CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS

DEL

INSTITUTO POLITECNICO NACIONAL

DEPARTAMENTO DE INGENIERIA ELECTRICA

SECCION DE COMPUTACION



# "IMPLEMENTACION DEL LENGUAJE POLUX PARA COMPUTACION DISTRIBUIDA"



Tesis que presenta el Ing. Roberto Félix Cárdenas para obtener el grado de MAESTRO EN CIENCIAS en la especialidad de INGENIERIA ELECTRICA con opción en COMPUTACION.

Director: DR. JAN JANECEK HYAN

Becario de CONACYT

Mexico, D. F., Junio de 1990

Mis mas profundos agradecimientos a las siguientes instituciones por su ayuda brindada:

A la Sección de Computación del Departamento de Ingenieria Electrica del CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL IPN y en forma particular al Dr. JAN JANECEK HYAN por su paciencia y consejos en la dirección de este trabajo.

Al CONSEJO NACIONAL DE CIENCIA Y TECNOLOGIA por su apoyo economico.

A la UNIVERSIDAD AUTONOMA DE SINALOA y en particular a la Escuela de Ingeniería unidad Los Mochis, por su apoyo.

# C O N T E N I D O

# R E S U M E N

El presente trabajo describe la implementación de primitivas de llamadas a procedimientos remotos (RPC) para POLUX, lenguaje orientado hacia la computación distribuida. Asi mismo una descripción de un compilador para dicho lenguaje.

El primer capitulo contiene una breve introducción a la programación distribuida, asi mismo habla un poco sobre el modelo cliente-servidor, pasando a continuación a un mecanismo de llamadas a procedimientos remotos. En este mismo capítulo se introducen algunos terminos usados en el desarrollo del trabajo.

El segundo capitulo contiene principalmente una descripción de aquellas construcciones sintacticas orientadas a los procesos distribuidos del lenguaje Polux, junto con los cuales se enumeran algunas acciones semanticas de cada construcción. Tambien se hace el estudios de un ejemplo tipico de un productor y varios consumidores.

El Capitulo tres trata sobre los mecanismos de llamadas a procedimientos remotos (en ingles conocidos como *remote procedure call* ó *RPC*), describiendo cada uno de los mensajes utilizados por este nivel, asi mismo lo referente a la función y organización del controlador de comunicación de procesos. De la misma forma se decriben los detalles de implementación del sistema.

A continuación en el capitulo cuatro, se mensionan algunos aspectos del compilador, como son, la descripción de las principales tablas usadas, una definición de los codigos intermedios generados por la parte frontal del compilador (P-Code), asi mismo se describen algnos detalles de la generación del codigo objeto (para el 8086 de Intel).

En el capítulo cinco se decribe en forma breve la estructura y funcionamiento de la red usada para la comunicación entre maquinas.

El apendice A contiene las cartas sintacticas de lenguaje Polux.

Se tiene tambien un apendice B, el cual incluye algunos ejemplos usando el lenjuage Polux, junto con los cuales para varios de ellos, se tienen algunos comentarios utiles para su comprensión, asi mismo se muestran las tablas genaradas por el compilador.

Finalmente se tiene una lista de bibliografia que pueden servir de referencia al presente trabajo.

# 1

## INTRODUCCION

Antes de entrar en detalles del presente trabajo, definamos algunos terminos usados en los diferentes capitulos:

*PROCESO* - usado para describir un programa en ejecución, al cual estan asociados: un código ejecutable que puede accesar, una *pila* donde opera sus datos, un *descriptor de procesos*, etc.

*PILA* - conocida tambien como *medio ambiente de ejecución*, que define basicamente una área de memoria donde se ejecuta un proceso, en la cual residen sus parametros y variables locales, y donde se almacenan resultados parciales que el programa genera.

*DESCRIPTOR DE PROCESOS* - contiene toda la información relativa a un proceso, como son apuntadores a su código y su pila, áreas para almecenar los registros del procesador, etc.

*AGENTE* - termino usado para identificar un bloque *agent* definido en el lenguaje Polux, el cual contiene un código, una interface que lo relaciona con otros agentes, en cuanto a sus variables y procedimientos. De un agente pueden ser

creados varios procesos.

*TRAMA* - denominado al conjunto de información que viaja por la red, y que son dirigidos de estaciones a estaciones.

*MENSAJE* - usado por el stubs del sistema, los cuales son generados por los procesos y van dirigidos a otros procesos, estos mensajes generados por un proceso, pasan al stubs, para que éste los transforme en tramas que se pondran en la red. Asi mismo el stubs recibe tramas de la red y forma los mensajes dirigidos a alguno de sus procesos.

*STUBS* - conjunto de procedimientos que controla la comunicación de mensajes entre procesos, asi como el control de la ejecución de los procesos. Sirbe como mediador entre el programa de aplicación y la entidad de transporte, recibiendo y mandando mensajes del primero y tramas del segundo.

*ENTIDAD DE TRANSPORTE* - es la encargada de recibir tramas del stubs y ponerlas en la red para la estación destino, y de manera inversa recoger las tramas destinadas a su estación y darselas al stubs.

## 1.1. PROGRAMACION Y SISTEMAS DISTRIBUIDOS

En un sistema distribuido se considera que se tiene mas de un proceso corriendo de manera simultanea, los cuales pueden encontrarse en máquinas distintas. El conjunto de estos procesos forman un sistema distribuido, por lo que posiblemente todos tengan algo en común, como pueden ser áreas de variables compartidas, recursos periféricos únicos que todos y cada uno usan en su momento, un mismo código, etc.

Para obtener una cooperación entre todos los procesos existentes, es necesario la conctrucción de mecanismos de

coordinación, los cuales permitan que por ejemplo, un proceso haga uso de una variable de algún otro proceso, o bién que cualquier proceso pueda ejecutar algún procedimiento contenido en un proceso distinto.

En tal sistema distribuido se debe pensar una justa distribución de recursos, para que todos los procesos puedan correr adecuadamente, y entre los recursos mas importantes podemos encontrar el tiempo de procesador asignado a cada proceso (sobre todo si estos procesos corren en la misma máquina), por lo cual es de suma importancia contar con construcciones sintácticas y/o semánticas en el lenguaje de programación para que todos los procesos puedan correr el tiempo requerido por cada uno de ellos.

Un lenguaje orientado a los procesos distribuidos debe contener construcciones sintácticas, tales que hagan transparente al usuario todo el control distribuido, esto es, que el usuario solo se preocupe por la designación de la estación en la que debera correr cada proceso, o bien decidir cuales serán los procedimiento que seran usados por otros procesos en sus llamadas remotas.

## 1.2. EL MODELO CLIENTE-SERVIDOR

En un modelo *cliente-servidor* se tienen dos procesos (posiblemente en diferentes máquinas), uno conocido como *cliente* y el otro denominado *servidor*. El servidor contiene una serie de servicios y/o tareas, las cuales puede ofrecer al proceso cliente. Para que éste pueda hacer uso de dichos servicios, la cooperación entre un par cliente-servidor se puede describir asi.

El cliente hace una *solicitud* al servidor de alguno de los servicios que éste ofrece, y el cliente se dedica a cumplir esta solicitud recibida, mientras tanto el cliente espera bloqueado, y una vez que termina el proceso de dicha solicitud, el servidor envia de regreso la *respuesta* para que el cliente pueda continuar con su proceso normal, como puede apreciarse en la siguiente figura.

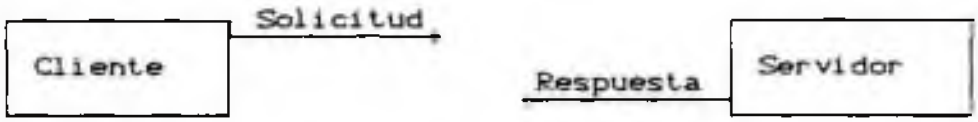



Fig donde el cliente envia una solicitud y espera por la respuesta del servidor.

Es importante aclarar que cuando el cliente envia la solicitud de servicio, éste detiene su ejecución hasta que recibe la respuesta por parte del servidor.

## 1.3. EL MODELO LLAMADA A PROCEDIMIENTO REMOTO (RPC)

Implementar un mecanisno cliente-servidor donde el cliente envia una solicitud y espera (bloqueado) una respuesta por parte del servidor, hacer que las llamadas a procedimientos contenidos en otros procesos (clientes) sean transparentes al usuario, esto es conocido como un esquema de *llamada a procedimiento remoto* (a continuación llamdo RPC);

En un esquema RPC es posible permitir mediante la comunicación entre máquinas que las llamadas a procedimientos, se hagan como si éstos procedimientos estuvieran realmente en el proceso llamador.

El compilador del lenguaje distribuido, debe ser capaz de distinguir cuando un procedimiento se encuentra dentro del proceso, o bien este procedimiento se encuentra en algun otro proceso. Cuando la llamada es local (al mismo proceso) ésta es realizada de la forma acostumbrada, colocando los parámetros en la pila y realizado la llamada al procedimiento, mediante un simple call. Cuando el procedimiento invocado esta contenido en otro proceso, el cual corre en otra máquina, los parámatros y la información derivada del proceso llamador y del proceso llamado,

son pasados al nivel de paso de mensajes entre procesos, para que este módulo (paso de mensajes) los ponga en la red, la estación destino los recoga, desempaquete los parámetros contenidos en el mensaje y realize la llamada como si esta fuera una llamada local, una vez terminado con la ejecución del procedimiento los resultados son empaquetados, puestos en la red, recibidos por el cliente, continuando este su ejecución normal, por lo que un mecanismo RPC se puede representar como lo muestra la siguiente figura.

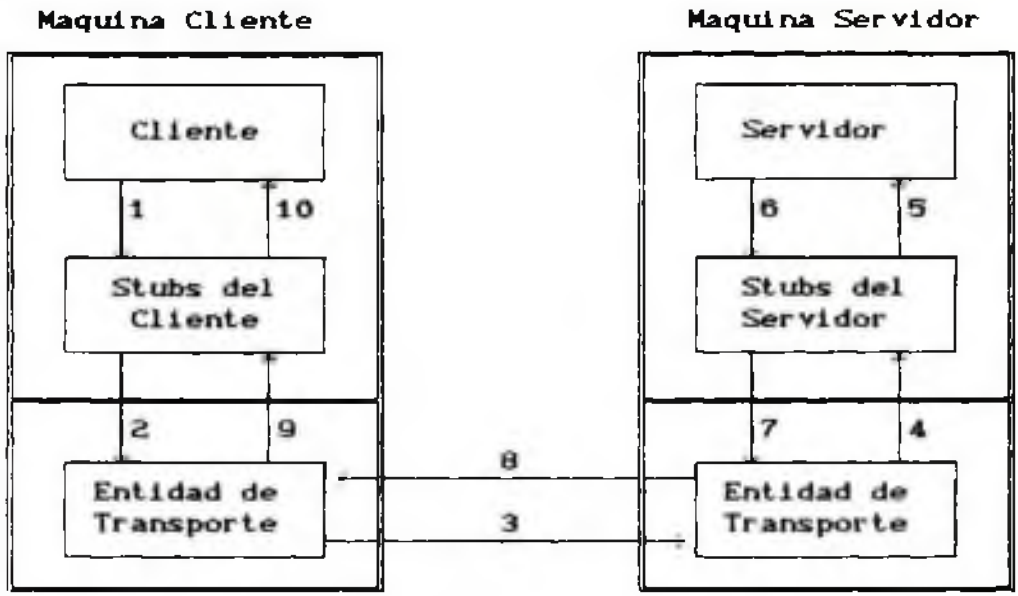



**Fig. que muestra los diez pasos para ejecutar una LLamada a Procedimiento Remoto**

Paso 1. El cliente realiza una llamada a un procedimiento, que al ser reconocido como un procedimiento remoto, el control le es dado al Stubs del clinete, mientras el cliente queda bloqueado en espera de respuesta a su solicitud.

Paso 2. Se construye el mensaje de llamada remota, empaquetando parámetros de llamada del procedimiento y estableciendo los orígenes y destinos de la llamada, para ser entregado a la entidad de transporte.

Paso 3. La entidad de transporte coloca las tramas en la red conforme le van llegando .

Paso 4. La trama es recibida de la red y entregado al stubs del servidor.

Paso 5. El stubs del servidor desempaqueta el mensaje y realiza la llamada al procedimiento remoto solicitado, como una llamada local (colocando parámetros en la pila, etc.).

Paso 6. El Servidor ejecuta el procedimiento, y una vez terminada su ejecución normal, pasa el control al stubs del servidor.

Paso 7. El Server stub construye el paquete con los resultados obtenidos , decide el destino de los resultados y pasa el paquete al Transport entity.

Paso 8. Los paquetes de respuesta son puestos en la red para la maquina cliente.

Paso 9. Los paquetes de respuesta son recibidos y entregados al stubs del cliente.

Paso 10. El stubs del cliente desempaqueta los resultados, actualizando el contenido de las variables, y pasa el control al cliente para que éste pueda continuar con su ejecución.

# EL LENGUAJE

## 2.1. INTRODUCCION

La idea y la forma en que, en computación distribuida se controlan y comparten recursos, fue tomada de los sistemas operativos, por lo que muchos de estos conceptos han sido incorporados a los lenguajes de programación distribuida. Entre las características que un lenguaje de programación distribuida debe tener, podemos enumerar:

1) La facilidad para expresar la distribución de procesos,
2) la sincronización de procesos, y
3) la comunicación entre procesos.

En este capítulo se describirán las construcciones sintácticas y las acciones semánticas más imporatantes utilizadas para los procesos distribudios en el lenguaje Polux.

## 2.2. ORIGENES Y BASES

El diseño del lenguaje Polux, llevado a cabo por el Dr. Jan Janecek Hyan, director de esta tesis, tomó como bases un

subconjunto del pascal, llamado Pascal-S del cual se tenían sus construcciones sintácticas y un compilador que produce un código intermedio, que era utilizado por un intérprete para su ejecución.

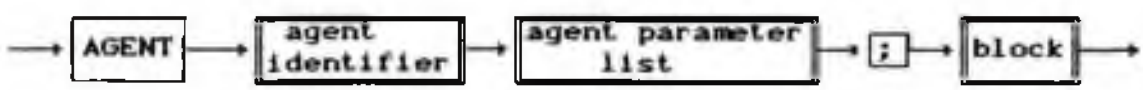
A este subconjunto de pascal le fueron añadidas algunas construcciones sintácticas orientadas hacia los procesos distribuidos para buscar un lenguaje de diseño simple, de tal forma que, a la vez de que se aprovechara un lenguaje de reconocida capacidad y claridad, al agregarle éstas nuevas construcciones, conservara sus características originales, también debia aprovecharse el compilador desarrollado para dicho subconjunto del Pascal para la inmediata implentación de sus nuevas capacidades.

## 2.3. CONSTRUCCIONES SINTACTICAS Y SEMANTICAS PARA PROCESOS DISTRIBUIDOS

### Block AGENT

Una de las primeras construcciones nuevas tanto por orden sintáctico como por importancia, el la relacionada con la declaración de bloques utilizados para la creación de procesos, que en el lenguaje Polux son conocidos como agentes, esta declaración se da en el nivel de bloques de la definición del lenguaje.

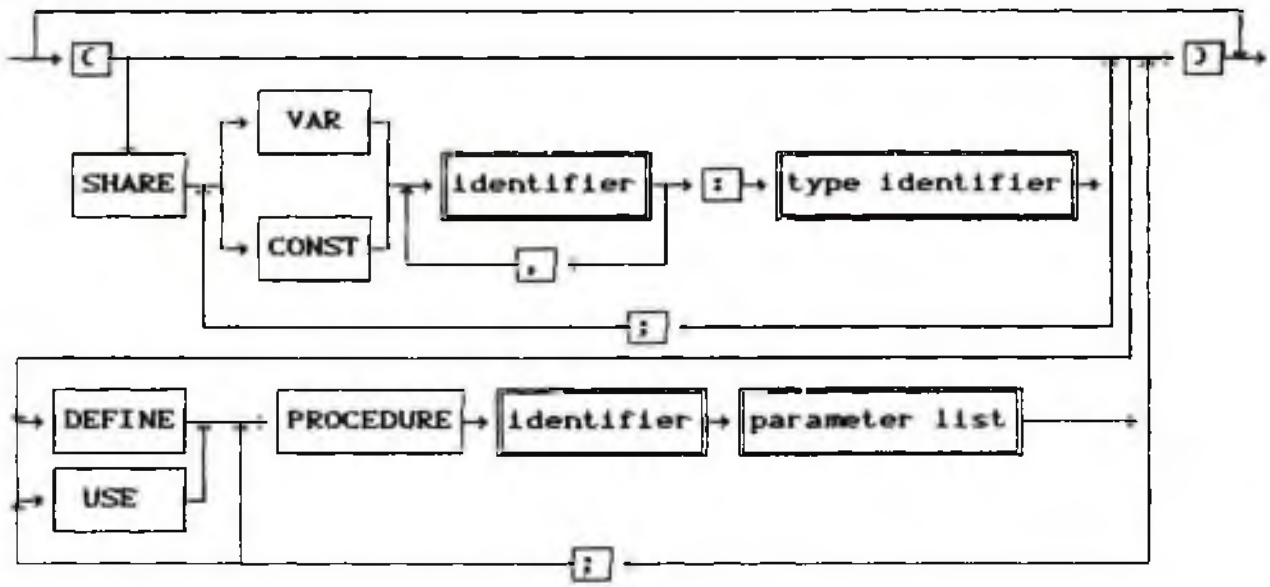
Su construcción sintáctica es la siguiente:




Con la definición de este bloque se pretende declarar un nuevo procedimiento conocido como agente, que pueda ser tratado para la declaración de procesos que correrán en sus propios ambientes de ejecución, y que a diferencia de la declaración de

procedimientos y/o funciones, su declaración de parámetros varía, por lo que la lista de parámetros de un agente se define asi:

**Agent parameter list**




En la lista de declaraciones en primer término podemos encontrar las declaraciones de parámetros, mediante SHARE, al cual pueden seguirlo VAR o CONST.

Una variable tipo SHARE VAR implica que los procesos generados con dicho agente van a compartir variables con al menos los procesos generados con dicho agente, asi como otros con los cuales pueda relacionarse la variable en cuestión. Esta situación puede considerarse similar al caso, cuando en el Pascal se pasa un parámetro por referencia (su dirección). Esto significa que cuando uno de los procesos actualiza una variable de este tipo, todos los demás procesos que tienen una copia de esta variable, deben saber que dicha variable fue actualizada y conocer su nuevo valor

Para el binomio SHARE CONST significará que tal parámetro será tratado como una varible local al proceso que la contenga y los cambios que pueda hacer con ella sólo le interesa a él mismo y a nadie más, regresando con la contraparte en Pacal, se refiere

al caso cuando se pasa el parámetro por valor.

El resto en la declaración de parametros puede observarse, que es similar al Pascal.

Después de la declaración de los parámetros variables, es posible encontrar las referencias a los procedimientos relacionados directamente con dicho agente, lo cual se hace con las palabras claves DEFINE o USE acompañadas del PROCEDURE.

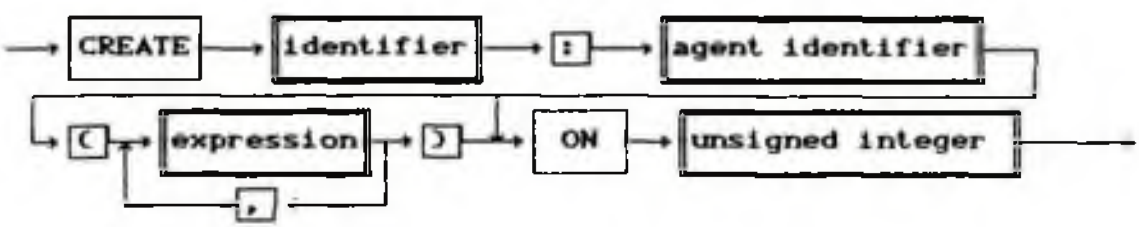
El DEFINE PROCEDURE significa que el procedimiento mencionado a continuación, estará definido dentro de dicho agente, por lo que este tipo de procedimiento está disponible tanto para el agente en cuestión, de forma directa, como para otros agentes en forma indirecta, mediante las llamadas remotas que se hagan a ellos.

Mientras que el USE PROCEDURE especifica que dicho procedimiento podrá ser usado por este agente como si tal procedimiento estuviera definido e implementado en él, cuando en realidad, tal procedimiento será definido en otro agente, por lo que obviamente su invocación generará una llamada remota a dicho procedimiento, aun encontrándose en diferente maquina.

Es importante observar que la declaración de parámetros para estos procedimientos se da tal como si fuera una declaración real del procedimieto.

**CREATE**

La siguiente construcción a considerar es la referida a la creación de procesos, la cual es tratada como una declaración dentro de la gramática de Polux, y su sintaxis es como sigue:

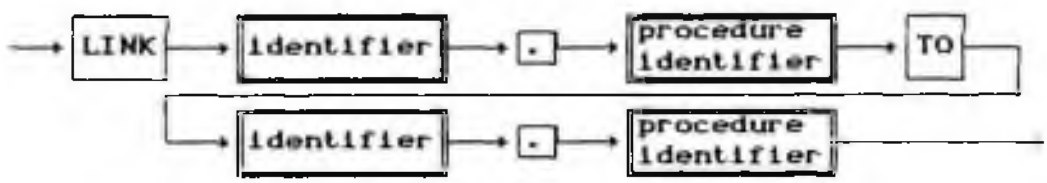


El CREATE usa un agente para crear un proceso, el cual podrá ser creado en cualquier máquina. El identificador que utiliza para el reconocimiento del proceso, no es necesario que haya sido declarado con anterioridad, pues las operaciones en que se involucra dicho identificador son minimas, si no es que nulas.

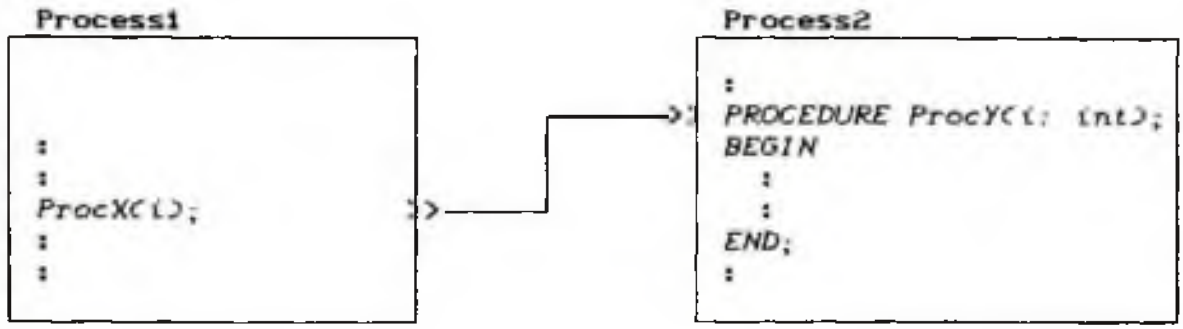
Como una acción semántica importante es que con la ejecución de este statement se construye un paquete que ira a la red, cuyo destino será la estación especificada a continuación del símbolo ON.

LINK

Otra sentencia, es la utilizada para ligar procedimientos definidos en un proceso con procedimientos usados por otros procesos, es la sentecia LINK, cuya sintaxis es la siguiente:




Consideremos el caso cuando se tiene por un lado, un proceso que tiene definido dentro de su código un procedimiento, el cual puede ser utilizado tanto por ese mismo proceso como por otros procesos ajenos, por otro lado tenemos un agente que hace uso de un procedimiento externo, que dicho agente ignora donde se encuentra definido, lo cual puede representarse asi:




Como se puede observar Process2 contiene en su código la definición e implementación del procedimiento ProcY, mientras que el proceso Process1 contiene en su código, una llamada a ProcX, cuya definición no se ecuentra en su código, lo cual involucra una llamada remota a ProcY encontrado en Process2.

Para representar dicha liga, nuestra sentencia quedaría asi:

*LINK Process1:ProcX TO process2:ProcY;*

Sólo que para lograr hacer esto, debió ser necesario que al momento de la declaración del agente usado para el proceso *Process1*, dentro de su lista de parámetros, debió especifiarce el uso del procecedimiento *ProcX*, mientras que en el agente usado para el proceso *Process2*, debió encontrarse la definición del procedimiento *ProcY*.

Como una acción semántica importante al momento de establecer la liga es el chequeo de tipos de los parametros de los procedimientos, con el objeto de corroborar la igualdad de tipos. De la misma forma, la liga real es realizada al momento de compilación.

**WHEN**

La última sentencia para los procesos distribuidos, es la utilizada con propósitos de sincronización de procesos, utilizada en forma general para la coordinación entre los procesos. Esta

sentencia es el WHEN cuya sintaxis es:


Donde *expression* es una expresión booleana, la cual al ser verdadera, se continua con la ejecución del statement que sigue al simbolo DO, de lo contrario, si es falsa, se concede el procesador al siguiente proceso en turno, y se deja el apuntador de instrucciones del proceso actual, apuntando al principio del WHEN, de tal forma que cuando este proceso vuelva a tomar el procesador, inicie su ejecución en dicho punto, y asi sucesivamente hasta que expression sea verdadera.

En realidad el WHEN, como mecanismo de sincronización se esta ejecutando indefinidamente, hasta que *expression* sea verdadera, el hecho de que en nustra implementación provoque un cambio de contexto, se debe a una púra acción semantica.

El WHEN podemos describirlo asi:

*si expression es verdadera*
*entonces ejecuta el statement*
*de lo contrario almacena el IP apuntando al principio del WHEN y realiza un cambio de contexto*

## 2.4. UN EJEMPLO

Consideremos el problema de los tres fumadores, propuesto por S. Patil en 1971, el cual se puede enunciar asi:

Considerar un sistema con tres procesos fumadores y un proceso agente. Cada fumador continuamente toma un cigarro y lo fuma. Pero para fumar un cigarro, tres ingredientes son necesarios: tabaco, papel y cerillos. Uno de los procesos tiene papel, otro tabaco y el tercero tiene cerillos. El agente tiene

una fuente infinita de los tres ingredientes. El agente coloca dos de los ingredientes en la mesa. El fumador que tiene el ingrediente restante puede hacerse y fumarse el cigarro, indicándole al agente que ya fumo. El agente entonces pone en la mesa otros dos de los ingredientes y el ciclo se repite.

Su codigo en Polux podria ser:

```
PROGRAM Smoker;

CONST   TOBACCO = 0;
        PAPER   = 1;
        MATCHES = 2;

VAR
    Ingr1,Ingr2: integer;
    table: boolean;

AGENT Smoker1(SHARE VAR tableempty: boolean;
                        Ingred1,Ingred2: integer;
              USE PROCEDURE Smoking(i: integer));     { with tobacco }
BEGIN
  WHILE TRUE DO
  BEGIN
    WHEN (NOT tableempty) AND (Ingred1 <> TOBACCO) AND
         (Ingred2 <> TOBACCO) DO
      Smoking(1);
  END; { WHILE }
END; { AGENT }

AGENT Smoker2(SHARE VAR tableempty: boolean;
                        Ingred1,Ingred2: integer;
              USE PROCEDURE Smoking(i: integer));     { with paper }
BEGIN
  WHILE TRUE DO
  BEGIN
    WHEN (NOT tableempty) AND (Ingred1 <> PAPER) AND
         (Ingred2 <> PAPER) DO
      Smoking(2);
  END; { WHILE }
END; { AGENT }

AGENT Smoker3(SHARE VAR tableempty: boolean;
                        Ingred1,Ingred2: integer;
              USE PROCEDURE Smoking(i: integer));     { with matches }
BEGIN
  WHILE TRUE DO
  BEGIN
```

```
    WHEN (NOT tableempty) AND (Ingred1 <> MATCHES) AND
         (Ingred2 <> MATCHES) DO
      Smoking(3);
  END; ( WHILE )
END; ( AGENT )

AGENT AgentManager(SHARE VAR tableempty: boolean;
                           Ingred1,Ingred2: integer;
                   DEFINE PROCEDURE Make(i: integer));

  PROCEDURE Make(i: integer);
  VAR j: integer;
  BEGIN
    WriteLn('                                                smoker',i);
    FOR j:=0 TO 10000 DO;          ( smoking )
    tableempty := TRUE;
  END; ( Make )

BEGIN
  WHILE TRUE DO
    WHEN tableempty DO
    BEGIN
      Ingred1 := random(3);
      REPEAT
        Ingred2 := random(3);
      UNTIL Ingred1 <> Ingred2;
      Write('agent supply ');
        CASE Ingred1 OF
          TOBACCO : Write('tobacco ');
          PAPER   : Write('paper ');
          MATCHES : Write('Matches ');
        END;
        Write('and ');
        CASE Ingred2 OF
          TOBACCO : Write('tobacco ');
          PAPER   : Write('paper ');
          MATCHES : Write('Matches ');
        END;
        WriteLn;
      tableempty := FALSE;
    END;
END;

BEGIN
  table := TRUE;
  CREATE ag    :AgentManager(table,Ingr1,Ingr2) ON 0;
  CREATE smok1:Smoker1(table,Ingr1,Ingr2) ON 1;
  CREATE smok2:Smoker2(table,Ingr1,Ingr2) ON 2;
  CREATE smok3:Smoker3(table,Ingr1,Ingr2) ON 3;
  LINK smok1.Smoking TO ag.Make;
  LINK smok2.Smoking TO ag.Make;
  LINK smok3.Smoking TO ag.Make;
END.
```

De entrada, se tienen cuatro agentes los cuales son utilizados para crear cuatro procesos distintos, estos son:

*Smoker1*, el cual poseee como ingredientes el papel y los cerillos.

*Smoker2*, que posee tabaco y cerillos.

*Smoker3*, que en su propiedad contiene tabaco y papel.

*AgentManager*, el cual contiene en forma indefinida los tres ingredientes, de los cuales cada vez aporta dos de ellos, para que alguno de los fumadores pueda cumplir su cometido.

Para el caso de los agentes fumadores (SmokerX), éstos tienen dentro de su lista de parámetros, tres variables compartidas (SHARE VAR), dos de tipo entero (Ingred1, Ingred2) y una detipo boolean (tableempty), asi mismo especifican el uso de un procedimiento remoto (Smoking).

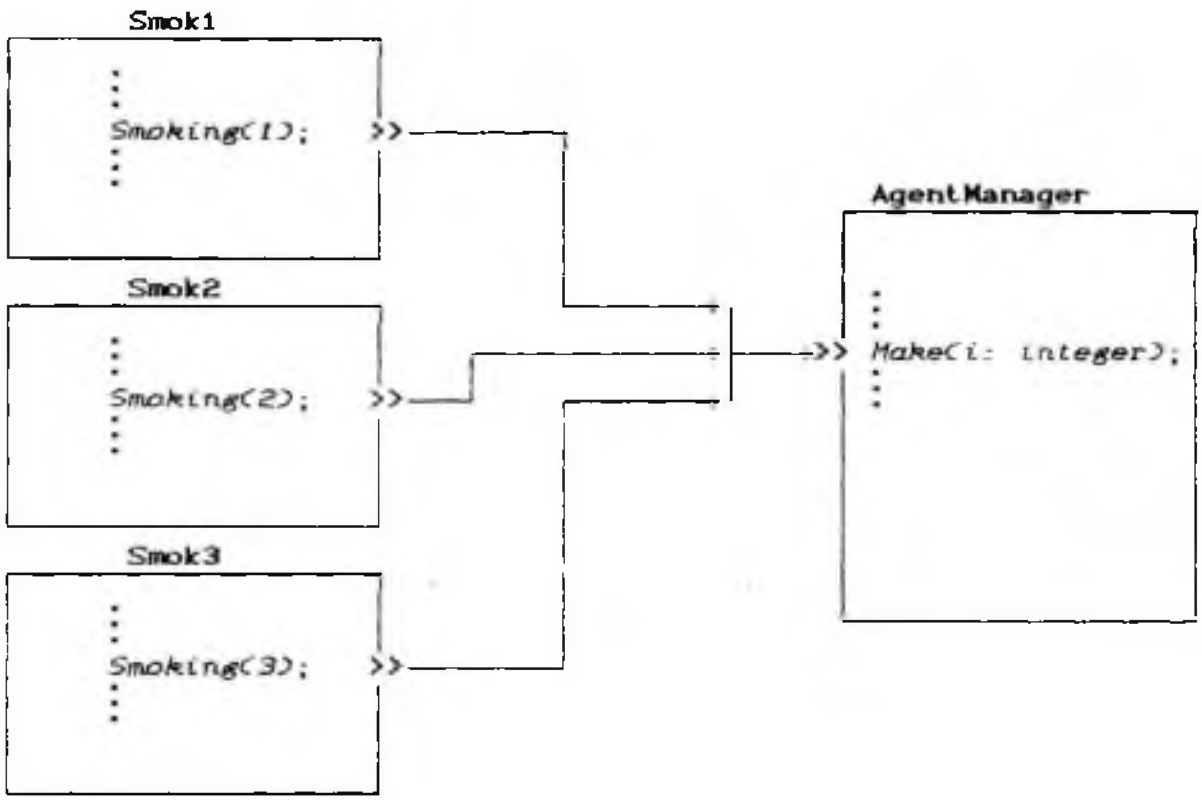
Mientras que el agente manejador (AgentManager), de la misma forma tiene dentro de su lista de parámetros, dos variables enteras y una booleana en forma compartida (SHARE VAR), y su diferencia con respecto a los agentes fumadores, es que, contiene la definición (e implementación) de un procedimiento (Make), el cual va a ser llamado en forma remota por los agentes fumadores.

En el bloque principal encontramos en primer término, las sentencias CREATE, las cuales se encargan de crear los procesos correspondientes que son: ag, smok1, smok2 y smok3, los cuales utilizan los agentes Smoker1, Smoker2, Smoker3 y AgentManager respectivamente. Observándose que cada uno de ellos correrá en máquinas diferentes (0,1,2 y 3) especificadas a continuación de ON. Al momento de la creación de los procesos, son pasados los parámetros con los valores actuales, que como se ve, únicamente la variale *table* es inicializada.

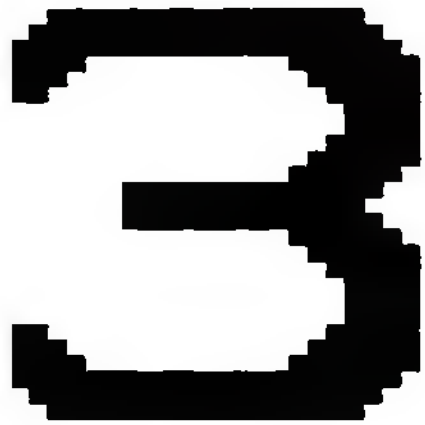
En tanto Ingr1 e Ingr2 quedan con un valor indefinido, pero lo importante aqui, es que, éstas variables (table, Ingr1 e

Ingr2) son las que sirven de base para establecer la liga de variables compartidas, por lo que al final de los CREATE, las variables tableempty, Ingred1 e Ingred2 en los procesos ag, smok1, smok2 y smok3 quedan compartidas junto con las variables del bloque principal table, Ingr1 e Ingred2.

A continuación encontramos las sentencias LINK, utilizadas para establecer las ligas de los procedimientos usados en los procesos smok1, smok2 y smok3 con el procedimiento definido en el proceso ag (Smok1.Smoking, Smok2.Smoking, Smok3.Smoking con ag.Make). esto podemos verlo asi:




En el apéndice A se muestran todos los diagramas sintácticos de Polux, así como muestra la ubicación de las anteriores construcciones dentro del lenguaje. Al inicio de ese mismo apéndice se encuentran algunas notas importantes para comprender los mismos diagramas.



# REMOTE PROCEDURE CALL (STUBS)

## 3.1. INTRODUCCION

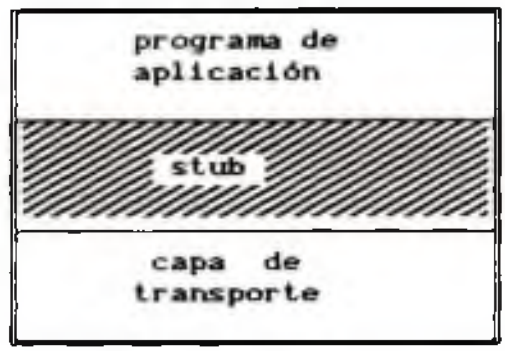
En este capitulo describiremos el módulo del sistema denominado *stubs*, cuya responsabilidad es la de recibir indicaciones del nivel superior (programa de aplicación) normalmente en forma de mensajes, organizar estos mensajes, y ponerlos en la red para la(s) estación(es) destino. Asi mismo recoger de la red las trmas dirigidos a su estación, dedicarse a su interpretación y realizar las acciones correspondientes, normalmente sobre el mismo nivel superior (programa de aplicación).

Se discutira tanto su organización lógica, asi como los detalles de implementación.

## 3.2. FUNCION DEL STUBS EN EL SISTEMA

Miremos primeramente la ubicación del módulo *stubs* dentro de la globalidad del sistema. El stub es encontrado entre las capas

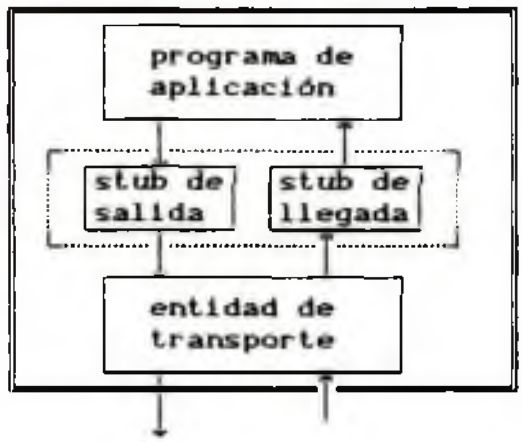
del programa de aplicación y la capa de transporte, como se muestra en la siguiente figura:




Dada la ubicación del stubs, al servir de enlace entre el programa de aplicación y la capa de transporte, éste es el encargado de obtener mensajes del programa de aplicación, generados como consecuencia de la comunicación entre procesos, por lo que el stub interpreta el tipo de mensaje recibido, hace las adecuaciones necesarias respetando las características de la red, por la que envia cada uno de los mensajes fraccionados del paquete original. De la misma forma, de manera inversa, recoje tramas del nivel de red e inicia a formar los mensajes destinados a algún proceso corriendo en su máquina, los cuales de forma similar que los mensajes de salida, son interpretados y responsables de alguna acción sobre el proceso destino.

## 3.3. ORGANIZACION DEL STUBS

Por las diferencias en operación de los mensajes de salida como de llegada, el stubs es dividido en dos partes. El stubs de salida y el stubs de llegada, esquematizado asi:




En forma general, el stubs de salida considerado de operación más sencilla que el de llegada, basa su operación en mensajes de llegada, sobre los cuales trabaja, dichos mensajes son recibidos en un buffer llamado *message* de tamaño definido al momento de compilación, dimensionado a 1024 bytes (1 KB), de acceso por byte, mostrado esquematicamente:

*message*

| 1 | 2 | 3 | 4 | . . . | 1021 | 1022 | 1023 | 1024 |
|---|---|---|---|---|---|---|---|---|

Este buffer es llenado byte por byte por el código generado para el programa de aplicación para los distintos mensajes usados. Una vez que dicho buffer es completado con algún mensaje, es pasado al stubs de salida, por lo que al ser recibido por dicho stubs, como primer paso, está el identificar el tipo de servicio solicitado, para posteriormente remitirlo a la sección correspondiente del mismo stubs, enseguida inicia el desempaquetamiento y van siendo preparadas las tramas, las cuales cada vez que una trama es completada, es pasada a la entidad de transporte para que ésta la ponga en la red. Como el tamaño de una trama con la que la red puede trabajar, es más pequeña que el buffer message, es posible que sea necesario enviar varias tramas para que el buffer quede liberado.

En el momento en que el programa de aplicación le pasa un

mensajeal stub, dicho proceso detiene su ejecución, en espera de que el stub termine de mandar el mensaje, esto es si dicha trama no requiere respuesta por parte del proceso destino. Pero si el servicio solicitado, como puede ser una llamada a un procedimiento remoto, necesita una respuesta por parte del proceso servidor, el proceso cliente debe esperar por dicha respuesta para poder continuar con su ejecución. Aunque esta respuesta no le va a llegar por el stub de salida, sino por el stub de llegada.

El stub de llegada, tiene como función la de recibir tramas del nivel de red (entidad de transporte) y realizar acciones sobre el ambiente del proceso destino. Para este estubs, cuando le llega una trama de la red, su primer paso consiste en identificar el tipo de la trama.

Algunas tramas requieren que sean almacenadas hasta que se complete el mensaje, por lo que van siendo almacenadas en un buffer auxiliar llamado *messaux*, de igual tamaño que *message*. Una vez completado el mensaje de llegada, es pasado al proceso, para que éste lo desempaquete y actualice sus variables.

Para que una trama sea recibida por este stubs, se hace por medio de pooling, y el encargado es la rutina de interrupción que atiende los ticks del reloj del sistema, de tal forma, que cuado se detecta la llegada de una trama se le pasa el control al stubs para que recoja la trama y la empaquete si es que así tiene que ser, o bién que realice las acciones adecuadas.

## 3.4. PAQUETES

Con el objeto de comunicación entre el stubs y sus capas vecinas, se han implementado algunos mensajes para los diferentes servicios necesarios para la coordinación de los procesos.

Estos mensajes son llenados por cada proceso conforme su ejecución, pues dicha conformación del mensaje forma parte del código del proceso, y una vez que el mensaje es completado, es

pasado a la capa de stubs.

Cuando se tienen que pasar valores de datos, éstos van precedidos por su tipo, los cuales pueden ser:

| Tipo | No. byte |
|---|---|
| integer | 2 |
| real | 6 |
| boolean | 1 |
| char | 1 |
| array | * |
| record | * |
| address | 2 |

A continuación se detallan los mensajes, cuyos nombres son:

1 - CCREATE
2 - CRUN
3 - CVALUE
4 - CCALL
5 - CREPLY
6 - CREQVAL

En ésta descripción de los paquetes, son mostrados, tanto los mensajes usados entre el programa y el stubs, así como las tramas que viajan por la red.

* Para el caso de registros y arreglos, su tamaño puede diferir en cada caso, pues cada uno de ellos puede estar formado por otros arreglos o registros, y estos a su vez contengan en su definición otros arreglos y registros, y así sucesivamente.

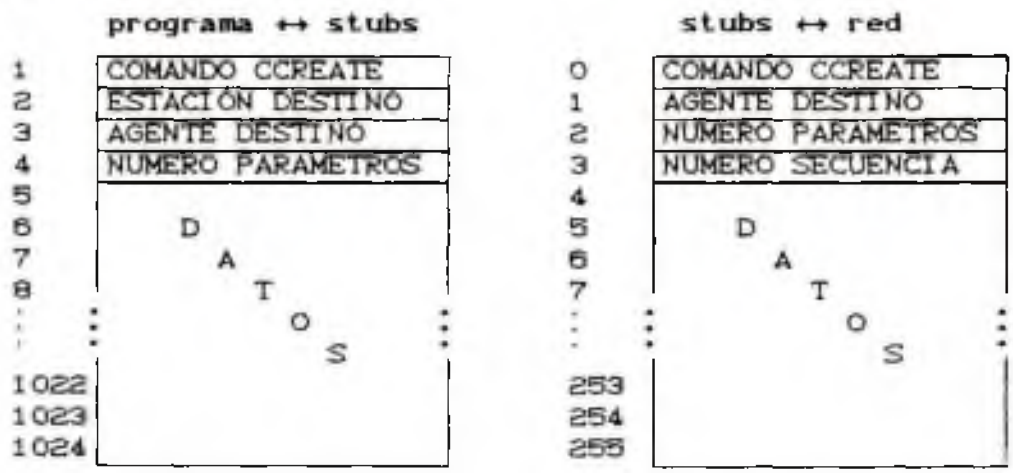
Nombre :

CCREATE

Función :

Generado al momento de ejecución de una sentencia CREATE del lenguaje. Es utilizado para la creación de los procesos en la estación especificada. En este mensaje van los valores iniciales de los parámetros especificados.

Formato :




Respuesta :

Ninguna respuesta, excepto el reconocimiento.

Nombre :

CRUN

Función :

Indicarle a un agente ya creado que puede empesar a competir por el procesador, pues lo que hace el agente destino es poner su bandera en listo, este mensaje es formado como las últimas instrucciones de un bloque donde existen las sentencias CREATE.

Formato :

| | programa ↔ stubs | | stubs ↔ red |
|---|---|---|---|
| 1 | COMANDO CRUN | 0 | COMANDO CRUN |
| 2 | ESTACION DESTINO | 1 | AGENTE DESTINO |
| 3 | AGENTE DESTINO | | |

Respuesta :

Ninguna respuesta, excepto el reconocimiento.

Nombre :

CVALUE

Función :

Paquete formado cuando una variable tipo SHARE VAR es actualizada, con lo cual en este mensaje es enviado su nuevo valor al resto de los agentes que contengan una copia de ella. O bién para las variables globales (declaradas en el nivel 0) cuando se envia un valor solicitado por algún agente.

Formato :

| | programa ↔ stubs | | stubs ↔ red |
|---|---|---|---|
| 1 | COMANDO CVALUE | 0 | COMANDO CVALUE |
| 2 | ESTACION BROADCAST | 1 | AGENTE FUENTE |
| 3 | AGENTE FUENTE | 2 | DIRECCION EN |
| 4 | DIRECCION EN | 3 | FUENTE |
| 5 | FUENTE | 4 | TIPO |
| 6 | TIPO | 5 | VALOR |
| 7 | VALOR | | |

Respuesta :

Ninguna respuesta, excepto el reconocimiento.

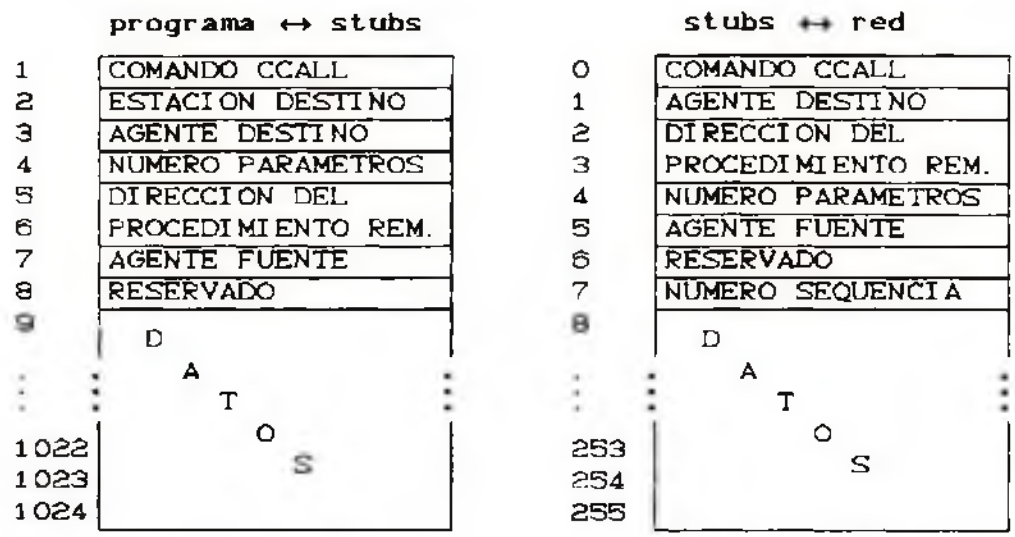
Nombre :

CCALL

Función :

Realizar una llamada remota de un procedimiento encontrado en otro proceso, en el mensaje van los valores de los parámetros transferidos.

Formato :




Respuesta :

Con este mensaje después del reconocimiento, el proceso espera por la respuesta de fin del procedimiento, junto con el cual vienen los nuevos valores de los parametros tipo VAR que el procedimiento remoto actualizo.

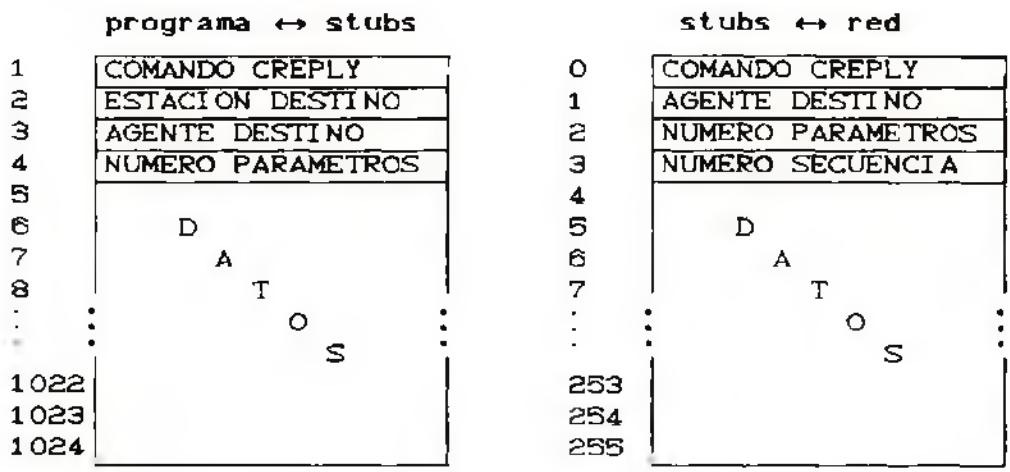
Nombre :

CREPLY

Función :

Este mensaje es construido cuando termina su ejecución un procedimiento llamado remotamente, y en él van los nuevos valores de los parámetros tipo VAR, este mensaje va dirigido al proceso que realizó la llamada. Si el procedimiento no tiene parámetros VAR, el mensaje se manda sin valores, sólo para indicar que el procedimiento terminó su ejecución.

Formato :




Respuesta :

Ninguna respuesta, excepto el reconocimiento.

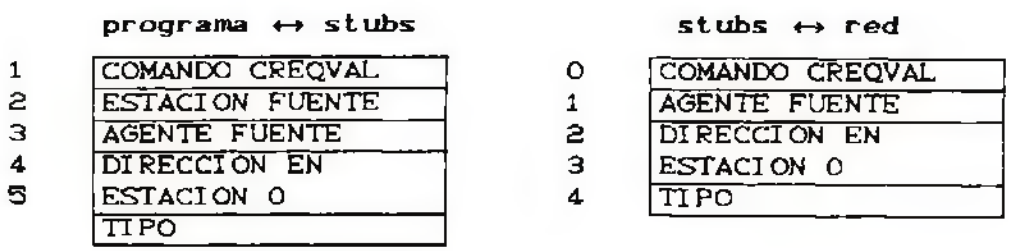
Nombre :

CREQVAL

Función :

Solicitar el contenido de una variable global (declarada en el nivel 0 lexicográfico), la cual siempre se encontrará en la estación 0.

Formato :




Respuesta :

Ninguna respuesta excepto el reconocimiento.

## 3.5. IMPLEMENTACION

Para su implementación fue desarrollado en Computadores personales IBM PC/XT/AT compatibles, que usan los microprocezadores de Intel: 8086,8088 o 80286. El sistema corre bajo el sistema operativo MS-DOS de Mricrosoft.

Todo el sistema esta desarrollado en el lenguaje de alto nivel Pascal, usando el compilador Turbo Pascal version 5.0.

A continaución analizaremos los detalles de implementación com son los mecanismos de comunicación y coordinación, estructuras, manejo de memoria, etc.

### 3.5.1. Tabla de procesos

En cada estación donde este corriendo un proceso, existirá una tabla de procesos (idéntica en todas), la cual contiene información relativa a cada proceso.

Esta tabla se encuentra en la zona de variables globales del sistema (direccionada mediante el DS).

Los campos de la tabla quedan definidos mediante la declaración:

```
AgTab: ARRAY[0..pmax] OF RECORD
           name    : alfa;
           station : byte;
           code    : word;
           stack   : pointer;
           StShar  : ListAgent;
           ProcUse : ListProc;
       END;
```

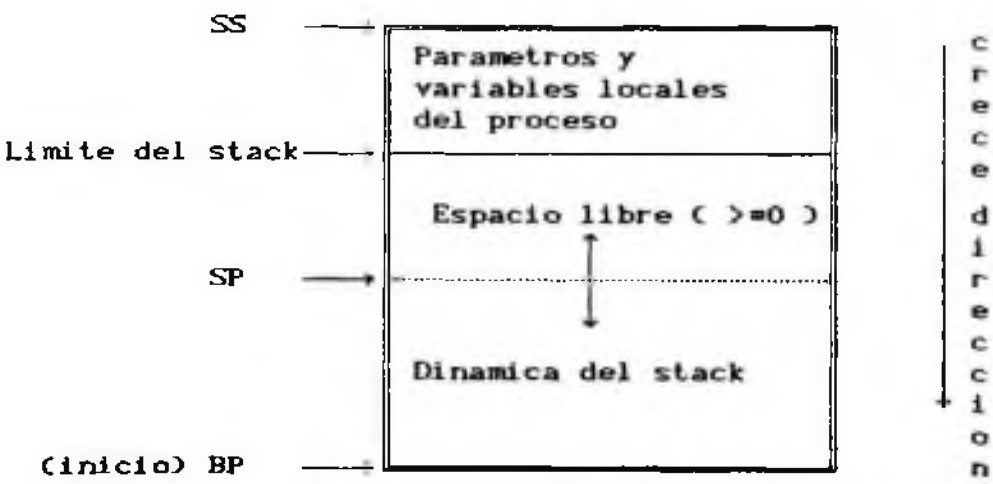
donde:

*name* - contiene el nombre del proceso.
*station* - indica en que estación se encuentra el proceso.
*code* - contiene la dirección relativa del inicio del código.
*stack* - apunta al principio de la pila.
*StShar* - puntero a su lista de variables compartida.
*ProcUse* - puntero a su lista de procedimientos definidos.

### 3.5.2. Medios ambientes de los procesos

Al momento de crear un proceso, se gestiona memoria donde éste correrá (llamada medio ambiente del proceso), esta memoria es gestionada en forma dinámica (del Heap del sistema) y su tamaño queda definido al momento de compilación, mediante la constante *stacksize* encontrada en el archivo de declaraciones *PDECTION.PAS*.

En esta pila de proceso, su parte alta (direcciones más bajas) es usada para los parámetros y variables locales del proceso (agente), mientras la parte baja (direcciones más altas) es usada para la dinámica da la pila. esquematicamente tenemos:




Para el acceso a los parámetros y variables locales del

proceso, siempre se usa el Stack Segment (SS) para direccionarlos, mientras el Data Segment (DS) es usado (nunca cambiado) para las variables globales del sistema, y del programa en caso de la maquina cero.

### 3.5.3. Variables compartidas (SHARE VAR)

Las variables declaradas en la lista de parámetros de los agentes tipo SHARE VAR, es una forma de compartir datos comunes entre los procesos (almacenadas al principio del stack), y para asegurar su globalidad, se han implemantado algunos mecanismos que permiten la identificación y el cambio de sus contenidos. Para esto, cada proceso que tiene variables compartidas con otros, cuenta con una lista (ligada) de cada una de los procesos y las direcciones de sus variables, incluyendo la suya. En esta lista se tiene al proceso con el que comparte variable y la dirección de dicha variable en su medio ambiente. La estructura que define esto es:

```
ListShare = ^Share;
Share     = RECORD
              Agent: byte;
              adr  : word;
              Next : ListShare;
            END;
```

y la estructura que liga cada proceso con su lista compartida es:

```
ListAgent = ^Agent;
Agent     = RECORD
              varsahr: ListShare;
              Next   : ListAgent;
            END;
```

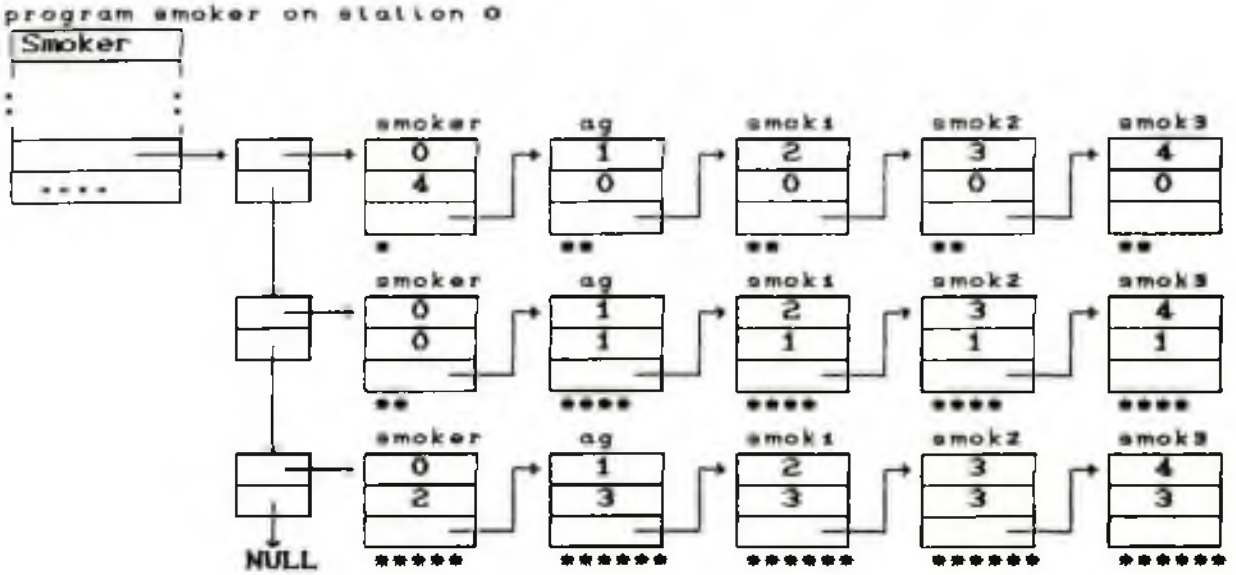
Para el caso del ejemplo mostrado en el capitulo 2 (los fumadores) como un clásico de un productor y tres consumidores, donde la declaración de sus agentes se da asi:

```
AGENT Smoker1(SHARE VAR tableempty: boolean;
              SHARE VAR Ingred1,Ingred2: integer;
              USE PROCEDURE Smoking(i: integer));     ( with tobacco .

AGENT Smoker2(SHARE VAR tableempty: boolean;
              SHARE VAR Ingred1,Ingred2: integer;
              USE PROCEDURE Smoking(i: integer));     ( with paper )

AGENT Smoker3(SHARE VAR tableempty: boolean;
              SHARE VAR Ingred1,Ingred2: integer;
              USE PROCEDURE Smoking(i: integer));     ( with matches .

AGENT AgentManager(SHARE VAR tableempty: boolean;
                   SHARE VAR Ingred1,Ingred2: integer;
                   DEFINE PROCEDURE Make(i: integer));
```

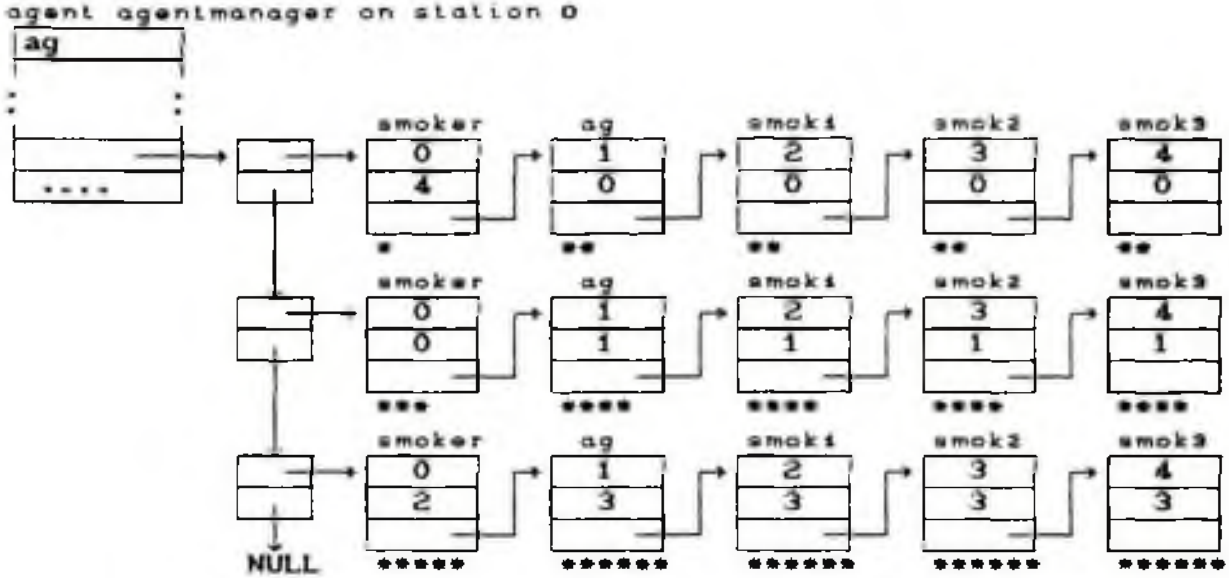
y la creación de los procesos de la siguiente forma:

```
  CREATE ag    :AgentManager(table,Ingr1,Ingr2) ON 0;
  CREATE smok1:Smoker1(table,Ingr1,Ingr2) ON 1;
  CREATE smok2:Smoker2(table,Ingr1,Ingr2) ON 2;
  CREATE smok3:Smoker3(table,Ingr1,Ingr2) ON 3;
```

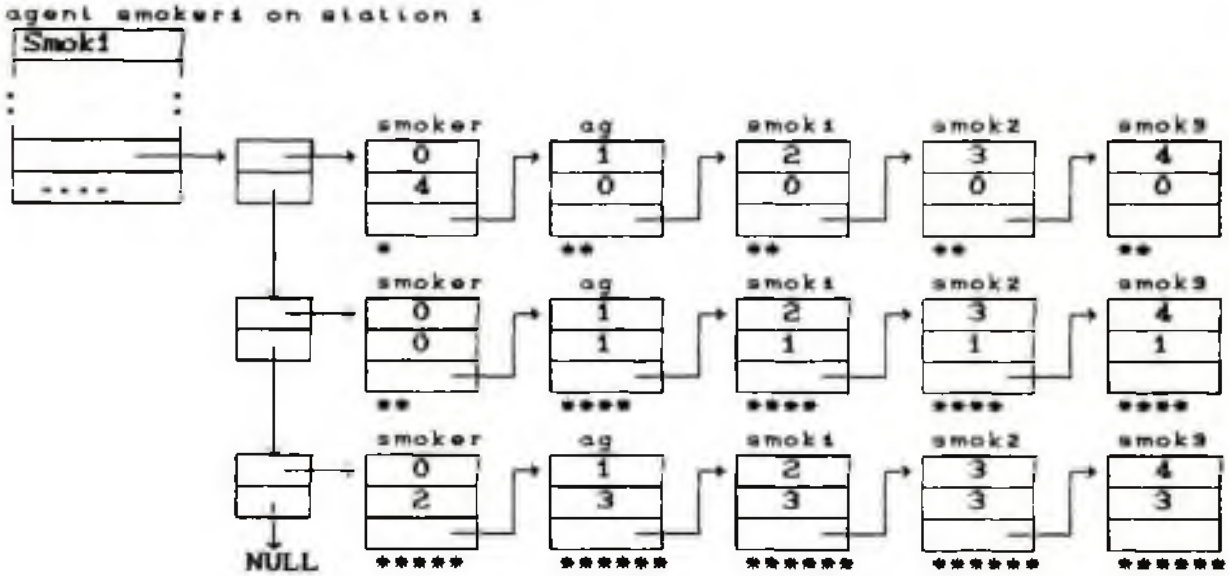
Tendremos que la tabla de procesos quedaria asi:

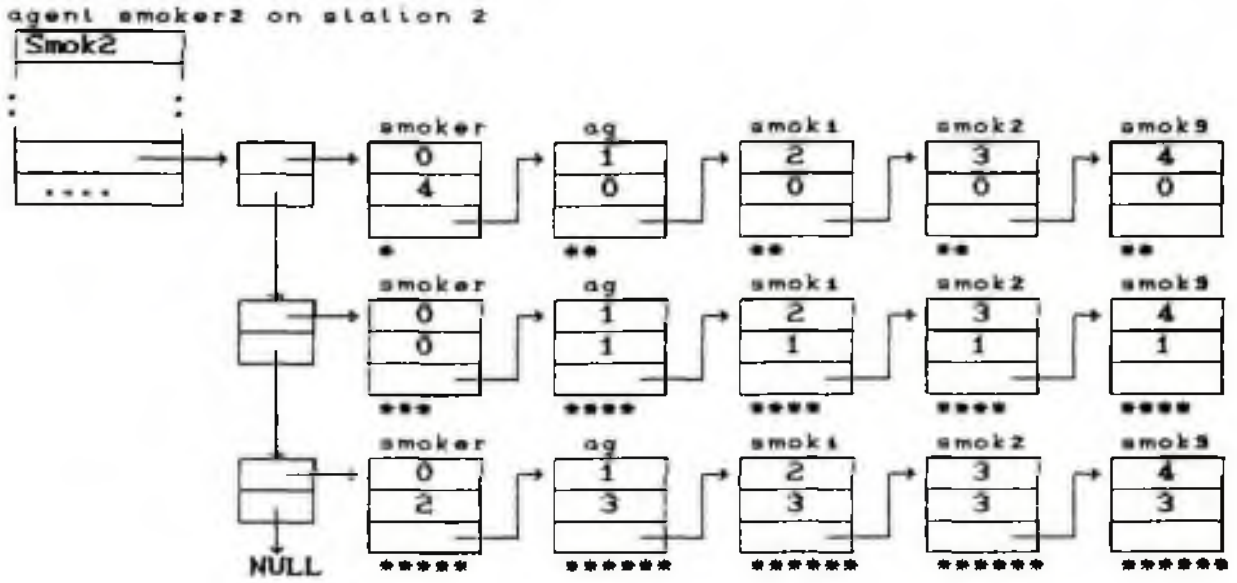


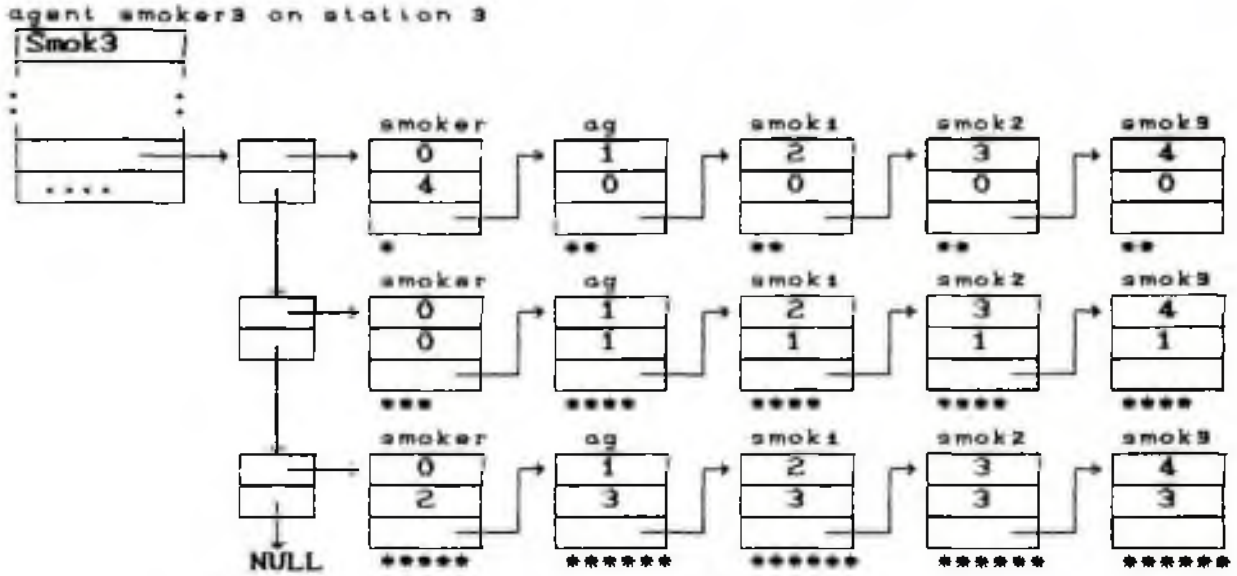
agent agentmanager on station 0




agent smoker1 on station 1






* tabi

** tableempty

*** ingr1

**** ingred1

***** ingr2

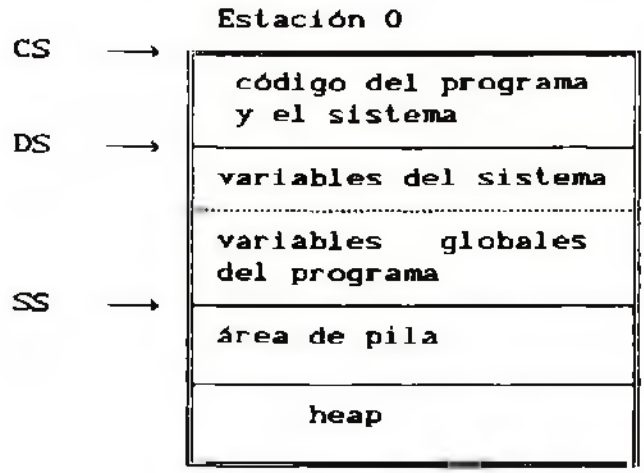
****** ingred2

Para que una variable compartida pueda ser actualizada por todos los procesos que la comparten, cuando uno de ellos la modifica, es necesario hacerlo mediante el operador de asignación (:=), de tal forma que si una variable de este tipo es actualizada por otro medio, como pasada por referencia a un procedimiento, los otros proceso no podrán enterarse de este cambio, por lo que es necesario que si una de estas variables necesita ser actualizada y el resto de los procesos lo registren (necesitan registrarlo), es indispensable que se haga mediante asignación.

Cuando una variable es actualizada, se forma un mensaje CVALUE y este se manda a todas las estaciones y a todos los procesos, de tal forma que el stubs de llegada de cada estación, busca para cada una de sus listas de variables de sus procesos contenidos, el identificador de la variable modificada. Su identificación se da mediante el número de proceso y su dirección, una vez que localiza su identifcación, busca la localidad donde reside en su pila dicha copia de la variable, la cual se encuentra en la misma lista, y actualiza su valor.

### 3.5.4. Variables globales

Son aquellas declaradas en el nivel 0 lexicográfico (únicamente), y son localizadas en la zona de variables globales del sistema, pero únicamente contenidas en la estación 0. Su ubicación es:

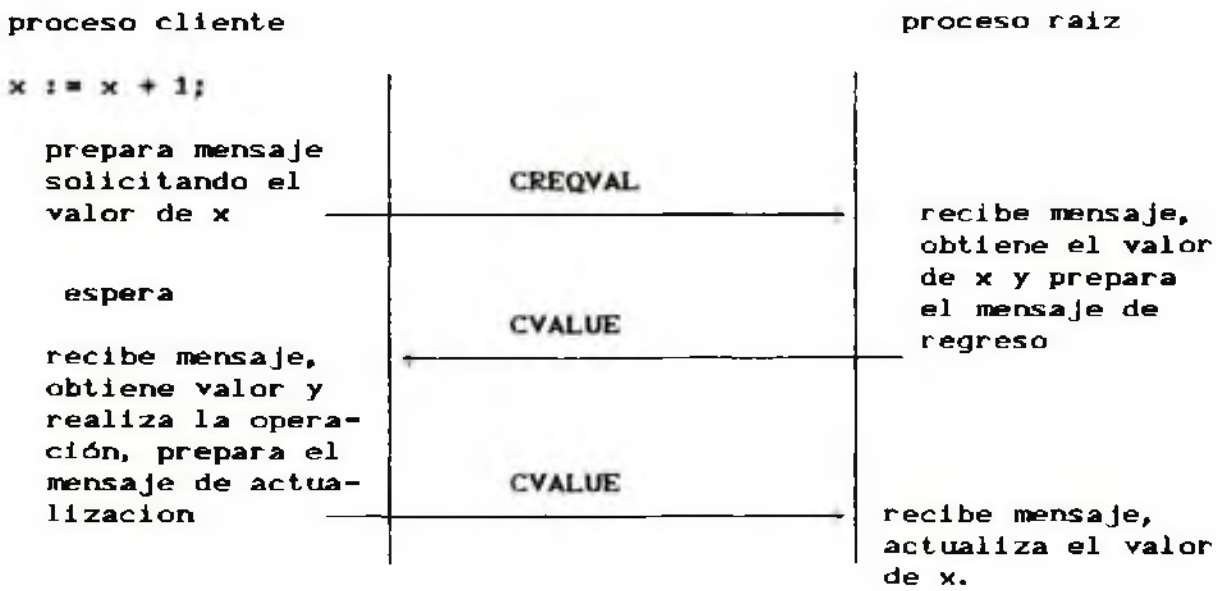



Cuando un proceso que no sea el programa principal, requiere el contenido de una variable global, prepara un mensaje CREQVAL solicitando el valor de dicha variable. Al llegarle al stubs de la estación 0 esta solicitud, el stubs obtiene el valor de la variable, prepara un mensaje CVALUE y lo manda a la estación que lo solicita. De la misma forma que las variables compartidas, para hacer que una actualización de su valor se refleje en la estación 0 , es necesario hacerlo mediante el operador de asignación.

Consideremos a x una variable global, y algún proceso diferente al principal hace uso de ella en una expresión

```
x := x + 1;
```

este podemos representarlo asi:




### 3.5.5. Constantes (SHARE CONST)

Este tipo de variables, solamente residen en el medio ambiente del proceso que las contiene, y su actualización no genera ningún cambio en ningún otro proceso. Al momento de que son creados los procesos, estas variables reciben su valor inicial, y si bién pueden variar su contenido, sólo dicho proceso es actualizado.

### 3.5.6. Procedimientos remotos

Con el propósito de dar servicio a las llamadas remotas de procedimientos, cada proceso debe tener la información necesaria para saber a quien solicitarle la ejecución de un procedimiento remoto, por lo que cada proceso que hace uso de procedimientos remotos, cuenta con una lista de todos aquellos procedimientos de los que hace uso. De ésta forma cuando un proceso solicita al stubs el llamado remoto de un procedimiento, el stubs obtiene de su lista toda la información necesaria para completar el mensaje CCALL y enviar el mensaje al proceso que contenga dicho

procedimiento.

La estructura declarada que define a dicha lista es:

```
ListProc = ^Proc;
Proc     = RECORD
               procuse : integer;
               agentdef: byte;
               addrdef : word;
               next    : ListProc;
           END;
```

donde:

*procuse* - identifica al procedimiento.

*agentdef* - especifica que agente lo tiene definido.

*addrdef* - contiene la dirección del código del procedimiento.

*next* - puntero al siguiente procedimiento en la lista.

Cuando un agente hace una llamada remota de un procedimiento éste, queda en espera de que se ejecute el procedimiento y le llegue la respuesta, por lo que es necesario que el proceso sea bloqueado y deje el procesador a otro proceso que pueda hacer uso de él, de tal manera que cuando llega la respuesta del procedimiento ejecutado (mensaje CREPLY), dicho proceso es desbloqueado y puede competir nuevamente por el procesador.

## 3.6. TIEMPO DE EJECUCION (RUNTIME)

A continuación se describirán los detalles relacionados con el tiempo de ejecución (runtime)

### El núcleo del sistema

El sistema cuenta con un pequeño núcleo, el cual se encarga del manejo de los procesos. Para iniciar su análisis, veremos

primero, la estructura que define el descriptor de los procesos:

```
wspoint = ^wspace;
wspace  = RECORD
     (00)   next: wspoint;
     (04)   ipcnt, childs: integer;
     (08)   regs : reg86;
     (24)   s: pointer;     { stack area }
     (28)   status,procexe: boolean;
     (2A)   AgCurr: byte;
     (2B)   ListProc: procspc;
          END ;
```

Donde:

*next* : apunta al siguiente proceso.

*childs* : indica si "cuelgan" de él algunos otros procesos.

*regs* : contiene cuando esta suspendido, el estado de sus registos. donde reg86 esta declarada asi:

```
reg86 = record
          AX, BX, CX, DX, SP, BP, SI,
          DI, IP, CS, DS, SS, ES, Flags: word;
          END;
```

*s* : apunta al medio ambiente de ejecución del proceso.

*status* : indica si el proceso esta bloqueado o listo para correr, que son los unicos estados posibles.

*procexe* : indica si el proceso está ejecutando su código o está ejecutando un procedimiento llamado remotamente.

*AgCurr* : contiene el identificador del proceso.

*ListProc*: contiene una lista de procedimientos que han sido llamados remotamente y esperan ser ejecutados. Los cuales son encolados en la siguiente estructura:

```
procspc = ^pspace;
pspace  = RECORD
            regs : reg86;
            pp:  pointer;       {pointer to work data}
            next: procspc;      {next procedure wait }
          END;
```

Todos los procesos (listos o bloqueados) se encuentran en

una misma lista, la cual atiende el núcleo, el estado del proceso se indica en el campo status, que es el que mira el despachador para poder disponer del proceso. Cuando el despachador se encuentra con un proceso bloqueado, lo ignora y pasa al siguiente en la lista. La estructura que define la cola de procesos para el despachador es la siguiente:

```
pqueue  = RECORD
            first: wspoint;
            last:  wspoint;
            cnt:   word
          END;
```

Cuando un proceso tiene encolados en su descriptor, procedimientos remotos, primero atiende éstos, antes de ejecutarse el mismo.

Para que un proceso pueda atender llamadas remotas a sus procedimientos, el proceso tiene que estar activo, es decir, que no haya terminado su ejecución y de esa manera salga de la cola del despachador

# 4

# EL COMPILADOR

## 4.1. INTRODUCCION

En éste capitulo se describirá los aspectos más importantes del compilador, como son tablas, generación de código, etc.

EL compilador usa el método de decenso recursivo generalmente usado en compiladores con propósito educativo, en el que se encuentra la siguiente estructura de su parser de compilación:

```
Block
    ...
    Funciones auxiliares
    ...
  typ
    ArrayTyp
  ParameterList
  AgentParList
  ConstantDeclaration
  TypeDeclaration
  VariableDeclration
  ProcDeclaration
  AgentDeclaration
  Statement
    Selector
    CreateStatement
    LinkStatement
    Call
    ResultType
```

```
Expression
   SimpleExpression
      Term
         Factor
            StandFct
Assignment
CompoundStatment
IfStatement
WhenStatement
CaseStatement
   CaseLabel
   OneCase
RepeatStatement
WhileStatement
ForStatement
StandProc
```

## 4.2. TABLAS

El compilador da como resultado un conjunto de tablas, las cuales son usadas en el mismo proceso de compilación así como para los pasos siguientes. Las tablas generadas son :

1. de identifiacdes,
2. de bloques,
3. de arreglos y
4. de agentes.

Asi mismo genera arreglos:

5. de cadenas,
6. de reales y
7. de código.

### 4.2.1. Tabla de identificadores (Tab)

Empesemos a describir la tabla de identificadores, que es en la que se almacenan todos los identificadores (no simbolos

del lenguaje) encontrados en el programa, excepto los identificadores de procesos (los cuales se almacenan en la tabla de agentes). El registro que declará sus campos es:

```
Tab : ARRAY[0..tmax] OF
      PACKED RECORD
          name   : alfa;
          link   : index;
          obj    : object;
          typ    : types;
          ref    : index;
          normal: boolean;
          lev    : 0..lmax;
          adr    : integer;
          offs   : word;
          stk    : word;
          share  : ListShare;
      END;
```

donde:

*name* - contiene el nombre del identificador, de 1 a 10 caracteres (alfa).

*link* - establece la liga con aquellos identificadores de su mismo nivel de bloque, de aparición anterior

*obj* - especifica el tipo de objeto (constante, variable, tipo, precedimiento, función o agente).

*typ* - especifica el tipo al que pertenece el identificador (no-tipo, entero, real, lógico, carácter, arreglo, registro, agente o dirección).

*ref* - establece una referencia a alguna otra tabla, por ejemplo, los objetos tipo registro, función o procedimiento tienen una referencia a la tabla de bloques, o bién, un tipo arreglo tiene una referencia a la tabla de arreglos.

*normal* - normalmente usado en parámetros o de agentes o procedimientos, para indicar si es parámetros pasado por referencia o por valor etc.

*lev* - indica el nivel de profundidad en el nivel léxico.

*adr* - indica la dirección relativa del identificador (ya no usado)

*offs* - indica el desplazamiento relativo en su medio ambiente de ejecución.

*stk* - indica el desplazamiento relativo ya sumado dicho identificador al medio ambiente de ejecución.

*share* - lista para aquellos identificadores tipo variables que son compartidas entre los agentes.

### 4.2.2. Tabla de bloques (BTab)

Utilizada para establecer la iformación relativa a registros, procedimientos y funciones, donde su registro de campos es el siguiente:

```
BTab : ARRAY[0..bmax] OF
       PACKED RECORD
          link   : index;
          last   : index;
          lastpar: index;
          psize  : index;
          vsize  : index;
          adr    : integer;
          bytsiz: integer;
          def    : boolean;
       END;
```

donde:

*link* - establece una liga a otros elementos de la misma tabla, que tienen un nivel lexicográfico menor, del cual él depende.

*last* - apunta a la tabla de identificadores, al último identifcador de dicho bloque.

*lastpar* - apunta a la tabla de identificadores, al último parámetro del procedimiento, función o agente.

*psize* - usado por funciones, procedimientos y agentes para especificar el desplazamiento en su medio ambiente hasta su último parámetro.

*vsize* - usado tanto por registros como por bloques de código.

*adr* - usado por funciones, procedimientos y agentes para

indicar la dirección del inicio de su código.

*bytsiz* - usado por registros para especificar su tamaño en bytes.

*def* - usado por procedimientos para indicar si es un procedimiento definido o sólo usado.

### 4.2.3. Tabla de arreglos (ATab)

Usada para guardar la información relativa a los arreglos. Su registro de campos es el siguiente:

```
ATab : PACKED RECORD
          inxtyp : types;
          eltyp  : types;
          elref  : index;
          low    : index;
          high   : index;
          elsize : index;
          size   : index;
          bytsiz : index;
       END;
```

donde:

*inxtyp* - especifica el tipo de dato del indice del arreglo.

*eltyp* - especifica el tipo de dato de los elementos del arreglo.

*elref* -

*low* - contiene el menor indice con que se puede indexar el arreglo.

*high* - contien el mayor indice con que se puede indexar el arreglo.

*elsize* - tamaño en número de componentes, de cada elemento del arreglo.

*size* - tamaño en componentes de todo el arreglo.

*bytsiz* - tamaño en bytes de todo el arreglo.

### 4.2.4. Tabla de agentes (AgTab)

Usada para conservar la información relativa a los procesos creados en el sistema, cabe recordar que esta tabla es la única que se conserva (casi integra) junto con el código objeto (pues forma parte de él). Su definición de campos se da asi:

```
AgTab : ARRAY[0..pmax] OF
        PACKED RECORD
          name    : alfa;
          station : byte;
          ref     : index;
          stack   : word;
          code    : word;
          StShar  : ListAgent;
          ProcUse : ListProc;
        END;
```

donde:

*name* - contiene el nombre del proceso (de 1 a 10 caracteres)

*station* - indica en que estación se encuentra el proceso.

*ref* - usado sólo al momento de compilación, para apuntar a la tabla de bloques, indicando el agente usado para dicho proceso.

*stack* - puntero al medio ambiente donde corre el proceso.

*code* - puntero al inicio del código usado por el proceso.

*StShar* - lista de procesos con los que comparte variables.

*ProcUse* - lista de los procedimientos remotos de los que puede hacer uso.

## 4.3. CODIGO INTERMEDIO (P-CODE)

Como resultado de compilación se obtiene un código intermedio comunmente llamado P-Code, el cual es dejado en un arreglo de código definido asi:

```
Code : ARRAY[0..cmax] OF
       PACKED RECORD
          f : -omax..+omax;
          x : -bmax..+bmax;
          y : -nmax..+nmax;
       END;
```

Donde *f* contiene el código, mientras los campos *x,y* especifican parámetros de operación de dicho código, éstos parametros son utilizados en forma distinta por los diferentes códigos.

### 4.3.1. Descripción de los P-Code

Número : 0
P-Code : LADR
Función : mete en la pila la dirección contenida en y.

Número : 1
P-Code : LVAL
Función : mete en la pila el valor en y (integer, char o byte).

Número : 2
P-Code : LIND
Función : mete en la pila el valor indirecto.

Número : 8
P-Code : ARIT
Función : evalua la función del sistema contenida en y.

Número : 9
P-Code : OFFS
Función : le suma al tope de la pila el offset y.

Número : 10
P-Code : JMP
Función : salto incondicional a y.

Número : 11
P-Code : CJMP
Función : salto condicional a y si el tope de pila es verdadero.

Número : 12
P-Code : SWCH
Función : evalúa para el case y

Número : 14
P-Code : FOR1
Función : evalúa para entrar por primera vez al FOR-TO.

Número : 15
P-Code : FOR2
Función : evalúa para ver si puede seguir con el FOR-TO.

Número : 16
P-Code : FOR3
Función : evalúa para entrar por primera vez al FOR-DOWNTO.

Número : 17
P-Code : FOR4
Función : evalúa para ver si puede seguir con el FOR-DOWNTO.

Número : 19
P-Code : CALL
Función : hace una llamada a un procedimiento o función local.

Número : 20
P-Code : INXS
Función : checa el índice del arreglo y.

Número : 21
P-Code : INXL

Función : checa el índice del arreglo y.

Número : 22
P-Code : LDBL
Función : carga bloque de dirección fuente al tope de la pila.

Número : 23
P-Code : CDBL
Función : copia bloque de fuente a destino.

Número : 24
P-Code : LCON
Función : mete en la pila la constante y.

Número : 25
P-Code : LREA
Función : mete en la pila la constante real apuntada por y.

Número : 26
P-Code : FLOT
Función : convierte al tope de la pila de entero a real.

Número : 27
P-Code : READ
Función hace un read para la dirección del tope de la pila.

Número : 28
P-Code : WRTS
Función : imprime una cadena apuntada por y.

Número : 29
P-Code : WRT1
Función : imprime con formato libre.

Número : 30

P-Code : WRT2
Función : imprime con formato especificado.

Número : 31
P-Code : FIN
Función : termina la ejecución de un proceso.

Número : 32
P-Code : EXIP
Función : retorna un procedimiento

Número : 33
P-Code : EXIF
Función : retorna una función

Número : 34
P-Code : LD
Función : carga el valor de la dirección especifacada en pila.

Número : 35
P-Code : NOT
Función : negación lógica del tope de la pila.

Número : 36
P-Code : MIN
Función : negación aritmética del tope de la pila.

Número : 37
P-Code : WRTR
Función : imprime real con formato especificado.

Número : 38
P-Code : ST
Función : almacena el tope de la pila en la dirección bajo el tope de la pila.

Número : 39
P-Code : REQ
Función : operador relacional "=" real.

Número : 40
P-Code : RNEQ
Función : operador relacional "<>" real.

Número : 41
P-Code : RLT
Función : operador relacional "<" real.

Número : 42
P-Code : RLE
Función : operador relacional "<=" real.

Número : 43
P-Code : RGT
Función : operador relacional ">" real.

Número : 44
P-Code : RGE
Función : operador relacional ">=" real.

Número : 45
P-Code : IEQ
Función : operador relacional "=" entero.

Número : 46
P-Code : INEQ
Función : operador relacional "<>" entero.

Número : 47
P-Code : ILT
Función : operador relacional "<" entero.

Número : 48
P-Code : ILE
Función : operador relacional "<=" entero.

Número : 49
P-Code : IGT
Función : operador relacional ">" entero.

Número : 50
P-Code : IGE
Función : operador relacional ">=" entero.

Número : 51
P-Code : OR
Función : operador "o" lógico (or).

Número : 52
P-Code : I+
Función : operador aritmético entero "+".

Número : 53
P-Code : I-
Función : operador aritmético entero "-".

Número : 54
P-Code : R+
Función : operador aritmético real "+".

Número : 55
P-Code : R-
Función : operador aritmético real "-".

Número : 56
P-Code : AND

```
Función : operador "y" lógico (and).

Número  : 57
P-Code  : I*
Función : operador aritmético entero "*".

Número  : 58
P-Code  : IDIV
Función : operador aritmético entero "div"

Número  : 59
P-Code  : IMOD
Función : operador aritmético entero "mod".

Número  : 60
P-Code  : R*
Función : operador aritmético real "*"

Número  : 61
P-Code  : R/
Función : operador aritmético real "/"

Número  : 62
P-Code  : EOF
Función : mira por un fin de archivo.

Número  : 63
P-Code  . WRLN
Función : genera un salto de linea en la salida (writeln).

Número  : 66
P-Code  : SWCX
Función : cambio de contexto condicional

Número  : 67
```

P-Code : FSTK
Función : libera el medio ambiente del proceso.

Número : 71
P-Code : CLRM
Función : borra el buffer message y pone su puntero en y.

Número : 72
P-Code : PBYT
Función : mete en el buffer message la constante byte de y.

Número : 73
P-Code : PINT
Función : mete en message la constante entera y.

Número : 74
P-Code : PREA
Función : mete en message la constante real apuntada por y.

Número : 75
P-Code : PBLK
Función : mete en message el bloque especificado.

Número : 76
P-Code : PADR
Función : mete en message la dirección contenida en y.

Número : 77
P-Code : PVAL
Función : mete en message el tope de la pila.

Número : 78
P-Code : PCOM
Función : mete en message el comando y.

Número : 79
P-Code : PSTA
Función : mete en message la estación contenida en y.

Número : 80
P-Code : PAGE
Función : mete en message el agente contenido en y.

Número : 81
P-Code : PNPA
Función : mete en message el número de parámetros contenido en y.

Número : 82
P-Code : SMES
Función : pasa el buffer message al stubs.

Número : 83
P-Code : PTYP
Función : mete en message el tipo especificado en y.

Número : 84
P-Code : PSTK
Función : mete en message el último valor sacado de la pila.

Número : 85
P-Code : PAEM
Función : mete en message el agente que emite mensaje.

Número : 86
P-Code : CALR
Función : envia message al stub para que haga una llamada remota.

Número : 87
P-Code : PVEX
Función : mete en message el tope de pila resultado de expresión.

Número : 88
P-Code : RPLY
Función : envia respuesta a llamada de procedimiento remoto.

Número : 89
P-Code : GADR
Función : saca dirección y valor de buffer y actualiza localidad.

Número : 90
P-Code : SWPR
Función : cambio de contexto condicional en sentencia WHEN, para el caso de procedimientos llamados remotamente.

## 4.4. CODIGO OBJETO

Un programa Polux en código objeto, estará formado por tres módulos que son:

- Módulo del programa Polux.
- Módulo del stubs (RPC).
- Módulo de la libreria del sistema.

El módulo del programa es obtenido a partir del P-Code generado para el programa en Polux, mediante una macro-expansión de cada P-Code, es decir, se toma cada uno de los p-code y se traduce toda su acción a sus códigos de máquinas correspondientes.

El generador de código objeto, solo obtiene el código correspondiente a la máquina señalada, por lo cual es necesario una busqueda de aquellos procesos y procedimientos que estan involucrados a dicha estación.

Con el objeto de generar los códigos de máquina adecuados se cuenta con un archivo (PDEFCOD.PAS) que contiene la definición de

constantes para cada uno los diferentes códigos del 8086, asi como la designación de registros y los diferentes modos de direccionamiento, el cual se puede observar asi:

```
CONST
  stacksize = 1024;

  { -- w=1 -- }
  ax = 0  {000};
  cx = 1  {001};
  dx = 2  {010};
  bx = 3  {011};
  sp = 4  {100};
  bp = 5  {101};
  si = 6  {110};
  di = 7  {111};
  { -- w=0 -- }
  al = 0  {000};
  cl = 1  {001};
  dl = 2  {010};
  bl = 3  {011};
  ah = 4  {100};
  ch = 5  {101};
  dh = 6  {110};
  bh = 7  {111};
  { segment  }
  es = 0  {00};
  cs = 1  {01};
  ss = 2  {10};
  ds = 3  {11};

  { ---- mod ---- }
  MOD00 = $00 {00};
  MOD01 = $40 {01};
  MOD10 = $80 {10};
  MOD11 = $C0 {11};

  { ---- r/m ---- }
  BXSID = $0  {000};
  BXDID = $1  {001};
  BPSID = $2  {010};
  BPDID = $3  {011};
  SID   = $4  {100};
  DID   = $5  {101};
  BPD   = $6  {110};
  BXD   = $7  {111};

  { -- codes -- }
MOVRMRM = $8800 {move r/m to/from register};
MOVIRM  = $C600 {move immediate to r/m};
```

```
MOVIR    = $B0    {move immediate to register};
MOVMA    = $A0    {move memory to accumulator};
MOVAM    = $A2    {move accumulator to memory};
MOVRMS   = $8E00 {move r/m to segment register};
MOVSRM   = $8C00 {move segment register to r/m};
PUSHRM   = $FF30 {push r/m};
PUSHR    = $50    {push register};
PUSHRS   = $06    {push segment register};
POPRM    = $8F00 {pop r/m};


IRET     = $CF    {interrupt return};
CLC      = $F8    {clear carry};
STC      = $F9    {set carry};
CMC      = $F5    {complement carry};
NOP      = $90    {not operation};
CLD      = $FC    {clear directio};
STD      = $FD    {set direction};
CLI      = $FA    {clear interrupt};
STI      = $FB    {set interrupt};
HLT      = $F4    {halt};
WAIT     = $9B    {wait};
LOCK     = $F0    {lock bus};
ESC      = $D800 {escape};
```

Con estas constantes se contruyen los códigos de máquina para el 8086. Por ejemplo construir un código que en ensamblador podemos mirarlo asi.

```
mov word ptr[di + 1C], 1831
```

Es obtenido de la siguiente forma:

```
emit1(MOVIRM or $0100 or MOD01 or DID);
emit($1C);
emit1($1831);
```

Que como puede verse en la porción del anterior archivo mostrado:

MOVIRM - mueve un valor inmediato a registro o a memoria,

$0100 - empleado para indicar que se movera un word (w = 1).

MOD01 - indica un desplazamiento de un byte con extensión de

signo.

DID - especifica el modo de direccionamiento usando el registro DI mas un desplazamiento ($1C).

$1C - indica el desplazamiento usado y

$1831 - especifica el valor inmediato a ser cargado.

Los procedimientos *emit* y *emit1* son utilizados para meter los códigos generados en el arreglo de código (obj).

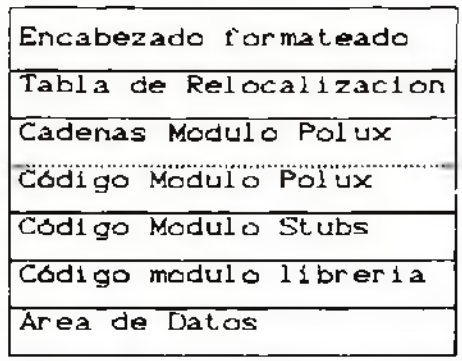
Se cuenta tambien con un archivo (SYSVAR.PAS) que contiene las direcciones relativas al principio del Data Segment (DS) de cada una de la variables usadas tanto por el módulo de libreria como el módulo del stubs, e igualmente usado para albergar las variables globales del programa Polux.

El módulo del stubs, fue compilado a partir de su código fuente (en Pascal), y extraido para ser encadenado al programa Polux.

Mientras el módulo de libreria del sistema, se obtuvo de la unidad SYSTEM del compilador Turbo Pascal 5.0, y en ella se encuentran todas las funciones matemáticas (trigonométricas, logaritmicas, etc.), de entrada/salida y manejo de memoria, entre otras.

Tanto para el módulo del stubs como para el de libreria, los puntos de relocalización son introducidos en la tabla de relocalización utilizada por el generador de código.

El compilador emite como producto final el código objeto del programa Polux (encadenado con los otros módulos), en formato de archivo ejecutable (extensión .EXE), el cual contiene la siguiente estructura:




Donde el encabezado es formateado como sigue. (Los offset son en hexadecimal.)

| offset | Contenido |
|---|---|
| 00-01 | Debe contener $4D, $5A. |
| 02-03 | Número de bytes contenidos en la ultima pagina. |
| 04-05 | Tamaño del archivo en paginas de 512-bytes, incluyendo el encabezado. |
| 06-07 | Número de entradas en la tabla de relocaclización. |
| 08-09 | Tamaño del encabezado en parrafos de 16-bytes. |
| 0A-0B | Número minimo de parrafos de 16-bytes requeridos arriba del fin del programa cargado. |
| 0C-0D | Número maximo de parrafos de 16-bytes requeridos arriba del fin del programa cargado. |
| 0E-0F | Valor inicial a ser cargado en el Stack Segment antes de inicial la ejecución del programa. |

10-11 Valor a ser cargado en el registro SP antes de iniciar la ejecución del programa.

12-13 Suma negativa de todas las palabras en el archivo.

14-15 Valor inicial a ser cargado en el registro IP antes de iniciar la ejecución del programa.

16-17 valor inicial a ser cargado en el registro CS antes de iniciar la ejecución del programa.

18-19 Desplazamiento relativo en bytes del principio del archivo ejecutable a la tabla de relocalización.

1A-1B Número generado por MS-LINK.

# 5

# LA RED (NETWORK)

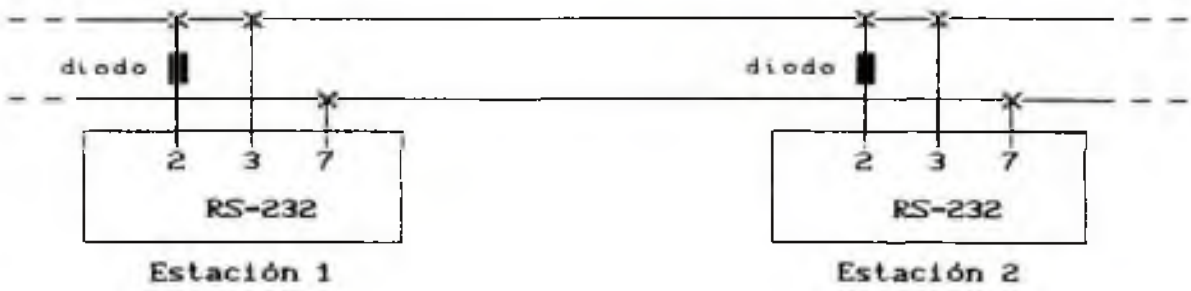
## 5.1. INTRODUCCION

La comunicación entre máquinas es llevada a cabo por Polux, mediante una red lenta tipo bus, que hace uso de las interfaces de comunicación serie RS-232 contenidas en la mayoria de las computadoras personales IBM-PC/XT o AT compatibles. Dicha red fue diseñada por el Dr. Jan Janecek Hyan, y recibe el nombre de *Modem Interface Local Area Network (MILAN)*

## 5.2. CARACTERISTICAS FISICAS (PHISICAL LAYER)

La parte física de la comunicación en la red, se desarrollo utilizando la interfaz de comunicación serie (RS-232). De ésta manera, cada estación está conectada con el resto de las estaciones, mediante la interfaz RS-232.

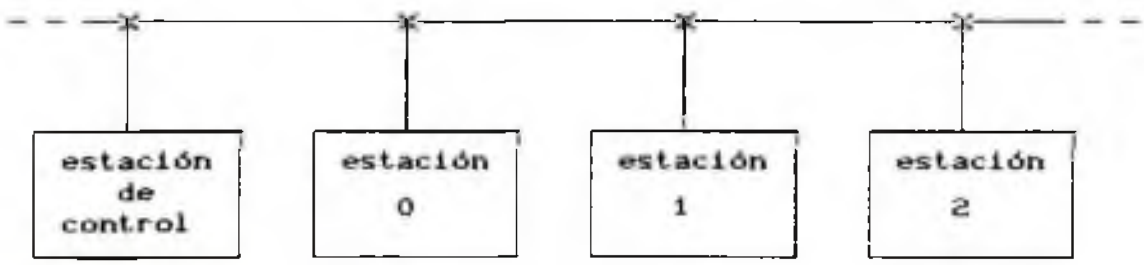
Solamente las terminales 2, 3 y 7 del RS-232, son usadas para la comunicación en la red, haciendo uso de solamente dos (2) hilos. Uno para transmisión de datos en ambos sentidos, y el otro para tierra, lo cual podemos apreciarlo asi:




Actualmente dicha interfaz fue programada, como caracteristicas principales, para comunicación asincrona, en el modo half-duplex, 8 bits de información y a 9600 bauds.

## 5.3. ESTRUCTURA Y TOPOLOGIA

La red esta formada por un bus compartido, en modo broadcasting, es decir, una estación transmite y todas (incluyendo ella) escuchan, lo cual podemos observarlo asi:




La estación de control, es la encargada de manejar el flujo de datos por la red, pues, de manera cíclica, le sede el bus a cada una de las estaciones para que transmitan si es que tienen algo que transmitir, o bién que lo intente con la siguiente estación.

Actualmente la red cuenta con 8 estaciones (del 0 al 7) de trabajo, más la estación de control. Para Polux, la estación 0 será la responsable de inicializar el trabajo, pues en ella estará siempre el código del módulo principal, y es precisamente este el recomendado para crear todos los procesos.

## 5.4. SERVICIOS

Primeramente mostraremos el tipo para la estructura que define los campos de una trama para la red:

```
NCB = RECORD
        Command : command;
        RemAddr : byte;
        LocAddr : byte;
        InfPtr  : packet;
        Lng     : byte;
        TmOut   : byte;
        Complete: retc;
      END;
```

Donde:

*Command* : Contiene el servicio que lleva la trama, los cuales pueden ser:

- CINIT : inicaliza la estación destino.
- CSDGR : envia un datagrama por la red.
- CRDGR : recoge un datagrama de la red.
- CSBRD : envia un datagrama en broadcasting.
- CRBRD : recoge un datagrama en broadcasting.
- CSHUT : dsconecta la estación de la red.

*RemAddr* : Especifica la estación destino a la cual va dirigido el paquete.

*LocAddr* : Especifica la estación local, que es quien envia el paquete.

*InfPtr* : Contiene los datos que trasporta la trama, el cual puede contener un maximo de 256 bytes.

*Lng* : Contiene la longitud en bytes que lleva el paquete.

*TmOut* : especifica el tiempo en segudos que tendrá el paquete para poder ser recogido o recibido, de los contrario se generará un error de timeout (TMOUT).

*Complete*: contiene el estado en que se encuentra el paquete, los cuales pueden ser:

WTNG : está en espera de ser enviado o recibido
OK : ha sido recogido o recibido.
TMOUT: se ha cumplido el tiempo y no pudo se tratado.
ERRC : sucedió un error (probablememte físico).

Enseguida veremos en detalle los paquetes:

**CINIT**

Usado para dar de alta la estación en la red, que por lógica este paquete tiene que ser enviado por la misma estación que se quiere dar de alta, este paquete no es necesario que contenga RemAddr, Lng, TmOut, pero si InfPtr, el cual debe apuntar al buffer donde serán recibidos por el nivel de red los paquetes enviados a esta estación. Cuando se envia de nuevo este servicio, y la estación ya esta conectada, se genera un error ERRC.

**CSDGR**

Usado para poner un paquete por la red, dirigido a la estación contenida en RemAddr, en el cual es conveniente especificar un tiempo de salida (TmOut) suficiente para que la estación destino pueda recibirlo. Cuando RemAddr y LocAddr coinsiden, el paquete no baja al autómata de transmisión, solamente es puesto por el mismo MILAN en le buffer de recepción.

El timeout debe ser suficiente para el caso extremo, que es cuando todas las estaciones tienen que transmitir su paquete completo (256 byte). y están conectadas todas las estaciones posibles a la red (8 estaciones).

**CRDGR**

Usado para recoger un paquete de la red, este servicio es solicitado cuando se detectó la llegada de un paquete para la

estación local, dicha detección de llegada puede ser echa mediante pooling, o bien mediante interrupciones.

**CSBRD**

Similar a CSDGR sólo que es enviada en modo broadcasting (para todas las estaciones).

**CRBRD**

Similar a SRDGR sólo que el paquete fue enviado en modo broadcasting, y en ese modo tiene que ser recibido.

**CSHUT**

Usado para cuando se quiere dar de baja la estación de la red.

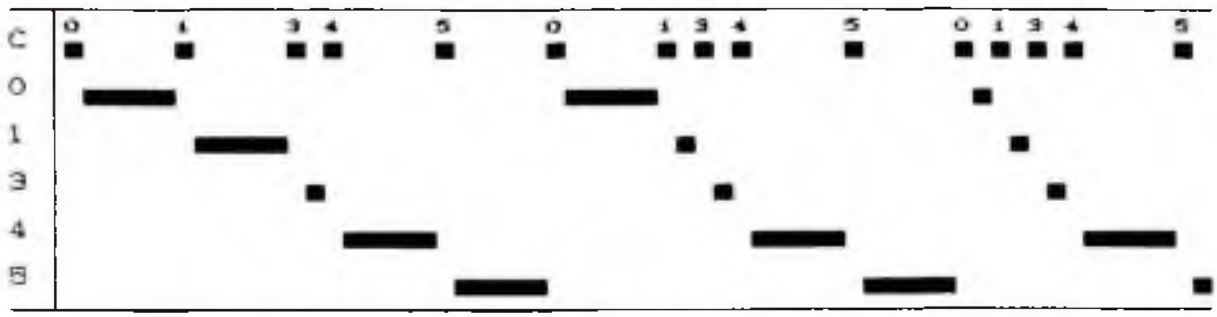
## 5.5. PROTOCOLO

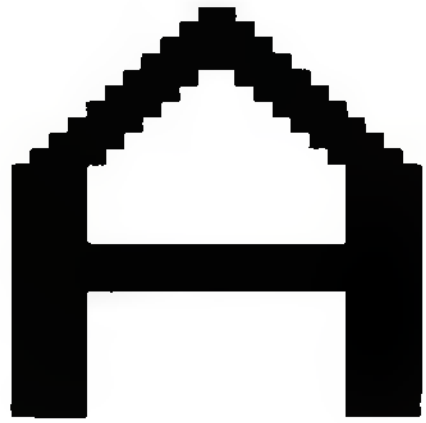
En MILAN para que cada estación pueda hacer uso del bus, debe esperar su turno, el cual es dado cíclicamente por la estación de control (estación de pooling). Cuando a una estación le toca su turno, si tiene paquete que mandar, le informa a la estación control y lo envía, de lo contrario solo le informa que no tiene nada que mandar. Esto es:


EJEMPLO: supongamos que se tienen conectadas las estaciones 0, 1, 3, 4 y 5, y cada una de ellas tienen que mandar los siguientes paquetes:

0 - 2 paquates
1 - 1 paquete
3 - no tine paquetes
4 - 3 paquetes
5 - 2 paquetes

Lo cual podemos apreciarlo asi:





# DIAGRAMAS SINTACTICOS

A continuación se presentan los diagramas sintácticos del lenguaje POLUX, que como se ha mensionado, éste por ser derivado de un sub-Pascal, contiene la sintaxis del Pascal, apareciendo aqui con las construcciones sintácticas orientadas a los procesos ditribuidos.

Con el objeto de una mejor interpretación, consideremos las siguientes notas.

NOTAS:

1 - Las cajas simples denotan simbolos del lenguaje Polux, y las cajas dobles denotan construcciones sintácticas representadas por diagramas.

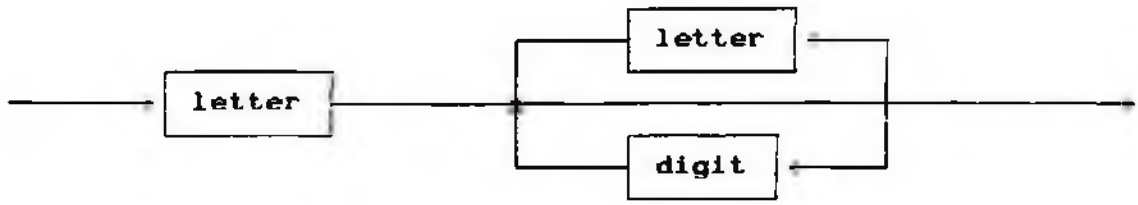
2 - Los separadores pueden ser insertados entre cualquier par de simbolos. Mas no deben ocurrir entre números e identificadores.

3 - Al menos un separador debe ocurrir entre identificadores consecutivos, números y palabras claves (tales como BEGIN, END).

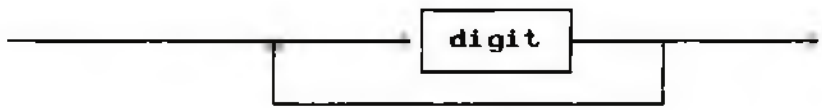
4 - Los separadores son blancos, fin de linea, y comentarios. Un comentario es una secuencia arbitraria de caracteres encerrados entre un par de paréntesis de comentario (* y *) o { y }.

5 - La ocurrencia de palabras identificadores sin calificativos en un diagrama sintáctico, implica que a este apunta un tipo arbitrario, nuevos identificadores pueden ser selecciones.

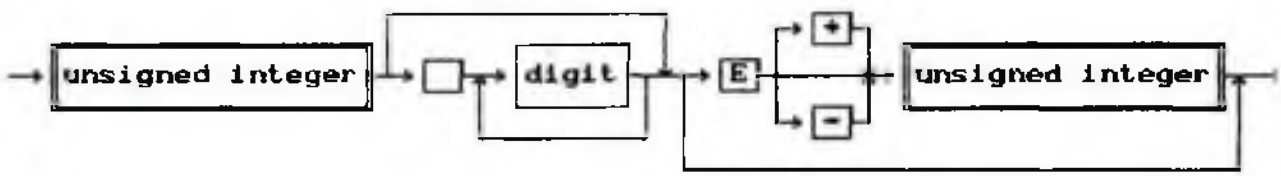
**Identifier**



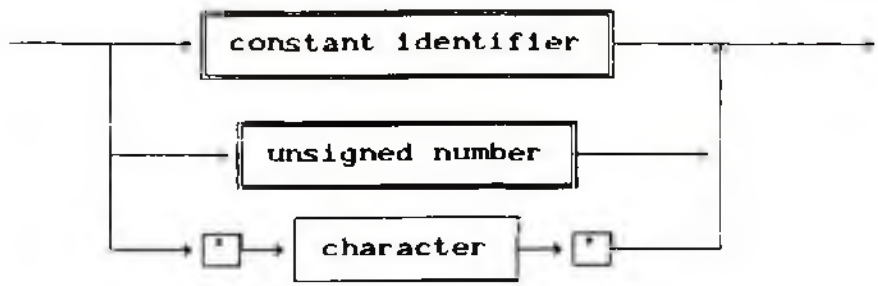

**Unsigned integer**



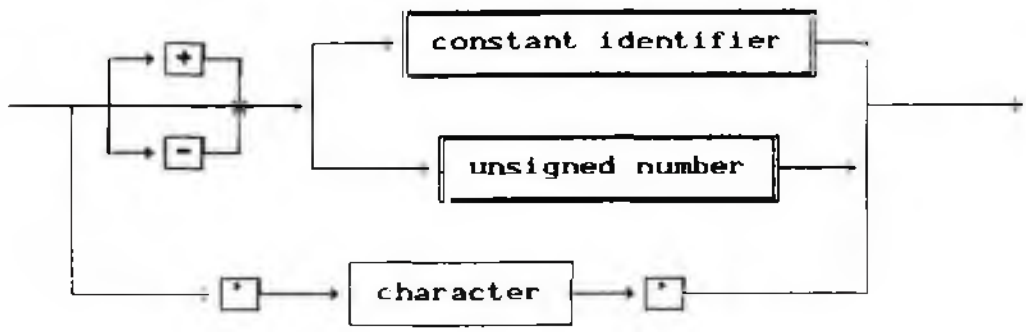

**Unsigned number**



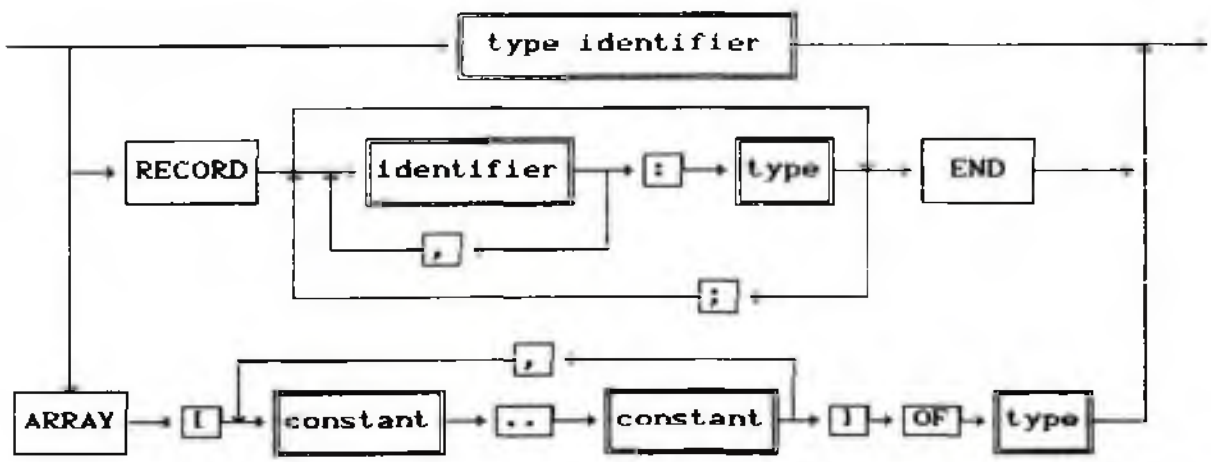
Unsigned constant



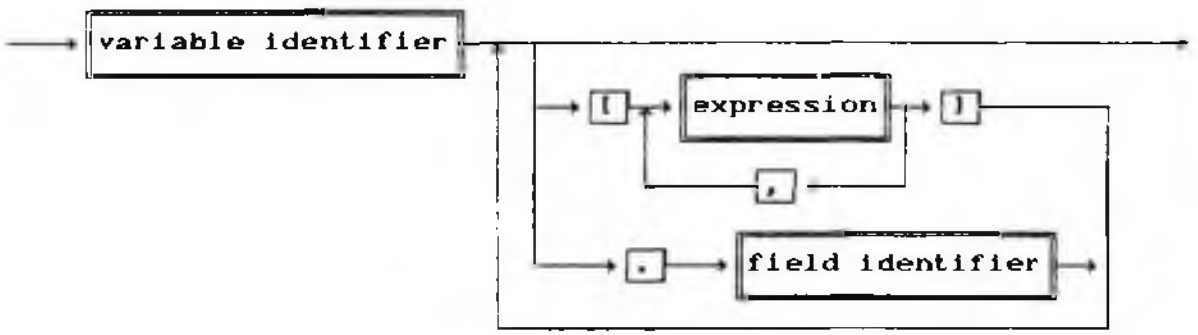

Constant



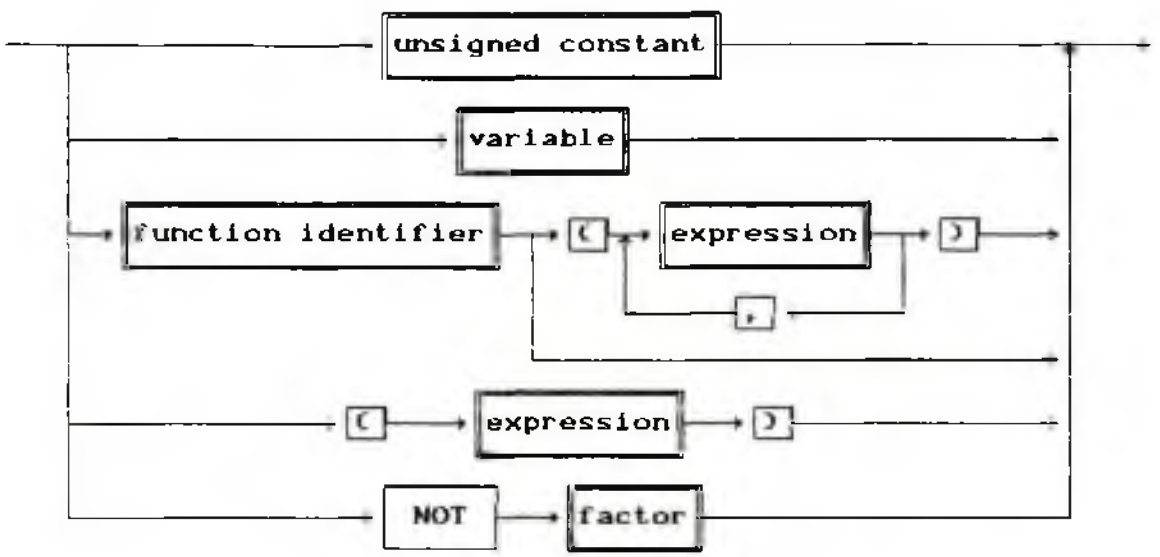

type



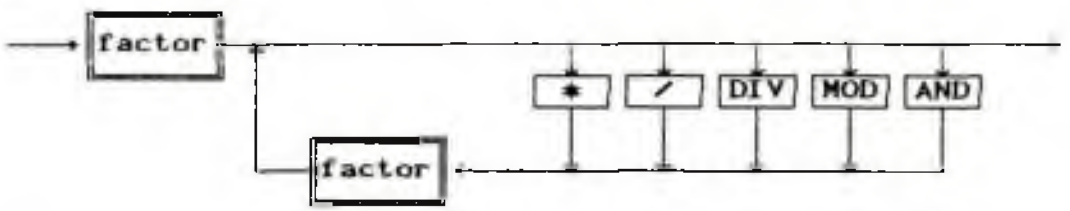
Variable



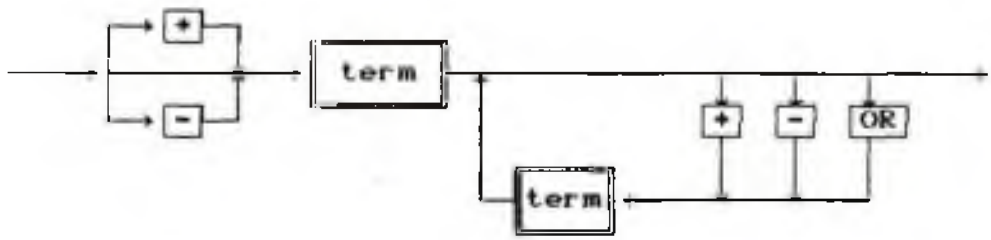

Factor



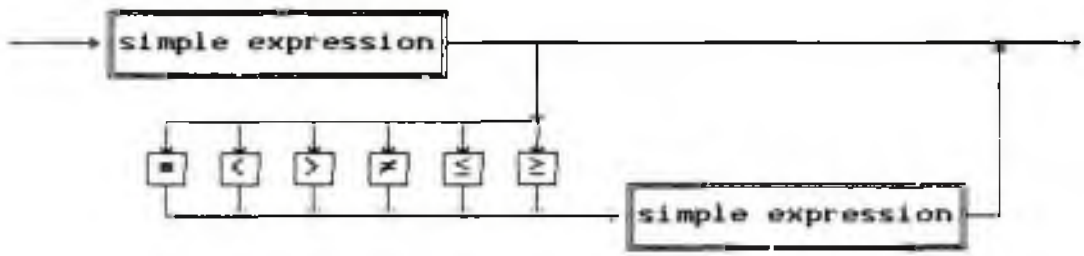

Term



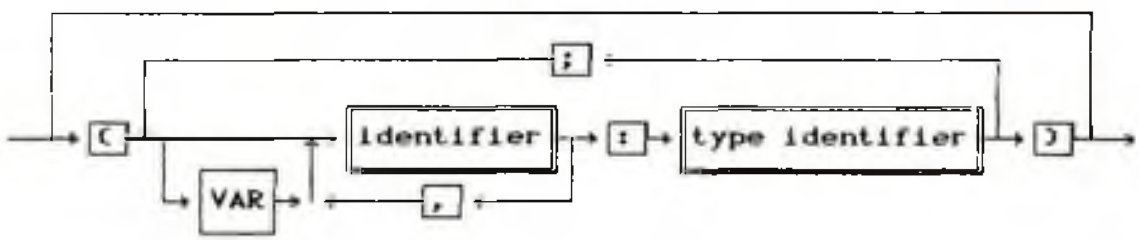
Simple expression



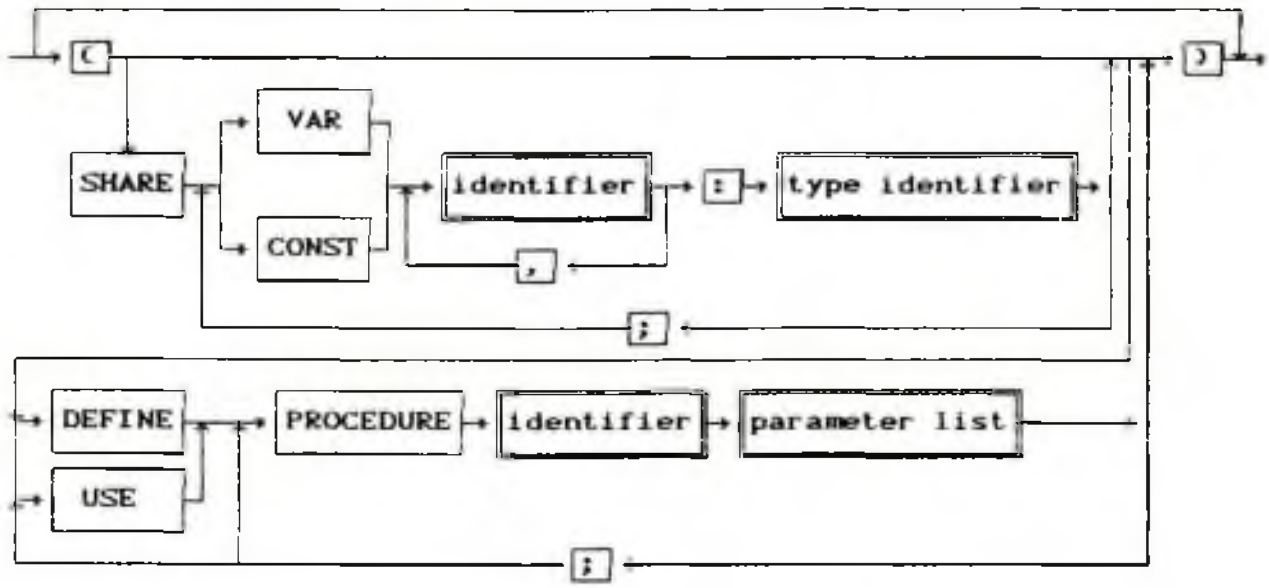

Expression



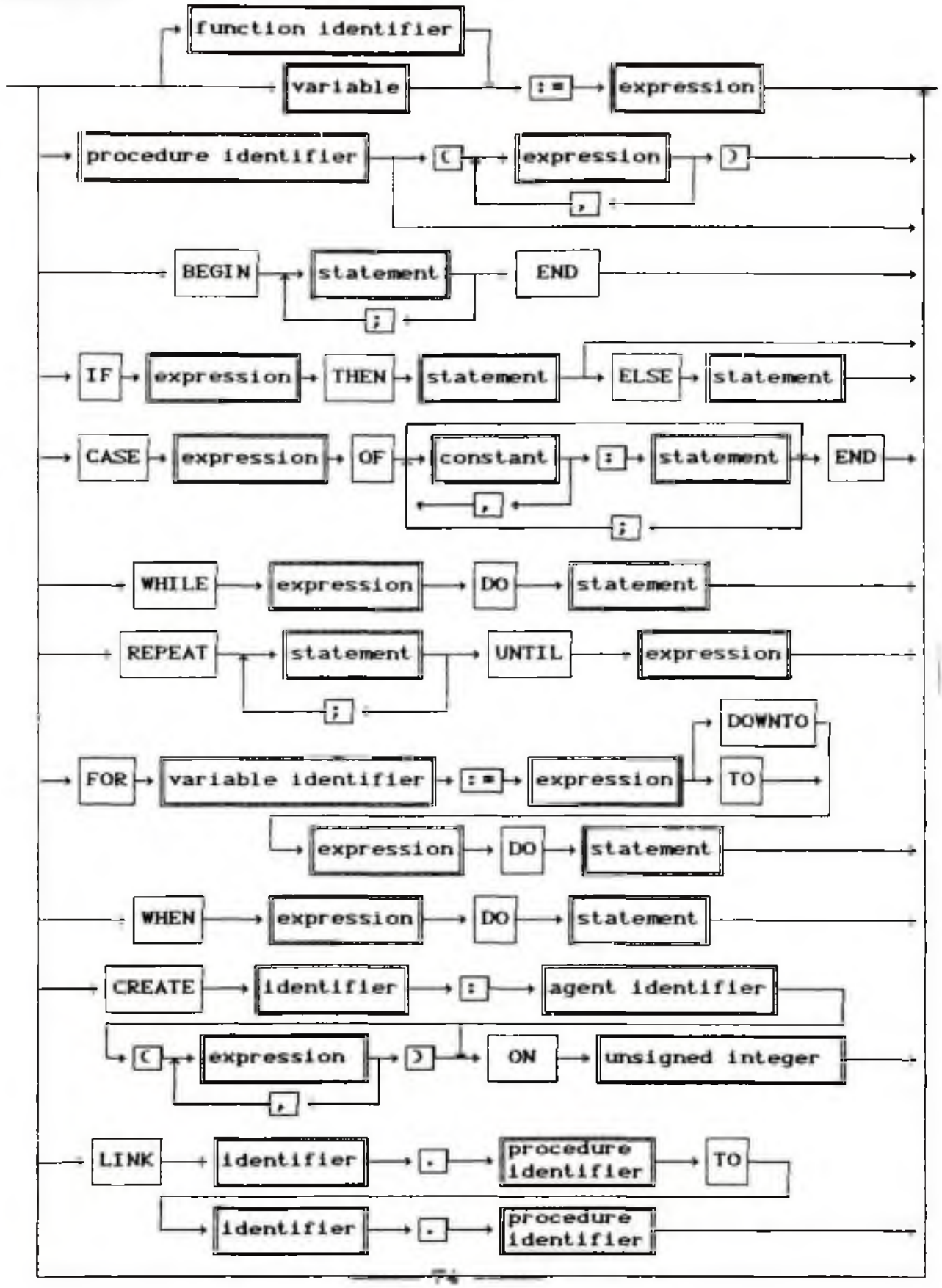

Parameter list



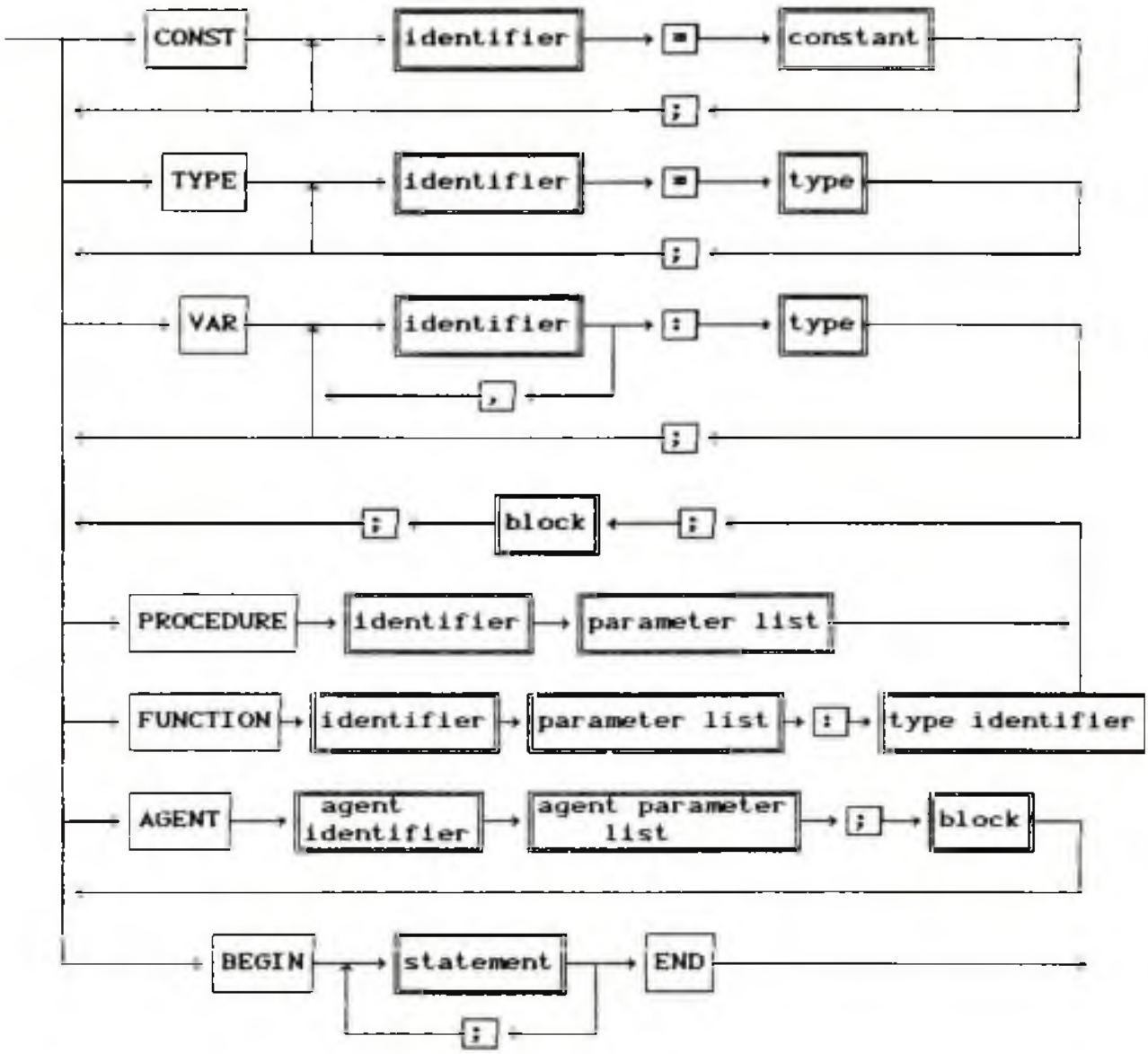
Agent parameter list



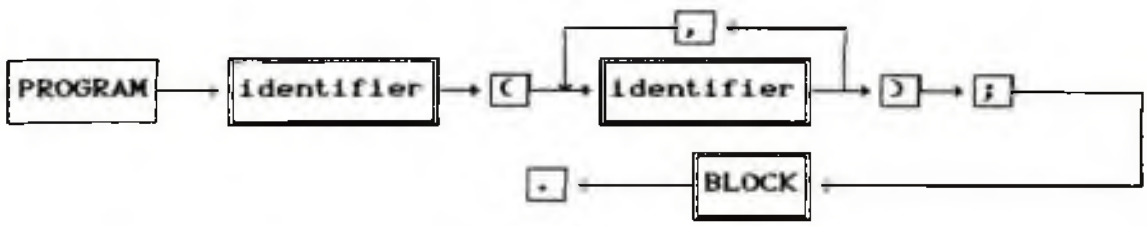
Statement

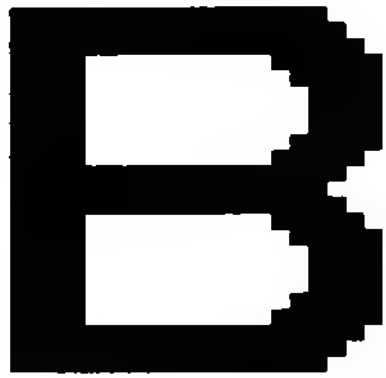

function identifier
variable
:=
expression
procedure identifier
(
expression
,
)
BEGIN
statement
;
END
IF
expression
THEN
statement
ELSE
statement
CASE
expression
OF
constant
,
:
statement
;
END
WHILE
expression
DO
statement
REPEAT
statement
;
UNTIL
expression
FOR
variable identifier
:=
expression
DOWNTO
TO
expression
DO
statement
WHEN
expression
DO
statement
CREATE
identifier
:
agent identifier
(
expression
,
)
ON
unsigned integer
LINK
identifier
.
procedure identifier
TO
identifier
.
procedure identifier

Block




Program





# EJEMPLOS

En este apéndice se mostrarán algunos progamas de aplicación para ejemplificar la programación en el lenguaje POLUX.

## LOS FILOSOFOS COMELONES

Cinco filósofos están sentados a una mesa circular. Cada uno de ellos tiene un plato de espagueti especialmente resbaladizo. El espagueti es tan resbaladizo que un filósofo necesita dos tenedores para comerlo. Entre cada plato hay un tenerdor.

La vida de un filósofo consta de periodos alternados de comer y pensar. Cuando uno filósofa siente hambre, intenta tomar el tenedor de la izquierda y el de la derecha, uno a la vez, en cualquier orden. Si logra asir dos tenedores, él toma unos bocados y después baja los cubiertos y sigue pensando.

```
PROGRAM Philosophers;

  AGENT philosopher ( SHARE CONST this:integer;
                      USE   PROCEDURE JoinToTable(i:integer);
                            PROCEDURE LeaveTable (i:integer) );
    BEGIN
      REPEAT
        JoinToTable(this);
```

```
      writeln('philosopher',this,' this eating');
      LeaveTable(this);
      writeln('philosopher',this,' this thinking');
    UNTIL FALSE
  END;


AGENT table( DEFINE PROCEDURE Join (i:integer);
                    PROCEDURE Leave(i:integer) );
  VAR Fork: ARRAY[1..5]OF boolean;
      i: integer;

  PROCEDURE Join(i:integer);
    VAR j: integer;
    BEGIN
      writeln('                    entry to join with ',i);
      WHEN Fork[i] AND Fork[i MOD 5 + 1] DO
        BEGIN
          Fork[i] := FALSE; Fork[i MOD 5 + 1] := FALSE;
          FOR j:=1 TO i DO Write('          ');
          WriteLn(' JOIN PH',i:1)
        END
    END;

  PROCEDURE Leave(i:integer);
    VAR j: integer;
    BEGIN
      writeln('                    entry to leave with ',i);
      WHEN TRUE DO
        BEGIN
          Fork[i] := TRUE; Fork[i MOD 5 + 1] := TRUE;
          FOR j:=1 TO i DO Write('          ');
          WriteLn('LEAVE PH',i:1)
        END
    END;

  BEGIN
    FOR i:=1 TO 5 DO
      Fork[i] := TRUE;
    WHEN FALSE DO
  END;

BEGIN
  CREATE Tb: table ON 0;
  CREATE Ph1:philosopher(1) ON 1;
  CREATE Ph2:philosopher(2) ON 2;
  CREATE Ph3:philosopher(3) ON 3;
  CREATE Ph4:philosopher(4) ON 4;
  CREATE Ph5:philosopher(5) ON 5;
  LINK Ph1.JoinToTable TO Tb.Join;
  LINK Ph1.LeaveTable  TO Tb.Leave;
  LINK Ph2.JoinToTable TO Tb.Join;
  LINK Ph2.LeaveTable  TO Tb.Leave;
  LINK Ph3.JoinToTable TO Tb.Join;
```

```
    LINK Ph3.LeaveTable  TO Tb.Leave;
    LINK Ph4.JoinToTable TO Tb.Join;
    LINK Ph4.LeaveTable  TO Tb.Leave;
    LINK Ph5.JoinToTable TO Tb.Join;
    LINK Ph5.LeaveTable  TO Tb.Leave;
  END.
```

Las tablas producidas por el compilador se muestran a continuación:

```
POLUX-S Experimental Compiler          Thursday  6/ 7/90    5:31 PM

  Identifiers      link   obj   typ   ref   nrm   lev   adr  offs top  shar

  31  philosophe      0     2     7     3     1     1     0     0     0
  32  this            0     1     1     0     1     2     5     0     2
  33  jointotabl     32     3     0     4     0     2     6     0     0
  34  i               0     1     1     0     1     2     5     0     2
  35  leavetable     33     3     0     5     0     2     7     0     0
  36  i               0     1     1     0     1     2     5     0     2
  37  table          31     2     7     6     1     2     0     0     0
  38  join            0     3     0     7     1     2     0     0     0
  39  i               0     1     1     0     1     2     5     0     2
  40  leave          38     3     0     8     1     2     0     0     0
  41  i               0     1     1     0     1     2     5     0     2
  42  fork           40     1     5     1     1     2     7     0     5
  43  i              42     1     1     0     1     2    12     5     7
  44  j              39     1     1     0     1     2     6     2     4
  45  j              41     1     1     0     1     2     6     2     4

  Blocks       link last lpar psze vsze  adr  fin bytes def

   1              0   30    1    0    0    0    0     0  Not
   2              0   37   30    5    5  134  215     0  Not
   3              0   35   35    8    8    0   30     0  Not
   4              0   34   34    6    0    0    0     0  Not
   5              0   36   36    6    0    0    0     0  Not
   6              7   43   40    7   13  121  133     0  Not
   7              8   44   39    6    7   31   81     0  Yes
   8              0   45   41    6    7   82  120     0  Yes

  Agent              Station ref stack code stshare

   0  philosophe          0   2     0    0
   1  tb                  0   6     0    1
   2  ph1                 1   3     0    1
   3  ph2                 2   3     0    1
   4  ph3                 3   3     0    1
   5  ph4                 4   3     0    1
   6  ph5                 5   3     0    1
```

```
Arrays      xtyp etyp eref   low high elsz size bytes

 1             1    3    0    1    5    1    5     1


Code

  0  CLRM      8    PCOM      3    LVAL 2    0    PTYP      1
     PVEX
  5  PNPA      1    CALR     33    LCON     11    WRTS      0
     LVAL 2    0
 10  WRT1      1    LCON     12    WRTS     12    WRLN
     CLRM      8
 15  PCOM      3    LVAL 2    0    PTYP      1    PVEX
     PNPA      1
 20  CALR     35    LCON     11    WRTS     25    LVAL 2    0
     WRT1      1
 25  LCON     14    WRTS     37    WRLN           LCON      0
     CJMP      0
 30  FSTK           LCON     35    WRTS     52    LVAL 2    0
     WRT1      1
 35  WRLN           LADR 2    0    LVAL 2    0    INXS      1
     LIND     34
 40  LADR 2    0    LVAL 2    0    LCON      5    IMOD
     LCON      1
 45  I+             INXS      1    LIND     34    AND
     SWPR     36
 50  LADR 2    0    LVAL 2    0    INXS      1    LCON      0
     ST        3
 55  LADR 2    0    LVAL 2    0    LCON      5    IMOD
     LCON      1
 60  I+             INXS      1    LCON      0    ST        3
     LADR 2    2
 65  LCON      1    LVAL 2    0    FOR1     71    LCON      9
     WRTS     88
 70  FOR2     68    LCON      8    WRTS     98    LVAL 2    0
     LCON      1
 75  WRT2      1    WRLN           CJMP     82    CLRM      6
     PCOM      4
 80  PNPA      0    RPLY           LCON     36    WRTS    107
     LVAL 2    0
 85  WRT1      1    WRLN           LCON      1    SWPR     87
     LADR 2    0
 90  LVAL 2    0    INXS      1    LCON      1    ST        3
     LADR 2    0
 95  LVAL 2    0    LCON      5    IMOD           LCON      1
     I+
100  INXS      1    LCON      1    ST        3    LADR 2    2
     LCON      1
105  LVAL 2    0    FOR1    110    LCON      9    WRTS    144
     FOR2    107
110  LCON      8    WRTS    154    LVAL 2    0    LCON      1
     WRT2      1
115  WRLN           CJMP    121    CLRM      6    PCOM      4
```

```
     PNPA      0
120  RPLY           LADR 2    5    LCON      1    LCON      5
     FOR1    131
125  LADR 2    0    LVAL 2    5    INXS      1    LCON      1
     ST        3
130  FOR2    125    LCON      0    SWCX    131    FSTK
     CLRM      4
135  PCOM      0    PAGE      1    PNPA      0    PSTA      0
     SMES
140  CLRM      4    PCOM      0    PAGE      2    LCON      1
     PTYP      1
145  PVEX           PNPA      1    PSTA      1    SMES
     CLRM      4
150  PCOM      0    PAGE      3    LCON      2    PTYP      1
     PVEX
155  PNPA      1    PSTA      2    SMES           CLRM      4
     PCOM      0
160  PAGE      4    LCON      3    PTYP      1    PVEX
     PNPA      1
165  PSTA      3    SMES           CLRM      4    PCOM      0
     PAGE      5
170  LCON      4    PTYP      1    PVEX           PNPA      1
     PSTA      4
175  SMES           CLRM      4    PCOM      0    PAGE      6
     LCON      5
180  PTYP      1    PVEX           PNPA      1    PSTA      5
     SMES
185  CLRM      3    PCOM      1    PSTA      0    PAGE      1
     SMES
190  CLRM      3    PCOM      1    PSTA      1    PAGE      2
     SMES
195  CLRM      3    PCOM      1    PSTA      2    PAGE      3
     SMES
200  CLRM      3    PCOM      1    PSTA      3    PAGE      4
     SMES
205  CLRM      3    PCOM      1    PSTA      4    PAGE      5
     SMES
210  CLRM      3    PCOM      1    PSTA      5    PAGE      6
     SMES
215  FIN
```

**PRODUCTORES CONSUMIDORES**

A continuación se listan algunos productores consumidores:

```
PROGRAM ProdCons1;

VAR l:integer;

AGENT buffer ( SHARE  VAR l,k,m:integer;
               DEFINE PROCEDURE push(x:integer);
                      PROCEDURE pop(VAR x:integer)
             );
CONST n = 10;
TYPE contents = ARRAY [1..n] OF integer;
VAR  head,tail,length : integer;
     ring : contents;

PROCEDURE push(x:integer);
VAR j:integer;
BEGIN
  WHEN length<n DO
    BEGIN
      j:=1; l:=1;
      ring[tail] := x;
      WriteLn('           push =',x);
      tail := (tail mod n)+1; length := length+1
    END;
END;

PROCEDURE pop(var x:integer);
VAR j,l:integer;
BEGIN
  WHEN length>0 DO
    BEGIN
      j:=3; l:=3;
      x := ring[head];
      writeln('      pop =',x);
      head := (head mod n)+1; length := length-1
    END;
END;

BEGIN
  head := 1; tail := 1; length := 0
END;

AGENT producer ( SHARE VAR l:integer;
                 USE   PROCEDURE push(x:integer) );
VAR i : integer;

BEGIN
  writeln('start producer');
  i := 1;
  WHILE true DO
    BEGIN
      l:=0; push(i); i := i+1
    END;
  writeln('ending producer');
END;
```

```
AGENT consumer ( share const l:integer;
                 USE   PROCEDURE pop(VAR x:integer) );
VAR i : integer;
BEGIN
  writeln('start consumer');
  WHILE true DO
    BEGIN
      l:=2; pop(i)
    END;
  writeln('ending consumer');
END;

BEGIN
  l := 1;
  CREATE a:buffer(1,1,1) ON 1;
  CREATE b:buffer(1,1,1) ON 2;
  CREATE p:producer(1) ON 3;
  CREATE c:consumer(1) ON 3;
  LINK p.push TO a.push;
  LINK c.pop  TO a.pop;
  writeln('start main');
END.
```

```
PROGRAM ProCon2;

AGENT producer ( USE PROCEDURE push(x:char) );
VAR c: char;
BEGIN
  WHILE NOT Eof(Input) DO
    BEGIN
      WHILE NOT EoLn(Input) DO
        BEGIN
          Read(c); Push(c)
        END;
      ReadLn; Push(chr(10)); Push(chr(13));
    END;
  Push(chr(26));
END;

AGENT consumer ( DEFINE PROCEDURE pop(x:char) );
VAR c: char;
    Free, Stop: boolean;
PROCEDURE Pop(x:char);
BEGIN
  WHEN Free DO
    BEGIN
      c := x; Free := FALSE
    END
```

```
END;
BEGIN
  Free := TRUE; Stop := FALSE;
  REPEAT
    WHEN NOT Free DO
      IF c<>chr(26) THEN Write(c)
      ELSE Stop := TRUE;
      Free := TRUE
  UNTIL Stop
END;

BEGIN
  CREATE c:consumer on 1;
  CREATE p:producer on 2;
  LINK p.push TO c.pop;
END.
```

```
PROGRAM ProCon3;

  TYPE Buff = ARRAY[1..10]OF char;

  VAR Buffer: Buff;
      IPtr, OPtr, Cnt: integer;

  AGENT producer ( SHARE VAR Data: Buff;
                             i, c: integer );
    VAR ch: char;

    PROCEDURE Push(x:char);
      BEGIN
        WHEN c<3 DO
          BEGIN
            Data[i] := x;
            i := (i MOD 3)+1; c := c+1
          END
      END;

    BEGIN
      WHILE NOT Eof(Input) DO
        BEGIN
          WHILE NOT EoLn(Input) DO
            BEGIN
              Read(ch); Push(ch)
            END;
          ReadLn; Push(chr(10)); Push(chr(13));
        END;
      Push(chr(0));
    END;
```

```
  AGENT consumer ( SHARE VAR Data: Buff;
                             o, c: integer );
    VAR ch: char;
        Stop: boolean;

    PROCEDURE Pop(VAR x:char);
      BEGIN
        WHEN c>0 DO
          BEGIN
            x := Data[o];
            o := (o MOD 3)+1; c := c-1
          END
      END;

    BEGIN
      Stop := FALSE;
      REPEAT
        Pop(ch);
        IF ch<>chr(0) THEN Write(ch)
                      ELSE Stop := TRUE;
      UNTIL Stop
    END;

  BEGIN
    CREATE c:consumer(Buffer,OPtr,Cnt) on 1;
    CREATE p:producer(Buffer,IPtr,Cnt) on 2;
    IPtr := 1; OPtr := 1; Cnt := 0
  END.
```

```
PROGRAM ProCon4;

  AGENT producer ( USE PROCEDURE push(x:char) );
    VAR c: char;
    BEGIN
      WHILE NOT Eof(Input) DO
        BEGIN
          WHILE NOT EoLn(Input) DO
            BEGIN
              Read(c); Push(c)
            END;
          ReadLn; Push(chr(10)); Push(chr(13));
        END;
      Push(chr(26));
    END;

  AGENT consumer ( USE PROCEDURE pop(VAR x:char) );
    VAR c: char;
        Stop: boolean;
    BEGIN
```

```
    Stop := FALSE;
    REPEAT
      Pop(c);
      IF c<>chr(26) THEN Write(c)
                    ELSE Stop := TRUE;
    UNTIL Stop
  END;

AGENT buffer( DEFINE PROCEDURE recv(x:char);
                     PROCEDURE send(VAR x:char) );
  VAR Data: ARRAY[1..10]OF char;
      IPtr, OPtr, Cnt: integer;

  PROCEDURE Recv(x:char);
    BEGIN
      WHEN Cnt<3 DO
        BEGIN
          Data[IPtr] := x;
          IPtr := (IPtr MOD 3)+1; Cnt := Cnt+1
        END
    END;

  PROCEDURE Send(VAR x:char);
    BEGIN
      WHEN Cnt>0 DO
        BEGIN
          x := Data[OPtr];
          OPtr := (OPtr MOD 3)+1; Cnt := Cnt-1
        END
    END;

  BEGIN
    IPtr := 1; OPtr := 1; Cnt := 0
  END;

BEGIN
  CREATE b:buffer on 1;
  CREATE c:consumer on 2;
  CREATE p:producer on 3;
  LINK p.push TO b.recv;
  LINK c.pop  TO b.send
END.
```

### LOS TRES FUMADORES

Considerar un sistema con tres procesos fumadores y un proceso agente. Cada fumador continuamente toma un cigarro y lo fuma. Pero para fumar un cigarro, tres ingredientes son

necesarios: tabaco, papel y cerillos. Uno de los procesos tiene papel, otro tabaco y el tercero tiene cerillos. El agente tiene una fuente infinita de los tres ingredientes. El agente coloca dos de los ingredientes en la mesa. El fumador que tiene el ingrediente restante puede hacerse y fumarse el cigarro, indicándole al agente que ya fumo. El agente entonces pone en la mesa otros dos de los ingredientes y el ciclo se repite.

```
PROGRAM Smoker;

CONST   TOBACCO = 0;
        PAPER   = 1;
        MATCHES = 2;

VAR
    Ingr1,Ingr2: integer;
    table: boolean;

AGENT Smoker1(SHARE VAR tableempty: boolean;
                        Ingred1,Ingred2: integer;
              USE PROCEDURE Smoking(i: integer));    { with tobacco }
BEGIN
  WHILE TRUE DO
  BEGIN
    WHEN (NOT tableempty) AND (Ingred1 <> TOBACCO) AND
         (Ingred2 <> TOBACCO) DO
      Smoking(1);
  END; { WHILE }
END; { AGENT }

AGENT Smoker2(SHARE VAR tableempty: boolean;
                        Ingred1,Ingred2: integer;
              USE PROCEDURE Smoking(i: integer));    { with paper }
BEGIN
  WHILE TRUE DO
  BEGIN
    WHEN (NOT tableempty) AND (Ingred1 <> PAPER) AND
         (Ingred2 <> PAPER) DO
      Smoking(2);
  END; { WHILE }
END; { AGENT }

AGENT Smoker3(SHARE VAR tableempty: boolean;
                        Ingred1,Ingred2: integer;
              USE PROCEDURE Smoking(i: integer));    { with matches }
BEGIN
  WHILE TRUE DO
  BEGIN
```

```
    WHEN (NOT tableempty) AND (Ingred1 <> MATCHES) AND
         (Ingred2 <> MATCHES) DO
      Smoking(3);
  END; { WHILE }
END; { AGENT }

AGENT AgentManager(SHARE VAR tableempty: boolean;
                           Ingred1,Ingred2: integer;
                    DEFINE PROCEDURE Make(i: integer));

  PROCEDURE Make(i: integer);
  VAR j: integer;
  BEGIN
    WriteLn('                                              smoker',i);
    FOR j:=0 TO 10000 DO;          { smoking }
    tableempty := TRUE;
  END; { Make }

BEGIN
  WHILE TRUE DO
    WHEN tableempty DO
    BEGIN
      Ingred1 := random(3);
      REPEAT
        Ingred2 := random(3);
      UNTIL Ingred1 <> Ingred2;
      Write('agent supply ');
        CASE Ingred1 OF
          TOBACCO : Write('tobacco ');
          PAPER   : Write('paper ');
          MATCHES : Write('Matches ');
        END;
        Write('and ');
        CASE Ingred2 OF
          TOBACCO : Write('tobacco ');
          PAPER   : Write('paper ');
          MATCHES : Write('Matches ');
        END;
        WriteLn;
      tableempty := FALSE;
    END;
END;

BEGIN
  table := TRUE;
  CREATE ag   :AgentManager(table,Ingr1,Ingr2) ON 0;
  CREATE smok1:Smoker1(table,Ingr1,Ingr2) ON 1;
  CREATE smok2:Smoker2(table,Ingr1,Ingr2) ON 2;
  CREATE smok3:Smoker3(table,Ingr1,Ingr2) ON 3;
  LINK smok1.Smoking TO ag.Make;
  LINK smok2.Smoking TO ag.Make;
  LINK smok3.Smoking TO ag.Make;
END.
```

# B I B L I O G R A F I A


AHO ALFRED V., SETHI RAVI, ULLMAN JEFFRY D.: Compilers Principles, Techniques, and Tools. Addison-Wesley, N. Y., 1986.

BIRREL A. D. and NELSON B. J.: "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems*, Vol 2, pp. 39-59, February 1984.

CHANBERS FRED B., DUCE DAVID A. and JONES GILLIAN P.: Distributed Computing. Academic Press, Inc. LTD, Orlando Florida, 1984.

DULAY NARANKER, KRAMER JEFF, MAGEE JEFF, SLOMAN MORRIS and TWIDLE KEVIN: "Distributed System Construction: Experience whit the Conic Toolkit," pp. 189-212.

HANSEN P. BRINCH: "Distributed Process: A Concurrent Programing Concepts," *Comm ACM*, 21(11), pp. 934-941, 1978.

HANSEN P. BRINCH: "Joyse - A Programming Language for Distributed System," *Software-Practice and experience*, vol 17(1), pp. 29-50, January 1987

HANSEN P. BRINCH: "The Joyce Language Report," *Software-Practice and experience*, vol 19(6), pp. 553-578, June 1989.

HANSEN P. BRINCH: "A Multiprocessor Implementation of Joyce," *Software-Practice and experience*, vol 19(6), pp. 579-592, June 1989.

HOARE C. A. R.: "Communicating Sequential Process," *Comm ACM*, 21(8), pp. 666-677, 1978.

KRAMER J., MAGEE J., SLOMAN M. and LISTER A.: "Conic: An Integrated Approach to Distributed Computer Control System,"

*IEEE Proceedings*, 130 Part E(1), pp. 1-10, January 1983.

LAMPSON B. W., PAUL M. and SIERGERT H. J.: Distributed System Architecture and implementation. Springer-Verlang, Berlin Heidelberg, 1981.

MAGEE JEFF, KRAMER JEFF and SLOMAN MORRIS: "Constructing Distributed System in Conic," *IEEE Transactios on Software Engineering*, Vol 15, pp. 663-675, June 1989.

MORGAN CHRISTOPHER L. WAITE MITCHELL: Introducción al Microprocesador 8086/8088 (16 Bit). McGraw-Hill, Byte Books, Mexico, 1985.

MORSE STEPHEN P.: The 8086/8088 Primer, Hayden Book C. I., Hasbrouck Heights, N. J., 1982.

SHRIVASTAVA S. K. and PANZIERI F.: "The design of reliable Remote Procedure Call Mechanism," *IEEE Transactions on Computers*, Vol C-31, pp. 692-697, July 1982. Second Edition,

TANENBAUM A. S.: Computer Networks. Second Edition, Prentice-Hall, Englewood Cliffs, N. J., 1988.

TANENBAUM A. S. and RENESSE R. V.: "Distributed Operating Systems," *Computing Surveys*, Vol 17, pp. 419-470, December 1985.

WIRTH N.: Pascal-S: A Subset ans its Implementation (Edited by Barron D.W., Chap. 12, Pascal-The Language and its Implementation). John Wiley & Sons, Ltd, 1981.

El jurado designado por la Sección de Computación del Departamento de Ingeniería Electrica del Centro de Investigación y de Estudios Avanzados del I.P.N. Aprobo esta tesis el 21 de Junio de 1990.

DR. JAN JANECEK HYAN

DR. MARIO ALBARRAN FIGUEROA

M. EN C. OSCAR OLMEDO AGUIRRE