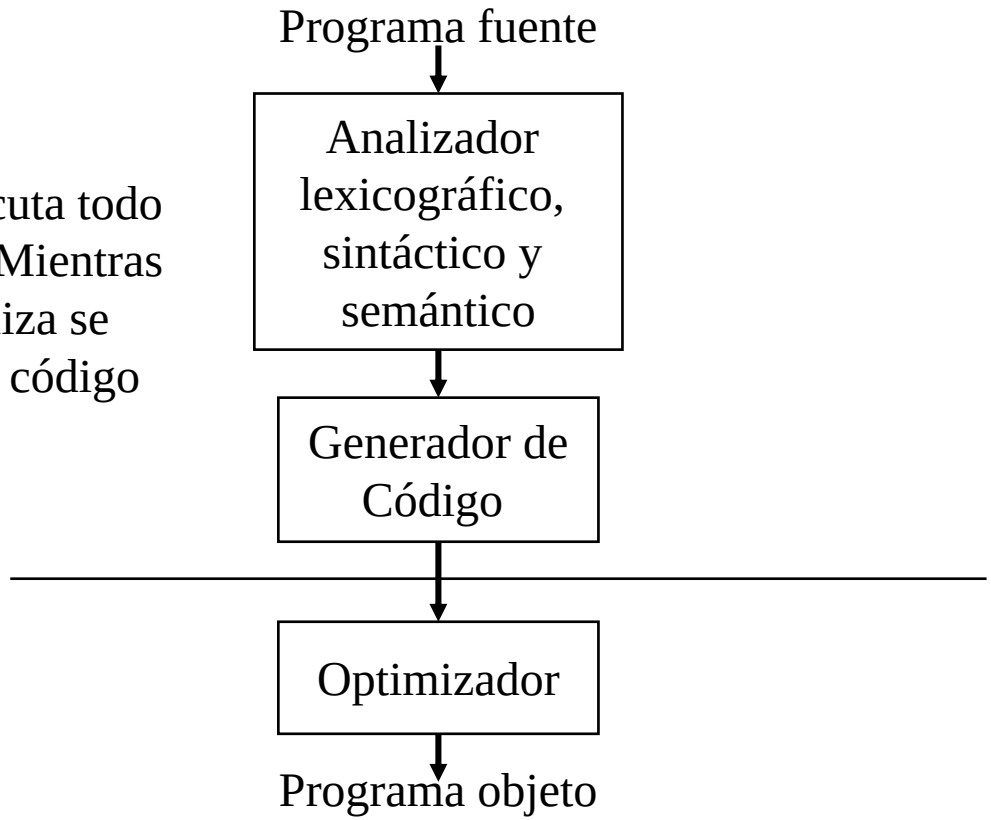
Optimización de Código

Generación de Código y Optimización

- Generación de código
  - Se realiza mientras se analiza el programa
  - “Libre del contexto”
- Optimización
  - Se realiza después de la generación de código de todo el programa o de un elemento ejecutable del programa (función, procedimiento, etc).
  - “Dependiente del contexto”

Generación de Código y Optimización

Se ejecuta todo junto. Mientras se analiza se genera código



Optimización

- Objetivo
  - Obtener código que se ejecuta más eficientemente según los criterios
    - Tiempo de ejecución (optimización temporal)
    - Espacio de memoria utilizado (optimización espacial)
- Funcionamiento
  - Revisa el código generado a varios niveles de abstracción y realiza las optimizaciones aplicables al nivel de abstracción
    - Representaciones de código intermedio de más a menos abstractas
      - Árbol sintáctico abstracto: optimizar subexpresiones redundantes, reducción de frecuencia, etc.
      - Tuplas o cuadruplas: optimizar en uso de los registros o de las variables temporales.
      - Ensamblador/Código máquina: convertir saltos a saltos cortos, reordenar instrucciones

Optimización

- Funcionamiento (continuación)
  - Representaciones de código para extraer información: grafos.
- Condiciones que se han de cumplir
  - El código optimizado se ha de comportar igual que el código de partida excepto por ser más rápido o ocupar menos espacio.
  - Hay que buscar transformaciones que no modifiquen el comportamiento del código según el comportamiento definido para el lenguaje de programación. Ejemplo
    - Si no se ha definido el orden de evaluación de los operandos la siguiente optimización es válida

```
B=2*A+(A=c*d);
```

Pasar a

```
A=c*d;
B=A*3;
```

Optimización Local

- Las optimizaciones locales se realizan sobre el bloque básico
- Optimizaciones locales
  - Folding
  - Propagación de constantes
  - Reducción de potencia
  - Reducción de subexpresiones comunes

Bloque Básico

- Un bloque básico es un fragmento de código que tiene una única entrada y salida, y cuyas instrucciones se ejecutan secuencialmente. Implicaciones:
  - Si se ejecuta una instrucción del bloque se ejecutan todas en un orden conocido en tiempo de compilación.
- La idea del bloque básico es encontrar partes del programa cuyo análisis necesario para la optimización sea lo más simple posible.

Bloque Básico (ejemplos)

- Ejemplos (separación errónea):

```
for (i=1;i<10;++i) {
    b=b+a[i];
    c=b*i;
}
a=3;
b=4;
goto l1;
c=10;
l1: d=3;
e=4;
```

Bloque Básico (ejemplos)

- Separación correcta

```
for (i=1;i<10;++i) {
    b=b+a[i];
    c=b*i;
}
a=3;
b=4;
goto l1;
c=10;
l1: d=3;
e=4;
```

```
BB1:
        i=1;
BB2:
        i<10;
BB3:
        b=b+a[i];
        c=b*i;
        ++i
BB4:
        a=3;
        b=4;
        goto l1;
BB5:
        c=10;
BB6:
        l1: d=3;
        e=4;
```

Ensamblamiento (Folding)

- El ensamblamiento es remplazar las expresiones por su resultado cuando se pueden evaluar en tiempo de compilación (resultado constante).
  - Ejemplo: A=2+3+A+C -> A=5+A+C
- Estas optimizaciones permiten que el programador utilice cálculos entre constantes representados explícitamente sin introducir ineficiencias.

Implementación del Folding

- Implementación del floding durante la generación de código realizada conjuntamente con el análisis sintáctico
  - Se añade el atributo de constante temporal a los símbolos no terminales y a las variables de la tabla de símbolos.
  - Se añade el procesamiento de las constantes a las reglas de análisis de expresiones.
  - Optimiza: 2+3+b -> 5+b
    - Hay una suma de constantes (2+3)+b
  - No optimiza: 2+b+3 -> 2+b+3
    - No hay una suma de constantes (2+b)+3

Implementación del Folding

- Implementación posterior a la generación de código
  - Buscar partes del árbol donde se puede aplicar la propiedad conmutativa:
    - Sumas/restas: como la resta no es conmutativa se transforma en sumas: a+b-c+d -> a+b+(-c)+d
    - Productos/divisiones: como la división no es conmutativa se transforma en productos: a*b/c*e -> a*b*(1/c)*e
  - Buscar las constantes y operarlas
  - Reconstruir el árbol

Ejemplo de Folding

Expresión: `3-(5+6)-A*10+4`

Árbol:

```
-
+- +
|  +- -
|  |  +- 3
|  |  +- +
|  |     +- 5
|  |     +- 6
|  +- *
|     +- A
|     +- 10
+- 4
```

Términos:

3
-5
-6
-(A*10)
4

Términos constantes:

3-5-6+4 = -4

Resultado: `-4-(A*10)`

Propagación de constantes

- Desde que se asigna a una variable un valor constante hasta la siguiente asignación, se considera a la variable equivalente a la constante.
- Ejemplo: Propagación Ensamblamiento

  PI=3.14 -> PI=3.14 -> PI=3.14

  G2R=PI/180 -> G2R=3.14/180 -> G2R=0.017

  PI y G2R se consideran constantes hasta la próxima asignación.
- Estas optimizaciones permiten que el programador utilice variables como constantes sin introducir ineficiencias. Por ejemplo en C no hay constantes y será lo mismo utilizar
  - Int a=10;
  - #define a 10

  Con la ventaja que la variable puede ser local.
- Actualmente en C se puede definir const int a=10;

Implementación de la Propagación de Constantes

- Separar el árbol en bloques básicos
  - Cada bloque básico será una lista de expresiones y asignaciones
- Para cada bloque básico
  - Inicializar el conjunto de definiciones a conjunto vacío.
    - Definición: (variable,constante)
  - Procesar secuencialmente la lista de expresiones y asignaciones
  - Para expresión y asignación
    - Sustituir las apariciones de las variables que se encuentran en el conjunto de definiciones por sus constantes asociadas.
  - Para asignaciones
    - Eliminar del conjunto de definiciones la definición de la variable asignada
    - Añadir la definición de la variable asignada si se le asigna una constante

Ejemplo de Separación en Bloques Básicos

```
Fun
+- []
+- =>
   +- f
   |  +- x
   |  +- y
   +- InstrComp
      +- ;
         +- =
         |  +- i
         |  +- 0
         +- ;
            +- =
            |  +- b
            |  +- 5
            +- ;
               +- while
               |  +- <
               |  |  +- i
               |  |  +- x
               |  +- InstrComp
               |     +- ;
               |        +- Print
               |        |  +- *
               |        |     +- i
               |        |     +- b
               |        +- =
               |           +- i
               |           +- +
               |              +- i
               |              +- 1
               +- *
                  +- x
                  +- b
```

```
Fun f(x,y)=>
    {
        i=0;
        b=5;
        while (i<x) {
            print(i*b);
            i=i+1;
        }
        x*b
    }
```

i=0;
b=5;

i<x

print(i*b);
i=i+1;

X*b

Cada bloque básico es una lista de apuntadores a las expresiones o asignaciones del árbol

Ejemplo

Código	Resultado	Conjunto de definiciones
		{}
a=10;	a=10;	
		{(a,10)}
b=c*a;	b=c*10;	
		{(a,10)}
a=b+a;	a=b+10;	
		{}

Extensión de la P.C. a Variables Indexadas

- Variables indexadas
  - Se asocian constantes a expresiones de acceso a variables indexadas. Ejemplo
    - A[i]=10
    - Se asocia 10 a A[i] aun no sabiendo el valor de i.
    - En el caso de hacer una asignación a cualquier elemento de A se deshace la asociación.
- Ejemplo

		{}
a[i]=10;	a[i]=10;	{(a[i],10)}
b=a[i]+a[j];	b=10+a[j];	{(a[i],10)}
a[j]=20;	a[j]=20;	{(a[j],20)}
b=a[i]+a[j];	b=a[i]+20;	{(a[j],20)}
		{(a[j],20)}

Extensión de la P.C. Fuera del Bloque Básico

- Extender la aplicación fuera del bloque básico
  - Hay que realizar un análisis del flujo de ejecución y es complejo. Ejemplo:

```
i=0;
loop:
i=i+1
if (i<10) goto loop;
```

Se transforma **erróneamente** en

```
i=0;
loop:
i=1
if (1<10) goto loop;
```

Reducción de potencia(strength reduction)

- Se busca sustituir operaciones costosas por otras mas simples. Ejemplo:
  - sustituir productos por sumas.

    a=2*a

    a=a+a
  - Evitar la operación append (++)

    A=length(s1 ++ s2)

    convertirlo en

    A=length(s1)+length(s2)

Implementación de la Reducción de Potencia

- Buscar subárboles que se puedan sustituir por otros menos costosos. Estas sustituciones se representan como reglas que aplicará la reducción de potencia
  - Exp*1 -> Exp
  - 1*Exp -> Exp
  - Exp+0 -> Exp
  - 0+Exp -> Exp
  - Exp*cte(110) -> (Exp << 2)+(Exp << 1)
  - Exp*0 -> { llamar a las funciones de Exp } ;0
  - Etc.

Eliminación de subexpresiones redundantes.

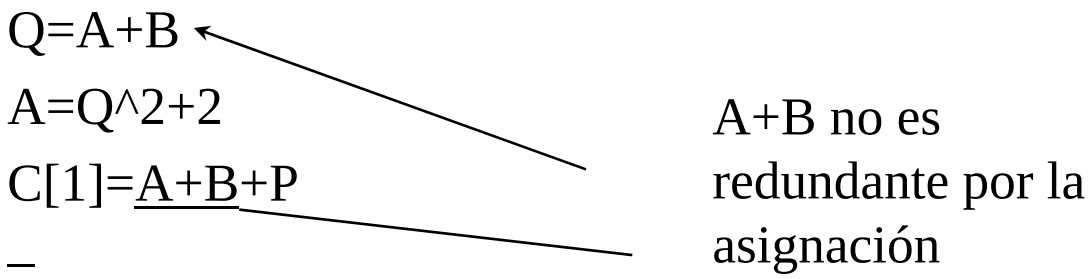
- Las subexpresiones que aparecen más de una vez se calculan una sola vez y se reutiliza el resultado.
- Idea: Detectar las subexpresiones iguales y que las compartan diversas ramas del árbol. Problema: Hay que trabajar con un **grafo acíclico**.
- Dos expresiones pueden ser equivalentes y no escribirse de la misma forma A+B es equivalente a B+A. Para comparar dos expresiones se utiliza la **forma normal**
    - Se ordenan los operandos: Primero las constantes, después variables ordenadas alfabéticamente, las variables indexadas y las subexpresiones. Ejemplo

      X=C+3+A+5 -> X= 3+5+A+C

      Y=2+A+4+C -> Y=2+4+A+C
    - divisiones y restas se ponen como sumas y productos para poder conmutar
        - A-B -> A+ (-B)
        - A/B -> A * (1/B)

Implementación

- Primero se aplica el folding y la propagación de constantes ("simplificar las expresiones")
- Después se reordena el árbol sintáctico hasta obtener la forma normal.
- Se inicia la eliminación de las subexpresiones redundantes.
- Hay que considerar las asignaciones que pueden convertir una subexpresion redundante en no redundante. Ejemplo:



- Sustituir todas las variables asignadas por la expresión que se les ha asignado
  - Generar un orden de ejecución
  - Sustituir siguiendo este orden de ejecución

Implementación

- Comparar los subárboles desde las ramas hacia la raíz y sustituir los subárboles repetidos.
- Recorrer el grafo acíclico generado y sustituir la primera aparición de cada rama que sea hijo de dos o más nodos por una asignación a una variable local y las otras ramas por la variable local.

Optimizaciones Dentro de Bucles

- La optimización de bucles es muy importante por las mejoras en tiempo de ejecución que se obtienen
- Estrategias de optimización dentro de bucles
  - Expansión de bucles (loop unrolling)
  - Reducción de frecuencia (frequency reduction)
  - Reducción de potencia (strength reduction)

Expansión de bucles(loop unrolling)

- La expansión de bucles solo se puede aplicar a los bucles cuyo número de iteraciones se conoce en tiempo de compilación. Ejemplo:
  - Se puede aplicar a los bucles

    for i=1 to 10 do …
  - No se puede aplicar a los bucles

    for i=a to b do …
- La expansión de un bucle puede ser muy costosa en espacio. Hay que poner un criterio heurístico para decidir si se aplica la expansión.
  - Se puede aplicar una expansión parcial en la que sigue existiendo el bucle, pero cada iteración del nuevo bucle corresponde a varias iteraciones del bucle original.
- En un bucle expandido se ha de sustituir el índice del bucle por el valor constante correspondiente

Reducción de frecuencia. (frequency reduction)

- La reducción de frecuencia detecta las operaciones invariantes de bucle y las calcula una única vez delante del bucle. Ejemplo:

  for i=1 to n do c=i*sin(a);

  – sin(a) es una operación invariante del bucle que puede pasar de calcularse n veces a una con la siguiente transformación

  tmp=sin(a);

  for i=1 to n do c=i*tmp;

  – Esta transformación no tiene en cuenta que el bucle igual no se ejecuta ninguna vez y esto supondría perder tiempo de ejecución calculando la invariante de bucle innecesariamente. Además el calculo de la invariante puede producir un error de ejecución. Por ejemplo una división por cero.

Reducción de frecuencia. (frequency reduction)

- Las invariantes de bucle sólo pueden estar formadas por operaciones que son funciones puras
  - Su único efecto es el cálculo del resultado
  - El resultado solo depende de los operandos.
- Ejemplo:
  - Invariantes: +. -, *, / , sin, ln
  - No invariantes: printf, getchar, ++, random
- Consideración práctica
  - Sólo pueden ser invariantes operaciones primitivas del lenguaje (son aquellas que implementa directamente el compilador). Ejemplo: operaciones aritméticas, comparaciones, etc.
  - Algunos compiladores permiten indicar que una función es pura para que pueda formar parte de invariantes

Implementación de la Reducción de Frecuencia

- Pasos
  - Detectar todas las variables que se modifican en el bucle.
  - Marcar todas las operaciones no invariantes
  - Aplicar la siguiente regla recursiva para detectar las operaciones invariantes
    - Una operación es invariante si sus operandos son invariantes.
  - Asociar a cada expresión invariante una variable temporal.
  - Sustituir en el bucle las operaciones invariantes por las correspondientes variables.
  - Añadir delante del bucle la asignación de la expresión invariante a su variable temporal.
- Problema del método anterior
  - Cuando no se entra en el bucle se calculan sus operaciones invariantes. La solución es evaluar antes la condición de salida del bucle.

Patrones de transformación que utiliza la reducción de frecuencia

- while (*cond*) *instrucción*

  if (*cond*) {

  cálculo de invariantes;

  do *instrucción* while (*cond*);

  }

- do *instrucción* while (*cond*);

  cálculo de invariantes;

  do *instrucción* while (*cond*);

- for (*inicialización*; *condición*; *incremento*) *instrucción*

  inicialización;

  if (*cond*) {

  cálculo de invariantes;

  do *instrucción*; *incremento*; while (*cond*);

  }

Reducción de potencia(strength reduction)

- Reducción de potencia aplicada a bucles
  - Sustituir productos entre variables inductivas e invariantes de bucle por sumas

    for(i=1; i<10;++i) a[i]=3*i;

    convertir en

    for(i=1,j=3;i<10;++i,j+=3) a[i]=j;
- Problemas a resolver
  - Detectar las invariantes de bucle
    - Ya esta solucionado
  - Detectar las variables inductivas

Variables Inductivas

- Una variable V es inductiva cuando la única forma en que se modifica su código es V=V+K, donde K es una invariante de bucle.
- Se considerará la necesidad de generar una variable inductiva temporal T a partir de encontrar expresiones de la forma V*C, donde C es una invariante de bucle.
  - Se sustituirá V*C por T
  - Se inicializa T después de la inicialización de V como T=V*C (solo se ejecuta al entrar en el bucle)
  - Al final de cada iteración se añade T=T+C*K

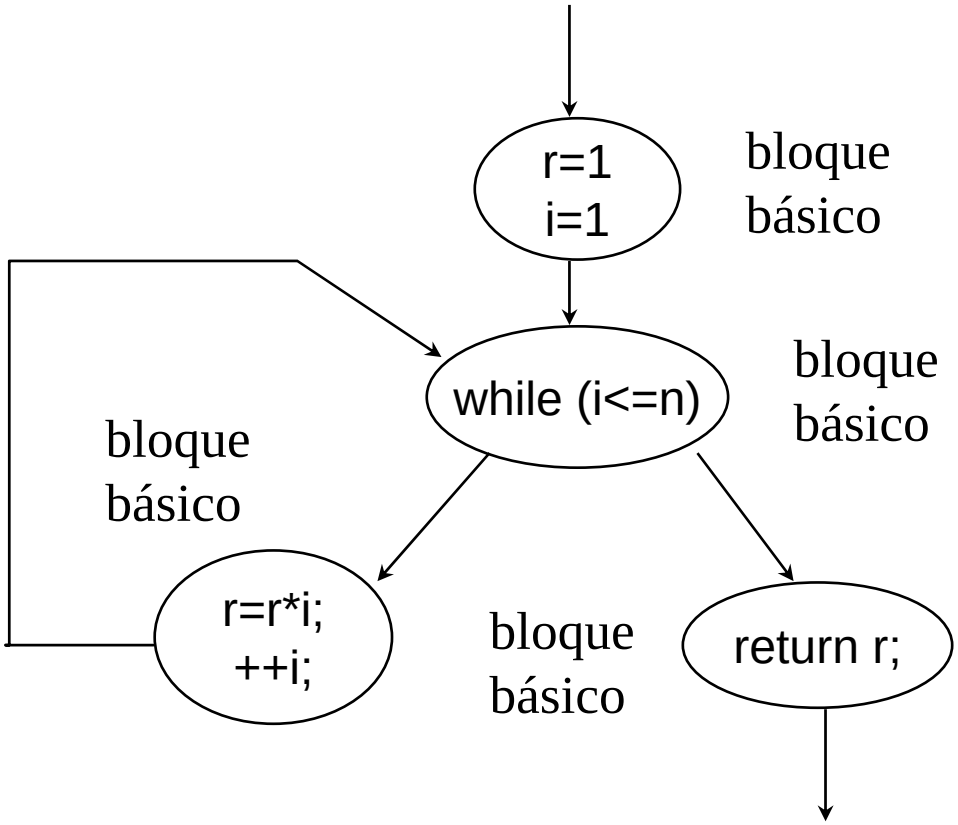
Optimización Global

- Grafo del flujo de ejecución
  - Antes de realizar una optimización global es necesario crear el grafo de flujo de ejecución.
  - El grafo de flujo de ejecución representa todos los caminos posibles de ejecución del programa.
  - La información contenida en el grafo es útil para
    - el programador y
    - el optimizador
- La optimización global a partir del análisis del grafo del flujo de ejecución permite
  - Una propagación de constantes fuera del bloque básico.
  - Eliminación del código no utilizado
  - Una mejor asignación de los registros.
  - Etc.
- Problema: la optimización global es muy costosa en tiempo de compilación

Construcción del Grafo del Flujo de Ejecución

- Tipos de grafo
  - Orientado a procedimiento/función
  - Grafo de llamadas
- Ejemplo

```
int fact(int n) {
    int r;
    r=1;
    i=1;
    while (i<=n) {
            r=r*i;
            ++i;
    }
    return r;
}
```



Construcción del Grafo del Flujo de Ejecución

- Pasos
  - Dividir el programa en bloques básicos
    - Se representa el programa en un código intermedio donde queden explícitamente representados los saltos condicionales e incondicionales.
    - Un bloque básico será cualquier trozo de código que no contenga saltos ni etiquetas en su interior (es posible tener etiquetas al inicio del bloque y saltos al final).
  - En el grafo, los vértices representan los bloques básicos y las aristas representan los saltos de un bloque básico a otro.

Detección de Código no Utilizado (Código Muerto)

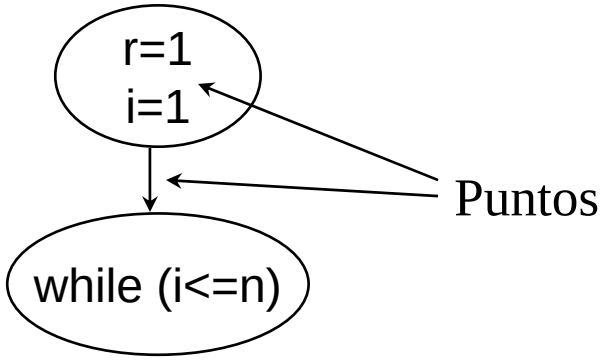
- Código muerto: es código que nunca se ejecutará.
- Detección de código no utilizado
  - El código no utilizado son los bloques básicos donde no llega ninguna arista.
  - Se quita el bloque y las aristas que salen de él.
  - Repetir hasta no eliminar ningún bloque básico.

Análisis del Grafo del Flujo de Ejecución

- Hay que considerar como la información sobre las variables y expresiones se propaga a través del grafo.
- Esta información la representaremos en forma de conjuntos como los siguientes:
  - expresiones disponibles
  - Alcance de las definiciones
  - variables vivas
  - expresiones muy utilizadas

Análisis del Grafo del Flujo de Ejecución

- Los conjuntos se calculan para los puntos del grafo de flujo de ejecución.
- Un punto es una posición dentro del grafo y se encuentra entre instrucciones o bloques básicos.



Expresiones Disponibles (Available expresions)

- El problema de las expresiones disponibles consiste en determinar que expresiones están disponibles al inicio de cada bloque básico.
- Una expresión esta disponible en un punto si el resultado que se obtuvo cuando se calculo aún es el mismo que si se vuelve a calcular en el punto en que estamos.
- Ejemplo:

```
k=b+c;
r=d*e
b=5;
```

  - En este punto d*e es una expresión disponible y b+c no lo es porque se ha modificado b.

Expresiones Disponibles (Available expresions)

- Conjuntos considerados
  - Para cada bloque básico se definen 4 conjuntos
    - AE_TOP(BB): la expresión está disponible en el punto que precede a BB
    - AE_KILL(BB): la expresión ya no será válida después de la ejecución de BB por que se ha modificado algún operando.
    - AE_GEN(BB): Se ha evaluado la expresión en BB sin que se modifiquen sus operandos.
    - AE_BOT(BB): La expresión está disponible justo después de la ejecución de BB
  - Ecuación para las expresiones disponibles

    $AE\_BOT(BB)=(AE\_TOP(BB)-AE\_KILL(BB)) \cup AE\_GEN(BB)$

    $AE\_TOP(BB)= \cap_{p \text{ precedente de BB}} AE\_BOT(P)$

Expresiones Disponibles (Available expresions)

- Uso de las expresiones disponibles
  - Evitar recalcular la misma expresión (eliminación de subexpresiones redundantes)
    - Se ha calculado una expresión y tenemos guardado el resultado.
    - Si más adelante hay que repetir el cálculo no será necesario hacerlo si es una expresión disponible.

Alcance de las Definiciones (reaching definitions)

- El valor de una variable se define cuando se le asigna un valor.
- Este valor definido se pierde cuando se realiza una nueva asignación.
- El problema del alcance de las definiciones es el mismo que el problema de las expresiones disponibles, pero en el caso de las variables.
- Ecuaciones:

$$RD\_BOT(BB) = (RD\_TOP(BB) - RD\_KILL(BB)) \cup RD\_GEN(BB)$$

$$RD\_TOP(BB) = \bigcup_{p \text{ precedente de } BB} RD\_BOT(P)$$

Alcance de las Definiciones (reaching definitions)

- Usos del alcance de las definiciones
  - Reutilización de copias en registros
    - Asignamos un valor a una variable. Este se encuentra en un registro del procesador.
    - Si más adelante necesitamos el valor de la variable podemos evitar acceder a memoria y reutilizar el valor que hay en el registro si pertenece al conjunto de las definiciones.
  - Detectar el uso de variables no inicializadas
    - Se produce cuando intentamos leer el valor de una variable no definida.
  - Propagación de constantes
    - Hay que ver que una variable está definida con una constante y se podrá sustituir por esta cuanto se quiera leer

Variables Vivas (live variables)

- Una variable esta viva en un punto p cuando su valor es requerido más adelante en un un camino de ejecución que pasa por p.
- Un camino requiere el valor de una variable cuando hay una lectura de la variable antes de una asignación
- Ecuaciones

$$LV\_TOP(BB) = (LV\_BOT(BB) - LV\_DEF(BB)) \cup LV\_USE(BB)$$

$$LV\_BOT(BB) = \cup_{s \text{ sucesor de } BB} LV\_TOP(S)$$

Variables Vivas (live variables)

- Usos
  - delimitar exactamente en que partes del código es necesaria una variable.
  - Eliminación de variable muertas y asignaciones innecesarias.

Expresiones muy Utilizadas (Very Busy Expression)

- Una expresión es muy utiliza en un punto p cuando el valor de la expresión se requiere antes que el valor de cualquiera de sus términos a lo largo de cualquier camino que empieza en p.
- Ecuaciones

$$VBE\_TOP(BB)=(VBE\_BOT(BB)-BVE\_DEF(BB)) \cup BVE\_USE(BB)$$

$$VBE\_BOT(BB)= \cap_{s \text{ sucesor de } BB} VBE\_TOP(S)$$

Algoritmos de Análisis del Flujo de Ejecución

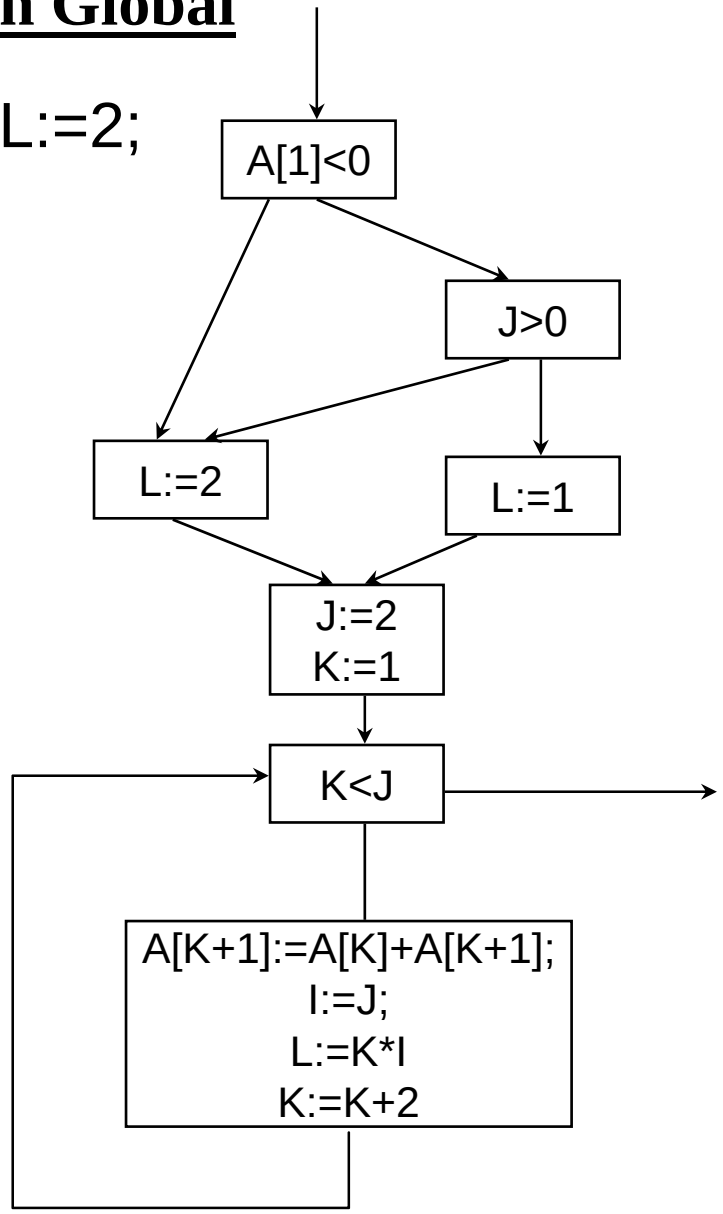
- Tipos
  - Algoritmos basados en la estructura de los bucles
    - Son rápidos
    - Hay que reducir el grafo a bucles sin saltos que entren en medio del bucle. Los programas estructurados son reducibles.
  - Algoritmos iterativos
    - Son lentos pero genéricos
    - Tipos:
      - lista de trabajos
      - round robin

Aplicaciones a la Optimización de Programas

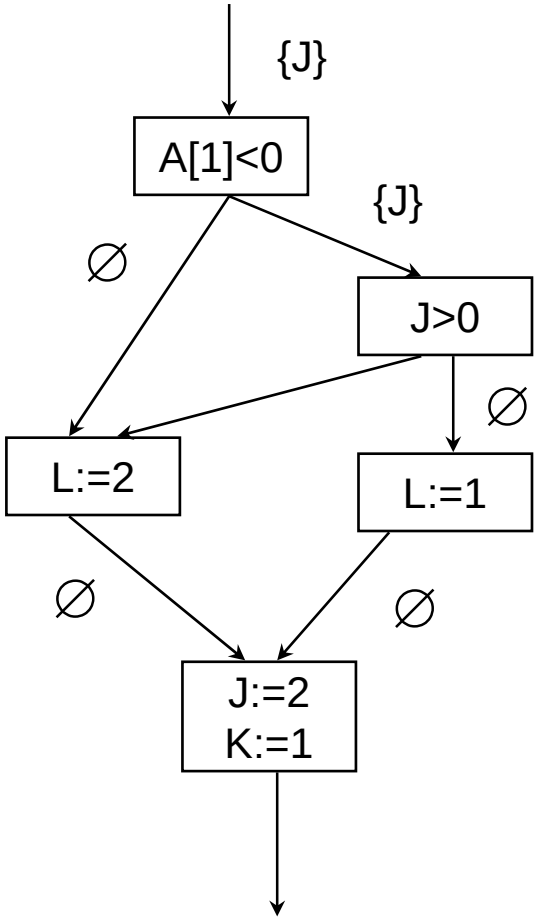
- expresiones disponibles (al seguir la ejecución sigue siendo válido el resultado obtenido)
  - Eliminar expresiones redundantes
- Alcance de las definiciones
  - Reutilizar las copias en registro de los valores de las variables.
  - Propagación de constantes
  - Reducción de frecuencia
- variables vivas
  - Reutilizar el espacio de variables.
  - Eliminar variables innecesarias
- expresiones muy utilizadas
  - optimizar la asignación de registros

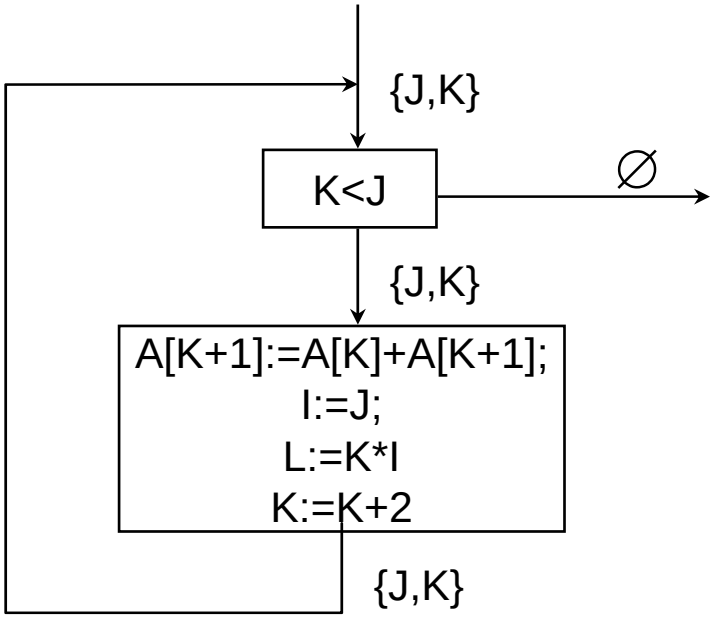
Ejemplo: Optimización Global

```
IF A[1]<0 & J>0 THEN L:=1 ELSE L:=2;
J:=2;
FOR K:=1 STEP 2 UNTIL J DO
BEGIN
   A[K+1]:=A[K]+A[K+1];
   I:=J;
   L:=K*I;
END
```

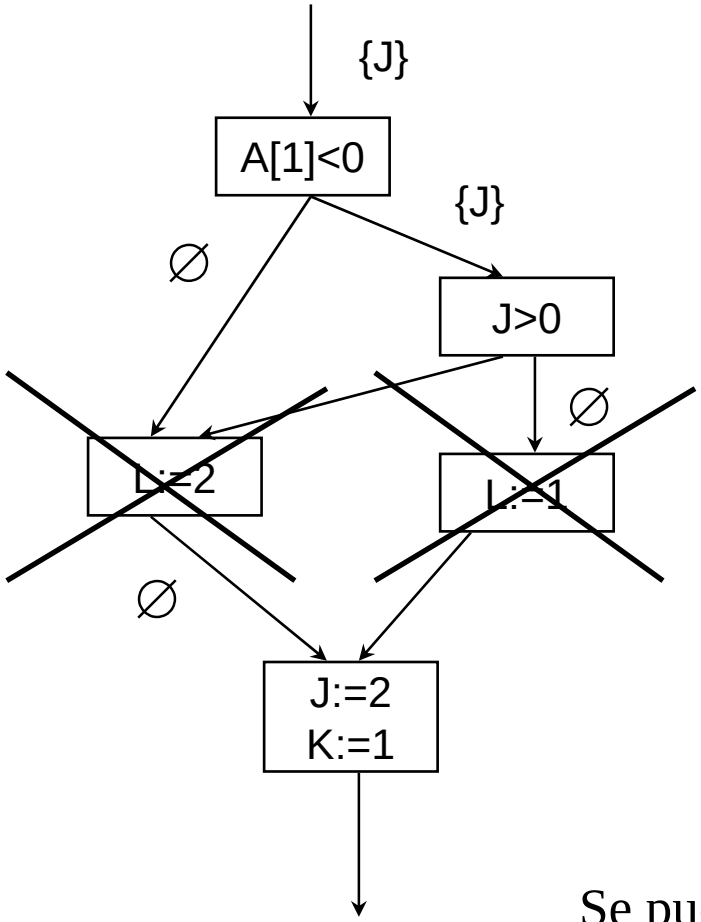


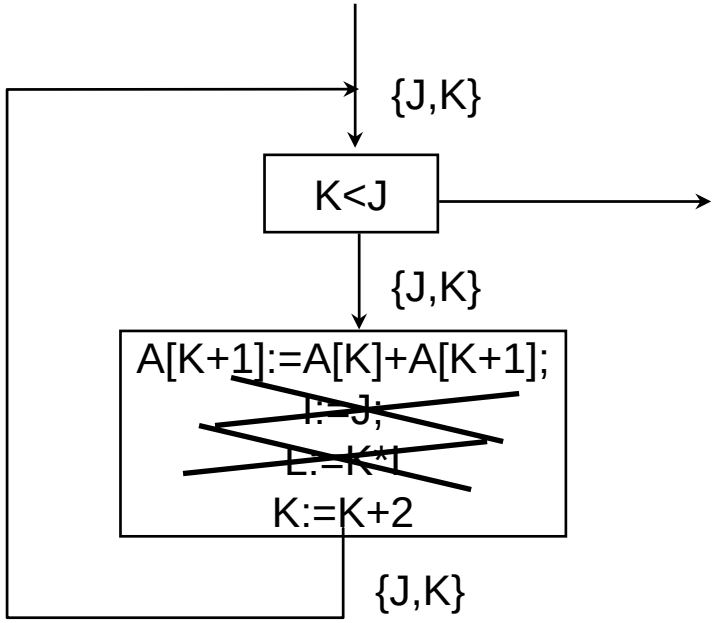
Ejemplo de Optimización Global: Variables Vivas





Ejemplo de Optimización Global: Eliminación de Variables Innecesarias

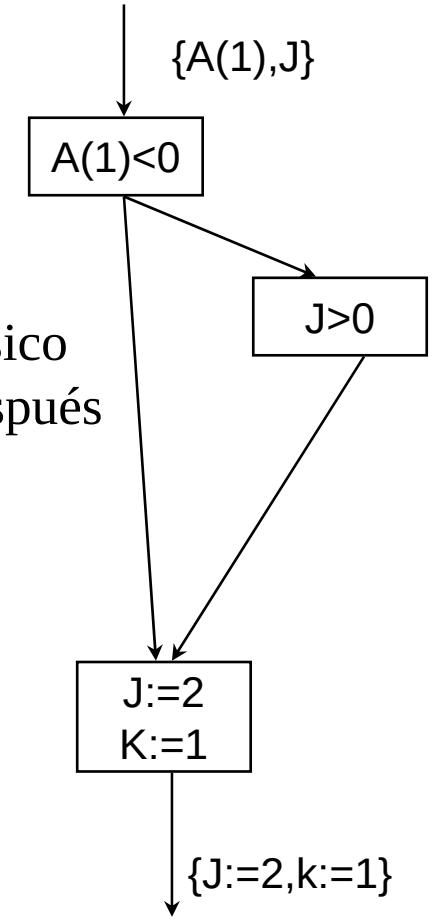


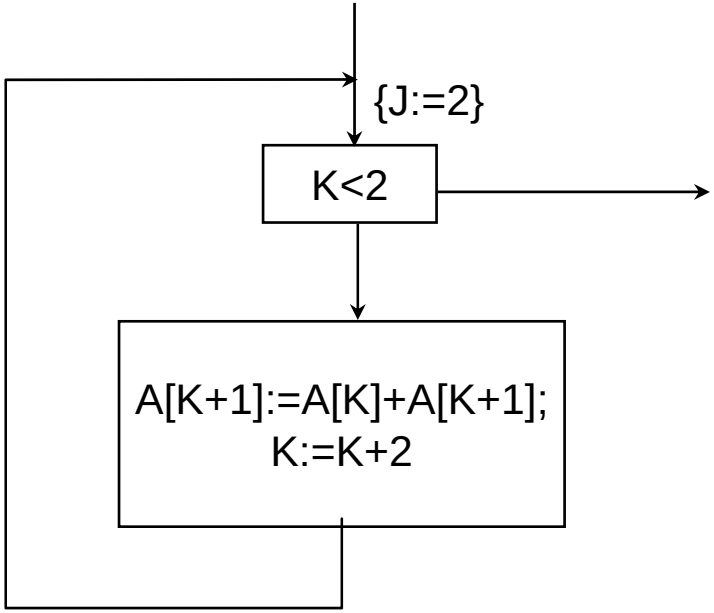


Se puede eliminar L e I
Las A[K],A[K+1] no se eliminan pues no se han podido considerar en el cálculo de variables vivas

Ejemplo de Optimización Global
Propagación de Constantes

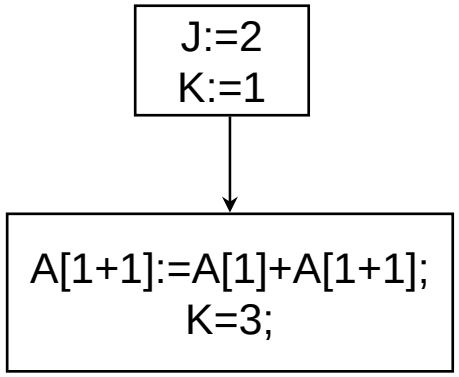
Eliminar J>0 por no utilizarse y saltar al mismo bloque básico
Eliminar A[1]<0 después de eliminar J>0





Se puede expandir el bucle

Ejemplo de Optimización Global
Expandir el Bucle



Eliminar J y k por que no se utilizan y realizar los cálculos entre constantes