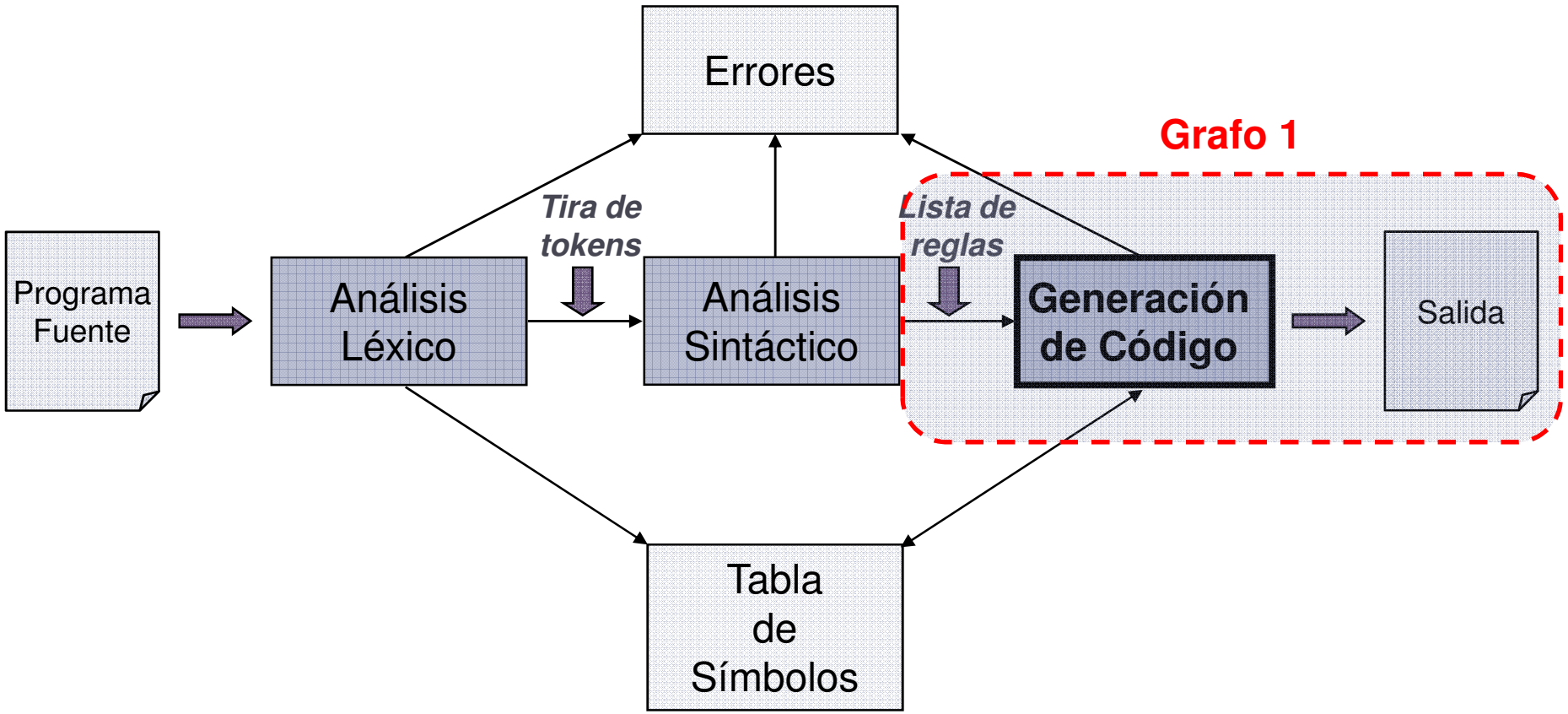
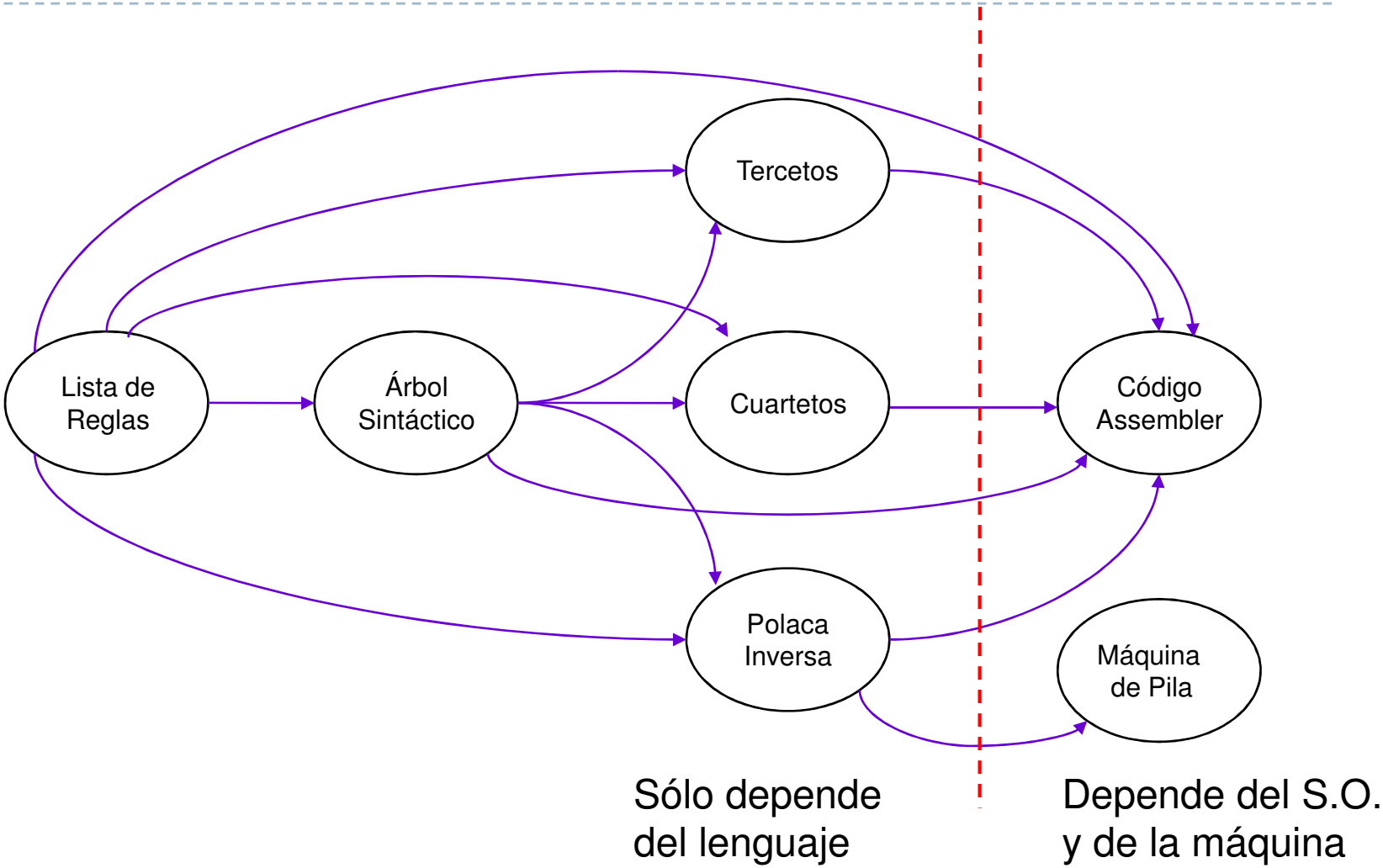
Diseño de Compiladores I

Optimizaciones

Fases de la Compilación



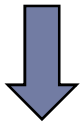
Generación de Código



Comparación de caminos

Compilador rápido (el más rápido)

Ejecutable grande y lento

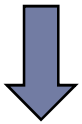


Lista de Reglas → Assembler

Comparación de caminos

Ejecutable rápido

Compilador lento



L.Reglas → Á.Sintáctico → Tercetos → Assembler

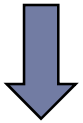
(Optimizaciones)

Comparación de caminos

Desarrollo Rápido

Compilador relativamente rápido

Ejecutable relativamente bueno



Lista de Reglas → Polaca Inversa → Assembler
(Algunas Optimizaciones)

Optimización

Objetivo:

Lograr que el código sea lo mejor posible

Resultados:

Nunca se obtiene código óptimo

Optimización

Propiedades:

- Una transformación debe preservar la semántica del programa.
- Debe acelerar el programa en una cantidad medible.
- Debe valer la pena.

Optimización

Causas:

- El programador comete errores de programación.

  Ejemplo:

  a:=2*3*b en lugar de a:=6*b

- El programador prefiere legibilidad.

  Ejemplo:

```
#define LARGO 20
#define ANCHO 10
…
volumen := LARGO * ANCHO * alto;
```

- El programador no puede evitarlo.

Redundancia Simple
(subexpresiones comunes)

- Una expresión E es una subexpresión común si E ha sido calculada previamente, y los valores de las variables usadas en E no han cambiado desde dicho cálculo.
- Se puede evitar recalcular la expresión utilizando el valor calculado previamente (se eliminan operaciones duplicadas)
- Se tienen en cuenta propiedades de conmutatividad y asociatividad.
- Ejemplo:

a := b * c * d + b * c * e

a := b * c * (d + e)

Redundancia Simple

- Se puede aplicar en:
  - **Tercetos**:
    - Buscar tercetos repetidos hasta la asignación
    - Cambiar la referencia en el o los tercetos que referencian al terceto redundante.
    - Eliminar la/s repetición/es del terceto redundante.
  - **Cuartetos**
    - Solución similar a Tercetos
  - **Polaca Inversa**
    - Insertar operadores especiales: DUP y SWAP
  - **Árbol Sintáctico**
    - Detectar subárboles iguales
    - Muy costoso → ***No se hace***

Reducción Simple

- Se evalúan expresiones que son conocidas en *tiempo de compilación*, y se sustituyen por su valor.
  - Operaciones entre constantes.
  - Otras operaciones. ¿Cuáles?

Reducción Simple

- Se puede aplicar en:
  - **Árbol Sintáctico**
    - Durante la construcción.
    - A la salida (hacia otra representación). (*)
    - Sobre el árbol terminado (2 pasadas).
  - **Polaca Inversa**
    - Durante la construcción. Si se está generando PI a partir del Árbol Sintáctico, se aplicará el algoritmo (*).
    - A la salida (en la traducción a Assembler).
    - Sobre la Polaca terminada (2 pasadas).
  - **Tercetos**:
    - Durante la construcción. Si se están generando Tercetos a partir del Árbol Sintáctico, se aplicará el algoritmo (*).
    - A la salida (en la traducción a Assembler).
    - Sobre los Tercetos terminados (2 pasadas).

Reordenamiento de código

- Intenta minimizar el uso de registros.
- Aún habiendo registros suficientes, reduce la cantidad de operaciones.
- Sólo se puede aplicar sobre Árbol Sintáctico.

Otras optimizaciones

- Reducción no simple
  - 2 * a * 3
    - Se tienen en cuenta las reglas de conmutatividad y asociatividad
    - Se puede aplicar reordenamiento de árboles

- Redundancia no simple
  - z = a * b + 7 + b * a
    - Se puede tratar la expresión como lista de listas y ordenar alfabéticamente.
    - Se puede tener en cuenta la conmutatividad, al buscar las subexpresiones redundantes.

Otras optimizaciones

- Propagación de copias de constantes
  - a := 2
  - b := a * 3 $\rightarrow$ b := 6

  Para saber si el valor es conocido y está vigente:

  Tabla de Símbolos

Variable	Valor	Vigente
a	2	True

Otras optimizaciones

- Reducción de esfuerzo

Más costoso	Más económico
a^2	$a \times a$
$2 \times a$	$a + a$
$a \div 2$	$a \times 0.5$

Otras optimizaciones

- Código muerto (IF, WHILE, Asignación)

Ejemplo 1: Asignaciones

```
a := b * c + d * e * exp ( m * n );
…
a := 6;
```

⟸ Se elimina

con Tabla de Símbolos

Ejemplo 2: IF

```
#define LARGO 20
#define ANCHO 10
IF ( LARGO > ANCHO )
        …
ELSE
        …
```

⟸ Se elimina

⟸ Se elimina

Optimizaciones
Aspectos a considerar

- Tiempo de Aplicación
- Orden de Aplicación
  - Problemas de fase

    Ejemplo: (Propagación de Ctes → Código muerto → Propagación de Ctes)

    ```
    x = 1
    …
    y = 0;
    …
    if (y) x = 0;
    …
    if (x) y =1;
    ```
  - Aplicación iterativa
- Ámbito de Aplicación

Optimizaciones
Ámbito de aplicación

- Optimizaciones Locales
  - Segmentos lineales de código
    - Asignaciones
    - Bloques básicos (sin bifurcaciones)

      Redundancia Simple: se puede extender dentro del mismo bloque siempre que no haya modificación de los elementos que participan en la subexpresión común.

      Ejemplo:

      ```
      a := b * c + 7;
      …
      x := b * c – 2;
      ```

      Mecanismo: Tabla de Símbolos
- Optimizaciones Globales
  - Dentro de un procedimiento
- Optimizaciones Interprocedurales
  - Todo el programa

Optimizaciones globales

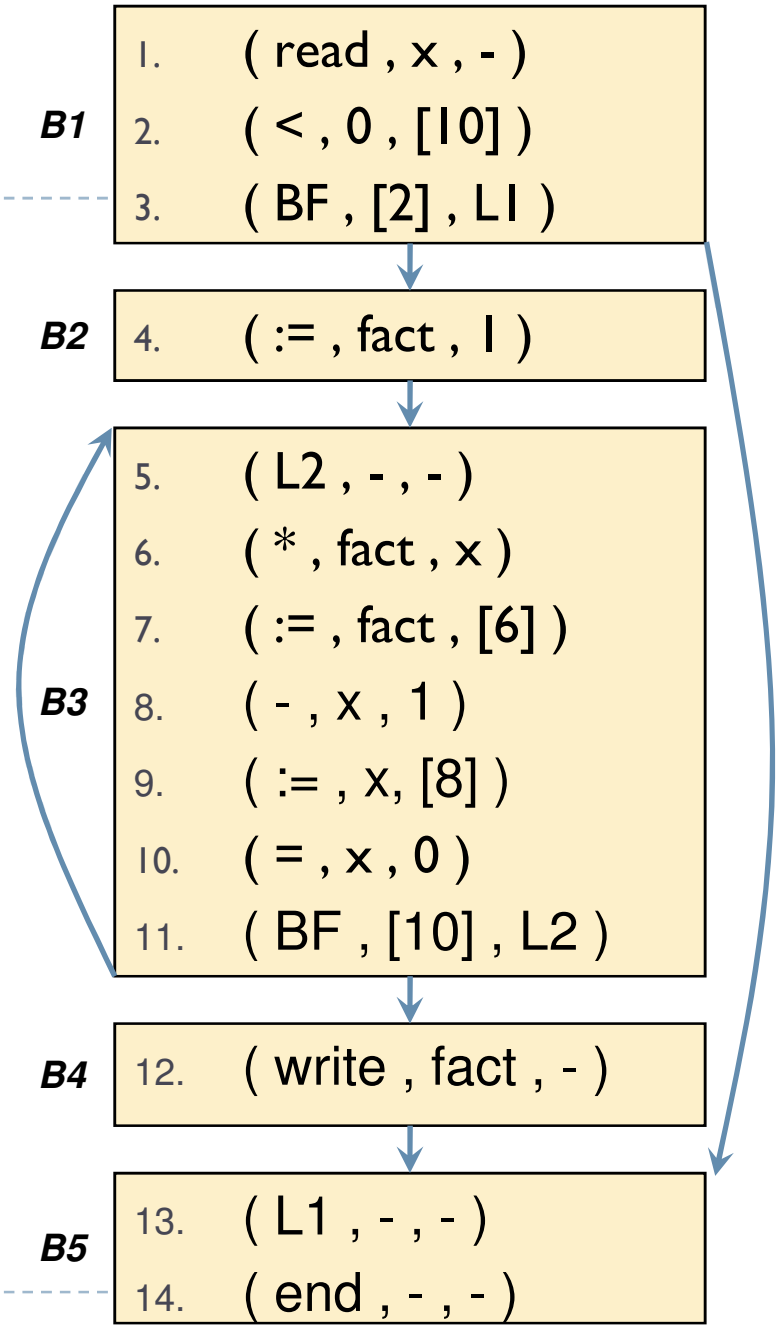
- Análisis de flujo de datos
  - El árbol sintáctico resulta inadecuado
  - Grafo de flujo
    - Nodos: bloques básicos
      - La primera instrucción comienza un bloque básico
      - Cada etiqueta destino de una bifurcación comienza un nuevo bloque básico
      - Cada instrucción posterior a una bifurcación comienza un nuevo bloque básico
    - Arcos: Bifurcaciones

Optimizaciones globales

Grafo de flujo

```
read(x);
If (0<x) then
{
        fact := 1;
        repeat
        {
                fact := fact * x;
                x := x -1;
        }
        until (x = 0);
        write(fact);
}
end;
```

```
1.   ( read , x , - )
2.   ( < , 0 , [10] )
3.   ( BF , [2] , L1 )
4.   ( := , fact , 1 )
5.   ( L2 , - , - )
6.   ( * , fact , x )
7.   ( := , fact , [6] )
8.   ( - , x , 1 )
9.   ( := , x, [8] )
10.  ( = , x , 0 )
11.  ( BF , [10] , L2 )
12.  ( write , fact , - )
13.  ( L1 , - , - )
14.  ( end , - , - )
```

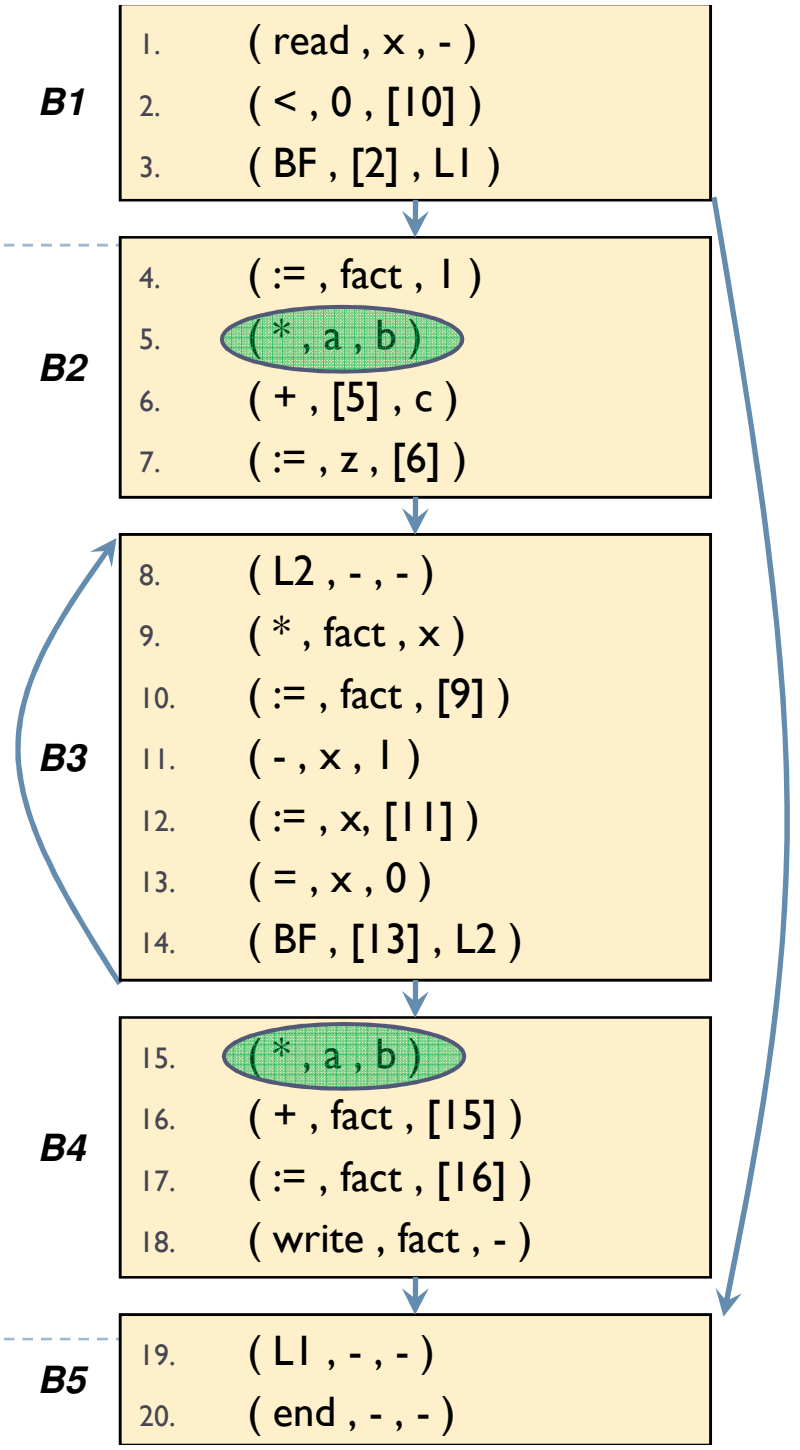


Optimizaciones globales

Grafo de flujo

```
read(x);
If (0<x) then
{
        fact := 1;
        z := a * b + c;
        repeat
        {
                fact := fact * x;
                x := x -1;
        }
        until (x = 0);
        fact := fact + a * b;
        write(fact);
}
end;
```

```
1.   ( read , x , - )
2.   ( < , 0 , [10] )
3.   ( BF , [2] , L1 )
4.   ( := , fact , 1 )
5.   ( * , a , b )
6.   ( + , [5] , c )
7.   ( := , z , [6] )
8.   ( L2 , - , - )
9.   ( * , fact , x )
10.  ( := , fact , [9] )
11.  ( - , x , 1 )
12.  ( := , x, [11] )
13.  ( = , x , 0 )
14.  ( BF , [13] , L2 )
15.  ( * , a , b )
16.  ( + , fact , [15] )
17.  ( := , fact , [16] )
18.  ( write , fact , - )
19.  ( L1 , - , - )
20.  ( end , - , - )
```



Optimizaciones interprocedurales

- Consideración de diferentes mecanismos de pasaje de parámetros.
- Acceso a variables no locales.
- Cálculo de información simultánea en todos los procedimientos que pueden llamarse entre sí.
- Compilaciones separadas
  - Optimización → Linker
- Muy costoso

Ejercicio

- Considerar las siguientes optimizaciones:
  - Reducción Simple,
  - Redundancia Simple,
  - Reordenamiento de código, y
  - Eliminación de código muerto en selecciones IF THEN ELSE.
- Si se utiliza el siguiente camino para la GC:

  Lista de Reglas → Tercetos → Assembler
- ¿Cuál sería la forma más conveniente de organizar las 4 optimizaciones? ¿En qué momento de la GC se aplicaría cada una?
- Tener en cuenta que, si resultara conveniente, se podría aplicar más de una optimización en forma simultánea.

¿PREGUNTAS?