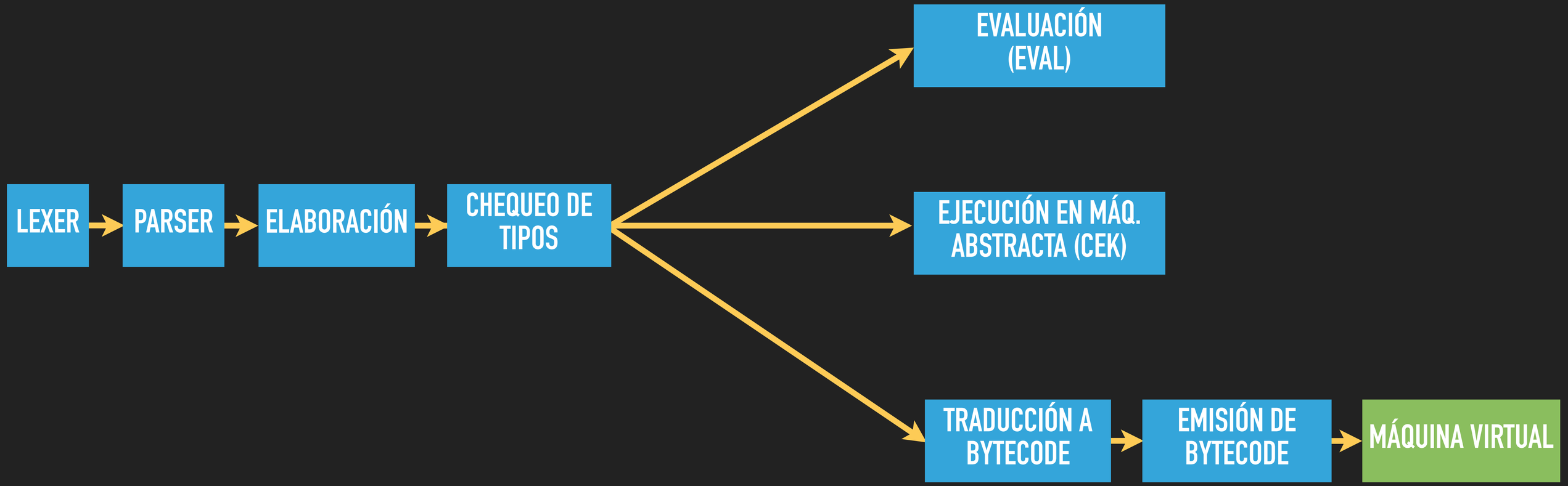
COMPILADORES

OPTIMIZACIONES

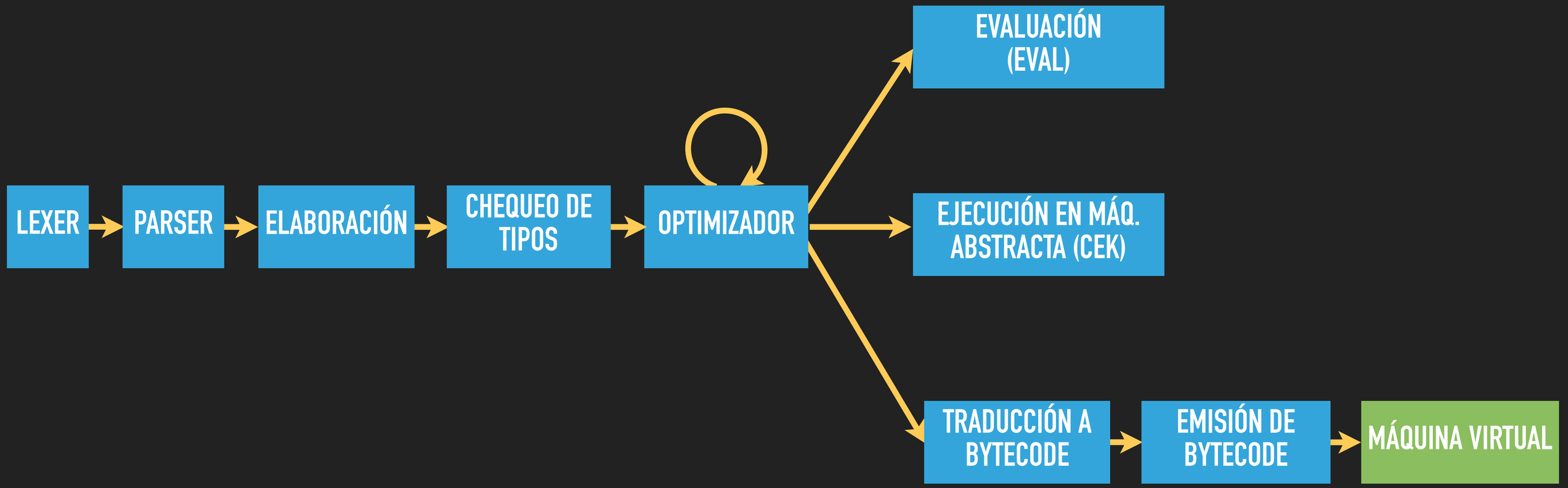
OPTIMIZACIONES EN UN COMPILADOR

- En general, es posible optimizar en todas las etapas del compilador
- Vamos a ver algunos optimizaciones clásicas que se podrían aplicar a nuestro compilador
- Estas se aplican en las etapas iniciales (después del chequeo de tipos)

PIPELINE(S)


LEXER
PARSER
ELABORACIÓN
CHEQUEO DE TIPOS
EVALUACIÓN (EVAL)
EJECUCIÓN EN MÁQ. ABSTRACTA (CEK)
TRADUCCIÓN A BYTECODE
EMISIÓN DE BYTECODE
MÁQUINA VIRTUAL

PIPELINE(S) CON OPTIMIZADOR



OPTIMIZADOR

- Trabaja sobre términos Core.
- Una optimización puede habilitar a que se aplique otra.
- Hace varias pasadas, posiblemente aplicando varias optimizaciones en cada pasada.
- Termina cuando:
  - no hay más optimizaciones para aplicar, o bien
  - se llega a un límite de pasadas.

OPTIMIZACIONES

- Hay muchas optimizaciones comunes.
  - Dead-code elimination
  - Constant folding and propagation
  - Inline expansion
  - Common subexpression elimination

ELIMINACIÓN DE CÓDIGO MUERTO (DEAD CODE ELIMINATION)

- Se trata de eliminar código que no se va a ejecutar.
  - Por ejemplo una función que nunca se llama.
- Reduce el tamaño del código resultante.
- En general, el código queda muerto por otras optimizaciones.
- Puede habilitar otras optimizaciones.

EXPANSIÓN EN LÍNEA (INLINE EXPANSION)

- Una técnica común de optimización es la expansión en línea.
- Se reemplaza una variable por su definición.
- En el caso de funciones hay que tener cuidado de no capturar variables.
- Si todos los llamados a una función son expandidos, la función no es necesaria y se puede eliminar (dead code elimination).

ALGORITMO DE EXPANSIÓN EN LÍNEA DE FUNCIONES

$$\text{let } f\ x = A$$

- Si los argumentos son simples variables
  - la expresión $f\ v$ se reescribe a $A[x \mapsto v]$.
- Si los argumentos son más complejos
  - la expresión $f\ E$ se reescribe a $\text{let } z = E \text{ in } A[x \mapsto z]$ con z variable fresca.
- Si la función toma varios argumentos se procede de forma análoga.

EXPANSIÓN DE FUNCIONES RECURSIVAS

$$\text{letrec map } f\ xs = \text{if null } xs \text{ then } [] \text{ else } f\ (\text{head } xs) : \text{map } f\ (\text{tail xs})$$

- Consideremos la expansión map en la expresión $\text{let } g\ xs = \text{map } (+1)\ xs$

$$\text{let } g\ xs = \text{if null } xs \text{ then } [] \text{ else head } xs + 1 : \text{map } (+1)\ (\text{tail xs})$$

- Esto es correcto pero sólo expande la primera iteración y luego llama a map.

EXPANSIÓN DE FUNCIONES RECURSIVAS

- Para expandir correctamente funciones recursivas vamos a seguir los siguientes pasos:
  - Identificar los argumentos invariantes de la función
  - Dividir la función en un preludio que calcula los argumentos invariantes y luego llama a una función recursiva
  - Aplicar la expansión en línea.

EXPANSIÓN DE FUNCIONES RECURSIVAS

- Dividimos las funciones recursivas en un preludio llamado desde el exterior y una función recursiva interior:

```
let map f xs =
    letrec map' xs' = if null xs'      then []
                                       else f (head xs') : map' (tail xs')
    in map' xs
```

- Notar que sacamos el argumento invariante f de la función recursiva.
- Ahora al expandir map en la expresión let $g\ xs = \text{map}\ (+1)\ xs$ obtenemos

```
let g xs =
    letrec map' xs' = if null xs'      then []
                                       else (head xs' + 1) : map' (tail xs')
    in map' xs
```

CRITERIOS PARA APLICAR EXPANSIÓN EN LÍNEA

- Si expandimos demasiado podemos tener una explosión de código
  - Tener en cuenta que luego de una expansión pueden surgir más oportunidades de expansión
- Heurísticas comunes para decidir qué expandir son:
  - Expandir llamadas a funciones que son ejecutadas frecuentemente
  - Expandir funciones con cuerpos chicos
  - Expandir funciones que se llaman una única vez (la función puede ser eliminada)

CALCULANDO EXPRESIONES CONSTANTES (CONSTANT FOLDING)

- Si tenemos una expresión constante como $2 + 3$, podemos calcular la constante y reemplazar la expresión por 5.
- Si una condición es constante, podemos eliminarla y dejar sólo la rama correspondiente
- Podemos utilizar álgebra y simplificar $x + 0$ a x, $x * 1$ a x, $x * 0$ a 0, etc, siempre y cuando no se pierdan efectos.
  - ¿Es correcto reemplazar `(print 3) * 0` por `0`?
  - En nuestro lenguaje los efectos son la impresión en pantalla y la divergencia.
- Otras optimizaciones cómo *inline expansion* exponen más expresiones a calcular.

PROPAGANDO CONSTANTES (CONSTANT PROPAGATION)

- Especialmente útil si no se hace *inline expansion*.
- Dada una expresión como let $x = 3$ in $x + 1$, *constant folding* no va a funcionar
- Se detecta que x es constante y se obtiene let $x = 3$ in $3 + 1$.
- Ahora sí *constant folding* va a ser efectivo.

ELIMINACIÓN DE SUBEXPRESIONES COMUNES (COMMON SUBEXPRESSION ELIMINATION)

- *Common Subexpression Elimination* (CSE) busca en el código una misma expresión que se ejecuta dos veces
- La expresión $(2 * x) + (2 * x)$ tiene una subexpresión común $(2 * x)$
- Se transforma en $\text{let } y = 2 * x \text{ in } y + y$
- Hay que tener cuidado que la optimización no modifique los efectos laterales.

OTRAS OPTIMIZACIONES

- Fusión de programas:
  - Cuando se tiene una aplicación de una función f que consume un árbol al resultado de una función g que lo produce

$$f\ (g\ x)$$

  - A veces es posible fusionar las dos funciones sin tener que generar el resultado intermedio.
- Evaluación parcial/Especialización.

EL COSTO DE LAS CLAUSURAS

- Cada aplicación da lugar a:
  - La carga de la dirección de la clausura clos[0]
  - El llamada a la dirección computada.
- Llamar a una dirección computada es típicamente 10 veces más caro que llamar a una función que se conoce estáticamente!

GENERANDO LLAMADAS ESTÁTICAS

- Las dos llamadas a f no llaman a otro código

$$\text{let } f\ x = x * 2 \text{ in } f\ (f\ 3)$$

- La función recursiva f siempre llama al código actual

$$\text{letrec } f\ x = \ldots\ f\ a\ \ldots$$

- Todas las aplicaciones de sortBy llaman a $(\text{fun } x\ y \to x < y)$

$$\text{let sortBy } ord\ xs = \ldots \text{ in}$$
$$\ldots \text{sortBy } (\text{fun } x\ y \to x < y)\ ys \ldots$$

GENERANDO LLAMADAS ESTÁTICAS

- En estos casos deberíamos generar una llamada a dirección estática:
  - aplicación de la función en el alcance de su definición,
  - llamadas recursivas,
  - funciones de alto orden aplicadas una sola vez.
- Alternativamente, se podría intentar eliminar el llamado usando expansión en línea.

RESUMEN

- Vimos varias optimizaciones:
  - Expansión en línea
  - Eliminación de código muerto
  - Eliminación de subexpresiones comunes
  - Cálculo de expresiones (constantes, álgebra)
  - Llamadas estáticas
- Habíamos visto optimización por llamada de cola.