

# Construção de Compiladores - INE5426

Relatório - AL 1  
Construção de Analisador Léxico

Universidade Federal de Santa Catarina

Eduardo Gutterres [17200439]  
Felipe de Campos Santos [17200441]  
Ricardo Giuliani [17203922]

# 1. Identificação dos tokens

## Operadores Aritméticos

- PLUS → símbolo de adição
- MINUS → símbolo de subtração
- TIMES → símbolo de multiplicação (asterisco)
- DIVIDE → símbolo de divisão (barra)
- MOD → símbolo de resto de divisão (porcento)
- ASSIGN → símbolo de igual

## Operadores Relacionais

- EQUAL → dois símbolos de igual
- DIFFERENT → exclamação seguida de símbolo de igual
- LT (LESS THAN) → símbolo de menor
- LE (LESS THAN OR EQUAL) → símbolo de menor seguido de símbolo de igual
- GT (GREATER THAN) → símbolo de maior
- GE (GREATER THAN OR EQUAL) → símbolo de maior seguido de símbolo de igual

## Outros Operadores

- COMMA → vírgula
- SEMICOLON → ponto e vírgula
- RPARENTHESSES → parênteses direito
- LPARENTHESSES → parênteses esquerdo
- LBRACKET → colchete esquerdo
- RBRACKET → colchete direito
- LBRACE → chave esquerda
- RBRACE → chave direita
- DOT → ponto

## Constantes

- FLOATCONSTANT → número com decimal
- INTCONSTANT → número (decimal, hexadecimal, octal ou binário)
- STRINGCONSTANT → constante entre aspas

## Tokens ignorados

- LINEBREAK → quebra de linha
- COMMENT → qualquer coisa entre /\* e \*/ ou após // na mesma linha
- TAB → espaço tabulado

## 2. Definições regulares dos tokens

### Operadores Aritméticos

- PLUS  $\rightarrow +$
- MINUS  $\rightarrow -$
- TIMES  $\rightarrow *$
- DIVIDE  $\rightarrow /$
- MOD  $\rightarrow \%$
- ASSIGN  $\rightarrow =$

### Operadores Relacionais

- EQUAL  $\rightarrow ==$
- DIFFERENT  $\rightarrow !=$
- LT (LESS THAN)  $\rightarrow <$
- LE (LESS THAN OR EQUAL)  $\rightarrow <=$
- GT (GREATER THAN)  $\rightarrow >$
- GE (GREATER THAN OR EQUAL)  $\rightarrow >=$

### Outros Operadores

- COMMA  $\rightarrow ,$
- SEMICOLON  $\rightarrow ;$
- RPARENTHESSES  $\rightarrow )$
- LPARENTHESSES  $\rightarrow ($
- LBRACKET  $\rightarrow [$
- RBRACKET  $\rightarrow ]$
- LBRACE  $\rightarrow \{$
- RBRACE  $\rightarrow \}$
- DOT  $\rightarrow .$

### Constantes

- FLOATCONSTANT  $\rightarrow (\text{NUMERO}).(\text{NUMERO})^+$
- INTCONSTANT  $\rightarrow (\text{NUMERO})^+ \mid (0|1)^+ [b|B] \mid (\text{NUMERO} \mid A-F)^+ [h|H] \mid (0-7)^+ [o|O]$
- STRINGCONSTANT  $\rightarrow " (\text{NUMERO} \mid \text{LETRA}) (^ \backslash n \mid \backslash r \mid " ) "$

### Tokens ignorados

- LINEBREAK  $\rightarrow \backslash n^+$
- COMMENT  $\rightarrow // (\text{LETRA} \mid \text{NUMERO})^+ \mid /* (\text{LETRA} \mid \text{NUMERO})^+ */$
- TAB  $\rightarrow \backslash t^+$
- LETRA  $\rightarrow (a \mid b \mid c \mid d \mid \dots \mid z \mid A \mid B \mid C \mid \dots \mid Z)$
- NUMERO  $\rightarrow (0 \mid 1 \mid 2 \mid \dots \mid 0)$

### 3. Diagramas de transição dos tokens

Ver anexo “Diagramas de Transição” no diretório principal do repositório do GitHub.

### 4. Descrição da tabela de símbolos

#### a. Implementação

A tabela de símbolos foi implementada usando uma estrutura *dict* do python, onde a chave dos valores é o nome do identificador passado no código.

#### b. Símbolos armazenados

A tabela de símbolos armazena todos os identificadores do programa, ou seja, nomes de variáveis de funções.

#### c. Atributos escolhidos para armazenar

Os atributos escolhidos para se armazenar na tabela de símbolos foi o nome do identificador, seu tipo, linha que foi declarada e linhas onde o identificador foi referenciado.

Identificadores que não tenham sido declarados (variáveis usadas sem declaração ou funções chamadas sem declaração), ficam com a flag ‘NOT\_DECLARED’ no atributo ‘declared\_line’ e ‘NO TYPE’ no atributo ‘type’.

Ex: para um identificador “aux”, de tipo *int*, declarada na linha 7 e referenciada apenas na linha 8, sua entrada na tabela de símbolos seria:

```
{
    'aux':{
        'name':'aux', 'type': 'int', 'declared_line': 7,
        'referenced_lines':[8]
    },
}
```

### 5. Descrição do PLY

PLY é uma implementação das ferramentas de parsing LEX (um gerador de analisador léxico) e YACC (Yet Another Compiler Compiler - ferramenta para análise sintática) feita totalmente em Python, por David M. Beazley.

Neste primeiro trabalho, utilizamos apenas o `ply.lex`, ferramenta de análise léxica. Para isso, primeiro precisamos definir o que serão nossos *tokens*.

Para isso, precisamos seguir o padrão: Criar uma variável “`t_<nome do token>`” que recebe uma expressão regular que reconhece a sequência de caracteres daquele token.

Por exemplo, para definir os tokens PLUS (símbolo de adição) e FLOATCONSTANT (constante numérica com parte decimal) fazemos:

```
t_PLUS = r'\+'
```

# reconhece o caracter "+"

```
t_FLOATCONSTANT = r'\d+\ . \d+'
```

# reconhece numeros seguido de um ponto seguido de um outro número

Além disso, precisamos definir quais as palavras reservadas da nossa linguagem (pode ser feito da maneira acima, mas o `ply.lex` oferece um jeito mais fácil) criando um dict com a key sendo a palavra (“*regex*”) e o valor sendo seu token. Por exemplo, para as palavras reservadas *int*, *float* e *string*, fariamos:

```
reserved = {'int': 'INT', 'float': 'FLOAT', 'string': 'STRING'}
```

Ou seja, toda vez que o analisador “ler” a palavra “*int*”, ele vai associá-la ao token “*INT*”.

Definimos também o que será ignorado pelo nosso analisador. Quais entradas ele não vai reconhecer como token nenhum. Exemplo, ignoramos em nosso programa quaisquer espaços e tabulações, seguindo o padrão definido pelo `ply.lex` de criar uma variável *t\_ignore* e atribuir à ela o *regex* (ou no nosso caso, a “*string*”) que reconhece aquilo que deve ser ignorado:

```
t_ignore = '\t'
```

Feito tudo isso, criaremos nossa coleção de tokens. Criamos uma variável *tokens* que recebe uma lista com o nome de todos os tokens que foram definidos até agora, ou seja, tudo que vem depois de *t\_* nas variáveis.

```
tokens = ['INT', 'FLOAT', 'STRING', 'PLUS', 'FLOATCONSTANT']
```

Depois, é só fazermos um

```
lexer = lex.lex()
```

que nosso analisador léxico está construído e associado à variável *lexer*. Agora, precisamos passar para ele a entrada que queremos analisar. Para isso, recebemos essa entrada de alguma forma (lendo um arquivo, pedindo para o usuário digitar no terminal, etc) e passamos para nosso *lexer* como

```
lexer.input(dados)
```

onde *dados* é a representação em string do que vamos analisar.

Agora, sempre que chamarmos

```
tok = lexer.token()
```

o lexer atribui à nossa variável *token* o próximo token do input que passamos à ele, de modo que podemos iterar sobre todos os tokens reconhecidos pelo nosso lexer e fazer as operações necessárias à ele.

Por exemplo, um resumo de como criamos nossa tabela de símbolos foi identificar se o token era do tipo 'IDENT', e caso fosse adicionamos ele à nossa tabela de símbolos (fazendo as checagens necessárias, mas que aqui foram omitidas)

```
tok = lexer.token()
if tok.type == 'IDENT':
    ## operações feitas para alocação do token na tabela de símbolos
```

Assim, podemos criar o output do nosso programa a cada token que é lido. Adicioná-lo a uma lista de todos os tokens reconhecidos, criar a tabela de símbolos, etc.

No nosso caso, nossa tabela de símbolos é retornada com as informações de nome, tipo, linha onde o identificador foi declarado e linhas onde foi referenciado. Exemplo, se nossa entrada tiver

```
10| (...)
11| int a = 1;
12| int b = a + 1;
13| (...)
```

Nossa tabela de símbolos terá as entradas

NOME	TIPO	LINHA DECLARADA	LINHA REFERENCIADA
a	int	11	12
b	int	12	-

Para exemplificar melhor uma entrada e uma saída, vamos pegar parte do programa exemplo (prog1.lcc) que entregamos com o trabalho:

```
def functionName(int a) {
    int b;
    int z;
    b = a + 2;
    z = b + 1;

    string message;
    message = "mensagem";
    print(message)
    print(z);
} |

// esse comentario será ignorado

/* esse
tambem */
```

O que esperamos de saída:

1. O aviso do erro léxico na última linha, no carácter “|”
2. A tabela de símbolos mostrando a declaração, tipo e referências das variáveis *a*, *b*, *z*, e *message* e da função *functionName*

Saída:

```
D:\Documents\Git\UFSC\Compiladores\AL1>py main.py programas/prog1.lcc
```

#### ERRO LÉXICO ENCONTRADO

```
| -> Character: '|'
| -> Linha:    11
| -> Coluna:    3
```

#### ==== Symbols Table =====

NAME	TYPE	DECLARED	REFERENCED
functionName	def	1	[]
a	int	1	[4]
b	int	2	[4, 5]
z	int	3	[5, 10]
message	string	7	[8, 9]

## Link para o repositório do Github:

[https://github.com/felipecampossantos/AL1\\_INE5426](https://github.com/felipecampossantos/AL1_INE5426)

## Links Úteis

<https://www.dabeaz.com/ply/>

<https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/lexical-analysis.html>

<https://www.dabeaz.com/ply/PLYTalk.pdf>

<https://sites.google.com/site/2012pcs25086482782/home/o-analisador-lexico>

<https://earthly.dev/blog/python-makefile/>