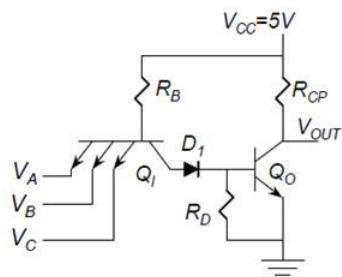
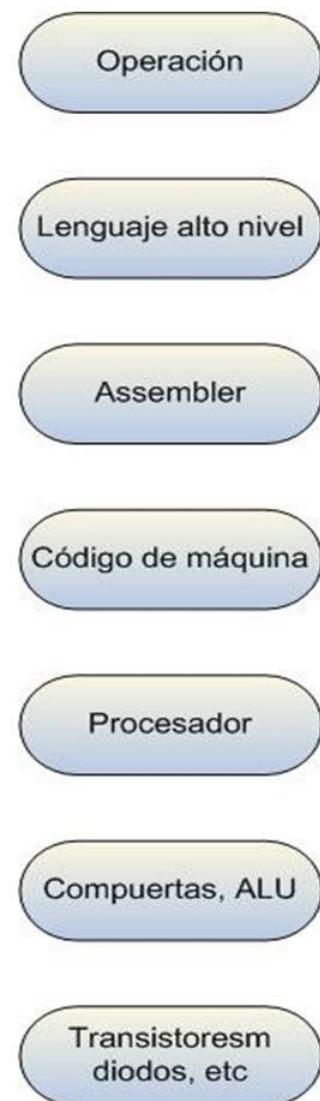


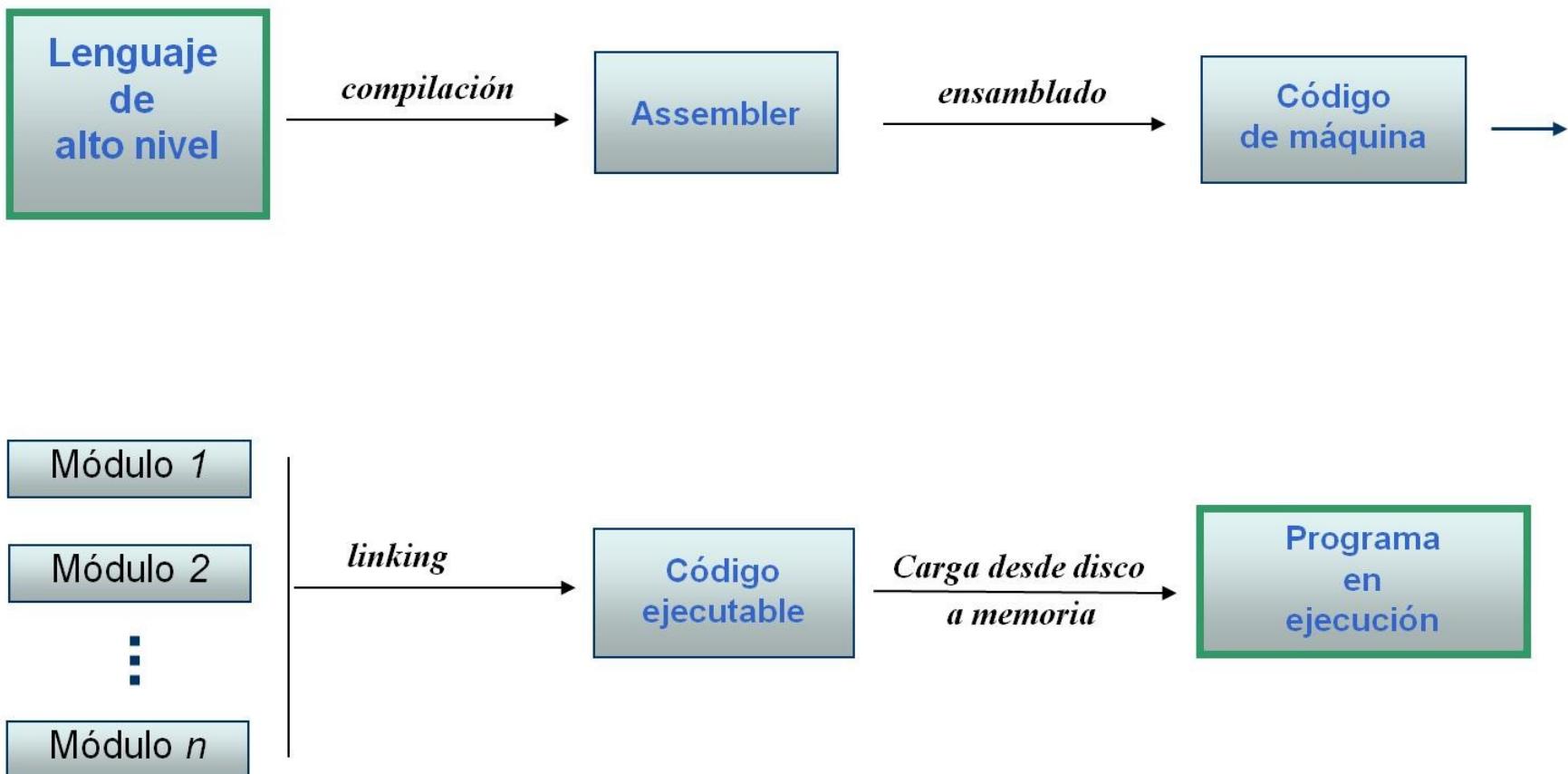
66.70 Estructura del Computador

**El lenguaje  
y la máquina**



## Niveles de abstracción en una computadora

# Desde el diseño a la ejecución



# Compiladores

- Compilador de **una sola pasada o de pasadas múltiples**

Completa el proceso en un solo recorrido del programa fuente o en varios

- **Compilador incremental**

Genera código instrucción por instrucción (no para todo el programa) cuando usuario pulsa una tecla. Entorno de depuración. Intérpretes

- **Cross-compilador**

Genera código en lenguaje assembler para una máquina diferente de la que se está utilizando para compilar.

# El proceso de compilación

Datos de  
entrada

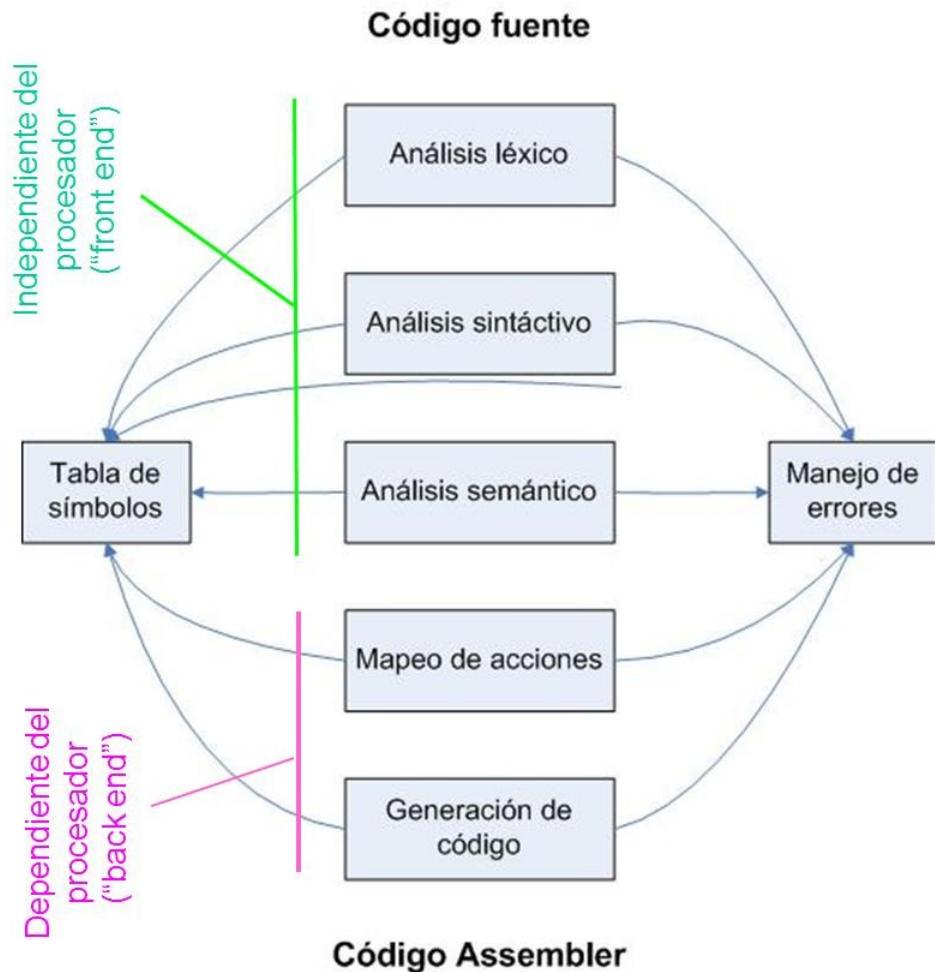
Código fuente



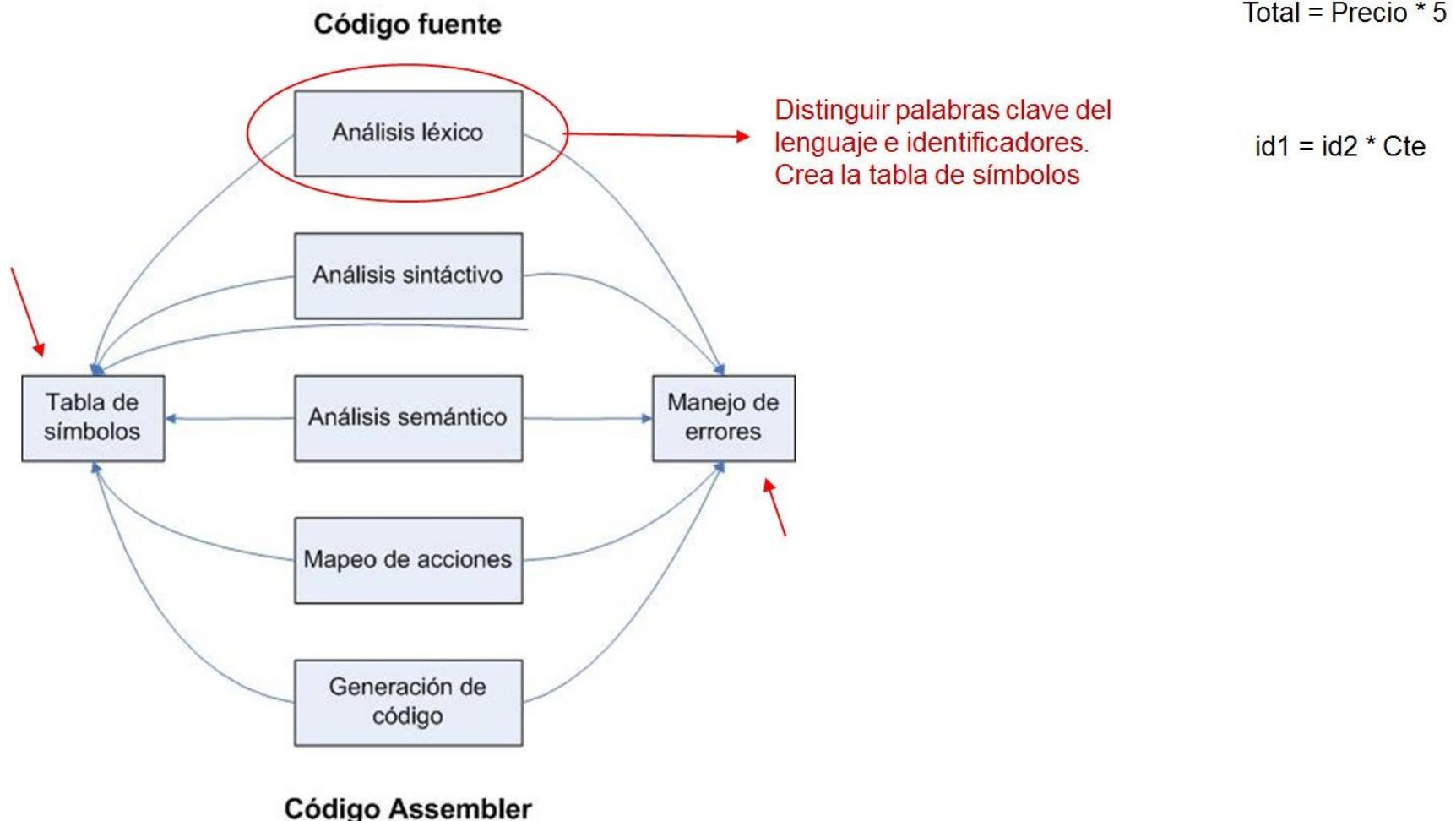
Salida del  
proceso

Código Assembler

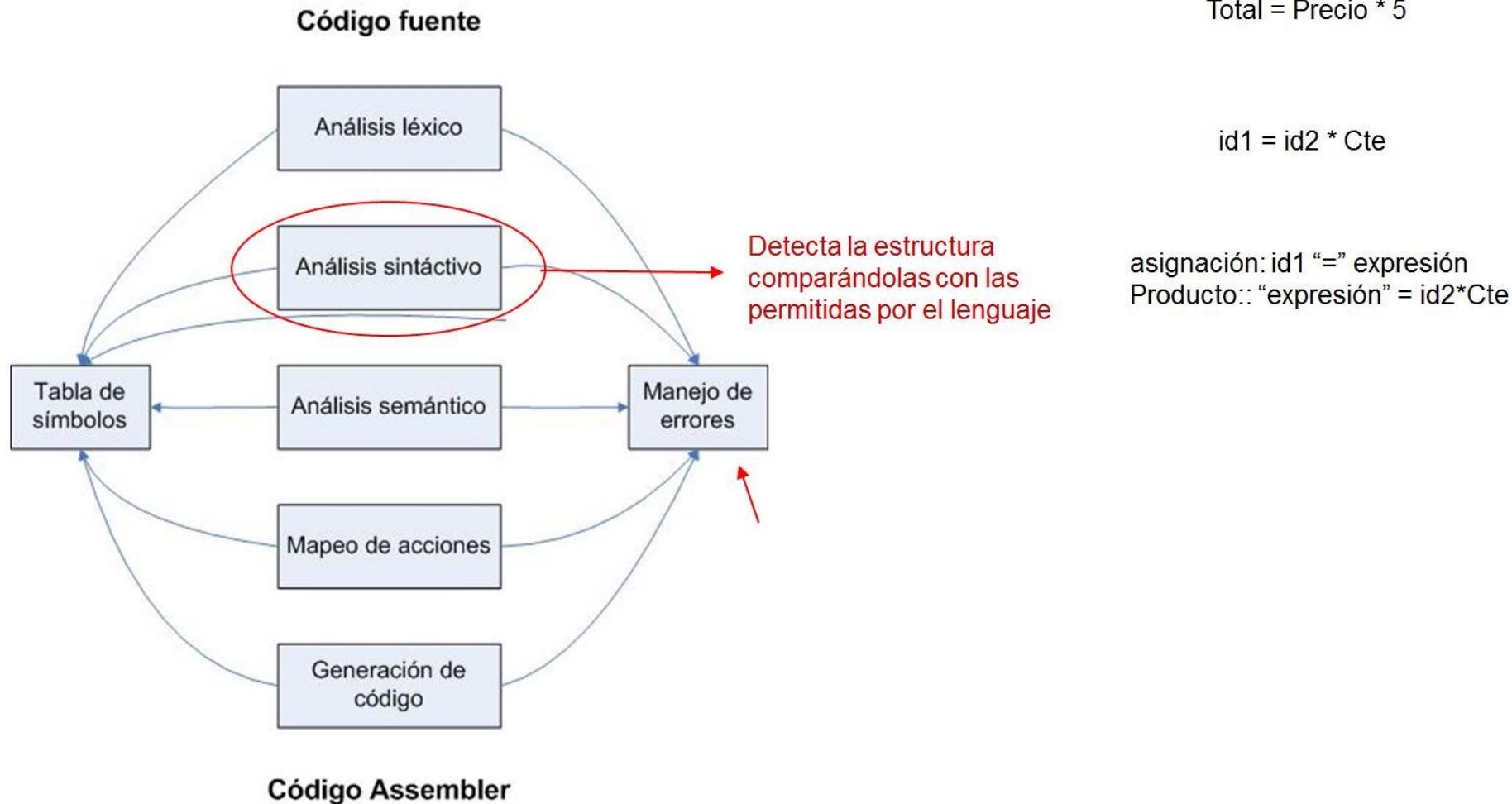
# El proceso de compilación



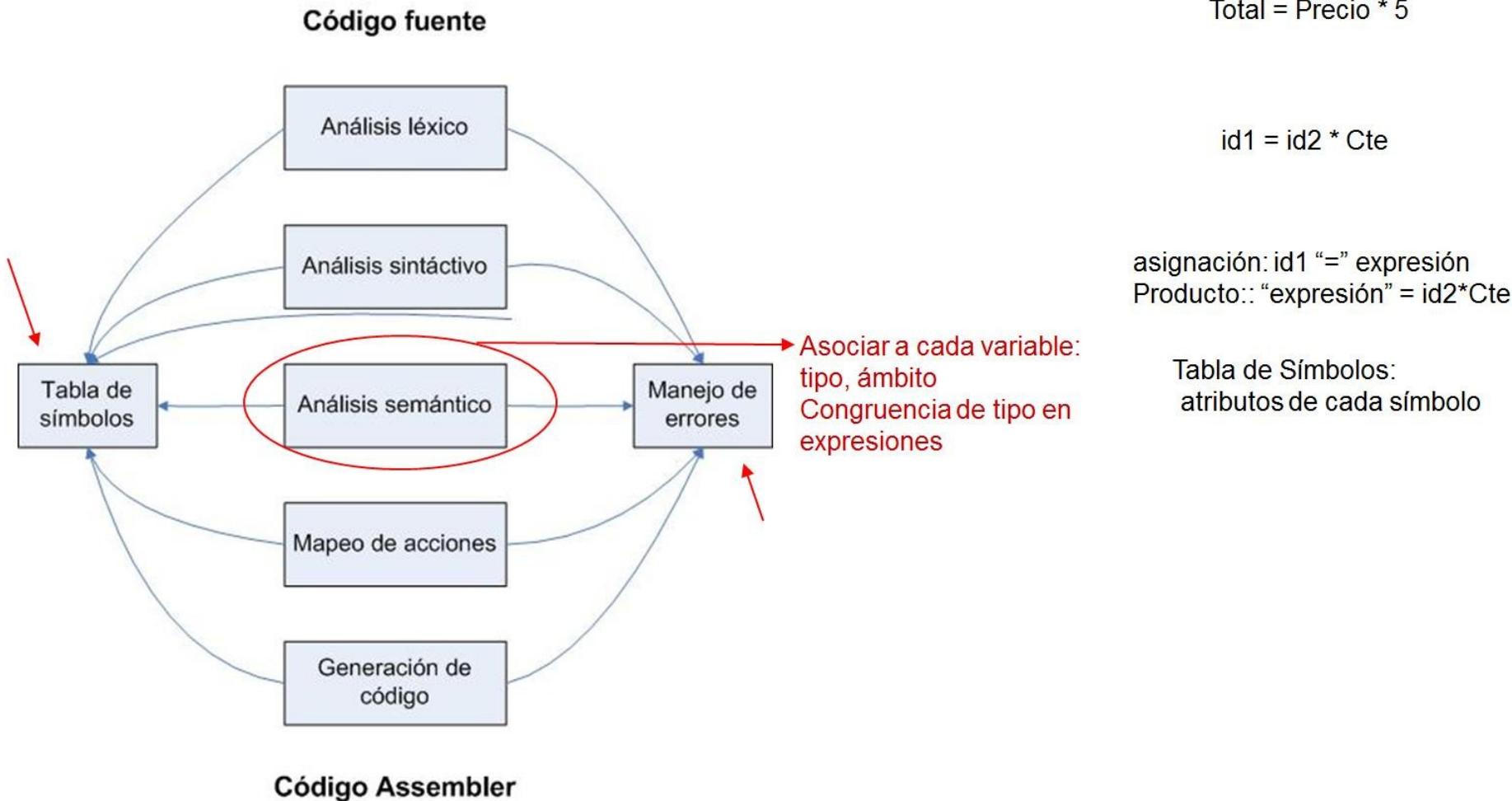
# El proceso de compilación



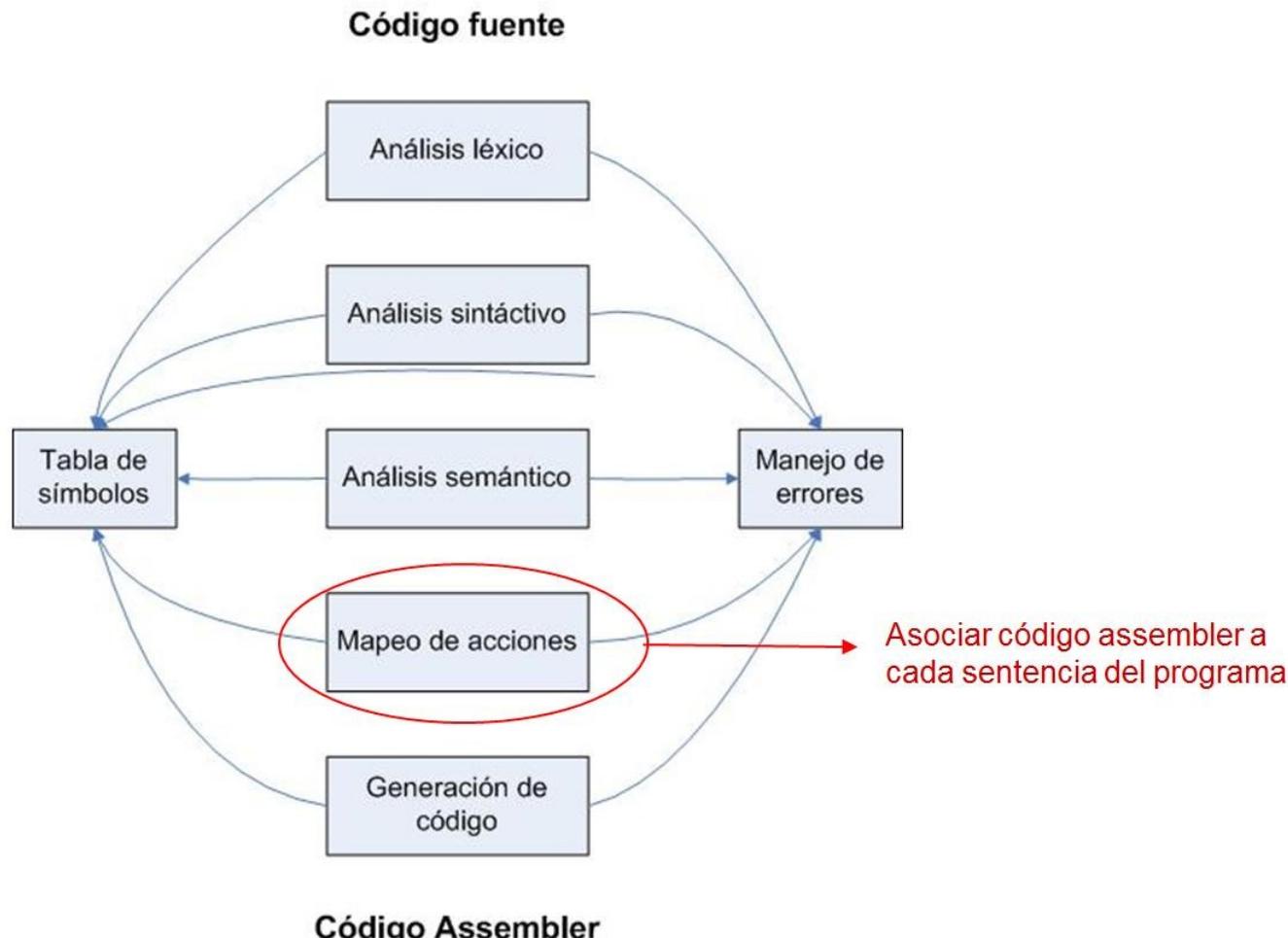
# El proceso de compilación



# El proceso de compilación



# El proceso de compilación



Total = Precio \* 5

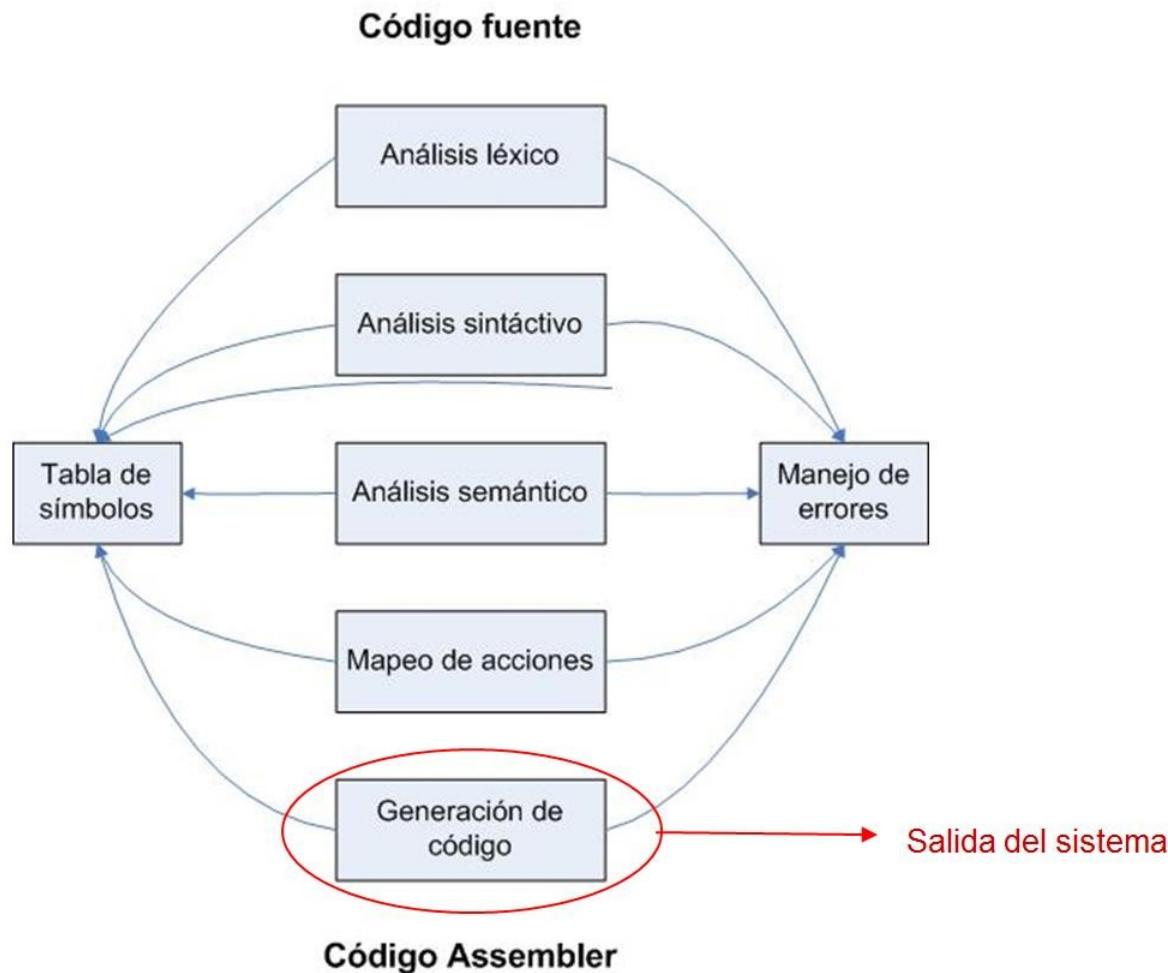
id1 = id2 \* Cte

asignación: id1 “=” expresión  
Producto:: “expresión” = id2\*Cte

Tabla de Símbolos:  
atributos de cada símbolo

- Código Assembler
- Localiz de las variables.
- Uso de registros

# El proceso de compilación



Total = Precio \* 5

id1 = id2 \* Cte

asignación: id1 "=" expresión  
Producto:: "expresión" = id2\*Cte

Tabla de Símbolos:  
atributos de cada símbolo

- Código assembler
- Variables en registros
- Uso de registros

# Mapeo de Acciones



Tres tipos de instrucciones:

- **Operaciones aritméticas/lógicas**
- **Control de flujo del programa**
- **Movimiento de datos**

# Mapeo de Acciones

## Operaciones aritméticas/lógicas

" $A=B+4$ "

! Sentencia de asignación

ld [B], %r1	! guardar variable B en un registro
add %r1, 4, %r2	! calcular el valor de la expresión
st %r2, [A]	! hacer la asignación del valor

- ❖ Modos de direccionamiento a los operadores dependen del ISA
- ❖ En máquinas RISC: operandos siempre en registros
  - Cantidad de registros
  - Cuando es excedida => registros al stack
  - El compilador debe decidir cuáles registros contienen valores que ya no son necesarios.

Requiere de técnicas sofisticadas  
¿Porqué se justifica utilizarlas?

# Mapeo de Acciones

## Estructuras de control de flujo del programa

<b>Goto</b> statement	ba label												
<b>if</b> A=B stmt1 <b>else</b> stmt2;	<pre> subcc %r1, %r2, %r0 ! setea flags, descarta resultado bne    else1       !... código de stmt1 ba     fin else1:       !código de stmt2 fin:       ! ... </pre>												
<b>while</b> (r1 == r2)    %r3 = %r3 + 1;	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 30%;"></td> <td style="width: 30%; text-align: center;">ba</td> <td style="width: 40%; text-align: center;">Test</td> </tr> <tr> <td>True:</td> <td>add    %r2, 1, %r3</td> <td></td> </tr> <tr> <td>Test:</td> <td>subcc %r1, %r2, %r0</td> <td></td> </tr> <tr> <td></td> <td>be     True</td> <td></td> </tr> </table>		ba	Test	True:	add    %r2, 1, %r3		Test:	subcc %r1, %r2, %r0			be     True	
	ba	Test											
True:	add    %r2, 1, %r3												
Test:	subcc %r1, %r2, %r0												
	be     True												
<b>Do</b> r3 =r3 + 1 <b>while</b> (r1 == r2)	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 30%;"></td> <td style="width: 30%; text-align: center;">add    %r2, 1, %r3</td> <td style="width: 40%; text-align: center;">True</td> </tr> <tr> <td>True:</td> <td>subcc %r1, %r2, %r0</td> <td></td> </tr> <tr> <td>Test:</td> <td>be     True</td> <td></td> </tr> </table>		add    %r2, 1, %r3	True	True:	subcc %r1, %r2, %r0		Test:	be     True				
	add    %r2, 1, %r3	True											
True:	subcc %r1, %r2, %r0												
Test:	be     True												

# Mapeo de Acciones

## Dónde almacenar variables en RAM

- ❖ **Variables estáticas (globales)**

Permanecen a lo largo del tiempo de ejecución de la aplicación

=> se guardan en posición de memoria conocida en tiempo de compilación

- ❖ **Variables locales a un procedimiento**

Desaparecen cuando termina el procedimiento en el que están declaradas

=> se guardan en el stack

- ❖ **Guardar variables en registros es óptimo!**

# Ensamblador



- ❖ Transcodificación: Relación 1 a 1 entre código Assembler y código de máquina  
(Comparar con compilación)
- ❖ Ofrece al programador
  - Representación simbólica para direcciones y constantes
  - Definir la ubicación de las variables en memoria
  - Variables inicializadas antes de ejecución
  - Provee cierto grado de aritmética en tiempo de ensamblado
  - Utilizar variables declaradas en otros módulos
  - Utilizar macros (dar nombre a fragmentos de texto)

# Directivas al ensamblador (ARCtools)

- Indican al ensamblador como procesar una sección de programa
- -Las instrucciones son específicas de un procesador
  - Las pseudo-instrucciones o directivas son específicas de un programa ensamblador
- Algunas generan información en la memoria, otras no

Pseudo-Op	Usage	Meaning
.equ	x .equ #10	Treat symbol x as $(10)_{16}$
.begin	.begin	Start assembling
.end	.end	Stop assembling
.org	.org 2048	Change location counter to 2048
.dwb	.dwb 25	Reserve a block of 25 words
.global	.global y	y is used in another module
.extern	.extern z	z is defined in another module
.macro	.macro M a, b, ...	Define macro M with formal parameters a, b, ...
.endmacro	.endmacro	End of macro definition

# Ensamblador

## Obtención del código de máquina

```
! This program adds two numbers
.begin
.org 2048
main: ld      [x], %r1          ! Load x into %r1
       ld      [y], %r2          ! Load y into %r2
       addcc %r1, %r2, %r3      ! %r3 ← %r1 + %r2
       st      %r3, [z]          ! Store %r3 into z
       jmp1  %r15 + 4, %r0      ! Return
x:     15
y:     9
z:     0
.end
```

ld [x], %r1  
ld [y], %r2



11 00001 000000 00000 1 0100000010100  
op rd op3 rs1 i simm13

11 00010 000000 00000 1 0100000011000  
op rd op3 rs1 i simm13

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00					
1	1																																			
		rd																																		

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00								
1	1																																						
		rd																																					

Formato de acceso a memoria:  $op=11$

$op3$	direcc. de memoria (ld o st)
000000 ld rd=reg.destino	= $rs1 + rs2$
000100 st rd=reg.origen	= $rs1 + simm13$ (constante)      si $i = 1$

# Proceso de ensamblado en dos pasadas

- ❖ Preproceso: expansión de macros => (1) Registrar definiciones (2) Reemplazar
- ❖ Primer pasada
  - Detecta identificadores y les asigna una posición de memoria
  - Crea la tabla de símbolos
- ❖ Segunda pasada
  - Cada instrucción es convertida a código de máquina
  - Cada identificador es reemplazado por su ubicación en memoria según indica la tabla de símbolos
  - Cada línea es procesada completamente antes de avanzar a la siguiente
  - Genera el código objeto y el listado

# Creación de la tabla de símbolos

(Se completa en la primera pasada)

```
! This program sums LENGTH numbers
! Register usage:      %r1 - Length of array a
!                      %r2 - Starting address of array a
!
!                      %r3 - The partial sum
!
!                      %r4 - Pointer into array a
!                      %r5 - Holds an element of a
.
.a_start           .begin          ! Start assembling
.org 2048          .org 2048        ! Start program at 2048
.equ 3000          .equ 3000        ! Address of array a
ld   [length], %r1 ! %r1 ← length of array a
ld   [address],%r2 ! %r2 ← address of a
andcc %r3, %r0, %r3 ! %r3 ← 0
loop:             andcc %r1, %r1, %r0 ! Test # remaining elements
be   done           ! Finished when length=0
addcc %r1, -4, %r1 ! Decrement array length
addcc %r1, %r2, %r4 ! Address of next element
ld   %r4, %r5       ! %r5 ← Memory[%r4]
addcc %r3, %r5, %r3 ! Sum new element into r3
ba   loop           ! Repeat loop.

done:             jmpl %r15 + 4, %r0 ! Return to calling routine
length:           20              ! 5 numbers (20 bytes) in a
address:          a_start         ! Start of array a
.a:               .org a_start     ! Start of array a
                  25              ! length/4 values follow
                  -10
                  33
                  -5
                  7
.end              ! Stop assembling
```

- Recorre el texto linea por linea.
- Busca identificadores e incluye su nombre en la tabla
- Le asigna un valor al momento de encontrarlo y si no es así deja su valor en blanco y lo completa cuando encuentra en líneas siguientes información para ello

Símbolo	Valor
a_start	3000
length	2092
address	2096
loop	2060
done	2088

- ❖ Contador de posición (análogo a un *program counter* en tiempo de ensamblado)
- Inicializado a cero al inicio de la primer pasada y por cada directiva `.org`
  - Incrementado por cada instrucción según su tamaño (en ARC 4 bytes)

# Archivos creados por el ensamblador

(Se completa en la segunda pasada)

## Salidas

- 1) Archivo de texto (listado)
- 2) Archivo con código objeto

Forma parte del archivo de listado

Símbolo	Valor
a_start	3000
length	2092
address	2096
loop	2060
done	2088

Tabla de símbolos

Location counter	Instruction	Object code
	.begin	
	.org 2048	
	a_start .equ 3000	
2048	ld [length],%r1	11000010 00000000 00101000 00101100
2052	ld [address],%r2	11000100 00000000 00101000 00110000
2056	andcc %r3,%r0,%r3	10000110 10001000 11000000 00000000
2060	loop: andcc %r1,%r1,%r0	10000000 10001000 01000000 00000001
2064	be done	00000010 10000000 00000000 00000110
2068	addcc %r1,-4,%r1	10000010 10000000 01111111 11111100
2072	addcc %r1,%r2,%r4	10001000 10000000 01000000 00000010
2076	ld %r4,%r5	11001010 00000001 00000000 00000000
2080	ba loop	00010000 10111111 11111111 11111011
2084	addcc %r3,%r5,%r3	10000110 10000000 11000000 00000101
2088	done: jmpl %r15+4,%r0	10000001 11000011 11100000 00000100
2092	length: 20	00000000 00000000 00000000 00010100
2096	address: a_start	00000000 00000000 00001011 10111000
	.org a_start	
3000	a: 25	00000000 00000000 00000000 00011001
3004		11111111 11111111 11111111 11110110
3008	-10	
3012	33	00000000 00000000 00000000 00100001
3016	-5	11111111 11111111 11111111 11111011
	7	00000000 00000000 00000000 00000111
	.end	

Listado

# Archivos creados por el ensamblador

(Se completa en la segunda pasada)

## Salidas

- 1) Archivo de texto (listado )
- 2) Archivo con código objeto



- Código de máquina

Incluye también un encabezamiento de archivo conteniendo:

- Dirección de primera instrucción a ejecutar (si corresponde): *main()*
- *Librerías externas* que son utilizadas por el módul
- *Tabla de símbolos con información sobre:*
  - .Símbolos que están declarados en otros módulo: *externos*
  - .Símbolos que se declaran localmente pero se le da permisos de acceso desde *otros módulos*
  - .Información sobre la *relocalización* del código

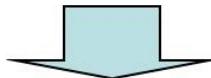
Symbol	Value	Global/ External	Reloc- atable
sub	-	External	-
main	2048	No	Yes
x	2064	No	Yes
y	2068	No	Yes

# Tipo de ensamblador

- ❖ ¿Podría hacerse un ensamblador de sólo una pasada ?
- ❖ ¿Sería este funcionamiento notado por el programador?

# Localización del programa en memoria

- En general no se sabe donde va a ser cargado el programa  
 $\Rightarrow \text{“.org 2048”}$
- Si varios módulos son vinculados no se sabe donde va a ser cargado el programa ni cada módulo  
 $\Rightarrow \text{“.org 2048”}$



## Código relocalizable

- El ensamblador es responsable de marcar direcciones relocalizables y direcciones absolutas
- Esta información es necesaria por el linker
- Tabla de símbolos

# EL LINKER

- ✓ Combina dos o más módulos que fueron ensamblados separadamente

Origen de estos módulos:

- Aplicación
- librería del ambiente de desarrollo
- sistema operativo

- ✓ El linker:

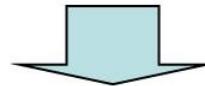
- Resuelve referencias de memoria externa al módulo
- Relocaliza los módulos combinándolos y reasignando las direcciones internas a cada uno para reflejar su nueva localización
- Define en el módulo a cargar la dirección de la primer instrucción a ser ejecutada (“main”)

# Referencias externas

En un módulo dado hay

Símbolos locales

Símbolos globales



Directivas al ensamblador

`.global` → declara un símbolo global

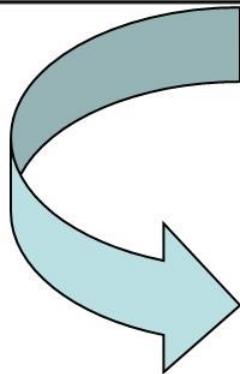
`.extern` → utiliza símbolo declarado en otro módulo

Declarar una cláusula ".equ" como global no tiene sentido. ¿Porqué?

# Referencias externas

```
! Main program
.begin
.org 2048
.extern sub
main: ld [x], %r2
      ld [y], %r3
      call sub
      jmp1 %r15 + 4, %r0
x: 105
y: 92
.end
```

```
! Subroutine library
.begin
ONE .equ 1
.org 2048
.global sub
sub: orncc %r3, %r0, %r3
      addcc %r3, ONE, %r3
      jmp1 %r15 + 4, %r0
.end
```



Symbol	Value	Global/External
sub	-	External
main	2048	No
x	2064	No
y	2068	No

Main Program

Symbol	Value	Global/External
ONE	1	No
sub	2048	Global

Subroutine Library

INCOMPLETAS  
(sigue en próx transparencia)

# Símbolos relocalizables

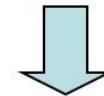
TIEMPO DE ENSAMBLADO

```
! Main program
.begin
.org 2048
.extern sub
main: ld    [x], %r2
      ld    [y], %r3
      call  sub
      jmpl %r15 + 4, %r0
      x: 105
      y: 92
      .end
```

```
! Subroutine library
.begin
ONE .equ 1
.org 2048
.global sub
sub: orncc %r3, %r0, %r3
     addcc %r3, ONE, %r3
     jmpl %r15 + 4, %r0
     .end
```

TIEMPO DE LINKING

~~Addr(Main) = Addr(Sub) = 2048~~



Direcciones relocalizables

- **Ensamblador**

Marca direcciones como relocalizables o no relocalizables (absolutas)

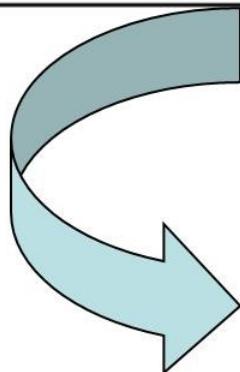
- **Linker**

Redefine las direcciones relocalizables a partir de la nueva dirección de origen

# Símbolos relocalizables

```
! Main program
.begin
.org 2048
.extern sub
main: ld [x], %r2
      ld [y], %r3
      call sub
      jmp1 %r15 + 4, %r0
x: 105
y: 92
.end
```

```
! Subroutine library
.begin
ONE .equ 1
.org 2048
.global sub
sub: orncc %r3, %r0, %r3
      addcc %r3, ONE, %r3
      jmp1 %r15 + 4, %r0
.end
```



Symbol	Value	Global/ External	Reloc- atable
sub	-	External	-
main	2048	No	Yes
x	2064	No	Yes
y	2068	No	Yes

Main Program

Symbol	Value	Global/ External	Reloc- atable
ONE	1	No	No
sub	2048	Global	Yes

Subroutine Library

# Símbolos relocalizables

## EL ENSAMBLADOR

- ✓ Determina cuáles símbolos son relocalizables y cuáles no
- ✓ Los marca en el módulo ensamblado
- ✓ Identifica código que debe ser modificado como resultado de la relocalización

> No tiene sentido marcar como relocalizables símbolos declarados en otros módulos

### No son relocalizables:

- ✓ *Direcciones de entrada/salida*
- ✓ *Rutinas del sistema*

### Sí son relocalizables:

- ✓ Posiciones de memoria relativas a un .org (p.e.: x o y)

# Carga del programa en memoria

*El loader toma el programa de disco y lo carga en memoria principal  
¿en qué dirección de memoria?*

- En ambientes multitarea la ram es compartida entre varios procesos
- El assembler no puede saber donde puede ser cargado
- El linker no puede saber donde puede ser cargado



- ✓ El loader debe relocalizar todos los símbolos relocalizables

Algunos loader también tienen capacidad de

- ✓ combinar módulos en tiempo de carga

## ■ Link-editor

- Produce una versión linkeada del programa que normalmente es guardada en archivo para ser ejecutada en otro momento

## ■ Relocating loader

- Linkea en tiempo de carga.
- Relocaliza y carga el programa en memoria para su ejecución

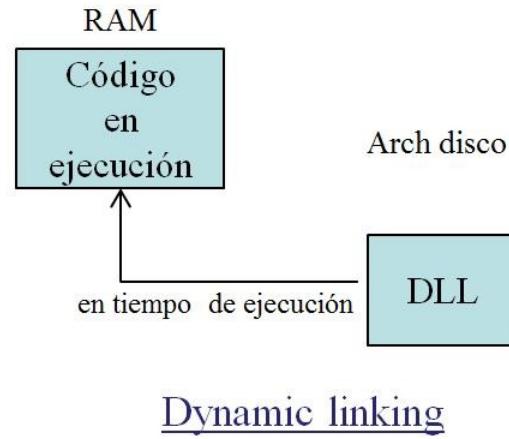
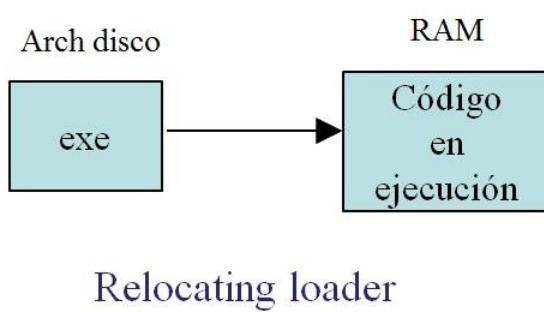
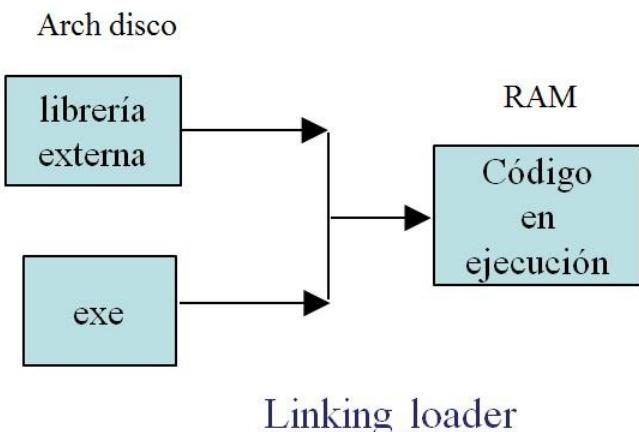
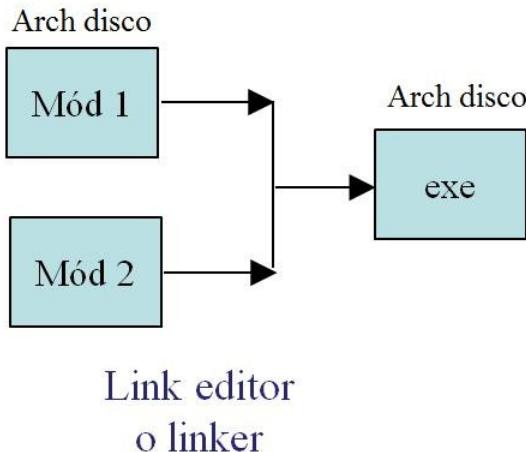
## ■ Linking loader

- Linkea en tiempo de carga.
- Relocaliza, busca automáticamente librerías, linkea y carga el programa en memoria para su ejecución

## ■ Linking Loader dinámico

- Linkea en tiempo de ejecución
- Carga rutinas sólo cuando el programa las necesita
- Utiliza **Dynamic Link Libraries (dll)**

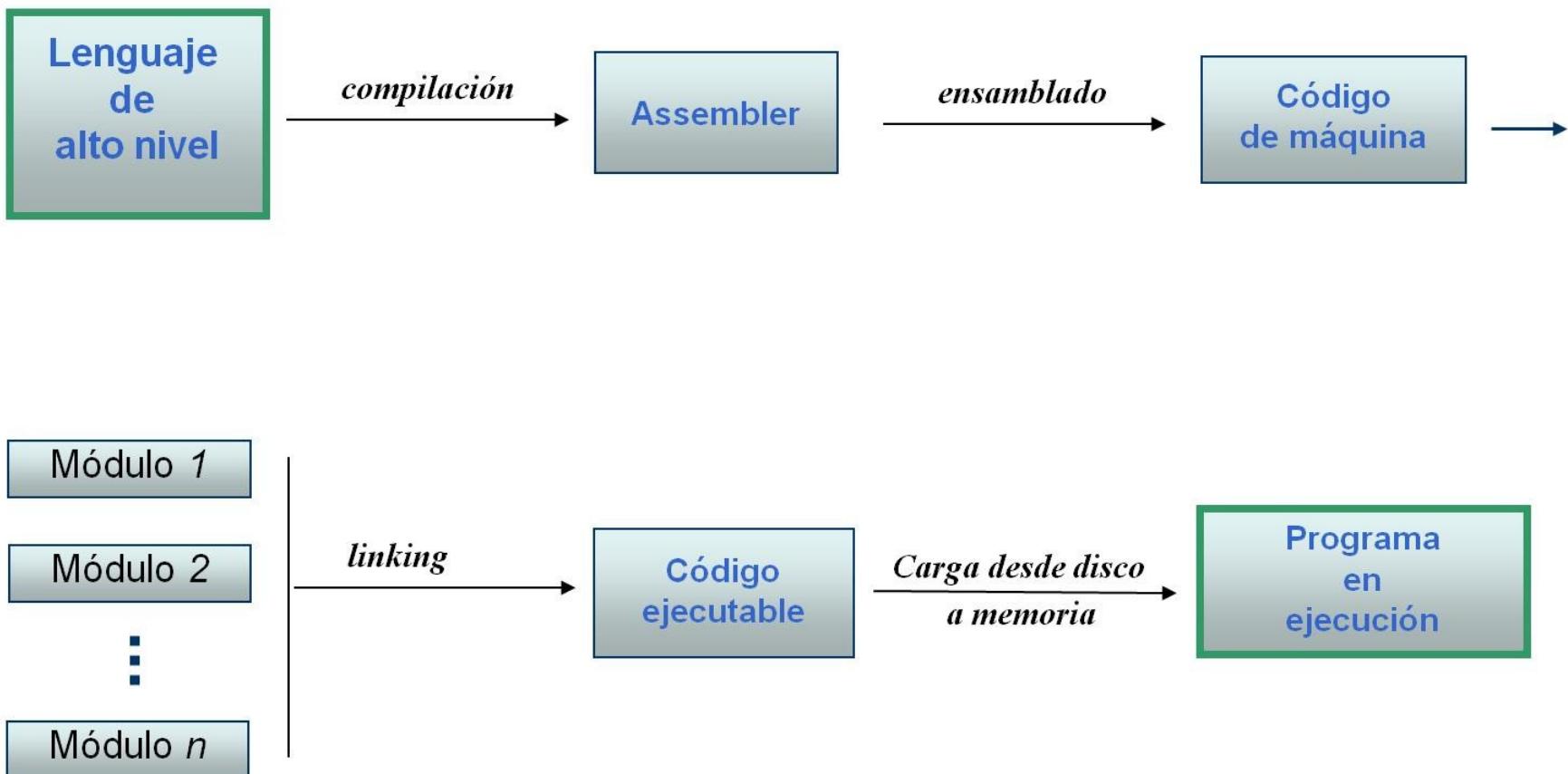
# Linking & loading



# Archivos objeto

- ✓ Archivo objeto *relocateable* : código binario y datos en un formato que permite combinarlo con otros archivos objeto relocateables (linking)
- ✓ Archivo objeto *ejecutable*: código binario y datos en un formato que permite cargarlo directamente a memoria y ejecutarlo
- ✓ Archivo objeto *compartido*: tipo especial de archivo objeto relocateable que puede ser cargado en memoria y vinculado dinámicamente
- El formato de los archivos objeto es dependiente del sistema donde van a ejecutarse

# Desde el diseño a la ejecución



# Eficiencia del código

## Lenguajes de alto nivel

### ✓ Constantes:

- Donde se guardan?
- Influye su valor?
- Aritmética de constantes en tiempo de compilación
- Variables inicializadas vs. Constantes

```
#define Mil 1000;  
int x = 0;  
int main() {  
    x= x + 1000;  
    x= x + Mil;  
    return 0;  
}
```

```
#define Mil 1000;  
int x = 0;  
int main() {  
    x= x + 100000;  
    x= x + Mil;  
    return 0;  
}
```

```
#define Mil 1000 ;  
#define DosMil (Mil*2);  
int x = 0;  
int main() {  
    x= x + 1000;  
    x= x + Mil;  
    x= x + DosMil;  
    return 0;  
}
```

¿Y la declaración **const** de C++?  
p.e.: const int Constant1=96;

# Eficiencia del código

## Lenguajes de alto nivel

### ✓ Variables:

- La elección de tipo no es gratuita
- Tipos enteros

long x = 10; long y = 15; long z; int main() { z= x + y; return 0; }	signed char x = 10; signed char y = 15; signed char z; int main() { z= x + y; return 0; }	signed char x = 10; long y = 15; long z; int main() { z= x + y; return 0; }	long long x = 10; long long y = 15; long long z; int main() { z= x + y; return 0; }
ld [x], %or1 ld [y], %or2 add %or1, %or2, %or3 st %or3, [z]	lds [x], %or1 lds [y], %or2 add %or1, %or2, %or3 stb %or3, [z]	? ?	? ?

¿Y si es de 64 bits en un procesador de 32 bits?

# Eficiencia del código

## Lenguajes de alto nivel

### ✓ Variables:

- La elección de tipo no es gratuita
- Tipos enteros
- Tipos punto flotante

long x = 10; long y = 15; long z; int main() { z= x + y; return 0; }	float x = 10; float y = 15; float z; int main() { z= x + y; return 0; }	float x = 10; long y = 15; float z; int main() { z= x + y; return 0; }	double x = 10; float y = 15; double z; int main() { z= x + y; return 0; }
ld [x], %or1 ld [y], %or2 add %or1, %or2, %or3 st %or3, [z]	ldf [x], %f1 ldf [y], %f2 fadds %f1, %f2, %f3 stf %f3, [z]	i?	i?

# Eficiencia del código

## Lenguajes de alto nivel

- ✓ **Arreglos unidimensionales y multidimensionales**
  - Eficiencia en el cálculo del offset dentro del espacio de memoria del array

- ✓ **Arreglos inicializados**
  - Arreglo globales vs Arreglos locales

```
static long arrayX[4] = {1,2,3,4};  
int main() {  
    :  
    :  
}
```

```
int functionX()  
{  
    long arrayX[4] = {1,2,3,4};  
    :  
}  
} // end functionX()
```

ASM

```
arrayX:    1  
           2  
           3  
           4  
           ,org 2048  
main:
```

?

# Eficiencia del código

## Lenguajes de alto nivel

### ✓ Invocar funciones y procedimientos

- Llamar una función no es gratuito

```
int main( ) {  
    float m, n ;  
    printf ( "\nIngrese un numero: \n");scanf ( "%f",&m ) ;  
    n = m * m ;  
    printf ( "\nSu cuadrado es %f is %f",m,n );  
}
```

```
float square ( float x );  
int main( ) {  
    float m, n ;  
    printf ( "\nIngrese un numero: \n"); scanf ( "%f", &m );  
    n = square ( m ) ;  
    printf ( "\nSu cuadrado es %f is %f",m,n );}  
  
float square ( float x )  
{  float p ;  
   p = x * x ;  
   return ( p ) ; }
```

- ✓ Tiempo del código de la función



- ✓ Backup de la dirección de regreso actual
- ✓ Setear la dirección actual como nuevo punto de regreso
- ✓ Al volver recuperar la dirección de regreso anterior
- ✓ Pasar parámetros (particularmente si no es mediante registros)
- ✓ Devolver parámetros
- ✓ Backup de los registros que utiliza el proced, y su recuperación

# Eficiencia del código

## Lenguajes de alto nivel

### ✓ Invocar funciones y procedimientos

**«El orden con que se declaran parámetros y variables locales no es indiferente»**

- Compiladores generalmente reservan registros para ese uso
- Los registros son escasos
- Prioridad para los que aparecen primero
- Identificar parámetros y variables con mayor acceso durante la ejecución

