TINY 1

Grupo 13
Gorka Silva Ramón
Sofía Capmany Fernández

ÍNDICE DE CONTENIDOS

1. ESPECIFICACIÓN SINTÁCTICA

1.1 Especificación sintáctica

I. Terminales y No Terminales

— No Terminales:

PROG, DECS, LDEC, DEC, DEC_VAR, DEC_TIPO, DEC_PROC, PARFOR**,** LPF, PF, TIPO, TIPO_BASICO, TIPO_ID, TIPO_ARRAY, TIPO_RECORD, TIPO_POINTER, TAM, LCAMPO, CAMPO, LINST, INST, INST_ASIG, INST_IF_THEN, INST_IF_THEN_ELSE, INST_WHILE, INST_READ, INST_WRITE, INST_NL, INST_NEW, INST_CALL, INST_BLOQ, PARREL, LEREL, BLOQ, E0, E1, E2, E3, E4, E5, E6, E7, OPBAI1, OPBAI2, OPBNA, OPINDEX, OPACCREG, ACC, AUX_LINST

— Terminales:

sep_prog ≡ **&&**
pto_coma ≡ **;**
asig ≡ **=**
mas ≡ \+
menos ≡ \-
por ≡ *
div ≡ /
mod = **%**
blt ≡ **<**
bgt ≡ **>**
ble ≡ **<=**
bge ≡ **>=**
beq ≡ **==**
bne ≡ **!=**
pap ≡ \(
pcierre ≡ \)
cap = \[
ccierre = \]
llap = \{
llcierre = \}
flecha = →
coma = \,
punto = \.
et = **&**

r_int ≡ **int**
r_real ≡ **real**
r_bool ≡ **bool**
r_true ≡ **true**

r_false ≡ **false**
r_and ≡ **and**
r_or ≡ **or**
r_not ≡ **not**
r_string = **string**
r_null = **null**
r_proc = **proc**
r_if = **if**
r_then = **then**
r_else = **else**
r_endif = **endif**
r_while = **while**
r_do = **do**
r_endwhile = **endwhile**
r_call = **call**
r_record = **record**
r_array = **array**
r_of = **of**
r_pointer = **pointer**
r_new = **new**
r_read = **read**
r_write = **write**
r_nl = **nl**
r_var = **var**
r_type = **type**
r_del = **del**

id = **Identificadores** nombre de una variable. Comienzan necesariamente por una letra, seguida de una secuencia de cero o más letras, dígitos, o subrayado (_)

lit_int = **Literales enteros** literal numérico entero. Comienzan, opcionalmente, con un signo + o -. Seguidamente debe aparecer una secuencia de 1 o más dígitos (no se admiten ceros no significativos a la izquierda).

lit_real = **Literales reales** literal numérico real. Comienzan, obligatoriamente, con una parte entera, cuya estructura es como la de los números enteros, seguida de bien una parte decimal, bien una parte exponencial, o bien una parte decimal seguida de una parte exponencial. La parte decimal comienza con un ., seguido de una secuencia de 1 o más dígitos (no se permite la aparición de ceros no significativos a la derecha). Por último, y también opcionalmente, puede aparecer una parte exponencial (e o E, seguida de un exponente, cuya estructura es igual que la de los números enteros).

lit_cad = **Literales cadena** literal cadena. Comienzan con comilla doble ("), seguida de una secuencia de 0 o más caracteres distintos de ", retroceso (\b), retorno de carro (\r), y salto de línea (\n), seguida de ".

II: Gramática

PROG → LDECS **sep_prog** LINST;
PROG → LINST;
LDECS → LDECS **pto_coma** DEC;
LDECS → DEC ;

DEC → **r_var** TIPO **id** ;
DEC → **r_type** TIPO **id** ;
DEC → **r_proc id pap** LPF **pcierre** BLOQ ;

LPF → LPF **coma** PF ;
LPF → PF ;
PF → TIPO **id** ;
PF → TIPO **et id** ;

TIPO → **r_int**
TIPO → **r_real**
TIPO → **r_bool**
TIPO → **r_string** ;
TIPO → **id** ;
TIPO → **r_array cap lit_int ccierre r_of** TIPO ;
TIPO → **r_record llap** LCAMPO **llcierre** ;
TIPO → **r_pointer** TIPO ;

LCAMPO → LCAMPO **pto_coma** CAMPO ;
LCAMPO → CAMPO ;
CAMPO → TIPO **id** ;

LINST → LINST **pto_coma** INST ;
LINST → INST ;

INST → E0 **asig** E0 ;
INST → **r_if** E0 **r_then** AUX_LINST **r_endif** ;
INST → **r_if** E0 **r_then** AUX_LINST **r_else** AUX_LINST **r_endif** ;
INST → **r_while** E0 **r_do** AUX_LINST **r_endwhile** ;
AUX_LINST → LINST
AUX_LINST → λ ;

INST → **r_read** E0 ;
INST → **r_write** E0 ;
INST → **r_nl** ;
INST → **r_new** E0 ;
INST → **r_delete** E0 ;
INST → **r_call id pap** LPARREG **pcierre** ;
LPARREG → E0 **coma** LPARREG ;
LPARREG → E0 ;

INST → BLOQ ;

BLOQ → **llap** PROG **llcierre** ;

E0 → E1 **mas** E0 ;
E0 → E1 **menos** E1 ;
E0 → E1 ;

E1 → E1 OPBAI1 E2 ;
E1 → E2 ;

E2 → E2 OPBAI2 E3 ;
E2 → E3 ;

E3 → E4 OPBNA E4 ;
E3 → E4 ;

E4 → **menos** E5 ;
E4 → **r_not** E4 ;
E4 → E5 ;

E5 → E5 **cap** E0 **ccierre** ;
E5 → E5 **flecha** id;
E5 → E5 **punto** id;

E5 → E6 ;

E6 → **por** E6 ;
E6 → E7 ;

E7 → **pap** E0 **pcierre**;
E7 → **null**;
E7 → **id**;
E7 → **false**;
E7 → **true**;
E7 → **lit_cad**;
E7 → **lit_real**;
E7 → **lit_int**;

OPBAI1 → **r_and** | **r_or** ;
OPBAI2 → **blt** | **bgt** | **ble**| **bge** | **beq** | **bne** ;
OPBNA → **por** | **div** | **mod** ;

2. SINTAXIS ABSTRACTA

2.1 Géneros o conceptos sintácticos

Programa, Linst, Ldecs, Dec, Tipo, LPF, ParamF, LCampo, Campo, Inst, AuxLinst, AuxInst, Exp, LParReg, ParReg.

2.2 Constructoras

prog_vacio: Programa
prog_sin_dec: Linst → Programa
prog_con_dec: Ldecs x Linst → Programa

lista_dec_una: Dec → Ldecs
lista_dec_muchas: Ldecs x Dec → Ldecs

dec_var: Tipo x String → Dec
dec_tipo: Tipo x String → Dec
dec_proc: String x LPF x Bloque→ Dec

param_vacio: LPF
param_una: ParamF → LPF
param_muchas: LPF x ParamF → LPF
param_con_et: Tipo x String → ParamF
param_sin_et: Tipo x String → ParamF

tipo_int: Tipo
tipo_real: Tipo
tipo_bool: Tipo
tipo_string: Tipo
tipo_id: String → Tipo
tipo_array: String x Tipo → Tipo
tipo_pointer: Tipo → Tipo
tipo_record: LCampo → Tipo

campo_una: Campo → LCampo
campo_muchas: LCampo x Campo → LCampo
campo: Tipo x String → Campo

lista_inst_una: Inst → Linst
lista_inst_muchas: Linst x Inst → Linst

aux_inst_vacio: AuxLinst
aux_inst_una: AuxInst → Linst

inst_asig: Exp x Exp → Inst
inst_if_then: Exp x AuxLinst → Inst

inst_if_then_else:	Exp x AuxLinst xAuxLinst → Inst
inst_while:	Exp x AuxLinst → Inst
inst_read:	Exp → Inst
inst_write:	Exp → Inst
inst_nl:	Inst
inst_new:	Exp → Inst
inst_call:	String x LParReg → Inst
inst_compuesta:	Bloque → Inst
par_reg_vacio:	LParReg
par_reg_una:	ParReg → LParReg
par_reg_muchas:	LParReg x ParReg → LParReg
bloque:	Programa → Bloque
bloque_vacio:	
suma:	Exp x Exp → Exp
resta:	Exp x Exp → Exp
mul:	Exp x Exp → Exp
div:	Exp x Exp → Exp
mod:	Exp x Exp → Exp
beq:	Exp x Exp → Exp
bne:	Exp x Exp → Exp
ble:	Exp x Exp → Exp
bge:	Exp x Exp → Exp
blt:	Exp x Exp → Exp
bgt:	Exp x Exp → Exp
c_and:	Exp x Exp → Exp
c_or:	Exp x Exp → Exp
c_not:	Exp → Exp
menos_unario:	Exp→ Exp
num_real:	String → Exp
num_int:	String → Exp
identificador:	String → Exp
lit_cad:	String → Exp
c_false:	Exp
c_true:	Exp
c_null:	Exp
c_str:	String → Exp
index:	Exp x Exp → Exp
access:	Exp x String → Exp
pointer:	Exp x String → Exp
indir:	Exp → Exp

3. CONSTRUCTOR DE ÁRBOLES DE SINTAXIS ABSTRACTA

PROG := LDECS **sep_prog** LINST;
PROG.a = prog_con_dec(LDECS.a, LINST.a)
PROG := LINST;
PROG.a = prog_sin_dec(LINST.a)
LDECS := LDECS **pto_coma** DEC;
$LDECS_0$.a = lista_dec_muchas ($LDECS_1$.a, DEC.a)
LDECS := DEC;
LDECS.a = lista_dec_una(DEC.a)
DEC := **var** TIPO **id**;
DEC.a = dec_var(TIPO.a, id.lex)
DEC := **type** TIPO **id**;
DEC.a = dec_type(TIPO.a, id.lex)
DEC := **proc id pap** LPF **pcierre** BLOQ;
DEC.a = dec_proc_con_params(id.lex, LPF.a, BLOQ.a)
DEC := **proc id pap pcierre** BLOQ;
DEC.a = dec_proc_sin_params(id.lex, BLOQ.a)
LPF := LPF **coma** PF;
LPF_0.a = param_muchas(LPF_1.a, PF.a)
LPF := PF;
LPF.a = param_una(PF.a)
PF := TIPO **et id**;
PF.a = param_con_et(TIPO.a, id.lex)
PF := TIPO **id**;
PF.a = param_sin_et(TIPO.a, id.lex)

TIPO := **r_int**;
TIPO.a = tipo_Entero()
TIPO := **r_bool**;
TIPO.a = tipo_Bool()
TIPO := **r_real**;
TIPO.a = tipo_Real()
TIPO := **r_string**;
TIPO.a = tipo_String()
TIPO := **id**;
TIPO.a = tipo_Id(id.lex)
TIPO := **r_array cap lit_int ccierre r_of** TIPO ;
$TIPO_0$.a = tipo_Array($TIPO_1$.a)
TIPO := **r_record llap** LCAMPO **llcierre** ;
TIPO.a = tipo_Record(LCAMPO.a)
TIPO := **r_pointer** TIPO;
TIPO.a = tipo_Pointer($TIPO_1$.a)

LCAMPO := LCAMPO **pto_coma** CAMPO;
$LCAMPO_0$.a = campo_muchas($LCAMPO_1$.a, CAMPO.a)
LCAMPO := CAMPO;

LCAMPO.a = campo_una(CAMPO.a)

CAMPO := TIPO **id**;

CAMPO.a = campo(TIPO.a, id.lex)

LINST := LINST **pto_coma** INST;

$LINST_0$.a = lista_inst_muchas($LINST_1$.a, INST.a)

LINST := INST;

$LINST_0$.a = lista_inst_una(INST.a)

INST := E0 **asig** E0;

INST.a = inst_asig($E0_0$.a, $E0_1$.a)

INST := **if** E0 **then** AUX_LINST **endif**;

INST.a = inst_if_then(E0.a, AUX_LINST.a)

INST := **if** E0 **then** AUX_LINST **else** AUX_LINST **endif**;

INST.a = inst_if_then_else(E0.a, AUX_LINST_0.a, AUX_LINST_1.a)

INST := **while** E0 **do** AUX_LINST **endwhile**;

INST.a = inst_while(E0.a, AUX_LINST.a)

AUX_LINST := LINST;

AUX_LINST.a = aux_linst_una(LINST.a)

AUX_LINST := **ƛ**;

AUX_LINST.a = aux_linst_vacia()

INST := **read** E0;

INST.a = inst_read(E0.a)

INST := **write** E0;

INST.a = inst_write(E0.a)

INST := **nl**;

INST.a = inst_nl()

INST := **new** E0;

INST.a = inst_new(E0.a)

INST := **delete** E0;

INST.a = inst_delete(E0.a)

INST := **call id pap** LPARREG **pcierre**;

INST.a = inst_call_con_params(id.lex, LPARREG.a)

INST := **call id pap pcierre**;

INST.a = inst_call_sin_params(id.lex)

LPARREG := E0 **coma** LPARREG;

$LPARREG_0$.a = par_reg_muchas(E0.a, $LPARREG_1$.a)

LPARREG := E0;

LPARREG.a = par_reg_una(E0.a)

INST := BLOQ;

INST.a = inst_compuesta(BLOQ.a)

BLOQ := **llap** PROG **llcierre**;

BLOQ.a =bloque(PROG.a)

E0 := E1 **mas** E0;

$E0_0$.a = exp("+", E1.a, $E0_1$.a)

E0 := E1 **menos** E1;

E0.a = exp("-", $E1_0$.a, $E1_1$.a)

E0 := E1;
E0.a = E1.a
E1 := E1 OPBAI1 E2;
$E1_0$.a =exp(OPBAI1.op, $E1_1$.a, E2.a)
E1 := E2;
E1.a = E2.a
E2 := E2 OPBAI2 E3;
$E2_0$.a = exp(OPBAI2.op, $E2_1$.a, E3.a)
E2 := E3;
E2.a = E3.a
E3 := E4 OPBNA E4;
E3.a = exp(OPBNA.op, $E4_0$.a, $E4_1$.a)
E3 := E4;
E3.a = E4.a
E4 := **menos** E5;
E4.a = menos_unario(E5.a)
E4 := **not** E4;
$E4_0$.a = c_not($E4_1$.a)
E4 := E5;
E4.a = E5.a
E5 := E5 **cap** E0 **ccierre** ;
$E5_0$.a = index($E5_1$.a, E0.a)
E5 := E5 **flecha** id;
$E5_0$.a = flecha($E5_1$.a, id.lex)
E5 := E5 **punto** id;
$E5_0$.a = punto($E5_1$.a,id.lex)
E5 := E6;
E5.a = E6.a
E6 := **por** E6
$E6_0$.a = indireccion($E6_1$.a)
E6 := E7;
E6.a = E7.a
E7 := **pap** E0 **pcierre**;
E7.a = E0.a
E7 := **null**;
E7.a = c_null()
E7 := **id**;
E7.a = identificador(id.lex)
E7 := **false**;
E7.a = c_false()
E7 := **true**;
E7.a = c_true()
E7 := **lit_cad**;
E7.a = c_str(lit_cad.lex)
E7 := **lit_real**;
E7.a = num_real(real.lex)
E7 := **lit_int**;
E7.a = num_int(ent.lex)

```
OPBNA := por;
        OPBNA.op = "*"
OPBNA := div;
        OPBNA.op = "/"
OPBNA := mod;
        OPBNA.op = "%"
OPBAI2 := bne;
        OPBAI2.op = "!="
OPBAI2 := beq;
        OPBAI2.op = "=="
OPBAI2 := ble;
        OPBAI2.op = "<="
OPBAI2 := bge;
        OPBAI2.op = ">="
OPBAI2 := blt;
        OPBAI2.op = "<"
OPBAI2 := bgt;
        OPBAI2.op = ">"
OPBAI1 := and;
        OPBAI1.op = "and"
OPBAI1 := or;
        OPBAI1.op = "or"
```

Funciones semánticas

```
fun prog(LDECS.a, LINST.a) {
        if LDECS.a != null
                return prog_con_decs(LDECS.a, LINST.a)
        else
                return prog_sin_decs(LINST.a)
}

fun exp(Op,Arg0,Arg1) {
        switch (Op)
                case '+':       return suma(Arg0,Arg1)
                case '-':       return resta(Arg0,Arg1)
                case '*':       return mul(Arg0,Arg1)
                case '/':       return div(Arg0,Arg1)
                case '%':       return mod(Arg0,Arg1)
                case '==':      return beq(Arg0,Arg1)
                case '<=':      return ble(Arg0,Arg1)
                case '>=':      return bge(Arg0,Arg1)
                case '!=':      return bne(Arg0,Arg1)
                case '<':       return blt(Arg0,Arg1)
                case '>':       return bgt(Arg0,Arg1)
```

```
            case 'and':     return and(Arg0,Arg1)
            case 'or':      return or(Arg0,Arg1)
}
```