

Diseño.

1.- El compilador.

El compilador LCR está constituido por un único fichero ejecutable, llamado LCR.EXE y ha sido programado en lenguaje C, utilizando como herramientas de desarrollo las siguientes:

- Compilador optimizador C, versión 6.0 de Microsoft Corp.
- Enlazador de programas ejecutables segmentados (*segmented-executable linker*) versión 5.10, de Microsoft Corp.
- Depurador simbólico CodeView versión 3.00, de Microsoft Corp.
- Generador de analizadores léxicos PCLEX versión 2.05, de Abraxas Software Inc.
- Generador de analizadores sintácticos PCYACC versión 3.0, de Abraxas Software Inc.
- Utilidad de mantenimiento de programas NMAKE versión 1.11, de Microsoft Corp.

Para la edición del código fuente se utilizaron los editores

- SideKick versión 1.52A, de Borland International Inc.
- Editor del entorno integrado de Turbo Pascal versión 4.0, de Borland International Inc.

Y para la preparación de la documentación se usó el procesador de texto

-
- ChiWriter versión 3.02, de Horstmann Software Design Corporation.

además de su diseñador de fuentes

- Font Design version 2.52, de Horstmann Software Design Corporation.

El formato de llamada al compilador LCR es el siguiente:

```
lcr [opciones] nfich
```

donde *nfich* es el nombre del fichero de código fuente que se quiere compilar. Si especificamos la extensión, se usará la que demos. Si no la especificamos, el compilador añadirá la extensión `'lcr'` automáticamente.

Del manejo de las opciones, la creación de nombre y apertura de los ficheros que se van a generar, las inicializaciones, etcétera, se ocupa la función `main`, que tras acabar estas tareas llama a la función `yyparse`, que realiza el grueso del trabajo (la compilación). Cuando la función `yyparse` termina, `main` recupera el control, hace unas comprobaciones de última hora y finaliza.

2.- La distribución de los ficheros de desarrollo.

Para el desarrollo del compilador se ha usado una filosofía de módulos: hay una serie de ficheros de código fuente C y de cabecera que forman el programa. La reconstrucción del programa cada vez que se modifica un fichero se realiza con la utilidad `NMAKE`.

Los ficheros que se han utilizado en el desarrollo del

compilador LCR son los siguientes (los ficheros marcados con † tienen un fichero de cabecera con del mismo nombre y extensión .h asociado):

<u>Fichero:</u>	<u>Contenido:</u>
main.c	contiene todo el código destinado a tratamiento de la línea de comandos, apertura y cierre de los ficheros generados, preparación para la compilación, llamada al analizador sintáctico y comprobaciones finales.
getopt.c	el analizador de línea de comandos getopt es muy usado en UNIX. A partir de los argumentos a main (argc y argv) y de una cadena describiendo las posibles opciones, lee la línea de llamada al programa y determina qué opciones se le han pasado.
lcr.y	este es el fichero de descripción de la gramática del lenguaje LCR. Contiene, además, todas las acciones semánticas que se realizan durante la compilación. PCYACC genera el fichero lcr.c a partir de éste.
lcr_scan.l	es el fichero de descripción del analizador léxico. Es la entrada a PCLEX, que genera el fichero lcr_scan.c basándose en él.
yaccpar.c	se utiliza como esqueleto alternativo para el analizador generado por PCYACC. Yaccpar permite un mejor manejo y detección de errores que el esqueleto usado por defecto.
errores.ct	contienen las declaraciones y definiciones de las rutinas de notificación de errores usadas en el compilador.
defs.h	es un fichero de cabecera que contiene algunas

definiciones generales, entre ellas, por ejemplo, la del tipo de datos BYTE (un carácter sin signo).

- deb.cf se ocupan de declarar y definir las funciones de depuración. Así, la macro DEB(x) se evalúa a su argumento sólo si el modo de depuración está activo.
- mem.cf contienen las funciones y demás elementos necesarios para el manejo de memoria mejorado: NEW, las funciones alternativas DebMalloc y DebFree...
- tsim.cf se ocupan del tratamiento de las tablas de símbolos y de módulos.
- tipos.cf engloban las funciones que serán usadas por las acciones semánticas de construcción de tipos y declaraciones.
- cod.cf tratan las funciones de generación de código pseudo-ensamblador.
- cmv.cf contienen declaraciones y definiciones relativas a la generación de código máquina virtual.
- ctr.cf se ocupan de la generación de código C traducido.
- ctrl.cf contienen las estructuras y funciones usadas durante la generación de instrucciones de control de flujo (si-entonces-si_no, mientras, etc.).
- exp.cf declaración y definición de los elementos necesarios para manejar expresiones.

opers.ct funciones para generar el código de evaluación de
operaciones, llamadas a función, etcétera.

3.- El fichero de construcción.

El fichero de construcción o *makefile* es el texto que, con una sintaxis inteligible por el programa NMAKE, describe los pasos que han de darse para construir el fichero objetivo (LCR.EXE) a partir de los ficheros componentes. Este fichero se compone de una primera parte de definiciones y de una segunda sección compuesta por bloques de descripción que especifican el árbol de relaciones que existe entre los módulos.

3.1.- Sección de definiciones.

La sección de definiciones se encarga de crear una serie de macros que serán usadas luego en los bloques de descripción. Además, y dependiendo de la existencia o no de algunas macros que actúan como indicadores, prepara una compilación rápida, completa o de depuración.

Las opciones que se utilizan en la compilación de los módulos en C son

-c	No llama al enlazador. Sólo compila.
-Fo< <i>f-obj</i> >	Almacena el código objeto en el fichero definido por <i>f-obj</i> .
-AL	Modelo de memoria grande (<i>large</i>).
-W4	Máximo nivel de avisos.
-Za	Restricción a compatibilidad ANSI.

Y para el enlazado se usa un tamaño de pila de 12288 bytes (3000_H bytes).

En caso de que se no pida una compilación completa (la macro NMKRELEASE no está definida), se añade la opción -qc a la

compilación. Esta opción hace que la compilación sea más breve, a costa de un código objeto y ejecutable menos optimizado, tanto en tiempo como en espacio.

Si se pide una compilación para depuración (macro NMKDEBUG definida) se añaden las opciones -Od y -Zi en compilación (no optimización e información simbólica), la opción /CODEVIEW en el enlazado, la opción -n para PCYACC y la opción -l para PCLEX (ambas implican la no generación de directivas #line, que confunden a CodeView).

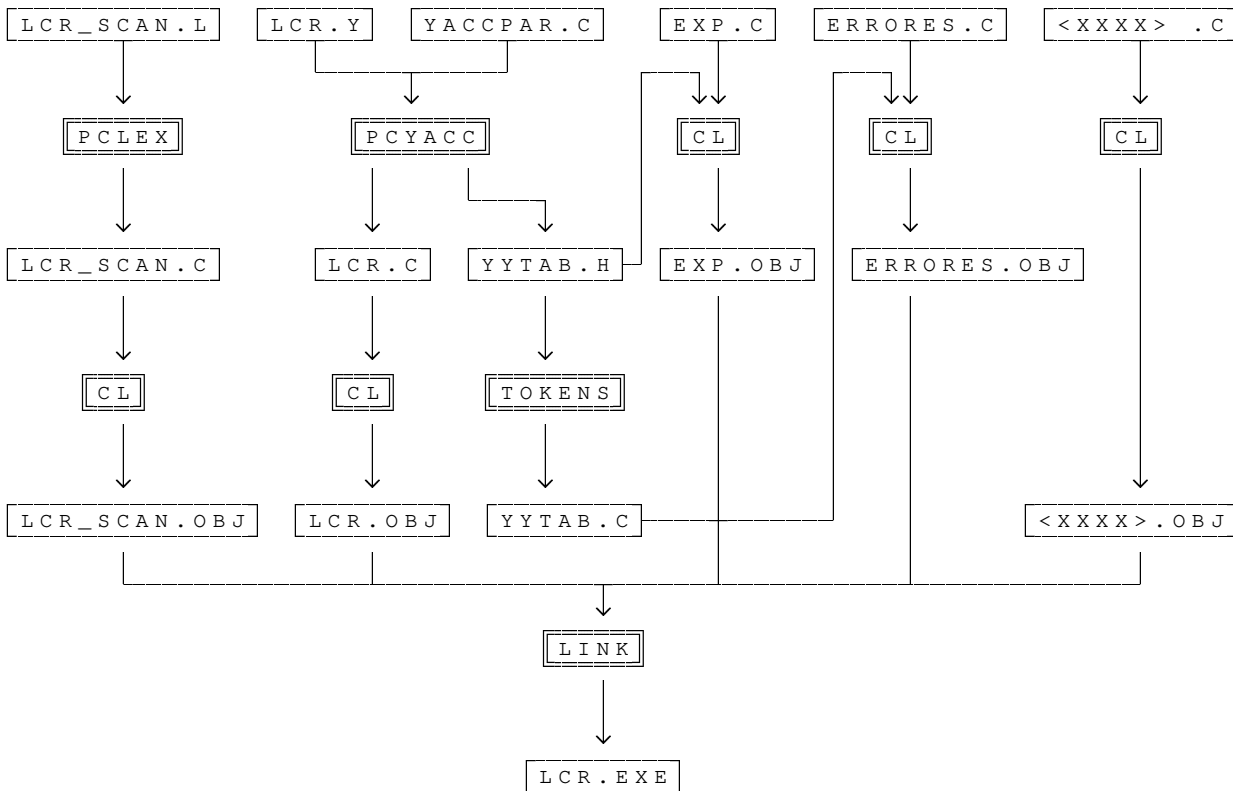
3.2.- Sección de bloques de descripción.

Esta sección está compuesta por una serie de bloques de descripción de relaciones, cada uno de los cuales tiene la siguiente estructura:

```
fich-dependiente : fichs-componentes
                  comando[s] del sistema operativo
```

La fecha del fichero dependiente se compara con las de los ficheros componentes. Si alguno de los últimos es posterior al primero, se considera que hay que reconstruirlo. Para ello se ejecutan el o los comandos que se especifican.

El árbol de dependencias de LCR.EXE se muestra a continuación. Se enmarcan con línea simple los ficheros, y con línea doble los procesos.



En el diagrama anterior, <xxxx> no representa otra cosa que el resto de los módulos que componen LCR. Es decir: main.c, tsim.c, tipos.c, exp.c,opers.c, cod.c, ctrl.c, mem.c, getopt.c, ctr.c, deb.c y cmv.c

4.- Las estructuras de datos más importantes.

El compilador LCR utiliza una gran cantidad de tipos de datos y estructuras. Vamos a revisar ahora las más importantes y su utilidad. Todas están documentadas, además, en el código.

Comenzaremos por los tipos de datos.

Nombre: Cometido:

DESCR Estructura definida en `tsim.h`. Forma una lista encadenada de elementos, cada uno de los cuales contiene un carácter y un entero. Esta lista se usa para describir el tipo de un símbolo. El carácter puede ser 'R', 'E' o 'P' (real, entero o puntero). Si es 'P', el entero indica en número de dimensiones de la matriz a que apunta el puntero (1 si es un puntero simple).

Así, el descriptor de un puntero a real es `P[1]R` y el de una matriz de `5x3` punteros a entero será `P[5]P[3]P[1]E`.

SIMBOLO Estructura definida en `tsim.h`. Describe las características de un símbolo: el módulo a que pertenece (0 si es global), su nombre, el tipo de símbolo (función, subrutina, variable, parámetro o parámetro ficticio), el tipo de almacenamiento, su tipo (mediante un descriptor), su longitud, y su posición. Además, lo enlaza con la lista de símbolos que forman sus parámetros (si es función o subrutina) o con el siguiente parámetro de dicha lista (si él mismo es un parámetro).

MODULO Definida en `tsim.h`. Sirve para describir un módulo. Contiene información sobre su nombre, su tipo (función, subrutina, programa principal o zona de datos globales), el tamaño de sus zonas para variables locales, temporales y parámetros y las posiciones que en los ficheros CMV y COD ocupan las instrucciones ENTER (para efectuar un *backpatch*).

TIPO Esta estructura, definida en `tipos.h`, se usa como paso intermedio en la generación de un nuevo

símbolo. Contiene una descripción completa de un tipo: el descriptor, el tipo de símbolo (función, subrutina, variable, parámetro o parámetro ficticio) y el modo de almacenamiento (local, global o estático).

LISTA_DIMENSIONES No es una lista encadenada, sino una estructura única que contiene el número de dimensiones de una matriz y los límites de éstas. Está definida en `tipos.h`.

LISTA_IDS Definida en `tipos.h`, es el nodo de una lista de nombres (identificadores).

LISTA_DESCRIPTORES Forma una lista de descriptores (es decir, una lista de listas, ya que un descriptor puede ser una lista). Está definida en `tipos.h`.

REFER Forma una lista de valores tipo `long` que indican las posiciones del fichero CMV en que se ha referenciado la etiqueta que esté asociada con la lista. Se usa para las operaciones de *backpatch* y está definida en `ctrl.h`.

BACKP Esta estructura, definida en `ctrl.h`, formará la tabla de *backpatches*. Contiene el nombre de una etiqueta y una lista de estructuras tipo `REFER` que indican dónde se ha mencionado la etiqueta y, por tanto, dónde han de realizarse los *backpatches*.

RESUELTA Forma una lista de etiquetas resueltas y de posiciones en el fichero CMV de dichas etiquetas.

ESTADO Esta estructura, definida en `exp.h`, describe el

estado en que se hallan los registros y la zona de variables temporales del microprocesador virtual durante la evaluación de una expresión. Contiene un array de tantos elementos como registros del microprocesador virtual, cada uno de los cuales indica el estado actual de ese registro: ocupado o libre. Contiene, además, un mapa de la zona de variables temporales que cumple la misma misión.

VALOR

Esta estructura se usa para almacenar los valores que son utilizados en la generación de código para evaluar expresiones. Define el tipo del valor (constante, registro, temporal o identificador), si es Lvalue o Rvalue, su posición (si es registro o temporal) o su localización en memoria, su longitud, su tipo de almacenamiento, su descriptor, su nombre (si es un identificador) y su valor entero o real (si es una constante). Está definida en exp.h.

FUNC_INT

Definida en opers.h. Define una función integrada: su nombre (con el que se la llamará desde C), su código de operación para la máquina virtual, su tipo de retorno, los tipos de sus parámetros y el tamaño en bytes de los mismos.

VOLCADOS

Esta estructura se usa para indicar qué registros se han volcado en la pila (porque estaban ocupados) antes de una llamada a un subproceso. Habrán de ser restaurados tras el retorno.

5.- La gramática.

La descripción de la gramática del lenguaje LCR se halla en el fichero `lcr.y`, mezclada con las acciones semánticas del compilador.

Hay 163 reglas en la gramática. Algunas reglas podrían eliminarse si no existiesen acciones semánticas asociadas a ellas. Por ejemplo, la regla que inicia una expresión es

$$\text{exp} \rightarrow \text{expr}$$

y es realmente en `expr` donde se define la sintaxis de las expresiones. Esto se ha tenido que hacer así porque hay unas acciones semánticas de depuración asociadas con las expresiones, y de esta forma se ejecutan cada vez que se evalúa una.

La gramática contiene un conflicto desplazamiento/reducción, motivado por el eterno problema del SI-ENTONCES-SI_NO. Diré en mi descargo que la gramática ANSI para C tiene ese mismo conflicto (cfr. K&R, p. 234). Dada la forma que tiene PCYACC de resolver este tipo de conflictos, no hay que preocuparse por él.

La unión que se usa para la pila atributos del analizador es la siguiente:

```
%union
{
  int entero;
  float real;

  TIPO          *tipoptr;
  LISTA_DIMENSIONES *lista_dimensiones;
  LISTA_IDS      *lista_ids;
  SIMBOLO        *simboptr;
  LISTA_DESCRIPTORES *lista_descriptores;
```

```
DESCR          *descr;
VALOR          *valor;

char cad[80];
char car;
}
```

Las precedencias que existen entre los operadores del lenguaje LCR están tomadas de las del lenguaje C. En orden creciente de precedencia, los operadores de LCR se clasifican de la siguiente forma:

- Asignación (=).
- Disyunción lógica (||).
- Conjunción lógica (&&).
- Operadores algebraicos de bits (& y |).
- Comparación de igualdad y desigualdad (== y !=).
- Otras comparaciones (<, >, >=, <=).
- Suma y resta (+ y -).
- Multiplicación y división (* y /).
- Negación (-), operador Tamano, complemento a uno (~), negación lógica (!), indirección (*), operador de dirección (&) y conversión de tipo ((entero) y (real)).
- Indexación ([y]).

A continuación, y antes de estudiar las acciones semánticas, comentaremos las características de la máquina virtual.

6.- La máquina virtual.

La máquina virtual o microprocesador virtual para el que se va a generar código no es otra cosa que un programa (por eso es una máquina virtual o lógica, si no sería física) que leerá el

fichero objeto y lo ejecutará, interpretando las instrucciones una a una.

Estudiaremos primero la organización de la máquina para luego ver las instrucciones y las estructuras de llamada.

6.1.- Organización.

En el aspecto físico (*hardware*), la máquina está compuesta por una unidad de ejecución de instrucciones (que incluye una unidad aritmética con capacidad de cálculo en punto flotante de precisión simple), una serie de registros, un conjunto de indicadores, una pila y un banco de memoria.

6.1.1.- Los registros.

Actualmente, la máquina tiene cinco registros generales (un número fácilmente modificable) y tres registros especializados. Los registros generales son ortogonales (es decir, se puede ejecutar cualquier operación sobre ellos), tienen una anchura de 16 bits y se nombran R0 a R4. Los tres registros especiales son SP (el puntero de pila o *stack pointer*), FP (el puntero del marco de llamada o *frame pointer*) e IP (el puntero de instrucción o *instruction pointer*).

EL registro IP apunta en todo momento a la instrucción que se ejecutará en siguiente lugar.

6.1.2.- Los indicadores.

Hay una serie de indicadores que se ven afectados al realizar

una operación de comparación. Dependiendo de estos indicadores se realizan los saltos condicionales.

Los indicadores de la máquina virtual son:

<u>Indicador:</u>	<u>Significado:</u>
G	En la última comparación, el primer operando era mayor que el segundo.
GE	En la última comparación, el primer operando era mayor o igual que el segundo.
L	En la última comparación, el primer operando era menor que el segundo.
LE	En la última comparación, el primer operando era menor o igual que el segundo.
EQ	En la última comparación, el primer operando era igual que el segundo.
NE	En la última comparación, el primer operando era distinto del segundo.

Por supuesto, hay redundancias, pero se mantienen por razones de claridad.

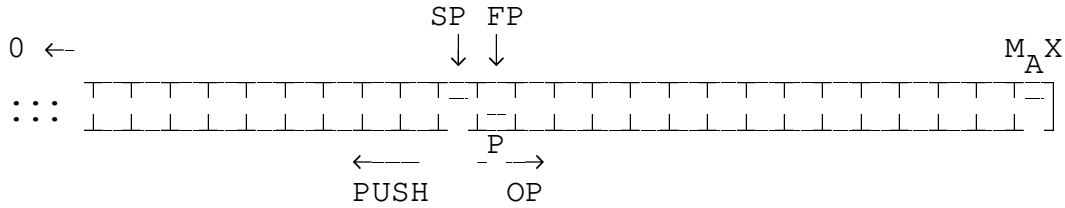
6.1.3.- **La pila.**

Dado que los valores de punto flotante tienen un tamaño de 32 bits, no se pueden almacenar en los registros. Para efectuar operaciones de punto flotante se usan las variables temporales. Una variable temporal es una zona de memoria situada en la pila.

Además de las variables temporales, en la pila se almacenan las variables locales y parámetros de los procedimientos y otros valores.

La pila de la máquina virtual crece hacia abajo. Su unidad

mínima de acceso es el byte, pero en la práctica sólo se accede por palabras de 16 bits.



Las instrucciones PUSH y POP acceden a la pila y actualizan el registro SP, incrementándolo o decrementándolo en dos:

- La instrucción PUSH op decrementa en dos unidades el registro SP y escribe su operando a la dirección apuntada por dicho registro. Por lo tanto, PUSH op es equivalente a

```
SUB SP, 2
MOV PTR [SP], op
```

- La instrucción POP dest mueve el contenido de la dirección a que apunta SP al operando dest e incrementa en dos el registro SP. POP dest equivale a

```
MOV dest, PTR [SP]
ADD SP, 2
```

Como se ve, el registro SP apunta al último elemento introducido en la pila.

Como ejemplo, si la pila comienza en la posición 2048 (recuérdese que crece hacia abajo) y su estado actual es

2048	F A	
2047	0 D	
2046	1 2	
2045	2 3	← SP
2044		
2043		
2042		
2041		

tras una operación PUSH #1981 ($1981_{10} = 07BD_H$), quedará

2048	F A	
2047	0 D	
2046	1 2	
2045	2 3	
2044	0 7	
2043	b d	← SP
2042		
2041		

(las palabras, como en los microprocesadores Intel, se almacenan con el byte menos significativo primero). Si ahora ejecutamos la instrucción POP R1, tendremos de nuevo

2048	F A	
2047	0 D	
2046	1 2	
2045	2 3	← SP
2044		
2043		
2042		
2041		

y el registro R1 contendrá el valor $07BD_H$.

Otro método de acceso a la pila es de forma indirecta mediante el registro FP: la expresión FP+d se refiere a la posición de memoria que se obtiene al sumar d (que puede ser negativo) al contenido del registro FP. Este direccionamiento se usa para acceder a las variables locales, a los parámetros de los procedimientos y a las variables temporales.

6.1.4.- **La memoria.**

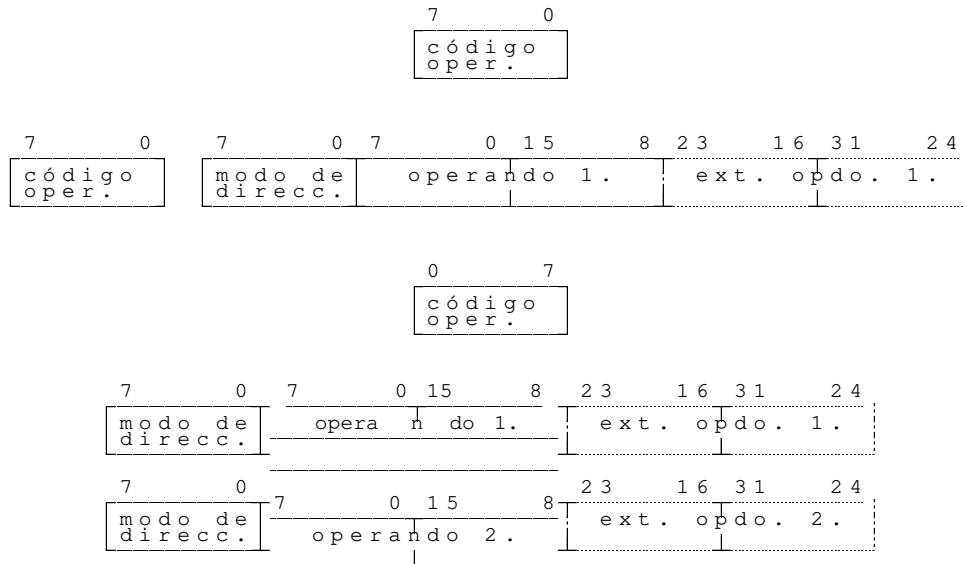
La máquina virtual tiene una capacidad de direccionamiento de 16 bits. Esto le otorga un espacio de memoria máximo de 64KB (65536 bytes).

La memoria es un bloque independiente de la pila (la pila también puede tener un máximo de 64KB), y su unidad de direccionamiento es el byte, aunque al igual que en la pila se suele acceder por palabras o dobles palabras.

6.1.5- **Las instrucciones.**

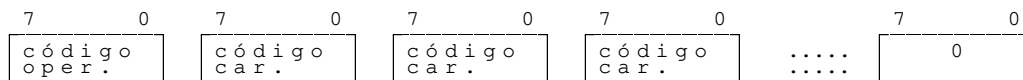
La máquina virtual tiene un formato de instrucción que se

compone de un código de operación de 8 bits y dos operandos opcionales, cada uno de los cuales está constituido por un byte que indica el modo de direccionamiento y uno o dos bytes que definen el operando en sí. Gráficamente, una instrucción tiene uno de los siguientes formatos:



Como se ve, las palabras tienen los bytes en orden inverso: primero el menos significativo. Igualmente pasa con las dobles palabras: primero va la palabra menos significativa.

Hay una excepción a estos formatos: la instrucción WRSTR imprime en pantalla una serie de caracteres (los bytes que siguen al código de operación) hasta encontrar un carácter cero. El formato es, pues:



Los modos de direccionamiento de la máquina virtual son siete:

- Inmediato: se representa #n, siendo n un valor entero de 16 bits. Se evalúa a n.
- Inmediato real: se representa #Fn, siendo n un valor real de precisión simple (32 bits). Se evalúa también a n.
- Registro: se representa Rx, siendo x el número (0-5) de uno de los registros generales. Se evalúa al contenido de ese registro.
- Directo Absoluto: se representa PTR d, siendo d una dirección de memoria (16 bits). Se evalúa al contenido de la posición de memoria d. Dependiendo de la instrucción se tomarán 16 o 32 bits.
- Directo respecto a FP: se representa PTR FP+d, siendo d un desplazamiento en la pila respecto al registro FP (d puede ser negativo). Se evalúa al contenido de la posición de la referida de la pila. Dependiendo de la instrucción se tomarán 16 o 32 bits.
- Indirecto respecto a FP: se representa PTR [FP+d] y se evalúa al contenido de la posición de memoria designada por la palabra (16 bits) que se halle en la posición FP+d en la pila. Así, si FP vale 8192 y d es -32, se tomará la palabra de la posición 8160 de la pila. Si esa palabra es, por ejemplo, 23043, el direccionamiento se evaluará al contenido de la dirección de memoria 23043.
- Indirecto respecto a un registro: se representa PTR [Rx] y se evalúa al contenido de la dirección de memoria apuntada por el registro x.

Las instrucciones de la máquina virtual pueden clasificarse

en seis grupos: manejo de valores y comparación, indirección, control, entrada/salida, aritméticas e integradas. Veremos cada grupo por separado.

6.1.5.1.- Instrucciones de manejo de valores y comparación.

Estas instrucciones mueven operandos de un lugar a otro o efectúan comparaciones entre ellos. Este grupo comprende las instrucciones

MOV, CMP, PUSH, POP, FMOV, FCMP, FICMP,
FICMPR, FPUSH, FPOP, FILD y FIST,

Las instrucciones MOV y FMOV aceptan dos operandos y mueven el segundo al primero. La diferencia entre las dos es que la segunda (FMOV) trabaja con valores de punto flotante. Como ejemplo, la instrucción

MOV R2, PTR [R0]

almacenará en el registro 2 el valor contenido en la posición de memoria contenida en R0. La instrucción

FMOV PTR FP+4, PTR 2048

cargará la posición de memoria FP+4 (en la pila) con el contenido de la posición de memoria 2048 (en la memoria).

Las instrucciones PUSH, POP, FPUSH y FPOP actúan sobre la pila como ya se vió anteriormente.

La instrucción FILD considera su primer argumento como real, y el segundo como entero, y carga el primero con el segundo.

FILD PTR 100, #20

convertirá el valor 20 a punto flotante y lo almacenará en la posición 100.

FIST hace lo contrario:

FIST R3, PTR 100

convertirá el valor flotante que haya en la posición 100 a entero y lo almacenará en el registro 3.

CMP y FCMP comparan sus dos operandos y activan los indicadores correspondientes. El orden de los operandos, por supuesto, es importante.

FICMP considera el primer argumento real y el segundo entero y los compara.

FICMPR sirve para efectuar comparaciones entero-real (cuando el entero ha de ser el primer operando). Hace una comparación inversa.

6.1.5.2.- Instrucción de indirección.

LEA es el equivalente en código máquina al operador & de C y LCR. La instrucción

LEA op1, op2

almacena en op1 la dirección real de op2. Si op2 es un registro, habrá un error. Esta instrucción tiene la misma forma que una instrucción MOV, pero en lugar de almacenar el contenido

de op2, almacena su dirección.

Como ejemplo, si FP vale 32767, la instrucción

```
LEA R0, PTR FP+48
```

no almacenará en R0 el valor almacenado en la posición de pila 32815 (32767+48), sino el propio valor 32815 (la dirección resultante).

6.1.5.3.- Instrucciones de control.

Este grupo de instrucciones engloba las siguientes:

```
ENTER, HLT, JG, JGE, JL, JLE,  
JEQ, JNE, JMP, CALL, RET,
```

Las instrucciones Jxx efectúan saltos condicionales (excepto JMP, que salta incondicionalmente): se comprueba si el indicador designado por xx está activado y, si lo está, se retoma la secuencia de instrucciones en la posición que indique el operando de la instrucción (es decir, se carga el operando en el registro IP).

La instrucción CALL llama a una subrutina: almacena en la pila el contenido actual del registro IP (que apunta a la instrucción siguiente) y almacena en dicho registro el operando.

La instrucción RET devuelve el control desde una subrutina al código que la llamó. Para ello, simplemente toma la palabra superior de la pila y la almacena en el registro IP. RET es, por tanto, igual a

POP IP

La instrucción HLT detiene la ejecución del programa en curso y (ya que estamos en una máquina virtual) retorna al sistema operativo.

La instrucción ENTER sirve para crear espacio en los marcos de llamada para las variables locales y temporales de los procedimientos de alto nivel. Su argumento es una palabra que indica el número de bytes que queremos reservar en la pila para dichas variables. Se comprueba si hay espacio en la pila para almacenar esa cantidad de bytes (es decir, si SP más el número de bytes es mayor que el límite inferior de la pila) y, si lo es, se suma el número de bytes a SP.

6.1.5.4.- Instrucciones de entrada/salida.

Las tres instrucciones de entrada/salida de la máquina virtual son WRT, FWRT, READ, FREAD y WRSTR.

La instrucción WRT escribe en pantalla su argumento. La instrucción READ lee un valor y lo almacena en su argumento. Las instrucciones FWRT y FREAD hacen lo mismo considerando a sus argumentos como de tipo real.

La instrucción WRSTR no tiene el formato normal de las instrucciones, sino que tiene como argumento una serie de bytes con un cero al final de la misma. Su misión es imprimir en pantalla los caracteres ASCII definidos por esos bytes. En la cadena ASCIIIZ a imprimir se aceptan las secuencias de escape del lenguaje C.

6.1.5.5.- Instrucciones aritméticas.

Este grupo incluye las siguientes instrucciones:

ADD, SUB, MUL, DIV, NEG, NOT, AND, OR, FADD, FIADD, FSUB, FISUB,
FSUBR, FISUBR, FMUL, FIMUL, FDIV, FIDIV, FDIVR, FIDIVR, FNEG,
SEN, COS, TAN, COT, EXP y LN

Las instrucciones ADD, SUB, MUL, DIV, AND y OR, efectúan las operaciones binarias que representan (suma, resta, multiplicación, división, producto lógico y suma lógica) entre valores enteros.

Las instrucciones FADD, FSUB, FMUL y FDIV se ocupan de la suma, resta, multiplicación y división entre números de punto flotante.

Las instrucciones aritméticas de punto flotante que tienen

una 'I' tras la 'F' consideran que el segundo operando es entero. Estas instrucciones son FIADD, FISUB, FISUBR, FIMUL, FIDIV, FIDIVR.

Las instrucciones aritméticas de punto flotante que terminan con una 'R' efectúan la operación de modo inverso (operan el segundo operando con el primero, aunque siguen almacenando el resultado en el primero). Estas instrucciones son FSUBR, FISUBR, FDIVR y FIDIVR.

Las instrucciones NOT, NEG, FNEG, SEN, COS, TAN, COT, EXP y LN aceptan un sólo operando y calculan su complemento a uno, negación (entera y real), seno, coseno, tangente, cotangente, exponencial y logaritmo neperiano respectivamente. Las dos primeras instrucciones (NOT y NEG) actúan sobre operandos enteros, y el resto sobre operandos de punto flotante.

6.1.5.6.- Instrucciones integradas.

Estas instrucciones motivan que el microprocesador virtual efectúe una llamada a la rutina asociada a ellas. Son, entre otras, las funciones de control del robot móvil.

6.2.- Marcos de llamada.

Una vez estudiadas las instrucciones de la máquina virtual, explicaré cuál es el marco de llamada de una rutina de alto nivel (del lenguaje LCR).

La secuencia de llamada de una rutina en LCR es la siguiente:

- Se vuelcan en la pila todos los registros generales que

estén ocupados y se toma nota de cuáles se han volcado.

- Se evalúan los parámetros de la rutina y se almacenan en la pila. ATENCION: los parámetros se almacenan en la pila en orden inverso, al igual que en C. Es decir, el último parámetro se introduce en la pila el primero.

- Se llama a la rutina con CALL.

- La rutina, al tomar el control, salva primero en la pila el puntero al marco de llamada (registro FP) de la rutina anterior.

- A continuación, inicializa su propio puntero de marco, haciendo que apunte al límite actual de la pila (SP): MOV FP, SP.

- Como último paso antes de la ejecución del cuerpo de la rutina, se crea espacio en la pila para las variables locales y temporales, mediante la instrucción ENTER.

- Se ejecuta el cuerpo de la rutina.

- Al finalizar la rutina, se restaura el valor inicial de SP (que está almacenado en FP): MOV SP, FP.

- Se restaura el marco de llamada anterior, que estaba almacenado en la pila: POP FP.

- Se retorna de la subrutina: RET.

- Al recuperar el control, el programa que ha llamado a la subrutina elimina los parámetros de la pila, añadiendo al registro SP el número de bytes que éstos ocupen.

Veremos a continuación un ejemplo de la secuencia de llamada

a una rutina. Supongamos que la rutina F acepta dos parámetros: el primero un valor entero (16 bits) y el segundo un puntero (16 bits) a entero.

La línea de código LCR

```
Llama_a F ( 11476, &x )
```

generará el código que se muestra a continuación:

```
(0) LEA R0, PTR 216
    PUSH R0
    PUSH #11476
(1) CALL F
(7) ADD SP, #4
```

Hemos supuesto que la variable entera x se halla almacenada en la memoria (es una variable global o estática), en la posición 216. Como se ve, primero se introduce en la pila el último parámetro (la dirección de x) y a continuación el primero (el número 11476).

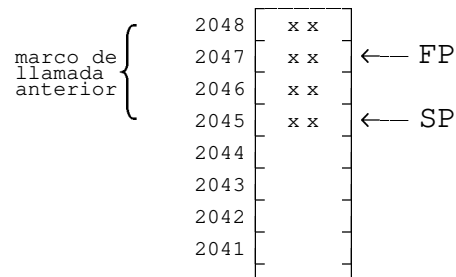
La rutina F tendrá el siguiente código:

```
(2) PUSH FP
    MOV FP, SP
(3) ENTER 4
(4) <código de la rutina>
    MOV SP, FP
(5) POP FP
(6) RET
```

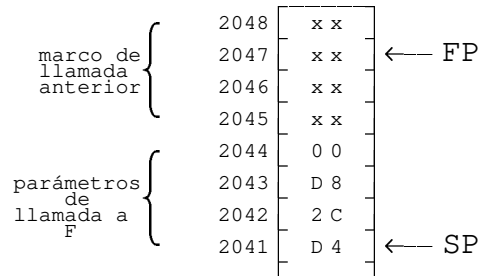
Hemos supuesto que la rutina F necesita 4 bytes para el almacenamiento local de dos variables enteras E1 y E2.

Los números que aparecen a la izquierda de algunas instrucciones son referencias. Se considera tomada la referencia justo antes del comienzo de la ejecución de esa línea.

Consideremos que la pila tiene inicialmente esta disposición (referencia 0):



El programa de llamada introduce sus parámetros en la pila, que queda de la siguiente forma (referencia 1):



Ha de considerarse que $11476_{10} = 2CD4_H$, que $216_{10} = 00D8_H$ y que las palabras se almacenan en memoria con los bytes invertidos. Además los parámetros están, ya se ve, en orden inverso.

Se llama a la función F (CALL F) y se almacena la posición de la instrucción siguiente (digamos que esa posición es $32012_{10} = 7D0C_H$). La pila queda así (referencia 2):

	2048	x x	
marco de llamada anterior	2047	x x	← FP
	2046	x x	
	2045	x x	
	2044	0 0	
parámetros de llamada a F	2043	D 8	
	2042	2 C	
	2041	D 4	
dirección de retorno	2040	7 D	
	2039	0 C	← SP

Cuando la rutina F toma el control, vuelca en la pila el registro FP (vale $2047_{10} = 07FF_H$) y lo carga con el valor actual de SP. La pila tendrá este aspecto (referencia 3):

	2048	x x	
marco de llamada anterior	2047	x x	
	2046	x x	
	2045	x x	
	2044	0 0	
parámetros de llamada a F	2043	D 8	
	2042	2 C	
	2041	D 4	
dirección de retorno	2040	7 D	
	2039	0 C	
antiguo FP	2038	0 7	
	2037	F F	← SP y FP

Tras la llamada a la instrucción ENTER 4, tenemos (referencia 4):

	2048	x x	
marco de llamada anterior	2047	x x	
	2046	x x	
	2045	x x	
	2044	2 C	
parámetros de llamada a F	2043	D 4	
	2042	0 0	
	2041	D 8	
dirección de retorno	2040	7 D	
	2039	0 C	
antiguo FP	2038	0 7	
	2037	F F	← FP
zona de variables locales (E1 y E2)	2036	x x	
	2035	x x	
	2034	x x	
	2033	x x	← SP

Ahora se ejecutará el código de la rutina F. Las referencias a los parámetros se efectúan con desplazamiento positivo respecto a FP. Así, el valor entero 11476 (2CD4_H) está en FP+6, el puntero &x (es decir, la dirección 00D8_H), en FP+4.

Las referencias a variables locales y temporales tienen un desplazamiento negativo respecto a FP: la variable entera local E1 comienza en FP+2 y ocupa las posiciones FP+2 y FP+1. La variable E2 ocupa FP+3 y FP+4 y comienza en FP+4.

Una vez ejecutado el cuerpo de la rutina, la parte final de ésta restaura el registro SP, eliminando así la zona de temporales. Referencia 5:

marco de llamada anterior	2048	x x	← FP y SP
	2047	x x	
	2046	x x	
	2045	x x	
parámetros de llamada a F	2044	0 0	
	2043	D 8	
	2042	2 C	
	2041	D 4	
dirección de retorno antiguo	2040	7 D	
	2039	0 C	
FP	2037	F F	

Tras esto, tomará el valor anterior de FP (referencia 6):

marco de llamada anterior	2048	x x	← FP
	2047	x x	
	2046	x x	
	2045	x x	
parámetros de llamada a F	2044	0 0	
	2043	D 8	
	2042	2 C	
	2041	D 4	
dirección de retorno	2040	7 D	
	2039	0 C	← SP

Y, por fin, retornará al programa que la llamó (referencia 7):

	2048	x x	
marco de llamada anterior	2047	x x	← FP
	2046	x x	
	2045	x x	
	2044	0 0	
parámetros de llamada a F	2043	D 8	
	2042	2 C	
	2041	D 4	← SP

El programa de llamada restaurará el estado inicial de la pila añadiendo 4 a SP para borrar los parámetros. La pila quedará en el estado final siguiente (que es idéntico al inicial):

	2048	x x	
marco de llamada anterior	2047	x x	← FP
	2046	x x	
	2045	x x	← SP

Una vez estudiada bastante a fondo la máquina virtual, su organización, las instrucciones y las secuencias de llamada de alto nivel, pasaremos a tratar las acciones semánticas.

7.- Las acciones semánticas.

Veremos ahora cuáles son las acciones semánticas que se llevan a cabo durante el análisis del fichero de entrada, y que tienen por objeto la comprobación de la validez semántica del texto y la generación de código (ya sea traducción a C o código máquina virtual).

Inicialmente, y antes de comenzar el análisis (es decir, aún en main), se crea el primer módulo, que será el número cero, y cuyo nombre es \$GLOBALES. Este módulo ficticio será el propietario de todas las variables de tipo global.

7.1.- Programa (axioma), rutinas.

Al comenzar el análisis, el axioma nos indica que un programa está conformado por una serie de rutinas. Una rutina, a su vez, consta de una cabecera, una sección opcional de declaración de rutinas (sección USA), otra sección también opcional de declaración de variables y un bloque obligatorio de código (texto).

Las tres primeras metanociones no producen código: su función es puramente semántica. Sin embargo, el bloque de texto sí genera código y por ello es necesario incluir la cabecera de la rutina antes de él. Esta cabecera contiene las instrucciones de preparación del marco de llamada: volcado del registro FP, actualización del mismo para que apunte al marco de llamada (FP←SP) y generación de la instrucción ENTER. Dado que en ese momento no se conoce el tamaño de la zona de pila a reservar (se tienen las variables locales, pero se desconoce la longitud de las temporales), se almacena la posición del argumento de ENTER en la estructura MODULO y se emite un valor ficticio, para efectuar posteriormente un *backpatch*. Además, en la cabecera se resuelve la etiqueta de entrada al módulo.

Tras esta cabecera se evalúa el texto y, por tanto, se genera el código equivalente. Como último paso en el tratamiento del módulo se hace el *backpatch* del argumento de la instrucción ENTER de entrada, se resuelve la etiqueta de salida del módulo (EXITxxx, siendo xxx el código del módulo), se restauran los registros FP y SP y se emite una instrucción RET.

7.2.- Cabecera.

El tratamiento de la cabecera de una rutina (nótese que estamos en una definición) depende del tipo de rutina de que se

trate:

- Si es el programa principal, únicamente se añade el módulo \$MAIN a la tabla de módulos.

- Si es una función o una subrutina, además de añadir el módulo, se efectúa un tratamiento de la cabecera, que considera la lista de parámetros opcionales y el tipo de retorno (en el caso de las funciones). Este tratamiento, para el caso de una definición, sigue la siguiente estrategia:

- Si el procedimiento ya existe, debe haber sido declarado previamente (o puede haber sido definido. Este error se habría detectado al añadir el módulo). En este caso, se comprueban los parámetros y el tipo de retorno, considerando que prevalecen los de la definición sobre los demás.
- Si no existe, los introducimos en la tabla de símbolos y lo conectamos con sus parámetros (cuyos símbolos correspondientes ya han sido creados por la metanoción lista-pars-op).

7.3.- Lista de parámetros opcionales.

Esta lista consta de cero, una o más estructuras de tipo 'lista-ids **es** tipo-simple' y constituye la lista de argumentos de un subproceso. Para cada una de estas estructuras se crea un tipo (parámetro local con el tipo simple indicado) que, junto con la lista de identificadores, se usa para crear una lista de símbolos que es el resultado de esta metanoción.

7.4.- Declaración de rutinas opcional.

Esta sección es opcional, pero si existe comienza con la palabra reservada **usa**, a la cual siguen una o más declaraciones de rutinas.

Una declaración es idéntica a una cabecera, con la salvedad de que la lista de parámetros opcionales se sustituye por una lista de tipos de parámetros opcionales.

La acción semántica asociada también es el tratamiento de la cabecera, pero considerándola como una declaración.

7.5.- Lista de tipos de parámetros opcionales.

Esta metanoción genera, al igual que la lista de parámetros opcionales, una lista de símbolos. Los tipos de los símbolos se conocen, pero los nombres de los parámetros no. Por lo tanto, se usan unos nombres ficticios no repetibles: la cadena '\$parfict' seguida de un número que se incrementa cada vez.

7.6.- Declaración de variables opcional.

Si existe, esta sección comienza con la palabra reservada **variables**. Le sigue una serie de una o más estructuras de tipo 'lista-ids **es** tipo'. Como tenemos el tipo completo, no es necesario crearlo (como hicimos en lista-pars), sino que creamos directamente una lista de símbolos a partir de la lista de identificadores y del tipo. A continuación introducimos esos símbolos en la tabla.

7.7.- Tipos.

Las metanociones de creación de tipos manejan descriptores (listas de estructuras DESCR), estructuras TIPO y estructuras LISTA_DIMENSIONES.

7.8.- Texto ejecutable.

La sección obligatoria de un módulo es el texto ejecutable, es decir, las instrucciones que componen el cuerpo de dicho módulo.

El texto ejecutable es una serie de una o más instrucciones

encuadradas por los delimitadores **inicio** y **fin**. Cada instrucción debe terminar en punto y coma (';'). Las posibles instrucciones y sus acciones semánticas correspondientes se estudiarán a continuación.

- Instrucción nula: no tiene ninguna acción semántica asociada.

- Instrucción ETIQUETA: esta instrucción define en la posición actual una etiqueta que posteriormente será referenciada por una instrucción SALTAR_A. La acción semántica asociada es simplemente resolver la etiqueta indicada.

- Asignación: esta instrucción tiene la forma 'exp = exp'. La acción semántica asociada comprueba en primer lugar que el primer término es un *Lvalue*. A continuación se chequea la igualdad de los tipos de las dos expresiones. Por último, se emite una instrucción MOV y se eliminan las estructuras VALOR asociadas a las dos expresiones.

- Llamada a otro módulo: se comprueba que el símbolo existe y es una función o subrutina. A continuación se vuelcan los registros que estén ocupados y se apila esta información en la pila de volcados. Se introduce el símbolo que define al módulo llamado en la tabla de símbolos y se procesa la metanoción lista-pars-act-op, que se ocupa de introducir los parámetros en la pila (recuerde, en orden inverso) y de crear una lista con los descriptores de los mismos.

Se saca el símbolo llamado de la pila de símbolos y la información sobre registros volcados de la pila de volcados. Se comprueba que los tipos de los parámetros actuales (la lista de descriptores creada por lista-pars-act-op) concuerdan con los tipos de los parámetros formales, y se calcula su longitud.

Por último, se crea la llamada a la función (se emite un CALL y se eliminan los parámetros con 'ADD SP,xx'), se recuperan los registros volcados y se descarta el valor de retorno.

- Bloque de instrucciones: no hay acciones semánticas asociadas, excepto las de conversión a C. Pero, como se dijo, sólo se tratarán aquí las acciones semánticas de generación de código máquina virtual.

- Instrucción condicional SI-ENTONCES-SI_NO: tras la evaluación de la expresión a comprobar y la consiguiente generación de código para la misma se comprueba el resultado de esa evaluación y se salta si es falso a la etiqueta correspondiente al SI_NO. Si, por el contrario, es verdadero, se continúa y se ejecuta la parte del ENTONCES. Tras la ejecución de esta parte, un salto incondicional pasará por encima del código correspondiente a la alternativa SI_NO.

Todo esto es válido si existe una alternativa SI_NO. En caso de que no exista, el incumplimiento de la condición motivará el salto fuera de la instrucción.

El código generado para los dos tipos de construcciones condicionales es el siguiente:

```

SI
  expr          <evaluación de expr>
ENTONCES
  <comparación de expr con cero y
  salto si es cero a ET_ELSE>
instr          <código para instr>
SI_NO
  <salto incondicional a ET_FINSI>
  <definición de ET_ELSE>
instr          <código para instr>
               <definición de ET_FINSI>

```

Y para el condicional simple:

```

SI
  expr                <evaluación de expr>
ENTONCES              <comparación de expr con cero y
                      salto si es cero a ET_ELSE>
instr                 <código para instr>
                      <definición de ET_ELSE>

```

- Bucle MIENTRAS-HACER: antes de del código para la evaluación de la expresión de permanencia en el bucle se definen las etiquetas de inicio de bucle y de continuación. Se evalúa la expresión y se compara con cero. Si es cero, se salta a la etiqueta ET_FIN_BUCLE. Si no, se ejecutan las instrucciones que componen el cuerpo del bucle y se salta incondicionalmente a la etiqueta de inicio de bucle. Tras la generación de este código se resuelven las etiquetas de fin de bucle y de salida (break).

El código que se genera es, por lo tanto:

```

MIENTRAS              <definición de ET_INI y ET_CONT>
  expr                <evaluación de expr>
HACER                 <comparación con cero y salto
                      si lo es a ET_FIN>
instr                 <código para instr>
                      <salto a ET_INI>
                      <definición de ET_FIN y ET_BREAK>

```

- Bucle REPETIR-HASTA_QUE: antes del código del cuerpo del bucle se resuelve la etiqueta de inicio. A continuación se genera dicho código, se resuelve la etiqueta de continuación, se evalúa la expresión de salida (y si es cero se salta al inicio) y se resuelve la etiqueta de salida (break).

Se genera el siguiente código:

REPETIR	<definición de ET_INI>
instrs	<código para instrs>
HASTA_QUE	<definición de ET_CONT>
expr	<evaluación de expr>
	<si es cero, salto a ET_INI>
	<definición de ET_BREAK>

- Bucle PARA-HACER: primero se almacena el valor inicial del bucle en la variable índice. A continuación se resuelve la etiqueta correspondiente a la comprobación (test) del bucle. Se compara el valor del índice con el valor final y si el primero es mayor se salta a la etiqueta de fin de bucle. Tras esto se genera el código correspondiente al cuerpo del bucle y se resuelve la etiqueta de continuación. Por último se añade el incremento al índice (uno si no se especifica otro), se salta incondicionalmente al test y se resuelve la etiqueta de fin de bucle (que es también la de salida por break).

Se genera el siguiente código:

PARA ID DESDE exp1 HASTA exp2	<carga de ID con exp1>
INCREMENTO exp3 HACER	<definición de ET_TEST>
	<comparación de ID con exp2 y salto si es mayor a ET_FIN>
instr	<código para instr>
	<definición de ET_CONT>
	<incremento de ID>
	<salto a ET_TEST>
	<definición de ET_FIN>

- Instrucción SALIR: al comienzo de cada bucle se almacenan en una pila de etiquetas especial las etiquetas de salida y continuación, que son eliminadas al acabar el mismo. La

instrucción SALIR genera un salto incondicional a la etiqueta que esté en la cima de la pila de etiquetas de salida (break). Esta etiqueta es la de salida del bucle que se esté evaluando en ese momento (el más profundo). Si la pila de etiquetas de salida está vacía es que se ha usado una instrucción SALIR fuera de un bucle. Se generará entonces el error pertinente.

- Instrucción CONTINUAR: al igual que la instrucción SALIR, esta orden genera un salto incondicional a la etiqueta que se encuentre en la cima de la pila de etiquetas de continuación. El error de uso se trata de la misma forma que en la instrucción SALIR.

- Instrucción SALTAR_A: esta instrucción genera un salto incondicional a la etiqueta indicada. La función que genera este salto se ocupará de comprobar si la etiqueta está ya resuelta para emitir la dirección correcta (si lo está) o un espacio que después será cumplimentado por un *backpatch*.

- Instrucción RETORNAR: si la instrucción no tiene argumento, se limita a generar un salto incondicional al punto de salida del procedimiento que se esté compilando.

Si la instrucción lleva adjunta una expresión cuyo resultado se ha de retornar como valor de la función en que se halla, se cargarán los registros con dicho resultado.

Las convenciones de retorno de valores de funciones en LCR son las siguientes:

- Los valores enteros y de tipo puntero se devuelven en el registro R0.

- Los valores reales se devuelven en los registros R0 y

R1.

- Instrucción PARAR: esta instrucción genera una directiva HLT de código máquina virtual, que tiene como resultado el fin de la ejecución del programa.

- Instrucción ESCRIBIR: si el argumento es una cadena se emite la instrucción WRTSTR de código máquina virtual y a continuación los caracteres de la cadena, para acabar con un carácter cero.

Si el argumento es una expresión se emite una instrucción WRT o FWRT dependiendo de su tipo.

- Instrucción LEER: tras comprobar que no se está intentando leer un puntero y que la expresión es una *Lvalue*, se calcula su dirección y se emite ésta como operando de una instrucción READ o FREAD (según el tipo de la expresión).

7.9.- **Expresiones.**

Una expresión puede ser un elemento simple (una constante o un identificador que sea una variable) o un operador aplicado a una expresión. Consideramos también que las funciones son operadores (operador paréntesis).

La gestión de las expresiones se realiza usando las estructuras de tipo VALOR. Una estructura de este tipo representa un valor (obvio, por otra parte), indicando su localización (memoria, registro ...), su tipo, si es o no es una *Lvalue* y otras características necesarias. Hay una serie de funciones que, partiendo de uno o varios valores, generan el código necesario para realizar una operación entre ellos (considerando su tipo,

posición y demás características) y generan un nuevo valor que es el resultado de la operación.

- Constantes (enteras o reales) e identificadores: si se encuentra un elemento de estos tipos se utiliza la función *CreaValor*, que genera una estructura de tipo VALOR que describe al elemento. Si éste es un identificador, la función lo busca en la tabla de símbolos: primero comprueba si es local; luego, si es estático (los identificadores estáticos van precedidos por '\$xx', donde xx es el código del módulo al que pertenecen); y por último comprueba si es global. Si aún así no lo encuentra es que no está definido. Si lo ha hallado, antes de seguir adelante se comprueba que el símbolo es una variable y no una función, subrutina u otra cosa.

- Operador TAMANO: se crea como resultado un nuevo valor entero constante que contiene el tamaño en bytes ocupado por el argumento.

- Llamada a función: se efectúa de igual forma que la instrucción LLAMAR_A, pero con la diferencia de que el valor de retorno no se desprecia.

- Indexación (operador '['): se llama a la función *EfectuaIndireccion* con la base y el desplazamiento como parámetros. Esta función comprueba que la base es de tipo puntero y que el desplazamiento es entero y genera código para hallar la dirección resultante, considerándola como *Lvalue* (ya que es una dirección). El tipo que se retorna es el mismo de la base pero sin el primer 'puntero_a'. Es decir, que si la base es un puntero_a puntero_a entero se retornará un puntero_a entero.

- Menos unario ('-'): si el operando es un puntero, se genera un mensaje de error. Si no lo es, se emite código para cambiar su

signo.

- Complemento a uno ('~'): sólo se puede aplicar a un operando entero. Si no lo es se emite un error.

- Negación lógica ('!'): la negación se realiza por flujo de control y no por aritmética (que, por otra parte, sería más sencillo). Se genera un nuevo valor y se inicializa a uno. Se compara el operando con cero y, si es igual, se continúa. Si no es cero se almacena un cero en el valor resultado. Algorítmicamente quedaría:

```

resultado ← 1
si operando == 0 saltar a ET_NOT
resultado ← 0
ET_NOT:

```

- Indirección ('*'): el operador de indirección es equivalente al operador de indexación con desplazamiento cero:

```
*p == p[0]
```

Por lo tanto, para evaluarlo se efectúa una llamada a la función `EfectuaIndireccion` con el segundo parámetro (el desplazamiento) nulo.

- Dirección ('&'): la función `DireccionDe` genera el código para este operador. Devuelve un *Rvalue* que es la dirección en la que se halla el argumento. Por supuesto, el argumento ha de ser *Lvalue*.

- Conversores de tipo ((ENTERO) y (REAL)): ninguno de los dos puede operar sobre punteros. Tras comprobar que la conversión es realmente necesaria, la efectúan llamando a la función

EfectuaOpUnaria.

- Suma, resta, multiplicación, división, producto lógico y suma lógica ('+', '-', '*', '/', '&' y '|'): de todas estas operaciones se ocupa la función EfectuaOpBinaria, que genera el código adecuado dependiendo de tipo y de la situación de los valores que hay que operar.

La suma lógica y el producto lógico necesitan que sus dos operandos sean enteros, y lo comprueban antes de llamar a EfectuaOpBinaria.

- Comparaciones ('>', '>=', '<', '<=', '==' y '!='): todas las comparaciones se realizan a través de la función EfectuaTest, que genera el código por flujo de control: genera un valor resultado y almacena un uno en él. Llama entonces a la función EfectuaComparacion, que emite la instrucción correspondiente dependiendo del tipo y localización de los valores. Por último emite el salto adecuado para continuar. Si la comparación no se cumple, se almacena un cero en el resultado. El esquema es:

```

resultado ← 1
<efectúa la comparación>
J<cn> ET_COMP
resultado ← 0
ET_COMP:
```

En este esquema, <cn> es una de las condiciones de salto (G, GE, L, LE, EQ, NE) que coincide con la comparación que se está haciendo.

- Conjunción copulativa ('&'): se efectúa por flujo de control. Se crea el valor resultado y se hace que valga cero. Se compara el primer elemento con cero. Si lo es, se salta al final.

Si no lo es, se compara el segundo elemento con cero. Si lo es, se salta al final. Si no lo es se almacena un uno en el resultado.

Esquemáticamente:

```

resultado ← 0
<compara e1 con 0>
JEQ ET_FALSO
<compara e2 con 0>
JEQ ET_FALSO
resultado ← 1
ET_FALSO:

```

- Conjunción disyuntiva ('||'): al igual que la anterior se realiza por flujo de control: el resultado se inicializa a uno. Se compara el primer elemento con cero y si no es cero se salta al final. Si es cero, se compara el segundo. Si no es cero, se salta al final. Si lo es, se carga un cero en el resultado. De forma algorítmica:

```

resultado ← 1
<compara e1 con 0>
JNE ET_VERDADERO
<compara e2 con 0>
JNE ET_VERDADERO
resultado ← 0
ET_VERDADERO:

```

- Paréntesis: no se genera código. Su única misión es sintáctica.

- Función matemática: se llama a la función `EfectuaFuncionMatematica`, que comprueba que el argumento es real, emite el código necesario y devuelve el valor resultante.

- Función integrada: se trata casi como si fuera una función

normal. La única excepción es que la llamada no se efectúa mediante CALL, sino que es una instrucción propia de la máquina virtual. Por consiguiente no se almacena la dirección de retorno ni se guarda el *Frame Pointer* antiguo (FP). El marco de llamada es, pues, diferente. El registro SP apunta al último parámetro introducido en la pila (que es, en realidad, el primero de la función).

Apéndice A: Otros Programas.

A.- Apéndice A: otros programas.

El sistema LCR incluye, además del compilador (LCR.EXE), un programa de ayuda a la depuración en desarrollo (TRATAMEM.EXE), un desensamblador virtual (DEV.EXE) y un simulador de máquina virtual (en realidad, el propio microprocesador virtual, MPV.EXE).

A.1.- Tratamem.Exe.

Este programa se ha usado durante la fase de desarrollo para comprobar que toda la memoria dinámica que se reserva en el compilador se libera al final, y que no se libera memoria que no se haya reservado.

Como se ha dicho, el compilador LCR con la opción -m (o -M<n-fich>) genera un fichero .MEM que contiene información sobre la memoria dinámica que se ha reservado y la que se ha liberado. Este fichero está estructurado en una serie de líneas que tienen el siguiente formato:

```
T nnnn pppp:pppp ffff llll
```

donde

- T es un carácter que indica el tipo de operación de memoria: reserva (malloc, 'M') o liberación (free, 'F').

- nnnn es el número de bytes reservados o liberados.

- pppp:pppp es la dirección de memoria (segmento:desplazamiento en hexadecimal) que ocupa el bloque reservado o liberado.

- ffff es el nombre del fichero de desarrollo donde se ha efectuado la reserva o liberación.

- llll es la línea del fichero de desarrollo donde se efectuó la operación.

EL programa Tratamem.Exe actúa sobre este fichero emparejando las líneas de reserva con sus correspondientes de liberación (siempre, claro está, que la dirección del bloque sea la misma). Cuando acaba este proceso lista las líneas no emparejadas, que son errores en el manejo de memoria.

La sintaxis de Tratamem.Exe es

```
tratamem [-v] n-fich-mem
```

La opción -v hace que el programa vaya emitiendo mensajes que muestran su funcionamiento.

A.2.- **Dev.Exe.**

El desensamblador virtual (DEV) lee un fichero de código máquina virtual (.CMV) y lo traduce a ensamblador virtual. Esta traducción tiene su salida por pantalla.

La sintaxis de Dev.Exe es

```
dev n-fich-cmv
```

Si la extensión del fichero de código máquina virtual es la estándar (.CMV) se puede omitir en la línea de comandos: Dev.Exe añadirá automáticamente la extensión.

A.3.- **Mpv.Exe.**

Este programa es el microprocesador virtual. Su misión es la de leer una a una las instrucciones contenidas en un fichero de código máquina virtual (.CMV) y ejecutarlas. Para ello, primero carga el código en el segmento de código (cs) y luego lo ejecuta, interactuando con el segmento de datos ds (que, además de los datos, contiene en su zona superior la pila) y con los registros.

La sintaxis de Mpv.Exe es la siguiente:

```
mpv [-v] n-fich-cmv
```

Al igual que en Dev.Exe se puede omitir la extensión .CMV de fichero de código máquina virtual.

La opción -v efectúa un trazado paso a paso de la ejecución.

Este trazo implica la monitorización (antes de la ejecución de cada instrucción) del estado de los registros (FP, SP, PC, Rx) y de la pila.

Apéndice B: Protocolo de adición
de funciones integradas.

B.- Apéndice B: protocolo de adición de funciones integradas.

Como ya se ha dicho, el lenguaje LCR permite que en cualquier momento se le añadan nuevas funciones integradas.

De hecho todas las funciones de control del robot que ya posee el lenguaje son funciones integradas añadidas a la estructura básica.

Las funciones integradas se añaden incluyendo una serie de elementos (declaraciones, definiciones, código) en los lugares adecuados. Como manda el estándar usado en la codificación (ver manual de código), estos lugares se han señalado con una texto de la forma

/*\$

seguido por la palabra FINT y una explicación del elemento que se necesita insertar en esa posición.

Una función integrada sólo se puede utilizar dentro de una expresión. No se puede invocar una función integrada mediante una instrucción LLAMAR_A.

Una función integrada tiene una serie de valores característicos asociados:

- Su representación en el lenguaje o nombre de entrada: es la cadena de caracteres, palabra reservada o token que se utiliza para nombrarla en un programa LCR.

- Su código simbólico: es la constante simbólica que indica el número de token que se utiliza internamente para la

comunicación entre el analizador léxico y el sintáctico.

- Su código de operación: es una constante simbólica de la forma I_xxxx, donde <xxxx> suele ser el nombre de la función. Representa el código de operación que corresponde a la instrucción de código máquina virtual.

- Su mnemónico asociado: es la palabra que se corresponde en el código ensamblador virtual con el código de operación. Por consiguiente es la cadena que se emitirá al fichero de código ensamblador virtual (.COD).

- Su nombre de salida: es la cadena de caracteres que se utilizará para su representación en el fichero de traducción a C (.CTR). Es, por ende, el nombre real de la función C que implementa la operación.

- Su estructura de descripción: es una estructura de datos que contiene una descripción de interfaz con la función y que se usa para comprobaciones semánticas. Contiene por orden los siguientes campos:

- El nombre de salida (de 35 caracteres de longitud máxima).

- El código de operación.

- Un carácter que indica el tipo del valor de retorno. Una función integrada sólo puede retornar un valor entero o uno real.

- Una matriz de MAX_PARS_FINT cadenas de caracteres, cada una de las cuales indica el tipo del parámetro que tiene igual número de orden. La forma de construir estas

cadenas es la misma que para la construcción de los descriptores de tipo: 'P' significa 'puntero_a', 'E' y 'R' significan 'entero' y 'real'. El parámetro siguiente al último es la cadena NULL.

- El número de bytes que ocupan estos parámetros. Recuérdese que un puntero y un entero ocupan dos bytes, y un real cuatro.

El protocolo (conjunto de operaciones) que hay que seguir para añadir una nueva función integrada al lenguaje LCR es el siguiente:

- En primer lugar hay que añadir el nombre de entrada de la función y su código simbólico de token a la lista de palabras reservadas que hay en el fichero LCR_SCAN.L. Esta lista consta de una serie de elementos compuestos por una cadena y un entero. Hay que añadir una línea de la forma

```
{ "n-entr", COD-SIMB }
```

Donde n-entr es la palabra reservada que va a corresponder a esa función integrada y COD-SIMB es la constante simbólica que representa su número de token (se suele poner en mayúsculas, ya que es una macro que se crea con #define).

La lista de palabras reservadas ha de estar en todo momento ordenada alfabéticamente por el nombre de entrada de cada línea. Esta condición es necesaria debido a que se realiza una búsqueda secuencial en ella durante el análisis léxico.

- En segundo lugar se ha de añadir el código simbólico de token antes referido al fichero LCR.Y, en una directiva %token. No hace falta #definir la constante simbólica ni darle ningún valor. De ello se encarga PCYACC durante la creación de la gramática.

- En el fichero OPERS.H hay que añadir una constante simbólica (en la enumeración fintS) con un nombre de la forma FI_xxxx, donde xxxx es el nombre de la nueva función integrada.

- Esa misma constante (FI_xxxx) ha de incluirse en una línea del fichero LCR.Y con la forma

```
COD-SIMB: { $$= FI_xxxx; }
```

Donde COD-SIMB es el token simbólico.

- En los ficheros CMV.H, MPV.H y DEV.H hay que añadir a las enumeraciones correspondientes el código de operación de la función integrada, de la forma I_xxxx.

- En MPV.H, DEV.H y CMV.C hay que incluir el mnemónico asociado en la matriz instrstr.

- En los ficheros MPV.H y DEV.H hay que incluir, además, un cero como número de parámetros de la instrucción de código máquina virtual anterior. Esto se hace en la matriz num_ops.

- En OPERS.C hay que incluir la estructura de descripción de la función integrada, que tiene la forma que se indicó antes. Tal vez haya que modificar las constantes MAX_FINTS y/o MAX_PARS_FINT del fichero OPERS.H.

- Por último, en MPV.C hay que añadir primero el fichero de cabecera correspondiente a la nueva función (a su representación en C) y en segundo lugar el código de ejecución de la nueva función integrada, en una sentencia case de la forma

```
case I_xxxx: <código de ejecución>
```

```
break;
```

Dentro de este código de ejecución se pueden usar una serie de funciones que permiten acceder a los parámetros con los que se ha llamado a la función desde LCR, así como devolver valores como retorno de la función.

El tipo de datos PTRPARS declara un elemento que se puede usar como puntero para ir recopilando los parámetros que se han pasado a la función integrada. Esto se hace de forma parecida al tratamiento de los parámetros múltiples en C (va_arg, va_start, va_end, etc).

En primer lugar se inicializa el puntero con la función
InicPars(&xx)

Donde xx es el nombre del elemento de tipo PTRPARS.

A partir de entonces se pueden recolectar los parámetros: la función ParEnt(&xx) devuelve el siguiente parámetro considerado como un entero (o un puntero). La función ParFlt(&xx) lo devuelve considerado como real.

Para los valores de retorno se usan las funciones FIntRetEnt y FIntRetFlt que aceptan un entero y un real respectivamente y los devuelven como valor de retorno.

Como ejemplo, veremos el código de ejecución para una función integrada llamada PRUEBA que acepta un puntero a real, un entero y un real como parámetros y devuelve un valor real. Su estructura de descripción (en OPERS.C) será por lo tanto la siguiente:

```
{ "Prueba", I_PRUEBA, 'R', { "PR", "E", "R", NULL }, 8 }
```

Y su código de ejecución (en MPV.C) será:

```

case I_PRUEBA:
{
PTRPARS  pp;
int      p2
float    p1, p3, retorno;
unsigned dir_p1;

InicPars( &pp );
dir_p1= ParEnt( &pp );
memcpy( &p1, &ds[dir_p1], 4 ); /* valor actual de p1 */
p2= ParEnt( &pp );
p3= ParFlt( &pp );

retorno= Prueba( &p1, p2, p3 );
/* llamada a la función real (en C) */

memcpy( &ds[dir_p1], &p1, 4 ); /* nuevo valor de p1 */
FIntRetFlt( retorno );
}

```

A modo de ejemplo hay actualmente varias funciones integradas ya definidas en el lenguaje. Son las correspondientes a los subsistemas de dirección, movimiento y cámaras (ver Documento Inicial).

Como resumen, daré una lista de comprobación para la adición de nuevas funciones integradas. Esta tabla se puede usar como protocolo cada vez que se añada una nueva función integrada.

Fichero	Adiciones
1	LCR_SCAN.L Palabras reservadas y códigos de token en la tabla reservadas .

2	LCR.Y	Códigos de token en las directivas %token.
3	OPERS.H	Códigos de función integrada (FI_xxxx) en el enum fints .
4	LCR.Y	Reglas gramaticales (\$\$=FI_xxxx) en la metanoción función_integrada .
5	CMV.H, MPV.H y DEV.H	Códigos de operación (I_xxxx) en el enum instrucciones .
6	CMV.C, MPV.H y DEV.H	Representación textual de las instrucciones (mnemónicos).
7	MPV.H y DEV.H	Número de argumentos de las instrucciones de código máquina virtual (siempre cero).
8	OPERS.C	Estructuras de descripción.
8a	OPERS.H	Opcionalmente, modificar el valor de MAX_FINTS y/o MAX_PARS_FINT.
9	MPV.C	Código de ejecución de las instrucciones de código máquina virtual.

Con esta lista de comprobaciones se finaliza el estudio del protocolo de adición de nuevas funciones integradas al lenguaje LCR.