**Constructor de Árboles de Sintaxis Abstracta**

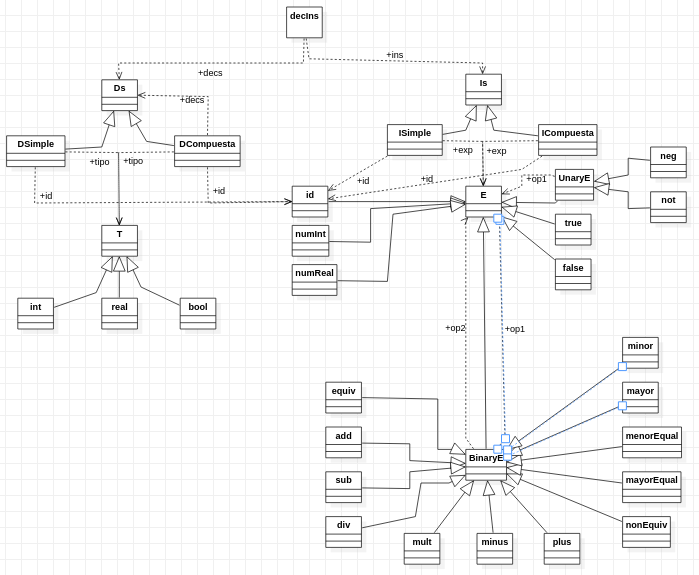
**Jaime Sáez de Buruaga Brouns**

**Julia Miguélez Fernández-Villacañas**

1. **Conjunto de funciones constructoras**

|  |  |
| --- | --- |
| **Regla** | **Constructora** |
| S → Ds && Is  Ds → T id  Ds → Ds ; T id  T → num  T → bool  T → real  Is → id = E  Is → Is ; id = E  E → E + E  E → E – E  E → E and E  E → E or E  E → E < E  E → E > E  E → E <= E  E → E >= E  E → E == E  E → E != E  E → E \* E  E → E / E  E → - E  E → not E  E → (E)  E → id  E → numReal  E → numInt  E → true  E → false | **decIns:** Ds x Is → S  **dSimple**: T x string → Ds  **dCompuesta**: Ds x T x string → Ds  **tInt**: T  **tBool**: T  **tReal**: T  **iSimple**: string x E → Is  **iCompuesta**: Is x string x E → Is  **suma**: E x E → E  **resta**: E x E → E  **and**: E x E → E  **or**: E x E → E  **minor**: E x E → E  **mayor**: E x E → E  **minorEqual**: E x E → E  **mayorEqual**: E x E → E  **equiv**: E x E → E  **nonEquiv**: E x E → E  **mult**: E x E → E  **div**: E x E → E  **neg**: E → E  **not**: E → E  **id**: string → E  **numReal**: string → E  **numInt**: string → E  **true**: E  **false**: E |

1. **Diseño de sintaxis abstracta mediante diagrama de clases**

****

1. **Especificación del constructor de árboles de sintaxis abstracta**

Se supone una función semántica auxiliar:

type : {Op1, Op2, Op} x E x E → E

*fun op(op, arg1, arg2){*

*switch(op{*

*case “+”: return suma(arg1, arg2);*

*case “-”: return resta(arg1, arg2);*

*case “\*”: return mult(arg1, arg2);*

*case “/”: return div(arg1, arg2);*

*case “>”: return mayor(arg1, arg2);*

*case “<”: return minor(arg1, arg2);*

*case “>=”: return mayorEqual(arg1, arg2);*

*case “<=”: return minorEqual(arg1, arg2);*

*case “==”: return equiv(arg1, arg2);*

*case “!=”: return nonEquiv(arg1, arg2);*

*}*

*}*

**Gramática de atributos para el constructor de árboles**

|  |  |
| --- | --- |
| **Regla** | **Constructora** |
| S → Ds && Is | S.a = decIns(Ds.a, Is.a) |
| Ds → D  Ds → Ds; D | Ds.a = dSimple(D.type, D.iden)  Ds0.a = dCompuesta(Ds1.a, D.type, D.iden) |
| D → T id | D.type = T.a  D.id = id.lex |
| T → int  T → real  T → bool | T.a = tInt()  T.a = tReal()  T.a = tBool() |
| Is → I  Is → Is ; I | Is.a = iSimple(I.iden, I.exp)  Is0.a = iCompuesta(Is1.a, I.iden, I.exp) |
| I → id = E0 | I.a = iSimple(id.lex, E0.a) |
| E0 → E0 Op1 E1  E0 → E1 | E00.a = op(Op1.op, E01.a, E1.a)  E0.a = E1.a |
| E1 → E2 and E1  E1 → E2 or E2  E1 → E2 | E10.a = and(E2.a, E11.a)  E1.a = or(E20.a, E21.a)  E1.a = E2.a |
| E2 → E3 Op E3  E2 → E3 | E2.a = op(Op.op, E30.a, E31.a)  E2.a = E3.a |
| E3 → E3 Op2 E4  E3 → E4 | E31.a = op(Op2.op, E31.a, E4.a)  E3.a = E4.a |
| E4 → - E4  E4 → not E5  E4 → E5 | E40.a = neg(E41.a)  E4.a = not(E5.a)  E4.a = E5.a |
| E5 → (E0)  E5 → id  E5 → numReal  E5 → numInt  E5 → true  E5 → false | E5.a = E0.a  E5.a = id(id.lex) E5.a = numReal(numReal.lex)  E5.a = numInt(numInt.lex)  E5.a = true()  E5.a = false() |
| Op1 → +  Op1 → - | Op1.op = +  Op1.op = - |
| Op2 → \*  Op2 → / | Op2.op = \*  Op2.op = / |
| Op → <  Op → >  Op → <=  Op → >=  Op → ==  Op → != | Op.op = <  Op.op = >  Op.op = <=  Op.op = >=  Op.op = ==  Op.op = != |

1. **Acondicionamiento de dicha especificación para implementación descendente**

|  |  |
| --- | --- |
| **Regla** | **Constructora** |
| S → Ds && Is | S.a = *decIns(*Ds.a, Is.a*)* |
| Ds → D FD | FD.ah = *dSimple(*D.type, D.iden*)*  Ds.a = FD.a |
| D → T id | D.type = T.a  D.iden = id.lex |
| FD → ; D FD  FD → epsilon | FD1.a = *dCompuesta(*FD0.ah, Ds.type, Ds.exp*)*  FD0.a = FD1.a  FD.a = FD.ah |
| T → int  T → real  T → bool | T.a = *tInt()*  T.a = *tReal()*  T.a = *tBool()* |
| Is → I FI | FI.ah = *iSimple(*I.iden, I.exp*)*  Is.a = FI.a |
| I → id = E0 | I.iden = id.lex  I.exp = E0.a |
| FI → ; I FI  FI → epsilon | FI1.ah = *iCompuesta(*FI0.ah, I.type, I.exp*)*  FI0.a = FI1.a  FI.a = FI.ah |
| E0 → E1 E0’ | E0’.ah = E1.a  E0.a = E0’.a |
| E0’ → Op1 E1 E0’  E0’ → epsilon | E0’0.ah = *op(*E0’0.ah, E1.a*)*  E0’0.a = E0’1.a  E0’.a = E0’.ah |
| E1 → E2 EE1 | EE1.ah = E2.a  E1.a = EE1.a |
| EE1 → and E1  EE1 → or E2  EE1 → epsilon | EE1.a = and(EE1.ah, E1.a*)*  EE1.a = *or(*EE1.ah, E2.a*)*  EE1.a = EE1.ah |
| E2 → E3 EE2 | E2.a = EE2.a  EE2.ah = E3.a |
| EE2 → Op E3  EE2 → E3 | EE2.a = *op(*Op.op, EE2.ah, E3.a*)*  EE2.a = E3.a |
| E3 → E4 E3’ | E3’.ah = E4.a  E3.a = E3’.a |
| E3’ → Op2 E4 E3’  E3’ → epsilon | E3’1.ah = *op(*Op2.op, E3’0.ah, E4.a*)*  E3’0.a = E3’1.a |
| E4 → - E4  E4 → not E5  E4 → E5 | E40.a = *neg(*E41.a*)*  E4.a = *not(*E5.a*)*  E4.a = E5.a |
| E5 → id  E5 → numReal  E5 → numInt  E5 → true  E5 → false | E5.a = *id(*id.lex*)*  E5.a = *numReal(*numReal.lex*)*  E5.a = *numInt(*numInt.lex*)*  E5.a = *true()*  E5.a = *false()* |
| Op1 → +  Op1 → - | Op1.op = +  Op1.op = - |
| Op2 → \*  Op2 → / | Op2.op = \*  Op2.op = / |
| Op → <  Op → >  Op → <=  Op → >=  Op → ==  Op → != | Op.op = <  Op.op = >  Op.op = <=  Op.op = >=  Op.op = ==  Op.op = != |