Projeto da Linguagem

Definição da Gramática Livre de Contexto

A gramática da linguagem é representada pela seguinte quádrupla:

G =(V, T, P, S)

em que:

- **T** = conjunto dos símbolos terminais(listados abaixo)
- **V** = conjunto dos símbolos não terminais(listados abaixo)
- **P** = conjunto de produções(especificado abaixo)
- **S** = INICIO

Símbolos Terminais

- function
- (
-)
- int
- char
- float
- se
- entao
- senao
- enquanto
- faca
- repita
- ate
- =
- op_rel
- +
- -
- *
- /
- ^
- {
- }
- ;
- ,
- :
- ID
- CONST_CHAR
- CONST_NUM

Símbolos Não Terminais

- INICIO
- BLOCO
- DECLARACOES_VARS
- SEQ_COMANDOS
- DECLARACAO_VAR
- TIPO
- LISTA_IDS
- LISTA_IDS'
- COMANDO
- CMD_ATRIB
- CMD_REPET1
- CMD_REPET2
- CMD_SELECAO
- CMD_OU_BLOCO
- EXPRESSAO
- TERMO
- EXPONENC
- FATOR
- COND

Produções da Gramática(usando notação BNF)

```
<INICIO>              ::= function ID () <BLOCO>
<BLOCO>               ::= {<DECLARACOES_VARS><SEQ_COMANDOS>}
<DECLARACOES_VARS>    ::= <DECLARACAO_VAR><DECLARACOES_VARS> | ε
<DECLARACAO_VAR>      ::= <TIPO>:<LISTA_IDS>;
<TIPO>                ::= int | float | char
<LISTA_IDS>           ::= ID<LISTA_IDS'>
<LISTA_IDS'>          ::= ,ID<LISTA_IDS'> | ε
<SEQ_COMANDOS>        ::= <COMANDO><SEQ_COMANDOS> | ε
<COMANDO>             ::= <CMD_ATRIB> | <CMD_REPET1>
                        | <CMD_REPET2> | <CMD_SELECAO>
<CMD_ATRIB>           ::= ID = <EXPRESSAO>;
<CMD_REPET1>          ::= enquanto (<COND>) faca <CMD_OU_BLOCO>
<CMD_REPET2>          ::= repita <CMD_OU_BLOCO> ate (<COND>)
<CMD_SELECAO>         ::= se (<COND>) entao <CMD_OU_BLOCO>
                        | se (<COND>) entao <CMD_OU_BLOCO> senao <CMD_OU_BLOCO>
<CMD_OU_BLOCO>        ::= <COMANDO> | <BLOCO>
<COND>                ::= <EXPRESSAO> op_rel <EXPRESSAO>
<EXPRESSAO>           ::= <EXPRESSAO> \+ <TERMO> | <EXPRESSAO> - <TERMO> | <TERMO>
<TERMO>               ::= <TERMO> \* <EXPONENC> | <TERMO> / <EXPONENC> | <EXPONENC>
<EXPONENC>            ::= <EXPONENC> ^ <FATOR> | <FATOR>
<FATOR>               ::= \(<EXPRESSAO>\) | CONST_CHAR | CONST_NUM | ID
```

Definição dos Padrões(Expressões Regulares) de cada token

letra -> [A-Za-z]
digito -> [0-9]
digitos -> digito+

ID -> letra[letra digito]*
op_rel -> < | > | <= | >= | == | <>
ws -> (“ “ | \t | \n)
CONST_CHAR -> '.'
CONST_NUM -> digitos(.digitos)?(E[+-]?digitos)?
COMM -> \/*(.|\n)**\/

Tabela com Identificação dos Tokens

Lexemas	**Nome do Token**	**Valor do Atributo**
function	function	-
(	(	-
)	)	-
int	int	-
float	float	-
char	char	-
se	se	-
entao	entao	-
senao	senao	-
enquanto	enquanto	-
faca	faca	-
repita	repita	-
ate	ate	-
=	=	-
<	op_rel	LT
<=	op_rel	LE
==	op_rel	EQ
<>	op_rel	NE
>	op_rel	GT
>=	op_rel	GE
+	+	-
-	-	-

*	*	-
/	/	-
^	^	-
{	{	-
}	}	-
;	;	-
,	,	-
:	:	-
começa em /* e termina em */	-	-
qualquer identificador	ID	posição na tabela de símbolos
qualquer constante numérica	CONST_NUM	posição na tabela de símbolos
qualquer constante char	CONST_CHAR	posição na tabela de símbolos
qualquer ws	-	-

Análise Léxica

Diagramas de Transição para os Tokens

Diagrama do Token “op_rel”

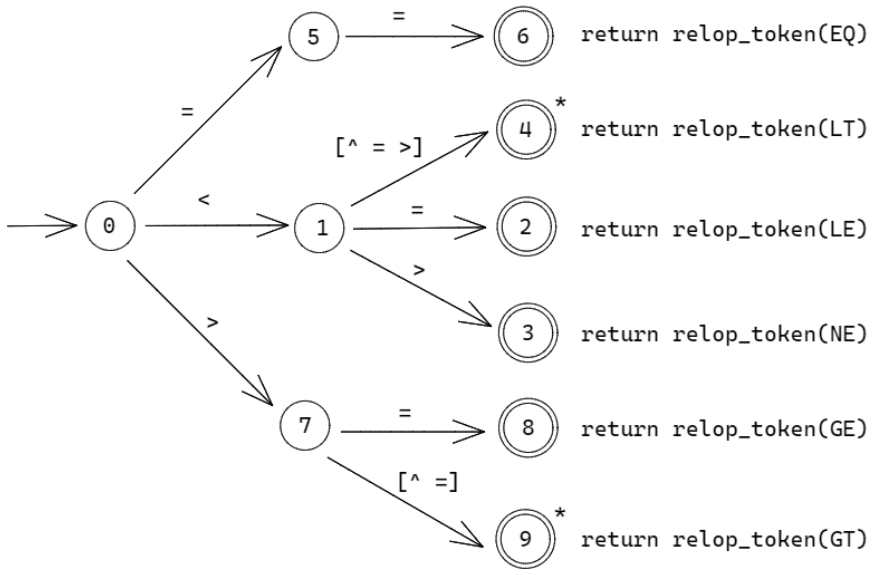


Diagrama de identificadores e palavras-chave

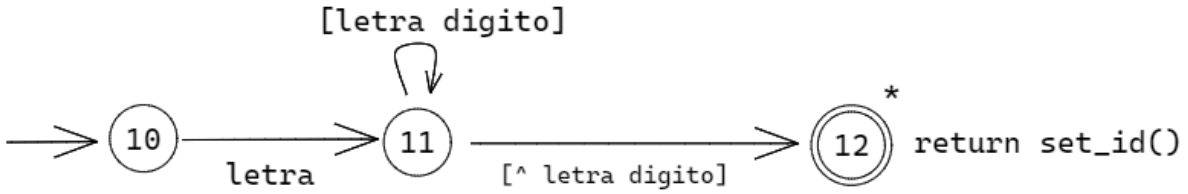


Diagrama para word-separators

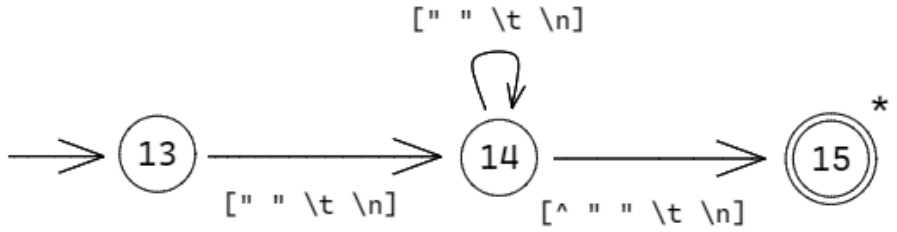
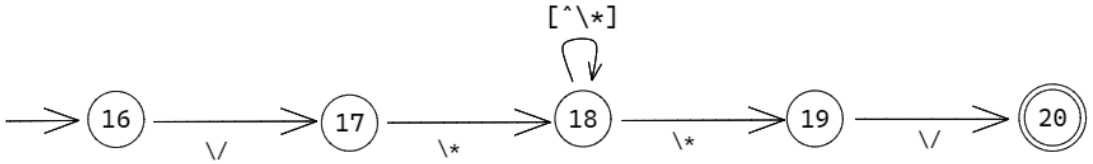
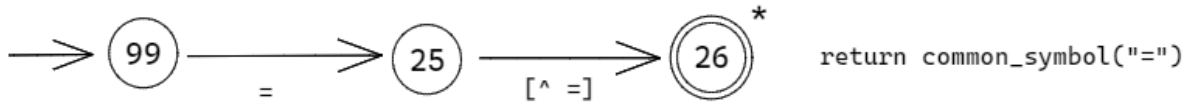


Diagrama para comentários

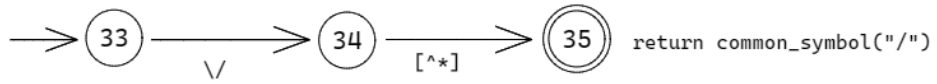


Diagramas para (e)

Diagrama para =



Diagramas para +, -, *, / e ^



Diagramas para {, } e ;

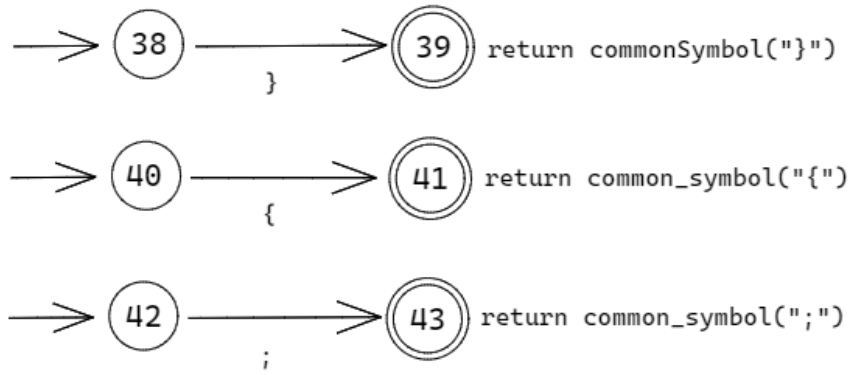


Diagrama para CONST_CHAR

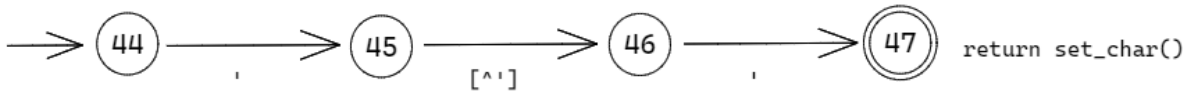


Diagrama para CONST_NUM

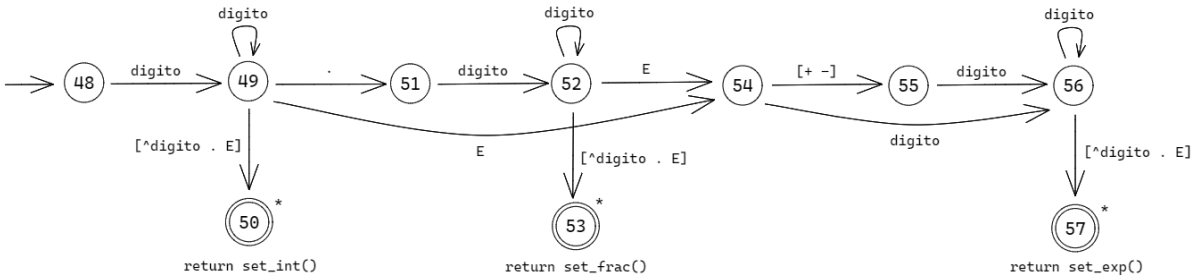
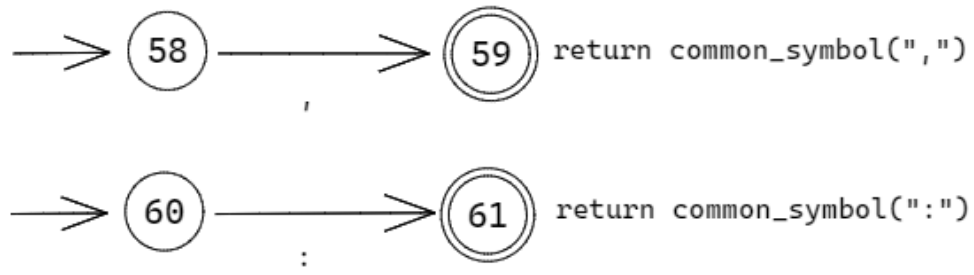


Diagrama para , e :



OBS: Para melhor visualização, segue link para os arquivos em formato SVG: https://drive.google.com/drive/folders/1_VWQgJwMmg99lGuZBN-vMGipKlqxqvvg?usp=sharing

Diagrama unificado não determinístico

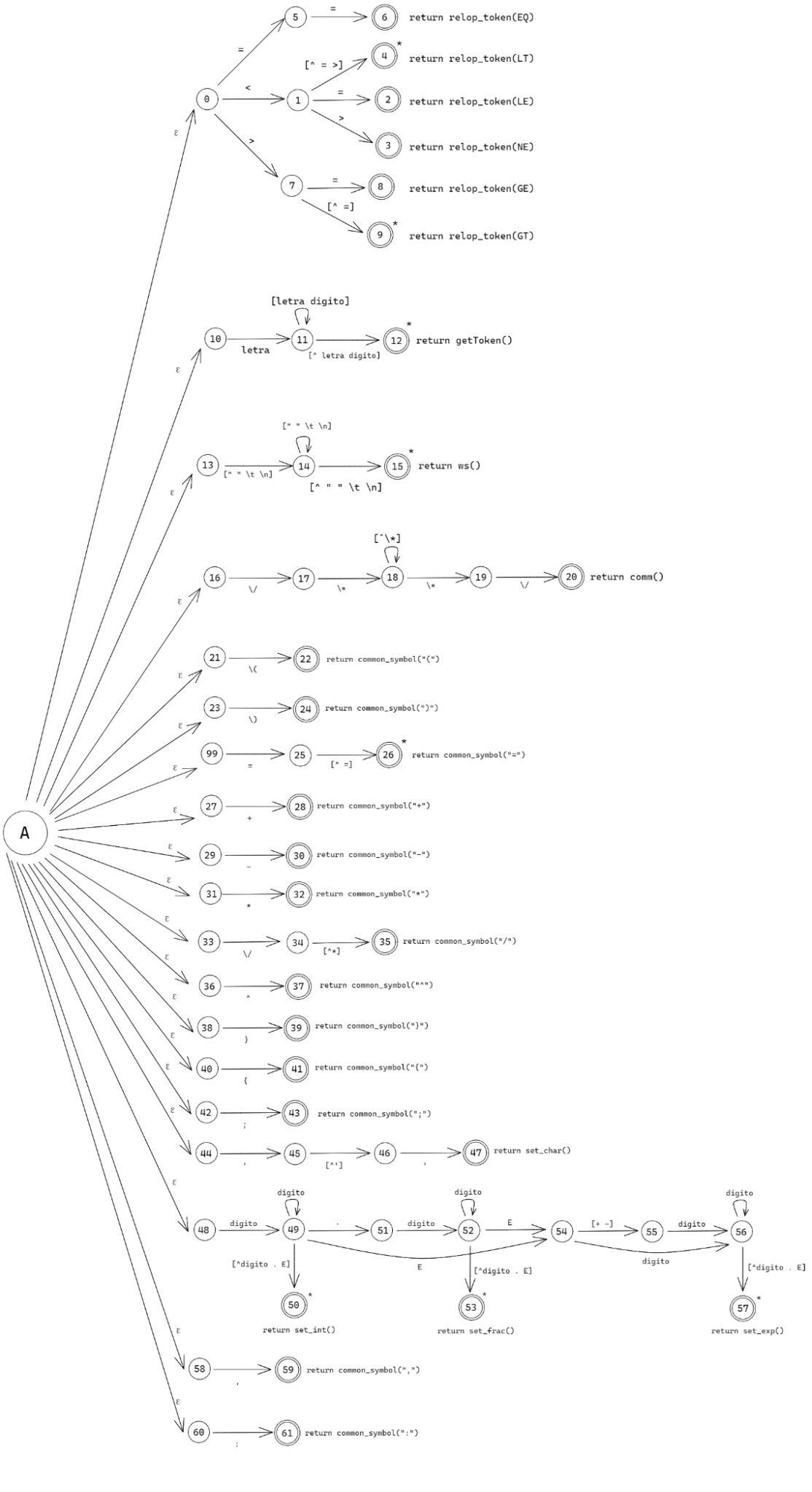
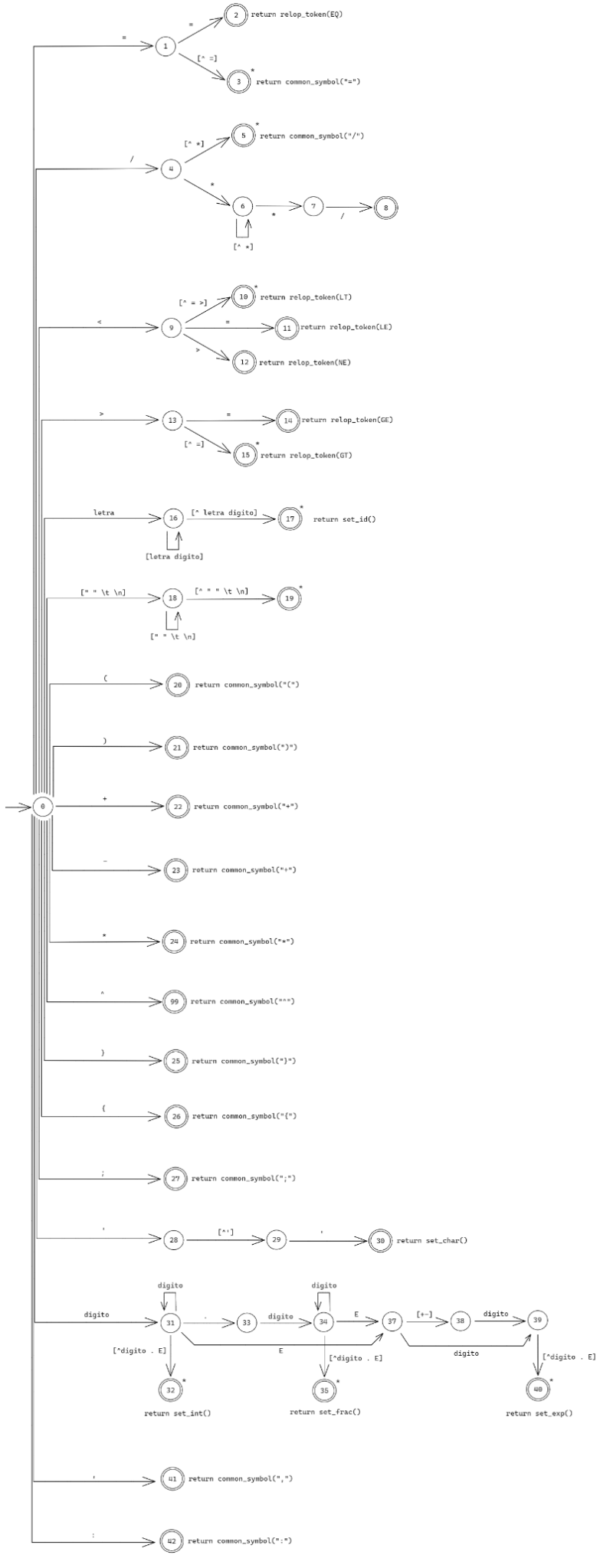

A
ε
0
=
5
=
6
return relop_token(EQ)
=
4
*
return relop_token(LT)
[^ = >]
<
1
=
2
return relop_token(LE)
>
3
return relop_token(NE)
>
7
=
8
return relop_token(GE)
[^ =]
9
*
return relop_token(GT)
10
letra
11
[letra digito]
12
*
return getToken()
[^ letra digito]
13
[" " \t \n]
14
[" " \t \n]
15
*
return ws()
[^ " " \t \n]
16
\/
17
*
18
[^*]
*
19
\/
20
return comm()
21
\(
22
return common_symbol("(")
23
\)
24
return common_symbol(")")
99
=
25
[^ =]
26
*
return common_symbol("=")
27
+
28
return common_symbol("+")
29
-
30
return common_symbol("-")
31
*
32
return common_symbol("*")
33
\/
34
[^*]
35
return common_symbol("/")
36
^
37
return common_symbol("^")
38
}
39
return common_symbol("}")
40
{
41
return common_symbol("{")
42
;
43
return common_symbol(";")
44
'
45
[^']
46
'
47
return set_char()
48
digito
49
digito
.
51
digito
52
digito
E
54
[+ -]
55
digito
56
digito
[^digito . E]
50
*
return set_int()
E
[^digito . E]
53
*
return set_frac()
digito
[^digito . E]
57
*
return set_exp()
58
,
59
return common_symbol(",")
60
:
61
return common_symbol(":")

Diagrama unificado determinístico


0
1
2
return relop_token(EQ)
=
=
[^ =]
3
return common_symbol("=")
4
/
[^ *]
5
return common_symbol("/")
*
6
*
7
/
8
[^ *]
9
<
[^ = >]
10
return relop_token(LT)
=
11
return relop_token(LE)
>
12
return relop_token(NE)
13
>
=
14
return relop_token(GE)
[^ =]
15
return relop_token(GT)
16
letra
[^ letra digito]
17
return set_id()
[letra digito]
18
[" " \t \n]
[^ " " \t \n]
19
[" " \t \n]
20
(
return common_symbol("(")
21
)
return common_symbol(")")
22
+
return common_symbol("+")
23
-
return common_symbol("+")
24
*
return common_symbol("*")
99
^
return common_symbol("^")
25
}
return common_symbol("}")
26
{
return common_symbol("{")
27
;
return common_symbol(";")
28
'
[^']
29
'
30
return set_char()
31
digito
.
33
digito
34
digito
E
37
[+-]
38
digito
39
[^digito . E]
E
[^digito . E]
digito
[^digito . E]
32
return set_int()
35
return set_frac()
40
return set_exp()
41
,
return common_symbol(",")
42
:
return common_symbol(":")

Análise Sintática

Gramática na forma LL

Removendo recursão à esquerda imediata para <EXPRESSAO>

```
<EXPRESSAO>             ::= <TERMO> <EXPRESSAO'>
<EXPRESSAO'>            ::= \+ <TERMO> <EXPRESSAO'> | - <TERMO> <EXPRESSAO'> | ε
<TERMO>                 ::= <EXPONENC> <TERMO'>
<TERMO'>                ::= \* <EXPONENC> <TERMO'> | / <EXPONENC> <TERMO'> | ε
<EXPONENC>              ::= <FATOR> <EXPONENC'>
<EXPONENC'>             ::= ^ <FATOR> <EXPONENC'> | ε
<FATOR>                 ::= \(<EXPRESSAO>\) | CONST_CHAR | CONST_NUM | ID
```

Removendo ambiguidade para problema do "else vazio"(Fatoração à esquerda):

```
<CMD_SELECAO>           ::= se (<COND>) entao <CMD_OU_BLOCO> <CMD_SELECAO'>
<CMD_SELECAO'>          ::= senao <CMD_OU_BLOCO> | ε
```

Gramática GLC LL(1) Resultante

Símbolos Terminais

- function
- (
-)
- int
- char
- float
- se
- entao
- senao
- enquanto
- faca
- repita
- ate
- =
- op_rel
- +
- -
- *
- /
- ^
- {
- }
- ;
- ,
- :
- ID

- CONST_CHAR
- CONST_NUM

Símbolos Não-Terminais

- INICIO
- BLOCO
- DECLARACOES_VARS
- SEQ_COMANDOS
- DECLARACAO_VAR
- TIPO
- LISTA_IDS
- LISTA_IDS'
- COMANDO
- CMD_ATRIB
- CMD_REPET1
- CMD_REPET2
- CMD_SELECAO
- CMD_SELECAO'
- CMD_OU_BLOCO
- COND
- EXPRESSAO
- EXPRESSAO'
- TERMO
- TERMO'
- EXPONENC
- EXPONENC'
- FATOR

Produções da Gramática(usando notação BNF)

```
<INICIO>              ::= function ID () <BLOCO>
<BLOCO>               ::= {<DECLARACOES_VARS><SEQ_COMANDOS>}
<DECLARACOES_VARS>    ::= <DECLARACAO_VAR><DECLARACOES_VARS> | ε
<DECLARACAO_VAR>      ::= <TIPO>:<LISTA_IDS>;
<TIPO>                ::= int | float | char
<LISTA_IDS>           ::= ID<LISTA_IDS'>
<LISTA_IDS'>          ::= ,ID<LISTA_IDS'> | ε
<SEQ_COMANDOS>        ::= <COMANDO><SEQ_COMANDOS> | ε
<COMANDO>             ::= <CMD_ATRIB> | <CMD_REPET1> | <CMD_REPET2> | <CMD_SELECAO>
<CMD_ATRIB>           ::= ID = <EXPRESSAO>;
<CMD_REPET1>          ::= enquanto (<COND>) faca <CMD_OU_BLOCO>
<CMD_REPET2>          ::= repita <CMD_OU_BLOCO> ate (<COND>)
<CMD_SELECAO>         ::= se (<COND>) entao <CMD_OU_BLOCO> <CMD_SELECAO'>
<CMD_SELECAO'>        ::= senao <CMD_OU_BLOCO> | ε
<CMD_OU_BLOCO>        ::= <COMANDO> | <BLOCO>
<COND>                ::= <EXPRESSAO> op_rel <EXPRESSAO>
<EXPRESSAO>           ::= <TERMO> <EXPRESSAO'>
<EXPRESSAO'>          ::= \+ <TERMO> <EXPRESSAO'> | - <TERMO> <EXPRESSAO'> | ε
<TERMO>               ::= <EXPONENC> <TERMO'>
<TERMO'>              ::= \* <EXPONENC> <TERMO'> | / <EXPONENC> <TERMO'> | ε
<EXPONENC>            ::= <FATOR> <EXPONENC'>
```

```
<EXPONENC'>             ::= ^ <FATOR> <EXPONENC'> | ε
<FATOR>                 ::= \(<EXPRESSAO>\) | CONST_CHAR | CONST_NUM | ID
```

FIRST e FOLLOW dos símbolos da gramática

Tabela com o FIRST de todos os símbolos da gramática

Símbolo	**Conjunto do FIRST**
function	function
(	(
(	)
int	int
float	float
char	char
se	se
entao	entao
senao	senao
enquanto	enquanto
faca	faca
repita	repita
ate	ate
=	=
op_rel	op_rel
+	+
-	-
*	*
/	/
^	^
{	{
}	}
;	;

,	,
:	:
ID	ID
CONST_CHAR	CONST_CHAR
CONST_NUM	CONST_NUM
INICIO	function
BLOCO	{
DECLARACOES_VARS	Ɛ, int, float, char
SEQ_COMANDOS	Ɛ, ID, enquanto, repita, se
DECLARACAO_VAR	int, float, char
TIPO	int, float, char
LISTA_IDS	ID
LISTA_IDS'	**Ɛ**, **,**
COMANDO	ID, enquanto, repita, se
CMD_ATRIB	ID
CMD_REPET1	enquanto
CMD_REPET2	repita
CMD_SELECAO	se
CMD_SELECAO'	senao
CMD_OU_BLOCO	ID, enquanto, repita, se, {
COND	(, CONST_CHAR, CONST_NUM, ID
EXPRESSAO	(, CONST_CHAR, CONST_NUM, ID
EXPRESSAO'	+, -, Ɛ
TERMO	(, CONST_CHAR, CONST_NUM, ID
TERMO'	*, /, Ɛ
EXPONENC	(, CONST_CHAR, CONST_NUM, ID
EXPONENC'	^, Ɛ
FATOR	(, CONST_CHAR, CONST_NUM, ID

Tabela com o FOLLOW de todos os símbolos não-terminais

Símbolo Não-Terminal	Conjunto do FOLLOW
INICIO	$
BLOCO	FOLLOW(INICIO) + FOLLOW(CMD_OU_BLOCO) = $, ID, enquanto, repita, se, senao, ate
DECLARACOES_VARS	FIRST(<SEQ_COMANDOS>) - {Ɛ} + FIRST(}) = ID, enquanto, repita, se, }
SEQ_COMANDOS	}
DECLARACAO_VAR	FIRST(<DECLARACOES_VARS>) - {Ɛ} + FOLLOW(<DECLARACOES_VARS>) = int, char, float, ID, enquanto, repita, se, }
TIPO	:
LISTA_IDS	;
LISTA_IDS'	FOLLOW(<LISTA_IDS'>) = ;
COMANDO	FIRST(<SEQ_COMANDOS>) - {Ɛ} + FIRST(}) + FOLLOW(<CMD_OU_BLOCO>) = ID, enquanto, repita, se, }, ate, senao
CMD_ATRIB	FOLLOW(<COMANDO>) = ID, enquanto, repita, se, }, ate, senao
CMD_REPET1	FOLLOW(<COMANDO>) = ID, enquanto, repita, se, }, ate, senao
CMD_REPET2	FOLLOW(<COMANDO>) = ID, enquanto, repita, se, }, ate, senao
CMD_SELECAO	FOLLOW(<COMANDO>) = ID, enquanto, repita, se, }, ate, senao
CMD_SELECAO'	FOLLOW(<CMD_SELECAO>) = ID, enquanto, repita, se, }, ate, senao
CMD_OU_BLOCO	FOLLOW(<CMD_REPET1>) + FIRST(ate) + FIRST(<CMD_SELECAO'>) - {Ɛ} + FOLLOW(<CMD_SELECAO'>) = ID, enquanto, repita, se, }, ate, senao
COND	)
EXPRESSAO	op_rel,), ;
EXPRESSAO'	FOLLOW(<EXPRESSAO>) = op_rel,), ;
TERMO	FIRST(<EXPRESSAO'>) - {Ɛ} + FOLLOW(<EXPRESSAO'>)

Símbolo Não-Terminal	Conjunto do FOLLOW
	+ FOLLOW(<EXPRESSAO>)= +, -, op_rel,), ;
TERMO'	FOLLOW(<TERMO>) = +, -, op_rel,), ;
EXPONENC	FIRST(<TERMO'>) - {Ɛ} + FOLLOW(<TERMO'>) + FOLLOW(<TERMO>) = *, /, +, -, op_rel,), ;
EXPONENC'	FOLLOW(<EXPONENC>) = *, /, +, -, op_rel,), ;
FATOR	FIRST(<EXPONENC'>) - {Ɛ} + FOLLOW(<EXPONENC'>) + FOLLOW(<EXPONENC>)= ^, *, /, +, -, op_rel,), ;

Tabela de Análise Preditiva

TABELA DE PRODUÇÕES	
1	function ID () <BLOCO>
2	{<DECLARACOES_VARS><SEQ_COMANDOS>}
3	<DECLARACAO_VAR><DECLARACOES_VARS>
4	Ɛ
5	<TIPO>:<LISTA_IDS>;
6	int
7	float
8	char
9	ID<LISTA_IDS'>
10	,ID<LISTA_IDS'>
11	Ɛ
12	<COMANDO><SEQ_COMANDOS>
13	Ɛ
14	<CMD_ATRIB>
15	<CMD_REPET1>
16	<CMD_REPET2>
17	<CMD_SELECAO>
18	ID = <EXPRESSAO>;

19	enquanto (<COND>) faca <CMD_OU_BLOCO>
20	repita <CMD_OU_BLOCO> ate (<COND>)
21	se (<COND>) entao <CMD_OU_BLOCO> <CMD_SELECAO'>
22	senao <CMD_OU_BLOCO>
23	ε
24	<COMANDO>
25	<BLOCO>
26	<EXPRESSAO> op_rel <EXPRESSAO>
27	<TERMO> <EXPRESSAO'>
28	\+ <TERMO> <EXPRESSAO'>
29	- <TERMO> <EXPRESSAO'>
30	ε
31	<EXPONENC> <TERMO'>
32	* <EXPONENC> <TERMO'>
33	/ <EXPONENC> <TERMO'>
34	ε
35	<FATOR> <EXPONENC'>
36	^ <FATOR> <EXPONENC'>
37	ε
38	\(<EXPRESSAO>\)
39	CONST_CHAR
40	CONST_NUM
41	ID

Tabela de Análise Sintática Preditiva

Da tabela contendo a enumeração das produções, iremos escrever de modo declarativo qual produção será chamada quando estivermos com um símbolo não terminal no topo da pilha e lermos um símbolo terminal. Para os casos de não haver registro para a leitura de um símbolo terminal, assume-se que não há produção.

NÃO-TERMINAL	TERMINAL	PRODUÇÃO
INICIO	function	P1
BLOCO	{	P2
DECLARACOES_VARS	int	P3
	float	P3
	char	P3
	ID	P4
	enquanto	P4
	repita	P4
	se	P4
	}	P4
DECLARACAO_VAR	int	P5
	float	P5
	char	P5
TIPO	int	P6
	float	P7
	char	P8
LISTA_IDS	ID	P9
LISTA_IDS’	,	P10
	;	P11
SEQ_COMANDOS	ID	P12
SEQ_COMANDOS	enquanto	P12
	repita	P12
	se	P12
	}	P13
COMANDO	ID	P14
	enquanto	P15
	repita	P16

NÃO-TERMINAL	TERMINAL	PRODUÇÃO
	se	P17
CMD_ATRIB	ID	P18
CMD_REPET1	enquanto	P19
CMD_REPET2	repita	P20
CMD_SELECAO	se	P21
CMD_SELECAO’	senao	P22
	ID	P23
	enquanto	P23
	repita	P23
	se	P23
	}	P23
	ate	P23
CMD_OU_BLOCO	ID	P24
	enquanto	P24
	repita	P24
	se	P24
	{	P25
COND	(	P26
	CONST_CHAR	P26
	CONST_NUM	P26
	ID	P26
EXPRESSAO	(	P27
	CONST_CHAR	P27
	CONST_NUM	P27
	ID	P27
EXPRESSAO’	+	P28
	-	P29
	op_rel	P30

NÃO-TERMINAL	TERMINAL	PRODUÇÃO
	)	P30
	;	P30
TERMO	(	P31
	CONST_CHAR	P31
	CONST_NUM	P31
	ID	P31
TERMO’	*	P32
	/	P33
	+	P34
	-	P34
	op_rel	P34
	)	P34
	;	P34
EXPONENC	(	P35
	CONST_CHAR	P35
	CONST_NUM	P35
	ID	P35
EXPONENC’	^	P36
	*	P37
	/	P37
EXPONENC’	+	P37
	-	P37
	op_rel	P37
	)	P37
	;	P37
FATOR	(	P38
	CONST_CHAR	P39
	CONST_NUM	P40

NÃO-TERMINAL	TERMINAL	PRODUÇÃO
	ID	P41

Tradução Dirigida por Sintaxe

Definição dos Atributos

Os seguintes símbolos possuem o atributo ***t***, para guardar informações como tipo de dado e posição na tabela de símbolos.

Esquemas de Tradução

Símbolo	Corpo da produção	Regra Semântica
<DECLARACAO_VAR>	::= <TIPO>:<LISTA_IDS>;	update_sym_table_entries(<LISTA_IDS>.t, <TIPO>.t)
<TIPO>	::= int	<TIPO>.t = int
	::= float	<TIPO>.t = float
	::= char	<TIPO>.t = char
<LISTA_IDS>	::= ID<LISTA_IDS'>	<LISTA_IDS>.t = [get_sym_table_entry(ID)] ++ <LISTA_IDS'>.t
<LISTA_IDS'>	::= ,ID<LISTA_IDS'>	<LISTA_IDS'>[0].t = [get_sym_table_entry(ID)] ++ <LISTA_IDS'>[1].t
	::= ε	<LISTA_IDS'>.t = []
<CMD_ATRIB>	::= ID = <EXPRESSAO>;	validate_attr_type(get_sym_table_entry(ID).data_type, <EXPRESSAO>.t)
<COND>	::= <EXPRESSAO> op_rel <EXPRESSAO>	validate_expr_type(<EXPRESSAO>[0].t, <EXPRESSAO>[1].t)
<EXPRESSAO>	::= <TERMO> <EXPRESSAO'>	<EXPRESSAO>.t = validate_expr_type(<TERMO>.t, <EXPRESSAO'>.t)
<EXPRESSAO'>	::= \+ <TERMO> <EXPRESSAO'>	<EXPRESSAO'>[0].t = validate_expr_type(<TERMO>.t, <EXPRESSAO'>[1].t)
	::= - <TERMO> <EXPRESSAO'>	<EXPRESSAO'>[0].t = validate_expr_type(<TERMO>.t, <EXPRESSAO'>[1].t)
	::= ε	<EXPRESSAO'>.t = None
<TERMO>	::= <EXPONENC> <TERMO'>	<TERMO>.t = validate_expr_type(<EXPONENC>.t, <TERMO'>.t)
<TERMO'>	::= * <EXPONENC> <TERMO'>	<TERMO'>[0].t = validate_expr_type(<EXPONENC>.t, <TERMO'>[1].t)
	::= / <EXPONENC> <TERMO'>	<TERMO'>[0].t = validate_expr_type(<EXPONENC>.t, <TERMO'>[1].t)
	::= ε	<TERMO'>[0].t = None

<EXPONENC>	::= <FATOR> <EXPONENC'>	<EXPONENC>.t = validate_expr_type(<FATOR>.t, <EXPONENC'>.t)
<EXPONENC'>	::= ^ <FATOR> <EXPONENC'>	<EXPONENC'>[0].t = validate_expr_type(<FATOR>.t, <EXPONENC'>[1].t)
	::= ε	<EXPONENC'>.t = None
<FATOR>	::= \(<EXPRESSAO>\)	<FATOR>.t = <EXPRESSAO>.t
	::= CONST_CHAR	<FATOR>.t = get_sym_table_entry(CONST_CHAR).data_type
	::= CONST_NUM	<FATOR>.t = get_sym_table_entry(CONST_NUM).data_type
	::= ID	<FATOR>.t = get_sym_table_entry(ID).data_type