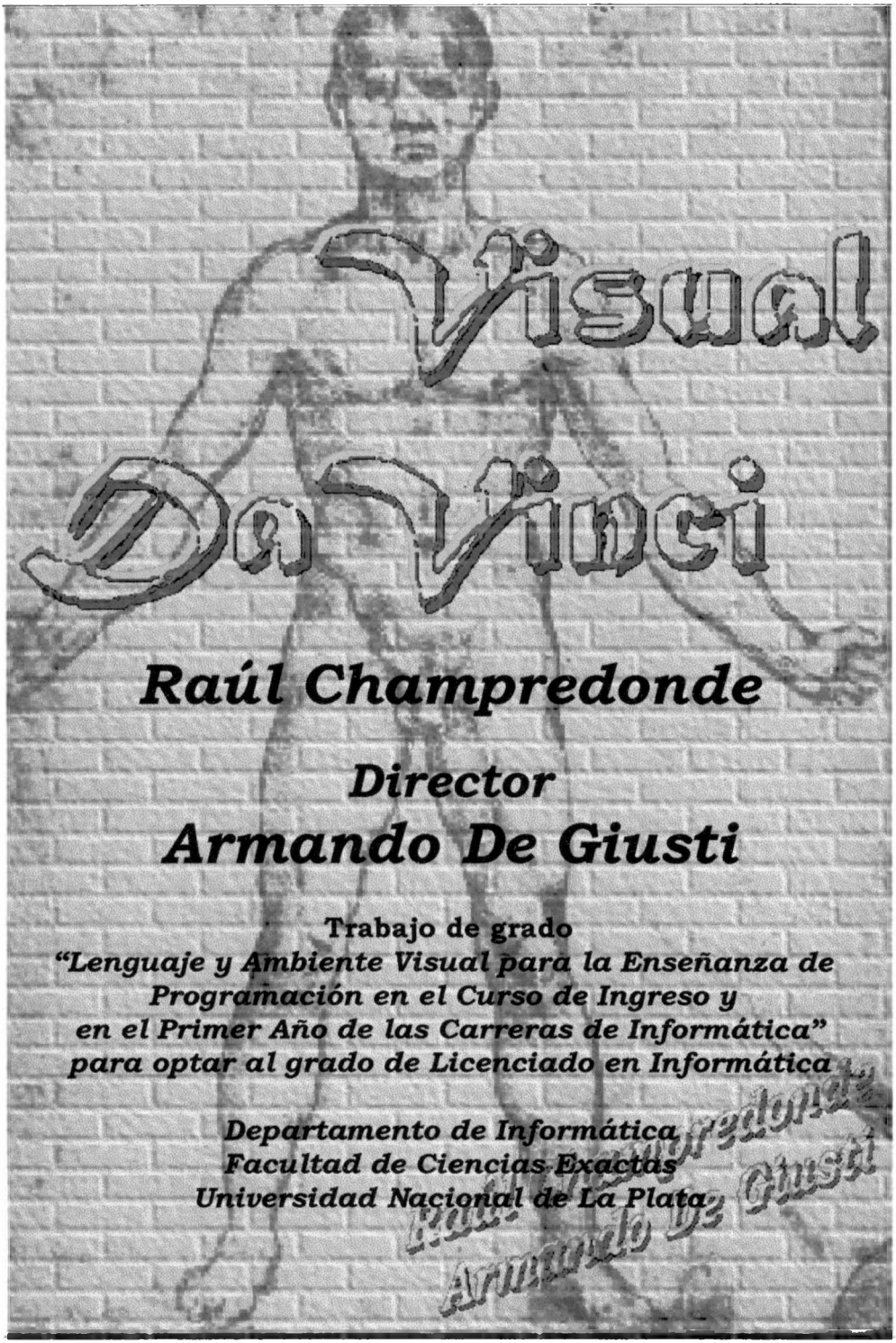


Raúl Champredonde

Director

Armando De Giusti

Trabajo de grado

"Lenguaje y Ambiente Visual para la Enseñanza de Programación en el Curso de Ingreso y en el Primer Año de las Carreras de Informática"

para optar al grado de Licenciado en Informática

Departamento de Informática
Facultad de Ciencias Exactas
Universidad Nacional de La Plata

A mi familia

Prólogo

Este trabajo describe el Visual DaVinci y su ambiente, y presenta los aspectos más importantes de su implementación.

Visual DaVinci está específicamente diseñado para la enseñanza de programación estructurada en el Curso de Ingreso a la carrera Licenciatura en Informática y en parte de un primer curso de Programación de Computadoras.

Visual DaVinci es un lenguaje visual que utiliza el paradigma de flujo de control por medio de la especificación de un diagrama con características similares a las de Nassi-Schneiderman, obligando al programador a desarrollar código estructurado.

La especificación del diagrama es derivada automática y simultáneamente a código textual, el cual puede ser modificado por el programador.

Se permite también el desarrollo de código en forma textual, por medio de una sintaxis restringida que también obliga a generar código estructurado y a respetar ciertas reglas que hacen al estilo de programación.

En ambos casos, el desarrollo, la verificación de sintaxis y la ejecución de un algoritmo se realiza desde el mismo ambiente.

Visual DaVinci está provisto de un conjunto limitado de tipos y no tiene constructores de tipos definidos por el usuario, ya que está orientado más a los algoritmos que a las estructuras de datos.

Además, Visual DaVinci es la base para futuras extensiones, destinadas a la investigación sobre lenguajes visuales y a la educación.

Agradecimientos

Quiero agradecer a todos aquellos que hicieron que disfrutara durante todos estos años del estudio de esta carrera. Sin ellos no lo hubiera hecho tanto como lo hice.

Debo agradecer también a Destino y a mis padres que hicieron lo suyo.

A mi familia, que tiene una paciencia de "fierro".

Un agradecimiento especial merece mi hermano Jorge por haberme obligado a volver de Pigüé para comenzar esta carrera.

Otro para Mariano de quien logre aprender unas cuantas cosas.

A todos mis compañeros del LIDI que me han soportado tanto todo este tiempo, en especial a Tito quien no tenía ninguna obligación de hacerlo.

A mis compañeros de estudios, por su amistad, su tiempo y sus mates.

A la Universidad Pública y Gratuita, y a todos los profesores de la misma que, a pesar de todo, han logrado que algo aprenda.

Por último, gracias a todos los escollos, por haberme dado la posibilidad de sortearlos.

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

Parte 1

Introducción

En esta primera parte se repasan brevemente los conceptos generales que, de una forma u otra, debieron ser considerados en este trabajo.

Capítulo 1

Los Primeros Lenguajes de Programación

La programación es la actividad de describir un algoritmo en una notación particular, un lenguaje de programación, para que luego sea ejecutado por una computadora.

Los primeros programas debieron ser desarrollados en lenguajes de muy bajo nivel, sin asistencia alguna de un sistema operativo y reflejando estrechamente la arquitectura de la computadora para la cual esos programas fueron desarrollados.

Una década después, a fines de los '50, aparecen los primeros lenguajes de alto nivel, orientados a problemas específicos. Y luego los de propósito general [Ghezzi87].

A medida que las memorias de las computadoras crecían en capacidad y los procesadores en velocidad de ejecución, aumentaba la complejidad de los problemas que se pretendían solucionar. Así, se fueron creando técnicas de desarrollo de software. La disciplina que se encargó del estudio y construcción de esas técnicas fue denominada Ingeniería de Software [Ghezzi91] [Yourdon93].

Una de las metodologías de las que se ocupa y ocupó la Ingeniería de Software, y que aún hoy es muy utilizada para un vasto conjunto de problemas, abarca el análisis, diseño y programación estructurados [Aguilar88].

1.1. Programación Modular y Estructurada

La programación modular es un método de diseño flexible y potente que apunta a mejorar la productividad de un programa. Está basada en la célebre frase "devide and conquer".

La idea central de la programación modular es dividir un programa en pequeños módulos independientes entre sí, cada uno encargado de realizar una única tarea.

Cada módulo se analiza, se codifica y se depura por separado.

Cada programa tiene un módulo principal que controla todo lo que sucede durante su ejecución. Es quien transfiere el control a los submódulos, de modo tal que estos ejecuten las acciones de las cuales están encargados. Cuando un submódulo concluye su tarea, devuelve el control al módulo principal.

De igual manera, un módulo cualquiera puede transferir el control temporalmente a un submódulo. Este debe devolver el control al módulo del cual lo recibió.

Dado que cada módulo es independiente de los demás, distintos programadores pueden trabajar simultáneamente sobre diferentes partes de un mismo

programa. Esto reduce el tiempo de diseño de un programa y su posterior codificación.

Además, un módulo se puede modificar sin afectar a los demás.

El término programación estructurada se refiere a un conjunto de técnicas que aumentan la productividad de un programa reduciendo considerablemente el tiempo requerido para su escritura, verificación, depuración y mantenimiento.

Utiliza un número limitado de estructuras de control que minimizan la complejidad de los problemas y por consiguiente la cantidad de errores.

La programación estructurada utiliza un diseño top-down o descendente, recursos abstractos y estructuras básicas.

Descomponer un programa en términos de los recursos abstractos es descomponer una determinada acción compleja en una cierta cantidad de pequeñas acciones simples. Una acción simple es aquella que no puede ser descompuesta.

El diseño top-down es el proceso mediante el cual un problema se descompone en una serie de niveles o pasos sucesivos de refinamiento, conformando una estructura jerárquica que relaciona los distintos refinamientos.

Böhm y Jacopini demostraron en 1966, a través de lo que se conoce como el teorema de la programación estructurada, que todo programa propio puede ser escrito utilizando únicamente las estructuras de control básicas secuencia, selección e iteración. Se entiende por programa propio a aquel que cumple con las siguientes características:

- *Posee un único punto de entrada y un único punto de salida*
- *Existen caminos que unen la entrada con la salida, que se pueden seguir y que pasan por todas las partes del programa*
- *Todas las instrucciones son ejecutables y no existen iteraciones infinitas*

1.2. Algunas Notaciones de Especificación

Existe un conjunto variado de notaciones de especificación, de las cuales algunas están muy difundidas y aplicadas [Ghezzi91].

Los diagramas de flujo de datos (DFD) son una notación muy usada para la especificación de las funciones de un sistema de información, que describe a un sistema como colecciones de datos manipulados por funciones [Stevens74] [Yourdon75].

Las máquinas de estados finitos o los diagramas de transición de estados son importantes modelos simples para la descripción de los aspectos relacionados con el control de un sistema.

Las redes de Petri son un formalismo gráfico que relaciona estados y datos [Petri62] [Peterson81]. Existe una gran cantidad de extensiones de redes de Petri que buscan adecuarlas a la especificación de distintos tipos de sistemas (por ejemplo, redes de Petri temporizadas) [Ghezzi89].

El modelo de entidad/relación está motivado en la necesidad de modelar conceptualmente los datos, en una forma apropiada para la especificación de la visión del usuario y de los requerimientos lógicos de los sistemas centrados en grandes colecciones de datos interrelacionados [Chen76].

Hay otras notaciones que son realmente formales, como las especificaciones lógicas o algebraicas, basadas en la lógica de primer orden y en álgebras heterogéneas respectivamente.

Las notaciones anteriores son utilizadas fundamentalmente en lo que tiene que ver con las etapas de análisis y diseño.

Las herramientas que se mencionan a continuación, en cambio, se utilizan básicamente para la especificación de procesos; es decir, para la descripción detallada de las acciones que debe realizar una unidad del sistema para convertir sus entradas en salidas.

El lenguaje estructurado es un subconjunto del leguaje cotidiano, pero con fuertes restricciones sobre el tipo de frases que se pueden utilizar y la forma en que esas frases pueden ser unidas. Intenta ser un punto medio entre la formalidad de los lenguajes de programación y la informalidad del lenguaje corriente.

Las precondiciones y las postcondiciones son útiles para especificar la función que se debe realizar, sin ocuparse mucho del algoritmo que llevará a cabo dicha función.

Si se debe producir alguna salida o tomar alguna decisión sobre la base de condiciones complejas, puede utilizarse una tabla de decisiones. Este método relaciona

cada una de las posibles combinaciones de valores de las variables que intervienen, con las acciones que se deben ejecutar [McDaniel70] [Pollack71] [Gildersleeve70].

Pasando a las notaciones gráficas de especificación de procesos, la primera que se menciona es el legendario diagrama de flujo. Este diagrama puede resultar muy útil para describir algoritmos que serán luego implementados en lenguaje assembler. Pero su utilización en la especificación estructurada requiere de muchos cuidados. Por ejemplo, es responsabilidad de quien confecciona el diagrama de flujo mantenerlo estructurado [Scanlan87a] [Böhm66].

Contrariamente a los diagramas de flujo, los diagramas de Nassi-Schneiderman fueron creados especialmente para la especificación estructurada de procesos.

1.8. Diagramas de Nassi-Schneiderman

Los diagramas de Nassi-Schneiderman nacieron a mediados de la década del 70, cuando la programación estructurada comenzó a popularizarse [Nassi73] [Chapin74].

En estos diagramas, cada sentencia simple está representada por una caja (un rectángulo). Una sucesión de cajas, como la de la figura 1.1, representa una secuencia de sentencias. Para la selección binaria se utiliza la notación de la figura 1.2 y la construcción iterativa se representa por medio de la notación gráfica de la figura 1.3

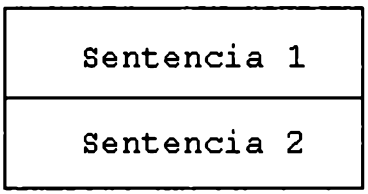


***Figura 1.1**: Representación de Secuencia de Sentencias*

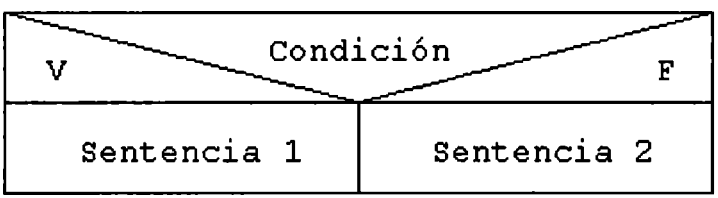


***Figura 1.2**: Representación de la construcción Si-entonces-Sino*

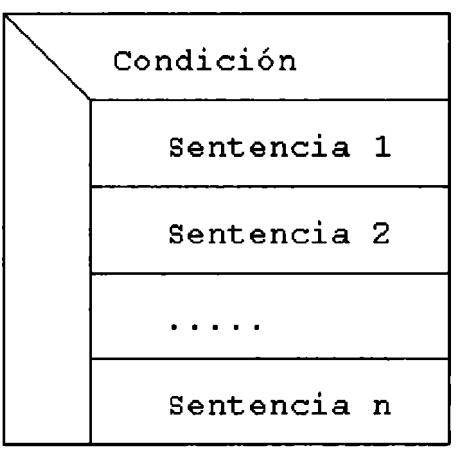


***Figura 1.3**: Representación de la construcción Mientras*

Los diagramas de Nassi-Schneiderman son estructurados por naturaleza, y usualmente más organizados y comprensibles que los diagramas de flujo.

Por ese motivo, generalmente se los prefiere para la especificación de procesos.

Sin embargo, si se trata de la especificación de una tarea compleja, puede requerir de una considerable cantidad de gráficos. Razón por la cual, para muchos analistas expertos, los diagramas de Nassi-Schneiderman resultan tediosos.

Por otro lado, se han realizado ciertos estudios [Scanlan87b] a partir de los cuales se puede concluir que los estudiantes de informática se sienten muy a gusto con los diagramas de Nassi-Schneiderman cuando se trata de estudiar algoritmos complejos.

1.4. Otros Paradigmas y Lenguajes

Desde el auge de la programación estructurada, fueron apareciendo también otros paradigmas de programación tales como Orientación a Objetos (el de mayor impulso desde hace algunos años), programación funcional, programación lógica, etc., con el fin de facilitar el proceso de implementación y disminuir los costos de desarrollo.

Simultáneamente fue apareciendo una inmensa cantidad de lenguajes de programación para cada uno de esos paradigmas, así como distintas metodologías de análisis y diseño específicas [Ghezzi91].

Todos esos lenguajes son textuales por naturaleza. En otras palabras, todo programa desarrollado en un

lenguaje de programación textual está compuesto de frases formadas por cadenas lineales de caracteres, los cuales son los símbolos de un alfabeto determinado.

Esa linealidad puede verse como una limitación de los lenguajes textuales, y se debe en gran medida a que la primera forma de comunicación entre computadoras y humanos que se pensó, estaba muy ligada a la forma de comunicación entre humanos. Es decir, a través de un lenguaje compuesto por frases lineales.

Por otro lado, refleja las limitaciones propias de los dispositivos de entrada/salida disponibles (los cuales, seguramente fueron diseñados bajo las mismas influencias).

Además, en general, es comparativamente más sencillo verificar la sintaxis de una notación textual.

Capítulo 2

Otra Forma de Programar

Aproximadamente en el año 1960 aparecen los primeros dispositivos de hardware gráficos.

En 1963, Ivan Sutherland desarrolló el primer programa de aplicación gráfico, al cual llamó Sketchpad [Sutherland63].

Tres años más tarde, William, hermano de Ivan, creó lo que puede ser considerado el primer lenguaje de programación visual [Sutherland66]. Se trata de un sistema basado en el paradigma de flujo de datos que permite a sus usuarios seleccionar operadores desde un menú y conectarlos por medio de líneas, para asignar de esa manera los valores de entrada al circuito. El paradigma de flujo de datos es brevemente explicado en la sección 2.1.

También en la década del 60 fueron creados otros tres lenguajes de programación visual. AMBIT/L [Christensen71] para reescribir representaciones gráficas de estructuras de listas, AMBIT/G [Christensen68] orientado a grafos generales. Ambos lenguajes están basados en "pattern-matching" bidimensional y reescritura de diagramas. El lenguaje GRAIL [Ellis69] usa la notación

del diagrama de flujo para la especificación visual de programas.

Desde entonces, paralelamente a la evolución del hardware (pantallas gráficas "bit-mapped", dispositivos de indicación como el mouse, etc., equipos que hoy son estándar), ha surgido una gran cantidad de lenguajes visuales. Algunos de ellos se enumeran en las tablas 2.1 y 2.2.

1966			
		R. Sutherland	[Sutherland66]
1968			
	Ambit/G y Ambit/L	Chistensen et.al.	[Christensen68]
1969			
	GRAIL	Ellis et.al.	[Ellis69]
1974			
	PLAN2D	Denert et.al.	[Denert74]
1975			
	Pygmailion	Smith	[Smith77]
1980			
	Outline	Lakin	[Lakin80]
1983			
	Prograph	Pietryzchowski	[Pietryzchowski83]
	ML-like VL	Cardelli	[Cardelli83]
1984			
	Pict	Glinert	[Glinert84]
	Programming by Rehearsal	Finzer & Gould	[Finzer84]

***Tabla 2.1**: Lenguajes visuales desarrollados entre 1966 y 1995*

1986			
	HI-VISUAL	Ichikawa	[Ichikawa86a] [Hirakawa90a]
	LabView		[LabView]
	PC-TILES	Glinert & Smith	[Glinert86a]
	Show & Tell	Kimura	[Kimura86c] [Kimura89]
	ThingLab	Borning	[Borning86]
	Tinkertoy	Edel	[Edel86]
1987			
	ARK	Smith	[Smith97]
1988			
	C^2	Kopache	[Kopache88]
	Fabrik	Ingalls	[Ingalls88]
1989			
	SunPICT	Glinert & McIntyre	[Glinert89]
1990			
	Cube	Najork	[Najork91] [Najork94]
	Hypersignal	Carlson	[Carlson94]
	Miro	Heydon	[Heydon90]
	NoPumpG	Lewis	
	Novis	Norton	[Norton90]
1991			
	Agentsheets	Repenning	[Agentsheets WEB]
	Forms/3	Burnett	[Burnett92]
	Hence 1.4	Beguelin	[Beguelin91]
	Mondrian	Lieberman	
	Visavis		

***Tabla 2.1**: Lenguajes visuales desarrollados entre 1966 y 1995 (Continuación)*

1992			
	ChemTrains	Bell	[Bell92]
	CODE 2.0	Newton	[Newton92]
	Hyperpascal	Lyons	[Lyons93]
	Iconicode		
	Vampire	McIntyre	[McIntyre92b]
	Visavis	Poswig	[Poswig92]
	Voice Dialog D.E.	Repenning & Summer	[Agentsheets WEB]
1993			
	MViews	Grundy & Hosking	[G&H93b]
	SPE	Grundy & Hosking	[G&H93a]
	MEANDER	Wirtz	[Wirtz93]
1994			
	Escalante	McWhirter	
	PhonePro	Cypress Research	[GACote94]
	Vipers	Mosconi	
	VIPR	Citrin & Zorn	
	WinPict	McIntyre	
1995			
	LEGOsheets	Repenning et.al.	[Agentsheets WEB]
	ViTABal	Grundy & Hosking	[Grundy95]

***Tabla 2.1**: Lenguajes visuales desarrollados entre 1966 y 1995 (Continuación)*

De propósito general		
Prograph	Picturius, Inc	800-927-4847
AppWare	Novell	800-277-2717
Iconicode	IconIcon	
Design/CPN	Meta Software	617-576-6920
SystemSpecs	IvyTeam, Bern Switz.	

***Tabla 2.2:** Lenguajes visuales comercialmente disponibles*

Layout	Objects, Inc	508-777-2800
LabView	National Instruments	512-794-0100
VPLus	SimPhonics, Inc	813-623-9917
N!Power	Signal Technology	805-899-8300 x350
EiffelBuild	ISE	info@eiffel.com
Basados en componentes		
Visual AppBuilder	Novell	800-453-1267
Capture	Metaphor / IBM	800-426-3333
SynchroWorks	Oberon Software, Inc	800-524-5459
Parts	Digitalk	800-531-2344
Synergy	Prodea Software Corp	800-PRODEA-1
VisualAge	IBM	800-426-3333
Eiffel libraries	ISE	info@eiffel.com
Herramientas multimedia y de creación de programas de entrenamiento basado en computadoras		
Authorware	Macromedia, Inc	800-945-4061
IconAuthor	AimTech Corp	800-289-2884
ForShow	Bourbaki, Inc	800-289-1347
HSC InterActive	HSC Software	800-566-6699
Telefonía		
PhonePro	Cypress Research	408-752-2700
PhoneOne	Information Gateway	703-760-0000
Adquisición de datos		
LabView	National Instruments	512-794-0100
DT VEE	Data Translation, Inc	800-525-8528
Análisis y visualización de datos		
Khoros	Khoral Research	505-837-6500
AVS	Advanced Visual Systems	617-890-4300
Diseño y testeo		
Dataflo MP	Dynetics, Inc	800-922-9261
Design/CPN	Meta Software	617-576-6920

Tabla 2.2: *Lenguajes visuales comercialmente disponibles (Continuación)*

Análisis y diseño en DSP		
Hypersignal	Hyperception	214-343-8525

Tabla 2.2: *Lenguajes visuales comercialmente disponibles (Continuación)*

2.1. Lenguajes Visuales

Hay una gran cantidad de definiciones acerca de qué es un lenguaje visual. Algunos autores hablan de lenguajes visuales, otros de programación visual, y aún otros de lenguajes de programación visual.

Si bien todos se refieren más o menos al mismo concepto, hay quienes prefieren hacer notar algunas pequeñas diferencias entre una denominación y otra. Por ejemplo, un lenguaje visual no necesariamente debe ser de programación.

Aquí, todos esos términos se utilizan indistintamente.

A continuación se reproducen las definiciones dadas por algunos de los autores e investigadores más reconocidos del tema.

- *Programación visual se refiere a cualquier sistema que permite al usuario especificar un programa en dos o más dimensiones. Los lenguajes textuales convencionales no son considerados bidimensionales ya que los compiladores o intérpretes los procesan como una gran secuencia unidimensional [Myers90].*
- *Un lenguaje visual manipula información visual o soporta interacción visual, o permite la programación*

con expresiones visuales. Esto último es tomado como la definición de un lenguaje de programación visual.

- *Los lenguajes de programación visual pueden ser mejor clasificados de acuerdo al tipo y al alcance de las expresiones visuales utilizadas, en lenguajes basados en íconos, lenguajes basados en formas y en lenguajes diagramáticos (o basados en diagramas).*
- *Los ambientes de programación visual proveen elementos gráficos o icónicos que pueden ser manipulados por el usuario en forma interactiva de acuerdo a alguna gramática espacial específica para la construcción de programas [Golin90b].*
- *La programación visual es comúnmente definida como el uso de expresiones visuales (tales como gráficos, dibujos, animaciones o íconos) en el proceso de programación. Esas expresiones visuales pueden ser usadas en ambientes de programación como interfaces gráficas para lenguajes de programación textuales; ellos pueden usados para formar la sintaxis de nuevos lenguajes de programación visual conduciendo a nuevos paradigmas tales como la programación por demostración; o pueden ser utilizados en la representación gráfica del comportamiento o estructura de un programa [McIntyre92a].*
- *Un lenguaje visual es un lenguaje de programación que utiliza predominantemente una notación gráfica [Najork94].*

Si bien los primeros trabajos se remontan a la década del 60, la mayoría de las investigaciones sobre lenguajes de programación visual se ha llevado a cabo en los últimos diez años, cuando el costo de las computadoras

con capacidades gráficas de alta resolución comenzó a ser accesible.

Con el tiempo, se han creado una considerable cantidad de paradigmas visuales. Aquí sólo se mencionan los dos de mayor interés: flujo de control y flujo de datos [Chang90] [Shu88].

El paradigma de flujo de control utiliza un diagrama visual del tipo diagrama de flujo para describir el flujo de control de un programa.

Los lenguajes visuales que siguen este paradigma están basados en el mismo modelo semántico de los lenguajes procedurales de la familia de Fortran, Algol, Pascal.

Las operaciones simples, tales como las primitivas, la asignación o la invocación a subprogramas, son representadas por cajas. La secuencia es denotada por arcos dirigidos que conectan dos cajas. Las estructuras de control comunes, como los condicionales o la iteración, también tienen su representación.

Pict [Glinert84] es un ejemplo de lenguajes de programación visual basados en el paradigma de flujo de control.

El paradigma de flujo de datos utiliza cajas para denotar funciones y arcos dirigidos para conectar las salidas de ciertas funciones con las entradas de otras.

Los lenguajes que siguen este paradigma están basados en el modelo semántico utilizado por los lenguajes de programación funcional.

Show&Tell [Kimura86] es un lenguaje típico del paradigma de flujo de datos.

2.2. Puntos a Favor de la Programación Visual

Hay muchos argumentos en favor de la programación visual. Generalmente están centrados en el hecho de que los humanos procesan, por su naturaleza, más fácil y rápidamente las imágenes que el texto. Es decir, adquieren más información en menos tiempo descubriendo las relaciones gráficas de una imagen que leyendo las letras de un texto [Raeder85].

Algunos de esos argumentos son enumerados a continuación:

- *El texto es estrictamente secuencial, mientras que las imágenes permiten acceso aleatorio a cualquiera de sus partes.*
- *Las imágenes permiten una visión global o detallada indistintamente, dependiendo de lo que espere encontrar el observador.*
- *El sistema sensorial humano está "diseñado" para el procesamiento de imágenes en tiempo real. Por esa razón, las imágenes pueden ser accedidas y decodificadas más rápidamente.*
- *El texto es unidimensional por naturaleza, mientras que las imágenes son multidimensionales.*
- *Las imágenes proveen un lenguaje mucho más rico a través de sus propiedades visuales, como los colores, las formas, los tamaños. Esto conduce a una*

codificación de la información significativamente más compacta que la del texto.

- *Las imágenes pueden capturar más fácilmente una idea abstracta.*
- *El texto es una secuencia de palabras y símbolos de puntuación. A su vez, cada palabra es una secuencia de letras. Y cada letra no es otra cosa que una pequeña imagen que representa sencillamente a un único símbolo de todo el texto.*

Es común el uso de imágenes o diagramas en la especificación de algoritmos y estructuras de datos que un programador o analista debe comunicar a otros.

Lo mismo sucede en muchas otras áreas. Por ejemplo, los ingenieros civiles utilizan, entre otros, diagramas de Grant, diagramas de Pert, organigramas, etc. Los arquitectos utilizan planos como una forma de modelar la realidad. Un plano detallado de un proyecto puede tener un tamaño considerable. Pero, ¿qué tamaño ocuparía toda esa información si tuviese que ser expresado en forma textual?

Los argumentos antes enumerados son absolutamente razonables, aunque lamentablemente sólo pueden ser verificados por medio de estudios empíricos.

Se han realizado distintos estudios con el objetivo de medir los beneficios de utilizar la programación visual para el desarrollo de programas [Pandey93]. A partir de ellos no se puede determinar contundentemente que, en un sentido general, es mejor programar en un lenguaje visual que en uno textual. Sin embargo, como mínimo, sugieren que para muchas tareas, un lenguaje de programación visual

adecuado es potencialmente mejor que cualquier lenguaje textual.

2.3. Puntos en Contra de la Programación Visual

Está claro que la programación visual tiene un gran potencial. Sin embargo, hay una serie de problemas aparejados a los lenguajes visuales existentes.

Algunos de esos problemas están originados por la notación visual empleada, y otros por las tendencias en el diseño de los lenguajes que más han influido hasta el momento.

Los lenguajes visuales tienden a utilizar una notación relativamente dispersa. Esto puede significar que utilizan más espacio de la pantalla que los lenguajes textuales.

Este problema puede ser aliviado utilizando el concepto de abstracción procedural. Es decir, representando subdiagramas con un único símbolo simple, y tratando a ese símbolo como a una "caja negra". Esto es análogo a la explosión e implosión de las burbujas de un diagrama de flujo de datos [Ghezzi91] [Yourdon93], o a la utilización de funciones como operadores simples, como es el caso del overloading de operadores que provee el lenguaje Ada [Olsen83], o incluso a la abstracción procedural de los lenguajes imperativos.

Muchos lenguajes visuales son interpretados y, en muchos casos, el intérprete opera directamente sobre las

representaciones de los componentes del diagrama. Obviamente, esto resulta en una eficiencia que deja mucho que desear.

La gran mayoría de los lenguajes visuales existentes son latentemente tipados, lo cual significa que la verificación de que un operador reciba valores de tipos adecuados sólo se puede realizar cuando dicho operador es aplicado, es decir, en tiempo de ejecución.

La razón de lo anterior es que gran parte de la investigación sobre lenguajes visuales estuvo centrada en el desarrollo de lenguajes orientados a no programadores.

Sin embargo, no hacer una verificación de tipos en tiempo de compilación significa que los errores que esto puede ocasionar sólo pueden ser descubiertos por el método de "prueba y error". Además, como ya fue expresado, los lenguajes latentemente tipados requieren de una comprobación de tipos en tiempo de ejecución, generando así un significativo "overhead".

2.4. Los Mal Llamados Visual

Visual BASIC, o más bien, la familia completa de herramientas visuales de Microsoft y muchos otros productos comerciales que incorporan la palabra "visual" a su nombre, no son realmente lenguajes de programación visual.

En realidad son lenguajes textuales que utilizan un constructor de interfaces gráficas que, en todo caso, puede

verse como una herramienta de generación de interfaces por demostración, ya que las mismas se programan demostrando cómo deben ser.

La demostración de la interfaz de una aplicación, es luego traducida automáticamente a código textual. En algunos casos, el código generado puede ser accedido y modificado por el programador, y en otros, ni siquiera eso.

Capítulo 3

Compiladores e Intérpretes

Un programa escrito en un lenguaje de alto nivel debe ser traducido a un programa en código de máquina para poder ser ejecutado en una computadora. Esa traducción es a su vez llevada a cabo por un programa.

Los programas encargados de la traducción de cualquier programa escrito en un lenguaje de alto nivel a otro programa equivalente en algún lenguaje distinto, se denomina "compilador" [Hunter85] [Ghezzi87].

La compilación de un programa consiste del análisis, el cual determina el efecto deseado del programa, seguido de la síntesis, que genera el programa en código máquina equivalente.

En la etapa de análisis, un compilador debe ser capaz de detectar si el programa de entrada es inválido en algún sentido, y si es así, retornar un mensaje apropiado al programador. Este aspecto de la compilación es denominado "detección de error".

En vez de traducir cada programa en el código máquina equivalente y luego ejecutarlo, una aproximación alternativa es primero traducir el programa a un lenguaje

intermedio y luego traducir y ejecutar cada sentencia en lenguaje intermedio, a medida que es encontrada.

El programa que traduce y ejecuta un programa escrito en un lenguaje de alto nivel es llamado "intérprete".

Los intérpretes tienen ciertas ventajas y desventajas respecto de los compiladores.

Las ventajas más importantes son:

- *Generalmente es más sencillo comunicar al usuario los mensajes de error en términos del problema original.*
- *La versión en lenguaje intermedio de un programa, generalmente es más compacta que el código máquina generado por un compilador.*
- *La alteración de una parte del programa, no necesariamente involucra la recompilación del programa completo.*

Y las principales desventajas son:

- *Los programas interpretados tienden a ser ineficientes en cuanto a la velocidad de ejecución, debido a que cada sentencia en lenguaje intermedio debe ser traducida cada vez que se ejecute.*
- *Los intérpretes no generan programas ejecutables, lo que dificulta la distribución. Algunos lenguajes interpretados intentan solucionar este problema creando un programa ejecutable compuesto por el código en lenguaje intermedio y el intérprete mismo. Otros logran algo similar poniendo el intérprete en una librería de enlace dinámico (DLL, Dynamic Link Library).*

Sin embargo, muchas de las ideas utilizadas para la construcción de compiladores son las mismas que se usan para la construcción de intérpretes.

Volviendo a la etapa de análisis de la compilación, una forma conveniente de representar la estructura de un programa, es por medio de un árbol, usualmente llamado árbol de parse. Por ejemplo, la figura 3.1 muestra el árbol de parse correspondiente al siguiente segmento de código Pascal:

```
begin
  x := 3;
  write(x)
end
```

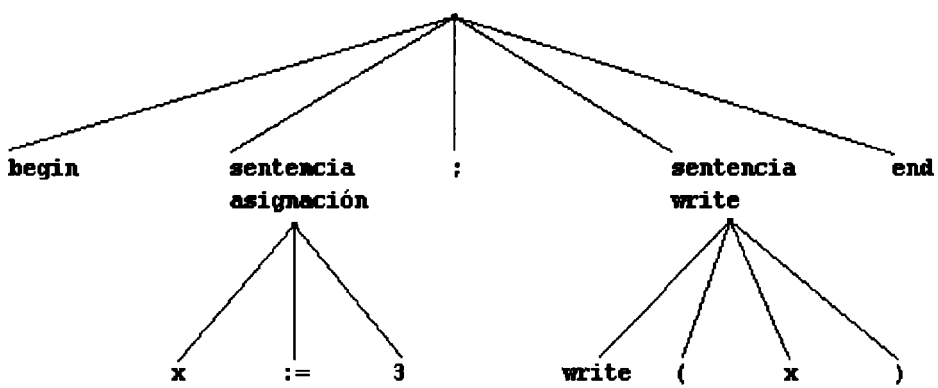


***Figura 3.1**: Arbol de Parse*

Se puede pensar que el analizador construye el árbol y el sintetizador lo recorre para generar código máquina.

Normalmente la primer fase del proceso de análisis consiste en agrupar caracteres en lo que usualmente se

conoce como símbolos; por ejemplo, write, :=, begin, end. Esta fase de la compilación es denominada análisis léxico. La parte restante del análisis, la que construye el árbol, es llamada análisis sintáctico o "parsing".

Para verificar ciertos requerimientos del lenguaje, generalmente los compiladores construyen tablas.

Por ejemplo, en la mayoría de los lenguajes, la instrucción `x := y` *es legal sólo si* `x` *e* `y` *están declaradas y son de tipos adecuados. A partir de la declaración* `x: integer`*, se inserta una entrada en una tabla, llamada tabla de símbolos, indicando el tipo de la variable* `x`*. Normalmente, la tabla de símbolos es construida durante el análisis sintáctico y son requeridas durante la generación de código y la verificación de validez del programa.*

Muchos compiladores, durante el análisis léxico, reemplazan los identificadores de longitud variable por símbolos de longitud fija, manteniendo la correspondencia entre estos símbolos y los identificadores originales en una tabla de identificadores.

Los compiladores de lenguajes que permiten definiciones de tipos complejos, pueden necesitar una tabla similar a la anterior pero de tipos.

Parte 2

Visual DaVinci

Hace algunos años, un grupo de investigadores del LIDI, desarrollaron la especificación de una máquina abstracta (el robot Lubo-I) y la de un lenguaje de programación asociado a la misma, ambas destinadas a la enseñanza de los conceptos introductorios de la programación estructurada [De Giusti88][De Giusti89].

Esas especificaciones se están utilizando desde aquel entonces en el curso de ingreso a la carrera de Informática y en la materia curricular Programación de Computadoras del primer año de la misma carrera.

¿Cuán beneficioso es que el estudiante cuente con un ambiente de desarrollo en el cual especificar y probar sus programas? ¿Es conveniente una presentación temprana de nuevas tecnologías, tales como la programación visual? ¿Qué ventajas tiene un ambiente visual sobre uno textual en la enseñanza de programación? Estos interrogantes son los que motivaron la implementación del Visual DaVinci.

Capítulo 4

Especificaciones del Robot Lubo-1

Este capítulo describe la especificación de la máquina abstracta anteriormente mencionada.

Se trata de un robot virtual que realiza ciertas acciones sobre una ciudad, determinadas por la ejecución de un programa.

El lenguaje con el cual se implementan los programas que ejecuta el robot, se presenta en el capítulo 5.

4.1. La Ciudad

En la ejecución de un programa, el robot circula por una ciudad.

Dicha ciudad tiene cien calles por cien avenidas. Las calles son horizontales mientras que las avenidas son verticales.

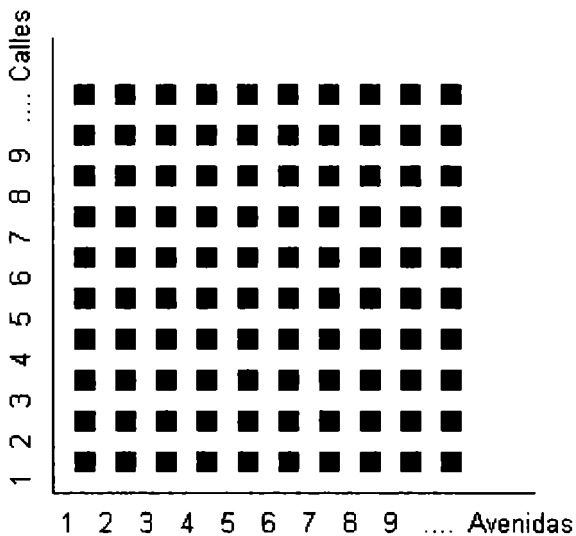


Figura 4.1: *Ciudad*

Como resulta obvio, las esquinas quedan determinadas por las intersecciones de las calles con las avenidas. Cada esquina es identificada por un par ordenado (Av, Ca), donde Av y Ca son números enteros en el rango 1..100.

Además, es posible que en cada esquina haya un número cualquiera de flores y un número cualquiera de papeles, los cuales pueden ser juntados por el robot. A su vez, el robot es capaz de depositar flores y/o papeles, como se explica en la siguiente sección.

También puede haber obstáculos que impiden el paso del robot por una esquina, debiendo entonces desviarse de su trayectoria normal, para encontrar un camino alternativo. Existen obstáculos simples o barreras.

Un obstáculo es simple si no hay otro obstáculo en ninguna de las esquinas adyacentes. Un conjunto de obstáculos conforma una barrera si estos están dispuestos en línea recta sobre esquinas consecutivas.

4.2. Lubo-I

El robot Lubo-I es una máquina abstracta que ejecuta un programa. Para ello conoce la forma en la que debe ejecutar un conjunto de instrucciones.

Inicialmente, el robot es posicionado en la esquina determinada por el par (1, 1) orientado hacia el norte.

La orientación del robot puede ser modificada de forma tal que "quede mirando" hacia uno de los cuatro puntos cardinales.

Cada movimiento del robot, conduce a este de una esquina de la ciudad a la siguiente, según la dirección hacia la cual se encuentre orientado.

El robot lleva consigo dos bolsas: una de flores y otra de papeles. Cada vez que junta una flor, la guarda en su bolsa de flores. Asimismo, cada papel que junta lo guarda en su bolsa de papeles.

También puede depositar flores y/o papeles en la esquina en la que se encuentra, siempre y cuando la bolsa respectiva no se encuentre vacía.

Como en cada esquina puede haber un obstáculo, el robot debe tener la posibilidad de verlo desde la esquina anterior, de manera de prevenir una posible "colisión".

Si el robot intenta avanzar sobre una esquina en la que hay un obstáculo, se producirá un error en tiempo de ejecución.

Lo mismo sucederá si trata de avanzar hacia afuera de los límites de la ciudad.

También ocurrirá un error en tiempo de ejecución si se pretende juntar una flor o un papel en una esquina en la que no hay ninguno de tales elementos, o si se intenta depositar un elemento cuya respectiva bolsa se encuentra vacía.

Capítulo 5

El Lenguaje de Programación del Robot Lubo-I

Este capítulo describe las especificaciones del lenguaje de programación del robot Lubo-I.

El lenguaje está compuesto por palabras claves, primitivas, sentencias simples, sentencias compuestas o estructuras de control, expresiones y constructores de subprogramas. La sintaxis exacta de cada uno de estos elementos se especifica en la sección 5.7.

5.1. Estructura General de un Programa

Todo programa está compuesto por tres partes: encabezamiento, declaraciones y cuerpo.

El encabezamiento comienza con la palabra clave `programa`*, la cual debe estar seguida por un identificador que determina el nombre del programa.*

En la segunda parte, se declaran los subprogramas y las variables que utilizará el programa principal. El orden de estas declaraciones es importante. No se permiten declaraciones de variables anteriores a subprogramas, de forma tal que no exista la posibilidad de hacer referencia a variables globales desde un subprograma.

La declaración de subprogramas comienza con la palabra clave `procesos`*, y termina donde se encuentra la palabra clave* `variables` *o bien la palabra clave* `comenzar`.

Por otro lado, la declaración de las variables comienza con la palabra clave `variables` *y termina donde se encuentra la palabra clave* `comenzar`.

El cuerpo del programa principal es una secuencia de sentencias, delimitada por las palabras clave `comenzar` *y* `fin`. *Los cuerpos de los subprogramas siguen la misma regla, como se verá en la siguiente sección.*

5.2. Subprogramas

Los subprogramas, llamados procesos en Visual DaVinci, tienen una estructura casi idéntica a la del programa principal.

Cada subprograma está dividido en las mismas tres partes que un programa; es decir, encabezamiento, declaraciones y cuerpo.

Las declaraciones de procesos locales y variables locales siguen exactamente las mismas reglas que las declaraciones del programa principal.

Lo mismo sucede con el cuerpo del subprograma.

La única diferencia en la especificación de un subprograma y la del programa principal es que el primero puede definir parámetros formales en su encabezamiento. La forma en que esto es llevado a cabo se discute en la sección 5.4.

5.3. Declaración de Variables

La palabra clave `variables` *indica el comienzo de la sección de declaraciones de variables del programa o de un subprograma, la que llega hasta la palabra clave* `comenzar` *del cuerpo de la unidad correspondiente.*

Cada variable tiene un nombre y un tipo. Los tipos posibles son solamente dos: `numero` *y* `boolean`.

El hecho de que el conjunto de tipos posibles sea reducido a sólo dos elementos es una restricción importante del lenguaje. Sin embargo, el objetivo de este lenguaje es la enseñanza de programación estructurada. Es por eso que se pone atención principalmente en los aspectos relacionados con los algoritmos y no tanto en lo que tiene que ver con los tipos y estructuras de datos.

Como en todo lenguaje de programación una variable puede ser asignada con un valor del tipo correspondiente, y puede intervenir en cualquier expresión.

La visibilidad de una variable está limitada al cuerpo del subprograma que la define.

Desde el punto de vista del usuario, el alcance es igual a la visibilidad; pero, como se describe en el capítulo 9, desde el punto de vista de la implementación del Visual DaVinci, hay algunas consideraciones que se deben tener en cuenta.

5.4. Parámetros Formales

Los parámetros formales de un subprograma son definidos entre paréntesis en el encabezamiento de este.

Cada parámetro formal tiene un nombre, un tipo y un modo.

El nombre de un parámetro lo identifica dentro del subprograma que lo define. Su alcance y su visibilidad son los mismos que los de una variable local.

Los tipos posibles de un parámetro son, al igual que para las variables, `numero` *y* `boolean`.

Los modos de pasaje de parámetro que se pueden utilizar son: entrada, salida y entrada/salida indicados respectivamente por `E`, `S` *y* `ES`.

Un parámetro formal de entrada actúa como una constante local. Es decir, puede intervenir en cualquier expresión, pero no puede ser modificado; no puede estar a la izquierda de una asignación.

Por el contrario, la única operación permitida sobre un parámetro de salida es asignarle un valor. No puede participar en una expresión; sólo puede aparecer a la izquierda de una asignación.

Un parámetro de entrada/salida es equivalente a una variable local. Puede aparecer en cualquier lugar que pueda hacerlo una variable.

El pasaje de parámetros en Visual DaVinci es conceptualmente igual al del lenguaje Ada [USDoD83]. Fue elegido este modelo de pasaje de parámetros debido a que resulta más representativo que otros de los distintos modos usuales en los lenguajes de programación convencionales.

6.5. Sentencias

Como ya se ha dicho, el cuerpo de cada módulo, programa o proceso, es una secuencia de sentencias.

El conjunto de las sentencias está dividido en tres subconjuntos: primitivas, sentencias simples y sentencias compuestas.

Las primitivas son aquellas instrucciones directamente ejecutables por el robot. Ellas son: `iniciar`, `derecha`, `mover`, `tomarFlor`, `tomarPapel`, `depositarFlor`, `depositarPapel`.

Las sentencias simples son la asignación y la invocación a procesos definidos por el usuario o a procesos del sistema (`Pos`, `Informar`*).*

Para los procesos con parámetros, la invocación debe incluir los respectivos parámetros reales. La correspondencia entre parámetros formales y reales es determinada posicionalmente.

Las sentencias compuestas son las estructuras de control `mientras`, `repetir`, `si` *y* `si/sino`.

El `mientras` *es una estructura iterativa condicional. Contiene una expresión booleana y una secuencia de sentencias, la cual se ejecuta mientras el resultado de evaluar la condición es verdadero.*

El `repetir` *es también una estructura iterativa, pero incondicional. Tiene una expresión aritmética y una secuencia de sentencias. El resultado de la evaluación de la expresión aritmética determina la cantidad de veces que se repite la ejecución del cuerpo del* `repetir`.

La estructura `si` *tiene una condición booleana y una o dos secuencias de sentencias. Si tiene dos, ambas se encuentran separadas por la palabra clave* `sino`. *La primer secuencia se ejecuta sólo si se cumple la condición. Si la condición resulta falsa y el* `si` *tiene una segunda secuencia de sentencias (la secuencia del* `sino`*), entonces es esta la que se ejecuta.*

5.6. Expresiones

Las expresiones que intervienen en un programa Visual DaVinci pueden ser lógicas o aritméticas.

Se construyen relacionando variables definidas por el usuario, variables del sistema, valores lógicos y numéricos, y parámetros de entrada y entrada/salida, por medio de operadores lógicos, aritméticos y/o relacionales.

Un valor, una variable o un parámetro son también considerados expresiones.

Las variables del sistema son: `HayFlorEnLaEsquina`, `HayPapelEnLaEsquina`, `HayFlorEnLaBolsa`, `HayPapelEnLaBolsa`, `HayObstaculo`, `PosAv`, `PosCa`.

Los operadores aritméticos son: `+` *(suma),* `-` *(resta),* `*` *(multiplicación) y* `/` *(división). Los booleanos:* `~` *(not),* `&` *(and) y* `|` *(or). Y los relacionales:* `=` *(igual),* `<>` *(distinto),* `>` *(mayor),* `<` *(menor),* `>=` *(mayor o igual) y* `<=` *(menor o igual).*

5.7. Sintaxis

En esta sección se describen las reglas sintácticas del lenguaje por medio de una notación del estilo de la de Backus, en la que `<X>` *indica que* `X` *es un símbolo no terminal,* `[X]` *que* `X` *no es obligatorio,* `{X}` *que pueden aparecer cero o más repeticiones de* `X`, `X|Y` *es la selección de uno de* `X` *o* `Y`. *Los símbolos en negrita son símbolos terminales. El símbolo* `__` *representa la indentación de dos espacios, obligatoria en muchos casos.*

```
<Programa>::=
     <Encabezamiento de programa>
     [<Declaraciones>]
     <Cuerpo>

<Encabezamiento de programa>::=
     programa <Ident. de programa>

<Ident. programa> ::=
     <Identificador>

<Identificador>::=
     (A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U
     |V|W|X|Y|Z|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p
     |q|r|s|t|u|v|w|x|y|z){A|B|C|D|E|F|G|H|I|J|
     K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e|
     f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z}

<Declaraciones>::=
     [<Declaración de procesos>]
     [<Declaración de variables>]

<Declaración de procesos>::=
     procesos
     __{<Declaración de un proceso>}

<Declaración de un proceso>::=
     <Encabezamiento de proceso>
     [<Declaraciones>]
     <Cuerpo>

<Encabezamiento de proceso>::=
     proceso <Ident.proceso>[(<Parámetros
     formales>)]

<Ident.proceso>::=
     <Identificador>
```

```
<Parámetros formales>::=
     {<Parámetro formal>, }<Parámetro formal>

<Parámetro formal>::=
     <Parám. Formal de entrada>|
     <Parám. Formal de salida>|
     <Parám. Formal de entr./sal.>

<Parám. formal de entrada>::=
     E <Ident. parám. formal entr.>: <Tipo>

<Ident. parám. formal entr.> ::=
     <Identificador>

<Parám. formal de salida>::=
     S <Ident. parám. formal sal.>: <Tipo>

<Ident. parám. formal sal.> ::=
     <Identificador>

<Parám. formal de entr./sal.> ::=
     ES <Ident. parám. formal entr./sal.>:
     <Tipo>

<Ident. parám. formal entr./sal.> ::=
     <Identificador>

<Tipo>::=
     numero|boolean

<Declaración de variables>::=
     variables
     __<Declaración de una variable>
     __{<Declaración de una variable>}


<Declaración de una variable>
     <Ident. variable>: <Tipo>
```

```
<Ident. variable>::=
     <Identificador>

<Cuerpo>::=
     comenzar
     __{<Secuencia de sentencias>}
     fin

<Secuencia de sentencias>::=
     <Sentencia>
     {<Sentencia>}

<Sentencia>::=
     <Primitiva>|<Sentencia simple>|<Sentencia
     compuesta>

<Primitiva>::=
     iniciar|mover|derecha|tomarFlor|tomarPapel
     |depositarFlor|depositarPapel

<Sentencia simple>::=
     <Asignación>|<Invocación>

<Asignación>::=
     (<Ident. variable>|
     <Ident. parám. formal sal.>|
     <Ident. parám formal entr./sal.>) :=
     <Expresión>

<Expresión>::=
     {(<Valor>|<Ident. variable>|
     <Ident. var. sistema>|
     <Ident. parám. formal entr.>|
     <Ident. Parám  forma entr./sal.>|
     (<Expresión>)|-<Expresión>|
     ~<Expresión>) <Operador>}
     (<Valor>|<Ident. variable>|
```

```
        <Ident. var. sistema>|
        <Ident. parám. formal entr.>|
        <Ident. parám. formal entr./sal.>|
        (<Expresión>)|-<Expresión>|~<Expresión>)

<Valor>::=
        <Valor numérico>|<Valor booleano>

<Valor numérico>::=
        0|1|2|3|4|5|6|7|8|9{0|1|2|3|4|5|6|7|8|9}
        [.0|1|2|3|4|5|6|7|8|9{0|1|2|3|4|5|6|7|8|9}
        ]

<Valor booleano>::=
        V|F

<Ident. var. sistema>::=
        PosAv|PosCa|HayFlorEnLaEsquina|
        HayFlorEnLaBolsa|HayPapelEnLaEsquina|
        HayPapelEnLaBolsa

<Operador>::=
        <Operador aritmético>|<Operador booleano>|
        <Operador relacional>

<Operador aritmético>::=
        +|-|*|/

<Operador booleano>::=
        ~|&||

<Operador relacional>::=
        =|<>|<|>|<=|>=

<Invocación>::=
        (<Ident. proceso>|<Ident. proc. sistema>)
        [<Parámetros reales>]
```

```
<Ident. proc. sistema>::=
     Pos|Informar

<Parámetros reales>::=
     <Parám. real entr.>|<Parám.realsal.>|
     <Parám. real entr./sal.>

<Parám. real entr.>::=
     <Expresión>

<Parám. real sal.> ::=
     <Ident. variable>|
     <Ident. parám. formal sal.>|
     <Ident. parám. formal entr./sal.>

<Parám. real entr./sal.> ::=
     <Ident. variable>|
     <Ident. parám. formal entr./sal.>

<Sentencia compuesta>::=
     <Selección>|<Iteración condicional>|
     <Iteración incondicional>

<Selección>::=
     si <Expresión>
     __<Secuencia de sentencias>
     [sino
      __<Secuencia de sentencias>]

<Iteración condicional>::=
     mientras <Expresión>
     __<Secuencia de sentencias>]

<Iteración incondicional>::=
     repetir <Expresión>
     __<Secuencia de sentencias>
```

5.8. Semántica

`programa`	*Indica el comienzo de la especificación de un programa. Debe estar seguido por un identificador que da el nombre al programa.*
`procesos`	*Indica el comienzo de la sección de declaraciones de procesos.*
`variables`	*Indica el comienzo de la sección de declaraciones de variables.*
`comenzar`	*Indica el comienzo de la secuencia de sentencias que componen el cuerpo del programa principal o de un proceso.*
`fin`	*Indica el final de la secuencia de sentencias que componen el cuerpo del programa principal o de un proceso.*
`proceso`	*Indica el comienzo de la definición de un proceso. Debe estar seguido de un identificador que da el nombre al proceso.*
`E`	*Indicador de modo de pasaje de parámetro para parámetros de entrada.*

S	*Indicador de modo de pasaje de parámetro para parámetros de salida.*
ES	*Indicador de modo de pasaje de parámetro para parámetros de entrada/salida.*
numero	*Nombre del conjunto de todos los valores numéricos.*
boolean	*Nombre del conjunto de los valores lógicos.*
mientras	*Indica el comienzo de una estructura de control iterativa condicional. Debe estar seguido por una condición (expresión booleana) y una secuencia de sentencias. Esta secuencia es ejecutada mientras se cumpla la condición. Esto significa que será ejecutada cero o más veces.*
repetir	*Indica el comienzo de una estructura de control iterativa incondicional. Debe estar seguido por una expresión aritmética y una secuencia de sentencias. Esta secuencia es ejecutada una cantidad fija de veces, la cual es determinada por el resultado de la evaluación de la expresión.*

`si` — *Indica el comienzo de una estructura de control condicional. Debe estar seguida de una condición (expresión booleana) y una secuencia de sentencias. Esta secuencia se ejecuta solamente si se cumple la condición. Puede estar acompañado por un* `sino` *y otra secuencia de sentencias. En este caso, cuando la condición del* `si` *no se cumple, se ejecuta esta última secuencia.*

`iniciar` — *Instrucción primitiva que posiciona al robot en la esquina (1, 1) orientado hacia el norte, e inicializa las variables del sistema.*

`derecha` — *Instrucción primitiva que cambia la orientación del robot hacia el punto cardinal que difiere en 90° en sentido horario respecto de la orientación actual.*

`mover` — *Instrucción primitiva que conduce al robot de la esquina en la que se encuentra a la siguiente, en la dirección hacia la cual está orientado. Es responsabilidad del programador que esta instrucción sea ejecutada sólo cuando no haya un obstáculo en la esquina destino. En caso contrario se producirá un error en tiempo de ejecución y el programa será abortado.*

`TomarFlor`	*Instrucción primitiva que junta una flor de la esquina en la que se encuentra el robot y la pone en la bolsa de flores del mismo. Es responsabilidad del programador que esta instrucción sea ejecutada sólo cuando haya al menos una flor en dicha esquina. En caso contrario se producirá un error en tiempo de ejecución y el programa será abortado.*
`TomarPapel`	*Instrucción primitiva que junta un papel de la esquina en la que se encuentra el robot y lo pone en la bolsa de papeles del mismo. Es responsabilidad del programador que esta instrucción sea ejecutada sólo cuando haya al menos un papel en dicha esquina. En caso contrario se producirá un error en tiempo de ejecución y el programa será abortado.*
`DepositarFlor`	*Instrucción primitiva que saca una flor de la bolsa de flores del robot y la deposita en la esquina en la que este está posicionado. Es responsabilidad del programador que esta instrucción sea ejecutada sólo cuando haya al menos una flor en dicha bolsa. En caso contrario se producirá un error en tiempo de ejecución y el programa será abortado.*

`DepositarPapel`	*Instrucción primitiva que saca un papel de la bolsa de papeles del robot y lo deposita en la esquina en la que este está posicionado. Es responsabilidad del programador que esta instrucción sea ejecutada sólo cuando haya al menos un papel en dicha bolsa. En caso contrario se producirá un error en tiempo de ejecución y el programa será abortado.*
`PosAv`	*Variable del sistema, cuyo valor es un número entero en el rango 1..100, que indica la avenida en la que el robot está actualmente posicionado. Su valor no puede ser modificado por el programador.*
`PosCa`	*Variable del sistema, cuyo valor es un número entero en el rango 1..100, que indica la calle en la que el robot está actualmente posicionado. Su valor no puede ser modificado por el programador.*
`HayFlorEnLaEsquina`	*Variable del sistema, cuyo valor es* `V` *si hay al menos una flor en la esquina en la que el robot está actualmente posicionado, o* `F` *en caso contrario. Su valor no puede ser modificado por el programador.*

`HayPapelEnLaEsquina`	*Variable del sistema, cuyo valor es* `V` *si hay al menos un papel en la esquina en la que el robot está actualmente posicionado, o* `F` *en caso contrario. Su valor no puede ser modificado por el programador.*
`HayFlorEnLaBolsa`	*Variable del sistema, cuyo valor es* `V` *si hay al menos una flor en la bolsa de flores del robot, o* `F` *en caso contrario. Su valor no puede ser modificado por el programador.*
`HayPapelEnLaBolsa`	*Variable del sistema, cuyo valor es* `V` *si hay al menos un papel en la bolsa de papeles del robot, o* `F` *en caso contrario. Su valor no puede ser modificado por el programador.*
`HayObstaculo`	*Variable del sistema, cuyo valor es* `V` *si hay un obstáculo en la esquina siguiente a la que está posicionado actualmente el robot, o* `F` *en caso contrario. Su valor no puede ser modificado por el programador.*
`Pos`	*Proceso del sistema que recibe dos parámetros de entrada Av y Ca, cada uno de ellos en el rango 1..100, y posiciona al robot en la esquina determinada por el par (Av, Ca).*
`Informar`	*Proceso del sistema que recibe una cantidad cualquiera de parámetros de entrada, y muestra en pantalla sus valores.*

5.9. Estilo de Programación

Se imponen ciertas reglas sintácticas adicionales que no son muchas, pero sí estrictas, las cuales se ocupan únicamente de la indentación y el uso de mayúsculas y minúsculas.

Aunque pueden resultar algo incómodas para el programador, esas reglas intentan formar un buen estilo de codificación. Si el término "buen estilo" resultara discutible, en todo caso se podría decir "un estilo". Lo importante aquí es presentar al estudiante algunos criterios a tener en cuenta en el momento de crear su propio estilo.

El objetivo de un estilo de programación es mejorar en todo lo posible la legibilidad del código de forma tal que resulte sencillo entenderlo, modificarlo, adaptarlo y reusarlo, ayudando así a maximizar la productividad y minimizar el costo de desarrollo y mantenimiento.

En el código textual se deben respetar las siguientes reglas de indentación:

- *La palabra clave programa debe comenzar en la primer columna.*
- *Las palabras clave procesos, variables, comenzar y fin de un programa deben comenzar en la misma columna que la palabra clave programa.*
- *Las palabras clave procesos, variables, comenzar y fin de un proceso deben comenzar en la misma columna que la palabra clave proceso.*

- *La definición de un proceso debe comenzar dos columnas más a la derecha que la palabra clave procesos que indica el comienzo de la sección de declaraciones de procesos.*
- *Las declaraciones de variables deben comenzar dos columnas más a la derecha que la palabra clave variables que indica el comienzo de la sección de declaraciones de variables.*
- *Las sentencias de los cuerpos del programa y de los procesos deben comenzar dos columnas más a la derecha que las palabras clave comenzar y fin que delimitan a esos cuerpos.*
- *Las sentencias que pertenecen al cuerpo de una estructura de control deben comenzar dos columnas más a la derecha que la palabra clave que identifica a la estructura de control.*
- *La indentación es la única forma de indicar si una sentencia está dentro de una estructura de control o no. Esta forma de indentación está inspirada en la utilizada por el lenguaje OCCAM [CSA90].*

Por otro lado, todas las palabras claves definidas por el lenguaje, así como las primitivas, las estructuras de control y los tipos, deben ser escritas siempre con letras minúsculas, excepto que su nombre esté compuesto por más de una palabra, en cuyo caso, de la segunda palabra en adelante, cada una comienza con mayúscula. Por ejemplo, `iniciar`, `tomarFlor`, `mientras`, `numero`.

Por el contrario, las variables del sistemas y los procesos del sistema deben comenzar cada palabra que compone su nombre con una letra mayúscula y las demás

minúsculas. Por ejemplo, `PosAv`, `HayFlorEnLaBolsa`, `Pos`, `Informar`.

Los indicadores de modo de pasaje de parámetros (`E`, `S`, `ES`*) van siempre en mayúsculas.*

Se sugiere además, que los identificadores definidos por el usuario sigan las convenciones de las variables y procesos del sistema.

El usuario tiene absoluta libertad en el uso de mayúsculas y minúsculas cuando define un identificador. Sin embargo, cada vez que se haga referencia a un identificador definido por el usuario, este debe ser escrito exactamente igual. En otras palabras, Visual DaVinci es "case sensitive". Con este criterio, los identificadores `cantidadFlores`, `CantidadFlores`, `cantidadflores` *y* `Cantidadflores` *son todos distintos.*

Esta sensibilidad al tipo de letra es la misma que la del lenguaje C [Kernighan88].

También se recomienda la utilización frecuente de comentarios lógicos que aclaren ciertos algoritmos, indiquen la utilidad de las variables y procesos utilizados, el objetivo de los parámetros, expliquen los motivos de algún compromiso tomado, etc.

5.10. Ejemplo

El siguiente es un ejemplo de un programa desarrollado en forma textual en Visual DaVinci.

En el mismo se recorren 10 cuadrados concéntricos, comenzando en el cuadrado determinado por las esquinas (1, 1) y (20, 20) hasta el cuadrado determinado por las esquinas (10, 10) y (11, 11), juntando y contando todas las flores que se encuentren en el camino. Al finalizar el recorrido, se informa la cantidad de flores encontradas.

```
{ Recorre 10 cuadrados concéntricos, comenzando por el }
{ cuadrado determinado por las esquinas (1, 1) y       }
{ (20, 20), hasta llegar al determinado por las squinas}
{ (10, 10) y (11, 11), juntando y contando todas las   }
{ flores que encuentra en su camino. Al finalizar el   }
{ recorrido informa la cantidad de flores juntadas.    }

programa Concentricos

procesos
  { En los siguientes dos procesos se utiliza un       }
  { parámetro de entrada/salida. Ese parámetro podria  }
  { haber sido de sólo salida.                         }
  { Pero asi se evita tener que utilizar variables     }
  { locales auxiliares.                                }

  proceso HacerLado(E Largo: numero;
                    ES CantFlores: numero)
    { Se encarga de recorrer Largo cuadras en la       }
    {  dirección hacia la cual está orientado, juntando}
    { todas las flores que encuentra en su camino. Por }
    { cada flor que junta incrementa en 1 el parámetro }
    { de entrada/salida CantFlores.                    }
  comenzar
```

```
    repetir Largo
      mover
      mientras HayFlorEnLaEsquina
        tomarFlor
        CantFlores := CantFlores + 1
  fin

  proceso HacerCuadrado(E Tamano: numero;
                        ES Flores: numero)
    { Se encarga de recorrer un cuadrado de lado       }
    { Tamano, juntando las flores que encuentra en su  }
    { camino.                                          }
    { Por cada flor que junta incrementa en 1 el       }
    { parámetro de entrada/salida Flores.              }
  comenzar
    repetir 4
      HacerLado(Tamano, Flores)
      derecha
  fin

variables
  Lado: numero
  Flores: numero

comenzar
  iniciar
  Lado := 19
  Flores := 0
  repetir 10
    HacerCuadrado(Lado, Flores)
    { El siguiente cuadrado a recorrer tiene 2 cuadras }
    { menos de lado que el cuadrado recientemente      }
    { recorrido.                                       }
    Lado := Lado - 2
    { El próximo cuadrado comienza una avenida más a la}
    { derecha y una calle más arriba que el            }
    { recientemente recorrido.                         }
    Pos(PosAv + 1, PosCa + 1)
  Informar(Flores)
fin
```

Capítulo 6

El Ambiente de Desarrollo

El ambiente de desarrollo de Visual DaVinci está compuesto por cinco ventanas: la ventana principal, el editor de diagramas visuales, el editor de código textual, la ciudad y el inspector de variables del sistema.

6.1. Ventana Principal

La ventana principal es la que se muestra en la figura 6.1.

Su objetivo es la asistencia en la manipulación global de archivos de programa y en la edición de código. Es también la que provee las opciones de verificación de sintaxis, ejecución y depuración.

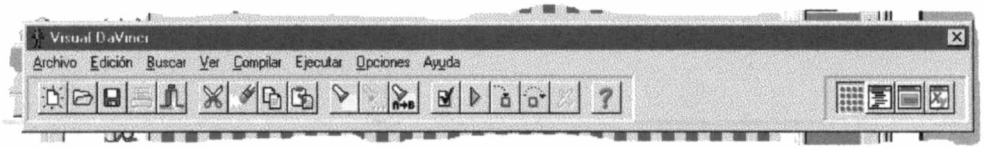


***Figura 6.1**: Ventana Principal*

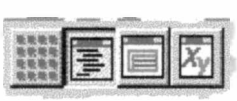
Cuenta con un menú y dos barras de botones. Todas las acciones que se pueden realizar por medio de los botones de la primera barra (situada en el lado izquierdo), pueden ser también realizadas utilizando las opciones correspondientes del menú.

Esas acciones incluyen:

Crear un archivo de programa nuevo. Abrir un programa existente. Guardar el programa corriente. Imprimir el programa corriente. Salir.

Estas son opciones de edición de código y se describen en la sección 6.3.

Verificar la sintaxis del programa corriente. Ejecutar el programa corriente. Ejecutar una única instrucción. Si se trata de la invocación a un proceso definido por el usuario, es posible ejecutar el proceso invocado en forma completa o entrar en su cuerpo. Abortar la ejecución del programa corrientemente ejecutado.



La barra de botones de la derecha permite cambiar el foco, de la ventana corriente a otra. Por eso tiene cuatro botones correspondientes a las ventanas de la ciudad, del editor de código, del editor de diagramas y del inspector de variables del sistema.

6.2. Editor de Diagramas

La ventana del editor de diagramas, que se muestra en la figura 6.2, es la que se utiliza para la especificación de un programa en forma visual.

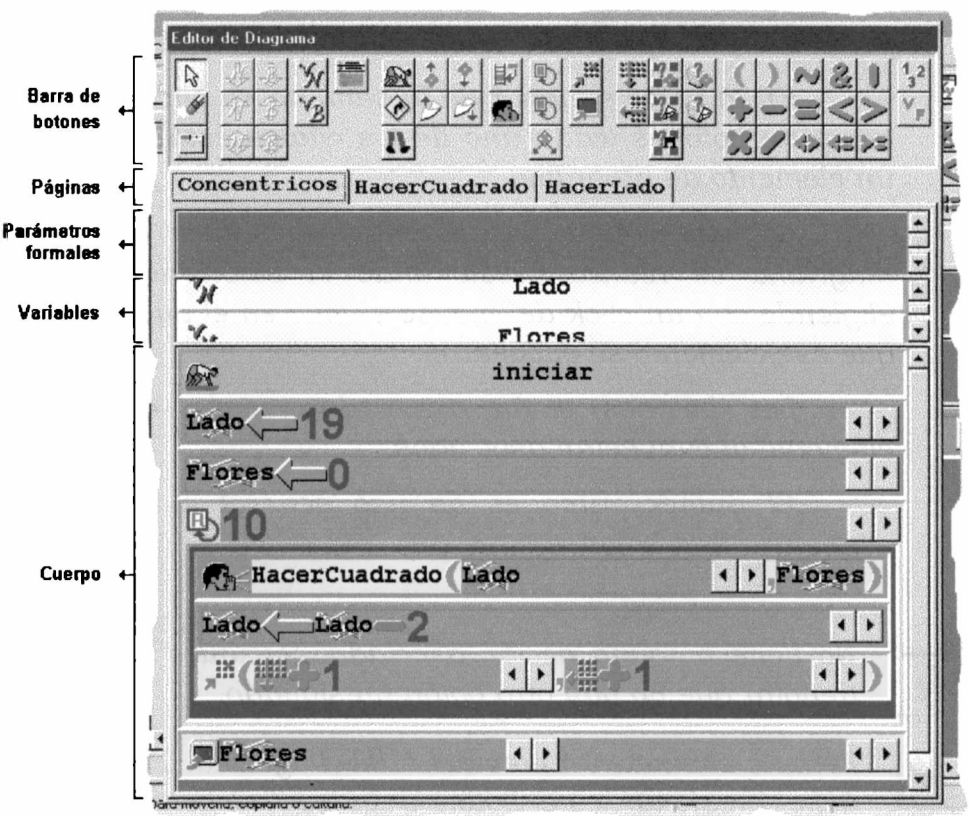


***Figura 6.2**: Editor de Diagrama*

La mayor parte de esa ventana está dedicada al diagrama propiamente dicho, el cual puede contener varias páginas. En la primera de esas páginas se especifica el programa principal y en las demás, un proceso en cada una.

Cada página está dividida en tres partes: una parte para la definición de los parámetros formales, otra para la declaración de las variables locales y la última para el cuerpo.

Naturalmente, la parte de definición de parámetros formales de la primera página no es utilizable, ya que el programa principal no recibe ni devuelve parámetros.

En la parte superior de la ventana se encuentra una barra de botones, cada uno de los cuales está asociado a un elemento de programa.

La inserción de los distintos elementos dentro del diagrama se realiza presionando el botón adecuado y eligiendo con un click del mouse el sitio en el que se desea que ese elemento se encuentre.

Los botones están agrupados según el tipo de elemento al cual están asociados.

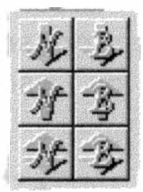
El primer grupo es de uso general e incluye los siguientes tres botones:

No tiene ninguna acción asociada. Su único objetivo es denotar que no hay otro botón presionado.

Permite eliminar un elemento del diagrama. Para eso se debe presionar este botón y luego "clickear" sobre el elemento del diagrama que se desea borrar.

Presenta el diálogo de la figura 6.3 por medio del cual es posible modificar el nombre de la unidad (programa o proceso) correspondiente a la página actual.

El segundo grupo de botones es el que se encarga de las declaraciones. En él se encuentran los botones para la declaración de parámetros formales, variables y procesos.

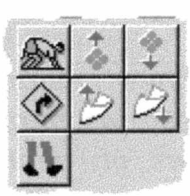


Permiten declarar parámetros numéricos de entrada, booleanos de entrada, numéricos de salida, booleanos de salida, numéricos de entrada/salida y booleanos de entrada/salida respectivamente. Con uno de estos seis botones presionados y una vez seleccionada la parte de definición de parámetros formales en la que desea hacer la declaración, aparece un diálogo similar al de la figura 6.3 para ingresar el nombre del parámetro.

Permiten declarar variables numéricas y boolenas respectivamente. Una vez seleccionada la parte de definición de variables en la que desea hacer la declaración, aparece un diálogo similar al de la figura 6.3 para ingresar el nombre de la variable.

Crea una página nueva y muestra el diálogo de la figura 6.3 de manera que el usuario asigne un nombre al proceso correspondiente a esa página.

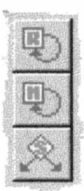
El tercer grupo es el de los botones asociados a las instrucciones. Contiene a las primitivas, a las sentencias simples y a las sentencias compuestas.



Permiten agregar las instrucciones primitivas `iniciar`, `derecha`, `mover`, `tomarFlor`, `tomarPapel`, `depositarFlor` *y* `depositarPapel` *en la sección del diagrama destinada al cuerpo de la unidad actual. La*

posición del elemento visual que se agrega, depende del punto que indique el usuario con el mouse.

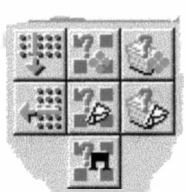
Permiten agregar al diagrama las instrucciones simples asignación e invocación a un proceso definido por el usuario. En este último caso aparece el diálogo que se muestra en la figura 6.4. En este diálogo es posible seleccionar uno de los procesos existentes o bien ingresar un identificador nuevo. Si se agrega un identificador, se crea una página para el nuevo proceso con ese identificador como nombre.



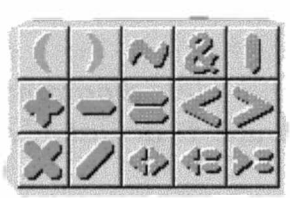
Permiten agregar las estructuras de control `repetir`, `mientras` *y* `si`.

Permiten agregar las invocaciones correspondientes a los procesos del sistema `Pos` *e* `Informar`.

Otro grupo de botones es el que corresponde a las variables del sistema.



Estos son los botones que se deben utilizar para agregar una de las siguientes variables del sistema a una expresión: `HayFlorEnLaEsquina`, `HayPapelEnLaEsquina`, `PosAv`, `PosCa`, `HayFlorEnLaBolsa`, `HayPapelEnLaBolsa` *y* `HayObstaculo`.



En este conjunto de botones están agrupados los operadores necesarios para la especificación de expresiones ya sean lógicas o aritméticas.

Este último grupo contiene a los botones que permiten agregar valores numéricos y booleanos a una expresión. Estando presionado uno de estos botones, al "clickear" en un elemento visual correspondiente a una expresión, aparece el diálogo de la figura 6.5(a) o el de la figura 6.5(b), según se trate de un valor numérico o booleano respectivamente.

***Figura 6.3**: Diálogo de nombre de unidad*

Para las siguientes tres situaciones, en vez de presionar un botón de la barra superior y clickear dentro del diagrama, se utiliza la técnica conocida con el nombre "drag&drop":

- *Cambiar de posición el elemento visual correspondiente a una sentencia del cuerpo.*
- *Poner una variable, un parámetro de salida o un parámetro de entrada/salida a la izquierda de una asignación.*

- *Poner una variable, un parámetro de entrada o un parámetro de entrada/salida como parte de una expresión.*

En los dos últimos casos, el elemento se "arrastra" desde su declaración y se "suelta" en el sitio adecuado.

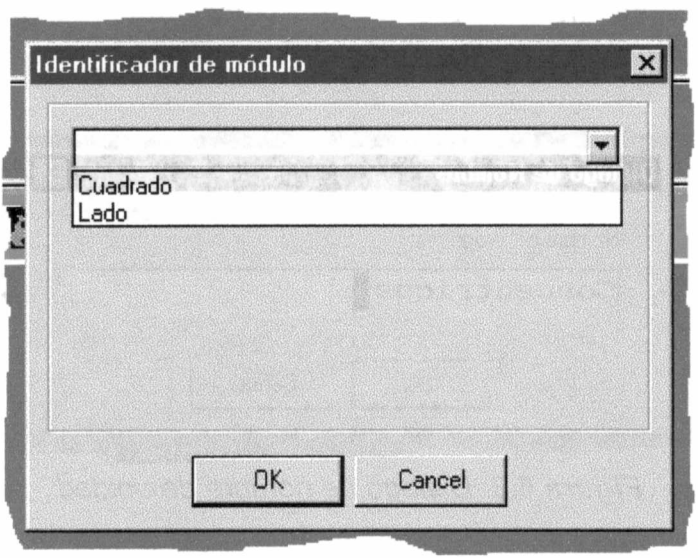


***Figura 6.4**: Selección de Proceso Invocado*

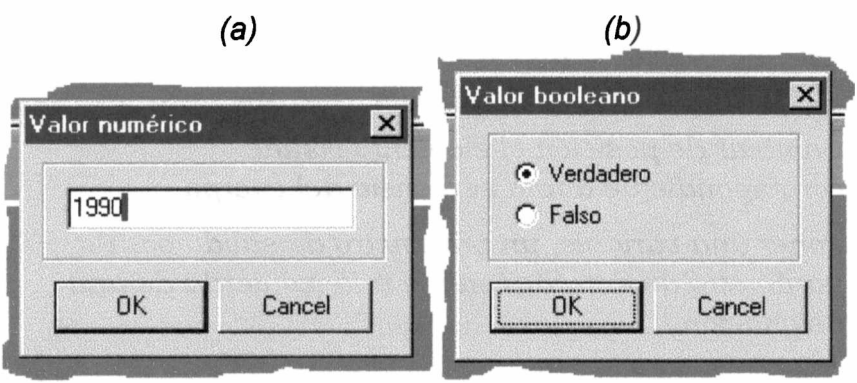


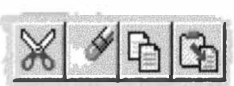
***Figura 6.5**: (a) Diálogo de valor numérico; (b) Diálogo de valor booleano*

Todo elemento visual que se agregue al diagrama, se modifique o elimine, es automáticamente traducido a código textual. Esa traducción se ve reflejada instantáneamente en la posición correspondiente del editor de código.

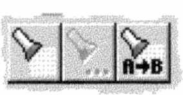
6.3. Editor de Código

La ventana de edición de código (figura 6.6) es un editor de texto muy simple, cuya única asistencia al programador es una indentación semiautomática (cada línea nueva comienza en la misma columna que la anterior).

Las facilidades de edición provistas son las que se detallan a continuación:



Cortar el texto seleccionado y enviarlo al portapapeles. Borrar el texto seleccionado. Copiar el texto seleccionado en el portapapeles. Hacer una copia del contenido del portapapeles en la posición actual del cursor.



Buscar un texto en el código. Volver a buscar el último texto buscado. Reemplazar una o más ocurrencias de un texto por otro.

En el momento de la ejecución de un programa, ya sea total o paso a paso, se resalta la instrucción corrientemente ejecutada, excepto que se indique lo contrario en las opciones del ambiente, como se detalla en la sección 6.6.

6.4. Ciudad

La ventana de la ciudad se muestra en la figura 6.7. Solamente contiene el dibujo de la ciudad de cien calles por cien avenidas.

En la ejecución de un programa, se puede ver al robot circulando por la ciudad y dejando una línea roja en todo lugar por el que pasa.

Como la ciudad completa no cabe dentro de la ventana, se utilizan barras de desplazamiento, también llamadas de "scroll", de modo tal que el usuario pueda ver otras partes de la ciudad según desee. Si durante la ejecución de un programa el robot se escapara de la zona visible de la ciudad, se producirá un desplazamiento automático de manera que el usuario pueda seguir su trayectoria permanentemente.

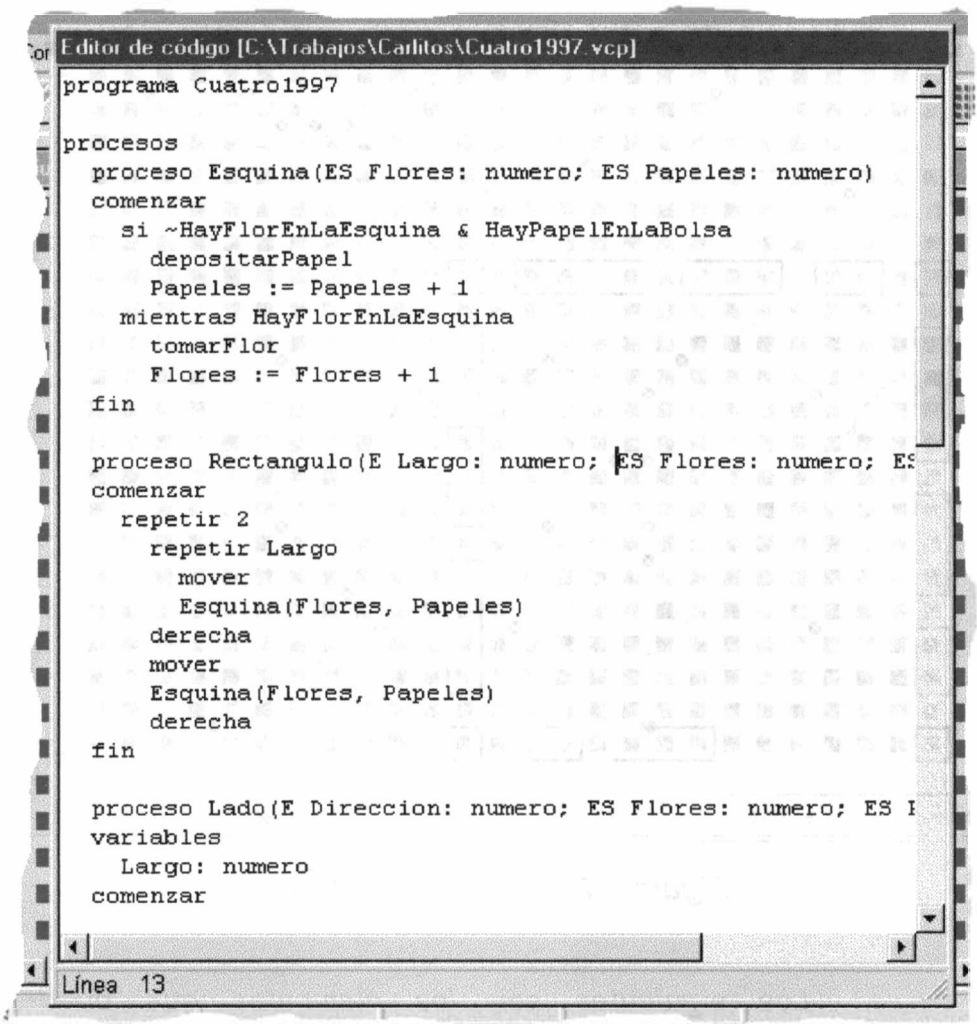


***Figura 6.6**: Editor de código*

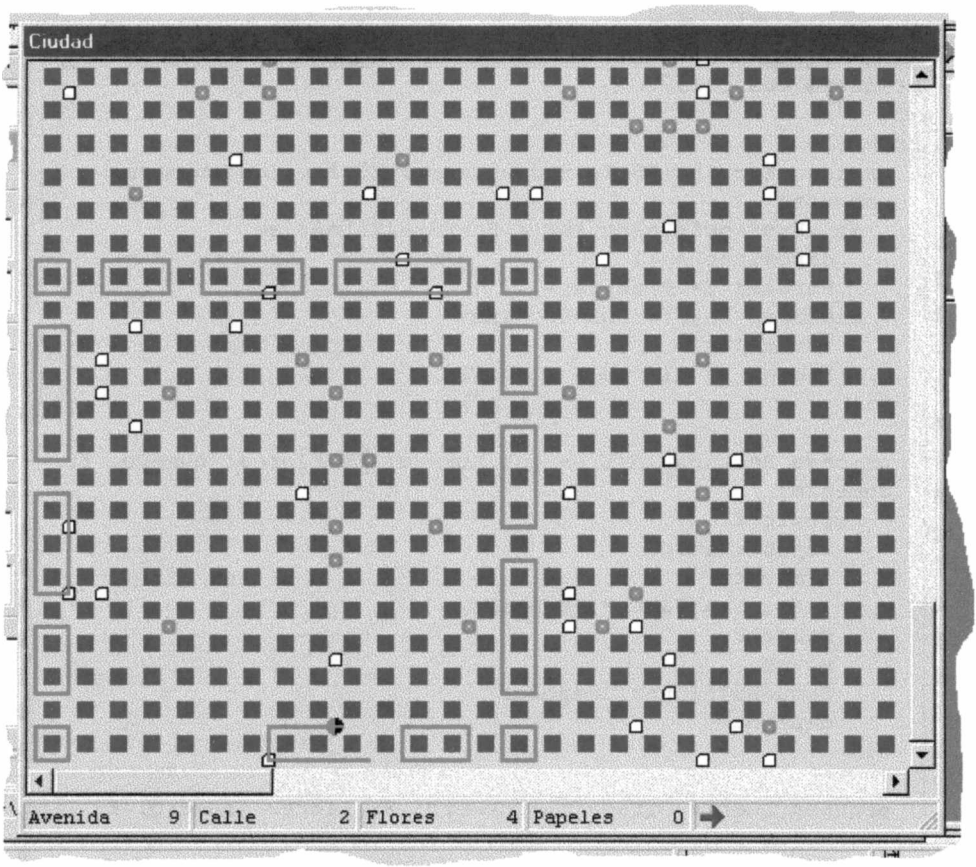


***Figura 6.7**: Ventana de la Ciudad*

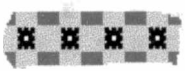
Los siguientes son los elementos que pueden intervenir en la ejecución de un programa apareciendo en las esquinas de la ciudad cuando se ejecuta un programa:

 *Robot*

 *Flor*

Papel

Obstáculo



Barrera

En la parte inferior de la ventana de la figura 6.7, se muestran los valores de las variables del sistema más importantes y su variación en el progreso de la ejecución de un programa. Aunque la misma información se puede ver también en la ventana del inspector de variables del sistema, como se describe en la siguiente sección, puede ocurrir que esta se encuentre "tapada" por otra ventana.

6.5. Inspector de Variables del Sistema

Como se puede ver en la figura 6.8, en el inspector de variables del sistema simplemente se muestran los valores actuales de las variables del sistema relacionadas con el robot durante la ejecución de un programa. Esos valores van cambiando con el progreso de la ejecución.

Los valores que se muestran en el inspector son los correspondientes a las coordenadas actuales del robot, es decir, la avenida y la calle en la que está posicionado, la dirección que lleva, y la cantidad de flores y papeles que tiene en sus respectivas bolsas.

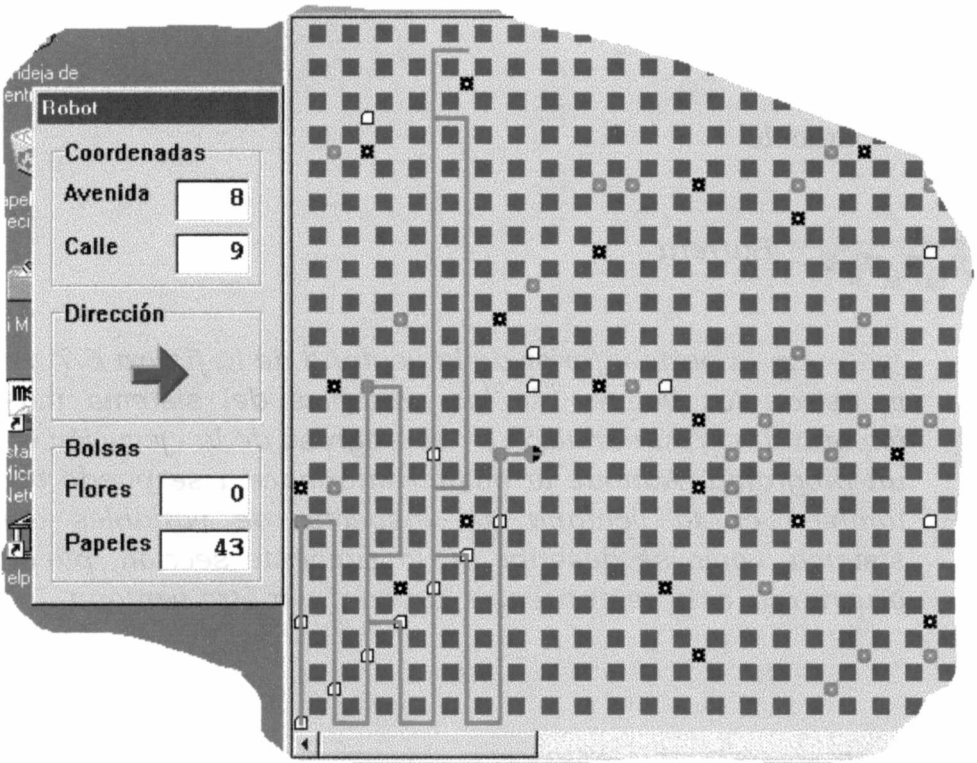
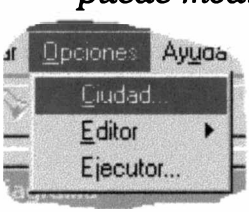


***Figura 6.8**: Inspector de Variables del Sistema*

6.6. Opciones del Ambiente

El ambiente provee ciertas opciones que el usuario puede modificar.

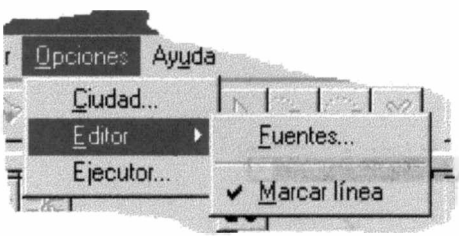


La opción Ciudad del menú Opciones muestra el diálogo de las figuras 6.9. Este diálogo está compuesto por tres páginas. En la página de la figura 6.9(a), el usuario puede seleccionar

la cantidad de flores que aparecerán en la ciudad durante la ejecución, la zona de la ciudad en la que esas flores aparecerán, determinada por los vértices inferior izquierdo y superior derecho, y la cantidad inicial de flores que el robot lleva en su bolsa respectiva.

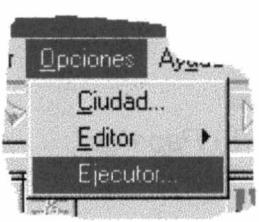
En la página de la figura 6.9(b), el usuario inicializa los mismo valores pero para los papeles.

Por último, en la página de la figura 6.9(c), se puede indicar la cantidad de obstáculos que deben aparecer automáticamente, la zona en la que deben aparecer y si pueden conformar barreras o deben ser sólo obstáculos simples.



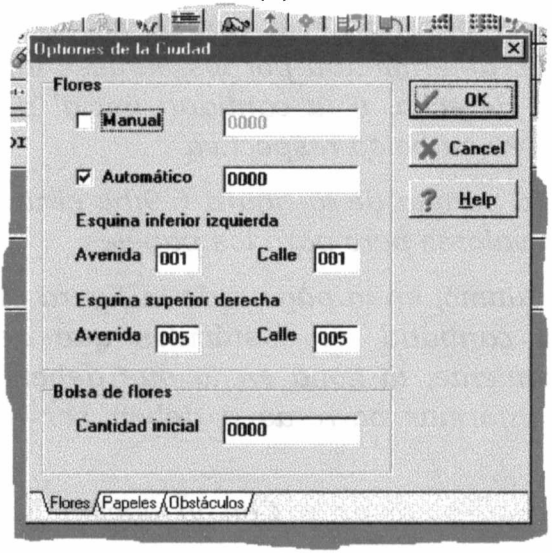
Con la opción Editor, se puede modificar el tipo de letra utilizado en el editor de código. Sin embargo se recomienda utilizar siempre la fuente Courier New, por ser una de las más comunes entre las que utilizan la misma cantidad de puntos como ancho de las letras. Esto es importante porque permite ver claramente si la indentación es correcta o no.

También se puede indicar que se resalte cada línea de código cuando es ejecutada. Esto resulta sumamente útil sobretodo cuando la ejecución de un programa se realiza paso a paso.



La última de las opciones permite incorporar un retraso entre la ejecución de una instrucción y la siguiente. El sentido de esto es reducir la velocidad de ejecución para poder seguir cada paso de un algoritmo sin necesidad de ejecutarlo paso a paso.

(a)



(b)

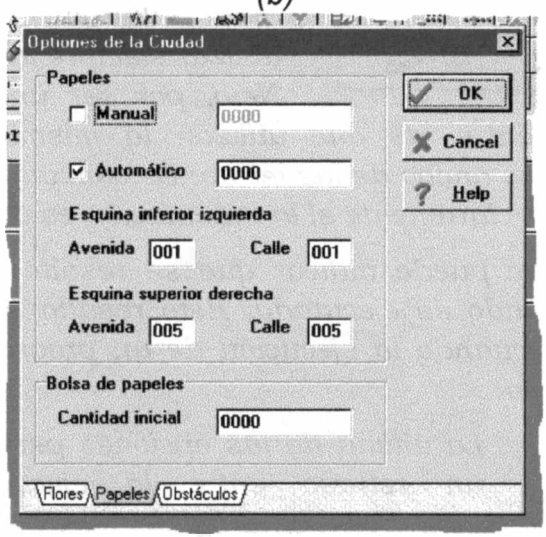


***Figura 6.9**: Opciones de la ciudad. (a) Flores; (b) Papeles; (c) Obstáculos*

(c)

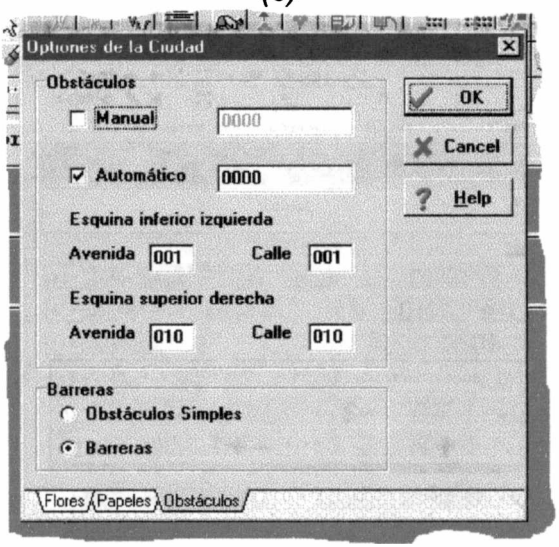


***Figura 6.9**: Opciones de la ciudad. (a) Flores; (b) Papeles; (c) Obstáculos (Continuación)*

6.7. Ejemplo

Este ejemplo muestra la solución al mismo problema planteado en la sección 5.10, pero esta vez desarrollada en forma visual.

En las figuras 6.10(a), 6.10(b) y 6.10(c) se puede ver los diagramas correspondientes al programa principal, al proceso `HacerCuadrado` *y al proceso* `HacerLado` *respectivamente.*

(a)

(b)

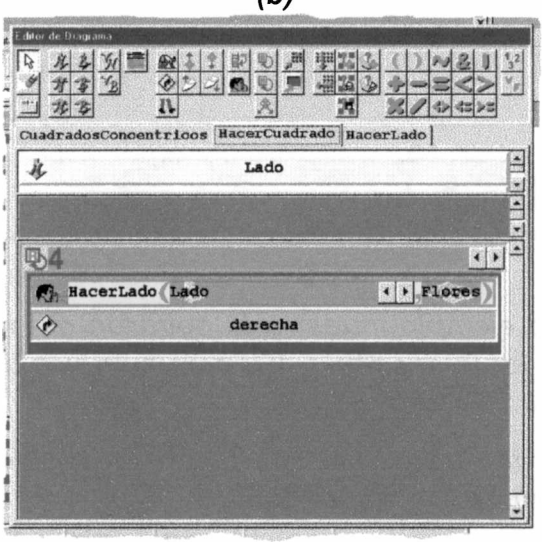


***Figura 6.10**: Diagrama Visual. (a) Programa* `CuadradosConcentricos`*; (b) Proceso* `HacerCuadrado`*; (c) Proceso* `HacerLado`

(c)

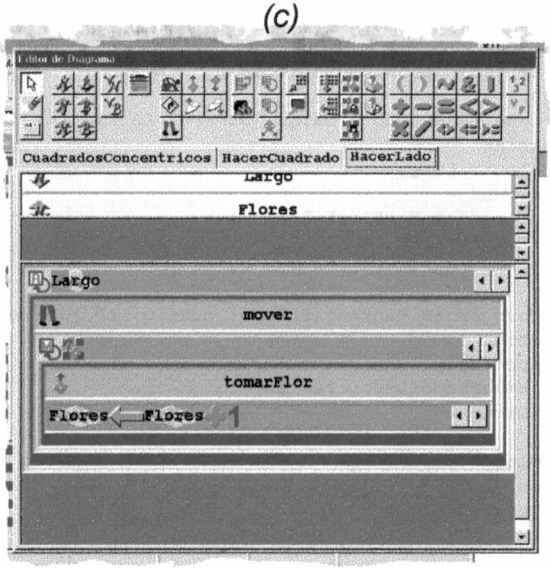


***Figura 6.10**: Diagrama Visual. (a) Programa* `CuadradosConcentricos`*; (b) Proceso* `HacerCuadrado`*; (c) Proceso* `HacerLado` *(Continuación)*

Con la confección visual de tales diagramas se genera automáticamente el código de la figura 6.11.

El resultado parcial de la ejecución del programa generado y previamente verificado, se muestra en la figura 6.12.

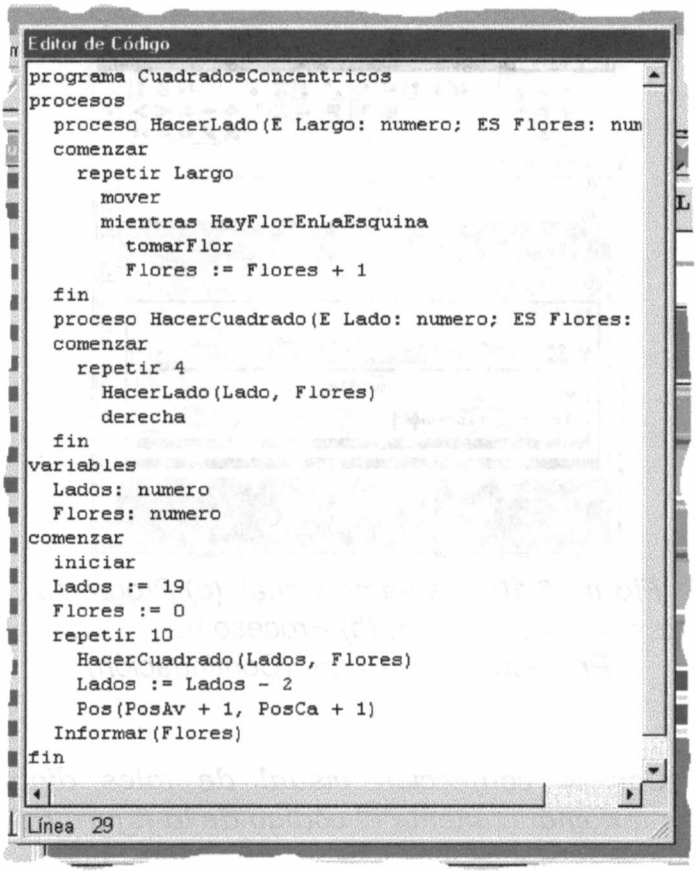


***Figura 6.11**: Código generado automáticamente*

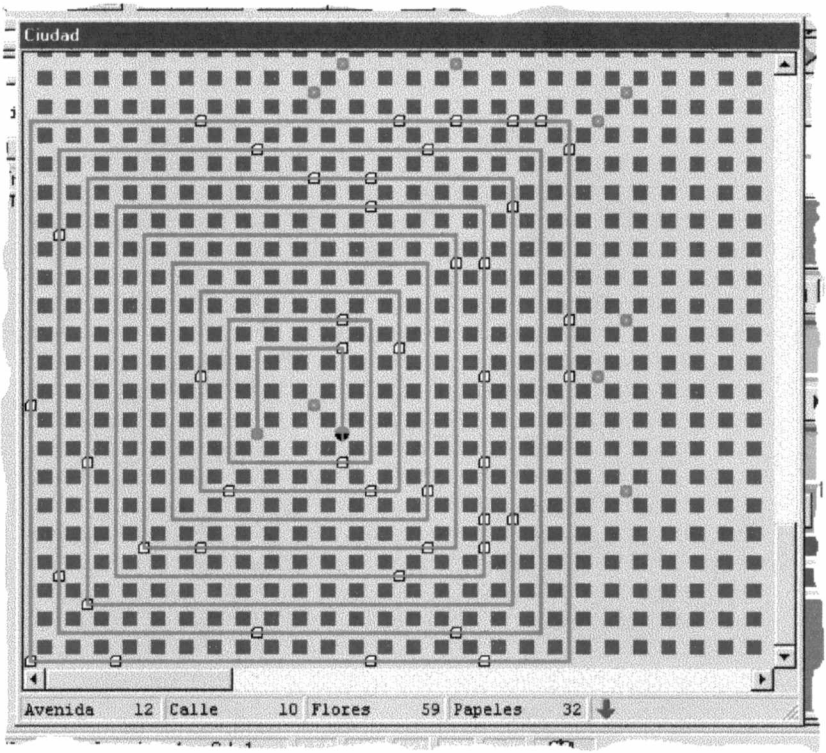


***Figura 6.12**: Ejecución*

Parte 3

Aspectos de Implementación

En esta parte se describen los aspectos más importantes de la implementación del Visual DaVinci.

El proceso de desarrollo en Visual DaVinci puede ser dividido en las siguientes partes:

- *Edición de programas (visual y/o textualmente)*
- *Verificación de sintaxis*
- *Asignación de valores a las variables del sistema*
- *Ejecución y visualización (completa o paso a paso)*

En la especificación de programas intervienen tanto la edición visual como la textual. Sin embargo, para la edición textual de código se precisa solamente un editor de textos sencillo, cuya implementación no contribuye en nada. Esto no significa que el editor de diagramas sea muy original, pero al menos merece unos párrafos.

Por otro lado, la asignación de valores a variables del sistema es ni más ni menos que un conjunto de asignaciones.

En consecuencia, en los próximos cuatro capítulos se pone atención sólo en los aspectos de implementación de la edición visual de diagramas, de la traducción automática de diagramas a código textual, de la verificación de código y por último de la ejecución de programas y visualización de sus efectos. No obstante, se modifica el orden en el que estos temas son tratados para una mejor explicación.

Capítulo 7

Verificación y Ejecución

En términos generales, el Visual DaVinci es un ambiente de desarrollo de programas. Un programa puede ser especificado en forma visual o textual. En ambos casos, se debe verificar la sintaxis del código desarrollado, lo cual genera código intermedio, que es utilizado por un intérprete para la ejecución.

La elección de un intérprete por sobre un compilador está basada en la sencillez relativa del desarrollo del primero respecto del segundo.

Un compilador debería generar código de máquina combinado con mensajes a Windows, y esto no contribuye a los objetivos de este trabajo. Mientras tanto, un intérprete puede hacer uso de recursos de más alto nivel.

Por otro lado, no hay requerimientos importantes acerca de la eficiencia de ejecución.

7.1. Verificación de sintaxis

La verificación de sintaxis, así como la ejecución, se realiza directamente sobre la base del código textual. De esta manera se logra que la eficiencia de la ejecución de un programa sea independiente de la naturaleza visual del lenguaje. De allí proviene gran parte de la importancia de la derivación automática de diagramas a código textual.

Dado el código de un programa, generado visual o textualmente, el mismo debe pasar aceptablemente la verificación de sintaxis, para poder ser luego ejecutado. Si un programa es sintácticamente correcto, el resultado de la verificación es directamente el código intermedio que utiliza el intérprete para su ejecución.

Como se describe en el capítulo 3, los compiladores e intérpretes generalmente arman ciertas estructuras de datos tales como árboles de parsing, tablas de símbolos, etc.

En Visual DaVinci, se optó por un esquema significativamente distinto, el cual es uno de los motivos más importantes por los que se hizo uso de la orientación a objetos en la implementación.

En vez de recorrer el código, para asegurar la corrección sintáctica del mismo, armando un árbol de parse y luego recorrer a este para generar algún código intermedio, se crea un objeto `programa`*, instancia de la*

clase TPrograma, *que "sabe" cómo debe analizar el código correspondiente.*

Este objeto, a su vez, crea un objeto proceso *por cada uno de los procesos definidos en el programa, un objeto* variable *por cada variable declarada y un objeto* cuerpo. *Es decir que un programa tiene una lista de objetos* TSubprograma, *una lista de objetos* TVariable *y un cuerpo, el cual es instancia de la clase* TSecuencia, *como sus atributos.*

De igual manera, un objeto proceso *tiene como atributos una lista de objetos* parametro *(instancias de la clase* TParametro*), una lista de objetos* proceso, *una lista de objetos* variable *y un cuerpo.*

La figura 7.1 muestra la jerarquía que forman las clases involucradas en la verificación sintáctica.

Cada uno de los objetos que intervienen en un programa tienen un método llamado Analizar, *a través del cual sabe cómo verificar la sintaxis de su propio código. Si la sintaxis es incorrecta, el objeto que la verifica produce una excepción que aborta la verificación e indica al usuario cuál es el error detectado en tiempo de compilación, por medio de un mensaje.*

Las clases TPrograma *y* TSubprograma *son herederas de* TCustomPrograma, *la cual es la que realmente define los atributos* Variables, Procedimientos *y* Cuerpo, *y el método* Analizar.

Este método verifica el encabezamiento, los procesos, las variables y el cuerpo, en ese orden, invocando respectivamente a los métodos AnalizarEncabezado, *AnalizarProcesos, AnalizarVariables y AnalizarCuerpo.*

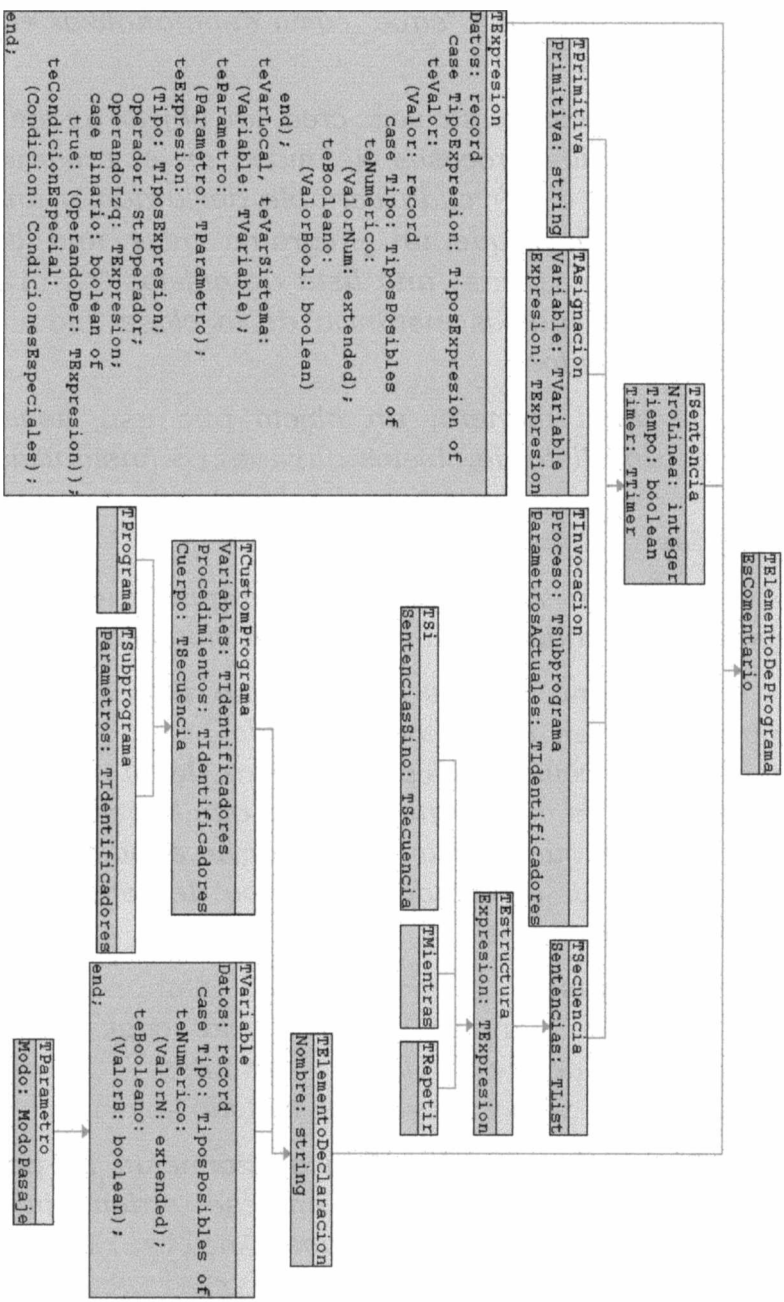


***Figura 7.1**: Jerarquía de objetos de verificación y ejecución*

AnalizarEncabezado *busca una de las palabras clave* programa *o* proceso, *según la unidad de la que se trate, e invoca al método* AnalizarParámetros. *Este último es un método virtual abstracto, razón por la cual se ejecutará el método* AnalizarEncabezado *sobreescrito por las clases* TPrograma *o* TSubprograma. *En el primer caso, no hay nada para hacer, ya que los programas no reciben parámetros. En el segundo, se crea un objeto de la clase* TParametro *por cada uno de los parámetros especificados en el código, con su respectivo modo, identificador y tipo. Los parámetros definidos correctamente son agregados a la lista de parámetros formales; es decir, al atributo* Parametros, *el cual es una instancia de la clase* TIdentificadores.

El método AnalizarProcesos *primeramente busca la palabra clave* procesos, *la cual indica el comienzo de la definición de tales unidades. Si la encuentra, crea un objeto instancia de la clase* TSubprograma *por cada proceso definido, le pide a estos que analicen su código y los agrega a la lista* Procedimientos, *que es un atributo de la unidad propietaria de dichos procesos.*

Las acciones llevadas a cabo por el método AnalizarVariables *son similares a las del anterior. Se busca la palabra clave* variables *y si se encuentra, por cada declaración se crea una instancia de la clase* TVariable, *se le pide que se analice y se agrega a la lista* Variables, *atributo de la unidad propietaria de esas variables.*

AnalizarCuerpo *es el último de los métodos invocados por el* Analizar *de* TCustomPrograma. *Comprueba la existencia de la palabra clave* comenzar *e invoca al método virtual abstracto* AnalizarIniciar, *que*

no hace nada en el caso de un proceso, y asegura que se encuentre la primitiva `iniciar` *como primer instrucción del cuerpo en el caso de un programa. Luego pide al atributo* `Cuerpo`*, el cual es una instancia de la clase* `TSecuencia`*, que analice la secuencia de sentencias que componen el cuerpo. Finalmente, se asegura que exista la palabra clave* `fin`.

La clase `TSentencia` *tiene un método virtual abstracto* `Analizar` *que obliga a sus descendientes, a definir el comportamiento de verificación sintáctica adecuado. Los descendientes directos de* `TSentencia` *son* `TPrimitiva`, `TAsignacion`, `TInvocacion` *y* `TSecuencia`.

Un objeto `TSecuencia` *se encarga de manipular una lista de sentencias, para lo cual tiene un atributo llamado* `Sentencias`. *Es utilizado como cuerpo de los programas y de los procesos. Además, también es el cuerpo de las estructuras de control, ya que las clases* `TSi`, `TMientras` *y* `TRepetir` *están definidas como herederas de* `TEstructura`*, la cual, a su vez, hereda de* `TSecuencia`.

La verificación sintáctica que lleva a cabo el método `Analizar` *de un objeto* `TSecuencia` *analiza línea por línea, identificando para cada instrucción si se trata de una primitiva, una asignación, una invocación, una iteración condicional, una iteración incondicional, o una selección. Simultáneamente se comprueba que la indentación de cada línea de código sea la correcta. Esta es la única forma de determinar el comienzo y el fin de la secuencia de sentencias.*

El análisis de cada elemento consiste en crear el objeto correspondiente, pedirle que analice su código y agregarlo a la lista de sentencias, atributo de la clase `TSecuencia`.

En el caso de las primitivas, el método `Analizar` *simplemente verifica que la secuencia de caracteres encontrada pertenezca al conjunto de las primitivas.*

La asignación busca el identificador correspondiente a una variable definida o a un parámetro de salida o entrada/salida, cuyo nombre no sea igual al de una variable del sistema. Luego le pide a su atributo `Expresion` *que analice la expresión de la derecha del signo* `:=`.

El método `Analizar` *de un objeto expresión comienza verificando que los paréntesis utilizados, si los hay, estén balanceados. Luego busca los operadores que intervienen en la expresión. Si encuentra alguno, identifica al principal para determinar la aridad (unario o binario), y el tipo de la expresión. Además, en función de la existencia o no de al menos un operador, determina si la expresión está compuesta por subexpresiones o solamente por un valor, una variable o un parámetro. En todos los casos crea la cantidad necesaria de objetos expresión y les dice que se analicen.*

En una invocación se debe buscar el identificador del proceso invocado para asegurar que el mismo está definido, ya sea como local al proceso en cuyo cuerpo se encuentra la invocación, definido fuera de este, o bien sea el nombre de un proceso del sistema (`Pos`*,* `Informar`*). Luego se examina si el proceso invocado tiene parámetros formales definidos o no. Si los tiene, se debe comprobar que la cantidad de parámetros actuales sea la misma que la de parámetros formales y, para cada uno de ellos, la correspondencia de los modos en que son pasados y sus tipos.*

Si el proceso invocado no tiene parámetros formales definidos, puede ser que se trate del proceso del sistema `Informar`. *Este, inicialmente no tiene parámetros formales definidos, los cuales son creados en tiempo de ejecución, dependiendo de los parámetros actuales con los que es invocado. Esto es así como una forma de permitir una cantidad dinámicamente variable de parámetros pasados al proceso* `Informar`. *Como los parámetros sólo pueden ser de entrada, se analiza cada una de las expresiones que aparecen separadas por comas entre los paréntesis de la invocación.*

Un objeto `TSi` *busca la palabra clave* `si` *y analiza una expresión comprobando que el tipo de la misma sea* `boolean`. *A continuación invoca al método* `Analizar` *heredado de* `TSecuencia` *para realizar la verificación del conjunto de sentencias asociado al resultado verdadero de la expresión. Luego examina si existe la palabra clave* `sino` *en la estructura y de ser así, eventualmente crea otro objeto* `TSecuencia`, *que asigna al atributo* `SentenciasSino`, *al cual se le pide que analice la secuencia de sentencias asociadas al resultado falso de la expresión.*

El análisis de un `mientras` *o de un* `repetir` *es un tanto más sencillo que el anterior. Se debe analizar una expresión, comprobando que sea de tipo* `boolean` *para el primero y de tipo* `numero` *para el segundo, e invocar al método* `Analizar` *heredado de la clase* `TSecuencia` *para que verifique la sintaxis de la secuencia de sentencias asociadas.*

El resultado de la verificación (aunque suene pretencioso podría decirse compilación. Ver capítulo 3) es en sí mismo el código intermedio, ya que es el que se utiliza

para la ejecución del programa, como se describe en la siguiente sección.

Este esquema permite desatender por completo todos aquellos aspectos relacionados con la implementación de árboles y tablas utilizados en los compiladores e intérpretes convencionales. Sin embargo, si se hace un análisis exhaustivo de los atributos de los objetos involucrados en la generación del código intermedio, que son en definitiva los que lo componen, y se supone que se tiene acceso a ellos en todo momento y desde cualquier lugar, se puede pensar que lo que se obtiene es definitivamente un árbol. Aún así, no hay necesidad de ocuparse de su implementación. Como un ejemplo, en la figura 7.2 se muestra el esquema del código intermedio obtenido a partir del ejemplo de la sección 6.7. En esa figura se supone acceso incondicional a los atributos de los objetos.

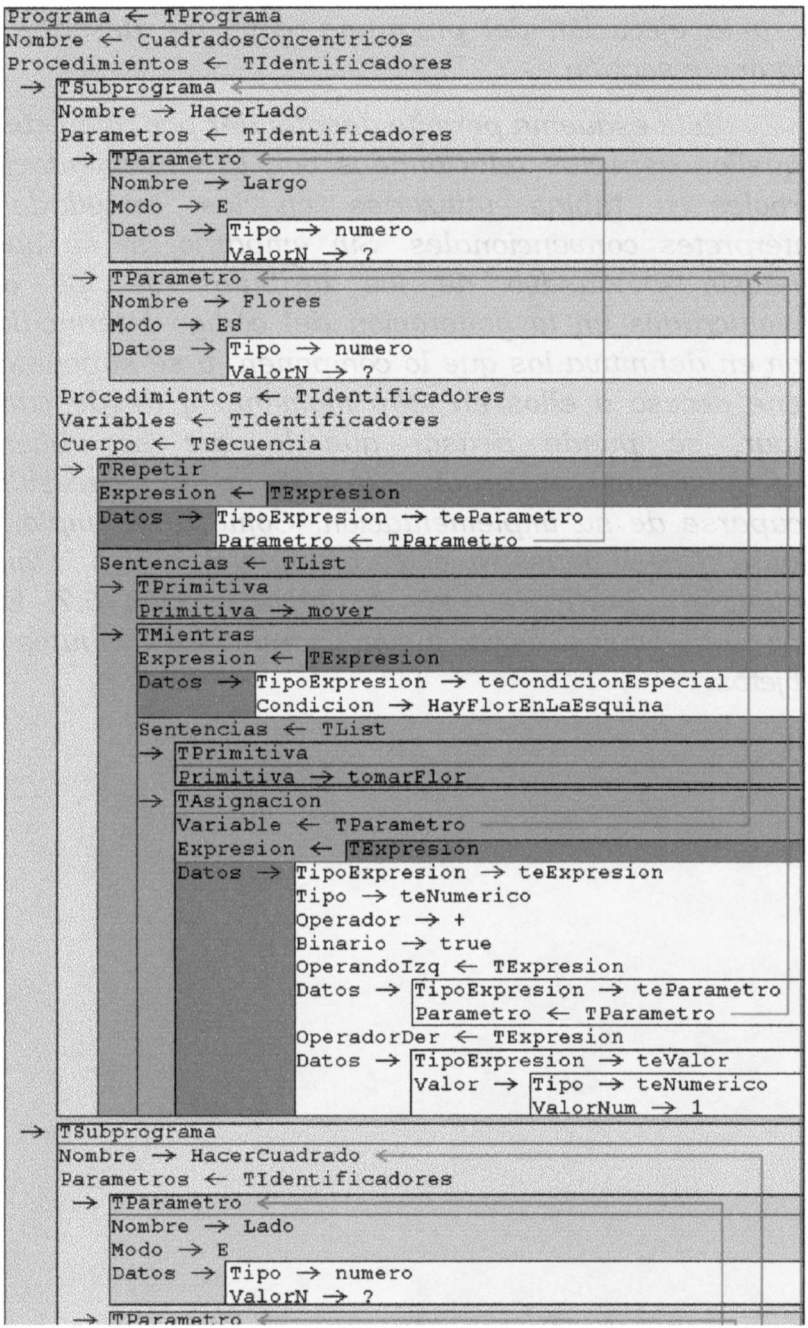


***Figura 7.2**: Código intermedio*

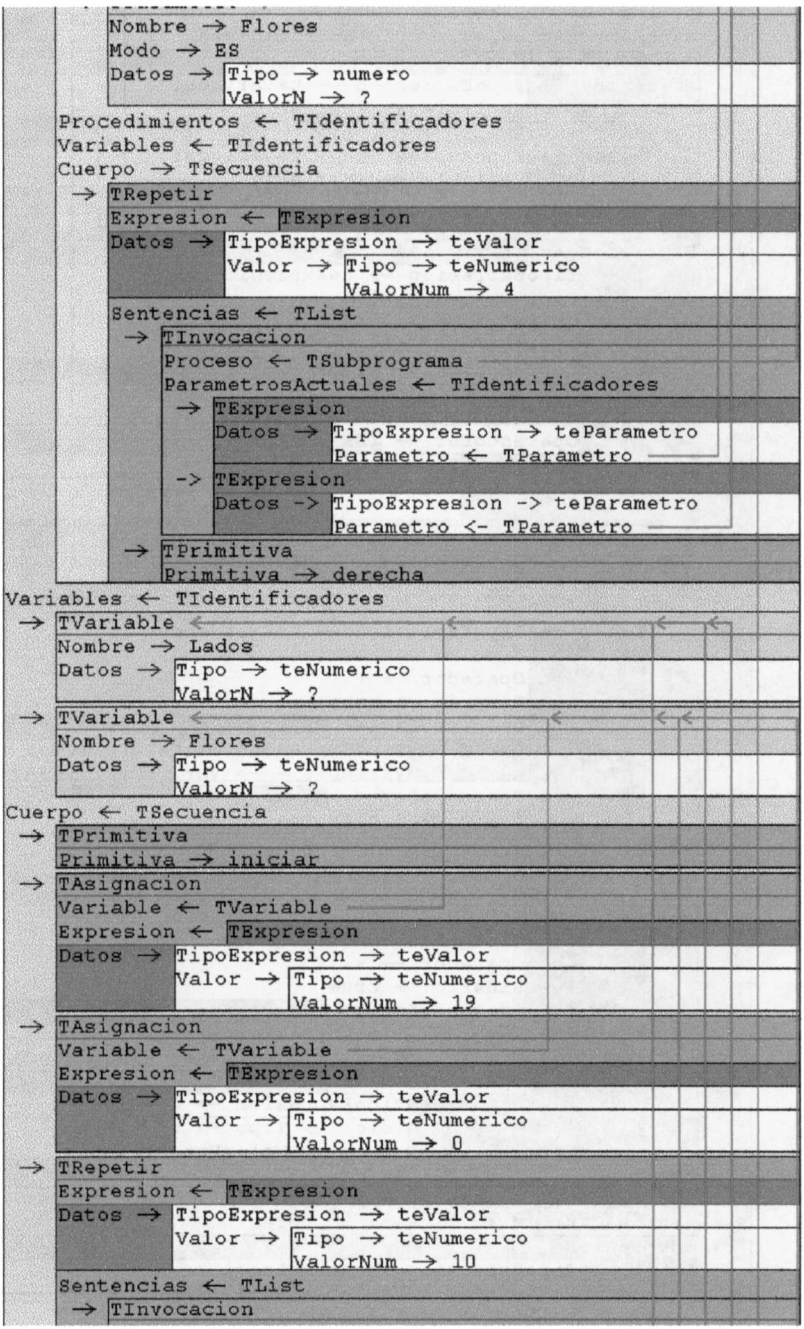


***Figura 7.2**: Código intermedio (Continuación)*

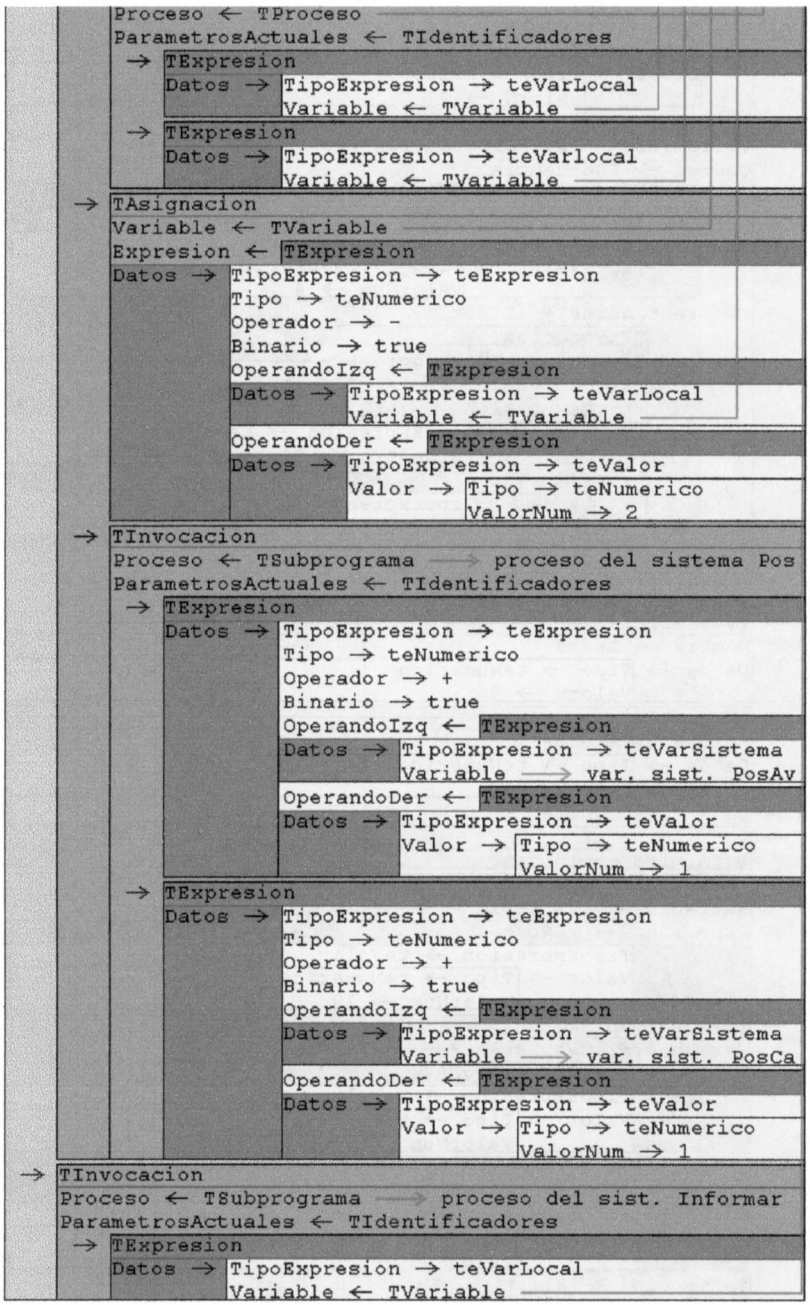


***Figura 7.2**: Código intermedio (Continuación)*

7.2. Ejecución de Programas

El resultado de la verificación sintáctica es el código intermedio que se utilizará para la ejecución. Ese código intermedio es, en definitiva, un objeto programa con sus procesos, sus variables y su cuerpo como atributos.

Siguiendo la misma filosofía descripta en la sección anterior, cada uno de los objetos que forman parte del código intermedio, conoce cómo debe ser ejecutado. Es decir, sabe ejecutarse a sí mismo.

En consecuencia, el resultado obtenido del proceso de verificación es una conjunción del código intermedio y el intérprete.

Así, para ejecutar un programa sólo hace falta decirle a una instancia de la clase `TPrograma` *que se ejecute.*

Al igual que antes, el objeto `programa` *le dirá a cada una de las sentencias que componen su cuerpo que se ejecute, siguiendo la secuencia en que esas sentencias fueron especificadas. Las sentencias a las que un programa deberá pedir que se ejecuten, pueden ser primitivas, asignaciones, invocaciones a procesos, o estructuras de control. A su vez, las estructuras de control pueden ser selección (*`si`*), iteración condicional (*`mientras`*) o iteración incondicional (*`repetir`*).*

Cuando el programa le dice a una primitiva que se ejecute, esta sólo debe decirle al robot que ejecute las

acciones correspondientes. El robot es una instancia de la clase `TRobot`.

Si la primitiva es `iniciar`, *el robot inicializa las variables del sistema, se posiciona en la esquina (1, 1) de la ciudad orientado hacia el norte, y le pide al inspector de variables del sistema y a la ventana de la ciudad que muestren los valores de tales variables.*

Las variables del sistema reciben esa denominación debido al rol que desempeñan, aunque en realidad están implementadas como atributos del objeto `robot`. *Por eso es este el encargado de inicializarlas. Esto se debe a la consideración de una posible extensión del ambiente que soporte una cantidad cualquiera de robots en la ciudad. La esquina en la que se encuentra un robot, su orientación, las cantidades de sus bolsas, etc. ya no podrían ser vistas como variables del sistema sino, necesariamente, como atributos propios de cada robot.*

Si la primitiva es `mover`, *el robot "camina" hacia la siguiente esquina dependiendo de su orientación, por medio de una pequeña animación de veinte puntos intermedios. Modifica los valores de las variables del sistema adecuadas e informa de estos cambios al inspector de variables del sistema y a la ventana de la ciudad para que los reflejen. La ventana de la ciudad, además dibuja una línea roja entre la posición anterior y la actual del robot.*

Ante la primitiva `derecha`, *el robot modifica su orientación, y esto es denotado también por el inspector de variables del sistema y la ventana de la ciudad.*

Cuando se trata de las primitivas `depositarFlor` *o* `depositarPapel`, *el robot primero verifica que haya al menos un elemento en la bolsa respectiva. Si no lo hay,*

genera una excepción que aborta la ejecución del programa y muestra un mensaje explicando cuál fue el error en tiempo de ejecución. Si por el contrario, hay un elemento para depositar, el robot indica a la ciudad que lo dibuje en la esquina en la que él se encuentra. Luego decrementa la cantidad de elementos de la bolsa de la cual extrajo el que depositó, y avisa este hecho a la ciudad y al inspector de variables del sistema.

Para `tomarFlor` *y* `tomarPapel`*, la situación es análoga. Si en la esquina en la que está corrientemente el robot no hay al menos una flor, la primitiva* `tomarFlor` *producirá un error en ejecución. Lo mismo sucederá si* `tomarPapel` *se ejecuta cuando el robot está en una esquina en la que no hay ningún papel. Si se puede ejecutar, se harán las modificaciones pertinentes a las variables del sistema relacionadas con las bolsas del robot y, como antes, se avisará a la ciudad y al inspector de variables del sistema.*

Si el programa pide a una asignación que se ejecute, esta debe evaluar una expresión y asignar su resultado a una variable. Para eso, el objeto `asignacion` *le pide a su atributo* `Expresion`*, instancia de la clase* `TExpresion`*, que se evalúe. El resultado de esa evaluación es asignado al objeto referenciado por el atributo* `Variable`*, el cual puede ser una variable, un parámetro de salida o un parámetro de entrada/salida.*

Una expresión puede estar compuesta por un valor, una variable definida por el usuario, una variable del sistema, un parámetro de entrada, un parámetro de entrada/salida o un conjunto de los anteriores conectados por operadores, como fue descripto en la sección 5.7.

La evaluación de una expresión devuelve en todos los casos otra expresión que representa un valor. Para formar esta expresión es necesario primero identificar la estructura de la que se debe evaluar.

Si se trata de un valor, la expresión resultante es el mismo valor.

Si es una variable, la expresión resultante es el valor actual de la variable.

Lo mismo sucede si se trata de un parámetro, ya sea de entrada o de entrada/salida.

Si es una expresión compuesta, se debe identificar previamente si el operador principal es unario o binario. Si es unario, se evalúa la subexpresión que se obtiene de sacar dicho operador a la expresión original, el que luego es aplicado al valor resultante. Si es un operador binario, se evalúan las expresiones que este conecta, y luego se ejecuta la operación entre los valores obtenidos.

La ejecución de una invocación requiere de una serie de pasos. Primero es necesario evaluar las expresiones pasadas como parámetros reales de entrada y obtener los valores de los parámetros reales de entrada/salida. Luego se copian esos valores en los parámetros formales correspondientes del proceso invocado, y se pide al objeto `proceso` *(instancia de la clase* `TSubprograma`*) que se ejecute. Una vez concluida la ejecución del proceso, se copian los valores con los que han quedado los parámetros formales de entrada/salida y de salida, en los correspondientes parámetros reales.*

Se debe realizar una consideración especial para el caso en el que se invoque al proceso del sistema `Informar`*. Como en principio no es conocida la cantidad de*

parámetros que este proceso recibe ni el tipo de cada uno de ellos, antes de copiar los valores correspondientes, se deben crear los parámetros formales en tiempo de ejecución.

Si el subprograma al cual una invocación le pide que se ejecute es el proceso del sistema `Pos`*, este solamente debe indicar al robot que se posicione en donde indican los parámetros. Si en cambio es el proceso del sistema* `Informar`*, sólo se muestra un mensaje con el valor deseado. En cualquier otro caso, la ejecución de un proceso es similar a la del programa principal. Debe ejecutar una secuencia de sentencias.*

La ejecución de una estructura de control, evalúa primeramente la expresión asociada. Esta resultará en un valor lógico para las instancias de las clases `TSi` *y* `TMientras`*, y en un valor aritmético para las de la clase* `TRepetir`*.*

En el caso de un `si`*, cuando la evaluación de la expresión resulte verdadera, se ejecutará la secuencia de sentencias listadas en el atributo* `Sentencias`*. Si por el contrario resultará falso y el atributo* `SentenciasSino` *fuese una* `TSecuencia` *no vacía, entonces esta será la ejecutada.*

Para un `mientras`*, se ejecutará la secuencia* `Sentencias` *cada vez que el resultado de evaluar la expresión sea verdadero.*

Un `repetir` *ejecuta* `Sentencias` *la cantidad de veces que indique el valor obtenido de la expresión.*

Capítulo 8

Edición y Traducción de Diagramas

El editor de diagramas está implementado por medio de una jerarquía de objetos. Cada uno de estos objetos sabe cómo debe visualizarse dentro de un diagrama y cómo traducirse a código textual.

Esta es la forma en que es llevada a cabo la traducción automática y simultánea de cada elemento que se inserta, elimina o modifica en el diagrama visual.

8.1. Edición de Diagramas

El editor en sí es una instancia de la clase `TDiagramForm`*, la cual es derivada de* `TForm`*, y tiene, entre otros, los atributos* `PaginaPrograma` *y* `ListaPaginas`.

El primero de estos es de la clase `TPagina` *y es el que contiene el diagrama asociado al programa principal.*

El otro es una lista de las páginas que contienen los diagramas de los procesos que son invocados desde el programa principal, desde los procesos invocados por el principal, y así sucesivamente. Esta lista se utiliza para determinar el orden en el cual deben ser traducidos a código los distintos procesos que intervienen en el programa. También se utiliza para identificar a aquellos procesos que habiendo sido especificados visualmente, no son invocados en ningún momento.

Los objetos `TPagina` *tienen, entre otros, los atributos* `ParametrosPanel`, `VariablesPanel` *e* `InstruccionesPanel` *de las clases* `TParametrosPanel`, `TVariablesPanel` *y* `TCuerpoPanel` *respectivamente.*

Las clases `TParametrosPanel` *y* `TVariablesPanel` *son herederas de* `TCuerpoPanel`. *Estas tres se encargan del manejo de la parte visible de las distintas partes de un programa o subprograma, es decir, declaración de parámetros formales (sólo para procesos), declaración de variables y especificación del cuerpo.*

Las declaraciones de parámetros y variables que aparecen respectivamente en el `ParametrosPanel` *y en el* `VariablesPanel`, *están representadas por instancias de alguna de las siguientes clases herederas de la clase* `TParVarPanel`*:*

- `TParametroEntradaNumPanel`
- `TParametroSalidaNumPanel`
- `TParametroEntradaSalidaNumPanel`
- `TParametroEntradaBoolPanel`
- `TParametroSalidaBoolPanel`
- `TParametroEntradaSalidaBoolPanel`

- TVariableNumericaPanel
- TVariableBooleanaPanel

Hay una clase para cada posible combinación de modo de pasaje y tipo de parámetro, y una clase para cada posible tipo de variable.

Para las sentencias del cuerpo se tiene la clase TSentenciaPanel, *de la cual heredan* TInstruccionPanel *y* TSentenciaCompuestaPanel.

Las siguientes clases son las encargadas de las primitivas e instrucciones simples y son de TInstruccionPanel*:*

- TInstruccionIniciar
- TInstruccionMover
- TInstruccionDerecha
- TInstruccionTomarFlor
- TInstruccionTomarPapel
- TInstruccionDepositarFlor
- TInstruccionDepositarPapel
- TInstruccionAsignacionPanel
- TInstruccionInvocacionPanel
- TInstruccionInformarPanel
- TInstruccionPosPanel

TInstruccionAsignacionPanel *tiene los atributos* Variable *de la clase* TParVarImage *y* Expresion *de la clase* TExpresionPanel.

TParVarImage *es simplemente una imagen que adecua su tamaño al nombre que identifica de la variable o*

del parámetro al cual representa. Tiene como heredera a la clase TValorImage *que se encarga de representar a los valores lógicos y aritméticos.*

TExpresionPanel *es un panel que puede manejar las representaciones de variables, parámetros, valores y operadores. Tiene como atributo un objeto de la clase* TMiUpDown *que permite el desplazamiento para el caso de las expresiones largas que no pueden ser visualizadas completamente.*

Los atributos más importantes de la clase TInstruccionInvocacionPanel *son* NombreProceso, ParametrosActuales, PaginaDelInvocado *y* PaginaDeLaInvocacion. NombreProceso *contiene el identificador del proceso que se invoca.* ParametrosActuales *es una lista de objetos* TExpresionPanel *y* TParVarImage *según el modo de pasaje de los parámetros formales definidos en el proceso invocado.* PaginaDelInvocado *es una referencia a la página del editor de diagramas en la cual está definido el proceso invocado. Por último,* PaginaDeLaInvocacion *es una referencia a la página en la cual el objeto* TInstruccionInvocacionPanel *es insertado y se utiliza para verificar que no se produzcan llamados recursivos o de recursión mutua de los cuales nunca se salga.*

TInstruccionInformarPanel *tiene un único atributo, llamado* Expresiones, *el cual es una lista de objetos* TExpresionPanel.

TPosPanel *tiene los atributos* ExpresionAv *y* ExpresionCa, *ambos de la clase* TExpresionPanel.

La clase TSecuenciaCompuesta *tiene como herederas a* TSentenciaSi, TSentenciaMientras *y*

TSentenciaRepetir. *Sus atributos son* Cuerpo *y* Expresion *de las clases* TCuerpoPanel *y* TExpresionPanel *respectivamente.*

De las herederas, la única que define nuevos atributos es TSentenciaSi *con su* CuerpoSino *de la clase* TCuerpoPanel.

La jerarquía que forman las clases anteriores se muestra en la figura 8.1.

8.2. Traducción a Código

Cada objeto de la figura 8.1 (exceptuando TMiUpDown, TListaPaginas *y* TDiagramForm*), sabe cómo traducirse a sí mismo a código textual a partir de su representación visual. Para llevar a cabo esa tarea cada uno tiene un método llamado* Codificar *propio o heredado.*

Cada vez que se produce un cambio en el diagrama, ya sea una inserción, una eliminación o una modificación, se invoca al método Codificar *del atributo* PaginaPrograma *del* DiagramForm.

Este método traduce a código el encabezamiento del programa principal, cada uno de los procesos invocados desde su cuerpo, las variables que declara en VariablesPanel *y el cuerpo constituido por las sentencias especificadas en* InstruccionesPanel.

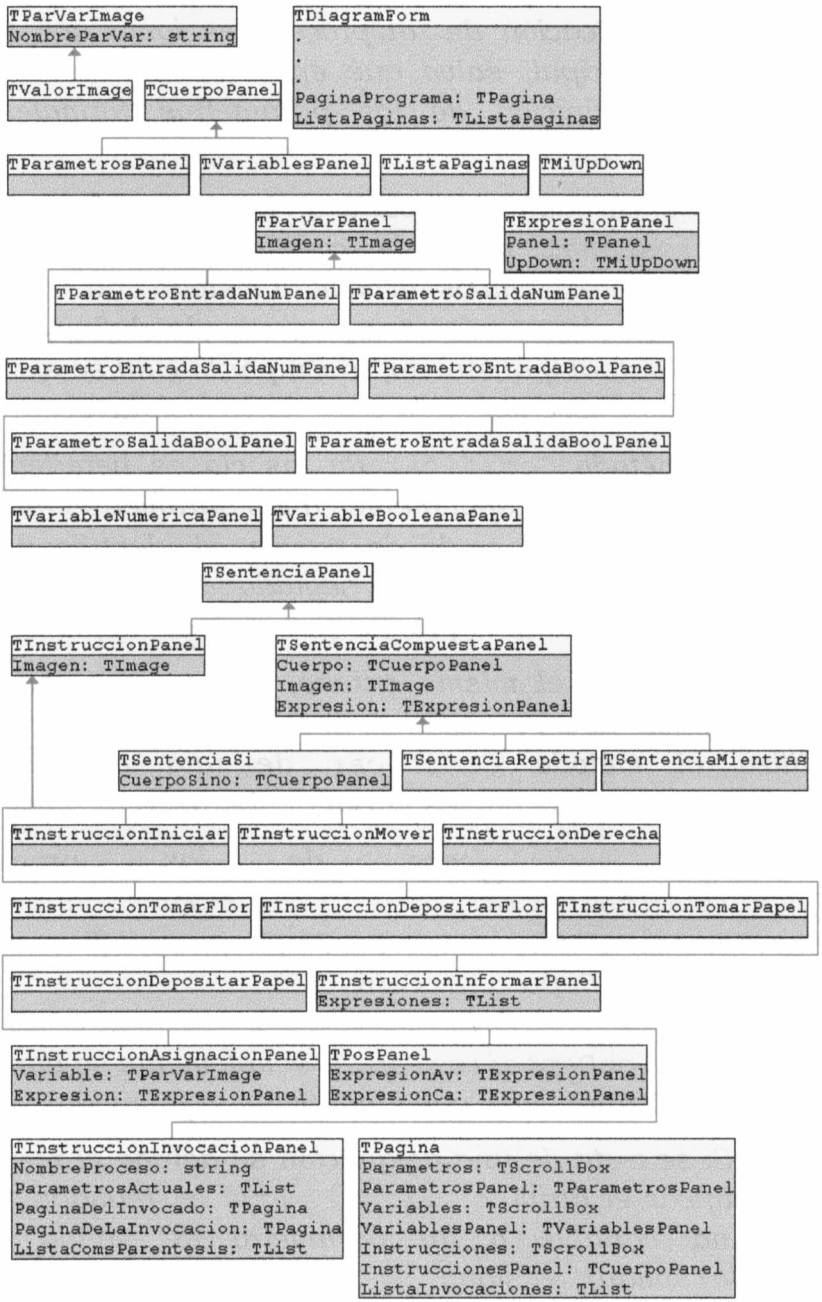


***Figura 8.1**: Jerarquía de objetos del editor de diagramas*

La codificación de un proceso es igual que la del programa principal, salvo que en el encabezamiento se agrega la definición de los parámetros formales del `ParametrosPanel`.

El encargado de codificar los parámetros es el `ParametrosPanel` *y el de codificar las variables el* `VariablesPanel`. *Ambos invocan al método* `Codificar` *de una instancia de* `TParVarPanel` *por cada definición. Ese método es virtual y abstracto, razón por la cual se ejecutará el método sobreescrito por la clase correspondiente.*

El método `Codificar` *de las clases herederas de* `TParVarPanel` *correspondientes a los parámetros formales agrega al código el modo de pasaje, el identificador del parámetro y el símbolo ":" seguido del tipo. Para las variables es similar excepto que por el modo de pasaje.*

Siguiendo el mismo criterio, `InstruccionesPanel` *es el responsable de codificar las instrucciones. Para ello utiliza el método* `Codificar` *de una instancia de* `TSentenciaPanel`, *que es virtual y abstracto.*

Para el caso de las primitivas, `TInstruccionIniciar`, `TInstruccionDerecha`, `TInstruccionMover`, `TInstruccionTomarFlor`, `TInstruccionDepositarFlor`, `TInstruccionTomarPapel` *y* `TInstruccionDepositarPapel`, *simplemente se agrega al código la palabra clave que identifica a la primitiva.*

Si se trata de una asignación se agrega el resultado de codificar el atributo `Variable` *(que en realidad puede ser una variable o un parámetro de salida o de entrada/salida), el signo ":=" y el resultado de codificar al atributo* `Expresion`.

Cuando se codifica la invocación a un proceso, se agrega el nombre del proceso invocado y el resultado de codificar un `TExpresionPanel` *o un* `TParVarPanel` *por cada parámetro actual, dependiendo del modo de pasaje definido por los parámetros formales.*

Codificar un `TInstruccionPosPanel` *significa agregar al código la palabra clave que identifica al proceso del sistema* `Pos` *y, entre paréntesis, el resultado de codificar las dos expresiones correspondientes a la avenida y la calle.*

El `TInstruccionInformarPanel`*, al codificarse agrega al código una línea conteniendo la palabra clave* `Informar` *y, entre paréntesis, el resultado de codificar cada una de las expresiones pasadas como parámetro (salvo la última pues con cada especificación se incorpora una más, por lo tanto siempre sobra una).*

La codificación de expresiones no guarda analogía con la verificación explicada en el capítulo anterior. Sencillamente se arma una línea de código en la que se refleja en forma directa cada elemento involucrado.

Las sentencias compuestas agregan la palabra clave que las identifica seguida del resultado de codificar una expresión. Luego indican a su cuerpo, instancia de `TCuerpoPanel`*, que se codifique. El único caso especial es el* `TSentenciaSi`*, que eventualmente agregará la palabra clave* `sino` *y la codificación de su atributo* `CuerpoSino`*.*

Obviamente, se antepondrá la indentación adecuada para aquellas líneas de código que la necesiten.

8.3. Analogía de las Jerarquías

Analizando la jerarquía de clases mencionadas en este capítulo, se puede encontrar cierta analogía con la descripta en el capítulo 7.

Por ejemplo:

- `TExpresion` ⇔ `TExpresionPanel`
- `TSentencia` ⇔ `TSentenciaPanel`
- `TEstructura` ⇔ `TSentenciaCompuestaPanel`
- `TSi` ⇔ `TSentenciaSi`
- `TRepetir` ⇔ `TSentenciaRepetir`
- `TMientras` ⇔ `TSentenciaMientras`
- `TAsignacion` ⇔ `TInstruccionAsignacionPanel`
- `TInvocacion` ⇔ `TInstruccionInvocacionPanel`
- `TSecuencia` ⇔ `TCuerpoPanel`
- `etc.`

Llevado al extremo, se puede encontrar una analogía tal que las clases encargadas de la verificación sintáctica, generación de código intermedio y ejecución del programa, sean también las responsables de la visualización de cada elemento y de la traducción del diagrama visual al código textual.

Esta idea hubiera permitido que las clases necesarias para llevar a cabo todas esas tareas fuesen aproximadamente la mitad.

Sin embargo se considera apropiado respetar el concepto de la mayoría de las jerarquías de objetos (Smalltalk, C++, Delphi, ObjectWindows, Turbo Vision, etc.), en el sentido de separar las clases visibles de las no visibles. Por otro lado, cada una de esas clases tendría el doble de complejidad.

Parte 4

Posibles Extensiones

El diseño y la implementación del Visual DaVinci han sido desarrollados poniendo énfasis en las distintas posibilidades de extensión o adaptación a otras áreas.

En los sucesivos capítulos de esta parte se comentan las mejoras y extensiones que se proponen realizar.

Capítulo 9

Mejoras Propuestas

9.1. Del Diagrama al Código y del Código al Diagrama

La traducción automática de diagramas a código tiene muchas ventajas. Como por ejemplo, la eliminación de procesos definidos inútilmente, la independencia de la sintaxis a la cual se deriva, la organización del código, el estilo, además de lo relacionado con la eficiencia tanto del desarrollo como de la ejecución.

Sin embargo, si el usuario modifica el código generado automáticamente, este pierde la correspondencia con el diagrama. Una vez modificado manualmente el código generado, no se debería alterar nada en el diagrama visual pues esto haría que se pierdan las modificaciones manuales.

Lo mismo sucede en la gran mayoría de los lenguajes visuales generadores de código conocidos.

Aunque esto puede servir como un consuelo conformista, resulta sumamente interesante mejorar el ambiente en este sentido. La forma de hacerlo es sincronizando el diagrama y el código de forma tal que toda modificación en el diagrama se refleje automáticamente en el código (como se hace actualmente) y que toda modificación en el código se refleje automáticamente en el diagrama.

Para ello se propone cambiar el editor de código por otro de los que denominan "asistidos". Cada línea de código textual ingresada debe ser analizada para determinar el tipo de sentencia de la cual se trata y crear el objeto visible correspondiente según lo descripto en el capítulo 8. Finalmente se debe insertar dicho objeto en el sitio adecuado dentro del diagrama visual.

Lógicamente esto requiere de mayor control sobre las posiciones de los objetos visibles y de las líneas de código, y modificar levemente el traductor para respetar correctamente esas posiciones.

9.2. Reusabilidad

La única reusabilidad posible, por el momento, es la más primitiva: "copy and past" sobre el código textual. Pero eso haría que se pierda la correspondencia con el diagrama.

Obviamente es deseable algún mecanismo de reusabilidad que facilite el proceso de desarrollo y se aplique al diagrama y al código en forma simultánea.

La solución propuesta es encapsular a los procesos especificados en entidades con características similares a los componentes del Borland Delphi™, las cuales serían almacenadas en un repositorio.

Dicho repositorio estaría implementado como un "stream" y un índice que permita la selección y búsqueda de entidades. Así, la reusabilidad estaría dada por la creación de una instancia a partir de la entidad recuperada desde el repositorio.

9.3. Visual DaVinci MultiRobots

Uno de los objetivos de este trabajo es crear la base a partir de la cual llegar a un lenguaje visual de programación concurrente destinado también a la educación.

En este, una cantidad arbitraria de robot podrán circular por la ciudad, ejecutando cada uno de ellos su propio algoritmo. Por ejemplo, si se debe juntar todos los papeles de la ciudad, en vez de poner un robot que recorra todas las esquinas, se podrá poner un robot por cada avenida, que se encargue de juntar los papeles de esta. O, en el caso extremo, un robot por cada esquina.

Considerando esta extensión, las variables del sistema se implementaron como atributos del robot, pues en un ambiente multirobots habrá tantas de cada una de tales variables como robots.

Se implementarán distintos mecanismos de sincronización y comunicación entre robots tales como semáforos, monitores, pasaje de mensajes sincrónico y asincrónico, etc., y una especie de "cobegin/coend" para la creación dinámica de "threads", de manera de cubrir la mayor cantidad posible de las construcciones de especificación de concurrencia.

9.4. Un Robot Real

Está en los planes del LIDI comprar un robot con las siguientes características:

Se modificarán las instrucciones primitivas que ejecuta una instancia de la clase `TRobot` *para que, además de realizar las acciones pertinentes y mostrar sus resultados en la ciudad de la pantalla, envíe las sentencias adecuadas al robot real de forma tal que ejecute las mismas acciones.*

Aunque puede resultar una experiencia sumamente interesante, no se pretenden resultados novedosos a partir de la implementación de esto. El aspecto más importante está centrado en la motivación que se puede lograr sobre el alumno cuando este vea que los algoritmos que desarrolla son ejecutados por un robot real.

Conclusiones

Este trabajo ha sido una exploración sobre programación visual y lenguajes de programación visual, considerando los temas relacionados tales como sintaxis y semántica de lenguajes, compiladores e intérpretes, notaciones de especificación, interfaces gráficas con el usuario, derivación de código, etc.

Se ha implementado un ambiente de desarrollo que permite la especificación de programas tanto en forma visual como textual. Los diagramas visuales son traducidos automática y simultáneamente a código, de manera que se mantenga una correspondencia permanente entre ambos.

El objetivo de este ambiente es el aprendizaje de programación estructurada, intentando considerar los diversos aspectos que hacen a un buen estilo de programación de manera que alumno tenga los elementos necesarios para formar el suyo propio.

Se han solucionado los problemas habituales de los lenguajes de programación visual.

El problema de la notación dispersa se soluciona por medio de abstracción procedural.

La interpretación se realiza sobre el código textual y no sobre la representación visual, con lo cual se logra mayor eficiencia, pero por sobre todo, esto permite que la

eficiencia de ejecución no dependa de la condición visual del lenguaje. Así, el problema de la eficiencia es el mismo que para cualquier lenguaje textual convencional.

El código textual generado es compilado a código intermedio. El chequeo de tipos se realiza durante esa compilación, evitando así otro de los problemas frecuentes de los lenguajes visuales.

Quizás el aspecto más innovador de este trabajo sea la forma en que se llevan a cabo los pasos necesarios para la compilación a código intermedio y la ejecución interpretada del mismo.

En vez de crear y mantener diversas estructuras auxiliares para la generación de código intermedio y para la ejecución, se crea una única estructura de objetos. Cada uno de los objetos que pueden intervenir en esa estructura tiene el conocimiento necesario para su propia verificación sintáctica y para su propia ejecución, combinando así el código intermedio con el intérprete.

Para ello se ha desarrollado una jerarquía de objetos que contempla a cada uno de los elementos que pueden intervenir en un programa.

Asimismo se ha desarrollado una jerarquía de objetos encargada de la visualización de los elementos del diagrama, separando así los objetos visibles de los no visibles.

No se presentan conclusiones acerca de las ventajas y desventajas de la utilización del Visual DaVinci en la enseñanza de programación estructurada, debido a que aún no ha sido utilizado sistemáticamente con alumnos.

Sin embargo, se presume que su utilización mejorará significativamente el grado de aprendizaje gracias a la motivación que produce sobre el alumno.

Durante el Curso de Ingreso 1997, la sola mención de la existencia de este lenguaje provocó que una gran cantidad de alumnos se movilizara para conseguir una copia. Muchos de ellos llevaron a las teorías y a las prácticas, listados de los algoritmos que desarrollaron, con inquietudes sobre su corrección, funcionamiento, etc.

Este trabajo sienta las bases de los trabajos futuros.

Una mejora prevista es implementar una relación bidireccional entre el diagrama visual y el código para que además de traducir los diagramas a código, se traduzca el código a diagrama. Para ello se debe cambiar el editor de textos simple por otro asistido que verifique y reconozca cada elemento de programa que se ingrese textualmente.

Otra mejora es permitir la reusabilidad encapsulando los procesos implementados en componentes, e implementando un repositorio de componentes en el cual se puedan realizar altas, modificaciones, consultas y búsquedas.

Se pretende realizar dos extensiones: una que permita la concurrencia de un número cualquiera de robot en la ciudad, cuyo objetivo está centrado en la enseñanza de programación concurrente, y la otra consiste en incorporar un robot real que ejecute las mismas acciones sobre una maqueta de la ciudad que el robot de la ciudad de la pantalla.

Referencias

Libros

[Aguilar88] "Fundamentos de Programación. Algoritmos y Estructuras de Datos". Aguilar. McGraw-Hill. 1988

[Burnett94] "Visual Object-Oriented Programming". M. M. Burnett, A. Goldberg, T. G. Lewis. Prentice Hall and Manning. 1994

[Chang86] "Visual Languages". S.-K. Chang, T. Ichikawa, P.A. Ligomenides. Plenum Press. 1986

[Chang90] "Visual Languages and Visual Programming". S.-K. Chang. Plenum Press. 1990

[CSA90] "Occam and the Transputer". Computer System Architects. 1990

[Denert74] "PLAN2D - Towards a Two-Dimensional Programming Language". Serie "Lecture Notes in Computer Science". Número 26. Springer Verlag. 1974

[Ghezzi87] "Programming Languages Concepts". Ghezzi, M. Jazayeri. John Wiley and Sons. 1987

[Ghezzi91] "Fundamentals of Software Engineering". Ghezzi, M. Jazayeri, D. Mandrioli. Prentice Hall. 1991

[Gildersleeve70] "Decision Tables and their Practical Applications in Data Processing". T. R. Gildersleeve. Prentice Hall. 1970

[Glinert90a] "Visual Programming Environment: Paradigms and Systems". E. P. Glinert. IEEE Computer Society Press. 1990

[Glinert90b] "Visual Programming Environment: Applications and Issues". E. P. Glinert. IEEE Computer Society Press. 1990

[Hunter85] "Compilers. Their Design and Construction Using Pascal". R.Hunter. John Wiley & Sons. 1985

[Kernighan88] "The C Programing Language". B. Kernighan, D. Ritchie. Englewood Cliffs. Prentice Hall. 1988

[McDaniel70] "Application of Decision Tables". H. McDaniel. Brandon Systems Press. 1970

[Meyer94] "Reusable Software: The Base Object-Oriented Component Libraries". B. Meyer. Prentice Hall. 1994

[Olsen83] "Ada for programmers". E. Olsen, S. Whitehill. Prentice-Hall. 1983

[Peterson81] "Petri Net Theory and the Modeling of Systems". J.L.Peterson. Prentice Hall. 1981

[Pollack71] "Decision Tables: Theory and Practice". S.Pollack, H.Hicks, W. Harrison. Willey. 1971

[Shu88] "Visual Programming". N. C. Shu. Van Nostrand Reinhold. 1988

[Smith77] "Pygmailion - A Computer Program to Model and Stimulate Creative Thought". D. C. Smith. Birkhauser. 1977

[USDoD83] "Reference Manual for the Ada Programming Languaje". ANSI-MIL-STD-1815ª. U.S. Department of Defense, 1983

[Yourdon75] "Structured Design". E. Yourdon, L. Constantine. Yourdon Press. 1975

[Yourdon93] "Análisis Estructurado Moderno". Yourdon. Prentice Hall. 1993

Tesis Doctorales

[Bell92a] "Using Programming Walkthroughs to Design a Visual Language". B. Bell. University of Colorado at Boulder. 1992

[Golin90a] "A Method for the Specification and Parsing of Visual Languages". E. J. Golin. Brown University. 1990

[McIntyre92b] "A Visual Method for Generating Iconic Programming Environment". D. W. McIntyre. Rensselaer Polytechnic Institute. 1992

[Najork94] "Programming in Three Dimensions". M. Najork. University of Illinois. 1994

[Petri62] "Kommunikationen Mit Automaten". C. A. Petri. University of Bonn. 1962

[Sutherland66] "On-Line Graphical Specification of Computer Procedures". W. R. Sutherland. MIT. 1966

Articulos

[Böhm66] "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules". C. Böhm, G. Jacopini. Communications of the ACM. Volumen 9. Número 5. 1966

[Borning86] "Graphically Defining New Building Blocks in ThingLab". A. Borning. Human Computer Interaction. Volumen 2. 1986

[Brooks87] "No Silver Bullet". F. P. Brooks, Jr. IEEE Computer. Volumen 20. Número 4. 1987

[Burnett89] "Influence of Visual Technology on the Evolution of Language Environments". A. L. Ambler, M. M. Burnett. IEEE Computer. Volumen 6. Número 2. 1989

[Chapin74] "New Format for Flowcharts". Software-Practice and Experience. Volumen 4. Número 4. 1974

[Finzer84] "Programming by Rehearsal". W. Finzer, L. Gould. BYTE. Volumen 9. Número 6. 1984

[GACote94] "Desktop Telephony". R. GA Cote. BYTE. Volumen 19. Número 3. 1994

[Glinert84] "Pict: An Interactive Graphical Programming Language". E. P. Glinert, S. Tanimoto. IEEE Computer. Volumen 7. Número 25. 1984

[Golin90b] "The Specification of Visual Language Syntax". E. J. Golin, S. P. Reiss. J. Visual Languages and Computing. Volumen1. Número 2. 1990

[Golin91] "Parsing Visual Languages with Picture Layout Grammars". E. J. Golin. J. Visual Languages and Computing. Volumen2. Número4. 1991

[Helm91] "A Declarative Specification and Semanthics for Visual Languages". R. Helm, K. Marriott. J. Visual Languages and Computing. Volumen 2. Número 4. 1991

[Heydon90] "Miro: Visual Specification of Security". A. Heydon, M. W. Maimone, J. D. Tygar, J. M. Wing, A. M. Zaremski. IEEE Trans. Software Engineering. Volumen 16. Número 10. 1990

[Hirakawa90] "An Iconic Programming System, HI--VISUAL". M. Hirakawa, M. Tanaka, T. Ichikawa. IEEE Trans. Software Engineering. Volumen16. Número10. 1990

[LabView] "LabView: Laboratory Virtual Instrument Engineering Workbench". G. M. Vose, G. Williams. BYTE. Septiembre1986.

[Myers90] "Taxonomies of Visual Programming and Program Visualization". B. A. Myers. J. Visual Languages and Computing. Volumen 1. Número 1. 1990

[Nassi73] "Flowchart Techniques for Structured Programming". I.Nassi, B.Shneiderman. ACM SIGPLAN Notices. Volumen 8. Número 8. 1973

[OBrien93] "Issues of Programming". L. O'Brien. Computer Language. Volumen10. Número1. 1993

[Raeder85] "A Survey of Current Graphical Programming Techniques". G. Raeder. IEEE Computer. Volumen 18. Número 8. 1985

[Repenning94] "Programming Substrates to Create Interactive Learning Environment". A. Repenning. Journal of Interactive Learning Environment. Volumen 4. Número 1. 1994

[Repenning95a] "Agentsheets: A Medium for Creating Domain--Oriented Visual Languages". A. Repenning, T. Summer. IEEE Computer. Volumen 28. Número 3. 1995

[Smith87] "Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic". R. B. Smith. IEEE CG \& A. Septiembre1987

[Stevens74] "Structured Design". W. Stevens, G. Myers, L. Constantine. IBM Systems Journal. Mayo 1974

[Wittenburg91] "Unification--based Grammars and Tabular Parsing for Graphical Languages". K. Wittenburg, L. Weitzman, J. Talley. J. Visual Languages and Computing. Volumen2. Número4. 1991

Proceedings

[Beguelin91] "Graphical Development Tools for Newtork-Based Concurrent Supercomputing". A. Beguelin et.al. Supercomputing 91. 1991

[Bell91] "Usability Testing of a Graphical Programming System: Things We Missed in a Programming Walkthrough". B. Bell, J. Rieman, C. Lewis. CHI'91. 1991

[Bell92b] "Simulation of Comunications Protocols through Graphical transformation Rules". B. Bell, W. Citrin. Advanced Visual Interfaces. 1992

[Bell93] "ChemTrains: A Language for Creating Behaving Pictures". B. Bell, C. Lewis. 1993 IEEE Workshop Visual Languages. 1993

[Burnett92] "A Declarative Approach to Event-handling in Visual Programming Languages". M. M. Burnett, A. L. Ambler. 1993 IEEE Workshop Visual Languages. 1992

[Cardelli83] "Two-Dimensional Syntax for Functional Languages". L. Cardelli. Integrated Interactive Computing Systems. 1983

[Carlson94] "Hypersignal for Windows Block Diagram DSP Development Environment". B. G. Carlson. 1994 DSPx Expositions and Symposium. 1994

[Chen76] "The Entity Relationship Model - Towards a Unified View of Data". P. Chen. ACM Transactions on Database Systems. 1976

[Christensen68] "An Example of the Manipulation of Directed Graphs in the AMBIT/G Programming Language". C. Christensen. Interactive Systems for Experimental and Applied Mathematics. Academic Press. 1968

[Christensen71] "An Introduction to AMBIT/L, and Diagrammatic Language for List Processing". C. Christensen. Second Symposium on Symbolic and Angebraic Manipulation. 1971

[De Giusti88] "LUBO-1: Una Máquina Abstracta para un Primer Curso de Programación". A. De Giusti, C. Madoz, P. Pesado. Anales de XIV Conferencia Latinoamericana de Informática. 1988

[De Giusti89] "Abstract Machines in a first Course of Computer Science". A. De Giusti, L. Lanzarini, C. Madoz. Proceedings 11th International Symposium "Computer at the University". Zagreb. Yugoeslavia. 1989

[Edel86] "The Tinkertoy Graphical Programming Environment". M. Edel. COMPSAC'86. 1986

[Furnas91] "New Graphical Reasoning Models for Understanding Graphical Interfaces". G. W. Furnas. CHI '91. 1991

[Ghezzi89] "A General Way to Put Time in Petri Nets". C. Ghezzi, D. Mandrioli, S. Morasca, M. Pezzè. 4th International Workshop on Software Specification and Design. IEEE Computer Society Press. 1987

[Gindling95] "LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick". J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo, A. Repenning. 1995 IEEE Workshop Visual Languages. 1995

[Glinert89] "The User's View of SunPict, an Extensible Visual Environment for Intermediate—Scale Procedural Programming". E. P. Glinert, D. W. McIntyre. Fourth Israel Conference on Computer Systems and Software Engineering. 1989

[Grundy93a] "Integrated OO Software Development in SPE". J. C. Grundy, J. G. Hosking. 1993 NZCS Conference. 1993

[Grundy93b] "Construction Multi-View Editing Environments Using MViews". J. C. Grundy, J. G. Hosking. 1993 IEEE Workshop Visual Languages. 1993

[Grundy95] "ViTABal: A Visual Language Supporting Design by Tool Abstraction". J. C. Grundy, J. G. Hosking. 1995 IEEE Workshop Visual Languages. 1995

[Ingalls88] "Fabrik -- A Visual Programming Environment". D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph, K. Doyle. OOPSLA '88. 1988

[Kopache88] "C^2: A Mixed Textual/Graphical Environment for C". M. E. Kopache, E. P. Glinert. 1988 IEEE Workshop Visual Languages. IEEE Computer Society Press. 1988

[Lakin80] "Computing with Text-Graphic Forms". F. Lakin. 1980 LISP Conference. 1980

[Lakin87] "Visual Grammars for Visual Languages". F. Lakin. 6th Nat. Conf. on Artificial Intelligence.1987

[Lyons93] "Hyperpascal: A Visual Language to Model Idea Space". P. Lyons, C. Simmons, M. Apperley. 13th New Zealand Computer Society Conf.. 1993

[McIntyre92a] "Visual Tools for Generating Iconic Programming Environments". D. W. McIntyre, E. P. Glinert. 1992 IEEE Workshop Visual Languages. 1992

[Monden86] "HI--VISUAL: A Language Supporting Visual Interaction in Programming". N. Monden, I. Yoshimoto, M. Hirakawa, M. Tanaka, T. Ichikawa. 1984 IEEE Workshop Visual Languages. 1984

[Najork91] "The Cube Language". M. Najork, S. Kaplan. 1991 IEEE Workshop Visual Languages. 1991

[Newton92] "The CODE 2.0 Graphical Parallel Programming Language". P. Newton, J. C. Browne. ACM Int. Conf. Supercomputing. 1992

[Norton90] "A Visual Environment for Designing and Simulating Execution of Processor Arrays". C. D. Norton, E. P. Glinert. 1990 IEEE Workshop Visual Languages. 1990

[Pandey93] "Is it Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study". R. Pandey, M. Burnett. IEEE Symposium on Visual Languages. 1993

[Pietrzykowski83] "The Programming Language PROGRAPH: Yet Another Application of Graphics". T. Pietrzykowski, S. Matwin, T. Muldner. Graphics Interface '83. 1983

[Poswig92] "VisaVis - Contributions to Practice and Theory of Highly Interactive Visual Languages". J. Poswig, K. Teves, G. Vrankar, C. Moraga. 1992 IEEE Workshop Visual Languages. 1992

[Scanlan87a] "A Niche for Structured Flowchart". D. Scanlan. 1987 ACM Computer Science Conference. 1987

[Scanlan87b] "Cognitive Factors in Preference for Structured Flowcharts: A Factor Analytic Study". Primera Conferencia de Autores de Yourdon Press. 1987

[StDenis90] "Specification by Example using Graphical Animation and a Production System". R. St.-Denis. 23rd Hawaii Intl Conf Syst Sci. 1990

[Sutherland63] "Sketchpad, A Man-Machine Graphical Communication System". I. E. Sutherland. AFIPS Spring Joint Computer Conference. 1963

[Wirtz93] "A Visual Approach for Developing, Understanding and Analyzing Parallel Programs". G. Wirtz. 1993 IEEE Workshop Visual Languages. 1993

Informes Técnicos

[Citrin93a] "Formal Definition of Control Semantics in a Completely Visual Language". W. Citrin, M. Doherty, B. Zorn. Dept. of Computer Science, Univ. of Colorado, Boulder. CU-CS-672-93. 1993

[Citrin93b] "Formal Semantics of Control in a Completely Visual Programming Language". W. Citrin, M. Doherty, B. Zorn. Dept. of Computer Science, Univ. of Colorado, Boulder. CU-CS-673-93. 1993

[Ellis69] "The GRAIL Project: An Experiment in Man-Machine Communications". T. O. Ellis, J. F. Heafner, W. F. Sibley. RAND. RM-5999-ARPA. 1969

[Glinert86] "PC--TILES: A Visual Programming Environment for Personal Computers Based on the BLOX Methodology". E. P. Glinert, C. D. Smith. Dept. of Computer Science, Rensselaer Polytechnic Institute. 1986

[Kimura86] "A Visual Language for Keyboardless Programming". T. D. Kimura, J. W. Choi, J. M. Mack. Washington University. WUCS-86-6. 1986