TERCERA FASE: CONSTRUCTOR DE ASTs PARA TINY(0)

Javier Gómez Moraleda
Mario Quiñones Pérez
Grupo 18

1. Especificación de la sintaxis abstracta de Tiny(0) mediante la enumeración de las signaturas (cabeceras) de las funciones constructoras de ASTs.

CONSTRUCTORAS
programa: Declaraciones X Instrucciones → Programa
decs_muchas: Declaraciones X Declaracion → Declaraciones **decs_una**: Declaraciones → Declaraciones
declaracion: Tipo X String→ Declaracion
int_cons: → Tipo **real_cons**: → Tipo **bool_cons**: → Tipo
insts_muchas: Instrucciones X Instruccion → Instrucciones **insts_una**: Instruccion → Instrucciones
instruccion: String X Exp → Instruccion
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 **id**: String → Exp

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

Definición de la gramática	**Definiciones auxiliares**
PROGRAMA → DECLARACIONES && INSTRUCCIONES PROGRAMA .a = programa(DECLARACIONES.a, INSTRUCCIONES.a)	
DECLARACIONES → DECLARACIONES ; DECLARACION DECLARACIONES_0.a = decs_muchas(DECLARACIONES_1.a, DECLARACION.a) DECLARACIONES → DECLARACION DECLARACIONES.a = decs_una(DECLARACION.a)	
DECLARACION → **TIPO** ID DECLARACION.a = declaracion(TIPO.a, ID.lex)	TIPO → int TIPO.a = int_cons() TIPO → real TIPO.a = real_cons() TIPO → bool TIPO.a = bool_cons()
INSTRUCCIONES → INSTRUCCIONES ; INSTRUCCION INSTRUCCIONES_0.a = ins_muchas(INSTRUCCIONES_1.a, INSTRUCCION.a) INSTRUCCIONES → INSTRUCCION INSTRUCCIONES.a = ins_una(INSTRUCCION.a)	
INSTRUCCION → ID = E0 INSTRUCCION.a = instruccion(id.lex, E0.a)	
E0 → E1 + E0 E0_0.a = exp('+', E0_1.a, E1.a) E0 → E1 **-** E1 E0.a = exp('-', 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 = '/'
E4 → **not** E4 E4_0.a = exp("not", E4_1.a) E4 → **-** E5 E4.a = exp("-", E5.a) E4 → E5 E4.a = E5.a	
E5 → EXPRESION E5.a = EXPRESION.a E5 → (E0) E5.a = E0.a	EXPRESION → true EXPRESION.a = litTrue() EXPRESION → false EXPRESION.a = litFalse() EXPRESION → LIT_ENT EXPRESION.a = litEnt(LIT_ENT.lex) EXPRESION → LIT_REAL EXPRESION.a = litReal(LIT_REAL.lex) EXPRESION → ID EXPRESION.a = id(ID.lex)

Funciones Semánticas

Vamos a hacer una distinción entre los operadores binarios que ocupen un único carácter y los que necesiten más de uno.

```
// Operadores binarios
Exp exp(char op, Exp arg0, Exp 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 mayor(arg0, arg1);
	case '>':	return menor(arg0, arg1);
	throw new UnsupportedOperationException("exp " + op);
}

Exp exp(String op, Exp arg0, Exp arg1) {
	switch (op) {
		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);
}

// Operadores unarios
Exp exp(String op, Exp arg0) {
	switch (op) {
		case "-":
			return menosUnario(arg0);
		case "not":
			return not(arg0);
		}
		throw new UnsupportedOperationException("exp " + op);
}
```

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

Definición de la gramática	**Definiciones auxiliares**
PROGRAMA → DECLARACIONES && INSTRUCCIONES PROGRAMA .a = programa(DECLARACIONES.a, INSTRUCCIONES.a)	
DECLARACIONES → DECLARACION RDEC RDEC.h = decs_una(DECLARACION.a) DECLARACIONES.a = RDEC.a RDEC → ; DECLARACION RDEC RDEC_1.h = decs_muchas(DECLARACION.a, RDEC_0.h) RDEC_0.a = RDEC_1.a RDEC → ε RDEC.a = RDEC.h	
DECLARACION → **TIPO** ID DECLARACION.a = declaracion(TIPO.a, ID.lex)	TIPO → int TIPO.a = int_cons() TIPO → real TIPO.a = real_cons() TIPO → bool TIPO.a = bool_cons()
INSTRUCCIONES → INSTRUCCION RINS RINS.h = insts_una(INSTRUCCION.a) INSTRUCCIONES.a = RINS.a RINS → ; INSTRUCCION RINS RINS_1.h = insts_muchas(INSTRUCCION.a, RINS_0.h) RINS_0.a = RINS_1.a RINS → ε RINS.a = RINS.h	
INSTRUCCION → ID = E0 INSTRUCCION.a = instruccion(id.lex, E0.a)	
E0 → E1 RE0 RE0_1.h = E1.a E0.a = RE0.a RE0 → + E0 RE0_1.h = exp('+', RE0_0.h, E0.a) RE0_0.a = RE0_1.a RE0 → - E1 RE0.a = exp('-', RE0.h, E1.a) 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, RE1_0.h, E2.a) 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, RE2_0.h, E3.a) 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, RE3.h, E4.a) RE3 → ε RE3.a = RE3.h	OPBN3 → * OPBN3.op = '*' OPBN3 → / OPBN3.op = '/'
E4 → **not** E4 E4_0.a = exp("-", E4_1.a) E4 → **-** E5 E4.a = exp("-", E5.a) E4 → E5 E4.a = E5.a	
E5 → EXPRESION E5.a = EXPRESION.a E5 → (E0) E5.a = E0.a	EXPRESION → true EXPRESION.a = litTrue() EXPRESION → false EXPRESION.a = litFalse() EXPRESION → LIT_ENT EXPRESION.a = litEnt(LIT_ENT.lex) EXPRESION → LIT_REAL EXPRESION.a = litReal(LIT_REAL.lex) EXPRESION → ID EXPRESION.a = id(ID.lex)