INSTITUTO TECNOLÓGICO Y DE ESTUDIOS
SUPERIORES DE MONTERREY

CAMPUS ESTADO DE MÉXICO

TECNOLÓGICO
DE MONTERREY®

Diseño e Implementación del Lenguaje Huntul: Una Herramienta para la Construcción de Software con Objetos Persistentes

TESIS QUE PARA OBTENER EL GRADO DE:
MAESTRO EN CIENCIAS COMPUTACIONALES

PRESENTA

ARIEL ORTIZ RAMÍREZ

Siendo integrado el jurado por:

Dr. Héctor Saldaña Aldana	Sinodal y asesor
M. en C. Néstor Batres Guzmán	Sinodal
M. en C. Francisco Camargo Santacruz	Sinodal

Estado de México. Junio, 1994.

P R E F A C I O

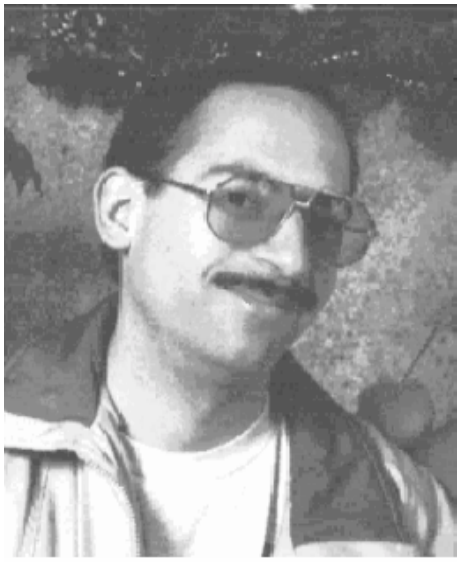
Hace casi nueve años aprendí por mi propia cuenta a programar, utilizando el lenguaje BASIC. Desde entonces, para mi la computación y la programación han sido sinónimos. De todas las actividades intelectuales que he desarrollado en mi vida, la programación de computadoras ha sido la que más me ha apasionado. Esta fue la motivación que me llevó a estudiar computación a nivel de licenciatura y posteriormente en la maestría.

En todo el tiempo que llevo programando he tenido contacto con un gran número de lenguajes que reflejan paradigmas de programación distintos. Conocí el paradigma de orientación a objetos en 1988, cuando cursaba el quinto semestre de la carrera. Junto con mis compañeros aprendí Smalltalk, y debo confesar que tardé mucho tiempo, posiblemente más de un año, en entender bien la filosofía que había detrás de este lenguaje.

Mi experiencia con lenguajes orientados a objetos ha sido diversa. Conozco bien C++ y Object Pascal. Tengo algo de experiencia con Smalltalk y CLOS. Aunque cada uno de estos lenguajes tiene características sobresalientes, sentía que para aprender a programar orientado a objetos éstos resultan inadecuados en algunos aspectos. Por tal motivo decidí diseñar e implementar mi propio lenguaje, al cual llamé Huntul, que combina la sintaxis de los lenguajes estructurados convencionales con el poder de los lenguajes orientados a objetos puros, además de contar con persistencia. La palabra Huntul significa en maya "objeto animado", y es justamente el calificativo que deben tener los componentes que conforman a cualquier programa escrito en este lenguaje.

En el capítulo 1 de este trabajo se explican las motivaciones por las que surge la programación orientada a objetos. Los capítulos 2 y 3 describen a detalle el lenguaje Huntul y su biblioteca de clases. El capítulo 4 aborda la forma en que Huntul fue implementado. Las conclusiones se encuentran en el capítulo 5, donde se hace una comparación de Huntul frente a otros lenguajes orientados a objetos y se indican sus principales limitaciones. El apéndice A es la guía del usuario para la implementación. Por último, el apéndice B presenta tres ejemplos de programas completos escritos en lenguaje Huntul.

El diseñar e implementar este lenguaje ha sido una labor monumental, pero tengo la gran satisfacción de ver este trabajo terminado y funcionando casi como originalmente lo concebí. Desde que comencé a programar soñé con el día en que yo mismo diseñara mi propio lenguaje. Ese día por fin llegó y estoy muy complacido por los resultados logrados.



Ing. Ariel Ortiz Ramírez

Cuautitlán Izcalli, Estado de México.

24 de junio, 1994.

A G R A D E C I M I E N T O S

La ayuda, apoyo e inspiración de mucha gente es fundamental para estudiar una maestría y elaborar un trabajo de tesis. Primeramente quiero dar gracias a Dios, pues Él es quien brinda la vida, la salud y el talento para realizar cualquier proyecto. Agradezco a mi esposa Lupita, quién desde recién casados me apoyó y soportó resignadamente todo mi trabajo de maestría. Así mismo, le doy gracias a mi hijo Ariel, de un año y medio de edad, quien me dejó trabajar por largos ratos en la computadora, interrumpiéndome esporádicamente sólo para ponerse a jugar con el teclado de la máquina.

Quiero agradecer también a las siguientes personas : al Dr. Héctor Saldaña por fungir como asesor de esta tesis; al Ing. Néstor Batres y al Ing. Francisco Camargo por ser mis sinodales; al Lic. Miguel Salais y al Ing. Juan López por brindarme su apoyo para que pudiera estudiar la maestría; a mi padre el Dr. Ariel Ortiz y mi madre la Sra. Norma de Ortiz por el apoyo y excelente ejemplo que me brindaron desde niño; a mi gran amigo y compañero Ing. Thomas Gersten quien fue mi primer profesor de programación orientada a objetos; a mis alumnos Roberto Pérez y Rafael Mayoral quienes desinteresadamente me brindaron su apoyo y amistad; a mis alumnas Mónica Frías, Lizbeth Alpizar y Vanessa Islas quienes me ayudaron a realizar correcciones sobre este documento escrito; a mis profesores de maestría, en particular al Dr. Martín López, al Dr. Isaac Rudomín y al Dr. Oscar Chavoya por su gran labor y dedicación dentro del ITESM; y a todos mis alumnos presentes, pasados y futuros, pues son ellos principalmente quienes me motivan a seguirme superando.

Por último, quiero agradecer al Tecnológico de Monterrey, Campus Estado de México, del cual soy también orgulloso egresado de profesional, por haberme brindado tantos años de inmensos retos y grandes satisfacciones.

C O N T E N I D O

PREFACIO **i**

AGRADECIMIENTOS **iii**

CONTENIDO **iv**

1 Lenguajes y Abstracciones **1**

1.1 Evolución de los lenguajes de programación *2*

1.1.1 Lenguajes de bajo nivel *2*

1.1.2 Lenguajes de alto nivel *4*

1.2 Mecanismos de abstracción *5*

1.2.1 Subrutinas *6*

1.2.2 Módulos *8*

1.2.3 Tipos de datos abstractos *11*

1.2.4 Objetos *13*

1.3 Programación orientada a objetos *14*

1.3.1 Encapsulamiento *14*

1.3.2 Herencia *15*

1.3.3 Polimorfismo *15*

1.3.4 Persistencia *16*

2 El Lenguaje de Programación Huntul **18**

2.1 Características del lenguaje *19*

2.1.1 Huntul es un lenguaje orientado a objetos puro *19*

2.1.2 Huntul es un lenguaje imperativo *20*

2.1.3 Huntul es un lenguaje de programación estructurada *21*

2.1.4 Huntul es un lenguaje con persistencia *21*

2.1.5 Huntul es un lenguaje con manejo automático de memoria *22*

2.1.6 Huntul es un lenguaje con tipos latentes *23*

2.1.7 Huntul es un lenguaje en español *23*
2.2 Componentes de un sistema *24*
2.3 Convenciones léxicas *26*
2.3.1 Componentes léxicas *26*
2.3.2 Comentarios *26*
2.3.3 Identificadores *27*
2.3.4 Palabras reservadas *28*
2.3.5 Literales *28*
2.3.5.1 Constantes de la clase Nulo *28*
2.3.5.2 Constantes de la clase Entero *29*
2.3.5.3 Constantes de la clase Booleano *29*
2.3.5.4 Constantes de la clase Carácter *29*
2.3.5.5 Constantes de la clase Cadena *29*
2.3.5.6 Constantes de la clase Arreglo *29*
2.4 Conceptos básicos *30*
2.4.1 Objetos *30*
2.4.2 Variables *30*
2.4.2.1 Alcance de una variable *30*
2.4.2.2 Tiempo de vida de una variable *31*
2.4.2.3 Categorías de variables *31*
2.4.3 Clases *31*
2.4.4 Jerarquía de clases *32*
2.4.5 Clases primitivas *32*
2.4.6 Clase Metaclase *33*
2.4.7 Mensajes y métodos *34*
2.4.8 Expresiones *35*
2.5 Módulo de aplicación (MA) *36*
2.5.1 Declaración de variables comunes *37*
2.5.2 Declaración de variables persistentes *37*
2.5.3 Declaración de variables locales *39*
2.5.4 Sentencias ejecutables *39*
2.6 Módulo de definición de clases (MDC) *40*

2.6.1 definstancia y defclase *41*
2.6.2 Variables de instancia y de clase *41*
2.6.3 Métodos de instancia y de clase *42*
2.6.4 Receptor de un método *45*
2.6.5 Antecesor de un método *45*
2.6.6 El valor de regreso de un método *46*
2.7 Sentencias *47*
2.7.1 Sentencia de expresión *47*
2.7.2 Sentencia de asignación *48*
2.7.3 Sentencia condicional *48*
2.7.4 Sentencia de selección *49*
2.7.5 Sentencia de iteración *50*
2.7.6 Sentencia de regreso *51*
2.7.7 Sentencia de bajo nivel *51*
3 Biblioteca de clases primitivas de Huntul 52
3.1 Clase Genérico *53*
3.1.1 Métodos de clase *53*
3.1.2 Métodos de instancia *53*
3.2 Clase Nulo *56*
3.2.1 Métodos de clase *56*
3.2.2 Métodos de instancia *56*
3.3 Clase Entero *57*
3.3.1 Variables de clase *57*
3.3.2 Métodos de clase *57*
3.3.3 Métodos de instancia *58*
3.4 Clase Booleano *61*
3.4.1 Métodos de clase *62*
3.4.2 Métodos de instancia *62*
3.5 Clase Carácter *64*
3.5.1 Métodos de clase *64*
3.5.2 Métodos de instancia *64*
3.6 Clase Cadena *66*

3.6.1 Métodos de clase *67*

3.6.2 Métodos de instancia *67*

3.7 Clase Arreglo *70*

3.7.1 Métodos de clase *70*

3.7.2 Métodos de instancia *70*

3.8 Clase Código *73*

3.8.1 Métodos de clase *73*

3.8.2 Métodos de instancia *73*

4 Implementación del Sistema Huntul 76

4.1 Programación del sistema *76*

4.1.1 Módulos de rutinas de soporte general *77*

4.1.2 Módulos de rutinas de inicialización y control general *78*

4.1.3 Módulos de rutinas para el compilador *79*

4.1.4 Módulos de rutinas para el manejo de clases y métodos *79*

4.1.5 Módulos de rutinas de tiempo de corrida *80*

4.2 El compilador de Huntul *80*

4.2.1 Analizador léxico *80*

4.2.2 Analizador sintáctico *81*

4.2.3 Analizador semántico *83*

4.2.4 Generador de código *84*

4.3 Manejo de memoria *84*

4.3.1 Memoria dinámica convencional *85*

4.3.2 Memoria dinámica para objetos *86*

4.4 Estructuras de archivo *88*

4.4.1 Archivo del receptáculo para almacenar clases *88*

4.4.2 Archivo del receptáculo para almacenar métodos *89*

4.4.3 Archivo del receptáculo para almacenar datos persistentes *90*

4.4.4 Archivo de control de ejecución *91*

4.5 Control de ejecución *92*

4.5.1 Convenciones de uso de registros *92*

4.5.2 Rutinas de soporte *93*

5 Conclusiones **96**

5.1 Huntul frente a otros lenguajes *96*

5.2 Limitaciones de Huntul *99*

5.3 Trabajo futuro *100*

BIBLIOGRAFÍA **102**

APÉNDICE A : Guía del Usuario Huntul Ver. 1.00 **105**

APÉNDICE B : Programas ejemplo **113**

C A P Í T U L O

1

Lenguajes y Abstracciones

Cualquier tipo de problema requiere de un lenguaje para poder ser planteado y posteriormente resuelto (el término lenguaje se utiliza aquí para denotar cualquier medio que se emplee para expresar ideas). Para un problema dado, existirán lenguajes que permitan que dicho problema sea resuelto de manera rápida y sencilla; por otro lado, existirán otros lenguajes que hagan imposible su resolución.

Tomando un ejemplo de la aritmética, la operación de la división era bien conocida por los antiguos griegos y romanos. Sin embargo, muy poca gente de aquel entonces era capaz de resolver inclusive un problema sencillo de división. En la actualidad, un niño de ocho años puede hacer divisiones no triviales con relativa facilidad. Esta diferencia no se debe a que las personas de hoy sean más inteligentes que las de otras épocas. La diferencia estriba en el lenguaje usado para hacer divisiones. Mientras que los antiguos usaban el sistema de numeración romano, actualmente se utiliza un sistema de numeración posicional basado en los números arábigos. Este último permite que las divisiones se realicen de manera directa, mientras que el anterior las hacía prácticamente imposibles.

Los lenguajes se utilizan como un instrumento en el proceso de comunicación, ya sea entre dos o más personas, o incluso entre una persona y una máquina. El estudio de los lenguajes utilizados para comunicar a los seres humanos con las computadoras ha sido central dentro de las ciencias computacionales. El propósito de este capítulo es mostrar cómo los lenguajes computacionales han cambiado a través del tiempo, incorporando cada vez nuevos y mejores mecanismos de abstracción, proporcionando así herramientas que permitan manejar adecuadamente la complejidad involucrada en la construcción de software.

1.1 Evolución de los lenguajes de programación

Los primeros lenguajes de programación tienen su origen con las primeras computadoras digitales. En ese entonces los lenguajes eran un reflejo directo del hardware que controlaban. Poco a poco esto fue cambiando, de tal forma que la mayoría de los lenguajes contemporáneos son independientes de cualquier plataforma computacional.

1.1.1 Lenguajes de bajo nivel

A finales de los años cuarenta, John von Neumman del Instituto de Estudios Avanzados de Princeton, New Jersey, con la colaboración de H. H. Goldstine y A. W. Burks, propusieron dos conceptos importantes :

1. El sistema de numeración usado por las computadoras debería ser el binario en lugar del decimal, dado que es más sencillo para los componentes electrónicos el tener que representar únicamente dos estados.
2. La memoria de la computadora, además de almacenar los datos de un programa, debería almacenar al programa mismo[1].

El verdadero origen de estos conceptos es discutible, pero lo importante a notar aquí es que estas ideas han sido la base de la arquitectura de la gran mayoría de las computadoras construidas hasta la fecha.

La organización de la máquina de von Neumman es relativamente simple. De manera general está compuesta de una unidad de control, una unidad aritmética, una unidad de entrada/salida, y un bloque de memoria[2]. En la memoria se almacenan datos enteros, a los cuales se les puede aplicar operaciones aritméticas cuyos resultados son depositados en un registro llamado acumulador. Es posible modificar las localidades de memoria mediante una operación conocida como asignación. Las instrucciones del programa se toman de la memoria y se ejecutan de manera secuencial, a no ser que se encuentre una instrucción de

---

[1] Donald H. Sanders, *Computers Today.* (McGraw-Hill. New York, New York. 1983) p. 38.

[2] Ravi Sethi, *Programming Languages Concepts and Constructs.* (Addison-Wesley. Reading, Massachusets. 1989) pp. 5-7.

salto, en cuyo caso la siguiente instrucción a ejecutar debe ser tomada de la localidad de memoria indicada por dicha instrucción.

La descripción anterior nos da nociones de lo que por lo menos los primeros lenguajes computacionales debían ser capaces de hacer. En la figura 1.1 se muestra un conjunto elemental de instrucciones de un lenguaje hipotético para una máquina de von Neumman.

Instrucción	Significado
`A := M[i]`	Copia al acumulador el contenido de la localidad i de memoria.
`M[i] := A`	Copia a la localidad i de memoria el contenido del acumulador.
`A := A + M[i]`	Suma al acumulador el contenido de la localidad i de memoria.
`A := A - M[i]`	Resta al acumulador el contenido de la localidad i de memoria.
`A := - M[i]`	Deja en el acumulador la negación del contenido de la localidad i de memoria.
`goto M[i]`	Salta incondicionalmente a la dirección contenida en la localidad i de memoria.
`if A >= 0 goto M[i]`	Si el acumulador es mayor o igual a cero salta a la dirección contenida en la localidad i de memoria.

Figura 1.1 Instrucciones hipotéticas de una máquina de von Neumman

A los lenguajes que una computadora puede entender de manera directa se les conoce como *lenguajes de máquina*. De manera simplista, a cada instrucción de lenguaje de máquina se le asigna un código de operación, el cual es un simple número. Estos códigos de operación son almacenados en la memoria en su representación binaria. Por ejemplo, para incrementar en uno el valor del acumulador en el procesador Intel i486 se tiene el siguiente código de operación en binario :

```
01100110 01000000
```

Aunque para una máquina esta serie de unos y ceros es muy lógica, para los seres humanos resulta bastante incomprensible. Es por esto que surgió una variante de lenguaje de máquina conocido como *lenguaje ensamblador*, en donde los códigos de operación junto con otros elementos de un programa son reemplazados por nombres simbólicos. Dado el ejemplo anterior, su equivalente en ensamblador sería :

```
inc eax
```

Aunque el ensamblador aún resulta un tanto críptico, es definitivamente más claro que su representación binaria. Tanto al lenguaje ensamblador como al lenguaje máquina se les refiere como *lenguajes de bajo nivel*.

1.1.2 Lenguajes de alto nivel

Cuando las computadoras digitales se convirtieron en un producto comercial, se vio la necesidad imponente de poder programarlas utilizando lenguajes más adecuados y sencillos que el ensamblador. Además, el lenguaje ensamblador tiene otra gran limitación : cada modelo de computadora tiene un lenguaje de máquina distinto a los demás, por lo que los programas escritos para una máquina específica solamente funcionan ahí.

Era evidente que para superar los problemas que existían en el desarrollo de los sistemas computacionales, la programación de éstos se debía realizar a un nivel superior que el impuesto por las máquinas. Es así que surge el concepto de *lenguaje de alto nivel*. Este tipo de lenguajes ofrecen al menos un par de ventajas significativas sobre los lenguajes de bajo nivel :

- *Transportabilidad.* Un lenguaje de alto nivel no es dependiente de una computadora específica; esto implica que los programas pueden correr en distintas computadoras siendo el único requisito el que exista el traductor del lenguaje para la máquina en cuestión.
- *Fácil comprensión.* Los programas escritos en lenguajes de alto nivel son más fáciles de escribir y entender. Esto generalmente permite una reducción en el tiempo de no tan solo la codificación, sino también de las fases de depuración y mantenimiento.

Las distintas características específicas que se han incorporado en los lenguajes de alto nivel a través de los años son muy variadas. Booch resume la evolución de los lenguajes de alto nivel en tres claras tendencias[3]. En la figura 1.2 se puede observar esquemáticamente estas tendencias. Los primeros lenguajes de alto nivel fueron diseñados para resolver problemas de índole numérico, por lo tanto tenían una tendencia hacia la representación de expresiones matemáticas. Posteriormente el énfasis cambió hacia *cómo* se debían hacer las cosas, es decir, los algoritmos comenzaron a jugar el papel preponderante.

---

[3] Grady Booch, *Object-Oriented Analysis and Design With Applications* 2nd Edition. (Benjamin Cummings. Redwood City, California. 1994) p. 29.

Finalmente, comenzando desde los años setenta y hasta la fecha, ha habido una tendencia hacia lo que se conoce como *abstracción de datos*, que consiste en diseñar los sistemas de tal forma que gran parte de un programa puede ser especificado con independencia de la representación interna de los datos[4].

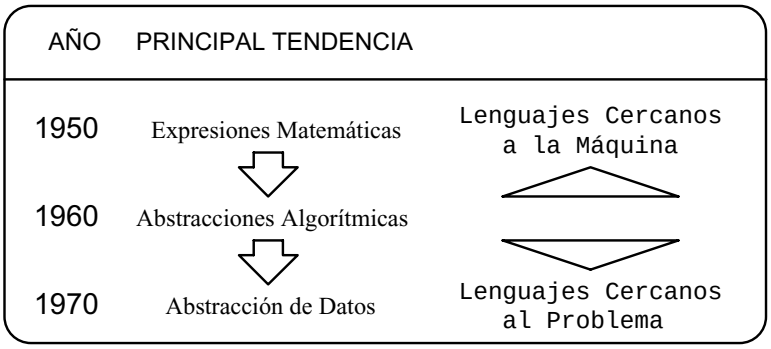


Figura 1.2 Evolución de las principales tendencias de los lenguajes de alto nivel.

En resumen, los lenguajes de alto nivel empezaron con una orientación hacia la máquina y se transformaron posteriormente hacia una orientación a la resolución de problemas.

1.2 Mecanismos de abstracción

Autores como Booch[5] y Budd[6] coinciden en que el manejo de la complejidad es el principal problema que afrontan los desarrolladores de software. Booch explica que la complejidad inherente en el desarrollo de software se debe a cuatro factores :

1. La complejidad propia del problema que se intenta resolver.
2. La dificultad que surge al administrar el proceso de desarrollo del software.

---

[4] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* (MIT Press. Cambridge, Massachusetts. 1985) p. 126.

[5] Booch, *op. cit.*, pp. 3-24.

[6] Timothy Budd, *An Introduction to Object-Oriented Programming.* (Addison-Wesley. Reading, Massachusetts. 1991) p. 10.

3. La tentación de "reinventar la rueda", esto es, reescribir componentes ya existentes.

4. Dado que un programa es un sistema discreto, existe un número finito posible de estados en los que se puede encontrar. El número de estados puede ser bastante grande en programas de tamaño respetable. Como no es prácticamente posible generar cada uno de esos estados, el correcto funcionamiento de un programa no se puede garantizar totalmente[7].

Para poder manejar la complejidad del software, los programadores se han valido de la abstracción. La idea principal detrás de la abstracción está en considerar aisladamente cada uno de los elementos que conforman un todo. La abstracción es indispensable por el hecho de que el ser humano no puede lidiar en su mente con más de siete u ocho trozos de información a la vez[8].

Los cuatro mecanismos de abstracción más importantes que han proporcionado los lenguajes de programación han sido : las subrutinas, los módulos, los tipos de datos abstractos, y los objetos[9]. Algunos han hecho un mejor papel que otros, pero todos han resultado útiles por el hecho de que sólo requieren que unas cuantas partes del sistema sean comprendidas en un momento dado.

1.2.1 Subrutinas

La primera forma de abstracción de programación ampliamente utilizada fue la subrutina, conocida también como procedimiento. Uno de los principales atractivos de la subrutina es la de permitir que las tareas que requieren ser ejecutadas muchas veces se codifiquen una sola vez y luego solamente sean invocadas cuando sean requeridas, reduciendo con ello el tamaño del código.

A manera de ejemplo, supóngase que se requiere hacer un programa que maneje una tabla indexada de cadenas de carácteres. Se requieren dos operaciones : 1) dada una cadena, insertarla al final de la tabla; y 2) dada una cadena, buscarla en la tabla y devolver el número de índice en el que se encuentra (el primer índice debe ser 1) o cero en caso de no encontrarla.

---

[7] Booch, *op. cit.*, p. 5.-8

[8] Steve McConnell, *Code Complete.* (Microsoft Press. Redmond, Washington. 1993) p. 108.

[9] Budd, *op. cit.*, p. 11.

Utilizando el lenguaje BASIC[10], se pueden utilizar subrutinas para implementar dicha tabla con la funcionalidad descrita[11]. Primero es necesario declarar una variable que contenga a un arreglo de cadenas de carácteres y una variable numérica que nos indique el índice de la siguiente localidad disponible del arreglo. Para que las subrutinas puedan tener acceso a dichas variables, éstas deberán ser globales (de cualquier forma BASIC solo cuenta con este clase de variables). A continuación está el código requerido :

```
  10 REM ---- Declaración de Variables -----------------------------
  20     LET Maximo = 20
  30     DIM Tabla$(Maximo)
  40     LET Nivel = 1
  50     GOTO 5000 : REM Ir a la rutina principal.
1000 REM ------------------------------------------------------------
1010 REM Subrutina de búsqueda en la tabla de un elemento dado.
1020 REM   Entradas : Info$       Cadena a buscar.
1030 REM   Salidas  : Resultado
1040 REM                    0 indica que no se halló el elemento.
1050 REM                    Otro número indica el índice de la tabla
1060 REM                    donde se encontró el elemento.
1070 REM   Variables modificadas : I
1080 REM ------------------------------------------------------------
1090     LET Resultado = 0
1100     FOR I = 1 TO NIVEL - 1
1110         IF Tabla$(I) = Info$ THEN LET Resultado = I : RETURN
1120     NEXT I
1130 RETURN
2000 REM ------------------------------------------------------------
2010 REM Subrutina de inserción en la tabla de un elemento dado.
2020 REM   Entradas : Info$       Cadena a insertar.
2030 REM   Variables modificadas : Tabla$(), Nivel
2040 REM ------------------------------------------------------------
2050     IF Nivel = Maximo THEN PRINT "Tabla desbordada" : END
2060     LET Tabla$(Nivel) = Info$
2070     LET Nivel = Nivel + 1
2080 RETURN
```

La subrutina de búsqueda realiza una búsqueda lineal sobre todos los elementos contenidos en el arreglo hasta ese momento. La subrutina de inserción solamente verifica que exista espacio para insertar el elemento indicado, y si es así lo inserta al final. La rutina principal del programa podría ser la siguiente :

---

[10] Thomas Kurtz and John Kemmeny, *Structured BASIC Programming.* (Willey. New York, New York. 1987)

[11] Aquí se utiliza el lenguaje BASIC en su versión original. En dialectos modernos del lenguaje, tales como True BASIC y QBasic, no se aplican muchas de las restricciones indicadas aquí.

```
5000 REM ---- Rutina Principal -------------------------------------
5010 Info$ = "Pedro"  : GOSUB 2000 : REM Insertar "Pedro"
5020 Info$ = "Juan"   : GOSUB 2000 : REM Insertar "Juan"
5030 Info$ = "Marcos" : GOSUB 2000 : REM Insertar "Marcos"
5040 Info$ = "Juan"   : GOSUB 1000 : REM Obtener índice de "Juan"
5050 PRINT Resultado
5060 Info$ = "Lucas"  : GOSUB 1000 : REM Obtener índice de "Lucas"
5070 PRINT Resultado
```

La rutina principal utiliza las subrutinas definidas antes sin tener que conocer los detalles de su implementación. Idealmente, lo único que se requiere para usar las subrutinas es conocer su interfase. En este caso, es necesario saber en qué variables se mandan los argumentos y en qué otras quedan los resultados. Al no tener que conocer como una subrutina se implementa se logra la forma más básica de *ocultación de información*, permitiendo que se puedan construir bibliotecas de procedimientos y funciones utilizables por personas ajenas al código original.

El problema con el esquema del ejemplo planteado es que las variables usadas para implementar la tabla, dado que son globales, pueden ser leídas o modificadas por cualquier parte del programa, y esto puede ocurrir de manera intencional o incluso por accidente. Por tanto, el programador debe conocer adicionalmente los nombres de las variables que afectan a una subrutina, y debe comprometerse a no utilizar dichos nombre en formas que afecte la funcionalidad prevista. En programas grandes, elaborados por varias personas, lo anterior es simplemente pedir demasiado.

1.2.2 Módulos

El problema de las subrutinas planteado en la sección anterior puede ser solucionado si el lenguaje de programación provee de algún medio para separar la parte pública de la parte privada de un conjunto de subrutinas relacionadas. Si el usuario de una subrutina no requiere cierta información, entonces no se le debe permitir su acceso a ésta. Los módulos sirven a este propósito. Un módulo está compuesto por una parte pública, en donde se da a conocer al usuario la interfase de las subrutinas, y por una parte privada, donde se implementan de manera protegida las subrutinas.

En un momento se presenta el mismo ejemplo de la tabla indexada de cadenas de carácteres, pero ahora implementada con módulos utilizando el lenguaje Modula-2[12]. En

---

[12] Niklaus Wirth, *Programming in Modula-2.* 2nd Edition (Springer-Verlag. New York, New York. 1985)

este lenguaje, los módulos tienen tres utilidades básicas : a) Almacenar el código del programa principal, b) construir bibliotecas, y c) alterar las reglas de alcance de los identificadores. Los módulos a los que se han hecho referencia en la discusión son los que sirven para construir bibliotecas.

Primero se crea un archivo llamado *módulo de definición* en donde se indica qué partes de la biblioteca son públicas para sus clientes. Todo lo que aparece en este archivo corresponde a la interfase de la biblioteca. Para el ejemplo de la tabla indexada, el módulo de definición sería el siguiente :

```
DEFINITION MODULE TablaIndexada;
    CONST  MaxCadena = 30;
    TYPE   Cadena = ARRAY [0 .. MaxCadena] OF CHAR;
    PROCEDURE Buscar(c : Cadena) : CARDINAL;
    PROCEDURE Insertar(c : Cadena);
END TablaIndexada.
```

La constante, el tipo y los dos procedimientos declarados aquí son de acceso público. La implementación de la biblioteca se encuentra en otro archivo llamado *módulo de implementación*, que se presenta a continuación :

```
IMPLEMENTATION MODULE TablaIndexada;

    FROM IO  IMPORT WrStr;
    FROM Str IMPORT Compare, Copy;
    CONST    MaxTabla   = 20;
    VAR      Tabla : ARRAY [1 .. MaxTabla] OF Cadena;
             Nivel : CARDINAL;

    PROCEDURE Buscar(c : Cadena) : CARDINAL;
        VAR  i : CARDINAL;
        BEGIN
            FOR i := 1 TO Nivel - 1 DO
                IF Compare(c, Tabla[i]) = 0 THEN
                    RETURN i
                END;
            END;
            RETURN 0;
        END Buscar;

        PROCEDURE Insertar(c : Cadena);
        BEGIN
            IF Nivel > MaxTabla THEN
                WrStr("Desborde en tabla");
                HALT;
            END;
            Copy(Tabla[Nivel], c);
            INC(Nivel);
        END Insertar;

BEGIN (* Código de inicialización. *)
    Nivel := 1;
END TablaIndexada.
```

En este archivo se pueden declarar nuevas variables, tipos, procedimientos, etc., los cuales son independientes y desconocidos a cualquier otra parte del programa. El usuario de la biblioteca nada más requiere conocer exclusivamente la interfase de ésta para poder usarla. Ya no existe el peligro de modificar incidentalmente los datos privados pertenecientes a un conjunto de procedimientos. Un programa cliente de esta biblioteca podría ser el siguiente :

```
MODULE Principal;
    FROM IO            IMPORT WrCard, WrLn;
    FROM TablaIndexada IMPORT Buscar, Insertar;
BEGIN
    Insertar("Pedro");
    Insertar("Juan");
    Insertar("Marcos");
    WrCard(Buscar("Juan"), 5);   WrLn;
    WrCard(Buscar("Lucas"), 5);  WrLn;
END Principal.
```

Aunque los módulos corrigen un grave problema que tienen las subrutinas por sí solas, existe todavía otro problema que comparten con ellas. Los dos programas expuestos hasta el momento funcionan bien cuando solamente se requiere una tabla indexada. Pero, ¿qué pasa si se requieren dos o más tablas indexadas? Ni las subrutinas ni los módulos por sí solos consienten esta flexibilidad.

1.2.3 Tipos de datos abstractos

Los módulos son un buen medio para ocultar información, pero no permiten *instanciación*, es decir, la capacidad de poder crear múltiples copias de las áreas de datos. Para resolver este problema surge el concepto de *tipo de dato abstracto* (TDA), que se define como *un tipo de dato definido por el programador* que puede ser manipulado de manera parecida a como son manipulados los tipos básicos incorporados del sistema[13]. Cada TDA tiene asociado un conjunto de valores que puede tomar, y una serie de operaciones que se pueden realizar sobre dichos valores. Según Budd, para tener TDAs se requiere que :

- Se puedan crear instancias múltiples del mismo tipo.
- Se pongan a disposición una serie de operaciones para manipular a las instancias del tipo.
- Se protejan los datos que conforman al tipo, de tal forma que sólo se les pueda tener acceso mediante las operaciones legítimamente establecidas para hacerlo[14].

Como ejemplo, nuevamente se implementará el programa de la tabla indexada, ahora con TDAs utilizando el lenguaje Scheme[15]. Scheme en un lenguaje muy versátil debido principalmente a que las funciones tienen los mismos privilegios que cualquier otro tipo de dato. Esto quiere decir que las funciones se pueden pasar como argumentos, regresar como resultado de otras funciones, e incluso se pueden construir nuevas funciones a tiempo de corrida. La siguiente porción de código, que implementa el TDA esperado, se vale justamente de esta propiedad :

---

13 Budd, *op. cit.*, p. 13.

14 Budd, *ibidem*.

15 William Clinger and Jonathan Rees, *eds.*, *Revised[4] Report on the Algorithmic Language Scheme.* (Lisp Pointers IV, 3, 1991)

```
(define crear-instancia-tabla
  (lambda ()
    (let ((tabla (make-vector 20))
          (nivel 0))

      (let ((buscar
              (lambda (elemento)
                (letrec ((buscar-aux
                           (lambda (indice)
                             (cond
                               ((= indice nivel) 0)
                               ((equal? (vector-ref tabla indice)
                                        elemento)
                                (+ 1 indice))
                               (else (buscar-aux (+ 1 indice)))))))
                  (buscar-aux 0))))

            (insertar
              (lambda (elemento)
                (if (= nivel (vector-length tabla))
                    (error elemento "Desborde de tabla")
                    (begin (vector-set! tabla nivel elemento)
                           (set! nivel (+ 1 nivel)))))))

        (lambda (nombre-operacion)
          (case nombre-operacion
            ((buscar) buscar)
            ((insertar) insertar)
            (else (error nombre-operacion
                         "Operación no comprendida")))))))
```

La función sin argumentos `crear-instancia-tabla` regresa una nueva función cada vez que es aplicada, que para fines conceptuales es una instancia del TDA en discusión. Cada una de estas nuevas funciones generadas cuenta con las siguientes dos características :

- Es una función *despachadora* que funciona de la siguiente manera : cuando es invocada con el argumento `'insertar`, regresa la función local `insertar`; y si es invocada con el símbolo `'buscar` regresa la función local `buscar`.
- Tiene su propia copia de las variables locales `tabla` y `nivel`.

Una vez que se obtienen como resultado las funciones locales `buscar` o `insertar`, solamente se requiere aplicarlas, pasándoles un único argumento que es el elemento que se desea buscar o insertar. Las variables `tabla` y `nivel` utilizadas para este efecto serán las locales de la instancia conceptual involucrada. Para fines de simplicidad de uso, se proveen un par de funciones globales que facilitan el manejo de las instancias :

```
(define insertar
  (lambda (tabla elemento)
    ((tabla 'insertar) elemento)))

(define buscar
  (lambda (tabla elemento)
    ((tabla 'buscar) elemento)))
```

La siguiente serie de expresiones muestra como un programa podría utilizar el TDA definido. Primero se crean las instancias con la función `crear-instancia-tabla`, y se depositan en variables para su uso posterior. Luego a cada instancia por separado se les aplica las funciones globales `insertar` y `buscar`. El comportamiento de cada instancia es justamente el esperado.

```
(define tabla1 (crear-instancia-tabla))
(define tabla2 (crear-instancia-tabla))
(insertar tabla1 "Pedro")
(insertar tabla1 "Juan")
(insertar tabla2 "Marcos")
(buscar tabla1 "Juan")            ; Regresa 2
(buscar tabla2 "Juan")            ; Regresa 0
(buscar tabla2 "Marcos")          ; Regresa 1
```

Esta técnica de manejo de TDAs utilizando Scheme es descrita con mayor detalle por Abelson[16] y Friedman[17].

1.2.4 Objetos

Los *objetos* son en esencia tipos de datos abstractos, pero además incorporan innovaciones importantes sobre el compartimiento y reutilización de código. En particular, los objetos incorporan los conceptos de *herencia* y *polimorfismo*, cuyos significados se estudiarán de manera general en la sección 1.3.

A diferencia del manejo convencional de procedimientos y funciones, en donde una rutina es invocada con ciertos datos como argumentos, el manejo de objetos ocurre desde una perspectiva distinta : *a un objeto se le envía un mensaje indicándole que realice cierta acción*. Bajo este enfoque el agente activo es el objeto y no el código, al revés de como

---

16 Abelson, *op. cit.*, p. 168-174.

17 Daniel Friedman, *et al*. *Essentials of Programming Languages.* (MIT Press. Cambridge, Massachussets. 1992) pp. 126-131.

sucede convencionalmente. Aunque este cambio de enfoque aparenta ser solo a nivel conceptual, en la práctica tiene repercusiones importantes que se discutirán posteriormente.

1.3 Programación orientada a objetos

Booch define a la *programación orientada a objetos* (POO) de la siguiente manera :

> *"La programación orientada a objetos es un método de implementación en donde los programas se organizan como una colección cooperativa de objetos, cada uno de los cuales representa una instancia de alguna clase, y cuyas clases pertenecen a una jerarquía de clases unidas a través de una relación de herencia."*[18]

Un *lenguaje orientado a objetos* debe soportar y encausar la programación orientada a objetos. Se debe hacer la distinción entre los lenguajes que *promueven* la programación orientada a objetos y aquellos que simplemente la *permiten*. En teoría, se puede programar orientado a objetos en lenguajes que no fueron originalmente diseñados con ese propósito, pero el hacerlo constituye una labor extraordinaria.

Parece ser universalmente aceptado que para que un lenguaje sea considerado orientado a objetos debe contar con por lo menos las siguientes tres propiedades : *encapsulamiento*, *polimorfismo* y *herencia*. Existen propiedades adicionales, pero solamente se mencionará aquí una más por ser de interés para este trabajo de tesis : *persistencia*. Cada una de estas propiedades se describe brevemente a continuación. En el siguiente capítulo se retomarán los conceptos introducidos aquí para explicar a detalle la semántica del lenguaje Huntul.

1.3.1 Encapsulamiento

El encapsulamiento es la propiedad que los objetos tomaron prestada de sus primos los tipos de datos abstractos.

Un objeto tiene dos caras. La primera cara, la interfase, es la que el objeto da al mundo exterior; es la que muestra lo que puede hacer, mas no dice cómo lo hace. La otra

---

[18] Booch, *op. cit.*, p. 38.

cara, la implementación, es la que se encarga de hacer el trabajo y de mantener el estado del objeto. Solamente esta cara puede modificar el estado del objeto.

1.3.2 Herencia

Cada objeto es una instancia de una *clase*. Una clase es parecida a lo que se ha manejado como tipo de dato abstracto : es una descripción abstracta de los datos y comportamiento que comparten objetos similares[19]. La *herencia* es la manera de establecer relaciones entre las distintas clases que conforman a un sistema orientado a objetos. Con la herencia se establecen jerarquías del tipo "es un", en donde una subclase hereda la estructura y comportamiento de una o más superclases más generalizadas. Típicamente una subclase especializa a su superclase al aumentar o redefinir la funcionalidad es esta última[20] .

Existen lenguajes que solamente permiten que las subclases tengan un solo antecesor directo, mientras que otros permiten que tengan más de uno. A los primeros se le conoce como *lenguajes con herencia simple*, mientras que a estos últimos se les llama *lenguajes con herencia múltiple*.

Debido a que las clases tienen cierto comportamiento que comparten a través de una jerarquía, se logra reducir el código de un programa y se promueve la reutilización. El atractivo de la herencia consiste en que al requerir un nuevo componente para un programa, en lugar de diseñarlo y construirlo desde cero, se busca una clase que proporcione una funcionalidad lo más parecido a la deseada; una vez encontrada, se crea una nueva clase que herede de ésta, y se agregan y/o modifican únicamente aquellos detalles que sean precisos para obtener el componente que originalmente se estaba necesitando.

Lo anterior suena bien, pero no es tan sencillo de lograr. Para ello se necesita una jerarquía bien diseñada, y para lograr un buen diseño normalmente se requiere de mucha experiencia; pero una vez logrado, los beneficios son realmente compensadores.

1.3.3 Polimorfismo

Tal como se explicó en la sección 1.2.4, en un programa con objetos las acciones ocurren cuando se les mandan *mensajes* a los objetos. El *polimorfismo* es una característica que permite a distintos objetos responder al mismo mensaje de manera única. El polimorfismo

---

[19] Budd, *op. cit.*, p. 372.

[20] Booch, *op. cit.*, p. 514.

permite utilizar clases completamente nuevas en aplicaciones existentes, siendo el único requisito de las nuevas clases el que implementen los mensajes requeridos por la aplicación[21].

Un nombre de variable puede contener a diferentes momentos referencias a objetos de distintas clases que tienen un mismo antecesor. Cuando esto ocurre, dicho nombre de variable puede responder a un conjunto de operaciones comunes de diferentes maneras[22].

1.3.4 Persistencia

Booch define a la persistencia como *"la propiedad de un objeto a través de la cual su existencia trasciende en el tiempo (el objeto sigue existiendo después de que su creador dejó de existir) y/o el espacio (el objeto es movido de la dirección en la que originalmente fue creado)"*[23].

Gabriel explica de manera muy ilustrativa el porqué es necesaria la persistencia en un ambiente de programación orientada a objetos :

> Suponga que usted tuviera un perro al cual estuviera enseñándole a dar vueltas y a hacerse el muertito. Y además suponga que al perro se le olvida todo lo que le enseña cuando lo deja solo. Mientras usted esté frente al perro, éste aprende y recuerda todo lo que le ha enseñado, incluso si esto dura muchos meses; pero al momento en que usted se va, el perro lo olvida todo. Usted pensaría que hay algo malo en este perro porque no puede recordar las cosas, y quizás intentaría devolverlo para obtener un reembolso o darlo a unas amistades que vivan en el campo. Si una persona tuviera este problema seguramente lo llevaríamos a un doctor o lo internaríamos en un hospital.
>
> Pero muchos programas son así : Cuando se están corriendo, construyen información respecto a la tarea siendo realizada; pero cuando terminan, la información se ha ido. Afortunadamente, muchos programas *pueden* ser así de tontos. Sin embargo, en aplicaciones del mundo real (por ejemplo, un sistema de nómina), este comportamiento es inaceptable porque los datos almacenados representan a personas u objetos que existen a través del tiempo, y las representaciones de éstos deben persistir también.
>
> En la programación orientada a objetos, nos enfrentamos más seguido con este problema que bajo los esquemas tradicionales por el hecho de que los

---

21 Digitalk, *Smalltalk/V Windows Tutorial and Programming Handbook.* (Digitalk Inc. Los Angeles, California. 1991) p. 85.

22 Booch, *op. cit.*, p. 517.

23 Booch, *op. cit.*, p.77.

> objetos creados y mantenidos en programas orientados a objetos tienen más parecido a personas que a estructuras de datos; los objetos [al igual que las personas] tienen estado y comportamiento, y un programa típico crea objetos y los manipula[24].

El atributo de persistencia solamente debe estar presente en aquellos objetos que una aplicación requiera mantener entre corridas, de otra forma se estarían almacenando una cantidad probablemente enorme de objetos innecesarios.

La persistencia se logra almacenando en un dispositivo de almacenamiento secundario la información necesaria de un objeto para poder restaurarlo posteriormente. Típicamente la persistencia ha sido dominio de la tecnología de base de datos, por lo que esta propiedad rara vez se ha visto incorporada en la arquitectura básica de los lenguajes orientados a objetos.

En general, tal como Damon lo establece, son dos los medios para lograr la persistencia en un ambiente de programación orientado a objetos :

- Que el programador utilice un esquema, ya sea propio o de terceros, que proporcione la semántica explícita para escribir y leer instancias almacenadas en un dispositivos de almacenamiento masivo. Esta solución requiere a menudo que se escriba código adicional por cada nueva clase que necesite instancias persistentes.
- Que el lenguaje provea de un mecanismo para referirse a objetos persistentes de manera transparente para el programador. Esta es la opción más deseable, pues no impone una carga adicional al diseño ni a la codificación de un programa[25].

Lenguajes como C++ y Smalltalk generalmente utilizan el primer medio para lograr persistencia. El segundo medio es por el momento casi exclusivo de los lenguajes de aplicación para las bases de datos orientadas a objetos, por ejemplo, los lenguajes OPAL y COP para Gemstone y Vbase respectivamente.

---

[24] Richard Gabriel, "Persistence in a Programming Environment", *Dr. Dobb´s Journal.* (No. 195, December, 1992) p. 46.

[25] Craig Damon, "C++ and COP : A Brief Comparison", *Object-Oriented Databases With Applications to CASE, Networks and VLSI CAD.* (Prentice-Hall. Englewood Cliffs, New Jersey. 1991) p. 362.

C A P Í T U L O

2

El Lenguaje de Programación Huntul

El propósito del lenguaje Huntul es el de proporcionar una herramienta adecuada para la enseñanza de los principios de la programación orientada a objetos a estudiantes de habla hispana. La idea es de que Huntul resulte fácil de aprender por personas que ya saben programar de manera estructurada, pero sin sacrificar la consistencia que ofrecen los lenguajes orientados a objetos puros. La implementación de Huntul debe ser modesta en cuanto a sus requerimientos de hardware y software, y debe funcionar de manera similar a como funcionan las herramientas de programación convencional. El lenguaje Huntul debe permitir la construcción de programas de propósito general en donde el modelo de objetos sea aplicable.

El lenguaje Huntul es un lenguaje de programación orientada a objetos con persistencia. Su diseño está influenciado principalmente por el lenguaje Smalltalk, aunque toma elementos de otros lenguajes como Modula-2 y C++.

En este capítulo se explicará la sintaxis y semántica de este lenguaje de manera independiente a su implementación. Para la descripción de la sintaxis se hace uso de la notación extendida de la forma Backus Naur (EBNF), de manera similar a como la utilizan Trembley y Sorenson[1]. Las categorías sintácticas (no terminales) se denotan en itálicas. Los

[1] Jean-Paul Trembley and Paul Soreson, *The Theory and Practice of Compiler Writing* (McGraw-Hill. Singapore. 1989) pp. 64-66.

terminales se denotan entre comillas. Cada categoría sintáctica es definida por una serie finita de *reglas* o *producciones*. Cada regla indica que ciertos valores deben estar en la categoría sintáctica. Cada regla comienza nombrando la categoría siendo definida, seguida por el metasímbolo ::=. Las convenciones usadas aquí son :

(1)	(Z)	Evaluar primero a Z.
(1)	{ Z }	Z se repite cero o más veces.
(1)	[Z]	Z ocurre cero o una vez.
(2)	X Y	Primero ocurre X y luego Y.
(3)	X | Y	Ocurre X o Y, pero no ambas.
	!	Representa una o más nuevas líneas.

El número a la izquierda denota su nivel de precedencia. El número uno tiene la mayor precedencia, mientras que el tres tiene la menor. Cuando dos operadores tienen la misma precedencia, se asocian de izquierda a derecha.

2.1 Características del lenguaje

El lenguaje Huntul cuenta con varias características, las cuales afectan directamente la manera en que debe ser usado. También colaboran a la aceptación o rechazo del lenguaje por parte de sus usuarios potenciales.

2.1.1 Huntul es un lenguaje orientado a objetos puro

El lenguaje Huntul incorpora las propiedades de encapsulamiento, herencia simple y polimorfismo. Tal como se describió en la sección 1.3, estas tres propiedades deben estar presentes para que el lenguaje sea considerado orientado a objetos. Huntul maneja también los conceptos de clase, instancia, paso de mensajes y método.

En el lenguaje Huntul todos los datos son objetos, desde los definidos por el sistema hasta las mismas clases. Esto significa que se cuenta con un sistema homogéneo que opera consistentemente por paso de mensajes. Lo anterior es esencial para que Huntul sea considerado un *lenguaje puro*[2]. Otros ejemplos de lenguajes puros son Eiffel y Smalltalk.

[2] Ben Ezzel, *Object Oriented Programming in Pascal.* (Addison-Wesley. Reading, Massachusetts. 1989) pp. 354-355.

En contraste, los lenguajes que están basados en un lenguaje convencional y que incorporan además las características de la POO reciben el nombre de *lenguajes híbridos*. C++ y Object Pascal son ejemplos de este tipo de lenguajes.

Booch afirma que el paradigma de la programación orientada a objetos tiene como estructura conceptual el *modelo de objetos*. Dicho modelo consta de cuatro elementos mayores : abstracción, encapsulamiento, modularidad y jerarquía[3]. Un lenguaje orientado a objetos no debe tan solo *permitir* trabajar con este modelo, sino que debe *promover* a que así se haga. El lenguaje Huntul está diseñado para promover el modelo de objetos, aunque finalmente es el programador quién debe responsabilizarse de su adecuado seguimiento.

2.1.2 Huntul es un lenguaje imperativo

Existen lenguajes, como ML, cuyo estilo de programación es puramente funcional, es decir, cada expresión es evaluada exclusivamente para obtener su valor. La gran mayoría de los lenguajes de programación funcionan bajo otro esquema conocido como *programación imperativa*, en donde las operaciones se realizan no para obtener su valor sino para lograr algún efecto colateral[4].

La operación más importante manejada en la programación imperativa es la *asignación*. La asignación consiste en modificar el valor contenido en una localidad de la memoria de la computadora. Para entender el porqué este estilo de programación es tan popular basta con recordar la sección 1.1.1 para darse cuenta que la máquina de von Neumman trabaja justamente bajo este esquema. Es válido afirmar que la mayoría de los lenguajes de programación asumen que las computadoras en las que sus programas corren son máquinas basadas en la arquitectura de von Neumman.

El estilo imperativo parece ser el único adecuado para el manejo de objetos. Cada objeto tiene una identidad propia debido a que cuenta con un estado, y ese estado sólo puede ser mutado gracias a que existen asignaciones. Por esta razón Huntul es un lenguaje imperativo.

---

[3] Grady Booch, *Object-Oriented Analysis and Design With Applications* 2nd Edition. (Benjamin Cummings. Redwood City, California. 1994) pp. 40-65.

[4] Daniel Friedman, *et al*. *Essentials of Programming Languages.* (MIT Press. Cambridge, Massachussets. 1992) pp. 117-118.

Los programas escritos en lenguajes imperativos son generalmente más eficientes que los programas escritos en lenguajes funcionales, pero también son más difíciles de probar correctos.

2.1.3 Huntul es un lenguaje de programación estructurada

A menudo se escucha que gracias a la programación orientada a objetos la *programación estructurada* ha pasado a la historia. Esto dista mucho de ser cierto, ya que muchos lenguajes orientados a objetos incorporan las ideas básicas de la programación estructurada, y Huntul no es la excepción. Sethi, refiriéndose a la programación estructurada, dice que en resumidas cuentas *"la estructura del texto de un programa debe ayudar a comprender lo que el programa hace"*[5].

Debido a la misma naturaleza de los lenguajes imperativos, éstos introdujeron la necesidad de contar con mecanismos de control de flujo en los programas. La principal idea que aportó la programación estructurada fue justamente la referente al *flujo de control estructurado*[6], en donde se considera indispensable que todo programa pueda contar con las tres siguientes formas de control :

1. *Secuenciación.* Dado un grupo de sentencias, estás se ejecutan una después de la otra en el orden en que aparecen en el texto del programa.
2. *Selección.* Un conjunto de sentencias se ejecutan solamente si una determinada condición se cumple.
3. *Iteración.* Un conjunto de sentencias se ejecutan de manera repetitiva hasta que cierta condición se cumpla.

El que Huntul utilice las estructuras de control convencionales permite que sea aprendido con mayor facilidad por personas familiarizadas con lenguajes como Pascal o C. Otros lenguajes orientados a objetos que usan este mismo esquema son : C++ y Object Pascal. Ejemplos de lenguajes que usan otros esquemas un tanto diferentes para controlar el flujo de un programa son : Smalltalk, CLOS y SCOOPS.

---

5 Ravi Sethi, *Programming Languages Concepts and Constructs.* (Addison-Wesley. Reading, Massachusets. 1989) pp. 66.

6 Sethi, *op. cit.*, pp. 74-76.

2.1.4 Huntul es un lenguaje con persistencia

A diferencia de la mayoría de los lenguajes orientados a objetos, Huntul incorpora a la persistencia desde su diseño. Para que el estado de un objeto permanezca de una corrida a otra en una aplicación, el programador solamente requiere informar al sistema que dicho objeto es persistente.

2.1.5 Huntul es un lenguaje con manejo automático de memoria

En un lenguaje como C, el programador debe utilizar la instrucción `malloc` o alguna similar para obtener memoria de manera dinámica. Cuando el programa ya no la requiere, se debe explícitamente liberar usando la instrucción `free`. En resumidas cuentas, la administración de la memoria corre por cuenta del programador. Aunque esto puede parecer tener sus ventajas, en la práctica son más los problemas que ocasiona.

Un programa que no libera la memoria que ya no va necesitando se dice que tiene una *fuga de memoria*. Estas porciones de memoria que no son usadas por el programa, tampoco pueden ser reutilizadas para otros propósitos porque para el sistema siguen estando ocupadas. Este es un problema demasiado común, y es bastante difícil de detectar en la mayoría de los casos. Generalmente los programas con este problema funcionan muy bien durante largos periodos de tiempo, y de repente el sistema avisa, aparentemente sin causa, que la memoria está agotada.

Otro problema relacionado es cuando una parte del programa libera una porción de memoria, pero en otra parte del mismo programa se sigue pensando que se tiene acceso a ella. Aquí normalmente sucede que la memoria liberada se comienza a utilizar para otros fines, causando una corrupción de memoria para la parte del programa que no se enteró de la liberación en primer lugar. Normalmente los programas que tienen este problema funcionan bien por un momento, y repentinamente empiezan a fallar sin causa aparente.

Una alternativa a la administración manual de la memoria es la de permitir que el mismo sistema la administre. El sistema lleva el control de todos los objetos creados hasta el momento; cuando se da cuenta que la memoria está agotada, realiza una revisión para encontrar a aquellos objetos que ya nadie ocupa para poder liberar su memoria, haciendo lugar así para nuevos objetos. A lo anterior se le conoce con el poco atractivo nombre de *recolección de basura*. Este concepto ha demostrado ser una necesidad primordial en la mayoría de los lenguajes de programación modernos.

Huntul cuenta con un mecanismo de recolección de basura, y además no requiere del concepto de apuntador como lo manejan lenguajes tales como C o Pascal, por lo que los problemas mencionados antes no existen. Otros lenguajes que también realizan recolección automática de basura son : Smalltalk, Scheme, CLOS y Oberon.

2.1.6 Huntul es un lenguaje con tipos latentes

De acuerdo con Booch, un tipo es *"la definición del dominio de valores permisibles que un objeto puede tener y del conjunto de operaciones que pueden ser realizadas sobre éste"*[7]. Para fines de esta discusión, *tipo* es sinónimo de *clase*.

Huntul maneja el concepto de *tipos latentes*, en oposición a los *tipos manifiestos*. Esto significa que los tipos están asociados a valores y no a variables. Los lenguajes con tipos latentes se les conoce también como *lenguajes débilmente tipados* o *tipados dinámicamente*, mientras que a los que utilizan tipos manifiestos se les conoce como *lenguajes fuertemente tipados* o *tipados estáticamente*. Ejemplos del primer caso son Smalltalk, CLOS, y Snobol; del segundo caso se tienen a C, Modula-2 y Ada.

La ventaja de los lenguajes con tipos latentes es que ofrecen una gran flexibilidad al no requerir reglas complicadas y restrictivas para determinar si un programa es correcto a nivel de tipos. Su desventaja principal es de que muchos errores que podrían ser detectados a tiempo de compilación son detectados hasta tiempo de corrida.

2.1.7 Huntul es un lenguaje en español

La gran mayoría de los lenguajes computacionales utilizados en todo el mundo en general, y en nuestro país en particular, tienen un léxico basado en el idioma inglés. Con el único afán de promover la riqueza de nuestra lengua dentro del ambiente técnico, el léxico del lenguaje Huntul requiere y promueve el uso del idioma español.

Como se verá más adelante, los nombres de variables de Huntul permiten contener a la letra eñe y a las vocales con acento. Así mismo, todas las palabras reservadas del lenguaje están en español.

---

[7] Booch, *op cit.*, p. 519.

2.2 Componentes de un sistema

Una implementación del sistema Huntul debe consistir de por lo menos los dos siguientes componentes :

- Compilador de Huntul.
- Unidad de tiempo de ejecución (UTE).

Cuando se programa con objetos, ya no tiene caso hablar de código y datos por separado. Un objeto es una entidad compuesta por un estado (datos) y un comportamiento definido (código). Por lo tanto, un compilador de un lenguaje orientado a objetos no debe generar código, más bien debe generar una representación de bajo nivel de lo que son lógicamente los objetos. De manera convencional, un compilador toma un archivo fuente de entrada y genera un archivo ejecutable[8] como salida. El compilador de Huntul toma también un archivo fuente como entrada, pero a diferencia del compilador convencional, su salida son objetos de "bajo nivel" que se colocan en un archivo lógico llamado "receptáculo". El receptáculo es el contenedor de todos los objetos persistentes que conforman un sistema en Huntul.

Los archivos fuente de Huntul (ver figura 2.1) son de dos tipos : 1) *módulo de definición de clase* (MDC); y 2) *módulo de aplicación* (MA). Un MDC define los elementos que componen a una clase en particular. Dichos elementos son las variables y métodos de instancias y clase. Cuando el compilador de Huntul toma un MDC de entrada, su salida (el objeto que corresponde a la clase definida) se coloca en el receptáculo correspondiente. Por otro lado, los módulos de aplicación sirven como guión para indicar los eventos principales que han de ocurrir en un programa, además de ser el sitio donde se declaran las variables comunes y persistentes. Si la entrada del compilador de Huntul es un módulo de aplicación, la salida generada es un *archivo de control de ejecución* (ACE). Los módulos de aplicación son simplemente una conveniencia que sirve para indicar, entre otras cosas, dónde comienza la ejecución de un programa. Esta distinción permite, adicionalmente, que varios ACE compartan un mismo receptáculo.

---

[8] Generalmente a la salida de un compilador se le llama "archivo objeto", pero no tiene nada que ver con los objetos en discución. Dicho término no se utiliza para evitar confusiones.

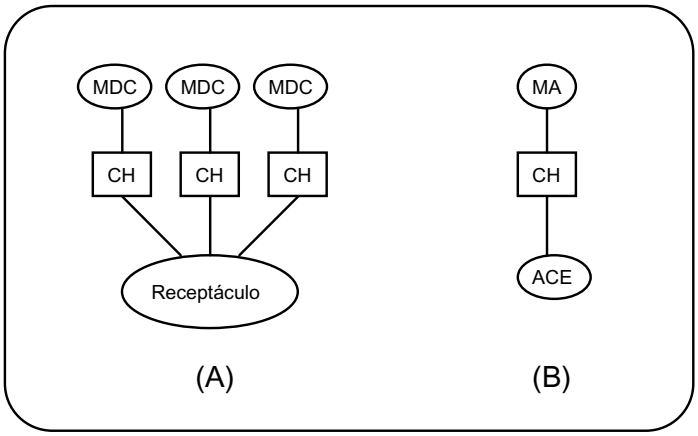


Figura 2.1 Relación entre distintos componentes de un programa en Huntul a tiempo de construcción. A) Un módulo de definición de clase (MDC) es la entrada del compilador de Huntul (CH), y la salida se deposita en el receptáculo. B) Un módulo de aplicación (MA) se compila y genera un archivo de control de ejecución (ACE).

La unidad de tiempo de ejecución (UTE) es el componente encargado de cargar un ACE a memoria, cederle el control, y asistir en general a todas las tareas de bajo nivel requeridas por el sistema (ver figura 2.2). Algunas de estas tareas son : creación de objetos, administración de memoria, recolección de basura, almacenamiento de objetos persistentes, y servicios de las clases primitivas. La UTE es responsable de dejar el receptáculo en un estado consistente al finalizar la ejecución de un programa.

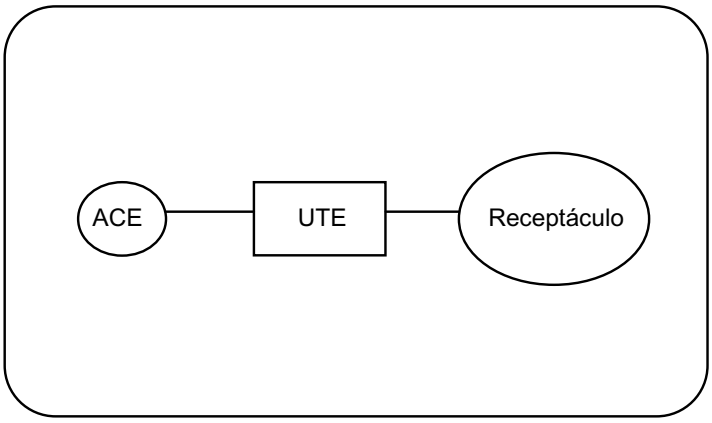


Figura 2.2 Relación entre distintos componentes de un programa de Huntul a tiempo de corrida. El archivo de control de ejecución (ACE) es cargado a memoria por la unidad de tiempo de ejecución (UTE), la cual a su vez tiene una interacción con el receptáculo.

2.3 Convenciones léxicas

2.3.1 Componentes léxicos

Los componentes léxicos (tokens) se dividen en las siguientes categorías : identificadores, palabras reservadas, constantes y carácteres de significado especial. Los carácteres conocidos como *espacios blancos* (espacio, tabulador horizontal, retorno de carro y comentarios), son en general ignorados, salvo que se requieran como delimitadores entre identificadores y palabras reservadas adyacentes.

De la misma manera como ocurre con otros lenguajes de programación, si la secuencia de entrada se ha separado en componentes léxicos hasta un carácter determinado, el siguiente componente es la cadena de carácteres más larga que se puede construir.

2.3.2 Comentarios

Huntul tiene dos tipos de comentarios : de línea y de bloque. Los comentarios de línea comienzan con un carácter de punto y coma (`;`), y terminan al momento en que se encuentra el siguiente retorno de carro. Los comentarios de bloque, por otro lado, comienzan con un carácter de llave izquierda (`{`) y terminan cuando se encuentra un carácter de llave derecha (`}`). No se permite anidar un comentario de bloque dentro de otro de este mismo tipo; pero sí se permite anidar comentarios de línea dentro de un comentario de bloque.

Todo el texto que compone un comentario es ignorado por el compilador. Su función es permitir documentación en línea dentro de los archivos fuentes. La sintaxis de un comentario es :

```
comentario ::=          comentario-línea | comentario-bloque
comentario-línea ::=    ';' { carácter }!
comentario-bloque ::=   '{' { carácter | !} '}'
carácter ::=            cualquier carácter distinto a la nueva línea
```

Ejemplos :

```
; Este es un comentario de línea
{ Este es
  un comentario
  de bloque. }
```

```
{ Este es un comentario de bloque
  ; con comentarios
  ; de línea anidados
}
```

2.3.3 Identificadores

Un identificador debe comenzar con una letra, y estar seguido de cero o más letras, dígitos o carácteres de subrayado (_). Los carácteres á, é, í, ó, ú, ü, Ñ y ñ, pertenecientes al alfabeto español, se consideran letras válidas[9]. Las letras mayúsculas y minúsculas se consideran distintas. Es decir, los indentificadores `Sumatoria`, `sumatoria`, `SUMATORIA` y `SuMaToRiA` no son considerados iguales.

Si un identificador comienza con una letra mayúscula se le llama *identificador compartido*, pero si comienza con una letra minúscula se le llama *indentificador restringido* (el porqué de estos nombres será evidente más adelante). Los identificadores no tienen límite en cuanto a su extensión, pero solamente los primeros 32 carácteres son significativos. La sintaxis de un identificador es como sigue :

ident ::=	*ident-comp* | *ident-rest*
ident-comp ::=	*letra-may* { *letra* | *dígito* | *subrayado* }
ident-rest ::=	letra-min { letra | dígito | subrayado }
letra ::=	*letra-may* | *letra-min*
letra-may ::=	'A' | ... | 'Z' | 'Ñ'
letra-min ::=	'a' | ... | 'z' | 'á' | 'é' | 'í' | 'ó' | 'ú' | 'ü' | 'ñ'
dígito ::=	'0' | ... | '9'
subrayado ::=	'_'

Ejemplos :

```
Un_Identificador_Largo    ; Válido
Ñoño                      ; Válido
Septiembre_16             ; Válido
16_de_Septiembre          ; Inválido, comienza con un dígito
_Otro_Día                 ; Inválido, comienza con un subrayado
Todos@Juntos              ; Inválido, no se permite '@'
```

---

[9] Se asume que el ambiente de implementación cuenta con un código de carácteres que pueda representar las letras españolas mencionadas. Como consecuencia, los sistemas que solamente cuentan con el código ASCII estándar de 7 bits no pueden ser ambientes destino de Huntul. Esta no es una restricción muy grave, puesto que en la actualidad están teniendo mucha difusión algunos códigos más completos, como son el ASCII extendido de IBM, el código de carácteres ANSI, y UNICODE.

2.3.4 Palabras reservadas

Los 24 identificadores que se indican a continuación se consideran palabras con significado especial para el sistema; por tanto, están reservadas y no pueden ser usadas mas que para su propósito original :

```
antecesor     defclase       método        receptor
aplicación    definstancia   nulo          regresa
bajonivel     falso          opción        selección
ciclo         fin            otro          si
clase         hasta          otrosi        var
común         hereda         persistente   verdad
```

Hay que notar que todas las palabras reservadas se escriben en minúsculas y con acentos donde corresponde ortográficamente.

2.3.5 Literales

Instancias de las clases primitivas Nulo, Entero, Booleano, Carácter, Cadena y Arreglo pueden ser representadas en un programa de manera literal (como *constantes*). La implementación es libre de decidir si dos constantes iguales se refieren o no al mismo objeto. La sintaxis de una constante es :

constante ::= *constante-nula* | *constante-entero* | *constante-booleana* | *constante-carácter* | *constante-cadena* | *constante-arreglo*
constante-nulo ::= 'nulo'
constante-entero ::= *núm-decimal* | *núm-hexadecimal*
núm-decimal ::= ['-'] *dígito* { *dígito* }
núm-hexadecimal ::= '$' *dígito-hex* { *dígito-hex* }
dígito ::= '0' | ... | '9'
dígito-hex ::= *dígito* | 'A' | ... | 'F' | 'a' | ... | 'f'
constante-booleana ::= 'verdad' | 'falso'
constante-carácter ::= *car-literal* | *car-ascii*
car-literal ::= ''' *carácter* '''
carácter ::= cualquier carácter distinto a la nueva línea
car-ascii ::= '@' *num-decimal*
constante-cadena ::= '"' { *carácter* } '"'
constante-arreglo ::= '[' [*lista-elementos*] ']'
lista-elementos ::= *constante* { ',' *constante* }

2.3.5.1 Constante de la clase Nulo. La palabra reservada `nulo` denota el único valor constante de esta clase.

2.3.5.2 Constantes de la clase Entero. Una constante que representa a una instancia de la clase Entero es una secuencia de uno o más dígitos, precedida opcionalmente por un signo menos (-). El valor de una instancia entera se almacena internamente en la computadora con 32 bits, por tanto, el rango de estas constantes va desde el -2,147,483,648 hasta el 2,147,483,647. De manera alternativa, una constante entera se puede representar como un número en hexadecimal. Para tal efecto, la secuencia de uno o más dígitos hexadecimales debe ir precedida por un signo de pesos ($). Un dígito hexadecimal es uno de los siguientes : dígitos 0 al 9 o letras A a la F, mayúsculas y minúsculas indistintas. Las constantes hexadecimales no pueden ir precedidas del signo menos, por lo que su rango va del $00000000 al $FFFFFFFF.

2.3.5.3 Constantes de la clase Booleano. Las palabras reservadas `verdad` y `falso` son las dos únicas constantes de la clase Booleano.

2.3.5.4 Constantes de la clase Carácter. Una constante de la clase Carácter es cualquier carácter (excepto el retorno de carro) que esté rodeado por un par de apóstrofes ('). Otra opción para representar constantes de carácter es con un signo de arroba (@) seguido de una secuencia de uno o más dígitos decimales; estos últimos corresponden al código numérico del carácter que se desea representar. La implementación es libre de decidir si los carácteres se almacenan internamente en ocho o dieciséis bits[10], por lo que el rango de las constantes en cuestión puede ir del @0 al @255, o del @0 al @65535.

2.3.5.5 Constantes de la clase Cadena. Las constantes de la clase Cadena son una secuencia de carácteres rodeados por un par de comillas ("). La constante *cadena vacía* se representa por un par de comillas adyacentes (""). Para representar un carácter de comillas dentro de la secuencia de carácteres, basta con colocar juntos un par de comillas en el lugar deseado. El número máximo de carácteres que componen a una constante de esta clase, así como su representación interna, son dependientes de la implementación.

2.3.5.6 Constantes de la clase Arreglo. Una constante de clase Arreglo es una lista delimitada por un par de corchetes ([]), y compuesta por constantes de cualquier clase, incluyendo la clase Arreglo, separadas entre sí por comas. El número máximo de elementos que puede haber en una constante de esta clase, así como los niveles de anidamiento, son dependientes de la implementación.

---

[10] Esta libertad se concede pensando en una futura implantación de Huntul sobre un ambiente que utilice UNICODE, el cual es un código de carácteres de dieciséis bits.

2.4 Conceptos básicos

En esta sección se exponen la mayoría de los conceptos y definiciones que se utilizarán en secciones posteriores.

2.4.1 Objetos

Los objetos son los bloques básicos con los que se construyen programas en Huntul. Todo objeto es una *instancia* de una clase. La clase define la estructura y comportamiento de todas sus instancias.

Un objeto está lógicamente compuesto por datos y por las operaciones que manipulan a dichos datos. Los objetos se pueden considerar como estructuras de datos protegidas. Los datos contenidos dentro de un objeto solamente se pueden accesar a través de *mensajes*.

2.4.2 Variables

Una variable es un identificador que se refiere a una localidad en el espacio de almacenamiento. El valor almacenado en dicho espacio es la dirección de memoria en donde se encuentra un objeto. Dicho de otra forma, una variable contiene un apuntador a un objeto. Cada variable tiene asociado un *alcance* y un *tiempo de vida*.

El nombre de una variable puede ser usado en una expresión para hacer referencia al objeto al cual apunta. Una variable puede contener distintos apuntadores a objetos en diferentes momentos, es decir la variable no tiene un tipo asociado tal como ocurre en muchos otros lenguajes. El objeto referenciado por una variable puede ser cambiado a través de una asignación. Una asignación produce una copia del apuntador al objeto, y no una copia del objeto en sí.

2.4.2.1 Alcance de una variable. Por su alcance, las variables pueden ser *compartidas* o *restringidas*. Una variable compartida es accesible por todo el programa. Una variable restringida solamente la puede accesar un objeto o alguna otra región limitada del programa. Los nombres de las variables compartidas deben comenzar con una letra mayúscula, mientras que los nombres de las variables restringidas deben comenzar con una letra minúscula.

2.4.2.2 Tiempo de vida de una variable. Por su existencia o tiempo de vida, las variables pueden ser *temporales*, *volátiles* o *permanentes*. Las variables temporales son creadas y existen solamente durante la activación de un método. Las variables volátiles existen durante todo el transcurso de la ejecución del programa, pero dejan de existir cuando éste termina. Por último, las variables permanentes son aquellas que retienen su valor lógico aún después de que haya terminado de correr el programa.

2.4.2.3 Categorías de variables. Las variables de Huntul se dividen en cinco categorías, cada una con su propio alcance y tiempo de vida :

- ***Variables locales.*** Son de existencia temporal, e incluyen también los nombres de los parámetros formales (argumentos) asociados a los métodos. Su alcance está siempre restringido a su bloque de declaración.
- ***Variables comunes.*** Son de existencia volátil y de alcance compartido. Se declaran en el módulo de aplicación.
- ***Variables persistentes.*** Son de existencia permanente y de alcance compartido. Se declaran en el módulo de aplicación. Si un objeto persistente contiene a su vez a otros objetos, éstos también son persistentes, y así de manera recursiva.
- ***Variables de clase.*** Son de existencia permanente y de alcance restringido a los métodos de clase en cuestión. Se declaran en un módulo de definición de clase.
- ***Variables de instancia.*** Son los componentes con los que cuenta individualmente cada instancia de una clase determinada. El tiempo de vida de una variable de instancia es el mismo que el de la instancia en cuestión. Su alcance está restringido a los métodos de instancia respectivos, y su declaración se efectúa en un módulo de definición de clase.

2.4.3 Clases

Una clase es una descripción abstracta de datos y comportamientos de una colección de objetos similares. Un objeto que pertenece a dicha colección recibe el nombre de *instancia de esa clase*. Todas las instancias de una clase tienen la misma estructura (cuentan con las mismas variables de instancia), responden a los mismos mensajes, y comparten los mismos métodos.

Para mantener cierta consistencia, las clases en Huntul son también objetos contenidos en variables persistentes, por lo tanto tienen alcance compartido. Por este motivo, los nombres de las clases deben comenzar con una letra mayúscula. El nombre de una clase se puede utilizar dentro de expresiones como cualquier otro objeto. La definición de una clase se hace en el módulo de definición de clase (MDC).

2.4.4 Jerarquía de clases

Las clases forman entre ellas una jerarquía en forma de árbol, siendo la raíz la clase Genérico. Cada clase hereda la funcionalidad de todos sus antecesores (superclases) de la jerarquía. Una subclase se construye agregando a una superclase existente métodos y variables que permitan una funcionalidad adicional a la que ya se tenía.

Como es evidente por el párrafo anterior, Huntul sólo soporta herencia simple, esto es, una clase puede tener a lo más un antecesor directo (padre).

2.4.5 Clases primitivas

El funcionamiento básico del lenguaje Huntul está suministrado por las clases primitivas. Éstas viene a jugar el papel que los tipos del sistema hacen en otros lenguajes. Se les llama primitivas por el hecho de que gran parte de su funcionalidad está compuesta de llamadas a rutinas del sistema de bajo nivel. La definición de dichas clases debe ser provista por la implementación.

Cualquier implementación debe contar con al menos las siguientes ocho clases primitivas :

1. ***Genérico*** Provee el comportamiento común para todos los objetos. Todas las clases deben tener a Genérico como antecesor directo o indirecto. Es posible crear instancias de esta clase, aunque realmente no tiene utilidad hacerlo.
2. ***Nulo*** Toda variable que es creada y que no tiene un valor explícito es inicializada con `nulo`, que es la única instancia de esta clase. Todo método o módulo de aplicación que no regresa un objeto explícito, también regresa a `nulo`.
3. ***Entero*** Las instancias de esta clase son las encargadas de realizar principalmente las operaciones aritméticas convencionales.

4. ***Booleano*** Esta clase maneja los valores lógicos de verdadero y falso, y realiza las operaciones lógicas comunes como la conjunción y disyunción. Las instancias de esta clase son las únicas válidas para la sentencias de control condicional, selectivo e iterativo.

5. ***Carácter*** Representa a carácteres individuales del código requerido por la implementación. Instancias de esta clase pueden ser comparadas o convertidas a su valor entero del código correspondiente.

6. ***Cadena*** Las instancias de esta clase almacenan un número finito de carácteres. Las operaciones que se pueden realizar con ellas incluyen la concatenación, comparación e indexación.

7. ***Arreglo*** Permiten almacenar una secuencia de objetos heterogéneos, cuyo acceso se realiza mediante un índice.

8. ***Código*** Las instancias de esta clase son expresiones compiladas a tiempo de corrida que permiten ser evaluadas en cualquier momento utilizando argumentos.

Cada una de estas clases debe tener a Genérico como antecesor directo, excepto la clase Genérico que es la única clase que no debe tener antecesor. No se permite que ninguna de las clases primitivas, a excepción de Genérico, sea utilizada como antecesor de alguna clase derivada. Por último, ninguna de las clases primitivas tiene permitido tener variables de instancia.

Los mensajes que pueden ser aplicados a estas clases se estudiarán en el capítulo 3.

2.4.6 Clase Metaclase

En Huntul se considera que todas las clases son a su vez objetos, es decir son instancias de alguna clase. A esa clase, la clase de todas las clases, se le conoce como *Metaclase*. La clase Metaclase existe en Huntul sólo a nivel conceptual, y esto es debido a que si existiera realmente habría que decidir de qué clase es instancia, continuando de esta forma con un razonamiento circular al estilo de *"¿quién vino primero, el huevo o la gallina?"*.

Aunque sólo sea a nivel conceptual, la labor principal de la clase `Metaclase` es la de crear clases. Esta función está delegada de forma exclusiva al compilador de Huntul, pues no se provee de un mecanismo para crear clases a tiempo de corrida.

2.4.7 Mensajes y métodos

La mayor parte del procesamiento en Huntul consiste en enviar mensajes a objetos. Los mensajes son el medio con el cual se indica a los objetos que realicen una determinada acción. Cuando un objeto recibe un mensaje, se ejecuta el método correspondiente. Un método es similar en muchos aspectos a una función o procedimiento de otros lenguajes.

Para poder determinar qué método se debe ejecutar, es necesario conocer cuál es el objeto receptor y qué mensaje se le envía. Los métodos que componen a la clase del receptor son examinados para verificar si existe un método que corresponda al mensaje enviado. En caso afirmativo, el método es ejecutado. Cuando lo anterior no ocurre, se continúa la búsqueda en los métodos de la superclase del objeto receptor. Esto se repite hasta que se encuentre un método correspondiente, o hasta que se llega al final de la cadena de las superclases, en cuyo caso se genera un error a tiempo de corrida.

Cuando se invoca un mensaje, el método recibe, además de los argumentos correspondientes, una referencia al objeto receptor del mensaje, la cual es accesible dentro de dicho método a través de la palabra reservada `receptor`. La palabra reservada `antecesor` también es una referencia al mismo objeto receptor, pero cualquier invocación de mensaje sobre ésta garantiza que la búsqueda para determinar el método correspondiente comienza a partir de la superclase inmediata de la clase siendo definida. Lo anterior es útil para invocar un método definido por la superclase que está siendo redefinido por la subclase.

Huntul cuenta con dos tipos de mensajes : *ordinarios* y *binarios*. Los mensajes binarios utilizan notación infija, mientras que los mensajes ordinarios utilizan notación funcional. Estos últimos tiene mayor precedencia que los primeros.

Existen 15 operadores binarios, todos con la misma precedencia y con asociatividad de izquierda a derecha. Los operadores binarios que soporta Huntul son los convencionales para representar expresiones algebraicas, relacionales y booleanas.

Parte de la sintaxis de los mensajes es :

```
mensaje-ord ::=     ':' ident '(' [lista-args] ')'
op-binario ::=      '=' | '==' | '<' | '<=' | '>' | '>=' |
                    '<>' | '&' | '|' | '+' | '-' | '/' |
                    '*' | '%' | '^'
lista-args ::=      expresión {',' expresión}
```

Los métodos, a su vez, se dividen en *métodos de clase* y *métodos de instancia*. Los métodos de clase implementan los mensajes enviados a la clase. El receptor de un mensaje de clase debe ser siempre la clase misma y no una instancia de ésta. Por otro lado, los métodos de instancia implementan los mensajes enviados a las instancias de la clase. Todos los métodos se definen en el MDC de la clase a la cual pertenecen.

2.4.8 Expresiones

Una expresión en Huntul es la combinación de constantes, variables y mensajes que producen como resultado un objeto. Las expresiones tienen la siguiente sintaxis :

expresión ::= *término* { *op-binario término* }
término ::= ('(' *expresión* ')' | 'receptor' | 'antecesor' | *identificador* | *constante*) [*mensaje-ord*]

A continuación se presentan varios ejemplos de expresiones, junto con comentarios que indican la manera en que se debe interpretar cada expresión.

```
9           ; Es una expresión simple que representa al
            ; objeto 9.

2 + 3       ; Al objeto 2 se le aplica el mensaje '+' con un
            ; argumento, el cual es el objeto 3. Regresa como
            ; resultado el objeto 5.

20:neg()    ; Al objeto 20 se le aplica el mensaje 'neg' el
            ; cual no lleva argumentos. Regresa como resultado
            ; el objeto -20.

1 - 4 * 2   ; Al objeto 1 se le aplica el mensaje '-' con un
            ; un argumento que es el objeto 4. El resultado
            ; es el objeto -3, a quien ahora se le aplica el
            ; mensaje '*' con el argumento 2. El resultado
            ; final es el objeto -6. Nótese que no existe la
            ; precedencia normal de operadores que hay en otros
            ; lenguajes.

1 + -6:neg()
            ; Al objeto 1 se la aplica el mensaje '+' con
            ; un argumento, el cual es el resultado de aplicar
            ; al objeto -6 el mensaje sin argumentos 'neg'.
            ; El resultado final es el objeto 7. Nótese
            ; que los mensajes ordinarios (en este caso 'neg')
            ; tienen mayor precedencia que los mensajes
            ; binarios (en este caso '+').
```

```
[10, 4, -7]:obtén(3)
            ; Aplica al objeto [10, 4, -7], el cual es un
            ; arreglo, el mensaje 'obtén' con 3 como argumento.
            ; El resultado es el objeto -7, que corresponde
            ; al índice 3 del arreglo.

6 + 5 = 7 + 4
            ; Aplica al objeto 6 el mensaje '+' con 5 como
            ; argumento. El resultado es el objeto 11, al cual
            ; se le aplica el mensaje '=' con 7 como argumento.
            ; Esto da como resultado parcial al objeto falso,
            ; pues 11 no es igual a 7. Por último, al objeto falso
            ; se le aplica el mensaje '+' con 4 como argumento.
            ; Pero como el objeto falso no sabe como responder al
            ; mensaje '+', se produce un error a tiempo de corrida.

(6 + 5) = (7 + 4)
            ; Al objeto resultante de la subexpresión
            ; '(6 + 5)' se le aplica el mensaje '=' con un
            ; argumento que es el resultado de la subexpresión
            ; '(7 + 4)'. El resultado final es el objeto
            ; verdad.

(6:neg()):neg()
            ; Al objeto resultante de la subexpresión
            ; '(6:neg())' se le aplica el mensaje 'neg' sin
            ; argumentos. El resultado final es el objeto 6.
            ; Nótese que para invocar un mensaje ordinario
            ; sobre el resultado de otro mensaje ordinario,
            ; tal como sucede en este caso, es necesario poner
            ; entre paréntesis la subexpresión que contiene
            ; al mensaje que se ejecuta primero.
```

2.5 Módulo de aplicación (MA)

El propósito del módulo de aplicación ya se explicó en la sección 2.2. Su sintaxis es :

mod-aplic ::= 'aplicación' !
{ *decl-común* | *decl-persist* }
{ *decl-local* }
{ *sentencia* }
'fin' 'aplicación'

decl-común ::= 'común' *lista-vars-comp* !

decl-persist ::= 'persistente' *lista-vars-comp* !

decl-local ::= 'var' *lista-vars-rest* !

lista-vars-comp ::= *ident-comp* { ',' *ident-comp* }

lista-vars-rest ::= *ident-rest* { ',' *ident-rest* }

Como ejemplo de una aplicación muy simple, el ya famoso programa de "¡Hola mundo!" quedaría codificado en las siguientes tres líneas :

```
aplicación
    "¡Hola Mundo!":imprimeNL()
fin aplicación
```

2.5.1 Declaración de variables comunes

Una variable común es similar a las variables globales que existen en otros lenguajes. Las variables comunes sólo pueden declararse en el módulo de aplicación mediante el uso de la palabra reservada `común`. Después de ésta, va una lista de uno o más identificadores compartidos (deben comenzar con mayúscula), separados entre sí por comas. Se pueden tener tantas declaraciones de variables comunes como se desee, siempre y cuando se encuentren antes de la declaración de variables locales y de las sentencias ejecutables. Por ejemplo :

```
aplicación

    común Var1, Var2
    común Var3

    {...}

fin aplicación
```

Las variables `Var1`, `Var2`, `Var3` son comunes, por lo que se les puede tener acceso desde cualquier parte del programa y no solamente desde el módulo de aplicación. Específicamente, cualquier método de cualquier clase puede leer y/o modificar estas variables. Las variables comunes son creadas al momento en que la aplicación en donde son declaradas comienza a correr, y dejan de existir cuando ésta termina. Mientras no se les asigne algún otro valor, estas variables hacen referencia a `nulo`.

2.5.2 Declaración de variables persistentes

Una variable persistente es aquella que su valor sobrevive a la ejecución de la aplicación. Se declaran en el mismo sitio y de manera similar a las variables comunes. La palabra reservada `persistente` denota el inicio de una lista de una o más variables de esta categoría. Los identificadores utilizados para nombrar estas variables deben comenzar con mayúscula. Se pueden intercalar declaraciones de variables comunes con declaraciones de variables persistentes.

Cuando un módulo de aplicación con declaraciones de variables persistentes es compilado, las variables persistentes se inicializan con el objeto `nulo` y se depositan en el receptáculo. Por tal motivo, siempre que se ejecuta una aplicación por primera vez, sus variables persistentes valen `nulo`, y es responsabilidad de la aplicación asignarles otro valor. El siguiente ejemplo muestra de una manera sencilla cómo se usa una variable persistente :

```
aplicación

    persistente Contador     ; La variable Contador es
                             ; persistente.

    si Contador:esNulo()     ; La primera vez que se corre esta
        Contador <- 0        ; aplicación la variable Contador
    fin si                   ; vale nulo y por tanto se
                             ; inicializa con cero.

    Contador <- Contador + 1 ; Cada vez que se ejecuta esta
    Contador:imprimeNL()     ; aplicación, la variable Contador
                             ; se incrementa en uno.
fin aplicación
```

El alcance de las variables persistentes es el mismo que el de las variables comunes, pero con una adición : las variables persistentes pueden ser leídas y/o modificadas por todos los módulos de aplicación que compartan el mismo receptáculo. Hay que recordar que es posible que un receptáculo tenga asociado más de un módulo de aplicación. Una variable persistente puede estar declarada en más de un módulo de aplicación, pero mientras el receptáculo sea compartido la variable es la misma. Inclusive, basta con que se compile un módulo de aplicación que contenga la declaración de una variable persistente para que otros módulos de aplicación tengan acceso a dicha variable, aún sin haberla declarado.

Una variable persistente puede hacer referencia a cualquier objeto creado por el programa. Cuando la aplicación termina de manera normal, el objeto referenciado, el cual se encuentra en memoria, es almacenado en el receptáculo. Cuando se vuelve a ejecutar cualquier aplicación que utilice el mismo receptáculo, el objeto es nuevamente traído a memoria. Si la aplicación termina de manera anormal (por ejemplo, debido a un error de tiempo de corrida) el receptáculo no se actualiza.

En Huntul, un objeto es persistente si cumple con al menos una de las dos siguientes condiciones :

- Está referenciado por una variable persistente.
- Es parte de otro objeto, que a su vez es persistente.

Las variables persistentes permanecen siempre en el receptáculo. La única manera de eliminarlas es reconstruyendo el receptáculo sin volver a compilar algún módulo de aplicación que las declare.

2.5.3 Declaración de variables locales

La variables locales se declaran utilizando a la palabra reservada `var` seguida de una lista de uno o más identificadores restringidos (deben comenzar con minúscula), separados entre sí por comas. Se pueden tener cero o más declaraciones de variables locales, las cuales deben estar localizadas después de las declaraciones de variables comunes y persistentes, pero antes de las sentencias ejecutables.

Estas variables sólo pueden ser leídas y/o modificadas directamente por las sentencias contenidas en el mismo módulo de aplicación. Las variables locales son creadas e inicializadas con `nulo` al momento en que el programa comienza a ejecutarse, y dejan de existir cuando éste finaliza. A continuación se tiene un ejemplo en donde se declaran y usan variables locales :

```
aplicación

    var número, resultado

    "Introduce un número : ":imprime()
    número     <- Entero:lee()
    resultado  <- número ^ 2
    "El cuadrado de este número es : ":imprime()
    resultado:imprimeNL()

fin aplicación
```

2.5.4 Sentencias ejecutables

Después de las declaraciones de todas las variables vienen las sentencias que controlan el curso de la aplicación. En este punto se puede utilizar cualquier sentencia válida; el único detalle a cuidar es que ninguna expresión a este nivel puede utilizar las palabras reservadas

`receptor` ni `antecesor`, pues éstas están reservadas exclusivamente para uso dentro de los métodos.

El uso de la sentencia `regresa` dentro de un módulo de aplicación provoca la terminación del programa, y algún valor numérico asociado a la expresión de esta sentencia es devuelto al sistema operativo. La forma en que sucede esto último es dependiente de la implementación. Si no existe una sentencia `regresa` explícita, el programa termina como si la última sentencia de la aplicación fuera :

```
regresa nulo
```

2.6 Módulo de definición de clase (MDC)

La descripción completa de una clase en Huntul se realiza en un módulo de definición de clase. Es aquí donde se encuentran plasmados los elementos principales de un programa desarrollado en Huntul : jerarquía de clases, variables y métodos de instancia, y variables y métodos de clase. El lenguaje requiere que se utilice un MDC para definir a cada clase de manera individual. Su sintaxis es :

mod-defclase ::= `'clase'` *ident-comp*
`['hereda'` *ident-comp* `]!`
{ *defclase* | *definst* }
`'fin' 'clase'`

defclase ::= `'defclase'!`
{ *decl-var* }
{ *decl-métd* }

definst ::= `'definstancia'!`
{ *decl-var* }
{ *decl-métd* }

decl-var ::= `'var'` *lista-vars* !

decl-métd ::= `'método'` *nombre-métd*
`'('[` *lista-params* `]')'!`
{ *decl-local* }
{ *sentencia* }
`'fin' 'método'!`

nombre-métd ::= *ident-rest* | *op-binario*

lista-params ::= *params* { `','` *params* }

params ::= *ident-rest* `'?'` *ident-comp* |
ident-rest `'!'` *ident-comp*

decl-local ::= `'var'` *lista-vars-rest* !

lista-vars-rest ::= *ident-rest* { `','` *ident-rest* }

Después de la palabra reservada clase se debe indicar el nombre de la clase que se está definiendo, el cual debe comenzar con una letra mayúscula. El nombre de la superclase a la cual se está heredando debe seguir después de la reservada `hereda`. Únicamente la clase Genérico está exenta de indicar algún antecesor.

2.6.1 definstancia y defclase

Las palabras reservadas `definstancia` y `defclase` se utilizan para indicar que de ahí en lo sucesivo y hasta indicar otra cosa se presentan las definiciones de elementos propios de las instancias de la clase (variables y métodos de instancia) o de la clase misma (variables y métodos de clase), según corresponda. Éstas se pueden encontrar en cualquier orden y se pueden repetir un número de veces arbitrario.

2.6.2 Variables de instancia y de clase

Cada instancia de una clase cuenta con un juego propio de *variables de instancia*. Los métodos de instancia son los únicos que puede tener acceso a estas variables, por lo que se dice que son *variables privadas* o *protegidas*. Las variables de instancia son heredables, esto quiere decir que las instancias de una subclase tendrán las mismas variables de instancia que las instancias de las clases antecesoras, más las nuevas variables de instancia que defina su clase.

Una clase, la cual también es un objeto, puede también tener sus propias variables, a las que se les llama *variables de clase*. Puesto que cada clase es única, las variables de clase también lo son. Solamente a los métodos de clase se les tiene permitido leer y/o modificar a las variables de clase. Las variables de clase son heredables, de tal forma que cada subclase cuenta con un juego propio de variables de clase conformado por las suyas propias y las que heredó.

Una subclase no tiene permitido duplicar variables de clase o instancias que ya existan en alguna de sus clases antecesoras.

Para declarar cualquiera de estas dos categorías de variables se utiliza la palabra reservada `var` seguida de una lista de identificadores restringidos (que comienzan con una letra minúscula), separados entre sí por comas. Se permiten cero o más declaraciones de variables, siempre y cuando precedan a las declaraciones de los métodos.

2.6.3 Métodos de instancia y de clase

Los métodos de instancia son la implementación de los mensajes a los que debe responder la instancia de una clase. Por otro lado, los métodos de clase son la implementación de los mensajes a los que debe responder una clase. Los mensajes típicos a los que responden las clases tienen que ver generalmente, si bien no siempre, con la creación de nuevas instancias.

Los métodos se definen comenzando con la palabra reservada `método`, seguido de su nombre, que puede ser un identificador restringido (que comience con una letra minúscula) o un símbolo reconocido como operador binario. Luego viene una lista de parámetros formales (que puede estar vacía), rodeada de una par de paréntesis. Para cada parámetro formal se debe indicar su nombre (que debe ser un identificador restringido), el símbolo `'!'` o `'?'`, seguido del nombre de clase a la cual se está asociando. Los símbolos `'!'` y `'?'` sirven para realizar validación de instanciación de los parámetros actuales a tiempo de corrida. Todo parámetro actual debe estar relacionado al nombre de la clase asociado al parámetro formal bajo las siguientes reglas : Si se usa el símbolo `'!'`, el parámetro actual debe ser exactamente una instancia de la clase asociada al parámetro formal; por otro lado, el símbolo `'?'` indica que el parámetro actual debe ser una instancia de la clase asociada o de cualquier clase que descienda de ésta.

Los métodos cuyos nombres son un operador binario están obligados a declarar un único parámetro formal, pues los dos operandos que requiere el operador de este tipo son el receptor del mensaje y el argumento.

El cuerpo de un método comienza con declaraciones de los nombres de variables locales, que deben ser identificadores restringidos (deben empezar con una letra minúscula). Luego vienen la sentencias que conforman el código del método. Tal como ya se ha dicho, dentro de un método de instancia se tiene acceso a todas las variables de instancia propias y heredadas de la clase, a los parámetros formales y variables locales del método, así como a todas las variables de alcance compartido. Es importante que no existan conflictos ni duplicidad entre los nombres de las variables mencionadas aquí. Los métodos de clase son similares, excepto que en lugar de tener acceso a las variables de instancia se tiene acceso a las variables de clase.

Es válido, y muy a menudo necesario, que una subclase pueda redefinir métodos que estén presentes en cualquiera de sus clases antecesoras.

A continuación se presenta un ejemplo de un MDC completo :

```
clase Alumno hereda Genérico

defclase

    var totalAlumnos
    var sumaCalif

    ;-- Inicializa la clase Alumno.
    método inicializa()
        totalAlumnos <- 0
        sumaCalif <- 0
    fin método

    ;-- Crea una nueva instancia de la clase Alumno.
    método crea(unaMatrícula ! Entero)
        totalAlumnos <- totalAlumnos + 1
        regresa (receptor:nuevo()):inicializa(unaMatrícula)
    fin método

    ;-- Recibe el reporte de la calificación de un alumno
    ;   (llamado por el método de instancia "califica").
    método reporta(unaCalificación ! Entero)
        sumaCalif <- sumaCalif + unaCalificación
    fin método

    ;-- Informa el promedio de todos los alumnos.
    método informa()
        var temp
        "El promedio de ":imprime()
        totalAlumnos:imprime()
        " alumno(s) es : ":imprime()
        temp <- (sumaCalif * 10 / totalAlumnos)
        (temp / 10):imprime()
        ".":imprime()
        (temp % 10):imprimeNL()
    fin método

definstancia

    var matrícula
    var calificación

    ;-- Inicializa una instancia de la clase Alumno
    ;   (llamado por el método de clase "crea").
    método inicializa(unaMatrícula ! Entero)
        matrícula <- unaMatrícula
        regresa receptor
    fin método

    ;-- Califica a un alumno.
    método califica(unaCalificación ! Entero)
        calificación <- unaCalificación
        Alumno:reporta(unaCalificación)
    fin método
```

```
    ;-- Informa matrícula y calificación de un alumno.
    método informa()
        "Alumno con la matrícula : ":imprime()
        matrícula:imprime()
        ", calificación : ":imprime()
        calificación:imprimeNL()
    fin método

fin clase
```

La clase se llama Alumno, y hereda de la clase Genérico. Tiene dos variables de clase (totalAlumnos y sumaCalif), cuatro métodos de clase (inicializa, crea, reporta e informa), dos variables de instancia (matrícula y calificación), y tres métodos de instancia (inicializa, califica e informa).

Tal como resulta evidente, las instancias de esta clase representan a alumnos. La clase proporciona el mecanismo para crear nuevos alumnos; a cada alumno creado se le puede calificar y éste después puede informar qué calificación obtuvo. La clase lleva el control de : 1) cuántos alumnos se han creado hasta el momento; y 2) la suma de las calificaciones reportadas. Cuando un alumno es calificado, éste reporta inmediatamente a su clase la calificación obtenida. Por último, la clase puede informar el promedio de todos los alumnos. Un ejemplo de aplicación que utiliza esta clase es la siguiente :

```
aplicación
    var alumno1, alumno2, alumno3

    ;-- Inicializar la clase Alumno.
    Alumno:inicializa()

    ;-- Crear tres alumnos.
    alumno1 <- Alumno:crea(431528)
    alumno2 <- Alumno:crea(435617)
    alumno3 <- Alumno:crea(438934)

    ;-- Calificar a los tres alumnos.
    alumno1:califica(7)
    alumno2:califica(10)
    alumno3:califica(9)

    ;-- Que los alumnos informen sus calificaciones.
    alumno1:informa()
    alumno2:informa()
    alumno3:informa()

    ;-- Ver el promedio de los alumnos.
    Alumno:informa()
fin aplicación
```

Como se puede observar, la clase Alumno mantiene la información relevante de todas sus instancias, mientras que cada instancia solamente conoce su propio estado.

2.6.4 Receptor de un método

Dentro del cuerpo del método resulta común tener que referirse al objeto receptor del mensaje para poder aplicarle otro mensaje. Por ejemplo, la clase Genérico define al método '<>' (diferente de), tal como se muestra a continuación :

```
método <> (unObjeto ? Genérico)
    regresa (receptor = unObjeto):no()
fin método
```

La palabra reservada `receptor` usada aquí se refiere al mismo objeto al que se le aplicó originalmente el mensaje `'<>'`. Supóngase que se tienen las siguientes sentencias :

```
objetoX <- 4
resultado <- objetoX <> 6
```

El aplicar el mensaje `'<>'` sobre el objeto referenciado por `objetoX` provoca que el método indicado sea invocado. Dentro del método, la subexpresión `(receptor = unObjeto)` equivale a `(objetoX = unObjeto)`, puesto que `objetoX` es el receptor del mensaje. El resto del código del método sigue su curso normal.

2.6.5 Antecesor de un método

Bajo algunas circunstancias, es necesario aplicar al receptor de un método un mensaje que está implementado tanto por la clase propia del receptor como por un antecesor cercano a ésta última, y que por alguna razón especial se requiere utilizar el método del antecesor. Para este caso se cuenta con la palabra reservada `antecesor`. Generalmente su utiliza para invocar un método definido por la superclase que está siendo redefinido por la subclase. Por ejemplo, sea el MDC de la clase Empleado como sigue :

```
clase Empleado hereda Genérico

definstancia

    var nombre, sueldo

    método inicializa(unNombre ! Cadena, unSueldo ! Entero)
        nombre <- unNombre
        sueldo <- unSueldo
    fin método

fin clase
```

Ahora se requiere definir una nueva clase Secretaria que hereda de la clase Empleado :

```
clase Secretaria hereda Empleado

definstancia

    var habilidades

    método inicializa(unNombre ! Cadena, unSueldo ! Entero)
        antecesor:inicializa(unNombre, unSueldo)
        habilidades <- ["mecanografía", "taquigrafía"]
    fin método

fin clase
```

El método `inicializa` de la clase Empleado sigue siendo útil para la clase Secretaria, pero es necesario además inicializar una nueva variable de instancia exclusiva de la subclase. En lugar de duplicar código, el método `inicializa` de Secretaria manda llamar al método de su superclase y agrega solamente una sentencia adicional. Para que lo anterior funcione adecuadamente, la palabra reservada `antecesor` es indispensable. Si en su lugar se utilizara la palabra reservada `receptor`, se provocaría un ciclo recursivo sin fin.

La palabra `antecesor` tiene el mismo efecto que `receptor` cuando se utiliza como argumento de un mensaje.

2.6.6 El valor de regreso de un método

Todos los métodos deben regresar un objeto como resultado. Para indicar el valor de regreso de un método se utiliza la sentencia `regresa`. Si un método no cuenta con esta sentencia de manera explícita, el método regresa al objeto `nulo`. Un método puede tener múltiples sentencias `regresa` si así lo requiere.

2.7 Sentencias

Las sentencias determinan el flujo de control de un programa en Huntul. Excepto en donde así se describe, las sentencias se ejecutan de manera secuencial. La sintaxis general de una sentencia se presenta a continuación :

sentencia ::= *sent-expr* | *sent-asig* | *sent-cond* | *sent-selec* | *sent-itera* | *sent-regresa* | *sent-bajonivel*

Tal como se podrá observar más adelante, cada sentencia debe terminar con una nueva línea. Si se desea dividir una sentencia en más de una línea, se puede utilizar el símbolo de diagonal invertida (\) en el punto donde se quiere partir la sentencia. La partición de una sentencia sólo puede ocurrir en el sitio donde sea válido colocar un espacio en blanco. Después de la diagonal invertida, y antes de la nueva línea, solamente pueden ir espacios en blanco. Ejemplos :

```
x <- 1 + 2 ; Una sentencia normal.
x <- \     ; La misma sentencia de arriba, pero
     1 + \ ; partida en tres líneas.
     2
```

2.7.1 Sentencia de expresión

Cualquier expresión puede ser una sentencia, aunque existen expresiones que por sí solas de nada sirven. Normalmente las expresiones que se utilizan como sentencias son las que contienen invocaciones de mensajes cuyos métodos producen efectos laterales y que además regresan algún objeto que puede ser despreciado. Una sentencia de expresión tiene la siguiente sintaxis :

sent-expr ::= *expresión* !

Ejemplos :

```
1 + 2                 ; Sin utilidad.
(1 + 2):imprimeNL()   ; Imprime el resultado de la
                      ; subexpresión (1 + 2)
```

2.7.2 Sentencia de asignación

Una sentencia de asignación permite a una variable cambiar su objeto referenciado por otro que corresponde al resultado de una expresión. Su sintaxis es la siguiente :

sent-asig ::= *ident* '<-' *expresión* !

El identificador puede ser cualquier nombre de variable excepto los nombres de clase, los cuales se prohibe modificar por considerarse una práctica innecesaria e indeseable.

Ejemplos :

```
unaVariable <- 1 + 2    ; Asigna a unaVariable el objeto
                        ; resultado de la expresión (1 + 2),
                        ; es decir, el objeto 3.

OtraVariable <- [1, -3, 5]:obtén(2)
                        ; Asigna a OtraVariable el objeto
                        ; resultado de la expresión
                        ; ([1, -3, 5]:obtén(2)), esto es,
                        ; -3.
```

2.7.3 Sentencia condicional

La sintaxis de una sentencia condicional es :

sent-cond ::= 'si' *expresión* !
{ *sentencia* }
{ 'otrosi' *expresión* !
{ *sentencia* } }
['otro' !
{ *sentencia* }]
'fin' 'si' !

Las partes del `otrosi` y `otro` son opcionales. Los resultados de las expresiones que aparecen después de las palabras reservadas `si` y `otrosi` deben ser instancias de la clase Booleano, de lo contrario se genera un error a tiempo de corrida. La primera expresión es evaluada siempre, y las restantes sólo se evalúan si la expresión anterior produjo a `falso`. El grupo de sentencias que se ejecuta corresponde a la expresión que produjo a `verdad`. Si todas las expresiones fueron falsas, se ejecutan las sentencias que corresponden a `otro`, en caso de estar presente.

Ejemplos :

```
si x < 10          ; Condición simple.
    y <- x + 1
fin si

si y + x < 20      ; Condición con alternativa.
    x <- 1
otro
    x <- x + 1
fin si

si x = 5           ; Condición mútliple.
    y <- 6
otrosi x + 2 = z
    y <- 1
otrosi x + z - y = 0
    y <- 10
otro
    y <- 0
fin si
```

2.7.4 Sentencia de selección

Una sentencia de selección tiene la siguiente sintaxis :

sent-selec ::= 'selección' *expresión* !
{ 'opción' *expresión* !
{ *sentencia* } }
['otro' !
{ *sentencia* }]
'fin' 'selección' !

Sea $\texttt{expresión}_0$ la expresión que sigue después de la palabra reservada `selección`, y $\texttt{expresión}_i$ la expresión asociada a la i-ésima opción. Solamente se ejecuta el grupo de sentencias asociadas a la i-ésima opción cuando el siguiente mensaje resulta verdadero a tiempo de corrida :

$$(\texttt{expresión}_0) = (\texttt{expresión}_i)$$

Las evaluaciones se hacen en el orden de aparición de las opciones. Una vez que se cumple una opción, ninguna otra es revisada. Si resulta que ninguna opción cumple con la condición indicada, entonces se ejecutan las sentencias que corresponden al grupo de `otro`, si es que éste existe.

Ejemplo :

```
selección x
opción 1
    "uno":imprimeNL()
opción 2
    "dos":imprimeNL()
opción 3
    "tres":imprimeNL()
otro
    "desconocido":imprimeNL()
fin selección
```

2.7.5 Sentencia de iteración

Huntul cuenta con una sola sentencia de iteración. Su sintaxis es :

sent-itera ::= `'ciclo' !`
{ *sentencia* }
`'hasta'` *expresión* `!`
{ *sentencia* }
`'fin' 'ciclo' !`

La expresión después de la palabra reservada `hasta` debe regresar una instancia de la clase Booleano, de otra forma se produce un error a tiempo de ejecución. Cuando se ejecuta un ciclo, primero se ejecutan las sentencias que se encuentran antes de la palabra reservada `hasta`. Luego se evalúa la expresión respectiva, y si resulta cierta se transfiere el control a la siguiente sentencia después de las reservadas `fin ciclo`. De otra forma se ejecutan las sentencias que se encuentran después del hasta, y al llegar al final del ciclo se brinca otra vez al inicio y se repite todo nuevamente.

A este tipo de ciclo se lo conoce como *ciclo con salida*, y solamente tiene un punto de entrada y un punto de salida. Aunque aún no está ampliamente difundido, se ha demostrado que este tipo de ciclos son los más fáciles de entender y los que mejor modelan la forma en que la gente piensa respecto al control iterativo[11].

Ejemplos :

```
i <- 1              ; Ciclo estilo FOR.
ciclo
    hasta i > 10
    { ... }
    i <- i + 1
fin ciclo
```

---

[11] Steve McConnell, *Code Complete.* (Microsoft Press. Redmond, Washington. 1993) pp. 327-238.

```
ciclo               ; Ciclo con la condición al inicio
    hasta x > y     ; (se ejecuta cero o más veces).
    { ... }
fin ciclo

ciclo               ; Ciclo con la condición al final
    { ... }         ; (se ejecuta una o más veces).
    hasta x > y
fin ciclo

ciclo               ; Ciclo con la condición en medio
    { ... }         ; (parte del cuerpo de ejecuta al menos
    hasta x > y     ; una vez, pero la otra parte puede ser
    { ... }         ; que nunca se ejecute).
fin ciclo
```

2.7.6 Sentencia de regreso

Tanto los métodos como los módulos de aplicación regresan un objeto como resultado (ver secciones 2.5.4 y 2.6.6). La forma de especificar el objeto a devolver es utilizando la reservada `regresa`. Su sintaxis es como sigue :

sent-regresa ::= `'regresa'` *expresión* `!`

Ejemplos :

```
regresa 1 + 2       ; Regresar el objeto 3.
regresa verdad      ; Regresar el objeto verdad.
```

2.7.7 Sentencia de bajo nivel

La sintaxis de una sentencia de bajo nivel es :

sent-bajonivel ::= `'bajonivel' !`
`{` *lista-nums* `}`
`'fin' 'bajonivel' !`

lista-nums ::= *constante-entero* `{ ',' ` *constante-entero* `}!`

Las sentencias de bajo nivel permiten insertar directamente en un programa instrucciones en código máquina. Cada constante entera representa un byte, por lo que su rango debe ir del 0 al 255. La interpretación exacta de los números incluidos en esta sentencia es dependiente de la implementación.

C A P Í T U L O

3

Biblioteca de clases primitivas de Huntul

En este capítulo se describen las clases junto con todos los métodos que una implementación mínima de Huntul debe poseer. Aunque el juego de clases y métodos que ofrece Huntul dista mucho de ser tan completo como los ofrecidos por otros lenguajes tales como Smalltalk o Eiffel, sí cumple con la función de proporcionar los elementos más básicos para construir aplicaciones sofisticadas dentro del modelo de objetos.

Las grandes bibliotecas de clases ofrecen muchas ventajas en cuanto a código reutilizable, pero el problema que surge frecuentemente es la dificultad de encontrar la clase y el método que suplen una determinada necesidad, y que por tanto conduce a que se re-escriba código que ya existía antes. Este problema se ve sobre todo con las personas que comienzan a usar un nuevo lenguaje. En Huntul, por ser de naturaleza didáctica, se decidió que la biblioteca de clases y métodos fuera relativamente pequeña, para que de esta forma fuera más fácil de aprender. Las clases primitivas de Huntul y sus métodos respectivos emulan más a los lenguajes convencionales respecto a la funcionalidad que ofrecen los tipos de sistema incluidos en éstos. La filosofía de Huntul es la de permitir la construcción de nuevas clases complejas mediante el uso de instancias de clases primitivas, las cuales sirven como componentes simples pero indispensables para la construcción de programas.

A continuación se detalla cada método de las clases primitivas y se incluye casi siempre un ejemplo sencillo para demostrar su uso. El símbolo griego omega (Ω) en la misma línea que el nombre de un método indica que se realiza una mutación (cambio de estado) sobre el objeto al cual se aplica el mensaje respectivo.

3.1 Clase Genérico

La clase Genérico proporciona el comportamiento común de todos los objetos que conforman un programa en lenguaje Huntul, pues todas las clases heredan de aquí.

3.1.1 Métodos de clase

Todos los métodos de instancia para Genérico se repiten en los métodos de clase. Esto se debe a que Genérico proporciona el comportamiento básico al que deben responder todos los objetos, sean instancias de clases normales o instancias de Metaclase.

A continuación está el único método de clase no incluido en los métodos de instancia.

método nuevo()

Crea una nueva instancia de la clase receptora. Si la clase receptora define variables de instancia, éstas toman a `nulo` como valor inicial. Este es el método que todas las clases derivadas no primitivas deben invocar para crear nuevas instancias.

3.1.2 Métodos de instancia

método = (unObjeto ? Genérico)

Regresa verdad si el receptor es igual a unObjeto, de otra forma regresa falso. A menos de que este método sea redefinido por alguna clase derivada, la acción que se logra es la misma que si se aplica el mensaje '=='.

método == (unObjeto ? Genérico)

Regresa verdad si el receptor es idéntico a unObjeto, de otra forma regresa falso. Dos objetos son idénticos sí son exactamente el mismo objeto, es decir, se refieren a la misma localidad de memoria. Ejemplo:

```
var x, y, z, b1, b2
x <- Genérico:nuevo()
y <- Genérico:nuevo()
z <- x
b1 <- x == y    ; b1 es igual a falso.
b2 <- x == z    ; b2 es igual a verdad.
```

método <> (unObjeto ? Genérico)

Regresa verdad si el receptor es distinto (no igual) a unObjeto, de otra forma regresa falso. Ejemplo:

```
var x, y, z, b1, b2
x <- Genérico:nuevo()
y <- Genérico:nuevo()
z <- x
b1 <- x <> y    ; b1 es igual a verdad.
b2 <- x <> z    ; b2 es igual a falso.
```

método aborta()

Aborta la ejecución de la aplicación, sin actualizar el receptáculo. Regresa al sistema operativo el valor numérico asociado al receptor. Ejemplo:

```
1:aborta()   ; Regresa al sistema operativo el valor
             ; numérico asociado al objeto 1.
```

método comoCadena()

Regresa la representación del receptor como una cadena de carácteres. Si este método no es redefinido por las clases derivadas regresa la cadena "Instancia de *Clase*", donde *Clase* es el nombre de la clase de la cual el receptor es instancia. Ejemplo:

```
var x, c
x <- Genérico:nuevo()
c <- x:comoCadena()      ; c es igual a la cadena
                         ; "Instancia de Genérico".
```

método copia()

Regresa una copia profunda del receptor, es decir un objeto nuevo igual (pero no idéntico) al receptor. Si el receptor está compuesto a su vez por otros objetos, a éstos también se les hace una copia profunda, y así de manera recursiva. Ejemplo:

```
var x, y, z, b1, b2
x <- Genérico:nuevo()
y <- x
z <- x:copia()
b1 <- x == y    ; b1 es igual a verdad.
b2 <- x == z    ; b2 es igual a falso.
```

método error(unMensaje ! Cadena)

Aborta el programa debido a un error detectado a tiempo de corrida, imprimiendo primero el mensaje de error contenido en unMensaje. Ejemplo:

```
var x
x <- Genérico:nuevo()
x:error("Se detectó algún error")
    ; Imprime el mensaje de error y aborta el programa.
```

método esArreglo()

Regresa verdad si el receptor es una instancia de la clase Arreglo, de otra forma regresa falso.

método esBooleano()

Regresa verdad si el receptor es una instancia de la clase Booleano, de otra forma regresa falso.

método esCadena()

Regresa verdad si el receptor es una instancia de la clase Cadena, de otra forma regresa falso.

método esCarácter()

Regresa verdad si el receptor es una instancia de la clase Carácter, de otra forma regresa falso.

método esCódigo()

Regresa verdad si el receptor es una instancia de la clase Código, de otra forma regresa falso.

método esEntero()

Regresa verdad si el receptor es una instancia de la clase Entero, de otra forma regresa falso.

método esMetaclase()

Regresa verdad si el receptor es una instancia de la clase Metaclase (el receptor es una clase), de otra forma regresa falso.

método esNulo()

Regresa verdad si el receptor es una instancia de la clase Nulo, de otra forma regresa falso.

método imprime()

Imprime en la salida estándar (normalmente la pantalla) al receptor en su representación como cadena. Ejemplo:

```
12:imprime()  ; Imprime 12.
```

método imprimeNL()

Imprime en la salida estándar (normalmente la pantalla) al receptor en su representación como cadena seguida de una nueva línea. Ejemplo:

```
25:imprimeNL() ; Imprime 25 seguido de una nueva línea.
```

método nombreClase()

Regresa como cadena el nombre de la clase a la cual el receptor pertenece. Ejemplo:

```
var x, c1, c2
x <- Genérico:nuevo()
c1 <- x:nombreClase()        ; c1 es igual a la cadena
                             ; "Genérico".
c2 <- Genérico:nombreClase() ; c2 es igual a la cadena
                             ; "Metaclase".
```

3.2 Clase Nulo

La clase Nulo sirve para distinguir a objetos que no han tomado un valor inicial apropiado. Su funcionalidad es bastante limitada.

3.2.1 Métodos de clase

método nuevo()

Regresa una nueva instancia de la clase Nulo. Ejemplo:

```
var n1, n2
n1 <- Nulo:nuevo()  ; Estas dos asignaciones tienen el
n2 <- nulo          ; mismo efecto.
```

3.2.2 Métodos de instancia

método = (unNulo ? Genérico)

Regresa verdad si el receptor es igual a unNulo, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- nulo = nulo   ; b1 es igual a verdad.
b2 <- b1 = nulo     ; b2 es igual a falso.
```

método comoCadena()

Regresa la representación del receptor como una cadena. Ejemplo:

```
var n, c
n <- Nulo:nuevo()
c <- n:comoCadena()   ; c es igual a la cadena "nulo".
```

método esNulo()

Regresa verdad si el receptor es una instancia de la clase Nulo, de otra forma regresa falso. Ejemplo:

```
var n, b1, b2
n <- Nulo:nuevo()
b1 <- n:esNulo()      ; b1 es igual a verdad.
b2 <- b1:esNulo()     ; b2 es igual a falso.
```

3.3 Clase Entero

El poder de cómputo numérico de Huntul esta definido en la clase Entero. Los valores numéricos de las instancias se almacenan en 32 bits, utilizando complemento a dos para representar números negativos.

3.3.1 Variables de clase

semilla

Contiene el valor de la semilla utilizada en la generación de números aleatorios.

3.3.2 Métodos de clase

método aleatorio(unRango ! Entero) Ω

Regresa un entero generado aleatoriamente con un valor entre 0 y unRango - 1. Cambia el valor de la semilla por uno nuevo recién calculado. Ejemplo:

```
var x
x <- Entero:aleatorio(10)  ; x es igual a un número
                           ; al azar entre 0 y 9.
```

método lee()

Regresa un entero leído de la entrada estándar (normalmente el teclado). Ejemplo:

```
var x
x <- Entero:lee() ; x es igual a un número leído de la
                  ; entrada estándar.
```

método modificaSemilla(nuevaSemilla ! Entero) Ω

Cambia el valor de la variable de clase semilla por el de nuevaSemilla. Ejemplo:

```
Entero:modificarSemilla(12345) ; Cambia el valor de
                               ; la variable de clase
                               ; semilla por 12345.
```

método nuevo()

Regresa una nueva instancia de la clase Entero igual a cero. Ejemplo:

```
var x
x <- Entero:nuevo()   ; Estas dos asignaciones tienen
x <- 0                ; el mismo efecto.
```

3.3.3 Métodos de instancia

método = (unEntero ? Genérico)

Regresa verdad si el receptor tiene el mismo valor numérico que unEntero, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 10 = 10      ; b1 es igual a verdad.
b2 <- 11 = 10      ; b2 es igual a falso.
```

método < (unEntero ! Entero)

Regresa verdad si el receptor es menor a unEntero, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 1 < 5        ; b1 es igual a verdad.
b2 <- 8 < 6        ; b2 es igual a falso.
```

método <= (unEntero ! Entero)

Regresa verdad si el receptor es menor o igual a unEntero, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 7 <= 6        ; b1 es igual a falso.
b2 <- 8 <= 10       ; b2 es igual a verdad.
```

método > (unEntero ! Entero)

Regresa verdad si el receptor es mayor a unEntero, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 8 > 6        ; b1 es igual a verdad.
b2 <- -8 > -1      ; b2 es igual a falso.
```

método >= (unEntero ! Entero)

Regresa verdad si el receptor es mayor o igual a unEntero, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 5 >= 6        ; b1 es igual a falso.
b2 <- 5 >= 5        ; b2 es igual a verdad.
```

método + (unEntero ! Entero)

Regresa el entero resultante de la adición aritmética del receptor más unEntero. Ejemplo:

```
var x
x <- 110 + 5   ; x es igual a 115.
```

método - (unEntero ! Entero)

Regresa el entero resultante de la substracción aritmética del receptor menos unEntero. Ejemplo:

```
var x
x <- 10 - 24     ; x es igual a -14.
```

método * (unEntero ! Entero)

Regresa el entero resultante del producto aritmético del receptor por unEntero. Ejemplo:

```
var x
x <- 11 * -4     ; x es igual a -44.
```

método / (unEntero ! Entero)

Regresa el entero resultante del cociente aritmético del receptor entre unEntero. Ejemplo:

```
var x
x <- 58 / 10     ; x es igual a 5.
```

método % (unEntero ! Entero)

Regresa el entero resultante del residuo aritmético del receptor entre unEntero. Ejemplo:

```
var x
x <- 73 % 10     ; x es igual a 3.
```

método ^ (unEntero ! Entero)

Regresa el entero resultante de elevar el receptor a la potencia unEntero, la cual debe ser mayor o igual a cero. Ejemplo:

```
var x
x <- 2 ^ 4        ; x es igual a 16.
```

método abs()

Regresa el valor absoluto del receptor. Ejemplo:

```
var x, y
x <- -5:abs()     ; x es igual a 5.
y <- 10:abs()     ; y es igual a 10.
```

método comoCadena()

Regresa la representación del receptor como una cadena. Ejemplo:

```
var c
c <- 128:comoCadena()  ; c es igual a la cadena "128".
```

método comoCarácter()

Regresa la representación del receptor como el carácter equivalente del código utilizado por la implementación. Ejemplo:

```
var c
c <- 65:comoCarácter()  ; c es igual al carácter 'A'.
```

método esCero()

Regresa verdad si el receptor es cero, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 0:esCero()       ; b1 es igual a verdad.
b2 <- -3:esCero()      ; b2 es igual a falso.
```

método esEntero()

Regresa verdad si el receptor es una instancia de la clase Entero, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 12:esEntero()    ; b1 es igual a verdad.
b2 <- b1:esEntero()    ; b2 es igual a falso.
```

método esImpar()

Regresa verdad si el receptor es impar, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 8:esImpar()      ; b1 es igual a falso.
b2 <- 111:esImpar()    ; b2 es igual a verdad.
```

método esNegativo()

Regresa verdad si el receptor es negativo (menor a cero), de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- -10:esNegativo()  ; b1 es igual a verdad.
b2 <- 0:esNegativo()    ; b2 es igual a falso.
```

método esPar()

Regresa verdad si el receptor es par, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 68:esPar()        ; b1 es igual a verdad.
b2 <- 7:esPar()         ; b2 es igual a falso.
```

método esPositivo()

Regresa verdad si el receptor es positivo (mayor o igual a cero), de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- -8:esPositivo()   ; b1 es igual a falso.
b2 <- 0:esPositivo()    ; b2 es igual a verdad.
```

método mayor(unEntero ! Entero)

Regresa al receptor si éste es mayor a unEntero, de otra forma regresa a unEntero. Ejemplo:

```
var x
x <- 67:mayor(34)   ; x es igual a 67.
```

método mcd(unEntero ! Entero)

Regresa el máximo común divisor del receptor y de unEntero. Ejemplo:

```
var x
x <- 45:mcd(20)   ; x es igual a 5.
```

método mcm(unEntero ! Entero)

Regresa el mínimo común múltiplo del receptor y de unEntero. Ejemplo:

```
var x
x <- 45:mcm(20)    ; x es igual a 180.
```

método menor(unEntero ! Entero)

Regresa al receptor si éste es menor a unEntero, de otra forma regresa a unEntero. Ejemplo:

```
var x
x <- 12:menor(-16)   ; x es igual a -16.
```

método neg()

Regresa la negación aritmética del receptor. Ejemplo:

```
var x, y
x <- -6:neg()     ; x es igual a 6.
y <- 9:neg()      ; y es igual a -9.
```

método signo()

Regresa 1 si el receptor es positivo, -1 si es negativo, ó 0 si es igual a cero. Ejemplo:

```
var n1, n2, n3
n1 <- -4:signo()  ; n1 es igual a -1.
n2 <- 7:signo()   ; n2 es igual a 1.
n2 <- 0:signo()   ; n3 es igual a 0.
```

3.4 Clase Booleano

Todas las operaciones lógicas se realizan a través de la clase Booleano. Las sentencias de condición, selección e iteración asumen que sus expresiones correspondientes evalúan a un objeto de esta clase.

3.4.1 Métodos de clase

método nuevo()

Crea una nueva instancia de la clase Booleano igual a falso. Ejemplo:

```
var b1, b2
b1 <- Booleano:nuevo()   ; Estas dos asignaciones
b2 <- falso              ; tienen el mismo efecto.
```

3.4.2 Métodos de instancia

método = (unBooleano ? Genérico)

Regresa verdad si el receptor es igual a unBooleano (tienen el mismo valor lógico), de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- falso = falso      ; b1 es igual a verdad.
b2 <- falso = verdad     ; b2 es igual a falso.
```

método & (unBooleano ! Booleano)

Regresa la conjunción lógica (AND) entre el receptor y unBooleano. Regresa verdad solamente si el receptor y unBooleano son ambos verdad, de otra forma regresa falso. Ejemplo:

```
var b1, b2, b3
b1 <- verdad & verdad    ; b1 es igual a verdad.
b2 <- falso & verdad     ; b2 es igual a falso.
b3 <- falso & falso      ; b3 es igual a falso.
```

método | (unBooleano ! Booleano)

Regresa la disyunción lógica inclusiva (OR) entre el receptor y unBooleano. Regresa verdad si al menos uno de los dos es verdad, de otra forma regresa falso. Ejemplo:

```
var b1, b2, b3
b1 <- falso | falso      ; b1 es igual a falso.
b2 <- falso | verdad     ; b2 es igual a verdad.
b3 <- verdad | verdad    ; b3 es igual a verdad.
```

método ^ (unBooleano ! Booleano)

Regresa la disyunción lógica exclusiva (XOR) entre el receptor y unBooleano. Regresa verdad si el receptor tiene un valor opuesto a unBooleano, de otra forma regresa falso. Ejemplo:

```
var b1, b2, b3
b1 <- verdad ^ falso     ; b1 es igual a verdad.
b2 <- falso ^ falso      ; b2 es igual a falso.
b3 <- verdad ^ verdad    ; b3 es igual a falso.
```

método / (unBooleano ! Booleano)

Regresa la implicación lógica (IMP) entre el receptor y unBooleano. Regresa falso sólo cuando el receptor es verdad y unBooleando es falso, de otra forma regresa verdad. Ejemplo:

```
var b1, b2, b3, b4
b1 <- verdad / verdad     ; b1 es igual a verdad.
b2 <- verdad / falso      ; b2 es igual a falso.
b3 <- falso / verdad      ; b3 es igual a verdad.
b4 <- falso / falso       ; b4 es igual a verdad.
```

método * (unBooleano ! Booleano)

Regresa la equivalencia lógica (EQV) entre el receptor y unBooleano. Regresa verdad cuando el receptor y unBooleano tienen el mismo valor lógico, de otra forma regresa falso. Ejemplo:

```
var b1, b2, b3
b1 <- verdad * falso     ; b1 es igual a falso.
b2 <- falso * falso      ; b2 es igual a verdad.
b3 <- verdad * verdad    ; b3 es igual a verdad.
```

método comoCadena()

Regresa la representación del receptor como una cadena. Ejemplo:

```
var c1, c2
c1 <- falso:comoCadena()   ; c1 es igual a la cadena
                           ; "falso".
c2 <- verdad:comoCadena()  ; c2 es igual a la cadena
                           ; "verdad".
```

método esBooleano()

Regresa verdad si el receptor es una instancia de la clase Booleano, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- verdad:esBooleano()   ; b1 es igual a verdad.
b2 <- b1:esBooleano()       ; b2 es igual a verdad.
```

método no()

Regresa el complemento lógico (NOT) del receptor. Regresa verdad si el receptor es falso, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- verdad:no()    ; b1 es igual a falso.
b2 <- falso:no()     ; b2 es igual a verdad.
```

3.5 Clase Carácter

La clase Carácter representa a carácteres individuales del código utilizado por la implementación. En los ejemplos de esta sección se asume que el código utilizado es el ASCII convencional.

3.5.1 Métodos de clase

método lee()

Regresa un carácter leído de la entrada estándar (normalmente el teclado). Ejemplo:

```
var c
c <- Carácter:lee() ; c es igual a un carácter leído de
                    ; la entrada estándar.
```

método nuevo()

Regresa una nueva instancia de la clase Carácter igual al carácter @0. Ejemplo:

```
var c1, c2
c1 <- Carácter:nuevo()   ; Estas dos asignaciones
c2 <- @0                 ; tienen el mismo efecto.
```

3.5.2 Métodos de instancia

método = (unCarácter ? Genérico)

Regresa verdad si el receptor es igual a unCarácter, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 'A' = @65     ; b1 es igual a verdad.
b2 <- 'a' = 'b'     ; b2 es igual a falso.
```

método < (unCarácter ! Carácter)

Regresa verdad si el receptor es menor a unCarácter, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 'A' < 'B'     ; b1 es igual a verdad.
b2 <- @32 < @13     ; b2 es igual a falso.
```

método <= (unCarácter ! Carácter)

Regresa verdad si el receptor es menor o igual a unCarácter, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- @65 <= 'Z'    ; b1 es igual a verdad.
b2 <- '0' <= @0     ; b2 es igual a falso.
```

método > (unCarácter ! Carácter)

Regresa verdad si el receptor es mayor a unCarácter, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 'b' > 'a'     ; b1 es igual a verdad.
b2 <- @13 > @32     ; b2 es igual a falso.
```

método >= (unCarácter ! Carácter)

Regresa verdad si el receptor es mayor o igual a unCarácter de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 'a' >= @97    ; b1 es igual a verdad.
b2 <- 'a' >= 'b'    ; b2 es igual a falso.
```

método comoAscii()

Regresa la representación del receptor como un entero con el valor ASCII correspondiente. Ejemplo:

```
var n
n <- 'A':comoAscii()  ; n es igual a 65.
```

método comoCadena()

Regresa la representación del receptor como una cadena. Ejemplo:

```
var c
c <- 'X':comoCadena()  ; c es igual a la cadena "X".
```

método comoMayúscula()

Si el receptor es un carácter en minúscula regresa el carácter en mayúscula correspondiente, de otra forma regresa al receptor sin cambio. Ejemplo:

```
var c1, c2
c1 <- 'a':comoMayúscula()  ; c1 es igual al carácter 'A'.
c2 <- '1':comoMayúscula()  ; c2 es igual al carácter '1'.
```

método comoMinúscula()

Si el receptor es un carácter en mayúscula regresa el carácter en minúscula correspondiente, de otra forma regresa al receptor sin cambio. Ejemplo:

```
var c1, c2
c1 <- 'B':comoMinúscula()  ; c1 es igual al carácter 'b'.
c2 <- '?':comoMinúscula()  ; c2 es igual al carácter '?'.
```

método esCarácter()

Regresa verdad si el receptor es una instancia de la clase Carácter, de otra forma regresa falso. Ejemplo.

```
var b1, b2
b1 <- 'a':esCarácter()  ; b1 es igual a verdad.
b2 <- b1:esCarácter()   ; b2 es igual a falso.
```

método esDígito()

Regresa verdad si el receptor es uno de los dígitos del '0' al '9', de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- '7':esDígito()   ; b1 es igual a verdad.
b2 <- '$':esDígito()   ; b2 es igual a falso.
```

método esLetra()

Regresa verdad si el receptor es una letra (mayúscula o minúscula), de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- 'a':esLetra()   ; b1 es igual a verdad.
b2 <- '1':esLetra()   ; b2 es igual a falso.
```

3.6 Clase Cadena

Las instancias de la clase Cadena son una secuencia finita e indexable de instancias de la clase Carácter. El primer elemento de una cadena se encuentra en el índice o posición 1.

Para fines de comparación entre cadenas se sigue la siguiente regla : Se dice que una cadena A es menor que otra cadena B solamente si A se encontrara antes que B en un diccionario. En este caso, el orden del diccionario está dado por el código ASCII.

3.6.1 Métodos de clase

método lee()

Regresa una cadena leída de la entrada estándar (normalmente el teclado). Ejemplo:

```
var c
c <- Cadena:lee()  ; c es igual la cadena leída de
                   ; la entrada estándar.
```

método nuevo()

Regresa una nueva instancia de la clase Cadena igual a la cadena vacía. Ejemplo:

```
var c1, c2
c1 <- Cadena:nuevo()     ; Estas dos asignaciones
c2 <- ""                 ; tienen el mismo efecto.
```

3.6.2 Métodos de instancia

método = (unaCadena ? Genérico)

Regresa verdad si el receptor es igual a unaCadena, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- "algo" = "nada"   ; b1 es igual a falso.
b2 <- "nada" = "nada"   ; b2 es igual a verdad.
```

método < (unaCadena ! Cadena)

Regresa verdad si el receptor es menor a unaCadena, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- "Hola" < "hola"   ; b1 es igual a verdad.
b2 <- "bueno" < "algo"  ; b2 es igual a falso.
```

método <= (unaCadena ! Cadena)

Regresa verdad si el receptor es menor o igual a unaCadena, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- "niña" <= "niñas"  ; b1 es igual a verdad.
b2 <- "feo" <= "bonito"  ; b2 es igual a falso.
```

método > (unaCadena ! Cadena)

Regresa verdad si el receptor es mayor a unaCadena, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- "Hombre" > "Mujer"  ; b1 es igual a falso.
b2 <- "Niño" > "Niña"     ; b2 es igual a verdad.
```

método >= (unaCadena ! Cadena)

Regresa verdad si el receptor es mayor o igual a unaCadena, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- "Pedro" >= "Pedro"  ; b1 es igual a verdad.
b2 <- "Juan" >= "Juana"   ; b2 es igual a falso.
```

método + (unaCadena ! Cadena)

Regresa una cadena que es el resultado de concatenar el receptor con unaCadena. Ejemplo:

```
var c
c <- "Todo " + "junto"  ; c es igual a la cadena
                        ; "Todo junto".
```

método | (unCarácter ! Carácter)

Regresa una cadena que es el resultado de agregar al final del receptor unCarácter. Ejemplo:

```
var c
c <- "persona" | 's'  ; c es igual a la cadena
                      ; "personas".
```

método buscaSubcadena(unaSubcadena ! Cadena)

Regresa la posición donde unaSubcadena aparece por primera vez dentro del receptor. Si no se encuentra regresa 0. Ejemplo:

```
var p1, p2
p1 <- "Esta es una cadena":buscaSubcadena("una")
           ; p1 es igual a 9.
p2 <- "Esta es otra cadena":buscaSubcadena("una")
           ; p2 es igual a 0.
```

método comoCadena()

Regresa la representación del receptor como una cadena, en este caso una copia del receptor. Ejemplo:

```
var c
c <- "Lucas":comoCadena()  ; c es igual a la cadena
                           ; "Lucas".
```

método comoEntero()

Regresa la conversión a entero del receptor, o cero en caso de que no se pueda convertir. El receptor se interpreta como una secuencia de dígitos en base decimal. Ejemplo:

```
var n1, n2
n1 <- "1541":comoEntero()  ; n1 es igual a 1541.
n2 <- "7a12":comoEntero()  ; n2 es igual a 0.
```

método comoMayúsculas()

Regresa una cadena que es el resultado de convertir todos los carácteres del receptor a mayúsculas. Los carácteres que no son letras minúsculas permanecen sin cambio. Ejemplo:

```
var c
c <- "El Número 218!":comoMayúsculas()
      ; c es igual a la cadena "EL NúMERO 218!".
```

método comoMinúsculas()

Regresa una cadena que es el resultado de convertir todos los carácteres del receptor a minúsculas. Los carácteres que no son letras mayúsculas permanecen sin cambio. Ejemplo:

```
var c
c <- "¿Un Mensaje Corto?":comoMinúsculas()
      ; c es igual a la cadena "¿un mensaje corto?".
```

método esCadena()

Regresa verdad si el receptor es una instancia de la clase Cadena, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- "hola":esCadena()  ; b1 es igual a verdad.
b2 <- b1:esCadena()      ; b2 es igual a falso.
```

método longitud()

Regresa el número de carácteres con los que cuenta el receptor. Ejemplo:

```
var n
n <- "algo":longitud()   ; n es igual a 4.
```

método modifica(índice ! Entero, unCarácter ! Carácter) Ω

Coloca en la posición índice del receptor al elemento unCarácter. Regresa al receptor. Ejemplo:

```
var c
c <- "Perro":modifica(3, 'd') ; c es igual a la cadena
                              ; "Pedro".
```

método obtén(índice ! Entero)

Regresa el carácter de la posición índice del receptor. Ejemplo:

```
var c
c <- "Marcos":obtén(3)  ; c es igual al carácter 'r'.
```

método subcadena(inicio ! Entero, contador ! Entero)

Regresa la subcadena del receptor que comienza a partir del carácter del índice inicio y mide un total de contador carácteres. Ejemplo:

```
var c
c <- "Hola Todos":subcadena(3, 2)
        ; c es igual a la cadena "la".
```

método subcadenaDer(contador ! Entero)

Regresa la subcadena que consta de los últimos contador carácteres del receptor. Ejemplo:

```
var c
c <- "Al final":subCadenaDer(5)
        ; c es igual a la cadena "final".
```

método subcadenaIzq(contador ! Entero)

Regresa la subcadena que consta de los primeros contador carácteres del receptor. Ejemplo:

```
var c
c <- "Al inicio":subCadenaDer(2)
        ; c es igual a la cadena "Al".
```

3.7 Clase Arreglo

Las instancias de la clase Arreglo permiten almacenar una secuencia de objetos heterogéneos (instancias de diversas clase), cuyo acceso se realiza mediante un índice. El primer elemento se encuentra siempre en el índice o posición 1.

3.7.1 Métodos de clase

método nuevo(númeroElementos ! Entero)

Crea una nueva instancia de la clase Arreglo de tamaño númeroElementos, cada elemento es igual a la constante nulo. Ejemplo:

```
var a
a <- Arreglo:nuevo(3)    ; Estas dos asignaciones tienen
a <- [nulo, nulo, nulo]  ; el mismo efecto.
```

3.7.2 Métodos de instancia

método = (unArreglo ? Genérico)

Regresa verdad si el receptor es igual a unArreglo (cada elemento del receptor es igual al elemento correspondiente de unArreglo), de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- [1, [2]] = [1, 2]   ; b1 es igual a falso.
b2 <- [1, [2]] = [1, [2]] ; b2 es igual a verdad.
```

método busca(unObjeto ? Genérico)

Regresa la posición en donde unObjeto ocurre por primera vez en el receptor, o cero en caso de no hallarlo. Ejemplo:

```
var p1, p2
p1 <- [1, 2, [], nulo]:busca([])  ; p1 es igual a 3.
p2 <- [1, 2, [verdad]]:busca(3)   ; p2 es igual a 0.
```

método cambiaLongitud(nuevaLongitud ! Entero)

Regresa una nueva instancia de la clase Arreglo de tamaño nuevaLongitud. Cada elemento del nuevo arreglo es el mismo que el correspondiente elemento del receptor, hasta donde lo permita el tamaño del nuevo arreglo. Si el nuevo arreglo es mayor en tamaño que el receptor, los elementos restantes son igualados a la constante nulo. Ejemplo:

```
var a1, a2, a3
a1 <- [1, 2, 3]            ; a1 es igual a [1, 2, 3].
a2 <- a1:cambiaLongitud(2) ; a2 es igual a [1, 2].
a3 <- a2:cambiaLongitud(4) ; a3 es igual a
                           ; [1, 2, nulo, nulo].
```

método comoCadena()

Regresa la representación del receptor como una cadena. Ejemplo:

```
var c
c <- [1, 2, 3]:comoCadena() ; c es igual a la cadena
                            ; "[1 2 3]".
```

método esArreglo()

Regresa verdad si el receptor es una instancia de la clase Arreglo, de otra forma regresa falso. Ejemplo:

```
var b1, b2
b1 <- [1, 2, 3]:esArreglo() ; b1 es igual a verdad.
b2 <- b1:esArreglo()        ; b2 es igual a falso.
```

método filtra(unaExpresiónBooleana ! Código)

Regresa un nuevo arreglo que solamente incluye a los elementos del receptor que al ser evaluados individualmente como argumentos de unaExpresiónBooleana regresan verdad. Ejemplo:

```
var e, a
e <- Código:compila("¿ x ? x:esPar()")
a <- [1, 2, 3, 4]:filtra(e)
     ; a es igual a [2, 4].
```

método longitud()

Regresa el número de elementos con los que cuenta el receptor. Ejemplo:

```
var n
n <- [1, 2, [3, 4]]:longitud()  ; n es igual a 3.
```

método mapea(unaExpresión ! Código)

Regresa un nuevo arreglo en donde cada elemento es el resultado de aplicar unaExpresión a cada elemento correspondiente del receptor. Ejemplo:

```
var e, a
e <- Código:compila("¿ x ? 2 ^ x")
a <- [0, 1, 2, 3, 4]:mapea(e)
     ; a es igual a [1, 2, 4, 8, 16].
```

método modifica(índice ! Entero, unObjeto ? Genérico) Ω

Coloca en la posición índice del receptor al elemento unObjeto. Regresa al receptor. Ejemplo:

```
var a
a <- [verdad, 5, nulo]:modifica(2, [1, 2, 3])
     ; a es igual a [verdad, [1, 2, 3], nulo].
```

método obtén(índice ! Entero)

Regresa el elemento de la posición índice del receptor. Ejemplo:

```
var x
x <- [1, 2, [nulo, 4]]:obtén(3)
     ; x es igual a [nulo, 4].
```

método ordena() Ω

Ordena al receptor de forma ascendente. Regresa al receptor. Es indispensable que todos los elementos del receptor puedan responder al mensaje '<' entre ellos mismos. Ejemplo:

```
var a
a <- [7, 2, 5, 1, 3]
a:ordena()              ; a es igual a [1, 2, 3, 5, 7]
```

método ordenaConExpresión(unaExpresiónBooleana ! Código) Ω

Ordena al receptor utilizando a unaExpresiónBooleana como función de comparación, la cual debe tomar dos argumentos y regresar verdad si los dos argumentos se encuentran entre sí en el orden deseado, de otra forma debe regresar falso. Ejemplo:

```
var a, e
a <- [7, 2, 5, 1, 3]
e <- Código:compila("¿ x, y ? x > y")
a:ordenaConExpresión(e) ; a es igual a [7, 5, 3, 2, 1]
```

3.8 Clase Código

Las instancias de la clase Código son expresiones compiladas a tiempo de corrida que permiten ser evaluadas en cualquier momento utilizando cero o más argumentos, dependiendo de como se hayan construido inicialmente. Para crear una instancia de esta clase se invoca el método de clase *compila* con una cadena como argumento. El contenido de dicha cadena debe sujetarse a la siguiente sintaxis :

cadena-expresión ::= ['¿' *lista-vars-rest* '?'] *expresión*

Las categorías sintácticas *lista-vars-rest* y *expresión* fueron definidas en las secciones 2.5 y 2.4.8 respectivamente. Dentro de la cadena, puede ir una lista de parámetros formales rodeados de los carácteres '¿' y '?', los cuales serán sustituidos por los argumentos deseados al momento en que la expresión sea evaluada.

Gracias a esta clase, Huntul tiene la capacidad de tratar ciertas porciones de código como objetos de primera clase.

3.8.1 Métodos de clase

método compila(unaCadena ! Cadena)

Regresa una instancia de la clase Código resultado de compilar unaCadena. Si existe algún error sintáctico en la cadena, se produce un error a tiempo de corrida. Ejemplo:

```
var c
c <- Código:compila("2 + 3")  ; c es igual una instancia
                              ; de la clase Código que
                              ; corresponde al resultado
                              ; de compilar 2 + 3.
```

método nuevo()

Produce un error, pues no se pueden crear instancias de la clase Código con este método.

3.8.2 Métodos de instancia

método esCódigo()

Regresa verdad si el receptor es una instancia de la clase Código, de otra forma regresa falso. Ejemplo:

```
var b1, b2, e
e <- Código:compila("3 * 4")
b1 <- e:esCódigo()            ; b1 es igual a verdad.
b2 <- b1:esCódigo()           ; b2 es igual a falso.
```

método evalúa()

Evalúa una instancia de la clase Código sin argumentos. Regresa el resultado de la evaluación. Ejemplo:

```
var e, n
e <- Código:compila("12 + (4 * 2)")
n <- e:evalúa()   ; n es igual a 20.
```

método evalúa1(unObjeto ? Genérico)

Evalúa una instancia de la clase Código con un argumento. Regresa el resultado de la evaluación. Ejemplo:

```
var e, n1, n2
e <- Código:compila("¿ x ? (3 * (x ^ 2)) + (2 * x) + 3")
n1 <- e:evalúa1(2)   ; n1 es igual a 19.
n2 <- e:evalúa1(3)   ; n2 es igual a 36.
```

método evalúa2(obj1 ? Genérico, obj2 ? Genérico)

Evalúa una instancia de la clase Código con dos argumentos. Regresa el resultado de la evaluación. Ejemplo:

```
var e, b1, b2
e <- Código:compila("¿ x, y ? (x + y) = (x * y)")
b1 <- e:evalúa2(2, 2)  ; b1 es igual a verdad.
b2 <- e:evalúa2(4, 3)  ; b2 es igual a falso.
```

método evalúa3(obj1 ? Genérico, obj2 ? Genérico, obj3 ? Genérico)

Evalúa una instancia de la clase Código con tres argumentos. Regresa el resultado de la evaluación. Ejemplo:

```
var e, n1, n2
e <- Código:compila("¿ a, b, c ? (a + b + c) / 3")
n1 <- e:evalúa3(5, 10, 15)  ; n1 es igual a 10.
n2 <- e:evalúa3(4, 6, 3)    ; n2 es igual a 4.
```

C A P Í T U L O

4

Implementación del Sistema Huntul

El diseño conceptual del lenguaje Huntul junto con el diseño general del sistema se realizó aproximadamente en dos meses. La implementación en sí, que incluye diseño detallado, programación, depuración, y pruebas del sistema, se hicieron en un periodo un poco mayor a los cuatro meses.

En este capítulo se presentarán a nivel de descripción general las porciones más relevantes de la implementación. Para una descripción más detallada se sugiere consultar directamente la documentación en línea (comentarios) contenida en los archivos fuente que componen al sistema, los cuales se proporcionan en un disco flexible adjunto.

4.1 Programación del sistema

La presente versión del sistema Huntul fue desarrollada para funcionar sobre cualquier equipo PC o compatible basado en el microprocesador de la familia del Intel 8086, corriendo bajo el sistema operativo DOS. Fue desarrollado principalmente en lenguaje C con algunas rutinas de soporte escritas en ensamblador. El compilador utilizado para este propósito fue el Borland C++ versión 3.1, junto con el Turbo Assembler versión 3.0. El equipo en el que se desarrolló fue una computadora Gama 386sx a 20 Mhz, 4 Mb en RAM y 80 Mb en disco duro, corriendo con MS-DOS versión 6.0.

El sistema final está integrado por 15 módulos escritos en C y 7 módulos escritos en ensamblador; adicionalmente, hay 23 archivos de tipo *header*. El número aproximado del

total de líneas escritas en C es de 9000, en ensamblador es de 2000, mientras que de los archivos header son casi otras 2000 líneas. Cabe aclarar que las cifras mencionadas son de archivos con un gran número de comentarios y líneas en blanco que tienen el propósito de ayudar a la legibilidad y el mantenimiento del sistema.

Los módulos del sistema pueden ser clasificados en : módulos de rutinas de soporte general, módulos de rutinas de inicialización y control general, módulos de rutinas para el compilador, módulos de rutinas para el manejo de clases y métodos, y módulos de rutinas de tiempo de corrida. Cada una de estas categorías se describe a continuación.

4.1.1 Módulos de rutinas de soporte general

El sistema Huntul requiere de una administración de memoria bastante peculiar. Se utiliza un gran cantidad de memoria dinámica para diversas estructuras de datos, algunas de las cuales requieren incluso recolección de basura (ver sección 4.3.2). La forma convencional de administración de memoria utilizada por la mayoría de los compiladores de C bajo DOS resulta bastante inadecuada para satisfacer las necesidades del sistema. Por tanto se decidió que se debía implementar un mecanismo propio para controlar de manera más completa la memoria de la computadora.

Ya que es bien conocido que los problemas relacionados con memoria dinámica son los más difíciles de encontrar[1], el manejador implementado debía sustituir a las funciones estándar de C *malloc*, *calloc*, *realloc* y *free*, por otras que fueran más seguras. Las nuevas funciones debían proporcionar la misma funcionalidad que las anteriores pero también debían ayudar a la detección temprana de fugas, inconsistencias, y corrupción de memoria. Se debía contar con dos manejadores distintos de memoria, uno para ser controlado directamente por el código del sistema y otro para ser administrado de manera automática en donde resultara indispensable la recolección de basura.

Dado que las funciones convencionales de memoria dinámica fueron eliminadas por completo, tomando de esta forma la responsabilidad total del manejo y disposición de la memoria, se presentó otro problema un tanto más serio. Muchas funciones estándar de la biblioteca de C, como *printf* por ejemplo, hacen uso extensivo internamente de las funciones de memoria dinámica convencional. Para eliminar toda dependencia por completo, se descartó el uso de todas las funciones estándar de C, y se tuvieron que implementar todas las funciones básicas que fueron resultando necesarias.

---

[1] Steve Maguire, *Writing Solid Code.* (Microsoft Press. Redmond, Washington. 1993) p. 46.

A continuación se describen los módulos compuestos por rutinas que comúnmente provee la biblioteca de C, y que por tanto sirven como apoyo general hacia el resto del sistema :

HU_CAD.C	Funciones para el manejo de cadenas de carácteres.
HU_CAR.C	Funciones para el manejo de carácteres.
HU_ES.C	Funciones para el manejo de entrada y salida con formato.
HU_MEM.C	Funciones para la administración de memoria dinámica convencional (sin recolección de basura).
HU_DOS.ASM	Funciones de interfase con DOS utilizando la interrupción 21h.
HU_i86.ASM	Funciones de interfase con las instrucciones de manipulación de cadenas de bytes del 8086.
HU_SALT.ASM	Funciones para el manejo de excepciones mediante saltos no locales.

4.1.2 Módulos de rutinas de inicialización y control general

Los compiladores de C bajo DOS normalmente proveen un módulo de inicialización que se encarga de todos aquellos detalles que se deben realizar antes de que la función *main* del módulo principal sea llamada. Entre estos detalles están : regresar a DOS la memoria no utilizada, leer y separar los argumentos de la línea de comando, inicializar el código para la emulación del coprocesador matemático, etc. En Borland C++ el nombre del archivo fuente de incialización es C0.ASM, y en su forma binaria se llama C0T.OBJ, C0S.OBJ, C0M.OBJ, C0C.OBJ, C0L.OBJ o C0H.OBJ, dependiendo si el modelo de memoria seleccionado es tiny, small, medium, compact, large o huge, respectivamente.

Como ya se mencionó en el apartado anterior, el sistema Huntul requiere de una disposición distinta de memoria a las ofrecidas por el fabricante del compilador. El módulo de inicialización C0.ASM es el que determina la disposición de la memoria en todo el programa, por tanto se tuvo que proporcionar uno módulo diferente de inicialización que permitiera un control más flexible en este aspecto. Este módulo posteriormente manda llamar a las rutinas que determinan el flujo de control de todo el programa.

Los módulos que intervienen en la inicialización y control del programa son :

HU_INIC.ASM	Módulo de inicialización.
HU_SEGS.ASM	Determina la disposición general de todos los segmentos de datos y código adicionales a los facilitados por el compilador de C.
HU_ISR.ASM	Instalación y control de rutinas de servicio de interrupción requeridas por el sistema.
HU_PRE.C	Rutinas de control para la inicialización de todos los módulos del sistema que lo requieran.
HU_CNTR.C	Control general de la ejecución del sistema.
HU_ERR.C	Manejo de todo tipo de errores.

4.1.3 Módulos de rutinas para el compilador

El compilador del sistema Huntul está compuesto por los siguientes módulos :

HU_LEX.C	Analizador léxico o *scanner* del compilador.
HU_SINT.C	Analizador sintáctico o *parser* , y generador de código y de objetos compilados.
HU_SIMB.C	Manejo de la tabla de símbolos.

4.1.4 Módulos de rutinas para el manejo de clases y métodos

La administración de las clases, junto con sus métodos respectivos, se realiza tanto a tiempo de compilación como a tiempo de corrida. La información de estos elementos es de tipo persistente y por tanto debe ser almacenada en el receptáculo. Los siguientes módulos colaboran a estas labores :

HU_CLS.C	Creación y control de clases.
HU_MTD.C	Creación y control de métodos.

4.1.5 Módulos de rutinas de tiempo de corrida

La unidad de tiempo de ejecución (UTE) requiere de un número considerable de rutinas para soportar la administración de objetos a tiempo de corrida. Los módulos encargados de esto son :

HU_OBJS.C	Manejo de objetos, incluye recolección de basura.
HU_PRIM.C	Funciones de soporte para las clases primitivas.
HU_RTC.ASM	Despacho y control de rutinas de tiempo de corrida.
HU_PRS.C	Manejo de datos persistentes.

4.2 El compilador de Huntul

La traducción de los archivos fuente a su representación de bajo nivel es llevada a acabo por el compilador de Huntul. Los archivos fuente pueden ser módulos de definición de clase o módulos de aplicación. En ambos casos, al receptáculo se le agregan los nuevos datos persistentes resultantes de la compilación. Adicionalmente, si se compila un módulo de aplicación se genera un archivo de control de ejecución (ver sección 2.2). El compilador de Huntul se puede dividir en cuatro partes que realizan una labor claramente definida : analizador léxico, analizador sintáctico, analizador semántico, y generador de código.

4.2.1 Analizador de léxico

El analizador de léxico (*scanner*) lee y convierte una serie de carácteres de entrada (del archivo fuente) en una secuencia de componente léxicos (*tokens*) que sirven de entrada para el analizador sintáctico[2]. A la secuencia de carácteres de entrada que conforman un componente de léxico individual se le llama *lexema*. El analizador de léxico aísla la representación de lexemas de los componentes léxicos para simplificar la labor del analizador sintáctico. Las convenciones léxicas de Huntul se describen detalladamente en la sección 2.3.

---

[2] Alfred Aho, *et al.*, *Compilers Principles, Techniques, and Tools.* (Addison-Wesley. Reading, Massachusetts. 1986) pp. 54-60.

4.2.2 Analizador sintáctico

El análisis sintáctico (*parsing*) es el proceso que consiste en determinar si una cadena de componentes léxicos pueden ser generados por una gramática[3]. Para el compilador de Huntul, se implementó un *analizador sintáctico de descenso recursivo* por resultar sencillo y adecuado para los propósitos del lenguaje. El análisis por descenso recursivo es un método de análisis sintáctico de arriba hacia abajo en donde se ejecutan una serie de funciones recursivas con el propósito de procesar la entrada. Cada símbolo no terminal tiene asociado una función. En la gramática de Huntul, el siguiente componente léxico determina de manera no ambigua el próximo símbolo no terminal a procesar. Por esta razón, el análisis léxico se puede realizar de manera predictiva. Al ser llamadas las funciones durante el procesamiento de la entrada, se construye de manera implícita el árbol de análisis sintáctico de ésta.

A continuación se presenta toda junta la gramática completa del lenguaje Huntul en orden alfabético. Algunas porciones de ella son manejadas por el analizador léxico (las indicadas en la sección 2.3) y el resto por el analizador sintáctico. La notación utilizada para esta gramática es la misma que se describe al inicio del capítulo 2. Las categorías sintácticas iniciales son *mod-aplic* y *mod-defclase*.

```
carácter ::=              cualquier carácter distinto a la nueva línea
car-ascii ::=             '@' num-decimal
car-literal ::=           ''' carácter '''
constante ::=             constante-nula | constante-entero |
                          constante-booleana | constante-carácter |
                          constante-cadena | constante-arreglo
constante-arreglo ::=     '[' [ lista-elementos ] ']'
constante-booleana ::= 'verdad' | 'falso'
constante-cadena ::=      '"' { carácter } '"'
constante-carácter ::=    car-literal | car-ascii
constante-entero ::=      núm-decimal | núm-hexadecimal
constante-nulo ::=        'nulo'
decl-común ::=            'común' lista-vars-comp !
decl-local ::=            'var' lista-vars-rest !
```

---

[3] Aho, *op. cit.,* pp. 40-48.

decl-métd ::= `'método'` *nombre-métd*
`'('` [*lista-params*] `')'` !
{ *decl-local* }
{ *sentencia* }
`'fin' 'método'` !

decl-persist ::= `'persistente'` *lista-vars-comp* !

decl-var ::= `'var'` *lista-vars* !

defclase ::= `'defclase'` !
{ *decl-var* }
{ *decl-métd* }

definst ::= `'definstancia'` !
{ *decl-var* }
{ *decl-métd* }

dígito ::= `'0' | ... | '9'`

dígito-hex ::= *dígito* | `'A' | ... | 'F' | 'a' | ... | 'f'`

expresión ::= *término* { *op-binario término* }

ident ::= *ident-comp* | *ident-rest*

ident-comp ::= *letra-may* { *letra* | *dígito* | *subrayado* }

ident-rest ::= *letra-min* { *letra* | *dígito* | *subrayado* }

letra ::= *letra-may* | *letra-min*

letra-may ::= `'A' | ... | 'Z' | 'Ñ'`

letra-min ::= `'a' | ... | 'z' | 'á' | 'é' | 'í' | 'ó' | 'ú' | 'ü' | 'ñ'`

lista-args ::= *expresión* { `','` *expresión* }

lista-elementos ::= *constante* { `','` *constante* }

lista-nums ::= *constante-entero* { `','` *constante-entero* } !

lista-params ::= *params* { `','` *params* }

lista-vars-comp ::= *ident-comp* { `','` *ident-comp* }

lista-vars-rest ::= *ident-rest* { `','` *ident-rest* }

mensaje-ord ::= `':'` *ident* `'('` [*lista-args*] `')'`

mod-aplic ::= `'aplicación'` !
{ *decl-común* | *decl-persist* }
{ *decl-local* }
{ *sentencia* }
`'fin' 'aplicación'`

mod-defclase ::= `'clase'` *ident-comp*
[`'hereda'` *ident-comp*] !
{ *defclase* | *definst* }
`'fin' 'clase'`

nombre-métd ::= *ident-rest* | *op-binario*

núm-decimal ::= [`'-'`] *dígito* { *dígito* }

núm-hexadecimal ::= `'$'` *dígito-hex* { *dígito-hex* }

```
op-binario ::=      '=' | '==' | '<' | '<=' | '>' | '>=' |
                    '<>' | '&' | '|' | '+' | '-' | '/' |
                    '*' | '%' | '^'
params ::=          ident-rest '?' ident-comp |
                    ident-rest '!' ident-comp
sentencia ::=       sent-expr | sent-asig | sent-cond |
                    sent-selec | sent-itera | sent-regresa |
                    sent-bajonivel
sent-asig ::=       ident '<-' expresión !
sent-bajonivel ::=  'bajonivel' !
                          { lista-nums }
                    'fin' 'bajonivel' !
sent-cond ::=       'si' expresión !
                          { sentencia }
                    { 'otrosi' expresión !
                          { sentencia }}
                    [ 'otro' !
                          { sentencia }]
                    'fin' 'si' !
sent-expr ::=       expresión !
sent-itera ::=      'ciclo' !
                          { sentencia }
                    'hasta' expresión !
                          { sentencia }
                    'fin' 'ciclo' !
sent-regresa ::=    'regresa' expresión !
sent-selec ::=      'selección' expresión !
                    { 'opción' expresión !
                          { sentencia }}
                    [ 'otro' !
                          { sentencia }]
                    'fin' 'selección' !
subrayado ::=       '_'
término ::=         ( '(' expresión ')' | 'receptor' |
                    'antecesor' | identificador | constante )
                    [ mensaje-ord ]
```

4.2.3 Analizador semántico

Junto con el análisis sintáctico se realiza simultáneamente el análisis semántico. Esta fase consiste en realizar las validaciones de semántica que se pueden hacer a tiempo de compilación, por ejemplo : todas las variables restringidas deben estar declaradas antes de ser usadas; toda clase, excepto Genérico, debe heredar de otra clase previamente compilada,

etc. Para ello se utilizan varias estructuras de datos, entre ellas una tabla de símbolos globales, la cual es parte de los datos persistentes del receptáculo. Dicha tabla únicamente incluye a los símbolos compartidos definidos por el usuario, es decir, no incluye las palabras reservadas del lenguaje.

Las validaciones semánticas que no se hacen a tiempo de compilación se deben hacer a tiempo de corrida. Esto es así por el hecho de que Huntul es un lenguaje dinámicamente tipado, y muchas validaciones convencionales de tipos (o clases) no pueden hacerse sino hasta tiempo de corrida. Como ya se ha mencionado, esta es una desventaja en el sentido que muchos errores son detectados hasta que una aplicación es ejecutada, siendo que bajo otro esquema pudieron haber sido detectados a tiempo de compilación. De cualquier forma, para poder sacar ventaja del polimorfismo, es necesario evitar validaciones estrictas a tiempo de compilación, y por esta razón es que Huntul se diseñó dinámicamente tipado.

4.2.4 Generador de código

Al igual que el analizador semántico, el generador de código también funciona de manera integral con el analizador sintáctico. La implementación del compilador de Huntul realiza la generación de código a través de la especificación sintáctica. El resultado de este proceso es una secuencia de códigos de operación (*opcodes*) del 8086 junto con sus parámetros requeridos. El código generado por los MDCs y los MAs es prácticamente el mismo, la diferencia principal radica en el lugar en donde dicho código es depositado.

4.3 Manejo de memoria

El manejo de memoria del sistema Huntul está fuertemente condicionado por la arquitectura del microprocesador Intel 8086. La principal restricción radica en que cualquier porción del sistema no puede extenderse más allá de un segmento de 64 Kb. Para poder tener acceso a diferentes regiones de memoria se requieren de dos tipos de apuntadores : 1) apuntadores intrasegmentales, que miden 16 bits y que sirven sólo dentro de un segmento específico; y 2) apuntadores intersegmentales, que miden 32 bits y permiten el acceso a cualquier porción de memoria dentro de 1 Mb. El sistema Huntul utiliza principalmente los del segundo tipo, por lo que todos los módulos en C fueron compilados utilizando el modelo de memoria *large*, ya que este modelo trabaja por omisión con apuntadores intersegmentales.

A continuación se presenta en orden la forma en que toda la memoria requerida por el sistema es organizada :

Nombre del Segmento	Tamaño en bytes	Descripción
_TEXT	2,039	Código escrito en ensamblador.
HU_XXX_TEXT	31,598	Código escrito en C.
_DATA	7,056	Datos estáticos con valor inicial exp (datos inicializados).
_BSS	34,080	Datos estáticos con valor inicial cer implícito (datos no inicializados).
PILA	8,192	Pila del sistema.
_SEGS_TABLA_SIMBOLOS	65,536	Tabla de símbols globales.
_SEGS_MONTICULO_GENERAL	65,536	Montículo de memoria dinámica administrada de forma manual.
_SEGS_MONTICULO_OBJETOS	65,536	Montículo de memoria dinámica administrada de forma automática (almacenamiento de objetos).
_SEGS_CODIGO_METODOS	65,536	Código de los métodos y de la aplic en ejecución.

4.3.1 Memoria dinámica convencional

Tal como se mencionó en la sección 4.1.1, la memoria dinámica del sistema tiene una administración distinta a la ofrecida por la biblioteca estándar de C. Las funciones implementadas para el manejo de memoria dinámica realizan las siguientes labores : asignación de nueva memoria, liberación de memoria ya no requerida, y modificación del tamaño de una porción de memoria previamente asignada. Además, estas rutinas realizan siempre que es posible una verificación de consistencia para evitar problemas de corrupción de memoria. Al final de la ejecución del sistema se reporta si hay fugas de memoria y si se efectuaron asignaciones a la memoria apuntada por NULL.

El conjunto de las rutinas para el manejo de memoria dinámica proporcionaron un medio confiable para garantizar el correcto funcionamiento del sistema, por lo menos en lo que a este aspecto se refiere. Es muy probable que sin este esquema algunos errores de memoria nunca se hubieran detectado o su detección hubiera sido muy difícil de lograr. La mayoría de los errores de memoria pudieron ser detectados desde la primera prueba, agilizando así el proceso de depuración del sistema.

4.3.2 Memoria dinámica para objetos

En el sistema Huntul existe un segmento de 64 Kb dedicado exclusivamente al manejo de los objetos, los cuales son dinámicos por su misma naturaleza. La estructura de datos utilizada por cada objeto se muestra en la figura 4.1.

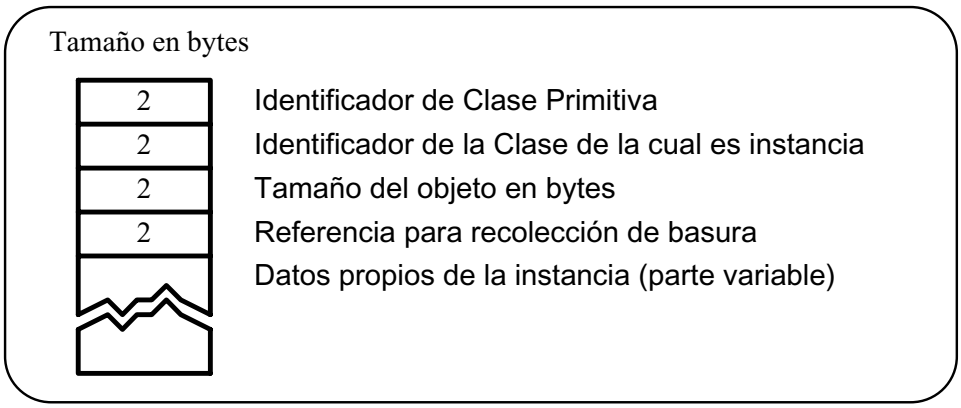


Figura 4.1 Estructura de datos de un objeto.

Los primeros dos bytes de un objeto se utilizan para reconocer la clase primitiva a la que pertenece. Puede ser uno de los siguiente valores :

0 Instancia de la clase Nulo.
1 Instancia de la clase Genérico.
2 Instancia de la clase Booleano.
3 Instancia de la clase Carácter.
4 Instancia de la clase Entero.
8 Instancia de la clase Cadena.
9 Instancia de la clase Arreglo.
10 Instancia de la clase Código.
11 Instancia de la clase Metaclase.
12 Instancia de alguna clase derivada (no primitiva).

Los dos siguientes bytes indican la clase de la cual el objeto es instancia. Este es un identificador de 16 bits generado por el manejador de la tabla de símbolos. Para los objetos que son a su vez clases (instancias de Metaclase), este identificador corresponde a la clase a la que representan.

Los dos bytes que corresponden al tamaño del objeto y los siguientes dos que indican una referencia al objeto, son utilizados por el proceso de recolección de basura. Los últimos bytes son los que almacenan la información propia del objeto. Para objetos simples (instancias de las clases Nulo, Entero, Booleano, Carácter y Entero) esta porción mide 4 bytes, y se guarda aquí lo que es un valor numérico propiamente dicho. Para las clases compuestas (instancias de las clases Cadena, Arreglo, Código, Metaclase y derivadas) el tamaño es variable y parte de la información pueden ser apuntadores a otros objetos.

Todos los objetos son creados a solicitud del programa. Cuando un objeto ya no es útil, el sistema puede decidir reclamarlo para que la memoria utilizada por éste sea liberada. Lo anterior ocurre de manera automática sin la intervención del programador. Para este efecto se requiere de un esquema de recolección de basura.

En el sistema Huntul se implementó un algoritmo de recolección de basura conocido como *colección por copiado*[4]. En este algoritmo la memoria es dividida en dos espacios, y los objetos son colocados en uno de estos espacios hasta que se llene. Cuando esto ocurre, todos los objetos accesibles son copiados al otro espacio, ajustando también los apuntadores de éstos para que apunten a las nuevas copias. Los objetos copiados distan de llenar el nuevo espacio porque debió haber habido algo de basura en el espacio anterior. Cuando continúa la ejecución del programa, cualquier objeto nuevo es alojado en el nuevo espacio. Si este último se llena, los papeles de los espacios se intercambian y se repite el proceso.

---

[4] Andrew Appel, *Topics in Advanced Language Implementations.* (MIT Press. Cambridge, Massachusetts. 1990) pp. 89-94.

4.4 Estructuras de archivo

La presente implementación de Huntul utiliza cuatro tipos diferentes de archivos. Tres de ellos almacenan la información del receptáculo y el restante corresponde al archivo de control de ejecución. A continuación se detalla la estructura de cada uno.

4.4.1 Archivo del receptáculo para almacenar clases

La figura 4.2 muestra la estructura del archivo para almacenar la información referente a las clases que componen el receptáculo.

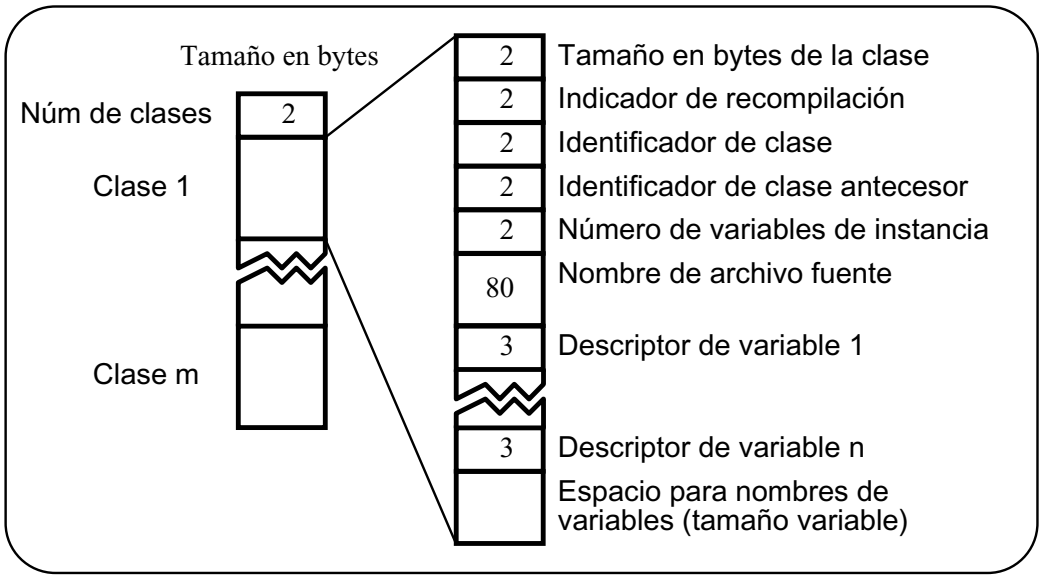


Figura 4.2 Estructura de archivo de clases.

Los primeros dos bytes del archivo indican el número de clases que se encuentran almacenadas en éste. Posteriormente viene la información individual de cada clase. El campo de tamaño indica el número de bytes total ocupados por la clase. El campo de indicador de recompilación es una bandera que informa si la clase requiere ser recompilada; una clase requiere ser recompilada si algún antecesor suyo cambió su disposición o número de variables de clase o instancia. El sistema siempre está pendiente de esta bandera para efectuar las recompilaciones de manera automática y garantizar así la consistencia del sistema.

El campo de identificador de clase es un número único generado por el manejador de la tabla de símbolos. Después viene el campo de indentificador de la clase antecesor. El campo de nombre del archivo fuente indica el nombre completo del módulo de definición de clase correspondiente; éste se utiliza cuando se requiere recompilar la clase. Después viene un arreglo de tamaño variable donde cada entrada es un descriptor de variable. Un descriptor de variable está formado por un byte de control (indica si la variable es de instancia, de clase, o si es la última entrada en el arreglo), y por un apuntador de dos bytes al espacio de nombres de variables (el último campo de cada clase del archivo) para indicar donde inicia el nombre de la variable respectiva.

4.4.2 Archivo del receptáculo para almacenar métodos

La estructura del archivo de métodos se muestra en la figura 4.3. Algunos campos tienen una función similar a los del archivo de clases. Los nombres de los campos son en general explícitos. Cabe hacer la observación de que este archivo está dividido por clases, cada clase tiene varios métodos, y cada método tiene varios argumentos. Cada argumento está representado por un descriptor, el cual está conformado por un byte de control y dos bytes para un identificador de clase, los cuales en conjunto determinan de qué clase se espera el parámetro actual y si éste debe ser una instancia exacta o puede ser derivada.

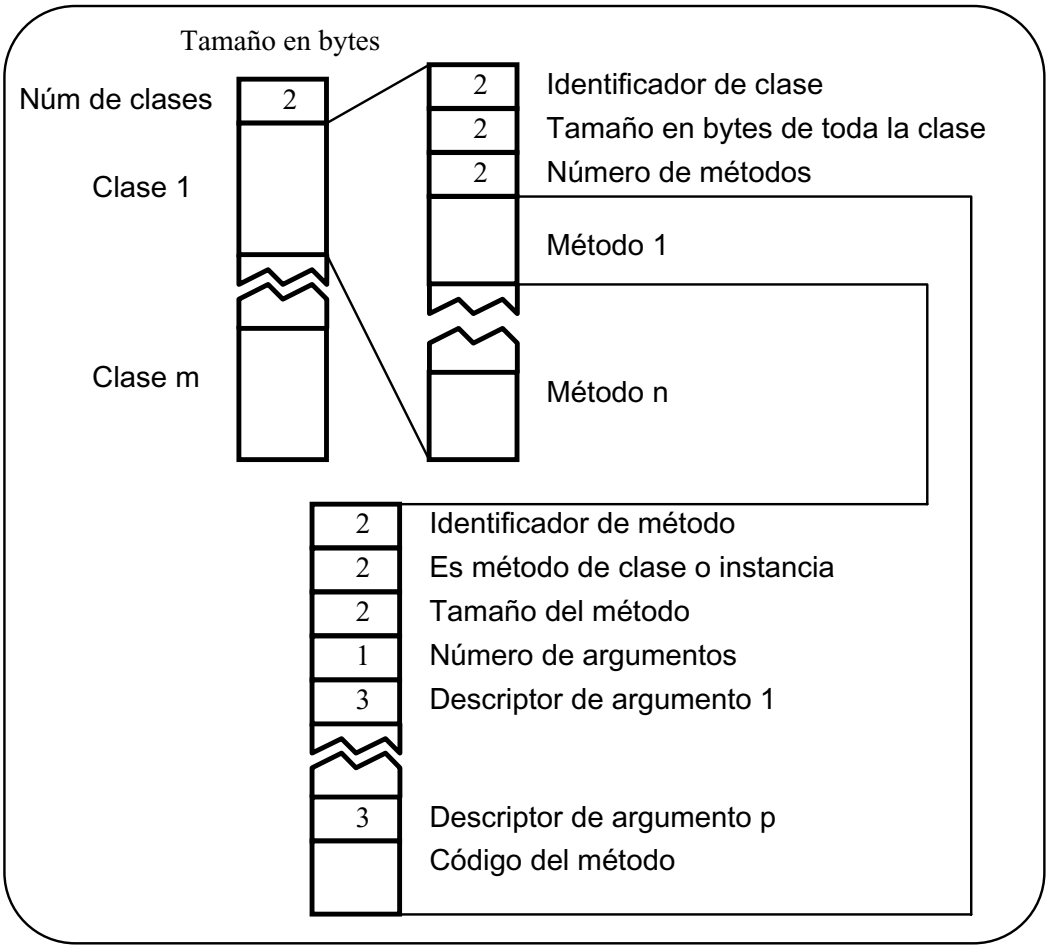


Figura 4.3 Estructura de archivo de métodos.

4.4.3 Archivo del receptáculo para almacenar datos persistentes

Los datos persistentes, que incluyen tabla de símbolos, objetos persistentes, tabla de variables persistentes, y literales de cadenas de carácter, se almacenan en la estructura de archivo mostrada en la figura 4.4.

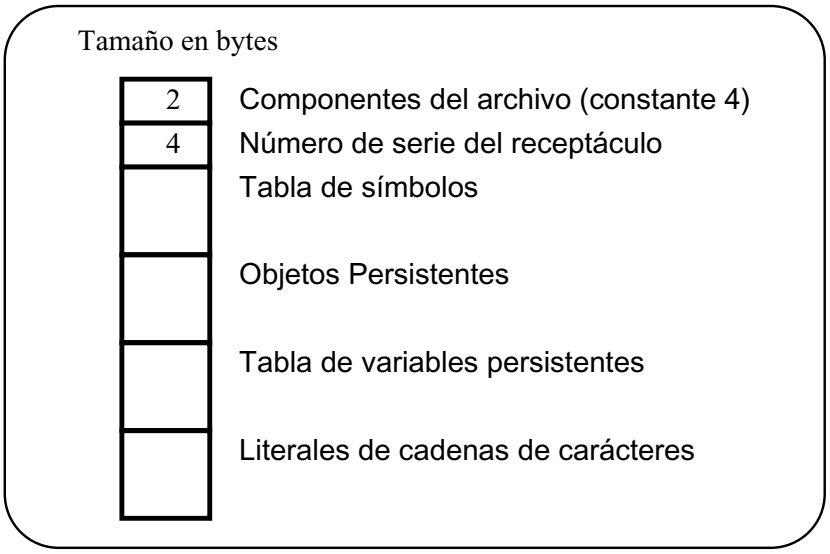


Figura 4.4 Estructura de archivo de datos persistentes.

El campo de número de serie se genera cada vez que el receptáculo es creado o reconstruido; y para ello se utiliza la fecha y hora del sistema. Los demás campos son prácticamente una copia de las estructuras de datos correspondientes tal como se almacenan en memoria.

4.4.4 Archivo de control de ejecución

El archivo de control de ejecución es el resultado de compilar un módulo de aplicación. Es principalmente un archivo de código (parecido a un archivo .COM de DOS) pero con seis bytes de encabezado (ver figura 4.5). Los únicos datos almacenados en este archivo son todas las constantes que aparecen en el archivo fuente, excepto las constantes de cadena de carácter, las cuales se almacenan en el archivo de datos persistentes.

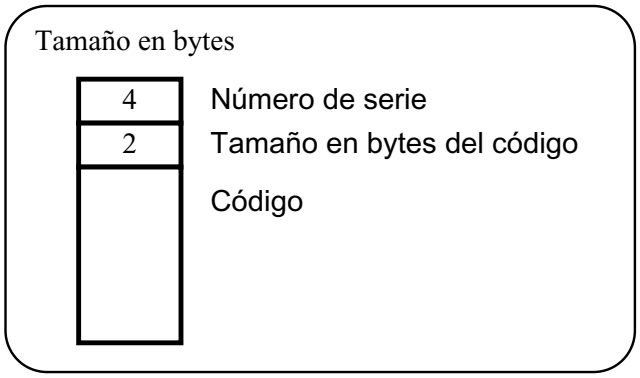


Figura 4.5 Estructura del archivo de control de ejecución.

4.5 Control de ejecución

Para correr un programa en Huntul se requiere que la unidad de tiempo de ejecución (UTE) cargue a memoria el receptáculo y un archivo de control de ejecución, y después de hacer las inicializaciones correspondientes se le debe transferir el control a este último.

La UTE proporciona una serie de funciones de bajo nivel para dar servicio a los programas que así lo requieran. Todas las porciones ejecutables del sistema deben coordinarse entre sí para funcionar adecuadamente, por ello existe una convención para el uso de los registros del procesador. Estos asuntos son tratados a continuación.

4.5.1 Convenciones de uso de registros

Todo el código máquina generado por el compilador de Huntul debe respetar la siguiente convención para el uso de los registros del 8086. Si alguna rutina cambia el valor de los registros que contienen un valor especial, antes de terminar debe restaurar sus valores originales.

- El registro de segmento de datos DS siempre debe estar colocado en el segmento donde se encuentran todos los datos estáticos del sistema. Aquí es donde se halla la tabla de variables compartidas, la región de variables temporales, y la región de literales de cadenas de carácteres.
- El registro de segmento de datos extras ES debe estar indicando el segmento donde se almacenan los objetos (montículo de objetos).

- Los registros de pila SS y SP establecen la memoria dedicada a la pila del sistema.
- Los registros CS e IP corresponden a la porción de código actualmente siendo ejecutada.
- El registro de banderas se utiliza de manera convencional.
- Los registros BP y DI se utilizan como variables temporales.
- El registro SI, junto con DS, apunta siempre a la siguiente localidad de dos bytes disponible en la región de variables temporales.
- El registro BX, junto con DS, apunta a la dirección del receptor del método siendo ejecutado, o al objeto nulo en caso de ser una aplicación. Las direcciones de los argumentos de un método se encuentran en DS:[BX + 0002], DS:[BX + 0004], etc. dependiendo del número de argumentos esperados. Inmediatamente después de los argumentos, y también numerándose incrementalmente de dos en dos, se encuentran la direcciones de los objetos referidos por las variables locales.
- El registro AX sirve para indicar el resultado de los métodos. También se utiliza para indicar el número de servicio en la rutinas de soporte de tiempo de corrida.
- Los registros CX y DX se utilizan como registros de uso temporal y para enviar los valores de los argumentos de la rutinas de servicio.

4.5.2 Rutinas de soporte

El sistema Huntul cuenta con 34 rutinas de bajo nivel destinadas a dar servicio especializado a ciertos métodos de las clases primitivas. Estas rutinas están implementadas como servicios de la interrupción A0h, donde se coloca en el registro AH el número de servicio deseado. Dependiendo del servicio, otros registros pueden usarse para pasar argumentos.

Entre los servicios que brindan las rutinas de soporte cabe mencionar los siguientes: creación de objetos constantes, creación de objetos de clases derivadas, paso de mensajes, interacción con DOS, compilación y evaluación de código a tiempo de corrida, y realización de operaciones aritméticas.

Para ejemplificar el uso de rutinas de soporte, la convención de uso de registros, y la generación de código en general, se presenta a continuación la traducción de la siguiente aplicación :

```
aplicación
    var x
    si x:esNulo()
        x <- 1
    otro
        x <- 2
    fin si
fin aplicación
```

Del código que sigue, la columna de números a la izquierda corresponde al código máquina generado por el compilador, mientras que a la derecha se presenta el mismo código pero en ensamblador del 8086. Los comentarios explican las acciones que se llevan a cabo. Los valores iniciales de los registros DS, ES, BX y SI son los indicados en la sección 4.5.1.

```
; var x
; Crear una nueva variable local x en la región de
; variables temporales en una localidad disponible
; (DS:[SI]), e indicar que su valor inicial es el
; objeto nulo.
33C0h           xor    ax,ax
8904h           mov    [si],ax

; SI apunta a una nueva localidad disponible en la
; región de variables temporales.
46h             inc    si
46h             inc    si

; Invocar mensaje x:esNulo()
; Guardar en la siguiente localidad disponible de
; la región de variables temporales el
; receptor del mensaje.
8B870200h       mov    ax,[bx+0002h]
8904h           mov    [si],ax

; SI apunta a una nueva localidad disponible en la
; región de variables temporales.
46h             inc    si
46h             inc    si
```

```
; AH tiene el número de servicio 7 (paso de mensaje).
B407h           mov    ah,07
; Identificador de método en DX (1Ah = método esNulo).
BA1A00h         mov    dx,001Ah
; Número de argumentos en CX (0).
B90000h         mov    cx,0000
BDFFFFh         mov    bp,0FFFFh
CDA0h           int    A0h

; Probar qué valor booleano quedó en el objeto
; referido por AX.
; CX tiene el apuntador al objeto a probar.
8BC8h           mov    cx,ax
; AH tiene el número de servicio 8 (probar objeto
; booleano).
B408h           mov    ah,08h
CDA0h           int    A0h
; Si fue verdad (CF = 1) saltar a la parte correspondiente.
7203h           jc     @@parte_verdad
E91100h         jmp    @@parte_falso

@@parte_verdad :
; x <- 1
; AH tiene el número de servicio 3 (crear constante
; entera).
B403h           mov    ah,03h
; DX:CX tiene el valor de la constante a crear.
B90100h         mov    cx,0001h
BA0000h         mov    dx,0000h
CDA0h           int    A0h
; Asignar nuevo objeto a x.
89870200h       mov    [bx+0002h],ax
E90E00h         jmp    @@continuar

@@parte_falso :
; x <- 2
; AH tiene el número de servicio 3 (crear constante
; entera).
B403h           mov    ah,03h
; DX:CX tiene el valor de la constante a crear.
B90200h         mov    cx,0002h
BA0000h         mov    dx,0000h
CDA0h           int    A0h
; Asignar nuevo objeto a x.
89870200h       mov    [bx+0002h],ax

@@continuar :
; Terminar aplicación con código de regresa 0.
33C0h           xor    ax,ax
CBh             retf
```

C A P Í T U L O

5

Conclusiones

Este último capítulo compara Huntul con otros lenguajes orientados a objetos y expone las ventajas y desventajas que tiene el lenguaje y su presente implementación. También se menciona brevemente el trabajo futuro que podría realizarse a partir de lo que ya se tiene hasta el momento.

5.1 Huntul frente a otros lenguajes

En la actualidad existen posiblemente más de una centena de lenguajes orientados a objetos, de los cuales menos de una docena son ampliamente conocidos. Es esta sección se realiza una comparación del lenguaje Huntul con cinco de los lenguajes orientados a objetos más difundidos en la actualidad : Smalltalk, C++, Object Pascal, CLOS e Eiffel. En la figura 5.1 se muestra una tabla en donde se destacan los principales elementos que componen a cada uno de los lenguajes en cuestión.

	Smalltalk	C++	Object Pascal	CLOS	Eiffel	Huntul
ABSTRACCIÓN						
Variables de Instancia	Sí	Sí	Sí	Sí	Sí	Sí
Métodos de Instancia	Sí	Sí	Sí	Sí	Sí	Sí
Variables de Clase	Sí	Sí	No	Sí	No	Sí
Métodos de Clase	Sí	Sí	No	Sí	No	Sí
ENCAPSULAMIENTO						
De variables	(3)	(1,2,3)	(2)	(2)	(3)	(3)
De métodos	(2)	(1,2,3)	(2)	(2)	(2,3)	(2)
MODULARIDAD						
Compilación separada	No	Sí	Sí	Sí	Sí	Sí
JERARQUÍA						
Herencia	(S)	(M)	(S)	(M)	(M)	(S)
Clases parametrizadas	No	Sí	No	No	Sí	No
Metaclases	Sí	No	No	Sí	No	Sí
MANEJO DE TIPOS						
Fuertemente tipado	No	Sí	Sí	No	Sí	No
Polimorfismo	Sí	Sí	Sí	Sí	Sí	Sí
CONCURRENCIA						
Multitareas	No	No	No	Si	No	No
PERSISTENCIA						
Objetos persistentes	No	No	No	No	No	Si

Figura 5.1 Tabla comparativa de 5 de los principales lenguajes de programación orientada a objetos junto con el lenguaje Huntul. Los elementos utilizados aquí para realizar la comparación son los que Grady Booch considera en su modelo de objetos. Las claves usadas para el encapsulamiento son : (1) Privado, (2) Público, y (3) Protegido. Las claves de la herencia son : (S) Simple, y (M) Múltiple.

Como se puede observar, no existe un lenguaje que incluya desde su diseño todos los elementos del modelo de objetos propuesto por Booch[1].

---

[1] Grady Booch, *Object-Oriented Analysis and Design With Applications* 2nd Edition. (Benjamin Cummings. Redwood City, California. 1994) pp. 474-489.

Respecto al manejo de abstracción, Huntul es tan bueno como Smalltalk, C++ y CLOS. Eiffel y Object Pascal no cuentan con los componentes de clase : variables y métodos de clase.

En cuanto al encapsulamiento, Smalltalk y Huntul consideran que los métodos deben ser públicos y los datos protegidos. En este aspecto, el lenguaje más flexible es C++, pues aquí tanto datos como código pueden ser públicos, privados o incluso protegidos; estas distinciones permiten controlar la visibilidad de los componentes de una clase hasta el nivel de subclases. Eiffel queda en un término medio, pues requiere que los datos sean privados, pero los métodos pueden ser protegidos o públicos. Object Pascal y CLOS son los más flojos es este aspecto, pues tanto datos como métodos son públicos, por lo que el encapsulamiento aquí es una disciplina impuesta al programador y no algo que el lenguaje promueva.

En el aspecto de modularidad, todos los lenguajes a excepción de Smalltalk soportan la construcción de programas a través de módulos que pueden ser compilados por separado.

Todos los lenguajes aquí presentados requieren del manejo de herencia para que sean considerados en primer lugar como lenguajes orientados a objetos. Smalltalk, Object Pascal y Huntul solamente permiten herencia simple. La herencia múltiple, que es más flexible pero compleja, es una característica de los lenguajes de peso completo : C++, CLOS e Eiffel. El manejo de *tipos parametrizados* permite que una clase sirva como molde genérico para otras clases; C++ e Eiffel son los únicos que incorporan esta facilidad.

Los lenguajes que consideran a las clases como objetos requieren del concepto de metaclase. Smalltalk, CLOS y Huntul están en esta situación. Los lenguajes que ven a las clases como una especie de extensión a los tipos que se encuentran en los lenguajes convencionales no requieren de metaclases. Esta es una característica común, pero no necesariamente única, de los lenguajes híbridos. Bajo este esquema funcionan C++, Object Pascal e Eiffel.

Con respecto al manejo de tipos, y tal como ya se mencionó en la sección 2.1.6, Smalltalk, CLOS y Huntul manejan tipos latentes, es decir, no son fuertemente tipados. Esto implica que muchos errores que podrían ser detectados a tiempo de compilación son detectados hasta tiempo de corrida. C++, Object Pascal e Eiffel sí son fuertemente tipados, y por ello son preferidos para construcción de proyectos grandes.

Al igual que en la cuestión de la herencia, el aspecto de polimorfismo es esencial en cualquier lenguaje orientado a objetos, por lo que los seis lenguajes lo incluyen.

CLOS es el único lenguaje que soporta concurrencia, es decir, que más de una hebra de control se ejecute simultáneamente.

Por último, Huntul es el único de los seis lenguajes que soporta de manera directa objetos persistentes.

Resumiendo, el lenguaje Huntul tienen ventajas y desventajas cuando se compara a nivel de funcionalidad con otros lenguajes. Sin embargo, para la audiencia a la cual está dirigido el lenguaje, éste tiene un aliciente fundamental sobre los demás lenguajes : es pequeño pero sin dejar de estar completo. La presente implementación corre sobre cualquier modesta PC compatible con 512 Kb de memoria RAM, un monitor monocromático y un disco flexible, lo que lo hace bastante atractivo para escuelas y/o estudiantes de modestos recursos.

5.2 Limitaciones de Huntul

Tal como se mostró en el capítulo 2, Huntul es un lenguaje que cuenta con todas las propiedades para programar utilizando el paradigma de orientación a objetos. Adicionalmente, Huntul ofrece objetos persistentes, que es una característica que no tiene ninguno de los lenguajes populares orientados a objetos.

En la sección anterior se presentó una comparación del lenguaje Huntul con otros cinco lenguajes orientados a objetos. La referencia utilizada fue el modelo de objetos. Desde esta perspectiva, Huntul probó ser un digno competidor por contar con muchas de las cualidades de los otros lenguajes que son ampliamente reconocidos.

Ahora bien, el lenguaje Huntul tiene varias limitaciones que es importante tener en cuenta. Para empezar, las clases primitivas están diseñadas para ser utilizadas únicamente a nivel de *composición* por otras clases y no a nivel de *herencia*. Esto es, una clase hipotética *Pila* puede tener una variable de instancia que sea un arreglo, pero dicha clase no puede heredar de la clase Arreglo ni de ninguna otra clase primitiva. Esta restricción se hizo por simplificar la jerarquía de clases, pero puede resultar limitante pues prohibe la especialización de clases primitivas.

Otra limitación del lenguaje Huntul es el hecho de que no tiene métodos protegidos. Es frecuente que la implementación de una clase cuente con métodos que solamente deben ser de acceso protegido. Sin embargo, puesto que todos los métodos son públicos, el

usuario de una clase puede hacer uso de porciones de código que no debería, produciendo resultados imprevistos y/o indeseados.

A nivel de implementación, el sistema Huntul tiene algunas desventajas bastante serias. El manejo de persistencia se hace mediante un sistema de archivos convencional. No existe ningún nivel de protección, seguridad o recuperación de fallas como los que se pueden encontrar en los sistemas manejadores de bases de datos.

Otro aspecto negativo es la generación del código máquina hecha por el compilador, la cual no está optimizada. Y agregando a lo anterior, está la forma en que se implementa el paso de mensajes (por una serie de búsquedas secuenciales), provocando que el sistema resulte excesivamente lento en algunas aplicaciones.

El sistema no cuenta a la fecha con un ambiente de desarrollo adecuado. Hace falta seriamente un depurador (*debugger*) para que el lenguaje resulte más atractivo y sencillo de usar. El diseño del lenguaje está hecho para no requerir hardware costoso, pero si éste está presente no se le saca provecho alguno.

Por último, es importante mencionar que algunas de las decisiones en cuanto al diseño del lenguaje no se han aún demostrado que sean válidas. Específicamente, se ha dicho que Huntul es más fácil de aprender que otros lenguajes con objetos por la similitud que tienen las estructuras de control con las de lenguajes como C o Pascal, pero el autor no ha demostrado esto en la práctica. Es claro que hace falta realizar todavía una gran cantidad de investigación para corroborar si el diseño del lenguaje cumple con los objetivos originalmente planteados.

5.3 Trabajo futuro

Dadas la limitaciones expuestas en la sección anterior, el trabajo futuro que se haga sobre el lenguaje Huntul debe estar encaminado principalmente hacia corregir sus fallas.

Primero, es necesario validar el diseño del lenguaje exponiéndolo al auditorio al cual está dirigido : alumnos que ya sepan programar en algún lenguaje imperativo convencional y que se desee enseñarles el paradigma de objetos. Para este efecto se necesitarían al menos dos grupos, uno de los cuales se le enseñaría Huntul y al otro (el grupo de control) un lenguaje como C++ o Smalltalk. En ambos grupos el contenido teórico sería el mismo, la única diferencia debe ser el lenguaje utilizado para la exposición práctica y el desarrollo de proyectos. En ambos casos se deben medir los conocimientos obtenidos y

el tiempo tardado. A partir de esto se puede hacer una evaluación más objetiva del lenguaje para ver si cumple con su propósito.

Otras mejoras posibles están en el área de la implementación : construir un depurador, un editor inteligente, agregar ayuda en línea, generar código optimizado, portar el sistema a otras plataformas y/o sistemas operativos, mejorar el manejo de persistencia, agregar objetos concurrentes, etc. Un producto de software siempre tendrá oportunidad de ser mejorado.

B I B L I O G R A F Í A

Abelson, Harold and Sussman, Gerald. *Structure and Interpretation of Computer Programs.* The MIT Press & McGraw-Hill. Cambridge, Massachusetts. 1985.

Aho, Alfred et al. *Compilers Principles, Techniques and Tools.* Addison-Wesley. Reading, Massachusetts. 1986.

Bartels, Dirk and Robie, Jonathan. "Persistent Objetcts and Object Oriented Databases for C++", *C++ Report..* Volume 4, No. 7; September 1992. 49-56.

Booch, Grady. *Object-Oriented Analysis and Design With Applications.* 2nd Edition. Benjamin/Cummings. Redwood City, California. 1994.

Borland International. *Borland C++ Version 3.1 Programmer's Guide.* Borland International Inc. Scotts Valley, California. 1992.

Borland International. *Borland C++ Version 3.1 Tools and Utilities Guide.* Borland International Inc. Scotts Valley, California. 1992.

Borland International. *Borland C++ Version 3.1 User's Guide.* Borland International Inc. Scotts Valley, California. 1992.

Borland International. *Turbo Assembler Version 3.0 Quick Reference Guide.* Borland International Inc. Scotts Valley, California. 1992.

Borland International. *Turbo Assembler Version 3.0 User's Guide.* Borland International Inc. Scotts Valley, California. 1992.

Budd, Timothy. *A Little Smalltalk.* Addison-Wesley. Reading, Massachusetts. 1987.

Budd, Timothy. *An Introduction to Object-Oriented Programming.* Addison-Wesley. Reading, Massachusetts. 1991.

Clinger, William and Rees, Jonathan eds. *The Revised^4 Report on the Algorithmic Language Scheme.* Lisp Pointers IV, 3, 1991.

Digitalk. *Smalltalk/V Windows Tutorial and Programming Handbook.* Digitalk, Inc. Los Angeles, California. 1991.

Ezzel, Ben. *Object-Oriented Programming in Pascal.* Addison-Wesley. Reading, Massachusetts. 1989.

Ferguson, Iain et al. *The Schemer's Guide.* Schemers Inc. Fort Lauderdale, Florida. 1990.

Fleming, Jim. "The C+@ Programming Language", *Dr. Dobb's Journal.* No. 206; October, 1993. 24-32.

Floyd, Michael. "Comparing Object-Oriented Languages", *Dr. Dobb's Journal.* No. 206; October, 1993. 104-118.

Friedman, Daniel et al. *Essentials of Programming Languages.* The MIT Press & McGraw-Hill. Cambridge, Massachusetts. 1992.

Gabriel, Richard. "Persistence in a Programming Environment", *Dr. Dobb's Journal.* No. 195; December, 1992. 46-55.

Gupta, Rajiv and Horowitz, Ellis eds. *Object-Oriented Databases with Applications to CASE, Networks and VLSI CAD.* Prentice-Hall. Englewood Cliffs, New Jersey. 1991.

Hartnell, Tim. *Exploring Artificial Intelligence on your Microcomputer.* Interface Publications. Bristol, UK. 1984.

Howard, Robert. "The Eiffel Programming Language", *Dr. Dobb's Journal.* No. 206; October, 1993. 68-73.

Kernighan, Brian and Ritchi, Dennis. *The C Programming Language*. 2nd Edition. Prentice-Hall. Englewood Cliffs, N.J. 1988.

Kurtz, Thomas and Kemmeny, John. *Structured BASIC Programming.* Willey. New York, New York. 1987.

Lee, Peter ed. *Topics in Advanced Language Implementations.* The MIT Press. Cambridge, Massachusetts. 1990.

Maguire, Stephen . *Writing Solid Code.* Microsoft Press. Redmond, Washington. 1993.

Mak, Ronald. *Writing Compilers and Interpreters An Applied Approach.* Willey. New York, New York. 1991.

McConnel, Steven *Code Complete.* Microsoft Press. Redmond, Washington. 1993.

Nelson, Ross. *Microsoft's 80386/80486 Programming Guide.* Microsoft Press. Redmond, Washington. 1991.

Norton, Peter et al. *The Peter Norton PC Programmer's Bible.* Microsoft Press. Redmond, Washington. 1993.

Sanders, Donald. *Computers Today.* McGraw-Hill. New York, New York. 1983.

Schildt, Herbert. *Advanved C.* McGraw-Hill. New York, New York. 1986.

Sebesta, Robert. *Concepts of Programming Languages.* 2nd Edition. Benjamin/Cummings. Redwood City, California. 1993.

Sethi, Ravi. *Programming Languages Concepts and Constructs.* Addison-Wesley. Reading, Massachusetss. 1990.

Stevens, Al. "Persistent Objects in C++"., *Dr. Dobb's Journal.* No. 195; December, 1992. 34-44.

Stroustrup, Bjarne. *The C++ Programming Language.* 2nd Edition. Addison-Wesley, Reading. Massachusetss. 1990.

Trembley, Jean-Paul and Soreson, Paul. *An Introduction to Data Structures with Applications.* McGraw-Hill. Singapore. 1984.

Trembley, Jean-Paul and Soreson, Paul. *The Theory and Practice of Compiler Writing*. McGraw-Hill. Singapore. 1989.

Winston, Patrick and Horn, Paul. *Lisp* 3rd Edition. Addison-Wesley. Reading, Massachusetss. 1989.

Wirth, Niklaus. *Programming in Modula-2.* 2nd Edition. Springer-Verlag. New York, New York. 1985.

A P É N D I C E

A

Guía del Usuario Huntul Ver. 1.00

El presente apéndice tiene como propósito describir la operación a nivel de usuario del sistema Huntul versión 1.00. Para sacar el máximo provecho de este sistema es necesario leer primero los capítulos 2 y 3 de esta tesis. En el apéndice B se encuentran varios ejemplos de programas en donde se pueden contemplar algunas de las principales virtudes del lenguaje Huntul.

El archivo LEAME.TXT contenido en el disco donde se distribuye el software contiene información adicional a la que se encuentra en este apéndice.

A.1 Requisitos de hardware y software

Para correr el sistema Huntul versión 1.00 se requiere de la siguiente configuración de hardware y software : Computadora personal PC, XT, AT, PS o compatible con procesador 8088/86 o superior con al menos 420 Kb de memoria RAM convencional disponible, un disco flexible de 360 Kb o mejor, sistema operativo DOS versión 3.3 o superior, y un editor de textos convencional. Un disco duro con unos 300 Kb disponibles es deseable pero no indispensable.

A.2 Instalación

Si se desea trabajar sobre disco duro, es recomendable crear un nuevo subdirectorio. Desde el prompt de DOS teclear cada una de las siguientes líneas (al final de cada línea es necesario presionar la tecla <Enter>) :

```
c:
cd \
md huntul
cd huntul
```

Después se requiere copiar el archivo INSTALA.EXE (contenido también en el disco de distribución) al disco duro de la siguiente manera (suponiendo que la unidad donde se encuentra el disco es la unidad A:) :

```
copy a:\instala.exe
```

Finalmente, para instalar el sistema Huntul teclear :

```
instala -d
```

Una vez que termine de ejecutarse el comando anterior, todos los archivos que se requieren para el sistema estarán presentes en el subdirectorio C:\HUNTUL.

Si no se cuenta con disco duro será necesario instalar el sistema en un disco flexible. Primero se debe copiar el archivo INSTALA.EXE del disco de distribución a otro disco que tenga al menos 300 Kb de espacio libre, utilizando el siguiente comando (se asume que el disco de distribución se encuentra en la unidad B: y el disco destino en la unidad A:) :

```
copy b:\instala.exe a:\
```

Posteriormente, teclear lo siguiente :

```
a:
cd \
instala -d
```

Una vez que termina el último comando el sistema queda instalado.

A.3 Opciones del sistema

La presente implementación del sistema Huntul sólo puede ser utilizada a nivel de línea de comando. El nombre del archivo ejecutable se llama HUNTUL.EXE. Si se invoca sin argumentos, de la siguiente forma :

```
huntul
```

se obtendrá el siguiente mensaje :

```
Sistema Huntul  Versión 1.00  ITESM CEM; A. Ortiz, 1994.

Argumentos incorrectos.
Para obtener ayuda, teclear : HUNTUL /?
```

A continuación se presentan las distintas opciones con las que cuenta el sistema. Es importante mencionar que todas las opciones deben ir forzosamente en minúscula.

A.3.1 Opción /?

La opción /? sirve para obtener ayuda del sistema. Cuando se teclea :

```
huntul /?
```

el siguiente mensaje aparece en pantalla :

```
Sistema Huntul  Versión 1.00  ITESM CEM; A. Ortiz, 1994.

Sintaxis  :  HUNTUL <opción>
Donde opción es uno de los siguientes :
     /?                     Esta información.
     /r                     Reconstruir receptáculo.
     /c <archivo-fuente>    Compilar archivo fuente.
     /e <archivo-control>   Ejecutar archivo de control.
```

A.3.2 Opción /r

La opción /r comunica al sistema que se debe reconstruir (o construir por primera vez) el receptáculo. Esto conduce a que los archivos que componen el receptáculo sean borrados, y que se generen unos nuevos. Todas las clases primitivas son recompiladas conforme al orden en que se encuentren en el archivo de configuración (ver sección A.4). Como es de suponerse, todos los objetos persistentes no conformados por las clases primitivas ya no estarán en el receptáculo, por lo que esta opción se debe utilizar con el debido cuidado.

A.3.3 Opción /c

Para compilar un archivo fuente de Huntul, ya sea un módulo de definición de clases o un módulo de aplicación, se utiliza la opción /c. Por ejemplo, para compilar un archivo llamado PRUEBA.HUN, se debe teclear lo siguiente :

```
huntul /c prueba
```

Si no se indica extensión, por defecto se asume ".HUN". Si el archivo fuente no contiene errores, en la pantalla se debe ver el siguiente mensaje :

```
Sistema Huntul  Versión 1.00  ITESM CEM; A. Ortiz, 1994.

Compilando           : PRUEBA.HUN
Número de errores    : 0
```

En caso de contener errores, se indica el motivo y el número de línea del archivo donde fue detectado. Solamente cuando la compilación es exitosa ocurre la actualización del receptáculo. Adicionalmente, si el archivo fuente resulta ser un módulo de aplicación, y éste es compilado sin errores, entonces se produce el *archivo de control de ejecución*, que tiene el mismo nombre que el archivo fuente pero con extensión ".ACE".

Cualquier módulo puede ser compilado en cualquier momento y en cualquier orden, la única restricción que existe es que una subclase no puede ser compilada antes de que su superclase directa haya sido compilada por lo menos la primera vez. Al recompilar una superclase que ya cuenta con una o más subclases en el receptáculo, el sistema evalúa si es necesario recompilar las subclases para garantizar la integridad, en cuyo caso procede a hacerlo; esto nada más ocurre cuando es realmente necesario.

A.3.4 Opción /e

Una vez generado un archivo de control de ejecución por el compilador, se puede proceder a ejecutarlo utilizando la opción /e. Por ejemplo, si el archivo PRUEBA.HUN es un módulo de aplicación que ya fue compilado, obteniendo como resultado al archivo PRUEBA.ACE, la manera de correrlo es tecleando lo siguiente :

```
huntul /e prueba
```

Se asume la extensión ".ACE" por omisión. Lo que sigue de aquí en adelante es responsabilidad de la aplicación. En caso de una terminación normal, el receptáculo es actualizado con los cambios que hayan ocurrido durante el transcurso de la ejecución, y se regresa un código de error a DOS que corresponde al valor numérico de la expresión de la

sentencia *regresa* que finalizó la aplicación (si no hay tal sentencia, el valor regresado es cero).

No se permite ejecutar un ACE que sea más antiguo que el receptáculo correspondiente. Cuando un receptáculo es reconstruido, se le asigna un número de serie único. Cuando un módulo de aplicación es compilado, al ACE generado se le asigna el mismo número de serie que el de su receptáculo. Cuando se intenta ejecutar el ACE, su número de serie debe ser el mismo que el del receptáculo, de otra forma se impide que continúe la ejecución.

A.4 El archivo de configuración

Para que el sistema Huntul funcione adecuadamente en todas sus modalidades, debe contar siempre con un archivo de configuración. Dicho archivo debe estar presente siempre en el directorio actual, y debe llamarse HUNTUL.CFG; en caso de no ser así se produce un error.

El archivo de configuración es un archivo de texto que puede ser creado y modificado en cualquier editor de texto común. El archivo de configuración que se provee por defecto es el siguiente :

```
Alfa
Primitiv\Generico
Primitiv\Nulo
Primitiv\Booleano
Primitiv\Caracter
Primitiv\Entero
Primitiv\Cadena
Primitiv\Arreglo
Primitiv\Codigo
Primitiv\SysOp
```

La primera línea del archivo es el nombre del receptáculo, sin extensión. Se puede incluir una ruta de acceso completa, incluyendo la unidad de disco, de lo contrario se toman las opciones por omisión provistas por DOS. Las líneas restantes son los nombres de los archivos fuente de los módulos de definición de las clases consideradas como primitivas (puede haber clases adicionales a las mencionadas en la sección 2.4.5). Igualmente, estos nombres de archivo pueden incluir una ruta completa y la unidad de disco que se desee.

La opción /r del sistema utiliza toda la información de este archivo para reconstruir el receptáculo y compilar en orden cada uno de los archivos fuente indicados ahí. Las

opciones /c y /e solamente requieren de la primer línea para conocer con que receptáculo se habrá de trabajar.

A.5 Sugerencias adicionales

Si se va a trabajar sobre un disco flexible, la instalación recomendada en la sección A.2 debe bastar para que el sistema trabaje de manera adecuada. Por otro lado, si se desea realizar un proyecto en Huntul sobre el disco duro, existen algunas sugerencias adicionales para el mejor funcionamiento del sistema.

Primero, se debe modificar el archivo AUTOEXEC.BAT. Este archivo debe tener en algún sitio una línea parecida a la siguiente :

```
PATH C:\DOS;C:\WINDOWS
```

Agregar al final de esa línea la ruta de acceso donde quedó instalado Huntul, de tal forma que quede similar a lo siguiente :

```
PATH C:\DOS;C:\WINDOWS;C:\HUNTUL
```

Para que haga efecto este cambio, se requiere reinicializar la computadora. Hecho lo anterior, el sistema Huntul puede ser invocado ahora desde cualquier subdirectorio.

Es práctico tener un subdirectorio específico para cada proyecto, sobre todo si es algo extenso y requiere de más de un puñado de archivos fuente. Para crear un nuevo subdirectorio PROY1 debajo del directorio HUNTUL, se debe teclear desde el prompt de DOS los siguientes comandos :

```
c:
cd \huntul
md proy1
cd proy1
```

Posteriormente, en el subdirectorio actual (C:\HUNTUL\PROY1) se debe crear un nuevo archivo de configuración HUNTUL.CFG con la siguiente información :

```
Proy1
C:\Huntul\Primitiv\Generico
C:\Huntul\Primitiv\Nulo
C:\Huntul\Primitiv\Booleano
C:\Huntul\Primitiv\Caracter
C:\Huntul\Primitiv\Entero
C:\Huntul\Primitiv\Cadena
C:\Huntul\Primitiv\Arreglo
C:\Huntul\Primitiv\Codigo
C:\Huntul\Primitiv\SysOp
```

Habiendo hecho todo lo anterior, se puede entonces generar un nuevo receptáculo PROY1 en el subdirectorio C:\HUNTUL\PROY1 con tan solo teclear :

```
huntul /r
```

En lo sucesivo, cada vez que se ejecute el sistema Huntul sobre este subdirectorio, tanto para compilar como para ejecutar programas, se utilizará el mismo receptáculo. En este mismo subdirectorio se recomienda también que se coloquen todos los archivos fuente del proyecto.

A.6 Archivos del receptáculo

Aunque conceptualmente el receptáculo es un solo archivo lógico, esta versión de Huntul utiliza tres archivos físicos para almacenar toda la información necesaria. A continuación se describe de modo general el contenido de cada uno de estos archivos. Lo más importante a tener en cuenta es que estos archivos no deben ser borrados ni alterados mas que por el sistema Huntul, de lo contrario pueden ocurrir serias fallas. Los archivos que componen al receptáculo son :

- *Archivo de datos persistentes* (extensión ".PRS"). Incluye a todas las variables y objetos persistentes, los nombres de todos los símbolos usados por el sistema (tabla de símbolos), y las literales de cadenas de carácter.
- *Archivo de clases* (extensión ".CLS"). Contiene la información individual de cada una de las clases : nombre del archivo de definición de clase, antecesor directo, y variables de clase e instancia.

- *Archivo de métodos* (extensión ".MTD"). Aquí se almacena todo el código de los métodos, junto con la información de control sobre las clases a las que pertenecen, número de argumentos, etc.

A P É N D I C E

B

Programas ejemplo

En este apéndice se muestran tres ejemplos de programas escritos en lenguaje Huntul. Se procuró que cada uno de ellos fuera interesante y que utilizara las propiedades principales de la programación orientada a objetos. Se incluyó persistencia en donde resultó útil.

Para cada ejemplo se expone brevemente lo que el programa hace, seguido de los listados fuente. Cada uno de los archivos que componen los ejemplos, así como la información para compilarlos y ejecutarlos, se encuentran en el disco de distribución.

B.1 El problema de las ocho reinas

En el juego de ajedrez, la reina puede atacar a cualquier otra pieza que se encuentre en el mismo renglón, la misma columna, o en la diagonal. El problema de las ocho reinas consiste en colocar precisamente ocho reinas en el tablero sin que ninguna pueda atacar a otra. Una solución es la que se ilustra en la figura B.1. Dicha solución es la que calcula el programa que se presenta a continuación, y cabe mencionar que no es una solución única.

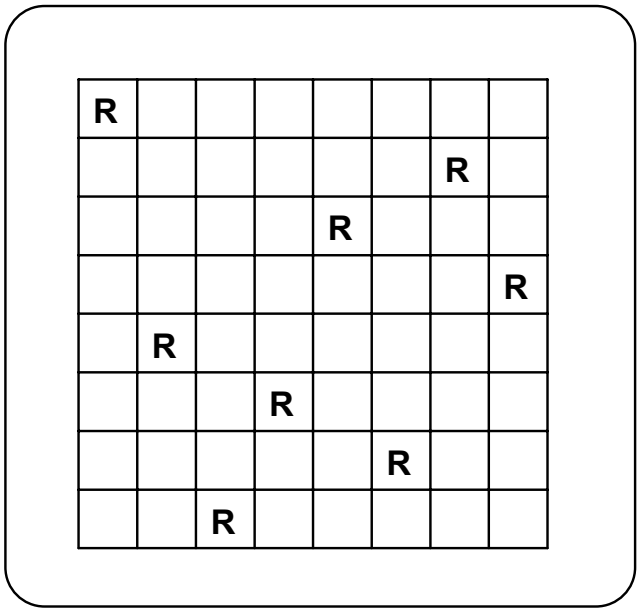


Figura B.1 Una solución al problema de las ocho reinas.

La esencia del programa es de que cada reina sea un objeto, y de que todas cuenten con el poder necesario para que ellas mismas encuentren la solución. La idea es realizar una simulación en donde se establezcan las condiciones iniciales del universo para posteriormente echarlo a andar. La solución es encontrada una vez que el universo se estabiliza.

Budd describe detalladamente el cómo se llega al programa que encuentra la solución[1]. Este es un ejemplo de la corrida del programa :

```
Acertijo de las Ocho Reinas en el Tablero de Ajedrez . Ver. 1.00
ITESM CEM; A. Ortiz, 1994.

Resuelto
renglón : 1  columna : 1
renglón : 5  columna : 2
renglón : 8  columna : 3
renglón : 6  columna : 4
renglón : 3  columna : 5
renglón : 7  columna : 6
renglón : 2  columna : 7
renglón : 4  columna : 8
```

---

[1] Timothy Budd, *An Introduction to Object-Oriented Programming.* (Addison-Wesley. Reading, Massachusetts. 1991) pp. 75-80.

El programa está conformado por dos clases. La primera es la clase *Reina*, cuyas instancias representan las fichas que se desean colocar sobre el tablero. La segunda clase es *Tablero*, que se encarga de iniciar la resolución del problema y posteriormente imprimir el resultado. Estos son los listados del programa :

```
{-----------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Módulo de aplicación para resolver el problema de las ocho
    reinas. Basado en :

        Budd, Timothy. "An Introduction to Object-Oriented
        Programming". Addison-Wesley. Reading, Massachussets. 1991.
        pp. 75-80.

    Requiere de RE_REINA.HUN y RE_TABL.HUN.
-----------------------------------------------------------------------}

aplicación

    var unTablero

    unTablero <- Tablero:nuevo()
    unTablero:resuelve()

fin aplicación

;-------- Fin de Archivo <RE_APLIC.HUN> ----------------------------



{-----------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Módulo de definición para la clase Reina, que sirve para
    resolver el problema de las ocho reinas.

    Requiere de RE_APLIC.HUN y RE_TABL.HUN.
-----------------------------------------------------------------------}
clase Reina hereda Genérico
```

```
{============================} defclase {============================}

    ;---------------------------------------------------------------
    ;   Crea una instancia de la clase Reina.
    ;---------------------------------------------------------------
    método crea(unaColumna ! Entero, unaVecina ? Genérico)
        regresa (antecesor:nuevo()):inicializa(unaColumna, unaVecina)
    fin método

{=========================} definstancia {=========================}

    var renglón
    var columna
    var vecina

    ;---------------------------------------------------------------
    ;   Inicializa una instancia de la clase Reina con unaColumna y
    ;   unaVecina.
    ;---------------------------------------------------------------
    método inicializa(unaColumna ! Entero, unaVecina ? Genérico)
        si ((unaVecina:nombreClase() = "Reina")| unaVecina:esNulo()):no()
            unaVecina:error( \
                "Se espera nulo o instancia de la clase Reina")
        fin si
        columna <- unaColumna
        vecina  <- unaVecina
        regresa receptor
    fin método

    ;---------------------------------------------------------------
    ;   Regresa verdad si una reina o vecinas pueden atacar la
    ;   posición (r, c), de otra forma regresa falso.
    ;---------------------------------------------------------------
    método puedeAtacar(r ! Entero, c ! Entero)

        var dif

        si renglón = r
            regresa verdad
        otro
            dif <- c - columna
            si (renglón + dif = r) | (renglón - dif = r)
                regresa verdad
            otrosi vecina:esNulo()
                regresa falso
            otro
                regresa vecina:puedeAtacar(r, c)
            fin si
        fin si

    fin método
```

```
;-----------------------------------------------------------------
;   Genera la primera posible solución.
;-----------------------------------------------------------------
método primeraSolución()
    renglón <- 1
    si vecina:esNulo()
        regresa verdad
    otrosi (vecina:primeraSolución()):no()
        regresa falso
    otro
        regresa receptor:pruebaAvanza()
    fin si
fin método

;-----------------------------------------------------------------
;   Prueba y posiblemente avanza solución.
;-----------------------------------------------------------------
método pruebaAvanza()
    si vecina:esNulo()
        regresa verdad
    otrosi vecina:puedeAtacar(renglón, columna)
        regresa receptor:siguienteSolución()
    otro
        regresa verdad
    fin si
fin método

;-----------------------------------------------------------------
;   Genera la siguiente solución aceptable.
;-----------------------------------------------------------------
método siguienteSolución()
    si renglón = 8
        si vecina:esNulo()
            regresa falso
        otrosi (vecina:siguienteSolución()):no()
            regresa falso
        otro
            renglón <- 1
            regresa receptor:pruebaAvanza()
        fin si
    otro
        renglón <- renglón + 1
        regresa receptor:pruebaAvanza()
    fin si
fin método
```

```
    ;-------------------------------------------------------------
    ;   Despliega la solución.
    ;-------------------------------------------------------------
    método despliega()
        si (vecina:esNulo()):no()
            vecina:despliega()
        fin si
        "renglón : ":imprime()
        renglón:imprime()
        "  columna : ":imprime()
        columna:imprimeNL()
    fin método

fin clase

;-------- Fin de Archivo <RE_REINA.HUN> ----------------------------



{-------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Módulo de definición para la clase Tablero, que sirve para
    resolver el problema de las ocho reinas.

    Requiere de RE_APLIC.HUN y RE_REINA.HUN.
-------------------------------------------------------------------}
clase Tablero hereda Genérico

{============================} defclase {============================}

    ;-------------------------------------------------------------
    ;   Crea una instancia de la clase Tablero.
    ;-------------------------------------------------------------
    método nuevo()
        regresa (antecesor:nuevo()):inicializa()
    fin método

{=========================} definstancia {=========================}

    var unaReina

    ;-------------------------------------------------------------
    ;   Inicializa una instancia de la clase Tablero.
    ;-------------------------------------------------------------
    método inicializa()

        var vecina, índice

        vecina <- nulo
        índice <- 1
        ciclo
```

```
            hasta índice > 8
            unaReina <- Reina:crea(índice, vecina)
            vecina <- unaReina
            índice <- índice + 1
        fin ciclo
        regresa receptor
    fin método

    ;-------------------------------------------------------------
    ;   Resuelve el problema de las ocho reinas.
    ;-------------------------------------------------------------
    método resuelve()
        "Acertijo de las Ocho Reinas en el Tablero de Ajedrez":imprime()
        " . Ver. 1.00":imprimeNL()
        "ITESM CEM; A. Ortiz, 1994.":imprimeNL()
        "":imprimeNL()
        "Resolviendo...":imprime()
        si unaReina:primeraSolución()
            ("" | @13 + "Resuelto      "):imprimeNL()
            unaReina:despliega()
        fin si
    fin método

fin clase

;-------- Fin de Archivo <RE_TABL.HUN> -----------------------------
```

B.2 Juego de adivinanzas de animales

Este programa es bastante conocido. Consiste en pedir al usuario que piense en un animal, y luego tratar de adivinar de qué animal se trata haciendo preguntas a las que se debe contestar simplemente con sí o no. Aquí se muestra un ejemplo (la entrada del usuario está en letras negritas) :

```
Juego de adivinanzas de Animales. Ver. 1.00
ITESM CEM; A. Ortiz, 1994.

PIENSA EN UN ANIMAL.
YO TRATARE DE ADIVINAR CUAL ES HACIENDOTE VARIAS PREGUNTAS.

¿ES UN ANIMAL DOMESTICO? (S/N) : S
¿ES UN (UNA) PERRO? (S/N) : N
* ESCRIBE EL NOMBRE DEL ANIMAL QUE PENSASTE : TORO
* ESCRIBE UNA AFIRMACION QUE SEA VERDAD PARA UN (UNA) TORO
  PERO QUE SE FALSA PARA UN (UNA) PERRO : TIENE CUERNOS

¿DESEAS JUGAR OTRA VEZ? (S/N) : S
```

```
¿ES UN ANIMAL DOMESTICO? (S/N) : S
¿TIENE CUERNOS? (S/N) : S
¿ES UN (UNA) TORO? (S/N) : N
* ESCRIBE EL NOMBRE DEL ANIMAL QUE PENSASTE : VACA
* ESCRIBE UNA AFIRMACION QUE SEA VERDAD PARA UN (UNA) VACA
  PERO QUE SE FALSA PARA UN (UNA) TORO : DA LECHE

¿DESEAS JUGAR OTRA VEZ? (S/N) : S

¿ES UN ANIMAL DOMESTICO? (S/N) : N
¿ES UN (UNA) COCODRILO? (S/N) : S
¡¡¡ ADIVINE !!!

¿DESEAS JUGAR OTRA VEZ? (S/N) : N
```

Como se puede observar, el programa aprende de sus errores, de tal forma que cada vez que no acierta se incrementa su conocimiento. El programa está implementado utilizando un árbol binario que sirve para tomar decisiones. Cada nodo interno del árbol es una pregunta y cada hoja es el nombre de un animal. A partir de las respuestas del usuario, el árbol se recorre hacia la derecha o izquierda, dependiendo si se contestó de manera negativa o positiva respectivamente. Al llegar a una hoja, se pregunta si el animal contenido ahí fue el que se pensó. En caso de no serlo, se hacen dos preguntas que causan que el árbol crezca, agregando una nueva hoja y un nuevo nodo intermedio. En la figura B-2 se muestra el árbol al inicio y al final de la corrida anterior.

El programa consiste de dos clases : una clase para manejar árboles binarios en general, y la otra para manejar árboles de juego; esta última hereda de la primera. La aplicación declara una variable persistente que contiene una instancia de la clase de árbol de juego. Esto permite que el conocimiento de una corrida quede almacenado para futuras corridas. Los tres listados fuente de este programa comienzan en la siguiente página.

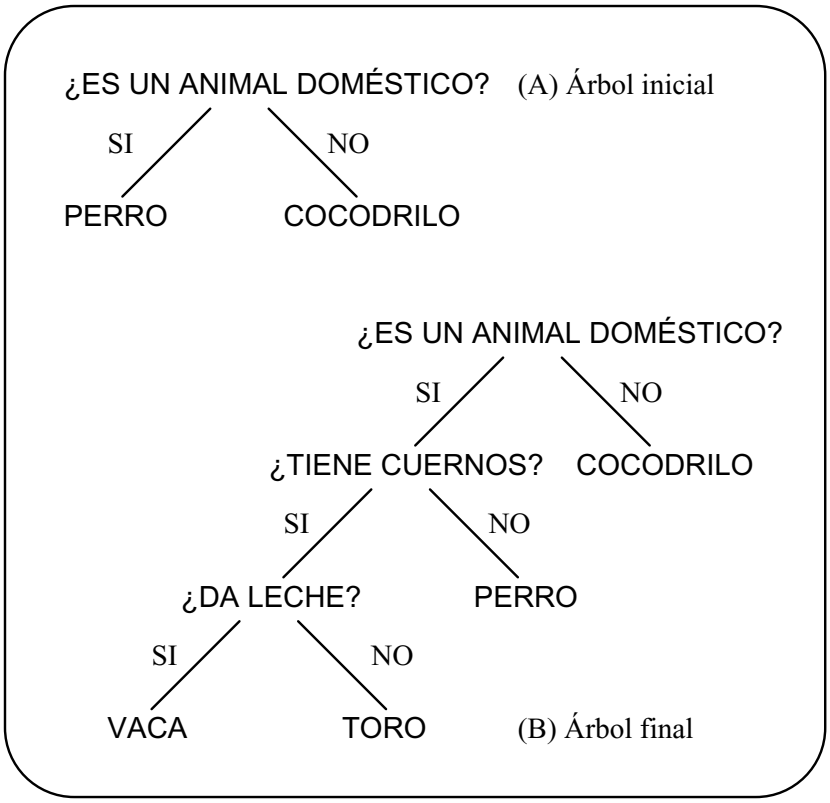


Figura B-2 Estructura de datos para el juego de *Animal*. (A) Árbol binario al inicio del juego. (B) Árbol binario al final del juego.

```
  {--------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Módulo de aplicación para el juego de adivinar animales.

    Requiere de AR_BIN.HUN y AR_JUEGO.HUN.
-----------------------------------------------------------------------}

aplicación

    persistente Juego

    si Juego:esNulo()
        Juego <- ArbolJuego:crea()
    fin si
    Juego:inicia()

fin aplicación

;-------- Fin de Archivo <AR_APLIC.HUN> ----------------------------
```

```
{-------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Módulo de definición para la clase ArbolBinario.
-------------------------------------------------------------------}
clase ArbolBinario hereda Genérico

{============================} defclase {============================}

    ;-----------------------------------------------------------------
    ;   Crea una instancia de la clase ArbolBinario que corresponde
    ;   a la raíz hoja con unObjeto como valor inicial.
    ;-----------------------------------------------------------------
    método crea(unObjeto ? Genérico)

        var unArbol

        unArbol <- receptor:nuevo()
        unArbol:modificaInfo(unObjeto)
        regresa unArbol

    fin método

{============================} definstancia {============================}

    var izquierda
    var derecha
    var información

    ;-----------------------------------------------------------------
    ;   Regresa el miembro izquierdo del receptor.
    ;-----------------------------------------------------------------
    método izq()
        regresa izquierda
    fin método

    ;-----------------------------------------------------------------
    ;   Regresa el miembro derecho del receptor.
    ;-----------------------------------------------------------------
    método der()
        regresa derecha
    fin método

    ;-----------------------------------------------------------------
    ;   Regresa la información del receptor.
    ;-----------------------------------------------------------------
    método info()
        regresa información
    fin método
```

```
    ;-------------------------------------------------------------
    ;   Cambia el miembro izquierdo del receptor por unNodo.
    ;-------------------------------------------------------------
    método modificaIzq(unNodo ? ArbolBinario)
        izquierda <- unNodo
    fin método

    ;-------------------------------------------------------------
    ;   Cambia el miembro derecho del receptor por unNodo.
    ;-------------------------------------------------------------
    método modificaDer(unNodo ? ArbolBinario)
        derecha <- unNodo
    fin método

    ;-------------------------------------------------------------
    ;   Cambia la información del receptor por unObjeto.
    ;-------------------------------------------------------------
    método modificaInfo(unObjeto ? Genérico)
        información <- unObjeto
    fin método

;-------- Fin de Archivo <AR_BIN.HUN> -----------------------------



{-------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Módulo de definición para la clase ArbolJuego, para el juego
    de adivinar animales.

    Requiere de AR_APLIC.HUN y AR_BIN.HUN.
-------------------------------------------------------------------}
clase ArbolJuego hereda ArbolBinario

{============================} defclase {============================}

    ;-------------------------------------------------------------
    ;   Crea una instancia de la clase ArbolJuega.
    ;-------------------------------------------------------------
    método crea()

        var unArbolJuego

        unArbolJuego <- antecesor:crea("ES UN ANIMAL DOMESTICO")
        unArbolJuego:modificaIzq(antecesor:crea("PERRO"))
        unArbolJuego:modificaDer(antecesor:crea("COCODRILO"))
        regresa unArbolJuego

    fin método
```

```
    ;-------------------------------------------------------------
    ;   Crea un nuevo nodo para la clase ArbolJuega, inicializado
    ;   con unaCadena.
    ;-------------------------------------------------------------
    método nuevoNodo(unaCadena ! Cadena)
        regresa antecesor:crea(unaCadena)
    fin método

{=========================} definstancia {=========================}

    ;-------------------------------------------------------------
    ;   Comienza el juego de un ArbolJuego.
    ;-------------------------------------------------------------
    método inicia()

        var op

        "Juego de adivinanzas de Animales. Ver. 1.00":imprimeNL()
        "ITESM CEM; A. Ortiz, 1994.":imprimeNL()
        "":imprimeNL()
        "PIENSA EN UN ANIMAL.":imprimeNL()
        ("YO TRATARE DE ADIVINAR CUAL ES HACIENDOTE VARIAS " + \
         "PREGUNTAS."):imprimeNL()
        ciclo
            "":imprimeNL()
            receptor:juega()
            "":imprimeNL()
            "¿DESEAS JUGAR OTRA VEZ?":imprime()
            op <- Booleano:leeSiNo()
            hasta op:no()
        fin ciclo

    fin método

    ;-------------------------------------------------------------
    ;   Continúa el juego a nivel del receptor. Regresa al receptor
    ;   o el resultado de procesar una hoja.
    ;-------------------------------------------------------------
    método juega()
        var op
        si izquierda:esNulo()
            regresa receptor:procesaHoja()
        otro
            ("¿" + información + "?"):imprime()
            op <- Booleano:leeSiNo()
            si op
                izquierda <- izquierda:juega()
            otro
                derecha <- derecha:juega()
            fin si
            regresa receptor
        fin si
    fin método
```

```
    ;-------------------------------------------------------------
    ;   Se llegó a una hoja. Si el jugador perdió, regresa el
    ;   receptor. En otro caso regresa un nuevo nodo el cual
    ;   contiene una nueva pregunta con una nueva respuesta
    ;   como miembro izquierdo y el receptor como miembro
    ;   derecho.
    ;-------------------------------------------------------------
    método procesaHoja()

        var op
        var nombre
        var afirmación
        var nuevoNodo

        ("¿ES UN (UNA) " + información + "?"):imprime()
        op <- Booleano:leeSiNo()
        si op
            "¡¡¡ ADIVINE !!!":imprimeNL()
            regresa receptor
        otro
            "* ESCRIBE EL NOMBRE DEL ANIMAL QUE PENSASTE : ":imprime()
            nombre <- (Cadena:lee()):comoMayúsculas()
            ("* ESCRIBE UNA AFIRMACION QUE SEA VERDAD PARA UN (UNA) " + \
                nombre):imprimeNL()
            ("  PERO QUE SE FALSA PARA UN (UNA) " + \
                información + " : "):imprime()
            afirmación <- (Cadena:lee()):comoMayúsculas()
            nuevoNodo <- ArbolJuego:nuevoNodo(afirmación)
            nuevoNodo:modificaIzq(ArbolJuego:nuevoNodo(nombre))
            nuevoNodo:modificaDer(receptor)
            regresa nuevoNodo
        fin si

    fin método

fin clase

;-------- Fin de Archivo <AR_JUEGO.HUN> ----------------------------
```

B.3 Juego de Gato

A continuación se presenta el típico programa del juego de Gato. El programa juega contra un jugador humano. Gana el que logre colocar tres de sus símbolos juntos en forma horizontal, vertical o diagonal. Una muestra del juego es la siguiente :

```
Juego de Gato. Ver. 1.00
ITESM CEM; A. Ortiz, 1994.

1¦2¦3   ¦ ¦
-+-+-  -+-+-
4¦5¦6   ¦ ¦
-+-+-  -+-+-
7¦8¦9   ¦ ¦

Turno de Ser Humano (X) : 1

1¦2¦3  X¦ ¦
-+-+-  -+-+-
4¦5¦6   ¦ ¦
-+-+-  -+-+-
7¦8¦9   ¦ ¦

Turno de Ser Máquina (O)

1¦2¦3  X¦ ¦
-+-+-  -+-+-
4¦5¦6   ¦ ¦
-+-+-  -+-+-
7¦8¦9  O¦ ¦

Turno de Ser Humano (X) : 6

1¦2¦3  X¦ ¦
-+-+-  -+-+-
4¦5¦6   ¦ ¦X
-+-+-  -+-+-
7¦8¦9  O¦ ¦

Turno de Ser Máquina (O)

1¦2¦3  X¦ ¦O
-+-+-  -+-+-
4¦5¦6   ¦ ¦X
-+-+-  -+-+-
7¦8¦9  O¦ ¦
```

```
Turno de Ser Humano (X) : 5

1¦2¦3  X¦ ¦O
-+-+-  -+-+-
4¦5¦6   ¦X¦X
-+-+-  -+-+-
7¦8¦9  O¦ ¦

Turno de Ser Máquina (O)

1¦2¦3  X¦ ¦O
-+-+-  -+-+-
4¦5¦6  O¦X¦X
-+-+-  -+-+-
7¦8¦9  O¦ ¦

Turno de Ser Humano (X) : 9

1¦2¦3  X¦ ¦O
-+-+-  -+-+-
4¦5¦6  O¦X¦X
-+-+-  -+-+-
7¦8¦9  O¦ ¦X

¡ Ganó Ser Humano !
¡ Perdió Ser Máquina !

¿Jugar otro juego? (S/N) : N

Total juegos : 1
Jugador Ser Humano. Ganados : 1
Jugador Ser Máquina. Ganados : 0
```

Al igual que en el programa anterior, este programa aprende cada vez que se juega. El programa consta de seis clases : (1) clase tablero, que se encarga de arbitrar el juego e imprimir el tablero; (2) clase jugador, que es una clase abstracta que sirve de antecesor para otras dos clases; (3) clase jugador-humano, que representa al usuario del juego (hereda de clase jugador); (4) clase jugador-computadora, que es el contrincante del jugador-humano (también hereda de la clase jugador); (5) clase conocimiento; y (6) clase bolsa. Estas dos últimas clases se utilizan por la clase jugador-computadora para aprender durante cada juego.

La forma en que una instancia de la clase jugador-computadora aprende es la siguiente : Cada jugada individual a la que se enfrenta alguna vez tiene asociada una estructura de datos conocida como *bolsa*; cada bolsa contiene en un principio tres instancias enteras por cada cuadro sin ocupar en el tablero, excepto por el cuadro central, en cuyo caso se tienen 18 instancias. De esta bolsa se obtiene un número al azar y es en esta posición

donde se tira. Si se gana el juego, se agregan cinco instancias de cada posición tirada en cada una de las bolsas respectivas que conformaron todo el juego, en caso de empate se agrega una sola instancia, y se pierde se retira una instancia por bolsa. Esto provoca que entre más juegos se jueguen, el jugador-computadora aprende de sus aciertos y errores, permitiendo que cada vez lo haga mejor.

Todos los objetos protagonistas son persistentes, permitiendo que el conocimiento adquirido en una serie de juegos se preserve para otra sesión. Es fácil modificar la aplicación para que jueguen dos personas entre sí, o que la computadora juegue contra sí mismo. Los listados fuente son los siguientes :

```
{------------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Juego de Gato. Módulo aplicación.

    Requiere de GT_TABL.HUN, GT_JUG, GT_COMP.HUN, GT_HUMAN,
    GT_CONOC.HUN, GT_BOLSA.
------------------------------------------------------------------------}
aplicación
    persistente Tablero
    persistente Computadora
    persistente Humano
    var op

    si Tablero:esNulo()
        Humano       <- JugadorHumano:nuevo("Ser Humano", 'X')
        Computadora  <- JugadorComputadora:nuevo("Ser Máquina", 'O')
        Tablero      <- TableroGato:nuevo(Humano, Computadora)
    fin si
    Tablero:inicio()
    ciclo
        Tablero:juega()
        "":imprimeNL()
        "¿Jugar otro juego?":imprime()
        hasta (Booleano:leeSiNo()):no()
    fin ciclo
    "":imprimeNL()
    "Total juegos : ":imprime()
    (Tablero:juegosJugados()):imprimeNL()
    Humano:estadísticas()
    Computadora:estadísticas()

fin aplicación

;-------- Fin de Archivo <GT_APLIC.HUN> ----------------------------
```

```
{--------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Juego de Gato. Módulo de definición para la clase TableroGato.

    Requiere de GT_APLIC.HUN, GT_JUG.HUN, GT_COMP.HUN, GT_HUMAN,
    GT_CONOC.HUN, GT_BOLSA.
--------------------------------------------------------------------}
clase TableroGato hereda Genérico

{============================} defclase {============================}

    ;-----------------------------------------------------------------
    ;   Crea una nueva instancia de la clase TableroGato con
    ;   jugadores j1 y j2.
    ;-----------------------------------------------------------------
    método nuevo(j1 ? Jugador, j2 ? Jugador)
        regresa (antecesor:nuevo()):inicializa(j1, j2)
    fin método

{============================} definstancia {============================}

    var tablero        ; (Cadena) Indica las posciones del tablero.
    var jugador1       ; (derivado de Jugador) Primero jugador.
    var jugador2       ; (derivado de Jugador) Segundo jugador.
    var últimoGanador  ; (derivado de Jugador o Nulo) Jugador que
                       ;   ganó último juego o nulo si hubo empate.
    var últimoPerdedor ; (derivado de Jugador o Nulo) Jugador que
                       ;   perdió último juego o nulo si hubo empate.
    var juegosJugados  ; (Entero) Total de juegos que han sido
                       ;   jugados.

    ;-----------------------------------------------------------------
    ;   Inicializa una instancia de la clase TableroGato con
    ;   jugadores j1 y j2.
    ;-----------------------------------------------------------------
    método inicializa(j1 ? Jugador, j2 ? Jugador)
        jugador1 <- j1
        jugador2 <- j2
        juegosJugados <- 0
        regresa receptor
    fin método
```

```
;-----------------------------------------------------------------
;   Imprime títulos del juego.
;-----------------------------------------------------------------
método inicio()
    "Juego de Gato. Ver. 1.00":imprimeNL()
    "ITESM CEM; A. Ortiz, 1994.":imprimeNL()
fin método

;-----------------------------------------------------------------
;   Regresa la disposición del tablero como una cadena.
;-----------------------------------------------------------------
método disposición()
    regresa tablero
fin método

;-----------------------------------------------------------------
;   Regresa el número de juegos jugados.
;-----------------------------------------------------------------
método juegosJugados()
    regresa juegosJugados
fin método

;-----------------------------------------------------------------
;   Imprime el tablero.
;-----------------------------------------------------------------
método imprime()

    var índice
    var columna
    var renglón
    var barra

    barra <- "" | @196 | @197 | @196 | @197 | @196
    "":imprimeNL()
    índice <- 1
    renglón <- 1
    ciclo
        selección renglón
        opción 1
            ("1" | @179 + "2" | @179 + "3  "):imprime()
        opción 2
            ("4" | @179 + "5" | @179 + "6  "):imprime()
        opción 3
            ("7" | @179 + "8" | @179 + "9  "):imprime()
        fin selección
        columna <- 1
        ciclo
            si (tablero:obtén(índice)):esDígito()
                " ":imprime()
            otro
                (tablero:obtén(índice)):imprime()
            fin si
            índice <- índice + 1
            hasta columna = 3
```

```
            @179:imprime()
            columna <- columna + 1
        fin ciclo
        "":imprimeNL()
        hasta renglón = 3
        (barra + "  " + barra):imprimeNL()
        renglón <- renglón + 1
    fin ciclo
    "":imprimeNL()

fin método

;-----------------------------------------------------------------
;   Regresa verdad si existe empate (todas las casillas están
;   ocupadas) en el tablero, de otra forma regresa falso.
;-----------------------------------------------------------------
método hayEmpate()

    var índice

    índice <- 1
    ciclo
        hasta índice > 9
        si (tablero:obtén(índice)):esDígito()
            regresa falso
        fin si
        índice <- índice + 1
    fin ciclo
    regresa verdad

fin método

;-----------------------------------------------------------------
;   Regresa verdad si símbolo se repite tres veces en la
;   vertical, horizontal o diagonal, regresa falso de otra
;   forma.
;-----------------------------------------------------------------
método ganóJuego(símbolo ! Carácter)

    var p1, p2, p3, p4, p5, p6, p7, p8, p9

    p1 <- tablero:obtén(1) = símbolo
    p2 <- tablero:obtén(2) = símbolo
    p3 <- tablero:obtén(3) = símbolo
    p4 <- tablero:obtén(4) = símbolo
    p5 <- tablero:obtén(5) = símbolo
    p6 <- tablero:obtén(6) = símbolo
    p7 <- tablero:obtén(7) = símbolo
    p8 <- tablero:obtén(8) = símbolo
    p9 <- tablero:obtén(9) = símbolo
    si (p1 & p2 & p3) | (p4 & p5 & p6) | (p7 & p8 & p9) | \
       (p1 & p4 & p7) | (p2 & p5 & p8) | (p3 & p6 & p9) | \
       (p1 & p5 & p9) | (p3 & p5 & p7)
        regresa verdad
```

```
    otro
        regresa falso
    fin si

fin método

;-------------------------------------------------------------------
;   Regresa verdad si pos hace referencia a una localidad
;   disponible en el tablero, de otra forma regresa falso.
;-------------------------------------------------------------------
método estáLibre(pos ! Entero)
    si (1 <= pos) & (pos <= 9)
        regresa (tablero:obtén(pos)):esDígito()
    otro
        regresa falso
    fin si
fin método

;-------------------------------------------------------------------
;   Arbitrea un juego de gato.
;-------------------------------------------------------------------
método juega()

    var jugAnterior
    var jugActual
    var temp

    tablero <- "123456789"
    jugador1:nuevoJuego()
    jugador2:nuevoJuego()

    ; Determinar quién juega primero.
    si últimoGanador:esNulo()
        si Entero:aleatorio(2) = 0
            jugAnterior <- jugador2
            jugActual <- jugador1
        otro
            jugAnterior <- jugador1
            jugActual <- jugador2
        fin si
    otro
        jugAnterior <- últimoPerdedor
        jugActual <- últimoGanador
    fin si

    ; Comenzar el juego.
    juegosJugados <- juegosJugados + 1
    ciclo
        receptor:imprime()
        tablero:modifica(jugActual:tira(receptor), \
            jugActual:símbolo())
        si receptor:ganóJuego(jugActual:símbolo())
            receptor:imprime()
            jugActual:ganó()
```

```
                jugAnterior:perdió()
                últimoGanador <- jugActual
                últimoPerdedor <- jugAnterior
                regresa nulo
            fin si
            hasta receptor:hayEmpate()
            temp        <- jugActual
            jugActual   <- jugAnterior
            jugAnterior <- temp
        fin ciclo

        ; No hubo ganadores.
        jugador1:empató()
        jugador2:empató()
        receptor:imprime()
        "":imprimeNL()
        "Juego Empatado":imprimeNL()
        últimoGanador <- nulo
        últimoPerdedor <- nulo

    fin método

fin clase

;-------- Fin de Archivo <GT_TABL.HUN> -----------------------------



{---------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Juego de Gato. Módulo de definición para la clase Jugador.

    Requiere de GT_APLIC.HUN, GT_TABL.HUN, GT_COMP.HUN, GT_HUMAN,
    GT_CONOC.HUN, GT_BOLSA.
---------------------------------------------------------------------}
clase Jugador hereda Genérico

{============================} defclase {============================}

    ;-------------------------------------------------------------
    ;   Crea una nueva instancia de la clase Jugador con unNombre
    ;   y unSímbolo.
    ;-------------------------------------------------------------
    método nuevo(unNombre ! Cadena, unSímbolo ! Carácter)
        regresa (antecesor:nuevo()):inicializa(unNombre, unSímbolo)
    fin método

{=========================} definstancia {=========================}

    var nombre          ; (Cadena) Nombre del jugador.
    var símbolo         ; (Carácter) Símbolo del jugador (X ó O).
```

```
var juegosGanados   ; (Entero) Número de juegos ganados.
var juegosPerdidos  ; (Entero) Número de juegos perdidos.

;-----------------------------------------------------------------
;   Inicializa una instancia de la clase Jugador con unNombre y
;   unSímbolo.
;-----------------------------------------------------------------
método inicializa(unNombre ! Cadena, unSímbolo ! Carácter)
    nombre <- unNombre
    símbolo <- unSímbolo
    juegosGanados <- 0
    juegosPerdidos <- 0
    regresa receptor
fin método

;-----------------------------------------------------------------
;   Inicializa las variables de instancia necesarias para
;   comnezar un nuevo juego.
;-----------------------------------------------------------------
método nuevoJuego()
fin método

;-----------------------------------------------------------------
;   Regresa el símbolo con el juega la instancia.
;-----------------------------------------------------------------
método símbolo()
    regresa símbolo
fin método

;-----------------------------------------------------------------
;   Se invoca cuando la instancia gana un juego.
;-----------------------------------------------------------------
método ganó()
    juegosGanados <- juegosGanados + 1
    ("¡ Ganó " + nombre + " !"):imprimeNL()
fin método

;-----------------------------------------------------------------
;   Se invoca cuando la instancia pierde un juego.
;-----------------------------------------------------------------
método perdió()
    juegosPerdidos <- juegosPerdidos + 1
    ("¡ Perdió " + nombre + " !"):imprimeNL()
fin método

;-----------------------------------------------------------------
;   Se invoca cuando la instancia empata un juego.
;-----------------------------------------------------------------
método empató()
fin método
```

```
 ;-------------------------------------------------------------
    ;   Imprime el número de juegos que ha ganado la instancia.
    ;-------------------------------------------------------------
    método estadísticas()
        ("Jugador " + nombre + ". Ganados : "):imprime()
        juegosGanados:imprimeNL()
    fin método

fin clase

;-------- Fin de Archivo <GT_JUG.HUN> ----------------------------



{-------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Juego de Gato. Módulo de definición para la clase JugadorHumano.

    Requiere de GT_APLIC.HUN, GT_TABL.HUN, GT_JUG.HUN, GT_COMP,
    GT_CONOC.HUN, GT_BOLSA.
-------------------------------------------------------------------}
clase JugadorHumano hereda Jugador

{=========================} definstancia {=========================}

    ;-------------------------------------------------------------
    ;   Regresa el lugar donde se desea realizar un tiro.
    ;-------------------------------------------------------------
    método tira(unTablero ! TableroGato)

        var op

        ciclo
            ("Turno de " + nombre + " (" | símbolo + ") : "):imprime()
            op <- Entero:lee()
            hasta unTablero:estáLibre(op)
            "Jugada inválida. Se repite.":imprimeNL()
        fin ciclo
        regresa op

    fin método

fin clase

;-------- Fin de Archivo <GT_HUMAN.HUN> ----------------------------
```

```
{-------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Juego de Gato. Módulo de definición para la clase
    JugadorComputadora.

    Requiere de GT_APLIC.HUN, GT_TABL.HUN, GT_JUG.HUN, GT_HUMAN,
    GT_CONOC.HUN, GT_BOLSA.
-------------------------------------------------------------------}
clase JugadorComputadora hereda Jugador

{=========================} definstancia {=========================}

    var conocimiento     ; (Arreglo) Registro de todo el nuevo
                         ;   conocimiento producido.
    var historiaBolsas   ; (Arreglo) Bolsas de la jugada actual.
    var historiaTiradas  ; (Arreglo) Tiradas del juego actual.
    var siguiente        ; (Entero) Indice de la siguiente posición
                         ;   disponible en historiaBolsas e
                         ;   historiaTiradas.

    ;-----------------------------------------------------------------
    ;   Inicializa una instancia de la clase JugadorComputadora
    ;   con unNombre y unSímbolo.
    ;-----------------------------------------------------------------
    método inicializa(unNombre ! Cadena, unSímbolo ! Carácter)
        conocimiento <- Conocimiento:nuevo()
        regresa antecesor:inicializa(unNombre, unSímbolo)
    fin método

    ;-----------------------------------------------------------------
    ;   Inicializa las variables de instancia necesarias para
    ;   comnezar un nuevo juego.
    ;-----------------------------------------------------------------
    método nuevoJuego()
        historiaBolsas     <- Arreglo:nuevo(5)
        historiaTiradas    <- Arreglo:nuevo(5)
        siguiente          <- 1
    fin método

    ;-----------------------------------------------------------------
    ;   Regresa el lugar donde se desea realizar un tiro.
    ;-----------------------------------------------------------------
    método tira(unTablero ! TableroGato)

        var bolsaActual
        var op

        ("Turno de " + nombre + " (" | símbolo + ") "):imprimeNL()
        "Pensando ...":imprime()
        op <- receptor:mejorTirada(unTablero:disposición())
        si op = 0
```

```
        bolsaActual <- conocimiento:buscaGenera( \
            unTablero:disposición())
        op <- bolsaActual:sortea()
        historiaBolsas:modifica(siguiente, bolsaActual)
        historiaTiradas:modifica(siguiente, op)
        siguiente <- siguiente + 1
    fin si
    ("" | @13 + "            "):imprime()
    regresa op

fin método

;-----------------------------------------------------------------
;   Se invoca cuando la instancia gana un juego.
;-----------------------------------------------------------------
método ganó()

    var i

    antecesor:ganó()
    i <- 1
    ciclo
        hasta i = siguiente
        (historiaBolsas:obtén(i)): \
            agregaNVeces(5, historiaTiradas:obtén(i))
        i <- i + 1
    fin ciclo

fin método

;-----------------------------------------------------------------
;   Se invoca cuando la instancia pierde un juego.
;-----------------------------------------------------------------
método perdió()

    var i

    antecesor:perdió()
    i <- 1
    ciclo
        hasta i = siguiente
        (historiaBolsas:obtén(i)): \
            retiraNVeces(1, historiaTiradas:obtén(i))
        i <- i + 1
    fin ciclo
    (historiaBolsas:obtén(siguiente - 1)): \
        retiraTodas(historiaTiradas:obtén(siguiente - 1))

fin método
```

```
;-------------------------------------------------------------
;   Se invoca cuando la instancia empata un juego.
;-------------------------------------------------------------
método empató()

    var i

    antecesor:empató()
    i <- 1
    ciclo
        hasta i = siguiente
        (historiaBolsas:obtén(i)): \
            agregaNVeces(1, historiaTiradas:obtén(i))
        i <- i + 1
    fin ciclo

fin método


;-------------------------------------------------------------
;   Regresa la mejor tirada para la instancia dentro del tablero.
;   Determina primero si puede ganar, y sino entonces verifica
;   si puede evitar perder. De otra forma regresa 0.
;-------------------------------------------------------------
método mejorTirada(disposición ! Cadena)

    var p1, p2, p3, p4, p5, p6, p7, p8, p9

    p1 <- receptor:pondera(disposición:obtén(1))
    p2 <- receptor:pondera(disposición:obtén(2))
    p3 <- receptor:pondera(disposición:obtén(3))
    p4 <- receptor:pondera(disposición:obtén(4))
    p5 <- receptor:pondera(disposición:obtén(5))
    p6 <- receptor:pondera(disposición:obtén(6))
    p7 <- receptor:pondera(disposición:obtén(7))
    p8 <- receptor:pondera(disposición:obtén(8))
    p9 <- receptor:pondera(disposición:obtén(9))

    ; Busca opción para ganar.
    si (p1 + p2 + p3) = 20
        regresa receptor:selecciona(p1, 1, p2, 2, p3, 3)
    otrosi (p4 + p5 + p6) = 20
        regresa receptor:selecciona(p4, 4, p5, 5, p6, 6)
    otrosi (p7 + p8 + p9) = 20
        regresa receptor:selecciona(p7, 7, p8, 8, p9, 9)
    otrosi (p1 + p4 + p7) = 20
        regresa receptor:selecciona(p1, 1, p4, 4, p7, 7)
    otrosi (p2 + p5 + p8) = 20
        regresa receptor:selecciona(p2, 2, p5, 5, p8, 8)
    otrosi (p3 + p6 + p9) = 20
        regresa receptor:selecciona(p3, 3, p6, 6, p9, 9)
    otrosi (p1 + p5 + p9) = 20
        regresa receptor:selecciona(p1, 1, p5, 5, p9, 9)
    otrosi (p3 + p5 + p7) = 20
```

```
        regresa receptor:selecciona(p3, 3, p5, 5, p7, 7)
    fin si

    ; Busca opción para no perder.
    si (p1 + p2 + p3) = 2
        regresa receptor:selecciona(p1, 1, p2, 2, p3, 3)
    otrosi (p4 + p5 + p6) = 2
        regresa receptor:selecciona(p4, 4, p5, 5, p6, 6)
    otrosi (p7 + p8 + p9) = 2
        regresa receptor:selecciona(p7, 7, p8, 8, p9, 9)
    otrosi (p1 + p4 + p7) = 2
        regresa receptor:selecciona(p1, 1, p4, 4, p7, 7)
    otrosi (p2 + p5 + p8) = 2
        regresa receptor:selecciona(p2, 2, p5, 5, p8, 8)
    otrosi (p3 + p6 + p9) = 2
        regresa receptor:selecciona(p3, 3, p6, 6, p9, 9)
    otrosi (p1 + p5 + p9) = 2
        regresa receptor:selecciona(p1, 1, p5, 5, p9, 9)
    otrosi (p3 + p5 + p7) = 2
        regresa receptor:selecciona(p3, 3, p5, 5, p7, 7)
    fin si

    ; No se encontró nada.
    regresa 0

fin método

;-----------------------------------------------------------------
;   Regresa 10 si en pos se encuentra el símbolo de la instancia,
;   1 si se encuentra el símbolo del contrincante, y 0 en
;   caso de estar vacía la localidad.
;-----------------------------------------------------------------
método pondera(pos ! Carácter)
    si pos = símbolo
        regresa 10
    otrosi pos:esDígito()
        regresa 0
    otro
        regresa 1
    fin si
fin método

;-----------------------------------------------------------------
;   Dados tres posiciones en las localidades v1, v2 y v3
;   para los valores ponderados p1, p2 y p3, regresa la
;   única localidad que debe estar vacía.
;-----------------------------------------------------------------
método selecciona(p1 ! Entero, v1 ! Entero, \
                  p2 ! Entero, v2 ! Entero, \
                  p3 ! Entero, v3 ! Entero)
    selección 0
    opción p1
        regresa v1
    opción p2
```

```
            regresa v2
        opción p3
            regresa v3
        fin selección
    fin método

fin clase

;-------- Fin de Archivo <GT_COMP.HUN> -----------------------------



{--------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Juego de Gato. Módulo de definición para la clase Conicimiento.

    Requiere de GT_APLIC.HUN, GT_TABL.HUN, GT_JUG, GT_COMP.HUN,
    GT_HUMAN, GT_BOLSA.
--------------------------------------------------------------------}
clase Conocimiento hereda Genérico

{============================} defclase {============================}

    ;-----------------------------------------------------------------
    ;   Crea una nueva instancia de la clase Conocimiento.
    ;-----------------------------------------------------------------
    método nuevo()
        regresa (antecesor:nuevo()):inicializa()
    fin método

{=========================} definstancia {=========================}

    var llave      ; (Arreglo) Llaves para indexar el conocimiento.
    var valor      ; (Arreglo) Bolsas donde se almacena los valores
                   ;   del conocimiento generado.
    var siguiente  ; (Entero) Indice de la siguiente posición
                   ;   disponible en llave y valor.
    var tamaño     ; (Entero) Tamaño de los arreglos llave y valor.

    ;-----------------------------------------------------------------
    ;   Inicializa una instancia de la clase Conocimiento.
    ;-----------------------------------------------------------------
    método inicializa()
        tamaño    <- 20
        llave     <- Arreglo:nuevo(tamaño)
        valor     <- Arreglo:nuevo(tamaño)
        siguiente <- 1
        regresa receptor
    fin método
```

```
    ;-------------------------------------------------------------
    ;   Busca una jugada igual a la de disposición para regresar
    ;   la bolsa de conocimiento asociada. En caso de no
    ;   enconontrarla regresa una nueva bolsa.
    ;-------------------------------------------------------------
    método buscaGenera(disposición ! Cadena)

        var índice
        var nuevaBolsa

        índice <- 1
        ciclo
            hasta índice = siguiente
            si disposición = llave:obtén(índice)
               regresa valor:obtén(índice)
            fin si
            índice <- índice + 1
        fin ciclo
        si siguiente > tamaño
            tamaño <- tamaño + 20
            llave  <- llave:cambiaLongitud(tamaño)
            valor  <- valor:cambiaLongitud(tamaño)
        fin si
        llave:modifica(siguiente, disposición:copia())
        nuevaBolsa <- Bolsa:ponInfo(disposición)
        valor:modifica(siguiente, nuevaBolsa)
        siguiente <- siguiente + 1
        regresa nuevaBolsa

    fin método

fin clase

;-------- Fin de Archivo <GT_CONOC.HUN> ----------------------------



{--------------------------------------------------------------------
    ITESM, CEM; A. Ortiz, 1994.

    Archivo fuente del lenguaje Huntul Ver. 1.00

    Juego de Gato. Módulo de definición para la clase Bolsa.

    Requiere de GT_APLIC.HUN, GT_TABL.HUN, GT_JUG, GT_COMP.HUN,
    GT_HUMAN, GT_CONOC.
--------------------------------------------------------------------}
clase Bolsa hereda Genérico

{============================} defclase {============================}
```

```
    ;-------------------------------------------------------------
    ;   Crea una nueva instancia de la clase Bolsa.
    ;-------------------------------------------------------------
    método nuevo()
        regresa (antecesor:nuevo()):inicializa()
    fin método

    ;-------------------------------------------------------------
    ;   Crea una nueva instancia de la clase Bolsa, rellenándola
    ;   de información con cierta heurística.
    ;-------------------------------------------------------------
    método ponInfo(unTablero ! Cadena)

        var nuevaBolsa
        var i

        nuevaBolsa <- Bolsa:nuevo()
        i <- 1
        ciclo
            hasta i > 9
            si (unTablero:obtén(i)):esDígito()
                nuevaBolsa:agregaNVeces(3, i)
            fin si
            i <- i + 1
        fin ciclo
        si (unTablero:obtén(5)):esDígito()
                nuevaBolsa:agregaNVeces(24, 5)
        fin si
        regresa nuevaBolsa

    fin método

{=========================} definstancia {=========================}

    var bolsa   ; (Arreglo) Información de la bolsa.
    var tamaño  ; (Entero) Tamaño del arreglo bolsa.

    ;-------------------------------------------------------------
    ;   Inicializa una instancia de la clase Bolsa.
    ;-------------------------------------------------------------
    método inicializa()

        var i
        bolsa <- Arreglo:nuevo(9)
        tamaño <- 0
        i <- 1
        ciclo
            hasta i > 9
            bolsa:modifica(i, 0)
            i <- i + 1
        fin ciclo
        regresa receptor

    fin método
```

```
    ;-------------------------------------------------------------
    ;   Agrega al receptor un elemento el número de veces indicado.
    ;-------------------------------------------------------------
    método agregaNVeces(veces ! Entero, elemento ! Entero)
        tamaño <- tamaño + veces
        bolsa:modifica(elemento, bolsa:obtén(elemento) + veces)
    fin método

    ;-------------------------------------------------------------
    ;   Retira del receptor un elemento el número de veces indicado.
    ;   Siempre permanece al menos un elemento.
    ;-------------------------------------------------------------
    método retiraNVeces(veces ! Entero, elemento ! Entero)
        si (bolsa:obtén(elemento) - veces) > 1
            tamaño <- tamaño - veces
            bolsa:modifica(elemento, bolsa:obtén(elemento) - veces)
        fin si
    fin método

    ;-------------------------------------------------------------
    ;   Elimina todos las instancias de un elemento dado de la bolsa.
    ;-------------------------------------------------------------
    método retiraTodas(elemento ! Entero)
        tamaño <- tamaño - bolsa:obtén(elemento)
        bolsa:modifica(elemento, 0)
    fin método

    ;-------------------------------------------------------------
    ;   Realiza un sorteo para regresar un elemento contenido en
    ;   la bolsa.
    ;-------------------------------------------------------------
    método sortea()
        var aleatorio, acumulado, i

        si tamaño = 0
            regresa 0
        fin si
        aleatorio <- Entero:aleatorio(tamaño) + 1
        acumulado <- 0
        i <- 1
        ciclo
            si (acumulado < aleatorio) & \
                (aleatorio <= (acumulado + bolsa:obtén(i)))
                regresa i
            fin si
            acumulado <- acumulado + bolsa:obtén(i)
            i <- i + 1
            hasta falso
        fin ciclo
    fin método

fin clase

;-------- Fin de Archivo <GT_BOLSA.HUN> ----------------------------
```