Intérpretes y Diseño de Lenguajes de Programación

Book · November 2003

CITATIONS
0

READS
6,805

6 authors, including:

Jose Emilio Labra Gayo
University of Oviedo
201 PUBLICATIONS **1,286** CITATIONS
SEE PROFILE

Juan Manuel Cueva Lovelle
University of Oviedo
207 PUBLICATIONS **1,707** CITATIONS
SEE PROFILE

Raúl Izquierdo Castanedo
University of Oviedo
22 PUBLICATIONS **70** CITATIONS
SEE PROFILE

Aquilino Adolfo Juan
University of Oviedo
41 PUBLICATIONS **103** CITATIONS
SEE PROFILE

Intérpretes y Diseño de Lenguajes de Programación

Jose Emilio Labra Gayo
Juan Manuel Cueva Lovelle
Raúl Izquierdo Castanedo
Aquilino Adolfo Juan Fuente
Mª Cándida Luengo Díez
Francisco Ortín Soler

Tabla de Contenidos

1. Intérpretes

1.1. Definición

Un **intérprete** es un programa que analiza y ejecuta simultáneamente un programa escrito en un lenguaje fuente.

En la Figura 1 se presenta el esquema general de un intérprete visto como una *caja negra.* Cualquier intérprete tiene dos entradas: un programa P escrito en un lenguaje fuente LF (en lo sucesivo, se denotará P/LF) junto con los **datos** de entrada; a partir de dichas entradas, mediante un proceso de interpretación va produciendo unos **resultados**.

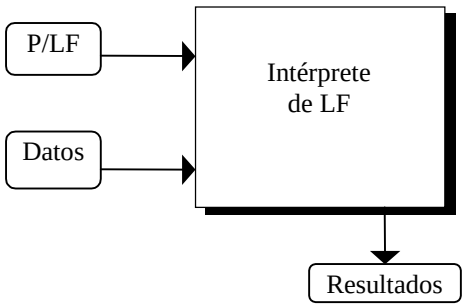

Figura 1: Esquema general de un intérprete

Los compiladores, a diferencia de los intérpretes, transforman el programa a un programa equivalente en un código objeto (fase de compilación), y en un segundo paso generan los resultados a partir de los datos de entrada (fase de ejecución).

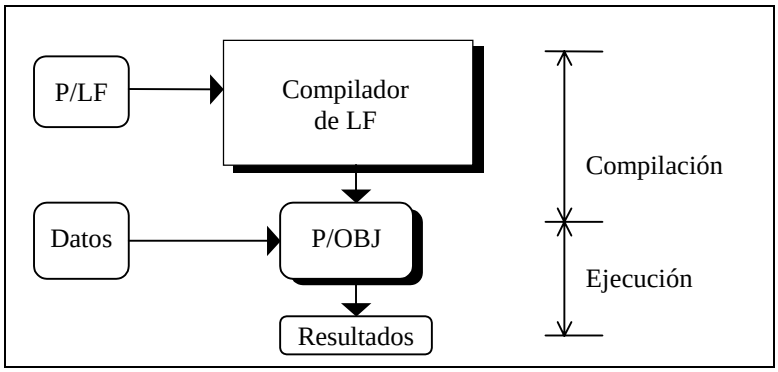

Figura 2: Esquema general de un compilador

1.2. Estructura de un intérprete

A la hora de construir un intérprete es conveniente utilizar una Representación Interna (RI) del lenguaje fuente a analizar. De esta forma, la organización interna de la mayoría de los intérpretes se descompone en los módulos:

Traductor a Representación Interna: Toma como entrada el código del programa *P* en Lenguaje Fuente, lo analiza y lo transforma a la representación interna correspondiente a dicho programa *P*.

Representación Interna (P/RI): La representación interna debe ser consistente con el programa original. Entre los tipos de representación interna, los **árboles sintácticos** son los más utilizados y, si las características del lenguaje lo permiten, pueden utilizarse estructuras de pila para una mayor eficiencia.

Tabla de símbolos: Durante el proceso de traducción, es conveniente ir creando una tabla con información relativa a los símbolos que aparecen. La información a almacenar en dicha tabla de símbolos depende de la complejidad del lenguaje fuente. Se pueden almacenar etiquetas para instrucciones de salto, información sobre identificadores (nombre, tipo, línea en la que aparecen, etc.) o cualquier otro tipo de información que se necesite en la etapa de evaluación.

Evaluador de Representación Interna: A partir de la Representación Interna anterior y de los datos de entrada, se llevan a cabo las acciones indicadas para obtener los resultados. Durante el proceso de evaluación es necesario contemplar la aparición de errores.

Tratamiento de errores: Durante el proceso de evaluación pueden aparecer diversos errores como desbordamiento de la pila, divisiones por cero, etc. que el intérprete debe contemplar.

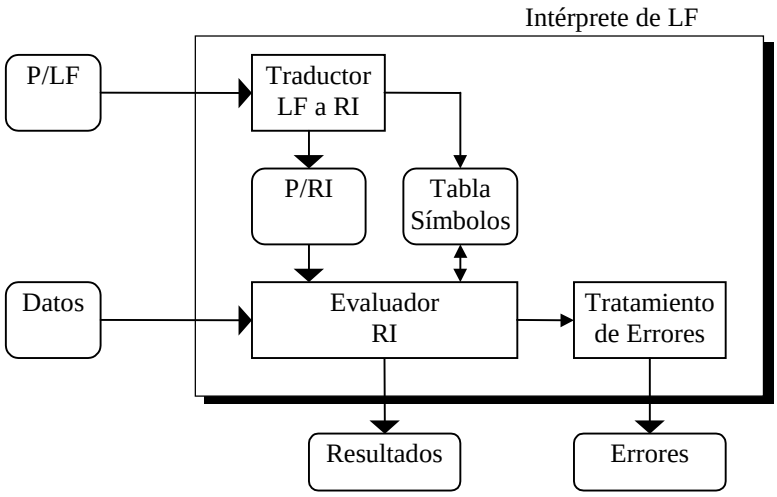

Figura 3: Organización interna de un intérprete

Dependiendo de la complejidad del código a analizar, el intérprete puede contener módulos similares a los de un compilador tradicional: Análisis léxico, Sintáctico y Semántico. Durante la evaluación, el intérprete interactúa con los recursos del sistema como la memoria, discos, etc. Muchos sistemas interpretados liberan al programador del manejo explícito de memoria mediante técnicas de **recolección de basura.**

A la hora de evaluar la representación interna, existen dos métodos fundamentales: la interpretación iterativa y la interpretación recursiva.

1.2.1. Interpretación Iterativa

La interpretación iterativa es apropiada para lenguajes sencillos, donde se analiza y ejecuta cada expresión de forma directa, como podrían ser los códigos de máquinas abstractas o lenguajes de sentencias simples. La interpretación consiste en un ciclo básico de búsqueda, análisis y ejecución de instrucciones.

El esquema sería:

```
Inicializar
REPETIR
  Buscar siguiente Instrucción i
  SI encontrada ENTONCES
                  Analizar i
                  Ejecutar i
HASTA (que no haya más instrucciones)
```

Figura 4: Interpretación iterativa

Cada instrucción se busca en el almacenamiento (memoria o disco) o, en algunos casos, es introducida directamente por el usuario. Luego la instrucción es analizada en sus componentes y ejecutada. Normalmente, el lenguaje fuente contiene varios tipos de instrucciones, de forma que la ejecución se descompone en varios casos, uno por cada tipo de instrucción. En la página 42, se construye un intérprete iterativo de un sencillo lenguaje intermedio.

1.2.2. Interpretación Recursiva

Comúnmente, el diseño de nuevos lenguajes de programación se realiza en dos fases:

Una **primera fase** de especificación semántica mediante la construcción de un intérprete prototipo que actúa como una especificación ejecutable y una **segunda fase** de implementación del compilador de dicho lenguaje.

Para la construcción de prototipos suele utilizarse un modelo de interpretación recursiva donde las sentencias pueden estar compuestas de otras sentencias y la ejecución de una sentencia puede lanzar la ejecución de otras sentencias de forma recursiva.

Los intérpretes recursivos no son apropiados para aplicaciones prácticas debido a su ineficiencia y se utilizan únicamente como prototipo ejecutable del lenguaje.

El problema de especificar un lenguaje mediante un intérprete prototipo es decidir en qué lenguaje se implementa dicho intérprete. Dicho lenguaje debe ser suficientemente expresivo y no ambigüo para definir claramente cómo funcionan las diferentes construcciones. En muchos casos se opta por utilizar lenguajes ya implementados pero que carecen de una especificación semántica clara. La tendencia actual es investigar técnicas de especificación semántica formal que permitan generar automáticamente este tipo de intérpretes [Espinosa94], [Liang96], [Steele94].

1.3. Ventajas de la utilización de intérpretes

En general, la utilización de compiladores permite construir programas más eficientes que los correspondientes interpretados. Esto es debido a que durante la ejecución de código compilado no es necesario realizar complejos análisis (ya se hicieron en tiempo de compilación), además, un buen compilador es capaz de detectar errores y optimizar el código generado.

Los intérpretes, por definición, realizan la fase de análisis y ejecución a la vez, lo cual imposibilita tales optimizaciones. Por esta razón, los sistemas interpretados suelen ser menos eficientes que los compilados. No obstante, los nuevos avances informáticos aumentan la velocidad de procesamiento y capacidad de memoria de los ordenadores. Actualmente, la eficiencia es un problema menos grave y muchas veces se prefieren sistemas que permitan un desarrollo rápido de aplicaciones que cumplan fielmente la tarea encomendada.

A continuación se enumeran una serie de ventajas de los sistemas interpretados:

Los intérpretes, en general, son más **sencillos de implementar**. Lo cual facilita el estudio de la corrección del intérprete y proporciona nuevas líneas de investigación como la generación automática de intérpretes a partir de las especificaciones semánticas del lenguaje.

Proporcionan una **mayor flexibilidad** que permite modificar y ampliar características del lenguaje fuente. Muchos lenguajes como Lisp, APL, Prolog, etc. surgieron en primer lugar como sistemas interpretados y posteriormente surgieron compiladores.

No es necesario contener en memoria todo el código fuente. Esto permite su utilización en sistemas de poca memoria o en **entornos de red**, en los que se puede obtener el código fuente a medida que se necesita [Plezbert 97].

Facilitan la **meta-programación**. Un programa puede manipular su propio código fuente a medida que se ejecuta. Esto facilita la implementación de sistemas de aprendizaje automatizado y reflectividad [Aït Kaci 91] H. Aït Kaci, *Warren's Abstract Machine, a tutorial reconstruction*. MIT Press, 1991

.

Aumentan la **portabilidad** del lenguaje: Para que el lenguaje interpretado funcione en otra máquina sólo es necesario que su intérprete funcione en dicha máquina.

Puesto que no existen etapas intermedias de compilación, los sistemas interpretados facilitan el desarrollo rápido de **prototipos**, potencian la utilización de sistemas **interactivos** y facilitan las tareas de **depuración**.

1.4. Aplicaciones de los sistemas basados en intérpretes

Los sistemas interpretados han tenido una gran importancia desde la aparición de los primeros ordenadores. En la actualidad, la evolución del *hardware* abre nuevas posibilidades a los sistemas interpretados. La preocupación ya no es tanto la eficiencia como la capacidad de desarrollo rápido de nuevas aplicaciones. Las principales aplicaciones podrían resumirse en:

Intérpretes de Comandos: Los sistemas operativos cuentan con intérpretes de comandos como el *Korn-Shell, C-Shell, JCL,* etc. Estos intérpretes toman un lenguaje fuente que puede incluir sentencias de control (bucles, condiciones, asignaciones, etc.) y ejecutan los diferentes comandos a medida que aparecen en el lenguaje.

Lenguajes basados en Escritos (Scripting Languages), diseñados como herramientas que sirvan de enlace entre diferentes sistemas o aplicaciones. Suelen ser interpretados con el fin de admitir una mayor flexibilidad a la hora de afrontar las peculiaridades de cada sistema. Podrían destacarse *Perl, Tcl/Tk*, *JavaScript, WordBasic* [Ousterhout 97]

Entornos de Programación: Existen ciertos lenguajes que contienen características que impiden su compilación o cuya compilación no es efectiva. Estos lenguajes suelen disponer de un complejo entorno de desarrollo interactivo con facilidades para la depuración de programas. Entre estos sistemas pueden destacarse los entornos de desarrollo para Lisp, Visual Basic, Smalltalk, etc.

Lenguajes de Propósito Específico: Ciertos lenguajes incluyen sentencias que realizan tareas complejas en contextos específicos. Existe una gran variedad de aplicaciones en las que se utilizan este tipo de lenguajes como consultas de Bases de Datos, simulación, descripción de hardware, robótica, CAD/CAM, música, etc.

Sistemas en Tiempo Real: Entornos que permiten modificar el código de una aplicación en tiempo de ejecución de forma interactiva.

Intérprete de Código Intermedio: Una tendencia tradicional en el diseño de compiladores es la generación de un código intermedio para una máquina abstracta, por ejemplo, el P-Code de Pascal o los *bytecodes* de Java. El siguiente paso puede ser: generación del código objeto a partir del código intermedio para una máquina concreta, finalizando el proceso de compilación o **interpretar** dicho código intermedio en una máquina concreta. La tendencia habitual es definir un lenguaje intermedio independiente de una máquina concreta. Para ello, suele definirse una máquina virtual que contenga las instrucciones definidas por el lenguaje intermedio, permitiendo una mayor portabilidad. Un ejemplo sería la Máquina Virtual de Java, que es simulada en la mayoría de los visualizadores Web.

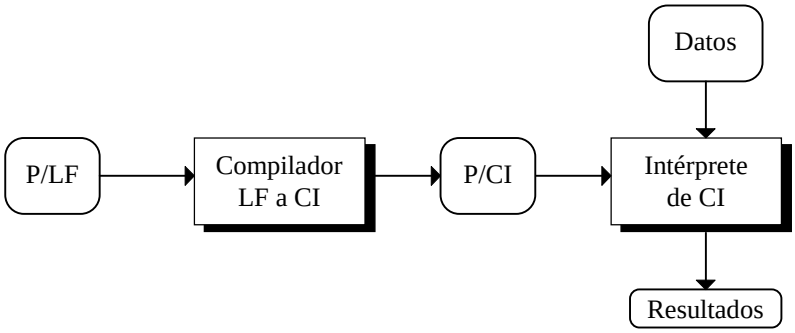

Figura 5: Esquema de Compilador con Intérprete de código intermedio

En la siguiente tabla, tomada de [Hudak, 98] se resumen algunos de los principales lenguajes de propósito específico con sus respectivas aplicaciones.

Lenguaje	**Aplicación**
Perl	Manipulación de textos y ficheros. *scripting*
VHDL	Descripción de Hardware
Tex, Latex, troff	Formateo de documentos
HTML, SGML, XML	Estructuración de documentos
Lex, Yacc	Análisis léxico y sintáctico
SQL, LDL, QUEL	Bases de datos
pic, PostScript	Gráficos en 2 dimensiones
Open GL	Gráficos en 3 dimensiones de alto nivel
Tcl, Tk	Interfaces gráficos de usuario
Mathematica, Maple	Computación simbólica matemática
Autolisp/AutoCAD	Diseño asistido por ordenador
Csh, Ksh	Intérpretes de Comandos
IDL	Tecnología de componentes
Emacs Lisp	Edición de texto
Visual Basic	*scripting*

1.5. Tipos de intérpretes

A continuación se va a realizar una clasificación de los diferentes métodos de interpretación según la estructura interna del intérprete. Es conveniente observar que algunos métodos podrían considerarse híbridos, ya que mezclan los procesos de compilación e interpretación.

1.5.1. Intérpretes puros

Los **intérpretes puros** son los que analizan y ejecutan sentencia a sentencia todo el programa fuente. Siguen el modelo de interpretación iterativa y, por tanto, se utilizan principalmente para lenguajes sencillos.

Los intérpretes puros se han venido utilizando desde la primera generación de ordenadores al permitir la ejecución de largos programas en ordenadores de memoria reducida, ya que sólo debían contener en memoria el intérprete y la sentencia a analizar y ejecutar en cada momento. El principal problema de este tipo de intérpretes es que si a mitad del programa fuente se producen errores, se debe de volver a comenzar el proceso.

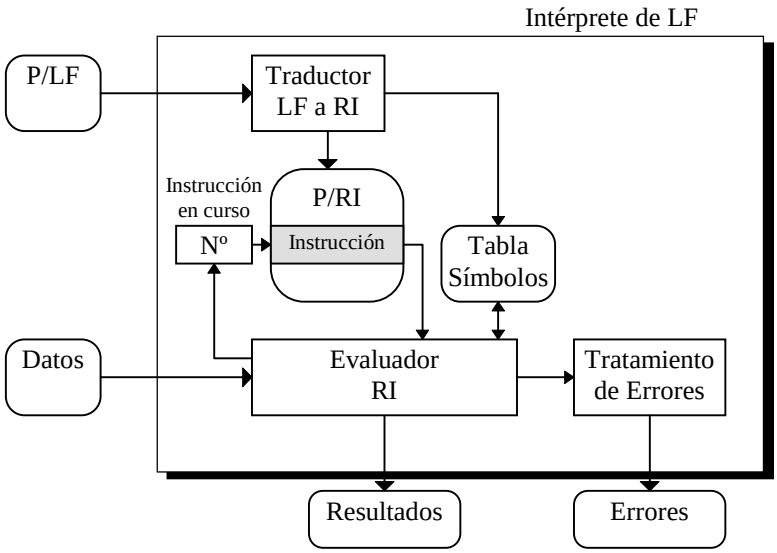

Figura 6: Esquema de un intérprete puro

En la figura se representa el esquema general de un intérprete puro. Se puede observar que el lenguaje fuente se traduce a una **representación interna** (texto o binaria) que puede ser almacenada en memoria o en disco. Esta representación interna tiene todas las instrucciones numeradas o colocadas consecutivamente en estructuras de tamaño fijo (por ejemplo un array o posiciones consecutivas de memoria, o un fichero binario de estructuras de tamaño fijo). Mientras se realiza este paso se puede construir la tabla de símbolos o etiquetas, que es una tabla que contiene una estructura donde están todas las etiquetas y su posición en el programa fuente (las etiquetas se utilizan tanto en las instrucciones de salto como en las llamadas a procedimientos y funciones). Una vez que este proceso ha finalizado, comienza la ejecución por la primera instrucción del código, que se envía al **evaluador de instrucciones**, éste la ejecuta (recibiendo datos si es necesario o enviando un mensaje de error). El evaluador de instrucciones también determina la instrucción siguiente a ejecutar, en algunos casos previa consulta a la tabla de etiquetas. En caso de que no haya saltos (GOTO) se ejecuta la siguiente instrucción a la instrucción en curso.

1.5.2.Intérpretes avanzados

Los **intérpretes avanzados** o normales incorporan un paso previo de análisis de todo el programa fuente. Generando posteriormente un lenguaje intermedio que es ejecutado por ellos mismos. De esta forma en caso de errores sintácticos no pasan de la fase de análisis. Se utilizan para lenguajes más avanzados que los intérpretes puros, ya que permiten realizar un análisis más detallado del programa fuente (comprobación de tipos, optimización de instrucciones, etc.)

1.5.3. Intérpretes incrementales

Existen ciertos lenguajes que, por sus características, no se pueden compilar directamente. La razón es que pueden manejar objetos o funciones que no son conocidos en tiempo de compilación, ya que se crean dinámicamente en tiempo en ejecución. Entre estos lenguajes, pueden considerarse Smalltalk, Lisp o Prolog. Con el propósito de obtener una mayor eficiencia que en la interpretación simple, se diseñan **compiladores incrementales**. La idea es compilar aquellas partes estáticas del programa en lenguaje fuente, marcando como dinámicas las que no puedan compilarse. Posteriormente, en tiempo de ejecución, el sistema podrá compilar algunas partes dinámicas o recompilar partes dinámicas que hayan sido modificadas. Estos sistemas no producen un código objeto independiente, sino que acompañan el sistema que permite compilar módulos en tiempo de ejecución (*run time system*) al código objeto generado.

Normalmente, los compiladores incrementales se utilizan en sistemas interactivos donde conviven módulos compilados con módulos modificables [Rober94].

1.5.4. Evaluadores Parciales

La utilización de evaluadores parciales o especializadores surge al considerar que muchos programas contienen dos tipos de datos de entrada. Existen una serie de datos de entrada que son diferentes en cada ejecución mientras que otros datos no varían de una ejecución a otra. El primer conjunto, se conoce como datos de entrada dinámicos (se denotará como *Din*), mientras que el segundo conjunto, serían los datos de entrada estáticos (*Est*). Dado un programa *P*, el proceso de evaluación parcial consiste en construir otro programa especializado P_{Est} para los datos estáticos de *P.* El programa P_{Est} suele estar escrito en el mismo lenguaje fuente que *P* y se debe garantizar que cuando se le

presenten los datos dinámicos produzca los mismos resultados que si se hubiesen presentado todos los datos al programa *P* original.

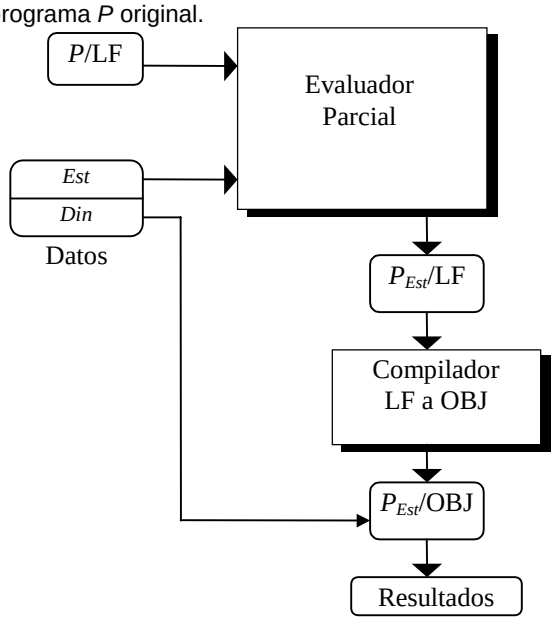

Figura 7: Evaluación Parcial

Ejemplo: Considérese el siguiente fragmento de programa *P1* que toma la entrada de dos ficheros diferentes, `fichEst` y `fichDin` y escribe el resultado en la salida estándar.

```
read(fichEst,a);
while (a > 0) do
begin
 read(fichDin,b);
 if (a > 10) then write (b - 2 * a)
             else write (a * a + b);
 read(fichEst,a);
end
```

Figura 8: Programa *P1*

Si el contenido de `fichEst` fuese siempre `5 12 7 -1` el evaluador parcial podría generar un programa especializado para dicho conjunto de datos, obteniendo $P1_{Est}$:

```
read(fichDin,b); write(25 + b);
read(fichDin,b); write(b - 24);
read(fichDin,b); write(49 + b);
```

Figura 9: Programa $P1_{Est}$

La principal ventaja de la evaluación parcial es la eficiencia. Si se conoce de antemano que un programa *P* va a ejecutarse muchas veces con un mismo conjunto de datos *Est* pero diferentes datos *Din*, será más eficiente evaluar parcialmente *P* para obtener P_{Est} y ejecutar luego P_{est}.

En el ejemplo, puede observarse que es posible eliminar el bucle “while” y la sentencia “if” debido a que las condiciones dependen de datos estáticos. Sin embargo, las cosas no son siempre tan fáciles, considérese que en el programa *P1* se elimina la última sentencia `“read(fichEst,a)”` del bucle. Entonces el evaluador parcial, podría entrar en un bucle infinito intentando generar el programa especializado.

Por este motivo, los evaluadores parciales deben realizar un complejo análisis del programa fuente para detectar que el proceso no genere un bucle infinito. El *análisis de tiempo de enlace* (*binding-time analysis)* es una técnica que se encarga de detectar qué valores son estáticos y pueden evaluarse y cuáles no.

Una aplicación interesante de la evaluación parcial es la posibilidad de generar compiladores a partir de intérpretes. Para ello, supóngase que a la entrada del evaluador parcial se presenta el intérprete de un lenguaje de programación *LF* escrito en un lenguaje de transición *LT*, junto con un programa *P* escrito en en *LF*. El evaluador parcial generará un intérprete especializado para el programa *P* en el lenguaje *LT.* Suponiendo la existencia de un compilador para el lenguaje *LT*, se puede obtener el intérprete especializado en código objeto que, una vez presentados los datos de entrada *D* genere los resultados.

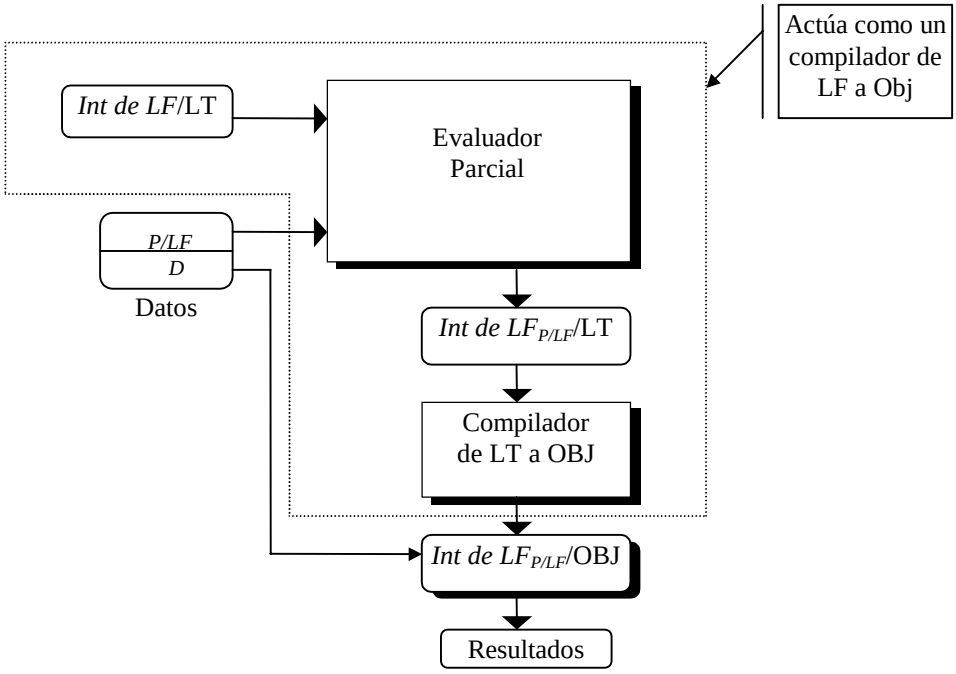

Figura 10: Obtención de un compilador a partir de un intérprete mediante evaluación parcial

Dado que los intérpretes se utilizan como especificaciones semánticas de un lenguaje. La obtención automatizada de un compilador a partir de la definición del intérprete permite alcanzar la eficiencia de un compilador sin perder la corrección semántica de un intérprete.

La evaluación parcial tiene otras aplicaciones interesantes en campos como el *ray-tracing* [Ander94], modelización de mundos virtuales [Besh97], reconocimiento de patrones, consultas a bases de datos, redes neuronales, etc.

Para una revisión general de la evaluación parcial puede consultarse [Jones 93] [Jones 96] y [Pagan 91].

1.5.5. Compiladores "Just in Time"

Con la aparición de *Internet* surge la necesidad de distribuir programas de una forma independiente de la máquina permitiendo su ejecución en una amplia variedad de plataformas. Los códigos de bytes de la máquina Virtual de Java permiten la ejecución de programas distribuidos, ya que la mayoría de los visualizadores tienen un mecanismo capaz de interpretarlos. La interpretación de códigos de bytes supone una demora en los tiempos de ejecución.

Para evitar la interpretación, muchos sistemas transforman los códigos de bytes en código nativo siguiendo el modelo *"**just in time***". En este modelo, una unidad de compilación o clase se transmite en el formato de códigos de bytes, pero no se realiza la interpretación. En lugar de ello, el código es compilado a código nativo justo en el momento en que lo necesita el programa que se está ejecutando.

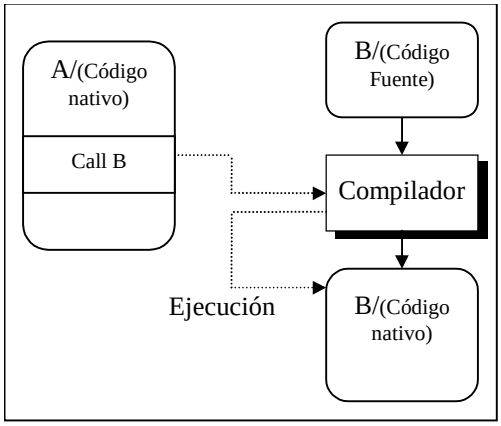

Figura 11: Compilación de una clase *"Just in Time"*

En la figura se muestra el ejemplo de una unidad de compilación A compilada (en código nativo) que encuentra una instrucción de llamada a otra unidad B en código fuente (códigos de byte[1]). El sistema realiza dos acciones:

- `Compila la unidad B a código nativo.`
- `Continúa la ejecución con el código nativo compilado de la unidad B`

Las principales ventajas de la compilación *Just in Time* son:

1.- Los programas *grandes* contienen porciones de código que no son ejecutadas en una ejecución típica del programa. Por ejemplo, un visualizador de páginas Web, podría contener rutinas para manejar diversos formatos de los datos transmitidos, pero algunos de dichos formatos podrían no ser utilizados en una determinada ejecución del visualizador. Puesto que la compilación *Just in Time* sólo traduce aquéllas porciones de código que se necesitan, se evita el tener que compilar código que no se va a utilizar.

2.- Los sistemas tradicionales realizan la compilación de todo el código antes de la ejecución, lo que para el usuario puede presentar un lapso de tiempo substancial entre el momento en que todas las unidades de compilación han sido transmitidas y el momento en que la ejecución puede comenzar. Esta técnica tiende a repartir el tiempo de compilación a lo largo de la ejecución del programa. El efecto producido al interrumpir la ejecución para compilar una unidad es similar al producido por la recolección de basura.

1.5.6. Compilación Continua

La compilación continua surge como un intento de mejorar la compilación *"Just in Time"*. El sistema mezcla el proceso de compilación a código nativo con el proceso de interpretación. Para conseguirlo, el sistema dispone de dos módulos: un módulo de intérpretación de los códigos de bytes y otro módulo de compilación de códigos de bytes a código nativo. La idea consiste en que ambos módulos actúen a la vez (lo ideal sería disponer de dos procesadores), de forma que el sistema no se detenga a compilar un módulo, sino que vaya interpretándolo hasta que el compilador haya generado el código nativo.

En la figura se distinguen los módulos que intervienen:

Código: El código contiene una mezcla de código fuente y código nativo del programa. Inicialmente todo el código está sin compilar, pero a medida que el programa es ejecutado, el compilador genera traducciones a código nativo de las unidades de compilación.

Compilador: El módulo compilador traduce las unidades de compilación a código nativo. A medida que se finaliza la traducción de una unidad, la versión en código nativa se deja disponible al intérprete.

Intérprete: El módulo intérprete se responsabiliza de la ejecución actual del programa. Comienza interpretando el código fuente, haciendo saltos a las versiones en código nativo a medida que éstas están disponibles.

Monitor: Se encarga de coordinar la comunicación entre los dos módulos anteriores.

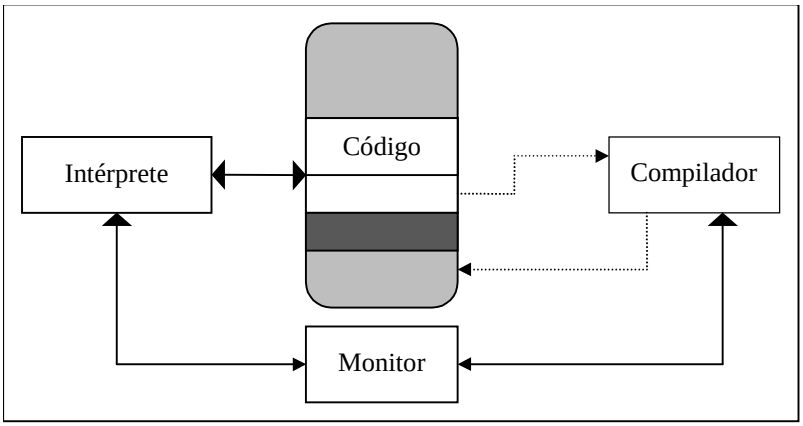

Figura 12: Compilación Continua

La principal ventaja de la compilación continua respecto a la compilación *Just in Time* radica en no tener que esperar a compilar una unidad para comenzar su ejecución. En [Plezbert 96], [Plezbert 97] se estudian diferentes estrategias y se presenta con más detalle este modelo de compilación.

A modo de repaso de las estrategias *tradicional, Just in time* y *continua*, considérese el siguiente problema:

[1] Obsérvese que la unidad B, podría no estar cargada en la máquina, siendo necesario acceder a ella a través de la red para obtener sus códigos de byte.

"Desde una máquina cliente, desea ejecutarse un programa P cargado en un servidor. El programa consta de 3 módulos 'A', 'B' y 'C' en códigos de bytes. El tiempo de transmisión de cada módulo desde el servidor al cliente es de 2sg. El tiempo de compilación de códigos de bytes a código nativo es de 0.2sg por módulo y el tiempo de ejecución interpretando códigos de bytes es el doble del tiempo de ejecución en código nativo.

Estudiar una ejecución particular que comienza por el módulo A, llama al módulo C y, tras ejecutar el módulo C, finaliza. Si dicha ejecución se realiza en código nativo, el módulo A tarda 1.4sg. mientras que el módulo C tarda 1sg. Los tiempos de transmisión de órdenes entre el servidor y el cliente se consideran despreciables.

Se trazará un diagrama de secuencias a fin de mostrar las diferencias entre cada estrategia.

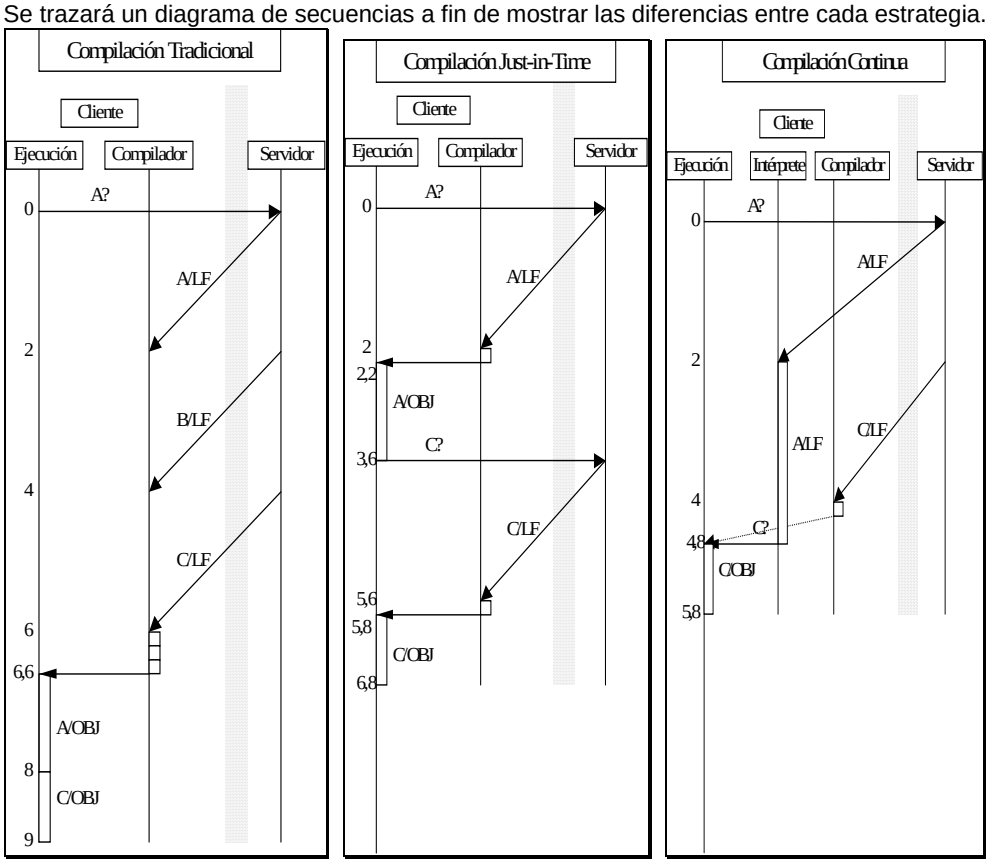

Figura 13: Diagrama de Secuencias comparando técnicas de compilación

En el **esquema tradicional**, el sistema solicita el módulo A al servidor y éste envía todos los módulos del programa (A, B y C). Puesto que cada módulo tarda 2sg en transmitirse, la transmisión finaliza a los 6 sg, momento en el que se compilan los tres módulos (tardando 0,2 sg por módulo). Al finalizar la compilación de los tres módulos, comienza la ejecución por el módulo A. Por tanto, el **tiempo de retardo**, desde que el usuario solicita el módulo hasta que éste comienza a ejecutarse es de 6,6sg. Tras ejecutarse el módulo A, se ejecuta el módulo C y el **tiempo total de ejecución** del programa serían 9 sg.

En el esquema ***Just-in-time*** el servidor envía solamente el módulo A (2 sg) , el cual es compilado en 0,2sg. El **tiempo de retardo** será 2,2 sg. Al finalizar la ejecución de A, se solicita el módulo C al servidor, tras recibirlo y compilarlo, comienza su ejecución y el **tiempo total de ejecución** serán 6,8 sg.

Finalmente, en el esquema de **compilación continua**, tras solicitar y recibir el módulo A, el cliente comienza la ejecución interpretando códigos de bytes. El **tiempo de retardo** para el usuario serán 2 sg . El tiempo de ejecución del módulo A será 2,8 sg, el doble de 1,4 sg, puesto que se interpretan códigos de bytes. Mientras se realiza la interpretación, el cliente solicita otros módulos. En un caso óptimo, el cliente solicitaría el módulo C y éste se compilaría. Al finalizar la ejecución de A, podría continuarse con la ejecución de C. El tiempo de ejecución total sería 5,8 sg.

Obsérvese que se ha considerado una situación óptima en la que el cliente solicite el módulo adecuado. El cliente podría haberse equivocado y solicitar el módulo B, comportándose peor este esquema.

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”; };
      . . .
};

. . .
```

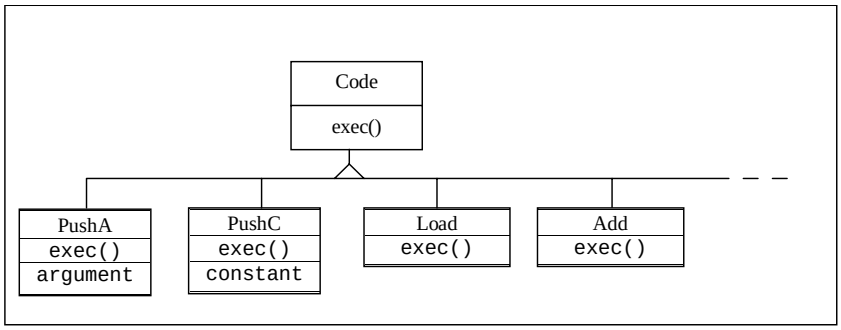

Figura 15: Representación de códigos de instrucción

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

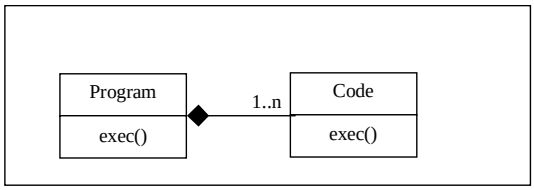

Figura 16: Representación de Programas

```
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`)

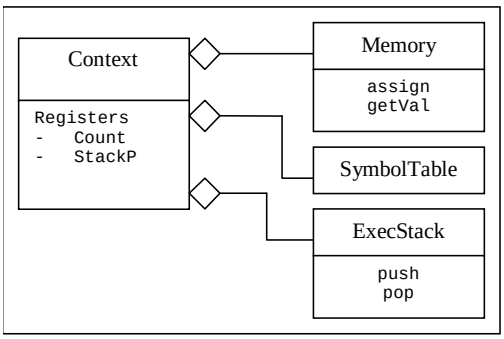

Figura 17: Contexto de ejecución

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.

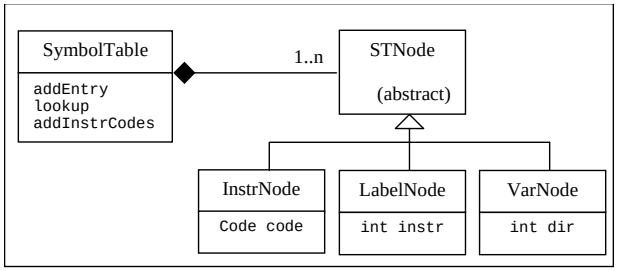

Figura 18: Tabla de Símbolos

```
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.

2. Diseño de Lenguajes de Programación

2.1. Aspectos lingüísticos

2.1.1. Lenguajes como instrumentos de comunicación

El *lenguaje natural* es el principal instrumento de comunicación utilizado por los seres humanos. Para que exista comunicación, debe existir una comprensión mutua de cierto conjunto de símbolos y reglas del lenguaje.

Los lenguajes de programación tienen como objetivo la construcción de programas, normalmente escritos por personas humanas. Estos programas se ejecutarán por un computador que realizará las tareas descritas. El *programa* debe ser comprendido tanto por personas como por computadores.

La utilización de un *lenguaje de programación* requiere, por tanto, una comprensión mutua por parte de personas y máquinas. Este objetivo es difícil de alcanzar debido a la naturaleza diferente de ambos. En un lenguaje natural, el significado de los símbolos se establece por la costumbre y se aprende mediante la experiencia. Sin embargo, los lenguajes de programación se definen habitualmente por una autoridad, que puede ser el diseñador individual del lenguaje o un determinado comité.

Para que el computador pueda comprender un lenguaje humano, es necesario diseñar métodos que traduzcan tanto la estructura de las frases como su significado a código máquina. Los diseñadores de lenguajes de programación construyen lenguajes que saben cómo traducir o que creen que serán capaces de traducir. Si los computadores fuesen la única audiencia de los programas, éstos se escribirían directamente en código máquina o en lenguajes mucho más mecánicos.

Por otro lado, el programador debe ser capaz de leer y comprender el programa que está construyendo y las personas humanas no son capaces de procesar información con el mismo nivel de detalle que las máquinas.

Los lenguajes de programación son, por tanto, una solución de compromiso entre las necesidades del emisor (programador - persona) y del receptor (computador - máquina).

De esa forma, las declaraciones, tipos, nombres simbólicos, etc. son concesiones de los diseñadores de lenguajes para que los humanos podamos entender mejor lo que se ha escrito en un programa. Por otro lado, la utilización de un vocabulario limitado y de unas reglas estrictas son concesiones para facilitar el proceso de traducción.

En 1938, C. Morris [Morris71] realiza una división del estudio de los signos (semiótica) en tres partes:

- Sintaxis: relación de los signos entre sí
- Semántica: Relación de los signos con los objetos a los que se aplican
- Pragmática: Relación de los signos con sus intérpretes

Adaptando dichas definiciones al caso particular de lenguajes de programación, la sintaxis se refiere al formato de los programas del lenguaje, la semántica estudia el comportamiento de los programas y la pragmática estudia aspectos relacionados con las técnicas empleadas para la construcción de programas.

2.2. Principios de diseño

Una pregunta natural al estudiar los lenguajes de programación es si existe un lenguaje perfecto. Si existiese tal lenguaje, entonces sería importante identificar sus características y no perder el tiempo utilizando lenguajes imperfectos.

Al diseñar lenguajes de programación a menudo es necesario tomar decisiones sobre las características que se incluyen de forma permanente, las características que no se incluyen pero que existen mecanismos que facilitan su inclusión y las que no se permiten.

Estas decisiones pueden afectar al diseño final del lenguaje y a menudo entrar en conflicto con otros aspectos del lenguaje.

A continuación se resumen algunos principios de diseño de lenguajes de programación:

- ***Concisión notacional***: el lenguaje proporciona un marco conceptual para pensar algoritmos y expresar dichos algoritmos con el nivel de detalle adecuado. El lenguaje debe ser una ayuda al programador (incluso antes de comenzar a codificar) proporcionando un conjunto de conceptos claro, simple y unificado. La sintaxis debe ser legible por el programador (o por otras personas que vayan a utilizar esos programas). Deben buscarse soluciones de compromiso entre lenguajes demasiado crípticos (por ejemplo, C) y lenguajes demasiado prolijos (Cobol, XSLT).
- ***Ortogonalidad***. Dos características de un lenguaje son ortogonales si pueden ser comprendidas y combinadas de forma independiente. Cuando las características del lenguaje son ortogonales, el lenguaje es ***más sencillo de comprender***, porque hay menos situaciones excepcionales a memorizar. La ortogonalidad ofrece la posibilidad de combinar características de todas las formas posibles (sin excepciones). La falta de

ortogonalidad puede suponer la enumeración de situaciones excepcionales o la aparición de incoherencias. Un ejemplo de falta de ortogonalidad es la limitación que impone Pascal para que una función devuelva determinados tipos de valores.

- ***Abstracción***. El lenguaje debe evitar forzar a los programadores a tener que enunciar algo más de una vez. El lenguaje debe permitir al programador la identificación de patrones repetitivos y automatizar tareas mecánicas, tediosas o susceptibles de cometer errores. Ejemplos de técnicas de abstracción son los procedimientos y funciones, la genericidad, los lenguajes de patrones de diseño, etc.
- ***Seguridad***. La fiabilidad de los poductos software es cada vez más importante. Lo ideal es que los programas incorrectos no pertenezcan al lenguaje y sean rechazados por el compilador. Por ejemplo, los sistemas con chequeo de tipos establecen restricciones a los posibles programas que pueden escribirse en un lenguaje para evitar que en tiempo de ejecución se produzcan errores. Existen lenguajes como Charity que garantizan la terminación de sus programas [Charity].
- ***Expresividad***. El programador debe poder expresar sus intenciones. En ocasiones, demasiada expresividad puede implicar falta de seguridad. De hecho, algunos sistemas limitan la expresividad para mejorar la fiabilidad de los programas (por ejemplo, la aritmética de punteros no es permitida en algunos lenguajes).
- ***Extensiblidad***. El lenguaje debe facilitar mecanismos para que el programador pueda aumentar la capacidad expresiva del lenguaje añadiendo nuevas construcciones. En Haskell, por ejemplo, el programador puede definir sus propias estructuras de control.
- ***Portabilidad***. El lenguaje debe facilitar la creación de programas que funcionen en el mayor número de entornos computacionales. Este requisito es una garantía de supervivencia de los programas escritos en el lenguaje y, por tanto, del propio lenguaje. Para conseguir la portabilidad, es necesario limitar las características dependientes de una arquitectura concreta.
- ***Eficiencia***. El programador debe poder expresar algoritmos suficientemente eficientes o el lenguaje debe incorporar técnicas de optimización de los programas escritos en él.
- ***Librerías e interacción con el exterior***. La innclusión de un conjunto de librerías que facilita el rápido desarrollo de aplicaciones es una componente esencial de la popularidad de los lenguajes. Si no se dispne de tales librerías, es necesario contemplar mecanismos de enlace con otros lenguajes que facilitan la incorporación de librerías externas.
- ***Entorno***. Aunque el entorno no forma parte del lenguaje, muchos lenguajes débiles técnicamente son ampliamente utilizados debido a que disponen de un entorno de desarrollo potente o agradable. De la misma forma, la disposición de documentación, ejemplos de programas e incluso programadores pueden ser factores clave de la popularidad de un lenguaje de programación.

2.3. Definición de un lenguaje

Cuando se extiende la utilización de un lenguaje de programación, es fundamental disponer de una definición completa y precisa del lenguaje que permita desarrollar implementaciones para diferentes entornos y sistemas. Los programas escritos en un lenguaje determinado, deben poder ser procesados por cualquier implementación de dicho lenguaje.

El proceso de estandarización se desarrolla como respuesta a esta necesidad. El estándar de un lenguaje es una definición formal de la sintaxis y semántica. Debe ser completo y no ambigüo. Los aspectos del lenguaje que son definidos, deben quedar claramente especificados, mientras que algunos aspectos que se salgan de los límites del estándar debe ser claramente designados como “indefinidos”. El procesador de un lenguaje que implementa un estándar debe ajustarse a todos los aspectos definidos, mientras que en los aspectos indefinidos puede recurrir a soluciones propias.

La autoridad que define el estándar de un lenguaje o que cambia la definición de un lenguaje puede variar desde un diseñador individual a una agencia de estandarización como ANSI, ISO o ECMA. En el caso de las agencias de estandarización, es habitual recurrir a la creación de un comité de personas con orígenes diversos (gente del mundo industrial o del mundo académico). El proceso no suele ser fácil ya que es necesario decidir entre numerosos dialectos y combinaciones de ideas. En numerosas ocasiones, el proceso llega a durar varios años y durante el mismo se producen diversas versiones a menudo incompatibles entre sí.

El primer estándar de un lenguaje a menudo limpia algunas ambigüedades, fija algunos defectos obvios y define un lenguaje mejor y más portable. Los implementadores de dichos lenguajes deben entonces realizar un proceso de ajuste de sus implementaciones para adoptar el estándar. En dicho proceso aparecen varios tipos de desviaciones:

- **Extensiones**: Numerosas implementaciones añaden nuevas características al estándar sin romper la compatibilidad con el mismo.
- **Modificaciones:** En ocasiones, los implementadores de un lenguaje consideran necesario modificar algunas características del estándar. Este tipo de modificaciones puede perjudicar la compatibilidad de los programas escritos en el lenguaje.

- **Errores:** Finalmente, las implementaciones de un lenguaje pueden desviarse del estándar sin pretenderlo, bien por una falta de comprensión de la especificación, bien por un error de la implementación. Los errores o *bugs* de las implementaciones son muy abundantes y peligrosos.

Independientemente de la postura que se tenga respecto al estándar de un lenguaje, a la hora de construir una aplicación es necesario tomar con precaución la decisión de incluir características no estándar. Cada inclusión de una característica no estándar de un lenguaje supone un paso atrás en la portabilidad del programa y decrementa su posterior usabilidad y tiempo de vida. Los programadores que utilizan características no estándar en sus programas deberían segregar los segmentos con dichas características y documentarlos claramente.

2.4. Técnicas de Especificación semántica

Es importante distinguir entre la sintaxis y semántica de un lenguaje de programación. La sintaxis describe la estructura aparente del lenguaje: qué constituye un *token*, un bloque, un procedimiento, un identificador, etc. La semántica asume que el programa ya ha sido analizado sintácticamente y relaciona la estructura del programa con su comportamiento: qué hace el programa, qué cálculos realiza, qué muestra por pantalla, etc.

En general, la sintaxis es más fácil de afrontar. En la definición de Algol 60, se utilizó con gran éxito la notación BNF (Backus-Naur Form) para especificar la sintaxis del lenguaje. Desde entonces, esta notación ha sido aceptada universalmente y ha suplantado a otras técnicas de especificación sintáctica.

Sin embargo, no existe una notación aceptada universalmente de especificación semántica. Por el contrario, se han inventado un gran número de notaciones y se siguen inventando nuevas notaciones de forma regular. La razón es que la descripción del comportamiento de los programas tiene una mayor complejidad que la de su estructura.

La búsqueda de técnicas de especificación semántica se ve motivada por las siguientes aplicaciones:

- En el **diseño** de lenguajes de programación. Los diseñadores necesitan una técnica que les permite registrar las decisiones sobre construcciones particulares del lenguaje y descubrir posibles irregularidades u omisiones.
- Durante la **implementación** del lenguaje, la semántica puede ayudar a asegurar que la implementación se comporta de forma adecuada.
- La **estandarización** del lenguaje se debe realizar mediante la publicación de una semántica no ambigüa. Los programas deben poder transportarse de una implementación a otra exhibiendo el mismo comportamiento.
- La **comprensión** de un lenguaje por parte del programador requiere el aprendizaje de su comportamiento, es decir, de su semántica. La semántica debe aclarar el comportamiento del lenguaje y sus diversas construcciones en términos de conceptos familiares, haciendo aparente la relación entre el lenguaje considerado y otros lenguajes familiares para el programador.
- La semántica asiste al programador durante el **razonamiento** sobre el programa: verificando que hace lo que se pretende. Esto requiere la manipulación matemática de programas y significados para poder demostrar que los programas cumplen ciertas condiciones.
- Finalmente, el estudio de las especificaciones semánticas ayudará a comprender las relaciones entre diferentes lenguajes de programación, aislando propiedades comunes que permitan avanzar la **investigación** de nuevos lenguajes de programación.

Las principales técnicas de especificación semántica de un lenguaje de programación son:

- Descripción en **lenguaje natural**: La especificación de la mayoría de los Lenguajes desde Fortran hasta Java se ha realizado en lenguaje natural más o menos formal. Las descripciones en lenguaje natural acarrean una serie de problemas como la falta de rigurosidad, la ambigüedad, etc. que dificultan la verificación formal de programas y corrección de las implementaciones.
- **Implementaciones prototipo**: Se define un intérprete estándar para el lenguaje que funciona en una determinada máquina. El problema es decidir qué lenguaje se utiliza en la construcción de dicho intérprete. En muchas ocasiones se utilizan lenguajes ya implementados con el fin de ofrecer especificaciones ejecutables. En otras ocasiones, se utiliza el mismo lenguaje que se está definiendo. En todos los casos, se produce una sobre-especificación (es necesario especificar no solamente el lenguaje objeto, sino el lenguaje de implementación).
- **Semántica denotacional** [Tennent 94]: Se describe el comportamiento del programa modelizando los significados (efectos de las diversas construcciones del lenguaje) mediante entidades matemáticas. La denotación de una programa se considera como una función del estado anterior al comienzo de la ejecución al estado final.
- **Semántica operacional**: Los significados del programa son descritos en términos operacionales. Se utiliza un lenguaje basado en reglas de inferencia lógicas en las que

se describen formalmente las secuencias de ejecución de las diferentes instrucciones en una máquina abstracta.

- **Semántica axiomática**: Emplea un sistema formal de razonamiento sobre los significados de los programas como una descripción del comportamiento del lenguaje. El sistema permite estudiar formalmente las propiedades del lenguaje y se requiere la utilización de sistemas consistentes y completos que no siempre son asequibles.

Existen sistemas híbridos como la semántica de acción de Mosses o la semántica monádica modular que facilitan la legibilidad sin perjuicio de la rigurosidad matemática. Permitiendo, además, la automatización de la construcción de prototipos.

2.5. Características de Lenguajes

2.5.1. Representación

Una representación de un objeto es conjunto de hechos relevantes sobre ese objeto. Una representación en un ordenador de un objeto es una asignación de los hechos relevantes de un objeto a elementos del ordenador.

Algunos lenguajes soportan ***representaciones de alto nivel*** que especifican propiedades funcionales o nombres simbólicos y tipos de datos de los campos de la representación. Dicha representación será asignada a una determinada porción de memoria por el procesador. El número y orden de bytes necesarios para representar el objeto puede variar entre un procesador y otro.

Por el contrario, una ***representación de bajo nivel*** si describe una implementación particular del objeto en un ordenador, como la cantidad de bytes y la posición de cada campo.

2.5.2. Tipos básicos

Los lenguajes de programación contienen un repertorio de tipos básicos o primitivos junto con una serie de operaciones sobre dichos tipos. Los tipos básicos más habituales son:

- Números enteros: Normalmente se representan mediante un número de bytes fijo, limitando su rango. Algunos lenguajes, como Haskell, incluyen una representación ilimitada.
- Caracteres: Como en el caso anterior, suelen representarse mediante un número fijo de bytes. En algunos lenguajes, los caracteres se identifican con los enteros.
- Números reales representados en punto flotante. La representación también suele realizarse mediante un número fijo de bytes, limitando la precisión de los programas.
- Booleanos: Con los valores verdadero o falso.
- Referencias. Algunos lenguajes incluyen un tipo básico que se utiliza para hacer referencia a otros elementos. Estas referencias pueden implementarse mediante direcciones de memoria.

Cada tipo básico contiene un conjunto de operaciones primitivas. Por ejemplo, los enteros y flotantes incluyen operaciones aritméticas, los caracteres, operaciones de conversión y los booleanos, operaciones lógicas.

Los lenguajes con **chequeo estático de tipos** permiten comprobar en tiempo de compilación que en tiempo de ejecución no se van a producir errores de tipos. El chequeo estático de tipos aumenta la seguridad de los programas, al detectar errores antes de la ejecución. Otra ventaja es la eficiencia, ya que en la fase de ejecución no es necesario realizar comprobaciones de tipo.

Otros lenguajes, como LISP, BASIC, Perl, Prolog, etc. no incluyen chequeo estático de tipos. Las ventajas de no incluirlo sn una mayor flexibilidad (es posible construir más programas) y sencillez para el programador. El programador no se preocupa de incluir declaraciones de tipos y los programas dan menos *errores de tipo* al compilar (aunque pueden darlos al ejecutarse).

Algunos lenguajes, como Haskell o ML, incluyen además un **sistema de inferencia de tipos**. El programador no tiene obligación de declarar el tipo de las expresiones, ya que el sistema es capaz de inferirlo. En caso de que el programador lo hubiese declarado, el sistema puede comprobar que el tipo declarado coincide con el tipo inferido.

2.5.3. Tipos de datos compuestos

Además de la inclusión de tipos de datos básicos o primitivos. Los lenguajes de programación incluyen mecanismos para definir nuevos tipos de datos a partir de dichos tipos básicos. Los principales mecanismos son:

- **Enumeración:** Se define un tipo de datos como la enumeración de un conjunto de valores básicos.
- **Productos:** Se define un tipo de datos como el producto cartesiano de varios conjuntos. En numerosas ocasiones, a dichos conjuntos se les asigna un nombre, denominándose ***estructuras*** o ***registros***.

```
type Person = Record { name    : String,
                       age     : Int,
                       married : Bool
                     }
```

Es posible utilizar definiciones recursivas en los tipos producto:

```
type Tree = Record {info : Integer,
                    left : Tree,
                   right: Tree
                  }
```

- **Uniones:** Un tipo de datos puede definirse como la unión de un conjunto de tipos. En las uniones también se pueden dar nombres a los componentes y pueden también denominarse ***variantes.***

```
type Event = Union { keyboard : Char,
                     mouse : Point
                   }
```

- **Listas o Arrays**: Un array define una lista ordenada de valores de un determinado tipo. Aunque a nivel teórico, los arrays pueden representarse mediante registros o funciones, en la práctica, muchos lenguajes incluyen la posibilidad de definir arrays. El lenguaje incorpora además unas operaciones predefinidas para inicializar y obtener elementos que ofrecen mayor eficiencia que otros tipos de representación. La representación interna de los arrays en algunos lenguajes, como C y Pascal, se realiza mediante una porción de memoria en la que los elementos aparecen de forma consecutiva. Para ello, el programador debe especificar a priori el tamaño del array.

Muchos lenguajes permiten asignar nombres a los tipos compuestos favoreciendo la mantenibilidad de las aplicaciones.

Los lenguajes orientados a objetos permiten definir y restringir el conjunto operaciones disponibles sobre los tipos de datos compuestos. De esa forma, favorecen el control que el programador puede tener sobre las posteriores modificaciones del código.

2.5.4. Expresiones y Operadores

Las expresiones de un lenguaje forman el núcleo del lenguaje y se utilizan para definir valores. Las expresiones se forman a partir de un conjunto de funciones y operadores predefinidos o definidos por el usuario. Las expresiones pueden incluir variables cuyo valor debe buscarse en el contexto particular en el que se evalúan. Por ejemplo:

```
2 * pi + sin x / distancia(x,3)
```

Las expresiones pueden representarse mediante un árbol, cuyos nodos son los operadores o funciones y cuyas hojas son los operandos.

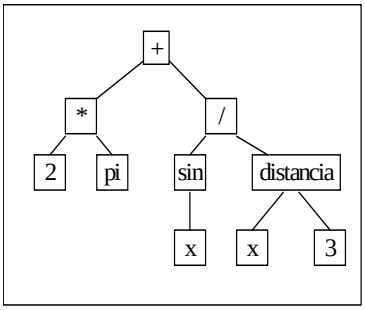

Figura 20: Árbol de la expresión `2 * pi + sin x / distancia(x,3)`

Mediante la ***definición de operadores del usuario,*** algunos lenguajes permiten que el programador defina un conjunto de operadores propios además de los operadores predefinidos. La definición de estos operadores suele requerir la especificación de la asociatividad y la precedencia para resolver el análisis de expresiones.

A continuación se indica cómo es posible añadir un operador definido por el usuario en el lenguaje Haskell

```
infixr <*>

x <*> y = (x + y) * (x – y)

main = print (5 <*> 2)
```

Obsérvese que internamente, un operador puede representarse de la misma forma que una función. La diferencia es a nivel sintáctico, no semántico.

La ***sobrecarga de operadores*** consiste en redefinir operadores ya existentes para que puedan aceptar como argumentos nuevos tipos de datos. Un ejemplo de aplicación es la la construcción de una librería de funciones polinómicas que redefina los operadores aritméticos para que puedan aceptar polinomios como argumentos.

2.5.5. Declaraciones y Ámbitos

Una de las características más importantes de los lenguajes de programación que permiten simplificar la definición de expresiones complejas es la utilización de declaraciones locales. La expresión:

```
(245 + 3) * (245 + 3) – 7 * (245 + 3)
```

puede definirse como

```
let v = 245 + 3 in v * v – 7 * v
```

Las declaraciones permiten asociar a un nombre un objeto en un contexto. Una posible sintaxis sería:

```
let v = objeto in exp
```

Pueden utilizarse varias de claraciones.

```
let x = 12 + 3
    y = 12
in x * x + 2 * x * y
```

En general, en los lenguajes de programación, la utilización de declaraciones aparece en múltiples contextos. Por ejemplo, los programas en C pueden considerarse como una serie de declaraciones de funciones. En este caso, las declaraciones asocian a un identificador (nombre de la función) una definición de la función (que incluye el tipo y el código a ejecutar).
Por ejemplo, el programa:

```
int f (int x) {
 …
}

float g (int x) {
…
}

int main() {
…
}
```

Podría considerarse como una serie de declaraciones locales:

```
let f = ...
    g = ...
    main = …
in main
```

El sistema de módulos de los lenguajes también puede considerarse como una serie de declaraciones que asocian a un identificador (nombre del módulo) un objeto (conjunto de declaraciones).

Las declaraciones también pueden hacer referencia a otras definiciones e incluso a sí mismas, denominándose **declaraciones recursivas**. El siguiente programa imprime “Hola” un número *potencialmente infinito* de veces.

```
let x = write (“Hola”) ; x
in  x
```

Otra posibilidad es permitir **declaraciones mutuamente recursivas**. El siguiente programa imprime “HolaAdios” un número *potencialmente infinito* de veces.

```
let x = write (“Hola”) ; y
    y = write (“Adios”); x
in  x
```

La evaluación de las expresiones con declaraciones locales requiere la utilización de un entorno o contexto en el cual se busca el valor de la variable. El ámbito de una expresión define el conjunto de asociaciones de variables a valores que se utilizan al evaluar dicha expresión.

Las variables pueden obtener sus valores del entorno en el que se definen (***ámbito estático***) o del entorno en el que son utilizadas (***ámbito dinámico***). Por ejemplo, al evaluar la expresión

```
let x = 1
    y = x
    z = let x = 2
        in y + x
in z
```

con ámbito estático se obtiene un 3 ya que el valor de la variable y es x y en el entorno en el que se define x toma el valor 1. Sin embargo, con ámbito dinámico se obtendría 4, ya que el valor de x en el entorno en que se utiliza es 2.

La mayoría de los lenguajes estructurados en bloques utilizan ámbito estático. Históricamente, el ámbito dinámico fue utilizado por los lenguajes LISP, SNOBOL4 y APL. Aunque en los últimos tiempos ha estado un poco denostado, existen varios lenguajes ampliamente utilizados que emplean ámbito dinámico como Postscript, TeX, Tcl y Perl. De hecho su utilización restringida puede facilitar el mantenimiento de aplicaciones y librerías que trabajan en entornos altamente configurables [Hanson, 01], [Lewis, 00].

2.5.6. Variables, referencias y Asignaciones

La utilización de declaraciones permite definir variables como identificadores a los que se asocia un valor en un determinado ámbito. Este tipo de declaraciones son las que se utilizan tradicionalmente en otras disciplinas, como las matemáticas. La asociación de un valor a una variable se realiza externamente y no puede modificarse en la expresión. Así, en la expresión:

$$\sum_{i=0}^{n} 2 \times i \times a_i$$

La variable i toma valores de 0 a n en la expresión interna. Dentro de dicha expresión no es posible modificar el valor de la variable.

La principal característica de un lenguaje imperativo es la utilización de ***variables actualizables.*** El valor de estas variables hace referencia a una porción de memoria que puede modificarse durante la ejecución del programa. Una instrucción de modificación suelen ser la asignación destructiva

```
x := expresión
```

Esta instrucción evalúa la expresión y asigna a la variable x el valor obtenido.

En una expresión del tipo

```
x := x + 1
```

existe una diferencia semántica entre las dos apariciones del nombre x. En la parte izquierda de la asignación, x se refiere a la variable en sí, mientras que en la parte derecha, x se refiere al valor de la variable.

La utilización de asignaciones puede enturbiar la comprensión de los programas. ya que el valor de las variables va a depender del momento en que se evalúan.

2.5.7. Control de la Ejecución

El operador ***secuencia*** toma dos instrucciones como argumentos, ejecuta la primera instrucción y al finalizar ésta, ejecuta la segunda. El operador se denota habitualmente mediante punto y coma.

```
x := 1; x := x + 2
```

La construcción más elemental para cambiar el flujo de control de un programa es la utilización de etiquetas y sentencias ***goto***. Aunque la utilización de este tipo de saltos puede limitar la capacidad de razonamiento sobre los programas [Dijkstra 68]. Aunque la programación estructurada promueve la limitación de su uso, existen múltiples lenguajes actuales que soportan esta construcción.

El operador ***condicional***

```
if cond then instThen
        else instElse
```

evalúa la condición y, si se cumple, ejecuta las instrucciones instThen, si no se cumple, ejecuta instElse. En los lenguajes imperativos, la parte **else** es opcional.

El operador de selección elige la instrucción a ejecutar en base a una serie de condiciones (o al valor de una expresión). Habitualmente se incluye una alternativa (***otherwise***) para contemplar la situación en que ninguna de las condiciones se cumpla. Una posible sintaxis podría ser:

```
switch {
       case (x > 5) : write (“X > 5”);
       case (x < 0) : write (“X < 0 ”);
       otherwise: write (“X between 0 and 5”);
}
```

Las sentencias repetitivas ejecutan reiteradamente una serie de instrucciones hasta que se cumpla una condición.

```
y := 1;
while (x > 0) {
  x := x – 1;
  y := y * x
}
```

Muchos lenguajes de alto nivel incluyen un mecanismo de excepciones para manejar posibles situaciones erróneas. Estas situaciones son señaladas lanzando (*throw*) una excepción que es capturada (*catch)* por el manejador (*handler*) correspondiente.

Se consideran ***excepciones síncronas*** las que son lanzadas como consecuencia de la ejecución del programa, por ejemplo: división por cero, accesos a arrays fuera de rango, etc. Estas excepciones aparecen en un punto concreto de la ejecución debido a la evaluación de una expresión o a la ejecución de alguna instrucción. Lenguajes como ML, Java, C#, Haskell, Prolog permiten un tratamiento estructurado de estas excepciones mediante bloques similares al siguiente:

```
try
  ...código a ejecutar (con posibles errores)
catch
   DivideByCero    -> write (“Division by zero”);
   FileDoesntExist -> write(“File does not exist”);
   e               -> write(“Unknown Excepción :” + e);
```

El sistema ejecuta el código de la sentencia `try` de forma normal. Si se lanza una excepción, se selecciona y ejecuta el manejador que encaja con dicha excepción.

Habitualmente, los lenguajes permiten que el programador pueda definir nuevos tipos de excepciones y lanzarlas cuando lo estime oportuno.

Las ***excepciones asíncronas*** [Marlow 01] son aquellas que se producen por causas ajenas a la ejecución del programa. Por ejemplo, interrupciones del usuario, finalización de tiempos límite, errores del sistema, etc. Este tipo de excepciones son difíciles de capturar en entornos con concurrencia ya que pueden surgir en cualquier momento de la ejecución del programa.

2.5.8. Mecanismos de Abstracción

Una de las principales características de la programación es facilitar la abstracción. Cuando un patrón se repite en numerosas ocasiones, es posible definir una abstracción que capture la parte común y tome como parámetros las partes que varían. La notación lambda suele utilizarse para especificar abstracciones (`λx . x * x`) indica una abstracción que toma un parámetro x y devuelve x * x.

La aplicación de una abstracción consiste simplemente en substituir el parámetro por un valor concreto, denominado argumento. Al evaluar

```
(λx . x * x ) 3
```

se obtiene 3 * 3 = 9.

Las abstracciones en los lenguajes de programación toman el nombre de funciones o procedimientos. Es posible combinar abstracciones con declaraciones locales.

```
let f = λx . x * x – 7 * x
    v = 245 + 3
in f v
```

Otra posibilidad es la utilización de definiciones recursivas

```
let fact = λx . if x = 0 then 1
                         else x * fact (x – 1)
in fact 5
```

Obsérvese que la utilización de definiciones recursivas puede generar computaciones potencialmente infinitas. Por ejemplo, si en la expresión anterior se substituye el 5 por (-5).

La notación lambda se utiliza habitualmente en lenguajes funcionales. La definición anterior suele escribirse también como:

```
let fact x = if x = 0 then 1
                      else x * fact (x – 1)
in fact 5
```

Una de las características que distinguen a los lenguajes funcionales es la utilización de **funciones de orden superior**, es decir, funciones que pueden tomar como argumentos otras funciones y que pueden devolver como valores funciones.

```
let reapply f = λx . f (f x)
    double x = x * 2
in reapply double 5
```

Al evaluar la expresión anterior se obtendría

```
reapply double 5 → double (double 5) → double (5 * 2) → double 10 → 10 * 2 → 20
```

Mediante ***curryficación***, las funciones de varios argumentos pueden simularse mediante funciones de orden superior de un argumento.

```
add = λx . λy . x + y
```

La ventaja de definiciones como la anterior es que la expresión `(add 3)` tiene sentido por sí misma (denota una función que al pasarle un número, le suma 3).

Existen varios *mecanismos de paso de parámetros*:

1. El ***paso por valor***, utilizado por la mayoría de los lenguajes imperativos consiste en evaluar los argumentos antes de la llamada a la función (también se conoce como ***evaluación ansiosa*** (eager). Ejemplo

   ```
   (λx . x * x) (2 + 3) → (λx . x * x) 5 → 5 * 5 → 25
   ```

   La principal ventaja de este mecanismo es su eficiencia y facilidad de implementación.

2. En el ***paso por nombre,*** se evalúan antes las definiciones de función que los argumentos. Ejemplo:

   ```
   (λx . x * x) (2 + 3) → (2 + 3) * (2 + 3) → 5 * (2 + 3) → 5 * 5 → 25
   ```

   Este tipo de evaluación se conoce como **evaluación *normal***, y no suele utilizarse en la práctica debido a su menor eficiencia.

3. El ***paso por necesidad*** evita el problema de la eficiencia de la evaluación normal mediante la utilización de una especie de grafo. También se conoce como ***evaluación perezosa*** (lazy) o ***just-in-time*** [Meijer 01]***.*** Se utiliza habitualmente en enguajes puramente funcionales como Haskell y Miranda.

   ```
   (λx . x * x) (2 + 3) → . * . → . * . → 25
                           \ /     \ /
                          2 + 3     5
   ```

   Una ventaja de este tipo de evaluación es la posibilidad de definir y trabajar con estructuras potencialmente infinitas

4. El ***paso por referencia*** es utilizado por lenguajes imperativos con el fin de aumentar la eficiencia del paso por valor. Se transmite la dirección de memoria del valor, el cual puede modificarse en la definición de la función. El paso de parámetros por referencia

puede ser útil cuando el valor ocupa mucho espacio de memoria y resulta interesante no duplicar dicho valor en posibles llamadas recursivas.

```
void cambia(int &x, int &y) {
 int z;
 z := x; x:= y; y := z;
}
```

2.5.9. Entrada/Salida

Cualquier lenguaje de programación debe ofrecer mecanismos de interacción con el mundo exterior. A nivel semántico, el estudio de programas interactivos se complica debido a la presencia de efectos laterales.

Los lenguajes incluyen varias instrucciones predefinidas que interactúan con el exterior mediante flujos de caracteres (*streams*). La popularidad de los sistemas Unix han influido en la utilización habitual de 3 flujos predefinidos: stdin, stdout y stderr, que indican la entrada estándar, la salida estándar y la salida de errores.

Otro mecanismo habitual de comunicación con el exterior inspirado en los sistemas Unix es el paso de argumentos en línea de comandos y la lectura de variables de entorno. Estos mecanismos de comunicación son empleados por el protocolo CGI (*Common Gateway Interface)* para la transmisión de información a programas generadores de información dinámica en la Web.

Una decisión a tomar en el diseño de lenguajes es la inclusión de sistema de Entrada/Salida como un mecanismo del lenguaje o como una librería.

2.5.10. Concurrencia

Existen numerosos lenguajes que incluyen facilidades para el desarrollo de código que se ejecute mediante varios procesos concurrentes. Una característica habitual de estos lenguajes es la posible creación de programas no deterministas.

El lenguaje Java, por ejemplo, incluye facilidades para la concurrencia mediante monitores. Durante la ejecución de un programa pueden crearse múltiples hilos (*threads*) de ejecución que pueden sincronizarse mediante monitores.

2.5.11. Objetos, Clases y Herencia

Un objeto encapsula unos recursos o variables locales y proporciona operaciones de manipulación de dichos recursos. Esta encapsulación da lugar al concepto de *abstracción de datos* ya que el recurso encapsulado no puede ser manipulado directamente, sólo a través de las operaciones proporcionadas por el objeto. El conjunto de recursos encapsulados se denomina ***estado de un objeto***, mientras que las operaciones exportadas forman el conjunto de ***métodos de un objeto***.

La expresión o $\mapsto$ m se utilizará para denotar la selección del método m del objeto o (también puede leerse como el resultado de enviar el mensaje m al objeto o).

En la definición de un objeto existe la posibilidad de referirse a sí mismo mediante la variable self (en C++ y Java se utiliza la variable this). Una posible sintaxis para definir objetos sería:

```
let p = Object
      locals  x = 5
              y = 5
      methods dist = sqrt( x^2 + y^2)
              move (dx,dy) = (x := x + dx; y := y + dy )
              closer p = self ↦ dist < p ↦ dist
in p ↦ dist
```

Los lenguajes orientados a objetos más populares utilizan el concepto de *clase* como una descripción de la estructura y comportamiento de objetos.[3]

Una clase puede considerarse como un patrón que permite crear objetos denominados instancias de dicha clase. Todos los objetos generados a partir de una clase comparten los mismos métodos, aunque pueden contener valores diferentes de las variables locales.

Dada una clase C, el operador New C devuelve un nuevo objeto a partir de dicha clase (normalmente denominado instancia de C).

El siguiente ejemplo define una clase `Point`.

```
let Point = Class
      locals  x = 5
              y = 5
      methods dist = sqrt( x^2 + y^2)
```

[3] Habitualmente, se distingue la clase de un objeto, que define la estructura y el comportamiento, del tipo, que define únicamente la estructura.

```
                move (dx,dy) = (x := x + dx; y := y + dy )
                closer p = self ↦ dist < p ↦ dist
    in (p := new Point; p ↦ dist)
```

El concepto de clase no es estrictamente necesario en un lenguaje Orientado a Objetos. Existen lenguajes en los que una clase se representa como un objeto que permite generar otros objetos. En [Abadi96], los lenguajes que utilizan la noción de clases se denominan ***lenguajes basados en clases*** para distinguirlos de los ***lenguajes basados en objetos,*** o ***lenguajes basados en prototipos*** que no requieren un concepto de clase independiente.

La noción de ***herencia*** y ***subclase*** es una de las piezas básicas de los lenguajes orientados a objetos. Como cualquier clase, una subclase describe el comportamiento de un conjunto de objetos. Sin embargo, lo hace de forma incremental, mediante extensiones y cambios de una ***superclase*** ó ***clase base***. Las variables locales de la superclase son heredadas de forma implícita, aunque la subclase puede definir nuevas variables locales. Los métodos de la superclase pueden ser heredados por defecto o redefinidos por métodos de tipo similar en la subclase (En Simula y C++ sólo es posible redefinir los métodos marcados como virtuales en la superclase).

En el siguiente ejemplo[4] se declara Circle como una subclase de Point añadiendo una variable local radius con valor inicial 2 y modificando el método que calcula la distancia al origen.

```
Circle = subclassOf Point
    locals
        radius = 2
    methods
       dist = max (super ↦ dist - radius,0)
```

Sin las subclases, la aparición de self en la declaración de una clase se refiere siempre a un objeto de dicha clase. Con subclases, esta afirmación no se cumple. En un método m heredado por una subclase D de C, self se refiere a un objeto de la subclase D, no a un objeto de la clase original C.

Para poder hacer referencia a los métodos de la clase original se utiliza super.

La ***herencia múltiple*** se obtiene cuando una subclase hereda de más de una clase. Un problema inmediato que surge al heredar de varias clases es decidir qué política seguir en caso de conflictos y duplicaciones entre los métodos y variables locales de las superclases. Aunque la solución más simple es identificar y prohibir dichos conflictos, existen diversas soluciones más elegantes [Agesen 93].

Un concepto relacionado con la utilización de subclases, es el de enlace dinámico. Si D es una subclase de C que redefine el método m y se tiene un objeto o de la clase D, con ***enlace dinámico***, la expresión o ↦ m selecciona el método redefinido en la subclase D, mientras que con enlace estático se utiliza el método de la clase base.

Por ejemplo, en la siguiente definición:

```
near : Point  → Bool
near p = p ↦ dist < 6
```

La función near toma como parámetro un objeto de la clase Point y chequea si su distancia al origen es menor que 5. Dado que todos los objetos de la clase Circle pertenecen a su vez a la clase Point, es posible pasarle tanto un objeto Point como un objeto Circle. Supóngase que se declara la siguiente expresión:

```
let p = New Point
    q = New Circle
      in near p = near q
```

Al calcular `near q` se debe buscar el método `dist` de `q` que pertenece a la clase `Circle`. Existen dos posibilidades:

- Con enlace estático se utiliza el método `dist` de la clase `Point`
- Con enlace dinámico se utiliza el método `dist` de la clase `Circle`

Con enlace dinámico, a la hora de implementar la función `near` no es posible conocer qué método `dist` se va a utilizar. Esta información sólo se conoce en tiempo de ejecución ya que depende del tipo de punto que se esté usando.

El enlace dinámico ofrece un importante mecanismo de abstracción: cada objeto sabe cómo debe comportarse de forma autónoma, liberando al programador de preocuparse de examinar qué objeto está utilizando y decidir qué operación aplicar.

---

[4] Se ha escogido este ejemplo porque muestra claramente los mecanismos de redefinición de métodos. Desde el punto de vista de diseño no es un buen ejemplo, ya que un círculo no suele considerarse una subclase de un punto.

Como se ha indicado, la setencia o $\mapsto$ m puede tomar distintas formas, dependiendo del valor concreto del objeto o. Este tipo de polimorfismo se conoce como polimorfismo de inclusión. En algunos contextos se denomina simplemente polimorfismo, sin embargo, se considera más conveniente distinguirlo del polimorfismo paramétrico (o genericidad) y del polimorfismo ad-hoc (o sobrecarga) [Cardelli, 85]

2.5.12. Genericidad

En ocasiones, un mismo algoritmo es independiente del tipo de datos con el que trabaja. Por ejemplo, el algoritmo de ordenación *quicksort* debería poder aplicarse a listas de valores de cualquier tipo. En un lenguaje con chequeo estático de tipos, es importante poder definir este tipo de algoritmos genéricos sin perder la capacidad de chequeo estático.

Los lenguajes ML, Haskell y Miranda admiten la definición de funciones genéricas utilizando lo que denominan ***polimorfismo paramétrico***. Siguiendo esa misma línea, el lenguaje C++ incluyó la posibilidad de definir plantillas o ***templates***. C++ incluye una librería estándar basada en este tipo dedefiniciones genéricas denominada STL (*Standard Template Library).* El lenguaje Java no admite este tipo de definiciones, aunque se han definido algunas extensiones que sí lo admiten como GJ (Generic Java) ó Pizza [Bracha 98]

```
public class Stack<A> {
 private A[] store;

 public Stack() { store = new A[10]; }
 public void push(A elem) {
      …
 }
 public A pop() {
       …
 }
}
```

Una posible utilización de la clase anterior desde un programa sería

```
Stack<int> x = New Stack<int>();
x.push(2);
y := x.pop() + 1;
```

Los lenguajes que no incluyen genericidad, como Pascal o Java, deben recurrir a la duplicación de código (crear una versión del mismo algoritmo para cada tipo de datos que vaya a utilizarse) o utilizar versiones inseguras del algoritmo mediante la utilización de una clase padre universal (habitualmente la clase Object).

```
public class Stack {
 private Object[] store;

 public Stack() { store = new Object[10]; }
 public void push(Object elem) {
       …
 }
 public Object pop() {
       …
 }
}
```

La utilización de la clase sería

```
Stack x = New Stack();
x.push(2);
y := (int) x.pop() + 1;
```

Es necesario realizar una conversión del objeto devuelto por pop al tipo de datos deseado. Esta conversión podría ser incorrecta y no se detectaría en tiempo de ejecución. Sin embargo, mediante la utilización de *templates* se garantiza una mayor seguridad de tipos.

Se han producido varios intentos para añadir genericidad a Java [Bracha, 98],[Viroli, 00] y CLR [Kennedy01].

2.5.13. Descomposición modular

Mediante la descomposición modular, es posible facilitar el trabajo independiente de varios desarrolladores. Los requisitos de un buen sistema de descomposición modular son:

- ***Gestión de espacio de nombres***. En el desarrollo de productos software formados por miles de líneas de código es necesario disponer de un sistema que evite las colisiones entre los nombres utilizados para designar elementos. La gestión del espacio de nombres puede utilizar una ***estructura plana*** donde todos los módulos están a un mismo nivel, o una ***estructura jerárquica*** (por ejemplo, los paquetes de Java).
- ***Compilación separada*** de los módulos. Los grandes sistemas de software están formados de un gran número de módulos cuya compilación puede ser lenta. Es necesario que un cambio en un módulo no implique una recompilación de todo el sistema, sino de los módulos afectados.
- ***Independencia de la implementación***: El sistema debe facilitar la creación de una interfaz sobre la cual puedan definirse una o varias implementaciones. Una vez fijada la interfaz, los usuarios del módulo no deberán verse afectados por cambios en la implementación.
- El sistema debe verificar que la implementación encaja con la interfaz proporcionando cierta ***seguridad*** al programador. En ocasiones, se utilizan condiciones lógicas denominadas *contratos.*
- La protección de la implementación puede realizarse mediante ***mecanismos de encapsulación*** que impidan a ciertos programadores modificar módulos sin permiso. Algunos sistemas recientes proporcionan incluso sistemas de encriptación.
- ***Sistema de control de versiones.*** La modificación de módulos puede acarrear grandes problemas a los usuarios. La plataforma .NET tiene un sistema de control de versiones integrado en el sistema de módulos que contempla una solución sistemática de este tipo de problemas.
- ***Módulos de primera clase.*** Algunos lenguajes permiten controlar el propio sistema de módulos desde el mismo lenguaje.

2.6. Familias de Lenguajes

2.6.1. Lenguajes imperativos

Los lenguajes imperativos se inspiran directamente en la máquina de Von-Neumann y toman como modelo teórico las máquinas de Turing.

Algunos lenguajes imperativos:

- **FORTRAN,** desarrollado por J. Backus desde 1955 hasta 1958. Fue uno de los primeros lenguajes de cálculo científico de un relativo alto nivel. FORTRAN = FORmula Translation. Actualmente continua siendo aplicado en entornos científicos. El lenguaje facilitaba la descripción de fórmulas matemáticas, disponía de estructuras de control avanzadas (muchas de ellas basadas en la sentencia GOTO) y facilidades para definición de subprogramas. Hasta la versión FORTRAN 90, no se permitían procedimientos recursivos.
- **COBOL** (COmmon Business Oriented Language) Desarrollado a principios de los años 60 para el desarrollo de aplicaciones de gestión. El lenguaje intenta utilizar una sintaxis inspirada en lenguaje natural (inglés) con el objetivo de facilitar la legibilidad.
- **ALGOL** (ALGOrithmic Language) Fue diseñado por un comité internacional a principios de los años 60 con el propósito de definir un lenguaje universal de descripción de algoritmos. Contenía muchas características existentes en los modernos lenguajes de programación (diversos mecanismos de paso de parámetros, bloques estáticos, recursividad, funciones de primera clase, sistema estático de tipos, etc.). Se utilizó más bien en la descripción de algoritmos ya que existían pocas implementaciones del lenguaje. Para su definición sintáctica se utilizó por primera vez la notación BNF. Originalmente el lenguaje ALGOL no incluía rutinas de Entrada/Salida
- **BASIC** (Beginner's All-purpose Symbolic Instruction Code) fue desarrollado por T. Kurtz y J. Kemeny a principios de los años 60 con el objetivo de proporcionar un entorno de computación sencillo utilizable por cualquier persona no necesariamente informática. El lenguaje no requería la declaración de variables y permitía la sentencia de salto incondicional GOTO. Ha alcanzado gran popularidad debido a su adopción por parte de Microsoft (con el nombre de Visual Basic) como lenguaje de macros de los populares programas Office.
- **PASCAL**. Creado por N. Wirth en 1970 con el propósito de desarrollar un lenguaje simple y general que facilitase la enseñanza de la programación de una forma sistemática mediante un lenguaje fiable y eficiente (tanto al ejecutarse como al compilarse). Se utilizó una máquina abstracta denominada P-Code. El lenguaje se basa en ALGOL aunque simplifica muchas de sus características.
- **C**. Desarrollado a principios de los años 70 como lenguaje de programación sistemas para el sistema operativo Unix. El lenguaje fua adquiriendo popularidad emparejado a dicho sistema operativo debido a la eficiencia de los programas escritos en él y a su

nivel relativamente alto que ofrecía al programador un nivel de abstracción adecuado. Actualmente, el lenguaje sigue utilizándose exhaustivamente debido a la disponibilidad de compiladores del lenguaje en múltiples plataformas. Se ha utilizado, incluso, como lenguaje destino de muchos compiladores.

- **ADA**. Desarrollado entre 1975 y 1980 a partir de una propuesta del Departamento de Defensa de Estados Unidos que pretendía diseñar un lenguaje fiable para aplicaciones *empotradas* en sistemas en tiempo real. La especificación del lenguaje fue desarrollada antes de disponer de implementaciones. Proporciona mecanismos de definición de tipos abstractos de datos, verificación de programas, concurrencia y tratamiento de excepciones.

2.6.2. Lenguajes funcionales

Los lenguajes funcionales toman como elemento fundamental la noción de función. Una característica importante de estos lenguajes es la utilización de funciones de orden superior, es decir, funciones que pueden tomar otras funciones como argumentos y devolver funciones.

Varios lenguajes funcionales:

- **LISP** (LISt Processing). Desarrollado por J. McCarthy en 1958 con el propósito de crear un lenguaje flexible para aplicaciones de inteligencia artificial. El lenguaje utiliza una sintaxis prefija para operadores. Además utiliza una única categoría sintáctica (S-expresiones) para datos y programas y no contempla el chequeo estático de tipos. De esa forma, facilita la metaprogramación y la expresividad algorítmica. El lenguaje mantiene su popularidad en el campo de la inteligencia artificial. El lenguaje es funcional ya que permite la definición de funciones de orden superior. Sin embargo, algunos autores lo califican como lenguaje no-puramente funcional debido a la inclusión de asignaciones destructivas.
- **Scheme**. Desarrollado por G. J. Sussman y Steele Jr. en 1975 como un dialecto simplificado de LISP utilizando ámbito estático. El lenguaje sobrevivió posteriormente como lenguaje de enseñanza de la programación debido a su sencillez y a la disponibilidad de potentes entornos de programación.
- **ML**. Creado por R. Milner en 1975 como un Meta-Lenguaje del sistema de demostración automática de teoremas LCF. ML se independizó en un lenguaje de propósito general. Contiene un sistema estático de chequeo e inferencia de tipos con polimorfismo paramétrico. El lenguaje ML utiliza evaluación ansiosa y los programas pueden ser compilados, llegando a rivalizar en eficiencia con lenguajes imperativos. También contiene un potente sistema de módulos y de tratamiento de excepciones.
- **Haskell**. Desarrollado por un comité internacional en 1990. Es un lenguaje puramente funcional con evaluación perezosa y resolución sistemática de sobrecarga.

2.6.3. Lenguajes Orientados a Objetos

Una noción importante de estos lenguajes es la herencia que facilita la reutilización de código. Algunos lenguajes:

- **Simula**. Desarrollado por O. J. Dahl y K. Nygaard entre 1962 y 1967 como un dialecto de ALGOL para simulación de eventos discretos. Fue el primer lenguaje que desarrolla el concepto de clases y subclases.
- **Smalltalk**. Desarrollado por A. C. Kay en 1971 como un sistema integral orientado a objetos que facilitaba el desarrollo de aplicaciones interactivas.
- **C++.** Creado en 1985 por B. Stroustroup añadiendo capacidades de orientación a objetos al lenguaje C. El objetivo era disponer de un lenguaje con las capacidades de abstracción y reutilización de código de la orientación a objetos y la eficiencia de C++. El lenguaje fue ganando popularidad a la par que se incluían numerosas características como genericidad, excepciones, etc.
- **Java**. Desarrollado por J. Gosling en 1993 como un lenguaje orientado a objetos para dispositivos electrónicos empotrados. Alcanzó gran popularidad en 1996 cuando Sun Microsystems hizo pública su implementación. La sintaxis del lenguaje se inspira en la de C++ pero contiene un conjunto más reducido de características. Incluye un sistema de gestión de memoria y un mecanismo de tratamiento de excepciones y concurrencia. Las implementaciones se basan en una máquina virtual estándar (denominada JVM - *Java Virtual Machine*). El lenguaje alcanza gran popularidad como lenguaje para desarrollo de aplicaciones en Internet puesto que la JVM es incorporada en muchos servidores y clientes.
- **Python**. Creado por Guido van Rossum en 1990 como un lenguaje orientado a objetos interpretado. Utiliza una sintaxis inspirada en C++ y contiene un sistema dinámico de tipos. El lenguaje ha ganado popularidad debido a su capacidad de desarrollo rápido de aplicaciones de Internet, gráficos, bases de datos, etc. [Lutz, 97]

- **C#**. Creado por Microsoft para la plataforma .NET en 1999. Utiliza una sintaxis similar a la de Java y C++. También contiene un sistema de gestión dinámica de memoria y se apoya en la máquina virtual CLR. El lenguaje aprovecha muchas de las características de esta plataforma, como el mecanismo de excepciones, sistema de componentes, control de versiones.

2.6.4. Lenguajes de programación lógica

La noción básica es la utilización de relaciones. Utiliza el algoritmo de resolución para buscar soluciones. La búsqueda de soluciones suele obtenerse mediante *backtracking*.

Algunos lenguajes representativos:

- **Prolog**. Creado en 1972 por J. Colmerauer con el propósito de facilitar el desarrollo de aplicaciones de tratamiento del lenguaje natural. Originalmente se desarrollaba como un intérprete hasta que en 1980 se crea el primer compilador eficiente mediante la máquina abstracta de Warren. Alcanzó gran popularidad a mediados de los 80 debido al proyecto japonés de la quinta generación de ordenadores, llegando a rivalizar con LISP en las aplicaciones de Inteligencia Artificial. El lenguaje no contiene sistema estático de tipos.
- **Curry**. Creado por M. Hanus en 1996 con el propósito de unificar la programación funcional y la programación lógica. El lenguaje utiliza una técnica denominada *narrowing* que generaliza la unificación de Prolog y contiene un sistema de tipos similar al de Haskell.

2.6.5. Otros paradigmas

- Programación concurrente y no determinista
- Programación dirigida por eventos
- Programación visual
- Programación mediante restricciones (Constraint programming)
- Programación orientada al aspecto

2.7. Lenguajes de Dominio Específico

En todas las ramas de la ingeniería, aparecen técnicas genéricas junto con técnicas específicas. La aplicación de técnicas genéricas proporciona soluciones generales para muchos problemas, aunque tales soluciones pueden no ser óptimas. Las técnicas específicas suelen estar optimizadas y aportan una solución mejor para un conjunto reducido de problemas.

En los lenguajes de programación también se produce esta dicotomía, lenguajes de dominio específico respecto a lenguajes de propósito general. Recientemente, hay un interés creciente en el estudio de las técnicas de implementación de lenguajes de dominio específico. Una posible definición podría ser.

Un ***Lenguaje de Dominio Específico*** es un lenguaje de programación o un lenguaje de especificación ejecutable que ofrece potencia expresiva enfocada y restringida a un dominio de problema concreto.

Entre las principales características de estos lenguajes están:

- Estos lenguajes suelen ser *pequeños*, ofreciendo un conjunto limitado de notaciones y abstracciones. En ocasiones, sin embargo, pueden contener un completo sublenguaje de propósito general. Proporcionando, además de las capacidades generales, las del dominio del problema concreto. Esta situación aparece cuando el lenguaje es empotrado en un lenguaje general.
- Suelen ser *lenguajes declarativos*, pudiendo considerarse lenguajes de especificación, además de lenguajes de programación. Muchos contienen un compilador que genera aplicaciones a partir de los programas, en cuyo caso el compilador se denomina *generador de aplicaciones*. Otros, como Yacc o ASDL, no tienen como objetivo la generación o especificación de aplicaciones completas, sino generar librerías o componentes.
- Un objetivo común de muchos de estos lenguajes es la *programación realizada por el usuario final*, que ocurre cuando son los usuarios finales los que se encargan de desarrollar sus programas. Estos usuarios no suelen tener grandes conocimientos informáticos pero pueden realizar tareas de programación en dominios concretos con un vocabulario cercano a su especialización.

Existen varias técnicas para desarrollar lenguajes de dominio específico.

- La primera posibilidad es crear un ***lenguaje específico independiente*** y procesarlo como cualquier lenguaje de programación ordinario. Esta opción requiere el diseño completo del lenguaje y la implementación de un *intérprete* o un compilador siguiendo las técnicas tradicionales.

- Otra posibilidad es ***empotrar*** el lenguaje específico en un lenguaje de propósito general. La versatilidad sintáctica y semántica de algunos lenguajes como Haskell, favorecen el empotramiento de lenguajes de dominio específico.
- Mediante ***Preprocesamiento*** o ***proceso de macros*** se empotra un lenguaje de propósito específico en otro lenguaje incluyendo una fase intermedia que convierte el lenguaje específico en el lenguaje anfitrión. Por ejemplo, el preprocesador de C++ puede utilizarse para crear un completo lenguaje específico para tareas concretas.
- Finalmente, puede desarrollarse un ***intérprete extensible*** que incluya elementos que permitan modificar el intérprete para que analice lenguajes diferentes.

2.8. Máquinas abstractas

2.8.1. Definición

Una ***máquina abstracta*** puede definirse como un procedimiento para ejecutar un conjunto de instrucciones en algún lenguaje formal.

La definición de máquina abstracta no requiere que exista implementación de dicha máquina en Hardware, en cuyo caso sería una máquina concreta.

De hecho, las máquinas de Turing son máquinas abstractas que ni siquiera pueden implementarse en Hardware.

Una ***máquina virtual*** es una máquina abstracta para la que existe un intérprete.

Varios ejemplos de máquinas virtuales:

- **SECD**. Desarrollada por P. Landin en 1966 para lenguajes funcionales. Posteriormente, la implementación de lenguajes funcionales ha recurrido a la utilización de máquinas basadas en transformación de grafos [Jones 87] como la máquina G.
- **P-CODE**. Máquina de pila desarrollada para implementar el lenguaje Pascal.
- **WAM** (Warren Abstract Machine). Desarrollada por D. H. D. Warren para el lenguaje Prolog en 1980 [Aït Kaci 91]
- **JVM**. Máquina virtual desarrollada para el lenguaje Java. Descrita en [Lindholm00]
  - **Pila de ejecución.** La JVM se basa en la utilización de una pila de ejecución y un repertorio de instrucciones que manipulan dicha pila.
  - **Código multi-hilo.** La máquina virtual puede dar soporte a varios hilos (*threads*) de ejecución concurrente. Estos hilos ejecutan código de forma independiente sobre un conjunto de valores y objetos que residen en una memoria principal compartida. La sincronización de los hilos se realiza mediante monitores,un mecanismo que permite ejecutar a un solo hilo una determinada región de código.
  - **Compilación JIT**. Un programa compilado se representa mediante un conjunto de ficheros de código de bytes (ficheros *class*) que se cargan de forma dinámica y que contienen una especificación sobre el comportamiento de una clase. La separación en módulos independientes permite la compilación a código nativo de los códigos de bytes en el momento en que se necesita ejecutar un módulo.
  - **Verificación estática de Código de bytes**. Los ficheros *class* contienen información sobre el comportamiento del módulo que puede verificarse antes de su ejecución. Es posible verificar estáticamente de que el programa no va a producir determinados errores al ejecutarse.
  - ***Gestión de memoria dinámica***. La máquina integra un recolector de basura, liberando al programador de gestionar la memoria dinámica.
  - **Dependencia del lenguaje Java**. La máquina ha sido diseñada para ejecutar programas Java, adaptándose fuertemente al modelo de objetos de Java. Incluye instrucciones de gestión de clases, interfaces, etc.Esta característica perjudica la compilación a JVM de lenguajes diferentes a Java [Gough 00]
- **CLR (Common Language Runtime)**. Entorno computacional desarrollado por Microsoft para la plataforma .NET. Utiliza un lenguaje intermedio denominado CIL (Common Intermediate Language). Ofrece algunas características similares a la máquina virtual de Java como la utilización de una pila de ejecución, código multi-hilo, compilación JIT, verificación estática y gestión dinámica de memoria. Además de las anteriores, pueden destacarse:
  - **Independencia de lenguaje**. En el diseño del CLR se ha prestado gran importancia a su utilización como plataforma de ejecución de numerosos lenguajes de programación. Aunque se adopta un determinado modelo de objetos determinado (similar al de Java), se incluyen facilidades que permiten desarrollar otros mecanismos de paso de parámetros e incluir tipos de datos primitivos en la pila de ejecución (boxing/unboxing). De hecho, la plataforma ha

sido adoptada como destino de lenguajes de diversos paradigmas como Basic, Haskell, Mercory, etc.

- **Sistema de Componentes**. Un programa .NET se forma a partir de un conjunto de ficheros de ensamblados (fichero *assembly)* que se cargan de forma dinámica. Estos ficheros contienen la especificación de un módulo que puede estar formado por varias clases. Incluyen especificación de control de versiones, firmas criptográficas, etc. Esta información puede verificarse estáticamente antes de la ejecución del programa.

Ejercicios Propuestos

1.-Ampliar el código intermedio con otros tipos básicos como `Char, Float, Array`, etc.

2.-Añadir estructuras de control más avanzadas como bucles (while, repeat, for), condicionales (if-then-else, case), etc.

3.- Ampliar el código intermedio con procedimientos mediante las instrucciones:

```
PROC p       // Define un procedimiento P
ENDPROC  // Finaliza la definición de un procedimiento
RETURN       // Finaliza la ejecución de un procedimiento
CALL p   // Llama al procedimiento P
```

Estudiar la implementación de diversas técnicas de paso de parámetros y de declaraciones de variables locales.

4.- Ampliar el código intermedio con instrucciones de creación de clases y objetos:

```
CLASS c   // Define una nueva clase c, a partir de ahí se definen los atributos y
                // métodos de la clase
ENDCLASS  // Finaliza la definición de una clase
REF   r   // Declara r como una variable de tipo referencia
NEW   c   // Crea un nuevo objeto de la clase c en la memoria.
   // Saca el elemento superior de la pila (se supone que es una referencia) y le
   // asigna la dirección de memoria del objeto creado
```

Admitir un nuevo tipo de argumentos que seleccionen atributos o métodos (formados por dos identificadores separados por punto). La ejecución de los programas comenzaría a partir del método *main* de la clase *Main*.

El siguiente programa declararía una clase *Triángulo* y una clase Main que maneja Triángulos:

```
CLASS Triangulo //class Triangulo {
INT a           //   int a, b, c;
INT b
INT c
PROC Ver        //   ver() {
OUTPUT a        //   write a,b,c;
OUTPUT b
OUTPUT c
ENDPROC

PROC Perimetro   //  perimetro() {
INT p            //  int p;
PUSHA p          //  p = a+b +c;
PUSHA a
LOAD
PUSHA b
LOAD
ADD
PUSHA c
LOAD
ADD
STORE
OUTPUT p         //  write p;
ENDPROC
ENDCLASS         // }
```

```
CLASS Main   // class Main {

PROC main   // main() {
REF t1           //  Ref t1 = new
Triangulo
PUSHA t1
NEW Triangulo
CALL t1.Ver  //   t1->ver()
INT x   // x = (t1.a + t1.b +
t1.c)/2
PUSHA x
PUSHA t1.a
LOAD
PUSHA t1.b
LOAD
ADD
PUSHA t1.c
LOAD
ADD
PUSHC 2
DIV
STORE
OUTPUT x       //  write x
ENDPROC        // }
ENDCLASS
```

5.- Estudiar la implementación de lenguajes con funciones de orden superior, es decir, lenguajes que permitan pasar funciones como argumentos, crear funciones que devuelvan otras funciones y utilizar funciones en estructuras de datos. Lenguajes de este tipo son *Haskell, ML, etc.*

6.- Caracterizar semánticamente algún lenguaje real como Java, C++, Pascal, etc.

7.- Construir una herramienta que genere automáticamente intérpretes a partir de especificaciones semánticas.

Referencias

[Abadi 96] M. Abadi, L. Cardelli, *A Theory of Objects.* Monographs in Computer Science, Springer-Verlag, ISBN: 0-387-94775-2, 1996

[Abelson 85] H. Abelson, G. J. Sussman, J. Sussman, *Structure and Interpretation of Computer Programs*. MIT Press, 1985

[Agesen93] O. Agesen and J. Palsberg and M. I. Schwartzbach, Type inference of SELF: analysis of objects with dynamic and multiple inheritance, European Conference on Object Oriented Programming, Springer-Verlag, 1993

[Aho 85] A. V. Aho, R. Sethi, J.D. Ullman*, Compilers: Principles, Techniques and Tools.* Addison-Wesley, ISBN: 0-201-10088-6, 1985

[Aït Kaci 91] H. Aït Kaci, *Warren's Abstract Machine, a tutorial reconstruction*. MIT Press, 1991

[Anderson 94]Andersen, P. H. *"Partial Evaluation applied to ray tracing"*. Res. Report, DIKU. Dept. of Computer Science, Univ. of Copenhagen

[Appel 97] A. W. Appel, *Modern Compiler Implementation in ML*. Cambridge University Press, ISBN: 0-521-58775-1, 1997

[Appel 98] A. W. Appel, *Modern Compiler Implementation in Java*. Cambridge University Press, ISBN: 0-521-58388-8, 1998

[Beshers97] C. Beshers, S. Feiner. *"Generating Efficient Virtual Worlds for Visualization Using Partial Evaluation and Dynamic Compilation"*. ACM SIGPLAN Conference on Partial Evaluation and Semantics-based Program Manipulation.

[Bracha, 98] G. Bracha, M. Ordersky, D. Stoutamire, P. Wadler. *Making the future safe for the past: Adding Genericity to the Java programming language*, Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1998

[Cardelli, 85] L. Cardelli, P. Wegner, *On Understanding Types, Data Abstraction, and Polymorphism, Computing Surveys,* vol 17, number 4, 1985

[Cardelli, 97] L. Cardelli, *Type Systems*. Handbook of Computer Science and Engineering, Ch. 103, CRC Press 1997

[Charity] The Charity Project Home Page http://www.cpsc.ucalgary.ca/Research/charity/home.html

[Cueva91] Cueva Lovelle, J. M. Lenguajes, *Gramáticas y Autómatas*. Cuaderno Didáctico nº36, Dto. de Matemáticas, Universidad de Oviedo, 1991.

[Cueva93a] Cueva Lovelle, J. M. *Análisis léxico en procesadores de lenguaje.* Cuaderno Didáctico nº48, Dto. de Matemáticas, Universidad de Oviedo, 2ª Edición, 1993.

[Cueva94a] Cueva Lovelle J. M., Mª P. A. García Fuente, B. López Pérez, Mª C. Luengo Díez, y M. Alonso Requejo. *Introducción a la programación estructurada y orientada a objetos con Pascal*. Cuaderno Didáctico nº69, Dto. de Matemáticas, Universidad de Oviedo, 1994.

[Cueva95a] Cueva Lovelle, J.M. *Conceptos básicos de Traductores, Compiladores e Intérpretes*. Cuaderno Didáctico nº9, Dpto. de Matemáticas, Universidad de Oviedo, 5ª Edición, 1995.

[Cueva95b] Cueva Lovelle, J.M. *Análisis sintáctico en Procesadores de Lenguaje*. Cuaderno Didáctico Nº 61, Dpto. de Matemáticas, Universidad de Oviedo, 2ª Edición, 1995.

[Cueva95c] Cueva Lovelle, J.M. *Análisis semántico en Procesadores de Lenguaje*. Cuaderno Didáctico Nº 62, Dpto. de Matemáticas, Universidad de Oviedo, 2ª Edición, 1995.

[Derensart, 96] P. Derensart, A. Eb-Dbali, L. Cervoni, *Prolog: The Standard, Reference Manual*. Springer Verlag, ISBN: 3-540-59304-7, 1996

[Dijkstra 68] E. W. D. Dijkstra, *Goto statement considered Harmful" Communications of the ACM, vol. 11, no. 3 (Marzo 1968), pp. 147-148*

[Ellis 94] M.A. Ellis, B. Stroustrup, *C++, Manual de Referencia con Anotaciones*. Addison-Wesley, 1994

[Espinosa 94] D. Espinosa, "Building Interpreters by Transforming Stratified Monads". Columbia University.

[Gamma 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "*Design Patterns, Elements of Reusable Object Oriented Software".* Addison-Wesley, ISBN: 0-201-63361-2, 1995

[Gossling 96] J. Gossling, B. Joy, G. Steele, *The Java Language Specification*. Addison-Wesley. ISBN 0-201-63451-1, 1996

[Gough 00] K. J. Gough, D. Corney, *Evaluating the Java Virtual Machine as a target for languages other than Java*, Joint Modula Languages Conference, JMLC 2000, Zurich, Sep. 2000

[Hanson, 01] D. R. Hanson, T. A. Proebsting, *Dynamic Variables*, ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001

[Holmes 95] J. Holmes, *Object Oriented Compiler Construction*. Prentice-Hall Intl. ISBN: 0-13-192071-5, 1995

[Hudak 98] P. Hudak, *Domain Specific Languages,* Handbook of Programming Languages, vol. III, Little Languages and Tools, Macmillan Technical Pub., 1998

[Jones 93] Neil D. Jones, *"Partial Evaluation and Automatic Program Generation",* Prentice-Hall, ISBN: 0-13-020249-5

[Jones 96] Neil D. Jones, *"An Introduction to Partial Evaluation"*. ACM Computing Surveys, Vol 28, Nº3, Sep. 96

[Jones 87] S. P. Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987

[Kernighan 91] B.W. Kernighan, D. M. Ritchie, *El lenguaje de Programación C*, 2nd Ed. Prentice Hall, ISBN 968-880-205-0, 1991

[Kennedy, 01] A. Kennedy, D. Syme, *Design and Implementation of Generics for .NET Common Language Runtime*,ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001

[Kirkerud, 97] Björn Kirkerud, Programming Language Semantics. Imperative and Object Oriented Languages. Intl. Thomson Computer Press, 1997

[Labra 95] Jose E. Labra G. *"Análisis y Desarrollo de un Sistema Prolog",* Cuaderno Didáctico 85. Depto de Matemáticas. Universidad de Oviedo. Dic. 95

[Lewis, 00] J. R. Lewis, M. B. Shields, E. Meijer, J. Launchbury, *Implicit Parameters: dynamic scope with static types*, ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, 2000

[Liang 96] S. Liang, P. Hudak, *"Modular Denotational Semantics for Compiler Construction"*. European Symposium of Programming

[Lindholm 00] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification, 2nd. Ed.*

[Lutz, 97] M. Lutz, *Python, an Object Oriented Scripting Language*, Handbook of Programming Languages, Little Languages and Tools, McMillan Technical Publishing, Editor. P. H. Salus, 1997

[Marlow 01] S. Marlow, S. Peyton Jones, A. Moran, J. Reppy, *Asynchronous Exceptions in Haskell,* ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001

[Meijer 01] J. Smith, N. Perry, E. meijer, *Mondrian for .NET*, Dr. Dobbs Journal, 2001

[Mitchell, 96] J. C. Mitchell, *Foundations for Programming Languages*. The MIT Press. ISBN: 0-262-13321-0, 1996

[Morris78] C. Morris, *Writings on the general theory of signs*, Mouton and Co.,1971

[Ousterhout97] John K. Ousterhout, *"Scripting: Higuer Level Programming for the 21st Century"* Sun Microsystems Laboratories.

[Pagan 91] Frank G. Pagan, "Partial Computation and the Construction of Language Processors". Prentice Hall, ISBN 0-13-6511415-4

[Plezbert96] Michael Plezbert, Ron K. Cytron, *"Continuous Compilation for Software Develpment and Mobile Computing".* Master Thesis, Washington University.

[Plezbert97] Michael Plezbert, Ron K. Cytron, *Does "Just in Time" = "Better Late than Never"?.* 24th Annual SIGPLAN-SIGACT Symposium of Principles of Programming Languages, Paris, 1997

[Proebsting 95] Todd A. Proebsting, "Optimizing an ANSI C Interpreter with Superoperators," POPL'95, January 1995, pages 322—332

[Roberts 94] G. Roberts, "Experiences with Building Incremental Compilers using Objects". OOPSLA'94

[Schmidt 95] D.A. Schmidt, *Programming Language Semantics*. ACM Computing Surveys, 95

[Steele 94] G. L. Steele Jr. *"Building Interpreters by Composing Monads"*. Proceedings of the 21st Symposium on Principles of Programming Languages.

[Tennent 94] R. D. Tennent, *"Denotational Semantics"*. Handbook of Logic in Computer Science. Vol. 3 Semantic Structures. S. Abramsky, D. M. Gabbay, T. S. Maibaum (Ed.) Oxford Science Publications

[Viroli, 00] M. Viroli, A. Natali, *Parametric Polymorphism in Java: an approach to translation based on reflective features.* ACM Conference on Object-Oriented Programming Systems, Languages and Applications, 2000

[Wansbrough 97] K. Wansbrough, *A modular Monadic Action Semantics*. Masters Thesis, Univ. of Auckland, Feb. 1997