PATRICIA BRENNY RIBEIRO – 2021018870
ANDRÉ LUIZ MELO DOS SANTOS FRANCO – 2021012740
MATEUS ALEXANDRE MARTINS DE SOUZA – 2021004023

DEFINIÇÕES FORMAIS DA LINGUAGEM GRACE

Projeto entregue à prof. Thatyana Faria Piola Seraphim como requisito parcial para aprovação na disciplina de Compiladores.

UNIVERSIDADE FEDERAL DE ITAJUBÁ
2023

"Qualquer tolo consegue escrever código que um computador entenda. Bons programadores escrevem código que humanos possam entender."

- Martin Fowler

PREFÁCIO

Este projeto inclui a definição inicial da análise sintática e léxica de uma linguagem de programação estruturada, com possibilidade de expansão ao paradigma orientado a objetos. O nome "Grace" é tanto uma homenagem a Grace Murray Hopper, pioneira de programação que em 1952 escreveu o primeiro compilador, destinado a linguagem A-0, quanto um trocadilho com a o significado de "grace". Grace, do inglês, pode ser interpretado como elegância, característica de procedimento que revela cortesia, distinção; decoro, fineza, gentileza. Para muitos, programação de computadores deve ser feita, justamente, com elegância.

1. Expressões Regulares e Tokens:

Abaixo estão definidos alguns Tokens e expressões regulares mais simples que serão utilizadas na linguagem. As definições de VAR, REAL, INT, STRING e CHAR incluem autômatos dedicados na sessão 2 dedicada aos autômatos e gramáticas. Podemos observar a intenção de uma linguagem narrativa, mas simplificada, com uso de símbolos curtos comumente utilizados pelos programadores em outras linguagens populares.

Os operadores e símbolos especiais ganharam subseção própria, entretanto, o foco desse trabalho inicial é o reconhecimento das cadeias, ou seja, as etapas de análises léxicas e sintáticas do compilador, sem detalhes muito extensos de análise semântica, como verificações de tipos, nem geração e otimização de código.

1.1. Definições gerais:

BEGINPROGRAM	begin
ENDPROGRAM	end
LETTER	a-z|A-Z
DIGIT	0-9
ENDLINE	\n
VAR	LETTER DIGIT * LETTER
REAL	- ? DIGIT+ . DIGIT+
INT	-? DIGIT+
STRING	“ (ASCII)* ”
CHAR	‘ASCII’
BOOLEAN	1 | 0
LITERAL	INT| STRING| CHAR| REAL | BOOLEAN
TYPE	real | int | bool | char | string
ASSIGN	<-
RETURNTYPE	->
FUNC	func
REPEAT	repeat
WHILE	when
IF	is
ELSE	otherwise
RETURN	return

Tabela 1: Definições gerais

Nesta seção vale um ponto de atenção para a definição geral de relacional, logico e operadores, pois na seção de autômatos serão definidos o reconhecimento de operações aritméticas, lógicas e relacionais, inclusive em cadeia. Não se deve confundir os tokens para o reconhecimento léxico

(símbolos conforme estrutura do programa fonte), e os tokens para o reconhecimento sintático (sequencias de tokens léxicos válidas para operações diversas), apesar de terem nomes similares.

1.2. Operadores aritméticos, relacionais e lógicos:

SUM	+
SUBTRACTION	-
MULTIPLICATION	*
DIVISION	/
EXPONENTIATION	^
RESTDIV	%
RELATIONAL	{>, <, <=, >=, !=, ==}
LOGICAL	{ && , || }
NEG	!
OPERATION	SUM| SUBTRACTION | MULTIPLICATION | DIVISION | EXPONENTIATION| RESTDIV

Tabela 2: Definições de símbolos operacionais.

1.3. Símbolos especiais:

OPENPAR	(
CLOSEPAR	)
OPENBR	[
CLOSEBR	]
OPENCUR	{
CLOSECUR	}
SEPARATOR	,
DECIMAL	.
COMENTARY	## (.*) ##
COLON	:
APOSTROPHE	‘
QUOTATION	“
PRINT	write
SCAN	read
QUESTION	?

Tabela 3: Definições especiais.

2. **Autômatos:**

Abaixo estão definidos os autômatos e gramáticas para o reconhecimento léxico e/ou sintático das estruturas da linguagem, cada autômato/expressão está atribuído a um novo token, e incluem uma breve descrição de funcionamento.

2.1.Variáveis

VAR são as variáveis do programa, aceitas tanto para nomear sub-rotinas, funções, o programa principal, quanto para ponteiros e variáveis comuns.

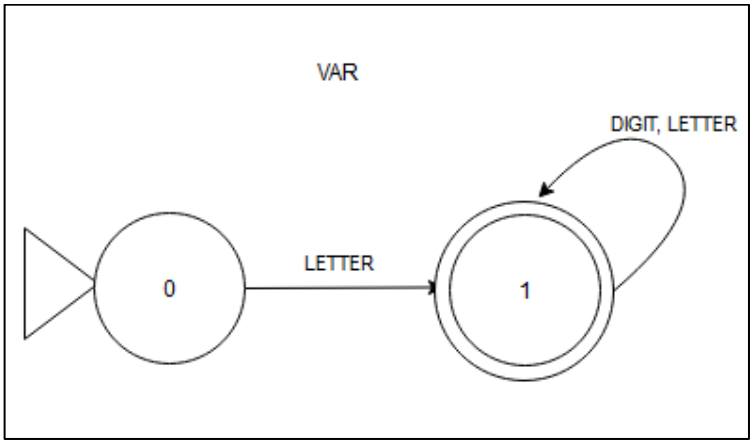

Figura 1: Variáveis.

2.2.Tipos de Dados:

Os autômatos abaixo são os reconhecedores léxicos para identificar tipos de dados suportados pela linguagem, são aceitos caracteres, strings, inteiros, números reais, ponteiros.

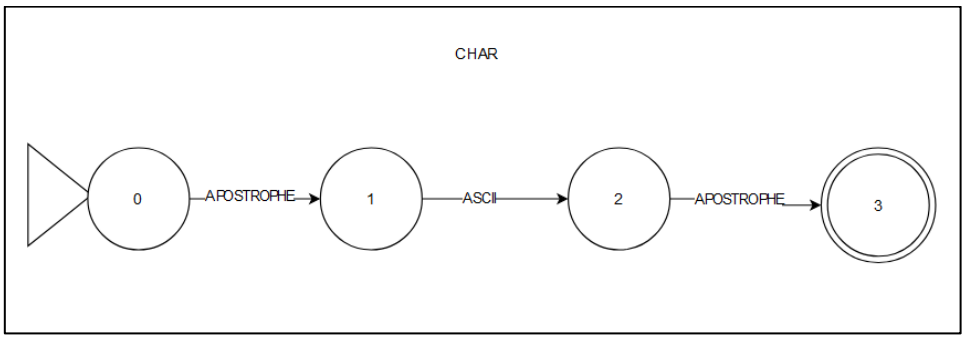

Figura 2: Characteres.

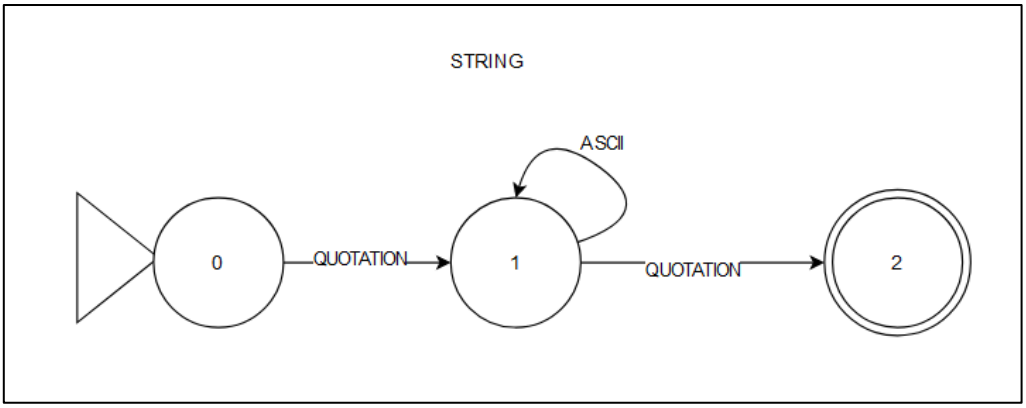

Figura 3: Strings.

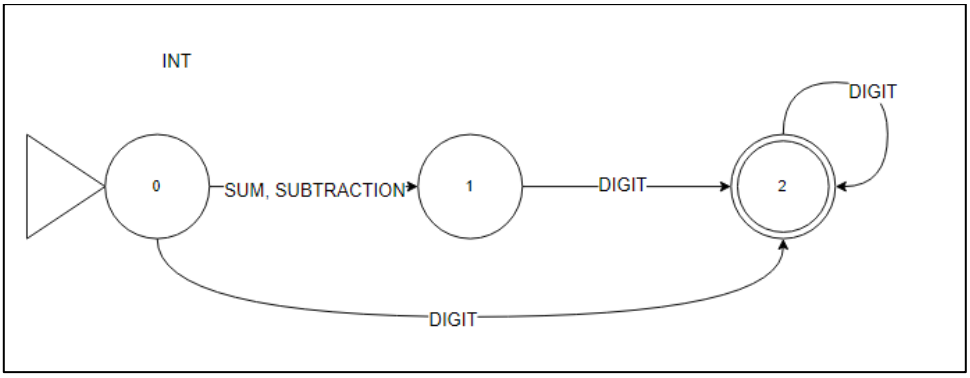

Figura 4: Inteiros

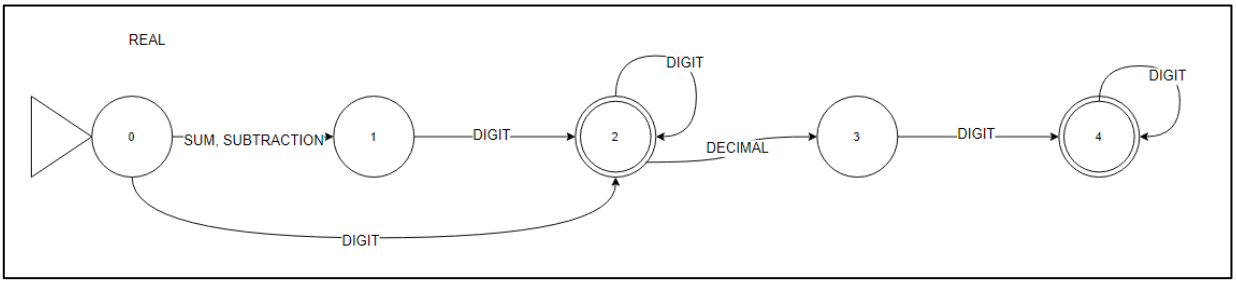

Figura 5: Reais

2.3.Declaração e Atribuição

Os autômatos abaixo são os reconhecedores sintáticos para declaração e atribuição de variáveis, podendo ser realizada a atribuição no mesmo comando que a declaração.

DECLARARE-NOASSIGN -> VAR COLON TYPE

DECLARE-ASSIGN-> VAR COLON TYPE (ASSIGN [LITERAL|FUNCCALL|EXP| SCAN| VAR])?

DECLARATION -> DELCARE-ASSIGN| DECLARE-NOASSIGN

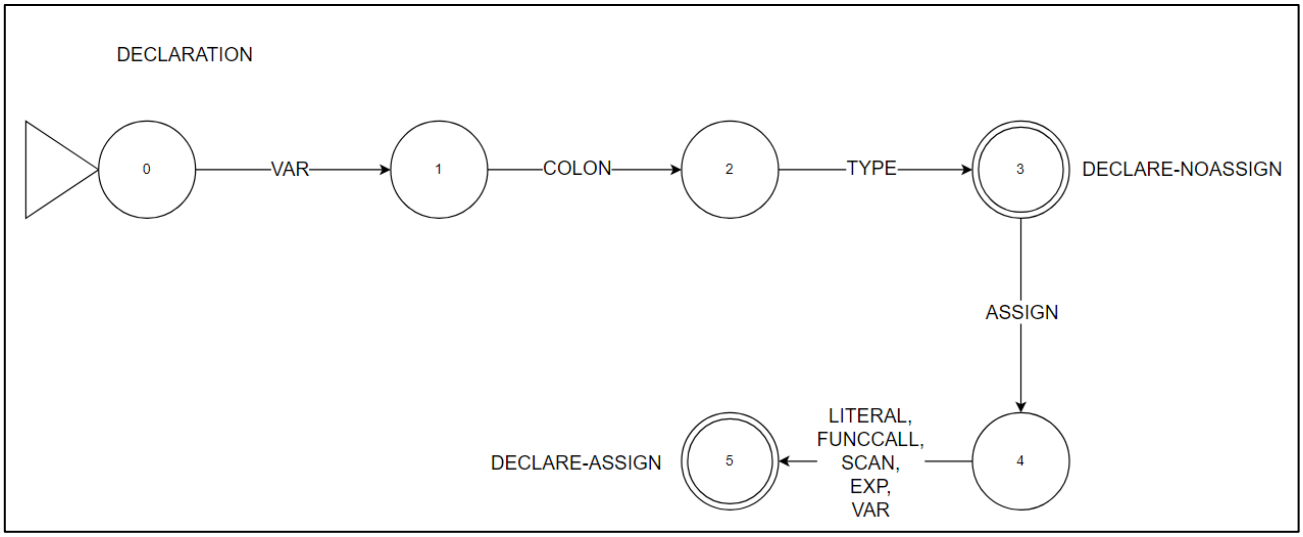

Figura 6: Declaração

ASSIGNMENT-> VAR ASSIGN [LITERAL|FUNCCALL| SCAN| EXP|VAR]

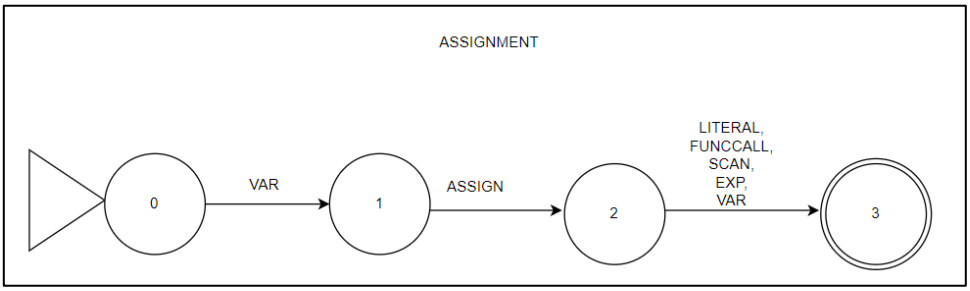

Figura 7: Atribuição

2.4.Expressões e Operações Diversas

Os autômatos abaixo são os reconhecedores sintáticos para as expressões gerais, ou seja, operações aritméticas, logicas e relacionais, essenciais para as estruturas de loops e condicionais. As expressões aritméticas e logicas permitem uma cadeia de associações, ou seja, que na mesma expressão sejam realizadas diversas operações.

EXP -> ARITHMETIC | LOGIC | RELATION

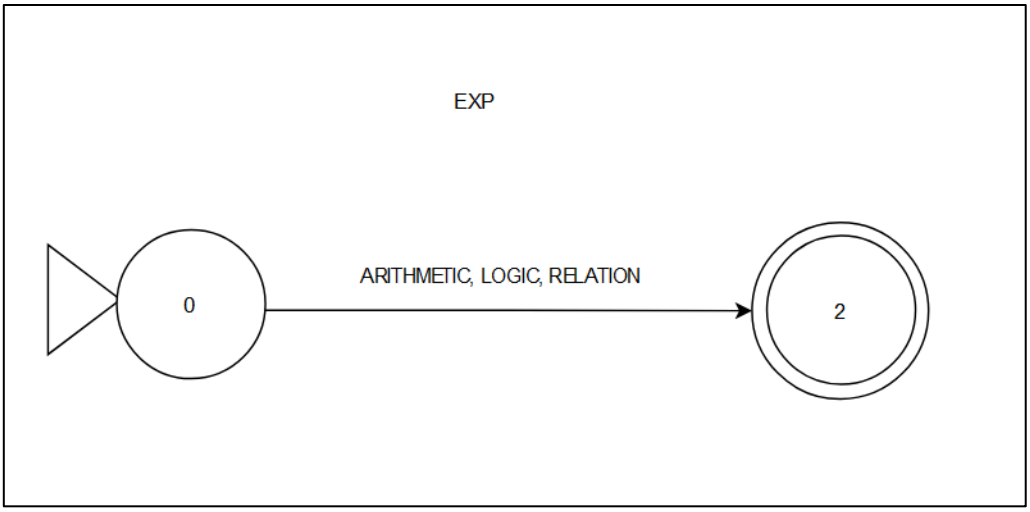

Figura 8: Expressões.

LOGIC -> NEG[[(RELATION|LOGIC) LOGICAL NEG*(RELATION|LOGIC)] | VAR]*

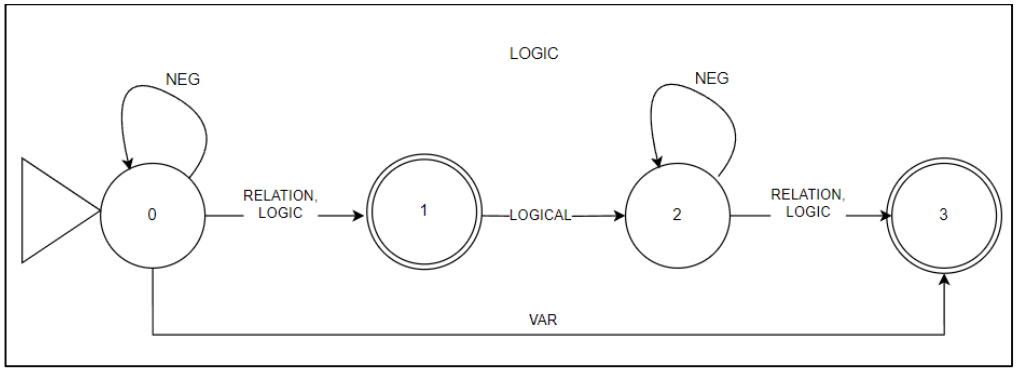

Figura 9: Operações lógicas.

ARITHMETIC -> [VAR| LITERAL|ARITHMETIC] OPERATION [VAR|LITERAL|ARITHMETIC]

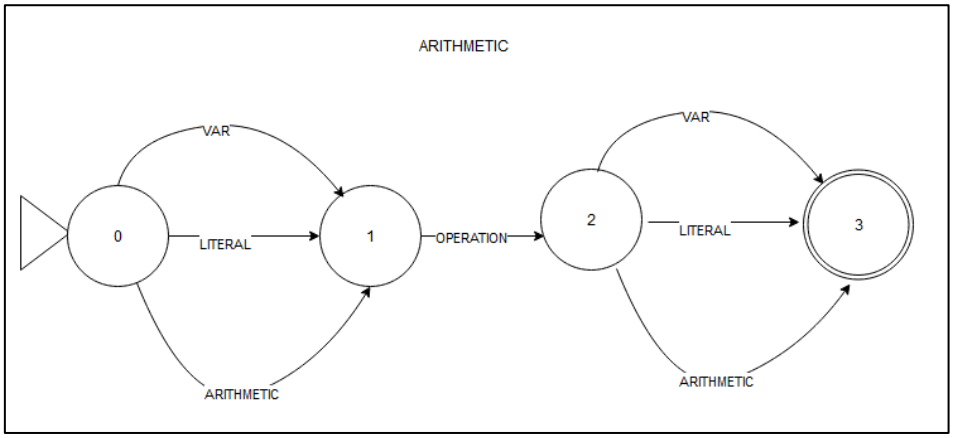

Figura 10: Operações aritméticas.

RELATION -> (VAR|LITERAL) RELATIONAL (VAR|LITERAL)

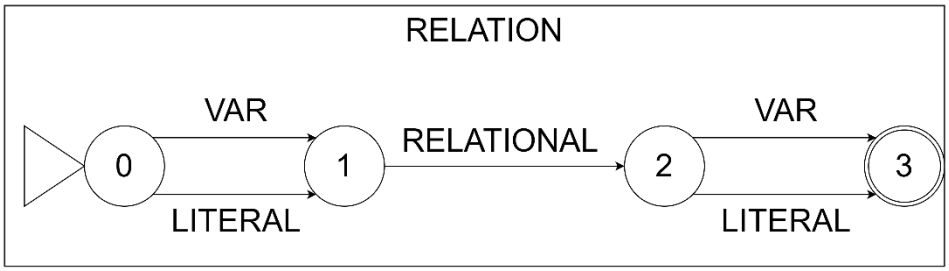

Figura 11: Operações relacionais.

2.5.Declaração e Acesso a Vetores.

Os autômatos abaixo são os reconhecedores sintáticos para a declaração e o acesso a vetores sendo obrigatório determinar o tamanho do vetor, e não sendo possível atribuir um vetor literal ou não literal diretamente, para essa função utilize ponteiros.

VECTOR -> DECLARE-NOASIGN (OPENBR INT CLOSEBR)+

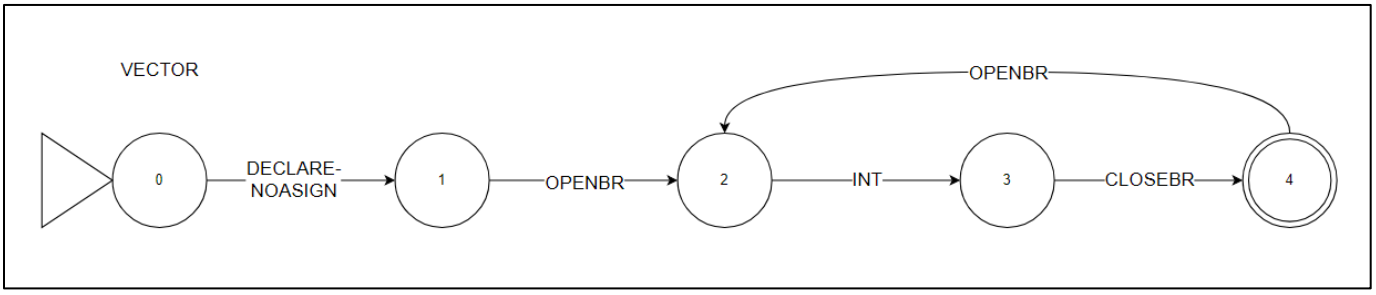

Figura 12: Declaração de vetores

VECACCESS -> VAR (OPENBR INT CLOSEBR)+

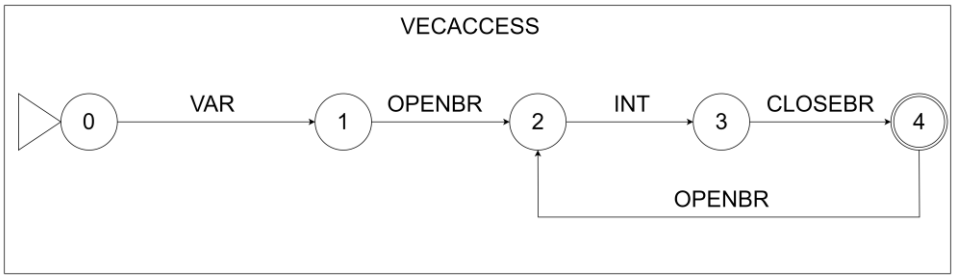

Figura 13: Acesso a vetores

2.6.Repetições e Condicionais.

Os autômatos abaixo são os reconhecedores sintáticos para repetições e condicional, existindo opções para else if diretos, não sendo necessário uso aninhado de condicionais nesse caso.

LOOP -> WHEN LOGIC REPEAT COLON COMMAND+

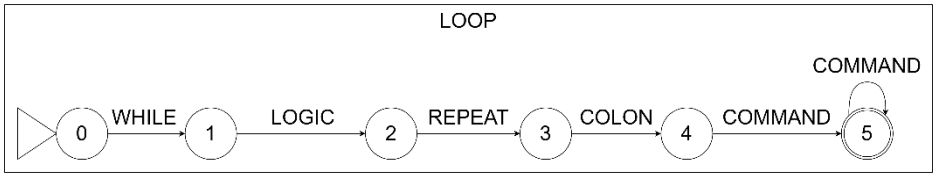

Figura 14: Repetições.

CONDITIONAL -> IF LOGIC QUESTION COMMAND+ [ELSE [CONDITIONAL] | [COLON COMMAND+]]?

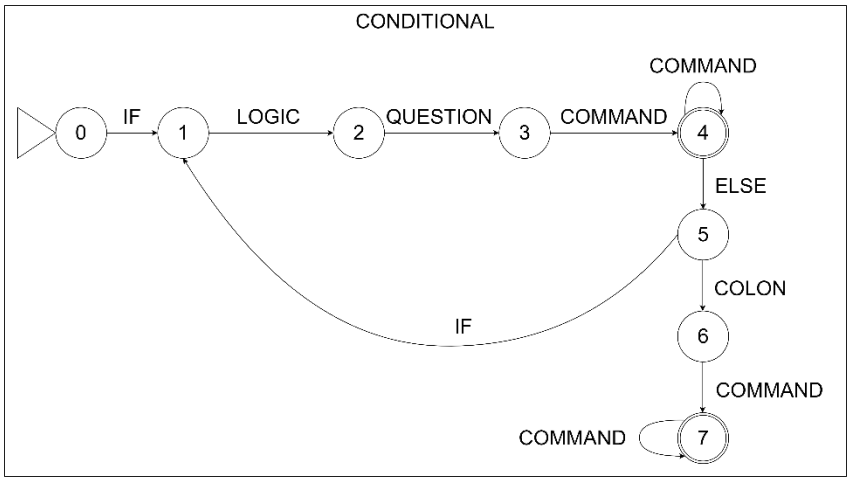

Figura 15: Condicionais.

2.7.Declaração e Chamadas de Funções.

Os autômatos abaixo regem a declaração, chamada e retorno de funções e sub-rotinas.

A declaração de funções permite que a função não tenha parâmetros ou tenha parâmetros, que podem ser atribuídos com um valor para se tornarem parâmetros padrão. Os parâmetros padrão devem ser declarados por último para haver a possibilidade de serem omitidos em uma chamada de função.

FUNCDECLARATION -> FUNC VAR OPENPAR [DECLARE-NOASIGN (SEPARATOR DECLARE-NOASIGN)]? [DECLARE-ASIGN (SEPARATOR DECLARE-ASIGN)*]? CLOSEPAR [(RETURNTYPE TYPE OPENCUR COMMAND* FUNCRETURN)|(OPENCUR COMMAND*)] CLOSECUR*

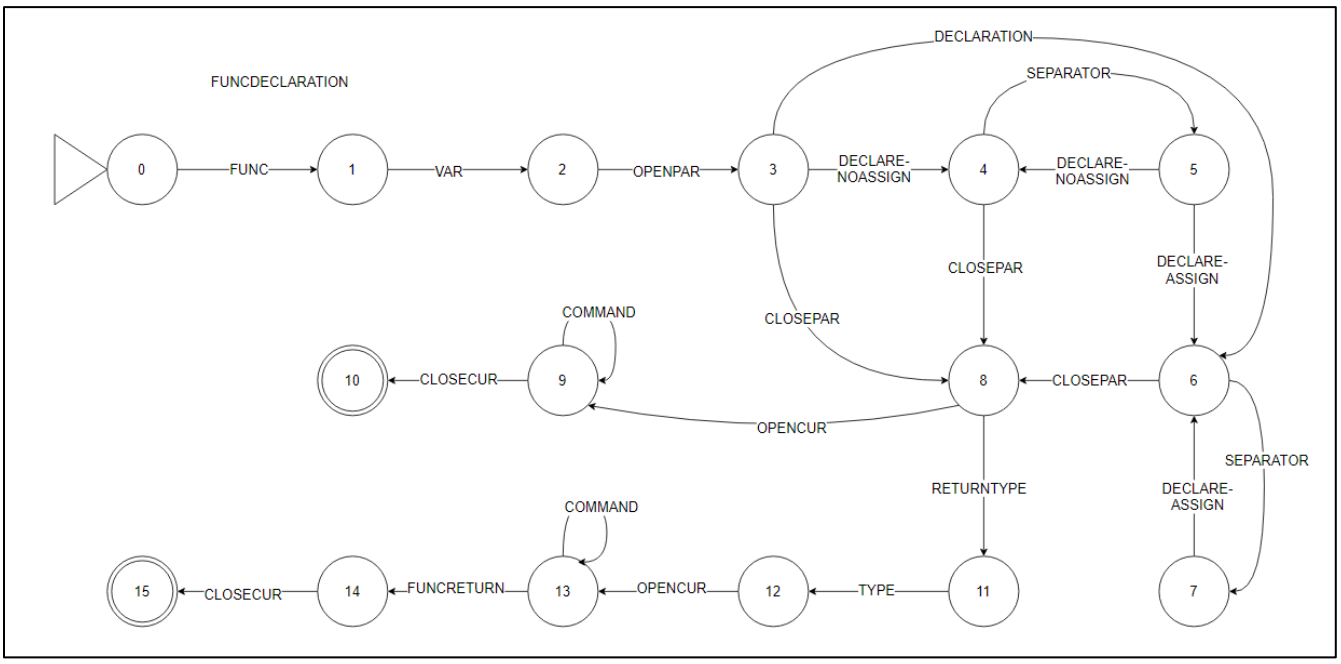

Figura 15: Declaração de funções.

FUNCRETURN -> RETURN (EXP|VAR|LITERAL)

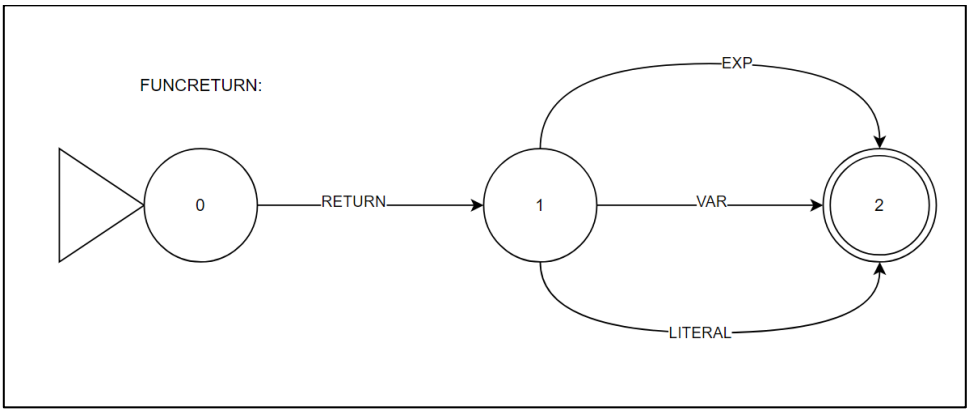

Figura 16: retorno de funções.

FUNCCALL -> VAR OPENPAR [VAR+ [SEPARATOR VAR+]]? CLOSEPAR*

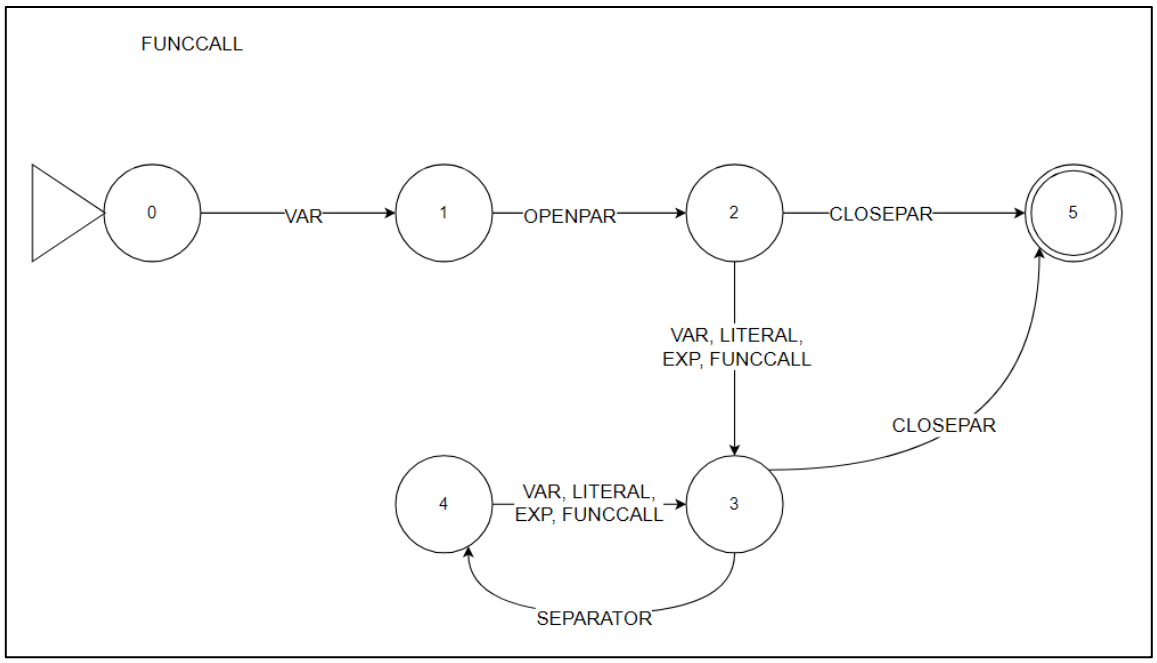

Figura 17: Chamada de funções.

2.8.Programa Principal e Comandos

Os autômatos abaixo regem a estrutura do programa principal e dos comandos gerais, funções devem ser declaradas externamente ao programa principal e apenas invocadas nesta estrutura central, que aceita, também, comentários como os definidos na primeira tabela de definições gerais do projeto. Comentários e comandos são essencialmente diferentes nesta linguagem.

PROGRAM -> BEGINPROGRAM VAR COMMAND| COMMENTARY ENDPROGRAM FUNCTION**

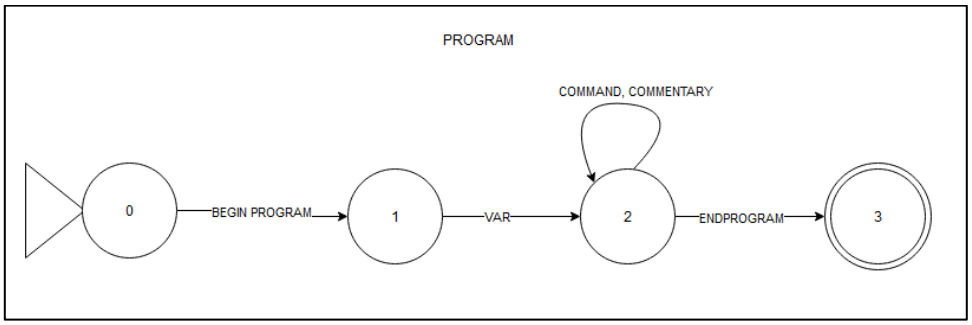

Figura 18: Programa principal.

COMMAND -> EXP | VECTOR | VECACESS | FUNCCALL | CONDITIONAL | LOOP | DECLARE-ASSIGN | DECLARE-NOASSIGN | ASSIGN

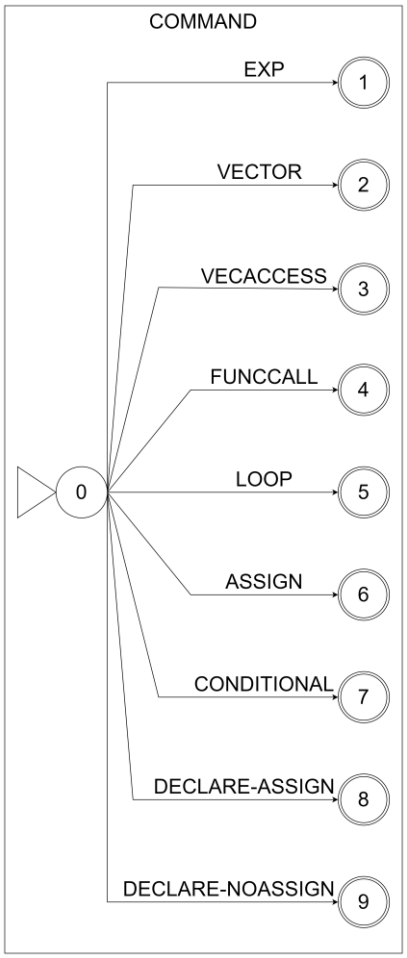

Figura 18: Comandos.

2.9.Leitura e escrita

Abaixo estão descritas as gramáticas e os autômatos de leitura do teclado e escrita na tela, lembrando que também estão inclusas nas definições de DECLARATION e ASSIGNMENT o comando de leitura como comando válido, sendo aqui especificado apenas o READ separadamente, mas levar em consideração todo o conjunto é importante.

READ -> [VAR| VAR COLON TYPE] ASSIGN SCAN

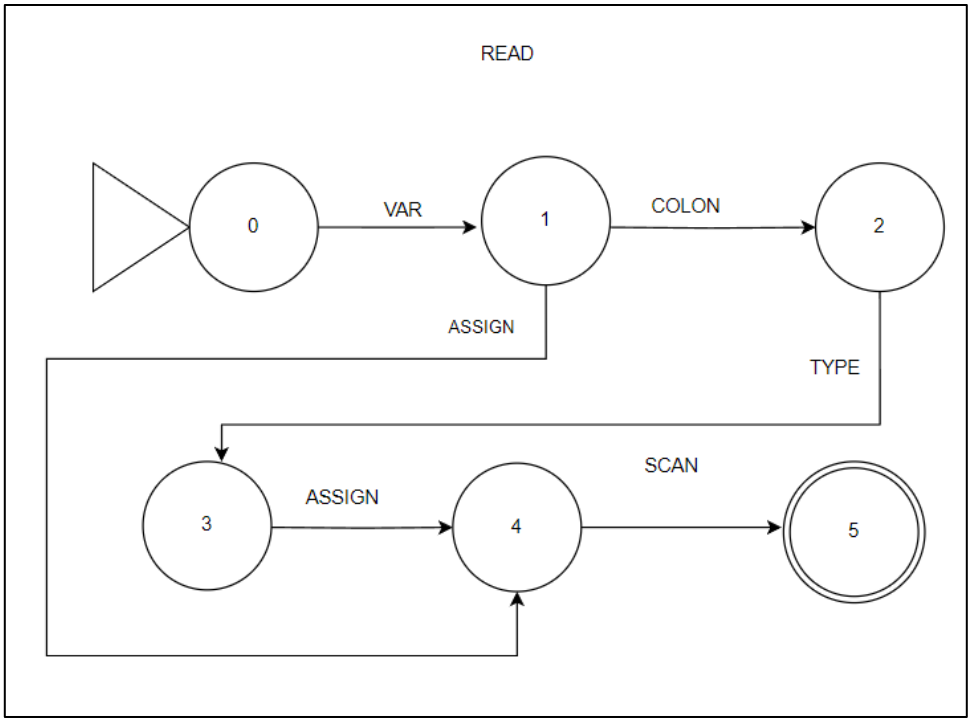

Figura 19: Leitura.

WRITE -> PRINT [VAR| LITERAL| FUNCCALL | EXP]*

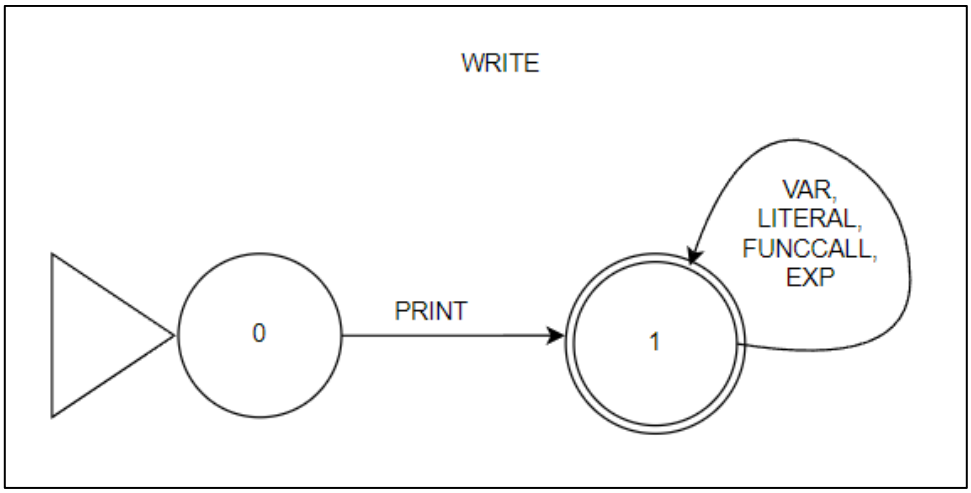

Figura 20: Escrita.

3. Exemplos:

Abaixo está o exemplo de dois programas escritos em Grace, um deles trabalha vetores, outro trabalha funções, todos trabalham declaração, leitura e escrita.

```
func helloWorld() -> string
{
    return "Hello World!\n"
}

func getAge(birthdate:int) -> string
{
    return 2023 - birthdate
}

begin main
    write helloWorld()

    name: string
    birthdate: int

    write "Type your name: "
    name <- read

    write "Type your age: "
    birthdate <- read

    write "Hello " name  "!\n"
    write "You are " getAge(birthdate) " years old.\n"

end
```

```
begin vectorProgram
    ## accessing and using vectors ##

    temp : real[5]
    sum : real <- 0

    i : int <- 0

    when i < 5 repeat:
        write "#" i " Type a tempeature value: "
```

```
        temp[i] <- read
        sum = sum + temp[i]
        i <- i+1

    avg : real <- sum / 5

    write "Average temperatures = " avg  "\n"

    is avg > 25?
        write "Very hot!\n"
    otherwise:
        write "Nice and chill!\n"
end
```