Procesadores de lenguaje

Tema 8 – Generación de código y optimización

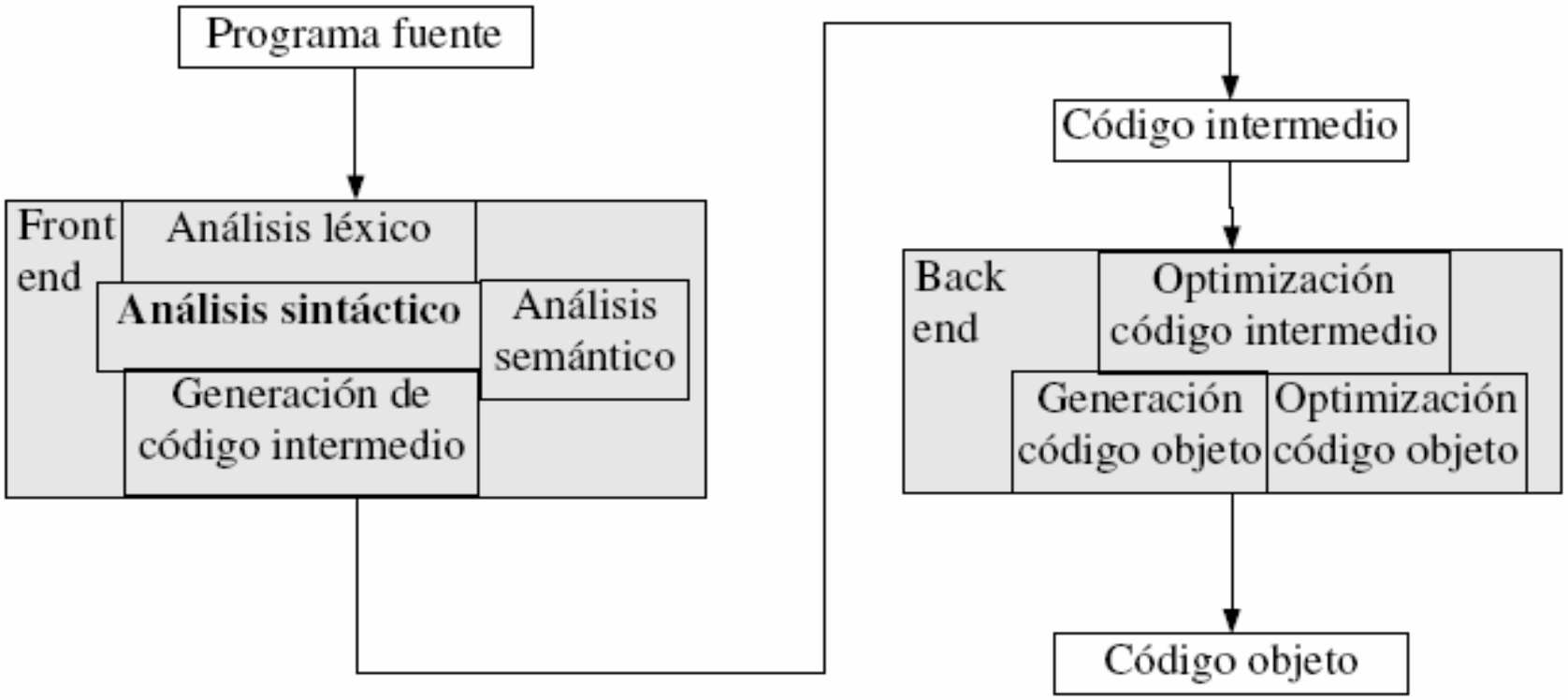
Salvador Sánchez, Daniel Rodríguez
Departamento de Ciencias de la Computación
Universidad de Alcalá

Resumen

- Tipos de código objeto
- Operaciones básicas en la generación de código.
- Optimización del código

Introducción

- En la fase de síntesis (back-end) se genera el código objeto de salida del compilador.



Introducción

- En el caso de lenguajes de programación depende de:
  - código intermedio
  - plataforma de ejecución.
- La generación de código puede funcionar de dos formas:
  - Como una fase independiente y traducir el código intermedio al código objeto equivalente.
  - Integrado con el análisis semántico, generando código objeto a medida que se encuentran construcciones semánticamente correctas.
    - Opción más rápida, pero también más compleja.

Introducción

- Se trata de una etapa de gran importancia: la ejecución de un código objeto eficiente es mucho más rápida que la de un código objeto de poca calidad.

- Sin embargo, una optimización demasiado “agresiva” del código generado puede resultar incluso errónea:

```
int f (int x){
 y = x; // y es una variable global
 return x*x;
}
int main(){
  //…
  printf("La función vale: %d", f(10));
} //fin del programa
```


Tipos de código objeto

- Lenguaje de ensamblador. Al utilizar instrucciones simbólicas, simplifica el proceso de generación de código.
  - Se necesita utilizar un ensamblador para obtener el código máquina.
- Lenguaje máquina absoluto. Este código es directamente ejecutable, ya que utiliza direcciones de memoria absolutas (fijas).
  - Es un código eficiente, pero poco flexible
  - Muy utilizado por los compiladores antiguos, menos usado actualmente.
- Lenguaje máquina relocalizable. Está constituido por módulos objeto. El código se genera con desplazamientos de direcciones, lo cual permite enlazar diferentes módulos objeto (ej., bibliotecas).
  - Muy flexible, permite compilar rutinas por separado y llamar a rutinas ya compiladas en otros módulos.
  - Método más utilizado en compiladores comerciales
  - Es necesario un enlazador para crear el ejecutable.

Generación automática de código

- La conversión de código intermedio en forma de cuádruplas a código ensamblador es sencilla:
  - Cada tipo de cuádrupla se sustituye por un conjunto de instrucciones equivalentes en código ensamblador. Por ejemplo:

```
x = y + z          →    LOAD Reg1, y
                        LOAD Reg2, z
                        ADD Reg2, Reg1, Reg2
                        STR x, Reg2
```

  - Es necesario disponer de un conjunto de rutinas de ayuda que faciliten ciertas tareas propias de la generación de código (escribir, cargar, etc.).

Asignación

- Para las instrucciones de asignación, **`A=B`**, el código en cuádruplas es del tipo **`(assign,A,B,-)`** y el código de ensamblador equivalente que hay que generar es el siguiente:

Código de alto nivel	Código objeto de ensamblador
`a = b`	`LOAD Reg1, b` `STR a, Reg1`

Condicionales

Código de 3 direcciones	Código objeto de ensamblador

`if_false tmp1 goto L1`
`…`
`L1: …`
	`LOAD Reg1, tmp1`
	`FJMP Reg1, L1`
	`…`
	`L1:…`

Arrays

Código de 3 direcciones	Código objeto de ensamblador

`x = A[i]`	`LOAD Reg1, i`
	`MULT Reg1, Reg1, 2 (*)`
	`LOAD Reg2, A(Reg1)`
	`STR x, Reg2`

(*) A es un array cuyos elementos ocupan 2 bytes, p.e. enteros

Otras consideraciones

- Para obtener código de buena calidad:
  - Evitar las cargas y descargas innecesarias desde/hacia memoria
  - Utilizar eficientemente los registros del procesador
    - Son más rápidos que la memoria convencional.
- Las particularidades del *hardware* influyen en la calidad y eficiencia del código generado. Por ejemplo:
  - INC es más eficiente que ADD cuando el incremento es 1.
  - CLEAR es más eficiente que MOV para guardar el valor 0.
  - Etc.

Asignación de memoria

- En la fase de sintesis, la información almacenada en la tabla de símbolos sobre los identificadores (tipo, ámbito, etc.) es necesaria para transformar los nombres en direcciones de memoria.
  - Es necesario reservar un espacio de memoria para guardar:
    - los nombres y su contenido
    - las variables auxiliares definidas por el compilador o información adicional de las subrutinas, por ejemplo la dirección de retorno
- En un lenguaje de programación tenemos distintos tipos de datos, que requieren ser tratados de maneras diferentes:
  - Los que siempre están almacenados en el mismo lugar (por ejemplo las variables globales) pueden tener el espacio reservado en tiempo de compilación.
  - A los que pueden cambiar de lugar durante la ejecución del programa se les debe asignar dirección dinámicamente.

Asignación de memoria estática

- La asignación estática de memoria se utiliza cuando el programa fuente define un identificador al que se debe asignar espacio una sola vez durante el programa.
  - Por ejemplo, una variable con ámbito global o bien un valor estático de una rutina (como el tipo static de C).
- Los lenguajes que utilizan asignación estática de memoria –Fortran– necesitan conocer el tipo y el tamaño de memoria de todos los objetos en compilación:
  - Es decir, hay que declarar los nombres antes de utilizarlos.
- En estos lenguajes no se permiten
  - Llamadas recursivas a subrutinas
  - Estructuras de datos dinámicas.

Asignación de memoria dinámica

- Los métodos de gestión dinámica de la memoria actúan en tiempo de ejecución del programa.
- Permiten asignar y liberar espacio de memoria para un mismo nombre durante la ejecución del programa:
- Las variables de una subrutina no están ligadas a la misma dirección de memoria durante todo el programa: cada vez que se hace una llamada a la subrutina se asignan a nuevas posiciones.
  - Se pueden hacer llamadas recursivas a subrutinas.
  - Se puede reservar una cantidad determinada de espacio de memoria y asignarla a un nombre. Esto permite gestionar listas encadenadas, etc.
- Dos tipos de gestión de memoria dinámica:
  - por pila (recursividad)
  - por montículo (listas y árboles).

Optimización de código

- La velocidad de ejecución del código objeto creado, es de especial importancia
  - Sobre todo en aplicaciones de tiempo real y de cálculo intensivo.
- Dado que el proceso de generación de código objeto es básicamente un proceso de sustitución de macros, el código generado puede acabar siendo de mala calidad.
- La tarea del *proceso de optimización* es mejorar la calidad y eficiencia del código objeto para incrementar la velocidad de ejecución y reducir su espacio ocupado.
- Las optimizaciones de espacio suelen ser incompatibles con las de velocidad:
  - Mejorar la velocidad suele aumentar el espacio requerido y viceversa.

Optimización de código

- El proceso de mejora de código se puede hacer sobre:
  - El código intermedio generado tras el análisis semántico.
  - El código objeto tras la fase de generación de código.
- Las técnicas de optimización se basan en un análisis de la estructura del programa y del flujo de datos que subdividen el programa en regiones de optimización.
- Las técnicas de mejora se pueden aplicar en dos ámbitos:
  - Local, si sólo utiliza información de un bloque básico (ej., un bucle).
  - Global, si utiliza información de un conjunto de bloques básicos.

Bloques básicos de optimización

- Un **bloque básico** es una unidad fundamental de código, una secuencia de instrucciones en la que el flujo de control entra en el inicio del bloque y sale al final.
  - En optimizaciones sobre un bloque básico, los valores de las variables a la entrada y salida del bloque deben coincidir con los del código sin transformar.

- Las mejoras sobre bloques básicos permiten una optimización por partes completas, más eficiente que una optimización por líneas y menos complejo que una optimización global.

Bloques básicos: Ejemplo

Código alto nivel (Fortran)	Bloque Número

`cosc = mta(k) + 2.0 * mt(i)`
`e = abs(cosc)`
`if (codigo .ne. 0) then 10`
	1

`b = 2.0`
`cosc = cosc*2`
`goto 20`
	2

`10: b = 10.0` `cosc = sqrt(cosc)`	3
`20: cuota = (b ** exp) * cosc` `return`	4

Eliminación de subexpresiones comunes

- Si una expresión se calcula más de una vez, se puede utilizar el resultado obtenido la primera vez para reemplazar los cálculos posteriores SIEMPRE QUE los operandos involucrados no se hayan modificado entre ambas apariciones.

```
t1 = 4 - 2
t2 = t1 / 2
t3 = a * t2
t4 = t3 * t1
t5 = t4 + b
t6 = t3 * t1
t7 = t6 + b
c = t5 * t7
```

```
t1 = 4 - 2
t2 = t1 / 2
t3 = a * t2
t4 = t3 * t1
t5 = t4 + b
t6 = t4
t7 = t6 + b
c = t5 * t7
```


Eliminación de código muerto

- Código muerto se refiere a:
  - Código que no se ejecutará nunca.
  - Operaciones insustanciales, por ejemplo, declaraciones nulas o asignaciones de una variable a sí misma.
  - Código no alcanzable.

- Un identificador está vivo al final de un bloque básico si su valor es referenciado en otro bloque básico. Por el contrario, está muerto si no es referenciado en el resto del programa.

Eliminación de código muerto

`b := false;`
`if (a AND b) then`
`instrucciones`
	`b := false ;`

`int global;`
`void f () {`
`  int i;`
`  i = 1;`
`  global = 1;`
`  global = 2;`
`  return;`
`  global = 3;`
`}`
	`int global;`
	`void f () {`
	`  global = 2;`
	`  return;`
	`}`

Propagación de copias

- Después de ejecutar asignaciones de la forma **a = b**, **a** y **b** tienen el mismo valor y por tanto, podemos reemplazar las apariciones de **a** por **b**
- Los identificadores muertos, que aparecen como consecuencia de optimizaciones de propagación de copias, pueden ser eliminadas.

t1 = 4 - 2	t1 = 4 - 2	t1 = 4 - 2	t1 = 4 - 2	t1 = 4 - 2
t2 = t1 / 2	t2 = t1 / 2	t2 = t1 / 2	t2 = t1 / 2	t2 = t1 / 2
t3 = a * t2	t3 = a * t2	t3 = a * t2	t3 = a * t2	t3 = a * t2
t4 = t3* t1	t4 = t3 * t1	t4 = t3 * t1	t4 = t3 * t1	t4 = t3 * t1
t5 = t4 + b	t5 = t4 + b	t5 = t4 + b	t5 = t4 + b	t5 = t4 + b
t6 = t3 *t1	**t6 = t4**	t6 = t4	t6 = t4	
t7 = t6 + b	**t7 = t6 + b**	**t7 = t5**	t7 = t5	
c = t5 * t7	c = t5 * t7	**c = t5 * t7**	**c = t5 * t5**	c = t5 * t5

Optimizaciones aritméticas

- Se pueden aplicar transformaciones para reducir el número de operaciones o sustituir operaciones costosas por otras equivalentes más simples.

- Algunos tipos de transformaciones:
  - Calculo previo de constantes
  - Transformaciones algebraicas

Cálculo previo de constantes

- Cuando en una expresión aparecen diferentes constantes, se pueden combinar en tiempo de compilación para formar una sola.

t1 = 4 - 2	**t1 = 2**			
t2 = t1/2	**t2 = t1/2**	**t2 = 1**	t2 = 1	
t3 = a*t2	t3 = a*t2	**t3 = a * t2**	**t3 = a*1**	t3 = a * 1
t4 = t3*t1	t4 = t3*t1	**t4 = t3 * 2**	**t4 = t3*2**	t4 = t3*2
t5 = t4+b	t5 = t4+b	t5 = t4+ b	t5 = t4 + b	t5 = t4+ b
c = t5*t5	c = t5*t5	c = t5 * t5	c = t5*t5	c = t5*t5

Transformaciones algebraicas

- Cuando en el código aparece una identidad algebraica, se puede simplificar. Las transformaciones más normales son las siguientes:
  - suma: a + 0 = 0 + a = a
  - resta: a - 0 = a
  - multiplicación: a * 1 = 1 * a = a.
  - división: a / 1 = a

Reducción de fuerza

- Esta técnica consiste en sustituir operaciones costosas por otras menos costosas equivalentes.

- Ej.:
  - La multiplicación es más costosa que la suma, ej., `a * 2` → `a + a`
  - La elevación al cuadrado y el producto: x^2 → `x * x`
  - La división y multiplicación por una potencia de dos es más fácil de implantar con desplazamientos binarios.

Reorganizaciones

- Con las propiedades asociativa y distributiva de las operaciones se puede variar el código de las expresiones aritméticas y reducir el número de identificadores involucrados en el cálculo.

x = x + y + x * y	**x = x * y + x + y**

MOV Reg1, x
MOV Reg3, y
ADD Reg1, Reg3
MOV Reg2, Reg1
MOV Reg1, x
MUL Reg1, Reg3
ADD Reg1, Reg2
STR x, Reg1
	MOV Reg1, x
	MOV Reg2, y
	MUL Reg2, Reg1
	ADD Reg2, Reg1
	MOV Reg1, y
	ADD Reg2, Reg1
	STR x, Reg2

Empaquetamientos temporales

- Después de haber aplicado una técnica para mejorar el código, se puede reducir el número de identificadores temporales utilizados.
- Podemos reducir dos nombres temporales a uno solo, si no hay ningún punto en el que los dos estén “vivos” simultáneamente.

Mejora de bucles

- Uno de los puntos donde el programa suele consumir más tiempo de procesamiento es en los bucles.
  - Pequeñas mejoras dentro de un bucle puede representar grandes mejoras en la velocidad de ejecución.
- Técnicas para mejorar la eficiencia de los bucles:
  - Reducción de frecuencia o traslado de código
  - Optimización por combinación
  - Optimización por desarrollo

Reducción de frecuencia

- Consiste en mover de lugar ciertos cálculos para que se ejecuten con menos frecuencia.

```
i := 0
repeat
x := x + h[i + j-k]
  i := i + 3
until ( i > max-3)
```

```
i := 0
t1 := j-k
t2 := max-3
repeat
  x := x + h[i + t1]
  i := i + 3
until ( i > t2)
```


Optimización por combinación

- Esta técnica se basa en transformar dos o más bucles consecutivos en uno solo equivalente que mejore la eficiencia y la velocidad.

```
for i:=1 to 10000 do
  A[i] := 1;
for j:=1 to 10000 do
  B[j] := j;
```

```
for k:=1 to 10000 do
begin
  A[k] := 1;
  B[k] := k;
end;
```


Optimización por desarrollo

- Cuando el cuerpo del bucle tiene pocas instrucciones y el número de repeticiones es pequeño y conocido en tiempo de compilación, es posible evitar todo el control del bucle y sustituirlo por instrucciones secuenciales individuales.

```
for i:=1 to 5 do
h[i] := i;
```

```
h[1] := 1;
h[2] := 2;
h[3] := 3;
h[4] := 4;
h[5] := 5;
```


Bibliografía

- *Básica:*

  - *Construcción de compiladores. Principios y práctica.* Kenneth C. Louden. Thomson-Paraninfo. 2004.

  - *Compiladores: principios, técnicas y herramientas*. A.V. Aho, R. Sethi, J.D. Ullman. Addison-Wesley Iberoamerica. 1990.