UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

U. A. C. P. y P. DEL C. C. H.
I. I. M. A. S.

LEPPOOC:

"Un Lenguaje de Programación Persistente Orientado a Objetos con Objetos Compuestos"

T E S I S

Que para obtener el Grado de
MAESTRA EN CIENCIAS DE LA COMPUTACION
p r e s e n t a

ALMA ROSA GARCIA GAONA

México, D. F., Febrero 1996

TESIS CON
FALLA DE ORIGEN

TESIS

COMPLETA

AGRADECIMIENTO

♠ Al Laboratorio Nacional de Informática Avanzada **(LANIA)**:

Por haber sido, una vez más, el medio preculsor para la superación de las personas, la ciencia y la educación... Agradeciendo sinceramente todo el apoyo y las facilidades brindadas para el logro de esta meta.

♠ A mi Director de Tésis

Dr. Vladimir Estivill Castro:

Por reconocer en él: la disponibilidad de tiempo, la compartición de conocimientos, el valor de la experiencia, la certeza en la orientación... verdadero significado de " Maestro ".

♠ A mis Sinodales:

Dra. Hanna Oktaba

M.C. Guadalupe Ibargüengoitia

M.C. Elisa Viso

M.I. María Luz Gasca

Por sus valiosos comentarios, sugerencias y el tiempo dedicado a la revisión de este trabajo.

DEDICATORIA

♠ A la memoria de mi Madre (q.e.p.d.).

♠ A mi padre:

Por la dicha de tenerlo conmigo.

♠ A mi esposo e hija:

Por su comprensión y amor incondicional, presente en los sacrificios y venturas compartidos en todo momento.

♠ A todos aquellos que siempre han estado cerca de mi.

INDICE

INTRODUCCION

Este trabajo tiene como objetivo, mostrar los conceptos del modelo orientado a objetos, desde la perspectiva de bases de datos e introduciendo la manipulación de objetos compuestos y persistencia, a través del diseño e implantación de un prototipo de lenguaje de programación orientado a objetos, con la ventaja de una sintaxis sencilla y funcional al estilo de LISP, mismo que se le ha denominado LEPPOOC.

Las metas planteadas para alcanzar el objetivo de este trabajo son:

1) Elaborar un prototipo que maneje los conceptos del modelo orientado a objetos en bases de datos como son:

   - Herencia
   - Atributos y Métodos
   - Objetos compuestos
   - Clases
   - Encapsulación
   - Extensibilidad
   - Persistencia
   - Identidad
   - Integridad computacional

2) El lenguaje deber tener las siguientes características de programación funcional:

- Asignación dinámica de tipos.
- Lenguaje interpretado.
- Sintáxis funcional al estilo de LISP y SCHEME.

3) Manejar objetos compuestos, estableciendo diferentes niveles de la relación de pertenencia.

4) Discutir los mecanismos de manipulación de objetos compuestos.

El trabajo se encuentra estructurado en seis capítulos, donde el primero tratará los conceptos del modelo orientado a objetos en bases de datos, en el segundo se explicarán los objetos compuestos, los niveles de la relación de pertenencia, y los mecanismos de manipulación de objetos compuestos. El tercer capítulo tratará de la sintaxis de LEPPOOC, en el capítulo cuarto se hablarán de los principios de diseño de lenguajes de programación, mencionando los que se aplicaron en LEPPOOC, en el quinto se explicará su semántica y en el sexto y último capítulo se mencionará la estructura del intérprete LEPPOOC.

1 MODELO ORIENTADO A OBJETOS EN BASES DE DATOS

Actualmente los Sistemas de Bases de Datos Orientados a Objetos (SBDOO) están recibiendo una gran atención, desde los puntos de vista experimental y teórico, y existe un gran debate acerca de la definición de dichos sistemas, esto debido a la falta de un modelo de datos común y una fundamentación formal. Realmente se pueden encontrar muchas propuestas de un modelo de datos orientado a objetos, pero no existe un claro consenso sobre una sola.

El grupo de desarrolladores de Bases de Datos Orientadas a Objetos (BDOO) y algunos centros de investigación y desarrollo, han intentado definir las características que las distinguen, y han acordado que es importante agruparlas en tres categorías[1]:

- **Obligatorias**, las que el sistema debe satisfacer como mínimo.
- **Opcionales**, las que se pueden agregar para mejorar el sistema.
- **Abiertas**, las que el diseñador puede seleccionar de un número de soluciones igualmente aceptables.

Estas características están descritas en el *"The Object Oriented Database System Manifesto"*. En este trabajo se utilizarán para el marco de referencia de los propósitos del mismo. A continuación se revisan tales conceptos.

1.1 CARACTERISTICAS OBLIGATORIAS

Un sistema de base de datos orientado a objetos debe satisfacer dos criterios: Ser un Sistema Manejador de Base de Datos y un Sistema Orientado a Objetos, es decir debe ser consistente con el conjunto de lenguajes de programación orientados a objetos tanto como sea posible.

El primer criterio se traduce en cinco características: ***persistencia, manejo de almacenamiento secundario, concurrencia, recuperación y facilidad de consulta.*** El segundo criterio se traduce en ocho características: ***Objetos Compuestos, Identidad de Objetos, Encapsulación, Tipos o Clases, Herencia*** combinada con ***Atadura Tardía, Extensibilidad*** e ***Integridad Computacional.***

El propósito de LEPPOOC es concentrarse en características de orientación a objetos e incorporar persistencia de las áreas de bases de datos. Por esto, los aspectos de bases de datos como manejo de almacenamiento secundario, concurrencia, recuperación y facilidad de consulta no serán el foco de estudio e incluso quedan fuera del alcance del prototipo que se desarrolló.

Sin embargo, se considera que el modelo de datos de LEPPOOC, su sintaxis y semántica son aspectos valiosos que se han logrado implantar y que constituyen la base para versiones futuras, que incorporen más conceptos de base de datos.

Los elementos en que se ha concentrado el diseño de LEPPOOC son: *objetos compuestos, identidad, encapsulación, clases, herencia, traslape, sobrecarga, asignación tardía, extensibilidad, integridad computacional y persistencia.*

A continuación se revisará la definición de estos conceptos.

1.1.1 Objetos Compuestos

Un OBJETO[4] es una entidad que tiene ***estado, identidad*** y ***comportamiento***.

◊ El estado de un objeto se compone de todas las propiedades del objeto (generalmente estáticas), más los valores actuales (generalmente dinámicos) de cada una de estas propiedades. Es decir, el estado corresponde a la asignación de valores de los atributos de un objeto. Cuando los valores de los atributos de un objeto son de tipo primitivo, el objeto es simple, en cambio los ***objetos compuestos*** son un conjunto heterogéneo de objetos que forman una jerarquía de partes.

Los ***objetos compuestos*** se construyen aplicando constructores a objetos simples. Los objetos simples son los enteros, caracteres, variables, cadenas, lógicos y números reales. El conjunto mínimo de constructores que un sistema debe proveer incluye los conjuntos, listas y tuplas.

Se deben proporcionar los operadores apropiados para manejar los ***objetos compuestos*** y todo lo que sea su composición. Esto significa, que las operaciones sobre un objeto complejo deben propagarse transitivamente a todas sus componentes. Operaciones como la recuperación o destrucción de un objeto complejo completo, o la propagación de una copia a profundidad, en contraste con una copia superficial, donde las componentes no son replicadas, sino que son referenciadas sólamente por la copia del objeto raíz. Las operaciones adicionales sobre objetos complejos las pueden definir los usuarios del sistema a través de la extensibilidad. LEPPOOC maneja un modelo de objetos compuestos que se discutirá más adelante.

◊ La identidad de un objetoes la propiedad que lo distingue de los demás.

◊ El comportamiento es la forma como actúa un objeto, en términos de sus cambios de estado. Es decir, el comportamiento de un objeto está completamente definido por sus acciones.

1.1.2 Identidad

La identidad del objeto ha existido desde hace algún tiempo en los lenguajes de programación, el concepto es más reciente en bases de datos. La idea estriba en que en un Modelo con Identidad de Objetos, un objeto tiene una existencia que es independiente de sus valores.

El modelo de datos soportará la identidad en la medida en que un objeto exista independientemente de sus valores (o estado) y por lo tanto, su existencia es independiente de otros objetos que pudiesen estar en el mismo estado y de los cambios de estado del objeto mismo.

Dos nombres de objetos pueden ser ***idénticos*** (ser ***alias*** del objeto), si tienen el mismo identificador (o nombres para una misma representación del objeto).

Dos nombres de objetos pueden ser ***iguales*** si son nombres de objetos que tienen el mismo valor en todos sus atributos. Los nombres de objetos idénticos pueden distinguirse de los nombres de los objetos iguales en la compartición y actualización de objetos.

- **Compartición de Objetos.** En un modelo basado en identidad, dos objetos pueden compartir una componente, por lo que entonces, la representación de un objeto compuesto es una gráfica, mientras que en un sistema sin identidad, se limita a un árbol. Para aclarar este aspecto considere el

siguiente ejemplo: Suponga que se tiene un objeto PERSONA, que tiene como atributos, el ***nombre***, ***edad***, y ***nom-de-los-hijos***. Suponga que JOSE y ALICIA tienen un hijo llamado RUBEN de 14 años. En la realidad pueden ocurrir dos situaciones:

1) Que JOSE y ALICIA pueden ser padres del mismo joven llamado RUBEN
2) Que sean dos muchachos distintos llamados RUBEN.

En un sistema sin identidad, JOSE se representaría como:

(jose, 50, {(ruben, 14, {})})

y ALICIA se representaría por:

(alicia, 45, {(ruben, 14, {})})

De esta forma no habría manera de expresar si JOSE y ALICIA son los padres del mismo muchacho.

Sin embargo, en un modelo basado en la identidad, estas dos estructuras pueden o no compartir la parte común (ruben, 14, {}) y así distinguir entre las dos posibles situaciones.

- **Actualización de Objetos.** En un sistema basado en la identidad todas las actualizaciones referentes a un objeto se propagan a sus componentes, mientras que en un sistema basado en el valor, los objetos y sus componentes se deben actualizar separadamente. Retomaremos el ejemplo anterior para explicar este aspecto. Suponga que JOSE y ALICIA son verdaderamente los padres de un joven llamado RUBEN.

  En un sistema basado en la identidad, todas las actualizaciones referentes al hijo de ALICIA serán aplicadas al objeto RUBEN, y consecuentemente también al hijo de JOSE.

En un sistema basado en el valor, por contraste, los dos subobjetos deben ser actualizados de manera independiente.

El soporte de identidad de objetos implica ofrecer operaciones tales como: *Asignación de objetos, copia de objetos, pruebas de igualdad de objetos.* Uno puede simular identidad de objetos en un sistema basado en el valor, introduciendo identificadores de objetos explícitos, sin embargo esto le deja la carga al usuario para asegurar la unicidad de identificadores de objetos y para mantener la integridad referencial, siendo una carga delicada, ya que el sistema debe asegurar que el identificador no sea modificado. La mayoría de los sistemas no permiten que el usuario accese directamente el identificador. LEPPOOC tiene asignación de objetos, copia de objetos, e identidad de objetos basado en el valor, el sistema asegura que el identificador no sea modificado por el usuario.

Los modelos basados en la identidad, son la norma en los lenguajes de programación imperativos, donde cada objeto manipulado en un programa tiene una identidad y puede ser actualizado; tal identidad viene ya sea del nombre de una variable o de una localidad física en memoria, pero desde la perspectiva de la gente de bases de datos relacionales puros, donde las relaciones están basadas en el valor, el concepto es completamente nuevo.

Resulta interesante analizar los diferentes enfoques usados en los sistemas actuales para construir identificadores. En el sistema ORION[10], un identificador consiste de la pareja <*identificador de clase, identificador de objeto* > donde el primero es el identificador de la clase a la cual pertenece el objeto y el segundo identifica el objeto dentro de la clase, es decir el elemento de la clase. Cuando se invoca una operación sobre un objeto, el sistema extrae el identificador de la clase desde el identificador y determina el método para

ejecutar la operación. Este enfoque tiene la desventaja de realizar la migración de un objeto desde una clase a otra por demás difícil. Esto es debido a que involucra la modificación de todos los identificadores de los objetos migrados. En tales situaciones, toda referencia a objetos migrados es invalidada.

En otro enfoque, usado por ejemplo en SMALLTALK[10], y en el sistema IRIS, el identificador de la clase a la que pertenece el objeto se almacena generalmente como información de control en el mismo objeto. Para ejecular una operación sobre un objeto, éste tiene que ser accesado, de modo que el identificador pueda ser extraído desde él. En el caso de operaciones inválidas, la verificación de tipos es costosa, ya que realiza un acceso innecesario a disco.

Ciertos sistemas, como GEMSTONE y O_2, permiten al usuario asignar nombres de variables (nombres de usuario) a objetos. Estos nombres, en el caso de GEMSTONE, son almacenados en un diccionario de símbolos. De esta forma diferentes usuarios pueden tener diccionarios distintos. Los nombres le permiten al usuario accesar directamente un objeto dado, desde la base de datos.

Ciertos modelos de datos también soportan un tercer tipo de igualdad que se denomina con frecuencia igualdad superficial (SHALLOW), donde dos objetos son iguales-superficialmente, aunque no sean idénticos, si todos sus atributos comparten los mismos valores y las mismas referencias.

1.1.3 Encapsulación

La encapsulación no se introdujo inicialmente en los lenguajes de programación orientados a objetos, sino que es el resultado final de un proceso

de evolución que empezó con los lenguajes imperativos. La razón fue por la necesidad de hacer una clara distinción entre la especificación y la implantación de una operación, y la necesidad de la modularidad. La encapsulación en los lenguajes de programación se deriva de la noción de tipos de datos abstractos. Een este contexto un objeto consiste de una ***interfaz*** y una ***implantación***. La interfaz es la especificación del conjunto de operaciones que se pueden invocar sobre el objeto. La interfaz es visible sólo a los usuarios del tipo de datos abstracto. La implantación contiene los datos, es decir, la representación del estado del objeto y los métodos que proporciona. En cualquier lenguaje de programación la sección de implantación instrumenta toda operación que proporciona el objeto.

Tradicionalmente, en lenguajes de programación este principio se traduce en la noción de que un objeto concentra las operaciones y los datos, pero con la diferencia que en las bases de datos, en general, no es claro si la estructura es parte de la interfaz o no, mientras que en los lenguajes de programación la estructura de datos es claramente parte de la implantación y no es visible.

La encapsulación proporciona una forma de *"independencia de datos lógica"*, lo cual significa que podemos cambiar la implantación de un tipo sin cambiar los programas que usan el tipo. Con lo cual los programas de aplicación vienen a estar protegidos de los cambios en la implantación en las capas más bajas del sistema.

El manifiesto de ATKINSON, DEWITT Y MAIER[1], hace una observación interesante de que la encapsulación real se obtiene solamente cuando las operaciones son visibles y el resto del objeto (datos y la implantación de las operaciones) se oculta. Sin embargo, existen casos en los que la encapsulación no es necesaria. No obstante, el uso del sistema puede

simplificarse significativamente si los sistemas permiten que la encapsulación sea violada bajo ciertas condiciones.

1.1.3.1 Métodos

Los objetos en los Sistemas Manejadores de Bases de Datos Orientados a Objetos (SMBDOO), son manipulados vía métodos. En general la definición de un método consiste de dos componentes: FIRMA o SIGNATURA (del inglés "signature") y CUERPO (del inglés "body"). La FIRMA o SIGNATURA especifica el nombre del método, los nombres y clases de los argumentos y la clase del resultado, si el método regresare alguno. Por lo tanto, la FIRMA o SIGNATURA es la especificación de la operación a implantarse por el método.

En O_2 la firma o signatura sería el PROTOCOLO, y el cuerpo sería el METODO. Algunos sistemas, tales como ORION, no requieren especificar la clase del argumento. En este sistema la verificación de tipos se realiza al tiempo de ejecución. Aún en lenguajes de programación como SMALLTALK, esta especificación no se requiere. En CLOS se utiliza un enfoque intermedio, ya que la especificación da la clase del argumento así como de la clase del dominio de los atributos del objeto, son opcionales.

El CUERPO representa la implantación del método y consiste de un conjunto de instrucciones expresadas en cualquier lenguaje de programación. Varios SMBDOO usan lenguajes diferentes; ORION usa LISP, GEMSTONE usa una extensión de SMALLTALK, O_2 usa C, VBASE/ONTOS, VERSANT y OBJECTSTORE usan C++.

1.1.3.2 Acceso y Manipulación de Atributos

Algunos SMBDOO permiten que los valores de los atributos de los objetos sean leídos y escritos directamente, violando así el principio de la encapsulación. El objetivo es hacer menos complejo el desarrollo de aplicaciones que simplemente accesen o modifiquen atributos de objetos. Obviamente estas aplicaciones son muy frecuentes en la manipulación de datos. De esto se deducen dos ventajas:

- Evitarle al programador tener que desarrollar un gran número de métodos, generalmente convencionales.
- Incrementar la eficiencia de las aplicaciones, ya que el acceso directo a los atributos de los objetos se implanta como operaciones proporcionadas por el sistema.

Obviamente, la violación del principio de encapsulación puede causar problemas, debido a que la definición de los atributos de un objeto puede ser modificado. Por eso los SMBDOO proporcionan diferentes soluciones. VBASE/ONTOS[6], proporciona métodos "definidos-por-el-sistema" para leer y escribir los atributos de un objeto. Sin embargo estos métodos pueden ser redefinidos por el usuario. O_2 permite al usuario establecer que atributos y métodos son visibles en la interfaz del objeto y cuales pueden ser invocados desde afuera.

Estos atributos y métodos se dice que son públicos. Los atributos y métodos que no son visibles desde afuera se dice que son privados. Un enfoque similar se da en C++. En otros sistemas como ORION, todos los atributos pueden accesarse directamente mientras se leen y escriben, y todos los métodos pueden invocarse. En ORION los mecanismos de autorización pueden usarse para prevenir el acceso a ciertos atributos y la ejecución de ciertos métodos.

LEEPOOC maneja los atributos de un objeto como privados al exterior, pero públicos a sus subclases. Cualquier otro modo de operación será modelable en LEPPOOC vía métodos. Así mismo manejará nombres de atributos y métodos, restricciones de integridad semántica elementales así como uniformidad en el acceso a atributos (no como C++).

1.1.4 Tipos y Clases

Existen dos categorías principales de sistemas oriantados a objetos, aquellos qua soportan la noción de CLASES y los que soportan la noción de TIPOS. Exista mucha discusión sobre la distinción entre clases y tipos, y la falta de una definición formal viene a crear problemas. Sin embargo, aunqua no exista una clara línea de la diferancia entre ambos conceptos, son fundamantalmente diferentes.

Los TIPOS son utilizados en la clasificación, organización, abstracción, conceptualización y transformación de colecciones da valores. Existen diversas varsiones de la dafinición de tipo. Cada una tiene un grupo de adeptos para el cual su visión es la principal. En particular para esta trabajo no se entrará en detallas de cada visión, sólo se resaltará que el término TIPO está excesivamante sobrecargado. Se tomará la idea de que TIPO se usa para capturar lo que necesita el programador de un compilador para escribir la estructura da expresiones, y el término CLASE para capturar lo qua necesita el programador da la aplicación para describir al comportamiento de valores.

Una vez ubicados en una idea clara de la diferencia de tipos y clases, se abundará en ellas.

- Una CLASE, es un conjunto de objetos que comparten una estructura y comportamiento común.

- Un TIPO, en un sistema orientado a objetos, resume las características comunes de un conjunto de objetos.Corresponde a la noción de tipos de datos abstractos, tiene dos partes, *la interfaz y la implantación.* Solamente la interfaz es visible a los usuarios del tipo, la implantación del objeto es vista solamente por el diseñador del tipo.

La ***implantación*** del tipo consiste de una parte de datos y una de operaciones. En la parte de datos, se describe la estructura interna de los datos del objeto, dependiendo del poder del sistema, la estructura de esta parte puede ser más o menos compleja. La parte de ***operación*** consiste de procedimientos que implantan las operaciones enunciadas en la interfaz.

En lenguajes de programación, los tipos son herramientas que incrementan la productividad del programador, asegurando la corrección del programa. Existen lenguajes de programación que usan esta herramienta y otros que no. En los que sí, por fuerza el usuario debe declarar los tipos de las variables y expresiones que manipula y si el sistema de tipos está bien diseñado, entonces puede hacer la verificación de tipos al tiempo de compilación. De otro modo, algunos de ellos tienen que diferirse al tiempo de ejecución. En general, en los sistemas basados en tipos, un tipo "*no es un ciudadano de primera clase*", ya que tiene un estado especial y no puede ser modificado al tiempo de ejecución.

La noción de CLASE es diferente de la de tipo, aunque su especificación es la misma, pero es más que una noción al tiempo de ejecución. Tiene dos aspectos: Una ***fábrica de objetos*** y un ***almacén de objetos***. La fábrica de objetos puede usarse para crear nuevos objetos, ejecutando para ello una operación NEW sobre la clase, o reproduciendo algún objeto prototipo representativo de la clase.

El almacén de objetos permite que la clase se conecte a sus extensiones (conjunto de objetos que son instancias de la clase). El usuario puede manipular el almacén aplicando operaciones en todos los elementos de la clase.

Las clases no son usadas para verificar la corrección de un programa, sino para la creación y manipulación de objetos. En la mayoría de los sistemas que emplean el mecanismo de clase, las clases ***"son ciudadanos de primera clase"***, ya que pueden ser manipuladas al tiempo de ejecución, es decir actualizados o pasados como parámetros.

Teóricamente sería correcto usar los tres conceptos siguientes de la extensión de tipos en el modelo de base de datos orientado a objetos:

- **El tipo**, significa la especificación de un conjunto de objetos.
- **La clase**, significa la estructura e implantación de un conjunto de objetos.
- **La colección de objetos**, soporta el concepto de la extensión de un tipo.

Pero las implicaciones para otros aspectos del modelo y para la implantación serían serias. Por ejemplo, se requerirían tres mecanismos de jerarquías diferentes para la herencia. Más aún, cada objeto necesitaría tener su tipo y clase asociados a él.

Por otro lado, la instanciación significa que la misma definición puede usarse para generar objetos con la misma estructura y comportamiento, en otras palabras, es un mecanismo por el cual las definiciones pueden ser reutilizadas. Los modelos de datos Orientados a Objetos, usan el concepto de clase como

una base para la instanciación. En este sentido, una clase es un objeto que actúa como un patrón o modelo. En particular especifica:

- ***Una estructura***, es decir, el conjunto de atributos de las instancias.
- ***Un conjunto de operaciones,*** es decir las que manipulan los datos.
- ***Un conjunto de métodos***, que implanta las operaciones.

Los objetos que responden a todas las operaciones definidas para una clase dada, pueden genererse usando la operación equivalente NEW. Claramente, los valores de los atributos de cada objeto puede almacenarse seperadamente, pero la definición de las operaciones y de los métodos no tienen que repetirse. De hecho cada clase esta asociade con un objeto conocido como un ***objeto-clase*** (Meteclase), que contiene la información común a las instancias de le clase, y en particular, los métodos de la clase, por lo tanto el ***objeto-clase*** se almacene separado de les instencias.

Un enfoque elternetivo pera generar objetos es usar ***objetos prototipos***. Esto involucra le genereción de un objeto nuevo a partir de un objeto existente, modificendo sus etributos y su comportemiento. Este es un enfoque útil cuando los objetos cambian rápidemente y son más diferentes que similares.

En generei, un enfoque basedo en la ***instanciación***, es más apropiado donde los ambientes de la epliceción en uso están más esteblecidos, ya que hace difícil experimentar en estructuras de objetos alternativas, mientres que un enfoque basado en ***prototipos*** es más ventejoso, donde la experimentación se hace en las etapes de diseño inicial de epilcaciones, o en embientes que cambian rápidamente y donde existen pocos objetos establecidos.

Por supuesto, existen fuertes similitudes entre ***clases*** y ***tipos***. Los nombres se usan con ambos significados y las diferencias pueden ser sutiles en algunos

sistemas, por lo que la elección entre los dos enfoques la debe hacer el diseñador del sistema. Para este caso LEPPOOC usará ***clases***, pero será muy débil en su sistema de tipos.

1.1.5 Herencia

La herencia es el segundo mecanismo de reutilización y el concepto más poderoso de la programación orientada a objetos. Con la herencia, una clase llamada ***subclase*** puede ser definida en la base de la definición de otra clase llamada ***superclase***. Las subclases heredan los atributos, métodos y mensajes de sus superclases. Además, una subclase puede tener sus propios atributos, métodos y mensajes que no son heredados.

La herencia tiene dos ventajas: es una herramienta poderosa para modelar, porque da una descripción precisa y concisa del mundo; y es una ayuda en el diseño de especificaciones e implantaciones compartidas en las aplicaciones.

La herencia, además, es una ayuda de la reutilización del código, porque cada programa se encuentra en el nivel donde un mayor número de objetos pueden compartirlo.

Con el enfoque de la herencia, se escribe menos código. Existe la ventaja de agregar, ya que la herencia soporta una descripción más precisa y concisa de la realidad que se quiere modelar.

En ciertos sistemas, una clase puede tener varias superclases, en tales casos uno habla de ***herencia múltiple***, mientras que otros imponen la restricción de una sola superclase, ***herencia simple***. La posibilidad de definir una clase en

término de otras clases simplifica la tarea de definición de clases. Sin embargo, pueden surgir conflictos, especialmente en la herencia múltiple.

Generalmente, cuando el nombre de un atributo o método definido explícitamente en una clase es el mismo que se definió en una superclase, el atributo de la superclase no es heredado, sino que se cubre por la nueva definición. En este caso se habla de TRASLAPE.

Aunque la relación ***clase-subclase*** define la jerarquía de herencia más desarrollada en la literatura, existen diferentes variantes de ***herencia*** conforme a los enfoques de los diferentes lenguajes de programación orientados a objetos. La diferencia de estos conceptos depende de aspectos como el significado de las ***clases*** y de los ***tipos***. En la literatura[3] se pueden encontrar algunas definiciones de jerarquías análogas a la jerarquía de herencia que se ha discutido, tal es el caso de:

- La jerarquía de especificación,
- La jerarquía de implantación,
- La jerarquía de clasificación.

Cada jerarquía relaciona ciertas propiedades de los ***sistemas de tipos*** y de los ***sistemas de clases***. Sin embargo, estas propiedades se combinan en un simple mecanismo de herencia.
Una discusión similar se hace en ATKINSON, DEWITT Y MAIER [1], donde introducen los conceptos de herencia por sustitución e inclusión y hace la siguiente clasificación manejando cuatro clases de herencia:

1) Herencia de Sustitución
2) Herencia de Inclusión
3) Herencia de Restricción
4) Herencia de Especialización

En cierto grado, estos cuatro tipos de herencia se proporcionan en los sistemas y prototipos existentes, y no prescriben un estilo específico de herencia.

LEPPOOC maneja clases y subclases, los objetos de una subclase heredarán el protocolo y los atributos de su superclase, pero podrán redefinirlos. LEPPOOC se concentra en el concepto de herencia más usado en orientación a objetos, es decir ***Herencia Simple*** y jerarquía ***Clase/Subclase***.

Un problema de considerable importancia se refiere a que si la estructura de los elementos de una clase deben también encapsularse, con respecto a las subclases. De hecho, los métodos de una clase pueden accesar directamente todos los atributos de sus ejemplares. Sin embargo, donde se aplica la herencia, el conjunto de atributos de los elementos de una clase consisten de la unión de los atributos heredados y los atributos especificados de la clase.

Por lo tanto, la implantación de un método es dependiente, de los atributos que están siendo definidos, no en la clase en la que el método está definido sino en cualquier superclase. Una modificación a la estructura de los elementos de cualquier superclase puede invalidar un método definido en cualquier subclase. Esto sucede con LEPPOOC.

Esto limita el beneficio de la encapsulación en cuanto a los efectos de las modificaciones a una clase que no están limitados a la misma clase. Se han propuesto soluciones, limitando la visibilidad de los atributos con respecto a las subclases. LEPPOOC no limita la visibilidad.

1.1.6 Traslape, Sobrecarga y Asignación Tardía.

En esta sección se mostrarán varias ideas de importancia que un buen lenguaje de programación debería contar.

El concepto de ***polimorfismo*** es ortogonal al concepto de herencia. El polimorfismo en los lenguajes de programación se genera al admitir que una expresión o valor tome más de un tipo, lo que permite definir funciones generales que son aplicables a diferentes tipos de objetos.

En los lenguajes polimórficos es posible declarar una sola función que realice las mismas operaciones definidas en ella para diferentes tipos de datos, de esta forma logramos evitar tener código repetitivo.

El polimorfismo es, sin duda, una forma de facilitar la tarea de programar y no es algo tan novedoso. De hecho, algunos lenguajes monomórficos ya ofrecían facilidades que se pueden considerar como formas restringidas de polimorfismo. Concretamente, se puede hacer referencia a dos: ***sobrecarga*** y ***coerción***.

- **SOBRECARGA:** Es el caso en el que un símbolo o valor toma diferentes tipos que pueden o no estar relacionados entre sí. Se puede considerar que un mismo símbolo está representando a diferentes funciones o valores (ejemplo el operador "+"). Sin embargo hay una distinción entre sobrecarga y polimorfismo, la ***sobrecarga*** es el mismo nombre para dos funciones, mientras que polimorfismo es la misma función para diferentes tipos.

- **COERCION:** Es la facilidad que ofrecen algunos lenguajes para manipular datos con cierta flexibilidad en cuanto a la compatibilidad de sus tipos, el lenguaje C permite efectuar operaciones entre objetos de diferentes tipos, como sumar enteros y reales, o caracteres y enteros, sin que el programador se vea en la necesidad de hacer explícitamente la transformación de tipos que se requerirá en otros lenguajes.

De esta discusión se desprenden las siguientes ideas.

Cuando se desea realizar una misma operación para diferentes objetos, cuyos miembros son de tipos que no se conocen al tiempo de compilación se habla de ***polimorfismo***. En una aplicación usando un sistema convencional, se tendría una operación para cada tipo de objeto. Esto obliga al programador a estar enterado de todos los tipos posibles de objetos en el conjunto, a estar consciente de las operaciones correspondientes, y a usarlas correctamente.

En un sistema orientado a objetos, se define la operación en el nivel del tipo del objeto (el tipo más general en el sistema), así esta operación solo tendrá un nombre y puede ser utilizada indiferentemente sobre cualquier tipo de objeto. Sin embargo, se redefine la implantación de la operación para cada uno de los tipos conforme al tipo; a tal redefinición se le denomina ***traslape***, mismo que resulta en un sólo nombre para programas diferentes, a esto se le llama ***sobrecarga dinámica***. De esta manera, se aplica tal operación y el sistema se encarga de seleccionar la implantación apropiada al tiempo de ejecución. Aquí se obtiene una ventaja más, los implantadores del tipo aún escriben el mismo número de programas, pero los programadores de aplicación no tienen que preocuparse acerca de los programas diferentes. El código se vuelve más simple y mantenible. Cuando un nuevo tipo se introduce y se agregan nuevas instancias, el programa seguirá trabajando sin modificación (siempre que traslape el método adecuado para ese nuevo tipo).

Para proporcionar esta nueva funcionalidad, el sistema no puede atar nombres de operaciones a programas al tiempo de compilación, por lo tanto, los nombres de las operaciones deberán ser resueltos al tiempo de ejecución. Esta traducción retardada se le llama ***asignación tardía***[A]. Esta hace la verificación

de tipos más difícil (y en algunos casos imposible), pero no la impide completamente.

El polimorfismo ha demostrado su utilidad en la programación a gran escala, por lo que hace deseable contar con un lenguaje que lo soporte de manera eficiente y flexible.

1.1.7 Extensibilidad

Los sistemas de bases de datos vienen con un conjunto de tipos predefinidos, estos son extensibles. Esto significa que deben ser un medio para definir nuevos tipos y no debe existir distinción en el uso entre los tipos definidos por el sistema y los definidos por el usuario. Por supuesto, puede existir una fuerte diferencia en la forma en que los tipos definidos por el sistema y los definidos por el usuario son soportados por el sistema, pero esto es invisible tanto a la aplicación como a los programadores de aplicación.

La definición de un tipo incluye la definición de operaciones del tipo. Los requerimientos de encapsulación implican que exista un mecanismo para definir nuevos tipos, los requerimientos de extensibilidad fortalecen la capacidad de requerir tipos creados nuevamente para tener el mismo estado que uno existente. Sin embargo, no se requiere que la colección de constructores de tipos (tuplas, conjuntos, entre otras) sea extensible.

Los tipos en LEPPOOC son diferentes a las clases definidas por el usuario.

[A] Algunas autores traducen late binding como ligado tardío.

Los tipos en LEPPOOC son muy pocos (números, cadenas y listas), pero los tipos de un objeto pueden variar al tiempo y durante el tiempo de ejecución, además cuenta con:

- Asignación dinámica de tipos.
- No hay verificación de tipos.
- No hay constructores de tipo (sólo el de listas).
- No puede construir subtipos.

En cuanto a las clases en LEPPOOC:

- Se pueden definir clases.
- Se pueden definir subclases.
- La clase de un objeto no puede cambiar dinámicamente.

1.1.8 Integridad Computacional

Desde el punto de vista de un lenguaje de programación, la integridad computacional es obvia, significa que uno puede expresar cualquier expresión computable. Sin embargo no lo es para bases de datos. Por ejemplo, SQL no tiene los constructores necesarios para expresar todas las funciones computables posibles. La integridad computacional en bases de datos se puede alcanzar con una interfaz razonable usando un lenguaje de programación existente. Por ejemplo ORION usa LISP, O^2 usa su propio lenguaje C extendido, GemStone se basa principalmente en Smalltalk, pero también puede importar código desde otros lenguajes.

Sin embargo, la ***integridad computacional*** es diferente de la ***integridad de recursos***, es decir, la posibilidad de accesar todos los recursos del sistema, tal como video y acceso remoto. El lenguaje, aunque fuera computacionalmente

completo, puede no ser suficientemente poderoso para expresar una aplicación completa.

LEPPOOC maneja el álgebra de los enteros y reales, manipulacion de listas, así como funciones recursivas, por lo que es computacionalmente completo.

1.2 Características de Bases de Datos

A continuación se presenta una descripción de las características de un Sistema Manejador de Base de Datos cualquiera, pero para efectos del prototipo sólo se consideró la persistencia.

1.2.1 Persistencia

La persistencia es un requerimiento evidente desde el punto de vista de base de datos, pero es una "novedad" desde el punto de vista de un lenguaje de programación. La persistencia es la capacidad del programador para hacer que el dato sobreviva después de la ejecución de un proceso, de modo que pueda utilizarlo eventualmente en otro proceso.

En otras palabras la persistencia se refiere al tiempo de vida de un dato, y en bases de datos orientadas a objetos, no sólo persiste el ***estado*** sino también su ***clase***. De modo que cada programa interprete en la misma forma el estado salvado.

En LEPPOOC la persistencia está asociada a la ***Creación/Destrucción*** de objetos, por lo que su manejo es explícito, sin embargo, si es un objeto compuesto, hace persistentes de manera implícita sus partes, como lo es en O_2 o en STONE.

1.2.2 Concurrencia

Para cierta clase de problemas, un sistema automatizado puede tener la necesidad de manejar muchos eventos diferentes simultáneamente. Otros, pueden involucrar demesiados cálculos que excedan le capacidad de cualquier procesador simple. Para cede uno de estos casos, es natural considerar el uso de un conjunto de computadoras distribuidas para su implantación, o el uso de procesadores cepaces de trabajar con multitareas. Un proceso simple es un canal de control. Cada programa tiene el menos un canal de control, pero un sistema que involucra concurrencia puede tener muchos canales de control. Algunos son transitorios y otros duran hasta el final de la ejecución del sistema. Los sistemas que se ejecuten a través de múltiples CPU's, permiten canales de control realmente concurrentes, mientras que los sistemas que se ejecutan sobre un solo CPU, pueden realizar solamente la simulación de canales de control concurrentes, a través de algunos algoritmos especiales.

El enfoque de la programación orientada a objetos se encuentra basado en la abstracción de datos, encapsulación y herencia. El enfoque de concurrencia se basa en la abstracción de procesos y su sincronización. En este caso, el objeto viene a ser el concepto que unifica estos dos puntos de vista diferentes. Cada objeto (una abstracción del mundo real) puede representar un canal de control separado (una abstracción del proceso). Tales procesos son llamados activos.

En un sistema orientado a objetos, se puede conceptualizar el mundo como un conjunto de objetos cooperativos, algunos de los cuales están activos y otros sirven como centros de actividad independientes.

1.2.3 Recuperación

El sistema proporcionaría el mismo nivel de servicios de recuperación que los sistemas de bases de datos concurrentes, esto significa que en casos de fallas de hardware y software, el sistema debe brindar por sí mismo el regreso a algún estado coherente de los datos para garantizar la consistencia de la base de datos. Las fallas de hardware incluyen las fallas del procesador y del disco.

1.2.4 Facilidad de Consultas

El sistema debe proporcionar la funcionalidad de un lenguaje de consulta "ad hoc". Los servicios deben consistir en permitir al usuario realizar consultas simples a la base de datos.

Una facilidad de consulta debe satisfacer tres criterios:

- Ser de alto nivel, esto es, una que sea capaz de expresar consultas no triviales de manera breve. Esto implica que la facilidad sea razonablemente declarativa, enfatizando el qué y no el cómo.
- Ser eficiente, esto es, que la formulación de las consultas conduzcan por sí mismas a alguna forma de optimización de consultas.
- Ser independiente de la aplicación, esto es, que trabajen cualquier base de datos posible.

1.3 Caracteristicas Opcionales

Las características opcionales mejoran el sistema, pero no son obligatorias para que sea un sistema de base de datos orientado a objetos, aunque lo hacen más orientado a objetos, no son requerimientos centrales. Otras características sólo perfeccionan la funcionalidad de un sistema de base de datos pero no son requerimientos centrales y no están relacionadas al aspecto orientado a objetos, estando más destinados a servir a nuevas aplicaciones que a tecnología.

1.3.1 Herencia Múltiple

Debido a que los investigadores del modelo Orientado a Objetos aún no se ponen de acuerdo sobre la herencia múltiple, se considera que su participación es opcional, una vez que el diseñador decide soportar herencia múltiple, se pueden encontrar muchas soluciones posibles para tratar con los problemas de resolución de conflictos, como los mencionados previamente en la sección de herencia.

1.3.2 Verificacion de Tipos o Inferencia de Tipos

El grado de verificación de tipos que el sistema debe ejecutar al tiempo de compilación está abierto. La situación óptima es cuando un programa que fue aceptado por el compilador no produzca errores de tipos al tiempo de ejecución. La cantidad de inferenciación de tipos se le deja también al diseñador del sistema. La situación ideal se encuentra donde solamente los tipos base se han declarado y el sistema infiere los tipos temporales.

1.3.3 Distribución

La distribución de los datos e información es ortogonal a la naturaleza de los sistemas orientados a objetos, esto significa que los sistemas de bases de datos pueden ser o no distribuidos.

1.3.4 Transacciones de Diseño

En la mayoría de las aplicaciones, el modelo de transacciones de un sistema de base de datos orientado a objetos clásico no es satisfactorio. Las transacciones tienden a ser muy grandes y los criterios de serialización usuales no son adecuados. Por esto, muchos sistemas de bases de datos orientados a objetos soportan transacciones grandes o anidadas.

1.3.5 Versiones

La mayoría de las nuevas aplicaciones, tales como CAD/CAM y CASE, involucran una actividad de diseño que requiere alguna forma de manejar versiones, por lo que la mayoría de los ***Sistemas de Base de Datos Orientados a Objetos (SBDOO)*** soportan versiones, proporcionando un mecanismo de versiones que no es un requerimiento central.

1.4 Características Abiertas

Todo sistema que satisface las reglas anteriores merece la etiqueta de SBDOO. Sin embargo, aquí se presentan otras características que no se sabe si son más o menos orientadas a objetos.

1.4.1 Paradigma de Programación

No se ve razón para imponer un paradigma de programación en lugar de otro. Los estilos de programación funcional, lógico o imperativo podrían ser elegidos. Otra solución es que el sistema sea independiente del estilo de programación y que soporte múltiples paradigmas de programación. Por supuesto, la elección de la sintaxis sería libre.

LEPPOOC manejará el estilo de programación funcional de SCHEME[8].

1.4.2 Sistema de Representación

El sistema de representación está definido por el conjunto de tipos atómicos y el conjunto de constructores, mismos que pueden ser extendidos en muchas formas diferentes.

1.4.3 Sistema de Tipos

Existe también libertad con respecto a los constructores de tipos más allá de los precedentes, sólo se requiere la encapsulación para facilitar la formación del tipo. Otra opción es que el sistema de tipos esté en segundo orden. Los sistemas de tipos para variables son quizá más ricos que los sistemas de tipos para objetos.

1.4.4 Uniformidad

Existe una gran discusión sobre el grado de uniformidad que uno espera de los SBDOO: ¿Un objeto es un tipo? ¿Un objeto es un método?. Podemos ver este problema en tres niveles diferentes:

- El nivel de implantación,
- El nivel de Lenguaje de Programación
- El nivel de Interfaz

En el ***nivel de implantación*** uno puede decidir si la información de tipos será almacenada como objetos o si se puede implantar un sistema "ad hoc". La decisión estaría basada en el rendimiento de la implantación. Cualquier decisión que se tome sin embargo, será independiente de la que se tome en el siguiente nivel.

En el ***nivel de lenguajes de programación***, la pregunta es: ¿Son los tipos, entidades de ***primera-clase*** en la semántica del lenguaje? La mayoría de las discusiones se concentran en esta pregunta. Probablemente existen estilos de uniformidad diferentes, sintácticos o semánticos. La uniformidad en este nivel es inconsistente con la verificación estática de tipos.

En el ***nivel de interfaz*** uno quisiera presentar al usuario una vista uniforme de tipos, objetos y métodos, aún si en la semántica de los lenguajes de programación estas son nociones distintas, o viceversa, uno podría presentarlos como entidades diferentes, aunque los lenguajes de programación los vean como el mismo. Esta decisión se debe basar en criterios de factor humano, por parte del usuario.

Todo LEPPOOC es un mismo lenguaje. La idea de la orientación a objetos en Bases de Datos es romper el ***"Language impedance"***, es decir la diferencia entre el sistema de tipos del lenguaje de programación y el sistema de tipos de los lenguajes de definición de datos (DDL) y manipulación de datos (DML) de los SMBD, así como también el requerir que el programador aprenda dos lenguajes de programación, lo que involucra escribir código para traducir las

estructuras de datos desde el lenguaje de programación a las estructuras del DDL y DML del DBMS.

La solución al problema de ***"Language impedance***[B]**"** requiere la integración de la tecnología de los lenguajes de programación con la tecnología de bases de datos.

1.5 Comentarios Finales

A manera de conclusión, se debe aclarar que tanto las características opcionales como las abiertas, aunque se consideraron en la explicación, no fueron el interés primordial en la construcción y diseño del prototipo LEPPOOC. La intención fue poner al descubierto un punto de vista proporcionado por investigadores en el área que desean precisar los conceptos en los que se deben basar los sistemas de bases de datos orientados a objetos.

[B] En ausencia de una traducción en español convincente, se decidió dejar el término en inglés.

2 OBJETOS COMPUESTOS

La mayoría de las extensiones semánticas que se han propuesto por distintos investigadores en el modelo orientado a objetos descrito en el ***capítulo uno*** de este trabajo, aún están en investigación, razón por la cual no se encuentran disponibles en varios de los ***Sistemas Manejadores de Bases de Datos Orientados a Objetos*** (SMBDOO) actuales. La única excepción es el concepto de los ***objetos compuestos*** que se ha incorporado en el modelo de datos de ORION[10].

Como se explicó, en el modelo orientado a objetos del capítulo uno, los ***objetos compuestos*** pueden definirse en términos de otros objetos. Sin embargo, una relación de agregación bajo este modelo no establece semántica adicional entre dos objetos. Para ciertas aplicaciones, por ejemplo hipertexto, es importante tener la capacidad para describir el hecho de que un objeto es parte de otro. Adicionar tales semánticas en las relaciones de agregación entre objetos tiene una repercusión considerable en la funcionalidad de las operaciones sobre los objetos.

El concepto de herencia en el modelo de datos orientado a objetos es poderoso; sin embargo, modela bien la relación ***es-un*** entre subclase y superclase, pero no modela agregación de objetos. Estas relaciones de

agregación de un objeto, son usualmente relaciones ***es-parte-de***, los cuales representan que un objeto es parte de otro objeto.

El concepto de objeto compuesto se ha introducido en algunos SMBDOO, como ORION[10], y en algunos lenguajes de programación, para facilitar las aplicaciones bajo el modelo de que varios objetos (conocidos como objetos componentes) constituyan una entidad lógica.

A continuación se presentan los dos modelos de objetos compuestos que Won Kim introdujo y experimentó en ORION[10].

2.1 MODELO INICIAL DE OBJETOS COMPUESTOS IMPLANTADO EN ORION

Un modelo de objetos compuestos inicial se propuso e implantó en el proyecto ORION[10]. Al probar ORION sobre un número considerable de aplicaciones se mostró que el concepto de objetos compuestos es extremadamente útil. Sin embargo, también surgieron a la luz otro número de problemas.

◊ El primer problema que se presentó, es que un objeto componente puede pertenecer a un solo objeto (la propiedad de exclusividad). Esto se ajusta precisamente al modelo de *"jerarquía física de partes "*, en la cual un objeto no puede ser parte de más de un objeto. Sin embargo, esto no es aceptable para una *"jeraquía lógica de partes "*, ya que esta restricción es una limitante para algunas aplicaciones, por ejemplo, en un sistema de hipertexto, es lógico pensar que un capítulo podría pertenecer a dos libros diferentes.

En este punto, WON KIM[10] distingue dos tipos de jerarquía de partes: ***física*** y ***lógica***. En la jerarquía física de partes, todas las referencias compuestas son

exclusivas; mientras que una jerarquía lógica de partes puede también contener referencias compuestas compartidas.

◊ El segundo problema es que el modelo requiere que los objetos compuestos se construyan de arriba hacia abajo (en forma descendente).

Por lo tanto, el objeto componente O no se puede crear si el objeto padre no se ha creado antes (el objeto padre de O es el objeto del cual O es una componente directa). Esta restricción significa que los objetos compuestos no pueden ser creados de abajo hacia arriba (en forma ascendente), es decir que no se pueden ensamblar objetos ya existentes.

◊ Finalmente, el modelo requiere la dependencia existencial de los objetos componentes desde los objetos compuestos a los que pertenecen.

Si un objeto compuesto se destruye, todos los objetos componentes son automáticamente destruidos por el sistema. Esto es útil porque las aplicaciones no tienen que buscar y destruir explícitamente todos los objetos componentes. Sin embargo, esto significa que no es posible reutilizar las componentes de un objeto compuesto destruido para crear un nuevo objeto compuesto.

A continuación presentamos el modelo de objetos compuestos extendido, propuesto también por WON KIM[10] para tratar de eliminar estos problemas.

2.2 MODELO DE OBJETOS COMPUESTOS EXTENDIDO.

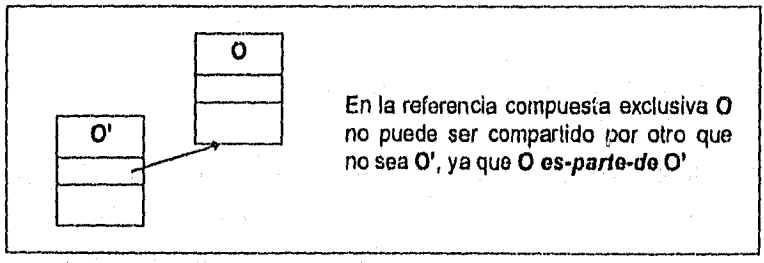
Un segundo modelo que elimina las desventajas que se mostraron con el primer modelo, también se definió e implantó en ORION[10]. En este modelo se definieron dos tipos de referencias entre objetos: ***débiles*** y ***compuestas***.

- ◊ REFERENCIA DEBIL: Una referencia débil es una referencia normal, entre objetos sobre la cual no se agrega ninguna semántica adicional, esta es la definida por la herencia. Un objeto O tiene una referencia a un objeto *O'* si esta referencia es el valor de un atributo de *O*.

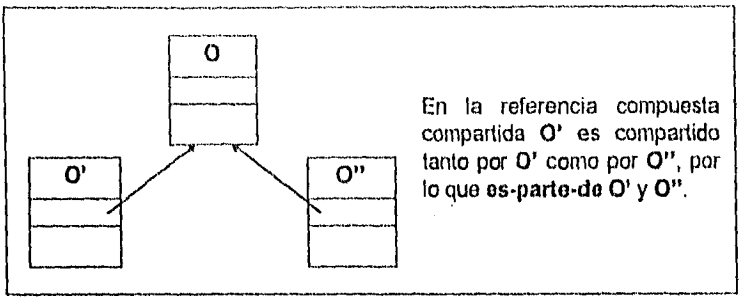
- ◊ REFERENCIA COMPUESTA: Una referencia compuesta es una referencia en la cual se agrega la referencia **es-parte-de**. Una referencia compuesta puede ser a su vez, ***exclusiva*** o ***compartida***.

En el caso de la referencia compuesta exclusiva, el objeto referenciado debe pertenecer a un solo objeto compuesto. La semántica de una referencia compuesta se refina introduciendo la diferencia entre ***referencias compuestas dependientes*** e ***independientes***. En el primer caso, la existencia de un objeto referenciado es dependiente de la existencia del objeto al cual pertenece, mientras que en el último caso es independiente. La destrucción de objetos compuestos, solamente resulta en la destrucción de los objetos componentes que son dependientes de su existencia. Los objetos cuya existencia es independiente no se destruyen. Dado que la característica de ***dependencia/independencia*** es ortogonal con respecto a la característica del estado ***exclusivo/compartido***, se obtienen cuatro tipos posibles de referencias compuestas.

Para aclarar esta discusión, se muestran las gráficas 2.1 y 2.2. que explican estas referencias.



Gráfica 2.1 Referencia Compuesta Exclusiva



Gráfica 2.2 Referencia Compuesta Compartida

2.2.1 Tipos de Referencias Compuestas

Los cuatro tipos de referencias compuestas que se deducen de la explicación anterior son:

- Referencias compuestas dependientes exclusivas.
- Referencias compuestas independientes exclusivas.
- Referencias compuestas dependientes compartidas.
- Referencias compuestas independientes compartidas.

El único tipo de referencia que se definió en el primer modelo de objetos compuestos de ORION[10], es el de las referencias compuestas dependientes exclusivas, además de la referencia débil.

Bajo el modelo implantado en ORION, la raíz de un objeto compuesto nunca cambia (debido a la creación de arriba-abajo de un objeto compuesto). Bajo el modelo extendido, la raíz de un objeto compuesto puede cambiar, es decir, un objeto que es la raíz actual de un objeto compuesto puede ser el destino de una referencia compuesta desde otro objeto.

Una referencia que va de un objeto O' a otro objeto O, significa que el valor de un atributo de O' es el identificador de O.

Si un atributo tiene una referencia compuesta, el atributo es llamado atributo compuesto.

Los diferentes tipos de referencias particionan el conjunto de objetos que hacen referencia a un objeto dado, en cuatro conjuntos diferentes de objetos. Se pueden formalizar aspectos importantes de la semántica extendida de objetos compuestos, en términos de restricciones de estos conjuntos de objetos.

Definiciones:

- Sea $IX(O)$ el conjunto de objetos que tienen referencias compuestas exclusivas independientes a O.
- Sea $DX(O)$ el conjunto de objetos que tienen referencias compuestas exclusivas dependientes a O.
- Sea $IS(O)$ el conjunto de objetos que tienen referencias compuestas compartidas independientes a O.
- Sea $DS(O)$ el conjunto de objetos que tienen referencias compuestas compartidas dependientes a O.

Supóngase una referencia compuesta desde el objeto O' al objeto O, entonces tendríamos que la operación $del(O')$ correspondería a la destrucción del objeto O'. Precisando $del(O')$, con respecto a los cuatro tipos de referencias compuestas desde O' a otro objeto O, tendríamos que:

- Si se tiene una referencia compuesta exclusiva independiente desde O' a O, entonces: $del(O') \not\Rightarrow del(O)$, lo que significa que la destrucción del objeto O' no implica forzosamente la destrucción del objeto O.

- Si se tiene una referencia compuesta exclusiva dependiente desde O' a O, entonces: $del(O') \Rightarrow del(O)$, es decir que la destrucción del objeto O' implica la destrucción del objeto O.

- Si se tiene una referencia compuesta compartida independiente desde O' a O, entonces: $del(O') \not\Rightarrow del(O)$, es decir que la destrucción del objeto O' no implica forzosamente la destrucción del objeto O.

- Si se tiene una referencia compuesta compartida dependiente desde O' a O, entonces: $del(O') \Rightarrow del(O)$ solamente si $DS(O) = \{O'\}$, esto significa que la destrucción del objeto O' implicaría la destrucción del objeto O, solamente si todas las referencias compuestas compartidas dependientes a O han sido borradas, de otro modo $DS(O) = DS(O) - O'$. Aclarando esta definición, se dice que el destruir un objeto O' que tiene como componente a otro objeto O implicaría borrar el objeto O sólo si este no está siendo compartido con otros objetos; de otro modo sólo se borraría la referencia O' de O.

Los cuatro tipos de referencias tratados previamente se pueden combinar para definir entre los elementos de un conjunto de objetos una serie de relaciones correspondientes a una generalización rigurosa conocidas como topologías.

Suponga que ***card(S)*** denota la cardinalidad del conjunto S.

◊ **Regla de topología 1:**

$$card(IX(O)) = 1 \Rightarrow card(DX(O)) = 0$$
$$card(DX(O)) = 1 \Rightarrow card(IX(O)) = 0$$

Esto significa que si un objeto O tiene una referencia compuesta exclusiva independiente a él, entonces no puede tener una referencia compuesta exclusiva dependiente desde otro objeto y viceversa.

◊ **Regla de topología 2:**

$$(card(IX(O)) = 1 \text{ o } card(DX(O)) = 1) \Rightarrow (card(IS(O)) = O \text{ y } card(DS(O)) = 0)$$
$$(card(IS(O)) > 0 \text{ o } card(DS(O)) > 0) \Rightarrow (card(IX(O)) = 0 \text{ y } card(DX(O)) = 0)$$

Esto significa que si un objeto *O* tiene una referencia compuesta exclusiva desde otro objeto, entonces no puede tener referencias compuestas compartidas y viceversa.

◊ **Regla de topología 3:**

> *Un objeto O puede tener cualquier número de referencias débiles a él, aún cuando tenga referencias compuestas a él.*

Suponga que se quiere hacer un objeto *O* parte de un objeto *O'* a través de un atributo *A* de *O'*, para esto se deben satisfacer las siguientes condiciones.

REGLA DE CREACION DE COMPONENTES:

I) Si *A* es un atributo compuesto exclusivo, el objeto *O* no debe tener referencias compuestas a él.

II) Si *A* es un atributo compuesto compartido, el objeto *O* no debe tener referencias compuestas exclusivas.

La siguiente regla resume la semántica de destrucción de un objeto en un objeto compuesto.

REGLA DE DESTRUCCION

Suponga que *O'* es la raíz de un objeto compuesto y que *O* es una componente de *O'*.

del(O') ⇒ ***del(O)*** si cumple con cualquiera de las tres condiciones siguientes:

I) *O'* tiene una referencia exclusiva dependiente al objeto *O*.
II) *O'* tiene una referencia compartida dependiente al objeto *O y DS(O)={O'}*.
III) Un objeto *O"* existe de forma tal que *del(O')* => *del(O")* si se cumple al menos una de las condiciones siguientes:
   a) *O"* tiene una referencia compuesta exclusiva dependiente el objeto *O*
   b) *O"* tiene una referencia compuesta compartida dependiente a *O* y DS(O) = {O"}.

Para terminar, hablaremos de la evolución del esquema, que aunque no se incluye en el prototipo, es interesante mencionarlo para contar con un panorama más amplio de este modelo de objetos compuestos.

2.3 EVOLUCION DEL ESQUEMA

La evolución del esquema es la especificación de un conjunto de cambios dinámicos al esquema de la base de datos y a la semántica de cada uno de los cambios.

2.3.1 Destrucción de un Atributo Compuesto.

El modelo de objetos compuestos original que se implantó en ORION, determina que todos los objetos referenciados a través de un atributo compuesto sean destruidos si el atributo se elimina; sin embargo, el modelo extendido borra solamente aquellos objetos que son referenciados a través de atributos compuestos dependientes cuando los atributos son borrados.

1) Dar de baja un atributo *A* desde una clase *C*.

Esta operación causa que todos los ejemplares de la clase *C* pierdan sus valores para el atributo *A*. Si A es un atributo compuesto, los objetos que están referenciados a través de *A* son destruidos de acuerdo a la Regla de Destrucción. El atributo también debe darse de baja desde todas las subclases que lo heredan.

2) Cambiar la herencia (padres) de un atributo (hereda otro atributo con el mismo nombre).

Dependiendo del origen del viejo y nuevo atributo, esta operación puede causar que el atributo viejo se destruya. Si el atributo dado de baja es un atributo compuesto, esta operación es idéntica a la (1).

3) Borrar una clase *S* como superclase de una clase *C*.

Si esta operación causa que la clase *C* pierda un atributo compuesto *A*, los objetos (de otras clases) que son recursivamente referenciados por ejemplares de *C* y sus subclases a través de A son borrados de acuerdo a (1).

4) Dar de baja una clase *C* existente.

Si la clase *C* tiene uno o más atributos compuestos, los objetos referenciados a través de los atributos son dados de baja de acuerdo a la Regla de Destrucción. Todas las subclases de C vienen a ser inmediatamente subclases de la superclase de *C*.

2.3.2 Cambios a los Tipos de los Atributos.

Se explorarán cambios significativos desde un tipo de atributos a uno diferente en el contexto del modelo extendido de objetos compuestos. Los cambios pueden ser de uno de dos tipos con respecto a la implantación: ***cambios independientes del estado*** y ***cambios dependientes del estado***.

En forma general se puede decir que un cambio independiente del estado es un cambio que borra una restricción desde una referencia compuesta, mientras que un cambio dependiente del estado, agrega una restricción a una referencia.

A continuación, se muestran los cambios independientes del estado a los tipos de atributos que son significativos bajo la semántica extendida de referencias compuestas.

1) Cambiar un atributo compuesto a un atributo no compuesto.

2) Cambiar un atributo compuesto exclusivo a un atributo compuesto compartido.

3) Cambiar un atributo compuesto dependiente a un atributo compuesto independiente.

4) Cambiar un atributo compuesto independiente a un atributo compuesto dependiente.

Los cambios dependientes del estado para los atributos compuestos son como sigue, supóngase que la clase *C* es el dominio de un atributo *A* de una clase *C'*, y que *A* se cambiará.

1) Cambiar un atributo no compuesto a un atributo compuesto exclusivo.

No deben existir referencias compuestas a ejemplares de la clase C que son referenciados por los ejemplares de la clase *C'*.

2) Cambiar un atributo no-compuesto a un atributo compuesto compartido.

La regla de topología 2 debe verificarse para asegurar que no existen referencias compuestas exclusivas a los ejemplares de la clase *C* que son referenciados por los ejemplares de la clase *C'*.

3) Cambiar un atributo compuesto compartido a un atributo compuesto exclusivo.

La regla de topología 2 debe verificarse para garantizar que existe al menos una referencia compartida a ejemplares de la clase *C* que son referenciados por ejemplares de la clase *C'*.

2.3.3 Implantación de los Cambios.

La implantación de los cambios a los tipos de atributos, involucran el acceso a los objetos referenciados y la actualización de banderas que indican si son dependientes *(D)* o exclusivos *(X)* de las referencias compuestas que van hacia atrás. La validación de un cambio dependiente del estado depende de la consistencia de estas banderas, esto es, si las banderas no están en un estado consistente con los requerimientos semánticos de los cambios, el cambio se rechaza. Por esta razón, los cambios dependientes del estado simplemente requieren verificación "inmediata" de las banderas, sin embargo, los cambios independientes del estado simplemente requieren actualizar a estas banderas, como tales, los cambios pueden hacerse "inmediatamente" o de modo "diferido", hasta que el objeto realmente necesite ser accesado.

A continuación se describirán las dos opciones para implantar cambios independientes del estado.

2.3.3.1 Cambios Independientes del Estado.

La implantación "inmediata" de los cambios es como sigue. Suponga que una clase *C'* tiene un atributo compuesto *A* cuyo dominio es una clase *C*.

i. Cambiar un atributo compuesto a un atributo no compuesto. Esto se implementa accesando todas las instancias de la clase *C* y dando de baja referencias compuestas hacia atrás a ejemplares de la clase *C'*.

ii. Cambiar un atributo compuesto exclusivo para compartir atributos compuestos. Esto es implantado, accesando todos los ejemplares de la clase *C* y apagando la bandera ***"X"*** en la referencia compuesta hacia atrás a ejemplares de la clase *C'*.

iii. Cambiar un atributo compuesto dependiente a un atributo compuesto independiente. Esto se implanta, accesando todos los ejemplares de la clase *C* y apagando la bandera *D* en las referencias compuestas hacia atrás a ejemplares de la clase *C'*.

iv. Cambiar un atributo compuesto independiente a un atributo compuesto dependiente. Esto se implanta accesando todos los ejemplares de la clase *C* y encendiendo la bandera *D* en las referencias compuestas hacia atrás a ejemplaress de la clase *C'*.

La implantación de los cambios independientes del estado, involucran llevar una ***bitácora*** de operaciones de cambios a los tipos de atributos en una clase. Una clase tiene ***n*** bitácoras de operaciones, una para cada atributo de la cual la ***clase*** es el dominio. Una bitácora de operación para una clase *C* mantiene, para cada cambio, el tipo de cambio y un contador *(CC)* de cambios, así como también el identificador de la clase de cuyo atributo *C* es el dominio. Inicialmente *CC* es cero y se incrementa en uno cada vez que se cambia el tipo de atributo en una clase.

Ahora se discutirá la implantación de cambios dependientes del estado para tipos de atributos.

2.3.3.2 Cambios Dependientes del Estado.

La operación de cambiar una referencia débil a una referencia compuesta compartida se implanta como sigue. Supongamos que una clase *C'* tiene un atributo *A* cuyo dominio es una clase *C*.

i. Accesar todos los ejemplares de la clase *C'* y determinar todos los ejemplares de la clase *C* las cuales son referenciadas a través de *A*.

ii. Para todos los ejemplares de la clase *C* determinadas antes, verificar que no tengan referencias exclusivas.

iii. Si cualquier ejemplar tiene una referencia exclusiva a ella, rechazar el cambio, de otro modo, agregar referencias compuestas hacia atrás a los ejemplares de *C*.

La operación para cambiar una referencia compuesta compartida a una referencia compuesta exclusiva se implanta en una forma similar.

i. Accesar todos los ejemplares de la clase *C*.

ii. Rechazar el cambio si un ejemplar *O* existe tal que *O* tiene más de una referencia compuesta hacia atrás, y al menos una de las referencias compuestas hacia atrás va de un ejemplar de la clase *C'*. De otro modo, enciende la bandera ***"X"*** en todas las referencias compuestas hacia atrás a los ejemplares da la clase *C'*.

En el modelo extendido, WON KIM[10] también incluyó cambios al modelo de versiones de objetos compuestos implantado en ORION, así como el modelo de autorización, pero como esto se sale por completo del alcance de este trabajo, no será discutido.

2.4 COMENTARIOS FINALES.

Finalmente sólo resta determinar los niveles de pertenencia que maneja LEPPOOC, por lo que de acuerdo a lo previamente descrito, se considera que LEPPOOC cuenta con:

I. Referencia débil, por incluir el concepto de herencia.

II. Referencias compuestas dependientes exclusivas, ya que la destrucción de un objeto compuesto, sólamente se da si se destruyen los objetos componentes que son dependientes de su existencia (listas).

III. Referencias compuestas independientes compartidas, ya que la destrucción de un objeto compuesto, cuyas componentes las comparte con otros objetos, no implica la destrucción de estas componentes.

3 SINTAXIS DEL LENGUAJE

Existen dos clases de traductores: Los lenguajes compilados y los interpretados[13].

En los ***lenguajes compilados*** los programas generalmente son traducidos al lenguaje máquina de la computadora que está siendo utilizada antes de su ejecución. Típicamente el traductor para un lenguaje compilado es relativamente grande y complejo, y el énfasis de la traducción se encuentra en la producción de un programa traducido que se ejecuta tan eficientemente como sea posible. Mientras que en la implantación de los ***lenguajes interpretados***, el traductor o intérprete es el código ejecutable que se conduce por nuestro programa. El intérprete se ejecuta cada vez que se ejecuta nuestro programa. El uso de un intérprete de software ordinariamente resulta en la ejecución de programas relativamente lentos. Los lenguajes interpretados tienden a requerir más memoria, pero son más simples de usar. Nuestro prototipo es un lenguaje interpretado.

A continuación en la figura 3.1 se muestra como funcionan los lenguajes compilados e interpretados.

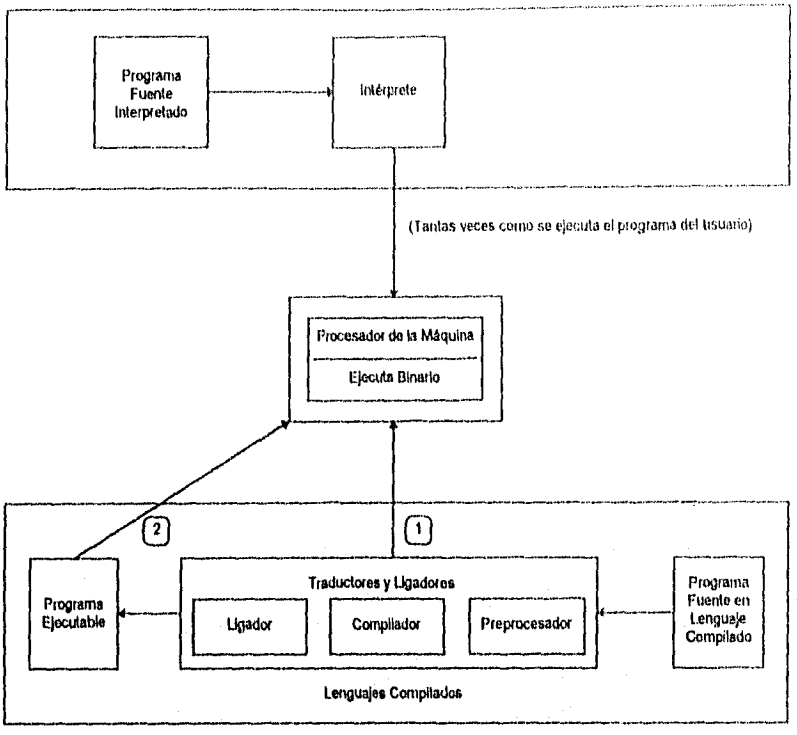


Figura 3.1: Funcionamiento de los Lenguajes Compilados e Interpretados.

3.1 EL LENGUAJE

En esta sección se describe el lenguaje LEPPOOC cuyos constructores son considerados un subconjunto de LISP. Así mismo consideramos que es un lenguaje simple, por ser un intérprete. En LEPPOOC las instrucciones se encuentran clasificadas en: ***Expresiones***, que sólo se evalúan, ***Definiciones***, que sólo se recuerdan y en ***Comandos***, que ayudan en la administración de memoria.

◊ **Expresiones:**

- Por ejemplo,

```
(set x 5)
(doblar 8)
(quote y)
(begin (+ (set x 7) 9))
```

◊ **Definiciones:**

- De *función*, tales como:

```
(define doblar (x) (+ x x))
```

- De *clase*, tales como:

```
(class circulo object (x y r)
      (define xc () x)
      (define yc () y)
      (define rad () r))
```

◊ **Comandos:**

- Comando para volver persistentes las clases y los objetos:

```
(persist-class)
```

- Comando que lae el archivo que contiene el código fuente del programa en LEPPPOOC:

```
(read "complejo")
```

- Comando que permite salir de LEPPOOC:

```
(bye)
```

A continuación, corresponde hablar de la sintaxis de LEPPOOC.

3.2 SINTAXIS

Cualquier notación para la descripción de algoritmos y estructuras de datos puede ser denominado un lenguaje de programación, aunque generalmente también requiere que se implante en una computadora. Una habilidad

importante en cualquier disciplina de diseño es el uso de herramientas descriptivas para formular, registrar y evaluar varios aspectos del diseño desarrollado. Aunque el español y otros lenguajes naturales pueden con frecuencia ser usados para este propósito, muchos diseñadores han desarrollado lenguajes especializados, diagramas y notaciones para representar aspectos de su trabajo que de otro modo sería difícil expresar. En este trabajo se usará la notación BNF extendida para representar la sintaxis del lenguaje, por ser la más universal.

Sintaxis De LEPPOOC Bajo La Notación BNF

```
<input>          ::= <exp> | <definicion> | <comando>
<definicion>     ::= <fundef> | <classdef>
<comando>        ::= <readcom> | <persist-class> | <byecom>
<exp>            ::= valor | <variable> | (if exp exp exp)
                     | (set variable exp) | (while exp exp)
                     | (begin exp+) | (persist variable)
                     | (new n-class) | (optr exp*)
                     | (unpersist variable)
                     | (set-shallow nombre variable)
                     | (quote nombre)
<optr>           ::= <n-function> | <n-objeto> | <n-mensaje>
                     | <value-op>
<valor>          ::= <entero> | <real> | <cadena>
<value-op>       ::= + | - | * | / | = | < | > | print | lista
                     | car | cdr
<n-objeto>       ::= <nombre>
<n-mensaje>      ::= (nombre+)
<variable>       ::= (cualquier caracter+)
<entero>         ::= (+-) <digito+>
<real>           ::= (+-) <digito+.digito+>
<cadena>         ::= "#cualquier caracter*"
<nombre>         ::= <letra(letra | digito*)>
<letra>          ::= a |...| z
<digito>         ::= 0 | 1 |...| 9
<classdef>       ::= (class n-class n-class inst-var methoddef*)
<inst-var>       ::= <(variable*)>
<methoddef>      ::= <fundef>
<n-class>        ::= <nombre>
<n-function>     ::= <nombre>
<persist-class>  ::= (persist-class)
<readcom>        ::= (read "nombre")
<byecom>         ::= (bye)
```

NOTA:

Los símbolos * y + que aparecen en superíndice denotan uno o más y cero o más apariciones respectivamente.

Bajo esta sintaxis es obvio que una función no debe ser una de *las "palabras clave"* ***define, if, while, begin, set, unpersist, set-shallow, quote, new, class, persist-class, read*** o cualquiera de los ***value-op***. El usuario de LEPPOOC debe ser cuidadoso al crear sus nombres de *variables, funciones, objetos y clases*, ya que la actual versión de LEPPOOC no verifica si se alteran o no las palabras claves. Por otro lado, las cadenas pueden usar cualquier caracter. Una sesión se concluye introduciendo ***(bye)***, por lo que es completamente inconveniente, aunque sea posible, usar este nombre para una variable, objeto o clase.

Como en LISP, en LEPPOOC, las expresiones se escriben entre paréntesis. El propósito es simplificar la sintaxis eliminando problemas de reconocimiento sintáctico como la precedencia de operadores, entre otros. Esta forma puede no ser atractiva, pero su uso es más práctico.

En la siguiente sección se dará un vistazo a LEPPOOC, para mostrar su funcionamiento.

3.2.1 Un Preámbulo a LEPPOOC

En este apartado se hará una presentación informal del significado de las expresiones de LEPPOOC, en el capítulo cinco se tratará con mayor formalidad.
En los ejemplos que se mostrarán, se considerarán las siguientes convenciones:

1) **LEPPOOC>** representa el "prompt" de nuestro intérprete, por lo que en seguida se podrá escribir la expresión que se desee, resultando ser la entrada del usuario.

2) El intérprete permite expresiones que se introducen en más de una línea, si se introduce una expresión incompleta, el intérprete espera por más entradas.

3) En la línea de abajo aparecerá el resultado de la expresión introducida previamente, que viene a ser la respuesta del intérprete.

4) En la línea siguiente aparecerá de nuevo el "prompt".

Es importante mencionar de manera informal (ya que serán tratados en el capítulo seis con más detalle), que LEPPOOC maneja dentro de su estructura interna tres ambientes, que le permiten manipular los valores que regresa.

3.2.1.1 Ambientes de LEPPOOC

LEPPOOC maneja tres ambientes: ***local (Al), global (Ag)*** y ***persistente (Ap).***

En el ***ambiente local*** se guardan las variables que ocurren dentro de la definición de una función o clase (los parámetros formales o variables de instancia). No existen variables locales por sí mismas, solamente parámetros formales o variables de instancia.

En el ***ambiente global*** se guardan las variables, objetos, definiciones de clases y funciones que ocurren a nivel superior dentro de un programa escrito en LEPPOOC.

En el ***ambiente persistente*** se guardan las variables, objetos, definiciones de clases y funciones que el usuario especifica con la instrucción que LEPPOOC proporciona para este caso y entre varias sesiones del intérprete.

De este modo, cuando una referencia a una variable:

- ◊ Ocurre en una expresión en el nivel superior de un programa en LEPPOOC, necesariamente es global, siempre que no se encuentre en el ambiente persistente.

- ◊ Si ocurre dentro de la definición de una función, y si la función tiene un parámetro formal con este nombre, la variable es este parámetro, de otro modo sería una variable global o persistente.

- ◊ Si ocurre dentro de la definición de una clase, y si la clase tiene una variable de instancia con este nombre, la variable es esta; de otro modo sería una variable global o persistente.

Por otro lado, como se mencionó, se pueden tener variables en el ambiente persistente, cuando el usuario las haga así explícitamente.

En la siguiente sección se explica el significado de los elementos de LEPPOOC.

3.2.1.2 Elementos de LEPPOOC.

Los elementos de LEPPOOC, van desde los más simples como los ***números*** y ***cadenas***, hasta los complejos como la instrucción ***(n-objeto n-met vi1...vn)***.

- Los ***enteros*** y ***reales*** son el tipo NUMERO que reconoce LEPPOOC, y las ***cadenas*** del tipo CADENA que reconoce LEPPOOC; y son sólo valores.

  Ejemplos de la interpretación de números y cadenas.

  Ejemplo NUMERO:

```
LEPPOOC>5
5
```

  Ejemplo CADENA:

```
LEPPOOC>"#soy una cadena"
"#soy una cadena"
LEPPOOC>
```

- Los ***value-op*** toman dos argumentos, excepto ***print, car*** y ***cdr,*** que toman uno y *lista* que toma cero o varios.

  - Los operadores aritméticos **+, -, *** y **/**, hacen lo obvio.

    Ejemplos de OPERADORES ARITMETICOS:

```
LEPPOOC>(+ 5 6)
11
LEPPOOC>(- 10 5)
5
LEPPOOC>(* 4 7)
28
LEPPOOC>(/ 10 2)
5
LEPPOOC>
```

◊ Los operadores de comparación, hacen la verificación indicada y regresan, ya sea cero para falso, o cualquier otro valor distinto de cero para verdadero.

Ejemplos de OPERADORES DE COMPARACION:

```
LEPPOOC>(< 5 7)
1
LEPPOOC>(> 2 5)
0
LEPPOOC>(= 6 6)
1
LEPPOOC>
```

◊ El value-op ***(print arg).*** Evalúa su argumento ***arg***, imprime su valor y regresa su valor, por lo que el efecto es que el usuario ve dos veces el valor impreso del argumento ***arg***.

Ejemplos de PRINT:

```
LEPPOOC>(print 6)
6
6
LEPPOOC>(print "#soy cadena")
"#soy cadena"
"#soy cadena"
LEPPOOC>
```

Como vemos ***(print 6)***, imprime primero el valor de **6**, y luego regresa el valor de esta expresión; el intérprete siempre imprime el valor de una expresión, de este modo para este caso, **6** es impreso dos veces.

◊ El value-op ***(lista a1 a2 ... an).*** Evalúa sus argumentos ***a1, a2, ..., an*** crea una lista con ellos y regresa la lista;

Ejemplos de LISTA:

```
LEPPOOC>(lista)
NIL
LEPPOOC>(lista 8)
(8)
LEPPOOC>(lista 2 3 4 5)
(2 3 4 5)
LEPPOOC>(set x 5)
5
LEPPOOC>(lista (set y 4) x 10)
(4 5 10)
LEPPOOC>(lista (+ 4 5) 4)
(9 4)
LEPPOOC>(lista (lista 1 2 3 4) 10 22)
((1 2 3 4) 10 22)
LEPPOOC>
```

◊ El value-op ***(car a1).*** Evalúa sus argumentos a_1 y si determina que es una lista, regresa como resultado el primero de la lista, de lo contrario envía un mensaje de error.

Ejemplos de CAR.

Suponiendo que se tienen almacenadas las listas siguientes:

ℓ_0	con la lista vacía	()
ℓ_1	con la lista	(4 5 6)
ℓ_2	con la lista	((1 2 3 4) 10 22)
ℓ_3	con la lista	(3)

Entonces tenemos:

```
LEPPOOC>(car ℓ0)
nil
LEPPOOC>(car ℓ3)
3
LEPPOOC>(car ℓ1)
4
LEPPOOC>(car ℓ2)
(1 2 3 4)
LEPPOOC>
```

◊ El value-op ***(cdr a1)***. Evalúa su argumento ***a1*** y si determina que es una lista, regresa como resultado el resto de la lista, de lo contrario envía un mensaje de error.

Ejemplos de CDR:

```
LEPPOOC>(cdr ℓ0)
nil
LEPPOOC>(cdr ℓ3)
nil
LEPPOOC>(cdr ℓ1)
(5 6)
LEPPOOC>(cdr ℓ2)
(10 22)
LEPPOOC>
```

Las expresiones como ***if, while, set, set-shallow, quote, persist, unpersist, new,*** son llamadas operaciones de control.

➢ La expresión ***(set x e)***. Evalúa **e**, asigna su valor a la variable ***x***, y regresa su valor.

Ejemplos de la expresión SET:

```
LEPPOOC>(set y 5)
5
LEPPOOC>(set r (+ y 6))
11
LEPPOOC>r
11
LEPPOOC>(set ℓ1 (llsta y 1 2 3 4))
(5 1 2 3 4)
LEPPOOC>ℓ1
(5 1 2 3 4)
LEEPOOC>(set x 5)
5
LEPPOOC>(set d (set z (* x x)))
25
LEPPOOC>d
25
LEPPOOC>z
25
```

- La expresión ***(begin e1 ... en)***. Evalúa cada uno de los ***e1*** ... ***en***, en este orden, y regresa el valor de ***en***.

  Ejemplo de la expresión BEGIN:

```
LEPPOOC> (set x 6)
6
LEPPOOC>(begin (print x) (print y) (* x y))
6
5
30
LEPPOOC>
```

En las condicionales ***if*** y ***while***, ***cero*** representa falso y ***uno*** (o cualquier otro valor distinto de cero) representa verdadero.

- La expresión ***(if e1 e2 e3)***. El intérprete evalúa esta expresión de la siguente forma, si ***e1*** se evalúa a cero, entonces evalúa ***e2***, de lo contrario evalúa ***e3***.

  Ejemplo de la condicional IF:

```
LEPPOOC>(if (< y x) 7 10)
7
LEPPOOC>
```

- La expresión ***(while e1 e2)***. Evalúa ***e1***, si se evaluó a uno, evalúa ***e2***, y vuelve a reevaluar ***e1***; así continúa hasta que ***e1*** se evalúa a uno. La expresión ***while*** siempre regresa el mismo valor, el cual no importa, ya que se evalúa solamente para efectos colaterales.

  Casi siempre el cuerpo de una expresión ***while*** (su segundo argumento), es una expresión ***begin***.

Ejemplo de la condicional WHILE:

```
LEPPOOC> (set y 5)
5
LEPPOOC> (set w 2)
2
LEPPOOC> (while (> y 0)
                (begin (set w (+ w w)) (set y (- y 1))))
0               Este resultado es de la evaluación del while no de su cuerpo.
LEPPOOC> Y
0
LEPPOOC> W
64
LEPPOOC>
```

Como se vió en la sección de ambientes de LEPPOOC se puede hacer que una variable, objeto, función o clase sea persistente, utilizando sólamente la expresión ***persist***. Así mismo existe su contraparte, en caso de que se quiera eliminar una variable, objeto, función o clase del ambiente persistente, bastará con aplicar la expresión ***unpersist***.

- La expresión ***(persist nom)***. Esta busca la variable, nombre de objeto, clase o función *"nom"*, en un ambiente global y si existe lo pasa al ambiente persistente, siempre que no se encuentre ya en él, de lo contrario envía un mensaje indicándolo.

Ejemplos de PERSIST:

```
LEPPOOC>(persist w)
64
LEPPOOC>(persist ℓ1)
(5 1 2 3 4)
LEPPOOC>
```

Si después de estas ejecuciones se hace referencia a ***w*** o ***ℓ1***, entonces las buscará y tomará sus valores del ambiente persistente.

- La expresión ***(unpersist var).*** Esta busca la variable, nombre de objeto, clase o función *"var"* en el ambiente persistente, si existe lo manda al ambiente global, de lo contrario envía un mensaje de error.

  Ejemplo de UNPERSIST.

```
LEPPOOC>(unpersist w)
64
LEPPOOC>
```

  Si luego de esta ejecución se hace referencia a ***w***, entonces la buscará y tomará su valor del ambiente global.

Las expresiones ***set-shallow*** y ***quote*** sirven para crear *"alias"* de objetos o valores, lo que permitirá tener referencias a objetos o valores sin tener una réplica de ellos, permitiéndo compartir objetos. Así mismo permitirán distinguir entre el ***nombre del objeto*** y el ***objeto en sí mismo***, manejando así el concepto de ***identidad***, que viene a ser uno de los características que debe satisfacer al modelo orientado a objetos, mismo que se discutió ampliamente en al capítulo uno.

- La expresión ***(set-shallow n1 n2).*** Esta busca la variable o nombre de objeto ***n2***, de existir crea un ***"alias"*** del objeto o variable ***n2***, de lo contrario envía un mensaje de error.

  Ejemplo de SET-SHALLOW.

```
LEPPOOC>(set-shallow nvo-w w)
w
LEPPOOC>nvo-w
w
LEPPOOC>
```

➢ La expresión ***(quote n1)***. Esta crea un ***"alias"*** de ***n1***, sin verificar si existe o no, el nombre ***n1*** en los ambientes.

Ejemplo de QUOTE:

```
LEPPOOC>(quote h)
h
LEPPOOC>
```

En el ejemplo, y en general, el hacer ***(set nvo-w (quote w))***, es prácticamente lo mismo que hacer ***(set-shallow nvo-w w)***. Simplemente son dos formas de implantar la identidad. Por otro lado la expresión ***quote*** sirve para manejar los nombres de los objetos compuestos como tales cuando se requiera, en virtud de que LEPPOOC siempre evalúa las expresiones, de esta manera respetará el nombre del objeto.

En LEPPOOC, el usuario introduce las definiciones de *función, clase* y *expresiones* interactivamente. Las definiciones de las *funciones* y las *clases* son almacenadas e impresas; las *expresiones* son evaluadas y sus valores impresos.

➢ La expresión ***(define f (v1 ... vn) e1)***. Con ésta se define la función ***f***, el intérprete devuelve tal definición.

Ejemplos de la definición de FUNCION:

```
LEPPOOC>(define inc (x) (+ x 1))
((x) (+ x 1))
LEPPOOC>(define dobla (x) (+ x x))
((x) (+ x x))
LEPPOOC>
```

Para ejecutar una función, los argumentos en ésta son evaluados primero, luego se evalúa el cuerpo de la misma con sus parámetros formales (es

decir, las variables que ocurren en *listarg* de la función) *"limitados"* a los parámetros reales (es decir los argumentos evaluados en la aplicación).

- La expresión ***(f e1 ... en)***. Esta evalúa cada una de las ***e1 ... en***, y aplica la función ***f*** a aquellos valores. ***f*** puede ser un ***value-op*** o una ***función definida por el usuario***, si se encuentra la función definida por el usuario, la expresión definiendo su cuerpo se evalúa con las variables de su ***listarg*** asociados con los valores de ***e1 ... en***.

Ejemplos de la aplicación de una FUNCION:

```
1) LEPPOOC>(set x 6)
   6
   LEPPOOC>(inc x)
   7
   LEPPOOC>(dobla 4)
   8

2) LEPPOOC>(set x 6)
   6
   LEPPOOC>(define doblar (x)
                  (begin
                     (print x)
                     (set x (+ x x))
                     (print x)
                  )
            )
   ((x) ((print x) (set x(+ x x)) (print x)).        "Es el resultado.
```

Es importante comprender que la variable "*x*" que ocurre como parámetro formal de la función definida ***doblar*** en el segundo ejemplo, es distinta de la "*x*" que se ha venido usando anteriormente, a partir de su aparición dentro de la expresión ***set***, quien le asigna un valor, como variable global. Una referencia a "*x*" dentro de la función se refiere a los parámetros formales, una referencia fuera de la función se refiere a la variable global en este caso. El lenguaje usa exclusivamente el paso de parámetros llamados ***por-valor*** (no parámetros ***var***); esto significa que una variable global dentro

de una función, nunca puede tener sus valores alterados por una asignación a un parámetro formal dentro de la función (salvo que el parámetro por valor sea el nombre de un objeto, como en C, cuando el parámetro actual es precedido por &), por ejemplo:

```
LEPPOOC> (doblar x)
6                    ☜ Valor del primer (print x) que se
                       encuentra en la función doblar.
12                   ☜ Valor del segundo (print x) que se
                       encuentra en la función doblar.
12                   ☜ Valor de la función doblar
LEPPOOC> x
6                    ☜ Valor de x tomado del Ag
```

LEPPOOC introduce las clases, definiéndolas a través de la expresión ***class***, y después se pueden usar creando objetos de las clases definidas con la expresión ***new***, de este modo se podrán aplicar los métodos de la clase a los objetos previamente creados.

➢ ***(class n-class1 n-class2 (vi1 ... vin)***
(define f1 (v1 ... vn) e1) ...
(define fn (var1 ... varn) en))

Se busca que la superclase ***n-class2*** exista en algún ambiente, si existe se buscan las variables de instancia ***vi1 ... vin*** en los ambientes, si existen se definen las funciones ***f1 ... fn***, creando así la subclase ***n-class1***. Si algo no concuerda se emite el mensaje de error correspondiente.

Ejemplo de CLASS.

```
LEPPOOC> (class persona object (nombre direccion rfc grado)
                (define esc-nom () (print nombre))
                (define esc-dir () (print direccion))
                (define esc-rfc () (print rfc))
                (define esc-grado ()
                (if (= grado 0) (print "#sin grado")
                  (if (= grado 1) (print "#Lic")
                    (if (= grado 2) (print "#Esp")
                      (if (= grado 3) (print "#Maestria")
                            (print "#Doctorado")
```

```
                                                                    )))))
                    (define set-nom (nom) (set nombre nom)))
```

Es el resultado de evaluar la expresión anterior.

```
(OBJECT (NOMBRE DIRECCION RFC GRADO) ESC-NOM (FUNCION
(NIL (PRINT NOMBRE))) ESC-DIR (FUNCION (NIL (PRINT DIRECCION)))
ESC-RFC (FUNCION (NIL (PRINT RFC))) ESC-GRADO (FUNCION (NIL
(IF (= GRADO 0) (PRINT "#sin grado") (IF (= GRADO 1) (PRINT "#Lic")
(IF (= GRADO 2) (PRINT "#Esp") (IF (= GRADO 3) (PRINT "#Maestria")
(PRINT "#Doctorado")))))) SET-NOM (FUNCION ((NOM) (SET NOMBRE
NOM))))

LEPPOOC>
```

En LEPPOOC, el usuario puede crear objetos utilizando la expresión ***new***, misma que cree un objeto vacío cuyo tipo de las variables de instancia de las clases de las que depende el objeto, por omisión es ***número***.

- ***(new n-class)*** Verifica que ***n-class*** (nombre de la clase) exista en uno de los dos ambientes ***Ag*** o ***Ap*** del contexto y entonces crea un objeto vacío y lo imprime como tal sin guardarlo en los ambientes.

Ejemplos de la ejecución de NEW.

```
LEPPOOC> (new persona)
```

Es el resultado

```
(NOMBRE (NUMERO 0) (DIRECCION (NUMERO 0) RFC (NUMERO 0)
GRADO (NUMERO 0))
LEPPOOC>
```

Como podemos observar ***new*** crea el objeto tipo ***persona*** y los valores por defecto de sus variables de instancia son ***cero*** y de tipo ***número***.

Para estar en condiciones de aplicar los métodos de la clase a los objetos creados, estos deben guardarse usando la función ***set***, en la siguiente forma:

```
LEPPOOC> (set obj-persona (new persona))
(NOMBRE (NUMERO 0) (DIRECCION (NUMERO 0) RFC (NUMERO 0)
GRADO (NUMERO 0))
LEPPOOC>obj-persona
(NOMBRE (NUMERO 0) (DIRECCION (NUMERO 0) RFC (NUMERO 0)
GRADO (NUMERO 0))
LEPPOOC>
```

De esta forma le hemos dado un nombre al objeto y ya podemos aplicarle los métodos de la clase del mismo.

- La expresión ***(n-objeto n-met vi1 ... vn).*** Con esta expresión se verifica que exista el objeto ***n-objeto***, para posteriormente verificar la existencia de la definición del método ***n-met***, y evaluar sus variables de instancia ***vi1 ... vin***, finalmente procede a aplicar el método ***n-met***, dando como resultado su valor, si el método es observador no cambia el estado del objeto, pero si no, entonces el estado del objeto ***n-objeto*** cambia, y se refleja en sus variables de instancia afectadas.

  A continuación se muestran algunos ejemplos de la aplicación de métodos en objetos previamente definidos con ***new*** y ***set.***

  Ejemplos de aplicación de métodos a objetos definidos previamente.

  Siguiendo con el ejemplo de la clase ***persona***, tenemos:

```
LEPPOOC> (obj-persona set-nom "#Isabel Portillo")
"#Isabel Portillo"
LEPPOOC> (obj-persona esc-nom)
```

```
"#Isabel Portillo"
"#Isabel Portillo"
LEPPOOC> obj-persona
(NOMBRE (CADENA "#Isabel Portillo") (DIRECCION (NUMERO 0) RFC
(NUMERO 0) GRADO (NUMERO 0))
LEPPOOC>
```

Es importante notar que en el ejemplo de la clase ***persona***, ésta solo tiene un método que permite modificar el estado del objeto (el método ***set-nom***) cambiando el estado de la variable de instancia ***nombre*** del valor 0 y de tipo ***número***, al valor ***"#Isabel Portillo"*** y tipo ***cadena***, dejando los demás intactos.

3.3 COMENTARIOS FINALES

Se cree que el prototipo tiene una sintáxis sencilla y fácil de recordar, como se pudo observar al tener un primer acercamiento con LEPPOOC. Así mismo, el número de expresiones que ofrece es mínimo.

Algunas de las limitaciones de la versión actual de LEPPOOC pueden ser el que no verifique si se alteran o no las palabras claves, no tiene mecanismos para comentarios, as decir, no se puede comentar ningún programa escrito en LEPPOOC, ya que no existe alguna expresión sintáctica que los reconozca.

4 PRINCIPIOS DE DISEÑO CONSIDERADOS EN LEPPOOC

La diversidad de situaciones en las que los lenguajes de programación se utilizan es tan grande, que cualquier lenguaje que trate de acomodarse a ellos sería muy pobre. La mejor solución es identificar una amplia gama de usos, usuarios y computadoras similares y crear lenguajes cuyos diseños sean optimizados para dicha clase. Por supuesto, el número óptimo de lenguajes es un problema de ingeniería difícil de determinar, porque los usos, usuarios y computadoras aparecen y desaparecen conforme se desarrolle la tecnología de las computadoras.

El hecho de que no exista un lenguaje perfecto, no evita tener una estructura de lenguaje perfecta. Esta idea se basa en la observación de que aunque los lenguajes difieren en sus detalles, en su estructura general frecuentemente son similares. Esto significa que todo lenguaje tiene un conjunto de tipos de datos orientados a la aplicación, funciones y operadores que los optimizan para una clase de usos particulares, afectando la clase de usuarios y computadoras para los cuales el lenguaje es apropiado.

Por otro lado, las características y facilidades particulares de usos, usuarios y computadoras, son relativamente independientes. A las facilidades independientes de la aplicación se les llama ***estructura del lenguaje***.

Algunos científicos creen que lenguajes aplicativos como LISP pueden proporcionar tal estructura, otros piensan que los lenguajes orientados a objetos o la programación lógica es una mejor elección. Este aspecto sigue siendo una área de investigación muy importante.

Existen algunos principios basados en la experiencia, que hacen que la balanza de estas facilidades se equilibren en lo cualitativo. Sin embargo estos principios no son independientes, algunos de ellos son corolarios de los otros y algunas veces son contradictorios: si uno se satisface, podría implicar que otro o más no puedan satisfacerse.

Discutiremos el alcance de estos lineamientos de diseño en LEPPOOC, en cuanto a cuáles y en qué proporción se cumplen en nuestro prototipo.

4.1 PRINCIPIOS DE DISEÑO.

En general, ***Mac Lennan***[12] considera ***dieciseis principios de diseño de lenguajes de programación.*** LEPPOOC cumple sólo con algunos de éstos. A continuación se describen y se especifica que sucede con LEPPOOC en cada uno.

1) **Abstracción**. Aún con la mayoría de los lenguajes de programación naturales para una aplicación, existe siempre un margen substancial entre el tipo de datos abstractos, incluyendo las operaciones que caracterizan la solución a un problema; y las estructuras de datos primitivas, considerando las operaciones propias del lenguaje. Una parte substancial de la tarea del programador es diseñar las abstracciones apropiadas para la solución al problema e implantarlos usando los tipos de datos primitivos proporcionados

por el lenguaje de programación, así mismo este puede ocultar la expresión de estas abstracciones. Idealmente, un lenguaje de programación permite estructuras de datos, tipos de datos y operaciones que se definen y mantienen como abstracciones contenidas en sí mismas, de modo que los programadores los puedan usar en otras partes del programa, conociendo sólamente sus propiedades abstractas, sin referirse a los detalles de implantación. Es útil abstraer el código común para situaciones similares ya que construyendo abstracciones se eliminan algunas fuentes de error. En LEPPOOC se tienen las bases para implantar abstracciones, ya que podemos encapsular los datos y sus operaciones. Un ejemplo de abstracción en LEPPOOC, es el uso de funciones para implantar alguna operación que se utilice varias veces en un programa escrito en LEPPOOC, sin que el código tenga que replicarse en los lugares donde se requiera.

2) **Automatización**. Las actividades tediosas y mecánicas propensas a error se automatizan a través de un lenguaje de más alto nivel, como los intérpretes cuya importancia estriba en que fueron los primeros que implantaron la llamada computadora virtual, con su propio conjunto de tipos de datos y operaciones en términos de la computadora real. Una ventaja de la computadora virtual sobre la real, es que es de más alto nivel, esto es, proporciona facilidades más adecuadas a las aplicaciones y elimina muchos detalles de programación; lo cual es un ejemplo del principio de automatización. Al ser LEPPOOC un lenguaje interpretado, se cumple con el principio de automatización. Un ejemplo más concreto viene a ser cuando al despedirse de LEPPOOC con el comando ***(bye)*** se verifica automáticamente que no queden referencias colgantes entre el ambiente persistente y el ambiente global, de ser así cuenta con una opción de recuperación.

3) **Defensa en profundidad**. Consiste en tener una serie de defensas, de modo que si un error no se detecta por un lado (ej. verificación sintáctica), probablemente sea atrapado por otro (ej. verificación de tipos). LEPPOOC cumple con este principio en algunos aspectos, por ejemplo el caracter ***"blanco"*** se usa para separar los ***"átomos"*** (palabras y símbolos). No tiene verificación de tipos, sigue liga dinámica. Pueden crearse referencias colgantes en el ambiente global. Es un lenguaje interpretado, por lo tanto algunas instrucciones pueden estar equivacadas y entonces no detectarse. Un aspecto donde sí cumple este principio es que no quedan referencias colgantes en el ambiente persistente.

4) **Ocultamiento de información**. Consiste en que el lenguaje permita diseñar módulos de modo que:

- El usuario tenga toda la información necesaria para usar el módulo correctamente.
- El implantador tenga toda la información necesaria para implantar el módulo correctamente.

Esto significa que cualquier manipulación de la estructura de datos debe hacerse a través de los procedimientos provistos por el módulo, ya que la representación de la estructura de datos está oculta en el módulo. LEPPOOC proporciona las bases para el ocultamiento de información al manejar los atributos de un objeto como privados al exterior, pero públicos para sus subclases.

5) **Etiquetación**. Consiste en evitar que se tengan secuencias arbitrarias. El usuario no requiere conocer la posición absoluta de un elemento dentro de una lista, en vez de esto, asocia una etiqueta a cada elemento y le permite al elemento ocurrir en otro. Esto significa darle nombre a las posiciones de

3) **Defensa en profundidad**. Consiste en tener una serie de defensas, de modo que si un error no se detecta por un lado (ej. verificación sintáctica), probablemente sea atrapado por otro (ej. verificación de tipos). LEPPOOC cumple con este principio en algunos aspectos, por ejemplo el caracter ***"blanco"*** se usa para separar los ***"átomos"*** (palabras y símbolos). No tiene verificación de tipos, sigue liga dinámica. Pueden crearse referencias colgantes en el ambiente global. Es un lenguaje interpretado, por lo tanto algunas instrucciones pueden estar equivacadas y entonces no detectarse. Un aspecto donde sí cumple este principio es que no quedan referencias colgantes en el ambiente persistente.

4) **Ocultamiento de Información**. Consiste en que el lenguaje permita diseñar módulos de modo que:

   - El usuario tenga toda la información necesaria para usar el módulo correctamente.
   - El implantador tenga toda la información necesaria para implantar el módulo correctamente.

Esto significa que cualquier manipulación de la estructura de datos debe hacerse a través de los procedimientos provistos por el módulo, ya que la representación de la estructura de datos está oculta en el módulo. LEPPOOC proporciona las bases para el ocultamiento de información al manejar los atributos de un objeto como privados al exterior, pero públicos para sus subclases.

5) **Etiquetación**. Consiste en evitar que se tengan secuencias arbitrarias. El usuario no requiere conocer la posición absoluta de un elemento dentro de una lista, en vez de esto, asocia una etiqueta a cada elemento y le permite al elemento ocurrir en otro. Esto significa darle nombre a las posiciones de

memoria tanto a las variables como a las clases, funciones y operaciones con las que cuenta el lenguaje. Al ser LEPPOOC un lenguaje de alto nivel, cumple con este principio. Por ejemplo, LEPPOOC maneja nombres para objetos, variables, funciones y clases.

6) **Costo de localidad**. Significa que el usuario debe pagar sólo por lo que usa, evitando costos que se encuentran distribuidos en las proposiciones del lenguaje. Algunos lenguajes tienen instrucciones generales que conllevan a un tiempo de ejecución mayor sin que, muchas veces, se requiera usar todas las opciones pagando el usuario por tal instrucción a pesar de no explotar toda su potencialidad por no requerirlo. LEPPOOC cuenta con lo indispensable de un lenguaje orientado a objetos, cumpliendo hasta cierto grado con este principio, ya que la versión actual del prototipo no optimiza muchos aspectos en cuanto a eficiencia en ejecución. Tal es el caso de la verificación de referencias colgantes al abandonar la sesión de LEPPOOC.

7) **Interfaz clara**. Todas las interfaces son claras en la sintáxis. Las clases en LEPPOOC hacen que cumpla con este principio, ya que la interfaz para un objeto es precisamente su ***protocolo***: el conjunto de mensajes a los que responde. Por lo tanto, los objetos con el mismo protocolo (interfaz) pueden intercambiarse. No se tienen comentarios.

8) **Ortogonalidad**. Quiere decir que las funciones independientes serán controladas por mecanismos independientes. Este es un corolario del principio de regularidad. La ortogonalidad simplifica un lenguaje ya que si asignamos un número arbitrario a cada operación, sería necesario que el programador recordara ***n*** hechos independientes para ***n*** operaciones. En lugar de esto, reflejamos en la codificación la distinción entre las operaciones directa e inversa. Por lo tanto, la estructura en las operaciones

es reflejada en la estructura de la codificación, reduciendo el número de hechos a recordar. Ortogonal significa ángulo recto. ¿Qué tiene que hacer un ángulo recto con el diseño de lenguajes?. Si tenemos dos ejes significativamente independientes, uno con m posiciones y otro con n posiciones, entonces podemos describir mn diferentes posibilidades aunque sólo tenga que memorizar $m+n$ hechos independientes. Como m y n se incrementan, mn crece mucho más rápido que $m+n$. Entonces el diseño ortogonal resulta ser más importante cuanto más posibilidades se describan.

Cuando existen muchas posibilidades, puede ser ventajoso tener más de dos ejes ortogonales.

Demasiada ortogonalidad puede dañar un lenguaje ya que el lenguaje puede estar agrupado en facilidades que han sido incluidas por simetría pero son poco usadas. Esto es, algunas de las mn posibilidades pueden ser menos usadas o difícil de implantar. Algunas de las mn posibilidades pueden ser aún ilegales, en este caso el programador debe recordarlas como excepciones (violando el principio de regularidad).

LEPPOOC cumple parcialmente con el principio de ortogonalidad, ya que la manera de destruir objetos está en la misma dimensión que hacerlos o no persistentes. Sin embargo, para definir los métodos de una clase la mecánica es la misma que para definir funciones.

9) **Portabilidad.** Significa que el lenguaje debe evitar características o facilidades que son dependientes de una máquina particular o de una clase pequeña de máquinas. LEPPOOC cumple con este principio ya que se utilizó para su programación el lenguaje COMMON LISP[17], con sus instrucciones básicas, y se diseñó en una máquina SUN[15] con sistema

operativo UNIX[15], de tal forma que es fácil de transportar a otro equipo de las mismas características, habiendo muchos que son compatibles con este tipo de sistemas. Además, no hay limitaciones específicas de implantación como longitud de los identificadores, de las listas o del tamaño del ambiente persistente, excepto el tamaño de la memoria de la computadora en la que se ejecute nuestro prototipo.

10)**Preservación de Información**. Los lenguajes permiten la representación de información que los usuarios deben conocer y que el compilador necesitará. En otras palabras, si los usuarios conocen sus requerimientos al nivel más abstracto, no requeriría establecerlos en el nivel más concreto, ya que esto coloca dentro del programa, decisiones de diseño dependientes de la máquina que es mejor dejárselas al compilador. En este caso, LEPPOOC cumple hasta cierto grado con tal principio, ya que cuenta con expresiones que nos ayudan a definir clases y subclases, logrando con esto la abstracción en los requerimientos de los usuarios.

11)**Regularidad**. Significa que, un lenguaje con reglas regulares, sin excepciones, es más fácil de aprender, usar, describir e implantar. LEPPOOC es un lenguaje regular. Por ejemplo solo hay tres tipos de instrucciones, a decir; ***expresiones, comandos*** y ***definiciones***. Toda expresión regresa el valor de evaluar la expresión, toda clase debe tener una superclase *(object)* y toda referencia en el ambiente persistente debe existir. Pero el ***car*** y el ***cdr*** de una lista regresa una lista vacía, esto podría ser extraño pero LEPPOOC está siendo congruente a COMMON LISP.

12)**Seguridad**. Un programa que viole la definición del lenguaje o su propia estructura deseada, evitaría deteclarse. Por ejemplo, suponga que una celda de memoria se libera pero aún está referenciada por otras listas, quedando

ahora algunas referencias colgantes, es decir, apuntadores que no apuntan a una celda con información, de tal manera que cuando la asignación de memoria reutiliza esta celda, puede aún ser accesible desde el programa (a través de las referencias colgantes), con la posibilidad de corromper la estructura de datos. Creemos que LEPPOOC es seguro hasta cierto grado, ya que no permite destruir objetos ni clases que aún tengan referencias asociadas. También se pueden usar algunas palabras reservadas. Se puede redefinir el constructor ***lista*** pero podría crear confusiones.

13)**Simplicidad**. Un lenguaje deberá ser tan flexible como sea posible. De tal manera que exista un número mínimo de conceptos con reglas mínimas para su combinación, ya que si un lenguaje tiene un gran número de constructores diferentes, los programadores pueden no familiarizarse por completo con ellos, lo cual puede conducir al abuso de algunas características y un desuso de otras que pueden ser más elegantes o más eficientes, o ambas, que aquellas que son usadas. Creemos que LEPPOOC es simple ya que cuenta con pocas operaciones, basadas en conceptos mínimos, que se pueden combinar para generar nuevas funcionalidades del lenguaje.

14) **Estructura**. Este principio significa que la estructura estática del programa, corresponderá en forma simple con la estructura dinámica de los cálculos correspondientes. Es decir, será posible visualizar el comportamiento del programa fácilmente desde la forma en como está escrito. LEPPOOC no cumple con este principio, ya que usa regla dinámica como LISP[17], lo que ocasiona que al escribir expresiones como las siguientes, el usuario puede esperar valores distintos del resultado correcto:

```
LEPPOOC>(set y 4)
4
LEPPOOC>(define sumay (x) (+ x y))
((x) (+ x y))
LEPPOOC>(sumay 5)
9
LEPPOOC>...                          ... Muchas instrucciones
...
LEPPOOC>(set y 8)
8
LEPPOOC>...                          ... Muchas instrucciones
...
LEPPOOC>(sumay 3)
11
```

En este punto el usuario podría esperar el valor 7, sobre todo si busca la definición de ***sumay*** y junto está la instrucción anterior que dice que e 'y' se le asignó el valor 4. Sin embargo, ya hubo un cambio en la variable 'y' por lo cual LEPPOOC arrojará un valor distinto al esperado.

15)**Consistencia Sintáctica**. Significa que las coses similares se ven similares y les diferentes se ven diferentes. Es mejor eviter formas sintáctices que puedan ser convertidas en otras formes legeles por un simple error. Por ejemplo, en FORTRAN el uso de "**" pare la exponenciación ha sido criticedo, ya que si por error se escribe sólo un asterisco, se convierte en le estructura legal de la multiplicación con el signo "*". LEPPOOC meneja consistencia sintáctice en sus operaciones en cierto gredo, ya que existen casos donde puede cometerse errores, sobre todo en programas extensos escritos en nuestro intérprete. Por ejemplo, al emplear expresiones como las siguientes, se crean confusiones aún siendo válidas sintácticamente.

```
LEPPOOC>(set lista 3)
LEPPOOC>(lista)
nil
LEPPOOC>lista
3
```

16) **Cero-Uno-Infinito**. Los valores razonables para una estructura deben ser sólamente, ***cero, uno*** e ***infinito***. Por ejemplo, en LEPPOOC las listas pueden tener cero, uno o más elementos, de modo que satisfacen este principio, los elementos de las listas pueden ser listas, de modo que también satisfacen dicho principio. El número de subclases de una clase también desde diseño, en LEPPOOC puede ser cero, uno o cualquier natural.

4.2 Comentarios Finales.

Como vimos, tener cuidado de alcanzar o cubrir los principios de diseño de lenguajes de programación, puede garantizar un lenguaje de programación con calidad y eficiencia, aunque no precisamente con éxito comercial. En LEPPOOC, se consideraron sólo algunos de los principios, tal y como lo mostramos. Sin embargo no hay que perder de vista que este lenguaje no es más que un prototipo y que por lo consiguiente es posible mejorarlo.

5 SEMANTICA DE LEPPOOC

Corresponde ahora hablar de la semántica o significado de las expresiones de nuestro intérprete. El poder y la naturaleza de las notaciones disponibles para describir la sintaxis es relativamente simple; sin embargo, no existe una notación universalmente aceptada para la semántica. Existen varios métodos propuestos para describirla tales como la ***semántica operacional, semántica axiomática*** y la ***semántica denotativa***[7].

En la siguiente sección se dará una breve explicación de cada uno de los métodos formales que describen la semántica de un lenguaje, con la finalidad de que el lector tenga una idea de las mismas, pero para fines de nuestro trabajo emplearemos la semántica denotativa y lenguaje natural para describir lo mejor posible el significado de las proposiciones de LEPPOOC.

5.1 NOTACIONES DE SEMANTICA

5.1.1 Semántica Operacional.

La Semántica Operacional describe el significado de un programa ejecutando sus declaraciones en una máquina ya sea real o simulada. Los cambios que ocurren en la condición o estado de la máquina cuando se ejecuta una proposición dada define el significado de esta operación. Estas operaciones

pueden describirse con denotaciones orientadas a la máquina, o usando técnicas denotativas como teoría de autómatas.

5.1.2 Semántica Axiomática.

La estructura de la semántica axiomática fue definida con la meta principal de desarrollar un método para probar la correctez de programas. Para hacer esto, las reglas matemáticas de inferencia llamadas axiomas son definidas por cada categoría sintáctica del lenguaje. Dadas las restricciones sobre los datos afectados antes de que una proposición sea ejecutada, estas reglas son usadas para deducir los valores de los datos después de la ejecución de la proposición. Este enfoque usa expresiones lógicas que relatan valores de datos, más que el estado completo de una máquina abstracta, como la semántica operacional.

La semántica axiomática está basada en la lógica matemática. Las expresiones lógicas son llamadas predicados o aseveraciones. Una aseveración inmediatamente antes de una proposición describe las restricciones en las variables del programa. Una aseveración inmediatamente después de una proposición describe la nueva restricción sobre aquellas variables. Las aseveraciones son llamadas la precondición y postcondición de la proposición, respectivamente.

5.1.3 Semántica Denotativa.

Esta notación es el método más riguroso para describir el significado de programas. Está basada sólidamente en la lógica y en las matemáticas, por lo que en un principio se le llamó ***semántica matemática***, pero este término se

abandonó pues sugería que otras clases de semánticas (como las explicadas previamente), no son matemáticas.

El concepto fundamental de la semántica denotativa, es definir objetos matemáticos para cada entidad del lenguaje y una función que mapee instancias de esa entidad sobre instancias del objeto matemático. Debido a que los objetos son definidos rigurosamente, representan el significado exacto de sus entidades correspondientes. La dificultad con este método descansa en la creación de los objetos y la función de mapeo. El método es llamado denotativo, porque los objetos denotan el significado de las entidades sintácticas.

5.2 DESCRIPCION SEMANTICA DE LEPPOOC

Se trató de diseñar LEPPOOC de manera que:

a) El intérprete, siempre que es posible regrese valores (para retroalimentación con el usuario).

b) Internamente, LEPPOOC maneje nombres de objetos (como tipo "alias") para:

- Construir objetos compuestos que compartan componentes.
- Para manejar identidad de objetos.
- Para establecer la diferencia entre dos objetos con el mismo valor (o en el mismo estado) pero distintos y dos nombres de objetos del mismo objeto.

- Se permita un contructor de nombres de objetos (la expresión 'quote') y por lo tanto, se puede obligar al intérprete a regresar un nombre de un objeto como en:

```
LEPPOOC> (quote h)
h
```

mas ésta es la única excepción de LEPPOOC.

Por otro lado, hemos diseñado LEPPOOC para que su manejo de expresiones sea lo más uniforme y regular posible. En particular, las expresiones que permiten definir control de flujo u orden de ejecución son simples y por esto podemos describir su semántica formalmente mediante semántica denotativa, como se describe a continuación.

Sea ***S[[E,C]]*** la función significado de las expresiones de LEPPOOC, donde **E** es una expresión de nuestro intérprete y **C** un contexto o ambiente en donde se aplica tal expresión. Por lo tanto ***S[[E,C]]=(valor, nvo-contexto)***, significa que:

$$S : E \times C \rightarrow Valores \times C$$

Los ***Valores*** son un conjunto de la unión disjunta de los dominios (conjuntos) ***cadenas, números, funciones, clases*** y ***objetos.***

El contexto **C** está compuesto por una terna de funciones de nombres a valores u objetos. A las funciones les llamamos ambientes: ***local (Al), global (Ag)*** y ***persistente (Ap)***, mismos que ya han sido explicados brevemente en el capítulo tres y que serán descritos ampliamente en el capítulo seis. Todo esto se reduce a $C = Al \times Ag \times Ap$.

Ahora sí, bajo estas convenciones procedemos a describir la semántica de nuestro lenguaje.

1) Si **E** es un número:

S[[número, C]] = (valor del número, C)

Esto quiere decir que una constante numérica regresa el valor que denota, sin modificar el contexto.

2) Si **E** es una cadena:

S[[cadena, C]] = (valor de la cadena, C)

Lo cual significa que una constante de tipo cadena, regresa el valor que denota, sin modificar el contexto.

3) Si **E** es una variable:

S[[variable, C]] = (contexto[variable], C)

Esto indica que toda variable regresa su valor que se encuentra en el contexto, sin modificarlo.

4) Si **E** es un **set:**

S[[(set var E1), C]] = S[[valor, C'$_{var|valor}$]] donde S[[E1,C]] = (valor, C')

y

C' los evalúa en el Al o Ag

Esto quiere decir que se evalúa la expresión **E1** en el contexto **C** y el valor que resulta se asigna a la variable *var* en el contexto **C'**, resultado de evaluar **E1**. El contexto **C'** se modifica en su ambiente local o ambiente global según donde aparezca el ***set***.

5) Si **E** es un **if:**

$$S[(\text{if } E0\ E1\ E2),\ C]] = \begin{cases} S[[E1, C']] \text{ si } S[[E0, C]] = (k, C') \\ \qquad\qquad\qquad\qquad\qquad\qquad y\ k <> 0 \\ (0, C') \text{ si } S[[E0, C]] = (0, C') \end{cases}$$

Lo cual significa que se evalúa **E0** en el contexto **C** y que el valor que resulta se guarda en el contexto **C'**, resultado de evaluar **E0**:

- Si este valor es diferente de cero, entonces se evalúa **E1** en el contexto **C'**. De tal manera que el contexto se modifica.
- Si el valor es igual a cero, entonces se evalúa **E2** en el contexto **C'**, de tal manera que el contexto **C'** se puede volver a modificar.

Un ejemplo de la evolución de los contextos, lo ilustra el siguiente anidamiento de expresiones "if" y "set":

```
(if (set x y) (set z x) (set z 0))
```

Si antes de ejecutar esto tenemos las expresiones:

```
(set x 1)
(set y 2)
```

pudiera parecer que la condición del ***if(set x y)***, se evalúa en algo distinto de cero, por lo que se ejecuta ***(set z x)***. Al terminar el ***"if"***, el valor de ***z*** es **2**, pues al evaluar ***(set x y)*** modificó el ambiente en que ***(set z x)*** es ejecutado.

6) Si E es un while:

$$S[[(\text{while } E0\ E1), C]] = \begin{cases} (0,C') \text{ si } S[[E0,C]] = (0, C') \\ S[[(\text{begin } E1\ (\text{while } E0, E1), C'), C'']] \\ \quad \text{si } S[[E0, C]] = (k, C') \wedge k \neq 0 \end{cases}$$

Es decir, se evalúa **E0** en el contexto **C**, y valor que resulta se guarda en el contexto **C'**, resultado de evaluar **E0**:

- Si este valor es cero , lo devuelve como resultado del ***while***.
- Si el valor es diferente de cero, evalúa **E1** en el contexto **C'** y el valor que resulta se guarda en el contexto **C''** , y vuelve a evaluar recursivamente el ***while***, pero ahora en el nuevo contexto **C''**.

7) Si E es un **begin:**

$$S[[(\text{begin } E1 \ldots En), C]] = S[[E2,\ldots, En, C']] \text{ donde } S[[E1,C]] = (valor, C')$$

Esto es, se evalúa **E1** en el contexto **C** y el valor que resulta se guarda en el contexto **C'**, resultado de evaluar **E1**, y vuelve a evaluar el ***begin*** a partir de la siguiente expresión **E2**, recursivamente con el nuevo contexto **C'**.

8) Si E es un **persist:**

$$S[[(\text{persist var}), C]] = S[[var, C'_{var|valor}]] \text{ donde } C'_{var|valor} \text{ esta en Ap}$$

Esto es, se localiza el valor de ***var*** en el contexto dentro del ***Ag*** solamente, y que de no encontrarse en el ***Ap*** se pasa del ***Ag*** al ***Ap*** y el resultado de esta actualización se guarda en el contexto **C'** resultado de ésta actualización.

9) Si **E** es un **unpersist:**

$$S[[(unpersist\ var), C]] = S[[var, C'_{var|valor}]] \text{ donde } C'_{var|valor} \text{ esta en Ag}$$

(Esto significa que se localiza el valor de ***var*** en el contexto dentro del ***Ap*** solamente, y se pasa al ***Ag*** y el resultado de esta actualización se guarda en el contexto **C'** resultado de esta actualización).

El resto de los enunciados sintácticos de LEPPOOC requiere de una estructura de dominios más compleja y proporcionar la semántica denotativa de todo LEPPOOC está fuera del alcance de este proyecto. Sin embargo, a continuación explicamos en lenguaje natural de la manera más precisa posible el significado de otros enunciados sintácticos y los detalles de ***set, persist*** y ***unpersist***.

6.3 SEMANTICA DE OTRAS EXPRESIONES DE LEPPOOC DESCRITAS EN LENGUAJE NATURAL.

Comenzaremos por explicar a detalle, el significado de las expresiones ***set, persist*** y ***unpersis***t, dado que ya fueron explicadas previamente con la semántica denotativa.

5.3.1 Semántica de la Expresión (set var exp).

El ***cuadro 1*** nos muestra de manera concentrada el significado de la expresión sintáctica ***(set var exp)***.

Cuadro 1: Semántica de la expresión ***(set variable exp)***.

Llamado del Set / Definición de variable u objeto	En el ambiente local (una función de usuario).	En el ambiente de un método de la clase.	En el ambiente global (programa)
En el ambiente local	Solo cambiará el valor que tiene la variable u objeto-variable- en el ambiente local.	Sólo cambiará el valor que tiene la variable u objeto-variable- en el ambiente local	NO ES FACTIBLE
En el ambiente global	Sólo cambiará el valor que tiene la variable u objeto-variable- en el ambiente global	Sólo cambiará el valor que tiene la variable u objeto-variable- en el ambiente global	Sólo cambiará el valor que tiene la variable u objeto en el ambiente global
En el ambiente persistente	ERROR: variable u objeto en ambiente persistente (solo PERSIST modifica este ambiente)	ERROR: variable u objeto en ambiente persistente (solo PERSIST modifica este ambiente)	ERROR: variable u objeto en ambiente persistente (solo PERSIST modifica este ambiente)
No está definida en ningún ambiente	La variable u objeto -variable- se inserta en el ambiente global	La variable u objeto -variable- se inserta en el ambiente global	La variable u objeto -variable-se inserta en el ambiente global

Como podemos apreciar en el ***cuadro 1*** las intersecciones entre las filas y columnas de la misma, nos muestran el significado del ***set*** dependiendo de las situaciones que se encuentran en la primera fila y columna del cuadro.

Por ejemplo, podemos leer:

- Que el ***set*** se llame desde una función del usuario y que esta se encuentre definida en el ***AI***, significa que solo cambiará el valor que tiene la variable u objeto en el ***AI***.

Del mismo modo podríamos leer las otras.

Para ilustrar el uso de esta expresión presentamos el siguiente código escrito en LEPPOOC.

```
(define suma (n)
   (begin
      (set sum 0)
      (set i 1)
      (set n 5)
      (while (> n i)
         (begin
            (set i (+ i 1))
            (set sum (+ sum i))
         )
      )
      sum
   )
)
```

Para este ejemplo no importa el valor que tenga ***n*** desde afuera, ya que siempre evaluará al valor ***20***, por la modificación de ésta dentro de la función. Este cambio se lleva acabo en el ambiente local y no en los otros ambientes, de modo que al salir de la función, el valor que tenía ***n*** se recupera del ambiente global o persistente, quedando intacto en este punto.

5.3.2 Semántica de la Expresión (persist variable).

El ***cuadro 2*** nos muestra de manera concentrada el significado de la expresión sintáctica ***(persist var).***

Cuadro 2: Semántica de la expresión ***(persist variable)***.

Llamado de persis / Localización o definición de la variable	En el ambiente local (función del usuario)	En el ambiente de un método de una clase	En el ambiente global (nivel del programa)
En ambiente local	ERROR: no se permite hacer persistentes valores que se encuentran en el ambiente local	ERROR: no se premite hacer persistentes valores que se encuentran en el ambiente local	NO ES FACTIBLE
En ambiente global	Se inserta en el ambiente persistente y desaparece del ambiente global	Se inserta en el ambiente persistente y desaparece del ambiente global	Se inserta en el ambiente persistente y desaparece del ambiente global
En ambiente persistente	ERROR: variable u objeto ya existente	ERROR: variable u objeto ya existente	ERROR: variable u objeto ya existente
No está definida en ambiente alguno	ERROR: variable u objeto no existente	ERROR: variable u objeto no existente	ERROR: variable u objeto no existe

Al igual que con el ***set,*** podemos apreciar en el ***cuadro 2*** que las intersecciones entre la filas y columnas de la misma, nos muestran el significado del ***persist*** dependiendo de las situaciones que se encuentran en la primera fila y columna del cuadro.

Si observamos con cuidado el ***cuadro 2***, podemos ver que se reduce al significado siguiente:

1) Si ***persist*** se llamó en el ambiente local (función del usuario), o de un método de una clase, o del ambiente global (nivel del programa) y la variable ***var*** se encontró en el ambiente global, entonces se inserta la variable ***var*** en el ambiente persistente y desaparece del ambiente global.

2) En cualquier otro caso se emiten los mensajes de error que se encuentran en las intersecciones del cuadro, exceptuando el caso en el que no es factible que suceda.

5.3.3 Semántica de la Expresión (unpersist variable).

El ***cuadro 3*** nos muestra de manera concentrada el significado de la expresión sintáctica ***(unpersist var)***.

Cuadro 3: Semántica de la expresión ***(unpersist variable)***.

Definición de la variable \ Llamado a unpersist	En el ambiente local (función del usuario)	En el ambiente de un método de una clase	En el ambiente global (nivel del programa)
En el ambiente local	ERROR "el nombre de variable u objeto -variable- se encuentra en ambiente local"	ERROR "el nombre de variable u objeto -variable- se encuentra en el ambiente local"	NO ES FACTIBLE
En el ambiente global	ERROR "el nombre de variable u objeto se encuentra en ambiente global"	ERROR "el nombre de variable u objeto se encuentra en ambiente global"	ERROR "el nombre de variable u objeto se encuentra en ambiente global"
En el ambiente persistente	Se inserta en el ambiente global y desaparece del ambiente persistente	Se inserta en el ambiente global y desaparece del ambiente persistente	Se inserta en el ambiente global y desaparece del ambiente persistente
No está definida en ningún ambiente	ERROR "variable u objeto *-variable-* indefinido"	ERROR "variable u objeto *-variable-* indefinido"	ERROR "variable u objeto *-variable-* indefinido"

Al igual que con las expresiones anteriores, podemos apreciar en el ***cuadro 3*** que las intersecciones entre la filas y columnas de la misma, nos muestran el significado del ***unpersist*** dependiendo de las situaciones que se encuentran en la primera fila y columna del cuadro.

Si observamos con cuidado el ***cuadro 3***, podemos ver que el significado del ***unpersist*** se reduce a lo siguiente:

1) Si ***unpersist*** se llamó en el ambiente local (función del usuario), o de un método de una clase, o del ambiente global (nivel del programa) y la variable ***var*** se encontró en el ambiente persistente, entonces se inserta la variable ***var*** en el ambiente global y desaparece del ambiente persistente.

2) En cualquier otro caso, se emiten los mensajes de error que se encuentran en las intersecciones del cuadro, exceptuando el caso en el que no es factible que suceda.

5.3.4 Semántica de la Expresión (define n-function arglist exp).

El ***cuadro 4*** nos muestra de manera concentrada el significado de la expresión sintáctica ***(define n-function arglist exp).***

Cuadro 4: Semántica de la expresión ***(define n-function arglist exp).***

Llamado del define / Definición de n-function	Desde el ambiente global	Desde un método de una clase
En el ambiente local	NO ES FACTIBLE	NO ES FACTIBLE
En el ambiente global	MENSAJE: "Nombre de la función ***n-function*** ya existe"	NO ES FACTIBLE
No está en ningún ambiente	la función ***n-function*** se inserta en el ambiente global	La función ***n-function*** se integra a la definición de la clase

Al igual que con las expresiones anteriores, podemos apreciar en el ***cuadro 4***, que las intersecciones entre las filas y columnas de la misma, nos muestran el significado del ***define*** dependiendo de las situaciones que se encuentran en la primera fila y columna del cuadro.

Si observamos con cuidado el ***cuadro 4***, podemos ver que el significado del ***define*** se reduce a lo siguiente:

1) Si ***define*** se llama desde el ambiente global y el nombre ***n-function*** de la función no se encuentra definido en los ambientes, entonces la función se inserta en el ambiente global.

2) Si ***define*** se llama desde un método de una clase y el nombre ***n-function*** de la función no se encuentra definido en los ambientes, entonces la función se integra a la definición de la clase.

3) Si ***define*** se llama desde el ambiente global y el nombre de la función ***n-function*** se encuentra en el ambiente global, nivel del programa, es un error ya que el nombre de la función ya existe, por lo que se emite tal mensaje de error.

4) Cualquier otro caso no es factible que suceda.

Es importante dejar claro la diferencia que existe entre enunciados como los siguientes:

(set doblar (lista (quote ((x) (+ x x)))) *aquí el tipo de doblar es ALIAS*

y **(define doblar (x) (+ x x))** *aquí el tipo de doblar es FUNCION*

En ambos casos el resultado que LEPPOOC emite es la expresión ***((x) (+ x x))***, sin embargo, en el primer caso con el ***set***, internamente LEPPOOC guarda en los ambientes la pareja: ***(doblar (alias ((x) (+ x x))))***, donde el tipo de ***"doblar"*** no es una función, como lo sería al definir ***"doblar"*** con el define: ***(doblar (funcion ((x) (+ x x))))***, por lo que al aplicar ***"doblar"*** en el primer caso, el interprete nos contestaría el mensaje de error 'EXPRESION-INVALIDA', mientras que con el otro, la aplicaría.

Otro ejemplo de expresiones que arrojan resultados *"iguales"* pero internamente se representan de manera diferente, sería *"definir"* funciones a través del uso combinado de ***set, lista*** y ***quote***.

Ejemplo 1.

```
LEPPOOC> (set inc
           (lista
              (lista (quote x))
              (lista (quote +) (quote x) (quote 1))
           )
         )
((X) (+ X 1))                    ☜ es el resultado.
LEPPOOC>inc
((X) (+ X 1))                    ☜ es el resultado.
LEPPOOC> (inc 1)
EXPRESION-INVALIDA               ☜ es el resultado.
LEPPOOC>
```

Ejemplo 2.

```
LEPPOOC> (define inc (x) (+ x 1))
((X) (+ X 1))                    ☜ es el resultado
LEPPOOC>inc
((X) (+ X 1))                    ☜ es el resultado
LEPPOOC> (inc 1)
2                                ☜ es el resultado
LEPPOOC>
```

En ambos casos el valor que arroja LEPPOOC es el mismo **((X) (+ X 1))**, sin embargo el tipo de ***inc*** en el primer ejemplo es ***lista*** , mientras que en el segundo es ***funcion***, razón por la cual puede aplicar la función en el último ejemplo, en tanto que en el primer ejemplo LEPPOOC detecta un problema y regresa el mensaje "EXPRESION-INVALIDA".

El valor interno que LEPPOOC guarda para ***inc*** en el primer ejemplo es:

```
(inc (lista ((lista ((alias x)))
   (lista ((alias +) (alias x)
   (alias 1))))))
```

mientras que para el segundo ejemplo sería:

```
(inc (funcion ((x) (+ x 1))))
```

completamente distintos.

El tratamiento de definición de funciones y definiciones de clases, es distinto al nombramiento de valores. En este sentido, LEPPOOC difiere de la programación funcional en que las funciones son 'ciudadanos de segunda clase'. Se tomó esta decisión por uniformidad con los lenguajes orientados a objetos como C++.

5.3.5 Semántica de la Expresión (class n-class1 n-class2 inst-var methoddef⁺).

El ***cuadro 5*** nos muestra de manera concentrada el significado de la expresión sintáctica ***(class n-class1 n- class2 inst-var methoddef⁺).***

Cuadro 5: Semántica de la expresión ***(class n-class1 n-class2 inst-var methoddef)***.

Llamado de class / Definición de la superclase n-class2	Desde el ambiente ***global*** y ***n-class1*** está en el ***ambiente global***	Desde el ambiente ***global*** y ***n-class1*** existe en el ***ambiente persistente***	Desde el ambiente ***global*** y n-cass1 no está definida en ***ningún ambiente***
En el ambiente ***global***	MENSAJE: "Clase con nombre repetido"	MENSAJE: "Clase con el nombre repetido"	Se crea la definición de la clase n-class1
En el ambiente ***persistente***	MENSAJE: "Clase con nombre repetido"	MENSAJE: "Clase con nombre repetido"	Se crea la definición de la clase ***n-class1***
No está definida en ***ambiente alguno***	ERROR: La superclase ***n-class2*** no está definida	ERROR:La superclase ***n-class2*** no está definida	ERROR:La superclase ***n-class2*** no está definida

Al igual que con las expresiones anteriores, podemos apreciar en el ***cuadro 5***, que las intersecciones entre la filas y columnas del mismo, nos muestran el significado de la expresión ***class*** dependiendo de las situaciones que se encuentran en la primera fila y columna del cuadro.

Si observamos con cuidado el ***cuadro 5***, podemos ver que el significado del ***class*** se reduce a lo siguiente:

1) Si ***class*** se llama desde el ambiente global y ***n-class1*** no está definida en ningún ambiente y además la clase ***n-class2*** está definida en el ambiente global o persistente, entonces se crea la definición de la clase ***n-class1***.

2) Si ***class*** se llama desde el ambiente global y ***n-class1*** está en el ***Ag***, o en el ***Ap*** y la superclase ***n-class2*** está definida en el ***Ag*** o en el ***Ap***, entonces significa que esta clase ya existe y no se puede redefinir, por lo que se envía el mensaje correspondiente.

3) Si ***class*** se llama desde el ambiente global y ***n-class1*** está en el ***Ag***, o en el ***Ap*** y la superclase ***n-class2*** no está definida en ningún ambiente, significa que ha ocurrido un error, ya que se está intentando definir una subclase de un tipo de clase no existente, por lo que se envía el mensaje correspondiente.

Discutiremos el uso de la expresión ***class*** respecto del uso combinado de ***set*** y ***quote,*** a través de los siguientes ejemplos.

Ejemplo 1. Definición de una "clase" con ***set*** y ***quote***.

```
LEPPOOC>(set figura
            (quote
               (OBJECT (PI R)
                  VPI (FUNCION (NIL (SET PI 3.1416)))
                  VR (FUNCION ((VALOR) (SET R VALOR )))
               )
            )
         )

OBJECT (PI R) VPI (FUNCION (NIL (SET PI 3.1416))) VR (FUNCION
((VALOR) (SET R VALOR ))))
LEPPOOC>(new figura)
ERROR-EN-NEW-CLASE-INDEFINIDA.
LEPPOOC>
```

Ejemplo 2. Definición de una "clase" con ***class***.

```
LEPPOOC>(class figura object
            (pi r)
            (define vpi () (set pi 3.1416))
            (define vr (valor) (set r valor))
        )

OBJECT (PI R) VPI (FUNCION (NIL (SET PI 3.1416))) VR (FUNCION
((VALOR) (SET R VALOR ))))
LEPPOOC>(new figura)
(PI (NUMERO 0) R (NUMERO 0))
LEPPOOC>
```

Como podemos observar en los ejemplos, en ambos casos el resultado que LEPPOOC arroja es la expresión:

```
OBJECT (PI R) VPI (FUNCION (NIL (SET PI 3.1416))) VR (FUNCION
((VALOR) (SET R VALOR ))))
```

Que el usuario de LEPPOOC interpretaría como la definición de una clase (FIGURA, para este ejemplo)

Esto es en apariencia, ya que internamente su representación, es diferente para ambos ejemplos.

En el primer caso LEPPOOC guarda en los ambientes el siguiente valor

```
(FIGURA (ALIAS (OBJECT (PI R) VPI (FUNCION (NIL (SET PI 3.1416)))
VR (FUNCION ((VALOR) (SET R VALOR ))))))
```

Donde el tipo de FIGURA no es una clase sino un ALIAS, como lo sería para el segundo caso; ya que en lugar del ALIAS tendría el tipo CLASE. Por esta razón al evaluar (new figura) para el primer ejemplo, LEPPOOC detecta que no hay una clase FIGURA definida y emite el mensaje de error "ERROR- EN-NEW-CLASE-INDEFINIDA". Sin embargo para el caso del ejemplo 2 sí la encuentra y entonces regresa el objeto creado:

```
(PI (NUMERO 0) R (NUMERO 0))
```

Ahora discutiremos el uso de la expresión class respecto del uso combinado de ***set***, ***quote*** y ***lista***, a través de los ejemplos ya mencionados y del ejemplo 3.

Ejemplo 3. Definición de una clase con set, quote y lista.

```
LEPPOOC>(set figura
                (lista
                   (quote object)
                   (lista (quote pi) (quote r))
                   (quote vpi)
                   (lista (quote funcion)
                      (lista (quote nil)
                         (lista (quote set)  (quote pi) quote 3.1416))
                      )
                   )
                   (quote vr)
                   (lista (quotefuncion)
                      (lista
                         (lista (quote valor))
                            (lista (quote set) (quote r) (quote valor))
                         )
                      )
                   )
                )
             )

(OBJECT (PI R) VPI(FUNCION) (NIL (SET PI 3.1416))) VR
(FUNCION ((VALOR) (SET R VALOR))))
LEPPOOC>figura
OBJECT (PI R VPI (FUNCION(NIL (SET PI 3.1416))) VR
(FUNCION ((VALOR (SET R VALOR ))))
LEPPOOC> (new figura)
ERROR-EN-NEW-CLASE-INDEFINIDA
LEPPOOC>
```

En este caso, el valor interno de la clase "figura" se representa como:

```
(FIGURA (LISTA ((LISTA ((ALIAS PI) (ALIAS R))) (LISTA ((ALIAS VPI)
(ALIAS NIL) ((ALIAS SET) (ALIAS PI) (ALIAS 3.14.16))))) (LISTA ((ALIAS
VR) (LISTA ((ALIAS VALOR))) (LISTA ((ALIAS SET) (ALIAS R) (ALIAS
VALOR ))))))))
```

que es un valor muy diferente del que internamente representa la definición de una clase con la expresión ***class***, para el ejemplo 2. Sin embargo el usuario puede confundirse con estas dos formas de construir el valor que arroja LEPPOOC.

A través de estos ejemplos, podemos darnos cuenta que haciéndole algunas modificaciones a LEPPOOC para que interprete estas "definiciones" como funciones o clases según sea el caso, podemos tener un lenguaje más poderoso, con menos instrucciones que permita incluir evolución del esquema, una de las características de base de datos discutida en este trabajo, entre otras.

5.3.6 Semántica de la Expresión (new n-class).

El ***cuadro 6*** nos muestra de manera concentrada el significado de la expresión sintáctica ***(new n-class).***

Cuadro 6: Semántica de la expresión ***(new n-class).***

Llamado de new / Definición de n-class	Desde el ambiente local (función del usuario)	Desde el ambiente global (nivel del programa)	Desde un método de clase
En el ambiente local	NO ES FACTIBLE	NO ES FACTIBLE	NO ES FACTIBLE
En el ambiente global	Se crea un nuevo objeto	Se crea un nuevo objeto	Se crea un nuevo objeto
En el ambiente persistente	Se crea un nuevo objeto	Se crea un nuevo objeto	Se crea un nuevo objeto
No está en ambiente alguno	ERROR: La clase n-class no está definida	ERROR: La clase n-class no está definida	ERROR: La clase n-class no está definida

Al igual que con las expresiones anteriores, podemos apreciar en el ***cuadro 6***, que las intersecciones entre la filas y columnas de la misma, nos muestran el significado del ***new*** dependiendo de las situaciones que se encuentran en la primera fila y columna del cuadro.

Si observamos con cuidado el ***cuadro 6***, podemos ver que el significado del ***new*** se reduce a lo siguiente:

1) Si ***new*** se llama desde el ambiente local o global o en un método de una clase y si ***n-class*** está en el ambiente global o persistente, entonces se crea un nuevo objeto vacío.

2) Si ***new*** se llama desde el ambiente local o global o en un método de una clase y si ***n-class*** no está definida en ningún ambiente, significa que no se pueden crear objetos de clases indefinidas, ya que todo objeto siempre pertenece al menos a una clase previamente definida, por lo que, en este caso, se envía un mensaje de error.

3) No es factible que ***new*** ocurra en las situaciones antes mencionadas y que el nombre de la clase ***n-class*** esté en el ambiente local.

5.3.7 Semántica de la Expresión (set-shallow nombre variable)

El ***cuadro 7*** nos muestra de manera concentrada el significado de la expresión sintáctica (set-shallow nombre variable).

Cuadro 7: Semántica de la expresión ***(set-shallow nombre variable).***

Llamado de Set-shallow / Definición de nombre	En el ambiente local (una función del usuario) y *variable* se encuentra definida en cualquier ambiente	En el ambiente de un método de la clase, y *variable* se encuentra definida en cualquier ambiente	En el ambiente global (nivel del programa) y *variable* se encuentra definida en cualquier ambiente	Desde cualquier ambiente y *variable* no se encuentra definida en ningún ambiente
En el ambiente local	Solo cambia el valor de la *variable* u objeto *nombre*, dejándolo en el local como (*alias variable*). Este sería su valor. Creando así un apuntador al objeto *variable*	Solo cambia el valor de la *variable* u objeto *nombre*, dejándolo en el local como (*alias variable*). Este sería su valor. Creando así un apuntador al objeto *variable*	Solo cambia el valor de la *variable* u objeto *nombre*, dejándolo en el local como (*alias variable*). Este sería su valor. Creando así un apuntador al objeto *variable*	ERROR: No está definido el objeto
En el ambiente global	Solo cambia el valor de la *variable* u objeto *nombre*, dejándolo en el global como (*alias variable*). Creando así un apuntador al objeto *variable*	Solo cambia el valor de la *variable* u objeto *nombre*, dejándolo en el global como (*alias variable*). Creando así un apuntador al objeto *variable*	Solo cambia el valor de la *variable* u objeto *nombre*, dejándolo en el global como (*alias variable*). Creando así un apuntador al objeto *variable*	ERROR: No está definido el objeto
En el ambiente persistente	ERROR: No se puede modificar el ambiente persistente con Set-*shallow*	ERROR: No se puede modificar el ambiente persistente con Set-*shallow*	ERROR: No se puede modificar el ambiente persistente con Set-*shallow*	ERROR: No está definido el objeto
No definido en ambiente alguno	El objeto se crea en ambiente global *pero* su valor es(*alias variable*), que sería un apuntador al objeto *variable*	El objeto se crea en ambiente global pero su valor es (*alias variable*), que sería un apuntador al objeto *variable*	El objeto se crea en ambiente global pero su valor es (*alias variable*), que sería un apuntador al objeto *variable*	ERROR: No está definido el objeto

Al igual que con las expresiones anteriores, podemos apreciar en el ***cuadro 7***, que las intersecciones entre las filas y columnas del mismo, nos muestran el significado del ***set-shallow*** dependiendo de las situaciones que se encuentran en la primera fila y columna del cuadro.

Si observamos con cuidado el ***cuadro 7***, podemos ver que el significado de la expresión ***set-shallow*** se reduce a los siguientes casos:

CASO 1. Si ***set-shallow*** se llamó en el ambiente local o del ambiente de un método de la clase, o del ambiente global y la variable se encuentra definida en cualquier ambiente, y si el nombre se encuentra definido en el ambiente local, entonces LEPPOOC procede a cambiar sólo el valor de la variable u objeto nombre, dejándolo en el ambiente local como ***(alias variable)***. Lo mismo sucede si el ***nombre*** se encuentra definido en el ambiente global, con la diferencia de que lo deja en el ambiente global.

CASO 2. Si ***set-shallow*** se llamó en el ambiente local o del ambiente de un método de la clase, o del ambiente global y la variable se encuentra definida en cualquier ambiente, y si el ***nombre*** se encuentra definido en el ambiente persistente, entonces LEPPOOC emite un mensaje de error, dado que no se puede modificar el ambiente persistente por ningún otro medio que no sea a través de la expresión ***persist*** como ya se ha comentado previamente en este texto.

CASO 3. Si ***set-shallow*** se llamó en el ambiente local o del ambiente de un método de la clase, o del ambiente global y la variable se encuentra definida en cualquier ambiente, y si el nombre no se encuentra definido en ningún ambiente, entonces LEPPOOC crea un nombre

de objeto en el ambiente global, siendo su valor ***(alias variable)*** un apuntador al objeto variable.

CASO 4. Si ***set-shallow*** se llamó desde cualquier ambiente y la variable no se encuentra definida en ningún ambiente, esté o no definido el ***nombre*** en algún ambiente, LEPPOOC emite un mensaje de error, que especifica que no está definido el objeto al que se desea apuntar. Recordemos que es a través de éste que se logra parte de la ***identidad de objetos.***

5.3.8 Semántica de la Expresión (quote nombre).

La expresión ***quote*** es tan simple en su aplicación que su significado se reduce a crear un apuntador de tipo nombre, en la forma ***(alias nombre)***, pegándolo a los ambientes sin guardarlo dentro de ellos, con la sola finalidad de mostrar el efecto al usuario. Si se deseara guardar tendría que asociarse a una expresión ***set***.

Como hemos visto en la sección 5.3.7 y en esta, las instrucciones ***set-shallow*** y ***quote*** permiten estructurar objetos compuestos, cuyas componentes pueden compartirse por varios objetos. La idea que el objeto compuesto tiene un subobjeto componente, pero esta relación no es directa. Un objeto componente es parte de un objeto compuesto, cuando uno de los atributos del objeto compuesto es un ***nombre*** pera un objeto componente

5.3.9 Semántica de la Expresión (persist-class).

El comando ***(persist-class)*** se encarga de verificar si existen referencias colgantes, para lo cual revisa los ambientes en la siguiente forma:

1) Si no hay elementos en el ***ambp*** (variable interna del programa que representa el ambiente persistente) significa que no hay referencias colgantes y regresa a ***eval-leppooc*** saliendo del mismo con el valor ***bye***.

2) Si existe algún objeto en el ambiente persistente y su clase no se encuentra en éste, procede a llamar a ***ev-persist***, para hacerla persistente y recursivamente vuelve a entrar en ***persist-class*** hasta cumplir con el paso 1

3) Si existen objetos en el ambiente persistente y sus partes son ***"aliases"*** de otros que no se encuentran en este ambiente, los vuelve persistentes utilizando ***ev-persist***. Y de manera recursiva vuelve a llamar a ***persist-class*** para cada parte, hasta revisar todos los objetos del ambiente persistente, cumpliendo con el paso 1.

Cabe aclarar que, se utiliza un ambiente auxiliar que contine los objetos originales más los nuevos que eran referencias colgantes, y otro que nos sirve para examinar cada uno de los objetos que se encuentran en el ambiente persistente original, de tal manera que se van eliminando de él los que ya fueron revisados. Esto para detener la recursividad, lo cual indicaría que ya no existen referencias colgantes.

5.3.10 Semántica del llamado de los métodos de un objeto.

En general, la forma como se llaman los métodos de un objeto en LEPPOOC es como sigue:

(nombre-del-objeto nombre-del-método lista-de-argumentos-del-método)

por ejemplo

```
(obj-dircomp v-calle "#Benito juárez")
```

Por lo tanto, para enviar un mensaje a un objeto componente de un objeto compuesto, la clase del objeto compuesto debe tener un selector del objeto compuesto. Por ejemplo, la clase de objetos compuestos **"persona"** va a tener un atributo **"dirección"** que será una componente. El atributo **"dirección"** tendrá valores en la clase de objetos "direc-comp".

Para seleccionar el atributo **"dirección"** de un objeto compuesto, la clase persona tiene un método selector. Este método selector es "esc-direc" que no tiene argumentos, simplemente regresa el valor del atributo. Es decir, si se escribe

```
(obj-persona esc-direc)
```

regresa el valor **(una replica)** del objeto **(la componente)**.

Recuérdese que el intérprete siempre presenta al usuario un valor y, para hacer esto, en algunos casos (como el ejemplo anterior) se ve obligado a replicar el objeto.

Sin embargo, volviendo a como se envía un mensaje al objeto componente, esto se hace con el selector del objeto componente como primer elemento del formato (A). Así, en nuestro ejemplo, esto se hace de la siguiente manera:

```
((obj-persona esc-direc) v-calle "#Lucio")
```

Por la naturaleza de (A), en este caso

```
(obj-persona esc-direc)
```

se evalúa al nombre del objeto componente.

En el apéndice C se muestra otro programa (obj-comp) en LEPPOOC, que muestra el uso del llamado de los métodos de un objeto compuesto.

5.4 EJEMPLOS DE LEPPOOC

En esta sección se presentan varios ejemplos de programas escritos en nuestro intérprete, que muestran sus características más relevantes respecto al modelo orientado a objetos en el que se basa, con la finalidad de proporcionar un panorama más amplio de su uso.

Se consideran las siguientes convenciones para fines de la explicación de cada uno de nuestros ejemplos.

1) Para todos los ejemplos se utilizará una notación gráfica para representar las clases y sus objetos mediante el símbolo que se muestra en la figura 5.1.

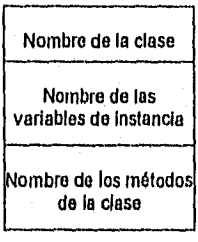


Figura 5.1 Símbolo que representa clases y objetos.

2) Además no se debe olvidar, que LEPPOOC cuenta con una clase base definida por él, llamada OBJECT, que no tiene ni variables de instancia, ni métodos, solo sirve para darle significado a las creadas por el usuario.

3) Para fines prácticos, vamos a suponer que todos nuestros programas se introdujeron previamente a la máquina a través de un editor de texto.

4) Hemos incluido una numeración y explicación en prosa, empezando para esto con (☺) en los resultados que arroja LEPPOOC, al interpretar cada una de las expresiones y comandos que introduce el usuario, a través del archivo fuente previamente creado como lo marca la convención 3. Para relacionar esto con el programa fuente también se han enumurado las expresiones del mismo.

◊ Ejemplo 1. Programa CIRCULO.

Suponga qua tenemos las clases que se muestran en la Figura 5.2.

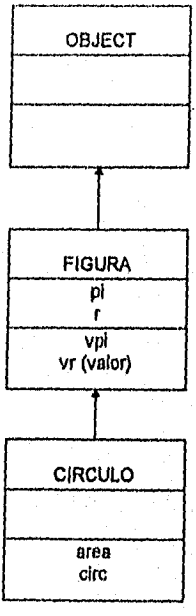


Figura 5.2 Clases, Figura y Circulo.

Como podemos observar, la clase FIGURA es subclase del tipo OBJECT, con variables de instancia ***pi*** y ***r***, que nos representan los valores matemáticos ***pi*** y ***radio*** de las figuras círculo, circunferencia o esfera; así mismo cuenta con el método ***vpi*** que lo único que hará es asignar el valor constante del valor ***pi***, 3.1416, y el método ***vr*** que cuenta con un argumento, que será un valor variable para el ***radio***.

Por otro lado, tenemos la clase CIRCULO que viene a ser una subclase de la clase FIGURA, mostrando así de un modo sencillo la jerarquía de herencia de clases, ya que en el ejemplo gráfico de la Figura 5.2, un objeto de clase CIRCULO heredará las variables de instancia y los métodos de FIGURA.

A continuación mostramos el código de la definición de estas clases y creamos objetos de las mismas, manipulando sus métodos para estudiar el comportamiento de cada objeto.

Programa CIRCULO, escrito en LEPPOOC.

```
(2)  (class figura object
     (pi r)
     (define vpi () (set pi 3.1416))
     (define vr (valor (set r valor))
     )
(3)  (class circulo figura
     ()
     (define área ()
     (begin (set a (* pi (* r r)))
     (print "área")
     (print a)) )
     (define circ ()
     (begin (set c (* 2 (* pi r)))
     (print "circunferencia")
     (print c)) )
     )
(4)  (set mi-fig (new figura))
(5)  (set mi-circulo (new circulo))
(6)  mi-circulo
(7)  (mi-circulo vpi)
(8)  (mi-circulo vr 4)
(9)  (mi-circulo area)
(10) mi-circulo
(11) (persist mi-circulo)
(12) (persist mi-fig
```

Al ejecutar este programa en nuestro intérprete sucede lo siguiente:

(1) **LEPPOOC> (read "circulo")**

- Cuando introducimos esta expresión a LEPPOOC, este busca que exista un archivo con el nombre ***"circulo"***, y entonces interpreta expresión por expresión leyéndola desde este archivo.

(2) (OBJECT (PI R) VPI (FUNCION (NIL (SET PI 3.1416))) VR (FUNCION ((VALOR) (SET R VALOR))))

- Por la definición de la clase ***figura***.

(3) (FIGURA NIL AREA (FUNCION (NIL BEGIN (SET A (* PI (* R R))) (PRINT "#area") (PRINT A)))) CIRC (FUNCION (NIL (BEGIN SET C (* 2 (* PI R))) (PRINT "#circunferencia") (PRINT C)))))

- Por la definición de la clase ***circulo***

(4) (PI (NUMERO 0) R (NUMERO 0))

- Por el (set mi-figura (***new*** figura))

(5) (PI (NUMERO 0) R (NUMERO 0))

- Por el (set ***mi-circulo*** (***new*** circulo))

(6) (PI (NUMERO 0) R (NUMERO 0))

- Por la expresión ***mi-circulo***, que al interpretarla LEPPOOC nos devuelve su valor

(7) 3.1416

Por haber enviado el mensaje ***vpi*** del objeto ***mi-circulo*** a la clase CIRCULO y como no lo encontró ahí, lo envió a la clase FIGURA, quien tiene definido el método en cuestión.

(8) 4

Por haber enviado el mensaje ***vr*** con argumento 4 del objeto ***mi-circulo*** a la clase CIRCULO y como no lo encontró lo remitió a la clase FIGURA, quien tiene definido el método en cuestión.

(9) "#area"
50.2656
50.2656

Por haber enviado el mensaje ***area*** del objeto ***mi-circulo*** a la clase ***circulo***, misma que tiene definido el mensaje ***area***, quien escribe ***"#area"*** e imprime el valor del área 50.2656, y como la última expresión del ***begin*** de este método es el ***print***, vuelve a escribir como resultado el valor 50.2656.

(10) (PI (NUMERO 3.1416) R (NUMERO 4))

Por la expresión ***mi-circulo***, misma que LEPPOOC interpreta enviando el valor de este objeto.

(11) (PI (NUMERO 3.1416) R (NUMERO 4))

Por la expresión ***(persist mi-circulo)***, que hace persistente al objeto ***mi-circulo***, y devuelve su valor.

(12) (PI (NUMERO 0) R (NUMERO 0))

- Por la expresión *(persist mi-fig)*, que hace persistente al objeto ***mi-fig***, y devuelve su valor.

Finalmente, al salir de LEPPOOC con el comando ***(bye)*** se guarda el ambiente persistente en el archivo llamado ***"arch-pers"***.

◊ Ejemplo 2. Programa SETSHALLOW.

Haciendo uso de las definiciones del ejemplo anterior y dado que quedaron almacenadas en el ambiente persistente, presentaremos ahora el uso de la expresión ***set-shallow***, que nos permite manejar el concepto de identidad en el modelo orientado a objetos.

Sea el programa "setshallow"

```
(2)  (set-shallow nvo-circ mi-circulo)
(3)  nvo-cir
(4)  mi-circulo
(5)  (nvo-cir vr 7)
(6)  mi-circulo
(7)  nvo-circ
```

Al ejecutar este programa en nuestro intérprete sucede lo siguiente:

(1) LEPPOOC> (read "setshallow")

- LEPPOOC busca que exista un archivo con el nombre ***setshallow***, y entonces interpreta expresión por expresión leyéndolas desde este archivo.

(2) MI-CIRCULO

- Por la expresión ***set-shallow*** que al interpretarla LEPPOOC, genera un "apuntador" al objeto ***mi-circulo***.

(3) MI-CIRCULO

- Por la expresión ***nvo-circ***, que al interpretarla LEPPOOC devuelve su valor, que en este caso es el apuntador al objeto ***mi-circulo***.

(4) (PI (NUMERO 3.1416) R (NUMERO 4))

- Por la expresión ***mi-circulo***, que al interpretarla LEPPOOC devuelve su valor, que en este caso es el objeto ***mi-circulo***, con el valor del radio igual a 4.

(5) 7

- Por haber enviado el mensaje ***vr*** con argumento 7 del objeto ***nvo-circ***, quien como es un "apuntador" al objeto real, ***mi-circulo***, hace referencia a él y este lo envía a la clase CIRCULO, que es a la que pertenece y como no lo encuentra ahí lo remite a la clase FIGURA, quien tiene definido el método en cuestión, devolviendo el nuevo valor para el ***radio***.

(6) (PI (NUMERO 3.1416) R (NUMERO 7))

Por la expresión ***mi-circulo***, que al interpretarla LEPPOOC devuelve su valor, que en este caso es el objeto ***mi-circulo***, con el valor del radio actualizado a 7, por la expresión previa.

(7) MI-CIRCULO

Por la expresión ***nvo-clrc***, que al interpretarla LEPPOOC devuelve su valor, que en este caso es el apuntador al objeto ***mi-circulo***.

◊ Ejemplo 3. Uso del UNPERSIST, BYE y PERSIST-CLASS.

Nuevamente haciendo uso de los resultados de los ejemplos anteriores vamos a dar un ejemplo que nos permita ver el uso del ***unpersist*** y ***persist-class***. Por ser solo tres expresiones utilizaremos LEPPOOC, interactivamente:

LEPPOOC> (unpersist figura)

(OBJECT (PI R) VPI (FUNCION (NIL (SET PI 3.1416))) VR (FUNCION ((VALOR) (SET R VALOR))))

LEPPOOC interpreta la expresión ***unpersist***, moviendo en este caso la definición de la clase FIGURA, hacia el ambiente global y "borrándola" del ambiente persistente, mostrando al usuario su valor.

LEPPOOC> (bye)

ADVERTENCIA-PROBABLEMENTE-HAY-OBJETOS-PERSISTENTES-CON-SUS-CLASES-EN-AMBIENTE-GLOBAL-VUELVALAS-PERSISTENTES-O-BORRE-LOS-OBJETOS-DEL-AMB-PERSIST

- LEPPOOC interpreta el comando ***(bye)***, verificando que no existan ***referencias colgantes***, de ser así como en este caso, envía el mensaje previo, dado que habíamos pasado al ambiente global la definición de la clase FIGURA, y aún teníamos objetos de esta clase en el ambiente persistente. Podemos observar que a través de esta verificación estamos asegurando la integridad referencial en nuestros datos.

LEPPOOC> (persist-class)
BYE
>

- En este caso, al interpretar LEPPOOC la expresión ***persist-class***, realiza una búsqueda secuencial para hacer persistentes todas las referencias que existan en objetos del ambiente persistente a definiciones que se encuentren en el ambiente global, volviéndolas persistentes, para dejar nuevamente la consistencia en los datos del ambiente persistente, devolviendo en este caso el valor ***bye*** y saliendo de LEPPOOC.

Ejemplo 4. Programa HERENCIA.

Suponga que se tiene la jerarquía de clases que se muestra en la Figura 5.3.

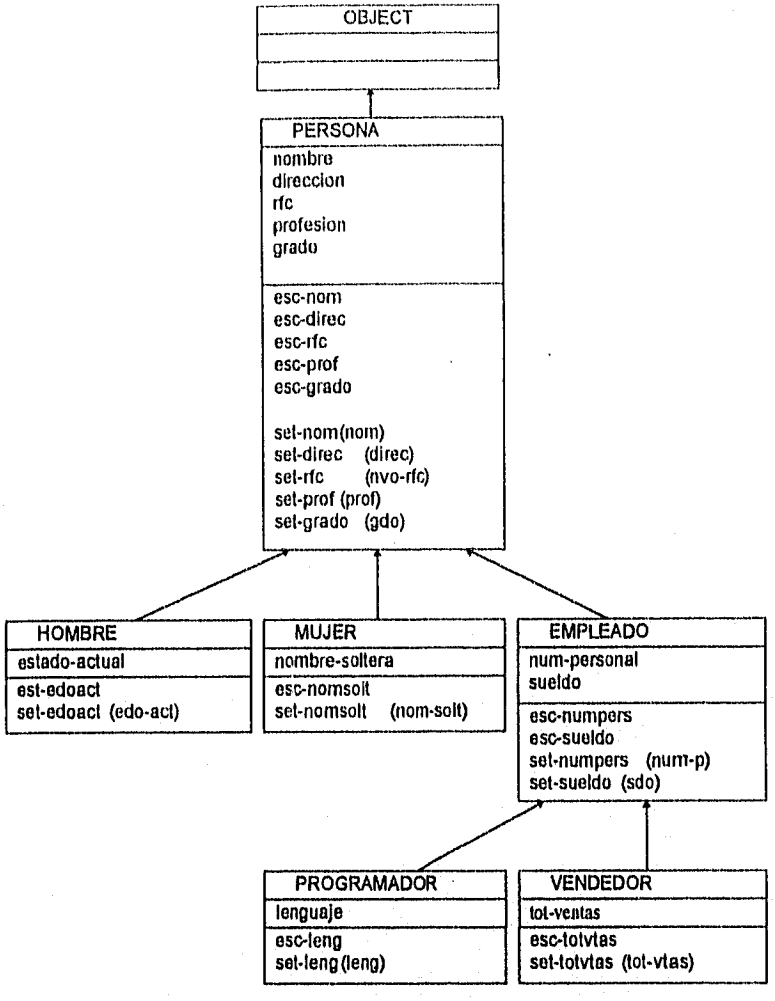


Figura 5.3 Jerarquía de clases: HERENCIA.

Podemos observar que la clase PERSONA es subclase del tipo OBJECT, con variables de instancia: ***nombre, dirección, rfc, profesión,*** y ***grado***; así mismo

cuenta con los métodos ***esc-nom, esc-direc, esc-rfc, esc-prof*** y ***esc-grado***, que lo único que harán será imprimir el valor del ***nombre, dirección, rfc, profesión*** y ***grado del empleado*** respectivamente. También tiene los métodos ***set-nom, set-direc, set-rfc set-prof*** y ***set-grado***, todos ellos con un argumento cada uno que será un valor variable para el ***nombre, dirección, rfc, profesión*** y ***grado*** respectivamente.

Además, están definidas tres ***clases: hombre, mujer*** y ***empleado*** que son subclases de la clase PERSONA. Las clases ***programador*** y ***vendedor*** que son subclases de la clase PERSONA. Todas con sus respectivas variables de instancia y métodos.

En este ejemplo, podemos apreciar con más claridad el concepto de ***herencia*** ya que las subclases heredan los atributos y métodos de sus clases bases.

A continuación, se muestra el código de la definición de estas clases y se crean objetos de las mismas, manipulando sus métodos para estudiar el comportamiento de cada objeto.

Programa HERENCIA escrito en LEPPOOC

```
(DS)  (class persona object
      (nombre dirección rfc profesión grado)
      (define esc-nom () (print nombre))
      (define esc-direc () (print dirección))
      (define esc-rfc () (print rfc))
      (define esc-prof () (print profesión))
      (define esc-grado ()
```

continúa...

...continuación, programa HERENCIA.

```
            (if (= grado 0) (print "#Sin grado")
              (if (= grado 1) (print "#Licenciatura")
                (if (= grado 2) (ptint "Especialidad)
                  (if (= grado 3) (print "#Maestria")
                    (print "#Doctorado") )))))
            (define set-nom (nom) (set nombre nom))
            (define set-direc (direc) (set dirección direc))
            (define set-rfc (nvo-rfc) (set rfc nvo-rfc))
            (define set-prof (prof) (set profesión prof))
            (define set-grado (gdo) (set grado gdo))
        )
(3)         (class hombre persona
            (estado-actual)

              (define esc-edoact ()
                (if (= estado-actual 0) (print "#soltero")
                  (if (= estado-actual 1) (print "#casado")
                    (if (= estado-actual 2) (print "#viudo")
                      (print "#divorciado") ))))
              (define set-edoact (edo-act)
                (set estado-actual edo-act))
        )

(4)     (class mujer persona
        (nombre-soltera)

          (define esc-nomsolt () (print nombre-soltera))
          (define set-nomsolt (nom-solt)
              (set nombre-soltera nom-solt))
        )

(5)     (class empleado persona
          (num-personal sueldo jefe)

          (define esc-numpers () (print num-personal))
          (define esc-sueldo () (print sueldo))
          (define set-numpers (num-pers)
              (set num-personal num -pers))
          (define set-sueldo (sdo) (set sueldo sdo))
          (define set-jefe (mi-jefe) (set jefe mijefe))
        )
(6)     (class programador emmpleado
        (lenguaje)
        (define esc-leng () (print lenguaje))
        (define set-leng (leng)(set lenguaje leng))
        )
(7)     (class vendedor empleado
```

continúa...

...continuación, programa HERENCIA.

```
        (tot-ventas)
        define esc-totvtas () (print tot-ventas))
        (define (set-totvtas(tot-vtas) (set tot-ventas tot-vtas))
    )
(8)  (set obj-persona (new persona))
(9)  (set obj-hombre (new hombre))
(10) (set obj-mujer (new mujer))
(11) (set onj-empleado (new empleado))
(12) (set mi-jefe (new empleado))
(13) (set obj-programador (new programador))
(14) (set obj-vendedor (new vendedor))

     (obj-hombre set-nom "#Manuel Gonzalez Avila")
     (obj-hombre set-direc "#Lucio 25, Xalapa, Ver.")
     (obj-hombre set-rfc "#GOAM701125")
     (obj-hombre set-prof "#Contador Publico")
(15) (obj-hombre set-grado 1)
     (obj-hombre set-edoact 3)
     (obj-hombre esc-nom)
     (obj-hombre esc-direc)
     (obj-hombre esc-grado)
     (obj-hombre esc-edoact)
(16) obj-hombre

     (obj-empleado set-nom "#Roberto Faria Arias")
     (obj-empleado set-direc "#Revolucion 253, Xalapa, Ver.")
     (obj-empleado set-rfc "#FAAR720230")
     (obj-empleado set-prof "#Matematico")
     (obj-empleado set-grado 3)
     (obj-empleado set-numpers 1212)
(17) (obj-empleado set-sueldo 2300)
     (obj-empleado esc-nom)
     (obj-empleado esc-direc)
     (obj-empleado esc-grado)
     (obj-empleado esc-numpers)
     (obj-empleado esc-sueldo)

(18) obj-empleado
```

continúa...

...continuación, programa HERENCIA.

```
                (obj-vendedor set-nom "#Arturo Arista Velez")
                (obj-vendedor "#Avila Camacho 221, Xalapa, Ver.")
                (obj-vendedor set-rfc "#FARVA631202")
                (obj-vendedor set-prof "#Contador Publico")
                (obj-vendedor set-numpers 100)
                (obj-vendedor set-sueldo 1500)
(19)            (obj-vendedor set-totvtas 12000)
                (obj-vendedor esc-nom)
                (obj-vendedor esc-direc)
                (obj-vendedor esc-grado)
                (obj-vendedor esc-numpers)
                (obj-vendedor esc-sueldo)
                (obj-vendedor esc-totvtas)

(20)    obj-vendedor

                (mi-jefe set-nom "#Cecilia Rojas Estevez")
                (mi-jefe se-direc "#Juarez 34, Xalapa, Ver.")
                (mi-jefe set-rfc "#ROEC450611")
                (mi-jefe set-prof "#Ingeniero")
                (mi-jefe set-grado 4)
                (mi-jefe set-numpers 600)
(21)            (mi-jefe set-sueldo 6000)
                (mi-jefe esc-nom)
                (mi-jefe esc-direc)
                (mi-jefe esc-grado)
                (mi-jefe esc-numpers)
                (mi-jefe esc-sueldo)
                (mi-jefe esc-totvtas)
                (mi-jefe set-jefe  "#yo soy")

(22)    mi-jefe
```

A continuación, se describe la ejecución del programa HERENCIA, en LEPPOOC.

(1) **LEPPOOC>(read "herencia")**

Cuando se introduce esta expresión, se busca que exista un archivo con el nombre ***herencia***, entonces interpreta expresión por expresión leyéndolas desde el archivo.

(2) (OBJECT (NOMBRE DIRECCION RFC PROFESION GRADO) ESC-NOM (FUNCION (NIL (PRINT NOMBRE))) ESC-DIREC (FUNCION (NIL (PRINT DIRECCION))) ESC-RFC (FUNCION (NIL (PRINT RFC))) ESC-PROF (FUNCION (NIL (PRINT PROFESION))) ESC-GRADO (FUNCION (NIL (IF (= GRADO 0) (PRINT "#Sin grado") (IF (= GRADO 1) (PRINT "#Licenciatura") (IF (= GRADO 2) (PRINT "#Especialidad") (IF (= GRADO 3) (PRINT "#Maestria") (PRINT "#Doctorado")))))) SET-NOM (FUNCION ((NOM) (SET-NOMBRE NOM))) SET-DIREC (FUNCION ((DIREC) (SET DIRECCION DIREC))) SET-RFC (FUNCION ((NVO-RFC) (SET REF NVO-RFC))) SET-PROF (FUNCION ((PROF) (SET PROFESION PROF))) SET-GRADO (FUNCION ((GDO) (SET GRADO GDO))))

Este es el resultado de definir la clase persona a través de la expresión ***(class persona object ...)***

(3) (PERSONA (ESTADO-ACTUAL) ESC-EDOACT (FUNCION (NIL (IF (=ESTADO-ACTUAL 0) (PRINT "#soltero") (IF (= ESTADO-ACTUAL 1) (PRINT "#casado") (IF (= ESTADO-ACT 2) (PRINT "#viudo") (PRINT "#divorciado")))))) SET-EDOACT (FUNCIO ((EDO-ACT) (SET ESTADO-ACTUAL EDO-ACT))))

Este es el resultado de definir la clase ***hombre*** a través de la expresión ***(class hombre persona ...)***, generando así una subclase de la clase ***persona***.

(4) (PERSONA (NOMBRE-SOLTERA) ESC-NOMSOLT (FUNCION (NIL (PRINT NOMBRE-SOLTERA))) SET-NOMSOLT (FUNCION ((NOM-SOLT) (SET NOMBRE-SOLTERA NOM-SOLT))))

Este es el resultado de definir la clase ***mujer*** a través de la expresión ***(class mujer persona...)***, generando así una subclase de la clase ***persona***

(5) (PERSONA (NUM-PERSONAL SUELDO JEFE) ESC-NUMPERS (FUNCION (NIL (PRINT NUM-PERSOONAL))) ESC-SUELDO (FUNCION (NIL (PRINT SUELDO))) SET NUMPERS (FUNCION ((NUM-PERS) (SET NUM-PERSONAL NUM-PERS))) SET-SUELDO (FUNCION ((SDO) (SET SUELDO SDO))) SET-JEFE (FUNCION ((MIJEFE) (SET JEFE MIJEFE))))

Este es el resultado de definir la clase ***empleado*** a través de la expresión ***(class empleado persona ...)***, generando así una subclase de la clase ***persona***.

(6) (EMPLEADO (LENGUAJE) ESC-LENG (FUNCION (NIL (PRINT LENGUAJE))) SET-LENG (FUNCION ((LENG) (SET LENGUAJE LENG))))

Este es el resultado de definir la clase programador a través de la expresión ***(class programador empleado ...)***, generando así la definición de una subclase de la clase ***empleado*** que a su vez es una subclase de la clase ***persona***.

(7) (EMPLEADO (TOT-VENTAS) ESC-TOTVTAS (FUNCION (NIL (PRINT TOT-VTAS))) SET-TOTVTAS (FUNCION ((TOT-VTAS) (SET TOT-VENTAS TOT-VTAS))))

Este es el resultado de definir la clase ***vendedor*** a través de la expresión ***(class vendedor empleado ...)***, generando así la definición de una subclase de la clase ***empleado*** que a su vez es una subclase de la clase ***persona***.

(8) (NOMBRE (NUMERO 0) DIRECCION (NUMERO 0) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Esto es el resultado de crear el objeto ***obj-persona*** de la clase ***persona***, vacío y con los tipos por omisión.

(9) (ESTADO-ACTUAL (NUMERO 0) NOMBRE (NUMERO 0) DIRECCION (NUMERO 0) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Esto es el resultado de crear el objeto ***obj-hombre*** de la clase ***hombre***, vacío y con los tipos por omisión.

(10) (NOMBRE-SOLTERA (NUMERO 0) NOMBRE (NUMERO 0) DIRECCION (NUMERO 0) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Esto es el resultado de crear el objeto ***obj-mujer*** de la clase ***mujer***, vacío y con los tipos por omisión.

...

(15) "#Manuel Gonzalez Avila"
"#Lucio 25, Xalapa, Ver."
"#GOAM701125"
"#Contador Publico"
1
3
"#Manuel Gonzalez Avila"
"#Manuel Gonzalez Avila"
"#Lucio 25, Xalapa, Ver."
"#Lucio 25, Xalapa, Ver."
"#Licenciatura"
"#Licenciatura"
"#divorciado"
"#divorciado"

Es el resultado de aplicar los métodos de la clase del objeto ***obj-hombre***, y de sus superclases.

(16) (ESTADO-ACTUAL (NUMERO 3) NOMBRE (CADENA "#Manuel Gonzalez Avila") DIRECCION (CADENA "#Lucio 25, Xalapa, Ver.") RFC (CADENA "#GOAM701125") PROFESION (CADENA "#Contador Publico") GRADO (NUMERO 1))

Es el resultado de la expresión obj-hombre, que LEPPOOC interpreta regresando su valor (para este caso el del objeto obj-hombre), con los valores de sus variables de instancia ya modificados de acuerdo a la aplicación previa de los métodos de sus clases y superclases.

(17) "#Roberto Farias Arias"
"#Revolución 253, Xalapa, Ver."
"#FAAR720230"
"#Matematico"
3
1212
2300
"#Roberto Farias Arias"
"#Roberto Farias Arias"
"#Revolución 253, Xalapa, Ver."
"#Revolución 253, Xalapa, Ver."
"#Maestria"
"#Maestria"
1212
1212
2300
2300

Es el resultado de aplicar los métodos de la clase del objeto ***obj-empleado***, y de sus superclases.

(18) (NUM-PERSONAL (NUMERO 1212) SUELDO (NUMERO 2300) JEFE (NUMERO 0) NOMBRE (CADENA "#Roberto Farias Arias") DIRECCION (CADENA "#Revolución 253, Xalapa, Ver.") RFC (CADENA "FAAR720230") PROFESION (CADENA "#Matematico") GRADO (NUMERO 3))

Es el resultado de la expresión ***obj-empleado***, que LEPPOOC interpreta regresando su valor (para este caso el del objeto ***obj-empleado***), con los valores de sus variables de instancia ya modificados de acuerdo a la aplicación previa de los métodos de sus clases y superclases.

...

(22) (NUM-PERSONAL (NUMERO 500) SUELDO (NUMERO 6000) JEFE (CADENA "#yo soy") NOMBRE (CADENA "#Cecilia Rojas Estevez") DIRECCION (CADENA "#Juarez 34, Xalapa, Ver.") RFC (CADENA "#ROEC450511") PROFESION (CADENA "#Ingeniero") GRADO (NUMERO 4))

Es el resultado de la expresión ***mi-jefe***, que LEPPOOC interpreta regresando su valor (para este caso el del objeto ***mi-jefe***), con los valores de sus variables de instancia ya modificados de acuerdo a la aplicación previa de los métodos de sus clases y superclases.

◊ Ejemplo 5. Programa COMPARTIDO.

Es importante mostrar como se pueden compartir objetos, tanto con réplicas, como con apuntadores, implantando con esto el concepto de ***identidad*** y el concepto de ***objetos compuestos***.

A continuación mostramos la estructura de las clases que serán utilizadas para mostrar los conceptos de ***identidad*** y ***objetos compuestos*** ilustrados en la Figura 5.4.

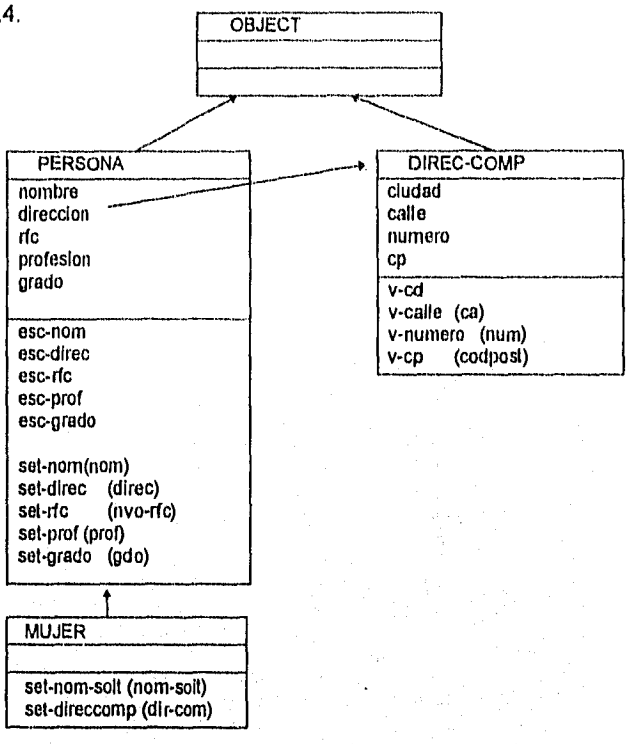


Figura 5.4: Objetos Compuestos.

Utilizando la misma clase ***persona*** que en el ejemplo anterior y nuevas clases como la clase ***mujer*** como subclase de la clase ***persona***, sin variables de instancia, y dos métodos que modifican el estado de objetos de esta clase, ***set-nomsolt*** con argumento ***nom-solt***, para cambiar el valor de la variable de instancia nombre de ***persona***; y el método ***set-direccomp***, con argumento ***dir-com*** que permite cambiar el estado de la variable de instancia dirección de persona.

Así mismo, una nueva clase ***direc-comp*** de la clase ***object***, con variables de instancia ***ciudad, calle, número*** y ***cp***, y cuatro métodos: ***v-cd*** sin argumentos y que establece una ciudad fija, ***v-calle*** con argumento ***ca***, ***v-numero*** con argumento ***num***, ***v-cp*** con argumento ***codpost***, donde cada uno de estos tres últimos métodos con sus argumentos establecen un valor variable para calle, número y código postal.

A continuación, se muestra el código del programa en LEPPOOC que implanta estas clases y crea objetos que manipulan los métodos que les cambian su estado.

Código del programa COMPARTIDO escrito en LEPPOOC.

```
(2)   (class persona object
        (nombre dirección rfc profesión grado)
        (define esc-nom () (print nombre))
        (define esc-direc () (print dirección))
        (define esc-rfc () (print rfc))
        (define esc-prof () (print profesión))
        (define esc-grado ()
          (if (= grado 0) (print "#Sin grado")
            (if (= grado 1) (print "#Licenciatura")
              (if (= grado 2) (print "Especialidad)
                (if (= grado 3) (print "#Maestría")
                  (print "#Doctorado") )))))
        (define set-nom (nom) (set nombre nom))
        (define set-direc (direc) (set dirección direc))
        (define set-rfc (nvo-rfc) (set rfc nvo-rfc))
        (define set-prof (prof) (set profesión prof))
        (define set-grado (gdo) (set grado gdo))
      )
(3)   (class mujer persona
        ()
        (define set-nomsolt (nom-solt)
                (set nombre nom-solt))
        (define set-direccomp (dir-com)
                (set dirección dir-com))
      )
(4)   (class direc-comp object
        (ciudad calle numero cp)
(define v-cd () (set ciudad "#Xalapa, Ver."))
        (define v-calle (ca) (set calle ca))
```

continúa...

...continuación., programa COMPARTIDO.

```
        (define v-numero (num) (set numero num))
        (define v-cp (codpost) (set cp codpost))
    )
(5) (set obj-persona (new persona))
(6) (set obj-mujer (new mujer))
(7) (set obj-dircomp (new direc-comp))
(8) (obj-mujer set-nom "#Juana")
(9) obj-mujer

(10) (obj-dircomp v-cd)
     (obj-dircomp v-calle "#Benito Juarez")
     (obj-dircomp v-numero 55)
     (obj-dircomp v-cp 91000

(11) obj-dircomp
(12) (obj-mujer set-direccomp obj-dircomp)
(13) obj-mujer
(14) (set obj-soltera (new mujer))
(15) obj-soltera
(16) (obj-soltera set-nom "#Juana Comparte")
(17) (obj-soltera set-direccomp (quote obj-dircomp))
(18) obj-soltera
(19) (obj-soltera esc-direc)
(20) ((obj-soltera esc-direc) v-calle "#Lucio")
(21) obj-soltera
(22) obj-dircomp
```

En seguida mostramos la ejecución del programa ***"compartido"***.

(1) **LEPPOOC> (read "compartido")**

Cuando introducimos esta expresión a LEPPOOC, éste busca que exista un archivo con el nombre ***compartido***, y entonces interpreta expresión por expresión leyéndolas desde este archivo.

(2) (OBJECT (NOMBRE DIRECCION RFC PROFESION GRADO) ESC-NOM (FUNCION (NIL (PRINT NOMBRE))) ESC-DIREC (FUNCION (NIL (PRINT DIRECCION))) ESC-RFC (FUNCION (NIL (PRINT RFC))) ESC-PROF

(FUNCION (NIL (PRINT PROFESION))) ESC-GRADO (FUNCION (NIL (IF (= GRADO 0) (PRINT "#Sin grado") (IF (= GRADO 1) (PRINT "#Licenciatura") (IF (= GRADO 2) (PRINT "#Especialidad") (IF (= GRADO 3) (PRINT "#Maestria") (PRINT "#Doctorado)))))))) SET-NOM (FUNCION ((NOM) (SET-NOMBRE NOM))) SET-DIREC (FUNCION ((DIREC) (SET DIRECCION DIREC))) SET-RFC (FUNCION ((NVO-RFC) (SET REF NVO-RFC))) SET-PROF (FUNCION ((PROF) (SET PROFESION PROF))) SET-GRADO (FUNCION ((GDO) (SET GRADO GDO))))

Este es el resultado de definir la clase ***persona*** a través de la expresión ***(class persona object ...)***

(3) (PERSONA NIL SET-NOMSOLT (FUNCION ((NOM-SOLT) (SET NOMBRE NOM-SOLT))) SET-DIRECCOMP (FUNCION ((DIR-COM) (SET DIRECCION DIR-COM))))

Este es el resultado de definir la clase ***mujer*** a través de la expresión ***(class mujer persona ...)***

(4) (OBJECT (CIUDAD CALLE NUMERO CP) V-CD (FUNCION (NIL (SET CIUDAD "#Xalapa, Ver."))) V-CALLE (FUNCION ((CA) (SET CALLE CA))) V-NUMERO (FUNCION ((NUM) (SET NUMERO NUM))) V-CP (FUNCION ((CODPOST) (SET CP CODPOST))))

Este es el resultado de definir la clase ***direc-comp*** a través de la expresión ***(class direc-comp object ...)***.

(5) (NOMBRE (NUMERO 0) DIRECCION (NUMERO 0) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Esto es el resultado de crear el objeto ***obj-persona*** de la ***clase persona***, vacío y con los tipos por omisión.

(6) (NOMBRE (NUMERO 0) DIRECCION (NUMERO 0) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Esto es el resultado de crear el objeto ***obj-mujer*** de la clase ***mujer***, vacío y con los tipos por omisión.

(7) (CIUDAD (NUMERO 0) CALLE (NUMERO 0) NUMERO (NUMERO 0) CP (NUMERO 0))

Esto es el resultado de crear el objeto ***obj-dircomp*** de la clase ***dirc-comp***, vacío y con los tipos por omisión.

(8) "#Juana"

Este es el resultado de aplicar el método ***set-nom*** de la superclase ***persona***, que envió como mensaje el objeto ***obj-mujer***.

(9) (NOMBRE (CADENA "#Juana") DIRECCION (NUMERO 0) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Es el resultado de la expresión ***obj-mujer*** que LEPPOOC lo interpreta enviando su valor, que para este caso resultó ser el objeto ***obj-mujer***, ya modificado por la aplicación del método ***set-nom***.

(10) "#Xalapa, Ver."
"#Benito Juarez"
55
91000

Son el resultado de aplicar los métodos de la clase ***direc-comp*** a la que pertenece el objeto ***obj-dircomp***, mismo que envió los mensajes.

(11) (CIUDAD (CADENA "#Xalapa, Ver.") CALLE (CADENA "#Benito Juarez") NUMERO (NUMERO 550) CP (NUMERO 9100))

Es el resultado de la expresión ***obj-dircomp*** que LEPPOOC la interpreta enviando su valor, que para este caso resultó ser el objeto ***obj-dircomp***, ya modificado por la aplicación de los métodos de su clase.

(12) (CIUDAD (CADENA "#Xalapa, Ver.") CALLE (CADENA "#Benito Juarez") NUMERO (NUMERO 550) CP (NUMERO 9100))

Es el resultado de aplicar el método ***set-direccomp*** con el parámetro compuesto ***obj-dircomp***, al objeto ***obj-mujer***. Este método se encarga de imprimir el valor del argumento. Aquí podemos apreciar la **composición** del objeto ***obj-mujer*** con el objeto ***obj-dircomp***.

(13) (NOMBRE (CADENA "#Juana") DIRECCION (DIREC-COMP (CIUDAD (CADENA "#Xalapa, Ver.") CALLE (CADENA "#Benito Juarez") NUMERO (NUMERO 550) CP (NUMERO 9100))) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Es el resultado de la expresión ***obj-mujer*** que LEPPOOC lo interpreta enviando su valor, que para este caso resultó ser el objeto ***obj-mujer***, ya modificado por la aplicación de los métodos de su clase. Aquí podemos apreciar la composición del objeto ***obj-mujer*** con el objeto con la clase ***direc-comp***, a través de una réplica como valor, de la variable de instancia dirección, del objeto ***obj-mujer***.

(14) (NOMBRE (NUMERO 0) DIRECCION (NUMERO 0) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Esto es el resultado de crear el objeto ***obj-soltera*** de la clase mujer, vacío y con los tipos por omisión

(15) (NOMBRE (NUMERO 0) DIRECCION (NUMERO 0) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Es el resultado de la expresión ***obj-soltera*** que LEPPOOC interpreta enviando su valor, que para este caso resultó ser el objeto ***obj-soltera***, recién creado.

(16) "#Juana Comparte"

Es el resultado de aplicar el método ***set-nom*** al objeto ***obj-soltera***.

(17) OBJ-DIRCOMP

Es el resultado de aplicar el método ***set-direccomp*** con el parámetro siendo el nombre del objeto ***object-dircomp***, creado por una expresión ***quote,*** al objeto ***obj-soltera***, resultando como valor el mismo nombre OBJ-DIRCOMP.

(18) (NOMBRE (CADENA "#Juana Comparte") DIRECCION (ALIAS OBJ-DIRCOMP) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Es el resultado de la expresión ***obj-soltera***, que LEPPOOC interpreta enviando su valor, mismo que ha sido modificado al aplicar los métodos de sus clases y superclases. Aquí podemos observar el concepto de **objeto compuesto**, ya que estamos haciendo que la variable de instancia

direccion sea un apuntador al objeto DIRCOMP, asegurando con esto que siempre que cambie el objeto real ***obj-dircomp***, se garantiza apuntar al nuevo valor.

(19) OBJ-DIRCOMP
OBJ-DIRCOMP

Es el resultado de enviar el mensaje ***esc-direc*** a la clase del objeto ***obj-soltera.***

(20) OBJ-DIRCOMP
"#Lucio 34"

Son el resultado de enviar el mensaje ***esc-direc*** al objeto ***obj-soltera,*** quien devuelve el nombre de otro objeto, OBJ-DIRCOMP, mismo que se utiliza como transmisor para enviar el método ***v-calle*** con un argumento constante, a la clase a la que pertenece, regresando como resultado la dirección real.

(21) (NOMBRE (CADENA "#Juana Comparte") DIRECCION (ALIAS OBJ-DIRCOMP) RFC (NUMERO 0) PROFESION (NUMERO 0) GRADO (NUMERO 0))

Es el resultado de la expresión ***obj-soltera*** que LEPPOOC interpreta regresando su valor, que podemos observar no ha cambiado en lo que respecta a la variable de instancia ***direccion***, dado que éste es un "apuntador" a la dirección real.

(22) (CIUDAD (CADENA "#Xalapa, Ver.") CALLE (CADENA "#Lucio 34") NUMERO (NUMERO 550) CP (NUMERO 9100))

Es el resultado de la expresión ***obj-dircomp*** que LEPPOOC interpreta regresando su valor, que como podemos observar ha cambiado de estado en lo que respecta a la variable de instancia ***calle,*** al haber aplicado el método ***v-calle*** en el objeto compuesto ***obj-soltera***.

5.6 PROBLEMAS CON LEPPOOC.

Para tener atributos cuyo valor es el ***nombre*** de un objeto, el constructor de nombres de objetos (la expresión ***quote***) aparece en LEPPOOC.

Para asignar nombres de objetos al atributo del objeto compuesto se requiere de una expresión de asignación que no evalúa su argumento y en consecuencia construye una réplica, sino que, no evalúa su argumento y lo pasa como un nombre. Esto es lo que hace la expresion ***set-shallow***.

En ambos casos se corre el riesgo que el argumento no sea un objeto y por lo tanto se obtenga el nombre de "algo" que no es un objeto, pero esto es un aspecto que LEPPOOC debería revisar; sin embargo nuestro prototipo no lo hace en su versión actual. Este problema es el siguiente. El uso de ***set-shallow*** y ***quote*** como en el siguiente ejemplo:

```
(set mi-circulo(new circulo))
(set-shallow otro-nombre mi-circulo)
```

y esto es equivalente a

```
(set otro-nombre(quote mi-circulo))
```

En este ejemplo, ***mi-circulo*** es un objeto de la clase ***figura***. Pero es posible en la versión actual ejecutar

```
( set otro-nombre(quote((x) (+ x 1))))
```

lo cual es un error, pues *((x) (+ x 1))* no es objeto de ninguna clase.

Otro ejemplo de este problema en el intérprete actual es *(quote 5)*, lo cual también es un error. Hemos corregido hasta cierto punto este problema, pues el intérprete actual verifica que el segundo argumento de ***set-shallow*** sea un nombre en los ambientes por lo que *(set-shallow x 5)*, que es un error de programación (pues 5 es un valor simple y no es el nombre de un objeto) es detectado por nuestro intérprete.

5.6 COMENTARIOS FINALES.

Como vimos, no existe una universalidad aceptada en la notación para describir la semántica de un lenguaje, por lo que hemos dado una explicación lo más precisa posible de las expresiones más importantes de LEPPOOC haciendo uso de la técnica formal denominada ***"semántica denotativa"*** y lenguaje natural, de tal manera que al lector le sea fácil su manejo y entendimiento.

Cabe aclarar que la semántica de los comandos ***read*** y ***bye*** es muy obvia, por lo que se omitieron, ya que fue explicada en el capítulo tres, y consideramos que no era necesaria mayor explicación.

Del mismo modo, los significados de las expresiones correspondientes a los operadores de LEPPOOC como: ***print, lista, car, cdr*** y los operadores aritméticos básicos como +, -, * y /, así como los operadores de relación: =, <, y >, consideramos que fueron suficientemente explicados en la sección 3.2.2.2 del capítulo tres, correspondiente a la sintaxis, al hacer una visita a LEPPOOC. En el mismo caso se encuentra la aplicación de funciones definidas por el usuario.

6 ESTRUCTURA DE LEPPOOC

El código de nuestro intérprete, escrito en LISP, aparece en el ***apéndice A***. Este capítulo es la documentación de este programa. El listado del programa se encuentra dividido en funciones, y la documentación la hemos estructurado de acuerdo a la función de mayor importancia: EVAL-INPUT. Tal función es invocada por la función EVAL-LEPPOOC quien controla la ejecución de nuestro intérprete.

La función EVAL-LEPPOOC se encuentra estructurada como se muestra en el diagrama de módulos general de LEPPOOC de la Figura 6.1.

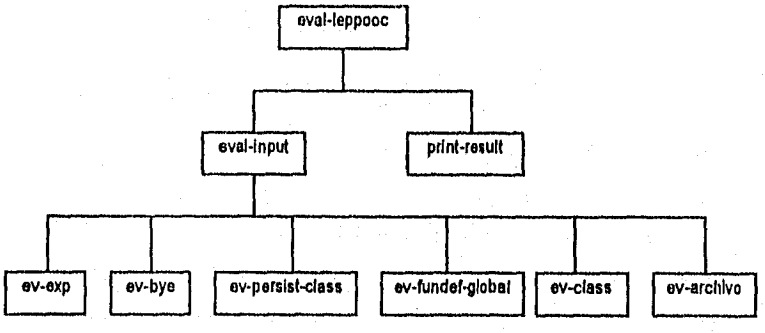


Figura 6.1: Diagrama de módulos general de la estructura de LEPPOOC.

Como podemos observar en la Figura 6.1, en el primer nivel tenemos que la función EVAL-LEPPOOC hace uso de la función: EVAL-INPUT misma que se explicará con detalle en la sección 6.2, así como de la función ***print-result***.

La función eval-leppooc se encarga de abrir el archivo con nombre ***"arch-pers"***, que contiene el ambiente persistente, el cual debe definirse antes de ejecutar nuestro intérprete. Quizá ésta es una limitante del mismo en la versión actual, pero como para fines de este trabajo esto no fue relevante, se dejó al lector la decisión de modificar esta función para que se pueda abrir un archivo con el nombre que el usuario desee. Del mismo modo, es aquí donde se crea el contexto o ambiente, ***C*** o ***amb***, para las variables, objetos, funciones y clases que manejará LEPPOOC y que se explica en la sección 6.1. Como se mencionó en el capítulo 3, este contexto consta de tres ambientes el ***local***, ***global*** y ***persistente***, por lo que el ambiente inicial se forma con el ambiente local vacío, el ambiente global con un elemento que viene a ser ***object***, la clase base del sistema, y el ambiente persistente que se encuentra en el archivo ***"arch-pers"***.

Esta función se encuentra estructurada con la proposición LOOP de LISP para imprimir el "prompt" de nuestro intérprete, que es: ***LEPPOOC>***. Es aquí donde se leen las entradas en la variable global ***"prog"*** que LEPPOOC interpretará llamando a la función EVAL-INPUT con el ambiente ***amb*** previamente formado.

Después de salir de la función EVAL-INPUT se reconstruye el contexto dejando vacío el ambiente local y si el tipo del resultado de la variable global ***prog*** es advertencia, se imprime el mensaje y se regresa a pedir una nueva entrada para LEPPOOC.

Si el tipo del resultado de la variable ***prog*** es fin, se guarda el ambiente persistente en el archivo ***"arch-pers"***, se cierra y se sale de LEPPOOC quedando en el ambiente de LISP.

Si el tipo del resultado de ***prog*** es ***lista*** se imprime la lista que se formó con la función auxiliar para imprimir listas ***print-result*** y se regresa a pedir una nueva entrada para LEPPOOC.

Por último, si el tipo del resultado de ***prog*** no fue ninguno de los anteriores, entonces imprime el resultado y regresa a pedir una nueva entrada para LEPPOOC.

6.1 CONTEXTO O AMBIENTES DE LEPPOOC

El contexto o ambiente ***amb***, es una lista de tres elementos: ***ambiente local, global*** y ***persistente***, donde un elemento de cualquiera de estos ambientes tienen la estructura siguiente:

(identificador (tipo valor))

donde ***identificador*** es un nombre y su atributo es la pareja ***(tipo valor)***.

Interpretando la estructura de cada elemento de un ambiente, tenemos que ***identificador*** viene a ser el nombre de una variable, objeto, función o clase en LEPPOOC; ***tipo*** determinará el tipo del ***identificador;*** existen nueve tipos que son: ***numero, cadena, clase, función, alias, lista, object*** (los asigna el sistema y es el tipo de la clase base), ***error*** (lo asigna el sistema para determinar cuando existe un error de sintaxis principalmente) y ***el nombre de***

una clase a la que pertenece un objeto específico. ***valor*** es, como su nombre lo indica, el valor que tiene asociado el ***identificador***.

En el ambiente local se guardarán los valores de los parámetros de las funciones escritas en LEPPOOC. Este ambiente desaparece cuando se sale de la función que se está ejecutando.

En el ambiente global se guardarán los valores de cualquier variable, objeto, función, clase o lista que el usuario esté manejando dentro de sus proposiciones, a menos que explícitamente pida que se guarden en el ambiente persistente.

En el ambiente persistente se guardarán los valores que el usuario de LEPPOOC desee, siempre que éste lo haga explícitamente a través de la expresión ***persist*** o del comando ***persist-class,*** como se explicó en el capítulo 3.

A continuación presentamos un ejemplo de un ambiente en LEPPOOC, después de ejecutar el programa que aparece en el apéndice B:

Ambiente del programa círculo.

(NIL (OBJECT (CLASE (OBJECT NIL NIL)) A (NUMERO 50.2626) (MI-FIG (FIGURA (PI (NUMERO 0) R (NUMERO 0))) FIGURA (CLASE (OBJECT (PI R) VPI (FUNCION (NIL (SET PI 3.1416))) VR (FUNCION ((VALOR) (SET R VALOR))))) CIRCULO (CLASE (FIGURA NIL AREA (FUNCION (NIL (BEGIN (SET A (* PI (* R R))) (PRINT "#area") (PRINT A)))) CIRC (FUNCION (NIL (BEGIN (SET C (* 2 (* PI R))) (PRINT "#circunferencia") (PRINT C)))))) MI- CIRCULO (CIRCULO (PI (NUMERO 3.1416) R (NUMERO 4)))))

6.2 EVAL-INPUT

Como ya se mencionó, ***eval-input*** es la función más importante de nuestro intérprete, ya que es aquí donde se determinará inicialmente el tipo de proposición que introduce el usuario como entrada para LEPPOOC, enviándola junto con el ambiente a la función que la interpretará. En el diagrama anterior podemos visualizar la funciones de que hace uso EVAL-INPUT ***(ev-bye, ev-persist-class, ev-class, ev-fundef-global, ev-archivo y ev-exp)*** y que serán explicadas en las siguientes secciones.

6.2.1 Ev-bye

La función ***ev-bye*** se encarga de revisar los ambientes para determinar si existen referencias colgantes, y dependiendo de la situación que se detecte se envía uno de los siguientes mensajes de ADVERTENCIA:

1) "advertencia-probablemente-hay-objetos-persistentes-con-sus-clases-en-ambiente-global-vuelvalas-persistentes-o-borre-los-objetos-del-amb-persist".

2) "advertencia-probablemente-hay-objetos-persistentes-cuyos-tipos-de-sus-componentes-se-encuentran-en-ag-vuelvalos-persistentes-o-borre-los-objetos-del-amb-pers".

3) "advertencia-probablemente-hay-objetos-en-amb-global-que-son-aliases-de-objetos-de-otros-en-amb-pers-solve-los-persistentes-o-salve-los-del-global"

En caso de no ocurrir ninguna de las situaciones que describen los mensajes, significa que todo está normal, devuelve el tipo ***fin*** con el valor ***bye*** y regresa

el control a la función EVAL-LEPPOOC, de donde sale de LEPPOOC, sin ningún problema.

6.2.2 Ev-persist-class

La función ***ev-persist-class*** se encarga de revisar los ambientes en forma estructurada, por lo que los recibe como parámetros de la siguiente manera: ***amb*** (ambiente completo), ***ambp*** (ambiente persistente, donde se revisa si existe algún objeto, eliminando los que se van revisando) ***ambpo*** (ambiente persistente original, permanece intacto). El objetivo de esta función es dejar consistentes los ambientes. Esta función es recursiva y se encuentra estructurada en los siguientes casos:

1) En el caso que el ambiente ***ambp*** sea nulo, regresa el control a la función ***eval-leppooc*** y sale de LEPPOOC sin mayor problema. Lo que significa que no hay ambiente persistente, o se revisó y no se encontraron referencias colgantes. Esto ocurre así porque la función en cuestión es recursiva.

2) En el caso que el ***ambp*** no esté vacío, se procede a determinar si el primer elemento del ***ambp*** es un objeto y si la clase del mismo se encuentra definida en el ambiente persistente, de no ser así, se procede a hacerla persistente y se vuelve a llamar recursivamente la función ***ev-persist-class*** pero ahora sin el primer elemento en el ambiente ***ambp*** y el nuevo ambiente persistente ***amb*** = ***amb-act***, y el ***ambpo*** sin alteraciones.

3) En el caso que no ocurra ninguna de las dos situaciones anteriores, se procede a determinar si el primer elemento del ***ambp*** es un objeto y de ser así, se revisa mediante una función recursiva si alguna de las partes del

objeto tiene definido su tipo en el ambiente persistente; de no ser así se procede a hacer persistente cada una de los tipos de las partes del objeto que estén en esta situación. Una vez hecho esto, se llama recursivamente a la función ***ev-persist-class***, del mismo modo que en la situación previa.

4) Si no sucedió ninguno de los casos anteriores, se vuelve a llamar recursivamente a la función ***ev-persist-class*** pero ahora sin el primer elemento del ambiente ***ambp***, y los demás parámetros intactos.

6.2.3 Ev-class

La función ***ev-class*** se utiliza para definir las clases del usuario. El diagrama de módulos que se presenta en la Figura 6.2 nos muestra la estructura de la misma.

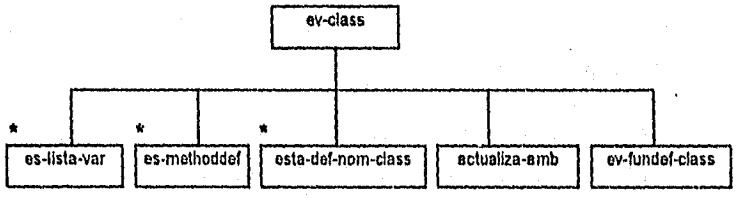


Figura 6.2: Diagrama de módulos de la estructura de ev-class.

Las funciones marcadas con asterisco, son predicados que ayudan a revisar la sintaxis de la expresión ***class***. Los mensajes de ERROR que puede enviar la función ***ev-class*** dependiendo de la situación en la que se encuentre son:

1) "error-la palabra-clave-class-debe-estar-seguida-por-la-def)

2) "error-faltan-argumentos-en-la-def-clase"

3) "error-sintaxis-debe-ser-nombre-de-clase-a-definir"

4) "error-en-def-class-no-se-permite-alterar-tipos"

5) "error-sintaxis-debe-ser-nombre-de-una-superclase"

6) "error-sintaxis-no-es-lista-inst-var"

7) "error-sintaxis-no-es-lista-methoddefú"

8) "error-sintaxis-superclase-nclass2-no-esta-definida"

9) "error-clase-con-nombre-repetido"

10) "error-def-methodos-duplicados"

11) "error-en-def-de-la-clase"

Si ninguno de los mensajes anteriores ocurre, es porque al llamar a ***ev-fundef-class*** todo estuvo en orden y entonces se procede a llamar a la función recursiva ***actualiza-amb*** para introducir la definición de la nueva clase.

6.2.4 Ev-fundef-global

La función ***ev-fundef-global*** se encarga de definir una función en el ambiente global, para lo cual se estructuró como se muestra en la Figura 6.3.

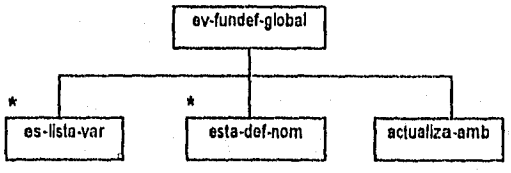


Figura 6.3: Diagrama de módulos de la estructura de la función ***ev-fundef-global.***

Las funciones marcadas con asterisco, son predicados que se encargan de revisar que los parámetros que se encuentran en la definición de la clase ***(es-lista-var)*** son una lista de argumentos, o si la función por definir ya existe ***(esta-def-nom)***, de no ocurrir así en cualquiera de los casos se envía el mensaje correspondiente (4 ó 5). Ambas funciones son recursivas. Los mensejes que puede envier la función ***ev-fundef-global*** dependiendo de la situación que ocurra son:

1) "error-no-existe-def-de-funcion"

2) "error-parametros-isuficientes"

3) "error-nombre-de-función-no-v-lido"

4) "error-en-función-no-es-lista-argumentos"

5) "error-nombre-de-función-ya-existe"

De no ocurrir ninguna de estas situaciones, se procede a actualizar el ambiente global con la nueva definición de función, a través de la función ***actualiza-amb***, devolviendo así el nuevo ambiente completo actualizado.

6.2.5 Ev-archivo

La función ***ev-archivo*** trabaja de manera similar e la función ***eval-leppooc***, con la excepción de que ***ev-archivo*** lee las entrades e LEPPOOC desde un archivo que contiene el programa escrito en nuestro intérprete, una vez ejecutada ésta, regresa el control e ***eval-leppooc***, volviendo a ser interactiva su ejecución.

6.3 EV-EXP

La función central de nuestro intérprete es ***ev-exp***, es aquí donde se determina que expresión se introdujo como entrada para LEPPOOC, enviándola junto con el ambiente a la función que la interpretará.

El diagrama de módulos que representa la estructura de esta función es el que se muestra en la Figura 6.4, mismo que consta de quince funciones o módulos básicos para realizar las expresiones del lenguaje.

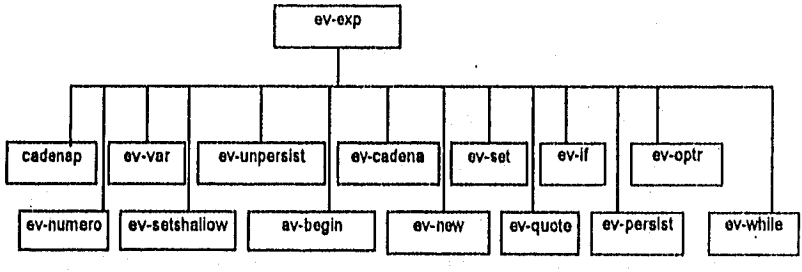


Figura 6.4: Diagrama de módulos de la estructura de la función ***ev-exp***.

Como se puede apreciar en la figura previa, las funciones de que consta ***ev-exp*** son las expresiones básicas que maneja LEPPOOC, siendo por lo tanto el corazón del intérprete; a continuación se explicará cada una de éstas.

6.3.1 Cadenap, Ev-numero, Ev-cadena.

Las funciones ***cadenap, ev-numero*** y ***ev-cadena*** se encargan de realizar operaciones sencillas para los tipos básicos de nuestro intérprete.

◊ La función ***cadenap***, es un predicado que verifica si la expresión que recibe como parámetro es una cadena, de ser así devuelve el valor ***t (verdadero)***, en caso contrario devuelve ***nil (falso)***.

◊ La función ***ev-numero*** simplemente se encarga de crear la ***pareja (numero expresión)*** y la anexa al ambiente. Tanto la expresión como el ambiente son parámetros de esta función.

◊ Por último la función ***ev-cadena*** crea la pareja ***(cadena expresión)*** y la anexa al ambiente. Tanto la expresión como el ambiente son parámetros de esta función.

6.3.2 Ev-var

La función ***ev-var*** tiene como principal función verificar si la variable que recibe como uno de sus parámetros, se encuentra en el ambiente, que también recibe como parámetro, y de ser así la devuelve junto con su tipo y valor; en caso contrario regresa el mensaje "variable-indefinida". Esta función consta de dos funciones como se muestra en la Figura 6.5, con las que realiza las operaciones antes mencionadas.

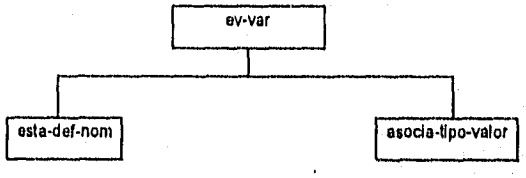


Figura 6.5 Diagrama de módulos de ***ev-var***.

Para verificar si se encuentra la variable, ***ev-var*** hace uso de la función recursiva ***esta-def-nom*** que funciona como un predicado, que nos devuelve ***t*** o ***nil***, según sea el caso. Los parámetros que recibe esta función son, el nombre de la variable y el ambiente donde la va a buscar. La búsqueda que realiza

sobre el ambiente es secuencial: compara el nombre con el de las parejas que se encuentran en el ambiente, si resulta exitosa la comparación, regresa ***t***, de otro modo, si se agota el ambiente y no se encontró, devuelve ***nil***.

Esta-def-nom es una función muy utilizada en varias de las funciones principales de LEPPOOC.

Si la variable existe, ***ev-var*** llama a la función recursiva ***asocia-tipo-valor*** (también de bastante utilidad en todo el contexto de LEPPOOC), quien se encarga, como su nombre lo indica, de asociar el tipo y valor al nombre de la variable. Los parámetros que recibe son el nombre de la variable y el ambiente (uno de los tres ambientes) donde va a buscar su tipo y valor.

6.3.3 Ev-set

La función ***ev-set*** se encarga de "asignar valor" a variables, para lo cual hace uso de cuatro funciones como se muestra en la figura 6.6, siendo una de ellas ***ev-exp***. Los parámetros que recibe ***ev-set*** son la expresión que tiene la forma ***(Identificador subexpresión)*** y el ambiente ***(Al, Ag, Ap)***.

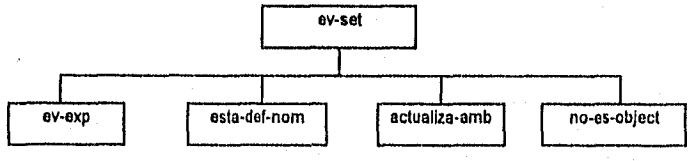


Figura 6.6 Diagrama de módulos de ***ev-set***.

Para que ***ev-set*** pueda realizar su función, primero evalúa la subexpresión en el ambiente de entrada, llamando para esto de manera recursiva a la función ***ev-exp***, creando así un nuevo ambiente. Se determinan lo siguientes casos:

CASO 1: Si hay error en la evaluación de la subexpresión, lo devuelve así a la función que la llamó, mostrándole al usuario el error.

CASO 2 Y 3: Si el ***set*** ocurre en uno de los ambientes ***Al*** o ***Ag***, ya sea que el identificador se encuentre o no en ellos, entonces actualiza el ambiente correspondiente con la función recursiva ***actualiza-amb***, misma que se explicó en la sección 6.2.3, y se genera un nuevo ambiente con los otros intactos.

CASO 4: Si el identificador se encuentra definido en el ambiente persistente, ***ev-set*** emite el mensaje: ***"error-no-se-puede-modificar-amb-pers-con-set"***. Si se recuerda, el ambiente persistente sólo puede ser modificado por ***persist***, por tal motivo ***ev-set*** evita modificarlo, haciendo esta verificación.

CASO 5: Si el identificador es un objeto del tipo ***object***, mismo que es la clase base que define el sistema, ***ev-set***, envía el mensaje ***"error-sintaxis-set-clase-object-no-se-redefine"***, evitando así inconsistencias en las clases, o problemas mayores.

Cabe recalcar que ***ev-set*** elige el caso que sucede y son mutuamente excluyentes.

6.3.4 Ev-persist

La función ***ev-persist*** se encarga de volver persistentes todas las variables, funciones, clases u objetos que el usuario le indique. Los parámetros que recibe son: la expresión (ésta debe ser un ***nombre***) y el ambiente (llamado ***amb***). La Figura 6.7 nos muestra la estructura de la función ***ev-persist***.

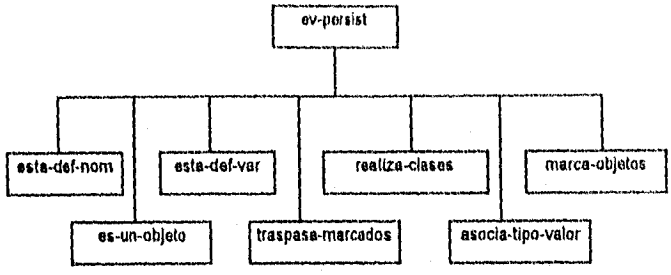


Figura 6.7 Diagrama de módulos de ***ev-persist***.

Lo primero que realiza ***ev-persist***, es una verificación sintáctica, enviando uno de los mensajes siguientes para tal caso:

1) "error-sintaxis-exp-debe-ser-variable-u-objeto".

Este mensaje se envía cuando no existe el nombre de la variable u objeto que se desea hacer persistente.

2) "error-sintaxis-exceso-de-parametros-en-persist".
Persist solo acepta un parámetro por lo que si se excede, se emite este mensaje.

Si no ocurrió ninguno de los mensajes anteriores, ***ev-persist*** se encarga de verificar si el nombre se encuentra en alguno de los ambientes que se encuentra en el parámetro ***amb***, a través de la función recursiva ***esta-def-nom***, de no existir envía el mensaje de error: ***"variable-u-objeto-no-definido"***.

Por otro lado, si el ***nombre*** se localizó en el ambiente local emite el mensaje de error: ***"error-en-persist-variable-u-objeto-en-ambiente-local"***, en caso de

haberse encontrado en ambiente persistente envía el mensaje: "***variable-u-objeto-ya-existente-en-ambiente-persistente***".

La función ***esta-def-var*** verifica que el tipo de la variable sea ***cadena***, ***número***, ***lista***, ***función*** o ***clase*** de ser así pasa la variable con su tipo y valor al ambiente persistente borrándola del global, el ambiente local permanece intacto, dejando así los ambientes actualizados.

La función ***es-un-objeto*** determina si el ***nombre*** es un objeto, y de ser así se proceda a marcar los objetos que son parte del mismo así como sus clases y subclases, para después traspasarlos. En caso de que ya se encuentren en el ambiente persistente, no se marcan. Todo esto lo realiza con las funciones ***marca-objetos***, ***traspasa-marcados*** y ***revisa-objetos***.

6.3.5 Ev-unpersist

La función ***ev-unpersist*** se encarga de pasar al ambiente global la variable, objeto, función o clase que se encuentre en el ambiente parsistente y que el usuario desee *"borrar"* de éste, para lo cual lo primero que realiza es una verificación sintáctica pudiendo ocurrir uno de los dos mensajes siguientes:

1) "error-sintaxis-unpersist".

Este mensaje ocurre cuando no existe un ***nombre*** de por medio.

2) "error-sintaxis-exceso-de-parametros-en-unpersist".

Este mensaje se emite cuando existe además del ***nombre***, otros datos de por medio.

En el caso de que no suceda ninguno de los mensajes previos, puede ser que ocurra alguno de los siguientes:

3) "error-unpersist-variable-u-objeto-en-ambiente-local"
   ó
   "error-unpersist-variable-u-objeto-en-ambiente-global"

Si ocurre cualquiera de estos mensajes, significa que la variable u objeto no es persistente, ***no es error***, simplemente le avisa al usuario que ese elemento ya está en los ambientes volátiles.

Si no ocurre ninguno de los mensajes anteriores, significa que el ***nombre*** se encuentra en el ambiente persistente y se procede a pasarlo al ambiente global *"borrándolo"* del persistente.

6.3.6 Ev-begin, Ev-if, Ev-while

La función ***ev-begin*** evalúa todos los argumentos y regresa el último valor. Esta función es recursiva.

Por otro lado, la función *ev-if* evalúa el primer argumento; si lo evalúa a un valor verdadero (diferente de cero), entonces evalúa y regresa el valor del segundo argumento, de otro modo, evalúa y regresa el valor del tercer argumento.

Del mismo modo, la función ***ev-while***, evalúa repetidamente el primer argumento y luego los otros, hasta que el primero se evalúa a falso (cero), que es cuando sale del ciclo.

6.3.7 Ev-new, Ev-setshallow, Ev-quote

La función ***ev-new*** se encarga de crear un nuevo objeto de una clase predefinida haciendo uso de la función recursiva ***crea-objeto***. Si la clase ***esta-indefinida*** se emite un mensaje de error. Internamente el nuevo objeto es de la forma: ***(nobj1 (numero 0) nobj2 (numero 0) ... nobjn (numero 0))***. Así mismo, asocia la clase del objeto y crea una lista de ejemplos, generando toda la definición del objeto. Para realizar esto, ***ev-new*** hace uso también de otra función recursiva llamada ***cons-valor-obj.***

Por otro lado la función ***ev-setshallow*** se encarga de crear ***alias*** de objetos, es decir *"apuntadores"* o referencias a objetos ya existentes, sin replicar su contenido, generando con esto, en parte, el concepto de identidad. Cabe aclarar que no permite crear objetos alias en el ambiente persistente, porque debe recordarse una vez más, que solo ***persist*** permite modificar tal contexto.

Alternativamente a ***ev-setshallow***, tenemos la función ***ev-quote*** que se encarga de generar alias o apuntadores de objetos que no precisamente se encuentren en los ambientes.

6.3.8 Ev-optr

La estructura de la función ev-optr se puede apreciar en la Figura 6.8, su función es realizar la operación indicada, ya sea operaciones aritméticas

básicas (+, -, *, /), operaciones de relación (=, <, >), operaciones de manipulación de listas: ***(lista, car, cdr).*** También tiene operaciones para aplicar funciones, métodos y paso de mensajes. Así mismo, cuenta con la función ***print*** que permite imprimir el valor de las expresiones que lleva como párametro.

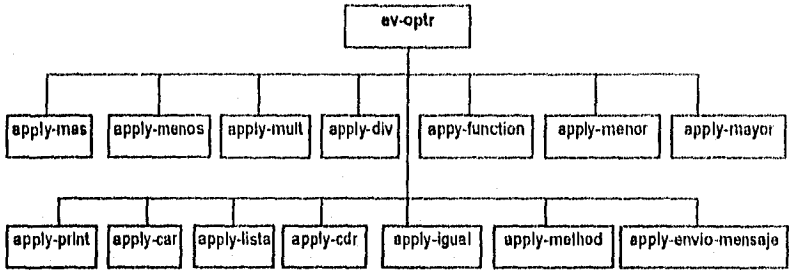


Figura 6.8 Diagrama de módulos de la función ***ev-optr.***

6.3.8.1 Apply-mas, Apply-menos, Apply-mult, Apply-div

Las funciones ***apply-mas***, ***apply-menos***, ***apply-mult*** y ***apply-div*** son operaciones binarias, que realizan lo obvio. La forma en que se implantaron es como sigue:

1) Primero se verifica que sólo existan dos operandos,

2) Después se evalúan los argumentos correspondientes a través de la función ***ev-exp***, creando nuevos ambientes,
3) Luego se verifica que los argumentos sean del tipo ***número***, para proceder al paso 4.

4) Se aplica la operación correspondiente (+, -, *, /) y se evalúa la expresión a través de la función ***ev-exp***, creando el ambiente actualizado.

5) Para la división (/), se verifica la división por cero.

6.3.8.2 Apply-igual, Apply-menor, Apply-mayor

Del mismo modo que las operaciones aritméticas, las operaciones de relación =, <, >, realizan pasos similares para su implantación.

1) Se verifica que sólo existan dos operandos,

2) Se evalúan los argumentos correspondientes a través de la función ***ev-exp***, creando nuevos ambientes,

3) Luego se verifica que los argumentos sean correctos, para proceder al paso 4.

4) Se aplica la operación correspondiente (=, <, >) y se evalúa la expresión. Si la comparación de los operandos resulta satisfactoria regresa como valor el ***tipo c*** y ***el valor 1*** que sirve para indicar que resultó verdadera la evaluación, en caso contrario se regresa el ***tipo c*** y el ***valor 0*** que nos indica que la evaluación resultó falsa, creando así el ambiente actualizado.

6.3.8.3 Apply-print

La función ***print*** despliega en pantalla el resultado, verificando que tenga argumentos y evaluando la expresión que recibe como parámetro, en el contexto que también se recibe como tal, a través de la función ***ev-exp***, creando un nuevo contexto. En caso de ser una lista, escribe su valor y los tipos de sus componentes, al igual que para cualquier caso, excepto que su resultado desde ***eval-leppooc***, es la lista de sus componentes sin sus tipos, ya que esto lo hace a través de ***eval-leppoc*** con una función expresa para este caso.

6.3.8.4 Apply-lista, Apply-car, Apply-cdr

Las funciones ***apply-lista***, ***apply-car***, ***apply-cdr*** son el constructor del tipo ***lista***, y sus operaciones de manipulación para obtener el primero y el resto de la lista respectivamente.

1) La función ***apply-lista*** revisa cada uno de los argumentos de la operación ***lista*** y los evalúa con la función ***ev-exp***, si están correctos construye la lista con cada nombre de los argumentos su tipo y valor, de manera recursiva a través de la función ***const-list***, para cada argumento de la operación lista.

2) La función ***apply-car*** verifica que la expresión que recibe como parámetro no sea nula y que solo sea un parámetro; después se evalúa la expresión, con ***ev-exp*** se obtiene el primero de la ***nueva-expresión*** y se llama a la función ***obtiene-car***, quien determina que el argumento sea una lista, de no ser así envía el mensaje: *"error-no-es-una-lista-el-*

parametro-de-car", pero si efectivamente es una lista se envía realmente el primero de ésta.

3) Por último, la función ***apply-cdr*** hace lo mismo que el ***apply-car***, con la diferencia de que ésta llama a ***obtiene-cdr*** quien verifica si el argumento es una lista, de ser así devuelve el resto de la lista, en caso contrario emite el mensaje: *"error-no-es-lista-el-parametro-de-cdr"*.

6.3.8.5 Es-funcion, Apply-function

La función ***es-función*** es un predicado que verifica si el nombre que recibe como parámetro es el nombre de una función definido en el ambiente; esto lo realiza llamando a la función recursiva ***esta-def-funcion,*** quien verifica que el tipo del nombre sea ***funcion***, de ser así se aplica la función ***apply-function***, a partir de ***ev-optr***.

La función ***apply-function*** recibe como parámetros una expresión y un ambiente. Si el nombre que se encuentra como primer elemento de la expresión es un ***nombre*** definido en el ambiente global, se procede a obtener el tipo y valor del mismo desde el ambiente global con la función ***asocia-tipo-valor***, y en caso de que el tipo sea ***funcion*** se evalúa la función a través de la función ***evalua-function***. Esta función recibe como parámetros la expresión, el valor del nombre de la función y el ambiente, y su objetivo es revisar la lista de variables o argumentos de la función a evaluar, para lo cual se vale de la función ***ev-argtos*** donde se actualiza el ambiente y aplica la función deseada.

6.3.8.6 Es-un-objeto, Apply-method

La función ***es-un-objeto*** recibe como parámetro un nombre y el ambiente. ***es-un-objeto*** es un predicado que verifica si el nombre es un objeto definido en los ambientes, de ser así regresa el valor *t*, en caso contrario regresa *nil*.

Apply-method recibe como parámetros una expresión y el contexto donde se evalúa, esta función tiene como objetivo aplicar el método de un objeto dado en la expresión. La Figura 6.9 muestra la estructura de ***apply-mhethod***.

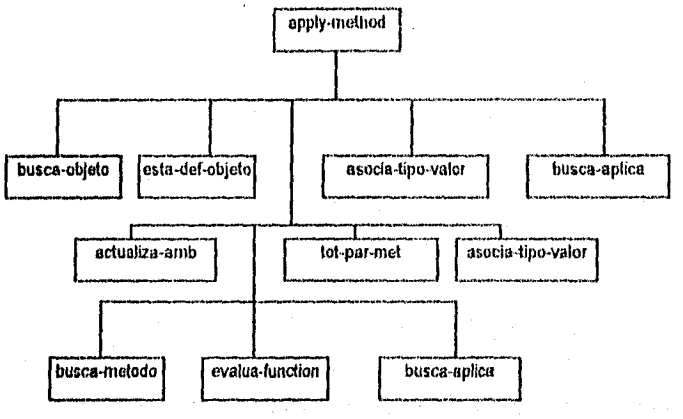


Figura 6.9 Diagrama de módulos de la función ***apply-method***.

Como se puede observar en la Figura 6.9 ***apply-method*** hace uso de seis módulos. La manera en la que se utilizan estas funciones es la siguiente.

1) Determina la longitud del ambiente local, ya que es aquí donde trabajará nuestra aplicación del método. Esto lo hace con ***tot-par-met***.

2) Busca el nombre del objeto en los ambientes a través de la función recursiva ***busca-objeto***,

3) Si no resultó ser nombre de objeto, entonces se envía el mensaje *"error-en-apply-method-no-es-nom-objeto"*.

4) Si resultó ser un nombre de objeto, entonces se busca en cada ambiente si se encuentra definido el nombre del objeto a través del predicado ***esta-def-objeto;*** una vez encontrado se obtiene el tipo y valor del nombre del objeto (con la función ***asocia-tipo-valor***), se crea un ambiente local auxiliar ***(n-amb-loc)*** con el ambiente local original y la lista de variables de instancia del método, tomada de la nueva pareja recién creada.

5) Una vez realizados los pasos anteriores se procede a aplicar el método a través de la función ***busca-aplica,*** quien se encarga de revisar que la definición del objeto no sea de la clase ***object*** y se procede a separar la clase del objeto usando la función ***asocia-tipo-valor.*** En seguida se busca el método con la función recursiva ***busca-metodo***, la cual se encarga de localizarlo en la lista de métodos (inicialmente, de la clase padre), que se encuentran definidos en la clase del objeto. Posteriormente, si lo encontró procede a crear un nuevo ambiente, agregando al ambiente local original el ambiente local auxiliar y dejando intactos los dos restantes y se procede a aplicar el método a través de la función ***evalua-funcion***, misma que es utilizada para evaluar funciones globales, descrita en la sección 6.3.7.5.

En caso de no haber encontrado el método procede a continuar la búsqueda del mismo llamando recursivamente a la función ***busca-aplica*** para localizarlo en sus subclases. Este proceso continúa hasta que se haya

encontrado el método o no, en este último caso se envía el mensaje de error: *"error-no-se-encontró-del-metodo"*.

6) ***Sexto***, en el caso de que no haya habido error al llamar ***busca-aplica***, se procede a dejar intacto el ambiente local, y se actualiza el ambiente global, creando así el ambiente actualizado después de haber aplicado el método.

6.3.8.7 Es-lista-mensajes, Apply-envio-mensaje

La función ***es-lista-mensajes*** recibe como parámetros una lista de mensajes ***(l-mensajes)*** y el ambiente ***(amb).*** Es un predicado que se encarga simplemente de verificar que el parámetro ***l-mensajes*** sea una lista, de serlo devuelve el valor ***t*** y de otro modo devuelve el valor ***nil***.

Si resultó verdadero ***(t)*** el predicado ***es-lista-mensajes***, el programa procede a llamar a la función ***apply-envio-mensaje*** que se encarga de aplicar cada uno de los métodos del ***objeto compuesto***. La forma en que lo hace es separando el mensaje inicial y verificando que el nombre del mensaje es un objeto a través del predicado ***es-un-objeto;*** de ser así, ***apply-envio-mensaje*** llama a la función ***apply-method*** para aplicar el método que envió el objeto. El resultado viene a ser otro objeto que se obtiene de la aplicación del método, ya que no se olvide que el primero es un ***objeto compuesto***, por lo que ahora, se vuelve a determinar que el segundo también sea un objeto y que el método exista. De ser así, se aplica el nuevo método usando la función ***apply-method*** con el nombre del método y sus variables de instancia, así como el nuevo contexto que resultó después de la aplicación del método anterior.

6.4 COMENTARIOS FINALES

Como se puede observar, el prototipo se encuentra estructurado de manera modular a través de pequeñas funciones recursivas. Por otro lado, también se puede apreciar que la función ***ev-exp*** es una función por demás recursiva y poderosa, por lo consiguiente vital, ya que es ésta la que se encarga de verificar el tipo de expresión introducida por el usuario, para que LEPPOOC la pueda interpretar. Entre las deficiencias de LEPPOOC, está quizá el diseño de la función ***print*** para algunos de los casos que esta función se encarga de imprimir, ya que cuando el valor de alguna variable que debe imprimir tiene valores anidados, la escribe con todo y sus tipos (el caso de imprimir un objeto o una lista). Sin embargo, se cree que esto es algo que se puede resolver sin mucho problema.

Otra deficiencia que tiene LEPPOOC es el diseño de la función ***ev-var***, ya que casi siempre devuelve un valor excepto cuando se utiliza ***quote*** o ***set-shallow*** donde regresa el nombre del objeto (apuntador), en lugar del valor de éste (la réplica).

CONCLUSIONES

En la actualidad se está viviendo un vertiginoso cambio en la tecnología informática, y cada vez es mayor el número de aplicaciones complejas que requieren de soluciones eficientes a través de la computadora.

Este trabajo se centró en las áreas de los lenguajes de programación y las bases de datos. Uno de los modelos de diseño que está causando polémica, es el ***modelo orientado a objetos***. A pesar de la popularidad relativamente reciente del modelo orientado a objetos, no existe una fundamentación formal que lo sustente, como sería el álgebra de conjuntos y relaciones del ***modelo relacional***. Sin embargo, el modelo orientado a objetos parece ser una alternativa de solución para los problemas a los que actualmente se enfrentan los desarrolladores de software.

Después de haber realizado este trabajo y construído un primer prototipo de Lenguaje de Programación Persistente Orientado a Objetos con manipulación de objetos Compuestos e identidad **(LEPPOOC)**,se mostraron las bondades de los conceptos del modelo orientado a objetos en un lenguaje de programación simple pero con persistencia. Esta idea de persistencia llega a los lenguajes de programación desde la perspectiva de bases de datos. Se

ha tratado de conservar una manipulación de objetos compuestos y se concluye lo siguiente:

◊ Se considera que el prototipo construido:

- Resultó en un lenguaje sencillo y funcional a la manera de LISP, que permite escribir con facilidad programas que muestran algunos de los conceptos orientados a objetos.
- Es un lenguaje de formato libre, maneja minúsculas y mayúsculas indistintamente.
- Incluye el concepto de persistencia de bases de datos.
- Maneja un mecanismo básico de objetos compuestos propuesto por KIM[2].

◊ Se sabe que existen muchos prototipos de lenguajes de programación y bases de datos orientados a objetos, sin embargo, en ocasiones es importante conocer como construirlos y así:

- Aprovechar las bondades del modelo orientado a objetos y categorizar los niveles con que es posible manejar la abstracción de objetos compuestos y de los lenguajes de programación orientados a objetos existentes, con mayor profundidad.
- Entender como se manipulan los objetos compuestos.
- Categorizar niveles de manipulación de objetos compuestos.
- Delinear los elementos que debe tener un sistema que soporte objetos compuestos.
- Definir como es la sintaxis y la interfaz de programación, para proporcionarle al programador o usuario las herramientas para usar con facilidad y flexibilidad los niveles de objetos compuestos.

◊ El código de LEPPOOC es en sí, una serie de ***listas***.

◊ El constructor de objetos ***lista*** y la expresión ***quote*** combinados, permiten construir valores que en apariencia hacen lo mismo que algunas expresiones de LEPPOOC; tal es el caso de la expresión que permite definir funciones ***(define)***, y que en los ambientes se guarda como sigue: ***(nombre-funcion (funcion (definicion)))***, donde ***definicion*** resulta de otra expresión válida en LEPPOOC. Con el constructor ***lista*** y la expresión ***quote*** se puede lograr un ambiente como éste, al usar expresiones como las siguientes: ***(set nombre-funcion (lista (quote (definicion))))***, dejando un ambiente parecido: ***(nombre-funcion (lista (alias (definicion))))***. Del mismo modo se podrían construir definiciones de clases, entre otras; no obstante, la limitante en LEPPOOC a diferencia de LISP, es que al evaluar la función o utilizar la clase, éstas no las podría reconocer como tales y causaría un error que LEPPOOC detectaría.

Se cree que esta sutileza podría resolverse modificando la interpretación de la definición creada en los ambientes por el constructor ***lista***, para que LEPPOOC la pueda interpretar adecuadamente. El hacerlo así, resultaría en las siguientes consecuencias:

- Simplificaría la sintaxis del lenguaje (en particular desaparecen ***define*** y ***class***).
- Perdería ortogonalidad.
- Ganaría regularidad.
- Se lograría la construcción dinámica de funciones y clases, permitiendo así, implantar el concepto de ***evolución del esquema***.

◊ Dentro de los defectos o limitaciones que se le pueden encontrar a esta primera versión de nuestro intérprete, se mencionan las siguientes:

- Una limitante de LEPPOOC es el hecho de no considerar los comentarios dentro de un programa escrito en éste, ya que no existe una expresión sintáctica que los reconozca.

- El no verificar si se alteran o no las palabras claves de LEPPOOC.

- Se considera que la expresión ***print*** podría modificarse, para que ésta *"escriba"* las listas y los objetos sin los tipos de sus componentes, sólo sus valores.

- La función ***ev-var***, que se encarga de determinar si una variable u objeto existe en los ambientes, devolviendo su tipo y valor inmediatos, no verifica si su valor es compuesto, lo que ocasiona que lo devuelva, pudiendo tener componentes que por consiguiente incluyen su tipo. El diseño de ***ev-var*** puede considerarse un *"defecto"* por el hecho de que LEPPOOC muestra la *"representación interna"* de los objetos o las listas, e inclusive *"esconde"* el valor de un ***alias***, ya que sólo permite ver el ***nombre del objeto***. Modificar esta función para que escriba el valor primitivo, es relativamente fácil, sin embargo, esto afectaría a otras funciones de LEPPOOC, como la que aplica los métodos, por lo que habría que tener cuidado de hacerlo cambiando también el diseño de esta función, misma que requiere de otros procesos.

  Por otro lado, cabe aclarar que se diseñó así por la conveniencia de utilizar la misma función que permite aplicar funciones definidas a nivel global, para aplicar los métodos de una clase, por lo que cuando se envía un mensaje a un ***objeto compuesto***, éste sólo toma el valor, que viene a ser el ***nombre del objeto*** que tiene el método que se quiere aplicar; permitiendo así su aplicación sin modificar el objeto compuesto directamente, sino el objeto real, respetando de este modo la identidad e integridad de objetos.

◊ De los dieciséis principios de diseño de lenguajes de programación que se discutieron en el capítulo cuatro, aunque LEPPOOC considera hasta cierto grado a cada uno, se pueden resaltar los principios que menos cubre.

- ***El costo de localidad*** es un principio que cumple someramente LEPPOOC, ya que no optimiza aspectos de eficiencia en la ejecución al verificar referencias colgantes cada vez que se desea abandonar una sesión de LEPPOOC.

- ***La estructura*** estática de un programa en LEPPOOC no corresponde a la estructura dinámica de los cálculos inherentes, ya que usa regla dinámica como LISP, ocasionando confusiones en los posibles valores que pueda tener algún objeto o variable utilizado, y que el usuario ya haya cambiado y no recuerde.

- ***La consistencia sintáctica***, es un principio que dice que las cosas similares se ven similares y las diferentes se ven diferentes. Esto significa que se debe evitar formas sintácticas que puedan convertirse en otras formas legales, sólo por un simple error. LEPPOOC puede incurrir en faltar a este principio, sobretodo en programas extensos, pues existen expresiones legales parecidas, como ***(lista)*** y ***lista*** que son diferentes pero se ven similares.

- ***Defensa en profundidad.*** LEPPOOC es pobre en este principio, ya que no cuenta con verificación de tipos.

Cabe aclarar que no fue motivo de preocupación el conseguir alcanzar los principios de diseño, fue algo colateral; ya que no era objetivo primordial crear un lenguaje de programación *"bien diseñado"*. Sin embargo, se considera que el diseño del prototipo cumple con los requerimientos mínimos de diseño discutidos en el capítulo cuatro.

Otro aspecto que conviene resaltar en este punto, es que el diseño de un lenguaje es labor de muchos años, no es fácil; y que un buen diseño no garantiza éxito comercial o de difusión.

◊ Uno de los objetivos principales de este trabajo, fue implantar objetos compuestos en el prototipo diseñado, se considera que se han puesto las

bases del modelo de objetos compuestos en LEPPOOC, dado que cuenta con:

- ***Referencia débil***, ya que incluye el concepto de herencia.
- ***Referencias compuestas dependientes exclusivas***, ya que la destrucción de un objeto compuesto, sólo ocurrirá si se destruye todas sus componentes que son dependientes de su existencia (por el constructor ***lista***).
- ***Referencias compuestas independientes compartidas***, ya que la destrucción de un objeto compuesto no implica borrar sus componentes.
- El sistema controla implícitamente las referencias compuestas.

◊ ***Trabajos futuros***: Como se sabe, el trabajo de diseño y construcción de un lenguaje es una ardua y larga tarea que involucra un gran equipo de profesionales, por lo que este prototipo puede ser mejorado y ampliado. Los trabajos que podrían desarrollarse a partir de este prototipo son:

1. Ampliar la semántica formal de todo LEPPOOC, utilizando la notación denominada ***semántica denotativa,*** misma que se empleó en este trabajo para explicar algunas expresiones de LEPPOOC.

2. Incluir en LEPPOOC los otros tipos de referencias compuestas descritas en este trabajo y propuestas por KIM[2], así como los conceptos de evolución del esquema discutidos.

3. Mejorar la implantación de algunos de los conceptos del modelo orientado a objetos que en LEPPOOC quedaron *"pobres"*, e incluir otros que no se consideraron en esta versión, tales como:

- La ***encapsulación***, ya que actualmente el usuario de una aplicación escrita en LEPPOOC, puede llegar a modificar la implantación de los métodos de la clase, violando así este concepto. Además, considerar alguna solución para limitar la visibilidad de los atributos con respecto a las subclases.

- Implantar más ***tipos***, ya que actualmente sólo cuenta con ***números*** (enteros y reales), ***cadenas*** y ***listas***. Podrían incluirse por ejemplo, conjuntos. Por otro lado, incluir un sistema de tipos para variables más que para objetos.

- Incluir el concepto de ***herencia múltiple***, ya que actualmente sólo se cuenta con ***herencia simple***. Aunque sea opcional la participación de herencia múltiple en el modelo, recordemos que puede resolver problemas conflictivos como los mostrados en el capítulo uno.

4. Ampliar el lenguaje para incluir otras características de bases de datos tales como: la concurrencia, recuperación de fallas en hardware y software, facilidad de consultas a la base de datos e integridad de transacciones.

REFERENCIAS BIBLIOGRAFICAS

[1] Atkinson, M., DeWitt, D., Maier, D."***The Object-Oriented Database System Manifesto***". In Proceedings of the first international DOOD Conference. 1990.

[2] Batini C., Ceri S., Navathe S. B. "***Conceptual Database Design: An Entity-Relationship Approach***". The Benjamin/Cummnings Publishing Company, Inc. 1992.

[3] Bertino E. Martino L. "***Sistemas de Bases de Datos Orientados a Objetos***". Addison- Wesley/Diaz de Santos. 1995.

[4] Booch, G. "***Object Oriented Design with Applications***". The Benjamin/Cummnings Publishing Company, Inc. 1991.

[5] Cardenas A. F., McLeod D. "***An Overview of Object-Oriented and Semantic Database Systems***". Prentice Hall Englewood Cliffs. 1990.

[6] Estivill, V. "***El panorama de las bases de datos orientadas a objetos***". Soluciones Avanzadas. Año 2, Num. 10, Junio 1994.

[7] Gordon M. J. C. "***The Denotational Description on Programming Languages. An Introduction***". Springer-Verlag, New York. 1979.

[8] Kamin, S., "***Programming Languages An Interpreter-Based Approach***". Addison-Wesley Publishing Company, Inc. 1990.

[9] Khoshafian S. "***Object-Oriented Databases***". John Wiley & Sons, Inc. 1993.

[10] Kim, W., Bertino E., Garza, J. "***Composite Objects Revised***". In Proc. ACM. Microelectronics and Computer Technology Corporation. Sigmod Int. Conf. on Management of Data, Portland, Oregon, Junio 1989.

[11] Kim, W. "***Introduction to Object-Oriented Databases***". The MIT Press. 1990.

[12] Pratt T. "***Programming Languages Design and Implementation***". Second Edition, Prentice Hall, 1984.

[13] MacLennan B. "***Principles of Programming Languages: Design, Evaluation, and Implementation***". CBS COLLEGE PUBLISHING. 1983.

[14] Sebesta, R. "***Concepts of Programming Languages***". The Benjamin/Cummings Publishing Company, Inc. 1989.

[15] "***Sun Common Lisp 4.0 User's Guide***". 1990.

[16] Tennet R.D. "***Principles of Programming Languages***". Prentice-Hall International, Inc. 1981.

[17] Winston, P. H., Horn, B. K. P. "***LISP***". Addisson-Wesley Iberoamericana. 3a. edición. 1991.

APENDICE A

Código de LEPPOOC en LISP

```
(in-package "USER")

;*****************************************************************************
;                        P R O T O T I P O                                    *
;                                                                             *
; LEPPOOC: LENGUAJE DE PROGRAMACION PERSISTENTE ORIENTADO A OBJETOS           *
;          CON OBJETOS COMPUESTOS E IDENTIDAD.                                *
; TESISTA: ALMA ROSA GARCIA GAONA                                             *
; DIRECTOR DE TESIS: DR. VLADIMIR ESTIVILL CASTRO                             *
;                                                                             *
;                             "leppooc"                                       *
;                                                                             *
;*****************************************************************************


(defun eval-leppooc ()
  (with-open-file (ape "arch-pers" :direction :input)
                  (setq ap (read ape nil))
  );with
  (close-all-files)
  (setq amb (list '() '(object (clase (object () ())) ap) )
  (loop
       (print 'LEPPOOC>)
       (setq prog (read))
    (let ((result (eval-input prog amb)))
         (setq amb (cons '() (cdadr result)))
       (cond
              ((eq (caar result) 'advertencia)
                   (print (cadar result)) )
          ((eq (caar result) 'fin)
                   (setq pe (caddr amb))
                   (with-open-file (aps "arch-pers" :direction :output)
                   (print pe aps) );
                   (close-all-files)
                   (return 'bye)                    )
              ((eq (caar result) 'lista)
                   (print (print-result (cadar result) '()) ))
            (t    (print (cadar result))   )
          )
       )
  )
)
;-----------------------------------------
; funcion: EVAL-INPUT                    |
;-----------------------------------------
```

```
(defun eval-input (prog amb)
  (cond
      ((atom prog)                    (ev-exp prog amb)                    )
      ((and (not (atom prog))
            (eq (car prog) 'bye)
            (null (cdr prog)) )    (ev-bye amb)                    )
    ((and (not (atom prog))
            (eq (car prog) 'persist-class)
            (null (cdr prog)) )    (ev-persist-class amb (caddr amb) (caddr amb)))
      ((eq (car prog) 'define)   (ev-fundef-global (cdr prog) amb) )
      ((eq (car prog) 'class)             (ev-class (cdr prog) amb)       )
      ((eq (car prog) 'read)              (ev-archivo (cdr prog) amb)      )
      (t                                  (ev-exp prog amb)               )
  )
)
;-----------------------------------------
; funcion: EV-BYE                  |
;-----------------------------------------
(defun ev-bye (amb)
  (cond
      ((and (hay-clases-ag (cadr amb))
            (not (objp-con-clase-ambp (caaddr amb) (caddr amb) (caddr amb))))
                  (list (list 'ADVERTENCIA 'advertencia-probablemente-hay-objetos-persistentes-
con-sus-clases-en-ambiente-global-vuelvalas-persistentes-o-borre-los-objetos-del-amb-persist)
amb))
      ((and (hay-alias-ag (cadr amb))
            (not (objp-con-alias-ambp (caddr amb) (caddr amb))) )
                  (list (list 'ADVERTENCIA 'ADVERTENCIA-PROBABLEMENTE-HAY-
OBJETOS-PERSISTENTES-CUYOS-TIPOS-DE-SUS-COMPONENTES-SE-ENCUENTRAN-
EN-AG-VUELVALOS-PERSISTENTES-O-BORRE-LOS-OBJETOS-DEL-AMB-PERS) amb) )
      ((and (hay-obj-ag (cadr amb))
            (not (objp-con-alias-ambp (caddr amb) (caddr amb))) )
                  (list (list 'ADVERTENCIA 'ADVERTENCIA-PROBABLEMENTE-HAY-
OBJETOS-EN-AMB-GLOBAL-QUE-SON-ALIASES-DE-OTROS-EN-AMB-PERS-BORRE-LOS-
PERSISTENTES-O-SALVE-LOS-DEL-GLOBAL) amb))
      (t          (list (list 'fin 'bye) amb))
  )
); ev-bye
;
;-----------------------------------------
;funcion: HAY-CLASES-AG     |
;-----------------------------------------
;
(defun hay-clases-ag (amb)
  (cond
      ((null amb)       nil)
      ((and (eq (caadr amb) 'clase)
            (not (eq (car amb) 'object)) )          t)
      (t          (hay-clases-ag (cddr amb)) )
  )
); hay-clases-ag
;
;-----------------------------------------
; funcion: HAY-ALIAS-AG        |
;-----------------------------------------
;
(defun hay-alias-ag (amb)
  (cond
      ((null amb)       nil)
      ((eq (caadr amb) 'ALIAS)          t)
      (t          (hay-alias-ag (cddr amb)))
```

```
  )
); hay-alias-ag

;-----------------------------------------
; funcion: HAY-OBJ-AG          |
;-----------------------------------------
(defun hay-obj-ag (amb)
  (cond
        ((null amb)      nil)
        ((and (not (eq (caadr amb) 'funcion))
              (not (eq (caadr amb) 'clase))
              (not (eq (caadr amb) 'error)) )    t)
        (t (hay-obj-ag (cddr amb)))
  )
);hay-obj-ag
;-----------------------------------------
; funcion: OBJP-CON-CLASE-AMBP   |
;-----------------------------------------

(defun objp-con-clase-ambp (obj amb amborig)
  (cond
        ((null amb)      t)
        ((esta-def-objeto obj amborig)  ; se determina si el elemento es un objeto

; SE DETERMINA SI EXISTE LA CLASE DEL OBJETO Y DE SUS PARTES EN EL
AMBIENTE PERSISTENTE
                (let ((valor-obj (asocia-tipo-valor obj amborig)))
                  (cond
                   ((existe-clase-objeto valor-obj valor-obj amborig)
                                (objp-con-clase-ambp (caddr amb) (cddr amb) amborig))
                        (t                                  nil)) ))
        (t      (objp-con-clase-ambp (caddr amb) (cddr amb) amborig))
  )
); obj-p-con-clase-amp

;-----------------------------------------
; funcion: EXISTE-CLASE-OBJETO   |
;-----------------------------------------

(defun existe-clase-objeto (partes parte-ant amb)
  (cond
        ((null partes)  t)
        ((and (eq (car partes) 'alias)
              (esta-def-obj-alias (cadr partes) amb) )
                (existe-clase-objeto (cddr parte-ant) (cddr parte-ant) amb))
        ((or (eq (car partes) 'numero)
             (eq (car partes) 'cadena)
             (eq (car partes) 'lista))
                (existe-clase-objeto (cddr parte-ant) (cddr parte-ant) amb))
        ((existe-clase-base-y-derivadas (car partes) amb)
                (existe-clase-objeto (cadadr partes) parte-ant amb))
        (t                  nil)
  )
); existe-clase-objeto

;-----------------------------------------
; funcion: ESTA-DEF-OBJ-ALIAS      |
;-----------------------------------------

(defun esta-def-obj-alias (objeto amb)
  (cond
```

```
        ((esta-def-objeto objeto amb)
          (let ((v-obj (asocia-tipo-valor objeto amb)))
               (cond
                  ((existe-clase-objeto v-obj  v-obj amb)        t)
                  (t                                             nil))))
        (t        nil)
   )
); esta-def-obj-alias
;
;-----------------------------------------------------
; funcion: EXISTE-CLASE-BASE-Y-DERIVADAS !
;-----------------------------------------------------
;
(defun existe-clase-base-y-derivadas (n-class amb)
   (cond
        ((null amb)                  nil)
        ((esta-def-nom-class n-class amb)
          (let ((v-clase (asocia-tipo-valor n-class amb)))
               (cond
                  ((and (eq (car v-clase) 'CLASE)
                        (not (eq (caadr v-clase) 'OBJECT)))
                             (existe-clase-base-y-derivadas (car v-clase) amb))
                  (t                         t) )))
        (t                nil)
   )
); existe-clase-base-y-derivadas
;
;-----------------------------------------------
; funcion: OBJP-CON-ALIAS-AMBP     !
;-----------------------------------------------
;
(defun objp-con-alias-ambp (ambp amborig)
  (cond
        ((null ambp)     t)
        ((and (eq (caadr ambp) 'ALIAS)
              (esta-def-objeto (cadadr ambp) amborig))
          (objp-con-alias-ambp (cddr ambp) amborig))
        ((and (eq (caadr ambp) 'ALIAS)
              (not (esta-def-objeto (cadadr ambp) amborig)))        nil)
        ((revisa-partes-objeto (cadadr ambp) amborig)
              (objp-con-alias-ambp (cddr ambp) amborig))
        (t  nil)
  )
); objp-con-alias-ambp
;
;-----------------------------------------------
; funcion: REVISA-PARTES-OBJETO   !
;-----------------------------------------------
;
(defun revisa-partes-objeto (partes amb)
  (cond
        ((null partes)     t)
        ((and (atom partes)
              (or (numberp partes)
                   (cadenap partes)))      t)
        ((and (eq (caadr partes) 'ALIAS)
              (or (not (esta-def-objeto (cadadr partes) amb))
                   (not (esta-def-nom (cadadr partes) amb)))) nil)
        (t         (revisa-partes-objeto (cddr partes) amb))
  )
); revisa-partes-objeto
```

```
;-------------------------------------
; funcion: EV-PERSIST-CLASS           |
;-------------------------------------

(defun ev-persist-class (amb ambp ambpo)
  (cond
        ((null ambp)     (list (list 'fin 'clases-de-objetos-persistentes-salvadas) amb))
        ((and (esta-def-objeto (car ambp) ambpo)
              (not (existe-clase-objeto (cadr ambp) (cadr ambp) ambpo)) )
                  (let* ((n-a (cadr (ev-unpersist (list (car ambp)) amb)))
                    (amb-act (cadr (ev-persist (list (car ambp)) n-a))))
                          (ev-persist-class amb-act (cddr ambp) ambpo) ) )
        ((and (esta-def-objeto (car ambp) ambpo)
              (not (revisa-partes-objeto (cadr ambp) ambpo)) )
                  (let* ((n-a (cadr (ev-unpersist (list (car ambp)) amb)))
                    (amb-act (cadr (ev-persist (list (car ambp)) n-a))))
                          (ev-persist-class amb-act (cddr ambp) ambpo) ) )
        (t        (ev-persist-class amb (cddr ambp) ambpo))
  )
)
;-------------------------------------
; funcion: EV-ARCHIVO                 |
;-------------------------------------

(defun ev-archivo (nomb-arch amb)
  (cond ((and(null (cdr nomb-arch))
             (atom (car nomb-arch))
             (not (symbolp (car nomb-arch)))
             (not (numberp (car nomb-arch))))
    (setq na (car nomb-arch))
    (with-open-file (pp na :direction :input)
          (loop
            (setq prog-arch (read pp nil))
            (cond ((and (atom prog-arch)
                        (null prog-arch))
                   (return (setq result (list (list 'bye 'bye) amb))))
                  (t    (let ((result (eval-input prog-arch amb)))
                              (setq amb (cons '() (cdadr result)))
                              (cond
                                    ((eq (caar result) 'advertencia)
                                          (print (cadar result)) )
                                ((eq (caar result) 'bye)
                                          (return (setq result (list (list 'bye 'bye) amb)))  )
                                          ((eq (caar result) 'lista)
                                                (print (print-result (cadar result) '()) ) )
                                  (t     (print (cadar result))          )
                                ))))
          )
        )
    ); with
    ); and
    (t              (list (list 'error 'error-nomb-arch) amb))
  ); cond
); ev-archivo

;-------------------------------------------------------------------
; funcion: EV-CLASS                                                 |
; def-class = (n-class1 n-class2 inst-var methoddef+)               |
; (car def-class)   = n-class1                                      |
; (cadr def-class)  = n-class2                                      |
```

```
; (caddr def-class)  = inst-var                                    |
; (cadddr def-class) = methoddef+                                  |
; l-def-fun-c        = VALOR                                       |
; (car amb)          = LE                                          |
; (cadr amb)         = GE                                          |
; (caddr amb)        = PE                                          |
;-----------------------------------------------------------------
;
(defun ev-class (def-class amb)
  (cond
        ((null def-class)           (list (list  'error 'error-la-palabra-clave-class-debe-estar-seguida-
por-la-def ) amb))
        ((< (length def-class) 4)
              (list (list 'error 'error-faltan-argumentos-en-la-def-clase ) amb))
        ((not (symbolp (car def-class)))
             (list (list 'error 'error-sintaxis-debe-ser-nombre-de-clase-a-definir) amb))
        ((or (eq (car def-class) 'numero)
             (eq (car def-class) 'object)
             (eq (car def-class) 'cadena)
             (eq (car def-class) 'lista)
             (eq (car def-class) 'alias)
             (eq (car def-class) 'funcion))
                (list (list 'error 'error-en-def-class-no-se-permite-alterar-tipos) amb))
        ((not (symbolp (cadr def-class)))
             (list (list 'error 'error-sintaxis-debe-ser-nombre-de-una-superclase) amb))
        ((not (es-lista-var (caddr def-class)))
              (list (list 'error 'error-sintaxis-no-es-lista-inst-var) amb))
        ((not (as-methoddef (cdddr def-class)))
              (list (list 'error 'error-sintaxis-no-es-lista-methoddef+) emb))
        ((and (not (esta-def-nom-class (cadr def-class) (cadr amb)))
            (not (esta-def-nom-class (cadr def-class) (caddr amb))) );and
              (list (list 'error 'error-sintaxis-superclase-nclass2-no-esta-definida) amb))
        ((or (esta-def-nom-class (car def-class) (cadr amb))
            (esta-def-nom-class (car def-class) (caddr amb)) ); or
              (list (list 'error 'error-clase-con-nombre-repetido) amb))
        ((or (esta-def-nom-class (cadr def-class) (cadr amb))
            (esta-def-nom-class (cadr def-class) (caddr amb)) ); or
              (let* ((lista (list (cadr def-class) (caddr def-class)))
                     (l-def-fun-c (ev-fundef-class (cdddr def-class)))
                     (VALOR (append lista l-def-fun-c)))
                    (cond ((eq (car l-def-fun-c) 'error) (list (list 'error 'error-def-methodos-
duplicados) amb))
                          (t
                              (list (list 'clase VALOR)
                                (list (car amb)
                                      (actualiza-amb (car def-class) 'clase VALOR (cadr amb))
                                      (caddr amb)
                                )
                              )
                          )
                    )
              )
        )
        (t      (list (list 'error 'error-en-def-de-la-clase) amb))
  );cond
); ev-class
;
;-----------------------------------
; funcion: ES-LISTA-VAR          |
;-----------------------------------
;
```

```
(defun es-lista-var (exp)
   (cond
         ((null exp)                              t  )
         ((atom exp)                              nil )
      ((and  (symbolp (car exp))
             (es-lista-var (cdr exp)) )          t  )
         (t                                            nil )
   ); cond
); es-lista-var

;-----------------------------------------------
; FUNCION: ESTA-DEF-NOM-CLASS      !
;-----------------------------------------------
(defun esta-def-nom-class (nom-class amb)
   (cond
         ((null amb)                                   nil)
         ((and (eq (caadr amb) 'clase)
               (eq (car amb) nom-class))               t )
         (t                  (esta-def-nom-class nom-class (cddr amb)))
   ); cond
); esta-def-nom-class

;-----------------------------------------
; funcion: ES-METHODDEF        !
;-----------------------------------------

(defun es-methoddef (exp)
   (cond
         ((null exp)      nil)
         ((null (car exp)) nil)
         (t              (ev-methoddef exp))
   ); cond
); es-methoddef

;-----------------------------------------

(defun ev-methoddef (exp)
   (cond
         ((null (car exp))           t)
         ((not (eq (caar exp) 'define)) nil)
         ((es-defun-clase (car exp))    (ev-methoddef (cdr exp)))
         (t                                nil)
   );cond
); ev-methoddef

;-----------------------------------------

(defun es-defun-clase (exp)
   (cond
         ((null (cdr exp))              nil)
         ((not (eq 3 (length (cdr exp))))  nil)
         ((not (symbolp (cadr exp)))          nil)
         ((es-lista-var (caddr exp))      t  )
         (t                                nil)
   );cond
); es-defun-clase

;-----------------------------------------

(defun ev-fundef-class (exp)
   (let ((listamet '()))
```

```
        (cons-lista-met exp listamet)
    ); let
); ev-fundef-class

;-----------------------------------------

(defun cons-lista-met (fundef l-met)
    (cond
        ((null (car fundef))                l-met)
        ((member (cadar fundef) l-met)  (list 'error l-met))
        (t          (cons-lista-met (cdr fundef) (inserta-metodo (car fundef) l-met)))
    ); cond
); cons-lista-met

;-----------------------------------------
(defun inserta-metodo (metodo l-met)
  (let ((met-def (list (cadr metodo)
                       (list 'funcion (cddr metodo))
                   ); list
        ))
        (append l-met met-def)
  );let
); inserta-metodo

;-----------------------------------------
; funcion: EV-EXP                 |
;-----------------------------------------

(defun ev-exp (exp amb)
    (cond
        ((numberp exp)                          (ev-numero exp amb)     )
        ((cadenap exp)                      (ev-cadena exp amb)     )
      ((atom exp)                       (ev-var exp amb)            )
        ((eq (car exp) 'set     )               (ev-set (cdr exp) amb)      )
        ((eq (car exp) 'persist )       (ev-persist (cdr exp) amb)  )
        ((eq (car exp) 'unpersist)              (ev-unpersist (cdr exp) amb) )
        ((eq (car exp) 'new     )               (ev-new (cdr exp) amb)      )
        ((eq (car exp) 'set-shallow)            (ev-setshallow (cdr exp) amb))
        ((eq (car exp) 'if      )       (ev-if (cdr exp) amb)       )
        ((eq (car exp) 'while   )       (ev-while (cdr exp) amb)    )
        ((eq (car exp) 'begin   )               (ev-begin (cdr exp) amb)    )
        ((eq (car exp) 'quote   )       (ev-quote exp amb)          )
        (t                                      (ev-optr exp amb)           )
    )
)

;-----------------------------------------
; funcion: CADENAP                  |
;                                   |
; Es un predicado que verifica si la    |
; EXP es una cadena                 |
;                                   |
;-----------------------------------------

(defun cadenap (exp)
  (cond
        ((symbolp exp)                  nil)
        ((listp exp)                nil)
        ((and (atom exp)
              (eq (elt exp 0) '#\#))    t)
        (t                              nil)
```

```
  )
); cadenap
;
;-----------------------------------------------
; funcion: EV-NUMERO                           !
;                                              !
; (list 'numero exp) = (TIPO VALOR)            !
; TIPO = NUMERO                                !
;                                              !
;-----------------------------------------------
;
(defun ev-numero (exp amb)
  (list (list 'numero exp) amb)
);ev-numero
;
;-----------------------------------------------
; funcion: EV-CADENA                           !
;                                              !
; (list 'cadena exp) = (TIPO VALOR)            !
; TIPO = CADENA                                !
;-----------------------------------------------
;
(defun ev-cadena (exp amb)
  (list (list 'cadena exp) amb)
);ev-cadena
;
;-----------------------------
; funcion: EV-VAR             !
;-----------------------------
(defun ev-var (variable amb); amb= (LE GE PE)
  (cond
; VERIFICA SI LA VARIABLE ESTA INDEFINIDA
;
        ( (and (not (esta-def-nom variable (car amb)))
               (not (esta-def-nom variable (cadr amb)))
               (not (esta-def-nom variable (caddr amb))))
                      (list (list 'error 'variable-indefinida) amb)
        )
; VERIFICA SI LA VARIABLE SE ENCUENTRA EN: LE
        ( (esta-def-nom variable (car amb))
                (list (asocia-tipo-valor variable (car amb)) amb)
        )
; VERIFICA SI LA VARIABLE SE ENCUENTRA EN: GE
        ( (esta-def-nom variable (cadr amb))
                (list (asocia-tipo-valor variable (cadr amb)) amb)
        )
; LA VARIABLE ESTA EN: PE
        (t      (list (asocia-tipo-valor variable (caddr amb)) amb)
        )
  );cond
);ev-var
;
;-----------------------------------------------
; funcion: ASOCIA-TIPO-VALOR                   !
;-----------------------------------------------
;
(defun asocia-tipo-valor (nom-var a)
  (cond
        ((eq (car a) nom-var)       (cadr a))
        (t          (asocia-tipo-valor nom-var (cddr a)))
  )
```

```
); asocia-tipo-valor

;-----------------------------------------------
; funcion: EV-SET                               |
;                                               |
; (car exp) = IDENTIFICADOR                     |
; tipo: TIPO DEL IDENTIFICADOR                  |
; valor: VALOR DEL IDENTIFICADOR                |
; (car nuevo-amb) = LE                          |
; (cadr nuevo-amb) = GE                         |
; (caddr nuevo-amb) = PE                        |
;                                               |
;-----------------------------------------------

(defun ev-set (exp amb) ; amb = (LE GE PE)
  (let*
        (       (nueva-pareja (ev-exp (cadr exp) amb))
                (tipo (caar nueva-pareja))
                (valor (cadar nueva-pareja))
                (result-tv (car nueva-pareja))
                (nuevo-amb (cadr nueva-pareja))
        )
        (cond
            ((eq tipo 'error) nueva-pareja)
            ((esta-def-nom (car exp) (car nuevo-amb)) ; Si esta def. en LE
                (list
                 result-tv
                 (list
                  (actualiza-amb (car exp) tipo valor (car nuevo-amb))
                  (cadr nuevo-amb)
                  (caddr nuevo-amb)
                 )
                )
            ); busqueda en LE
            ((esta-def-nom (car exp) (caddr nuevo-amb)); Si esta def. en PE
             (list (list 'error 'ERROR-NO-SE-PUEDE-MODIFICAR-AMB-PERS-CON-SET )
nuevo-amb)
            ); busqueda en PE
            ((no-es-object (car exp) (cadr nuevo-amb)); Si esta def en GE o en ningun amb
                (list
                  result-tv
                  (list
                   (car nuevo-amb)
                   (actualiza-amb (car exp) tipo valor (cadr nuevo-amb))
                   (caddr nuevo-amb)
                  ); list
              );cons
            ); busqueda en GE
            (t                          (list (list 'error 'error-sintaxis-set-clase-object-no-se-redefine)
emb))
        ); COND
  ); LET*
); EV-SET

;-----------------------------------
; funcion: ESTA-DEF-NOM      |
;-----------------------------------

(defun esta-def-nom (nombre amb)
   (cond
        ( (null amb)                    '()     )
```

```
        ( (eq (car amb) nombre)          t      )
        (t          (esta-def-nom nombre (cddr amb)))
   ); cond
); esta-def-nom

;-------------------------------------------------------------
; funcion: NO-ES-OBJECT                                      |
; comentario: se busca que no sea la palabra OBJECT          |
;-------------------------------------------------------------

(defun no-es-object (nombre amb)
   (cond
        ((null amb)                          t)
        ((eq nombre 'object)              nil)
        ((eq (car amb) nombre)            t )
        (t          (no-es-object nombre (cddr amb)))
   ); cond
); es-object
;-----------------------------------------
; FUNCION: ACTUALIZA-AMB          |
;-----------------------------------------

(defun actualiza-amb (nom tipo valor amb)
   (cond

; SI EL AMBIENTE ES VACIO SE CREA CON EL IDENTIFICADOR Y SUS ATRIBUTOS

        ((null amb)              (list nom (list tipo valor))  )

; SE AGREGA EL NUEVO IDENTIFICADOR EN EL AMBIENTE

        ((eq (car amb) nom)
                                 (append (list nom (list tipo valor))
                                         (cddr amb))  )
; SIGUE BUSCANDO EL IDENTIFICADOR
        (t                       (append (list (car amb) (cadr amb))
                                         (actualiza-amb nom tipo valor (cddr amb))) )
   );cond
);actualiza-amb

;-----------------------------
; funcion: EV-PERSIST        |
;-----------------------------

(defun av-persist (exp amb)
   (cond
        ((null exp) (list (list 'error 'error-sintaxis-exp-debe-ser-variable-u-objeto ) amb) )
        ((> (length exp) 1) (list (list 'error 'ERROR-SINTAXIS-EXCESO-DE-PARAMETROS-EN-
PERSIST ) amb))
        ((and (not (esta-def-nom (car exp) (car amb)))
              (not (esta-def-nom (car exp) (cadr amb)))
              (not (esta-def-nom (car exp) (caddr amb)))  )
                        (list (list 'error 'variable-u-objeto-no-definido ) amb))
        ((esta-def-nom (car exp) (car amb))
                (list (list 'error 'error-en-persist-variable-u-objeto-en-ambiente-local ) amb))
        ((esta-def-nom (car exp) (caddr amb))
            (list (list 'error 'variable-u-objeto-ya-existente-en-amb-persistente) amb))
        ((let ((var (esta-def-var (car exp) amb)))
                var))
        ((es-un-objeto (car exp) amb)
                (let* ((amb-ob)-marcados (marca-objetos (car exp) amb))
```

```
                    (amb-sin-obj-marc (traspasa-marcados amb-obj-marcados (cadr amb-obj-
marcados)) )
                    (nueva-pareja (asocia-tipo-valor (car exp) (caddr amb-sin-obj-marc)))
                    (nva-exp (list (car nueva-pareja)) )
                    (list nueva-pareja (revisa-clases nva-exp amb-sin-obj-marc)) )
  )
);ev-persist
;
;-------------------------------------
; funcion: REVISA-CLASES     |
;-------------------------------------
;
(defun revisa-clases (exp-orig amb)
  (cond
      ((esta-def-nom (car exp-orig) (car amb))
        (let* ((npar-orig (asocia-tipo-valor (car exp-orig) (car amb)) )
               (nva-exp (list (caadr npar-orig))))
              (cond
                    ((eq (car exp-orig) 'object)        amb)
                    ((es-una-clase (car exp-orig) amb)
                      (let ((n-amb (list
                                        (elimina (car exp-orig) (car amb))
                                  (cadr amb)
                                    (agrega-nuevo (car exp-orig) npar-orig (caddr amb)))) )
                            (revisa-clases nva-exp n-amb)
                      ))
                      (t        amb)
              )))
      ((esta-def-nom (car exp-orig) (cadr amb))
        (let* ((npar-orig (asocia-tipo-valor (car exp-orig) (cadr amb)) )
               (nva-exp (list (caadr npar-orig))) )
              (cond
                    ((eq (car exp-orig) 'object)        amb)
                    ((es-una-clase (car exp-orig) amb)
                      (let ((n-amb (list (car amb)
                                        (elimina (car exp-orig) (cadr amb))
                                  (agrega-nuevo (car exp-orig) npar-orig (caddr amb)))) )
                            (revisa-clases nva-exp n-amb)
                      ))
                      (t        amb)
              )))
      ((esta-def-nom (car exp-orig) (caddr amb))
        (let* ((npar-orig (asocia-tipo-valor (car exp-orig) (caddr amb)) )
               (nva-exp (list (caadr npar-orig))) )
              (cond
                    ((eq (car exp-orig) 'object)        amb)
                    ((es-una-clase (car exp-orig) amb)
                      (revisa-clases nva-exp amb))
                    (t        amb)
              )))

      (t        amb)
  )
); revisa-clases
;
;-------------------------------------
; funcion: ESTA-DEF-VAR       |
;-------------------------------------
;
(defun esta-def-var (variable amb)
  (let ((np (asocia-tipo-valor variable (cadr amb))))
```

```
        (cond
              ((or (eq (car np) 'cadena)
                   (eq (car np) 'numero)
                   (eq (car np) 'lista)
                   (eq (car np) 'funcion)
                   (eq (car np) 'clase))
                      (list np
                            (list (car amb)
                                  (elimina variable (cadr amb))
                                  (agrega-nuevo variable np (caddr amb)) )))
               (t   nil)
              );cond
    );let
);esta-def-var

;-------------------------------------
; funcion: MARCA-OBJETOS   |
;-------------------------------------

(defun marca-objetos (obj amb)
    (let* ((np (asocia-tipo-valor obj (cadr amb)) )
           (objeto-tipo-val (list obj np))
           (nvo-amb-marc (list (car amb)
                               (pega-marca objeto-tipo-val (cadr amb))
                               (caddr amb))) )
        (cond ((eq (car np) 'alias)
               (persist-partes-obj np np nvo-amb-marc) )
          (t    (persist-partes-obj (cadr np) (cadr np) nvo-amb-marc))
          ); cond
    ); let*
); marca-objetos

;-------------------------------------
; funcion: PEGA-MARCA          |
;-------------------------------------

(defun pega-marca ( obj-a-marc ag)
    (cond
          ((null ag)          nil)
          ((eq (car ag) (car obj-a-marc))
                (append (list '-PERSIST obj-a-marc) (cddr ag)) )
          (t          (append (list (car ag) (cadr ag))
                              (pega-marca obj-a-marc (cddr ag))))
    ); cond
); pega-marca

;-------------------------------------
; funcion: PERSIST-PARTES-OBJ       |
;-------------------------------------

(defun persist-partes-obj (parte-ant partes amb)
  (cond
          ((null partes)     amb)
          ((eq (car partes) 'alias)
                (let ((valor-persist  (ev-persist (list (cadr partes)) amb)) )
                   (persist-partes-obj (cddr parte-ant) (cddr parte-ant) (cadr valor-persist))
                ))
          ((or (eq (car partes) 'numero)
               (eq (car partes) 'cadena))
                (persist-partes-obj (cddr parte-ant) (cddr parte-ant) amb))
          (t          (persist-partes-obj partes (cadr partes)  amb))
```

```
  );cond
); persist-partes-obj

;-----------------------------------------------
; funcion: TRASPASA-MARCADOS     !
; amb = ambiente global          !
;-----------------------------------------------

(defun traspasa-marcados (amb ambg)
  (cond
      ((null ambg)   amb)
      ((eq (car ambg) '-persist)
           (let ((nvo-amb
                        (list (car amb)
                              (elimina (car ambg) (cadr amb))
                              (agrega-nuevo (caadr ambg) (cadadr ambg) (caddr amb)) )))
                (traspasa-marcados nvo-amb (cddr ambg))))
      (t        (traspasa-marcados amb (cddr ambg)))
  ); cond
); traspasa-marcados

;-----------------------------------------
; funcion: ES-UNA-CLASE       !
;-----------------------------------------

(defun es-una-clase (nom-c amb)
  (cond
      ((listp nom-c)              nil)
      ((not (symbolp nom-c)) nil)
      ((busca-tipo-clase nom-c (car amb))     t)
      ((busca-tipo-clase nom-c (cadr amb))    t)
      ((busca-tipo-clase nom-c (caddr amb))   t)
      (t                                      nil)
  )
); es-una-clase

;-----------------------------------------------
; funcion: BUSCA-TIPO-CLASE          !
;-----------------------------------------------

(defun busca-tipo-clase (nom-c amb)
  (cond
      ((null amb)                 nil)
      ((eq (car amb) nom-c)
          (cond
              ((eq (caadr amb) 'clase)  t)
              (t                        nil)
          ))
      (t        (busca-tipo-clase nom-c (cddr amb)) )
  )
); busca-tipo-clase

;-----------------------------------------
; funcion: AGREGA-NUEVO     !
;-----------------------------------------

(defun agrega-nuevo (nom tip-val amb)
  (cond
      ((null amb)       (list nom tip-val))
      (t                (append (list nom tip-val) amb))
  ); cond
```

```
); agrega-nuevo

;-----------------------------------
; funcion: ELIMINA                  |
;-----------------------------------

(defun elimina (nom amb)
  (cond
        ((null amb)                 '())
        ((eq (car amb) nom)    (cddr amb))
        (t                          (append (list (car amb) (cadr amb))
                                            (elimina nom (cddr amb))) );append
  ); cond
); elimina

;-----------------------------------------------
; funcion: EV-UNPERSIST                        |
; (car exp): NOMBRE                            |
; (caar nueva-pareja): TIPO                    |
; (cadar nueva-pareja): VALOR                  |
; (cadr amb): GE                               |
;-----------------------------------------------

(defun ev-unpersist (exp amb)
   (cond
        ((null exp)         (list (list 'error 'error-sintaxis-unpersist ) amb))
        ((> (length exp) 1) (list (list 'error 'ERROR-SINTAXIS-EXCESO-DE-PARAMETROS-EN-
UNPERSIST ) amb))
        ((and (not (esta-def-nom (car exp) (car amb)))
              (not (esta-def-nom (car exp) (cadr amb)))
              (not (esta-def-nom (car exp) (caddr amb)))  )
                    (list (list 'error 'variable-u-objeto-indefinido ) amb)  )
        ((esta-def-nom (car exp) (car amb))
                (list (list 'error 'error-unpersist-variable-u-objeto-en-ambiente-local ) amb) )
        ((esta-def-nom (car exp) (cadr amb))
                (list (list 'error 'error-unpersist-variable-u-objeto-en-ambiente-global ) amb) )
        ((esta-def-nom (car exp) (caddr amb) )
                (let ((nueva-pareja (list (asocia-tipo-valor (car exp) (caddr amb)) amb)))
                     (list (car nueva-pareja)
                           (list (car amb)
                                 (actualiza-amb (car exp) (caar nueva-pareja) (cadar nueva-
pareja) (cadr amb))
                                 (elimina (car exp) (caddr amb)))) )
                ); let
   ); cond
); ev-unpersist

;-------------------------
; funcion: EV-NEW        |
;-------------------------

(defun ev-new (exp amb)
  (cond
        ((null exp)         (list (list 'error 'error-new-falta-nombre-clase) amb))
        ((not (null (cdr exp)))
                            (list (list 'error 'error-exceso-de-parametros-en-new) amb))
        ((or (esta-def-nom-class (car exp) (cadr amb))
             (esta-def-nom-class (car exp) (caddr amb)) );or
                            (crea-objeto (car exp) amb))
        (t                  (list (list 'error 'error-en-new-clase-indefinida) amb))
  );cond
```

```
); ev-new
;
;-------------------------------------------------------------------------
; funcion: CREA-OBJETO                                                   |
;                                                                        |
; CLASE-DEF:      (N-CLASS1 (CLASE N-CLASS2 (VAR-INST) ...))             |
; VALOR-C:        (CLASE N-CLASS2 (VAR-INST) ...))                       |
; (cdr valor-c):  (N-CLASS2 (VAR-INST) ...)                              |
; AMB-CLASE:      (VAR (NUMERO 0) VAR (NUMERO 0) ...)                    |
; (caaddr valor-c: (VAR-INST)                                            |
;-------------------------------------------------------------------------
;
(defun crea-objeto (n-class amb)
  (cond
        ((esta-def-nom-class n-class (cadr amb))
                (let* ((clase-def (asocia-tipo-valor n-class (cadr amb)))
                       (valor-c (cadr clase-def))
                       (amb-clase (crea-parejas (cadr valor-c) '())) )
                  (list (list n-class (cons-valor-obj valor-c amb amb-clase)) amb)
                ); let*
        )
        ((esta-def-nom-class n-class (caddr amb))
                (let* ((clase-def (asocia-tipo-valor n-class (caddr amb)))
                       (valor-c (cadr clase-def))
                       (amb-clase (crea-parejas (cadr valor-c) '())) )
                  (list (list n-class (cons-valor-obj valor-c amb amb-clase)) amb)
                ); let*
        )
  );cond
); crea-objeto
;
;-------------------------------
; funcion: CREA-PAREJAS      |
;-------------------------------
;
(defun crea-parejas (l-var-inst amb-c)
  (cond
        ((null l-var-inst)          amb-c)
        (t      (crea-parejas (cdr l-var-inst)
                              (actualiza (car l-var-inst) 'numero 0 amb-c)))
  ); cond
); crea-parejas
;
;-------------------------------
; funcion: ACTUALIZA         |
;-------------------------------
;
(defun actualiza (nom tipo valor amb)
  (cond
        ((null amb)     (list nom (list tipo valor)))
        ((eq (car amb) nom)     amb)
        (t              (append (list (car amb) (cadr amb))
                                (actualiza nom tipo valor (cddr amb))))
  ); cond
); actualiza
;
;-------------------------------------------------------
; funcion: CONS-VALOR-OBJ                    |
; exp-clase: (N-CLASS2 (VAR-INST) ... )      |
;-------------------------------------------------------
;
```

```
(defun cons-valor-obj (exp-clase amb amb-clase)
  (cond
        ((eq (car exp-clase) 'object                    amb-clase)
        ((esta-def-nom-class (car exp-clase) (cadr amb))
                (let* ((c-def (asocia-tipo-valor (car exp-clase) (cadr amb)))
                     (v-c (cadr c-def)))
                    (cons-valor-obj v-c amb
                                    (crea-parejas (cadr v-c) amb-clase))
                );let*
        )
        ((esta-def-nom-class (car exp-clase) (caddr amb))
                (let* ((c-def (asocia-tipo-valor (car exp-clase) (caddr amb)))
                     (v-c (cadr c-def)))
                    (cons-valor-obj v-c amb
                                    (crea-parejas (cadr v-c) amb-clase))
                );let*
        );
  );cond
); cons-valor-obj

;-------------------------------------
; funcion: EV-SETSHALLOW    !
;-------------------------------------

(defun ev-setshallow (exp amb)
  (cond
        ((null exp)                 (list (list 'error 'error-faltan-los-parametros-en-setshallow ) amb))
        ((and (not (esta-def-nom (car exp) (car amb)))
            (not (esta-def-nom (car exp) (cadr amb)))
            (not (esta-def-nom (car exp) (caddr amb)))
            (esta-def-nom-obj (cadr exp) amb))
                                    (crea-nvo-ambg exp amb) );and
;BUSCA EN AMBIENTE LOCAL

        ((and (esta-def-nom (car exp) (car amb))
            (esta-def-nom-obj (cadr exp) amb))
                        (list
                            (list 'alias (cadr exp))
                            (list
                                (actualiza-amb (car exp) 'alias (cadr exp) (car amb))
                                (cadr amb) (caddr amb));list
                        );list
        );and
;BUSCA EN AMBIENTE GLOBAL

        ((and (esta-def-nom (car exp) (cadr amb))
            (esta-def-nom-obj (cadr exp) amb))
                                    (crea-nvo-ambg exp amb));and
;BUSCA EN AMBIENTE PERSISTENTE

        ((and (esta-def-nom (car exp) (caddr amb))
            (esta-def-nom-obj (cadr exp) amb))
                    (list (list 'error 'ERROR-NO-SE-PUEDE-MODIFICAR-AMB-PERS-CON-
SETSHALLOW ) amb))
        (t                  (list (list 'error 'error-en-setshallow-no-esta-def-el-objeto) amb))
  );cond
); ev-setshallow

;-------------------------------------
; funcion: ESTA-DEF-NOM-OBJ         !
;-------------------------------------
```

```
;
(defun esta-def-nom-obj (nom-obj amb)
  (cond
        ((null amb)                                  nil)
        ((esta-def-nom nom-obj (car amb))            t)
        ((esta-def-nom nom-obj (cadr amb))           t)
        ((esta-def-nom nom-obj (caddr amb))          t)
        (t                                           nil)
  );cond
); esta-def-nom-obj
;
;-----------------------------------
; funcion: CREA-NVO-AMBG    |
;-----------------------------------
(defun crea-nvo-ambg (exp amb)
        (list
          (list 'alias (cadr exp))
          (list
                (car amb)
                (actualiza-amb (car exp) 'alias (cadr exp) (cadr amb))
                (caddr amb)
          );list
        );list
); crea-nvo-ambg
;
;-----------------------------------------
; funcion: EV-IF                         |
; exp = (expresion expresion expresion)|
;-----------------------------------------
;
(defun ev-if (exp amb)
   (cond
        ((null exp)    (list (list 'error 'error-sintaxis-if ) amb))
        (t             (let ((acond (ev-exp (car exp) amb)))
                            (cond
                                ((not (numberp (cadar acond)))
                                (list (list 'error 'error-en-clausula-if ) (cadr acond)))
                                ((not (eq 0 (cadar acond)))
                                        (ev-exp (cadr exp) (cadr acond)))
                                (t      (ev-exp (caddr exp) (cadr acond)))
                            );cond
                        );let
        );t
  );cond
); ev-if
;
;-----------------------------------
; funcion: EV-WHILE              |
; FINAL= Nueva-pareja            |
; (cadar (setq FINAL ...)) = VALOR |
;-----------------------------------
;
(defun ev-while (exp amb)
   (cond
        ((null exp)    (list (list 'error 'error-sintaxis-en-clausula-while) amb))
        (t             (let ((FINAL (ev-exp (car exp) amb)))
                            (cond
                                ((not (numberp (cadar FINAL)))
                                        (list (list 'error 'error-sintaxis-en-clausula-while) amb))
                                ((not (eq 0 (cadar FINAL)))
                                        (ev-while exp (cadr (ev-exp (cadr exp) (cadr FINAL)))) )
```

```
                                        (t      FINAL)
                                    ); cond
                                ); let
        );t
    );cond
); ev-while

;-------------------------------------
; funcion: EV-BEGIN          |
;-------------------------------------

(defun ev-begin (lista-exp amb)
    (cond
        (( null (cdr lista-exp))    (ev-exp (car lista-exp) amb))
        (t                              (ev-begin (cdr lista-exp)
                                                (cadr (ev-exp (car lista-exp) amb))))
    );cond
); ev-begin

;-------------------------------------
; funcion: EV-QUOTE          |
;-------------------------------------

(defun ev-quote (exp amb)
    (cond
        ((not(listp exp)) (list (list 'error 'error-en-sintaxis-de-quote) amb))
        ((> (length (cdr exp)) 1) (list (list 'error 'error-en-quote-exceso-de-parametros) amb))
      (t  (list (list 'alias (cadr exp)) amb))
    )
); ev-quote

;-------------------------------------
; funcion: EV-OPTR           |
;-------------------------------------

(defun ev-optr (exp amb)
    (cond
        ((null exp)                     (list (list 'cuidado 'expresion-nula ) amb) )
        ((eq (car exp) '+)              (apply-mas (cdr exp) amb)       )
        ((eq (car exp) '-)              (apply-menos (cdr exp) amb)     )
        ((eq (car exp) '*)              (apply-mult (cdr exp) amb)      )
        ((eq (car exp) '/)          (apply-div (cdr exp) amb)       )
        ((eq (car exp) '=)              (apply-igual (cdr exp) amb)     )
        ((eq (car exp) '<)              (apply-menor (cdr exp) amb)     )
        ((eq (car exp) '>)          (apply-mayor (cdr exp) amb)     )
        ((eq (car exp) 'print)          (apply-print (cdr exp) amb)     )
        ((eq (car exp) 'lista)          (apply-lista (cdr exp) amb)     )
        ((eq (car exp) 'car)            (apply-car (cdr exp) amb)       )
        ((eq (car exp) 'cdr)            (apply-cdr (cdr exp) amb)       )
        ((es-funcion (car exp) amb )    (apply-function exp amb))
        ((es-un-objeto (car exp) amb)   (apply-method exp amb)                  )
        ((es-lista-mensajes (car exp) amb) (apply-envio-mensaje exp amb))
        (t                              (list (list 'error 'expresion-invalida) amb) )
    )
); ev-optr

;-------------------------------------
; funcion: ES-FUNCION        |
;-------------------------------------

(defun es-funcion (nom amb)
```

```
  (cond
    ((listp nom)            nil)
    ((not (symbolp nom))        nil)
    ((esta-def-funcion nom (cadr amb))  t)
    (t                          nil)
  ); cond
); es-funcion

;-----------------------------------------------
; funcion: ESTA-DEF-FUNCION          |
;-----------------------------------------------
(defun esta-def-funcion (nom amb)
   (cond
        ((null amb)                                     nil)
        ((and (eq (car amb) nom)
              (eq (caadr amb) 'funcion))            t )
        (t                  (esta-def-funcion nom (cddr amb)))
   );cond
); esta-def-funcion

;-----------------------------------------------
; funcion: ES-LISTA-MENSAJES         |
;-----------------------------------------------

(defun es-lista-mensajes (l-mensajes amb)
   (cond
        ((or (not (listp l-mensajes))
             (not (symbolp (car l-mensajes))))              nil)
        (t                                              t )
   )
)

;-----------------------------------
; funcion: APPLY-MAS            |
;-----------------------------------

(defun apply-mas (exp amb)
   (cond
        ((and (not (null exp))
              (eq 2 (length exp)) )
;accion
                (let*
                    ((op1 (ev-exp (car exp) amb))
                     (op2 (ev-exp (cadr exp) (cadr op1))) )
                    (cond
                        ((and (eq 'numero (caar op1))
                              (eq 'numero (caar op2)) )
;accion
                            (ev-exp (+ (cadar op1) (cadar op2)) (cadr op2)))
                            (t      (list (list 'error 'error-en-operandos-de-+ ) (cadr op2)) )
                        );cond
                );let*
        );and
        ((> (length exp) 2)
                (list (list 'error 'error-exceso-de-argumentos-en-+) amb) )
        (t      (list (list 'error 'error-no-hay-operandos-en-+ ) amb))
  );cond
); apply-mas

;-----------------------------------
; funcion: APPLY-MENOS       |
```

```
;-----------------------------------------
;
(defun apply-menos (exp amb)
  (cond
        ((and (not (null exp))
              (eq 2 (length exp)) )
;accion
                    (let*
                        ((op1 (ev-exp (car exp) amb))
                         (op2 (ev-exp (cadr exp) (cadr op1))) )
                        (cond
                                ((and (eq 'numero (caar op1))
                                      (eq 'numero (caar op2)) )
;accion
                                    (ev-exp (- (cadar op1) (cadar op2)) (cadr op2)))
                                    (t      (list (list 'error 'error-en-operandos-de-resta ) (cadr op2)) )
                        );cond
                    );let*
        );and
        ((> (length exp) 2)
                (list (list 'error 'error-exceso-de-argumentos-en-resta) amb) )
        (t      (list (list 'error 'error-no-hay-operandos-en-resta ) amb))
  );cond
); apply-menos
;
;-----------------------------------------
; funcion: APPLY-MULT          |
;-----------------------------------------
;
(defun apply-mult (exp amb)
  (cond
        ((and (not (null exp))
              (eq 2 (length exp)) )
;accion
                    (let*
                        ((op1 (ev-exp (car exp) amb))
                         (op2 (ev-exp (cadr exp) (cadr op1))) )
                        (cond
                                ((and (eq 'numero (caar op1))
                                      (eq 'numero (caar op2)) )
;accion
                                    (ev-exp (* (cadar op1) (cadar op2)) (cadr op2)))
                                    (t      (list (list 'error 'error-en-operandos-de-* ) (cadr op2)) )
                        );cond
                    );let*
        );and
        ((> (length exp) 2)
                (list (list 'error 'error-exceso-de-argumentos-en-*) amb) )
        (t      (list (list 'error 'error-no-hay-operandos-en-* ) amb))
  );cond
); apply-mult
;
;-----------------------------------------
; funcion: APPLY-DIV           |
;-----------------------------------------
;
(defun apply-div (exp amb)
  (cond
        ((and (not (null exp))
              (eq 2 (length exp)) )
;accion
```

```
                (let*
                    ((op1 (ev-exp (car exp) amb))
                     (op2 (ev-exp (cadr exp) (cadr op1))) )
                    (cond
                        ((and (eq 'numero (caar op1))
                              (eq 'numero (caar op2))
                              (not (eq 0 (cadar op2))) )
;accion
                            (ev-exp (/ (- (cadar op1) (mod (cadar op1) (cadar op2))) (cadar op2))
(cadr op2)))
                            ((eq 0 (cadar op2)) (list (list 'error 'error-division-por-cero) (cadr op2)))
                            (t      (list (list 'error 'error-en-operandos-de-/ ) (cadr op2)) )
                    );cond
                );let*
        );and
        ((> (length exp) 2)
                (list (list 'error 'error-exceso-de-argumentos-en-/) amb) )
        (t      (list (list 'error 'error-no-hay-operandos-en-/ ) amb))
  );cond
); apply-div

;-------------------------------------
; funcion: APPLY-IGUAL          |
;-------------------------------------

(defun apply-igual (exp amb)
  (cond
        ((and (not (null exp))
              (eq 2 (length exp)) )
                (let*
                    ( (op1 (ev-exp (car exp) amb))
                     (op2 (ev-exp (cadr exp) (cadr op1))))
                    (cond
                        ((eq (caar op1) 'error) op1)
                        ((eq (caar op2) 'error) op2)
                        ((equal (cadar op1)
                            (cadar op2))
                                (list (list 'c 1) (cadr op2)))
                        (t   (list (list 'c 0) (cadr op2)))
                    );cond
                ); let*
        );and
        ((> (length exp) 2)
                (list (list 'error 'error-exceso-de-argumentos-en-igual) amb) )
        (t      (list (list 'error 'error-en-argumentos-de-igual ) amb))
  );cond
); apply-igual

;-------------------------------------
; funcion: APPLY-MENOR          |
;-------------------------------------

(defun apply-menor (exp amb)
  (cond
        ((and (not (null exp))
              (eq 2 (length exp)) )
                (let
                    ( (op1 (ev-exp (car exp) amb)))
                    (cond
                        ((< (cadar op1)
                            (cadar (ev-exp (cadr exp) (cadr op1))))
```

```
                                    (list (list 'c 1) (cadr op1)))
                                (t   (list (list 'c 0) (cadr op1)))
                            );cond
                        ); let
            );and
            ((> (length exp) 2)
                        (list (list 'error 'error-exceso-de-argumentos-en-<) amb) )
            (t          (list (list 'error 'error-en-argumentos-de-< ) amb))
    );cond
); apply-menor

;-------------------------------------
; funcion: APPLY-MAYOR          |
;-------------------------------------

(defun apply-mayor (exp amb)
    (cond
            ((and (not (null exp))
                  (eq 2 (length exp)) )
                        (let
                            ( (op1 (ev-exp (car exp) amb)))
                            (cond
                                    ((> (cadar op1)
                                (cadar (ev-exp (cadr exp) (cadr op1))))
                                            (list (list 'c 1) (cadr op1)))
                                (t   (list (list 'c 0) (cadr op1)))
                            );cond
                        ); let
            );and
            ((> (length exp) 2)
                        (list (list 'error 'error-exceso-de-argumentos-en->) amb) )
            (t          (list (list 'error 'error-en-argumentos-de-> ) amb))
    );cond
); apply-mayor

;-------------------------------------
; funcion: APPLY-PRINT          |
;-------------------------------------

(defun apply-print (e amb)
    (cond
            ((not (null e))
                        (let* ((op (ev-exp (car e) amb))
                               (nuevo-amb (cadr op)) )
                            (cond
                                    ((null (cdr e))
                                            (print (cadar op))
                                            (list (car op) nuevo-amb))
                                (t  (list (list 'error 'error-print-tiene-exceso-de-parametros ) nuevo-amb) )
                            );cond
                        ); let*
            ); not
            (t          (list (list 'error 'error-print-no-tiene-argumentos ) amb))
    ); cond
); apply-print

;-------------------------------------------
; funcion: APPLY-LISTA              |
; OBJETIVO: Constructor de listas   |
;-------------------------------------------
```

```
(defun apply-lista (exp amb)
  (cond
        ((null exp)   (list (list 'lista '() ) amb))
        (t            (const-list exp amb '()))
  )
); apply-lista

;-----------------------------------
; funcion: CONST-LIST              !
;-----------------------------------

(defun const-list (exp amb a-lista)
  (cond
        ((null exp)     (list (list 'lista (reverse a-lista)) amb))
        (t                (let* ((nva-exp (ev-exp (car exp) amb))
                                 (nva-lista (cons (car nva-exp) a-lista)))
                            (cond
                                ((eq (caar nva-exp) 'error) (list (list 'error 'parametro-indefinido-
en-lista) amb))
                                ((eq (caar nva-exp) 'funcion) (list (list 'error 'parametro-def-
funcion-no-permitido-en-lista) amb))
                                (t   (const-list (cdr exp) amb nva-lista)))))
  )
); const-list

;-------------------------------------------------------
; funcion: APPLY-CAR                                   !
; OBJETIVO: Obtener el CAR de una lista                !
;-------------------------------------------------------

(defun apply-car (exp amb)
  (cond
        ((null exp)     (list (list 'PRECACUCION '()) amb))
        ((> (length exp) 1)       (list (list 'ERROR 'EXCESO-DE-PARAMETROS-EN-CAR)
amb))
        (t        (let* ((nva-exp (ev-exp (car exp) amb))
                         (l (car nva-exp)))
                    (obtiene-car l amb)))
  )
); apply-car

;-----------------------------------
; funcion: OBTIENE-CAR             !
;-----------------------------------

(defun obtiene-car (l amb)
  (cond
        ((eq (car l) 'lista)
                (list (caadr l) amb))
        (t        (list (list 'ERROR 'ERROR-NO-ES-UNA-LISTA-EL-PARAMETRO-DE-CAR)
amb))
  )
); obtiene-car

;-------------------------------------------------------
; funcion: APPLY-CDR                                   !
; OBJETIVO: Obtener el CDR de una lista                !
;-------------------------------------------------------

(defun apply-cdr (exp amb)
  (cond
```

```
        ((null exp)     (list (list 'PRECAUCION '()) amb))
        ((> (length exp) 1)         (list (list 'ERROR 'EXCESO-DE-PARAMETROS-EN-CAR)
amb))
        (t          (let* ((nva-exp (ev-exp (car exp) amb))
                           (l (car nva-exp)))
                      (obtiene-cdr l amb)))
  )
); apply-cdr

;-----------------------------------
; funcion: OBTIENE-CDR          l
;-----------------------------------

(defun obtiene-cdr (l amb)
  (cond
        ((eq (car l) 'lista)
                (list (list 'lista (cdadr l)) amb))
        (t      (list (list 'ERROR 'ERROR-NO-ES-LISTA-EL-PARAMETRO-DE-CDR) amb))
  )
); obtiene-cdr

;-----------------------------------
; funcion: APPLY-FUNCTION    l
;-----------------------------------

(defun apply-function (exp amb)
   (cond
        ((not (esta-def-nom (car exp) (cadr amb)))
                (list (list 'error 'error-funcion-indefinida ) amb) )
        (t          (let* ((np (list (asocia-tipo-valor (car exp) (cadr amb)) amb))
                           (tipo (caar np))
                           (valor (cadar np)));np
                      (cond
                            ((not (eq tipo 'funcion))
                               (list (list 'error 'error-no-es-funcion) amb))
                            (t (evalua-function exp valor amb))
                      );cond
                ); let*
        ); t
   );cond
); apply-function

;-----------------------------------------

(defun evalua-function (exp valor amb)
  (cond
        ((and (es-lista-var (car valor))
            (eq (length (cdr exp)) (length (car valor)) );eq
            (not (null (car valor))) ); and
;accion
                (let* ((nueva-pareja (ev-argtos (cdr exp) (car valor) amb))
                             (nuevo-amb (cadr nueva-pareja)) );
                      (ev-exp (cadr valor) nuevo-amb)
                ); let*
        );and
        ((and (es-lista-var (car valor)) (null (car valor)) (null (cdr exp)) )
            (ev-exp (cadr valor) amb) );and
        (t          (list (list 'error 'error-en-lista-de-parametros ) amb))
  ); cond
); evalua-funcion
```

```
;-----------------------------------------

(defun ev-argtos (l-valores l-variables amb)
  (let* ((nueva-p (ev-exp (car l-valores) amb))
         (tip-val (car nueva-p))
         (nuevo-amb (cadr nueva-p)) );
         (cond
            ((eq (car tip-val) 'error) nueva-p)
            (t (let ((nuevo-al (actualiza-amb (car l-variables) (car tip-val)
                                              (cadr tip-val) (car nuevo-amb)) ) );
                     (cond
                        ((null (cdr l-valores))
                          (list tip-val (list nuevo-al (cadr nuevo-amb) (caddr nuevo-amb))) );null
                        (t (ev-argtos (cdr l-valores) (cdr l-variables)
                                      (list nuevo-al (cadr nuevo-amb) (caddr nuevo-amb))) );t
                     ); cond
                   ); let
            );t
           );cond
  );let*
); ev-argtos

;
;-----------------------------------------
; funcion: EV-FUNDEF-GLOBAL          |
;-----------------------------------------

(defun ev-fundef-global (exp amb)
   (cond
         ((null exp)                  (list (list 'error 'error-no-existe-def-de-funcion ) amb))
         ((not (eq 3 (length exp)))  (list (list 'error 'error-parametros-insuficientes-en-define )
amb))
         ((not (symbolp (car exp)))  (list (list 'error 'error-nombre-funcion-no-valido ) amb))
         ((not (es-lista-var (cadr exp))) (list (list 'error 'error-en-funcion-no-es-lista-argumentos)
amb))
         ((esta-def-nom (car exp) (cadr amb))    (list (list 'error 'error-nombre-funcion-ya-existe )
amb))
         (t
;accion
                  (list (list 'funcion (cdr exp))
                      (list (car amb)
                            (actualiza-amb (car exp) 'funcion (cdr exp) (cadr amb))
                        (caddr amb)
                      );list
                  );list
         )
  );cond
); ev-fundef

;
;-----------------------------------------
; funcion: ES-UN-OBJETO       |
;-----------------------------------------

(defun es-un-objeto (nom-ob amb)
   (cond
         ((listp nom-ob)              nil)
         ((not (symbolp nom-ob)) nil)
         ((esta-def-objeto nom-ob (car amb))     t)
         ((esta-def-objeto nom-ob (cadr amb))    t)
         ((esta-def-objeto nom-ob (caddr amb))  t)
  );con
```

```
); es-un-objeto

;-----------------------------------------------------
; funcion: ESTA-DEF-OBJETO                           !
;-----------------------------------------------------

(defun esta-def-objeto (nom amb)
  (cond
        ((null amb)                                    nil)
        ((and (eq (car amb) nom)
              (and (not (eq (caadr amb) 'numero))
                   (not (eq (caadr amb) 'cadena))
                   (not (eq (caadr amb) 'lista))
                (not (eq (caadr amb) 'funcion))
                (not (eq (caadr amb) 'clase))))    t)
        (t                  (esta-def-objeto nom (cddr amb)) )
  );cond
); esta-def-objeto

;-----------------------------------------------------
; funcion: APPLY-METHOD                              !
;                                                    !
; T-V = (nomclass (var (numero 0) ...))              !
;-----------------------------------------------------
(defun apply-method (exp amb)
(let ((tam-al (length (car amb)))
     (nom-obj (busca-objeto (car exp) amb)))
    (cond
        ((not nom-obj)              (list (list 'ERROR 'ERROR-EN-APPLY-METHOD-NO-ES-NOM-
OBJETO) amb))
        ((esta-def-objeto nom-obj (car amb))
            (let*  ((t-v (asocia-tipo-valor nom-obj (car amb)))
                    (n-amb-loc (append (car amb) (cadr t-v)))
                    (nueva-p (busca-aplica t-v exp amb n-amb-loc))
                    (na (cadr nueva-p)))
                    (cond
                        ((eq (caar nueva-p) 'error)  nueva-p)
                        (t
                        (let* ((tot-par-met (- (length (car na)) (length (cadr t-v)))))
                            (list (car nueva-p)
                                    (list (actualiza-amb nom-obj (car t-v)
                                                                (butlast (car na) tot-par-met) (car na))

                                                (cadr na)
                                                (caddr na))))));cond
            );let*
        )
        ((esta-def-objeto nom-obj (cadr amb))
            (let*  ((t-v (asocia-tipo-valor nom-obj (cadr amb)))
                    (n-amb-loc (append (car amb) (cadr t-v)) )
                    (nueva-p (busca-aplica t-v exp amb n-amb-loc))
                    (na (cadr nueva-p)))
                    (cond
                        ((eq (caar nueva-p) 'error)  nueva-p)
                        (t
                        (let*       ((tot-par-met (- (length (car na)) (length (cadr t-v)))))
                            (list (car nueva-p)
                                    (list (car na)
                                            (actualiza-amb nom-obj (car t-v)
                                                                (butlast (car na) tot-par-met) (cadr na))
                                            (caddr na))))));cond
```

```
            );let*
          )
          ((esta-def-objeto nom-obj (caddr amb))
             (let*  ((t-v (asocia-tipo-valor nom-obj (caddr amb)))
                     (n-amb-loc (append (car amb) (cadr t-v)) )
                     (nueva-p (busca-aplica t-v exp amb n-amb-loc))
                     (na (cadr nueva-p)))
                    (cond
                          ((eq (caar nueva-p) 'error)  nueva-p)
                          (t
                           (let* ((tot-par-met (- (length (car na)) (length (cadr t-v)))))
                              (list (car nueva-p)
                                    (list (car na)
                                          (cadr na)
                                          (actualiza-amb nom-obj (car t-v)
                                                         (butlast (car na) tot-par-met) (caddr
na)) )))));cond
                 );let*
          )
    );cond
)); apply-method
;
;-------------------------------------------------------
; funcion:BUSCA-OBJETO                          |
;-------------------------------------------------------
;
(defun busca-objeto (objeto amb)
  (cond
        ((null amb)       nil)
        (t        (let ((nvo-obj (busca-alias objeto amb)))
                    (cond
                          ((and (es-un-objeto nvo-obj amb)
                                (es-alias nvo-obj amb))
                                      (busca-objeto nvo-obj amb))
                          (t        nvo-obj))
                    ); let
        )
  )
); busca-objeto
;
;-------------------------------------------------------
; funcion: ES-ALIAS                             t
;-------------------------------------------------------
;
(defun es-alias (objeto amb)
  (cond
        ((listp objeto)                                 nil)
        ((not (symbolp objeto))                 nil)
        ((busca-alias-amb objeto (car amb))       t )
        ((busca-alias-amb objeto (cadr amb))      t )
        ((busca-alias-amb objeto (caddr amb))     t )
        (t                                              nil)
  );cond
); es-alias
;
;-------------------------------------------------------
; funcion: BUSCA-ALIAS-AMB                      |
;-------------------------------------------------------
;
(defun busca-alias-amb (objeto amb)
  (cond
```

```
        ((null amb)                                     nil)
        ((eq (car amb) objeto)
                (cond
                  ((eq (caadr amb) 'alias)          t )
                  (t                                nil)
                ))
        (t      (busca-alias-amb objeto (cddr amb)))
  );cond
); busca-alias-amb

;-----------------------------------------------------
; funcion: BUSCA-ALIAS                               |
;-----------------------------------------------------

(defun busca-alias (nom amb)
  (cond
        ((null amb)                 nom)
        (t          (let ((obj (busca-alias-obj nom (car amb))))
                        (cond
                          (obj  obj)
                          (t                (busca-alias nom (cdr amb)))
                        );cond
                    ); let
        )
  ); cond
); busca-alias

;-----------------------------------------------------
; funcion: BUSCA-ALIAS-OBJ                           |
;-----------------------------------------------------

(defun busca-alias-obj (nom amb)
  (cond
        ((null amb)                             nil)
        ((and (eq (car amb) nom)
            (eq (caadr amb) 'alias))
                                (cadadr amb))
        (t                      (busca-alias-obj nom (cddr amb)))
  );cond
); busca-alias-obj

;-----------------------------------------------------------------------
; funcion: BUSCA-APLICA                                                |
;                                                                      |
; DEF-CLASE = (n-class (CLASE (n-class2 (var inst) metodos)))          |
;-----------------------------------------------------------------------

(defun busca-aplica (def-c-m exp amb a-loc)
  (cond
        ((eq (car def-c-m) 'object)  (list (list 'error 'error-no-se-encontro-def-metodo) amb))
        ((esta-def-nom-class (car def-c-m) (car amb))
                (let* ((def-clase (asocia-tipo-valor (car def-c-m) (car amb)))
                       (metodo (busca-metodo (cadr exp) (cddadr def-clase))))
                  (cond
                        (metodo (let* ((nuevo-amb (list (append (car amb) a-loc)
                                                        (cadr amb)
                                                        (caddr amb))))
                                        (evalua-function (cdr exp) (cadr metodo) nuevo-amb)))
                                (t      (busca-aplica (cadr def-clase) exp amb a-loc)))))
        ((esta-def-nom-class (car def-c-m) (cadr amb))
                (let* ((def-clase (asocia-tipo-valor (car def-c-m) (cadr amb)))
```

```
                    (metodo (busca-metodo (cadr exp) (cddadr def-clase))))
                   (cond
                        (metodo (let* ((nuevo-amb (list (append (car amb) a-loc)
                                                        (cadr amb)
                                                        (caddr amb))))
                                      (evalua-function (cdr exp) (cadr metodo) nuevo-amb)))
                                (t      (busca-aplica (cadr def-clase) exp amb a-loc)))))
        ((esta-def-nom-class (car def-c-m) (caddr amb))
                (let* ((def-clase (asocia-tipo-valor (car def-c-m) (caddr amb)))
                    (metodo (busca-metodo (cadr exp) (cddadr def-clase))))
                   (cond
                        (metodo (let* ((nuevo-amb (list (append (car amb) a-loc)
                                                        (cadr amb)
                                                        (caddr amb))))
                                      (evalua-function (cdr exp) (cadr metodo) nuevo-amb)))
                                (t      (busca-aplica (cadr def-clase) exp amb a-loc)))))
  );cond
); busca-aplica
;
;-----------------------------------------------
; funcion: BUSCA-METODO                        |
;                                              |
;(cadr l-met) = (funcion ((arglist) exp))     |
;-----------------------------------------------
;
(defun busca-metodo (met l-met)
  (cond
        ((null l-met)                   nil)
        ((eq (car l-met) met)   (cadr l-met))
        (t              (busca-metodo met (cddr l-met)))
  );cond
); busca
;
;-----------------------------------------------
; funcion: APPLY-ENVIO-MENSAJE    |
;-----------------------------------------------
;
(defun apply-envio-mensaje (exp amb)
  (let ((mensaje (car exp)))
      (cond
          ((not (es-un-objeto (car mensaje) amb))
                  (list (list 'error 'error-no-es-objeto-en-lista-mensajes) amb))
          (t      (let* ((resultado (apply-method mensaje amb))
                      (nvo-nom-metodo (cadar resultado))
                        (nvo-amb (cons '() (cdadr resultado))) )
                   (cond
                     ((not (es-un-objeto nvo-nom-metodo nvo-amb))
                                (list (list 'error 'error-metodo-no-existe-en-lista-mensajes) nvo-
amb))
                                (t      (let ((nvo-metodo (cons nvo-nom-metodo (cdr exp)) ))
                                        (apply-method nvo-metodo nvo-amb) )); let-2 y t
                     );cond-2
                  ); let*
          );t
      );cond-1
  ); let-1
); apply-envio-mensaje
;
;-----------------------------------------------
; funcion: PRINT-RESULT                        |
; objetivo: imprimir los elementos de una lista |
```

```
;-----------------------------------------------------------

(defun print-result (resu nvo-result)
  (cond
        ((null resu)        (reverse nvo-result))
        ((eq (caar resu) 'lista)
                (let ((nva-list (cadar resu))
                    (otro-result '()))
                  (print-result (cdr resu) (cons (print-result nva-list otro-result) nvo-result))))
        (t                  (print-result (cdr resu) (cons (cadar resu) nvo-result)))
  )
); print-result
```

APENDICE B

Código del ejemplo CIRCULO en LEPPOOC.

```
(class figura object
    (pi r)
    (define vpi () (set pi 3.1416))
    (define vr (valor) (set r valor))
)

(class circulo figura
    ()

    (define area ()
        (begin (set a (* pi (* r r)))
                (print "#area")
                (print a)) )
    (define circ ()
        (begin (set c (* 2 (* pi r)))
                (print "#circunferencia")
                (print c)) )
)

(set mi-fig (new figura))
(set mi-circulo (new circulo))
mi-circulo
(mi-circulo vpi)
(mi-circulo vr 4)
(mi-circulo area)
mi-circulo
(persist mi-circulo)
(persist mi-fig)
```

Programa OBJ-COMP

```
(class persona object
      (nombre dirección rfc profesión vehic grado)

      (define esc-nom () (print nombre))
      (define esc-direc () (print dirección))
      (define esc-rfc () (print rfc))
      (define esc-prof () (print profesión))
      (define esc-vehic () (print vehic))
      (define esc-grado ()
        (if (= grado 0) (print "#Sin grado")
            (if (= grado 1) (print "#Licenciatura")
                (if (= grado 2) (print "#Especialidad)
                    (if (= grado 3) (print "#Maestría")
                        (print "#Doctorado")
                    )
                )
            )
        )
      )
      (define set-nom (nom) (set nombre nom))
      (define set-direc (direc) (set dirección direc))
      (define set-rfc (nvo-rfc) (set rfc nvo-rfc))
      (define set-prof (prof) (set profesión prof))
      (define set-grado (gdo) (set grado gdo))
   )
   (class direc-comp object
      (ciudad calle numero cp)
      (define v-cd () (set ciudad "#Xalapa, Ver."))
      (define v-calle (ca) (set calle ca))
      (define v-numero (num) (set numero num))
      (define v-cp (codpost) (set cp codpost))
   )

   (set obj-persona (new persona))
   obj-persona
   (obj-persona set-nom "#Juana")
   (set obj-dircomp (new direc-comp))
   (obj-persona set-direc obj-dircomp)
   (obj-persona set-direc (quote obj-dircomp))
   (obj-persona set-nom "#Juana Comparte")
   (obj-persona esc-direc)
   ((obj-persona esc-direc) v-calle "#Lucio 34")
```