



Práctica 3

Procesadores de Lenguajes

Borja Aday Guadalupe Luis
Diego Enrique de Miguel López
Grupo 13

Índice

Índice.....	2
1. Sintaxis abstracta.	3
2.Constructor de árboles de sintaxis abstracta (ASTs).	5
1.1. Funciones semánticas.....	8
2. Acondicionamiento para implementación descendente.	9

1. Sintaxis abstracta.

Gramática Tiny 1	Sintaxis abstracta
Programa -> LDecs && LIns Programa -> LIns	programa_sin_decs: LIns -> Programa programa_con_decs: LDecs x LIns -> Programa
LDecs -> LDecs ; Dec LDecs -> Dec LDecs -> ε	decs_1: Dec -> LDecs decs_muchas: LDecs x Dec -> LDecs
Dec -> var Tipo identificador Dec -> type Tipo identificador Dec -> proc identificador ParForm Bloque ParForm -> (LParams)	dec_var: Tipo x string -> Dec dec_type: Tipo x string -> Dec dec_proc_sin_pf: string x Bloque -> Dec dec_proc_con_pf: string x LParams x Bloque -> Dec
LParams -> LParams , Param LParams -> Param LParams -> ε	bloque_sin_programa: -> Bloque bloque_con_programa: Programa -> Bloque
Param -> Tipo Referencia identificador Referencia -> & Referencia -> ε	param_con_referencia: Tipo x Referencia x string -> Param param_sin_referencia: Tipo x string -> Param referencia: -> Referencia
Bloque -> { ProgramaBloque } ProgramaBloque -> Programa ProgramaBloque -> ε	lparams_1: Param -> Lparams lparams_muchos: Lparams x Param -> LParams
Tipo -> int Tipo -> real Tipo -> bool Tipo -> string Tipo -> identificador Tipo -> array [numEnt] of Tipo Tipo -> record { LCampos }	tipo_int: -> Tipo tipo_real: -> Tipo tipo_bool: -> Tipo tipo_string: -> Tipo tipo_id: string -> Tipo tipo_array: -> string x Tipo -> Tipo tipo_pointer: Tipo -> Tipo tipo_record: LCampos -> Tipo
LCampos -> LCampos ; Campo LCampos -> Campo	campo: Tipo x string -> Campo
Campo -> Tipo identificador Tipo -> pointer Tipo	lcampos_1: Campo -> Lcampos lcampos_muchos: LCampos x Campo -> LCampos
LIns -> LIns ; Ins LIns -> Ins	lins_1: Ins -> LIns lins_muchas: LIns x Ins -> LIns
Ins -> E0 = E0	ins_asignacion: Exp x Exp -> Ins ins_read: Exp -> Ins ins_write: Exp -> Ins ins_nl: -> Ins ins_new: Exp -> Ins ins_delete: Exp -> Ins ins_bloque: Bloque -> Ins
Ins -> if E0 then LInsV endif Ins -> if E0 then LInsV else LInsV endif Ins -> while E0 do LInsV endwhile	ins_call_con_params: string x LParamsReales -> Ins ins_call_sin_params: string -> Ins
LInsV -> LInsV ; Ins LInsV -> Ins LInsV -> ε	preales_1: Exp -> LParamsReales preales_muc: -> LParamsReales x Exp -> LParamsReales
Ins -> read E0 Ins -> write E0 Ins -> nl Ins -> new E0 Ins -> delete E0 Ins -> call identificador (LParamsReales)	

LParamsReales -> LParamsReales , E0 LParamsReales -> E0 LParamsReales -> ε Ins -> Bloque E0 -> E1 + E0 E0 -> E1 - E1 E0 -> E1 E1 -> E1 OpN1 E2 E1 -> E2 E2 -> E2 OpN2 E3 E2 -> E3 E3 -> E4 OpN3 E4 E3 -> E4 E4 -> - E5 E4 -> not E4 E4 -> E5 E5 -> E5 [E0] E5 -> E5 . identificador E5 -> E5 -> identificador E5 -> E6 E6 -> * E6 E6 -> E7 E7 -> identificador E7 -> numEnt E7 -> numReal E7 -> true E7 -> false E7 -> cadena E7 -> null E7 -> (E0) OpN1 -> and OpN1 -> or OpN2 -> < OpN2 -> > OpN2 -> <= OpN2 -> >= OpN2 -> == OpN2 -> != OpN3 -> * OpN3 -> / OpN3 -> %	ins_if_con_ins: Exp x LInsV -> Ins ins_if_sin_ins: Exp -> Ins ins_if_else_con_ins: Exp x LInsV x LInsV -> Exp ins_if_else_insarg0: Exp x LInsV -> Exp ins_if_else_insarg1: Exp x LInsV -> Exp ins_if_else_sin_ins: Exp -> Ins ins_while_con_ins: Exp x LInsV -> Ins ins_while_sin_ins: Exp -> Ins linsv_1: Ins -> LInsV linsv_muchas: LInsV x Ins -> LInsV suma: Exp x Exp -> Exp resta: Exp x Exp -> Exp and: Exp x Exp -> Exp or: Exp x Exp -> Exp menor: Exp x Exp -> Exp mayor: Exp x Exp -> Exp menor_igual: Exp x Exp -> Exp mayor_igual: Exp x Exp -> Exp igualdad: Exp x Exp -> Exp distinto: Exp x Exp -> Exp mul: Exp x Exp -> Exp div: Exp x Exp -> Exp mod: Exp x Exp -> Exp menos: Exp -> Exp not: Exp -> Exp index: Exp x Exp -> Exp access_reg_flecha: Exp x string -> Exp access_reg_punto: Exp x string -> Exp indireccion: Exp -> Exp identificador: string -> Exp numEnt: string -> Exp numReal: string -> Exp cadena: string -> Exp true: -> Exp false: -> Exp null: -> Exp
--	--

2. Constructor de árboles de sintaxis abstracta (ASTs).

Constructor de árboles de sintaxis abstracta. Gramática S-Atribuida

```
Programa -> LDecs && LIns
  Programa.a = programa_con_decs(LDecs.a, LIns.a)
Programa -> LIns
  Programa.a = programa_sin_decs(LIns.a)

LDecs -> LDecs ; Dec
  LDecs0.a = decs_muchas(LDecs1.a, Dec.a)
LDecs -> Dec
  LDecs.a = decs_1(Dec.a)
LDecs -> ε
  LDecs.a = null

Dec -> var Tipo identificador
  Dec.a = dec_var(Tipo.a, identificador.lexema)
Dec -> type Tipo identificador
  Dec.a = dec_type(Tipo.a, identificador.lexema)
Dec -> proc identificador ParForm Bloque
  Dec.a = dec_proc(identificador.lexema, ParForm.a, Bloque.a)
ParForm -> ( LParams )
  ParForm.a = LParams.a

LParams -> LParams , Param
  LParams0.a = lparams_muchos(LParams1.a, Param.a)
LParams -> Param
  LParams.a = lparams_1(Param.a)
LParams -> ε
  LParams.a = null

Param -> Tipo Referencia identificador
  Param.a = param(Tipo.a, Referencia.a, identificador.lexema)
Referencia -> &
  Referencia.a = referencia()
Referencia -> ε
  Referencia.a = null

Bloque -> { ProgramaBloque }
  Bloque.a = bloque(ProgramaBloque.a)
ProgramaBloque -> Programa
  ProgramaBloque.a = Programa.a
ProgramaBloque -> ε
  ProgramaBloque.a = null

Tipo -> int
  Tipo.a = tipo_int()
Tipo -> real
  Tipo.a = tipo_reall()
Tipo -> bool
  Tipo.a = tipo_bool()
Tipo -> string
  Tipo.a = tipo_string()
Tipo -> identificador
  Tipo.a = tipo_id(identificador.lexema)
Tipo -> array [ numEnt ] of Tipo
  Tipo.a = tipo_array(numEnt.lexema, Tipo.a)
Tipo -> record { LCampos }
  Tipo.a = tipo_record(LCampos.a)
```

LCampos -> LCampos ; Campo
 LCampos₀.a = lcampos_muchos(LCampos₁.a, Campo.a)
LCampos -> Campo
 LCampos.a = lcampos_1(Campo.a)

Campo -> Tipo **identificador**
 Campo.a = campo(Tipo.a, **identificador**.lexema)
Tipo -> **pointer** Tipo
 Tipo.a = tipo_pointer(Tipo.a)

LIns -> LIns ; Ins
 LIns₀.a = ins_muchas(LIns₁.a, Ins.a)
LIns -> Ins
 LIns.a = ins_1(Ins.a)

Ins -> E0 = E0
 Ins.a = ins_asignacion(E0₀.a, E0₁.a)

Ins -> **if** E0 **then** LInsV **endif**
 Ins.a = ins_if(E0.a, LInsV.a)
Ins -> **if** E0 **then** LInsV **else** LInsV **endif**
 Ins.a = ins_if_else(E0.a, LInsV₀.a, LInsV.a)

Ins -> **while** E0 **do** LInsV **endwhile**
 Ins.a = ins_while(E0.a, LInsV.a)

LInsV -> LInsV ; Ins
 LInsV₀.a = linsv_muchas(LInsV₁.a, Ins.a)
LInsV -> Ins
 LInsV.a = linsv_1(Ins.a)
LInsV -> ε
 LInsV.a = **null**

Ins -> **read** E0
 Ins.a = ins_read(E0.a)
Ins -> **write** E0
 Ins.a = ins_write(E0.a)
Ins -> **nl**
 Ins.a = ins_nl()
Ins -> **new** E0
 Ins.a = ins_new(E0.a)
Ins -> **delete** E0
 Ins.a = ins_delete(E0.a)
Ins -> **call identificador** (LParamsReales)
 Ins.a = ins_call(**identificador**.lexema, LParamsReales.a)

LParamsReales -> LParamsReales , E0
 LParamsReales₀.a = preales_muc(LParamsReales₁.a, E0.a)
LParamsReales -> E0
 LParamsReales.a = preales_1 (E0.a)
LParamsReales -> ε
 LParamsReales.a = **null**

Ins -> Bloque
 Ins.a = ins_bloque(Bloque.a)

E0 -> E1 + E0

E0₀.a = suma(E1.a, E0₁.a)

E0 -> E1 - E1

E0.a = resta(E1₀.a, E1₁.a)

E0 -> E1

E0.a = E1.a

E1 -> E1 OpN1 E2

E1₀.a = exp(OpN1.op, E1₁.a, E2.a)

E1 -> E2

E1.a = E2.a

E2 -> E2 OpN2 E3

E2₀.a = exp(OpN2.op, E2₁.a, E3.a)

E2 -> E3

E2.a = E3.a

E3 -> E4 OpN3 E4

E3.a = exp(OpN3.op, E4₀.a, E4₁.a)

E3 -> E4

E3.a = E4.a

E4 -> - E5

E4.a = menos(E5.a)

E4 -> **not** E4

E4₀.a = not(E4₁.a)

E4 -> E5

E4.a = E5.a

E5 -> E5 [E0]

E5₀.a = index(E5₁.a, E0.a)

E5 -> E5 . **identificador**

E5₀.a = access_reg_punto(E5₁.a, **identificador**.lexema)

E5 -> E5 -> **identificador**

E5₀.a = access_reg_flecha(E5₁.a, **identificador**.lexema)

E5 -> E6

E5.a = E6.a

E6 -> * E6

E6₀.a = indireccion(E6₁.a)

E6 -> E7

E6.a = E7.a

E7 -> **identificador**

E7.a = identificador(**identificador**.lexema)

E7 -> **numEnt**

E7.a = identificador(**numEnt**.lexema)

E7 -> **numReal**

E7.a = identificador(**numReal**.lexema)

E7 -> **true**

E7.a = true()

E7 -> **false**

E7.a = false()

E7 -> **cadena**

E7.a = cadena(**cadena**.lexema)

E7 -> **null**

E7.a = null()

E7 -> (E0)

E7.a = E0.a

```

OpN1 -> and
  opN1.a = "and"
OpN1 -> or
  opN1.a = "or"

OpN2 -> <
  opN2.a = "<"
OpN2 -> >
  opN2.a = ">"
OpN2 -> <=
  opN2.a = "<="
OpN2 -> >=
  opN2.a = ">="
OpN2 -> ==
  opN2.a = "=="
OpN2 -> !=
  opN2.a = "!="

OpN3 -> *
  opN3.a = "*"
OpN3 -> /
  opN3.a = "/"
OpN3 -> %
  opN3.a = "%"

```

2.1. Funciones semánticas.

```

fun exp(Op,Arg0,Arg1) {
  switch Op
  case "and": return and(Arg0, Arg1)
  case "or": return or(Arg0, Arg1)
  case "<": return menor(Arg0, Arg1)
  case ">": return mayor(Arg0, Arg1)
  case "<=": return menor_igual(Arg0, Arg1)
  case ">=": return mayor_igual(Arg0, Arg1)
  case "==": return igualdad(Arg0, Arg1)
  case "!=": return distinto(Arg0, Arg1)
  case "*": return mul(Arg0, Arg1)
  case "/": return div(Arg0, Arg1)
  case "%": return mod(Arg0, Arg1)
}

fun dec_proc(Id, LParams, Bloque) {
  if LParams == null then return dec_proc_sin_pf(Id, Bloque)
  else return dec_proc_con_pf(Id, LParams, Bloque)
}

fun param(Tipo, Ref, Id) {
  if Ref == null then return param_sin_referencia(Tipo, Id)
  else return param_con_referencia(Tipo, Referencia, Id)
}

fun ins_if(Exp, LInsV) {
  if LInsV == null then return ins_if_sin_ins(Exp)
  else return ins_if_con_ins(Exp, LInsV)
}

```



```

fun ins_if_else(Exp, LInsV1, LInsV2) {
  if LInsV1 == null && LInsV2 == null then return ins_if_else_sin_ins(Exp)
  else if LInsV1 != null && LInsV2 != null then return ins_if_else_con_ins(Exp, LInsV1, LInsV2)
  else if LInsV1 != null then return ins_if_else_insarg0(Exp, LInsV1)
  else return ins_if_else_insarg1(Exp, LInsV2)
}

fun ins_while(Exp, LInsV) {
  if LInsV == null then return ins_while_sin_ins(Exp)
  else return ins_while_con_ins(Exp, LInsV)
}

fun ins_call(Id, LParamsReales) {
  if LParamReales == null then return ins_call_sin_params(Id)
  else return ins_call_con_params(Id, LParamsReales)
}

fun bloque(Programa) {
  if Programa == null then return bloque_sin_programa()
  else return bloque_con_programa(Programa)
}

```

3. Acondicionamiento para implementación descendente.

Constructor de árboles de sintaxis abstracta. Gramática S-Atribuida
Programa -> LDecs && LIns Programa.a = programa_con_decs(LDecs.a, LIns.a) Programa -> LIns Programa.a = programa_sin_decs(LIns.a) LDecs -> Dec RLDecs RLDecs.ah = decs_1(Dec.a) LDecs.a = RLDecs.a LDecs -> ε LDecs.a = null RLDecs -> ; Dec RLDecs RLDecs ₁ .ah = decs_muchas(RLDecs ₀ .ah, Dec.a) RLDecs ₀ .a = RLDecs ₁ .a RLDecs -> ε RLDecs.a = RLDecs.ah Dec -> var Tipo identificador Dec.a = dec_var(Tipo.a, identificador .lexema) Dec -> type Tipo identificador Dec.a = dec_type(Tipo.a, identificador .lexema) Dec -> proc identificador ParForm Bloque Dec.a = dec_proc(identificador .lexema, ParForm.a, Bloque.a) ParForm -> (LParams) ParForm.a = LParams.a LParams -> Param RLParams RLParams.ah = lparams_1(Param.a) LParams.a = RLParams.a LParams -> ε LParams.a = null RLParams -> , Param RLParams

RLParams₁.ah = lparams_muchos(RLParams₀.ah, Param.a)
RLParams₀.a = RLParams₁.a
RLParams -> ε
RLParams.a = RLParams.ah

Param -> Tipo Referencia **identificador**
Param.a = param(Tipo.a, Referencia.a, **identificador**.lexema)
Referencia -> **&**
Referencia.a = referencia()
Referencia -> ε
Referencia.a = **null**

Bloque -> { ProgramaBloque }
Bloque.a = bloque(ProgramaBloque.a)
ProgramaBloque -> Programa
ProgramaBloque.a = Programa.a
ProgramaBloque -> ε
ProgramaBloque.a = **null**

Tipo -> **int**
Tipo.a = tipo_int()
Tipo -> **real**
Tipo.a = tipo_real()
Tipo -> **bool**
Tipo.a = tipo_bool()
Tipo -> **string**
Tipo.a = tipo_string()
Tipo -> **identificador**
Tipo.a = tipo_id(**identificador**.lexema)
Tipo -> **array [numEnt] of** Tipo
Tipo.a = tipo_array(**numEnt**.lexema, Tipo.a)
Tipo -> **record { LCampos }**
Tipo.a = tipo_record(LCampos.a)

LCampos -> Campo RLCampos
RLCampos.ah = lcampos_1 (Campo.a)
LCampos.a = RLCampos.a
RLCampos -> ; Campo RLCampos
RLCampos₁.ah = lcampos_muchos(RLCampos₀.ah, Campo.a)
RLCampos₀.a = RLCampos₁.a
RLCampos -> ε
RLCampos.a = RLCampos.ah

Campo -> Tipo **identificador**
Campo.a = campo(Tipo.a, **identificador**.lexema)
Tipo -> **pointer** Tipo
Tipo.a = tipo_pointer(Tipo.a)

LIns -> Ins RLIns
RLIns.ah = ins_1 (Ins.a)
LIns.a = RLIns.a
RLIns -> ; Ins RLIns
RLIns₁.ah = ins_muchas(RLIns₀.ah, Ins.a)
RLIns₀.a = RLIns₁.a
RLIns -> ε
RLIns.a = RLIns.ah

Ins -> E0 = E0
Ins.a = ins_asignacion(E0₀.a, E0₁.a)

```

Ins -> if E0 then LInsV RInsIfThen
    RInsIfThen.ah0 = E0.a
    RInsIfThen.ah1 = LInsV.a
    Ins.a = RInsIfThen.a
RInsIfThen -> endif
    RInsIfThen.a = ins_if(RInsIfThen.ah0, RInsIfThen.ah1)
RInsIfThen -> else LInsV endif
    RInsIfThen.a = ins_if_else(RInsIfThen.ah0, RInsIfThen.ah1, LInsV.a)

```

```

Ins -> while E0 do LInsV endwhile
    Ins.a = ins_while(E0.a, LInsV.a)

```

```

LInsV -> Param RLInsV
    RLInsV.ah = linsv_1(Ins.a)
    LInsV.a = RLInsV.a
LInsV -> ε
    LInsV.a = null
RLInsV -> ; Param RLInsV
    RLInsV1.ah = linsv_muchas(RLInsV0.ah, Ins.a)
    RLInsV0.a = RLInsV1.a
RLInsV -> ε
    RLInsV.a = RLInsV.ah

```

```

Ins -> read E0
    Ins.a = ins_read(E0.a)
Ins -> write E0
    Ins.a = ins_write(E0.a)
Ins -> nl
    Ins.a = ins_nl()
Ins -> new E0
    Ins.a = ins_new(E0.a)
Ins -> delete E0
    Ins.a = ins_delete(E0.a)
Ins -> call identificador ( LParamsReales )
    Ins.a = ins_call(identificador.lexema, LParamsReales.a)

```

```

LParamsReales -> E0 RLPParamsReales
    RLPParamsReales.ah = preales_1(E0.a)
    LParamsReales.a = RLPParamsReales.a
LParamsReales -> ε
    LParamsReales.a = null
RLParamsReales -> , E0 RLPParamsReales
    RLPParamsReales1.ah = preales_muc(RLPParamsReales0.ah, E0.a)
    RLPParamsReales0.a = RLPParamsReales1.a
RLParamsReales -> ε
    RLPParamsReales.a = RLPParamsReales.ah

```

```

Ins -> Bloque
    Ins.a = ins_bloque(Bloque.a)

```

```

E0 -> E1 RE0
    RE0.ah = E1.a
    E0.a = RE0.a
RE0 -> + E0
    RE0.a = suma(RE0.ah, E0.a)
RE0 -> - E1
    RE0.a = resta(RE0.ah, E1.a)
RE0 -> ε
    RE0.a = RE0.ah

```

E1 -> E2 RE1
 RE1.ah = E2.a
 E1.a = RE1.a
 RE1 -> OpN1 E2 RE1
 RE1₁.ah = exp(opN1.op, RE1₀.ah, E2.a)
 RE1 -> ε
 RE1.a = RE1.ah

E2 -> E3 RE2
 RE2.ah = E3.a
 E2.a = RE2.a
 RE2 -> OpN2 E3 RE2
 RE2₁.ah = exp(opN2.op, RE2₀.ah, E3.a)
 RE2 -> ε
 RE2.a = RE2.ah

E3 -> E4 RE3
 RE3.ah = E4.a
 E3.a = RE3.a
 RE3 -> OpN3 E4
 RE3.a = exp(opN3.op, RE3.ah, E4.a)
 RE3 -> ε
 RE3.a = RE3.ah

E4 -> - E5
 E4.a = menos(E5.a)
 E4 -> **not** E4
 E4₀.a = not(E4₁.a)
 E4 -> E5
 E4.a = E5.a

E5 -> E6 RRE5
 RRE5.ah = E6.a
 E5.a = RRE5.a
 RRE5 -> RE5 RRE5
 RRE5₁.ah = RE5.a
 RRE5₀.a = RRE5₁.a
 RE5.ah = RRE5₀.ah
 RRE5 -> ε
 RRE5.a = RRE5.ah
 RE5 -> [E0]
 RE5.a = index(RE5.ah, E0.a)
 RE5 -> . **identificador**
 RE5.a = access_reg_punto(RE5.ah, **identificador**.lexema)
 RE5 -> -> **identificador**
 RE5.a = access_reg_flecha(RE5.ah, **identificador**.lexema)

E6 -> * E6
 E6₀.a = indireccion(E6₁.a)
 E6 -> E7
 E6.a = E7.a

E7 -> **identificador**
 E7.a = identificador(**identificador**.lexema)
 E7 -> **numEnt**
 E7.a = numEnt(**numEnt**.lexema)
 E7 -> **numReal**
 E7.a = numReal(**numReal**.lexema)
 E7 -> **true**

```
E7.a = true()
E7 -> false
E7.a = false()
E7 -> cadena
E7.a = cadena(cadena.lexema)
E7 -> null
E7.a = null()
E7 -> ( E0 )
E7.a = E0.a
```

```
OpN1 -> and
  opN1.a = "and"
OpN1 -> or
  opN1.a = "or"
```

```
OpN2 -> <
  opN2.a = "<"
OpN2 -> >
  opN2.a = ">"
OpN2 -> <=
  opN2.a = "<="
OpN2 -> >=
  opN2.a = ">="
OpN2 -> ==
  opN2.a = "=="
OpN2 -> !=
  opN2.a = "!="
```

```
OpN3 -> *
  opN3.a = "*"
OpN3 -> /
  opN3.a = "/"
OpN3 -> %
  opN3.a = "%"
```