LEDAC

DISEÑO E IMPLEMENTACION DE UN TRADUCTOR DE SDL A C.

Manual de Desarrollo.

Proyecto Fin de Carrera

Realizado por D. Eloy-Rafael Sanz Tapia.

Dirigido por D. Manuel Roldán Castro.

Departamento de Lenguajes y Ciencias de la Computación.

FACULTAD DE INFORMATICA.

UNIVERSIDAD DE MALAGA.

SEPTIEMBRE 1994.

A quien me ha dado algunos de mis mejores momentos.

A quien ha soportado mis 'no, aún no está

acabado'.

A quien me ha estado insistiendo para que lo estuviese.

A quien más quiero.

A Mari Carmen.

Indice.

1.- Introducción. 1

2.- El lenguaje SDL. 6
- 2.1.- Estructura. 6
- 2.2.- Señales y conexiones. 7
- 2.3.- Comportamiento. 8
- 2.4.- Tipos de datos. 10
- 2.5.- Otros aspectos. 10

3.- El lenguaje C. 12

4.- Introducción a yacc. 13
- 4.1.- Generalidades. 13
- 4.2.- Sección de declaraciones. 13
- 4.3.- Sección de reglas de la gramática. 15
  - 4.3.1.- Conflictos reducción/reducción. 16
  - 4.3.2.- Conflictos desplazamiento/reducción. 16
- 4.4.- Sección de código. 17
- 4.5.- Funciones auxiliares obligatorias. 17
- 4.6.- Línea de comandos. 18

5.- Introducción a lex. 20
- 5.1.- Generalidades. 20
- 5.2.- Sección de definiciones. 21
- 5.3.- Sección de patrones. 21
- 5.4.- Sección de código. 24
- 5.5.- Línea de comandos. 25

6.- El traductor (LEDAC). 26
- 6.1.- Terminología. 26
- 6.2.- Los ficheros. 27
- 6.3.- La fase de análisis. 30
  - 6.3.1.- Estructuras. 30
  - 6.3.2.- Las variables globales. 45
  - 6.3.3.- Los procedimientos. 47

6.4.- La fase de síntesis. 53
6.4.1.- Las estructuras. 53
6.4.2.- Las variables globales. 53
6.4.3.- Los procedimientos. 54
6.5.- Línea de comandos. 55

7.- El sistema equivalente (EQV). 57
7.1.- El esquema de traducción. 57
7.1.1.- Introduccion. 57
7.1.2.- Funciones asociadas. 58
7.1.3.- La ejecución. 59
7.1.4.- Los canales, las rutas de señales, las listas de señales... 60
7.1.5.- Expresiones. 60
7.1.6.- Valores iniciales: el modo Geode. 62
7.1.7.- Las transiciones. 63
7.1.8.- Señales contínuas y transiciones espontáneas. 63
7.1.9.- Resguardo de señales (SAVE). 64
7.1.10.- Condiciones de activación (PROVIDED). 65
7.1.11.- Etiquetas. 65
7.1.12.- Decisiones. 65
7.1.13.- NEXTSTATE. 67
7.1.14.- RETURN. 67
7.1.15.- STOP. 67
7.1.16.- CREATE. 67
7.1.17.- CALL. 68
7.1.18.- Llamadas a procedimientos. 68
7.1.19.- Variables exportadas: IMPORT. 70
7.1.20.- Variables reveladas: VIEW. 70
7.1.21.- OUTPUT. 70
7.1.22.- Temporizadores: SET, RESET, ACTIVE, NOW. 71
7.1.23.- TASK. 72
7.1.24.- ANY. 73
7.1.25.- PARENT, SELF, SENDER, OFFSPRING. 73
7.1.26.- Expresiones condicionales. 73
7.1.27.- Estructuras, arrays, powersets y strings. 73
7.2.- Los ficheros. 74
7.2.1.- Generados. 74
7.2.2.- Del armazón: 75
7.3.- Las estructuras. 76
7.4.- Las variables globales. 79
7.5.- Los procedimientos. 80
7.6.- El modo de depuración. 81
7.7.- Línea de comandos. 83

8.- El preprocesador. 84
- 8.1.- Los ficheros. 84
- 8.2.- Las estructuras. 85
- 8.3.- Las variables globales. 85
- 8.4.- Las funciones. 86
- 8.5.- La gramática del preprocesador. 87
- 8.6.- Línea de comandos. 88

Apéndice A.- Modificaciones respecto al SDL original. 89

Apéndice B.- Gramática. 92

Apéndice C.- Expansión. 109
- C.1.- Adición de nuevas instrucciones. 109
- C.2.- Adición de nuevas funciones. 111

Apéndice D.- Bibliografía. 116

1.- Introducción.

Los lenguajes de especificación y descripción formal sirven para definir las características y comportamiento de un sistema físico o lógico. Esta definición puede ser después utilizada en la implementación del sistema o en las pruebas del mismo (para comprobar que las cumple).

La definición puede hacerse a distintos niveles, y las características descritas pueden ser varias: comportamiento, estructura, dimensiones, consumo, etcétera.

Se debe considerar la diferencia entre especificación y descripción. La especificación de un sistema se refiere más a los requerimientos y comportamiento del mismo, sin considerar su estructura interna. La descripción, sin embargo, considera en mayor profundidad los aspectos relacionados con la estructura del sistema y la conexión entre sus componentes. Sin embargo, en adelante consideraremos ambos términos bajo el nombre único de especificación.

Un lenguaje de especificación proporciona ventajas que no pueden ser superadas por lenguajes de programación o por técnicas de descripción formales (o, por supuesto, por el lenguaje natural): claridad y falta de ambigüedad en las especificaciones, posibilidad de comprobar matemáticamente que el sistema descrito cumple una serie de propiedades o que una determinada implementación concuerda con la especificación, posibilidad de usar herramientas software para gestionar la especificación (crearla, analizarla, simularla...), etcétera.

Es cada vez más importante el disponer de herramientas que puedan simular el comportamiento de un sistema descrito mediante un lenguaje de especificación formal.

Entre los lenguajes formales de especificación y descripción se está imponiendo últimamente SDL (Specification and Description Language), estandarizado por el CCITT y la ISO. SDL permite la especificación y la descripción tanto de las características de comportamiento de un sistema como de la estructura del mismo.

En esta Facultad se está utilizando el lenguaje SDL en el marco del Plan Banda Ancha de Telefónica para describir protocolos de comunicación. Con este uso ha surgido la necesidad de simular sistemas descritos en SDL, para comprobar su corrección, y por ello se decidió construir un software propio (existen en el mercado algunos sistemas comerciales) para la ejecución de especificaciones SDL.

Las posibilidades para construir ese software eran varias:

- Podía hacerse un sistema de interpretación que tomase una especificación escrita en SDL, la analizase y la ejecutase, y que tuviese que repetir el análisis cada vez que se deseara ejecutar el sistema descrito.

- También podía construirse un compilador SDL que generase código objeto a partir de especificaciones SDL. Este código sería luego unido a una librería de funciones y se generaría un programa ejecutable que simularía el sistema.

- O existía la posibilidad de implementar un traductor que, a partir de una sistema especificado en SDL, generase un programa en un lenguaje de programación (no ya de descripción). Este programa resultante podría ser compilado y ejecutado para simular el sistema inicial.

Como se ve, las tres alternativas son un intérprete, un compilador y un traductor. Sus fases de análisis (front-end) serían seguramente idénticas. La única diferencia residiría en sus fases de síntesis (back-end): una simularía directamente el sistema analizado, otra generaría código objeto a partir de éste y la tercera generaría código en otro lenguaje. Por consiguiente, la decisión habría de basarse en estas diferencias.

La primera opción tiene la principal desventaja de todos los intérpretes: la necesidad de analizar la entrada cada vez que se desea ejecutarla. Sin embargo, posee como ventaja la simplicidad característica de los sistemas interpretados.

La opción de construir un compilador de SDL plantea dos graves inconvenientes: la complicación asociada a la generación de código objeto y la falta de portabilidad, ya que éste sólo serviría para una máquina determinada. Como ventaja principal se puede aducir la velocidad de ejecución una vez analizado y compilado el sistema SDL.

La tercera solución (el traductor) tiene un inconveniente (la necesidad de ejecutar dos pasos: primero traducir y luego compilar) y, sin embargo, bastantes ventajas: la dificultad de la implementación es intermedia, el código sería modificable después de la traducción, se podría integrar en otros sistemas de forma sencilla, sería portable (dependiendo del lenguaje objeto escogido) y fácil de ampliar, etcétera.

Se escogió la tercera opción y se decidió implementar un traductor de SDL a otro lenguaje.

Este proyecto tiene como objetivo el diseño e implementación de un sistema traductor de lenguaje SDL a lenguaje C. Además del presente proyecto, hay otra línea de diseño que prepara la traducción de SDL a Parlog.

El resultado del proyecto, pues, será un software que lea un fichero que contenga la descripción en SDL textual de un sistema, y que genere una serie de ficheros de código C que, tras ser compilados y enlazados con otras funciones, darán un programa ejecutable. Este programa simulará el comportamiento del sistema especificado.

El traductor tendrá dos fases claramente diferenciadas: la fase de análisis y la de síntesis. La primera se encargará de comprobar la corrección sintáctica y semántica de la especificación de entrada y simultáneamente generará una representación interna de la estructura del sistema descrito. Para ello usará un analizador sintáctico generado por la herramienta yacc (de la que se ofrecerá posteriormente una referencia) a partir de la gramática del lenguaje SDL.

La fase de síntesis (o back-end) se iniciará cuando termine la lectura del fichero fuente y es, como se dijo antes, la que hace que el programa sea un traductor. Se ha intentado hacer que las dos fases sean lo más independientes posible, de forma que la fase de análisis comparte con la de síntesis sólo el conocimiento de las estructuras internas que representan el sistema y pocas cosas más. Por lo tanto, es posible sustituir esta segunda fase por una distinta que, en lugar de generar código C, genere código en otro lenguaje de programación, o código objeto para una máquina determinada, o no genere código en absoluto, sino que interprete directamente el sistema especificado.

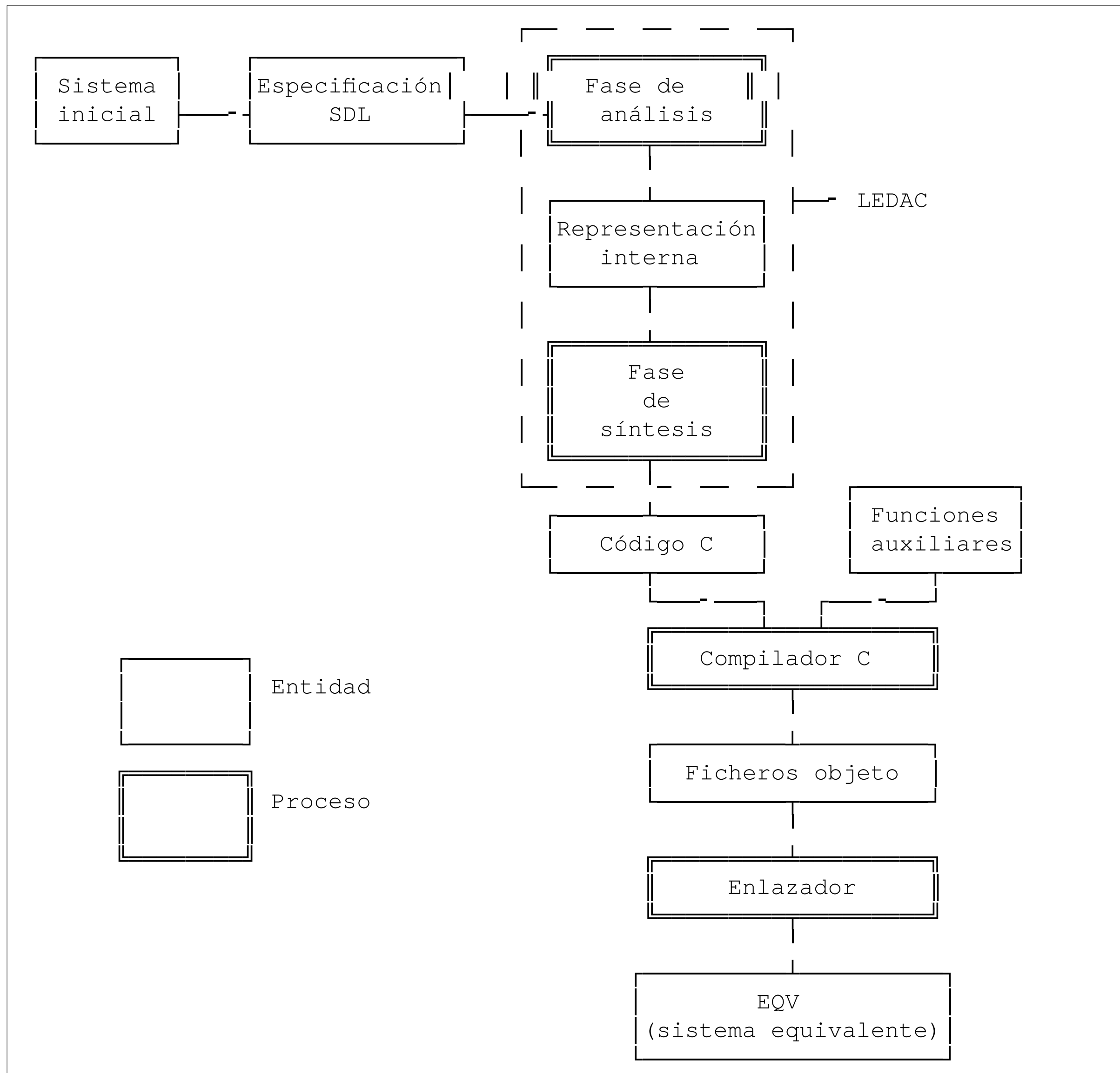
La fase de síntesis, tal y como está implementada originalmente, tiene como misión crear una serie de ficheros que contendrán el código C que constituye el equivalente al código SDL de entrada, según el esquema de traducción que se describirá más adelante en este documento. Este esquema de traducción es una de las piezas claves del proyecto y determina cómo se implementará en el lenguaje C cada una de las características del lenguaje SDL.

El programa traductor se llamará LEDAC y contendrá las fases de análisis y de síntesis, así como código para iniciar el sistema y ejecutarlas. Otro programa que se creará como parte del proyecto es LEDPP, un preprocesador que gestionará las macros que puedan existir en un programa SDL.

Los ficheros de código C que se generarán como salida del traductor habrán de ser compilados y enlazados con una librería de funciones de

ejecución y auxiliares para formar el sistema equivalente (es decir, el programa ejecutable que simulará el sistema especificado, que se llamará EQV).

El proceso desde la concepción de un sistema hasta que se obtenga a partir de él el programa EQV se muestra gráficamente a continuación.



Como se ve, la especificación del sistema en SDL ha de pasar por las fases de análisis y de síntesis (es decir, por el traductor LEDAC), y el código C resultante ha de ser compilado y enlazado junto con las funciones auxiliares.

A pesar de que, como se ha dicho, el sistema traductor consta de al

menos dos programas, de ahora en adelante el nombre LEDAC designará al sistema completo, no al programa de traducción (salvo que se indique lo contrario).

Ambos programas, y algunos otros de depuración y auxiliares, han sido escritos en lenguaje C. Se ha tratado de conseguir la mayor portabilidad posible. A pesar de que LEDAC se ejecutará principalmente sobre ordenadores SUN Microsystems con sistema operativo UNIX (SUN OS), también funciona sobre plataformas MS-DOS.

En los sistemas UNIX, LEDAC ha sido compilado con éxito usando los compiladores gcc, acc y cc. En los sistemas basados en MS-DOS, se ha usado Microsoft C V6.00. Aparte de estos compiladores de C, son necesarias para la compilación las herramientas lex y yacc, u otras compatibles con éstas (flex, bison, pclex, pcyacc...).

Junto con los compiladores y las herramientas de programación citadas se ha usado un gran número de otras utilidades y programas: TextEditor y DBXTool de SUN Microsystems, GDB, GhostView y MTools de la Free Software Foundation (Proyecto GNU), lint (la utilidad de chequeo de programas de UNIX), Clint de R&D Associates, CFT y CST de J.M., QEdit de SemWare Corp., WordPerfect de WP Corp., etcétera.

A continuación se describirán brevemente los dos lenguajes con los que está relacionado este proyecto: el lenguaje fuente, SDL, y el lenguaje destino (que es al mismo tiempo el lenguaje de implementación), C. Tras este corto estudio se presentarán las herramientas yacc y lex, junto con una breve referencia de las mismas, para seguidamente tratar en profundidad el programa traductor, el sistema equivalente y el preprocesador.

2.- El lenguaje SDL.

SDL (Specification and Description Language; también se usa el acrónimo español LED) es un lenguaje formal de especificación estandarizado por el Comité consultivo internacional de telefonía y telegrafía (CCITT), ahora ITU (International telecommunications union).

La descripción de SDL se publicó en el libro azul de 1988 del CCITT, con el identificativo Z.100 [CCCI88]. Esa versión es SDL88 y será la que se trate en este proyecto.

Desde el inicio de su desarrollo, en 1.972, SDL ha estado orientado a la especificación formal de sistemas, especialmente de telecomunicación. Así, puede ser usado para describir protocolos, estructuras jerárquicas, etc.

Una de las características principales de SDL es su doble representación: un sistema SDL puede estar descrito en representación gráfica o en representación textual. En este proyecto se utilizará exclusivamente la representación textual. De todas formas, existen herramientas para facilitar la conversión de uno a otro tipo de representación.

Se tratarán a continuación algunas de las características de SDL.

2.1.- Estructura.

Como lenguaje de especificación, SDL proporciona un buen número de construcciones para describir la estructura de un sistema de forma jerárquica, ordenada y modular: en forma de árbol, estando contenidos unos módulos en otros de nivel superior.

SDL permite separar la descripción de la estructura de la definición de los módulos que la componen, introduciendo así un nivel de abstracción adicional. Un elemento puede declararse dentro de la estructura, y sin embargo no definirse (es decir, no describir su contenido). Así se consigue tener una visión general de la estructura sin que ésta quede enturbiada por la abundancia de detalles sobre los miembros de la jerarquía.

Los componentes principales de una especificación jerárquica en SDL son los bloques: un sistema, para SDL, consta de uno o más de éstos. Un bloque es una unidad contenedora que puede albergar definiciones de otros tipos de unidades jerárquicas, de tipos de datos, de señales, de vías de comunicación...

Así pues, la partición inicial de un sistema en SDL se hace mediante bloques. Cada uno de éstos puede a su vez ser subdividido en otros (lo que se denomina subestructura de bloque). SDL no impone un límite a la subdivisión de bloques, así que un sistema, por complejo que sea, se puede representar mediante subsistemas, a su vez divididos en partes más pequeñas.

Como se ha dicho, un bloque puede contener en su interior otros bloques, pero también definiciones de tipos de datos, de señales y otras estructuras jerárquicas. Estas otras unidades son los procesos y los procedimientos, que son los que describen el comportamiento del sistema especificado (en tanto que los bloques describen su estructura).

Los procesos y procedimientos son máquinas de estados que reaccionan frente a la llegada de señales dirigidas a ellos (más adelante se tratarán las señales y las vías de comunicación), ejecutando una serie de instrucciones y posiblemente pasando a otro estado. En realidad, la unidad principal de ejecución de SDL son los procesos. Los procedimientos son siempre invocados por los procesos para efectuar tareas repetitivas, y se utilizan para mejorar la abstracción, eliminando detalles de la especificación del proceso.

Un proceso sólo puede ser definido en el interior de un bloque. Un procedimiento sólo puede definirse dentro de un proceso o de otro procedimiento. En cuanto a la invocación, tanto los procesos como los procedimientos puede crear otros procesos e invocar a otros procedimientos.

Todas estas estructuras están enmarcadas en una de nivel máximo: el sistema SDL. Sin embargo el sistema no está aislado. SDL considera que la jerarquía de nivel superior está inmersa en un entorno del cual puede recibir y al cual puede enviar información (en forma de señales, como veremos a renglón seguido). SDL no especifica el modelo del entorno, y es responsabilidad de la implementación el decidir cómo se van a introducir en el sistema SDL las señales procedentes del exterior y cómo se van a presentar las señales de salida del sistema.

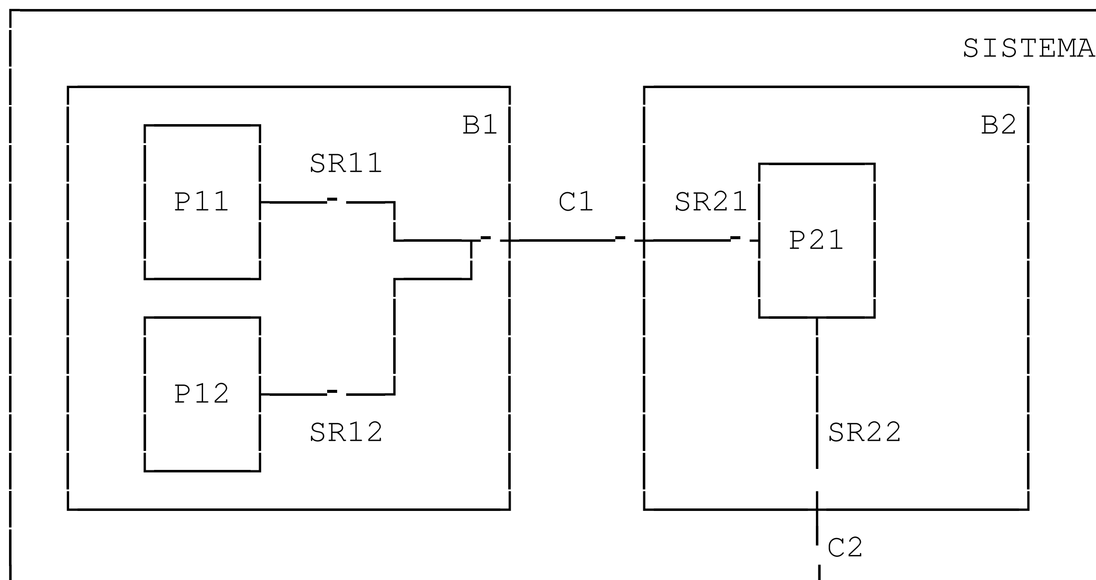
En esta implementación particular, el entorno está modelado dentro del propio sistema SDL, mediante un proceso especial definido a primer nivel, que es el encargado de interactuar con el entorno físico (el usuario, generalmente) y servir como interfaz al resto de la especificación.

2.2.- Señales y conexiones.

Se ha hablado de las señales. El sistema de comunicación entre las unidades ejecutables de SDL (los procesos y procedimientos) son las señales. Una señal es una entidad enviada desde una unidad ejecutable a otra, a través de una vía de comunicación. La señal puede tener significación completa por sí sola o puede transportar una serie de valores llamados argumentos. En este segundo caso, la unidad receptora tendrá acceso a esos argumentos y, por tanto, a información adicional.

Las vías de comunicación o vías de señales pueden ser de dos tipos, dependiendo de los elementos que conecten. Las rutas de señales (signalroutes) se definen en los bloques y comunican un proceso con otro proceso o con el entorno del bloque. Los canales sólo se pueden especificar en la jerarquía principal (el sistema) y conectan bloques con otros bloques o con el entorno del sistema SDL. Las rutas de señales que vayan al entorno del bloque deben conectarse con canales que lleguen a ese bloque. Un canal puede conectarse con varias rutas de señales, pero una ruta sólo puede estar conectada a un canal.

Se presenta a continuación un gráfico que muestra la estructura de un sistema SDL que consta de dos bloques (B1 y B2). El primer bloque tiene dos procesos (P11 y P12) y el segundo bloque tiene un proceso (P21).



Como se ve, hay dos rutas de señales (SR11 y SR12) que conectan los procesos P11 y P12 (respectivamente) con el entorno del bloque B1. Estas rutas están conectadas con el canal C1, que conecta los bloques B1 y B2. La llegada del canal C1 al bloque B2 está conectada a su vez con la ruta de señales SR21, que va hasta el proceso P21. Por lo tanto, los procesos P11 y

P21 pueden enviar señales al proceso P21 (no al contrario). Incluso pueden enviar las mismas señales, sin que P21 tenga posibilidad de saber qué proceso envió cada señal.

Por otra parte, P21 está conectado al exterior del bloque B2 mediante la ruta de señales SR22. En la jerarquía base (el sistema) está el canal C2 que recoge las señales de SR22 y las envía al entorno.

Cada vía de comunicación se especifica indicando su origen, su destino y el conjunto de señales que puede transportar. Se pueden definir vías bidireccionales, pero se consideran vías separadas.

2.3.- Comportamiento.

Como se ha visto hasta ahora, un sistema SDL puede considerarse, si del comportamiento se trata, como un conjunto de procesos (máquinas de estados) conectados entre sí por vías de comunicación, que se transmiten señales de uno a otro y cambian de estado dependiendo de la señal que reciban.

De forma más precisa, un proceso consta de una serie de transiciones. Una transición es un conjunto de instrucciones que se pueden ejecutar si, estando el proceso en un estado determinado, recibe un estímulo determinado. Estas instrucciones pueden provocar un cambio de estado del proceso (instrucción NEXTSTATE). Una transición consta, pues, de un conjunto de estados (en los cuales la transición será disparable), un estímulo y un conjunto de instrucciones.

El estímulo que puede activar la ejecución de las instrucciones de la transición puede ser:

- la presencia de una señal determinada en la cabecera de la cola de señales del proceso. Cada proceso tiene asociada una cola de señales en la que se almacenan por orden de llegada las que ha recibido, provenientes de otros procesos o del entorno. Si hay una transición disparable en el estado actual del proceso y que especifica como estímulo la señal que está la primera en la cola, se ejecutarán las instrucciones de esa transición. Si no hay ninguna transición desde el estado actual que requiera como estímulo la señal de la cabecera, no se ejecutará nada.

- el hecho de que la cola de señales esté vacía. Si no hay ninguna señal en la cola del proceso y en el estado actual hay al menos una transición de este tipo, se ejecutará. Las transiciones de este tipo se llaman señales contínuas y su ejecución siempre está condicionada por una

condición de activación (una expresión lógica que debe ser cierta para que se ejecuten las instrucciones de la transición).

- un suceso aleatorio. Existe la posibilidad de definir transiciones que se disparen de forma aleatoria. Reciben el nombre de transiciones espontáneas.

Un caso particular de transición es el resguardo de señales (SAVE). Consta de un conjunto de estados y un conjunto de señales. Su efecto es el siguiente: si el proceso está en uno de los estados de activación y en la cabecera de la cola se encuentra una de las señales especificadas, se tomará la siguiente señal y se buscará una transición para ella. La señal de la cabecera permanecerá en su puesto. Simplemente será ignorada (salvada) para ser usada posteriormente.

Todos los procesos tienen una transición inicial (START) cuyas instrucciones se ejecutarán al iniciarse el proceso. Esta transición no requiere estímulos.

En la especificación del sistema se puede indicar para cada proceso los números de instancias inicial y máximo que puede tener. Si el número inicial es mayor que cero, el proceso formará parte del grupo de procesos inicial del sistema.

La ejecución de un sistema SDL consiste en la consideración, para cada proceso en ejecución, del estado actual y de los estímulos presentes. Se comprobarán todas las transiciones disparables, se buscará una cuyo estímulo de disparo se cumpla y se ejecutarán sus instrucciones.

Las instrucciones proporcionadas por SDL posibilitan el cambio de estado de un proceso (NEXTSTATE), el envío de señales (OUTPUT), la creación de procesos (CREATE), la invocación de procedimientos (CALL), las asignaciones de variables (TASK)...

SDL también proporciona construcciones de decisión (DECISION) y de ejecución no determinista (DECISION ANY), así como saltos locales (JOIN). No hay en SDL, sin embargo, posibilidad de ejecutar bucles.

2.4.- Tipos de datos.

Los procesos y procedimientos pueden definir variables con las que trabajar, así como recibir parámetros. Los tipos de estas variables, de los parámetros, de los argumentos de las señales, etcétera, han de ser escogidos de entre los tipos básicos de SDL o definidos en función de estos.

El lenguaje SDL proporciona una serie de tipos de datos básicos, unos tipos de datos compuestos y la posibilidad de definir tipos abstractos (utilizando axiomas, en forma de álgebras iniciales).

Los tipos de datos simples son los usuales: booleano, entero, natural, real. Además hay algunos relacionados con conceptos de SDL: tiempo, duración, PID...

Estos tipos se pueden combinar para formar tipos compuestos. Para ello se utilizan los generadores predefinidos y las estructuras. Los generadores son en realidad tipos abstractos parametrizados. Hay tres de ellos incluídos en el SDL estándar: array, powerset y string. El cuarto tipo de composición de datos es la estructura, que permite definir un tipo compuesto de varios campos de tipos diferentes.

Además de los tipos simples y los generadores predefinidos, SDL ofrece la posibilidad de crear nuevos tipos de datos definiendo sus literales y operadores. Si estos tipos de datos aceptan parámetros que son otros tipos, se les llama generadores.

2.5.- Otros aspectos.

Otros aspectos avanzados de SDL son los temporizadores, que permiten a un proceso iniciar un reloj que, al llegar a un tiempo determinado, le enviará una señal. Las variables, locales al proceso que las definió, pueden ser exportadas a otros procesos mediante construcciones especiales.

No existen instrucciones de entrada/salida en este lenguaje. La comunicación con el exterior del sistema se realiza mediante señales a través de vías conectadas con el entorno. SDL no da ninguna norma sobre cómo de deben introducir las señales desde el entorno del sistema, o sobre cómo se deben leer las que salgan. Por ello, esta implementación ha introducido unas pequeñas modificaciones en la gramática del lenguaje, para permitir la descripción de un proceso llamado ENV que modelará el entorno, leyendo las señales que salgan del sistema e introduciendo otras.

Sin embargo, esto aún no proporciona una comunicación real con el exterior. Para solucionar este problema se han añadido al lenguaje dos nuevas instrucciones: Leer y Escribir. La instrucción Leer acepta una expresión modificable o LValue (ver discusión en la sección dedicada a las expresiones en el esquema de traducción) y permite introducir por teclado el valor que le será asignado. La instrucción Escribir acepta una expresión general y la imprime en la salida estándar (generalmente será la pantalla).

Con estas dos instrucciones y la unidad ENV se puede implementar el entorno del sistema de forma interactiva.

Con esto acaba esta breve introducción a SDL. Su cometido ha sido dar unas nociones básicas del lenguaje para facilitar la comprensión de las secciones que siguen. Para una explicación más detallada y amplia de todas las características de SDL, se remite a [BEL91] y a [CCI88].

3.- El lenguaje C.

El lenguaje de programación C fue desarrollado en los Laboratorios Bell de AT&T (American Telephone and Telegraph) hacia 1.972. Su creador fue Dennis M. Ritchie, ayudado por Brian W. Kernigham. Ritchie en esa época estaba trabajando junto con Kenneth Thompson en el diseño del sistema operativo UNIX.

C es un lenguaje difícil de clasificar en un nivel. Tiene características de los lenguajes de bajo nivel (como el ensamblador): el acceso directo a la memoria, a los puertos de entrada/salida, los tipos de datos relativamente simples... Sin embargo presenta una serie de características propias de un lenguaje de alto nivel (estructuras de control eficientes, facilidades para la programación estructurada...) que lo han catapultado al primer puesto entre los lenguajes preferidos para la implementación de herramientas. La principal característica del sucesor de B es la sencillez de su potencia.

A pesar de la competencia de Ada en entornos gubernamentales y militares (sobre todo estadounidenses) y en menor medida de Pascal en entornos científicos, C es uno de los lenguajes más utilizados en programación hoy en día. No sólo en el ya mencionado campo del diseño de herramientas de programación, sino en otros muchos: control, reconocimiento de formas, cálculo, gestores de BBDD e incluso aplicaciones de gestion comercial. C puede permitirse entrar en estos campos gracias a la gran cantidad de librerías de rutinas especializadas escritas para él.

No sólo librerías. También muchas herramientas de programación están especialmente orientadas a trabajar con código C. De hecho, es una clara tendencia en los sistemas UNIX actuales el sustituir los compiladores de Pascal y/o Fortran por traductores a C. Herramientas tan usadas como lex y yacc generan código C, aunque hay otras versiones que proporcionan analizadores en otros lenguajes: Pascal, Fortran y (en desarrollo) Parlog.

C es pequeño, potente, flexible y portable. Esta última cualidad es muy necesaria en un lenguaje tan usado en la implementación de herramientas, ya que éstas deben ser fáciles de portar a otros sistemas. A la portabilidad del lenguaje contribuye su estandarización. En efecto, en 1.983 el Instituto Nacional Americano de Estándares (ANSI) creó un comité llamada X3J11 para estandarizar el lenguaje C. El 14 de Diciembre de 1.989 se decidió el estándar X3.159-1989 [ANS89]. En 1.990 la Organización Internacional de Estándares (ISO) adoptó el estándar ANSI de forma internacional, llamándolo ISO/IEC 9899:1990.

La portabilidad de C queda en parte demostrada en este proyecto por

el hecho de que el código es compilable tanto en plataformas MS-DOS (bajo Microsoft C V6.0) como en plataformas UNIX genéricas (SUN en particular) bajo los compiladores cc (el compilador K&R original de UNIX), gcc (de la Free Software Foundation, Proyecto GNU) y acc. Los dos últimos compiladores, así como el de Microsoft en DOS, cumplen la norma ANSI. El compilador cc, en cambio, se basa en el 'estándar' de Kernigham y Ritchie, es decir, en su primer libro [KYR78].

4.- Introducción a yacc.

4.1.- Generalidades.

Yacc es un generador de analizadores sintácticos ascendentes de tipo LALR(1). Traduce un fichero de descripción de gramática (grammar description file, GDF) en un programa C que contiene, además de las tablas de análisis, una función entera sin parámetros (llamada yyparse) que analiza la sintaxis definida por esa gramática.

YACC fue desarrollado en 1975 en los laboratorios Bell de AT&T (lugar donde también nacieron C y UNIX) por Stephen C. Johnson y actualmente es, junto con lex, una herramienta estándar del sistema operativo UNIX. PCYACC es una implementación de yacc para ordenadores personales, creada por Abraxas Software.

Otras implementaciones de yacc son Bison (de la Free Software Foundation, proyecto GNU), byacc (de la Universidad de Berkeley), ...

El formato del fichero de descripción de la gramática que constituye la entrada de yacc es el siguiente:

```
declaraciones
%%
reglas de la gramática
%%
código
```

Tanto el último separador (%%) como la última sección (código) son opcionales.

Trataremos una por una las secciones.

4.2.- Sección de declaraciones.

La primera sección de un fichero de descripción de gramática recoge las declaraciones que serán necesarias para la correcta generación del analizador.

Se puede incluir código C en la parte de definiciones, delimitándolo con los separadores '%{' y '%}':

```
%{
<código C>
%}
```

La principal declaración que se ha de hacer es la de la unión que contendrá los atributos asociados a cada símbolo de la gramática. Esta declaración se puede hacer de dos formas:

- declarando en C un tipo llamado YYSTYPE que sea la unión que se usará como elemento de la pila de valores (o atributos)

- usando la instrucción %union, seguida de la definición de la unión en sintaxis C.

Han de declararse también todos los símbolos terminales (tokens) que se utilizarán en la gramática, excepto aquellos que consten de un sólo carácter, que pueden incluírse literalmente en las reglas sin previa declaración.

Esta declaración se hace usando la instrucción %token, seguida por los tokens que se vayan a declarar separados por espacios. Se pueden usar varias instrucciones de este tipo.

Algunos símbolos de la gramática (terminales o no) tendrán asociado un valor o atributo, que se usará para pasar información entre las reglas. La asignación de un tipo de atributo a un símbolo de la gramática se realiza mediante la instrucción %type, cuya sintaxis es la siguiente:

```
%type <campo> símbolo [símbolo ...]
```

El argumento campo es el nombre del campo de la unión que se usará como tipo de atributo de los símbolos listados.

Se pueden también usar varias instrucciones de este tipo.

Las instrucciones de definición de precedencia y asociatividad son %left, %right y %nonassoc. Seguidas por una lista de símbolos o identificadores separados por espacios, otorgan a esos símbolos la asociatividad que indica cada instrucción (izquierda, derecha y ninguna, respectivamente).

La precedencia de un símbolo es mayor cuanto más tarde aparezca su definición de precedencia. Así, en las declaraciones

```
%left '+' '-'
%left '*' '/'
%left MENOS_UNARIO
```

los símbolos '+' y '-' tienen igual precedencia y se evalúan de izquierda a derecha. Los símbolos '*' y '/' tienen más precedencia que los anteriores y también se evalúan de izquierda a derecha. El símbolo con mayor precedencia es MENOS_UNARIO, que tiene el mismo orden de evaluación. Este último no es propiamente un símbolo, sino un indicador de precedencia.

Cada regla de la gramática tiene asociada una precedencia que es la del último símbolo terminal de su parte derecha. Si dicho símbolo no tiene asignada una precedencia, la regla no tiene precedencia.

Se puede asignar a una regla una precedencia mediante la instrucción %prec.

Las relaciones de precedencia y asociatividad sirven para resolver conflictos. Veremos más adelante cómo se lleva a cabo este proceso.

La última declaración que se puede incluir en esta zona del fichero de descripción de gramática es la instrucción %start. Seguida por un símbolo no terminal, lo designa como axioma de la gramática. Si no existe esta declaración se considera que el axioma es la primera regla que se encuentre en la zona destinada a ellas.

4.3.- Sección de reglas de la gramática.

En esta sección se enumeran las reglas de derivación que componen la gramática. Las reglas se especifican en una variante de la notación BNF (Backus-Naur Form): la parte derecha se separa de la izquierda mediante un símbolo de dos puntos (':'). Las distintas alternativas se separan mediante barras verticales ('|') y cada producción se ha de terminar con un punto y coma (';').

```
parte izquierda : Pd1
| Pd2
| Pd3
...
| Pdn
;
```

Los elementos Pdx son las distintas partes derechas alternativas. Una

parte derecha puede estar vacía.

En las partes derechas de las reglas de la gramática se pueden intercalar fragmentos de código C entre llaves, que forman las llamadas acciones semánticas. Las acciones semánticas se consideran como símbolos y se ejecutan cuando son reducidas.

La parte derecha de una regla está, pues, formada por una serie de símbolos terminales, símbolos no terminales y acciones semánticas. Los tokens (símbolos terminales) de un sólo carácter de longitud se pueden poner entre apóstrofes y no tienen por qué declararse en la sección anterior.

Dentro de una acción semántica nos podemos referir al atributo de cualquier símbolo que la preceda en la parte derecha (si es que el símbolo tiene asignado un atributo) usando la notación $x, donde x es el número de orden del símbolo en la parte derecha (como se dijo, las acciones semánticas también se cuentan).

El término $$ se refiere al atributo del símbolo que forma la parte izquierda de la regla. Un ejemplo:

```
operación: opdo '+' { printf("más "); }
opdo { $$= $1+$4; }
;

opdo: NUMERO { $$= $1; printf (" %d ", $1 ); }
;
```

Para que estas reglas sean correctas, en la sección de declaraciones ha de haberse definido el token NUMERO, y los símbolos NUMERO, opdo y operación han de tener un tipo entero asignado como atributo. El atributo del símbolo terminal NUMERO lo rellenará el analizador léxico como veremos pronto.

Una gramática LALR puede contener conflictos, que serán detectados y notificados por yacc. Los conflictos que pueden aparecer en una gramática son conflictos reducción/reducción y conflictos desplazamiento/reducción.

4.3.1.- Conflictos reducción/reducción.

Un conflicto reducción/reducción se da cuando hay varias posibilidades de reducción y el token de antelación o previsión (look-ahead token) no proporciona al analizador sintáctico suficiente información para decidir cuál de varias reglas se ha de reducir.

Hay pocos ejemplos de conflictos reducción/reducción.

Cuando se da un conflicto de este tipo es que la gramática está mal construida. Yacc, por defecto, soluciona estos conflictos aplicando para reducción la regla que primero aparezca. Pero lo mejor es revisar la gramática y tratar de modificarla para resolver el conflicto.

4.3.2.- Conflictos desplazamiento/reducción.

Estos conflictos, mucho más frecuentes que los de reducción/reducción, aparecen cuando el analizador tiene un estado en el que no sabe si reducir lo que ya hay en la pila (puesto que se ajusta a una parte derecha) o seguir desplazando (leyendo tokens de entrada), puesto que hay una regla que contiene lo que ya hay en la pila y algunos símbolos más.

El ejemplo más claro y típico de este conflicto se presenta en la mayoría de las gramáticas de los lenguajes de programación: es el problema de los ELSE.

En una gramática, las dos reglas siguientes

```
instr: IF expr THEN instr
| IF expr THEN instr ELSE instr
;
```

generan un conflicto desplazamiento/reducción. Cuando el analizador reduce el primer no terminal instr, tiene en la pila la cadena de símbolos 'IF expr THEN instr'. Ahora se presenta una alternativa: ¿se debe reducir esa cadena y convertirla en la metanoción instr o por el contrario hay que seguir desplazando para tratar de obtener los símbolos 'ELSE' e 'instr'?

Yacc resuelve estos conflictos decidiéndose siempre por el desplazamiento. Así se cumple la regla de la mayoría de los lenguajes de que cada IF está emparejado con su ELSE más próximo.

Pero a veces no se desea que se tome esa acción por defecto, y se quiere que el analizador opte por la reducción. Para ayudar al control de estas acciones se usan las reglas de precedencia y asociación de la siguiente forma:

- En un conflicto reducción/reducción o desplazamiento/reducción, si

ninguna de las reglas implicadas tiene asociada una precedencia se efectúan las acciones por defecto.

- Si en un conflicto desplazamiento/reducción las reglas tienen asociadas precedencia y asociatividad, se hace lo siguiente:

  - Si la precedencia del token de entrada es mayor que la de la regla, se efectúa un desplazamiento.

  - Si es menor, se lleva a cabo la reducción.

  - Si son iguales, manda la asociatividad de la regla:

    - Si es a izquierdas, se reduce.

    - Si es a derechas, se desplaza.

    - Si no es asociativa, se genera un error.

4.4.- Sección de código.

Todo lo que se encuentra en esta sección es directamente escrito en el fichero de código C generado por yacc.

4.5.- Funciones auxiliares obligatorias.

Estas funciones son básicamente dos: la función yylex() y la función yyerror().

- La función yylex() es una función entera sin parámetros

```
int yylex( void );
```

que, en cada llamada, devuelve el código del token que lea del fichero de entrada. Esta función suele ser generada por lex, aunque nada impide construir una a mano.

La función yylex() se ocupa también de almacenar en el campo correspondiente de la unión global yylval el atributo asociado al token que devuelve, si esto es necesario. Así, al reconocer el analizador léxico un número entero, antes de retornar la constante ENTERO (por ejemplo), almacenará el valor del número en el campo correspondiente de la unión

yylval.

La función yylex() es llamada por el analizador sintáctico cada vez que éste necesita un nuevo token.

- La función yyerror() es una función sin valor de retorno que acepta un puntero a carácter (una cadena de caracteres):

```
void yyerror( char * );
```

Esta función es llamada cada vez que se produce un error. La cadena contiene el texto del error.

4.6.- Línea de comandos.

La línea de comandos de PCYACC tiene el siguiente formato:

```
pcyacc [opciones] fdg
```

Donde fdg es el nombre del fichero de descripción de gramática que queremos que PCYACC procese. Estos ficheros suelen tener la extensión '.y'.

Las opciones que acepta PCYACC son las siguientes:

-c llama al fichero de salida yytab.c, en lugar de <fdg>.c.

-C<fc> llama al fichero de salida <fc>.

-d genera un fichero llamado yytab.h que contiene los #defines de los tokens de la gramática y la declaración de la unión, entre otras cosas.

-D<fh> genera el mismo fichero, pero con el nombre <fh>.

-h muestra una pantalla de ayuda sobre las opciones.

-p<fe> utiliza el fichero fe como esqueleto (driver) del analizador.

-s hace que los elementos de las tablas generadas sean de tipo short int, en lugar de int.

-S revisa únicamente la sintaxis del fdg, sin generar nada.

-t construye el analizador de tal modo que genere un árbol de reconocimiento y lo almacene en el fichero yy.ast

-T<fa> igual que -t, pero almacena el árbol en <fa>.

-v genera una tabla de información sobre los estados del analizador en el fichero yy.lrt.

-V<ft> genera la misma tabla en el fichero <ft>.

-n no genera directivas #line, que pueden confundir a algunos depuradores simbólicos (CodeView, SymDeb, etcétera).

-r va informando de los pasos que realiza en la generación.

La línea de llamada de yacc es de la forma siguiente:

```
yacc [opciones] fdg
```

Las opciones son:

-d: genera y.tab.h (equivalente a yytab.h).

-v: genera y.output (equivalente a yy.lrt).

-l: no genera directivas #line (equivalente a la opción -n de pcyacc).

-t: #define la macro yydebug, que permite el modo de depuración.

5.- Introducción a lex.

5.1.- Generalidades.

El generador de analizadores lexicográficos lex es una herramienta que fue descrita por primera vez en un artículo de Michael Lesk y Eric Schmidt. Este último creó la primera versión del programa basándose en ideas de Johnson y Aho. Posteriormente se diseñó flex, que incorpora un algoritmo de generación más rápido y potente. Tanto lex como flex son dos de las herramientas estándar que se suministran con el sistema operativo UNIX.

Abraxas Software ha creado una versión de lex (basada en el algoritmo flex) para ordenadores personales (PC y MacIntosh) llamada PCLEX. PCLEX es compatible y superior (upward-compatible) con lex y flex. Esto quiere decir que las características de estos dos últimos son un subconjunto de las del primero. Los ficheros de descripción de PCLEX son interpretables por lex y flex si se excluyen ciertas características añadidas.

Existen también otras versiones de la utilidad lex, tanto para plataformas UNIX como DOS.

Trataremos aquí de todas las herramientas lex en general, y de lex de UNIX y PCLEX en particular.

El cometido de este generador de analizadores léxicos es crear un código fuente en C a partir de un fichero de descripción de patrones. El código C contendrá, además de las tablas de reconocimiento de patrones, una función entera sin parámetros, llamada yylex, que leerá el texto de un fichero de entrada e irá devolviendo códigos de tokens conforme vaya encontrándolos. Estos códigos devueltos por yylex serán utilizados seguramente por un analizador sintáctico construido con yacc.

El fichero de entrada es una variable externa de tipo FILE * cuyo nombre es yyin. Por defecto, está asignado a stdin, la entrada estándar. Para reasignarlo, basta con utilizar la función fopen().

La estructura de un fichero de definición de patrones es la similar a la de un fichero de descripción de gramática yacc:

```
definiciones
%%
patrones
%%
```

```
código
```

Al igual que en yacc, tanto el separador (%%) que inicia la sección de código como la propia sección de código son opcionales.

Veremos estas secciones por separado.

5.2.- Sección de definiciones.

En la parte de definiciones se pueden crear macros que serán luego utilizadas en la sección de patrones. La definición de una macro ha de comenzar en la primera columna y consta del nombre de la macro y la expansión de la macro separada de éste por uno o varios espacios. Una macro se puede expandir en la sección de patrones simplemente utilizando la construcción

```
{ macro }
```

Hay que tener muy en cuenta que PCLEX y flex, al expandir una macro, ponen entre paréntesis esa expansión, en tanto que lex no lo hace así.

En esta zona se puede insertar código C delimitándolo, al igual que en yacc, con los separadores %{ y %}.

En la sección de definiciones se han de indicar, además, las condiciones de inicio y las condiciones exclusivas (sólo en PCLEX) que se vayan a dar en la zona de patrones. Las condiciones de inicio se declaran como

```
%s nombre
    o
%Start nombre
```

y las condiciones exclusivas como

```
%x nombre
```

Más adelante veremos qué son las condiciones de inicio y exclusivas.

5.3.- Sección de patrones.

Esta parte del fichero de descripción del analizador está compuesta

por una serie de definiciones de patrones, cada uno de los cuales tiene asociada una instrucción C (que, lógicamente, puede ser una instrucción compuesta entre llaves) que se ejecutará siempre que ese patrón sea encontrado en el texto de entrada.

Cada patrón ha de estar separado por espacios de su acción C correspondiente. Si en el interior de un patrón se requieren espacios, se usarán secuencias de escape o se entrecomillarán éstos.

Un patrón es una combinación de caracteres y caracteres especiales con el cual pueden coincidir una o más palabras.

Lex sólo puede trabajar con el juego de caracteres ASCII puro, es decir, con los códigos 1 a 127 (no admite el 0). Se permiten además las secuencias de escape del C de Kernigham y Ritchie (no del C ANSI): \n, \t, \0xx (número octal), ...

Los caracteres especiales que admite lex son los siguientes:

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

Explicaremos el significado de cada uno:

- El símbolo comillas dobles ('"') sirve para delimitar cadenas literales de texto que han de ser 'encajadas' en el fichero de entrada. Los caracteres especiales pierden sus características (se convierten en caracteres normales) cuando se encierran entre comillas.

- La barra atrás (backslash, '\') convierte el carácter que la acompaña en un carácter normal. Por lo tanto, "[" y \[son equivalentes. Para indicar una barra atrás se usa la secuencia '\\'.

- Los corchetes abiertos y cerrados enmarcan una clase de caracteres: una serie de caracteres de entre los cuales se ha de tomar sólo uno. Así, la expresión [abq] encaja lo mismo con el carácter 'a' que con el 'b' que con el 'q', pero no con combinaciones de éstos.

- El acento circunflejo ('^') sólo tiene significado en dos contextos:

    - tras un corchete abierto, es decir, al principio de una clase, niega ésta. Una clase negada encajará con cualquier carácter excepto con los que pertenecen a ella. Por ejemplo, [^aeiou] permitirá cualquier carácter excepto una vocal minúscula.

    - al principio de una línea servirá como anclaje de lo siguiente.

Sólo se tomará el patrón si se halla a principio de línea. Como ejemplo: ^"#define" encajará la cadena "#define" sólo si es lo primero que aparece en una línea. No aceptará " #define".

- El guión ('-') sirve para simplificar las enumeraciones en clases: se expande a los caracteres comprendidos entre el que le precede y el que lo sucede. Así, [a-z] son todas las letras minúsculas y [0-9.] son todos los dígitos y el punto. Para incluirlo en una clase, basta con ponerlo al principio o al final de la misma: tanto [-09] como [09-] comprenden los dígitos 0 y 9 y el guión.

- El símbolo de interrogación cerrada ('?') hace que el elemento que le precede se considere opcional.

- El punto ('.') es un comodín que significa "cualquier carácter excepto un retorno de carro".

- El asterisco ('*') indica que el elemento anterior se puede repetir cero o más veces. Por tanto, t[ae]* aceptará cualquier secuencia de 'a'es y 'e's que comienze con una 't', incluída la secuencia 't'.

- El signo más ('+') indica posibilidad de repetición de lo anterior una o más veces (una vez como mínimo).

- La barra vertical ('|') hace que se encaje el elemento anterior o el elemento posterior. El patrón pep(ito|ita) aceptará 'pepito' o 'pepita'.

- Los paréntesis agrupan una serie de elementos y hacen que se consideren uno único (para usar sobre él, por ejemplo, los operadores +, * o ?).

- El símbolo de dólar ('$') debe ir al final del patrón y sirve para anclar éste a fin de línea: sólo se aceptará el patrón si tras él hay un retorno de carro.

- La barra de división permite especificar el contexto derecho necesario para la aceptación de un texto. Así, el patrón perro/s aceptará el texto 'perro' sólo si va seguido por una 's'. N.B.: el texto aceptado y recogido será 'perro'. La 's' quedará en el fichero de entrada para la siguiente búsqueda.

- Los corchetes ('{' y '}') cumplen tres misiones:

  - Con un número en su interior permiten indicar el número de repeticiones del elemento anterior. El patrón 0{25} es una secuencia de veinticinco ceros.

- Si encierran dos números separados por una coma, definen el número máximo y mínimo de veces que puede aparecer el elemento anterior: [0-9]{1,5} es un número entero de entre una y cinco cifras.

- Si contienen en su interior el nombre de una macro, se sustituyen por el texto de la macro. En PCLEX y flex, el texto de la macro se inserta entre paréntesis; en lex, sin embargo, no.

- El símbolo de porcentaje ('%'), como ya vimos, sirve para la declaración de las condiciones de inicio y exclusivas.

- Los símbolos 'menor que' y 'mayor que' ('<' y '>') determinan las reglas activas con condiciones de inicio o exclusivas. Entre estos dos símbolos se escribirán, separados por comas, el o los nombres de las condiciones de inicio o exclusivas durante el transcurso de las cuales será activo ese patrón o regla.

Tras conocer los caracteres utilizados en la construcción de los patrones explicaremos qué son las condiciones de inicio y las condiciones exclusivas.

Una condición de inicio es un estado del analizador léxico durante el cual están activas, además de todas las reglas normales, las reglas que pertenecen a esa condición de inicio.

Una regla pertenece a una condición de inicio si tiene como prefijo el nombre de ésta entre ángulos. Una regla puede pertenecer a más de una condición (ya sea de inicio o exclusiva).

Para entrar en una condición de inicio, se usa la construcción BEGIN(nombre_de_la_condición). Para terminar un estado de este tipo y volver al estado normal, se usa BEGIN(0).

Las condiciones de inicio exclusivas difieren de las normales en que durante el tiempo que el analizador se encuentre en ellas, serán válidas sólo las reglas pertenecientes a esa condición. El resto de las reglas se ignorarán.

Un caso típico de uso de las condiciones exclusivas es el tratamiento de comentarios. El siguiente fragmento de un fichero de descripción sirve para saltar los comentarios de un texto. Consideramos que un comentario comienza por '{' y termina por '}', como en Pascal.

```
"{"                     BEGIN( COMENTARIO );
```

```
<COMENTARIO>\n        ;
<COMENTARIO>"}"      BEGIN( 0 );
<COMENTARIO>.        ;
```

En la zona de definiciones deberá encontrarse la línea

```
%x COMENTARIO
```

La primera línea detecta el carácter de inicio de comentario y entra en la condición exclusiva COMENTARIO. A partir de entonces sólo son válidos los patrones de la segunda, tercera y cuarta líneas.

El patrón de la segunda línea divide los comentarios en líneas. Esto evita que un comentario demasiado largo llene el buffer del analizador y lo desborde.

En la tercera línea detectamos un carácter de fin de comentario y, usando BEGIN(0), volvemos a las condiciones normales, durante las cuales los patrones COMENTARIO no son reconocidos.

En la cuarta línea tomamos cualquier otro carácter que pueda haber dentro del comentario.

Este conjunto de patrones podría haberse sustituido por

```
"{"(.|\n)*"}"
```

Es decir, por una secuencia de cero o más caracteres cualesquiera (incluídos los saltos de línea) encerrados entre corchetes.

Pero este patrón puede provocar, si se encuentra con un comentario muy largo (o con un comentario no cerrado) un desbordamiento del analizador.

5.4.- Sección de código.

Esta última parte de un fichero de descripción de patrones es opcional, al igual que el separador (%%) que la precede.

Está pensada para que un analizador léxico pueda estar contenido en un único fichero. Todo lo que aparezca después del separador que da entrada a esta sección se copiará literalmente al fichero C generado por lex. Por lo tanto, aquí se pueden incluir las funciones que vayan a ser utilizadas por el

analizador léxico e incluso la función main si éste va a constituir un programa autónomo.

Tras estudiar las tres secciones de que consta un fichero de descripción del analizador léxico, vamos a ver cuál es la sintaxis de la línea de comandos de PCLEX y de lex, los dos principales generadores de analizadores léxicos.

5.5.- Línea de comandos.

La línea de llamada de PCLEX tiene este formato

```
pclex [opciones] fich-descr
```

El argumento fich-descr es el nombre del fichero de descripción de patrones que se quiere traducir a C. Estos ficheros suelen llevar, por convenio, la extensión '.l'.

Las opciones que acepta PCLEX son las siguientes:

-c llama al fichero de salida yylex.c, en lugar de <fich-descr>.c.

-C<fc> llama al fichero de salida <fc>.

-h muestra información de ayuda sobre las opciones.

-i genera un analizador que no distingue mayúsculas de minúsculas (case-insensitive). Por defecto, las distingue.

-l no genera directivas #line, que pueden confundir a algunos depuradores simbólicos (CodeView, SymDeb, etcétera).

-p<fe> utiliza el fichero <fe> como esqueleto (driver) del analizador.

-s si el analizador no puede encajar un texto en ninguno de los patrones, lo imprime en pantalla por defecto. Con esta opción, se detiene con un error.

La línea de llamada de lex es de la forma siguiente:

```
lex [opciones] fich-descr
```

Lex acepta las opciones -f (ejecución rápida), -v (genera estadísticas), -n (no las genera) y -t (genera el código en la salida estándar).

6.- El traductor (LEDAC).

6.1.- Terminología.

Antes de comenzar las explicaciones sobre el traductor, presentaremos y definiremos algunos términos que son bastante usados en la discusión.

- Unidades de rango: las unidades de rango o unidades de visibilidad son entidades que permiten que otras entidades se definan en su interior.

- Unidades: una unidad es un proceso o un procedimiento. Se usa este término cuando se habla de cualquiera de los dos. Así, por ejemplo, la tabla de entidades en ejecución, que forma parte del sistema equivalente, no se llama tabla de procesos (como es lo tradicional), ya que eso induciría a cofusión. Se llama, en cambio, tabla de unidades.

- Jerarquía: un sistema SDL está estructurado en forma de árbol: la unidad de rango superior (SYSTEM) contiene una serie de bloques, y éstos a su vez contienen otros bloques y unidades. El sistema está organizado como una jerarquía. El término se aplica a la estructura general del sistema. Pero también puede referirse a uno de los elementos de ésta. El significado quedará claro por el contexto.

- Vías: al igual que el término 'unidad' significa tanto 'proceso' como 'procedimiento', el término 'vías' (más correctamente 'vías de señales') significa indistintamente 'ruta de señales' y 'canal'.

- Listas y listas de referencias: muchas de las estructuras que se utilizan este traductor son enlazables. Es decir, pueden formar listas encadenadas. Todas las estructuras enlazables tienen un campo cuyo tipo es el mismo de la estructura y cuyo nombre (por establecer un criterio homogénero y para permitir el uso de macros de tratamiento genérico de listas) es <sig>.

Estas estructuras enlazadas suelen utilizarse para almacenar los elementos de un determinado tipo definidos hasta el momento. Así sucede con las listas de tipos, de variables, de unidades... Pero a veces es necesario referenciar unos cuantos de estos elementos, pero

a) no todos los definidos,

b) no en el mismo orden en que están dispuestos en la lista y/o

c) no pertenecientes a la misma lista (tipos definidos en distintas unidades de visibilidad).

Se podría crear una nueva lista duplicando los elementos necesarios, pero enlazándolos de otra manera. Esto sería costoso en tiempo y memoria y, además, dificultaría enormemente las actualizaciones de los elementos, ya que éstas deberían producirse en cada una de las copias.

En estos casos, se necesita una lista de referencias a elementos del tipo necesario. Se definen, pues, estructuras enlazables cuyos únicos campos son un puntero al elemento en cuestión y un campo de enlace. Estas estructuras suelen tener el mismo nombre que la estructura que describe al elemento, pero con el sufijo 'L_'. Así, por ejemplo, la estructura TIPO define un tipo, o una lista de ellos, puesto que es enlazable. Sin embargo, la estructura L_TIPOS permite crear listas de referencias a estructuras TIPO. Lo mismo se puede aplicar a las estructuras VAR y L_VARS, EXPR y L_EXPRS, etcétera.

- Transiciones e instrucciones: en SDL, una transición es el conjunto de instrucciones que se ejecutan al cambiar de un estado a otro. El traductor que se está describiendo usa una notación diferente: una transición es un conjunto que consta de una serie de estados, un estímulo y una serie de instrucciones. La semántica de una transición es la siguiente: cuando la unidad a la que pertenece la transición se halla en uno de los estados y aparece el estímulo, se ejecutan las instrucciones. El estímulo puede ser uno de los siguientes:

- Una señal determinada está en la cabecera de la cola de señales de la unidad. En SDL, esto se representa con la construcción INPUT <señal...>.

- La cola de señales de la unidad está vacía. Esto no es otra cosa que una señal contínua.

- Se ha decidido ejecutar una transición espontánea (¿quién lo ha decidido? digamos que el azar). La construcción en SDL es INPUT NONE.

- Hay una señal en la cola de señales que debe ser resguardada, es decir, en el sistema SDL se especificó SAVE <señal...>.

El único caso en que las instrucciones no son ejecutadas es el último, ya que una directiva SAVE no tiene instrucciones asociadas.

6.2.- Los ficheros.

El código fuente del traductor se encuentra contenido en una serie de ficheros de cabecera (.h), descripción de gramática (.y), descripción de analizador léxico (.l) y código C (.c). Aparte de estos ficheros, hay dos más para las utilidades MAKE de UNIX y NMAKE de Microsoft C v6.0 bajo DOS.

Listaremos a continuación los nombres de los ficheros que constituyen el código del traductor y daremos una breve explicación sobre su contenido.

- **ABREFICH.C**
- **ABREFICH.H**: estos dos ficheros contienen el código y el prototipo, respectivamente, de la función AbreFichero().

- **AUXL.C**
- **AUXL.H**: funciones auxiliares y sus prototipos: DupCad(), CadIguales(), LiberaID()...

- **DEB.C**
- **DEB.H**: función DebEmite(), flags y macros necesarias para la depuración.

- **DEFS.H**: definiciones de tipos básicos (UWORD, BYTE, ID, CADS...), macros y constantes importantes (TRUE, FALSE, MAX_NOMBRE...).

- **ERR.C**
- **ERR.H**: función Error(), su prototipo y constantes asociadas.

- **EXPRS.C**: funciones de tratamiento de expresiones: control semántico, generación de representación interna...

- **GENERA.C**
- **GENERA.H**: back-end. Funciones de generación del sistema equivalente y definiciones relacionadas.

- **GETOPT.C**
- **GETOPT.H**: tratamiento de la línea de comandos.

- **JER.C**: funciones de manejo de la jerarquía.

- **LISTAS.H**: macros para simplificar las operaciones sobre listas enlazadas.

- **LISTASIS.C**: funciones de listado del sistema SDL. Se usan para depuración.

- **MAIN.C**: funciones principales. Exploración de la línea de comandos.

Apertura de ficheros. Llamada a las fases de análisis y síntesis. Generación de informes.

- **MAKEFILE**: fichero de construcción del sistema actual. Puede ser MAKEFILE.DOS o MAKEFILE.UNX dependiendo de la plataforma.

- **MAKEFILE.DOS**: fichero de construcción para la utilidad make (NMAKE) de Microsoft C v6.0.

- **MAKEFILE.UNX**: fichero de construcción para la utilidad MAKE de UNIX.

- **MEM.C**
- **MEM.H**: funciones y macros de reserva y liberación de memoria.

- **PARSER.C**: fichero generado por la utilidad yacc (o pcyacc) a partir del de descripción de la gramática. Contiene el algoritmo de análisis LALR(1), las tablas y las acciones semánticas.

- **PARSER.Y**: fichero de especificación de la gramática, con acciones semánticas asociadas a las reducciones.

- **PRTFUN.H**: fichero que facilita el uso de dos clases de prototipos de funciones: ANSI y K&R (UNIX cc). En el estándar ANSI de C se pueden especificar los tipos de los argumentos en un prototipo de función. Esto permite al compilador un mayor control de la semántica. Sin embargo, en el compilador cc de UNIX (y, en general, en todos los compiladores que usan el C K&R) no se permite esta característica.

Este fichero contiene una macro PRTFUN que permite describir un prototipo ANSI (con especificación de los tipos de argumento) y, en un compilador K&R, ocultar estos datos, que no serían aceptados.

- **SCANNER.C**: funciones de análisis léxico generadas por la utilidad LEX (o PCLEX) a partir del fichero SCANNER.L.

- **SCANNER.L**: descripción del analizador lexicográfico.

- **SENALES.C**: tratamiento y chequeos semánticos relativos a las señales.

- **STRUCTS.H**: fichero principal de definiciones: en él se hallan descritas todas las estructuras de representación (JER, SENAL, TIPO, INSTRUCCIONES...) y se encuentran los prototipos de la mayoría de las funciones de tratamiento y semántica.

- **TIPOS.C**: funciones de control y procesado relativas a tipos.

- **UNIDADES.C**: funciones relativas a unidades. Efectúan tareas de manejo y control semántico de las mismas.

- **VARS.C**: funciones semánticas relativas a variables.

- **YACCPAR.ESQ**: esqueleto de la función de análisis LALR(1) con manejo de errores mejorado. Se puede usar en analizadores generados por pcyacc. La herramienta yacc de UNIX no permite especificar otra función de análisis que no sea la contenida en el fichero /usr/lib/yaccpar.

- **YYERRS.C**
- **YYERRS.H**: funciones de error sintáctico llamadas por el analizador LALR(1). Hay dos versiones: las sencillas, llamadas por los analizadores generados por yacc, y las mejoradas, que son usadas por los analizadores que genera pcyacc usando el esqueleto YACCPAR.C.

- **YYTAB.H**: definiciones de la unión usada para transferir atributos entre el analizador léxico y el sintáctico y de los códigos correspondientes a los símbolos terminales.

- **YYTOK.H**: representación textual de los símbolos terminales de la gramática. Será usada por las funciones de error sintáctico mejoradas. Es generado a partir de YYTAB.H por la herramienta TOKENS. Sólo es útil si se está generando el analizador con pcyacc.

6.3.- La fase de análisis.

6.3.1.- Estructuras.

- **De representación.**

Como se dijo antes, el traductor está dispuesto en dos fases: la de análisis y la de síntesis. La información sobre el sistema de entrada que es recogida por la fase de análisis se almacena en una serie de estructuras en memoria para ser utilizada posteriormente tanto en los siguientes pasos de la fase de análisis como durante la síntesis.

A estas estructuras en memoria se las llamará genéricamente 'estructuras de representación'. Veremos a continuación cada una de ellas, explicando la función de sus componentes. El orden de descripción es el siguiente: comenzaremos por la estructura JER y, tras cada estructura, trataremos las que hayan sido referenciadas por ella y no estén aún descritas. Como se ve, seguiremos una estructura arborescente.

- **JER**:

Las estructuras de tipo JER se ocupan de mantener el orden jerárquico entre los componentes del sistema SDL. Están preparadas para formar un árbol a partir de la estructura inicial, que representa la raíz del sistema.

Una estructura de tipo JER puede describir un bloque, un proceso, un procedimiento... En general, una unidad de rango.

Las estructuras tipo JER tienen un campo <sig> que permite referenciar estructuras del mismo tipo y, por consiguiente, crear listas enlazadas. Así, una estructura JER se puede considerar como un elemento único o como la cabeza de una lista (si el campo <sig> es no nulo). Un segundo campo de referencia (<padre>), permite crear estructuras en árbol con referencias ascendentes.

Sus componentes son:

- nombre [char *]: contiene el nombre de la unidad de rango descrita.

- tipo [BYTE]: define el tipo de unidad de rango que describe la estructura. Se utilizan para ello las constantes simbólicas de tipo UDR_... (UDR_SYSTEM, UDR_BLOCK, UDR_PROCESS, etc, excepción hecha de

UDR_SIGNAL y UDR_TYPE, que no se describen mediante estructuras JER.

- estado [BYTE]: indica el estado de la unidad de rango descrita: referenciada o definida. Se usan las constantes EST_REF y EST_DEF. Sólo se pueden mantener referencias a determinados tipos de unidad de rango.

- bloques [JER *]: apunta a una lista de estructuras JER (posiblemente vacía, posiblemente unitaria) que describen los bloques que han sido definidos dentro de esta unidad de rango. Sólo puede ser considerado en el caso de que la unidad de rango actual permita definir bloques en su interior, como es el caso del sistema y otros bloques.

- unidades [JER *]: es un puntero a una lista de unidades (descritas como estructuras JER) pertenecientes a esta unidad de rango. Sólo válido si se pueden definir unidades en ésta jerarquía.

- tipos [TIPO *]: lista de tipos definidos en la unidad de rango actual. Cualquier unidad de rango que se pueda representar mediante estructuras JER permite la definición de tipos en su interior.

- senales [SENAL *]: apunta a la lista de señales que se hayan definido en la unidad de rango.

- unidad [UNIDAD *]: en caso de que la jerarquía actual sea una unidad (UDR_PROCESS o UDR_PROCEDURE), este campo apunta a la descripción de dicha unidad.

- constantes [CTE *]: puntero a la lista de constantes definidas en la unidad de rango.

- vias [VIA_SENALES *]: describe las vías (canales o rutas de señales) que se han declarado en la jerarquía actual.

- signallists [LISTA_SENALES *]: las listas de señales que se han definido en la unidad de rango.

- padre [JER *]: apunta al padre de la jerarquía definida. Sólo puede ser NULL si la jerarquía es la raíz del sistema (tipo UDR_SYSTEM).

- sig [JER *]: si la jerarquía en definición forma parte de una lista y no es el último elemento, <sig> apunta a la siguiente estructura JER en la lista. Si no forma parte de una lista o es el último elemento, <sig> es NULL.

- **TIPO**:

Las estructuras TIPO describen los tipos predefinidos del lenguaje (BOOLEAN, INTEGER, DURATION...) y los que se definan en el sistema SDL. Al igual que las estructuras JER, tienen un puntero <sig> que les permite enlazarse formando listas.

Los campos de las estructuras TIPO son:

- nombre [char *]: nombre del tipo.

- tipo [BYTE]: clase de tipo. Usa las constantes TIPO_... que definen los tipos básicos, los generadores POWERSET, ARRAY y STRING, las estructuras y los tipos enumerados (LITERALS).

- datos [unión]: la unión <datos> contiene las precisiones necesarias dependiendo de la clase de tipo que se describa:

- Si el tipo es TIPO_POWERSET (conjunto), el campo <item> [TIPO *] apunta al tipo de los elementos que contendrá ese conjunto.

- Si el tipo es TIPO_ARRAY (vector), la estructura <array> contiene campos que indican los tipos del índice y de los elementos del vector.

- Si el tipo es TIPO_STRING (cadena), la estructura <string> describe el tipo de los elementos de la cadena.

- Si el tipo es TIPO_STRUCT se usa la estructura <estruct>, que describe los nombres de los campos (campo <nombres> [CADS *]) y sus tipos (campo <tipos> [L_TIPOS *]).

- Si el tipo es cualquier otro, la unión datos no es necesaria.

- sig [TIPO *]: es el puntero que permite crear listas enlazadas de tipos.

- **CADS**:

CADS es una estructura que permite representar listas de cadenas. Sus campos son:

- cad [char *]: la cadena.

- sig [CADS *]: el puntero al siguiente elemento de la lista (NULL si no hay más).

- **L_TIPOS**:

La estructura L_TIPOS se usa para crear listas de referencias a tipos. Obsérvese que se habla de listas de referencias, no de listas de tipos. Para crear listas de tipos se pueden usar estructuras TIPO. Pero muchas veces (por ejemplo, en listas de tipos de argumentos) es necesario especificar una serie de tipos que pueden estar definidos en diferentes lugares. Estos tipos están insertos en la lista de tipos de una estructura JER, y sus campos <sig> se usan para mantener dicha lista. Por consiguiente no se pueden enlazar. Para ello se usa la estructura L_TIPOS.

Sus campos son:

- tipo [TIPO *]: puntero al tipo.

- sig [L_TIPOS *]: puntero al siguiente elemento de la lista.

- **SENAL**:

Una estructura de tipo SENAL se emplea para almacenar los datos relativos a una señal definida. Se puede enlazar en listas.

Sus campos:

- nombre [char *]: el nombre de la señal.

- id [UWORD]: identificador único de la señal. Se usará en el sistema equivalente.

- tipos_args [L_TIPOS *]: tipos de los argumentos de la señal. NULL si no tiene argumentos.

- es_timer [BYTE]: indicador de que la señal es un temporizador.

- sig [SENAL *]: puntero de enlace.

- **UNIDAD**:

Esta estructura describe detalladamente una unidad (proceso o procedimiento). Es el complemento a la estructura JER que coloca la unidad en su lugar en la jerarquía.

Sus campos son:

- id [UWORD]: identificador unívoco de la unidad. Se utilizará en el sistema equivalente.

- inst_min, inst_max [UWORD]: número de instancias mínimo y máximo. Sólo se usan si la unidad es un proceso. Se considera INST_INF el valor infinito.

- num_vars [UWORD]: número de variables propias de la unidad (no se cuentan las heredadas). Se cuentan como variables los argumentos.

- num_args [UWORD]: número de argumentos de la unidad.

- vars [VAR *]: lista que describe las variables de la unidad.

- num_estados [UWORD]: número de estados de la unidad. No se contabiliza el estado START.

- estados [CADS *]: lista de cadenas con los nombres de los estados definidos en la unidad, excepto START.

- transiciones_espontáneas [TRANSIC *]: lista de las transiciones espontáneas descritas en la unidad.

- senales_continuas [TRANSIC *]: lista de las señales contínuas descritas en la unidad.

- transicion_start [INSTRUCCIONES *]: instrucciones a ejecutar al inicio de la ejecución (START).

- resto_transiciones [TRANSIC *]: lista de las transiciones definidas en la unidas.

- num_llamadas_pto [UWORD]: número de instrucciones CALL ejecutadas en la unidad. Se usará en la generación del sistema equivalente.

- **VAR**:

Las estructuras de tipo VAR describen las variables de una unidad. Se pueden enlazar en listas.

Sus componentes son:

- nombre [char *]: nombre de la variable.

- id [UWORD]: identificador de la variable. Debe ser único dentro de una unidad.

- tipo_decl [BYTE]: tipo de declaración de la variable: normal (DCL_NRM), visible (DCL_REV), exportada (DCL_EXP) o visible-exportada (DCL_REX).

- es_par_inout [BYTE]: indica si la variable es un parámetro declarado como de entrada/salida.

- tipo [TIPO *]: tipo de la variable.

- inicial [EXPR *]: expresión de inicialización de la variable (puede ser NULL).

- poseedor [JER *]: este campo permite saber a qué unidad pertenece la variable.

- sig [VAR *]: campo de enlace.

- **EXPR**:

Esta es una de las estructuras más importantes. Describe una expresión, es decir, una serie de elementos (funciones, constantes, variables...) enlazados por operadores.

Sus componentes son:

- tipo [BYTE]: indica la clase de expresión. Se usan las constantes simbólicas EXP_..., entre las que se pueden citar EXP_VINT (constante de tipo entero), EXP_VAR (variable), EXP_OPI (operación infija), EXP_FUNCION (aplicación de función), etcétera.

- t [TIPO *]: tipo de la expresión.

- datos [unión]: la unión <datos> da más información sobre la expresión, en caso de que sea necesaria.

- Si la expresión es una constante INTEGER, REAL, BOOLEAN, PID, CHARSTRING o CHAR, los campos <vinteger>, <vreal>, <vboolean>, <vpid>, <vcharstring> o <vchar> la contendrán.

- Si la expresión es una variable, se almacena una referencia en el campo <var>.

- Si la expresión es una aplicación de operador infijo, la estructura <operador> contiene campos que definen los dos operandos (dos estructuras de tipo EXPR a su vez) y el operador (un campo de tipo BYTE que puede contener las constantes OPI_...).

- Si la expresión es la aplicación de un operador unario ('-' o NOT) se usa la misma estructura, pero se ignora el segundo operando.

- Si la expresión es una aplicación del operador IF-THEN-ELSE, la estructura <ite> contiene tres expresiones que definen la condición, el valor en caso de que ésta sea cierta y el valor en caso de que sea falsa.

- Las referencias a estructura utilizan la estructura <ref_estr> (valga la redundancia), que contiene un campo de tipo EXPR * para indicar la estructura referenciada y otro campo que indica el número del campo dentro de la misma.

- Si se trata de la aplicación de una función, la estructura <funcion> proporciona un campo de tipo BYTE que (mediante una constante FUN_...) define la función y otro campo (una lista de expresiones: L_EXPRS *) que referencia los argumentos a la misma.

- Las referencias a vectores se representan en la estructura <ind_array>, que contiene la expresión de tipo array y la expresión que se usa como índice.

- Si la expresión es la aplicación de la función ACTIVE, un campo de la estructura <active> apunta a la señal (temporizador en este caso) y otro campo contiene una lista de expresiones a usar como argumentos de la señal.

- Las funciones IMPORT y VIEW se describen en la estructura <import_o_view> mediante dos campos: uno apunta a la descripción de la variable externa (EXTERNA *) y el otro es una expresión (opcional) de tipo PID que indica la unidad de la que se debe tomar la variable.

- **L_EXPRS**:

Esta estructura permite definir listas de referencias a EXPR. Se usan, por ejemplo, en listas de parámetros reales.

Sus campos son:

- expr [EXPR *]: la expresión que constituye el elemento de la lista.

- sig [L_EXPRS *]: el puntero de enlace.

- **EXTERNA**:

Las estructuras de tipo EXTERNA describen variables exportadas y vistas. Permiten crear referencias a una variable de estos tipos cuando ésta aún no ha sido definida. Estas referencias serán resueltas cuando la variable se declare.

Sus elementos son:

- tipo [BYTE]: tipo de variable externa: DCL_EXP si es una variable exportada. DCL_REV si es una variable vista.

- resuelta [BYTE]: indicador de si la variable externa está resuelta o sólo es una referencia.

- num_refs [UWORD]: número de referencias que se han hecho de la variable.

- j [JER *]: jerarquía a la que pertenece la variable. Sólo tiene un valor utilizable si está resuelta la referencia externa.

- v [VAR *]: variable externa. Sólo si ya está resuelta.

- nombre [char *]: nombre de la variable. Sólo es útil cuando la referencia está aún sin resolver: cuando se resuelva, el nombre estará en la estructura VAR apuntada por <v>

- t [TIPO *]: tipo de la variable. Su utilidad también está restringida a las referencias no resueltas. En las resueltas, <v> contiene este dato.

- sig [EXTERNA *]: campo de enlace.

- **TRANSIC**:

Esta estructura se usa para definir una o más (es enlazable) transiciones. Como se explicó antes, una transición es una combinación de

una serie de estados, un estímulo y una serie de instrucciones.

Los campos de una estructura de tipo TRANSIC son los siguientes:

- estados [CADS *]: lista de cadenas que representan los estados en los que es válida la transición o, si el indicador <asterisco> es TRUE, los estados en los que NO es válida.

- asterisco [BYTE]: indicador de cómo se debe usar el campo <estados>.

- tipo [BYTE]: tipo de estímulo. Se usan las constantes TTR_...

- datos [unión]: información adicional al tipo de estímulo.

- Si el estímulo es una entrada (TTR_INPUT) o se trata de una transición espontánea (TTR_ESPONTANEA), la estructura <input_o_esp> contiene campos que indican las señales de entrada (como una lista de tipo INPUTS *, que puede ser NULL si se trata de todas las señales), la posible condición de activación y si la entrada es prioritaria.

- Si el estímulo es la llegada a la cola de señales de una señal que debe ser resguardada, el campo <lista_save> es una lista de referencias a señales que indica qué señales merecen ese tratamiento. Si son todas (SAVE *), <lista_save> será NULL.

- Si el estímulo es la cola de entradas vacía (TTR_CONTINUA), los campos de la estructura <continua> determinan la posible condición de activación y la prioridad.

- instrucciones [INSTRUCCIONES *]: es la lista de instrucciones que se deben ejecutar si, en uno de los estados de activación, uno de los estímulos indicados.

- sig [TRANSIC *]: es el enlace a la siguiente estructura en la lista.

- **INPUTS**:

Las estructuras de tipo INPUTS forman listas que determinan las señales que actúan como estímulos en una transición y las variables en las que se deben almacenar los parámetros de esas señales.

Sus componentes son:

- senal [SENAL *]: señal de la que se trata.

- vars [L_VARS *]: lista de variables en las que se almacenarán los argumentos que porta la señal (si es que tiene alguno).

- sig [INPUTS *]: puntero de enlace.

- **L_VARS**:

Es una estructura que permite definir listas de variables (referencias a variables, realmente). Sus campos son

- var [VAR *]: la referencia a variable.

- sig [L_VARS *]: puntero al siguiente elemento de la lista (NULL si no hay más).

- **L_SENALES**:

Permite definir listas de referencias a señales. No confundir con las listas de señales (signallist), que se representan mediante estructuras LISTA_SENALES.

Sus campos son:

- senal [SENAL *]: puntero a la señal referenciada.

- sig [L_SENALES *]: campo de enlace.

- **INSTRUCCIONES**:

Esta estructura enlazable permite definir las instrucciones que se ejecutarán en una transición. Al igual que otras estructuras de representación consta de un campo que indica el tipo de instrucción, una unión que proporciona los datos necesarios dependiendo de ese tipo y un puntero de enlace con el siguiente elemento de la lista.

- tipo [BYTE]: es una constante simbólica TIN_... que determina la instrucción.

- datos [unión]: la unión datos contiene una serie de elementos que describen en detalle la instrucción representada:

- Las instrucciones JOIN y NEXTSTATE, así como las etiquetas (consideradas también como instrucciones) se especifican mediante una cadena de caracteres con el nombre del estado (NEXTSTATE) o de la etiqueta (JOIN o etiqueta).

- Una estructura de tipo DEC contiene los datos de las instrucciones DECISION, DECISION ANY y TRANSITION OPTION.

- La estructura <create_o_call> contiene dos campos que describen una instrucción CALL o CREATE: la unidad que se ha de crear o llamar (de tipo JER *) y la lista de argumentos que se le han de pasar (de tipo L_EXPRS *).

- Una lista de referencias a variables (L_VARS *) determina las que deben ser exportadas en una instrucción EXPORT.

- La instrucción OUTPUT está asociada con una estructura de tipo CUERPO_OUTPUT. Si es prioritaria, con una estructura SALIDAS (que, como se verá, es un subconjunto de CUERPO_OUTPUT).

- Si se trata de instrucciones SET o RESET se usa una estructura de tipo SETS_O_RESETS.

- Las tareas se describen mediante estructuras de tipo TAREA.

- sig [INSTRUCCIONES *]: es el puntero al siguiente elemento de la lista.

- **DEC**:

Las decisiones (normales, no deterministas o de transición) se representan mediante estructuras de tipo DEC. Los campos son:

- cuestion [EXPR *]: es la expresión cuyo valor se usará para decidir entre una de las respuestas.

- respuestas [RESPUESTAS *]: es una lista enlazada de posibles valores (o condiciones de rango) con instrucciones asociadas.

- hay_else [BYTE]: indicador de si hay una parte ELSE.

- trans_else [INSTRUCCIONES *]: instrucciones que se ejecutarán si ninguna de las respuestas encaja con la cuestion.

- **RESPUESTAS**:

Esta estructura se enlaza para formar listas de respuestas e instrucciones en las decisiones. Sus campos son:

- cr [COND_RANGO *]: lista de condiciones de rango que hay que comprobar con el valor de la cuestión.

- trans [INSTRUCCIONES *]: instrucciones que se ejecutarán si la cuestión coincide con la condición de rango.

- sig [RESPUESTAS *]: campo de enlace.

- **COND_RANGO**:

Permite especificar una serie de rangos de valores.

- tipo [BYTE]: clase de rango descrita. Se usan las constantes RAN_...

- datos [unión]:

  - Si el rango es un único valor (en forma de expresión), el campo <expr> contiene a ésta.

  - Si se trata de un rango cerrado, la estructura <limites> proporciona dos expresiones que marcan los límites inferior y superior.

  - En caso de que el rango sea abierto (relacional), se tiene una expresión y un operador relacional, que están contenidos en la estructura <relacional>.

- sig [COND_RANGO *]: puntero al siguiente elemento en la lista.

- **CUERPO_OUTPUT**:

Una estructura CUERPO_OUTPUT especifica las distintas señales que se han de emitir en una instrucción OUTPUT, así como el PID del destinatario (parte TO) y la ruta de envío (parte VIA). Los componentes de una estructura de este tipo son:

- salidas [SALIDAS *]: lista las señales que se han de despachar y los

argumentos de éstas.

- destino [EXPR *]: expresión de tipo PID que indica el destinatario. NULL si no se especificó.

- via [CADS *]: lista de nombres de vías que deberá atravesar la señal, si es que se especificó dicha lista mediante la construcción VIA. NULL si no se ha especificado.

- linea [int]: la línea del fichero SDL donde se encuentra la instrucción OUTPUT representada. Se usa si se encuentra algún error durante la fase de síntesis, que es cuando se calculan las unidades de destino.

- **SALIDAS**:

Esta estructura permite especificar una serie de señales (posiblemente con argumentos, dados como expresiones) que serán emitidas por una instrucción OUTPUT. Los campos de una estructura SALIDAS son:

- senal [SENAL *]: la señal que será emitida.

- args [L_EXPRS *]: los argumentos a la señal (si hay alguno).

- sig [SALIDAS *]: puntero de enlace.

- **UWORDS**:

Una lista de valores enteros sin signo. Los campos son los típicos en una lista:

- uword [UWORD]: la palabra si signo que constituye el elemento actual.

- sig [UWORDS *]: el puntero al siguiente elemento de la lista.

- **SETS_O_RESETS**:

Las instrucciones SET y RESET tienen en común que especifican una serie de señales (temporizadores) con sus argumentos. La función SET, además, necesita una expresión de tipo TIME que será el valor del temporizador indicado. Por consiguiente, la estructura SETS_O_RESETS, que se usa para describir estas instrucciones, tendrá los siguientes campos:

- senal [SENAL *]: el temporizador sobre el que se ejecutará el SET o RESET.

- args [L_EXPRS *]: los argumentos al temporizador.

- valor [EXPR *]: la expresión de iniciación del temporizador. Sólo se tiene en cuenta si se trata de una instrucción SET.

- sig [SET_O_RESETS *]: el campo de enlace.

- **TAREA**:

Hay en SDL dos tipos de tareas: las asignaciones y el texto informal. Esta estructura permite describirlas.

- tipo [BYTE]: es el tipo de tarea descrita. TAR_ASIGNACION o TAR_TEXTO.

- datos [unión]: la unión <datos> proporciona información sobre la tarea descrita.

- Si la tarea es una asignación, el campo <asignaciones> apunta a una lista de estructuras que describen las operaciones.

- Si la tarea es texto informal, el campo <cads> contiene dicho texto, en forma de lista de cadenas.

- **ASIGNACIONES**:

Con esta estructura se puede definir una serie de asignaciones.

- var [PARTE_IZQ *]: descripción de la parte izquierda de la asignación: la variable, elemento de array o campo de estructura donde debe almacenarse el valor.

- valor [EXPR *]: el valor que se debe almacenar en el espacio indicado por <var>.

- sig [ASIGNACIONES *]: siguiente elemento en la lista.

- **PARTE_IZQ**:

Define la parte izquierda de una asignación.

- tipo [BYTE]: clase de parte izquierda descrita: variable, elemento de vector o campo de estructura. Se usan las constantes PIZ_...

- t [TIPO *]: tipo de la parte izquierda. Los chequeos semánticos se encargarán de comprobar que coincide con el de la expresión a la derecha de la asignación.

- datos [unión]: da información sobre el elemento descrito.

  - Si es una variable, el campo <var> (de tipo VAR *) apuntará a ella.

  - Si es una referencia a vector, el vector y la expresión que indexa sobre él están contenidos en la estructura <ind_arr>.

  - En caso de que se trate de un campo de estructura se usa <ref_estr>, en cuyos elementos se describen la estructura y el número del campo.

- **CTE**:

Las constantes definidas en el sistema SDL (ya sea mediante enumeración en LITERALS o mediante SYNONYM) se almacenan en listas de constantes, formadas mediante el enlace de estas estructuras.

- nombre [char *]: nombre de la constante.

- valor [EXPR *]: valor de la constante. Determina también el tipo de la misma.

- sig [CTE *]: puntero de enlace.

- **VIA_SENALES**:

La estructura VIA_SENALES es de tipo enlazable (contiene un puntero al siguiente elemento de la lista) y permite especificar vías de señales: canales y rutas.

- tipo [BYTE]: tipo de vía: VIA_CANAL si es un canal (conexión entre

bloques) o VIA_RUTA si es una ruta de señales (conexión entre unidades).

- nombre [char *]: nombre de la vía de señales.

- tipo_from [BYTE]: tipo de elemento al que está conectada la entrada del conducto de señales en definición: CON_VIA si es la salida de otra vía. CON_JER si es una jerarquía (un bloque o una unidad, dependiendo del valor de <tipo>).

- datos_from [unión]: datos sobre la conexión de entrada a la vía actual:

- Si la entrada es una o más vías, el campo <vias> las representa.

- Si la entrada es una jerarquía, ésta está apuntada por el campo <jer>.

- tipo_to [BYTE] y datos_to [unión]: estos dos campos tienen el mismo significado que los anteriores, pero se aplican a la salida de la vía.

- with [L_SENALES *]: conjunto de señales que pueden circular por la vía definida.

- via_inversa [VIA_SENALES *]: puntero a la estructura VIA que describe el canal o ruta de señales inverso, si es una vía bidireccional. NULL si es una vía unidireccional.

- sig [VIA_SENALES *]: siguiente elemento de la lista.

- **L_VIAS**:

Sencillamente, una lista de vías de señales.

- via [VIA_SENALES *]: la vía referenciada por el elemento de la lista.

- sig [L_VIAS *]: el siguiente elemento en la lista.

- **LISTA_SENALES**:

Con la estructura LISTA_SENALES se pretende representar una construcción SDL de tipo SIGNALLIST. Estas listas son enlazables.

- nombre [char *]: nombre de la lista de señales.

- lista [L_SENALES *]: señales que la componen.

- sig [LISTA_SENALES *]: puntero al siguiente elemento.

- **Otras estructuras**.

Las siguientes estructuras no han sido referenciadas por ninguna de las anteriores. Por tanto, no forman parte de las estructuras de representación propiamente dichas. Sin embargo se usan en el sistema como elementos auxiliares y son de gran importancia.

- **ID**:

Esta estructura representa un identificador SDL. Estos consisten en un nombre opcionalmente precedido por un cualificador. El cualificador es una lista de unidades de rango separado por '/'. Cada unidad de rango consiste en un tipo de unidad de rango (BLOCK, PROCEDURE, PROCESS, SERVICE, SIGNAL, SUBSTRUCTURE, SYSTEM o TYPE) y su nombre. Los campos de la estructura ID reflejan directamente esta definición:

- cualif [ITEMS_DE_RUTA *]: lista de unidades de rango que constituyen el cualificador.

- nombre [char *]: nombre representado por el cualificador.

- **ITEMS_DE_RUTA**:

Las estructuras ITEMS_DE_RUTA se enlazan constituyendo el cualificador de un identificador. Sus campos son:

- unidad_de_rango [BYTE]: una constante UDR_... que indica el tipo de unidad de rango a que se refiere el ítem.

- nombre [char *]: el nombre de la unidad de rango.

- sig [ITEMS_DE_RUTA *]: puntero al siguiente elemento de la lista.

- **IDS**:

Las estructuras IDS posibilitan la creación de listas de identificadores.

- id [ID *]: el identificador.

- sig [IDS *]: el siguiente elemento.

- **L_CADS_ID**:

Una lista de pares CADS-ID (es decir, listas de cadenas-identificador) se puede representar mediante una serie de estructuras L_CADS_ID encadenadas. Su utilidad suele ser la descripción de series de nombres con un tipo asociado.

- cads [CADS *]: lista de cadenas.

- id [ID *]: identificador asociado.

- sig [L_CADS_ID *]: enganche con el elemento posterior.

- **L_JERS**:

No es otra cosa que una lista de referencias a jerarquías. Una vez más se debe resaltar que se trata de referencias: las listas de jerarquías se crean con las propias estructuras JER.

- jer [JER *]: la referencia a jerarquía.

- sig [L_JERS *]: el siguiente elemento.

- **L_TRANSIC**:

Al igual que la anterior, esta lista contiene referencias, pero esta vez a transiciones. Sus campos son:

- trans [TRANSIC *]: la transición referenciada.

- sig [L_TRANSIC *]: la siguiente estructura en la lista enlazada.

6.3.2.- Las variables globales.

El traductor utiliza una serie de variables globales que almacenan información a la que deben acceder diferentes funciones. Analizaremos a continuación las variables globales más importantes, dando el tipo y una breve descripción de cada una de ellas.

- yyin [FILE *]: esta variable, usada por el analizador léxico pero iniciada por la función principal, contiene el descriptor asociado al fichero de entrada: el fichero fuente SDL.

- yyline [int]: la variable yyline se usa para almacenar el número de la línea actualmente en proceso del fichero de entrada (yyin). Se utiliza en los mensajes de error y de depuración y se actualiza en el analizador lexicográfico.

- yyfile [char [MAX_NOMFICH]]: contiene el nombre completo del fichero de entrada. Se asigna en la función principal y se usa en depuración y salidas de error.

- yylval [YYSTYPE]: yylval es la estructura que se usa para pasar información (atributos) sobre símbolos terminales entre el analizador léxico y el sintáctico. El tipo YYSTYPE es la unión definida al inicio del fichero de descripción de gramática. Es el tipo de la pila de atributos del analizador. Se usa, pues, en los analizadores léxico y sintáctico.

- sistema_SDL [JER *]: apunta a la jerarquía raíz del sistema definido, es decir, a una estructura de tipo JER con el campo <tipo> asignado a UDR_SYSTEM. De esta estructura deriva toda la jerarquía que representa el sistema fuente. Se utiliza en las funciones de manejo de la jerarquía y en las de generación de código.

- jer_actual [JER *]: es un puntero a la jerarquía que se está definiendo. Se usa en múltiples funciones.

- jer_env [JER *]: apunta a la pseudounidad entorno si está definida (extensiones ENVIRONMENT y ENDENVIRONMENT). Vale NULL si no se ha definido aún. Se usa en las funciones de conexión de vías de señales para evitar la búsqueda de la unidad.

- referencias [L_JERS *]: mantiene una lista de jerarquías que están referenciadas pero no definidas. Cuando se encuentra una definición de jerarquía fuera del sistema (después de ENDSYSTEM), debe haberse hecho una referencia a esa jerarquía. Si la definición tiene un identificador cualificado, es fácil encontrar la referencia correspondiente (si existe). Si no lo tiene, se busca en la lista de referencias, lo que acelera el proceso. Ver la explicación de la función InstalaEnJerarquia() para más información.

- pila_jer [JER **]: es un puntero a puntero a JER que, al ser iniciado mediante calloc(), se podrá usar como un vector de punteros a JER. Mantiene una estructura LIFO (es decir, una pila) de jerarquías. Se usa en el analizador sintáctico para representar la secuencia de definiciones. Así, si en una jerarquía A (la jerarquía actual: jer_actual) se encuentra la definición de otra B, se introduce la primera en la pila y se asigna la nueva jerarquía a jer_actual. Al acabar la definición de la jerarquía B, se asigna a jer_actual el valor que se extraiga de la pila, es decir, la jerarquía A, y se continúa con la definición de ésta.

- punt_pila_jer [UWORD]: puntero asociado a la pila de jerarquías. Da el índice del siguiente hueco en la pila.

- contador_senales [UWORD]: se usa para asignar los identificadores a las señales que se definan, durante la fase de análisis. Durante la fase de síntesis, su valor es el número de señales del sistema (constante NUM_SENALES en el fichero _ctes.h). Se inicia a cero al comienzo del análisis y se incrementa con cada definición de señal (el incremento es posterior a la asignación del identificador, así que las señales se numeran desde cero).

- contador_unidades [UWORD]: análogo a contador_senales, pero dedicado a mantener la cuenta de las unidades definidas en el sistema.

También se usará en la generación de código (constante NUM_UNIDADES en _ctes.h).

- contador_vars [UWORD]: este contador se usa para asignar números a las variables dentro de un procedimiento. Se inicia a cero al inicio de la definición de una unidad. Esto motiva la restricción siguiente: no se puede definir una unidad anidada en otra y luego continuar con la declaración de variables de la segunda. Es decir, las instrucciones DCL deben preceder cualquier definición de unidad anidada. Ver sección de restricciones (apéndice A).

- num_max_estados [UWORD]: esta variable mantiene el número de estados (excluído el inicial) de la unidad que más tenga. Se usa en la generación de código para dar valor a la constante NUM_MAX_ESTADOS, definida en _ctes.h.

- num_max_campos [UWORD]: número de campos de la estructura que tenga más. Es el valor que, en la fase de síntesis, se dará a NUM_MAX_CAMPOS, constante definida en _ctes.h.

- num_max_vars [UWORD]: número de variables de la unidad que más tenga. Se usa en la generación de código para dar valor a la constante NUM_MAX_VARS, definida en _ctes.h.

- t_basicos [TIPO * [NUM_T_BASICOS]]: esta variable global es un array de punteros que designan los tipos básicos de SDL. Se usa para simplificar la asignación de tipos, por ejemplo en las estructuras EXPR.

- ..._dummy [varios tipos]: las variables ..._dummy (senal_dummy, tipo_dummy, var_dummy, expr_dummy y otras) representan valores sencillos de los tipos correspondientes. Estos valores se utilizan como 'relleno' cuando, debido seguramente a un error, no se puede usar un valor correcto. Por ejemplo, si en una expresión se referencia una variable que no existe, el traductor usará una variable de relleno (var_dummy) para poder continuar el análisis.

- lista_externas [EXTERNA *]: esta es la lista encadenada que representa las variables que han sido declaradas como REVEALED o EXPORTED. También contiene las referencias que se han hecho con VIEW e IMPORT. Las funciones del módulo VARS.C se ocupan de su manipulación.

- debugflag [BYTE]: indicador de si se está en modo depuración. Este modo genera algunas salidas que permiten trazar la ejecución del traductor.

- deb [FILE *]: fichero al cual se deben enviar los mensajes de

depuración.

- memflag [BYTE]: indica si el modo de depuración de memoria está activo.

- mem [FILE *]: fichero de trazo de memoria.

- errf [FILE *]: fichero de salida de los mensajes de error.

- errc [UWORD]: número de errores hallados. Se inicia a cero y se incrementa con cada salida de error.

6.3.3.- Los procedimientos.

- **Análisis léxico.**

Durante el análisis lexicográfico sólo hay tres funciones que se deben tratar:

- yylex(): es la función generada por lex a partir del fichero de descripción de patrones. Devuelve un valor entero que indica el símbolo terminal que se ha leído desde el texto fuente. Esta función contiene las acciones semánticas que se han especificado para cada patrón.

- TipoId(): cuando la función de análisis léxico encuentra un nombre SDL (es decir, una serie de caracteres alfanuméricos y/o especiales) debe decidir si ese nombre corresponde a una palabra reservada o es un nombre realmente. La función TipoId() se ocupa de efectuar esta comprobación, buscando el nombre hallado en una lista de palabras reservadas. Si se encuentra en la lista, se retorna el código de token correspondiente. Si no, se retorna el token NOMBRE.

- Compara(): para acelerar la búsqueda que la función TipoId() debe efectuar en la lista de palabras reservadas se usa un algoritmo dicotómico. Este necesita una función que compare dos elementos, y para eso sirve Compara().

- **Análisis sintáctico y semántico.**

En este epígrafe describiremos los métodos que se utilizan para gestionar algunas de las entidades más importantes que formarán la representación del sistema SDL.

- **Gestión de las jerarquías.**

Como se ha dicho anteriormente, el sistema SDL se representa internamente mediante un árbol de estructuras que mantiene la jerarquía de bloques y unidades definida por dicho sistema. El traductor siempre tiene conocimiento de cuál es la jerarquía que se está analizando (jer_actual). Cada vez que se encuentra una construcción que implique una modificación de la estructura jerárquica (una definición de o referencia a bloque, proceso o procedimiento), se llama a la función InstalaEnJerarquia(). Esta función acepta tres parámetros: el identificador de la jerarquía, el tipo de la misma (UDR_...) y el tipo de acceso: definición (EST_DEF) o referencia (EST_REF). Devuelve un puntero a la estructura JER de la que se trata. Nótese que una llamada a InstalaEnJerarquia() no necesariamente tiene que crear una estructura JER. Puede que la jerarquía ya hubiese sido creada e instalada en una llamada anterior, y en esta llamada sólo haga falta modificar algunos de sus campos. Un ejemplo típico de esto sería una primera llamada a InstalaEnJerarquía para referenciar el elemento (llamada que crearía la estructura JER y la insertaría en su lugar adecuado) y una segunda llamada cuando se encontrase la definición.

La lógica de la función InstalaEnJerarquia() es la siguiente:

- si se trata de una referencia a jerarquía, se comprueba que no esté duplicada, se crea una estructura JER y se añade a la lista correspondiente de jer_actual (a la lista de unidades si la jerarquía que se está instalando es un procedimiento o un proceso, o a la lista de bloques si se trata de un bloque).

- en caso contrario (es una definición), pueden darse dos casos:

  - que la definición no se encuentre enmarcada en ninguna otra jerarquía (es el caso de las definiciones remotas, que se encuentran después del ENDSYSTEM). En este caso la jerarquía debe haber sido referenciada obligatoriamente. Se buscará la estructura JER correspondiente y se cambiará su estado.

  - que la definición esté enmarcada en otra. En este caso sólo hay que decidir cuál es el padre de la jerarquía definida (si es que está cualificada) y añadirla a la lista correspondiente del mismo (o cambiar su estado, si ya había sido referenciada).

La función está muy autodocumentada, así que se recomienda complementar esta explicación con un repaso al código.

Aparte de la función InstalaEnJerarquia() hay otras que trabajan con estructuras de tipo JER: las que buscan una jerarquía determinada por un identificador (IdAJer()), las que manejan la lista de referencias (que se usa para acelerar la instalación de las jerarquías en ciertos casos especiales), las que manejan la pila de jerarquías, etc.

- **Gestión de las expresiones.**

Las estructuras de representación deben almacenar no sólo la disposición estructural del sistema (jerarquía), sino también las instrucciones de las unidades. Y estas instrucciones muy a menudo implican la evaluación, asignación, paso... uso, en general, de expresiones. Por lo tanto hay que proporcionar una forma de almacenamiento y representación de expresiones. La implementación elegida es la más directa: una expresión se representa mediante una estructura de tipo EXPR. Esta estructura consta, como muchas otras del sistema, de un tipo y una unión de datos. El tipo de expresión es una constante simbólica de la forma EXP_... y de su valor se infiere qué campo de la unión de datos es significativo. Así, por ejemplo, la constante EXP_VBOOL indica que la expresión es un valor de tipo booleano: el designado por el miembro <vboolean> de la unión. La constante EXP_OPI indica que la expresión es una aplicación de operador infijo. El miembro <operador> es una estructura que proporciona el tipo de operador infijo (como una constante simbólica de tipo OPI_...) y las expresiones que constituyen los operandos.

Como se ve, la estructura EXPR es recursiva: las expresiones de tipo 'aplicación de operador', 'if-then-else-endif', 'índice de array'... apuntan a una o varias expresiones que completan la descripción. De hecho, una expresión se puede considerar como un árbol y así es como se representa en esta implementación.

Hay una serie de funciones para crear, modificar y borrar las estructuras formadas para describir expresiones. La mayoría de estas funciones son llamadas desde el analizador sintáctico (desde el interior de las acciones semánticas). Algunas de ellas son llamadas una sóla vez. Eso no justificaría el hecho de su existencia como funciones independientes: su código podría sustituírse en el lugar de la llamada. Sin embargo, hay varias razones para aislar el código y convertirlo en una función: facilita el mantenimiento, no produce una sobrecarga apreciable en el tiempo de ejecución y, principalmente, mantiene relativamente limpio de código el fichero de descripción de la gramática (el fichero parser.y, que es la entrada a yacc). Este fichero es la columna vertebral del traductor (al menos de la fase de análisis) y debe estar codificado con la máxima claridad.

Trataremos a continuación brevemente algunas de las funciones que se han implementado para gestionar las expresiones:

- OperadorInfijo(): acepta el tipo de operador y las dos expresiones que constituyen los operandos y, tras una serie de chequeos semánticos, retorna una estructura EXPR que representa la aplicación de ese operador.

- TrataNombreEnExpr(): acepta una cadena de caracteres que se supone que es un literal, una constante o una variable. Trata de descubrir de qué tipo es el elemento (entero, real, booleano, pid...) y genera una estructura EXPR representativa, que retorna.

- Array(): se le suministran una expresión que se supone que es el array que se quiere referenciar, y una lista de expresiones que son los índices sobre ese array. La implementación actual sólo permite un índice. Se efectúan comprobaciones semánticas y se genera una estructura EXPR de tipo EXP_IND_ARRAY, que será retornada.

- Funcion(): se ocupa de tratar las aplicaciones de funciones. Como parámetros necesita la función a aplicar (constante simbólica de tip FUN_...) y una lista de expresiones que constituyen los argumentos de la función. Los chequeos semánticos incluyen comprobación de número de argumentos y de tipos de los mismos. Como valor de retorno se genera una EXPR de tipo EXP_FUNCION.

- **Gestión de los tipos, constantes, variables y señales.**

El SDL, un tipo de datos es visible sólo en la jerarquía en que se definió y en las inferiores. Los tipos básicos de datos se consideran definidos en la jerarquía de nivel máximo (el sistema). La implementación de esta característica no ofrece dificultades: cada estructura JER tiene entre sus campos una lista de estructuras TIPO, que representa los tipos que han sido definidos en esa jerarquía. La búsqueda de un tipo se realiza partiendo de la jerarquía en la que se haya encontrado la referencia a dicho tipo. Si no se encuentra en ella, se busca en la jerarquía superior (apuntada por el campo <padre>) y se sigue la línea de ascendencia hasta que se llega a la jerarquía sistema_SDL (la de nivel superior). Si el tipo no está en esta última, es que no se ha definido o no es visible. Se genera el error correspondiente.

Como se ha visto en la sección que estudia las estructuras de datos, una estructura TIPO es enlazable y consta de un nombre, un tipo (valga la redundancia), una unión de datos y el puntero de enlace. Basándose en esta estructura simple, la definición de tipos sinónimos (SYNTYPEs) es también

sencilla: basta con duplicar la estructura, cambiando únicamente el nombre. La implementación de la construcción CONSTANTS aún no se ha llevado a cabo, pero está diseñada sobre el papel: se ampliará la estructura TIPO para que contenga una condición de rango (tipo COND_RANGO). Durante la fase de síntesis se generará una nueva función ChequeaTipoXXXX() (donde XXXX es el nombre del tipo) por cada sinónimo con rango restringido. Estas funciones aceptarán un argumento que será un VALOR (ver discusión sobre expresiones en el capítulo dedicado al esquema de traducción) del tipo a comprobar y retornarán un valor booleano indicando si el tipo está dentro de los límites o no. Las funciones ChequeaTipoXXXX() se llamarán únicamente desde la función Asigna() para comprobar la validez del segundo argumento. Para ello será necesario modificar la función Asigna() para que acepte un nuevo parámetro que le indique qué función de chequeo de tipo debe usar (o que sea NULL si no se debe usar ninguna). Debe existir la posibilidad de desactivar la generación de comprobaciones.

Entre las principales funciones de manejo de tipos se encuentran BuscaTipo(), TEq() e InstanciaGenerador(). La primera acepta un identificador y retorna el tipo correspondiente, efectuando una búsqueda como la descrita más arriba. La segunda implementa la operación de equivalencia de tipos. A pesar de que SDL determina equivalencia (o compatibilidad) nominal, aquí se implementa la estructural. La función InstanciaGenerador() acepta un nombre de generador predefinido (ARRAY, POWERSET o STRING) y una lista de argumentos. Tras hacer una serie de comprobaciones semánticas, construye una estructura TIPO que representa la instanciación del generador, y la retorna.

Los tipos enumerados (LITERALS) se tratan exactamente como si fueran enteros, y sus constantes se introducen en las tablas correspondientes.

Las constantes son otro aspecto de SDL fácil de implementar: una constante, al igual que un tipo, es sólo visible en la jerarquía en que se definió o en las inferiores. Las estructuras JER también tienen listas de estructuras CTE para almacenar el conjunto de valores que se han definido con SYNONYM o con LITERALS. Hay una serie de constantes simbólicas predefinidas, que representan los códigos ASCII menores que 32 (ESC, CR, LF...). Estas constantes son incluídas en la lista de la jerarquía principal durante la iniciación del sistema.

Las variables sólo pueden existir en las jerarquías de tipo unidad (esto es, procedimientos o procesos). Por ello, la lista de estructuras de tipo VAR que mantiene las variables definidas en una unidad no forma parte de la estructura JER, sino de UNIDAD. Las estructuras VAR mantienen, entre otros, campos para indicar el tipo de declaración (normal, revelada o exportada), el

nombre de la variable, su tipo...

Dos puntos importantes han de ser tratados cuando se habla de las variables: la herencia y las exportaciones y vistas.

Una variable es sólo visible en la unidad que la creó y el los procedimientos que sean definidos dentro de ésta. Por lo tanto, un procedimiento hereda (tiene acceso a) todas las variables de su unidad llamadora. La función BuscaVar(), cuyo cometido es encontrar una variable dado un identificador, tiene en cuenta esta regla y busca ascendentemente a través del árbol de jerarquía, hasta llegar a una jerarquía que no sea una unidad.

Por otra parte, una variable puede ser declarada como REVEALED (revelada) o EXPORTED (exportada). El estándar SDL especifica que una variable revelada puede ser vista por otra unidad. Esta segunda unidad debe declarar la variable como visible utilizando la construcción VIEWED. A partir de entonces puede acceder al valor que en ese momento tenga la variable a través de la función VIEW. La exportación de variables funciona de manera algo diferente: la unidad exportadora debe declarar la variable como EXPORTED y, despúes, ejecutar la instrucción EXPORT. La unidad importadora declarará la variable importada (IMPORTED) y accederá al valor exportado mediante la función IMPORT. El valor exportado de la variable permanecerá indefinido mientras no se ejecute la instrucción EXPORT. Para aclararlo todo: el acceso a una variable revelada proporciona el valor actual de dicha variable; el acceso a una variable exportada proporciona el último valor exportado de la variable (puede que su valor actual sea distinto).

La implementación de las variables reveladas y exportadas es única: no se hablará en adelante de variables exportadas o reveladas, sino de variables externas.

Se plantea el problema de las referencias adelantadas: durante el análisis puede encontrarse una unidad que importe o vea una variable cuya unidad exportadora o reveladora no haya sido tratada aún. Para solucionar esta dificultad se mantiene una lista de variables externas (estructuras EXTERNA). Los elementos de esta lista pueden ser referencias a variables externas o punteros ya resueltos. Así, una declaración DCL-EXPORTED o DCL-REVEALED buscará en la lista. Si encuentra una referencia, la resolverá. Si no la encuentra, añadirá un nuevo nodo a la lista. Este nodo será una externa resuelta (se conocerá ya la ubicación de la variable). Una declaración IMPORTED o VIEWED o una llamada a las funciones IMPORT o VIEW motivará una busqueda en la lista de externas. Si se encuentra un nodo coincidente, se apuntará a él desde la estructura correspondiente (esté o no resuelto). Si no se encuentra, se añadirá uno nuevo: una referencia sin resolver.

La gestión de las señales tiene pocos puntos interesantes para recalcar: las señales (incluídos los temporizadores) se almacenan el las listas de estructuras SENAL que forman parte de las estructuras JER. Las listas de señales son abreviaciones que agrupan una serie de nombres de señales bajo un nombre común (el de la lista). Se mantienen mediante estructuras LISTA_SENALES, enlazadas en listas dependientes de JER. Los chequeos semánticos relativos a las señales suelen ser las comprobaciones de coincidencia de los tipos de argumentos (en OUTPUT, INPUT, ACTIVE, SET...).

- **Gestión de las vías de señales.**

Como se dijo en el apartado dedicado a la terminología, se han agrupado los canales y las rutas de señales bajo la denominación común de vías de señales. Estas se representan mediante estructuras VIA y se agrupan en listas enlazadas que pertenecen a las estructuras JER. Las nuevas vías se crean mediante las funciones NuevaRuta() y NuevoCanal(). La única función que modificará esas vías creadas será EstableceConexion(). Las demás funciones relativas a las vías de señales se limitan a consultarlas para calcular, por ejemplo, la unidad de destino de una salida.

En una primera implementación, las vías de señales bidireccionales se representaban mediante dos elementos en la lista de vías: la vía directa y la vía inversa (cada una con su conjunto de señales). Cuando llegó el momento de implementar las funciones de búsqueda de destinos de salidas y de conexión de vías se pensó que esto podía dificultar ciertas operaciones y se modificó la implementación, haciendo que una estructura VIA representase tanto la vía directa como la inversa (había, pues, dos conjuntos de señales transportadas). Tras una serie de intentos fallidos de codificar las funciones de cálculo de destinos y conexión se descubrió que en realidad la primera solución era la correcta, pero necesitaba una modificación. Se retornó a la primera implementación (VIAs separadas) añadiendo un nuevo campo a la estructura VIA: el elemento <via_inversa> apunta a la estructura VIA que representa la dirección contraria a la actual. Si la vía es unidireccional, no habrá inversa y el campo valdrá NULL. Esta sencilla modificación simplifica en gran medida las búsquedas en el cálculo de destinos y en las conexiones. Inicialmente la búsqueda debía hacerse por toda la lista de vías definidas, ya que no se sabía si la vía era uni- o bidireccional. Ahora, cuando se encuentra la primera VIA, no es necesario seguir buscando la segunda (si es que existe). Hay otra serie de oscuros motivos que ayudaron a tomar esta decisión de implementación.

Las mencionadas funciones de cálculo de destinos y de conexión sirven respectivamente para hallar la unidad destino de un OUTPUT

(posiblemente con modificador VIA) y para modificar las vías de señales de modo que reflejen las conexiones establecidas por una construcción CONNECT. Las funciones que ejecutan estas tareas están bastante autodocumentadas, y se remite a ellas para más información.

6.4.- La fase de síntesis.

La función principal del programa LEDAC llama primero al analizador sintáctico generado por yacc, contenido en la función yyparse(), para que se encargue de efectuar el análisis léxico, sintáctico y semántico del fichero de entrada. Una vez terminada esta primera fase, se tiene en memoria la información necesaria para generar el sistema equivalente. De esto se ocupa la fase de síntesis.

6.4.1.- Las estructuras.

Durante la fase de síntesis se van a usar casi todas las estructuras de representación y auxiliares, que serán consultadas para generar el código. Sólo se usará una nueva estructura que no se empleó en la fase de análisis.

- struct tdu_str:

Esta estructura almacena información sobre una unidad. Se emplea en forma de vector para agrupar los datos de las unidades que se van generando. Cuando se hayan generado todas las unidades, se construirá la declaración y asignación de la variable <tdu> con los datos almacenados en estas estructuras. La variable <tdu> es la tabla de descripción de unidades, una parte fundamental del sistema equivalente (véase la discusión sobre el sistema equivalente).

6.4.2.- Las variables globales.

- dir_salida [char [MAX_NOMFICH]]: nombre del directorio en el que se pondrán los ficheros generados. Por defecto es '.\eqv', pero puede cambiarse con la opción -G.

- n_XXXX [char [MAX_NOMFICH]]: donde XXXX es el nombre de cada fichero generado (sustituyendo el punto por '_'). Son los nombres completos (incluído directorio) de los ficheros generados.

- XXXX [FILE *]: son los ficheros donde se generará el código.

- nom_sen [char **]: con calloc() se convertirá en un vector de

cadenas, con tantos elementos como señales hay en el sistema, que contendrá los nombres de las señales definidas.

- nom_est [char **]: será un vector de <n> cadenas (siendo n= número de unidades en el sistema * número máximo de estados) que almacenará, para cada unidad, el nombre de sus estados.

- nom_var [VAR **]: será un vector de tantos punteros a VAR como unidades haya en el sistema. Cada puntero a VAR apuntará a la lista de variables de la unidad correspondiente.

- tdc [UWORD *]: la tabla de comportamiento. Se considerará una matriz tridimensional que a cada terna (unidad, señal, estado) le hará corresponder un UWORD que será el número de transición a ejecutar, TCDS si hay que salvar la señal o TDCI si la combinación es incorrecta.

- tdu [struct tdu_str *]: será un vector de tantos elementos como unidades se hayan definido en el sistema. Cada elemento será una estructura de tipo tdu_str, que contendrá información sobre la unidad correspondiente. Se usará para generar la tabla de descripción de unidades (en el fichero _tdu.c).

- contador_et_dec [UWORD]: mantendrá la cuenta de las etiquetas de decisión para evitar repeticiones. Podría reiniciarse en cada unidad, pero se ha preferido no hacerlo así.

- cont_transiciones [UWORD]: se reinicia a 1 al comienzo de la generación de una unidad. Se incrementa cada vez que se genera una transición. Este es el valor que, en la tabla de comportamiento, define la transición que se ha de ejecutar.

- punto_de_entrada [UWORD]: se inicia a cero al comenzar a generar una unidad. Se usa para mantener la cuenta de los puntos de entrada de procedimientos.

- gnivel [UWORD]: indica a la función de generación de código gprintf() cuántos espacios debe indentar antes de escribir sus argumentos.

6.4.3.- Los procedimientos.

El procedimiento de nivel superior en la generación de código es la función GeneraSistema(). Esta se encarga de iniciar variables, abrir los ficheros y generar la jerarquía principal (sistema_SDL). La función GeneraJer() acepta una jerarquía genera el código necesario para

implementarla. Puede que incluso se llame a sí misma de forma recursiva durante el proceso. Si la jerarquía pasada a GeneraJer() es una unidad, ésta será tratada por la función GeneraUnidad(), que se encargará de construir la función equivalente C, generando las transiciones espontáneas, las señales contínuas y las transiciones normales. En esta tarea usará otras dos funciones: GeneraInstrucciones() y GeneraTransicion(). La primera acepta una lista de instrucciones y escribe en el fichero _uniddes.c el código equivalente. La segunda función también generará instrucciones, pero antes modificará la tabla de comportamiento para que refleje la necesidad de ejecutar esas instrucciones cuando, en un estado determinado, se reciba una señal determinada.

Otras funciones de generación de código son GeneraExpr(), que se encarga de transformar las estructuras EXPR en cadenas de llamadas a funciones C; GeneraDec(), que sirve para crear código que maneje decisiones; GeneraVar(), que comprueba si la variable a utilizar es local o heredada y genera la función equivalente C correspondiente; GeneraOutput(), que calcula los destinos de las señales y crea el código C para enviarlas.

6.5.- Línea de comandos.

La llamada al traductor LEDAC se hace mediante una línea de comandos cuyo formato es el siguiente:

```
LEDAC [<opciones>] [fich.sdl]
```

Si no se especifica un nombre de fichero SDL a traducir, se tomará como fichero la entrada estándar.

Las opciones son letras precedidas por guiones ('-'). El tipo es significativo: no son iguales las mayúsculas que las minúsculas. Algunas opciones deben ir seguidas por un argumento, que puede o no estar separado de la letra por espacios. Se pueden agrupar varias letras de opción tras el guión (por, ejemplo, -dcm).

Las opciones disponibles son:

-d: se generará un fichero llamado debug.deb y en él se volcarán mensajes de depuración.

-D <fdeb>: se generará una salida de depuración, pero esta vez en el fichero <fdeb>.

-c: se producirá una salida de depuración de conexiones a debug.deb.

-C <fdeb>: la salida de depuración de conexiones irá a <fdeb>.

-m: salida de trazo de memoria a memoria.mem.

-M <fmem>: salida de trazo de memoria a <fmem>.

-E <ferr>: los mensajes de error se enviarán al fichero <ferr> (por defecto, se envían a stdout).

-a: se redirigirán las salidas de depuración, depuración de conexiones, trazo de memoria y error a stdout.

-l: Se generará un listado del sistema para referencia y comprobaciones.

-n: no se generará código.

-G <dir>: se pondrán los ficheros generados en el directorio <dir>. Por defecto se ponen en el directorio '.\GEN'.

-h: se listará una referencia de las opciones y el programa terminará.

7.- El sistema equivalente (EQV).

7.1.- El esquema de traducción.

7.1.1.- Introduccion.

Como se ha visto, un sistema SDL es un conjunto de procesos en ejecución que se comunican mediante señales. Esta estructura ha motivado la consideración de una serie de ideas para la implementación del sistema equivalente C (es decir, métodos para implementar la semántica de SDL en C).

Una primera idea consistía en traducir cada proceso a una función C que se ejecutaría de forma concurrente a las demás funciones (procesos) del sistema. Las vías de comunicación se implementarían mediante ficheros compartidos, sockets, paso de mensajes, zonas de memoria compartida u otros sistemas, generalmente implicando el uso de semáforos.

Esta solución resolvía de forma sencilla el problema de la ejecución, ya que cada función sería autónoma y comprobaría su propia cola de señales para decidir su evolución. Sin embargo, se aprecian algunas desventajas:

- la transmisión de señales parece complicada y, en todo caso, es bastante dependiente del sistema operativo empleado (los sockets, los semáforos, los mensajes, etcétera, se implementarían en UNIX).

- la multiprogramación también requeriría primitivas muy específicas del sistema operativo (por ejemplo, fork() en UNIX) e impediría la simulación en plataformas DOS.

- la concurrencia haría difícil la implementación de un entorno de depuración y trazado de ejecución.

La segunda solución que se ideó no usaba multiprogramación, sino la ejecución secuencial de una transición de cada unidad en ejecución. Las transiciones (aquí se usa el término en el sentido estándar de SDL, es decir, representando un conjunto de instrucciones) se asociarían a funciones C. Una tabla global indicaría la función que se debería ejecutar dependiendo de la situación de la unidad.

Esta segunda implementación solucionaría gran parte de los problemas de la primera: es bastante independiente de la plataforma y, por tanto,

portable. Sin embargo, algo que puede parecer poco importante motivó el abandono de esta idea: la dificultad de implementar saltos a etiquetas (JOIN) era grande. En SDL, se puede saltar mediante la instrucción JOIN a una etiqueta colocada en cualquier lugar dentro de la unidad (proceso o procedimiento) donde está la instrucción de salto. Si la unidad está desperdigada por una serie de funciones C, la cosa se complicaría en gran medida. Se pensó incluso en usar saltos no locales, usando las funciones C longjmp() y setjmp(), pero se desechó la idea tan pronto como apareció.

Una tercera idea apareció entonces: se podían concentrar todas las funciones asociadas a transiciones para formar una única función que estaría asociada a una unidad. Esta función equivalente aceptaría una serie de parámetros que le indicaran qué transición debía ser ejecutada (o si se debía ejecutar una señal contínua o una transición espontánea).

Las instrucciones SDL se implementarían mediante funciones C equivalentes que modificarían las estructuras del sistema equivalente para producir los efectos deseados.

Una tabla global (la tabla de descripción de unidades o tdu) mantendría los datos de cada una de las unidades definidas en el sistema, entre los cuales se encontraría un puntero a la función equivalente. Otra estructura global, esta vez una lista encadenada (la tabla de unidades o tu), se encargaría de almacenar información sobre cada una de las instancias de unidad activas en el sistema. Entre otros datos, cada entrada contendría el tipo de unidad (que, indexando sobre la tabla de descripción de unidades, daría la función equivalente y el resto de los datos), el estado actual de la unidad, las variables y la cola de señales.

Hay que distinguir entre tipos de unidad e instancias de unidades: puede haber varias instancias de un tipo de unidad ejecutándose en el sistema, cada una de ellas con su estado, variables y cola de señales propios (almacenadas en la tabla de unidades), pero con otros datos comunes (almacenados en la tabla de descripción de unidades), como el número de variables, el número máximo permitido de instancias y, por supuesto, la función equivalente.

Las instancias de una unidad compartirían, pues, las estructuras que definen su comportamiento: la funcion equivalente y las entradas de la tabla de comportamiento.

La tabla de comportamiento o tdc es otra estructura global (una matriz de dimensión 3) que a cada terna (unidad, estado, señal) le hace corresponder un valor que indica la acción que se debe ejecutar si la unidad especificada está en el estado especificado y tiene como primera señal de la

cola la especificada. La acción puede ser el número de una transición que hay que ejecutar (mediante la función equivalente), un indicador de que hay que salvar la señal, u otro indicador de que la situación es incorrecta.

Tras esta introducción general, pasaremos a ver en más detalle (aún a riesgo de ser redundantes) los diferentes aspectos de la implementación en C de un sistema SDL.

7.1.2.- Funciones asociadas.

En la traducción a C se considerará cada proceso o procedimiento SDL (es decir, cada unidad) como una función C, llamada función asociada. Además de cada proceso y procedimiento, se considera un seudoproceso ENV que modela el entorno. Las funciones asociadas devuelven un valor de tipo UWORD y aceptan
dos argumentos: el número de la transición que deben ejecutar y la señal que ha motivado esa decisión (si la hay).

El valor de retorno será la constante RET_UNIDAD_OK si se ejecutó sin problemas la transición pedida. Se retornará la constante RET_SENAL_NO_CONSUMIDA si no se pudo ejecutar la transición (porque no se cumplió la condición de activación, por ejemplo).

Una de estas funciones consistirá en un switch que, dependiendo del primer parámetro, ejecutará una de las transiciones de la unidad leyendo los argumentos de la señal (si ésta los tiene). El primer parámetro, como se ha dicho, indicará la transición que se debe ejecutar. Pero también podrá ser TRANS_ESPONTANEA o SENAL_CONTINUA, en cuyo caso se ejecutará una transición espontánea elegida aleatoriamente o la señal contínua que tenga mayor prioridad y cumpla la condición de activación. Por supuesto, sólo se considerarán para su ejecución las transiciones espontáneas y señales contínuas que estén definidas en el estado actual.

Los nombres de las funciones asociadas serán de la forma Unidad_xxxx, siendo xxxx el identificador numérico de la unidad.

7.1.3.- La ejecución.

En la función principal se inician los procesos del sistema y, dependiendo de si se ha especificado el modo de depuración, se ejecutan las unidades del sistema de forma contínua y autónoma o de forma controlada. La función principal relacionada con la ejecución de las unidades es EjecutaUee(). Esta función comprueba cuál es la acción que debe realizar la

unidad en ejecución (uee), dependiendo de su estado y de las señales en la cola de entrada, la realiza y calcula la nueva unidad en ejecución. La función EjecutaUee() realizará las siguientes tareas:

- Comprobará la lista de temporizadores para ver si hay que disparar alguno.

- Si la unidad a ejecutar es nueva (primera_ejecucion==TRUE), ejecutará la función de iniciación de variables asociada (InicVarsXXXX).

- Actuará de acuerdo con la situación de la unidad a ejecutar:

    - Si está en situación SIT_MUERTA:

        - Si es un procedimiento despertará al padre. En cualquier caso eliminará la unidad de la tabla de unidades y comprobará los límites.

        - Calculará la nueva unidad en ejecución y retornará.

    - Si es un proceso y está esperando a que acabe un procedimiento (SIT_ESPERANDO_FIN_PTO) la ignorará y pasará a la siguiente unidad de la tabla de unidades, haciéndola la nueva unidad en ejecución.

    - Si la unidad está activa (SIT_ACTIVA):

        - Si la unidad está en la transición START (transición 0), la ejecutará.

        - Si la unidad tiene transiciones espontáneas y se decide ejecutar una de ellas, se llamará a la función asociada a la unidad con los parámetros TRANS_ESPONTANEA y NULL (ninguna señal).

        - Si no (no tiene transiciones espontáneas o no queremos ejecutar una) se irán tomando señales de la cola y, dependiendo de la respuesta que la uee de en el estado actual a esa señal, (respuesta que se encuentra almacenada en la tabla de comportamiento) haremos lo siguiente:

            - Si la respuesta es TDC_SALVAR_SENAL pasaremos a la siguiente señal de la cola y reiniciaremos el bucle.

            - Si la respuesta es TDC_INCORRECTO es que esta unidad no puede recibir esa señal en este estado. Se generará un error.

- Si no, ejecutaremos la función asociada a la unidad pasándole como parámetros la respuesta (que es el número de la transición que debe ejecutarse) y la señal. A continuación saldremos del bucle de toma de señales de la cola.

Este bucle se ejecutará a lo largo de la cola de señales hasta que se encuentre una que no sea salvable o hasta que se acabe la cola. Si el bucle acaba por esta segunda razón, se debe ejecutar una señal contínua. Para ello se llamará a la función asociada con los argumentos SENAL_CONTINUA y NULL. Si no se han especificado señales contínuas para la unidad la función asociada simplemente ignorará la petición.

- Calculará la siguiente unidad en la tabla de procesos.

7.1.4.- Los canales, las rutas de señales, las listas de señales...

No necesitan ser traducidas a C. Sólo proporcionan información semántica sobre el sistema, y esta ya queda reflejada en la tabla de comportamiento. Permiten, además, saber los destinos en los OUTPUT VIA.

7.1.5.- Expresiones.

Todas las expresiones manejarán estructuras de tipo VALOR. Esta estructura consta de un entero que indica el tipo del valor y de una unión que contiene todos los tipos utilizados en el sistema.

Se usarán solo los tipos predefinidos en SDL (Boolean, Character, Charstring, Integer, Natural, Real, Pid, Duration y Time) con sus operadores predefinidos, los generadores String, Array y PowerSet con sus operadores y las estructuras.

En una primera implementación las estructuras de tipo VALOR que se creaban no se destruían cuando dejaban de ser necesarias, excepción hecha de las listas de variables y de argumentos de señales. Así, por ejemplo, una simple construcción de señal contínua con la cláusula PROVIDED TRUE generaba un nuevo VALOR (de tipo booleano y dato TRUE) con cada evaluación. Estos valores no eran liberados y su acumulación terminaba agotando la memoria. Se ha introducido una mejora que consiste en considerar VALORes de dos tipos: estáticos y volátiles. El tipo de un VALOR queda reflejado por el estado de su indicador <estatico>. En general, un

VALOR estático es aquel que debe ser eliminado explícitamente, y uno volátil es el que puede ser liberado de forma implícita tras ser usado en una función. Ejemplos de VALORes estáticos son las variables, los argumentos de las señales, los elementos de los arrays, cadenas, conjuntos y estructuras... VALORes volátiles son las constantes y los VALORes intermedios que aparecen en la evaluación de una expresión.

Decidida esta nueva implementación, se modificaron las funciones que aceptan argumentos de tipo VALOR para que, antes de retornar, los liberen si éstos son volátiles. Harán esto llamando a la función LiquidaValor() para cada uno de los VALORes que deseen liberar. La función LiquidaValor() acepta un VALOR como argumento y lo libera sólo si es volátil. Como ejemplo, si a la función Suma() (que acepta dos VALORes de tipos compatibles retorna un nuevo VALOR que contiene la suma de ambos) se le pasan como argumentos los VALORes Variable(1) y ValorEntero(32), la función tratará de liberarlos al finalizar. El primer VALOR será de tipo estático (la función Variable() se habrá encargado de activar el indicador) y, por tanto, no será liberado. El segundo VALOR, en cambio, es volátil y será eliminado.

Hubo, sin embargo, que hacer otra modificación a las funciones que aceptan estructuras VALOR. A veces alguna de estas funciones (sobre todo CCierto(), Asigna() e Igual()) es llamada desde el interior de otra función, y no desde código generado por LEDAC. En algunos de estos casos, el VALOR pasado a la función puede ser volátil, pero no debe ser eliminado, porque se va a seguir usando localmente. La solución a este problema ha sido la adición de un nuevo argumento a todas las funciones que aceptan parámetros VALOR. Este argumento (<t>) define el tipo de llamada que se realiza. Si es igual a la constante simbólica EXTERNA, se llamará a la función LiquidaValor() al finalizar. Si, en cambio, <t> es INTERNA, no se tratará de liberar los VALORes.

Para mantener la simplicidad del código generado, se cambiaron los nombres de todas las funciones que aceptaban el nuevo parámetro de tipo de llamada, anteponiéndoles un subrayado ('_') y se #definieron macros con los nombres originales, cuyo cometido es llamar a las funciones con el argumento EXTERNA. Así, por ejemplo, la función Suma(), cuyo prototipo era

```
VALOR *Suma( VALOR *, VALOR * )
```

y que, con el cambio, había pasado a ser

```
VALOR *Suma( VALOR *, VALOR *, BYTE )
```

ahora se convertía en

```
VALOR *_Suma( VALOR *, VALOR *, BYTE )
```

y se añadía al sistema la definición de macro

```
#define Suma(v1,v2) _Suma( (v1), (v2), EXTERNA )
```

con lo cual el código generado podía seguir siendo el mismo. Todas las llamadas internas a estas funciones modificadas deben hacerse de la forma

```
_función( args..., INTERNA )
```

Todos los VALORes serán creados (mediante la función NuevoValor()) con el indicador <estatico> desactivado, es decir, como volátiles. Algunas funciones que retornan VALORes activarán el indicador antes de devolverlo (Variable(), por ejemplo). Otras funciones (las de iniciación, como IniciaEstructura()) también activarán el indicador.

Una función llamada Asigna() se encargará de las asignaciones. Aceptará dos argumentos: el primero será un puntero a VALOR que le indicará dónde debe almacenarse el dato suministrado (también como puntero a VALOR) en el segundo. Las formas típicas del primer argumento serán

```
  Variable( int n )
la variable <n>-sima de la zona de variables de la uee.

  IndArray( VALOR *array, VALOR *indice )
el elemento <indice> del array <array>.

  Campo( VALOR *estr, UWORD campo )
el campo <campo> de la estructura <estr>.

  ArgSenal( SENAL *senal, UWORD num_arg )
el argumento <num_arg> de la señal <senal>.
```

Estas funciones podrán ser utilizadas como LValores o como RValores. En terminología de diseño de compiladores (heredada, a su vez, de C), un LValor es un elemento que puede colocarse en la parte izquierda de una asignación (es decir, un valor cuya dirección se conoce). Un LValor se puede también colocar en la parte derecha. Sin embargo, un RValor sólo se puede colocar a la derecha de una asignación. Ejemplos de LValores en C son

```
var, arr[ind][23], str.campo, *cosa (si cosa es, por ejemplo, un
```

```
char *)
```

y ejemplos de RValores son

```
5, &var, str.campo+34, *cosa-'z'
```

7.1.6.- Valores iniciales: el modo Geode.

Según el estándar SDL, el uso de una variable antes de su primera asignación es ilegal y debe generar un error dinámico. Sin embargo, al menos un programa comercial de simulación de sistemas SDL (Geode) considera que las variables no iniciadas tienen valores nulos, y permite usarlas.

Se implementará un modo de funcionamiento del sistema equivalente que permita usar variables no iniciadas, pero únicamente en comparaciones. Estas variables tendrán valores nulos cuando sean comparadas.

7.1.7.- Las transiciones.

Una unidad (es decir, un procedimiento o un proceso) constará de varias transiciones y ejecutará la que le indique la función EjecutaUee() mediante el primer argumento de la función C asociada.

Cada transición será un bloque asociado a una etiqueta case dentro de un switch.

7.1.8.- Señales contínuas y transiciones espontáneas.

Si la cola de señales está vacía la función EjecutaUee() llamará a la función asociada con el argumento SENAL_CONTINUA. Dentro del switch de dicha función, el bloque correspondiente a la etiqueta 'case SENAL_CONTINUA:' contendrá sólo una instrucción return(RET_UNIDAD_OK) en caso de que no haya señales contínuas especificadas para esa unidad. Si, por el contrario, las hay, estarán ordenadas de mayor a menor prioridad (valores numéricos crecientes).

En la función de ejecución también se decidirá si debe ejecutarse una transición espontánea. Para ello, se generará un número aleatorio entre cero y cien y, si es menor o igual que la probabilidad de que se dispare una transición espontánea (constante PROB_TRANS_ESP) se llamará a la función asociada con el argumento TRANS_ESPONTANEA. La función asociada tiene

en su switch la etiqueta 'case TRANS_ESPONTANEA:', en la que están todas las transiciones espontáneas de la unidad.

Las señales contínuas y las transiciones espontáneas están todas agrupadas en la misma etiqueta case, pero pueden pertenecer a estados diferentes. Por eso, cada transición está resguardada por un if que comprueba si el estado actual es el mismo en el que fue definida. Como la transición (o señal) puede pertenecer a más de un estado, se hace la comprobación mediante la función auxiliar EstadoActualEs(), que acepta un número variable de estados, terminados por S_NULL (el estado nulo) y devuelve TRUE si el estado actual de la unidad en ejecución (uee) es uno de los pasados, y FALSE si no lo es.

Pero, además de la comprobación de que se esté en el estado correcto, hay que comprobar que se cumpla la condición de activación (PROVIDED). Esta es obligatoria en las señales contínuas, y opcional en las transiciones espontáneas. La comprobación se hará a través de la función CCierto().

La función CCierto() (Cierto para C) extrae el valor de una expresión booleana. Acepta un argumento de tipo VALOR *, supone que es de tipo BOOLEAN y retorna un valor BYTE que es el campo booleano de la estructura VALOR.

Tenemos, por lo tanto, que una señal contínua se representará así:

```
if( EstadoActualEs( <estados...> ) )
if( CCierto( <expresión booleana> ) )
{
<transición correspondiente>
}
```

Y una transición espontánea generará el siguiente código:

```
if( EstadoActualEs( <estados...> ) )
[if( CCierto( <expresión booleana> ) )]
{
<transición correspondiente>
}
```

Como se ve, el chequeo de la expresión de activación es opcional.

Podrían comprobarse ambas condiciones (la del estado de definición y la de activación) en el mismo if. De hecho, así se hizo en una primera versión. Pero eso implica que posiblemente se evalúe la condición de

activación aunque no se esté en el estado de definición. El estándar ANSI C indica que el operador AND lógico (&&) garantiza la evaluación de izquierda a derecha con cortocircuito. Es decir, si tuviésemos

```
if(EstadoActualEs(<estados...>) &&
     CCierto(<expresión booleana>))
```

y la primera premisa (estado de definición) fuese falsa, no se evaluaría la condición de activación (que puede suponer cosas no ciertas en el estado actual).

Pero alguno de los compiladores que se usan para el desarrollo de este proyecto no son ANSI, y no es seguro que evalúen en cortocircuito, así que se ha elegido la forma que asegura la evaluación secuencial y condicional.

Se tendrán en cuenta las dos restricciones siguientes:

- Si hay más de una señal contínua en una unidad, se deben especificar prioridades.

- En una señal contínua, SENDER es SELF. Igualmente para una transición espontánea. Esto se implementa al inicio de cada función asociada: si la señal pasada es NULL, sender se hace self.

7.1.9.- Resguardo de señales (SAVE).

El hecho de que una señal <s> debe ser salvada en un estado <e> de la unidad <u> se representará haciendo que el valor de la tabla de comportamiento correspondiente a (<u>, <s>, <e>) sea TDC_SALVAR_SENAL. La función EjecutaUee() se encargará de tratar las señales que deban salvarse, como se explicó más arriba.

7.1.10.- Condiciones de activación (PROVIDED).

Si una transición tiene una condición de activación, ésta se comprobará al inicio del bloque case correspondiente, usando la función CCierto():

```
case <n>:
if( !CCierto( <expr. booleana> ) )
return( RET_UNIDAD_NO_CONSUMIDA );
<resto de la transición>
```

7.1.11.- Etiquetas.

Ya que una etiqueta debe ser local a una unidad, y cada unidad se traducirá a una función C, las etiquetas SDL equivaldrán a las etiquetas de C y JOIN a la instrucción goto.

El hecho de que las etiquetas SDL deban ser visibles en toda la unidad a la que pertenecen ha sido uno de los principales motivos de la implementación de las unidades como funciones asociadas. En principio se pensó en crear una función para cada transición, pero eso dificultaría enormemente los saltos.

7.1.12.- Decisiones.

- **Deterministas.**

Una construcción de la forma

```
  DECISION <expr1>;
(<expr2>): <trans1>                ; Rango abierto.
(<op.relacional><expr3>): <trans2> ; Rango abierto.
(<expr4>:<expr5>): <trans3>        ; Rango cerrado.
else: <trans4>
  ENDDECISION
```

se traducirá a:

```
{
VALOR *aux;

aux= <expr>;
if( CCierto( IgualGeneral( aux, <expr2> ) )
{
<trans1>
goto _EtDec_xxxx;
}

if( CCierto( <func.op.relacional>( aux, <expr3> ) ) )
{
<trans2>
```

```
goto _EtDec_xxxx;
}

if( CCierto(MayigGeneral( aux, <expr4> )) &&
CCierto(MenigGeneral( aux, <expr5> )) )
{
<trans3>
goto _EtDec_xxxx;
}

/* ELSE */
<trans4>

_EtDec_xxxx:;
}
...
```

En caso de que una de las condiciones sea una conjunción de rangos se usará el operador || de C.

Como se ve, desde la declaración de aux hasta la etiqueta de salida hay un bloque C. Esto permite anidar SELECTs y usar el mismo nombre de variable (aux), pero en distintas profundidades. Por supuesto, así tenemos variables distintas.

- **No deterministas.**

Una estructura de la forma

```
DECISION ANY;
(): <trans1>
(): <trans2>
...
...
(): <transj>
ENDDECISION;
```

se traducirá a

```
switch( GeneraUniforme( 1, j ) )
{
case 1:
<trans1>
```

```
break;

case 2:
<trans2>
break;

...
...

case j:
<transj>
break;
}
```

donde GeneraUniforme(a,b) es una función que genera números aleatorios que se ajustan a una distribución uniforme en el intervalo [a,b].

- **Opción de transición (TRANSITION OPTION).**

Esta construcción es básicamente igual a una DECISION (en lo que a la simulación concierne) y se implementará de forma idéntica.

7.1.13.- NEXTSTATE.

La función C NextState() se encargará de cambiar el estado actual de la unidad en ejecución (uee->estado_actual) al que se le indique.

7.1.14.- RETURN.

La función equivalente Return() debe ser ejecutada sólo desde un procedimiento. Pondrá su situación a SIT_MUERTO y retornará a la función EjecutaUee(). En la función equivalente, la llamada a Return() irá seguida por una instrucción return(RET_UNIDAD_OK).

7.1.15.- STOP.

Esta instrucción sólo se puede ejecutar desde un proceso. Su función equivalente C hará que uee->situación valga SIT_MUERTO y retornará a EjecutaUee(). Al igual que con la instrucción Return(), la función Stop() irá siempre acompañada por una instrucción return(RET_UNIDAD_OK).

7.1.16.- CREATE.

Se traducirá a la función C Create(). Esta toma como primer parámetro el identificador del tipo de proceso que se desea crear. A continuación se pasan una lista de cero o más expresiones (punteros a VALOR) que son los argumentos del proceso. Si un parámetro se omite, se indicará con NULL. La función Create() comprobará que no se ha superado el máximo número de instancias del proceso, creará una nueva entrada en la tabla de unidades, iniciará los valores de esa entrada (creando, entre otras cosas, la zona de variables), actualizará el valor OFFSPRING de la unidad en ejecución (si es que el proceso tiene padre; podría estar siendo creado por la iniciación del sistema) y, por último, copiará los argumentos que no sean NULL en las variables correspondientes.

7.1.17.- CALL.

Esta instrucción SDL se traducirá a la función Call(), que es prácticamente igual que Create(). Pero antes de llamar a la función se modificará el valor uee->punto_de_entrada. Ver 'Llamadas a procedimientos' para más información. Después de la instrucción Call() se insertarán una un instrucción return(RET_UNIDAD_OK) y una etiqueta de la forma _EtPE_xxxx, siendo xxxx un identificador numérico único dentro de la unidad. En el epígrafe 'Llamadas a procedimientos' se explicará el significado de esta etiqueta.

7.1.18.- Llamadas a procedimientos.

En las llamadas a procedimientos se plantea un problema. La unidad mínima de ejecución suele ser la transición. Pero si en medio de una transición se llama a un procedimiento, ésta debe quedar interrumpida. La unidad llamadora queda en estado SIT_ESPERANDO_FIN_PTO y se crea una nueva unidad correspondiente al procedimiento llamado. Cuando el procedimiento termina, la función C equivalente Return() despierta a su padre (su unidad llamadora) y ésta debe continuar la transición en la que estaba cuando se ejecutó el CALL. Esto implica la posible entrada desde la función de ejecución en la mitad de una transición. Pero no hay modo de indicar a la función equivalente que no debe comenzar la ejecución de la transición desde el principio, sino desde el punto donde se quedó al ejecutar la instrucción Call().

La solución que se ha implementado es la siguiente: cada entrada de la tabla de unidades tiene un campo llamado punto_de_entrada. Este campo

toma como valor inicial 0 (en Create o Call). En caso de que se ejecute un procedimiento, antes de llamar a la función Call() se asignará a este campo un número determinado:

```
uee->punto_de_entrada= <x>;
```

Todas las unidades que contengan llamadas a procedimientos incluirán un código que en cada llamada a la unidad comprobará el valor de punto_de_entrada. Si es 0 se ejecutará la transición correspondiente de forma normal (es decir, desde el principio). Si no es 0, se hará 0 y se saltará a una etiqueta colocada después de la llamada Call() que motivó el corte de la transición. Esta es la etiqueta de que se habló con anterioridad.

Por tanto, la instrucción SDL CALL se traducirá a la función equivalente Call(), una instrucción return(RET_UNIDAD_OK) (para abandonar la transición) y una etiqueta.

El siguiente es un ejemplo con una unidad que tiene dos llamadas a procedimiento:

```
UWORD Unidad_xxxx( UWORD transicion, SENAL *s )
{
...
...

switch( uee->punto_de_entrada )
{
case 0:
break;

case 1:
uee->punto_de_entrada= 0;
goto _EtPE_0001;

case 2:
uee->punto_de_entrada= 0;
goto _EtPE_0002;

default:
Error( FATAL,
"Punto de entrada no válido." );
}

switch( transicion )
```

```
{
...

case n1:
...
uee->punto_de_entrada= 1;
Call( ... );
return(RET_UNIDAD_OK);
_EtPE_0001:;
...

case n2:
...
uee->punto_de_entrada= 2;
Call( ... );
return(RET_UNIDAD_OK);
_EtPE_0002:;
...

...
}
}
```

Otros posibles métodos que se barajaron para solucionar este problema (entre ellos el uso se setjmp() y longjmp()) fueron descartados debido a su complejidad.

7.1.19.- Variables exportadas: IMPORT.

Una variable declarada como EXPORTED constará en realidad de dos variables: la del programa y la de exportación. La función Export() copiará el valor de la primera en la segunda. La función Import() devolverá un VALOR * que será el de la segunda variable. La función Import() acepta tres argumentos: el índice de la variable a importar (no de la variable duplicada, sino de la original), el tipo de unidad de la que se debe importar la variable y una expresión de tipo PID que indica la unidad de la que se importará. Esta última es opcional, y en caso de que se omita (se hace NULL) no deberá haber más de una instancia de la unidad exportadora.

7.1.20.- Variables reveladas: VIEW.

El valor de una variable que se declaró con DCL REVEALED puede ser

visto mediante la función C equivalente View(). Muy similar a Import(), acepta tres argumentos (el índice de la variable, el tipo de unidad que la reveló y el PID opcional de una unidad de ese tipo, en caso de que haya más de una).

7.1.21.- OUTPUT.

La función Output() aceptará los siguientes parámetros:

- El identificador de la señal que debe ser enviada (UWORD).

- El tipo de unidad al que se debe enviar la señal. Es válido tanto si el OUTPUT es implícito (la señal se enviará a la unidad de ese tipo, que debe ser única) como si es explícito (se usará para comprobar que el PID dado designa una unidad del tipo correcto).

- El PID al que hay que enviar la señal. Se da en forma VALOR *. Si es NULL, significa que el Output es implícito (se usa el tipo de unidad).

- El número de argumentos que tiene la señal. Se podría omitir, pero entonces debería haber una tabla que describiese las señales, indicando entre otras cosas su número de argumentos (esto se hace con las unidades, en la tabla de descripción de unidades o tdu).

- Cero o más VALOR * que contienen los argumentos de la señal.

La función determinará el tipo de salida (implícita si pid==NULL o explícita en caso contrario) y buscará la cola de la unidad receptora.

A continuación creará la señal, almacenará sus argumentos (si los hay) y la añadirá al final de la cola.

7.1.22.- Temporizadores: SET, RESET, ACTIVE, NOW.

En el estándar SDL no se especifica cuál debe ser la unidad de tiempo. En esta implementación, se traduce el tipo predefinido TIME al tipo C clock_t. El tipo clock_t permite almacenar el tiempo que ha transcurrido desde el inicio del programa. La función clock() nos da ese tiempo. El estándar ANSI C no especifica tampoco las unidades en que se mide este tiempo. En el compilador C de SUN, las unidades son microsegundos. En MSC, milisegundos.

La función equivalente Now() devolverá un puntero a un VALOR de tipo

TIME que contendrá el tiempo de ejecución (clock()).

Los temporizadores se implementarán de la siguiente forma: una lista enlazada de estructuras de tipo TIMER (l_timers) mantendrá los temporizadores activos en el sistema. La función equivalente Active() aceptará un identificador de señal (de timer, en este caso) y una serie de VALORes que constituyen los argumentos del temporizador. Buscará por toda la lista de temporizadores y retornará un VALOR de tipo booleano, que será TRUE si encontró el temporizador especificado, y FALSE si no lo halló.

La función Set() recibirá como parámetros el identificador del timer, sus argumentos y un VALOR de tipo time que es el tiempo de disparo del temporizador. La función borrará de l_timers el timer indicado, si es que existe (y, en caso de que haya enviado una señal que aún no se haya consumido, la borrará de la cola del proceso correspondiente). A continuación, creará un nuevo timer y lo añadirá a la lista. Aquí hay que tener en cuenta algo muy importante: la función añadirá el timer a la lista manteniéndola ordenada por valores crecientes de los campos <t_disparo>. Así se simplifica la tarea de la función de comprobación ChequeaTimers(), que se llama al inicio de cada ciclo de control.

La instrucción SDL RESET se implementa mediante la función equivalente C Reset(), que acepta los mismos argumentos que Active(): el identificador del timer y sus parámetros. Esta función buscará en l_timers el timer indicado. Si no lo encuentra, simplemente retornará. Si lo encuentra lo borrará de la lista, comprobando previamente si el timer ya se ha disparado (ha enviado su señal). Si se ha disparado y está en la lista es que la señal no ha sido consumida (a continuación veremos que tras consumir una señal de tipo timer, la unidad desactiva el temporizador), y por lo tanto ha de ser eliminada de la cola de la unidad correspondiente.

Como se ve, los estados de un temporizador se pueden determinar fácilmente: si el temporizador está en la lista l_timers es que está activo. Si además su campo <senal_enviada> no es NULL, es que se ha disparado, ha enviado la señal y está esperando a que la unidad receptora la consuma y lo desactive. Un timer está inactivo si no está en la lista de temporizadores.

En cada llamada a EjecutaUee() se llama a la función ChequeaTimers(), que recorre la lista de temporizadores buscando aquellos cuyo tiempo de disparo haya pasado y lanzando sus señales. Gracias a que la lista está ordenada, los timers que están los primeros son los que primero se dispararán. Así, si la función ChequeaTimers() encuentra en su recorrido el primer timer que no se dispara, puede ignorar los siguientes, ya que tampoco les habrá llegado su tiempo de disparo. Puede que un timer se haya disparado ya y aún siga en la lista esperando a que su señal sea consumida.

La función ChequeaTimers() detecta esto y no vuelve a lanzar la señal.

Hemos visto cómo detectar si un timer está activo (Active()), cómo activarlo (Set()), cómo desactivarlo (Reset()) y cómo lanzar las señales de los timers que se hayan disparado (ChequeaTimers()). Pero el estándar SDL dice que un timer sólo se desactiva implícitamente cuando su señal ha sido consumida. Para ello, se emplea la función LiquidaTimer(), que acepta como argumento una señal, busca en la lista de temporizadores el timer que la envió y lo borra (lo desactiva). Esta función se utiliza en _uniddes.c en los switch de carga de argumentos que aparecen en la cabecera de las transiciones de tipo INPUT.

7.1.23.- TASK.

Hay dos tipos de tareas: texto informal y asignaciones.

- **Texto informal.**

Simplemente se imprimirá el texto. En un principio se usaba la función C printf() de la siguiente forma:

```
printf( "%s", "<texto informal>" );
```

o incluso

```
printf( "%s\n", "<texto informal>" );
```

Pero, creyéndolo más claro, se cambió por:

```
printf( "<texto informal>" );
```

Más tarde se descubrió que esto podía dar problemas en caso de que se quisieran imprimir caracteres especiales que la función printf() reconociera como códigos de formato, así que se volvió a la primera de las tres formas.

Por último se adoptó la solución de usar una función equivalente, TextoInformal(), que acepta como parámetro el texto que se ha de imprimir. La función se limita a llamar a printf de la primera forma explicada y a vaciar el buffer de salida, para mantener actualizado stdout en caso de que varias salidas se dirijan hacia allá (por ejemplo, la de depuración o la de error).

El que se use la cadena de texto informal literalmente y entre comillas permite que en ella sean usados los caracteres de control de impresión de C

('\n', '\t', etcétera).

- **Asignaciones.**

Se usará la función Asigna() (ver más arriba, en el apartado dedicado a expresiones).

7.1.24.- ANY.

La función equivalente Any() acepta un parámetro de tipo BYTE que debe ser una constante simbólica TVL_... designando el tipo de VALOR que retornará la función. Any() se encarga de crear un valor arbitrario del tipo indicado.

7.1.25.- PARENT, SELF, SENDER, OFFSPRING.

Se traducirán a cuatro funciones C que retornarán punteros a VALOR. Las estructuras VALOR serán de tipo PId. En el modo Geode, SENDER no se hará SELF al entrar en una transición espontánea o señal contínua (ver apéndice A).

7.1.26.- Expresiones condicionales.

Las construcciones SDL de tipo IF <expr> THEN <expr> ELSE <expr> FI se traducirán a una función C IfThenElse() que aceptará tres parámetros de tipo puntero a VALOR (la expresión de decisión y las dos alternativas) y retornará el VALOR * adecuado.

7.1.27.- Estructuras, arrays, powersets y strings.

Una estructura se representará mediante el campo <estruc> de la unión de datos del tipo VALOR. Este campo es un array de punteros a estructuras de tipo VALOR, con tantas entradas como campos tenga la mayor estructura definida en el sistema.

El acceso a un campo de una estructura se hará mediante la función Campo(), que retornará el puntero a VALOR correspondiente. Este puntero se puede usar tanto el primer argumento de la función Asigna() (es decir, como LValor) como en cualquier expresión (como RValor).

Los arrays estarán implementados como listas de estructuras VALOR.

Un array de n elementos será una lista de 2n VALORes. Cada elemento del array estará representado por dos valores consecutivos de la lista. El primero será el índice y el segundo el ítem asociado a ese índice. La función IndArray() acepta dos argumentos de tipo VALOR *: el array que hay que referenciar y el índice. IndArray() buscará en el campo <array> de la unión <dato> del primer argumento hasta que encuentre el índice. Si lo encuentra, retornará el siguiente elemento de la lista enlazada, que es el ítem asociado al índice. Si no encuentra el índice, creará un par índice-ítem nuevo y retornará el ítem. Obsérvese que ese ítem será un VALOR de tipo TVL_VACIO. Así, las referencias de lectura a un elemento inexistente en un array darán resultados no previstos.

Los conjuntos (powersets) estarán representados como listas enlazadas de VALORes, al igual que las cadenas (strings).

7.2.- Los ficheros.

El sistema equivalente se construye a partir de una serie de ficheros de código C (extensión .c) y de cabecera (extensión .h). Entre estos ficheros hay que distinguir dos tipos: los ficheros generados y los del armazón.

Los ficheros generados son los que cambian dependiendo del sistema SDL que se simule. Por lo tanto, estos son los ficheros creados por el traductor. Para distinguirlos, su nombre comienza por '_'.

Los ficheros del armazón contienen las funciones 'de librería' del sistema equivalente: control de la tabla de unidades, acciones, expresiones, depuración, ejecución... Estas funciones son independientes del sistema SDL que se use como fuente. Todos los ficheros cuyos nombres no comiencen por '_' forman parte del armazón.

7.2.1.- Generados.

Se dará a continuación una breve reseña sobre los ficheros generados.

- **_CTES.H**: define una serie de constantes que serán usadas por otros ficheros:
  - NUM_UNIDADES: número de unidades en el sistema. Incluye ENV.
  - NUM_SENALES: número de señales en el sistema.
  - NUM_MAX_ESTADOS: numero de estados menos uno de la unidad que más tenga. Menos uno porque el estado 0 (START) no se considera.
  - NUM_MAX_CAMPOS: número de campos de la estructura que más tenga.
  - NUM_MAX_VARS: número de variables de la unidad que más tenga.
  - NOM_SISTEMA: nombre del sistema SDL simulado.
  - FICH_ORIGEN: fichero de código SDL en el cual está almacenado el sistema generado.

- **_NOMSEN.C**: definición del vector <nom_senales>, que contiene los nombres de las señales definidas en el sistema.

- **_NOMEST.C**: definición del vector <nom_estados>, que contiene los nombres de los estados de cada unidad.

- **_NOMVAR.C**: definición del vector <nom_variables>, que contiene los nombres de las variables de cada unidad.

- **_TDC.C**: definición de la tabla de comportamiento de las unidades del sistema.

- **_TDU.C**: definición del vector <tdu>. Por cada unidad se incluyen los datos de iniciación de una estructura tipo ENTRADA_TDU.

- **_UNIDDES.C**
- **_UNIDDES.H**: funciones equivalentes, funciones de iniciación de variables y prototipos.

- **_DEFINES.H**: definición de constantes asociadas a las variables y a los estados de cada unidad.

- **_INICIA.C**: contiene la función IniciaSistema(), que crea los procesos iniciales.

7.2.2.- Del armazón:

Los ficheros que forman el armazón del sistema equivalente son:

- **GENERADS.C**: fichero que agrupa todos los ficheros generados para facilitar la compilación.

- **ACCIONES.C**
- **ACCIONES.H**: contienen las funciones equivalentes a las instrucciones SDL (Create, Call, Stop, Return, NextState...) y sus prototipos.

- **AUXL.C**
- **AUXL.H**: definen funciones auxiliares de búsqueda en la tabla de unidades, creación y borrado de estructuras...

- **DEB.C**
- **DEB.H**: función Debug y su prototipo.

- **DEFS.H**: define los principales tipos, estructuras de datos y constantes: UWORD, BYTE, T_..., SIT_..., TVL_..., VALOR, ENTRADA_TU, ENTRADA_TDU...

- **ERR.C**
- **ERR.H**: función Error y su prototipo.

- **EXPRS.C**
- **EXPRS.H**: definen las funciones de tratamiento de expresiones y

valores.

- **GETOPT.C**
- **GETOPT.H**: tratamiento de la línea de comandos.

- **LISTAS.H**: macros para simplificar las operaciones sobre listas enlazadas.

- **MAIN.C**: contiene la función main(), que inicia el sistema y ejecuta el bucle principal y el sistema de depuración.

- **MAKEFILE**: fichero de construcción de la plataforma actual. Puede ser MAKEFILE.DOS o MAKEFILE.UNX

- **MAKEFILE.DOS**: fichero de construcción para la utilidad NMAKE de Microsoft C v6.0.

- **MAKEFILE.UNX**: fichero de construcción para la herramienta MAKE de UNIX.

- **MEM.C**
- **MEM.H**: funciones y macros de reserva y liberación de memoria.

- **PRTFUN.H**: definición de la macro PRTFUN. Para más información, ver el comentario sobre PRTFUN.H en la sección del traductor.

7.3.- Las estructuras.

El sistema equivalente usa durante su ejecución una serie de estructuras de datos que mantienen información sobre el estado de la ejecución. Las veremos a continuación, comenzando por la de nivel superior (ENTRADA_TDU) y tratando a continuación de una dada las que englobe.

- **ENTRADA_TDU**:

Las estructuras de tipo ENTRADA_TDU describen unidades. Se agruparán en la tabla de descripción de unidades (tdu). Los campos de que constan son los siguientes:

- tipo [BYTE]: tipo de unidad: TIP_PTO si es un procedimiento o TIP_PSO si es un proceso.

- nombre [char *]: nombre de la unidad.

- inst_min, inst_max [UWORD]: números mínimo y máximo de instancias permitidas. Sólo son útiles si la unidad es un proceso. La constante INST_INF indica un número infinito de instancias.

- inst_act [UWORD]: número actual de instancias activas de la unidad, en caso de que ésta sea un proceso.

- tiene_trans_espontaneas [BYTE]: indicador de si se ha definido alguna transición espontánea en la unidad.

- numero_de_variables [UWORD]: número de variables locales de la unidad. Incluye los parámetros.

- cuerpo [UWORD (*)(UWORD, SENAL *)]: puntero a una función que acepta dos parámetros de tipo UWORD y SENAL * respectivamente, y retorna un valor de tipo UWORD. Esta es la función equivalente de la unidad.

- inicia_vars [void (*)(void)]: puntero a una función sin argumentos y sin valor de retorno que es la encargada de iniciar las variables de la unidad.

- tabla_de_comportamiento [UWORD *]: puntero a una matriz bidimensional de UWORDs que constituye la tabla de comportamiento de la función.

- primero_en_tu [ENTRADA_TU *]: puntero a la primera unidad del tipo descrito que esté en la tabla de unidades. NULL si no hay ninguna.

- numero_de_parametros [UWORD]: número de argumentos que recibe la unidad.

- **ENTRADA_TU**:

Esta estructura formará la lista de unidades en ejecución, la principal fuente de información sobre los procesos del sistema. Sus campos son:

- tipo [UWORD]: índice dentro del vector tdu de la estructura ENTRADA_TDU que describe a esta unidad.

- self, sender, parent, offspring [T_PID]: PIDs propio, del remitente de la última señal, del padre y del último hijo creado, respectivamente.

- situacion [BYTE]: una constante de tipo SIT_... indicando cómo está la unidad:

- SIT_MUERTA indica que la unidad ejecutó una instrucción STOP o RETURN (dependiendo de si es un proceso o un procedimiento) y está esperando a ser elegida para ejecutarse. Cuando lo sea, el bucle de ejecución detectará su situación y la eliminará de la tabla de procesos.

- SIT_ACTIVA significa que la unidad está lista para ejecutar transiciones. Cuando el bucle principal la seleccione, decidirá qué transición debe ejecutar en base a su estado actual y a su cola de señales.

- SIT_ESPERANDO_FIN_PTO quiere decir que la unidad llamó a un procedimiento (instrucción CALL) y está hibernada esperando que éste termine.

- primera_ejecucion [BYTE]: indicador de la primera ejecución de la unidad. Cuando el bucle principal detecta que una unidad se ejecuta por primera vez, llama a la función de iniciación de variables y pone este indicador a FALSE.

- estado_actual [UWORD]: estado actual de la unidad. El estado cero es el inicial (START).

- punto_de_entrada [UWORD]: en caso de que la unidad haya llamado a un procedimiento, este campo almacena el número de orden de la llamada dentro del código de la unidad. Esto permite que la transición que se ejecutaba cuando se procesó el CALL se reanude justo después de la llamada a procedimiento.

- pto_creado [ENTRADA_TU *]: indica el último procedimiento creado por la unidad. Se utiliza en la función EjecutaUee().

- cola [SENAL *]: cola de señales asociada a la unidad.

- vars [VALOR *]: lista de valores que representa las variables locales y argumentos de la unidad.

- sig, ant [ENTRADA_TU *]: punteros a los elementos siguiente y anterior dentro de la tabla de unidades.

- sig_hermano, ant_hermano [ENTRADA_TU *]: punteros a la siguiente y anterior unidades del mismo tipo que la actual en la tabla de unidades.

- **SENAL**:

Esta estructura se usa, enlazada, para representar las colas de señales de las unidades. Por separado, permite especificar una señal. Los campos de una estructura de tipo SENAL son:

- senal [UWORD]: código de la señal.

- remitente [T_PID]: PID del remitente de la señal.

- argums [VALOR *]: lista de valores que representa los argumentos de la señal.

- sig [SENAL *]: puntero a la siguiente señal en la lista, si es que la hay.

- **VALOR**:

En el sistema equivalente, las estructuras de tipo VALOR representan valores de cualquier tipo definido. Se usan en expresiones, como variables, como argumentos...

- tipo [BYTE]: una constante TVL_... indicando el tipo de valor contenido en la estructura (TVL_VACIO, TVL_BOOLEAN, TVL_POWERSET, TVL_STRUCTURE, etc.).

- estatico [BYTE]: TRUE si se trata de un valor de tipo estático (que no debe ser borrado implícitamente). FALSE si es un valor volátil. Para más información, ver la discusión sobre expresiones en el esquema de traducción.

- dato [unión]: unión que contiene el valor en sí. Hay campos para cada uno de los tipos básicos, punteros a listas de valores para los tipos string, powerset, array (ya que se implementan así) y un vector de punteros a VALOR para implementar las estructuras.

- sig [VALOR *]: puntero al siguiente VALOR en la lista (las estructuras VALOR son enlazables).

- **L_VALORES**:

Se utiliza esta estructura cuando hay que mantener una lista de referencias a VALORes, pero no se desea enlazarlos internamente. Los campos son:

- valor [VALOR *]: el ítem.

- sig [L_VALORES *]: el puntero al siguiente nodo.

- **TIMER**:

Esta estructura enlazable almacena información sobre los temporizadores activos en el sistema. Sus campos son:

- senal [UWORD]: identificador de la señal asociada al temporizador.

- args [VALOR *]: lista de argumentos del timer.

- t_disparo [T_TIME]: punto en el que se disparará el temporizador.

- llamador [T_PID]: PID del proceso que activó el temporizador.

- senal_enviada [SENAL *]: NULL si el temporizador no se ha disparado aún. En el caso de que sí se haya disparado, apunta a la SENAL enviada como resultado.

- sig [TIMER *]: puntero de enlace.

7.4.- Las variables globales.

El sistema equivalente mantiene variables globales encargadas de almacenar información sobre el estado y la configuración del mismo. Las veremos a continuación, explicando brevemente su función.

- tdu [ENTRADA_TDU [NUM_UNIDADES]]: este vector de NUM_UNIDADES elementos de tipo ENTRADA_TDU contiene la descripción de los tipos de unidad definidos en el sistema. Se inicia en el fichero _tdu.h y se usa en una gran parte de las funciones del sistema equivalente.

- tu [ENTRADA_TU *]: esta es la cabeza de la lista enlazada de las unidades en ejecución, es decir, de la tabla de unidades. Esta lista contiene a todas las unidades que están disponibles para su ejecución, las que están esperando el final de un procedimiento invocado por ellas y las que están terminadas (por una instrucción STOP o RETURN) esperando a que el bucle principal las elimine de la tabla. Las funciones equivalentes de CREATE y CALL, así como el bucle principal, se ocupan del manejo de esta tabla.

- uee [ENTRADA_TU *]: apunta a la unidad de la tabla de unidades que está seleccionada para ejecución. Es utilizada por un buen número de

funciones.

- l_timers [TIMER *]: lista de los temporizadores activos en el sistema. Es modificada y consultada por las funciones Set(), Reset(), Active(), ChequeaTimers() y LiquidaTimer().

- nom_senales [char * [NUM_SENALES]]: contiene los nombres de las señales definidas en el sistema. Se usa para las salidas de depuración y error.

- nom_estados [char * [NUM_UNIDADES][NUM_NAX_ESTADOS+1]]: el elemento [i][j] es una cadena de caracteres que contiene el nombre del estado j-ésimo de la unidad cuyo identificador es <i>. También se usa en depuración y errores.

- debflag [BYTE]: indicador de si el modo de salida de depuración está activo.

- deb [FILE *]: fichero de salida de depuración.

- modo_deb [BYTE]: indicador de si el modo de depuración está activo. Este modo es el modo de trazo, y no debe confundirse con el de salida de depuración.

- niv_deb [int]: cuando se está en modo de salida de depuración, cada mensaje de información va acompañado de un nivel. Si el nivel es menor o igual que <niv_deb>, el mensaje se imprimirá. La variable <niv_deb> debe contener un valor entero entre 0 y 2. El nivel de depuración 0 es el menos explícito. El nivel 2 genera una gran cantidad de mensajes de depuración.

- modo_geode [BYTE]: indicador de si se está en modo Geode. En el modo Geode, el uso de una variable no iniciada en una comparación no produce un error dinámico. La variable tendrá un valor nulo.

7.5.- Los procedimientos.

Entre las funciones que componen el sistema equivalente se puede establecer una clasificación bastante clara. En primer lugar están las funciones equivalentes, que modelan el comportamiento de las unidades definidas en el sistema SDL. Las funciones de manejo de VALORes sirven para evaluar expresiones. Las funciones equivalentes a acciones son la implementación de las instrucciones SDL (OUTPUT, CREATE, TASK, etcétera). Las funciones auxiliares son usadas por otras funciones para realizar tareas generales. Por fin, las funciones de control son las que guían la ejecución del

sistema, llamando a las funciones equivalentes con los argumentos adecuados.

Las funciones equivalentes más importantes son IniciaSistema(), Unidad_XXXX() e InicVars_XXXX() (donde XXXX es el identificador de la unidad). La primera se encarga de crear las unidades iniciales del sistema. La segunda es la función que implementa el comportamiento de la unidad XXXX (ver la sección dedicada al esquema de traducción). La última es llamada antes de la primera ejecución de una unidad, y su cometido es iniciar los valores de las variables de la unidad (de las que tengan valores iniciales).

Como se dijo en la sección del esquema de traducción, las expresiones en el sistema equivalente están implementadas mediante funciones y estructuras de tipo VALOR. Las funciones de manejo de VALORes son múltiples. Algunas de las más importantes y representativas son Suma(), ValorBoolean(), Variable(), Asigna(), Active()... La primera de ellas acepta dos VALORes y los suma, creando un nuevo VALOR y retornándolo. La segunda construye un VALOR a partir de un valor de tipo booleano (obsérvese que no es lo mismo un VALOR que un valor). La función Variable(), con parámetro entero <n>, devuelve la variable n-sima de la unidad en ejecución. Asigna() acepta dos VALORes y hace el primero igual al segundo. Active() implementa la función SDL ACTIVE, que comprueba si un temporizador con unos argumentos opcionales está activo.

Las acciones de SDL (SET, RESET, OUTPUT...) se implementan mediante funciones equivalentes con el mismo nombre (como marca el manual de estilo, la primera letra en mayúscula y el resto en minúscula). Así, la función Call() acepta el identificador del tipo de procedimiento que ha de crear y cero o más argumentos para el mismo. La función NextState acepta el identificador del nuevo estado, etcétera.

Todas estas funciones (y las de control, que veremos a continuación) deben realizar una serie de tareas genéricas que están implementadas por las funciones auxiliares. Hay funciones auxiliares para crear nuevos VALORes (CreaValor()), para crear nuevos PID (NuevoPid()), para añadir entradas a la tabla de unidades (AnadeEntradaTU())... También podemos considerar auxiliar la función AnalizaLineaDeComandos(), llamada desde main(), que activa los indicadores adecuados de acuerdo con las opciones pasadas en la línea de llamada.

Y por fin, el control de la ejecución del sistema equivalente se lleva a cabo por las funciones main() y EjecutaUee(), principalmente, ayudadas en la depuración por ListaTU(), ListaTDU(), ListaInfo()...

La función main() puede controlar la ejecución de dos formas: libre o de depuración. En la ejecución libre, simplemente llama a EjecutaUee() hasta que se vacíe la tabla de procesos. Si se especifica la opción -D en la línea de comandos, la ejecución se hará en modo de depuración. Se nos indicará en cada paso la unidad actual y se permitirá ejecutarla, listar información, cambiar parámetros, ver variables... Este modo está explicado con más detalle en la sección siguiente.

7.6.- El modo de depuración.

El sistema equivalente se ejecuta llamando a las funciones equivalentes de las unidades que hay en la tabla de ejecución (tabla de unidades) hasta que esta quede vacía. Estas llamadas pueden hacerse de forma contínua e independiente o de una forma controlada por el usuario. Esta segunda forma es el modo de depuración.

Cuando se especifica la opción -D en la ejecución del sistema equivalente, se activa un modo especial que permite trazar el sistema paso a paso, obteniendo información sobre el estado del mismo en cada momento. El modo de depuración es un bucle que presenta en cada iteración los datos básicos de la unidad en ejecución (nombre, PID, estado...) y permite introducir una orden representada por un carácter (posiblemente con argumentos).

Las órdenes de depuración actualmente implementadas son las siguientes (entre paréntesis se indica el carácter la representa y la sintaxis):

- **TU** (T): lista información sobre todas las unidades activas en el sistema, es decir, lista la tabla de unidades. Para cada una de ellas da su PID, tipo, situación, nombre, estado actual y señal en cabeza de la cola.

- **TDU** (D): lista los contenidos de la tabla de descripción de unidades (tdu). Para cada unidad definida en el sistema, da los siguientes datos: tipo de la unidad, clase (proceso o procedimiento), nombre, números inicial, máximo y actual de instancias, indicador de si tiene transiciones espontáneas, número de variables y de argumentos y PID de la primera instancia de esa unidad en la tabla de unidades (si hay alguna).

- **EXEC** (X [n]): ejecuta la transición correspondiente de la unidad en ejecución (llamando a la función EjecutaUee()) y calcula la siguiente unidad a ejecutar, asignándola a <uee>. Repite esta operación n veces, en caso de que se especifique el argumento n.

- **GO** (G): entra en un bucle de ejecución que llama a EjecutaUee()

hasta que la tabla de unidades esté vacía.

- **UEE** (U pid): hace que la unidad en ejecución sea la indicada por el PID dado.

- **INFO** (I [pid]): lista información sobre una unidad. Si se da el argumento opcional pid, proporciona datos detallados sobre la unidad con ese PID: tipo, nombre, valores SELF, SENDER, PARENT y OFFSPRING, situación, estado de primera ejecución, estado actual, cola de señales y variables. Si no se da ningún argumento, la información dada será la de la unidad en ejecución (uee).

- **PRINT** (P var [pid]): permite especificar el nombre de una variable y un PID opcional (por defecto será el de la unidad en ejecución, uee) e imprime el tipo y el valor de la variable dada en la unidad especificada por el PID.

- **VERB** (V [n]): si se indica un argumento, ajusta el nivel de mensajes de depuración. Si no se indica nada, informa de dicho nivel.

Cuando se ejecuta una transición con la función EjecutaUee(), se muestran mensajes en la salida estándar indicando qué instrucciones SDL se están ejecutando y qué acciones de control se están llevando a cabo. La cantidad de estos mensajes depende del nivel de mensajes. Este es un valor entre 0 y 2. El nivel 0 no proporciona mensajes. El nivel 1 produce una salida de depuración normal. El nivel 2 genera gran cantidad de mensajes de depuración.

- **QUIT** (Q): termina la ejecución del sistema equivalente y vuelve al sistema operativo.

- **HELP** (H): proporciona una breve referencia de estos comandos.

7.7.- Línea de comandos.

La línea de comandos del sistema equivalente es la siguiente:

```
EQV [opciones]
```

Las opciones pueden ser:

-D: se activará el modo de depuración. Véase el apartado anterior.

-d <num>: se generará una salida de depuración con nivel <num> (el nivel ha de estar en el rango 0-2, y a mayor nivel corresponde mayor información en la salida).

-m: se activará el trazo de memoria y la salida se dirigirá a memoria.mem.

-M <fmem>: igual que -m, pero la salida de trazo de memoria irá al fichero <fmem>.

-a: las salidas de depuración y trazo de memoria se activarán, e irán juntas a stdout (la salida estándar).

-G: se activará el modo Geode, en el cual las variables no iniciadas presentan valores nulos cuando son usadas en comparaciones. En el modo normal (estándar SDL) el uso de cualquier variable antes de su primera asignación genera un error dinámico.

8.- El preprocesador.

Uno de los programas diseñados en este proyecto es LEDPP, un preprocesador SDL de una sola pasada que acepta las definiciones de macros y los usos de las mismas. Por ser de una sola pasada, no permite el uso de una macro antes de su definición (el estándar SDL especifica que esto debe ser posible).

En un principio se intentó implementar el preprocesador usando yacc y una gramática restringida que sólo considerase las producciones relativas a la definición y uso de las macros. Pero este planteamiento presentó una serie de inconvenientes que motivaron el abandono del análisis LARL(1) que proporciona yacc y la adopción de un analizador descendente recursivo implementado mediante funciones. Teniendo en cuenta la simplicidad de la gramática, esta solución es mucho más sencilla y evita los problemas que daba la anterior.

Para el análisis léxico sí se ha mantenido la implementación mediante un fichero de descripción de patrones y la herramienta lex.

8.1.- Los ficheros.

Describiremos brevemente los ficheros que contienen el código fuente del programa LEDPP y el contenido de cada uno.

- **AUXL.C**
- **AUXL.H**: contienen algunas funciones auxiliares (y sus prototipos) de tratamiento de cadenas (DupCad, MataEspacios...) y la función de impresión de mensajes de error.

- **GETOPT.C**
- **GETOPT.H**: tratamiento de las opciones pasadas en la línea de comandos.

- **LEDPP.C**: fichero generado por lex (o pclex o cualquier herramienta compatible) a partir de LEDPP.L.

- **LEDPP.H**: definiciones de algunas constantes y tipos de datos necesarios para el sistema.

- **LEDPP.L**: contiene el fichero de descripción de patrones que forma el analizador lexicográfico (básicamente es un subconjunto del analizador léxico del traductor LEDAC) y las funciones de análisis sintáctico

descendente recursivo.

- **LISTAS.H**: macros para simplificar las operaciones sobre listas enlazadas.

- **MAKEFILE**: fichero de construcción del sistema actual. Puede ser MAKEFILE.DOS o MAKEFILE.UNX dependiendo de la plataforma.

- **MAKEFILE.DOS**: fichero de construcción para la utilidad make (NMAKE) de Microsoft C v6.0.

- **MAKEFILE.UNX**: fichero de construcción para la utilidad MAKE de UNIX.

- **PRTFUN.H**: macros para permitir el uso de prototipos con parámetros en los compiladores no-ANSI. Para una discusión más detallada, ver la sección dedicada a los ficheros del traductor.

8.2.- Las estructuras.

Sólo hay dos estructuras definidas en este programa:

- **CADS**:

Permite representar listas enlazadas de cadenas. Tiene dos campos:

- cad [char *]: el elemento (la cadena) contenido en este nodo.

- sig [CADS *]: puntero al siguiente nodo de la lista, o NULL si no hay más nodos.

- **MACRO**:

Esta estructura enlazable formará la lista de macros definidas. Sus campos son:

- nombre [char *]: nombre de la macro.

- n_args [int]: número de argumentos.

- args [CADS *]: nombres de los argumentos.

- expansion [CADS *]: cuerpo de la macro, en forma de lista de cadenas.

- sig [MACRO *]: puntero de enlace con la siguiente macro definida, si hay alguna. Si esta es la última de la lista, <sig> es NULL.

8.3.- Las variables globales.

Hay una serie de variables globales definidas en el programa LEDPP. Trataremos a continuación las más importantes.

- fichpp y fichsalpp [char [MAX_FICH]]: son los nombres de los ficheros de entrada y salida del preprocesador. Se asignan en la función AnalizaLineaDeComandos().

- yyin y ppout [FILE *]: yyin es el fichero de entrada estándar a un analizador léxico generado por lex. Por defecto es la entrada estándar (stdin), pero la función IniciaElInvento() puede reasignarlo. El fichero ppout es la salida del preprocesador. Se inicia en la misma función.

- lineapp [int]: indica la línea del fichero SDL que se está analizando en este momento. Se inicia a cero y se actualiza en las acciones asociadas a los patrones. Se usa en los mensajes de error.

- debug [int]: indicador que vale 1 si se ejecuta el programa en modo de depuración. Vale 0 en caso contrario. Se asigna en AnalizaLineaDeComandos, dependiendo de la presencia de la opción '-d'.

- macroid_count [int]: es un contador que se inicia a 0 y se incrementa con cada uso. Sirve como fuente para los identificadores macroid.

- macros [MACRO *]: lista enlazada de macros definidas en el sistema. Se inicia a NULL y se le van añadiendo macros conforme se definen. Es consultada durante el análisis de una construcción MACRO <nombre> ... (es decir, un uso de macro).

8.4.- Las funciones.

Analizaremos a continuación las funciones principales que forman parte del programa LEDPP. El orden de análisis es el orden de aparición en el árbol de dependencias generado por la herramienta CST (un analizador de estructura de programas C diseñado por J.M. Software): cada función está inmediatamente seguida por las funciones que invoca. Una función sólo aparece la primera vez que es invocada.

- void main(int, char *[]): la función principal. Llama a las de iniciación y a la cabecera del análisis.

- void AnalizaLineaDeComandos(int, char *[]): obtiene las opciones (switches) pasadas en la línea de comandos (llamando a la función getopt()) y los nombres de los ficheros, si éstos se han dado. En caso contrario, les asigna valores por defecto.

- int getopt(int, char *[], char *): en cada llamada busca en lo que le queda de la línea de comandos una de las opciones especificadas por el tercer argumento y la retorna (las opciones pueden tener argumentos asociados). Como se ve, esta es una función que recuerda su historia, es decir, que no tiene por qué dar el mismo valor de retorno ante dos llamadas con los mismos argumentos.

- void IniciaElInvento(): efectúa tareas de iniciación del sistema: abre los ficheros de entrada y salida y da valores a algunas variables globales.

- void LeeSistema(): función de cabecera del análisis recursivo. Lee símbolos terminales (usando la función Token()) hasta que se acabe la entrada. Si los símbolos son tokens normales, los copia a la salida. Si son palabras clave que delimitan el inicio de una construcción que ha de ser tratada de forma diferente (MACRODEFINITION o MACRO) llama a las funciones correspondientes (LeeDefMacro() y LeeUsoMacro()).

- void Token(int): llama a la función yylex() para obtener el siguiente token del fichero de entrada. Si el argumento le indica que debe saltar los espacios (SKIP), repite este proceso hasta que el token retornado por yylex() sea un no-espacio.

- int yylex(): función generada por lex a partir de los patrones. Contiene una serie de acciones que deben ser ejecutadas al reconocer cada patrón.

- void Error(int, char *, ...): función de generación de mensajes de error. Su comportamiento es idéntico a printf(), pero antepone al texto impreso el mensaje 'ERROR: (<fich>, <lin>): ', donde <fich> es el nombre del fichero de entrada y <lin> es el número de línea que se está analizando.

- void LeeDefMacro(): analiza una construcción de definición de macro y añade un nuevo elemento a la lista de macros definidas.

- unsigned char *NewMalloc(size_t): reserva un número de bytes en memoria mediante la función malloc() y comprueba que no haya habido

problemas.

- void LeeNombre(char **): lee el siguiente token y, si es un nombre, lo retorna en el argumento. Si no lo es, genera un error de sintaxis.

- char *DupCad(char *): crea un duplicado de la cadena pasada como argumento.

- void LeeParsFormales(MACRO *): efectúa el análisis de los parámetros formales de la macro en definición y los almacena en el campo correspondiente de ésta.

- void LeeCuerpoMacro(MACRO *): lee la expansión de la macro.

- void LeeUsoMacro(): analiza una construcción de tipo MACRO <nombre>..., es decir, un uso de macro.

- int LeeParsReales(CADS **): comprueba si la referencia a macro que se está analizando proporciona parámetros reales. Retorna los parámetros en su primer argumento y el número de estos como su valor de retorno.

- char *CortaEspacios(char *): duplica su argumento, eliminando los espacios iniciales y finales.

8.5.- La gramática del preprocesador.

La gramática que acepta el analizador descendente recursivo implementado en LEDPP se presenta a continuación en formato EBNF. Los símbolos terminales aparecen en negrita y los no-terminales en minúscula.

El símbolo terminal **TOKEN** ha de aceptar cualquier token. Esto es fácil de hacer con un analizador por funciones, pero no con uno por tablas generado por yacc. Este fue el problema que motivó la elección del descenso recursivo.

Obsérvese que las producciones se corresponden con las funciones del analizador con el mismo nombre.

```
sistema: { MACRODEFINITION defmacro | MACRO usomacro | otro }*


defmacro: nombre [FPAR parsformales] cuerpomacro [nombre] ;

nombre: NOMBRE
```

```
parsformales: nombre {, nombre}*

cuerpomacro: ; {TOKEN}* ENDMACRO


usomacro: nombre parsreales ;

parsreales: [ ( {TOKEN}* {, {TOKEN}*}* ) ]
```

8.6.- Línea de comandos.

La línea de llamada al preprocesador de SDL (LEDPP) es la siguiente:

```
LEDPP [opciones] [fich.sdl] [fich.pp]
```

donde fich.sdl es el fichero de entrada y fich.pp es el fichero de salida preprocesado. Los valores por defecto son la entrada estándar y la salida estándar, respectivamente.

Las opciones pueden ser:

-d: se activará el modo de depuración y se generará la salida en stdout.

-h: se mostrará un resumen de las opciones disponibles.

Apéndice A.- Modificaciones respecto al SDL original.

Este traductor tiene algunas características nuevas que no posee el lenguaje SDL y que se han añadido por motivos de facilidad de uso. También tiene restricciones que impiden el uso de algunas características del SDL estándar.

Veremos a continuación las diferencias entre el lenguaje SDL estándar (al que llamaremos SDLs) y el lenguaje SDL implementado por este traductor (al que llamaremos SDLi).

- Los tipos de datos abstractos y los generadores (tipos de datos parametrizados) no han sido implementados, debido a la dificultad que entraña dicha implementación y a la poca necesidad de los mismos en las tareas a que se dedicará este traductor.

- Las subestructuras de canal y de bloque no están implementadas. Es relativamente simple introducir las segundas, y puede que se haga en próximas versiones. Las segundas complicarían las tareas de conexión, aunque su implementación también está en estudio.

- Los procesos deben definirse de forma completa: no se pueden dividir en servicios. Esta es otra característica que puede ser implementada próximamente.

- Una declaración de vía de señales debe encontrar las jerarquías (bloques o unidades) que intervienen en ella ya instaladas en la estructura (declaradas o definidas). Es decir, una construcción del tipo

```
SIGNALROUTE sr FROM pr1 TO pr2 WITH ...
```

debe encontrar los procesos pr1 y pr2 instalados en la jerarquía. Esto puede ser porque hayan sido previamente definidos o se haya usado la construcción

```
PROCESS prx REFERENCED;
```

Esta restricción simplifica los procesos de análisis y mejora la semántica del sistema, ya que no produce referencias implícitas, bastante desorientadoras.

- Siguiendo con la línea de evitar referencias implícitas, todos los tipos de datos deben definirse antes de su primera utilización (por ejemplo, como argumentos de una señal en una construcción SIGNAL).

- Una construcción CONNECT debe encontrar las vías que conecta ya definidas.

- El preprocesador LEDPP no permite usar una macro antes de su definición. Eso requeriría una segunda pasada.

- Los parámetros de tipo IN/OUT no están implementados, aunque se está desarrollando un método para incluirlos en SDLi.

- La semántica de las declaraciones EXPORTED-REVEALED no está muy clara en SDLs, así que una declaración de ese tipo se considerará como EXPORTED.

- Las construcciones SYNONYM ... = EXTERNAL no son necesarias, ya que los parámetros que deban ser introducidos desde fuera del sistema se pueden pedir a través de la pseudounidad ENV.

- Las construcciones SELECT IF no pertenecen a SDLi.

- Las construcciones SYNTYPE no aceptan valores por defecto (DEFAULT) ni especificación de rango de valores permitidos (CONSTANTS).

- Los tipos de datos enumerados (LITERALS) se implementan considerándolos enteros y definiendo constantes para los elementos de la enumeración. Por lo tanto, las constantes enumeradas no pueden coincidir con literales ya existentes.

- Las declaraciones de conjuntos de señales admisibles (SIGNALSET) se ignoran.

- Según SDLs, cuando una señal circula por un canal, sufre un retraso no determinista. En SDLi ese retraso no existe: las señales llegan a la cola de destino inmediatamente después de ser enviadas.

- Las variables reveladas y exportadas tienen un espacio de nombres único en el sistema. Esto implica que no puede haber dos variables externas con el mismo nombre en un sistema SDLi.

- No se implementan los valores de tipo estructura (listas de valores entre los marcadores '(.' y '.)'.

- Ciertas características internas obligan a declarar todas las variables de una unidad antes de que otra unidad sea definida (definida, no declarada) de forma anidada. Así, el código

```
PROCESS p;
     DCL a INTEGER;

     PROCEDURE q;
          ...
     ENDPROCEDURE q;

     DCL x REAL;

     ...
```

es incorrecto. La declaración de la variable real x ha de hacerse antes de la definición del procedimiento q.

- La semántica de la construcción ALTERNATIVE es idéntica a la de las decisiones (DECISION).

- No se admite texto informal en decisiones.

- Debido a la utilización del generador de analizadores léxicos lex, los ficheros de entrada sólo podrán contener caracteres ASCII con códigos menores que 128. Eso elimina la posibilidad de usar caracteres nacionales en los programas SDL. El uso de cualquier carácter superior a 127 situará al analizador léxico en un bucle infinito.

- Los tipos NATURAL e INTEGER son sinónimos.

- Se ha añadido a la gramática una ampliación que permite definir una pseudounidad llamada ENV que modelará el entorno. Su definición es idéntica a la de un proceso, pero se han de sustituir las construcciones PROCESS <nombre> y ENDPROCESS <nombre> por ENVIRONMENT y ENDENVIRONMENT.

- Para permitir la comunicación interactiva con el usuario se han implementado dos nuevas instrucciones: Leer y Escribir. La instrucción Leer acepta una expresión de la forma adecuada (un LValue, véase discusión en el esquema de traducción) y permite introducir por el teclado el valor que se asignará. Sólo se pueden leer expresiones de ciertos tipos básicos: BOOLEAN, CHARACTER, CHARSTRING, INTEGER y REAL. La instrucción Escribir acepta una expresión e imprime su valor. La expresión puede ser de tipo BOOLEAN, CHARACTER, CHARSTRING, INTEGER, REAL, TIME o DURATION.

- Según SDLs, los tipos DURATION y TIME tienen como literales los números reales. Implementar esta dualidad de literales era complicado, así que se optó por crear dos nuevas funciones que cambien el tipo de un literal (o expresión) real para hacerlo de tipo duración o tiempo. La función CAST_DURATION acepta una expresión de tipo REAL y devuelve un valor de tipo DURATION. La función CAST_TIME hace lo mismo para el tipo TIME.

- Según SDLs, si se usa una variable antes de que le sea asignado un valor, se debe producir un error dinámico. Pero algunas herramientas como Geode no trabajan así, y consideran que una variable no iniciada tiene valor nulo. En el sistema equivalente se incluye un modo de funcionamiento (el modo Geode) que permite usar variables sin iniciar, pero solamente en comparaciones. Para activar el modo Geode hay que indicar la opción -G en la línea de comandos del sistema equivalente.

- En SDLs, el valor SENDER ha de ser el PID de la señal que se ha consumido en último lugar. En las señales contínuas y en las transiciones espontáneas, este valor es el del PID de la unidad (SELF). Sin embargo, Geode no lo hace así: en una señal contínua o transición espontánea, SENDER sigue siendo el PID de la última señal usada. El modo Geode también tiene esta característica en cuenta.

Apéndice B.- Gramática.

En este apéndide se dará la gramática que se ha utilizado para generar, mediante yacc, el analizador sintáctico del traductor LEDAC. Se presentará esta gramática en forma BNF porque es la aceptada por yacc, aunque las referencias empleadas ([BEL91] y [CCI88]) la daban en EBNF y diagramas de Conway, respectivamente.

La referencia principal para el diseño de la gramática ha sido [BEL91]. Sobre la gramática propuesta en este libro (muy redundante) se han hecho una serie de modificaciones que la han simplificado. Para pasar de la forma EBNF a BNF se han usado producciones intermedias cuyos nombres siguen unas normas:

- para simular una construcción EBNF de la forma

```
{xxxx}*
```

se han usado producciones

```
cn_xxxx:
      | cn_xxxx xxxx
      ;
```

- para simular construcciones

```
{xxxx}+
```

se han creado reglas de la forma

```
un_xxxx: xxxx
      | un_xxxx xxxx
      ;
```

- las construcciones opcionales

```
[xxxx]
```

se han sustituído por producciones

```
cu_xxxx:
      | xxxx
      ;
```

Como se ve, la norma general es anteponer un prefijo de dos letras al nombre de la producción. La primera letra indica el número mínimo de apariciones. La segunda, el máximo. Así, cu_expr (equivalente al EBNF [expr]) indica cero o una apariciones del no-terminal expr; cn_parte_estado ({parte_estado}*) indica de cero a n apariciones de parte_estado; un_parte_respuesta indica una o más apariciones de parte_respuesta. Las construcciones EBNF que requieren que sus elementos estén separados por comas o puntos y comas se implementan de forma similar, y al prefijo se le añade 'c' o 'pc'. Así, una lista de uno o más identificadores separados por comas, que en EBNF se representaría

```
ident {, ident}*
```

se implementará mediante la producción

```
unc_ident: ident
     | unc_ident ',' ident
     ;
```

En todos los casos se ha optado por la recursividad izquierda, ya que sobrecarga menos la pila del analizador.

A continuación se lista la gramática usada por el traductor.

```
def_sist: SYSTEM NOMBRE fin un_parte_sist
     ENDSYSTEM cu_nombre fin cn_definicion
     ;

un_parte_sist: parte_sist
     | un_parte_sist parte_sist
     ;

cu_nombre:
     | NOMBRE
     ;

parte_sist: def_bloque
     | def_canal
     | def_datos
     | def_seleccion
     | def_senal
     | def_lista_senales
     | ref_textual_bloque
```

```
      | def_pseudounidad_env
      ;

def_pseudounidad_env: ENVIRONMENT fin cu_param_formales
      cu_signalset cn_parte_proceso cuerpo_o_descomp
      ENDENVIRONMENT fin
      ;

cn_definicion:
      | cn_definicion definicion
      ;

definicion: def_bloque
      | def_subestr
      | def_proceso
      | def_procedimiento
      | def_servicio
      ;

def_canal: CHANNEL NOMBRE FROM NOMBRE TO NOMBRE
      WITH lista_senales fin cu_from_inverso
      cu_subestr ENDCHANNEL cu_nombre fin
      ;

cu_from_inverso:
      | FROM NOMBRE TO NOMBRE WITH lista_senales fin
      ;

ident: cu_cualificador NOMBRE
      ;

cu_cualificador:
      | cualificador
      ;

cualificador: item_de_ruta
      | cualificador '/' item_de_ruta
      ;

item_de_ruta: unidad_de_rango NOMBRE
      ;

unidad_de_rango: BLOCK
      | PROCEDURE
```

```
      | PROCESS
      | SERVICE
      | SIGNAL
      | SUBSTRUCTURE
      | SYSTEM
      | TYPE
      ;

cu_subestr:
      | def_subestr
      | SUBSTRUCTURE ident REFERENCED fin
      ;

def_senal: SIGNAL unc_senal_en_def fin
      ;

unc_senal_en_def: NOMBRE cu_lista_tipos cu_refinamiento_senal
      | unc_senal_en_def ','
          NOMBRE cu_lista_tipos cu_refinamiento_senal
      ;

cu_lista_tipos:
      | '(' unc_ident ')'
      ;

unc_ident: ident
      | unc_ident ',' ident
      ;

cu_refinamiento_senal:
      | REFINEMENT un_parte_refin ENDREFINEMENT
      ;

un_parte_refin: cu_reverse def_senal
      | un_parte_refin cu_reverse def_senal
      ;

cu_reverse:
      | REVERSE
      ;

def_lista_senales: SIGNALLIST NOMBRE '=' lista_senales fin
      ;
```

```
lista_senales: ident
     | '(' ident ')'
     | lista_senales ',' ident
     | lista_senales ',' '(' ident ')'
     ;

fin: ';'
     | COMMENT CAD ';'
     ;

ref_textual_bloque: BLOCK ident REFERENCED fin
     ;

def_bloque: BLOCK ident fin cn_parte_bloque
     cu_subestr ENDBLOCK cu_ident fin
     ;

cn_parte_bloque:
     | cn_parte_bloque parte_bloque
     ;

parte_bloque: conexion_vias
     | def_datos
     | def_proceso
     | def_seleccion
     | def_senal
     | def_lista_senales
     | def_ruta_senales
     | ref_textual_proceso
     ;

cu_ident:
     | ident
     ;

conexion_vias: CONNECT NOMBRE AND unc_nombre fin
     ;

def_ruta_senales: SIGNALROUTE NOMBRE FROM NOMBRE TO NOMBRE
     WITH lista_senales fin cu_from_inverso
     ;

ref_textual_proceso: PROCESS ident cu_num_instancias
     REFERENCED fin
```

```
	;

def_proceso: PROCESS ident cu_num_instancias fin
	cu_param_formales cu_signalset cn_parte_proceso
	cuerpo_o_descomp ENDPROCESS cu_ident fin
	;

cu_num_instancias:
	| '(' cu_expr ',' cu_expr ')'
	;

cu_expr:
	| expr
	;

cu_param_formales:
	| FPAR unc_nombres_tipo fin
	;

unc_nombres_tipo: unc_nombre ident
	| unc_nombres_tipo ',' unc_nombre ident
	;

unc_nombre: NOMBRE
	| unc_nombre ',' NOMBRE
	;

cu_signalset:
	| SIGNALSET fin
	| SIGNALSET lista_senales fin
	;

cn_parte_proceso:
	| cn_parte_proceso parte_proceso
	;

parte_proceso: def_datos
	| def_import
	| def_procedimiento
	| def_seleccion
	| def_senal
	| def_lista_senales
	| ref_textual_procedimiento
	| def_timer
```

```
      | def_var
      | def_vista
      ;

cuerpo_o_descomp: cuerpo_proceso
      | descomp_en_servicios
      ;

def_import: IMPORTED unc_nombres_tipo fin
      ;

ref_textual_procedimiento: PROCEDURE ident REFERENCED fin
      ;

def_timer: TIMER unc_temporizador fin
      ;

unc_temporizador: NOMBRE cu_lista_tipos
      | unc_temporizador ',' NOMBRE cu_lista_tipos
      ;

def_var: tok_dcl unc_decl_var fin
      ;

tok_dcl: DCL
      | DCL '-' EXPORTED
      | DCL '-' REVEALED
      | DCL '-' REVEALED '-' EXPORTED
      | DCL '-' EXPORTED '-' REVEALED
      ;

unc_decl_var: unc_nombre ident cu_inicializacion
      | unc_decl_var ',' unc_nombre ident cu_inicializacion
      ;

cu_inicializacion:
      | ASIGN expr
      ;

def_vista: VIEWED unc_nombres_tipo fin
      ;

cuerpo_proceso: START fin transicion cn_estado
      ;
```

```
cn_estado:
     | cn_estado STATE lista_de_estados fin
          cn_parte_estado cu_fin_estado
     ;

cn_parte_estado:
     | cn_parte_estado parte_estado
     ;

parte_estado: PROVIDED expr fin transicion
     | PROVIDED expr fin PRIORITY NOMBRE fin transicion
     | INPUT lista_input fin cu_cond_activacion transicion
     | PRIORITY INPUT unc_estimulo fin transicion
     | SAVE lista_save fin
     | INPUT NONE fin cu_cond_activacion transicion
     ;

cu_fin_estado:
     | ENDSTATE cu_nombre fin
     ;

cu_cond_activacion:
     | PROVIDED expr fin
     ;

lista_de_estados: unc_nombre
     | '*'
     | '*' '(' unc_nombre ')'
     ;

lista_input: '*'
     | unc_estimulo
     ;

unc_estimulo: ident cu_vars_recepcion
     | unc_estimulo ',' ident cu_vars_recepcion
     ;

cu_vars_recepcion:
     | '(' unc_cu_ident ')'
     ;

unc_cu_ident: cu_ident
```

```
     | unc_cu_ident ',' cu_ident
     ;

lista_save: lista_senales
     | '*'
     ;

transicion: cadena_transicion cu_terminador
     | terminador
     ;

cadena_transicion: cu_etiq parte_cadena_transicion fin
     | cadena_transicion cu_etiq parte_cadena_transicion fin
     ;

cu_etiq:
     | NOMBRE ':'
     ;

parte_cadena_transicion: CREATE ident cu_param_reales
     | CALL ident cu_param_reales
     | EXPORT '(' unc_ident ')'
     | OUTPUT unc_salida cu_destino cu_via
     | PRIORITY OUTPUT unc_salida
     | RESET '(' unc_parte_reset ')'
     | SET '(' unc_parte_set ')'
     | TASK cuerpo_tarea
     | decision_no_determinista
     | decision
     | opcion_transicion
     | LEER '(' parte_izq ')'
     | ESCRIBIR '(' expr ')'
     ;

cu_terminador:
     | terminador
     ;

terminador: cu_etiq term fin
     ;

term: JOIN NOMBRE
     | NEXTSTATE NOMBRE
     | NEXTSTATE '-'
```

```
      | RETURN
      | STOP
      ;

cu_param_reales:
      | '(' unc_cu_expr ')'
      ;

unc_cu_expr: cu_expr
      | unc_cu_expr ',' cu_expr
      ;

unc_salida: ident cu_param_reales
      | unc_salida ',' ident cu_param_reales
      ;

cu_destino:
      | TO expr
      ;

cu_via:
      | VIA unc_nombre
      ;

unc_parte_reset: ident cu_args_set_o_reset
      | unc_parte_reset ',' ident cu_args_set_o_reset
      ;

cu_args_set_o_reset:
      | '(' unc_expr ')'
      ;

unc_expr: expr
      | unc_expr ',' expr
      ;

unc_parte_set: expr ',' ident cu_args_set_o_reset
      | unc_parte_set ',' expr ',' ident cu_args_set_o_reset
      ;

cuerpo_tarea: unc_asignacion
      | unc_cad
      ;
```

```
unc_asignacion: parte_izq ASIGN expr
     | unc_asignacion ',' parte_izq ASIGN expr
     ;

unc_cad: CAD
     | unc_cad ',' CAD
     ;

parte_izq: ident
     | parte_izq '(' unc_expr ')'
     | parte_izq '!' NOMBRE
     | parte_izq '!' '(' unc_nombre ')'
     ;

decision: DECISION cuestion fin un_parte_respuesta
     cu_parte_else ENDDECISION
     ;

un_parte_respuesta: '(' rango_o_texto ')' ':' cu_transicion
     | un_parte_respuesta '(' rango_o_texto ')' ':'
          cu_transicion
     ;

rango_o_texto: condic_rango
     | CAD
     ;

cu_transicion:
     | transicion
     ;

cu_parte_else:
     | ELSE ':' cu_transicion
     ;

cuestion: expr
     | CAD
     ;

decision_no_determinista: DECISION ANY fin
     un_parte_dec_no_det ENDDECISION
     ;

un_parte_dec_no_det: '(' ')' ':' cu_transicion
```

```
     | un_parte_dec_no_det '(' ')' ':' cu_transicion
     ;

def_procedimiento: PROCEDURE ident fin
     cu_param_formales_procedimiento cn_parte_procedimiento
     cuerpo_proceso ENDPROCEDURE cu_ident fin
     ;

cu_param_formales_procedimiento:
     | FPAR unc_param_formal_proced fin
     ;

unc_param_formal_proced: tipo_par unc_nombre ident
     | unc_param_formal_proced ',' tipo_par unc_nombre ident
     ;

tipo_par:
     | INOUT
     | IN
     ;

cn_parte_procedimiento:
     | cn_parte_procedimiento parte_procedimiento
     ;

parte_procedimiento: def_datos
     | def_procedimiento
     | def_seleccion
     | ref_textual_procedimiento
     | def_var
     ;

def_subestr: SUBSTRUCTURE ident fin un_parte_subestr
     ENDSUBSTRUCTURE cu_ident fin
     ;

un_parte_subestr: parte_subestr
     | un_parte_subestr parte_subestr
     ;

parte_subestr: def_bloque
     | conexion_vias
     | def_canal
     | def_datos
```

```
        | def_seleccion
        | def_senal
        | def_lista_senales
        | ref_textual_bloque
        ;

def_seleccion: SELECT IF '(' expr ')' fin un_parte_seleccion
        ENDSELECT fin
        ;

un_parte_seleccion: parte_seleccion
        | un_parte_seleccion parte_seleccion
        ;

parte_seleccion: def_bloque
        | def_canal
        | conexion_vias
        | def_datos
        | def_import
        | def_procedimiento
        | def_proceso
        | def_seleccion
        | def_servicio
        | def_senal
        | def_lista_senales
        | def_ruta_senales
        | ref_textual_bloque
        | ref_textual_procedimiento
        | ref_textual_proceso
        | ref_textual_servicio
        | def_timer
        | def_var
        | def_vista
        ;

opcion_transicion: ALTERNATIVE cuestion fin un_parte_respuesta
        cu_parte_else ENDALTERNATIVE
        ;

descomp_en_servicios: parte_descomp
        | descomp_en_servicios parte_descomp
        ;

parte_descomp: def_seleccion
```

```
      | def_servicio
      | def_ruta_senales
      | conexion_vias
      | ref_textual_servicio
      ;

ref_textual_servicio: SERVICE ident REFERENCED fin
      ;

def_servicio: SERVICE ident fin cu_signalset cn_parte_proceso
      cuerpo_proceso ENDSERVICE cu_ident fin
      ;

def_datos: def_parcial_tipo fin
      | def_syntype fin
      | SYNONYM NOMBRE cu_ident '=' expr fin
      | SYNONYM NOMBRE cu_ident '=' EXTERNAL fin
      | def_generador fin
      ;

def_parcial_tipo: NEWTYPE NOMBRE alternativa_tipo
      cu_literales_oper cu_operators_oper cu_axiomas
      cu_mapa_literales cu_default ENDNEWTYPE cu_nombre
      ;

alternativa_tipo:
      | regla_de_herencia
      | un_instanciacion_generador
      | STRUCT unpc_nombres_tipo cu_fin cu_adding
      ;

unpc_nombres_tipo: unc_nombre ident
      | unpc_nombres_tipo ';' unc_nombre ident
      ;

cu_literales_oper:
      | LITERALS unc_signat_literal cu_fin
      ;

unc_signat_literal: signat_literal
      | unc_signat_literal ',' signat_literal
      ;

cu_operators_oper:
```

```
        | OPERATORS unf_signat_operador cu_fin
        ;

unf_signat_operador: signat_operador
        | unf_signat_operador fin signat_operador
        ;

cu_axiomas:
        | AXIOMS axiomas
        ;

cu_mapa_literales:
        | MAP unf_ecuacion_literal cu_fin
        ;

unf_ecuacion_literal: ecuacion_literal
        | unf_ecuacion_literal fin ecuacion_literal
        ;

cu_default:
        | DEFAULT expr cu_fin
        ;

un_instanciacion_generador: inst_gen_fin_add
        | un_instanciacion_generador inst_gen_fin_add
        ;

inst_gen_fin_add: ident '(' unc_param_real_gen ')' cu_fin
        cu_adding
        ;

unc_param_real_gen: param_real_gen
        | unc_param_real_gen ',' param_real_gen
        ;

cu_fin:
        | fin
        ;

cu_adding:
        | ADDING
        ;

regla_de_herencia: INHERITS ident cu_literales_her
```

```
     cu_opers_her cu_adding
     ;

cu_literales_her:
     | LITERALS unc_par_renombr_literal fin
     ;

unc_par_renombr_literal: signat_renombrado_literal '='
          signat_renombrado_literal
     | unc_par_renombr_literal ',' signat_renombrado_literal
          '=' signat_renombrado_literal
     ;

cu_opers_her:
     | cu_operators todos_o_lista cu_adding
     ;

cu_operators:
     | OPERATORS
     ;

todos_o_lista: ALL
     | '(' lista_herencia ')'
     ;

signat_renombrado_literal: NOMBRE
     | CAD
     ;

lista_herencia: nombre_oper
     | nombre_oper '=' nombre_oper
     | lista_herencia ',' nombre_oper
     | lista_herencia ',' nombre_oper '=' nombre_oper
     ;

signat_literal: NOMBRE
     | CAD
     | NAMECLASS expr_regular
     ;

expr_regular: elem_regular cu_modificador
     | expr_regular cu_or elem_regular cu_modificador
     ;
```

```
cu_modificador: NOMBRE
     | '+'
     | '*'
     ;

cu_or:
     | OR
     ;

elem_regular: '(' expr_regular ')'
     | CAD
     | CAD ':' CAD
     ;

signat_operador: nombre_oper ':' unc_ident IMPL ident
     | ORDERING
     ;

nombre_oper: NOMBRE
     | NOMBRE '!'
     | oper_entrecom
     ;

oper_entrecom: '\"' oper_infijo '\"'
     | '\"' NOT '\"'
     ;

oper_infijo: DOBLE_IMPL_1
     | OR
     | XOR
     | AND
     | IN
     | DIST
     | '='
     | '>'
     | '<'
     | MENIG
     | MAYIG
     | '+'
     | '-'
     | '/'
     | '*'
     | CONCAT
     | MOD
```

```
      | REM
      ;

axiomas: ecuacion cu_fin
      | axiomas fin ecuacion cu_fin
      ;

ecuacion: termino '=' '=' termino
      | termino
      | FOR ALL unc_nombre IN ident '(' axiomas ')'
      | unc_restriccion DOBLE_IMPL_2 restriccion
      | CAD
      ;

unc_restriccion: restriccion
      | unc_restriccion ',' restriccion
      ;

termino: term_de_base_o_comp
      | ERROR '!'
      | SPELLING '(' ident ')'
      ;

term_de_base_o_comp: ident
      | cu_cualificador CAD
      | ident '(' unc_term_de_base_o_comp ')'
      | '(' term_de_base_o_comp ')'
      | ident_oper_ext '(' unc_term_de_base_o_comp ')'
      | termino '*' termino
      | termino '+' termino
      | termino '-' termino
      | termino '/' termino
      | termino '<' termino
      | termino '=' termino
      | termino '>' termino
      | termino AND termino
      | termino CONCAT termino
      | termino DIST termino
      | termino DOBLE_IMPL_1 termino
      | termino IN termino
      | termino MAYIG termino
      | termino MENIG termino
      | termino MOD termino
      | termino OR termino
```

```
     | termino REM termino
     | termino XOR termino
     | '-' term_de_base_o_comp  %prec MENOS_UNARIO
     | NOT term_de_base_o_comp  %prec MENOS_UNARIO
     | IF termino THEN termino ELSE termino FI
     ;

unc_term_de_base_o_comp: term_de_base_o_comp
     | unc_term_de_base_o_comp ',' term_de_base_o_comp
     ;

ident_oper_ext: ident '!'
     | NOMBRE
     | cu_cualificador oper_entrecom
     ;

restriccion: termino '=' '=' termino
     | termino
     ;

ecuacion_literal: FOR ALL unc_nombre IN ident LITERALS
     '(' unf_e_o_el cu_fin ')'
     ;

unf_e_o_el: e_o_el
     | unf_e_o_el fin e_o_el
     ;

e_o_el: ecuacion
     | ecuacion_literal
     ;

def_syntype: SYNTYPE NOMBRE '=' ident cu_default cu_constants
          ENDSYNTYPE cu_nombre
     | NEWTYPE NOMBRE alternativa_tipo cu_literales_oper
          cu_operators_oper cu_axiomas cu_mapa_literales
          cu_default CONSTANTS condic_rango
          ENDNEWTYPE cu_nombre
     ;

cu_constants:
     | CONSTANTS condic_rango
     ;
```

```
condic_rango: rc_o_ra
     | condic_rango ',' rc_o_ra
     ;

rc_o_ra: expr ':' expr
     | expr
     | op_relac expr
     ;

op_relac: '='
     | DIST
     | '>'
     | '<'
     | MENIG
     | MAYIG
     ;

def_generador: GENERATOR NOMBRE '(' unc_param_gener ')'
     cn_instanciacion_gener cu_literales_oper
     cu_operators_oper cu_axiomas cu_mapa_literales cu_default
     ENDGENERATOR cu_nombre
     ;

unc_param_gener: tipo_param_gener unc_nombre
     | unc_param_gener tipo_param_gener unc_nombre
     ;

tipo_param_gener: TYPE
     | LITERAL
     | OPERATOR
     | CONSTANT
     ;

cn_instanciacion_gener:
     | cn_instanciacion_gener inst_gen_fin_add
     ;

param_real_gen: signat_literal
     | nombre_oper
     | term_de_base_o_comp
     ;

expr: expr DOBLE_IMPL_1 expr
     | expr OR expr
```

```
| expr XOR expr
| expr AND expr
| expr '=' expr
| expr DIST expr
| expr '>' expr
| expr MAYIG expr
| expr '<' expr
| expr MENIG expr
| expr IN expr
| expr '+' expr
| expr '-' expr
| expr CONCAT expr
| expr '*' expr
| expr '/' expr
| expr MOD expr
| expr REM expr
| '-' expr  %prec MENOS_UNARIO {...}
| NOT expr  %prec MENOS_UNARIO {...}
| NOMBRE
| CAD
| '\"' oper_infijo '\"' '(' unc_expr ')'
| FUNCION '(' unc_expr ')'
| expr '(' unc_expr ')'
| IF expr THEN expr ELSE expr FI
| expr '!' NOMBRE
| '(' expr ')'
| ACTIVE '(' ident ')'
| ACTIVE '(' ident '(' unc_expr ')' ')'
| ANY '(' ident ')'
| IMPORT '(' ident ')'
| IMPORT '(' ident ',' expr ')'
| NOW
| OFFSPRING
| PARENT
| SELF
| SENDER
| VIEW '(' ident ')'
| VIEW '(' ident '(' expr ')' ')'
;
```

Apéndice C.- Expansión.

El sistema se ha diseñado para permitir la adición sencilla de nuevas funciones e instrucciones. En el código hay marcas que indican dónde se han de introducir las modificaciones.

C.1.- Adición de nuevas instrucciones.

Una nueva instrucción (con o sin parámetros) puede ser añadida en cualquier momento al sistema traductor. Para ello, hay que añadir algunas modificaciones en el código en los lugares marcados con el identificador $ADD_INS.

Como ejemplo, supondremos que se desea añadir a la implementación de SDL una nueva intrucción llamada INSTR que debe aceptar dos argumentos entre paréntesis y separados por una coma. El primer argumento debe ser de tipo real y el segundo de tipo entero.

Los pasos que hay que seguir son los siguientes:

- Modificar el fichero structs.h, añadiendo una instrucción de la forma

```
#define TIN_XXXX  YY
```

(donde XXXX es el identificador de la nueva instrucción, e YY es una constante entera que debe ser distinta de las ya utilizadas por las constantes simbólicas TIN_...) y añadiendo un elemento a la unión <datos> de la estructura de tipo INSTRUCCIONES. Este nuevo elemento deberá poder contener los parámetros que necesite la nueva instrucción. En caso de que ésta no necesite parámetros, no tiene por qué añadirse el elemento.

En el ejemplo que se está tratando, se añadiría la línea

```
#define TIN_INSTR 20
```

delante del primer marcador $ADD_INS, y la declaración

```
struct
     {
     EXPR *e1, *e2;
     } expresiones;      /* TIN_INSTR */
```

delante del segundo.

- Declarar el símbolo terminal asociado a la nueva instrucción en el fichero parser.y, con la instrucción yacc %token:

```
%token INSTR
```

- Añadir el símbolo terminal y su representación textual a la lista de palabras reservadas en el fichero scanner.l:

```
{ "INSTR",          INSTR },
```

teniendo siempre en cuenta que la lista debe quedar ordenada alfabéticamente.

- Modificar la regla de la gramática correspondiente al símbolo no-terminal 'parte_cadena_transicion', en el fichero parser.y. Hay que añadir a esta regla una parte izquierda que describa la sintaxis de la instrucción que se desea añadir. Así, para nuestro ejemplo, la regla a añadir sería:

```
| INSTR '(' expr ',' expr ')'
```

Además de la regla gramatical, hay que añadir una acción semántica para crear la estructura de representación correspondiente y, si se desea, chequear la validez semántica de la instrucción. En el caso que estamos tratando, la acción semántica sería:

```
{
if( $3->t->tipo!=TIPO_REAL )
Error( FICH, NOFATAL,
"INSTR: primer argumento ha de ser REAL." );

if( $5->t->tipo!=TIPO_INTEGER )
Error( FICH, NOFATAL,
"INSTR: segundo argumento ha de ser INTEGER." );

$$= NEW(INSTRUCCIONES);
$$->tipo= TIN_INSTR;
$$->datos.expresiones.e1= $3;
$$->datos.expresiones.e2= $5;
$$->sig= NULL;
}
```

Como se ve, se ha de chequear la semántica, crear la estructura INSTRUCCIONES e iniciarla.

- Añadir nuevo código a la función GeneraInstruccion() en el fichero genera.c. Se debe insertar una etiqueta case y el código para escribir la nueva instrucción en el fichero _uniddes.c:

```
case TIN_INSTR:
gprintf( uniddes_c, gnivel, "Instr(" );
GeneraExpr( ins->datos.expresiones.e1 );
gprintf( uniddes_c, 0, ", " );
GeneraExpr( ins->datos.expresiones.e2 );
gprintf( uniddes_c, 0, ");\n" );
break;
```

- Modificar la función ListaInstrucciones() en el fichero listasis.c, para que el switch acepte la nueva instrucción:

```
case TIN_INSTR:
iprintf( "Instr(...)\n" );
break;
```

- Crear la función equivalente C en el fichero acciones.c del sistema equivalente. Debe tenerse en cuenta el tipo de argumentos que aceptará la función equivalente: en nuestro ejemplo, y como se ha visto en la generación de código, la función Instr() aceptará dos VALOR *. La función equivalente quedará:

```
void Instr( vr, vi )
VALOR *vr, *vi;
{
...
/* código para implementar la instrucción
INSTR, usando vr->dato.vreal y
vi->dato.vinteger */
...
LiquidaValor(vr);
LiquidaValor(vi);
/* ver discusión sobre expresiones en
el esquema de traducción */
}
```

- Añadir el prototipo de esta función al fichero acciones.h del sistema equivalente:

```
PRTFUN( void, Instr, VALOR * COMA VALOR * );
```

Una vez ejecutados estos pasos, debe recompilarse el traductor y la nueva instrucción podrá ser usada.

C.2.- Adición de nuevas funciones.

Para ilustrar el protocolo de adición de nuevas funciones al lenguaje implementado, supondremos que se desea introducir una función FUNC que acepta dos argumentos, de tipos lógico y carácter, y retorna un valor de tipo entero. Se deben hacer las siguientes operaciones:

- Añadir en structs.h una línea que defina la constante simbólica asociada a la nueva función. Esta línea será de la forma

```
#define FUN_XXXX YY
```

(donde XXXX es el nombre de la nueva función e YY es su identificador entero, que no debe estar duplicado en las constantes FUN_...). En el ejemplo que se está proponiendo, la línea a añadir antes del marcador $ADD_FUN será:

```
#define FUN_FUNC 14
```

- Permitir que el analizador léxico conozca el nuevo nombre: se deberán incluir en el fichero scanner.l un patrón que reconozca el nombre de la función y una acción semántica que retorne al analizador sintáctico el valor adecuado. La línea a añadir a scanner.l tendrá el formato

```
<NRM>ZZZZ           {
yylval.byte= FUN_XXXX;
LEXRETURN(FUNCION);
}
```

donde XXXX es el nombre de la función y ZZZZ es el patrón que reconocerá dicho nombre en mayúsculas o minúsculas. Cada carácter alfabético C de ZZZZ debe ser sustituído por una llamada a macro lex {C}. Así, la línea que se debe añadir en el ejemplo que se ha propuesto será

```
<NRM>{f}{u}{n}{c}   {
yylval.byte= FUN_FUNC;
LEXRETURN(FUNCION);
```

```
}
```

- Modificar el fichero fuente exprs.c para que la función Funcion() acepte el nuevo elemento. Debe añadirse una etiqueta case seguida de una comprobación del número y tipo de argumentos de la función y de la asignación del tipo retornado. En el ejemplo propuesto, el código a añadir es:

```
case FUN_FUNC:
if( !NArgsOk( args, 2 ) )
{
free(e);
return(expr_dummy);
}

if( ARG_1->t->tipo!=TIPO_BOOLEAN )
{
Error( FICH, NOFATAL,
"FUNC requiere primer argumento BOOLEAN." );
free(e);
return(expr_dummy);
}

if( ARG_2->t->tipo!=TIPO_CHARACTER )
{
Error( FICH, NOFATAL,
"FUNC requiere segundo argumento CHARACTER." );
free(e);
return(expr_dummy);
}

e->t= t_basicos[TIPO_INTEGER];

break;
```

- Modificar genera.c para que se genere el código equivalente a la función añadida. En la función GeneraExpr() hay que incluir código de la forma

```
case FUN_XXXX:
gprintf( uniddes_c, 0, "TTTT(" );
break;
```

donde XXXX es el nombre de la función y TTTT es el nombre de la función equivalente C. En el ejemplo propuesto, se añadirá lo siguiente:

```
case FUN_FUNC:
gprintf( uniddes_c, 0, "Func(" );
break;
```

- Hacer que la función ListaExpr() del fichero listasis.c acepte la nueva función. Para ello, se ha de añadir una etiqueta case y código de la forma

```
case FUN_XXXX:
printf( "XXXX" );
break;
```

lo que en nuestro ejemplo equivale a

```
case FUN_FUNC:
printf( "func" );
break;
```

- Crear la función equivalente en exprs.c (el fichero del sistema equivalente). El esqueleto de dicha función es el siguiente:

```
VALOR *_Xxxx( v1, v2, ... , t)
VALOR *v1, *v2, ...;
BYTE t;
{
VALOR *v;

v= NuevoValor();

v->tipo= <tipo de retorno>;

<código de la función>

if( t==EXTERNA )
{
LiquidaValor(v1);
LiquidaValor(v2);
...
}
```

```
return(v);
}
```

Obsérvese que el nombre de la función equivalente debe comenzar por un carácter subrayado ('_'). La causa de este cambio de nombres está explicada en el esquema de traducción. Como se ve, hay que llamar a la función LiquidaValor() para cada parámetro de tipo VALOR * que haya recibido la función.

En el ejemplo propuesto, la función equivalente será

```
VALOR *_Func( vb, vc, t)
VALOR *vb, *vc;
BYTE t;
{
VALOR *v;

v= NuevoValor();

v->tipo= TVL_INTEGER;

<código de la función>

if( t==EXTERNA )
{
LiquidaValor(vb);
LiquidaValor(vc);
}

return(v);
}
```

- Incluir el prototipo de la función equivalente en el fichero exprs.h del sistema equivalente:

```
PRTFUN( VALOR *, _Func, VALOR * COMA VALOR * );
```

y precederlo de la macro usada para las llamadas externas a la función equivalente (ver sección dedicada a las expresiones en el esquema de traducción):

```
#define Func(vb,vc) _Func((vb),(vc),EXTERNA)
```

Por último, se debe recompilar el traductor y la nueva función estará lista para ser utilizada.

Apéndice D.- Bibliografía.

[ABR88]: Abraxas Software Inc, "**Compiler construction on personal computers (with PCYACC)**". 1988, Abraxas Software.

El manual de PCYACC. Poco técnico en algunos aspectos. Deja poco claro el manejo de errores sintácticos.

[AHO77]: Aho, Alfred V. y Ullman, Jeffrey D, "**Principles of compiler design**". 1977, Addison-Wesley.

Algo antiguo, aunque los métodos de análisis sintáctico sigan siendo los mismos. Muy extenso en análisis de flujo y optiminzación.

[ANS89]: ANSI, "**American National Standard for Information Systems -- Programming Language -- C, ANSI X3.159-1989**". 1989, ANSI.

El estándar ANSI de C.

[BEL91]: Belina, Ferenc et al., "**SDL with applications from protocol specification**". 1991, Prentice-Hall.

Un buen libro sobre SDL, con una gramática del lenguaje en EBNF y algunos errores en la misma.

[CCI88]: CCITT, "**CCITT Recommendation Z.100: Specification and Description Language SDL**", Blue book, Vol. X.1-X.5. 1988, ITU General Secretariat.

El estándar SDL. Incluye la gramática del lenguaje en forma de diagramas de Conway.

[HOL90]: Holub, Allen I, "**Compiler design in C**". 1990, Prentice-Hall.

Magnífica encuadernación y perfecta impresión. Buena mezcla entre teoría y práctica, explicaciones fácilmente comprensibles y extensión. Sin embargo, se dice que hay bastantes erratas y errores en él.

[IND90]: Cannon, L.W. et al., "**Recommended C Style and Coding Standards**". 1990, dominio público.

Este documento es una versión actualizada y ligeramente modificada del estándar de codificación Indian Hill, de los Laboratorios Bell. Este último está también en el dominio público, y su URL es:

ftp:cs.washington.edu//~ftp/pub/cstyle.tar.Z

[KYR88]: Kernigham, Brian W. y Ritchie, Dennis M, "**The C programming language. 2nd. Ed.**". 1988, Prentice-Hall.

La edición ampliada del 'Informe C'. Incluye una versión reducida del estándar ANSI en el apéndice A.

[PYS88]: Pyster, Arthur B, "**Compiler design and construction. Tools and techniques. 2nd. Ed.**". 1988, Van Nostrand Reinhold.

Bueno e inteligible, pero algo restringido y breve.

[SAN88]: Sanchís Llorca, F. y Galán Pascual, C., "**Compiladores e intérpretes. Teoría y construcción**". 1988, Paraninfo.

Bueno en algunos aspectos. Muy teórico. Mala composición y múltiples erratas.

[SAN91]: Eloy-Rafael Sanz, "**LCR: un lenguaje de control de robot móvil**". 1991

Un proyecto que implementó un compilador para el lenguaje LCR (cuya creación fué parte del mismo), especializado en el control de una plataforma móvil.

[TRE85]: Tremblay, Jean Paul y Sorenson, Paul G., "**The theory and practice of compiler writing**". 1985, McGraw-Hill.

Especialmente bueno el debate sobre diseño de lenguajes y estructuras de control.