Programas de ordenador (software)

Jorge Juan Chico <jjchico@dte.us.es>, Julián Viejo Cortés <julian@dte.us.es> 2011 – 2020
Departamento de Tecnología Electrónica
Universidad de Sevilla

Objetivos

- Conocer el tipo de instrucciones que ejecuta la CPU de un ordenador
- Comprender los procesos de compilación y ensamblado de los programas de ordenador
- Conocer la forma en que se organiza el software para crear programas complejos

Contenidos

- Introducción al software
- Lenguaje máquina y ensamblador
- Lenguajes compilados
- Lenguajes interpretados
- Jerarquía del software

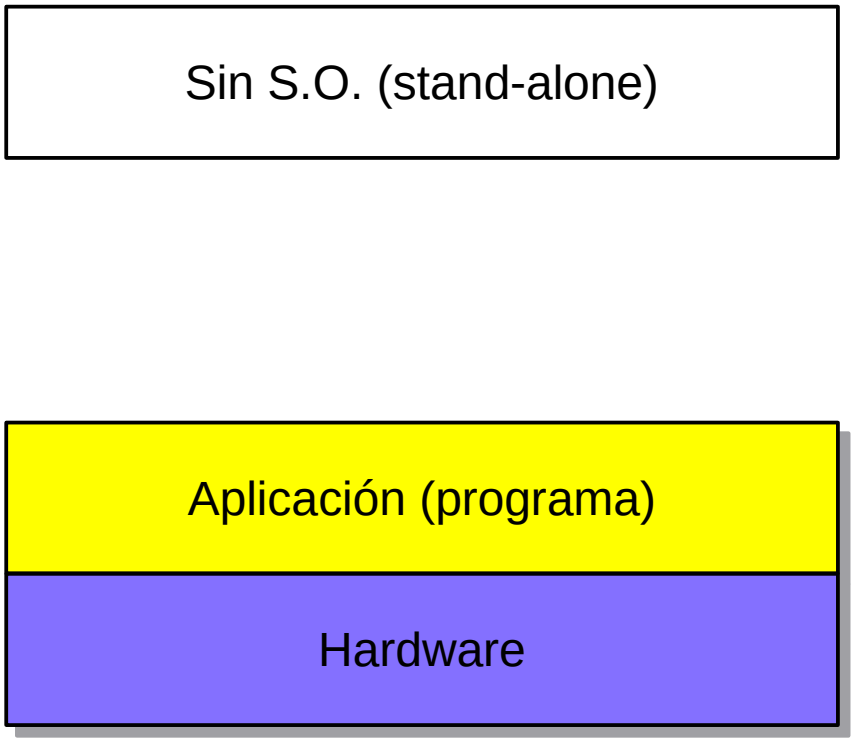
Bibliografía

- Programming from the Ground Up
  - Introducción a la programación en ensamblador con énfasis en el funcionamiento del software y el computador.
  - Leer capítulos 1 a 3
- Apuntes y ejemplos ensamblador x86
  - Revisar y probar los ejemplos
  - La tarea del tema se basará en estos ejemplos
- x86 Assembly
  - Referencia y ejemplos de programación para la arquitectura x86
  - Usar como referencia

Defininiones

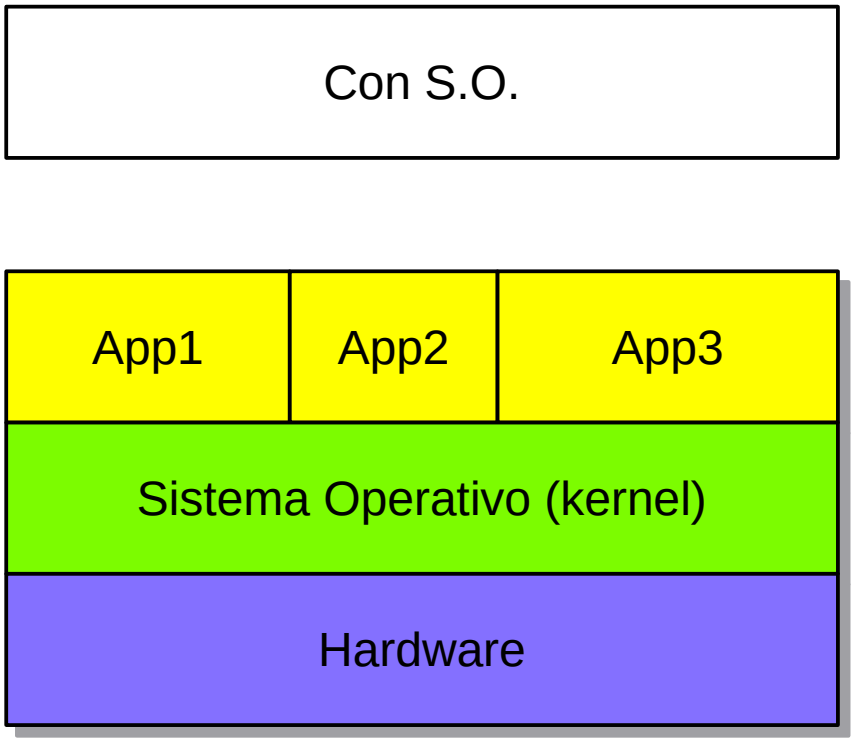
- Programa de ordenador
  - Definición de tareas, procedimientos y conjuntos de datos para ser procesador por un ordenador.
  - Se describen mediante lenguajes de programación.
- Software (~programas de ordenador)
  - Conjunto de programas de ordenador, en sentido general o como conjunto específico. Ej:
    - Software de sistemas
    - Software para diseño gráfico
    - Conjunto de software instalado en un ordenador
- Sistema operativo (kernel)
  - Programa que gestiona la operación del ordenador
  - Proporciona funciones básicas para el resto de los programas: llamadas al sistema
  - Facilita la programación y uso de los ordenadores

Software y sistema operativo



- El programa incluye todas las funciones necesarias
- Complejo: hay que programar todos los detalles
- Alto coste: programar tareas repetitivas
- Conocimiento detallado del hardware

Software y sistema operativo



- El sistema operativo gestiona el hardware y organiza la ejecución de los programas
- Operaciones comunes programadas en el S.O.
  - Llamadas al sistema
- Programación más fácil
- Ejecución de varios programas
- No se necesita conocimiento detallado del hardware

Lenguajes de programación

- Lenguaje máquina
  - Instrucciones en formato binario que el procesador. interpreta y ejecuta.
  - Único software que realmente reconoce el procesador.
- Lenguaje ensamblador
  - Representación simbólica del lenguaje máquina.
  - Facilita la programación del procesador en lenguaje máquina.
- Lenguaje compilado
  - Lenguaje de alto nivel, independiente del procesador.
  - Es convertido en código ensamblador por un compilador.
- Lenguaje interpretado
  - Lenguaje que es ejecutado por otro programa “intérprete”.
  - Menor rendimiento, mayor facilidad de desarrollo.

Contenidos

- Introducción al software
- **Lenguaje máquina y ensamblador**
- Lenguajes compilados
- Lenguajes interpretados
- Jerarquía del software

Lenguaje máquina y ensamblador

- Lenguaje (código) máquina: código binario de las instrucciones que ejecuta la CPU
- Lenguaje ensamblador: representación del lenguaje máquina mediante palabras clave fáciles de entender para el programador

Ensamblador

```
movq $10,  %rax
movq $14,  %rdx
addq %rax, %rdx
movq $60,  %rax
movq %rdx, %rdi
syscall
```

Código máquina

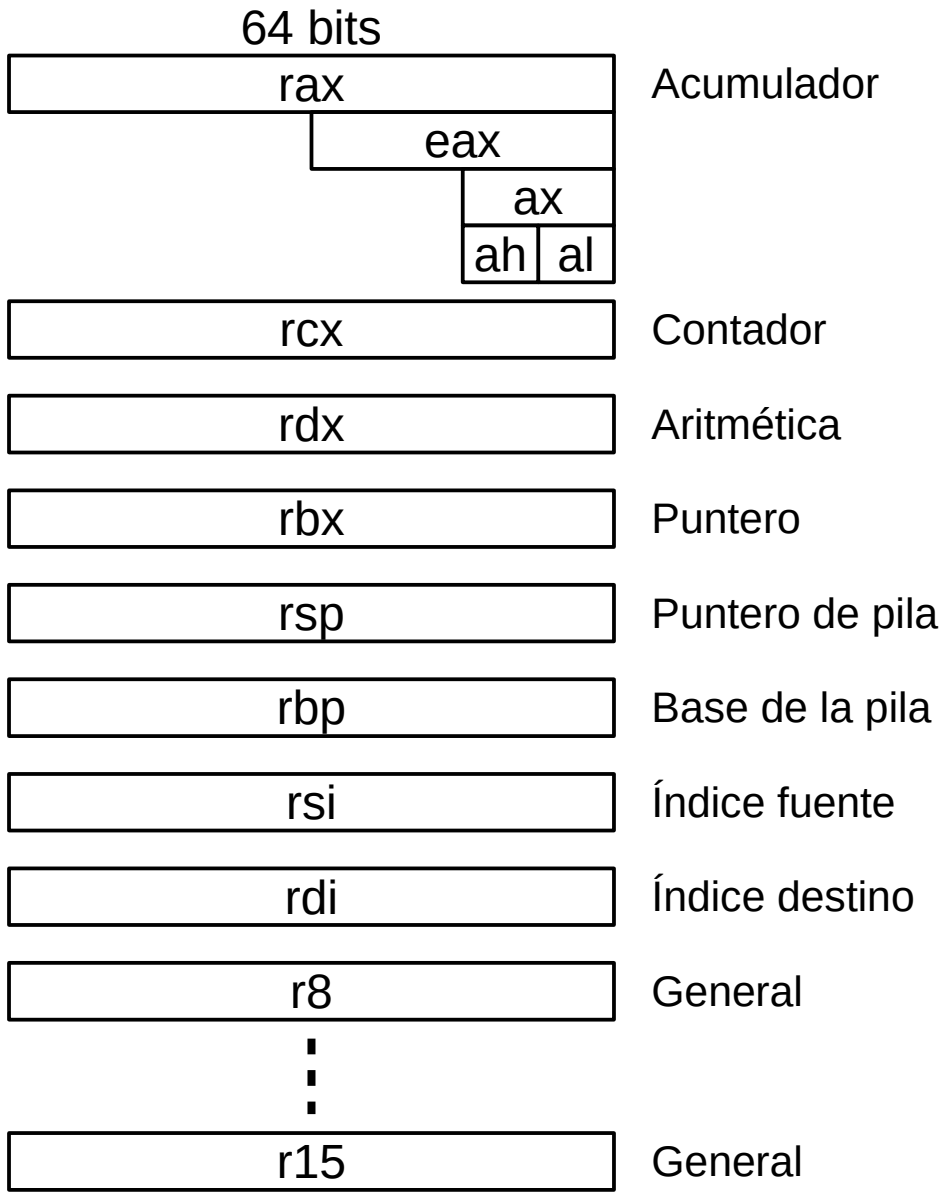
```
400078: 48c7c00a000000
40007f: 48c7c20e000000
400086: 4801c2
400089: 48c7c03c000000
400090: 4889d7
400093: 0f05
```

Programar en ensamblador ¿Qué hay que saber?

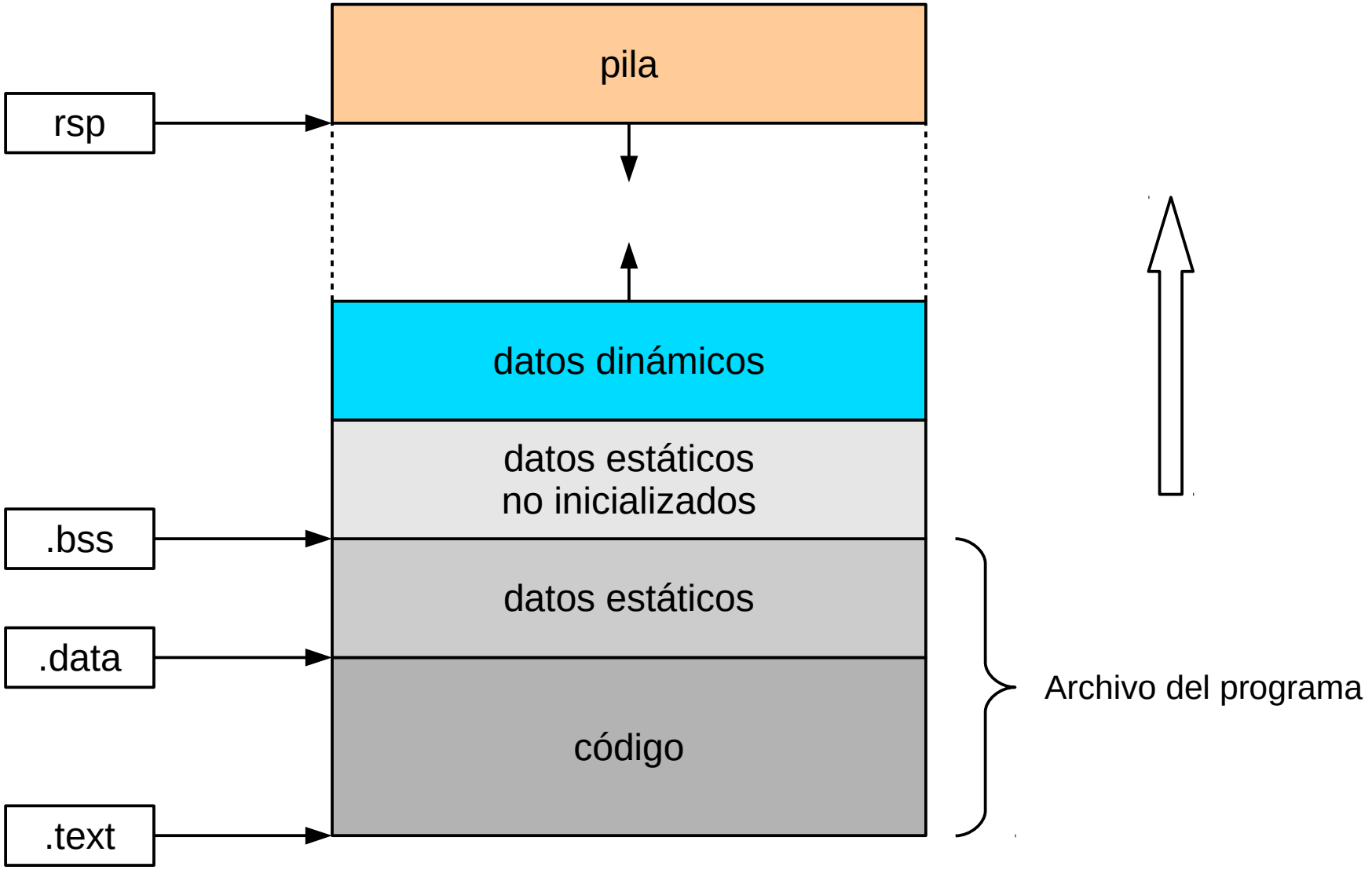
- El conjunto de instrucciones del procesador.
- Los tipos de datos que maneja (32bits, 64bits, etc.).
- La organización de la memoria del procesador y los programas.
- El conjunto de registros disponibles para el programador, y su relación con las instrucciones.
- Métodos para pedir ayuda al sistema operativo
  - Llamadas al sistema
- Métodos para usar funciones de bibliotecas
  - Funciones de la biblioteca de C, etc.
- Herramientas de programación:
  - Ensamblador, enlazador, depurador, etc.

- También hay que ¡saber programar!

Registros propósito general x86



Memoria de un programa



Ensamblador x86. Sintaxis

instrucción

```
<nemónico> <origen>, <destino>
```

números

```
$17       # decimal
$0xa7     # hexadecimal
$0b1001   # binario
```

sufijos instr.

```
b (byte): 8 bits
w (word): 16 bits
l (long): 32 bits
q (quad): 64 bits

movq $7, %rax
addl $7, %eax
cmpw $7, %ax
movb $7, %al
```

direcciones de memoria

```
<desplazamiento>(<rbase>,<ríndice>,<factor>)

dirección = desplazamiento+rbase+ríndice*factor

my_data             # dirección de la etiqueta
(%rbx)              # dirección en %rbx
16(%rbx)            # %rbx + 16
(%rbx,%rcx,8)  # %rbx + %rcx * 8
my_data(,%ecx,4)    # my_data + %ecx * 4
```

directivas ensamb.

```
.section .text
.section .data
.section .bss
.byte 1,2,3
.int 4,5,6
.long 4,5,6
.quad 7,8,9
.ascii “Hello!”
.equ myvar, 24
.lcomm result, 8
```

Ensamblador x86. Instrucciones de ejemplo

```
# Mover datos

movq $7, %rax        # rax ← 7
movq %rax, %rbx      # rbx ← rax
movq %rbx, result    # mem(result) ← rbx
movq (%rbx), %rax    # rax ← mem(%rbx)
movq -8(%rbx), %rax # rax ← mem(%rbx-8)


# Operaciones aritméticas

addq %rdx, %rax      # rax ← rax + rdx
subq $7, %rbp        # rbp ← rbp - 7
incq %rax            # rax ← rax + 1
decq %rdx            # rdx ← rdx - 1
cmpq $7, %rax        # compara: %rax - $7

# Lógicas

andq %rdx, %rax      # rax ← rax AND rdx
orq $7, %rbp         # rbp ← rbp OR 7
xorq %rax            # rax ← rax XOR 1
notq %rdx            # rdx ← NOT rdx


# Desplazamiento de bits

shrq %rax            # a la derecha
shlq %rdx            # a la izquierda
```

```
# Salto

jmp  continue        # incondicional
je   loop            # si igual
jg   calculate       # si mayor
jl   islower         # si menor
jz   iszero          # si cero
jnz  isnotzero       # si no cero
call printf          # llamada a subrutina
ret                  # retornos de subrutina

# Varios

int $0x80       # interrupción software
syscall         # llamada al sistema
pushq %rax      # guardar en la pila
popq %rax       # recuperar de la pila
```

Ensamblador x86
Llamadas al sistema y funciones

- Llamada al sistema
  - Código de la llamada: rax
  - Primeros parámetros: rdi, rsi, rdx, r10, r8, r9
  - Parámetros adicionales: en la pila (push)
  - Llamada: **syscall**
  - Lista: /usr/include/asm/unistd_64.h
  - Manual: man syscalls
- Llamada a funciones (parámetros enteros)
  - al=0 (rax=0)
  - Primeros parámetros: rdi, rsi, rdx, rcx, r8, r9
  - Parámetros adicionales: en la pila (push)
  - Llamada: **call <etiqueta>**
- Biblioteca estándar de C
  - Disponible en todos los sistemas tipo UNIX
  - Preferible a las llamadas al sistema
  - 'man libc'
  - Ej: 'man 3 printf'

Herramientas

- Herramientas básicas: **binutils**
  - Ensamblador (as), enlazador (ld) y utilidades varias
  - https://www.gnu.org/software/binutils/
- Compilador de C y biblioteca estándar de C: **gcc** + **libc**
  - https://www.gnu.org/software/gcc/
  - https://www.gnu.org/software/libc/
- Depurador: **gdb**
  - https://www.gnu.org/software/gdb/
  - Múltiples interfaces gráficos: ddd, eclipse, etc.

Lenguaje ensamblador
Ejemplos

```
# Descripción: suma dos números.

.section .data          # datos del programa
                        # no hay datos en este programa

.section .text          # código del programa

.globl _start           # '_start' es una etiqueta que indica dónde comienza
_start:                 # el programa. Debe ser un símbolo 'global'

    movq $10, %rax      # rax = 10
    movq $14, %rdx      # rdx = 14
    addq %rax, %rdx     # rdx = rdx + rax

    movq $60, %rax      # 'syscall' activa una interrupción que es el
    movq $0, %rdi       # canal de comunicación con el kernel de s.o.
                        # rax indica la función requerida: 60 -> exit
                        # rdi es el valor de retorno del programa
    syscall             # 0 -> todo correcto (ver con 'echo $?')
```

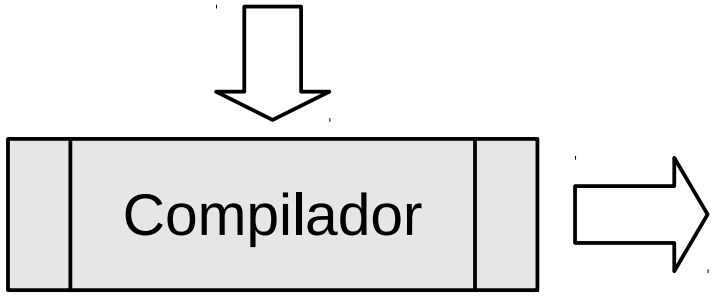
Contenidos

- Introducción al software
- Lenguaje máquina y ensamblador
- **Lenguajes compilados**
- Lenguajes interpretados
- Jerarquía del software

Lenguajes compilados

add.c

```
// Suma dos número en lenguaje C.

long main()
{
    int a, d;

    a = 10;
    d = 14;
    d = a + d;

    printf("a + d = %d\n", d);

    return 0;
}
```



add.s

x86-64

```
.LC0:
     .string   "a + b = %d\n"
     .text
     .globl    main
     .type     main, @function
main:
.LFB0:
     .cfi_startproc
     pushq     %rbp
     .cfi_def_cfa_offset 16
     .cfi_offset 6, -16
     movq %rsp, %rbp
     .cfi_def_cfa_register 6
     subq $16, %rsp
     movl $10, -8(%rbp)
     movl $14, -4(%rbp)
     movl -8(%rbp), %eax
     addl %eax, -4(%rbp)
     movl -4(%rbp), %eax
     movl %eax, %esi
     leaq .LC0(%rip), %rdi
     movl $0, %eax
     call printf@PLT
     movl $0, %eax
     leave
     .cfi_def_cfa 7, 8
     ret
     .cfi_endproc
```

Lenguajes compilados
Comparación arquitecturas

```
// Suma dos número en lenguaje C.

long main()
{
    int a, d;

    a = 10;
    d = 14;
    d = a + d;

    printf("a + d = %d\n", d);

    return 0;
}
```

Lenguajes compilados
Comparación arquitecturas

x86-64

```
main:
  .LFB0:
  pushq   %rbp
  movq    %rsp, %rbp
  subq $16, %rsp
  movl $10, -8(%rbp)
  movl $14, -4(%rbp)
  movl -8(%rbp), %eax
  addl %eax, -4(%rbp)
  movl -4(%rbp), %eax
  movl %eax, %esi
  leaq .LC0(%rip), %rdi
  movl $0, %eax
  call printf@PLT
  movl $0, %eax
  leave
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
```

PC

```
// Suma dos número en lenguaje C.

long main()
{
    int a, d;

    a = 10;
    d = 14;
    d = a + d;

    printf("a + d = %d\n", d);

    return 0;
}
```

Lenguajes compilados
Comparación arquitecturas

x86-64

```
main:
  .LFB0:
  pushq   %rbp
  movq    %rsp, %rbp
  subq $16, %rsp
  movl $10, -8(%rbp)
  movl $14, -4(%rbp)
  movl -8(%rbp), %eax
  addl %eax, -4(%rbp)
  movl -4(%rbp), %eax
  movl %eax, %esi
  leaq .LC0(%rip), %rdi
  movl $0, %eax
  call printf@PLT
  movl $0, %eax
  leave
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
```

PC

armv7-a

```
main:
  push {r7, lr}
  sub sp, sp, #8
  add r7, sp, #0
  movs r3, #10
  str r3, [r7, #4]
  movs r3, #14
  str r3, [r7]
  ldr r2, [r7]
  ldr r3, [r7, #4]
  add r3, r3, r2
  str r3, [r7]
  ldr r1, [r7]
  ldr r3, .L3
.LPIC0:
  add r3, pc
  mov r0, r3
  bl printf(PLT)
  movs r3, #0
  mov r0, r3
  adds r7, r7, #8
  mov sp, r7
  @ sp needed
  pop {r7, pc}
```

```
en lenguaje C.

%d\n", d);
```

R-PI

Lenguajes compilados
Comparación arquitecturas

x86-64

```
main:
  .LFB0:
  pushq   %rbp
  movq    %rsp, %rbp
  subq $16, %rsp
  movl $10, -8(%rbp)
  movl $14, -4(%rbp)
  movl -8(%rbp), %eax
  addl %eax, -4(%rbp)
  movl -4(%rbp), %eax
  movl %eax, %esi
  leaq .LC0(%rip), %rdi
  movl $0, %eax
  call printf@PLT
  movl $0, %eax
  leave
  .cfi_def_cfa 7, 8
  ret
  .cfi_endproc
```

PC

armv7-a

```
main:
  push {r7, lr}
  sub sp, sp, #8
  add r7, sp, #0
  movs r3, #10
  str r3, [r7, #4]
  movs r3, #14
  str r3, [r7]
  ldr r2, [r7]
  ldr r3, [r7, #4]
  add r3, r3, r2
  str r3, [r7]
  ldr r1, [r7]
  ldr r3, .L3
.LPIC0:
  add r3, pc
  mov r0, r3
  bl printf(PLT)
  movs r3, #0
  mov r0, r3
  adds r7, r7, #8
  mov sp, r7
  @ sp needed
  pop {r7, pc}
```

R-PI

avr

```
main:
  push r28
  push r29
  rcall .
  rcall .
  in r28,__SP_L__
  in r29,__SP_H__
.L__stack_usage = 6
  ldi r24,lo8(10)
  ldi r25,0
  std Y+2,r25
  std Y+1,r24
  ldi r24,lo8(14)
  ldi r25,0
  std Y+4,r25
  std Y+3,r24
  ldd r18,Y+3
  ldd r19,Y+4
  ldd r24,Y+1
  ldd r25,Y+2
  add r24,r18
  adc r25,r19
  std Y+4,r25
  std Y+3,r24
  ldd r24,Y+4
  push r24
  ldd r24,Y+3
  push r24
  .
```

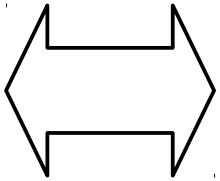
```
  ...
  ldi r24,lo8(.LC0)
  ldi r25,hi8(.LC0)
  mov r24,r25
  push r24
  ldi r24,lo8(.LC0)
  ldi r25,hi8(.LC0)
  push r24
  rcall printf
  pop __tmp_reg__
  pop __tmp_reg__
  pop __tmp_reg__
  pop __tmp_reg__
  ldi r24,0
  ldi r25,0
  ldi r26,0
  ldi r27,0
  mov r22,r24
  mov r23,r25
  mov r24,r26
  mov r25,r27
  pop __tmp_reg__
  pop __tmp_reg__
  pop __tmp_reg__
  pop __tmp_reg__
  pop r29
  pop r28
  ret
```

Arduino

Lenguaje compilado vs ensamblador

add.s

```
# Descripción: suma dos números.

.section .text
.globl _start
_start:
    movq $10, %rax
    movq $14, %rdx
    addq %rax, %rdx
    movq $60, %rax
    movq $0, %rdi
    syscall
```



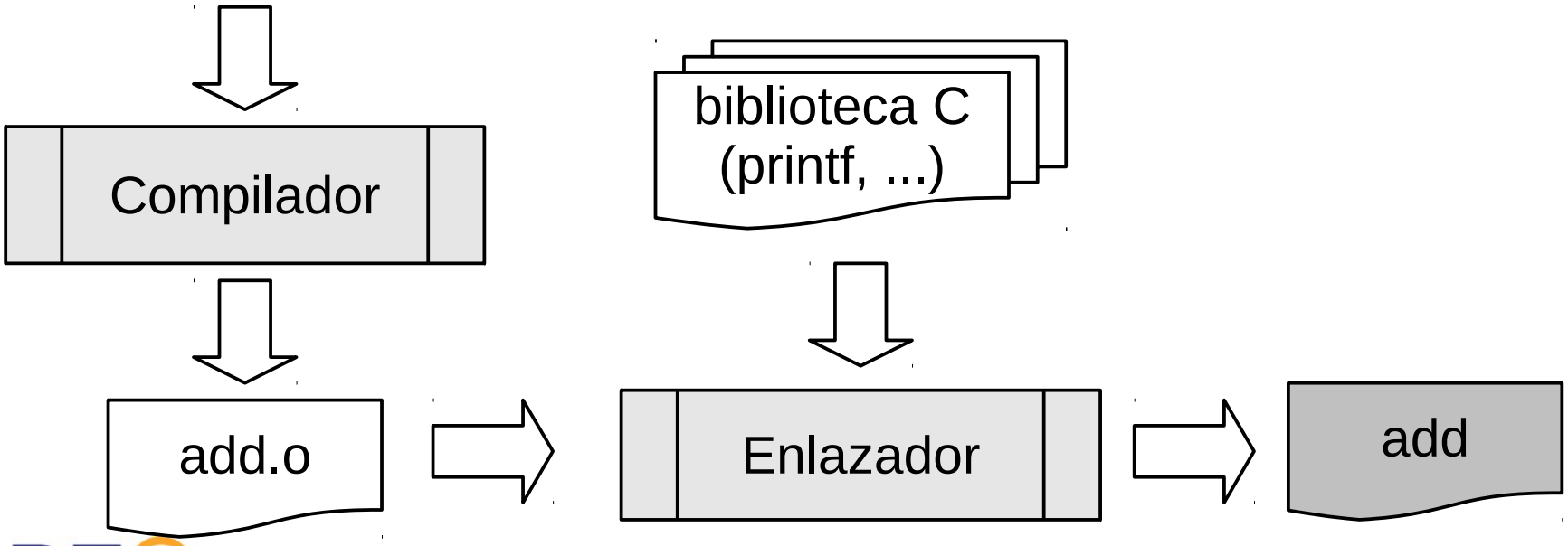
add.c → add.s

```
      .text
      .globl    main
main:
.LFB0:
      pushq     %rbp
      movq %rsp, %rbp
      movq $10, -16(%rbp)
      movq $14, -8(%rbp)
      movq -16(%rbp), %rax
      addq %rax, -8(%rbp)
      movq -8(%rbp), %rax
      popq %rbp
      ret
```

Compiladores. Enlazado

```
// Suma dos número en lenguaje C.

long main()
{
    int a, d;

    a = 10;
    d = 14;
    d = a + d;

    printf("a + d = %d\n", d);

    return 0;
}
```

- Enlazado: combina código compilado para generar un programa ejecutable completo
- Permite uso eficiente de bibliotecas de código
- Tipos:
  - Estático
  - Dinámico



Compiladores
Ejemplo enlazado dinámico

```
$ ldd /bin/ls
    linux-vdso.so.1 (0x00007ffec8da6000)
    libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (...)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (...)
    libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (...)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (...)
    /lib64/ld-linux-x86-64.so.2 (0x00007fe46dfd6000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (...)

$ ldd /bin/cp
    linux-vdso.so.1 (0x00007ffcd68f5000)
    libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (...)
    libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (...)
    libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (...)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (...)
    libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (...)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (...)
    /lib64/ld-linux-x86-64.so.2 (0x00007f9c7e6f8000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (...)
```

Compiladores
Ejemplo enlazado dinámico

```
$ $ ldd /usr/bin/gnome-calculator
        linux-vdso.so.1 (0x00007ffcd8826000)
        libglib-2.0.so.0 => /usr/lib/x86_64-linux-gnu/libglib-2.0.so.0 (0x00007f3b2ce88000)
        libgobject-2.0.so.0 => /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0 (0x00007f3b2cc34000)
        libgtk-3.so.0 => /usr/lib/x86_64-linux-gnu/libgtk-3.so.0 (0x00007f3b2c32c000)
        libgdk-3.so.0 => /usr/lib/x86_64-linux-gnu/libgdk-3.so.0 (0x00007f3b2c036000)
        libatk-1.0.so.0 => /usr/lib/x86_64-linux-gnu/libatk-1.0.so.0 (0x00007f3b2be10000)
        libgio-2.0.so.0 => /usr/lib/x86_64-linux-gnu/libgio-2.0.so.0 (0x00007f3b2ba71000)
        libgtksourceview-3.0.so.1 => /usr/lib/x86_64-linux-gnu/libgtksourceview-3.0.so.1 (0x00007f3b2b7d0000)
        libmpc.so.3 => /usr/lib/x86_64-linux-gnu/libmpc.so.3 (0x00007f3b2b5b8000)
        libmpfr.so.6 => /usr/lib/x86_64-linux-gnu/libmpfr.so.6 (0x00007f3b2b338000)
        libsoup-2.4.so.1 => /usr/lib/x86_64-linux-gnu/libsoup-2.4.so.1 (0x00007f3b2b045000)
        libxml2.so.2 => /usr/lib/x86_64-linux-gnu/libxml2.so.2 (0x00007f3b2ac84000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3b2a893000)
        libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f3b2a621000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f3b2a402000)
        libffi.so.6 => /usr/lib/x86_64-linux-gnu/libffi.so.6 (0x00007f3b2a1fa000)
        libgmodule-2.0.so.0 => /usr/lib/x86_64-linux-gnu/libgmodule-2.0.so.0 (0x00007f3b29ff6000)
        libpangocairo-1.0.so.0 => /usr/lib/x86_64-linux-gnu/libpangocairo-1.0.so.0 (0x00007f3b29de9000)
        libX11.so.6 => /usr/lib/x86_64-linux-gnu/libX11.so.6 (0x00007f3b29ab1000)
        libXi.so.6 => /usr/lib/x86_64-linux-gnu/libXi.so.6 (0x00007f3b298a1000)
        libXfixes.so.3 => /usr/lib/x86_64-linux-gnu/libXfixes.so.3 (0x00007f3b2969b000)
        libcairo-gobject.so.2 => /usr/lib/x86_64-linux-gnu/libcairo-gobject.so.2 (0x00007f3b29492000)
        libcairo.so.2 => /usr/lib/x86_64-linux-gnu/libcairo.so.2 (0x00007f3b29175000)
        libgdk_pixbuf-2.0.so.0 => /usr/lib/x86_64-linux-gnu/libgdk_pixbuf-2.0.so.0 (0x00007f3b28f51000)
        libatk-bridge-2.0.so.0 => /usr/lib/x86_64-linux-gnu/libatk-bridge-2.0.so.0 (0x00007f3b28d20000)
        libepoxy.so.0 => /usr/lib/x86_64-linux-gnu/libepoxy.so.0 (0x00007f3b28a1f000)
        libpangoft2-1.0.so.0 => /usr/lib/x86_64-linux-gnu/libpangoft2-1.0.so.0 (0x00007f3b28809000)
        libpango-1.0.so.0 => /usr/lib/x86_64-linux-gnu/libpango-1.0.so.0 (0x00007f3b285bc000)
        libfontconfig.so.1 => /usr/lib/x86_64-linux-gnu/libfontconfig.so.1 (0x00007f3b28377000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f3b27fd9000)
        libXinerama.so.1 => /usr/lib/x86_64-linux-gnu/libXinerama.so.1 (0x00007f3b27dd6000)
        libXrandr.so.2 => /usr/lib/x86_64-linux-gnu/libXrandr.so.2 (0x00007f3b27bcb000)
        libXcursor.so.1 => /usr/lib/x86_64-linux-gnu/libXcursor.so.1 (0x00007f3b279c1000)
        libXcomposite.so.1 => /usr/lib/x86_64-linux-gnu/libXcomposite.so.1 (0x00007f3b277be000)
        libXdamage.so.1 => /usr/lib/x86_64-linux-gnu/libXdamage.so.1 (0x00007f3b275bb000)
        libxkbcommon.so.0 => /usr/lib/x86_64-linux-gnu/libxkbcommon.so.0 (0x00007f3b2737c000)
        libwayland-cursor.so.0 => /usr/lib/x86_64-linux-gnu/libwayland-cursor.so.0 (0x00007f3b27174000)
        libwayland-egl.so.1 => /usr/lib/x86_64-linux-gnu/libwayland-egl.so.1 (0x00007f3b26f72000)
        libwayland-client.so.0 => /usr/lib/x86_64-linux-gnu/libwayland-client.so.0 (0x00007f3b26d63000)
...
```

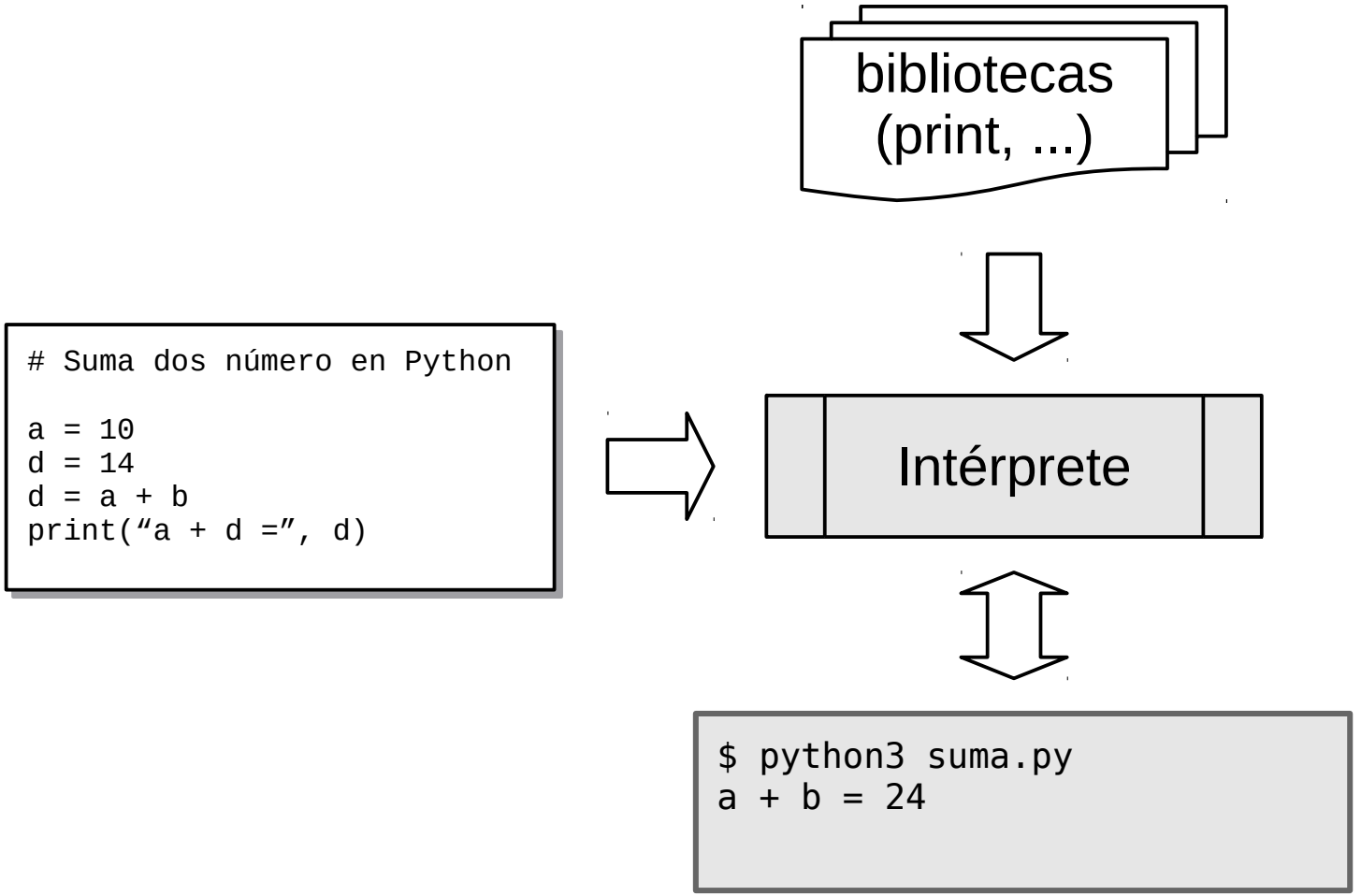
```
$ ldd /usr/bin/gnome-calculator | wc -l
83
```

Contenidos

- Introducción al software
- Lenguaje máquina y ensamblador
- Lenguajes compilados
- **Lenguajes interpretados**
- Jerarquía del software

Lenguajes interpretados

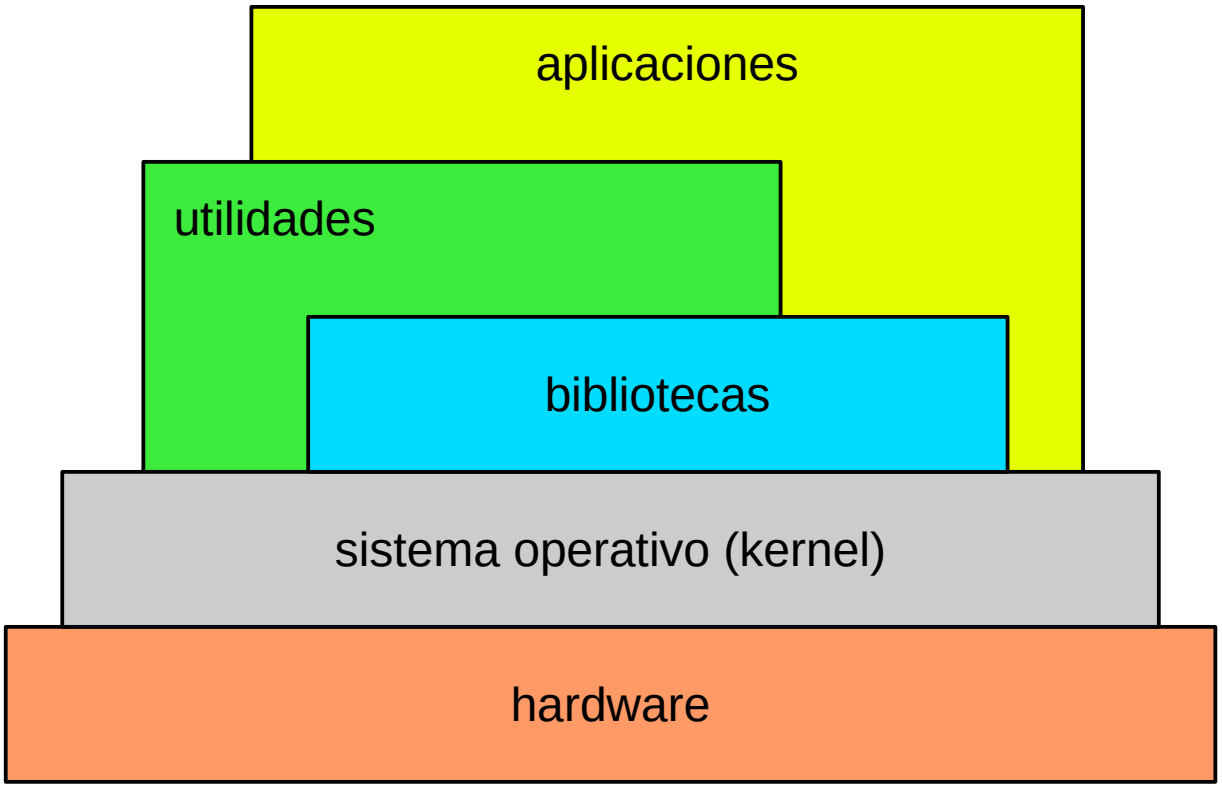
- Un programa lee, “interpreta” y ejecuta directamente el código fuente.



Lenguajes interpretados

- Inconvenientes:
  - ☹ Rendimiento muy inferior a los programas compilados.
  - ☹ Los programas ocupan más espacio.
- Ventajas:
  - ☺ No necesita compilar.
  - ☺ Desarrollo más rápido de aplicaciones.
  - ☺ Más fácil portar a distintos tipos de ordenadores.
  - ☺ Mayor riqueza en tipos de datos y elementos del lenguaje.
  - ☺ Mayor independencia del hardware.
  - ☺ Aprendizaje más fácil.
  - ☺ Programación más fácil.
- Ejemplos:
  - Python, Java, R, javascript, PHP, Ruby, ...

Jerarquía del software



Resumen

- Los computadores ejecutan programas formados por instrucciones sencillas en binario: **código máquina**.
- La forma más básica de programar los ordenadores es mediante una representación del código máquina denominada **lenguaje ensamblador**.
- Cada tipo de procesador tiene su propio código máquina y lenguaje ensamblador (**dependientes del hardware**)
- Los **lenguajes de programación de alto nivel** buscan la independencia del hardware y la facilidad de la programación.
- Los **lenguajes compilados** son transformados en código máquina por los **compiladores**.
- Los **lenguajes interpretados** son “ejecutados” por otros programas llamados **intérpretes**.
- Los programas en lenguajes interpretados se ejecutan de forma mucho menos eficiente que los de lenguajes compilados.
- La mayoría de programas se construyen combinando funciones de bibliotecas de programación.