Complutense de Madrid

Práctica Procesadores de Lenguajes.

Cuarta Fase.

Kyle Tan & Lukas Häring
Grupo 9

May 9, 2019

Tabla de contenidos

1 Diagrama sintáctico

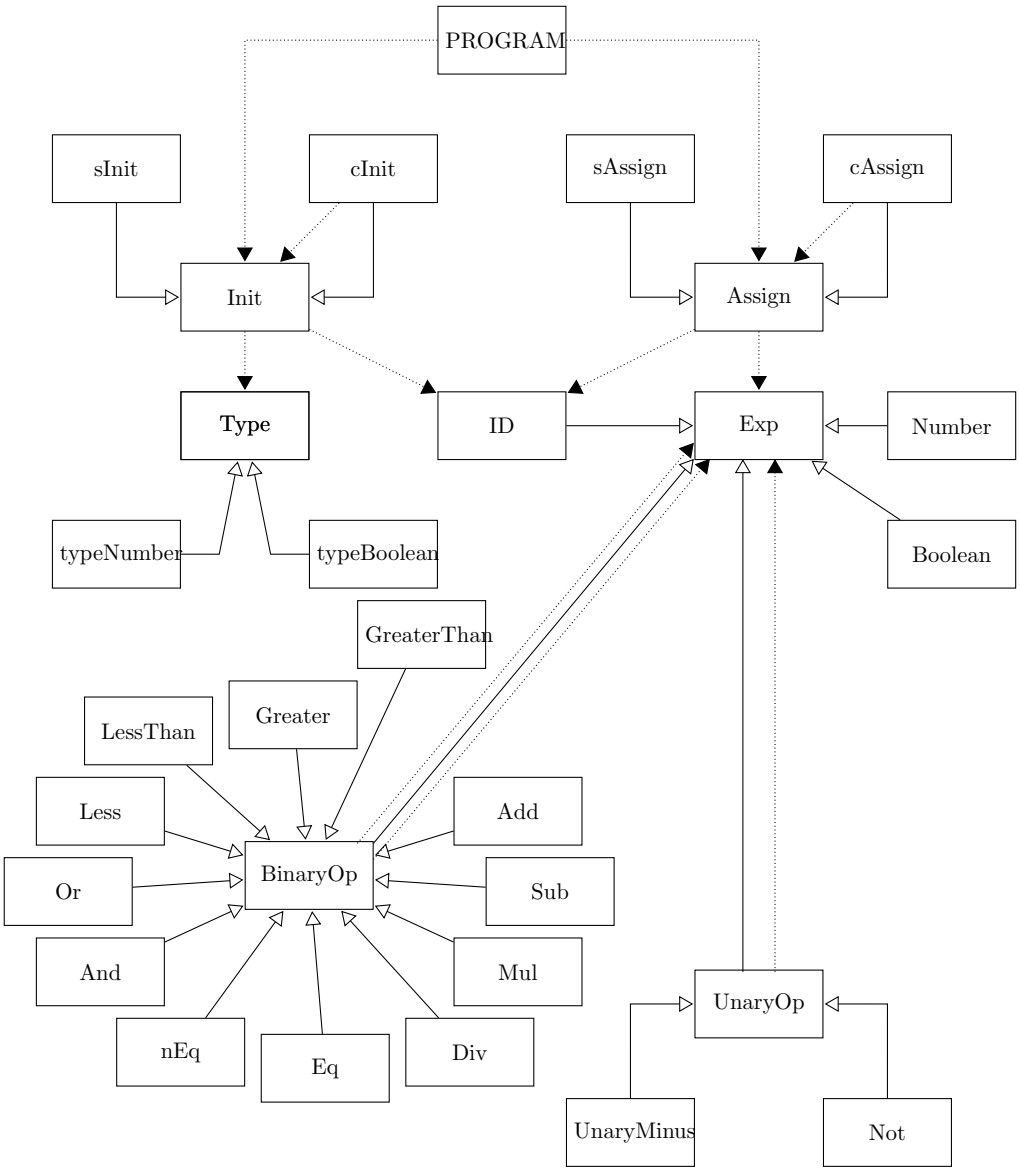
La descripción de la función constructora está basada en el lenguaje matemático:

$$<\text{funcion}>:<\text{arg}_1> \times \cdots \times <\text{arg}_n> \rightarrow <\text{Valor a devolver}>$$

Para la función constructora de inicialización:
$iInit \rightarrow Type$ ID ; $iInit \mid Type$ ID
$Type \rightarrow bool|num$
Para la función constructora de asignación:
$iAssign \rightarrow$ ID = exp ; $iAssign \mid$ ID = exp

Expresión	Función Constructora Programa
iInit && iAssign → Program	PROGRAM: iInit × iAssign → Program
Expresión	Función Constructora Inicialización
Type ID; iInit → iInit	: $Type$ × **string** × iInit → iInit
Type ID → iInit	sInit: $Type$ × **string** → iInt
Expresión	Función Constructora Asignación
ID = Exp; iAssign → iAssign	cAssign: **string**×Exp×iAssign → iAssign
ID = Exp → iAssign	sAssign: **string** × Exp → iAssign
Expresión	Función Constructora Operaciones
Exp + Exp → Exp	Add: Exp × Exp → Exp
Exp − Exp → Exp	Sub: Exp × Exp → Exp
Exp ∗ Exp → Exp	Mul: Exp × Exp → Exp
Exp / Exp → Exp	Div: Exp × Exp → Exp
Exp and Exp → Exp	And: Exp × Exp → Exp
Exp or Exp → Exp	Or: Exp × Exp → Exp
Expresión	Función Constructora Comparaciones
Exp == Exp → Exp	Eq: Exp × Exp → Exp
Exp != Exp → Exp	nEq: Exp × Exp → Exp
Exp < Exp → Exp	Less: Exp × Exp → Exp
Exp <= Exp → Exp	LessThan: Exp × Exp → Exp
Exp > Exp → Exp	Greater: Exp × Exp → Exp
Exp >= Exp → Exp	GreaterThan: Exp × Exp → Exp
Expresión	Función Constructora Unarias
not Exp → Exp	Not: Exp → Exp
−Exp → Exp	UnaryMinus: Exp → Exp
Expresión	Función Constructora Tokens
bool → $Type$	typeBoolean: **num** → Type
num → $Type$	typeNumber: **bool** → Type
ID → Exp	ID: **string** → Exp
number → Exp	Number: **string** → Exp
boolean → Exp	Boolean: **string** → Exp

2 Sintaxis abstracta mediante diagrama de clases



3 Gramática de atributos

Utilizando la implementación de la gramática incontextual de la segunda práctica.

Gramática	Gramática Programa
Program → iInit&&iAssign	Program.a $= PROGRAM(iInit.a, iAssign.a)$
Gramática	**Gramática Inicialización**
iInit → Init; iInit	iInit.a = $sInit(Init.type, Init.id)$
iInit → Init	iInit.a = $cInit(iInit.a, Init.type, Init.id)$
Init → Type ID	Init.type = $Type \wedge$ Init.id = ID.lex
Gramática	**Gramática Asignación**
iAssign → Asgn; iAssign	iAssign.a = $sAssign(Asgn.id, Asgn.exp)$
iAssign → Asgn	iAssign.a = $cAssign(iAssign.a, Asn.id, Asn.exp)$
Asn → ID = Exp	Asn.id = $ID.lex \wedge$ Asn.exp = Exp.a
Gramática	**Gramática Expresiones**
$Exp \rightarrow Exp + E1$	$Exp.a = Add(Exp.a, ELv1.a)$
$Exp \rightarrow Exp - E1$	$Exp.a = Sub(Exp.a, ELv1.a)$
$Exp \rightarrow ELv1$	$Exp.a = ELv1.a$
$ELv1 \rightarrow Exp + E1$	$ELv1.a = And(ELv2.a, ELv1.a)$
$ELv1 \rightarrow Exp - E1$	$ELv1.a = Or(ELv2.a, ELv2.a)$
$ELv1 \rightarrow ELv2$	$ELv1.a = ELv2.a$
$ELv2 \rightarrow ELv3$ Ob $ELv3$	$ELv1.a = opRel(Ob.op, ELv3.a, ELv3.a)$
$Ob \rightarrow op$	$Ob.op = op$
$\forall op \in \{or, not, ==, !=, <, <=, >, >=\}$	
$ELv2 \rightarrow ELv3$	$ELv2.a = ELv.3$
$ELv3 \rightarrow ELv3 * ELv4$	$ELv3.a = Mul(ELv3.a, ELv4.a)$
$ELv3 \rightarrow ELv3/ELv4$	$ELv3.a = Div(ELv3.a, ELv4.a)$
$ELv3 \rightarrow ELv4$	$ELv3.a = ELv4.a$
$ELv4 \rightarrow -ELv4$	$ELv4.a = UnaryMinus(ELv4.a)$
$ELv4 \rightarrow$ not $ELv5$	$ELv4.a = Not(ELv5.a)$
$ELv4 \rightarrow ELv5$	$ELv4.a = ELv5.a$
$ELv5 \rightarrow (Exp)$	$ELv5.a = Exp.a$
$ELv5 \rightarrow num$	$ELv5.a = Number(num.lex)$
$ELv5 \rightarrow bool$	$ELv5.a = Boolean(bool.lex)$
$ELv5 \rightarrow ID$	$ELv5.a = ID(ID.lex)$

Nota: *Asn* es equivalente a *Assign*.

$$opRel(op, left, right) = \begin{cases} op = \text{" == "} & \Rightarrow Eq(left, right) \\ op = \text{"!="} & \Rightarrow nEq(left, right) \\ op = \text{" < "} & \Rightarrow Less(left, right) \\ op = \text{" <= "} & \Rightarrow LessThan(left, right) \\ op = \text{" > "} & \Rightarrow Greater(left, right) \\ op = \text{" >= "} & \Rightarrow GreaterThan(left, right) \end{cases}$$

4 Acondicionamiento para AST descendiente

Utilizando la implementación LL(1) de la práctica 2.

Gramática	Gramática Programa
Program → iInit&&iAssign	Program.a $= PROGRAM(iInit.a, iAssign.a)$
Gramática	**Gramática Inicialización**
iInit → Init iInit'	iInit.a = $sInit(Init.type, Init.id)$
iInit' →; iInit iInit'	iInit'.a = $cInit(iInit.a, Init.type, Init.id)$
iInit' → ϵ	iInit'.a = $iInit.ah$
Init → Type ID	Init.type = $Type \wedge$ Init.id = ID.lex
Gramática	**Gramática Asignación**
iAssign → Asgn; iAssign	iAssign.a = $sAssign(Asgn.id, Asgn.exp)$
iAssign → Asgn	iAssign.a = $cAssign(iAssign.a, Asn.id, Asn.exp)$
Asn → ID = Exp	Asn.id = $ID.lex \wedge$ Asn.exp = Exp.a
Gramática	**Gramática Expresiones**
$Exp \rightarrow Exp + E1$	$Exp.a = Add(Exp.a, ELv1.a)$
$Exp \rightarrow Exp - E1$	$Exp.a = Sub(Exp.a, ELv1.a)$
$Exp \rightarrow ELv1$	$Exp.a = ELv1.a$
$ELv1 \rightarrow Exp + E1$	$ELv1.a = And(ELv2.a, ELv1.a)$
$ELv1 \rightarrow Exp - E1$	$ELv1.a = Or(ELv2.a, ELv2.a)$
$ELv1 \rightarrow ELv2$	$ELv1.a = ELv2.a$
$ELv2 \rightarrow ELv3$ Ob $ELv3$	$ELv1.a = opRel(Ob.op, ELv3.a, ELv3.a)$
$Ob \rightarrow op$	$Ob.op = op$
$\forall op \in \{or, not, ==, !=, <, <=, >, >=\}$	
$ELv2 \rightarrow ELv3$	$ELv2.a = ELv.3$
$ELv3 \rightarrow ELv3 * ELv4$	$ELv3.a = Mul(ELv3.a, ELv4.a)$
$ELv3 \rightarrow ELv3/ELv4$	$ELv3.a = Div(ELv3.a, ELv4.a)$
$ELv3 \rightarrow ELv4$	$ELv3.a = ELv4.a$
$ELv4 \rightarrow -ELv4$	$ELv4.a = UnaryMinus(ELv4.a)$
$ELv4 \rightarrow$ not $ELv5$	$ELv4.a = Not(ELv5.a)$
$ELv4 \rightarrow ELv5$	$ELv4.a = ELv5.a$
$ELv5 \rightarrow (Exp)$	$ELv5.a = Exp.a$
$ELv5 \rightarrow num$	$ELv5.a = Number(num.lex)$
$ELv5 \rightarrow bool$	$ELv5.a = Boolean(bool.lex)$
$ELv5 \rightarrow ID$	$ELv5.a = ID(ID.lex)$