PROCESADORES DE LENGUAJES

FASE 3.1:

DESARROLLO DE CONSTRUCTORES DE ASTs PARA TINY(0).

Grupo 07:
HongXiang Chen y
Andrés Teruel Fernández

1. Especificación de la sintaxis abstracta.

Constructora	Explicación
prog: Decs x Instrs → Prog	Construye un programa, dados una lista de declaraciones y una lista de instrucciones.
decs_muchas: Decs x Dec → Decs	Dada una lista de declaraciones y una declaracion, construye una lista de declaraciones.
decs_una: Dec → Decs	Dada una declaracion, construye una lista de declaraciones.
dec: Tipo x string → Dec	Dado su tipo y su indentificador, construye una declaracion de variables.
int: Tipo	Construye el tipo int
real: Tipo	Construye el tipo real
bool: Tipo	Construye el tipo bool
instr_muchas: Instrs x Instr → Instrs	Dada una lista de instrucciones y una instrucción, construye una lista de instrucciones.
instr_una : Instr → Instrs	Dada una instrucción, construye una lista de instrucciones..
instruccion: string x Expresion → Instr	Dados un id y una expresión, construye una instrucción de asignacion.
num_ent: string → Exp	Construye una expresión que representa un entero a partir de la cadena asociada a dicho literal.
num_real: string → Exp	Construye una expresión que representa un real a partir de la cadena asociada a dicho literal
id: string → Exp	Construye una expresión que representa una variable a partir del nombre de dicha variable
boolean: string → Exp	Construye una expresión que representa un booleano
suma: Exp x Exp → Exp	Construye una expresión a partir de la suma de dos expresiones
resta: Exp x Exp → Exp	Construye una expresión a partir de la resta de dos expresiones
and: Exp x Exp → Exp	Construye una expresión a partir de lógica and de dos expresiones
or: Exp x Exp → Exp	Construye una expresión a partir de lógica or de

	dos expresiones
menor: Exp x Exp → Exp	Construye una expresión a partir de la comparación menor de dos expresiones
mayor: Exp x Exp → Exp	Construye una expresión a partir de la comparación mayor de dos expresiones
menor_igual: Exp x Exp → Exp	Construye una expresión a partir de la comparación menor igual de dos expresiones
mayor_igual: Exp x Exp → Exp	Construye una expresión a partir de la comparación mayor igual de dos expresiones
diferente: Exp x Exp → Exp	Construye una expresión a partir de la comparación de diferencia de dos expresiones
igual: Exp x Exp → Exp	Construye una expresión a partir de la comparación de igualdad de dos expresiones
mult: Exp x Exp → Exp	Construye una expresión a partir de la multiplicación de dos expresiones
div: Exp x Exp → Exp	Construye una expresión a partir de la división de dos expresiones
not: Exp → Exp	Construye una expresión a partir de lógica not de una expresiones
negativo: Exp → Exp	Construye una expresión a partir del operador de negativo de una expresiones

2. Especificación del constructor de ASTs.

S → SecDec SEPARADOR SecInstr;
S.a = prog(SecDec.a, SecInsr.a).
SecDec → SecDec PUNTOCOMA Declaracion;
SecDec0.a = decs_muchas(SecDec1.a, Declaracion.a).
SecDec → Declaracion;
SecDec.a = decs_una(Declaracion.a).
Declaracion → Tipo IDENTIFICADOR;
Declaracion.a = dec(Tipo.a, IDENTIFICADOR.lex).
Tipo → INT;
Tipo.a = int().
Tipo → REAL;
Tipo.a = real().
Tipo → BOOL;
Tipo.a = bool().
SecInstr → SecInstr PUNTOCOMA Instruccion;
SecInstr0.a = instr_muchas(SecInstr1.a, Instruccion.a).
SecInstr → Instrucción;
SecInstr.a = instr_una(Instruccion.a).
Instruccion → IDENTIFICADOR ASIGNACION E0;
Instruccion.a = instruccion(IDENTIFICADOR.lex, E0.a).
E0 → E1 SUMA E0;
$E0_1$.a = suma(E1.a, $E0_2$.a).
E0 → E1 RESTA E1;
E0.a = resta($E1_1$.a, $E1_2$.a).
E0 → E1;
E0.a = E1.a.
E1 → E1 Op1 E2;
$E1_1$.a = exp(Op1.op, $E1_2$.a, E2.a).
E1 → E2;
E1.a = E2.a.
E2 → E2 Op2 E3;
$E2_1$.a = exp(Op2.op, $E2_2$.a, E3.a).
E2 → E3;
E2.a = E3.a.
E3 → E4 Op3 E4;
E3.a = exp(Op3.op, $E4_1$.a, $E4_2$.a).
E3 → E4;
E3.a = E4.a
E4 → RESTA E5;
E4.a = negativo(E5.a).
E4 → NOT E4;
$E4_1$.a = not($E4_2$.a).
E4 → E5;
E4.a = E5.a .

E5 → NUM_ENT;
E5.a = num_ent(NUM_ENT.lex).
E5 → NUM_REAL;
E5.a = num_real(NUM_REAL.lex).
E5 → BOOLEAN;
E5.a = boolean(BOOLEAN.lex).
E5 → IDENTIFICADOR;
E5.a = id(IDENTIFICADOR.lex).
E5 → PAR_ABIERTO E0 PAR_CERRADO;
E5.a = E0.a.
Op1 → AND;
Op1.op = 'and'.
Op1 → OR;
Op1.op ='or'.
Op2 → MENOR;
Op2.op ='menor'.
Op2 → MAYOR;
Op2.op ='mayor'.
Op2 → MENOR_IGUAL;
Op2.op ='menor_igual'.
Op2 → MAYOR_IGUAL;
Op2.op ='mayor_igual'.
Op2 → DIFERENTE;
Op2.op ='diferente'.
Op2 → IGUAL;
Op2.op ='igual'.
Op3 → MULT;
Op3.op = 'mult'.
Op3 → DIV;
Op3.op = 'div'.

FUNCIONES SEMÁNTICAS:

```
fun exp(Op, Arg0, Arg1){
        switch Op
        case 'and':
                return and(Arg0,Arg1)
        case 'or':
                return or(Arg0,Arg1)
        case 'menor':
                return menor(Arg0,Arg1)
        case 'mayor':
                return mayor(Arg0,Arg1)
        case 'menor_igual':
                return menor_igual(Arg0,Arg1)
```

```
        case ’mayor_igual’:
            return mayor_igual(Arg0,Arg1)
        case ’diferente’:
            return diferente(Arg0,Arg1)
        case ’igual’:
            return igual(Arg0,Arg1)
        case ‘mult’:
            return mult(Arg0,Arg1)
        case ‘div’:
            return div(Arg0,Arg1)
}
```

3. Acondicionamiento para implementación descendente

SAux → S |-;
SAux.a = S.a.
S → SecDec SEPARADOR SecInstr;
S.a = prog(SecDec.a, SecInsr.a).
SecDec → Declaracion SecDecAux;
SecDec.a = SecDecAux.a .
SecDecAux.ah = decs_una(Declaracion.a).
SecDecAux → PUNTOCOMA Declaracion SecDecAux;
SecDecAux0.a = SecDecAux1.a.
SecDecAux1.ah = decs_muchas(SecDecAux0.ah, Declaracion.a).
SecDecAux → ε;
SecDecAux.a = SecDecAux.ah.
Declaracion → Tipo IDENTIFICADOR;
Declaracion.a = dec(Tipo.a, IDENTIFICADOR.lex).
Tipo → INT;
Tipo.a = “int”
Tipo → REAL;
Tipo.a = “real”
Tipo → BOOL;
Tipo.a = “bool”
SecInstr → Instruccion SecInstrAux;
SecInstr.a = SecInstrAux.a.
SecInstrAux.ah = intr_una(Instruccion.a).
SecInstrAux → PUNTOCOMA Instruccion SecInstrAux;
SecInstrAux0.a = SecInstrAux1.a.
SecInstrAux1.ah = instr_muchas (SecInstrAux0.ah, Instruccion.a).
SecInstrAux → ε;
SecInstrAux.a = SecInstrAux.ah.

Instruccion → IDENTIFICADOR ASIGNACION E0;

Instruccion.a = instr(IDENTIFICADOR.lex, E0.a).

E0 → E1 E0_AUX;

E0_AUX.ah = E1.a.

E0.a = E0_AUX.a.

E0_AUX → SUMA E0;

E0_AUX.a = suma(E0_aux.ah, E0.a).

E0_AUX → RESTA E1;

E0_AUX.a = resta(E0_aux.ah, E1.a).

E0_AUX → ε;

E0_AUX.a = E0_AUX.ah.

E1 → E2 E1_AUX;

E1_AUX.ah = E2.a.

E1.a = E1_AUX.a.

E1_AUX → Op1 E2 E1_AUX;

E1_AUX0.a = E1_AUX1.a.

E1_AUX1.ah = exp (op1.op, E1_AUX0.ah, E2.a).

E1_AUX → ε;

E1_AUX.a = E1_AUX.ah..

E2 → E3 E2_AUX;

E2_AUX.ah = E3.a.

E2.a = E2_AUX.a.

E2_AUX → Op2 E3 E2_AUX ;

E2_AUX0.a = E2_AUX1.a.

E2_AUX1.ah = exp (op2.op, E2_AUX0.ah, E3.a).

E2_AUX →ε ;

E2_AUX.a = E2_AUX.ah..

E3 → E4 E3_AUX;

E3.a = E3_AUX.a.

E3_AUX.ah = E4.a.

E3_AUX → Op3 E4;

E3_AUX.a = exp(Op3.op,E3_AUX.ah, E4.a).

E3_AUX → ε;

E3_AUX.a = E3_AUX.ah.

E4 → RESTA E5;

E4.a = negativo(E5.a).

E4 → NOT E4;

$E4_1.a = not(E4_2.a)$.

E4 → E5;

E4.a = E5.a.

E5 → NUM_ENT;

E5.a = num_ent(NUM_ENT.lex).

E5 → NUM_REAL;

E5.a = num_real(NUM_REAL.lex).

E5 → BOOLEAN;

E5.a = boolean(BOOLEAN.lex).

E5 → IDENTIFICADOR;
E5.a = id(IDENTIFICADOR.lex).
E5 → PAR_ABIERTO E0 PAR_CERRADO;
E5.a = E0.a.
Op1 → AND;
Op1.op = ‘and’
Op1 → OR;
Op1.op =’or’
Op2 → MENOR;
Op2.op =’menor’
Op2 → MAYOR;
Op2.op =’mayor’
Op2 → MENOR_IGUAL;
Op2.op =’menor_igual’
Op2 → MAYOR_IGUAL;
Op2.op =’mayor_igual’
Op2 → DIFERENTE;
Op2.op =’diferente’
Op2 → IGUAL;
Op2.op =’igual’
Op3 → MULT;
Op3.op = ‘mult’
Op3 → DIV;
Op3.op = ‘div’