Paso 1: construcción del grafo de control de flujo

Un *bloque básico* es una secuencia de instrucciones en la que se puede entrar sólo por la primera instrucción y de la que se puede salir sólo por la última instrucción.

La primera instrucción (*líder*) de un bloque básico puede ser:

1. El punto de entrada de una rutina (procedimiento o función).
2. El destino de un salto (condicional o incondicional).
3. Una instrucción que sigue inmediatamente a un salto o a un return.

Para determinar los bloques básicos que constituyen una rutina, identificamos sus líderes y, a continuación, por cada líder, incluimos en su bloque básico la secuencia de instrucciones desde él hasta la inmediatamente anterior al siguiente líder o hasta el final de la rutina en caso de que no haya otro líder posterior.

Un *grafo de control de flujo* es un grafo dirigido que contiene un nodo por cada bloque básico más otros dos nodos especiales llamados *entrada* y *salida*. Las aristas del grafo conectan los bloques básicos de la siguiente forma:

1. Desde el nodo *entrada* hay una arista dirigida al primer bloque básico.
2. Desde cualquier bloque sin sucesor hay una arista dirigida al nodo *salida*.
3. Entre los bloques básicos se realizan las conexiones correspondientes al control de flujo del programa.

Ejemplo:

Supongamos el siguiente código C:

```
unsigned int fib(unsigned int m)
{
     unsigned int f0 = 0, f1 = 1, f2, i;
     if (m <= 1) {
          return m;
     }
     else {
          for (i = 2; i <= m; i++) {
               f2 = f0 + f1;
               f0 = f1;
               f1 = f2;
          }
          return f2;
     }
}
```

El código intermedio correspondiente (notación de Muchnick) es el siguiente:

```
     receive m
     f0 = 0
     f1 = 1
```

```
      if m <= 1 goto L3
      i = 2
L1:   if i <= m goto L2
      return f2
L2:   f2 = f0 + f1
      f0 = f1
      f1 = f2
      i = i + 1
      goto L1
L3:   return m
```

La instrucción *receive m* recupera el primer argumento de la función (en el ensamblador de destino podría estar en la pila o en un registro) y lo almacena en la variable *m*. Las demás operaciones del código intermedio no requieren explicación.

El diagrama de control de flujo es el siguiente:

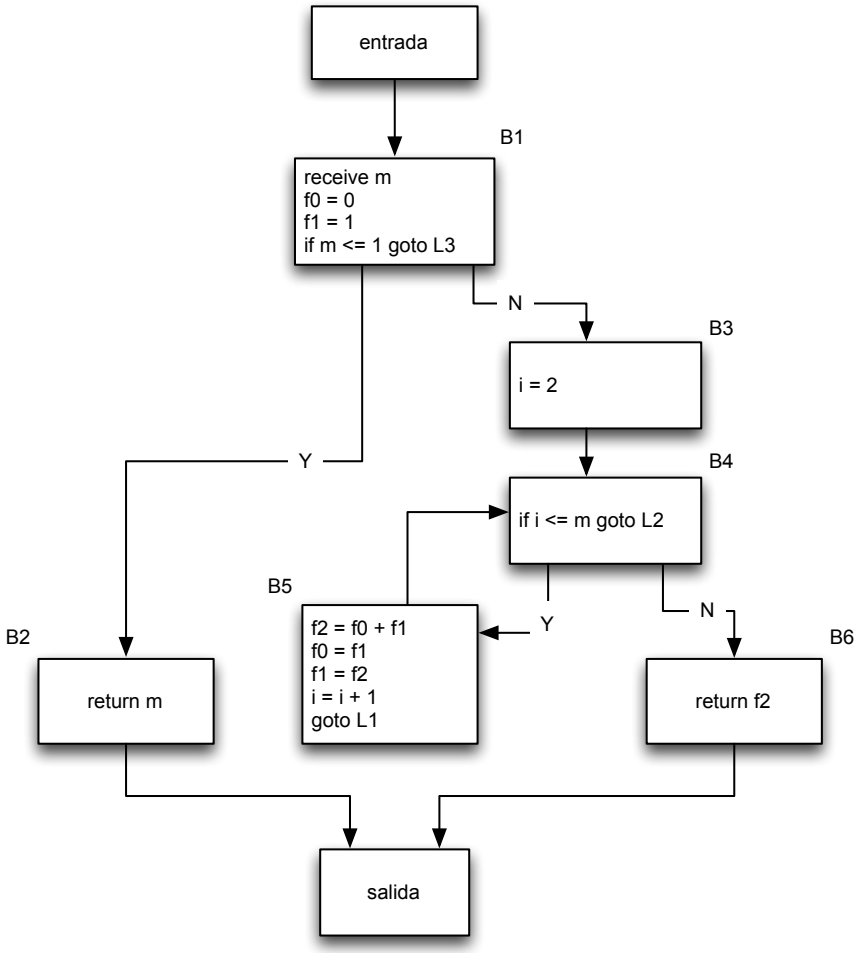

Figura 1. Grafo de control de flujo de la función Fibonacci

En el resto de los pasos usaremos algunas definiciones formales sobre el grafo de control de flujo.

Podemos definirlo como $G = < N, E >$ donde *N* es el conjunto de nodos y *E* es el conjunto de aristas, $E \subseteq N \times N$. Representaremos las aristas como $a \rightarrow b$.
Cuando hablemos de sucesor o predecesor de un nodo, nos referiremos a los siguientes conjuntos:

$$Succ(b) = \{n \in N \mid \exists e \in E \; tal \; que \; e = b \rightarrow n\}$$
$$Pred(b) = \{n \in N \mid \exists e \in E \; tal \; que \; e = n \rightarrow b\}$$

Paso 2: construcción del árbol de dominio

Decimos que un nodo *a domina* a un nodo *b* (*a dom b*) si todos los posibles caminos de ejecución desde el nodo *entrada* hasta *b* pasan por *a*. Esta relación es:

1. Reflexiva (todo nodo se domina a sí mismo *a dom a*).
2. Transitiva (si *a dom b* y *b dom c* entonces *a dom c*).
3. Antisimétrica (el único modo de que *a dom b* y *b dom a* es que *a = b*).

Podemos definir el conjunto:

$$Domin(a) = \{n \in N \mid a \; dom \; n \}$$

Usando el grafo de la Figura 1, podemos calcular:

Nodo a	Domin(a)
entrada	{entrada}
B1	{entrada,B1}
B2	{entrada,B1,B2}
B3	{entrada,B1,B3}
B4	{entrada,B1,B3,B4}
B5	{entrada,B1,B3,B4,B5}
B6	{entrada,B1,B3,B4,B6}
salida	{entrada,B1,salida}

Decimos que un nodo *a domina estrictamente* a un nodo *b*, escrito *a sdom b*, si *a dom b* y $a \neq b$.

Decimos que un nodo *a domina inmediatamente* a un nodo *b* si y sólo si *a sdom b* y no existe ningún nodo *c* tal que *a sdom c* y *c sdom b*. Es decir, no hay ningún nodo intermedio entre *a* y *b* en la relación de dominio.

La relación de dominio inmediato es única, es decir, sólo hay un nodo *a* que domine inmediatamente a otro nodo *b*. Denotamos esta relación con *a idom b* y al nodo que domina inmediatamente a *b* lo simbolizamos con *idom(b)*. Sólo el nodo *entrada* carece de dominador inmediato.

Usando el grafo de la Figura 1, podemos calcular:

Nodo *a*	*idom(a)*
entrada	Ø
B1	entrada
B2	B1
B3	B1
B4	B3
B5	B4
B6	B4
salida	B1

La relación *idom* permite formar un árbol de dominio a partir de un grafo de control de flujo.

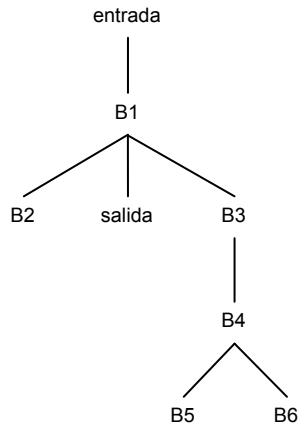

Figura 2. Árbol de dominio de la función Fibonacci

Paso 3: cálculo de la frontera de dominio

Dado un nodo *x* del grafo de control de flujo, su *frontera de dominio*, denotada con *DF(x)*, es el conjunto de nodos *y* tales que *x* domina a un predecesor inmediato de *y*, pero no domina estrictamente a *y*:

$$DF(x) = \{\, y \mid (\exists z \in Pred(y)\ tal\ que\ x\ dom\ z)\ \wedge\ x\ !\ sdom\ y \,\}$$

Si pensamos en que los bloques del grafo de control de flujo pueden contener instrucciones que dan valores a variables, la frontera de dominio de *x* nos está indicando los bloques en los que los valores asignados a variables en *x* podrían no ser relevantes porque se puede llegar a esos bloques por otros caminos de ejecución que pueden dar otros valores distintos a las mismas variables.

El cálculo de *DF(x)* se puede dividir en dos pasos: cálculo local y cálculo recursivo:

$$DF(x) = DF_{local}(x) \bigcup_{z \in N \mid idom(z) = x} DF_{rec}(x, z)$$

La definición de $DF_{local}(x)$ es la siguiente:

$$DF_{local}(x) = \{y \in Succ(x) \mid idom(y) \neq x\}$$

Es decir, forman parte de la frontera de dominio de un nodo *x* aquellos sucesores *y* de x que *x* no domine inmediatamente. Esto encaja con la definición de *DF(x)* ya que se permite que el nodo *z* que es predecesor de *y* sea el propio nodo *x*, puesto que *x dom x* siempre se cumple (propiedad reflexiva).

Por ejemplo, con el grafo de control de flujo de la Figura 1 y el árbol de dominio de la Figura 2 podemos ver que:

$$DF_{local}(entrada) = \emptyset$$
$$DF_{local}(B1) = \emptyset$$
$$DF_{local}(B2) = \{ salida \}$$
$$DF_{local}(B3) = \emptyset$$
$$DF_{local}(B4) = \emptyset$$
$$DF_{local}(B5) = \{ B4 \}$$
$$DF_{local}(B6) = \{ salida \}$$
$$DF_{local}(salida) = \emptyset$$

Para el caso de los sucesores *z* que *x* sí domina inmediatamente, tenemos que calcular recursivamente *DF(z)* ya que cualquier nodo que esté en *DF(z)* y no esté dominado inmediatamente por *x* estará también en *DF(x)*. Para ello usamos:

$$DF_{rec}(x, z) = \{ y \in DF(z) \mid idom(z) = x \ \wedge idom(y) \neq x\}$$

Para ver cómo se aplica $DF_{rec}(x, z)$ podemos hacer un recorrido en postorden del árbol de dominio:

- Como *x*=B6 no domina inmediatamente a ningún nodo, no hay ningún *z* para poder calcular $DF_{rec}(x, z)$, de modo que:

$$DF(B6) = DF_{local}(B6) = \{ salida \}$$

- Con *x*=B5 sucede algo parecido:

$$DF(B5) = DF_{local}(B5) = \{ B4 \}$$

- Con *x*=B4 sí podemos usar $DF_{rec}(B4, B5)$ y $DF_{rec}(B4, B6)$, ya que B4 domina inmediatamente a B5 y B6:

$$DF_{rec}(B4, B5) = \{ y \in DF(B5) \mid idom(y) \neq B4\} = \{B4\}$$
$$DF_{rec}(B4, B6) = \{ y \in DF(B6) \mid idom(y) \neq B4\} = \{salida\}$$

$$DF(B4) = DF_{local}(B4) \cup DF_{rec}(B4, B5) \cup DF_{rec}(B4, B6) = \{B4, salida\}$$

Es interesante ver que un nodo (B4 en este caso) puede estar en su propia frontera de dominio.

- Con x=B3 podemos usar $DF_{rec}(B3, B4)$:

$$DF_{rec}(B3, B4) = \{\, y \in DF(B4) \mid idom(y) \neq B3\} = \{salida\}$$

$$DF(B3) = DF_{local}(B3) \cup DF_{rec}(B3, B4) = \{salida\}$$

- Con *x*=salida tenemos $DF(salida) = \emptyset$.

- Como *x*=B2 no domina inmediatamente a ningún nodo, no hay ningún *z* para poder calcular $DF_{rec}(x, z)$, de modo que:

$$DF(B2) = DF_{local}(B2) = \{\, salida \,\}$$

- Con x=B1, que domina inmediatamente a B2, B3 y *salida*, podemos usar:

$$DF_{rec}(B1, B2) = \{\, y \in DF(B2) \mid idom(y) \neq B1\} = \emptyset$$
$$DF_{rec}(B1, B3) = \{\, y \in DF(B3) \mid idom(y) \neq B1\} = \emptyset$$
$$DF_{rec}(B1, salida) = \{\, y \in DF(salida) \mid idom(y) \neq B1\} = \emptyset$$

$$DF(B1) = DF_{local}(B1) \cup DF_{rec}(B1, B2) \cup DF_{rec}(B1, B3) \cup DF_{rec}(B1, salida) = \emptyset$$

La siguiente tabla resume todas las fronteras de dominio:

Nodo b	DF(b)
entrada	∅
B1	∅
B2	{salida}
B3	{salida}
B4	{B4, salida}
B5	{B4}
B6	{salida}
salida	∅

Paso 4: cálculo de la frontera de dominio iterada

Dado un conjunto de nodos *S* del grafo de control de flujo, podemos definir la frontera de dominio de dicho conjunto S como la unión de las fronteras de dominio de los nodos de S:

$$DF(S) = \bigcup_{x \in S} DF(x)$$

Por ejemplo, en la Figura 1 podemos ver el conjunto D_v formado por los bloques en los que se asignan valores a cada variable:

Variable v	D_v
f0	{B1,B5}
f1	{B1,B5}
f2	{B5}
i	{B3,B5}
m	{B1}

Tomando S={B3,B5} como ejemplo, podemos calcular:

$$DF(\{B3, B5\}) = DF(B3) \cup DF(B5) = \{B4, salida\}$$

Se denomina *frontera de dominio iterada* de un conjunto S, denotado con $DF^+(S)$, a:

$$DF^+(S) = \lim_{i \to \infty} DF^i(S)$$

donde $DF^1(S) = S$ y $DF^{i+1}(S) = DF(S \cup DF^i(S))$.

Si *S* fuese el conjunto de bloques en los que se asignan valores a una cierta variable *v* más el nodo *entrada*, entonces $DF^+(S)$ serían los nodos que pueden necesitar funciones ϕ para la variable *v*. Se tiene que usar el nodo *entrada* porque podría haber variables no definidas antes de su uso. Se considera que toman un primer valor por defecto en el nodo *entrada*. Por otra parte, la iteración de la frontera de dominio permite tener en cuenta que la inserción de una función ϕ para la variable *v* redefine el valor de dicha variable, de modo que los bloques en los que se insertan pueden aumentar la frontera de dominio.

Hacemos los cálculos para cada variable:

- Para v=f0 o v=f1 podemos ver que:

$$DF^1(\{entrada, B1, B5\}) = \{B4\}$$
$$DF^2(\{entrada, B1, B4, B5\}) = \{B4, salida\}$$
$$DF^3(\{entrada, B1, B4, B5, salida\}) = \{B4, salida\}$$

- Para v=f2 podemos calcular:

$$DF^1(\{entrada, B5\}) = \{B4\}$$
$$DF^2(\{entrada, B4, B5\}) = \{B4, salida\}$$
$$DF^3(\{entrada, B4, B5, salida\}) = \{B4, salida\}$$

- Para v=i podemos calcular:

$$DF^1(\{entrada, B3, B5\}) = \{B4, salida\}$$
$$DF^2(\{entrada, B3, B4, B5, salida\}) = \{B4, salida\}$$

- Para v=m podemos calcular:

$$DF^1(\{entrada, B1\}) = \emptyset$$

Paso 5: cálculo de los conjuntos de variables definidas y usadas

Se puede refinar el conjunto de nodos en los que tenemos que insertar las funciones ϕ para la variable *v* analizando la *vida* (liveness) de cada variable.
Como paso previo para el cálculo del liveness, tenemos que obtener dos conjuntos en cada nodo de grafo de control de flujo:

DEF(x) = { variables asignadas en x antes de algún uso de esa variable en x }
USE(x) = { variables usadas en x antes de alguna asignación a esa variable en x }

Partiendo del grafo de la Figura 1, podemos definir los siguientes conjuntos:

Bloque x	DEF(x)	USE(x)
B1	{m,f0,f1}	Ø
B2	Ø	{m}
B3	{i}	Ø
B4	Ø	{i,m}
B5	{f0,f1,f2,i}	{f0,f1,i}
B6	Ø	{f2}

Paso 6: variables vivas a la entrada y salida de un bloque

Para conocer cuáles son las variables que están vivas a la entrada y salida de cada bloque, calculamos los conjuntos IN(x) y OUT(x). El cálculo se realiza con una serie de ecuaciones que modelan el flujo de variables entre bloques.

El conjunto de variables vivas a la salida de un bloque x se define como:

$$OUT(x) = \bigcup_{y \in Succ(x)} IN(y)$$

es decir, el conjunto de las variables vivas a la entrada de los bloques sucesores de x debe coincidir con el conjunto de variables vivas a la salida de x.

Las variables que deben estar vivas a la entrada de un bloque x se definen como:

$$IN(x) = USE(x) \cup (OUT(x) - DEF(x))$$

Por un lado, las variables que se usan en un bloque x (antes de ninguna asignación en x) deben estar vivas a la entrada del bloque. Además, se deben añadir aquellas variables que necesiten otros bloques sucesores (estarán en OUT(x) previamente

calculado) menos las que han sido asignadas en el propio bloque x. Se dice que una asignación de una variable *v* en x "mata" la variable. En realidad, mata la variable *v* en la "versión" que llega a x desde un bloque predecesor, pero crea una nueva versión para ella que puede ser usada por bloques sucesores, tomando como punto de partida de esa nueva versión el bloque x.

Los conjuntos IN(x) para todos los bloques, excepto *entrada*, se inicializan con Ø (también IN(*salida*)= Ø). Las ecuaciones anteriores se aplican iterativamente hasta que no cambia ningún conjunto.

Podemos ver la evolución de los valores de IN(x) y OUT(x) del grafo de la Figura 1:

Iteración	B1	B2	B3	B4	B5	B6
0	IN= Ø	IN=Ø	IN=Ø	IN=Ø	IN=Ø	IN=Ø
1	OUT=Ø IN=Ø	OUT=Ø IN={m}	OUT=Ø IN=Ø	OUT=Ø IN={i,m}	OUT={i,m} IN={f0,f1,i,m}	OUT=Ø IN={f2}
2	OUT={m} IN=Ø	OUT=Ø IN={m}	OUT={i,m} IN={m}	OUT={f0,f1,f2,i,m} IN={f0,f1,f2,i,m}	OUT={f0,f1,f2,i,m} IN={f0,f1,i,m}	OUT=Ø IN={f2}
3	OUT={m} IN=Ø	OUT=Ø IN={m}	OUT={f0,f1,f2,i,m} IN={f0,f1,f2,m}	OUT={f0,f1,f2,i,m} IN={f0,f1,f2,i,m}	OUT={f0,f1,f2,i,m} IN={f0,f1,i,m}	OUT=Ø IN={f2}
4	OUT={f0,f1,f2,m} IN={f2}	OUT=Ø IN={m}	OUT={f0,f1,f2,i,m} IN={f0,f1,f2,m}	OUT={f0,f1,f2,i,m} IN={f0,f1,f2,i,m}	OUT={f0,f1,f2,i,m} IN={f0,f1,i,m}	OUT=Ø IN={f2}

Paso 7: inserción de las funciones ϕ

En el paso 6 hemos obtenido los bloques en los que es necesario insertar funciones ϕ para cada variable:

Variable v	Bloques en los que insertar funciones ϕ
f0	{B4, salida}
f1	{B4, salida}
f2	{B4, salida}
i	{B4, salida}
m	Ø

Podemos comprobar que IN(B4) = {f0,f1,f2,i,m}, pero IN(*salida*) = Ø. Puesto que ninguna variable está viva al llegar a *salida*, no es necesario insertar funciones ϕ en este bloque. Sí que es necesario hacerlo al comienzo de B4 para cada variable de IN(B4).

¿Cuántos argumentos debe tener cada función ϕ? Por ejemplo, dada la función ϕ para f0, hay que observar que B4 es sucesor de dos nodos en el grafo de control de flujo (B3 y B5) que tienen en sus conjuntos OUT la variable f0. Por esa razón, la función ϕ para f0 tiene dos argumentos: una variable f0 correspondiente a B3 y otra a B5. Sucede algo similar para las demás funciones ϕ de B4.

Denotamos que una variable *v* usada en una función ϕ procede de un bloque b con *v*.b.

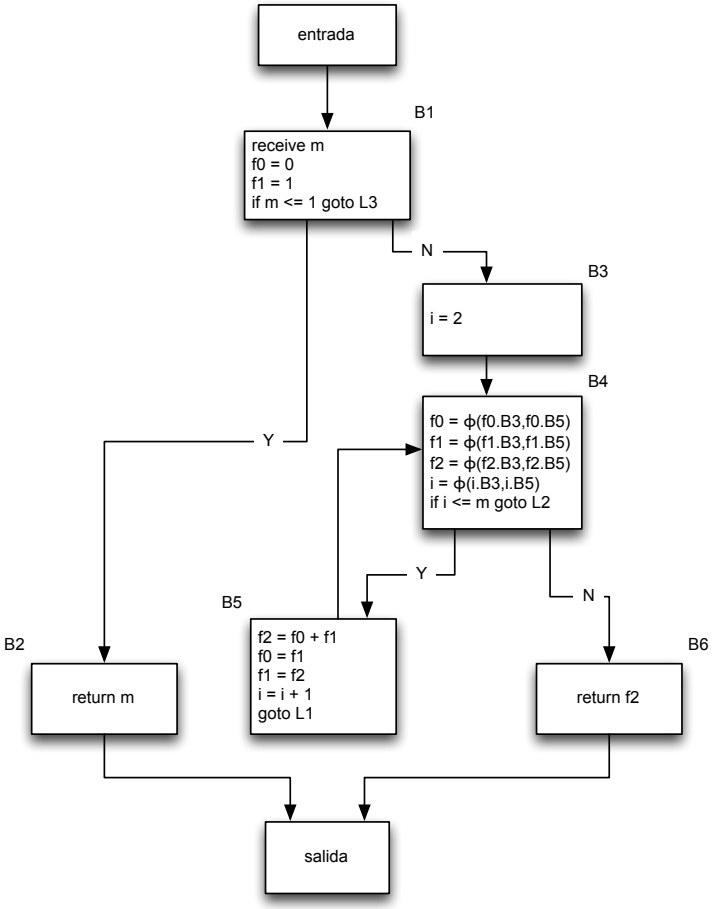

Figura 3. Grafo de control de flujo con funciones ϕ insertadas

En la siguiente sección se explica cómo se obtiene la versión adecuada de cada variable, incluyendo el caso de las instrucciones que usan ϕ.

Paso 8: renombrado de variables

Para obtener la forma SSA del código intermedio, es necesario realizar un renombrado de variables. Este proceso asocia a cada rango de vida de una variable un nombre nuevo, también llamado versión.

Podemos denotar las distintas versiones de una variable *v* con *v_n*, donde *n* es un entero único. Gracias a las propiedades de dominio, se puede realizar este proceso haciendo un recorrido **primero en profundidad** del árbol de dominio.

Durante el recorrido, es necesario recordar en cada punto *p* del grafo la versión correspondiente a la única definición (asignación) de la variable que alcanza dicho punto. Esta definición es la más cercana que domina *p*.

Sea *v.*reachingDef la versión "actual" de la variable *v* en cierto punto *p* del grafo de control de flujo. Esta versión se actualiza con el algoritmo de renombrado.

Sea *v*.maxVersion la versión más alta de la variable *v* creada hasta el momento. Este valor se emplea y actualiza cuando hay que crear una versión nueva.

Sea b[v] el último valor tomado por v.reachingDef en el nodo b. Esta es una información auxiliar que puede convenir para acelerar la búsqueda de la versión actual de una variable.

Consideramos que inicialmente *v*.reachingDef = *v*.maxVersion = b[v] = 0 para todas las variables. Si en cierto punto *p* el valor de *v*.reachingDef es 0 significa que no ha sido definida hasta llegar a *p*.

El algoritmo de renombrado se apoya en el método actualizaReachingDef(v,b). Este método actualiza v.reachingDef en el bloque b.

actualizaReachingDef(v,b):

- Si b[v] != 0:
    - v.reachingDef = b[v]
- Si b[v] == 0:
    - Asciende en el árbol de dominio hasta el primer b' | b'[v] != 0 o b' == entrada
    - v.reachingDef = b'[v]

El algoritmo de renombrado es el siguiente:

- Por cada nodo b en preorden primero en profundidad del árbol de dominio:
    - Por cada instrucción i en la secuencia de b:
        - Si i **no** es una función ϕ:
            - Por cada variable **v usada** en i:
                - Invocar a actualizaReachingDef(v,b)
                - Reemplaza v en i por v.reachingDef
                - b[v] = v.reachingDef
        - Por cada variable **v definida** en i (puede ser con una función ϕ):
            - v.maxVersion += 1
            - v.reachingDef = v.maxVersion
            - Reemplaza v por v.reachingDef
            - b[v] = v.reachingDef
    - Por cada función ϕ en un bloque sucesor de b que use una variable *v.b*
        - Invocar a actualizaReachingDef(v,b)
        - Reemplaza *v*.b en función ϕ por v.reachingDef

El algoritmo, aplicado a los bloques del grafo de la Figura 3, funcionaría recorriendo el árbol de la Figura 2 del siguiente modo:

- Bloque B1.

1. Instrucción **receive m**:

Se considera que esta instrucción define la variable m. Es la primera versión de esta variable, de modo que se reemplaza por **receive m_1**.
B1[m] = m_1

2. Instrucción **f0 = 0**:
   Se define f0, reemplazándose por f0_1 = 0
   B1[f0] = f0_1

3. Instrucción **f1 = 1**:
   Se define f1, reemplazándose por f1_1 = 1
   B1[f1] = f1_1

4. Instrucción **if m<1 goto L3**:
   La variable m se emplea en esta instrucción. Como B1[m] == m_1, se reemplaza por if m_1 < 1 goto L3

5. Ninguno de los bloques sucesores de B1 tiene funciones ϕ, de modo que esto finaliza el tratamiento de este bloque.

- Bloque B2

1. Instrucción **return m**:
   Se trata de un uso de la variable m. Usando el método actualizaReachingDef(m,B2) se encuentra que la primera definición que encontramos en el árbol de dominio es B1[m] == m_1, de modo que se la instrucción se sustituye por return m_1

2. El bloque sucesor de B2 es *salida*, que no tiene funciones ϕ.

- Bloque B3

1. Instrucción **i=2**:
   Se define por primera vez la variable i, de modo que se reemplaza por i_1 = 2.
   B3[i] = i_1

3. El bloque sucesor de B3 es B4, que tiene funciones ϕ.
   a. La función f0 = ϕ (f0.B3,f0.B5) se sustituye por f0 = ϕ (f0_1,f0.B5), ya que actualizaReachingDef(f0, B4) encuentra en B1[f0] la primera definición de la variable.
   b. La función f1 = ϕ (f1.B3,f1.B5) se sustituye por f1 = ϕ (f1_1,f1.B5), ya que actualizaReachingDef(f1,B4) encuentra en B1[f1] la primera definición de la variable.
   c. La función f2 = ϕ (f2.B3,f2.B5) se sustituye por f2 = ϕ (f2_0,f2.B5), ya que actualizaReachingDef(f2,B4) llega hasta *entrada* sin encontrar una definición de f2.

d. La función i = ϕ (i.B3,i.B5) se sustituye por i = ϕ (i_1,f2.B5), ya que actualizaReachingDef(i,B4) encuentra en B3[i] la primera definición de la variable.

- Bloque B4

1. Para las cuatro primeras instrucciones del bloque:
    a. Se define f0, reemplazándose por f0_2 = …
    B4[f0] = f0_2
    b. Se define f1, reemplazándose por f1_2 = …
    B4[f1] = f1_2
    c. Se define f2, reemplazándose por f2_1 = …
    B4[f2] = f2_1
    d. Se define i, reemplazándose por i_2 = …
    B4[i] = i_2

2. Instrucción **if i<m goto L2**:
    a. Se usan las variables i y m. En el caso de la variable i, la actualización más cercana es la de la instrucción inmediatamente anterior en el mismo bloque (i_2). En el caso de m, la actualización más cercana es m_1 en B1. Por tanto, la instrucción se reescribe como:
    if i_2 < m_1 goto L2

3. Ninguno de los sucesores de B4 tiene funciones ϕ.

- Bloque B5

1. Instrucción **f2 = f0+f1**:
    a. Las variables usadas f0 y f1 tienen sus definiciones más cercanas en B4, de modo que se cambian por f0_2 y f1_2.
    b. Se define f2, reemplazándose por f2_2. La instrucción queda reescrita del siguiente modo:
    f2_2 = f0_2 + f1_2
    B5[f2] = f2_2

2. Instrucción **f0 = f1**:
    a. La variable usada f1 tiene su definición más cercana en B4, de modo que se cambia por f1_2.
    b. Se define f0, reemplazándose por f0_3. La instrucción queda reescrita del siguiente modo:
    f0_3 = f1_2
    B5[f0] = f0_3

3. Instrucción **f1 = f2**:
    a. La variable usada f2 tiene su definición más cercana en el propio bloque B5, cambiándose por f2_2.

b. Se define f1, reemplazándose por f1_3. La instrucción queda reescrita del siguiente modo:
   f1_3 = f2_2
   B5[f1] = f1_3

4. Instrucción **i = i + 1**:
   a. La variable usada i tiene su definición más cercana en B4, reemplazándose por i_2
   b. Se define i, reemplazándose por i_3. La instrucción queda reescrita del siguiente modo:
      i_3 = i_2
      B5[i] = i_3

5. El nodo sucesor de B5 es B4, y tiene funciones ϕ en las que se usan variables con valores que llegan de B5:
   a. La función f0_2 = ϕ (f0_1,f0.B5) se sustituye por f0_2 = ϕ (f0_1,f0_3), ya que actualizaReachingDef(f0,B5) encuentra en B5[f0] la definición más cercana de f0.
   b. La función f1_2 = ϕ (f1_1,f1.B5) se sustituye por f1_2 = ϕ (f1_1,f1_3), ya que que actualizaReachingDef(f1,B5) encuentra en B5[f1] la definición más cercana de f1.
   c. La función f2_1 = ϕ (f2_0,f2.B5) se sustituye por f2_1 = ϕ (f2_0,f2_2), ya que actualizaReachingDef(f2,B5) encuentra en B5[f2] la definición más cercana de f2.
   d. La función i_2 = ϕ (i_1,i.B5) se sustituye por i_2 = ϕ (i_1,i_3), ya que actualizaReachingDef(i,B5) encuentra en B5[i] la definición más cercana de i.

- Bloque B6

1. Instrucción **return f2**:
   a. La variable f2 tiene su definición más cercana en B4, de modo que la instrucción se reescribe como return f2_1

El resultado completo del grafo de control de flujo en formato SSA se puede encontrar en la Figura 4.

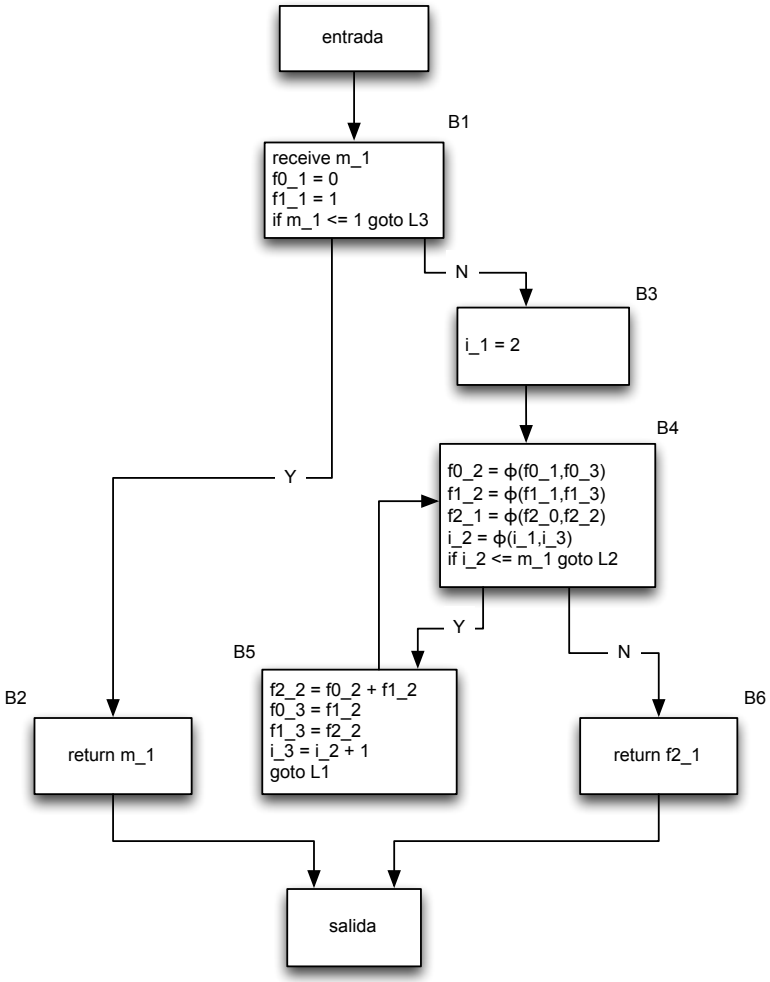

Figura 4. Grafo de control de flujo de la función Fibonacci en formato SSA