UNIVERSIDADE FEDERAL DE SANTA CATARINA

CONSTRUÇÃO DE COMPILADORES

AUTÔMATO DO ANALISADOR LÉXICO

PATRICK HOECKLER

ARARANGUÁ

OUTUBRO - 2019

Sumário

1. Definição do autômato

O autômato de estados finitos formalmente é definido como uma 5-tupla com os seguintes elementos $(Q, \Sigma, \delta, \mathrm{q}_0, \mathrm{F})$, aonde esses elementos são:

- Q – o conjunto de estados do autômato.
- Σ – o alfabeto, é um conjunto de símbolos de entrada para o autômato.
- δ – uma função de transição que, a partir de um estado e um símbolo de entrada, leva a um outro estado.
- q_0 – um estado inicial, deve ser um elemento que pertence ao conjunto Q.
- F – um conjunto de estados finais aonde $F \subseteq Q$.

Nesta seção serão descritos o alfabeto e os estados finais e inicial do autômato. Na seção 2 serão construídos os diagramas do automato, o que indicará os estados intermediários e as funções de transição do mesmo.

1.1 Definindo o alfabeto

O analisador léxico recebe um arquivo de entrada e fará a leitura do mesmo um byte de cada vez. Sendo assim, o alfabeto deste autômato será o conjunto dos 256 possíveis valores de um byte e mais um símbolo que indica o fim do arquivo.

Uma consideração sobre a escolha desse alfabeto é que, a linguagem de programação em questão só utiliza letras, números e símbolos diversos (todos esses são representados dentro da tabela ASCII) para representar: palavras reservadas, identificadores, operadores e literais. Ou seja, dos 256 possíveis valores de um byte, menos da metade são utilizados para escrever os programs.

Entretanto existe uma exceção, literais do tipo *char* e *string* podem ser criados utilizando valores não usados no resto da linguagem, por exemplo, o literal dado por “Hello World!” causaria um erro léxico, pois o caractere ‘!’ não é um terminal da linguagem. Portanto, para não restringir a declaração de literais *char* e *string*, o alfabeto aceita qualquer valor de byte.

Neste documento o valor do byte lido será mostrado como decimal (0 – 255), ou como o símbolo correspondente na tabela ASCII (por exemplo: o símbolo ‘b’ representa o valor 98). Também serão usados subconjuntos do alfabeto para representar um grupo de símbolos (por exemplo: o subconjunto LETRA é o conjunto de todos os valores que representam letras

maiúsculas e minúsculas). A tabela 1 mostra os subconjuntos que serão utilizados no resto deste documento.

Valor do símbolo	Equivalente na tabela ASCII	Nome do conjunto
48 – 57	0 – 9	NUMERO
65 – 90	A – Z	LETRA
97 – 122	a – z	
1 – 9	caracteres de controle	EXTRA
11 – 12		
14 – 31		
33	!	
35	#	
37	%	
38	&	
63	?	
64	@	
92	\	
94	^	
96	`	
123	{	
124	|	
125	}	
126	~	
127	DEL	
128 – 255	N/A	

Tabela 1 - Subconjuntos do alfabeto

Os subconjuntos LETRA e NUMERO da tabela 1 possuem o significado tradicional de seus nomes, já o subconjunto EXTRA engloba todos os símbolos que não são usados em operadores, palavras reservadas e identificadores da linguagem, exceto os bytes com valor 0, 10 (end of line) e 13 (carriage return) – esses bytes são tratados separadamente pois os mesmos tem relação com o controle do buffer de leitura (byte 0), ou precisam ser ignorados (bytes 10 e 13).

1.2 Definindo os estados finais e inicial

Para este autômato, todo estado final deverá representar um token (terminal) diferente que será retornado pelo analisador léxico. Como a linguagem possui 52 terminais, isso indica que também serão precisos 52 estados finais mais os estados intermediários.

Para não adicionar complexidade desnecessária, e facilitar a compreensão do autômato, o conjunto de 52 terminais serão divididos em subconjuntos, a relação entre nome do terminal, nome do estado final e nome do subconjunto para todos os terminais está dada na tabela 2.

Terminal / Lexema	Estado Final	Subconjunto
write	WRITE	PALAVRA
while	WHILE	
until	UNTIL	
to	TO	
then	THEN	
string	STRING	
repeat	REPEAT	
real	REAL	
read	READ	
program	PROGRAM	
procedure	PROCEDURE	
or	OR	
of	OF	
integer	INTEGER	
if	IF	
for	FOR	
end	END	
else	ELSE	
do	DO	
declaravariaveis	DECLARAVARIAVEIS	
const	CONST	
char	CHAR	
chamaprocedure	CHAMAPROCEDURE	
begin	BEGIN	
array	ARRAY	
and	AND	

Terminal / Lexema	Estado Final	Subconjunto
=	=	SIMPLES
]	]	
[	[	
;	;	
:	:	
,	,	
*	*	
)	)	
(	(	
$	$	
î	VAZIO	
identificador	ID	
>	GT	
>=	GTE	
<	LT	
<=	LTE	
<>	DIF	
+	PLUS	
–	MIN	
/	DIV	
.	DOT	
..	DDOT	
numreal	L_REAL	
numinteiro	L_INT	
nomestring	L_STR	
nomechar	L_CHAR	

Tabela 2 - Divisão dos terminais

O subconjunto PALAVRA da tabela 2 contém todas as palavras reservadas da linguagem. Já o subconjunto SIMPLES contém todos os terminais com apenas 1 caractere, que são trazidos

do estado inicial ao final apenas com a leitura do mesmo, e que, uma vez no estado final, não existe como sair do mesmo.

O estado final VAZIO indica a leitura de uma palavra vazia, portanto este também é o estado inicial do autômato. Os estados finais para comentários são: CL para comentário de linha; e CB para comentário de bloco; a seção 2.3 mostra como estes serão usados.

2. Construíndo o diagrama do autômato

Nesta seção o diagrama do automato será construído em partes e depois essas serão unidas para fazer o diagrama final.

2.1 Automato para palavras reservadas

O funcionamento desse autômato é muito simples, os caracteres serão lidos um a um e mudarão para estados intermediários conforme isso acontece, até a palavra ser completada e chegar ao estado final. A figura 1 mostra como o autômato para reconhecer a palavra reservada *write* funciona.

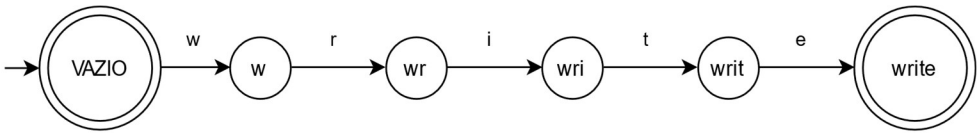

Figura 1 - Automato para reconhecer a palavra *write*

Apesar do funcionamento desse autômato ser simples, problemas começam a aparecer ao considerar o analisador léxico inteiro. Um dos problemas é o grande número de palavras reservadas, se o diagrama fosse feito adicionando estados intermediários da maneira mostrada na figura 1 para cada palavra reservada, seria muito difícil entender o funcionamento deste autômato pois existiriam centenas de estados.

Outro problema é que cada estado intermediário, na verdade não gera um erro léxico, pois a palavra seria considerada um identificador. Sendo assim, cada estado intermediário seria o estado final ID, aumentando ainda mais a confusão do diagrama.

Considerando estes problemas, fica claro que é necessário trocar o modo de representação do autômato. A figura 2 é um modo mais compacto de representar a mesma coisa, utilizando palavras inteiras para indicar a transição.

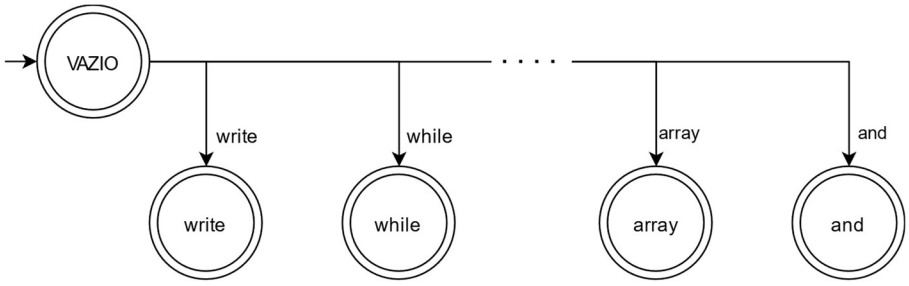

Figura 2 - Automato para reconhecer palavras reservadas

O autômato da figura 2 é mais compacto do que o da figura 1, entretanto ainda assim é muito grande considerando a quantidade de palavras reservadas. Por esse motivo, a seção 1 se focou em definir os subconjuntos de estados e símbolos – utilizando o subconjunto PALAVRA, é possível reduzir o diagrama considerávelmente e não prejudicar o entendimento do autômato. A figura 3 mostra o diagrama reduzido.

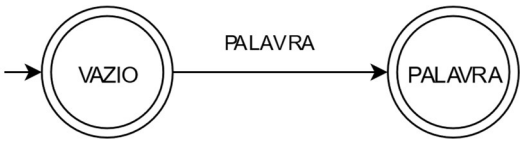

Figura 3 - Autômato para reconhecer palavras reservadas reduzido

A figura 3 mostra que qualquer lexema de entrada que pertence ao conjunto PALAVRA, levará ao estado final correspondente que também pertence a PALAVRA.

É importante ressaltar que, quando está sendo mencionado reduzir o autômato, a intenção é apenas reduzir o diagrama de modo a facilitar a interpretação do mesmo. O autômato em si não está sendo alterado, pois o mesmo ainda lerá byte por byte para chegar ao estado final.

2.2 Autômato para símbolos simples e símbolos extras

Esse autômato reconhecerá os terminais do conjunto SIMPLES. Como os lexemas desse conjunto possuem apenas 1 caractere cada, seria possível fazer um autômato similar ao da figura 2. Entretanto, pelos mesmos motivos dados para o caso anterior é possível reduzir o diagrama desse autômato e chegar no resultado mostrado na figura 4.

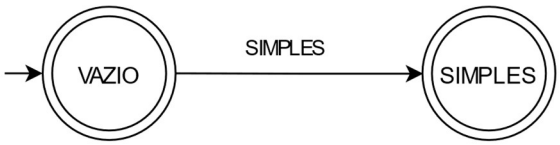

Figura 4 - Autômato para reconhecer símbolos simples

Uma coisa importante a mencionar sobre o funcionamento do autômato do analisador léxico é que: um programa escrito nessa linguagem terá mais do que um único token, entretanto o analisador não saberá dizer aonde um token acaba e outro começa. Por exemplo, suponha que deseja-se analisar a seguinte sequência de caracteres: *=2*2*. Essa sequência deveria ser interpretada como os 4 tokens: = INT * INT. Entretanto, seguindo o autômato da figura 4, ao ler o símbolo = o autômato chegaria no estado final de mesmo nome, mas continuaria lendo o próximo

caractere, isso o levaria a um estado de erro pois não existe nenhuma função que leva do estado = para um outro estado utilizando o caractere 2.

O que foi descrito não é um problema do autômato, e na verdade pode ser utilizado para descobrir onde um lexema acaba e outro começa. Para fazer isso, todos os autômatos mostrados nesse documento seguem a seguinte regra: se o autômato estiver em um dado estado S e ler um símbolo α, aonde não existe uma função de transferência que sai de S pelo símbolo α; ou mais formalmente, se não existir nenhum $\delta(S, \alpha) = S_0 \in Q$, então o símbolo α faz parte do próximo lexema. Ao acabar de ler o lexema, se o estado S for um estado final, então o lexema representará o token dado por S, se não for estado final, então aí sim será considerado erro léxico.

Em outras palavras, se não existir uma transição com o símbolo lido no estado atual, então o lexema acabou, e o estado atual dirá se o lexema é token (estado final) ou erro léxico (estado intermediário). Nunca uma falta de transição entre estados é considerada erro léxico, a mesma apenas indica o fim de um lexema.

Sabendo disso é preciso adicionar ao autômato da figura 4 um estado que gere erro léxico caso se inicie um lexema com símbolos inválidos (dados pelo conjunto EXTRA), a figura 5 mostra essa adição.

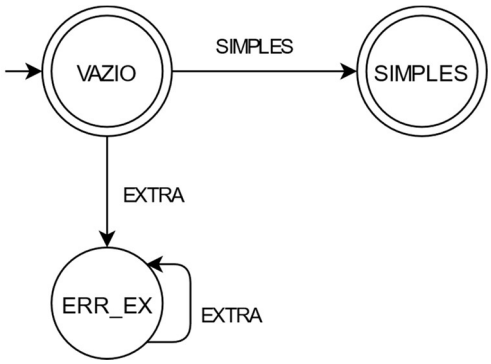

Figura 5 - Automato que reconhece símbolos simples e símbolos inválidos

2.3 Autômato para comentários

O diagrama desse autômato é mostrado na figura 6.

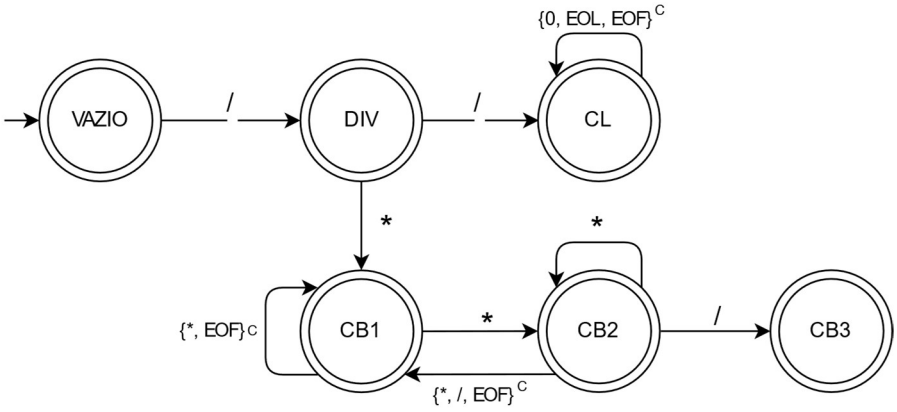

Figura 6 - Autômato que reconhece comentários

Os valores EOL e EOF representam fim de linha e fim de arquivo respectivamente, as chaves {} indicam um conjunto de possíveis símbolos, por exemplo, o conjunto dado por {EOL, EOF} indica que existe uma transição se for lido um símbolo EOL ou EOF. Por último, a letra C em superscrito indica o complemento do conjunto, por exemplo, o conjunto $\{*/, EOF\}^C$ contém qualquer coisa que não é o elemento EOF, ou a sequência de caracteres */.

2.4 Autômato para identificadores

A figura 7 mostra esse autômato.

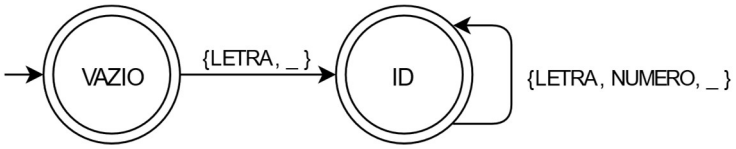

Figura 7 - Autômato que reconhece identificadores

Vale lembrar que nenhuma palavra reservada irá levar ao estado ID (elas seguirão a transição vista na figura 3). Como explicado anteriormente, o diagrama é simplificado para ser possível entender o funcionamento do autômato, mas sem mudar o seu funcionamento.

2.5 Autômato para números inteiros e reais

Esse autômato é representado na figura 8 a seguir.

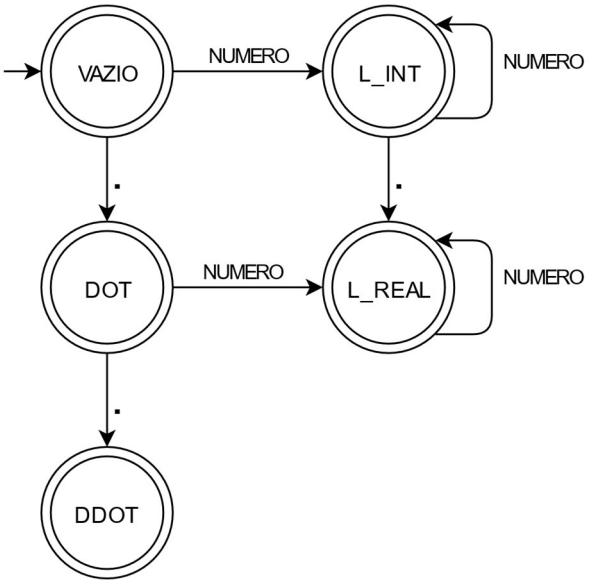

Figura 8 - Autômato que reconhece números

2.5 Autômato para char e string

Para esta linguagem, deseja-se poder usar sequências de escape para representar bytes que não podem ser representados normalmente em literais *char* e *string*, por exemplo, nessa linguagem esse tipo de literal nao pode ter uma quebra de linha em seu valor. Sendo assim, sequências de escape são usadas para substituir por estes valores, como por exemplo: “Hello World\n”.

O caractere que indica uma sequência de escape é a barra inversa (\ ), o próximo caractere lido vai determinar o byte resultante dessa sequência. A tabela 3 mostra as sequências suportadas e os valores de byte correspondentes.

Sequência	Byte	Byte em ASCII
\0	0	NUL
\a	7	BEL
\b	8	BS
\t	9	TAB
\n	10	EOL
\v	11	VT
\f	12	FF
\r	13	CR
\e	27	ESC
\”	34	”
\’	38	’
\\	92	\

Tabela 3 - Sequências de escape suportadas

A figura 9 mostra o diagrama desse autômato, aonde a palavra BYTE indica o conjunto de todos os 256 possíveis valores de um byte exceto por: {0, 10, 13, \, “, ‘}. Além desta, a palavra ESCAPE indica todos os caracteres que fazem parte de alguma sequência de escape, conforme visto na tabela 3.

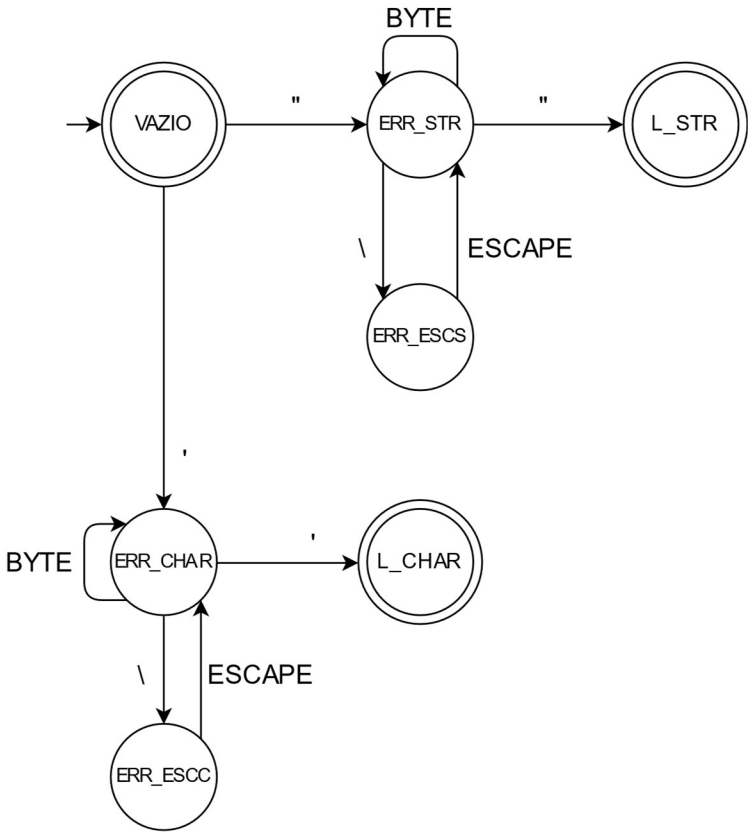

Figura 9 - Autômato que reconhece char e string

2.6 Autômato para comparadores

O diagrama desse autômato é mostrado na figura 10, lembrando que o comparador de igualdade já é reconhecido pelo autômato de símbolos simples (figura 5).

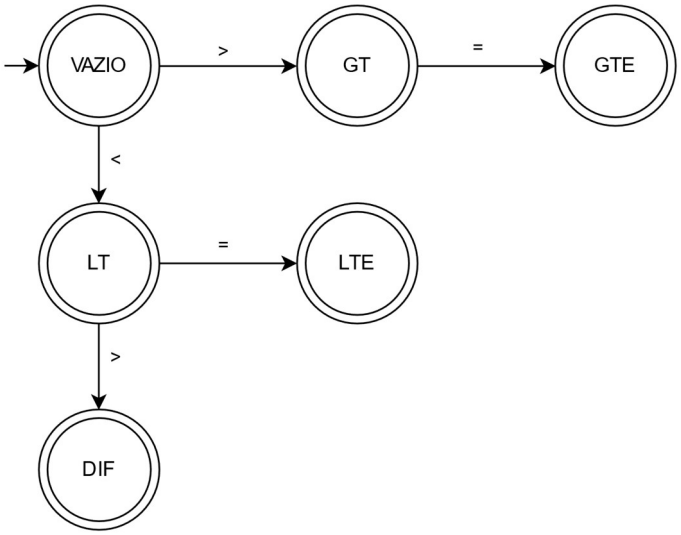

Figura 10 - Autômato que reconhece comparadores

2.7 Autômato para operadores aritméticos

Esse autômato é representado na figura 11, lembrando que os operadores de multiplicação (*) e divisão (/) já foram reconhecidos pelos autômatos de símbolos simples (figura 5), e de comentários (figura 6), respectivamente.

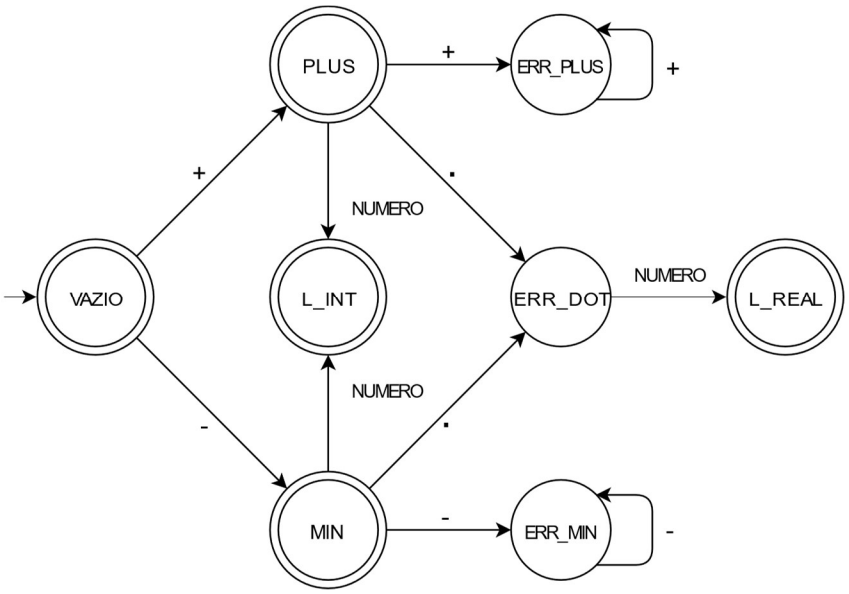

Figura 11 - Autômato que reconhece operadores aritméticos

Como é desejado reconhecer inputs aonde existe um número com um sinal na frente (por exemplo: -3, +4.5) como apenas um token de número, esse autômato possui estados finais que retornam INT ou REAL.

2.8 Autômato do analisador léxico

Por fim, basta combinar todos os autômatos vistos anteriormente para obter o autômato completo que será usado no analisador léxico. A figura 12 mostra o resultado final, foi adicionado uma transição do estado vazio para si mesmo a fim do analisador ignorar espaços em branco (ESPAÇO), quebras de linha (EOL), carriage return (CR), tabuladores (TAB) e o valor zero.

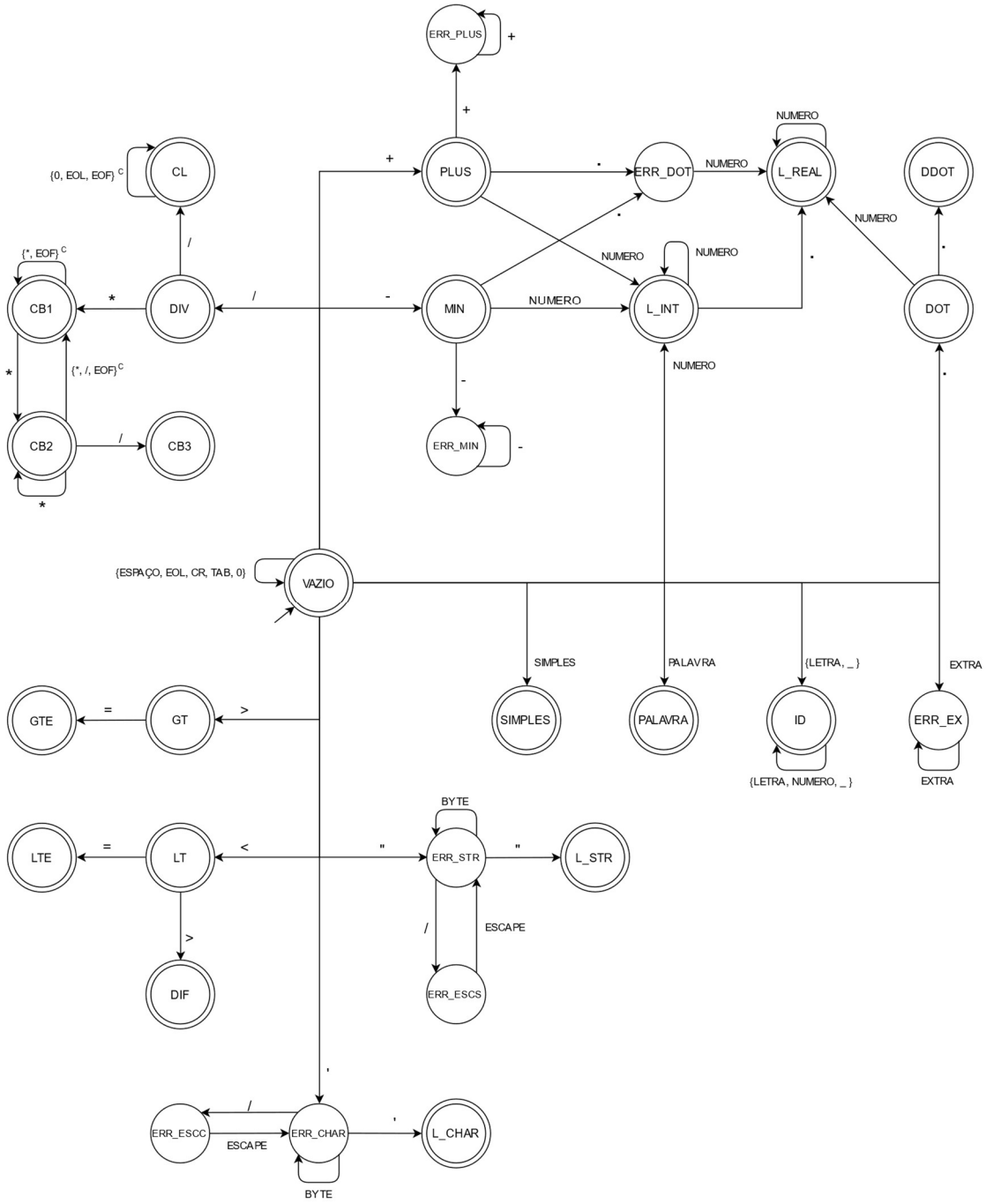

Figura 12 - Autômato completo do analisador léxico