Práctica PL

Elena Kaloyanova Popova y Álvaro Borja Velasco García

2018

Índice general

Capítulo 1

Fase 1: Analizador léxico

1.1. Clases Léxicas

Todo programa consta de dos secciones: una para las declaraciones y otra para las instrucciones, separadas por un token «&&». La sección de declaraciones está formada por una serie de declaraciones compuestas por el nombre de tipo y el de variable y separadas por un punto y coma. La sección de instrucciones, por su parte, consta de una serie de asignaciones (variable=expresión), separadas también por un punto y coma. Las clases léxicas que hemos considerado para representar los tokens del lenguaje son las siguientes:

- **SEC:** Representa el seccionador de las dos partes del programa («&&»).
- **NUM:** Palabra reservada «num».
- **BOOL:** Palabra reservada «bool».
- **VAR:** Representa el nombre de la variable. Comienza necesariamente por una letra, seguida por una secuencia de cero o más letras, dígitos o el símbolo «_».
- **ASIG:** Representa el signo igual de las asignaciones.
- **NXT:** Representa el signo «;» que marca el comienzo de la siguiente instrucción.
- **TRUE:** Palabra reservada «true».
- **FALSE:** Palabra reservada «false».

- **NUMR:** Representa un número real. Puede empezar opcionalmente con un signo seguido de una secuencia de uno o más digitos cualesquiera, pudiendo poner ceros no significativos a la izquierda. Puede opcionalmente estar seguido por una parte decimal y/o una parte exponencial.
- **MAS:** Operador suma (\+).
- **MENOS:** Operador resta (\-).
- **POR:** Operador multiplicación (*).
- **DIV:** Operador división (/).
- **AND:** Palabra reservada «and».
- **OR:** Palabra reservada «or».
- **NOT:** Palabra reservada «not».
- **MAY:** Operador mayor (>).
- **MEN:** Operador menor (<).
- **MAYI:** Operador mayor o igual (>=).
- **MENI:** Operador menor o igual (<=).
- **IGUAL:** Operador igual a (==).
- **DIST:** Operador distinto a (!=).
- **PAP:** Signo de apertura de paréntesis.
- **PCI:** Signo de cierre de paréntesis.
- **EOF:** Representa el final de fichero.

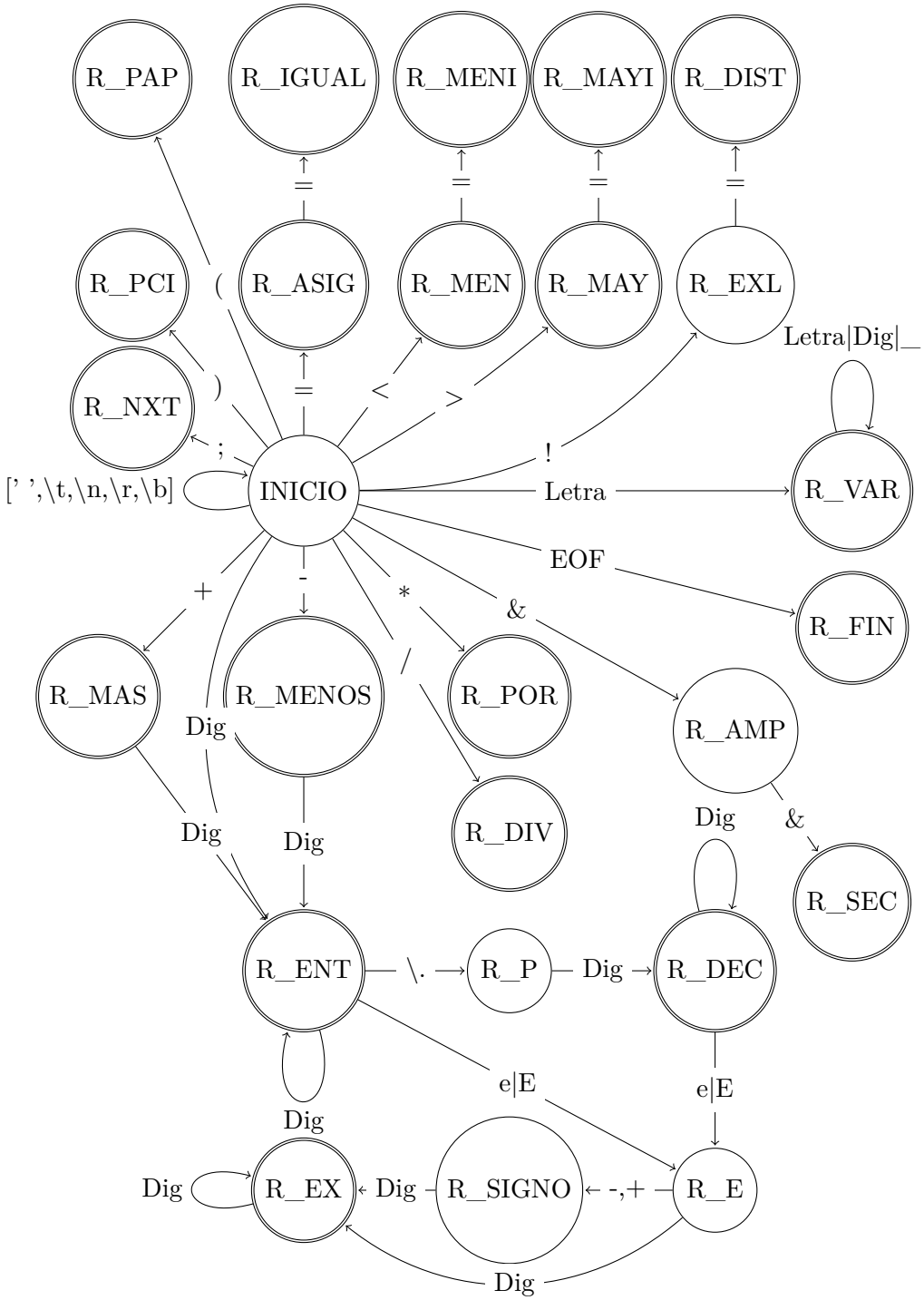
1.2. Especificación Formal

Las definiciones regulares correspondientes a las clases léxicas definidas son:

(⋆) **SEC:** [&][&]
(⋆) **VAR:** LETRA([LETRA|DIG|_]*)
LETRA: ([a-z,A-Z])

DIG: ([0-9])
(⋆) **NUM:** ([n][u][m])
(⋆) **BOOL:** ([b][o][o][l])
(⋆) **TRUE:** ([t][r][u][e])
(⋆) **FALSE:** ([f][a][l][s][e])
(⋆) **NUMR:** SIGNO?(DIG+(DEC)?(EX)?)
DEC: (\.)DIG+
EX: [e|E](SIGNO?DIG+)
SIGNO: [\+,\-]
DIG: [0-9]
(⋆) **AND:** ([a][n][d])
(⋆) **OR:** ([o][r])
(⋆) **NOT:** ([n][o][t])
(⋆) **MAS:** (\+)
(⋆) **MENOS:** (\-)
(⋆) **DIV:** (/)
(⋆) **POR:** (*)
(⋆) **MAY:** (>)
(⋆) **MEN:** (<)
(⋆) **MAYI:** ([>][=])
(⋆) **MENI:** ([<][=])
(⋆) **IGUAL:** ([=][=])
(⋆) **DIST:** ([!][=])
(⋆) **ASIG:** (=)
(⋆) **NXT:** (;)
(⋆) **PAP:** (\()
(⋆) **PCIERRE:** (\))
(⋆) **SEP:** [« »,\t,\n,\r,\b]

1.3. Diseño


R_PAP
R_IGUAL
R_MENI
R_MAYI
R_DIST
=
=
=
=
R_PCI
(
R_ASIG
R_MEN
R_MAY
R_EXL
Letra|Dig|_
R_NXT
)
=
<
>
;
!
[' ',\t,\n,\r,\b]
INICIO
Letra
R_VAR
EOF
R_FIN
+
-
*
&
R_MAS
R_MENOS
/
R_POR
R_AMP
Dig
Dig
Dig
R_DIV
Dig
&
R_SEC
R_ENT
\.
R_P
Dig
R_DEC
e|E
e|E
Dig
Dig
R_EX
Dig
R_SIGNO
-,+
R_E
Dig

Capítulo 2

Fase 2: Analizador sintáctico

En esta fase desarrollaremos el analizador sintáctico descendente predictivo para el lenguaje descrito en la primera fase.

2.1. Gramática incontextual

2.1.1. Operadores

Empezaremos definiendo la gramática incontextual que define el lenguaje. Los operadores que utiliza nuestro lenguaje aparecen en la tabla 2.1.

Operador	Prioridad	Tipo	Asociatividad
+,-	0	Binario infijo	Asocia Izquierda
and or	1	Binarios infijos	Asocia Derecha No asocia
Relacionales	2	Binario infijo	No asocia
*,/	3	Binario infijo	Asocia Izquierda
- not	4	Unarios prefijos	Asocia No asocia

Cuadro 2.1: Operadores

2.1.2. Gramática incontextual

La gramática incotextual obtenida apartir de la definición y los operadores es la siguiente:

S → Programa EOF

Programa → LDs SEC LIs

LDs → LDs NXT Declaracion
LDs → Declaracion
Declaracion → NUM VAR
Declaracion → BOOL VAR
LIs → LIs NXT Instruccion
LIs → Instruccion
Instruccion → VAR ASIG EXP0

EXP0 → EXP0 OP0 EXP1
EXP0 → EXP1
EXP1 → EXP2 AND EXP1
EXP1 → EXP2 OR EXP2
EXP1 → EXP2
EXP2 → EXP3 OP2 EXP3
EXP2 → EXP3
EXP3 → EXP3 OP3 EXP4
EXP3 → EXP4
EXP4 → MENOS EXP4
EXP4 → NOT EXP5
EXP4 → EXP5
EXP5 → NUMR
EXP5 → VAR
EXP5 → TRUE
EXP5 → FALSE
EXP5 → PAP EXP0 PCIERRE
OP0 → MAS
OP0 → MENOS
OP2 → MAY
OP2 → MEN
OP2 → MAYI
OP2 → MENI
OP2 → IGUAL
OP2 → DIST
OP3 → POR
OP3 → DIV

2.1.3. Gramática transformada LL(1)

Necesitamos transformar la gramática a una LL(1). Una vez transformada, la gramática queda de la siguiente manera:

S → Programa EOF
Programa → LDs SEC LIs
LDs → Declaracion RLDS
RLDS → NXT Declaracion RLDS
RLDS → ε
Declaracion → NUM VAR
Declaracion → BOOL VAR
LIs → Instruccion RLIS
RLIS → NXT Instruccion RLIS
RLIS → ε
Instruccion → VAR ASIG EXP0

EXP0 → EXP1 R0
R0 → OP0 EXP1 R0
R0 → ε
EXP1 → EXP2 R1
R1 → AND EXP1
R1 → OR EXP2
R1 → ε
EXP2 → EXP3 R2
R2 → OP2 EXP3 R2
R2 → ε
EX3 → EXP4 R3
R3 → OP3 EXP4 R3
R3 → ε
EXP4 → MENOS EXP4
EXP4 → NOT EXP5
EXP4 → EXP5
EXP5 → NUMR
EXP5 → VAR
EXP5 → TRUE
EXP5 → FALSE
EXP5 → PAP EXP0 PCIERRE
OP0 → MAS
OP0 → MENOS
OP2 → MAY
OP2 → MEN
OP2 → MAYI
OP2 → MENI
OP2 → IGUAL

OP2 → DIST
OP3 → POR
OP3 → DIV

2.1.4. Primeros y siguientes

Los **primeros** de nuestra gramática son:
PRIM(S) = {NUM, BOOL}
PRIM(PROGRAMA) = {NUM, BOOL}
PRIM(LDS) = {NUM, BOOL}
PRIM(RLDS) = {NXT}
PRIM(DECLARACION) = {NUM, BOOL}
PRIM(LIS) = {VAR}
PRIM(RLIS) = {NXT}
PRIM(INSTRUCCION) = {VAR}
PRIM(R0) = {MAS,MENOS}
PRIM(R1) = {AND,OR}
PRIM(EXP1) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
PRIM(EXP2) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
PRIM(R2) = {MAY, MEN, MAYI, MENI, IGUAL, DIST}
PRIM(EXP3) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
PRIM(R3) = {POR, DIV, MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
PRIM(EXP4) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
PRIM(EXP5) = {NUMR, VAR, TRUE, FALSE, PAP}
PRIM(EXP0) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
PRIM(OP0) = {MAS, MENOS}
PRIM(OP2) = {MAY, MEN, MAYI, MENI, IGUAL, DIST}
PRIM(OP3) = {POR, DIV}
Los **siguientes** son:
SIG(S) = {ε}
SIG(PROGRAMA) = {EOF}
SIG(LDS) = {SEC}
SIG(RLDS) = {SEC}
SIG(DECLARACION) = {NXT, SEC}
SIG(LIS) = {EOF}
SIG(RLIS) = {EOF}
SIG(INSTRUCCION) = {NXT,EOF}
SIG(R0) = {PCI, NXT, EOF}
SIG(R1) = {PCI, MAS, MENOS, NXT, EOF}

SIG(EXP1) = {PCI, MAS, MENOS, NXT, EOF}
SIG(EXP2) = {PCI, AND, OR, MAS, MENOS, NXT, EOF}
SIG(R2) = {PCI, AND, OR, MAS, MENOS, NXT, EOF}
SIG(EXP3) = {PCI, MAY, MEN, MAYI, MENI, IGUAL, DIST, AND, OR, MAS, MENOS, NXT, EOF}
SIG(R3) = {PCI, MAY, MEN, MAYI, MENI, IGUAL, DIST, AND, OR, MAS, MENOS, NXT, EOF}
SIG(EXP4) = {PCI, POR, DIV, NOT, NUMR, VAR, TRUE, FALSE, PAP, MAY, MEN, MAYI, MENI, IGUAL, DIST, AND, OR, MAS, MENOS, NXT, EOF}
SIG(EXP5) = {PCI, POR, DIV, NOT, NUMR, VAR, TRUE, FALSE, PAP, MAY, MEN, MAYI, MENI, IGUAL, DIST, AND, OR, MAS, MENOS, NXT, EOF}
SIG(EXP0) = {PCI, NXT, EOF}
SIG(OP0) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
SIG(OP2) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
SIG(OP3) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}

2.1.5. Directores

Obtenidos los primeros y los siguientes podemos proceder a calcular los **directores**:

DIR(S → Programa EOF) = {NUM, BOOL}
DIR(Programa → LDs SEC LIs EOF) = {NUM, BOOL}
DIR(LDs → Declaracion RLDS) = {NUM, BOOL}
DIR(RLDS → NXT Declaracion RLDS) = {NXT}
DIR(RLDS → ε) = {SEC}
DIR(Declaracion → NUM VAR) = {NUM}
DIR(Declaracion → BOOL VAR) = {BOOL}
DIR(LIs → Instruccion RLIS) = {VAR}
DIR(RLIS → NXT Instruccion RLDS) = {NXT}
DIR(RLIS → ε) = {EOF}
DIR(Instruccion → VAR ASIG EXP0) = {VAR}
DIR(EXP0 → EXP1 R0) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
DIR(R0 → OP0 EXP1 R0) = {MAS, MENOS}
DIR(R0 → ε) = {PCI, NXT, EOF}
DIR(EXP1 → EXP2 R1) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
DIR(R1 → AND EXP1 R1) = {AND}

DIR(R1 → OR EXP2) = {OR}
DIR(R1 → ε) = {PCI, MAS, MENOS, NXT, EOF}
DIR(EXP2 → EXP3 R2) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
DIR(R2 → OP2 EXP3 R2) = {MAY, MEN, MAYI, MENI, IGUAL, DIST}
DIR(R2 → ε) = {PCI, AND, OR, MAS, MENOS, NXT, EOF}
DIR(EX3 → EXP4 R3) = {MENOS, NOT, NUMR, VAR, TRUE, FALSE, PAP}
DIR(R3 → OP3 EXP4 R3) = {MUL,DIV}
DIR(R3 → ε) = {PCI, MAY, MEN, MAYI, MENI, IGUAL, DIST, AND, OR, MAS, MENOS, NXT, EOF}
DIR(EXP4 → MENOS EXP4) = {MENOS}
DIR(EXP4 → NOT EXP5) = {NOT}
DIR(EXP4 → EXP5) = {NUMR, VAR, TRUE, FALSE, PAP}
DIR(EXP5 → NUMR) = {NUMR}
DIR(EXP5 → VAR) = {VAR}
DIR(EXP5 → TRUE) = {TRUE}
DIR(EXP5 → FALSE) = {FALSE}
DIR(EXP5 → PAP EXP0 PCIERRE) = {PAP}
DIR(OP0 → MAS) = {MAS}
DIR(OP0 → MENOS) = {MENOS}
DIR(OP2 → MAY) = {MAY}
DIR(OP2 → MEN) = {MEN}
DIR(OP2 → MAYI) = {MAYI}
DIR(OP2 → MENI) = {MENI}
DIR(OP2 → IGUAL) = {IGUAL}
DIR(OP2 → DIST) = {DIST}
DIR(OP3 → POR) = {POR}
DIR(OP3 → DIV) = {DIV}

Capítulo 3

Fase 3: Analizador sintáctico ascendente

En esta fase desarrollaremos una versión diferente del analizador sintáctico, esta vez ascendente LR. Se implementará con JLex y Cup haciendo uso de la gramática incontextual desarrollada en la fase 2.

Capítulo 4

Fase 4: Construcción de árboles de sintaxis abstracta

En esta fase desarrollaremos los constructores ascendentes y descendentes del árbol de sintaxis abstracta de la práctica.

4.1. Funciones constructoras

Lo primero que debemos hacer es simplificar la gramática incontextual desarrollada en la fase 2, eliminando las estructuras introducidas con el propósito de evitar ambigüedades. Esta es la gramática resultante de la simplificación:

Programa → LDs SEC LIs

LDs → LDs NXT NUM VAR | LDs NXT BOOL VAR | NUM VAR | BOOL VAR

LIs → LIs NXT VAR ASIG EXP | VAR ASIG EXP

EXP → EXP MAS EXP | EXP MENOS EXP | EXP AND EXP | EXP OR EXP | EXP MAY EXP | EXP MEN EXP | EXP MAYI EXP | EXP MENI EXP | EXP IGUAL EXP | EXP DIST EXP | EXP POR EXP | EXP DIV EXP | MENOS EXP | NOT EXP | NUMR | VAR | TRUE | FALSE| PAP EXP PCIERRE

Una vez tenemos esta nueva gramática podemos obtener los constructores. Cada producción semánticamente significativa representa una función constructora. En la tabla 4.1 podemos ver la lista de constructores obtenidos.

Regla	Constructora
Programa → LDs SEC LIs	**programa**: LDs X LIs → Programa
LDs → LDs NXT NUM VAR	**dnCompuesta**: LDs X String → LDs
LDs → LDs NXT BOOL VAR	**dbCompuesta**: LDs X String → LDs
LDs → NUM VAR	**dnSimple**: String → LDs
LDs → BOOL VAR	**dbSimple**: String → LDs
LIs → LIs NXT VAR ASIG EXP	**liCompuesta**: LIs X String X EXP → LIs
LIs → VAR ASIG EXP	**liSimple**: String X EXP → LIs
EXP → EXP MAS EXP	**suma**: EXP X EXP → EXP
EXP → EXP MENOS EXP	**resta**: EXP X EXP → EXP
EXP → EXP AND EXP	**conj**: EXP X EXP → EXP
EXP → EXP OR EXP	**disy**: EXP X EXP → EXP
EXP → EXP MAY EXP	**mayor**: EXP X EXP → EXP
EXP → EXP MEN EXP	**menor**: EXP X EXP → EXP
EXP → EXP MAYI EXP	**mayori**: EXP X EXP → EXP
EXP → EXP MENI EXP	**menori**: EXP X EXP → EXP
EXP → EXP IGUAL EXP	**igual**: EXP X EXP → EXP
EXP → EXP DIST EXP	**distinto**: EXP X EXP → EXP
EXP → EXP POR EXP	**mul**: EXP X EXP → EXP
EXP → EXP DIV EXP	**div**: EXP X EXP → EXP
EXP → MENOS EXP	**signo**: EXP → EXP
EXP → NOT EXP	**neg**: EXP → EXP
EXP → NUMR	**real**: String → EXP
EXP → VAR	**id**: String → EXP

Cuadro 4.1: Constructoras

4.2. Diagrama de clases

A continuación se muestra el diagrama de clases que nos ayudará a diseñar la sintaxis abstracta del lenguaje.

4.3. Constructor de árboles de sintaxis abstracta

S → Programa EOF
    S.a = Programa.a
Programa → LDs SEC LIs
    Programa.a = programa(LDs.a, LIs.a)
LDs → LDs NXT Declaracion
    LDs_0.a = dCompuesta(LDs.a, Declaracion.a)
LDs → Declaracion
    LDs.a = Declaracion.a
Declaracion → NUM VAR
    Declaracion.a = dnSimple(VAR.lex)
Declaracion → BOOL VAR
    Declaracion.a = dbSimple(VAR.lex)
LIs → LIs NXT Instruccion
    LIs_0.a = liCompuesta(LDs.a, Instruccion.id, Instruccion.exp)
LIs → Instruccion
    LIs.a = liSimple(Instruccion.id, Instruccion.exp)
Instruccion → VAR ASIG EXP0
    Intruccion.id = VAR.lex
    Instruccion.exp = EXP0.a

EXP0 → EXP0 OP0 EXP1
    EXP0_0.a = mkexpbin(OP0.op, EXP0.a, EXP1.a)
EXP0 → EXP1
    EXP0.a = EXP1.a
EXP1 → EXP2 AND EXP1
    EXP1_0.a = mkexpbin(«and», EXP2.a, EXP1.a)
EXP1 → EXP2 OR EXP2
    EXP1.a = mkexpbin(«or», EXP2_0.a, EXP2.a)
EXP1 → EXP2
    EXP1.a = EXP2.a
EXP2 → EXP3 OP2 EXP3
    EXP2.a = mkexpbin(OP2.op, EXP3_0.a, EXP3.a)

EXP2 → EXP3
    EXP2.a = EXP3.a
EXP3 → EXP3 OP3 EXP4
    EXP3_0.a = mkexpbin(OP3.op, EXP3.a, EXP4.a)
EXP3 → EXP4
    EXP3.a = EXP4.a
EXP4 → MENOS EXP4
    EXP4_0.a = mkexpun(«-», EXP4.a)
EXP4 → NOT EXP5
    EXP4.a = mkexpun(«!», EXP5.a)
EXP4 → EXP5
    EXP4.a = EXP5.a
EXP5 → NUMR
    EXP5.a = real(NUMR.lex)
EXP5 → VAR
    EXP5.a = id(VAR.lex)
EXP5 → TRUE
    EXP5.a = «true»
EXP5 → FALSE
    EXP5.a = «false»
EXP5 → PAP EXP0 PCIERRE
    EXP5.a = EXP0.a
OP0 → MAS
    OP0.op = «+»
OP0 → MENOS
    OP0.op = «-»
OP2 → MAY
    OP2.op = «>»
OP2 → MEN
    OP2.op = «<»
OP2 → MAYI
    OP2.op = «>=»
OP2 → MENI
    OP2.op = «<=»
OP2 → IGUAL
    OP2.op = «==»
OP2 → DIST
    OP2.op = «!=»
OP3 → POR
    OP3.op = «/»

OP3 → DIV
    OP3.op = «/»

Para poder hacer una implementación descendente es necesario acondicionar la gramática:

S → Programa EOF
    S.a = Programa.a
Programa → LDs SEC LIs
    Programa.a = programa(LDs.a, LIs.a)
LDs → Declaracion RLDS
    RLDS.ah = dSimple(Declaracion.a)
    LDs.a = RLDS.a
RLDS → NXT Declaracion RLDS
    RLDS_1.a = dCompuesta(RLDS_0.ah, Declaracion.a)
    RLDS_0.a = RLDS_1.a
RLDS → ε
    RLDS.a = RLDS.ah
Declaracion → NUM VAR
Declaracion → BOOL VAR
LIs → Instruccion RLIS
    RLIS.ah = liSimple(Instruccion.id, Instruccion.exp)
    LIs.a = RLIS.a
RLIS → NXT Instruccion RLIS
    RLIS_1.a = liCompuesta(RLIS_0.ah, Instruccion.a)
    RLIS_0.a = RLIS_1.a
RLIS → ε
    RLIs.a = RLIs.ah
Declaracion → NUM VAR
    Declaracion.a = dnSimple(VAR.lex)
Declaracion → BOOL VAR
    Declaracion.a = dbSimple(VAR.lex)
Instruccion → VAR ASIG EXP0
    Intruccion.id = VAR.lex
    Instruccion.exp = EXP0.a

EXP0 → EXP1 R0
    R0.ah = EXP1.a
    EXP0.a = R0.a
R0 → OP0 EXP1 R0
    R0_1.ah = mkexpbin(OP0.op, R0_0.ah, EXP1.a)
    R0_0.a = R0_1.a

R0 → ε
    R0.a = R0.ah
EXP1 → EXP2 R1
    R1.ah = EXP2.a
    EXP1.a = R1.a
R1 → AND EXP1
    R1_1.ah = mkexpbin(."nd", R1_1.ah, EXP1.a)
    R1_0.a = R1_1.a
R1 → OR EXP2
    R1_1.ah = mkexpbin(."r", R1_0.ah, EXP2.a)
    R1_0.a = R1_1.a
R1 → ε
    R1.a = R1.ah
EXP2 → EXP3 R2
    R2.ah = EXP3.a
    EXP2.a = R2.a
R2 → OP2 EXP3 R2
    R2_1.ah = mkexpbin(OP2.op, R2_0.ah, EXP3.a)
    R2_0.a = R2_1.a
R2 → ε
    R2.a = R2.ah
EX3 → EXP4 R3
    R3.ah = EXP4.a
    EXP3.a = R3.a
R3 → OP3 EXP4 R3
    R3_1.ah = mkexpbin(OP3.op, R3_0.ah, EXP4.a)
    R3_0.a = R3_1.a
R3 → ε
    R3.a = R3.ah
EXP4 → MENOS EXP4
    EXP4_0.a = mkexpun(", EXP4.a)
EXP4 → NOT EXP5
    EXP4.a = mkexpun(¡", EXP5.a)
EXP4 → EXP5
    EXP4.a = EXP5.a
EXP5 → NUMR
    EXP5.a = real(NUMR.lex)
EXP5 → VAR
    EXP5.a = id(VAR.lex)
EXP5 → TRUE

EXP5.a = "true"
EXP5 → FALSE
EXP5.a = "false"
EXP5 → PAP EXP0 PCIERRE
EXP5.a = EXP0.a
OP0 → MAS
OP0.op = «+»
OP0 → MENOS
OP0.op = «-»
OP2 → MAY
OP2.op = «>»
OP2 → MEN
OP2.op = «<»
OP2 → MAYI
OP2.op = «>=»
OP2 → MENI
OP2.op = «<=»
OP2 → IGUAL
OP2.op = «==»
OP2 → DIST
OP2.op = «!=»
OP3 → POR
OP3.op = «/»
OP3 → DIV
OP3.op = «/»