Desarrollo de constructores de ASTs para Tiny(0)

Óscar Morujo Fernández

Sofía Capmany Fernández

G16

Especificación sintáctica:

PROGRAMA := LDECS **sep_di** LINST;
LDECS := LDECS **semicolon** DEC;
LDECS := DEC;
DEC := NOMBRE_TIPO **id**;
NOMBRE_TIPO := **r_real**;
NOMBRE_TIPO :=**r_int;**
NOMBRE_TIPO :=**r_bool;**
LINST := LINST **semicolon** INST;
LINST := INST;
INST := **id igual** E0;

E0 := E1 **mas** E0;
E0 := E1 **menos** E1;
E0 := E1;

E1 := E1 OP1AI E2;
E1 := E2;

E2 := E2 OP2AI E3;
E2 := E3;

E3 := E4 OP3NA E4;
E3 := E4;

E4 := **menos** E5;
E4 := **not** E4;
E4 := E5;

E5 := **ent**;
E5 := **real**;
E5 := **id**;
E5 := **true**;

E5 := **false**;
E5 := **pap** E0 **pcierre**;

OP3NA := **por**;
OP3NA := **div**;

OP2AI := **bne**;
OP2AI := **beq**;

OP2AI := **ble**;
OP2AI := **bge**;
OP2AI := **blt**;
OP2AI :=**bgt**;
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
resta:	Exp x Exp -> Exp
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
bgt:	Exp x Exp -> Exp
and:	Exp x Exp -> Exp
or:	Exp x Exp -> Exp
not:	Exp -> Exp
menos_unario:	Exp-> Exp
num_real:	String -> Exp
num_ent:	String -> Exp
identificador:	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;

$LDECS_0.a$ =lista_dec_muchas ($LDECS_1.a$, 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 **mas** E0;

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

E0 := E1 **menos** E1;

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

E0 := E1;

$E0.a = E1.a$

E1 := E1 OP1AI E2;

$E1_0.a$ =exp_binaria(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_binaria(OP3NA.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;

$E4_0.a$ = not($E4_1.a$)

E4 := E5;

$E4.a = E5.a$

E5 := **ent**;

$E5.a$ = num(ent.lex)

E5 := **real**;

$E5.a$ = num(real.lex)

E5 := **id**;

$E5.a$ = 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**;

$OP2AI.op = " > "$

OP1AI := **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)
                    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)
            }
    }
```

Acondicionamiento para implementación descendente:

Primer paso: Factorizar.

E0 := E1 **mas** E0;
$E0_0.a$ = exp_binaria(“+”,$E1.a$, $E0_1.a$)
E0 := E1 **menos** E1;
$E0.a$ = exp_binaria(“-”,$E1_0.a$, $E1_1.a$)
E0 := E1;
$E0.a = E1.a$
	E0 := E1 RES0;
	$RES0.ah = E1.a$
	$E0.a = RES0.a$
	RES0 := **mas** E0;
	$RES0.a$ = exp_binaria(“+”,$RES0.ah$, $E0.a$)
	RES0 := **menos** E1;
	$RES0.a$ = exp_binaria(“-”,$RES0.ah$, $E1.a$)
	RES0 := **ƛ**;
	$RES0.a = RES0.ah$

E3 := E4 OP3NA E4;
$E3.a$ =exp_binaria(OP3NA.op,$E4_0.a$, $E4_1.a$)
E3 := E4;
$E3.a = E4.a$
	E3 := E4 RES3;
	$RE3.ah = E4.a$
	$E3.a = RES3.a$
	RES3 := OP3NA E4;
	$RES3.a$ =exp_binaria(OP3NA.op,$RES3.ah$,$E4.a$)
	RES3 := **ƛ**;
	$RES3.a = RES3.ah$

Segundo paso : Eliminar recursión a izquierdas

LDECS := LDECS **semicolon** DEC;
$LDECS_{0}.a$ =lista_dec_muchas ($LDECS_{1}.a,$ DEC.a)
LDECS := DEC;
$LDECS.a$= lista_dec_una(DEC.a)
	LDECS := DEC RLDECS;
	$RLDECS.ah$ =lista_dec_una(DEC.a)
	$LDECS.a = RLDECS.a$
	RLDECS := **semicolon** DEC RLDECS;
	$RLDECS_{1}.ah$=lista_dec_muchas($RLDECS_{0}.ah, DEC.a)$
	$RLDECS_{0}.a = RLDECS_{1}.a$
	RLDECS := **ƛ**;
	$RLDECS.a$= $RLDECS.ah$

LINST := LINST **semicolon** INST;
$LINST_{0}.a$ =lista_inst_muchas ($LINST_{1}.a,$ INST.a)
LINST := INST;
$LINST.a$ =lista_inst_una (INST.a)
	LINST := INST RLINST;
	$RLINST.ah$ =lista_inst_una(INST.a)
	$LINST.a = RLINST.a$
	RLINST := **semicolon** INST RLINST;
	$RLINST_{1}.ah =$ lista_inst_muchas($RLINST_{0}.ah, INST.a)$
	$RLINST_{0}.a = RLINST_{1}.a$
	RLINST := **ƛ**;
	$RLINST.a = RLINST.ah$

E1 := E1 OP1AI E2;
$E1_{0}.a$ =exp_binaria(OP1AI.op,$E1_{1}.a,$ $E2.a)$
E1 := E2;
$E1.a = E2.a$
	E1 := E2 RES1;
	$RES1.ah = E2.a$
	$E1.a = RES1.a$
	RES1 := OP1AI E2 RES1;
	$RES1_{1}.ah$=exp_binaria(OP1AI.op, $RES1_{0}.ah, E2.a)$
	RES1 := **ƛ**;
	$RES1.a = RES1.ah$

E2 := E2 OP2AI E3;
$E2_{0}.a$ =exp_binaria(OP2AI.op,$E2_{1}.a,$ $E3.a)$
E2 := E3;
$E2.a = E3.a$
	E2 := E3 RES2;
	$RES2.ah = E3.a$
	$E2.a = RES2.a$
	RES2 := OP2AI E3 RES2;
	$RES2_{1}.ah$ =exp_binaria($OP2AI.op, RES2_{0}.ah, E3.a)$
	$RES2_{0}.a = RES2_{1}.a$

	RES2 := ƛ; $RES2.a = RES2.ah$

Gramática transformada:

Reglas que son producto de factorizar

Reglas que son producto de eliminar recursión a izquierdas

PROGRAMA := LDECS **sep_di** LINST; PROGRAMA.a = prog(LDECS.a, LINST.a) LDECS := DEC RLDECS; $RLDECS.ah$ =lista_dec_una(DEC.a) $LDECS.a = RLDECS.a$ RLDECS := **semicolon** DEC RLDECS; $RLDECS_1.ah$=lista_dec_muchas($RLDECS_0.ah, DEC.a$) $RLDECS_0.a = RLDECS_1.a$ RLDECS := ƛ; $RLDECS.a$= $RLDECS.ah$ 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 := INST RLINST; $RLINST.ah$ =lista_inst_una(INST.a) $LINST.a = RLINST.a$ RLINST := **semicolon** INST RLINST; $RLINST_1.ah$ = lista_inst_muchas($RLINST_0.ah, INST.a$) $RLINST_0.a = RLINST_1.a$ RLINST := ƛ;	E2 := E3 RES2; $RES2.ah = E3.a$ $E2.a = RES2.a$ RES2 := OP2AI E3 RES2; $RES2_1.ah$ =exp_binaria($OP2AI.op, RES2_0.ah, E3.a$) $RES2_0.a = RES2_1.a$ RES2 := ƛ; $RES2.a = RES2.ah$ E3 := E4 RES3; $RES3.ah = E4.a$ $E3.a = RES3.a$ RES3 := OP3NA E4; $RES3.a$ =exp_binaria(OP3NA.op, $RES3.ah$,$E4.a$) RES3 := ƛ; $RES3.a = RES3.ah$ E4 := **menos** E5; $E4.a$ = menos_unario(E5.a) E4 := **not** E4; $E4_0.a$ = not($E4_1.a$) E4 := E5; $E4.a = E5.a$ E5 := **ent**; $E5.a$ = num(ent.lex) E5 := **real**; $E5.a$ = num(real.lex) E5 := **id**; $E5.a$ = identificador(id.lex)

$RLINST.a \ = \ RLINST.ah$

INST := **id igual** E0;

$INST.a$ = inst(id.lex,E0.a)

E0 := E1 RES0;

$RES0.ah \ = \ E1.a$

$E0.a \ = RES0.a$

RES0 := **mas** E0;

$RES0.a$ = exp_binaria(“+”,$RES0.ah, E0.a$)

RES0 := **menos** E1;

$RES0.a$ = exp_binaria(“-”,$RES0.ah, E1.a$)

RES0 := ƛ;

$RES0.a \ = \ RES0.ah$

E1 := E2 RES1;

$RES1.ah \ = \ E2.a$

$E1.a \ = RES1.a$

RES1 := OP1AI E2 RES1;

$RES1_1.ah$=exp_binaria(OP1AI.op, $RES1_0.ah, E2.a$)

RES1 := ƛ;

$RES1.a \ = \ RES1.ah$

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**;

$OP2AI.op \ = \ " > "$

OP1AI := **and**;

$OP1AI.op \ = \ "and"$

OP1AI := **or**;

$OP1AI.op \ = \ "or"$