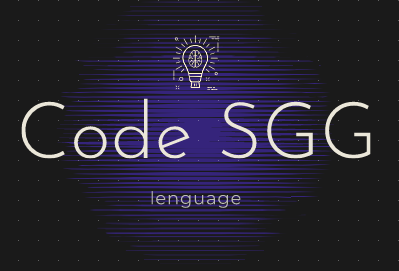
1. **Nombre del lenguaje**

****

Code SGG es un lenguaje de programación desarrollado por el equipo conformado por Gonzalez Joel, Segovia Grabois Lihuen.

El nombre es un acrónimo que hace referencia a las iniciales de los apellidos de los creadores. Es un lenguaje amigable con el usuario, simple de utilizar, pero efectivo.

En el que un programa está basado en una secuencia de sentencias, que a su vez ésta puede ser una asignación, lectura o escritura. Este lenguaje utiliza números reales como valores para sus variables, permite la realización de expresiones aritméticas y operación binarias como adición, sustracción, producto, cociente, radicación y potenciación.También soportas listas de valores reales y operaciones sobre esta.

Guia de usuario:

Crear un archivo donde esta el ejecutable que se llama entrada.txt, cargar las intrucciones a ejecutar y luego ejecutar el analizadorsintaxtico.exe

1. **Definición formal de la sintaxis mediante una gramática en notación BNF.**

<Programa> ::= “var” <Variables> “{“ <Cuerpo> “}”

<Variables> ::= “id” “:” <Tipo> < IdVar>

< IdVar > ::= “,” “ id” “:” <Tipo> <IdVar> | epsilon

<Tipo> ::= “real” | “listaReal”

<Cuerpo> ::= <Sent> “;” <Cuerpo> | epsilon

<Sent> ::= “id” “=” <U> | “leer” “(“ “cadena” “,” “id” “)” | “escribir” “(“ “cadena” “,” <expArit> “)” | “Si” <Condiciones> “{“ <Cuerpo> “}” <Sino> | “mientras” <Condiciones> “{“ <Cuerpo> “}” | “agregar” “(“ “id” “,” <expArit> “)” | “eliminar” “(“ “id” “,” <expArit>) | “reemplazar” “(“ “id” “,” <expArit> “,” <expArit> “)”

<U> ::= <expArit> | “[“ <L> “]”

<L> ::= <expArit> <T> | epsilon

<T> ::= “,” <expArit> <T> | epsilon

<Condiciones> ::= <Cond> <Z><W>

<W> ::= “or” <Condiciones> | epsilon

<Z> ::= “and” <Cond> <Z> | epsilon

<Cond> ::= “not” <Cond> | “[“ <Condiciones> “]” | <expArt> “opRel” <expArt>

<Sino> ::= “Sino” “{“ <Cuerpo> “}” | epsilon

<expArit> ::= <A><B>

<B> ::= “+”<A><B> | “-”<A><B> | epsilon

<A> ::= <C><D>

<D> ::= “\*”<C><D> | “/”<C><D> | epsilon

<C> ::= “root””(“<expArit>”)”<E> | <F><E>

<E> ::= “^”<F><E> | epsilon

<F> ::= “id” | “constante” | “(“ <expArit> “)” | “-”<F> | “cantidad” “(“ “id” “)” | “elemento” “(“ “id” “,” <expArit> “)”

1. **Gramática modificada LL(1) y TAS.**

Programa -> var Variables {Cuerpo}

Variables -> id : Tipo IdVar

IdVAr-> ,id : Tipo IdVar | epsilon

Tipo -> real | listaReal

Cuerpo -> Sent ; Cuerpo | epsilon

Sent -> id = U | leer (cadena, id) | escribir(cadena, expArit) | Si Condiciones {Cuerpo} Sino | mientras Condiciones {Cuerpo} | agregar( id ,expArit) | eliminar(id, expArit) | reemplazar(id, expArit, expArit)

U -> expArit | [L]

L -> expArit T | epsilon

T -> , expArit T | epsilon

Condiciones -> Cond ZW

W -> or Condiciones | epsilon

Z -> and Cond Z | epsilon

Cond -> not Cond | [Condiciones] | expArt opRel expArt

Sino -> Sino {Cuerpo} | epsilon

expArit -> AB

B -> +AB | -AB | epsilon

A -> CD

D -> \*CD | /CD | epsilon

C -> root(expArit)E | FE

E -> ^FE | epsilon

F -> id | constante | (expArit) | -F | cantidad (id) | elemento(id, expArit)

* Los corchetes([,]) que abren y cierran, sirven para cambiar el orden de evaluación de una condición.

TAS

**Para** Programa -> var Variables {Cuerpo}

Primero (var Variables {Cuerpo}) = {var}

**Para** Variables -> id : Tipo IdVar

Primero (id : Tipo IdVar)= {id}

**Para** IdVar -> ,id : Tipo IdVar

Primero (,id : Tipo IdVar)= {,}

**Para** IdVar -> epsilon

Primero (epsilon)= {epsilon}

Siguiente ( IdVar)= { { }

Siguiente (Variables)= { { }

**Pata** Tipo -> real

Primero (real)={rela}

**Pata** Tipo -> listaReal

Primero (listaReal)={listaReal}

**Para** Cuerpo -> Sent ; Cuerpo

Primero (Sent ; Cuerpo)= {id, leer, escribir, mientras, si, agregar, eliminar, reemplazar}

Primero (Sent)= {id, leer, escribir, mientras, si, agregar, eliminar, reemplazar}

**Para** Cuerpo ->epsilon

Primero (epsilon)= {epsilon}

Siguiente (Cuerpo)= { } }

**Para** Sent -> id = U

Primero (id = U) = {id}

**Para** Sent -> leer (cadena, id)

Primero (leer (cadena, id))= {leer}

**Para** Sent -> escribir (cadena,expArit)

Primero (escribir(cadena, expArit) )= {escribir}

**Para** Sent -> mientras Condiciones {Cuerpo}

Primero (mientras Condiciones {Cuerpo} )= {mientras}

**Para** Sent -> Si Condiciones {Cuerpo} Sino

Primero ( Si Condiciones {Cuerpo} Sino)= {Si}

**Para** Sent -> agregar( id ,expArit)

Primero ( agregar( id ,expArit))= {agregar}

**Para** Sent -> eliminar ( id ,expArit)

Primero ( eliminar ( id ,expArit))= {eliminar}

**Para** Sent -> reemplazar(id, expArit, expArit)

Primero ( reemplazar(id, expArit, expArit))= {reemplazar}

**Para** U -> expArit

Primero (expArit) = {root, id, constante, ( , - , cantidad, elemento}

Primero (AB)= {root, id, constante, ( , - , cantidad, elemento}

Primero (A)= {root, id, constante, ( , - , cantidad, elemento}

Primero (CD)= {root, id, constante, ( , - , cantidad, elemento}

Primero (C)= {root, id, constante, ( , - , cantidad, elemento}

Primero (FE)= {id, constante, ( , - , cantidad, elemento}

Primero (F)= {id, constante, ( , - , cantidad, elemento}

**Para** U -> [L]

Primero ([L])={[}

**Para** L -> expArit T

Primero (expArit T) =

Primero (expArit) = {root, id, constante, ( , - , cantidad, elemento}

Primero (AB)= {root, id, constante, ( , - , cantidad, elemento}

Primero (A)= {root, id, constante, ( , - , cantidad, elemento}

Primero (CD)= {root, id, constante, ( , - , cantidad, elemento}

Primero (C)= {root, id, constante, ( , - , cantidad, elemento}

Primero (FE)= {id, constante, ( , - , cantidad, elemento}

Primero (F)= {id, constante, ( , - , cantidad, elemento}

**Para** L -> épsilon

Primero (epsilon) = {epsilon}

Siguiente (L)= {]}

**Para** T -> , expArit T

Primero (, expArit T) = {,}

**Para** T -> épsilon

Primero (épsilon )= {épsilon}

Siguiente (T)= {]}

Siguiente (L)= {]}

**Para** Condiciones -> Cond Z W

Primero (Cond Z W)= { not, [, root, id, constante, ( , - , cantidad, elemento }

Primero (Cond )= { not, [, root, id, constante, ( , - , cantidad, elemento }

Primero (expArit) = {root, id, constante, ( , - , cantidad, elemento}

Primero (AB)= {root, id, constante, ( , - , cantidad, elemento}

Primero (A)= {root, id, constante, ( , - , cantidad, elemento}

Primero (CD)= {root, id, constante, ( , - , cantidad, elemento}

Primero (C)= {root, id, constante, ( , - , cantidad, elemento}

Primero (FE)= {id, constante, ( , - , cantidad, elemento}

Primero (F)= {id, constante, ( , - , cantidad, elemento}

**Para** W -> or Condicones

Primero (or Condiciones )= {or}

**Para** W -> épsilon

Primero (épsilon) = { épsilon }

Siguiente (W) = { {, ] }

Siguiente (Condiciones) = { { , ] }

**Para** Z -> and Cond Z

Primero (and CondZ Z) = { and }

**Para** Z -> épsilon

Primero (épsilon) = { épsilon }

Siguiente ( Z ) ={ {, ], or }

Primero (W) = { { , ], or }

**Para** Cond -> not Cond

Primero (not Cond)= {not}

**Para** Cond -> [Condiciones]

Primero ([Condiciones]= { [ }

**Para** Cond -> expArit opRel expArit

Primero (expArit opRel expArit)= {root, id, constante, ( , - , cantidad, elemento}

Primero (expArit) = {root, id, constante, ( , - , cantidad, elemento}

**Para** Sino -> sino {Cuerpo}

Primero (sino {Cuerpo}) = {sino}

**Para** Sino -> epsilon

Primero (epsilon)= {epsilon }

Siguiente (Sino)= { ;}

Siguiente (Sent)= { ;}

**Para** expArit -> AB

Primero (AB) = {rot, id, constante, (, -, cantidad, elemento}

Primero (A) = {rot, id, constante, (, -, cantidad, elemento}

Primero (CD)= {rot, id, constante, (, -, cantidad, elemento}

Primero (C)= {rot, id, constante, (, -, cantidad, elemento}

**Para** B -> +AB

Primero (+AB)= {+}

**Para** B -> -AB

Primero (-AB) = {-}

**Para** B -> epsilon

Primero (epsilon)= { epsilon }

Siguiente (B) = { ;,), opRel, {,] , and , or , **,** }

Siguiente (expArit)= {), opRel, {,] , and , or ,**,** }

Primero (T) = { **,**, L}

Siguiente (L) = {]}

Siguiente (U) = {;}

Siguiente (Sent) = {;}

**Para** A -> CD

Primero (CD)= {rot, id, constante, (, -, cantidad, elemento}

Primero (C)= {rot, id, constante, (, -, cantidad, elemento}

**Para** D -> \*CD

Primero (\*CD)= {\*}

**Para** D -> /CD

Primero (/CD )= {/ }

**Para** D -> epsilon

Primero (epsilon)={epsilon }

Siguiente (D)= { +,-, ), opRel, **,**, ], ; }

Siguiente (A)= {+,-, ), opRel, **,**, ], ; }

Primero (B)= {+,-}

Siguiente (B)={ ), opRel, **,**, ], ; }

**Para** C -> root(expArit)E

Primero (root(expArit)E) = {root}

**Para** C -> FE

Primero (FE) = { id, constante, ( , - , cantidad, elemento }

Primero (F) = { id, constante, ( , - , cantidad, elemento }

**Para** E -> ^FE

Primero (^FE) = {^ }

**Para** E -> epsilon

Primero (epsilon)= {epsilon}

Siguiente (E)= { \*, /, +, -, ;, ), opRel, {,] , and , or, **,** }

Siguiente (C)= { \*, /, +, -, ), opRel, T, **,** }

Primero (D)= { \*, /, +, -, ), opRel, T, **,** }

Siguiente (D) = {+,-, ), opRel, T, **,** }

Siguiente (A)= {+,-, ), opRel, T, **,** }

Primero (B)= {+,-, ), opRel, T, **,** }

Siguiente (B)={ ), opRel, **,**, ], ; }

Siguiente (expArit) = { ), opRel, ], **,** }

Siguiente (Cond)={Z}

Primero(Z)={and, epsilon}

Siguiente(Z)={W}

Primero(W)={or, epsilon}

Siguiente(W)={condiciones}

Siguiente(condiciones)={ ], { }

**Para** F -> id

Primero (id)= { id }

**Para** F -> constante

Primero (constante)= { constante}

**Para** F -> (expArit)

Primero ((expArit))= { ( }

**Para** F -> -F

Primero (-F)= { -}

**Para** F -> cantidad (id)

Primero (cantidad (id))= { cantidad}

**Para** F -> elemento (id, expArit)

Primero (elemento (id, expArit)) = { elemento}

1. **Descripción de la semántica asociada.**

Programa -> var Variables {Cuerpo}

evalPrograma(arbol,estado) //arbol: puntero arbol //estado: lista de nombres variables, tipo y el valor de la variable

evalVariables(arbol.hijos[2], estado)

evalCuerpo(arbol.hijos[4], estado)

Variables -> id : Tipo IdVar

evalVariables(arbol, estado)

evalTipo(arbol.hijos[3], tipodevar)

agregarvariable(estado, arbol.hijos[1].lexema, tipodevar )

evalIdVar(arbol.hijos[4],estado)

IdVAr-> ,id : Tipo IdVar | epsilon

evalIdVar(arbol, estado)

if produccion<>epsilon

evalTipo(arbol.hijos[4], tipodevar)

agregarvariable(estado, arbol.hijos[2].lexema, tipodevar )

evalIdVar(arbol.hijos[5],estado)

Tipo -> real | listaReal

evalTipo (arbol, tipodevar)

if produccion = ‘real’

tipodevar=Treal

if produccion = ‘listaReal’

tipodevar=TlistaReal

Cuerpo -> Sent ; Cuerpo | epsilon

evalCuerpo(arbol, estado)

if produccion<>epsilon

evalSent(arbol.hijos[1], estado)

evalCuerpo(arbol.hijos[3], estado)

Sent -> id = U | leer (cadena, id) | escribir(cadena, expArit) | Si Condiciones {Cuerpo} Sino | mientras Condiciones {Cuerpo} | agregar( id ,expArit) | eliminar(id, expArit) | reemplazar(id, expArit, expArit)

evalSent(arbol, estado)

If produccion= ’id’

evalU(arbol.hijo[3], estado, resultado) ///resultado: definir un registro con tipos, real, lista

asignarEstado(estado, arbol.hijo[1].lexema,resultado)

If produccion= ‘leer’

var

X:Real

write(arbol.hijos[3].Lexema)

read(X)

asignarEstado(estado, arbol.hijo[5].lexema,resultado)

If produccion = ‘escribir’

evalexpArit(arbol.hijo[5], estado, resultado)

writeln(arbol.hijo[3].lexema,’ ’, resultado)

If produccion= ‘si’

evalCondiciones (arbol.hijo[2], estado, resultado)

If resultado

evalCuerpo (arbol.hijo[4], estado)

else

evalSino (arbol.hijo[6], estado)

If produccion= ‘mientras’

evalCondiciones (arbol.hijo[2], estado, resultado)

while resultado do

evalCuerpo (arbol.hijo[4], estado)

evalCondiciones (arbol.hijo[2], estado, resultado)

U -> expArit | [L]

evalU(arbol, estado, resultado)

If produccion= ‘expArit’

evalexpArit(arbol.hijo[1], estado, resultado)

If produccion= ‘[L]’

crearlista(resultado.L)

evalL(arbol.hijo[2], estado, resultado)

L -> expArit T | epsilon

evalL(arbol, estado, resultado)

if produccion<>epsilon

evalexpArit (arbol.hijo[1], estado, resultado2)

agregarLista(resultado.L, resultado2);

evalT (arbol.hijo[2], estado, resultado)

T -> , expArit T | epsilon

evalT (arbol, estado, resultado)

if produccion<>epsilon

evalexpArit (arbol.hijo[2], estado, resultado2)

agregarLista(resultado.L, resultado2);

evalT (arbol.hijo[3], estado, resultado)

Condiciones -> Cond ZW

evalCondiciones (arbol, estado,resultado)

evalCond(arbol.hijo[1], estado, operando1)

evalZ(arbol.hijo[2],estado, operando1, resultadoAND)

evalW(arbol.hijo[3],estado, resultadoAND, resultado)

W -> or Condiciones | epsilon

evalW (arbol, estado, operando1, resultado)

if produccion<>epsilon

evalCondiciones(arbol.hijo[2], estado,operando2)

resultado := operando1 or operando2

else

resultado := operando1

Z -> and Cond Z | epsilon

evalZ (arbol, estado, operando1, resultado)

if produccion<>epsilon

evalCond(arbol.hijo[2], estado, operando2)

aux:= operando1 AND operando2

evalZ(arbol.hijo[3], estado, aux, resultado)

else

resultado:=operando1

Cond -> not Cond | [Condiciones] | expArit opRel expArit

evalCond(arbol, estado, resultado)

If produccion= ‘Not’

evalCond(arbol.hijo[2], estado, resultado)

resultado = not resultado

If produccion= ‘[condiciones]’

evalCondiciones(arbol.hijo[2], estado, resultado)

If produccion= ‘expArit opRel expArit’

evalexpArit(arbol.hijo[1], estado,resultado1)

evalexpArit(arbol.hijo[3], estado, resultado2)

if opRel = <

resultado = resultado1 < resultado2

if opRel = <=

resultado = resultado1 <= resultado2

if opRel = >

resultado = resultado1 > resultado2

if opRel = >=

resultado = resultado1 >= resultado2

if opRel = <>

resultado = resultado1 <> resultado2

if opRel = =

resultado = resultado1 = resultado2

Sino -> Sino {Cuerpo} | epsilon

evalSino(arbol, estado)

if produccion<>epsilon

evalCuerpo(arbol.hijo[3], estado)

expArit -> AB

evalexpArit(arbol,estado, resultado)

evalA(arbol.hijo[1],estado,resultadoA)

evalB(arbol.hijo[2], estado, resultadoA, resultado)

B -> +AB | -AB | epsilon

evalB(arbol,estado, operando1,resultado)

If produccion = ‘+’

evalA(arbol.hijo[2],estado,resultadoA)

suma:= operando1 + resultadoA

evalB(arbol.hijo[3], estado, suma, resultado)

If produccion = ‘-’

evalA(arbol.hijo[2],estado,resultadoA)

resta:= operando1 - resultadoA

evalB(arbol.hijo[3], estado, resta, resultado)

If produccion = epsilon

resultado:=operando1

A -> CD

evalA(arbol,estado, resultado)

evalC(arbol.hijo[1],estado,resultadoC)

evalD(arbol.hijo[2], estado, resultadoC, resultado)

D -> \*CD | /CD | epsilon

evalD(arbol,estado, operando1,resultado)

If produccion = ‘\*’

evalC(arbol.hijo[2],estado,resultadoC)

produc:= operando1 \* resultadoC

evalD(arbol.hijo[3], estado, produc, resultado)

If produccion = ‘/’

evalC(arbol.hijo[2],estado,resultadoC)

div:= operando1 / resultadoC

evalD(arbol.hijo[3], estado, div, resultado)

If produccion = epsilon

resultado:=operando1

C -> root(expArit)E | FE

evalC(arbol, estado, resultado)

If produccion = ‘root’

evalexpArit(arbol.hijo[3], estado, resultado1)

aux:= sqrt(resultado1)

evalE(arbol.hijo[5], estado, aux, resultado)

If produccion = ‘FE’

evalF(arbol.hijo[1], estado,resultado1)

evalE(arbol.hijo[2], estado, resultado1, resultado2 )

E -> ^FE | epsilon

evalE(arbol, estado, operando1, resultado)

if produccion<>epsilon

evalF(arbol.hijo[2], estado, operando2)

aux:= power(operando1,operando2)

evalE(arbol.hijo[3], estado, aux, resultado)

else

resultado=operando1

F -> id | constante | (expArit) | -F | cantidad (id) | elemento(id, expArit)

evalF(arbol, estado, resultado)

If produccion = ‘id’

obtenervalordeID (arbol.hijo[1].lexema , estado, resultado)

If produccion = ‘constante’

pasarentero (arbol.hijo[1].lexema , resultado)

If produccion = ‘(expArit)’

evalexpArit(arbol.hijo[2], estado, resultado)

if produccion = ‘-’

evalF(arbol.hijo[2], estado, resultado)

resultado = resultado \* -1

If produccion = ‘cantidad’

obtenervalordeID(arbol.hijo[3].lexema, estado, resultado1)

resultado := cantidadElementos (resultado1)

If produccion = ‘elemento’

evalexpArit(arbol.hijo[5], estado, pos)

obtenervalordeID(arbol.hijo[3].lexema, estado,Lista)

resultado := obtenerElemento (Lista, pos)