Procesadores de lenguajes

Ejercicios del Tema 7

ASPECTOS GENERALES

En los ejercicios descritos a continuación (salvo que se indique otra cosa) se considera que el código intermedio a generar estará formado por las siguientes instrucciones de código de 3 direcciones:

```
var1 = constante
var1 = var2
var1 = - var2
var1 = var2 + var3
var1 = var2 - var3
var1 = var2 * var3
var1 = var2 / var3
var1 = var2 % var3
var1 = MEM[var2 + var3]
if var1 == var2 goto etiqueta
if var1 != var2 goto etiqueta
if var1 > var2 goto etiqueta
if var1 < var2 goto etiqueta
if var1 >= var2 goto etiqueta
if var1 <= var2 goto etiqueta
goto etiqueta
```

GENERACIÓN DE CÓDIGO PARA EXPRESIONES

Ejercicio 7.1

La sintaxis de la instrucción de asignación es la siguiente:

Asignación → **id** "**=**" *Expresion* "**;**"

Expresión → *Expresión* "+" *Término*

Expresión → *Expresión* "**-**" *Término*

Expresión → *Término*

Expresión → "**-**" *Término*

Término → *Término* "*****" *Factor*

Término → *Término* "/" *Factor*

Término → *Factor*

Factor → **num**

Factor → **id**

Factor → "**(**" *Expresión* "**)**"

- Escriba un ETDS que genere el código intermedio asociado a las instrucciones de asignación. Considere que se dispone de la función *getNewTemp()*, que devuelve la referencia a una nueva variable temporal.

- Obtenga el árbol de análisis sintáctico (AAS) y el código intermedio generado por el ETDS anterior para la siguiente instrucción de asignación: "c = - a*(3 + b); "

Ejercicio 7.2

La sintaxis de la instrucción de asignación es la siguiente:

Asignación → **id** "=" *Expresion* "**;**"

Expresión → *Término ListaTérminos*

ListaTérminos → "+" *Término ListaTérminos*

ListaTérminos → "**-**" *Término ListaTérminos*

ListaTérminos → λ

Término → *Factor ListaFactores*

ListaFactores → "*" *Factor ListaFactores*

ListaFactores → "**-**" *Factor ListaFactores*

ListaFactores → λ

Factor → **num**

Factor → **id**

Factor → "**(**" *Expresión* "**)**"

- Escriba un ETDS que genere el código intermedio asociado a las instrucciones de asignación. Considere que se dispone de la función *getNewTemp()*, que devuelve la referencia a una nueva variable temporal.

Ejercicio 7.3

La siguiente figura muestra la definición de una instrucción de asignación en el formato de la herramienta JavaCC.

```
void Asignacion() :
{}
{
  <ID> <ASIG> Expresion() <PYC>
}

void Expresion() :
{}
{
   Termino()  (  (<MAS> | <MENOS>)  Termino() )*
}

void Termino() :
{}
{
  Factor()  (  (<PROD> | <DIV>)  Factor()  )*
}

void Factor() :
{}
{
   <ID>
 | <NUM>
 | <PARAB> Expresion() <PARCE>
}
```

Se pretende enriquecer la gramática anterior para construir el código intermedio de 3 direcciones correspondiente a la instrucción de asignación. Para ello, la función *Asignacion()* deberá devolver un objeto *String* que contenga dicho código.

Se asume que se dispone de las funciones *getNewTemp()* para obtener una referencia a una nueva variable temporal y *getPos()* para obtener la referencia a una variable declarada a partir de su lexema.

Se pide:

(a) Determinar los atributos que es necesario asociar a los símbolos *Expresión*, *Término* y *Factor* para construir el código intermedio.

(b) Definir una clase en Java que contenga dichos atributos.

(c) Realizar las modificaciones necesarias a la especificación en JavaCC para generar el código intermedio. Estas modificaciones requieren: (1) modificar el tipo de dato que devuelven las funciones; (2) añadir las variables necesarias; (3) asignar a estas variables los valores devueltos por los símbolos no terminales y por los tokens; (4) insertar las acciones semánticas necesarias.

NOTAS:

(1) Los tokens <ASIG>, <PYC>, <PARAB> y <PARCE> representan los símbolos “=”, “;”, “(“ y “)”, respectivamente.

(2) Los tokens <MAS>, <MENOS>, <PROD> y <DIV> se refieren a los operadores de suma, resta, multiplicación y división, y representan los símbolos "+", "-", "*" y "/", respectivamente.

(3) El token <NUM> se refiere a las constantes numéricas

(4) El token <ID> se refiere a los identificadores de variables

(5) En JavaCC, el analizador léxico devuelve los tokens como objetos de la clase *Token*. El lexema de cada token se almacena en el campo "*image*" de estos objetos.

Ejercicio 7.4

Se pretende generar el código intermedio asociado a las expresiones condicionales, considerando que los datos booleanos se representan como números (falso = 0, verdadero = 1). La siguiente figura muestra las instrucciones booleanas de un código de 3 direcciones:

```
var3 = var1 & var2
var3 = var1 | var2
var3 = ! var1
var3 = var1 == var2
var3 = var1 != var2
var3 = var1 > var2
var3 = var1 < var2
var3 = var1 <= var2
var3 = var1 >= var2
```

Considere que la sintaxis las expresiones condicionales viene dada por la siguiente gramática:

Condición → *CondiciónAnd Disyunción*

Disyuncion → "||" *CondiciónAnd Disyunción*

Disyunción → λ

CondiciónAnd → *CondiciónBase Conjunción*

Conjunción → "**&&**" *CondiciónBase Conjunción*

Conjunción → λ

CondiciónBase → *Expresión Operador Expresión*

CondiciónBase → "(" *Condición* ")"

Operador → "= ="

Operador → "**!**="

Operador → ">"

Operador → "<"

Operador → ">="

Operador → "<="

Se desea asociar al símbolo *Condición* los atributos *code*, con el código intermedio de la condición, y *temp*, con la referencia a la variable donde se almacena el resultado de la expresión. Por su parte, el símbolo *Expresión* (que se refiere a expresiones aritméticas) tiene también los atributos *code* y *temp*. Considere que se dispone de la función *getNewTemp()*, que devuelve una referencia a una nueva variable temporal (es decir, una variable temporal que no se ha utilizado hasta el momento).

Escriba un ETDS que genere el valor de los atributos de las expresiones condicionales, es decir, que genere el código intermedio y la referencia a la variable que almacena el valor de la expresión. Por ejemplo, considerando la entrada

```
a >= b && ( a<c || f == g)
```

el código intermedio a generar debe ser:

```
temp1 = a >= b
temp2 = a < c
temp3 = f == g
temp4 = temp2 | temp3
temp5 = temp1 & temp4
```

y el valor del atributo *temp* sería *temp5*.

Ejercicio 7.5

Los lenguajes C, C++ y Java disponen de un operador ternario (conocido en inglés como *hook* por la interrogación) formado por una condición y dos expresiones. Cuando la condición es cierta, se evalúa la primera expresión y se devuelve su resultado. Cuando la condición resulta falsa se evalúa la segunda expresión y se devuleve su resultado. A continuación se describe la gramática del operador en formato JavaCC.

```
void Hook() :
{}
{
   <LPAREN>  Cond()  <HOOK>
             Expr()  <COLON>
             Expr()  <RPAREN>
}
```

Se pide:

(a) Describir la estructura del código intermedio asociado a este operador.

(b) Desarrollar el ETDS que genere el código intermedio asociado al operador ternario.

Para ello se deberán tener en cuenta las siguientes consideraciones:

- Se dispone del método *getNewLabel()* para generar nuevas etiquetas.
- Se dispone del método *getNewTemp()* para generar nuevas variables temporales.
- El símbolo *Cond()* devuelve un objeto de la clase *Condition* con los siguientes campos: *code*, que contine el código intermedio de la condición; *label_true*, que contiene el nombre de la etiqueta a la que se salta cuando la condición es cierta; *label_false*, que contiene el nombre de la etiqueta a la que se salta cuando la condición es falsa.
- El símbolo *Expr()* devuelve un objeto de la clase *Expression* con los siguientes campos: *code*, que contiene el código intermedio para evaluar la expresión; *temp*, que contiene el nombre de la variable en el que se almacena el resultado de la expresión.
- El símbolo *Hook()* deberá devolver un objeto de tipo *Expression*.

GENERACIÓN DE CÓDIGO PARA CONDICIONES

Ejercicio 7.6

Considere la siguiente sintaxis para las expresiones condicionales:

Condición → *Condición* "||" *CondiciónAnd*

Condición → *CondiciónAnd*

CondiciónAnd → *CondiciónAnd* "**&&**" *CondiciónBase*

CondiciónAnd → *CondiciónBase*

CondiciónBase → *Expresión Operador Expresión*

CondiciónBase → "(" *Condición* ")"

Operador → "= ="

Operador → "**!**="

Operador → ">"

Operador → "<"

Operador → ">="

Operador → "<="

Se desea asociar al símbolo *Condición* los atributos *code* (con el código intermedio de la condición), *label_true* (nombre de la etiqueta a la que se salta con la condición cierta) y *label_false* (nombre de la etiqueta a la que se salta con la condición falsa).

Se pide:

(a) Escribir un ETDS que genere el valor de los atributos de las expresiones condicionales, es decir, que genere el código intermedio y los valores de las etiquetas.

(b) Escribir el árbol de análisis sintáctico, el código intermedio y el valor de las etiquetas para la siguiente expresión condicional:

```
a >= b && ( a<c || f == g)
```

NOTAS:

(1) Se dispone de la función *getNewLabel()*, que devuelve el nombre de una nueva etiqueta (es decir, una etiqueta que no se ha utilizado en ningún punto del código).

(2) El símbolo *Expresión* tiene dos atributos: *code*, con el código intermedio que genera el valor de la expresión; y *temp*, con la referencia a la variable donde se almacena el valor de la expresión.

Ejercicio 7.7

Considere la siguiente sintaxis LL(1) para las expresiones condicionales:

Condición → *CondiciónAnd Disyunción*

Disyuncion → "||" *CondiciónAnd Disyunción*

Disyunción → λ

CondiciónAnd → *CondiciónBase Conjunción*

Conjunción → "**&&**" *CondiciónBase Conjunción*

Conjunción → λ

CondiciónBase → *Expresión Operador Expresión*

CondiciónBase → "(" *Condición* ")"

Operador → "= ="

Operador → "**!**="

Operador → ">"

Operador → "<"

Operador → ">="

Operador → "<="

Genere un ETDS que permita obtener el código intermedio asociado a una condición teniendo en cuenta las siguientes consideraciones:

(a) Se dispone del método *getNewLabel()* para generar nuevas etiquetas.

(b) El símbolo *Expresión* tiene asociados dos atributos sintetizados: *code*, con el código intermedio que genera el valor de la expresión; y *temp*, con la referencia a la variable donde se almacena el valor de la expresión.

(c) El símbolo *Condición* debe tener asociados los siguientes atributos: *code*, atributo sintetizado con el código intermedio asociado a la condición; *label_true*, atributo heredado con el nombre de la etiqueta a la que debe saltar el código de la condición en caso de que ésta sea cierta; *label_false*, atributo heredado con el nombre de la etiqueta a la que debe saltar el código de la condición en caso de que ésta sea falsa.

Ejercicio 7.8

Se pretende construir un ETDS que permita generar el código asociado a las expresiones condicionales por medio de la herramienta JavaCC. A continuación se muestra la sintaxis de las expresiones condicionales en el formato de esta herramienta:

```
void Condicion() :
{}
{
  CondicionAnd()  (  <OR>  CondicionAnd()  ) *
}

void CondicionAnd() :
{}
{
 CondicionSimple()  ( <AND>  CondicionSimple()  )*
}

void CondicionSimple() :
{}
{
  Expresion()  (  <EQ>  |  <NE>  |  <GT>  |  <GE> |  <LT>  | <LE> )  Expresion()
 |  <PARAB>  Condicion()  <PARCE>
}
```

A tal fin, la definición de los símbolos anteriores debe modificarse para devolver objetos de la clase *Condition* y para aceptar como atributos heredados los parámetros *label_true* y *label_false* (ambos de tipo *String*). Se asume que el símbolo *Expresion* devuelve un objeto de tipo *Expression* y que se dispone de los métodos *getNewTemp()* y *getNewLabel()*. La descripción de las clases *Condition* y *Expression* es la siguiente:

```
public class Condition {
  public String code;
  public String label_true;
  public String label_false;

  public Condition() { }
}

public class Expression {
  public String code;
  public String temp;

  public Expression() { }
}
```

GENERACIÓN DE CÓDIGO PARA INSTRUCCIONES

Ejercicio 7.9

La siguiente figura muestra la descripción sintáctica de la instrucción IF-THEN-ELSE en el formato de la herramienta JavaCC.

```
void InstIf() :
{}
{
  <IF> <PARAB> Condicion() <PARCE> Instruccion()
  [ <ELSE> Instruccion() ]
}
```

Se pretende enriquecer la gramática anterior para construir el código intermedio de 3 direcciones correspondiente a la instrucción de salto condicional. Para ello, la función *InstIf()* deberá devolver un objeto *Inst* cuyo campo *code* contenga dicho código.

Considere que la función *Condicion()* devuelve un objeto de la clase *Condition* con los siguientes campos: *code*, código intermedio que describe la condición; *label_true*, etiqueta a la que salta el control en el caso de que la condición sea cierta; *label_false*, etiqueta a la que salta el control en el caso de que la condición sea falsa. Considere, asímismo, que la función *Instruccion()* devuelve un objeto de la clase *Inst* que contiene el campo *code* con el código asociado a la instrucción. Por último, se dispone del método *getNewLabel()* que devuelve una nueva etiqueta (es decir, no utilizada en ningún punto del código).

Ejercicio 7.10

La siguiente figura muestra la descripción sintáctica de la instrucción FOR (con el significado habitual en C, C++ y Java) en el formato de la herramienta JavaCC.

```
void InstFor() :
{}
{
  <FOR> <PARAB> Instruccion() <PYC> Condicion() <PYC> Instruccion() <PARCE>
   Instruccion()
}
```

Se pretende enriquecer la gramática anterior para construir el código intermedio de 3 direcciones correspondiente a la instrucción FOR. Para ello, la función *InstFor()* deberá devolver un objeto *Inst* cuyo campo *code* contenga dicho código.

Considere que la función *Condicion()* devuelve un objeto de la clase *Condition* con los siguientes campos: *code*, código intermedio que describe la condición; *label_true*, etiqueta a la que salta el control en el caso de que la condición sea cierta; *label_false*, etiqueta a la que salta el control en el caso de que la condición sea falsa. Considere, asímismo, que la función *Instruccion()* devuelve un objeto de la clase *Inst* que contiene el campo *code* con el código asociado a la instrucción. Por último, se dispone del método *getNewLabel()* que devuelve una nueva etiqueta (es decir, no utilizada en ningún punto del código).

Ejercicio 7.11

La siguiente figura muestra la descripción sintáctica de la instrucción SWITCH (con el significado habitual en C, C++ y Java) en el formato de la herramienta JavaCC.

```
void InstSwitch() :
{}
{
  <SWITCH> <PARAB> Expresion() <PARCE> <LLAVEAB>
  ( ClausulaCase() )*  ( ClausulaDefault() )? <LLAVECE>
}

void ClausulaCase() :
{}
{
  <CASE> <VALOR> <DP> ( Instruccion() )* ( InstBreak() )?
}

void ClausulaDefault() :
{}
{
  <DEFAULT> <DP> ( Instruccion() )*
}

void InstBreak() :
{}
{
  <BREAK> <PYC>
}
```

Se pretende enriquecer la gramática anterior para construir el código intermedio de 3 direcciones correspondiente a la instrucción SWITCH. Para ello, la función *InstSwitch()* deberá devolver un objeto *Inst* cuyo campo *code* contenga dicho código. De igual modo, las funciones *ClausulaCase()*, *ClausulaDefault()* e *InstBreak()* deberán devolver un objeto *Inst* cuyo campo *code* contenga el código asociado a estos bloques.

Considere que la función *Expresión()* devuelve un objeto de la clase *Expr* con los campos *code*, que contiene el código intermedio que describe la expresión, y *temp*, con la referencia a la variable en que se almacena el valor de la expresión. Considere, asímismo, que la función *Instruccion()* devuelve un objeto de la clase *Inst* que contiene el campo *code* con el código asociado a la instrucción. Por último, se dispone del método *getNewLabel()* que devuelve una nueva etiqueta (es decir, no utilizada en ningún punto del código) y del método *getNewTemp()* que devuelve una referencia a una nueva variable temporal.

Ejercicio 7.12

La siguiente figura muestra la descripción sintáctica del bucle REPEAT-UNTIL en el formato de la herramienta JavaCC.

```
void InstRepeat() :
{}
{
  <REPEAT> <LLAVEAB> ( Instruccion() )* <LLAVECE>
  <UNTIL> <PARAB> Condicion() <PARCE> <PYC>
}
```

Se pretende enriquecer la gramática anterior para construir el código intermedio de 3 direcciones correspondiente a la instrucción REPEAT. Para ello la función *InstRepeat()* deberá devolver un objeto *Inst* cuyo campo *code* contenga dicho código.

Considere que la función *Condicion()* devuelve un objeto de la clase *Condition* con los siguientes campos: *code*, código intermedio que describe la condición; *label_true*, etiqueta a la que salta el control en el caso de que la condición sea cierta; *label_false*, etiqueta a la que salta el control en caso de que la condición sea falsa. Por su parte, la función *Instruccion()* devuelve un objeto de la clase *Inst* cuyo campo *code* contiene el código intermedio asociado a a dicha instrucción. Asímismo, se dispone del método *getNewLabel()*, que devuelve una nueva etiqueta (es decir, una etiqueta no utilizada en ningún punto del código) y del método *getNewTemp()* que devuelve una referencia a una nueva variable temporal.

Nota: el funcionamiento del bucle REPEAT consiste en ejecutar la lista de instrucciones y a continuación evaluar la condición. En caso de que la condición sea cierta se sale del bucle. Si la condición es falsa se repite el proceso ejecutando la lista de instrucciones y evaluando de nuevo la condición.

Ejercicio 7.13

La siguiente figura muestra la descripción sintáctica del bucle FOR-NEXT en el formato de la herramienta JavaCC.

```
void InstForNext() :
{}
{
  <FOR> <ID> <EQ> <NUM> <TO> <NUM> ( <STEP> <NUM> )?
   ( Instruccion() )* <NEXT>
}
```

Se pretende enriquecer la gramática anterior para construir el código intermedio de 3 direcciones correspondiente a la instrucción FOR-NEXT. Para ello la función *InstForNext()* deberá devolver un objeto *Inst* cuyo campo *code* contenga dicho código.

Considere que la función *Instruccion()* devuelve un objeto de la clase *Inst* cuyo campo *code* contiene el código intermedio asociado a a dicha instrucción. Se dispone del método *getNewLabel()*, que devuelve una nueva etiqueta (es decir, una etiqueta no utilizada en ningún punto del código) y del método *getNewTemp()* que devuelve una referencia a una nueva variable temporal.

Nota: Ejemplo de instrucción FOR-NEXT

```
FOR I = 1 TO 10 STEP 3
PRINT I
NEXT
```

Ejercicio 7.14

La siguiente figura muestra la descripción sintáctica de las instrucciones WHILE, BREAK y CONTINUE en el formato de la herramienta JavaCC.

```
void WhileInst() :
{}
{
  <WHILE> <PARAB> Condicion() <PARCE> Instruccion()
}

void BreakInst() :
{}
{
  <BREAK> <PYC>
}

void ContinueInst() :
{}
{
  <CONTINUE> <PYC>
}

void BlockInst() :
{}
{
  <LLAVEAB>  ( Instruccion() )*  <LLAVECE>
}

void Instruccion() :
{}
{
   WhileInst()
 | BreakInst()
 | ContinueInst()
 | BlockInst()
 | OtraInst()
}
```

Considere que la función *Condicion()* devuelve un objeto de la clase *Condition* con los siguientes campos: *code*, código intermedio que describe la condición; *label_true*, etiqueta a la que salta el control en el caso de que la condición sea cierta; *label_false*, etiqueta a la que salta el control en caso de que la condición sea falsa. Por su parte, la función *OtraInst()* devuelve un objeto de la clase *Inst* cuyo campo *code* contiene el código intermedio asociado a dicha instrucción. Asímismo, se dispone del método *getNewLabel()*, que devuelve una nueva etiqueta (es decir, una etiqueta no utilizada en ningún punto del código).

Se pretende enriquecer la gramática anterior para construir el código intermedio de 3 direcciones correspondiente a las instrucciones WHILE, BREAK y CONTINUE, así como a los bloques de instrucciones. Para ello las funciones *WhileInst()*, *BreakInst()*, *ContinueInst()*, *BlockInst()* e *Instruccion()* deberán devolver un objeto *Inst* cuyo campo *code* contenga el correspondiente código intermedio.

Ejercicio 7.15

La siguiente figura muestra la descripción sintáctica de la instrucción SWITCH (con el significado habitual en C, C++ y Java) en el formato de la herramienta JavaCC.

```
void InstSwitch() :
{}
{
  <SWITCH> <PARAB> Expresion() <PARCE> <LLAVEAB>
  ( Clausula() )*  <LLAVECE>
}

void Clausula() :
{}
{
   <CASE> Value() <DP> ( Instruccion() )*
 | <DEFAULT> <DP> ( Instruccion() )*
}
```

(a) Describa la estructura del código intermedio asociado a esta instrucción

(b) Modifique la gramática para que la función *InstSwitch()* devuelva un objeto *Inst* cuyo campo *code* contenga dicho código.

Tenga en cuenta las siguientes consideraciones:

(a) La función *Expresión()* devuelve un objeto de la clase *Expr* con los campos *code*, que contiene el código intermedio que describe la expresión, y *temp*, con la referencia a la variable en que se almacena el valor de la expresión.

(b) La función *Instruccion()* devuelve un objeto de la clase *Inst* que contiene el campo *code* con el código asociado a la instrucción.

(c) La función *Value()* devuelve un entero (int) con el valor de la constante que aparece en la cláusula *case*.

(d) Se dispone del método *getNewLabel()* que devuelve una nueva etiqueta (es decir, no utilizada en ningún punto del código)

(e) Se dispone del método *getNewTemp()* que devuelve una referencia a una nueva variable temporal.

(f) Se dispone del método *pushBreakLabel(String)* que apila la etiqueta a la que deben saltar las instrucciones break.

(g) Se dispone del método *popBreakLabel()* que desapila la etiqueta a la que deben saltar las instrucciones break.

(h) Se dispone de la clase *CaseDefaultClause* con el siguiente código:

```
public class CaseDefaultClause {

  public boolean isDefault;

  public int caseValue;

  public Inst[] instBlock;

  public CaseDefaultClause(int caseValue) { ... } // Cláusula case

  public CaseDefaultClause() { ... } // Cláusula default

  public void addInstruction(Inst inst) { ... } // Añade una instrución
}
```

Ejercicio 7.16

El lenguaje C# introduce una instrucción, llamada *foreach*, que desarolla un bucle sobre los elementos de un objeto *Collection*. A continuación se muestra un ejemplo de uso de esta instrucción:

```
double[] lista = { 1.0, 3.0, 5.0, 7.0 };
foreach( double elem in lista ) {
  System.out.println("* "+elem);
}
```

El comportamiento de este bucle es similar al de un bucle ***for*** que realizara un recorrido sobre los elementos de la lista:

```
double[] lista = { 1.0, 3.0, 5.0, 7.0 };
for(int i=0; i<lista.length; i++) {
  double elem = lista[i];
  System.out.println("* "+elem);
}
```

La siguiente figura muestra la descripción sintáctica de la instrucción ***foreach*** en el formato de la herramienta JavaCC:

```
void InstForeach() :
{}
{
  <FOREACH> <PARAB> Tipo() <ID> <IN> Expresion() <PARCE>
  Instruccion()
}
```

Se pide:

(a) Describir la estructura del código intermedio asociado al bucle ***foreach***.

(b) Modificar la gramática anterior para construir el código intermedio de 3 direcciones correspondiente a esta instrucción. Para ello, la función *InstForeach()* deberá devolver un objeto *Inst* cuyo campo *code* contenga dicho código.

Para ello se deberán tener en cuenta las siguientes consideraciones:

- La función *Tipo()* devuelve un valor entero con el tamaño en bytes del tipo de datos de la iteración (por ejemplo, *int* ocupa 4 bytes y *double* ocupa 8 bytes).

- La función *Expresion()* devuelve un objeto de la clase *Expression* con los siguientes campos: *code*, código intermedio que describe la expresión; *temp*, nombre de la variable donde se almacena el resultado de la expresión. Dado que la expresión es de tipo matricial, esta variable contendrá la dirección de memoria donde comienza la matriz (*base*).

- Se dispone de los métodos *getNewLabel()* para generar nuevas etiquetas, *getNewTemp()* para generar nuevas variables y *addTemp(String name)* para añadir una variable a la tabla de símbolos.

- La dirección base de un array contiene el número de elementos del array (*MEM[base]*). El primer elemento del array se encuentra en *MEM[base+4]*. El siguiente elemento se encuentra en *MEM[base+4+tamaño]*. El elemento i-ésimo se encuentra en *MEM[base+4+i*tamaño]*.

Ejercicio 7.17

La siguiente figura muestra una gramática que describe las referencias a variables, campos o llamadas a métodos de un lenguaje orientado objetos que no dispone del tipo de datos matricial (*arrays*).

Referencia → **this** *SigueReferencia*

Referencia → **id** *LlamadaOpcional SigueReferencia*

SigueReferencia → **punto id** *LlamadaOpcional SigueReferencia*

SigueReferencia → λ

LlamadaOpcional → **parab** *ListaDeExpresiones* **parce**

LlamadaOpcional → λ

Para desarrollar el Árbol de Sintaxis Abstracta de esta gramática se dispone de las clases listadas en la siguiente página.

Se dispone además de la función *getVariable(String name)* que devuelve un objeto *VariableReference* a partir del nombre de una variable local o del identificador "*this*". Se asume también que el símbolo *ListaDeExpresiones* devuelve como atributo sintetizado una matriz de objetos *Expression* que puede ser utilizada en el constructor de *MethodReference*.

El objetivo de este ejercicio es introducir el tipo de datos matricial (*array*), lo que conlleva modificaciones de la gramática tanto a nivel sintáctico como a nivel semántico (estructura del árbol de sintaxis abstracta y verificaciones semánticas). A continuación se muestra una lista de ejemplos de referencias a variables, campos y llamadas a funciones en los que se contempla el tipo de datos matricial.

```
pos               // referencia a la variable pos
lista[5]          // referencia al 6º elemento de lista
lista[2*i+1]      // referencia a un elemento de lista
matriz[4][5]      // referencia al 6º elem. del 5º elem. de matriz
sqrt(2.0)         // resultado del método sqrt()
this              // referencia al objeto
this.x            // referencia al campo x del objeto
this.x[5]         // referencia al 6º elem. del campo x del objeto
point.y           // referencia al campo y del objeto point
line.paint()      // referencia al método paint del objeto line
panel.getPosition().x    // referencia al campo x del resultado
                         // del método getPosition del objeto panel
panel.crearListaDePuntos(10)[0].y
  // referencia al campo y del primer elem. del resultado del método
  // crearListaDePuntos del objeto panel
```

Se pide:

(a) Modificar la gramática descrita en el enunciado para considerar el acceso a los elementos en las variables y campos matriciales.

```
public abstract class Reference {
  public Type getType();
}

public class VariableReference extends Reference {
  private String name;
  private Type type;

  public String getName() { ... }
  public Type getType() { ... }
}

public class FieldReference extends Reference {
  private String name;
  private Type type;
  private Reference object;

  public FieldReference(String name, Reference object) { ... }
  public Type getType() { ... }
  public Reference getObject() { ... }
}

public class MethodReference extends Reference {
  private Method method;
  private Type type;
  private Reference object;
  private Expression[] param;

  public MethodReference(String name, Expression[] param,
                         Reference object) { ... }
  public Type getType() { ... }
}

public class Expression {
  public Type getType() { ... }
}

public class Type {
  public static final int INTEGER = 0;
  public static final int FLOAT = 1;
  public static final int CHAR = 2;
  public static final int BOOLEAN = 3;
  public static final int VOID = 4;
  public static final int REFERENCE = 5;
  public static final int ARRAY = 6;

  private int code;
  private ClasDeclaration decl;
  private Type arrayof;

  public int getCode() { return this.code; }
  public ClassDeclaration getClassDecl() { return this.decl; }
  public Type getArrayType() { return this.arrayof; }
}
```

(b) Definir la estructura de datos (es decir, la clase) necesaria para almacenar la información referida al acceso a elementos de un tipo matricial. (La clase *Type* ya ha sido modificada para incluir la información acerca de tipos matriciales).

(c) Enumerar los atributos (heredados y sintetizados) de cada símbolo de la gramática

(d) Añadir a la gramática las acciones semánticas necesarias para obtener la estructura de datos que describa estas referencias.

(e) Describir las verificaciones semánticas que es necesario realizar para incorporar los elementos matriciales. No se pide programar los método de verificación, sino tan sólo explicar que verificaciones semánticas son necesarias.

EJEMPLO:

Se considera la entrada "*figura.getComponentAt(10,20).height*" donde:

- "*figura*" es una variable de tipo "*Panel*".
- *getComponentAt(int,int)* es un método de la clase *Panel* que devuelve un dato de tipo *Component.*
- *Component* es una clase que contiene un campo llamado "*height*" de tipo entero.

El árbol de sintaxis abstracta que describe esta entrada es el siguiente: