Compiladores - Análise Léxica

Alisson da S. Vieira

[1]Ciência da Computação – Universidade Tecnológica Federal do Paraná (UTFPR)
Caixa Postal 15.064 – 91.501-970 – Campo Mourão – PR – Brasil

alissonv@alunos.utfpr.edu.br

Abstract. *For this work, a start code for Lexical Analysis was provided by the teacher. The Analysis is the first step of the compilation process, where the language tokens are extracted.The language chosen was T++, and the work describes a little about the characteristics of the language. It also discusses which language and library was used to perform the analysis, as well as showing output examples at the end.*

Resumo. *Para este trabalho, foi disponibilizado pelo professor, um código inicial para a Análise Léxica. Esta que é a primeira etapa do processo de compilação, onde basicamente se extrai os tokens da linguagem. A linguagem escolhida foi o T++, e o trabalho descreve um pouco sobre as características da linguagem. Ele também aborda sobre qual linguagem e biblioteca foi utilizada para realizar a análise, além de ao final mostrar exemplos de saída.*

1. Introdução

A análise léxica, é um processo de compilação que, faz uma varredura no código fonte, com o objetivo de converter os caracteres presentes no código em um conjunto de marcas (tokens). Esses tokens representam caracteres do código, como: laços de repetições (for, while, foreach, etc), comparações (if, else, etc), símbolos especiais (+, -, *, /, etc). A saída desse processo de análise léxica é uma lista desses tokens, que serão utilizados nas próximas etapas do processo de compilação.

2. Objetivo

Estaremos realizando na disciplina um compilador para a linguagem de programação fictícia T++, esta que foi desenvolvida com o intuito de auxiliar com o aprendizado dos alunos na disciplina. Neste trabalho especificamente, estaremos realizando a análise léxica a primeira parte do processo de compilação, e para trabalho foi utilizado a linguagem de programação `Python` com a biblioteca `PLY`. A decisão de se utilizar `Python`, se deve ao fato de que é uma linguagem voltada para a produtividade, possuindo diversas ferramentas nativas que auxiliam o desenvolvedor a apenas focar em produzir. Já a escolha de usar a biblioteca `PLY`, se deve ao fato de que ela fornece a maioria dos recursos `lex`/`yacc` padrões, incluindo suporte para: produções vazias, regras de precedência, recuperação de erros e suporte para gramáticas ambíguas, além de ser relativamente simples de se usar.

3. Linguagem de Programação T++

Como já foi dito, a linguagem T++ foi desenvolvida para auxiliar no processo de aprendizagem dos alunos, e ela é totalmente em português. Quanto aos tipos básicos suportados pela linguagem são tipos inteiro e flutuante, além de contar com suporte a arranjos unidimensionais e bidimensionais. Nessa linguagem quando o retorno de uma função é omitido ela automaticamente vira um procedimento, caso contrário vira uma função. Nas subseções abaixo iremos ver mais detalhes sobre as características dessa linguagem.

3.1. Tipos de Variáveis

Os tipos de variáveis suportadas pela linguagem são:

- Números inteiros
- Números reais
- Números com notação científica

3.2. Operações

Os tipos de operações suportadas pela linguagem são apresentados na Tabela 1.

OPERAÇÃO	***SÍMBOLO***
Soma	+
Subtração	-
Multiplicação	*
Divisão	/
OR	||
AND	&&
NOT	!

Tabela 1. Operações suportadas na linguagem T++.

3.3. Comparações

Os tipos de comparações suportadas pela linguagem se encontram na Tabela 2.

OPERAÇÃO	***SÍMBOLO***
Maior	>
Menor	<
Maior igual	>=
Menor igual	<=
Diferença	<>

Tabela 2. Comparações suportadas pela linguagem T++.

3.4. Palavras Reservadas

As palavras reservadas pela linguagem são mostradas na Tabela 3.

se
então
senão
fim
repita
flutuante
retorna
ate
escreva
inteiro

Tabela 3. Palavras reservadas da linguagem.

3.5. Exemplo de Código

No Código 1 temos um exemplos de um código em T++. Este código é um teste simples que utiliza quase tudo que foi discutido nas seções: variáveis, tipo de função, laço de repetição, comparação, igualdade, etc.

Código 1. Exemplo em T++

```
1   inteiro principal()
2       inteiro: a
3
4       a := 1
5
6       repita
7
8           se a = 10
9               escreva(a)
10          fim
11
12          a := a + 1
13      ate a = 10
14
15      retorna(0)
16
17  fim
```

Nesse exemplo, temos apenas uma função, a função principal que retorna um valor inteiro. De inicio declaramos uma variável de nome a do tipo inteiro, e setamos o valor de 1 a ela. Lopo após, temos um loop, que se repete até que essa variável a seja igual ao valor 10. Dentro desse loop fazemos apenas uma verificação simples, verificamos se o valor de a é 10, caso positivo escrevemos esse valor com a função “escreva”, caso não seja 10, aumentamos em 1 no valor da variável a. E ao final do código, retornamos 0.

4. Expressões Regulares

Como foi dito anteriormente, a análise léxica faz uma varredura no código fonte, com o objetivo de identificar tokens. Para realizar esse processo de identificação, foi utilizado expressões regulares para cada um dos símbolos, tanto simples quanto complexos. Um expressão regular é uma técnica que provê uma forma concisa e flexível de identificar

cadeias de caracteres especificas. A Tabela 4 mostra as expressões simples que foram usadas para identificação de alguns tokens.

OPERAÇÃO	*SÍMBOLO*
Adição	\+
Subtração	-
Multiplicação	*
Divisão	\
Abre parenteses	\(
Fecha parenteses	\)
Abre colchete	[
Fecha colchete	]
Virgula	,
Atribuição	:=
Dois pontos	:
E lógico	&&
OU lógico	\—\—
Negação	!
Diferença	<>
Menor igual	<=
Maior igual	>=
Menor	<
Maior	>
Igual	=

Tabela 4. Expressões regulares simples

A Tabela 5 mostra as expressões regulares restantes, essas que são mais complexas ao se comparar com as da Tabela 4.

TOKEN	*EXPRESSÃO REGULAR*
Dígito	*([0-9])*
Letra	*([a-zA-ZÁãÃàÀéÉíÍóÓõÕ])*
Sinal	*([\-\+]?)*
ID	*(([a-zA-ZáÁãÃàÀéÉíÍóÓõÕ]) (([0-9])+| _ | ([a-zA-ZáÁãÃàÀéÉíÍóÓõÕ]))*)*
Inteiro	\d+
Flutuante	\d+[eE][-+]?\d+| (\.\d+| $nd + n.nd*$) ([eE][-+]?\d+)?
Notação Científica	([-\+]?)([1-9])\.([0-9]) +[eE]([-\+]?)([0-9]+)
Comentário	({((.—\n)*?)})
Nova linha	\n+

Tabela 5. Expressões regulares complexas.

5. Autômatos

Uma expressão regular define formalmente um padrão, mostrando quais os sub-padrões ou expressões regulares que a formam, com qual regularidade ou sequencia cada uma delas aparece, e quais são os diferentes sub-padrões que podem ser reconhecidos em uma mesma posição do padrão. Vamos pegar por exemplo a Expressão 1, onde o "a" representa o carácter, o "*" significa que a sequência de caracteres "[0-9]" (quais quer números de 0 a 9) podem se repetir.

$$(a * [0 - 9]) \tag{1}$$

Dado a seguinte sequência de caracteres: "a a1 a09 a123", os caracteres retornados após a aplicação dessa expressão regular seria: "a1", "a09" e "a123". Qualquer expressão regular pode ser convertida em um autômato finito que ela descreve, e vice versa, ou seja, podemos converter a Expressão 1 em um automato finito, como vemos na Figura 1.

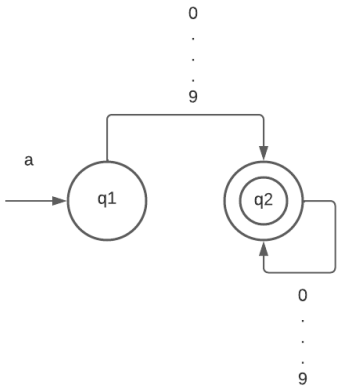

Figura 1. Autômato resultante da conversão da Expressão 1.

Assim como fizemos com a Expressão 1, podemos fazer o mesmo com todas as expressões regulares já mostradas acima, e para exemplificar iremos usar a Expressão 2 que identifica o token DIGITO.

$$([0 - 9]) \tag{2}$$

Para essa expressão, o automato resultante da conversão é o representado na Figura 2.

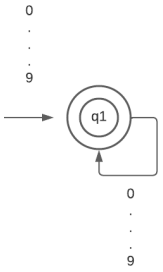

Figura 2. Autômato resultante da conversão da Expressão 2.

Também podemos observar na Figura 3, um automato para identificar a palavra reservada da linguagem “repita”.

Figura 3. Autômato que identifica a palavra reservada “repita”.

6. Detalhes de Implementação

Como já foi citado anteriormente, para realizar esse trabalho foi disponibilizado pelo professor um código inicial, utilizando a linguagem de programação `Python`, e a biblioteca `Ply` (Python Lex-Yacc). O `Ply` conta com 2 módulos separados: `lex.py` e `yacc.py`, para a análise léxica iremos estar utilizando o `lex.py`. O `lex.py` já prove uma implementação para pegar os tokens de um dado arquivo, como pode ser observado no Código 2. Para utilizar essa função, é necessário antes construir a instância do lexer, e passar o arquivo para ele realizar a leitura.

Código 2. Construção da instancia do lex

```
1   # construir a instancia do lexer
2   lexer = lex.lex(optimize=True, debug=True, debuglog=log)
3   lexer.input(source_file)
4
5    while True:
6
7   # pegar os tokens
8   tok = lexer.token()
9
10  # No more input
11  if not tok:
12      break
13
14  print(tok)
```

E dessa forma, para todos os tokens que definimos, podemos criar funções que irão se executar assim que o `lex` encontrar o token referenciado, como mostra a Figura **??**. No

caso dessa função, toda vez que o `lex` identificar o token da notação científica (expressão regular identificada na Tabela 5) ele irá apenas retornar o token, que é justamente oque queremos.

Código 3. Exemplo função lex.

```
@TOKEN(notacao_cientifica)
def t_NUM_NOTACAO_CIENTIFICA(token):

    return token
```

Enquanto no Código 3, apenas retornamos o token, temos casos como o do Código 4 em que temos que realizar um tratamento, neste caso em específico, temos que pular as linhas.

Código 4. Exemplo função nova linha do `lex`.

```
@TOKEN(nova_linha)
def t_NEWLINE(token):

    # para pular as linhas referentes ao valor
    token.lexer.lineno += len(token.value)

    # nao a necessidade de retornar o token
```

Foram feitas algumas melhorias em relação a saída do programa, foi implementado um argumento adicional 'd', onde na hora de execução do programa conseguimos observar uma saída mais detalhada da execução como vemos no Código 5.

Código 5. Saída mais detalhada.

```
while True:

    # pegar os tokens
    tok = lexer.token()

    # no more input
    if not tok:
        break

    if detailed:
        print(tok)
        arq.write(str(tok) + '\n')
    else:
        print(tok.type)
        arq.write(str(tok.type) + '\n')
```

Também foi implementado uma mensagem de erro mais detalhada também como mostra o Código 6.

Código 6. Função erro com adição de uma saída mais detalhada.

```
# tratamento de erros
def t_error(token):

    # variaveis para controle
    global error, detailed, arq

    error = True

    if detailed:
        print(
            'Foi encontrado um caracter invalido: "{}", na linha: {} e na coluna {}'
            .format(token.value, token.lineno, token.lexpos)
        )

        arq.write(
            'Foi encontrado um caracter invalido: "{}", na linha: {} e na coluna {}'
            .format(token.value, token.lineno, token.lexpos)
        )

    else:
        print(
            "Caracter invalido '{}'"
            .format(token.value[0])
        )

        arq.write(
            "Caracter invalido '{}'"
            .format(token.value[0])
        )

    # pulo o erro
    token.lexer.skip(1)
```

Juntamente com o argumento adicional, também foi feito uma melhoria da saída da execução do código, onde toda vez que o mesmo for executado, seu resultado estará disponível em um arquivo chamado “saida.txt” na pasta do programa.

7. Execução do Código

Para testar nosso código, foi usado como código de entrada, um exemplo em T++ que implementa o algoritmo de busca linear, como mostra o Código 7.

Código 7. Exemplo da busca linear em T++

```
1     inteiro: A[20]
2
3     inteiro busca(inteiro: e)
4
5         inteiro: retorno
6         inteiro: i
7
8         retorno := 0
9         i := 0
10
11        repita
12
13            se A[i] = 0
14                retorno := 1
15            fim
16
17            i := i + 1
18
19        ate i = 20
20
21        retorna(retorno)
22
23    fim
24
25    inteiro principal()
26
27        inteiro: e
28        inteiro: i
29
30        i := 0
31
32        repita
33            A[i] := i
34            i := i + 1
35        ate i = 20
36
37        leia(e)
38        escreva(busca(e))
39        retorna(0)
40    fim
```

E para esse exemplo, temos a saída completa sem o parâmetro “d” na hora da execução na Figura 4.

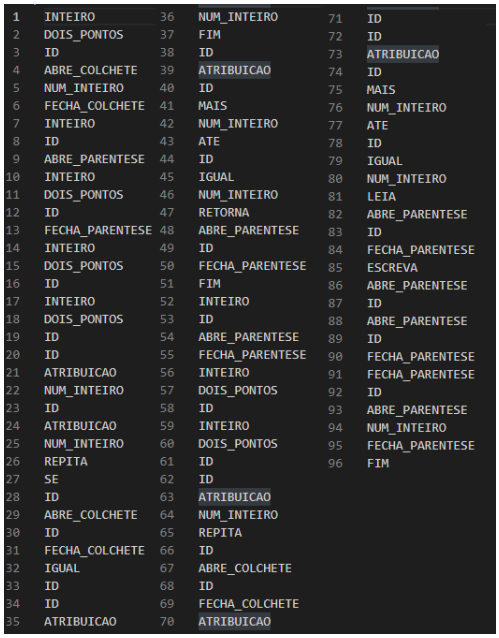

Figura 4. Saída da execução da análise léxica.

Já a Figura 5 mostra a saída com o parâmetro “d”.

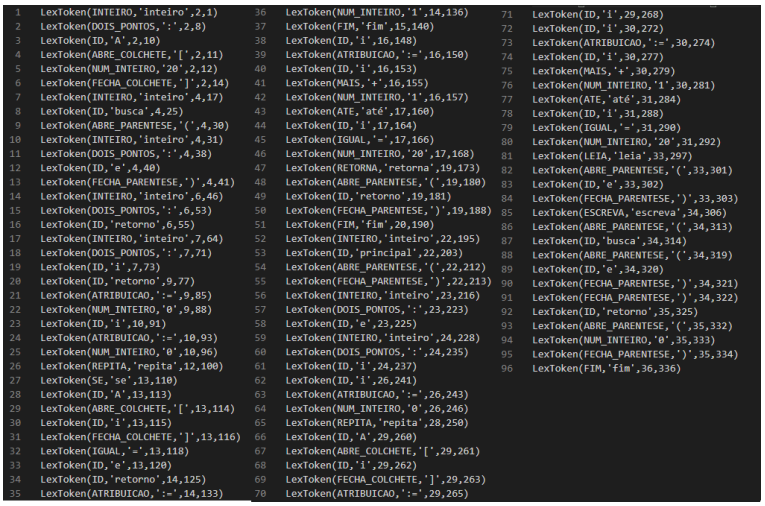

Figura 5. Saída detalhada da execução da análise léxica.

References

Beazley, D. M. Documentação ply.py. ttps://www.dabeaz.com/ply/ply.tml.

Gonçalves, R. A. Conjunto de testes. ttps://moodle.utfpr.edu.br/mod/resource/view.pp?id=725874.

Gonçalves, R. A. Slides - análise léxica. ttps://moodle.utfpr.edu.br/pluginfile.pp/239681/mod$_resource/content/4/aula-05-analise-lexica-lex-ply.md.slides.pdf$.