Práctica de Procesadores de Lenguaje
Primera Parte

Fecha de entrega: **Lunes 20 de enero de 2014**

Número de grupo:
Componentes del Grupo:

- Marcos Lorenzo, Raúl
- Núñez de Arenas Besteiro, Irene
- Rodríguez Rodrigo, Pedro Javier
- Saavedra Palacios,Luis Antonio

Índice de Contenidos

1. Especificación del léxico del lenguaje

Los caracteres especiales *espacio*, *tabulador*, *retorno de carro* y *nueva línea* así como las líneas de comentario no se tendrán en cuenta a la hora de analizar lexicamente los ficheros de entrada.

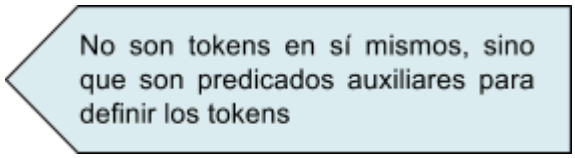
Casos especiales:

WS ≡ [\t\r\n]+ → skip
COMMENT ≡ '@' ~[\r\n]* → skip

Símbolos y operadores:

SEP ≡ ';'
PAR_AP ≡ '('
PAR_CI ≡ ')'

LETRA ≡ 'A'..'Z' | 'a'..'z'
GUION_BAJO ≡ '_'
CERO ≡ '0'
DIGITO ≡ '1'..'9'
PUNTO ≡ '.'



Operadores de Casting:

OP_CAST ≡ PAR_AP TIPO PAR_CI

Operador Unario (Negación Lógica):

OP_LOGNOT ≡ '!'

Operadores Multiplicativos:

OP_MULTI ≡ (OP_MULT | OP_DIV | OP_MOD | OP_LOGAND)
OP_MULT ≡ '*'
OP_DIV ≡ '/'
OP_MOD ≡ '%'
OP_LOGAND ≡ '&&'

Operadores Aditivos y Operador Unario (Negación):

OP_ADD ≡ '+'
OP_SUB ≡ '-'
OP_LOGOR ≡ '||'

Operadores de Comparación:

OP_COMP ≡ (OP_LT | OP_GT | OP_LET | OP_GET | OP_E | OP_NE)
OP_LT ≡ '<'
OP_GT ≡ '>'
OP_LET ≡ '<='
OP_GET ≡ '>='

OP_E ≡ '=='
OP_NE ≡ '!='

Operador de asignación:

OP_ASIG ≡ '='

Operadores de Lectura/Escritura:

OP_IN ≡ 'in'
OP_OUT ≡ 'out'

Definición de Tipos:

TIPO ≡ TIPO_INT | TIPO_REAL
TIPO_INT ≡ ('i'|'I')('n'|'N')('t'|'T')
TIPO_REAL ≡ ('r'|'R')('e'|'E')('a'|'A')('l'|'L')

IDENT ≡ (LETRA | GUION_BAJO) (LETRA | DIGITO | GUION_BAJO)*
NUM ≡ DIGITO (DIGITO | CERO)* (PUNTO (DIGITO | CERO)+)?

2. Especificación de la sintaxis del lenguaje

Definición de los operadores

Operador	Nivel de prioridad	Aridad	Asociatividad
Lectura (in)	0	1	Ninguna
Escritura (out)	0	1	Ninguna
Asignación (=)	1	2	Derecha
Menor (<)	2	2	Ninguna
Mayor (>)	2	2	Ninguna
Menor o Igual (<=)	2	2	Ninguna
Mayor o Igual (>=)	2	2	Ninguna
Igual (==)	2	2	Ninguna
Distinto (!=)	2	2	Ninguna
O lógico (||)	3	2	Izquierda
Suma (+)	3	2	Izquierda
Resta (-)	3	2	Izquierda
Multiplicación (*)	4	2	Izquierda
División (/)	4	2	Izquierda
Módulo (%)	4	2	Izquierda
Y lógico (&&)	4	2	Izquierda
Cambio de signo (-)	5	1	Si
Negación lógica (!)	5	1	Si
Conversión a entero ((int))	6	1	No
Conversión a real ((real))	6	1	No

Formalización de la sintaxis:

tipo ≡ TIPO
id ≡ IDENT
num ≡ NUM

programa → decs accs

decs → decs dec
decs → dec
decs → ʎ

dec → tipo id SEP

accs → accs acc
accs → acc

acc → ioExpr SEP

ioExpr → OP_IN id
ioExpr → OP_OUT id
ioExpr → OP_OUT asigExpr
ioExpr → asigExpr

asigExpr → id OP_ASIG asigExpr
asigExpr → compExpr

compExpr → adiExpr OP_COMP adiExpr
compExpr → adiExpr

adiExpr → adiExpr OP_ADD multExpr
adiExpr → adiExpr OP_SUB multExpr
adiExpr → adiExpr OP_LOGOR multExpr
adiExpr → multExpr

multExpr → multExpr OP_MULTI unaryExpr
multExpr → unaryExpr

unaryExpr → OP_SUB castExpr
unaryExpr → OP_LOGNOT castExpr
unaryExpr → castExpr

castExpr → OP_CAST term
castExpr → term

term → PAR_AP ioExpr PAR_CI
term → id
term → num

3. Estructura y construcción de la tabla de símbolos

3.1. Estructura de la tabla de símbolos

La tabla de símbolos está compuesta por los siguientes campos:

- **id:** Identificador o nombre de la variable declarada.
- **type:** Tipo asignado a esta variable.
- **mem_addr:** Dirección de memoria asignada a esta variable.
- **value:** Valor que, en un momento dado de la ejecución, toma la variable.

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, mem_addr: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 ese identificador existe en la tabla de símbolos, es decir, ha sido previamente declarado.

tipoDe(ts:TS, id:String) : Tipo

Dada una tabla de símbolos y el campo id de un identificador, devuelve el tipo del identificador. Si no existe, devuelve terr (tipo error).

direDe(ts:TS, id:String) : MemAddr

Dada una tabla de símbolos y el campo id de un identificador, devuelve la dirección de memoria del identificador.

3.2.2 Atributos semánticos

- **ts:** Tabla de símbolos sintetizada.
- **tsh:** Tabla de símbolos heredada.
- **id:** Nombre del identificador.
- **type:** Tipo de la declaración.
- **addr:** Dirección de memoria.
- **value:** Valor de la variable.

3.2.3 Gramática de atributos

A continuación se detalla la construcción de los atributos relevantes para la construcción de

la tabla de símbolos. Otros atributos como la tabla de símbolos heredada o los valores que las variables irán tomando a lo largo del programa se detallarán más adelante en sus correspondientes secciones.

programa → decs accs
accs.tsh = decs.ts

decs → decs dec
decs0.addr = desc1.addr + 1
desc0.ts = añadeID(desc1.ts, dec.id, dec.type, decs0.addr, dec.value)

decs → dec
decs.addr = 0
desc.ts = añadeID(creaTS(), dec.id, dec.type, 0, dec.value)

decs → ʎ
desc.ts = creaTS()
desc.addr = 0

dec → tipo id SEP
dec.id = id.lex
dec.type = tipo.type
dec.value = ?

4. Especificación de las restricciones contextuales

4.1. Funciones semánticas

unario(op:Oper, tipo:Tipo) : Tipo

Dado un operador unario y el tipo al que es aplicado comprobamos si se puede aplicar. Por ejemplo, no se puede aplicar el operador de negación lógica a un real, en este caso devolvemos terr.

Operador	Tipo	Tipo devuelto
Lectura (in)	int	int
Lectura (in)	real	real
Escritura (out)	int	int
Escritura (out)	real	real
Negación Lógica (!)	int	int
Negación Lógica (!)	real	terr
Negación (-)	int	int
Negación (-)	real	real
Casting Entero ((int))	int	int
Casting Entero ((int))	real	int
Casting Real ((real))	int	real
Casting Real ((real))	real	real

compruebaTipo(tipo1:Tipo, op:Oper, tipo2:Tipo) : Tipo

Dados dos tipos diferentes y un operador comprobamos que los tipos puedan aplicar el operador. Devolvemos el tipo correspondiente al aplicar el operador o terr en caso de que no se pueda aplicar.

Tipo 1	Operador	Tipo 2	Tipo devuelto
int	Aritmético (+, -, *, /)	int	int
int	Aritmético (+, -, *, /)	real	real
real	Aritmético (+, -, *, /)	int	real
real	Aritmético (+, -, *, /)	real	real
int	Comparación (<, >, <=, >=, ==, !=)	int	int

int	Comparación (<, >, <=, >=, ==, !=)	real	int
Tipo 1	**Operador**	**Tipo 2**	**Tipo devuelto**
real	Comparación (<, >, <=, >=, ==, !=)	int	intl
real	Comparación (<, >, <=, >=, ==, !=)	real	int
int	Lógico (||, &&)	int	int
int	Lógico (||, &&)	real	terr
real	Lógico (||, &&)	int	terr
real	Lógico (||, &&)	real	terr
int	Módulo (%)	int	int
int	Módulo (%)	real	terr
real	Módulo (%)	int	terr
real	Módulo (%)	real	terr

asignacionValida(tipoVar:Tipo, tipoExp:Tipo) : Boolean

Dados el tipo de una variable y el tipo de una expresión, comprueba si a la variable se le asigna un tipo permitido. Por ejemplo, a una variable declarada como entero no se le puede asignar un valor real.

Tipo Var	Tipo Exp	Tipo devuelto
int	int	true
int	real	false
real	int	true
real	real	true

4.2. Atributos semánticos

- **ts:** Tabla de símbolos sintetizada. Se crea en la parte de declaraciones.
- **tsh:** Tabla de símbolos heredada. Se hereda en la parte de instrucciones.
- **type:** Atributo que indica el tipo de la var, exp, etc. Tomar los valores int, real y terr.
- **err:** Atributo que indica si se ha detectado algún error. Es un atributo de tipo booleano.

4.3. Gramática de atributos

A continuación se detalla la construcción de los atributos relevantes para las restricciones contextuales, como son el tipo de las expresiones o los errores de contexto.

programa → decs accs
accs.tsh = decs.ts
programa.err = decs.err v accs.err

decs → decs dec
decs0.addr = desc1.addr + 1
desc0.ts = añadeID(desc1.ts, dec.id, dec.type, decs0.addr, dec.value)
decs0.err = desc1.err v exiteID(desc1.ts, dec.id)

decs → dec
decs.addr = 0
desc.ts = añadeID(creaTS(), dec.id, dec.type, 0, dec.value)
desc.err = false

decs → ʎ
desc.ts = creaTS()
desc.addr = 0
desc.err = false

dec → tipo id SEP

accs → accs acc
accs1.tsh = accs0.tsh
acc.tsh = accs0.tsh
accs0.err = accs1.err v acc.err

accs → acc
acc.tsh = accs.tsh
accs.err = acc.err

acc → ioExpr SEP
ioExpr.tsh = acc.tsh
acc.err = ioExpr.err

ioExpr → OP_IN id
ioExpr.type = tipoDe(ioExpr.tsh, id.lex)
ioExpr.err = ¬existeID(ioExpr.tsh, id.lex)

ioExpr → OP_OUT id
ioExpr.type = tipoDe(ioExpr.tsh, id.lex)
ioExpr.err = ¬existeID(ioExpr.tsh, id.lex)

ioExpr → OP_OUT asigExpr

asigExpr.tsh = ioExpr.tsh
ioExpr.type = asigExpr.type
ioExpr.err = asigExpr.err

ioExpr → asigExpr

asigExpr.tsh = ioExpr.tsh
ioExpr.type = asigExpr.type
ioExpr.err = asigExpr.err

asigExpr → id OP_ASIG asigExpr

asigExpr1.tsh = ioExpr0.tsh
asigExpr0.type = tipoDe(asigExpr0.ts, id.lex)
asigExpr0.err = asigExpr1.err v (¬existeID(asigExpr0.ts, id.lex)) v
(¬asignacionValida(tipoDe(asigExpr0.ts, id), asigExp.type))

asigExpr → compExpr

compExpr.tsh = asigExpr.tsh
asigExpr.type = compExpr.type
asigExpr.err = compExpr.err

compExpr → adiExpr OP_COMP adiExpr

adiExpr0.tsh = compExpr.tsh
adiExpr1.tsh = compExpr.tsh
compExpr.type = compruebaTipo(adiExpr0.type, OP_COMP, adiExpr1.type)
compExpr.err = adiExpr0.err v adiExpr1.err v (compExpr.type == terr)

compExpr → adiExpr

adiExpr.tsh = compExpr.tsh
compExpr.type = adiExpr.type
compExpr.err = adiExpr.err

adiExpr → adiExpr OP_ADD multExpr

adiExpr1.tsh = adiExpr0.tsh
multExpr.tsh = adiExpr0.tsh
adiExpr0.type = compruebaTipo(adiExpr1.type, OP_ADD, multExpr.type)
adiExpr0.err = adiExpr1.err v multExpr.err v (adiExpr0.type == terr)

adiExpr → adiExpr OP_SUB multExpr

adiExpr1.tsh = adiExpr0.tsh
multExpr.tsh = adiExpr0.tsh
adiExpr0.type = compruebaTipo(adiExpr1.type, OP_SUB, multExpr.type)
adiExpr0.err = adiExpr1.err v multExpr.err v (adiExpr0.type == terr)

adiExpr → adiExpr OP_LOGOR multExpr

adiExpr1.tsh = adiExpr0.tsh
multExpr.tsh = adiExpr0.tsh
adiExpr0.type = compruebaTipo(adiExpr1.type, OP_LOGOR, multExpr.type)
adiExpr0.err = adiExpr1.err v multExpr.err v (adiExpr0.type == terr)

adiExpr → multExpr

multExpr.tsh = adiExpr.tsh
adiExpr.type = multExpr.type
adiExpr.err = multExpr.err

multExpr → multExpr OP_MULTI unaryExpr

multExpr1.tsh = multExpr0.tsh
unaryExpr.tsh = multExpr0.tsh
multExpr0.type = compruebaTipo(multExpr1.type, OP_MULTI, unaryExpr.type)
multExpr0.err = multExpr1.err v unaryExpr.err v (multExpr0.type == terr)

multExpr → unaryExpr

unaryExpr.tsh = multExpr.tsh
multExpr.type = unaryExpr.type
multExpr.err = unaryExpr.err

unaryExpr → OP_SUB castExpr

castExpr.tsh = unaryExpr.tsh
unaryExpr.type = unario(OP_SUB, castExpr.type)
unaryExpr.err = castExpr.err v unaryExpr.type == terr)

unaryExpr → OP_LOGNOT castExpr

castExpr.tsh = unaryExpr.tsh
unaryExpr.type = unario(OP_LOGNOT, castExpr.type)
unaryExpr.err = castExpr.err v (unaryExpr.type == terr)

unaryExpr → castExpr

castExpr.tsh = unaryExpr.tsh
unaryExpr.type = castExpr.type
unaryExpr.err = castExpr.err

castExpr → OP_CAST term

term.tsh = castExpr.tsh
castExpr.type = unario(OP_CAST, term.type)
castExpr.err = term.err v (unaryExpr.type == terr)

castExpr → term

term.tsh = castExpr.tsh
castExpr.type = term.type
castExpr.err = term.err

term → PAR_AP ioExpr PAR_CI

ioExp.tsh = term.tsh
term.type = ioExpr.type
term.err = ioExpr.err

term → id

term.type = tipoDe(term.tsh, id.lex)
term.err = false

term → num

term.type = num.type
term.err = false;

5. Especificación de la traducción

5.1. Lenguaje objeto

5.1.1. Arquitectura de la máquina P

La arquitectura de la máquina P que se va a emplear consta de los siguientes componentes:

- **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.

5.1.2. Instrucciones en el lenguaje objeto

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

desapila-dir(dirección)

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

Operaciones aritméticas

más

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

menos (binario)

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

mul

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

div

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

modulo

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

menos (unario)

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

Operaciones lógicas

or

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

and

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

not

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

Operaciones de comparación

mayor

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

menor

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

mayorig

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

menorig

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

igual

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

distinto

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

Operaciones de conversión

castInt

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

castReal

Pila[CPila] ← (real) Pila[CPila]
CProg ← CProg + 1

Operaciones de Entrada-Salida

in

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

Otras operaciones

stop

P ← 1

5.2. Funciones semánticas

No vamos a usar ninguna función semántica adicional.

5.3. Atributos semánticos

- **cod:** Atributo sintetizado de generación de código.
- **op:** Enumerado que nos dice cuál es el operador utilizado, en caso de que haya varias posibilidades.

5.4. Gramática de atributos

A continuación se detalla la construcción del atributo relevante para la construcción del código pila que formaliza la traducción.

programa → decs accs

accs.tsh = decs.ts

programa.cod = accs.cod || stop

decs → decs dec

decs0.addr = desc1.addr + 1

desc0.ts = añadeID(desc1.ts, dec.id, dec.type, decs0.addr, dec.value)

decs → dec

decs.addr = 0

desc.ts = añadeID(creaTS(), dec.id, dec.type, 0, dec.value)

decs → ʎ

desc.ts = creaTS()

desc.addr = 0

dec → tipo id SEP

accs → accs acc

accs1.tsh = accs0.tsh

acc.tsh = accs0.tsh

accs0.cod = accs1.cod || acc.cod

accs → acc

acc.tsh = accs.tsh
accs.cod = acc.cod

acc → ioExpr SEP

ioExpr.tsh = acc.tsh
acc.cod = ioExpr.cod

ioExpr → OP_IN id

ioExpr.cod = in || desapila-dir(direDe(ioExpr.tsh, id.lex))

ioExpr → OP_OUT id

ioExpr.cod = apila-dir(direDe(ioExpr.tsh, id.lex)) || out

ioExpr → OP_OUT asigExpr

asigExpr.tsh = ioExpr.tsh
ioExpr.cod = asigExpr.cod || out

ioExpr → asigExpr

asigExpr.tsh = ioExpr.tsh
ioExpr.cod = asigExpr.cod

asigExpr → id OP_ASIG asigExpr

asigExpr1.tsh = ioExpr0.tsh
asigExpr0.cod = asigExpr1.cod || desapila-dir(direDe(asigExpr.tsh, id.lex))

asigExpr → compExpr

compExpr.tsh = asigExpr.tsh
asigExpr.cod = compExpr.cod

compExpr → adiExpr OP_COMP adiExpr

adiExpr0.tsh = compExpr.tsh
adiExpr1.tsh = compExpr.tsh
compExpr.cod = adiExpr0.cod || adiExpr1.cod || OP_COMP.op

compExpr → adiExpr

adiExpr.tsh = compExpr.tsh
compExpr.cod = adiExpr.cod

adiExpr → adiExpr OP_ADD multExpr

adiExpr1.tsh = adiExpr0.tsh
multExpr.tsh = adiExpr0.tsh
adiExpr0.cod = adiExpr1.cod || multExpr.cod || mas

adiExpr → adiExpr OP_SUB multExpr

adiExpr1.tsh = adiExpr0.tsh
multExpr.tsh = adiExpr0.tsh
adiExpr0.cod = adiExpr1.cod || multExpr.cod || menos

adiExpr → adiExpr OP_LOGOR multExpr

adiExpr1.tsh = adiExpr0.tsh
multExpr.tsh = adiExpr0.tsh
adiExpr0.cod = adiExpr1.cod || multExpr.cod || or

adiExpr → multExpr

multExpr.tsh = adiExpr.tsh
adiExpr.cod = multExpr.cod

multExpr → multExpr OP_MULTI unaryExpr

multExpr1.tsh = multExpr0.tsh
unaryExpr.tsh = multExpr0.tsh
multExpr0.cod = multExpr1.cod || unaryExpr.cod || OP_MULTI.op

multExpr → unaryExpr

unaryExpr.tsh = multExpr.tsh
multExpr.cod = unaryExpr.cod

unaryExpr → OP_SUB castExpr

castExpr.tsh = unaryExpr.tsh
unaryExpr.cod = castExpr.cod || menos(unario)

unaryExpr → OP_LOGNOT castExpr

castExpr.tsh = unaryExpr.tsh
unaryExpr.cod = castExpr.cod || not

unaryExpr → castExpr

castExpr.tsh = unaryExpr.tsh
unaryExpr.cod = castExpr.cod

castExpr → OP_CAST term

term.tsh = castExpr.tsh
castExpr.cod = term.cod || OP_CAST.op

castExpr → term

term.tsh = castExpr.tsh
castExpr.cod = term.cod

term → PAR_AP ioExpr PAR_CI

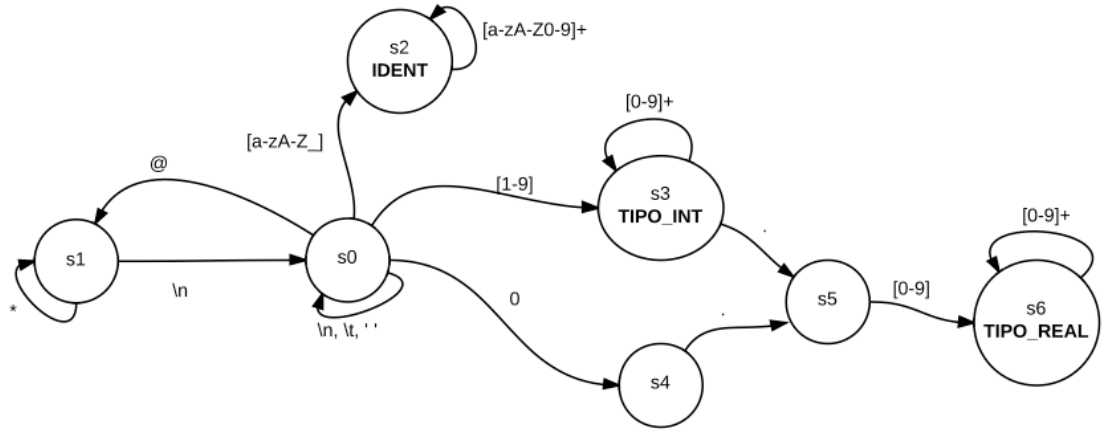
ioExp.tsh = term.tsh
term.cod = ioExpr.cod

term → id
term.cod = apila-dir(direDe(term.tsh, id.lex))

term → num
term.cod = apila(num.value)

6. Diseño del Analizador Léxico

El siguiente gráfico representa parte de la máquina de estados que forma el analizador léxico de nuestro programa:



Los estados s2, s3 y s6 son estados finales en los que el autómata reconoce la aparición de un identificador, un número entero y un número real respectivamente.
El estado s1 es utilizado por el autómata para permitir la inclusión de comentarios de línea en el código. Estas líneas de texto ni siquiera llegan al parser, pues el lexer se encarga de eliminarlas, sirviendo como una pequeña optimización del proceso.

El reconocimiento de las palabras reservadas del lenguaje se realiza mediante autómatas mucho más simples.
El autómata completo que reconoce el lenguaje es demasiado complejo para representarlo entero y la parte representada es, a nuestro parecer, la más importante.

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 símbolos

Nuestra gramática no ha necesitado de ningún acondicionamiento para la construcción de la

tabla de símbolos.

7.2. Acondicionamiento de la Gramática para la Comprobación de las Restricciones Contextuales

```
num returns [String basic_type]
        : NUM_ENTERO { $basic_type = "int"; }
        | NUM_REAL { $basic_type = "real"; }
        ;
```

El atributo 'num' ha sido modificado para que devuelva un valor distinto en función de si es un número entero o real.
Todas las producciones disponen de la variable local 'basic_type' que indica si la operación devuelve un tipo 'int' o 'real'. En el caso de producciones con un único hijo, el valor de esta variable es el mismo que el de su hijo. En el caso de producciones con más de un nodo hijo, el valor de esta variable se calcula en función de varios parámetros (nodos hijos, operador, etc.)

Este atributo 'basic_type' es utilizado por el compilador para comprobar las restricciones de tipo mientras se va generando el código P.

7.3. Acondicionamiento de la Gramática para la Traducción

Gracias a la utilización de ANTLR en la creación del lexer y el parser utilizados para analizar el código fuente original, no ha sido necesario el acondicionamiento de la gramática para poder realizar la traducción.

8. Esquema de traducción orientado a las gramáticas de atributos

programa → decs
{
accs.tsh = decs.ts

}
accs
{ programa.err = decs.err v accs.err
programa.cod = decs.cod || accs.cod }

decs → decs
{ decs0.addr = desc1.addr + 1
desc0.ts = añadeID(desc1.ts, dec.id, dec.type, decs0.addr, dec.value) }
dec

{ decs0.err = desc1.err v exiteID(desc1.ts, dec.id)
decs0.cod = decs1.cod || dec.cod }

decs → dec

{ decs.addr = 0
desc.ts = añadeID(creaTS(), dec.id, dec.type, 0, dec.value)
desc.err = false
decs.cod=dec.cod }

decs → ʌ

{ desc.ts = creaTS()
desc.addr = 0
desc.err = false }

dec → tipo id SEP

{ dec.err = tipo.err v ¬existeID(dec.ts, id.lex)
dec.cod = tipo.cod || id.cod }

accs → accs

{ accs1.tsh = accs0.tsh
acc.tsh = accs0.tsh
accs0.err = accs1.err v acc.err }
acc
{ accs0.cod = accs.cod || acc.cod }

accs → acc

{ acc.tsh = accs.tsh
accs.err = acc.err
accs.cod = acc.cod }

acc → ioExpr

{ ioExpr.tsh = acc.tsh
acc.err = ioExpr.err
acc.cod=ioExpr.cod || SEP.cod }
SEP

ioExpr → OP_IN id

{ ioExpr.type = tipoDe(ioExpr.tsh, id.lex)
ioExpr.err = ¬existeID(ioExpr.tsh, id.lex)
ioExpr.cod = OP_IN.cod || id.cod }

ioExpr → OP_OUT id

{ ioExpr.type = tipoDe(ioExpr.tsh, id.lex)
ioExpr.err = ¬existeID(ioExpr.tsh, id.lex)
ioExpr.cod = OP_OUT.cod || id.cod }

ioExpr → OP_OUT asigExpr

{ asigExpr.tsh = ioExpr.tsh

ioExpr.type = asigExpr.type
ioExpr.err = asigExpr.err
ioExpr.cod = OP_OUT.cod || asigExpr.cod }

ioExpr → asigExpr
{ asigExpr.tsh = ioExpr.tsh
ioExpr.type = asigExpr.type
ioExpr.err = asigExpr.err
ioExpr.cod = asigExpr.cod }

asigExpr → id OP_ASIG asigExpr
{ asigExpr1.tsh = ioExpr0.tsh
asigExpr0.type = tipoDe(asigExpr0.ts, id.lex)
asigExpr0.err = asigExpr1.err v (¬existeID(asigExpr0.ts, id.lex)) v
(¬asignacionValida(tipoDe(asigExpr0.ts, id), asigExp.type))
asigExpr0.cod= id.cod|| OP_ASIG.cod asigExpr1.cod }

asigExpr → compExpr
{ compExpr.tsh = asigExpr.tsh
asigExpr.type = compExpr.type
asigExpr.err = compExpr.err
asigExpr.cod = compExpr.cod }

compExpr → adiExpr
{ adiExpr0.tsh = compExpr.tsh
adiExpr1.tsh = compExpr.tsh }
OP_COMP adiExpr
{ compExpr.type = compruebaTipo(adiExpr0.type, OP_COMP, adiExpr1.type)
compExpr.err = adiExpr0.err v adiExpr1.err v (compExpr.type == terr)
compExpr.cod= adiExpr.cod || OP_COMP.cod || adiExpr.cod }

compExpr → adiExpr
{ adiExpr.tsh = compExpr.tsh
compExpr.type = adiExpr.type
compExpr.err = adiExpr.err
compExpr.cod= adiExpr.cod }

adiExpr → adiExpr
{ adiExpr1.tsh = adiExpr0.tsh
multExpr.tsh = adiExpr0.tsh }
OP_ADD multExpr
{ adiExpr0.type = compruebaTipo(adiExpr1.type, OP_ADD, multExpr.type)
adiExpr0.err = adiExpr1.err v multExpr.err v (adiExpr0.type == terr)
adiexpr.cod=adiExpr.cod|| OP_ADD.cod ||multExpr.cod }

adiExpr → adiExpr

{ adiExpr1.tsh = adiExpr0.tsh
multExpr.tsh = adiExpr0.tsh }
OP_SUB multExpr
{ adiExpr0.type = compruebaTipo(adiExpr1.type, OP_SUB, multExpr.type)
adiExpr0.err = adiExpr1.err v multExpr.err v (adiExpr0.type == terr)
adiExpr0.cod=adiExpr1.cod || OP_SUB.cod||multExpr.cod }

adiExpr → adiExpr
{adiExpr1.tsh = adiExpr0.tsh
multExpr.tsh = adiExpr0.tsh }
OP_LOGOR multExpr
{ adiExpr0.type = compruebaTipo(adiExpr1.type, OP_LOGOR, multExpr.type)
adiExpr0.err = adiExpr1.err v multExpr.err v (adiExpr0.type == terr)
adiExpr0.cod=adiExpr1.cod||OP_LOGOR.cod || multExpr.cod }

adiExpr → multExpr
{ multExpr.tsh = adiExpr.tsh
adiExpr.type = multExpr.type
adiExpr.err = multExpr.err
adiExpr.cod=multExpr.cod}

multExpr → multExpr
{ multExpr1.tsh = multExpr0.tsh
unaryExpr.tsh = multExpr0.tsh }
OP_MULTI unaryExpr
{ multExpr0.type = compruebaTipo(multExpr1.type, OP_MULTI, unaryExpr.type)
multExpr0.err = multExpr1.err v unaryExpr.err v (multExpr0.type == terr)
multExpr0.cod=multExpr1.cod || OP_MULT.cod||unaryExpr.cod }

multExpr → unaryExpr
{ unaryExpr.tsh = multExpr.tsh
multExpr.type = unaryExpr.type
multExpr.err = unaryExpr.err
multExpr.cod=unaryExpr.cod }

unaryExpr → OP_SUB castExpr
{ castExpr.tsh = unaryExpr.tsh
unaryExpr.type = unario(OP_SUB, castExpr.type)
unaryExpr.err = castExpr.err v unaryExpr.type == terr)
unaryExpr.cod=OP_SUB.cod || castExpr.cod }

unaryExpr → OP_LOGNOT castExpr
{ castExpr.tsh = unaryExpr.tsh
unaryExpr.type = unario(OP_LOGNOT, castExpr.type)
unaryExpr.err = castExpr.err v (unaryExpr.type == terr)
unaryExpr.cod = OP_LOGNOT.cod || castExpr.cod }

unaryExpr → castExpr

{ castExpr.tsh = unaryExpr.tsh
unaryExpr.type = castExpr.type
unaryExpr.err = castExpr.err
unaryExpr.cod=castExpr.cod }

castExpr → OP_CAST term

{ term.tsh = castExpr.tsh
castExpr.type = unario(OP_CAST, term.type)
castExpr.err = term.err v (unaryExpr.type == terr)
castExpr.cod= OP_CAST.cod || term.cod }

castExpr → term

{ term.tsh = castExpr.tsh
castExpr.type = term.type
castExpr.err = term.err
castExpr.cod=term.cod }

term → PAR_AP ioExpr

{ ioExp.tsh = term.tsh
term.type = ioExpr.type
term.err = ioExpr.err
term.cod=PAR_AP.cod||ioExpr.cod||PARCI.cod }
PAR_CI

term → id

{ term.type = tipoDe(term.tsh, id.lex)
term.err = false
term.cod=id.cod }

term → num

{ term.type = num.type
term.err = false;
term.cod=num.cod }

9. Esquema de traducción orientado al traductor predictivo – recursivo

9.1. Variables globales

En nuestra implementación no disponemos de variables globales. La tabla de símbolos y las variables que almacenan el código y los errores se gestionan de la siguiente manera:

La clase 'Compiler' es la encargada de realizar la mayor parte de la conversión del código (utilizando las clases auxiliares generadas por ANTLR) y de comprobar que se cumplen las restricciones.
A esta clase pertenecen las 2 variables siguientes:

- **TS:** la tabla de símbolos utilizada para llevar a cabo la traducción del programa.
- **code:** es la variable en la que el compilador va almacenando sucesivamente las instrucciones generadas que darán lugar al código P final.

Para gestionar los mensajes de error generados tanto por el parser como por el compilador disponemos de una clase Logs, que se encarga de almacenar ordenadamente los mensajes de error recibidos para posteriormente mostrarlos en la consola de errores del interfaz.

9.2. Nuevas operaciones y transformación de ecuaciones semánticas

Compiler.class

- **addCode(String ins):** instrucción privada que concatena la nueva instrucción generada con el resto del código P generado anteriormente.
- **getCode():** esta instrucción permite a cualquier otra clase obtener el código generado por el compilador. La interfaz gráfica la utiliza para mostrar el código una vez finalizado el proceso de traducción.

Logs.class

- **addError(String error):** esta instrucción permite a cualquier otra clase añadir un error a los logs de la aplicación. Tanto el parse como el compilador la utilizan durante el proceso de traducción para almacenar de manera ordenada los mensajes de error.
- **getErrorsLog():** instrucción utilizada por la interfaz gráfica para obtener la lista completa de mensajes de error y mostrarlos en la consola de errores.

9.3. Esquema de traducción

programa(out ts0, err0, cod0) ::= decs(out ts1, err1, cod1)
{ ts0 ← ts1 }

accs(out err2, cod2)
{ err0 ← err1 v err2, cod0 ← cod1, cod2 }

decs(out ts0, addr0, err0, cod0) ::= decs(out ts1, addr1, err1, cod1)
{ addr0 ← addr1 + 1, ts0 ← añadeID(ts1, dec.id, dec.type, addr1, dec.value) }
dec(out cod2)
{ err0 ← err1 v existeID(ts1, dec.id), cod0 ← cod1 || cod2 }

decs(out ts0, err0, cod0) ::= dec(out cod1)
{ ts0 ← añadeID(creaTS, dec.id, dec.type, 0, dec.value), cod0 ← cod1)

decs(out ts0, err0) ::= ʌ
{ ts0 ← creaTS(), err0 ← false }

dec(in tsh0, out err0, cod0) ::= tipo id
{ tsh1 ← tsh0 }
SEP(in tsh1, out err1, cod1)
{ err0 ← tipo.err v ¬existeID(err1, id.lex)
cod0 ← tipo.cod || cod1 }

accs(in tsh0, out err0, cod0) ::= accs(in tsh1, out err1, cod1)
{ tsh1 ← tsh0
err0 ← err1 v err2 }
acc(out err2, cod2)
{ cod0 ← cod1 || cod2 }

accs(out err0, cod0) ::= acc(out err1, cod1)

acc(out err0, cod0) ::= ioExpr(out err1, cod1)
{ err0 ← err1, cod0 ← cod1 }
SEP

ioExpr(in tsh0, out type0, err0, cod0) ::= OP_IN id
{ type0 ← tipoDe(tsh0, id.lex)
err0 ← ¬existeID(tsh0, id.lex)
cod0 ← OP_OUT.cod || id.cod }

ioExpr(in tsh0, out type0, err0, cod0) ::= OP_OUT id
{ type0 ← tipoDe(tsh0, id.lex
err0 ← ¬existeID(tsh0, id.lex)
cod0 ← OP_OUT.cod || id.cod }

ioExpr(out type0, err0, cod0) ::= OP_OUT asigExpr(out type1, err1, cod1)

ioExpr(out type0, err0, cod0) ::= asigExpr(out type1, err1, cod1)

asigExpr(in tsh0, out ts0, type0, err0, cod0) ::= id OP_ASIG
{ tsh1 ← tsh0}
asigExpr(in tsh1, out type1, err1, cod1)
{ type0 ← type1,
err0 ← err1 v (¬existeID(ts0, id.lex)) v (¬asignacionValida(tipoDe(ts0, id), type0))
cod0 ← id.cod || OP_ASIG.cod || cod1 }

asigExpr(out type0, err0, cod0) ::= compExpr(out type1, err1, cod1)

compExpr(in tsh0, out type0, err0, cod0) ::= adiExpr(in tsh1, out type1, err1, cod1)
{ ts1 ← ts0 }
OP_COMP adiExpr(out type2, err2, cod2)
{ type0 ← compruebaTipo(type1, OP_COMP, type2 }
err0 ← err1 v err2 v (type1 == terr)
cod0 ← cod1 || OP_COMP.cod1 || cod2 }

compExpr(out type0, err0, cod0) ::= adiExpr(out type1, err1, cod1)

adiExpr(in tsh0, out type0, err0, cod0) ::= adiExpr(in tsh1, out type1, err1, cod1)
{ ts1 ← ts0 }
OP_ADD multExpr(out type2, err2, cod2)
{ type0 ← compruebaTipo(type1, OP_ADD, type2 }
err0 ← err1 v err2 v (type1 == terr)
cod0 ← cod1 || OP_ADD.cod1 || cod2 }

adiExpr(in tsh0, out type0, err0, cod0) ::= adiExpr(in tsh1, out type1, err1, cod1)
{ ts1 ← ts0 }
OP_SUB multExpr(out type2, err2, cod2)
{ type0 ← compruebaTipo(type1, OP_SUB, type2 }
err0 ← err1 v err2 v (type1 == terr)
cod0 ← cod1 || OP_SUB.cod1 || cod2 }

adiExpr(in tsh0, out type0, err0, cod0) ::= adiExpr(in tsh1, out type1, err1, cod1)
{ ts1 ← ts0 }
OP_LOGOR multExpr(out type2, err2, cod2)
{ type0 ← compruebaTipo(type1, OP_LOGOR, type2 }
err0 ← err1 v err2 v (type1 == terr)
cod0 ← cod1 || OP_LOGOR.cod1 || cod2 }

adiExpr(out type0, err0, cod0) ::= multExpr(out type1, err1, cod1)

multExpr(in tsh0, out err0, cod0) ::= multExpr(in tsh1, out err1, cod1)
{ ts1 ← ts0 }
OP_MULTI unaryExpr(out type2, err2, cod2)
{ type0 ← compruebaTipo(type1, OP_MULTI, type2)
err0 ← err1 || err2 v (type1 == terr)

cod0 ← cod1 || OP_MULT.cod || cod2 }

multExpr(out type0) ::= unaryExpr(out type1)

unaryExpr(in tsh0, out type0, err0, cod0) ::= OP_SUB
{ tsh1 ← tsh0 }
castExpr(in tsh1, out type1, err1, cod1)
{ type0 ← type1, err0 ← err1, cod0 ← cod1 }

unaryExpr(in tsh0, out type0, err0, cod0) ::= OP_LOGNOT
{ tsh1 ← tsh0 }
castExpr(in tsh1, out type1, err1, cod1)
{ type0 ← type1, err0 ← err1, cod0 ← cod1 }

unaryExpr(out type0, code0) ::= castExpr(out type1, code1)
{ type0 ← type1, code0 ← code1 }

castExpr(in tsh0, out type0, cod0) ::= OP_CAST
{ts1 ← ts0}
term(in tsh1, out type1, cod1)
{ type0 ← type1, cod0 ← cod1 }

castExpr(out type0) ::= term(out type1)
{ type0 ← type1 }

term(in tsh0, out type0, cod0) ::= PAR_AP
{ tsh1 ← tsh0 }
ioExpr(in tsh1, out type1, cod1)
{ type0 ← type1, cod0 ← cod1 }
PAR_CI

term(out type0, err0) ::= id
{ type0 ← tipoDe(tsh0, id.lex), err0 = false }

term(out type0, err0) ::= num(out type1)
{ type0 ← type1, err0 ← false}

10. Formato de representación del código P

Los archivos que almacenan un programa en código P están formados por una sucesión de instrucciones separadas por saltos de línea. Por el momento no se utiliza ningún método para comprimir los programas una vez traducidos, y estos pueden ser modificados directamente si se dispone de conocimiento del lenguaje aceptado por el intérprete de la máquina P.

El intérprete de la máquina P lee el archivo de código fuente que se le haya proporcionado almacenando las instrucciones en su memoria, para después ejecutar las mismas.

11 Notas sobre la Implementación

11.1. Descripción de archivos

El código fuente de la práctica está estructurado en los siguientes paquetes y clases:

- **cli**: archivos relativos a la ejecución. Archivos:
  - **CmdLineInterface.java.** Se encarga de iniciar todo lo necesario: el lexer, el parser y la gramática, además de los posibles errores.
    Se puede utilizar como una aplicación sin interfaz gráfica. Si no se le pasa el nombre del fichero de entrada como argumento, la aplicacion lo pedirá más adelante.

- **compiler:** archivos relativos al compilador. Archivos:
  - **Compiler.java.** En esta clase está todo el código necesario para la traducción del programa y el control de restricciones. Va generando el código y gestiona los errores que se van produciendo en las diferentes fases.
  - **GrammarBaseListener.java.** Generado por ANTLR a partir de la gramática.
  - **GrammarListener.java.** Generado por ANTLR a partir de la gramática.
  - **DescriptiveErrorListener.java.** Listener para los errores basado en el listener base de errores ofrecido por ANTLR a partir de la gramática.
    Envía los errores de sintáxis que encuentra ANTLR a nuestro Log.
  - **GrammarLexer.java.** Generado por ANTLR a partir de la gramática.
  - **GrammarParser.java.** Generado por ANTLR a partir de la gramática.
  - **Grammar.tokens.** Generado por ANTLR a partir de la gramática.
  - **GrammarLexer.tokens.** Generado por ANTLR a partir de la grámatica.

- **gui:** archivos relativos a la interfaz gráfica. Archivos:
  - **AppWindow.java.** Clase con todos los elementos de la interfaz gráfica, así como los listeners de los botones, que interactúan con la aplicación mediante el CmdLineInterface.

- **utils:** archivos adicionales, por ahora sólo los logs. Archivos:
  - **Logs.java.** En esta clase se definen los logs estáticos en los que la aplicación va guardando los errores que se tengan que mostrar luego en la interfaz gráfica.