

**Procesadores de Lenguajes**

**Analizador Sintáctico**

**Docente**: Cristian Prats

**Integrantes**:

Geronimo Ratcliffe

Javier Villagra

Santiago Russo

Fecha: 11/09/2013

Versión: 1.0

Tabla de Revisiones

|  |  |  |  |
| --- | --- | --- | --- |
| Fecha | Versión | Descripción | Autor |
| 10/10/2013 | 1.0 | Creación del Documento | Santiago Russo |

Contents

[I: INTRODUCCION 4](#_Toc369523898)

[II: Tareas 4](#_Toc369523899)

[III: Definiciones GRAMATICA 5](#_Toc369523900)

[3.1 RESTRICCIONES DEL LENGUAJE: 8](#_Toc369523901)

[3.1 Gramática en el Algoritmo 8](#_Toc369523902)

[3.1.1 Archivo .grammar 9](#_Toc369523903)

[IV: Armado de la Tabla SLR 12](#_Toc369523904)

[4.1 Cálculo primeros y siguientes 14](#_Toc369523905)

[4.2 Autómata Finito del Algoritmo SLR 15](#_Toc369523906)

[4.3 Tabla SLR 15](#_Toc369523907)

[4.3.1 Carga de la tabla SLR 15](#_Toc369523908)

[4.4 Casos de PRUEBA 16](#_Toc369523909)

[4.4.1 Primer Caso 16](#_Toc369523910)

[4.4.2 Segundo Caso 17](#_Toc369523911)

[4.4.3 Tercer Caso 18](#_Toc369523912)

[4.4.4 Cuarto Caso 19](#_Toc369523913)

[V: PROCESO DE IMPLEMENTACION 22](#_Toc369523914)

[MECANISMO DE RECUPERACION DE ERRORES 22](#_Toc369523915)

[CONCLUSIONES 22](#_Toc369523916)

[COMENTARIOS 23](#_Toc369523917)

[AXENO 23](#_Toc369523918)

[Bibliografia 27](#_Toc369523919)

[Indice 27](#_Toc369523920)

# I: Introducción

Una gramática LR es aquella en la que su tabla de chequeo de sintaxis no posee entradas múltiples, o sea, es suficiente con examinar sólo un símbolo a la entrada, para saber qué regla aplicar.

Se ha elegido este método ya que se puede trabajar con gramáticas S-Atribuidas, facilitando luego el trabajo en el analizador semántico.

# II: Definiciones Gramatica

Los no terminales son los Tokens escritos sin <> o minúscula.

#### Algunos Terminales:

Numero /\* CONSUMO UN TOKEN ENTERO\*/

Booleano -> true|false /\*CONSUMO UN TOKEN BOOLEANO \*/

Id -> /\* CONSUMO UN TOKEN DE TIPO ID \*/

Cadena -> ‘algo’ /\* CONSUMO UN TOKEN DE TIPO CADENA \*/

Tipo -> integer |boolean

Operador\_Booleano -> = | <> 🡪Token

Operador\_Booleano\_E -> < | > | <= | >=

Suma\_Resta -> + | - 🡪Token

Mult\_Div -> \* | / 🡪TOKEN

By -> // para el byval o byref

Show -> //aplica para show y showln

#### Principal:

**INICIAL SRL**

S’ -> <S>

INICIAL S -> <GLOBALES><BLOQUE>

GLOBALES -> <DECGL>|

λ

BLOQUE -> <FP><BLOQUE> |

λ

FP -> function Id(<PARAM>): Tipo; <BLOQUE1> end-func <EXP>; |

Procedure Id(<PARAM>); <BLOQUE1> end-proc;

#### Bloques:

BLOQUE1 -> <DECL> begin <BLOQUE2> |

λ

BLOQUE2 -> <LINEA>;<BLOQUE2> |

λ

LINEA -> Id<ASIG> |

if<EXP>then<BLOQUE2><BLOQUESI>end-if|

while<EXP> do <BLOQUE2> end-while|

<FUNC> |

ASIG -> (<PASAJE>)|

:= <EXP> |

[<EXP>] := <EXP>

BLOQUESI -> else <BLOQUE2> |

λ

FUNC -> read Id<IDAUX> |

Show <MOSTRAR>

#### Parametrizaciónes:

PARAM -> <TIPOPARAM> Id : Tipo <PARAM1> |

λ

PARAM1 -> ,<TIPOPARAM> Id : Tipo <PARAM1>|

λ

TIPOPARAM -> By |

λ

#### Declaraciones:

DECGL -> <VARG><DECGL> |

<CONSTS><DECGL> |

λ

DECL -> <VARS><DECL> |

<CONSTS><DECL> |

λ

#### Constantes y variables:

CONSTS -> const Id : <CONST1>;

CONST1 -> Tipo = <EXP> <CONST2>

CONST2 -> , Id : <CONST1> |

λ

VARS -> var Id <VAR1>;

VAR1 -> , Id <VAR1> |

: Tipo

Globales ----------------------------------------

VARG -> var Id <VARG1>;

VARG1 -> ,Id <VARG1> |

: Tipo |

[<Numero>] <VARG1>

#### Funciones embebidas:

MOSTRAR -> Cadena <MOSTRAR1> |

Id <IDAUX> <MOSTRAR1>

MOSTRAR1 -> , <MOSTRAR> | λ

IDAUX -> [EXP] | λ

#### Expresiones:

<EXP> -> <EXP> or <TERMBOOL> |

<TERMBOOL>

TERMBOOL -> <TERMBOOL> and <MINITERMBOOL> |

<MINITERMBOOL>

MINITERMBOOL -> not <FACTBOOL>|

<FACTBOOL>

FACTBOOL -> <EXPBOOL> |

<EXP2>

EXP2 -> <EXP2> Suma\_Resta <TERM>

| <TERM>

TERM -> <TERM> Mult\_Div <FACT>

| <FACT>

FACT -> (EXP)|

Id <ID1>|

Numero |

Booleano

EXPBOOL -> <EXP2> Operador\_Booleano <EXP2> | //tipos cualquieras, A = true o A = 2

<EXP2> Operador\_Booleano\_E <EXP2> //requiere enteros

ID1 -> [EXP] | (PASAJE) | λ

PASAJE -> Id <PASAJE1> |

Numero <PASAJE1> |

Booleano <PASAJE1> |

λ

PASAJE1 -> , <PASAJE> |

λ

## 2.1 Restricciones Del Lenguaje:

* La función “mostrar” solo pude mostrar variables, constantes o vectores.
* La función “leer” solo puede leer un número o booleano por vez invocada.
* En la producción EXPBOOL hay que chequear que ambos lados sean enteros.
* En la declaración de Arreglos solo se le puede pasar un entero, no se pueden pasar constantes.

## 2.1 Gramática en el Algoritmo

La gramática es pasada al Parser mediante un archivo .grammar en el cual se construye la gramática de la siguiente forma:

1. Los no terminales son los tokens escritos en mayúsculas. Ej: ID
2. Los terminales del lado izquierdo están escritos en mayúsculas. Ej: EXP
3. Los terminales del lado derecho de las producciones están rodeados por <>. Ej: <EXP>
4. Los terminales o no terminados deben estar separados entre sí por un ‘ ‘ (1 espacio o más) y toda la producción debe estar en la misma línea.
5. La separación izquierda de la derecha es realizada por “->” y se debe dejar un espacio a la izquierda y derecha del mismo
6. **Es de suma importancia que los terminales tengan el mismo nombre que aquellos usados en la tabla!!**
7. Notar que el símbolo de vacío utilizado es LAMBDA (λ). El proyecto de eclipse fue configurado como UTF-8 para reconocer ese careacter.
8. La gramática pasada al parser debe ser aquella que se corresponda con aquella tabla que se le pase al mismo parser.

El archivo .grammar debe llamarse Gramatica.grammar y debe ubicarse en la carpeta Gramatica del root.

La gramática se carga ordenada en un ArrayList, donde su número de gramática a ser referenciada está representada por el índice.

**Ej de Producción:**

LINEA -> IF <EXP> THEN <BLOQUE2> <BLOQUESI> END\_IF

### 2.1.1 Archivo .grammar

En nuestro compilador, el archivo .grammar es el siguiente:

S' -> <S>

S -> <GLOBALES> <BLOQUE>

GLOBALES -> <DECGL>

GLOBALES -> λ

BLOQUE -> <FP> <BLOQUE>

BLOQUE -> λ

FP -> FUNCTION ID PARENTESIS\_APERTURA <PARAM> PARENTESIS\_CIERRE DEF\_TIPO TIPO ENDLINE <BLOQUE1> END\_FUNC <EXP> ENDLINE

FP -> PROCEDURE ID PARENTESIS\_APERTURA <PARAM> PARENTESIS\_CIERRE ENDLINE <BLOQUE1> END\_PROC ENDLINE

BLOQUE1 -> <DECL> BEGIN <BLOQUE2>

BLOQUE2 -> <LINEA> ENDLINE <BLOQUE2>

BLOQUE2 -> λ

LINEA -> ID <ASIG>

LINEA -> IF <EXP> THEN <BLOQUE2> <BLOQUESI> END\_IF

LINEA -> WHILE <EXP> DO <BLOQUE2> END\_WHILE

LINEA -> <FUNC>

ASIG -> PARENTESIS\_APERTURA <PASAJE> PARENTESIS\_CIERRE

ASIG -> ASIGNACION <EXP>

ASIG -> CORCHETE\_APERTURA <EXP> CORCHETE\_CIERRE ASIGNACION <EXP>

BLOQUESI -> ELSE <BLOQUE2>

BLOQUESI -> λ

FUNC -> READ ID <IDAUX>

FUNC -> SHOW <MOSTRAR>

PARAM -> <TIPOPARAM> ID DEF\_TIPO TIPO <PARAM1>

PARAM -> λ

PARAM1 -> COMMA <TIPOPARAM> ID DEF\_TIPO TIPO <PARAM1>

PARAM1 -> λ

TIPOPARAM -> BY

TIPOPARAM -> λ

DECGL -> <VARG> <DECGL>

DECGL -> <CONSTS> <DECGL>

DECGL -> λ

DECL -> <VARS> <DECL>

DECL -> <CONSTS> <DECL>

DECL -> λ

CONSTS -> CONST ID DEF\_TIPO <CONST1> ENDLINE

CONST1 -> TIPO CONST\_IGUAL <EXP> <CONST2>

CONST2 -> COMMA ID DEF\_TIPO <CONST1>

CONST2 -> λ

VARS -> VAR ID <VAR1> ENDLINE

VAR1 -> COMMA ID <VAR1>

VAR1 -> DEF\_TIPO TIPO

VARG -> VAR ID <VARG1> ENDLINE

VARG1 -> COMMA ID <VARG1>

VARG1 -> DEF\_TIPO TIPO

VARG1 -> CORCHETE\_APERTURA NUMERO CORCHETE\_CIERRE <VARG1>

MOSTRAR -> CADENA <MOSTRAR1>

MOSTRAR -> ID <IDAUX> <MOSTRAR1>

MOSTRAR1 -> COMMA <MOSTRAR>

MOSTRAR1 -> λ

IDAUX -> CORCHETE\_APERTURA <EXP> CORCHETE\_CIERRE

IDAUX -> λ

EXP -> <EXP> OR <TERMBOOL>

EXP -> <TERMBOOL>

TERMBOOL -> <TERMBOOL> AND <MINITERMBOOL>

TERMBOOL -> <MINITERMBOOL>

MINITERMBOOL -> NOT < MINITERMBOOL>

MINITERMBOOL -> <FACTBOOL>

FACT -> BOOLEANO

FACTBOOL -> <EXPBOOL>

FACTBOOL -> <EXP2>

EXP2 -> <EXP2> SUMA\_RESTA <TERM>

EXP2 -> <TERM>

TERM -> <TERM> MULT\_DIV <FACT>

TERM -> <FACT>

FACT -> PARENTESIS\_APERTURA <EXP> PARENTESIS\_CIERRE

FACT -> ID <ID1>

FACT -> NUMERO

EXPBOOL -> <EXP2> OPERADOR\_BOOLEANO <EXP2>

EXPBOOL -> <EXP2> OPERADOR\_BOOLEANO\_E <EXP2>

ID1 -> CORCHETE\_APERTURA <EXP> CORCHETE\_CIERRE

ID1 -> PARENTESIS\_APERTURA <PASAJE> PARENTESIS\_CIERRE

ID1 -> λ

PASAJE -> ID <PASAJE1>

PASAJE -> NUMERO <PASAJE1>

PASAJE -> BOOLEANO <PASAJE1>

PASAJE -> λ

PASAJE1 -> COMMA <PASAJE>

PASAJE1 -> λ

BLOQUE1 -> λ

# III: Armado de la Tabla SLR

Para realizar las pruebas de la gramática y para el armado de la tabla utilizamos el programa “JFLAP”, el cuál permite dada una gramática LR armar su tabla SLR(1).

Debido a que el programa no permite generar la gramática con palabras enteras, fue necesario traducir nuestra gramática a solamente símbolos, a continuación se presenta la tabla de conversiones.

En el caso de los no terminales se utilizaron Tokens para simplificar la visualización y ya que la gramática acepta de por si Tokens y no lexemas.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Nomeclatura** | | | | | |
| **Terminales** | **Simbolo** | **Token Type** |  | **No Terminales** | **Simbolo** |
| ( | ( | PARENTESIS\_APERTURA |  | ASIG | A |
| ) | ) | PARENTESIS\_CIERRE |  | BLOQUE | B |
| , | , | COMMA |  | BLOQUE1 | Ò |
| : | : | DEF\_TIPO |  | BLOQUE2 | Ô |
| := | = | ASIGNACION |  | BLOQUESI | Ö |
| ; | ; | ENDLINE |  | CONSTS | C |
| [ | [ | CORCHETE\_APERTURA |  | CONST1 | Ç |
| ] | ] | CORCHETE\_CIERRE |  | CONST2 | È |
| = | 0 | CONST\_IGUAL |  | DECGL | É |
| And | a | AND |  | DECL | D |
| begin | { | BEGIN |  | EXP | E |
| Booleano | b | BOOLEANO |  | EXP2 | K |
| By | & | BY |  | EXPBOOL | O |
| Cadena | l | CADENA |  | FACT | Ñ |
| const | c | CONST |  | FACTBOOL | J |
| do | d | DO |  | FP | F |
| else | e | ELSE |  | FUNC | Ê |
| end-func | # | END\_FUNC |  | GLOBALES | G |
| end-if | q | END\_IF |  | ID1 | Â |
| end-proc | } | END\_PROC |  | IDAUX | I |
| end-while | u | END\_WHILE |  | LINEA | L |
|  |  | EOF |  | MINITERMBOOL | Z |
| function | f | FUNCTION |  | MOSTRAR | M |
| Id | i | ID |  | MOSTRAR1 | N |
| if | s | IF |  | PARAM | P |
| Mult\_Div | \* | MULT\_DIV |  | PARAM1 | Q |
| not | z | NOT |  | PASAJE | Á |
| Numero | n | NUMERO |  | PASAJE1 | Ú |
| Operador\_Booleano | ? | OPERADOR\_BOOLEANO |  | S | S |
| Operador\_Booleano\_E | > | OPERADOR\_BOOLEANO\_E |  | TERM | H |
| Or | o | OR |  | TERMBOOL | T |
| procedure | p | PROCEDURE |  | TIPOPARAM | R |
| read | r | READ |  | VARS | V |
| show | m | SHOW |  | VAR1 | W |
| Suma\_Resta | + | SUMA\_RESTA |  | VARG | X |
| then | y | THEN |  | VARG1 | Y |
| Tipo | t | TIPO |  |  |  |
| var | v | VAR |  |  |  |
| while | w | WHILE |  |  |  |
| λ | λ |  |  |  |  |

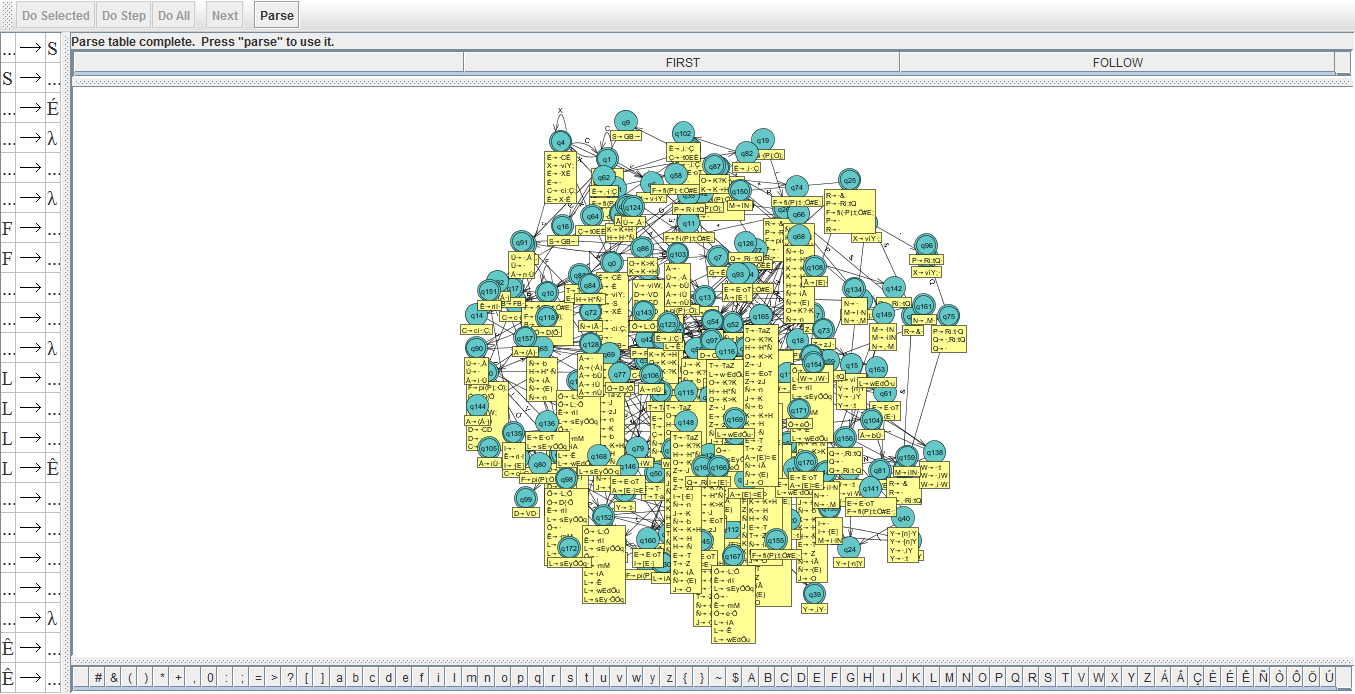
## 3.1 Cálculo primeros y siguientes

Los cálculos de los primeros y siguientes se muestran con la traducción de símbolos al igual que el autómata finito utilizado para el armado de la tabla.

|  |  |  |
| --- | --- | --- |
| **No Terminal** | **Primeros** | **Siguientes** |
| A | ([= | ; |
| B | f!p | $ |
| C | c | fvc$p{ |
| D | !vc | { |
| E | bn(zi | d];o),y |
| F | fp | f$p |
| G | !vc | f$p |
| H | bn(i | da\*;]o+)?>y, |
| I | ![ | ;, |
| J | bn(i | da;]o)y, |
| K | bn(i | da];+o)?>,y |
| L | wsrmi | ; |
| M | li | ; |
| N | !, | ; |
| O | bn(i | da];o),y |
| P | !&i | ) |
| Q | !, | ) |
| R | !& | i |
| S | f!vcp | $ |
| T | bn(zi | da;]o)y, |
| V | v | vc{ |
| W | :, | ; |
| X | v | fvc$p |
| Y | :[, | ; |
| Z | bn(zi | da];o),y |
| Á | !bni | ) |
| Â | !([ | da\*;]o+)?>y, |
| Ç | t | ; |
| È | !, | ; |
| É | !vc | f$p |
| Ê | rm | ; |
| Ñ | bn(i | da];\*+o)?>,y |
| Ò | !vc{ | #} |
| Ô | w!srmi | #ueq} |
| Ö | !e | q |
| Ú | !, | ) |

## 3.2 Autómata Finito del Algoritmo SLR

A continuación se presenta la máquina de estados que fue utilizada para la construcción de la tabla.



## 3.3 Tabla SLR

Como la tabla SLR es muy grande como para mostrarse en este documento, se agrega una referencia a la ubicación del Excel.



### 3.3.1 Carga de la tabla SLR

La carga de la tabla se realiza mediante un archivo Excel depositado en la carpeta Gramatica. El archivo debe llamarse TablaSLR y debe ser de extensión .xls, no se admiten archivos de extensión .xlsx. Además, este archivo Excel debe contener una hoja llamada “Tabla”, “Table”, “Tabla SLR” o “SLR Table” y toda la tabla, IR\_A y Acción en esa misma hoja. En este caso, el símbolo EOF debe ser el último no terminal. Opcionalmente, se puede cargar la tabla con un Excel con 2 hojas llamas “Accion” o “Action” e “Ir A” o “Goto” y cada hoja debe tener la parte de la tabla acción o ir A respectivamente; ambas tablas deben tener en su primer fila el número de estado.

La carga de la gramática se realizó en un ArrayList para utilizar el calor del índice como el estado (Fila de la misma) y cada ArrayList contiene un HashMap el cual contiene como claves los terminales o no terminales; esto se realizó para agilizar las búsquedas. En caso de que se encuentre con un Token de ERROR, se devuelve la acción ERROR sin siquiera buscar en la Tabla.

Para hacer un diseño más amigable, cada Celda de la tabla es una TableAction, la cual contiene la acción a realizar y en caso de ser Reducción, Desplazamiento o un GOTO, el número de Estado o de producción por el cual reducir.

## 3.4 Casos de Prueba

A continuación se listan algunos casos para probar la gramática. Se muestra primero el código completo y a continuación la transformación que se tuvo que hacer para poder utilizar el programa. La transformación de palabras a símbolos se realizó utilizando la tabla anteriormente mencionada. No se aceptan los espacios en blanco para delimitar cada instrucción.

### 3.4.1 Primer Caso

**Prueba simple Caso de aceptación**

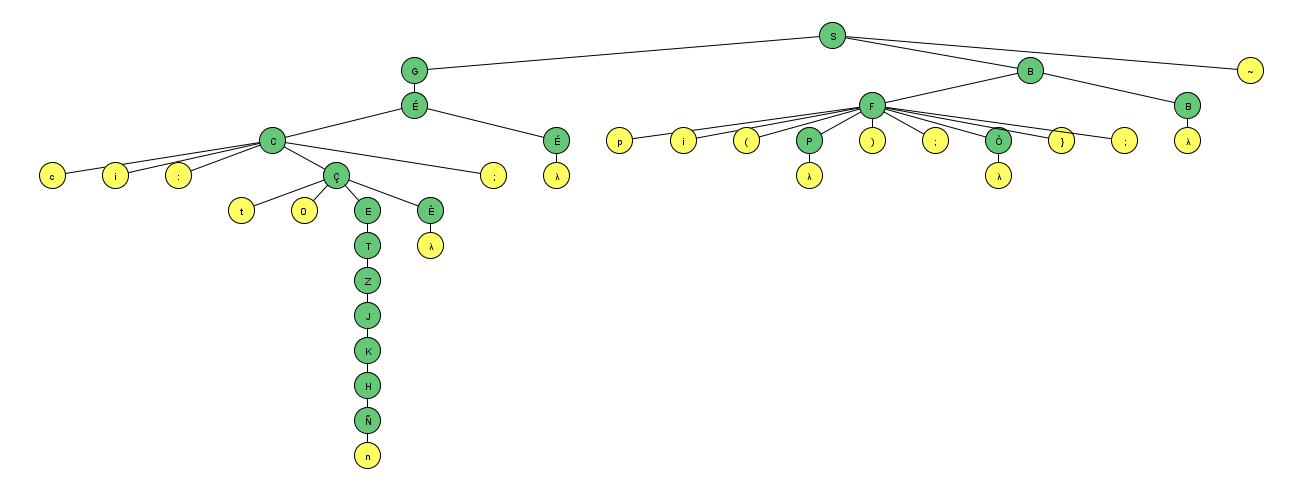
* Código:

Const A : integer = 9;

Procedure B();

End-proc;

* Transformación: ci:t0n;pi();};
* Árbol de derivación: Los círculos verdes representan los nodos no terminales, mientras que los amarillos los terminales.



### 3.4.2 Segundo Caso

**Prueba media Caso de aceptación**

* Código:

Const A : integer = 9 , B:Boolean = true;

Var y : integer;

Procedure B(byval R:integer);

Var Z : integer;

Begin

Z := A+B;

End-proc;

Function C(A:integer, byval G:boolean):integer;

begin

If G>10 then

Y := 4;

Else

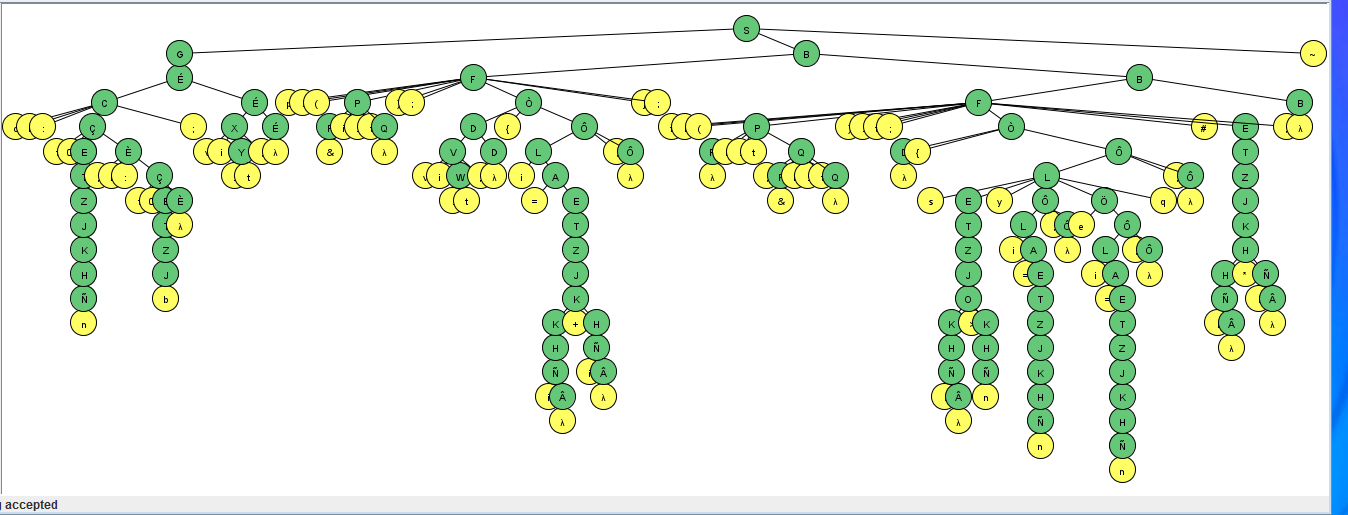
Y := 0;

End-if;

End-func C\*G;

* Transformación: ci:t0n,i:t0b;vi:t;pi(&i:t);vi:t;{i=i+i;};fi(i:t,&i:t):t;{si>nyi=n;ei=n;q;#i\*i;

Árbol de derivación: Los círculos verdes representan los nodos no terminales, mientras que los amarillos los terminales.



### 3.4.3 Tercer Caso

**Prueba simple Caso de rechazo, ídem anterior sin el end-if**

* Código:

Const A : integer = 9 , B:Boolean = true;

Var y : integer;

Procedure B(byval R:integer);

Var Z : integer;

Begin

Z := A+B;

End-proc;

Function C(A:integer, byval G:boolean):integer;

begin

If G>10 then

Y := 4;

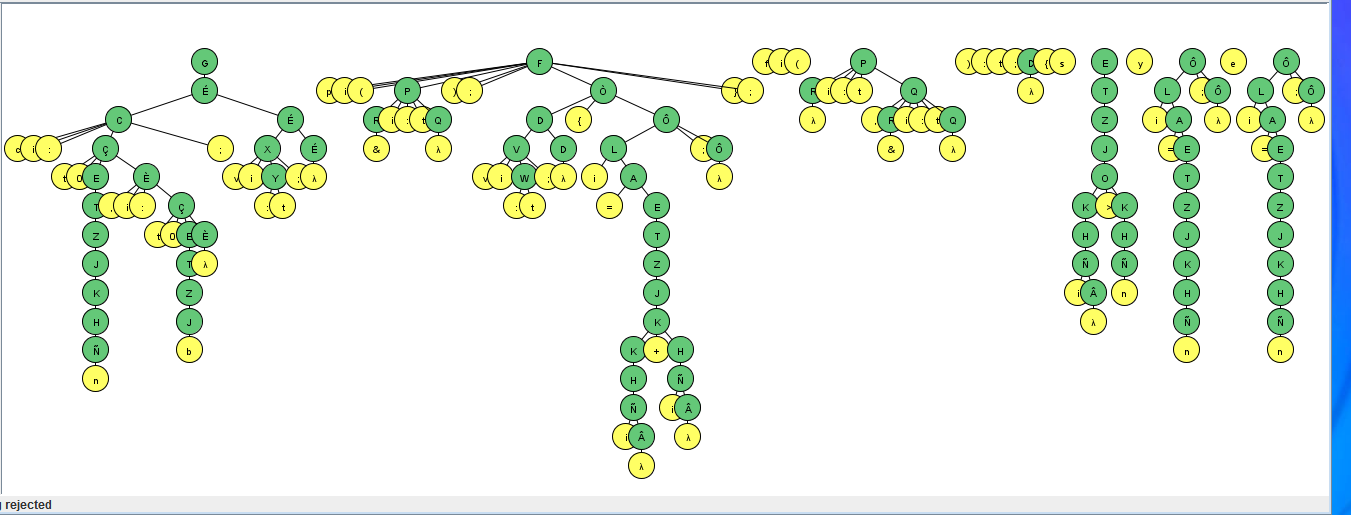
Else

Y := 0;

End-func C\*G;

* Transformación: ci:t0n,i:t0b;vi:t;pi(&i:t);vi:t;{i=i+i;};fi(i:t,&i:t):t;{si>nyi=n;ei=n;#i\*i;

Árbol de derivación:



### 3.4.4 Cuarto Caso

**Prueba compleja: Caso de aceptacion**

* Código:

const M : integer = 7, R : integer = 90;

var N, S : integer;

var A[12],D : integer;

var C[5] : boolean;

procedure PROC1();

var A : integer;

var B : boolean;

begin

B := not C[3] and (N < A or S <> R);

C[1] := true;

while B do

if A <= 10 then

A := A + 1;

end-if;

showLN 'Visualizacion', B, ' ', A;

B := C[3] and A + N < R;

end-while;

end-proc;

procedure PROC2(byref R : integer);

const T : integer = 67;

var W11, W12, Q : integer;

begin

C[2] := false;

S := (S + M) \* 2;

Q := 1;

read W11; {lectura de teclado}

W12 := AW11 \* 2 + S;

while (W12 - 2) <= R + S do

W12 := W12 \* 2;

end-while;

while W12 + M > Y / 2 do

Q := Q \* 2;

W12 := W12 / 2;

if W12 <= R then

R := R - W12;

Q := Q + 1;

end-if;

end-while;

A[1] := Q;

end-proc;

function FUN1(T: integer, byval N2 : integer) : boolean;

var N: integer;

begin

if T > 0 then

N := -45;

else

N := 70;

end-if;

end-func N \* 2 >= 0 or not N2 = 0;

procedure MAIN();

var X: boolean;

begin

PROC1();

S := M + 1;

PROC2(S);

X := FUN1(5, 8);

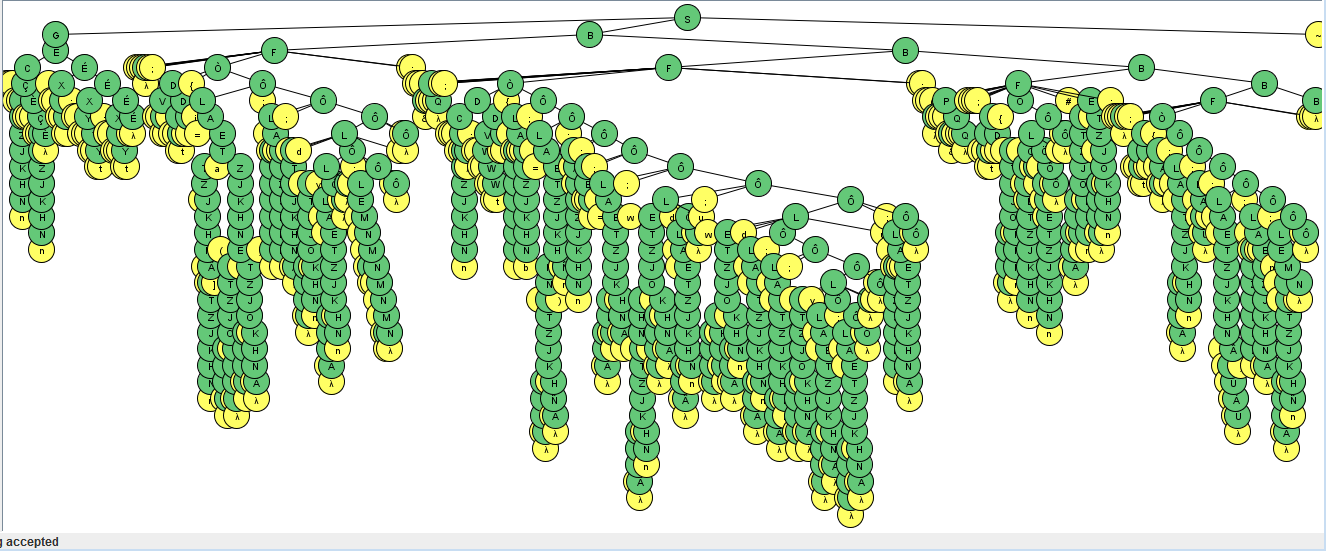
C[5] := X;

show A[S + 1];

end-proc;

* Transformación: ci:t0n,i:t0n;vi,i:t;vi[n],i:t;vi[n]:t;pi();vi:t;vi:t;{i=zi[n]a(i>ioi?i);i[n]=b;widsi>nyi=i+n;q;ml,i,l,i;u;};pi(&i:t);ci:t0n;vi,i,i:t;{i[n]=b;i=(i+i)\*n;i=n;ri;i=i\*n+i;w(i+n)>i+idi=i\*n;u;wi+i>i\*ndi=i\*n;i=i\*n;si>iyi=i+i;i=i+i;q;u;i[n]=i;};fi(i:t,&i:t):t;vi:t;{si>nyi=n;ei=n;q;#i\*n>nozi?n;pi();vi:t;{i();i=i+n;i(i);i=i(n,n);i[n]=i;mi[i+n];};

Árbol de derivación:



# IV: Proceso De Implementación

Al igual que con la implementación del módulo Analizador Léxico, generamos un nuevo módulo denominado Analizador Sintáctico. Dicho módulo se encuentra compuesto de tres paquetes fundamentales, los mismos son: sintáctico, estructuras y helper. En sintáctico se encuentra el Analizador Sintáctico propiamente dicho, una interfaz para la creación del Árbol y la implementación del Árbol que se va armando; en estructuras se encuentran las clases necesarias con sus respectivas interfaces para manejar el Algoritmo de Desplazamiento Reducción, algunas de estas son la Pila, la Tabla, la Gramática con sus Producciones, entre otras. Por último, el paquete helper tiene las clases para leer un Archivo de texto o un Excel.

La clase de la Tabla como se explicó anteriormente carga un Excel de nombre TablaSLR que contenga las hojas mencionadas, la tabla es cargada en dos estructuras bidimensionales ArrayList de HashMaps que contienen Acciones a realizar, las acciones se descomponen a partir de leer las celdas como Strings. Las dos estructuras son una para manejar la parte de la Tabla Ir A y la otra para manejar la parte de la Tabla Acción. La clase de la Pila provee los mecanismos para manejar la misma según los requerimientos del algoritmo Reducir o Sintetizar y la Gramática guarda la gramática LR utilizada para su uso en el Algoritmo.

Las clases del helper ExcelReader y StringReader leen de un Excel o un archivo de texto respectivamente.

La clase más importante es Sintactic Analyzer la cual implementa el algoritmo de Desplazamiento Reducción haciendo uso de la Pila, la Tabla, la Gramática y el Analizador Léxico para ir obteniendo los Tokens. Se implementó el Algoritmo tal cual se encuentra en el libro del Dragón.

## 4.1 Mecanismo De Recuperación De Errores

En este punto es cuando las versiones 1 y 2 del Analizador Sintáctico difieren:

Los errores se manejan en tratarErrorLexico y tratarErrorSintactico respectivamente para los errores léxico y sinteacticos.

### 4.1.1 Versión 1

En esta versión, el mecanismo de recuperación de errores evita el modo de pánico de modo estricto, de esta forma, cuando hay un error Léxico, para el análisis Sintáctico informando solo los errores léxicos hasta el final del programa. Si en cambio se encuentra un error Sintáctico, lo informa y luego informa todos los errores léxicos hasta el final del programa.

## 4.1.2 Versión 2

Como aplicar el método descripto en el Dragon Book era muy engorroso, tedioso y largo, se optó por diseñar un mecanismo de recuperación de errores que intente saltear de a bloques erróneos. Es decir que si se encuentra un error en el If intentara saltear todo el bloque If. Si bien su funcionamiento no es del todo correcto, muestra varios errores léxicos reduciendo los errores ocasionados por el modo de pánico a solo bloques en lugar de todo el código.

Se implementó este método de la siguiente manera a grandes rasgos:

* + Se lee de la entrada hasta que se detecta un ; o un EOF
  + Se saca de la pila hasta que se encuentra un No terminal o un token de ENDLINE (;). Esto lo que hace es intentar sacar de la pila la línea que no se redujo que es errónea. La pila contiene un método que se denominó error mode para el manejo de errores.

Este método si bien no funciona correctamente para todos los errores sintácticos, permite analizar todo el código.

Ante el primer error Léxico, el programa corta el análisis de errores sintácticos mostrando todos los errores Lexicos.

# V: Pruebas

Las pruebas realizadas se encuentras anexas en la carpeta Test.

# VI: Cambios Realizados al Léxico

Se ha tenido que cambiar el análisis léxico para que detectara más tipos de Tokens y no solo los que se le habían colocado en el TP inicial. Además, se corrigió el problema de que el análisis léxico no detectaba A:= (-10).

# VII: Conclusiones

El problema principal con el que nos encontramos en esta entrega fue la correcta definición de la gramática y el armado de la tabla. Por suerte, la herramienta JFLAP nos ha ayudado a revisar y probar la gramática para poder encontrar los posibles errores.

Sin embargo, el problema más difícil de este tipo de compiladores LR es su manejo de errores, el cual no hemos podido solucionar correctamente; sin embargo con un poco de esfuerzo hemos podido encontrar un método casi “adecuado” para el anejo de errores.

# Bibliografia

* COMPILADORES: Principios, Técnicas y Herramientas

*Autores: Alfred V. Aho – RaviSethi*

* JFLAP: <http://www.jflap.org/>

[*Susan H. Rodger*](http://www.cs.duke.edu/~rodger) *Professor of the Practice  
Department of Computer Science, Box 90129  
LSRC, D237  
Duke University, Durham, NC 27708-0129  
Phone: (919)-660-6595, 419-7098  
Email: rodgerAT(@) cs.duke.edu*