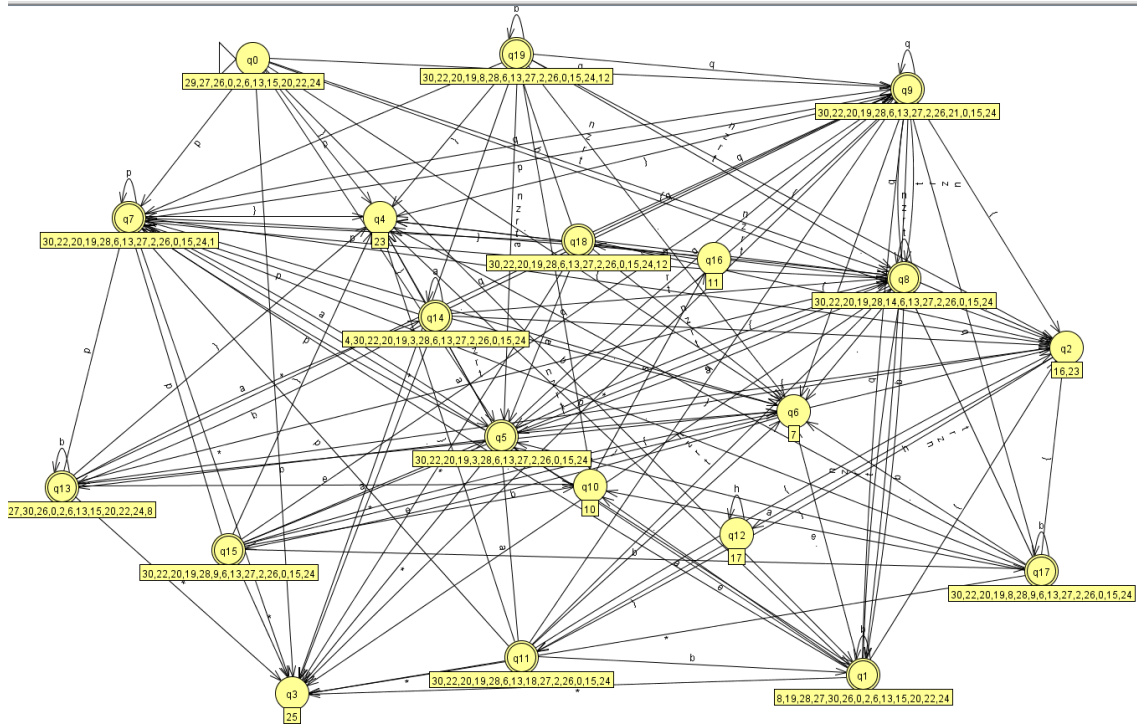


Figura 3: Autômato Finito Determinístico da união das partes do analisador.

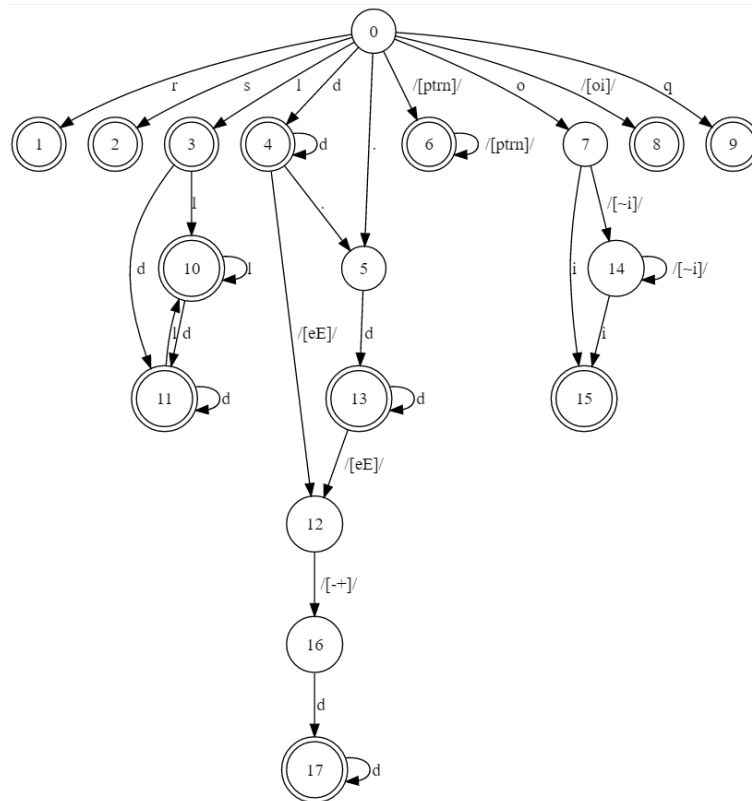


Figura 4: Segundo autômato da união das partes do analisador.

4 Processo de Implementação

A linguagem adotada para o projeto foi Java, por ser uma linguagem amigável, com ampla documentação disponível e nesse caso, por ter um simples tutorial de implementação de compiladores quase pronto. Com base no projeto Compiladores para Humanos de autoria de fulano e ciclano - referências no final do trabalho - foi possível realizar o projeto com base em implementações mais simples com uma pseudo-linguagem e um exemplo em Pascal, o que permitiu a melhor compreensão do processo de análise léxica e também, na melhor interpretação das expressões regulares.

Também segue nas referências bibliográficas, informações do livro sobre expressões regulares, que por sinal usa uma abordagem simples para ensinar as regras de interpretação de uma ER.

Em decorrência da implementação desse projeto houve uma contra-partida para com o projeto Compiladores para Humanos que se encontra disponível na forma de e-book, com participações e contribuições do autor desse relato com meras revisões de textos e reinterpretação de instruções do tutorial para que futuros estudantes de Compiladores possam desfrutar dessa obra.

No modelo do Compiladores para Humanos é feita uma introdução sobre o processo de interpretação, a descrição de classes de tokens e noções de como o programa deve ser implementado. Em seguida há um breve exemplo para um trecho de código na linguagem Pascal e com base nesse código foram implementadas as regras para a linguagem T++.

4.1 Classes do Programa

O analisador tem três classes básicas e uma quarta classe que é gerada automaticamente pelo JFlex, a biblioteca do java que implementa o Lex.

4.1.1 Classe: *GeneratorTpp*

É a classe responsável pela leitura do arquivo que contém a declaração das regras de interpretação baseadas em Lex. Uma vez que o caminho de acesso até o arquivo está pronto, a execução dessa classe gera automaticamente a classe *LexicalAnalyzer.java* que implementa as regras do arquivo *.lex* que especifica as regras da linguagem. Já que essa é a classe dinâmica onde não é feita nenhuma alteração, não há porque descrevê-la.

4.1.2 Classe: *TppLexicalAnalyzer*

Por outro lado, essa é classe responsável pela leitura do arquivo de entrada que leva a amostra da linguagem à ser interpretada e compilada e também gera a saída com o par <token, valor> depois de feita a análise.

4.2 Entrada e Saída

A entrada desse programa é composta de dois arquivos: o primeiro contém a descrição das regras de interpretação da linguagem em Lex; o segundo, contém o código da linguagem que deseja-se compilar.

O processo trata da leitura caractere a caractere do arquivo de entrada do programa - código fonte - como um *stream*, onde a partir dele, são buscadas combinações de regras léxicas dada a máquina de estados da linguagem que indica a criação das marcas dessa linguagem.

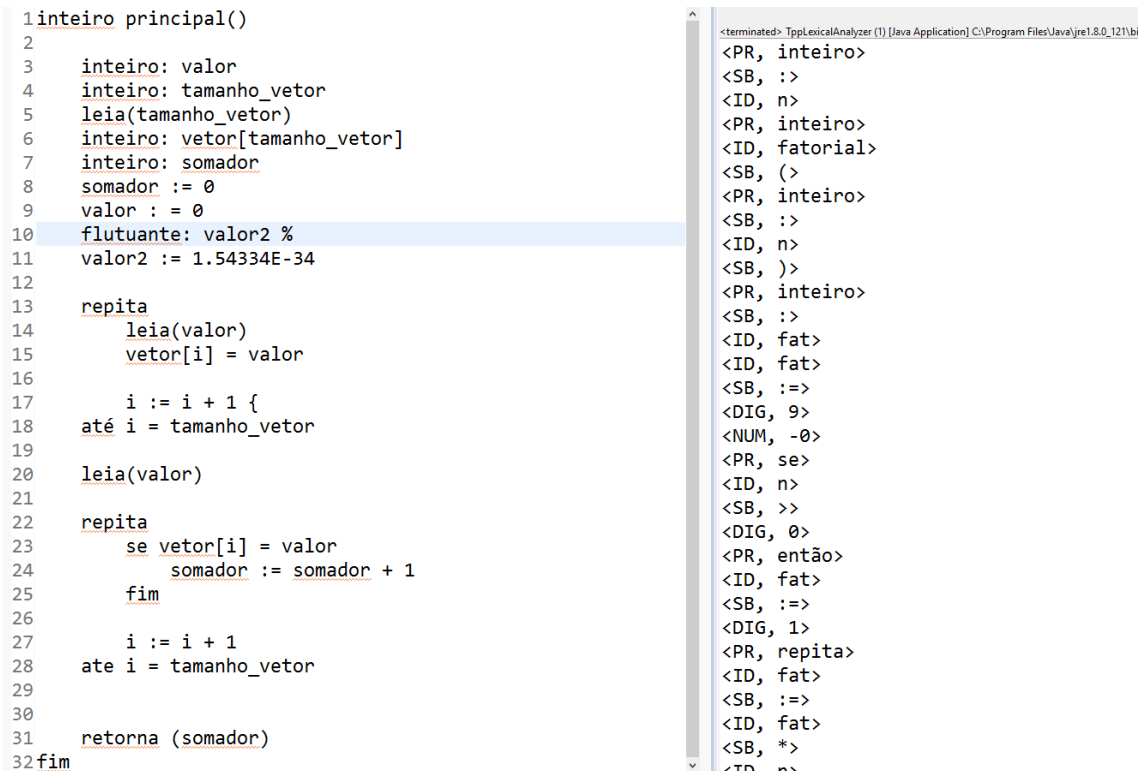
Como saída, é exibida no console do programa, a combinação <classe, valor> para cada marca válida da linguagem. Em situação de caractere ou sequência inválidos é apontado um erro, com especificação ou não do erro, dependendo da sua natureza, por exemplo, abertura e não fechamento de *chaves*.

4.3 Execução

Uma fase da execução do programa consiste na geração de uma classe automática para uso sequente por outra classe desse programa, então:

1. Execute a classe **GeneratorTpp**;
2. Atualize o canal de execução: *refresh* do diretório para reconhecimento da classe gerada;
3. Execute então a classe **TppLexicalAnalyzer**.

Assumindo que o programa chegou até o caminho dos arquivos de entrada a saída será exibida dessa forma:



```
1 inteiro principal()
2
3     inteiro: valor
4     inteiro: tamanho_vetor
5     leia(tamanho_vetor)
6     inteiro: vetor[tamanho_vetor]
7     inteiro: somador
8     somador := 0
9     valor := 0
10    flutuante: valor2 %
11    valor2 := 1.54334E-34
12
13    repita
14        leia(valor)
15        vetor[i] = valor
16
17        i := i + 1 {
18    até i = tamanho_vetor
19
20    leia(valor)
21
22    repita
23        se vetor[i] = valor
24            somador := somador + 1
25        fim
26
27        i := i + 1
28    ate i = tamanho_vetor
29
30
31    retorna (somador)
32 fim
```

```
<terminated> TppLexicalAnalyzer (1) [Java Application] C:\Program Files\Java\jre1.8.0_121\bin
<PR, inteiro>
<SB, :=>
<ID, n>
<PR, inteiro>
<ID, fatorial>
<SB, (>
<PR, inteiro>
<SB, :=>
<ID, n>
<SB, )>
<PR, inteiro>
<SB, :=>
<ID, fat>
<ID, fat>
<SB, :=>
<DIG, 9>
<NUM, -0>
<PR, se>
<ID, n>
<SB, >>
<DIG, 0>
<PR, então>
<ID, fat>
<SB, :=>
<DIG, 1>
<PR, repita>
<ID, fat>
<SB, :=>
<ID, fat>
<SB, *>
<ID, n>
```

Figura 5: Situação de saída do programa.

5 Conclusão

Sendo a primeira vez que há a prática da implementação dos conceitos visto em Teoria de Autômatos e uso de expressões regulares, o projeto de implementação do analisador léxico para o compilador da linguagem T++ permitiu a fixação desses conceitos e agregou outros conhecimentos da área de interpretação e execução de linguagens de programação, assim como a implementação prática também de conceitos vistos no levantamento de questões de projetos de linguagens de programação. [?]