

# Compiladores

## CS3025

*Julio Eduardo Yarasca Moscol*

- Unidad 5: Generación de código
  - 5.1. Llamadas a procedimientos y métodos en envío.
  - 5.2. Compilación separada; vinculación. Selección de instrucciones.
  - 5.3. Asignación de registros. Optimización por rendija (peephole)

## Logro de la sesión:

Discutir por qué la compilación separada limita la optimización debido a efectos de llamadas desconocidas. Discutir oportunidades para la optimización introducida por la traducción y enfoques para alcanzar la optimización, tales como la selección de la instrucción, planificación de instrucción, asignación de registros y optimización de tipo mirilla (peephole optimization).



- ▶ Un **bloque básico** es una secuencia de instrucciones con las siguientes características:
  - ▶ **Entrada única:**  
Solo se puede ingresar al bloque por su primera instrucción.
  - ▶ **Ejecución secuencial:**  
Si se ejecuta la primera instrucción, todas las demás se ejecutan en orden, sin bifurcaciones.
  - ▶ **Sin saltos ni etiquetas internas:**  
Solo la última instrucción puede ser un salto, y solo la primera puede tener etiqueta.

Bloque máximo = secuencia máxima de código lineal



- ▶ Es una representación gráfica del flujo de ejecución de un programa, utilizada en el análisis y optimización del código intermedio por parte del compilador.
- ▶ Los nodos de una gráfica de flujo de control representan los **bloques básicos**, es decir, secuencias de instrucciones que se ejecutan de forma secuencial sin bifurcaciones internas.
- ▶ Las aristas indican posibles transiciones en el flujo de control del programa. Estas se generan a partir de **instrucciones de salto condicional o incondicional**, y deben conectar con el inicio de otros bloques básicos.
- ▶ La gráfica de flujo permite identificar estructuras como **ciclos**, **condicionales** y **código inalcanzable**, facilitando optimizaciones como:
  - ▶ Eliminación de código muerto
  - ▶ Propagación de constantes
  - ▶ Reordenamiento de bloques



```
fun void main()  
var int x,fac;  
x=5;  
fac = 1;  
while 0<x do  
  fac = fac * x;  
  x = x - 1  
endwhile;  
print(fac);  
return(0)  
endfun
```

## Elabore la gráfica de flujo

```

.data
print_fmt: .string "%ld \n"
.text
.globl main
main:
pushq %rbp
movq %rsp, %rbp
subq $16, %rsp
movq $5, %rax
movq %rax, -8(%rbp)
movq $1, %rax
movq %rax, -16(%rbp)
while_0:
movq $0, %rax
pushq %rax
movq -8(%rbp), %rax
movq %rax, %rcx

```

```

popq %rax
cmpq %rcx, %rax
movl $0, %eax
setl %al
movzbq %al, %rax
cmpq $0, %rax
je endwhile_0
movq -16(%rbp), %rax
movq %rax, %rcx
movq -8(%rbp), %rax
imulq %rcx, %rax
movq %rax, -16(%rbp)
movq -8(%rbp), %rax
pushq %rax
movq $1, %rax
movq %rax, %rcx
popq %rax

```

```

subq %rcx, %rax
movq %rax, -8(%rbp)
jmp while_0
endwhile_0:
movq -16(%rbp), %rax
movq %rax, %rsi
leaq print_fmt(%rip), %rdi
movl $0, %eax
call printf@PLT
movq $0, %rax
jmp .end_main
.end_main:
leave
ret
.section .note.GNU-stack,"",@progbits

```



## DAG de un bloque básico

- ▶ El DAG (Directed Acyclic Graph) de un bloque básico es una estructura utilizada en la optimización de compiladores para representar las dependencias entre operaciones dentro de un bloque básico.
- ▶ Es un **grafo acíclico dirigido (DAG)** que permite optimizar bloques de instrucciones en Assembly x86-64, mediante:
  - ▶ Representar operaciones aritméticas y movimientos (addq, imulq, movq, etc.) realizadas en un bloque básico.
  - ▶ Detectar y eliminar subexpresiones comunes (como multiplicaciones o sumas repetidas).
  - ▶ Reutilizar resultados ya calculados en registros, evitando operaciones redundantes.
- ▶ Elementos del DAG en Assembly:
  - ▶ **Nodos:** Representan registros, constantes inmediatas o resultados de instrucciones aritméticas.
  - ▶ **Aristas:** Indican dependencias de datos entre registros e instrucciones (por ejemplo, qué operandos se usan en una suma o multiplicación).



# Preguntas y cierre de la sesión

- ▶ ¿En qué consiste la optimización mirilla?

# GRACIAS

*JULIO EDUARDO YARASCA MOSCOL*

