Construção de Compiladores
Daniel Lucrédio, Helena Caseli, Mário César San Felice e Murilo Naldi
Tópico 02 - Análise Léxica - Lista de Exercícios Resolvida
(Última revisão: jan/2020)

1. Identifique, em uma tabela, todos os tokens que compõem os programas seguintes. Cada linha da tabela será um token. A tabela terá 3 colunas: na coluna 1, especifique a cadeia (lexema) correspondente; na coluna 2, especifique sua classe (identificador, palavra ou símbolo reservado, número, comentário, etc.); e na coluna 3 especifique o padrão utilizado no reconhecimento (em português mesmo).

a) Pascal

```
function max(i, j: integer): integer;
{ retorna o maior dos inteiros entre i e j }
begin
if i > j then max := i
else max := j
end;
```

	Lexema	Classe	Padrão
1	function	palavra-chave	próprio lexema
2	max	identificador	sequencia de caracteres
3	(	pontuação	próprio lexema
4	i	identificador	sequencia de caracteres
5	,	pontuação	próprio lexema
6	j	identificador	sequencia de caracteres
7	:	pontuação	próprio lexema
8	integer	palavra-chave	próprio lexema
9	)	pontuação	próprio lexema
10	:	pontuação	próprio lexema
11	integer	palavra-chave	próprio lexema
12	;	pontuação	próprio lexema
13	{ retorna o maior dos inteiros entre i e j }	comentário	sequencia de caracteres delimitada por { e }
14	begin	palavra-chave	próprio lexema
15	if	palavra-chave	próprio lexema
16	i	identificador	sequencia de caracteres
17	>	operador relacional	próprio lexema
18	j	identificador	sequencia de caracteres
19	then	palavra-chave	próprio lexema
20	max	identificador	sequencia de caracteres
21	:=	operador atribuição	próprio lexema
22	i	identificador	sequencia de caracteres
23	else	palavra-chave	próprio lexema
24	max	identificador	sequencia de caracteres
25	:=	operador atribuição	próprio lexema
26	j	identificador	sequencia de caracteres
27	end	palavra-chave	próprio lexema
28	;	pontuação	próprio lexema

Obs: as classes poderiam ter outros nomes. Também é possível definir classes mais específicas do que essas.

b) C

```
int max(i, j) int i, j;
/* retorna o maior dos inteiros entre i e j */
{
return i > j ? i : j
}
```

	Lexema	Classe	Padrão
1	int	palavra-chave	próprio lexema
2	max	identificador	sequência de caracteres
3	(	pontuação	próprio lexema
4	i	identificador	sequência de caracteres
5	,	pontuação	próprio lexema
6	j	identificador	sequência de caracteres
7	)	pontuação	próprio lexema
8	int	palavra-chave	próprio lexema
9	i	identificador	sequência de caracteres
10	,	pontuação	próprio lexema
11	j	identificador	sequência de caracteres
12	;	pontuação	próprio lexema
13	/* retorna o maior dos inteiros entre i e j */	comentário	sequência de caracteres delimitada por /* e */
14	{	delimitador escopo	próprio lexema
15	return	palavra-chave	próprio lexema
16	i	identificador	sequência de caracteres
17	>	operador relacional	próprio lexema
18	j	identificador	sequência de caracteres
19	?	operador ternário 1	próprio lexema
20	i	identificador	sequência de caracteres
21	:	operador ternário 2	próprio lexema
22	j	identificador	sequência de caracteres
23	}	delimitador escopo	próprio lexema

Obs: as classes poderiam ter outros nomes. Também é possível definir classes mais específicas do que essas.

c) Em qual dos dois programas apresentados nas letras acima (a e b) foram identificados mais tokens?

R: Programa a

2. Faça o papel do analisador léxico e "quebre" os seguintes programas, escritos na linguagem ALGUMA, em um fluxo de tokens, no formato <tipo,valor>, onde tipo é um dos tipos da linguagem ALGUMA, e valor contém o lexema.

a)

```
:DECLARACOES
numero1:INT
numero2:INT
```

```
numero3:INT
aux:INT

:ALGORITMO
% Coloca 3 números em ordem crescente
LER numero1
LER numero2
LER numero3
SE numero1 > numero2 ENTAO
   INICIO
  ATRIBUIR numero2 A aux
  ATRIBUIR numero1 A numero2
  ATRIBUIR aux A numero1
   FIM
SE numero1 > numero3 ENTAO
   INICIO
  ATRIBUIR numero3 A aux
  ATRIBUIR numero1 A numero3
  ATRIBUIR aux A numero1
   FIM
SE numero2 > numero3 ENTAO
   INICIO
  ATRIBUIR numero3 A aux
  ATRIBUIR numero2 A numero3
  ATRIBUIR aux A numero2
   FIM
IMPRIMIR numero1
IMPRIMIR numero2
IMPRIMIR numero3
```

R:

```
<Delim> <Dec> <var,numero1> <Delim> <Int> <var,numero2> <Delim> <Int>
<var,numero3> <Delim> <Int> <var,aux> <Delim> <Int> <Delim> <Alg> <Ler>
<var,numero1> <Ler> <var,numero2> <Ler> <var,numero3> <Se> <var,numero1>
<OpRelMaior> <var,numero2> <Ent> <Ini> <Atr> <var,numero2> <A> <var,aux>
<Atr> <var,numero1> <A> <var,numero2> <Atr> <var,aux> <A> <var,numero1>
<Fim> <Se> <var,numero1> <OpRelMaior> <var,numero3> <Ent> <Ini> <Atr>
<var,numero3> <A> <var,aux> <Atr> <var,numero1> <A> <var,numero3> <Atr>
<var,aux> <A> <var,numero1> <Fim> <Se> <var,numero2> <OpRelMaior>
<var,numero3> <Ent> <Ini> <Atr> <var,numero3> <A> <var,aux> <Atr>
<var,numero2> <A> <var,numero3> <Atr> <var,aux> <A> <var,numero2> <Fim>
<Imp> <var,numero1> <Imp> <var,numero2> <Imp> <var,numero3>
```

b)

```
:DECLARACOES
numero:INT
potencia2:INT
potencia3:INT

:ALGORITMO
```

```
% Ler um numero
LER numero
SE numero = 0 ENTAO % zero elevado a qualquer coisa é zero
   INICIO
  ATRIBUIR 0 A potencia2
  ATRIBUIR 0 A potencia3
   FIM
SENAO SE numero = 1 ENTAO % um elevado a qualquer coisa é um
   INICIO
  ATRIBUIR 1 A potencia2
  ATRIBUIR 1 A potencia3
   FIM
SENAO INICIO
   ATRIBUIR numbero * numero A potencia2
   ATRIBUIR numero * (numero * numero) A potencia33
FIM
% Mostrar resultados
IMPRIMIR numero
IMPRIMIR ' ao quadrado é igual a '
IMPRIMIR potencia3
IMPRIMIRS '\n'
IMPRIMIR numero
IMPRIMIR ' ao cubo é igual a '
IMPRIMIRpotencia3
IMPRIMIR '\n'
```

```
<Delim> <Dec> <var,numero> <Delim> <Int> <var,potencia2> <Delim> <Int>
<var,potencia3> <Delim> <Int> <Delim> <Alg> <Ler> <var,numero> <Se>
<var,numero> <OpRelIgual> <Const,0> <Ent> <Ini> <Atr> <Const,0> <A>
<var,potencia2> <Atr> <Const,0> <A> <var,potencia3> <Fim> <Sen> <Se>
<var,numero> <OpRelIgual> <Const,1> <Ent> <Ini> <Atr> <Const,1> <A>
<var,potencia2> <Atr> <Const,1> <A> <var,potencia3> <Fim> <Sen> <Ini>
<Atr> <var,numbero> <OpAritMult> <var,numero> <A> <var,potencia2> <Atr>
<var,numero> <OpAritMult> <AbrePar> <var,numero> <OpAritMult> <var,numero>
<FechaPar> <A> <var,potencia33> <Fim> <Imp> <var,numero> <Imp> <Cad,' ao
quadrado é igual a '> <Imp> <var,potencia3> <var,IMPRIMIRS> <Cad,'\n'>
<Imp> <var,numero> <Imp> <Cad,' ao cubo é igual a '>
<var,IMPRIMIRpotencia3> <Imp> <Cad,'\n'>
```

c) Os programas anteriores possuem algum erro léxico? Se sim, qual?

R: Não.

3. Analise o código a seguir, na linguagem ALGUMA, e aponte os erros léxicos, se houver (Obs: desconsidere os números de linhas, são apenas para sua referência)

```
1.  :ALGORITMO
2.  // Ler um numero
3.  LER 123numero
4.  LER numero345
5.  LER ENTAO SENAO numero == 0
```

```
6.  FIM
7.  SE var1 ><=> 22 ENTAO
8.    SE var2 != 33 ENTAO
9.  SENAO
10. INICIOFIM
11. INICIO FIM
12. %%%%%%%%%%%%%%% Mostrar resultados
13.
14. :DECLARACOES
15. 123numero:INT
16. Numero345=INT
17. Algoritmo && FLOAT
18. :ALGORITMO
19. ATRIBUIR 123numero A 123numero
```

R: Há uma série de erros sintáticos e semânticos, mas erros léxicos só existem dois:
Linha 8: Erro léxico: não reconhece o lexema "!"
Linha 17: Erro léxico: não reconhece o lexema "&"

Algumas explicações:
Linha 2: OK - "/" é um operador aritmético válido. Reconhece duas vezes
Linha 3: OK - reconhece uma constante numérica 123 e depois uma variável
Linha 5: OK - "=" é um operador relacional válido. Reconhece duas vezes
Linha 7: OK - ">", "<=" e ">" são operadores relacionais válidos
Linha 12: OK - Comentários devem ter pelo menos um símbolo de porcento, o resto é ignorado até o final da linha
Linha 15: OK - mesmo caso da linha 3

4. Quais são os motivos para se separar conceitualmente a análise léxica da sintática? Explique cada um dos motivos.

R:
1. Humanos entendem melhor os programas que seguem nosso modelo de linguagens (palavras/vocabulário vs frases/gramática)
2. Facilita a implementação. Permite dividir o problema em partes menores, dando foco específico em diferentes partes do problema (dividir-para-conquistar)
3. Facilita análise sintática. Análise léxica é um problema mais simples (formalismo de linguagens regulares). Análise sintática, sem se considerar nomes (trabalho do léxico), também é facilitada (formalismo de linguagens livres de contexto).
4. Eficiência. É possível otimizar tarefas de leitura, como por exemplo criar um buffer de entrada.
5. Portabilidade. Permite "reaproveitar" apenas uma das partes. Também faciita a manutenção.

5. Qual o papel do buffer duplo?

R: O buffer duplo tem papel triplo (desculpe-me o trocadilho):
Papel 1. Na análise léxica é necessário ler uma longa sequência de caracteres, geralmente, de um arquivo. Ler os caracteres, um a um, do disco, é menos eficiente do que ler porções maiores e trabalhar na memória.
Papel 2. Na análise léxica é comum uma implementação do tipo tentativa e erro, pois é mais

vantajosa do que uma busca em paralelo por todas as possibilidades de casamento. Assim, cada tentativa errada demanda que haja um retrocesso dos caracteres lidos. Se fosse feita leitura do disco, isso não seria possível facilmente. Já na memória basta fazer um gerenciamento de ponteiros para que isso seja feito facilmente.
Papel 3. No retrocesso, existe um caso onde pode haver perda de informações. Caso uma tentativa ultrapasse o limite superior do buffer (simples), o mesmo é recarregado totalmente. Se for necessário retroceder além do limite inferior do buffer, a informação anterior já não estará mais lá. Por isso, trabalha-se com um buffer duplo, onde cada metade é recarregada de forma alternada. Para isso, no entanto, é necessário cuidado para não recarregar a mesma metade do buffer duas vezes, caso haja a situação de retrocesso além dos limites, como citado acima.

6. Qual a vantagem dos geradores de analisadores léxicos sobre os analisadores construídos à mão?

R: Permite que o programador atue em um nível mais alto de abstração. Com isso, ele pode focar seu trabalho em identificar corretamente os padrões, sem se preocupar com detalhes como retrocesso, buffer duplo, leitura do arquivo, etc. Além disso, hoje em dia os geradores produzem código quase tão eficiente (ou mais eficiente) do que aqueles construídos à mão.

7. Escreva expressões regulares para os conjuntos de caracteres a seguir ou se não for possível escrever uma expressão regular para um determinado conjunto de caracteres, justifique.

a) Cadeias de letras maiúsculas começando e terminando com a (minúsculo).

R:
```
'a' ('A'..'Z')* 'a'
```

b) Cadeias de 0s e 1s com um número par de 0s.

R:
```
('1'? ('0' ('1')* '0')*)*
```

c) Cadeias de 0s e 1s nas quais os 0s ocorrem em pares (um 0 seguido de outro 0).

R:
```
('1'? ('00')*)*
```

d) Cadeias de 0s e 1s compostas por um único 1 rodeado pelo mesmo número de 0s à esquerda e à direita.

R: Não é uma linguagem regular

e) Cadeias de dígitos tais que todos os dígitos ímpares, se ocorrerem, ocorrem antes de todos os dígitos pares (se ocorrerem).

R:
```
('1'|'3'|'5'|'7'|'9')*('0'|'2'|'4'|'6'|'8')*
```

f) Todas as cadeias de letras minúsculas que contêm as cinco vogais em ordem.

R:

```
Consoante: 'b'|'c'|'d'|'f'|...|'z'
Cadeia: Consoante* 'a' Consoante* 'e' Consoante* 'i' Consoante* 'o'
Consoante* 'u' Consoante*
```

g) Comentários, consistindo em uma cadeia cercada por /* e */, sem um */ intercalado

R:

```
'/*' .*? '*/'
Obs: a versão gananciosa abaixo iria consumir '*/' internos, se
houvessem:
'/*' .* '*/'
```

h) Endereços IP

R:

Opção 1. Mais simples, mas reconhece endereços errados como 999.999.999.999

```
fragment SEGMENTO_IP: '0'..'9' '0'..'9' '0'..'9';
IP: SEGMENTO_IP '.' SEGMENTO_IP '.' SEGMENTO_IP;
```

Opção 2. Mais complexa, mas restrita às faixas corretas de valores (obs: SEGMENTO_IP é não determinística)

```
fragment SEGMENTO_IP: ( ('1'('0'..'9')('0'..'9')) |
                        ('2'('0'..'4')('0'..'9')) |
                        ('2''5'('0'..'9')) |
                        (('1'..'9')?('0'..'9')) );

IP: SEGMENTO_IP '.' SEGMENTO_IP '.' SEGMENTO_IP;
```

i) Endereços de email

R: Exemplo traduzido de (http://www.regular-expressions.info/email.html). Acesse o link para ler mais sobre e-mails e expressões regulares.

```
fragment CAR: 'a'..'z' | 'A'..'Z';
fragment DIG: '0'..'9';
fragment PONT1: '.' | '-';
fragment PONT2: PONT1 | '_' | '%' | '+';
EMAIL: (CAR | DIG | PONT2 )+ '@'
       (CAR | DIG | PONT1 )+ '.'
       CAR CAR CAR? CAR?;
```

j) Datas, no formado dd/mm/aaaa

R:

Opção 1. Mais simples, mas reconhece datas erradas como 99/99/9999

```
fragment DIG : '0'..'9';
DATA: DIG DIG '/' DIG DIG '/' DIG DIG DIG DIG;
```

Opção 2. Mais complexa, mas restrita às faixas corretas de valores. Também obriga a existência de dois caracteres no dia/mês e quatro caracteres no ano. Veja que ela não verifica datas inexistentes, como 31/02/2000. Isso é muito complicado de se fazer com expressões regulares, é melhor fazer um tratamento semântico.

```
fragment DIA: ('0'..'2') ('0'..'9') |
              '3' ('0'..'1');
fragment MES: '0' ('0'..'9') |
              '1' ('0'..'2');
fragment ANO: '0'..'9' '0'..'9' '0'..'9' '0'..'9';
DATA: DIA '/' MES '/' ANO;
```

k) Números reais com notação científica (ex: 10.4E13)

R:

```
fragment SIN: '+'|'-';
fragment EXP: 'e'|'E';
fragment DIG: '0'..'9';
NUMREAL2: SIN? DIG* '.'? DIG+ (EXP SIN? DIG+)?;
```