Procesadores de Lenguajes

Práctica Anual

Ingeniería en Informática 4º A

Facultad de Informática UCM

(2012-2013)

Grupo: 3

Miembros: Marina Bezares Álvarez, Daniel Escoz Solana, Antonio Irízar López, Raúl Marcos Lorenzo, Pedro Morgado Alarcón, Arturo Pareja García.

1. Definición Léxica

Formación de literales e identificadores

```
litnat ≡ _dign0 (_dig)* | "0"
litfloat ≡ litnat _partedec (_parteexp)? | litnat _parteexp
litchar ≡ "'"_alfanum1"'"
ident ≡ _min (_alfanum1)*
true ≡ "true"
false ≡ "false"
```

Palabras reservadas

```
program ≡ "program:"
varconsts ≡ "vars-consts"
instructions ≡ "instructions"
var ≡ "var"
const ≡ "const"
float ≡ "float"
integer ≡ "integer"
int ≡ "int"
boolean ≡ "boolean"
natural ≡ "natural"
nat ≡ "nat"
character ≡ "character"
char ≡ "char"
in ≡ "in"
out ≡ "out"
swap1 ≡ "swap1"
swap2 ≡ "swap2"
```

Símbolos y operadores

```
Asig ≡ "="
dpigual ≡ ":="
lpar ≡ "("
rpar ≡ ")"
illave ≡ "{"
fllave ≡ "}"
pyc ≡ ";"
men ≡ "<"
menoig ≡ "<="
may ≡ ">"
mayoig ≡ ">="
igual ≡ "=="
```

```
noigual ≡ "!="
mas ≡ "+"
menos ≡ "-"
mul ≡ "*"
div ≡ "/"
mod ≡ "%"
and ≡ "and"
or ≡ "or"
not ≡ "not"
lsh ≡ "<<"
rsh ≡ ">>"
```

Expresiones auxiliares

```
_min ≡ ['a'-'z']
_may ≡ ['A'-'Z']
_letra ≡ _min | _may
_dig ≡ ['0'-'9']
_dign0 ≡ ['1'-'9']
_alfanum1 ≡ _letra | _dig
_partedec ≡ "." ((_dig)*_dign0 | "0")
_parteexp ≡ ("e" | "E") "-"? litnat
fin ≡ <end-of-file>
```

Comentarios

La definición de la palabra reservada "`vars-consts`" recibió en su día el nombre de `varconsts` (por error). Se sigue manteniendo dicho nombre en el resto de la memoria.

2. Definición Sintáctica

2.1. Descripción de los operadores

Operador	Nivel de Prioridad	Aridad	Asociatividad
Igualdad (==)	0	2	Ninguna
Desigualdad (!=)	0	2	Ninguna
Menor que (<)	0	2	Ninguna
Menor o igual (<=)	0	2	Ninguna
Mayor que (>)	0	2	Ninguna
Menor o igual (>=)	0	2	Ninguna
Suma (+)	1	2	Izquierdas
Resta (−)	1	2	Izquierdas
Disyunción Lógica (or)	1	2	Izquierdas

División (*)	2	2	Izquierdas
División (/)	2	2	Izquierdas
Módulo (%)	2	2	Izquierdas
Conjunción (and)	2	2	Izquierdas
Despl. Izquierda (<<)	3	2	Derechas
Despl. Derecha (>>)	3	2	Derechas
Negación aritmética (−)	4	1	Si
Negación lógica (not)	4	1	No
Conversión	4	1	No

2.2. Formalización de la sintaxis

```
Program → program ident illave SDecs SInsts fllave fin

SDecs → varconsts illave Decs fllave
Decs → Decs pyc Dec | Dec
Dec → var Type ident | const Type ident dpigual Lit | ε

SInsts → instructions illave Insts fllave
Insts → Insts pyc Inst  | Inst
Inst → ident asig Expr | in lpar ident rpar | out lpar Expr rpar
       | swap1 lpar rpar | swap2 lpar rpar | ε

Type → boolean | character | integer | natural | float
Cast → char | int | nat | float

Expr → Term Op0 Term | Term
Term → Term Op1 Fact | Fact
Fact → Fact Op2 Shft | Shft
Shft → Unary Op3 Shft | Unary
Unary → Op4 Unary | lpar Cast rpar Paren | Paren
Paren →  lpar Expr rpar | Lit | ident

Op0 → igual | noigual | men | may | menoig | mayoig
Op1 → or | menos | mas
Op2 → and | mod | div | mul
Op3 → lsh | rsh
Op4 → not | menos

Lit → LitBool | LitNum | litchar
LitBool → true | false
LitNum → litnat | menos litnat| litfloat | menos litfloat
```

3. Tabla de Símbolos

3.1. Estructura de la tabla de símbolos

Necesitamos definir los siguientes campos de la tabla de símbolos:

- **`id`**: Nombre del identificador.
- **`type`**: Tipo asignado al identificador.
- **`const`**: Determina si el identificador es constante.
- **`dir`**: Dirección de memoria asignada a esta variable. Sólo para variables.
- **`value`**: Valor almacenado de la constante. Sólo para constantes.

3.2. Construcción de la tabla de símbolos

3.2.1. Funciones semánticas

`creaTS() : TS`
Crea una tabla de símbolos vacía.

`añadeID(ts:TS, id:String, type:Tipo, const:Boolean, dir:Int, value:?) : TS`
Dada una tabla de símbolos y un símbolo, devuelve una tabla de símbolos actualizada con un nuevo identificador. El tipo de *value* depende del atributo *type*.

`existeID(ts:TS, id:String) : Boolean`
Dada una tabla de símbolos y el campo id de un identificador, indica si el identificador existe en la tabla de símbolos (sensible a mayúsculas y minúsculas), es decir, ha sido previamente declarado.

`tipoDe(ts:TS, id:String) : Type`
Dada una tabla de símbolos y el campo id de un identificador, devuelve el tipo del identificador. Si no existe, devuelve terr (tipo error)

`stringToNat(v:String) : Natural`
Convierte el atributo pasado como string a un valor natural.

`stringToFloat(v:String) : Float`
Convierte el atributo pasado como string a un valor decimal.

`stringToChar(v:String) : Character`
Convierte el atributo pasado como string a un carácter

3.2.2. Atributos semánticos

- **`ts`**: tabla de símbolos sintetizada.
- **`type`**: tipo de la declaración.
- **`id`**: nombre del identificador.
- **`const`**: true si estamos declarando una constante, falso si no.
- **`dir`**: Dirección de memoria. Definido tanto para constantes como para variables.
- **`value`**: Valor de la constante. En el caso de variables, este atributo no es necesario, pero se define igualmente (con el valor ?) para poder pasarlo a la tabla de símbolos.

3.2.3. Gramática de atributos

A continuación se detalla la construcción de los atributos relevantes para la creación de la tabla de símbolos. Otros atributos, como la tabla de símbolos heredada (que tan solo se propaga) o el tipo y el valor de las expresiones se detallarán más adelante en sus correspondientes secciones.

```
Program → program ident illave SDecs SInsts fllave fin
       SInsts.tsh = SDecs.ts

SDecs → varconsts illave Decs fllave
       SDecs.ts = Decs.ts

Decs → Decs pyc Dec
       Decs0.dir = Decs1.dir + 1
       Decs0.ts = AñadeID( Decs1.ts, Dec.id, Dec.type, Dec.const, Decs0.dir,
Dec.value );

Decs → Dec
       Decs.dir = 0
       Decs.ts = AñadeID( CreaTS(), Dec.id, Dec.type, Dec.const, 0, Dec.value );

Dec → var Type ident
       Dec.type = Type.type
       Dec.id = ident.lex
       Dec.const = false
       Dec.value = ?

Dec → const Type ident dpigual Lit
       Dec.type = Type.type
       Dec.id = ident.lex
       Dec.const = true
       Dec.value = Lit.value

Lit → LitBool
       Lit.value = LitBool.value

Lit → LitNum
       Lit.value = LiNum.value
```

```
Lit → litchar
      Lit.value = stringToChar( litnat.lex )

LitBool → true
      LitBool.value = true

LitBool → false
      LitBool.value = false
LitNum → litnat
      LitNum.value = StringToNat( litnat.lex )

LitNum → menos litnat
      LitNum.value = - StringToNat( litnat.lex )

LitNum → litfloat
      LitNum.value = StringToFloat( litfloat.lex )

LitNum → menos litfloat
      LitNum.value = - StringToFloat( litfloat.lex )
```

4. Restricciones

4.1. Descripción informal de las restricciones contextuales

Enumeración y descripción de las restricciones contextuales extraídas directamente del enunciado.

4.1.1. Sobre declaraciones

- Las variables y constantes que se usen en la sección de instrucciones habrán debido de ser convenientemente declaradas en la sección de declaraciones.
- No se pueden declarar dos variables o constantes con el mismo identificador.

4.1.2. Sobre instrucciones de asignación

Una instrucción de asignación debe cumplir además estas condiciones:

- La variable en la parte izquierda debe haber sido declarada.
- No se pueden asignar o hacer instrucciones in a constantes
- A una variable de tipo real es posible asignarle un valor real, entero o natural (produciéndose automáticamente la correspondiente conversión), pero no un carácter o booleano.
- A una variable de tipo entero es posible asignarle un valor entero o natural (produciéndose automáticamente la correspondiente conversión), pero no un valor real, carácter o booleano.
- A una variable de tipo natural únicamente es posible asignarle un valor natural.
- A una variable de tipo carácter únicamente es posible asignarle un valor de tipo carácter.

- A una variable de tipo booleano únicamente es posible asignarle un valor de tipo booleano.

4.1.3. Sobre comparaciones

No se puede comprar naturales con caracteres, ni enteros con caracteres, ni reales con caracteres, ni booleanos con caracteres. Tampoco se puede comparar naturales con booleanos, ni enteros con booleanos, ni reales con booleanos, ni caracteres con booleanos.

4.1.4. Sobre operadores

- Los operadores +, -, *, / sólo operan con valores numéricos. No podemos aplicarlos ni a los caracteres ni a los booleanos.
- En la operación *módulo* **%** el primer operando puede ser entero o natural, pero el segundo operando sólo puede ser natural. El resultado de *a* **%** *b* será el resto de la división de *a* entre *b*. El tipo del resultado será el mismo que el del primer operando.
- Los operadores lógicos ‘or’, ‘and’, ‘not’ sólo operan sobre valores booleanos. No podemos aplicarlos ni a los numéricos ni a los caracteres.
- Los operadores << y >> sólo operan con valores numéricos naturales.

4.1.5. Sobre operadores de conversión

- **(float)** puede ser aplicado a cualquier tipo excepto al tipo booleano.
- **(int)** puede ser aplicado a cualquier tipo excepto el tipo booleano.
- **(nat)** puede ser aplicado al tipo natural y al tipo carácter. No admite operandos reales, enteros o booleanos..
- **(char)** puede ser aplicado al tipo carácter y al tipo natural. No admite operandos reales, enteros o booleanos.

4.2 Funciones semánticas

A continuación, describimos las funciones semánticas adicionales utilizadas en la descripción.

`casting (Type tipoCast, Type tipoOrg) : Type`

Dados dos tipos diferentes comprobamos si podemos hacer el casting: [`(tipoCast) tipoOrg`] Si podemos, devolvemos el `tipoCast` resultante de hacer el casting, y si no podemos, devolvemos terr. Describimos el comportamiento de la función en la siguiente tabla.

TipoCast	TipoOrg	Tipo devuelto
natural	natural	natural
natural	character	natural
natural	cualquier otro tipo	terr
boolean	—	terr
character	character	character
character	natural	character
character	cualquier otro tipo	terr

integer	boolean	terr
integer	tipo numérico o carácter	integer
float	boolean	terr
float	tipo numérico o carácter	terr

`unario(Type OpUnario, Type tipoUnario) : Type`

Dado un operador unario y el tipo al que es aplicado comprobamos si se puede aplicar. Por ejemplo, no podemos aplicar a un booleano el operador "-". Tampoco podemos aplicar a un entero el operador "not". En esos casos devuelve terr. Si aplicamos el operador "-" a un tipo nat devolvemos el tipo integer.

OpUnario	TipoUnario	Tipo devuelto
"-"	natural	integer
"-"	integer	integer
"-"	float	float
"-"	boolean	terr
"-"	character	terr
not	natural	terr
not	integer	terr
not	float	terr
not	boolean	boolean
not	character	terr

`tipoFunc(Type tipo1, Operator op, Type tipo2) : Type`

Dados dos tipos diferentes y un operador comprobamos que los tipos puedan aplicar el operador. Devolvemos el tipo correspondiente al aplicar el operador. Si el operador no puede ser aplicado entonces devolvemos terr.

Si pusiésemos todas las posibilidades la tabla resultante quedaría muy extensa. Para simplificar, se pondrán dos tablas. En la primera, se pondrán los operadores conmutativos. Es decir, aquellos que se comportan igual sean los tipos asignados al primer parámetro de la función o al segundo. En la segunda se pondrán los operadores no conmutativos. En los que importa quién sea el tipo1 y el tipo2.

También para que se vea mejor, dentro de las tablas, separaremos los tipos de operadores.

Operadores conmutativos:

Tipo1	Op	Tipo2	Tipo devuelto
tipo numérico	cualquier op. de comparación	tipo numérico	boolean
boolean	cualquier op. de comparación	boolean	boolean

character	cualquier op. de comparación	character	boolean
boolean	cualquier op aritmético	—	terr
character	cualquier op aritmético	—	terr
float	cualquier op aritmético	cualquier tipo numérico	float
integer	cualquier op aritmético	integer o natural	integer
natural	cualquier op aritmético	natural	natural
boolean	cualquier op lógico	boolean	boolean
tipo no boolean	cualquier op lógico	—	terr
natural	"<<"	natural	natural
natural	">>"	natural	natural
tipo no natural	"<<"	—	terr
—	">>"	tipo no natural	terr

Operadores no conmutativos:

Tipo1	Op	Tipo2	Tipo devuelto
integer o natural	"%"	natural	natural
—	"%"	tipo no natural	terr
ni integer ni natural	"%"	—	terr

`asignaciónVálida(Type tipoVar, Type tipoExp) :Boolean`

Dado un tipo de una variable y un tipo de una expresión, comprueba si a la variable se le asigna un tipo permitido. Por ejemplo, no podemos asignar a una variable de tipo char una expresión booleana. Si la asignación es incorrecta devolvemos false, si no devolvemos true.

Para que se vea mejor, dentro de las tablas, separaremos los tipos posibles de `tipoVar`.

TipoVar	TipoExp	Tipo devuelto
natural	natural	true
natural	integer o float o boolean o character	false
integer	natural	true
integer	integer	true
integer	float o boolean o character	false
float	tipo numérico	true
float	boolean o character	false

boolean	tipo numérico o character	false
boolean	boolean	boolean
character	tipo numérico o boolean	terr
character	character	character

Nota: En todas las funciones, si alguno de los tipos de entrada es el tipo terr, devolvemos siempre terr.

4.3 Atributos semánticos

- **`type:`** atributo que indica de qué tipo es la variable, expresión etc. Puede tomar los valores boolean, char, nat, int, float y terr.
- **`op:`** atributo que indica cuál es el operador usado.
- **`ts:`** tabla de símbolos. Se crea en la parte de declaraciones.
- **`tsh:`** tabla de símbolos heredada. Se hereda en la parte de instrucciones.
- **`err:`** atributo que indica si se ha detectado algún error. Es un atributo de tipo booleano.

4.4 Gramática de atributos

```
Program → program ident illave SDecs SInsts fllave fin
       SInts.tsh = SDecs.ts
       Program.err = SDecs.err ∨ SInsts err

SDecs → varconsts illave Decs fllave
       SDecs.ts = Decs.ts
       SDecs.err = Decs.err

Decs → Decs pyc Dec
       Decs0.ts = AñadeID( Decs1.ts, Dec.id, Dec.type, Dec.const, Dec.valor )
       Decs0.err = ExisteID(Decs1.ts, Dec.id)
Decs → Dec
       Decs.ts = AñadeID( CreaTS(), Dec.id, Dec.type, Dec.const, Dec.valor )

Dec → var Type ident
Dec → const Type ident dpigual Lit
Dec → ε

SInsts → instructions illave Insts fllave
       Insts.tsh = SInsts.tsh
       SInsts.err = Insts.err
```

```
Insts → Insts pyc Inst
       Inst.tsh = Insts0.tsh
       Insts1.tsh = Insts0.tsh
       Insts0.err = Insts1.err ∨ Inst.err
Insts → Inst
       Inst.tsh = Insts.tsh
       Insts.err = Inst.err

Inst → ident asig Expr
       Expr.tsh = Inst.tsh
       Inst.err = ( ¬asignaciónVálida(Inst.tsh[ident.lex].type, Expr.type)
       ∨¬existeID(Inst.tsh, ident.lex) ∨ Inst.tsh[ident.lex].const = true)
Inst → in lpar ident rpar
       Inst.err = (¬existeID(Inst.tsh, ident.lex) ∨ Inst.tsh[ident.lex].const = true)

Inst → out lpar Expr rpar
       Inst.err = (Expr.type == terr)

Type → boolean
       Type.type = boolean
Type → character
       Type.type = chararacter
Type → integer
       Type.type = integer
Type → natural
       Type.type = natural
Type → float
       Type.type = float

Cast → char
       Cast.type = char
Cast → int
       Cast.type = int
Cast → nat
       Cast.type = nat
Cast → float
       Cast.type = float

Expr → Term Op0 Term
       Expr.type = tipoFunc(Term0.type,Op0.op,Term1.type)
       Term0.tsh = Expr.tsh
       Term1.tsh = Expr.tsh
Expr → Term
       Expr.type = Term.type
       Term.tsh = Expr.tsh
```

```
Term → Term Op1 Fact
       Term0.type = tipoFunc(Term1.type, Op1.op, Fact.type)
       Term1.tsh = Term0.tsh
       Fact.tsh = Term0.tsh
Term → Fact
       Term.type = Fact.type
       Fact.tsh = Term.tsh

Fact → Fact Op2 Shft
       Fact0.type =  tipoFunc(Fact1.type, Op2.op, Shft.type)
       Fact1.tsh = Fact0.tsh
       Shft.tsh = Fact0.tsh
Fact → Shft
       Fact.type = Shft.type
       Shft.tsh = Fact.tsh

Shft → Unary Op3 Shft
       Shft0.type =  tipoFunc(Unary.type, Op3.op, Shft.type)
       Shft1.tsh = Shft0.tsh
       Unary.tsh = Shft0.tsh
Shft → Unary
       Shft.type = Unary.type
       Unary.tsh = Shft.tsh

Unary → Op4 Unary
       Unary0.type = opUnario(Op4.op, Unary1.type)
       Unary1.tsh = Unary0.tsh
Unary → lpar Cast rpar Paren
       Unary.type = casting(Cast.type, Paren.type)
       Paren.tsh = Unary.tsh
Unary → Paren
       Unary.type = Paren.type
       Paren.tsh = Unary.tsh

Paren →  lpar Expr rpar
       Paren.type = Expr.type
       Expr.tsh = Paren.tsh
Paren → Lit
       Parent.type = Lit.type
       Lit.tsh = Paren.tsh
Paren → ident
       Paren.type = tipoDe(ident.lex, Paren.tsh)
```

```
Op0 → igual
       Op0.op = igual
Op0 → noigual
       Op0.op = noigual
Op0 → men
       Op0.op = men
Op0 → may
       Op0.op = may
Op0 → menoig
       Op0.op = menoig
Op0 → mayoig
       Op0.op = mayoig

Op1 → or
       Op1.op = or
Op1 → menos
       Op1.op = menos
Op1 → mas
       Op1.op = mas

Op2 → and
       Op2.op = and
Op2 → mod
       Op2.op = mod
Op2 → div
       Op2.op = div
Op2 → mul
       Op2.op = mul

Op3 → lsh
       Op3.op = lsh
Op3 →rsh
       Op3.op = rsh

Op4 → not
       Op4.op = not
Op4 → menos
       Op4.op = menos

Lit → LitBool
       Lit.type = boolean
Lit → LitNum
       Lit.type = LitNum.type
Lit → litchar
       Lit.type = char

LitNum → litnat
       LitNum.type = natural
LitNum → menos litnat
       LitNum.type = integer
```

```
LitNum → litfloat | menos litfloat
       LitNum.type = float
```

5. Especificación de la traducción

5.1. Lenguaje objeto y máquina virtual

5.1.1. Arquitectura

- **Mem**: Memoria principal con celdas direccionables con datos. Los datos de la memoria incluyen información sobre *de qué tipo son*.
- **Prog**: Memoria de programa con celdas direccionables con instrucciones.
- **CProg**: Contador de programa con un registro para la dirección de la instrucción actualmente en ejecución
- **Pila**: Pila de datos con celdas direccionables con datos. Al igual que en la memoria, se incluye información sobre el tipo.
- **CPila**: Cima de la pila de datos con un registro para la dirección del dato situado actualmente en la cima de la pila.
- **P:** Flag de parada que detiene la ejecución si tiene valor 1.
- **S1**: Flag de swap1. Si tiene valor 1 intercambia suma por resta y viceversa.
- **S2**: Flag de swap2. Si tiene valor 1 intercambia multiplicación por división y viceversa.

5.1.2. Comportamiento interno

Pseudocódigo del algoritmo de su ejecución:

```
CPila ← -1
CProg ← 0
S1 ← 0
S2 ← 0
P ← 0
mientras P = 0
       ejecutar Prog[CProg]
fmientras
```

- `Mem[dirección]:` Dato de una celda de memoria principal localizado a través de una dirección.
- `Prog[dirección]:` Instrucción de una celda de memoria de programa localizado a través de una dirección.

La dirección *-1* en `CPila` indica que la pila está vacía.

5.1.3. Repertorio de instrucciones

Operaciones con la Pila

apila(valor)

```
CPila ← CPila + 1
Pila[CPila] ← valor
CProg ← CProg + 1
```

apila-dir(dirección)

```
CPila ← CPila + 1
Pila[CPila] ← Mem[dirección]
CProg ← CProg + 1
```

Nota: Si la dirección de memoria no ha sido cargada previamente con datos usando la siguiente instrucción (desapila-dir), esta instrucción dará un error de ejecución.

desapila-dir(dirección)

```
Mem[dirección] ← Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

Operaciones aritméticas

mas

```
si S1 = 0: Pila[CPila - 1] ← Pila[CPila - 1] + Pila[CPila]
si S1 = 1: Pila[CPila - 1] ← Pila[CPila - 1]  - Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

menos (binario)

```
si S1 = 0: Pila[CPila - 1] ← Pila[CPila - 1] - Pila[CPila]
si S1 = 1: Pila[CPila - 1] ← Pila[CPila - 1] + Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

mul

```
si S2 = 0: Pila[CPila - 1] ← Pila[CPila - 1] * Pila[CPila]
si S2 = 1: Pila[CPila - 1] ← Pila[CPila - 1] / Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

div

```
si S2 = 0: Pila[CPila - 1] ← Pila[CPila - 1] / Pila[CPila]
```

```
si S2 = 1: Pila[CPila - 1] ← Pila[CPila - 1] * Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

mod

```
Pila[CPila - 1] ← Pila[CPila - 1] % Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

menos (unario)

```
Pila[CPila] ← - Pila[CPila]
CProg ← CProg + 1
```

Operaciones de desplazamiento

lsh

```
Pila[CPila - 1] ← Pila[CPila - 1] << Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

rsh

```
Pila[CPila - 1] ← Pila[CPila - 1] >> Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

Operaciones de comparación

igual

```
Pila[CPila - 1] ← Pila[CPila - 1] == Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

noigual

```
Pila[CPila - 1] ← Pila[CPila - 1] != Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

may

```
Pila[CPila - 1] ← Pila[CPila - 1] > Pila[CPila]
```

```
CPila ← CPila - 1
CProg ← CProg + 1
```

men

```
Pila[CPila - 1] ← Pila[CPila - 1] < Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

mayoig

```
Pila[CPila - 1] ← Pila[CPila - 1] >= Pila[CPila]
CPila ← CPila - 1
CProg ← CPoprog + 1
```

menoig

```
Pila[CPila - 1] ← Pila[CPila - 1] <= Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

Operaciones lógicas

and

```
Pila[CPila - 1] ← Pila[CPila - 1] && Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

or

```
Pila[CPila - 1] ← Pila[CPila - 1] || Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

not

```
Pila[CPila] ← ! Pila[CPila]
CProg ← CProg + 1
```

Operaciones de conversión

castFloat

```
Pila[CPila] ← (float) Pila[CPila]
CProg ← CProg + 1
```

castInt

```
Pila[CPila] ← (int) Pila[CPila]
CProg ← CProg + 1
```

castNat

```
Pila[CPila] ← (nat) Pila[CPila]
CProg ← CProg + 1
```

castChar

```
Pila[CPila] ← (char) Pila[CPila]
CProg ← CProg + 1
```

Operaciones de Entrada-Salida

in(type)

```
CPila ← CPila + 1
Pila[CPila] ← Leer un valor de tipo type de BufferIN
CProg ← CProg + 1
```

out

```
Escribir en BufferOUT ← Pila[CPila]
CPila ← CPila - 1
CProg ← CProg + 1
```

Operaciones de intercambio

swap1

```
si S1 = 0: S1 ← 1
si S1 = 1: S1 ← 0
```

swap2

```
si S2 = 0: S2 ← 1
si S2 = 1: S2 ← 0
```

Otras operaciones

stop

```
P ← 1
```

Consideraciones sobre "Repertorio de instrucciones"

En la operación castNat, hemos creado la operación en la máquina virtual (nat), que no está predefinida en Java, pero cuyo comportamiento está definido en las tablas correspondientes a los tipos definidos.

5.2. Funciones semánticas

No hacemos uso de ninguna función semántica.

5.3. Atributos semánticos

- **cod**: Atributo sintetizado de generación de código.
- **op**: Enumerado que nos dice cuál es el operador utilizado.

5.4. Gramática de atributos

Gramática de atributos que formaliza la traducción.

```
Program → program ident illave SDecs SInsts fllave fin
       Program.cod = SInsts.cod || stop

SInsts → instructions illave Insts fllave
       SInst.cod = Inst.cod

Insts → Inst
       Insts.cod = Inst.cod
Insts → Insts pyc Inst
       Insts0.cod = Insts1.cod || Inst.cod

Inst → ident asig Expr
       Inst.cod = Expr.cod || desapila-dir(Inst.tsh[ident.lex].dir)
Inst → in lpar ident rpar
       Inst.cod = in(Inst.tsh[ident.lex].type) || desapila-
dir(Inst.tsh[ident.lex].dir)

Inst →  out lpar Expr rpar
       Inst.cod = Expr.cod || out
Inst → swap1 lpar rpar
       Inst.cod = swap1
Inst →  swap2 lpar rpar
       Inst.cod = swap2


Expr → Term Op0 Term
       Expr.cod = Term1.cod || Term2.cod || Op0.op
Expr → Term
       Expr.cod = Term.cod

Term → Term Op1 Fact
```

```
       Term0.cod = Term1.cod || Fact.cod || Op1.op
Term → Fact
       Term.cod = Fact.cod

Fact → Fact Op2 Shft
       Fact0.cod = Fact1.cod || Shft.cod || Op2.op
Fact → Shft
       Fact.cod → Shft.cod

Shft → Unary Op3 Shft
       Shft0.cod = Unary.cod || Shft1.cod || Op3.op
Shft → Unary
       Shft.cod = Unary.cod

Unary → Op4 Unary
       Unary0.cod = Unary1.cod || Op4.op
Unary → Lpar Cast rpar Paren
       Unary.cod = Paren.cod || Cast.type
Unary → Paren
       Unary.cod = Paren.cod

Paren → lpar Expr rpar
       Paren.cod = Expr.cod
Paren → Lit
       Paren.cod = apilar( Lit.value )
Paren → ident
       Paren.cod = apila-dir(Paren.tsh[ident.lex].dir)

Cast → char
       Cast.type = char
Cast → int
       Cast.type = int
Cast → nat
       Cast.type = nat
Cast → float
       Cast.type = float

Op0 → igual
       Op0.op = igual
Op0 → noigual
       Op0.op = noigual
Op0 → men
       Op0.op = men
Op0 → may
       Op0.op = may
Op0 → menoig
       Op0.op =  menoig
```

```
Op0 → mayoig
       Op0.op = mayoig

Op1 → or
       Op1.op = or
Op1 → menos
       Op1.op = menos
Op1 → mas
       Op1.op = mas

Op2 → and
       Op2.op = and
Op2 → mod
       Op2.op = mod
Op2 → div
       Op2.op = div
Op2 → mul
       Op2.op = mul

Op3 → lsh
       Op3.op = lsh
Op3 → rsh
       Op3.op = rsh

Op4 → not
       Op4.op = not
Op4 → menos
       Op4.op = menos
```

Anexo I: Gramática de atributos unificada

A continuación se detalla la gramática de atributos unificada a partir de las gramáticas de los apartados 2, 3, 4 y 5. Esta gramática es la que se utilizará como base para las transformaciones del resto de la práctica.

```
Program → program ident illave SDecs SInsts fllave fin
       SInts.tsh = SDecs.ts
       Program.err = SDecs.err ∨ SInsts.err
       Program.cod = SInsts.cod || stop

SDecs → varconsts illave Decs fllave
       SDecs.ts = Decs.ts
       SDecs.err = Decs.err
```

```
Decs → Decs pyc Dec
       Decs0.ts = AñadeID( Decs1.ts, Dec.id, Dec.type, Dec.const, Dec.value )
       Decs0.err = ExisteID(Decs1.ts,Dec.id)
Decs → Dec
       Decs.ts = AñadeID( CreaTS(), Dec.id, Dec.type, Dec.const, Dec.value )

Dec → var Type ident
       Dec.type = Type.type
       Dec.id = ident.lex
       Dec.const = false
       Dec.value = ?
Dec → const Type ident dpigual Lit
       Dec.type = Type.type
       Dec.id = ident.lex
       Dec.const = true
       Dec.value = Lit.value
Dec → ε

SInsts → instructions illave Insts fllave
       Insts.tsh = SInsts.tsh
       SInsts.err = Insts.err
       SInst.cod = Inst.cod

Insts → Insts pyc Inst
       Inst.tsh = Insts0.tsh
       Insts1.tsh = Insts0.tsh
       Insts0.err = Insts1.err ∨ Inst.err
       Insts0.cod = Insts1.cod || Inst.cod
Insts → Inst
       Inst.tsh = Insts1.tsh
       Insts.err = Inst.err
       Insts.cod = Inst.cod

Inst → ident asig Expr
       Expr.tsh = Inst.tsh
       Inst.err = ( ¬asignaciónVálida(Inst.tsh[ident.lex].type, Expr.type)
       ∨¬existeID(Inst.tsh, ident.lex) ∨ Inst.tsh[ident.lex].const = true)
       Inst.cod = Expr.cod || desapila-dir(Inst.tsh[ident.lex].dir)
Inst → in lpar ident rpar
       Inst.err = (¬existeID(Inst.tsh, ident.lex) ∨ Inst.tsh[ident.lex].const = true)

       Inst.cod = in(Inst.tsh[ident.lex].type) || desapila-
dir(Inst.tsh[ident.lex].dir)

Inst → out lpar Expr rpar
       Inst.err = (Expr.type == terr)
       Inst.cod = Expr.cod || out
Inst → swap1 lpar rpar
```

```
       Inst.cod = swap1
Inst →  swap2 lpar rpar
       Inst.cod = swap2


Type → boolean
       Type.type = boolean
Type → character
       Type.type = character
Type → integer
       Type.type = integer
Type → natural
       Type.type = natural
Type → float
       Type.type = float

Cast → char
       Cast.type = char
Cast → int
       Cast.type = int
Cast → nat
       Cast.type = nat
Cast → float
       Cast.type = float

Expr → Term Op0 Term
       Expr.type = tipoFunc(Term0.type,Term1.type)
       Term0.tsh = Expr.tsh
       Term1.tsh = Expr.tsh
       Expr.cod = Term1.cod || Term2.cod || Op0.op
Expr → Term
       Expr.type = Term.type
       Term.tsh = Expr.tsh
       Expr.cod = Term.cod

Term → Term Op1 Fact
       Term0.type = tipoFunc(Term1.type, Op1.op, Fact.type)
       Term1.tsh = Term0.tsh
       Fact.tsh = Term0.tsh
       Term0.cod = Term1.cod || Fact.cod || Op1.op
Term → Fact
       Term.type = Fact.type
       Fact.tsh = Term.tsh
       Term.cod = Fact.cod
```

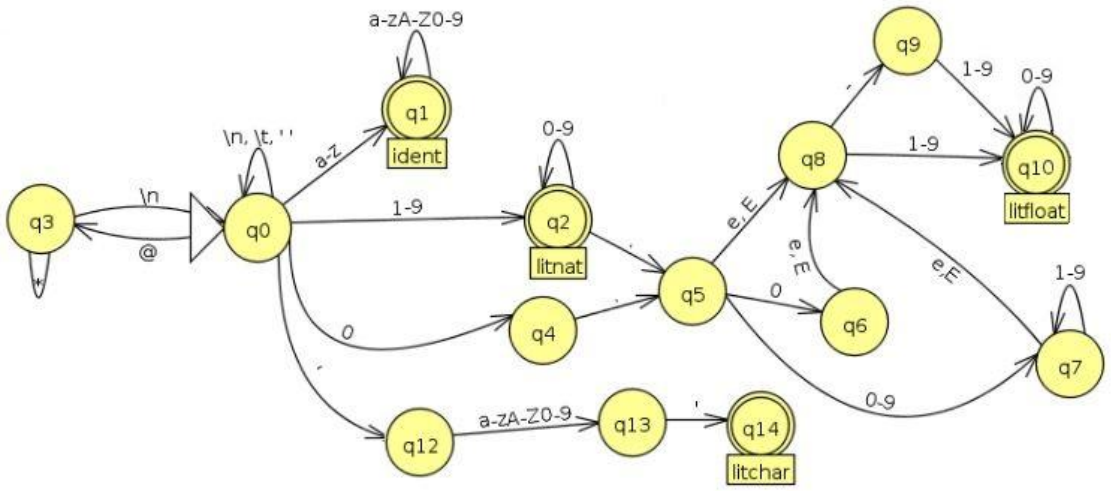
```
Fact → Fact Op2 Shft
       Fact0.type =  tipoFunc(Fact1.type, Op2.op, Shft.type)
       Fact1.tsh = Fact0.tsh
       Shft.tsh = Fact0.tsh
       Fact0.cod = Fact1.cod || Shft.cod || Op2.op
Fact → Shft
       Fact.type = Shft.type
       Shft.tsh = Fact.tsh
       Fact.cod = Shft.cod

Shft → Unary Op3 Shft
       Shft0.type =  tipoFunc(Unary.type, Op3.op, Shft.type)
       Shft1.tsh = Shft0.tsh
       Unary.tsh = Shft0.tsh
       Shft0.cod = Unary.cod || Shft1.cod || Op3.op
Shft → Unary
       Shft.type = Unary.type
       Unary.tsh = Shft.tsh
       Shft.cod = Unary.cod

Unary → Op4 Unary
       Unary0.type = opUnario(Op4.op, Unary1.type)
       Unary1.tsh = Unary0.tsh
       Unary0.cod = Unary1.cod || Op4.op
Unary → lpar Cast rpar Paren
       Unary.type = casting(Cast.type, Paren.type)
       Paren.tsh = Unary.tsh
       Unary.cod = Paren.cod || Cast.type
Unary → Paren
       Unary.type = Paren.type
       Paren.tsh = Unary.tsh
       Unary.cod = Paren.cod

Paren →  lpar Expr rpar
       Paren.type = Expr.type
       Expr.tsh = Paren.tsh
       Paren.cod = Expr.cod
Paren → Lit
       Parent.type = Lit.type
       Lit.tsh = Paren.tsh
       Paren.cod = apilar(Lit.value)
Paren → ident
```

```
        Paren.type = tipoDe(ident.lex, Paren.tsh)
        Paren.cod =
                Si Paren.tsh[ident.lex].const = true
                        apila(Paren.tsh[ident.lex].value)
                Si no
                        apila-dir(Paren.tsh[ident.lex].dir)

Op0 → igual
        Op0.op = igual
Op0 → noigual
        Op0.op = noigual
Op0 → men
        Op0.op = men
Op0 → may
        Op0.op = may
Op0 → menoig
        Op0.op = menoig
Op0 → mayoig
        Op0.op = mayoig
Op1 → or
        Op1.op = or
Op1 → menos
        Op1.op = menos
Op1 → mas
        Op1.op = mas


Op2 → and
        Op2.op = and
Op2 → mod
        Op2.op = mod
Op2 → div
        Op2.op = div
Op2 → mul
        Op2.op = mul
Op3 → lsh
        Op3.op = lsh
Op3 → rsh
        Op3.op = rsh
Op4 → not
        Op4.op = not
Op4 → menos
        Op4.op = menos

Lit → LitBool
        Lit.type = LitBool.type
```

```
        Lit.value = LitBool.value

Lit → LitNum
        Lit.type = LitNum.type
        Lit.value = LitNum.value

Lit → litchar
        Lit.type = char
        Lit.value = stringToChar( litchar.lex )

LitBool → true
        LitBool.type = boolean
        LitBool.value = true
LitBool → false
        LitBool.type = boolean
        LitBool.value = false

LitNum → litnat
        LitNum.type = natural
        LitNum.value = StringToNat( litnat.lex )
LitNum → menos litnat
        LitNum.type = integer
        LitNum.value = - StringToNat( litnat.lex )
LitNum → litfloat
        LitNum.type = float
        LitNum.value = StringToFloat( litfloat.lex )
LitNum → menos litfloat
        LitNum.type = float
        LitNum.value = - StringToFloat( litfloat.lex )
```

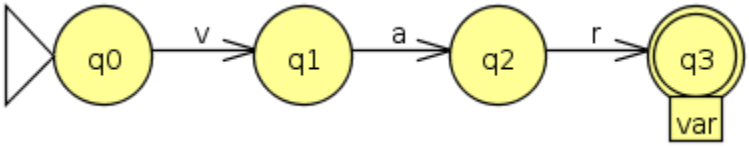
6. Diseño del analizador léxico

La siguiente máquina de estados implementa la parte del analizador léxico que se encarga de reconocer los literales naturales, flotantes y caracteres, así como identificadores.



***Nota**: Las transiciones (q0 → q12) y (q13 → q14) se realizan con comillas simples, y las de (q2 → q5) y (q4 → q5) con puntos.*

Para las palabras reservadas y los símbolos (que al final no son más que palabras reservadas que no usan letras ni números) se utilizarían autómatas similares al siguiente:



Por motivos obvios, no incluimos estas partes del autómata en el general: No hay forma de hacer que quepan o se vean bien en una visión global, ni que decir que la complejidad del autómata es suficiente tal y como está.
A la hora de realizar la implementación de este autómata, hay que tener algunas cosas en cuenta:

- Al reconocer un identificador, debemos seguir leyendo caracteres hasta que aparezca un blanco o un símbolo.
- Al reconocer un número natural, debemos comprobar que no aparece tras él un punto o una letra E (mayúscula o minúscula), ya que esto nos haría reconocer un natural donde en realidad hay un flotante.
- Al leer una palabra reservada, debemos seguir leyendo caracteres si los hubiera hasta encontrar un espacio o símbolo, para evitar reconocer como palabras reservadas lo que en realidad son identificadores.
- Al terminar cualquier reconocimiento, se ha de volver al estado inicial, *releyendo cualquier carácter que se haya comprobado después del propio token, como espacios o símbolos, puesto*

Todas estas consideraciones se tendrán en cuenta y se explicará más adelante (en el punto 11.2) cómo se han solventado.

7. Acondicionamiento de las gramáticas de atributos

Transformaciones realizadas sobre las gramáticas de atributos para permitir la traducción predictivo-recursiva.

7.1. Acondicionamiento de la gramática para la construcción de la tabla de sím-

bolos

Dejamos este apartado en blanco, como se indica en la plantilla, ya que nuestra gramática no necesita acondicionamiento en este apartado.

7.2. Acondicionamiento de la gramática para la comprobación de las Restricciones Contextuales

Dejamos este apartado en blanco, como se indica en la plantilla, ya que nuestra gramática no necesita acondicionamiento en este apartado.

7.3. Acondicionamiento de la gramática para la traducción

En este apartado realizamos dos tipos de acondicionamiento sobre la gramática del anexo I. Primero factorizamos las recursiones necesarias y, segundo, eliminamos la recursión a izquierda de las producciones necesarias.

7.3.1. Factorización de la gramática

Producción original	Producción factorizada
Expr → Term Op0 Term **Expr.type = tipoFunc(Term0.type, Op0, Term1.type)** **Term0.tsh = Expr.tsh** **Term1.tsh = Expr.tsh** **Expr.cod = Term1.cod || Term2.cod || Op0.op** **Expr → Term** **Expr.type = Term.type** **Term.tsh = Expr.tsh** **Expr.cod = Term.cod**	Expr→ Term FExpr FExpr.typeh = Term.type Expr.type = FExpr.type Term.tsh = Expr.tsh FExpr.tsh = Expr.tsh Expr.cod = Term.cod || FExpr.cod FExpr→ Op0 Term FExpr.type = tipoFunc(FExpr.typeh,Op0,Term.type) Term.tsh = FExpr.tsh FExpr.cod = Term.cod || Op0.op FExpr→ ε FExpr.type = FExpr.typeh FExpr.cod = ε
Shft → Unary Op3 Shft **Shft0.type = tipoFunc(Unary.type, Op3.op, Shft.type)** **Shft1.tsh = Shft0.tsh** **Unary.tsh = Shft0.tsh** **Shft0.cod = Unary.cod || Shft1.cod || Op3.op** **Shft → Unary** **Shft.type = Unary.type** **Unary.tsh = Shft.tsh** **Shft.cod = Unary.cod**	Shft→ Unary FShft FShft.typeh = Unary.type Shft.type = FShft.type Unary.tsh = Shft.tsh FShft.tsh = Shft.tsh Shft.cod = Unary.cod || FShft.cod FShft→ Op3 Shft FShft.type = tipoFunc(FShft.typeh, Op3.op, Shft.type) Shft.tsh = FShft.tsh FShft.cod = Shft.cod || Op3.op FShft→ ε FShft.type = FShft.typeh

	FShft.cod = ε
LitNum → menos litnat **LitNum.type = integer** **LitNum.value = - StringToNat(litnat.lex)** **LitNum → menos litfloat** **LitNum.type = float** **LitNum.value = - StringToFloat(litfloat.lex)**	LitNum → menos FLitNum LitNum.type = FLitNum.type LitNum.value = FLitNum.value FLitNum → litnat FLitNum.type = integer FLitNum.value = -StringToNat(litnat.lex) FLitNum → litfloat FLitNum.type = float FLitNum.value = - StringToFloat(litfloat.lex)

7.3.2 Eliminación de la recursión a izquierdas de la gramática

Producción original	Producción sin recursión a izquierdas
Decs → Decs pyc Dec **Decs0.ts = AñadeID(Decs1.ts, Dec.id, Dec.type, Dec.const, Dec.value)** **Decs0.err = ExisteID(Decs1.ts,Dec.id)** **Decs → Dec** **Decs.ts = AñadeID(CreaTS(), Dec.id, Dec.type, Dec.const, Dec.value)**	Decs → Dec RDecs RDecs.tsh = AñadeID(CreaTS(), Dec.id, Dec.type, Dec.const, Dec.value) Decs.ts = RDecs.ts Decs.err=RDecs.err RDecs → pyc Dec RDecs RDecs1.tsh = AñadeID(RDecs0.tsh, Dec.id, Dec.type, Dec.const, Dec.value) RDecs0.ts = RDecs1.ts RDecs1.errh=ExisteID(RDecs0.ts$_h$,Dec.id) RDecs0.err = RDecs1.err RDecs → ε RDecs.ts = RDecs.tsh RDecs.err = RDecs.errh
Insts → Insts pyc Inst **Inst.tsh = Insts0.tsh** **Insts1.tsh = Insts0.tsh** **Insts0.err = Insts1.err V Inst.err** **Insts0.cod = Insts1.cod || Inst.cod** **Insts → Inst** **Inst.tsh = Insts1.tsh** **Insts.err = Inst.err** **Insts.cod = Inst.cod**	Insts → Inst RInsts RInsts.tsh = Inst.tsh = Insts.tsh RInsts.errh = Inst.err Insts.err = RInsts.err RInsts.codh = Inst.cod Insts.cod = RInsts.cod RInsts → pyc Inst RInsts RInsts1.tsh = Inst.tsh = RInsts0.tsh RInsts1.errh = RInsts0.errh v Inst.err RInsts0.err = RInsts1.err RInsts1.codh = RInsts0.codh || Inst.cod RInsts0.cod = RInsts1.cod RInsts → ε RInsts.err = RInsts.errh RInsts.cod = RInsts.codh
	Term → Fact RTerm

Term → Term Op1 Fact **Term0.type = tipoFunc(Term1.type, Op1.op, Fact.type)** **Term1.tsh = Term0.tsh** **Fact.tsh = Term0.tsh** **Term0.cod = Term1.cod || Fact.cod || Op1.op** **Term → Fact** **Term.type = Fact.type** **Fact.tsh = Term.tsh** **Term.cod = Fact.cod**	RTerm.tsh = Fact.tsh = Term.tsh RTerm.typeh = Fact.type Term.type = RFact.type RTerm.codh = Fact.cod Term.cod = RFact.cod RTerm → Op1 Fact RTerm RTerm1.tsh = Fact.tsh = RTerm0.tsh RTerm1.typeh = tipoFunc(RTerm0.typeh, Op1.op, Fact.type) RTerm0.type = RTerm1.type RTerm1.codh = Term0.codh || Fact.cod || Op1.op RTerm0.cod =RTerm1.cod RTerm → ε RTerm.type = RTerm.typeh RTerm.cod = RTerm.codh
Fact → Fact Op2 Shft **Fact0.type = tipoFunc(Fact1.type, Op2.op, Shft.type)** **Fact1.tsh = Fact0.tsh** **Shft.tsh = Fact0.tsh** **Fact0.cod = Fact1.cod || Shft.cod || Op2.op** **Fact → Shft** **Fact.type = Shft.type** **Shft.tsh = Fact.tsh** **Fact.cod = Shft.cod**	Fact → Shft RFact RFact.tsh = Shft.tsh = Fact.tsh RFact.typeh = Shft.type Fact.type = RFact.type RFact.codh = Shft.cod Fact.cod = RFact.cod RFact → Op2 Shft RFact RFact1.tsh = Shft.tsh = Fact0.tsh RFact1.typeh = tipoFunc(RFact0.typeh, Op2.op, Shft.type) RFact0.type = RFact1.type RFact1.codh = RFact0.codh || Shft.cod || Op2.op RFact0.cod = RFact1.cod RFact → ε RFact.type = RFact.typeh RFact.cod = RFact.codh

8. Esquema de traducción orientado a las gramáticas de atributos

A continuación se detalla la gramática de atributos transformada a partir de las gramáticas del . Esta gramática Anexo I y los dos puntos anteriores a este. Es la que se utilizará como base para la implementación.

Program → program ident illave SDecs
{ SInts.tsh = SDecs.ts }
SInsts
{ Program.err = SDecs.err ∨ SInsts.err

```
      Program.cod = SInsts.cod || stop }
      fllave fin

SDecs → varconsts illave Decs
      { SDecs.ts = Decs.ts
      SDecs.err = Decs.err }
      fllave

Decs → Dec
      { RDecs.tsh = AñadeID(CreaTS(), Dec.id, Dec.type, Dec.const, Dec.value ) }
      RDecs
      { Decs.ts = RDecs.ts
      Decs.err=RDecs.err }

RDecs → pyc Dec
       { RDecs1.tsh = AñadeID( RDecs0.tsh, Dec.id, Dec.type, Dec.const, Dec.value )
      RDecs1.errh=ExisteID(RDecs0.ts,Dec.id)}
       RDecs
      { RDecs0.ts = RDecs1.ts
      RDecs0.err = RDecs1.err}

RDecs → ε
      { RDecs.ts = RDecs.tsh
      RDecs.err = RDecs.errh}

Dec → var Type ident
      { Dec.const = false
      Dec.value = ?
      Dec.type = Type.type
      Dec.id = ident.lex }

Dec → const Type ident
      { Dec.const = true
      Dec.type = Type.type
       Dec.id = ident.lex }
       dpigual Lit
       { Dec.value = Lit.value }

Dec → ε

SInsts → instructions
       { Insts.tsh = SInsts.tsh }
       illave Insts
       { SInsts.err = Insts.err
       SInst.cod = Inst.cod }
        fllave

Insts → { Inst.tsh = Insts.tsh }
```

```
       Inst
      { RInsts.tsh = Inst.tsh
       RInsts.errh = Inst.err
       RInsts.codh = Inst.cod }
      RInsts
      { Insts.err = RInsts.err
      Insts.cod = RInsts.cod }

RInsts → pyc
        {Inst.tsh = RInsts0.tsh }
         Inst
        { RInsts1.tsh = Inst.tsh
        RInsts1.errh =   RInsts0.errh v Inst.err
        RInsts1.codh =  RInsts0.codh || Inst.cod }
        RInsts
        { RInsts0.cod =  RInsts1.cod
        RInsts0.err = RInsts1.err }

RInsts → ε
         {  RInsts.err  =  RInsts.errh
         RInsts.cod =  RInsts.codh }

Inst → ident asig
       { Expr.tsh = Inst.tsh }
       Expr
      { Inst.err = ( ¬asignaciónVálida(Inst.tsh[ident.lex].type, Expr.type)
      V¬existeID(Inst.tsh, ident.lex) V Inst.tsh[ident.lex].const = true)
      Inst.cod = Expr.cod || desapila-dir(Inst.tsh[ident.lex].dir) }

Inst → in lpar ident
      { Inst.err = (¬existeID(Inst.tsh, ident.lex) V Inst.tsh[ident.lex].const =
      true)
      Inst.cod = in(Inst.tsh[ident.lex].type) || desapila-
      dir(Inst.tsh[ident.lex].dir) }
      Rpar
Inst → out lpar Expr
       {Inst.err = (Expr.type == terr)
       Inst.cod = Expr.cod || out}
       rpar

Inst → swap1 lpar rpar
       { Inst.cod = swap1 }

Inst →  swap2 lpar rpar
      {Inst.cod = swap2 }

Type → boolean
       { Type.type = boolean }

Type → character
       { Type.type = character }
```

```
Type → integer
       { Type.type = integer }

Type → natural
       { Type.type = natural }

Type → float
       { Type.type = float }

Cast → char
       { Cast.type = char }

Cast → int
       { Cast.type = int }

Cast → nat
       { Cast.type = nat }

Cast → float
       { Cast.type = float }

Expr→  { Term.tsh = Expr.tsh }
       Term
       { FExpr.typeh = Term.type
       FExpr.tsh = Expr.tsh }
       FExpr
       { Expr.type = FExpr.type
       Expr.cod = Term.cod || FExpr.cod }

FExpr→ Op0
       { Term.tsh = FExpr.tsh }
       Term
       { FExpr.type = tipoFunc(FExpr.typeh,Op0, Term.type)
       FExpr.cod = Term.cod || Op0.op }

FExpr→  ε
       { FExpr.type = FExpr.typeh
       FExpr.cod = ε }

Term → { Fact.tsh = Term.tsh }
       Fact
       { RTerm.tsh = Fact.tsh
       RTerm.typeh = Fact.type
       RTerm.codh = Fact.cod }
       RTerm
       { Term.type = RTerm.type
       Term.cod = RTerm.cod }

RTerm → Op1
       { Fact.tsh = RTerm0.tsh }
       Fact
```

```
          { RTerm1.tsh = Fact.tsh
          RTerm1.typeh = tipoFunc(RTerm0.typeh, Op1.op, Fact.type)
          RTerm1.codh = RTerm0.codh || Fact.cod || Op1.op }
          RTerm
          { RTerm0.type = RTerm1.type
          RTerm0.cod =RTerm1.cod }

RTerm → ε
       { RTerm.type = RTerm.typeh
       RTerm.cod = RTerm.codh }

Fact → { Shft.tsh = Fact.tsh }
       Shft
       { RFact.tsh = Shft.tsh
       RFact.typeh = Shft.type
       RFact.codh = Shft.cod }
       RFact
       { Fact.type = RFact.type
      Fact.cod = RFact.cod }

RFact →Op2
       { Shft.tsh  = RFact0.tsh }
       Shft
       { RFact1.tsh = Shft.tsh
       RFact1.typeh = tipoFunc(RFact0.typeh, Op2.op, Shft.type)
       RFact1.codh = RFact0.codh || Shft.cod || Op2.op }
       RFact
       { RFact0.type = RFact1.type
       RFact0.cod = RFact1.cod }

RFact → ε
        { RFact.type = RFact.typeh
        RFact.cod = RFact.codh }

Shft→  { Unary.tsh = Shft.tsh }
       Unary
       { FShft.typeh = Unary.type
       FShft.tsh = Shft.tsh }
       FShft
       { Shft.type = FShft.type
       Shft.cod = Unary.cod || FShft.cod }

FShft→  Op3
       { Shft.tsh = FShft.tsh }
       Shft
       { FShft.type = tipoFunc(FShft.typeh, Op3.op, Shft.type)
       FShft.cod = Shft.cod || Op3.op }
```

```
FShft→ ε
       { FShft.type = FShft.typeh
       FShft.cod =  ε }

Unary → Op4
       { Unary1.tsh = Unary0.tsh }
       Unary
       { Unary0.type = opUnario(Op4.op, Unary1.type)
       Unary0.cod = Unary1.cod || Op4.op }

Unary → lpar Cast rpar
       { Paren.tsh = Unary.tsh }
       Paren
       { Unary.type = casting(Cast.type, Paren.type)
       Unary.cod = Paren.cod || Cast.type }

Unary →{ Paren.tsh = Unary.tsh }
       Paren
       { Unary.type = Paren.type
       Unary.cod = Paren.cod }

Paren →  lpar
       { Expr.tsh = Paren.tsh }
       Expr
       { Paren.type = Expr.type
       Paren.cod = Expr.cod }
       rpar

Paren →{ Lit.tsh = Paren.tsh }
       Lit
       { Parent.type = Lit.type
       Paren.cod = apilar(Lit.value) }

Paren → ident
       { Paren.type = tipoDe(ident.lex, Paren.tsh)
       Paren.cod =
              Si Paren.tsh[ident.lex].const = true
                     apila(Paren.tsh[ident.lex].value)
              Si no
                     apila-dir(Paren.tsh[ident.lex].dir)
        }

Op0 → igual
       { Op0.op = igual }

Op0 → noigual
       { Op0.op = noigual }
```

```
Op0 → men
       { Op0.op = men }

Op0 → may
       { Op0.op = may }

Op0 → menoig
       { Op0.op = menoig }

Op0 → mayoig
       { Op0.op = mayoig }

Op1 → or
       { Op1.op = or }

Op1 → menos
       { Op1.op = menos }

Op1 → mas
       { Op1.op = mas }

Op2 → and
       { Op2.op = and }

Op2 → mod
       { Op2.op = mod }

Op2 → div
       { Op2.op = div }

Op2 → mul
       { Op2.op = mul }

Op3 → lsh
       { Op3.op = lsh }

Op3 →rsh
       { Op3.op = rsh }

Op4 → not
       { Op4.op = not }

Op4 → menos
       { Op4.op = menos }

Lit → LitBool
       { Lit.type = LitBool.type
       Lit.value = LitBool.value }

Lit → LitNum
       { Lit.type = LitNum.type
       Lit.value = LitNum.value}
```

```
Lit → litchar
       { Lit.type = char
       Lit.value = stringToChar( litchar.lex ) }

LitBool → true
        { LitBool.type = boolean
       LitBool.value = true }

LitBool → false
       { LitBool.type = boolean
       LitBool.value = false }


LitNum → litnat
       { LitNum.type = natural
       LitNum.value = StringToNat( litnat.lex ) }

LitNum → litfloat
       { LitNum.type = float
       LitNum.value = StringToFloat( litfloat.lex ) }

LitNum → menos FLitNum
       { LitNum.type = FLitNum.type
       LitNum.value = FLitNum.value }

FLitNum → litnat
       { FLitNum.type = integer
       FLitNum.value = -StringToInt( litnat.lex ) }

FLitNum → litfloat
       { FLitNum.type = float
       FLitNum.value = -StringToFloat( litfloat.lex ) }
```

9. Esquema de traducción orientado al traductor predictivo – recursivo

9.1. Variables globales

- ts: la tabla de símbolos se hace global ya que se necesita en todas las producciones.
- cod: variable donde se va concatenando el código.
- err: variable donde se van concatenando todos los errores.

9.2. Nuevas operaciones y transformación de las ecuaciones semánticas

- emite(instruccion/es): función que se encarga de ir concatenando las instrucciones del código en la variable global cod.

9.3. Esquema de la traducción

```
Program(out err0, cod0) ::= program ident illave
                    SDecs(out ts1, err1)
                    { tsh2 ← ts1 }
                    SInts(in tsh2, out err2, cod2)
                    {err0 ← err1 ∨ err2, cod0 ← cod2 || stop}
                    fllave fin


SDecs(out ts0, err0) ::= varconsts illave
                    Decs(out ts1, err1)
                    {ts0 ← ts1, err0 ← err1}
                    fllave


Decs(out ts0, err0) ::= Dec(out id1, type1, const1, value1)
             { tsh2 ← AñadeID(CreaTS(), id1, type1, const1, value1) }
             RDecs(in tsh2, out ts2, err2)
             { ts0 = ts2, err0 = err2 }

RDecs(in tsh0, errh0, out ts0, err0) ::= pyc
       Dec(out id1, type1, const1, value1)
       {tsh2←AñadeID(tsh0, id1, type1, const1, value1 ), errh2 ←ExisteID(ts0, id1)}
       RDecs(int tsh2, errh2, out ts2, err2) { ts0 ← ts2, err0 ← err2 }



RDecs(in tsh, errh, out ts, err) ::= ε
       { ts = tsh, err = errh }

Dec(out id0, type0, const0, value0) := var
             Type(out type1)
             ident
             { const0 ←  false, value0 = ?, type0 = type1, id0 = ident.lex }


Dec(out const0, type0, id0) ::= const
                    Type(out type1)
                    {const0 ← true, type0 ← type1}
                    ident
                    {id0 ← ident.lex}
                    dpigual
                    Lit(out value2)
```

```
                    {value 0 ← value2}


Dec → ε


SIntst(in tsh0, out err0, cod0) ::= instructions illave

                    {tsh1 ← tsh0}
                    Insts(in tsh1, out err1, cod1)
                    {err0 ← err1, cod0 ← cod1}
                    fllave


Insts (in tsh0, out err0, cod0) := {tsh1 ← tsh0}
                           Inst(in tsh1, out tsh1 err1, cod1)
                           {tsh2 ←  tsh1, errh2 ← err1, codh2 ← cod1}
                           RInsts(in tsh2, errh2, codh2)
                           {err0 ← errh2, cod2 ← codh2}


RInsts(in tsh0, errh0, codh0, out cod0, err0) ::= pyc
                    {tsh1 ← tsh0}
                    Inst(in tsh1, out tsh1, err1, cod1)
                    {tsh2 ← tsh1, errh2 ← errh0 v err1, codh2 ← codh0 || cod1 }
                    RInsts(in tsh2, errh2, codh2, out cod2, err2)
                    {cod0 ← cod 2, err0 ← err2}


RInsts(in errh0, codh0, out err0, cod0) ::= ε
                                  {err0 = errh0, cod0 = codh0}


Inst(in tsh0, out err0, cod0) ::= ident asig
       { tsh1 ← tsh0 }
       Expr(in tsh1, out cod1)

       { err0 ← (¬asignacionValida(tsh0[ident.lex].type, type1), ∨ ¬existeID(tsh0,
       ident.lex) ∨ tsh0[ident.lex].const = true), cod0 ← cod1 || desapila-
       dir(tsh0[ident.lex].dir)}



Inst( int tsh0, out err0, cod0 ) ::= in lpar ident

       { err0 ← (¬existeID(tsh0, ident.lex) ∨ tsh0[ident.lex].const = true), cod0 ←
       in(tsh0[ident.lex].type || desapila-dir(tsh0[ident.lex].dir) }
       rpar


Inst(out err0, cod 0) ::= out lpar

              Expr(out err1, type1, cod1)
              {err0 ← type1 = terr, cod0 ← cod1 || out}
```

```
            rpar


Inst(out cod0) ::= swap1 lpar rpar
            { cod0 ← swap1 }


Inst(out cod0) ::= swap2 lpar rpar
            { cod0 ← swap2 }


Type(out type0) ::= boolean
            { type0 ← boolean }


Type(out type0) ::= character
            { type0 ← character }


Type(out type0) ::= integer
            { type0 ← integer }


Type(out type0) ::= natural
            { type0 ← natural }


Type(out type0) ::= float
            { type0 ← float }


Cast(out type0) ::= char
            { type0 = char }


Cast(out type0) ::= int
            { type0 = int }


Cast(out type0) ::= nat
            { type0 = nat }


Cast(out type0) ::= float
            { type0 = float }


Expr(in tsh0, out type0, cod0) ::= { tsh1 ← tsh0 }
                           Term( out type1, cod1 )
                           { typeh2 ← type1, tsh2 ← tsh0 }
```

```
                          FExpr( in typeh2, tsh2, out type2, cod2 )
                          {type 0 ← type2, cod0 ← cod1 || cod2}

FExpr( in tsh0, typeh0,  out type0, cod0 ) ::= Op0(out op1)

                                { tsh2 ← tsh0 }
                                Term(out type2, cod2)
                          { type0 ← tipoFunc(typeh0, type2), cod0 ← cod2 || op1 }


FExpr(in typeh0, out type0, cod0) ::= ε
                          {type0 ← typeh0, cod0 ← ε}


Term(in tsh0, out type0, cod0) ::= {tsh1 ← tsh0}

                          Fact(out tsh1, type1, cod1)
                          {tsh2 ← tsh1, typeh2 ← type1, codh2 ← cod1}
                          RTerm(in tsh2, typeh2, codh2, out type2, cod2)
                          {type0 ← type2, cod0 ← cod2 }


RTerm( int tsh0, typeh0, codh0, out type0, cod0 ) ::= Op1(out op1)
             { tsh2 ← tsh0 }
             Fact( out tsh2, type2, cod2 )
             { tsh3 ← tsh2, typeh3 ← tipoFunc(typeh0, op1, type2), codh3 ← codh0 ||
             cod2 || op1 }
             RTerm( in tsh3, typeh3, codh3, out type3, cod3 )
             {type0 ← type3, cod0 ← cod3}


RTerm(in typeh0, codh0, out type0, cod0) ::= ε
                                {type0 ← typeh0,  cod0 ← codh0 }


Fact(in tsh0, out type0, cod0) ::= { tsh1 ← tsh0 }

                          Shft(in tsh1, out tsh1, typ1, cod1)
                          { tsh2 ← tsh1, typeh2 ← type1, codh2 ← cod1 }
                          RFact(in tsh2, typeh2, codh2, out type2, cod2)
                          { type0 ← type2, cod0 ← cod2 }


RFact(in tsh0, typeh0 codh0, out type0, cod0) ::= Op2(out op1)
      { tsh2 ← tsh0 }
      Shft(in tsh2, out type2, cod2)
      { tsh3 ← tsh2, typeh3 ← tipoFunc(typeh0, op1, type2), codh3 ← codh0 || cod2 ||
op1 }
      RFact(in tsh3, typeh3, codh3, out type3, cod3)
      {type0 ← type3, cod0 ← cod3}


RFact(in typeh0, codh0, out type0, cod0) ::= ε
```

```
                                { type0<-- typeh0, cod0 ← codh0 }

Shft(in tsh0, out type0, cod0) ::= { tsh1 ← tsh0 }

                            Unary(in tsh1, out type1, cod1)
                            { typeh2 ← type1, tsh2 ← tsh0 }
                            FShtf(in typeh2, tsh2, out type2, cod2)
                            { type0 ← type2, cod0 ← cod1 || cod2 }


FShft( in tsh0, typeh0 out type0, cod0 ) ::= Op3(out op1)

                        { tsh1 ← tsh0 }
                        Shft( in tsh2, out type2, cod2 )
                        { type0 ← tipoFunc(typeh0, op1, type2), cod0 ← cod2 || op1 }


FShft( in typeh0, out type0 ) ::= ε

                        { type0 ← typeh0 }

Unary(in tsh0, out type0) ::= Op4(out op1)

                        { tsh2 ← tsh0 }
                        Unary(in tsh2, out type2, cod2)
                        { type0 ← opUnario(op1, type2), cod0 ← cod2 || op1 }


Unary(in tsh0, out type0, cod0) ::= lpar

                               Cast(out type1)
                               rpar
                               { tsh2 ← tsh0 }
                               Paren(in tsh2, out type2, cod2)
                               { type0 ← casting(type1, type2), cod0 ← cod2 || type1}


Unary(in tsh0, out type0, cod0) ::= { tsh1 ← tsh0 }

                               Paren(in tsh1, out type1, cod1)
                               { type0 ← type1, cod0 ← cod1 }


Paren(in tsh0, out type0, cod0) ::= lpar

                               { tsh1 ← tsh0 }
                               Expr(in tsh1, out type1, cod1)
                               { type0 ← type1, cod0 ← cod1 }
                               rpar


Paren(in tsh0, out type0, cod0) ::= { tsh1 ← tsh0 }

                               Lit(in tsh1, out type1, value1)
                               {type0 ← type1, cod0 ← apilar(value1)}
```

```
Paren(in tsh0, out type0, cod0) ::= ident
             { type0 ← tipoDe(ident.lex. tsh0),
       cod0 ← Si tsh0[ident.lex].const = true
                    apila(tsh0[ident.lex].value)
             Si no
                    apila-dir(tsh0 [ident.lex].dir) }

Op0(out op0) ::= igual
             { op0 ← igual }

Op0(out op0) ::= noigual
             { op0 ← noigual }

Op0(out op0) ::= men

             { op0 ← men }

Op0(out op0) ::= may
             { op0 ← may }

Op0(out op0) ::= menoig
             { op0 ← menoig }

Op0(out op0) ::= mayoig
             { op0 ← mayoig }

Op1(out op0) ::= or
             { op0 ← or }

Op1(out op0) ::= menos
             { op0 ← menos }

Op1(out op0) ::= mas
             { op0 ← mas }

Op2(out op0) ::= and
             { op0 ← and }

Op2(out op0) ::= mod
             { op0 ← mod }

Op2(out op0) ::= div
```

```
            { op0 ← div }

Op2(out op0) ::= must
            { op0 ← must }

Op3(out op0) ::= lsh
            { op0 ← lsh }

Op3(out op0) ::= rsh
            { op0 ← rsh }



Op4(out op0) ::= not
            { op0 ← not }

Op4(out op0) ::= menos
            { op0 ← menos }


Lit(out type0, value0) ::= LitBool(out type1, value1)
                  { type0 ← type1, value0 ← value1}


Lit(out type0, value0) ::= LitNum(out type1, value1)
                  { type0 ← type1, value0 ← value1}


Lit(out type0, value0) ::= litchar
                  { type0 ← char, value0 ← stringToChar( litchar.lex ) }

LitBool(out type0, value0) ::= true
                  { type0 ← boolean, value0 ← true }


LitBool(out type0, value0) ::= false
                  { type0 ← boolean, value0 ← false }


LitNum(out type0, value0) ::- litnat
                  { type0 ← natural, value0 ← stringToNat( litnat.lex) }

LitNum(out type0, value0) ::- litfloat
                  { type0 ← float, value0 ← stringToFloat( litfloat.lex) }


LitNum(out type0, value0) ::= menos
                  FLitNum(out type1, value1)
```

```
                    { type0 ← type1, value0 ← value1 }


FLitNum(out type0, value0) ::= litnat
                    {type0 ← integer, value0 ← -stringToFloat(litnat.lex)}


FLitNum(out type0, value0) ::= litfloat
                    {type0 ← float, value0 ← -stringToFloat(litfloat.lex)}
```

Optimización:

```
global ts, cod, err;

Program( ) ::= program ident illave SDecs( ) SInts( ) { emite( stop ) } fllave fin

SDecs( ) ::= varconsts illave Decs( ) fllave

Decs( ) ::= Dec( out id1, type1, const1, value1 )
             { ts ← AñadeID( CreaTS( ), id1, type1, const1, value1 ) }
             RDecs( )

RDecs( ) ::= pyc Dec( out id1, type1, const1, value1 )
       { ts ← AñadeID( ts, id1, type1, const1, value1 ), err ← ExisteID (ts, id1 ) }
       RDecs( )


RDecs() ::= ε

Dec(out id0, type0, const0, value0):= var
                    Type(out type1)
                    ident
                    { const0 ←  false, value0 = ?, type0 = type1, id0 = ident.lex }


Dec(out const0, type0, id0) ::= const
                    Type(out type1)
                    {const0 ← true, type0 ← type1}
                    ident
                    {id0 ← ident.lex}
                    dpigual
                    Lit(out value2)
                    {value 0 ← value2}


Dec → ε
```

```
SIntst() ::= instructions illave

      Insts( )
      fllave


Insts () := Inst() RInsts()


RInsts() ::= pyc Inst() RInsts()


RInsts() ::= ε


Inst() ::= ident asig Expr(out type1, cod1)

      { err ← err ∨ (¬asignacionValida(ts[ident.lex].type, type1), ∨ ¬existeID(ts,
      ident.lex) ∨ ts[ident.lex].const = true), emite(cod1 || desapila-
      dir(ts[ident.lex].dir))}



Inst() ::= in lpar ident

      { err ← err ∨ (¬existeID(ts, ident.lex) ∨ ts[ident.lex].const = true),
      emite(in(ts[ident.lex].type || desapila-dir(ts[ident.lex].dir)) }
      rpar


Inst() ::= out lpar

      Expr( out type1, cod1)
      {err ← err ∨ type1 = terr, emite(cod1 || out)}
      rpar


Inst() ::= swap1 lpar rpar
            { emite(swap1) }


Inst() ::= swap2 lpar rpar
            { emite(swap2) }


Type(out type0) ::= boolean
            { type0 ← boolean }


Type(out type0) ::= character
            { type0 ← character }
```

```
Type(out type0) ::= integer
            { type0 ← integer }


Type(out type0) ::= natural
            { type0 ← natural }


Type(out type0) ::= float
            { type0 ← float }


Cast(out type0) ::= char
            { type0 = char }


Cast(out type0) ::= int
            { type0 = int }


Cast(out type0) ::= nat
            { type0 = nat }


Cast(out type0) ::= float
            { type0 = float }


Expr(out type0, cod0) ::= Term( out type1, cod1 )
                          { typeh2 ← type1}
                          FExpr( in typeh2, out type2, cod2 )
                          {type 0 ← type2, cod0 ← cod1 || cod2}

FExpr( in typeh0,  out type0, cod0 ) ::= Op0(out op1)

                          Term(out type2, cod2)
                          { type0 ← tipoFunc(typeh0, type2), cod0 ← cod2 || op1 }


FExpr(in typeh0, out type0, cod0) ::= ε
                          {type0 ← typeh0, cod0 ← ε}


Term(out type0, cod0) ::= Fact(out type1, cod1)

                          {typeh2 ← type1, codh2 ← cod1}
                          RTerm(in typeh2, codh2, out type2, cod2)
                          {type0 ← type2, cod0 ← cod2 }
```

```
RTerm( int typeh0, codh0, out type0, cod0 ) ::= Op1(out op1)
            Fact( out type2, cod2 )
            { typeh3 ← tipoFunc(typeh0, op1, type2), codh3 ← codh0 || cod2 || op1 }
            RTerm( typeh3, codh3, out type3, cod3 )
            {type0 ← type3, cod0 ← cod3}


RTerm(in typeh0, codh0, out type0, cod0) ::= ε
                                {type0 ← typeh0,  cod0 ← codh0 }


Fact(out type0, cod0) ::= Shft(out type1, cod1)

                          { typeh2 ← type1, codh2 ← cod1 }
                          RFact(in typeh2, codh2, out type2, cod2)
                          { type0 ← type2, cod0 ← cod2 }


RFact(in typeh0 codh0, out type0, cod0) ::= Op2(out op1)
      Shft(out type2, cod2)
      { typeh3 ← tipoFunc(typeh0, op1, type2), codh3 ← codh0 || cod2 || op1 }
      RFact(in typeh3, codh3, out type3, cod3)
      {type0 ← type3, cod0 ← cod3}


RFact(in typeh0, codh0, out type0, cod0) ::= ε
                                { type0 ← typeh0, cod0 ← codh0 }


Shft(out type0, cod0) ::= Unary(out type1, cod1)

                          { typeh2 ← type1 }
                          FShtf(in typeh2, out type2, cod2)
                          { type0 ← type2, cod0 ← cod1 || cod2 }


FShft( in typeh0 out type0, cod0 ) ::= Op3(out op1)

                                Shft( out type2, cod2 )
                                { type0 ← tipoFunc(typeh0, op1, type2), cod0 ← cod2
                          || op1 }


FShft( in typeh0, out type0 ) ::= ε

                    { type0 ← typeh0 }
Unary(out type0) ::= Op4(out op1)

                    Unary(out type2, cod2)
                    { type0 ← opUnario(op1, type2), cod0 ← cod2 || op1 }


Unary(out type0, cod0) ::= lpar

                          Cast(out type1)
```

```
                        rpar
                        Paren(out type2, cod2)
                        { type0 ← casting(type1, type2), cod0 ← cod2 || type1}

Unary(out type0, cod0) ::= Paren(out type1, cod1)

                        { type0 ← type1, cod0 ← cod1 }

Paren(out type0, cod0) ::= lpar

                        Expr(out type1, cod1)
                        { type0 ← type1, cod0 ← cod1 }
                        rpar

Paren(out type0, cod0) ::= Lit(out type1, value1)
 {type0 ← type1, cod0 ← apilar(value1)}


Paren(out type0, cod0) ::= ident
             { type0 ← tipoDe(ident.lex. ts),
             cod0 ← Si Paren.tsh[ident.lex].const = true
                    apila(Paren.tsh[ident.lex].value)
             Si no
                    apila-dir(Paren.tsh[ident.lex].dir)
}


Op0(out op0) ::= igual
             { op0 ← igual }

Op0(out op0) ::= noigual
             { op0 ← noigual }

Op0(out op0) ::= men

             { op0 ← men }

Op0(out op0) ::= may
             { op0 ← may }

Op0(out op0) ::= menoig
             { op0 ← menoig }

Op0(out op0) ::= mayoig
             { op0 ← mayoig }
```

```
Op1(out op0) ::= or
             { op0 ← or }

Op1(out op0) ::= menos
             { op0 ← menos }

Op1(out op0) ::= mas
             { op0 ← mas }

Op2(out op0) ::= and
             { op0 ← and }

Op2(out op0) ::= mod
             { op0 ← mod }

Op2(out op0) ::= div
             { op0 ← div }

Op2(out op0) ::= must
             { op0 ← must }

Op3(out op0) ::= lsh
             { op0 ← lsh }

Op3(out op0) ::= rsh
             { op0 ← rsh }



Op4(out op0) ::= not
             { op0 ← not }

Op4(out op0) ::= menos
             { op0 ← menos }


Lit(out type0, value0) ::= LitBool(out type1, value1)
                   { type0 ← type1, value0 ← value1}


Lit(out type0, value0) ::= LitNum(out type1, value1)
                   { type0 ← type1, value0 ← value1}


Lit(out type0, value0) ::= litchar
```

```
                    { type0 ← char, value0 ← stringToChar( litchar.lex ) }

LitBool(out type0, value0) ::= true
                    { type0 ← boolean, value0 ← true }


LitBool(out type0, value0) ::= false
                    { type0 ← boolean, value0 ← false }


LitNum(out type0, value0) ::- litnat
                    { type0 ← natural, value0 ← stringToNat( litnat.lex) }

LitNum(out type0, value0) ::- litfloat
                    { type0 ← float, value0 ← stringToFloat( litfloat.lex) }


LitNum(out type0, value0) ::= menos
                     FLitNum(out type1, value1)
                     { type0 ← type1, value0 ← value1 }


FLitNum(out type0, value0) ::= litnat
                     {type0 ← integer, value0 ← -stringToFloat(litnat.lex)}


FLitNum(out type0, value0) ::= litfloat
                     {type0 ← float, value0 ← -stringToFloat(litfloat.lex)}
```

10. Formato de representación del código P

El código de la máquina a pila está representado con un formato binario. Cada instrucción consta de un único byte que la define completamente, más un número variable de bytes entre 0 y 4 dependiendo de los operandos que necesite.

Instrucciones

- **Instrucciones aritméticas y de comparación:** Todas estas instrucciones han sido englobadas en lo que hemos llamado *instrucciones con operador*. Su código es 000X XXXX, donde X es el código del operador, que se detalla más adelante. De esta manera estamos definiendo un total teórico de 32 instrucciones, de las cuales en realidad sólo utilizamos 17 de ellas. El resto de códigos (desde 0001 0001 hasta 0001 1111) se leerán como instrucciones con operador, pero con un operador no reconocido, mostrando el correspondiente error.

- **Instrucción de apilamiento de literales:** Su código es 0010 0XXX, donde X es el código del tipo, que se detalla más adelante. Seguido de este byte irá un operando del tipo especificado, de 1, 2 o 4 bytes. Los formatos de los operandos según los tipos se detallan más adelante. Puesto que los tres bits del tipo nos dejan 8 posibilidades pero sólo tenemos 5 tipos, la situación es similar a la del apartado anterior: Leeremos la instrucción pero con tipo no reconocido, lo cual generará un error.

- **Instrucción de entrada:** Su código es 0010 1XXX, donde X es el código del tipo que se desea introducir, que permitirá a la máquina virtual reconocer la entrada correctamente.

- **Instrucción de conversión de tipos:** Su código es 0011 0XXX, donde X es el código del tipo al que se desea convertir.

- **Instrucción de carga de memoria (load):** Su código es 0011 1000, y vendrá seguida de un operando natural de 4 bytes, que indica la dirección de memoria de la que se cargará el valor.

- **Instrucción de almacenamiento en memoria (store):** Su código es 0011 1001, y vendrá seguida de un operando natural de 4 bytes, que indica la dirección de memoria en la que se almacenará el valor.

- **Instrucción de salida:** Su código es 0011 1010 y no necesita ningún operando.

- **Instrucción de parada:** Su código es 0011 1011 y no necesita ningún operando.

- **Instrucciones de intercambio de operadores swap:** Su código es 0011 110X, donde X será el tipo de swap: 0 para swap1, 1 para swap2. No necesita ningún operando.

La siguiente tabla muestra un resumen de todas las instrucciones definidas y no definidas en orden numérico según el byte de código.

Código	Instrucción
0000 0000 - 0001 0000 (00h-10h)	Instrucciones con operador
0001 0001 - 0001 1111 (11h-1Fh)	Instrucciones con operador - Operador no definido
0010 0000 - 0010 0100 (20h-24h)	Instrucciones push
0010 0101 - 0010 0111 (25h-27h)	Instrucciones push - Tipo no definido
0010 1000 - 0010 1100 (28h-2Ch)	Instrucciones casting
0010 1101 - 0010 1111 (2Dh-2Fh)	Instrucciones casting - Tipo no definido
0011 0000 - 0011 0100 (30h-34h)	Instrucciones input
0011 0101 - 0011 0111 (35h-37h)	Instrucciones input - Tipo no definido
0011 1000 (38h)	Instrucción load
0011 1001 (39h)	Instrucción store
0011 1010 (3Ah)	Instrucción output
0011 1011 (3Bh)	Instrucción stop
0011 1100 (3Ch)	Instrucción swap1
0011 1101 (3Dh)	Instrucción swap2
0011 1110 - 1111 1111 (3Eh-FFh)	Instrucción no definida

Operadores

Código	Operador
00000 (00h)	Suma (+)
00001 (01h)	Resta (-)
00010 (02h)	Multiplicación (*)
00011 (03h)	División (/)
00100 (04h)	Módulo (%)
00101 (05h)	Igual (==)

00110 (06h)	Desigual (!=)
00111 (07h)	Menor (<)
01000 (08h)	Menor o igual (<=)
01001 (09h)	Mayor (>)
01010 (0Ah)	Mayor o igual (>=)
01011 (0Bh)	Conjunción lógica (and)
01100 (0Ch)	Disyunción lógica (or)
01101 (0Dh)	Desplazamiento a la izquierda (<<)
01110 (0Eh)	Desplazamiento a la derecha (>>)
01111 (0Fh)	Negación aritmética (- unario)
10000 (10h)	Negación lógica (not)

Tipo

Código	Tipo	Formato de operando
000 (0h)	natural	4 bytes, número entero con signo de 32 bits, representado en complemento a 2, big endian, cuyos valores negativos son inválidos.
001 (1h)	integer	4 bytes, número entero con signo de 32 bits, representado en complemento a 2, big endian.
010 (2h)	float	4 bytes, número decimal en punto flotante de 32 bits, en estándar IEEE 754 (precisión simple).
011 (3h)	boolean	1 byte, 00h para false, 01h para true, aunque cualquier valor distinto de 00h se reconocerá como true.
100 (4h)	character	2 bytes, número entero sin signo de 16 bits que codifica el valor Unicode del carácter, tal y como lo especifica Java.

11. Notas sobre la implementación

11.1. Descripción de paquetes

Todo el código de la práctica incluye JavaDoc, el cual se entregará junto con el código. Cualquier pega con la información detallada a continuación podrá consultarse en dicha documentación.

plg.gr3.code

Contiene todas las clases de lectura y escritura de código. Su base son las clases abstractas CodeReader y CodeWriter, de las que existen implementaciones para leer y cargar de fichero, así como una implementación de CodeWriter que permite la escritura directa en una lista.

plg.gr3.vm.instr

Contiene las definiciones de instrucciones, todas ellas descendientes de una clase abstracta Instruction. Este paquete es el que implementa la ejecución de código, mediante Instruction#execute(VirtualMachine), método abstracto que todas las instrucciones deben implementar.

plg.gr3.data

Contiene todo lo relacionado con la gestión de datos, es decir: los tipos, los valores del lenguaje y los operadores.

La clase Type representa los tipos de nuestro lenguaje. Esta clase es similar a un enumerado, cuyos valores pueden crearse en tiempo de ejecución. Puesto que lo que diferencia a los tipos es su nombre, nunca habrá dos instancias de la clase Type con el mismo nombre. Con esto conseguimos evitar los problemas que conllevaría tener una segunda instancia de un tipo nativo (por ejemplo, integer) creado con el constructor, puesto que no tendría bien definido el código del tipo.
Los operadores se representan mediante las clases BinaryOperator y UnaryOperator, que implementan la interfaz Operator por cuestiones de comodidad.
Los valores de nuestro lenguaje vienen representados usando las subclases de la clase abstracta Value, los cuales envuelven los tipos primitivos de Java, añadiendo la restricción a los naturales de que sólo se pueden usar valores positivos.

plg.gr3.debug

Paquete de depuración que incluye utilidades para escribir por consola errores y mensajes, indicando en ellos línea y columna (para compilación) o número de instrucción (para ejecución).

plg.gr3.errors

Paquete base para la representación de errores. Sólo incluye una clase abstracta Error, superclase de los errores de ejecución y compilación.

plg.gr3.errors.runtime

Errores en tiempo de ejecución, con base en la clase abstracta RuntimeError. Los errores de ejecución se dan en una posición del programa e instrucción concretas, lo cual queda reflejado con los atributos. Las subclases de esta clase abstracta incluidas en este paquete son los tipos de errores que podemos tener en ejecución.

plg.gr3.errors.compile

Errores en tiempo de compilación, con base en la clase abstracta CompileError. Los errores de compilación se dan en una posición del fichero fuente, incluyendo línea y columna, lo cual queda reflejado con los atributos. Las subclases de esta clase abstracta incluidas en este paquete son los tipos de errores que podemos tener en compilación.

plg.gr3.lexer

Contiene el analizador léxico y todas las clases que necesita. Su contenido se limita a la definición del analizador como tal en la clase `Lexer` y la definición de las categorías léxicas y los tokens, usando las clases `TokenType`, `Token` y `LocatedToken`.
La clase Lexer tiene únicamente un método público además del constructor, nextToken(), que devuelve un LocatedToken, es decir, un Token y su posición en el fichero. Este token se lee mediante las clases del paquete java.util.regex Pattern y Matcher, usando las expresiones regulares que definen las instancias del tipo enumerado TokenType.

plg.gr3.parser

Contiene el analizador sintáctico y todas las clase que necesita. Su contenido se limita a la definición del analizador como tal en la clase Parser, la definición de la tabla de símbolos en SymbolTable y la clase Attributes representar los atributos de la gramática.

plg.gr3.vm

Definición de la máquina virtual en la clase VirtualMachine, que mantiene el estado de la máquina virtual y define métodos para que pueda manipularse externamente.

11.2. Notas de la implementación

Lexer vs Autómata

La implementación del Lexer no es un autómata como tal, sino que se apoya en las clases Pattern y Matcher de la librería estándar de Java.
Mediante Pattern definimos expresiones regulares usando la sintaxis típica de Perl, que posteriormente podremos reconocer en un String usando la clase Matcher.
Para solventar los problemas que se nos planteaban en el punto 6 de tener que comprobar que tras un token no podía haber ciertos caracteres, se utilizan lo que llaman *lookahead patterns*, es decir, trozos de la expresión regular que, aunque han de cumplirse y se reconocen, no producen el consumo de caracteres y no forman parte del String reconocido.
Para el resto de casos, las expresiones regulares usadas son fácilmente reconvertibles en AFDs, que al final vienen a reconocer las categorías léxicas de la misma forma que lo haría el AFD del apartado 6.