# Memoria práctica 3:

Procesadores de Lenguajes:

Integrantes: Suhuai Chen, Daniel Pizarro, David Starry y Jiahui You

# Índice:



## 1. Diseño de la Sintaxis Abstracta:

| **No terminal** | **Género** |
|---|---|
| programa | Prog |
| bloque | Bloq |
| declaraciones_opt | DecsOp |
| declaraciones | Decs |
| declaracion | Dec |
| instrucciones_opt | InstrsOp |
| instrucciones | Instrs |
| instruccion | Instr |
| T | Tipo |
| E | Exp |
| campos | Campos |
| campo | Campo |
| parametrosFormales_opt | ParsFOp |
| parametrosFormales | ParsF |
| parametroFormal | ParF |
| parametrosReales_opt | ParsReOp |
| parametrosReales | ParsRe |

| **Regla** | **Constructor** |
|---|---|
| programa → bloque | prog: Bloq → Prog |
| bloque → declaraciones_opt instrucciones_opt | bloq: DecsOp x InstrsOp → Bloq |
| declaraciones_opt → declaraciones | siDecs: Decs → DecsOp |
| declaraciones_opt → ε | noDecs: → DecsOp |
| instrucciones_opt → instrucciones | siInstrs: Instrs → InstrsOp |
| instrucciones_opt → ε | noInstrs: → InstrsOp |
| T → T literalEntero | tipoLista: Tipo x string → Tipo |
| T → ^T | tipoCircum: Tipo → Tipo |
| T → campos | tipoStruct: Campos → Tipo |
| T → identificador | tipoIden: string → Tipo |
| T → int | tipoInt: → Tipo |
| T → real | tipoReal: → Tipo |
| T → bool | tipoBool: → Tipo |
| T → string | tipoString: → Tipo |
| campos → campos campo | muchosCampos: → Campos x Campo → Campos |
| campos → campo | unCampo: Campo → Campos |
| campo → T identificador | creaCampo: Tipo x string → Campo |
| declaraciones → declaraciones declaración | muchasDecs: Decs x Dec → Decs |
| declaraciones → declaración | unaDec: Dec → Decs |
| declaracion → T identificador | decVariable: Tipo x string → Dec |
| declaracion → type T identificador | decTipo: Tipo x string → Dec |
| declaracion → proc identificador parametrosFormales_opt bloque | decProc: string x ParsFOp x Bloq → Dec |

| | |
|---|---|
| parametrosFormales_opt → parametrosFormales | siParsF: ParsF → ParsFOp |
| parametrosFormales_opt → ε | noParsF: → ParsFOp |
| parametrosFormales → parametrosFormales parametroFormal | muchosParsF: ParsF x ParF→ ParsF |
| parametroFormales → parametroFormal | unParF: ParF → ParsF |
| parametroFormal → T & identificador | paramF: string x Tipo → ParF |
| parametroFormal → T identificador | param: string x Tipo → ParF |
| instrucciones → instrucciones instrucción | muchasInstrs: Instrs x Instr → Instrs |
| instrucciones → instruccion | unaInstr: Instr → Instrs |
| instruccion → E | instrEval: Exp → Instr |
| instruccion → E bloque | instrIf: Exp x Bloq → Instr |
| instruccion → E bloque bloque | instrIfElse: Exp x Bloq x Bloq → Instr |
| instruccion → E bloque | instrWhile: Exp x Bloq →Instr |
| instruccion → E | instrRead: Exp → Instr |
| instruccion → E | intrWrite: Exp → Instr |
| instruccion → nl | instrNl: -> Instr |
| instruccion → E | instrNew: Exp → Instr |
| instruccion → E | instrDel: Exp → Instr |
| instruccion → call identificador parametrosReales_opt | instrCall: string x ParsReOp → Instr |
| instruccion → bloque | instrBloque: Bloq → Instr |
| parametrosReales_opt → parametrosReales | siParsRe: ParsRe → ParsReOp |
| parametrosReales_opt → ε | noParsRe: → ParsReOp |
| parametrosReales → parametrosReales E | muchosParsRe: ParsRe x Exp → ParsRe |
| parametrosReales → E | unParRe: Exp → ParsRe |
| E → E = E | asig: Exp x Exp → Exp |

| | |
|---|---|
| E → E <= E | menorI: Exp x Exp → Exp |
| E → E < E | menor: Exp x Exp → Exp |
| E → E >= E | mayorI: Exp x Exp → Exp |
| E → E > E | mayor: Exp x Exp → Exp |
| E → E == E | igual: Exp x Exp → Exp |
| E → E != E | distint: Exp x Exp → Exp |
| E → E + E | suma: Exp x Exp → Exp |
| E → E - E | resta: Exp x Exp → Exp |
| E → E * E | multiplicacion: Exp x Exp → Exp |
| E → E / E | division: Exp x Exp → Exp |
| E → E % E | modulo: Exp x Exp → Exp |
| E → E and E | and: Exp x Exp → Exp |
| E → E or E | or: Exp x Exp → Exp |
| E → not E | negacion: Exp → Exp |
| E → - E | menosUnario: Exp → Exp |
| E → E E | indexacion: Exp x Exp → Exp |
| E → E identificador | acceso: Exp x string → E |
| E → E ^ | indireccion: Exp → Exp |
| E → identificador | identificador: string → Exp |
| E → literalEntero | literalEntero: string → Exp |
| E → literalReal | literalReal: string → Exp |
| E → true | true: → Exp |
| E→ false | false: → Exp |
| E → literalCadena | literalCadena: string → Exp |
| E → null | null: → Exp |

- Hay que eliminar todos los terminales que no tienen carga semántica.
  Por ejemplo; símbolos de puntuación (“.”, “,”, “;”)
- Se fusionan todos los no terminales que sean equivalentes entre sí.
  A y B son equivalentes cuando A→* B y B→* A. Por ejemplo eliminando E0 → E1.
- Aplicando transformaciones que preservan la equivalencia.
  Por ejemplo, eliminar OP1, que se factoriza para >,>=,<,<=...

En negro está la gramática anterior. En rojo la gramática simplificada.

programa → bloque
bloque → { declaraciones_opt instrucciones_opt }
programa → declaraciones_opt instrucciones_opt

declaraciones_opt → declaraciones &&
declaraciones_opt → declaraciones
declaraciones_opt → ε
declaraciones_opt → ε
instrucciones_opt → instrucciones
instrucciones_opt → instrucciones
instrucciones_opt→ ε
instrucciones_opt → ε

- Tipos
tipo1 → tipo1[literalEntero]
tipo1 → tipo2
T → T literalEntero
tipo2 → ^tipo2
tipo2 → tipo3
T → ^T

tipo3 → struct { campos }
T → campos
tipo3 → identificador
T → identificador
tipo3 → tipoBasico
tipoBasico → int
tipoBasico → real
tipoBasico → bool
tipoBasico → string
T → int
T → real
T → bool
T → string

campos → campos, campo
campos →campos campo
campos → campo
campos → campo
campo → tipo1 identificador
campo → T identificador

- Declaraciones

declaraciones → declaraciones ; declaración
declaraciones → declaraciones declaración
declaraciones → declaración
declaraciones → declaración

variable → tipo1 identificador
declaracion → variable
declaracion → T identificador
declaracion → type variable
declaracion → type T identificador
declaracion → header bloque
header -> proc identificador ( parametrosFormales_opt )
declaracion → proc identificador parametrosFormales_opt bloque

parametrosFormales_opt → parametrosFormales
parametrosFormales_opt → parametrosFormales

parametrosFormales_opt → ε
parametrosFormales_opt → ε

parametrosFormales → parametrosFormales, parametroFormal
parametrosFormales → parametrosFormales parametroFormal
parametrosFormales → parametroFormal
parametroFormales → parametroFormal

parametroFormal → tipo1 referencia_opt identificador
referencia_opt → &
referencia_opt → ε
parametroFormal → T & identificador
parametroFormal → T identificador

- Instrucciones

instrucciones → instrucciones ; instrucción
instrucciones → instrucciones instrucción
instrucciones → instruccion
instrucciones → instruccion

instruccion → @ Eo
instruccion → E
instruccion → if Eo bloque
instruccion → E bloque
instruccion → if Eo bloque else bloque
instruccion → E bloque bloque

instruccion → while Eo bloque
instruccion → E bloque
instruccion → read Eo
instruccion → E
instruccion → write Eo
instruccion → E
instruccion → nl
instruccion → nl
instruccion → new Eo
instruccion → E
instruccion → delete Eo
instruccion → E
instruccion → call identificador ( parametrosReales_opt )
instruccion → identificador parametrosReales_opt
instruccion → bloque
instruccion → bloque

parametrosReales_opt → parametrosReales
parametrosReales_opt → parametrosReales
parametrosReales_opt → ε
parametrosReales_opt → ε

parametrosReales → parametrosReales, Eo
parametrosReales → parametrosReales E
parametroReales→ Eo
parametrosReales → E

E0 → E1 = E0
E0 → E1
E → E = E

E1 → E1 OP1 E2
E1 → E2
OP1 → <
OP1 → >
OP1 → <=
OP1 → >=
OP1 → ==
OP1 → !=

E → E < E
E → E > E
E → E <= E
E → E >= E
E → E == E
E → E != E

E2 → E2 + E3 //izquierdas
E2 → E3 - E3
E2 → E3

E → E + E
E → E - E

E3 → E4 and E3
E3 → E4 or E4
E3 → E4
E → E and E
E → E or E

E4 → E4 OP4 E5
E4 → E5
OP4 → *
OP4 → /
OP4 → %
E → E * E
E → E / E
E → E % E

E5 → OP5 E5
E5 → E6

OP5 → not
OP5 → -
E → not E
E → - E

E6 → E6[ Eo ] // E es una expresión, a.c[x + 1]
E → E E

E6 → E6. identificador // . un nombre de campo (un identificador)
E → E identificador

E6 → E6^ // l^.sig
E → E

E6 → E7
E → E

E → identificador
E → identificador
E → literalEntero
E → literalEntero
E → literalReal
E → literalReal
E → true
E → true
E→ false
E→ false
E → literalCadena
E → literalCadena
E → null
E → null

## 2. Especificación del constructor de ASTs mediante una gramática s-atribuida

En color morado está el acondicionamiento para implementación descendente.

programa → bloque
programa.a = prog(bloque.a)

bloque → { declaraciones_opt instrucciones_opt }
bloque.a = bloq(declaraciones_opt.a,instrucciones_opt.a)

declaraciones_opt → declaraciones &&
declaraciones_opt.a = siDecs(declaraciones.a)
declaraciones_opt → ε
declaraciones_opt.a = noDecs()

instrucciones_opt → instrucciones
instrucciones_opt.a = siInstrs(instrucciones.a)
instrucciones_opt→ ε
instrucciones_opt.a = noInstrs()
- Tipos
tipo1 → tipo1[literalEntero]
tipo1.t = tipoLista(tipo1.t, literalEntero.lex)
tipo1 → tipo2
tipo1.t = tipo2.t
tipo1 → tipo2 rtipo1
rtipo1.th = tipo2.t
tipo1 .v = rtipo1.v
rtipo1 → [literalEntero] rtipo1
$rtipo1_1$.th = tipoLista($rtipo1_0$.th, literalEntero)
$rtipo1_0$.t = $rtipo1_1$.t
rtipo1 → ε
rtipo1.t = rtipo1.th

tipo2 → ^tipo2
tipo2.t = tipoCircum(tipo2.t)
tipo2 → tipo3
tipo2.t = tipo3.t

tipo3 → struct { campos }
tipo3.t = tipoStruct(campos.a)
tipo3 → identificador
tipo3.t = tipoIden(identificador.lex)
tipo3 → int

tipo3 .t = tipoInt()
tipo3 → real
tipo3 .t = tipoReal()
tipo3 → bool
tipo3 .t = tipoBool()
tipo3 → string
tipo3 .t =tipoString()

campos →campos, campo
Campos0.a = muchosCampos(campos1.a,campo.a)
campos → campo
campos.a = unCampo(campo.a)

campos → campo Rcampos
Rcampos.ah = unCampo(campo.a)
campos.a = Rcampos.a
$Rcampos_0$ → campo $Rcampos_1$
$Rcampos_1$.ah = muchosCampos (campo.a, $Rcampos_0$.ah)
$Rcampos_0$.a = $Rcampos_1$.a
Rcampos → ε
Rcampos.a = Rcampos.ah

campo → tipo1 identificador
campo.a = creaCampo(tipo1.a, identificador.a)

- Declaraciones
declaraciones → declaraciones ; declaracion
declaraciones → declaracion
declaraciones0.a= muchasDecs(declaraciones1.a,declaracion.a)
declaraciones.a= unaDec(declaracion.a)
declaraciones → declaración DR
DR.ah = unaDec(declaración.a)
declaraciones.a = DR.a
DR –> declaración DR
$DR_1$.ah = muchasDecs($DR_0$.ah, declaración.a )
$DR_0$.a = $DR_1$.a
DR → ε
DR .a = DR .ah

declaracion → tipo1 identificador
declaracion.a = decVariable( tipo1.t, identificador.lex)
declaracion → type tipo1 identificador
declaracion.a = decTipo(tipo1.t, identificador.lex)
declaracion → proc identificador ( parametrosFormales_opt ) bloque
declaracion.a = decProc( identificador.lex,parametrosFormales_opt.a bloque.a)

parametrosFormales_opt → parametrosFormales
parametrosFormales_opt.a= siParsF(parametrosFormales.a)
parametrosFormales_opt → ε
parametrosFormales_opt.a= noParsF()

parametrosFormales → parametrosFormales, parametroFormal
parametrosFormales.a= muchosParsF(parametrosFormales.a, parameterFormal.a)
parametrosFormales → parametroFormal
parametrosFormales.a= unParF(parametroFormal.a)
parametroFormal → tipo1 & identificador
parameterFormal.a=paramFf(tipo1.a, identificador.lex)
parametroFormal → tipo1 identificador
parameterFormal.a=param(tipo1.a, identificador.lex)
parametrosFormales → parametroFormal RparametrosFormales
    RparametrosFormales.ah = unParF(parametroFormal.a)
    $parametrosFormales_0$.a = RparametrosFormales.a
RparametrosFormales → , parametroFormal RparametrosFormales
    $RparametrosFormales_1$.ah =
    muchosParsF($RparametrosFormales_0$.ah, parametroFormal.a)
    $RparametrosFormales_0$.a = $RparametrosFormales_1$.a
RparametrosFormales → ε
    RparametrosFormales.a = RparametrosFormales.ah

- Instrucciones
instrucciones → instrucciones ; instrucción
instrucciones → instruccion
instrucciones0.a= muchasInstrs(instrucciones1.a,instrucción.a)
instrucciones.a=unaInstr(instrucción.a)
instrucciones → instrucción IR
    IR.ah = unaInstr(instrucción .a)
    instrucciones.a = IR.a
IR –> instruccion IR
    $IR_1$.ah = muchasInstrs($IR_0$.ah, instruccion.a )
    $IR_0$.a = $IR_1$.a
IR → ε
    IR .a = IR .ah

instruccion → @ E0
instruccion.a=instrEval(E0.a)
instruccion → if E0 bloque
instruccion.a=instrIf(E0.a,bloque.a)
instruccion → if E0 bloque else bloque
instruccion.a=instrIfElse(E0.a,bloque$_1$.a, bloque$_2$.a)
instruccion → E0 bloque IFR
IFR.ah1 = E0.a
IFR.ah2 = bloque.a
instruccion.a = IFR.a
IFR → else bloque
IFR.a = instrIfE(IFR.ah1,IFR.ah2, bloque.a)
IFR → ε
IFR.a = instrIf(IFR.ah1.a,IFR.ah2.a)

instruccion → while E0 bloque
instruccion.a=instrWhile(E0.a ,bloque.a)
instruccion → read E0
instruccion.a=instrRead(E0.a)
instruccion → write E0
instruccion.a=instrWrite(E0.a)
instruccion → nl
instruccion.a=instrNl()
instruccion → new E0
instruccion.a=instrNew(E0.a)
instruccion → delete E0
instruccion.a=instrDel( E0.a)
instruccion → call identificador ( parametrosReales_opt )
instruccion.a=instrCall(identificador.lex, parametrosReales_opt.a)
instruccion → bloque
instruccion.a=instrBloque(bloque.a)

parametrosReales_opt → parametrosReales
parametrosReales_opt .a= siParsRe(parametrosReales.a)
parametrosReales_opt → ε
parametrosReales_opt.a= noParsRe()

parametrosReales → parametrosReales, E0
parametrosReales.a= muchosParsRe(parametrosReales.a, E0.a)
parametroReales → E0
parametroReales .a= unParRe(E0.a)
parametrosReales → E0 RparametrosReales
RparametrosReales.ah = unParRe(E0.a)
parametrosReales.a = RparametrosReales.a
RparametrosReales → E0 RparametrosReales
$RparametrosReales_1$.ah =
muchosParsRe($RparametrosReales_0$.ah, E0.a)
$RparametrosReales_0$.a = RparametrosReales $_1$.a
RparametrosReales → ε
RparametrosReales.a = RparametrosReales.ah

E0 → E1 = E0
E0.a = asig(E1.a, E0.a)
E0 → E1
E0.a = E1.a
**E0 → E1 FE0**
FE0.ah = E1.a
E0.a = FE0.a
**FE0 → = E0**
FE0.a= mkop("=" FE0.ah, E0.a)
**FE0 → ε**
FE0.a = FE0.ah

E1 → E1 OP1 E2 //izquierdas
E1.a = mkop(OP1.op, E1.a, E2.a)
E1 → E2
E1.a = E2.a
**E1 → E2 RE1**
RE1.ah = E2.a
E1.a = RE1.a
**RE1 → OP1 E2 RE1**
$RE1_1$.ah = mkop(OP1.op,$RE1_0$.ah ,E2.a)
$RE1_0$.a = RE11.a
**RE1 → ε**
RE1.a = RE1.ah

OP1 → <
OP1.op = "<"
OP1 → >
OP1.op = ">"
OP1 → <=
OP1.op = "<="
OP1 → >=
OP1.op = ">="
OP1 → ==
OP1.op = "=="
OP1 → !=
OP1.op = "!="

E2 → E2 + E3 //izquierdas
E2.a = mkop("+",E2.a,E3.a)
E2 → E3 - E3
E2.a = mkop("-",E3.a,E3.a)
E2 → E3
E2.a = E3.a
**E2 → E3 FE3 RE2**
FE3.ah = E3.a
RE2.ah = FE3.a
E2.a = RE2.a
**RE2 → + E3 RE2**
$RE2_1$.ah = mkop("+", $RE2_0$.ah, E3.a)
$RE2_0$.a = $RE2_1$.a
**RE2 → ε**
RE2.a = RE2.ah
**FE3 → - E3**
FE3.a = mkop("-", FE3.ah,E3.a)
**FE3 → ε**
FE3.a = FE3.ah

E3 → E4 and E3
E3.a = mkop("and",E4.a,E3.a)
E3 → E4 or E4
E3.a = mkop("or",E4.a,E4.a)
E3 → E4
E3.a = E4.a
**E3 → E4 FE4**
FE4.ah = E4.a
E3.a = FE4.a

**FE4 → and E3**
FE4.a = mkop("and", FE4.ah,E3.a) // ?
**FE4 → or E4**
FE4.a = mkop("or", FE4.ah,E4.a)
**FE4 → ε**
FE4.a = FE4.ah

E4 → E4 OP4 E5
E4.a = mkop(OP4.op,E4.a,E5.a)
E4 → E5
E4.a =E5.a
**E4 → E5 RE4**
RE4.ah = E5.a
E4.a = RE4.a
**RE4 → OP4 E5 RE4**
$RE4_1$.ah = mkop(OP4.op,$RE4_0$.ah ,E5.a)
$RE4_0$.a = $RE4_1$.a
**RE4 → ε**
RE4.a = RE4.ah

OP4 → *
OP4.op = "*"
OP4 → /
OP4.op = "/"
OP4 → %
OP4.op = "%"

E5 → OP5 E5
E5.a = mkopUn(OP5.op, E5.a)
E5 → E6
E5.a =E6.a

OP5 → not
OP5.op = not
OP5 → -
OP5.op = "-"

E6 → E6[ E0 ] // E es una expresión, a.c[x + 1]
E6.a = indexacion (E6, E0)
E6 → E6. identificador // . un nombre de campo (un identificador)
E6.a = acceso(E6, identificador)
E6 → E6^ // l^.sig
E6.a = indireccion (E6)
E6 → E7
E6.a =E7.a
**E6 → E7 RFE6**

RFE6.ah = E7.a
E6.a = RFE6.a
**RFE6$_0$ → FE6 RFE6$_1$**
FE6.ah = RFE6$_0$.ah
RFE6$_1$.ah = FE6.a
RFE6$_0$.a = RFE6$_1$.a

**RFE6 → ε**
RFE6.a = RFE6.ah
**FE6 → [E0]**
FE6.a = indexacion(FE6.ah, E0.a)
**FE6 → .identificador**
FE6.a = acceso(FE6.ah, identificador)
**FE6 → ^**
FE6.a = indireccion(FE6.ah)

E7 → identificador
E7.a = iden(iden.lex)
E7 → literalEntero
E7.a = literalEntero (literalEntero .lex)
E7 → literalReal
E7.a = literalReal(literalReal.lex)
E7 → true
E7.a = true()
E7→ false
E7.a = false()
E7 → literalCadena
E7.a = literalCadena (literalCadena .lex)
E7 → ( E0 )
E7.a = E0.a
E7 → null
E7.a = null()

**fun** mkop(op,opnd1,opnd2):

op = "<" → **return menor**(opnd1,opnd2)

op = ">" → **return mayor**(opnd1,opnd2)

op = "<=" → **return menorl**(opnd1,opnd2)

op = ">=" → **return mayorl**(opnd1,opnd2)

op = "==" → **return igual**(opnd1,opnd2)

op = "!=" → **return distint**(opnd1,opnd2)

op = "+" → **return suma**(opnd1,opnd2)

op = "-" → **return resta**(opnd1,opnd2)

op = "*" → **return multiplicacion**(opnd1,opnd2)

op = "/" → **return division**(opnd1,opnd2)

op = "%" → **return modulo**(opnd1,opnd2)

op = “and” → **return and** (opnd1, opnd2)

op = “or” → **return or** (opnd1, opnd2)

**fun** mkopUn(op,opnd):

op = "not" → **return negacion**(opnd1)

op = "-" → **return menosUnario**(opnd1)

# 3. Especificación de un procesamiento que imprima los tokens del programa leído

programa → bloque
imprime(**prog(Bloq)**):
    imprime(Bloq)

bloque → declaraciones_opt instrucciones_opt
imprime(**bloq**(DecsOp, InstrsOp)):
    print "{"
    nl
    imprime(DecsOp)
    imprime(InstrsOp)
    nl
    print "}"

declaraciones_opt → declaraciones
imprime(**siDecs**(Decs)):
    imprime(Decs) // , Dec?
    print "&&"
    nl
declaraciones_opt → ε
imprime(**noDecs**(Decs)):
    noop // vacio, NO OPeration

instrucciones_opt → instrucciones
imprime(**siInstrs**(Instrs)):
    imprime(Instrs)

instrucciones_opt→ ε
imprime(**noInstrs**(Instrs)):
    noop // vacio, NO OPeration

instrucciones → instrucciones instrucción
imprime(muchasInstrs( Instrs , Instr):
    imprime(Instrs)
    print(";")
    imprime(Instr)

instrucciones → instruccion
imprime(unaInstr(Instr)):
    imprime(Instr)

instruccion → E
```
imprime(instrEval(Exp)):
        print("@"++ " ")
        imprime(Exp)
```

instruccion → E bloque
```
imprime(instrIf(Exp, Bloq)):
        print("if(")
        imprime(Exp)
        print("){")
        nl
        imprime(Bloq)
        print("}")
```

instruccion → E bloque bloque
```
imprime(instrIfElse(Exp, Bloq)):
        print("if(")
        imprime(Exp)
        print("){")
        nl
        imprime(Bloq)
        print("}")
        nl
        print("else{")
        nl
        imprime(Bloq)
        print("}")
```

instruccion → E bloque
```
imprime(instrWhile(Exp, Bloq)):
        print("while(")
        imprime(Exp)
        print("){")
        nl
        imprime(Bloq)
        print("}")
```

instruccion → E
```
imprime(instrRead(Exp)):
        print("read"++ " ")
        imprime(Exp)
```

instruccion → E

```
imprime(instrWrite(Exp)):
        print("write"++ " ")
        imprime(Exp)

instruccion → nl
imprime(instrNl()):
        print("nl")

instruccion →  E
imprime(instrNew(Exp)):
        print("new"++ " ")
        imprime(Exp)

instruccion →  E
imprime(instrDel(Exp)):
        print("delete"++ " ")
        imprime(Exp)

instruccion → identificador  parametrosReales_opt
imprime(instrCall(Id, ParsReOp)):
        print("call"++ " ")
        print(Id)
        print("(")
        imprime(ParsReOp)
        print(")")

parametrosReales_opt → parametrosReales
imprime(siParsRe(ParsRe)):
        imprime(ParRe)

parametrosReales_opt → ε
imprime(noParsRe()):
        noop // vacio, NO OPeration

parametrosReales → parametrosReales E
imprime(muchosParsRe(ParsRe, Exp)):
        imprime(ParsRe)
        print ","
        imprime(Exp)
parametrosReales → E
imprime(unParRe(Exp)):
        imprime(Exp)
```

instruccion → bloque
imprime(instrBloque(Bloq)):
    imprime(Bloq)

declaraciones → declaraciones ;  declaración
imprime(**muchasDecs**(Decs, Dec)):
    imprime(Decs) // , Dec?
    print ";"
    imprime(Dec)
declaraciones → declaración
imprime(**unaDec**(Dec)):
    imprime(Dec)

declaracion →  T identificador
imprime(**decVariable**(Tipo, Id)):
    imprime(Tipo)
    imprime(Id)
declaracion → type T identificador
imprime(**decTipo**(Tipo, Id)):
    print "type"
    imprime(Tipo)
    imprime(Id)

declaracion → proc identificador  parametrosFormales_opt bloque
imprime(**decProc**(Id,ParsFOp,Bloq)):
    print "proc"
    imprime(Id)
    print "("
    imprime(ParsFOp)
    print ")"
    imprime(Bloq)

T →  T literalEntero
imprime(**tipoLista**(Tipo,literalEntero)):
    imprime(Tipo)
    print "["
    imprime(literalEntero)
    print "]"

T →  ^T
imprime(**tipoCircum**(Tipo)):

```
        print ”^”
        imprime(Tipo)

T → campos identificador
imprime(tipoStruct(Campos, Id)):
        print ”struct”
        print ”{”
        imprime(Campos)
        print ”}”
        print Id

T → identificador
imprime(tipoIden(id)):
        print(id)
T → int
imprime(tipoInt()):
        print(“int”)
T → real
imprime(tipoReal()):
        print(“real”)
T → bool
imprime(tipoBool()):
        print(“bool”)
T → string
imprime(tipoString()):
        print(“string”)

// imprime(E()): // Creo que con lo de abajo sirve

campos → campos campo
imprime(muchosCampos(Campos,Campo)):
        imprime(Campos)
        print “,”
        imprime(Campo)


campos→ campo
imprime(unCampo(Campo)):
        imprime(Campo)

campo→ Tipo Identificador
imprime(CreaCampo(Tipo, Id)):
        imprime(Tipo)
        print “ ” ++ Id
```

parametrosFormales_opt → parametrosFormales
imprime(**siParsF**(ParsF)):
    imprime(ParsF)
parametrosFormales_opt → ε
imprime(**noParsF**()):
    noop // vacio, NO OPeration
parametrosFormales → parametrosFormales parametroFormal
imprime(**muchosParsF**(ParsF, ParF)):
    imprime(ParsF)
    print ","
    imprime(ParF)
parametroFormales → parametroFormal
imprime(**unParF**(ParF)):
    imprime(ParF)
parametroFormal → T & identificador
imprime(**paramF**(string, Tipo)):
    imprime(string)
    print " & "
    imprime(Tipo)
parametroFormal → T  identificador
imprime(**param**(string, Tipo)):
    imprime(string)
    imprime(Tipo)

imprime(**identificador**(Id)):
    print Id
imprime(**literalEntero**(num)):
    print  num
imprime(**literalReal**(num)):
    print num
imprime(**literalCadena**(S)):
    print S
imprime(**tipoBool**(B)):
    print B

**imprimeExpBin**(Opnd0,Op,Opnd1,np0,np1):
    imprimeOpnd(Opnd0,np0)
    print " "++Op++" "
    imprimeOpnd(Opnd1,np1)
**imprimeExpUnarPre**(Opnd,Op, np):
    print " "++Op++" "
    imprimeOpnd(Opnd,np)  // Asociativo
**imprimeExpUnarPost**(Opnd,Op, np):

```
        imprimeOpnd(Opnd,np) // Asociativo
        print " "++Op++" "
imprimeOpnd(Opnd,MinPrior):
        if prioridad(Opnd) < MinPrior
                print "("
        end if
        imprime(Opnd)
        if prioridad(Opnd) < MinPrior
                print ")"
        end if

imprime(suma(Opnd0,Opnd1)):
        imprimeExpBin(Opnd0,"+",Opnd1, 2,3) // Asocia a izquierdas
imprime(resta(Opnd0,Opnd1)):
        imprimeExpBin(Opnd0,"-",Opnd1, 3,3) // No asocia

imprime(and(Opnd0,Opnd1)):
        imprimeExpBin(Opnd0,"and",Opnd1, 4,3)   // Asocia a derechas
imprime(or(Opnd0,Opnd1)):
        imprimeExpBin(Opnd0,"or",Opnd1, 4,4) // No asocia


Operadores binarios, infijos, asociativos a izquierdas
imprime(multiplicacion(Opnd0,Opnd1)):
        imprimeExpBin(Opnd0,"*",Opnd1, 4,5)
imprime(division(Opnd0,Opnd1)):
        imprimeExpBin(Opnd0,"/",Opnd1, 4,5)
imprime(modulo(Opnd0,Opnd1)):
        imprimeExpBin(Opnd0,"%",Opnd1, 4,5)

Operadores unarios, prefijos, asociativos
imprime(menosUnario(Opnd)):
        imprimeExpUnarPre(Opnd,"-", 5)
imprime(not(Opnd)):
        imprimeExpUnarPre(Opnd,"not", 5)

Operadores unarios posfijos, asociativos

E → E E
imprime(indexacion(Opnd1, Opnd2)):
        imprime(Opnd1)
        print("[")
        imprime(Opnd2)
        print("]")
```

E → E identificador
imprime(**acceso**(Opnd, Id)):
    imprime(Opnd)
    print(“.”)
    print(“ ”++ Id)

E → E
imprime(**indireccion**(Opnd)):
    imprime(Opnd)
    print(“^”)

prioridad(**asignacion**(_,_)): return 0

prioridad(**menorI**(_,_)): return 1
prioridad(**menor**(_,_)): return 1
prioridad(**mayor**(_,_)): return 1
prioridad(**mayorI**(_,_)): return 1
prioridad(**igual**(_,_)): return 1
prioridad(**distint**(_,_)): return 1

prioridad(**suma**(_,_)): return 2
prioridad(**resta**(_,_)): return 2

prioridad(**and**(_,_)): return 3
prioridad(**or**(_,_)): return 3

prioridad(**multiplicacion**(_,_)): return 4
prioridad(**division**(_,_)): return 4
prioridad(**modulo**(_,_)): return 4

prioridad(**not**(_)): return 5
prioridad(**menosUnario**(_)): return 5

prioridad(**indexacion**(_)): return 6 // Exp x Exp
prioridad(**acceso**(_)): return 6 // Exp x Exp
prioridad(**indereccion**(_)): return 6

# 4. Especificación JavaCC (ASTs Descendente)

```
TOKEN:{<#letra:["a"-"z","A"-"Z","_"]>}
 TOKEN:{<#digitoPositivo:["1"-"9"]>}
 TOKEN:{<#digito:<digitoPositivo>|"0">}
 TOKEN:{<#parteEntera:<digitoPositivo> (<digito>)* |"0">}
 TOKEN:{<#parteDecimal: (<digito>)* <digitoPositivo> | "0">}
 TOKEN:{<#parteExponencial: ("E"|"e")(["+","-"])?(<parteEntera>)>}
  SKIP:{<["\t"," ","\r","\b","\n"]>}
 SKIP:{<"#"(~["\n"])*>}
  TOKEN: {<TRUE:"true">}
 TOKEN: {<FALSE:"false">}
 TOKEN: {<and:"and">}
 TOKEN: {<or:"or">}
 TOKEN: {<not:"not">}
 TOKEN: {<bool:"bool">}
 TOKEN: {<real:"real">}
 TOKEN: {<INT:"int">}
 TOKEN: {<string:"string">}
 TOKEN: {<NULL:"null">}
 TOKEN: {<proc:"proc">}
 TOKEN: {<IF:"if">}
 TOKEN: {<ELSE:"else">}
 TOKEN: {<WHILE:"while">}
 TOKEN: {<struct:"struct">}
 TOKEN: {<NEW:"new">}
 TOKEN: {<delete:"delete">}
 TOKEN: {<read:"read">}
 TOKEN: {<write:"write">}
 TOKEN: {<nl:"nl">}
 TOKEN: {<type:"type">}
 TOKEN: {<call:"call">}

  TOKEN:{<identificador:<letra>(<letra>|<digito>)*>}
 TOKEN:{<literalEntero:(["+","-"])?<parteEntera>>}

TOKEN:{<literalReal:<literalEntero>(<parteExponencial>|("."<parteDecimal>)|("."<parteDecimal
><parteExponencial>))>}
 TOKEN:{<literalCadena:"\"" (~["\""])* "\"">}
 TOKEN:{<suma:"+">}
 TOKEN:{<resta:"-">}
 TOKEN:{<mul:"*">}
 TOKEN:{<div:"/">}
 TOKEN:{<mayor:">">}
 TOKEN:{<mayorIgual:">=">}
 TOKEN:{<menor:"<">}
 TOKEN:{<menorIgual:"<=">}
 TOKEN:{<igual:"==">}
 TOKEN:{<distinto:"!=">}
 TOKEN:{<asignar:"=">}
 TOKEN:{<puntoycoma:";">}
 TOKEN:{<parentesisApertura:"(">}
```

```
TOKEN:{<parentesisCierre:")">}
TOKEN:{<llaveApertura:"{">}
TOKEN:{<llaveCierre:"}">}
TOKEN:{<eval:"@">}
TOKEN:{<modulo:"%">}
TOKEN:{<corcheteApertura:"[">}
TOKEN:{<corcheteCierre:"]">}
TOKEN:{<punto:".">}
TOKEN:{<coma:",">}
TOKEN:{<circunflejo:"^">}
TOKEN:{<ampersand:"&">}
TOKEN:{<separador:"&&">}
 Prog inicial() :
{Prog prog;}
{
        prog = programa() <EOF>
        {return prog;}
}
Prog programa() :
{Bloq bloq;}
{
        bloq = bloque()
        {return sem.prog(bloq);}
}
Bloq bloque() :
{DecsOp decsOp; InstrsOp instrsOp ;}
{
        <llaveApertura> decsOp = declaraciones_opt() instrsOp = instrucciones_opt() <llaveCierre>
        {return sem.bloq(decsOp, instrsOp);}
}
DecsOp declaraciones_opt() :
{Decs decs;}
{
        decs = declaraciones() <separador>
        {return sem.si_decs(decs);}
|
        {return sem.no_decs();}
}
InstrsOp instrucciones_opt() :
{ Instrs instrs;}
{
        instrs = instrucciones()
        {return sem.si_instrs(instrs);}
|
        {return sem.no_instrs();}
}
Tipo tipo1() :
{Tipo t2, rt1;}
{
        t2 = tipo2() rt1 = rtipo1(t2)
        {return rt1;}
}
Tipo rtipo1(Tipo th) :
```

```
{Token t; Tipo t1;}
{
        <corcheteApertura> t = <literalEntero> <corcheteCierre> t1 = rtipo1(th)
        {return sem.tipo_lista(th,t);}
|
        {return th;}
}
Tipo tipo2() :
{Tipo t;}
{
        <circunflejo> t= tipo2()
        {return sem.tipo_circum(t);}
|
        t= tipo3()
        {return t;}
}
Tipo tipo3() :
{Tipo t; Campos c; Token id; }
{
        t= tipoBasico()
        {return t;}
|
        <struct> <llaveApertura> c = Campos() <llaveCierre>
        {return sem.tipo_struct(c);}
|
        id= <identificador>
        {return sem.tipo_iden(id);}
}
Tipo tipoBasico() :
{Token t;}
{
        t = <INT>
        {return (Tipo)sem.tipo_int().ponFila(t.beginLine).ponCol(t.beginColumn);}
|
        t = <real>
        {return (Tipo)sem.tipo_real().ponFila(t.beginLine).ponCol(t.beginColumn);}
|
        t = <bool>
        {return (Tipo)sem.tipo_bool().ponFila(t.beginLine).ponCol(t.beginColumn);}
|
        t =<string>
        {return (Tipo)sem.tipo_string().ponFila(t.beginLine).ponCol(t.beginColumn);}
}
Campos Campos () :
{Campo campo; Campos campos;}
{
        campo = Campo() campos = Rcampos (sem.un_campo(campo))
        {return campos ;}
}
Campos Rcampos (Campos camposh) :
{Campo campo; Campos campos;}
{
        <coma> campo = Campo() campos = Rcampos (sem.muchos_campos(camposh, campo))
```

```
        {return campos;}
|
        {return camposh;}
}
Campo Campo() :
{Tipo t; Token id;}
{
        t = tipo1() id = <identificador>
        {return sem.crea_campo(t,id);}
}
Decs declaraciones() :
{Decs decs; Dec dec; }
{
        dec = declaracion() decs = DR(sem.una_dec(dec))
        {return decs;}
}
Decs DR(Decs dech) :
{Dec dec; Decs decs;}
{
        <puntoycoma> dec = declaracion() decs = DR(sem.muchas_decs(dech,dec))
        {return decs;}
|
        {return dech;}
}
Dec declaracion() :
{Tipo tipo; Token id; ParsFOp parsFOp; Bloq bloq;}
{
        tipo = tipo1() id = <identificador>
        {return (Dec)sem.dec_variable(tipo,id).ponFila(id.beginLine).ponCol(id.beginColumn);}
|
        <type> tipo = tipo1() id = <identificador>
        {return (Dec)sem.dec_tipo(tipo,id);}
|
        <proc> id = <identificador> <parentesisApertura> parsFOp = parametrosFormales_opt()
<parentesisCierre>
        bloq = bloque()
        {return (Dec)sem.dec_proc(id,parsFOp,bloq);}
}
ParsFOp parametrosFormales_opt() :
{ParsF parsF;}
{
        parsF= parametrosFormales()
        {return sem.si_parsF(parsF);}
|
        {return sem.no_parsF();}
}
ParsF parametrosFormales() :
{ParsF parsF; ParF parF;}
{
        parF = parametroFormal() parsF = RparametrosFormales(sem.un_parF(parF))
        {return parsF;}
}
ParsF RparametrosFormales(ParsF parsFh) :
```

```
{ParF parF; ParsF parsF;}
{
        < coma > parF = parametroFormal() parsF =
RparametrosFormales(sem.muchos_parsF(parsFh, parF))
        {return parsF;}
|
        {return parsFh;}
}
ParF parametroFormal () :
{Tipo tipo; Token id;}
{
        tipo = tipo1() id = <identificador>
        {return (ParF)sem.param(id.image,tipo); }
}
Instrs instrucciones () :
{Instr instr; Instrs instrs;}
{
        instr = instruccion() instrs = IR(sem.una_instr(instr))
        {return instrs;}
}
Instrs IR(Instrs instrh) :
{Instr instr; Instrs instrs;}
{
        <puntoycoma> instr = instruccion() instrs = IR(sem.muchas_instrs(instrh, instr))
        {return instrs;}
|
        {return instrh;}
}
Instr instruccion () :
{Exp e; Bloq bloq; Token id; ParsReOp parsReOp; Instr i;}
{
        <eval> e = Eo()
        {return sem.instr_eval(e);}
|
        <IF> e = Eo() bloq=bloque() i=IFR(e,bloq)
        //<IF> e = Eo() bloq=bloque() // Un solo if
        {return i;}|
        <WHILE> e = Eo() bloq = bloque()
        {return sem.instr_while(e, bloq);}
|
        <read> e = Eo()
        {return sem.instr_read(e);}
|
        <write> e = Eo()
        {return sem.instr_write(e);}
|
        <nl>
        {return sem.instr_nl();}
|
        <NEW>  e = Eo()
        {return sem.instr_new(e);}
|
        <delete> e = Eo()
```

```
        {return sem.instr_del(e);}
|
        <call> id = <identificador> <parentesisApertura> parsReOp= parametrosReales_opt()
<parentesisCierre>

        {return sem.instr_call(id,parsReOp); }
|
        bloq = bloque()
        {return sem.instr_bloque(bloq);}
}
Instr IFR(Exp eh, Bloq bloqh):
{Bloq bloq;}
{
        <ELSE> bloq=bloque()
        {return sem.instr_ifelse(eh,bloqh,bloq);}
|
        {return sem.instr_if(eh,bloqh);}
}
ParsReOp parametrosReales_opt() :
{ParsRe parsRe;}
{
        parsRe = parametrosReales()
        {return sem.si_parsRe(parsRe);}
|
        {return sem.no_parsRe();}
}
ParsRe parametrosReales()  :
{Exp exp; ParsRe parsRe;}
{
        exp = E0() parsRe = RparametrosReales(sem.un_parRe(exp))
{return parsRe;}
}
ParsRe RparametrosReales(ParsRe parsReh)  :
{Exp exp; ParsRe parsRe;}
{
        <coma> exp = E0() parsRe = RparametrosReales(sem.muchos_parsRe(parsReh, exp))
        {return parsRe;}
|
        {return parsReh;}
}

Exp E0 ()  :
{Exp e1,e2;}
{
        e1 = E1() e2 = FE0(e1)
        { return e2;}
}
Exp FE0(Exp eh)  :
{Exp e1,e2;}
{
        <asignar> e1 = E1() e2 = FE0(sem.mkop("=",eh,e1)) // op1?
        {return e2;}
```

```
|
        {return eh;}
}
Exp E1 () :
{Exp e1,e2;}
{
        e1 = E2() e2 = Re1(e1)
        { return e2;}
}
Exp Re1(Exp eh) :
{String op; Exp e1,e2;}
{
        op = OP1() e1 = E2() e2 =  Re1(sem.mkop(op,eh,e1))
        {return e2;}
|
        {return eh;}
}
String OP1() :
{}
{
        <menor>
        {return "<";}
|
        <mayor>
        {return ">";}
|
        <menorIgual>
        {return "<=";}
|
        <mayorIgual >
        {return ">=";}
|
        <igual>
        {return "==";}
|
        <distinto>
        {return "!=";}
}
Exp E2 () :
{Exp e1,e2,e3;}
{
        e1 = E3() e2 = FE3(e1) e3 = Re2(e2)
        { return e2;}
}
Exp Re2(Exp eh) :
{Exp e1,e2;}
{
        <suma> e1 = E3() e2 =  Re2(sem.mkop("+",eh,e1))
        {return e2;}
|
        {return eh;}
}
Exp FE3(Exp eh) :
```

```
{Exp e1;}
{
        <resta> e1 = E3()
        {return sem.mkop("-", eh,e1);}
|
        {return eh;}
}
Exp E3 () :
{Exp e1,e2;}
{
        e1 = E4() e2 = Fe4(e1)
        { return e2;}
}
Exp Fe4(Exp eh) :
{Exp e1;}
{
        <and> e1 = E3()
        {return sem.mkop("and",eh,e1);}
|
        <or> e1 = E4()
        {return sem.mkop("or",eh,e1);}
|
        {return eh;}
}
Exp E4() :
{Exp e1,e2;}
{
        e1 = E5() e2 = Re4(e1)
        {return e2;}
}
Exp Re4(Exp eh) :
{String op; Exp e1,e2;}
{
        op = OP4() e1 = E5() e2 = Re4(sem.mkop(op,eh,e1))
        {return sem.mkop("and",eh,e1);}
|
        {return eh;}
}
//Exp OP4() :
String OP4() :
{}
{
        <mul>
        {return "*";}
|
        <div>
        {return "/";}
|
        <modulo>
        {return "%";}
}
Exp E5() :
{String op; Exp e1;}
```

```
{
        op = OP5() e1 = E5()
        {return sem.mkopUn(op, e1);}
|
        e1=E6()
        {return e1;}
}
//Exp OP5() :
String OP5() :
{}
{
        <not>
        {return "not";}
|
        <resta>
        {return "-";}
}
Exp E6() :
{Exp e1,e2;}
{
        e1 = E7() e2 = RFE6(e1)
        {return e2;}
}
Exp RFE6(Exp eh) :
{Exp e1,e2;}
{
        e1 = FE6(eh) e2 = RFE6(e1)
        {return e2;}
|
        {return eh;}
}
Exp FE6(Exp eh) :
{Token iden; Exp e1;}
{
        <corcheteApertura> e1 = E0() <corcheteCierre> // "[" e1 = E0()"]"
        {return sem.indexacion(eh, e1);}
|
        <punto> iden = <identificador>//"." <identificador>
        {return (Exp) sem.acceso(eh, iden.image);} // cambiar en SintaxisAbstractaTiny
|
        <circunflejo> //"^"
        {return sem.indireccion(eh);}
}
Exp E7() :
{Token t; Exp e;}
{
        t = <identificador>
        {return (Exp)sem.iden(t.image).ponFila(t.beginLine).ponCol(t.beginColumn);}
|
        t = <literalEntero>
        {return (Exp)sem.lit_ent(t.image).ponFila(t.beginLine).ponCol(t.beginColumn);}
|
        t = <literalReal>
```

```
        {return (Exp)sem.lit_real(t.image).ponFila(t.beginLine).ponCol(t.beginColumn);}
|
        t = <TRUE> //"true"
        {return (Exp)sem.lit_true().ponFila(t.beginLine).ponCol(t.beginColumn);}
|
        t = <FALSE> //"false"
        {return (Exp)sem.lit_false().ponFila(t.beginLine).ponCol(t.beginColumn);}
|
        t = <literalCadena>
        {return (Exp)sem.lit_cadena(t.image).ponFila(t.beginLine).ponCol(t.beginColumn);}
|
        <parentesisApertura> e = Eo() <parentesisCierre>
        {return e;}
|
        t = <NULL> // "null"
        {return (Exp)sem.lit_null().ponFila(t.beginLine).ponCol(t.beginColumn);}
}
```

# 5. Especificación CUP +jflex (ASTs Ascendente)

```
package asint;

import java_cup.runtime.*;
import alex.AnalizadorLexicoTiny;
import alex.UnidadLexica;
import errors.GestionErroresTiny;


scan with {:
  return getScanner().next_token();
:};

parser code {:
   private GestionErroresTiny errores;
   public void syntax_error(Symbol unidadLexica) {
     errores.errorSintactico((UnidadLexica)unidadLexica);
   }
:};
init with {:
   errores = new GestionErroresTiny();
   AnalizadorLexicoTiny alex = (AnalizadorLexicoTiny)getScanner();
   alex.fijaGestionErrores(errores);
:};
action code {:
  ClaseSemanticaEval sem = new ClaseSemanticaEval();
:}




terminal        PAP, PCIERRE, COMA, EVAL, PYC, LLAP, LLCIERRE, SEP, TRUE,
FALSE, BOOL,ENT, REAL, CCIERRE, PUNTO, AMP, STRING, NULL, PROC, IF,
ELSE, WHILE, STRUCT, NEW, DELETE, READ, WRITE, NL, TYPE, CALL;

terminal        StringLocalizado IDEN, ASIG, MAS, MENOS, POR, DIV, GT, GE, LT,
LE, EQ, NE, MOD, AND, OR, NOT, CIRCUNFLEJO, LITERALENTERO,
LITERALREAL, LITERALCADENA, CAP;


non terminal            Prog programa;
```

```
non terminal        Bloq bloque;
non terminal        DecsOp declaraciones_opt;
non terminal        Decs declaraciones;
non terminal        Dec declaracion;
non terminal        InstrsOp instrucciones_opt;
non terminal        Instrs instrucciones;
non terminal        Instr instruccion;
non terminal        Tipo tipo1, tipo2, tipo3, tipoBasico;
non terminal        Exp E0, E1, E2, E3, E4, E5, E6, E7;
non terminal        Campos campos;
non terminal        Campo campo;
non terminal        ParsFOp parametrosFormales_opt;
non terminal        ParsF parametrosFormales;
non terminal        ParF parametroFormal;
non terminal        ParsReOp parametrosReales_opt;
non terminal        ParsRe parametrosReales;
non terminal        String OP1, OP4, OP5;


programa  ::= bloque: Bloq
        {: RESULT = sem.prog(Bloq);  :};

bloque ::= LLAP declaraciones_opt: DecsOp instrucciones_opt: InstrOp LLCIERRE
        {: RESULT = sem.bloq(DecsOp,InstrOp); :};

declaraciones_opt  ::= declaraciones: Decs  SEP
        {: RESULT = siDecs(Decs); :};
declaraciones_opt  ::=
        {: RESULT = sem.noDecs(); :};
instrucciones_opt   ::=  instrucciones  : Instrs
        {: RESULT = sem.siInstrs(Instrs); :};
instrucciones_opt   ::=
        {: RESULT = sem.noInstrs(); :};

tipo1 ::= tipo1:Tipo CAP LITERALENTERO:num CCIERRE
        {: RESULT = (Tipo)sem.tipoLista(Tipo,
        num.str()).ponFila(num.fila()).ponCol(num.col()); :};

tipo1 ::= tipo2: Tipo
        {: RESULT = Tipo; :};

tipo2 ::=  CIRCUNFLEJO tipo2 : Tipo
        {: RESULT = sem.tipoCircum(Tipo); :};
tipo2 ::= tipo3 : Tipo
```

```
        {: RESULT = Tipo; :};


tipo3 ::= tipoBasico: Tipo
        {: RESULT = Tipo; :};
tipo3 ::= STRUCT LLAP campos: Campos LLCIERRE
        {: RESULT = sem.tipoStruct(Campos); :};
tipo3 ::= IDEN: id
        {:  RESULT  =  (Tipo)sem.tipoIden(id.str()).ponFila(id.fila()).ponCol(id.col());
:};

tipoBasico ::= ENT
        {: RESULT = sem.tipoInt(); :};
tipoBasico ::= REAL
        {: RESULT = tipoReal(); :};
tipoBasico ::= BOOL
        {: RESULT = tipoBool(); :};
tipoBasico ::= STRING
        {: RESULT = tipoString(); :};

campos ::= campos: Campos COMA campo: Campo
        {: RESULT = sem.muchosCampos(Campos, Campo); :};

campos ::= campo: Campo
        {: RESULT = sem.unCampo(Campo)};

campo ::= tipo1: Tipo IDEN: id
        {: RESULT = (Campo)sem.creaCampo(Tipo,
id.str()).ponFila(id.fila()).ponCol(id.col()); :};

declaraciones  ::= declaraciones: Decs PYC declaracion: Dec
        {: RESULT = sem. muchasDecs(Decs,Dec); :};

declaraciones  ::= declaracion: Dec
        {: RESULT = sem.unaDec(Dec); :};

declaracion  ::= tipo1: Tipo IDEN: id
        {: RESULT = (Tipo)sem.decVariable(Tipo,
id.str()).ponFila(id.fila()).ponCol(id.col()); :};

declaracion  ::=TYPE tipo1: Tipo IDEN: id
        {: RESULT = (Tipo)sem.decTipo(Tipo,
id.str()).ponFila(id.fila()).ponCol(id.col()); :};
```

```
declaracion  ::=PROC  IDEN: id PAP  parametrosFormales_opt: ParsFOp PCIERRE
bloque: Bloq
        {: RESULT = (Dec)sem.decProc(id.str(), ParsFOp,
Bloq).ponFila(id.fila()).ponCol(id.col()); :};

parametrosFormales_opt  ::= parametrosFormales: ParsF
        {: RESULT = sem. siParsF(ParsF ); :};
parametrosFormales_opt  ::=
        {: RESULT = sem.noParsF(); :};
parametrosFormales ::= parametrosFormales: ParsF COMA parametroFormal: ParF
        {: RESULT = sem.muchosParsF(ParsF, ParF); :};

parametrosFormales ::= parametroFormal: ParF
        {: RESULT = sem. unParF(ParF); :};

parametroFormal  ::= tipo1: Tipo AMP IDEN: id
        {: RESULT = sem.paramF(Tipo, id.str()).ponFila(id.fila()).ponCol(id.col()); :};

parametroFormal  ::= tipo1: Tipo IDEN: id
        {: RESULT = sem.param(Tipo, id.str()).ponFila(id.fila()).ponCol(id.col()); :};

instrucciones ::= instrucciones: Instrs PYC instruccion: Instr
        {: RESULT = sem.muchasInstrs(Instrs, Instr); :};

instrucciones ::= instruccion: Instr
        {: RESULT = sem.unaInstr(Instr); :};

instruccion   ::= EVAL  E0: Exp
        {: RESULT = sem.instrEval(Exp); :};

instruccion   ::= IF E0: Exp bloque: Bloq
        {: RESULT = sem.instrIf(Exp,Bloq) ; :};

instruccion   ::= IF E0: Exp bloque: Bloq1 ELSE bloque: Bloq2
        {: RESULT = sem.instrIfElse(Exp,Bloq1,Bloq2) ; :};

instruccion   ::= WHILE E0: Exp bloque: Bloq
        {: RESULT = sem.instrWhile(Exp ,Bloq) ; :};

instruccion   ::= READ E0: Exp
        {: RESULT = sem.instrRead(Exp) ; :};

instruccion   ::= WRITE E0: Exp
```

```
        {: RESULT = sem.instrWrite(Exp) ; :};

instruccion   ::= NL
        {: RESULT = sem.instrNl(); :};

instruccion   ::= NEW E0: Exp
        {: RESULT = sem.instrNew(Exp) ; :};

instruccion   ::= DELETE E0: Exp
        {: RESULT = sem.instrDel(Exp); :};

instruccion   ::= CALL IDEN: id PAP parametrosReales_opt: ParsReOp PCIERRE
        {: RESULT = sem.instrCall(id.str(), ParsReOp
).ponFila(id.fila()).ponCol(id.col()); :};

instruccion   ::= bloque: Bloq
        {: RESULT = sem.instrBloque(Bloq); :};

parametrosReales_opt  ::= parametrosReales: ParsRe
        {: RESULT = sem.siParsRe(ParsRe ); :};
parametrosReales_opt  ::=
        {: RESULT = sem.noParsRe(); :};
parametrosReales  ::= parametrosReales: ParsRe COMA E0: Exp
        {: RESULT = sem.muchosParsRe(ParsRe, Exp); :};

parametrosReales  ::= E0: Exp
        {: RESULT = sem.unParRe(Exp); :};


E0 ::= E1: opnd1 ASIG:op E0: opnd2
        {: RESULT = (Exp)sem.asig(opnd1,
opnd2).ponFila(op.fila()).ponCol(op.col()); :};

E0 ::= E1: Exp
        {: RESULT = Exp; :};


E1 ::=  E1: opnd1 OP1: op E2: opnd2
        {: RESULT = (Exp)sem.mkop(op, opnd1 ,
opnd2).ponFila(op.fila()).ponCol(op.col()); :};

E1 ::= E2: Exp
        {: RESULT = Exp; :};
```

```
OP1 ::= LT
        {: RESULT = “<”; :};

OP1 ::= GT
        {: RESULT = “>” :};

OP1 ::= LE
        {: RESULT = “<=”; :};

OP1 ::= GE
        {: RESULT = “>=”; :};

OP1 ::= EQ
        {: RESULT = “==”; :};

OP1 ::= NE
        {: RESULT = “!=”; :};

E2 ::= E2: opnd1 MAS: op E3: opnd2
        {: RESULT = (Exp)sem.mkop(“+”, opnd1 ,
opnd2).ponFila(op.fila()).ponCol(op.col()); :};
E2 ::= E3 MENOS: op E3
        {: RESULT = (Exp)sem.mkop(“-”, opnd1 ,
opnd2).ponFila(op.fila()).ponCol(op.col()); :};
E2 ::=   E3: Exp
        {: RESULT = Exp; :};

E3 ::= E4: opnd1 AND: op  E3: opnd2
        {: RESULT = (Exp)sem.mkop(“and”, opnd1 ,
opnd2).ponFila(op.fila()).ponCol(op.col()); :};
E3 ::= E4: opnd1  OR: op  E4: opnd2
        {: RESULT = (Exp)sem.mkop(“or”, opnd1 ,
opnd2).ponFila(op.fila()).ponCol(op.col()); :};
E3 ::= E4: Exp
        {: RESULT = Exp; :};

E4 ::= E4: opnd1 OP4: op E5: opnd2
        {: RESULT = (Exp)sem.mkop(op, opnd1 ,
opnd2).ponFila(op.fila()).ponCol(op.col()); :};
E4 ::= E5: Exp
        {: RESULT = Exp; :};
OP4 ::= POR
        {: RESULT =”*”; :};
OP4 ::= DIV
```

```
        {: RESULT = ”/”; :};
OP4 ::= MOD
        {: RESULT = ”%”; :};

E5 ::= OP5: op E5: opnd
        {: RESULT = (Exp)mkopUn(op, opnd).ponFila(op.fila()).ponCol(op.col()); :};
E5 ::= E6: Exp
        {: RESULT = Exp; :};
OP5 ::= NOT
        {: RESULT = “not”; :};
OP5 ::= MENOS
        {: RESULT = “-”; :};

E6 ::= E6: opnd1 CAP:op E0: opnd2 CCIERRE
        {: RESULT = sem.indexacion (opnd1 ,
opnd2).ponFila(op.fila()).ponCol(op.col()); :};

E6 ::= E6: opnd1 PUNTO IDEN: id
        {: RESULT = sem.acceso (opnd1 , id.str()).ponFila(id.fila()).ponCol(id.col());
:};
E6 ::= E6: opnd CIRCUNFLEJO: op
        {: RESULT = sem.indireccion(opnd).ponFila(op.fila()).ponCol(op.col()); :};
E6 ::= E7: Exp
        {: RESULT = Exp; :};

E7 ::= IDEN: id
        {: RESULT = (Exp)sem.iden(id.str()).ponFila(id.fila()).ponCol(id.col()); :};

E7 ::= LITERALENTERO: num
        {: RESULT = (Exp)sem.literalEntero
(num.str()).ponFila(num.fila()).ponCol(num.col()); :};

E7 ::= LITERALREAL: num
        {: RESULT = (Exp)sem.literalReal(num).ponFila(num.fila()).ponCol(num.col()); :};

E7 ::= TRUE
        {: RESULT = sem.true(); :};

E7 ::= FALSE
        {: RESULT = sem.false(); :};

E7 ::= LITERALCADENA: cad
        {: RESULT = (Exp)literalCadena(cad.str()).ponFila(cad.fila()).ponCol(cad.col()); :};

E7 ::= PAP E0: exp PCIERRE
```

```
        {: RESULT = exp; :};

E7 ::= NULL
        {: RESULT = sem.null(); :};
```