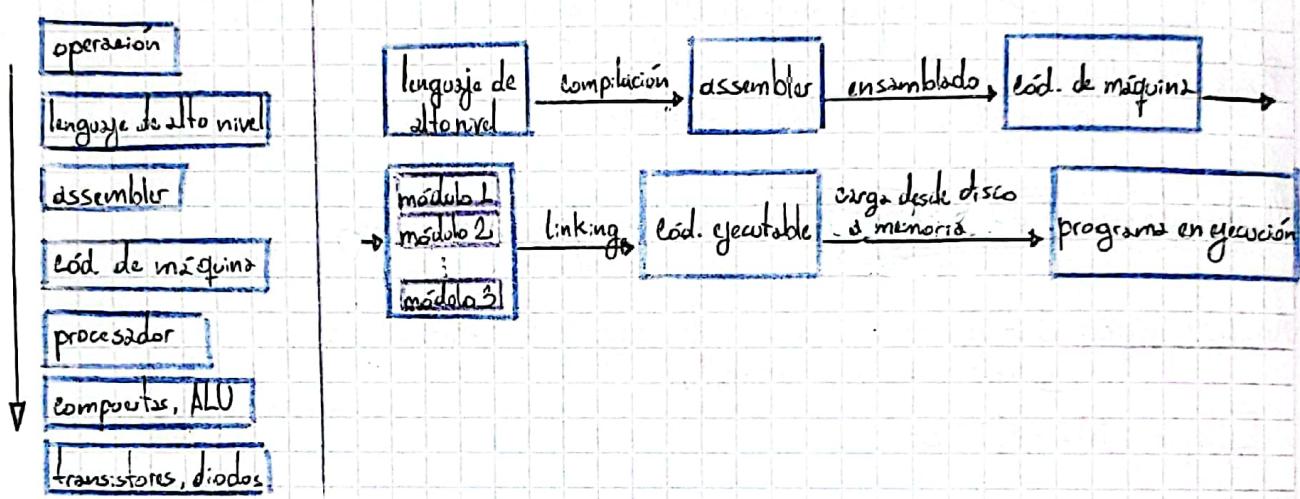


COMPILADORES Y ENSAMBLADORES

Desde el diseño hasta la ejecución



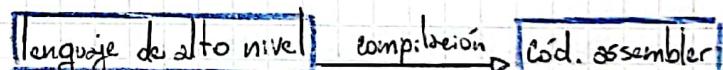
Compiladores

→ programa que traduce los programas escritos en lenguajes de alto nivel a cód. de máquina.

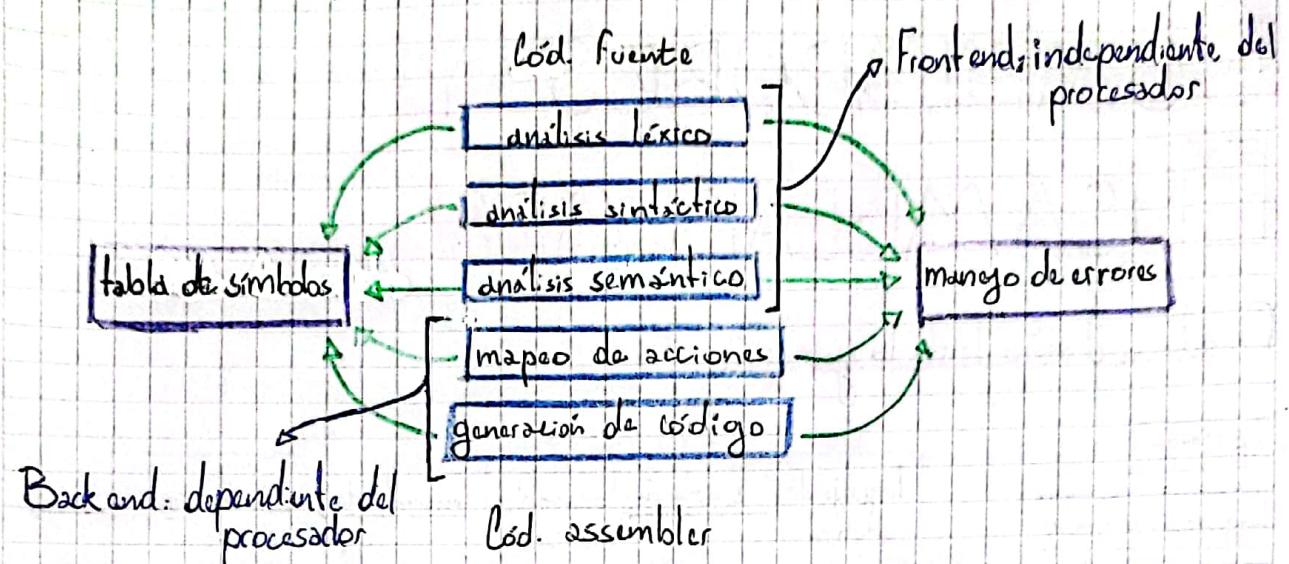
• Tipos

• De pasadas

- De una pasada: escanea el cód. fuente una sola vez.
- De varias pasadas: escanea el cód. fuente varias veces, usando como entrada el cód. generado en la pasada anterior
- Incremental: genera cód. binario instrucción por instrucción.
- Cross: crea cód. ejecutable para otra plataforma distinta a aquella en la que el compilador se ejecuta



El proceso de compilación



Código ejemplo: Total = Precio * 5

Análisis léxico: distingue palabras clave del lenguaje e identificadores.
se crea la tabla de símbolos.

Ejemplo: id1 = id2 * cte

Análisis sintáctico: detectar la estructura, comparándola con lo permitida por el lenguaje.

Ejemplo: asignación: id1 = "expresión"
producto: "expresión" = id2 * cte

Análisis semántico: asocia a cada variable su tipo, ámbito...
congruencia de tipos en expresiones.

Mapeo de acciones: asocia el código assembler a cada sentencia del programa

Mapa de acciones

alto nivel → assembler

- Tipos de instrucciones
- operaciones aritméticas / lógicas
 - control de flujo del programa
 - movimiento de datos.

Operaciones aritméticas / lógicas

$$\boxed{A = B + 4}$$

```

ld [B], %r2
add %r2, 4, %r2
st %r2, [A]

```

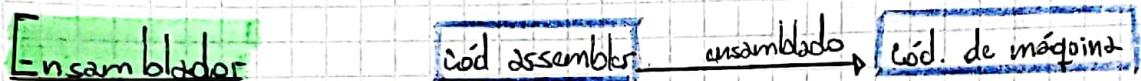
- Modos de direccionamiento a los operadores dependen del ISA
- En máquinas RISC: operandos siempre en registros.
 - cuando se usan todos los registros ⇒ registros al stack

Estructuras de Control del flujo del programa

goto statement	ba label
if A=B stmt1 else stmt2	<pre> subcc %r1, %r2, %r0 ! setcc flags bne elseL !stmt1 ba fin elseL !stmt2 fin: ...</pre>
while(A==B) C++	<pre> ba Test True: add %r3, 1, %r3 Test: subcc %r1, %r2, %r0. be True </pre>
do C++ while(A==B)	<pre> True: add %r3, 1, %r3 Test: subcc %r1, %r2, %r0 be !True </pre>

Almacenamiento de variables

- las variables no están junto al cód. que se ejecuta, sino que están en memoria RAM, y el sist. operativo le da a la aplicación un puntero a la primera línea de cód. y otro puntero a la sección de variables.
- 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.
- Variables en registros



→ programa informático que traduce cód. assembler a un fichero objeto que contiene cód. de máquina, ejecutable directamente por el microprocesador.

transcodificación: relación 1 a 1 entre cód. assembler y cód. de máquina.

- Representación simbólica para direcciones y etc.
- Define la ubicación de las variables en memoria
- Permite tener variables ya inicializadas
- Provee cierto grado de aritmética en tiempo de ensamblado: todo lo que pueda calcularse previo a la ejecución del programa, como aritmética de constantes.
- Permite utilizar variables declaradas en otros módulos.
- Permite dar nombre a fragmentos de cód. (utilizar macros)

$$\begin{aligned} &\text{ej: ld \%r3, [array] + 20, \%r4} \\ &\quad \left\{ \begin{aligned} &= \text{ld \%r3, [2062] + 20, \%r4} \\ &\quad \text{ld \%r3, [2082], \%r4} \end{aligned} \right. \end{aligned}$$

Proceso de ensamblado en dos pasadas

- Preproceso, expansión de macros
 - ↳ i. registros definiciones
 - ↳ ii. reemplazar
- Primera pasada
 - detecta identificadores y les asigna una posición de memoria.
 - crea la tabla de símbolos

NOTA

Segunda pasada

- Cada instrucción es convertida a cód. 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 cód. objeto y el listado

Ejemplo

! dirección

```

! this program sums LENGTH numbers
! registers usage:
!         %r2 - length of array a
!         %r2 - start of array a (address)
!         %r3 - partial sum
!         %r4 - pointer into array a
!         %r5 - holds an element of array a

directivas no cuentan
.a.start      .begin
              .org 2048
              .equ 3000

2048          ld    [%length], %r2
2052          ld    [%address], %r2
2056          andcc %r3, %r0, %r3
2060 loop:     andcc %r2, %r3, %r0
              be    done
              addcc %r2, -4, %r2
              addcc %r2, %r2, %r4
              ld    %r4, %r3
              addcc %r3, %r3, %r3
              bnd  loop;
2088 done:     jmpl %r15 +4, %r0
2092 length:   20
2096 address:  a.start

3000 a.start  .org  a.start
              25
              -10
              33
              -5
              7

              .end

```

empieza a ensamblar
empieza programa en dirección 2048.
→ nuevo símbolo

→ nuevo símbolo
→ nuevo símbolo
→ nuevo símbolo
→ nuevo símbolo
→ nuevo símbolo
→ nuevo símbolo
→ nuevo símbolo
→ nuevo símbolo
termina de ensamblar

Tabla de símbolos, creada en primera pasada

Símbolo	Valor
a.start	3000
length	2092
address	2096
loop	2060
done	2088

- Contador de posición (análogo a un program counter) pero en tiempo de ensamblado:
- Se inicializa en 0 al inicio de la primera pasada y se reinicia por cada directiva .org
- Se incrementa por cada instrucción según su tamaño (en ARC de 4 Bytes)

En segunda pasada crea:

i- Archivo de texto (listado)

Tabla de Símbolos - Listado:

Location Counter	Instruction	Object Code.
2048	ld [length], %r1	11000010 0000...
2052	ld [address], %r2	11000100 0000...
2056	index %r3,%r0,%r1	10000110 1000...
...

ii- Archivo con cód. objetos

- Cód. de máquina
- Dirección de primera instrucción a ejecutar.
- Símbolos creados en otros módulos: externos
- Librerías externas que son utilizadas por el módulo

• Información sobre la relocalización del cód.: si no está disponible la dirección donde empieza el programa, tiene que mover casi todas las direcciones del cód.
Si tiene que juntarse con una librería, el cód. de la librería va a ser desplazado; así que hay que relocalizarlo.
→ sólo se relocaliza: instrucciones del sist. operativo
• direcciones de dispositivos de entrada/salida

- Contenido de la tabla de símbolos

Localización del programa en memoria

- En gral. no se sabe dónde va a ser cargado el programa
- Si varios módulos son vinculados no se sabe dónde va a ser cargada ni los módulos

⇒ el cód. es relocalizable

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

⇒ el ensamblador determina cuáles símbolos son relocalizables y los marca en el módulo ensamblado:
• ignora los extern (se encarga el linker)
• identifica cód. que debe ser modificado debido a la relocalización.

→ no tiene sentido marcar como relocalizables símbolos declarados en otros módulos

Linker

→ Software especializado de programación que gestiona objetos y bibliotecas, combinándolas cuando sea oportuno y liberando los recursos que no están en uso.

- Combina dos o más módulos que fueron ensamblados de manera separada.

↳ pueden ser módulos originarios de:

- aplicación
- librerías del ambiente de desarrollo
- syst. operativo

• Genera el archivo ejecutable → todos los módulos que comprenden al programa y un encabezado donde está la tabla de símbolos y la primera posición de memoria para arrancar el programa.

- Resuelve referencias de memoria externa al módulo.

• Relocaliza los módulos combinándolos y reasignando las direcciones internas a cada uno para reflejar su nueva localización.

↳ Se puede relocalizar: posiciones de memoria relativos a un .org

- no se puede relocalizar: direcciones del syst. operativo, rutinas del syst. operativo y direcciones de dispositivos de entrada/salida.

- Define en el módulo a cargar la dirección de la primera instrucción a ser ejecutada.

Referencias Externas

Símbolos → locales
↳ globales

directivas al ensamblador

• global → declara símbolo global

• extern → usa símbolo declarado en otro módulo

Ejemplo

main program		subroutine library	
.begin		.begin	
.org 2048		.equ L	
.extern sub		.org 2048	
2048 main: ld [x], %r2		.global sub	
2052 ld [y], %r3		brcc %r3, %r0, %r3	
2054 call sub		addcc %r3, ONE, %r3	
2060 jmp %r2, 4, %r0		jmp1 %r1, 4, %r0	
2064 x: 205		.end	
2068 y: 92			
	.end		

main program

Symbol	Value	Global/External	Relocatable
sub	-	external	-
main	2048	no	yes
x	2064	no	yes
y	2068	no	yes
ONE	L	no	no
sub1	2048	global	yes

subroutine
library

Loader

- parte del sist. operativo, cuya función es tomar programas del disco y cargarlos en memoria principal
- Copia el .exe (ejecutable) en memoria RAM
- Hace una llamada a la primera posición de memoria
 - no siempre se coloca el programa en la misma posición de RAM: relocaliza el código
- Algunos loader pueden combinar módulos en tiempo de carga.

En resumen hoy,

• Link Editor: produce versión linkada del programa que normalmente es guardada en archivo para ser ejecutada en otro momento.

• Relocating Loader:

- linkes en tiempo de carga
- relocaliza y carga el programa en memoria para su ejecución
- no es capaz de buscar la librería y cargarla en tiempo de ejecución, con lo cual en disco guarda la librería como parte de mi programa

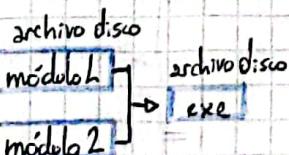
• Linking Loader:

- linkes en tiempo de carga
- relocaliza, busca automáticamente librerías, linkea y carga el programa en memoria para su ejecución
- es como un relocating loader que también busca las librerías, dejando ésta una sola vez en disco, y la levanta cuando corro la aplicación

• Linking Loader Dinámico:

- linkes en tiempo de ejecución (en vez de tener muchas aplicaciones levantando el cod. de la librería al mismo tiempo)
- carga rutinas solo cuando el programa las necesita
- Utiliza Dynamic Link Libraries (dll)
- como un linking loader, carga rutinas solo cuando el programa las necesita, toma la función que necesita de la librería

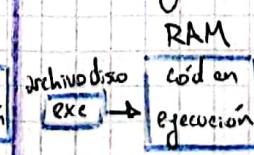
Link Editor / Linker



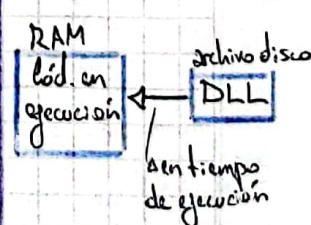
Linking Loader



Relocating Loader



Dynamic Linking



Archivos Objeto → archivo intermedio generado por el compilador antes de crear el ejecutable.

- Relocable: lód. binario y datos en formato que permite combinarlos con otros archivos objeto relocables (linking).
→ como una librería
- Ejecutable: lód. binario y datos en formato que permite cargarlo directamente a memoria y ejecutarlo.
→ como un programa
- Compartido: tipo especial de archivo objeto, relocable que puede ser cargado en memoria y vinculado dinámicamente.

→ el formato de los archivos objeto es dependiente del sist. donde van a ejecutarse

Eficiencia del código - Lenguajes de alto nivel

- Constantes: (`#define`)
 - Ocupa espacio en las instrucciones donde se usa → no se guarda en memoria → no tiene espacio reservado (el const sí)
- VARIABLES
 - Variables pequeñas: mientras ocupen el espacio libre de un registro, se tarda lo mismo es decir, `long` (4 Bytes) tarda lo mismo que `char` (1 Byte)
 - Casteos: para castear hay que extender el signo de la variable más pequeña → mucho más lento.
 - Variables grandes: si superan tamaño de registro (4B en ARC), habrá que usar varios registros → mucho más lento es decir, `long long` (8 Bytes) necesita 2 registros
 - Flot: difícil de manejar (mucho más lento) en comparación con trabajar con enteros. no importa que el procesador tenga un co-procesador matemático para cálculos con float
- Arreglos multidimensionales vs unidimensionales (vectores): es más eficiente trabajar con arreglos unidimensionales porque con los multidimensionales tengo que calcular las posiciones en memoria de cada elemento

- Arreglos inicializados: globales vs locales: son más eficientes los globales ya que lo declaro una sola vez, sino, si se encuentra dentro de la función, cada vez que la invoco, vuelve a declarar el arreglo.
- Invocación a funciones: me da mucho trabajo el tener funciones → más lento.
 - tener una función implica:
 - backup de la dirección de retorno actual (%ret)
 - setear la dirección actual como nuevo punto de retorno (pc)
 - al volver recuperar la dirección de retorno anterior
 - pasar parámetros (si es por stack/memoria es MUCHO más lento que si es por registros)
 - ejecutar código de la función
 - devolver parámetros
 - backup de los registros que utiliza el procedimiento y su recuperación
- Declaración de variables y parámetros: como guardo en orden, primero guardo en registros, y cuando no tengo más registros disponibles guardo en memoria(stack).
 - ⇒ me conviene declarar primero las variables/parámetros que más uso, así no accedo mucho a memoria(stack).