# Lenguajes y Representaciones Intermedias

## Diseño de Lenguajes de Programación

Miguel García Rodríguez

Material original de Francisco Ortin Soler

Escuela de Ingeniería Informática de Oviedo
Grado en Ingeniería Informática del Software

Universidad de Oviedo

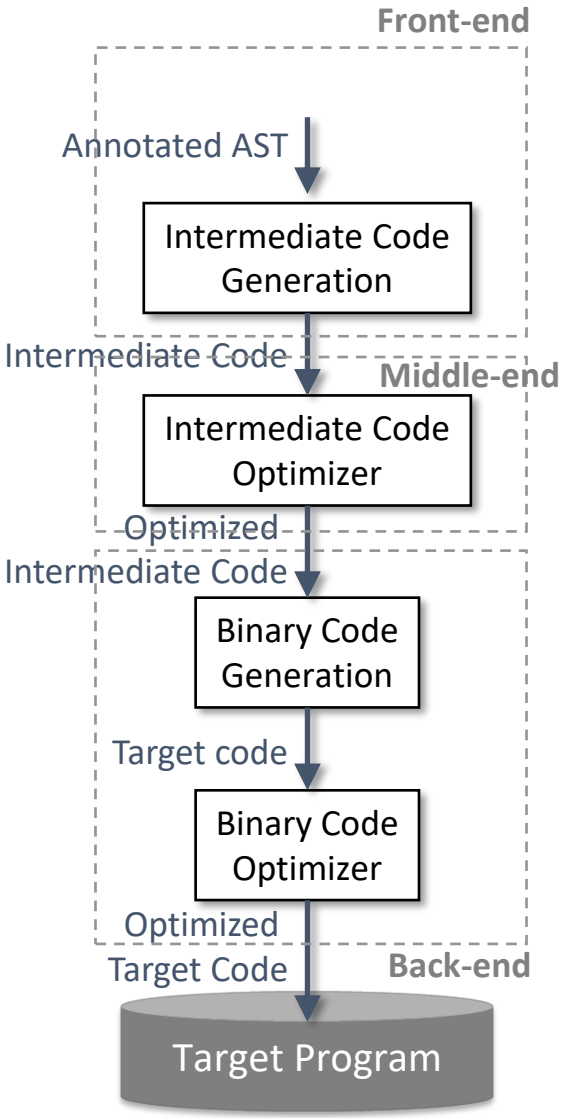
# Contenido



- **Introducción**
- Representaciones Intermedias de Alto Nivel
- Representaciones Intermedias de Medio Nivel
- Representaciones Intermedias de Bajo Nivel


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# Fases de un Traductor / Compilador




- La **representación intermedia** (***IR***) de un procesador de lenguaje representa el enlace entre su *front-* y su (*medium-*) *back-end*
- La representación es independiente de la plataforma de destino
- En la implementación de un compilador
  - Hay **múltiples representaciones intermedias**
  - En diferentes niveles de abstracción


Miguel García Rodríguez
garciaRmiguel@uniovi.es

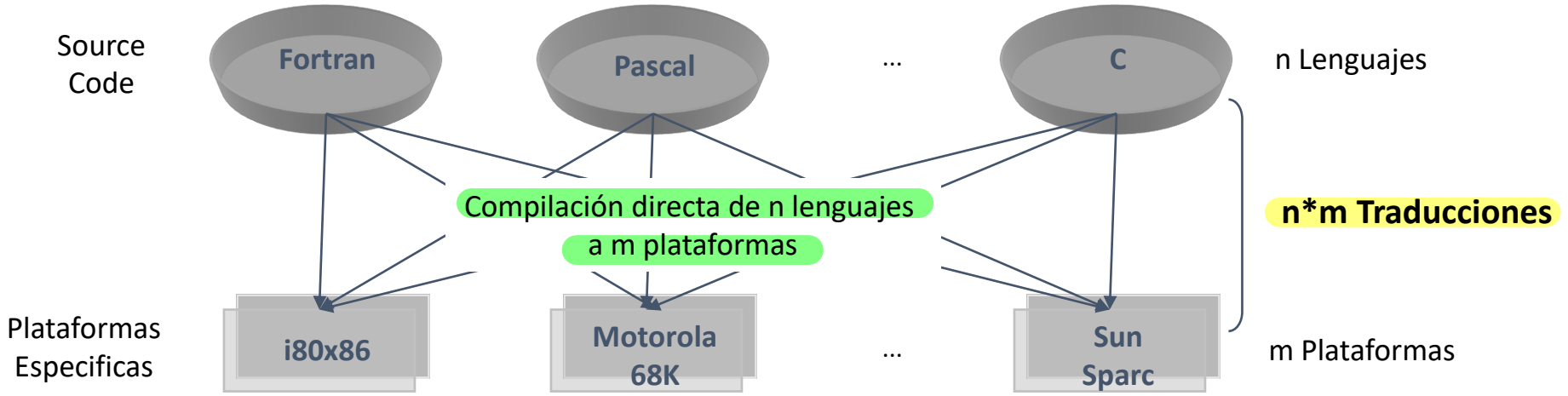
# Razón Fundamental

- Una representación intermedia común es el código de una **máquina abstracta**

  Una máquina teórica para un dispositivo hardware o una implementación software

- ¿Por qué se utilizan máquinas abstractas en la construcción de compiladores?

  Para reducir la cantidad de traducciones de código realizadas en un compilador *redirigible* (p. ej., GCC)

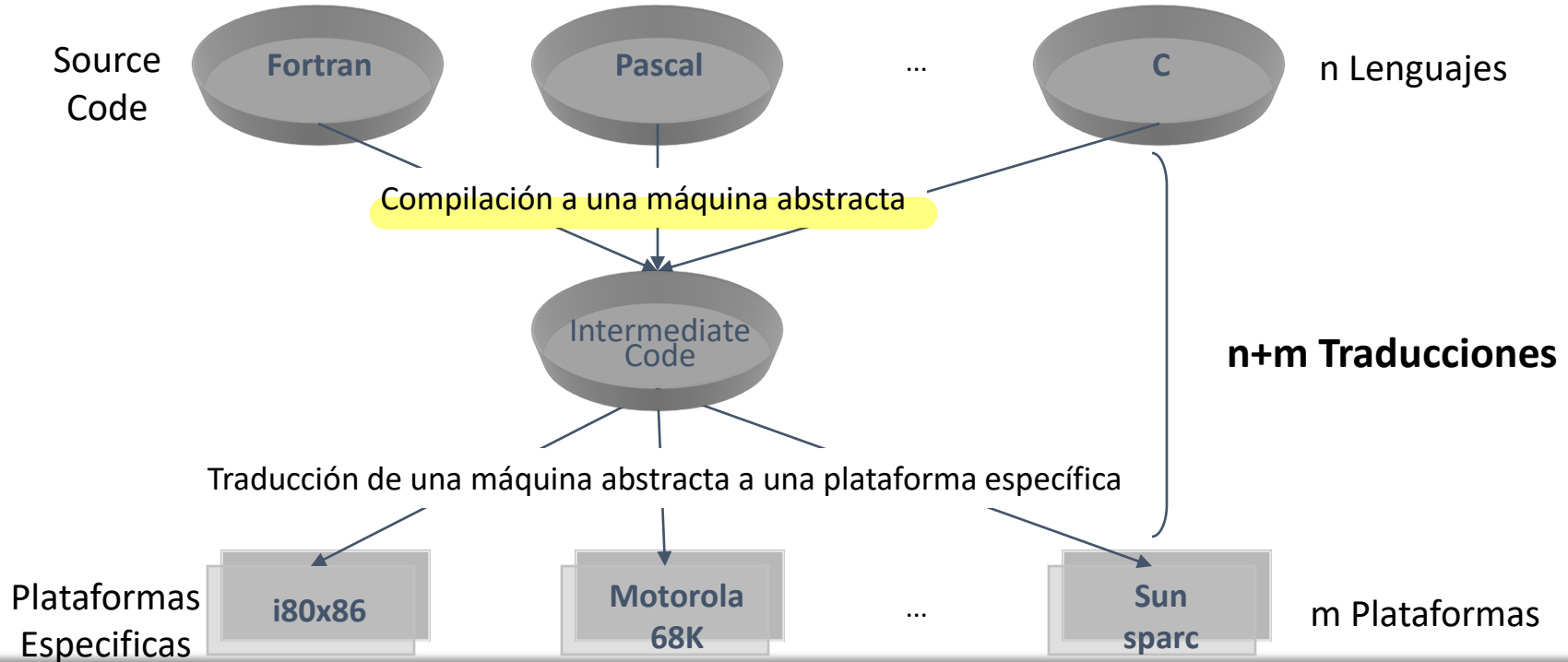




Miguel García Rodríguez
garciaRmiguel@uniovi.es

# Reducir el Número de Traducciones

- El número de traducciones se reduce de **`n*m`** a **`n+m`** cuando se utiliza una representación intermedia
- Se usa comúnmente cuando el mismo lenguaje se compila en diferentes plataformas para que sea portable (p. ej., Java)



# Tipos de Representaciones Intermedias

- Dependiendo de su nivel de abstracción, témenos lenguajes y representaciones intermedias de **alto**, **medio** y **bajo nivel**
  - También existen lenguajes y representaciones intermedias **multi nivel** (p. ej., JVM)
- La mayoría de los procesadores de lenguaje suelen utilizar **varias representaciones intermedias**, según la fase del compilador en la que se utilicen


Miguel García Rodríguez
garciaRmiguel@uniovi.es

- Introducción
- **Representaciones Intermedias de Alto Nivel**
- Representaciones Intermedias de Medio Nivel
- Representaciones Intermedias de Bajo Nivel


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# Representaciones Intermedias de Alto Nivel

- Se utilizan por primera vez en las fases de análisis
  - Por ejemplo, el analizador semántico recibe un **AST** y devuelve un AST decorado
- Ocasionalmente, estas estructuras se utilizan como código intermedio (p. ej., GNU RTL, *Register Transfer Level*)
  - Aunque principalmente son usadas internamente por el compilador (i. e., en la memoria)
- Este tipo de representaciones (lenguajes) son
  - **Dependientes** del lenguaje fuente, pero
  - **Independientes** de la Plataforma destino
- Facilitan las tareas de:
  - Inferencia y comprobación de tipos
  - Generación de código
  - Optimizaciones de Código independientes de la plataforma

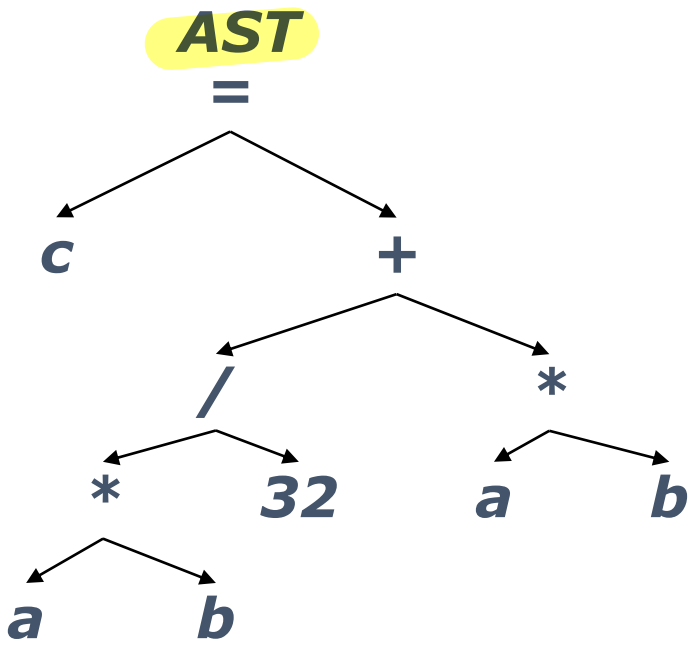

Miguel García Rodríguez
garciaRmiguel@uniovi.es

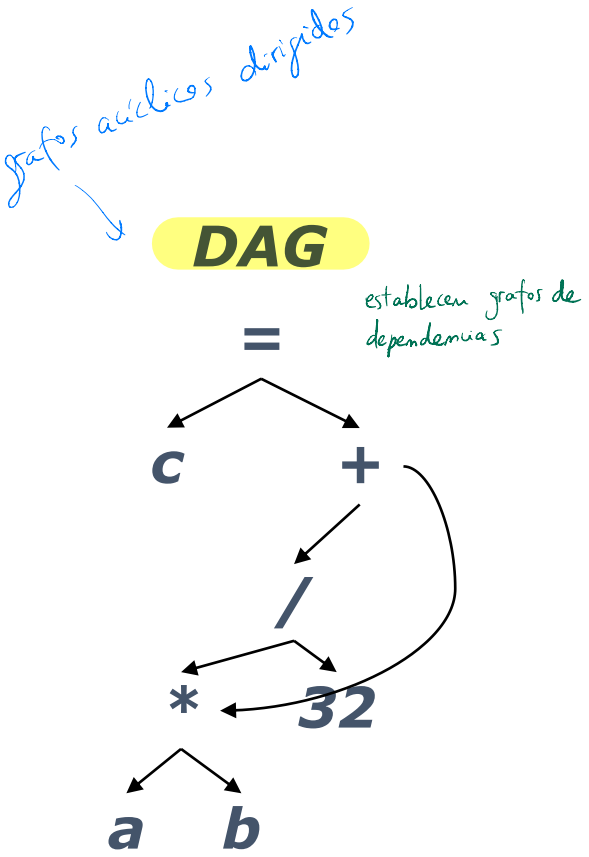
# Tipos *IR*s de Alto Nivel

- Las *IR*s de Alto Nivel más usadas son **ASTs**, **Directed Acyclic Graphs** (DAGs), **Control Flow Graphs** (CFGs) y **Data Dependency Graphs** (DDGs)

***Expresiones Java:***

c=a*b/32+a*b



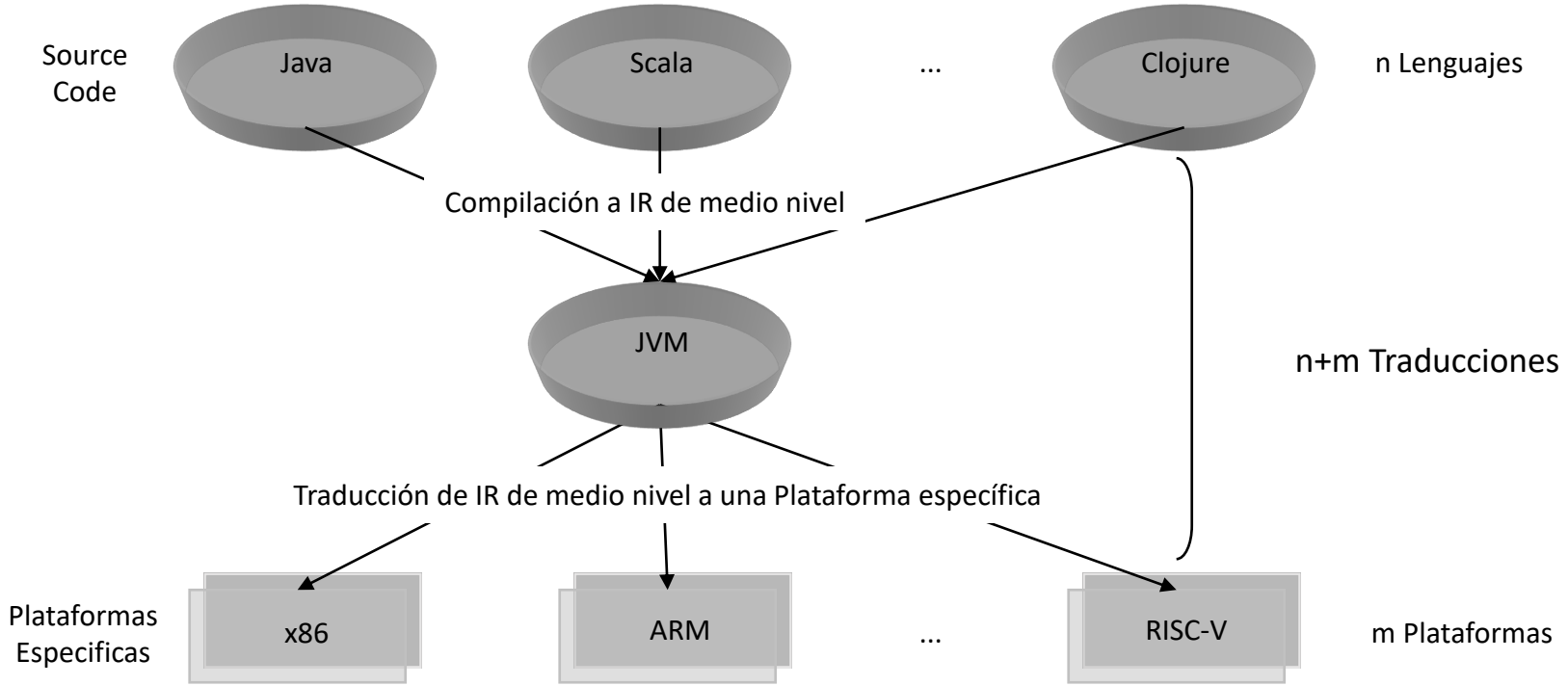



- Introducción
- Representaciones Intermedias de Alto Nivel
- **Representaciones Intermedias de Medio Nivel**
- Representaciones Intermedias de Bajo Nivel


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# Representaciones Intermedias de Medio Nivel

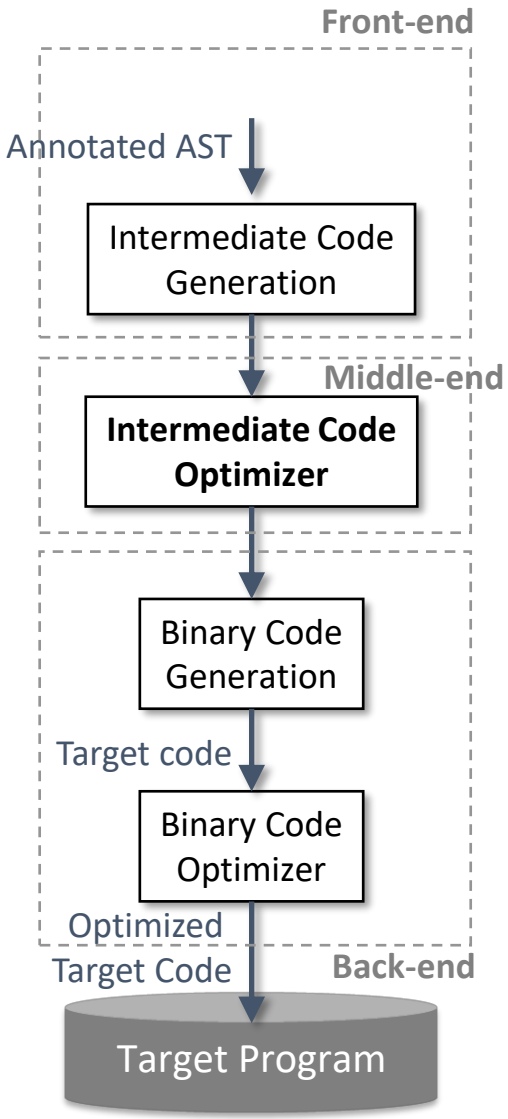
- Permiten
  - Representar una gran cantidad de lenguajes fuente
  - Generar código para una gran cantidad de plataformas de destino





Miguel García Rodríguez
garciaRmiguel@uniovi.es

# Representaciones Intermedias de Medio Nivel




- Muchas **optimizaciones** independientes del lenguaje y plataforma se realizan en este nivel (*middle-end*)
- Representan el **código intermedio** más utilizado en la implementación de un compilador
- Las **representaciones intermedias más comunes** son:
  - Máquinas de pila (p. ej., Java, .NET, p-machine)
  - Código de tres direcciones, *TAC* o *3AC* (LLVM, SunIR, UCode)
  - Notación Polaca Inversa, RPN


Miguel García Rodríguez
garciaRmiguel@uniovi.es

- Seminario: La Máquina de Pila MAPL


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# Ejercicio Obligatorio Autónomo

- Descargar MAPL del Campus Virtual
- Leer el documento MAPL.pdf
- Hacer las actividades explicadas en la última sección del documento



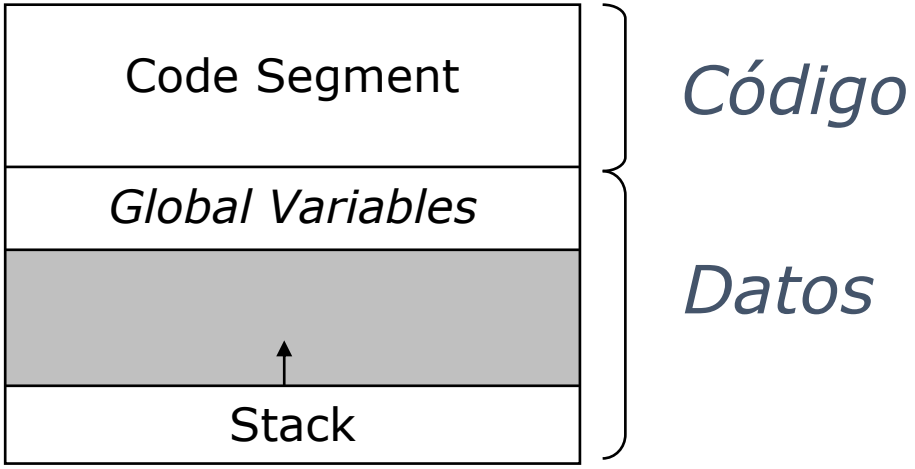

Miguel García Rodríguez
garciaRmiguel@uniovi.es

- **MAPL** es una máquina de pila abstracta usada para fines docentes
- Las máquinas virtuales textual y GUI está disponible para su descarga en el Campus Virtual
  - Es multiplataforma (precisan .NET Framework)
- La máquina virtual se utiliza en las clases de laboratorio y en el examen de laboratorio
- Aunque está diseñada para la enseñanza, es bastante similar a la JVM y la CLI (la VM de .NET)


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# MAPL: Arquitectura

- **Segmento de datos** entre 512 bytes y 16 KB (1KB por defecto, se puede configurar con **`#memory`** en el archivo de entrada)
  - No proporciona primitivas de memoria *heap*
- MAPL es una máquina de **16-bit** (2 bytes) de **palabra**
- Tiene un **segmento de código separado**
  - Tamaño fijo de instrucción (un byte por instrucción)



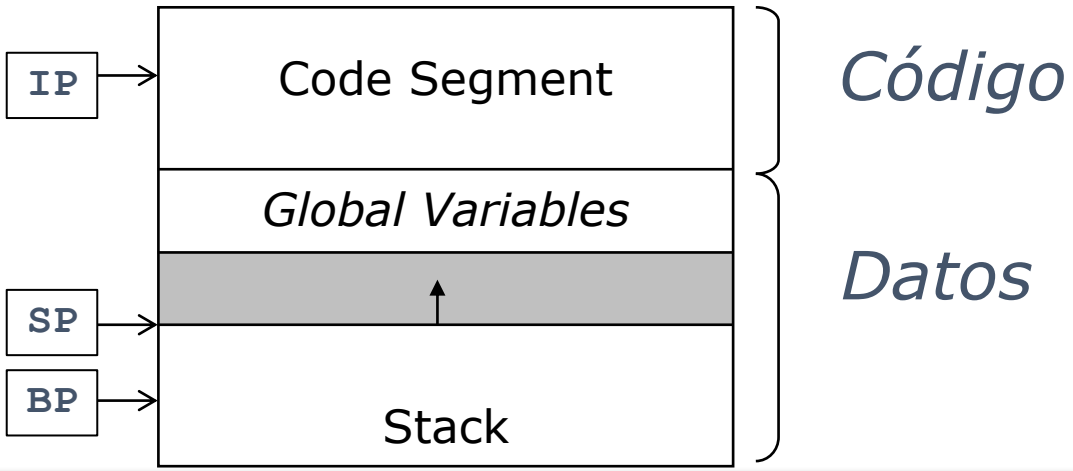


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# MAPL: Arquitectura

- **Registros**:
  - **IP**, *Instruction Pointer* -> dirección de la instrucción en ejecución (segmento de código)
  - **SP**, *Stack Pointer* -> dirección del tope de la pila (segmento de datos)
  - **BP**, *Base Pointer* -> dirección del marco de pila activo (segmento de datos)
- No hay SS, la pila comienza al final del segmento de datos

cuando mete la pila se decrementa el SP
cuando se decrementa la pila, aumenta el SP



# MAPL: Instrucciones *Push* y *Pop*

- **pushb** *ASCII_code*
  - Introduce un carácter (1 byte) en la pila
- **push[i]** *int_constant*
  - Introduce un número entero (2 bytes) en la pila
- **pushf** *real_constant*
  - Introduce un número real (4 bytes) en la pila
- **pusha** *int_constant*
  - Introduce una dirección (2 bytes) en la pila
- **push[a] bp**
  - Introduce el valor del registro **bp** (2 bytes)
- **popb**, **pop[i]**, **popf**
  - Extrae 1, 2 o 4 bytes, respectivamente, de la pila
- **dupb**, **dup[i]**, **dupf**
  - Duplica 1, 2 o 4 bytes, respectivamente, en el tope de la pila

útil para no tener que repetir operaciones largas


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# MAPL: Instrucciones *Load*

- **`loadb`, `load[i]`, `loadf`**
  1. Extrae una dirección de memoria de la pila (2 bytes)
  2. Introduce en la pila el contenido (1, 2 o 4 bytes) de la dirección extraída en el paso anterior
- Pregunta: ¿Cuál es el operador C con la misma semántica que *load*? Un puntero → *a


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# MAPL: Instrucciones *Store*

- **`storeb`, `store[i]`, `storef`**
  1. Extrae 1, 2 o 4 bytes de la pila
  2. Extrae una dirección de memoria de la pila (2 bytes)
  3. El contenido de la dirección de memoria es reemplazado con el valor extraído en el primer paso
- Pregunta: ¿Cuál es el operador C/Java con la misma semántica que *store*? La asignación → el = → a = 4


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# MAPL: Ejercicio

- Dado el siguiente programa de alto nivel

```
a = 3;
b = a;
```

```
PUSHA  0
PUSHI  3
STOREI
PUSHA  2
PUSHA  0
LOADI
STOREI
```

mete una dirección
mete un 3 ocupando 2 bytes
escribo un 3 ocupando 2 posiciones de memoria
va a la dirección 0 y lo deja en el tope de la pila

- Escribir el código destino MAPL
- Suponer que
  - Las direcciones de memoria de a y b son 0 y 2 respectivamente
  - Ambas son variables enteras
- Ejecutar el código generado con el emulador MAPL


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# MAPL: Operaciones

- Aritméticas:
  - **`add[i]`**, **`addf`**
  - **`sub[i]`**, **`subf`**
  - **`mul[i]`**, **`mulf`**
  - **`div[i]`**, **`divf`**
  - **`mod[i]`**
- Lógicas: **`and`**, **`or`**, **`not`**

- Comparación:
  - **`gt[i]`**, **`gtf`**
  - **`lt[i]`**, **`ltf`**
  - **`ge[i]`**, **`gef`**
  - **`le[i]`**, **`lef`**
  - **`eq[i]`**, **`eqf`**
  - **`ne[i]`**, **`nef`**

1. Extraen dos operandos (uno en el caso del **`not`**) de la pila
2. Realizan la operación correspondiente
3. Introducen el resultado en la pila


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# MAPL: I/O y Conversiones

- Input / Output:
  - **`outb`**, **`out[i]`**, **`outf`**: Extrae un valor de la pila (1, 2 o 4 bytes) y lo muestra en la consola
  - **`inb`**, **`in[i]`**, **`inf`**: Lee un valor del teclado e introduce su representación binaria en la pila
- Conversión:
  - **`b2i`**: Extrae un carácter (1 byte) y lo introduce como entero (2 bytes)
  - **`i2f`**: Extrae un entero (2 bytes) y lo introduce como número real (4 bytes)
  - **`f2i`**: Extrae un número real (4 bytes) y lo introduce como entero (2 bytes)
  - **`i2b`**: Extrae un entero (2 bytes) y lo introduce como carácter (1 byte)


Miguel García Rodríguez
garciaRmiguel@uniovi.es

# MAPL: Ejercicio

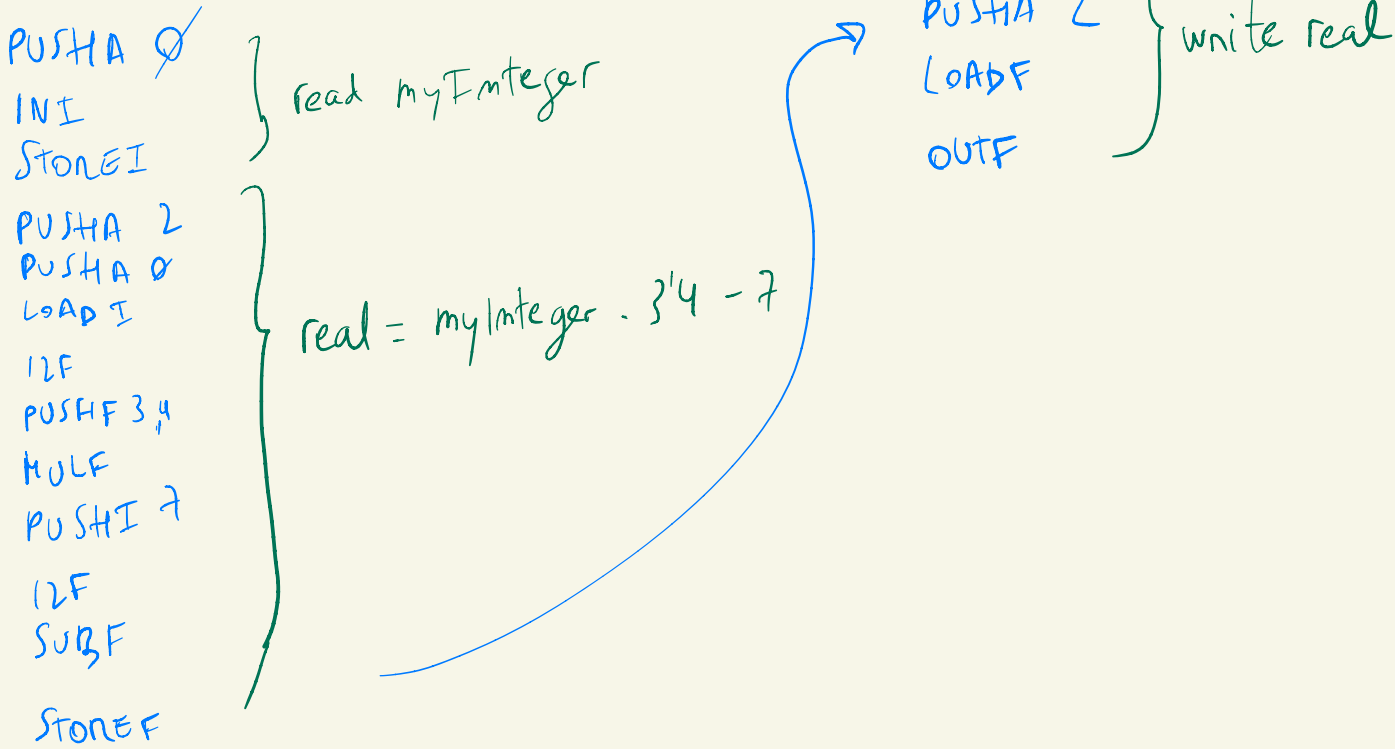
- Dado el siguiente programa de alto nivel

```
read myInteger; // Lee un número entero
real = myInteger * 3.4 – 7;
write real;  // Escribe real en la consola
```

- Escribir el Código destino MAPL
- Suponiendo que
  - Las direcciones de memoria de `myInteger` y `real` son 0 y 2 respectivamente
  - `myInteger` es un `integer` y real es `double`
- Ejecutar el código generado con el emulador MAPL


Miguel García Rodríguez
garciaRmiguel@uniovi.es

```
read myInteger; // Lee un número entero
real = myInteger * 3.4 - 7;
write real;  // Escribe real en la consola
```



- Introducción
- Representaciones Intermedias de Alto Nivel
- Representaciones Intermedias de Medio Nivel
- **Representaciones Intermedias de Bajo Nivel**

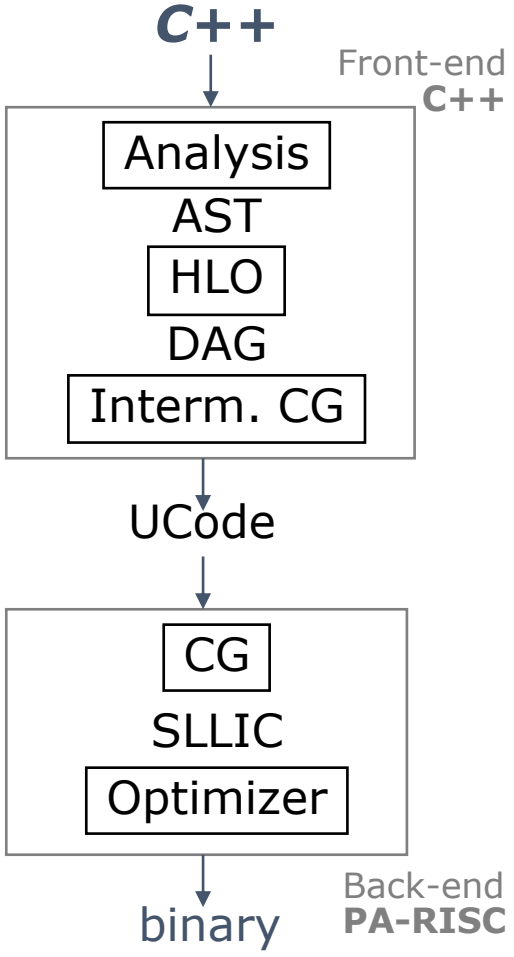
# Representaciones Intermedias de Bajo Nivel

- Lenguajes para un conjunto de microprocesadores de la misma arquitectura (p. ej., SLLIC para microprocesadores HP PA-RISC)
- Permiten lograr la **máxima optimización** para un microprocesador específico
- La semántica de sus instrucciones está cerca del microprocesador de destino
- Optimizaciones a nivel de registro
- La mayoría de ellas proporcionan **registros simbólicos**, cercanos a la arquitectura de destino


Miguel García Rodríguez
garciaRmiguel@uniovi.es

- La mayoría de los compiladores usan varias **representaciones intermedias**
- Un ejemplo es el compilador HP para la arquitectura PA-RISC

1. Los programas fuente (C, C++, Fortran y Pascal) son traducidos a representaciones intermedias de alto nivel (AST)
2. El AST es optimizado por el Optimizador de Alto Nivel (*High-Level Optimizer, HLO*) produciendo un *DAG*
3. El grafo optimizado es traducido a código de máquina de pila *UCode* (salida *front-end*)
4. De los *back-ends* existentes para HP, el de la arquitectura RISC traduce UCode a SLLIC (lenguaje de representación intermedia de bajo nivel para PA-RISC)
5. El último optimizador realiza diversas optimizaciones de bajo nivel generando el código binario



# Bibliografía




- Steven Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufman, 1997.
- Louden, K.C. Construcción de Compiladores Principios y Práctica. Thomson. 2004.
- M.L. Scott. Programming Language Pragmatics. 4th Edition. Morgan Kaufmann, 2015.
- R. Izquierdo. Máquina Abstracta MAPL, 2017.
- F. Ortin, R. Izquierdo. The MAPL Abstract Machine, 2017.




**Miguel García Rodríguez**
garciaRmiguel@uniovi.es

# ¿Preguntas?





Miguel García Rodríguez
garciaRmiguel@uniovi.es