PROYECTO INTEGRADOR
Sintaxis y Semántica de los Lenguajes
UTN FRCU

DRW LANGUAGE

ALUMNOS
Dutra, Francisco
Retamero, Marcos Sebastián

PROFESORES
Pascal, Andrés
Álvarez, Claudia

CURSO
Segundo año, ingeniería en sistemas de la información

AÑO LECTIVO
2021

Breve introducción.

DRW es un lenguaje que imita al lenguaje pascal, con ligeros cambios a la hora de escribirse tanto en la definición de las variables, como en el cuerpo del programa y otros ámbitos. A la hora de querer diseñarse un programa debemos seguir una determinada cantidad de reglas, para facilitarse esta tarea, puede observarse la gramática (CFG) con la cual se construyó lenguaje.

Programar en este lenguaje nos permite definir variables tipo lista y reales, con las que se pueden resolver diferentes algoritmos, y cuenta con distintas palabras reservadas para procedimientos sobre listas, las cuales pascal no posee.

Para todo esto, el programa utiliza un analizador léxico y sintáctico que comprueban que el código este bien escrito, y a continuación un evaluador lo ejecuta y devuelve los resultados., todo esto mediante Lazarus.

¿Qué podemos hacer con DRW?

Podemos definir un numero finito de variables de tipo lista o real, y trabajar con ellas en estructuras cíclicas o condicionales. Escribir las sentencias que se necesiten, ya sea para:

-Resolver expresiones aritméticas y asignarlas a variables, o imprimirlas.

-Agregar valores reales o variables a listas, remplazar y eliminar sus elementos.

-Usar estructuras condicionales con varias condiciones y operadores lógicos and, or y not.

¿Cómo se usa?

Como ya se comentó en la introducción, el código cuenta con un analizador léxico que nos facilita la construcción del lenguaje, para entender cómo se usa, primero debemos entender el funcionamiento del código.

En la sección léxica del proyecto se definió una tabla de símbolos que nos permite identificar cada lexema con su componente léxico (lexema es la cadena de caracteres que concuerda con un patrón que describe un componente léxico). Para saber que lexema corresponde con cada componente léxico se implementaron autómatas finitos. Para la parte sintáctica del código se incorporó una tabla de símbolos y una TAS que permiten al programa verificar si el código que escribimos es correcto o tiene algún defecto, en caso de que lo haya, el analizador sintáctico avisara al usuario cual es el error en su código. El evaluador, como lo dice su nombre, analiza cada una de las producciones de la gramática en base a un árbol de derivación y permite que se realicen las operaciones dentro de pascal.

Desde el punto de vista del usuario, se debe crear un archivo de texto (.txt) en **C:\DRW** (esto puede modificarse) y dentro escribir el código que desea programar, como guía de ayuda se pueden realizar derivaciones de la gramática para asegurarse que el código sea correcto. Luego solo se debe ejecutar el programa que reconocerá si el código que se escribió es válido y devolverá, en ese caso, resultados.

Gramática en notación BNF

<programa> ::= <variables> <bloque>

<variables>::= epsilon | <variables> “id” <tipo>

<bloque>::= “begin” <sentencia> ”end”

<tipo> ::= “lista” | “real”

<sentencia> ::= <sentencia> <sent> | <sent>

<sent> ::= <asig>”;” | <condicional>”;” | <para>”;” | <mientras>”;” | <leer>”;” | <escribir>”;” |
“agregar””(””id”, <expAritm>”)””;” | “eliminar””(“”id”, <expAritm>”)””;” |
“reemplazar””(“”id”, <expAritm>, <expAritm>”)””;”

<asig>::= “id” <opAsig> <A>

<A>::= <expAritm> | <constLISTA>

<constLISTA>::= “[“<elementos>”]” | “[“”]”

<elementos>::= <expAritm>,<elementos> | <expAritm>

<expAritm>::= <expAritm> “+” <T> | <expAritm> “-” <T> | <T>

<T>::= <T> “*” <D> | <T> “/” <D> | <D>

<D> ::= <D>”^”<J> | “sqrt””(“<D>”)”| <val> | “(”<expAritm>”)”

<val>::= “id” | “const” | “Cantidad””(“”id””)” | “elemento””(“”id”, <expAritm>”)”

<condicional>::= “if” <disyunción> “then” <bloque> <E>

<disyunción>::= <disyunción> “or” <conjunción> | <conjunción>

<conjunción> ::= <conjunción> “and” <negacion> | <negacion>

<negación> ::= “not” <cond> | <cond>

<cond> ::= <expAritm> <oprel> <expAritm> | “[“<disyunción>”]”

<oprel>::= “<” | “>” | “=” | “>=” | “<=”

<E>::= epsilon | “else” <bloque>

<para>::= “for” “id” “=” <expAritm> “to” <expAritm> “do” <bloque>

<mientras> ::= “while” “[“<cond>”]” “then” <bloque>

<leer >::= “read””(“”cadena”,”id””)”

<escribir>::= “write””(“”cadena”,<expAritm>”)”

Gramática modificada LL(1)

programa → variables bloque

variables → id tipo variables | epsilon

bloque → begin sentencia end

tipo → lista | real

sentencia → sent R

R→sent R | epsilon

sent → asig; | condicional; | para; | mientras; | leer; | escribir; | agregar(id, expAritm); | eliminar(id, expAritm); | reemplazar(id, expAritm, expAritm);

asig → id opAsig A

A → expAritm | constLISTA

constLISTA → [H]

H → elementos | epsilon

elementos → expAritmQ

Q → ,elementos | epsilon

expAritm → TS

S → +TS | -TS |epsilon

T → DZ

Z → *DZ | /DZ | epsilon

D → valY | sqrt(D)

Y → ^D | epsilon

val → id | const | Cantidad(id) | elemento(id, expAritm) | (expAritm)

condicional → if disyunción then bloque E

disyunción → conjunción P

P→ or conjuncion P | epsilon

conjunción → negación L

L → and negación L | epsilon

negación → not negación| cond

cond → expAritm oprel expAritm | [disyunción]

oprel → < | > | = | >= | <=

E → epsilon | else bloque

bloque → begin sentencia end

para → for id = expAritm to expAritm do bloque

mientras → while disyunción then bloque

leer → read(cadena,id)

escribir → write(cadena,expAritm)

Descripción semántica de la gramática mediante pseudocódigo.

programa → variables bloque

```
evalprograma(arbol,estado)
        evalvariables(arbol.hijos[1],estado)
        evalbloque(arbol.hijos[2],estado)
```

variables → id tipo variables | epsilon

```
evalvariables(arbol,estado)
        if arbol.cant > 0
                evaltipo(arbol.hijos[2],estado,tipo)
                agregarvariable(estado,arbol.hijos[1].lexema,tipo)
                evalvariables(arbol.hijos[3],estado)
```

bloque → begin sentencia end

```
evalbloque(arbol,estado)
        evalsentencia(arbol.hijos[2],estado)
```

tipo → lista | real

```
evaltipo(arbol,estado,tipo)
        if arbol.hijos[1].simbolos=tlistas
                tipo:=tlistas
        else
                tipo:=treal
```

sentencia → sent R

```
evalsentencia(arbol,estado)
        evalsent(arbol.hijos[1],estado)
        evalR(arbol.hijos[2],estado)
```

R→sent R | epsilon

```
evalR(arbol,estado)
        if arbol.cant > 0
                evalsent(arbol.hijos[1],estado)
                evalR(arbol.hijos[2],estado)
```

sent → asig; | condicional; | para; | mientras; | leer; | escribir; | agregar(id, expAritm); | eliminar(id, expAritm); | reemplazar(id, expAritm, expAritm);

```
evalsent(arbol,estado)
    case arbol.hijos[1].simbolos of
        vasig: evalasig(arbol.hijos[1],estado)
        vcondicional: evalcondicional(arbol.hijos[1],estado,valordecondicion)
        vpara: evalpara(arbol.hijos[1],estado)
        vmientras: evalmientras(arbol.hijos[1],estado)
        vleer: evalleer(arbol.hijos[1],estado)
        vescribir: evalescribir(arbol.hijos[1],estado)
        tagregar:
            evalexpAritm(arbol.hijos[5],estado,valor)
            lista:=obtenerlista(estado,arbol.hijos[3].lexema)
            agregar(lista,valor)
        teliminar:
            evalexpAritm(arbol.hijos[5],estado,posicion)
            lista:=obtenerlista(estado,arbol.hijos[3].lexema)
            eliminar(lista,posicion)
        treemplazar:
            evalexpAritm(arbol.hijos[5],estado,posicion)
            lista:=obtenerlista(estado,arbol.hijos[3].lexema)
            evalexpAritm(arbol.hijos[7],estado,valor)
            reemplazar(lista,posicion,valor)
```

asig → id opAsig A

```
evalasig(arbol,estado)
    evalA(arbol.hijos[3],estado,arbol.hijos[1].lexema)
```

```
A → expAritm | constLISTA
evalA(arbol,estado,lexema)
        if arbol.hijos[1]=expAritm
                evalexpAritm(arbol.hijos[1],estado,valor)
                asignarvalor(estado,lexema,valor)
        else
                evalconstLista(arbol.hijos[1],estado,lista)
                asignarlista(estado,lexema,lista)


constLISTA → [H]
evalconstLista(arbol,estado,lista)
        new(lista)
        inicializarlista(lista)
        evalH(arbol.hijos[2],estado, lista)

H → elementos | epsilon
evalH(arbol,estado,lista)
        if arbol.cant > 0
                evalelementos(arbol.hijos[1],estado,lista)

elementos → expAritmQ
evalElementos(arbol,estado,lista)
        evalexpAritm(arbol.hijos[1],estado,valor)
        evalQ(arbol.hijos[2],estado)
        agregar(lista,valor)

Q → ,elementos | epsilon
evalQ(arbol,estado,lista)
                if arbol.cant > 0
                evalelementos(arbol.hijos[2],estado,lista)
```

```
expAritm → TS
evalexpAritm(arbol,estado,valor)
        evalT(arbol.hijos[1],estado,operando1)
        evalS(arbol.hijos[2],estado,operando1, valor)

S → +TS | -TS | epsilon
evalS(arbol,estado,operando1, valor)
        if arbol.cant = 0
                valor:=operando1
        else
                if arbol.hijos[1].simbolo = tsuma then
                        evalT(arbol.hijos[2],estado,operando2)
                        aux:=operando1 + operando2;
                        evalS(arbol.hijos[3],estado,aux, valor)
                else
                        evalT(arbol.hijos[2],estado,operando2)
                        aux:=operando1 - operando2;
                        evalS(arbol.hijos[3],estado,aux, valor)

T → DZ
evalT(arbol,estado,valor)
        evalD(arbol.hijos[1],estado,operando1)
        evalZ(arbol.hijos[2],estado,operando1,valor)

Z → *DZ | /DZ | epsilon
evalZ(arbol,estado,operando1, valor)
        if arbol.cant = 0
                valor:=operando1
        else
                if arbol.hijos[1].simbolo = tmultiplicar then
                        evalD(arbol.hijos[2],estado,operando2)
                        aux:=operando1 * operando2;
                        evalZ(arbol.hijos[3],estado,aux, valor)
                else
```

```
                    evalD(arbol.hijos[2],estado,operando2)
                    aux:=operando1 / operando2;
                    evalZ(arbol.hijos[3],estado,aux, valor)
```

D → valY | sqrt(D)

```
evalD(arbol,estado,valor)
        case arbol.hijos[1].simbolos
                vval: evalval(arbol.hijos[1],estado,valor)
                      evalY(arbol.hijos[2],estado)
                tsqrt:
                     evalD(arbol.hijos[3],estado,valor)
```

Y → ^D | epsilon

```
evalY(arbol,estado)
                if arbol.cant > 0
                        evalelementos(arbol.hijos[2],estado,lista)
```

val → id | const | Cantidad(id) | elemento(id, expAritm) | (expArit)

```
        evalVal(arbol,estado,valor)
                case arbol.hijos[1].simbolos
                        id: valor:=obtenervalor(estado, arbol.hijos[1].lexema)
                        const: val(arbol.hijos[1].lexema,valor, error)
                        cantidad:
                                lista:=obtenerlista(estado, arbol.hijos[3].lexema)
                                valor:= cantidad(lista)
                        elemento:
                                lista:=obtenerlista(estado, arbol.hijos[3].lexema)
                                evalexpAritm(arbol.hijos[5],estado, posicion)
                                valor:= elemento(lista, posicion)
                        tparenti:
                                evalexpAritm(arbol^.hijos[2],estado,valor);
```

```
 condicional → if disyunción then bloque E
evalcondicional(arbol,estado, valordecondicion)
        evaldisyuncion(arbol.hijos[2],estado, valordecondicion)
        evalbloque(arbol.hijos[4],estado)
        evalE(arbol.hijos[5],estado)

disyunción → conjunción P
evaldisyuncion(arbol,estado, valordecondicion)
        evalconjuncion(arbol.hijos[1],estado, valordecondicion)
        evalP(arbol.hijos[2],estado)

P→ or conjuncion P | epsilon
evalP(árbol,estado, aux ,valordecondicion);
        if arbol.cant > 0
                evalconjuncion(arbol.hijos[2],estado, valordecondicion)
                evalP(arbol^.hijos[3],estado, aux ,valordecondicion);

conjunción → negación L
evalconjuncion(arbol,estado)
        evalnegacion(arbol.hijos[1],estado)
        evalL(arbol.hijos[2],estado)
L → and negación L | epsilon
        evalL(arbol,estado)
                if arbol.cant > 0
                evalnegacion(árbol,estado,valordecondicion)
                evalL(arbol.hijos[3],estado)

negación → not cond | cond
evalnegacion(árbol,estado,valordecondicion)
        if arbol^.hijos[1]^.simbolo=tnot   then
                begin
                evalcond(arbol^.hijos[2],estado,valordecondicion) ;
                valordecondicion:=not valordecondicion
                end
```

```
        else
                evalcond(arbol^.hijos[1],estado,valordecondicion)
        end;
```

cond → expAritm oprel expAritm | [disyuncion]

```
evalcond(arbol,estado,valor)
        if arbol.hijos[1].simbolos = vexpAritm
                evalexpAritm(arbol.hijos[1],estado,operando1)
                evalexpAritm(arbol.hijos[3],estado,operando2)
                evaloprel(arbol.hijos[2],estado)
        else
                evalcond(arbol.hijos[2],estado,valor)
```

oprel → < | > | = | >= | <=

```
evalOprel(arbol,estado)
        case arbol.hijos[1].lexema
                < :valor:=operando1<operando2
                >:valor:=operando1>operando2
                =:valor:=operando1=operando2
                <=:valor:=operando1<=operando2
                >=:valor:=operando1>=operando2
```

E → epsilon | else bloque

```
        evalE(arbol,estado)
                if arbol.cant > 0
                        evalbloque(arbol.hijos[2],estado)
```

bloque → begin sentencia end

```
evalbloque(arbol,estado)
        evalsentencia(arbol.hijos[2],estado)
```

```
para → for id = expAritm to expAritm do bloque
        evalPara(arbol,estado)
                evalExparit(arbol.hijos[4],estado,valorinicial)
                evalExparit(arbol.hijos[6],estado,valorfinal)
                aux:=arbol.hijos[2].lexema
                for aux := valorinicial to valorfinal do
                        evalBloque(arbol.hijos[8],estado)
                        arbol.hijos[2].lexema:=aux

mientras → while disyuncion then bloque
evalMientras(arbol,estado)
        evalDisyuncion(arbol.hijos[2],estado,valordecondicion)
        while valordecondicion do
                evalBloque(arbol.hijos[4], estado)
                evalDisyuncion(arbol.hijos[2],estado,valordecondicion)

leer → read(cadena,id)
evalLeer(arbol,estado)
        write(arbol^.hijos[3]^.lexema);
        readln(aux);
        AsignarValor (estado,arbol^.hijos[5]^.lexema,aux);

escribir → write(cadena,expAritm)
evalEscribir(arbol,estado)
        evalexparit(arbol.hijos[5],estado,valor)
        write(arbol^.hijos[3]^.lexema)
        write(valor)
```

Ejercicios sobre el lenguaje

1. Escribir un programa en este lenguaje que ingrese N números y los almacene en una lista. Luego calcular e imprimir la sumatoria, el promedio y la varianza de los valores almacenados.
2. Escribir un programa que ingrese valores en dos listas, ordenar sus elementos e intercalarlos en una tercer lista e imprimirla.
3. Escribir otro programa definido por el grupo. Escribir el enunciado, programar la solución y realizar pruebas.

1)

```
TLISTA lista
promedio real
cant real
I real
Sumatoria real
Varianza real
begin
TLISTA:_[2,1,3,5,6];
SUMATORIA:_ 0;
FOR I :_ 1 TO 5 DO
BEGIN
sumatoria:_ sumatoria + elemento(TLISTA, I);
END;
cant:_ cantidad(TLISTA);
promedio:_ sumatoria/cant;
FOR I :_ 1 TO 5 DO
BEGIN
Varianza:_ varianza + ((elemento(TLISTA, I) - Promedio)^2);
END;
varianza:_varianza/cant;
write("la sumatoria es: " ,sumatoria);
write("El promedio es:  " , promedio);
write("La varianza es:  " , varianza);
end
```

2)

```
lista1 lista
lista2 lista
lista3 lista
I real
J real
aux1 real
aux2 real
canttotal real
valor real
salir real
begin
lista1:_[2,6,4];
lista2:_[5,1,3,7,9,6];
for i :_ 1 to (cantidad(lista1)-1) do
begin
  for j:_ i+1 to cantidad(lista1) do
   begin
    if elemento(lista1,j)<elemento(lista1,i) then
    begin
     aux1:_elemento(lista1,j);
     aux2:_elemento(lista1,i);
     reemplazar(lista1, i, aux1);
     reemplazar(lista1, j, aux2);
    end;
  end;
end;
for i :_ 1 to (cantidad(lista2)-1) do
begin
  for j:_ i+1 to cantidad(lista2) do
   begin
    if elemento(lista2,j)<elemento(lista2,i) then
    begin
     aux1:_elemento(lista2,j);
     aux2:_elemento(lista2,i);
     reemplazar(lista2, i, aux1);
     reemplazar(lista2, j, aux2);
    end;
  end;
end;
i:_1;
j:_1;
while [i <= cantidad(lista1)] and [j <= cantidad(lista2)] then
begin

if elemento(lista1,i) < elemento(lista2,j) then
begin
 agregar(lista3,elemento(lista1,i));
 i:_i+1;
end
 else
```

```
  begin
   agregar(lista3,elemento(lista2,j));
    j:_j+1;
  end;

end;

while i<=cantidad(lista1) then
begin
  agregar(lista3,elemento(lista1,i));
  i:_i+1;
end;

while j<=cantidad(lista2) then
begin
  agregar(lista3,elemento(lista2,j));
  j:_j+1;
end;

for i:_1 to cantidad(lista3) do
begin
aux1:_elemento(lista3,i);
write("la lista 3 es: " ,aux1);
end;
end
```

3) Consigna: Para 10 alumnos de un curso, se necesita un programa que determine si cada uno de ellos reprobó, regularizo o promociono la materia en base a dos listas, una con las notas del primer cuatrimestre de la materia, y la otra con las notas del segundo cuatrimestre, de las cuales se saca el promedio, luego imprimir el número de reprobados, regularizados y promocionados, teniendo en cuenta que si la nota < 4 se reprueba, si 4 <= nota < 7 se regulariza, y si nota => 7 se promociona.

```
lista1 lista
lista2 lista
i real
nota real
promedio real
reprobados real
regularizados real
promocionados real
begin
for i:_1 to 10 do
begin
write("Notas del primer cuatrimestre, alumno N: " ,i);
read("ingrese la nota: " ,nota);
agregar(lista1, nota);
end;
for i:_1 to 10 do
begin
write("Notas del segundo cuatrimestre, alumno N: " ,i);
read("ingrese la nota: " ,nota);
agregar(lista2, nota);
end;
for i:_1 to 10 do
begin
promedio:_(elemento(lista1, i)+elemento(lista2,i))/2;
if promedio < 4 then
begin
reprobados:_reprobados+1;
end
else
begin
if promedio >= 7 then
begin
promocionados:_promocionados+1;
end
else
begin
regularizados:_regularizados+1;
end;
end;
end;
write("Los reprobados son un total de: " ,reprobados);
write("Los regularizados son un total de: " ,regularizados);
write("Los promocionados son un total de: " ,promocionados);
end
```