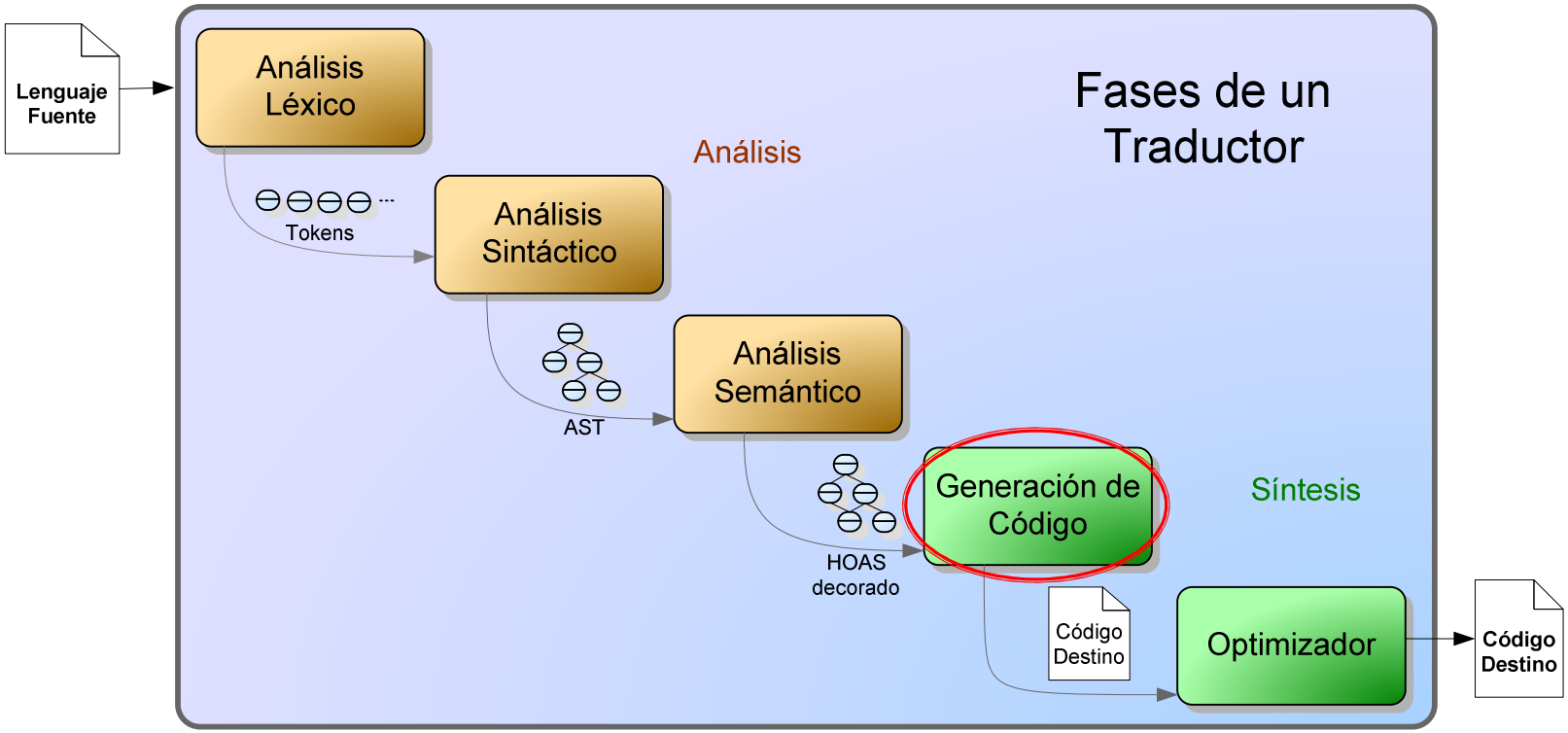
Generación de Código (II)

Diseño de Lenguajes de Programación
Ingeniería Informática
Universidad de Oviedo
(v1.6)

Raúl Izquierdo Castanedo

Usted está aquí…



Repaso

La fase de Generación de Código se encarga de determinar...

- Cómo se representan los Datos
  - Representar los tipos abstractos de alto nivel con las limitaciones de la plataforma de destino
- Cómo se traduce el Código
  - Traducir las sentencias de alto nivel a secuencias de instrucciones de la plataforma de destino

Subfases:

- Gestión de Memoria ✓
- Selección de Instrucciones ⇦

Selección de Instrucciones

Plataformas (I)

Objetivo

- Traducir las sentencias de alto nivel a secuencias de instrucciones de la *plataforma de destino*

Tipos de plataformas de destino:

- Máquinas Reales
- Máquinas Abstractas
  - Requieren Máquinas Virtuales

Plataformas (II)

Ventajas e inconvenientes

- Por facilidad de introducción de
  - Nuevos lenguajes
  - Nuevas plataformas
- Portabilidad
- Eficiencia
  - Por forma de ejecución
  - Por pérdida de contexto

Criterio para elegir un tipo de plataforma u otra

- Los requisitos de eficiencia
  - Que con el tiempo van cambiando

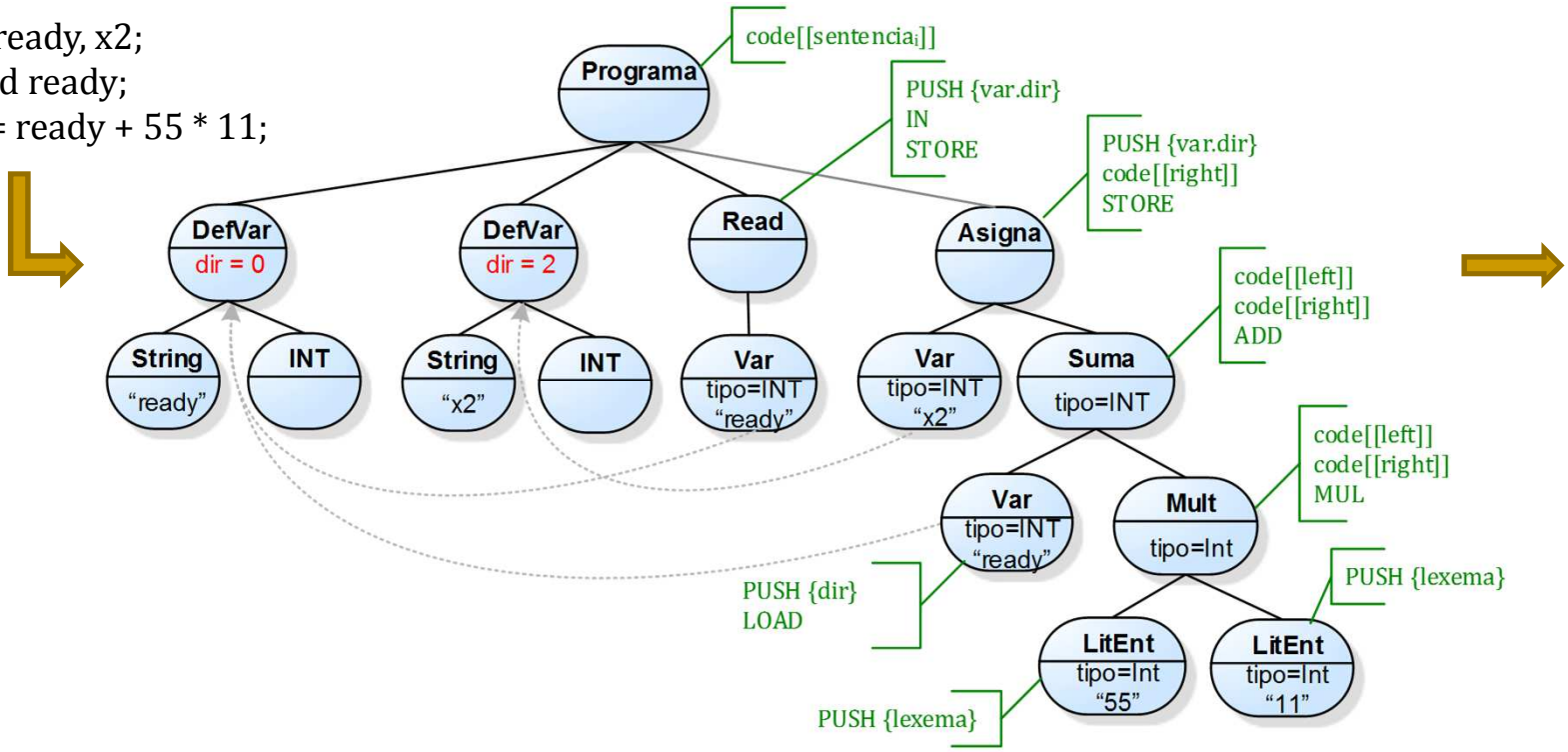
Selección de Instrucciones
Cómo se hace

¿*Cómo* se hace?

Generación Inductiva

- El código generado para un programa se obtiene mediante la unión del código generado por sus estructuras

```
int ready, x2;
Read ready;
x2 = ready + 55 * 11;
```




```
PUSH 0
IN
STORE

PUSH 2
PUSH 0
LOAD
PUSH 55
PUSH 11
MUL
ADD
STORE
```

¿*Cómo* se especifica?

Se necesita indicar qué instrucciones hay que generar para cada construcción del lenguaje

- ¿Notación?
  - ¿Lenguaje natural?

Metalenguaje
Especificación de Código

Especificación de Código

Definición de Especificación de Código

- Conjunto de *funciones de código* que especifican la traducción al lenguaje destino de todo programa válido

Función de Código

- Dada una categoría sintáctica ***P*** de la sintaxis abstracta del lenguaje de entrada, la función de código f_p asocia a dicha categoría un conjunto de instrucciones del lenguaje destino

$$\boldsymbol{f}_p : \boldsymbol{P} \rightarrow \mathit{Instrucción}*$$

Elementos de una Función de Código

1) Nombre

direccion[[expr]]
valor[[expr]]

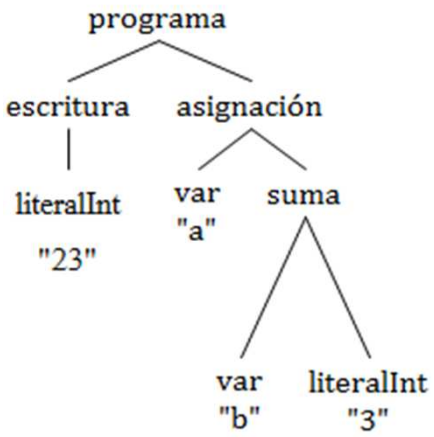
2) Conjunto de Plantillas de Código
   - Una por cada nodo de la categoría
     - Especifica el código al que hay que traducir la estructura (nodo)
       - Normalmente en términos del código obtenido en la traducción de sus subestructuras (hijos)

Ejemplo E1. Creación de una especificación

Dado el siguiente lenguaje

- Programa de ejemplo

```
print 23
a = b + 3
```



- Gramática Abstracta

programa → sentencia*

escritura:sentencia → expr
asignación:sentencia → left:expr right:expr

literalInt:expr → lexema:string
var:expr → nombre:string
suma:expr → left:expr right:expr

Las variables no se definen antes de usarse

El lenguaje solo tiene tipo ***int***

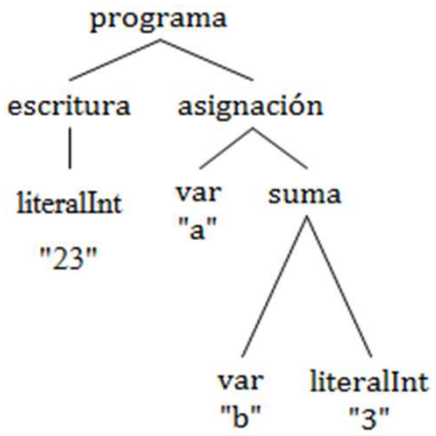
Hacer la especificación de código

Ejemplo E2. Uso de una especificación

Aplicar la especificación de código anterior

- Entrada

```
print 23;
a = b + 3;
```



- Debería salir

```
PUSH 23
OUT
PUSHA 0
PUSHA 2
LOAD
PUSH 3
ADD
STORE
```

Notación de las plantillas de código

Notación (I)

¿Qué puede aparecer dentro de una plantilla de código?

- No hay una notación estándar

Se seguirá la siguiente notación

- La plantilla genera su contenido *literalmente*

```
CALL func.nombre        // Genera "CALL func.nombre" (no  se evalúa)
```

- Evaluación de expresiones

```
CALL { func.nombre }    // Genera "CALL f" (suponiendo que f es el nombre)
```

- Llamadas a funciones de código

```
función[[nodo]]         // Genera las instrucciones de dicha función
```

Notación (II)

Se seguirá la siguiente notación (II)

- Pseudocódigo *si/sino*

valor [[**exprLogica** → *left*:expresion *operador*:string *right*:expresion]] =
    valor[[left]]
    valor[[right]]
    Si operador == "&&"
        AND
    Si operador == "||"
        OR

- Instrucciones *genéricas*
  - Abreviatura para la siguiente estructura habitual

instrucción<expresión_de_tipo>

Si *expr_de_tipo* == *tipo_entero*
    *instrucción***I**
sino si *expr_de_tipo* == *tipo_real*
    *instrucción***F**
sino si *expr_de_tipo* == *tipo_byte*
    *instrucción***B**

Si tipo == TipoInt
    ADDI
sino si tipo == TipoReal
    ADDF

ADD<tipo>

Si var.definición.tipo == TipoInt
    LOADI
sino si var.definición.tipo == TipoReal
    LOADF
sino si var.definición.tipo == TipoByte
    LOADB

LOAD<var.definición.tipo>

Tarea obligatoria

Antes de la siguiente clase de *prácticas*:

1) Bajar MAPL del Campus
2) Leer "*Manual MAPL.pdf*"
   - Se debería haber hecho ya
3) Seguir todos los ejemplos de las carpetas:
   - "*Tutorial\3 Uso del depurador*"
   - "*Tutorial\4 Metadatos*"

OBLIGATORIO

Ejercicio E3

programa → nombre:string definiciones:defVariable* sentencias:sentencia*

defVariable → tipo nombre:string
intType:tipo → λ
realType:tipo → λ

Nótese que el lenguaje tiene ahora dos tipos

print:sentencia → expresion
read:sentencia → variable
if:sentencia → condicion:expresion siCierto:sentencia* siFalso:sentencia*

literalInt:expresion → lexema:string
literalReal:expresion → lexema:string
variable:expresion → nombre:string
exprAritmetica:expresion → left:expresion operator:string right:expresion

Implementación de una Especificación de Código

Especificaciones de Código. Implementación (I)

Traslación a código de una especificación

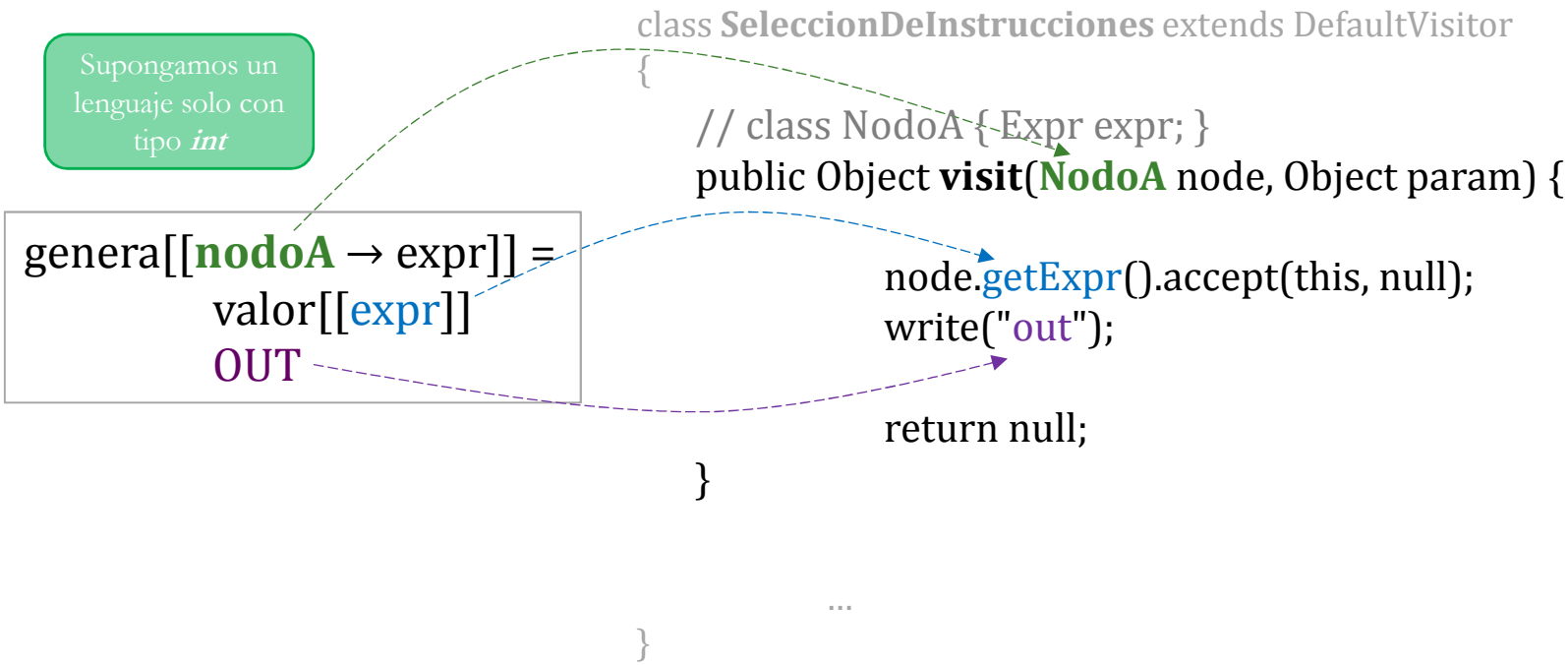
- Hay que implementar un recorrido del árbol que, en función del nodo actual, genere las instrucciones que correspondan
  - Visitor parametrizado con stream de salida

```
public class SeleccionDeInstrucciones extends DefaultVisitor
{
        private PrintWriter writer;

        public SeleccionDeInstrucciones(Writer writer) {
                this.writer = new PrintWriter(writer);
        }

        private void write(String text) {       // Opcional
                writer.println(text);
        }

        ...
}
```

Plantillas de Código. Implementación (II)

Implementación de una plantilla de código

- Una plantilla de código determina
  - El orden de recorrido de los hijos
  - Las instrucciones que deben generarse



Plantillas de Código. Implementación (III)

¿Cómo se hace si hay alguna categoría sintáctica con más de una función de código?

Code Function	Code Templates
run[[programa]]	…
…	…
valor[[expr]]	valor[[var → nombre:string]] = PUSHA {var.definición.dir} LOAD …
dirección[[expr]]	dirección[[var → nombre:string]] = PUSHA {var.definición.dir} …

```
public Object visit(Var node, Object param) {
   write("pusha " + node.getDefinicion().getDireccion());
   write("load");
   return null;
}
```

```
public Object visit(Var node, Object param) {
   write("pusha " + node.getDefinicion().getDireccion());
   return null;
}
```

Error

- ¿Solución?

Plantillas de Código. Implementación (IV)

Solución 1: un método *visit* por *nodo*

- El método *visit* implementa *todas* las plantillas del nodo
- El padre, al recorrer el hijo, deberá indicar qué función le quiere aplicar

Code Function	Code Templates
run[[programa]]	…
ejecuta[[sentencia]]	ejecuta[[asignación → left:expr right:expr]] = dirección[[left]] valor[[right]] STORE …
valor[[expr]]	valor[[var → nombre:string]] = PUSHA {var.definicion.dir} LOAD …
dirección[[expr]]	dirección[[var → nombre:string]] = PUSHA {var.definicion.dir} …

```
enum Funcion { VALOR, DIRECCION }

class SeleccionDeInstrucciones … {

    public Object visit(Asignacion node, Object param) {
      node.getLeft().accept(this, Funcion.DIRECCION);
      node.getRight().accept(this, Funcion.VALOR);
      write("store");
      return null;
    }

    public Object visit(Variable node, Object param) {
      if ( (Funcion)param == Funcion.VALOR) {
        write("pusha " + node.getDefinicion().getDireccion());
        write("load");
      } else  // Funcion.DIRECCION
        write("pusha " + node.getDefinicion().getDireccion());
      return null;
    }
  …
}
```

Plantillas de Código. Implementación (V)

Solución 2: un método *visit* por *plantilla*

- Por cada función de código habrá un Visitor con todas sus plantillas

```
class SeleccionDeInstrucciones ...
{
    private Visitor valorVisitor = new ValorVisitor();          // Plantillas de valor[[expr]]
    private Visitor direcciónVisitor = new DirecciónVisitor();  // Plantillas de dirección[[expr]]

    ...

    // class Print { Expr expr; }
    public Object visit(Print node, Object param) {
        node.getExpr().accept(valorVisitor, null);
        write("out");
    }

    // class Asignación { Expr left; Expr right; }
    public Object visit(Asignación node, Object param) {
        node.getLeft().accept(direcciónVisitor, null);
        node.getRight().accept(valorVisitor, null);
        write("store");
        return null;
    }
    ...
}
```

ejecuta[[print → expr]] =
    valor[[expr]]
    OUT

Sustituye a *this*

ejecuta[[asignación
 → left:expr right:expr]] =
    dirección[[left]]
    valor[[right]]
    STORE

Ejemplo E4

El lenguaje solo tiene tipo *int*

Code Function	Code Templates
run[[programa]]	run[[programa → sentencia*]] = ejecuta[[sentencia$_i$]]
ejecuta[[sentencia]]	ejecuta[[escritura → expr]] = valor[[expr]] OUT ejecuta[[asignación → left:expr right:expr]] = dirección[[left]] valor[[right]] STORE
valor[[expr]]	valor[[literalInt → lexema:string]] = PUSH {lexema} valor[[var → nombre:string]] = dirección[[var]] LOAD valor[[suma → left:expr right:expr]] = valor[[left]] valor[[right]] ADD
dirección[[expr]]	dirección[[var → nombre:string]] = PUSHA {var.definicion.dir}

```
public class SeleccionDeInstrucciones extends DefaultVisitor {
    private PrintWriter writer;
    public SeleccionDeInstrucciones(Writer writer, String sourceFile) {
        this.writer = new PrintWriter(writer);
    }

    private void write(String text) {
        writer.println(text);
    }

    // class Programa { List<Sentencia> sentencia; }
    public Object visit(Programa node, Object param) {
        for (Sentencia child : node.getSentencia())
            child.accept(this, null);
        return null;
    }
    // class Escritura { Expresion expresion; }
    public Object visit(Escritura node, Object param) {
        node.getExpresion().accept(this, Funcion.VALOR);
        write("out");
        return null;
    }
    // class Asignacion { Expresion left;  Expresion right; }
    public Object visit(Asignacion node, Object param) {
        node.getLeft().accept(this, Funcion.DIRECCION);
        node.getRight().accept(this, Funcion.VALOR);
        write("store");
        return null;
    }
    // class LiteralInt { String lexema; }
    public Object visit(LiteralInt node, Object param) {
        write("push " + node.getLexema());
        return null;
    }
    // class Variable { String nombre; }
    public Object visit(Variable node, Object param) {
        if ( (Funcion)param == Funcion.VALOR) {
            visit(node, Funcion.DIRECCION);
            write("load");
        } else  // Funcion.DIRECCION
            write("pusha " + node.getDefinicion().getDireccion());
        return null;
    }
    // class Suma { Expresion left;  Expresion right; }
    public Object visit(Suma node, Object param) {
        node.getLeft().accept(this, Funcion.VALOR);
        node.getRight().accept(this, Funcion.VALOR);
        write("add");
        return null;
    }
}
```

Ejercicio E5. Implementar E3

Función de Código	Plantillas de Código
run: **programa** → Instruccion*	run[[**programa** → *definiciones*:defVariable* *sentencias*:sentencia*]] = ejecuta[[sentencias$_i$]]
ejecuta : **sentencia** → Instruccion*	ejecuta [[**print** → *expresion*:expresion]] = valor[[expresion]] OUT$_{<expresion.tipo>}$ ejecuta [[**if** → *condicion*:expresion *siCierto*:sentencia* *siFalso*:sentencia*]] = valor[[condicion]] JZ else{n} / *n sea un entero distinto en cada aplicación de la plantilla* ejecuta[[siCierto$_i$]] JMP finElse{n} else{n}: ejecuta[[siFalso$_i$]] finElse{n}:
valor : **expresion** → Instruccion*	valor [[**literalInt** → *lexema*:string]] = PUSH {lexema} valor [[**variable** → *nombre*:string]] = PUSHA {variable.definición.dirección} LOAD$_{<variable.tipo>}$ valor [[**exprAritmetica** → *left*:expresion *operator*:string *right*:expresion]] = valor[[left]] valor[[right]] si operator == "+" ADD$_{<exprAritmetica.tipo>}$ si operator == "-" SUB$_{<exprAritmetica.tipo>}$ si operator == "*" MUL$_{<exprAritmetica.tipo>}$ si operator == "/" DIV$_{<exprAritmetica.tipo>}$

Ejercicio E5. Estructura de la solución (I)

Función de Código	Plantillas de Código
run: **programa** → Instruccion*	run[[**programa** → *definiciones*:defVariable* *sentencias*:sentencia*]] = ejecuta[[sentencias$_i$]]
ejecuta : **sentencia** → Instruccion*	ejecuta [[**print** → *expresion*:expresion]] = valor[[expresion]] OUT$_{<expresion.tipo>}$ ejecuta [[**if** → *condicion*:expresion *siCierto*:sentencia* *siFalso*:sentencia*]] = valor[[condicion]] JZ else{n} / *n distinto* ejecuta[[siCierto$_i$]] JMP finElse{n} else{n}: ejecuta[[siFalso$_i$]] finElse{n}:

```
public class SelecciónDeInstrucciones extends DefaultVisitor {
    private PrintWriter writer;

    public SeleccionDeInstrucciones(Writer writer, String sourceFile) {
        this.writer = new PrintWriter(writer);
    }

    private void write(String text) {
        writer.println(text);
    }
```

```
// class Programa { List<DefVariable> definiciones;
//       List<Sentencia> sentencias; }
public Object visit(Programa node, Object param) {
    for (Sentencia child : node.getSentencias())
        child.accept(this, null);

    return null;
}

// class Print { Expresion expresion; }
public Object visit(Print node, Object param) {
    node.getExpresion().accept(this, null);


    return null;
}

// class If { Expresion condicion;  List<Sentencia> siCierto;
//       List<Sentencia> siFalso; }
public Object visit(If node, Object param) {
    node.getCondicion().accept(this, null);


    for (Sentencia child : node.getSiCierto())
        child.accept(this, null);



    for (Sentencia child : node.getSiFalso())
        child.accept(this, null);


    return null;
}
```

Ejercicio E5. Estructura de la solución (II)

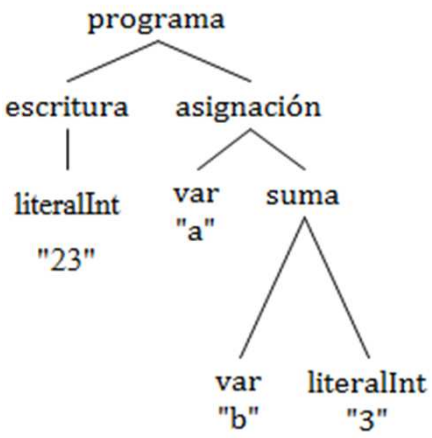
Función de Código	Plantillas de Código
valor : **expresion** → Instruccion*	valor [[**literalInt** → *lexema*:string]] = PUSH {lexema} valor [[**variable** → *nombre*:string]] = PUSHA {variable.definición.dirección} LOAD$_{\text{<variable.tipo>}}$ valor [[**exprAritmetica** → *left*:expresion *operator*:string *right*:expresion]] = valor[[left]] valor[[right]] si operator == "+" ADD$_{\text{<exprAritmetica.tipo>}}$ si operator == "-" SUB$_{\text{<exprAritmetica.tipo>}}$ si operator == "*" MUL$_{\text{<exprAritmetica.tipo>}}$ si operator == "/" DIV$_{\text{<exprAritmetica.tipo>}}$

```
    // class LiteralInt { String lexema; }
    public Object visit(LiteralInt node, Object param) {

        return null;
    }

// class Variable { String nombre; }
    public Object visit(Variable node, Object param) {


        return null;
    }

// class ExprAritmetica { Expresion left;  String operator;
       Expresion right; }
   public Object visit(ExprAritmetica node, Object param) {
        node.getLeft().accept(this, null);
        node.getRight().accept(this, null);









    }
}
```

Soluciones

Solución E1

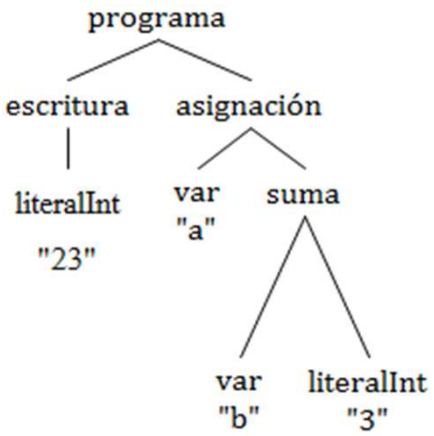
Code Function	Code Templates
run[[programa]]	run[[programa → sentencia*]] = ejecuta[[sentencia$_i$]]
ejecuta[[sentencia]] *Instrucciones que ejecutan un efecto lateral (E/S o modificación de variables) pero al acabar no han añadido ni quitado nada de la pila*	ejecuta[[escritura → expr]] = valor[[expr]] OUT ejecuta[[asignación → left:expr right:expr]] = dirección[[left]] valor[[right]] STORE
valor[[expr]] *Instrucciones que dejan un valor en la pila*	valor[[literalInt → lexema:string]] = PUSH {lexema} valor[[var → nombre:string]] = dirección[[var]] LOAD valor[[suma → left:expr right:expr]] = valor[[left]] valor[[right]] ADD
dirección[[expr]] *Instrucciones que dejan una dirección en la pila*	dirección[[literalInt → lexema:string]] = *error* dirección[[var → nombre:string]] = PUSHA {var.definicion.dir} dirección[[suma → left:expr right:expr]] = *error*



Solución E2

Aplicar la especificación de código anterior

print 23;
a = b + 3;



PUSH 23
OUT
PUSHA 0
PUSHA 2
LOAD
PUSH 3
ADD
STORE

run[[programa]]
- ejecuta[[escritura]]
  - valor[[literalInt]] → PUSH 23
  - OUT
- ejecuta[[asignación]]
  - dirección[[var]] → PUSHA 0
  - valor[[suma]]
    - valor[[var]] → dir[[var]] = PUSHA 2, LOAD
    - valor[[literalInt]] → PUSH 3
    - ADD
  - STORE

Solución E3

Función de Código	Plantillas de Código
run[[**programa**]]	run[[**programa** → *definiciones*:defVariable* *sentencias*:sentencia*]] = metadatos[[definiciones$_i$]] // Opcional ejecuta[[sentencias$_i$]]
metadatos[[**defVariable**]]	metadatos[[**defVariable** → *tipo*:tipo *nombre*:string]] = #GLOBAL {nombre}: {tipo}
ejecuta : **sentencia** → Instruccion*	ejecuta [[**print** → *expresion*:expresion]] = valor[[expresion]] OUT$_{\text{<expresion.tipo>}}$ ejecuta [[**read** → *variable*:variable]] = PUSHA { variable.definición.dirección } IN$_{\text{<variable.definición.tipo>}}$ STORE$_{\text{<variable.definición.tipo>}}$ ejecuta [[**if** → *condicion*:expresion *siCierto*:sentencia* *siFalso*:sentencia*]] = valor[[condicion]] JZ else{n} / *n sea un entero distinto en cada aplicación de la plantilla* ejecuta[[siCierto$_i$]] JMP finElse{n} else{n}: ejecuta[[siFalso$_i$]] finElse{n}:
valor : **expresion** → Instruccion*	valor [[**literalInt** → *lexema*:string]] = PUSH {lexema} valor [[**literalReal** → *lexema*:string]] = PUSHF {lexema} valor [[**variable** → *nombre*:string]] = PUSHA {variable.definición.dirección} LOAD$_{\text{<variable.tipo>}}$ valor [[**exprAritmetica** → *left*:expresion *operator*:string *right*:expresion]] = valor[[left]] valor[[right]] si operator == "+" ADD$_{\text{<exprAritmetica.tipo>}}$ si operator == "-" SUB$_{\text{<exprAritmetica.tipo>}}$ si operator == "*" MUL$_{\text{<exprAritmetica.tipo>}}$ si operator == "/" DIV$_{\text{<exprAritmetica.tipo>}}$

Solución E5 (I)

Función de Código	Plantillas de Código
run: **programa** → Instruccion*	run[[**programa** → *definiciones*:defVariable* *sentencias*:sentencia*]] = ejecuta[[sentencias$_i$]]
ejecuta : **sentencia** → Instruccion*	ejecuta [[**print** → *expresion*:expresion]] = valor[[expresion]] OUT$_{<expresion.tipo>}$ ejecuta [[**if** → *condicion*:expresion *siCierto*:sentencia* *siFalso*:sentencia*]] = valor[[condicion]] JZ else{n} / *n distinto* ejecuta[[siCierto$_i$]] JMP finElse{n} else{n}: ejecuta[[siFalso$_i$]] finElse{n}:

```
public class SelecciónDeInstrucciones extends DefaultVisitor {
    private PrintWriter writer;

    public SeleccionDeInstrucciones(Writer writer, String sourceFile) {
        this.writer = new PrintWriter(writer);
    }

    private void write(String text) {
        writer.println(text);
    }
```

```
// class Programa { List<DefVariable> definiciones;
//       List<Sentencia> sentencias; }
public Object visit(Programa node, Object param) {
    for (Sentencia child : node.getSentencias())
        child.accept(this, null);
    return null;
}

// class Print { Expresion expresion; }
public Object visit(Print node, Object param) {
    node.getExpresion().accept(this, null);
    write("out" + node.getExpr().getTipo().sufijo());
    return null;
}

// class If { Expresion condicion;  List<Sentencia> siCierto;
//       List<Sentencia> siFalso; }
public Object visit(If node, Object param) {
    node.getCondicion().accept(this, null);
    int n = contador++;
    write("jz else" + n);
    for (Sentencia child : node.getSiCierto())
        child.accept(this, null);
    write("jmp finElse" + n);

    write("else" + n + ':');
    for (Sentencia child : node.getSiFalso())
        child.accept(this, null);
    write("finElse" + n + ':');
    return null;
}

private int contador = 0;
```

Solución E5 (II)

Función de Código	Plantillas de Código
valor : **expresion** → Instruccion*	valor [[**literalInt** → *lexema*:string]] = PUSH {lexema} valor [[**variable** → *nombre*:string]] = PUSHA {variable.definición.dirección} LOAD$_{\text{<variable.tipo>}}$ valor [[**exprAritmetica** → *left*:expresion *operator*:string *right*:expresion]] = valor[[left]] valor[[right]] si operator == "+" ADD$_{\text{<exprAritmetica.tipo>}}$ si operator == "-" SUB$_{\text{<exprAritmetica.tipo>}}$ si operator == "*" MUL$_{\text{<exprAritmetica.tipo>}}$ si operator == "/" DIV$_{\text{<exprAritmetica.tipo>}}$

```
    // class LiteralInt { String lexema; }
    public Object visit(LiteralInt node, Object param) {
        write("push " + node.getLexema());
        return null;
    }

// class Variable { String nombre; }
    public Object visit(Variable node, Object param) {
        write("pusha " + node.getDefinicion().getDirección());
        write("load " + node.getTipo().sufijo());
        return null;
    }

// class ExprAritmetica { Expresion left;  String operator;
        Expresion right; }
    public Object visit(ExprAritmetica node, Object param) {
      node.getLeft().accept(this, null);
      node.getRight().accept(this, null);
      switch(node.getOperator()) {
          case "+": write("add" + node.getTipo().sufijo()); break;
          case "-": write("sub" + node.getTipo().sufijo()); break;
          case "*": write("mul" + node.getTipo().sufijo()); break;
          case "/": write("div" + node.getTipo().sufijo()); break;
      }
    }
}
```

Simplificable