# 1.6. Ejemplo de intérprete de código intermedio

## 1.6.1. Descripción del lenguaje

En esta sección se describirá la implementación de un intérprete de un lenguaje intermedio básico. El lenguaje consta únicamente de un tipo *int* y se basa en una pila de ejecución sobre la que se realizan las principales instrucciones aritméticas. Las instrucciones del lenguaje son:

| | |
|---|---|
| INT v | Declara *v* como una variable de tipo entero |
| PUSHA v | Mete en la pila la dirección de la variable *v* |
| PUSHC c | Mete en la pila la constante c |
| LOAD | Saca de la pila un elemento y carga en la pila el contenido en memoria de dicho elemento |
| STORE | Saca de la pila dos elementos y carga en la dirección del segundo elemento el contenido del primero |
| ADD | Saca dos elementos de la pila, los *suma* y mete el resultado en la pila |
| SUB | Saca dos elementos de la pila, los *resta* y mete el resultado en la pila |
| MUL | Saca dos elementos de la pila, los *multiplica* y mete el resultado en la pila |
| DIV | Saca dos elementos de la pila, los *divide* y mete el resultado en la pila |
| LABEL e | Establece una etiqueta *e* |
| JMPZ e | Saca un elemento de la pila y, si es igual a cero, continúa la ejecución en la etiqueta *e* |
| JMPGZ e | Saca un elemento de la pila y, si es mayor que cero, continúa la ejecución en la etiqueta *e* |
| GOTO e | Continúa la ejecución en la etiqueta *e* |
| JMPLZ e | Saca un elemento de la pila y, si es menor que cero, continúa la ejecución en la etiqueta *e* |
| OUTPUT v | Muestra por pantalla el contenido de la variable *v* |
| INPUT v | Lee un valor y lo inserta en la posición de memoria referenciada por *v* |
| ECHO cad | Muestra por pantalla la cadena *cad* |

Figura 14: Instrucciones del código intermedio

Desde el punto de vista sintáctico un programa está formado por una secuencia de instrucciones. Las variables y etiquetas son identificadores formados por una secuencia de letras y dígitos que comienzan por letra y se permiten comentarios con el formato de C++ (/*...*/) y (//....).

**Ejemplo:** El siguiente programa calcula el factorial de un número.

```
INT   var     // Read v
INPUT var
INT   i       // int i = 1;
PUSHA i
PUSHC 1
STORE
INT   Res     // int R = 1;
PUSHA Res
PUSHC 1
STORE
LABEL test    // while (i < v)
PUSHA i
LOAD
PUSHA v
LOAD
SUB
JMPLZ endLoop
```

```
PUSHA  Res     // R = R * i
PUSHA  Res
LOAD
PUSHA  i
LOAD
MUL
STORE
PUSHA  i       // i++;
PUSHA  i
LOAD
PUSHC  1
ADD
STORE
GOTO   test    // endWhile
LABEL  endLoop
OUTPUT Res     // write Res
```

## 1.6.2. Implementación del Intérprete

### 1.6.2.1. Representación del Código Intermedio

El código intermedio se representará mediante una clase abstracta *Code* con una subclase por cada tipo de instrucción.

```
class Code {
public:
 virtual void        exec(Context &) = 0;
 virtual string      id() const      = 0;
};
```

El método *exec* implementará la forma en que se ejecuta cada instrucción actuando sobre el contexto que se pasa como argumento. El método *id()* devuelve la cadena identificativa de cada código de instrucción.

Las diferentes instrucciones son subclases de la clase *Code*:

```
class PushA: public Code {
 public:
virtual void         exec(Context &c);
virtual string       id() const { return "PUSHA"; };
. . .
 private:
      string  argument;
};

class PushC: public Code {
public:
      virtual void          exec(Context &c);
      virtual string        id() const { return "PUSHC"; };
      . . .
private:
      int constant;
};

class Load: public Code {
public:
virtual void         exec(Context &c);
virtual string       id() const { return "LOAD"; };
      . . .
};

class Add: public Code {
public:
virtual void         exec(Context &c);
virtual string       id() const { return "ADD"; };
      . . .
};

. . .
```

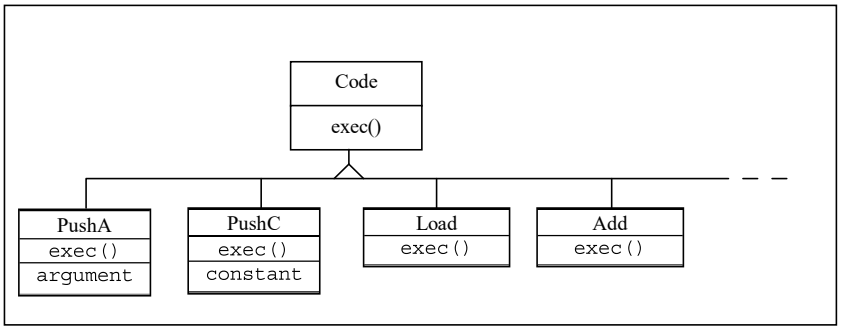


Figura 15: Representación de códigos de instrucción

Un programa está formado por una secuencia de códigos de instrucción:

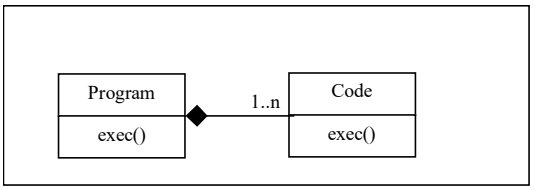


Figura 16: Representación de Programas

```
class Program: public ProgramNode {
public:
 void                addCode(Code *);
 int                 getCount();
 void                setCount(int );
 void                exec(Context &);
private:
 int                 _count; // instruction counter
 vector<Code*>       _code;
};
```

### 1.6.2.2. Contexto de Ejecución

El contexto de ejecución se representa mediante un objeto *Contexto* que agrupa las estructuras necesarias para simular el funcionamiento de la máquina abstracta en tiempo de ejecución.

En el lenguaje mostrado, los elementos del contexto son:

**Memoria (**`Memory`): Simula la memoria de la máquina y se forma mediante una lista de valores indexada por direcciones. En el ejemplo, tanto los valores como las direcciones son enteros.

**Pila de Ejecución** (`ExecStack`): Contiene la pila de ejecución donde se almacenan los valores. Sobre esta pila actúan directamente las instrucciones PUSHA, PUSHC, LOAD, STORE, etc.

**Tabla de Símbolos** (`SymTable`): Contiene información sobre los identificadores declarados y las etiquetas

**Registros:** El contexto de ejecución contiene una serie de registros o variables globales como el contador de instrucciones (`iCount`), la dirección de memoria libre (`StackP`) o el argumento de la instrucción actual (`arg`)

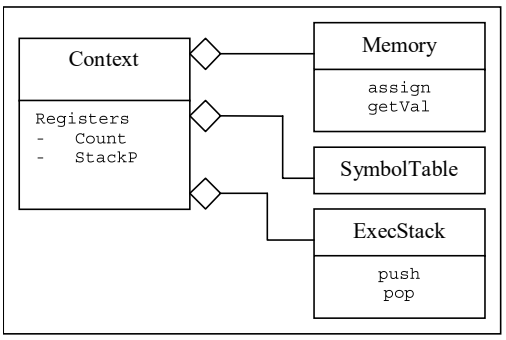


Figura 17: Contexto de ejecución

## Memoria

La memoria se simula mediante una clase *Memory* que contiene una lista de valores (enteros) indexada por direcciones (también enteros).

```
typedef int  TValue;
typedef int  TDir;

const TValue defaultValue = -1;

class Memory {
public:
Memory(TDir size = 100);
void         assign(TDir, TValue);
TValue               getVal(TDir);
TDir         getSize();
private:
TDir            _size;
vector<TValue>  _store;
void          checkRange(TDir n);
};
```

El constructor inicializa los contenidos de la memoria a los valores por defecto (se le pasa un argumento indicando el tamaño de memoria deseado). El método *assign* asocia una dirección de memoria con un valor. Internamente, la memoria se representa como un vector de valores.

La memoria almacena las variables declaradas por el programa. A medida que se declara una nueva variable, se reserva espacio para ella en la memoria. Aunque el lenguaje considerado no contiene variables locales (una vez declarada una variable, persiste hasta el final de la ejecución), una posible ampliación podría requerir este tipo de variables (por ejemplo, si se consideran procedimientos recursivos). En ese caso, conviene manejar la memoria como una pila LIFO, de forma que las variables locales se almacenan en la pila al entrar en un bloque y se sacan de la pila al salir del bloque. Este esquema se conoce como memoria *Stack*. Para su simulación, se utiliza la variable *StackP* que indica la siguiente posición libre de memoria. Cuando se ejecuta una instrucción INT id se asigna al identificador *id* de la tabla de símbolos la dirección *StackP* y se incrementa *StackP.*

***Nota**:* En el lenguaje intermedio propuesto tampoco es necesario manejar memoria dinámica o *Heap*. Si se contemplasen instrucciones de creación dinámica de elementos, sería necesario reservar una zona de la memoria para estos elementos y gestionar dicha zona mediante técnicas convencionales. Dependiendo del lenguaje a implementar, la gestión de memoria dinámica podría incluir recolección de basura.

## Pila de Ejecución

La pila de ejecución se implementa mediante una clase *execStack* con las operaciones clásicas sobre pilas, *push* y *pop*.

```
class ExecStack {
public:
void  push (const TValue& );
TValue       pop ();
      void   clearStack();
private:
      vector <TValue>       Stack;
}
```

Internamente, la pila de ejecución se representa también como un vector de valores.

## Tabla de Símbolos

La tabla de símbolos se utiliza en dos contextos diferentes:

1.- En la fase de **análisis léxico/Sintáctico**: La tabla contiene información de las palabras reservadas con el fin de detectar las diversas construcciones del lenguaje. A su vez, en estas fases, se almacenan en la tabla las etiquetas (con las direcciones en que aparecen) en la tabla.

2.- En la fase de **ejecución**: Se almacenan en la tabla las variables declaradas con la diracción de memoria a la que referencian. A su vez, en las instrucciones de salto, se consulta la dirección de la etiqueta en la tabla.

La tabla se implementa mediante una estructura *Hash* indexada por identificadores. Los nodos de la tabla son objetos de tres tipos:

**Palabras reservadas**: Códigos de las diferentes instrucciones (PUSHC, LOAD, ADD, etc.). Estos códigos contienen una referencia al objeto de la clase *Code* que indica el código de instrucción correspondiente.

**Etiquetas:** A medida que el analizador sintáctico encuentra etiquetas, inserta su posición en la tabla de símbolos. De esta forma, no es necesario realizar dos pasadas por el código fuente (una para buscar etiquetas y otra para ejecutar).

**Variables:** Se almacena información de las variables declaradas y la dirección de memoria en que fueron declaradas

Los tres tipos de nodos se representan mediante clases correspondientes con una superclase *STNode* común.

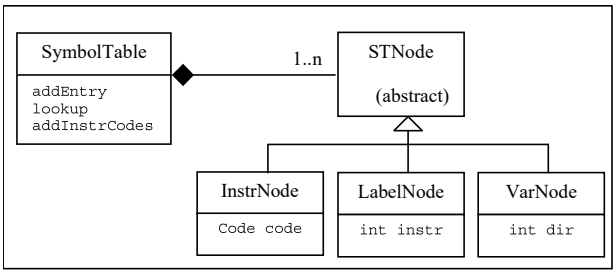


Figura 18: Tabla de Símbolos

```
typedef enum { SInstr, SLabel, SVar } SymbolType;

class STNode {
 public:
  virtual SymbolType type() = 0;
};

class InstrNode: public STNode {
 public:
  InstrNode(Code *);
  Code*              getCode() const;
  SymbolType         type();
private:
  Code*              _code;
};

class LabelNode: public STNode {
 public:
  LabelNode(int i);
  int                getI() const;
  SymbolType         type();
private:
```

```
  int                       _instr;
};

class VarNode: public STNode {
 public:
  VarNode(int);
  TDir              getDir() const;
  SymbolType        type();
 private:
  TDir              _dir;
};

class SymbolTable {
 public:
  STNode*           lookup         (string);
  void              addEntry       (string , STNode *);
  void              addInstrCodes  ();

 private:
   map<string, STNode*, less<string> > _table;
};
```

Los métodos *addEntry* y *lookup* insertan y buscan un valor en la tabla de símbolos. El método *addInstrCodes()* es llamado antes de realizar el análisis sintáctico para insertar los códigos de instrucción o palabras reservadas en la tabla de símbolos.

### 1.6.2.3. Ejecución de Instrucciones

El programa (formado por una lista de instrucciones) consta de un método *execute*. El método inicializa el contador de instrucciones a cero y se mete en un bucle ejecutando la instrucción indicada por el contador e incrementando el contador. El bucle se realiza mientras el contador de instrucciones sea menos que el número de instrucciones del programa.

```
void Program::exec(Context &context) {
 int i; // current instruction

 context.setCount(0);
 while ((i=context.getCount()) < _icount) {
      _code[i]->execute(context);
   context.incICount();
 }
}
```

Cada código de instrucción redefine el método virtual *execute* según su propia semántica:

```
void IntDecl::execute(Context &c) {
 c.newInt(argument);
}

void PushA::execute(Context &c) {
 c.push(c.lookup(argument));
}

void PushC::execute(Context &c) {
 c.push(constant);
}

void Load::execute(Context &c) {
 TValue d = c.pop();
 c.push(c.getVal(d));
}

void Store::execute(Context &c) {
 TValue v = c.pop();
 TValue d = c.pop();
 c.setVal(d,v);
}

void Input::execute(Context &c) {
 TValue v;
 cout << "Value of " << argument << " ?";
 cin >> v;
 TDir d = c.lookup(argument);
 c.setVal(d,v);
}
```

```
void Output::execute(Context &c) {
 TDir d       = c.lookup(argument);
 TValue  v    = c.getVal(d);
 cout << argument << " = " << v << endl;
}

void Add::execute(Context &c) {
 TValue v1 = c.pop();
 TValue v2 = c.pop();
 c.push(v1+v2);
}

void Sub::execute(Context &c) {
 TValue v1 = c.pop();
 TValue v2 = c.pop();
 c.push(v1-v2);
}

void Mul::execute(Context &c) {
 TValue v1 = c.pop();
 TValue v2 = c.pop();
 c.push(v1*v2);
}

void Div::execute(Context &c) {
 TValue v1 = c.pop();
 TValue v2 = c.pop();
 if (v2 == 0)
  RunningError("division by 0");
 c.push(v1/v2);
}

void Label::execute(Context &c) {
}

void JmpZ::execute(Context &c) {
 TValue v = c.pop();
 if (v == 0) {
  int newI = c.getLabel(argument);
  c.setICount(newI-1);
 }
}

void JmpLZ::execute(Context &c) {
 TValue v = c.pop();
 if (v < 0) {
  int newI = c.getLabel(argument);
  c.setICount(newI-1);
 }
}

void JmpGZ::execute(Context &c) {
 TValue v = c.pop();
 if (v > 0) {
  int newI = c.getLabel(argument);
  c.setICount(newI-1);
 }
}

void Goto::execute(Context &c) {
  int newI = c.getLabel(argument);
  c.setICount(newI-1);
}

void Echo::execute(Context &c) {
 cout << argument << endl;
}
```

El control de errores se realiza mediante el mecanismo de manejo de excepciones de C++. Cuando se produce un error de lanza una excepción con información del tipo de error. El programa principal se encarga de capturar los diferentes tipos de excepciones e informar al usuario de los mensajes correspondientes.

### 1.6.2.4. Análisis léxico y sintáctico

Como cualquier otro procesador, el intérprete requiere la implementación de un análisis léxico y sintáctico del código fuente para obtener el código intermedio que posteriormente será interpretado. Para ello, se debe especificar la gramática, que en este caso es muy sencilla:

```
<Program>  ::= { <Instr> }

<Instr>    ::= <código> <arg>

<código>   ::= INT | PUSHA | PUSHC | LOAD | STORE | INPUT | OUTPUT | ECHO
           |   ADD | SUB   | MUL   | DIV  | LABEL | GOTO  | JMPLZ  | JMPZ

<arg>      ::= <ident> |  <integer> | <vacio>
```

Figura 19: Gramática del Lenguaje

Dada la sencillez de la gramática, el analizador léxico y sintáctico se han desarrollado directamente sin herramientas tipo Lex o Yacc. El analizador léxico se implementó mediante una clase `Scanner`:

```
class Scanner
{
 public:
  Scanner(istream *);
  Token*     nextToken(void);
  void              tokenBack();
  . . .

 private:
  . . .
  Token*     _token;
  istream*   _input;

};
```

El constructor toma como argumento un flujo de entrada desde el cual se van a leer los caracteres. El método *nextToken* analiza dicho flujo de entrada y devuelve el siguiente *token*. El método *tokenBack* se utiliza para que la siguiente llamada a *nextoToken* devuelva el último *token* analizado.

El analizador sintáctico es recursivo descendente y se ha representado mediante una clase abstracta *Parser* de la cual derivan las clases que analizan las diversas construcciones del lenguaje. Con el fin de separar el análizador de la construcción de nodos, se utiliza una clase *Builder*[2] que se encarga de construir el árbol sintáctico a partir de las indicaciones del analizador sintáctico.

```
class Parser {
 public:
  virtual bool parse(Scanner&, Builder&, SymbolTable &) = 0;
};

class ParserProg:   public Parser {
 public:
  virtual bool parse(Scanner&, Builder&, SymbolTable &);
};

class ParserCode:   public Parser {
 public:
  virtual bool parse(Scanner&, Builder&, SymbolTable &);
};

class ParserArg:   public Parser {
 public:
  virtual bool parse(Scanner&, Builder&, SymbolTable &);
};

class Builder {
 public:
       virtual void         addCode(Code*);
       virtual ProgramNode* getProgram();

 protected:
       Builder();
};
```

[2] Se ha utilizado el patrón de diseño *Builder* [Gamma 95]

## *1.7. Ejemplo de intérprete de lenguaje recursivo*

En este apartado se describe la implementación de un intérprete de un sencillo lenguaje recursivo. Se describen tres posibles diseños indicando las ventajas e inconvenientes de cada uno.

### 1.7.1. Definición del lenguaje

El lenguaje consta de dos categorías sintácticas fundamentales: expresiones y órdenes, que tienen la siguiente estructura:

```
<comm> ::= while <expr> do <comm>
         | if <expr> then <comm> else <comm>
         | <var> := <expr>
         | <comm> ; <comm>

<expr> ::= <expr> <binOp> <expr>
         | <var>
         | <integer>

<binOp> ::= + | - | * | / | = | <
```

### 1.7.2. Diseño imperativo simple

La implementación imperativa simple consistiría en utilizar una instrucción de selección que, dependiendo del tipo de instrucción, ejecute una porción de código determinada. El código tendría la siguiente apariencia:

```
void exec(Context &c) {
 switch (c.code) {
   case WHILE: . . .
   case IF: . . .
   . . .
 }
}
```

Como en el lenguaje intermedio, la representación interna de las instrucciones se puede realizar mediante una unión.

El diseño imperativo tiene como principal ventaja la eficiencia y sencillez de aplicación. Sin embargo, su utilización puede perjudicar la eficiencia de la representación interna de las instrucciones, desperdiciando memoria para instrucciones pequeñas. Otras desventajas son la dificultad para añadir instrucciones sin modificar el código anterior y la falta de seguridad (es posible acceder a campos de la unión equivocados sin que el compilador lo detecte.

### 1.7.3. Diseño Orientado a Objetos simple

La representación de las instrucciones puede realizarse mediante una clase abstracta `Comm` que contiene un método `exec` que indica cómo ejecutar dicha instrucción en un contexto determinado.

```
abstract class Comm {
 void exec(Context ctx);
}
```

Cada una de las instrucciones será una subclase de la clase abstracta que definirá el método `exec` e incluirá los elementos necesarios de dicha instrucción.

```
class While extends Comm {
 Expr e;
 Comm c;
 void exec(Context ctx) {
  for (;;) {
    Bvalue v = (Bvalue) e.eval(ctx);
    if (!v.b) break;
    c.exec(ctx);
  }
 }
}

class If extends Comm {
 Expr e;
 Comm c1, c2;
 void exec(Context ctx) {
    Bvalue v = (Bvalue) e.eval(ctx);
```

```
    if (v.b) c1.exec(ctx);
    c2.exec(ctx);
 }
}

class Seq extends Comm {
 Comm c1, c2;
 void exec(Context ctx) {
    c1.exec(ctx);
    c2.exec(ctx);
 }
}

class Assign extends Comm {
 String name; Expr e;
 void exec(Context ctx) {
    Value v = e.eval(ctx);
    ctx.update(name,v);
 }
}

class Skip extends Comm {
 void exec(Context ctx) {}
}
```

En el caso de las expresiones, se define una clase abstracta con un método de evaluación de expresiones en un contexto. Obsérvese que el método de evaluación devuelve un valor.

```
abstract class Expr {
 Value eval(Context ctx);
}

class BinOp extends Expr {
 Operator op;
 Expr e1,e2;
 Value eval(Context ctx) {
    Value v1 = e1.eval(ctx);
    Value v2 = e2.eval(ctx);
    return op.apply(v1,v2);
 }
}

class Var extends Expr {
 String name;
 Value eval(Context ctx) {
    return ctx(lookup(name));
 }
}

class Const extends Expr {
 int n;
 Value eval(Context ctx) {
    return n;
 }
}
```

En el código anterior, se utilizan varias funciones auxiliares del contexto:

- `Value lookup(String)` busca el valor de un nombre
- `void update(String, Value)` actualiza el valor de un nombre

## 1.7.4. Utilización del patrón *visitor*

El problema del esquema anterior es que el código correspondiente a la interpretación está disperso en cada una de las clases del árbol sintáctico. En la práctica, la construcción de un procesador de un lenguaje puede requerir la realización de varias fases: impresión, generación de código, interpretación, chequeo de tipos, etc.

Mediante el patrón *visitor* es posible concentrar el código de cada fase en una sola clase. Para ello, se define un método `visita` en cada uno de los tipos de nodos del árbol (instrucciones o expresiones). Lo único que realiza dicho método es identificarse a sí mismo invocando el método correspondiente de la clase visitante.

```
abstract class Comm {
 public Object visita(Visitor v);
}
```

```
class While extends Comm {
 public Expr e;
 public Comm c;

 public Object visita(Visitor v) {
  return v.visitaWhile(this);
 }
}

class If extends Comm {
 public Expr e;
 public Comm c1, c2;

 public Object visita(Visitor v) {
  return v.visitaIf (this);
 }
}

class Seq extends Comm {
 public Comm c1, c2;

 public Object visita(Visitor v) {
  return v.visitaSeq(this);
 }
}

class Assign extends Comm {
 public String name;
 public Expr e;

 public Object visita(Visitor v) {
  return v.visitaAssign(this);
 }
}

class Skip extends Comm {

 Object visita(Visitor v) {
  return v.visitaSkip(this);
 }
}
```

En el caso de las expresiones, se realiza el mismo esquema.

```
abstract class Expr {
 public abstract Object visita(Visitor v);
}

class BinOp extends Expr {
 public Operator op;
 public Expr e1,e2;

 public Object visita(Visitor v) {
  return v.visitaBinOp(this);
 }
}

class Var extends Expr {
 public String name;

 public Object visita(Visitor v) {
  return v.visitaVar(this);
 }
}

class Const extends Expr {
 public int n;

 public Object visita(Visitor v) {
  return v.visitaConst(this);
 }
}
```

Se define una clase abstracta `Visitor` cuyas subclases representarán posibles recorridos. La clase incluye métodos del tipo `Object visitaX(X n)` para cada tipo de nodo X del árbol sintáctico.

```
abstract class Visitor {
 Object visitaWhile(While w);
 Object visitaIf(If i);
 Object visitaSeq(Seq s);
 Object visitaAssign(Assign a);
 Object visitaSkip(Skip s);
 Object visitaBinOp(BinOp b);
 Object visitaVar(Var v);
 Object visitaConst(Const c);
}
```

A continuación se define el intérprete como un posible recorrido del árbol sintáctico y por tanto, una subclase de `Visitor`.

```
class Interp extends Visitor {

 // Aquí se pueden definir los elementos del contexto (Tabla, Memoria, etc.)

 Object visitaWhile(While w) {
  for (;;) {
   BValue v = (BValue) w.e.visita(this);
   if (!v.b) break;
   w.c.visita(this);
  }
 }

 Object visitaIf(If i){
  BValue v = (BValue) i.e.visita(this);
  if (!v.b) i.c1.visita(this);
  else i.c2.visita(this);
 }

 Object visitaSeq(Seq s){
  s.c1.visita(this);
  s.c2.visita(this);
 }

 Object visitaAssign(Assign a) {
  Value v = (Value) a.e.visita(this);
  update(a.name,v);
 }

 Object visitaSkip(Skip s) {
 }

 Object visitaBinOp(BinOp b) {
  Value v1 = (Value) b.e1.visita(this);
  Value v2 = (Value) b.e2.visita(this);
  return (b.op.apply(v1,v2));
 }

 Object visitaVar(Var v){
  return lookup(v.name);
 }

 Object visitaConst(Const c) {
  return c.n;
 }

 // Función de ejecución de órdenes
 Object exec(Comm c) {
  c.visita(this);
 }
}
```

Las principales ventajas de este diseño son:

- Todo el código del intérprete está localizado en una clase, facilitando las modificaciones.
- Pueden añadirse nuevos tipos de recorridos como generación de código, impresión, chequeo de tipos, etc. de forma sencilla. Cada tipo de recorrido será una subclase de *Visitor*. Al añadir un tipo de recorrido, no es necesario modificar el código del árbol sintáctico.

Sin embargo, este diseño también tiene varias desventajas

- Es más difícil añadir nuevos tipos de nodos al árbol sintáctico. Al hacerlo, habría que modificar todas las subclases de *Visitor.*
- Se debe exponer el interior de los nodos del árbol sintáctico, perjudicando la encapsulación.
- Todos los métodos del tipo "`Object visitaX( X ...)`" devuelven un valor `Object` obligando a realizar ahormados que pueden acarrear problemas de seguridad.