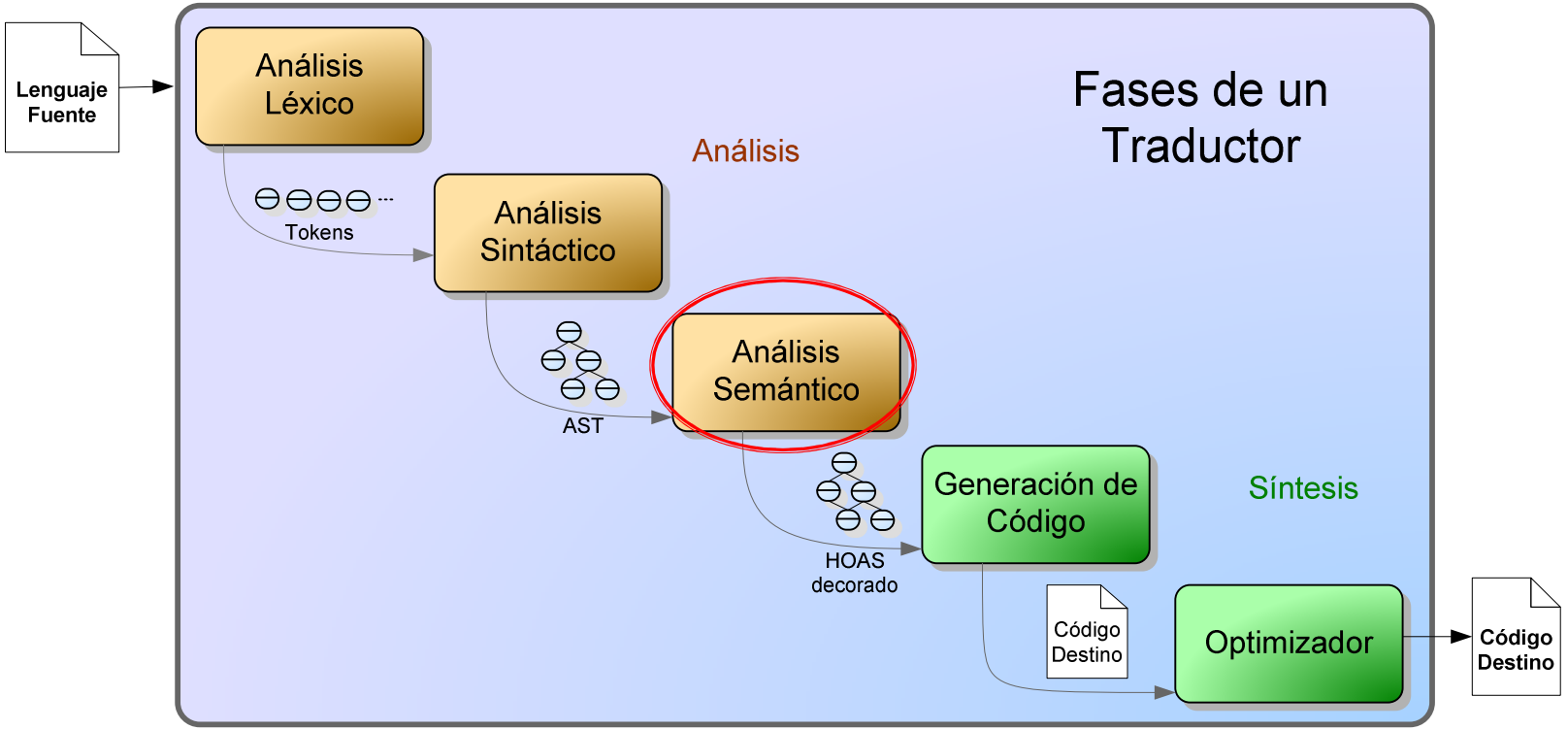
# Análisis Semántico (III)

Diseño de Lenguajes de Programación
Ingeniería Informática
Universidad de Oviedo
(v1.9)

Raúl Izquierdo Castanedo

# Usted está aquí…



# Comprobación de Tipos
# ¿Qué hace?

# Errores a detectar

**Errores que no han detectado el análisis léxico y sintáctico**

Fase de Identificación

- Chequeos de enlace
  - Uso de símbolos no definidos
- Chequeos de unicidad
  - Definiciones repetidas
  - Campos en una estructura
  - Enumerados
  - Sobrecarga de Funciones

Comprobación de Tipos

- Chequeos de Tipo
  - Expresiones
    - Que los operadores se apliquen a operandos del tipo adecuado
  - Número y tipo de los argumentos
  - Asignaciones compatibles

- Chequeos de control de flujo
  - Cada salida de una función debe retornar un valor
  - No puede haber un break fuera de un switch, o bucle *while* o *do-while*
- …

# Comprobador de Tipos

# Comprobador de Tipos

**Comprobador de Tipos**

- Parte del semántico que pretende detectar errores antes de la ejecución

- ¿Qué es un tipo?

# ¿Qué es un tipo? (I)

**Ejemplo 1**

- Supóngase un lenguaje sin tipos

```
b = f();
print !b;
```

  - ¿Se podrá hacer *!b*?

- Una forma de ver un tipo es como el conjunto de valores que puede tomar una expresión

  - Si se añaden tipos

```
real f() {...}          boolean f() {...}

b = f();                b = f();
print !b;               print !b;
```

**Un *Comprobador de Tipo* utiliza el tipo para determinar si se puede operar con una expresión:**

- Antes de ejecutar dicha operación
- Sin necesidad de conocer el valor que tendrá

# ¿Qué es un tipo? (II)

**Ejemplo 2**

- Supóngase

```
int a, b, c;
c = a + b;
```

```
int *pa, *pb, *pc;
pc = pa + pb;
```

- El Comprobador de Tipos rechaza la suma de punteros
  - Pero los valores que pueden tomar *a* y *pa* son los mismos!!

**Otra forma de ver un tipo**

- Es el conjunto de operaciones que pueden aplicarse a un símbolo
  - No solo se trata de que se pueda operar con los valores
    - Se trata además de qué intención de uso se tenga con ellos

# ¿Qué es un tipo? (III)

**En resumen**

- ¿Qué es un tipo?
  - La noción varía entre lenguajes
    - Incluso entre autores
  - Cierto consenso en que un tipo:
    - Se puede ver como un conjunto de valores
    - Se puede ver como un conjunto de operaciones

- El Comprobador de tipos utiliza ambas perspectivas antes de la ejecución para detectar:
  - Valores con los que no se puede operar
  - Operaciones en las que no se deberían usar

# ¿Cómo hacer la Comprobación de Tipos?

# Comprobador de Tipos (I)

**Se quiere detectar el uso inadecuado de expresiones en ciertas estructuras**

- Para ello hay que seguir 3 pasos

**Paso 1**

- Que cada estructura (nodo) que incorpore alguna expresión indique qué tipos (valores) acepta

| Nodo | Predicado |
|---|---|
| if | la condición debe ser de tipo booleano/entero |
| suma | Los operandos no pueden ser arrays ni estructuras |

  - Pero ¿de dónde sacan los predicados el tipo de cada expresión?

**Paso 2**

- Añadir un atributo *tipo* a todas los nodos *expresión*
  - Para que pueda ser consultado por los predicados

| Nodo | Predicado |
|---|---|
| if:expr → condicion:expr sent* sent* | condicion.tipo == BOOL |

  - Pero ¿cómo se ha calculado el valor de dicho atributo *tipo* para cada nodo?

# Comprobador de Tipos (III)

**Paso 3**

- Inferencia de Tipos
  - Proceso por el que se determina el tipo de cualquier expresión del programa
  - Se realiza siguiendo las reglas de inferencia
    - Cada una determina (infiere) cómo calcular el tipo de una expresión
      - Normalmente en función de sus hijos

| Nodo cuyo tipo queremos calcular | Predicados | Regla de inferencia para calcularlo |
|---|---|---|
| litEnt:expr → | | litEnt.tipo = ¿? |
| suma:expr → $expr_1$ $expr_2$ | | suma.tipo = ¿? |

- ¿Qué notación usar para todo esto?

# Ejemplo E1

**Supóngase**

- Solo se permite imprimir expresiones reales
- No hay conversiones implícitas

| Símbolo | Predicados | Reglas Semánticas |
|---|---|---|
| programa → defVariable* sentencia* | | |
| defVariable → tipo nombre:string | | |
| | | |
| int:tipo → λ | | |
| real:tipo → λ | | |
| | | |
| print:sentencia → expresión* | ¿ ? | |
| | | |
| literalInt:expresión → valor:string | | ¿ ? |
| literalReal:expresión → valor:string | | ¿ ? |
| variable:expresión → nombre:string | | ¿ ? |
| aritmética:expresión → left:expresión operator:string right:expresión | ¿ ? | ¿ ? |

| Nodo/Categoría | Atributo | Dominio | Heredado/Sintetizado |
|---|---|---|---|
| | ¿ ? | | |

# Tipos. Tareas y su Especificación

**Definiciones**

- Tareas a realizar
  - Inferencia de tipos (*Type Inference*)
    - Proceso de obtener los tipos
  - Comprobación de tipos (*Type Checking*)
    - Proceso de verificar un programa con información de tipos

- Formas de especificarlas
  - Sistema de Tipos (*Type System*)
    - Descripción formal de cómo se realizan las tareas anteriores
    - Se describe utilizando una notación formal
  - Comprobador de Tipos (*Type Checker*)
    - Implementación de un Sistema de Tipos
    - Se describe mediante un lenguaje de programación

# Ejercicio E2

| Nodo/Categoría | Atributo | Dominio (tipo) | H/S |
|---|---|---|---|
| | | | |

| Símbolo | Predicados | Reglas Semánticas |
|---|---|---|
| **programa** → defVariable* sentencia* | | |
| **defVariable** → tipo nombre:string | | |
| | | |
| **int**:tipo → λ | | |
| **real**:tipo → λ | | |
| **array**:tipo → tamaño:int tipoelementos:tipo | | |
| | | |
| **print**:sentencia → expresión | | |
| **while**:sentencia → expresión sentencia* | | |
| | | |
| **literalInt**:expresión → valor:string | | |
| **literalReal**:expresión → valor:string | | |
| **variable**:expresión → nombre:string | | |
| | | |
| **aritmetica**:expresión → left:expresión operator:string right:expresión | | |
| **and**:expresión → left:expresión right:expresión | | |
| **comparacion**:expresión → left:expresión operator:string right:expresión | | |
| | | |
| **cast**:expresión → destino:tipo expresión | | |
| **accesoArray**:expresión → left:expresión indice:expresión | | |

# Tipos

# Tipos. Representación (II)

**¿Cómo se modela un tipo?**

```
class DefVariable {
  ...
  private ¿...? tipo;
}
```

- Depende del lenguaje
  - Con tipos simples
  - Con tipos compuestos

# Tipos. Representación (II)

**Patrón Composite**

- Tipos primitivos
- Vector (*array*)
- Estructura
- ¿void?
- ¿tipo error?

# Ejemplo

**Implementar los tipos del siguiente lenguaje**

```
int v[10];
int w[5..25];

int *p;
float *q;
int **pp;

int *ap[25];

void (*pf)();
int (*pfp)(int a, float b);
```

# Tarea

**Dibujar los AST de cada una de las definiciones anteriores**

# Equivalencia de Tipos

# Equivalencia de tipos

**En distintas partes de un sistema de tipos hay que comprobar si dos expresiones son del mismo tipo**

- Operaciones aritméticas
- Asignaciones
- Paso de parámetros

**En los tipos simples es sencillo**

- ¿Y en los compuestos?

# Equivalencia de tipos. Estructuras

```
struct Persona {
   int edad;
   float sueldo;
}

void f(int a, Persona p) {
   …

}

struct Cliente{
   int edad;
   float sueldo;
}
Cliente raúl;
…
f(10, raúl);          // ¿se puede?
```

# Equivalencia de Tipos

## Nominal

- Dos expresiones son del mismo tipo si ambas usan el mismo nombre

```
struct Persona {
   int edad;
   float sueldo;
}

void f(int a, Persona p) {
   …
}

struct Cliente {
   int edad;
   float sueldo;
}

Cliente raúl;
…
f(10, raúl);
```

## Estructural

- Dos expresiones son del mismo tipo si
  - Cada componente del primer tipo es equivalente a un componente del segundo

# Resumen

**Comprobador de Tipos**

- Qué entra
  - Un AST con los símbolos enlazados a sus definiciones
- Qué hace
  - Detecta operaciones con valores incorrectos sin necesidad de esperar a la ejecución del programa
- Qué sale
  - Un AST con un atributo *tipo* en todas las expresiones

# Análisis Semántico
# Consideraciones finales

# Errores a detectar

**Errores que no han detectado el análisis léxico y sintáctico**

Fase de Identificación

- Chequeos de enlace
  - Uso de símbolos no definidos
- Chequeos de unicidad
  - Definiciones repetidas
  - Campos en una estructura
  - Enumerados
  - Sobrecarga de Funciones

Comprobación de Tipos

- Chequeos de Tipo
  - Expresiones
    - Que los operadores se apliquen a operandos del tipo adecuado
  - Número y tipo de los argumentos
  - Asignaciones compatibles

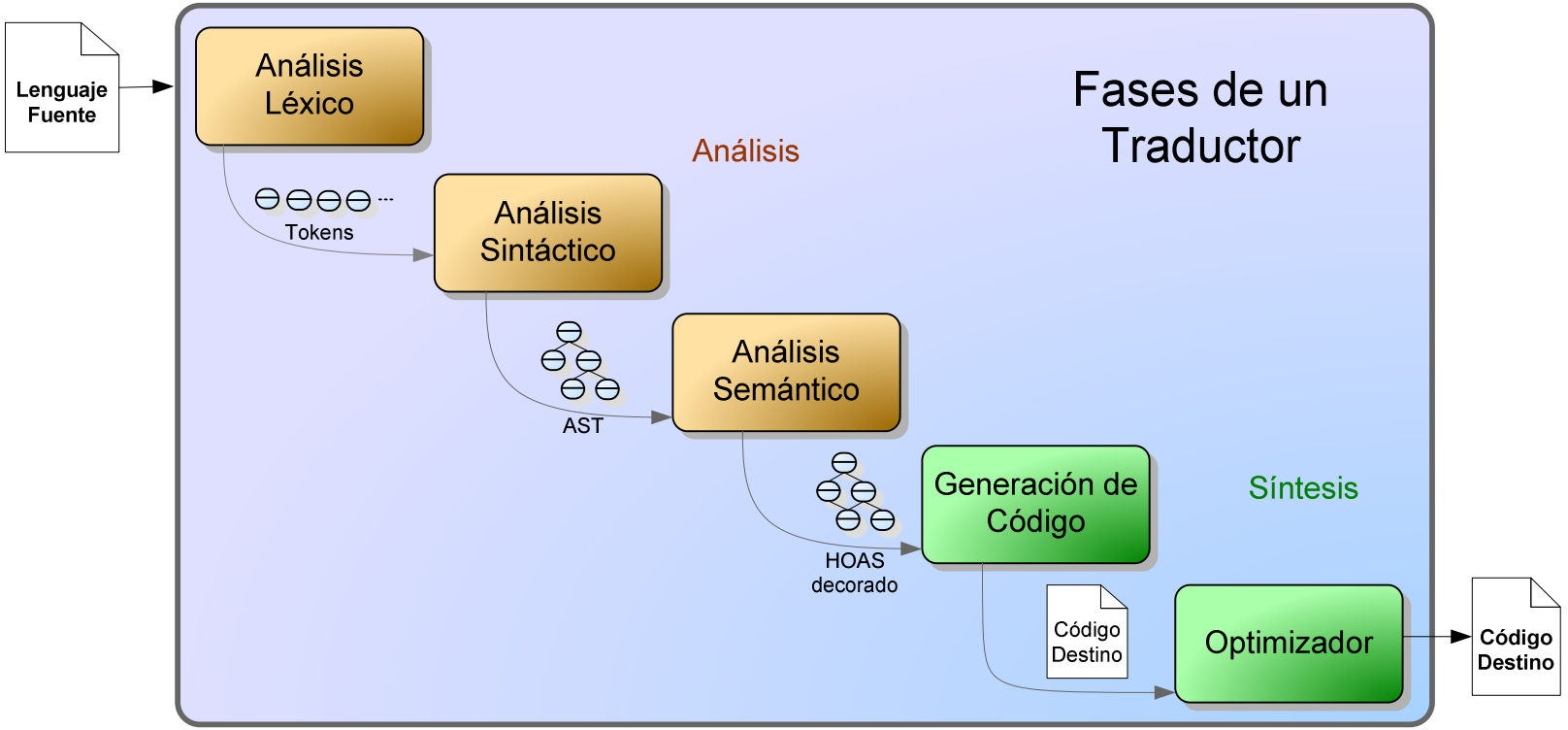
¿?

- Chequeos de control de flujo
  - Cada salida de una función debe retornar un valor
  - No puede haber un break fuera de un switch, o bucle *while* o *do-while*
- …

# Máquina Abstracta MAPL

Raúl Izquierdo Castanedo

# Usted *no* está aquí…



# MAPL

**En el campus está disponible**

- Máquina Virtual en modo texto
- Depurador Gráfico
- Manual
- Tutoriales

# Arquitectura de MAPL

# Arquitectura de MAPL (I)

**Segmentos de memoria**

- Segmento de datos de 1024 bytes
  - Ampliable
- Segmento de código separado de los datos
  - Cada instrucción ocupa una dirección

**Registros**

- IP (segmento de código). Dirección de la instrucción actual
- SP (segmento de datos). Dirección de la cima de la pila
- BP (segmento de datos). Dirección del *stack frame* (dirección de retorno y antiguo BP) de la función actual

# Arquitectura de MAPL (II)

**Tamaño de los tipos primitivo**

- char = 1 byte
- int = 2 bytes
- float = 4 bytes
- address/dirección/puntero = 2 bytes

**Distribución del segmento de datos**

- La memoria estática comienza en la dirección 0
- La pila comienza en la última dirección datos y crece hacia abajo (meter valores en la pila decrementa SP)

SP = 1024

| | |
|---|---|
| 1014 | 00 |
| 1015 | 00 |
| 1016 | 00 |
| 1017 | 00 |
| 1018 | 00 |
| 1019 | 00 |
| 1020 | 00 |
| 1021 | 00 |
| 1022 | 00 |
| 1023 | 00 |

push *128*

| | | |
|---|---|---|
| 1014 | 00 | |
| 1015 | 00 | |
| 1016 | 00 | |
| 1017 | 00 | |
| 1018 | 00 | |
| 1019 | 00 | |
| 1020 | 00 | |
| 1021 | 00 | |
| SP | 128 | **128** push |
| 1023 | 00 | |

pushf *2.14*

| | | |
|---|---|---|
| 1014 | 00 | |
| 1015 | 00 | |
| 1016 | 00 | |
| 1017 | 00 | |
| SP | 195 | |
| 1019 | 245 | **2,14** |
| 1020 | 08 | pushf |
| 1021 | 64 | |
| 1022 | 128 | 128 push |
| 1023 | 00 | |

pushb *65*

| | | |
|---|---|---|
| 1014 | 00 | |
| 1015 | 00 | |
| 1016 | 00 | |
| SP | 65 | **'A'** pushb |
| 1018 | 195 | |
| 1019 | 245 | 2,14 |
| 1020 | 08 | pushf |
| 1021 | 64 | |
| 1022 | 128 | 128 push |
| 1023 | 00 | |

# Juego de Instrucciones de MAPL

# Juego de Instrucciones

| Categoría | Bytes | Enteros (*) | Reales | Direcciones |
|---|---|---|---|---|
| Manipulación de la pila | pushb *cte* | pushi *cte* | pushf *cte* | pusha *cte* |
| | loadb | loadi | loadf | |
| | storeb | storei | storef | |
| | popb | popi | popf | |
| | dupb | dupi | dupf | |
| | | | | pusha bp |
| Aritméticas | | addi | addf | |
| | | subi | subf | |
| | | muli | mulf | |
| | | divi | divf | |
| | | mod | | |
| Lógicas | | and | | |
| | | or | | |
| | | not | | |
| Comparación > | | gti | gtf | |
| Comparación < | | lti | ltf | |
| Comparación >= | | gei | gef | |
| Comparación <= | | lei | lef | |
| Comparación == | | eqi | eqf | |
| Comparación != | | nei | nef | |
| E/S | inb | ini | inf | |
| | outb | outi | outf | |
| Conversiones (**) | | i2b | | |
| | b2i | | f2i | |
| | | i2f | | |

| Categoría | Instrucción |
|---|---|
| Salto | jmp *label* |
| | jz *label* *(**j**ump if **z**ero)* |
| | jnz *label* *(**j**ump if **n**o **z**ero)* |
| Funciones | call *label* |
| | ret *cte, cte, cte* |
| | enter *cte/-cte* |
| Otras | halt<br>nop |

(*) El sufijo *i* es opcional. Si no hay sufijo se asume que es una instrucción para enteros:

*push, load, add, gt, eq, in, „*

(**) La primera letra indica el tipo del valor y la última el tipo a convertirlo:

*i2f → Convertir entero a float*

# Secuencia de trabajo en MAPL

**Operativa en MAP**

- MAPL no tiene registros
  - Solo opera con valores en la pila

- Por tanto el proceso para operar es
  1. Llevar los valores de la memoria a la pila
  2. Operar con dichos valores
     - Las operaciones retiran sus operandos de la pila
     - Dejan en ella su resultado
  3. Guardar el resultado de la pila en la memoria

# Ejemplo E3

**Hacer en MAPL**

- Imprimir la tabla de multiplicar del 4

# Acceso a memoria

**Meter constantes**

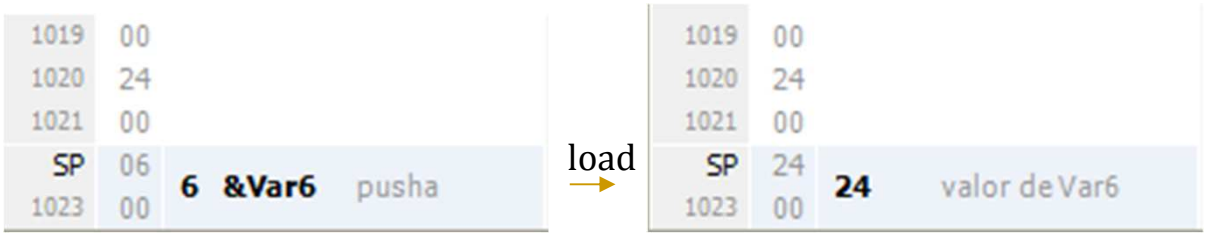
push *entero*
pusha *dirección*
push BP

**Guardar en memoria**

pusha 6
push 1492
*store*


**Leer de la memoria**

pusha 6
*load*

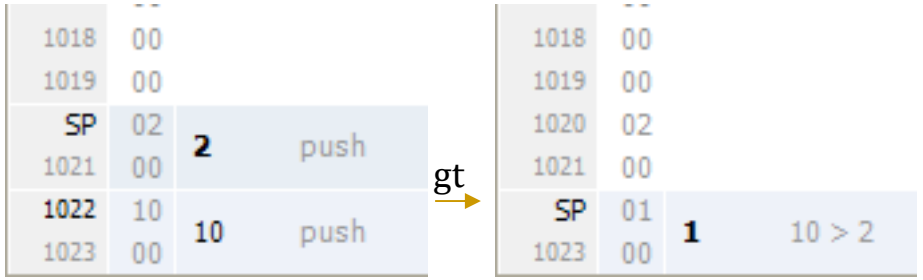


# Operadores relacionales, aritméticos y lógicos

**Operaciones**

- Comparaciones

| | | |
|---|---|---|
| *eq* | == | equal |
| *ne* | != | not equal |
| *gt* | > | greater than |
| *ge* | >= | greater or equal |
| *lt* | < | less than |
| *le* | <= | less or equal |

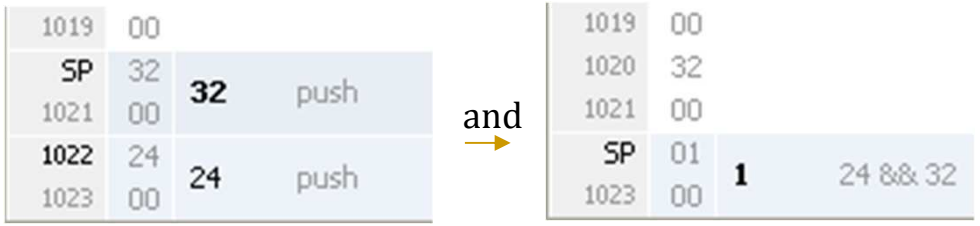



- Aritméticas

  *add*, *sub*, *mul*, *div*, mod

- Lógicas

  *and*, *or*, *not*



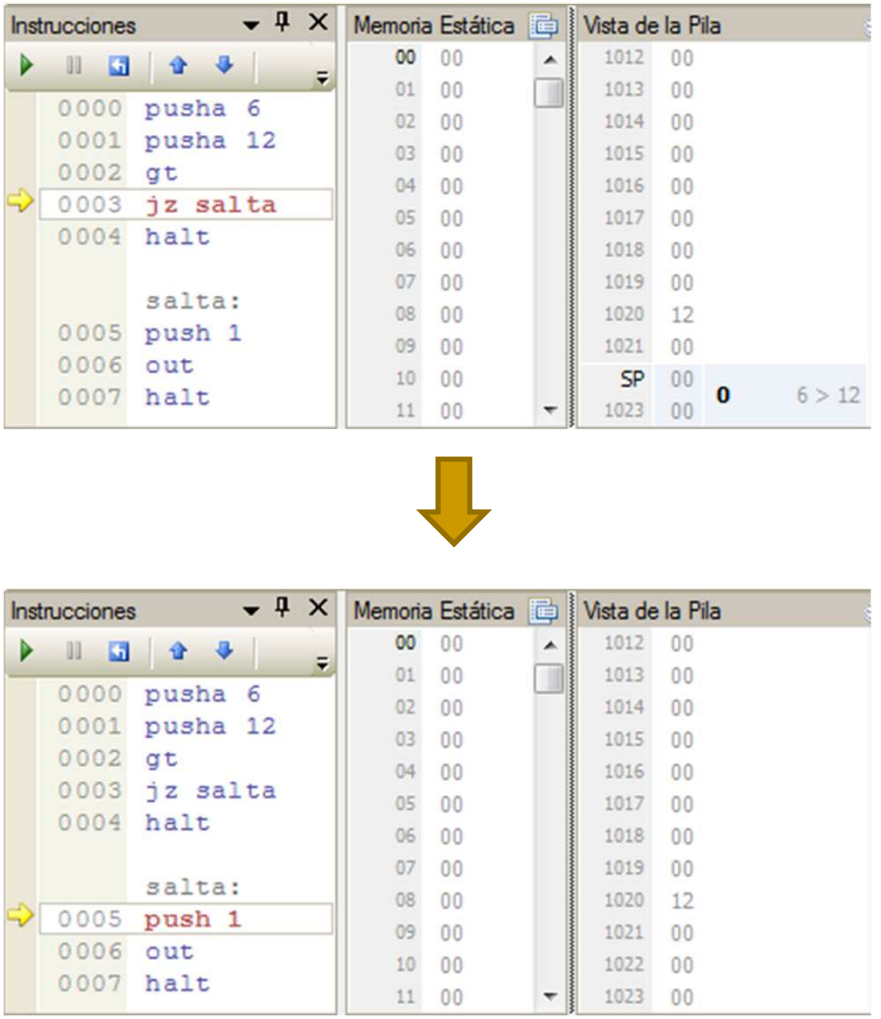
# Control de flujo

## Saltos

jmp *etiqueta*

jz *etiqueta*

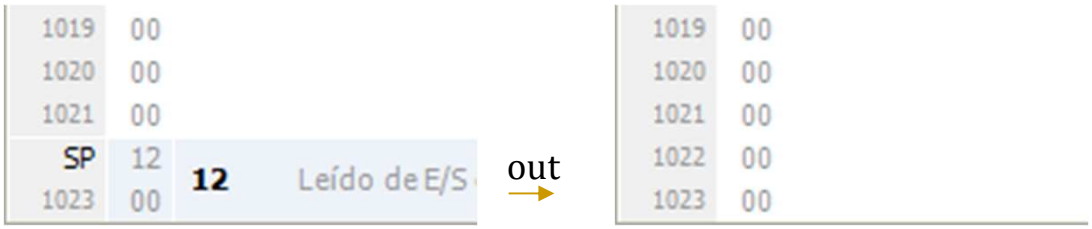
jnz *etiqueta*



# E/S

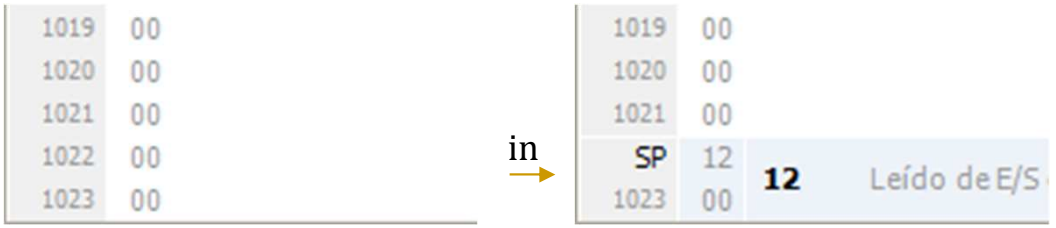
## Enviar a la Salida Estándar

push 12
*out*




## Meter un valor leído de la Entrada Estándar

*in*



# Tarea obligatoria

**Antes de la siguiente clase de teoría:**

1) Bajar MAPL del campus
2) Leer "*Manual MAPL.pdf*"
3) Seguir todos los ejemplos de la carpeta:
   "*Tutorial\1 Juego de Instrucciones*"

OBLIGATORIO

# Ejercicio E4

**Hacer en MAPL**

- Pedir dos números enteros a y b (suponemos a < b)
- Seguir pidiendo números hasta que el usuario escriba un 0
  - Para cada número introducido
    - Escribir un 1 si está entre a y b (no igual)
    - Escribir un 0 en caso contrario

# Soluciones

# Solución E1

| Símbolo | Predicados | Reglas Semánticas |
|---|---|---|
| programa → defVariable* sentencia* | | |
| defVariable → tipo nombre:string | | |
| | | |
| int:tipo → | | |
| real:tipo → | | |
| | | |
| print:sentencia → expresión* | $expresion_i$.tipo == real | |
| | | |
| literalint:expresión → valor:string | | literalInt.tipo = int |
| literalreal:expresión → valor:string | | literalReal = real |
| variable:expresión → nombre:string | | variable.tipo = variable.***definicion***.tipo |
| aritmética:expresión → left:expresión operator:string right:expresión | left.tipo == right.tipo | aritmética.tipo = left.tipo |

| Nodo/Categoría | Atributo | Dominio | Heredado/Sintetizado |
|---|---|---|---|
| expresión | tipo | tipo | sintetizado |

Atributo "definición" añadido en la Fase de Identificación

# Solución E2

| Nodo/Categoría | Atributo | Dominio (tipo) | H/S |
|---|---|---|---|
| expresión | tipo | Tipo | Sintetizado |

| Símbolo | Predicados | Reglas Semánticas |
|---|---|---|
| **programa** → defVariable* sentencia* | | |
| **defvariable** → tipo nombre:string | | |
| | | |
| **int**:tipo → | | |
| **real**:tipo → | | |
| **array**:tipo → tamaño:int tipoelementos:tipo | | |
| | | |
| **print**:sentencia → expresión | expresión.tipo ≠ array<br>(expresión.tipo == int OR<br>expresión.tipo == real)<br>expresión.tipo ∈ tiposSimples<br>simple(expresión.tipo) | |
| **while**:sentencia → expresión sentencia* | expresión.tipo == int | |
| | | |
| **literalInt**:expresión → valor:string | | literalInt.tipo = int |
| **literalReal**:expresión → valor:string | | literalReal.tipo = real |
| **variable**:expresión → nombre:string | | variable.tipo = variable.*definición*.tipo |
| | | |
| **aritmetica**:expresión → left:expresión operator:string right:expresión | left.tipo ∈ tiposSimples<br>left.tipo == right.tipo | aritmetica.tipo = left.tipo |
| **and**:expresión → left:expresión right:expresión | left.tipo == int<br>right.tipo == int | and.tipo = int |
| **comparacion**:expresión → left:expresión operator:string right:expresión | left.tipo ∈ tiposSimples<br>left.tipo == right.tipo | comparacion.tipo = int |
| | | |
| **cast**:expresión → destino:tipo expresión | expresión.tipo ∈ tiposSimples<br>destino ∈ tiposSimples<br>*expresión.tipo ≠ destino* | cast.tipo = destino |
| **accesoArray**:expresión → left:expresión indice:expresión | left.tipo == array<br>indice.tipo == int | accesoArray.tipo = left.tipo.tipoElementos |

Atributo "definición" añadido en la Fase de Identificación

**Funciones auxiliares**
*simple*(tipo) = (tipo == int) OR (tipo == real)

Equivalentes
≡

**Conjuntos auxiliares**
*tiposSimples = { int, real }*

# Solución E3

```
' dir 0: contador del 1 al 10


pusha 0
push 1
store


siguiente:
pusha 0
load
push 11
eq
jnz fin


pusha 0
load
push 4
mul
out

pusha 0
pusha 0
load
push 1
add
store

jmp siguiente

fin:
halt
```

# Solución E4

```
' dir 0: a
' dir 2: b
' dir 4: número a comprobar

pusha 0
in
store

pusha 2
in
store
```

```
otroNumero:
pusha 4
in
store

pusha 4
load
jz final

pusha 0
load
pusha 4
load
lt
```

```
pusha 2
load
pusha 4
load
gt

and
out

jmp otroNumero

final:
halt
```