Lenguajes de Programación

Al desarrollarse las primeras computadoras electrónicas, se vio la necesidad de programarlas, es decir, de almacenar en memoria la información sobre la tarea que iban a ejecutar. Las primeras se usaban como calculadoras simples; se les indicaban los pasos de cálculo, uno por uno.

John Von Neumann desarrolló el modelo que lleva su nombre, para describir este concepto de "programa almacenado". En este modelo, se tiene una abstracción de la memoria como un conjunto de celdas, que almacenan simplemente números. Estos números pueden representar dos cosas: los datos, sobre los que va a trabajar el programa; o bien, el programa en sí.

¿Cómo es que describimos un programa como números? Se tenía el problema de representar las acciones que iba a realizar la computadora, y que la memoria, al estar compuesta por switches correspondientes al concepto de bit, solamente nos permitía almacenar números binarios.

La solución que se tomó fue la siguiente: a cada acción que sea capaz de realizar nuestra computadora, asociarle un número, que será su código de operación (opcode) . Por ejemplo, una calculadora programable simple podría asignar los opcodes :

1 = SUMA, 2 = RESTA, 3 = MULTIPLICA, 4 = DIVIDE.

Supongamos que queremos realizar la operación 5 * 3 + 2, en la calculadora descrita arriba. En memoria, podríamos "escribir" el programa de la siguiente forma:

Localidad Opcode Significado Comentario 0 5 5 En esta localidad, tenemos el primer número de la fórmula 1 3 * En esta localidad, tenemos el opcode que representa la multiplicación. 2 3 3 En esta localidad, tenemos el segundo número de la fórmula 3 1 + En esta localidad, tenemos el opcode que representa la suma. 4 2 2 En esta localidad, tenemos el último número de la fórmula

Podemos ver que con esta representación, es simple expresar las operaciones de las que es capaz el hardware (en este caso, nuestra calculadora imaginaria), en la memoria.

La descripción y uso de los opcodes es lo que llamamos lenguaje de máquina . Es decir, la lista de códigos que la máquina va a interpretar como instrucciones, describe las capacidades de programación que tenemos de ella; es el lenguaje más primitivo, depende directamente del hardware, y requiere del programador que conozca el funcionamiento de la máquina al más bajo nivel.

los lenguajes más primitivos fueron los lenguajes de máquina. Esto, ya que el hardware se desarrolló antes del software, y además cualquier software finalmente tiene que expresarse en el lenguaje que maneja el hardware.

La programación en esos momentos era sumamente tediosa, pues el programador tenía que "bajarse" al nivel de la máquina y decirle, paso a pasito, cada punto de la tarea que tenía que realizar. Además, debía expresarlo en forma numérica; y por supuesto, este proceso era propenso a errores, con lo que la productividad del programador era muy limitada. Sin embargo, hay que recordar que en estos momentos, simplemente aún no existía alternativa.

El primer gran avance que se dio, como ya se comentó, fue la abstracción dada por el Lenguaje Ensamblador, y con él, el nacimiento de las primeras herramientas automáticas para generar el código máquina. Esto redujo

los errores triviales, como podía ser el número que correspondía a una operación, que son sumamente engorrosos y difíciles de detectar, pero fáciles de cometer. Sin embargo, aún aquí es fácil para el programador perderse y cometer errores de lógica, pues debe bajar al nivel de la forma en que trabaja el CPU, y entender bien todo lo que sucede dentro de él.

Con el desarrollo en los 50s y 60s de algoritmos de más elevado nivel, y el aumento de poder del hardware, empezaron a entrar al uso de computadoras científicos de otras ramas; ellos conocían mucho de Física, Química y otras ramas similares, pero no de Computación, y por supuesto, les era sumamente complicado trabajar con lenguaje Ensamblador en vez de fórmulas. Así, nació el concepto de Lenguaje de Alto Nivel, con el primer compilador de FORTRAN (FORmula TRANslation), que, como su nombre indica, inició como un "simple" esfuerzo de traducir un lenguaje de fórmulas, al lenguaje ensamblador y por consiguiente al lenguaje de máquina. A partir de FORTRAN, se han desarrollado innumerables lenguajes, que siguen el mismo concepto: buscar la mayor abstracción posible, y facilitar la vida al programador, aumentando la productividad, encargándose los compiladores o intérpretes de traducir el lenguaje de alto nivel, al lenguaje de computadora.

Hay que notar la existencia de lenguajes que combinan características de los de alto nivel y los de bajo nivel (es decir, Ensamblador). Mi ejemplo favorito es C: contiene estructuras de programación de alto nivel, y la facilidad de usar librerías que también son características de alto nivel; sin embargo, fue diseñado con muy pocas instrucciones, las cuales son sumamente sencillas, fáciles de traducir al lenguaje de la máquina; y requiere de un entendimiento apropiado de cómo funciona la máquina, el uso de la memoria, etcétera. Por ello, muchas personas consideramos a lenguajes como C (que fue diseñado para hacer sistemas operativos), lenguajes de nivel medio.

Java

El lenguaje de programación Java, fue diseñado por la compañía Sun Microsystems Inc, con el propósito de crear un lenguaje que pudiera funcionar en redes computacionales heterogéneas (redes de computadoras formadas por más de un tipo de computadora, ya sean PC, MAC's, estaciones de trabajo, etc.),y que fuera independiente de la plataforma en la que se vaya a ejecutar. Esto significa que un programa de Java puede ejecutarse en cualquier máquina o plataforma. El lenguaje fue diseñado con las siguientes características en mente:

- Simple. Elimina la complejidad de los lenguajes como "C" y da paso al contexto de los lenguajes modernos orientados a objetos. Orientado a Objetos. La filosofía de programación orientada a objetos es diferente a la programación convencional.
- Familiar. Como la mayoría de los programadores están acostumbrados a programar en C o en C++, el sintaxis de Java es muy similar al de estos.
- Robusto. El sistema de Java maneja la memoria de la computadora por ti. No te tienes que preocupar por apuntadores, memoria que no se esté utilizando, etc. Java realiza todo esto sin necesidad de que uno se lo indique.
- Seguro. El sistema de Java tiene ciertas políticas que evitan se puedan codificar virus con este lenguaje. Existen muchas restricciones, especialmente para los applets, que limitan lo que se puede y no puede hacer con los recursos críticos de una computadora.
- Portable. Como el código compilado de Java (conocido como byte code) es interpretado, un programa compilado de Java puede ser utilizado por cualquier computadora que tenga implementado el interprete de Java.
- Independiente a la arquitectura. Al compilar un programa en Java, el código resultante un tipo de código binario conocido como byte code. Este códido es interpretado por diferentes computadoras de igual manera, solamente hay que implementar un intérprete para cada plataforma. De esa manera Java

logra ser un lenguaje que no depende de una arquitectura computacional definida.
- Multithreaded. Un lenguaje que soporta multiples threads es un lenguaje que puede ejecutar diferentes líneas de código al mismo tiempo.
- Interpretado. Java corre en máquina virtual, por lo tanto es interpretado.
- Dinámico. Java no requiere que compiles todas las clases de un programa para que este funcione. Si realizas una modificación a una clase Java se encarga de realizar un Dynamic Bynding o un Dynamic Loading para encontrar las clases.

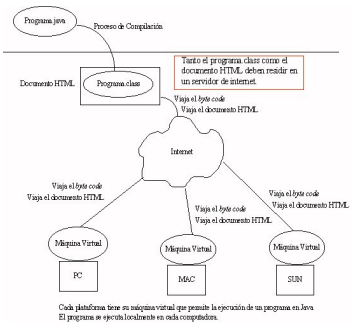
Java puede funcionar como una aplicación sola o como un "applet", que es un pequeño programa hecho en Java. Los applets de Java se pueden "pegar" a una página de Web (HTML), y con esto puedes tener un programa que cualquier persona que tenga un browser compatible podrá usar.

Nota:Diferencia entre Java y CGI La diferencia es esencialmente simple, un CGI se ejecuta en el servidor mientras que un programa en Java se ejecuta en la máquina del usuario.

Java funciona de la siguiente manera: El compilador de Java deja el programa en un Pseudo−código (no es código maquinal) y luego el intérprete de Java ejecuta el programa (lo que se conoce como el "Java Virtual Machine"). Por eso Java es multiplataforma, existe un intérprete para cada máquina diferente. ***Nota: El código maquinal es el código binario que la computadora entiende y puede ejecutar.***

Para entender bien como funciona un applet de Java vean el siguiente ejemplo:

- Existe un código de Java en un servidor de Web. (Los códigos de Java se caracterizan por tener la extensión *.class).
- Una persona en Internet, con un browser compatible con Java, realiza una conección al servidor.
- El servidor envía el documento HTML y el código en Java (*.class).
- En la computadora del usuario remoto llegan ambos, y la Máquina Virtual de Java, que está en el browser, transforma el código Java en un código que entienda la máquina local y se ejecuta el programa dentro de la página de Web.
- Si el usuario realiza otra conexión a otro URL o se sale del browser, el programa se deja de ejecutar y en la computadora no queda rastro de el.



Ejemplo de tutorial de Java:

En Java hay tres tipos de comentarios:

// comentarios para una sola línea

/* comentarios de una o

más líneas

*/

/** comentario de documentación, de una o más líneas

*/

Los dos primeros tipos de comentarios son los que todo programador conoce y se utilizan del mismo modo. Los comentarios de documentación, colocados inmediatamente antes de una declaración (de variable o función), indican que ese comentario ha de ser colocado en la documentación que se genera automáticamente cuando se utiliza la herramienta de Java, javadoc. Dichos comentarios sirven como descripción del elemento declarado permitiendo generar una documentación de nuestras clases escrita al mismo tiempo que se genera el código.

En este tipo de comentario para documentación, se permite la introducción de algunos tokens o palabras clave, que harán que la información que les sigue aparezca de forma diferente al resto en la documentación.

Identificadores

Los identificadores nombran variables, funciones, clases y objetos; cualquier cosa que el programador necesite identificar o usar.

En Java, un identificador comienza con una letra, un subrayado (_) o un símbolo de dólar ($). Los siguientes caracteres pueden ser letras o dígitos. Se distinguen las mayúsculas de las minúsculas y no hay longitud máxima.

Serían identificadores válidos:

identificador

nombre_usuario

Nombre_Usuario

_variable_del_sistema

$transaccion

y su uso sería, por ejemplo:

int contador_principal;

char _lista_de_ficheros;

float $cantidad_en_Ptas;

Unix

Ejemplo de Unix:

No todo el "árbol" de directorios está compuesto por directorios de usuario. Existen muchos de ellos que son

de uso general o del propio sistema y con los que habrá que familiarizarse. Los más importantes son:

/

El raíz, del que "cuelgan" todos.

/bin y /usr/bin

Contienen comandos UNIX ejecutables.

/etc

Es quizá el directorio más importante. Contiene ficheros de datos y configuración del sistema, el fichero de password, configuración de terminales, red, etc (de ahí su nombre).

/dev

Ficheros de dispositivos E/S.

/usr/man

Manual

/tmp

Directorio para arreglos temporales. TODOS los usuarios pueden leer y escribir en él.

C

C es un lenguaje de programación diseñado por Dennis Ritchie, de los Laboratorios Bell, y

se instaló en un PDP−11 en 1972; se diseñó para ser el lenguaje de los Sistemas Operativos

UNIX1. A su vez, UNIX es un Sistema Operativo desarrollado por Ken Thompson, quién

utilizó el lenguaje ensamblador y un lenguaje llamado B para producir las versiones originales de UNIX, en 1970. C se inventó para superar las limitaciones de B.

C es un lenguaje maduro de propósitos generales que se desarrolló a partir de estas raíces;

su definición aparece en 1978 en el apéndice ``C Reference Manual" del libro The C

Programming Language, de Brian W. Kernighan y Dennis M. Ritchie (Englewood Cliffs,

Nueva Jersey, Prentice−Hall 1978), pero el estándar recomendable más reciente apareció en

junio de 1983, en el documento de los Laboratorios Bell titulado The C Programming

Language−Reference Manual, escrito por Dennis M. Ritchie

Un programa en C

Generalizando, un programa en C consta de tres secciones. La primera sección es donde van todos los ``headers". Estos ``headers" son comúnmente los ``#define" y los ``#include". Como segunda sección se tienen las ``funciones". Al igual que Pascal, en C todas las funciones que se van a ocupar en el programa deben ir antes que la función principal (main()). Declarando las funciones a ocupar al principio del programa, se logra que la función principal esté antes que el resto de las funciones. Ahora, solo se habla de funciones ya que en C no existen los procedimientos.

Y como última sección se tiene a la función principal, llamada main. Cuando se ejecuta el programa, lo primero que se ejecuta es esta función, y de ahí sigue el resto del programa.

Los símbolos { y } indican ``begin" y ``end" respectivamente. Si en una función o en un ciclo while, por ejemplo, su contenido es de solamente una línea, no es necesario usar ``llaves" ({ }), en caso contrario es obligación usarlos.

Ejemplo de un programa en C

```
/*Programa ejemplo que despliega el contenido de "ROL" en pantalla*/

#include <stdio.h>

#define ROL "9274002−1"

despliega_rol() {

printf("Mi rol es : \%s\n", ROL);

}

void main() {

despliega_rol();

}

/* Fin programa */
```

Pascal

Pascal es un lenguaje de programación de alto nivel de propósito general; esto es, se puede utilizar para escribir programas para fines científicos y comerciales.

El lenguaje de programación Pascal fue desarrollado por el profesor Niklaus (Nicolás) Wirth en Zurich, Zuiza, al final de los años 1960s y principios de los 70s. Wirth diseñó este lenguaje para que fuese un buen primer lenguaje de programación para personas comenzando a aprender a programar. Pascal tiene un número relativamente pequeño de conceptos para aprender y dominar. Su diseño facilita escribir programas usando un estilo que está generalmente aceptado como práctica estándar de programación buena. Otra de las metas del diseño de Wirth era la implementación fácil. Él diseñó un lenguaje para el cual fuese fácil escribir un compilador para un nuevo tipo de computadora.

```
program Sorting;
```

{

Este programa lee un natural y una secuencia de N caracteres de la entrada estandar; construye un indice para ordenarlos de menor a mayor e imprime en la salida la secuencia ordenada.

}

```
uses CRT;

Const Max = 10;

Espacio = ' ';

Enter = chr (13);

type Indice = 1..Max;

Cantidad= 0..Max;

SecOfChar = record

elems : array [Indice] of char;

ult : Cantidad;

end;

SecOfInd = record

elems : array [Indice] of Indice;

ult : Cantidad;

end;

Natural = 0..MaxInt;

function PosMin (idx: SecOfInd; i: Indice; s: SecOfChar): Cantidad;

{ Devuelve la posicion en el indice idx del menor caracter en s, para

las posiciones >= i. }

var j: Indice;

pm: Cantidad;

begin

if i > idx.ult then

pm := 0
```

```
else begin
pm := i;
for j := i+1 to idx.ult do
if s.elems[idx.elems[j]] < s.elems[idx.elems[pm]] then
pm := j;
end;
PosMin := pm;
end;
procedure Swap (var idx: SecOfInd; i,j: Indice);
{ Intercambia las posiciones i j en idx. }
var tmp: Indice;
begin
if (i<=idx.ult) and (j<=idx.ult) then begin
tmp := idx.elems[i];
idx.elems[i] := idx.elems[j];
idx.elems[j] := tmp;
end;
end;
procedure InicInds (var idx: SecOfInd; cant: Indice);
{ Construye la secuencia de indices 1,2,3,...,n. Sera el indice
inicial para el ordenamiento de una secuencia de caracteres
c1,c2,...,cn. }
var n: Natural;
begin
n := cant;
idx.ult := n;
```

```
while n > 0 do begin
idx.elems [n] := n;
n := n−1;
end;
end;
procedure InicSecChar (var s: SecOfChar);
{ Devuelve la secuencia vacia. }
begin
s.ult := 0;
end;
function Llena (s: SecOfChar): Boolean;
begin
Llena := s.ult = Max;
end;
{ PRE: not Llena(s) }
procedure InsCar (var s: SecOfChar; c: char);
{ Inserta el caracter c en la secuencia s }
begin
s.ult := s.ult + 1;
s.elems [s.ult] := c;
end;
procedure IndSelSort (s: SecOfChar; var ind: SecOfInd);
{ Construye el indice que ordena la secuencia s. Ordena el indice
inicial 1,2, ..., n por el metodo de selection sort }
var i: Indice;
begin
```

```
InicInds (ind, s.ult);
for i := 1 to ind.ult−1 do begin
Swap (ind, i, PosMin (ind, i, s));
end
end;
procedure WriteSorted (idx: SecOfInd; s: SecOfChar);
{ Imprime en la salida estandar la secuencia s ordenada segun el
indice idx }
var i: Indice;
begin
write ('Ordenado: ');
for i := 1 to idx.ult do
write (s.elems[idx.elems[i]],' ');
writeln;
end;
procedure LeerCar (var c: char; var ok: boolean; sep: Char);
{ Lee de la entrada estandar un caracter seguido del caracter sep }
var c1, c2: char;
begin
c := ReadKey; write (c);
c1 := ReadKey; write (c1);
ok := c1 = sep;
end;
procedure LeerSecOfChar (var s: SecOfChar; cant: Natural; var ok: Boolean);
{ Construye una secuencia de cant caracteres provistos por el
procedimeinto LeerCar. Si cant > Max trunca. }
```

```
var bien: Boolean;
i: Natural;
ch, sep: Char;
begin
writeln ('Ingrese ',cant, ' caracteres separados por blancos. Enter para terminar ');
write (' > ');
InicSecChar (s);
i := 1;
ok := true;
sep := Espacio;
while ok and (i <= cant) and not Llena (s) do begin
if i = cant then sep := Enter;
LeerCar (ch, bien, sep);
i := i+1;
ok := ok and bien;
if ok then
InsCar (s, ch);
end;
end;
procedure LeerCant (var n: Natural);
{ Lee de la entrada estandar un natural <= Max }
begin
repeat
writeln ('Ingrese cantidad de caracteres (<=',Max,')');
write (' > ');
readln (n);
```

```
until n <= Max;
end;
procedure Continuar (var seguir: Boolean);
var car: Char;
begin
writeln;
writeln ('Otro ? (s/n)');
write (' > ');
car := ReadKey;
writeln (car);
seguir := car in ['s','S'];
end;
var cant: Natural;
cars: SecOfChar;
inds: SecOfInd;
seguir, ok: boolean;
begin
repeat
ClrScr;
LeerCant (cant);
LeerSecOfChar (cars, cant, ok);
if ok then begin
IndSelSort (cars, inds);
writeln;
WriteSorted (inds, cars);
end
```

```
else begin

writeln;

writeln ('Error en los datos');

end;

Continuar (seguir);

until not seguir;

end.
```

QBasic

Qbasic es un lenguaje de alto nivel, el cual consiste en instrucciones que los humanos pueden relacionar y entender. El compilador de Qbasic se encarga de traducir el mismo a lenguaje de máquina.

Un programa es una secuencia de instrucciones. El proceso de ejecutar esas instrucciones se llama correr el programa. Los programas contienen las funciones de entrada, procesamiento y salida. La persona que resuelve problemas mediante escribir programas en la computadora se conoce como programador. Después de analizar el problema y desarrollar un plan para solucionarlo, escribe y prueba el programa que instruye a la computadora como llevar a cabo el plan. El procedimiento que realiza el programador se define como "problem solving". Pero es necesario especificar que un programador y un usuario no son lo mismo. Un usuario es cualquier persona que use el programa.

Ejemplo de qbasic, para hacer una calculadora

```
DIM total AS DOUBLE

DIM number AS DOUBLE

DIM secondNumber AS DOUBLE

DIM more AS STRING

DIM moreNumbers AS STRING

DIM operation AS STRING

total = 0

more = "y"

moreNumbers = "c"

CLS

WHILE more = "y"
```

```
INPUT "Enter the first number"; number

total = number

WHILE moreNumbers = "c"

COLOR 14

PRINT "The total is:"; total

COLOR 7

PRINT "Select an operation"

COLOR 2

PRINT "(+)"

COLOR 5

PRINT "(−)"

COLOR 1

PRINT "(x)"

COLOR 4

INPUT "(/)"; operation

COLOR 7

CLS

IF operation = "+" THEN

REM where we do additions

PRINT "Enter the number to Add to"; total

INPUT secondNumber

total = secondNumber + total

COLOR 14

PRINT "The total is now:"; total

COLOR 7

ELSE
```

```
IF operation = "−" THEN
REM subtraction
PRINT "Enter the number to Subtract from"; total
INPUT secondNumber
total = total − secondNumber
COLOR 14
PRINT "The total is now:"; total
COLOR 7
ELSE
IF operation = "x" THEN
REM multiplication
PRINT "Enter the number to Multiply"; total; "by"
INPUT secondNumber
total = secondNumber * total
REM * is the multiplication sign in programs
COLOR 14
PRINT "The total is now:"; total
COLOR 7
ELSE
IF operation = "/" THEN
REM division
PRINT "Enter the number to Divide"; total; "by"
INPUT secondNumber
IF secondNumber = 0 THEN
COLOR 4
PRINT "You cannot divide by zero"
```

```
COLOR 7
ELSE
total = total / secondNumber
REM / is the division sign in programs
END IF
COLOR 14
PRINT "The total is now:"; total
COLOR 7
ELSE
PRINT "you must select an operation"
END IF
END IF
END IF
END IF
INPUT "Do you wish to continue (c) or start with new numbers
(n)";moreNumbers
CLS
WEND
COLOR 14
PRINT "The grand total is:"; total
COLOR 7
INPUT "Do you wish to make more calculations (y − n)"; more
moreNumbers = "c"
REM if we don't put "moreNumbers" back to y, it will always
REM come back to "Do you wish to make more calculations" and never REM ask
for numbers again
```

REM (try it)

total = 0

REM if we don't reset the total to 0, it will just

REM keep on adding to the total

WEND

END

Linux

Linux es una implementación del sistema operativo UNIX (uno más de entre los numerosos clónicos del histórico Unix), pero con la originalidad de ser gratuito y a la vez muy potente, que sale muy bien parado (no pocas veces victorioso) al compararlo con las versiones comerciales para sistemas de mayor envergadura y por tanto teóricamente superiores. Comenzó como proyecto personal del −entonces estudiante− Linus Torvalds, quien tomó como punto de partida otro viejo conocido, el Minix de Andy. S. Tanenbaum (profesor de sistemas operativos que creó su propio sistema operativo Unix en PCs XT para usarlo en su docencia). Actualmente Linus lo sigue desarrollando, pero a estas alturas el principal autor es la red Internet, desde donde una gigantesca familia de programadores y usuarios aportan diariamente su tiempo aumentando sus prestaciones y dando información y soporte técnico mútuo. La versión original −y aun predominante− comenzó para PCs compatibles (Intel 386 y superiores), existiendo también en desarrollo versiones para prácticamente todo tipo de plataformas:

PowerPC <http://www.cs.us.es/archive/linuxppc/>,

Sparc <http://www.geog.ubc.ca/sparclinux.html>,

Alpha <http://www.azstarnet.com/~axplinux>,

Mips <http://www.fnet.fr/linux−mips/>, etc.

De todas ellas la más reciente en este momento es la versión para PowerMac <http://www.mklinux.org> (el PowerPC de Apple) basada en el microkernel Mach 3.0 y de la que ya hay una distribución para desarrolladores avalada directamente por Apple y OSF pero conservando el espíritu (gratuito, de libre distribución, etc) de la version original. Un servidor la acaba de probar hace unos días y se ha llevado una grata sorpresa (aún tendrá muuuchos fallos, pero para ser una primerísima versión y el poco tiempo que lleva en marcha, ha avanzado más de lo que me esperaba).

Ejemplo de linux:

Compilar el Kernel

Dado que un diskette sólo almacena 1.44 Megabytes (1440 Kilobytes) de datos, no puedes el mismo kernel que utilizas al diskette. Primero debes conseguir los fuentes del núcleo y descomprimirlos en /usr/src/linux. Luego ejecuta la siguiente orden desde el directorio

/usr/src/linux:

make config

Configura solamente aquello que realmente necesites. Yo, personalmente, sólo configuro el soporte para "ext2", soporte para la disquetera (floppy disk), y soporte para "PPP". Tus elecciones pueden se diferentes en función de lo que decidas incluir. Ahora introduce el siguiente comando:

make dep; make clean; make zImage

¡make zImage es muy importante! Comprime el kernel definitivo. Después de que termine la compilación, deberás buscar el nuevo núcleo en /usr/src/linux/arch/i386/boot bajo el

nombre de zImage.

El sistema de ficheros: No es solamente un conjunto de ficheros

Ahora hemos de crear el sistema de ficheros (en inglés: filesystem, fs) para el diskette. En vez de copiar los ficheros tal cual directamente al diskette, los comprimiremos antes de copiarlos. Esto nos hará un poco más difícil la faena de modificar todo permanentemente. Primero tecleamos el siguiente

comando:

dd if=/dev/zero of=[DEVICE] bs=1k count=3000

Donde [DEVICE] es "lugar" en el disco duro donde vas a guardar el sistema de ficheros descomprimido. Luego, introduce el siguiente comando y pulsa ENTER, sustituyendo [DEVICE] por el directorio en tu disco duro donde estás guardando el sistema de ficheros descomprimido:

mke2fs −m 0 [DEVICE]

Si make2fs te pregunta si realmente quieres hacer esto (Do you really want to do this?), acepta tecleando "y" (yes).

Después tenemos que montar este sistema de ficheros que hemos creado. Para ello, el núcleo que utilices tiene que permitir "montar ficheros", en otras palabras, ha de tener habilitada la posibilidad de "loopback devices". Para ello has de compilar el núcleo de tu máquina (no el núcleo que hemos creado, sino el de tu propia máquina) con la opción:

Loopback device support (CONFIG_BLK_DEV_LOOP) [M/n/y/?]

bien como módulo (M) o en el mismo núcleo (Y). Si lo compilas como módulo (lo más recomendable) luego tienes que insertar el módulo modprobe loop !No olvides rearrancar la máquina si has tenido que recompilar el núcleo!

mount −t ext2 DEVICE /mnt

Si se queja la orden mount puedes intentar con la siguiente orden:

mount −o loop −t ext2 DEVICE /mnt

Ahora debes copiar todos los ficheros que necesites en el nuevo sistema de ficheros. Primero, ponte en el directorio /mnt, (cd /mnt), y crea los siguientes directorios:

/dev

/pro

/etc

/bin

/lib

/mnt

/usr

Ahora crearemos el directorio /dev tecleando lo siguiente:

cp −dpR /dev /mnt/dev

Si se te acaban los i−nodos del diskette, puedes ir a /mnt/dev y borrar los archivos de dispositivo que no necesites. Cuando acabes de copiar los ficheros necesarios para /dev, ves a /etc. Para estar seguro copia todos los ficheros de /etc a /mnt/etc:

cp −dpR /etc /mnt/etc

Luego copia todo del directorio /lib en /mnt:

cp −dpR /lib /mnt/lib

Para el directorio /bin, copia sólo aquello que creas que necesitas en /mnt/bin.

Copiar todo a tu diskette

Ahora hemos de copiar todo en el/los diskette/s. Para hacer esto, debemos comprimir ahora el sistema de ficheros tecleando las siguientes ordenes:

cd /

umount /mnt

dd if=[DEVICE] bs=1k | gzip −9 > rootfs.gz

Ahora es importante comprobar el tamaño del núcleo. Ponte en /usr/src/linux/arch/i386/boot y teclea "ls −l". Luego divide el tamaño del núcleo entre 1024.

Por ejemplo, si el tamaño es de 250000 bytes, entonces son 245 KB. En adelante, reemplaza [ROOTBEGIN] en las ordenes que aparezca por el número total de kilobytes que has calculado. Ahora copia el kernel al diskette usando el siguiente comando:

dd if=zImage of=/dev/fd0

Este comando grabará el kernel en el diskette. Luego introduce el siguiente comando para que el kernel pueda encontrar la raíz del sistema de ficheros en el diskette.

rdev /dev/fd0 /dev/fd0

Ahora tendrás que hacer un pequeño cálculo en hexadecimal. Suma 4000 al equivalente en hexadecimal de [ROOTBEGIN] (que en nuestro ejemplo es F5). Convierte el resultado a decimal y teclea el siguiente comando, sustituyendo 16629 con el resultado que tú has obtenido:

rdev −r /dev/fd0 16629

Finalmente, teclea lo siguiente para copiar el sistema de ficheros al diskette:

dd if=/rootfs.gz of=/dev/fd0 bs=1k seek=[ROOTBEGIN]

El sistema de ficheros raíz será copiado al diskette justo después del kernel. ¡Ya lo tienes! Para el segundo diskette, el proceso es más fácil. Copia los ficheros que quieras en el diskette. No obstante, para poder usar los ficheros que hay en el segundo disco, tendrás que entrar lo siguiente después de arrancar con el diskette:

mount /dev/fd0 /usr

Ensamblador

Cuando abstraemos los opcodes y los sustituimos por una palabra que sea una clave de su significado, a la cual comúnmente se le conoce como mnemónico , tenemos el concepto de Lenguaje Ensamblador . Así, podemos definir simplemente al Lenguaje Ensamblador de la siguiente forma:

Lenguaje Ensamblador es la primera abstracción del Lenguaje de Máquina , consistente en asociar a los opcodes palabras clave que faciliten su uso por parte del programador

Como se puede ver, el Lenguaje Ensamblador es directamente traducible al Lenguaje de Máquina, y viceversa; simplemente, es una abstracción que facilita su uso para los seres humanos. Por otro lado, la computadora no entiende directamente al Lenguaje Ensamblador; es necesario traducirle a Lenguaje de Máquina. Originalmente, este proceso se hacía a mano, usando para ello hojas donde se escribían tablas de programa similares al ejemplo de la calculadora que vimos arriba . Pero, al ser tan directa la traducción, pronto aparecieron los programas Ensambladores, que son traductores que convierten el código fuente (en Lenguaje Ensamblador) a código objeto (es decir, a Lenguaje de Máquina).

Una característica que hay que resaltar, es que al depender estos lenguajes del hardware, hay un distinto Lenguaje de Máquina (y, por consiguiente, un distinto Lenguaje Ensamblador) para cada CPU. Por ejemplo, podemos mencionar tres lenguajes completamente diferentes, que sin embargo vienen de la aplicación de los conceptos anteriores:

1.Lenguaje Ensamblador de la familia Intel 80x86 2.Lenguaje Ensamblador de la familia Motorola 68000 3.Lenguaje Ensamblador del procesador POWER, usado en las IBM RS/6000.

Tenemos 3 fabricantes distintos, compitiendo entre sí y cada uno aplicando conceptos distintos en la manufactura de sus procesadores, su arquitectura y programación; todos estos aspectos, influyen en que el lenguaje de máquina y ensamblador cambie bastante.

Ventajas y desventajas del Lenguaje Ensamblador

Una vez que hemos visto la evolución de los lenguajes, cabe preguntarse: ¿En estos tiempos "modernos", para qué quiero el Lenguaje Ensamblador?

El proceso de evolución trajo consigo algunas desventajas, que ahora veremos como las ventajas de usar el Lenguaje Ensamblador, respecto a un lenguaje de alto nivel:

1.Velocidad

2.Eficiencia de tamaño

3.Flexibilidad

Por otro lado, al ser un lenguaje más primitivo, el Ensamblador tiene ciertas desventajas respecto a los lenguajes de alto nivel:

1.Tiempo de programación 2.Programas fuente grandes 3.Peligro de afectar recursos inesperadamente 4.Falta de portabilidad

Velocidad

El proceso de traducción que realizan los intérpretes, implica un proceso de cómputo adicional al que el programador quiere realizar. Por ello, nos encontraremos con que un intérprete es siempre más lento que realizar la misma acción en Lenguaje Ensamblador, simplemente porque tiene el costo adicional de estar traduciendo el programa, cada vez que lo ejecutamos.

De ahí nacieron los compiladores, que son mucho más rápidos que los intérpretes, pues hacen la traducción una vez y dejan el código objeto, que ya es Lenguaje de Máquina, y se puede ejecutar muy rápidamente. Aunque el proceso de traducción es más complejo y costoso que el de ensamblar un programa, normalmente podemos despreciarlo, contra las ventajas de codificar el programa más rápidamente.

Sin embargo, la mayor parte de las veces, el código generado por un compilador es menos eficiente que el código equivalente que un programador escribiría. La razón es que el compilador no tiene tanta inteligencia, y requiere ser capaz de crear código genérico, que sirva tanto para un programa como para otro; en cambio, un programador humano puede aprovechar las características específicas del problema, reduciendo la generalidad pero al mismo tiempo, no desperdicia ninguna instrucción, no hace ningún proceso que no sea necesario.

Para darnos una idea, en una PC, y suponiendo que todos son buenos programadores, un programa para ordenar una lista tardará cerca de 20 veces más en Visual Basic (un intérprete), y 2 veces más en C (un compilador), que el equivalente en Ensamblador.

Por ello, cuando es crítica la velocidad del programa, Ensamblador se vuelve un candidato lógico como lenguaje.

Ahora bien, esto no es un absoluto; un programa bien hecho en C puede ser muchas veces más rápido que un programa mal hecho en Ensamblador; sigue siendo sumamente importante la elección apropiada de algoritmos y estructuras de datos. Por ello, se recomienda buscar optimizar primero estos aspectos, en el lenguaje que se desee, y solamente usar Ensamblador cuando se requiere más optimización y no se puede lograr por estos medios.

Tamaño

Por las mismas razones que vimos en el aspecto de velocidad, los compiladores e intérpretes generan más código máquina del necesario; por ello, el programa ejecutable crece. Así, cuando es importante reducir el tamaño del ejecutable, mejorando el uso de la memoria y teniendo también beneficios en velocidad, puede convenir usar el lenguaje Ensamblador. Entre los programas que es crítico el uso mínimo de memoria,

tenemos a los virus y manejadores de dispositivos (drivers). Muchos de ellos, por supuesto, están escritos en lenguaje Ensamblador.

Flexibilidad

Las razones anteriores son cuestión de grado: podemos hacer las cosas en otro lenguaje, pero queremos hacerlas más eficientemente. Pero todos los lenguajes de alto nivel tienen limitantes en el control; al hacer abstracciones, limitan su propia capacidad. Es decir, existen tareas que la máquina puede hacer, pero que un lenguaje de alto nivel no permite. Por ejemplo, en Visual Basic no es posible cambiar la resolución del monitor a medio programa; es una limitante, impuesta por la abstracción del GUI Windows. En cambio, en ensamblador es sumamente sencillo, pues tenemos el acceso directo al hardware del monitor.

Resumiendo, la flexibilidad consiste en reconocer el hecho de que

Todo lo que puede hacerse con una máquina, puede hacerse en el lenguaje ensamblador de esta máquina; los lenguajes de alto nivel tienen en una u otra forma limitantes para explotar al máximo los recursos de la máquina.

Tiempo de programación

Al ser de bajo nivel, el Lenguaje Ensamblador requiere más instrucciones para realizar el mismo proceso, en comparación con un lenguaje de alto nivel. Por otro lado, requiere de más cuidado por parte del programador, pues es propenso a que los errores de lógica se reflejen más fuertemente en la ejecución.

Por todo esto, es más lento el desarrollo de programas comparables en Lenguaje Ensamblador que en un lenguaje de alto nivel, pues el programador goza de una menor abstracción.

Programas fuente grandes

Por las mismas razones que aumenta el tiempo, crecen los programas fuentes; simplemente, requerimos más instrucciones primitivas para describir procesos equivalentes. Esto es una desventaja porque dificulta el mantenimiento de los programas, y nuevamente reduce la productividad de los programadores.

Peligro de afectar recursos inesperadamente

Tenemos la ventaja de que todo lo que se puede hacer en la máquina, se puede hacer con el Lenguaje Ensamblador (flexibilidad). El problema es que todo error que podamos cometer, o todo riesgo que podamos tener, podemos tenerlo también en este Lenguaje. Dicho de otra forma, tener mucho poder es útil pero también es peligroso.

En la vida práctica, afortunadamente no ocurre mucho; sin embargo, al programar en este lenguaje verán que es mucho más común que la máquina se "cuelgue", "bloquee" o "se le vaya el avión"; y que se reinicialize. ¿Por qué?, porque con este lenguaje es perfectamente posible (y sencillo) realizar secuencias de instrucciones inválidas, que normalmente no aparecen al usar un lenguaje de alto nivel.

En ciertos casos extremos, puede llegarse a sobreescribir información del CMOS de la máquina (no he visto efectos más riesgosos); pero, si no la conservamos, esto puede causar que dejemos de "ver" el disco duro, junto con toda su información.

Falta de portabilidad

Como ya se mencionó, existe un lenguaje ensamblador para cada máquina; por ello, evidentemente no es una

selección apropiada de lenguaje cuando deseamos codificar en una máquina y luego llevar los programas a otros sistemas operativos o modelos de computadoras. Si bien esto es un problema general a todos los lenguajes, es mucho más notorio en ensamblador: yo puedo reutilizar un 90% o más del código que desarrollo en "C", en una PC, al llevarlo a una RS/6000 con UNIX, y lo mismo si después lo llevo a una Macintosh, siempre y cuando esté bien hecho y siga los estándares de "C", y los principios de la programación estructurada. En cambio, si escribimos el programa en Ensamblador de la PC, por bien que lo desarrollemos y muchos estándares que sigamos, tendremos prácticamente que reescribir el 100 % del código al llevarlo a UNIX, y otra vez lo mismo al llevarlo a Mac.

Bibliografía:

Peter Abel. IBM PC Assembly Language and Programming. Fourth Edition. Prentice Hall. 1997.

http://sunsite.dcc.uchile.cl/webmastercl/mirrors/javatutor/

http://highland.dit.upm.es:8000/UNIX/movs1/2dirsist.html

http://docs.inf.utfsm.cl/pub/DI/labsw/manual_c/node1.html

http://docs.inf.utfsm.cl/pub/DI/labsw/manual_c/node2.html

http://docs.inf.utfsm.cl/pub/DI/labsw/manual_c/node3.html

http://www.fing.edu.uy/~yemurenk/indexsort.html

http://www.cs.us.es/archive/LinuxFocus/Castellano/May1998/article11.html