Desarrollo manual de un constructor de ASTs

FACULTAD DE INFORMÁTICA

UNIVERSIDAD
COMPLUTENSE
MADRID

Asignatura de Procesadores de Lenguajes

Curso 2021-2022

Grupo 20

Adrián Martín Tiscar
Gema Blanco Núñez

1. Especificación de la sintaxis abstracta

A continuación se realiza la enumeración de las signaturas (cabeceras) de las funciones constructoras de ASTs.

Sintaxis abstracta

programa: Decs x Instrucciones → Programa
decs_varias: Decs x Dec → Decs
decs_una: Dec → Decs
dec_int: id → Dec
dec_bool: id → Dec
dec_real: id → Dec
inst_varias: Instrucciones x Inst → Instrucciones
inst_una: Inst → Instrucciones
inst: string x Expr → Inst
suma: Expr x Expr → Expr
resta: Expr x Expr → Expr
and: Expr x Expr → Expr
or: Expr x Expr → Expr
mayor: Expr x Expr → Expr
menor: Expr x Expr → Expr
mayor_Igual: Expr x Expr → Expr
menor_Igual: Expr x Expr → Expr
equivalente: Expr x Expr → Expr
distinto: Expr x Expr → Expr
not: Expr → Expr
mul: Expr x Expr → Expr
div: Expr x Expr → Expr
negacion: Expr → Expr
id: string → Exp
numEntero: string → Exp
numReal: string → Exp
true: → Exp
false: → Exp

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

Constructor de árboles de sintaxis abstracta (ASTs):

Programa → Decs '&&' Instrucciones
    Programa.a = programa(Decs.a, Instrucciones.a)
Decs → Decs ';' Dec
    Decs.a = decs_varias(Decs.a, Dec.a)
Decs → Dec

Decs.a = decs_una(Dec.a)

Dec → int id

Dec.a = dec_int(id.lex)

Dec → bool id

Dec.a = dec_bool(id.lex)

Dec → real id

Dec.a = dec_real(id.lex)

Instrucciones → Instrucciones ‘;’ Inst

$Instrucciones_0$.a = inst_varias($Instrucciones_1$.a, Inst.a)

Instrucciones → Inst

Instrucciones.a = inst_una(Inst.a)

Inst → Id ‘=’ Expresion

Inst.a = inst(id.lex, Expresion.a)

Expresion → E0

Expresion.a = E0.a

E0 → E1 ‘+’ E0

$E0_0$.a = suma(E1.a, $E0_1$.a)

E0 → E1 ‘-’ E1

E0.a = resta($E1_1$.a, $E1_2$.a)

E0 → E1

E0.a = E1.a

E1 → E1 OP1 E2

$E1_0$.a = exp(OP1.a, $E1_1$.a, E2)

E1 → E2

E1.a = E2.a

E2 → E2 OP2 E3

$E2_0$.a = exp(OP2.a, $E2_1$.a, E3.a)

E2 → E3

E2.a = E3.a

E3 → E4 OP3 E4

E3.a = exp(OP3.a, $E4_1$.a, $E4_2$.a)

E3 → E4

E3.a = E4.a

E4 → ‘-’ E5

E4.a = negacion(E5.a)

E4 → not E4

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

E4 → E5

E4.a = E5.a

E5 → **numeroEntero**

E5.a = numEntero(**numeroEntero**.a)

E5 → **numeroReal**
E5.a = numReal(**numeroReal**.a)
E5 → **id**
E5.a = id(**id.lex**)
E5 → **true**
E5.a = true()
E5 → **false**
E5.a = false()
E5 → (E0)
E5.a = E0.a

OP1 → **and**
OP1.a = 'and'
OP1 → **or**
OP1.a = 'or'
OP2 → '>'
OP2.a = '>'
OP2 → '>='
OP2.a = '>='
OP2 → '<'
OP2.a = '<'
OP2 → '<='
OP2.a = '<='
OP2 → '=='
OP2.a = '=='
OP2 → '!='
OP2.a = '!='
OP3 → '*'
OP3.a = '*'
OP3 → '/'
OP3.a = '/'

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 equivalente(arg0, arg1)
        case '!=': return distinto(arg0, arg1)
        case '*': return mul(arg0, arg1)
        case '/': return div(arg0, arg1)
}
```

3. Acondicionamiento de la especificación

A continuación se realiza el condicionamiento de la especificación para permitir la implementación descendente. Se aplican dos transformaciones:

- Eliminación de factores comunes
- Eliminación de recursión a izquierdas

En la siguiente tabla se muestran las reglas sin acondicionar y sus respectivas transformaciones.

Eliminación de recursión a izquierdas	
Sin acondicionar	**Acondicionada**
Decs → Decs ';' Dec Decs.a = decs_varias(Decs.a, Dec.a)	Decs → Dec restoDecs restoDecs.ah = decs_una(Dec.a) Decs.a = restoDecs.a restoDecs → ; Dec restoDecs $restoDecs_1$.ah = decs_varias($restoDecs_0$.ah, Dec.a) $restoDecs_0$.a = $restoDecs_1$.a restoDecs → ε restoDecs.a = restoDecs.ah
Instrucciones → Instrucciones ';' Inst	Instrucciones → Inst restoIns restoIns.ah = inst_una(Inst.a) Instrucciones.a = restoIns.a restoIns → ; Inst restoIns $restoIns_1$.ah = inst_varias($restoIns_0$.ah, Inst.a) $restoIns_0$.a = $restoIns_1$.a restoIns → ε restoIns.a = restoIns.ah
E1 → E1 OP1 E2	E1 → E2 restoE1 restoE1.ah = E2.a E1.a = restoE1.a restoE1 → OP1 E2 restoE1 $restoE1_1$.ah = E2.a $restoE1_0$.a = exp(OP1.a, $restoE1_0$.ah, $restoE1_1$.a) restoE1 → ε restoE1.a = restoE1.ah
E2 → E2 OP2 E3 $E2_0$.a = exp(OP2.a, $E2_1$.a, E3.a)	E2 → E3 restoE2 restoE2.ah = E3.a E2.a = restoE2.a restoE2 → OP2 E3 restoE2 $restoE2_1$.ah = E3.a

	restoE2$_0$.ah = exp(OP2.a, restoE2$_0$.ah, restoE2$_1$.a) restoE2 → ε restoE2.a = restoE2.ah
Eliminación de factores comunes	
Sin acondicionar	**Acondicionada**
E0 → E1 '+' E0 E0$_0$.a = suma(E1.a, E0$_1$.a) E0 → E1 '-' E1 E0.a = resta(E1$_1$.a, E1$_2$.a) E0 → E1 E0.a = E1.a	E0 → E1 restoE0 restoE0.ah = E1.a E0.a = restoE0.a restoE0 → '+' E0 restoE0.a = suma(restoE0.ah, E0.a) restoE0 → '-' E1 restoE0.a = resta(restoE0.ah, E1.a) restoE0 → ε restoE0.a = restoE0.ah
E3 → E4 OP3 E4 E3.a = exp(OP3.a, E4$_1$.a, E4$_2$.a) E3 → E4 E3.a = E4.a	E3 → E4 restoE3 restoE3.ah = E4.a E3.a = restoE3.a restoE3 → OP3 E4 restoE3.a = exp(OP3.a, restoE3.ah, E4.a) restoE3 → ε restoE3.a = restoE3.ah

Aplicando las anteriores transformaciones obtenemos la siguiente gramática resultante acondicionada:

Programa → Decs '&&' Instrucciones
Programa.a = programa(Decs.a, Instrucciones.a)
Decs → Dec restoDecs
restoDecs.ah = decs_una(Dec.a)
Decs.a = restoDecs.a
restoDecs → ; Dec restoDecs
restoDecs$_1$.ah = decs_varias(restoDecs$_0$.ah, Dec.a)
restoDecs$_0$.a = restoDecs$_1$.a
restoDecs → ε
restoDecs.a = restoDecs.ah
Dec → int id
Dec.a = dec_int(id.lex)
Dec → bool id
Dec.a = dec_bool(id.lex)
Dec → real id
Dec.a = dec_real(id.lex)
Instrucciones → Inst restoIns

restoIns.ah = inst_una(Inst.a)
Instrucciones.a = restoIns.a

restoIns → ; Inst restoIns
restoIns$_1$.ah = inst_varias(restoIns$_0$.ah, Inst.a)
restoIns$_0$.a = restoIns$_1$.a

restoIns → ε
restoIns.a = restoIns.ah

Inst → Id '=' Expresion
Inst.a = inst(id.lex, Expresion.a)

Expresion → E0
Expresion.a = E0.a

E0 → E1 restoE0
restoE0.ah = E1.a
E0.a = restoE0.a

restoE0 → '+' E0
restoE0.a = suma(restoE0.ah, E0.a)

restoE0 → '-' E1
restoE0.a = resta(restoE0.ah, E1.a)

restoE0 → ε
restoE0.a = restoE0.ah

E1 → E2 restoE1
restoE1.ah = E2.a
E1.a = restoE1.a

restoE1 → OP1 E2 restoE1
restoE1$_1$.ah = E2.a
restoE1$_0$.a = exp(OP1.a, restoE1$_0$.ah, restoE1$_1$.a)

restoE1 → ε
restoE1.a = restoE1.ah

E2 → E3 restoE2
restoE2.ah = E3.a
E2.a = restoE2.a

restoE2 → OP2 E3 restoE2
restoE2$_1$.ah = E3.a
restoE2$_0$.ah = exp(OP2.a, restoE2$_0$.ah, restoE2$_1$.a)

restoE2 → ε
restoE2.a = restoE2.ah

E3 → E4 restoE3
restoE3.ah = E4.a
E3.a = restoE3.a

restoE3 → OP3 E4
restoE3.a = exp(OP3.a, restoE3.ah, E4.a)

restoE3 → ε
restoE3.a = restoE3.ah

E4 → '-' E5

E4.a = negacion(E5.a)

E4 → not E4

E4$_0$.a = not(E4$_1$.a)

E4 → E5

E4.a = E5.a

E5 → **numeroEntero**

E5.a = numEntero(**numeroEntero**.a)

E5 → **numeroReal**

E5.a = numReal(**numeroReal**.a)

E5 → **id**

E5.a = id(**id.lex**)

E5 → **true**

E5.a = true()

E5 → **false**

E5.a = false()

E5 → (E0)

E5.a = E0.a

OP1 → **and**

OP1.a = 'and'

OP1 → **or**

OP1.a = 'or'

OP2 → '>'

OP2.a = '>'

OP2 → '>='

OP2.a = '>='

OP2 → '<'

OP2.a = '<'

OP2 → '<='

OP2.a = '<='

OP2 → '=='

OP2.a = '=='

OP2 → '!='

OP2.a = '!='

OP3 → '*'

OP3.a = '*'

OP3 → '/'

OP3.a = '/'

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 equivalente(arg0, arg1)
        case '!=': return distinto(arg0, arg1)
        case '*': return mul(arg0, arg1)
        case '/': return div(arg0, arg1)
}
```