TERCERA FASE: CONSTRUCTOR DE ASTs PARA TINY(1)

Javier Gómez Moraleda
Mario Quiñones Pérez
Grupo 18

1. Especificación de la sintaxis abstracta de Tiny(1) mediante la enumeración de las signaturas (cabeceras) de las funciones constructoras de ASTs.

CONSTRUCTORAS
P_decs: Declaraciones X Instrucciones → Programa **P_nodecs**: Instrucciones → Programa
decs_muchas: Declaraciones X Declaracion → Declaraciones **decs_una**: Declaraciones → Declaraciones
decvar: Tipo X String → Declaracion
dectipo: Tipo X String → Declaracion
decproc: String X ParamForms X Bloque → Declaracion
tipo_id: String → Tipo **tipo_array**: String X Tipo→ Tipo **tipo_record**: Camps → Tipo **tipo_pointer**: Tipo→ Tipo
ins_muchas: Instrucciones X Instruccion → Instrucciones **ins_una**: Instruccion → Instrucciones **lista_inst_empty:** → Instrucciones
asig: Exp X Exp → Instruccion **if_inst**: Exp X Instrucciones → Instruccion **if_else**: Exp X Instrucciones X Instrucciones→ Instruccion **while_inst**: Exp X Instrucciones → Instruccion **read**: Exp → Instruccion **write**: Exp → Instruccion **nl**→ Instruccion **new_cons**: Exp → Instruccion **delete**: Exp → Instruccion **call**: String X Exps → Instruccion
bloque_inst: Bloque → Instruccion
pformales_empty: → ParamForms **pformal_muchos**: ParamForms X ParamForm → ParamForms **pformal_uno** : ParamForms → ParamForms **pformal_ref**: Tipo X String → ParamForm **pformal**: Tipo X String → ParamForm
int_cons: → Tipo **real_cons**: → Tipo **bool_cons**: → Tipo **string_cons** → Tipo **campo_muchos** : Camps X Camp → Camps **campo_uno** : Camp → Camps

campo : Tipo X String → Camp
lista_exp_empty: → Exps **exp_muchas** : Exps X Exp → Exps **exp_una** : Exp → Exps
bloque_prog: Programa → Bloque **no_bloque** → Bloque
suma: Exp X Exp→ Exp **resta**: Exp X Exp→ Exp
and_cons: Exp X Exp→ Exp **or_cons**: Exp X Exp→ Exp
menor: Exp X Exp→ Exp **mayor**: Exp X Exp→ Exp **menorIgual**: Exp X Exp→ Exp **mayorIgual**: Exp X Exp→ Exp **igual**: Exp X Exp→ Exp **distinto**: Exp X Exp→ Exp
mul: Exp X Exp→ Exp **div**: Exp X Exp→ Exp
neg: Exp→ Exp **not**: Exp→ Exp
litTrue: → Exp **litFalse**: → Exp **litEnt**: String → Exp **litReal**: String → Exp **litCad**: String → Exp **id**: String → Exp **litNull**: → Exp
star : String X Exp → Exp
expN5: Exp X Exp → Exp **corchete:** Exp → Exp **punto**: String → Exp **flecha**: String → Exp

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

Definición de la gramática	**Definiciones auxiliares**
PROGRAMA → DECLARACIONES && INSTRUCCIONES PROGRAMA .a = P_decs(DECLARACIONES.a, INSTRUCCIONES.a) PROGRAMA → INSTRUCCIONES PROGRAMA .a = P_nodecs(INSTRUCCIONES.a)	
DECLARACIONES → DECLARACIONES ; DECLARACION DECLARACIONES_0.a = decs_muchas(DECLARACIONES_1.a, DECLARACION.a) DECLARACIONES → DECLARACION DECLARACIONES.a = decs_una(DECLARACION.a)	
DECLARACION → DECVAR DECLARACION.a = DECVAR.a DECLARACION → DECTIPO DECLARACION.a = DECTIPO.a DECLARACION → DECPROC DECLARACION.a = DECPROC.a	
DECVAR → VAR **TIPO** ID DECVAR.a = decvar(TIPO.a, ID.lex)	
DECTIPO → TYPE **TIPO** ID DECTIPO.a = dectipo(TIPO.a, ID.lex)	
DECPROC → PROC ID **PFORMALES BLOQUE** DECPROC.a = decproc(ID.lex, PFORMALES.a, BLOQUE.a)	PFORMALES → (**LISTA_PFORMALES**) PFORMALES.a = LISTA_PFORMALES.a PFORMALES → ( ) PFORMALES.a = pformales_empty() LISTA_PFORMALES → LISTA_PFORMALES , **PFORMAL** LISTA_PFORMALES.a = pformal_muchos(LISTA_PFORMALES.a, PFORMAL.a) LISTA_PFORMALES → **PFORMAL** LISTA_PFORMALES.a = pformal_uno(PFORMAL.a) PFORMAL → **TIPO** & ID

	PFORMAL.a = pformal_ref(TIPO.a, ID.lex) PFORMAL → **TIPO** ID PFORMAL.a = pformal(TIPO.a, ID.lex)
TIPO → **TIPO_BASICO** TIPO.a = TIPO_BASICO.a TIPO → ID TIPO.a = tipo_id(ID.lex) TIPO → ARRAY CAP LIT_ENT CCIERRE OF TIPO TIPO_0.a = tipo_array(LIT_ENT.lex, TIPO_1.a) TIPO → RECORD LLAP **LISTA_CAMPOS** LLCIERRE TIPO.a = tipo_record(LISTA_CAMPOS.a) TIPO → POINTER TIPO TIPO_0.a = tipo_pointer(TIPO_1.a)	TIPO_BASICO→ INT TIPO_BASICO.a = int_cons() TIPO_BASICO→ REAL TIPO_BASICO.a = real_cons() TIPO_BASICO→ BOOL TIPO_BASICO.a = bool_cons() TIPO_BASICO→ STRING TIPO_BASICO.a = string_cons() LISTA_CAMPOS → LISTA_CAMPOS ; CAMPO LISTA_CAMPOs.a = campo_muchos(LISTA_CAMPOS.a, CAMPO.a) LISTA_CAMPOS → CAMPO LISTA_CAMPOS.a = campo_uno(CAMPO.a) CAMPO → TIPO ID CAMPO.a = campo(TIPO.a, ID.lex)
INSTRUCCIONES → INSTRUCCIONES ; INSTRUCCION INSTRUCCIONES_0.a = ins_muchas(INSTRUCCIONES_1.a, INSTRUCCION.a) INSTRUCCIONES → INSTRUCCION INSTRUCCIONES.a = ins_una(INSTRUCCION.a)	
INSTRUCCION → **EXPRESION** = **EXPRESION** INSTRUCCION.a = asig(EXPRESION_0.a, EXPRESION_1.a)	
INSTRUCCION → IF **EXPRESION** THEN **LISTA_INST_OP** ENDIF INSTRUCCION.a = if_inst(EXPRESION.a, LISTA_INST_OP.a) INSTRUCCION → IF **EXPRESION** THEN **LISTA_INST_OP** ELSE **LISTA_INST** ENDIF INSTRUCCION.a = if_else(EXPRESION.a, LISTA_INST_OP.a, RIFTHEN.a)	LISTA_INST_OP → LISTA_INST LISTA_INST_OP.a = LISTA_INST.a LISTA_INST_OP → ε LISTA_INST_OP.a = lista_inst_empty() LISTA_INST → LISTA_INST ; INSTRUCCION LISTA_INST.a = ins_muchas(LISTA_INST.a, INSTRUCCION.a) LISTA_INST → INSTRUCCION

	LISTA_INST.a = ins_una(INSTRUCCION.a)
INSTRUCCION → WHILE **EXPRESION** DO **LISTA_INST_OP** ENDWHILE INSTRUCCION.a = while_inst(EXPRESION.a, LISTA_INST_OP.a)	
INSTRUCCION → READ **EXPRESION** INSTRUCCION.a = read(EXPRESION.a)	
INSTRUCCION → WRITE **EXPRESION** INSTRUCCION.a = write(EXPRESION.a)	
INSTRUCCION → NL INSTRUCCION.a = nl()	
INSTRUCCION → NEW **EXPRESION** INSTRUCCION.a = new_cons(EXPRESION.a)	
INSTRUCCION → DELETE **EXPRESION** INSTRUCCION.a = delete(EXPRESION.a)	
INSTRUCCION → CALL ID PAP **LISTA_EXPR_OP** PCIERRE INSTRUCCION.a = call(ID.lex, LISTA_EXPR_OP.a)	LISTA_EXPR_OP → **LISTA_EXPR** LISTA_EXPR_OP.a = LISTA_EXPR.a LISTA_EXPR_OP → ε LISTA_EXPR_OP.a = lista_exp_empty() LISTA_EXPR → LISTA_EXPR , **EXPRESION** LISTA_EXPR.a = exp_muchas(LISTA_EXPR.a, EXPRESION.a) LISTA_EXPR → **EXPRESION** LISTA_EXPR.a = exp_una(EXPRESION.a)
INSTRUCCION → BLOQUE INSTRUCCION.a = bloque_inst(BLOQUE.a)	BLOQUE → { PROGRAMA } BLOQUE.a = bloque_prog(PROGRAMA.a) BLOQUE → { } BLOQUE.a = no_bloque()
EXPRESION → E0 EXPRESION.a = E0.a	EXPRESION_BASICA → true EXPRESION_BASICA.a = litTrue() EXPRESION_BASICA → false EXPRESION_BASICA.a = litFalse() EXPRESION_BASICA → LIT_ENT EXPRESION_BASICA.a = litEnt(LIT_ENT.lex) EXPRESION_BASICA → LIT_REAL EXPRESION_BASICA.a = litReal(LIT_REAL.lex) EXPRESION_BASICA → LIT_CAD

	EXPRESION_BASICA.a = litCad(LIT_CAD.lex) EXPRESION_BASICA → ID EXPRESION_BASICA.a = id(ID.lex) EXPRESION_BASICA → NULL EXPRESION_BASICA.a = litNull()
E0 → E0 + E1 E0_0.a = exp(mas.op, E0_1.a, E1.a) E0 → E1 - E1 E0.a = exp(menos.op, E1_0.h, E1_1.a) E0 → E1 E0.a = E1.a	
E1 → E1 **OPBN1** E2 E1_0.a = exp(OPBN1.op, E1_1.a, E2.a) E1 → E2 E1.a = E2.a	OPBN1 → and OPBN1.op = “and” OPBN1 → or OPBN1.op = “or”
E2 → E2 **OPBN2** E3 E2_0.a = exp(OPBN2.op, E2_1.a, E3.a) E2 → E3 E2.a = E3.a	OPBN2 → < OPBN2.op = ‘<’ OPBN2 → > OPBN2.op = ‘>’ OPBN2 → <= OPBN2.op = “<=” OPBN2 → >= OPBN2.op = “>=” OPBN2 → == OPBN2.op = “==” OPBN2 → != OPBN2.op = “!=”
E3 → E4 **OPBN3** E4 E3.a = exp(OPBN3.op, E4_0.a, E4_1.a) E3 → E4 E3.a = E4.a	OPBN3 → * OPBN3.op = ‘*’ OPBN3 → / OPBN3.op = ‘/’ OPBN3 → % OPBN3.op = ‘%’
E4 → **not** E4 E4_0.a = not(E4_1.a) E4 → - E5 E4.a = neg(E5.a)	

E4 → E5 E4.a = E5.a	
E5 → E5 **OPUN5** E5_0.a = exp(E5_1.a, OPUN5.a) E5 → E6 E5.a = E6.a	OPUN5 → [EXPRESION] OPUN5.a = corchete(EXPRESION.a) OPUN5 → OP_ACCESO OPUN5.a = OP_ACCESO.a OP_ACCESO → . ID OP_ACCESO.a = punto(ID.lex) OP_ACCESO → -> ID OP_ACCESO.a = flecha(ID.lex)
E6 → ***** E6 E6_0.a = exp('*', E6_1.a) E6 → E7 E6.a = E7.a	
E7 → **EXPRESION_BASICA** E7.a = EXPRESION_BASICA.a E7 → (E0) E7.a = E0.a	

Funciones Semánticas

```
public Exp exp(String op, Exp arg0, Exp arg1) {
        switch (op) {
        case "+":
                return suma(arg0, arg1);
        case "-":
                return resta(arg0, arg1);
        case "*":
                return mul(arg0, arg1);
        case "%":
                return percent(arg0, arg1);
        case "/":
                return div(arg0, arg1);
        case "<":
                return mayor(arg0, arg1);
        case ">":
                return menor(arg0, arg1);
        case "and":
                return and_cons(arg0, arg1);
        case "or":
                return or_cons(arg0, arg1);
```

```
        case "<=":
                return menorIgual(arg0, arg1);
        case ">=":
                return mayorIgual(arg0, arg1);
        case "==":
                return igual(arg0, arg1);
        case "!=":
                return distinto(arg0, arg1);
        }
        throw new UnsupportedOperationException("exp " + op);
}

public Exp exp(String op, Exp arg0) {
        switch (op) {
        case "-":
                return menosUnario(arg0);
        case "not":
                return not(arg0);
        case "*":
                return star(arg0);
        }
        throw new UnsupportedOperationException("exp " + op);
}

public Exp exp(Exp arg0, Exp arg1) {
        return expN5(arg0, arg1);
}
```

3. Acondicionamiento de dicha especificación para permitir la implementación descendente.

Definición de la gramática	**Definiciones auxiliares**
PROGRAMA → DECLARACIONES && INSTRUCCIONES PROGRAMA .a = P_decs(DECLARACIONES.a, INSTRUCCIONES.a) PROGRAMA → INSTRUCCIONES PROGRAMA .a = P_nodecs(INSTRUCCIONES.a)	
DECLARACIONES → DECLARACION **RDECLARACIONES** RDECLARACIONES.h = decs_una(DECLARACION.a) DECLARACIONES.a = RDECLARACIONES.a	RDECLARACIONES → ; DECLARACION RDECLARACIONES RDECLARACIONES_1.h = decs_muchas(RDECLARACIONES_0.h, DECLARACION.a) RDECLARACIONES_0.a = RDECLARACIONES_1.a RDECLARACIONES → ε RDEC.a = RDEC.h
DECLARACION → DECVAR DECLARACION.a = DECVAR.a DECLARACION → DECTIPO DECLARACION.a = DECTIPO.a DECLARACION → DECPROC DECLARACION.a = DECPROC.a	
DECVAR → VAR **TIPO** ID DECVAR.a = decvar(TIPO.a, ID.lex)	
DECTIPO → TYPE **TIPO** ID DECTIPO.a = dectipo(TIPO.a, ID.lex)	
DECPROC → PROC ID **PFORMALES BLOQUE** DECPROC.a = decproc(ID.lex, PFORMALES.a, BLOQUE.a)	PFORMALES → (RPFORMALES) PFORMALES.a = RPFORMALES.a RPFORMALES → **LISTA_PFORMALES** RPFORMALES.a = LISTA_PFORMALES() RPFORMALES → ε RPFORMALES.a = pformales_empty() LISTA_PFORMALES → **PFORMAL RLISTA_PFORMALES** RLISTA_PFORMALES.h = pformal_uno(PFORMAL.a) LISTA_PFORMALES.a = RLISTA_PFORMALES.a

	RLISTA_PFORMALES → , **PFORMAL** RLISTA_PFORMALES RLISTA_PFORMALES_1.h = pformal_muchos(RLISTA_PFORMALES_0.h, PFORMAL.a) RLISTA_PFORMALES_0.a = RLISTA_PFORMALES_1.a RLISTA_PFORMALES → ε RLISTA_PFORMALES.a = RLISTA_PFORMALES.h PFORMAL → **TIPO** RPFORMAL PFORMAL.a = RPFORMAL.a RPFORMAL.h = TIPO.a RPFORMAL → & ID RPFORMAL.a = pformal_ref(RPFORMAL.h, ID.lex) RPFORMAL → ID RPFORMAL.a = pformal(RPFORMAL.h, ID.lex)
TIPO → **TIPO_BASICO** TIPO.a = TIPO_BASICO.a TIPO → ID TIPO.a = tipo_id(ID.lex) TIPO → ARRAY CAP LIT_ENT CCIERRE OF TIPO TIPO_0.a = tipo_array(LIT_ENT.lex, TIPO_1.a) TIPO → RECORD LLAP **LISTA_CAMPOS** LLCIERRE TIPO.a = tipo_record(LISTA_CAMPOS.a) TIPO → POINTER TIPO TIPO_0.a = tipo_pointer(TIPO_1.a)	TIPO_BASICO→ INT TIPO_BASICO.a = int_cons() TIPO_BASICO→ REAL TIPO_BASICO.a = real_cons() TIPO_BASICO→ BOOL TIPO_BASICO.a = bool_cons() TIPO_BASICO→ STRING TIPO_BASICO.a = string_cons() LISTA_CAMPOS → CAMPO RLISTA_CAMPOS RLISTA_CAMPOS.h = campo_uno(CAMPO.a) LISTA_CAMPOS.a = RLISTA_CAMPOS.a RLISTA_CAMPOS → ; CAMPO RLISTA_CAMPOS RLISTA_CAMPOS_1.h = campo_muchos(RLISTA_CAMPOS_0.h, CAMPO.a) RLISTA_CAMPOS_0.a = RLISTA_CAMPOS_1.a RLISTA_CAMPOS → ε RLISTA_CAMPOS.a = RLISTA_CAMPOS.h CAMPO → TIPO ID CAMPO.a = campo(TIPO.a, ID.lex)

INSTRUCCIONES → INSTRUCCION **RINST** RINST.h = ins_una(INSTRUCCION.a) INSTRUCCIONES.a = RINST.a	RINST → ; INSTRUCCION RINST RINST_1.h = ins_muchas(RINST_0.h, INSTRUCCION.a) RINST_0.a = RINST_1.a RINST → ε RINST.a = RINST.h
INSTRUCCION → **EXPRESION** = **EXPRESION** INSTRUCCION.a = asig(EXPRESION_0.a, EXPRESION_1.a)	
INSTRUCCION → IF **EXPRESION** THEN **LISTA_INST_OP** RIF INSTRUCCION.a = RIF.a RIF.h0 = LISTA_INST_OP.a RIF.h1 = EXPRESION.a RIF → ENDIF INSTRUCCION.a = if_inst(RIF.h1, RIF.h0) RIF → ELSE **LISTA_INST** ENDIF RIF.a = if_else(RIF.h1, RIF.h0, LISTA_INST.a)	LISTA_INST_OP → LISTA_INST LISTA_INST_OP.a = LISTA_INST.a LISTA_INST_OP → ε LISTA_INST_OP.a = lista_inst_empty() LISTA_INST → INSTRUCCION **RLISTA_INST** RLISTA_INST.h = ins_una(INSTRUCCION.a) LISTA_INST.a = RLISTA_INST.a RLISTA_INST → ; INSTRUCCION RLISTA_INST RLISTA_INST_1.h = ins_muchas(RLISTA_INST_0.h, INSTRUCCION.a) RLISTA_INST_0.a = RLISTA_INST_1.a RLISTA_INST → ε RLISTA_INST.a = RLISTA_INST.h
INSTRUCCION → WHILE **EXPRESION** DO **LISTA_INST_OP** ENDWHILE INSTRUCCION.a = while_inst(EXPRESION.a, LISTA_INST_OP.a)	
INSTRUCCION → READ **EXPRESION** INSTRUCCION.a = read(EXPRESION.a)	
INSTRUCCION → WRITE **EXPRESION** INSTRUCCION.a = write(EXPRESION.a)	
INSTRUCCION → NL INSTRUCCION.a = nl()	
INSTRUCCION → NEW **EXPRESION** INSTRUCCION.a = new_cons(EXPRESION.a)	
INSTRUCCION → DELETE **EXPRESION** INSTRUCCION.a = delete(EXPRESION.a)	
INSTRUCCION → CALL ID PAP **LISTA_EXPR_OP** PCIERRE INSTRUCCION.a = call(ID.lex,	LISTA_EXPR_OP → **LISTA_EXPR** LISTA_EXPR_OP.a = LISTA_EXPR.a

LISTA_EXPR_OP.a)	LISTA_EXPR_OP → ε LISTA_EXPR_OP.a = lista_exp_empty() LISTA_EXPR → **EXPRESION RLISTA_EXPR** RLISTA_EXPR.h = exp_una(EXPRESION.a) LISTA_EXPR.a = RLISTA_EXPR.a RLISTA_EXPR → , **EXPRESION** RLISTA_EXPR RLISTA_EXPR_1.h = exp_muchas(RLISTA_EXPR_0.h, EXPRESION.a) RLISTA_EXPR_0.a = RLISTA_EXPR_1.a RLISTA_EXPR → ε RLISTA_EXPR.a = RLISTA_EXPR.h
INSTRUCCION → BLOQUE INSTRUCCION.a = bloque_inst(BLOQUE.a)	BLOQUE → { RBLOQUE } BLOQUE.a = RBLOQUE.a RBLOQUE → PROGRAMA RBLOQUE.a = bloque_prog(PROGRAMA.a) RBLOQUE → ε RBLOQUE.a = no_bloque()
EXPRESION → E0 EXPRESION.a = E0.a	EXPRESION_BASICA → true EXPRESION_BASICA.a = litTrue() EXPRESION_BASICA → false EXPRESION_BASICA.a = litFalse() EXPRESION_BASICA → LIT_ENT EXPRESION_BASICA.a = litEnt(LIT_ENT.lex) EXPRESION_BASICA → LIT_REAL EXPRESION_BASICA.a = litReal(LIT_REAL.lex) EXPRESION_BASICA → LIT_CAD EXPRESION_BASICA.a = litCad(LIT_CAD.lex) EXPRESION_BASICA → ID EXPRESION_BASICA.a = id(ID.lex) EXPRESION_BASICA → NULL EXPRESION_BASICA.a = litNull()
E0 → E1 RE0 RE0.h = E1.a E0.a = RE0.a RE0 → + E1 RE0 RE0_1.h = exp(mas.op, E1.a, RE0_0.h) RE0_0.a = RE0_1.a	

RE0 → - E1 RE0.a = exp(menos.op, E1.a, RE0.h) RE0 → ε RE0.a = RE0.h	
E1 → E2 RE1 RE1.h = E2.a E1.a = RE1.a RE1 → **OPBN1** E2 RE1 RE1_1.h = exp(OPBN1.op, E2.a, RE1_0.h) RE1_0.a = RE1_1.a RE1 → ε RE1.a = RE1.h	OPBN1 → and OPBN1.op = "and" OPBN1 → or OPBN1.op = "or"
E2 → E3 RE2 RE2.h = E3.a E2.a = RE2.a RE2 → **OPBN2** E3 RE2 RE2_1.h = exp(OPBN2.op, E3.a, RE2_0.h) RE2_0.a = RE2_1.a RE2 → ε RE2.a = RE2.h	OPBN2 → < OPBN2.op = '<' OPBN2 → > OPBN2.op = '>' OPBN2 → <= OPBN2.op = "<=" OPBN2 → >= OPBN2.op = ">=" OPBN2 → == OPBN2.op = "==" OPBN2 → != OPBN2.op = "!="
E3 → E4 RE3 RE3.h = E4.a E3.a = RE3.a RE3 → **OPBN3** E4 RE3.a = exp(OPBN2.op, E4.a, RE3.h) RE3 → ε RE3.a = RE3.h	OPBN3 → * OPBN3.op = '*' OPBN3 → / OPBN3.op = '/' OPBN3 → % OPBN3.op = "%"
E4 → **not** E4 E4_0.a = not(E4_1.a) E4 → **-** E5 E4.a = neg(E5.a) E4 → E5 E4.a = E5.a	
E5 → E6 RE5 RE5.h = E6.a E5.a = RE5.a RE5 → **OPUN5** RE5 RE5_1.h = exp(OPUN5.op, RE5_1.h) RE5_0.a = RE5_1.a	OPUN5 → [EXPRESION] OPUN5.a = corchete(EXPRESION.a) OPUN5 → OP_ACCESO OPUN5.a = OP_ACCESO.a OP_ACCESO → . ID

RE5 → ε RE5.a = RE5.h	OP_ACCESO.a = punto(ID.lex) OP_ACCESO → -> ID OP_ACCESO.a = flecha(ID.lex)
E6 → * E6 E6_0.a = exp('*', E6_1.a) E6 → E7 E6.a = E7.a	
E7 → **EXPRESION_BASICA** E7.a = EXPRESION_BASICA.a E7 → (E0) E7.a = E0.a	