Universidad Nacional Autónoma de México

U. A. C. P. y P. del C. C. H.

I. I. M. A. S.

"LPC: Un Paradigma de la Programación Concurrente"

T E S I S

Que para obtener el Título de

MAESTRO EN CIENCIAS DE LA COMPUTACION

p r e s e n t a

MARC BENVENISTE KUEHNE

México, D. F. 1988

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

Tabla de contenido

1. Preámbulo

1.1 Introducción

La programación concurrente se ha convertido en una disciplina de gran utilidad en áreas de la computación tan diversas como son: bases de datos, redes, diseño y manufactura por computadora, inteligencia artificial etc. Este hecho se puede atribuir, por un lado, a desarrollos tecnológicos que han permitido fabricar máquinas con varios procesadores, disponer de memorias gigantescas e interconectar máquinas heterogéneas y distantes y, por el otro, a estudios teóricos que han integrado el concepto de procesos concurrentes facultados para comunicarse, tradicionalmente un concepto de sistemas operativos, a los lenguajes de programación actuales. Por la importancia de los conceptos involucrados y por las múltiples áreas en las cuales es aplicable, resulta benéfico el estudio de esta joven disciplina.

Estudiar la programación concurrente significa más que aprender a usar un lenguaje de programación concurrente; abarca la comprensión tanto de los problemas abordados por esta disciplina como de los conceptos y notaciones que han probado ser útiles en la solución de éstos.

En este trabajo, se diseña y se programa un sistema destinado a facilitar el aprendizaje de la programación concurrente. Se define un laboratorio (*L*aboratorio de *P*rogramación *C*oncurrente) que permite presentar la concurrencia mediante el uso de robots programables que se desplazan en la pantalla de una microcomputadora. El laboratorio queda conformado por un lenguaje de programación concurrente, un compilador, un intérprete y un editor de mundos. *LPC* facilita el aprendizaje de esta disciplina permitiendo observar directamente procesos concurrentes. Promueve su práctica ofreciendo un ambiente para experimentar con los conceptos y las notaciones de los lenguajes modernos de programación. *LPC* resulta más accesible que los sistemas disponibles, reservados en su mayoría a estudiantes universitarios, ya que en él se adopta una definición sumamente sencilla y atractiva del concepto de proceso. Proporciona un entramado intelectual en el cual se pueden expresar, programar, observar y

analizar una multitud de problemas del paralelismo y del no determinismo. Una versión experimental del sistema fue presentada en [Benveniste88].

1.2 Estructura de la tesis

Al estructurar esta tesis, se intentó reflejar el desarrollo del trabajo realizado. Se escogió una presentación que acentuara el carácter constructivo del desarrollo de *LPC*. La tesis se compone de ocho capítulos cuyos temas se presentan a continuación.

En el presente capítulo, se discuten cuatro sistemas que tocan el mismo tema que el de esta tesis con distintos enfoques. Las características deseadas para *LPC* quedan asentadas en la última sección.

En el segundo capítulo, se habla de la programación concurrente y se trata de ubicar el problema abordado. Se destaca la importancia de la programación concurrente, se mencionan los sistemas que han sido diseñados para el estudio y la práctica de esta disciplina y se identifican las principales dificultades encontradas en su aprendizaje.

En el tercer capítulo, se presentan las consideraciones que se hicieron en el diseño de *LPC*. Se construyen los objetos de su universo semántico, tratando de justificar cada paso de esta edificación. Cabe resaltar la precedencia de una caracterización formal del universo semántico a la definición del lenguaje de programación de *LPC*, hechos que habitualmente acontecen en el orden inverso.

En los capítulos cuatro, cinco y seis, se define el lenguaje de programación de *LPC*. Se presentan, gradualmente, definiciones formales de su sintaxis y de su semántica. Este esfuerzo promete ser recompensado con una implementación relativamente sencilla y una rápida realización del sistema. En efecto, al tiempo que se define el lenguaje se está especificando su compilador e intérprete. La semántica del lenguaje se describe usando los objetos descritos en el tercer capítulo.

En el capítulo siete, se expone la estructura modular de la actual realización de *LPC*. El sistema se programó usando el lenguaje Modula-2 [Wirth85]. Se muestran las bondades de este lenguaje en cuanto al aprovechamiento de los esfuerzos de formalización realizados durante la fase de diseño. Se incluyen los listados de los módulos de definición de las principales componentes del sistema.

Finalmente, en el capítulo ocho se analizan los resultados obtenidos y se sugieren posibles direcciones en las cuales se podría continuar con el trabajo presentado.

1.3 Trabajos anteriores

En esta sección, se discuten cuatro sistemas que se relacionan al presente. Se menciona primero a *CSP* [Hoare78] ya que en éste se inspiran muchos sistemas para el estudio de la programación concurrente. Varios de los lenguajes que se derivan de las ideas plasmadas en *CSP* se utilizan actualmente en la enseñanza. Se menciona a *Pascal-S Concurrente* [Ben-Ari82] [Berber86] y a *Gus* [Lapalme87] que fueron diseñados y realizados expresamente para dar al estudiante una herramienta accesible con la cual pudiese experimentar. Con *LPC* se propone un enfoque diametralmente opuesto a estos sistemas. *Pascal-S Concurrente* y *Gus* adoptan el mismo punto de vista en cuanto a la presentación de la ejecución de procesos concurrentes. Lo que se puede observar, durante la ejecución en el caso de *Pascal-S Concurrente* y después de la ejecución en el caso de *Gus*, es la secuencia que resulta de intercalar las instrucciones de los distintos procesos. *LPC* permite observar en la pantalla una simulación de la ocurrencia simultánea de estas instrucciones. Finalmente, se discute el sistema *Karel Concurrente* [Benveniste84] por ser un anteproyecto de *LPC*.

1.3.1 C.S.P

Un programa en *CSP* es un conjunto estático de procesos explícitos. Los procesos se comunican unidireccionalmente en forma síncrona por medio de instrucciones de entrada y de salida. Comandos custodiados, en el sentido de Dijkstra [Dijkstra75], proporcionan el no determinismo.

CSP es en realidad un sistema híbrido de modelo y de lenguaje de programación y constituye un núcleo para varios lenguajes entre los cuales se encuentran: *CSP80* [Jazaveri80], *RBCSP* [Roper81], *Occam* [INMOS84] y más recientemente *Joyce* [B.Hansen87]. La definición de conceptos claros, fundamentales y matemáticamente caracterizables fueron puntos claves en el diseño realizado por el Profesor C.A.R. Hoare. Esta motivación se refleja en la gran cantidad de semánticas formales de *CSP* que se encuentran en la literatura ([Apt80], [Levin81] y [Misra81] por citar algunas), y en

limitaciones del lenguaje tales como la exclusión de procesos recursivos, la determinación a tiempo de compilación del número y especie de procesos y la fijación, también a tiempo de compilación, de la red de sus posibles comunicaciones.

En *CSP* no se marca ninguna diferencia entre una tarea y el agente quien la ejecuta: ambos son procesos. Se hace abstracción de la dicotomía procesador-proceso. En *LPC*, se define un proceso como la asignación de una tarea a un agente procesador. Cuando se piensa en una máquina con múltiples procesadores, esta dicotomía no parece tener importancia alguna. Sin embargo, el ámbito de las redes de máquinas heterogéneas podría conferirle cierta utilidad.

1.3.2 Pascal-S Concurrente

Pascal-S Concurrente es una extensión hecha por René Berber [Berber86] al sistema introducido por Mordechai Ben-Ari [Ben-Ari82] como material de apoyo a uno de los primeros libros de texto dedicados exclusivamente a la programación concurrente. *Pascal-S* es el nombre de un subconjunto de *Pascal* definido por Nicklaus Wirth en [Wirth81]. Ben-Ari extendió el sistema de Wirth con la construcción COBEGIN...COEND para expresar la ejecución concurrente de los procedimientos encerrados por ésta, y con las operaciones sobre semáforos enteros como primitivas de sincronización. René Berber introdujo al sistema de Ben-Ari la posibilidad de definir recursos y las construcciones necesarias para accederlos utilizando regiones críticas condicionadas al estilo de Per Brinch Hansen [B.Hansen72] y de C.A.R. Hoare [Hoare72].

Este sistema constituyó la primera posibilidad de experimentación para los estudiantes del curso de programación concurrente que imparte la Dra. H. Oktaba [Oktaba85] en la maestría en ciencias de la computación de la U.A.C.P.y P. del C.C.H. con sede en el I.I.M.A.S. de la Universidad Nacional Autónoma de México.

El uso de *Pascal-S Concurrente* muestra cuán benéfico es el reforzar el aprendizaje de conceptos con prácticas. Aunque en su realización Berber escogió que las operaciones de entrada/salida fueran indivisibles, la difícil observación de los procesos reduce considerablemente estos beneficios. En efecto, al ejecutar un programa que utiliza la instrucción WRITELN(...) repetidamente para desplegar informaciones relevantes, se observa en la pantalla de la computadora un desfile secuencial de mensajes

que de alguna manera indican el estado del programa. Estas observaciones, si bien no dejan de ser informativas, tienden a inducir en el observador una imagen secuencial de la ejecución de su programa. Por ello, quedan reducidos los beneficios de *Pascal-S Concurrente* en cuanto a las posibilidades de observación ofrecidas.

Pascal-S Concurrente forma parte del grupo de sistemas que establecen la comunicación entre procesos mediante el uso de memoria compartida y, por lo tanto, excluye la experimentación con sistemas basados en el envío de mensajes y/o en la activación remota de procedimientos. También quedan excluidos los experimentos con el no determinismo explícito ya que el lenguaje no provee notaciones para ello. Finalmente, el número y la especie de procesos quedan fijados a tiempo de compilación, lo que constituye otra limitante para el experimentador.

1.3.3 Karel Concurrente

Karel Concurrente [Benveniste84] es una extensión castellana de *Karel el Robot* de Richard E. Pattis [Pattis81]. *Karel el Robot* es un sistema que se utiliza como preludio a *Pascal* en las primeras semanas de un curso de introducción a la programación de nivel licenciatura (en la universidad de Stanford, el material se cubre en 4 días). Por medio de un robot programable que se mueve en la pantalla de una terminal, los estudiantes obtienen sus primeras experiencias de programación. Según Pattis:

> " Los estudiantes son mucho más adeptos a diseñar y a visualizar los programas de *Karel*, que mueven a un robot a través de las calles de una ciudad, que hacerlo con programas de *verdad*, que mueven información a través de los circuitos de una computadora."[1]

En *Karel Concurrente*, se retoma la idea de observar directamente la ejecución de un programa y se le agrega la posibilidad de tener a varios robots. Esto queda reflejado en el lenguaje de programación del sistema con dos construcciones: EJ..{/..}FIN y CUA<COND><LISTAINST>LG. La primera construcción permite especificar la ejecución concurrente de varios procesos separados por /; la segunda permite instruir que la lista de instrucciones <LISTAINST> se ejecute al cumplirse la condición especificada por <COND>, acaparando en forma exclusiva todos los recursos del sistema.

Karel Concurrente fue empleado en el mismo curso que *Pascal-S Concurrente*. Sus tres principales ventajas son: la observación directa de los procesos pro-

[1] En la sección 3.2.2, se exponen algunas razones que apoyan esta aseveración.

gramados, la simplicidad de sus conceptos y las grandes posibilidades de experimentación que ofrece. No se ahonda aquí en la exposición de éstas ya que *LPC* goza de bondades similares, descritas en el capítulo 3. Entre sus muchas debilidades, se pueden mencionar las siguientes:

- el concepto de proceso no está claramente definido en el sistema; tan es así que el lenguaje no exige que los procesos sean especificados como unidades referibles;
- el sistema carece de un formalismo que permita probar propiedades de los tareas con él programadas. No se dió una definición formal de la semántica del lenguaje de *Karel Concurrente*;
- el sistema no ofrece mecanismos para especificar explícitamente el no determinismo, no proporciona comunicaciones entre procesos que no se basen en el uso de memoria compartida e imposibilita la creación dinámica de procesos;
- al ejecutar un programa, el usuario no puede modificar interactivamente parámetros del sistema (tales como las velocidades de los procesadores), limitando así varios experimentos.

Karel Concurrente proporcionó sin embargo un punto de vista totalmente novedoso en el ámbito del estudio de la programación concurrente y constituye un innegable anteproyecto del presente.

1.3.4 Gus

Gus es el nombre de un sistema educativo para el estudio de las tareas de *Ada*™ [DoD80] presentado por G. Lapalme y P. Chartray en [Lapalme87]. Los autores reconocen la importancia de la programación concurrente y la necesidad, para su enseñanza, de herramientas que muestren la actividad realizada por los distintos procesos de un programa. Considerando que *Ada* es uno de los lenguajes de programación más difundidos y habiendo experimentado las dificultades de observación que se subrayaron al hablar de *Pascal-S Concurrente*, los autores desarrollaron un sistema para auxiliar a sus estudiantes en la programación de tareas con *Ada*. El sistema se compone de un lenguaje de programación (subconjunto de *Ada*), un compilador, un intérprete y un post-procesador.

El intérprete se encarga de generar huellas de los eventos *importantes* de las tareas (como son la activación, la suspensión, la ejecución de un rendez-vous etc.) que se le hayan encomendado con la instrucción PRAGMA de *Ada*. Al terminar la ejecución de un programa en *Gus*, el post-procesador

genera dos reportes: un listado de las huellas de las tareas en el que se indica, para cada evento ocurrido, el tiempo físico en el que tuvo lugar y ciertas características de la tarea que lo generó; y una gráfica que presenta la misma información bajo otra forma.

Estas huellas pueden confundir al estudiante ya que derivan de una ejecución de su programa entre las muchas posibles. Una cronología particular de los eventos no dice mucho acerca de la lógica involucrada en el control del flujo de un programa ya que este control debe ser independiente del tiempo en que ocurren los eventos (salvo en algunas aplicaciones de tiempo real). El primer reto que enfrenta el novato de la programación concurrente es precisamente el que representa olvidarse de la secuencia temporal estricta y alcanzar el nivel de abstracción en el cual las instrucciones no están totalmente ordenadas en el tiempo[2]. De lo anterior, se desprende que una herramienta que tiende a ordenar totalmente en el tiempo procesos concurrentes no puede serle de gran ayuda al novato; tal vez hasta le perjudique.

El enfoque de *LPC* es diametralmente opuesto al de *Gus*: *LPC* presenta la concurrencia al novato simulando la ocurrencia simultánea de eventos en una pantalla.

1.4 Objetivos

En esta tesis, se plantea el diseño un sistema destinado a facilitar el aprendizaje de la programación concurrente. Se desea obtener una herramienta para estudiantes y profesionistas de la computación que les sirva de introducción a los múltiples sistemas existentes para el estudio y la práctica de esta disciplina.

Las propiedades que se buscan para *LPC* son:

☞ *grandes posibilidades de experimentación*:

la práctica de una disciplina durante su estudio es generalmente gratificante. La programación concurrente no constituye una excepción y la oportunidad de observar, cuantas veces sea necesario, errores y aciertos seguramente repercutirá en una comprensión más profunda de esta disciplina;

[2] Se está aludiendo a las definiciones matemáticas de orden total y orden parcial.

- *simplicidad conceptual*:

  un sistema basado en conceptos sencillos y claros permite presentar nuevas ideas nítidamente ya que la exposición de éstas no queda eclipsada por elementos que no son importantes en una fase introductora;

- *expresividad*:

  un sistema simple y sencillo no tiene porqué imponer restricciones demasiado severas en su capacidad expresiva;

- *gran capacidad para ejemplificar*:

  el contar con un buen ejemplo resulta muy importante para introducir conceptos. Un sistema que facilite el estudio de la programación concurrente debería proporcionar ejemplos de los conceptos fundamentales de esta disciplina;

- *herramientas formales de apoyo*:

  la experimentación es benéfica siempre que las observaciones que de ella se derivan puedan ser analizadas. Un sistema para el estudio de la programación concurrente supone pues un conjunto de herramientas formales;

- *facilidad de uso*:

  un sistema de difícil uso reduce considerablemente su utilidad.

2. Programación Concurrente

En este capítulo, se ofrece una breve presentación de las principales características de la programación concurrente y de los problemas que le dieran origen. Se subrayan la importancia y la utilidad de sus conceptos para la programación de sistemas actuales y venideros. Los sistemas que sirven para el estudio y/o la práctica de esta disciplina exigen toda una serie de prerrequisitos que tal vez no sean indispensables para su aprendizaje. Se exponen estas exigencias catalogando los sistemas de acuerdo a su género: modelos matemáticos, lenguajes de programación e híbridos de ambos. Así, se intenta establecer porqué la programación concurrente sigue siendo una especialidad.

2.1 Una breve presentación

Los escritores y los diseñadores de sistemas operativos han sido los pioneros de la programación concurrente. En efecto, al surgir la posibilidad de compartir una máquina entre varios usuarios, los autores de sistemas operativos tuvieron que solucionar nuevos problemas entre los cuales se pueden identificar los siguientes:

- *la exclusión mutua*, que consiste en garantizar el acceso de cierto proceso a cierto recurso en forma exclusiva. Este problema se presenta, por ejemplo, cuando dos o más procesos tratan de usar al mismo tiempo un disco duro o una impresora;
- *la sincronización condicionada*, que es la forma de lograr que dos o más procesos cooperen exitosamente en la ejecución de una tarea. Un paradigma de este problema es la sincronización de productores, que depositan sus productos en un almacén, y consumidores, que acuden a dicho almacén, de manera que ningún productor deposite en un almacén lleno y que ningún consumidor consuma de un almacén vacío;

- *el bloqueo mortal*, que consiste en un inter-bloqueo de dos o más de los procesos existentes en un sistema. La siguiente situación ejemplifica un bloqueo mortal. Un proceso, A, tiene asignado en exclusiva el acceso al recurso R_a, y espera accesar al recurso R_b. Pero el acceso al recurso R_b ha sido otorgado en forma exclusiva a otro proceso, B, quien, a su vez, espera ganar el acceso al recurso R_a;

- *la inanición y el acaparamiento*, que tienen que ver con la equidad de la distribución del derecho de uso de los recursos compartidos.

Al buscar soluciones sistemáticas, estos autores iniciaron la disciplina que estudia la "vida" de los procesos concurrentes. La creación, la activación, la destrucción, la comunicación y la sincronización de éstos requirieron de nuevas notaciones. La construcción y la verificación de sistemas de procesos concurrentes no pudieron llevarse a cabo utilizando las técnicas y las herramientas que proporciona la programación secuencial puesto que, a diferencia de los procesos secuenciales, el orden de ejecución de los procesos involucrados puede variar, bajo condiciones iniciales idénticas, de una ejecución a otra. La programación concurrente resulta de las investigaciones que se han realizado y de las experiencias que se han obtenido. Estas se encuentran ampliamente descritas en la literatura. Entre las principales obras, pueden consultarse las siguientes: [Andrews83], [Ben-Ari82], [Filman84], [Oktaba85] y [Oktaba87].

La programación concurrente proporciona pues notaciones, técnicas y herramientas teóricas para construir y verificar programas más generales que los secuenciales: aquéllos que sólo especifican un orden parcial de ejecución para sus instrucciones, en lugar del más restrictivo orden total requerido por los primeros.

2.2 Una disciplina de vanguardia

En los últimos quince años la programación concurrente ha cambiado substancialmente gracias a un conjunto de avances tecnológicos y teóricos. Logros tecnológicos, tales como procesadores baratos y poderosos, memorias gigantescas y herramientas sofisticadas de comunicación, han permitido que la computación solucione un sinnúmero de problemas en las más diversas áreas, con la construcción de sistemas de cómputo distribuido y de sistemas de multiproceso. Paralelamente, estudios teóricos han formulado nuevas

notaciones para la programación concurrente que expresan con sencillez sintáctica el paralelismo, hacen explícitos los requisitos de sincronización entre procesos y facilitan las demostraciones formales de propiedades de los programas. El advenimiento de las computadoras personales y de las redes ha desplazado el concepto de centro de cómputo, cambiándolo por el de estaciones de trabajo en un sistema distribuido, creando una creciente demanda de programas sumamente sofisticados y confiables para estos nuevos equipos. Los diseñadores, los programadores y los usuarios de estas nuevas generaciones de programas quedan confrontados, con la cercanía que les demandan sus respectivos papeles, al paralelismo y al no determinismo.

Además, todo parece indicar que los procesadores actuales están por alcanzar las máximas capacidades ofrecidas por la tecnología empleada. Uno de los medios más viables para obtener mayor poder de procesamiento tal vez sea el emplear a varios procesadores para realizar el trabajo que venía realizando uno solo. Esto supone que el trabajo pueda ser dividido y que exista una forma de coordinar las distintas aportaciones de los procesadores. La programación concurrente se presenta como un buen vehículo para expresar y analizar estas situaciones.

Así, la programación concurrente ha dejado de ser dominio exclusivo de los diseñadores y programadores de sistemas operativos, para convertirse en una disciplina importante para los autores de un conjunto de aplicaciones en expansión acelerada entre las cuales se pueden mencionar los sistemas para el manejo de bases de datos, los de banca electrónica, los de control por cómputo en tiempo real, los de automatización de oficinas, los de diseño y de manufactura por computadora, los sistemas expertos etc.

La literatura describe varios tipos de sistemas que se han desarrollado en rededor de la programación concurrente. Esta diversidad corresponde a los distintos enfoques adoptados en la solución de problemas provenientes de múltiples y variados dominios.

Para la gente de sistemas operativos, el control eficiente de la máquina y la confiabilidad del sistema son factores determinantes. En los sistemas resultantes de este enfoque, se ha puesto énfasis en la correspondencia entre los comandos del lenguaje y las primitivas disponibles en la máquina física, y en los mecanismos de estructuración y su facilidad de prueba. De este enfoque provienen lenguajes como *Procesos Distribuidos* [B.Hansen78] y *SR* [Andrews81].

En un enfoque pragmático, los lenguajes proporcionan instrucciones que a veces no son fácilmente realizables por la máquina pero que auxilian al programador, permitiéndole tratar la concurrencia con mayor nivel de abstracción. El lenguaje *PLITS* [Feldman79] (lenguaje de programación en el cielo) es un ejemplo de este enfoque.

El enfoque semántico ha generado varios modelos con los cuales se estudian los sistemas concurrentes mirándoles como objetos matemáticos. Los sistemas engendrados por este enfoque permiten la verificación formal de propiedades de programas concurrentes. *Procesos Concurrentes* (*CCS*) [Milner80] y *Redes de Petri* [Petri62] son dos sistemas derivados de esta corriente.

El enfoque analítico hace hincapié en las facilidades brindadas por el sistema para analizar la eficiencia de los algorítmos con él programados. *Variables Compartidas* [Lynch81] es un ejemplo de este enfoque.

No obstante la gran cantidad de sistemas existentes, destinados ya sea a la práctica de esta disciplina o a la formalización de sus conceptos, la programación concurrente todavía está reservada a especialistas.

2.3 Una especialidad

La mayoría de los sistemas existentes tienen características que se derivan de uno o de varios de los enfoques recién mencionados. Sin embargo, se pueden identificar tres géneros: los modelos de sistemas de programación, los lenguajes de programación y los híbridos de ambos.

Los modelos de sistemas de programación permiten explicar y analizar características de un sistema complejo, mediante la abstracción de aquellas propiedades que determinan su comportamiento. Algunos ejemplos de estos modelos son: *Variables Compartidas* [Lynch81], *Procesos Concurrentes* [Milner80] y *Programación Aplicativa Indeterminista* [Friedman79]. *Variables Compartidas* modela sistemas distribuidos usando procesos, variables compartidas y una primitiva que permite que un proceso lea, procese y escriba una variable en forma indivisible. *Procesos Concurrentes* proporciona un cálculo formal para establecer matemáticamente una semántica de procesos concurrentes comunicantes basado en la teoría de conjuntos, dominios potencia (*power domains*) y funciones. *Programación Aplicativa Indeterminista* generaliza los ambientes funcionales del estilo *Lisp* puro [McCarthy65] o *Scheme* [Steele78] libre de efectos laterales para transformarlos en sistemas distribuidos.

Los modelos constituyen una excelente herramienta para formalizar aspectos de la programación concurrente, pero olvidan generalmente su parte operacional ya que no fueron diseñados con ese propósito. En su mayoría, no proporcionan un lenguaje de programación con el cual manipularlos. La dificultad que presentan estas manipulaciones impiden tener una visión global del funcionamiento de los modelados, aún en casos sencillos, y reducen las posibilidades de experimentación. Además, es necesaria cierta madurez matemática para comprender y utilizar estos modelos. Por ello, se considera que los modelos son, en su mayoría, de difícil acceso para el estudio de la programación concurrente.

Los lenguajes de programación concurrente actuales surgen de necesidades específicas de expresión para programar procesos concurrentes. Algunos ejemplos de estos lenguajes son: *Procesos Distribuidos* [B.Hansen78], *PLITS* [Feldman79] y *SR* [Andrews81].

El diseño de *Procesos Distribuidos*, sucesor de *Pascal Concurrente* [B.Hansen75], fue fuertemente motivado por problemas de asignación de recursos y de sistemas de cómputo en tiempo real. Estos lenguajes se encuentran en el extremo *sistemas operativos* de la gama de enfoques; son lenguajes diseñados pensando en la programación de sistemas robustos, sujetos a varias limitaciones tanto temporales como de implementación. La encapsulación de datos, y una estricta tipificación de éstos, proporcionan en *Procesos Distribuidos* un buen apoyo en la realización de estos sistemas.

PLITS deriva mucho del enfoque pragmático. Se basa en procesos que se comunican a través de mensajes asíncronos. Los procesos contituyen un tipo de datos y se benefician de creación dinámica, de autoterminación y de chequeos de existencia. Tal vez el aspecto más distintivo de este lenguaje sea la sofisticación de su manejo de mensajes: éstos son estructurados y proveen mecanismos integrados de filtrado y de seguridad.

SR se basa en procesos y recursos. Los recursos son conjuntos de procesos que comparten memoria. Las comunicaciones pueden ser tanto síncronas como asíncronas y se realizan por medio de invocaciones remotas de procedimientos. Este lenguaje proviene de los enfoques pragmático y de sistemas operativos. El diseño de *SR* pone énfasis en el problema de despacho de tareas. Así, las llamadas pueden ser atendidas en base a prioridades fijadas por el estado del servidor y/o del solicitante.

No todos los lenguajes existentes tienen asociada una semántica formal que permita establecer propiedades de los programas con ellos escritos. Si bien esto ya se había dado en los lenguajes de programación secuencial,

las consecuencias en este nuevo ámbito son mucho más graves que aquéllas. En general, los sistemas concurrentes son sistemas que no terminan (piénsese en un sistema de reservaciones) a diferencia de los sistemas secuenciales cuyo propósito es realizar una tarea de procesamiento determinada. Ello significa que los errores de programación pueden surgir en cualquier momento durante todo el tiempo que esté funcionando el sistema. La depuración de procesos concurrentes es sumamente difícil, si no imposible, por la naturaleza netamente no determinista de estos sistemas; los errores de programación pueden pasar desapercibidos durante períodos largos y el día menos esperado, con los datos habituales y bajo las circunstancias de operación normales, el sistema falla. En programación secuencial, se emplean las instrucciones de entrada y de salida para observar determinados aspectos del sistema programado. En programación concurrente, la observación del sistema programado se convierte inmediatamente en otro problema de programación concurrente. En efecto, si se desea observar la operación del programa que realiza una solución del problema de los cinco filósofos [Dijkstra72], escrito por ejemplo en *Pascal-S Concurrente* [Berber86], se tiene que programar un proceso espía que escriba un conjunto de variables de observación (por ejemplo el número de tenedores y el estado de cada filósofo) en algún dispositivo de salida[1]. La programación de este proceso es un problema de concurrencia adicional, lo que invalida hasta cierto punto las observaciones que pueda proporcionar.

Los lenguajes de programación concurrente actuales permiten, cuando no están encerrados en laboratorios de investigación, practicar la programación de sistemas distribuidos con las herramientas adecuadas. Sin embargo, como se señalaba en la introducción, el aprender a usar un lenguaje de programación no basta por sí solo para comprender la disciplina; son necesarios modelos para entenderla cabalmente y herramientas formales para establecer propiedades de los sistemas que con su auxilio son programados.

Los sistemas híbridos conjuntan características de los dos géneros de sistemas recién descritos. Puede decirse que son lenguajes de programación con una formalización semántica que permite establecer propiedades de los sistemas que programan y, a la vez, que son modelos de sistemas de progra-

[1] En la sección 1.3.2, se hizo hincapié en que las instrucciones de entrada/salida fueron implementadas por Berber como acciones atómicas. Esta atomicidad evita la necesidad de un proceso espía, asegurando que los mensajes escritos por los filósofos sean desplegados sin interferencias. Si la implementación no garantizara la atomicidad, el usuario tendría que implementarla en sus procedimientos de entrada/salida, lo que aclara la discusión.

mación cuya operación es realizable por una computadora. *CSP* [Hoare78] y *Cell* [Silberschatz80] son dos representantes de sistemas híbridos. Los híbridos reunen los beneficios de los lenguajes y de los modelos.

El marco teórico que proporcionan estos sistemas satisface la necesidad de un formalismo para establecer propiedades, tales como la ausencia de bloqueos mortales, de las aplicaciones programadas; también permite expresar, de manera no ambigua, los conceptos básicos que encuentran notaciones en los lenguajes de programación que proporcionan. Con estos lenguajes, es posible una programación efectiva de sistemas distribuidos, cumpliendo de esa manera con una de las funciones de la programación concurrente.

Los sistemas híbridos son pues, los más convenientes para el aprendizaje y la práctica de la programación concurrente. No obstante su conveniencia, los híbridos existentes no solventan totalmente el difícil acceso a este estudio. La complicada observación del comportamiento operacional de los programas se deriva directamente de los lenguajes de programación de estos sistemas. Otra desventaja es que estos sistemas se encuentran generalmente en universidades y laboratorios bajo la forma de proyectos de investigación; la preparación requerida para poder estudiarlos y usarlos constituye otra limitación. Estos inconvenientes reducen considerablemente el número de estudiantes y de profesionistas que pueden aprender la programación concurrente.

Las restricciones impuestas por los sistemas existentes para el estudio de la programación concurrente se pueden resumir para cada una de las dos vertientes, modelos y lenguajes[2], como sigue.

Los modelos existentes restringen el acceso al estudio de esta disciplina de dos maneras. La primera proviene naturalmente del bagaje matemático indispensable para comprender y usar los modelos propuestos. *Procesos Concurrentes* (*CCS*) es un buen ejemplo de ello[1]. La segunda estriba en que la comprensión de estos modelos no implica, por sí sola, la buena práctica de la disciplina estudiada. Se requiere de posibilidades de experimentación que difícilmente ofrecen los modelos.

Los lenguajes actuales limitan el estudio de la programación concurrente ya que la práctica no basta para comprender esta disciplina. A esta nueva programación le son aún más necesarias las herramientas formales de apoyo. Los sistemas híbridos las suministran pero, junto con ellas, aportan las

[2] En general, las limitantes de los sistemas híbridos se derivan de aquéllas.

[1] Véanse los comentarios acerca del sistema Clara en el último capítulo.

limitaciones que se señalaron en el párrafo anterior. El uso de los lenguajes de programación concurrente disponibles también presenta dificultades. En efecto, en su mayoría suponen el estudio de la programación secuencial. Los tipos de datos, las instrucciones secuenciales etc. parecen constituir un bagaje mínimo indispensable al estudiante que desee usarlos. Estos conocimientos tal vez pudieran ser adquiridos mediante el estudio de la programación concurrente.

Así, quedan delineadas las principales razones por las cuales la programación concurrente sigue siendo una especialidad cuando la tendencia en el tipo de equipos y sistemas de cómputo parece reclamar que ésta sustituya a la secuencial. Esto es lo que se pretende subsanar, en parte al menos, con *LPC*.

3. Diseño de LPC

En este capítulo, se presentan las dos grandes etapas del diseño de *LPC*. La primera establece las premisas en las cuales se apoya *LPC* considerando los objetivos apuntados en la sección 1.4. La segunda es la edificación del universo semántico en el cual tomará significado el lenguaje de programación de *LPC*, intentando satisfacer las premisas de la primera etapa.

3.1 Las premisas

Los objetivos de este trabajo se centran en la simplicidad conceptual aunada a la riqueza de expresión. Como se apuntó en el capítulo anterior, el concepto de proceso tiene un papel fundamental en la programación concurrente. Resulta entonces importante buscar un entorno que facilite su comprensión. La riqueza de expresión supone que el entorno buscado permita una representación de las múltiples situaciones que han dado origen a la programación concurrente. También supone que los mecanismos y notaciones de los sistemas existentes tengan su contraparte en este nuevo marco referencial.

3.1.1 La observación directa de los procesos

No existe una definición única de proceso en la literatura. En el ámbito de sistemas operativos, un proceso es un programa en ejecución. G.R.Andrews y F.B.Schneider [Andrews83] y H.Oktaba [Oktaba85] coinciden en definir un proceso secuencial como la ejecución [secuencial] de la lista de instrucciones especificada en un programa. Ambos autores definen el concepto de procesos concurrentes como la ejecución simultánea de dos o más procesos secuenciales en un mismo sistema. R.E.Filman y D.P.Friedman [Filman84] describen un proceso como un procesador lógico (o virtual) que ejecuta un programa, tiene memoria pero no está ligado a ningún objeto físico en particular. De las anteriores definiciones se puede sintetizar que un proceso tiene dos componentes: un programa y un agente que, al ejecutar

el programa, lo convierte en proceso. Como se apuntó en la sección 1.3.1, C.A.R.Hoare [Hoare83] no marca diferencia entre el proceso y el agente; para él, ambos se pueden definir como el patrón de operación de un objeto, descrito en términos de un conjunto limitado de eventos relevantes. Para R.Milner [Milner80], un proceso queda definido por sus capacidades para establecer comunicaciones; es un objeto matemático que se *regenera* con las comunicaciones que establece; para estos autores, tampoco tiene relevancia quién exhiba el comportamiento de un proceso.

Cuando se habla de proceso, se puede hacer referencia a una serie dada de sucesos o bien al procedimiento que los genera. Por ejemplo, se habla del proceso de pacificación centroamericana al mencionar la serie de sucesos históricos que han llevado a la mesa de negociaciones a las distintas partes en conflicto; pero se entiende por proceso de fabricación del pan al conjunto de procedimientos necesarios a la elaboración del mismo.

Este doble significado causa confusión en programación concurrente. Para evitar en lo posible tal confusión, se ha adoptado en *LPC* una definición de proceso que identifica claramente, en la ejecución de un programa, qué es lo que se está ejecutando (la tarea) y quién lo está haciendo (el agente). Un proceso es, pues, el patrón de operaciones que exhibe una pareja tarea-agente, descrito en términos de un conjunto limitado de eventos relevantes.

Una de las principales dificultades encontradas en el aprendizaje de la programación concurrente es sin duda la intangibilidad de los procesos. Observar un proceso es tan intangible como mostrar la ejecución de un programa. Se sabe que un programa se está ejecutando, se pueden "palpar" sus resultados, mas no se ve cómo se lleva a cabo. Si se consideran, por un lado, una computadora y su conjunto de procesos y, por el otro, una caja negra que transforma círculos en triángulos, se puede decir que las entradas observables de la computadora son análogas a los círculos de la caja negra, que las salidas observables lo son a los triángulos y que los procesos en la computadora se asemejan a lo que sucede dentro de la caja. Eso es lo que se quiere observar. Se quiere realizar en la computadora lo aná-

logo a abrir la caja negra y observar las transformaciones mientras éstas se llevan a cabo[1].

De los sistemas que permiten, intrínsecamente, la observación directa de procesos, *Logo* [Papert80] y *Karel* [Pattis81] son sin duda los más importantes[2]. En ambos sistemas, un procesador virtual es representado en la pantalla de una terminal por un objeto: la tortuga de *Logo* y el robot de *Karel*. Un programa es la descripción de un patrón de conducta de ese objeto. La ejecución de un programa es la conducta que exhibe el objeto actuando sus instrucciones. Así, lo que se observa en la pantalla cuando el procesador ejecuta un programa es precisamente un proceso, en el sentido que recién se ha adoptado.

Aunque en *Logo* y *Karel* se pueden observar los procesos, *Karel* ofrece mayores posibilidades de observación. El mundo, los trompos y los muros de *Karel* forman una estructura de datos en la que se codifica en forma homogénea y directamente observable, aunque muy primitiva, la información de las entradas, de las salidas y del medio ambiente. *Logo* carece de este concepto de "mundo cerrado" asociado a la pantalla; la tortuga tiene interacciones con otros periféricos, como son el teclado y la impresora, y su interacción con la pantalla es unilateral ya que las gráficas trazadas carecen de significado: la existencia de un punto en algún sitio de la pantalla es indiscernible tanto para la tortuga como para su programador. Por ello, *Karel* y su mundo constituyen los primeros ladrillos de la construcción del universo semántico de *LPC*.

3.1.2 La capacidad expresiva

Para que *LPC* sea un paradigma de los sistemas de programación concurrente, es necesario que permita expresar conceptos y notaciones presentes en los sistemas existentes. Robert Filman y Daniel Friedman han definido

---

[1] Cabe notar que este deseo de apertura se encuentra también en la evolución de los equipos de cómputo; una de las diferencias entre las máquinas de antaño y las microcomputadoras multi-ventanas de hogaño, la constituye precisamente esta apertura; en efecto, las primeras transformaban tarjetas perforadas en listados y las últimas muestran en la pantalla los cambios ocasionados por el movimiento de un ratón que se comporta como una extensión de la mano.

[2] Cuando se llevó a cabo la revisión bibliográfica esta aseveración era correcta. Sin embargo, han sido publicados, recientemente, una gran variedad de artículos acerca de sistemas que suministran una observación directa de sus procesos; destaca el artículo que describe al sistema Clara [Giacalone88] (consúltese el último capítulo de la tesis).

una serie de dimensiones en las cuales éstos pueden ser comparados [Filman84]. Han especificado un sistema "ideal" que engloba a los demás seleccionando, en cada dimensión, aquella medida que fuere la más primitiva, o sea, aquélla que permitiera expresar a las demás. Lo mismo se ha hecho para *LPC*. A continuación, se exponen estas dimensiones y se describen las decisiones tomadas.

3.1.2.1 Procesos explícitos

La presente dimensión considera la forma en la que se expresan las acciones que se pueden ejecutar en paralelo. La mayoría de los sistemas usan procesos explícitos para este propósito. En *CSP* [Hoare78], el proceso se llama "proceso" (process); en *Ada*® [DoD80], el proceso se llama "tarea" (task); "módulo" (module) en *PLITS* [Feldman79]. Existen sin embargo tres sistemas que no usan procesos explícitos: *Redes de Petri*, *Flujo de Datos* y *Programación Aplicativa Indeterminista*. Las *Redes de Petri* (Petri Nets) [Petri62] [Peterson77] se pueden simular con procesos explícitos teniendo un proceso para cada *lugar* y para cada *transición*. Los sistemas que se basan en *Flujo de Datos* (Data Flow) [Dennis74] se simulan teniendo un proceso explícito para cada *actor*. Para modelar la *Programación Aplicativa Indeterminista* (*IAP*) [Friedman79] con procesos explícitos, es necesaria la creación de un proceso en cada invocación de *cons* y de *frons*.

Es deseable que *LPC* cuente con procesos explícitos.

3.1.2.2 Creación dinámica de procesos

Se dice que un sistema tiene procesos estáticos si el número y la variedad de éstos no pueden modificarse durante la ejecución del programa que los usa; en caso contrario, se dice que el sistema ofrece creación dinámica de procesos. *Procesos Distribuidos* y *SR* tienen procesos estáticos mientras que *Ada*, *PLITS* e *IAP* ofrecen creación dinámica de procesos. La creación dinámica proporciona mayores posibilidades de experimentación, favorece una programación más flexible y permite el estudio de situaciones complejas que se presentan en la realización de varios sistemas. Es conveniente que *LPC* disponga de procesos que puedan variar dinámicamente. Además, la creación dinámica permite expresar situaciones que acontecen en *Redes de Petri* e *IAP* como se indicó en el punto anterior.

3.1.2.3 Comunicación asíncrona

Esta dimensión concierne al tipo de sincronización que utilizan los procesos para comunicarse. Las comunicaciones pueden ser síncronas o asíncronas. Se dice que dos procesos comunican en forma síncrona si ambos atienden la comunicación al mismo tiempo. Un proceso que ha iniciado una comunicación síncrona no puede empezar otra sin haber concluido la primera. Se entiende por comunicación conclusa aquélla en la que el mensaje ha sido emitido y recibido. *CSP*, *Procesos Distribuidos* [B.Hansen78], *Procesos Concurrentes* [Milner80], *SR* [Andrews81], *Ada* y *Cell* [Silberschatz80] son algunos de los sistemas que utilizan este tipo de comunicación. La comunicación asíncrona permite la emisión de mensajes sin la comparecencia del receptor. El proceso emisor puede mandar un mensaje y proseguir su computación sin preocuparse por la recepción del mismo. *Variables Compartidas* [Lynch81], *Redes de Petri* [Petri62], *Flujo de Datos*, *PLITS* y *SR* son sistemas que usan este tipo de comunicación. La comunicación síncrona se puede simular con asíncrona por medio de protocolos que bloqueen al emisor hasta que el receptor concluya la comunicación. La asíncrona se puede simular con síncrona creando un proceso secretario responsable de cada mensaje que se desee enviar. Si se asemejan las comunicaciones asíncronas a los principios del correo y las síncronas a los del teléfono, puede decirse que las primeras son más primitivas que las segundas.

Basándose en la subjetiva analogía recién descrita, las comunicaciones en *LPC* se escogen asíncronas.

3.1.2.4 Comunicación unidireccional

El flujo de información en las comunicaciones entre procesos puede ser unidireccional o bidireccional. Siendo el flujo unidireccional una necesidad para la comunicación asíncrona, se le requiere en *LPC*.

3.1.2.5 Capacidad de almacenaje ilimitada

La cantidad de mensajes que un sistema puede tener pendientes se conoce como su capacidad de almacenaje. Existen sistemas con capacidad ilimitada, como *Redes de Petri*, *PLITS* y *Actors* [Hewitt77], y otros con capacidad limitada, como *Procesos Distribuidos*, *Ada* y *Variables Compartidas*. La capacidad limitada se puede simular con capacidad ilimitada. La inversa no es siempre cierta. Por ello, se prefiere una capacidad ilimitada para *LPC*. Cabe notar que

las limitaciones de memoria acotan forzosamente la capacidad de cualquier realización.

3.1.2.6 Comunicación por nombre y por buzón

Los distintos medios de comunicación (la televisión, la radio, los periódicos, el correo, las pizarras de anuncios, el teléfono etc.) tienen, cada cual, características particulares de operación que determinan tanto las posibles topologías de comunicación como las propiedades temporales de las mismas. Una pizarra se emplea, en general, para comunicaciones de tipo uno a muchos. *Variables Compartidas* se apoya en la metáfora de la pizarra. El teléfono favorece más bien las de tipo uno a uno. Si los estados del transmisor y del receptor en los cuales se establece una comunicación son importantes para la misma, es necesario emplear un medio como el teléfono; es decir un medio síncrono en el que los estados de los cuales se hable en la comunicación puedan corresponder a los actuales del receptor y del transmisor. *CSP* y *Procesos Concurrentes* adoptan medios que imitan al teléfono. *PLITS* y *Actors* utilizan medios que se parecen al correo. El correo es un medio típicamente asíncrono que ofrece tanto comunicaciones de tipo uno a uno (usando los nombres del destinatario y del remitente), como las de tipo muchos a muchos (usando buzones en el sentido de la sección 2.1.2), por ello se le escoge como medio de comunicación para *LPC*.

El universo semántico de *LPC* toma pues la observación directa de procesos de *Karel* y su mundo, proporciona procesos explícitos, soporta la creación dinámica de estos últimos y ofrece comunicación unidireccional y asíncrona con capacidad de almacenaje ilimitada, por medio de buzones y de nombramiento directo.

3.2 El universo semántico de LPC

La construcción de este universo retoma los conceptos del mundo y de las acciones de *Karel* [Pattis81], y extiende estas nociones a varios robots. Esta generalización se hace agregando los elementos necesarios al establecimiento de las premisas de la sección3.1, y sirve así a la consecución de los objetivos de esta tesis.

Siendo *LPC* un sistema híbrido de modelo de sistema de programación y de lenguaje de programación, es necesario que su lenguaje cuente con una

semántica formal que permita establecer propiedades de los sistemas con él programados. La construcción de su universo semántico debe pues ser formal. Aunque *Karel* fue publicado hace ya más de siete años, no se reporta en la literatura formalización alguna de este sistema, y aún en publicaciones recientes ([Bajar86] y [Rising84]), se le describe informalmente. Una presentación informal de conceptos tiene desventajas bien conocidas y provoca en general malentendidos.

Para evitar en lo posible semejantes resultados, se reconstruyen el mundo y el robot de Karel definiendo los objetos matemáticos requeridos. Con este fin, se utilizan conjuntos y sus operaciones habituales, relaciones, secuencias y funciones. Este formalismo se deriva del lenguaje de especificación Z [Sufrin86] del cual no se emplea la notación de esquemas.

3.2.1 El mundo

El mundo de *LPC* constituye una cuadrícula formada por un conjunto de calles paralelas de orientación Este-Oeste, y por un conjunto de avenidas perpendiculares a las calles, de orientación Sur-Norte. Sin pérdida de generalidad, se puede asumir que el ancho y el largo de las manzanas son unitarios. Siendo N el conjunto de los enteros, se modelan calles y avenidas con los conjuntos:

M.1 $CALLE = N$
M.2 $AVENIDA = N$

Se le llama cruce a la intersección de una calle con una avenida. Las intersecciones de la cuadrícula se modelan con el conjunto de los cruces definido por:

M.3 $CRUCE = AVENIDA \times CALLE$

Sobre la cuadrícula transita un robot que se define en la sección 3.2.2. Por ahora, basta saber que un robot ocupa un cruce al que se llama su posición. Siendo bidimensionales los mundos, un determinado cruce sólo puede ser ocupado por un robot a la vez: no pueden estar en el mismo cruce dos o más robots al mismo tiempo.

Sea un predicado sobre el conjunto de cruces que es verdadero (V) si un cruce determinado está ocupado por un robot y que, de otro modo, es falso (F):

M.4 Ocupado?: $CRUCE \longmapsto BOOLEANO$

$$x \quad \dashrightarrow \quad \begin{cases} V \text{ si } x \text{ es la posición de un robot} \\ F \text{ en caso contrario.} \end{cases}$$

En la notación empleada, aparece el nombre de la relación que se está definiendo, en este caso Ocupado?, seguido por su funcionalidad. Ésta se denota con los conjuntos que relaciona, separados por $\longmapsto$. La siguiente línea establece las reglas de evaluación de la relación. En este caso, al cruce x se le asocia V si es la posición de algún robot y F si no la es.

El predicado Ocupado? agrupa a los cruces en dos conjuntos complementarios: el conjunto de los cruces que lo cumplen y el que forman el resto de los cruces. Considérese la relación de equivalencia, *MISMO_ESTADO*, que clasifica a los cruces de acuerdo a su estado de ocupación: ocupados o libres. Se denota esta relación con el conjunto:

M.5 $MISMO\_ESTADO = \{ (x,x') \in CRUCE \times CRUCE \mid Ocupado?(x) = Ocupado?(x') \}$

El conjunto cociente $(/)$[1] de *CRUCE* para la relación *MISMO_ESTADO* se compone de los conjuntos complementarios que se mencionaron en el párrafo anterior. Si estos conjuntos se denotan con OCUPADO y LIBRE, la definición del conjunto, *ESTADO*, de los estados de un cruce queda como sigue:

M.6 $ESTADO = CRUCE \,/\, MISMO\_ESTADO = \{$ OCUPADO , LIBRE $\}$

Se le llama *mundo* a una función que le asocia a cada cruce su estado: OCUPADO o LIBRE. El conjunto, *MUNDO*, de estas funciones se define pues como sigue:

M.7 $MUNDO = CRUCE \longmapsto ESTADO$

$$x \quad \dashrightarrow \quad \begin{cases} \text{OCUPADO si } x \in \text{OCUPADO} \\ \text{LIBRE de otro modo.} \end{cases}$$

Se forma una estructura, o tipo de dato abstracto (véase [Wand80]), con el conjunto *MUNDO*, dos funciones PonRobot y QuitaRobot (para cambiar los es-

[1] El conjunto cociente de un conjunto A y una relación de equivalencia R, es el conjunto de las clases de equivalencia formadas en A con R.

tados de los cruces) y el predicado HayRobot (para obtener el estado de cualquier cruce).

M.8 PonRobot: $MUNDO \times CRUCE \longmapsto MUNDO$

$$(m\ ,x) \rightarrow \begin{cases} m \cup \{x \rightarrow \text{OCUPADO}\} & \text{si } m(x) = \text{LIBRE} \\ m & \text{si } m(x) = \text{OCUPADO}, \end{cases}$$

donde $\cup$ es la operación de reescritura de relaciones. Una pareja (x,y) de una relación se denota por x -> y (-> se lee "...asocia..."). La operación de reescritura sirve tanto para agregar nuevas parejas a una relación como para alterar una asociación ya presente.

Por ejemplo, sea R = {a -> 1,b -> 2}, entonces:

R $\cup$ {c -> 3} = {a -> 1,b -> 2,c -> 3}, y

R $\cup$ {b -> 3} = {a -> 1,b -> 3}.

M.9 QuitaRobot: $MUNDO \times CRUCE \longmapsto MUNDO$

$$(m\ ,x) \rightarrow \begin{cases} m \cup \{x \rightarrow \text{LIBRE}\} & \text{si } m(x) = \text{OCUPADO} \\ m & \text{si } m(x) = \text{LIBRE}. \end{cases}$$

M.10 HayRobot: $MUNDO \times CRUCE \longmapsto BOOLEANO$

$$(m\ ,x) \rightarrow \begin{cases} \text{V} & \text{si } m(x) = \text{LIBRE} \\ \text{F} & \text{si } m(x) = \text{OCUPADO}. \end{cases}$$

Para identificar a cada uno de estos *mundos*, se les asocia un nombre único por medio de las declaraciones de mundos:

M.11 $\mathbf{DEC}MUN = \mathbf{ID}MUN \Longleftrightarrow MUNDO.$

Así, se puede definir una función para crear nuevos *mundos* como sigue:

M.12 CreaMundo: $\mathbf{DEC}MUN \times \mathbf{ID}MUN \longmapsto \mathbf{DEC}MUN$

$$(DeclM, IdM) \rightarrow DeclM \cup \{IdM \rightarrow MundoVacio\},$$

donde *MundoVacio* es aquel mundo en el que se cumple para todo cruce x, $MundoVacio(x)$ = LIBRE.

En la siguiente sección, se enriquece la estructura de los mundos recién formada, con un primer robot.

3.2.2 Un primer robot

Se le llama robot a un objeto que ocupa en forma exclusiva un cruce en algún mundo. Con esta primera definición, un robot queda completamente caracterizado por la pareja formada por el identificador del mundo en el que habita y el cruce que en éste ocupa:

R.1 *ROBOT* ≡ IDMUN × OCUPADO ⊆ IDMUN × *CRUCE*.

Un robot es capaz de desplazarse en las cuatro direcciones de su mundo, pasando del cruce que ocupa a uno de los cuatro aledaños siempre y cuando éste pertenezca a LIBRE. A continuación, se establecen algunas funciones auxiliares a la definición de los desplazamientos de un robot.

			CRUCE	⟹	*CRUCE*
R.2	Norte	:	(a, c)	↦	$(a, c+1)$
R.3	Este	:	(a, c)	↦	$(a+1, c)$
R.4	Sur	:	(a, c)	↦	$(a, c-1)$
R.5	Oeste	:	(a, c)	↦	$(a-1, c)$

$$\text{donde } n \perp 1 \equiv \begin{cases} n-1 & \text{si } n \geq 1 \\ 0 & \text{si } n = 0. \end{cases}$$

Al conjunto formado por estas cuatro funciones se le llama *DIRECCION*.

R.6 *DIRECCION* ≡ { Norte , Este , Sur , Oeste }.

Los desplazamientos de un robot quedan entonces expresados con la siguiente función:

R.7 Avanza: DEC*MUN* × *ROBOT* × *DIRECCION* ⟹ DEC*MUN* × *ROBOT*

$(DeclM, r, d)$ ↦ $(DeclM', r')$.

Sean *r.IdM* el identificador del mundo en el que habita el robot *r*, *r.x* su posición y *m* el mundo asociado a *r.IdM* en la presente declaración de mundos *DeclM*. Entonces la declaración *DeclM'* y el robot *r'* resultantes de aplicar la función Avanza quedan definidos como sigue:

si HayRobot$(m,\ d(r.x))$

$DeclM' = DeclM \cup \{\ IdM \mapsto \text{PonRobot}(\text{QuitaRobot}(m, r.x), d(r.x))\ \}$ y

$r' = (IdM, d(r.x))$

de otra manera: $DeclM' = DeclM$ y $r' = r$.

3.2.3 Los trompos y los muros

¿Qué tareas realizarán los robots de *LPC* si sólo saben desplazarse? Resulta difícil imaginar tareas interesantes ya que toda la información se tiene que manipular codificada en las posiciones del robot[4]. Hace falta introducir algún concepto nuevo para facilitar la expresión de tareas en *LPC*. Con este propósito se introducen los trompos y los muros al universo semántico de *LPC*.

Los trompos son objetos que los robots pueden manipular en sus mundos. Los robots logran detectar su presencia un cruce a la redonda; son capaces de cogerlos, de guardarlos y de dejarlos; son objetos acumulables en cualquier cruce. Se modelan con el conjunto de los enteros:

M.13 $TROMPO = N$

Los muros ocupan en forma exclusiva el cruce en el que se les coloque e impiden el paso de los robots por dicho cruce. Los robots no tienen control alguno sobre los muros; éstos son elementos inclusos durante la construcción del mundo con la ayuda de un editor de mundos. Los robots detectan su presencia un cruce a la redonda. Estos nuevos objetos inducen algunas modificaciones en las definiciones de *MUNDO* que a continuación se detallan.

Un cruce puede estar ocupado ya sea por un robot o bien por un muro. Resulta importante para lo que sigue poder distinguir el tipo de ocupante de un cruce. Por ello, se modifica la partición del conjunto *CRUCE* que se describió en M.6. En vez de utilizar al predicado Ocupado?, se recurre a la siguiente función:

---

[4] Para poder calcular cualquier función computable con los elementos con los que cuenta *LPC* hasta este punto, se tendría que suministrar un lenguaje para programar al robot que admita procedimientos recursivos. El robot, con este lenguaje, podría simular una máquina determinística de dos contadores, que es equivalente a una de Turing (véase [Kain72]); uno de los contadores se simularía con la abscisa de la posición del robot y el otro con su ordenada.

M.14 Status: $CRUCE \longmapsto \{ libre, robot, muro \}$

x	$\rightarrow$	*robot*	si en x hay un robot
x	$\rightarrow$	*muro*	si en x hay un muro
x	$\rightarrow$	*libre*	de otro modo.

Así, se obtiene un nuevo conjunto *ESTADO*, que corresponde al conjunto cociente de *CRUCE* para la relación *MISMO_ESTADO* (definida en M.5) redefinida como sigue:

M.15 $MISMO\_ESTADO \equiv \{ (x,x') \in CRUCE \times CRUCE \mid Status(x) = Status(x') \}$.

M.16 $ESTADO \equiv CRUCE \,/\, MISMO\_ESTADO \equiv \{ \text{LIBRE} , \text{ROBOT} , \text{MURO} \}$.

Con estos nuevos conjuntos y con la introducción de los trompos, es menester replantear las definiciones que se habían dado en la sección 3.2.1. Asimismo, es necesario establecer las funcionalidades de los muros y de los trompos.

Un *mundo* es una función que a cada cruce le asocia su estado de ocupación y el número de trompos que ahí tenga acumulados.

M.17 $MUNDO \equiv CRUCE \longmapsto ESTADO \times TROMPO$

Si $x \in$ MURO	x	$\rightarrow$	(MURO , t)
Si $x \in$ ROBOT	x	$\rightarrow$	(ROBOT , t)
de otro modo	x	$\rightarrow$	(LIBRE , t).

Además de ajustar las definiciones M.8, M.9 y M.10, se introducen funciones para la inserción y para la remoción de muros y trompos, y predicados para probar la existencia de éstos.

M.18 PonRobot: $MUNDO \times CRUCE \longmapsto MUNDO$

$$(m , x) \rightarrow \begin{cases} m \cup \{x \rightarrow \text{ROBOT}\} & \text{si } m(x) = \text{LIBRE} \\ m & \text{de otro modo.} \end{cases}$$

M.19 QuitaRobot: $MUNDO \times CRUCE \longmapsto MUNDO$

$$(m , x) \rightarrow \begin{cases} m \cup \{x \rightarrow \text{LIBRE}\} & \text{si } m(x) = \text{ROBOT} \\ m & \text{de otro modo.} \end{cases}$$

M.20 HayRobot: $MUNDO \times CRUCE \longmapsto BOOLEANO$

$$(m , x) \rightarrow (m(x) = \text{ROBOT}).$$

M.21 PonMuro: $MUNDO \times CRUCE \longmapsto MUNDO$

$$(m\ ,x) \dashrightarrow \begin{cases} m \cup \{x \dashrightarrow \text{MURO}\} & \text{si } m(x) = \text{LIBRE} \\ m & \text{de otro modo.} \end{cases}$$

M.22 QuitaMuro: $MUNDO \times CRUCE \longmapsto MUNDO$

$$(m\ ,x) \dashrightarrow \begin{cases} m \cup \{x \dashrightarrow \text{LIBRE}\} & \text{si } m(x) = \text{MURO} \\ m & \text{de otro modo.} \end{cases}$$

M.23 HayMuro: $MUNDO \times CRUCE \longmapsto BOOLEANO$

$$(m\ ,x) \dashrightarrow (m(x) = \text{MURO}).$$

M.24 PonTrompo: $MUNDO \times CRUCE \longmapsto MUNDO$

$$(m\ ,x) \dashrightarrow m \cup \{x \dashrightarrow (m(x).e\ ,\ m(x).t+1)\}.$$

M.25 QuitaTrompo: $MUNDO \times CRUCE \longmapsto MUNDO$

$$(m\ ,x) \dashrightarrow m \cup \{x \dashrightarrow (m(x).e\ ,\ m(x).t\perp 1)\}.$$

M.26 HayTrompo: $MUNDO \times CRUCE \longmapsto BOOLEANO$

$$(m\ ,x) \dashrightarrow (m(x).t > 0).$$

M.27 CreaMundo: $\mathbf{DEC}MUN \times \mathbf{ID}MUN \longmapsto \mathbf{DEC}MUN$

$$(DeclM, IdM) \dashrightarrow DeclM \cup \{IdM \dashrightarrow MundoVacio\},$$

donde *MundoVacio* es aquel mundo en el que se cumple para todo cruce x, $MundoVacio(x) = (\text{LIBRE}\ ,0)$.

También resulta necesario modificar las definiciones de *ROBOT* (R.1) y de Avanza (R.7). En efecto, ahora un robot tiene una bolsa en la que transporta sus trompos. Ya que se les ha otorgado a los robots la capacidad de detectar muros y trompos un cruce a la redonda, cobra importancia la dirección hacia la cual están orientados. Así, un robot queda determinado por el nombre del mundo en el que habita, el cruce que ocupa, la dirección hacia la cual está orientado y el número de trompos que va cargando. Expresando esto en notación de conjuntos obtenemos:

R.8 $ROBOT \subseteq \mathbf{ID}MUN \times CRUCE \times DIRECCION \times TROMPO.$

Antes de establecer los capacidades de un robot, se define el conjunto *DIRREL* de las direcciones relativas en los mundos. Resultan ser las cuatro biyecciones definibles sobre el conjunto *DIRECCION* (R.6):

R.9 $DIRREL \blacksquare DIRECCION \langle\!\Longrightarrow\!\rangle DIRECCION \blacksquare \{\underline{Proa}, \underline{Estribor}, \underline{Popa}, \underline{Babor}\}$

donde las direcciones relativas son:

R.10 $\underline{Proa} \blacksquare \underline{Identidad}(DIRECCION)$

R.11 $\underline{Estribor} \blacksquare \{\underline{Norte} \leftrightarrow \underline{Este}, \underline{Este} \leftrightarrow \underline{Sur}, \underline{Sur} \leftrightarrow \underline{Oeste}, \underline{Oeste} \leftrightarrow \underline{Norte}\}$

R.12 $\underline{Popa} \blacksquare \{\underline{Norte} \leftrightarrow \underline{Sur}, \underline{Este} \leftrightarrow \underline{Oeste}\}$

R.13 $\underline{Babor} \blacksquare \underline{Estribor}^{-1}$,

donde f^{-1} denota la función inversa de f.

Se pueden entonces caracterizar las acciones del robot como sigue:

R.14 Avanza: $\mathbf{DEC}MUN \times ROBOT \longmapsto \mathbf{DEC}MUN \times ROBOT$

$(DeclM, r) \rightarrow (DeclM', r')$

Siendo *r.IdM* el identificador del mundo en el que habita el robot *r*, *r.x* su posición, *r.d* su dirección y *m* el mundo asociado a *r.IdM* en la presente declaración de mundos *DeclM*, la declaración *DeclM'* y el robot *r'* resultantes de aplicar la función Avanza quedan definidos como sigue[5]:

si $\neg\ \underline{HayRobot}(m,\ r.d(r.x))\ \&\ \neg\ \underline{HayMuro}(m,\ r.d(r.x))$

$DeclM' = DeclM \cup \{\ r.IdM \rightarrow \underline{PonRobot}(\underline{QuitaRobot}(m, r.x), r.d(r.x))\ \}$ y

$r' = (r.IdM, r.d(r.x), r.d, r.t)$

de otra manera: $DeclM' = DeclM$ y $r' = r$,

R.15 Gira: $ROBOT \times DIRREL \longmapsto ROBOT$

$(r, dr) \rightarrow r' = (r.IdM\ ,\ r.x\ ,\ rd(r.d), r.t)$.

Estas dos funciones le permiten avanzar un cruce a la vez de acuerdo a su orientación y lo facultan para cambiar su dirección. Si el cruce destino está ocupado por un muro u otro robot, el avance es rechazado.

Un robot puede recoger cualquier trompo que se encuentre en el cruce frente a su posición. También puede dejar los que vaya cargando en el cruce de enfrente. Las siguientes funciones proveen estas operaciones:

---

[5] Los operadores lógicos de conjunción, disyunción y negación se denotan con los símbolos "&", "ó" y "¬" respectivamente.

R.16 Recoge: $\mathbf{DECMUN} \times ROBOT \models\Longrightarrow \mathbf{DECMUN} \times ROBOT$

$(DeclM, r) \dashrightarrow (DeclM', r')$.

Siendo *r.IdM* el identificador del mundo en el que habita el robot *r*, *r.x* su posición, *r.d* su dirección, *r.t* el número de trompos que carga y *m* el mundo asociado a *r.IdM* en la presente declaración de mundos *DeclM*, la declaración *DeclM'* y el robot *r'* resultantes de aplicar la función Recoge quedan definidos como sigue:

si $HayTrompo(m, r.d(r.x))$

$DeclM' = DeclM \cup \{ r.IdM \dashrightarrow QuitaTrompo(m, r.d(r.x)) \}$ y $r' = (r.IdM, r.x, r.d, r.t+1)$

de otra manera: $DeclM' = DeclM$ y $r' = r$.

R.17 Deja: $\mathbf{DECMUN} \times ROBOT \models\Longrightarrow \mathbf{DECMUN} \times ROBOT$

$(DeclM, r) \dashrightarrow (DeclM', r')$

si $r.t \geq 1$

$DeclM' = DeclM \cup \{ IdM \dashrightarrow PonTrompo(m, r.d(r.x)) \}$ y $r' = (r.IdM, r.x, r.d, r.t-1)$

de otra manera: $DeclM' = DeclM$ y $r' = r$.

Los predicados con los que cuentan los robots para indagar las condiciones prevalecientes en su hábitat son los siguientes:

$\mathbf{DECMUN} \times ROBOT \times DIRREL \models\Longrightarrow BOLEANO$

R.18 Norte?: $(DeclM, r, dr) \dashrightarrow (dr(r.d) = Norte)$

R.19 Muro?: $(DeclM, r, dr) \dashrightarrow HayMuro(DeclM(r.IdM), dr(r.d)(r.x))$

R.20 Robot?: $(DeclM, r, dr) \dashrightarrow HayRobot(DeclM(r.IdM), dr(r.d)(r.x))$

R.21 Trompo?: $(DeclM, r, dr) \dashrightarrow HayTrompo(DeclM(r.IdM), dr(r.d)(r.x))$.

R.22 Cargado?: $ROBOT \models\Longrightarrow BOLEANO$

$r \dashrightarrow (r.t \geq 1)$.

Con estas definiciones, los robots cuentan con todo lo necesario para realizar tareas interesantes. Sin embargo, aún no se han suministrado las herramientas necesarias para proveer a *LPC* de procesos concurrentes.

3.2.4 Varios robots

Al hablar de varios robots, lo primero que se necesita es poder distinguirlos; se requiere identificarlos. Se denota al conjunto de los identificadores de robots con **ID***ROB*. Se le llama **DEC***ROB* al conjunto de las funciones que nombran biunívocamente a los robots; formalmente:

R.23 **DEC***ROB* ≡ **ID***ROB* <==> *ROBOT*

Se necesita una función que permita la creación de un nuevo robot. Esta genitiva, llamada CreaRobot, queda definida de la siguiente manera:

R.24 CreaRobot:

$$\mathbf{DEC}ROB \times \mathbf{ID}ROB \times \mathbf{DEC}MUN \times \mathbf{ID}MUN \times CRUCE \times DIRECCION \times TROMPO \downarrow \mathbf{DEC}ROB \times \mathbf{DEC}MUN$$

$$(DeclR, IdR, DeclM, IdM, x, d, t) \quad \text{-►} \quad (DeclR', DeclM')$$

donde,

si $\neg$ HayRobot($DeclM(IdM)$, x) & $\neg$ HayMuro($DeclM(IdM)$, x)

$DeclM' = DeclM \cup \{ IdM \text{-►} PonRobot(DeclM(IdM) , x) \}$ y

$DeclR' = DeclR \cup \{ IdR \text{-►} (IdM, x, d, t) \}$

de otra manera: $DeclM' = DeclM$ y $DeclR' = DeclR$.

Asimismo, se puede destruir a cualquier robot nombrado, con la siguiente función:

R.25 DestruyeRobot: **DEC***MUN* × **DEC***ROB* × **ID***ROB* |==> **DEC***MUN* × **DEC***ROB*

$(DeclM, DeclR, IdR) \quad \text{-►} \quad (DeclM', DeclR')$

donde, siendo $r = DeclR(IdR)$ y $m = DeclM(r.IdM)$,

$DeclM' = DeclM \cup \{ r.IdM \text{ -►} QuitaRobot(m, r.x) \}$ y $DeclR' = DeclR - \{ IdR \text{ -►} r \}$

Ha llegado el momento de introducir las tareas. Una tarea es una secuencia de instrucciones. Por lo pronto, no se hacen explícitas las definiciones de las instrucciones; se denotan con el conjunto INST y son el conjunto de las acciones que pueden realizar los robots. Las funciones que se han expuesto serán utilizadas para darles significado a los elementos de INST en la sección 6.1.

El conjunto de las tareas queda definido por:

T.1 *TAREA* ≡ seq INST

Una tarea es pues un patrón de conducta para los robots. Para distinguir a cada una de las tareas, se les asocia un identificador único y se obtiene un conjunto de declaraciones de tareas que se denota de la siguiente manera:

T.2 $\mathbf{DEC}TAR \equiv \mathbf{ID}TAR \Longleftrightarrow TAREA$

Se han reunido los elementos suficientes para expresar lo que son los procesos de *LPC*. Al hablar de procesos concurrentes es indispensable definir su creación, su activación, su destrucción, su comunicación y su sincronización en términos del universo en el que se les esté ubicando.

La creación y la activación de un proceso en el presente universo se realizan como una sóla acción: crear un robot y asignarle una tarea. Un proceso es pues una pareja robot-tarea[6]. El conjunto de los procesos se puede caracterizar con:

P.1 $PROCESO \equiv \mathbf{ID}ROB \times \mathbf{ID}TAR$

El conjunto de los procesos activos queda definido de la siguiente manera:

P.2 $ACTIV \equiv \mathbf{seq}\ PROCESO$

La función que se provee para crear y activar procesos es la siguiente:

P.3 CreaProceso:

$$\mathbf{DEC}ROB \times \mathbf{ID}ROB \times \mathbf{DEC}MUN \times \mathbf{ID}MUN \times CRUCE \times DIRECCION \times TROMPO \times \mathbf{ID}TAR \times ACTIV$$
$$\downarrow$$
$$\mathbf{DEC}ROB \times \mathbf{DEC}MUN \times ACTIV$$

$$(DeclR, IdR, DeclM, IdM, x, d, t, IdT, a) \rightarrow (DeclR', DeclM', a')$$

donde, siendo $(DeclM', DeclR') \equiv \underline{\text{CreaRobot}}(DeclR, IdR, DeclM, IdM, x, d, t)$

si $DeclM' \neq DeclM \ \&\ DeclR' \neq DeclR$ (el robot fue creado exitosamente)

$a' \equiv \underline{\text{concat}}(a\ ,\ [\ (IdR\ ,\ IdT)\ ])$,

de otra manera, $a' \equiv a$[7].

---

[6] En la sección 6.1 se explica cómo un robot ejecuta la tarea que se le asigna.

[7] concat es el operador de concatenación de secuencias definido en [Sufrin86].

La destrucción de procesos se realiza con la función TerminaProceso:

P.4 TerminaProceso:

$$\mathbf{DECMUN} \times \mathbf{DECROB} \times PROCESO \times ACTIV \longmapsto \mathbf{DECMUN} \times \mathbf{DECROB} \times ACTIV$$

$$(DeclM, DeclR, (IdR, IdT), a) \rightarrow (DeclM', DeclR', a')$$

donde $(DeclM', DeclR') = \text{DestruyeRobot}(DeclR, DeclM, IdR)$.

Para poder expresar la remoción de un proceso, (IdR, IdT), de la secuencia actual de procesos, a, es necesario definir una función auxiliar ya que Z [Sufrin86] no la proporciona.

Considérese una secuencia $S=[a_1, a_2, .., a_p]$; S se puede reescribir en forma funcional de la siguiente manera: $S=\{1 \rightarrow a_1, 2 \rightarrow a_2, .., p \rightarrow a_p\}$. Así, una manera de expresar la remoción de un elemento de la secuencia, por ejemplo del k-ésimo, resulta de utilizar la composición funcional ($\bigcirc$); se compone a la derecha S con una función que es igual a la función identidad para los enteros menores que k, y es igual a la función sucesor para los enteros entre k y el número de elementos de la secuencia. A tal función se le llama Remueve.

Aux.1 $\text{Remueve}(i, j)$: $N \longmapsto N$

$$n \rightarrow \begin{cases} n & \text{si } n < i \\ n+1 & \text{si } i \leq n < j. \end{cases}$$

Si se denota al número de elementos de la secuencia S con $\#S$, se puede expresar la secuencia de procesos a' resultante de la operación TerminaProceso de la siguiente manera:

$$a' = a \bigcirc \text{Remueve}(k, \#a) \quad \text{donde } a(k) = (IdR, IdT).$$

Resta indicar cómo se comunican y cómo se sincronizan los robots de *LPC*. En la sección 3.1.2, se escogió un mecanismo asíncrono en base a envío de mensajes. Se consideraron deseables canales de nombramiento directo y buzones. Los canales de nombramiento directo soportan comunicaciones a nivel individual, mientras que los buzones permiten comunicaciones a nivel grupal. La principal diferencia entre los dos medios reside en los distintos niveles de comunicación que proporcionan.

La comunicación por nombramiento directo se realiza a nivel de los robots. Así, se le pone énfasis a las características de los individuos que se comunican.

El segundo mecanismo proporciona comunicaciones a nivel de las tareas. El énfasis se le pone a lo que "ejecutan" los comunicantes, sin importar "quiénes" sean. Este mecanismo permite que los robots se comuniquen por grupos, de acuerdo a la actividad que desempeñan.

Sean *CANAL* y *BUZON* el conjunto de las comunicaciones por nombramiento directo y el de las comunicaciones por buzón respectivamente. Se pueden denotar de la siguiente forma:

P.5 $CANAL = \mathbf{seq}\ (\ \mathbf{ID}ROB \times \mathbf{ID}ROB\ )$

P.6 $BUZON = \mathbf{seq}\ (\ \mathbf{ID}TAR \times \mathbf{ID}TAR\ ),$

donde la primera componente de las parejas es el nombre del emisor y la segunda es el del receptor. Cabe notar que los mensajes en *LPC* son vacíos. Los intercambios se reducen a simples señales. Se puede decir que los canales y los buzones son los alambres a través de los cuales viajan estas señales; excepto que estos alambres tienen la particularidad de conservar las señales hasta que sean removidas explícitamente.

Sean $\mathbf{ID}CAN$ e $\mathbf{ID}BUZ$ los conjuntos de identificadores de canales y de buzones respectivamente. Asimismo, sean $\mathbf{DEC}CAN$ y $\mathbf{DEC}BUZ$ los conjuntos de funciones identificadoras de canales y de buzones respectivamente.

P.7 $\mathbf{DEC}CAN = \mathbf{ID}CAN \langle\!\Longrightarrow CANAL$

P.8 $\mathbf{DEC}BUZ = \mathbf{ID}BUZ \langle\!\Longrightarrow BUZON$

Se pueden crear y destruir canales y buzones con las funciones que a continuación se definen:

$$\mathbf{DEC}CAN \times \mathbf{ID}CAN \ \longmapsto\ \mathbf{DEC}CAN$$

P.9 CreaCanal: $(DeclC, IdC) \rightarrow DeclC \cup \{\ IdC \rightarrow [\,]\ \}$

P.10 DestruyeCanal: $(DeclC, IdC) \rightarrow (\mathrm{dom}\ DeclC - \{\ IdC\ \}) \nabla DeclC$

donde $A \nabla f$ denota la restricción de f a A. Por lo tanto, DestruyeCanal restringe las declaraciones de canal, *DeclC*, al conjunto de los creados, que resulta de excluir el identificador del canal que se está destruyendo, $\mathrm{dom}\ DeclC - \{\ IdC\ \}$. En forma similar:

$$\mathbf{DEC}BUZ \times \mathbf{ID}BUZ \ \longmapsto\ \mathbf{DEC}BUZ$$

P.11 CreaBuzon: $(DeclB, IdB) \rightarrow DeclB \cup \{\ IdB \rightarrow [\,]\ \}$

P.12 DestruyeBuzon: $(DeclB, IdB) \rightarrow (\mathrm{dom}\ DeclB - \{\ IdB\ \}) \nabla DeclB$

Los procesos tienen ahora dos sistemas de comunicación a su alcance. Falta proveerlos con funciones para que puedan efectivamente mandar y recibir señales. Para enviar señales por un canal, se suministran las funciones Envia y Recibe.

P.13 Envia:

$$\mathbf{DEC}CAN \times \mathbf{DEC}ROB \times \mathbf{ID}CAN \times \mathbf{ID}ROB \times \mathbf{ID}ROB \longmapsto \mathbf{DEC}CAN$$

$$(DeclC,\ DeclR,\ IdC,\ IdRe, IdRr) \rightarrow DeclC', \quad \text{donde}$$

si $IdC \in \underline{\text{dom}}\ DeclC$ e $IdRe,\ IdRr \in \underline{\text{dom}}\ DeclR$

$$DeclC' = DeclC \cup \{\ IdC \rightarrow \underline{\text{concat}}(DeclC(IdC)\ ,\ [\ (IdRe, IdRr)\ ])\ \}$$

de otra manera: $DeclC' = DeclC$.

El robot cuyo identificador es *IdRe* envía una señal al que se llama *IdRr* por el canal nombrado *IdC* colocando la pareja formada por los nombres del emisor y del receptor al final de la secuencia de señales existentes en el mencionado canal. La señal no será enviada de no existir el canal o de no estar declarado alguno o ambos de los comunicantes. Al recibir *IdRr* una señal de *IdRe* por *IdC*, la primera pareja *(IdRe, IdRr)* en el canal , si existe alguna, será removida. De no existir señal alguna, el canal queda inalterado y se dice que no hubo recepción.

P.14 Recibe:

$$\mathbf{DEC}CAN \times \mathbf{DEC}ROB \times \mathbf{ID}CAN \times \mathbf{ID}ROB \times \mathbf{ID}ROB \longmapsto \mathbf{DEC}CAN$$

$$(DeclC,\ DeclR,\ IdC,\ IdRe,\ IdRr) \rightarrow DeclC'$$

donde, siendo $c = DeclC(IdC)$,

si $IdC \in DeclC$, $IdRe,\ IdRr \in \underline{\text{dom}}\ DeclR$ y existe $k = \underline{\min}\ \{\ i \mid c(i) = (IdRe, IdRr)\ \}$

$$DeclC' = DeclC \cup \{\ IdC \rightarrow c \bigcirc \underline{\text{Remueve}}(k, \#c)\ \}$$

en caso contrario: $DeclC' = DeclC$.

Antes de definir las funciones para solicitar y atender peticiones recurriendo a los buzones, se modifican las de envío y de recepción de señalesrecién expuestas. En efecto, hace falta un mecanismo para sincronizar procesos. Para ello, se requiere modificar las definiciones de *CANAL* (P.5) y de *BUZON* (P.6). Se desea poder expresar la suspensión de un proceso. Esto se logra si el medio de comunicación cuenta con una segunda se-

cuencia en la que conserva las señales esperadas, realizando la suspensión durante la recepción[9].

P.15 *CANAL* ≡ seq (ID*ROB* × ID*ROB*) × seq (ID*ROB* × *PROCESO*)

P.16 *BUZON* ≡ seq (ID*TAR* × ID*TAR*) × seq (ID*TAR* × *PROCESO*),

donde las primeras secuencias corresponden a las definiciones P.5 y P.6, y se las llama componentes "arribo" del medio. En las segundas secuencias, llamadas componentes "espera", la primera componente de las parejas es el nombre del emisor y la segunda es el proceso que quedó suspendido esperando recibir una señal de la primera componente. La secuencia componente "espera" de un medio de comunicación *medio* se denota con *medio.e* y la componente "arribo" con *medio.a*.

A continuación, se exhiben las modificaciones a las reglas P.9 a P.14 inducidas por estas nuevas definiciones.

P.17 CreaCanal: DEC*CAN* × ID*CAN* ⟹ DEC*CAN*

(DeclC, IdC) → *DeclC* ∪ { *IdC* → ([ ] , [ ]) }

P.18 CreaBuzon: DEC*BUZ* × ID*BUZ* ⟹ DEC*BUZ*

(DeclB , IdB) → *DeclB* ∪ { *IdB* → ([ ] , [ ]) }

Si un proceso *(IdRr , IdT)* se encuentra bloqueado en el canal llamado *IdC* en espera de un mensaje de *IdRe*, al enviárselo, *IdRe* activa a este proceso. Si por el contrario, no existe ninguna espera por la señal enviada, ésta queda inscrita al final de la secuencia de las señales que han arribado.

P.19 Envía: DEC*CAN* × DEC*ROB* × *ACTIV* × ID*CAN* × ID*ROB* × ID*ROB*

↓

DEC*CAN* × *ACTIV*

(DeclC, DeclR, a, IdC, IdRe, IdRr) → *(DeclC', a')*

donde, siendo c = *DeclC(IdC)* siempre que ello tenga sentido,

si existe k | *c.e(k)* = *(IdRe, (IdRr, IdT))*:

a' = concat(*a* , [*(IdRr , IdT)*]),

DeclC' = *DeclC* ∪ { *IdC* → *(c.a , c.e* ○ Remueve(*k*, #*c.e*)) },

[9] Se recuerda que las comunicaciones se han escogido asíncronas.

de otro modo, si $IdC \in \underline{dom}\ DeclC$ e $IdRe,\ IdRr \in \underline{dom}\ DeclR$:

$a' = a$ y $DeclC' = DeclC \cup \{\ IdC \rightarrow (\underline{concat}(c.a\ ,\ [\ (IdRe, IdRr)\ ])\ ,\ c.e)\ \}$ y

en caso contrario: $a' = a$ y $DeclC' = DeclC$.

Si un proceso llamado *(IdRr , IdT)* reclama una señal de *IdRe* en el canal llamado *IdC* y al menos existe una, se le entrega la primera que case con el patrón solicitado en la secuencia de las arribadas. De no existir señal alguna, *(IdRr , IdT)* queda bloqueado en la secuencia de las señales esperadas.

P.20 Recibe: $\mathbf{DEC}CAN \times \mathbf{DEC}ROB \times ACTIV \times \mathbf{ID}CAN \times \mathbf{ID}ROB \times \mathbf{ID}ROB$

$$\downarrow$$

$$\mathbf{DEC}CAN \times ACTIV$$

$$(DeclC, DeclR, a, IdC, IdRe, IdRr) \quad \rightarrow \quad (DeclC', a')$$

donde, si $IdC \in \underline{dom}\ DeclC$ & $IdRe,\ IdRr \in \underline{dom}\ DeclR$ & $c = DeclC(IdC)$ &

existe $k = \underline{min}\ \{\ i \in N\ |\ c.a(i) = (IdRe, IdRr)\ \}$:

$a' = a$ y $DeclC' = DeclC \cup \{\ IdC \rightarrow (c.a \cap \underline{Remueve}(k, \#c.a)\ ,\ c.e)\ \}$,

de otro modo,

si $IdC \in \underline{dom}\ DeclC$ & $IdRe,\ IdRr \in \underline{dom}\ DeclR$ & $c = DeclC(IdC)$ &

$p\ |\ a(p) = (IdRr, IdT)$:

$a' = a \cap \underline{Remueve}(p, \#a)$ y

$DeclC' = DeclC \cup \{\ IdC \rightarrow (c.a\ ,\ \underline{concat}(c.e\ ,\ [\ (IdRe, a(p))\ ])\ \}$ y

en caso contrario: $a' = a$ y $DeclC' = DeclC$.

Se incluye un predicado para que los procesos puedan checar si les ha llegado correspondencia sin peligro de quedarse bloqueados al hacerlo.

P.21 Correo?: $\mathbf{DEC}CAN \times \mathbf{ID}CAN \times \mathbf{ID}ROB \times \mathbf{ID}ROB \quad \Longrightarrow \quad BOOLEANO$

$$(DeclC,\ IdC,\ IdRe,\ IdRr)$$

$$\downarrow$$

$$(\ \text{existe}\ k \in N\ |\ DeclC(IdC).a(k) = (IdRe, IdRr)\ )$$

En forma semejante, se definen las primitivas asociadas al uso de los buzones, concluyendo así la edificación de los objetos semánticos de *LPC*.

P.22 Solicita:

$$\mathbf{DEC}BUZ \times \mathbf{DEC}ROB \times \mathbf{DEC}TAR \times ACTIV \times \mathbf{ID}BUZ \times \mathbf{ID}ROB \times \mathbf{ID}TAR \downarrow \mathbf{DEC}BUZ \times ACTIV$$

$$(DeclB, DeclT, DeclR, a, IdB, IdRe, IdTr) \rightarrow (DeclB', a')$$

donde, siendo $b = DeclB(IdB)$ y $j \mid a(j) = (IdRe, IdTe)$ siempre que ello tenga sentido,

si existe $k = \min \{ i \mid b.e(i) = (IdTe, p) \ \& \ p.IdT = IdTr \}$:

$$a' = \text{concat}(a, [\ p\ ])\ \text{y}$$

$$DeclB' = DeclB \cup \{ IdB \rightarrow (b.a\ ,\ b.e \bigcirc \text{Remueve}(k, \#b.e)) \},$$

de otro modo,

si $IdB \in \text{dom}\ DeclB\ \&\ IdTr \in \text{dom}\ DeclT\ \&\ IdRe \in \text{dom}\ DeclR$:

$$a' = a\ \text{y}$$

$$DeclB' = DeclB \cup \{ IdB \rightarrow (\text{concat}(b.a\ ,\ [\ (IdTe, IdTr)\ ])\ ,\ b.e) \}\ \text{y}$$

en caso contrario: $a' = a$ y $DeclB' = DeclB$.

P.23 Atiende:

$$\mathbf{DEC}BUZ \times \mathbf{DEC}TAR \times \mathbf{DEC}ROB \times ACTIV \times \mathbf{ID}BUZ \times \mathbf{ID}TAR \times \mathbf{ID}ROB \downarrow \mathbf{DEC}BUZ \times ACTIV$$

$$(DeclB, DeclT, DeclR, a, IdB, IdTe, IdRr) \rightarrow (DeclB', a')$$

donde, siendo $b = DeclB(IdB)$ y $p \mid a(p) = (IdRr, IdTr)$ siempre que ello tenga sentido,

si $IdB \in \text{dom}\ DeclB\ \&\ IdTe \in \text{dom}\ DeclT\ \&\ IdRr \in \text{dom}\ DeclR\ \&$

existe $k = \min \{ i \mid b.a(i) = (IdTe, IdTr) \}$:

$$a' = a\ \text{y}\ DeclB' = DeclB \cup \{ IdB \rightarrow (b.a \bigcirc \text{Remueve}(k, \#b.a)\ ,\ b.e) \},$$

de otro modo,

si $IdB \in \text{dom}\ DeclB\ \&\ IdTe \in \text{dom}\ DeclT\ \&\ IdRr \in \text{dom}\ DeclR$:

$$a' = a \bigcirc \text{Remueve}(p, \#a)\ \text{y}$$

$$DeclB' = DeclB \cup \{ IdB \rightarrow (b.a\ ,\ \text{concat}(b.e\ ,\ [\ (IdTe, (IdRr, IdTr))\ ])\ \}\ \text{y}$$

en caso contrario: $a' = a$ y $DeclB' = DeclB$.

P.24 Peticion?: DEC*BUZ* × ID*BUZ* × ID*TAR* × ID*TAR* ⟹ *BOOLEANO*

(DeclB, IdB, IdTe, IdTr)

(existe $k \in N$ | *DeclB(IdB).a(k)* = *(IdTe, IdTr)*).

4. El lenguaje de LPC : sintaxis

En este capítulo, se presenta el lenguaje de programación que literalmente anima el universo de *LPC*. Se expone la sintaxis del lenguaje, explicando informalmente las restricciones que no se pueden expresar con una gramática libre de contexto. Se define el conjunto INST de las instrucciones que conforman a las tareas de *LPC*. INST es pues un conjunto de expresiones sintácticas.

Se emplea el Formalismo Normal Extendido de Backus (EBNF) para expresar la sintaxis del lenguaje de *LPC* . Para evitar cualquier confusión entre los paréntesis cuadrados usados en la denotación de las secuencias, y aquéllos usados en EBNF para indicar que la forma sintáctica que encierran es optativa, se les sustituye en EBNF por los símbolos "//" y "]//". Asimismo, los corchetes de EBNF, que caracterizan cero o más apariciones de la forma sintáctica encerrada podrían confundirse con los corchetes que se usan para denotar conjuntos, por lo que se sustituyen los de EBNF por los símbolos "/{" y "}/".

4.1 Un programa

Se especifica primero la sintaxis de los identificadores y de los enteros que se usan en el lenguaje.

Id ::= Alfa | Id Alfa | Id Digito | Id_
Alfa ::= a | .. | z | A | .. | Z
Digito ::= 0 | .. | 9
Entero ::= Digito | Entero Digito

Un programa en *LPC* es una lista de declaraciones, seguida de una lista de asignaciones. La primera lista declara tres tipos de objetos: las líneas de comunicación, los robots y las tareas. La segunda lista asigna tareas a robots.

Programa ::= **programa** Id ; ListaDeDeclaraciones **es** ListaDeAsignaciones **fin** Id .
ListaDeDeclaraciones ::= /{ Comunicaciones ; }/ ListaDeRobots ListaDeTareas
ListaDeAsignaciones ::= Asignacion /{ ; ListaDeAsignaciones }/
Asignacion ::= Id : = Id // (ListaDeParametrosActuales) //
ListaDeParametrosActuales ::= ParametroActual // , ListaDeParametrosActuales //
ParametroActual ::= Id | **yo** | DireccionRelativa
DireccionRelativa ::= **prox** | **estribor** | **popa** | **babor**

A continuación, se exponen las distintas restricciones de contexto que debe checar un compilador para este lenguaje. El identificador que inicia el bloque **programa...fin** debe emplearse al final del mismo. En una asignación, el primer identificador corresponde a un robot al que no se le haya asignado tarea alguna. El segundo denota a una tarea. Los parámetros actuales deben casar en número y tipo con los formales. El significado de esta asignación es crear y activar un proceso cuyo agente es el robot referido y cuya tarea es la indicada. La palabra reservada **yo** sustituye al nombre del robot que esté ejecutando la tarea en la cual aparezca. Así, su tipo es **robot**.

prox, **estribor**, **popa** y **babor** denotan a las direcciones relativas Prox, Estribor, Popa y Babor correspondientes a las definiciones R.10 a R.13 de la sección 3.2.3.

4.2 Las declaraciones

El lenguaje de *LPC* exige que la declaración de un identificador siempre preceda su uso. Se comienza con la declaración de líneas de comunicación. Ya que los medios de comunicación son de dos tipos, se pueden declarar canales y/o buzones. Los nombres de éstos deben ser únicos.

Comunicaciones ::= DeclaracionCanal | DeclaracionBuzon
DeclaracionCanal ::= **canal** : Id /{ , Id }/
DeclaracionBuzon ::= **buzon** : Id /{ , Id }/

Las declaraciones de robots indican sus características iniciales, el nombre del mundo que habitan y si comparten su hábitat.

ListaDeRobots ::= Robots ; // ListaDeRobots //
Robots ::= **robot** : Robot /{ , Robot }/
Robot ::= Id **comparte** Id CondicionesIniciales | Id **habita** Id CondicionesIniciales
CondicionesIniciales ::= **en** Entero , Entero **norte** DireccionRelativa **con** Entero

El nombre de los robots debe ser único y será conocido sólo en el bloque (programa o tarea) en el que se haya declarado. Si un robot comparte su mundo con otros robots, lo cual queda indicado con la palabra **comparte**, una sola copia del mundo será desplegada. Si, por el contrario, se emplea la palabra **habita**, una copia del mundo será desplegada para cada robot que lo habite. Los mundos son creados con el editor de mundos de *LPC*. Este editor guarda los mundos en archivos cuyos nombres, sin extensión, son los que deben aparecer en la declaración de los robots.

En las condiciones iniciales, primero aparecen el número de avenida y el número de calle que conforman el cruce ocupado por el robot; les siguen su dirección y el número de trompos que habrá de tener en su bolsa al "nacer". El número de trompos no puede pasar de 99. El compilador debe checar si el mundo está disponible y, en caso de estarlo, debe verificar que la posición inicial declarada existe y que se encuentra libre. Si además el mundo es compartido, debe marcar la posición como ocupada para detectar "nacimientos" múltiples en la misma posición.

Finalmente, las declaraciones de tareas especifican las instrucciones que tendrán que ejecutar los robots cuando se les asignen dichas tareas.

ListaDeTareas ::= Tarea ; // ListaDeTareas //
Tarea ::= **tarea** Id // (ParametrosFormales) /// ListaDeRobots // **es** INST/{ ; INST }/ **fin** Id
ParametrosFormales ::= ParametroFormal // ; ParametrosFormales //
ParametroFormal ::= PasaDerechoDeUso | PasaNombre
PasaDerechoDeUso ::= **var** Id/{ , Id }/ : **robot** | **var** Id/{ , Id }/ : **tarea**
PasaNombre ::= Id/{ , Id }/ : **robot** | Id/{ , Id }/ : **tarea** | Id/{ , Id }/ : **dirrel**

El identificador que inicia el bloque **tarea**...**fin** debe emplearse al final del mismo y debe ser único. Este identificador será conocido a partir de su declaración hasta el final del bloque de programa. La declaración de robots locales y el paso var de parámetros de tipo **robot** permiten la creación de procesos a tiempo de ejecución. Un robot local puede ser usado, o "nacer", varias veces dentro del cuerpo de la tarea que lo declara; sin embargo, en todas sus "vidas" conservará el mismo nombre. El nombre que los robots locales adquieren al "nacer" es único en cada una de las distintas activaciones de la tarea. Los robots globales pueden ser pasados como parámetros **var** a alguna tarea. El hacerlo equivale a ceder el derecho de uso a la tarea de ese robot que, por lo tanto, ya no podrá ser objeto de una asignación de tarea en el bloque actual; solamente la tarea, a la cual se le ha pasado, podrá asignarle una tarea o volverlo a pasar como parámetro var. Los parámetros var de tipo tarea pueden ser asignados a ro-

bots. Se proporcionan para poder programar tareas mutuamente recursivas y tareas despachadoras. Los parámetros que no van precedidos por la palabra **var** denotan nombres de robots y de tareas que se emplean en las instrucciones de envío de señales, o bien, identificadores de direcciones relativas que se utilizan en la instrucción **gira** y en la mayoría de los predicados.

4.3 Las instrucciones

Aunque una definición formal de la semántica de estas instrucciones se presenta en el capítulo sexto, a continuación se exponen, junto con la información sintáctica, sus significados en forma aproximada.

INST ::= **nada** | **avanza** | **gira** Id | **gira** DireccionRelativa | **recoge** | **deja** |
EnvioDeMensajes | Alternativa | Iteracion | **rompe** |
LlamadaDeTarea | CreacionDeProceso

nada denota a una función que hace nada; **avanza**, **gira**, **recoge** y **deja** significan aproximadamente lo mismo que las funciones Avanza, Gira, Recoge y Deja correspondientes a las definiciones R.14 a R.17 de la sección 3.2.3. Gira toma su argumento directamente del valor denotado por el símbolo de DireccionRelativa, o bien lo hace del valor asociado a Id, de acuerdo a la forma sintáctica usada. La instrucción **rompe** solamente puede aparecer dentro de una Iteracion, o dentro de una Alternativa incluso en una Iteracion. Ejecutar un **rompe** equivale a brincar a la instrucción que siga el **fin** de la Iteracion más próxima.

EnvioDeMensajes ::= Id^1 ! Id^2 | Id^1 ? Id^2

Si Id^1 corresponde a un robot, Id^2 debe corresponder a un canal. Si el símbolo que los separa es !, entonces significa enviar una señal al robot asociado a Id^1 por el canal asociado a Id^2. Si el símbolo es ?, entonces significa suspenderse hasta poder recibir una señal del robot asociado a Id^1 por el canal asociado a Id^2. Si en cambio, Id^1 corresponde a una tarea, Id^2 debe corresponder a un buzón. La primera expresión sintáctica significa entonces enviar una señal a cualquier robot que se encuentre ejecutando la tarea asociada a Id^1 por el buzón asociado a Id^2. La segunda expresión significa suspenderse hasta poder recibir una señal de algún robot que esté ejecutando la tarea asociada a Id^1 por el buzón asociado a Id^2.

Alternativa ::= **escoge** ComandoCustodiado /{ | ComandoCustodiado }/// Salida // **fin**
ComandoCustodiado ::= Guardia -> INST > /{ ; INST }/
Salida ::= **ninguna** -> INST > /{ ; INST }/
Guardia ::= TerminoBooleano /{ **o** TerminoBooleano }/
TerminoBooleano ::= FactorBooleano /{ **y** FactorBooleano }/
FactorBooleano ::= **no** FactorBooleano | (Guardia) | **remite** Id **en** Id |
cargado | **cierto** | Predicado Id | Predicado DireccionRelativa
Predicado ::= **norte** | **muro** | **robot** | **trompo**

Esta instrucción permite especificar el no determinismo explícitamente en *LPC*. Ejecutar una instrucción **escoge...fin** significa escoger al azar una Guardia de entre las que se evaluen verdaderas, y ejecutar las instrucciones asociadas a la Guardia electa. Si ninguna resulta verdadera y si existe Salida, entonces significa ejecutar las instrucciones asociadas a la Salida. Si ninguna Guardia se satisface y no existe Salida, entonces el programa será abortado con el mensaje de error correspondiente. Unicamente la instrucción enmarcada por los símbolos -> y > goza del privilegio de tener garantizadas sus condiciones iniciales de ejecución. En efecto, si es la primera de ComandoCustodiado, se le garantiza que al iniciar su ejecución la Guardia se satisface. Si por el contrario, es la primera de Salida, se le garantiza que todas las Guardias resultan falsas. Una Guardia se evalúa como una expresión booleana. El predicado denotado por **remite** Id^1 **en** Id^2 significa aproximadamente lo mismo que Correo? (P.21) si Id^1 es el identificador de un robot e Id^2 lo es de un canal; si por el contrario, Id^1 denota una tarea e Id^2 un buzón, entonces significa algo parecido a Peticion? (P.24). Los predicados denotados por **norte**, **muro**, **robot**, **trompo** y **cargado** son Norte?, Muro?, Robot?, Trompo? y Cargado? correspondientes a las definiciones R.18 a R.22 de la sección 3.2.3. Los predicados Norte?, Muro?, Robot? y Trompo? toman su argumento directamente del valor denotado por el símbolo de DireccionRelativa, o bien, lo hacen del valor asociado a Id, de acuerdo a la forma sintáctica usada. El predicado denotado por **cierto** es aquél que siempre se evalúa verdadero.

Iteracion ::= **ciclo** INST /{ ; INST }/ **fin**

La lista de instrucciones encerrada por **ciclo** y **fin** se ejecutará una y otra vez por siempre, a menos que la lista incluya una instrucción **rompe** o una instrucción que la lleve inclusa. Combinando instrucciones de forma Iteracion con las de forma Alternativa se obtienen las instrucciones presentes en la mayoría de los lenguajes de programación actuales:

if...then /{ elsif }/// else //...end, while...end, repeat...until y loop.../// break //...end.

LlamadaDeTarea ::= yo := Id // (ListaDeParametrosActuales) //

Esta instrucción permite un cambio temporal de tarea (no de proceso). El robot denotado por yo ejecutará la tarea llamada Id. Al terminar su nueva tarea, si terminara, continuará con la actual, en la instrucción que sigue a LlamadaDeTarea. ListaDeParametrosActuales fue explicada al presentar a Asignacion.

CreacionDeProceso ::= Id := Id // (ListaDeParametrosActuales) //

El primer identificador debe ser el de un robot declarado localmente o el de un robot global pasado como parámetro var. El segundo debe ser el de una tarea y los parámetros actuales deben casar con los formales. El significado de esta instrucción es crear y activar un proceso; permite pues especificar la creación dinámica de procesos de *LPC*.

5. El lenguaje de LPC : contexto

En este capítulo, se llevan a cabo modificaciones a las definiciones presentadas en el capítulo tres para descender al nivel de abstracción requerido por la realización de *LPC* en una computadora. Se especifican formalmente los aspectos del lenguaje sensibles al contexto que se expresaron informalmente en el capítulo anterior; con este fin, se emplea una gramática de atributos [Knuth68]. Antes de formalizar lo que se ha narrado en las secciones del capítulo anterior, son necesarias ciertas modificaciones en algunas de las definiciones del capítulo tres. En efecto, al construir el universo semántico no se consideraron las limitaciones impuestas por su realización en una computadora. Tan sólo se expresaron conceptos. Estos no requieren de modificación alguna. Sin embargo, resulta necesario precisar la forma de ciertos objetos para poder implementarlos. Por otro lado, la introducción de un lenguaje de programación conlleva consideraciones pragmáticas que no figuraron en el tercer capítulo por no tener relevancia a nivel conceptual. La existencia de parámetros, de tipos, de llamadas de tareas etc. induce un mayor grado de precisión en las definiciones del universo. Han de modificarse las definiciones de los mundos, de los trompos, de **DEC***MUN*, de **DEC***ROB*, de **DEC***TAR* y de los procesos.

5.1 Hacia una implementación

Los mundos de *LPC* tienen un largo y un ancho finitos.

M.28 *CALLE* ≡ [1 .. MáximoLargo (= 11)]

M.29 *AVENIDA* ≡ [1 .. MáximoAncho (= 15)]

Aunque en la sección 7.5.1 se describen las representaciones gráficas de los objetos de *LPC*, conviene aclarar que el origen de un mundo, el cruce (1,1), queda representado en la esquina superior izquierda de la pantalla y que las calles se representan horizontales y las avenidas verticalmente. Ello justifica las definiciones que a continuación se exponen.

M.30 **DECMUN** ≡ IDMUN ⟸⟹ *MUNDO* × *AVENIDA* × *CALLE*

donde se denotan con *DeclM(IdM).A* y con *DeclM(IdM).L* al ancho y largo del mundo denotado con *DeclM(IdM).m*.

La creación de un mundo ahora requiere la determinación de su tamaño. Ello se realiza interactivamente con el editor de mundos de *LPC*.

M.31 CreaMundo :

DECMUN × IDMUN × *AVENIDA* × *CALLE* ⟾ **DECMUN**

(DeclM, IdM, A, L) ↦ *DeclM* ⩌ {*IdM* ↦ *(MundoVacio, A, L)*}

donde *MundoVacio* es aquel mundo en el que se cumple para todo cruce *x*, *MundoVacio(x)* = (LIBRE, 0).

Los trompos también quedan restringidos en la implementación. Ya que su repretación se limita a dos caracteres que indican su cantidad en cada cruce, el mayor número de trompos permitido es de 99.

M.32 *TROMPO* ≡ [0 .. MáximoTrompos (= 99)]

Las funciones dirección tienen que tomar en cuenta la finitud de los mundos. Los mundos no tenían fronteras al Norte ni al Este. Las nuevas condiciones se definen en forma homogénea a las anteriores: los cruces de cada una de las fronteras son puntos fijos de una de las funciones dirección; la frontera Este queda conformada por los cruces *x* tales que Este(*x*)=*x*, los de la Sur por aquéllos tales que Sur(*x*)=*x* etc. Así, las direcciones quedan especificadas de la siguiente manera:

		DECMUN × IDMUN ⟾	[*CRUCE* ⟾ *CRUCE*]
R.26	Norte:	*(DeclM, IdM)* ↦	[*(a, c)* ↦ *(a, c⊥1)*]
R.27	Este:	*(DeclM, IdM)* ↦	[*(a, c)* ↦ *(a⊦1, c)*]
R.28	Sur:	*(DeclM, IdM)* ↦	[*(a, c)* ↦ *(a, c⊤1)*]
R.29	Oeste:	*(DeclM, IdM)* ↦	[*(a, c)* ↦ *(a⊥1, c)*]

donde $n \perp 1 \equiv \begin{cases} n - 1 & \text{si } n > 1 \\ n & \text{de otra manera,} \end{cases}$

$n \vdash 1 \equiv \begin{cases} n + 1 & \text{si } n < DeclM(IdM).A \\ n & \text{de otra manera} \quad \text{y} \end{cases}$

$$n \top l \equiv \begin{cases} n+1 & \text{si } n < DeclM(IdM).L \\ n & \text{de otra manera.} \end{cases}$$

Al haber alterado los elementos del conjunto *DIRECCION*, deben ser modificadas todas las definiciones que los usan. Las modificaciones son menores: cada vez que se usa una dirección, se tiene que indicar en qué mundo está el usuario. Esto se logra sustituyendo *r.d* por *r.d(DeclM(r.IdM))* en cada una de las definiciones.

R.30 Avanza:

$$\mathbf{DEC}MUN \times ROBOT \longmapsto \mathbf{DEC}MUN \times ROBOT$$
$$(DeclM, r) \quad \rightarrow \quad (DeclM', r')$$

donde, siendo $m \equiv DeclM(r.IdM).m$, $A \equiv DeclM(r.IdM).A$ y $L \equiv DeclM(r.IdM).L$,

si $\neg$ HayRobot$(m, r.d(DeclM(r.IdM))(r.x))$ & $\neg$ HayMuro$(m, r.d(DeclM(r.IdM))(r.x))$

$DeclM' \equiv DeclM \cup \{ r.IdM \rightarrow (\text{PonRobot}(\text{QuitaRobot}(m, r.x), r.d(DeclM(r.IdM))(r.x)), A, L) \}$

y $r' \equiv (r.IdM, r.d(DeclM(r.IdM))(r.x), r.d, r.t)$

de otra manera: $DeclM' \equiv DeclM$ y $r' \equiv r$.

R.31 Gira:

$$\mathbf{DEC}MUN \times ROBOT \times DIRREL \longmapsto ROBOT$$
$$(DeclM, r, dr) \quad \rightarrow \quad (r.IdM, r.x, rd(DeclM(r.IdM))(r.d), r.t).$$

R.32 Recoge:

$$\mathbf{DEC}MUN \times ROBOT \longmapsto \mathbf{DEC}MUN \times ROBOT$$
$$(DeclM, r) \quad \rightarrow \quad (DeclM', r')$$

donde, siendo $A \equiv DeclM(r.IdM).A$ y $L \equiv DeclM(r.IdM).L$,

si HayTrompo$(DeclM(r.IdM).m, r.d(DeclM(r.IdM))(r.x))$

$DeclM' \equiv DeclM \cup \{ r.IdM \rightarrow (\text{QuitaTrompo}(DeclM(r.IdM).m, r.d(DeclM(r.IdM))(r.x)), A, L) \}$

y $r' \equiv (r.IdM, r.x, r.d, r.t+1)$

de otra manera: $DeclM' \equiv DeclM$ y $r' \equiv r$.

R.33 Deja: $\mathbf{DEC}MUN \times ROBOT \quad |\!\!\Longrightarrow \quad \mathbf{DEC}MUN \times ROBOT$

$(DeclM, r) \quad \rightarrow \quad (DeclM', r')$

donde, siendo $A = DeclM(r.IdM).A$ y $L = DeclM(r.IdM).L$,

si $r.t > 0$

$DeclM' = DeclM \cup \{\ r.IdM \rightarrow (\underline{PonTrompo}(DeclM(r.IdM).m, r.d(DeclM(r.IdM))(r.x)), A, L)\ \}$

y $r' = (r.IdM, r.x, r.d, r.t-1)$

de otra manera: $DeclM' = DeclM$ y $r' = r$.

$\mathbf{DEC}MUN \times ROBOT \times DIRREL \quad |\!\!\Longrightarrow \quad BOOLEANO$

R.34 $\underline{\text{Norte?}}$: $(DeclM, r, dr) \rightarrow (dr(DeclM(r.IdM))(r.d) = \underline{\text{Norte}})$

R.35 $\underline{\text{Muro?}}$: $(DeclM, r, dr) \rightarrow \underline{HayMuro}(DeclM(r.IdM), dr(r.d)(DeclM(r.IdM))(r.x))$

R.36 $\underline{\text{Robot?}}$: $(DeclM, r, dr) \rightarrow \underline{HayRobot}(DeclM(r.IdM), dr(r.d)(DeclM(r.IdM))(r.x))$

R.37 $\underline{\text{Trompo?}}$: $(DeclM, r, dr) \rightarrow \underline{HayTrompo}(DeclM(r.IdM), dr(r.d)(DeclM(r.IdM))(r.x))$.

Las declaraciones de robots tienen que indicar si tienen o no asignada una tarea. A cada identificador de robot se le asocia el robot correspondiente y un valor verdadero (V) si tiene tarea asignada, falso (F) en caso contrario. Se hace lo mismo para indicar si el mundo que habitan es compartido o no.

R.38 $ASIGNADO = BOOLEANO$

R.39 $COMPARTIDO = BOOLEANO$

R.40 $\mathbf{DEC}ROB = \mathbf{ID}ROB \iff ROBOT \times ASIGNADO \times COMPARTIDO$

Sean $DeclR \in \mathbf{DEC}ROB$ e $IdR \in \mathbf{ID}ROB$, entonces:

$DeclR(IdR).r$	denota al robot asociado a IdR,
$DeclR(IdR).r.IdM$	denota al identificador del mundo que habita r,
$DeclR(IdR).r.x$	denota la posición del robot,
$DeclR(IdR).r.d$	denota su dirección,
$DeclR(IdR).r.t$	denota la cantidad de trompos que lleva,
$DeclR(IdR).Asignado$	indica si el robot ha sido asignado o no y
$DeclR(IdR).Compartido$	indica si el robot comparte su mundo o no.

La creación de un robot conlleva ahora la inicialización del mundo que habitará. Los mundos, creados y diseñados con el editor de *LPC*, pueden ser compartidos por varios robots o pertenecer en exclusiva a alguno de ellos como se indicó al exponer las declaraciones de robots en la sección

4.2. En caso de ser compartido y al crear al primero de los robots que lo comparten, se construye una copia del mundo archivado por el editor. En caso contrario, una copia se constituye cada vez que un morador de dicho mundo es creado.

R.41 CreaRobot :

$$\begin{array}{lll} \mathbf{DEC}ROB \times \mathbf{ID}ROB \times \mathbf{DEC}MUN & \longmapsto & \mathbf{DEC}ROB \times \mathbf{DEC}MUN \times ROBOT \\ (DeclR, IdR, DeclM) & \rightarrow & (DeclR', DeclM', r') \end{array}$$

donde, siendo $r = DeclR(IdR).r$, $m = DeclM(r.IdM).m$, $DeclM'' = DeclM$, y

si $r.IdM \in DeclM$,

$$DeclM'' = DeclM \cup \{\, r.IdM \rightarrow \underline{\text{CargaMundo}}(r.IdM) \,\} \text{ y } m = DeclM''(r.IdM).m$$

en caso contrario,

$$A = DeclM''(r.IdM).A \text{ y } L = DeclM''(r.IdM).L,$$

☞ si $r.Compartido \;\&\; \neg\, \underline{\text{HayRobot}}(m\ ,\ r.x) \;\&\; \neg\, \underline{\text{HayMuro}}(m\ ,\ r.x)$

$$DeclM' = DeclM'' \cup \{\, r.IdM \rightarrow (\underline{\text{PonRobot}}(m\ ,\ r.x), A, L) \,\},$$

$$r' = r \text{ y siendo } IdR' \mid IdR' \notin \underline{\text{dom}}\ DeclR,$$

$$DeclR' = DeclR \cup \{\, IdR' \rightarrow (r',\ \underline{x}\ ,\ \underline{y}) \,\}$$

☞ si $r.Compartido \;\&\; m' = \underline{\text{CargaMundo}}(r.IdM).m \;\&\; \neg\, \underline{\text{HayMuro}}(m'\ ,\ r.x)$

siendo $IdR' \mid IdR' \notin \underline{\text{dom}}\ DeclR$ e $IdM' \mid IdM' \notin \underline{\text{dom}}\ DeclM''$,

$$DeclM' = DeclM'' \cup \{\, IdM' \rightarrow (\underline{\text{PonRobot}}(m'\ ,\ r.x)\ , A\ , L) \,\},$$

$$r' = (IdM', r.x, r.d, r.t) \text{ y}$$

$$DeclR' = DeclR \cup \{\, IdR' \rightarrow (r',\ \underline{x}\ ,\ \underline{y}) \,\}$$

☞ de otra manera: $DeclM' = DeclM''$ y $DeclR' = DeclR$.

CargaMundo es una función auxiliar que proporciona una tripleta. La primera componente de la tripleta es un mundo de características idénticas al creado con el editor de *LPC* y archivado bajo el nombre que toma por argumento. La segunda y la tercera son el ancho y el largo del mundo respectivamente. En caso de no existir tal archivo, CargaMundo creará la tripleta *(MundoVacio*, MáximoAncho, MáximoLargo*)*.

R.42 DestruyeRobot:

$$\mathbf{DEC}MUN \times \mathbf{DEC}ROB \times \mathbf{ID}ROB \;\longmapsto\; \mathbf{DEC}MUN \times \mathbf{DEC}ROB$$
$$(DeclM, DeclR, IdR) \;\dashrightarrow\; (DeclM', DeclR')$$

donde, siendo $r = DeclR(IdR).r$, $m = DeclM(r.IdM).m$, $A = DeclM(r.IdM).A$ y $L = DeclM(r.IdM).L$,

☞ si $DeclR(IdR).Compartido$

$DeclM' = DeclM \cup \{\ r.IdM \dashrightarrow (\mathrm{QuitaRobot}(m, r.x), A, L)\ \}$ y

$DeclR' = \mathrm{dom}\ DeclR - \{\ IdR\ \} \triangledown DeclR$

☞ si $\neg\ DeclR(IdR).Compartido$

$DeclM' = \mathrm{dom}\ DeclM - \{\ r.IdM\ \} \triangledown DeclM$ y

$DeclR' = \mathrm{dom}\ DeclR - \{\ IdR\ \} \triangledown DeclR$

donde $A \triangledown f$ denota la restricción de f a A.

Toca el turno de introducir los conjuntos necesarios para el manejo de parámetros. Se requiere un conjunto de identificadores para nombrarlos. Siendo T.3 *IDENTIFICADOR* el conjunto de todos los posibles identificadores, se define el conjunto de los nombres de parámetros como sigue:

T.4 $\mathbf{ID}PAR = IDENTIFICADOR - (\mathbf{ID}CAN \cup \mathbf{ID}BUZ \cup \mathbf{ID}ROB \cup \mathbf{ID}MUN \cup \mathbf{ID}TAR)$.

Los parámetros en el lenguaje de *LPC* pueden ser de cinco tipos:

T.5 $TIPO = \{$ ROBVAR, TARVAR, ROBPAR, TARPAR, DIRPAR $\}$.

Los parámetros formales de una tarea son una secuencia de parejas nombre-tipo:

T.6 $\mathbf{PAR}FORMAL = \mathbf{seq}\ (\ \mathbf{ID}PAR \times TIPO\ )$.

Así, si $Pf \in \mathbf{PAR}FORMAL$ y $k \in [1\ ..\ \#Pf]$, entonces $Pf(k).IdPf$ denota el nombre del k-ésimo parámetro y $Pf(k).Tipo$ indica su tipo. La declaración de una tarea incluye la declaración de sus parámetros formales y la de sus robots locales.

T.7 $\mathbf{DEC}TAR = \mathbf{ID}TAR \longmapsto (\ TAREA \times \mathbf{PAR}FORMAL \times \mathbf{DEC}ROB\ )$

Si $DeclT \in \mathbf{DEC}TAR$ e $IdT \in \mathbf{ID}TAR$, entonces $DeclT(IdT).t$ denota la tarea asociada al identificador IdT y $DeclT(IdT).Pf$ y $DeclT(IdT).RI$ denotan sus parámetros formales y sus robots locales respectivamente.

Los parámetros actuales son una secuencia de valores que pueden ser el nombre de un robot, el de una tarea, el de un parámetro o bien una dirección relativa.

T.8 **PARACTUAL** ≡ **seq** (**ID***ROB* ∪ **ID***TAR* ∪ **ID***PAR* ∪ *DIRREL*)

Al ejecutarse una tarea, es necesario mantener las identidades de los parámetros actuales de su asignación, así como las de los posibles robots locales y la del robot que la ejecuta. El conjunto *AMBIENTE* cumple con esa función.

T.9 *PARAM* ≡ **ID***PAR* <⟹> ((**ID***ROB* ∪ **ID***TAR* ∪ *DIRREL*) × *TIPO*)

T.10 *AMBIENTE* ≡ **ID***ROB* × **DEC***ROB* × *PARAM*

Dado un ambiente *A*, *A.Ejec* denota al identificador del robot que está ejecutando la tarea, *A.Rl* denota la declaración de robots locales, *A.Par* denota al conjunto que relaciona los parámetros formales a los actuales, *A.Par(Id).Val* denota al valor asociado a *Id* y *A.Par(Id).Tipo* a su tipo.

Para poder expresar la semántica del lenguaje de *LPC*, un proceso debería formarse con un robot, una tarea, el nombre de ésta y un ambiente. El robot es necesario ya que cada proceso posee un agente procesador. En la sección 6.1, se expresa la semántica de las instrucciones que conforman a las tareas de *LPC* mediante ecuaciones de transformación. Ello requiere que cada proceso "cargue con su copia del código", esto es, que cada proceso tenga su propia copia de la tarea asignada. El nombre de la tarea que conforma al proceso es necesario para las comunicaciones de *LPC*. La existencia de parámetros en la definición de las tareas genera la presencia de un ambiente en los procesos.

El significado de la forma sintáctica LlamadaDeTarea corresponde al de la llamada de procedimiento en lenguajes de tipo *Algol*. Se guardan la tarea, su nombre y su ambiente, y se ejecuta la nueva tarea en su nuevo ambiente. Al terminar la nueva tarea, se restaura lo guardado y se prosigue con la ejecución. Ello implica que la forma del proceso permita definir las operaciones de "guardar" y de "restaurar". Se utiliza una pila para expresar formalmente estas operaciones.

P.25 **PILA***TAR* ≡ **seq** (**ID***TAR* × *TAREA*)

P.26 **PILA***AMB* ≡ **seq** *AMBIENTE*

Así, un proceso queda definido por un robot, una pila de tareas y sus nombres y una pila de ambientes.

P.27 *PROCESO* ≡ *ROBOT* × PILA*TAR* × PILA*AMB*

Si $p \in PROCESO$, $p.r$ es el robot que está ejecutando la tarea $\underline{last}(PiT).t$, llamada $\underline{last}(PiT).IdT$, con el ambiente $\underline{last}(PiA)$. Donde *PiT* y *PiA* son las pilas de tareas y de ambientes respectivamente, y $\underline{last}(P)$ es el último elemento de la secuencia P, o sea $P(\#P)$, según la definición encontrada en [Sufrin86].

Esta modificación a la definición de proceso induce correcciones en todas las definiciones que la usan. Nótese que estas modificaciones no provienen de una alteración conceptual, sino que resultan del cambio de nivel de abstracción impuesto por la realización de los procesos de *LPC* en una computadora. Por lo tanto, las correcciones son básicamente de forma y las definiciones conservan prácticamente el sentido que se les diera en el capítulo tercero. Se reproducen los cambios pertinentes y con ello se concluye la sección de las modificaciones.

P.28 CreaProceso:

$$\mathbf{DEC}ROB \times \mathbf{DEC}MUN \times \mathbf{DEC}TAR \times \mathbf{ID}TAR \times AMBIENTE \times ACTIV \downarrow \mathbf{DEC}ROB \times \mathbf{DEC}MUN \times ACTIV$$

$$(DeclR, DeclM, DeclT, A, a) \quad \text{-}\!\!\blacktriangleright \quad (DeclR', DeclM', a')$$

donde, siendo $(DeclR', DeclM', r') = \underline{CreaRobot}(DeclR, A.IdR, DeclM)$

si $DeclM' \neq DeclM$ & $DeclR' \neq DeclR$ (el robot fue creado exitosamente)

siendo $IdR' \mid DeclR'(IdR').r = r'$

$$a' = a \triangle [\ (r'\ ,\ [\ (IdT, DeclT(IdT).T)\ ]\ ,\ [\ A\ ])\ ],$$

de otra manera: $a' = a$

donde $\triangle$ denota al concatenador de secuencias (concat) definido en [Sufrin86].

P.29 TerminaProceso:

$$\mathbf{DEC}MUN \times \mathbf{DEC}ROB \times PROCESO \times ACTIV \longmapsto \mathbf{DEC}MUN \times \mathbf{DEC}ROB \times ACTIV$$

$$(DeclM, DeclR, p, a) \quad \text{-}\!\!\blacktriangleright \quad (DeclM', DeclR', a')$$

donde $(DeclM', DeclR') = \underline{DestruyeRobot}(DeclR, DeclM, \underline{last}(p.A).Ejec)$

$$a' = a \bigcirc \underline{Remueve}(k, \#a) \quad \text{donde } k \text{ es tal que } a(k) = p.$$

Los cambios en las funciones de comunicación inciden solamente en la forma.

P.30 Envia: $\mathbf{DEC}CAN \times \mathbf{DEC}ROB \times ACTIV \times \mathbf{ID}CAN \times \mathbf{ID}ROB \times \mathbf{ID}ROB \downarrow \mathbf{DEC}CAN \times ACTIV$

$(DeclC, DeclR, a, IdC, IdRe, IdRr) \quad \rightarrow \quad (DeclC', a')$

donde, siendo $c = DeclC(IdC)$ siempre que ello tenga sentido,

si existe $k \mid c.e(k) = (IdRe, p)$ con $\underline{\text{last}}(p.A).Ejec = IdRr$:

$a' = a \blacktriangle [\ p\ ]$ y $DeclC' = DeclC \cup \{\ IdC \rightarrow (c.a\ ,\ c.e \bigcirc \underline{\text{Remueve}}(k, \#c.e))\ \}$,

de otro modo, si $IdC \in \underline{\text{dom}}\ DeclC$ e $IdRe \in \underline{\text{dom}}\ DeclR$:

$a' = a$ y $DeclC' = DeclC \cup \{\ IdC \rightarrow (c.a \blacktriangle [\ (IdRe, IdRr)\ ]\ ,\ c.e)\ \}$ y

en caso contrario: $a' = a$ y $DeclC' = DeclC$.

P.31 Recibe: $\mathbf{DEC}CAN \times \mathbf{DEC}ROB \times ACTIV \times \mathbf{ID}CAN \times \mathbf{ID}ROB \times \mathbf{ID}ROB \downarrow \mathbf{DEC}CAN \times ACTIV$

$(DeclC, DeclR, a, IdC, IdRe, IdRr) \quad \rightarrow \quad (DeclC', a')$

donde

si $IdC \in \underline{\text{dom}}\ DeclC$, $IdRe, IdRr \in \underline{\text{dom}}\ DeclR$, $c = DeclC(IdC)$ y

existe $k = \underline{\text{min}}\ \{\ i \in N \mid c.a(i) = (IdRe, IdRr)\ \}$:

$a' = a$ y $DeclC' = DeclC \cup \{\ IdC \rightarrow (c.a \bigcirc \underline{\text{Remueve}}(k, \#c.a)\ ,\ c.e)\ \}$,

de otro modo,

si $IdC \in \underline{\text{dom}}\ DeclC$, $IdRr \in \underline{\text{dom}}\ DeclR$, $c = DeclC(IdC)$ y $k \mid \underline{\text{last}}(a(k).A).Ejec = IdRr$:

$a' = a \bigcirc \underline{\text{Remueve}}(k, \#a)$ y

$DeclC' = DeclC \cup \{\ IdC \rightarrow (c.a\ ,\ c.e \blacktriangle [\ (IdRe, a(k))\ ])\ \}$ y

en caso contrario: $a' = a$ y $DeclC' = DeclC$.

P.32 Solicita:

$\mathbf{DEC}BUZ \times \mathbf{DEC}ROB \times \mathbf{DEC}TAR \times ACTIV \times \mathbf{ID}BUZ \times \mathbf{ID}ROB \times \mathbf{ID}TAR \downarrow \mathbf{DEC}BUZ \times ACTIV$

$(DeclB, DeclT, DeclR, a, IdB, IdRe, IdTr) \quad \rightarrow \quad (DeclB', a')$

donde, siendo $b = DeclB(IdB)$, $j \mid \underline{\text{last}}(a(j).A).Ejec = IdRe$ e $IdTe = \underline{\text{last}}(a(j).PiT).IdT$

siempre que ello tenga sentido,

si existe k = min { i | $(b.e(i) = (IdTe, p)$ & last$(p.PiT).IdT = IdTr)$ }:

$a' = a$ ▲ [p] y

$DeclB' = DeclB$ Ù { $IdB \rightarrow (b.a$, $b.e$ ○ Remueve$(k, \#b.e))$ },

de otro modo, si $IdB \in$ dom $DeclB$, $IdTr \in$ dom $DeclT$ e $IdRe \in$ dom $DeclR$:

$a' = a$ y

$DeclB' = DeclB$ Ù { $IdB \rightarrow (b.a$ ▲ [$(IdTe, IdTr)$]$), b.e)$ } y

en caso contrario: $a' = a$ y $DeclB' = DeclB$.

P.33 Atiende:

DECBUZ × **DECTAR** × **DECROB** × *ACTIV* × **IDBUZ** × **IDTAR** × **IDROB**
↓
DECBUZ × *ACTIV*

$(DeclB, DeclT, DeclR, a, IdB, IdTe, IdRr) \rightarrow (DeclB', a')$

donde, siendo $b = DeclB(IdB)$ y j | last$(a(j).A).Ejec = IdRr$ e $IdTr =$ last$(a(j).PiT).IdT$

siempre que ello tenga sentido,

si $IdB \in$ dom $DeclB$, $IdTe \in$ dom $DeclT$, $IdRr \in$ dom $DeclR$ y

existe k = min { i | $b.a(i) = (IdTe, IdTr)$ }:

$a' = a$ y $DeclB' = DeclB$ Ù { $IdB \rightarrow (b.a$ ○ Remueve$(k, \#b.a)$, $b.e)$ },

de otro modo,

si $IdB \in$ dom $DeclB$, $IdTe \in$ dom $DeclT$ e $IdRr \in$ dom $DeclR$:

$a' = a$ ○ Remueve$(j, \#a)$ y

$DeclB' = DeclB$ Ù { $IdB \rightarrow (b.a$, $b.e$ ▲ [$(IdTe, a(j))$]$)$ } y

en caso contrario: $a' = a$ y $DeclB' = DeclB$.

A continuación, se formaliza el contexto de las formas sintácticas adoptando la misma agrupación que la del cuarto capítulo. Dos tablas encabezan la exposición de cada grupo. La primera tabla nombra los atributos introducidos por el grupo e indica sus dominios (semánticos) respectivos. La segunda apunta los atributos sintetizados y los heredados por cada nodo no terminal del grupo. Cada forma sintáctica es "decorada" con las reglas de evaluación de los atributos involucrados. La transformación del miembro derecho de la forma sintáctica al izquierdo queda sujeta a que se satisfaga la clausula opcional Condición. Esta clausula es una expresión boole-

ana que dicta las propiedades que habrán de cumplir los valores de dichos atributos para que la forma sintáctica "decorada" en cuestión sea válida.

5.2 Un programa

Otra vez, se comienza por los identificadores y los enteros. Se introducen tres atributos:

Atributo	Valores
Longitud	*Enteros* [1 .. `MáximaLongitud` (= 16)]
Etiqueta	*IDENTIFICADOR* (T.3)
Valor	*TROMPO* (M.32)

No terminal	Atributo sintetizado
Id	Longitud, Etiqueta
Alfa	Longitud, Etiqueta
Digito	Longitud, Etiqueta, Valor
Entero	Valor.

Id^1 ::= Alfa | Id^2 Alfa | Id^2 Digito | Id^1 _
Condición: Longitud(Id^1) ≤ `MáximaLongitud`
Longitud(Id^1) <- Longitud(Alfa) | Longitud(Id^2) + Longitud(Alfa) |
Longitud(Id^2) + Longitud(Digito) | Longitud(Id^2) + 1
Etiqueta(Id^1) <- Etiqueta(Alfa) | Etiqueta(Id^2) ▲ Etiqueta(Alfa) |
Etiqueta(Id^2) ▲ Etiqueta(Digito) | Etiqueta(Id^2) ▲ ["_"].

Alfa ::= **a** | .. | **z** | **A** | .. | **Z**
Etiqueta(Alfa) <- ["A"] | .. | ["Z"] | ["A"] | .. | ["Z"]
Longitud(Alfa) <- 1

Digito ::= **0** | .. | **9**
Etiqueta(Digito) <- ["0"] | .. | ["9"]
Valor(Digito) <- 0 | .. | 9
Longitud(Digito) <- 1

$Entero^1$ ::= Digito | $Entero^2$ Digito
Condición: Valor($Entero^1$) ≤ `MáximoTrompos`
Valor($Entero^1$) <- Valor(Digito) | 10 * Valor($Entero^2$) + Valor(Digito).

Cabe notar que los identificadores de los cuales se ha hablado a lo largo del capítulo tercero y en la sección 5.1 son entes semánticos, esto es, son secuencias de caracteres. Los que se han mencionado en el cuarto capítulo, son objetos sintácticos a los cuales el atributo Etiqueta asocia sus correspondientes semánticos. Obsérvese que la especificación de los identificadores de *LPC* no marca distinción alguna entre letras mayúsculas y minúsculas. Las palabras reservadas siguen la misma regla.

El grupo de reglas sintácticas encabezado por la de Programa introduce dieciseis atributos cuyos valores pertenecen precisamente a los conjuntos definidos en las secciones anteriores. Algunos de estos figurarán en el capítulo sexto al expresar el medio ambiente de un programa en *LPC*.

Atributo	Valores	Atributo	Valores
DeclC,CLoc	**DEC***CAN* (P.7)	DeclB,BLoc	**DEC***BUZ* (P.8)
DeclR,RGlob,RLoc	**DEC***ROB* (R.40)	DeclM	**DEC***MUN* (M.30)
DeclT	**DEC***TAR* (T.7)	Activ	*ACTIV* (P.2)
Ambiente,Madre	*AMBIENTE* (T.10)	ParFor	**PAR***FORMAL*(T.6)
EaPrg	*BOOLEANO*	DirRel	*DIRREL* (R.9)
Par	*PARAM* (T.9)		

No terminal	Atributo heredado	Atributo sintetizado
Programa		DeclC,DeclB,DeclR,DeclM,DeclT,Activ.
ListaDeDeclaraciones		DeclC,DeclB,RGlob,DeclM,DeclT.
ListaDeAsignaciones	RGlob,DeclT.	Activ.
Comunicaciones	DeclC,DeclB.	CLoc,BLoc.
ListaDeRobots	DeclC,DeclB,DeclT,EaPrg,ParFor.	RGlob,DeclM,RLoc.
ListaDeTareas	DeclC,DeclB,RGlob.	DeclM,DeclT.
Asignacion	RGlob,DeclT.	Activ,Ambiente.
ListaDeParametrosActuales	DeclT,RGlob,ParFor,RLoc,EaPrg,Madre.	Par.
ParametroActual	DeclT,RGlob,ParFor,RLoc,EaPrg,Madre.	Par.
DireccionRelativa		DirRel.

Programa ::= **programa** Id^1 ; ListaDeDeclaraciones **es** ListaDeAsignaciones **fin** Id^2 .
Condición : Etiqueta(Id^1) = Etiqueta(Id^2)
DeclC(Programa) <- DeclC(ListaDeDeclaraciones), DeclB(Programa) <- DeclB(ListaDeDeclaraciones),
DeclR(Programa) <- RGlob(ListaDeDeclaraciones), DeclM(Programa) <- DeclM(ListaDeDeclaraciones),
DeclT(Programa) <- DeclT(ListaDeDeclaraciones), Activ(Programa) <- Activ(ListaDeAsignaciones),
RGlob(ListaDeAsignaciones) <- DeclR(Programa), DeclT(ListaDeAsignaciones) <- DeclT(Programa).

ListaDeDeclaraciones ::= /{ Comunicaciones ; }/ ListaDeRobots ListaDeTareas
DeclC(ListaDeDeclaraciones) <- ∅/{ ∪ CLoc(Comunicaciones) }/,
DeclC(Comunicaciones) <- DeclC(ListaDeDeclaraciones),
DeclB(ListaDeDeclaraciones) <- ∅/{ ∪ BLoc(Comunicaciones) }/,
DeclB(Comunicaciones) <- DeclB(ListaDeDeclaraciones), EnPrg(ListaDeRobots) <- v,
DeclM(ListaDeDeclaraciones) <- DeclM(ListaDeRobots) ∩ DeclM(ListaDeTareas),
ParFor(ListaDeRobots) <- [ ],
RGlob(ListaDeDeclaraciones) <- RGlob(ListaDeRobots),
DeclT(ListaDeDeclaraciones) <- DeclT(ListaDeTareas),
DeclC(ListaDeTareas) <- DeclC(ListaDeDeclaraciones),
DeclB(ListaDeTareas) <- DeclB(ListaDeDeclaraciones),
RGlob(ListaDeTareas) <- RGlob(ListaDeDeclaraciones),
DeclC(ListaDeRobots) <- DeclC(ListaDeDeclaraciones),
DeclB(ListaDeRobots) <- DeclB(ListaDeDeclaraciones),
DeclT(ListaDeRobots) <- DeclT(ListaDeDeclaraciones).

donde ∅ denota al conjunto vacío.

ListaDeAsignaciones^1 ::= Asignacion // ; ListaDeAsignaciones^2 //
Activ(ListaDeAsignaciones^1) <- Activ(Asignacion) // ∘ Activ(ListaDeAsignaciones^2) //,
RGlob(Asignacion) <- RGlob(ListaDeAsignaciones^1), DeclT(Asignacion) <- DeclT(ListaDeAsignaciones^1).

Asignacion ::= Id^1 := Id^2 // (ListaDeParametrosActuales) //
condición :
Etiqueta(Id^1) ∈ dom RGlob(Asignacion) & Etiqueta(Id^2) ∈ dom DeclT(Asignacion) &
¬ RGlob(Asignacion)(Etiqueta(Id^1)).Asignado &
DeclT(Asignacion)(Etiqueta(Id^2)).Pf = 0 // + # Par(ListaDeParametrosActuales) //.
RGlob(Asignacion)(Etiqueta(Id^1)).Asignado <- v,
// ParFor(ListaDeParametrosActuales) <- DeclT(Asignacion)(Etiqueta(Id^2)).Pf //,
// DeclT(ListaDeParametrosActuales) <- DeclT(Asignacion) //,
// RGlob(ListaDeParametrosActuales) <- RGlob(Asignacion) //,
// RLoc(ListaDeParametrosActuales) <- ∅ //, // EnPrg(ListaDeParametrosActuales) <- v //,
// Madre(ListaDeParametrosActuales) <- ∅ //,
Ambiente(Asignacion) <- (Ejec, RobotsLocales, ∅ // ∪ Par(ListaDeParametrosActuales) //),
Activ(Asignacion) <- [Proceso].

donde Ejec = Etiqueta(Id^1),
RobotsLocales = Renombra(DeclT(Asignacion)(Etiqueta(Id^2)).RI),
Proceso = (RGlob(Asignacion)(Etiqueta(Id^1)).r, PilaTarea, PilaAmbiente),
PilaTarea = [(Etiqueta(Id^2), DeclT(Asignacion)(Etiqueta(Id^2)).t)],
PilaAmbiente = [Ambiente(Asignacion)] y

Renombra es una función que renombra a cada uno de los robots, incluidos en la declaración de robots que toma por argumento, de manera que sus nuevos nombres sean únicos.

ListaDeParametrosActuales^1 ::= ParametroActual // , ListaDeParametrosActuales^2 //
condición : (Par(ParametroActual).Tipo = head(ParFor(ListaDeParametrosActuales^1)).Tipo)
Par(ListaDeParametrosActuales^1) <- Par(ParametroActual) // ∪ Par(ListaDeParametrosActuales^2) //,
DeclT(ParametroActual) <- DeclT(ListaDeParametrosActuales^1),
RGlob(ParametroActual) <- RGlob(ListaDeParametrosActuales^1),
RLoc(ParametroActual) <- RLoc(ListaDeParametrosActuales^1),
ParFor(ParametroActual) <- [head(ParFor(ListaDeParametrosActuales^1))],
// ParFor(ListaDeParametrosActuales^2) <- tail(ParFor(ListaDeParametrosActuales^1)) //,
EnPrg(ParametroActual) <- EnPrg(ListaDeParametrosActuales^1),
Madre(ParametroActual) <- Madre(ListaDeParametrosActuales^1).

ParametroActual ::= Id
Siendo (IdPf , Tipo) = head(ParFor(ParametroActual)), IdPa = Etiqueta(Id),
L = RLoc(ParametroActual), R = RGlob(ParametroActual), T = DeclT(ParametroActual) y
M = Madre(ParametroActual).

condición : (Tipo = ROBVAR) & EnPrg(ParametroActual) & ¬ R(IdPa).Asignado.
Par(ParametroActual) <- { IdPf <-> (IdPa,ROBVAR) }, R(IdPa).Asignado <- v.
condición : (Tipo = ROBVAR) & ¬ EnPrg(ParametroActual) & IdPa ∈ dom L.
Par(ParametroActual) <- { IdPf <-> (IdPa,ROBVAR) }.
condición : (Tipo = ROBPAR) & ((M.Par(IdPa).Tipo = ROBVAR) ó (M.Par(IdPa).Tipo = ROBPAR)).
Par(ParametroActual) <- { IdPf <-> (M.Par(IdPa).Val,ROBPAR) }.
condición : (Tipo = ROBPAR) & (IdPa ∈ dom L).
Par(ParametroActual) <- { IdPf <-> (IdPa,ROBPAR) }.
condición : (Tipo = TARVAR) & (IdPa ∈ dom T).
Par(ParametroActual) <- { IdPf <-> (IdPa,TARVAR) }.
condición : (Tipo = TARVAR) ó (Tipo = DIRPAR) ó (Tipo = ROBVAR).
Par(ParametroActual) <- { IdPf <-> M.Par(IdPa) }.
condición : (Tipo = TARPAR) & (IdPa ∈ dom T).
Par(ParametroActual) <- { IdPf <-> (IdPa,TARPAR) }.
condición : (Tipo = TARPAR) & (M.Par(IdPa).Tipo = TARPAR ó M.Par(IdPa).Tipo = TARVAR).
Par(ParametroActual) <- { IdPf <-> (M.Par(IdPa).Val,TARPAR) }.
| yo
condición : ¬ EnPrg(ParametroActual) & (Tipo = ROBVAR ó Tipo = ROBPAR).
Par(ParametroActual) <- { IdPf <-> (M.Ejec,Tipo) }.
| DireccionRelativa
condición : Tipo = DIRPAR.
Par(ParametroActual) <- { IdPf <-> (DirRel(DireccionRelativa),DIRPAR) }.

DireccionRelativa ::= **proa | estribor | popa | babor**
DirRel(DireccionRelativa) <- Proa | Estribor | Popa | Babor.

5.3 Las declaraciones

El grupo de las declaraciones de las líneas de comunicación no introduce atributo alguno. Las reglas de este grupo sintetizan los atributos CLoc y BLoc.

No terminal	Atributo heredado	Atributo sintetizado
Comunicaciones	DeclC,DeclB.	CLoc,BLoc.
DeclaracionCanal	DeclC,DeclB.	CLoc.
DeclaracionBuzon	DeclC,DeclB.	BLoc.

Comunicaciones ::= DeclaracionCanal | DeclaracionBuzon
CLoc(Comunicaciones) <- CLoc(DeclaracionCanal), BLoc(Comunicaciones) <- BLoc(DeclaracionBuzon),
DeclC(DeclaracionCanal) <- DeclC(Comunicaciones), DeclC(DeclaracionBuzon) <- DeclC(Comunicaciones),
DeclB(DeclaracionBuzon) <- DeclB(Comunicaciones), DeclB(DeclaracionCanal) <- DeclB(Comunicaciones).

DeclaracionCanal ::= **canal** : Id^1 /{ , Id^2 }/
condición: /{ Etiqueta(Id^3) $\neg\in$ dom CLoc(DeclaracionCanal) }/ &
Etiqueta(Id^1) $\neg\in$ dom DeclC(DeclaracionCanal) &
Etiqueta(Id^1) $\neg\in$ dom DeclB(DeclaracionCanal)
/{ & Etiqueta(Id^2) $\neg\in$ dom DeclC(DeclaracionCanal) &
Etiqueta(Id^2) $\neg\in$ dom DeclB(DeclaracionCanal) }/.
CLoc(DeclaracionCanal) <- CreaCanal($\emptyset$, Etiqueta(Id^1)),
/{ CLoc(DeclaracionCanal) <- CreaCanal(CLoc(DeclaracionCanal) , Etiqueta(Id^2)) }/.

DeclaracionBuzon ::= **buzon** : Id^1 /{ , Id^2 }/
condición: /{ Etiqueta(Id^3) $\neg\in$ dom BLoc(DeclaracionBuzon) }/ &
Etiqueta(Id^1) $\neg\in$ dom DeclC(DeclaracionBuzon) &
Etiqueta(Id^1) $\neg\in$ dom DeclB(DeclaracionBuzon)
/{ & Etiqueta(Id^2) $\neg\in$ dom DeclC(DeclaracionBuzon) &
Etiqueta(Id^2) $\neg\in$ dom DeclB(DeclaracionBuzon) }/.
BLoc(DeclaracionBuzon) <- CreaBuzon($\emptyset$, Etiqueta(Id^1)),
/{ BLoc(DeclaracionBuzon) <- CreaBuzon(BLoc(DeclaracionBuzon) , Etiqueta(Id^2)) }/.

Donde CreaCanal y CreaBuzon son las funciones definidas en la sección 3.2.4 bajo los números P.17 y P.18. El siguiente grupo es el de la declaración de robots que sintetiza los atributos RGlob, RLoc, y DeclM.

Atributo	Valores	Atributo	Valores
MLoc	**DEC***MUN* (M.30)	RAux	**DEC***ROB* (R.40)
Cl	*CRUCE* (M.3)	Di	*DIRECCION* (R.6)
Ti	*TROMPO* (M.32).		

No terminal	Atributo heredado	Atributo sintetizado
ListaDeRobots	DeclC,DeclB,DeclT,EnPrg,ParFor.	RGlob,DeclM,RLoc.
Robots	DeclC,DeclB,DeclT,DeclM,EnPrg,ParFor.	RGlob,MLoc,RLoc.
Robot	DeclC,DeclB,DeclT,DeclM,EnPrg,RGlob,RLoc,ParFor.	RAux,MLoc.
CondicionesIniciales		Cl,Di,Ti.

ListaDeRobots^1 ::= Robots ; // ListaDeRobots^2 //
DeclC(Robots) <- DeclC(ListaDeRobots^1), DeclB(Robots) <- DeclB(ListaDeRobots^1),
DeclT(Robots) <- DeclT(ListaDeRobots^1), EnPrg(Robots) <- EnPrg(ListaDeRobots^1),
DeclM(Robots) <- DeclM(ListaDeRobots^1), ParFor(Robots) <- ParFor(ListaDeRobots^1),
RGlob(ListaDeRobots^1) <- RGlob(Robots) // $\cup$ RGlob(ListaDeRobots^2) //,
DeclM(ListaDeRobots^1) <- MLoc(Robots) // $\emptyset$ DeclM(ListaDeRobots^2) //,
RLoc(ListaDeRobots^1) <- RLoc(Robots) // $\cup$ RLoc(ListaDeRobots^2) //.

Robots ::= robot : Robot^1 /{ , Robot^2 }/
DeclC(Robot^1) <- DeclC(Robots), DeclB(Robot^1) <- DeclB(Robots), DeclT(Robot^1) <- DeclT(Robots),
EnPrg(Robot^1) <- EnPrg(Robots), DeclM(Robot^1) <- DeclM(Robots), RGlob(Robot^1) <- RGlob(Robots),
RLoc(Robot^1) <- RLoc(Robots), ParFor(Robot^1) <- ParFor(Robots),
/{ DeclC(Robot^2) <- DeclC(Robots), DeclB(Robot^2) <- DeclB(Robots), DeclT(Robot^2) <- DeclT(Robots),
EnPrg(Robot^2) <- EnPrg(Robots), DeclM(Robot^2) <- DeclM(Robots), RGlob(Robot^2) <- RGlob(Robots),
RLoc(Robot^2) <- RLoc(Robots), ParFor(Robot^2) <- ParFor(Robots) /},
MLoc(Robots) <- MLoc(Robot^1) /{ $\emptyset$ MLoc(Robot^2) }/,
condición : EnPrg(Robots)
RGlob(Robots) <- RAux(Robot^1) /{ $\cup$ RAux(Robot^2) }/, RLoc(Robots) <- $\emptyset$.
condición : $\neg$ EnPrg(Robots)
RLoc(Robots) <- RAux(Robot^1) /{ $\cup$ RAux(Robot^2) }/, RGlob(Robots) <- $\emptyset$.

Robot ::= Id¹ **comparte** Id² CondicionesIniciales
Siendo IdR = Etiqueta(Id¹), IdM = Etiqueta(Id²), x = Ci(CondicionesIniciales), d = Di(CondicionesIniciales) y t = Ti(CondicionesIniciales).
declm = DeclM(Robot)(IdM) si IdM ∈ dom DeclM(Robot)
declm = CargaMundo(IdM) de otro modo y
m = declm.m, A = declm.A y L = declm.L

Condición : IdR ¬∈ dom DeclC(Robot) & IdR ¬∈ dom DeclB(Robot)
& IdR ¬∈ dom DeclT(Robot)
& no existe k | ParFor(Robot)(k).IdPf = IdR
& ((IdR ¬∈ dom RGlob(Robot) & EnPrg(Robot)) ó
(IdR ¬∈ dom RLoc(Robot) & ¬ EnPrg(Robot)))
& (1 ≤ x.a ≤ A) & (1 ≤ x.c ≤ L) & ¬ HayRobot(m,x) & ¬ HayMuro(m,x)
MLoc(Robot) <- (IdM <-» (PonRobot(m,x) , A , L)),
RAux(Robot) <- (IdR <-» ((IdM,x,d,t) , x , y)).

| Id¹ **habita** Id² CondicionesIniciales
Condición : IdR ¬∈ dom DeclC(Robot) & IdR ¬∈ dom DeclB(Robot)
& IdR ¬∈ dom DeclT(Robot)
& no existe k | ParFor(Robot)(k).IdPf = IdR
& ((IdR ¬∈ dom RGlob(Robot) & EnPrg(Robot)) ó
(IdR ¬∈ dom RLoc(Robot) & ¬ EnPrg(Robot)))
& (1 ≤ x.a ≤ A) & (1 ≤ x.c ≤ L) & ¬ HayMuro(m,x)
MLoc(Robot) <- (IdM <-» declm),
RAux(Robot) <- (IdR <-» ((IdM,x,d,t) , x , x)).

CondicionesIniciales ::= **en** Entero¹ , Entero² **norte** DireccionRelativa **con** Entero³
Ci(CondicionesIniciales) <- (Valor(Entero¹), Valor(Entero²)),
Di(CondicionesIniciales) <- DirRel(DireccionRelativa)⁻¹(*Norte*),
Ti(CondicionesIniciales) <- Valor(Entero³).

A continuación, se presenta el grupo de formas sintácticas con las que se declaran las tareas. Se introducen dos atributos auxiliares TLoc y Aux. En TLoc se forma la declaración de cada tarea antes de acumularla en *DeclT*. En Aux se constituye cada parámetro formal antes de coleccionarlo en ParFor. La forma sintáctica INST no sintetiza atributo alguno; en cambio, hereda todas las declaraciones "visibles" e introduce un nuevo atributo: EnCiclo, que registrará si la forma sintáctica va inclusa en un ciclo.

Atributo	Valores	Atributo	Valores
TLoc	**DECT***AR* (T.7)	Aux	**PAR***FORMAL* (T.6)
EnCiclo	*BOOLEANO*.		

No terminal	Atributo heredado	Atributo sintetizado
ListaDeTareas	DeclC,DeclB,RGlob.	DeclM,DeclT.
Tarea	DeclC,DeclB,DeclT,RGlob.	TLoc,DeclM.
INST	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
ParametrosFormales	DeclC,DeclB,DeclT,RGlob.	ParFor.
ParametroFormal	DeclC,DeclB,DeclT,RGlob,ParFor.	Aux.
PasaDerechoDeUso	DeclC,DeclB,DeclT,RGlob,ParFor.	Aux.
PasaNombre	DeclC,DeclB,DeclT,RGlob,ParFor.	Aux.

ListaDeTareas^1 ::= Tarea ; // ListaDeTareas^2 //
DeclC(Tarea) <- DeclC(ListaDeTareas^1), DeclB(Tarea) <- DeclB(ListaDeTareas^1),
DeclT(Tarea) <- DeclT(ListaDeTareas^1), RGlob(Tarea) <- RGlob(ListaDeTareas^1),
DeclM(ListaDeTareas^1) <- DeclM(Tarea) // ∪ DeclM(ListaDeTareas^2) //,
DeclT(ListaDeTareas^1) <- Tloc(Tarea) // ∪ DeclT(ListaDeTareas^2) //.

Tarea ::= **tarea** Id^1 // (ParametrosFormales) /// ListaDeRobots // **es** INST^1 /{ ; INST^2 }/ **fin** Id^2
Siendo IdT = Etiqueta(Id^1),
condición : IdT ¬∈ dom DeclC(Tarea) & IdT ¬∈ dom DeclB(Tarea) &
IdT ¬∈ dom DeclT(Tarea) & IdT ¬∈ dom RGlob(Tarea) & IdT = Etiqueta(Id^2).
TLoc(Tarea) <- (IdT ↦ (Codigo , ParFor , RLoc)),
DeclM(Tarea) <- ∅ // ∪ DeclM(ListaDeRobots) //,
// DeclC(ParametrosFormales) <- DeclC(Tarea), DeclB(ParametrosFormales) <- DeclB(Tarea),
DeclT(ParametrosFormales) <- DeclT(Tarea), RGlob(ParametrosFormales) <- RGlob(Tarea), //
// DeclC(ListaDeRobots) <- DeclC(Tarea), DeclB(ListaDeRobots) <- DeclB(Tarea),
DeclT(ListaDeRobots) <- DeclT(Tarea), EnPrg(ListaDeRobots) <- F,
ParFor(ListaDeRobots) <- ParFor //,
DeclC(INST^1)/{ , DeclC(INST^2) }/ <- DeclC(Tarea), DeclB(INST^1)/{ , DeclB(INST^2) }/ <- DeclB(Tarea),
DeclT(INST^1)/{ , DeclT(INST^2) }/ <- DeclT(Tarea), RLoc(INST^1)/{ , RLoc(INST^2) }/ <- RLoc,
ParFor(INST^1)/{ , ParFor(INST^2) }/ <- ParFor,
EnCiclo(INST^1)/{ , EnCiclo(INST^2) }/ <- F.

donde Codigo = [INST^1]/{ ▵ [INST^2] }/,
ParFor = [ ] // ▵ ParFor(ParametrosFormales) //,
RLoc = ∅ // ∪ RLoc(ListaDeRobots) //.

ParametrosFormales^1 ::= ParametroFormal // ; ParametrosFormales^2 //
DeclC(ParametroFormal) <- DeclC(ParametrosFormales^1),
DeclB(ParametroFormal) <- DeclB(ParametrosFormales^1),
DeclT(ParametroFormal) <- DeclT(ParametrosFormales^1),
RGlob(ParametroFormal) <- RGlob(ParametrosFormales^1),
ParFor(ParametroFormal) <- ParFor(ParametrosFormales^1),
ParFor(ParametrosFormales^1) <- Aux(ParametroFormal) // ▵ ParFor(ParametrosFormales^2 //.

```
ParametroFormal ::= PasaDerechoDeUso | PasaNombre
        DeclC(PasaDerechoDeUso) | DeclC(PasaNombre) <- DeclC(ParametroFormal),
        DeclB(PasaDerechoDeUso) | DeclB(PasaNombre) <- DeclB(ParametroFormal),
        DeclT(PasaDerechoDeUso) | DeclT(PasaNombre) <- DeclT(ParametroFormal),
        RGlob(PasaDerechoDeUso) | RGlob(PasaNombre) <- RGlob(ParametroFormal),
        ParFor(PasaDerechoDeUso) | ParFor(PasaNombre) <- ParFor(ParametroFormal),
        Aux(ParametroFormal) <- Aux(PasaDerechoDeUso) | Aux(PasaNombre).


PasaDerechoDeUso ::= var Id¹ /{ , Id² }/ : robot | var Id¹ /{ , Id² }/ : tarea
        Siendo IdPf¹ = Etiqueta(Id¹)/{ e IdPf² = Etiqueta(Id²) /},
                condición : IdPf¹ /{ ,IdPf² }/ ¬∈ dom DeclC(PasaDerechoDeUso)
                        & IdPf¹ /{ ,IdPf² }/ ¬∈ dom DeclB(PasaDerechoDeUso)
                        & IdPf¹ /{ ,IdPf² }/ ¬∈ dom DeclT(PasaDerechoDeUso)
                        & IdPf¹ /{ ,IdPf² }/ ¬∈ dom RGlob(PasaDerechoDeUso)
                        & no existe k | Aux(PasaDerechoDeUso)(k).IdP = IdPf¹ /{ ,IdPf² }/
                        & no existe k | ParFor(PasaDerechoDeUso)(k).IdPf = IdPf¹ /{ ,IdPf² }/.
                Aux(PasaDerechoDeUso) <- [ (IdPf¹,ROBVAR) ] /{ ▲ [ (IdPf²,ROBVAR) ]
        |       Aux(PasaDerechoDeUso) <- [ (IdPf¹,TARVAR) ] /{ ▲ [ (IdPf²,TARVAR) ].


PasaNombre ::= Id¹ /{ , Id² }/ : robot | Id¹ /{ , Id² }/ : tarea | Id¹ /{ , Id² }/ : dirrel
        Siendo IdPf¹ = Etiqueta(Id¹)/{ e IdPf² = Etiqueta(Id²) /},
                condición : IdPf¹ /{ ,IdPf² }/ ¬∈ dom DeclC(PasaNombre)
                        & IdPf¹ /{ ,IdPf² }/ ¬∈ dom DeclB(PasaNombre)
                        & IdPf¹ /{ ,IdPf² }/ ¬∈ dom DeclT(PasaNombre)
                        & IdPf¹ /{ ,IdPf² }/ ¬∈ dom RGlob(PasaNombre)
                        & no existe k | Aux(PasaNombre)(k).IdPf = IdPf¹ /{ ,IdPf² }/
                        & no existe k | ParFor(PasaNombre)(k).IdPf = IdPf¹ /{ ,IdPf² }/.
                Aux(PasaNombre) <- [ (IdPf¹,ROBPAR) ] /{ ▲ [ (IdPf²,ROBPAR) ]
        |       Aux(PasaNombre) <- [ (IdPf¹,TARPAR) ] /{ ▲ [ (IdPf²,TARPAR) ]
        |       Aux(PasaNombre) <- [ (IdPf¹,DIRPAR) ] /{ ▲ [ (IdPf²,DIRPAR) ].
```

5.4 Las instrucciones

En el caso de las instrucciones, sólo se exhiben las reglas de evaluación de atributos heredados cuando difieren de la regla trivial de herencia que a continuación se expone:

NoTerminalIzquierdo ::= $NoTerminalDerecho_1$ | ... | $NoTerminalDerecho_n$
DeclC($NoTerminalDerecho_i$) <- DeclC(NoTerminalIzquierdo),
DeclB($NoTerminalDerecho_i$) <- DeclB(NoTerminalIzquierdo),
...
EnCiclo($NoTerminalDerecho_i$) <- EnCiclo(NoTerminalIzquierdo).
para toda i | 1 ≤ i ≤ n.

No terminal	Atributo heredado	Atributo sintetizado
INST	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
EnvioDeMensajes	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
Alternativa	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
Iteracion	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
ComandoCustodiado	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
Salida	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
Guardia	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
TerminoBooleano	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
FactorBooleano	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
LlamadaDeTarea	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
CreacionDeProceso	DeclC,DeclB,DeclT,RLoc,ParFor,EnCiclo.	
ListaDeParametrosActuales	DeclT,RGlob,ParFor,RLoc,EnPrg,Madre.	Par.

INST ::= **nada** | **avanza**

| **gira** Id
condición: Existe k | ParFor(INST)(k) = (Etiqueta(Id),DIRPAR).

| **gira** DireccionRelativa | **recoge** | **deja** | EnvioDeMensajes | Alternativa | Iteracion

| **rompe**
condición: EnCiclo(INST).

| LlamadaDeTarea | CreacionDeProceso

EnvioDeMensajes ::= Id^1 ! Id^2 | Id^1 ? Id^2
Sean AoDe = Etiqueta(Id^1) y Por = Etiqueta(Id^2),
condición: ((AoDe ∈ **dom** RLoc(EnvioDeMensajes) ó existe k | ParFor(EnvioDeMensajes)(k) = (AoDe,ROBPAR)
ó existe k | ParFor(EnvioDeMensajes)(k) = (AoDe,ROBVAR))
& Por ∈ **dom** DeclC(EnvioDeMensajes))
ó (AoDe ∈ **dom** DeclT(EnvioDeMensajes) ó existe k | ParFor(EnvioDeMensajes)(k) = (Por,TARPAR)
ó existe k | ParFor(EnvioDeMensajes)(k) = (Por,TARVAR))
& Por ∈ **dom** DeclB(EnvioDeMensajes))).

Alternativa ::= **escoge** ComandoCustodiado /{ | ComandoCustodiado }/// Salida // **fin**

ComandoCustodiado ::= Guardia -> INST > /{ ; INST }/

Salida ::= **ninguna** -> INST > /{ ; INST }/

Guardia ::= TerminoBooleano /{ o TerminoBooleano }/

TerminoBooleano ::= FactorBooleano /{ **y** FactorBooleano }/

FactorBooleano ::= **no** FactorBooleano | (Guardia)

| **remite** Id^1 **en** Id^2
Sean Quien = Etiqueta(Id^1) y Por = Etiqueta(Id^2),
condición : ((Quien ∈ dom RLoc(FactorBooleano) ó existe k | ParFor(FactorBooleano)(k) = (Quien,ROBPAR)
ó existe k | ParFor(FactorBooleano)(k) = (Quien,ROBVAR))
& Por ∈ dom DeclC(FactorBooleano))
ó (Quien ∈ dom DeclT(FactorBooleano) ó existe k | ParFor(FactorBooleano)(k) = (Por,TARPAR)
ó existe k | ParFor(FactorBooleano)(k) = (Por,TARVAR))
& Por ∈ dom DeclB(FactorBooleano))).

| **cargado** | **cierto**

| Predicado Id
condición : Existe k | ParFor(FactorBooleano)(k) = (Etiqueta(Id),DIRPAR).

| Predicado DireccionRelativa

Predicado ::= **norte** | **muro** | **robot** | **trompo**

Iteracion ::= **ciclo** INST /{ ; INST }/ **fin**
EnCiclo(INST) <- v.

LlamadaDeTarea ::= yo := Id // (ListaDeParametrosActuales) //
Siendo IdT = Etiqueta(Id),
Condición:
(IdT ∈ dom DeclT(LlamadaDeTarea) ó existe k | ParFor(LlamadaDeTarea)(k) = (IdT,TARVAR) &
DeclT(LlamadaDeTarea)(IdT).Pf = 0 // + # Par(ListaDeParametrosActuales) //.
// ParFor(ListaDeParametrosActuales) <- DeclT(LlamadaDeTarea)(IdT).Pf //,
// DeclT(ListaDeParametrosActuales) <- DeclT(LlamadaDeTarea) //,
// RGlob(ListaDeParametrosActuales) <- Ø //, // EnPrg(ListaDeParametrosActuales) <- r //,
// RLoc(ListaDeParametrosActuales) <- RLoc(LlamadaDeTarea) //,
// Madre(ListaDeParametrosActuales) <- (Etiqueta(1), Ø, Extiende(ParFor(LlamadaDeTarea))) //.

donde Extiende queda definida como sigue:

P.34 Extiende **PAR***FORMAL* |==> *PARAM*

$$[\ (Idf_1,T_1)\ (Idf_2,T_2)\ \ldots\ (Idf_n,T_n)\ ]$$

$$\downarrow$$

$$\{\ Idf_1 \leftrightarrow (Idf_1,T_1),\ Idf_2 \leftrightarrow (Idf_2,T_2),\ \ldots,\ Idf_n \leftrightarrow (Idf_n,T_n)\ \}.$$

CreacionDeProceso ::= Id^1 := Id^2 // (ListaDeParametrosActuales) //
Siendo IdR = Etiqueta(Id^1) e IdT = Etiqueta(Id^2),
Condición :
(IdR ∈ dom RLoc(CreacionDeProceso) ó existe k | ParFor(CreacionDeProceso)(k) = (IdR,ROBVAR)
& (IdT ∈ dom DeclT(CreacionDeProceso) ó existe k | ParFor(CreacionDeProceso)(k) = (IdT,TARVAR)
& # DeclT(CreacionDeProceso)(IdT).Pf = 0 // + # Par(ListaDeParametrosActuales) //.
// ParFor(ListaDeParametrosActuales) <- DeclT(CreacionDeProceso)(IdT).Pf //,
// DeclT(ListaDeParametrosActuales) <- DeclT(CreacionDeProceso) //,
// RGlob(ListaDeParametrosActuales) <- Ø //, // EnPrg(ListaDeParametrosActuales) <- r //,
// RLoc(ListaDeParametrosActuales) <- RLoc(CreacionDeProceso) //,
// Madre(ListaDeParametrosActuales) <- (IdR, Ø, Extiende(ParFor(CreacionDeProceso))) //.

Concluye la formalización de las aspectos sensibles al contexto del lenguaje de *LPC*. En el siguiente capítulo, se aprovecha el esfuerzo realizado para definir en forma precisa la semántica del lenguaje.

6. El lenguaje de LPC : semántica

En este capítulo, se define en forma precisa la semántica utilizando los objetos construidos en los capítulos anteriores. Una exposición de los formalismos que se emplean para presentar el lenguaje de *LPC* se pueden encontrar en [Pagan81].

En *LPC*, al igual que en muchos otros sistemas, se considera que ejecutar un conjunto de procesos concurrentes equivale, lógicamente al menos, a ejecutar secuencialmente un intercalamiento aleatorio de las acciones que conforman a cada proceso constituyente. Una instrucción de un lenguaje de programación se descompone usualmente en una serie de instrucciones de más bajo nivel, que a su vez posiblemente se ejecuten como micro-instrucciones de una máquina aún más sencilla. Una acción es una "rebanada" de instrucción. Las "rebanadas" pueden ser gruesas o finas. Aquella acción que no se puede descomponer más, se llama acción atómica. Obviamente, la atomicidad de una acción queda sujeta a la máquina física en la cual se vaya a ejecutar. La equivalencia lógica entre una ejecución paralela de procesos concurrentes y una simulación secuencial de éstos, depende del "grueso" relativo de las acciones constitutivas de las dos ejecuciones. Si las acciones de la ejecución secuencial son al menos tan finas como las de la paralela, la equivalencia se cumple y se dice que la ejecución secuencial tiene grano menor o igual al de la ejecución paralela.

La máquina de virtual *LPC* (MV_{LPC}) se compone de un número teóricamente ilimitado de robots. Cada robot es capaz de ejecutar la tarea que se le asigne. Así, MV_{LPC} es una máquina multiprocesador en la cual cada procesador es un robot. Esta máquina es capaz de agregarse o de removerse procesadores bajo el control del programa que esté ejecutando. Las tareas que deben ejecutar los robots se han definido como secuencias cuyos elementos son formas sintácticas del conjunto INST. Se empieza por darles significado a estas formas para definir posteriormente la semántica de un programa completo.

6.1 Las tareas

Se define la semántica de las tareas exhibiendo los cambios de estado que provoca cada forma sintáctica en la máquina huésped, en este caso, la que se ha denominado MV_{LPC}. Se aprovecha así la definición recursiva de las secuencias. En efecto, la definición de *TAREA* permite escribir la siguiente equivalencia:

$$t \in TAREA \iff t = [\,] \text{ ó existe } i \in \text{INST} \mid t = [\,i\,] \,\triangle\, T \text{ con } T \in TAREA^{1},$$

y por lo tanto, para especificar la semántica de las tareas, basta asignar un significado a la secuencia vacía, [], y a un número finito de formas ([i]△T); una para cada elemento de INST.

Lo primero que debe ser caracterizado, es el conjunto *ESTADOS* de los estados de una MV_{LPC}.

S.1 $ESTADOS' \equiv \mathbf{DEC}CAN \times \mathbf{DEC}BUZ \times \mathbf{DEC}ROB \times \mathbf{DEC}MUN \times \mathbf{DEC}TAR \times ACTIV.$

Esta definición se deriva naturalmente de las construcciones anteriores y corresponde a la explicación intuitiva que se diera al presentar el lenguaje de *LPC* en el capítulo cuarto. Se incluye en *ESTADOS'* un estado especial, denotado por *err*, que representa la situación de error. Cuando MV_{LPC} alcanza el estado *err* al ejecutar un programa de *LPC*, se dice que el programa en cuestión contiene un error de programación de tipo semántico.

S.2 $ESTADOS \equiv ESTADOS' \cup \{\, err \,\}.$

El significado de cada instrucción depende del estado de la MV_{LPC}, del robot que la lleve a cabo y de la propia instrucción. La función semántica de las tareas, que se nombra Ejecuta, toma por argumentos un estado, el número del proceso cuyo robot será el ejecutor y la tarea que tiene asignada, y entrega como resultado un estado. Éste refleja el cambio impregnado al estado original por el robot al ejecutar la cabeza de secuencia de la tarea suministrada.

S.3 Ejecuta : $ESTADOS' \times N \times TAREA \longmapsto ESTADOS.$

Resta definir las "reglas de cambio", o ecuaciones semánticas, a las que se sujetan las instrucciones. En lo que sigue, los elementos constitutivos de objetos compuestos se denotan en forma consistente a la que se ha ve-

[1] El símbolo △ sigue denotando al concatenador de secuencias definido en [Sufrin 86].

nido empleando en las secciones anteriores. Por ejemplo, siendo e un elemento de *ESTADOS'*, *e.DeclC* denota la declaración de canales del estado e y *e.Activ[k].PiT* denota la pila de tareas del k-ésimo proceso del estado e.

Ejecuta$(e,k,\ [\ ]\ )$ = *(e.DeclC,e.DeclB,DeclR',DeclM',e.DeclT,Activ')* donde

$$(DeclM',DeclR',Activ') = \begin{cases} (e.DeclM,e.DeclR,Activ'') \text{ si } \underline{front}(e.Activ[k].PiT) \neq [\ ], \\ \underline{TerminaProceso}(e.DeclM,e.DeclR,e.Activ[k],e.Activ) \text{ de otro modo,} \end{cases}$$

Activ'' es idéntico a *e.Activ* excepto que su k-ésimo proceso queda como sigue[2]: *Activ''[k]* = *(e.Activ[k].r,*$\underline{front}$*(e.Activ[k].PiT),*$\underline{front}$*(e.Activ[k].PiA)).*

El significado de una secuencia vacía es pues la terminación exitosa de una tarea. En efecto, el robot no tiene más instrucciones que ejecutar y ha concluido ese trabajo. Si el proceso tiene más tareas en su pila de tareas, el robot ejecutará la del tope; en caso contrario, dejará de existir el proceso. El objetivo de todo robot en *LPC* es vaciar la secuencia de instrucciones que se le ha asignado. Algunas instrucciones son "consumidas" al ser ejecutadas por el robot, otras en cambio, "generan" más instrucciones.

Ejecuta$(e,k,$ [nada] $\frown T\ )$ = e' con e' idéntico a e excepto en la tarea del k-ésimo proceso que queda alterada de la siguiente forma:

$$\underline{last}(e'.Activ[k].PiT).t \leftarrow T.$$

En efecto, en cualquier estado, todo robot que ejecute nada dejará inalterado el estado de la MV_{LPC}, exceptuando a la tarea que tiene asignada, la cual queda acortada en una INST. Ello explica la actualización de $\underline{last}(e'.Activ[k].PiT).t$ para indicar que la secuencia que define a la tarea actual del k-ésimo proceso ha quedado descabezada, o sea, que su primer elemento ha sido "consumido".

Este empleo de la secuencia abstrae el papel que desempeña un contador de instrucciones en un procesador físico, y la secuencia en sí denota el código del proceso en una máquina real. Usualmente, el contador de instrucciones se modela con la composición funcional. Ya que el paralelismo y el no determinismo de *LPC* se modelan con un intercalamiento de instrucciones, no es posible emplear la composición funcional.

---

[2] $\underline{front}$ es el operador sobre secuencias definido por: $\underline{front}([\ ])$ = [ ] y $\underline{front}(\ L \frown [x]\ )$ = L.

En las ecuaciones subsecuentes, sólo se muestran las componentes del estado inicial que resulten afectadas por la ejecución de la instrucción que se esté definiendo; las componentes que no se exhiben quedan inalteradas.

$$\text{Ejecuta}(e,k,\ [\textbf{avanza}] \triangleleft T\ ) = \begin{cases} err \text{ si } (e.DeclM,e.Activ[k].r)=\underline{\text{Avanza}}(e.DeclM,e.Activ[k].r) \\ e' \text{ de otro modo,} \end{cases}$$

donde siendo $(DeclM',r') = \underline{\text{Avanza}}(e.DeclM,e.Activ[k].r)$,

$e'.DeclM \leftarrow DeclM'$, $e'.Activ[k].r \leftarrow r'$ y $\underline{\text{last}}(e'.Activ[k].PiT).t \leftarrow T$.

Esta ecuación establece que un robot sólo puede avanzar si está libre el cruce que tiene frente a sí. En ese caso, su componente posición así como la declaración de mundos se ven modificadas de acuerdo a la regla establecida por Avanza (R.30), definida en la sección 5.1. En caso contrario, el estado resultante es *err* y se ha definido un error.

Ejecuta$(e,k,$ [**gira** id] $\triangleleft T\ ) = e'$ donde siendo:

$Par = \underline{\text{last}}(e.Activ[k].PiA).Par$, $DirRel = Par($ Etiqueta(Id) $).Val$ y

$r' =$ Gira$(\ e.DeclM, e.Activ[k].r, DirRel\ )$,

$e'.Activ[k].r \leftarrow r'$ y $\underline{\text{last}}(e'.Activ[k].PiT).t \leftarrow T$.

Ejecuta$(e,k,$ [**gira** DireccionRelativa] $\triangleleft T\ ) = e'$ donde siendo:

$r' =$ Gira$(\ e.DeclM, e.Activ[k].r,$ DirRel(DireccionRelativa) $)$,

$e'.Activ[k].r \leftarrow r'$ y $\underline{\text{last}}(e'.Activ[k].PiT).t \leftarrow T$.

En ambas ecuaciones, Gira es la función definida por R.31, Etiqueta y DirRel son los atributos definidos en la sección 5.2.

$$\text{Ejecuta}(e,k,\ [\textbf{recoge}] \triangleleft T\ ) = \begin{cases} err \text{ si } (e.DeclM,e.Activ[k].r)=\underline{\text{Recoge}}(e.DeclM,e.Activ[k].r) \\ e' \text{ de otro modo,} \end{cases}$$

donde siendo $(DeclM';r') = \underline{\text{Recoge}}(e.DeclM,e.Activ[k].r)$,

$e'.DeclM \leftarrow DeclM'$, $e'.Activ[k].r \leftarrow r'$ y $\underline{\text{last}}(e'.Activ[k].PiT).t \leftarrow T$.

Se recuerda que la definición de Recoge (R.32) infiere que $\underline{\text{Recoge}}(DeclC,r)$ es igual a $(DeclC,r)$ si, y sólo si, no existe trompo alguno que recoger. Por lo tanto, el significado de ejecutar la instrucción **recoge** en esta situación queda definido como un error. En forma dual, el significado de ordenar la instrucción **deja** a un robot que no lleva trompos se define como un error.

$$\underline{\text{Ejecuta}}(e,k,\ [\text{deja}]\ \triangle\ T\ ) = \begin{cases} err \text{ si } (e.DeclM,e.Activ[k].r)=\underline{\text{Deja}}(e.DeclM,e.Activ[k].r) \\ e' \text{ de otro modo,} \end{cases}$$

donde siendo $(DeclM';r') = \underline{\text{Deja}}(e.DeclM,e.Activ[k].r)$,

$e'.DeclM$ <- $DeclM'$, $e'.Activ[k].r$ <- r' y $\underline{\text{last}}(e'.Activ[k].PiT).t$ <- T.

El significado de ejecutar $Id^1!Id^2$ e $Id^1?Id^2$ depende de si Id^2 denota el identificador de un canal ó el de un buzón. En el primer caso, se emplean las funciones Envía (P.30) y Recibe (P.31); en el segundo, se utilizan las funciones Solicita (P.32) y Atiende (P.33).

$$\underline{\text{Ejecuta}}(e,k,\ [Id^1!Id^2]\ \triangle\ T\ ) = \begin{cases} err \text{ si } (e.DeclC,e.DeclB,Activ'')=(DeclC',DeclB',Activ') \\ (DeclC',DeclB',e.DeclR,e.DeclM,e.DeclT,Activ''), \end{cases}$$

donde siendo $IdRe = \underline{\text{last}}(e.Activ[k].A).Ejec$, $IdCom = \text{Etiqueta}(Id^2)$,

$Activ''$ una secuencia idéntica a $e.Activ$ excepto en la tarea actual del k-ésimo proceso, definida por: $\underline{\text{last}}(Activ''[k].PiT).t$ <- T,

☞ si $IdCom \in \underline{\text{dom}}\ e.DeclC$:

$(DeclC',Activ') = \underline{\text{Envía}}(e.DeclC,e.DeclR,Activ'',IdCom,IdRe,IdRr)$ y

$DeclB' = e.DeclB$ con:

$$IdRr = \begin{cases} \text{Etiqueta}(Id^1) & \text{si Etiqueta}(Id^1) \in \underline{\text{dom}}\ \underline{\text{last}}(e.Activ[k].A).Rl \\ \underline{\text{last}}(e.Activ[k].A).Par(\text{Etiqueta}(Id^1)).Val & \text{de otro modo,} \end{cases}$$

☞ de otro modo :

$(DeclB',Activ') = \underline{\text{Solicita}}(e.DeclB,e.DeclR,e.DeclT,Activ'',IdCom,IdRe,IdTr)$ y

$DeclC' = e.DeclC$ con:

$$IdTr = \begin{cases} \text{Etiqueta}(Id^1) & \text{si Etiqueta}(Id^1) \in \underline{\text{dom}}\ e.DeclT \\ \underline{\text{last}}(e.Activ[k].A).Par(\text{Etiqueta}(Id^1)).Val & \text{de otro modo,} \end{cases}$$

Y en forma similar,

$$\underline{\text{Ejecuta}}(e,k,\ [Id^1?Id^2]\ \triangle\ T\ ) = \begin{cases} err \text{ si } (e.DeclC,e.DeclB,Activ'')=(DeclC',DeclB',Activ') \\ (DeclC',DeclB',e.DeclR,e.DeclM,e.DeclT,Activ'') \end{cases}$$

donde siendo $IdRr = \underline{\text{last}}(e.Activ[k].A).Ejec$, $IdCom = \text{Etiqueta}(Id^2)$,

$Activ''$ una secuencia idéntica a $e.Activ$ excepto en la tarea actual del k-ésimo proceso, definida por: $\underline{\text{last}}(Activ''[k].PiT).t$ <- T,

☞ si *IdCom* ∈ dom *e.DeclC* :

$$(DeclC', Activ') = \text{Recibe}(e.DeclC, e.DeclR, Activ'', IdCom, IdRe, IdRr) \text{ y}$$

$$DeclB' = e.DeclB \text{ con:}$$

$$IdRe = \begin{cases} \text{Etiqueta}(Id^1) & \text{si Etiqueta}(Id^1) \in \text{dom last}(e.Activ[k].A).Rl \\ \text{last}(e.Activ[k].A).Par(\text{Etiqueta}(Id^1)).Val & \text{de otro modo, } \curvearrowleft \end{cases}$$

☞ de otro modo :

$$(DeclB', Activ') = \text{Atiende}(e.DeclB, e.DeclT, e.DeclR, Activ'', IdCom, IdTe, IdRr) \text{ y}$$

$$DeclC' = e.DeclC \text{ con:}$$

$$IdTe = \begin{cases} \text{Etiqueta}(Id^1) & \text{si Etiqueta}(Id^1) \in \text{dom } e.DeclT \\ \text{last}(e.Activ[k].A).Par(\text{Etiqueta}(Id^1)).Val & \text{de otro modo, } \curvearrowleft \end{cases}$$

Las instrucciones que pertenecen al conjunto Iteracion solamente modifican la tarea actual del robot que las ejecute.

Ejecuta(*e,k,* [ciclo INST /{ INST }/ fin] ▲ *T*) = *e'* donde

last(*e'.Activ[k].PiT).t* <- [INST] /{ ▲ [INST] }/ ▲ [ciclo INST /{ INST }/ fin] ▲ *T*.

En forma similar, la instrucción rompe sólo altera la tarea actual del robot que la ejecuta.

Ejecuta(*e,k,* [rompe] ▲ *T*) = *e'* donde siendo

PART = {*(C,Q)* ∈ *TAREA*×*TAREA* | *T* = *C* ▲ [ciclo INST /{ INST }/ fin] ▲ *Q* },

existe[1] *(X,Z)* ∈ *PART* tal que *#X* < *#Y* para toda *(Y,Q)* ∈ *PART*, y

last(*e'.Activ[k].PiT).t* <- *Z*.

Antes de escribir las ecuaciones semánticas de las instrucciones del conjunto Alternativa, se definen dos funciones auxiliares : EvalBoo y MarcaGuardia. La primera es una función que calcula el valor booleano asociado a cada forma sintáctica del conjunto Guardia. La segunda es una función que sirve para construir una secuencia que se usa al modelar el no determinismo.

S.4 EvalBoo : *ESTADOS'* × *N* × Guardia ⟹ *BOOLEANO*.

S.5 *ALT* = seq *TAREA*.

---

[1] La semántica de las instrucciones del conjunto Iteracion y la condición impuesta a la utilización de la instrucción rompe con el atributo EnCiclo garantizan la existencia de las secuencias *X* y *Z*.

S.6 MarcaGuardia : $ESTADOS' \times N \times$ (ComandoCustodiado $\cup$ Salida) $\longmapsto$ ALT

Se comienza por definir a la función EvalBoo[4].

EvalBoo(*e,k,* TerminoBooleano^1 /{ o TerminoBooleano^2 }/) =

EvalBoo(*e,k,* TerminoBooleano^1) /{ ó EvalBoo(*e,k,* TerminoBooleano^2) }/.

EvalBoo(*e,k,* FactorBooleano^1 /{ y FactorBooleano^2 }/) =

EvalBoo(*e,k,* FactorBooleano^1) /{ & EvalBoo(*e,k,* FactorBooleano^2) }/.

EvalBoo(*e,k,* no FactorBooleano) = ¬ EvalBoo(*e,k,* FactorBooleano).

EvalBoo(*e,k,* (FactorBooleano)) = EvalBoo(*e,k,* FactorBooleano).

$$\text{EvalBoo}(e,k,\textbf{remite } Id^1 \textbf{ en } Id^2) = \begin{cases} \text{Correo?}(e.DeclC, IdCom, IdRe, IdRr) & \text{si } IdCom \in e.DeclC \\ \text{Peticion?}(e.DeclB, IdCom, IdTe, IdTr) & \text{de otro modo,} \end{cases}$$

donde $IdCom$ = Etiqueta(Id^3) y

☞ si $IdCom \in$ dom $e.DeclC$:

$IdRr$ = last($e.Activ[k].A$).$Ejec$ e

$$IdRe = \begin{cases} \text{Etiqueta}(Id^1) & \text{si Etiqueta}(Id^1) \in \text{dom last}(e.Activ[k].A).Rl \\ \text{last}(e.Activ[k].A).Par(\text{Etiqueta}(Id^1)).Val & \text{de otro modo, } \Omega \end{cases}$$

☞ de otro modo :

$IdTr$ = last($e.Activ[k].PiT$).IdT e

$$IdTe = \begin{cases} \text{Etiqueta}(Id^1) & \text{si Etiqueta}(Id^1) \in \text{dom } e.DeclT \\ \text{last}(e.Activ[k].A).Par(\text{Etiqueta}(Id^1)).Val & \text{de otro modo, } \Omega. \end{cases}$$

EvalBoo(*e,k,* **cargado**) = Cargado($e.Activ[k].r$) (R.22).

EvalBoo(*e,k,* **cierto**) = v.

[4] Se recuerda que los operadores lógicos de conjunción, disyunción y negación se denotan con los siguientes símbolos: &, ó y ¬ respectivamente.

EvalBoo(*e,k,* Predicado Id) = E(Predicado)(*e.DeclM,e.Activ[k].r,Val*),

donde *Val* = last(*e.Activ[k].A*).*Par*(Etiqueta(Id)).*Val*,

E = { **norte** -> Norte?, **muro** -> Muro?, **robot** -> Robot?, **trompo** -> Trompo? } y

los predicados siendo los correspondientes a las definiciones R.33 a R.36.

EvalBoo(*e,k,* Predicado DireccionRelativa) = E(Predicado)(*e.DeclM,e.Activ[k].r,DirRel*),

donde *DirRel* = DirRel(DireccionRelativa),

DirRel es el atributo definido en la sección 5.2 y

E es la función recién definida.

A continuación, se dan las reglas de evaluación de MarcaGuardia.

MarcaGuardia(*e,k,* Guardia -> INST > /{ ; INST }/) = *Alt* con,

$$Alt = \begin{cases} [\,] & \text{si } \neg \text{ EvalBoo}(e,k,\text{Guardia}), \\ [\ [\ \text{INST}\ ]\ /\{\ \blacktriangle\ [\ \text{INST}\ ]\ \}/\ ] & \text{en caso contrario.} \end{cases}$$

MarcaGuardia(*e,k,* **ninguna** -> INST > /{ ; INST }/) = [[INST] /{ ▲ [INST] }/]

Estas dos funciones permiten expresar con cierta sencillez la semántica de las instrucciones del conjunto Alternativa.

Ejecuta(*e,k,* [**escoge** ComandoCustodiado^1 /{ | ComandoCustodiado^2 }/ // Salida // **fin**] ▲ *T*)= *e'*

donde siendo :

Alt^1 = MarcaGuardia(*e,k,*ComandoCustodiado^1) /{ ▲ MarcaGuardia(*e,k,*ComandoCustodiado^2) }/,

$$//Alt^2 = \begin{cases} [\,] & \text{si } \#Alt^1 > 0\ //, \\ \text{MarcaGuardia}(e,k,\text{Salida}) & \text{de otro modo } // \text{ y} \end{cases}$$

$Alt = Alt^1\ //\ \blacktriangle\ Alt^2\ //$,

$$e' = \begin{cases} err & \text{si } Alt = [\,], \text{ (ninguna guardia se cumplió y no había salida)} \\ \text{Ejecuta}(e,k,Alt[\ \text{Aleatorio}(\#Alt)\ ]\ \blacktriangle\ T\ ) & \text{de otro modo.} \end{cases}$$

S.7 Aleatorio (: N ⟹ N) es una función que hace corresponder, a cada n de N, un entero i ∈ { 1, ..., n } que se determina aleatoriamente en cada una de sus aplicaciones. Se define Aleatorio con la siguiente propiedad:

Para toda n,i ∈ N | 1 ≤ i ≤ n, *Probabilidad(* Aleatorio(n) = i) = 1/n.

La semántica de las instrucciones que pertenecen al conjunto LlamadaDeTarea simplemente modifican la pila de tareas y la pila de ambientes del proceso cuyo robot las ejecute. Se usan los atributos ParFor, DeclT, RGlob, RLoc, EnPrg, Madre y Par de la sección 5.2 para expresar el paso de parámetros.

Ejecuta(e,k, [yo := Id // (ListaDeParametrosActuales) //] ▲ T) = e' donde siendo

$$IdT = \begin{cases} \text{Etiqueta(Id)} & \text{si Etiqueta(Id)} \in \underline{dom}\ e.DeclT \\ \underline{last}(e.Activ[k].A).Par(\text{Etiqueta(Id)}).Val & \text{de otro modo,} \end{cases}$$

donde: // ParFor(ListaDeParametrosActuales) <- $e.DeclT(IdT).Pf$ //,

// DeclT(ListaDeParametrosActuales) <- $e.DeclT$ //,

// RGlob(ListaDeParametrosActuales) <- $\emptyset$ //,

// RLoc(ListaDeParametrosActuales) <- $\underline{last}(e.Activ[k].A).Rl$ //,

// EnPrg(ListaDeParametrosActuales) <- Σ //,

// Madre(ListaDeParametrosActuales) <- $\underline{last}(e.Activ[k].A)$ //,

Ejec = $\underline{last}(e.Activ[k].A).Ejec$, RobotsLocales = $\underline{Renombra}(e.DeclT(IdT).Rl)$[3],

Ambiente = (Ejec, RobotsLocales, $\emptyset$ // ∪ Par(ListaDeParametrosActuales) //),

IdT = $\underline{last}(e.Activ[k].PiT).IdT$ y T' = $e.DeclT(IdT).t$,

☞ $e'.Activ[k].PiT$ <- $\underline{from}(e.Activ[k].PiT)$ ▲ [IdT, T] ▲ [IdT, T'] y

$e'.Activ[k].A$ <- $e.Activ[k].A$ ▲ Ambiente. ☜

El k-ésimo proceso queda con la tarea que venía desempeñando descabezada y sobrepuesta por la nueva tarea y su identificador. El último elemento de la pila de ambientes es el que genera el cambio de tarea. En forma similar, y completando la definición del significado asociado a las tareas por el lenguaje de *LPC*, se exhibe la semántica de las instrucciones cuyas formas sintácticas pertenecen al conjunto CreacionDeProceso.

[3] Renombra es una función que renombra a cada uno de los robots incluidos en la declaración de robots que toma por argumento de manera que sus nuevos nombres sean únicos.

$\underline{\text{Ejecuta}}(e,k,$ [Id^1 := Id^2 // (ListaDeParametrosActuales) //] ▲ T) = e' donde siendo

$$IdR = \begin{cases} \text{Etiqueta}(\text{Id}^1) & \text{si Etiqueta}(\text{Id}^1) \in \underline{\text{dom}}\ \underline{\text{last}}(e.Activ[k].A).Rl \\ \underline{\text{last}}(e.Activ[k].A).Par(\text{Etiqueta}(\text{Id}^1)).Val & \text{de otro modo,} \end{cases}$$

$$r = \begin{cases} \underline{\text{last}}(e.Activ[k].A).Rl(IdR) & \text{si } IdR \in \underline{\text{dom}}\ \underline{\text{last}}(e.Activ[k].A).Rl \\ e.DeclR(IdR) & \text{de otro modo,} \end{cases}$$

$$IdT = \begin{cases} \text{Etiqueta}(\text{Id}^2) & \text{si Etiqueta}(\text{Id}^2) \in \underline{\text{dom}}\ e.DeclT \\ \underline{\text{last}}(e.Activ[k].A).Par(\text{Etiqueta}(\text{Id}^2)).Val & \text{de otro modo,} \end{cases}$$

donde: // ParFor(ListaDeParametrosActuales) <- $e.DeclT(IdT).Pf$ //,

// DeclT(ListaDeParametrosActuales) <- $e.DeclT$ //,

// RGlob(ListaDeParametrosActuales) <- $\emptyset$ //,

// RLoc(ListaDeParametrosActuales) <- $\underline{\text{last}}(e.Activ[k].A).Rl$ //,

// EnPrg(ListaDeParametrosActuales) <- r //,

// Madre(ListaDeParametrosActuales) <- $\underline{\text{last}}(e.Activ[k].A)$ //,

RobotsLocales = $\underline{\text{Renombra}}(e.DeclT(IdT).Rl)$,

Ambiente = $(IdR,$ RobotsLocales, $\emptyset$ // $\cup$ Par(ListaDeParametrosActuales) //),

IdT' = $\underline{\text{last}}(e.Activ[k].PiT).IdT$, T' = $e.DeclT(IdT).t$,

$Activ''$ es idéntico a $e.Activ$ excepto que $\underline{\text{last}}(Activ''[k].PiT).t$ <- T y

$(DeclR',DeclM',Activ') = \underline{\text{CreaProceso}}(e.DeclR,e.DeclM,e.DeclT,IdT,\text{Ambiente},Activ'')$,

☞

$$e' = \begin{cases} err & \text{si } Activ'' = Activ', \text{ (el proceso no pudo ser creado)} \\ (e.DeclC,e.DeclB,DeclR',DeclM',e.DeclT,Activ') & \text{de otro modo.} \end{cases}$$ ☜

Siendo $\underline{\text{CreaProceso}}$ la función definida en la sección 5.1 (P.28). Sólo resta establecer el significado de un programa completo.

6.2 Un programa

Como se ha mencionado al principio de este capítulo, en *LPC* se considera que una ejecución paralela de procesos concurrentes equivale lógicamente a una ejecución secuencial, aleatoriamente intercalada, de los procesos involucrados. La función $\underline{\text{Ejecuta}}$ determina el grano de la MV_{LPC}.

Un programa en *LPC* comienza en un estado inicial y, aplicando la función Ejecuta, aleatoriamente va cambiando los estados de la MV_{LPC} hasta que, eventualmente, el programa termina al alcanzar el estado *err* o al quedar vacía la secuencia de procesos.

Se podría pensar en asignar el estado final como significado del programa. Sin embargo, como se apuntó en el segundo capítulo, los programas concurrentes suelen diseñarse para que no terminen. Es necesario entonces preguntar ¿cúal es el significado de un programa que no termina? La respuesta no es fácil si se piensa en un estado final.

En un segundo intento por establecer el significado de un programa, se define el rastro de un programa como la secuencia de los estados por los cuales va pasando la MV_{LPC} al ejecutarlo. Cada ejecución del programa generaría un rastro, posiblemente diferente, cuya representación gráfica es la que se observaría en la pantalla, en forma de animación. Parece razonable definir la semántica de un programa como el conjunto de los rastros que genera. Un programa que no terminara, produciría un conjunto de rastros infinitos. Esta semántica presenta dos aspectos que contradicen un tanto la intuición. La primera objeción se refiere al caracter infinito de los rastros. Intuitivamente, los rastros de un programa diseñado para no terminar aparecen más bien como ciclos, tal vez desordenados, que como secuencias infinitas. La segunda objeción cuestiona el concepto mismo de rastro: no parece importar mucho cómo se desplacen los robots al modificar sus mundos, ni en que orden (total) lo hagan; a priori, tienen importancia los estados alcanzados por la máquina. Además, al hablar de secuencias de estados, se introduce una dependencia temporal arbitraria entre los estados; esta cronología se disipa al considerar todos los posibles rastros del programa. Si con la programación concurrente se intenta suavizar la relación de orden prevaleciente entre las instrucciones de los programas, ¿porqué reforzarlas en el significado del lenguaje?

En vez de secuenciar estados en un rastro, piénsese en acumular los estados alcanzados por la MV_{LPC}. Un programa significa entonces la colección *desordenada* de estados alcanzados por la MV_{LPC} al ejecutarlo. El orden entre los estados queda determinado, parcialmente, por la función Ejecuta. Un programa diseñado para no terminar adquiere entonces el sentido intuitivo del conjunto de estados sobre el cual permanece estacionario. Un programa que no termina por error, tiene asignado el conjunto, posiblemente infinito, de los estados visitados por la MV_{LPC} en su perdición. Formalizando estas nociones, se obtienen las siguientes ecuaciones:

S.8 $RASTRO = \mathcal{P}ESTADOS.$[1]

S.9 Simula: $ESTADOS \longmapsto ESTADOS$

$e \rightarrow e'$

$err \rightarrow err$

donde, siendo $k =$ Aleatorio(#e.Activ) si e.Activ $\neq [\ ]$,

$$e' = \begin{cases} \text{Ejecuta}(e, k, \text{last}(e.Activ[k].PiT).t) & \text{si } e.Activ \neq [\ ], \\ e, \text{ de otro modo.} \end{cases}$$

Utilizando los atributos sintetizados por las formas sintácticas del conjunto Programa, definidos en la sección 5.2, se construye por inducción el conjunto S, perteneciente a *RASTRO*, de los estados alcanzados por la MV_{LPC} al ejecutar un programa P de la siguiente manera:

$S_0 = \{\ e_{ini}\ \}$ con $e_{ini} =$ (DeclC(P), DeclB(P), DeclR(P), DeclM(P), DeclT(P), Activ(P)),

$S_{k+1} = S_k \cup \{\ \text{Simula}(e) \mid e \in S_k\ \}$ y

$S = \cup \{\ S_k \mid k \in N\ \}$.

S es el menor subconjunto de *ESTADOS* que incluye a e_{ini} y está cerrado bajo Simula. Se define entonces la semántica de cualquier programa sintácticamente correcto asociándole precisamente ese subconjunto de estados; formalmente:

S.10 Significado: $Programa \longmapsto RASTRO$

$P \rightarrow S,$

con $e_{ini} =$ (DeclC(P), DeclB(P), DeclR(P), DeclM(P), DeclT(P), Activ(P)),

$S_0 = \{\ e_{ini}\ \}$,

$S_{k+1} = S_k \cup \{\ \text{Simula}(e) \mid e \in S_k\ \}$ y

$S = \cup \{\ S_k \mid k \in N\ \}$.

Tal vez quepa notar que esta semántica contrasta con la que tradicionalmente se define en otros lenguajes. En general, y hablando informalmente, se define como significado de un programa, cuyos datos de entrada son *Entradas* y los de salida *Salidas*, aquélla función parcial matemática que relaciona todas las posibles *Entradas* a las *Salidas* de la misma manera que lo ha-

[1] Donde $\mathcal{P}$ ESTADOS denota al conjunto potencia de los estados, o sea, el conjunto cuyos elementos son subconjuntos de ESTADOS.

ría el programa. En *LPC* se carece de entradas y salidas explícitas de datos. A diferencia de *CSP* y de *CCS*, el observador no participa en el sistema; se observa el sistema sin interferirlo. Un estado inicial contiene toda la información del programa y el efecto de ejecutar un programa es la metamorfosis de su estado (información) inicial en un conjunto de estados que guardan entre sí cierta relación. Se podría hablar de conjuntos de estados estables si la información contenida ya no cambia más de forma, esto es, ninguna ejecución del programa aporta algún estado que no esté ya presente; y de conjuntos inestables cuando la información contenida en el sistema se metamorfosee crónicamente, esto es, que el conjunto no se pueda cerrar bajo la función Simula. La relación prevaleciente entre los estados de un conjunto estable se antoja ser un orden parcial completo. Este resultado podría ser muy interesante pero rebasa los alcances del presente trabajo, por lo que únicamente se le señala como una prometedora dirección en la cual proseguir esta investigación.

El esfuerzo realizado para definir, y leer (!), la semántica formal del lenguaje de *LPC*, se laurea con las ventajas que se apuntan en el capítulo octavo. Con esta sección, se ha completado la presentación del lenguaje de *LPC*. Se emprende la descripción de cómo se realizó la actual versión de este sistema en el siguiente capítulo.

7. Realización de LPC

En este capítulo, se ofrece un panorama de la realización de *LPC* que se llevó a cabo usando el lenguaje *Modula-2* [Wirth85]. Se presenta la estructura general del sistema y se proporcionan los listados de los principales módulos de definición.

7.1 Estructura general

La implementación de *LPC* se compone de tres subsistemas: el editor de mundos, el compilador y el intérprete. Se accede a cada subsistema por medio de menús con ayuda integrada, lo cual elimina prácticamente la necesidad de un manual de usuario. En la figura 1, se muestra el diagrama de las dependencias entre los principales módulos del sistema. Se puede inferir de la figura que un módulo, A, consume de otro, B, (o sea que A importa artículos de B) cuando la "cajita" que representa a A se sitúa arriba de la de B y ambas están unidas por un trazo compuesto únicamente de secciones planas y/o descendientes[1]. Como se puede observar en la figura, la estructura de la realización es netamente jerárquica: los módulos de la parte inferior son, en su mayoría, implementaciones de tipos de datos abstractos primarios (Mundos, Robots, Colas Genéricas etc.) y, conforme se asciende, los módulos encierran más bien funcionalidades del sistema (Manejador de Mundos, Máquina Virtual etc).

Se suministran dos programas auxiliares con el sistema: un "desensamblador" de código *LPC* y el generador de las tablas del compilador. Ambos programas se describen en la sección 7.7.

[1] Se considera que estar más arriba es acercarse más al borde superior de la figura; en forma dual, descender es aproximarse al inferior.

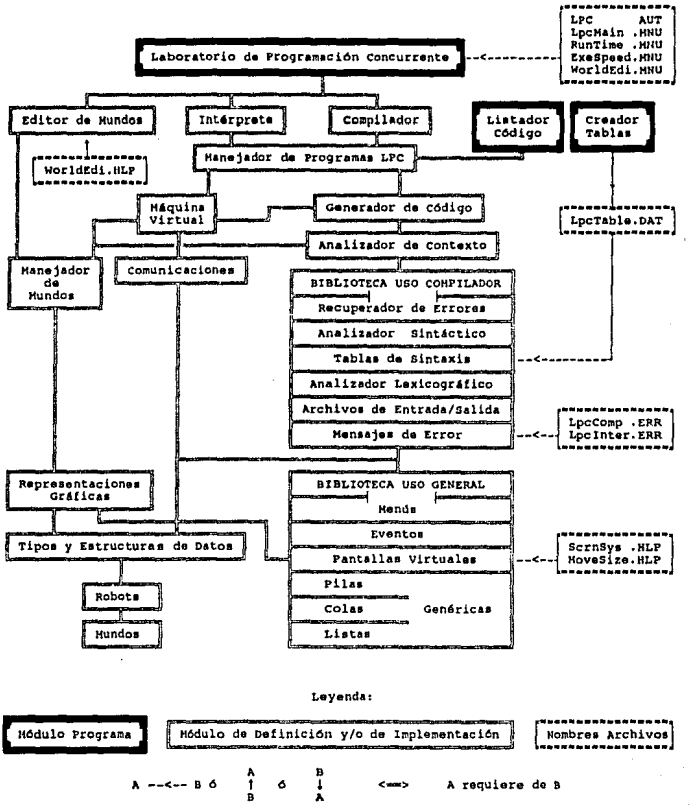


Figura 1 Diagrama de las dependencias entre módulos.

El sistema contiene 47 módulos de los cuales algunos forman parte de la biblioteca del compilador de *Modula-2* [LOGITECH86], otros pertenecen a la biblioteca diseñada y programada por el autor y los restantes son módulos de uso específico[2].

En las subsecuentes secciones se exponen tan solo algunas características de los módulos de mayor interés del sistema. Se adjuntan los listados de los módulos de definición seleccionados, los cuales incluyen comentarios[3].

7.2 Módulos primitivos

Los cuatro módulos que se presentan en esta sección se derivan directamente de los capítulos 3 y 5 de la tesis. Los mundos, los robots, los mecanismos de comunicación y muchos de los tipos y estructuras de datos definidos en los referidos capítulos conservan, al expresarlos en *Modula-2*, una forma casi idéntica. Obsérvense, para comprobarlo, los siguientes listados.

```
DEFINITION MODULE Worlds;
(*=========================================================================
  Global Description :  This module defines LPC's worlds. Definition
                        numbering has been taken from Thesis.
=========================================================================*)

 FROM SYSTEM IMPORT
 (* Type *)      BYTE;
 FROM VideoCard IMPORT
 (* Type *)      SCorner;
 FROM TextScreens IMPORT
 (* Type *)      Screen;

 EXPORT QUALIFIED
 (* Constant *)  MaxS, MaxA, MaxB,
 (* Type *)      STREET, AVENUE, CROSSING, BEEPER, STATE, WORLD,
                 CELL, INTERNAL, WorldDesc,
 (* Procedure *) PutState, RemoveState, IsState,
                 IncBeeper, DecBeeper, AnyBeeper;

 CONST
  MaxS      = 11;
  MaxA      = 15;
  MaxB      = 99;
```

[2] Para ser más preciso, los 47 se distribuyen de la siguiente manera: 17 módulos del sistema Logitech, 18 de la biblioteca del autor y 12 de uso específico.

[3] Estos comentarios aparecen en inglés por ser éste el idioma oficial del congreso en el cual se presentó una versión preliminar del sistema [Benveniste88].

```
TYPE
 STREET   = [1 .. MaxS];          (* M.28  STREET = N *)
 AVENUE   = [1 .. MaxA];          (* M.29  AVENUE = N *)
 CROSSING = RECORD                (* M.3   CROSSING = AVENUExSTREET *)
              a: AVENUE;
              s: STREET
            END;
 BEEPER   = BYTE;                 (* M.13 BEEPER = N *)
 STATE    = (free,wall,robot);    (* M.16 STATE = (Free,Wall,Robot) *)
 WORLD    = POINTER TO WorldDesc;
 CELL     = RECORD
              State : STATE;
              Beep  : BEEPER
            END;
 INTERNAL = ARRAY AVENUE, STREET OF CELL;
                                  (* M.17 WORLD = CROSSING ---> STATExBEEPER *)
 WorldDesc= RECORD
              DefPos  : SCorner;      (* Default position *)
              DefUp   : BOOLEAN;      (* of external view *)
              Width   : AVENUE;       (* World's          *)
              Length  : STREET;       (*           size   *)
              Habit   : CARDINAL;     (* Population       *)
              Ext     : Screen;       (* External view    *)
              Int     : INTERNAL      (* Internal view    *)
            END;


PROCEDURE PutState(s: STATE; VAR w: WORLD; c: CROSSING);
(*
                STATExWORLDxCROSSING   --->      WORLD
  if w(c)=(Free,b)     (s,w,c)         --->  w + { c --> (s,b) }
  otherwise            (s,w,c)         --->      w

        M.18 = PutState(robot)         M.21 = PutState(wall)
*)

PROCEDURE RemoveState(s: STATE; VAR w: WORLD; c: CROSSING);
(*
                STATExWORLDxCROSSING   --->      WORLD
  if w(c)=(s,b)        (s,w,c)         --->  w + { c --> (Free,b) }
  otherwise            (s,w,c)         --->      w

       M.19 = RemoveState(robot)      M.22 = RemoveState(wall)
*)

PROCEDURE IsState(s: STATE; w: WORLD; c: CROSSING) : BOOLEAN;
(*
               STATExWORLDxCROSSING   --->     BOOLEAN
                    (s,w,c)           --->  ( w(c)=(s,b) )

          M.20 = IsState(robot)      M.23 = IsState(wall).
*)

PROCEDURE IncBeeper(VAR w: WORLD; c: CROSSING; n: CARDINAL);
(*
  M.24 IncBeeper:  WORLDxCROSSINGxN   --->          WORLD
  if w(c).b+n <= MaxB   (w,c,n)       --->  w + { c --> (w(c).s,w(c).b+n) }
  otherwise             (w,c,n)       --->  w + { c --> (w(c).s,MaxB) }
*)

PROCEDURE DecBeeper(VAR w: WORLD; c: CROSSING; n: CARDINAL);
(*
  M.25 DecBeeper:  WORLDxCROSSINGxN ---->         WORLD
  if w(c).b > n         (w,c,n)     ---> w + { c --> (w(c).s,w(c).b-n) }
  otherwise             (w,c,n)     ---> w + { c --> (w(c).s,0) }
*)
```

```
PROCEDURE AnyBeeper(w: WORLD; c: CROSSING) : BOOLEAN;
(*
  M.26 AnyBeeper:  WORLDxCROSSING   --->      BOOLEAN
                   (w,c)            --->  ( w(c).b > 0 ).
*)

END Worlds.


DEFINITION MODULE Robots;
(*=========================================================================
  Global Description :  This module defines LPC's robots. Definition
                        numbering has been taken from Thesis.
=========================================================================*)

FROM Worlds IMPORT
(* Type *)      CROSSING, STATE, BEEPER, WORLD;
FROM TextScreens IMPORT
(* Type *)      Screen;

EXPORT QUALIFIED
(* Type *)      DIRECTION, RobotDesc, ROBOT, RELDIR,
(* Procedure *) Step, Spin, Pick, Drop,
                AskState, AskBeeper, AskNorth, AskLoaded;

TYPE
 DIRECTION = (north, east, south, west);  (* R.6 modified for efficiency *)
 ROBOT     = POINTER TO RobotDesc;
 RobotDesc = RECORD
               Wid      : POINTER TO WORLD;       (* World inhabited *)
               Pos      : CROSSING;               (* Occupied crossing *)
               Dir      : DIRECTION;              (* Direction faced *)
               Bag      : BEEPER;                 (* Beepers carried *)
               Rid      : ARRAY[0 .. 1] OF CHAR;  (* 2 digits Id *)
               Msg      : Screen                  (* In/Out display *)
             END;
                        (* R.8  ROBOT < WORLDxCROSSINGxDIRECTIONxBEEPER *)
 RELDIR    = (prow, starboard, poop, larboard);
                          (* R.9 modified for efficiency *)


PROCEDURE Step(VAR r: ROBOT);
(*
   R.30  Step: ROBOT ---> ROBOT
               r   --->  r'

      let m = r.Wid^,
   if NOT(IsRobot(m,r.Dir(r.Pos)) OR IsWall(m,r.Dir(r.Pos)))
     then
       r.Wid^ = PutRobot(RemoveRobot(m,r.Pos),r.Dir(r.Pos))
       r'     = (r.Wid,r.Dir(r.Pos),r.Dir,r.Bag)
     else
       r'     = r.
*)

PROCEDURE Spin(VAR r: ROBOT; rd: RELDIR);
(*
   R.31 Spin:  ROBOTxRELDIR ---> ROBOT
               (r,rd)       --->  r'

       r'     = (r.Wid,r.Pos,rd(r.Dir),r.Bag)
*)
```

```
PROCEDURE Pick(VAR r: ROBOT);
(*
   R.32 Pick: ROBOT ---> ROBOT
                r   --->  r'

      let m = r.Wid^,
   if AnyBeeper(m,r.Dir(r.Pos))
     then
       r.Wid^ = RemoveBeeper(m,r.Dir(r.Pos))
       r'     = (r.Wid,r.Pos,r.Dir,r.Bag+1)
     else
       r'     = r.
*)

PROCEDURE Drop(VAR r: ROBOT);
(*
   R.33 Drop: ROBOT ---> ROBOT
                r   --->  r'

      let m = r.Wid^,
   if  r.Bag > 0
     then
       r.Wid^ = PutBeeper(m,r.Dir(r.Pos))
       r'     = (r.Wid,r.Pos,r.Dir,r.Bag-1)
     else
       r'     = r.
*)

PROCEDURE AskState(s: STATE; r: ROBOT; rd: RELDIR): BOOLEAN;
(*
  AskState: STATExROBOTxRELDIR --->              BOOLEAN
              (s,r,rd)         --->   IsState(s,r.Wid^,rd(r.Dir)(r.Pos))

           R.35 AskState(wall)          R.36 AskState(robot).
*)

PROCEDURE AskBeeper(r: ROBOT; rd: RELDIR): BOOLEAN;
(*
  R.37 AskBeeper: ROBOTxRELDIR  --->           BOOLEAN
                    (r,rd)      --->  AnyBeeper(r.Wid^,rd(r.Dir)(r.Pos)).
*)

PROCEDURE AskNorth(r: ROBOT; rd: RELDIR): BOOLEAN;
(*
  R.34 AskNorth: ROBOTxRELDIR  --->        BOOLEAN
                   (r,rd)      ---> ( rd(r.Dir) = north ).
*)

PROCEDURE AskLoaded(r: ROBOT): BOOLEAN;
(*
  R.22 AskLoaded: ROBOT --->   BOOLEAN
                    r   ---> ( r.Bag > 0 ).
*)

END Robots.


DEFINITION MODULE Representations;
(*=========================================================================
  Global Description :  This module provides representations
                        for LPC's objects. It has an empty implementation.
=========================================================================*)

FROM Worlds IMPORT
(* Type *)       CROSSING, BEEPER, WORLD;
FROM Robots IMPORT
(* Type *)       DIRECTION, RELDIR, ROBOT;
```

```
FROM Queues IMPORT
(* Type *)        Queue;
FROM Stacks IMPORT
(* Type *)        Stack;
FROM TextScreens IMPORT
(* Type *)        Screen;

EXPORT QUALIFIED
(* Const *) MemEnd, MaxAlphaLgth, MaxAlphaNum,
            MaxEnvNum, MaxDeclNum, MaxLocEnvNum,
(* Type *)  AdrSpace,
            TableIndex, Alpha, NAMES, NamesRecord,
            DeclId, IdType, DeclEntry, DECLARATION, DeclRecord,
            EnvId, LocEnvId, EnvType, EnvEntry,
            ENVIRONMENT, LOCALenv, EnvRecord,
            RobCommId, LETTER, REQUEST, CHANNEL, MAILBOX,
            ChanExpected, MailExpected, ChannelDesc, MailboxDesc,
            SPEED, ProcessDesc, PROCESS;

CONST
 MemEnd       = 16*1024;  (* Maximum code size *)
 MaxAlphaLgth = 16;       (* Maximum identifier length *)
 MaxAlphaNum  = 500;      (* Maximum identifier's tag number *)
 MaxEnvNum    = 500;      (* Maximum environment item number *)
 MaxDeclNum   = 1000;     (* Maximum declaration number *)
 MaxLocEnvNum = 32;       (* Maximum local environment item number *)

TYPE
 AdrSpace     = [0 .. MemEnd-1];
 TableIndex   = [0 .. MaxAlphaNum-1];
 Alpha        = ARRAY[0 .. MaxAlphaLgth-1] OF CHAR;
 DeclId       = [0 .. MaxDeclNum];  (* 0 should always be the program Id *)
 RobCommId    = RECORD
                  Creator     : DeclId;
                  MyId        : DeclId;
                  RecurLevel : CARDINAL
                END;
 IdType       = (ChanId,MailId,RoboId,TaskId,WorlId,ParamId);
 EnvType      = (RobExe,RoboPar,RoboVarPar,RoboDecl,TaskPar,TaskVarPar,RelDPar);
 EnvId        = [0 .. MaxEnvNum-1];
 LocEnvId     = [1 .. MaxLocEnvNum];
 EnvEntry     = RECORD
                  CASE Kind : EnvType OF
                   RobExe,RoboPar,RoboVarPar,RoboDecl
                   : CommId   : RobCommId;
                     Assigned : BOOLEAN        (* valid iff RoboVarPar *)
                  | TaskPar, TaskVarPar
                   : TaskName : DeclId
                  | RelDPar
                   : Value    : RELDIR
                  END (*case*)
                END;
LOCALenv      = POINTER TO ARRAY LocEnvId OF EnvEntry;
LETTER        = RECORD
                  Sender,
                  Receiver : RobCommId
                END;
REQUEST       = RECORD
                  Client,
                  Server   : DeclId
                END;
SPEED         = [0 .. 10];                    (* relative speed of robots *)
PROCESS       = POINTER TO ProcessDesc;
```

```
ProcessDesc  = RECORD
                Agent        : ROBOT;
                Procid       : RobCommId;
                Level        : CARDINAL;   (* Activation level *)
                InstPerCycle : SPEED;
                IP           : AdrSpace;
                Stk          : Stack;
                Task         : DeclId;     (*WARNING:: order is important *)
                Static       : LOCALenv;   (* any change is lethal. *)
                Envstk       : Stack       (* Covers Task & Static *)
               END;
ChanExpected = RECORD
                Who  : PROCESS;
                From : RobCommId
               END;
MailExpected = RECORD
                Who  : PROCESS;
                From : DeclId
               END;
ChannelDesc  = RECORD
                Signal  : LETTER;
                Arrived : Queue;
                First   : ChanExpected;
                Blocked : Queue
               END;
MailboxDesc  = RECORD
                Signal  : REQUEST;
                Arrived : Queue;
                First   : MailExpected;
                Blocked : Queue
               END;
CHANNEL      = POINTER TO ChannelDesc;
MAILBOX      = POINTER TO MailboxDesc;
DeclEntry    = RECORD
                (*$A++)                    (* Compiler: turn word alignment on *)
                Tag : TableIndex;
                CASE Kind: IdType OF
                 ChanId
                  : Chan     : CHANNEL
                | MailId
                  : Mail     : MAILBOX
                | RoboId
                  : Wid      : DeclId;
                    Shared   : BOOLEAN;
                    Pos      : CROSSING;
                    Dir      : DIRECTION;
                    Bag      : BEEPER;
                    Assigned : BOOLEAN;
                    ParNum   : LocEnvId
                | TaskId
                  : EntryPt  : AdrSpace;
                    EnvNum   : LocEnvId;  (* at least 1  [executer's commId] *)
                    First    : EnvId;     (* valid iff EnvNum > 1 *)
                | WorlId
                  : World    : WORLD;
                    CopyNum  : CARDINAL   (* Number of copies *)
                | ParamId
                  : LocNum   : LocEnvId
                 END (*case*)
                (*$A**)                    (* Compiler: reset word alignment as it was *)
               END;
NAMES        = POINTER TO ARRAY TableIndex OF Alpha;
NamesRecord  = RECORD
                Size : [0 .. MaxAlphaNum];
                Table: NAMES
               END;
DECLARATION  = POINTER TO ARRAY DeclId OF DeclEntry;
```

```
  DeclRecord    = RECORD
                    Size : [0 .. MaxDeclNum];
                    Decl : DECLARATION
                  END;
  ENVIRONMENT   = POINTER TO ARRAY EnvId OF EnvEntry;
  EnvRecord     = RECORD
                    Size : [0 .. MaxEnvNum];
                    Env  : ENVIRONMENT
                  END;
END Representations.


DEFINITION MODULE Communications;
(*=============================================================================
  Global Description :  This module provides LPC's communication functions.
  ---------------------------------------------------------------------------

                                FirstEdit: 9/May/88
  ---------------------------------------------------------------------------

               Author : Marc Benveniste
               System : Logitech MODULA-2/86 V. 2.05
 ============================================================================*)

  FROM Representations IMPORT
  (* Type *)       PROCESS, RobCommId, DeclId,
                   LETTER, REQUEST, CHANNEL, MAILBOX;
  FROM VirtualMachine IMPORT
  (* Type *)       ASSIGN;

  EXPORT QUALIFIED
  (* Var *)        CommError,
  (* Procedure *) OpenChannel, OpenMailbox, CloseChannel, CloseMailbox,
                   Send, Receive, AnyLetter,
                   Ask, Answer, AnyRequest;

  VAR
   CommError : CARDINAL;

  PROCEDURE OpenChannel(): CHANNEL;
  (*
    Creates a new channel if possible.
    Out --> Either an empty channel or NIL
    SideEffects --> CommError :: 0 -> No errors
                                 QueueError (see Queues.DEF)
  *)

  PROCEDURE OpenMailbox(): MAILBOX;
  (*
    Creates a new mailbox if possible.
    Out --> Either an empty mailbox or NIL
    SideEffects --> CommError :: 0 -> No errors
                                 QueueError (see Queues.DEF)
  *)

  PROCEDURE CloseChannel(VAR c: CHANNEL);
  (*
    Closes the channel.
    In  --> c :: A well defined channel created with OpenChannel,
                 no checking is provided.
    SideEffects --> CommError :: 0 -> No errors
                                 QueueError (see Queues.DEF)
  *)
```

```
PROCEDURE CloseMailbox(VAR m: MAILBOX);
(*
  Closes the mailbox.
  In  --> m :: A well defined mailbox created with OpenMailbox,
              no checking is provided.
  SideEffects --> CommError :: 0 -> No errors
                               QueuError (see Queues.DEF)
*)

PROCEDURE Send(VAR l: LETTER; VAR c: CHANNEL; VAR a: ASSIGN);
(*
  Sends a letter l through channel c possibly awaking a blocked process
  to assignment a.
  In  --> l : A well defined letter (no checking provided),
          c : A well defined channel (no checking provided),
          a : A well defined assignment (no checking provided);
  Out --> c : The channel with l appended at its end if no process was
              waiting for it,
          a : A possibly new assignment if a process was waiting for l;
  SideEffects -> CommError :: 0 if no errors; QueuError otherwise.
*)

PROCEDURE Receive(VAR From: RobCommId; VAR c: CHANNEL; VAR p: PROCESS);
(*
  p tries to receive a letter from From in channel c. If that letter
  is present p will succeed; otherwise p will be blocked.
  In  --> From : A well defined robot communication id (no checking
                 provided),
          c    : A well defined channel (no checking provided),
          p    : A well defined process (no checking provided);
  Out --> c : The channel with letter removed if present, otherwise with
              p blocked waiting for From,
          p : NIL if blocked;
  SideEffects -> CommError :: 0 if no errors; QueuError otherwise.
*)

PROCEDURE AnyLetter(VAR l: LETTER; c: CHANNEL): BOOLEAN;
(*
  Finds out if there is any letter l on channel c.
  In  --> l : A well defined letter to look for (no checking provided),
          c : A well defined channel (no checking provided);
  Out --> TRUE  if Receive(l.Sender,c,p) would succeed;
          FALSE if p would be blocked;
  SideEffects -> CommError :: 0 if no errors; QueuError otherwise.
*)

PROCEDURE Ask(VAR r: REQUEST; VAR m: MAILBOX; VAR a: ASSIGN);
(*
  Sends a request r through mailbox m possibly awaking a blocked process
  to assignment a.
  In  --> r : A well defined request (no checking provided),
          m : A well defined mailbox (no checking provided),
          a : A well defined assignment (no checking provided);
  Out --> m : The mailbox with r appended at its end if no process was
              waiting for it,
          a : A possibly new assignment if a process was waiting for r;
  SideEffects -> CommError :: 0 if no errors; QueuError otherwise.
*)
```

```
PROCEDURE Answer(VAR From: DeclId; VAR m: MAILBOX; VAR p: PROCESS);
(*
  p tries to receive a request from From in mailbox m. If that request
  is present p will succeed; otherwise p will be blocked.
  In  --> From : A well defined Task id (no checking provided),
          m    : A well defined mailbox (no checking provided),
          p    : A well defined process (no checking provided);
  Out --> m : The mailbox with request removed if present, otherwise with
              p blocked waiting for From,
          p : NIL if blocked;
  SideEffects -> CommError :: 0 if no errors; QueuError otherwise.
*)

PROCEDURE AnyRequest(VAR r: REQUEST; m: MAILBOX): BOOLEAN;
(*
  Finds out if there is any request r on mailbox m.
  In  --> r : A well defined request to look for (no checking provided),
          m : A well defined mailbox (no checking provided);
  Out --> TRUE  if Answer(r.client,m,p) would succeed;
          FALSE if p would be blocked;
  SideEffects -> CommError :: 0 if no errors; QueuError otherwise.
*)
END Communications.
```

7.3 El compilador

Dada la relativa sencillez del lenguaje de *LPC*, se optó por un compilador de un solo paso. Se escogió el método de descenso recursivo para el analizador sintáctico del compilador. Se decidió programarlo usando tablas por tratarse de un lenguaje totalmente nuevo y, por ello, sujeto a modificaciones. Si bien esta implementación no es la más eficiente, ofrece varias ventajas:

- *flexibilidad*, la gramática reconocida es fácilmente modificable;
- *control*, el programador controla la recursión del analizador por medio de pilas y de tablas;
- *reusabilidad*, permite la construcción de un conjunto de analizadores sintácticos (uno por tabla) en lugar de sólo uno.

Las distintas etapas del proceso de compilación se reflejan claramente en los módulos y, como lo indica la figura 1, todo el análisis sintáctico se diseñó e implementó como una biblioteca que podrá utilizarse para desarrollar compiladores ulteriores. El análisis semántico (de contexto) y la generación del código intermedio son específicos del lenguaje de *LPC*.

7.3.1 Módulos reusables

A continuación se exhibe la biblioteca que realiza el análisis sintáctico.

```
DEFINITION MODULE ErrorMessages;
(*=============================================================================
  Global Description :  This module provides the error messages used by
                        Source.MarkError and Interpreter.MarkError.
 =============================================================================*)

 EXPORT QUALIFIED
 (* Const *)       MsgMaxLength, MaxErrorCode,
 (* Type *)        ErrorRange, MsgType, MsgTable,
 (* Var *)         ErrorMsgs,
 (* Procedure *)   LoadCompilerMsgs, LoadInterpreterMsgs;

 CONST
  MaxErrorCode      = 75;
  MsgMaxLength      = 56;
 TYPE
  ErrorRange  = [0 .. MaxErrorCode-1];
  MsgType     = ARRAY[0 .. MsgMaxLength-1] OF CHAR;
  MsgTable    = ARRAY ErrorRange OF MsgType;
 VAR
  ErrorMsgs      : MsgTable;

 PROCEDURE LoadCompilerMsgs;
 (*
   Loads Compiler error messages looking for them in the file
   LpcComp.Err on the current path and drive.
   Error messages should be ended by a period and filled to MsgMaxLength
   with blanks. Messages should not contain CR LF sequences.
 *)
 PROCEDURE LoadInterpreterMsgs;
 (*
   Loads Compiler error messages looking for them in the file
   LpcInter.Err on the current path and drive.
   Error messages should be ended by a period and filled to MsgMaxLength
   with blanks. Messages should not contain CR LF sequences.
 *)
END ErrorMessages.


DEFINITION MODULE Source;
(*=============================================================================
 Global Description : This module provides the source reader & error lister.
 =============================================================================*)

    FROM ErrorMessages IMPORT
    (* Type *)        ErrorRange;

    EXPORT QUALIFIED
    (* Var *)         CurChar,
    (* Procedure *)   OpenIO, GetNextChar, MarkError, CloseIO;

  VAR
    CurChar : CHAR;          (* ReadOnly *)

  PROCEDURE OpenIO(FileName: ARRAY OF CHAR): BOOLEAN;
    (*
       Opens input to be converted to a CHAR stream. Returns TRUE
       if successfully opened; FALSE otherwise.
       Side Effect ---> CurChar <- ' '.
    *)
```

```
  PROCEDURE GetNextChar;
    (*
       Provides the next CHAR found in opened Input.
       Side Effect ---> CurChar <- Next input CHAR if any;
                                   ASCII.sub otherwise.
    *)

  PROCEDURE MarkError(Code: ErrorRange);
    (*
      Marks the code number at current input position and when calling
      CloseIO, it attaches the explaining message corresponding to code
      as described in ErrorMessages.ErrorMsg[Code].
            Input  ---> Code  : Error code.
    *)

  PROCEDURE CloseIO;
    (*
      Closes input and writes resulting error explanation messages if any.
    *)
END Source.


DEFINITION MODULE Scanner;
(*=========================================================================
  Global Description :  This module provides a scanner for LPC's language.
=========================================================================*)

 FROM Representations IMPORT
 (* Type *)       TableIndex, NamesRecord;

 EXPORT QUALIFIED
 (* Const *)      EndOfSource,
 (* Type *)       TokenSym,
 (* Var *)        IdTable, ScanError, CurToken,
 (* Procedure *)  LoadResWord, GetNextToken, UnloadResWord;

  CONST
   EndOfSource       = 99;
  TYPE
   TokenSym      = (Void,ProgramSym,ChannelSym,MailboxSym,LivesSym,SharesSym,
                    OnSym,WithSym,TaskSym,RelDirSym,IsSym,EndSym,AssignSym,
                    VarSym,MyselfSym,
                    ProwSym,StarboardSym,PoopSym,LarboardSym,
                    StepSym,SpinSym,PickSym,DropSym,SendSym,ReceiveSym,SkipSym,
                    SelectSym,GuardSepSym,ElseSym,GuardAssignSym,EndAtomSym,
                    LoopSym,BreakSym,
                    NotSym,AndSym,OrSym,
                    MailedSym,NorthSym,RobotSym,WallSym,BeeperSym,
                    LoadedSym,TrueSym,
                    IdSym,CardSym,
                    Comma,Semicolon,Colon,OpenPar,ClosePar,Period);
   TOKEN         = RECORD
                     CASE T: TokenSym OF
                      IdSym   : Tag   : TableIndex
                     | CardSym : Value : CARDINAL
                      ELSE
                     END
                   END;
  VAR     (* READ ONLY VARs *)
   IdTable     : NamesRecord;   (* Largest valid table *)
   ScanError   : CARDINAL;      (* Initialized to 1 *)
   CurToken    : TOKEN;         (* Valid iff ScanError = 0 *)
```

```
  PROCEDURE LoadResWord;
  (*
     MUST be called before the first call to GetNextToken.
     SideEffects ---> The reserved word table is initialized;
                      IdTable is initialized;
                      ScanError <- 0.
  *)

  PROCEDURE GetNextToken;
  (*
    Gets the next available token if any.
    SideEffects ---> CurToken  <- next token if ScanError = 0;
                     ScanError <- 0 if no error occured;
                                  1 if LoadResWord not called;
                                  2 if AlphaTable is full;
                                  EndOfSource if EOF reached.
                     Screen message written if needed
                     based on ErrorMessages.ErrorMsgs.
  *)
  PROCEDURE UnloadResWord;
  (*
     MUST be called to recover storage.
     SideEffects ---> The reserved word table is destroyed
                      and its storage is recovered;
                      ScanError <- 1.
  *)
END Scanner.


DEFINITION MODULE ParseTables;
(*=========================================================================
  Global Description :  This module provides the parse tables needed to
                        recognize a valid LPC program.
=========================================================================*)

    FROM ErrorMessages IMPORT
    (* Type *)       ErrorRange;
    FROM Scanner IMPORT
    (* Type *)       TokenSym;

    EXPORT QUALIFIED
    (* Const *)      MaxStateNum, MaxArcNum, AcceptArcNum,
    (* Type *)       ArcRange, StateRange, ExtStateRange,
                     ArcType, ArcTable, StartTable, StateErrTable,
    (* Var *)        Arcs, ArcBegin, CodePerState,
    (* Procedure *) LoadTables, UnloadTables;

  CONST
    MaxStateNum   = 80;
    MaxArcNum     = 134;
    AcceptArcNum  = 14;
  TYPE
    ArcRange      = [1 .. MaxArcNum+1];
    StateRange    = [1 .. MaxStateNum];
    ExtStateRange = [0 .. MaxStateNum];
    ArcType       = RECORD
                      Next : ExtStateRange;
                      CASE Term: BOOLEAN OF
                       TRUE : Trigger : TokenSym |
                       FALSE: SubNet  : StateRange
                      END
                    END;
    ArcTable      = ARRAY ArcRange OF ArcType;
    StartTable    = ARRAY ExtStateRange OF ArcRange;
    StateErrTable = ARRAY StateRange OF ErrorRange; (* Error msgs. index *)
```

```
  VAR
    Arcs          : POINTER TO ArcTable;
    ArcBegin      : POINTER TO StartTable;
    CodePerState  : POINTER TO StateErrTable;

  PROCEDURE LoadTables(VAR Done : BOOLEAN);
  (* Allocates storage for tables and loads them.
     Output --> Done :: True if tables successfully loaded;
                        False otherwise.
  *)

  PROCEDURE UnloadTables;
  (* Recovers storage allocated by LoadTables. *)
END ParseTables.


DEFINITION MODULE Parser;
(*=============================================================================
  Global Description :  This module provides a procedure to parse a given
                        language. The parsing algorithm relies on the table
                        structure described in ParseTables.DEF.
 =============================================================================*)

    FROM Scanner IMPORT
    (* Type *)        TokenSym;
    FROM ParseTables IMPORT
    (* Const *)       AcceptArcNum,
    (* Type *)        ArcRange, StateRange, ExtStateRange;

    EXPORT QUALIFIED
    (* Var *)         CurArc, CurState, NewToken,
    (* Procedure *)   InitParser, TermParser,
                      GetNextArc, Follows, Starts;

  VAR
    CurArc    : ArcRange;       (* Arc number triggered by Token *)
    CurState  : ExtStateRange;  (* Parser's state number *)
    NewToken  : BOOLEAN;        (* Tells if the parser needs a new token *)

  PROCEDURE InitParser(VAR Done : BOOLEAN);
  (*
      Out           ---> Done <- TRUE if successful ignition;
                               FALSE otherwise.
      Side Effects  ---> Calls Scanner.LoadResWord,
                         Calls ParseTables.LoadTables,
                         Calls ErrorMessages.LoadCompilerMsgs,
                         CurArc      <- undefined,
                         CurState    <- 1,
                         NewToken    <- TRUE.
  *)

  PROCEDURE TermParser;
  (*
      Side Effects ---> Calls ParseTables.UnloadTables,
                        Calls Scanner.UnloadResWord,
                        CurArc      <- undefined,
                        CurState    <- undefined,
                        NewToken    <- undefined.
  *)
```

```
    PROCEDURE GetNextArc(Token : TokenSym): CARDINAL;
    (*
       Input  ---> Token : Driving token.
       Output ---> 0   means No error occured.
                   1   means Syntax error has been found.
                   >1  means Internal failure.
       Side Effects ---> CurArc   <- Found arc if no error; untouched otherwise.
                         CurState <- Last valid state.
                         NewToken <- TRUE if new token needed; FALSE otherwise.
      NOTE: it uses implicitely ParseTables.Arcs & ParseTables.ArcBegin.
    >> WARNING ::
       UNPREDICTABLE behavior results WHEN the tables are not loaded. <<
    *)
    PROCEDURE Follows(Tk: TokenSym; VAR Pr: TokenSym): BOOLEAN;
    (*
       Input  ---> Tk : Driving token.
       Output ---> Pr : Triggering token if Follows() = TRUE;
                        undefined otherwise.
                   TRUE  if GetNextArc(Pr) = 0 and GetNextArc(Tk) = 0 when
                         executed in that order and in the current state
                         of the parser;
                   FALSE otherwise.
    NOTE: it uses implicitely ParseTables.Arcs, ParseTables.ArcBegin & CurState.
    >> WARNING ::
        NEVER CALL THIS PROCEDURE IF CurState = 0.
        UNPREDICTABLE behavior results WHEN the tables are not loaded. <<
    *)

    PROCEDURE Starts(Tk: TokenSym; In: ExtStateRange): BOOLEAN;
    (*
       Input  ---> Tk : Driving token.
                   In : State.
       Output ---> TRUE  if GetNextArc(Tk) = 0 until NewToken = TRUE when
                         executed in the parser state In;
                   FALSE otherwise.
    NOTE: it uses implicitely ParseTables.Arcs, ParseTables.ArcBegin & CurState;
       if the tables are not loaded it returns FALSE.
    >> WARNING ::
       UNPREDICTABLE behavior results WHEN the tables are not loaded. <<
    *)
  END Parser.


  DEFINITION MODULE Recoverer;
  (*=============================================================================
    Global Description :  This module provides an error recovery procedure for
                          the language specified in ParseTables.DEF. It uses
                          the parsing algorithm provided by Parser.DEF and the
                          scanner supplied by Scanner.GetNextToken.
  =============================================================================*)

    EXPORT QUALIFIED
    (* Procedure *)  Recovery;

   PROCEDURE Recovery;
   (* Uses all parser available information, the scanner and the source
      handler to provide both a local error recovery of 1 token lookahead and
      a global recovery that skips the current subnet. *)
  END Recoverer.
```

```
DEFINITION MODULE Compiler;
(*=============================================================================
  Global Description :  This module provides a compiler for the language
                        specified in ParseTables.DEF. It uses the parsing
                        algorithm provided by Parser.DEF and the scanner
                        supplied by Scanner.GetNextToken.
 =============================================================================*)

 EXPORT QUALIFIED
 (* Procedure *) Compile;

 PROCEDURE Compile(ProgramFName: ARRAY OF CHAR): INTEGER;
 (*
   -2 ---> Parser failed to initialize,
   -1 ---> Program file Open failed,
   else    Number.of syntactic errors found.
 *)
END Compiler.
```

7.3.2 Módulos específicos

El análisis semántico corresponde a la realización de un evaluador para los atributos definidos en el capítulo quinto. El generador de código, en cambio, incorpora parte de la semántica al proceso de compilación realizando algunas de las funciones descritas en el capítulo sexto. Las secuencias de formas sintácticas (las tareas) son convertidas en localidades contiguas de memoria que contienen un código intermedio. Esto permite, con un contador de programa e instrucciones de brinco, realizar eficientemente las operaciones sobre secuencias; el código fue diseñado ex profeso para facilitar la implementación de la función Simula del capítulo seis.

```
DEFINITION MODULE ContextAnalyser;
(*=============================================================================
  Global Description :  This module provides the context analysis for
                        LPC's language. It uses the syntax described by
                        the parser's tables and the information provided
                        by Parser.CurState & Parser.CurArc.
 =============================================================================*)

  FROM Representations IMPORT
  (* Type *)       TableIndex, IdType, DeclId, DeclRecord, EnvRecord;
  FROM Robots IMPORT
  (* Type *)       RELDIR;

  EXPORT QUALIFIED
  (* Var *)        DeclTable, EnvTable, ChannelDecl, CycleLevel,
                   RelDir, TaskDeclared, TaskAssigned,
                   TaskPassed, RobotDeclared, CommDeclared, ParamDeclared,
  (* Procedure *)  InitContext, CheckContext, Lookup, ContextCleanup;
```

```
  VAR
   DeclTable     : DeclRecord; (* Largest declaration space *)
   EnvTable      : EnvRecord;  (* Largest environment descriptor *)

   (*  The following variables are meant to be READ ONLY & to be
       used by CodeGenerator.MOD. Any assignment to these is LETHAL. *)

   ChannelDecl   : BOOLEAN; (* TRUE iff channel declared, expected or used *)
   CycleLevel    : CARDINAL;(* Cycle depth level *)
   RelDir        : RELDIR;  (* Relative direction value *)
   TaskDeclared,            (* Currently being declared task *)
   TaskAssigned,            (* Currently being assigned task *)
   TaskPassed,              (* Currently being passed or using comm. task *)
   RobotDeclared,           (* Currently being declared or used robot *)
   CommDeclared,            (* Currently being declared or used comm. *)
   ParamDeclared : DeclId;  (* Currently being declared or used param. *)

  PROCEDURE InitContext;
  (* Must be called before the first call to CheckContext.
     Unpredictable behaviour would result otherwise.
  *)

  PROCEDURE CheckContext(): BOOLEAN;
  (* Uses Parser.CurArc and Parser.CurState to implement the context sensitive
     aspects of LPC's programming language. It returns TRUE if no error
     occured; FALSE otherwise and it writes the apropriate message.
     It performs its own cleanning on successful termination.
  *)

  PROCEDURE Lookup(Tag: TableIndex; Hint: IdType): DeclId;
  (*
    Looksup Tag in DeclTable according to Hint.
    In   --> Tag : The searched identifier tag,
             Hint: Identifier's type.
    Out  --> Index of declaration entry associated to Tag of type Hint;
             0 if no match.
  *)

  PROCEDURE ContextCleanup;
  (* Garbage collector. Should be called if CheckContext returns FALSE.*)
END ContextAnalyser.


DEFINITION MODULE CodeGenerator;
(*=============================================================================
  Global Description :  This module provides the code generator for
                        LPC's language. It uses the syntax described by
                        the parser's tables and the information provided
                        by Parser.CurState & Parser.CurArc.
 =============================================================================*)

 FROM VirtualMachine IMPORT
 (* Type *)       MemoryRecord;

 EXPORT QUALIFIED
 (* Var *)        RAM,
 (* Procedure *) InitCodeGenerator,
                 Translate, CodeGenCleanup;

 VAR
  RAM : MemoryRecord;   (* Largest available memory. *)

 PROCEDURE InitCodeGenerator;
 (* MUST be called before the first call to Translate. *)

 PROCEDURE Translate(): BOOLEAN;
 (* Uses Parser.CurArc and Parser.CurState to translate LPC's programming
    language into VirtualMachine.DEF machine language.  *)
```

```
PROCEDURE CodeGenCleanup;
 (* Garbage collector. Should be called when Translate returns FALSE. *)
END CodeGenerator.
```

7.4 Herramientas generales

Solo se presentan las principales herramientas importadas de la biblioteca de módulos de uso general del autor. Dichos módulos implementan dos tipos de objetos: estructuras de datos básicas y funciones para la entrada y la salida de datos.

7.4.1 Estructuras de datos básicas

Se describen tres módulos de este género: las listas, las colas y las pilas. Estas estructuras de datos se usan en repetidas ocasiones en la programación del sistema; estructuran datos de distintos tipos, o sea, que son genéricas. Ya que *Modula-2* no ofrece algún mecanismo equivalente al paquete genérico de *Ada*™, se diseñaron módulos especiales con los cuales se evitó programar estas estructuras para cada tipo de datos. Los listados que seguidamente se exhiben muestran cómo se obtienen estructuras de datos genéricas en *Modula-2* sin desechar la seguridad proporcionada por su chequeo estricto de tipos.

```
DEFINITION MODULE Stacks;
(*************************************************************************
  Global Description :  This module provides generic Stacks

     A stack is a couple ( Stk, Port ) where Stk is the stack structure and
  Port is a variable of the type of the elements that Stk can handle.
  When creating the structure a variable must be assigned to it. The role of
  that variable is double:
            1)- it provides the base type size of the stack structure;
            2)- it enforces strong type checking, relying on the language
                assignment construct.
    Operations on the stack structure IMPLICITELY use the Port variable.
  That's why these operations are described as side effects. A single Port
  variable may be the port to many stacks as long as all stacks are of the
  same base type.

                              ! WARNING !

    The port of a stack structure MUST have the same LIFELENGTH and SCOPE
  as the structure itself.
*************************************************************************)

FROM SYSTEM IMPORT
(* Type *)     ADDRESS;
```

```
EXPORT QUALIFIED
    (* Type       *) Stack,
    (* Var        *) StackError,
    (* Procedures *) NewStack, Push, Pop, Top, Empty, FreeStack;
TYPE
  Stack ;      (* Obscure type (!) *)
VAR
  StackError : CARDINAL;

(*  : Possible error codes :

      0 --------->        No error ocurred,
      1 --------->        Out of memory,
      2 --------->        Undefined stack,
      3 --------->        Tried to Pop or Top from an empty stack,
      4 --------->        Tried to FreeStack a non-empty stack.
*)

PROCEDURE NewStack(PortAdr: ADDRESS;PortTypeSize: CARDINAL) : Stack;
(*
       New creates, if possible, a stack whose base TYPE is the same as
 it's associated (Port) variable. The only way to interact with the stack
 is through the variable (Port) whose address is PortAdr and whose TYPE
 size is PortTypeSize. The port must be a VAR with the same life length and
 scope as the new stack.

 In --> PortAdr ::= The input/output port address to the desired stack.
                    It can be safely obtained with SYSTEM.ADR(Port).
        PortTypeSize ::= The base type size for the desired stack.
                    It can be safely obtained with SYSTEM.TSIZE(Port).
 Out -> An empty stack ready to handle elements of the TYPE of Port, if
        no error occured.

 SideEffect ---> StackError is set to 0,1 or 2.
*)

PROCEDURE Push(Stk : Stack);
(*
 In ---> Stk ::= Stack structure successfully created with NewStack.

 SideEffect ---> The contents of the port variable associated to Stk
                 is pushed onto Stk, if no error occured.
                 StackError is set to 0,1 or 2.
*)

PROCEDURE Pop(Stk : Stack);
(*
 In ---> Stk ::= Stack structure successfully created with NewStack.

 SideEffect ---> The top element of Stk, if any, is poped, if no error.
                 StackError is set to 0,2 or 3.
*)

PROCEDURE Top(Stk : Stack);
(*
 In ---> Stk ::= Stack structure successfully created with NewStack.

 SideEffect ---> The top element of Stk is in Port if no error.
                 StackError is set to 0,2 or 3.
*)

PROCEDURE Empty(Stk: Stack): BOOLEAN;
(*
 In  ---> Stk ::= Stack structure successfully created with NewStack.
 Out ---> TRUE if Stk is empty; FALSE if not empty or error occured.

 SideEffect ---> StackError is set to 0 or 2.
*)
```

```
  PROCEDURE FreeStack(VAR Stk: Stack);
  (*
   In  ---> Stk ::= Any empty stack structure successfully created with
                    NewStack.
   Out ---> Stk ::= Invalid data.

   SideEffect ---> StackError is set to 0,2 or 4.
  *)
END Stacks.
```

No se muestran los listados íntegros de los módulos de listas y de colas ya que son sumamente parecidos al de pilas.

```
DEFINITION MODULE Lists;
(*************************************************************************
  Global Description :  This module provides generic linked lists
 *************************************************************************)

   FROM SYSTEM IMPORT
   (* Type *)          ADDRESS;
   EXPORT QUALIFIED
   (* Type *)          List,
   (* Var *)           ListError,
   (* Procedure *)     NewList, IsEmpty, First, Remove,
                       Insert, Next, Dispose;

   TYPE       List;
   VAR        ListError : CARDINAL;

 (*_____: Possible error codes :_____________________________________

          0 --------->  No error ocurred,
          1 --------->  General failure,
          2 --------->  Out of memory,
          3 --------->  Tried to REMOVE or FIRST on an empty list or position,
          4 --------->  Tried to DISPOSE of a non-empty list,
          5 --------->  No next element available.
 ____________________________________________________________________*)
                ETC.
END Lists.


DEFINITION MODULE Queues;
(*************************************************************************
  Global Description :  This module provides generic FIFO queues
 *************************************************************************)

  FROM SYSTEM IMPORT
     (* Type       *) ADDRESS;
  EXPORT QUALIFIED
     (* Type       *) Queue,
     (* Var        *) QueueError,
     (* Procedures *) NewQueue, In, Out, Top, Size, Dispose;
  TYPE     Queue;
  VAR      QueueError : CARDINAL;

 (*_____: Possible error codes :_____________________________________

            0 --------->  No error ocurred,
            1 --------->  Out of memory,
            2 --------->  General failure,
            3 --------->  Tried to OUT or TOP from an empty queue,
            4 --------->  Tried to DISPOSE of a non-empty queue.
 ____________________________________________________________________*)
                ETC.
END Queues.
```

7.4.2 Entradas y salidas

El módulo de pantallas suministra elementos esenciales para la animación de *LPC* y contribuye en forma notable a la facilidad de uso del sistema. Los paquetes de ventanas agregan, por lo general, estética y funcionalidad a los sistemas que los incorporan. Aportan estética porque lucen mejor presentados los datos y funcionalidad porque facilitan un mejor aprovechamiento del espacio disponible en la pantalla física. Sin embargo, en su mayoría, estos sistemas no soportan el uso concurrente de las ventanas. Esta característica es necesaria para poder mostrar la actividad desempeñada por los robots cuando éstos habitan en distintos mundos. Por ello, se programó el módulo TextScreens que suministra pantallas virtuales: una pantalla física independiente es emulada por cada una de éstas. Las pantallas se pueden "prender" y "apagar" como si fueran terminales, se les puede escribir aunque estén "apagadas", son móviles y se colocan en la pantalla física cual hojas de papel sobre un escritorio, sus tamaños son variables y ajustables, proporcionan corrimientos hacia la izquierda, la derecha, arriba y abajo del texto contenido y funcionan como un dispositivo de entrada y salida. Los demás detalles de operación de estas pantallas quedan asentados en el listado que se presenta a continuación.

```
DEFINITION MODULE TextScreens;
(*-------------------------------------------------------------------------

  Global Description : This module provides Virtual Text Screens.
                       Screen definition and movement routines are provided.
                       Writing, reading, scrolling and cursor positioning
                       can be performed on any created Screen.
                       RealInOut, InOut, Terminal & Termbase may be used
                       with any screen by setting it active.

  -----------------------------------------------------------------------

                                  FirstEdit: 21/02/87
                                  LastEdit : 9/10/87

  -----------------------------------------------------------------------

              Author : Marc Benveniste
              System : Logitech MODULA-2/86 V. 2.05
     Developped while
        working at   : IIMAS-UNAM,
                       Apdo. Postal 20-726,
                       Mexico D.F., 01000
                       MEXICO
-------------------------------------------------------------------------*)

                FROM VideoCard IMPORT
                (* Type *)       sCorner, sBox, Attribute;
```

```
                EXPORT QUALIFIED
                (* Const *)          MaxL, MaxW,
                (* Type *)           Row, Column, Corner,
                                     Screen, ScreenPtr, Frame,
                (* Var *)            ScreenError, Default,
                (* Procedure *)
  (* Existence operations *)        Open, Close, Resize,
  (* Hidden Screens operations *)   Show,
  (* Shown screens operations *)    Hide, Move, PlaceOnTop, PlaceOnBottom,
                                     Echo, NoEcho,
  (* Screen predicates *)           Shown, EqualSn,
  (* Screen function *)             OnTop, ScreenAt, ScreenPos, ScreenSize,
                                     SetActive, GetActive,
  (* Screen shape operations *)     SetFrame, GetFrame, PutTitle, GetTitle,
  (* Screen contents movements *)   ScrollHor, ScrollVer,
  (* Cursors manipulations *)       SetCursor, GetCursor,
  (* Screen contents InOut *)       PutString, GetString;


  CONST
      MaxW       = 132;
      MaxL       =  43;

  TYPE
      Screen;
      ScreenPtr = POINTER TO Screen;
      Frame     = ARRAY[1 .. 6] OF CHAR;
                  (* UpLeft,Hor,UpRight,Ver,BottomLeft,BottomRight *)
      Row       = [1 .. MaxL];
      Column    = [1 .. MaxW];
      Corner    = RECORD
                    X : Column;
                    Y : Row
                  END;

  VAR
      ScreenError    : CARDINAL ;
      Default        : Frame;       (* ┌─┐│└┘ *)

   (*──────────────────────────────────────────────────────────────────────*
    *                                                                      *
    * ScreenError = 0 ---> Operation was successfully done                 *
    * ScreenError = 1 ---> Insufficient memory                             *
    * ScreenError = 2 ---> Screen not defined                              *
    * ScreenError > 2 ---> Meaning specific to the procedure               *
    *                                                                      *
    *──────────────────────────────────────────────────────────────────────*)

PROCEDURE Open(W: Column; L: Row): Screen ;
(*
 In           ----> W = Width of the screen.
                    L = Length of the screen.
 Out          ----> Open returns, if possible, a screen of dimentions WxL,
                    framed with the default (┌─┐│└┘), Echo option on,
                    cleared with attribute Atributos.TextoNormal,
                    with a home (1,1) cursor position and a hidden status.
 Side Effect ----> ScreenError may be set to 0 or 1.
──────────────────────────────────────────────────────────────────────────*)

PROCEDURE Close(VAR Sn: Screen) ;
(*
 In           ----> Sn = Any screen created with a successful Open operation.
 Out          ----> Sn = Meaningless data, the screen has been closed.
 Side Effect ----> ScreenError may be set to 0 or 2.
                    The output is redirected to the physical screen if
                    the Sn was the active screen for Termbase.
                    The physical screen is restored if Sn is the last screen.
──────────────────────────────────────────────────────────────────────────*)
```

```
PROCEDURE Resize(VAR Sn: Screen; Width: Column; Length: Row);
(*
 In            ----> Sn = Any screen created with a successful Open operation.
                     Width = New intended width.
                     Length = New intended length.
 Out           ----> Sn = A resized screen if successful, the old screen
                        otherwise.
 Side Effect ----> ScreenError may be set to 0, 1 or 2.
                     The contents of the old screen, the header and the footer
                     are preserved as they fit the new size. Cursor position
                     is either the last one or the bottom left corner
                     depending on the new size. The frame, position, order and
                     status of the screen are preserved.
-------------------------------------------------------------------------------*)

PROCEDURE Show(Sn: Screen; C: SCorner; UpLeft: BOOLEAN) ;
(*
 In            ----> Sn = Legal hidden screen.
                     C  = Physical corner for placing virtual frame corner.
                     UpLeft = True iff virtual frame corner is the upper left,
                              bottom right otherwise.
 Side Effect ----> Sn is placed on top of any other shown screen with its
                     upper left (bottom right) frame corner placed at C if
                     UpLeft is true (false).
                     When Sn is the first, physical screen is saved and cleared.
                     ScreenError may be set to 0,2 or 3.
                     ScreenError = 3 ---> Screen already shown.
-------------------------------------------------------------------------------*)

PROCEDURE Hide(Sn: Screen) ;
(*
 In            ----> Sn = Legal shown screen.
 Side Effect ----> Sn is now hidden.
                     If Sn is the last shown screen, the physical screen is
                     restored.
                     ScreenError may be set to 0,2 or 3.
                     ScreenError = 3 ---> Screen already hidden
-------------------------------------------------------------------------------*)

PROCEDURE Move(Sn: Screen; NewC: SCorner; UpLeft: BOOLEAN);
(*
 In            ----> Sn = Legal shown screen.
                     NewC = New physical corner to place virtual frame corner.
                     UpLeft = True iff upper left virtual frame corner, bottom
                              right otherwise.
 Side Effect ----> Sn is at the new position.
                     ScreenError may be set to 0, 2 or 3.
                     ScreenError = 3 ---> Screen hidden, cannot be moved.
-------------------------------------------------------------------------------*)

PROCEDURE PlaceOnTop(Sn: Screen);
(*
 In          -> Sn = Legal shown screen.
 Side Effect -> Sn is now the screen on top of any other shown one.
                ScreenError may be set to 0, 2 or 3.
                ScreenError = 3 -> Screen hidden, can't be placed on top.
-------------------------------------------------------------------------------*)

PROCEDURE PlaceOnBottom(Sn: Screen);
(*
 In          -> Sn = Legal shown screen.
 Side Effect -> Sn is now the screen below any other shown one.
                ScreenError may be set to 0, 2 or 3.
                ScreenError = 3 -> Screen hidden, can't be placed on bottom.
-------------------------------------------------------------------------------*)
```

```
PROCEDURE Echo(VAR Sn: Screen);
(*
 In           ----> Sn = Any shown screen.
 Out          ----> Sn = The same screen with the echo option turn on.
 Side Effect ----> Sn is refreshed.
                   ScreenError may be set to 0 or 3
                   ScreenError = 3 --> Sn is not shown.
 ____________________________________________________________________*)

PROCEDURE NoEcho(VAR Sn: Screen);
(*
 In           ----> Sn = Any shown screen.
 Out          ----> Sn = The same screen with the echo option turn off.
 Side Effect ----> ScreenError may be set to 0 or 3
                   ScreenError = 3 --> Sn is not shown.
 ____________________________________________________________________*)

PROCEDURE EqualSn(First,Second: Screen): BOOLEAN ;
(*
 In           ----> First & Second: Legal screens.
 Out          ----> EqualSn returns TRUE iff First=Second; FALSE otherwise.
 Side Effect ----> ScreenError is set to 0.
 ____________________________________________________________________*)

PROCEDURE Shown(Sn: Screen): BOOLEAN ;
(*
 In           ----> Sn = Legal screen.
 Out          ----> Shown returns TRUE iff Sn is shown; FALSE otherwise.
 Side Effect ----> ScreenError may be set to 0 or 2.
 ____________________________________________________________________*)

PROCEDURE OnTop(): Screen;
(*
 Out          ----> The top shown screen if any. Meaningless data otherwise.
 Side Effect ----> ScreenError may be set to 0 or 3.
                   ScreenError = 3 ----> No screen shown.
 ____________________________________________________________________*)

PROCEDURE ScreenPos(Sn: Screen; VAR C: SCorner; VAR UpLeft: BOOLEAN);
(*
 In           ----> Sn = Legal shown screen.
 Out          ----> C = Physical corner where Sn frame corner is placed.
                   UpLeft = TRUE iff Sn frame corner is upper left;
                            FALSE iff Sn frame corner is bottom right.
 Side Effect ----> ScreenError may be set to 0,2 or 3.
                   ScreenError = 3 ---> Screen is hidden.
 ____________________________________________________________________*)

PROCEDURE ScreenSize(Sn: Screen; VAR Width: Column; VAR Length: Row);
(*
 In           ----> Sn = Legal screen.
 Out          ----> Width = The width of Sn.
                   Length = The length of Sn.
 Side Effect ----> ScreenError may be set to 0 or 2.
 ____________________________________________________________________*)

PROCEDURE ScreenAt(C: SCorner;VAR Local: Corner): Screen;
(*
 In           ----> C = Physical corner.
 Out          ----> ScreenAt returns the shown screen, if any, that is
                          visible at C; invalid data otherwise.
                   Local = The corner of the returned screen, if any, placed
                           at C.
 Side Effect ----> ScreenError may be set to 0,3 or 4
                   ScreenError = 3 --> C is in no shown screen
                   ScreenError = 4 --> C is on the frame of the returned
                                       screen, but Local is not valid.
 ____________________________________________________________________*)
```

```
PROCEDURE SetActive(Sn: Screen);
(*
 In           ----> Sn = Legal screen.
 Side Effect  ----> The output of Termbase, Terminal and InOut is
                    redirected to Sn if ScreenError = 0; stays
                    the same otherwise.
                    ScreenError may be set to 0,2 or 3
                    ScreenError = 3  ---> Redirection Failed.

 WARNING: User is responsible for restoring redirection of output to the
          normal output device by closing the active screen if any or by
          calling SetActive with an undefined Screen variable.
 ------------------------------------------------------------------------*)

PROCEDURE GetActive(): Screen;
(*
 Out          ----> The currently active screen for Termbase,Terminal & InOut
                    if ScreenError = 0; invalid data otherwise.
 Side Effect  ----> ScreenError may be set to 0 or 3
                    ScreenError = 3  ---> No redirection has been done.
 ------------------------------------------------------------------------*)

PROCEDURE ScrollVer(Sn: Screen; A: Attribute; Up: BOOLEAN; Lines: Row) ;
(*
 In           ----> Sn = Legal screen.
                    A = Attribute for the incomming blank lines, if any.
                    Lines = The number of lines to scroll.
                    Up = TRUE iff scroll-up; FALSE iff scroll-down.
 Side Effect  ----> Sn = The input screen contents have been scrolled.
                    ScreenError may be set to 0 or 2.
 ------------------------------------------------------------------------*)

PROCEDURE ScrollHor(Sn: Screen;A: Attribute;Right: BOOLEAN;Col: Column) ;
(*
 In           ----> Sn = Legal screen.
                    A = Attribute for the incomming blank columns, if any.
                    Col = The number of columns to scroll.
                    Right = TRUE iff scroll-right; FALSE iff scroll-left.
 Side Effect  ----> Sn = The input screen contents have been scrolled.
                    ScreenError may be set to 0 or 2.
 ------------------------------------------------------------------------*)

PROCEDURE SetFrame(Sn: Screen; F: Frame; A: Attribute);
(*
 In           ----> Sn = Legal screen.
                    F = Frame to assign to Sn.
                    A = Attibute of that frame.
 Side Effect  ----> Sn = The same screen with the new frame,attribute.
                    ScreenError may be set to 0 or 2.
 ------------------------------------------------------------------------*)

PROCEDURE GetFrame(Sn: Screen;VAR F: Frame; VAR A: Attribute);
(*
 In           ----> Sn = Legal screen.
 Out          ----> F = Frame currently assigned to Sn.
                    A = Attribute of that frame.
 Side Effect  ----> ScreenError may be set to 0 or 2.
 ------------------------------------------------------------------------*)
```

```
PROCEDURE PutTitle(Sn:Screen;Top:BOOLEAN;Title:ARRAY OF CHAR;A:Attribute);
(*
 In            ----> Sn = Legal screen.
                     Top = TRUE iff the title should be placed on the top
                           horizontal part of the frame; FALSE if on the bottom.
                     Title = String to place on the frame. A Null (0c) char
                           is interpreted as an end of string.
                     A = Attribute to use when writing Title.
 Out           ----> Sn = The same screen with as much as possible of the title
                           on the frame.
 Side Effect ----> ScreenError may be set to 0, 2 or 3.
                   ScreenError = 3 ---> Title truncated.
_____________________________________________________________________________*)

PROCEDURE GetTitle(Sn: Screen; Top: BOOLEAN;VAR Title: ARRAY OF CHAR;
                                              VAR A: Attribute          );
(*
 In            ----> Sn = Legal screen.
                     Top = TRUE iff the title wanted is the top one;
                           FALSE if the bottom.
 Out           ----> Title = String to place the title in. A Null (0c) char
                           ends the string if title shorter than destination.
                     A = Attribute of the title.
 Side Effect ----> ScreenError may be set to 0, 2 or 3.
                   ScreenError = 3 ---> Title truncated.
_____________________________________________________________________________*)

PROCEDURE SetCursor(Sn: Screen; C: Corner; TurnOn: BOOLEAN);
(*
 In            ----> Sn = Legal screen.
                     C = Corner where the cursor should be placed.
                     TurnOn : TRUE if cursor should be visible;
                              FALSE otherwise.
 Side Effect ----> Sn = The same screen with the modified cursor position.
                   ScreenError may be set to 0,2 or 3.
                   ScreenError = 3 ---> C out of screen definition, cursor
                                        position unchanged.
_____________________________________________________________________________*)

PROCEDURE GetCursor(Sn: Screen; VAR C: Corner; VAR TurnOn: BOOLEAN);
(*
 In            ----> Sn = Legal screen.
 Out           ----> C = Corner where the cursor is placed.
                     TurnOn = TRUE if cursor visible; FALSE otherwise.
 Side Effect ----> ScreenError may be set to 0 or 2.
_____________________________________________________________________________*)

PROCEDURE PutString(Sn: Screen; Src: ARRAY OF CHAR; A: Attribute) ;
(*
 In            ----> Sn  = Any Screen optained by a successful call to Open.
                     Src = Array of char to be written.
                     A   = Attribute to be used when writing Src.
 Side Effect ----> Sn  = The Screen contents have been modified:
                         Scr is written on the screen Sn starting at the
                         current cursor position, wrapping and scrolling is
                         automatically done. If Scr contains a 0c (NULL),
                         it will be considered the ending character of the
                         string. The cursor position is actualized to be the
                         next to the last written.
                   ScreenError may be set to 0 or 2.
_____________________________________________________________________________*)
```

```
PROCEDURE GetString(Sn: Screen; VAR Dst: ARRAY OF CHAR; CharNum: CARDINAL);
(*
 In           ----> Sn = Legal screen.
                    Dst = Place holder for the string to be read.
                    CharNum = Number of chars to be read.
 Out          ----> Dst = String read from screen, starting at the cursor
                          position, ending either when the length of Dst is
                          reached or when CharNum chars have been read or when
                          there are no more chars to be read.
 Side Effect ----> ScreenError may be set to 0, 2 or 3.
                    ScreenError = 3 ---> Dst not fully used, 0c ends it.
                                                                          *)
END TextScreens.
```

Se aprovechó la función de lectura de las pantallas para programar un módulo de menús basados en archivos. Este módulo permite que se elaboren menús con cualquier editor de texto, que se utilice cualquier pantalla como menú y que se copien archivos de texto a las pantallas.

```
DEFINITION MODULE FileMenus;
(*=========================================================================
  Global Description : This module provides fixed vertical multifield
                       textmenus whose items are provided in a file.
=========================================================================*)

    FROM TextScreens IMPORT
    (* Type *)        Screen, Row, Column;

    EXPORT QUALIFIED
    (* Procedure *)  Build, Choose, Reflect;


  PROCEDURE Build(FileName: ARRAY OF CHAR): Screen;
(*
  This procedure opens a screen of the size specified in the file where
  the menu is described. A menu is correctly specified if the it has the
  following syntax:
                    ww11<item1>crlf.....<item11>crlf
    where ww is a 2 digits Column, 11 a 2 digits Row and |item*| = ww

  In    ---> FileName ::= Full file name where the menu is described.
  Out   --->              A Screen openned according to menu specification
                          if possible;
                          Invalid data otherwise.
*)

  PROCEDURE Choose(Sn: Screen;First,Width: Column;Def:Row): INTEGER;
(*
  In         ----> Sn    ::= Screen where the menu is exposed.
                   First ::= Column number where items start.
                   Width ::= Width of items.
                   Def   ::= Item selected by default.
  Out        ----> The number of the item chosen if any and
                   -1 if Cancel
                   -2 if TabLeft,CurLeft or Back&Top.
                   -3 if TabRight,CurRight or Forward&Bottom.
  Side Effects ----> If Sn is not shown, it will be;
                     otherwise it wil be placed on top.
*)
```

```
  PROCEDURE Reflect(Sn: Screen;First,Width: Column; Select: Row);
(*
  In          ----> Sn     ::= Screen where the menu is exposed.
                    First ::= Column number where items start.
                    Width ::= Width of items.
                    Select::= Choosed item number.
*)
END FileMenus.
```

7.5 La vista externa

El caracter de observable que se le ha conferido al universo de *LPC* se realiza con los módulos que se describen en esta sección.

7.5.1 Representaciones gráficas

La representación de un mundo en una pantalla es una cuadrícula; esta incluye en la parte superior la numeración correspondiente a las avenidas, y en la parte inferior aparece el nombre del mundo representado. Si el monitor es monocromático, los bordes aparecen en modo inverso; en caso contrario, figuran en negro sobre fondo rojo. La Figura 1 muestra la representación de seis mundos contemporáneos, cinco habitados y uno compartido.

En la cuadrícula que representa a un mundo en la pantalla, las calles son horizontales y se numeran de arriba hacia abajo, mientras que las avenidas son verticales y se numeran de izquierda a derecha. La existencia de un muro en un cruce determinado se representa en la pantalla con los siguientes caracteres en el cruce correspondiente:

En un monitor monocromático, estos caracteres aparecen en modo inverso; en uno a colores, vienen en negro sobre fondo rojo para dar la ilusión de ser ladrillos. Los espacios marcados con ⣿, son los lugares que ocupa la representación de los trompos que contiene el cruce.

Los trompos se representan en la pantalla por el número positivo que denota la cantidad acumulada de éstos en un determinado cruce; el dígito superior indica las decenas y el inferior las unidades (el cero queda representado con el espacio). En la versión a colores, los dígitos son amari-

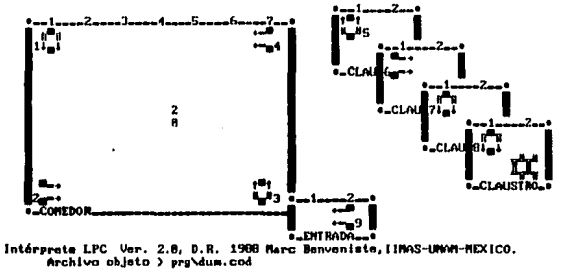


Figura 1 Los Mundos de *LPC*.

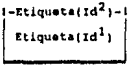
llos; en la otra, son blancos. Los robots quedan representados, de acuerdo a su dirección, por los siguientes juegos de caracteres:

NORTE	ESTE	SUR	OESTE
d u	d u	d u	d u

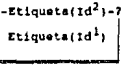
donde "d" y "u" representan el número del proceso al que pertenece el robot: "u" corresponde al dígito de las unidades y "d" al de las decenas (el cero sigue representándose con un espacio). Estas representaciones vienen en dos tonos de azul cuando el monitor puede pintar colores y en modo normal en caso contrario.

Como se asentó en los primeros capítulos de la tesis, esta animación de los robots y de sus mundos permite la observación directa de las acciones de los primeros al ejecutar las tareas asignadas con el lenguaje de *LPC*. En efecto, todas las instrucciones del lenguaje cuya semántica modifica el estado de la MV_{LPC} en sus componentes de declaración de mundos o de robots han de ser directamente observables en las representaciones de éstos. La creación y la destrucción de procesos también queda directamente observa-

ble con las representaciones adoptadas. Se representa la ejecución de los envíos de señales con las siguientes ventanas informativas:



Id¹ ! Id²



Id¹ ? Id²

donde la longitud de los identificadores ha sido limitada a ocho por razones de visibilidad. Ya que las ventanas son colocadas encima del robot que ejecuta la instrucción, y que permanecen puestas hasta que termine dicha ejecución, se requiere que cubran la menor área posible, manteniendo un ancho que permita el uso de identificadores significativos tanto para los robots y las tareas (Id^1), como para los buzones y los canales (Id^2). El marco de estas ventanas aparece en negro sobre fondo verde en monitores a color en el caso de la emisión de una señal, y en blanco sobre fondo rojo al tratarse de una recepción (se pensó en estos dos colores por su tradicional significado en los semáforos); en monitores monocromáticos, las de emisión se representan en modo inverso, y las de recepción en modo normal (se pierde la alusión a los semáforos, pero conforta a los daltonianos).

Así, los procesos bloqueados se caracterizan por tener la representación de sus robots traslapada por la ventana correspondiente a las señales esperadas, indicando tanto el nombre del emisor como el de la línea de comunicación.

Cabe resaltar que si todos los robots presentes están cubiertos por ventanas de recepción, se puede deducir de la observación que el programa ha caído en una situación de bloqueo mortal. El intérprete de *LPC* detecta esta situación y termina gentilmente la ejecución; queda sin embargo en la pantalla una representación del estado final que incluye la información relevante al análisis de lo acontecido.

El módulo que realiza las representaciones gráficas de *LPC* es el siguiente:

```
DEFINITION MODULE OutViews;
(*=============================================================================
  Global Description :  This module provides LPC's external view.
                        The procedures enclosed by this module are used
                        by VirtualMachine and WorldManager.
 =============================================================================*)

FROM Worlds IMPORT
(* Type *)       CROSSING, WORLD;
FROM Robots IMPORT
(* Type *)       ROBOT;
FROM Representations IMPORT
(* Type *)       DeclId;
```

```
FROM VideoCard IMPORT
(* Type *)       Attribute;

EXPORT QUALIFIED
(* Const *)      HorScale, VerScale,
(* Var *)        WalAttr, FreAttr,
(* Procedure *) ShowRobot, ShowFree, ShowWall, ShowBeeper,
                 ShowPoopBeeper, ShowInOut;

CONST
 HorScale = 5;       (* Horizontal scaling factor. *)
 VerScale = 2;       (* Vertical scaling factor. *)
VAR
 WalAttr,
 FreAttr : Attribute;   (* Wall & Free attributes *)

PROCEDURE ShowRobot(r: ROBOT);
(*
  Shows a robot in his world.
  In  --> r :: Robot to be shown.
*)

PROCEDURE ShowFree(VAR w: WORLD; c: CROSSING);
(*
  Displays a free cell at crossing c.
  In  --> w :: World where cell belongs,
          c :: Crossing where cell should be displayed;
  Out --> w :: Modified world in its external view.
*)

PROCEDURE ShowWall(VAR w: WORLD; c: CROSSING);
(*
  Displays a wall cell at crossing c.
  In  --> w :: World where cell belongs,
          c :: Crossing where cell should be displayed;
  Out --> w :: Modified world in its external view.
*)

PROCEDURE ShowBeeper(VAR w: WORLD; c: CROSSING);
(*
  Displays the beeper placed at crossing c.
  In  --> w :: World where cell belongs,
          c :: Crossing of which beepers should be displayed;
  Out --> w :: Modified world in its external view.
*)

PROCEDURE ShowPoopBeeper(r: ROBOT);
(*
  Displays the beeper placed at poop crossing of r.
  In  --> r :: Well defined robot;
  Out --> r.Wid :: Modified world in its external view.
*)

PROCEDURE ShowInOut(VAR r: ROBOT; To,By : DeclId; In: BOOLEAN);
(*
  Shows the input or output of r.
  In  --> r :: Robot executing an input or output command,
          To:: Index of VirtualMachine.Context.Decl indicating the
               sender's or the recipient's tags,
          By:: Index of VirtualMachine.Context.Decl indicating the
               communication mean's tag,
          In:: TRUE if input command; output command otherwise.
  Out --> r :: If r^.Msg was not created, it will be.
*)
END outViews.
```

7.5.2 El manejador de mundos

El analizador de contexto, el editor y el intérprete necesitan cargar los mundos de *LPC*. Por ello, se encerraron en el módulo que a continuación se presenta, los procedimientos requeridos para manipularlos.

```
DEFINITION MODULE WorldManager;
(*=============================================================================
  Global Description :  This module provides operations on LPC's worlds.
 =============================================================================*)

 FROM Worlds IMPORT
 (* Type *)       WORLD;

 EXPORT QUALIFIED
 (* Var *)        Done,
 (* Procedure *) LoadWorld,
                  FillScreen, SetHeader, CleanInternal, EditWorld,
                  SaveWorld;

 VAR
  Done : BOOLEAN;

 PROCEDURE LoadWorld(FileName: ARRAY OF CHAR): WORLD;
 (*
   Loads the world from FileName. Sets Done to TRUE if correctly loaded.
   Input     ---> FileName : Complete file name where world is saved.
   Output    ---> RETURNS: Initialized world if Done; NIL otherwise.
   SideEffects ---> Done   : TRUE if load was successful; FALSE otherwise,
                    The world saved in FileName is loaded and its screen
                    is filled accordingly.
 *)

 PROCEDURE FillScreen(VAR w: WORLD);
 (*
   Reflects w^.Int contents on w^.Ext.
   WARNING :: Unpredictable behavior if w not properly defined.
 *)

 PROCEDURE SetHeader(VAR w: WORLD);
 (*
   Writes w^.Ext's header according to w^.Width.
   WARNING :: Unpredictable behavior if w not properly defined.
 *)

 PROCEDURE CleanInternal(VAR w: WORLD);
 (*
   Clears w^.Int. For every c: CROSSING, W(c) = (free,0)
   WARNING :: Unpredictable behavior if w not properly defined.
 *)

 PROCEDURE EditWorld(VAR w: WORLD);
 (*
   Edits the world.
   Input     ---> w    : The world to be edited, it should be shown,
   Output    ---> w    : The edited world if Done; unmodified world otherwise,
   SideEffects ---> Done : TRUE if successful operation; FALSE otherwise,
                    A help screen is provided.
 *)
```

```
  PROCEDURE SaveWorld(w: WORLD; FileName: ARRAY OF CHAR);
  (*
    Saves the world to FileName.
    Input     ---> FileName : Complete file name where world should be saved,
                  w         : World to be saved.
    SideEffects ---> Done : TRUE if successful operation; FALSE otherwise,
                     The world w is saved in FileName if Done.
  *)
END WorldManager.
```

7.5.3 El editor

El editor permite el diseño de los mundos en los que habrán de programarse los robots. Al igual que el programa principal, recibe los comandos del usuario a través de un menú e incluye una pantalla de ayuda, evitando así el uso de un manual.

7.6 El intérprete

El intérprete proporciona las facilidades interactivas del sistema. Permite ajustar las posiciones de los mundos durante la ejecución, seleccionar la velocidad de ejecución de la máquina virtual y alterar las velocidades relativas de los robots.

```
DEFINITION MODULE Interpreter;
(*=============================================================================
  Global Description :  This module contains LPC's interpreter.
 =============================================================================*)

 EXPORT QUALIFIED
 (* Procedure *) Run;

 PROCEDURE Run;
 (*
    SideEffects --> The currently loaded LPC program is interpreted
                    Interactive facilities are provided.
    WARNING:: Unpredictable behaviour if no program is loaded.
 *)
END Interpreter.
```

El manejador de programas provee un procedimiento para cargar el código y el intérprete lo ejecuta iterando la función proporcionada por la máquina virtual.

7.6.1 La máquina virtual

La función `Meaning` implementa la función Simula (S.9) y los estados *ESTADOS* (S.2) quedan realizados por las variables `Memory`, `Names`, `Context`, `Environment` y `Assign`.

```
DEFINITION MODULE VirtualMachine;
(*=============================================================================
  Global Description :  This module defines the concurrency simulator used
                        by LPC's interpreter.
 =============================================================================*)

FROM SYSTEM IMPORT
(* Type *)       BYTE;
FROM Representations IMPORT
(* Const *)      MemEnd,
(* Type *)       AdrSpace, NamesRecord, PROCESS, DeclRecord, EnvRecord;
FROM Queues IMPORT
(* Type *)       Queue;
FROM Stacks IMPORT
(* Type *)       Stack;

EXPORT QUALIFIED
(* Const *)      MaxGuardNum, MaxBoolLgth, StopAdr,
(* Type *)       ASSIGN, InstSet, AdrMode, GuardIndex, MEMORY, MemoryRecord,
(* Var *)        Memory, Names, Context, Environment, Assign,
(* Procedure *) Meaning;

CONST
 MaxGuardNum    = 16;          (* Maximum guard number *)
 MaxBoolLgth    = 64;          (* Boolean evaluation stack size *)
 StopAdr        = 0;           (* Address where STOP is *)
TYPE
 ASSIGN         = RECORD
                    Actual : PROCESS;
                    Ready  : Queue
                  END;
 InstSet        = (STEP, SPIN, PICK, DROP, CALL, NEWPROCESS,
                   SEND, RECEIVE, ASK, ANSWER, SKIP, MAKEGUARD, ELSEGUARD,
                   EVALGUARD, JUMP, LogAND, LogOR, LogNOT, LetterPRED,
                   RequestPRED, NorthPRED, WallPRED, RobotPRED, BeeperPRED,
                   LoadedPRED, TruePRED, CLI, STI, RET, STOP);
 AdrMode        = (Immediate, Direct);
 GuardIndex     = [0 .. MaxGuardNum];
 MEMORY         = POINTER TO ARRAY AdrSpace OF BYTE;
 MemoryRecord= RECORD
                    Size : [0 .. MemEnd];
                    Cont : MEMORY
                  END;
VAR
 Memory         : MemoryRecord;             (* Task's code shared memory
                                               Memory.Cont^[StopAdr] = STOP *)
 Names          : NamesRecord;              (* Identifier tags *)
 Context        : DeclRecord;               (* Set of declarations *)
 Environment    : EnvRecord;                (* Task environment descriptions *)
 Assign         : ASSIGN;                   (* Assignments *)
```

```
  PROCEDURE Meaning(): CARDINAL;
  (*
    Input  --> Memory, Names, Context, Environment & Assign must be loaded.
               This should be done by the program module.
               No checks are performed.
    Output --> NoError: 0      --> No error
                        1      --> NormalTermination
                        2      --> Deadlock detected
                        Else   --> RunTime error.
    SideEffects --> One simulation cycle is performed.
  *)
END VirtualMachine.
```

7.6.2 El manejador de programas

Se programó un módulo para archivar, cargar y listar el código generado por el compilador:

```
DEFINITION MODULE ProgramManager;
(*=============================================================================
  Global Description :  This module provides LPC's code loading and saving.
 =============================================================================*)

 FROM Representations IMPORT
 (* Type *)       NamesRecord, DeclRecord, EnvRecord;
 FROM VirtualMachine IMPORT
 (* Type *)       MemoryRecord;

 EXPORT QUALIFIED
 (* Procedure *) OpenCommunications, CloseCommunications,
                 StartLoaded, LoadProgram, SaveProgram,
                 LoadCode, ListCode, MemoryDump;

 PROCEDURE OpenCommunications;
 (*
   Opens channels & mailboxes as declared in VirtualMachine.Context.
   SideEffects --> Communications.CommError is accordingly set.
   WARNING!: No verifications are performed. A valid LPC program
             should be correctly loaded before this procedure is
             called.
 *)

 PROCEDURE CloseCommunications;
 (*
   Closes channels & mailboxes as declared in VirtualMachine.Context.
   SideEffects --> Communications.CommError is accordingly set.
   WARNING!: No verifications are performed. A valid LPC program
             should be correctly loaded before this procedure is
             called.
 *)

 PROCEDURE StartLoaded;
 (*
   Prepares the initialization process.
   SideEffects --> Queues.QueuError is accordingly set.
   WARNING!: It is assumed that a valid LPC program is loaded and
             that VirtualMachine.Assign.Ready is empty.
             No verifications are performed.
 *)
```

```
PROCEDURE LoadProgram(FileName: ARRAY OF CHAR): BOOLEAN;
(*
  Loads an LPC program from FileName.
  It is assumed that:
   1- VirtualMachine.Names.Table                = NIL,
   2- VirtualMachine.Context.Decl               = NIL,
   3- VirtualMachine.Environment.Env            = NIL,
   4- VirtualMachine.Memory.Cont                = NIL,
   5- VirtualMachine.Assign.Actual              = NIL and
   6- Queues.Size(VirtualMachine.Assign.Ready) = 0.
  Input    ---> FileName : Complete file name where program is saved.
  Output   ---> TRUE if program successfully loaded,
                FALSE otherwise.
  SideEffects ---> The program saved in FileName is loaded
                   if TRUE is returned.
  VirtualMachine.Names,
  VirtualMachine.Context,
  VirtualMachine.Environment,
  VirtualMachine.Memory and
  VirtualMachine.Assign          are loaded.
  WARNING:: If a program was loaded storage will be lost.
*)

PROCEDURE SaveProgram(FileName: ARRAY OF CHAR): BOOLEAN;
(*
  Saves the currently compiled program to FileName.
  Input    ---> FileName : Complete file name where program should be saved.
  Output   ---> TRUE if successfully saved,
                FALSE otherwise.
  SideEffects ---> Dynamic storage is recovered from the following if
                   TRUE returned:
  Scanner.IdTable,
  ContextAnalyser.DeclTable,
  ContextAnalyser.EnvTable,
  CodeGenerator.RAM.
  WARNING:: Unpredictable results if no program has been compiled,
            if its code has already been loaded with LoadCode or
            if its code has already been saved with SaveProgram.
*)

PROCEDURE LoadCode(): BOOLEAN;
(*
  Loads the code that has just been generated by LPC's compiler.
  It is assumed that:
   1- VirtualMachine.Names.Table                = NIL,
   2- VirtualMachine.Context.Decl               = NIL,
   3- VirtualMachine.Environment.Env            = NIL,
   4- VirtualMachine.Memory.Cont                = NIL,
   5- VirtualMachine.Assign.Actual              = NIL and
   6- Queues.Size(VirtualMachine.Assign.Ready) = 0.
  Output   ---> TRUE if successfully loaded,
                FALSE otherwise.
  SideEffects ---> Dynamic storage is recovered from the following if
                   TRUE returned:
  Scanner.IdTable,
  ContextAnalyser.DeclTable,
  ContextAnalyser.EnvTable,
  CodeGenerator.RAM.
  WARNING:: If a program was loaded storage will be lost.
            Unpredictable results if no program has been compiled,
            if its code has already been loaded with LoadCode or
            if its code has already been saved with SaveProgram.
*)
```

```
PROCEDURE ListCode(VAR Na: NamesRecord;VAR D: DeclRecord; VAR E: EnvRecord;
                   VAR M: MemoryRecord);
(*
  Lists readable assembly code.
  WARNING:: Unpredictable results if unproper parameters.
*)
PROCEDURE MemoryDump(VAR M: MemoryRecord);
(*
  Writes M.Cont^ byte by byte in hexadecimal format.
  WARNING:: Unpredictable results if unproper parameters.
*)
END ProgramManager.
```

Los dos últimos procedimientos de éste módulo se desarrollaron para facilitar las pruebas del compilador y son los que usa el "desensamblador" que se describe en la siguiente sección.

7.7 Programas auxiliares

Para exhibir la utilidad del desensamblador, se presenta el resultado de una ejecución del mismo lo que permite también dar un vistazo al código intermedio y a las tablas del intérprete generados para un programa ejemplo. El programa *LPC* que se lista a continuación soluciona el problema del productor y del consumidor que comparten un almacén de tamaño unitario, transportado al universo de los robots en el mundo que aparece en la figura 2.

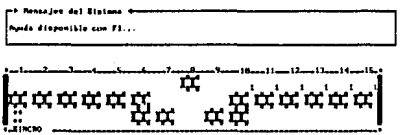


Figura 2 El mundos del programa Sincronización.

```
PROGRAMA Sicronizacion;

  BUZON: Vacio, Lleno, Acabe;

  ROBOT: C COMPARTE Sincro EN 7,2 NORTE Popa CON 0,
         P COMPARTE Sincro EN 9,2 NORTE Proa CON 0;
```

```
TAREA Produce ES
 Avanza; Gira estribor;
 CICLO
  ESCOGE
   Trompo proa -> Recoge>; Gira popa; rompe
  | NO(muro proa O robot proa O Trompo proa) -> Avanza>
   NINGUNA -> Gira popa>; rompe
  FIN (escoge)
 FIN; (ciclo)
 CICLO ESCOGE NO(muro proa O robot proa) -> Avanza> NINGUNA -> rompe> FIN FIN;
 Gira babor; Avanza
FIN Produce;

TAREA Consume ES
 Avanza; Gira babor;
 CICLO
  ESCOGE
   NO Trompo proa -> Deja>; Gira popa; rompe
  | NO(muro proa O robot proa) Y Trompo proa -> Avanza>
  | (muro proa O robot proa) Y Cargado -> Deja>; Gira popa; rompe
  FIN (escoge)
 FIN; (ciclo)
 CICLO ESCOGE NO(muro proa O robot proa) -> Avanza> NINGUNA -> rompe> FIN FIN;
 Gira estribor; Avanza
FIN Consume;

TAREA E_S_Almacen(d: DIRREL) ES
 Gira d; Avanza; Gira popa; Gira d
FIN E_S_Almacen;

TAREA Deposita ES
 YO := E_S_Almacen(estribor);
 ESCOGE
  Trompo Proa -> Gira popa>
  NINGUNA -> Deja>; Gira popa
 FIN; (escoge)
 YO := E_S_Almacen(estribor)
FIN Deposita;

TAREA Retira ES
 YO := E_S_Almacen(babor);
 ESCOGE
  Trompo proa -> Recoge>; Gira popa
  NINGUNA -> Gira popa>
 FIN; (escoge)
 YO := E_S_Almacen(babor)
FIN Retira;

TAREA Productor(C : TAREA) ES
 CICLO
  YO := Produce;
  ESCOGE
   Cargado -> nada>; C?Vacio; YO := Deposita; C!Lleno
   NINGUNA -> nada>; C!Acabe; rompe
  FIN (escoge)
 FIN (ciclo)
FIN Productor;

TAREA Consumidor(P : TAREA) ES
 P!Vacio;
 CICLO
  ESCOGE
   REMITE P En Lleno -> P?Lleno>; YO := Retira; P!Vacio; YO := Consume
  | REMITE P En Acabe -> P?Acabe>;rompe
   NINGUNA -> nada>
  FIN (escoge)
 FIN (ciclo)
FIN Consumidor;
```

```
ES
  C := Consumidor(Productor);
  P := Productor(Consumidor)
FIN Sicronizacion.
```

Tras haber compilado el programa Simulación, se ejecuta el desensamblador tomando, como dato de entrada, el código generado y se obtiene el siguiente listado:

```
Desensamblador LPC  Ver. 2.0, D.R. 1988 Marc Benveniste,IIMAS-UNAM-MEXICO.
      Archivo objeto> prg\sincro.cod

  CONTENIDO DE LA MEMORIA

<    0> 1D 00 01 00 01 1A 17 00 00 0B <   10> 14 00 02 1B 01 00 02 0E 36 00
<   20> 15 00 00 16 00 00 10 17 00 00 <   30> 10 11 0B 28 00 00 1B 0E 33 00
<   40> 0C 32 00 01 00 02 1B 0E 36 00 <   50> 0D 0E 05 00 1A 15 00 00 16 00
<   60> 00 10 11 0B 47 00 00 1B 0E 4F <   70> 00 0C 4E 00 1B 0E 52 00 0D 0E
<   80> 36 00 01 00 03 00 1C 00 01 00 <   90> 03 1A 17 00 00 11 0B 6B 00 03
<  100> 1B 01 00 02 0E 97 00 15 00 00 <  110> 16 00 00 10 11 17 00 00 0F 0B
<  120> 7F 00 00 1B 0E 94 00 15 00 00 <  130> 16 00 00 10 1B 0F 0B 93 00 03
<  140> 1B 01 00 02 0E 97 00 00 0E 5B <  150> 00 1A 15 00 00 16 00 00 10 11
<  160> 0B A8 00 00 1B 0E B0 00 0C AF <  170> 00 1B 0E B3 00 0D 0E 97 00 01
<  180> 00 01 00 1C 01 01 02 00 00 01 <  190> 00 02 01 01 02 00 1C 04 00 09
<  200> 00 00 01 1A 17 00 00 0B D9 00 <  210> 01 00 02 1B 0E E5 00 0C E4 00
<  220> 03 1B 01 00 02 0E E5 00 0D 04 <  230> 00 09 00 00 01 1C 04 00 09 00
<  240> 00 03 1A 17 00 00 0B 01 01 02 <  250> 1B 01 00 02 0E 0C 01 0C 0B 01
<  260> 01 00 02 1B 0E 0C 01 0D 04 00 <  270> 09 00 00 03 1C 04 00 07 00 1A
<  280> 1B 0B 31 01 0A 1B 09 01 02 00 <  290> 01 00 04 00 0A 00 0B 01 02 00
<  300> 02 00 0E 40 01 0C 3F 01 0A 1B <  310> 0B 01 02 00 03 00 0E 43 01 0D
<  320> 0E 13 01 1C 0B 01 02 00 01 00 <  330> 1A 13 01 02 00 02 00 0B 6C 01
<  340> 09 01 02 00 02 00 1B 04 00 0B <  350> 00 0B 01 02 00 01 00 04 00 0B
<  360> 00 0E 8B 01 13 01 02 00 03 00 <  370> 0B 7F 01 09 01 02 00 03 00 1B
<  380> 0E 8B 01 0C 87 01 0A 1B 0E 8B <  390> 01 0D 0E 4A 01 1C 05 04 00 00
<  400> 0D 00 00 0C 00 05 06 00 00 0C <  410> 00 00 0D 00 1D 00 00 00 00 00
```

```
** Tabla de Ident. **
*  0|SICRONIZACION*
*  1|VACIO*
*  2|LLENO*
*  3|ACABE*
*  4|C*
*  5|SINCRO*
*  6|P*
*  7|PRODUCE*
*  8|CONSUME*
*  9|E_S_ALMACEN*
* 10|D*
* 11|DEPOSITA*
* 12|RETIRA*
* 13|PRODUCTOR*
* 14|CONSUMIDOR*
**********************

*Tabla de Ambientes*
*  0|DirRelPar      *
*  1|TaskPar        *
*  2|TaskPar        *
*********************
```

```
**      Tabla de Declaraciones                    **
*   0|  0|TaskId|  396| 1|     |                  *
*   1|  1|MailId|                                 *
*   2|  2|MailId|                                 *
*   3|  3|MailId|                                 *
*   4|  4|RoboId|  5|V| 7| 2|S| 0| 1*
*   5|  5|WorlId|                                 *
*   6|  6|RoboId|  5|V| 9| 2|N| 0| 1*
*   7|  7|TaskId|    1| 1|     |                  *
*   8|  8|TaskId|   87| 1|     |                  *
*   9|  9|TaskId|  184| 2|  0  |                  *
*  10| 11|TaskId|  197| 1|     |                  *
*  11| 12|TaskId|  236| 1|     |                  *
*  12| 13|TaskId|  275| 2|  1  |                  *
*  13| 14|TaskId|  324| 2|  2  |                  *
*************************************************
```

```
** Código generado -->   414 **
<   0>TERMINA PROCESO
<   1>AVANZA
<   2>GIRA  I ESTRIBOR
<   5>EMPIEZA SECCION CRITICA
<   6>Trompo?  I PROA
<   9>MARCA GUARDIA    20
<  12>RECOGE
<  13>FIN SECCION CRITICA
<  14>GIRA  I POPA

<  37>BRINCA    51
<  40>SI NINGUNA    50
<  43>GIRA  I POPA
<  46>FIN SECCION CRITICA
<  47>BRINCA    54
<  50>EVALUA GUARDIAS
<  51>BRINCA    5
<  54>EMPIEZA SECCION CRITICA
<  55>Muro?  I PROA
<  58>Robot?  I PROA

<  79>BRINCA    54
<  82>GIRA  I BABOR
<  85>AVANZA
<  86>RETORNO
<  87>AVANZA
<  88>GIRA  I BABOR
<  91>EMPIEZA SECCION CRITICA
<  92>Trompo?  I PROA
<  95>NO
<  96>MARCA GUARDIA   107

< 119>MARCA GUARDIA   127
< 122>AVANZA
< 123>FIN SECCION CRITICA
< 124>BRINCA   148
< 127>Muro?  I PROA
< 130>Robot?  I PROA
< 133>O
< 134>Cargado?
< 135>Y
< 136>MARCA GUARDIA   147

< 159>NO
< 160>MARCA GUARDIA   168
< 163>AVANZA
< 164>FIN SECCION CRITICA
< 165>BRINCA   176
< 168>SI NINGUNA   175
< 171>FIN SECCION CRITICA
< 172>BRINCA   179
< 175>EVALUA GUARDIAS
< 176>BRINCA   151

< 204>Trompo?  I PROA
< 207>MARCA GUARDIA   217
< 210>GIRA  I POPA
< 213>FIN SECCION CRITICA
< 214>BRINCA   229
< 217>SI NINGUNA   228
< 220>DEJA
< 221>FIN SECCION CRITICA
< 222>GIRA  I POPA
< 225>BRINCA   229

<  17>BRINCA    54
<  20>Muro?  I PROA
<  23>Robot?  I PROA
<  26>O
<  27>Trompo?  I PROA
<  30>O
<  31>NO
<  32>MARCA GUARDIA    40
<  35>AVANZA
<  36>FIN SECCION CRITICA

<  61>O
<  62>NO
<  63>MARCA GUARDIA    71
<  66>AVANZA
<  67>FIN SECCION CRITICA
<  68>BRINCA    79
<  71>SI NINGUNA    78
<  74>FIN SECCION CRITICA
<  75>BRINCA    82
<  78>EVALUA GUARDIAS

<  99>DEJA
< 100>FIN SECCION CRITICA
< 101>GIRA  I POPA
< 104>BRINCA   151
< 107>Muro?  I PROA
< 110>Robot?  I PROA
< 113>O
< 114>NO
< 115>Trompo?  I PROA
< 118>Y

< 139>DEJA
< 140>FIN SECCION CRITICA
< 141>GIRA  I POPA
< 144>BRINCA   151
< 147>EVALUA GUARDIAS
< 148>BRINCA    91
< 151>EMPIEZA SECCION CRITICA
< 152>Muro?  I PROA
< 155>Robot?  I PROA
< 158>O

< 179>GIRA  I ESTRIBOR
< 182>AVANZA
< 183>RETORNO
< 184>GIRA  D     2
< 188>AVANZA
< 189>GIRA  I POPA
< 192>GIRA  D     2
< 196>RETORNO
< 197>EJECUTA  I     9 I ESTRIBOR
< 203>EMPIEZA SECCION CRITICA

< 228>EVALUA GUARDIAS
< 229>EJECUTA  I     9 I ESTRIBOR
< 235>RETORNO
< 236>EJECUTA  I     9 I BABOR
< 242>EMPIEZA SECCION CRITICA
< 243>Trompo?  I PROA
< 246>MARCA GUARDIA   257
< 249>RECOGE
< 250>FIN SECCION CRITICA
< 251>GIRA  I POPA
```

```
<  254>BRINCA   268                    <  280>Cargado?
<  257>SI NINGUNA   267                <  281>MARCA GUARDIA   305
<  260>GIRA  I POPA                    <  284>NADA
<  263>FIN SECCION CRITICA             <  285>FIN SECCION CRITICA
<  264>BRINCA   268                    <  286>ATIENDE D     2     1
<  267>EVALUA GUARDIAS                 <  292>EJECUTA  I    10
<  268>EJECUTA  I     9 I BABOR        <  296>SOLICITA D     2     2
<  274>RETORNO                         <  302>BRINCA   320
<  275>EJECUTA  I     7                <  305>SI NINGUNA   319
<  279>EMPIEZA SECCION CRITICA         <  308>NADA

<  309>FIN SECCION CRITICA             <  340>ATIENDE D     2     2
<  310>SOLICITA D     2     3          <  346>FIN SECCION CRITICA
<  316>BRINCA   323                    <  347>EJECUTA  I    11
<  319>EVALUA GUARDIAS                 <  351>SOLICITA D     2     1
<  320>BRINCA   275                    <  357>EJECUTA  I     8
<  323>RETORNO                         <  361>BRINCA   392
<  324>SOLICITA D     2     1          <  364>Petición? D     2     3
<  330>EMPIEZA SECCION CRITICA         <  370>MARCA GUARDIA   383
<  331>Petición? D     2     2        <  373>ATIENDE D     2     3
<  337>MARCA GUARDIA   364            <  379>FIN SECCION CRITICA

<  380>BRINCA   395                    <  392>BRINCA   330
<  383>SI NINGUNA   391                <  395>RETORNO
<  386>NADA                           <  396>CREA PROCESO     4 I    13 I    12
<  387>FIN SECCION CRITICA            <  405>CREA PROCESO     6 I    12 I    13
<  388>BRINCA   392                    <  414>TERMINA PROCESO
<  391>EVALUA GUARDIAS                **************************
```

Finalmente, se reproduce un extracto del listado del programa con el que se generan las tablas del analizador sintáctico.

```
MODULE BuildTables;
(*=============================================================================
  Global Description :  This module builds the parse tables and saves them
                        on the file LPCTable.DAT.
 =============================================================================*)

 FROM Scanner IMPORT
 (* Type *)       TokenSym;
 FROM ParseTables IMPORT
 (* Const *)      MaxStateNum, MaxArcNum, AcceptArcNum,
 (* Type *)       ArcType, ArcTable, StartTable, StateErrTable;
 FROM SYSTEM IMPORT
 (* Procedure *)  ADR, TSIZE;
 FROM FileSystem IMPORT
 (* Type *)       File, Response,
 (* Procedure *)  Lookup, SetWrite, WriteNBytes, Close;

  PROCEDURE InitArcs;
  CONST
   RET         = 0;   CommDecl  = 14;   RobotDecl = 18;   TaskDecl  = 32;
   Assignment = 46;   RelDir    = 50;   Command   = 51;   Guard     = 70;
   BTerm       = 72;  Factor    = 74;

   WITH Arcs[1]   DO Next := 2;        Term := TRUE; Trigger := ProgramSym END;
   WITH Arcs[2]   DO Next := 3;        Term := TRUE; Trigger := IdSym      END;
   WITH Arcs[3]   DO Next := 4;        Term := TRUE; Trigger := SemicolOn  END;
   WITH Arcs[4]   DO Next := 5;        Term := FALSE; SubNet := CommDecl   END;
   WITH Arcs[5]   DO Next := 6;        Term := FALSE; SubNet := RobotDecl  END;
   WITH Arcs[6]   DO Next := 7;        Term := FALSE; SubNet := TaskDecl   END;
   WITH Arcs[7]   DO Next := 8;        Term := TRUE; Trigger := IsSym      END;
   WITH Arcs[8]   DO Next := 9;        Term := TRUE; Trigger := IdSym      END;
   WITH Arcs[9]   DO Next := 10;       Term := TRUE; Trigger := AssignSym  END;
   WITH Arcs[10]  DO Next := 11;       Term := FALSE; SubNet := Assignment END;
          ....
```

```
    WITH Arcs[128] DO Next := 76;       Term := TRUE; Trigger := IdSym      END;
    WITH Arcs[129] DO Next := 77;       Term := TRUE; Trigger := OnSym      END;
    WITH Arcs[130] DO Next := RET;      Term := TRUE; Trigger := IdSym      END;
    WITH Arcs[131] DO Next := 79;       Term := FALSE; SubNet := Guard      END;
    WITH Arcs[132] DO Next := RET;      Term := TRUE; Trigger := ClosePar   END;
    WITH Arcs[133] DO Next := RET;      Term := TRUE; Trigger := IdSym      END;
    WITH Arcs[134] DO Next := RET;      Term := FALSE; SubNet := RelDir     END
  END InitArcs;

  PROCEDURE InitArcBegin;
  BEGIN
    ArcBegin[0]  := 1;   ArcBegin[1]  := 2;   ArcBegin[2]  := 3;
    ArcBegin[3]  := 4;   ArcBegin[4]  := 5;   ArcBegin[5]  := 6;
          ....

    ArcBegin[75] := 129; ArcBegin[76] := 130; ArcBegin[77] := 131;
    ArcBegin[78] := 132; ArcBegin[79] := 133; ArcBegin[80] := 135
  END InitArcBegin;

  PROCEDURE InitCodePerState;
  BEGIN
    ErrMsg[1]  := 0;  ErrMsg[2]  := 9 ; ErrMsg[3]  := 10; ErrMsg[4]  := 11;
    ErrMsg[5]  := 12; ErrMsg[6]  := 13; ErrMsg[7]  := 33; ErrMsg[8]  := 9 ;
              ....

    ErrMsg[73] := 45; ErrMsg[74] := 40; ErrMsg[75] := 9 ; ErrMsg[76] := 21;
    ErrMsg[77] := 9 ; ErrMsg[78] := 40; ErrMsg[79] := 44; ErrMsg[80] := 7
  END InitCodePerState;

 VAR f            : File;
     k,n          : CARDINAL;
     Arcs         : ArcTable;
     ArcBegin     : StartTable;
     ErrMsg       : StateErrTable;

BEGIN
 InitArcs;
 InitArcBegin;
 InitCodePerState;
 Lookup(f,"LPCTable.DAT",TRUE);
 IF f.res # done THEN HALT END;
 SetWrite(f);
 k := TSIZE(ArcTable);
 WriteNBytes(f,ADR(Arcs),k,n);
 IF (f.res # done) OR (k # n) THEN HALT END;
 k := TSIZE(StartTable);
 WriteNBytes(f,ADR(ArcBegin),k,n);
 IF (f.res # done) OR (k # n) THEN HALT END;
 k := TSIZE(StateErrTable);
 WriteNBytes(f,ADR(ErrMsg),k,n);
 IF (f.res # done) OR (k # n) THEN HALT END;
 Close(f)
END BuildTables.
```

8. Conclusiones

Se puede establecer un balance de logros y fallas en tres distintos niveles:

- *a nivel del sistema obtenido;*
- *a nivel de la metodología empleada;*
- *a nivel del cumplimiento de los objetivos.*

Las conclusiones se elaboran para cada uno de estos niveles y se presentan en las subsiguientes secciones.

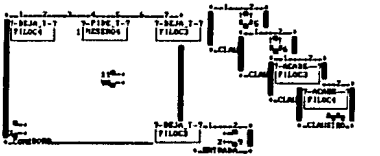
8.1 El sistema obtenido

Aunque se pretendía diseñar un sistema híbrido de modelo de sistemas de programación y de lenguaje de programación, se obtuvo sólo lo segundo. Esto se debe a que los objetos primitivos electos para *LPC* (mundos, robots, muros y trompos) no son lo suficientemente abstractos como para soportar un modelo general. Si se hubieran escogido como elementos primarios a los procesos de *CCS* [Milner80], se habría obtenido un sistema como *Clara* [Giacalone88]. *Clara* es un ambiente de especificación formal de sistemas concurrentes que a la vez permite la programación efectiva de los sistemas diseñados, la simulación de los mismos y su verificación automática; todo ello en forma gráfica.

Sin embargo, el haber escogido los mundos y los robots como objetos básicos brindó algunas ideas interesantes que podrían ser fuentes de futuras investigaciones: por ejemplo, el cuestionar la utilidad de la dicotomía agente-tarea en el concepto de proceso podría ser interesante en el ámbito de los sistemas auténticamente distribuidos, sobre todo a nivel del sistema operativo.

El lenguaje de programación de *LPC* resultó sencillo y expresivo. Permite solucionar problemas de sincronización y de exclusión mutua empleando prácticamente toda la gama de mecanismos que ofrece la programación concurrente, hasta los que se basan en memoria compartida. Como muestra de

ello, se exhiben dos soluciones al problema de los cinco filósofos [Dijkstra72] transportado a *LPC*.



```
PROGRAMA Filosofos;                    ( Primera versión )

          ( Basada en memoria compartida )

ROBOT: Filo1 COMPARTE ComedorA EN 1,4 NORTE BABOR CON 0,
       Filo2 COMPARTE ComedorA EN 4,7 NORTE PROA CON 0,
       Filo3 COMPARTE ComedorA EN 7,4 NORTE ESTRIBOR CON 0,
       Filo4 COMPARTE ComedorA EN 4,1 NORTE POPA CON 0;

TAREA Pensar ES
 CICLO
  ESCOGE
   CIERTO -> nada>
  | CIERTO -> nada>
  | CIERTO -> ROMPE>
   FIN (escoge)
 FIN (ciclo)
FIN Pensar;

TAREA CogeTenedor(d : DIRREL) ES
 Gira d; Avanza; Avanza;
 ESCOGE Trompo proa -> Recoge> NINGUNA -> nada> FIN;
 Gira popa; Avanza; Avanza; Gira d
FIN CogeTenedor;

TAREA DejaTenedor(d : DIRREL) ES
 Gira d; Avanza; Avanza; Deja;
 Gira popa; Avanza; Avanza; Gira d
FIN DejaTenedor;

TAREA CogeTenedores(d1,d2: DIRREL) ES
 YO := CogeTenedor(d1);
 ESCOGE
  Cargado -> Deja>; YO := CogeTenedor(d2);
             ESCOGE
              Cargado -> Recoge>
              NINGUNA -> Recoge>; YO := DejaTenedor(d1)
             FIN (escoge)
  NINGUNA -> nada>
 FIN (escoge)
FIN CogeTenedores;
```

```
TAREA Comer ES
 CICLO
  ESCOGE
   NO Cargado -> YO := CogeTenedores(babor,estribor)>
  | NO Cargado -> YO := CogeTenedores(estribor,babor)>
   NINGUNA    -> ROMPE>
  FIN (escoge)
 FIN; (ciclo)
 Avanza; Avanza;
 CICLO
  ESCOGE
    Trompo Proa     -> RECOGE>; ROMPE |
    Trompo Proa     -> RECOGE> |
    NO Trompo Proa  -> ROMPE>
  FIN (escoge)
 FIN; (ciclo)
 Gira Popa; Avanza; Avanza; Gira Popa;
 ESCOGE
  CIERTO -> YO := DejaTenedor(babor)>; YO := DejaTenedor(estribor)
| CIERTO -> YO := DejaTenedor(estribor)>; YO := DejaTenedor(babor)
 FIN (escoge)
FIN Comer;

TAREA Filosofar ES
 CICLO
  ESCOGE
   CIERTO -> YO := Comer>;
             ESCOGE                    { Deja lo que comiste }
              Cargado -> Gira popa>;
                         CICLO
                          ESCOGE Cargado -> Deja> NINGUNA -> ROMPE> FIN
                         FIN;
                         Gira popa
              NINGUNA -> ROMPE>
             FIN (escoge)
  | CIERTO -> YO := Pensar>                  {Piensa}
  FIN (escoge)
 FIN (ciclo)
FIN Filosofar;

ES
 Filo1 := Filosofar;
 Filo2 := Filosofar;
 Filo3 := Filosofar;
 Filo4 := Filosofar
FIN Filosofos.


PROGRAMA Filosofos;              { Sexta versión }

          { Basada en envío de mensajes y creación dinámica }

 CANAL: Pide_T, Deja_T, Acabe;
 BUZON: Entrar, Salir, Servir;

 ROBOT: Mesero1 COMPARTE ComedorB EN 1,1 NORTE POPA CON 1,
        Mesero2 COMPARTE ComedorB EN 1,7 NORTE BABOR CON 1,
        Mesero3 COMPARTE ComedorB EN 7,7 NORTE PROA CON 1,
        Mesero4 COMPARTE ComedorB EN 7,1 NORTE ESTRIBOR CON 1,
        Filo1 HABITA Claustro EN 1,1 NORTE PROA CON 0,
        Filo2 HABITA Claustro EN 1,1 NORTE BABOR CON 0,
        Filo3 HABITA Claustro EN 1,1 NORTE POPA CON 0,
        Filo4 HABITA Claustro EN 1,1 NORTE POPA CON 0,
        FiloC1 COMPARTE ComedorB EN 1,4 NORTE BABOR CON 0,
        FiloC2 COMPARTE ComedorB EN 4,7 NORTE PROA CON 0,
        FiloC3 COMPARTE ComedorB EN 7,4 NORTE ESTRIBOR CON 0,
        FiloC4 COMPARTE ComedorB EN 4,1 NORTE POPA CON 0,
        Guardia HABITA Entrada EN 2,1 NORTE ESTRIBOR CON 3;
```

```
TAREA Pensar ES
 CICLO
  ESCOGE
   CIERTO -> nada>
  | CIERTO -> nada>
  | CIERTO -> ROMPE>
  FIN {escoge}
 FIN {ciclo}
FIN Pensar;

 TAREA Cocinar(Creador: ROBOT) ES
  ESCOGE
   Robot Proa O Robot Popa -> Gira Babor>  {Colocate para no estorbar}
   NINGUNA                 -> NADA>
  FIN; {escoge}
  Avanza; Gira Popa;
  CICLO                                    {Rellena el plato}
   ESCOGE
    Cargado -> DEJA>
    NINGUNA -> ROMPE>
   FIN {escoge}
  FIN; {ciclo}
  Creador!Acabe                            {Revive a tu creador}
 FIN Cocinar;

 TAREA Comer(Creador, Izquierda, Derecha: ROBOT)
  ROBOT: Cocinero COMPARTE Comedor8 EN 4,4 NORTE PROA CON 20;
 ES
  Izquierda!Pide_T; Derecha!Pide_T;  {Pide ambos tenedores}
  Izquierda?Pide_T; Derecha?Pide_T;  {Espera que estén listos}
  Gira Babor; RECOGE; Gira Popa; RECOGE;       {Recoge ambos tenedores}
  Gira Babor; Avanza; Avanza;
  CICLO
   ESCOGE
    Trompo Proa                {Come si hay comida y salte}
     -> RECOGE>; ROMPE
   | Trompo Proa                {Come si hay comida y sigue comiendo}
     -> RECOGE>
   | NO Trompo Proa Y NO REMITE Comer EN Servir
     -> Comer!Servir>;          {Avisa que vas a servir}
        Cocinero := Cocinar(YO); {Solicita más comida}
        Cocinero?Acabe;         {Espera a que haya servido}
        Comer?Servir;           {Avisa que ya serviste}
        RECOGE;                 {Come}
        ESCOGE
         CIERTO -> ROMPE>       {A lo mejor estás satisfecho}
        | CIERTO -> NADA>
        FIN {escoge}
     NINGUNA -> NADA>
   FIN {escoge}
  FIN; {ciclo}
  Gira Popa; Avanza; Avanza;
  Gira Estribor; DEJA; Gira Popa; DEJA;        {Deja ambos tenedores}
  Gira Estribor;
  CICLO
   ESCOGE cargado -> deja> NINGUNA -> ROMPE> FIN   {Deja lo que comiste}
  FIN; {ciclo}
  Izquierda!Deja_T; Derecha!Deja_T;          {Avisa que están libres}
  Creador!Acabe                              {Revive a tu creador}
 FIN Comer;
```

```
TAREA Filosofo(VAR Hambriento :ROBOT; Izquierda, Derecha: ROBOT;
                                          GuardaComedor: TAREA) ES
 CICLO
  ESCOGE
   CIERTO -> GuardaComedor!Entrar>;
             GuardaComedor?Entrar;                  {Pide paso al comedor}
             Hambriento := Comer(YO,Izquierda,Derecha);    {Come}
             Hambriento?Acabe;                     {Espera hasta que termine}
             GuardaComedor!Salir                    {Avisa salida del comedor}
  | CIERTO -> YO := Pensar>                         {Piensa}
  FIN (escoge)
 FIN (ciclo)
FIN Filosofo;

TAREA Mesero(Frente,Otro : ROBOT; DirDelOtro: DirRel) ES
 CICLO
  ESCOGE
   REMITE Frente EN Pide_T
    -> Frente?Pide_T>;
       Avanza; DEJA; Gira popa; Avanza; Gira popa;
       Frente!Pide_T;
       Frente?Deja_T;
       Avanza; RECOGE; Gira popa; Avanza; Gira popa
  | REMITE Otro EN Pide_T
    -> Otro?Pide_T>; Gira DirDelOtro;
       Avanza; DEJA; Gira popa; Avanza; Gira popa;
       Otro!Pide_T;
       Otro?Deja_T;
       Avanza; RECOGE; Gira popa; Avanza; Gira popa;
       Gira DirDelOtro; Gira DirDelOtro; Gira DirDelOtro
   NINGUNA -> NADA>
  FIN (escoge)
 FIN (ciclo)
FIN Mesero;

TAREA Guardar(Comensales: TAREA) ES
 CICLO
  ESCOGE
   REMITE Comensales EN Entrar Y Cargado
    -> DEJA>; Comensales?Entrar; Comensales!Entrar
  | REMITE Comensales EN Salir
    -> Comensales?Salir>; RECOGE
  | NO Cargado
    -> Comensales?Salir>; RECOGE
   NINGUNA -> NADA>
  FIN (escoge)
 FIN (ciclo)
FIN Guardar;

ES
 Mesero1 := Mesero(FiloC1,FiloC4,babor);
 Mesero2 := Mesero(FiloC2,FiloC1,babor);
 Mesero3 := Mesero(FiloC3,FiloC2,babor);
 Mesero4 := Mesero(FiloC4,FiloC3,babor);
 Filo1 := Filosofo(FiloC1,Mesero1,Mesero2,Guardar);
 Filo2 := Filosofo(FiloC2,Mesero2,Mesero3,Guardar);
 Filo3 := Filosofo(FiloC3,Mesero3,Mesero4,Guardar);
 Filo4 := Filosofo(FiloC4,Mesero4,Mesero1,Guardar);
 Guardia := Guardar(Filosofo)
FIN Filosofos.
```

8.2 La metodología

El impacto que tuvo la metodología empleada es palpable en todo el trabajo. La definición del lenguaje fue gradual, desde conceptos primitivos hasta una sintaxis concreta. Se pudieron expresar, con relativa sencillez y elegancia, los mecanismos que proporcionan la mayoría de los lenguajes de programación concurrentes modernos como son: creación dinámica de procesos, comandos custodiados, envío de mensajes asíncronos etc.

De la construcción formal del universo semántico se derivó, en forma sumamente natural, una sintaxis para el lenguaje. Esta manera de proceder facilitó la expresión de su semántica.

La realización del sistema se llevó a cabo en aproximadamente cinco semanas y con un número de errores iniciales sumamente bajo tanto en número (del orden de diez) como en magnitud[1]. Además, como se indicó al presentar la figura 1 del capítulo siete, la definición de los objetos de *LPC* como tipos de datos abstractos delimitó naturalmente los módulos de definición del sistema, conservando en éstos una forma prácticamente idéntica a la original. Las relaciones inter-módulos se mantuvieron simples gracias a esta edificación gradual y formal, como se puede apreciar en la mencionada figura, permitiendo así verificar los componentes del sistema aisladamente.

8.3 El cumplimiento de los objetivos

Aún no se puede determinar si *LPC* cumplirá con el objetivo de facilitar el aprendizaje de la programación concurrente ya que no se ha probado su utilidad con alumnos. Existe la posibilidad de que se utilice el laboratorio en el curso de programación concurrente que están preparando en forma conjunta la Dra. Hanna. Oktaba del I.I.M.A.S. y el Dr. Mario Albarrán de la Facultad de Ciencias, ambos de la U.N.A.M. Esta experimentación arrojará sin duda luces sobre el éxito o fracaso de *LPC* como sistema didáctico de apoyo.

---

[1] Se está hablando exclusivamente de los módulos específicos del sistema; no se considera el tiempo de implementación de las bibliotecas. La realización del sistema de pantallas consumió alrededor de ocho meses y la biblioteca del compilador cerca de dos meses.

Consultando las revistas especializadas *COMPUTER* e *IEEE Transactions on Software Engineering* de los últimos meses (Enero a la fecha), se observa un aprecio creciente por los sistemas que cuentan con capacidades de animación. Tómense como ejemplos al sistema *Balsa-II* [Brown88] de análisis de algoritmos mediante animaciones de distintas representaciones gráficas de los mismos, al sistema *Clam* mencionado en la sección 8.1 y al artículo sobre arquitecturas de ambientes de programación [Young88] en el que se hace hincapié sobre la importancia de la observación directa de ciertos aspectos de los programas y se menciona un sistema de edición y animación para *Redes de Petri* [Morgan85]. El auge observado en esta área permite ser optimista en cuanto al éxito de *LPC*.

En cuanto a las características consideradas como deseables para *LPC* en el capítulo inicial, se puede concluir lo siguiente:

- *grandes posibilidades de experimentación*:

  como se indicó en las secciones anteriores, el lenguaje de *LPC* permite experimentar tanto con las notaciones presentes en la mayoría de los lenguajes de programación concurrente modernos, como con una amplia gama de mecanismos de sincronización y de exclusión mutua;

- *simplicidad conceptual*:

  aunque los mundos, los robots, los muros y los trompos no son objetos matemáticos suficientemente abstractos para que *LPC* sea un modelo de sistemas de programación, si proporcionan una gran sencillez conceptual ya que resulta sumamente intuitivo programar robots en un plano cartesiano;

- *expresividad*:

  los datos manipulados por los programas de *LPC* son relativamente pobres, tanto en tipos como en estructuras. Sin embargo, no impiden programar algoritmos relativamente complejos y transcendentes ya que el significado de los programas no se definió como una función que transforma datos de entrada en datos de salida, sino como un conjunto de estados que conaservan cierta relación;

- *gran capacidad para ejemplificar*:

  esta característica se heredó directamente de la metáfora de los robots que sirven para caricaturizar situaciones reales, abstrayendo condiciones irrelevantes y trabajando con sumamente concretos y cercanos;

☞ *herramientas formales de apoyo*:

la semántica formal de *LPC* no satisface este requerimiento ya que no es lo suficientemente abstracta. Para suplir esta deficiencia se tendría que desarrollar un cálculo sobre los estados de *LPC*, o bien, se podría establecer un isomorfismo entre *LPC* y un sistema como *CCS* que proporciona en sí un cálculo con el que se pueden establecer y verificar propiedades de los programas;

☞ *facilidad de uso*:

la realización actual cumple bastante bien con este requisito, aunque se recomiendo usar el sistema en máquinas rápidas para obtener animaciones que realmente den la ilusión de la concurrencia.

Bibliografía

[Andrews 81] Andrews, G.R., "Synchronizing Resources", ACM Trans. Program. Lang. Syst., vol.3, no.4, (Oct.81), pp.405-430.

[Andrews 83] Andrews, G.R., F.B. Schneider, "Concepts and Notations for Concurrent Programming", Compt. Surv., vol.15, no.1, (Mar.83), pp.3-43.

[Apt 80] Apt, K.R., N. Francez, W.P. de Roever, "A Proof System for Communicating Sequential Processes", ACMTOPLAS, vol.2, no.3, (Jul.80), pp.359-385.

[B.Hansen 72] Brinch Hansen, P., "Structured Multiprogramming", Comm. ACM, vol.15, no.7, (Jul.72), pp.574.

[B.Hansen 75] Brinch Hansen, P., "The Programming Language Concurrent Pascal", IEEETrans. Softw. Eng., vol.SE-1, no.2, (Jun.75), pp.199-207.

[B.Hansen 78] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept", Comm. ACM, vol.21, no.11, (Nov.78), pp.934-941.

[B.Hansen 87] Brinch Hansen, P., "Joyce: A Programming Language for Distributed Systems" Software-Practice and Experience, vol.17, no.1, (Ene.87), pp.29-50.

[Bajar 86] Bajar, V.R., Talavera, J.C., "Iniciación a la Computación por medio de Karel", Segunda Conferencia Internacional Las Computadoras en Instituciones de Educación, México D.F., (Dic.86), pp.35-43.

[Benveniste 84] Benveniste, M.V., Karel Concurrente, Comunicaciones técnicas, serie azul, no.82, (Ago.84), IIMAS-UNAM.

[Benveniste 88] Benveniste, M.V.,"LPC: A Concurrent Programming Laboratory" en Proceedings of the 5th Annual STACS, editado por R.Cori y M.Wirsing, Lecture Notes in Computer Science, vol.294, Springer-Verlag, Bordeaux1988, pp.391-392.

[Ben-Ari 82] Ben-Ari, M., Principles of Concurrent Programming, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1982.

[Berber 86] Berber, R., Pascal-S Concurrente, Comunicaciones Técnicas, serie azul, no.95, (Jun.86), IIMAS-UNAM.

[Brown 88] Brown, M.H., "Exploring Algorithms Using Balsa-II", Computer, vol.21, no.5, (May.88), pp.14-36.

[Dennis 74] Dennis, J.B., "First Version of a Data Flow Procedure Language", en Proceedings, Colloque sur la Programation, editado por B. Robinet, Lecture Notes in Computer Science, vol.19, Springer-Verlag, Berlin, 1974, pp.362-376.

[Dijkstra 68] Dijkstra, E.W., "Co-operating Sequential Processes", en Programming Languages: NATO Advanced Study Institute editado por F. Genuys, Academic Press, London, 1968.

[Dijkstra 72] Dijkstra, E.W., "Hierarchical Ordering of Sequential Processes", en Operating Systems Techniques, editado por C.A.R. Hoare y R.H. Perrot, Academic Press, New York, 1972, pp.72-93.

[Dijkstra 75] Dijkstra, E.W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs", Comm. ACM, vol.18, no.8 (Ago.75), pp.453-457.

[DoD 80] Department of Defense, "Military Standard Ada Programming Language", Report MIL-STD-1815, Naval Publications and Forms Center, Philadelphia, Pennsylvania, (Dic.80).

[Feldman 79] Feldman, J.A., "High Level Programming for Distributed Computing", Comm. ACM, vol.22, no.6, (Jun.79), pp.353-368.

[Filman 84] Filman, R.E., D.P. Friedman, Coordinated Computing: Tools and Techniques for Distributed Software, Computer Science Series, McGraw-Hill, 1984.

[Friedman 79] Friedman, D.P., D.S. Wise, "An Approach to Fair Applicative Multiprogramming", en Semantics of Concurrent Computation, editado por G. Kahn, Lecture Notes in Computer Science, vol.70, Springer-Verlag, New York, 1979, pp.203-225.

[Giacalone 88] Giacalone, A., S.A. Smolka, "Integrated Environments for Well-Founded Design and Simulation of Concurrent Systems", IEEE Trans. Softw. Eng., vol.SE-14, no.6, (Jun.88), pp.787-802.

[Hewitt 77] Hewitt, C.E., "Viewing Control Structures as Pattern of Passing Messages", Artif. Intell., vol.8, no.3, (Jun.77), pp.323-364.

[Hoare 72] Hoare, C.A.R., "Towards a theory of parallel programming", en Operating Systems Techniques, editado por C.A.R. Hoare y R.H. Perrot, Academic Press, NewYork, 1972, pp.61-71.

[Hoare 78] Hoare, C.A.R., "Communicating Sequential Processes", Comm. ACM, vol.21, no.8, (Ago.78), pp.666-677.

[Hoare 83] Hoare, C.A.R., "Notes On Communicating Sequential Processes", Technical Monograph PRG-33, Oxford, (Ago.83).

[INMOS 84] INMOS Limited, Occam Programming Manual, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Jazaveri 80] Jazaveri, M, et al., "CSP/80: A Language for Communicating Sequential Processes", IEEE Compcon Fall 80, pp.736-740.

[Kain 72] Kain, R.Y., Automata Theory: Machines and Languages, Computer Science Series, McGraw-Hill, 1972, pp.188-191.

[Knuth 68] Knuth, D.E., "Semantics of Context-free Languages", Mathematical Systems Theory, 2 (1968), pp.127-145. Correción en Mathematical Systems Theory, 5(1971), pp.95.

[Lapalme 87] Lapalme, G., P. Chartray, "An Educational System for the Study of Tasking in Ada", IEEE Transactions on education, vol.E-30, no.3 (Ago.87).

[Levin 81] Levin, G.M., D. Gries, "A Proof Technique for Communicating Sequential Processes", Acta Informatica15, pp.281-302, 1981.

[LOGITECH 86] LOGITECH, Inc., Modula-2/86 User's Manual, 3ra. edición, (Mar.86), Redwood City, CA.

[Lynch 81] Lynch, N.A., M.J. Fischer, "On Describing the Behavior and Implementation of Distributed Systems", Theoret. Comp. Sci., vol.13, no.1, (1981),pp.14-43.

[McCarthy 65] McCarthy, J., P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.I. Levin, Lisp 1.5 Programmer's Manual, M.I.T. Press, Cambridge, Massachusetts, 1965.

[Milner 80] Milner, R., A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol.92, Springer-Verlag, Berlin 1980.

[Misra 81] Misra, J., K.M. Chandy, "Proofs of Networks of Processes", IEEE Trans. Softw. Eng., vol.SE-7, no.4, (Jul.81).

[Morgan 85] Morgan, E.T., R.R. Razouk, "Computer-aided Analysis of Concurrent Systems", en Proceedings 5th International Workshop Protocol Specification, Verification and Testing, Toulouse, Francia, (Jun.85).

[Oktaba 85] Oktaba, H., Programación Concurrente, Comunicaciones técnicas, serie azul, no.86, (Jul.85), IIMAS-UNAM.

[Oktaba 87] Oktaba, H., Programación Concurrente, Comunicaciones técnicas, serie azul, no.100, (Jun.87), IIMAS-UNAM.

[Pagan 81] Pagan, F., Formal Specification of Programming Languages, a Panoramic Primer, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Papert 80] Papert, S., Mindstorms: Children, Computers & Powerful Ideas, Basic Books Inc., 1980. Edición en español: Desafío a la Mente, Computadoras y Educación, Ediciones Galápago, 1981.

[Pattis 81] Pattis, R.E., Karel the Robot: A Gentle Introduction to the Art of Programming, John Wiley & Sons Inc., NewYork, 1981. Edición en español: El Robot Karel, Introducción gradual a la Programación, Limusa, México, 1985.

[Peterson 77] Peterson, J.L., "Petri Nets", Comput. Surv., vol.9, no.3, (Sep.77), pp.223-252.

[Petri 62] Petri, C.A., "Kommunikation mit Automaten", Ph.D. dissertation, University of Bonn, Bonn (1962).

[Rising 84] Rising, L., "A Syntax-Directed Editor, World-Builder and Simulator for the Language of Karel The Robot", SIGPLAN Notices, vol.19, no.11, (Nov.84), pp.18-21.

[Roper 81] Roper, T.J., Barter, C.J., "A Communicating Sequential Process Language and Implementation", Software-Practice and Experience, 11, pp. 1215-1234, (1981).

[Silberschatz 80] Silberschatz, A., "Cell: A Distributed Computing Modularization Concept", Technical Report 155, Department of Computer Science, The University of Texas, Austin, Texas, (Sep.80). También en IEEE Trans. Softw. Eng., (Mar.84).

[Steele 78] Steele, G.L., Jr., and G.J. Sussman, "The Revised Report on SCHEME, a Dialect of Lisp", Memo 452, Artificial Intelligence Laboratory, M.I.T., Cambridge, Massachusetts, (Ene.78).

[Sufrin 86] Sufrin, B., "Z Handbook", Draft1.1, Oxford University Computing Laboratory, Programming Research Group, (Mar.86).

[Wand 80] Wand, M., Induction, Recursion, and Programming, Elsevier North Holland, Inc., Amsterdam, 1980.

[Wirth 81] Wirth, N., Pascal-The Language and its implementation, editado por D.W.Barron, John Wiley & Sons, New York, 1981.

[Wirth 85] Wirth, N., Programming in Modula-2, 3ra. edición corregida, Springer-Verlag, New York, 1985.

[Young 88] Young, M., R.N. Taylor, D.B. Troup, "Software Environment Architectures and User Interfase Facilities", IEEE Trans. Softw. Eng., vol.SE-14, no.6, (Jun.88), pp.697-708.

Indice de definiciones