Procesadores de Lenguaje: Práctica Anual

Ingeniería en Informática 4º C

Facultad de Informática UCM (2009-2010)

1ª Entrega

Grupo: 3

Miembros: Ortiz Jaureguizar, Gonzalo
Pérez Jiménez, Alicia
Reyero Sainz, Laura
Sanjuán Redondo, Héctor
Tarancón Garijo, Rubén

Introducción
Muy breve descripción del contenido de la memoria y de su estructura.
Se recomienda crear un índice para acceder directamente a cada una de las secciones, numerar las páginas, etc.

0. Descripción del lenguaje fuente
En esta sección se debe describir informalmente cómo es el lenguaje de programación de alto nivel que se va a procesar. En este apartado basta con copiar las características que os proponemos a continuación (salvo las restricciones contextuales que se llevarán al punto 4.1).

Características que debe tener el lenguaje definido:

Los programas constarán de una *sección de declaraciones* y de una *sección de instrucciones*, separadas mediante el símbolo **&**

La sección de declaraciones constará de una secuencia de una o más *declaraciones*, separadas entre sí por el símbolo **;**

El lenguaje *distingue cadenas según estén escritas con letras mayúsculas o minúsculas*, tanto si son palabras reservadas (que se escriben *siempre en minúsculas*) como si son identificadores de variables

Es decir, aunque dos identificadores tengan las mismas letras, se consideran distintos si no coinciden en mayúsculas y minúsculas

Cada declaración constará de una *variable*, seguida del símbolo **:** y a continuación el nombre de un *tipo.*

Los identificadores de las variables comienzan necesariamente por una letra, seguida de cero o más letras y dígitos

En la sección de declaraciones no podrá haber variables duplicadas

El tipo podrá ser boolean (valores booleanos true y false), character (caracteres alfanuméricos), natural (números naturales), integer (números enteros con signo) y float (números reales)

La sección de instrucciones constará de una secuencia de una o más *instrucciones*, separadas por el símbolo **;**

El lenguaje sólo tendrá tres tipos de instrucciones: *instrucciones de asignación*, *instrucciones de lectura* e *instrucciones de escritura*

Una instrucción de asignación consta de una variable, seguida de los símbolos **:=** y a continuación una *expresión*

Las expresiones usarán los siguientes operadores: <, >, <=, >=, =, =/=, +, -, *, /, %, and, or, not, - (unario), <<, >>, (float), (int), (nat), (char) y ||

- El menor nivel de prioridad (nivel 0) es el de los operadores de comparación < (menor que) > (mayor que), <= (menor o igual que), >= (mayor o igual que), = (igual) y =/= (distinto):
  - Todos estos operadores son binarios infijos y no asocian
  - Es posible comparar entre sí valores numéricos (natural con natural, natural con entero, natural con real, entero con natural, entero con entero, entero con real, real c on natural, real con entero, real con real), caracteres (según la ordenación del estándar de caracteres usado en la implementación, por ejemplo UNICODE) y booleanos (true se considera mayor que false)
  - El resultado de una comparación es un valor booleano (true cuando se cumple la comparación, false cuando no se cumple)
- El siguiente nivel de prioridad (nivel 1) es el de los operadores aritméticos + (suma) y (resta), así como el del operador lógico **or**
  - Todos ellos son operadores binarios infijos que asocian a izquierdas
  - + y - operaran sobre valores numéricos. El resultado será un valor real (siempre y cuando alguno de los operandos sea real), un valor entero (siempre y cuando alguno de los operandos sea entero y no haya operandos reales) o un natural (sólo cuando los dos operandos sea naturales)
  - **or** opera sobre valores booleanos. El resultado será el *o lógico* de los operandos

- El siguiente nivel de prioridad (nivel 2) es el de los operadores aritméticos ***** (multiplicación), / (división) y **%** (módulo), así como el operador lógico **and**
    - Todos ellos son operadores binarios infijos que asocian a izquierdas
    ***** y **/** operan sobre valores numéricos. El resultado será un valor real (siempre y cuando alguno de los operandos sea real), un valor entero (siempre y cuando alguno de los operandos sea entero y no haya operandos reales) o un natural (sólo cuando los dos operandos sea naturales). En el primer caso, la división funcionará como división real. En los dos últimos casos, la división funcionará como división entera
    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
    - Por último, **and** opera únicamente sobre valores booleanos. El resultado será el *y lógico* de los operandos
- El siguiente nivel de prioridad (nivel 3) es el de los operadores de desplazamiento << y >>
    - Ambos son operadores binarios infijos que asocian a derechas
    - Operan únicamente sobre valores naturales. Por un lado *a* << *b* es el natural resultante de desplazar *b* bits hacia la izquierda en la representación binaria de *a*. Por otro lado *a* >> *b* es el natural resultante de desplazar *b* bit hacia la derecha en la representación binaria de *a*
- El mayor nivel de prioridad (nivel 4) es el del operador lógico **not**, el - unario, los operadores de *conversión* **(float)**, **(int)**, **(nat)** y **(char)**, y el operador *valor absoluto* **|**
    - Todos son operadores unarios prefijos, salvo el valor absoluto que sitúa al operando dentro de sus dos barras verticales
    - El operador *negación lógica* **not** asocia, y opera sobre valores booleanos, siendo el resultado la negación lógica de su operando
    - El operador **–** unario también asocia, opera sobre valores numéricos y el resultado es su operando cambiado de signo. Este resultado será real si el tipo del operando es real, y entero en otro caso
    - Los operadores de conversión no asocian.
    - **(float) *a*** devuelve el propio *a* si es real. Si *a* es entero o natural el resultado es *a* convertido a real (añadiendo un único 0 en la parte decimal). Si *a* es un carácter el resultado es el código de dicho carácter convertido a real.
    - **(int) *a*** devuelve el propio *a* si es entero, Si *a* es real el resultado es la parte entera de *a*. Si *a* es natural el resultado es *a* convertido a entero (con signo positivo). Si *a* es un carácter el resultado es el código de dicho carácter convertido a entero.
    - **(nat) *a*** devuelve el propio *a* si es natural. No admite operandos reales o enteros. Si *a* es un carácter el resultado es el código de dicho carácter.
    - **(char) *a***.devuelve el propio *a* si es un carácter. Si *a* es natural el resultado es el carácter cuyo código es *a*. No admite operandos reales o enteros.
    - El operador *valor absoluto* está formado por el símbolo de barra vertical **|** seguido de una expresión, seguida otra vez del símbolo de barra vertical |. El resultado es el valor absoluto de la expresión, de tipo real si la expresión evalúa a un número real, o natural si la expresión evalúa a un número entero o natural. No admite expresiones que evalúen a booleano o carácter**.**
- Como expresiones básicas, que podrán ser combinadas mediante los operadores anteriores, se consideran las siguientes:
    - Literales *naturales*. Secuencias de uno o más dígitos, no admitiéndose ceros a la izquierda
    - Literales *reales*. Secuencia formada por una parte entera, seguida de una de estas tres cosas: una parte decimal, una parte exponencial, o una parte decimal seguida de una parte exponencial. La parte entera tiene la misma estructura que un literal natural. La parte decimal está formada por el símbolo **.** seguido de uno o más dígitos, no admitiéndose ceros a la derecha (salvo si se trata de un único cero, como por ejemplo 2.0). La parte exponencial está formada

por el símbolo **E** o el símbolo **e**, seguido opcionalmente del símbolo -, y seguido obligatoriamente de la misma estructura que tiene un literal natural
- Literales *booleanos*. Con valores **true** y **false**
- Literales *carácter*. Un símbolo de comilla simple ' seguido de un carácter alfanumérico, seguido otra vez del símbolo de comilla simple '
- Variables, que han debido ser convenientemente declaradas en la sección de declaraciones

- En las expresiones es posible utilizar paréntesis para alterar la forma en la que se aplican los operadores sobre los operandos
- Una instrucción de asignación debe cumplir además estas condiciones:
  - La variable en la parte izquierda debe haber sido declarada
  - A una variable de tipo real es posible asignarle un valor real, entero o natural (produciéndose automáticamente la correspondiente conversión), pero no un carácter
  - A una variable de tipo entero es posible asignarle un valor entero o natural (produciéndose automáticamente la correspondiente conversión), pero no un valor real o un carácter
  - 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
- Una instrucción de lectura tiene la forma **in(*v*)** donde *v* es una variable. Su efecto es leer un valor del tipo de la variable *v* por la entrada estándar del usuario y almacenar en la variable *v* el valor leído
- Una instrucción de escritura tiene la forma **out(*exp*)**, donde *exp* es una expresión. Su efecto es escribir por la salida estándar del usuario el valor de *exp*
- El lenguaje admite *comentarios de línea*. Dichos comentarios comienzan con el símbolo **#** y se extienden hasta el fin de la línea

Ejemplo de programa en el lenguaje definido:

```
# Programa de ejemplo
cantidad: float;
euros: integer;
centimos: float
&
in(cantidad);
euros := (int)cantidad;
centimos := cantidad – euros;
out(euros);
out('.');
out(centimos);
out((char)10)  # Código del carácter salto de línea
```

1. Definición léxica del lenguaje

Especificación formal del léxico del lenguaje (a veces llamado microsintaxis) utilizando definiciones regulares.

```
& ::= &
id ::= [a-zA-Z][a-zA-Z0-9]*
: ::= :
:= ::= :=
tipo ::= boolean | character | natural | integer | float
; ::= ;
< ::= <
> ::= >
<= ::= <=
>= ::= >=
```

```
=::=  =
=/= ::= =/=
+ ::= \+
- ::= \-
* ::= \*
 / ::= /
% ::= %
and ::= and
or ::= or
not ::= not
<< ::= <<
>> ::= >>
(nat) ::= \(nat\)
(int) ::= \(int\)
(char) ::= \(char\)
(float) ::= \(float\)
litNat ::= ([1-9][0-9]*|0)
litFlo ::= ([1-9][0-9]*|0)(
                        \.0|
                        \.([0-9]*[1-9])|
                        \.([0-9]*[1-9])(e|E)-?([1-9][0-9]*|0)|
                        (e|E)-?[1-9][0-9]*|0
          )
litTrue ::= true
litFalse ::= false
litCha ::= '[a-zA-Z0-9]'
| ::= \|
( ::= \(
) ::= \)
comentario ::= #.*\\n
in := in
out := out
```

2. Definición sintáctica del lenguaje

Especificación formal de los aspectos sintácticos del lenguaje.

2.1. Descripción de los operadores

Los operadores del lenguaje son los siguientes. Todos asocian a derechas, excepto los de nivel 3:

Operador	**Aridad:**
Nivel 0 – menor prioridad	
<	2
>	2
<=	2
>=	2
=	2
=/=	2
Nivel 1	
+	2
-	2
or	2
Nivel 2	
*	2
/	2
%	2
and	2
Nivel 3	
>>	2

<<	2

Nivel 4 – mayor prioridad

not	1
-	1
(float)	1
(int)	1
(nat)	1
(char)	1
| |	1

2.2. Formalización de la sintaxis

La gramática que formaliza nuestro lenguaje es la siguiente:

Programa → Declaraciones & Instrucciones

Declaraciones → Declaración ; Declaraciones
Declaraciones → Declaración
Declaración → id : Tipo

Tipo → Boolean
Tipo → character
Tipo → Float
Tipo → Natural

Instrucciones → Instrucción ; Instrucciones
Instrucciones → Instrucción
Instrucción → InsLectura
Instrucción → InsEscritura
Instrucción → InsAsignación

InsLectura → in(id)
InsEscritura → out(Expresion)
InsAsignación → id := Expresión

Expresión → ExpresiónNiv1 OpNiv0 ExpresiónNiv1
Expresión → ExpresiónNiv1

ExpresiónNiv1 → ExpresiónNiv1 OpNiv1 ExpresiónNiv2
ExpresiónNiv1 → ExpresiónNiv2

ExpresiónNiv2 → ExpresiónNiv2 OpNiv2 ExpresiónNiv3
ExpresiónNiv2 → ExpresiónNiv3

ExpresiónNiv3 → ExpresionNiv4 OpNiv3 ExpresiónNiv3
ExpresiónNiv3 → ExpresiónNiv4

ExpresiónNiv4 → OpNiv4 ExpresiónNiv4
ExpresiónNiv4 → | Expresión |
ExpresiónNiv4 → (Expresión)
ExpresiónNiv4 → Literal

Literal → id
Literal → litNat
Literal → litFlo
Literal → litTrue
Litera → litFalse
Literal → litCha

OpNiv0 → <
OpNiv0 → >

OpNiv0 → <=
OpNiv0 → >=
OpNiv0 → =
OpNiv0 → =/=

OpNiv1 → +
OpNiv1 → -
OpNiv1 → or

OpNiv2 → *
OpNiv2 → /
OpNiv2 → %
OpNiv2 → and

OpNiv3 → >>
OpNiv3 → <<

OpNiv4 → not
OpNiv4 → -
OpNiv4 → (float)
OpNiv4 → (int)
OpNiv4 → (nat)
OpNiv4 → (char)

3. Estructura y construcción de la tabla de símbolos

Nuestro lenguaje no está estructurado en bloques, por tanto no nos es necesario una gestión de ámbitos para controlar si en un determinado momento una variable esta disponible o no, ya que todas las variables se declaran al inicio del programa y con accesibles en todas las líneas hasta el fin del programa. Por este motivo hemos utilizado una única tabla de símbolos.

La información que necesitamos almacenar es la siguiente:

El símbolo que identifica la variable

Una colección de propiedades asociadas a la variable, como pueden ser el tipo o la dirección de memoria.

Tabla de Símbolos	
▪ gedad”	<dir:100,tipo:entero>
▪ gdni”	<dir:101,tipo:entero>
▪ c	...

3.1. Estructura de la tabla de símbolos

Descripción de las operaciones de la tabla de símbolos definiendo la cabecera de dichas operaciones, así como describiendo informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros.

creaTS():TS

El resultado es una TS vacía

inserta(ts:TS, id:String, ps: Propiedades):TS

El resultado es la TS resultante de añadir id y sus propiedades ps a ts

existe(ts: TS, id:String):Boolean

El resultado es true si id aparece en ts, false en caso contrario

getDir(ts:TS id:String):Natural

El valor de la propiedad “direccion” del atributo

getTipo(ts:TS id:String):Tipo

Devuelve el valor de la propiedad “tipo” del atributo.

3.2. Construcción de la tabla de símbolos

Formalización de la construcción de la tabla de símbolos mediante una gramática de atributos.

3.2.1 Funciones semánticas

3.2.2 Atributos semánticos

Para cada categoría sintáctica relevante en este procesamiento enumeramos sus atributos semánticos, indicando si son heredados o sintetizados, y describiendo informalmente su propósito.

Categoría Programa:

ts: sintetizado. Representa la tabla de símbolos del programa.

Categoría Declaraciones:

ts: sintetizado. Representa la tabla de símbolos del programa conforme se va construyendo.

Categoría Declaración:

tipo: sintetizado. Indica el tipo con el que se declara la variable.

id: sintetizado. Indica el id de la entrada de la tabla de símbolos.

3.2.3 Gramática de atributos

Gramática de atributos que formaliza la construcción de la tabla de símbolos.

Programa → Declaraciones & Instrucciones

Programa.ts = Declaraciones.ts

Declaraciones → Declaracion ; Declaraciones

Declaraciones0.ts = inserta(Declaraciones1.ts, Declaracion.id, <tipo:Declaracion.tipo>)

Declaraciones → Declaracion

Declaraciones.ts = inserta(creaTS(), Declaracion.id, <tipo:Declaracion.tipo>)

Declaracion → id : Tipo

Declaracion.id = id.lex

Declaracion.tipo = Tipo.tipo

Tipo → Boolean

Tipo.tipo = boolean

Tipo → character

Tipo.tipo = character

Tipo → Float

Tipo.tipo = float

Tipo → Natural

Tipo.tipo = natural

4. Especificación de las restricciones contextuales

4.1. Descripción informal de las restricciones contextuales

Enumeración y descripción informal de las restricciones contextuales del lenguaje. En este apartado basta con copiar las restricciones contextuales que os proponemos al principio de la plantilla, cuando describimos las características del lenguaje.

Las restricciones contextuales relativas a esta práctica tienen que ver principalmente con que el tipo de las expresiones estén bien definidos y coincidan con el que permiten los operadores con las que van asociadas. Las restantes tienen que ver con la declaración de las variables y las instrucciones. Para las primeras he definido un atributo “tipo” y para las segundas un atributo “error”. Estos dos atributos tienen su enlace, ya que un posible valor del atributo “tipo” es error.

Las restricciones que tienen que ver con la declaración de las variables y las instrucciones son:

En la sección de declaraciones no podrá haber variables duplicadas

En la instrucción de asignación, el identificador debe haber sido declarado.

A una variable de tipo real es posible asignarle un valor real, entero o natural.

A una variable de tipo entero es posible asignarle un valor entero o real.

A una variable de tipo nat, char o bool solo les puedes asignarles valores del mismo tipo.

En la instrucción de lectura el identificador debe haber sido definido antes.

Las restricciones contextuales de las de las expresiones son las siguientes:

Los operadores <, >, <=, >=, =/= son capaces de comparar entre si valores numéricos (natural con natural, natural con entero, natural con real, entero con natural, entero con entero, entero con real, real con natural con entero, real con real) caracteres y booleanos.

Los operadores +y –operan solo sobre valores numéricos. El resultado será un valor real (siempre y cuando alguno de los operadores sea real), un valor entero (siempre y cuando alguno del os operandos sea entero y no haya operando reales) o un natural (solo cuando los dos operandos sean naturales).

El operador **or**opera únicamente sobre valores booleanos.

Los operadores *****y / operan solo sobre valores numéricos. El resultado será un valor real (siempre y cuando alguno de los operandos sea real), un valor entero (siempre y cuando alguno de los operandos sea entero y no haya operandos reales) o un natural (solo cuando los dos operandos sea naturales) o un natural (solo cuando los dos operandos sean naturales).

En el operador **%** no opera sobre valores reales, caracteres y booleanos. El primer operando puede ser entero o natural, pero el segundo operando solo puede ser natural. El tipo del resultado será el mismo que el del primer operando.

El operador **and**opera solo sobre valores booleanos.

Los operadores <<y >>operan únicamente sobre valores naturales.

El operador **not**opera solo sobre valores booleanos.

El operador **–**opera solo sobre valores numéricos y el resultado es su operando cambiado de signo. Este resultado será real si el tipo del operando es real, y entero en otro caso.

El operador **(float)**admite expresiones reales, enteras, de naturales y de caracteres. En todas el resultado es real.

El operador **(int)**admite expresiones reales, enteras, de naturales y de caracteres. En todas el resultado es entero.

El operador **(nat)**admite solo expresiones de naturales y de caracteres y en ambas el resultado es natural.

El operador **(char)**admite solo expresiones de naturales y de caracteres y en ambas el resultado es caracter.

El operador **valor absoluto**solo admite expresiones numéricas y el resultado es real y la expresión es real o natural si la expresión que evalúa es natural o entera.

4.2. Funciones semánticas

4.3. Atributos semánticos

Para cada categoría sintáctica relevante en este procesamiento enumeramos sus atributos semánticos, indicando si son heredados o sintetizados, y describiendo informalmente su propósito.

Categoría Programa:

error: sintetizado. Sirve para indicar si hay errores contextuales del programa.
Categoría Declaraciones:
error: sintetizado. Indica si hay errores en las declaraciones.
Categoría Instrucciones:
error: sintetizado. Indica si hay errores en la sección de instrucciones.
Categoría InstrucciónAsig:
lex: sintetizado. Proviene del analizador léxico y nos da el nombre del identificador.
tipo: sintetizado. Indica el tipo del identificador.
Categoria InsLectura:
error: indica error en instrucción de lectura.
Categoría InsEscritura:
error: indica error en la instrucción de escritura.
Categoría Expresión:
tipo: sintetizado. Contiene el tipo de la expresión y, en su caso, error si existe un error de tipos.
Categorías opNivX:
op: sintetizado. Contiene el tipo de operación.

Por ultimo hay un atributo heredado "tsh" que es la tabla de símbolos que proviene de las declaraciones y que heredan las categorías sintácticas Instrucciones, Instruccion, InsLectura, InsEscritura, InsAsignacion, Expresion, ExpresionNiv1, ExpresionNiv2, ExpresionNiv3 y ExpresionNiv4

4.4. Gramática de atributos

Gramática de atributos que formaliza la comprobación de las restricciones contextuales.

Programa → Declaraciones & Instrucciones
Programa.error = Declaraciones.error v Instrucciones.error
Instrucciones.tsh = Declaraciones.ts
Declaraciones → Declaración ; Declaraciones
$Declaraciones_0$.error = existe($Declaraciones_0$.ts,Declaracion.id) v $Declaraciones_1$.error
Declaraciones → Declaración
Declaraciones.error = Declaracion.error
Declaración → id : Tipo
Declaración.id = id.lex
Declaración.tipo = Tipo.tipo
Declaración.error = (id=null || tipo =null)
Tipo → Boolean
Tipo.tipo = boolean
Tipo → character
Tipo.tipo = character
Tipo → Float
Tipo.tipo = float
Tipo → Natural
Tipo.tipo = natural
Instrucciones → Instrucción ; Instrucciones
$Instrucciones_0$.error = Instrucción.error v $Instrucciones_1$.error
$Instrucciones_1$.tsh = Instruccion.tsh = $instrucciónes_0$.tsh
Instrucciones → Instrucción
Instrucciones.error = Instrucción.error
Instrucción.tsh = Instrucciones.tsh
Instrucción → InsLectura
Instrucción.error = InsLectura.error
InsLectura.tsh = Instrucción.tsh
Instrucción → InsEscritura

Instrucción.error = InsEscritura.error
InsEscritura.tsh = Instrucción.tsh

Instrucción → InsAsignacion

Instrucción.error = InsAsignacion.error
InsAsignacion.tsh = Instrucción.tsh

InsLectura → in(id)

InsLectura.error = NOT existeID(InsLectura.tsh,id.lex)

InsEscritura → out(Expresion)

InsEscritura.error = (Expresion.tipo = error)

InsAsignación → id := Expresión

InsAsignacion.error = (NOT existeID(InsAsignacion.tsh,id.lex)) v (Expresion.tipo = error) v
(InsAsignación.tsh[id.lex].tipo = float ∧
(Expresion.tipo = character v Expresion.tipo = boolean)) v
(InsAsignación.tsh[id.lex].tipo = integer ∧
(Expresion.tipo=float v Expresion.tipo = character v Expresion.tipo = boolean)) v
(InsAsignación.tsh[id.lex].tipo = natural ∧ Expresion.tipo =/= natural) v
(InsAsignación.tsh[id.lex].tipo = character ∧ Expresion.tipo =/= character) v
(InsAsignación.tsh[id.lex].tipo = boolean ∧ Expresion.tipo =/= boolean)

Expresion.tsh = InsAsignacion.tsh

Expresión → ExpresiónNiv1 OpNiv0 ExpresiónNiv1

Expresion.tipo = si (ExpresionNiv1$_0$.tipo = error v ExpresionNiv1$_1$.tipo = error) v
(ExpresionNiv1$_0$.tipo = character ∧ ExpresionNiv1$_1$.tipo =/= character) v
(ExpresionNiv1$_0$.tipo =/= character ∧ ExpresionNiv1$_1$.tipo = character)
(ExpresionNiv1$_0$.tipo = boolean ∧ ExpresionNiv1$_1$.tipo =/= boolean) v
(ExpresionNiv1$_0$.tipo =/= boolean ∧ ExpresionNiv1$_1$.tipo = boolean))
error
sino boolean

ExpresionNiv1$_1$.tsh = ExpresionNiv1$_0$.tsh = Expresion.tsh

Expresión → ExpresiónNiv1

Expresion.tipo = ExpresionNiv1.tipo
ExpresionNiv1.tsh = Expresion.tsh

ExpresiónNiv1 → ExpresiónNiv1 OpNiv1 ExpresiónNiv2

ExpresionNiv1$_0$.tipo =
si (ExpresionNiv1$_1$.tipo = error v ExpresionNiv2.tipo = error v
ExpresionNiv1$_1$.tipo = char v ExpresionNiv2.tipo = char v
(ExpresionNiv1$_1$.tipo = boolean ∧ ExpresionNiv2.tipo =/= boolean) v
(ExpresionNiv1$_1$.tipo =/= boolean ∧ ExpresionNiv2.tipo = boolean))
error
sino case (OpNiv1.op)
suma,resta:
si (ExpresionNiv1$_1$.tipo=float v ExpresionNiv2.tipo = float)
float
sino si (ExpresionNiv1$_1$.tipo =integer v ExpresionNiv2.tipo = integer)
integer
sino si (ExpresionNiv1$_1$.tipo =natural ∧ ExpresionNiv2.tipo = natural)
natural
sino error
o:
si (ExpresionNiv1$_1$.tipo = boolean ∧ ExpresionNiv2.tipo = boolean)
boolean
sino error

ExpresionNiv2.tsh = ExpresionNiv1$_1$.tsh = ExpresionNiv1$_0$.tsh

ExpresiónNiv1 → ExpresiónNiv2

ExpresionNiv1.tipo = ExpresionNiv2.tipo
ExpresionNiv2.tsh = ExpresionNiv1.tsh

ExpresiónNiv2 → ExpresiónNiv2 OpNiv2 ExpresiónNiv3

ExpresionNiv2$_0$.tipo =

si (ExpresionNiv2$_1$.tipo = error v ExpresionNiv3.tipo = error v
ExpresionNiv2$_1$.tipo = character v ExpresionNiv3.tipo = character v
(ExpresionNiv2$_1$.tipo = boolean ∧ ExpresionNiv3.tipo =/= boolean v
(ExpresionNiv2$_1$.tipo =/= boolean ∧ ExpresionNiv3.tipo = boolean))
error
sino case (OpNiv2.op)
multiplica,divide:
si (ExpresionNiv2$_1$.tipo=float v ExpresionNiv3.tipo = float)
float
sino si (ExpresionNiv2$_1$.tipo =integer v ExpresionNiv3.tipo = integer)
integer
sino si (ExpresionNiv2$_1$.tipo =natural ∧ ExpresionNiv3.tipo=natural)
natural
sino error
modulo:
si (ExpresionNiv3.tipo = natural ∧
(ExpresionNiv2$_1$.tipo=natural v ExpresionNiv2$_1$.tipo=integer))
ExpresionNiv2$_1$.tipo
sino error
y:
si (ExpresionNiv2$_1$.tipo = boolean ∧ ExpresionNiv3.tipo = boolean)
boolean
sino error

ExpresionNiv3.tsh = ExpresionNiv2$_1$.tsh = ExpresionNiv2$_0$.tsh

ExpresiónNiv2 → ExpresiónNiv3

ExpresionNiv2.tipo = ExpresionNiv3.tipo
ExpresionNiv3.tsh = ExpresionNiv2.tsh

ExpresiónNiv3 → ExpresionNiv4 OpNiv3 ExpresiónNiv3

ExpresionNiv3$_0$.tipo =

si (ExpresionNiv4.tipo = error v ExpresionNiv3$_1$.tipo = error v
ExpresionNiv4.tipo =/= natural v ExpresionNiv3$_1$.tipo =/= natural)
error
sino natural

ExpresionNiv4.tsh = ExpresionNiv3$_1$.tsh = ExpresionNiv3$_0$.tsh

ExpresiónNiv3 → ExpresiónNiv4

ExpresionNiv3.tipo =ExpresionNiv4.tipo
ExpresionNiv4.tsh = ExpresionNiv3.tsh

ExpresiónNiv4 → OpNiv4 ExpresiónNiv4

ExpresionNiv4$_0$.tipo =

si (ExpresionNiv4$_1$.tipo = error)
error
sino case (OpNiv4.op)
no:
si (ExpresionNiv4$_1$.tipo=boolean)
boolean
sino error
menos:
si (ExpresionNiv4$_1$.tipo=float)
float
sino si (ExpresionNiv4$_1$.tipo=integer v ExpresionNiv4$_1$.tipo = natural)
integer
sino error

cast-float:
si (ExpresionNiv4$_1$.tipo=/=boolean)
float
sino error
cast-int:
si (ExpresionNiv4$_1$.tipo=/=boolean)
integer
sino error
cast-nat:
si (ExpresionNiv4$_1$.tipo=natural v ExpresionNiv4$_1$.tipo=character)
natural
sino error
cast-char:
si (ExpresionNiv4$_1$.tipo=natural v ExpresionNiv4$_1$.tipo=character)
character
sino error

ExpresionNiv4$_1$.tsh = ExpresionNiv4$_o$.tsh

ExpresiónNiv4 → | Expresión |

ExpresionNiv4.tipo =
si (Expresion.tipo = error v Expresion.tipo=boolean v Expresion.tipo=character)
error
sino si (Expresion.tipo = float)
float
sino si (Expresion.tipo = natural v Expresion.tipo = integer)
natural
sino error

Expresion.tsh = ExpresionNiv4.tsh

ExpresiónNiv4 → (Expresión)

ExpresionNiv4.tipo = Expresion.tipo
Expresion.tsh = ExpresionNiv4.tsh

ExpresiónNiv4 → Literal

Expresion.tipo = Literal.tipo
Literal.tsh = ExpresiónNiv4.tsh

Literal → id

Literal.tipo = Literal.tsh[id.lex].tipo

Literal → litNat

Literal.tipo = natural

Literal → litFlo

Literal.tipo = float

Literal → litTrue

Literal.tipo = boolean

Literal → litFalse

Literal.tipo = boolean

Literal → litCha

Literal.tipo = character

OpNiv0 → <

OpNiv0.op = menor

OpNiv0 → >

OpNiv0.op = mayor

OpNiv0 → <=

OpNiv0.op = menor-ig

OpNiv0 → >=

OpNiv0.op = mayor-ig

OpNiv0 → =

OpNiv0.op = igual

OpNiv0 → =/=

OpNiv0.op = no-igual

OpNiv1 → +

OpNiv1.op = suma

OpNiv1 → -
    OpNiv1.op = resta
OpNiv1 → or
    OpNiv1.op = o
OpNiv2 → *
    OpNiv2.op = multiplica
OpNiv2 → /
    OpNiv2.op = divide
OpNiv2 → %
    OpNiv2.op = modulo
OpNiv2 → and
    OpNiv2.op = y
OpNiv3 → >>
    OpNiv3.op = shl
OpNiv3 → <<
    OpNiv3.op = shr
OpNiv4 → not
    OpNiv4.op = no
OpNiv4 → -
    OpNiv4.op = menos
OpNiv4 → (float)
    OpNiv4.op = cast-float
OpNiv4 → (int)
    OpNiv4.op = cast-int
OpNiv4 → (nat)
    OpNiv4.op = cast-nat
OpNiv4 → (char)
    OpNiv4.op = cast-char

5. Especificación de la traducción

El compilador será capaz de compilar a dos lenguajes disenteroos. El primero es el codigo P definido en clase y el segundo es el java bytecode, definido en la **especificación de la máquina virtual java**. Por ello en este punto se desarrollará cada punto para ambos lenguajes.

5.1. Lenguaje objeto y máquina virtual

Explicar cómo es el lenguaje objeto y cómo es la arquitectura de la máquina P capaz de ejecutarlo (tipos de celdas según tipos primitivos, etc.), su comportamiento interno y todo su repertorio de instrucciones (mostrando su sintaxis y una descripción informal de su semántica).

5.1.1. Arquitectura

5.1.1.2 Arquitectura de la máquina P

La máquina P que se adjunta, implementada en Java, consiste en:

· Una pila teóricamente infinita (su tamaño máximo real depende del sistema en el cual se ejecute) capaz de guardar datos.

· Una lista teóricamente infinita que representa el programa a ejecutarse.

· Una memoria de tamaño configurable donde se guardan los valores de las variables.

· Un booleano “parar” que indica si la máquina debe o no leer la siguiente instrucción. Cuando valga cierto la máquina dejará de ejecutar el programa.

· Un entero “cp” cuya función es servir de índice dentro del programa, señalando la instrucción que se está ejecutando en cada momento.

Cada celda de la pila y de la memoria es capaz de contener cualquier tipo de dato independientemente de su tamaño.
Cada celda de la lista que representa el programa es capaz de guardar cualquier tipo de instrucción independientemente de su tamaño.

5.1.1.2 Arquitectura del código java bytecode

La especificación de Sun deja más o menos abierta la arquitectura de las maquinas virtuales java para así permitir su implementación sobre cualquier dispositivo. Una máquina virtual debe poder soportar la pila de argumentos, llamadas recursivas, tener un heap... Sin embargo no se especifica como debe implementarse la pila de argumentos (es decir, no dice si todos los datos ocupan el mismo número de celdas, por ejemplo). Sin embargo si hace diferencia entre datos de 32 bits y datos de 64 bits en las referencias a las variables locales y los argumentos de un método, de tal manera que podríamos considerar que existen celdas de 32 bits y que los argumentos de 64 bits ocupan dos de estas celdas.

La naturaleza orientada a objetos del java bytecode sobrepasa con creces el objetivo de la práctica.

5.1.2. Comportamiento interno

5.1.2.1 Comportamiento interno de la máquina P

El funcionamiento de la máquina P es:

```
pila = pilaVacia();
cp ← 0
parar ← false
mientras(no parar)
    instrucción = programa[cp]
    ejecutar(instruccion)
    cp++
```

Esto se repite un número indeterminado de veces, parando cuando ejecute una instrucción “parar” o

cuando se produzca un fallo en ejecucución (como podría ser una división por cero o encontrar tipos de valores incompatibles al realizar una operación).

5.1.2.2 Comportamiento interno de la JVM

La especificación de la JVM más que un comportamiento obliga a que cualquier implementación de una máquina virtual de java de soporte a una serie de funcionalidades y que interprete correctamente un fichero .class. Debe permitir crear elementos en el heap, tener una pila de marcos similar a la de un programa en c, ser capaz de recolectar automáticamente la memoria de heap a la cual no haya referencias, etc. Una vez cumplidas estas funcionalidades, cada jvm puede implementarlas de maneras disenteroas (con 64 para servidores o con recolector de basura incremental, por ejemplo) y añadir otras características como la precompilación a codigo nativo de las instrucciones bytecode en lugar de interpretarlas.

Dentro del subconjuto de instrucciones java bytecode que usamos, la jvm se comporta de una manera similar a la máquina P

5.1.3. Repertorio de instrucciones

5.1.3.1 Repertorio de instrucciones del código P

El lenguaje objeto es un sencillo lenguaje de pila, es decir, las instrucciones apilan o desapilan datos (según indique su semántica) en una pila teóricamente infinita, teniendo como apoyo una memoria de acceso aleatorio donde almacenar las variables.
El lenguaje tiene los mismos tipos que el lenguaje fuente:

- Bool puede ser “true” o “false”. Su código único es el 1
- Character es un dato que ocupa dos bytes que representa caracteres en formato UTF-8.
- Entero es un número entero en C2 de 32 bits.
- Natural es un número natural de 31 bits.
- Float es un número real en IEEE 754 de 32 bits.

La tabla siguiente muestra las instrucciones de este lenguaje.

- La columna “Código” indica el valor que tiene internamente la instrucción.
- Cada celda de la columna “Args” puede ser 0 (es decir, no tiene argumentos) o “<tipo del argumento> <nombre del argumento>”, el nombre del argumento puede ser usado en las dos siguientes celdas.
- La columna “Interacción con la pila” sigue la siguiente sintaxis:
  - Lo que se encuentre a la izquierda de → es el estado anterior de la pila y lo de la derecha el estado posterior (tras aplicar la instrucción).
  - Las “,” separan los elementos de la pila.
  - Los tres puntos (“…”) significan “el resto de la pila”.
  - La estructura “<tipo> <nombre>” especifica un valor en la pila.
- Los tipos pueden ser o bien los definidos en el lenguaje (Nat, Bool, Int, etc) o bien otra palabra en mayúsculas (por ejemplo “T”), en cuyo caso se tratará de un tipo genérico, pero todas las apariciones de esa palabra corresponderán al mismo tipo.

Por ejemplo: “…, T v1, T v2 → …, Bool res” indica que la instrucción desapila 2 variables y apila una variable booleana. Además v1 y v2 deben tener el mismo tipo.

Código	Nombre mnemotécnico	Args	Interacción con la pila	Descripción
0	Parar	0	… → …	Detiene la ejecución.
1	Apila	T dato	… → …, T dato	Apila en la pila el dato pasado como argumento.
2	Apila-dir	Nat dir	... → …, T M[dir]	Apila en la pila el contenido de la posición de memoria pasada como argumento.
3	Desapilar	0	…, T valor → …	Desapila el primer dato de la pila.
4	Desapilar-dir	Nat dir	…, T valor → …	Desapila el primer dato de la pila y lo almacena en la posición de memoria pasada como parámetro.
5	Menor	0	…, T v1, T v2 → …, Bool res	res = t1 < t2
6	Mayor	0	…, T v1, T v2 → …, Bool res	res = t1 > t2
7	MenorIg	0	…, T v1, T v2 → …, Bool res	res = t1 <= t2
8	MayorIg	0	…, T v1, T v2 → …, Bool res	res = t1 >= t2
9	Igual	0	…, T v1, T v2 → …, Bool res	res = t1 == t2
10	No-Igual	0	…, T v1, T v2 → …, Bool res	res = t1 != t2
11	Sumar	0	…, T v1, T v2 → …, T res	res = v1 + v2. T tiene que ser un tipo numérico.
12	Restar	0	…, T v1, T v2 → …, T res	res = v1 - v2. T tiene que ser un tipo numérico.
13	Mul	0	…, T v1, T v2 → …, T res	res = v1 * v2. T tiene que ser un tipo numérico.
14	Div	0	…, T v1, T v2 → …, T res	res = v1 / v2. T tiene que ser un tipo numérico.
15	Mod	0	…, T v1, Nat v2 → …, T res	res = v1 % v2. T tiene que ser un tipo numérico.
16	Y	0	…, Bool v1, Bool v2 → …, Bool res	res = v1 && v2
17	O	0	…, Bool v1, Bool v2 → …, Bool res	res = v1 || v2
18	No	0	…, Bool v → …, Bool res	res = !v1
19	Negativo	0	…, T v → T res	res = -v. T tiene que ser un tipo

				numérico distinto a Nat
20	Shl	0	…, T v1, Nat v2 → …, T res	res = v1 << v2. T tiene que ser un tipo numérico.
21	Shr	0	…, T v1, Nat v2 → …, T res	res = v1 >> v2. T tiene que ser un tipo numérico.
22	CastInt	0	…, T v → Int res	res = v como entero. Siguiendo la conversión especificada en este documento
23	CastChar	0	…, T v → Char res	res = v como caracter. Siguiendo la conversión especificada en este documento
24	CastFloat	0	…, T v → Float res	res = v como float. Siguiendo la conversión especificada en este documento
25	CastNat	0	…, T v → Nat res	res = v como natural. Siguiendo la conversión especificada en este documento
26	Abs	0	…, T v → …, T res	res = | v | (excepto char y bool)
27	Salida	0	… → T res	Muestra por pantalla el valor del identificador
28	Entrada_Bool	0	… → Bool id	Entrada para identificadores de tipo booleano.
29	Entrada_Char	0	… → Char id	Entrada para identificadores de tipo caracter.
30	Entrada_Float	0	… → Float id	Entrada para identificadores de tipo real.
31	Entrada_Int	0	… → Integer id	Entrada para identificadores de tipo entero.
32	Entrada_Nat	0	… → Nat id	Entrada para identificadores de tipo natural.

5.1.3.2 Repertorio de instrucciones del código java bytecode

El repertorio de instrucciones del java bytecode es muy extenso. Al igual que en el código P, estas instrucciones se codifican con 1 byte de información. A diferencia de las instrucciones del código P, las instrucciones jvm solo aceptan un tipo de argumento. Por ejemplo, en el código P la operación “suma” requiere que en la cabecera de la pila se encuentren dos argumentos numericos del mismo tipo. La instrucción “iadd” de jvm es más estricta, requiriendo 2 en la pila dos argumentos de tipo entero, existiendo además las instrucciones “ladd”, “fadd” y “dadd” para longs, floats y doubles respectivamente.

Debido al largo repertorio de instrucciones y la posible incorporación futura de otras tantas, así como la representación de byte que solo permite 256 instrucciones, la jvm no dispone de operaciones para trabajar con booleanos, caracteres o shorts. En la pila de un método java solo hay datos de tipo referencia, int, long, float y double. Los booleanos, caracteres y shorts se apilan como enteros (extendiendo el signo si hace falta) y se operan como tales. Es a la hora de guardarlo en la memoria asignada a la variable cuando se hace el casting al tipo de dato concreto.

Existen algunas instrucciones del lenguaje fuente que no son traducibles facilemente a jvm y hace falta hacer uso de objetos o saltos condicionales/incondicionales. Del primer tipo son las

instrucciones de lectura y escritura (que se acceden en java llamando a System.in y System.out respectivamente). Al segundo bloque pertenecen por ejemplo el valor absoluto, las operaciones lógicas (la or puede simularse con la or binaria, pero no ocurre lo mismo con la and lógica ni con la negación).

Podrían traducirse diréctamente estas operaciones a las llamadas a objetos o saltos condicionales, pero eso implicaría llevar numerosos atributos heredados y sintácticos que complicarían enormemente la gramática. Por eso se ha optado por poner en su lugar unas instrucciones inventadas (no pertenecientes a jvm) que en un proceso posterior se traducirán a jvm.

Se usaran las siguientes pseudoinstrucciones:

- Abs <tipo>: desapila el primer elemento y apila su valor absoluto. El tipo es necesario puesto que dependiendo del tipo este cálculo se hará de una u otra manera.
- ApilarFloat <valor>y ApilarInt <valor>: apilan un valor constante en la pila. En la pila de la JVM los chars, booleanos, bytes y shorts se manipulan como enteros. Es de destacar que para apilar un float distinto de 0, 1 o 2, hace falta crear una entrada en la tabla de constantes, momento a partir del cual se podra cargar usando esa entrada. Lo mismo ocurre con los enteros que se encuentran fuera del rango del tipo short de java.
- ApilarPrinter: para hacer llamadas a System.out hace falta apilar su referencia, luego el valor a escribir y luego hacer la llamada a la funcion correspondiente. Con esta pseudoinstrucción se apila la referencia a System.out.
- CargarDato <tipo> <dir>: carga una variable de tipo “tipo” de la dirección de memoria “dir”. Se podría sustituir por iload <dir> o fload <dir> según el tipo, pero la implementación de esta pseudoinstrucción usa instrucciones más rápidas para apilar las variables en las direcciones comprendidas entre la 0 y la 5.
- Lectura <tipo>: La clase que se compilará tiene algunas funciones auxiliares (implementadas en JVM por nosotros) que hacen llamadas a un atributo de tipo BufferedReader que permite leer los datos de la entrada estandar y luego transforman el string leido en el tipo correcto, dejándolo el dato en la cima de la pila
- GuardarDato <tipo> <dir>: Guarda un el dato de tipo “tipo” de la cima de la pila en la dirección “dir” de memoria.
- Negar: Niega el dato de la cima de la pila. Para JVM ese dato es entero (en la pila solo pueden haber enteros, floats, longs, doubles y referencias), pero en el código generado por nuestro compilador solo se aplica a booleanos.
- O: La o logica.
- Y: la y logica.
- Escritura <tipo>: Suponiendo que la cima de la pila sea de tipo “tipo” y justo debajo se encuentre una referencia a un printer, hace la llamada correspondiente a una función de la clase implementada por nosotros que como resultado imprime el dato por la salida estandar.

Como se ve, todas estas instrucciones hacen uso de funciones auxiliares definidas en la clase que se compila. Estas funciones estan todas ellas implementadas en JVM. Hay varias ventajas al hacerlo así en lugar de transformar cada instrucción en el código de la función.

Tal como está implementado el compilador, si no se usa una función auxiliar la clase resultante no la tendrá definida, por lo que no ocupará espacio en memoria.

Si no se usasen funciones la repetición de ciertas pseudoinstrucciones aumentarían considerablemente el tamaño del código.

Además dada una pseudoinstrucción el tamaño del codigo que generase no dependería solo del tipo de dato que trate, sino que si quisieramos conservar el punto 1 dependería de si esta pseudoinstrucción ha sido llamada con anterioridad o no.

Ciertas operaciones (como la O y la Y lógicas) requieren hacer saltos hacia adelante saltando código. Esto obliga a conocer exáctamente el número de bytes que ocuperá esa sección de código, lo cual es realmente complicado en dado el punto 3.

Puede obtenerse información más detallada de cada instrucción, ordenada alfabéticamente según su código mnemotécnico, en la página de la especificación oficial.

5.2. Funciones semánticas

5.3. Atributos semánticos

5.3.1 Atributos semánticos de codificación: codP y codJ

La categoria Programa, Instrucciones, Instrucción, InsLectura, InsEscritura, InsAsignacion, Expresion, ExpresionNiv1, ExpresionNiv2, ExpresiónNiv3, ExpresiónNiv4 y Literal tendrán un atributo “codP” que almacenará la traducción resultante en lenguaje P y otro “codJ” que almacenará la traducción resultante en java bytecode.

El atributo codJ almacenará el código de instrucciones jvm (más las añadidas de las que se habló antes). En un proceso posterior las instrucciones añadidas se traducirán a instrucciones jvm y este código se englobará dentro de un método estático main que a su vez se englobará dentro de una clase. Lo primero consiste en cambiar cada desplazamiento relativo de las instrucciones de salto a una dirección absoluta según la posición que ocupen y lo segundo en cambiar la instrucción de lectura y la de escritura por sus correspondientes llamadas a otros objetos. Este “objeto” será compilado desde java y se compilará en la misma dirección que el .class objeto, conteniendo las funciones necesarias para trasformar el flujo de caracteres que se obtiene con System.in a la representación interna de datos de la jvm y hará algo similar para la salida.

5.3.2 El atributo numVars

Para conocer la dirección de memoria de que corresponde a una variable se necesita un atributo sintetizado “numVars” que se inicie a 0 y aumente en uno por cada variable. Cada vez que se reconozca una declaración este se almacenará en la tabla de símbolos y luego se aumentará en una unidad.

De esta manera el atributo “numVars” servirá, además, para saber el número minimo de memoria que ha de tener el interprete (en caso del codigo P) o el número máximo de variables locales que requiere el método main en el caso de la compilación a jvm.
Este atributo será propio de las categoria Declaraciones.

5.4. Gramática de atributos

Programa → Declaraciones & Instrucciones
Programa.codP = Instrucciones.codP
Programa.codJ = Instrucciones.codJ
Declaraciones → Declaración ; Declaraciones
$Declaraciones_0$.ts = inserta(Declaraciones1.tsh, Declaracion.id, <dir:Declaraciones1.numVars>)
$Declaraciones_0$.numVars = Declaraciones1.numVars +1
Declaraciones → Declaración
Declaraciones.ts = inserta(creaTs(), Declaracion.id, <dir:0>)
Declaraciones.numVars = 1
Instrucciones → Instrucción ; Instrucciones
$Instrucciones_0$.codP = Instrucción.codP || $Instrucciones_1$.codP
$Instrucciones_0$.codJ = Instrucción.codJ || $Instrucciones_1$.codJ
Instrucciones → Instrucción
Instrucciones.codP = Instrucción.codP
Instrucciones.codJ = Instrucción.codJ
Instrucción → InsAsignación
Instrucciones.codP = InsAsignación.codP
Instrucciones.codJ = InsAsignación.codJ

Instrucción → InsLectura

Instruccion.codP = InsLectura.codP

Instruccion.codJ = InsLectura.codJ

Instrucción → InsEscritura

Instrucción.codP = InsEscritura.codP

Instrucción.codJ = InsEscritura.codJ

InsLectura → in(id)

InsLectura.codP = in InsLectura.tsh[id.lex].dir

InsLectura.codJ = lectura dameNumTipo(InsLectura.tsh[id.lex].tipo) InsLectura.tsh[id.lex].dir

InsEscritura → out(Expresion)

InsEscritura.codP = expresion.codP || out

InsEscritura.codJ = apilarPrinter || Expresión.codJ || escritura dameNumTipo(Expresion.tipo)

InsAsignación → id := Expresión

InsAsignación.codP = Expresión.codP || desapila-dir InsAsignación.tsh[id.lex].dir

InsAsignacion.codJ = Expresion.codJ || guardarDato InstAsignacion.tsh[id.lex].tipo

Expresión → ExpresiónNiv1 OpNiv0 ExpresiónNiv1

Expresión.codP =

case (OpNiv0.op)

menor:

case ($ExpresiónNiv1_o$.tipo)

float:

si ($ExpresiónNiv1_1$.tipo = float)

$ExpresiónNiv1_o$.codP || $ExpresiónNiv1_1$.codP || menor

sino

$ExpresiónNiv1_o$.codP || $ExpresiónNiv1_1$.codP || CastFloat || menor

entero:

si ($ExpresiónNiv1_1$.tipo = float)

$ExpresiónNiv1_o$.codP || CastFloat || $ExpresiónNiv1_1$.codP || menor

sino si (ExpresiónNiv12.tipo = natural)

$ExpresiónNiv1_o$.codP || $ExpresiónNiv1_1$.codP || CastInt || menor

sino

$ExpresiónNiv1_o$.codP || $ExpresiónNiv1_1$.codP || menor

natural:

si ($ExpresiónNiv1_1$.tipo = float)

$ExpresiónNiv1_o$.codP || CastFloat || $ExpresiónNiv1_1$.codP || menor

sino si (ExpresiónNiv12.tipo = entero)

$ExpresiónNiv1_o$.codP || CastInt || $ExpresiónNiv1_1$.codP || menor

sino

$ExpresiónNiv1_o$.codP || $ExpresiónNiv1_1$.codP || menor

otro:

$ExpresiónNiv1_o$.codP || $ExpresiónNiv1_1$.codP || menor

mayor

case ($ExpresiónNiv1_o$.tipo)

float:

si ($ExpresiónNiv1_1$.tipo = float)

$ExpresiónNiv1_o$.codP || $ExpresiónNiv1_1$.codP || mayor

sino

$ExpresiónNiv1_o$.codP || $ExpresiónNiv1_1$.codP || CastFloat || mayor

entero:

si ($ExpresiónNiv1_1$.tipo = float)

$ExpresiónNiv1_o$.codP || CastFloat || $ExpresiónNiv1_1$.codP || mayor

sino si ($ExpresiónNiv1_1$.tipo = natural)

$ExpresiónNiv1_o$.codP || $ExpresiónNiv1_1$.codP || CastInt || mayor

sino

$ExpresiónNiv1_o$.codP || $ExpresiónNiv1_1$.codP || mayor

natural:

si ($ExpresiónNiv1_1$.tipo = float)

$ExpresiónNiv1_o$.codP || CastFloat || $ExpresiónNiv1_1$.codP || mayor

sino si ($ExpresiónNiv1_1$.tipo = entero)

$ExpresiónNiv1_o$.codP || CastInt || $ExpresiónNiv1_1$.codP || mayor

```
                sino
                    ExpresiónNiv1o.codP || ExpresiónNiv11.codP || mayor
            otro:
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP ||  mayor
menor-ig
    case (ExpresiónNiv1o.tipo)
        float:
            si (ExpresiónNiv11.tipo = float)
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || menorIg
            sino
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || CastFloat || menorIg
        entero:
            si (ExpresiónNiv11.tipo = float)
                ExpresiónNiv1o.codP || CastFloat || ExpresiónNiv11.codP || menorIg
            sino si (ExpresiónNiv12.tipo = natural)
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || CastInt || menorIg
            sino
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || menorIg
        natural:
            si (ExpresiónNiv11.tipo = float)
                ExpresiónNiv1o.codP || CastFloat || ExpresiónNiv11.codP || menorIg
            sino si (ExpresiónNiv12.tipo = entero)
                ExpresiónNiv1o.codP || CastInt || ExpresiónNiv11.codP || menorIg
            sino
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || menorIg
        otro:
            ExpresiónNiv1o.codP || ExpresiónNiv11.codP ||  menorIg
mayor-ig
    case (ExpresiónNiv1o.tipo)
        float:
            si (ExpresiónNiv11.tipo = float)
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || mayorIg
            sino
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || CastFloat || mayorIg
        entero:
            si (ExpresiónNiv11.tipo = float)
                ExpresiónNiv1o.codP || CastFloat || ExpresiónNiv11.codP || mayorIg
            sino si (ExpresiónNiv12.tipo = natural)
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || CastInt || mayorIg
            sino
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || mayorIg
        natural:
            si (ExpresiónNiv11.tipo = float)
                ExpresiónNiv1o.codP || CastFloat || ExpresiónNiv11.codP || mayorIg
            sino si (ExpresiónNiv12.tipo = entero)
                ExpresiónNiv1o.codP || CastInt || ExpresiónNiv11.codP || mayorIg
            sino
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || mayorIg
        otro:
            ExpresiónNiv1o.codP || ExpresiónNiv11.codP ||  mayorIg
igual
    case (ExpresiónNiv1o.tipo)
        float:
            si (ExpresiónNiv11.tipo = float)
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || igual
            sino
                ExpresiónNiv1o.codP || ExpresiónNiv11.codP || CastFloat || igual
        entero:
            si (ExpresiónNiv11.tipo = float)
                ExpresiónNiv1o.codP || CastFloat || ExpresiónNiv11.codP || igual
```

```
                    sino si (ExpresiónNiv1₁.tipo = natural)
                        ExpresiónNiv1ₒ.codP || ExpresiónNiv1₁.codP || CastInt || igual
                    sino
                        ExpresiónNiv1ₒ.codP || ExpresiónNiv1₁.codP || igual
                natural:
                    si (ExpresiónNiv1₁.tipo = float)
                        ExpresiónNiv1ₒ.codP || CastFloat || ExpresiónNiv1₁.codP || igual
                    sino si (ExpresiónNiv1₁.tipo = entero)
                        ExpresiónNiv1ₒ.codP || CastInt || ExpresiónNiv1₁.codP || igual
                    sino
                        ExpresiónNiv1ₒ.codP || ExpresiónNiv1₁.codP || igual
                otro:
                    ExpresiónNiv1ₒ.codP || ExpresiónNiv1₁.codP ||  igual
        no-igual
            case (ExpresiónNiv1ₒ.tipo)
                float:
                    si (ExpresiónNiv1₁.tipo = float)
                        ExpresiónNiv1ₒ.codP || ExpresiónNiv1₁.codP || no-igual
                    sino
                        ExpresiónNiv1ₒ.codP || ExpresiónNiv1₁.codP || CastFloat || no-igual
                entero:
                    si (ExpresiónNiv1₁.tipo = float)
                        ExpresiónNiv1ₒ.codP || CastFloat || ExpresiónNiv1₁..codP || no-igual
                    sino si (ExpresiónNiv12.tipo = natural)
                        ExpresiónNiv1ₒ.codP || ExpresiónNiv1₁.codP || CastInt || no-igual
                    sino
                        ExpresiónNiv1ₒ.codP || ExpresiónNiv1₁.codP || no-igual
                natural:
                    si (ExpresiónNiv1₁.tipo = float)
                        ExpresiónNiv1ₒ.codP || CastFloat || ExpresiónNiv1₁.codP || no-igual
                    sino si (ExpresiónNiv12.tipo = entero)
                        ExpresiónNiv1ₒ.codP || CastInt || ExpresiónNiv1₁.codP || no-igual
                    sino
                        ExpresiónNiv1ₒ.codP || ExpresiónNiv1₁.codP || no-igual
                otro:
                    ExpresiónNiv1ₒ.codP || ExpresiónNiv1₁.codP ||  no-igual

Expresión.codJ =
    case (OpNiv0.op)
        menor:
            si (ExpresiónNiv1ₒ.tipo = float)
                si (ExpresiónNiv1₁.tipo = float)
                    ExpresiónNiv1ₒ.codJ ||
                    ExpresiónNiv1₁.codJ ||
                    fcmpg ||
                    if_ge +7
                    iconst_1
                    goto +4
                    iconst_0
                sino
                    ExpresiónNiv1ₒ.codJ ||
                    ExpresiónNiv1₁.codJ ||
                    i2f ||
                    fcmpg ||
                    if_ge +7
                    iconst_1
                    goto +4
                    iconst_0
            sino
                si(ExpresiónNiv1₁.tipo = float)
```

ExpresiónNiv1$_0$.codJ ||
i2f ||
ExpresiónNiv1$_1$.codJ ||
fcmpg ||
if_ge +7
iconst_1
goto +4
iconst_0
sino
ExpresiónNiv1$_0$.codJ ||
ExpresiónNiv1$_1$.codJ ||
if_icmpge +7
iconst_1
goto +4
iconst_0
mayor
si (ExpresiónNiv1$_0$.tipo = float)
si (ExpresiónNiv1$_1$.tipo = float)
ExpresiónNiv1$_0$.codJ ||
ExpresiónNiv1$_1$.codJ ||
fcmpg ||
if_le +7
iconst_1
goto +4
iconst_0
sino
ExpresiónNiv1$_0$.codJ ||
ExpresiónNiv1$_1$.codJ ||
i2f ||
fcmpg ||
if_le +7
iconst_1
goto +4
iconst_0
sino
si(ExpresiónNiv1$_1$.tipo = float)
ExpresiónNiv1$_0$.codJ ||
i2f ||
ExpresiónNiv1$_1$.codJ ||
fcmpg ||
if_le +7
iconst_1
goto +4
iconst_0
sino
ExpresiónNiv1$_0$.codJ ||
ExpresiónNiv1$_1$.codJ ||
if_icmple +7
iconst_1
goto +4
iconst_0
menor-ig
si (ExpresiónNiv1$_0$.tipo = float)
si (ExpresiónNiv1$_1$.tipo = float)
ExpresiónNiv1$_0$.codJ ||
ExpresiónNiv1$_1$.codJ ||
fcmpg ||
if_gt +7
iconst_1
goto +4

```
                iconst_0
            sino
                ExpresiónNiv1o.codJ ||
                ExpresiónNiv11.codJ ||
                i2f ||
                fcmpg ||
                if_gt +7
                iconst_1
                goto +4
                iconst_0
        sino
            si(ExpresiónNiv11.tipo = float)
                ExpresiónNiv1o.codJ ||
                i2f ||
                ExpresiónNiv11.codJ ||
                fcmpg ||
                if_gt +7
                iconst_1
                goto +4
                iconst_0
            sino
                ExpresiónNiv1o.codJ ||
                ExpresiónNiv11.codJ ||
                if_icmpgt +7
                iconst_1
                goto +4
                iconst_0
mayor-ig
    si (ExpresiónNiv1o.tipo = float)
        si (ExpresiónNiv11.tipo = float)
            ExpresiónNiv1o.codJ ||
            ExpresiónNiv11.codJ ||
            fcmpg ||
            if_lt +7
            iconst_1
            goto +4
            iconst_0
        sino
            ExpresiónNiv1o.codJ ||
            ExpresiónNiv11.codJ ||
            i2f ||
            fcmpg ||
            if_lt +7
            iconst_1
            goto +4
            iconst_0
    sino
        si(ExpresiónNiv11.tipo = float)
            ExpresiónNiv1o.codJ ||
            i2f ||
            ExpresiónNiv11.codJ ||
            fcmpg ||
            if_lt +7
            iconst_1
            goto +4
            iconst_0
        sino
            ExpresiónNiv1o.codJ ||
            ExpresiónNiv11.codJ ||
            if_icmplt +7
```

iconst_1
goto +4
iconst_0

igual

si (ExpresiónNiv1$_0$.tipo = float)

si (ExpresiónNiv1$_1$.tipo = float)

ExpresiónNiv1$_0$.codJ ||
ExpresiónNiv1$_1$.codJ ||
fcmpg ||
if_ne +7
iconst_1
goto +4
iconst_0

sino

ExpresiónNiv1$_0$.codJ ||
ExpresiónNiv1$_1$.codJ ||
i2f ||
fcmpg ||
if_ne +7
iconst_1
goto +4
iconst_0

sino

si(ExpresiónNiv1$_1$.tipo = float)

ExpresiónNiv1$_0$.codJ ||
i2f ||
ExpresiónNiv1$_1$.codJ ||
fcmpg ||
if_ne +7 ||
iconst_1 ||
goto +4 ||
iconst_0

sino

ExpresiónNiv1$_0$.codJ ||
ExpresiónNiv1$_1$.codJ ||
if_icmpne +7 ||
iconst_1 ||
goto +4 ||
iconst_0

no-igual

si (ExpresiónNiv1$_0$.tipo = float)

si (ExpresiónNiv1$_1$.tipo = float)

ExpresiónNiv1$_0$.codJ ||
ExpresiónNiv1$_1$.codJ ||
fcmpg ||
if_eq +7 ||
iconst_1 ||
goto +4 ||
iconst_0

sino

ExpresiónNiv1$_0$.codJ ||
ExpresiónNiv1$_1$.codJ ||
i2f ||
fcmpg ||
if_eq +7 ||
iconst_1 ||
goto +4 ||
iconst_0

sino

si(ExpresiónNiv1$_1$.tipo = float)

ExpresiónNiv1$_o$.codJ ||
i2f ||
ExpresiónNiv1$_1$.codJ ||
fcmpg ||
if_eq +7 ||
iconst_1 ||
goto +4 ||
iconst_0
sino
ExpresiónNiv1$_o$.codJ ||
ExpresiónNiv1$_1$.codJ ||
if_icmpeq +7 ||
iconst_1 ||
goto +4 ||
iconst_0

Expresión → ExpresiónNiv1

Expresión.codP = ExpresionNiv1.codP
Expresión.codJ = ExpresionNiv1.codJ

ExpresiónNiv1 → ExpresiónNiv1 OpNiv1 ExpresiónNiv2

ExpresionNiv1$_o$.codP =
case (OpNiv1.op)
suma:
case (ExpresionNiv1$_1$.tipo)
float:
si(ExpresionNiv2 .tipo = float)
ExpresionNiv1$_1$.codP || ExpresionNiv2.codP || sumar
sino
ExpresionNiv1$_1$.codP || ExpresionNiv2.codP || CastFloat || sumar
entero:
si (ExpresionNiv2.tipo = float)
ExpresionNiv1$_1$.codP || CastFloat || ExpresionNiv2.codP || sumar
sino si (ExpresionNiv2.tipo = natural)
ExpresionNiv1$_1$.codP || ExpresionNiv2.codP || CastInt || sumar
sino
ExpresionNiv1$_1$.codP || ExpresionNiv2.codP || sumar
natural:
si (ExpresionNiv2.tipo = float)
ExpresionNiv1$_1$.codP || CastFloat || ExpresionNiv2.codP || sumar
sino si (ExpresionNiv2.tipo = entero)
ExpresionNiv1$_1$.codP || CastInt || ExpresionNiv2.codP || sumar
sino
ExpresionNiv1$_1$.codP || ExpresionNiv2.codP || sumar
resta:
case (ExpresionNiv1$_1$.tipo)
float:
si(ExpresionNiv2 .tipo = float)
ExpresionNiv1$_1$.codP || ExpresionNiv2.codP || restar
sino
ExpresionNiv1$_1$.codP || ExpresionNiv2.codP || CastFloat || restar
entero:
si (ExpresionNiv2.tipo = float)
ExpresionNiv1$_1$.codP || CastFloat || ExpresionNiv2.codP || restar
sino si (ExpresionNiv2.tipo = natural)
ExpresionNiv1$_1$.codP || ExpresionNiv2.codP || CastInt || restar
sino
ExpresionNiv1$_1$.codP || ExpresionNiv2.codP || restar
natural:
si (ExpresionNiv2.tipo = float)
ExpresionNiv1$_1$.codP || CastFloat || ExpresionNiv2.codP || restar
sino si (ExpresionNiv2.tipo = entero)

ExpresionNiv1$_1$.codP || CastInt || ExpresionNiv2.codP || restar
sino
ExpresionNiv1$_1$.codP || ExpresionNiv2.codP || restar
o:
ExpresiónNiv1$_1$.codP || ExpresiónNiv2.codP || o
ExpresionNiv1$_o$.codJ =
case (OpNiv1.op)
suma:
si (ExpresionNiv1$_1$.tipo = float)
si(ExpresionNiv2.tipo = float)
ExpresiónNiv1$_1$.codJ || ExpresiónNiv2. codJ || fadd
sino
ExpresiónNiv1$_1$.codJ || ExpresiónNiv2. codJ || i2f || fadd
sino
si(ExpresionNiv2.tipo = float)
ExpresiónNiv1$_1$.codJ || i2f || ExpresiónNiv2. codJ || fadd
sino
ExpresiónNiv11.codJ || ExpresiónNiv2. codJ || iadd
resta:
si (ExpresionNiv1$_1$.tipo = float)
si(ExpresionNiv2.tipo = float)
ExpresiónNiv1$_1$.codJ || ExpresiónNiv2. codJ || fsub
sino
ExpresiónNiv1$_1$.codJ || ExpresiónNiv2. codJ || i2f || fsub
sino
si(ExpresionNiv2.tipo = float)
ExpresiónNiv1$_1$.codJ || i2f || ExpresiónNiv2. codJ || fsub
sino
ExpresiónNiv1$_1$.codJ || ExpresiónNiv2. codJ || isub
o:
ExpresiónNiv1$_1$.codJ ||
ExpresionNiv2.codJ ||
O

ExpresiónNiv1 → ExpresiónNiv2

ExpresionNiv1.codP = ExpresionNiv2.codP
ExpresionNiv1.codJ = ExpresionNiv2.codJ

ExpresiónNiv2 → ExpresiónNiv2 OpNiv2 ExpresiónNiv3

ExpresiónNiv2$_o$.codP =
case(OpNiv2.op)
Multiplica:
case (ExpresionNiv2$_1$.tipo)
float:
si(ExpresionNiv3.tipo = float)
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || Mul
sino
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || CastFloat || Mul
entero:
si (ExpresionNiv3 .tipo = float)
ExpresiónNiv2$_1$.codP || CastFloat || ExpresiónNiv3.codP || Mul
sino si (ExpresionNiv3 .tipo = natural)
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || CastInt || Mul
sino
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || Mul
natural:
si (ExpresionNiv3 .tipo = float)
ExpresiónNiv2$_1$.codP || CastFloat || ExpresiónNiv3.codP || Mul
sino si (ExpresionNiv3 .tipo = entero)
ExpresiónNiv2$_1$.codP || CastInt || ExpresiónNiv3.codP || Mul
sino
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || Mul

Divide:
case (ExpresionNiv2$_1$.tipo)
float:
si(ExpresionNiv3.tipo = float)
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || Div
sino
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || CastFloat || Div
entero:
si (ExpresionNiv3 .tipo = float)
ExpresiónNiv2$_1$.codP || CastFloat || ExpresiónNiv3.codP || Div
sino si (ExpresionNiv3 .tipo = natural)
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || CastInt || Div
sino
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || Div
natural:
si (ExpresionNiv3 .tipo = float)
ExpresiónNiv2$_1$.codP || CastFloat || ExpresiónNiv3.codP || Div
sino si (ExpresionNiv3 .tipo = entero)
ExpresiónNiv2$_1$.codP || CastInt || ExpresiónNiv3.codP || Div
sino
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || Div
Modulo:
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || Mod
y:
ExpresiónNiv2$_1$.codP || ExpresiónNiv3.codP || Y

ExpresiónNiv2$_o$.codJ =
case(OpNiv2.op)
Multiplica:
si(ExpresionNiv2$_1$.tipo = float)
si(ExpresionNiv3.tipo = float)
ExpresionNiv2$_1$.codJ || ExpresionNiv3.codJ || fmul
sino
ExpresionNiv2$_1$.codJ || ExpresionNiv3.codJ || i2f || fmul
sino
si(ExpresionNiv3.tipo = float)
ExpresionNiv2$_1$.codJ || i2f || ExpresionNiv3.codJ|| fmul
sino
ExpresionNiv2$_1$.codJ || ExpresionNiv3.codJ|| imul
Divide:
si(ExpresionNiv2$_1$.tipo = float)
si(ExpresionNiv3.tipo = float)
ExpresionNiv2$_1$.codJ || ExpresionNiv3.codJ || fdiv
sino
ExpresionNiv2$_1$.codJ || ExpresionNiv3.codJ || i2f || fdiv
sino
si(ExpresionNiv3.tipo = float)
ExpresionNiv2$_1$.codJ || i2f || ExpresionNiv3.codJ|| fdiv
sino
ExpresionNiv2$_1$.codJ || ExpresionNiv3.codJ|| idiv
Modulo:
ExpresiónNiv2$_1$. codJ || ExpresiónNiv3. codJ || imod
y:
ExpresionNiv2$_1$.codJ ||
ExpresiónNiv3.codJ ||
Y

ExpresiónNiv2 → ExpresiónNiv3

ExpresiónNiv2.codP = ExpresiónNiv3.codP
ExpresiónNiv2.codJ = ExpresiónNiv3.codJ

ExpresiónNiv3 → ExpresionNiv4 OpNiv3 ExpresiónNiv3

ExpresiónNiv3$_o$.codP =

case (OpNiv3.op)
shl:
ExpresiónNiv4.codP || ExpresiónNiv3$_1$.codP || shl
shr:
ExpresiónNiv4.codP || ExpresiónNiv3$_1$.codP || shr
ExpresiónNiv3$_o$.codJ =
case (OpNiv3.op)
shl:
ExpresiónNiv4.codJ || ExpresiónNiv3$_1$.codJ|| ishl
shr:
ExpresiónNiv4.codJ || ExpresiónNiv3$_1$.codJ || ishr

ExpresiónNiv3 → ExpresiónNiv4

ExpresiónNiv3.codP = ExpresiónNiv4.codP
ExpresionNiv3.codJ = ExpresionNiv4.codJ

ExpresiónNiv4 → OpNiv4 ExpresiónNiv4

ExpresiónNiv4$_o$.codP =
case (OpNiv4.op)
no:
ExpresiónNiv41.codP || no
negativo:
ExpresiónNiv41.codP || negativo
cast-float:
ExpresiónNiv41.codP || CastFloat
cast-int:
ExpresiónNiv41.codP || CastInt
cast-nat:
ExpresiónNiv41.codP || CastNat
cast-char:
ExpresiónNiv41.codP || CastChar
ExpresiónNiv4$_o$.codJ =
case (OpNiv4.op)
no:
ExpresiónNiv4.codJ ||
ifeq +7 ||
iconst_0 ||
goto +4 ||
iconst_1
negativo:
case (ExpresionNiv41.tipo)
entero:
ExpresiónNiv41.codJ || ineg
float:
ExpresiónNiv41.codJ || fneg
cast-float:
case (ExpresionNiv41.tipo)
character:
natural:
entero:
ExpresiónNiv41.codJ || i2f
float:
ExpresiónNiv41.codJ
cast-int:
case (ExpresionNiv41.tipo)
character:
natural:
entero:
ExpresiónNiv41.codJ
float:
ExpresiónNiv41.codJ || f2i
cast-nat:

case (ExpresionNiv41.tipo)
character:
natural:
entero:
ExpresiónNiv4.codJ
float:
ExpresiónNiv4.codJ || f2i
cast-char:
case (ExpresionNiv41.tipo)
character:
natural:
entero:
ExpresiónNiv41.codJ
float:
ExpresiónNiv41.codJ || f2i

ExpresiónNiv4 → | Expresión |
ExpresiónNiv4.codP = Expresión.codP || abs
ExpresionNiv4.codJ = Expresion.codJ || abs

ExpresiónNiv4 → (Expresión)
ExpresiónNiv4.codP = Expresión.codP
ExpresionNiv4.codJ = Literal.codJ

ExpresiónNiv4 → Literal
ExpresiónNiv4.codP = Literal.codP
ExpresionNiv4.codJ = Literal.codJ

Literal → id
Literal.codP = apila-dir Literal.tsh[id.lex].dir
Literal.codJ = cargarDato Literal.tsh[id.lex].tipo)

Literal → litNat
Literal.codP = apila LitNat.lex
Literal.codJ = apilarInt LitNat.lex

Literal → litFlo
Literal.codP = apila litFlo.lex
Literal.codJ = apilarFloat litFlo.lex

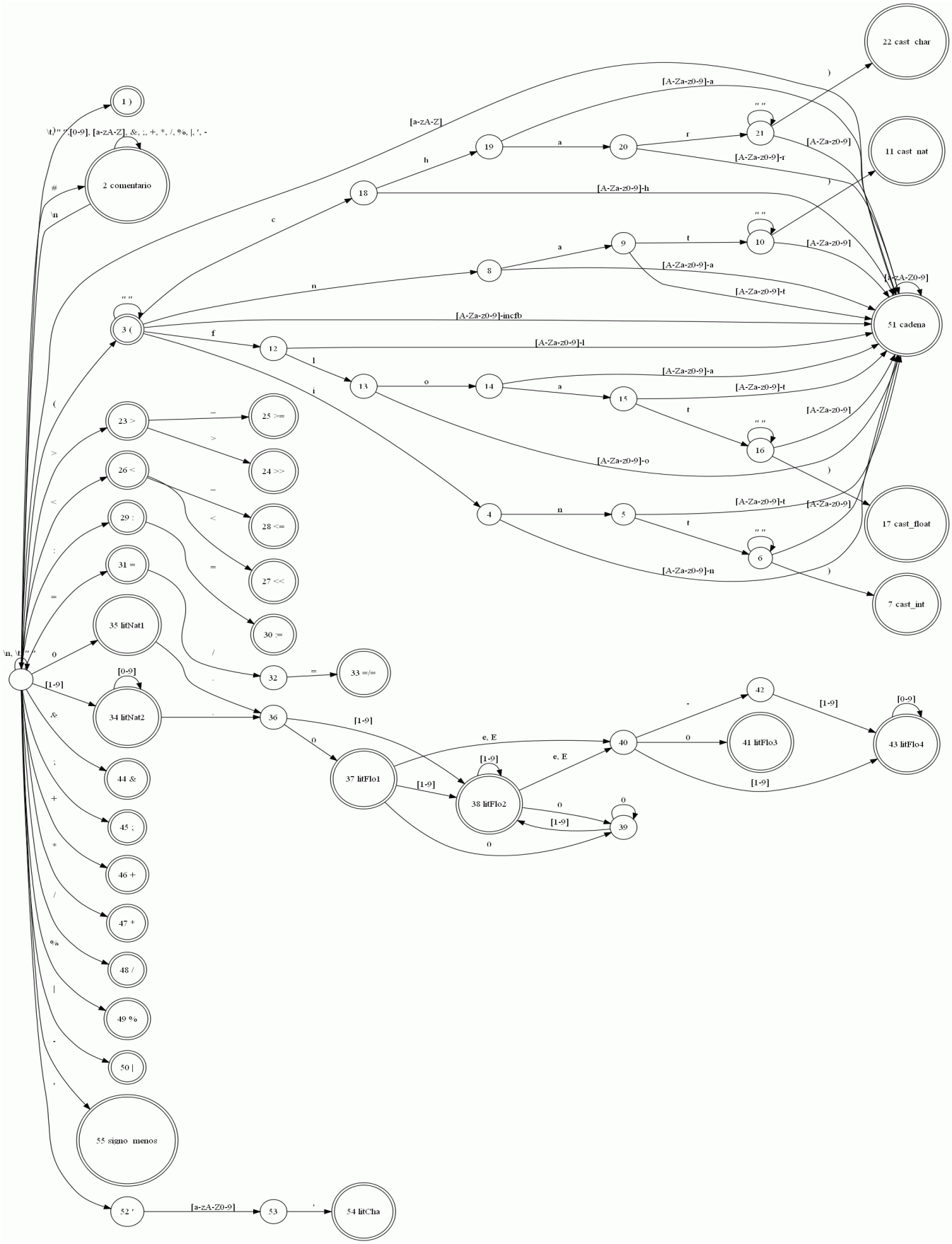
Literal → litTrue
Literal.codP = apila true
Literal.codJ = apilarInt 1

Literal → litFalse
Literal.codP = apila false
Literal.codJ = apilarInt 0

Literal → litCha
Literal.codP = apila litCha.lex
Literal.codJ = apilarInt (litCha.lex)

6. Diseño del analizador léxico

Diagrama de transición que caracterice el diseño del analizador léxico. La implementación del analizador léxico debe estar guiada por este diseño.



7. Acondicionamiento de las gramáticas de atributos

A continuación se presentan los acondicionamientos de las construcciones gramáticas de los puntos 2,3,4 que los necesitan, ya sea por ser recurrentes a izquierdas o por necesitar una factorización. Los acondicionamientos se han realizado ciñiéndonos a los esquemas proporcionados, incluso cuando no fuese estrictamente necesario, para facilitar una mejor compresión en el futuro y mantener una cierta coherencia entre la forma de proceder para las diferentes partes de las gramáticas.

7.1. Acondicionamiento de la Gramática para la Construcción de la tabla de símbolos

(factorizamos)
Declaraciones → Declaracion DeclaracionesFact
DeclaracionesFact.idh = Declaracion.id
DeclaracionesFact.tipoh = Declaracion.tipo
Declaraciones.ts = DeclaracionesFact.ts
DeclaracionesFact → λ
DeclaracionFact.ts = inserta(creaTS(),DeclaracionesFact.idh,<tipo:DeclaracionesFact.tipoh>)
DeclaracionesFact → ; Declaraciones
DeclaracionesFact.ts = inserta(Declaraciones.ts,DeclaracionesFact.idh,<tipo:DeclaracionesFact.tipoh>)

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

(factorizamos)
Declaraciones → Declaración DeclaracionesFact
DeclaracionesFact.errorh = Declaración.error
Declaraciones.error = DeclaracionesFact.error
DeclaracionesFact → λ
DeclaracionesFact.error = DeclaracionesFact.errorh
DeclaracionesFact → ; Declaraciones
DeclaracionesFact.error = DeclaracionesFact.errorh v
Declaraciones.error v existe(Declaraciones.ts,DeclaracionFact.idh)

(factorizamos)
Instrucciones → Instrucción InstruccionesFact
InstruccionesFact.errorh = Instrucción.error
Instrucción.tsh = Instrucciones.tsh
InstruccionesFact.tsh = Instrucciones.tsh
Instrucciones.error = InstruccionesFact.error
InstruccionesFact → λ
InstruccionesFact.error = InstruccionesFact.errorh
InstruccionesFact → ; Instrucciones
InstruccionesFact.error = Instrucciones.error v InstruccionesFact.errorh
Instrucciones.tsh = InstruccionesFact.tsh

(factorizamos)
Expresion → ExpresionNiv1 ExpresionFact
ExpresionFact.tipoh = ExpresionNiv1.tipo
Expresion.tipo = ExpresionFact.tipo
ExpresionNiv1.tsh = Expresion.tsh
ExpresionFact.tsh = Expresion.tsh
ExpresionFact → λ
ExpresionFact.tipo = ExpresionFact.tipoh
ExpresionFact → OpNiv0 ExpresionNiv1
ExpresionFact.tipo = si (ExpresionFact.tipoh = error v ExpresionNiv1.tipo = error) v
(ExpresionFact.tipoh = character ∧ ExpresionNiv1.tipo =/= character) v

(ExpresionFact.tipoh =/= character ∧ ExpresionNiv1.tipo = character)
(ExpresionFact.tipoh = boolean ∧ ExpresionNiv1.tipo =/= boolean) v
(ExpresionFact.tipoh =/= boolean ∧ ExpresionNiv1.tipo = boolean))
error
sino boolean
ExpresionNiv1.tsh = ExpresionFact.tsh

(eliminamos la recursión a izquierdas)
ExpresiónNiv1 → ExpresiónNiv2 ExpresionNiv1Rec
ExpresionNiv1Rec.tipoh = ExpresiónNiv2.tipo.
ExpresionNiv1.tipo = ExpresiónNiv2Rec.tipo
ExpresionNiv2.tsh = ExpresionNiv1.tsh
ExpresionNiv1Rec.tsh = ExpresionNiv1.tsh
ExpresiónNiv1Rec → OpNiv1 ExpresiónNiv2 ExpresiónNiv1Rec
$ExpresionNiv1Rec_1$.tipoh =
si ($ExpresionNiv1Rec_0$.tipoh = error v ExpresionNiv2.tipo = error v
$ExpresionNiv1Rec_0$.tipoh = char v ExpresionNiv2.tipo = char v
($ExpresionNiv1Rec_0$.tipoh = boolean ∧ExpresionNiv2.tipo =/= boolean) v
($ExpresionNiv1Rec_0$.tipoh =/= boolean ∧ExpresionNiv2.tipo = boolean))
error
sino case (OpNiv1.op)
suma,resta:
si ($ExpresionNiv1Rec_0$.tipoh=float v ExpresionNiv2.tipo = float)
float
sino si ($ExpresionNiv1Rec_0$.tipoh =integer v ExpresionNiv2.tipo = integer)
integer
sino si ($ExpresionNiv1Rec_0$.tipoh =natural ∧ExpresionNiv2.tipo = natural)
natural
sino error
o:
si ($ExpresionNiv1Rec_0$.tipoh = boolean ∧ExpresionNiv2.tipo = boolean)
boolean
sino error
$ExpresionNivlRec_0$.tipo = $ExpresionNiv1Rec_1$.tipo
ExpresionNiv2.tsh = $ExpresionNiv1Rec_0$.tsh
$ExpresionNiv1Rec_1$.tsh = $ExpresionNiv1Rec_0$.tsh
ExpresionNiv1Rec → λ
ExpresionNiv1Rec.tipo = ExpresionNiv1Rec.tipoh

(eliminamos la recursión a izquierdas)
ExpresiónNiv2 → ExpresionNiv3 ExpresiónNiv2Rec
ExpresionNiv2Rec.tipoh = ExpresiónNiv3.tipo
ExpresionNiv2.tipo = ExpresiónNiv2Rec.tipo
ExpresionNiv3.tsh = ExpresionNiv2.tsh
ExpresionNiv2Rec.tsh = ExpresionNiv2.tsh
ExpresiónNiv2Rec → OpNiv2 ExpresiónNiv3 ExpresiónNiv2Rec
$ExpresionNiv2Rec_1$.tipoh =
si ($ExpresionNiv2Rec_0$.tipoh = error v ExpresionNiv3.tipo = error v
$ExpresionNiv2Rec_0$. = character v ExpresionNiv3.tipo = character v
($ExpresionNiv2Rec_0$.tipoh = boolean ∧ExpresionNiv3.tipo =/= boolean v
($ExpresionNiv2Rec_0$.tipoh =/= boolean ∧ExpresionNiv3.tipo = boolean))
error
sino case (OpNiv2.op)

multiplica,divide:
si (ExpresionNiv2Rec$_0$.tipoh=float v ExpresionNiv3.tipo = float)
float
sino si (ExpresionNiv2Rec$_0$.tipoh =integer v ExpresionNiv3.tipo = integer)
integer
sino si (ExpresionNiv2Rec$_0$.tipoh =natural ∧ExpresionNiv3.tipo=natural)
natural
sino error
modulo:
si (ExpresionNiv3.tipo = natural ∧
(ExpresionNiv2Rec$_0$.tipoh=natural v ExpresionNiv2Rec0.tipoh=integer))
ExpresionNiv2Rec$_0$.tipoh
sino error
y:
si (ExpresionNiv2Rec$_0$.tipoh = boolean ∧ExpresionNiv3.tipo = boolean)
boolean
sino error

ExpresionNiv2Rec$_0$.tipo = ExpresionNiv2Rec$_1$.tipo

ExpresionNiv2Rec$_1$.tsh = ExpresionNiv2Rec$_0$.tsh

ExpresionNiv3.tsh = ExpresionNiv2Rec$_0$.tsh

ExpresiónNiv2Rec → λ

ExpresionNiv2Rec.tipo = ExpresionNiv2Rec.tipoh

(factorizamos)

ExpresionNiv3 → ExpresionNiv4 ExpresionNiv3Fact

ExpresiónNiv4.tsh = ExpresiónNiv3.tsh
ExpresiónNiv3Fact.tsh = ExpresiónNiv3.tsh
ExpresionNiv3Fact.tipoh = ExpresiónNiv4.tipo
ExpresionNiv3.tipo = ExpresionNiv3Fact.tipo

ExpresionNiv3Fact → λ

ExpresionNiv3Fact.tipo = ExpresionNiv3Fact.tipoh

ExpresionNiv3Fact → OpNiv3 ExpresiónNiv3

ExpresiónNiv3.tsh = ExpresiónNiv3Fact.tsh
ExpresionNiv3Fact.tipo =
si (ExpresionNiv3Fact.tipoh = error v ExpresionNiv3.tipo = error v
ExpresionNiv3Fact.tipoh =/= natural v ExpresionNiv3.tipo =/= natural)
error
sino natural

7.3. Acondicionamiento de la Gramática para la Traducción

(factorizamos)

Declaraciones → Declaracion DeclaracionesFact

DeclaracionFact.idh = Declaracion.id
Declaraciones.ts = DeclaracionFact.ts
Declaraciones.numVars = DeclaracionesFact.numVars

DeclaracionesFact → λ

DeclaracionesFact.ts = inserta(creaTS(),DeclaracionesFact.idh,<dir:0>)
DeclaracionesFact.numVars = 1

DeclaracionesFact → ; Declaraciones

DeclaracionesFact.ts = inserta(Declaraciones.ts,DeclaracionesFact.idh,<dir:Declaraciones.numVars>)
DeclaracionesFact.numVars = Declaraciones.numVars + 1

(factorizamos)
Instrucciones → Instrucción InstruccionesFact
InstruccionesFact.codPh = Instrucción.codP
InstruccionesFact.codJh = Instrucción.codJ
Instrucciones.codP = InstruccionesFact.codP
Instrucciones.codJ = InstruccionesFact.codJ
InstruccionesFact → λ
InstruccionesFact.codP = InstruccionesFact.codPh
InstruccionesFact.codJ = InstruccionesFact.codJh
InstruccionesFact → ; Instrucciones
InstruccionesFact.codP = InstruccionesFact.codPh || Instrucciones.codP
InstruccionesFact.codJ = InstruccionesFact.codJh || Instrucciones.codJ

(factorizamos)
Expresión → ExpresiónNiv1 ExpresiónFact
ExpresiónFact.codPh = ExpresionNiv1.codP
ExpresiónFact.codJh = ExpresiónNiv1.codJ
Expresión.codP = ExpresiónFact.codP
Expresión.codJ = ExpresiónFact.codJ
ExpresiónFact → λ
ExpresiónFact.codP = ExpresiónFact.codPh
ExpresiónFact.codJ = ExpresiónFact.codJh
ExpresiónFact → OpNiv0 ExpresiónNiv1
ExpresiónFact.codP =
case (OpNiv0.op)
menor:
case (ExpresiónFact.tipoh)
float:
si (ExpresiónNiv1.tipo = float)
ExpresiónFact.codPh || ExpresiónNiv1.codP || menor
sino
ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || menor
entero:
si (ExpresiónNiv1.tipo = float)
ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || menor
sino si (ExpresiónNiv1.tipo = natural)
ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || menor
sino
ExpresiónFact.codPh || ExpresiónNiv1.codP || menor
natural:
si (ExpresiónNiv1.tipo = float)
ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || menor
sino si (ExpresiónNiv1.tipo = entero)
ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || menor
sino
ExpresiónFact.codPh || ExpresiónNiv1.codP || menor
otro:
ExpresiónFact.codPh || ExpresiónNiv1.codP || menor
mayor
case (ExpresiónFact.tipoh)
float:
si (ExpresiónNiv1.tipo = float)
ExpresiónFact.codPh || ExpresiónNiv1.codP || mayor
sino
ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || mayor
entero:
si (ExpresiónNiv1.tipo = float)
ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || mayor
sino si (ExpresiónNiv1.tipo = natural)

```
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || mayor
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || mayor
            natural:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || mayor
                sino si (ExpresiónNiv1.tipo = entero)
                    ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || mayor
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || mayor
            otro:
                ExpresiónFact.codPh || ExpresiónNiv1.codP ||  mayor
menor-ig
    case (ExpresiónFact.tipoh)
            float:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || menorIg
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || menorIg
            entero:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || menorIg
                sino si (ExpresiónNiv1.tipo = natural)
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || menorIg
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || menorIg
            natural:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || menorIg
                sino si (ExpresiónNiv1.tipo = entero)
                    ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || menorIg
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || menorIg
            otro:
                ExpresiónFact.codPh || ExpresiónNiv1.codP ||  menorIg
mayor-ig
    case (ExpresiónFact.tipoh)
            float:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || mayorIg
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || mayorIg
            entero:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || mayorIg
                sino si (ExpresiónNiv1.tipo = natural)
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || mayorIg
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || mayorIg
            natural:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact0.codPh || CastFloat || ExpresiónNiv1.codP || mayorIg
                sino si (ExpresiónNiv1.tipo = entero)
                    ExpresiónFact0.codPh || CastInt || ExpresiónNiv1.codP || mayorIg
                sino
                    ExpresiónFact0.codPh || ExpresiónNiv1.codP || mayorIg
            otro:
                ExpresiónFact0.codPh || ExpresiónNiv1.codP ||  mayorIg
igual
    case (ExpresiónFact.tipoh)
```

```
                    float:
                        si (ExpresiónNiv1.tipo = float)
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || igual
                        sino
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || igual
                    entero:
                        si (ExpresiónNiv1.tipo = float)
                            ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || igual
                        sino si (ExpresiónNiv1.tipo = natural)
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || igual
                        sino
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || igual
                    natural:
                        si (ExpresiónNiv1.tipo = float)
                            ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || igual
                        sino si (ExpresiónNiv1.tipo = entero)
                            ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || igual
                        sino
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || igual
                    otro:
                        ExpresiónFact.codPh || ExpresiónNiv1.codP ||  igual
            no-igual
                case (ExpresiónFact.tipoh)
                    float:
                        si (ExpresiónNiv1.tipo = float)
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || no-igual
                        sino
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || no-igual
                    entero:
                        si (ExpresiónNiv1.tipo = float)
                            ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || no-igual
                        sino si (ExpresiónNiv1.tipo = natural)
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || no-igual
                        sino
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || no-igual
                    natural:
                        si (ExpresiónNiv1.tipo = float)
                            ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || no-igual
                        sino si (ExpresiónNiv1.tipo = entero)
                            ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || no-igual
                        sino
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || no-igual
                    otro:
                        ExpresiónFact0.codPh || ExpresiónNiv1.codP ||  no-igual

ExpresiónFact.codJ =
    case (OpNiv0.op)
        menor:
            si (ExpresiónFact.tipoh = float)
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    fcmpg ||
                    if_ge +7
                    iconst_1
                    goto +4
                    iconst_0
                sino
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
```

```
                    i2f ||
                    fcmpg ||
                    if_ge +7
                    iconst_1
                    goto +4
                    iconst_0
        sino
            si(ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codJh ||
                    i2f ||
                    ExpresiónNiv1.codJ ||
                    fcmpg ||
                    if_ge +7
                    iconst_1
                    goto +4
                    iconst_0
            sino
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    if_icmpge +7
                    iconst_1
                    goto +4
                    iconst_0
mayor
        si (ExpresiónFact.tipoh = float)
            si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    fcmpg ||
                    if_le +7
                    iconst_1
                    goto +4
                    iconst_0
            sino
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    i2f ||
                    fcmpg ||
                    if_le +7
                    iconst_1
                    goto +4
                    iconst_0
        sino
            si(ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codJh ||
                    i2f ||
                    ExpresiónNiv1.codJ ||
                    fcmpg ||
                    if_le +7
                    iconst_1
                    goto +4
                    iconst_0
            sino
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    if_icmple +7
                    iconst_1
                    goto +4
                    iconst_0
menor-ig
```

```
        si (ExpresiónFact.tipoh = float)
            si (ExpresiónNiv1.tipo = float)
                ExpresiónFact.codJh ||
                ExpresiónNiv1.codJ ||
                fcmpg ||
                if_gt +7
                iconst_1
                goto +4
                iconst_0
            sino
                ExpresiónFact.codJh ||
                ExpresiónNiv1.codJ ||
                i2f ||
                fcmpg ||
                if_gt +7
                iconst_1
                goto +4
                iconst_0
        sino
            si(ExpresiónNiv1.tipo = float)
                ExpresiónFact.codJh ||
                i2f ||
                ExpresiónNiv1.codJ ||
                fcmpg ||
                if_gt +7
                iconst_1
                goto +4
                iconst_0
            sino
                ExpresiónFact.codJh ||
                ExpresiónNiv1.codJ ||
                if_icmpgt +7
                iconst_1
                goto +4
                iconst_0
mayor-ig
        si (ExpresiónFact.tipoh = float)
            si (ExpresiónNiv1.tipo = float)
                ExpresiónFact.codJh ||
                ExpresiónNiv1.codJ ||
                fcmpg ||
                if_lt +7
                iconst_1
                goto +4
                iconst_0
            sino
                ExpresiónFact.codJh ||
                ExpresiónNiv1.codJ ||
                i2f ||
                fcmpg ||
                if_lt +7
                iconst_1
                goto +4
                iconst_0
        sino
            si(ExpresiónNiv1.tipo = float)
                ExpresiónFact.codJh ||
                i2f ||
                ExpresiónNiv1.codJ ||
                fcmpg ||
```

```
                    if_lt +7
                    iconst_1
                    goto +4
                    iconst_0
                sino
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    if_icmplt +7
                    iconst_1
                    goto +4
                    iconst_0
igual
        si (ExpresiónFact.tipoh = float)
            si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    fcmpg ||
                    if_ne +7
                    iconst_1
                    goto +4
                    iconst_0
            sino
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    i2f ||
                    fcmpg ||
                    if_ne +7
                    iconst_1
                    goto +4
                    iconst_0
        sino
            si(ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codJh ||
                    i2f ||
                    ExpresiónNiv1.codJ ||
                    fcmpg ||
                    if_ne +7 ||
                    iconst_1 ||
                    goto +4 ||
                    iconst_0
            sino
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    if_icmpne +7 ||
                    iconst_1 ||
                    goto +4 ||
                    iconst_0
no-igual
        si (ExpresiónFact.tipoh = float)
            si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    fcmpg ||
                    if_eq +7 ||
                    iconst_1 ||
                    goto +4 ||
                    iconst_0
            sino
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
```

i2f ||
fcmpg ||
if_eq +7 ||
iconst_1 ||
goto +4 ||
iconst_0

sino

si(ExpresiónNiv1.tipo = float)

ExpresiónFact.codJh ||
i2f ||
ExpresiónNiv1.codJ ||
fcmpg ||
if_eq +7 ||
iconst_1 ||
goto +4 ||
iconst_0

sino

ExpresiónFact.codJh ||
ExpresiónNiv1.codJ ||
if_icmpeq +7 ||
iconst_1 ||
goto +4 ||
iconst_0

(eliminamos la recursión a izquierdas)

ExpresiónNiv1 → ExpresiónNiv2 ExpresiónNiv1Rec

ExpresiónNiv1Rec.codPh = ExpresiónNiv2.codP
ExpresiónNiv1Rec.codJh = ExpresiónNiv2.codJ
ExpresiónNiv1.codP = ExpresiónNiv1Rec.codP
ExpresiónNiv1.codJ = ExpresiónNiv1Rec.codJ

ExpresiónNiv1Rec → λ

ExpresiónNiv1Rec.codP = ExpresiónNiv1Rec.codPh
ExpresiónNiv1Rec.codJ = ExpresiónNiv1Rec.codJh

ExpresiónNiv1Rec → OpNiv1 ExpresiónNiv2 ExpresiónNiv1Rec

$ExpresiónNiv1Rec_1$.codPh =

case (OpNiv1.op)
suma:
case ($ExpresiónNiv1Rec_0$.tipoh)
float:
si(ExpresionNiv2 .tipo = float)
$ExpresiónNiv1Rec_0$.codPh || ExpresionNiv2.codP || sumar
sino
$ExpresiónNiv1Rec_0$.codPh || ExpresionNiv2.codP || CastFloat || sumar
entero:
si (ExpresionNiv2.tipo = float)
$ExpresiónNiv1Rec_0$.codPh || CastFloat || ExpresionNiv2.codP || sumar
sino si (ExpresionNiv2.tipo = natural)
$ExpresiónNiv1Rec_0$.codPh || ExpresionNiv2.codP || CastInt || sumar
sino
$ExpresiónNiv1Rec_0$.codPh || ExpresionNiv2.codP || sumar
natural:
si (ExpresionNiv2.tipo = float)
$ExpresiónNiv1Rec_0$.codPh || CastFloat || ExpresionNiv2.codP || sumar
sino si (ExpresionNiv2.tipo = entero)
$ExpresiónNiv1Rec_0$.codPh || CastInt || ExpresionNiv2.codP || sumar
sino

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || sumar

resta:

case (ExpresiónNiv1Rec$_0$.tipoh)

float:

si(ExpresionNiv2 .tipo = float)

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || restar

sino

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || CastFloat || restar

entero:

si (ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codPh || CastFloat || ExpresionNiv2.codP || restar

sino si (ExpresionNiv2.tipo = natural)

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || CastInt || restar

sino

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || restar

natural:

si (ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codPh || CastFloat || ExpresionNiv2.codP || restar

sino si (ExpresionNiv2.tipo = entero)

ExpresiónNiv1Rec$_0$.codPh || CastInt || ExpresionNiv2.codP || restar

sino

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || restar

o:

ExpresiónNiv1Rec$_0$.codPh || ExpresiónNiv2.codP || o

ExpresiónNiv1Rec$_1$.codJh =

case (OpNiv1.op)

suma:

si (ExpresiónNiv1Rec$_0$.tipoh = float)

si(ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || fadd

sino

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || i2f || fadd

sino

si(ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codJh || i2f || ExpresiónNiv2. codJ || fadd

sino

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || iadd

resta:

si (ExpresiónNiv1Rec$_0$.tipoh = float)

si(ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || fsub

sino

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || i2f || fsub

sino

si(ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codJh || i2f || ExpresiónNiv2. codJ || fsub

sino

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || isub

o:

ExpresiónNiv1Rec$_0$.codJh ||

ifne +7 ||

ExpresionNiv2.codJ ||
ifeq +7 ||
iconst_1 ||
goto +4 ||
iconst_0

ExpresiónNiv1Rec$_0$.codP = ExpresiónNivRec$_1$.codP
ExpresiónNiv1Rec$_0$.codJ = ExpresiónNivRec$_1$.codJ

(eliminamos la recursión a izquierdas)
ExpresiónNiv2 → ExpresiónNiv3 ExpresiónNiv2Rec
ExpresiónNiv2Rec.codPh = ExpresiónNiv3.codP
ExpresiónNiv2Rec.codJh = ExpresiónNiv3.codJ
ExpresiónNiv2.codP = ExpresiónNiv2Rec.codP
ExpresiónNiv2.codJ = ExpresiónNiv2Rec.codJ
ExpresiónNiv2Rec → λ
ExpresiónNiv2Rec.codP = ExpresiónNiv2Rec.codPh
ExpresiónNiv2Rec.codJ = ExpresiónNiv2Rec.codJh
ExpresiónNiv2Rec → OpNiv2 ExpresiónNiv3 ExpresiónNiv2Rec
ExpresiónNiv2Rec$_1$.codPh =
case(OpNiv2.op)
Multiplica:
case (ExpresiónNiv2Rec$_0$.tipoh)
float:
si(ExpresionNiv3.tipo = float)
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Mul
sino
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || CastFloat || Mul
entero:
si (ExpresionNiv3 .tipo = float)
ExpresiónNiv2Rec$_0$.codPh || CastFloat || ExpresiónNiv3.codP || Mul
sino si (ExpresionNiv3 .tipo = natural)
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || CastInt || Mul
sino
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Mul
natural:
si (ExpresionNiv3 .tipo = float)
ExpresiónNiv2Rec$_0$.codPh || CastFloat || ExpresiónNiv3.codP || Mul
sino si (ExpresionNiv3 .tipo = entero)
ExpresiónNiv2Rec$_0$.codPh || CastInt || ExpresiónNiv3.codP || Mul
sino
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Mul
Divide:
case (ExpresiónNiv2Rec$_0$.tipoh)
float:
si(ExpresionNiv3.tipo = float)
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Div
sino
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || CastFloat || Div
entero:
si (ExpresionNiv3 .tipo = float)
ExpresiónNiv2Rec$_0$.codPh || CastFloat || ExpresiónNiv3.codP || Div
sino si (ExpresionNiv3 .tipo = natural)

ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || CastInt || Div

sino

ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Div

natural:

si (ExpresionNiv3 .tipo = float)

ExpresiónNiv2Rec$_0$.codPh || CastFloat || ExpresiónNiv3.codP || Div

sino si (ExpresionNiv3 .tipo = entero)

ExpresiónNiv2Rec$_0$.codPh || CastInt || ExpresiónNiv3.codP || Div

sino

ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Div

Modulo:

ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Mod

y:

ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Y

ExpresiónNiv2Rec$_1$.codJh =

case(OpNiv2.op)

Multiplica:

si(ExpresiónNiv2Rec$_0$.tipoh = float)

si(ExpresionNiv3.tipo = float)

ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ || fmul

sino

ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ || i2f || fmul

sino

si(ExpresionNiv3.tipo = float)

ExpresiónNiv2Rec$_0$.codJh || i2f || ExpresionNiv3.codJ|| fmul

sino

ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ|| imul

Divide:

si(ExpresiónNiv2Rec$_0$.tipoh = float)

si(ExpresionNiv3.tipo = float)

ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ || fdiv

sino

ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ || i2f || fdiv

sino

si(ExpresionNiv3.tipo = float)

ExpresiónNiv2Rec$_0$.codJh || i2f || ExpresionNiv3.codJ|| fdiv

sino

ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ|| idiv

Modulo:

ExpresiónNiv2Rec$_0$.codJh || ExpresiónNiv3.codJ || imod

y:

ExpresiónNiv2Rec$_0$.codJh ||

ifeq +11 ||

ExpresiónNiv3.codJ ||

ifeq +7 ||

iconst_1 ||

goto +4 ||

iconst_0

ExpresiónNiv2Rec$_0$.codP = ExpresiónNiv2Rec$_1$.codP

ExpresiónNiv2Rec$_0$.codJ = ExpresiónNiv2Rec$_1$.codJ

(factorizamos)
ExpresiónNiv3 → ExpresiónNiv4 ExpresiónNiv3Fact
ExpresiónNiv3Fact.codPh = ExpresiónNiv4.codP
ExpresiónNiv3Fact.codJh = ExpresiónNiv4.codJ
ExpresiónNiv3.codP = ExpresiónNiv3Fact.codP
ExpresiónNiv3.codJ = ExpresiónNiv3Fact.codJ

ExpresiónNiv3Fact → λ
ExpresiónNiv3Fact.codP = ExpresiónNiv3Fact.codPh
ExpresiónNiv3Fact.codJ = ExpresiónNiv3Fact.codJh

ExpresiónNiv3Fact → OpNiv3 ExpresiónNiv3
ExpresiónNiv3Fact.codP =
case (OpNiv3.op)
shl:
ExpresiónNiv3Fact.codPh || ExpresiónNiv3.codP || shl
shr:
ExpresiónNiv3Fact.codPh || ExpresiónNiv3.codP || shr
ExpresiónNiv3Fact.codJ =
case (OpNiv3.op)
shl:
ExpresiónNiv3Fact.codJh || ExpresiónNiv3.codJ|| ishl
shr:
ExpresiónNiv3Fact.codJh || ExpresiónNiv3.codJ || ishr

8. Esquema de traducción orientado a las gramáticas de atributos

El siguiente esquema se ha realizado a partir de la fusión de las las gramáticas del punto 3 (creación tabla de símbolos), 4 (restricciones contextuales) y 5 (traducción), sustituyendo además las construcciones iniciales por aquellas resultantes de realizar los acondicionamientos en el punto 7:

Programa →
Declaraciones
&
{Instrucciones.tsh = Declaraciones.ts}
Instrucciones
{
Programa.error = Declaraciones.error v Instrucciones.error
Programa.ts = Declaraciones.ts
Programa.codJ = Instrucciones.codJ
Programa.codP = Instrucciones.codP
}

Declaración → id : Tipo
{
Declaración.id = id.lex
Declaración.tipo = Tipo.tipo
Declaración.error = (Tipo.tipo=error)
}

Tipo → Boolean
Tipo.tipo = boolean
Tipo → character
Tipo.tipo = character
Tipo → Float
Tipo.tipo = float

Tipo → Natural

Tipo.tipo = natural

Declaraciones →

Declaracion

{

DeclaracionesFact.idh = Declaracion.id

DeclaracionesFact.tipoh = Declaracion.tipo

DeclaracionesFact.errorh = Declaracion.error

}

DeclaracionesFact

{

Declaraciones.ts = DeclaracionesFact.ts

Declaraciones.error = DeclaracionesFact.error

Declaraciones.numVars = DeclaracionesFact.numVars

}

DeclaracionesFact → λ

{

DeclaracionesFact.error = DeclaracionesFact.errorh

DeclaracionFact.ts =

inserta(creaTS(),DeclaracionesFact.idh,<tipo:DeclaracionesFact.tipoh, dir:0>)

DeclaracionesFact.numVars = 1

}

DeclaracionesFact →

;

Declaraciones

{

DeclaracionesFact.error = DeclaracionesFact.errorh v

Declaraciones.error v existe(Declaraciones.ts,DeclaracionFact.idh)

DeclaracionesFact.ts =

inserta(Declaraciones.ts, DeclaracionesFact.idh,

<tipo:DeclaracionesFact.tipoh, dir:Declaraciones.numVars>)

DeclaracionesFact.numVars = Declaraciones.numVars + 1

}

Instrucciones →

{

Instrucción.tsh = Instrucciones.tsh

InstruccionesFact.tsh = Instrucciones.tsh

}

Instrucción

{

InstruccionesFact.errorh = Instruccion.error

InstruccionesFact.codPh = Instrucción.codP

InstruccionesFact.codJh = Instrucción.codJ

}

InstruccionesFact

{

Instrucciones.error = InstruccionesFact.error

Instrucciones.codP = InstruccionesFact.codP

Instrucciones.codJ = InstruccionesFact.codJ

}

InstruccionesFact → λ

{

InstruccionesFact.error = InstruccionesFact.errorh

InstruccionesFact.codP = InstruccionesFact.codPh

InstruccionesFact.codJ = InstruccionesFact.codJh

}

InstruccionesFact →

;

{Instrucciones.tsh = InstruccionesFact.tsh}

Instrucciones

{

InstruccionesFact.error = InstruccionesFact.errorh v Instrucciones.error

InstruccionesFact.codP = InstruccionesFact.codPh || Instrucciones.codP

InstruccionesFact.codJ = InstruccionesFact.codJh || Instrucciones.codJ

}

Instrucción →

{InsLectura.tsh = Instrucción.tsh}

InsLectura

{

Instrucción.error = InsLectura.error

Instruccion.codP = InsLectura.codP

Instruccion.codJ = InsLectura.codJ

}

Instrucción →

{InsEscritura.tsh = Instrucción.tsh}

InsEscritura

{

Instrucción.error = InsEscritura.error

Instrucción.codP = InsEscritura.codP

Instrucción.codJ = InsEscritura.codJ

}

Instrucción →

{InsAsignacion.tsh = Instrucción.tsh}

InsAsignacion

{

Instrucción.error = InsAsignacion.error

Instrucciones.codP = InsAsignación.codP

Instrucciones.codJ = InsAsignación.codJ

}

InsLectura → in(id)

{

InsLectura.error = NOT existeID(InsLectura.tsh,id.lex)

InsLectura.codP = in InsLectura.tsh[id.lex].dir

InsLectura.codJ = lectura dameNumTipo(InsLectura.tsh[id.lex].tipo) InsLectura.tsh[id.lex].dir

}

InsEscritura → out(Expresión)

{

InsEscritura.error = (Expresión.tipo = error)

InsEscritura.codP = Expresión.codP || out

InsEscritura.codJ = apilaPrinter || Expresion.codJ || escritura dameNumTipo(InsEscritura.tsh[id.lex].tipo)

}

InsAsignación →

id :=

{Expresion.tsh = InsAsignación.tsh}

Expresión

{

InsAsignación.error = (NOT existeID(InsAsignación.tsh,id.lex)) v (Expresion.tipo = error) v
(id.tipo = float ∧ (Expresión.tipo = character v Expresión.tipo = boolean)) v
(id.tipo = integer ∧ (Expresión.tipo=float v Expresión.tipo = character v
Expresión.tipo = boolean)) v

(id.tipo = natural ∧ Expresión.tipo =/= natural) v
(id.tipo = character ∧ Expresión.tipo =/= character) v
(id.tipo = boolean ∧ Expresión.tipo =/= boolean)

InsAsignación.codP = Expresión.codP || desapila-dir InsAsignación.tsh[id.lex].dir

InsAsignacion.codJ =

case (InstAsignacion.tsh[id.lex].tipo)

boolean:

Expresion.codJ || i2b || istore InstAsignación.tsh[id.lex].dir

character:

Expresion.codJ || i2c || istore InstAsignación.tsh[id.lex].dir

natural:

entero:

Expresion.codJ || istore InstAsignación.tsh[id.lex].dir

float:

Expresion.codJ || fstore InstAsignación.tsh[id.lex].dir

}

Expresión →

{

ExpresionNiv1.tsh = Expresion.tsh

ExpresionFact.tsh = Expresion.tsh

}

ExpresiónNiv1

{

ExpresionFact.tipoh = ExpresionNiv1.tipo

ExpresiónFact.codPh = ExpresionNiv1.codP

ExpresiónFact.codJh = ExpresiónNiv1.codJ

}

ExpresiónFact

{

Expresion.tipo = ExpresionFact.tipo

Expresión.codP = ExpresiónFact.codP

Expresión.codJ = ExpresiónFact.codJ

}

ExpresiónFact → λ

{

ExpresionFact.tipo = ExpresionFact.tipoh

ExpresiónFact.codP = ExpresiónFact.codPh

ExpresiónFact.codJ = ExpresiónFact.codJh

}

ExpresiónFact →

OpNiv0

{ExpresionNiv1.tsh = ExpresionFact.tsh}

ExpresiónNiv1

{

ExpresionFact.tipo = si (ExpresionFact.tipoh = error v ExpresionNiv1.tipo = error) v
(ExpresionFact.tipoh = character ∧ ExpresionNiv1.tipo =/= character) v
(ExpresionFact.tipoh =/= character ∧ ExpresionNiv1.tipo = character)
(ExpresionFact.tipoh = boolean ∧ ExpresionNiv1.tipo =/= boolean) v
(ExpresionFact.tipoh =/= boolean ∧ ExpresionNiv1.tipo = boolean))
error
sino boolean

ExpresiónFact.codP =

case (OpNiv0.op)

menor:

case (ExpresiónFact.tipoh)

```
            float:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || menor
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || menor
            entero:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || menor
                sino si (ExpresiónNiv1.tipo = natural)
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || menor
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || menor
            natural:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || menor
                sino si (ExpresiónNiv1.tipo = entero)
                    ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || menor
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || menor
            otro:
                ExpresiónFact.codPh || ExpresiónNiv1.codP || menor
mayor
    case (ExpresiónFact.tipoh)
            float:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || mayor
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || mayor
            entero:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || mayor
                sino si (ExpresiónNiv1.tipo = natural)
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || mayor
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || mayor
            natural:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || mayor
                sino si (ExpresiónNiv1.tipo = entero)
                    ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || mayor
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || mayor
            otro:
                ExpresiónFact.codPh || ExpresiónNiv1.codP ||  mayor
menor-ig
    case (ExpresiónFact.tipoh)
            float:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || menorIg
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || menorIg
            entero:
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || menorIg
                sino si (ExpresiónNiv1.tipo = natural)
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || menorIg
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || menorIg
            natural:
                si (ExpresiónNiv1.tipo = float)
```

```
                    ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || menorIg
                sino si (ExpresiónNiv1.tipo = entero)
                    ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || menorIg
                sino
                    ExpresiónFact.codPh || ExpresiónNiv1.codP || menorIg
            otro:
                ExpresiónFact.codPh || ExpresiónNiv1.codP ||  menorIg
mayor-ig
    case (ExpresiónFact.tipoh)
        float:
            si (ExpresiónNiv1.tipo = float)
                ExpresiónFact.codPh || ExpresiónNiv1.codP || mayorIg
            sino
                ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || mayorIg
        entero:
            si (ExpresiónNiv1.tipo = float)
                ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || mayorIg
            sino si (ExpresiónNiv1.tipo = natural)
                ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || mayorIg
            sino
                ExpresiónFact.codPh || ExpresiónNiv1.codP || mayorIg
        natural:
            si (ExpresiónNiv1.tipo = float)
                ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || mayorIg
            sino si (ExpresiónNiv1.tipo = entero)
                ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || mayorIg
            sino
                ExpresiónFact.codPh || ExpresiónNiv1.codP || mayorIg
        otro:
            ExpresiónFact.codPh || ExpresiónNiv1.codP ||  mayorIg
igual
    case (ExpresiónFact0.tipoh)
        float:
            si (ExpresiónNiv1.tipo = float)
                ExpresiónFact.codPh || ExpresiónNiv1.codP || igual
            sino
                ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || igual
        entero:
            si (ExpresiónNiv1.tipo = float)
                ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || igual
            sino si (ExpresiónNiv1.tipo = natural)
                ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || igual
            sino
                ExpresiónFact.codPh || ExpresiónNiv1.codP || igual
        natural:
            si (ExpresiónNiv1.tipo = float)
                ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || igual
            sino si (ExpresiónNiv1.tipo = entero)
                ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || igual
            sino
                ExpresiónFact.codPh || ExpresiónNiv1.codP || igual
        otro:
            ExpresiónFact0.codPh || ExpresiónNiv1.codP ||  igual
no-igual
    case (ExpresiónFact0.tipoh)
        float:
            si (ExpresiónNiv1.tipo = float)
                ExpresiónFact.codPh || ExpresiónNiv1.codP || no-igual
            sino
                ExpresiónFact.codPh || ExpresiónNiv1.codP || CastFloat || no-igual
```

```
                    entero:
                        si (ExpresiónNiv1.tipo = float)
                            ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || no-igual
                        sino si (ExpresiónNiv1.tipo = natural)
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || CastInt || no-igual
                        sino
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || no-igual
                    natural:
                        si (ExpresiónNiv1.tipo = float)
                            ExpresiónFact.codPh || CastFloat || ExpresiónNiv1.codP || no-igual
                        sino si (ExpresiónNiv1.tipo = entero)
                            ExpresiónFact.codPh || CastInt || ExpresiónNiv1.codP || no-igual
                        sino
                            ExpresiónFact.codPh || ExpresiónNiv1.codP || no-igual
                    otro:
                        ExpresiónFact.codPh || ExpresiónNiv1.codP ||  no-igual

ExpresiónFact.codJ =
    case (OpNiv0.op)
        menor:
            si (ExpresiónFact.tipoh = float)
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    fcmpg ||
                    if_ge +7
                    iconst_1
                    goto +4
                    iconst_0
                sino
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    i2f ||
                    fcmpg ||
                    if_ge +7
                    iconst_1
                    goto +4
                    iconst_0
            sino
                si(ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codJh ||
                    i2f ||
                    ExpresiónNiv1.codJ ||
                    fcmpg ||
                    if_ge +7
                    iconst_1
                    goto +4
                    iconst_0
                sino
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
                    if_icmpge +7
                    iconst_1
                    goto +4
                    iconst_0
        mayor
            si (ExpresiónFact.tipoh = float)
                si (ExpresiónNiv1.tipo = float)
                    ExpresiónFact.codJh ||
                    ExpresiónNiv1.codJ ||
```

```
                fcmpg ||
                if_le +7
                iconst_1
                goto +4
                iconst_0
            sino
                ExpresiónFact.codJh ||
                ExpresiónNiv1.codJ ||
                i2f ||
                fcmpg ||
                if_le +7
                iconst_1
                goto +4
                iconst_0
        sino
            si(ExpresiónNiv1.tipo = float)
                ExpresiónFact.codJh ||
                i2f ||
                ExpresiónNiv1.codJ ||
                fcmpg ||
                if_le +7
                iconst_1
                goto +4
                iconst_0
            sino
                ExpresiónFact.codJh ||
                ExpresiónNiv1.codJ ||
                if_icmple +7
                iconst_1
                goto +4
                iconst_0
menor-ig
    si (ExpresiónFact.tipoh = float)
        si (ExpresiónNiv1.tipo = float)
            ExpresiónFact.codJh ||
            ExpresiónNiv1.codJ ||
            fcmpg ||
            if_gt +7
            iconst_1
            goto +4
            iconst_0
        sino
            ExpresiónFact.codJh ||
            ExpresiónNiv1.codJ ||
            i2f ||
            fcmpg ||
            if_gt +7
            iconst_1
            goto +4
            iconst_0
    sino
        si(ExpresiónNiv1.tipo = float)
            ExpresiónFact.codJh ||
            i2f ||
            ExpresiónNiv1.codJ ||
            fcmpg ||
            if_gt +7
            iconst_1
            goto +4
            iconst_0
```

```
            sino
                ExpresiónFact.codJh ||
                ExpresiónNiv1.codJ ||
                if_icmpgt +7
                iconst_1
                goto +4
                iconst_0
mayor-ig
    si (ExpresiónFact.tipoh = float)
        si (ExpresiónNiv1.tipo = float)
            ExpresiónFact.codJh ||
            ExpresiónNiv1.codJ ||
            fcmpg ||
            if_lt +7
            iconst_1
            goto +4
            iconst_0
        sino
            ExpresiónFact.codJh ||
            ExpresiónNiv1.codJ ||
            i2f ||
            fcmpg ||
            if_lt +7
            iconst_1
            goto +4
            iconst_0
    sino
        si(ExpresiónNiv1.tipo = float)
            ExpresiónFact.codJh ||
            i2f ||
            ExpresiónNiv1.codJ ||
            fcmpg ||
            if_lt +7
            iconst_1
            goto +4
            iconst_0
        sino
            ExpresiónFact.codJh ||
            ExpresiónNiv1.codJ ||
            if_icmplt +7
            iconst_1
            goto +4
            iconst_0
igual
    si (ExpresiónFact.tipoh = float)
        si (ExpresiónNiv1.tipo = float)
            ExpresiónFact.codJh ||
            ExpresiónNiv1.codJ ||
            fcmpg ||
            if_ne +7
            iconst_1
            goto +4
            iconst_0
        sino
            ExpresiónFact.codJh ||
            ExpresiónNiv1.codJ ||
            i2f ||
            fcmpg ||
            if_ne +7
            iconst_1
```

goto +4
iconst_0
sino
si(ExpresiónNiv1.tipo = float)
ExpresiónFact.codJh ||
i2f ||
ExpresiónNiv1.codJ ||
fcmpg ||
if_ne +7 ||
iconst_1 ||
goto +4 ||
iconst_0
sino
ExpresiónFact.codJh ||
ExpresiónNiv1.codJ ||
if_icmpne +7 ||
iconst_1 ||
goto +4 ||
iconst_0
no-igual
si (ExpresiónFact.tipoh = float)
si (ExpresiónNiv1.tipo = float)
ExpresiónFact.codJh ||
ExpresiónNiv1.codJ ||
fcmpg ||
if_eq +7 ||
iconst_1 ||
goto +4 ||
iconst_0
sino
ExpresiónFact.codJh ||
ExpresiónNiv1.codJ ||
i2f ||
fcmpg ||
if_eq +7 ||
iconst_1 ||
goto +4 ||
iconst_0
sino
si(ExpresiónNiv1.tipo = float)
ExpresiónFact.codJh ||
i2f ||
ExpresiónNiv1.codJ ||
fcmpg ||
if_eq +7 ||
iconst_1 ||
goto +4 ||
iconst_0
sino
ExpresiónFact.codJh ||
ExpresiónNiv1.codJ ||
if_icmpeq +7 ||
iconst_1 ||
goto +4 ||
iconst_0
}

ExpresiónNiv1 →
{
ExpresionNiv2.tsh = ExpresionNiv1.tsh

ExpresionNiv1Rec.tsh = ExpresionNiv1.tsh
}
ExpresiónNiv2
{
ExpresionNiv1Rec.tipoh = ExpresiónNiv2.tipo
ExpresiónNiv1Rec.codPh = ExpresiónNiv2.codP
ExpresiónNiv1Rec.codJh = ExpresiónNiv2.codJ
}
ExpresiónNiv1Rec
{
ExpresionNiv1.tipo = ExpresiónNiv2Rec.tipo
ExpresiónNiv1.codP = ExpresiónNiv1Rec.codP
ExpresiónNiv1.codJ = ExpresiónNiv1Rec.codJ
}

ExpresiónNiv1Rec → λ
{
ExpresionNiv1Rec.tipo = ExpresionNiv1Rec.tipoh
ExpresiónNiv1Rec.codP = ExpresiónNiv1Rec.codPh
ExpresiónNiv1Rec.codJ = ExpresiónNiv1Rec.codJh
}
ExpresiónNiv1Rec →
OpNiv1
{
ExpresionNiv2.tsh = ExpresionNiv1Rec0.tsh
}
ExpresiónNiv2
{
$ExpresionNiv1Rec_1$.tsh = $ExpresionNiv1Rec_0$.tsh
$ExpresionNiv1Rec_1$.tipoh =
si ($ExpresionNiv1Rec_0$.tipoh = error v ExpresionNiv2.tipo = error v
$ExpresionNiv1Rec_0$.tipoh = char v ExpresionNiv2.tipo = char v
($ExpresionNiv1Rec_0$.tipoh = boolean ∧ ExpresionNiv2.tipo =/= boolean) v
($ExpresionNiv1Rec_0$.tipoh =/= boolean ∧ ExpresionNiv2.tipo = boolean))
error
sino case (OpNiv1.op)
suma,resta:
si ($ExpresionNiv1Rec_0$.tipoh=float v ExpresionNiv2.tipo = float)
float
sino si ($ExpresionNiv1Rec_0$.tipoh =integer v ExpresionNiv2.tipo = integer)
integer
sino si ($ExpresionNiv1Rec_0$.tipoh =natural ∧ ExpresionNiv2.tipo = natural)
natural
sino error
o:
si ($ExpresionNiv1Rec_0$.tipoh = boolean ∧ ExpresionNiv2.tipo = boolean)
boolean
sino error

$ExpresiónNiv1Rec_1$.codPh =
case (OpNiv1.op)
suma:
case ($ExpresiónNiv1Rec_0$.tipoh)
float:
si(ExpresionNiv2 .tipo = float)

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || sumar

sino

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || CastFloat || sumar

entero:

si (ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codPh || CastFloat || ExpresionNiv2.codP || sumar

sino si (ExpresionNiv2.tipo = natural)

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || CastInt || sumar

sino

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || sumar

natural:

si (ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codPh || CastFloat || ExpresionNiv2.codP || sumar

sino si (ExpresionNiv2.tipo = entero)

ExpresiónNiv1Rec$_0$.codPh || CastInt || ExpresionNiv2.codP || sumar

sino

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || sumar

resta:

case (ExpresiónNiv1Rec$_0$.tipoh)

float:

si(ExpresionNiv2 .tipo = float)

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || restar

sino

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || CastFloat || restar

entero:

si (ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codPh || CastFloat || ExpresionNiv2.codP || restar

sino si (ExpresionNiv2.tipo = natural)

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || CastInt || restar

sino

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || restar

natural:

si (ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codPh || CastFloat || ExpresionNiv2.codP || restar

sino si (ExpresionNiv2.tipo = entero)

ExpresiónNiv1Rec$_0$.codPh || CastInt || ExpresionNiv2.codP || restar

sino

ExpresiónNiv1Rec$_0$.codPh || ExpresionNiv2.codP || restar

o:

ExpresiónNiv1Rec$_0$.codPh || ExpresiónNiv2.codP || o

ExpresiónNiv1Rec$_1$.codJh =

case (OpNiv1.op)

suma:

si (ExpresiónNiv1Rec$_0$.tipoh = float)

si(ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || fadd

sino

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || i2f || fadd

sino

si(ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codJh || i2f || ExpresiónNiv2. codJ || fadd

sino

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || iadd

resta:

si (ExpresiónNiv1Rec$_0$.tipoh = float)

si(ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || fsub

sino

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || i2f || fsub

sino

si(ExpresionNiv2.tipo = float)

ExpresiónNiv1Rec$_0$.codJh || i2f || ExpresiónNiv2. codJ || fsub

sino

ExpresiónNiv1Rec$_0$.codJh || ExpresiónNiv2. codJ || isub

o:

ExpresiónNiv1Rec$_0$.codJh ||

ifne +7 ||
ExpresionNiv2.codJ ||
ifeq +7 ||
iconst_1 ||
goto +4 ||
iconst_0

}

ExpresiónNiv1Rec

{

ExpresionNivlRec$_0$.tipo = ExpresionNiv1Rec$_1$.tipo

ExpresiónNiv1Rec$_0$.codP = ExpresiónNiv1Rec$_1$.codP

ExpresiónNiv1Rec$_0$.codJ = ExpresiónNiv1Rec$_1$.codJ

}

ExpresiónNiv2 →

{

ExpresionNiv3.tsh = ExpresionNiv2.tsh
ExpresionNiv2Rec.tsh = ExpresionNiv2.tsh
}

ExpresiónNiv3

{

ExpresionNiv2Rec.tipoh = ExpresiónNiv3.tipo
ExpresiónNiv2Rec.codPh = ExpresiónNiv3.codP
ExpresiónNiv2Rec.codJh = ExpresiónNiv3.codJ
}

ExpresiónNiv2Rec

{

ExpresionNiv2.tipo = ExpresiónNiv2Rec.tipo
ExpresiónNiv2.codP = ExpresiónNiv2Rec.codP
ExpresiónNiv2.codJ = ExpresiónNiv2Rec.codJ
}

ExpresiónNiv2Rec → λ

{

ExpresionNiv2Rec.tipo = ExpresionNiv2Rec.tipoh
ExpresiónNiv2Rec.codP = ExpresiónNiv2Rec.codPh
ExpresiónNiv2Rec.codJ = ExpresiónNiv2Rec.codJh
}

ExpresiónNiv2Rec →

OpNiv2

{

ExpresionNiv3.tsh = ExpresionNiv2Rec$_0$.tsh

}

ExpresiónNiv3

{

ExpresionNiv2Rec$_1$.tsh = ExpresionNiv2Rec$_0$.tsh

ExpresionNiv2Rec$_1$.tipoh =

si (ExpresionNiv2Rec$_0$.tipoh = error v ExpresionNiv3.tipo = error v

ExpresionNiv2Rec$_0$.tipoh = character v ExpresionNiv3.tipo = character v

(ExpresionNiv2Rec$_0$.tipoh = boolean ∧ ExpresionNiv3.tipo =/= boolean v

(ExpresionNiv2Rec$_0$.tipoh =/= boolean ∧ ExpresionNiv3.tipo = boolean))

error

sino case (OpNiv2.op)

multiplica,divide:

si (ExpresionNiv2Rec$_0$.tipoh=float v ExpresionNiv3.tipo = float)

float

sino si (ExpresionNiv2Rec$_0$.tipoh =integer v ExpresionNiv3.tipo = integer)

integer

sino si (ExpresionNiv2Rec$_0$.tipoh =natural ∧ ExpresionNiv3.tipo=natural)

natural

sino error

modulo:

si (ExpresionNiv3.tipo = natural ∧

(ExpresionNiv2Rec$_0$.tipoh=natural v ExpresionNiv2Rec$_0$.tipoh=integer))

ExpresionNiv2Rec$_0$.tipoh

sino error

y:

si (ExpresionNiv2Rec$_0$.tipoh = boolean ∧ ExpresionNiv3.tipo = boolean)

boolean

sino error

ExpresiónNiv2Rec$_1$.codPh =

case(OpNiv2.op)

Multiplica:

case (ExpresiónNiv2Rec$_0$.tipoh)

float:

si(ExpresionNiv3.tipo = float)

ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Mul

sino

ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || CastFloat || Mul

entero:

si (ExpresionNiv3 .tipo = float)

ExpresiónNiv2Rec$_0$.codPh || CastFloat || ExpresiónNiv3.codP || Mul

sino si (ExpresionNiv3 .tipo = natural)

ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || CastInt || Mul

sino

ExpresiónNiv2Rec0.codPh || ExpresiónNiv3.codP || Mul

natural:

si (ExpresionNiv3 .tipo = float)

ExpresiónNiv2Rec$_0$.codPh || CastFloat || ExpresiónNiv3.codP || Mul

sino si (ExpresionNiv3 .tipo = entero)
ExpresiónNiv2Rec$_0$.codPh || CastInt || ExpresiónNiv3.codP || Mul
sino
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Mul

Divide:
case (ExpresiónNiv2Rec$_0$.tipoh)
float:
si(ExpresionNiv3.tipo = float)
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Div
sino
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || CastFloat || Div
entero:
si (ExpresionNiv3 .tipo = float)
ExpresiónNiv2Rec$_0$.codPh || CastFloat || ExpresiónNiv3.codP || Div
sino si (ExpresionNiv3 .tipo = natural)
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || CastInt || Div
sino
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Div
natural:
si (ExpresionNiv3 .tipo = float)
ExpresiónNiv2Rec$_0$.codPh || CastFloat || ExpresiónNiv3.codP || Div
sino si (ExpresionNiv3 .tipo = entero)
ExpresiónNiv2Rec$_0$.codPh || CastInt || ExpresiónNiv3.codP || Div
sino
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Div

Modulo:
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Mod

y:
ExpresiónNiv2Rec$_0$.codPh || ExpresiónNiv3.codP || Y

ExpresiónNiv2Rec$_1$.codJh =
case(OpNiv2.op)
Multiplica:
si(ExpresiónNiv2Rec$_0$.tipoh = float)
si(ExpresionNiv3.tipo = float)
ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ || fmul
sino
ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ || i2f || fmul
sino
si(ExpresionNiv3.tipo = float)
ExpresiónNiv2Rec$_0$.codJh || i2f || ExpresionNiv3.codJ|| fmul
sino
ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ|| imul

Divide:
si(ExpresiónNiv2Rec$_0$.tipoh = float)
si(ExpresionNiv3.tipo = float)
ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ || fdiv
sino
ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ || i2f || fdiv
sino
si(ExpresionNiv3.tipo = float)

ExpresiónNiv2Rec$_0$.codJh || i2f || ExpresionNiv3.codJ|| fdiv

sino

ExpresiónNiv2Rec$_0$.codJh || ExpresionNiv3.codJ|| idiv

Modulo:

ExpresiónNiv2Rec$_0$.codJh || ExpresiónNiv3.codJ || imod

y:

ExpresiónNiv2Rec$_0$.codJh ||
ifeq +11 ||
ExpresiónNiv3.codJ ||
ifeq +7 ||
iconst_1 ||
goto +4 ||
iconst_0

}

ExpresiónNiv2Rec

{

ExpresionNiv2Rec$_0$.tipo = ExpresionNiv2Rec$_1$.tipo

ExpresiónNiv2Rec$_0$.codP = ExpresiónNiv2Rec$_1$.codP

ExpresiónNiv2Rec$_0$.codJ = ExpresiónNiv2Rec$_1$.codJ

}

ExpresiónNiv3 →

{

ExpresiónNiv4.tsh = ExpresiónNiv3.tsh

ExpresiónNiv3Fact.tsh = ExpresiónNiv3.tsh

}

ExpresiónNiv4

{

ExpresionNiv3Fact.tipoh = ExpresiónNiv4.tipo

ExpresiónNiv3Fact.codPh = ExpresiónNiv4.codP

ExpresiónNiv3Fact.codJh = ExpresiónNiv4.codJ

}

ExpresiónNiv3Fact

{

ExpresionNiv3.tipo = ExpresionNiv3Fact.tipo

ExpresiónNiv3.codP = ExpresiónNiv3Fact.codP

ExpresiónNiv3.codJ = ExpresiónNiv3Fact.codJ

}

ExpresiónNiv3Fact → λ

{

ExpresionNiv3Fact.tipo = ExpresionNiv3Fact.tipoh

ExpresiónNiv3Fact.codP = ExpresiónNiv3Fact.codPh

ExpresiónNiv3Fact.codJ = ExpresiónNiv3Fact.codJh

}

ExpresiónNiv3Fact →

OpNiv3

{

ExpresiónNiv3.tsh = ExpresiónNiv3Fact.tsh

}

ExpresiónNiv3

{

ExpresionNiv3Fact.tipo =

si (ExpresionNiv3Fact.tipoh = error v ExpresionNiv3.tipo = error v
ExpresionNiv3Fact.tipoh =/= natural v ExpresionNiv3.tipo =/= natural)

error

sino natural

ExpresiónNiv3Fact.codP =
case (OpNiv3.op)
shl:
ExpresiónNiv3Fact.codPh || ExpresiónNiv3.codP || shl
shr:
ExpresiónNiv3Fact.codPh || ExpresiónNiv3.codP || shr

ExpresiónNiv3Fact.codJ =
case (OpNiv3.op)
shl:
ExpresiónNiv3Fact.codJh || ExpresiónNiv3.codJ|| ishl
shr:
ExpresiónNiv3Fact.codJh || ExpresiónNiv3.codJ || ishr
}

ExpresiónNiv4 →
OpNiv4
{ExpresionNiv4$_1$.tsh = ExpresionNiv4$_o$.tsh}
ExpresiónNiv4
{
ExpresionNiv4$_o$.tipo =
si (ExpresionNiv4$_1$.tipo = error)
error
sino case (OpNiv4.op)
no:
si (ExpresionNiv4$_1$.tipo=boolean)
boolean
sino error
menos:
si (ExpresionNiv4$_1$.tipo=float)
float
sino si (ExpresionNiv4$_1$.tipo=integer v ExpresionNiv4$_1$.tipo = natural)
integer
sino error
cast-float:
si (ExpresionNiv4$_1$.tipo=/=boolean)
float
sino error
cast-int:
si (ExpresionNiv4$_1$.tipo=/=boolean)
float
sino error
cast-nat:
si (ExpresionNiv4$_1$.tipo=natural v ExpresionNiv4$_1$.tipo=character)
natural
sino error
cast-char:
si (ExpresionNiv4$_1$.tipo=natural v ExpresionNiv4$_1$.tipo=character)
character
sino error

ExpresiónNiv4$_o$.codP =
case (OpNiv4.op)

no:
ExpresiónNiv4$_1$.codP || no
negativo:
ExpresiónNiv4$_1$.codP || negativo
cast-float:
ExpresiónNiv4$_1$.codP || CastFloat
cast-int:
ExpresiónNiv4$_1$.codP || CastInt
cast-nat:
ExpresiónNiv4$_1$.codP || CastNat
cast-char:
ExpresiónNiv4$_1$.codP || CastChar

ExpresiónNiv4$_o$.codJ =
case (OpNiv4.op)
no:
ExpresiónNiv4$_1$.codJ ||
ifeq +7 ||
iconst_0 ||
goto +4 ||
iconst_1
negativo:
case (ExpresionNiv4$_1$.tipo)
entero:
ExpresiónNiv4$_1$.codJ || ineg
float:
ExpresiónNiv4$_1$.codJ || fneg
cast-float:
case (ExpresionNiv4$_1$.tipo)
character:
natural:
entero:
ExpresiónNiv4$_1$.codJ || i2f
float:
ExpresiónNiv4$_1$.codJ
cast-int:
case (ExpresionNiv4$_1$.tipo)
character:
natural:
entero:
ExpresiónNiv4$_1$.codJ
float:
ExpresiónNiv4$_1$.codJ || f2i
cast-nat:
case (ExpresionNiv4$_1$.tipo)
character:
natural:
entero:
ExpresiónNiv4$_1$.codJ
float:
ExpresiónNiv4$_1$.codJ || f2i
cast-char:

case (ExpresionNiv4$_1$.tipo)
character:
natural:
entero:
ExpresiónNiv4$_1$.codJ
float:
ExpresiónNiv4$_1$.codJ || f2i
}

ExpresiónNiv4 →
{Expresion.tsh = ExpresionNiv4.tsh}
| Expresión |
{
ExpresionNiv4.tipo =
si (Expresion.tipo = error v Expresion.tipo=boolean v Expresion.tipo=character)
error
sino si (Expresion.tipo = float)
float
sino si (Expresion.tipo = natural v Expresion.tipo = integer)
natural
sino error

ExpresiónNiv4.codP = Expresión.codP || abs
ExpresionNiv4.codJ =
case (Expresion.tipo)
float:
Expresion.codJ ||
dup ||
fconst_0 ||
fcmpg ||
ifge +4
fneg
otro:
Expresion.codJ ||
dup ||
ifge +4
fneg
}

ExpresiónNiv4 →
{Expresion.tsh = ExpresionNiv4.tsh}
(Expresión)
{
ExpresionNiv4.tipo = Expresion.tipo
ExpresiónNiv4.codP = Expresión.codP
ExpresionNiv4.codJ = Literal.codJ
}

ExpresiónNiv4 →
{Literal.tsh = ExpresiónNiv4.tsh}
Literal
{
Expresion.tipo = Literal.tipo
ExpresiónNiv4.codP = Literal.codP
ExpresionNiv4.codJ = Literal.codJ
}

Literal → id
{

Literal.tipo = Literal.tsh[id.lex].tipo
Literal.codP = apila-dir Literal.tsh[id.lex].dir
Literal.codJ = case(Literal.tsh[id.lex].tipo)
boolean:
i2b || iload Literal.tsh[id.lex].dir
tCha:
i2c || iload Literal.tsh[id.lex].dir
natural:
entero:
iload Literal.tsh[id.lex].dir
float:
fload Literal.tsh[id.lex].dir
}

Literal → litNat
{
Literal.tipo = natural
Literal.codP = apila litNat.lex
Literal.codJ = iconst getValor(litNat.lex))
}

Literal → litFlo
{
Literal.tipo = float
Literal.codP = apila litFlo.lex
Literal.codJ = fconst getValor(litFlo.lex))
}

Literal → litTrue
{
Literal.tipo = boolean
Literal.codP = apila true
Literal.codJ = iconst 1
}

Literal → litFalse
{
Literal.tipo = boolean
Literal.codP = apila false
Literal.codJ = iconst 0
}

Literal → litCha
{
Literal.tipo = character
Literal.codP = apila litCha.lex
Literal.codJ = iconst getValor(litCha.lex))
}

OpNiv0 → <
{OpNiv0.op = menor}

OpNiv0 → >
{OpNiv0.op = mayor}

OpNiv0 → <=
{OpNiv0.op = menor-ig}

OpNiv0 → >=
{OpNiv0.op = mayor-ig}

OpNiv0 → =
{OpNiv0.op = igual}

OpNiv0 → =/=
{OpNiv0.op = no-igual}

OpNiv1 → +
{OpNiv1.op = suma}

OpNiv1 → -
{OpNiv1.op = resta}

OpNiv1 → or
{OpNiv1.op = o}
OpNiv2 → *
{OpNiv2.op = multiplica}
OpNiv2 → /
{OpNiv2.op = divide}
OpNiv2 → %
{OpNiv2.op = modulo}
OpNiv2 → and
{OpNiv2.op = y}
OpNiv3 → >>
{OpNiv3.op = shl}
OpNiv3 → <<
{OpNiv3.op = shr}
OpNiv4 → not
{OpNiv4.op = no}
OpNiv4 → -
{OpNiv4.op = menos}
OpNiv4 → (float)
{OpNiv4.op = cast-float}
OpNiv4 → (int)
{OpNiv4.op = cast-int}
OpNiv4 → (nat)
{OpNiv4.op = cast-nat}
OpNiv4 → (char)
{OpNiv4.op = cast-char}

9. Esquema de traducción orientado al traductor predictivo – recursivo_

Esquema de traducción en el que se haga explícito los parámetros utilizados para representar los atributos, así como en los que se muestre la implementación de las ecuaciones semánticas como asignaciones a dichos parámetros.

9.1. Variables globales

ts: Almacena la tabla de símbolos.
numVars: Almacena la siguiente dirección de memoria libre.

Nota: Se ha considerado que las variables para el código generado fuesen globales. Sin embargo, nuestro traductor tiene la cualidad de hacer castings automáticos en función del tipo expresiones para facilitar la escritura de expresiones que incluyan operandos de varios tipos. Esto ha implicado que la imposibilidad de "emitir" código añadido siempre por el final a una variable global, pues los castings se determinan después de generar el código al que deben preceder, por lo que, por el momento, hemos optado por utilizar variables locales para la generación de código.

9.2. Nuevas operaciones y transformación de ecuaciones semánticas

Las siguientes operaciones devuelven un booleano indicando si el Token actual corresponde a lo esperado. En función de ese resultado se puede decidir entre varias reglas de la gramática, o se pueden detectar errores en el código.

in(): sirve para comprobar que lo siguiente es un token "in"
out(): sirve para comprobar que lo siguiente es un token "out"
Identificador(out: lex): reconoce el token del identificador léxico y devuelve su lexema

dosPuntosIgual(): reconoce :=. Afecta a InsAsignación.
ampersand(): reconoce &. Afecta a Programa.
puntoYComa(): reconoce ; Afecta a Declaraciones e Instrucciones
barraVertical(): reconoce |. Afecta a Instrucciones
AbrePar(), CierraPar(): reconocen los paréntesis. Afectan a Instrucciones.

9.3. Esquema de traducción

Programa(out: error1, codP1, codJ1) →
Declaraciones(out error2)
ampersand()
Instrucciones(out: error3, codP3, codJ3)
{
error1 ← error2 v error3
codP1 ← codP3
codJ1 ← codJ3
}

Declaración(out: error1, id1, tipo1) →
id
dosPuntos(out:error2)
Tipo(out: tipo3
{
id1 ← id.lex
tipo1 ← tipo3
error1 ← id = null v tipo = error v error2
}

Tipo(out tipo1) → Boolean
tipo1 ← boolean
Tipo(out tipo1) → character
tipo1 ← character
Tipo(out tipo1) → Float
tipo1 ← float
Tipo(out tipo1) → Natural
tipo1 ← natural

Declaraciones (out: error1)→
Declaracion (out: id2, tipo2, error2)
{
idh3 ← id2
tipoh3 ← tipo2
errorh3 ← error2
}
DeclaracionesFact(in: idh3, tipoh3, errorh3; out: error3)
{
error1 ← error3
}

DeclaracionesFact(in: idh1, tipoh1, errorh1; out: error1) → λ
{
error1 ← errorh1
ts ← inserta(creaTS(), idh1,<tipo:tipoh1, dir: 0>
numVars ← 1
}

DeclaracionesFact(in: idh1, tipoh1, errorh1; out: error1) →
puntoYComa()
Declaraciones(out: error2)
{
error1 ← errorh1 v error2 v existe(ts, idh1)
ts ← inserta(ts, idh1, <tipo:tipoh1, dir: numVars>)
numVars ← numVars + 1
}

Instrucciones(out: error1,codP1,codJ1) →

Instrucción(out: error2,codP2,codJ2)
{
errorh3 ← error2
codPh3 ← codP2
codJh3 ← codJ2
}
InstruccionesFact(in: errorh3,codPh3,codJh3; out: error3,codP3,codJ3)
{
error1 ← error3
codP1 ← codP3
codJ1 ← codJ3
}

InstruccionesFact(in: errorh1,codPh1,codJh1; out: error1,codP1,codJ1) → λ
{
error1 ← errorh1
codP1 ← codPh1
codJ1 ← codJ1
}

InstruccionesFact(in: errorh1,codPh1,codJh1; out: error1,codP1,codJ1) →
puntoYComa()
Instrucciones(out: error2,codP2,codJ2)
{
error1 ← errorh1 v error2
codP1 ← codPh1 || codP2
codJ1 ← codJh1 || codJ2
}

Instrucción(out: error1,codP1,codJ1) →
InsLectura(out: error2,codP2,codJ2)
{
error1 ← error2
codP1 ← codP2
codJ1 ← codJ2

}
Instrucción(out: error1,codP1,codJ1) →
InsEscritura(out: error2,codP2,codJ2)
{
error1 ← error2
codP1 ← codP2
codJ1 ← codJ2
}

Instrucción(out: error1,codP1,codJ1) →
InsAsignacion(out: error2,codP2,codJ2)
{
error1 ← error2

codP1 ← codP2
codJ1 ← codJ2
}

InsLectura(out:error1,codP1,codJ1) →
in()
abrePar()
{
error1 ← NOT existeID(ts,id.lex)
codP1 ← in ts[id.lex].dir
codJ1 ← lectura dameNumTipo(ts[id.lex].tipo) ts[id.lex].dir
}
cierraPar()

InsEscritura(out: error1,codP1,codJ1) →
out()
abrePar()
Expresión(out: tipo2, codP2, codJ2)
cierraPar()
{
error1 ← (tipo2=Error)
codP1 ← codP2 || out
codJ1 ← apilaPrinter || codJ2 || escritura dameNumTipo(tipo2)
}

InsAsignación(out: error1,codP1,codJ1) →
dosPuntosIgual()
Expresión(out: tipo3,codP3,codJ3)
{
error1 ← (NOT existeID(ts,id.lex)) v (tipo3 = error) v
(ts[id.lex].tipo = float ∧ (tipo3 = character v tipo3 = boolean)) v
(ts[id.lex].tipo = integer ∧ (tipo3=float v tipo3 = character v tipo3 = boolean)) v
(ts[id.lex].tipo = natural ∧ tipo3 =/= natural) v
(ts[id.lex].tipo = character ∧ tipo3 =/= character) v
(ts[id.lex].tipo = boolean ∧ tipo3 =/= boolean)

direccion ← ts[id.lex].dir

codP1 ← codP3 || desapila-dir direccion
codJ1 ← case (ts[id.lex].tipo)
boolean:
codJ3 || i2b || istore direccion
character:
codJ3 || i2c || istore InstAsignación.tsh[id.lex].dir
natural:
entero:
codJ3 || istore direccion
float:
codJ3 || fstore direccion

Expresión(out: tipo1,codP1,codJ1) →
ExpresiónNiv1(out: tipo2,codP2,codJ2)
{
tipo3h ← tipo2
codPh3 ← codP2
codJh3 ← codJ2
}
ExpresiónFact(in: tipo3h,codPh3,codJh3; out: tipo3,codP3,codJ3)
{

```
    tipo1 ← tipo3
    codP1 ← codP3
    codJ1 ← codJ3
    }

ExpresiónFact(in: tipo1h,codPh1,codJh1; out: tipo1,codP1,codJ1) → λ
    {
    tipo1 ← tipo1h
    codP1 ← codPh1
    codJ1 ← codJh1
    }

ExpresiónFact(in: tipo1h,codPh1,codJh1; out: tipo1,codP1,codJ1) →
    OpNiv0(out: op2)
    ExpresiónNiv1(out: tipo3,codP3,codJ3)
    {
    tipo1 ←  si (tipo1h = error v tipo3 = error) v
                    ( tipo1h = character ∧ tipo3 =/= character) v
                    ( tipo1h =/= character ∧ tipo3 = character) v
                    (tipo1h = boolean ∧ tipo3 =/= boolean) v
                    ( tipo1h =/= boolean ∧ tipo3 = boolean))
                        error
                sino  boolean


    codP1 ←
        case (op2)
            menor:
                case (tipo1h)
                    float:
                        si (tipo3 = float)
                            codPh1 || codP3 || menor
                        sino
                            codPh1 || codP3 || CastFloat || menor
                    entero:
                        si (tipo3 = float)
                            codPh1 || CastFloat || codP3 || menor
                        sino si (tipo3 = natural)
                            codPh1 || codP3 || CastInt || menor
                        sino
                            codPh1 || codP3 || menor
                    natural:
                        si (tipo3 = float)
                            codPh1 || CastFloat || codP3 || menor
                        sino si (tipo3 = entero)
                            codPh1 || CastInt || codP3 || menor
                        sino
                            codPh1 || codP3 || menor
                    otro:
                        codPh1 || codP3 || menor
            mayor
                case (tipo1h)
                    float:
                        si (tipo3 = float)
                            codPh1 || codP3 || mayor
                        sino
                            codPh1 || codP3 || CastFloat || mayor
                    entero:
                        si (tipo3 = float)
                            codPh1 || CastFloat || codP3 || mayor
```

```
                    sino si (tipo3 = natural)
                        codPh1 || codP3 || CastInt || mayor
                    sino
                        codPh1 || codP3 || mayor
                natural:
                    si (tipo3 = float)
                        codPh1 || CastFloat || codP3 || mayor
                    sino si (tipo3 = entero)
                        codPh1 || CastInt || codP3 || mayor
                    sino
                        codPh1 || codP3 || mayor
                otro:
                    codPh1 || codP3 ||  mayor
menor-ig
        case (tipo1h)
                float:
                    si (tipo3 = float)
                        codPh1 || codP3 || menorIg
                    sino
                        codPh1 || codP3 || CastFloat || menorIg
                entero:
                    si (tipo3 = float)
                        codPh1 || CastFloat || codP3 || menorIg
                    sino si (tipo3 = natural)
                        codPh1 || codP3 || CastInt || menorIg
                    sino
                        codPh1 || codP3 || menorIg
                natural:
                    si (tipo3 = float)
                        codPh1 || CastFloat || codP3 || menorIg
                    sino si (tipo3 = entero)
                        codPh1 || CastInt || codP3 || menorIg
                    sino
                        codPh1 || codP3 || menorIg
                otro:
                    codPh1 || codP3 ||  menorIg
mayor-ig
        case (tipo1h)
                float:
                    si (tipo3 = float)
                        codPh1 || codP3 || mayorIg
                    sino
                        codPh1 || codP3 || CastFloat || mayorIg
                entero:
                    si (tipo3 = float)
                        codPh1 || CastFloat || codP3 || mayorIg
                    sino si (tipo3 = natural)
                        codPh1 || codP3 || CastInt || mayorIg
                    sino
                        codPh1 || codP3 || mayorIg
                natural:
                    si (tipo3 = float)
                        codPh1 || CastFloat || codP3 || mayorIg
                    sino si (tipo3 = entero)
                        codPh1 || CastInt || codP3 || mayorIg
                    sino
                        codPh1 || codP3 || mayorIg
                otro:
                    codPh1 || codP3 ||  mayorIg
igual
```

```
                case (tipo1h)
                    float:
                        si (tipo3 = float)
                            codPh1 || codP3 || igual
                        sino
                            codPh1 || codP3 || CastFloat || igual
                    entero:
                        si (tipo3 = float)
                            codPh1 || CastFloat || codP3 || igual
                        sino si (tipo3 = natural)
                            codPh1 || codP3 || CastInt || igual
                        sino
                            codPh1 || codP3 || igual
                    natural:
                        si (tipo3 = float)
                            codPh1 || CastFloat || codP3 || igual
                        sino si (tipo3 = entero)
                            codPh1 || CastInt || codP3 || igual
                        sino
                            codPh1 || codP3 || igual
                    otro:
                        codPh1 || codP3 ||  igual
            no-igual
                case (tipo1h)
                    float:
                        si (tipo3 = float)
                            codPh1 || codP3 || no-igual
                        sino
                            codPh1 || codP3 || CastFloat || no-igual
                    entero:
                        si (tipo3 = float)
                            codPh1 || CastFloat || codP3 || no-igual
                        sino si (tipo3 = natural)
                            codPh1 || codP3 || CastInt || no-igual
                        sino
                            codPh1 || codP3 || no-igual
                    natural:
                        si (tipo3 = float)
                            codPh1 || CastFloat || codP3 || no-igual
                        sino si (tipo3 = entero)
                            codPh1 || CastInt || codP3 || no-igual
                        sino
                            codPh1 || codP3 || no-igual
                    otro:
                        codPh1 || codP3 ||  no-igual

codJ1 ←
    case (op)
        menor:
            si (tipo1h = float)
                si (tipo3 = float)
                    codJh1 ||
                    codJ3 ||
                    fcmpg ||
                    if_ge +7
                    iconst_1
                    goto +4
                    iconst_0
                sino
                    codJh1 ||
```

```
                codJ3 ||
                i2f ||
                fcmpg ||
                if_ge +7
                iconst_1
                goto +4
                iconst_0
        sino
            si(tipo3 = float)
                codJh1 ||
                i2f ||
                codJ3 ||
                fcmpg ||
                if_ge +7
                iconst_1
                goto +4
                iconst_0
            sino
                codJh1 ||
                codJ3 ||
                if_icmpge +7
                iconst_1
                goto +4
                iconst_0
mayor
    si (tipo1h = float)
        si (tipo3 = float)
            codJh1 ||
            codJ3 ||
            fcmpg ||
            if_le +7
            iconst_1
            goto +4
            iconst_0
        sino
            codJh1 ||
            codJ3 ||
            i2f ||
            fcmpg ||
            if_le +7
            iconst_1
            goto +4
            iconst_0
    sino
        si(tipo3 = float)
            codJh1 ||
            i2f ||
            codJ3 ||
            fcmpg ||
            if_le +7
            iconst_1
            goto +4
            iconst_0
        sino
            codJh1 ||
            codJ3 ||
            if_icmple +7
            iconst_1
            goto +4
            iconst_0
```

```
menor-ig
    si (tipo1h = float)
        si (tipo3 = float)
            codJh1 ||
            codJ3 ||
            fcmpg ||
            if_gt +7
            iconst_1
            goto +4
            iconst_0
        sino
            codJh1 ||
            codJ3 ||
            i2f ||
            fcmpg ||
            if_gt +7
            iconst_1
            goto +4
            iconst_0
    sino
        si(tipo3 = float)
            codJh1 ||
            i2f ||
            codJ3 ||
            fcmpg ||
            if_gt +7
            iconst_1
            goto +4
            iconst_0
        sino
            codJh1 ||
            codJ3 ||
            if_icmpgt +7
            iconst_1
            goto +4
            iconst_0
mayor-ig
    si (tipo1h = float)
        si (tipo3 = float)
            codJh1 ||
            codJ3 ||
            fcmpg ||
            if_lt +7
            iconst_1
            goto +4
            iconst_0
        sino
            codJh1 ||
            codJ3 ||
            i2f ||
            fcmpg ||
            if_lt +7
            iconst_1
            goto +4
            iconst_0
    sino
        si(tipo3 = float)
            codJh1 ||
            i2f ||
            codJ3 ||
```

```
                    fcmpg ||
                    if_lt +7
                    iconst_1
                    goto +4
                    iconst_0
                sino
                    codJh1 ||
                    codJ3 ||
                    if_icmplt +7
                    iconst_1
                    goto +4
                    iconst_0
igual
        si (tipo1h = float)
                si (tipo3 = float)
                    codJh1 ||
                    codJ3 ||
                    fcmpg ||
                    if_ne +7
                    iconst_1
                    goto +4
                    iconst_0
                sino
                    codJh1 ||
                    codJ3 ||
                    i2f ||
                    fcmpg ||
                    if_ne +7
                    iconst_1
                    goto +4
                    iconst_0
        sino
                si(tipo3 = float)
                    codJh1 ||
                    i2f ||
                    codJ3 ||
                    fcmpg ||
                    if_ne +7 ||
                    iconst_1 ||
                    goto +4 ||
                    iconst_0
                sino
                    codJh1 ||
                    codJ3 ||
                    if_icmpne +7 ||
                    iconst_1 ||
                    goto +4 ||
                    iconst_0
no-igual
        si (tipo1h = float)
                si (tipo3 = float)
                    codJh1 ||
                    codJ3 ||
                    fcmpg ||
                    if_eq +7 ||
                    iconst_1 ||
                    goto +4 ||
                    iconst_0
                sino
                    codJh1 ||
```

codJ3 ||
i2f ||
fcmpg ||
if_eq +7 ||
iconst_1 ||
goto +4 ||
iconst_0
sino
si(tipo3 = float)
codJh1 ||
i2f ||
codJ3 ||
fcmpg ||
if_eq +7 ||
iconst_1 ||
goto +4 ||
iconst_0
sino
codJh1 ||
codJ3 ||
if_icmpeq +7 ||
iconst_1 ||
goto +4 ||
iconst_0
}

ExpresiónNiv1(out: tipo1, codP1, codJ1) →
ExpresiónNiv2(out: tipo2, codP2, codJ2)
{
tipoh3 ← tipo2
codPh3 ← codP2
codJh3 ← codJ2
}
ExpresiónNiv1Rec(in: tipoh3, codPh3, codJh3; out: tipo3, codP3,codJ3)
{
tipo1 ← tipo3
codP1 ← codP3
codJ1 ← codJ3
}

ExpresiónNiv1Rec(in: tipoh1, codPh1, codJh1; out: tipo1, codP1,codJ1) → λ
{
tipo1 ← tipoh1
codP1 ← codPh1
codJ1 ← codJh1
}
ExpresiónNiv1Rec(in: tipoh1, codPh1, codJh1; out: tipo1, codP1,codJ1) →
OpNiv1 (out: op2)
ExpresiónNiv2 (out: tipo3, codP3, codJ3)
{
tipoh4 =
si (tipoh1 = error v tipo3 = error v
tipoh1 = char v tipo3 = char v
(tipoh1 = boolean ∧ tipo3 =/= boolean) v
(tipoh1 =/= boolean ∧ tipo3 = boolean))
error
sino case (op2)
suma,resta:
si (tipoh1=float v tipo3 = float)
float

```
                sino si (tipoh1 =integer v tipo3 = integer)
                 integer
                sino si (tipoh1 =natural ∧ tipo3 = natural)
                 natural
                      sino error
          o:
                si (tipoh1 = boolean ∧ tipo3 = boolean)
                   boolean
                sino error

codPh4 ←  case (op2)
          suma:
               case (tipoh1)
                    float:
                         si(tipo3 = float)
                              codPh1 || codP3 || sumar
                         sino
                              codPh1 || codP3 || CastFloat || sumar
                    entero:
                         si (tipo3 = float)
                              codPh1 || CastFloat || codP3 || sumar
                         sino si (tipo3 = natural)
                              codPh1 || codP3 || CastInt || sumar
                         sino
                              codPh1 || codP3 || sumar
                    natural:
                         si (tipo3 = float)
                              codPh1 || CastFloat || codP3 || sumar
                         sino si (tipo3 = entero)
                              codPh1 || CastInt || codP3 ||  sumar
                         sino
                              codPh1 || codP3 || sumar
          resta:
               case (tipoh1)
                    float:
                         si(ExpresionNiv2 .tipo = float)
                              codPh1 || codP3 || restar
                         sino
                              codPh1 || codP3 || CastFloat || restar
                    entero:
                         si (tipo3 = float)
                              codPh1 || CastFloat || codP3 || restar
                         sino si (tipo3 = natural)
                              codPh1 || codP3 || CastInt || restar
                         sino
                              codPh1 || codP3 || restar
                    natural:
                         si (tipo3 = float)
                              codPh1 || CastFloat || codP3 || restar
                         sino si (tipo3 = entero)
                              codPh1 || CastInt || codP3 ||  restar
                         sino
                              codPh1 || codP3 || restar
          o:
               codPh1 || codP3 || o

codJh4 =
     case (op2)
          suma:
               si (tipoh1 = float)
```

si(tipo3 = float)
codJh1 || codJ3 || fadd
sino
codJh1 || codJ3 || i2f || fadd
sino
si(tipo3 = float)
codJh1 || i2f || codJ3 || fadd
sino
codJh1 || codJ3 || iadd
resta:
si (tipoh1 = float)
si(tipo3 = float)
codJh1 || codJ3 || fsub
sino
codJh1 || codJ3 || i2f || fsub
sino
si(tipo3 = float)
codJh1 || i2f || codJ3 || fsub
sino
codJh1 || codJ3 || isub
o:
codJh1 ||
ifne +7 ||
codJ3 ||
ifeq +7 ||
iconst_1 ||
goto +4 ||
iconst_0
}
ExpresiónNiv1Rec(in: tipoh4, codPh4, codJh4; out: tipo4, codP4,codJ4)
{
tipo1 ← tipo4
codP1 ← codP4
codJ1 ← codJ4
}

ExpresiónNiv2(out: tipo1, codP1, codJ1) →
ExpresiónNiv3(out: tipo2, codP2, codJ2)
{
tipoh3 ← tipo2
codPh3 ← codP2
codJh3 ← codJ2
}
ExpresiónNiv2Rec(in: tipoh3, codPh3, codJh3; out: tipo3, codP3, codJ3)
{
tipo1 ← tipo3
codP1 ← codP3
codJ1 ← codJ3
}

ExpresiónNiv2Rec(in: tipoh1, codPh1, codJh1; out: tipo1, codP1, codJ1) → λ
{
tipo1 ← tipoh1
codP1 ← codPh1
codJ1 ← codJh1
}

ExpresiónNiv2Rec(in: tipoh1, codPh1, codJh1; out: tipo1, codP1, codJ1) →
OpNiv2(out: op2)

```
ExpresiónNiv3(out: tipo3, codP3, codJ3)
{
tipoh4 ←
        si (tipoh1 = error v tipo3 = error v
            tipoh1 = character v tipo3 = character v
           (tipoh1 = boolean ∧ tipo3 =/= boolean v
           (tipoh1 =/= boolean ∧ tipo3 = boolean))
                error
        sino case (op2)
                multiplica,divide:
                        si (tipoh1=float v tipo3 = float)
                           float
                        sino si (tipoh1 =integer v tipo3 = integer)
                                integer
                        sino si (tipoh1 =natural ∧ tipo3=natural)
                          natural
                       sino error
                modulo:
                        si (tipo3 = natural ∧
                        (tipoh1=natural v tipoh1=integer))
                             tipoh1
                        sino     error
                y:
                        si (tipoh1 = boolean ∧ tipo3 = boolean)
                           boolean
                        sino error

codPh4 ←
        case(op2)
                Multiplica:
                        case (tipoh1)
                                float:
                                        si(tipo3 = float)
                                                codPh1 || codP3 || Mul
                                        sino
                                                codPh1 || codP3 || CastFloat || Mul
                                entero:
                                        si (tipo3 = float)
                                                codPh1 || CastFloat || codP3 || Mul
                                        sino si (tipo3 = natural)
                                                codPh1 || codP3 || CastInt || Mul
                                        sino
                                                codPh1 || codP3 || Mul
                                natural:
                                        si (tipo3 = float)
                                                codPh1 || CastFloat || codP3 || Mul
                                        sino si (tipo3 = entero)
                                                codPh1 || CastInt || codP3 ||  Mul
                                        sino
                                                codPh1 || codP3 || Mul
                Divide:
                        case (tipoh1)
                                float:
                                        si(tipo3 = float)
                                                codPh1 || codP3 || Div
                                        sino
                                                codPh1 || codP3 || CastFloat || Div
                                entero:
                                        si (tipo3 = float)
                                                codPh1 || CastFloat || codP3 || Div
```

```
                        sino si (tipo3 = natural)
                            codPh1 || codP3 || CastInt || Div
                        sino
                            codPh1 || codP3 || Div
                    natural:
                        si (tipo3 = float)
                            codPh1 || CastFloat || codP3 || Div
                        sino si (tipo3 = entero)
                            codPh1 || CastInt || codP3 ||  Div
                        sino
                            codPh1 || codP3 || Div
            Modulo:
                codPh1 || codP3 || Mod
            y:
                codPh1 || codP3 || Y

codJh4 ←
    case(op2)
            Multiplica:
                si(tipoh1 = float)
                    si(tipo3 = float)
                        codJh1 || codJ3 || fmul
                    sino
                        codJh1 || codJ3 || i2f || fmul
                sino
                    si(tipo3 = float)
                        codJh1 || i2f || codJ3|| fmul
                    sino
                        codJh1 || codJ3|| imul
            Divide:
                si(tipoh1 = float)
                    si(tipo3 = float)
                        codJh1 || codJ3 || fdiv
                    sino
                        codJh1 || codJ3 || i2f || fdiv
                sino
                    si(tipo3 = float)
                        codJh1 || i2f || codJ3|| fdiv
                    sino
                        codJh1 || codJ3|| idiv
            Modulo:
                codJh1 || codJ3 || imod
            y:
                codJh1 ||
                ifeq +11 ||
                codJ3 ||
                ifeq +7 ||
                iconst_1 ||
                goto +4 ||
                iconst_0
}
ExpresiónNiv2Rec(in: tipoh4, codPh4, codJh4; out: tipo4, codP4, codJ4)
{
tipo1 ← tipo4
codP1 ← codP4
codJ1 ← codJ4
}
```

```
ExpresiónNiv3(out: tipo1, codP1, codJ1) →
    ExpresiónNiv4( out: tipo2, codP2, codJ2)
    {
    tipoh3 ← tipo2
    codPh3 ← codP2
    codJh3 ← codJ2
    }
    ExpresiónNiv3Fact(in: tipoh3, codPh3, codJh3; out: tipo3, codP3, codJ3)
    {
    tipo1 ← tipo3
    codP1 ← codP3
    codJ1 ← codJ3
    }
ExpresiónNiv3Fact(in: tipoh1, codPh1, codJh1; out: tipo1, codP1, codJ1) → λ
    {
    tipo1 ← tipoh1
    codP1 ← codPh1
    codJ1 ← codJh1
    }

ExpresiónNiv3Fact(in: tipoh1, codPh1, codJh1; out: tipo1, codP1, codJ1) →
    OpNiv3(out: op2)
    ExpresiónNiv3(out: tipo3, codP3, codJ3)
    {
    tipo1 ←
        si (tipoh1 = error v tipo3 = error v
                tipoh1 =/= natural v tipo3 =/= natural)
            error
        sino natural

    codP1 ←
        case (op2)
            shl:
                codPh1 || codP3 || shl
            shr:
                codPh1 || codP3 || shr

    codJ1 ←
        case (op2)
            shl:
                codJh1 || codJ3 || ishl
            shr:
                codJh1 || codJ3 || ishr
    }

ExpresiónNiv4(out: tipo1, codP1, codJ1) →
    OpNiv4(out: op2)
    ExpresiónNiv4(out tipo3, codP3, codJ3)
    {
    tipo1 ←
        si (tipo3 = error)
            error
        sino case (op2)
            no:
                si (tipo3=boolean)
                  boolean
                sino   error
            menos:
                si (tipo3=float)
                  float
```

```
                sino si (tipo3=integer v tipo3 = natural)
                        integer
                  sino error
        cast-float:
                si (tipo3=/=boolean)
                   float
                sino error
        cast-int:
                si (tipo3=/=boolean)
                   integer
                sino error
        cast-nat:
                si (tipo3=natural v tipo3=character)
                  natural
                sino error
        cast-char:
                si (tipo3=natural v tipo3=character)
                  character
                sino error


codP1 ←
    case (op2)
        no:
            codP3 || no
        negativo:
            codP3 || negativo
        cast-float:
            codP3 || CastFloat
        cast-int:
            codP3 || CastInt
        cast-nat:
            codP3 || CastNat
        cast-char:
            codP3 || CastChar

codJ1 ←
    case (op2)
        no:
            codJ3||
            ifeq +7 ||
            iconst_0 ||
            goto +4 ||
            iconst_1
        negativo:
            case (tipo3)
                entero:
                    codJ3 || ineg
                float:
                    codJ3 || fneg
        cast-float:
            case (tipo3)
                character:
                natural:
                entero:
                    codJ3 || i2f
                float:
                    codJ3
        cast-int:
            case (tipo3)
```

```
                    character:
                    natural:
                    entero:
                        codJ3
                    float:
                        codJ3 || f2i
            cast-nat:
                case (tipo3)
                    character:
                    natural:
                    entero:
                        codJ3
                    float:
                        codJ3 || f2i
            cast-char:
                case (tipo3)
                    character:
                    natural:
                    entero:
                        codJ3
                    float:
                        codJ3 || f2i
    }

ExpresiónNiv4(out: tipo1, codP1, codJ1) →
    barraVertical()
    Expresión(out: tipo2, codP2, codJ2)
    barraVertical()
    {
    tipo1 ←
        si (tipo2 = error v tipo2=boolean v tipo2=character)
            error
        sino si (tipo2 = float)
            float
        sino si (tipo2 = natural v tipo2 = integer)
            natural
        sino error


    codP1 ← codP2 || abs
    codJ1 ←
        case (tipo2)
            float:
                codJ2 ||
                dup ||
                fconst_0 ||
                fcmpg ||
                ifge +4
                fneg
            otro:
                codJ2 ||
                dup ||
                ifge +4
                fneg
    }

ExpresiónNiv4(out: tipo1, codP1, codJ1) →
    abrePar()
    Expresión(out: tipo2, codP2, codJ2)
    cierraPar()
```

{
tipo1 ← tipo2
codP1 ← codP2
codJ1 ← codJ2
}

ExpresiónNiv4(out: tipo1, codP1, codJ1) →
Literal(out: tipo2, codP2, codJ2)
{
tipo1 ← tipo2
codP1 ← codP2
codJ1 ← codJ2
}

Literal(out: tipo1, codP1, codJ1) → id
{
tipo1 ← ts[id.lex].tipo
dir ← ts[id.lex].dir
codP1 ← apila-dir dir
codJ1 ← case(ts[id.lex].tipo)
boolean:
i2b || iload dir
tCha:
i2c || iload dir
natural:
entero:
iload dir
float:
fload dir
}
Literal(out: tipo1, codP1, codJ1) → litNat
{
tipo1 ← natural
codP1 ← apila getValor(litNat.lex)
codJ1 ← iconst getValor(litNat.lex))
}
Literal(out: tipo1, codP1, codJ1) → litFlo
{
tipo1 ← float
codP1 ← apila getValor(litFlo.lex)
codJ1 ← fconst getValor(litFlo.lex))
}
Literal → litTrue
{
Literal.tipo ← boolean
Literal.codP ← apila true
Literal.codJ ← iconst 1
}
Literal → litFalse
{
Literal.tipo ← boolean
Literal.codP ← apila false
Literal.codJ ← iconst 0
}
Literal(out: tipo1, codP1, codJ1) → litCha
{
tipo1 ← character
codP1 ← apila getValor(litCha.lex)
codJ1 ← iconst getValor(litCha.lex))
}

OpNiv0(out: op) → **<**
{op = menor}
OpNiv0(out: op) → **>**
{op = mayor}
OpNiv0(out: op) → **<=**
{op = menor-ig}
OpNiv0(out: op) → **>=**
{op = mayor-ig}
OpNiv0(out: op) → **=**
{op = igual}
OpNiv0(out: op) → **=/=**
{op = no-igual}
OpNiv1(out: op) → **+**
{op = suma}
OpNiv1(out: op) → **-**
{op = resta}
OpNiv1(out: op) → **or**
{op = o}
OpNiv2(out: op) → ***
{op = multiplica}
OpNiv2(out: op) → **/**
{op = divide}
OpNiv2(out: op) → **%**
{op = modulo}
OpNiv2(out: op) → **and**
{op = y}
OpNiv3(out: op) → **>>**
{op = shl}
OpNiv3(out: op) → **<<**
{op = shr}
OpNiv4(out: op) → **not**
{op = no}
OpNiv4(out: op) → **-**
{op = menos}
OpNiv4(out: op) → **(float)**
{op = cast-float}
OpNiv4(out: op) → **(int)**
{op = cast-int}
OpNiv4(out: op) → **(nat)**
{op = cast-nat}
OpNiv4(out: op) → **(char)**
{op = cast-char}

10. Formato de representación del código P_

El código P se representa en binario. Cada instrucción de la máquina virtual se representa con un código (1 byte) seguido, si procede, del tipo de argumento (1 byte) y del valor en binario del argumento (4 bytes para números, 1 byte para char y booleanos). Se adjunta la tabla de códigos de operaciones de la máquina virtual como apéndice.

Cualquier máquina virtual de java ha de ser capaz de ejecutar archivos en el llamado “formato .class”, que representado como una estructura C equivaldría a lo siguiente:

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
     u2 constant_pool_count;
     cp_info constant_pool[constant_pool_count-1];
     u2 access_flags;
     u2 this_class;
     u2 super_class;
     u2 interfaces_count;
     u2 interfaces[interfaces_count];
     u2 fields_count;
     field_info fields[fields_count];
     u2 methods_count;
     method_info methods[methods_count];
     u2 attributes_count;
     attribute_info attributes[attributes_count];
}
```

Donde u1, u2 y u4 corresponden a una estructura de 1, 2 y 4 bytes respectivamente.
Lo siguiente es una pequeña descripción del significado de cada campo, en la especificación formal suministrada por Sun se puede encontrarse esta misma información mucho más detallada.

· u4 magic: este campo siempre tiene el valor hexadecimal 0xCAFEBABE.

· u2 minor_version, u2 major_version: estos campos especifican la versión de la jvm para la que fue compilador el .class. Si la versión del intérprete es menor detendrá la ejecución. Nosotros usaremos la versión 1.2.

· u2 constant_pool_count, interfaces_count, fields_count, methods_count y attributes_count: son el número de entradas de la constant_pool, interfaces, fields, methods y attributes respectivamente.

· cp_info constant_pool[]: Se trata de un array (hablando en términos de C) con una serie de información a la que luego se referirán otros campos. Su importancia en el .class es vital, pues entre otras cosas almacena:

  · La información de las clases heredadas.

  · La información (métodos, campos, etc) de las clases heredadas.

  · Cierta información sobre los campos y métodos definidos en la clase en cuestión, como son el tipo y nombre de los campos y la firma de los métodos.

  · Cierta información sobre los métodos de otras clases que se invoquen desde los métodos de esta.

· u2 access_flags: según estén activados o no una serie de bits se fija la visibilidad de la clase en cuestión (public, private, etc), así como otras características (por ejemplo, si es una interfaz)..

· u2 this_class: es la dirección dentro de constant pool donde se encuentra la CONSTANT_Class_info de esta clase. CONSTANT_Class_info es una estructura que representa el nombre de la clase, incluyendo el paquete a la que pertenece.

· u2 super_class: al igual que this_class es una referencia a un CONSTANT_Class_info de la constant pool. Sin embargo en este caso representa la información de la clase de la que hereda la clase que define este .class.

· u2 interfaces[], field_info fields[] y attribute_info attributes[]: son arrays que contienen información sobre las interfaces implementadas, los campos que declara la clase y los atributos que tiene. En esta práctica no se usarán.

· method_info methods[]: es un array de method_info que contiene, como su nombre indica, la información de los métodos definidos en esta clase. Methods_info consiste en lo siguiente:

```
method_info {
```

```
        u2 access_flags;
        u2 name_index;
        u2 descriptor_index;
        u2 attributes_count;
         attribute_info attributes[attributes_count];
    }
```

donde:

· u2 access_flags: tiene un significado que a nuestro nivel podemos considerar similar al access_flags de la clase.

· u2 name_index: es la dirección dentro de constant pool del nombre el metodo.

· u2 descriptor_index: es la dirección dentro de constant pool de un string que define la firma del método (es decir, los tipos de los argumentos que recibe y el tipo devuelto).

· u2 attributes_count: es el número de entradas de attributes.

· attribute_info attributes[]: es un array de atributos, de los cuales únicamente usaremos el de código, que a su vez consiste en una estructura como la que sigue:

```
    Code_attribute {
        u2 attribute_name_index;
        u4 attribute_length;
        u2 max_stack;
        u2 max_locals;
        u4 code_length;
        u1 code[code_length];
        u2 exception_table_length;
        {       u2 start_pc;
                u2 end_pc;
                u2  handler_pc;
                u2  catch_type;
        }       exception_table[exception_table_length];
         u2 attributes_count;
         attribute_info attributes[attributes_count];
    }
```

siendo:

· u2 attribute_name_index: el indice dentro de constant pool de una cadena “Code”

· u4 attribute_length: la longitud en bytes del resto del registro (sin contar attribute_name_index ni attribute_length)

· u2 max_stack: la profundidad máxima de la pila asociada al método.

· u2 max_locals: el número máximo variables locales y parámetros usados por el metodo. En esta cuenta los longs y doubles cuentan doble (aunque no se usan en la práctica).

· u4 code_length: la longitu del array code.

· u1 code[]: un array de bytes en el que se almacenan los códigos de cada instrucción y sus argumentos en el caso de tenerlos. A diferencia del codigo P propuesto, las instrucciones que requieren argumentos solo aceptan un tipo de argumento, por lo que la instrucción y el dato pueden almacenarse de manera consecutiva (en el codigo P entre una instrucción con argumento como apila y el valor de dicho argumento se escribe también un byte que indica el tipo del argumento).

· exception_table y attributes: son dos arrays que no se usarán, por lo que exception_table_length y attributes_count valdrán siempre 0.

11 Notas sobre la implementación

Descripción de la implementación realizada.

11.1. Descripción de archivos

Enumeración de los archivos con el código fuente de la implementación, y descripción de lo que contiene cada archivo.

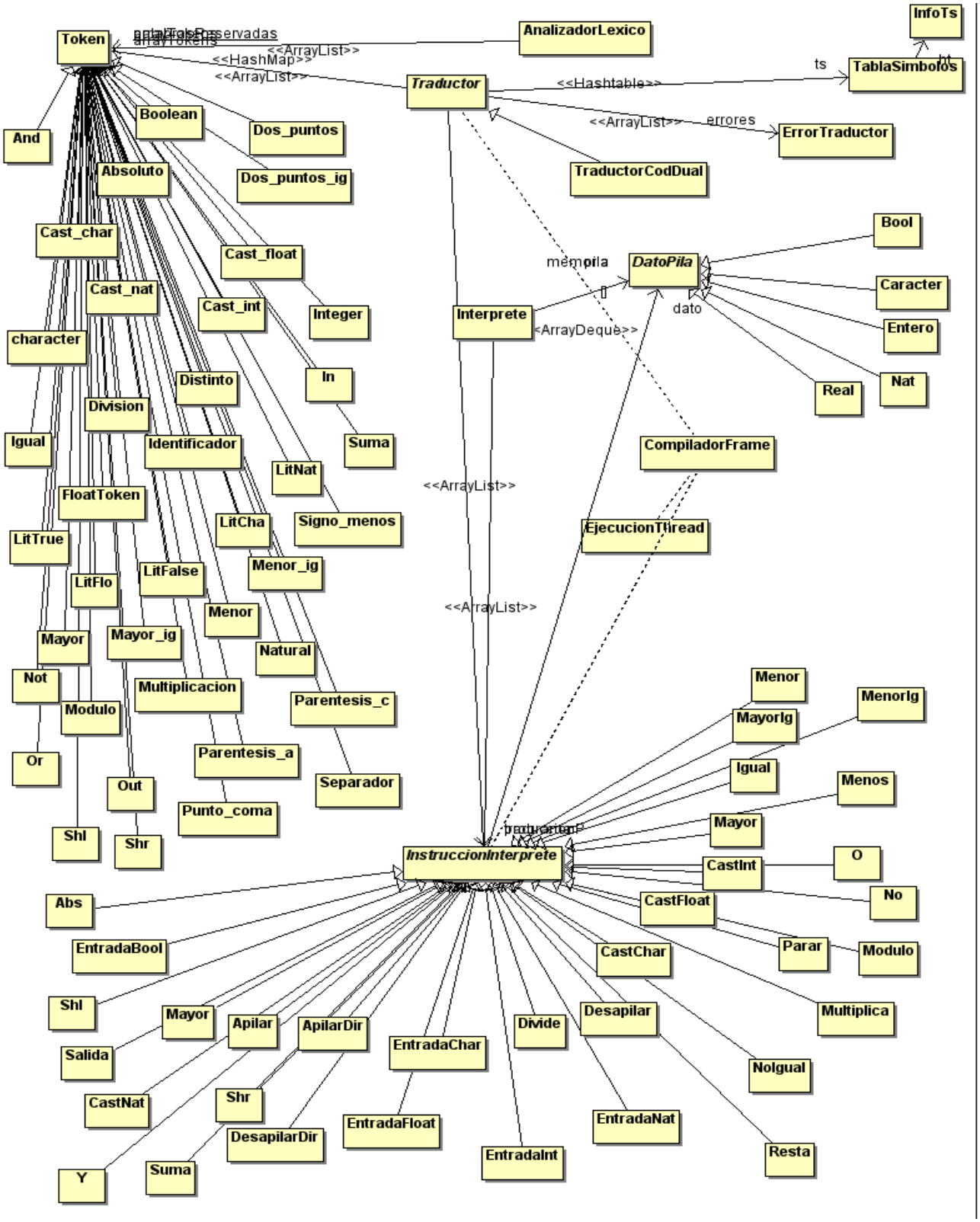
Hemos divido el proyecto es 4 paquetes principales:

- Compilador: Contiene las clases relativas al traductor de lenguaje. Este paquete contiene 3 paquetes:
  - Lexico: contiene la clase AnalizadorLexico, que recibe un flujo de entrada y devuelve un ArrayList de Tokens. Este paquete contiene un paquete Tokens con clases que corresponden a cada uno de los tipos de tokens que maneja el Analizador Lexico.
- TablaSimbolos: Contiene una clase que gestionará las distintas operaciones de la tabla de símbolos. También contiene una clase auxiliar que gestiona los datos que se asignan a cada identificador, en nuestro caso el tipo y la dirección de memoria de cada variable.
- Traductor: Contiene las clases necesarias para traducir un ArrayList de Tokens procedente del analizador léxico a un ArrayList de Objetos que contendrá el código binario. Como hemos implementado dos tipos de traducción (a código P y a código J) tenemos unas clases comunes a ambos que son ErrorTraductor (captura los errores del traductor), Traductor y TraductorCodDual que gestionan la traducción.
  - Traductor Codigo P: El código binario generado será ejecutado en nuestro intérprete Pila. La clase del codigo P es Codigo.java.
- Traductor Codigo J: Contiene las clases necesarias para traducir un ArrayList de Tokens procedente del analizador léxico a un ArrayList de Objetos que contendrá el código binario que será ejecutado en la Máquina Virtual de Java. La clase del codigo J es CodigoJVM
- Interfaz: Contiene las interfaces utilizadas de cara al usuario. Como hemos implementado dos programas separados (compilador e intérprete) tenemos dos interfaces:
  - Compilador: Contiene la interfaz del compilador. Esta interfaz da la opción de introducir el código en la propia interfaz o cargarla desde un fichero, una vez cargado puedes compilar y ver el código pila o ejecutar (en cuyo caso compilará y luego ejecutará el programa) También nos ofrece ejecutar el código en modo Traza (mostrando el contenido de la pila y la memoria en cada instrucción además de las entradas/salidas del programa) o en modo Normal (mostrando unicamente las entradas/salidas del programa)
- Pila: Contiene una interfaz que hemos utilizado para probar el intérprete a pila. Este panel hace de intermediario entre el bytecode del lenguaje a pila y el explicado en clase (con sentencias alfanumericas como "apila 3" o "suma"). Al decompilar un archivo en bytecode este se mostrará como cadenas alfanumericas. Al compilar, el texto escrito sera traducido a lenguaje de pila, siempre y cuando su sintáxis sea correcta
- org: Contiene las clases de la librería BCEL (http://jakarta.apache.org/bcel/) que se utilizan para generar el código Java y que la Máquina Virtual de Java sea capaz de interpretarlo.
- Pila: Contiene el intérprete encargado de simular la ejecución del código . Como tenemos dos tipos de código (código P y código J) este paquete contiene dos paquetes:
  - Intérprete: Se encarga de ejecutar el código P generado por el compilador de código P. Contiene 3 paquetes que se encargan de gestionar tanto los datos, como las instrucciones y excepciones que pueden surgir en el código y 3 clases que son las principales (EscritorPila, Interprete y LectorPila) que gestionan la entrada/salida de la ejecución y la propia ejecución.
- jvm: Se encarga de generar un .class entendible por la Máquina Virtual de Java. Para ello contiene un paquete instrucciones y una clase instrucción que codifican las distintas instrucciones del código

y una clase ClassConstructor que lleva la carga de
la generación del .class.

11.2. Otras notas_

Diagramas de clase UML describiendo la arquitectura del sistema.



Conclusiones

Qué se ha conseguido y qué se ha dejado pendiente para más adelante..

Referencias bibliográficas

Libros, artículos y otras fuentes de información utilizadas (por ejemplo páginas web).

http://jakarta.apache.org/bcel/

Apéndices

Codificación de las operaciones del intérprete de pila:

Cod	Operación	args
0	**parar**	
1	**apilar**	
2	**apilar-dir**	
3	**desapilar**	
4	**desapilar-dir**	
5	**menor**	
6	**mayor**	
7	**menor-ig**	
8	**mayor-ig**	
9	**igual**	
10	**no-igual**	
11	**suma**	
12	**resta**	
13	**multiplica**	
14	**divide**	
15	**modulo**	
16	**y**	
17	**o**	
18	**no**	
19	**menos**	
20	**shl**	
21	**shr**	
22	**cast-int**	
23	**cast-char**	
24	**cast-float**	
25	**cast-nat**	
26	**Abs**	
27	**Salida**	
28	**Entrada-bool**	
29	**Entrada-char**	
30	**Entrada-float**	
31	**Entrada-int**	
32	**Entrada-nat**	

Tipo	arg	Cod Tamaño (bytes)
error/desc	0	?
boolean	1	1
character	2	1
natural	8	4
integer	9	4
float	10	4