Lección 6: Optimización

1. Introducción
2. Mejoras de código fuente
3. Mejoras de código intermedio
4. Mejoras de código de máquina
5. Perspectiva

Lecturas: Scott, capítulo 13
Aho, capítulo 10
Fischer, capítulos 15, 16
Holub, capítulo 7
Bennett, capítulo 11
Muchnick, capítulos 7-20

1/5. Introducción

Premisa de compilación: compilar correctamente, lo más rápidamente posible.

Premisa de optimización: generar el código más eficiente en **tiempo** y/o **espacio**.

- Es difícil demostrar que los resultados son óptimos.
- Eficiencia .vs. tamaño de código.

Optimizaciones posibles:

1. A nivel de código fuente
2. A nivel de código intermedio
3. A nivel de máquina destino

2/5 A nivel de código fuente

- Ejemplo: instrucción de selección:

```
si <exp>
   ent
    <instr1>
   si_no
    <instr2>
fsi
```

```
; <exp>
  JMF SINO
; <instr1>
  JMP FSI
SINO:
; <instr2>
FSI:
```

```
si <exp>
   ent
    <instr1>
fsi
```

```
; <exp>
  JMF FSI
; <instr1>
FSI:
```

Selección:

```
seleccion: tSI expresion
{
   $<sino>$ = nueva etiqueta();
   generar (JMF, $<sino>$);
}
tENT lista_instrucciones
resto_seleccion tFSI
;
resto_seleccion:
{
   etiqueta ($<sino>-2);
}
| tSI_NO
{
   $<fin>$ = nueva etiqueta();
   generar (JMP,$<fin>$);
   etiqueta ($<sino>-2);
}
lista_instrucciones
{
   etiqueta($<fin>2);
}
```

```
si i = 1
   ent i:=i+1;
fsi
```

```
; exp
   SRF n o
   DRF
   STC 1
   EQ
   JMF L0
; 11
   SRF n o
   SRF n o
   DRF
   STC 1
   PLUS
   ASG
L0:
```

Mientras que:

```
while <exp> do <prop>
```

```
   ; while
   ; expresion
L1: <exp>
    JMF L2
   ; do
    <prop>
    JMP L1
L2:
```

```
   ; while
    JMP L1
   ; do
L2: <prop>
   ; expresion
L1: <exp>
    JMT L2
```

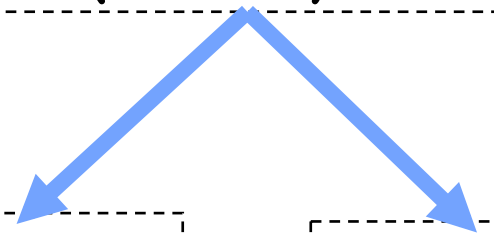
# instr.		# iter.
1	<exp>	
1	JMF	0
1	<prop>	
2	JMP	
2	<exp>	
3	JMF	1
2	<prop>	
4	JMP	
3	<exp>	
5	JMF	2
....	....	

# iter.		# instr.
	JMP	1
	<exp>	1
0	JMT	2
	<prop>	1
	<exp>	2
1	JMT	3
	<prop>	2
	<exp>	3
2	JMT	4
	....	....

Evaluación de constantes

- Efectuar durante la compilación el cálculo de todos los valores constantes.

```
a := (5 + 4) * 100;
```



```
SRF n o
STC 5
STC 4
PLUS
STC 100
TMS
ASG
```

```
SRF n o
STC 900
ASG
```

•Código más pequeño y eficiente.
•Compilador más lento.

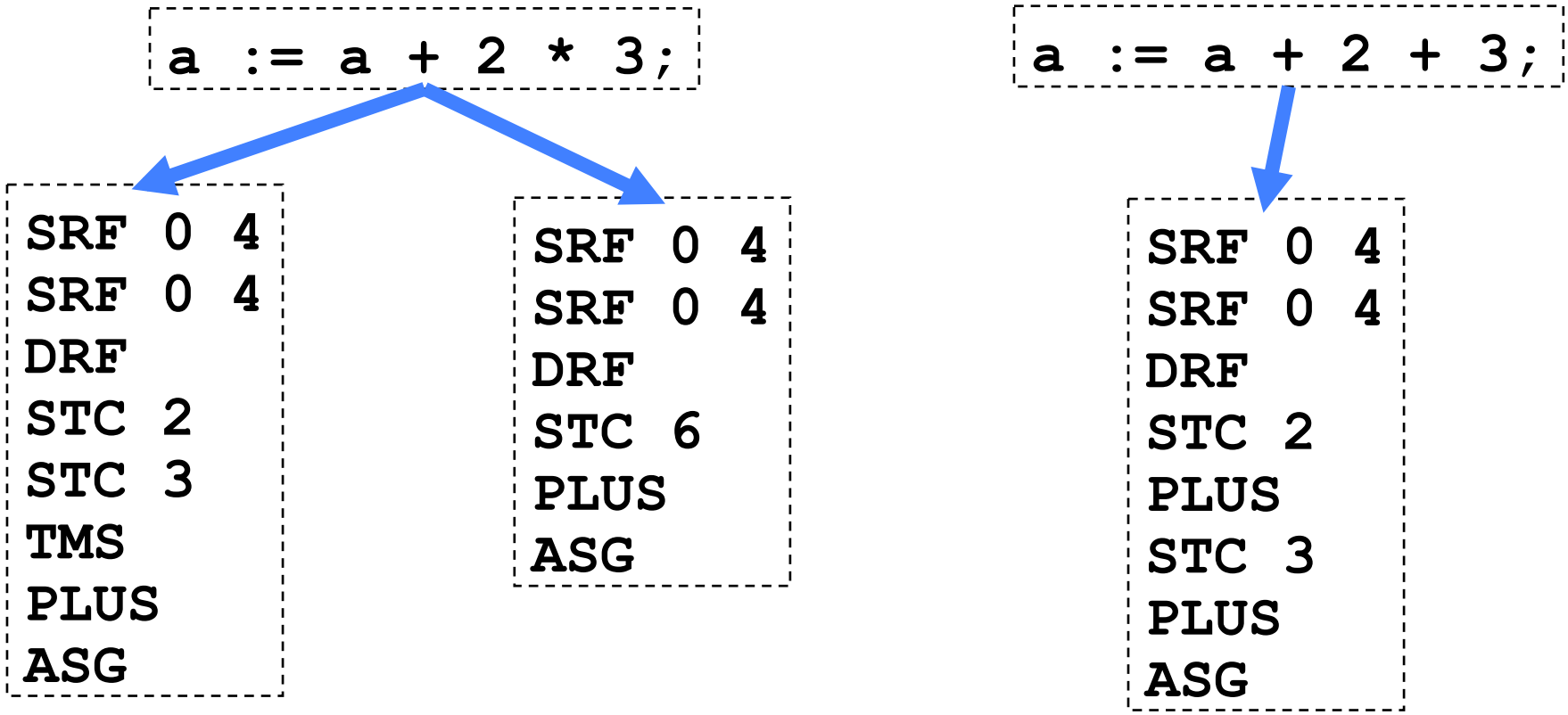
- C: es trabajo del preprocesador

```
#define TFRAMES 100

int frames[TFRAMES];
int stack[2*TFRAMES];
```

Evaluación de constantes

- Las técnicas de análisis sintáctico imponen algunas limitaciones a esta optimización.



Problema: asociatividad de +.

Bucles

- **Desenrollado**: se replica el código del cuerpo del bucle tantas veces como el bucle vaya a ser ejecutado.

```
for (i = 3; i > 0; --i)
    foo (i);
```

```
foo (3);
foo (2);
foo (1);
i = 0;
```

```
STC 3
OSF ...(foo)
STC 2
OSF ...(foo)
STC 1
OSF ...(foo)
SRF 0 i
STC 0
ASG
```

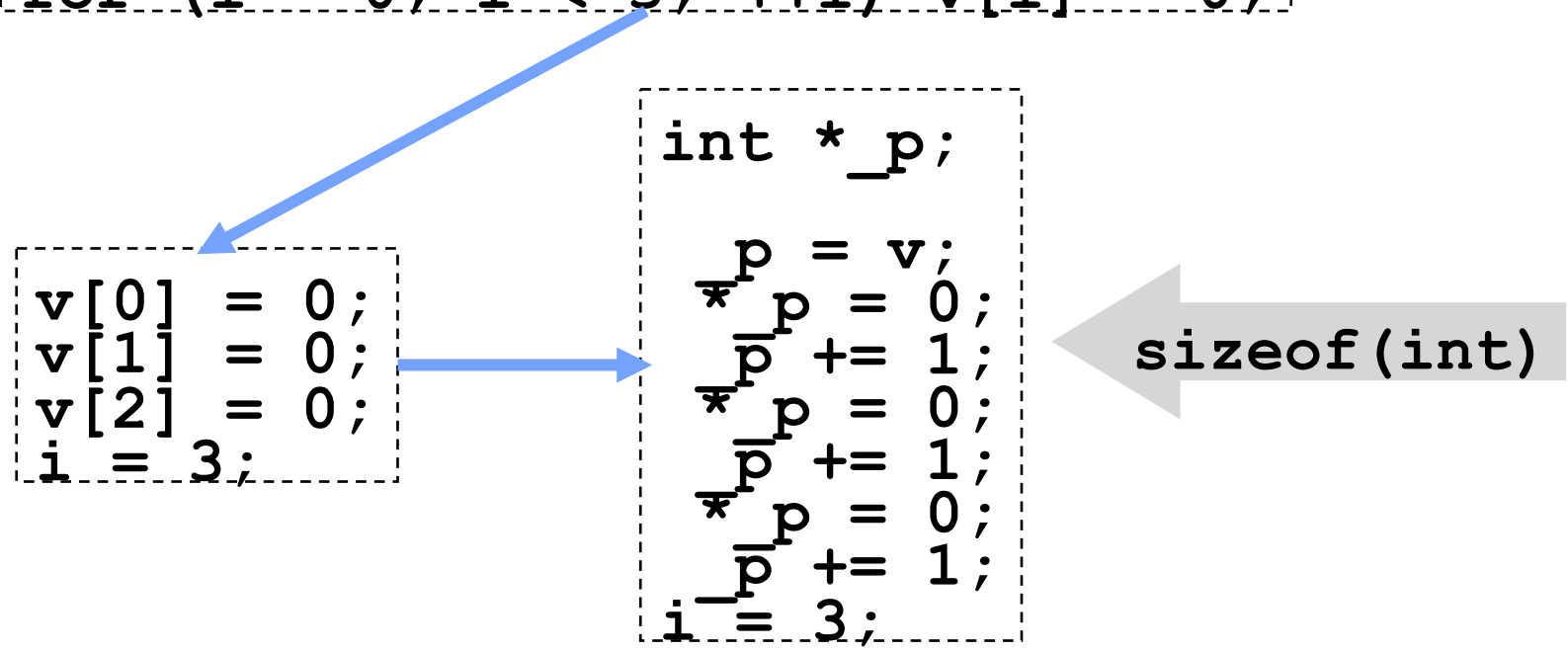
```
    SRF 0 i
    STC 3
    ASG
FOR:
    SRF 0 i
    DRF
    STC 0
    GT
    JMF FFOR
    SRF 0 i
    DRF
    OSF ...(foo)
    SRF 0 i
    SRF 0 i
    DRF
    STC 1
    SBT
    ASG
    JMP FOR:
FFOR:
```

- Compromiso tamaño código .vs. eficiencia ejecución.
- Debe ser posible determinar en compilación el número de iteraciones.

Índices por punteros

- El acceso a vectores mediante índices es lento. En algunos casos, este acceso puede agilizarse utilizando punteros.

```
int v[3];
for (i = 0; i < 3; ++i) v[i] = 0;
```



Análisis complejo. Si la variable `i` se utiliza para algo dentro del bucle habría que modificar su valor en cada iteración.

Reordenación

- La minimización de fallos de cache es de importancia creciente.

```
para i := 1 hasta n
  para j := 1 hasta n
       A[i,j] := 0
  fpara
fpara
```

- Sup. contigüidad por filas, c/línea de cache contiene m elementos, el num. de fallos será: n^2/m

- Sup. contigüidad por columnas, n > m, el número de fallos será:

- **Intercambio:**

```
para j := 1 hasta n
  para i := 1 hasta n
       A[i,j] := 0
  fpara
fpara
```

- Mejora en algunos casos, empeoramiento en otros:

```
para i := 1 hasta n
  para j := 1 hasta n
     A[i,j] := B[j,i];
  fpara
fpara
```

Reorganización

- **Distribución:**

```
para i := 1 hasta n
     A[i] := B[M[i]];
     C[i] := D[M[i]];
fpara
```

- Si solamente caben A y B, o C y D, en la cache a la vez:

```
para i := 1 hasta n
     A[i] := B[M[i]];
fpara
para i := 1 hasta n
     C[i] := D[M[i]];
fpara
```

- **Fusión:**

```
para i := 1 hasta n
     A[i] := A[i]+c;
fpara
para i := 1 hasta n
     si A[i] < 0
     ent A[i] := 0;
     fsi
fpara
```

- Si A no cabe en la cache:

```
para i := 1 hasta n
     A[i] := A[i]+c;
     si A[i] < 0
     ent A[i] := 0;
     fsi
fpara
```

Desplazamiento de código

- Extracción de código que no cambia durante la ejecución del bucle.

```
for (i=0; i<3; ++i)
    v[i] = n/m;
```

```
_r = n/m;
for (i=0; i<3; ++i)
    v[i] = _r;
```

¡CUIDADO!

```
for (i=0; i<3; ++i)
    if (m != 0)
        v[i] = n/m;
```

```
_r = n/m;
for (i=0; i<3; ++i)
    if (m != 0)
        v[i] = _r;
```

Algunos compiladores dejan esta responsabilidad en manos del programador

Inlining de funciones

- El código de la función se introduce directamente en el sitio de la invocación, evitando el paso de parámetros y el manejo de bloques de activación.

original

```
int max (int a, int b)
{
  return (a>b ? a: b);
}
...
i = max (i,j);
```

inline

```
_a = i;
_b = j;
i = (_a >_b ? _a:_b);
```

original

```
   SRF 0 4
   ASGI
   SRF 0 3
   ASGI
   SRF 0 3
   DRF
   SRF 0 4
   DRF
   GT
   JMF L0
   SRF 0 3
   DRF
   JMP L1
L0:
   SRF 0 4
   DRF
L1:
   CSF
```

inline

```
   SRF 0 _a
   DRF
   SRF 0 _b
   DRF
   GT
   JMF L0
   SRF 0 _a
   DRF
   JMP L1
L0:
   SRF 0 _b
   DRF
L1:
```

Inlining de funciones

- ¡NO es equivalente a macros en C!

```
int max (int a, int b)
{
  return (a>b ? a: b);
}
...
i = max (i++,j--);
```

inline

```
_a = i++;
_b = j--;
i = (_a >_b ? _a:_b);
```

```
#define max(a,b) ((a > b? a: b))
...
i = max (i++,j--);
```

macro

```
i = (i++ > j-- ? i++ : j--);
```

3/5. Código Intermedio

- El tipo de código determina qué optimizaciones son posibles.

- **Código intermedio más apropiado:** El TAC permite utilizar optimizaciones que en otros caso solo se pueden llevar a cabo en el código de máquina.

- **Bloque básicos:** Unidad fundamental de código. Secuencia de instrucciones consecutivas donde el flujo de control entra al comien-zo y sale al final.

```
l1:
        t1 := a * b
        t2 := t1 + c
        d := t2 * t2
        ifz d goto l2:
```

El flujo de control puede recibirse de más de un punto y puede ir en más de una dirección.

Bloques Básicos

Obtención de bloques:

1. Encontrar todas las instrucciones que determinan el comienzo de un bloque:
   - Primera instrucción del programa
   - Instrucción etiquetada que es el destino de un *branch*.
   - Instrucción siguiente a un *branch*.
2. Para cada instrucción que determina el comienzo de un bloque, el bloque está compuesto por esa instrucción y todas las siguientes hasta y excluyendo el comienzo de otro bloque o el fin del programa.

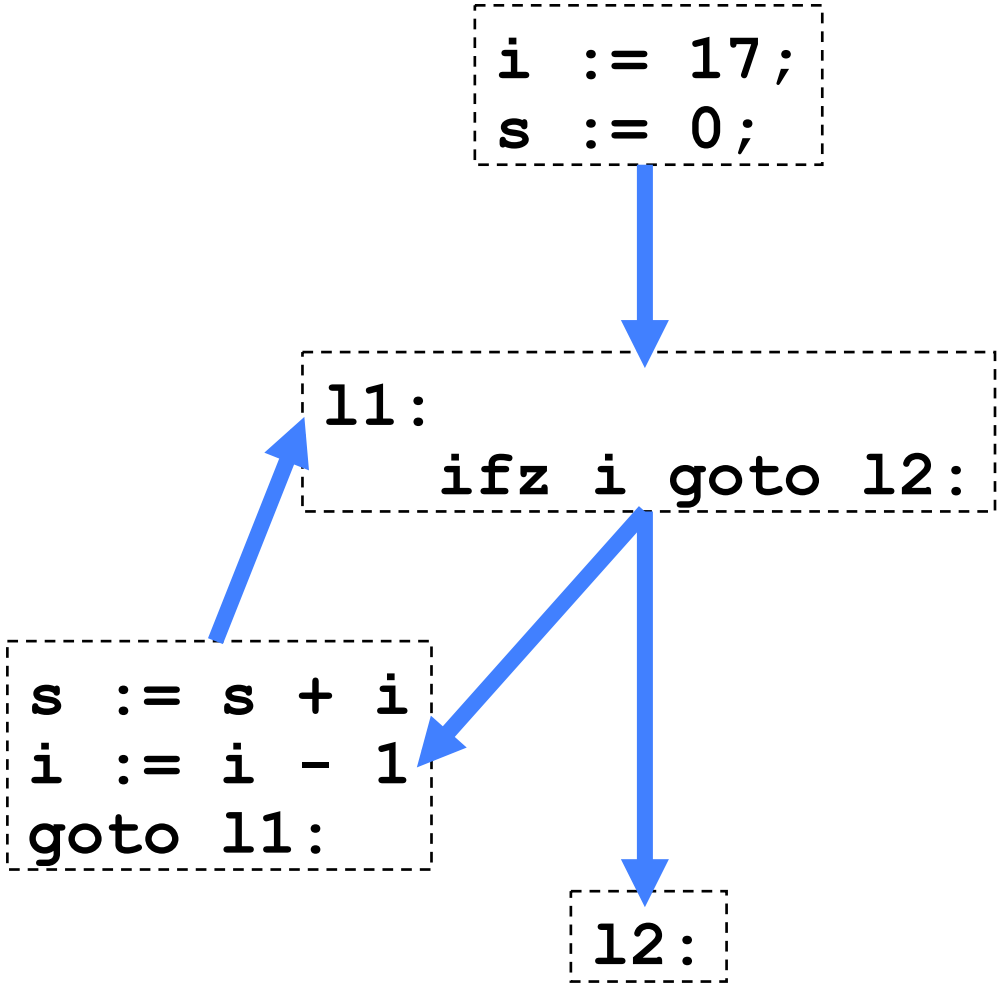
```
program p;

var i, s : integer;

begin
   i := 17;
   s := 0;
   while i <> 0 do begin
         s := s + i;
         i := i - 1
   end
end.
```

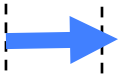
```
    i := 17;
    s := 0;
l1:
    ifz i goto l2:
    s := s + i
    i := i - 1
    goto l1:
l2:
```

Diagramas de Flujo



Optimización en bloques básicos: definiciones

```
b := 4 - 2;
d := (a*b/2*b+c)*(a*b/2*b+c);
```



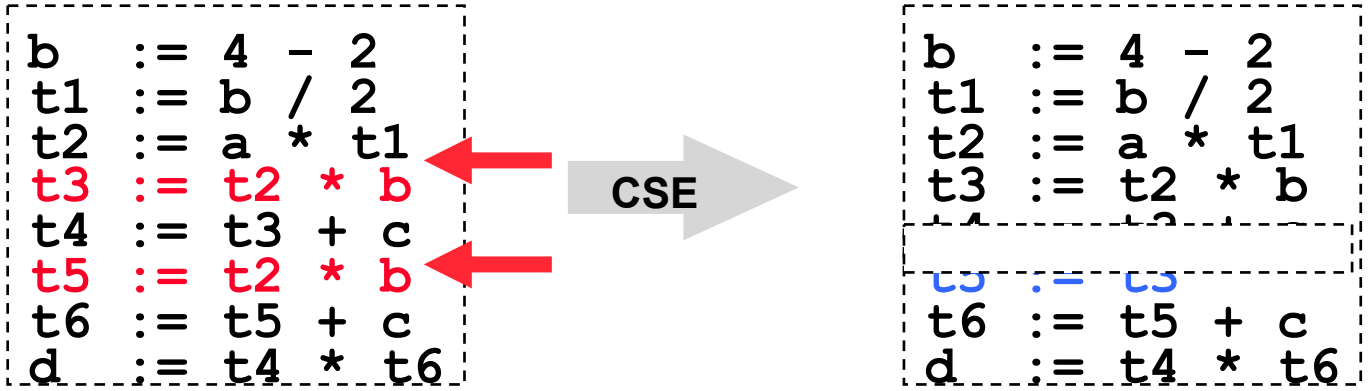
```
b  := 4 - 2
t1 := b / 2
t2 := a * t1
t3 := t2 * b
t4 := t3 + c
t5 := t2 * b
t6 := t5 + c
d  := t4 * t6
```

`a := b op c`

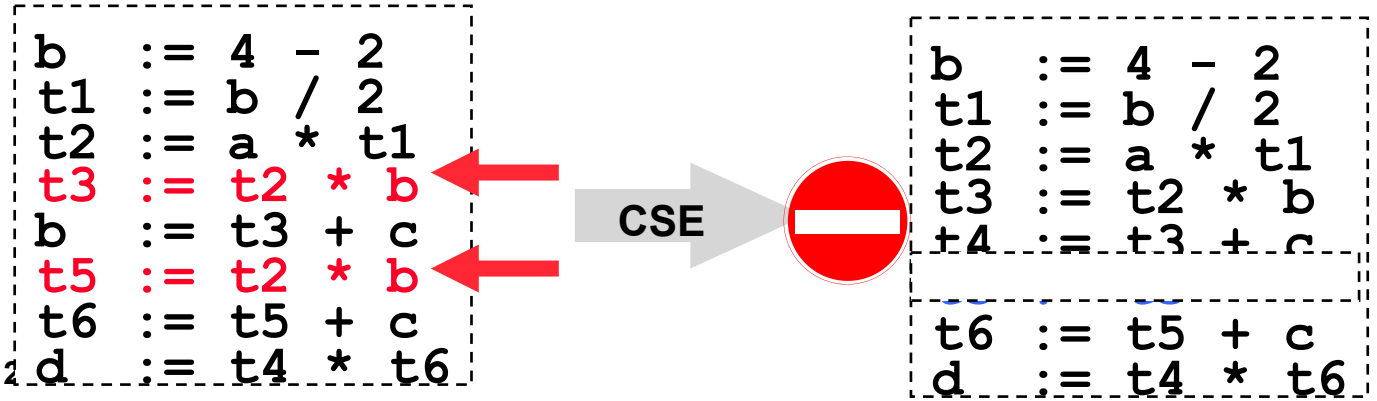
La instrucción *referencia* **`b`** y **`c`** y *define* **`a`**. Un nombre esta *vivo* si se referencia más adelante en un programa, sino está *muerto*.

Eliminación de subexpresiones comunes (CSE)

- Si una expresión se calcula más de una vez, podemos reemplazar el segundo cálculo por el primero.

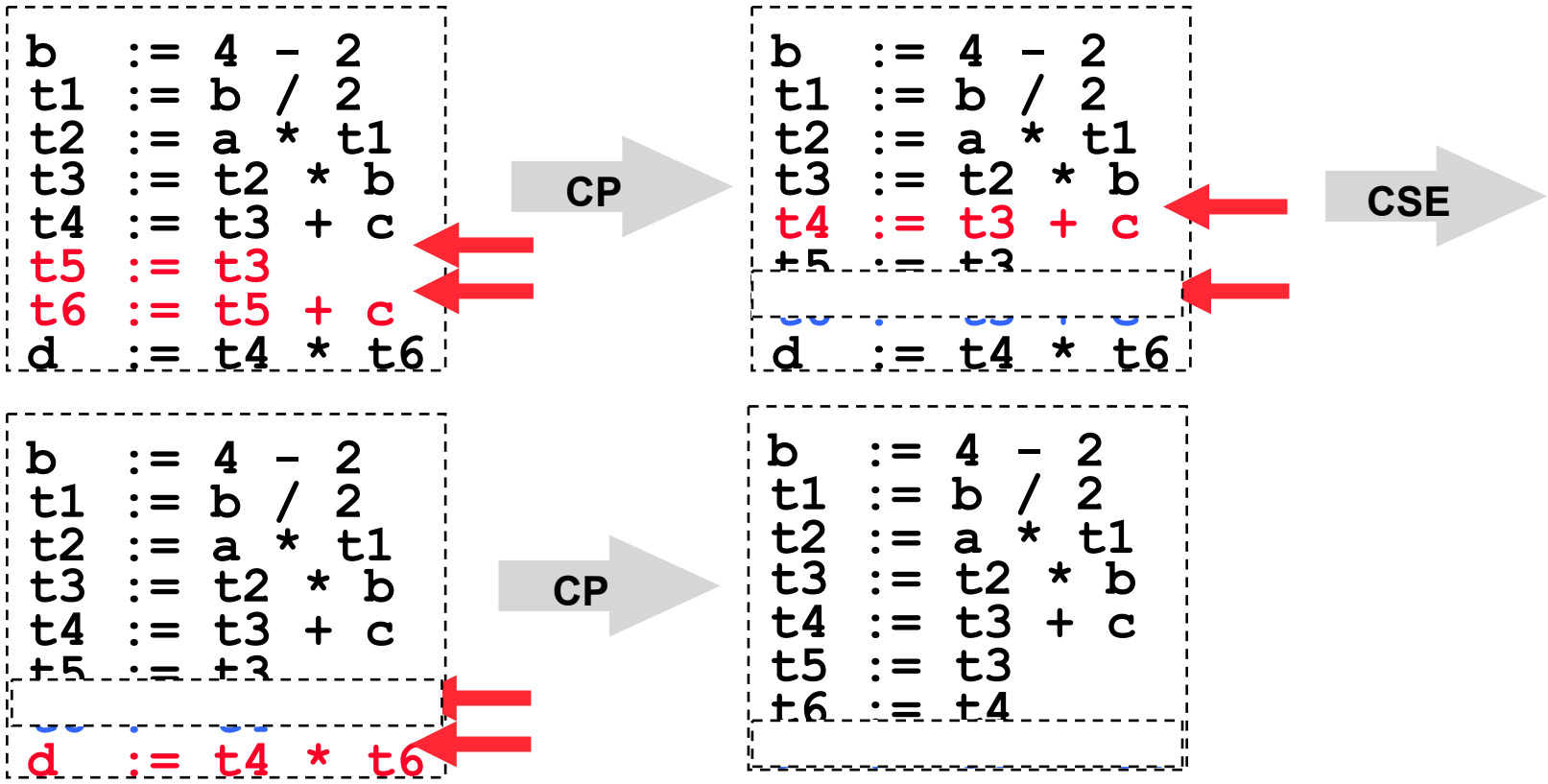


- Los valores de la subexpresión común pueden ser referenciados pero no definidos entre las instrucciones involucradas.



Propagación de la copia (CP)

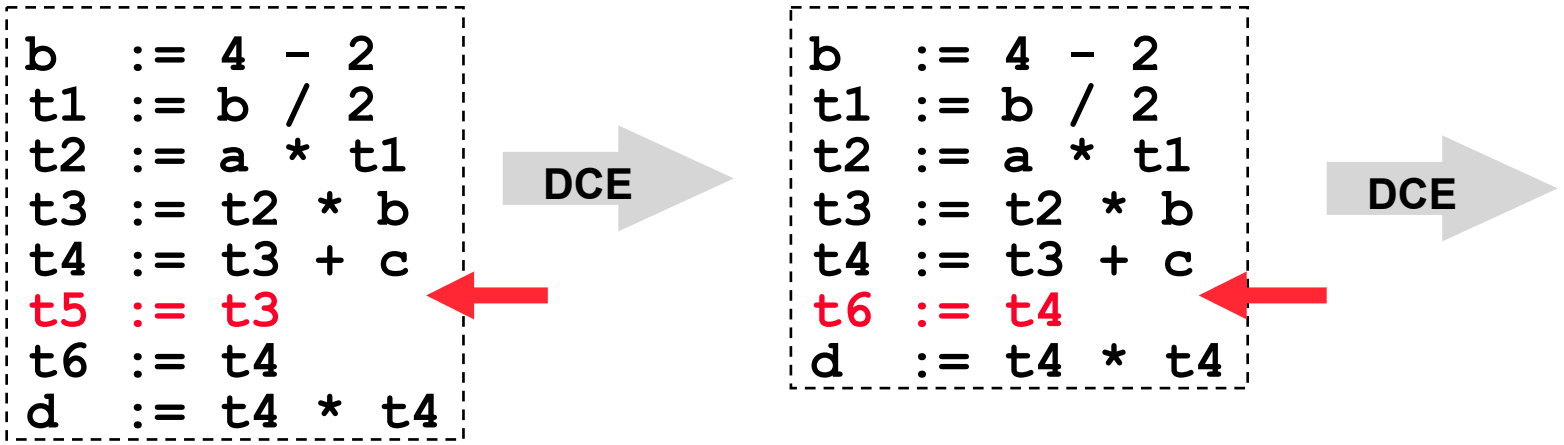
- Después de `a := b`, `a` y `b` tienen el mismo valor. Donde se utilice `a` posteriormente, puede ser reemplazado por `b`.



La propagación de la copia saca a la luz posibles eliminaciones de subexpresiones comunes.

Eliminación de código muerto (DCE)

- No tiene sentido generar instrucciones que definen valores muertos (valores que no se referencian posteriormente).

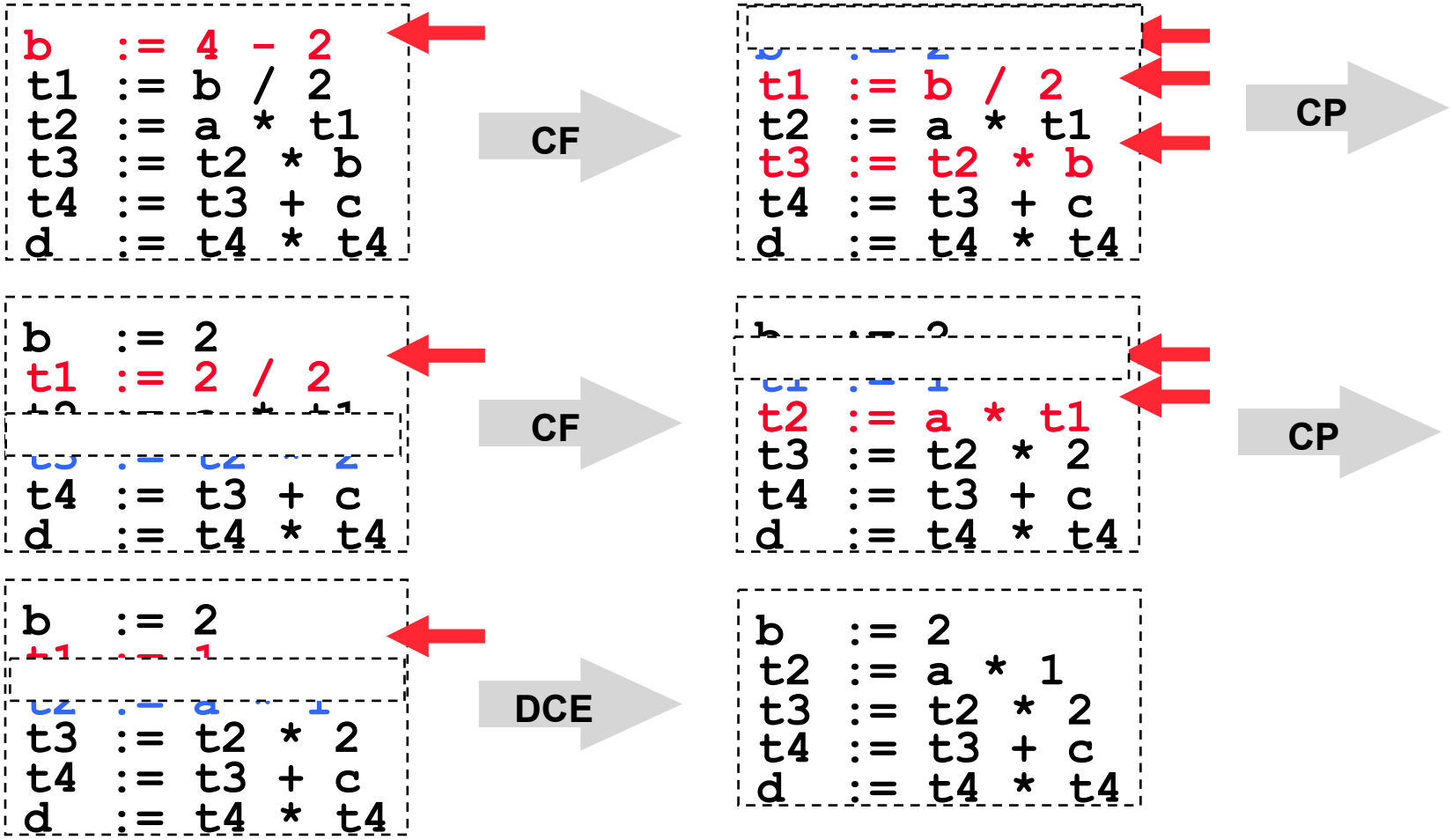


Para simplificar, suponemos que las variables no temporales estarán vivas al final del bloque y las temporales muertas.

Transformaciones aritméticas

- Se utilizan leyes sencillas del álgebra para reducir los cálculos necesarios.

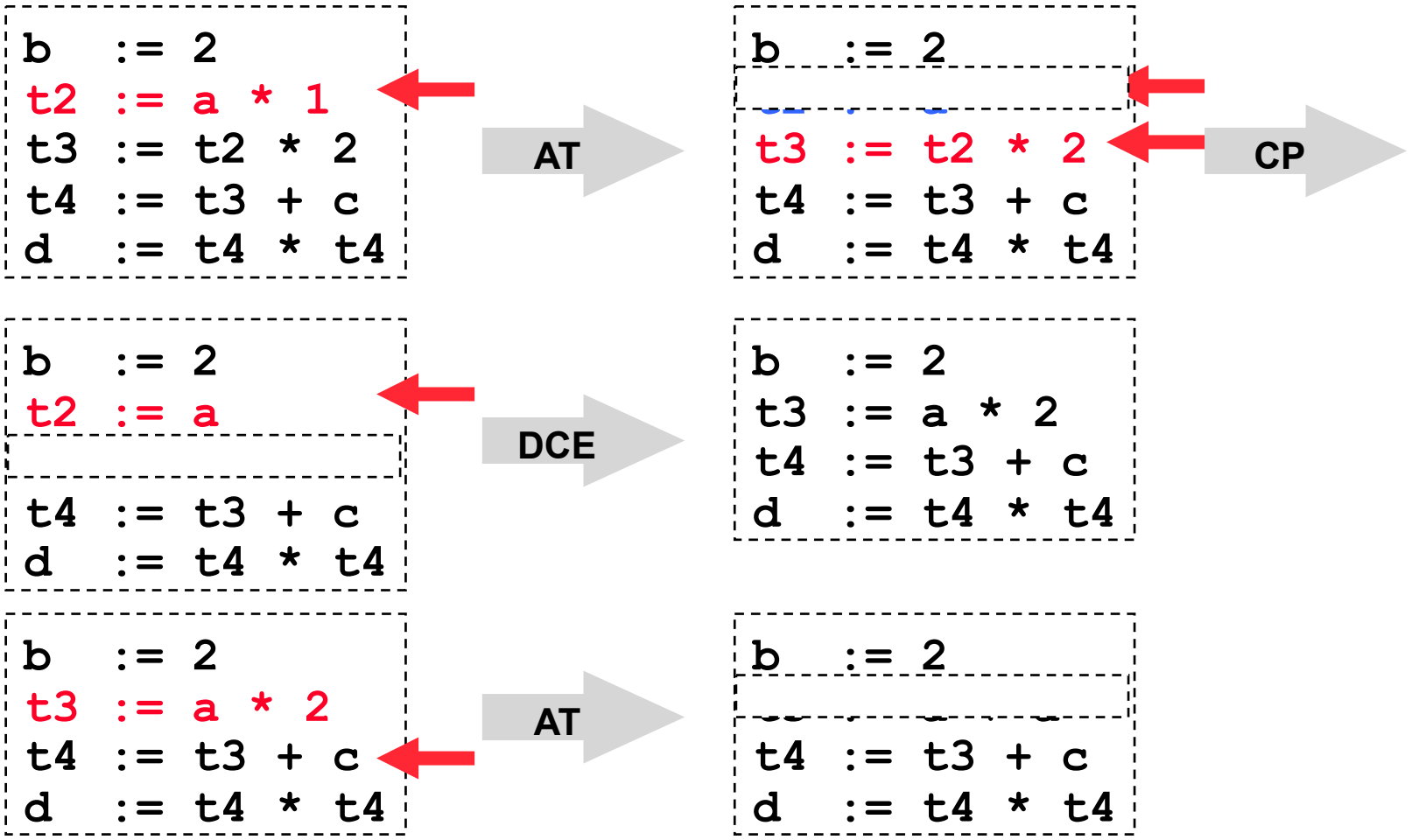
Plegado de constantes (CF)



Transformaciones algebráicas (AT)

$x + 0 = 0 + x = x - 0 = x$
$x \cdot 1 = 1 \cdot x = x / 1 = x$

$x^2 = x \cdot x$
$2 \cdot x = x + x$



Reducción de Intensidad (SR)

- Reemplazar operaciones costosas, como multiplicación o división, por equivalentes más sencillos, como suma y resta.

```
i = j * 21;
```

21 = 10101b

gcc:

SR

```
ld [%fp-16],%o0   ; o0 = j
mov %o0,%o2       ; o2 = j
sll %o2,2,%o1     ; o1 = o2 << 2 = 4j
add %o1,%o0,%o1   ; o1 = o0 + o1 = 5j
sll %o1,2,%o2     ; o2 = o1 << 2 = 20j
add %o2,%o0,%o0   ; o0 = o2 + o0 = 21j
st %o0,[%fp-12]   ; i = o0 = 21j
```

El código generado es más grande, pero las operaciones más rápidas.

Empaquetado de temporales

- Las temporales que no están simultáneamente vivas pueden ser la misma.

t1 y t2 extán muertas
t3 muere

```
b  := 2
t3 := a + a
t4 := t3 + c
d  := t4 * t4
```

original

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

TP

optimizado

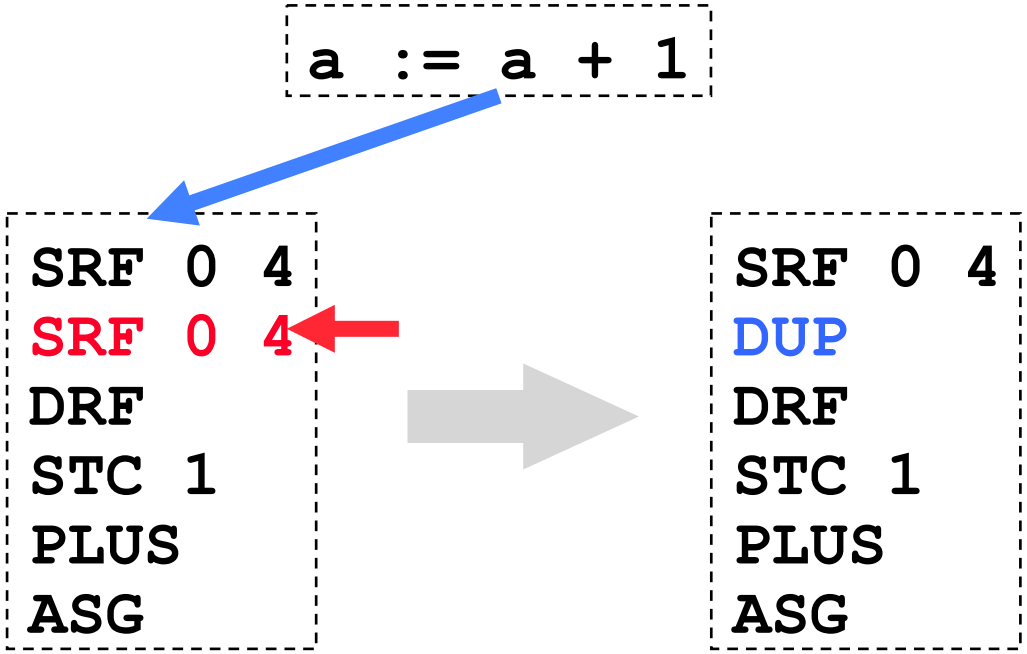
```
b  := 2
t1 := a + a
t1 := t1 + c
d  := t1 * t1
```

Cinco instrucciones y cinco temporales menos;
tres multiplicaciones, una división, y una resta menos.

4/5. Código de Máquina

- ***Optimizaciones de mirilla:*** inspección de secuencias pequeñas de código de máquina para eliminar redundancias y mejorar ciertas ineficiencias.

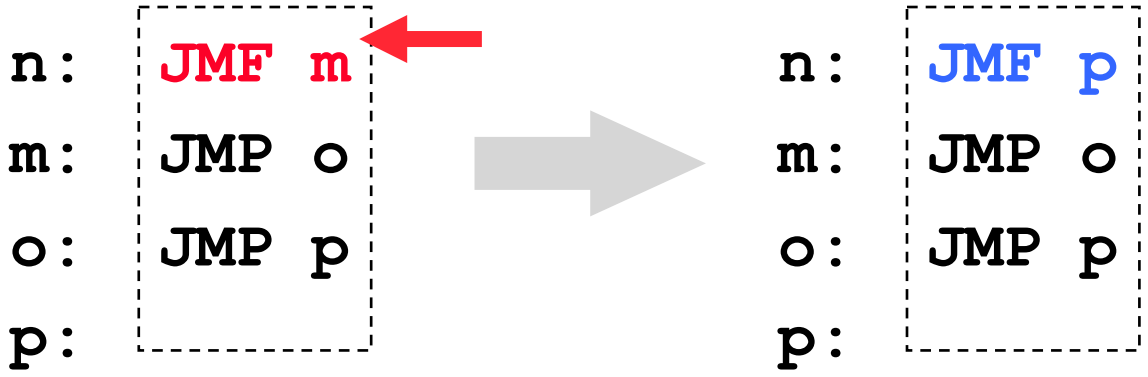
Referencias redundantes



DUP se supone más eficiente

Saltos inútiles o redundantes

- Saltos a la siguiente instrucción, o a otros saltos.



Código inalcanzable

- Código que al que nunca se llega.

```
m:  JMP o
    ...
o:  ...
```

Si en el programa no hay saltos a instrucciones entre la `m+1` y la `o`, es código muerto.

5/5. Perspectiva

- Algunos lenguajes tienen características que facilitan la labor del optimizador.

C:

Si la máquina tiene una instrucción de incremento en 1, puede utilizarse

```
++i;
```

No hace falta calcular dos veces la dirección de la componente del vector.

```
v[i*24-j*400] += 55*l -m
```

Recomienda al compilador almacenar la variable en un registro

```
void f (int a)
{
    register int i;

    for (i=0; i<a; ++i)
        ......
```

Precauciones

- En algunos casos, especialmente al programar hardware, las optimizaciones son ¡nocivas!

Ejemplo: manejo de puertos E/S

```
char function port_send (char c)
{
   volatile char *port, *data;

   /* direccion del puerto */
   port = (char *) 0x10;
   data = (char *) 0x11;

   /* esperar puerto libre */
   while (*port);

   /* almacenar dato en registro de salida */
   *data = c;

   /* dato listo */
   *port = 1;
}
```

¡*data y *port están muertas!

Obliga al optimizador considerar las variables vivas y con valor diferente en todo momento.

Epílogo

- ***¿Desarrollarás un compilador?*** Es poco probable. Los compiladores son herramientas de propósito general ampliamente disponibles comercialmente.

- ***¿Utilizarás un compilador?*** Altamente probable. Son herramientas fundamentales para el desarrollo de aplicaciones. Saber cómo funciona un compilador permite programar mejor.

- ***¿Utilizarás la teoría de compiladores?*** Probable. Muchas aplicaciones generan, reconocen o traducen descripciones formales de los objetos que manejan.