Práctica de Procesadores de Lenguaje

Nombre:
Alberto Ureña Herradón

Requisitos

El lenguaje a procesar

•Los programas constarán de una *sección de declaraciones*, seguida de una *sección de acciones*.
•La sección de declaraciones puede estar vacía. En caso de aparecer, constará de una secuencia de una o más *declaraciones*.
•Cada declaración constará de un nombre de *tipo* seguido de una *variable*, seguida de `;`.
•Los identificadores de las variables comienzan necesariamente por una letra o por `_`. A continuación puede aparecer cero o más letras, dígitos o `_`.
•Las mayúsculas y minúsculas se consideran indistinguibles (así, por ejemplo, `medusa`, `MEDUSA`, `mEdUsa` y `MeduSA` representan el mismo identificador)
•En la sección de declaraciones no podrá haber variables duplicadas
•El nombre de tipo podrá ser `int` (números enteros) y `real` (números reales). En los nombres de tipo tampoco se distingue entre mayúsculas y minúsculas (así, por ejemplo, las cadenas `int`, `InT`, `INT` representan todas ellas el nombre de tipo `int`)
•La sección de acciones constará de una secuencia de una o más *acciones*.
•Una acción es una *expresión* seguida de `;` (NOTA: obsérvese que en este lenguaje, el `;` actúa como terminador de declaraciones y acciones, y no como separador)
•Las expresiones usarán los siguientes operadores: `in`, `out`, `=`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `||`, `+`, `-`(resta), `*`, `/`, `%`, `&&`, `-`(cambio de signo), `!`, `(int)`, `(real)`
•El operando de `in` ha de ser una variable. Esta variable ha debido ser previamente declarada.
•El primer operando de `=` ha de ser una variable.
•El segundo operando de =, así como los operandos del resto de los operadores pueden ser expresiones arbitrarias, siempre y cuando se respeten las reglas de prioridad.
•Los operadores obedecen a las siguientes reglas de prioridad y asociatividad:
•El menor nivel de prioridad (nivel 0) es el de los operadores de *lectura* (`in`) y *escritura* (`out`)
-Ambos son operadores unarios, prefijos, no asociativos.
•El siguiente nivel de prioridad (nivel 1) es el *operador de asignación*: =
-Es un operador binario, infijo, que asocia a derechas.
•El siguiente nivel de prioridad (nivel 2) es el de los *operadores de comparación*: < (menor), > (mayor), <= (menor o igual), >= (mayor o igual), == (igual), != (distinto):
-Todos ellos son binarios, infijos, y no asociativos.
•El siguiente nivel de prioridad (nivel 3) es el de los *operadores aditivos*: || (o lógico), + (suma), - (resta):
-Todos ellos son binarios, infijos, y asocian a izquierdas.
•El siguiente nivel de prioridad (nivel 4) es el de los *operadores multiplicativos*: * (multiplicación), / (división), % (módulo), && (y lógico):
-Todos ellos son binarios, infijos, y asocian a izquierdas.
•El nivel de prioridad más alto (nivel 5) es el de los *operadores unarios*: - (cambio de signo), ! (negación lógica), `(int)` (conversión a entero), `(real)` (conversión a real):
`--` y `!` son operadores unarios, prefijos, y asociativos.
- `(int)` y `(real)` son operadores unarios, prefijos, y no asociativos.
•Reglas de tipo:
•Operadores de lectura / escritura (`in`, `out`):
-El operando puede ser de tipo entero o real.

-El tipo del resultado coincidirá con el tipo del operando.
•Operadores de comparación (<, >, <=, >=, ==, !=):
-El primer operando puede ser de tipo entero o real.
-El segundo operando puede ser de tipo entero o real.
-El resultado será de tipo entero.
•Operadores aritméticos binarios (+, - binario, *, /):
-Si ambos operandos son de tipo entero, el resultado es entero.
-Si ambos operandos son de tipo real, el resultado es real.
-Si uno de los operandos es de tipo entero y el otro es de tipo real, el resultado es real.
•Operadores lógicos binarios (||, &&):
-El tipo de ambos operandos ha de ser entero.
-El tipo del resultado será entero.
•Módulo (%):
-El tipo de ambos operandos ha de ser entero.
-El tipo del resultado será entero.
•Negación lógica (!):
-El tipo del operando ha de ser entero.
-El tipo del resultado será entero.
• Cambio de signo (- unario):
-El tipo del operando puede ser entero o real.
-El tipo del resultado será el del operando.
•Conversión a entero (`(int)`):
-El tipo del operando puede ser entero o real.
-El resultado será entero.
•Conversión a real (`(real)`):
-El tipo del operando puede ser entero o real.
-El resultado será real.
•Asignación (=):
-Si el tipo del primer operando es real, el del segundo operando puede ser entero o real; el resultado será real.
-Si el tipo del primer operando es entero, el del segundo operando ha de ser necesariamente entero; el resultado será entero.
•Reglas de evaluación:
•Regla de *alineamiento de tipos*: Si un operador binario: (1) admite operadores de distinto tipo, (2) uno de los operandos es de tipo entero, y (3) el otro es de tipo real, entonces: el valor del operando de tipo entero se convertirá a un número real equivalente antes de llevar a cabo la operación.
•Operador de lectura: `in` *v*:
-Si *v* es de tipo entero, leer un valor entero de la entrada estándar
-Si *v* es de tipo real, leer un valor real de la entrada estándar
-Almacenar el valor leído en *v*
-El resultado de la expresión es el valor leído
•Operador de escritura: `out` *e*:
-Obtener el valor de *e*
-Escribir dicho valor por la salida estándar, seguido de un salto de línea
-El resultado de la expresión es dicho valor.
•Operadores de comparación (<, >, <=, >=, ==, !=):
-Obtener el valor del primer operando
-Obtener el valor del segundo operando
-Aplicar la regla de alineamiento de tipos

-Comparar los valores
-Si la comparación es cierta, el resultado es 1. En otro caso, el resultado es 0

- Operadores +, - binario y *:

-Obtener el valor del primer operando
-Obtener el valor del segundo operando
-Aplicar la regla de alineamiento de tipos
-Realizar la operación
-El resultado es el valor obtenido

- Operador /:

-Obtener el valor del primer operando
-Obtener el valor del segundo operando. Si es 0, terminar la ejecución con error (*división por* 0).
-Aplicar la regla de alineamiento de tipos
-Si el tipo de ambos operandos es entero, realizar la división entera. En otro caso, realizar la división real
-El resultado es el valor obtenido

- Operador %:

-Obtener el valor del primer operando
-Obtener el valor del segundo operando. Si es menor o igual que 0, terminar la ejecución con error (*segundo operando de % no positivo*).
-Encontrar el resto de la división entera del valor del primer operando entre el del segundo operando
-El resultado es el valor obtenido

- Operador ||:

-Obtener el valor del primer operando
-Obtener el valor del segundo operando
-Si alguno de los valores es distinto de 0, el resultado es 1. En otro caso, el resultado es 0

- Operador &&:

-Obtener el valor del primer operando
-Obtener el valor del segundo operando
-Si ambos valores son distintos de 0, el resultado es 1. En otro caso, el resultado es 0

- Operador !:

-Obtener el valor del operando.
-Si es distinto de 0, el resultado es 0. En otro caso, el resultado es 1.

- Cambio de signo (- unario):

-Obtener el valor del operando.
-El resultado es dicho valor, cambiado de signo

- Conversión a entero (`((int)`):

-Obtener el valor del operando.
-Para operandos de tipo entero, el resultado es el obtenido.
-Para operandos de tipo real, el resultado es la parte entera del valor obtenido.

- Conversión a real (`((real)`):

-Obtener el valor del operando.
-Para operandos de tipo real, el resultado es el obtenido.
-Para operandos de tipo entero, es el valor real equivalente.

- Asignación (=):

-Obtener el valor del segundo operando
-Si el tipo del primer operando es real y el del segundo operando es entero, alinear este último al valor real equivalente.

-Almacenar dicho valor en *v*
-El resultado de la expresión será dicho valor

- Como expresiones básicas, que podrán ser combinadas mediante los operadores anteriores, se consideran las siguientes:
- Literales *enteros positivos*. 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 entero positivo. La parte decimal está formada por el símbolo `.` seguido de uno o más dígitos, no admitiéndose ceros a la derecha. 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 entero positivo
- 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
- 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
real cantidad;
int euros;
real centimos;

out euros = (int)(in cantidad);
out centimos = cantidad – euros;
```

Escrito de forma más *clara*, el programa es equivalente a:

```
@ Programa de ejemplo 2
real cantidad;
int euros;
real centimos;

in cantidad;
euros = (int)cantidad;
out euros;
centimos = cantidad – euros;
out centimos;
```

La implementación

Deberán implementarse dos programas separados:

1.Traductor del lenguaje fuente a código de una máquina P. Este programa funcionará en modo *línea de comandos*, tomando dos argumentos: el nombre del archivo que contiene el programa fuente y el nombre del archivo donde se va a generar el código objeto. El resultado dependerá de si el programa fuente es o no correcto:

- Si es correcto, se generará el archivo con el código objeto.
- Si no es correcto, se mostrará por pantalla un listado de errores y no se generará ningún archivo.
  - Cada error estará precedido por los números de fila y de columna que identifican el punto exacto en el programa fuente en el que se ha producido el error.
  - La ejecución del programa se interrumpe en el momento en que se encuentra el primer error sintáctico. El resto de los errores (léxicos y de violación de restricciones contextuales) no interrumpen la ejecución del traductor.

2.Intérprete trivial capaz de simular el comportamiento de la máquina P. Este programa también funcionará en modo *línea de comandos*, tomando como argumento el archivo donde está el código P a ejecutar, así como un valor que indicará si la ejecución debe ser en *modo normal* o en *modo traza*. El resultado será la ejecución de dicho código:

- En *modo normal* el único resultado visible será el producido por las instrucciones específicas de lectura/escritura.
- En *modo traza* el intérprete mostrará una traza por la consola de comandos con los distintos pasos de ejecución. En cada paso el intérprete debe:

1.Mostrar el contenido de la pila de ejecución y de las celdas relevantes en la memoria de datos.
2.Esperar a que el usuario pulse una tecla para proseguir la ejecución.
3.Mostrar la instrucción a ejecutar y ejecutarla, avanzando al siguiente paso.

Deberán entregarse un conjunto de archivos conteniendo los casos de prueba utilizados en la validación de la implementación, incluyendo tanto casos correctos como incorrectos, para demostrar que los programas implementados son capaces de detectar y tratar todos los tipos de errores posibles de una forma adecuada a los requisitos mencionados anteriormente.

1. Especificación del léxico del lenguaje

Especificación formal del léxico del lenguaje utilizando definiciones regulares.

letra ≡ [a-z, A-Z]
digPos ≡ [1-9]
dig ≡ {digPos}|0
dec ≡ .{dig}*{digPos}
exp ≡ [e, E]-?{litNat}

(*) litNat ≡ 0|({digPos}{dig}*)
(*) litReal ≡ {litNat}({dec}|({dec}?{exp}))
(*) pyc ≡ ;
(*) id ≡ (_|{letra})(_|{letra}|{dig})*
(*) tInt ≡ [i, I][n, N][t, T]
(*) tReal ≡ [r, R][e, E][a, A][l, L]
(*) opIgual ≡ ==
(*) opMenor ≡ <
(*) opMayor ≡ >
(*) opMenIg ≡ <=
(*) opMayIg ≡ >=
(*) opAsig ≡ =
(*) opIn ≡ [i,I][n,N]
(*) opOut ≡ [o,O][u,U][t,T]
(*) opDistinto ≡ !=
(*) opOr ≡ \|\|
(*) opAnd ≡ &&
(*) opSuma ≡ \+
(*) opMenos ≡ \-
(*) opMult ≡ *
(*) opDiv ≡ /
(*) opMod ≡ %
(*) opNot ≡ !
(*) pa ≡ \(
(*) pc ≡ \)
(*) opCastInt {pa}{tInt}{pc}
(*) opCastReal ≡ {pa}{tReal}{pc}

(I) com ≡ @[^'\n']*'\n'
(I) esp ≡ [' ', '\t', '\n']

2. Especificación de la sintaxis del lenguaje

Especificación formal de los aspectos sintácticos del lenguaje utilizando una gramática incontextual. Dicha gramática debe representar de manera natural las prioridades y asociatividades de los operadores

Prog → Decs Accs
Decs → Dec Decs | λ
Dec → Tipo **id pyc**
Accs → Accs Acc | Acc
Acc → Expr0 **pyc**

Tipo → **tInt** | **tReal**

Expr0 → **opIn id** | **opOut** Expr1 | Expr1

Expr1 → **id** Oper1 Expr1 | Expr2
Oper1 → **opAsig**

Expr2 → Expr3 Oper2 Expr3 | Expr3
Oper2 → **opMayor** | **opMenor** | **opMayIg** | **opMenIg** | **opIgual** | **opDistinto**

Expr3 → Expr3 Oper3 Expr4 | Expr4
Oper3 → **opSuma** | **opMenos** | **opOr**

Expr4 → Expr4 Oper4 Expr5 | Expr5
Oper4 → **opMult** | **opDiv** | **opMod** | **opAnd**

Expr5 → Oper5a Expr5 | Oper5n Expr6 | Expr6
Oper5a → **opMenos** | **opNot**
Oper5n → **opCastInt** | **opCastReal**

Expr6 → **litNat** | **litReal** | **id** | **pa** Expr0 **pc**

Prioridades y asociatividades:

Símbolo	Prioridad	Asociatividad
in _	0	No asocian
out _		
_ = _	1	Asocia a derechas
_ > _	2	No asocian
_ < _		
_ >= _		
_ <= _		
_ == _		
_ != _		
_ + _	3	Asocian a izquierdas
_ - _		
_ || _		
_ * _	4	Asocian a izquierdas
_ / _		
_ % _		
_ && _		
- _	5	Asocian
! _		
(int) _		No asocian
(real) _		

3. Estructura y construcción de la tabla de símbolos

Deberán contemplarse tanto los aspectos necesarios para comprobar las restricciones contextuales, como los necesarios para llevar a cabo la traducción

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
: Crea una tabla de símbolos vacía.

añadeId(t : Ts, id : String, <tipo : {real, int}, dir : int>) : Ts
: Produce la tabla de símbolos que resulta de añadir el identificador *id* de tipo *tipo*, a la tabla *t*, en la dirección *dir*.

existeId(t : Ts, id : String) : bool
: Permite comprobar si el identificador *id* está en la tabla *t*. Devuelve cierto si está, y falso en otro caso.

[ ] (id: String) : <tipo : {real, int}, dir : int>
: Permite acceder a una entrada de la tabla de símbolos.
Por ejemplo, para acceder al tipo de la variable "id" hacemos: ts[id].tipo

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

Descripción, si procede, de las funciones semánticas adicionales utilizadas en la especificación. Para cada función debe indicarse explícitamente su cabecera, así como informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros.

Esta sección puede dejarse vacía si no se van a usar funciones semánticas adicionales.

3.2.2 Atributos semánticos

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

Decs:
: Decs.ts (sintetizado): almacena la tabla de símbolos.
Decs.dir (sintetizado): almacena el número de entradas de la tabla de símbolos.

Dec:
: Dec.id (sintetizado): almacena el identificador de la declaración.
Dec.tipo (sintetizado): almacena el tipo del identificador de la declaración

Tipo:
: Tipo.tipo (sintetizado): almacena el tipo del identificador de la declaración.

3.2.3 Gramática de atributos

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

Decs → Dec Decs
    Decs(0).dir = Decs(1).dir + 1
    Decs(0).ts = añadeID(Decs(1).ts, Dec.id, <tipo: Dec.tipo, dir: Decs(1).dir>)
Decs → λ
    Decs.dir = 0
    Decs.ts = creaTS()

Dec → Tipo **id pyc**
    Dec.tipo = Tipo.tipo
    Dec.id = id.lex

Tipo → **tInt**
    Tipo.tipo = int
Tipo → **tReal**
    Tipo.tipo = real

4. Especificación de las restricciones contextuales

4.1. Funciones semánticas

Descripción, si procede, de las funciones semánticas adicionales utilizadas en la especificación. Para cada función debe indicarse explícitamente su cabecera, así como informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros. Esta sección puede dejarse vacía si no se van a usar funciones semánticas adicionales.

existeId(t : Ts, id : String) : bool

Descrita en el apartado 3.1.

tipoDeId(t : Ts, id : String) : bool

Si el identificador *id* aparece en la tabla de símbolos *t*, entonces devuelve su tipo. Si no, devuelve error.

tipoDeB(op : Operador, tipo1 : Tipo, tipo2 : Tipo) : Tipo

Devuelve el tipo resultante de realizar la operación binaria *op* con un primer operando de tipo *tipo1* y un segundo operando de tipo *tipo2*. En las siguientes tablas, se puede ver el valor de retorno de la función para todas las combinaciones posibles de valores de Operador y Tipos.

<, <=, >, >=, ==, !=		**Tipo del operando 2**		
		int	**real**	**error**
Tipo del operando 1	**int**	int	int	error
	real	int	int	error
	error	error	error	error

+, -, *, /		**Tipo del operando 2**		
		int	**real**	**error**
Tipo del operando 1	**int**	int	real	error
	real	real	real	error
	error	error	error	error

||, &&, %		**Tipo del operando 2**		
		int	**real**	**error**
Tipo del operando 1	**int**	int	error	error
	real	error	error	error
	error	error	error	error

=	**Tipo del operando 2**		
	int	**real**	**error**

Tipo del operando 1				
	int	int	error	error
	real	real	real	error
	error	error	error	error

tipoDeU(op : Operador, tipo : Tipo) : Tipo

Devuelve el tipo resultante de realizar la operación unaria *op* sobre un operando de tipo *tipo*. En la siguiente tabla, se puede ver el valor de retorno de la función para todas las combinaciones de Operador y Tipo.

	Tipo del operando		
	int	**real**	**error**
!	int	error	error
-, in, out	int	real	error
(int)	int	int	error
(real)	real	real	error

4.2. Atributos semánticos

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

Decs:

Decs.err (sintetizado): contiene un valor booleano que indica si se han violado las restricciones contextuales en la sección de declaraciones.

Accs:

Accs.err (sintetizado): contiene un valor booleano que indica si se han violado las restricciones contextuales en la sección de acciones.

Accs. tsh (heredado): contiene la tabla de símbolos heredada de la sección de declaraciones.

Acc:

Acc.err (sintetizado): contiene un valor booleano que indica si se han violado las restricciones contextuales en la expresión contenida en la acción.

Acc. tsh (heredado): contiene la tabla de símbolos heredada de la sección de declaraciones.

ExprY = {Expr0, Expr1, Expr2, Expr3, Expr4, Expr5, Expr6}:

ExprY.tipo (sintetizado): contiene el tipo de la expresión (“int” o “real”) o “error”, en caso de no haber respetado las restricciones contextuales.

ExprY. tsh (heredado): contiene la tabla de símbolos heredada de la sección de declaraciones.

OperY = {Oper0, Oper1, Oper2, Oper3, Oper4, Oper5a, Oper5n}:

OperY.op (sintetizado): contiene el lexema del operador.

4.3. Gramática de atributos

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

Prog → Decs Accs

Prog.err = Decs.err ∨ Accs.err

Accs.tsh = Decs.ts

Decs → Dec Decs
    Decs(0).err = Decs(1).err ∨ existeID(Decs(1).ts, Dec.id)
Decs → λ
    Decs.err = false

Accs → Accs Acc
    Accs(1).tsh = Accs(0).tsh
    Acc.tsh = Accs(0).tsh
    Accs(0).err = Accs(1).err ∨ Acc.err
Accs → Acc
    Acc.tsh = Accs(0).tsh
    Accs.err = Acc.err

Acc → Expr0 **pyc**
    Expr0.tsh = Acc.tsh
    Acc.err = (Expr0.tipo == error)

Expr0 → **opIn id**
    Expr0.tipo = tipoDeU(in, tipoDeId(Expr0.tsh, id.lex))
Expr0 → **opOut** Expr1
    Expr1.tsh = Expr0.tsh
    Expr0.tipo = tipoDeU(out, Expr1.tipo)
Expr0 → Expr1
    Expr1.tsh = Expr0.tsh
    Expr0.tipo = Expr1.tipo

Expr1 → **id** Oper1 Expr1
    Expr1(1).tsh = Expr1(0).tsh
    Expr1(0).tipo = tipoDeB(Oper1.op, tipoDeId(Expr1(0).tsh, Expr1(1).tipo))
Expr1 → Expr2
    Expr2.tsh = Expr1.tsh
    Expr1.tipo = Expr2.tipo

Oper1 → **opAsig**
    Oper1.op = =

Expr2 → Expr3 Oper2 Expr3
    Expr3(0).tsh = Expr2.tsh
    Expr3(1).tsh = Expr2.tsh
    Expr2.tipo = tipoDeB(Oper2.op, Expr3(0).tipo, Expr3(1).tipo)
Expr2 → Expr3
    Expr3.tsh = Expr2.tsh
    Expr2.tipo = Expr3.tipo

Oper2 → **opMayor**
    Oper2.op = >
Oper2 → **opMenor**
    Oper2.op = <
Oper2 → **opMayIg**

Oper2.op = >=

Oper2 → **opMenIg**

Oper2.op = <=

Oper2 → **opIgual**

Oper2.op = ==

Oper2 → **opDistinto**

Oper2.op = !=

Expr3 → Expr3 Oper3 Expr4

Expr3(1).tsh = Expr3(0).tsh

Expr4.tsh = Expr3(0).tsh

Expr3(0).tipo = tipoDeB(Oper3.op, Expr3(1).tipo, Expr4.tipo)

Expr3 → Expr4

Expr4.tsh = Expr3.tsh

Expr3.tipo = Expr4.tipo

Oper3 → **opSuma**

Oper3.op = +

Oper3 → **opMenos**

Oper3.op = -

Oper3 → **opOr**

Oper3.op = ||

Expr4 → Expr4 Oper4 Expr5

Expr4(1).tsh = Expr4(0).tsh

Expr5.tsh = Expr4(0).tsh

Expr4(0).tipo = tipoDeB(Oper4.op, Expr4(1).tipo, Expr5.tipo)

Expr4 → Expr5

Expr5.tsh = Expr4.tsh

Expr4.tipo = Expr5.tipo

Oper4 → **opMult**

Oper4.op = *

Oper4 → **opDiv**

Oper4.op = /

Oper4 → **opMod**

Oper4.op= %

Oper4 → **opAnd**

Oper4.op= &&

Expr5 → Oper5a Expr5

Expr5(1).tsh = Expr5(0).tsh

Expr5(0).tipo = tipoDeU(Oper5a.op, Expr5(1).tipo)

Expr5 → Oper5n Expr6

Expr6.tsh = Expr5.tsh

Expr5.tipo = tipoDeU(Oper5n.op, Expr6.tipo)

Expr5 → Expr6

Expr6.tsh = Expr5.tsh

Expr5.tipo = Expr6.tipo

Oper5a → **opMenos**
    Oper5a.op = -
Oper5a → **opNot**
    Oper5a.op = !
Oper5n → **opCastInt**
    Oper5n.op = (int)
Oper5n → **opCastReal**
    Oper5n.op = (real)

Expr6 → **id**
    Expr6.tipo = tipoDeId(Expr6.tsh, id.lex)
Expr6 → **litNat**
    Expr6.tipo = int
Expr6 → **litReal**
    Expr6.tipo = real
Expr6 → **pa** Expr0 **pc**
    Expr0.tsh = Expr6.tsh
    Expr6.tipo = Expr0.tipo

5. Especificación de la traducción

5.1. Lenguaje objeto

5.1.1. Arquitectura de la máquina P

Explicar cómo es la arquitectura de la máquina P que se va a emplear en esta práctica.

La máquina P está formada por los siguientes componentes:

- Memoria de datos: contiene los datos que han sido almacenados durante la ejecución.
- Memoria de programa: contiene las instrucciones del programa que se está ejecutando.
- Contador de programa: indica qué instrucción de la memoria de programa se está ejecutando.
- Pila de datos: contiene los datos que se están utilizando durante la ejecución del programa. Permite acceder al elemento de su cima.
- Registro de funcionamiento: indica si el programa continúa ejecutándose o ha terminado. Mientras el valor del registro sea 0, el programa se ejecutará.

5.1.2. Instrucciones en el lenguaje objeto

Enumeración de todo el repertorio de instrucciones del lenguaje objeto de la máquina a pila (máquina P) que se van a utilizar, así como descripción informal de su cometido.

- apilaEnt(num): apila el literal entero *num* en la pila de la máquina P.
- apilaReal(num): apila el literal real *num* en la pila de la máquina P.
- apila-dir(dir): apila el contenido de la dirección *dir* de la memoria de datos en la pila de la máquina P.
- cima-dir(dir): guarda el contenido de la cima de la pila en la dirección *dir* de memoria.
- suma(): suma el contenido de la cima y la subcima de la pila, colocando el resultado en la cima.
- resta(): resta el contenido de la cima y la subcima de la pila, colocando el resultado en la cima.
- multiplica(): multiplica el contenido de la cima y la subcima de la pila, colocando el resultado en la cima.
- divide(): divide el contenido de la cima y la subcima de la pila, colocando el resultado en la cima.
- modulo(): calcula el resto de dividir el contenido de la cima y la subcima de la pila, colocando el resultado en la cima.
- or(): calcula la or lógica de la cima y la subcima de la pila, colocando el resultado en la cima.
- and(): calcula la and lógica de la cima y la subcima de la pila, colocando el resultado en la cima.
- mayor(): determina si la cima de la pila es mayor que la subcima, colocando el resultado en la cima.
- menor(): determina si la cima de la pila es menor que la subcima, colocando el resultado en la cima.
- mayorOIgual(): determina si la cima de la pila es mayor o igual que la subcima, colocando el resultado en la cima.

- menorOIgual(): determina si la cima de la pila es menor o igual que la subcima, colocando el resultado en la cima.
- igual(): determina si la cima de la pila es igual a la subcima, colocando el resultado en la cima.
- distinto(): determina si la cima de la pila es distinta a la subcima, colocando el resultado en la cima.
- menosUnario(): invierte el signo del dato de la cima de la pila, colocando el resultado en la cima.
- negacion(): realiza la inversión lógica del dato de la cima de la pila, colocando el resultado en la cima.
- castInt(): realiza una conversión a entero del dato de la cima de la pila, colocando el resultado en la cima.
- castReal(): realiza una conversión a real del dato de la cima de la pila, colocando el resultado en la cima.
- imprime(): imprime por pantalla el contenido de la cima de la pila.
- lee(): lee un dato introducido por el usuario y lo coloca en la cima de la pila.
- Vacia-pila() : limpia la pila de datos al finalizar una sentencia (Acc).

5.2. Funciones semánticas

Descripción, si procede, de las funciones semánticas adicionales utilizadas en la especificación. Para cada función debe indicarse explícitamente su cabecera, así como informalmente su cometido, incluyendo el propósito de cada uno de sus parámetros.

Esta sección puede dejarse vacía si no se van a usar funciones semánticas adicionales.

toInt(lex : String) : int

Devuelve el número entero correspondiente al lexema recibido.

toReal(lex : String) : real

Devuelve el número real correspondiente al lexema recibido.

genCod1(ts : Ts, cod : LInst, op : Operador, lex: String) : LInst

Devuelve la lista de instrucciones resultante de añadir a la lista de instrucciones *cod* la instrucción generada a partir del resto de parámetros. En la siguiente tabla se puede ver la instrucción generada para cada operador.

Operador	Instrucción
=	cima_dir(ts[lex].dir)

genCod2(cod : LInst, op : Operador) : LInst

Devuelve la lista de instrucciones resultante de añadir a la lista de instrucciones *cod* la instrucción generada a partir del resto de parámetros. En la siguiente tabla se puede ver la instrucción generada para cada operador.

Operador	Instrucción
>	mayor()
<	menor()
>=	mayorOIgual()

<=	menorOIgual()
==	igual()
!==	distinto()

genCod3(cod : LInst, op : Operador) : LInst

Devuelve la lista de instrucciones resultante de añadir a la lista de instrucciones *cod* la instrucción generada a partir del resto de parámetros. En la siguiente tabla se puede ver la instrucción generada para cada operador.

Operador	Instrucción
+	suma()
-	resta()
||	or()

genCod4(cod : LInst, op : Operador) : LInst

Devuelve la lista de instrucciones resultante de añadir a la lista de instrucciones *cod* la instrucción generada a partir del resto de parámetros. En la siguiente tabla se puede ver la instrucción generada para cada operador.

Operador	Instrucción
*	multiplica()
/	divide()
%	modulo()
&&	and()

genCod5a(cod : LInst, op : Operador) : LInst

Devuelve la lista de instrucciones resultante de añadir a la lista de instrucciones *cod* la instrucción generada a partir del resto de parámetros. En la siguiente tabla se puede ver la instrucción generada para cada operador.

Operador	Instrucción
-	menosUnario()
!	negacion()

genCod5n(cod : LInst, op : Operador) : LInst

Devuelve la lista de instrucciones resultante de añadir a la lista de instrucciones *cod* la instrucción generada a partir del resto de parámetros. En la siguiente tabla se puede ver la instrucción generada para cada operador.

Operador	Instrucción
(int)	castInt()
(real)	castReal()

5.3. Atributos semánticos

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

Prog:

Prog.cod (sintetizado): almacena la secuencia de instrucciones del programa.

Accs:

Accs.cod (sintetizado): almacena la secuencia de instrucciones resultante de traducir las acciones.

Acc :

Acc.cod (sintetizado): almacena la secuencia de instrucciones resultante de traducir la acción.

ExpY={Exp0, Exp1, Exp2, Exp3, Exp4, Exp5, Exp6}:

ExpY.cod (sintetizado): almacena la secuencia de instrucciones resultante de traducir la expresión.

5.4. Gramática de atributos

Gramática de atributos que formaliza la traducción

Prog → Decs Accs

Prog.cod = Accs.cod

Accs → Accs Acc

Accs(0).cod = Accs(1).cod || Acc.cod

Accs → Acc

Accs.cod = Acc.cod

Acc → Expr0 **pyc**

Acc.cod = Expr0.cod || vacia-pila()

Expr0 → **opIn id**

Expr0.cod = lee() || cima-dir(Expr0.tsh[id.lex].dir)

Expr0 → **opOut** Expr1

Expr0.cod = Expr1.cod || imprime()

Expr0 → Expr1

Expr0.cod = Expr1.cod

Expr1 → **id** Oper1 Expr1

Expr1(0).cod = genCod1(Expr1.tsh, Expr1(1).cod, Oper1.op, id.lex)

Expr1 → Expr2

Expr1.cod = Expr2.cod

Expr2 → Expr3 Oper2 Expr3

Expr2.cod = genCod2(Expr3(0).cod || Expr3(1).cod, Oper2.op)

Expr2 → Expr3

Expr2.cod = Expr3.cod

Expr3 → Expr3 Oper3 Expr4
Expr3(0).cod = genCod3(Expr3(1).cod || Expr4.cod , Oper3.op)
Expr3 → Expr4
Expr3.cod = Expr4.cod

Expr4 → Expr4 Oper4 Expr5
Expr4(0).cod = genCod4(Expr4(1).cod || Expr5.cod, Oper4.op)
Expr4 → Expr5
Expr4.cod = Expr5.cod

Expr5 → Oper5a Expr5
Expr5(0).cod = genCod5a(Expr5(1).cod, Oper5a.op)
Expr5 → Oper5n Expr6
Expr5.cod = genCod5n(Expr6.cod, Oper5n.op)
Expr5 → Expr6
Expr5.cod = Expr6.cod

Expr6 → **litNat**
Expr6.cod = apilaEnt(toInt(litNat.lex))
Expr6 → **litReal**
Expr6.cod = apilaReal(toReal(litReal.lex))
Expr6 → **id**
Expr6.cod = apila-dir(Expr6.tsh[id.lex].dir)
Expr6 → **pa** Expr0 **pc**
Expr6.cod = Expr0.cod

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.

La implemantación ha sido realizada en javaCC, por lo tanto no ha sido necesario implementar un analizador léxico.

7. Acondicionamiento de las gramáticas de atributos

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

Únicamente deben incluirse la transformación de las producciones que se ven afectadas. Si alguna de las gramáticas no necesitan acondicionamiento, dejar el correspondiente subapartado en blanco.

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

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

Eliminación de la recursión a izquierdas:

Se sustituye:

Accs → Accs Acc
    Accs(1).tsh = Accs(0).tsh
    Acc.tsh = Accs(0).tsh
    Accs(0).err = Accs(1).err ∨ Acc.err

Accs → Acc
    Acc.tsh = Accs(0).tsh
    Accs.err = Acc.err

por:

Accs → Acc Raccs
    Acc.tsh = Accs.tsh
    Raccs.tsh = Accs.tsh
    Raccs.errh =Acc.err
    Accs.err = Raccs.err

Raccs → Acc Raccs
    Acc.tsh = Raccs(0).tsh
    Raccs(1).tsh = Raccs(0).tsh
    Raccs(1).errh = Acc.err ∨ Raccs(0).errh
    Raccs(0).err = Raccs(1).err

Raccs → λ
    Raccs.err=Raccs.errh

Se sustituye:

Expr3 → Expr3 Oper3 Expr4
    Expr3(1).tsh = Expr3(0).tsh
    Expr4.tsh = Expr3(0).tsh
    Expr3(0).tipo = tipoDeB(Oper3.op, Expr3(1).tipo, Expr4.tipo)

Expr3 → Expr4
Expr4.tsh = Expr3.tsh
Expr3.tipo = Expr4.tipo
por:
Expr3 → Expr4 Res3
Expr4.tsh = Expr3.tsh
Res3.tsh = Expr3.tsh
Res3.tipoh = Expr4.tipo
Expr3.tipo = Res3.tipo
Res3 → Oper3 Expr4 Res3
Expr4.tsh = Res3.tsh
Res3(1).tsh = Res3(0).tsh
Res3(1).tipoh = tipoDeB(Oper3.op, Res3(0).tipoh, Expr4.tipo)
Res3(0).tipo = Res3(1).tipo
Res3 → λ
Res3.tipo = Res3.tipoh

Se sustituye:
Expr4 → Expr4 Oper4 Expr5
Expr4(1).tsh = Expr4(0).tsh
Expr5.tsh = Expr4(0).tsh
Expr4(0).tipo = tipoDeB(Oper4.op, Expr4(1).tipo, Expr5.tipo)
Expr4 → Expr5
Expr5.tsh = Expr4.tsh
Expr4.tipo = Expr5.tipo
por:
Expr4 → Expr5 Res4
Expr5.tsh = Expr4.tsh
Res4.tsh = Expr4.tsh
Res4.tipoh = Expr5.tipo
Expr4.tipo = Res4.tipo
Res4 → Oper4 Expr5 Res4
Expr5.tsh = Res4.tsh
Res4(1).tsh = Res4(0).tsh
Res4(1).tipoh = tipoDeB(Oper4.op, Res4(0).tipoh, Expr5.tipo)
Res4(0).tipo = Res4(1).tipo
Res4 → λ
Res4.tipo = Res4.tipoh

Factorización:

Se sustituye:
Expr2 → Expr3 Oper2 Expr3
Expr3(0).tsh = Expr2.tsh
Expr3(1).tsh = Expr2.tsh
Expr2.tipo = tipoDeB(Oper2.op, Expr3(0).tipo, Expr3(1).tipo)
Expr2 → Expr3
Expr3.tsh = Expr2.tsh
Expr2.tipo = Expr3.tipo
por:

Expr2 → Expr3 Res2
Expr3.tsh = Expr2.tsh
Res2.tsh = Expr2.tsh
Res2.tipoh = Expr3.tipo
Expr2.tipo = Res2.tipo
Res2 → Oper2 Expr3
Expr3.tsh = Res2.tsh
Res2.tipo = tipoDeB(Oper2.op, Res2.tipoh, Expr3.tipo)
Res2 → λ
Res2.tipo = Res2.tipoh

Factorización profunda:

1. Se añaden las siguientes producciones a la gramática:
Expr6O → **litNat**
Expr6O.tipo = int
Expr6O → **litReal**
Expr6O.tipo = real
Expr6O → **pa** Expr0 **pc**
Expr0.tsh = Expr6O.tsh
Expr6O.tipo = Expr0.tipo
y se sustituyen las producciones de Expr6:
Expr6 → **id**
Expr6.tipo = tipoDeId(Expr6.tsh, id.lex)
Expr6 → **litNat**
Expr6.tipo = int
Expr6 → **litReal**
Expr6.tipo = real
Expr6 → **pa** Expr0 **pc**
Expr0.tsh = Expr6.tsh
Expr6.tipo = Expr0.tipo
por estas:
Expr6 → **id**
Expr6.tipo = tipoDeId(Expr6.tsh, id.lex)
Expr6 → Expr6O
Expr6O.tsh = Expr6.tsh
Expr6.tipo = Expr6O.tipo

2. Se sustituyen las producciones de Expr5:
Expr5 → Oper5a Expr5
Expr5(1).tsh = Expr5(0).tsh
Expr5(0).tipo = tipoDeU(Oper5a.op, Expr5(1).tipo)
Expr5 → Oper5n Expr6
Expr6.tsh = Expr5.tsh
Expr5.tipo = tipoDeU(Oper5n.op, Expr6.tipo)
Expr5 → Expr6
Expr6.tsh = Expr5.tsh
Expr5.tipo = Expr6.tipo
por estas:
Expr5 → Oper5a Expr5

Expr5(1).tsh = Expr5(0).tsh
Expr5(0).tipo = tipoDeU(Oper5a.op, Expr5(1).tipo)

Expr5 → Oper5n Expr6
Expr6.tsh = Expr5.tsh
Expr5.tipo = tipoDeU(Oper5n.op, Expr6.tipo)

Expr5 → **id**
Expr5.tipo = tipoDeId(Expr5.tsh, id.lex)

Expr5 → Expr6O
Expr6O.tsh = Expr5.tsh
Expr5.tipo = Expr6O.tipo

3. Se añaden las siguientes producciones a la gramática:

Expr5O → Oper5a Expr5
Expr5.tsh = Expr5O.tsh
Expr5O.tipo = tipoDeU(Oper5a.op, Expr5.tipo)

Expr5O → Oper5n Expr6
Expr6.tsh = Expr5O.tsh
Expr5O.tipo = tipoDeU(Oper5n.op, Expr6.tipo)

Expr5O → Expr6O
Expr6O.tsh = Expr5O.tsh
Expr5O.tipo = Expr6O.tipo

y se sustituyen las producciones de Expr5 obtenidas en el punto anterior por estas:

Expr5 → **id**
Expr5.tipo = tipoDeId(Expr5.tsh, id.lex)

Expr5 → Expr5O
Expr5O.tsh = Expr5.tsh
Expr5.tipo = Expr5O.tipo

4. Se sustituye la producción de Expr4:

Expr4 → Expr5 Res4
Expr5.tsh = Expr4.tsh
Res4.tsh = Expr4.tsh
Res4.tipoh = Expr5.tipo
Expr4.tipo = Res4.tipo

por estas:

Expr4 → **id** Res4
Res4.tsh = Expr4.tsh
Res4.tipoh = tipoDeId(Expr4.tsh, id.lex)
Expr4.tipo = Res4.tipo

Expr4 → Expr5O Res4
Expr5O.tsh = Expr4.tsh
Res4.tsh = Expr4.tsh
Res4.tipoh = Expr5O.tipo
Expr4.tipo = Res4.tipo

5. Se sustituye la producción de Expr3:

Expr3 → Expr4 Res3
Expr4.tsh = Expr3.tsh
Res3.tsh = Expr3.tsh
Res3.tipoh = Expr4.tipo

Expr3.tipo = Res3.tipo

por estas:

Expr3 → **id** Res4 Res3
Res4.tsh = Expr3.tsh
Res3.tsh = Expr3.tsh
Res4.tipoh = tipoDeId(Expr3.tsh, id.lex)
Res3.tipoh = Res4.tipo
Expr3.tipo = Res3.tipo

Expr3 → Expr5O Res4 Res3
Expr5O.tsh = Expr3.tsh
Res4.tsh = Expr3.tsh
Res3.tsh = Expr3.tsh
Res4.tipoh = Expr5O.tipo
Res3.tipoh = Res4.tipo
Expr3.tipo = Res3.tipo

6. Se sustituye la producción de Expr2:

Expr2 → Expr3 Res2
Expr3.tsh = Expr2.tsh
Res2.tsh = Expr2.tsh
Res2.tipoh = Expr3.tipo
Expr2.tipo = Res2.tipo

por estas:

Expr2 → **id** Res4 Res3 Res2
Res4.tsh = Expr2.tsh
Res3.tsh = Expr2.tsh
Res2.tsh = Expr2.tsh
Res4.tipoh = tipoDeId(Expr2.tsh, id.lex)
Res3.tipoh = Res4.tipo
Res2.tipoh = Res3.tipo
Expr2.tipo = Res2.tipo

Expr2 → Expr5O Res4 Res3 Res2
Expr5O.tsh = Expr2.tsh
Res4.tsh = Expr2.tsh
Res3.tsh = Expr2.tsh
Res2.tsh = Expr2.tsh
Res4.tipoh = Expr5O.tipo
Res3.tipoh = Res4.tipo
Res2.tipoh = Res3.tipo
Expr2.tipo = Res2.tipo

7. Se sustituyen las producciones de Expr1:

Expr1 → **id** Oper1 Expr1
Expr1(1).tsh = Expr1(0).tsh
Expr1(0).tipo = tipoDeB(Oper1.op, tipoDeId(Expr1(0).tsh, Expr1(1).tipo))

Expr1 → Expr2
Expr2.tsh = Expr1.tsh
Expr1.tipo = Expr2.tipo

por estas:

Expr1 → **id** Oper1 Expr1

Expr1(1).tsh = Expr1(0).tsh
Expr1(0).tipo = tipoDeB(Oper1.op, tipoDeId(Expr1(0).tsh, Expr1(1).tipo))

Expr1 → **id** Res4 Res3 Res2
Res4.tsh = Expr1.tsh
Res3.tsh = Expr1.tsh
Res2.tsh = Expr1.tsh
Res4.tipoh = tipoDeId(Expr2.tsh, id.lex)
Res3.tipoh = Res4.tipo
Res2.tipoh = Res3.tipo
Expr1.tipo = Res2.tipo

Expr1 → Expr5O Res4 Res3 Res2
Expr5O.tsh = Expr1.tsh
Res4.tsh = Expr1.tsh
Res3.tsh = Expr1.tsh
Res2.tsh = Expr1.tsh
Res4.tipoh = Expr5O.tipo
Res3.tipoh = Res4.tipo
Res2.tipoh = Res3.tipo
Expr1.tipo = Res2.tipo

y se eliminan las producciones Expr2 de la gramática.

8. Se aplica factorización sobre las producciones de Expr1:

Expr1 → Expr5O Res4 Res3 Res2
Expr5O.tsh = Expr1.tsh
Res4.tsh = Expr1.tsh
Res3.tsh = Expr1.tsh
Res2.tsh = Expr1.tsh
Res4.tipoh = Expr5O.tipo
Res3.tipoh = Res4.tipo
Res2.tipoh = Res3.tipo
Expr1.tipo = Res2.tipo

Expr1 → **id** Res1
Res1.tsh = Expr1.tsh
Res1.tipoh = tipoDeId(Expr1.tsh, id.lex)
Expr1.tipo = Res1.tipo

Res1 → Oper1 Expr1
Expr1.tsh = Res1.tsh
Res1.tipo = tipoDeB(Oper1.op, Res1.tipoh, Expr1.tipo)

Res1 → Res4 Res3 Res2
Res4.tsh = Res1.tsh
Res3.tsh = Res1.tsh
Res2.tsh = Res1.tsh
Res4.tipoh = Res1.tipoh
Res3.tipoh = Res4.tipo
Res2.tipoh = Res3.tipo
Res1.tipo = Res2.tipo

9. Para reducir el número de producciones y ecuaciones semánticas, cambiamos las producciones Expr3 y Expr4 por las que había al comienzo de la factorización profunda.

Expr3 → Expr4 Res3

Expr4.tsh = Expr3.tsh
Res3.tsh = Expr3.tsh
Res3.tipoh = Expr4.tipo
Expr3.tipo = Res3.tipo

Expr4 → Expr5 Res4
Expr5.tsh = Expr4.tsh
Res4.tsh = Expr4.tsh
Res4.tipoh = Expr5.tipo
Expr4.tipo = Res4.tipo

7.3. Acondicionamiento de la Gramática para la Traducción

Eliminación de la recursión a izquierdas:

Se sustituye:

Accs → Accs Acc
Accs(0).cod = Accs(1).cod || Acc.cod

Accs → Acc
Accs.cod = Acc.cod

por:

Accs → Acc Raccs
Raccs.codh = Acc.cod
Accs.cod = Raccs.cod

Raccs → Acc Raccs
Raccs(1).codh = Raccs(0).codh || Acc.cod
Raccs(0).cod = Raccs(1).cod

Raccs → λ
Raccs.cod = Raccs.codh

Se sustituye:

Expr3 → Expr3 Oper3 Expr4
Expr3(0).cod = genCod3(Expr3(1).cod || Expr4.cod ,Oper3.op)

Expr3 → Expr4
Expr3.cod = Expr4.cod

por:

Expr3 → Expr4 Res3
Res3.codh = Expr4.cod
Expr3.cod = Res3.cod

Res3 → Oper3 Expr4 Res3
Res3(1).codh = genCod3(Res3(0).codh || Expr4.cod, Oper3.op)
Res3(0).cod = Res3(1).cod

Res3 → λ
Res3.cod = Res3.codh

Se sustituye:

Expr4 → Expr4 Oper4 Expr5
Expr4(0).cod = genCod4(Expr4(1).cod || Expr5.cod ,Oper4.op)

Expr4 → Expr5
Expr4.cod = Expr5.cod

por:
Expr4 → Expr5 Res4
Res4.codh = Expr5.cod
Exp4.cod = Res4.cod
Res4 → Oper4 Expr5 Res4
Res4(1).codh = genCod4(Res4(0).codh || Expr5.cod, Oper4.op)
Res4(0).cod = Res4(1).cod
Res4 → λ
Res4.cod = Res4.codh

Factorización:

Se sustituye:
Expr2 → Expr3 Oper2 Expr3
Expr2.cod = genCod2(Expr3(0).cod || Expr3(1).cod, Oper2.op)
Expr2 → Expr3
Expr2.cod = Expr3.cod
por:
Expr2 → Expr3 Res2
Res2.codh = Expr3.cod
Expr2.cod = Res2.cod
Res2 → Oper2 Expr3
Res2.cod = genCod2(Res2.codh || Expr3.cod, Oper2.op)
Res2 → λ
Res2.cod = Res2.codh

Factorización profunda:

1. Se añaden las siguientes producciones a la gramática:
Expr6O → **litNat**
Expr6O.cod = apilaEnt(toInt(litNat.lex))
Expr6O → **litReal**
Expr6O.cod = apilaReal(toReal(litReal.lex))
Expr6O → **pa** Expr0 **pc**
Expr6O.cod = Expr0.cod
y se sustituyen las producciones de Expr6:
Expr6 → **litNat**
Expr6.cod = apilaEnt(toInt(litNat.lex))
Expr6 → **litReal**
Expr6.cod = apilaReal(toReal(litReal.lex))
Expr6 → **id**
Expr6.cod = apila-dir(Expr6.tsh[id.lex].dir)
Expr6 → **pa** Expr0 **pc**
Expr6.cod = Expr0.cod
por estas:
Expr6 → **id**
Expr6.cod = apila-dir(Expr6.tsh[id.lex].dir)
Expr6 → Expr6O
Expr6.cod = Expr6O.cod

2. Se sustituyen las producciones de Expr5:

Expr5 → Oper5a Expr5
    Expr5(0).cod = genCod5a(Expr5(1).cod, Oper5a.op)
Expr5 → Oper5n Expr6
    Expr5.cod = genCod5n(Expr6.cod, Oper5n.op)
Expr5 → Expr6
    Expr5.cod = Expr6.cod

por estas:

Expr5 → Oper5a Expr5
    Expr5(0).cod = genCod5a(Expr5(1).cod, Oper5a.op)
Expr5 → Oper5n Expr6
    Expr5.cod = genCod5n(Expr6.cod, Oper5n.op)
Expr5 → **id**
    Expr5.cod = apila-dir(Expr5.tsh[id.lex].dir)
Expr5 → Expr6O
    Expr5.cod = Expr6O.cod

3. Se añaden las siguientes producciones a la gramática:

Expr5O → Oper5a Expr5
    Expr5O.cod = genCod5a(Expr5.cod, Oper5a.op)
Expr5O → Oper5n Expr6
    Expr5O.cod = genCod5n(Expr6.cod, Oper5n.op)
Expr5O → Expr6O
    Expr5.cod = Expr6O.cod

y se sustituyen las producciones de Expr5 obtenidas en el punto anterior por estas:

Expr5 → **id**
    Expr5.cod = apila-dir(Expr5.tsh[id.lex].dir)
Expr5 → Expr5O
    Expr5.cod = Expr5O.cod

4. Se sustituye la producción de Expr4:

Expr4 → Expr5 Res4
    Res4.codh = Expr5.cod
    Exp4.cod = Res4.cod

por estas:

Expr4 → **id** Res4
    Res4.codh = apila-dir(Expr4.tsh[id.lex].dir)
    Expr4.cod = Res4.cod
Expr4 → Expr5O Res4
    Res4.codh = Expr5O.cod
    Expr4.cod = Res4.cod

5. Se sustituye la producción de Expr3:

Expr3 → Expr4 Res3
    Res3.codh = Expr4.cod
    Expr3.cod = Res3.cod

por estas:

Expr3 → **id** Res4 Res3
    Res4.codh = apila-dir(Expr3.tsh[id.lex].dir)
    Res3.codh = Res4.cod

Expr3.cod = Res3.cod
Expr3 → Expr5O Res4 Res3
Res4.codh = Expr5O.cod
Res3.codh = Res4.cod
Expr3.cod = Res3.cod

6. Se sustituye la producción de Expr2:
Expr2 → Expr3 Res2
Res2.codh = Expr3.cod
Expr2.cod = Res2.cod
por estas:
Expr2 → **id** Res4 Res3 Res2
Res4.codh = apila-dir(Expr2.tsh[id.lex].dir)
Res3.codh = Res4.cod
Res2.codh = Res3.cod
Expr2.cod = Res2.cod
Expr2 → Expr5O Res4 Res3 Res2
Res4.codh = Expr5O.cod
Res3.codh = Res4.cod
Res2.codh = Res3.cod
Expr2.cod = Res2.cod

7. Se sustituyen las producciones de Expr1:
Expr1 → **id** Oper1 Expr1
Expr1(0).cod = genCod1(Expr1.tsh, Expr1(1).cod, Oper1.op, id.lex)
Expr1 → Expr2
Expr1.cod = Expr2.cod
por estas:
Expr1 → **id** Oper1 Expr1
Expr1(0).cod = genCod1(Expr1.tsh, Expr1(1).cod, Oper1.op, id.lex)
Expr1 → **id** Res4 Res3 Res2
Res4.codh = apila-dir(Expr1.tsh[id.lex].dir)
Res3.codh = Res4.cod
Res2.codh = Res3.cod
Expr1.cod = Res2.cod
Expr1 → Expr5O Res4 Res3 Res2
Res4.codh = Expr5O.cod
Res3.codh = Res4.cod
Res2.codh = Res3.cod
Expr1.cod = Res2.cod
y se eliminan las producciones de Expr2 de la gramática.

8. Se aplica factorización sobre las producciones de Expr1:
Expr1 → Expr5O Res4 Res3 Res2
Res4.codh = Expr5O.cod
Res3.codh = Res4.cod
Res2.codh = Res3.cod
Expr1.cod = Res2.cod
Expr1 → **id** Res1
Res1.lexh = id.lex

Expr1.cod = Res1.cod

Res1 → Oper1 Expr1

Res1.cod = genCod1(Res1.tsh, Expr1.cod, Oper1.op, Res1.lexh)

Res1 → Res4 Res3 Res2

Res4.codh = apila-dir(Res1.tsh[Res1.lexh].dir)

Res3.codh = Res4.cod

Res2.codh = Res3.cod

Res1.cod = Res2.cod

9. Para reducir el número de producciones y ecuaciones semánticas, cambiamos las producciones Expr3 y Expr4 por las que había al comienzo de la factorización profunda.

Expr3 → Expr4 Res3

Res3.codh = Expr4.cod

Expr3.cod = Res3.cod

Expr4 → Expr5 Res4

Res4.codh = Expr5.cod

Exp4.cod = Res4.cod

8. Esquema de traducción orientado a las gramáticas de atributos

Esquema de traducción en el que se muestre, mediante acciones semánticas, los lugares en los que deben evaluarse las distintas ecuaciones contempladas en todas las gramáticas de atributos: la de construcción de la tabla de símbolos, la de comprobación de las restricciones contextuales y la de especificación de la traducción.

El recorrido del árbol será el realizado por un analizador predictivo-recursivo.

Prog → Decs
{Accs.tsh = Decs.ts}
Accs
{Prog.err = Decs.err ∨ Accs.err;
Prog.cod = Accs.cod}

Decs → Dec Decs
{Decs(0).dir = Decs(1).dir + 1;
Decs(0).err = Decs(1).err ∨ existeID(Decs(1).ts, Dec.id);
Decs(0).ts = añadeID(Decs(1).ts, Dec.id, <tipo: Dec.tipo, dir: Decs(1).dir>)}
Decs → λ
{Decs.dir = 0;
Decs.ts = creaTS();
Decs.err = false}

Dec → Tipo
{Dec.tipo = Tipo.tipo}
id
{Dec.id = id.lex}
pyc

Tipo → **tInt**
{Tipo.tipo = int}
Tipo → **tReal**
{Tipo.tipo = real}

Accs → {Acc.tsh = Accs.tsh}
Acc
{Raccs.codh = Acc.cod;
Raccs.errh = Acc.err;
Raccs.tsh = Accs.tsh}
Raccs
{Accs.err = Raccs.err;
Accs.cod = Raccs.cod}

Raccs → {Acc.tsh = Raccs(0).tsh}
Acc
{Raccs(1).tsh = Raccs(0).tsh;
Raccs(1).errh = Acc.err ∨ Raccs(0).errh;
Raccs(1).codh = Raccs(0).codh || Acc.cod}
Raccs
{Raccs(0).err = Raccs(1).err;

Raccs(0).cod = Raccs(1).cod}

Raccs → λ
{Raccs.err = Raccs.errh;
Raccs.cod = Raccs.codh}

Acc → {Expr0.tsh = Acc.tsh}
Expr0
{Acc.cod = Expr0.cod || vacia-pila();
Acc.err = (Expr0.tipo == error)}
pyc

Expr0 → **opIn id**
{Expr0.cod = lee() || cima-dir(Expr0.tsh[id.lex].dir);
Expr0.tipo = tipoDeU(in, tipoDeId(Expr0.tsh, id.lex))}
Expr0 → **opOut**
{Expr1.tsh = Expr0.tsh}
Expr1
{Expr0.cod = Expr1.cod || imprime();
Expr0.tipo = tipoDeU(out, Expr1.tipo)}
Expr0 → {Expr1.tsh = Expr0.tsh}
Expr1
{Expr0.cod = Expr1.cod;
Expr0.tipo = Expr1.tipo}

Expr1 → {Expr5O.tsh = Expr1.tsh}
Expr5O
{Res4.tipoh = Expr5O.tipo;
Res4.codh = Expr5O.cod;
Res4.tsh = Expr1.tsh}
Res4
{Res3.tipoh = Res4.tipo;
Res3.codh = Res4.cod;
Res3.tsh = Expr1.tsh}
Res3
{Res2.tipoh = Res3.tipo;
Res2.codh = Res3.cod;
Res2.tsh = Expr1.tsh}
Res2
{Expr1.cod = Res2.cod;
Expr1.tipo = Res2.tipo}
Expr1 → **id**
{Res1.tsh = Expr1.tsh;
Res1.tipoh = tipoDeId(Expr1.tsh, id.lex);
Res1.lexh = id.lex}
Res1
{Expr1.cod = Res1.cod
Expr1.tipo = Res1.tipo}

Res1 → Oper1
{Expr1.tsh = Res1.tsh}

Expr1
{Res1.tipo = tipoDeB(Oper1.op, Res1.tipoh, Expr1.tipo);
Res1.cod = genCod1(Res1.tsh, Expr1.cod, Oper1.op, Res1.lexh)}

Res1 → {Res4.codh = apila-dir(Res1.tsh[Res1.lexh].dir);
Res4.tipoh = Res1.tipoh;
Res4.tsh = Res1.tsh}
Res4
{Res3.codh = Res4.cod;
Res3.tipoh = Res4.tipo;
Res3.tsh = Res1.tsh}
Res3
{Res2.codh = Res3.cod;
Res2.tipoh = Res3.tipo;
Res2.tsh = Res1.tsh}
Res2
{Res1.cod = Res2.cod;
Res1.tipo = Res2.tipo}

Oper1 → **opAsig**
{Oper1.op = =}

Res2 → Oper2
{Expr3.tsh = Res2.tsh}
Expr3
{Res2.tipo = tipoDeB(Oper2.op, Res2.tipoh, Expr3.tipo);
Res2.cod = genCod2(Res2.codh || Expr3.cod,Oper2.op)}

Res2 → λ
{Res2.tipo = Res2.tipoh;
Res2.cod = Res2.codh}

Oper2 → **opMayor**
{Oper2.op = >}

Oper2 → **opMenor**
{Oper2.op = <}

Oper2 → **opMayIg**
{Oper2.op = >=}

Oper2 → **opMenIg**
{Oper2.op = <=}

Oper2 → **opIgual**
{Oper2.op = ==}

Oper2 → **opDistinto**
{Oper2.op = !=}

Expr3 → {Expr4.tsh = Expr3.tsh}
Expr4
{Res3.codh = Expr4.cod;
Res3.tsh = Expr3.tsh;
Res3.tipoh = Expr4.tipo}
Res3
{Expr3.cod = Res3.cod;
Expr3.tipo = Res3.tipo}

Res3 → Oper3
{Expr4.tsh = Res3.tsh}
Expr4
{Res3(1).tsh = Res3(0).tsh;
Res3(1).codh = genCod3(Res3(0).codh || Expr4.cod,Oper3.op);
Res3(1).tipoh = tipoDeB(Oper3.op, Res3(0).tipoh, Expr4.tipo)}
Res3
{Res3(0).tipo = Res3(1).tipo;
Res3(0).cod = Res3(1).cod}
Res3 → λ
{Res3.tipo = Res3.tipoh;
Res3.cod = Res3.codh}

Oper3 → **opSuma**
{Oper3.op = +}
Oper3 → **opMenos**
{Oper3.op = -}
Oper3 → **opOr**
{Oper3.op = ||}

Expr4 → {Expr5.tsh = Expr4.tsh}
Expr5
{Res4.codh = Expr5.cod;
Res4.tsh = Expr4.tsh;
Res4.tipoh = Expr5.tipo}
Res4
{Expr4.cod = Res4.cod;
Expr4.tipo = Res4.tipo}

Res4 → Oper4
{Expr5.tsh = Res4.tsh}
Expr5
{Res4(1).codh = genCod4(Res4(0).codh || Expr5.cod, Oper4.op);
Res4(1).tsh = Res4(0).tsh
Res4(1).tipoh = tipoDeB(Oper4.op, Res4(0).tipoh, Expr5.tipo)}
Res4
{Res4(0).cod = Res4(1).cod;
Res4(0).tipo = Res4(1).tipo}
Res4 → λ
{Res4.cod = Res4.codh;
Res4.tipo = Res4.tipoh}

Oper4 → **opMult**
{Oper4.op = *}
Oper4 → **opDiv**
{Oper4.op = /}
Oper4 → **opMod**
{Oper4.op= %}
Oper4 → **opAnd**

{Oper4.op= &&}

Expr5 → **id**
{Expr5.cod = apila-dir(Expr5.tsh[id.lex].dir);
Expr5.tipo = tipoDeId(Expr5.tsh, id.lex)}
Expr5 → {Expr5O.tsh = Expr5.tsh}
Expr5O
{Expr5.tipo = Expr5O.tipo;
Expr5.cod = Expr5O.cod}

Expr5O → Oper5a
{Expr5.tsh = Expr5O.tsh}
Expr5
{Expr5O.cod = genCod5a(Expr5.cod, Oper5a.op);
Expr5O.tipo = tipoDeU(Oper5a.op, Expr5.tipo)}
Expr5O → Oper5n
{Expr6.tsh = Expr5O.tsh}
Expr6
{Expr5O.cod = genCod5n(Expr6.cod, Oper5n.op);
Expr5O.tipo = tipoDeU(Oper5n.op, Expr6.tipo)}
Expr5O → {Expr6O.tsh = Expr5O.tsh}
Expr6O
{Expr5.cod = Expr6O.cod;
Expr5O.tipo = Expr6O.tipo}

Oper5a → **opMenos**
{Oper5a.op = -}
Oper5a → **opNot**
{Oper5a.op = !}
Oper5n → **opCastInt**
{Oper5n.op = (int)}
Oper5n → **opCastReal**
{Oper5n.op = (real)}

Expr6 → **id**
{Expr6.tipo = tipoDeId(Expr6.tsh, id.lex);
Expr6.cod = apila-dir(Expr6.tsh[id.lex].dir)}
Expr6 → {Expr6O.tsh = Expr6.tsh}
Expr6O
{Expr6.tipo = Expr6O.tipo;
Expr6.cod = Expr6O.cod}

Expr6O → **litNat**
{Expr6O.cod = apilaEnt(toInt(litNat.lex));
Expr6O.tipo = int}
Expr6O → **litReal**
{Expr6O.cod = apilaReal(toReal(litReal.lex));
Expr6O.tipo = real}
Expr6O → **pa**
{Expr0.tsh = Expr6O.tsh}

Expr0
{Expr6O.cod = Expr0.cod;
  Expr6O.tipo = Expr0.tipo}
pc

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

Descripción y propósito de las variables globales usadas.

Si no se usan variables globales, dejar este apartado en blanco.

errProg : Variable cuyo valor será “cierto” si se ha producido un error en la traducción.

cod: lista de instrucciones, el código que se va generando.

ts: tabla de símbolos, contendrá las variables y síbolos declarados.

9.2. Nuevas operaciones y transformación de ecuaciones semánticas

En caso de introducir nuevas operaciones (por ejemplo: el procedimiento “emite”), deben describirse aquí.

Debe describirse, así mismo, qué ecuaciones semánticas se ven transformadas y cómo (por ejemplo: la generación de código se implementa emitiendo la última instrucción).

emite(LInst: cod , Instruccion i) : emite una única instrucción, es decir la añade a la lista de instrucciones (variable global) cod.

emite(LInst: cod , Linst l) : emite una lista de instrucciones.

Las funciones semánticas de generación de código (genCod) pierden el argumento cod, ya que ahora no añaden instrucciones a una lista de código, sino que devuelven las instrucciones adecuadas que ya se encarga de concatenar al código el procedimiento “emite”.

9.3. Esquema de traducción

```
Prog() →        {errProg=falso }
                Decs( _ )
                Accs()

Decs(out dirDecs0) →
                {var idDec, tipoDec, dirDecs1}
                Dec(idDec, tipoDec)
                Decs(dirDecs1)
                {dirDecs0 = dirDecs1 + 1;
                  errProg = errProg ∨ existeID(ts, idDec);
                  añadeID(ts, idDec, <tipo: tipoDec, dir: dirDecs1>)}
Decs(out dirDecs) → λ
                                                {dirDecs = 1;
                                                  ts = creaTS(); }
```

```
Dec(out idDec, out tipoDec) → {var tipoTipo}
                              Tipo(tipoTipo)
                              {tipoDec = tipoTipo;
                               idDec = lex}
                              id
                              pyc

Tipo(out tipoTipo) → tInt
                     {tipoTipo = int}
Tipo(out tipoTipo) → tReal
                     {tipoTipo = real}

Accs() →
        Acc()
        Raccs()

Raccs() →
        Acc()
        Raccs()
Raccs() → λ

Acc() → {var tipoE0}
         Expr0(tipoE0)
          {errProg = (tipoE0 == error)
                emite(cod, vacia-pila());}
        pyc

Expr0(out tipoE0) → opIn
        {emite(cod, lee() || cima-dir(ts[lex].dir) ) ;
         tipoE0 = tipoDeU(in, tipoDeId(ts, id.lex))}
        id
Expr0(out tipoE0) →  {var tipoE1}
        opOut
        Expr1(tipoE1)
        {emite(cod,imprime());
         tipoE0 = tipoDeU(opOut, tipoE1)}
Expr0(out tipoE0) → {var tipoE1}
        Expr1(tipoE1)
        {tipoE0 = tipoE1}

Expr1(out tipoE1) → {var tipoE5O, tipohR4, tipohR3, tipohR2, tipoR2, tipoR3,
tipoR4 }
        Expr5O(tipoE5O)
        {tipohR4 = tipoE5O;}
        Res4(tipohR4, tipoR4)
        {tipohR3 = tipoR4;}
        Res3(tipohR3, tipoR3)
        {tipohR2 = tipoR3;}
        Res2(tipohR2, tipoR2)
```

{tipoE1 = tipoR2}
Expr1(out tipoE1) → {var tipohR1, lexhR1, tipoR1}
{tipohR1 = tipoDeId(tsh, lex);
lexhR1 = id.lex}
id
Res1(lexhR1, tipohR1, tipoR1)
{tipoE1 = tipoR1}

Res1(in lexhR1, in tipohR1, out tipoR1) → {var opO1, tipoE1}
Oper1(opO1)
Expr1(tipoE1)
{tipoR1 = tipoDeB(opO1, tipohR1, tipoE1);
emite(cod, genCod1(ts, opO1, lexhR1)}
Res1(in lexhR1, in tipohR1, out tipoR1) → {var tipohR4, tipoR4, tipohR3, tipoR3, tipohR2, tipoR2}
{emite(cod, apila-dir(ts[lexhR1].dir));
tipohR4 = tipohR1;}
Res4(tipohR4, tipoR4)
{tipohR3 = tipoR4;}
Res3(tipohR3, tipoR3)
{tipohR2 = tipoR3;}
Res2(tipohR2, tipoR2)
{tipoR1 = tipoR2;}

Oper1(out op) → **opAsig**
{op = '='}

Res2(in tipohR2, out tipoR2) → {var opO2, tipoE3}
Oper2(opO2)
Expr3(tipoE3)
{tipoR2 = tipoDeB(opO2, tipohR2, tipoE3);
emite(cod, genCod2(opO2))}
Res2(in tipohR2, out tipoR2) → λ
{tipoR2 = tipohR2;}

Oper2(out op) → **opMayor**
{op = >}
Oper2(out op) → **opMenor**
{op = <}
Oper2(out op) → **opMayIg**
{op = >=}
Oper2(out op) → **opMenIg**
{op = <=}
Oper2(out op) → **opIgual**
{op = ==}
Oper2(out op) → **opDistinto**
{op = !=}

Expr3(out tipoE3) → {var tipoE4, tipohR3, tipoR3}
Expr4(tipoE4)

{tipohR3 = tipoE4}
Res3(tipohR3, tipoR3)
{tipoE3 = tipoR3}

Res3(in tipohR30, out tipoR30) → {var opO3, tipoE4, tipohR31, tipoR31}
Oper3(opO3)
Expr4(tipoE4)
{emite(cod, genCod3(opO3));
tipohR31 = tipoDeB(opO3, tipohR30, tipoE4)}
Res3(tipohR31, tipoR31)
{tipoR30 = tipoR31;}
Res3(in tipohR3, out tipoR3) → λ
{tipoR3 = tipohR3;}

Oper3(out op) → **opSuma**
{op = +}
Oper3(op) → **opMenos**
{op = -}
Oper3(op) → **opOr**
{op = ||}

Expr4(out tipoE4) → {tipoE5, tipohR4, tipoR4}
Expr5(tipoE5)
{tipohR4 = tipoE5}
Res4(tipohR4, tipoR4)
{tipoE4 = tipoR4}

Res4(in tipohR40, out tipoR40) → {var opO4, tipoE5, tipohR41, tipoR41}
Oper4(opO4)
Expr5(tipoE5)
{emite(cod, genCod4(opO4));
tipohR41 = tipoDeB(opO4, tipohR40, tipoE5)}
Res4(tipohR41)
{tipoR40 = tipoR41}
Res4(in tipohR4, out tipoR4) → λ
{tipoR4 = tipohR4}

Oper4(out op) → **opMult**
{op = *}
Oper4(out op) → **opDiv**
{op = /}
Oper4(out op) → **opMod**
{op= %}
Oper4(out op) → **opAnd**
{op= &&}

Expr5(out tipoE5) →
{emite(cod, apila-dir(ts[lex].dir));
tipoE5 = tipoDeId(ts, lex)}
id

```
Expr5(out tipoE5) → {var tipoE5O}
        Expr5O(tipoE5O)
        {tipoE5 = tipoE5O}

Expr5O(out tipoE5O) → {var op, tipoE5}
        Oper5a(op)
        Expr5(tipoE5)
        {emite(cod, genCod5a(op));
         tipoE5O = tipoDeU(op, tipoE5)}
Expr5O(out tipoE5O) → {var op, tipoE6}
        Oper5n(op)
        Expr6(tipoE6)
        {emite(cod, genCod5n(op));
         tipoE5O = tipoDeU(op, tipoE6)}
Expr5O(out tipoE5O) → {var tipoE6O}
        Expr6O(tipoE6O)
        {tipoE5O = tipoE6O}

Oper5a(out op) → opMenos
        {op = -}
Oper5a(out op) → opNot
        {op = !}
Oper5n(out op) → opCastInt
        {op = (int)}
Oper5n(out op) → opCastReal
        {op = (real)}

Expr6(out tipoE6) →
                {tipoE6 = tipoDeId(ts, lex);
                 emite(cod, apila-dir(ts[lex].dir))}
                id
Expr6(out tipoE6) → {var tipoE6O}
        Expr6O(tipoE6O)
        {tipoE6 = tipoE6O}

Expr6O(out tipoE6O) →
                {emite(cod, apilaEnt(toInt(lex)));
                 tipoE6O = int}
                litNat
Expr6O(out tipoE6O) →
                {emite(cod, apilaReal(toReal(lex)));
                 tipoE6O = real}
                litReal
Expr6O(out tipoE6O) → {var tipoE0}
        pa
        Expr0(tipoE0)
        {tipoE6O = tipoE0}
        pc
```

10. Formato de representación del código P

Deberá describirse el formato de archivo aceptado por el intérprete de la máquina P, llamado código a pila (código P). Se valorará la eficiencia de dicho formato (por ejemplo: uso de *bytecode* binario en lugar de texto).

El intérprete de la máquina P acepta archivos de código P que contienen bytecode binario.

Los archivos son creados usando la clase ObjectOutputStream de Java. Esta clase permite escribir objetos y datos de tipos primitivos en cualquier *stream* de datos, como ficheros o sockets. Después, mediante la clase ObjectInputStream de Java, se puede leer del stream el bytecode binario y restaurar los objetos y datos escritos.

Para poder escribir las instrucciones en este formato, deben implementar la interfaz Serializable de Java. El mecanismo de serialización representa un objeto con el nombre de su clase, sus miembros, y los valores de los campos que no sean ni *static* ni *transient*.

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.

El código en sí deberá estar adecuadamente comentado, pero NO se incluirá en la memoria de la práctica. Aquí sólo debe describirse someramente qué contiene cada archivo.

A continuación se listan los paquetes en los que está organizado el código y los ficheros que contienen.

traductor.jj fichero de especificación de la gramática para el generador de JavaCC. Para generar el traductor se ha empleado la version de javaCC 5.0 sin ninguna opción adicional a las que ya se encuentran especificadas en dicho fichero.

traductor : paquete que contiene todo el código generado por javaCC. Especialmente importante la clase Traductor.java, que es la que realiza la traducción.

semantica :
FuncionesSemanticas.java: contiene las funciones semánticas utilizadas en el traductor.
TablaSimbolos.java: clase que define la estructura de datos para la tabla de símbolos usada por el traductor.
Int.java: envoltorio simple a int usado para simular el paso por referencia de enteros en el traductor.
Tipo.java : clase análoga a Int.java , pero para el enumerado TIPO de la tabla de símbolos.

maquina
MaquinaP.java: implementa una versión sencilla de la máquina P para ejecutar el código producido por el traductor.

programas
Compilador.java: contiene el programa que traduce código fuente a código P. Lee el código fuente de un fichero pasado como parámetro y, en caso de haber completado la traducción con éxito, serializa la lista de instrucciones de la máquina P obtenida y la escribe en un segundo fichero, pasado también como parámetro. Implica a las clases de los paquetes lexico, traductor y error.
Interprete.java: contiene el intérprete de la máquina P. Realiza la lectura de un fichero de código P (previamente generado por el ProgramaA) y realiza la ejecución del mismo haciendo uso de una instancia de la máquina P. Permite ejecutar en modo normal, o modo traza.