Desarrollo de constructores de ASTs para Tiny(0)

Óscar Morujo Fernández

Sofía Capmany Fernández

Especificación sintáctica:

```
PROGRAMA := LDECS sep_di LINST;
                                          E4 := menos E5;
LDECS := LDECS semicolon DEC;
                                          E4 := not E4;
LDECS := DEC;
                                          E4 := E5;
DEC := NOMBRE_TIPO id;
NOMBRE_TIPO := r_real;
                                          E5 := ent;
NOMBRE_TIPO :=r_int;
                                          E5 := real;
NOMBRE TIPO :=r_bool;
                                          E5 := id;
LINST := LINST semicolon INST;
                                          E5 := true;
LINST := INST;
INST := id igual E0;
                                          E5 := false;
                                          E5 := pap E0 pcierre;
E0 := E1 mas E0;
E0 := E1 menos E1;
                                          OP3NA := por;
                                          OP3NA := div;
E0 := E1;
E1 := E1 OP1AI E2;
                                          OP2AI := bne;
E1 := E2;
                                          OP2AI := beq;
                                          OP2AI := ble;
E2 := E2 OP2AI E3;
E2 := E3;
                                          OP2AI := bge;
                                          OP2AI := blt;
E3 := E4 OP3NA E4;
                                          OP2Al :=bgt;
E3 := E4;
                                          OP1AI := and;
                                          OP1AI := or;
```

Sintaxis abstracta:

Géneros o conceptos sintácticos:

Prog, Ldecs, Dec,Linst,Inst, Tipo y Exp.

Constructoras:

programa : Ldecs x Linst -> Prog

lista_dec_una : Dec -> Ldecs

lista_dec_muchas : Ldecs x Dec -> Ldecs

tipo_Entero: ->Tipo
tipo_Real: ->Tipo
tipo_Bool: ->Tipo

dec: Tipo x String -> Dec

lista_inst_una: Inst -> Linst

lista_inst_muchas: Linst x Inst -> Linst inst: String x Exp -> Inst suma: Exp x Exp -> Exp Exp x Exp -> Exp resta: mul: Exp x Exp -> Exp div: 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 Exp x Exp -> Exp bgt: and: Exp x Exp -> Exp or: Exp x Exp -> Exp

not:Exp -> Expmenos_unario:Exp-> Expnum_real:String -> Expnum_ent:String -> Expidentificador:String -> Exp

r_false: -> Exp **r_true**: -> Exp

Constructor de árboles de sintaxis abstracta(ASTs):

```
PROGRAMA := LDECS sep_di LINST;
     PROGRAMA.a = prog(LDECS.a, LINST.a)
LDECS := LDECS semicolon DEC;
     LDECS __a = lista_dec_muchas (LDECS __a, a, DEC.a)
LDECS := DEC;
     LDECS. a= lista_dec_una(DEC.a)
DEC := NOMBRE_TIPO id;
     DEC. a = dec(NOMBRE\_TIPO.a,id.lex)
NOMBRE_TIPO := r_int;
     NOMBRE\_TIPO . a = tipo\_Entero()
NOMBRE_TIPO := r_bool;
     NOMBRE\_TIPO. a = tipo\_Bool()
NOMBRE_TIPO := r_real;
     NOMBRE\_TIPO . a = tipo\_Real()
LINST := LINST semicolon INST;
     LINST _{0}. a =lista_inst_muchas (LINST _{1}. a, INST.a)
LINST := INST;
     LINST. a =lista_inst_una (INST.a)
INST := id igual E0;
    INST. a = inst(id.lex, E0.a)
E0 := E1 \text{ mas } E0;
     E0_{0}. a = \exp_{0} sinaria("+",E1. a, E0_{1}. a)
E0 := E1 menos E1;
     E0. a = \exp_{\text{binaria}}(\text{"-"}, E1_{0}. a, E1_{1}. a)
E0 := E1:
    E0.a = E1.a
E1 := E1 OP1AI E2;
    E1_{0}. a = \exp_{\text{binaria}}(\text{OP1AI.op}, E1_{1}, a, E2. a)
E1 := E2;
    E1. a = E2. a
E2 := E2 OP2AI E3;
    E2_0. a = \exp_binaria(OP2AI.op, E2_1.a, E3.a)
E2 := E3;
    E2. a = E3. a
E3 := E4 OP3NA E4;
    E3. a = \exp_{\text{o}} \cdot a, e^{-1} \cdot a, e^{-1} \cdot a
E3 := E4;
    E3. a = E4. a
E4 := menos E5;
```

```
E4. a = menos_unario(E5.a)
E4 := not E4;
    E4 \quad a = not(E4 \quad a)
E4 := E5;
    E4. a = E5. a
E5 := ent;
    E5. a = num(ent.lex)
E5 := real;
    E5. a = num(real.lex)
E5 := id;
    E5. \alpha = identificador(id.lex)
E5 := true;
    E5. a = r_true()
E5 := false;
    E5. a = r false()
E5 := pap E0 pcierre;
    E5. a = E0. a
OP3NA := por;
    OP3NA.op = "*"
OP3NA := div;
    OP3NA.op = "/"
OP2AI := bne;
    OP2AI.op = "! = "
OP2AI := beq;
    OP2AI.op = " == "
OP2AI := ble;
    OP2AI.op = " <= "
OP2AI := bge;
    OP2AI.op = ">= "
OP2AI := blt;
    OP2AI.op = " < "
OP2AI :=bgt;
    OP2AI.op = " > "
OP1Al := and;
    OP1AI.op = "and"
OP1AI := or;
    OP1AI.op = "or"
```

Funciones semánticas:

```
    fun exp_binaria(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 beq(Arg0,Arg1)
               case '<=':
                             return ble(Arg0,Arg1)
                             return bge(Arg0,Arg1)
              case '>=':
                             return bne(Arg0,Arg1)
              case '!=':
               case '<':
                             return blt(Arg0,Arg1)
              case '>':
                             return bgt(Arg0,Arg1)
              case 'and':
                             return and(Arg0,Arg1)
              case 'or':
                             return or(Arg0,Arg1)
       }
}
```

Acondicionamiento para implementación descendente:

Primer paso: Factorizar.

```
E0 := E1 mas E0;
                                                     E0 := E1 RES0;
E0_{0}. a = \exp_{0} ("+", E1. a, E0_{1}. a)
                                                     RES0.ah = E1.a
                                                     E0. a = RES0. a
E0 := E1 menos E1;
                                                    RES0 := mas E0;
E0. a = \exp_{\text{binaria}}(\text{"-"}, E1_{0}, a, E1_{1}, a)
                                                     RES0. a = \exp\_binaria("+", RES0. ah, E0. a)
E0 := E1;
                                                    RES0 := menos E1;
E0.a = E1.a
                                                     RES0. a = \exp\_binaria("-", RES0. ah, E1. a)
                                                     RES0 := \bar{\lambda};
                                                    RES0.a = RES0.ah
E3 := E4 OP3NA E4;
                                                    E3 := E4 RES3;
E3. a = \exp_{\text{binaria}}(\text{OP3NA.op}, E4_{0}. a, E4_{1}. a)
                                                    RE3.ah = E4.a
                                                     E3.a = RES3.a
                                                    RES3 := OP3NA E4;
E3 := E4;
                                                    RES3. a = \exp_binaria(OP3NA.op, RES3. ah, E4. a)
E3.a = E4.a
                                                    RES3 := \bar{\lambda};
                                                     RES3.a = RES3.ah
```

Segundo paso: Eliminar recursión a izquierdas

```
LDECS := LDECS semicolon DEC;
                                             LDECS := DEC RLDECS;
LDECS . a = lista_dec_muchas (LDECS . a,
                                             RLDECS. ah =lista_dec_una(DEC.a)
                                             LDECS.a = RLDECS.a
DEC.a)
                                             RLDECS := semicolon DEC RLDECS;
LDECS := DEC;
                                             RLDECS __. ah=lista_dec_muchas(
LDECS. a= lista_dec_una(DEC.a)
                                             RLDECS o. ah, DEC. a)
                                             RLDECS_{0}, a = RLDECS_{1}, a
                                             RLDECS := X:
                                             RLDECS. a = RLDECS. ah
LINST := LINST semicolon INST;
                                             LINST := INST RLINST;
LINST o. a = lista_inst_muchas (LINST o. a,
                                             RLINST. ah = lista inst una(INST.a)
                                             LINST.a = RLINST.a
INST.a)
                                             RLINST := semicolon INST RLINST;
LINST := INST;
                                             RLINST_{1}. ah = lista_inst_muchas(
LINST. a =lista_inst_una (INST.a)
                                             RLINST <sub>o</sub>. ah, INST. a)
                                             RLINST_0.a = RLINST_1.a
                                             RLINST := λ̄;
                                             RLINST.a = RLINST.ah
                                             E1 := E2 RES1;
E1 := E1 OP1AI E2;
E1_{0}. a = \exp_{\text{binaria}}(\text{OP1AI.op}, E1_{1}.a,
                                             RES1.ah = E2.a
                                             E1.a = RES1.a
E2.a)
                                             RES1 := OP1AI E2 RES1;
E1 := E2;
                                             RES1 _1. ah=exp_binaria(OP1AI.op,
E1. a = E2. a
                                             RES1 _{0}. ah, E2. a)
                                             RES1 := λ̄;
                                             RES1.a = RES1.ah
E2 := E2 OP2AI E3;
                                             E2 := E3 RES2;
E2 \alpha = exp_binaria(OP2Al.op,E2 \alpha,
                                             RES2. ah = E3. a
                                             E2.a = RES2.a
E3. a)
                                             RES2 := OP2AI E3 RES2;
E2 := E3;
                                             RES2_1. ah = \exp_binaria(
E2.a = E3.a
                                             OP2AI. op, RES2<sub>0</sub>. ah, E3. a)
                                             RES2_0. a = RES2_1. a
```

RES2 := \hbar ; RES2. a = RES2. ah

Gramática transformada:

Reglas que son producto de factorizar

Reglas que son producto de eliminar recursión a izquierdas

```
PROGRAMA := LDECS sep di LINST;
                                                 E2 := E3 RES2;
      PROGRAMA.a = prog(LDECS.a, LINST.a)
                                                       RES2.ah = E3.a
                                                       E2.a = RES2.a
LDECS := DEC RLDECS;
                                                 RES2 := OP2AI E3 RES2;
      RLDECS.ah =lista_dec_una(DEC.a)
                                                       RES2_1. ah = \exp_binaria(
      LDECS.a = RLDECS.a
                                                 OP2AI. op, RES2<sub>0</sub>. ah, E3. a)
RLDECS := semicolon DEC RLDECS:
                                                       RES2_0. a = RES2_1. a
      RES2 := λ:
RLDECS _{0}. ah, DEC. a)
                                                       RES2. a = RES2. ah
      RLDECS _{0}. a = RLDECS _{1}. a
RLDECS := λ:
                                                 E3 := E4 RES3;
      RLDECS, a = RLDECS, ah
                                                       RE3.ah = E4.a
                                                       E3.a = RES3.a
DEC := NOMBRE_TIPO id;
                                                 RES3 := OP3NA E4;
      DEC. a = dec(NOMBRE_TIPO.a,id.lex)
                                                       RES3. a = \exp binaria(OP3NA.op,
NOMBRE_TIPO := r_int;
                                                 RES3. ah,E4. a)
      NOMBRE_TIPO . a=tipo Entero()
                                                 RES3 := \lambda;
NOMBRE_TIPO := r_bool;
                                                       RES3. a = RES3. ah
      NOMBRE\_TIPO. a = tipo\_Bool()
  NOMBRE TIPO := r_real;
                                                 E4 := menos E5:
      NOMBRE\_TIPO . a = tipo Real()
                                                       E4. a = menos_unario(E5.a)
                                                 E4 := not E4;
LINST := INST RLINST;
                                                       E4_{0}.a = not(E4_{1}.a)
      RLINST. ah = lista inst una(INST.a)
                                                 E4 := E5;
      LINST.a = RLINST.a
                                                       E4. a = E5. a
RLINST := semicolon INST RLINST;
                                                 E5 := ent;
      RLINST_{1}. ah = lista_inst_muchas(
                                                       E5. a = num(ent.lex)
RLINST _{0}. ah, INST. a)
                                                 E5 := real;
                                                       E5. a = num(real.lex)
      RLINST_{0}.a = RLINST_{1}.a
                                                 E5 := id:
RLINST := \lambda;
                                                       E5. a = identificador(id.lex)
```

```
RLINST.a = RLINST.ah
                                                    E5 := true;
                                                           E5. a = r_true()
INST := id igual E0;
                                                    E5 := false;
      INST. a = inst(id.lex, E0.a)
                                                           E5. a = r_false()
                                                    E5 := pap E0 pcierre;
                                                           E5. a = E0. a
E0 := E1 RES0;
      RES0. ah = E1. a
                                                      OP3NA := por;
                                                           OP3NA.op = "*"
      E0. a = RES0. a
RES0 := mas E0;
                                                      OP3NA := div;
      RES0. a = \exp_{\text{binaria}}("+", RES0. ah, E0. a)
                                                           OP3NA.op = "/"
RES0 := menos E1;
                                                      OP2Al := bne;
      RES0. \alpha = exp_binaria("-",RES0. \alpha h, E1. \alpha)
                                                           OP2AI.op = "! = "
RES0 := \tilde{\lambda};
                                                      OP2AI := beq;
                                                           OP2AI.op = " == "
      RES0. a = RES0. ah
                                                      OP2AI := ble;
E1 := E2 RES1;
                                                           OP2AI.op = " <= "
      RES1. ah = E2. a
                                                      OP2AI := bge;
                                                           OP2AI.op = ">= "
      E1.a = RES1.a
RES1 := OP1AI E2 RES1;
                                                      OP2AI := blt;
      RES1 _. ah=exp_binaria(OP1AI.op,
                                                           OP2AI.op = " < "
                                                      OP2AI :=bgt;
RES1 _{0}. ah, E2. a)
                                                           OP2AI.op = " > "
RES1 := \lambda;
                                                      OP1AI := and;
       RES1. a = RES1. ah
                                                           OP1AI.op = "and"
                                                      OP1AI := or;
                                                           OP1AI.op = "or"
```