# UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

## UN LENGUAJE PARA PROGRAMACION CONCURRENTE BASADO EN CSP

T E S I S

QUE PARA OBTENER EL TITULO DE:

M A T E M A T I C O

P R E S E N T A :

LUIS ALFONSO GONZALEZ LUNA



SEPTIEMBRE, 1990


**UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso**

A mis padres y a mi hermano.

A Mamá Conchita.

A Mamá Refugio.

*Hodie labor cras fructus*

## AGRADECIMIENTOS

Deseo agradecer a mi director de tesis, Dr. Mario Albarrán, su apoyo y guía durante el desarrollo de la tesis; a mis sinodales: el Mat. Salvador López, Dr. Víctor Germán Sánchez, Mat. Laura Espitia y Mat. Guadalupe Ibargüengoitia, su ayuda y revisiones, sin las cuales este trabajo no sería posible. Asimismo, agradezco profundamente a la Dra. Hanna Oktaba y a la M. en C. Amparo López sus sugerencias y críticas al trabajo escrito.

A Mónica Leñero, José Antonio López y Alfonso González M., mil gracias por su apoyo, compañerismo, sugerencias y comentarios al trabajo desarrollado.

# INDICE

# INTRODUCCION

Este escrito resume el proyecto de desarrollo de un lenguaje de programación apropiado para ambientes distribuidos, basado en el lenguaje de programación Edison, y fue dirigido por el Dr. Mario Albarrán. El trabajo contó con la valiosa colaboración de los miembros del Grupo de Sistemas (GS) del Departamento de Matemáticas de la Facultad de Ciencias, U.N.A.M.

Es clara la influencia que ha tenido la programación concurrente en la filosofía de la programación y el diseño de nuevas tecnologías. Los estudios realizados han logrado una muy buena comprensión del tema en el caso de memoria centralizada, bajo la filosofía de mantener ocupado el mayor tiempo posible al procesador de la computadora.

Sin embargo, la disponibilidad de nuevas tecnologías de diseño de circuitos integrados (que favorecen la construcción de máquinas con varios, hasta miles de procesadores contenidos en una unidad), y el abatimiento del costo de adquisición de microcomputadoras (aunado al desarrollo de las redes de computadoras), han hecho posible configurar, a bajo costo, un sistema de cómputo donde un programa se forme por varios procesos, los cuales se ejecutan simultáneamente en las máquinas del sistema. Esta situación ha desembocado en el diseño de los llamados *sistemas distribuidos*, y su programación se conoce como *programación distribuida*. Es interesante contar con lenguajes de programación que auxilien en la implantación y diseño de tales sistemas.

El trabajo que se presenta es la evaluación y experiencia del diseño e implantación de un lenguaje de programación distribuida, elaborado a partir de un lenguaje de programación centralizada. El punto de partida fué el estudio del lenguaje de programación Edison ([Brinch Hansen-1981]), debido a la experiencia que el GS tiene con él en enseñanza de programación concurrente y diseño de sistemas ([Albarrán-1988]). En este estudio, se realizó un análisis de su diseño, filosofía y mecanismos de implantación.

La siguiente etapa fue el análisis del modelo de concurrencia *Communicating Sequential Processes* (*CSP*) de Hoare ([Hoare-1978]), un modelo orientado a la programación distribuida, y se estudió la posibilidad de integrarlo como nuevo modelo de concurrencia a Edison de una manera coherente. El resultado de esta etapa es el diseño del lenguaje E', que se presenta en este trabajo.

Este escrito se organiza de la siguiente manera:

En el capítulo I, se presentan las nociones fundamentales de la programación concurrente, con el fin de obtener una base común para las discusiones que se desarrollan a continuación.

El capítulo II presenta un resumen de las características más importantes de CSP, justificando el por qué de su elección como modelo de concurrencia para un lenguaje de programación distribuida. Se discuten también las colaboraciones que han enriquecido a dicho modelo.

El capítulo III es la presentación del lenguaje E'; se hace una descripción del lenguaje Edison, para justificar su elección como base de E' y para comprender las decisiones de diseño que dieron forma al nuevo lenguaje.

El capítulo IV detalla la implantación del ambiente actual de programación en E', en un medio centralizado. Se habla del funcionamiento de su compilador, de las rutinas de la biblioteca de apoyo (un kernel multiproceso y rutinas para comunicación CSP), y se concluye considerando la forma de implantar E' en un medio netamente distribuido.

Luis Alfonso González Luna
Ciudad Universitaria, Septiembre de 1990

# CAPITULO I : Programación concurrente.

## 1.1 PRESENTACION

En esta sección se revisan los conceptos fundamentales de la programación concurrente, mostrando las ventajas de esta última con respecto a la programación secuencial, así como los problemas únicos que presenta al programador. Se introducen también las soluciones que se han desarrollado en la programación concurrente tradicional, y que se han incorporado en los lenguajes de programación concurrente (LPC's).

## 1.2 PROCESOS

El concepto fundamental para entender la organización de computadoras es el "proceso". Es éste una entidad activa, capaz de efectuar cambios en su medio ambiente, y su estado está determinado en todo instante por los siguientes elementos:

(1) El programa: la lista de pasos que la computadora ha de ejecutar (secuencialmente) para llevar a cabo las acciones del proceso

(2) Los valores de las variables y datos que utiliza el programa (incluyendo los valores de los registros de la unidad de procesamiento, tales como acumulador, registro de estado y contador de programa).

Es importante resaltar que en un conjunto de procesos, puede haber dos ó más que compartan el mismo código. En este caso, los procesos se distinguen uno de otro por los valores de sus variables y punto de ejecución, los cuales pueden ser diferentes y

otorgan individualidad a cada proceso. Ya en implantación física, los procesos pueden compartir la zona de memoria donde se almacena el código, pero las zonas de memoria donde se almacenan sus variables deben ser disjuntas, si no van a compartir información.

En un lenguaje secuencial (como Pascal o C), el programador describe un solo proceso, que constituye el programa completo. Este enfoque es útil mientras no se consideren problemas cuyo planteamiento implique la especificación de una colección de acciones simultáneas o concurrentes, pues la misma naturaleza secuencial de un proceso impide describir dicha situación.

¿Cómo expresar entonces una solución cuando el programa deba tratar con una colección de eventos concurrentes? Esta situación suele presentarse en problemas tales como control de dispositivos y escritura de sistemas operativos, y la solución radica en la especificación de la ejecución simultánea de varios procesos, cada uno de los cuales puede dedicarse a atender una acción específica. De esta manera, el programador logra una descripción natural del problema a la máquina, y el lenguaje de programación empleado se encarga de los detalles de implantación. Es a la especificación de programas de esta forma lo que se conoce como "programación concurrente" (ya que la ejecución de los procesos constitutivos del programa "concurre", o sucede simultáneamente).

Con el fin de comparar la programación secuencial con la concurrente, considérese el siguiente ejemplo:

Supóngase un conjunto de 'n' dispositivos llamado DIS, cuyos elementos (DIS(1), ..., DIS(n)) efectúan trabajos que se les ordenan. Cuando algún $a \in$ DIS concreta su encargo, solicita la atención de la computadora que monitorea el conjunto; al detectar ésta la solicitud, atiende a a y le encarga nuevo trabajo.

Para mostrar soluciones secuenciales y concurrentes a este problema se usará el lenguaje Edison, que es muy parecido a Pascal e incorpora construcciones para realizar programación concurrente.

En las soluciones secuenciales no se emplean las construcciones de concurrencia.

La solución secuencial puede escribirse de la siguiente forma (la palabra reservada PROC equivale al PROCEDURE de Pascal):

```
CONST   N = .                  " Número de dispositivos a atender "
PROC monitoreo
BEGIN
  WHILE true DO
     IF Atender(1) DO          " Si está listo DISC1) .. "
        ...                    " ..ejecutar estas acciones "
     END;
     IF Atender(2) DO          " Si está listo DISC2) .. "
        ...                    " ..atenderlo "
     END;
     IF ..                     " Lo mismo para los demás "
        ...
     END
  END
END
```

No se puede describir la naturaleza simultánea del monitoreo, pues es necesario detallar a la computadora las acciones a tomar de una manera estrictamente secuencial. El ejemplo anterior se puede contrastar con la descripción con varios procesos, empleando las construcciones de concurrencia disponibles en Edison :

```
CONST   N = ..
PROC monitoreo
  PROC Dispositivo1            " Código para atender al DISC1) "
  BEGIN
     WHILE true DO
       WHILE NOT Atender(1) DO      " Esperar al DISC1) "
         SKIP
       END;
       ..                    " Atención al DISC1) "
     END
  END
..
  PROC DispositivoN            " Código para atender al DISCN) "
  BEGIN
     WHILE true DO
       WHILE NOT Atender(N) DO      " Esperar al DISCN) "
         SKIP
       END;
       ..                    " Atención a DISCN) "
     END
  END
```

(continúa)

```
BEGIN                    " Programa principal "
  COBEGIN 1 DO Dispositivo1
     ALSO 2 DO Dispositivo2
     ALSO ..
     ALSO N DO DispositivoN
  END
END
```

Cabe hacer las siguientes observaciones con respecto a esta solución:

- La proposición COBEGIN indica la ejecución simultánea de una colección de listas de proposiciones, cada una de las cuales describe un proceso (en la solución, los procesos son las llamadas a los procedimientos).

- La idea de proceso permite una mejor estructuración del código, así como una mejor descripción del problema a la máquina. De hecho, existen lenguajes que utilizan la idea de 'proceso' como la única construcción para la estructuración del código.

Queda por resolver un pequeño problema: la implantación física de los procesos, por parte del lenguaje y el hardware. Las alternativas principales son:

(1) Dedicar una computadora (o procesador) a la ejecución de cada proceso (lo cual no siempre es posible); o bien,

(2) Implantar un sistema de máquinas virtuales en la computadora (llamado "multiplexaje" del procesador). En este caso, el código y datos de cada proceso reciben un lugar específico en la memoria de la computadora, y se ejecuta uno de los procesos hasta que ocurre un evento (término de un intervalo de tiempo, señal de un dispositivo ó una acción específica del proceso) que dictamina la ejecución de un proceso diferente.

En este punto, se realiza un "cambio de contexto" hacia el proceso que se va a ejecutar a continuación, guardando el estado de los registros del procesador en el proceso anterior (para reiniciar su ejecución en un instante

posterior), y leyendo nuevos valores de los registros para el siguiente proceso.

Bajo el esquema de multiplexaje, la concurrencia sólo se da a nivel del lenguaje de programación, ya que en todo momento la computadora ejecuta una sola instrucción de lenguaje de máquina.

La segunda alternativa de implantación de concurrencia es más económica y es más empleada, pues una elección cuidadosa de los procesos permite aprovechar al máximo el procesador. Sin embargo, cuando los eventos requieran atención inmediata, y ocurran de manera muy frecuente, el multiplexaje del procesador resulta inconveniente, y hay que recurrir a la primera alternativa: asignar cada proceso a un procesador diferente.

### 1.2.1 Construcciones de concurrencia

La forma de especificar la concurrencia es también importante, pues influye en la claridad y estructuración del programa, como cualquier construcción del lenguaje.

La concurrencia se suele describir mediante la ejecución inicial de un solo proceso, que en un momento determinado crea a otros procesos, especificando el código y condiciones iniciales para ellos. El proceso inicial se conoce como *padre*, y los procesos creados son sus *hijos*. El esquema puede repetirse indefinidamente: los hijos pueden crear también nuevos procesos.

Si el lenguaje maneja el concepto de ambiente global (variables y procedimientos globales, accesibles a todo ente definido en cierto nivel), los procesos hijos tienen acceso al mismo ambiente global que su proceso padre (lo cual implica que pueden compartir variables entre sí).

En los ejemplos ya se ha utilizado la construcción COBEGIN, y

existen otras igualmente importantes, entre ellas:

**FORK** : Funciona como una "bifurcación": el nuevo proceso se ejecutará concurrentemente con su padre, quien usó el FORK; con la proposición JOIN, el padre espera a que concluya la ejecución de su hijo. Por ejemplo,

```
PROC Q1
BEGIN
    ..
    FORK Q2(..);    " Q2 se ejecutará en un hijo "
    ..;             " Acciones de Q1 concurrentes con Q2 "
    JOIN            " Q1 espera a que termine Q2 "
    ..              " Q1 reanuda su ejecución "
END
```

especifica la ejecución simultánea del procedimiento 'Q2' (con algunos parámetros) y del resto del proceso que llamó a 'Q1'.

**PARBEGIN**: ([Dijkstra-1968]) Cuando un proceso encuentra esta proposición queda suspendido, y las sentencias especificadas dentro del 'parbegin' se ejecutan de manera concurrente; cada proposición especifica un nuevo proceso. El proceso original sólo reinicia su operación cuando todos los procesos que ha creado terminan.

Estas son las formas básicas de especificación de la concurrencia, y se les suele calificar de la siguiente forma: el **FORK** es más flexible (pero menos estructurado), mientras el **PARBEGIN** es estructurado y facilita la verificación del programa.

Se pueden hallar variantes de las proposiciones **FORK-JOIN** en la biblioteca del compilador de C del S.O. UNIX (donde se le conoce como **FORK-WAIT**) y en el lenguaje de programación *Joyce* (donde sintácticamente no existen estas construcciones, pero la activación de procesos tiene toda su semántica). La construcción **COBEGIN** de Edison es muy parecida al **PARBEGIN**.

## 1.3 Comunicación y sincronización entre procesos

En el diseño de programas concurrentes es frecuente hallar el caso de dos (o más) procesos que necesitan intercambiar información, pues las partes de un programa suelen interactuar. Esto constituye el problema de la comunicación de procesos, y un lenguaje de programación concurrente (LPC) debe proveer algún mecanismo que lo solucione.

En esta sección se consideran los problemas de sincronización y comunicación en un medio donde los procesos accesan a una memoria común, como en máquinas multiprocesadores (donde los procesadores comparten la memoria), o donde se usa multiplexaje para implantar los procesos. El caso de arquitecturas donde los procesos se encuentran *distribuidos* se considera en el capítulo siguiente.

### 1.3.1 Comunicación entre procesos

En medios de memoria común, la comunicación se realiza mediante la compartición de variables, y la sincronización ha sido ampliamente estudiada y comprendida. Un caso típico es el siguiente:

```
PROC Ejemplo
VAR  dato, x, y: int
BEGIN
     dato:= .. ;              " Generar un valor para 'dato' "
     COBEGIN 1 DO x:= dato; display(x)   " Manda a pantalla "
        ALSO 2 DO y:= dato; print(y)     " Manda a impresora "
     END
END
```

Los dos procesos aprovechan el ambiente global del que disponen para leer información que necesitan; ambos usan una copia de la variable (aunque en el caso de lectura de variables compartidas esto no es necesario).

Se denota como "región crítica" de un proceso a una parte del código de éste, donde se accesa un recurso compartido (en este caso, la variable 'dato'). Tal nombre se debe a la posibilidad de conflicto entre procesos, si éstos se encuentran ejecutando

regiones críticas donde acceson al mismo recurso. Considérese otro ejemplo:

```
PROC Ejemplo
VAR  dato: char
BEGIN
     COBEGIN   1 DO dato:= . . ;          " Generar valor "
          ALSO 2 DO display(dato)         " Mandar a pantalla "
     END
END
```

En este caso, se pretende calcular el valor para la variable 'dato' simultáneamente con la escritura del valor de esta última. Un poco de meditación muestra que el programa puede realizar acciones incorrectas: ¿cómo se puede garantizar que 'dato' tiene un valor correcto cuando se le accesa en el segundo proceso? Aquí existe un problema de sincronización, pues los procesos pueden hallarse en sus regiones críticas simultáneamente, siendo que se necesita que el primer proceso actualice a 'dato' antes que el segundo use la variable. Un LPC debe proveer un mecanismo que realice esta sincronización de los procesos.

### 1.3.2. Sincronización de procesos

En el caso del multiplexaje del procesador, el problema de sincronización se puede resolver deshabilitando los cambios de contexto, aunque esta solución tiene el inconveniente de evitar la ejecución de todos los procesos, incluso de aquellos que no necesitaban sincronización. Es claro que se requiere una mejor solución.

El primer concepto desarrollado para sincronización de procesos en general fueron las banderas (*flags*), que son variables compartidas por los procesos y mediante las cuales se notifican mutuamente su estado de ejecución. Asociando una bandera a cada recurso compartido, los procesos pueden saber cuándo un recurso se encuentra libre. Un proceso que empieza a ejecutar una región crítica debe verificar el estado de una bandera, y al terminar cambiar su valor.

Considérese el siguiente ejemplo, donde un proceso *productor*

entrega números enteros a un proceso *consumidor*, quien los imprime; los números se almacenan en una variable global a ambos procesos. La sincronización se realiza mediante una bandera, que les permite averiguar cuándo es seguro accesar la variable de comunicación. Además, es necesaria otra variable que les permita saber cuándo el *productor* ha almacenado un nuevo valor, y cuándo el *consumidor* ha utilizado ese valor.

```
ENUM bandera( libre, ocupado)        " Def. del tipo bandera "

PROC Ejemplo
VAR  dato: char;
     válido: bool;
     b: bandera

  PROC Productor
  BEGIN
     WHILE true DO
       WHILE (b=ocupado) AND válido DO
          SKIP        " Espera a que se pueda hacer algo "
       END;
       b:= ocupado;          " Empieza la región crítica "
       dato:= ..;            " Producir un valor "
       válido:= true;        " Avisa que es válido "
       b:= libre             " Fin de la región crítica "
  END

  PROC Consumidor
  BEGIN
     WHILE true DO
       WHILE (b=ocupado) AND NOT válido DO
          SKIP        " Espera a que se pueda hacer algo "
       END;
       b:= ocupado;          " Empieza la región crítica "
       display(dato);        " Utiliza el valor "
       válido:= false;       " Avisa que ya se usó el dato "
       b:= libre             " Fin de la región crítica "
     END
  END

BEGIN
  válido:= false;
  b:= libre;
     COBEGIN   1 DO Productor
           ALSO 2 DO Consumidor
     END
END
```

Las banderas proveen una solución muy poco estructurada, y el programador debe cuidar en todo momento el uso que haga de ellas.

Además, si no se garantiza que la modificación de las banderas se hace *atómicamente* (sin que otro proceso interfiera), el problema de sincronización persiste. En el caso de multiplexaje de procesos, ocurre además el problema de *espera ocupada*: al esperar entrar a una región crítica, un proceso consume tiempo de procesador, que otros procesos podrían aprovechar.

Las banderas quedan como el recurso a emplear si el LPC no provee mejores medios de sincronización. Aunque existen soluciones completas al problema de sincronización que emplean banderas únicamente, como el algoritmo de Dekker (presentado en [Dijkstra-1968]), es preferible que el LPC administre todos los detalles mediante alguna otra construcción.

Los semáforos ([Dijkstra-1968]) proveen una solución más estructurada y clara al problema de la sincronización; dados:

α un conjunto de proposiciones,
Q un conjunto de procesos cuyos códigos llaman al procedimiento que contiene α
M un entero natural
$P(t) = \{ p \in Q : p \text{ ejecuta } \alpha \text{ en el instante } t \}$

un semáforo garantiza que en cualquier instante t, se cumple siempre $|P(t)| \leq M$. En particular, si M=1, se garantiza que sólo un proceso podrá ejecutar la región crítica.

Los semáforos forman un tipo de datos con tres operaciones ('init', 'wait' y 'signal'), y se les puede implantar mediante números enteros; las definiciones de sus operaciones, en términos de procedimientos de Edison, son:

```
PROC init(num_procesos: int; VAR semáforo: int)
BEGIN                          " Inicialización del semáforo "
    semáforo:= num_procesos
END
```

*(continúa)*

```
PROC wait(VAR semáforo: int)
" El proceso espera a entrar a la región si está 'llena' "
BEGIN
     WHILE semáforo=0 DO
       SKIP
     END;
     semáforo:= semáforo - 1
END

PROC signal(VAR semáforo: int)
" El proceso que lo ejecuta avisa que sale de la región"
BEGIN
     semáforo:= semáforo + 1
END
```

'wait' y 'signal' deben aparecer al principio y final de la región crítica que se desea proteger. Nuevamente, los semáforos son solución al problema de sincronización solamente si se garantiza que las operaciones 'wait' y 'signal' son atómicas. El ejemplo anterior, resuelto con semáforos, ofrece el siguiente aspecto:

```
PROC Ejemplo
VAR  dato: char;     " variable compartida de comunicación "
     válido: bool;   " señal de datos disponibles "
     s: int          " semáforo para coordinación "

     PROC Consumidor
     BEGIN
       WHILE true DO
         wait(s);          " Protege acceso a región crítica "
         IF válido DO      " Si hay un dato válido .. "
           display(dato);      " .. escribirlo.."
           válido:= false      " ..y avisar que ya se leyó  "
         END;
         signal(s)         " Indica fin de región crítica "
       END
     END

PROC Productor
BEGIN
  WHILE true DO
     wait(s);              " Inicio de región crítica "
     x:= ..;               " Generar un nuevo valor de 'x' "
     valido:= true;        " Avisar que hay datos disponibles "
     signal(s)             " Fin de región crítica "
  END
END
```

*(continúa)*

```
BEGIN      " Programa principal "
  init(1, s);              " Sólo un proceso podrá pasar "
  válido:= false;          " No hay datos generados "
  COBEGIN 1 DO Productor
     ALSO 2 DO Consumidor
  END
END
```

Los semáforos son muy empleados, pero tienen la desventaja de ser poco estructurados. Para solucionar esta cuestión, se introdujeron las llamadas "regiones críticas condicionales", entre las cuales se puede distinguir, como la más refinada, la proposición **WHEN** de Edison.

La proposición **WHEN** tiene como propósito custodiar la entrada a una región crítica mediante una expresión booleana. Si dicha expresión evalúa falso, el proceso queda detenido, para volver a evaluar la expresión posteriormente; si es verdadera, el proceso ingresa a la región crítica bajo exclusividad: será el único proceso ejecutándose dentro de la región. A continuación, se muestra el uso de **WHEN** para construir las operaciones 'wait' y 'signal' de semáforos:

```
PROC wait(VAR s: int)
BEGIN
  WHEN s>0 DO
     s:= s - 1
  END
END

PROC signal(VAR s: int)
BEGIN
  WHEN true DO
     s:= s + 1
  END
END
```

Estas no son todas las construcciones de sincronización que se han desarrollado y propuesto; se pueden destacar entre otras a los monitores ([Brinch Hansen-1972]; [Hoare-1974]), cuyo planteamiento comprende a las regiones críticas condicionales y al concepto de módulo.

Un monitor se define como un conjunto de procedimientos que

operan sobre una estructura de datos, la cual solamente ellos pueden accesar. Un proceso que desee accesar ó modificar la información de la estructura llama a ciertos procedimientos pertenecientes al monitor y definidos como "puntos de entrada"; al empezar a ejecutar un punto de entrada, al proceso se le garantiza ser el único ente activo "dentro" del monitor ("opera bajo exclusividad"). Un monitor se puede implementar mediante módulos y algún mecanismo de sincronización (regiones críticas condicionales, semáforos, etc.).

Cabe en este momento un comentario con respecto a los mecanismos de sincronización: cualquiera de ellos se puede utilizar para construir a los demás (por ejemplo, ya se ha empleado la proposición **WHEN** para construir las operaciones de semáforos). Al nivel más bajo, el mecanismo de sincronización que asegure atomicidad en operaciones, será:

(1) La deshabilitación del evento que ocasiona el cambio de contexto, en caso de multiplexaje; y

(2) Un mecanismo de hardware que garantice a un procesador acceso exclusivo a una localidad de memoria, en multiprocesadores.

## 1.4 Resumen

La concurrencia es una herramienta útil en el desarrollo y especificación de programas, que permite simplificar su estructura. Involucra nuevos problemas (especificación de procesos, comunicación y sincronización entre éstos) que se han atacado de manera muy diversa, y cuyas soluciones se han refinado con el paso del tiempo.

Sin embargo, las soluciones mencionadas hasta ahora hacen referencia solamente a los problemas de sincronización y

comunicación de variables cuando existe una memoria común, a la cual todos los procesos puedan accesar; las definiciones e implantaciones propuestas suponen que cada proceso es capaz de accesar ciertas variables en memoria (como los semáforos), y realizar ciertas operaciones sobre ellas.

En el caso de una máquina donde no exista memoria compartida, los procesos no tendrán más forma de comunicación que algún enlace físico (red de computadoras, un bus local de comunicación, etc.) que les permite enviar mensajes, y ya no es clara la implantación de los primitivos descritos en este capítulo (¿cómo pensar en tener variables que todos los procesos puedan accesar y modificar?). Es necesario además hallar alguna manera de efectuar la sincronización de los procesos. Este era un problema no bien comprendido hasta la aparición del modelo CSP (*Communicating Sequential Processes* - [Hoare-1978]) de concurrencia, el cual se discute en el siguiente capítulo.

# CAPITULO II: C. S. P.

## 2.1 PRESENTACION

Las soluciones a los problemas de sincronización y comunicación de procesos resultan relativamente sencillas cuando éstos cuentan con una memoria compartida. La comunicación se resuelve fácilmente usando variables globales a los procesos, y la sincronización se resuelve mediante alguno de los esquemas mencionados en el capítulo anterior (los cuales usan variables compartidas, para averiguar el estado del mecanismo de sincronización respecto de la ejecución de los procesos). Esta dependencia en la compartición de memoria hace poco factible la implantación de esas soluciones en un medio distribuido, pues los procesos no cuentan con más medio de comunicación que un enlace físico: los intentos de simular los mecanismos descritos en el capítulo anterior en un sistema distribuido (con el fin de ocultar al programador la distribución de los procesos) no son satisfactorios, pues la implantación se complica mucho y los programas no suelen utilizar de la mejor manera el hardware disponible. Es por ello que actualmente se considera como mejor solución el desarrollo de primitivas propias para la distribución.

Existen dos tendencias en las primitivas propuestas para el manejo de procesos distribuidos: el uso de mensajes (empleado desde que se empieza a programar sistemas distribuidos), y el llamado a procedimientos remotos (*Remote Procedure Call*) (basado en el trabajo [Brinch Hansen-1978]). Algunos autores clasifican al envío de mensajes como una solución de "bajo nivel" (pues refleja más fielmente la situación física), y a los procedimientos remotos como de "alto nivel". En este trabajo, se estudia el envío de mensajes como solución a la comunicación de los procesos distribuidos.

Hoare publica en 1978 un artículo fundamental ([Hoare-1978]) para el futuro desarrollo de formalizaciones y lenguajes para sistemas distribuidos. En dicho escrito, se supone la existencia

de un medio en el cual los procesos no comparten memoria y sólo se pueden comunicar mediante operaciones de entrada y salida de mensajes a través de un medio físico; con base en estas premisas se desarrollan un modelo y una notación (que se emplea como lenguaje), llamados *Communicating Sequential Processes* (CSP), y se muestra su utilidad para la programación, concluyendo con una serie de comentarios respecto a implantación y mejoras al modelo. Aunque Hoare hace énfasis en que se trata de una solución parcial, su trabajo es ampliamente aceptado y continuado por otras personas que hacen contribuciones teóricas (como [Apt-1980]) y prácticas ([Brinch Hansen-1987]; [Brinch Hansen-1989] entre otros).

## 2.2. El modelo original de Hoare

En CSP, los procesos son entes completamente independientes que ejecutan acciones sobre sus variables locales e interactúan con el medio ambiente por envío de mensajes; la recepción y envío de estos mensajes realiza simultáneamente tareas de sincronización y comunicación entre procesos. La especificación de la concurrencia se realiza mediante una variante del **PARBEGIN** de Dijkstra, el cual lista las proposiciones que se han de ejecutar simultáneamente como nuevos procesos; un ejemplo de su uso es

```
[A:: x:integer; x:= 65 || B:: y: integer; y:= 67 ]
```

Este ejemplo inicia la ejecución en paralelo de dos procesos, A y B, los cuales realizarían asignación de valores a sus variables locales. Es posible especificar también la ejecución de "arreglos de procesos", donde los procesos contienen copias de un mismo código, pero se diferencian por el subíndice de cada uno. En todo caso, la ejecución del comando concluye cuando todos los procesos participantes terminan.

La comunicación entre los procesos se realiza mediante intercambio de mensajes, empleando las dos nuevas operaciones primitivas que Hoare introduce en su lenguaje. Esta comunicación es totalmente síncrona (la operación suspende al proceso que la ejecuta, hasta que ha sido concluida), y además explícita: un

proceso no efectúa la recepción de ningún mensaje a menos que indique su disposición, efectuando el comando de entrada. Estas operaciones se describen en la siguiente sección.

### 2.2.1. Comandos de entrada y salida.

Las operaciones de entrada y salida de mensajes entre dos procesos son primitivas en CSP (al igual que la asignación de valores a variables), y permiten que éstos se comuniquen cuando uno de ellos nombra a otro como "proveedor de la entrada", y el otro nombra al primero como "destino de la salida":

```
[     A:: x: integer;
           B ? x       " El proceso B proveerá un valor para x "
||
      B::
           A ! 7       " El proceso A es el destino del valor "
]
```

El efecto neto de la comunicación en este ejemplo es la asignación del valor '7' a la variable 'x' de A. La sintaxis general es la siguiente:

```
ProposiciónDeEntrada:    <NombreDeProceso> '?' <Variable>
ProposiciónDeSalida:     <NombreDeProceso> '!' <Expresión>
```

La comunicación es simétrica, pues los procesos necesitan nombrarse mutuamente para llevarla a cabo. Además, tiene la característica de ser síncrona: si un proceso nombra a otro en un comando de entrada ó salida, quedará *suspendido* hasta el momento en que el otro proceso lo nombre a él en la operación opuesta, instante en el cual ambos continúan su ejecución. Dadas estas características, un proceso A que ejecuta una proposición de entrada ó salida, nombrando a un proceso B, puede fallar en las siguientes circunstancias:

1) El mensaje que B espera intercambiar con A no es del mismo tipo que la comunicación que A pretende realizar con B. Este caso es similar al fallo de la asignación cuando sus elementos no son de tipos compatibles entre sí.

2) B ha terminado su ejecución, por lo cual el comando jamás se concretará.

Mediante estas características de las proposiciones de entrada y salida se obtiene la sincronización y comunicación de procesos en una sola operación.

Los procesos también pueden intercambiar mensajes que no lleven contenido (llamados *señales*), y que sólo sirvan para sincronizar sus acciones. Más adelante se presenta un ejemplo completo de su uso.

2.2.2. Guardias y no determinismo

Al trabajar en un ambiente donde varios eventos ocurren a la vez, lo primero que resalta es la naturaleza no determinística de la situación; no es posible predecir en qué orden sucederán los eventos. Dijkstra ([Dijkstra-1975]) sugiere que la mejor forma de manejar dichas situaciones consiste en añadir proposiciones no deterministas a los lenguajes de programación, razón por la cual presenta los "comandos custodiados" (*Guarded Commands*).

Una "custodia" o "guardia" es una expresión booleana que precede a un conjunto de proposiciones; si la guardia evalúa verdadero, la lista de proposiciones es elegible para ejecución. De esta forma, se piensa que la expresión "custodia" a las proposiciones. Dados

$$G = \{ Gi \} \quad (1 \leq i \leq N),\ \text{el conjunto de guardias,}$$
$$LP = \{ LPj \} \quad (1 \leq j \leq N),\ \text{conjunto de listas de proposiciones asociadas a las guardias G}$$
$$GV = \{ g \in G : g \text{ es verdadera} \}$$

se elige un $Gk \in GV$ para ejecutar su LPk, de manera completamente no determinística. Este proceso de elección puede realizarse una sola vez (formando la proposición condicional no determinística) o bien mientras GV no sea vacío (proposición iterativa no determinística).

Hoare adopta los comandos custodiados de Dijkstra con un cambio de notación, e introduce una modificación relacionada con CSP: además de expresiones booleanas, las guardias pueden incluir la evaluación de la posibilidad de concretar un comando de entrada. En ese caso, una custodia evalúa verdadero si su expresión booleana es verdadera, y si existe un comando de salida suspendido que corresponda con el comando de entrada de la guardia. De todas las guardias que evalúen verdadero, se elige una de manera no determinística y se completa el comando de entrada que especifica. Mediante comandos custodiados, un proceso puede elegir una de varias alternativas de comunicación, lo que proporciona mayor flexibilidad al lenguaje.

La proposición condicional de CSP enlista los comandos custodiados, separándolos por el símbolo []; su apariencia es la siguiente:

```
[       <expr. booleana>; <proposición de entrada>   ->
          <lista de proposiciones>
  [] <expr. booleana>; <proposición de entrada>   ->
          <lista de proposiciones>
  [] ... más custodias
]
```

Si todas las custodias de la proposición condicional "fallan" (evalúan falso), la proposición condicional también falla (lo que puede causar el término anormal del programa).

La proposición iterativa denota la ejecución "tantas veces como sea posible" de una proposición condicional. Mientras existan guardias que evalúen verdadero, se repetirá la proposición condicional. Si no hay guardias verdaderas, pero algunas tienen expresiones booleanas verdaderas y los procesos que mencionan en comandos de entrada no han terminado, el proceso se suspende antes de repetir la proposición condicional. Una proposición iterativa termina cuando *todas* las guardias fallen, y esta condición no constituye un error; esta propiedad resulta muy importante.

La proposición iterativa tiene el siguiente aspecto:

```
*[      <expr. booleana>; <proposición de entrada>      ->
          <lista de proposiciones>
     [] <expr. booleana>; <proposición de entrada>      ->
          <lista de proposiciones>
     [] ... más custodias
]
```

2.2.3. Un ejemplo: Un cronómetro y un usuario.

A continuación se muestra un ejemplo de programación en CSP, con el fin de observar algunos aspectos interesantes y peculiaridades del lenguaje.

Un proceso "usuario" solicita un servicio de medición de tiempo a un proceso "reloj", con el fin de cronometrar sus acciones. El programa puede quedar escrito de la siguiente manera:

```
RELOJ=
     inicio, final: integer;

     ..;       " Código de inicialización del reloj "
     *[true  ->
          usuario? empezar();        " Aguarda la SEÑAL de inicio "
          inicio:= ..;               " Leer el reloj físico "
          usuario? terminar();       " Espera la SEÑAL de fin "
          final:= .. ;               " Leer otra vez el reloj "
          usuario! final-inicio;     " Envía cuenta de pulsos "
     ]

USUARIO=
     pulsos: integer;

     *[true  ->
          ..;                        " Algún código de inicialización "
          reloj! empezar();          " Pide cronómetro "
          ..;                        " Proposiciones a cronometrar "
          reloj! terminar();         " Detener el cronómetro "
          reloj? pulsos;             " Leer tiempo transcurrido "
          ..;                        " Hacer algo con el resultado "
     ]

[ reloj:: RELOJ  || usuario:: USUARIO ]        " Prog. principal "
```

Pueden notarse algunas de las características importantes de la programación en CSP:

(1) Se favorece una programación encapsulada: el proceso RELOJ se maneja como en un ente abstracto, que entiende

operaciones ó mensajes del tipo "empezar", "terminar" y que entrega resultados. Desde el punto de vista de usuario, no se necesita saber más que eso.

(2) Para asegurar el uso bajo exclusividad de algún recurso, basta escribir un proceso que lo administre y establecer un protocolo de empleo que sea respetado por los procesos usuarios. En el ejemplo, dicho protocolo consiste del envío de la señal "empezar" para iniciar la cuenta, "terminar" para detenerla y la lectura del resultado. Si hubiera algún otro proceso que deseara usar el reloj, 'usuario' tendría garantizado el acceso exclusivo al reloj una vez que ha ejecutado la proposición 'reloj! empezar()', pues en ese instante el reloj cambia de estado; si otro proceso solicitara servicio de cronómetro, quedaría detenido hasta que 'reloj' volviera a esperar una señal 'empezar()' como entrada.

(3) Por otra parte, el mecanismo de nombrar a los procesos dificulta la escritura de procesos "servidores", ya que debe conocerse de antemano el proceso con el cual se realiza la comunicación. Si existen dos usuarios para el proceso reloj, este puede reescribirse mediante custodias, de la siguiente manera:

```
RELOJ=
     inicio, final: integer;

     ..;          " Código de inicialización del reloj "
     *[ usuario1? empezar() ->        " Si U1 quiere el reloj .."
          inicio:= ..;                " Leer reloj físico "
          usuario1? terminar();       " Espera la SEÑAL de fin "
          final:= .. ;                " Leer pulso actual "
          usuario1! final-inicio;     " Envía cuenta de pulsos "

      [] usuario2? empezar() ->       " Si U2 quiere el reloj.."
          inicio:= ..;                " Leer reloj físico "
          usuario2? terminar();       " Espera la SEÑAL de fin "
          final:= .. ;                " Leer pulso actual "
          usuario2! final-inicio;     " Envía cuenta de pulsos "
     ]
```

Otra solución posible consiste en el uso de arreglos de

procesos para denotar a los usuarios del reloj:

```
RELOJ=
     inicio, final: integer;

     ..;          " Código de inicialización del reloj "
     *[ (i:1..10) usuario(i)? empezar() ->    "Si Ui quiere reloj"
          inicio:= ..;
          usuario(i)? terminar();
          final:= .. ;
          usuario(i)! final-inicio;
     ]

USUARIO=          ..          " Descripción idéntica a la primera "

[ reloj:: RELOJ   ||   usuario(i:1..10) :: USUARIO ]
```

## 2.3. Otras aportaciones.

Hoare hace énfasis en que CSP es sólo una solución parcial, que necesita mayor investigación y contribuciones antes de ser aceptada. Acepta haber realizado varias restricciones en el modelo, con el fin de concentrarse en los problemas fundamentales, pero al mismo tiempo señala direcciones para futuras investigaciones en CSP. Entre estas sugerencias, Hoare resalta la idea de "puertos" para realizar la comunicación entre procesos y la incorporación de comandos de salida en las guardias.

### 2.3.1. Puertos y vocabularios.

Como alternativa al nombramiento explícito de los procesos (que como se ha visto dificulta la escritura de procesos "servidores"), Hoare menciona los puertos como mecanismos que permitan entablar la comunicación entre procesos y realizar verificaciones en tiempo de compilación.

En un esquema de puertos, se considera que la comunicación entre los procesos tiene lugar a través de un *canal*. Los extremos del canal son los *puertos*, los cuales son conocidos por los procesos. Para comunicarse por un canal, un proceso necesita tener

un puerto de ese canal.

En una implantación de CSP con puertos, los procesos nombran al puerto como fuente ó destino de la operación, en vez de nombrar al proceso con el cual entablarán la comunicación. El puerto preserva el anonimato de los procesos que intervienen en las operaciones, pero permite que éstos entablen contacto y puede encargarse de la realización de los comandos de entrada y salida (pues conoce los procesos cuyos comandos pueden corresponder). El concepto de canal puede expresar físicamente el medio de comunicación entre los procesos (como en el lenguaje Occam, cuyo modelo de concurrencia es CSP, donde los puertos se mapean a los canales físicos de comunicación en el hardware de las *Transputers*) ([Fountain-1989]; [Sánchez-1990]).

Junto con los puertos aparece la idea del "vocabulario": el conjunto de tipos de datos de la información que se desea intercambiar por un canal. Un vocabulario consiste de una lista de nombres de mensajes, cada uno de éstos seguido del tipo de datos del contenido del mensaje; si un mensaje no indica contenido, se le trata como a las señales de CSP. Un ejemplo típico de declaración de un puerto y su contenido es la declaración de los tipos de puertos en el lenguaje de programación *Joyce* ([Brinch Hansen-1987.1]):

```
type
     flujo = [ open, read(char), close ]
```

La definición anterior en Joyce introduce un nuevo tipo de datos compuesto, del tipo puerto, cuyo nombre es 'flujo'; los mensajes 'open', 'read' y 'close' definen el vocabulario que se podrá transmitir por los puertos que pertenezcan a dicho tipo de datos. 'open' y 'close' son señales, y 'read' lleva un contenido de tipo caracter.

En Joyce, los procesos que van a comunicarse necesitan tener preparados los puertos que van a utilizar, antes de su activación. De esto se encarga un proceso padre, el cual declara los puertos

necesarios para la comunicación y activa a los procesos que se comunican, pasándoles como parámetros los puertos que necesiten.

El problema de procesos usuarios de un reloj puede describirse de la siguiente manera en Joyce:

```
agent programa
  const   N = 10;          " Número de procesos usuarios "
  type
     reloj_com = ( empezar, terminar, lectura(integer) );

     agent usuario(i: integer; p: reloj_com)
       var ...             " Vars. locales de usuarios "
     begin
       ...                 " Código de usuarios "
     end

     agent reloj(p: reloj_com)
       var inicio, final: integer;
     begin
       ..;      " Código de inicialización del reloj "
       while true do
       begin
          p? empezar;      " Espera a que alguien pida reloj "
          inicio:= ..;     " Leer valor actual "
          p? terminar;     " Esperar a que nos detengan "
          final:= .. ;     " Leer valor del reloj "
          usuario(i) ! final-inicio;
       end
     end        " *** Fin agente reloj *** "


   var
      per: reloj_com; " Puerto para comunicación con el reloj"
        i: integer;

begin                  " Programa principal "
     +per;                  " Inicializar el puerto "
     reloj(per);            " Activar proceso 'reloj' "
     " Activar los N usuarios: "
     for i:= 1 to N do
          usuario(i, per);      " Activar usuario i "
end  " Fin del programa, espera al término de los usuarios "
```

Cabe comentar que Joyce toma un enfoque radical en cuanto a la estructuración de código: la única unidad de estructuración son los "procedimientos agentes" (declaración agent), de forma tal que lo que aparece como llamada a procedimiento en otros lenguajes es una activación de procesos.

La estructuración de código mediante procesos permite enfatizar

el encapsulamiento de datos, pues permite pensar en las estructuras de datos como entes abstractos con los que existe comunicación, en vez de manipularlos directamente. Basta definir un agente, y declarar dentro de él la estructura de datos por encapsular. Cuando se desee una nueva instancia de la estructura, se activa una instancia del agente, (en vez de declarar una variable) y se establece comunicación con él mediante un puerto que defina mediante mensajes las operaciones válidas sobre la estructura.

2.3.2. Salidas en las guardias.

En CSP, un proceso puede elegir de manera no determinista al proceso que le suministrará una entrada, utilizando los comandos custodiados. ¿Por qué no permitir también a un proceso elegir a quién enviará un dato, de manera no determinista? Hoare argumenta a favor y en contra de esta propuesta, pues es interesante desde el punto de vista del modelo. En este trabajo, se considerarán los problemas relacionados con la implantación.

Al ejecutar un comando custodiado en CSP (donde sólo se permiten comandos de entrada en las custodias), se deben realizar los siguientes pasos:

(1) Evaluar las custodias, determinando cuáles de ellas evalúan verdadero; esto implica averiguar qué procesos hay detenidos en comandos de salida, y qué tipo de mensajes pretenden enviar.

(2) Elegir una custodia factible, y notificar al proceso nombrado en su proposición de entrada que la comunicación se concreta, de forma tal que éste pueda reanudar su ejecución.

En un medio ambiente distribuido, estos dos pasos se implantan mediante envío de mensajes por el canal físico, por parte de la biblioteca del compilador o sistema operativo; por ello es importante obtener un protocolo sencillo, con el menor número

posible de intercambio de mensajes.

Añadir proposiciones de salida en los comandos custodiados implica un protocolo más complejo, por las siguientes razones:

(a) Un proceso con comandos de salida en custodias necesita utilizar un algoritmo semejante al descrito para los procesos con comandos de entrada en custodias. En este algoritmo, el proceso con las guardias es el ente activo, quien "busca pareja".

(b) Custodias con comandos de salida pueden corresponder con custodias con comandos de entrada. O sea, ambos procesos intentarán "buscar pareja", y se hace necesario evitar confusiones.

El problema radica en el establecimiento de un protocolo y una definición de estados de procesos, que garanticen una comunicación cuando los correspondientes comandos de entrada y de salida se hallan en guardias. Bernstein ([Bernstein-1980]), Buckley y Silberschatz ([Buckley-1983]) y Bagrodia ([Bagrodia-1989]) describen protocolos que realizan este trabajo. Estos resultan complicados de implantar, y se pierde la sencillez de implantación de los comandos custodiados originales de CSP.

En un trabajo anterior al mencionado, Silberschatz ([Silberschatz-1979]) propone implantar custodias con comandos de salida, distinguiendo, en cada par de procesos (Pi, Pj) que se comunican, un amo y un esclavo. Un proceso amo es aquel que solicita servicios (se comporta como "cliente"), y un proceso esclavo es aquel que los proporciona (se comporta como "productor" o servidor). Los procesos esclavos, propone Silberschatz, podrán tener comandos de salida en sus custodias, y los esclavos no. De esta forma, el algoritmo de comunicación es muy parecido al algoritmo de comunicación que sólo permite custodias con comandos de entrada.

Brinch Hansen adopta esta idea en Joyce, al especificar que

comandos custodiados con comandos de entrada ó salida no pueden corresponder entre sí.

## 2.4. Resumen

En la programación de sistemas distribuidos, donde no existe una memoria centralizada, es necesario contar con mecanismos efectivos de sincronización y comunicación de los procesos. Las soluciones desarrolladas por la programación concurrente tradicional se han aplicado sin mucho éxito en este ambiente, por lo cual surgió la necesidad de desarrollar nuevos modelos específicamente para medios distribuidos.

Entre las soluciones desarrolladas se encuentra el modelo CSP de Hoare ([Hoare-1978]), que se basa en el intercambio de mensajes mediante proposiciones de entrada y salida. La sencillez de este modelo, y sus perspectivas como medio de prueba y verificación de programas lo han colocado como uno de los trabajos más influyentes en el campo. CSP se ha visto enriquecido por el trabajo de otros autores, y ha contribuido a entender la distribución de procesos, motivando el diseño de varios lenguajes de programación que lo adoptan como modelo de concurrencia.

En el próximo capítulo se presenta el lenguaje E', diseñado a partir del lenguaje Edison para realizar programación de sistemas en ambientes distribuidos. E' usa como modelo de concurrencia a CSP, lo que implica eliminar aspectos de Edison que no son compatibles con este modelo.

# CAPITULO III: El lenguaje E'

## 3.1 PRESENTACION

El lenguaje E' es un descendiente del lenguaje de programación Edison ([Brinch Hansen-1981]), del cual toma la mayor parte de su semántica y construcciones sintácticas, pero emplea un modelo de concurrencia diferente. Edison provee un entorno agradable y seguro para la programación de sistemas, y es ésta la razón por la cual se le eligió como base para un lenguaje de programación distribuida. La especificación de la concurrencia se basa en el modelo *Communicating Sequential Processes* (CSP) ([Hoare-1978]), y toma ideas de la implantación de CSP que hace el lenguaje Joyce ([Brinch Hansen-1987]).

De esta manera, E' proporciona un entorno para programación de sistemas en un ambiente de programación distribuida.

## 3.2. UN ANALISIS DE EDISON

El lenguaje de programación Edison ([Brinch Hansen-1981]) fue diseñado como una herramienta que permitiera la escritura segura y práctica de sistemas de tamaño mediano y aplicaciones en tiempo real, reemplazando a Pascal y Modula en este tipo de programas. Se trata de un lenguaje pequeño, fácil de implantar y elegante, que a partir de un análisis de Pascal elimina sus aspectos redundantes o inseguros, y añade conceptos que incrementan la seguridad de la programación. Edison realiza modificaciones a la sintaxis de Pascal que permiten una compilación más eficiente, e incorpora construcciones para la programación concurrente bajo un esquema centralizado: usa variables compartidas para comunicación, y regiones críticas para sincronización.

### 3.2.1. Variables, tipos de datos y control de flujo

Edison incorpora los tipos de datos que maneja Pascal: simples (int, char, bool) y estructurados (set, enum, array, record), pero se eliminan los apuntadores, con lo cual desaparecen ambigüedades del lenguaje y se refuerza la seguridad de la programación.

Se enfatiza la tipificación del lenguaje, forzando a toda variable a pertenecer a un tipo de datos definido con anterioridad. En Pascal esto es más un estilo del programador que una regla del lenguaje, lo cual puede crear problemas de interpretación; por ejemplo, en la mayoría de las implantaciones de Pascal, las siguientes dos variables son incompatibles: no se puede asignar una a la otra, a pesar de que tienen la misma estructura:

```
VAR
  a: Array [1..5] of Integer;
  b: Array [1..5] of Integer;
```

En Edison, la situación anterior no es posible; es necesario definir algún tipo de datos con arreglos:

```
ARRAY ArrEnteros [1:5] (int)   " Un nuevo tipo.. "

VAR
  a, b: ArrEnteros
```

Entre los conceptos novedosos en Edison se encuentra el constructor, que a partir de una lista de expresiones genera un objeto perteneciente a un tipo de datos determinado. Los constructores además evitan incluir en el lenguaje las funciones estándar de Pascal (*ord*, *chr*, *trunc*, etc). Con las declaraciones del ejemplo anterior, es posible inicializar la variable 'a' mediante un constructor del tipo de datos 'ArrEnteros':

```
a:= ArrEnteros(0,1,2,3,4);
```

Otra noción nueva es el enmascaramiento (*casting*): dados dos tipos de datos T1 y T2, cuyos elementos ocupan la misma cantidad de espacio en memoria, es posible "enmascarar" a un elemento (variable ó expresión) de T1 para que se le considere, durante un acceso (lectura ó escritura), como perteneciente al tipo T2. Este concepto da un escape a la fuerte tipificación cuando sea necesario (como ocurre frecuentemente en programación de sistemas operativos).

Las proposiciones de control de flujo secuencial de Edison son

solamente dos: condicional (if) e iterativa (while). Sin embargo, estas proposiciones son más flexibles que sus equivalentes de Pascal, pues usan una lista de expresiones booleanas y listas de proposiciones asociadas a cada expresión, de manera similar a los comandos custodiados descritos en el capítulo anterior; cuando una expresión resulte verdadera, se ejecutan las proposiciones que enmarca. La proposición de control no se ejecutará cuando todos los elementos de la lista evalúen falso. Por ejemplo,

```
if i=1 do
   ...
else i=2 do
   ...
else i=3 do
   ...
end
```

simula la proposición case de Pascal, y

```
while <cond1> do
   ...      " Proposiciones a ejecutar si 'cond1' es cierto "
else <cond2> do
   ...      " Proposiciones a ejecutar si 'cond2' es cierto "
end
```

iterará mientras 'cond1' y 'cond2' no sean simultáneamente falsas.

En cuanto a las proposiciones relacionadas con la concurrencia, se usa cobegin para denotar la ejecución concurrente de un conjunto de listas de proposiciones, como se explica en el capítulo I. La sincronización de procesos se realiza mediante la proposición when, una variante de regiones custodiadas que también permite incluir una lista de expresiones.

### 3.2.2. Estructuración de código en Edison

La unidad principal de estructuración de código en Edison sigue siendo el procedimiento, entendiendo a éste como un ente que realiza acciones sobre el ambiente que lo rodea. A diferencia de Pascal, en esta definición también entra el programa principal. El ambiente de cada procedimiento puede ser local ó global.

El ambiente local de los procedimientos está formado por los parámetros que recibe en el momento de su activación, así como por variables y procedimientos que el procedimiento declare, al igual

que en Pascal. Los parámetros pueden ser variables, constantes ó procedimientos, y en el caso de variables, se les puede pasar por valor o por referencia.

El ambiente global de los procedimientos se forma con las variables y procedimientos que rodean a cada procedimiento, los cuales pueden utilizarse sin declaración previa: el ambiente global es implícito. De esta manera, los procedimientos pueden accesar datos comunes a todos, permitiéndose la comunicación entre procesos mediante variables compartidas.

Es importante resaltar el hecho de que, a diferencia de Pascal, Edison no define procedimientos ni tipos de datos estándares para efectuar entrada y salida. Es responsabilidad del entorno que ejecuta el programa (o del programador) proveer estas rutinas como parte del ambiente. Los procedimientos y tipos de datos necesarios podrían escribirse en Edison mismo.

Considérese el siguiente programa:

```
PROC Ejemplo (
     PROC DisplayInt(i: int);
     alfa, beta: int
)

  PROC p1(z: int)
     VAR x: int

     PROC inicio
     BEGIN
       x:= 0
     END

  BEGIN
    ...    " Acciones del proc. p1 "
  END

BEGIN
  p1(10)
END
```

Como se puede observar, 'Ejemplo' (que es el programa principal) es un procedimiento que recibe tres parámetros del ambiente que lo ejecuta: un procedimiento que imprime enteros y dos variables enteras. El ambiente global de 'p1' consiste solamente de los

que en Pascal. Los parámetros pueden ser variables, constantes ó procedimientos, y en el caso de variables, se les puede pasar por valor o por referencia.

El ambiente global de los procedimientos se forma con las variables y procedimientos que rodean a cada procedimiento, los cuales pueden utilizarse sin declaración previa: el ambiente global es implícito. De esta manera, los procedimientos pueden accesar datos comunes a todos, permitiéndose la comunicación entre procesos mediante variables compartidas.

Es importante resaltar el hecho de que, a diferencia de Pascal, Edison no define procedimientos ni tipos de datos estándares para efectuar entrada y salida. Es responsabilidad del entorno que ejecuta el programa (o del programador) proveer estas rutinas como parte del ambiente. Los procedimientos y tipos de datos necesarios podrían escribirse en Edison mismo.

Considérese el siguiente programa:

```
PROC Ejemplo (
     PROC DisplayInt(i: int);
     alfa, beta: int
)

  PROC p1(z: int)
     VAR x: int

     PROC inicio
     BEGIN
       x:= 0
     END

  BEGIN
    ...    " Acciones del proc. p1 "
  END

BEGIN
  p1(10)
END
```

Como se puede observar, 'Ejemplo' (que es el programa principal) es un procedimiento que recibe tres parámetros del ambiente que lo ejecuta: un procedimiento que imprime enteros y dos variables enteras. El ambiente global de 'p1' consiste solamente de los

parámetros de 'Ejemplo', y su propia declaración de procedimiento (ya que se permite la recursión). En cambio, el ambiente global de 'inicio' se forma con el ambiente global de 'pi' más la variable 'x'.

Edison incluye también el concepto de módulo, el cual engloba en una sola estructura a un conjunto de datos, y una serie de operaciones definidas por procedimientos que actúan sobre dichos datos. Los valores de los datos pertenecientes al módulo permanecen constantes entre invocaciones a los procedimientos del módulo.

Un módulo define a un subconjunto de sus procedimientos como "exportables", lo cual incorpora a éstos al ambiente global donde se declara el módulo (a pesar de que las reglas de alcance no los incorporarían). Procedimientos ajenos al módulo no pueden accesar directamente los datos de éste (a no ser que específicamente se les permita hacerlo), pero pueden invocar a los procedimientos exportables para operar sobre ellos. De esta manera, se obtiene un acceso seguro y controlado a las estructuras de datos, al encapsular los datos y las operaciones que pueden operar sobre estos; el módulo crea un ambiente global restringido para los procedimientos que contiene.

Edison incluye otro concepto que lo hace apropiado para la escritura de sistemas operativos: los procedimientos de biblioteca. El lenguaje permite declarar procedimientos "externos", en el sentido de haber sido compilados con anterioridad y su código almacenado en algún dispositivo externo. Efectuar una llamada a un procedimiento de biblioteca implica (en tiempo de ejecución) localizar su código e incorporarlo a la memoria de programas para ejecución; al concluir esta, la memoria ocupada por el código se libera y queda disponible. Este esquema recuerda a la acción de un sistema operativo, el cual carga y ejecuta programas.

### 3.2.3. Requerimientos de hardware

Los conceptos de programación que un lenguaje incorpora determinan el tipo de máquinas en donde es más fácil implantarlo, así como el tamaño y complejidad de la biblioteca de tiempo de ejecución que requiere para sus programas.

En Edison se puede lograr una implantación sencilla y eficiente usando memoria compartida para implantar el ambiente global. A nivel del código generado por el compilador, los procesos pueden comunicarse y ejecutar los procedimientos que requieren fácilmente (bastan accesos a memoria y llamadas a subrutinas). Esto es cierto tanto en una máquina monoprocesador (donde los procesos se implantan mediante multiplexaje) como en un multiprocesador (donde los procesadores tienen acceso a una memoria común).

Considérese también el siguiente aspecto de la implantación del **cobegin** de Edison:

```
cobegin n1 do <lista de props. 1>
   also n2 do <lista de props. 2>
   also n3 do <lista de props. 3>
    ...
end
```

A cada proceso (lista de proposiciones) se asigna una constante que designa al procesador que ha de ejecutarlo. La asociación procesador-proceso es entonces dinámica, y es necesario que los procesadores compartan la memoria de programas para realizar esta asignación eficientemente, pues dos procesos pueden requerir el código de un mismo procedimiento durante su ejecución, para llamarlo.

## 3.3. Decisiones de diseño

Como se ha visto, Edison es un lenguaje apropiado para máquinas donde existe memoria compartida y se desea (o es necesario) estructurar los programas mediante procesos, pues incluso es posible aprovechar la presencia de varios procesadores.

Sin embargo, la tendencia actual en construcción de máquinas

con varios procesadores no es emplear una memoria común, pues los procesadores compiten entre sí por el acceso a la memoria. Se prefiere construir "multicomputadoras", máquinas formadas por varios procesadores, cada uno de los cuales tiene acceso a una memoria privada de datos y de programas. De preferencia, a cada procesador se le asigna la ejecución de un solo proceso (empleando multiplexaje cuando haya más procesos que procesadores), y la transferencia de información de uno a otro proceso se realiza por un medio físico que enlaza a los procesadores. Los sistemas distribuidos y las máquinas paralelas se pueden pensar como multicomputadoras, por lo cual es interesante contar con lenguajes que permitan especificar programas para ellas.

E' es un lenguaje de este corte; se le ha pensado para realizar programación de sistemas en un ambiente distribuido, para lo cual toma contribuciones de Edison, y usa CSP como único modelo de concurrencia.

Las definiciones de sintaxis en esta sección utilizan una notación BNF modificada, donde los símbolos terminales aparecen subrayados, los símbolos '[' y ']' denotan aparición opcional de los elementos que contienen, '{' y '}' denotan la repetición cero o más veces, y '|' denota opción de selección.

### 3.3.1. La herencia de CSP

Como modelo de concurrencia, E' adopta a CSP con los conceptos de puertos y vocabularios. La implantación de este modelo comprende:

1) La sintaxis original de CSP para los comandos de entrada y salida. La diferencia radica en que en vez de nombrar procesos, los comandos nombrarán puertos:

```
<PropEntradaSalida>: <PropEntrada> | <PropSalida>
<PropEntrada> :
   <Identificador> ? <Mensaje> [ , <Identificador> ]
<PropSalida> :
   <Identificador> ! <Mensaje> [ , <Identificador> ]
```

2) La introducción de los puertos como un nuevo tipo de datos compuesto. La definición de un tipo de datos puerto incluye el vocabulario que admite, y su sintaxis es compatible con la definición de tipos de Edison:

<Definición de puerto> :
port <identificador> ( <lista de mensajes> )
<Lista de mensajes> :
<Mensaje> { , <Mensaje> }

Un mensaje es un identificador y puede declarar a un tipo de datos como el tipo del contenido del mensaje. Si el mensaje es una señal no se especifica un tipo de datos:

<Mensaje> :
<identificador> [ ( <tipo de datos> ) ]

Los puertos de E* necesitan inicializarse; la sintaxis de esta inicialización es:

<IniPuerto> :
+ <Puerto>

3) La proposición no determinista con comandos custodiados de E*, llamada **WHEN**, es recuerdo de la proposición de sincronización de Edison. Esta proposición detiene al proceso que la ejecuta hasta que alguna de sus custodias tiene éxito. Se utiliza la palabra reservada **elsewhen** como separador de proposiciones custodiadas, para resaltar el no determinismo de la proposición.
Al emplear la proposición **when** con la proposición iterativa **while** de Edison, se implanta un comando iterativo.

La sintaxis de la proposición alternativa con custodias de E* es la siguiente:

<PropAlternativa> :

when <Custodia> do <ListaProps>

{ elsewhen <Custodia> do <ListaProps> }

end

Las custodias de E' permiten utilizar comandos de entrada ó de salida; al igual que en el lenguaje Joyce, comandos de salida en custodias no corresponden con comandos de entrada que también estén en custodias. Las custodias se evalúan de izquierda a derecha, y se forman de la siguiente manera:

<Custodia> :

<Expresión booleana> AND <PropEntradaSalida>

Las inclusiones de CSP que se han detallado bastan para proporcionar un ambiente sencillo, efectivo y fácil de implantar para la programación distribuida.

### 3.3.2. La herencia de Edison y decisiones de diseño

Edison es la influencia principal de E', al incorporar éste último la mayoría de las construcciones sintácticas y semánticas del lenguaje, así como la filosofía de programación. Sólo se omitieron aquellas construcciones que resultan inapropiadas, o que pueden crear puntos de conflicto con el modelo CSP de concurrencia.

En E' existen, sin modificaciones radicales, las siguientes construcciones de Edison:

1) Los tipos de datos simples (int, char, bool) y compuestos (array, record, set, enum). Sólo se añade el tipo de datos compuesto port (que se usa para comunicación CSP con puertos y vocabularios), y se propone la incorporación a futuro del tipo simple real.

| | SI | NO | OBSERVACIONES |
|---|---|---|---|
| 67.-LA PERSONA ENCARGADA DE LA - CAJA GENERAL TIENE ACCESO A - LOS REGISTROS CONTABLES? | X | | |
| 68.-LOS CHEQUES POSDATADOS SE --- GUARDAN EN FORMA SEGURA ANTES DE DEPOSITARSE (A FAVOR)? | X | | |
| 69.-LOS BIENES NEGOCIABLES DISTINTOS DEL EFECTIVO, CHEQUES O - LETRAS, SE ENCUENTRAN BAJO EL CUIDADO DE UN EMPLEADO DIFERENTE DE LAS PERSONAS DIRECTAMENTE RESPONSABLES DE LAS ENTRADAS DE CAJA? | X | | |
| 70.-TODAS LAS CUENTAS BANCARIAS - SE ENCUENTRAN AUTORIZADAS POR EL CONSEJO DE ADMINISTRACIÓN? | X | | |
| 71.-LOS CHEQUES INUTILIZADOS SE - CONSERVAN Y SE ARCHIVAN? | X | | |
| 72.-SE ANULAN DE TAL MANERA QUE | | | SE DESTRUYE LA FIRMA. |

2) Las declaraciones de variables y tipos de datos,los conceptos y sintaxis de enmascaramiento de tipos de datos y de constructores. Sin embargo, no es posible usar constructores con un puerto.

3) Las proposiciones de control secuencial. En cambio, en las proposiciones relacionadas con concurrencia, el cobegin varía ligeramente, y el caso del **when** ha sido descrito en la sección anterior.

Como unidad de estructuración de código se proponen los procedimientos y las ***unidades*** (estas últimas se describen más adelante, dentro de las características propias de E').

Los procedimientos en E' son estructuras de código cuya activación (llamado) causa la suspensión del proceso que los usa; cuando un procedimiento concluye el proceso que lo activó continúa; los procesos pueden ejecutar procedimientos, y los procedimientos pueden activar procesos. El propósito de la existencia de procedimientos en E' es proveer tanto un mecanismo familiar de estructuración de código, como un medio de extensión al lenguaje mediante operaciones definidas por el programador.

Entre los conceptos que desaparecen en E', pero que estaban presentes en Edison, se encuentran:

1) El concepto de ambiente global de los procesos, ya que proveería una forma de comunicación distinta a la entrada y salida de mensajes CSP (además de que su inclusión complicaría sobremanera la implantación del lenguaje en un ambiente distribuido). Debido a que un proceso puede llamar procedimientos, se elimina en E' el ambiente global de los procedimientos. La sintaxis de la declaración de variables locales a un procedimiento refleja este hecho (ver apéndice).

2) El paso de parámetros por referencia a procesos y procedimientos, por las mismas razones mencionadas en el párrafo anterior. También desaparece el paso de

procedimientos como parámetros.

3) Los procedimientos-funciones del lenguaje. Un llamado a función es una interacción de dos unidades de código que culmina cuando una de ellas (la función) emite un valor que la otra utiliza. Este esquema se puede implantar perfectamente mediante CSP.

4) El concepto de módulo, pues el encapsulamiento de datos se puede llevar a cabo mediante procesos y comandos CSP. Se puede usar un procedimiento que declare una estructura de datos, y efectúe operaciones sobre ella según los mensajes que reciba a través de un puerto. Cuando se requiera una instancia de la estructura de datos, se crea como proceso una activación del procedimiento.
Este esquema no es muy útil para implantar tipos de datos abstractos (como los módulos), pero resulta suficientemente flexible para los propósitos de E'.

En cuanto al cobegin, las modificaciones que experimenta reflejan la nueva situación:

1) Como no existe un ambiente global, no se pueden especificar listas de proposiciones como procesos, así que se restringen estos a activaciones de estructuras de código.

2) Desaparecen las constantes de asignación procesador a proceso, ya que ésta se realizará en forma estática por parte del programador (ya que es quien conoce las características de la máquina con que trabaja). A continuación se presenta la manera de realizar esa asignación.

En un sistema distribuido, es preferible realizar la asignación procesador-proceso de manera estática, pues se evitan conflictos en tiempo de ejecución. De aquí que la llamada a procedimientos en E' se restringe a los procedimientos cuyo código se halla en el mismo espacio de programas que el proceso que realiza la llamada.

La construcción *unidad* engloba a un conjunto de procedimientos, con el fin de asignarlos a ejecución por un mismo procesador. Su sintaxis es:

```
<Unidad> :
    unit <Identificador> [ ( <ListaParams> ) ]
    { <DeclTipos> } { <DeclProcedimiento> } { <DeclVars> }
    begin <ListaProps> end
```

Las unidades no se activan por llamada a procedimientos, sino que se les incluye en un cobegin, para convertirlas en procesos. Los procedimientos contenidos en una unidad sólo se conocen dentro de ella misma, y los parámetros y variables que declara sólo los puede utilizar el cuerpo principal de la unidad. Un programa en E' inicia su ejecución en una unidad llamada *main*.

Una consideración extra es la manera de realizar entrada y salida a dispositivos. En los ejemplos de CSP, Hoare utiliza los comandos de entrada y salida para comunicación con procesos que realizan esas operaciones con los dispositivos físicos, y se les supone implantados de alguna manera. La extensión natural de esta solución, cuando se piensa en CSP con vocabularios y puertos, son los *canales del sistema*. Una implantación de los puertos del sistema se puede encontrar en el lenguaje Joyce ([Brinch Hansen-1989]).

A través de un canal del sistema, un programa puede solicitar entrada y salida a dispositivos físicos mediante comandos CSP dirigidos a un puerto de ese canal. La diferencia entre un puerto de un canal del sistema y un puerto normal es que el puerto del sistema es una estructura que la implantación debe inicializar. En E', los puertos del sistema adquieren especial relevancia, ya que las unidades pueden necesitar utilizar los periféricos asociados a las máquinas donde se les asigna. Por ello, un puerto del sistema en E' debe ser local a cada unidad. Para declarar un puerto del sistema, es necesario especificar a qué tipo de datos de puerto pertenece (lo que define el vocabulario que acepta):

<PuertoSistema>:

system <Identificador> : <IdentificadorDeTipoPuerto>

Un puerto de sistema deberá ser inicializado por el ambiente que carga el programa E' en el medio distribuido, y es "global" a cada instancia que se active de la unidad (hay un solo puerto del sistema por cada instancia).

## 3.4. EJEMPLOS DE PROGRAMAS

Se presentan a continuación ejemplos resueltos en el lenguaje E'; se incluyen el problema clásico de los cinco filósofos de Dijkstra, y un problema de administración de un dispositivo.

### 3.4.1. Los cinco filósofos

Cinco filósofos pasan su vida entre pensar y comer. Cuando un filósofo decide comer, entra a un cuarto donde hay una mesa servida, con cinco lugares y cinco tenedores; sólo podrá empezar a comer cuando tenga dos tenedores: los que se encuentran a la izquierda y a la derecha de su lugar en la mesa.

Mientras un filósofo tenga sus tenedores, sus vecinos en la mesa no podrán comer, pues les faltaría al menos un tenedor; necesitan esperar a que el primer filósofo deje los tenedores en la mesa. Los filósofos que terminen de comer se retiran del cuarto a seguir pensando.

Hoare utiliza este ejemplo en el artículo de CSP ([Hoare-1978]), y presenta una solución donde los filósofos, los tenedores y el cuarto son procesos que se comunican mediante mensajes. Los filósofos aseguran la posesión de su tenedor de la izquierda, y esperan a tomar el de la derecha. Sin embargo, esta solución no impide que los cinco filósofos entren al mismo tiempo al cuarto, tomen sus tenedores izquierdos y fallezcan de inanición debido a que todos los tenedores derechos están ocupados (los procesos caerían en *deadlock*). Hoare propone como ejercicio reescribir la solución, empleando el número de filósofos que hay

en el cuarto para evitar la situación descrito.

En la solución escrita en E' se usan puertos para comunicación con los tenedores y el cuarto. El programa es el siguiente:

```
PORT roomcom( enter, exit )      " Comunicación con el cuarto "
PORT forkcom( pickup, putdown ) " Comunicación con los tenedores "

UNIT phil(r: roomcom; lf, rf: forkcom)
  " Un filósofo se puede comunicar con el cuarto y con los que
    serán sus tenedores izquierdo y derecho "

  PROC think
  BEGIN  " Acciones de pensar "  END

  PROC eat
  BEGIN  " Acciones de comer "  END

  BEGIN
    WHILE true DO
      think;
      r! enter;       " Entrar al cuarto, .."
      lf! pickup;     "  ..tomar tenedores izquierdo y derecho "
      rf! pickup;
      eat;            " Comer "
      lf! putdown;    "  Al  terminar  de comer, dejar tenedores "
      rf! putdown;
      r! exit         " .. y salir del cuarto "
    END
  END

UNIT fork(p: forkcom)
  " Si un filósofo toma este tenedor, se espera a que lo deje "
  BEGIN
    WHILE true DO
      p? pickup;      " Espera a que tomen este tenedor .. "
      p? putdown      " El tenedor ahora está ocupado "
    END
  END

UNIT room(r: roomcom)
  " El cuarto cuenta los filósofos que están dentro de él; cuando
    un filósofo pide entrar, se verifica la cuenta, para ver si se
    le puede admitir "
  VAR  ocup: int
  BEGIN
    ocup:= 0;
    WHILE true DO          " Atender solicitudes "
      WHEN (ocup<4) AND r? enter DO
        ocup:= ocup + 1
      ELSEWHEN r? exit DO
        ocup:= ocup - 1
      END
    END
  END
```



(continúa)

```
UNIT main  " Esta unidad recibe el control al iniciar ejecución "
  " Hay que definir un arreglo de puertos, para tener comunicación
    con cada uno de los tenedores "
  ARRAY allforks[0:4] (forkcom)
  VAR   f: allforks;
        r: roomcom;
        i: int

  BEGIN
    i:= 0;
    WHILE i<5 DO     " Activar cada uno de los puertos del arreglo "
      +f[i];
      i:= i + 1
    END;
    +r;  " Activación del puerto de comunicación con el cuarto "
    " Activación de todos los procesos que intervienen "
    COBEGIN  room(r)
       ALSO  phil(r, f[0], f[1])
       ALSO  phil(r, f[1], f[2])
       ALSO  phil(r, f[2], f[3])
       ALSO  phil(r, f[3], f[4])
       ALSO  phil(r, f[4], f[0])
       ALSO  fork(f[0])
       ALSO  fork(f[1])
       ALSO  fork(f[2])
       ALSO  fork(f[3])
       ALSO  fork(f[4])
    END
  END
```

### 3.4.2. Un manejador de dispositivo

El puerto del sistema también permite solicitar servicios al sistema operativo o kernel de cada máquina mediante operaciones de entrada y salida de mensajes. Los servicios que se proporcionen a través de dicho canal son básicos, y se les puede administrar desde un programa E'. El siguiente ejemplo implanta un manejador de un puerto serial a través de una unidad de E', la cual atiende las solicitudes y se comunica con su puerto del sistema para concretarlas. Dicho manejador no realiza mayor tratamiento de errores, por simplicidad. El funcionamiento es el siguiente:

Los usuarios obtienen acceso exclusivo al dispositivo mandando la señal 'open'; para comunicación por el puerto serial, pueden utilizar entonces los mensajes 'send' y 'receive'. El acceso al dispositivo concluye cuando el proceso usuario manda la señal 'close'. El respeto al protocolo asegura a los usuarios del

manejador un acceso exclusivo a este recurso.

---

```
" Esta sería la definición del tipo de datos de los mensajes que
  el sistema acepta en sus rutinas "

RECORD info(..)

" Definición del puerto y el vocabulario mediante los cuales un
  proceso usuario se puede comunicar con el administrador "

PORT serial_com(open, send(info), receive, answer(info), close)


"  *****  CODIGO DEL MANEJADOR DE PUERTO SERIAL:  *****  "

UNIT manejador_serial( p:serial_com )
" La definición del tipo de puertos de comunicación con el sistema
  incluye mensajes para todas las acciones que se puedan efectuar;
  en particular, se encuentran los mensajes para realizar
  operaciones básicas sobre el puerto serial "

  PORT system_com(
     ..        " Otras definiciones de mensajes .. "
     ser_reset,                " Inicialización del puerto serial "
     ser_send(info),           " Mensaje para solicitar envío "
     ser_receive(info),        " Mensaje para solicitar recepción "
     ..        " Otras definiciones de mensajes .. "
  )

  SYSTEM sistema: system_com  " Puerto de sistema de esta unidad"

  VAR
     paq: info;       " Auxiliar para guardar la información de
                        las solicitudes de los usuario "
     otravez: boolean

BEGIN
  sistema! ser_reset;       " Inicializar el dispositivo "
  WHILE true DO
    p? open;    " Espera a que alguien quiera usar el manejador "
    otravez:= true;
    WHILE otravez DO
      WHEN p? send(paq) DO            " Si se pide enviar algo.. "
        sistema! ser_send(paq)              " ..pasar la solicitud "
      ELSEWHEN p? receive DO          " Si se pide recibir algo.. "
        sistema? ser_receive(paq);          " ..efectuar recepción.. "
        p! answer(paq)                " ..y contestar al usuario "
      ELSEWHEN p? close DO
        otravez:= false
      END       " end when "
    END         " end while "
  END    " end while "
END      " end proc "
```

---

# Capitulo IV: El Ambiente Actual E'

## 4.1. Objetivos

El lenguaje se ha implantado en una máquina monoprocesador (la IBM PC) mediante un compilador que genera lenguaje ensamblador para los microprocesadores de la familia 80x86 de Intel. No obstante, el compilador es capaz de generar código para otras máquinas (como 680x0 de Motorola y PDP-11) mediante modificaciones mínimas, pues su diseño y estructura es modular. Las partes de la implantación que más cambios requerirían para portarlas a otro ambiente son la biblioteca de ejecución (que implanta los puertos y comandos de CSP) y el kernel multiproceso (que podría ser necesario implantar, o ya existiría).

Inicialmente se consideró construir el compilador de E' modificando al compilador de Edison desarrollado por Brinch Hansen para la IBM PC. Sin embargo, el trabajo presentado en el capítulo III muestra las importantes diferencias que existen entre Edison y el lenguaje E', lo que hizo más atractivo el diseño de un nuevo compilador. Este compilador tiene la ventaja adicional de generar código para una máquina física (y no para un simulador, como es el caso del compilador de Edison), lo que beneficia el desempeño de los programas compilados por él.

En este capítulo, se presenta el diseño e implantación del compilador bajo el sistema operativo MS-DOS, así como el medio ambiente de apoyo a las construcciones del lenguaje.

## 4.2. Procesos

E' es un lenguaje que maneja procesos, y como tal, en su implantación requiere algún medio que le permita administrarlos; al conjunto de rutinas que desarrollan esa función se le llama *kernel multiproceso*. Un kernel de este tipo se forma por un

conjunto de rutinas que instalan procesos, los terminan y, en el caso de multiplexaje, realizan los cambios de contexto.

Se describe en esta sección lo relativo a la implantación de procesos en E' y las rutinas de comunicación CSP, bajo el sistema operativo MS-DOS.

### 4.2.1. Un kernel multiproceso para MS-DOS

Normalmente, un kernel multiproceso se encuentra a nivel del sistema operativo, y el compilador inserta en el código generado llamadas a él para efectuar el manejo de procesos. Sin embargo, MS-DOS no es un sistema operativo capaz de manejar varios procesos: asume la ejecución de un solo programa secuencial (que recibe el control absoluto de la máquina), razón por la cual no es *reentrante* (dos procesos no podrían ejecutar simultáneamente llamadas a MS-DOS, pues un cambio de contexto puede corromper las estructuras internas del S.O.). Es esta la razón por la cual la escritura de los llamados "programas residentes en memoria" de MS-DOS es tan compleja.

El kernel desarrollado para los programas en E' se implanta entonces *sobre* MS-DOS (en vez de integrarse a él), y está escrito en Turbo-C y ensamblador. Resuelve el problema que ocurre con los cambios de contexto haciéndolos explícitos: un proceso seguirá su ejecución hasta que él mismo decida suspenderla (los procesos se comportan como *corrutinas*); así, no hay forma en que dos procesos en E' realicen un cambio de contexto en medio de una llamada a MS-DOS. A nivel de los procesos del lenguaje, los cambios de contexto son transparentes, pues el compilador los inserta automáticamente en lugares específicos del código generado.

Un proceso en este kernel requiere de una dirección de inicio de ejecución, y un espacio para su stack. Este espacio de stack se utiliza para almacenar tanto direcciones de retorno de subrutinas, como parámetros, variables temporales y locales del proceso y procedimientos que éste llame. Los procedimientos y procesos

traducidos por el compilador de E' satisfacen estos requisitos.

Los procesos se representan mediante una lista ligada de *bloques de control de procesos* (BCP); se mantienen dos apuntadores globales: *BCPactual* y *BCPanterior*, que señalan al BCP del proceso que se ejecuta actualmente, y al BCP del proceso que se ejecutaba anteriormente. Se les utiliza en la creación de nuevos procesos, cuyos BCP se insertan entre *BCPactual* y *BCPanterior*.

Cada BCP contiene la información necesaria para efectuar los cambios de contexto y administrar el espacio de stack del proceso (que se asigna mediante manejo dinámico de memoria). Los valores de los registros del procesador se almacenan en el stack de cada proceso:

```
# ifdef CPU_8086        /* Si es compilador para el 8086 .. */
# define MODIF far      /* MODIF es la cadena 'far' */
# else
# define MODIF          /* MODIF es la cadena vacia */
# endif

struct BCP {
     struct BCP MODIF * sig;    /* BCP del sig. proceso */
     char MODIF *stack;         /* Dir. del stack del proceso */
     struct BCP MODIF *padre;   /* BCP del padre de éste */
     void MODIF *bloque;        /* Mem. asignada al proceso */
     void MODIF *dir_trans;     /* Dirección de transferencia */
}
```

(NOTA: El modificador de apuntador *far* es exclusivo de los compiladores de C que generan código para la familia 80x86 de Intel; en otras máquinas, basta con omitirlos).

La organización de los campos de un BCP es la siguiente:

1) *sig* señala al BCP del proceso que se ejecutará cuando el proceso actual decida terminar ó suspender su ejecución.

2) *stack* contiene la dirección en memoria del tope del stack del proceso; como los registros del procesador se almacenan en el stack, para realizar un cambio de contexto basta cargar

el apuntador al tope del stack del 80x86 con esta dirección, y sacar (*pop*) a los demás registros del procesador.

3) *padre* contiene la dirección del BCP del proceso que crea a éste, con el fin de reactivarlo cuando termine una proposición **cobegin**.

4) *bloque* es el apuntador a la zona de memoria asignada para el proceso (la cual incluye espacio de stack y el mismo BCP).

5) *dir_trans* contiene la dirección de transferencia para un proceso suspendido en una operación de E/S de CSP. Si el proceso fue detenido en un comando de salida, contiene la dirección donde se encuentra el valor de la expresión por enviar. Si el proceso fue suspendido en un comando de entrada, contiene la dirección donde depositar el valor leído.

Las llamadas que soporta el kernel multiproceso de E' en MS-DOS son las siguientes:

*Process*(dir. inicial, tamaño de stack, parámetros):

Crea un nuevo proceso, el cual recibirá memoria para su stack con el tamaño indicado, y empezará su ejecución en la dirección indicada, recibiendo los parámetros que se especifican. El nuevo proceso se inserta al final de la lista de BCP's (antes de *BCPactual*).

*Suicide*():

El proceso que ejecuta esta llamada anuncia su terminación. Debe ser la última parte dentro del código del proceso, ya que provoca la liberación de la memoria asignada al proceso.

*Schedule*():

El proceso que la ejecuta indica que desea dejar utilizar el procesador a los demás; se usa el BCP

apuntado por el campo *sig* del proceso actual para elegir un nuevo proceso.. Cuando los demás procesos en la lista también hayan cedido el uso del procesador, el proceso será reactivado.

*Sleep*():

El proceso que lo ejecuta indica que desea suspender su ejecución hasta que alguien lo reactive. Su BCP es retirado de la lista de BCP's, pero no se le destruye ni se libera el espacio de stack.

*Awake*(proceso):

Especifica reactivación del proceso señalado, insertando el BCP en la lista de BCP's.

*Cobegin*(número de procesos):

Con esta llamada, se indica cuántos procesos se están lanzando; el número se usa dentro de la rutina *suicide* para determinar cuándo reactivar al proceso padre (ese momento será cuando el número de hermanos sea cero). En su fase actual, el kernel sólo permite anidar un *cobegin*.

4.2.2. Una implantación de comunicación CSP en memoria compartida

La implantación de las proposiciones de entrada, salida y when se realiza de forma similar a la descrita por Brinch Hansen ([Brinch Hansen-1987.b]) para el lenguaje Joyce.

Los puertos serán estructuras de datos compartidas. Por ello, y ya que los procesos necesitarán accesarlas para modificarlas y leerlas, es necesario garantizar su integridad. En el caso de este kernel de multiproceso, la integridad se asegura al no llamar a la función *Schedule* mientras se accesa una variable puerto; en general, es necesario utilizar semáforos u otra primitiva de sincronización para garantizar un uso ordenado de las estructuras.

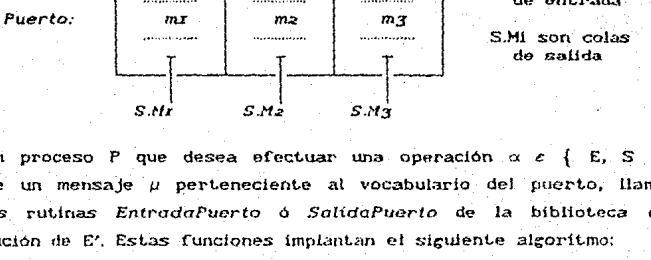
A cada mensaje dentro del vocabulario del puerto se le asigna

un número secuencial, empezando a partir de cero. Un puerto se implanta como un arreglo de tantos elementos como mensajes haya en el vocabulario del puerto. Cada elemento de este arreglo consta de dos colas de BCP's: los BCP's de los procesos que deseen hacer entrada sobre el mensaje, y los BCP de aquellos que deseen efectuar salida del mensaje. Hay que recordar que la estrategia de correspondencia de comandos de entrada con comandos de salida nos asegura que, en todo momento, una de las colas será vacía.

Por ejemplo, la definición de tipo

port *xxx*( m1(int), m2(char), m3(bool) )

provoca que las variables del tipo 'xxx' tengan el siguiente aspecto en memoria:




Un proceso P que desea efectuar una operación $\alpha \in \{ E, S \}$, sobre un mensaje $\mu$ perteneciente al vocabulario del puerto, llama a las rutinas *EntradaPuerto* ó *SalidaPuerto* de la biblioteca de ejecución de E'. Estas funciones implantan el siguiente algoritmo:

1) Con la operación contraria $\beta$ ($\beta \in \{E, S\} - \{ \alpha \}$), revisar la cola $\beta.\mu$;

2) Si $\beta.\mu = \emptyset$, no hay con quién concretar $\alpha$. Se añade P a la cola $\alpha.\mu$ (si el mensaje lleva contenido, se le direcciona con el campo *dir_trans* del BCP de P). Para terminar, se ejecuta *Sleep*(P), para suspender la ejecución del proceso.

3) Si $\beta.\mu \neq \emptyset$, elegir a $Q \in \beta.\mu$, el primer BCP de esa cola

como pareja en la comunicación. Si $\mu$ tiene contenido (no es señal), se transfiere éste empleando el campo *dir_trans* del BCP de Q. Finalmente, se efectúa *Awake*(Q), con lo cual Q reanuda su ejecución.

La implantación de las proposiciones de entrada y salida en custodias involucra una inspección de las colas del puerto mediante las funciones *CustodiaEntrada* y *CustodiaSalida*, que se usan para determinar cuándo es factible una comunicación determinada. De todas las comunicaciones factibles se elige una, la cual se concreta en ese momento.

Esta es una implantación sencilla, beneficiada por la restricción de que proposiciones de entrada o salida en custodias no corresponden entre sí. Emplear un comando generalizado reduciría mucho la eficacia y generalidad posible mediante esta sencilla implantación.

Un problema importante que esta implantación no ataca es el intento de comunicación con procesos cuya ejecución ya haya terminado. En CSP, los comandos de entrada y salida fallan cuando el proceso que mencionan ha concluido, y los comandos alternativo e iterativo tienen conductas específicas (fallo y terminación, respectivamente) cuando los procesos que mencionan en sus guardias han terminado. En el caso de E', se debe idear un mecanismo que permita a un puerto contar cuántos usuarios tiene; esto permitirá determinar cuándo es imposible concretar una operación de E/S a través de ese puerto. Se necesita más trabajo en este sentido antes de adoptar una solución definitiva.

La implantación de E' de los canales del sistema utiliza procesos "ocultos", creados antes de empezar la ejecución del programa, que esperan entrada de mensajes sobre el puerto del sistema para ejecutar operaciones llamando a MS-DOS. Los procesos "ocultos" no están escritos en E', pero usan las funciones de biblioteca de éste.

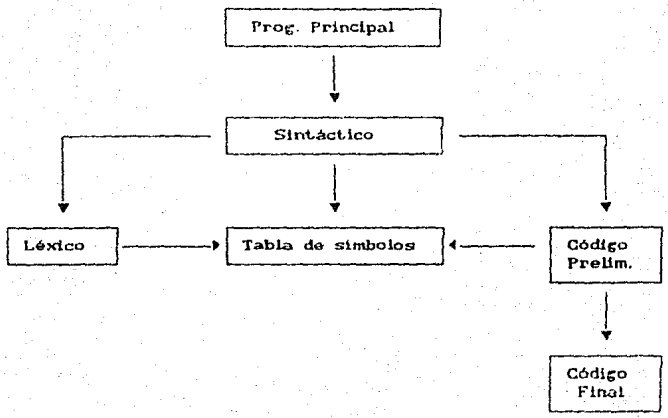
## 4.3. ORGANIZACION DEL COMPILADOR

El compilador de E' es un programa modular, para favorecer su portabilidad. Genera programas en lenguaje ensamblador de los microprocesadores 80x86, pero es posible cambiar esta generación.

A continuación se presentan los módulos empleados en el compilador. El analizador sintáctico es un programa en C generado por la herramienta *Yacc* (disponible en UNIX, MS-DOS y otros sistemas operativos), y todos los demás módulos (léxico, tabla de símbolos, código preliminar y código específico) son programas escritos en la versión ANSI de C, por la portabilidad a futuro y seguridad del lenguaje (de hecho, el módulo analizador léxico pudo haberse generado mediante la herramienta *Lex*, pero se prefirió escribirlo directamente en C, para obtener mayor flexibilidad y eficiencia).

Cada módulo del compilador provee rutinas que sirven para inicializarlo, emplearlo y terminar de usarlo. Los prototipos (declaraciones donde se indica qué parámetros y qué resultado entrega cada función) de las rutinas exportables de cada módulo se almacenan en archivos de encabezados de C (los archivos con extensión *.h*), de forma que los demás módulos los puedan conocer.

La siguiente figura muestra las relaciones entre los módulos del compilador de E'. Una flecha saliendo de la caja de un módulo indica que ese módulo llama rutinas exportables pertenecientes al módulo señalado por la cabeza de la flecha:




En el diagrama faltaría por incluir un módulo más, el cual contiene una rutina de impresión de errores que es utilizada por los demás módulos. El compilador actualmente no tiene recuperación de errores, así que termina su ejecución en cuanto encuentra un problema, llamando a la rutina mencionada.

### 4.3.1. Análisis léxico

El analizador sintáctico generado por *Yacc* espera que una función de análisis léxico le provea con dos elementos: el siguiente token en el archivo de entrada (palabra reservada, identificador, símbolo, etc.) y la cadena de texto asociada al token. Es requisito de *Yacc* que el analizador léxico se llame *yylex*.

El módulo de análisis léxico define como funciones exportables a *yylex*, *arlex_numlinea* y *arlex_numcaracter*, aparte de las funciones de inicialización y fin de uso.

*yylex* es un analizador léxico tradicional, que reconoce los tokens del lenguaje con la ayuda del módulo de la tabla de

símbolos: la tabla de símbolos en su inicialización declara que cadenas son palabras reservadas. Su funcionamiento general es el siguiente:

1) Leer un caracter del archivo de entrada. Si es inicio de comentario, lee caracteres hasta terminarlo.

2) Si el caracter leído es un dígito: lee caracteres mientras sigan siendo dígitos, guardándolos en la cadena *yytext*. Finaliza entregando el valor numérico de la cadena leída.

3) Si es una letra, lee más caracteres mientras sean alfanuméricos, guardándolos en *yytext*. Al concluir la lectura, averigua si se trata de una palabra reservada, llamando al módulo de tabla de símbolos; en tal caso, termina entregando el token correspondiente.
En otro caso, entrega el índice que le correspondería a la cadena en la tabla, predeclarándola si no existe. Este paso simplifica el funcionamiento del análisis semántico.

4) En otro caso, trata de formar un símbolo compuesto (>=, :=, etc.), leyendo más caracteres. Si se forma alguno, entrega el token.

5) Entrega el valor ASCII del caracter leído. Esto maneja el caso de los símbolos como +, -, *, etc.

Las rutinas *yynumlinea* y *yynumcaracter* son usadas para señalar el lugar aproximado en el programa fuente donde ocurrió un error.

4.3.2. Tabla de símbolos

Es muy importante la administración de la tabla de símbolos de un compilador, ya que en ella se almacena toda la información que el compilador obtiene del programador. Esta información se utilizará tanto para el análisis semántico (verificación de

compatibilidad de tipos, por ejemplo) como la generación de código (pues en la tabla se guardan los atributos asociados a los identificadores). Para este módulo, la búsqueda y recuperación de la información de la tabla se hace por metodos de dispersión.

En E', la inicialización de la tabla de símbolos incluye la declaración de las palabras reservadas (**begin**, **end**, **while**, etc.) como tales, así como la definición de los tipos simples (*int*, *char*, *bool*) y constantes predefinidas (*true*, *false*).

El módulo de tabla de símbolos en E' provee rutinas que casi todos los demás módulos emplean, debido a la información que en ella se almacena. Dichas funciones permiten declarar elementos en la tabla (variables, tipos de datos, etc.), establecer ligas entre la información existente en la tabla (como es el caso de declaración de variables locales a un procedimiento), dar de baja información de la tabla (como eliminar las declaraciones de variables locales al término de un procedimiento) y consultar su información.

En el compilador de E', el análisis semántico se encuentra distribuido en el módulo de tabla de símbolos y en las rutinas de generación preliminar de código. En la tabla de símbolos, el análisis semántico verifica que no se tengan identificadores duplicados en un mismo contexto sintáctico.

### 4.3.3 Análisis sintáctico

Como se mencionó previamente, el análisis sintáctico lo realiza el programa generado por *Yacc*. *Yacc* toma como entrada una descripción de la gramática del lenguaje, y permite determinar las acciones a tomar cuando se detecte la aparición de cierta regla. De esta forma se puede realizar el análisis semántico , al mismo tiempo que se hace análisis sintáctico y se produce el código preliminar.

Tómese por ejemplo la descripción gramatical de la declaración

del tipo de datos compuesto **PORT**, la cual incluye el vocabulario manejado por los puertos pertenecientes al tipo:

```
DeclPuerto:
        PORT Unident '(' ListaMensajes ')'
                                { at_declara_puerto($2, $4); }
        ;
```

La ocurrencia del token **PORT**, seguida de los elementos que se enlistan (identificador, paréntesis y lista de mensajes) causa la ejecución de la función 'at_declara_puerto' (que pertenece al módulo de la tabla de símbolos). Esta función hace las verificaciones pertinentes, define el nuevo tipo y los mensajes, en base a los argumentos que se reciben (índice en tabla del identificador, y la lista de mensajes).

El programa generado por *Yacc* se encarga de todo el análisis sintáctico, una vez que se le entrega la gramática del lenguaje.

4.3.4. Generación preliminar de código

Recibe dicho nombre porque las rutinas provistas por su módulo bastan para generar código para una máquina de cero direcciones (máquina de stack). En realidad, este módulo construye los árboles de expresiones, que son recorridos para la generación del código real. También se lleva a cabo la fase final del análisis semántico, verificando la compatibilidad de tipos (en las llamadas a procedimientos, evaluación de expresiones, etc.).

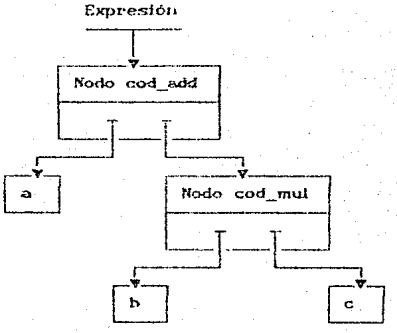
Por ejemplo, dentro de la descripción a *Yacc* de la gramática encontramos llamadas al módulo de código general:

```
Expr:
        NUMERO                  { $$= cod_int_constant($1); }
      | CARACTER                { $$= cod_char_constant($1); }
      | AccesoSimbolo           { $$= cod_value($1); }
      | '(' Expr ')'            { $$= $2; }
      | Expr '*' Expr           { $$= cod_add($1, $3); }
      | Expr '-' Expr           { $$= cod_sub($1, $3); }
      | Expr '/' Expr           { $$= cod_mul($1, $3); }
      ;
```

Cada una de estas llamadas entrega un nodo, que tiene como hijos a los parámetros indicados. De esta forma, una expresión aritmética como

a + b * c

queda representada en un árbol de expresión como se muestra en la siguiente figura:




Estos árboles se construyen para evaluación de expresiones, asignación de valores a variables, llamadas a procedimientos, etc. Constituyen una representación interna del código preliminar, que va creciendo hasta el punto en que es posible generar código correcto y eficiente en ensamblador.

4.3.5. Generación final de código

A partir de la representación de árboles, el módulo de código preliminar genera llamadas al módulo de código final para obtener la traducción a lenguaje ensamblador.

El propósito principal de este módulo es ocultar los detalles del lenguaje ensamblador de la máquina destino al módulo de código preliminar. Este último asume un procesador capaz de efectuar cualquier operación que se le pida, y el módulo de generación

final se encarga de "mapear" estas operaciones en el lenguaje ensamblador del procesador destino.

Considérese el fragmento del módulo de código preliminar que realiza el recorrido de un árbol de expresiones (el parámetro p es la raiz del árbol):

```
LOCAL void codigo_en_orden(Simbolo *p)
{
     if(p==NULL)  return;
     switch(p->clase) {
          case EL_CONSTANTE:
               if(modo_registro==DISPONIBLE)
                    opera_acum_inm(COPIA, p->inst.val);
               break;

          case EL_VARIABLE:
               if(modo_registro==DISPONIBLE)
                    opera_acum_mem(COPIA, p->inst.val);
               break;

          case EL_PUERTO:
               acceso_puerto(p->inst.val);
               break;

          case EL_BINARIO:
               op_binaria(p);
               break;

          case EL_UNARIO:
               op_unaria(p);
               break;
     }
}
```

Las funciones *op_binaria*, *op_unaria* manejan el caso en que en el nodo haya subexprosiones; ellas volverán a llamar a *codigo_en_orden* para evaluarlas.

Al recorrer el árbol de expresiones, la generación de código asume la existencia de un registro *acumulador* en el procesador, donde el código generado deposita los resultados de las evaluaciones parciales de la expresión. Al empezar a generar código para una expresión, dicho acumulador se encuentra libre.

Las llamadas *opera_acum_inm* (opera con el acumulador con direccionamiento inmediato), *opera_acum_mem* (opera el acumulador

con un argumento en memoria) y *acceso_puerto* forman parte del módulo de código específico, y manejan el acceso a un elemento particular (constante, variable y un puerto CSP). En el caso de *opera_acum_inm* y *opera_acum_mem*, se toma como parámetro adicional la operación a realizar; estas rutinas esconden los detalles de direccionamiento del procesador elegido. Por ejemplo, para el 8086, la implantación de *opera_acum_inm* es la siguiente:

```
LOCAL void opera_acum_inm(CodOps op, int val)
    {    /* 'op' es un símbolo asociado a las operaciones
              posibles;
            'val' es el número con el que hay que operar          */
    Mnemo buf;      /* Cadena de caracteres para el mnemónico */

    mnemonico(op, buf);  /* Obtener mnemónico de la operación */
    if(op>=COPIA AND op<=RESTA) {    /* Ops. COPIA, SUMA, RESTA */
        fprintf(fp, "%s ax,", buf);
        print_arg(val);
    }
    else if(op<=MODULO) {          /* Ops. MULT, DIV, MODULO */
        opera_regsec_inm(COPIA, val);
        if(op==DIVISION || op==MODULO)
        fputs("mov dx, 0", fp);
        fprintf(fp, "%s bx", buf);
        if(op==MODULO)
            fputs("mov ax, dx", fp);
    }
}
```

Para el caso de COPIA, SUMA y RESTA, se utiliza el argumento de número tal cual, ya que el 8086 es capaz de usar direccionamiento inmediato en estos casos. Sin embargo, el caso de las operaciones MULT, DIV, MODULO es más complejo, ya que en el 8086:

1) La multiplicación, división y módulo no pueden usar direccionamiento inmediato (sólo pueden operar con el acumulador y un argumento en memoria ó algún registro).

2) En división y módulo, el registro DX debe estar a cero, pues forma parte del dividendo junto con AX.

3) En el caso del módulo, la operación del procesador es la misma que la de la división, pero el residuo queda en el

registro DX y es necesario transferirlo al acumulador para continuar.

Para implantar el paso de parámetros y declaración de variables locales en los procedimientos y unidades se utiliza el stack del procesador de la siguiente manera (el caso de procedimientos y unidades es muy parecido, así que se ejemplificará mediante los primeros):

1) Antes de llamar a un procedimiento, las evaluaciones de sus parámetros se meten al stack del procesador, en el orden de declaración (o sea, el primer parámetro que se mete es el primero que se declara). Entonces, se genera el código que llama al código del procedimiento.

2) Dentro del procedimiento, se obtiene espacio para las variables locales en el stack, modificando el tope del stack del procesador. Las variables locales y parámetros se direccionan de manera relativa al tope del stack. Cuando concluya la ejecución del procedimiento, se restaura el tope del stack y se sacan los parámetros del stack.

Por ejemplo, el siguiente procedimiento

```
PROC Ejemplo(a,b: int)
  VAR  x: int
BEGIN
  x:= a + b
END
```

genera el siguiente código ensamblador la IBM PC (el registro 'BP' del 8086 se utiliza para direccionar el stack, ya que el registro de tope de stack 'SP' no es capaz de ello):

```
Ejemplo    PROC       NEAR
     push       bp
     mov        bp, sp        ;; BP tiene el tope de stack ;;
     sub        sp, 2         ;; Obtener espacio para 'x' ;;
     mov        ax, [bp + 6]      ;; Accesa parámetro 'a' ;;
     add        ax, [bp + 4]      ;; Suma parámetro 'b' ;;
     mov        [bp - 2], ax      ;; Asigna a 'x' ;;
     pop        bp            ;; Restaura 'BP' ;;
     ret        4             ;; Termina sacando cuatro bytes
                              ;; del tope del stack ;;
Ejemplo ENDP
```

Por otra parte, la llamada al procedimiento

```
Ejemplo( 5, 3);
```

genera el siguiente código:

```
mov     ax, 5
push    ax
mov     ax, 3
push    ax
call    Ejemplo
```

Mediante los llamados en la generación especifica de código es posible abstraer al compilador del tipo de CPU que se utiliza. Solamente es necesario reescribir el módulo de código especifico para generar instrucciones para una nueva CPU.

4.3.6. Uso del kernel y biblioteca CSP por el código generado

En esta sección se muestra el uso que hace el código generado de las rutinas del kernel y biblioteca CSP. Se muestran dos ejemplos: la proposición de entrada CSP y la proposición **cobegin**. No se utiliza el lenguaje ensamblador de una máquina especifica para los ejemplos, sino un pseudo-código.

Cuando el compilador encuentra la proposición de entrada de CSP

```
puerto? mensaje( variable )
```

genera el siguiente código:

```
Obtener dirección en memoria de 'variable'
Obtener el tamaño del tipo de datos de 'variable'
Usar el número asignado a 'mensaje'
Obtener la dirección de 'puerto'
Llamar a EntradaPuerto con esos parámetros
```

El código generado cuando 'mensaje' es una señal es muy semejante, salvo porque se indica este hecho pasando un cero como parámetro en el lugar del tamaño de la variable.

La compilación de la proposición **cobegin**

```
cobegin   q1( <parámetros reales> )
  also    q2( <parámetros reales> )
end
```

produce un código que interpreta el siguiente algoritmo:

1) Evaluación de los parámetros de cada proceso
2) Instalar como proceso a un programa que llame al código generado para cada 'q'. Después de dicha llamada, ese proceso se suicida.
3) Llamar a la rutina *cobegin* para activar los procesos.

El código generado por la proposición cobegin enlistada arriba tiene el siguiente aspecto:

```
Saltar a Etiq3
    Etiq1:
        Llama al código generado para 'q1'
        Llama a Suicide();
    Etiq2:
        Llama al código generado para 'q2'
        Llama a Suicide();
Etiq3:
    Evalúa parámetros para 'q1'
    Process(); con los parámetros y Etiq1
    Evalúa parámetros para 'q2'
    Process(); con los parámetros y Etiq2
    Cobegin(2);
```

Nótese que es posible optimizar este código si el procesador destino permite llamar subrutinas de forma indirecta, ya que las acciones en *Etiq1* y *Etiq2* son las mismas.

## 4.4. ACERCA DE LA IMPLANTACION EN UN AMBIENTE DISTRIBUIDO

Cabe en este momento señalar las modificaciones al ambiente E' que pueden realizarse en este momento. Se destaca especialmente la implantación de E' en un medio distribuido tal como una red de computadoras. Los problemas principales para implantar E' en un medio distribuido están relacionados con las unidades y la comunicación CSP.

### 4.4.1 Mapeo de código y unidades

El concepto de unidad permite relacionar código y procedimientos que deben accesar los mismos recursos físicos. Sin embargo, está fuera del lenguaje el problema de especificación de la máquina donde se desea ejecutar el código de una unidad, que se puede resolver mediante los siguientes pasos:

1) Usar directivas al nuevo compilador para especificar la máquina asociada a la unidad. El resultado de la compilación son varios archivos objeto, destinados a cada una de las máquinas, los cuales deben ligarse individualmente con la biblioteca de ejecución de E' y con ambientes locales que implanten los puertos de sistema de cada unidad. Asimismo, es necesario que el compilador genere tablas de puertos parámetros para cada unidad, así como la ubicación de las unidades en las máquinas, pues esta información representará comunicaciones entre máquinas, y es necesario emplearla en tiempo de ejecución.

2) Escritura de un cargador de programas, que utilice la información generada por el compilador para colocar los archivos ejecutables en la red. Además, se necesita llenar la información de las tablas de puertos, indicando la dirección en red de las máquinas que contienen unidades.

Debe asimismo manejarse la activación de unidades, por parte de otras máquinas. Esta se puede llevar a cabo cuando la máquina con la unidad reciba un mensaje de red solicitando una nueva activación; este mensaje debe contener información acerca de los parámetros de activación, especialmente respecto a qué máquina declaró e inicializó los puertos parametro. En ese momento, la máquina dueña de la unidad puede crear la infraestructura para la implantación de los puertos externos, e instalar como proceso una nueva instancia de la unidad.

### 4.4.2. La biblioteca de comunicación CSP

Las modificaciones principales a la biblioteca de comunicación CSP radican en el manejo de la situación distribuida. Los puertos externos a una unidad deben implantarse en dos niveles: interno (en la máquina donde se encuentra la unidad) y externo (en la máquina donde se inicializó el puerto), y la idea fundamental es reemplazar las colas de BCP por colas de mensajes de red, que contengan información respecto a qué máquina las ha enviado.

Internamente, el puerto externo se representa por una estructura similar a la descrita para memoria centralizada: como un arreglo de colas de BCP, cada una correspondiendo a un mensaje del vocabulario del puerto, mas un indicador de puerto externo y la tabla generada por el compilador. Cuando la unidad utilice un puerto externo, un proceso de interfaz debe usar la tabla de puertos (descrita en el inciso anterior) para conocer la dirección en red de la máquina que declaró e inicializó el puerto externo, y enviarle un mensaje de red indicando quién envia la solicitud, su localización y la operación que desea realizar. Se puede entonces suspender en las colas de BCP's al proceso que ejecuta el comando, en espera de una respuesta.

En el nivel externo, el puerto se puede representar como un arreglo de colas de mensajes de red, cada una correspondiendo a los mensajes que maneja el vocabulario del puerto. Al recibir un mensaje por la red, solicitando una operación CSP sobre el puerto, esta máquina puede emplear con el mensaje un algoritmo semejante al descrito en este capitulo para la memoria centralizada, averiguando qué mensaje de los que ya se hallan en el puerto puede corresponder a la nueva solicitud. Si existe un mensaje de tal índole, la máquina dueña del puerto puede poner en contacto a las máquinas interesadas, para que estas prosigan con la transferencia de información y sincronización. En caso de que no exista un mensaje correspondiente a la solicitud, esta se forma en la cola de mensajes correspondiente.

Para implantar la evaluación de posibilidad de comunicación

(para las guardias de la proposición alternativa), el proceso interfaz al nivel externo deberá recibir una solicitud de evaluación; el proceso elige una alternativa de comunicación (si la hay), y comunica su existencia, marcando a la alternativa como *en consulta*, para no cederla a nuevas solicitudes. Quien envió la solicitud de evaluación debe entonces responder indicando si acepta o no la comunicación; de aceptarla, se usa un algoritmo semejante al descrito para concretar la transferencia de información. Si la comunicación fué rechazada, el proceso de nivel externo debe marcar como *libre* a la solicitud, de manera que ésta pueda ser utilizada nuevamente.

La implantación de E' en un medio distribuido será un ejercicio muy interesante, pues implica el establecimiento de un protocolo que maneje los posibles errores de transmisión de los mensajes de red. Esta nueva implantación es promisoria, y con mucho, es el aspecto más interesante de trabajo futuro en E', y constituirá su prueba de fuego.

# CONCLUSIONES

El proyecto del lenguaje E' no se puede considerar terminado; hace falta más experiencia en el uso práctico del lenguaje (especialmente en un medio distribuido), antes de poder determinar las modificaciones o extensiones que se le han de hacer.

En este proyecto, se ha comprobado de manera práctica la notable diferencia entre una programación *centralizada* (donde existe una memoria común) y una programación distribuida. Edison es un ejemplo típico de un lenguaje de programación centralizada: su manejo de procesos, procedimientos y el concepto del ambiente global lo reflejan nítidamente, y su implantación se dificulta cuando no existe la memoria común.

CSP propone una visión de programación diferente: los entes constitutivos de un programa no cuentan con más medio de comunicación y sincronización que unas sencillas operaciones de entrada y salida (aunque una implantación general de CSP resulte complicada al momento de establecer protocolos de envío de mensajes). La visión de CSP es un reflejo claro del medio distribuido.

Dadas estas premisas, el diseño de un lenguaje de programación distribuida, que tome un lenguaje centralizado como base, debe empezar con un análisis concienzudo de las características y proposiciones del lenguaje, para eliminar aquellas que choquen o presenten conflictos con el modelo de distribución elegido (ya sea intercambio de mensajes, o llamada a procedimientos remotos). En todo momento, hay que preservar un *ambiente limpio y sencillo de programación*, lo que además de favorecer el aspecto formal del lenguaje, simplifica mucho su implantación. Las decisiones de diseño son muchas; la experiencia resultante de este trabajo muestra la cantidad e importancia de las modificaciones realizadas a Edison, que cambiaron el nombre del nuevo lenguaje de *Edison CSP* a E'. Sin embargo, el resultado es de gran interés, pues se favorece un diferente y agradable estilo de programación.

En cuanto a aspectos a considerar en E', el trabajo actual puede continuarse en los siguientes aspectos:

Implantación de E' en un ambiente distribuido: Esta podría desarrollarse en una red de computadoras, siguiendo las ideas delineadas en la sección 4.4, y permitiría evaluar dos formas de pensar: el programa en E' toma el control de la red (haciendo funcionar a ésta como una máquina paralela), o entra a *colaborar y usar* servicios que ofrezcan las máquinas que ya se hallan en la red (quizá mediante un concepto semejante al de puerto del sistema, llamémosle *puertos de servicios externos*, los cuales se localizarían mediante nombres).

Estudio del problema de terminación de los procesos: Hay que encontrar una solución sencilla que permita a la implantación de E' determinar cuándo una operación CSP ha fallado, por término de todos los procesos usuarios de un puerto. Este problema es importante de resolver, pues ayudará en la depuración de aplicaciones.

Trabajos sobre el compilador: El compilador puede extenderse en varios campos; se puede perfeccionar la portabilidad del compilador (generando código para máquinas tales como la PDP-11 y la familia 680x0 de Motorola), añadir el tipo de datos **real** al lenguaje (considerando además las implicaciones que tiene en cuanto a la semántica del lenguaje), efectuar recuperación de errores y realizar optimización de código. Las rutinas que forman el compilador pueden usarse dentro de la Facultad, para nuevos proyectos ó en enseñanza.

Una implantación distribuida que utilice un compilador que implante el tipo **real** puede evaluarse en un proyecto de aplicación, tal como usar una red de computadoras para generación de imágenes por el método de *ray-tracing*. Esta aplicación permitirá además calibrar el lenguaje, y la eficacia de su implantación.

La experiencia obtenida muestra la importancia de un diseño de

lenguaje adecuado al ambiente de programación donde se le piensa implantar. Seguramente E' no es el mejor lenguaje para realizar programación centralizada (no resulta muy eficiente en cuanto a aprovechamiento de ese hardware), pero su modelo basado en CSP le da ventaja en aplicaciones distribuidas. El punto importante, en el punto de vista de una aplicación, será la forma de realizar el enlace entre el nivel local y el global.

La experiencia futura en el uso de E' ha de dictar los nuevos rumbos del lenguaje.

# BIBLIOGRAFIA

ARTICULOS:


[Albarrán-1988]

Albarrán, M.; López, J.A.
*Diseño de Manejadores de Dispositivos en Edison*
Memorias del Congreso "30 años de la Computación en México" 1988

[Apt-1980]

Apt, K.R.; Francez, N.; De Roever, W.P.
*A Proof System for Communicating Sequential Processes*
ACM Trans. Program. Lang. Syst. 2(3); Jul.1980

[Bagrodia-1989]

Bagrodia, R.
*Synchronization of Asynchronous Processes in CSP*
ACM Trans. Program. Lang. Syst. 11(4); Oct.1989

[Bal-1989]

Bal, H.E.; Steiner, J.G.; Tanenbaum, A.S.
*Programming Languages for Distributed Systems*
ACM Comput. Surveys 21(3); Sep.1989

[Bernstein-1980]

Bernstein, A.J.
*Output Guards and Nondeterminism in "Communicating Sequential Processes"*
ACM Trans. Program. Lang. Syst. 2(2); Apr.1980

[Brinch Hansen-1972]

Brinch Hansen, Per
*Structured Multiprogramming*
Commun. ACM 15(7) Jul.1972

[Brinch Hansen-1978]
Brinch Hansen, Per
*Distributed Processes: A Concurrent Programming Concept*
Commun. ACM 21(11); Nov.1978

[Brinch Hansen-1981]
Brinch Hansen, Per
(a) *Edison - a multiprocessor language*
(b) *The design of Edison*
(c) *Edison programs*
Softw. Pract. Exper. Apr.1981

[Brinch Hansen-1987.a]
Brinch Hansen, Per
*Joyce - A Programming Language for Distributed Systems*
Softw. Pract. Exper. 17(1); Jan.1987

[Brinch Hansen-1987.b]
Brinch Hansen, Per
*A Joyce Implementation*
Softw. Pract. Exper. 17(4); Apr.1987

[Brinch Hansen-1989]
Brinch Hansen, Per
(a) *A Multiprocessor Implementation of Joyce*
(b) *The Joyce Language Report*
Softw. Pract. Exper. 19(6); Jun.1989

[Buckley-1983]
Buckley, G.N.; Silberschatz, A.
*An Effective Implementation for the Generalized Input-Output Construct of CSP*
ACM Trans. Program. Lang. Syst. 5(2); Apr.1983

[Dijkstra-1968]
Dijkstra, E.W.
*Cooperating Sequential Processes*
ed. F. Genuys en *Programming Languages*
Academic Press, New York. 1968

[Dijkstra-1975]
Dijkstra, E.W.
*Guarded Commands, Nondeterminancy and Formal Derivation of Programs*
Commun. ACM 18(8); Aug.1975

[Hoare-1974]
Hoare, C.A.R.
*Monitors: An Operating System Structuring Concept*
Commun. ACM. 17(10); Oct.1974

[Hoare-1978]
Hoare, C.A.R.
*Communicating Sequential Processes*
Commun. ACM 21(8); Aug.1978

[Pountain-1989]
Pountain, D.
*Occam II*
Byte 14(10); Oct.1989

[Sánchez-1990]
Sánchez, V.G.
*Programación Concurrente y Distribuida*
Notas I.I.M.A.S. - U.N.A.M. Mar.1990

[Silberschatz-1979]
Silberschatz, A.
*Communication and Synchronization in Distributed Systems*
IEEE Trans. Softw. Eng. SE-5(6); Nov.1979

REFERENCIAS:


- Borland International, Inc.
  *Turbo Pascal Version 3.0 - Reference Manual*
  1985

- Brinch Hansen, Per
  *Programming a Personal Computer*
  Prentice-Hall, Englewood Cliffs 1981

- Johnson, S.C.
  *Yacc - Yet Another Compiler-Compiler*
  Comp. Sci. Tech. Rep. #32
  Bell Laboratories, Murray-Hill Jul.1975

- Kernighan, B.; Pike, R.
  *El entorno de programación UNIX*
  Prentice-Hall, México. 1987

- Tanenbaum, A.S.
  *Organización de Computadoras*
  Prentice-Hall, México. 1988

- Whiddett, D.
  *Concurrent Programming for Software Engineers*
  Ellis Horwood Series in COMPUTERS AND THEIR APPLICATIONS
  John Wiley and Sons 1987

## APENDICE: GRAMATICA DE E'

Las definiciones de la gramática de E' utilizan la siguiente notación BNF modificada:

1) Los simbolos terminales aparecen subrayados.
2) Los simbolos no-terminales se enmarcan con '<' y '>'.
3) El simbolo ':' separa el nombre de la producción de su contenido.
3) El simbolo '||' denota opción de selección dentro del contenido de la producción.
4) Los simbolos'[' y ']' denotan aparición opcional de los elementos que contienen.
5) Los simbolos '{' y '}' denotan la repetición cero o más veces de los elementos que encierran.

---

*Declaración de un programa*

---

<Programa> :
{ <DeclInfo> }
<DeclUnidad> { <DeclUnidad> }

---

*Declaraciones informativas al compilador*

---

<DeclInfo> :
<ListaDeclConst> || <DeclTipo>

<ListaDeclConst> :
const <DeclConst> { ; <DeclConst> }

<DeclConst> :
<NombreConstante> = <Constante>

<DeclTipo> :
<DeclArreglo> || <DeclConjunto> || <DeclEnum>
|| <DeclPuerto> || <DeclRegistro>

<DeclArreglo> :
array <NombreTipo> [ <Rango> ] < <TipoElem> >

<DeclConjunto> :

set <NombreTipo> <TipoElem>

<DeclEnum> :

enum <NombreTipo> ( <ListaIdentificadores> )

<DeclRegistro> :

record <NombreTipo> ( <ListaDeCampos> )

<ListaDeCampos> : <GrupoDeCampos> { ; <GrupoDeCampos> }

<GrupoDeCampos> : <ListaIdentificadores> : <TipoDeCampos>

<TipoElem> : <Identificador>

<TipoDeCampos> : <Identificador>

<NombreConstante> : <Identificador>

<NombreTipo> : <Identificador>

<Rango> : <LimInferior> : <LimSuperior>

<LimInferior> : <Constante>

<LimSuperior> : <Constante>

---

*Declaraciones relativas a puertos*

---

<DeclPuerto> :

port <Identificador> ( <ListaMensajes> )

<DeclPuertosSistema> :

system <ListaDeclVars>

<Lista de mensajes> :

<Mensaje> { , <Mensaje> }

```
<Mensaje> :
        <Identificador> [ ≤ <Identificador> ≥ ]
```

*Declaraciones de estructuras de código*

```
<DeclUnidad> :
        unit <Identificador> [ ≤ <ListaParamsFormales> ≥ ]
        { <DeclInfo> }
        [ <DeclPuertoSistema> ]
        { <DeclProcedimiento> }
        { <DeclVars> }
        begin <ListaProps> end

<DeclProcedimiento> :
        proc <Identificador> [ ≤ <ListaParamsFormales ≥ ]
        { <DeclInfo> }
        { <DeclProcedimiento> }
        { <DeclVars> }
        begin <ListaProps> end
```

*Parámetros y variables*

```
<ListaParamsFormales> :
        <GrupoDeclaraciones> { ; <GrupoDeclaraciones> }

<GrupoDeclaraciones> :
        <ListaIdentificadores> : <Identificador>

<DeclVars> :
        var  <GrupoDeclaraciones> { ; <GrupoDeclaraciones> }

<ListaParamsReales> :
        <ListaExpresiones>
```

*Proposiciones*

```
<ListaProps> :
        <Proposición>    [ ; <Proposición> ]
```

```
<Proposición> :
        <ProposiciónConcurrente>
    || <ProposiciónCSP>
    || <ProposiciónSecuencial>
```

*Proposiciones concurrentes y relacionadas con CSP*

```
<ProposiciónConcurrente> :
    cobegin <ActivaciónProceso>
    { also <ActivaciónProceso> }
    end

<ActivaciónProceso> :
    <ActivaciónUnidad> || <ActivaciónProcedimiento>

<ActivaciónUnidad> :
    <Identificador> [ ( <ListaParamsReales> ) ]

<ActivaciónProc> :
    <LlamadaProcedimiento>

<ProposiciónCSP> :
        <IniPuerto>
    || <PropEntradaSalida>
    || <PropAlternativa>

<IniPuerto> :
    ± <AccesoVariable>

<PropEntradaSalida> :
    <PropEntrada> || <PropSalida>

<PropEntrada> :
    <AccesoVariable> ? <Mensaje> [ ( <AccesoVariable> ) ]

<PropSalida> :
    <AccesoVariable> ! <Mensaje> [ ( <Expresión> ) ]
```

<PropAlternativa> :

when <Custodia> do <ListaProps>

{ elsewhen <Custodia> do <ListaProps> }

end

<Custodia> :

[ <Expresión booleana> ] AND <PropEntradaSalida>

---

*Proposiciones secuenciales*

---

<ProposiciónSecuencial> :

<PropCondicional> || <PropIterativa>

|| <PropVacia> || <Asignación> || <LlamadaProc>

<PropCondicional> :

if <Expresión booleana> do <ListaProps>

[ else <Expresión booleana> do <ListaProps> ]

end

<PropIterativa> :

while <Expresión booleana> do <ListaProps>

[ else <Expresión booleana> do <ListaProps> ]

end

<PropVacia> : skip

<Asignación> : <AccesoVariable> := <Expresión>

<LlamadaProcedimiento> :

<Identificador> [ ( <ListaParamsActuales> ) ]

---

*Acceso a variables*

---

<AccesoVariable> :

<Identificador>

|| <AccesoCampo>

|| <AccesoElemArreglo>

|| <Enmascaramiento>

<AccesoCampo> :

<AccesoVariable> . <Identificador>

<AccesoElemArreglo> :

<AccesoVariable> [ <Expresión> ]

<Enmascaramiento> :

<AccesoVariable> : <Identificador>

---

*Expresiones*

---

<ListaExpresiones> :

<Expresión> { , <Expresión> }

<Expresión> :

<ExpresiónSimple> { <OperRelacional> <ExpresiónSimple> }

<ExpresiónSimple> :

<TérminoConSigno> { <OperAditivo> <Término> }

<TérminoConSigno> :

[ <OperSigno> ] <Término>

<Término> :

<Factor> [ <OperMultiplicativo> <Factor> ]

<Factor> :

<Constante>
| <AccesoVariable>
| <Constructor>
| ( <Expresión> )
| not <Factor>

<OperRelacional> : = | <> | < | > | <= | >= | in

<OperAditivo> : + | - | or

<OperSigno> : + | -

<OperMultiplicativo> : * || / || mod || and

---

*Constructores*

---

<Constructor> :
<NombreTipo> ( <ListaExpresiones> )

---

*Constantes*

---

<Constante> :
<NombreConstante>
|| <ConstanteNumérica>
|| <ConstanteCaracter>
|| <ConstantePredefinida>

<ConstanteNumérica> :
[ <OperSigno> ] <Dígito> { <Dígito> }

<ConstanteCaracter> :
' <Caracter> '
|| char ( <ConstanteNumérica> )

<ConstantePredefinida>: true || false

---

*Identificadores*

---

<ListaIdentificadores> :
<Identificador> { , <Identificador> }

<Identificador> :
<CaracterAlfabético> { <CaracterVálido> }

<CaracterVálido> :
<CaracterAlfabético>
|| <CaracterNumérico>
|| _ -- Caracter de subrayado --

<CaracterAlfabético> : a || .. || z ||A || .. || Z

<CaracterNumérico> : 0 || .. || 9