UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Compiladores

Espinosa González Isaac
Montoya Pérez Hector
Soto Vázquez Patricia
Pérez Dublán Juan Pablo

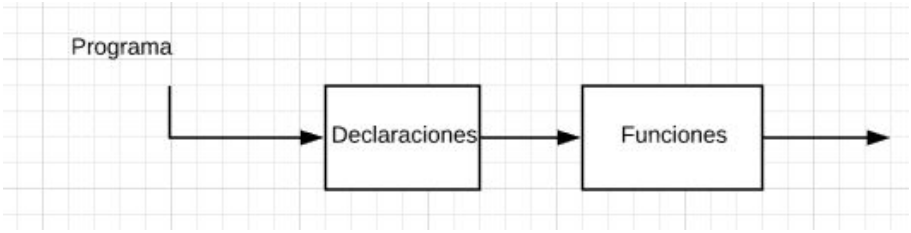
Programa 2(analizador sintáctico)

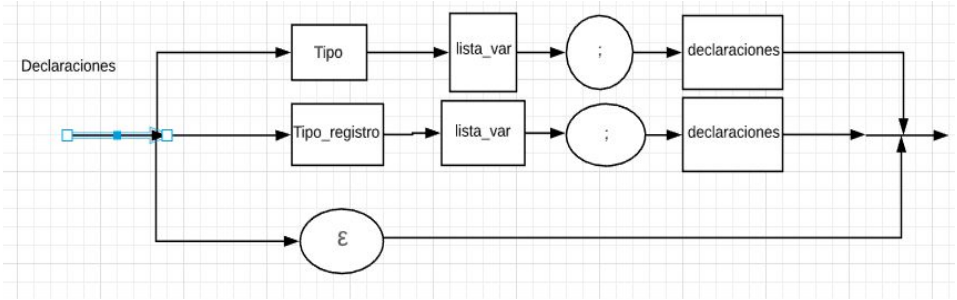
Universidad Nacional Autonoma de México
Semestre 2020-2
Compiladores
Análisis Sintáctico
Profesor: Adrián Ulises Mercado Martínez
Programa 2

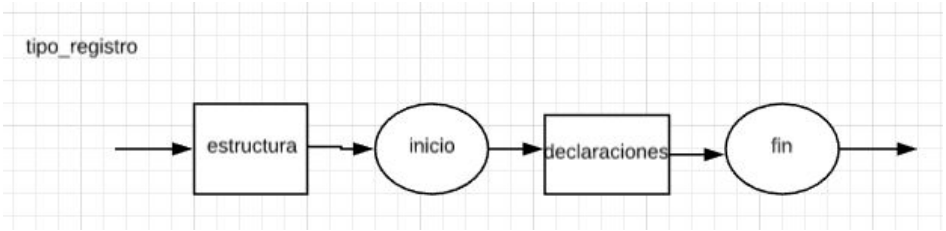
1. Para la gramatica siguiente elaborar el analizador sintactico usando utilizando bison o yacc.
2. Entregar un documento con lo siguiente
   (a) Analisis del problema
      i. Descripción del problema no del programa

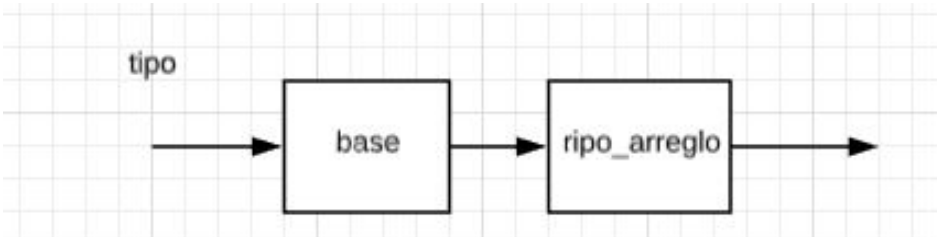
Realizar un analizador sintáctico el cual identificara que la gramática de un pseudocódigo se cumpla, es decir, revisará que se tenga una estructura gramatical correcta.

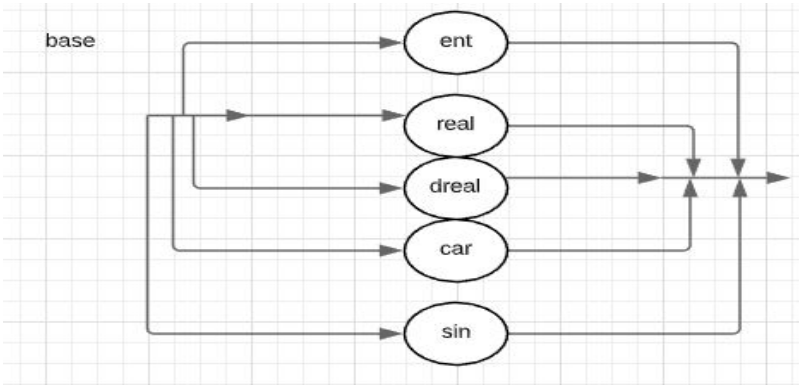
(b) Diseño de la solución
   i. Incluir diagramas de sintaxis

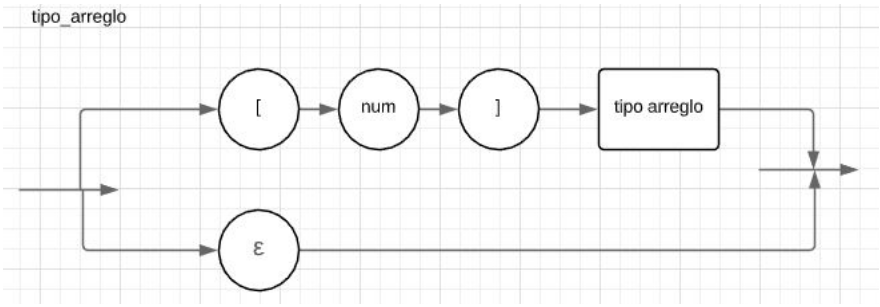


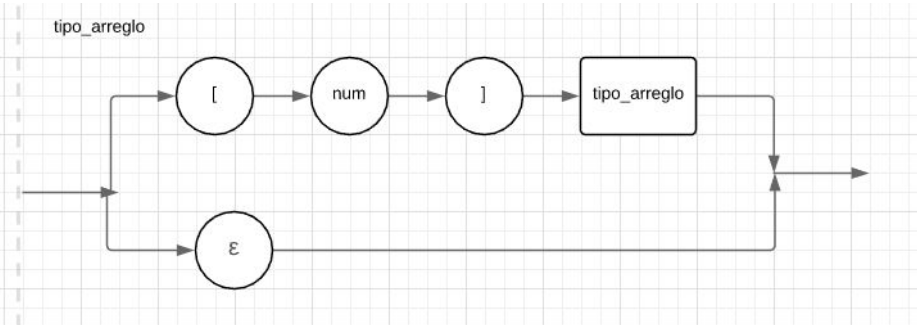


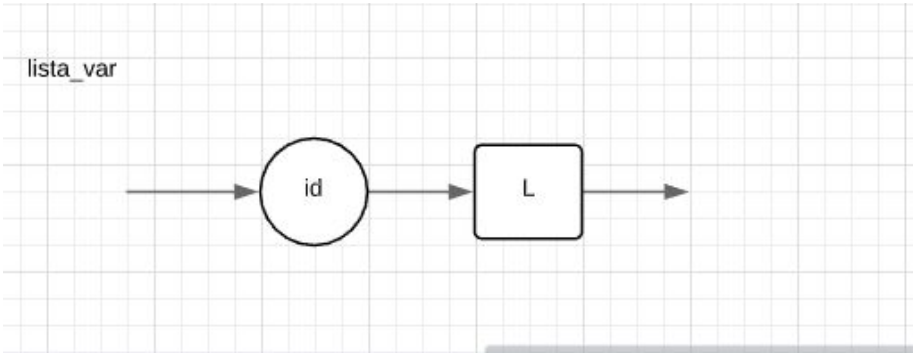


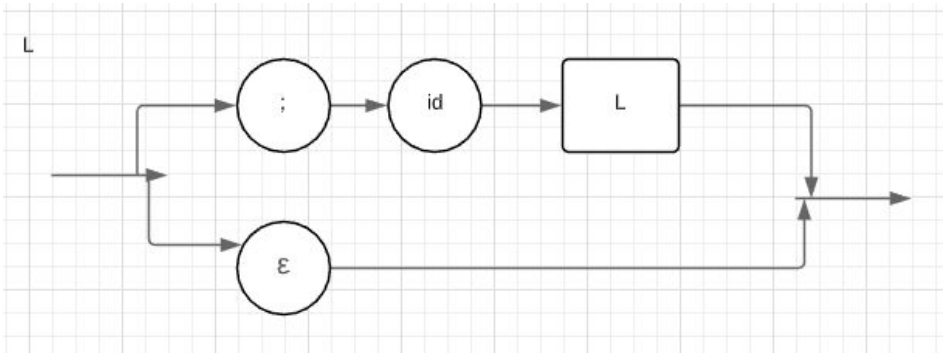


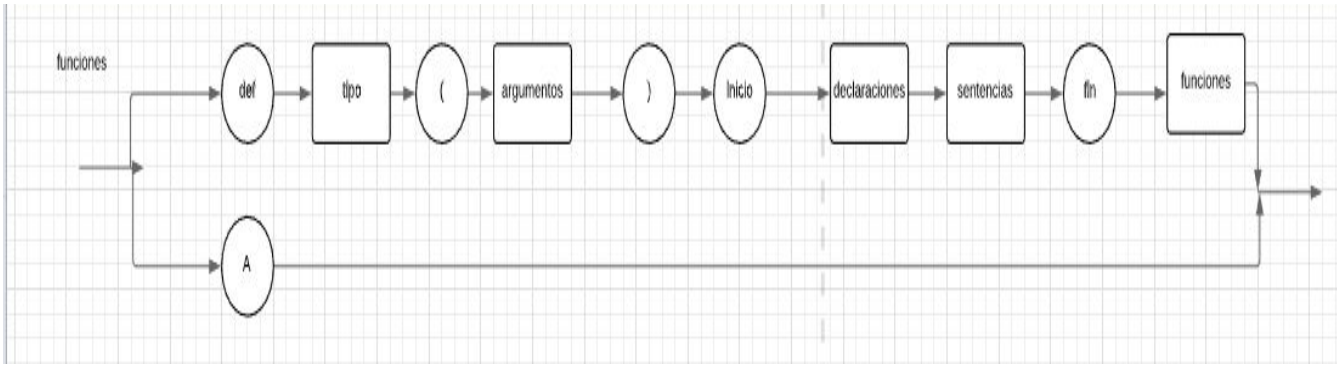

base
ent
real
dreal
car
sin

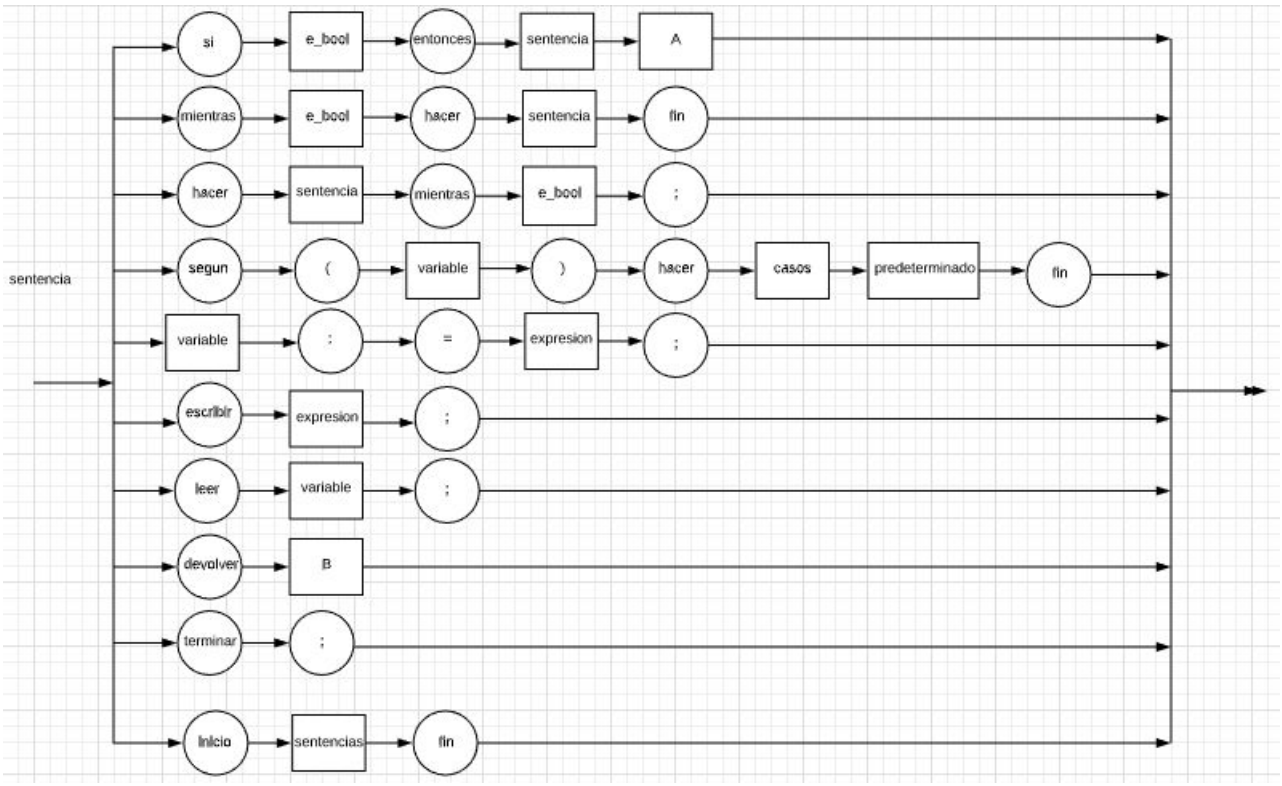

tipo_arreglo
[
num
]
tipo arreglo
ε

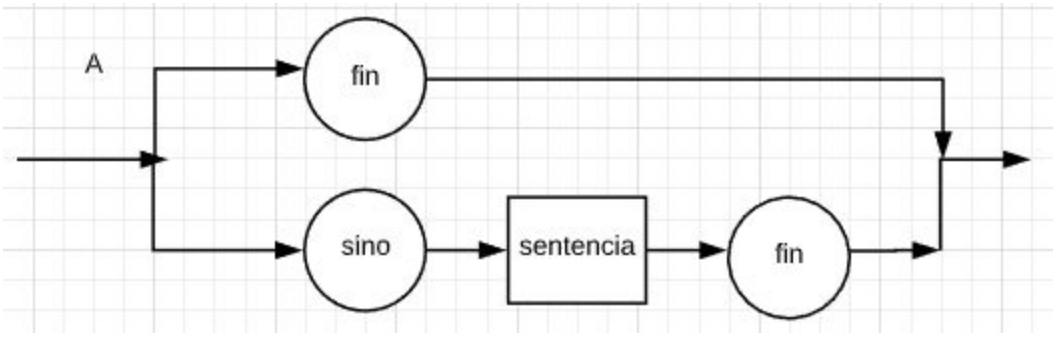

tipo_arreglo
[
num
]
tipo_arreglo
ε

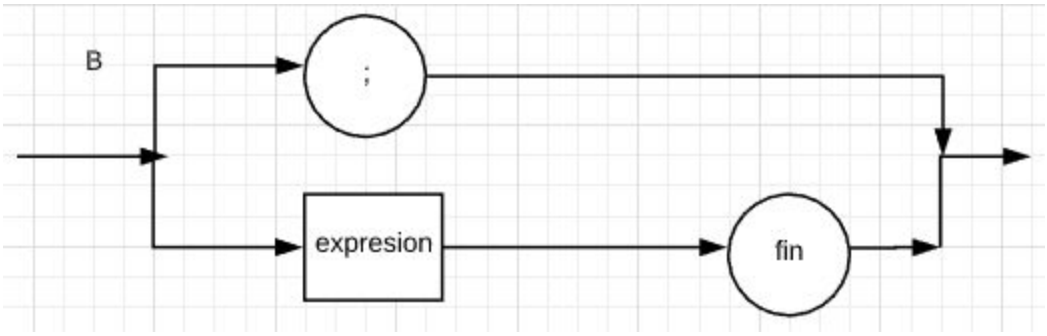

lista_var
id
L

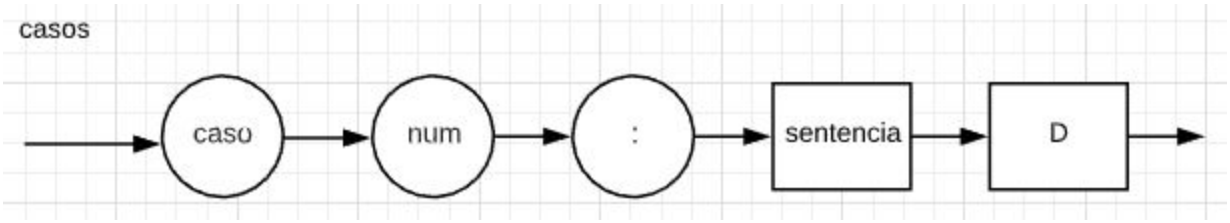

L
;
id
L
ε

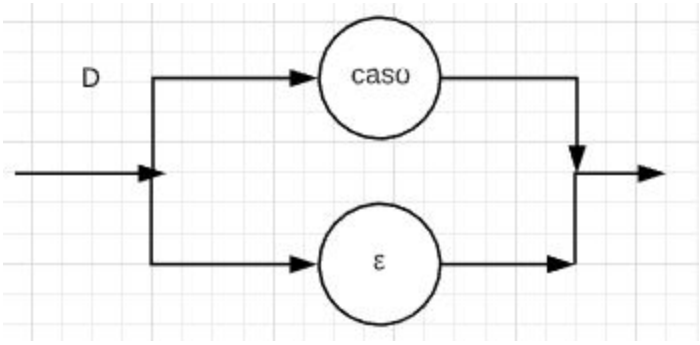

funciones
def
tipo
(
argumentos
)
Inicio
declaraciones
sentencias
fin
funciones
A

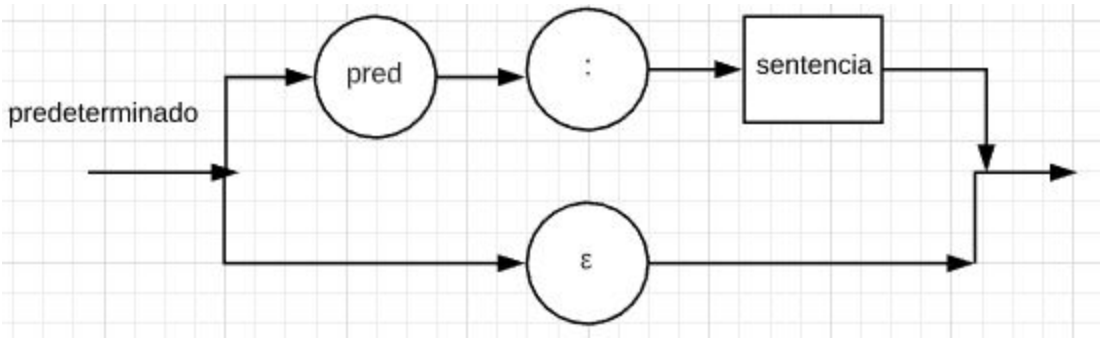

sentencia
si
e_bool
entonces
sentencia
A
mientras
e_bool
hacer
sentencia
fin
hacer
sentencia
mientras
e_bool
;
segun
(
variable
)
hacer
casos
predeterminado
fin
variable
:
=
expresion
;
escribir
expresion
;
leer
variable
;
devolver
B
terminar
;
Inicio
sentencias
fin

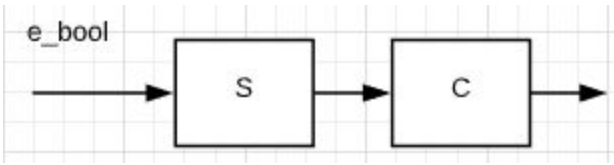

A
fin
sino
sentencia
fin

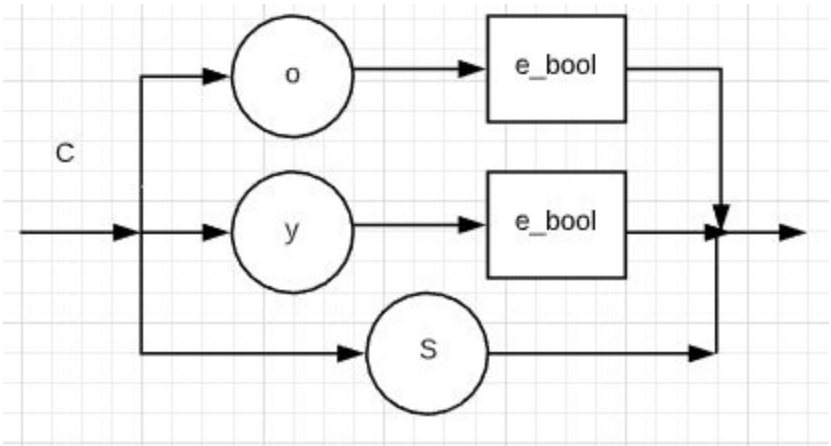

B
;
expresion
fin

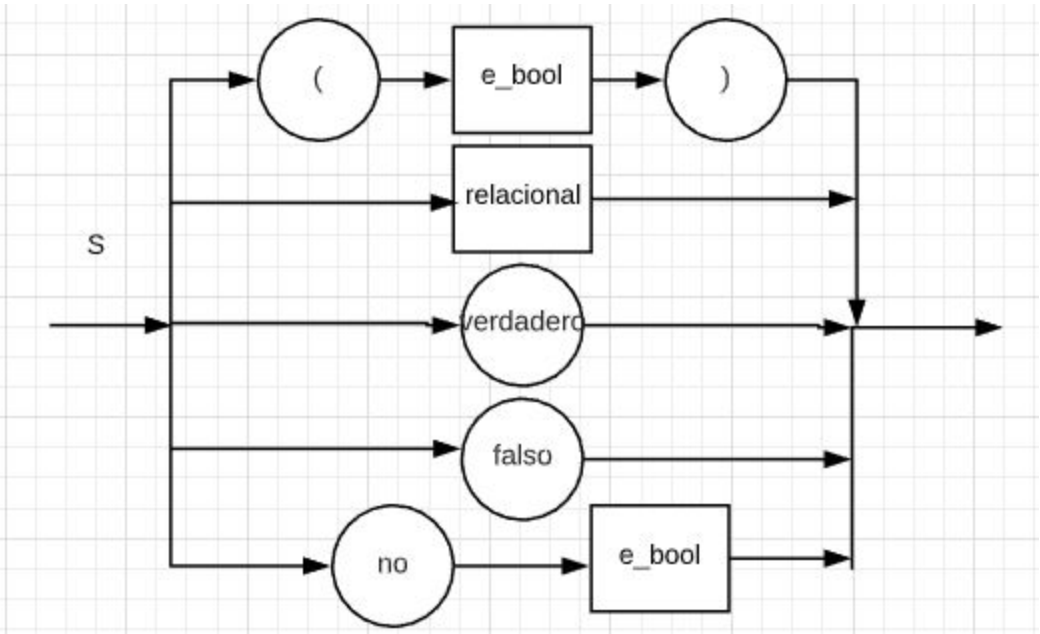

casos
caso
num
:
sentencia
D

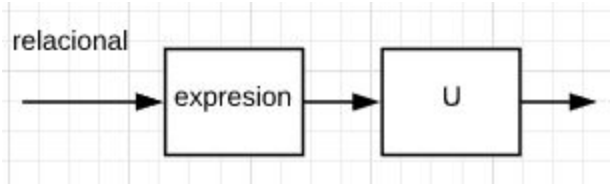

D
caso
ε

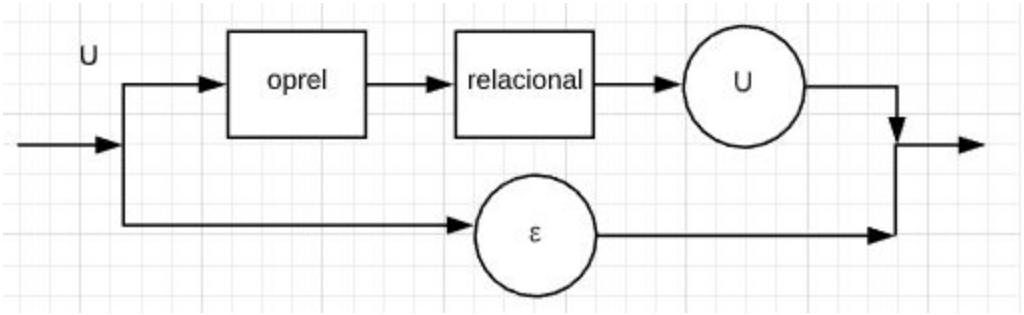

predeterminado
pred
:
sentencia
ε

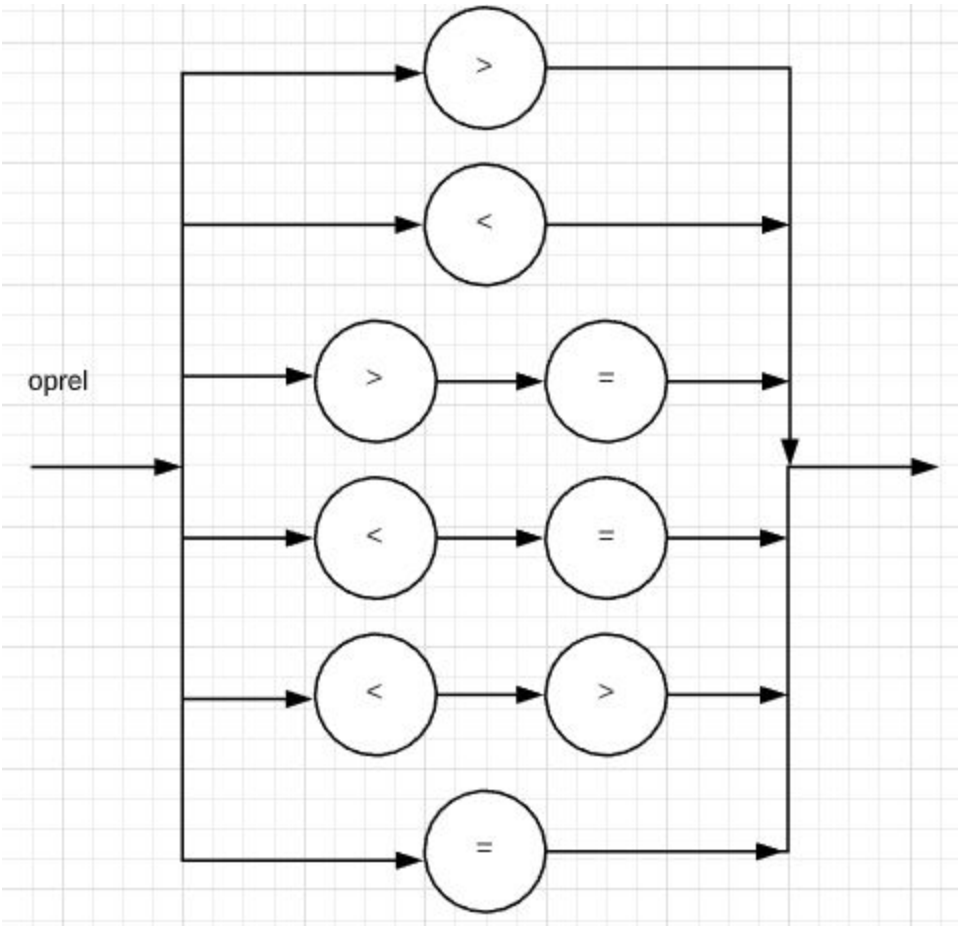

e_bool
S
C

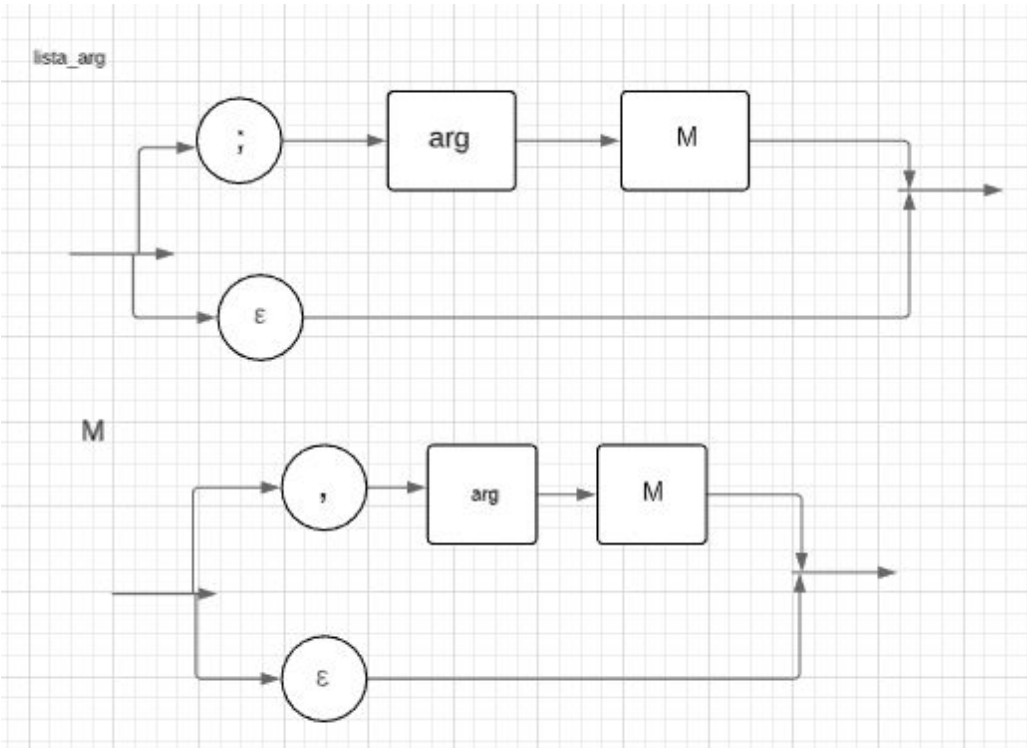

C
o
e_bool
y
e_bool
S

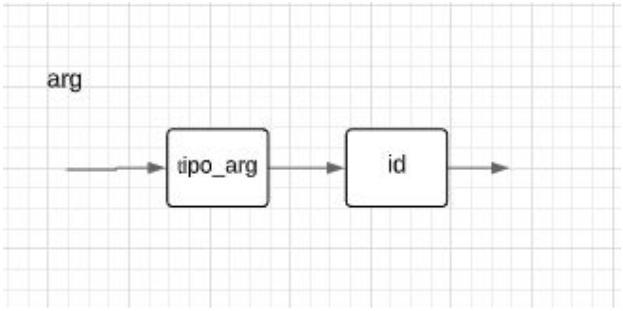

S
(
e_bool
)
relacional
verdadero
falso
no
e_bool

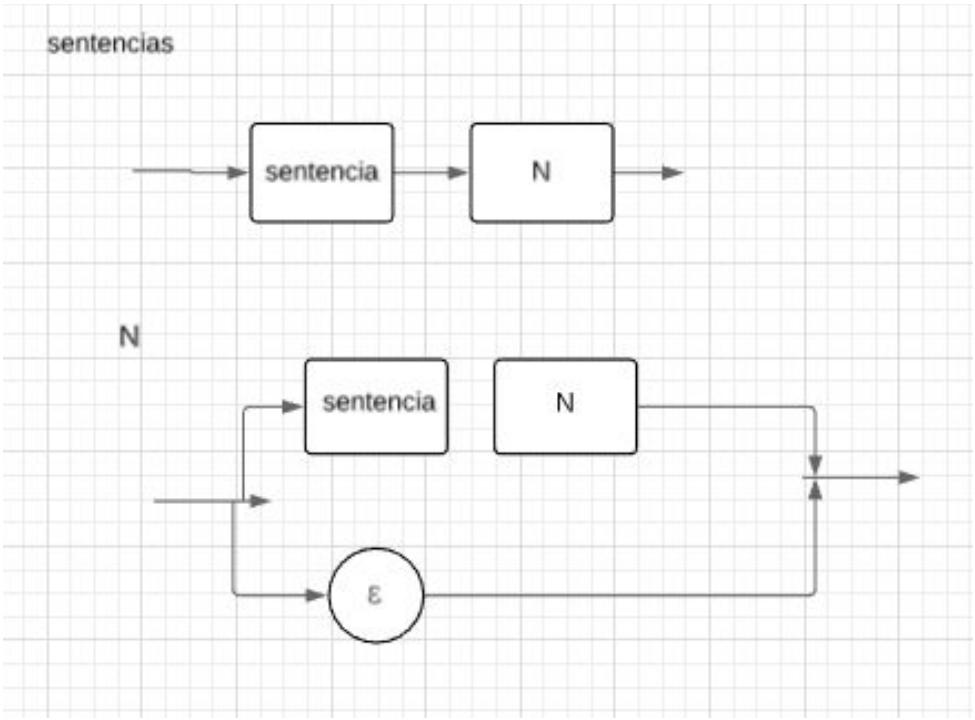

relacional
expresion
U

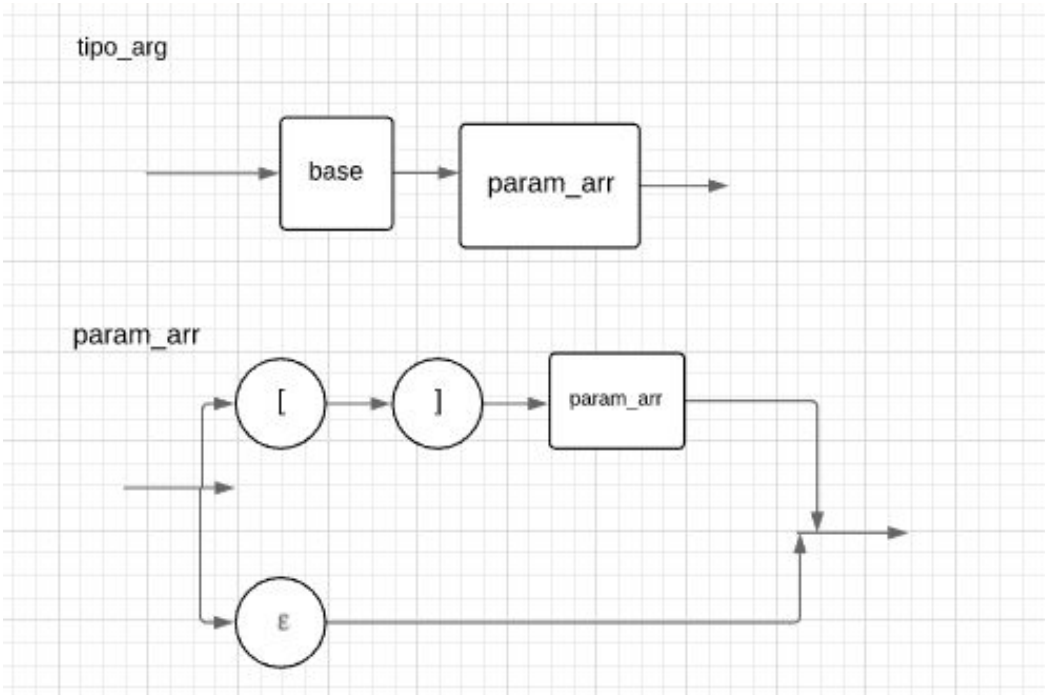

U
oprel
relacional
U
ε

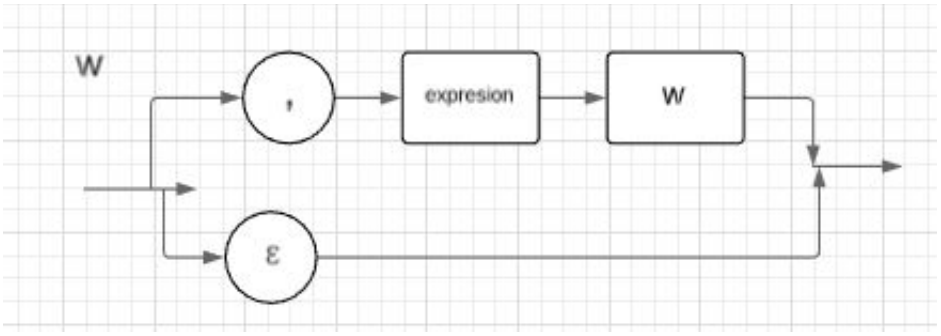

oprel
>
<
>
=
<
=
<
>
=

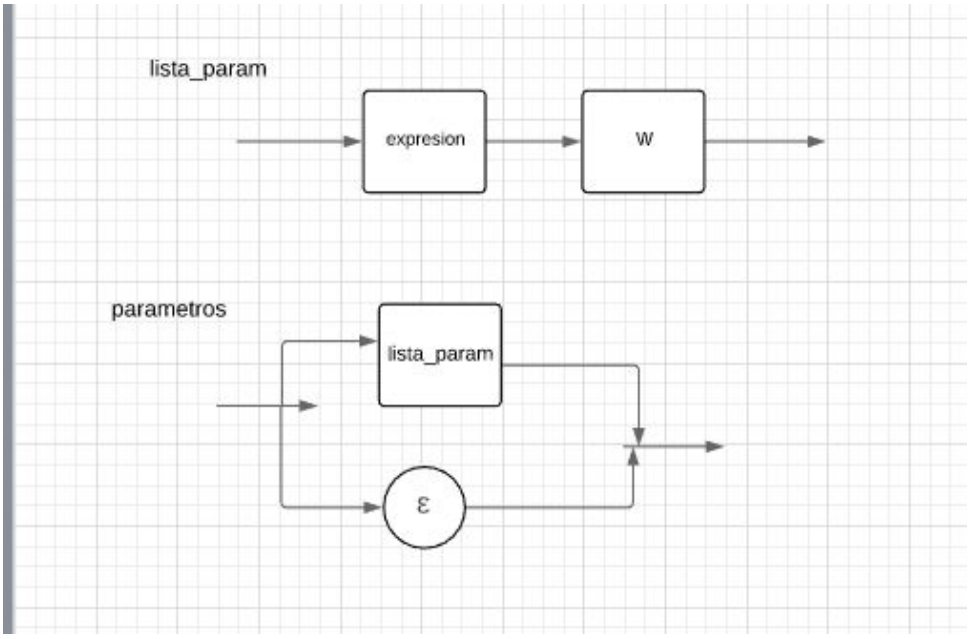

lista_arg
;
arg
M
ε
M
,
arg
M
ε

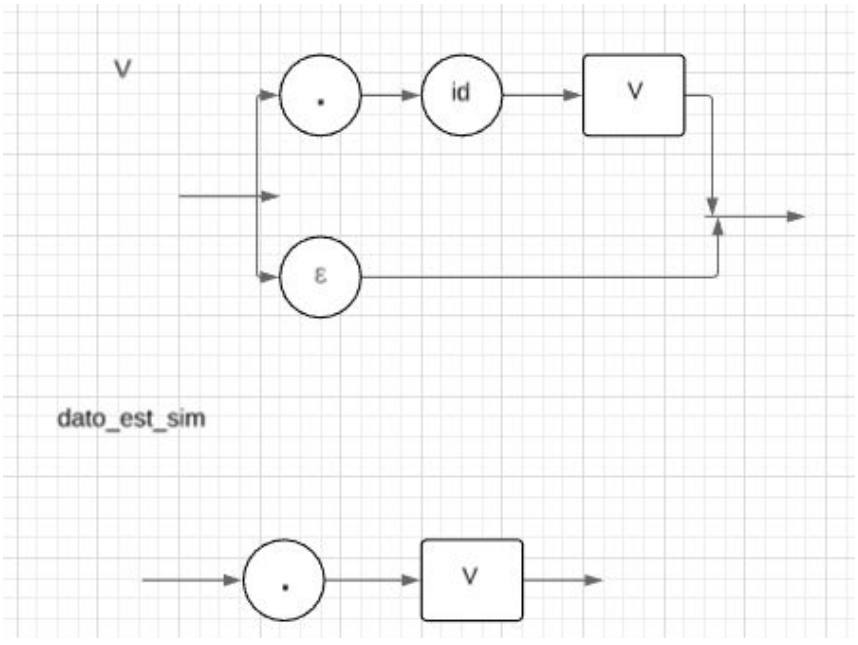

arg
tipo_arg
id

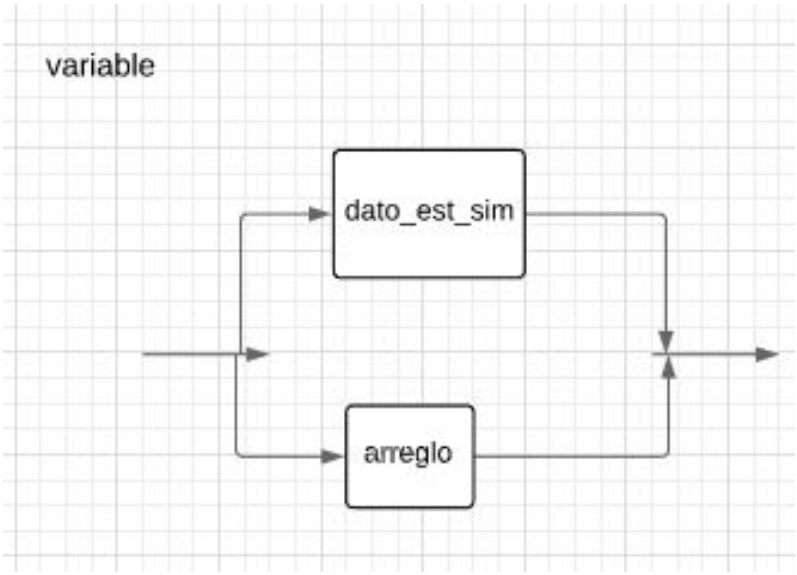

sentencias
sentencia
N
N
sentencia
N
ε

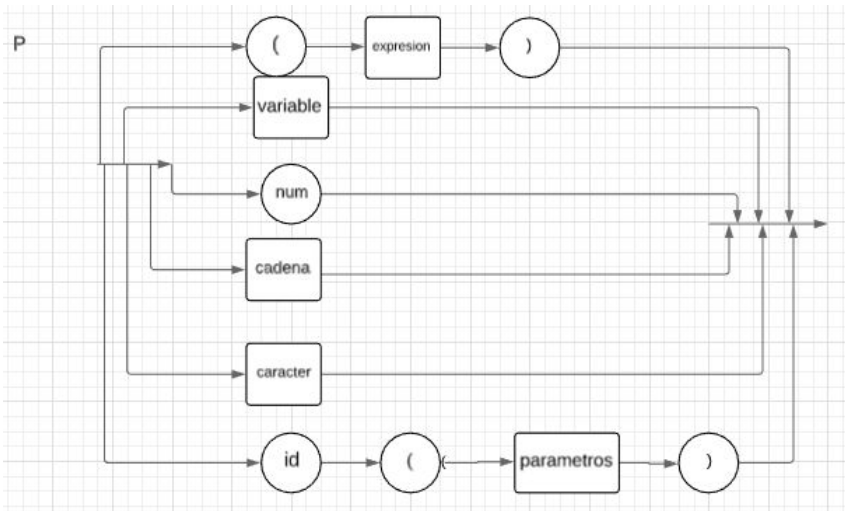

tipo_arg
base
param_arr
param_arr
[
]
param_arr
ε

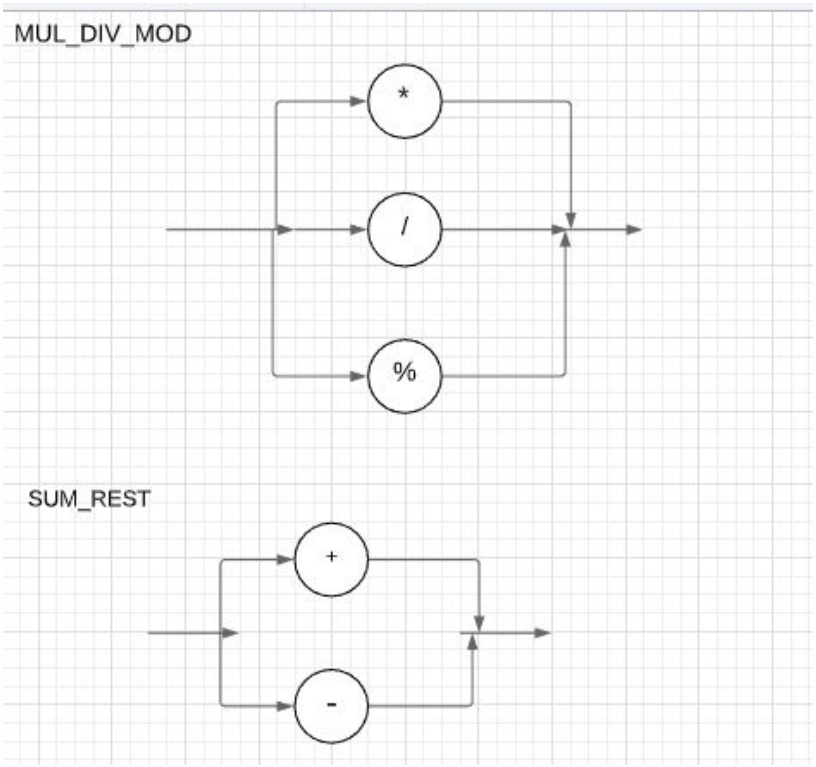

W
,
expresion
W
ε

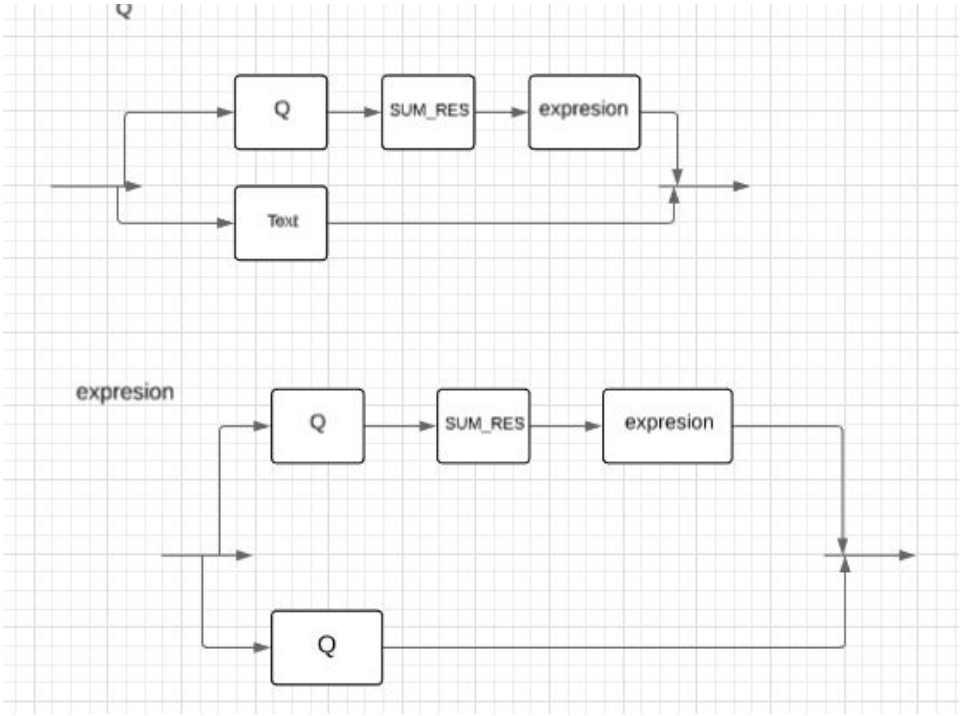

lista_param
expresion
W
parametros
lista_param
ε


V
.
id
V
ε
dato_est_sim
.
V


variable
dato_est_sim
arreglo


P
(
expresion
)
variable
num
cadena
caracter
id
(
parametros
)


MUL_DIV_MOD
*
/
%
SUM_REST
+
-



ii. En caso de quitar ambigüedad incluir el proceso

e_bool → e_bool o e_bool | e_bool y e_bool | no e_bool | (e_bool)
| relacional | verdadero | falso

e_bool → S o e_bool | S y e_bool | S

S →(e_bool) | relacional | verdadero | falso | no e_bool

expresion → expresion oparit expresion
| expresion % expresion | (expresion) | id
| variable | num | cadena | caracter | id(parametros)

oparit → + | − | ∗| /

expresion →Q SUM_RES expresion | Q

Q → P MUL_DIV Q| P

P → (expresion) | variable | num | cadena | caracter | id(parametros)

SUM_RES → **+** | −

MUL_DIV_MOD → ∗| / | %

iii. En caso de eliminar recursividad incluir el proceso

Regla para la eliminación de la recursividad izquierda:

A→ A α | β

A→ β A'

A'→ α A' | ε

lista_var → lista_var, id | id

lista_var → id L

L → ,id L | ε

lista_arg → lista_arg, arg | arg

lista_arg → arg M

M → ,arg M | ε

sentencias → sentencias sentencia | sentencia

sentencias → sentencia N

N → sentencia N | ε

relacional → relacional oprel relacional | expresion

relacional → expresion U

U → oprel relacional U | ε

dato_est_sim → dato_est_sim .id | id

dato_est_sim → id V

V → .id V | ε

lista_param → lista_param , expresion | expresion

lista_param → expresion W

W → ,expresion W | ε

casos → caso num: sentencia casos | caso num: sentencia

casos → caso num: sentencia

D → casos | ε

iv En caso de factorizar incluir el proceso.

sentencia → si e_bool entonces sentencia fin
| si e_bool entonces sentencia sino
sentencia fin | mientras e_bool hacer
sentencia fin
| hacer sentencia mientras e_bool;
| segun (variable) hacer casos predeterminado fin
| variable := expresion ;
| escribir expresion ;
| leer variable ; | devolver;
| devolver expresion;
| terminar;
| inicio sentencias fin

sentencia → si e_bool entonces sentencia A
| mientras e_bool hacer sentencia fin
| hacer sentencia mientras e_bool;
| segun (variable) hacer casos predeterminado fin
| variable := expresion ;
| escribir expresion ;
| leer variable ;
| devolver B
| terminar;
| inicio sentencias fin

A -> fin | sino sentencia fin

B → ; | expresion;

e_bool → S o e_bool | S y e_bool | S

S →(e_bool) | relacional | verdadero | falso | no e_bool

e_bool → S C

C → o e_bool | y e_bool | S

S →(e_bool) | relacional | verdadero | falso | no e_bool

casos → caso num: sentencia casos | caso num: sentencia

casos → caso num: sentencia D

D → casos | ε

A continuación se muestra la gramática libre de ambigüedad, recursividad por la izquierda y factorizada:

1. programa → declaraciones funciones
2. declaraciones → tipo lista_var; declaraciones

   |tipo_registro lista_var; declaraciones

   | ε
3. tipo_registro → estructura inicio declaraciones fin
4. tipo → base tipo_arreglo
5. base → ent | real | dreal | car | sin
6. tipo_arreglo → [num] tipo_arreglo | ε
7. lista_var → id L

L → ,id L | ε

8. funciones → def tipo id(argumentos) inicio declaraciones sentencias fin funciones | ε
9. argumentos → listar_arg | sin
10. lista_arg → arg M

M → ,arg M | ε

11. arg → tipo_arg id
12. tipo_arg → base param_arr
13. param_arr → [ ] param_arr | ε
14. sentencias → sentencia N

N → sentencia N | ε

15. sentencia → si e_bool entonces sentencia A

| mientras e_bool hacer sentencia fin

| hacer sentencia mientras e_bool;

| segun (variable) hacer casos predeterminado fin

| variable := expresion ;

| escribir expresion ;

| leer variable ;

| devolver B

| terminar;

| inicio sentencias fin

A → fin | sino sentencia fin

B → ; | expresion;

16. casos → caso num: sentencia D

D → caso | ε

17. predeterminado → pred: sentencia | ε

18. e_bool → S C

C → o e_bool | y e_bool | S

S →(e_bool) | relacional | verdadero | falso | no e_bool

19. relacional → expresion U

U → oprel relacional U | ε

20. oprel → > | < | >= | <= | <> | =

21. expresion →Q SUM_RES expresion | Q

Q → P MUL_DIV_MOD Q| P

P → (expresion) | variable | num | cadena | caracter | id(parametros)

SUM_RES → + | −

MUL_DIV_MOD → ∗ | / | %

22. variable → dato_est_sim | arreglo

23. dato_est_sim →id V

V → .id V | ε

24. arreglo → id [expresion] | arreglo [expresion]

25. parametros → lista_param | ε

26. lista_param → expresion W

27. W → ,expresion W | ε

(c) Implementación. Describir cómo está implementado su programa, las partes que lo componen (no es todo el código).

En el encabezado del programa lexer.l se encuentra declaradas las bibliotecas:

```
{%
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
%}
```

Posteriormente se definen las expresiones regulares para un identificador, cadena, números, carácter, operadores relacionales, comentarios, espacios y símbolo de asignación.

En la tercera parte del programa se incluyen los procedimientos auxiliares que nos permiten indicar el token de cada uno de los terminales que se retornan en cuando se identifique algún símbolo o palabra reservada según las expresiones regulares definidas anteriormente o aquellas indicadas antes de las llaves, por ejemplo:

```
[Ee][Nn][Tt][Oo][Nn][Cc][Ee][Ss]          {return ENTONCES;}

[Mm][Ii][Ee][Nn][Tt][Rr][Aa][Ss]        {return MIENTRAS;}

[sS][iI][nN][oO]                        {return SINO;}

[hH][aA][cC][eE][rR]                    {return HACER;}
```

Finalmente se incluye la función error que indica la existencia de algún error léxico la función yywrap que retorna un 1 cuando ya no va a leer otro archivo de la entrada.

En el encabezado del programa parser.y se encuentra declaradas las bibliotecas:

```
{%

#include <stdio.h>

#include <string.h>

%}
```

Adicionalmente se declaran variables y funciones auxiliares.

Posteriormente se encuentran las declaraciones en yacc que nos permiten definir los tokens de cada uno de los terminales.

En la tercera parte del programa se incluyen los procedimientos auxiliares que nos permiten mostrar la estructura de cada no terminal que conforman la gramática, haciendo uso del signo $, que nos permite hacer referencia al valor devuelto por el componente especificado por el número de token en el lado derecho de cada una de las reglas, que se lee de izquierda a derecha.

Finalmente se incluye la función error que indica la existencia de algún error sintáctico, la función nuevaTemp que nos permite crear una nueva temporal y finalmente la función nuevaEtiqueta que nos permite crear una nueva etiqueta.

(d) Forma de ejecutar el programa.

./lexer

(e) Utilizar el analizador léxico obtenido en el primer programa.

(f) Se debe incluir un archivo main.c donde se encuentre la función principal para ejecutar el programa.

(g) Cada función tiene que estar documentada con la fecha y persona que la programó. Además de una muy breve descripción de lo que hace la función.

(h) En caso de hacer cambios en alguna función agregar fecha de modificación y nombre de quien la modificó.

Gramática

sin: significa sin tipo, car: tipo carácter

1. programa → declaraciones funciones
2. declaraciones → tipo lista_var; declaraciones
   | tipo registro lista_var; declaraciones
   | *ε*
3. tipo_registro → estructura inicio declaraciones fin
4. tipo → base tipo_arreglo
5. base → ent | real | dreal | car | sin
6. tipo_arreglo → [num] tipo arreglo | *ε*
7. lista_var → lista_var, id | id
8. funciones → def tipo id(argumentos) inicio declaraciones sentencias fin funciones | *ε*
9. argumentos → listar arg | sin
10. lista arg → lista arg, arg | arg

11. arg → tipo_arg id

12. tipo _arg → base param_arr

13. param arr → [ ] param_arr | *ε*

14. sentencias → sentencias sentencia | sentencia

15. sentencia → si e_bool entonces sentencia fin
    | si e_bool entonces sentencia sino
    sentencia fin | mientras e_bool hacer
    sentencia fin
    | hacer sentencia mientras e_bool;
    | segun (variable) hacer casos predeterminado fin
    | variable := expresion ;
    | escribir expresion ;
    | leer variable ; | devolver;
    | devolver expresion;
    | terminar;
    | inicio sentencias fin

16. casos → caso num: sentencia casos | caso num: sentencia

17. predeterminado → pred: sentencia | *ε*

18. e_bool → e_bool o e_bool | e_bool y e_bool | no e_bool | (e_bool)
    | relacional | verdadero | falso

19. relacional → relacional oprel relacional | expresion

20. oprel → > | < | **>=** | **<=** | <> | **=**

21. expresion → expresion oparit expresion
    | expresion % expresion | (expresion) | id
    | variable | num | cadena | caracter | id(parametros)

22. oparit → **+** | **−** | ∗ | /

23. variable → dato_est_sim | arreglo

24. dato_est_sim → dato_est sim .id | id

25. arreglo → id [expresion] | arreglo [expresion]

26. parametros → lista_param | *ε*

27. lista param → lista_param, expresion | expresion