

SEMANTICA OPERACIONAL

ENTIDADES CON LAS QUE TRABAJAN LOS PROGRAMAS

3

SEMÁNTICA DE LOS LENGUAJES DE PROGRAMACIÓN

p014estructuras.rb

Ruby

```
var = 5
if var > 4
  puts "La variable es mayor que 4"
  puts "Puedo tener muchas declaraciones a la vez"
  if var == 5
    puts "Es posible tener if y else anidados"
  else
    puts "Too cool"
  end
else
  puts "La variable no es mayor que 4"
  puts "Puedo tener muchas declaraciones a la vez"
end

# Loops
var = 0
while var < 10
  puts var.to_s
  var += 1
end
```

¿Qué
elementos encontramos?
¿Cuáles son las
ENTIDADES principales?

```
program binding_example(input, output);
```

```
procedure A(I : integer; procedure B;
```

```
begin
  writeln(I);
end;
```

Pascal

```
begin (* A *)
  if I > 1 then
    P
  else
    A(2, B);
  end;

  procedure C; begin end;

  begin (* main *)
    A(1, C);
  end.
```

```
struct complex {
    double real, imaginary;
};
enum base {dec, bin, oct, hex};
```

```
int i;
complex x;
```

C++

```
void print_num(int n) { ...
void print_num(int n, base b) { ...
void print_num(complex c) { ...
```

```
print_num(i);          // uses the first if
print_num(i, hex);     // uses the second if
print_num(x);          // uses the third if
```

```
#include <stdio.h>
```

```
int x = 1;
```

```
int f() {
    x += 1;
    return x;
}
```

C

```
int p(int a, int b) {
    return a + b;
}
```

```
main() {
    printf("%d\n", p(x, f()));
    return 0;
}
```

generic

```
type T is private;
with function "<"(x, y : T) return Boolean;
function min(x, y : T) return T;
```

```
function min(x, y : T) return T is
begin
    if x < y then return x;
    else return y;
    end if;
end min;
```

ADA

```
function string_min is new min(string, "<");
function date_min is new min(date, date_precedes)
```

SEMÁNTICA DE LOS LENGUAJES DE PROGRAMACIÓN

ENTIDADES	ATRIBUTOS
Variable	Nombre Tipo área de memoria, etc
Rutina	nombre, parámetros formales, parámetros reales, etc
Sentencia	acción asociada

DESCRIPTOR:

Lugar(repositorio) donde se almacena la información de los atributos

CONCEPTO DE LIGADURA (BINDING)

Los **programas** trabajan con **entidades**



Las **entidades** tienen **atributos**



Estos **atributos** tienen que **establecerse** **antes** de poder **usar** la **entidad**



LIGADURA: es el **momento** en el que el **atributo** se **asocia** con un **valor**

LIGADURA

Diferencias entre los lenguajes de programación

- El **número de entidades**
- El **número de atributos** que se les pueden ligar
- El **momento** de la **ligadura** (**binding time**).
(estática y dinámica)
- La **estabilidad** de la **ligadura**: ¿una vez establecida se puede modificar? (¿constantes?)

MOMENTO Y ESTABILIDAD DE LA LIGADURA

○ Ligadura es Estática

- Se establece antes de la ejecución.
- No se puede cambiar.
- El termino **estática** referencia al momento del binding y a su **estabilidad**.

○ Ligadura es Dinámica

- Se establece en el momento de la **ejecución**
- Si puede cambiarse durante ejecución de acuerdo a alguna **regla especifica del lenguaje**.
- **Excepción: constantes** (el binding es en runtime pero no puede ser modifica luego de establecida)

MOMENTO DE LIGADURA

- Definición del lenguaje
- Implementación del lenguaje
- Compilación (procesamiento)
- Ejecución



E
S
T
A
T
I
C
O

D
I
N
A
M
I
C
O

Veamos el siguiente ejemplo

MOMENTO Y ESTABILIDAD

Ejemplos:

- En **Definición del lenguaje**

- **Forma** de las **sentencias**
- Estructura del **programa**
- **Nombres** de los **tipos** **predefinidos**

- En **Implementación**

- **Representación** de los **números**
- sus **operaciones**

- En **Compilación**

- Asignación del tipo a las **variables**

En lenguaje C

int

Se usa esto para denominar a los enteros

int

- **Representación de máquina**

- **Operaciones** que pueden realizarse sobre enteros

int a

- Se **liga tipo** a la **variable(atributos)**

MOMENTO Y ESTABILIDAD

Ejemplo en lenguaje C

- En Ejecución
 - Variables con sus valores
 - Variables con su lugar de almacenamiento

int a

a=10

a=15

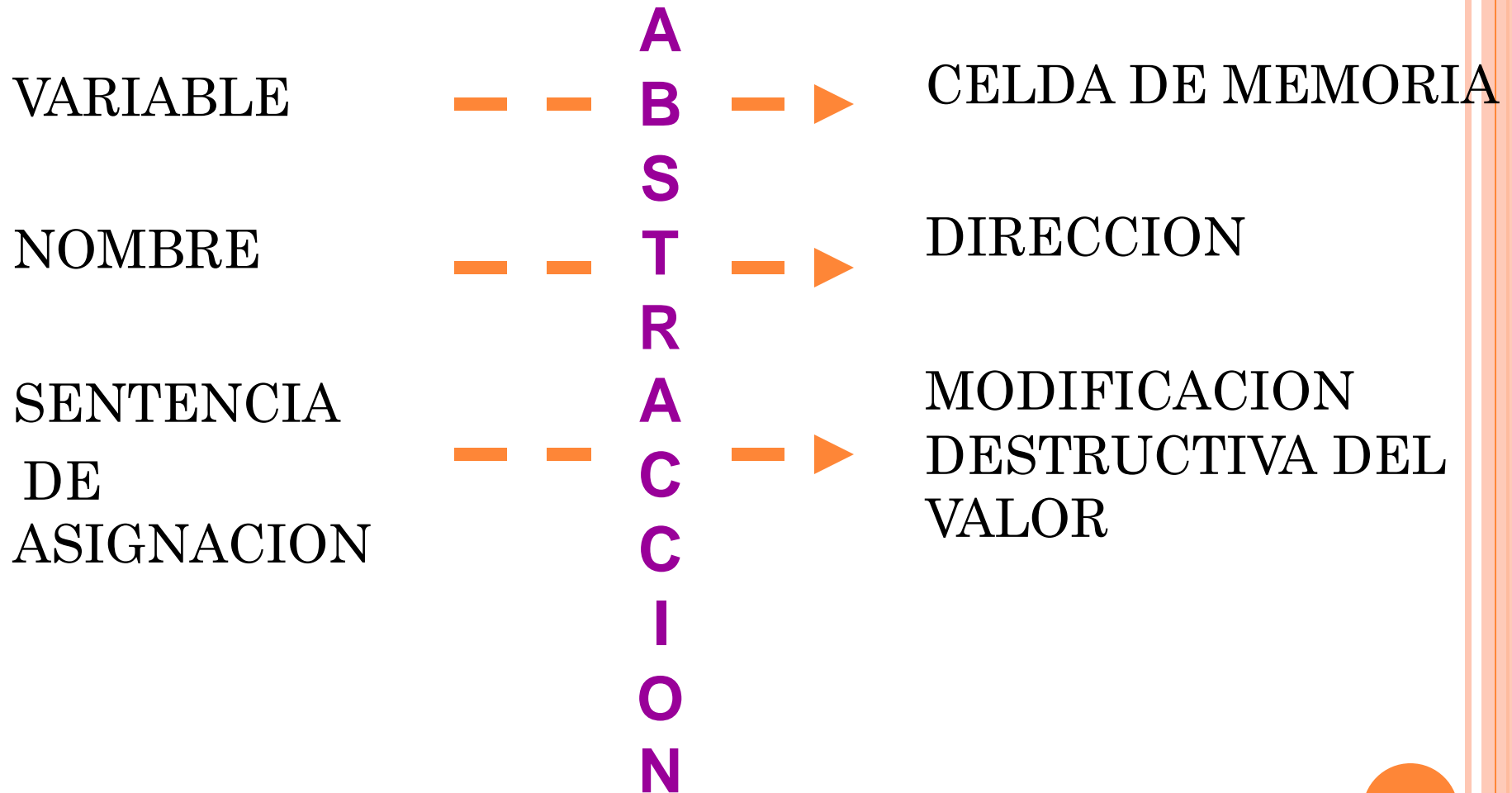
el **valor** de una variable entera se **liga en ejecución**.
puede cambiarse muchas veces.



SEMANTICA OPERACIONAL VARIABLES

12

VARIABLE



VARIABLES

CONCEPTO

$$\mathbf{x = 8 \dots x = y}$$

¿Qué me dispara esa sentencia?

¿Me da alguna información? ¿Cuál?

¿Es correcto?

¡¡Muchas cosas van a depender de sus atributos!!

VARIABLES

CONCEPTOS

Atributos de una variable

- **Nombre**
- **Alcance**
- **Tipo**
- **l-valor**
- **r-valor**

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

- ***Nombre:*** string de caracteres que se usa para referenciar a la variable. (identificador)
- ***Alcance:*** es el rango de instrucciones en el que se conoce el nombre, es visible, y puede ser referenciada
- ***Tipo:*** es el tipo de variables definidas, tiene asociadas rango de valores y operaciones permitidas
- ***L-value:*** es el lugar de memoria asociado con la variable, está asociado al tiempo de vida (variables se alocan y desalocan)
- ***R-value:*** es el valor codificado almacenado en la ubicación de la variable

ATRIBUTOS <NOMBRE, ALCANCE ,TIPO, L-VALUE, R-VALUE>

Aspectos de diseño del nombre:

- **Longitud máxima** (depende de definición del lenguaje)

Fortran: 6 caracteres

C: depende del **compilador**, suele ser de **32** caracteres y se ignora si hay más

Python, Pascal, Java, ADA: cualquier longitud

ATRIBUTOS <NOMBRE, ALCANCE ,TIPO, L-VALUE, R-VALUE>

Aspectos de diseño del nombre:

- ***Caracteres aceptados*** en el nombre (conectores)
 - Python, C, Pascal: _
 - Ruby:
 - solo letras minúsculas para variables locales
 - \$ para comenzar nombres de variables globales
- ***Sensitivos*** (Sum = sum = SUM ?)
 - C y Python sensibles a mayúsculas y minúsculas
 - Pascal no sensible a mayúsculas y minúsculas
- ***palabra reservada - palabra clave*** (reservada es aquella palabra clave propia del lenguaje que no puedo utilizar para asignar a un identificador, depende de cada LP)

ATRIBUTOS <NOMBRE, ALCANCE ,TIPO, L-VALUE, R-VALUE>

- El **alcance** de una **variable** es el **rango de instrucciones** en el que se **conoce el nombre**.
(visibilidad)
- Las **instrucciones** del programa pueden **manipular una variable** a través de su **nombre dentro de su alcance**
- Los diferentes **lenguajes** adoptan **diferentes reglas** para **ligar un nombre a su alcance**
- que veremos a continuación

ATRIBUTOS <NOMBRE, ALCANCE
,TIPO, L-VALUE, R-VALUE>

REGLAS PARA LIGAR UN NOMBRE A SU ALCANCE

- **ALCANCE ESTÁTICO**
- **ALCANCE DINÁMICO**

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

○ **Ligadura de Alcance estático**

- Llamado **alcance léxico**.
- Se define el **alcance** en términos de la **estructura léxica del programa**.
- Puede **ligarse estáticamente** a una **declaración** (explícita o implícita) **examinando el texto del programa, *sin necesidad de ejecutarlo***.
- La **mayoría de los lenguajes adoptan reglas de ligadura de alcance estático**.

ATRIBUTOS <NOMBRE, ALCANCE ,TIPO, L-VALUE, R-VALUE>

○ Ligadura de Alcance dinámico

- Define el **alcance** del nombre de la **variable** en términos de la **ejecución del programa**.
- Cada **declaración de variable** extiende su efecto *sobre todas las instrucciones ejecutadas posteriormente*, hasta que una **nueva declaración** para una **variable con el mismo nombre** es encontrada **durante la ejecución**.
- Usada por: **APL**, **Lisp** (original), **Afnix** (llamado *Aleph* hasta el 2003), **TCl** (Tool Command Language), **Perl**

EJEMPLO DE ALCANCE LENGUAJE C - LIKE

```
int x;
{
  /*bloque A*/
  int x;
  ....
}

{
  /*bloque B*/
  int x;
  ....
}

{
  /*bloque C*/
  x = ...;
  ...
}
...
```

Ejecución:

- Con alcance Dinámico

Nos preguntamos ¿quién lo llamó?
quién llamó a x de C?

Si:

A llama a C:
Toma x de A

B llama a C:
Toma x de B

- Con alcance Estático

Preguntamos ¿Dónde está contenida?
en ambos casos hace referencia a **x externa declarada**

Alcance Estático es el más usado por los lenguajes (ej. C, Pascal, Java, etc.)

- **Alcance Dinámico es menos legible,** pensar en grandes programas con cientos de instrucciones

EJEMPLO DE ALCANCE LENGUAJE PASCAL - LIKE

```
1 Program Alcance;
2   var
3       a : Integer;
4       z , b: Real;
5   procedure uno();
6       var
7           b: Integer;
8       procedure dos();
9           begin
10              z:=a+1+b;
11          end;
12   begin
13       b:= 20;    dos();
14   end;
15   procedure tres();
16       var
17           a: Real;
18       begin
19           a:=20;  uno();
20       end;
21   Begin
22   a:= 4;    b:= 2;    z:=10;    tres();
23   end.
```

Ejecución:

○ Alcance estático:

Al invocar a *tres*:

- Se invoca a **uno**
- Se invoca a **dos** y
 $z := a + 1 + b;$

Toca a **z** de **Alcance**

La variable **a** es de **Alcance**
(4)

La variable **b** es de **uno** (20)

○ Alcance dinámico:

Al invocar a *tres*:

- Se invoca a **uno** y
- Se invoca a **dos** y
 $z := a + 1 + b;$

Toca a **z** de **Alcance**

La variable **a** es la de **tres**
(20)

La variable **b** es la de **uno**
(20)

EJEMPLO DE ALCANCE LENGUAJE C – ALCANCE ESTÁTICO

compileonline.com - Compile and Execute C Online (GNU GCC)

Compile & Execute main.c input.txt

```
1 #include <stdio.h>
2 int x;
3 int y;
4
5 void uno()
6 {
7     printf ("\n EN uno \n");
8     x= x+y;
9     printf ("x en uno= %d \n", x);
10    printf ("e y en uno= %d\n", y);
11 }
12
13 void main()
14 {
15     x=1;
16     y=1;
17     printf (" ANTES de entrar al bloque \n");
18     printf ("x en main= %d\n", x);
19     printf ("y en main= %d\n", y);
20
21     {
22         printf ("\n EN el bloque \n");
23         int x;
24         x=10;
25         x=x+y;
26         printf ("x en el bloque= %d\n", x);
27         printf ("y en bloque= %d\n", y);
28         uno ();
29     }
30
31     printf ("\n DESPUES de salir al bloque \n");
32     printf ("x en main= %d\n", x);
33     printf ("y en main= %d\n", y);
34 }
35
36
```

X Y X'

Result

Compiling the source code....
\$gcc main.c -o demo -lm -pthread -lgmp -lreadline 2>&1

Executing the program....
\$demo

ANTES de entrar al bloque
x en main= 1
y en main= 1

EN el bloque
x en el bloque= 11
y en bloque= 1

EN uno
x en uno= 2
e y en uno= 1

DESPUES de salir al bloque
x en main= 2
y en main= 1

El alcance de un nombre se extiende desde su declaración hacia los bloques anidados a menos que aparezca otra declaración para el nombre

EJEMPLO DE ALCANCE LENGUAJE PASCAL – ALCANCE ESTÁTICO

compileonline.com - Compile and Execute Pascal Online (fpc 2.6.2)

Compile & Execute

Main Program

input.txt

Default Ace Editor

☐ Unit Support

```
1 Program Alcance;
2
3   var
4     x: integer;
5     y: integer;
6
7   procedure uno();
8   begin
9     x:= x+y;
10    writeln("x en uno= ", x, ' e "y" en uno= ', y);
11  end;
12
13  procedure dos();
14  var x:integer;
15
16  procedure tres();
17  begin
18    x:=x+10;
19    writeln("x en tres= ", x, ' e "y" en tres= ', y);
20    uno();
21  end;
22  begin
23    x:=10;
24    tres();
25  end;
26
27
28  begin
29    x:=1;
30    y:=1;
31    writeln("x en main= ", x, ' e "y" en main= ', y, ' ANTES de llamar a procedimiento dos');
32    dos();
33    writeln("x en main= ", x, ' e "y" en main= ', y, ' DESPUES de llamar a procedimiento dos');
34  end.
```

Result

Down

Compiling the source code....
\$fpc -v0 Alcance.pas 2>&1

Free Pascal Compiler version 2.6.2 [2013/02/16] for x86_64
Copyright (c) 1993-2012 by Florian Klaempfl and others
/usr/bin/ld: warning: link.res contains output sections; did you forget -

Executing the program....
\$Alcance

"x" en main= 1 e "y" en main= 1 ANTES de llamar a procedimiento dos
"x" en tres= 20 e "y" en tres= 1
"x" en uno= 2 e "y" en uno= 1
"x" en main= 2 e "y" en main= 1 DESPUES de llamar a procedimiento dos

X

Y

X'

Nos preguntamos ¿Dónde está contenida?

EJEMPLO DE ALCANCE LENGUAJE ADA - ALCANCE ESTÁTICO

Compile | Execute hello.adb x

```
1 with Text_IO, Ada.Integer_Text_IO;
2 use Text_IO, Ada.Integer_Text_IO;
3
4 procedure Principal is
5   y: integer;
6   procedure Prueba is
7     x: constant integer := 3+y;
8     y: integer:=4;
9     begin
10      Put("El valor de la constante x es:");
11      Put(x);
12      Put("    El valor de la variable y es:");
13      Put(y);
14    end Prueba;
15
16
17 begin
18
19   y:=7;
20   Prueba;
21
22 end Principal;
```

X Y

Y'

Demuestra que el
alcance es estático,
es desde dónde se
declara hacia abajo.
Toma el **y** de Principal

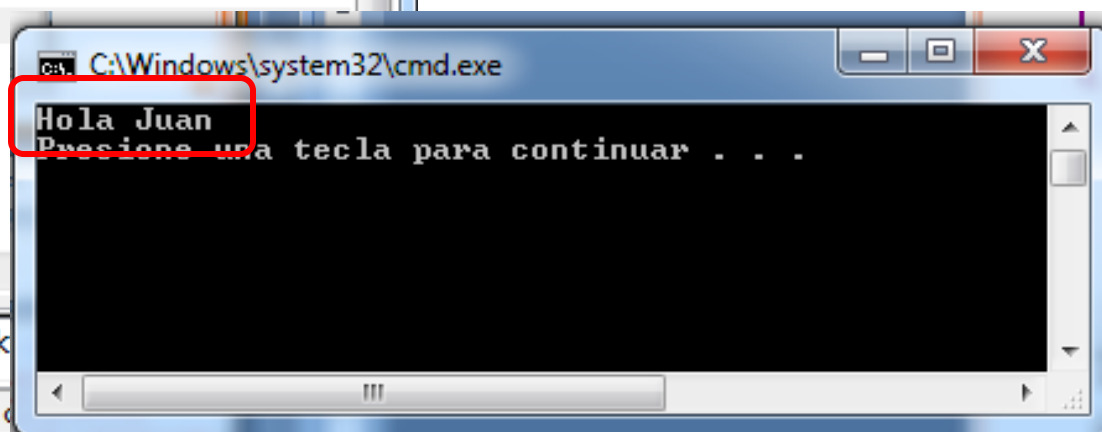
Terminal

```
gcc -c hello.adb
hello.adb:4:11: warning: file name does not match unit name, should be "principal.adb"
gnatbind -x hello.ali
gnatlink hello.ali -o hello
sh-4.2# hello
El valor de la constante x es: 10 El valor de la variable y es: 4
sh-4.2#
```

EJEMPLO DE ALCANCE LENGUAJE PYTHON - ALCANCE ESTÁTICO

```
1 def alcance1():  
2     print x+ ' Juan'  
3  
4  
5 def alcance2():  
6     x='Chau'  
7     alcance1()  
8  
9 x='Hola'  
10 alcance2()  
11  
12
```

Demuestra que el
alcance es estático. Por
más que a alcance1 se lo
llame desde alcance2, la
variable **X** tomada es la
del programa principal



Python tiene reglas de alcance estático, tipado dinámico y fuertemente tipado.
No confundir

ALCANCE ESTÁTICO VS DINÁMICO

Las reglas de Alcance Estático:

- Son las **más utilizadas** por los LP (C, PASCAL, ADA PYTHON, ETC.)

Las reglas de Alcance dinámico:

- Son **menos utilizadas** por los LP
- Se dice que son **más fáciles de implementar**
- Son **menos claras** en cuanto a disciplina de programación. **Encontrar una declaración en el flujo de ejecución puede ser duro.** El código se hace **más difícil de leer y seguir**, en **grandes programas** con cientos de sentencias es complejo

CONCEPTOS ASOCIADOS CON EL ALCANCE

CLASIFICACIÓN DE VARIABLES POR SU ALCANCE

- **Global:** Son todas las referencias creadas en el programa principal.
- **Local:** Son todas las referencias que se han creado dentro de una unidad (programa o subprograma).
- **No Local:** Son todas las referencias que se utilizan dentro del subprograma pero que no han sido creadas en el subprograma. (son externas a él)

CONCEPTOS ASOCIADOS CON EL ALCANCE - PASCAL

compileonline.com - Compile and Execute Pascal Online (fpc 2.6.2)

Compile & Execute Main Program input.txt Default Ace Editor Unit Support

```
1 Program Alcance;
2
3 var
4   x: integer;
5   y: integer;
6
7 procedure uno();
8 begin
9   x:= x+y;
10  writeln("x en uno= ", x, ' e "y" en uno= ', y);
11 end;
12
13 procedure dos();
14   var x:integer;
15
16   procedure tres();
17   begin
18     x:=x+10;
19     writeln("x en tres= ", x, ' e "y" en tres= ', y);
20     uno();
21   end;
22 begin
23   x:=10;
24   tres();
25 end;
26
27
28 begin
29   x:=1;
30   y:=1;
31   writeln("x en main= ", x, ' e "y" en main= ', y, ' ANTES de llamar a procedimiento dos');
32   dos();
33   writeln("x en main= ", x, ' e "y" en main= ', y, ' DESPUES de llamar a procedimiento dos');
34 end.
```

Referencia Global

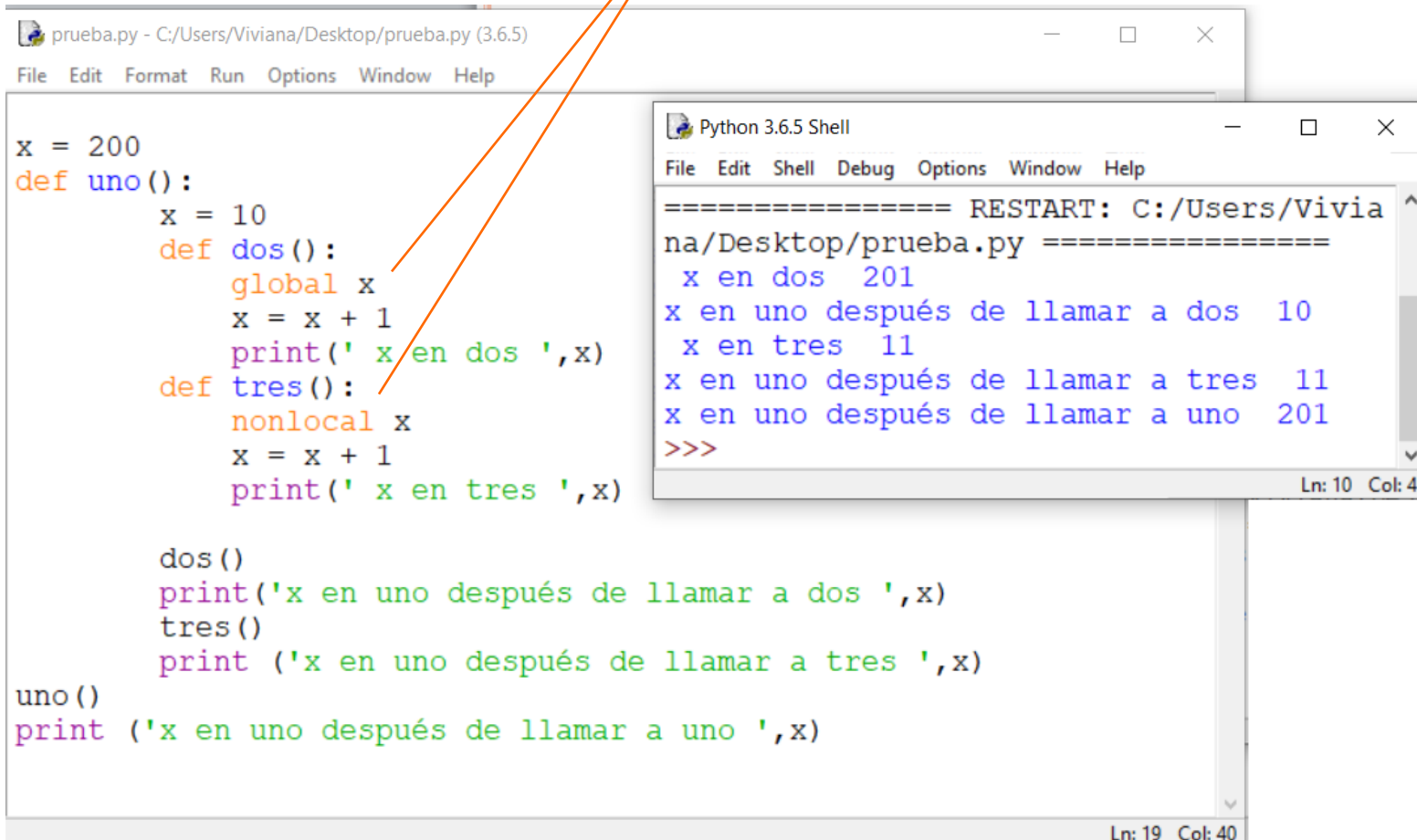
Referencia No Local

Referencia Global

Referencia Local

CONCEPTOS ASOCIADOS CON EL ALCANCE - PYTHON

Uso de palabras claves “**global**” y “**nonlocal**”
Sino daría error en este ejemplo



```
prueba.py - C:/Users/Viviana/Desktop/prueba.py (3.6.5)
File Edit Format Run Options Window Help

x = 200
def uno():
    x = 10
    def dos():
        global x
        x = x + 1
        print(' x en dos ',x)
    def tres():
        nonlocal x
        x = x + 1
        print(' x en tres ',x)

    dos()
    print('x en uno después de llamar a dos ',x)
    tres()
    print('x en uno después de llamar a tres ',x)

uno()
print('x en uno después de llamar a uno ',x)
```

```
Python 3.6.5 Shell
File Edit Shell Debug Options Window Help

===== RESTART: C:/Users/Vivia
na/Desktop/prueba.py =====
x en dos 201
x en uno después de llamar a dos 10
x en tres 11
x en uno después de llamar a tres 11
x en uno después de llamar a uno 201
>>>
```

Ln: 10 Col: 4

Ln: 19 Col: 40

ESPACIOS DE NOMBRES Y ALCANCE

○ Definición:

- Un **espacio de nombres** es una **zona separada abstracta del código** donde se pueden *agrupar, declarar y definir objetos* (**variables, funciones, identificador de tipo, clase, estructura, etc.**)
- Al espacio de nombre se le **asigna un nombre o identificador propio**. Cada lenguaje tiene su regla de nombrar y de delimitar la zona (ej. C++ `_` en nombre y llaves delimitar el bloque)

ESPACIOS DE NOMBRES Y ALCANCE

○ Utilidad:

- Es un **recurso** de los **lenguajes de programación**
- Ayudan a **evitar problemas con identificadores** con el **mismo nombre** en grandes proyectos o cuando se usan bibliotecas externas.
- Son utilizados por los **lenguajes de tipo dinámico**
- Ayudan a resolver el Alcance dentro de ese **espacio de nombres** (Phyton...)

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

Definición: el **tipo de una variable** es una especificación de un **conjunto** de:

- **Valores** a asociar a la variable
 - **Operaciones** permitidas para esas variables
- **Antes** que una **variable** pueda ser **referenciada** debe **ligársele a un tipo**.
 - Una **variable de este tipo** pasa a ser una **Instancia**

Ayuda a:

- **Proteger** a las **variables de operaciones no permitidas**
- **Chequear tipos** verifica el **uso correcto de las variables** (por ej. Cada lenguaje tiene sus reglas de combinaciones de tipos)

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE> EJEMPLO EN ADA



Compile

| Execute

hello.adb x

```
1 with Text_IO, Ada.Integer_Text_IO;
2 use Text_IO, Ada.Integer_Text_IO;
3
4 procedure Principal is
5   y: integer;
6   begin
7
8     y:=7;
9     y:= y + 9.0;
10    Put("    El valor de la variable y es:");
11    Put(y);
12
13 end Principal;
```

- Ada es **fuertemente tipado**, no se pueden **mezclar valores de tipo diferentes**.
- no aplica reglas de conversión implícitas.
- Si se puede aplicar conversiones explícitas entre tipos estrechamente relacionados

ERROR

Terminal

```
gcc -c hello.adb
hello.adb:4:11: warning: file name does not match unit name, should be "principal.adb"
hello.adb:9:09: invalid operand types for operator "+"
hello.adb:9:09: left operand has type "Standard.Integer"
hello.adb:9:09: right operand has type universal real
gnatmake: "hello.adb" compilation error
```

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE> EJEMPLO EN PYTHON

Python 3.8.0 Shell

File Edit Shell Debug Options Window Help

Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

```
>>> curso = 'Curso Nro. '
```

```
>>> c1 = curso + 1
```

ERROR

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

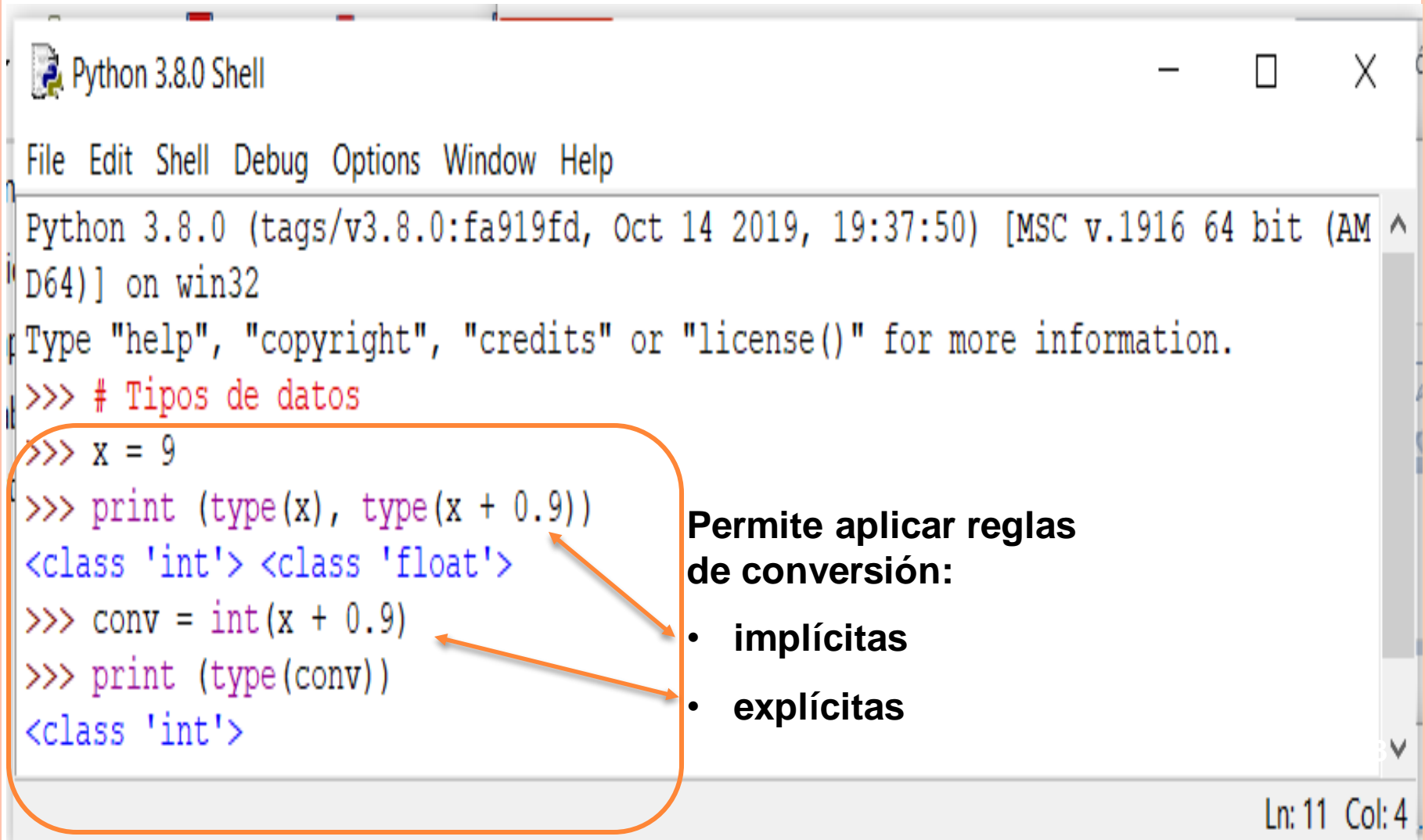
c1 = curso + 1

TypeError: can only concatenate str (not "int") to str

•Python es **fuertemente tipado**, no permite esta operación.

Ln: 9 Col: 4

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE> EJEMPLO EN PHYTON



The image shows a screenshot of a Python 3.8.0 Shell window. The window title is "Python 3.8.0 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area displays the following code and output:

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:37:50) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> # Tipos de datos
>>> x = 9
>>> print (type(x), type(x + 0.9))
<class 'int'> <class 'float'>
>>> conv = int(x + 0.9)
>>> print (type(conv))
<class 'int'>
```

Two orange arrows point from the text "Permite aplicar reglas de conversión:" to the code. One arrow points to the expression `x + 0.9` in the `print` statement, and the other points to the `int` function call in the assignment `conv = int(x + 0.9)`.

Permite aplicar reglas de conversión:

- implícitas
- explícitas

Ln: 11 Col: 4

ATRIBUTOS <NOMBRE, ALCANCE, **TIPO**, L-VALUE, R-VALUE>

CLASES DE TIPO

- **Predefinidos** - por el lenguaje
 - Tipos base definidos en el lenguaje
- **Definidos** - por el usuario
 - Constructores, permiten crear otros tipos
- **TAD** - Tipo Abstracto de **D**atos
 - listas, colas, pilas, arboles, grafos, etc...

Se verán en más detalle en otras clases

ATRIBUTOS <NOMBRE, ALCANCE, TIPO,
L-VALUE, R-VALUE>

○ Tipos Predefinidos:

- Son los tipos base que están **descriptos en la *Definición del Lenguaje*** (enteros, reales, flotantes, booleanos, etc....)
- Cada uno **tiene valores y operaciones**

Tipo boolean

valores: *true, false*

operaciones: *and, or, not*

- Los **valores se ligan en la *Implementación* a representación de máquina**

true string 000000.....1

false string 0000.....000

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

○ Tipos Definidos por el usuario:

- Permiten al programador mediante la declaración de tipos **definir nuevos tipos a partir de los tipos predefinidos y los constructores**

```
Compile | Execute main.pas x
1 Program Principal(output);
2 type t = array [1..10] of integer;
3 var a:t;
4     x: integer;
5 begin
6
7     x:=5;
8     a[1]:=x+1;
9     writeln('Todo paso bien!');
10 end.
```

Ejemplo en Pascal:

type t establece una ligadura (en ***momento de Traducción***) entre el nombre del **tipo t** con el ***arreglo de 10 elementos enteros***. El **tipo t** tiene **todas las operaciones** de la **estructura de datos (arreglo)**, y por lo tanto es posible leer y modificar cada componente de un objeto de tipo *t* indexando dentro del arreglo.

Pascal usa:

= cuando define el tipo
: cuando declara y
:= cuando asigna

ATRIBUTO <NOMBRE, ALCANCE, **TIPO**,
L-VALUE, R-VALUE>

○ **Tipos de Datos Abstractos (TAD):**

- Es una estructura de datos que **representa a un nuevo tipo** con un **nombre que lo identifica**
- Está **compuesto** por una colección de **operaciones definidas (rutinas)**
- **Rutinas** son usadas para **manipular los objetos** de este nuevo tipo

No hay ligadura por defecto, el programador debe especificar la representación y las operaciones

Se verá en más detalle en otra clase!

TIPOS ABSTRACTOS (EJEMPLO EN C++)

*Estructura
interna
(privada)*

*Comportamiento
(operaciones)
(pública)*

*La idea es que vean
que se programan,
no que entiendan el
código*

```
#include<iostream>
#include<process.h>
#include<conio.h>
using namespace std;

class Clistpila
{   protected:
    struct lista    // Estructura del Nodo de una lista
    {   int dato;
        struct lista *nextPtr;           //siguiente elemento de la lista
    };

    typedef struct lista *NODELISTA;    //tipo de dato *NODOLISTA

    struct NodoPila
    {   NODELISTA startPtr;           //tendrá la dirección del fondo de la pila
    } pila;

    typedef struct NodoPila *STACKNODE;    //Tipo Apuntador a la pila

public:
    Clistpila( );           // Constructor
    ~Clistpila( );          // Destructor

    void push(int newvalue); // Función que agrega un elemento a la pila
    int pop( );             // Función que saca un elemento de la pila
    int PilaVacia( );       // Verifica si la pila está vacía
    void MostrarPila( );    // Muestra los elementos de la Pila

    friend void opciones(void); // función amiga
};

//Funciones Miembro de la clase
Clistpila :: Clistpila( )
{   pila.startPtr = NULL;           //se inicializa el fondo de la pila.
}

int Clistpila :: PilaVacia( )
{   return((pila.startPtr == NULL)? 1:0); //note que si la pila esta vacía retorna 1, sino 0
}

void Clistpila :: push(int newvalue)           //se puede insertar en cualquier momento
{   NODELISTA nuevoNodo;                       //un nodo al tope de la pila
    nuevoNodo = new lista;                     //crear el nuevo nodo
    if(nuevoNodo != NULL)                      //si el espacio es disponible
```

ATRIBUTO <NOMBRE, ALCANCE, **TIPO**,
L-VALUE, R-VALUE>

o **Momentos de ligadura**

- **Estático**
- **Dinámico**

ATRIBUTO <NOMBRE, ALCANCE, **TIPO**,
L-VALUE, R-VALUE>

Momentos de ligadura - Estático

- **El tipo se liga en compilación y no puede ser cambiado**
 - **El chequeo de tipo también será estático**
 - **La ligadura puede ser realizada en forma:**
 - **Explícita**
 - **Implícita**
 - **Inferida**

Fortran, Pascal, Algol, Simula, ADA, C, C++,
Java, etc

ATRIBUTOS <NOMBRE, ALCANCE, **TIPO**, L-VALUE, R-VALUE> TIPO DE DECLARACIÓN

○ Momento Estático – **Explícito**

- La **ligadura** se establece mediante una declaración

```
int x, y;  
bool z;
```

○ Momento Estático - **Implícito**

- ☐ Si **no** fue **declarada** la **ligadura** se **deduce por reglas** propias del lenguaje

Ej. Fortran:

variables que empiezan con letra de **I a N** son **enteras**
variables que empiezan con el **resto de las letras** son **reales**

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE> TIPO DE DECLARACIÓN

○ Momento Estático - Inferido

- El **tipo** de una expresión **se deduce de los tipos de sus componentes**.
- Aplica en general a Lenguajes Funcionales.

Ejemplos:

If $x = 0$ puede inferir que x es entero

En Lisp: $\text{doble } x = 2 * x$ (script que calcula el doble de x , se infiere el tipo de x)

En Swift: $\text{var nombreCliente} = \text{"Pedro"}$ (la infiere string)

Si no está definido el tipo se infiere por reglas

$\text{doble} :: \text{num} \rightarrow \text{num}$

$\text{doble} :: \text{int} \rightarrow \text{int}$

$\text{verPersona} :: \text{Persona} \rightarrow \text{String}$

$\text{not} :: \text{Bool} \rightarrow \text{Bool}$

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

○ Momento – **Dinámico**

- **El tipo se liga en ejecución y puede cambiarse.**
- **Cambia cuando se le asigna un valor mediante una sentencia de asignación.**
 - **Mas flexible:** programación genérica. Permite cosas de este tipo: ej ADA
 - lista ← 3.5 8.3 0.7 10.1 (tipo: lista Reales long. 4)
 - lista ← 15 (tipo: variable entera)
 - **No se detectan incorrecciones de tipo en las asignaciones.** El tipo de la parte izquierda simplemente se **cambia al tipo de la derecha**

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

○ Momento – **Dinámico**

- El **coste de implementación** de la ligadura dinámica de atributos es **mayor**, sobre todo en **tiempo de ejecución** (*comprobación* de tipos, *mantenimiento del Descriptor* asociado a cada variable en el que se almacena el tipo actual, *cambio en el tamaño de la memoria* asociada a la variable, etc.)
- **Chequeo dinámico**
- **Menor legibilidad**

*Los lenguajes interpretados en general
adoptan ligadura dinámica de tipos*

APL, Snobol, Javascript, Python, Ruby, etc

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, **L-VALUE**, R-VALUE>

- Hay un área de memoria donde se alojan las variables. El área de memoria debe ser ligada a la variable en algún momento.

L-VALUE de una variable:

es el área de memoria ligada a la variable durante la ejecución. Las instrucciones de un programa acceden a la variable por su L-Valor.

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

- **Tiempo de vida** (lifetime) o extensión:

Periodo de tiempo que existe la ligadura

- **Alocación:**

Momento en que se reserva la memoria
para una variable

**El tiempo de vida es el tiempo en que la variable
está alocada en memoria
(alocación y desalocación)**

<NOMBRE, ALCANCE, TIPO, L-VALUE,
R-VALUE>

Momentos - **Alocación**

La **alocación depende de los lenguajes** y encontramos estos **tipos**:

- **Estática**
- **Dinámica**
- **Persistente**

<NOMBRE, ALCANCE, TIPO, L-VALUE,
R-VALUE>

Momentos - **Alocación**

- **Estática:** sensible a la historia, se hace **en compilación (*antes de la ejecución*)** cuando se carga el programa en memoria en zona de datos y **perdura hasta fin de la ejecución.**
- **Dinámica:** se hace **en tiempo de ejecución.**
 - **Automática:** cuando **aparece una declaración al ejecutarse**
 - **Explícita:** requerida **por el programador con la creación de una sentencia**, a través de algún constructor (por ej. algún puntero)

<NOMBRE, ALCANCE, TIPO, L-VALUE,
R-VALUE>

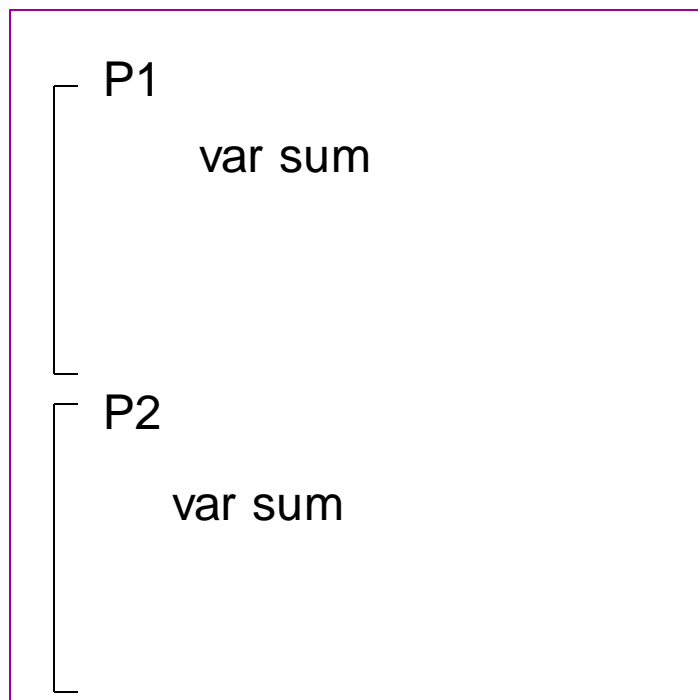
Momentos - **Alocación**

- **Persistente:** Los objetos persistentes **existen en el entorno en el cual un programa es ejecutado**, su **tiempo de vida no tiene relación con el tiempo de ejecución** del programa. Persisten más allá de la memoria.
 - Ejemplo: **archivos** una vez creados permanecen y pueden ser usados en diversas activaciones hasta que son borrados con un comando del sistema operativo.
 - Lo mismo sucede con **base de datos**

Esto se verá más adelante en más detalle

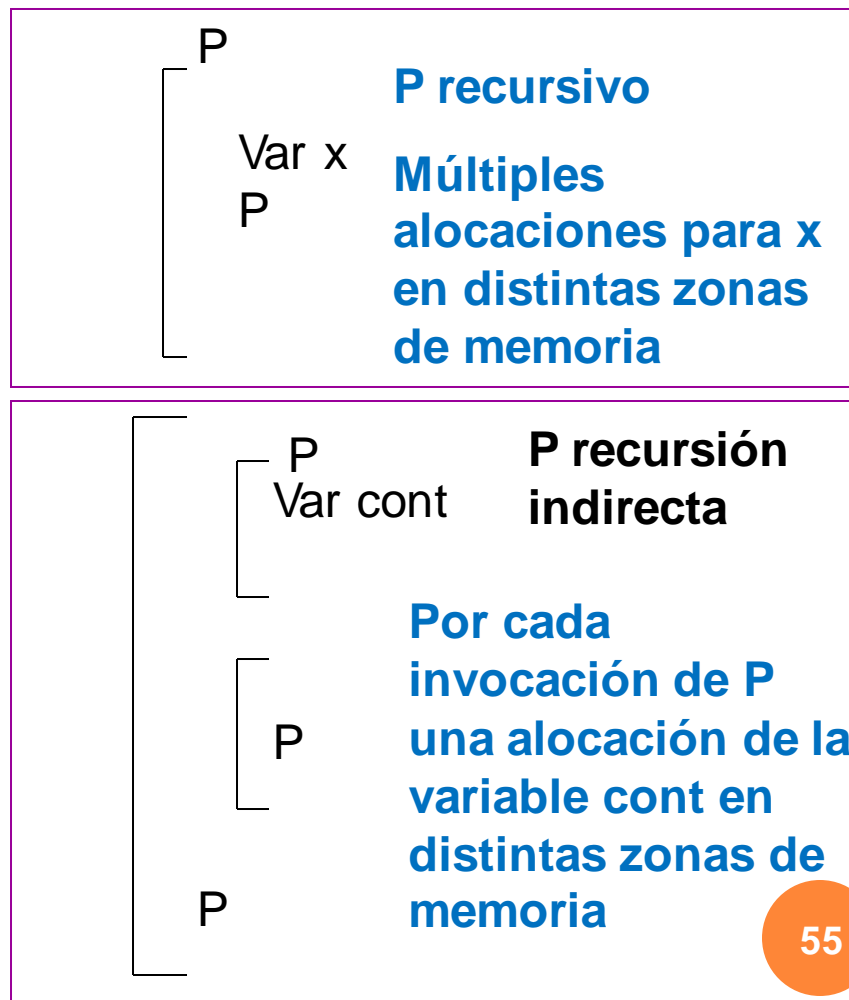
EJEMPLOS DE ALOCACIONES

recursión



Dos allocaciones diferentes para sum en distintas zonas de memoria:

- sum de P1, se aloca y luego muere
- sum de P2, se aloca y luego muere



cada uno tiene su propio tiempo de vida

EJEMPLOS DE ALOCACIONES

File 1

```
int x;
```

```
static int n;
```

```
int func1()
```

```
{
```

```
    extern int i;
```

```
    float z;
```

```
    static int m;
```

```
}
```

```
..
```

```
int main()
```

```
{
```

```
    .....
```

```
}
```

Para hacer uso de la cláusula **EXTERN**, la variable debe estar definida previamente y debe ser externa (con Z no se puede)

File 2

```
extern int x;
```

```
....
```

```
int i;
```

```
extern static int n;
```

```
float func2()
```

```
{
```

```
    int x
```

```
    int z;
```

```
}
```

```
.....
```

Alcance:

Con la declaración “**extern int x**” se extendió el **alcance** de x a File 2.

Ahora x de File 1 tiene **alcance** en todo File 1 y en File 2, MENOS en func2 (tiene int x)

La nueva declaración “**extern int i**” extiende el **alcance** de i del File 2 a la función func1 de File 1

Static m: Tiempo de vida: Todo el programa, lo fija al cargar

Static m: Alcance: SOLO en la función dónde está definida

Si está definida fuera de la función el alcance es desde dónde está al final del archivo. NO se le puede extender el alcance hacia otro archivo

Tiempo de vida:

Desde que comienza el programa hasta que termina

ATRIBUTOS <NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

- **R-Valor** de una variable es el **valor codificado almacenado en la locación asociada a la variable (l-valor)**
- La codificación se **interpreta** de acuerdo al **tipo de la variable**
- **Objeto: (l-valor, r-valor) (dirección memoria, valor)**

$x := x + 1$ sentencia asignación

l-valor r-valor

Se **accede** a la variable por el **l-valor (ubicación)**

Se puede modificar el **r-value (valor) (no en constantes)**

<NOMBRE, ALCANCE, TIPO, L-VALUE,
R-VALUE>

Momentos de ligadura:

Dinámico: más común en lenguajes imperativos (Fortran, C, Pascal, ADA). Permite cambiar el *r-valor*

- **$b := a$** se copia el *r-valor* de **a** en el *l-valor* de **b** y así **cambia** el *r-valor* de **b**
- **$a := 17$**
- **Constantes** se congela el valor (no puede ser cambiado)

Depende de cada lenguaje

<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE> EJEMPLO EN ADA

Const pi = 3.1416

No da
error toma
x=4

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Inicializacion is
3      x: Integer:=4;
4      procedure Uno is
5          z: constant Integer := x+5;
6      begin
7          Put_Line("Estoy en uno");
8      end Uno;
9  begin
10     Uno;
11 end Inicializacion;
```

<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE> EJEMPLO EN PASCAL

main.pas

```
1 {
2     Online Pascal Compiler.
3     Code, Compile, Run and Debug Pascal program online.
4 }
5 program Constantes;
6 var
7     i: integer;
8     function prueba(): integer;
9     const x: integer = 9 + i;
10    begin
11        prueba := x;
12    end;
13 begin
14     writeln('Variables constantes');
15     i := 1;
16
17     writeln('El valor retornado más el valor de i es: ', prueba() + i);
18 end.
```

se intenta inicializar una constante con el valor de una variable en una expresión y da error!

El binding del r-valor es en compilación y no puede obtenerlo hasta runtime

Esta expresión no está permitida.

input

stderr

Compilation failed due to following error(s).

```
Free Pascal Compiler version 2.6.2-8 [2014/01/22] for x86_64
Copyright (c) 1993-2012 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling main.pas
```

main.pas(8,28) Error: Illegal expression

main.pas(20) Fatal: there were 1 errors compiling module, stopping

Fatal: Compilation aborted

Error: /usr/bin/ppcx64 returned an error exitcode (normal if you did not specify a source file to be compiled)

<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

main.c

```
1  /*****
2
3      Online C Compiler.
4      Code, Compile, Run and Debug C program online.
5      Write your code in this editor and press "Run" button to compile and execute it.
6
7  *****/
8
9  #include <stdio.h>
10 i=4;
11
12 void prueba()
13 { const int k= 1 + i; ←
14   printf("%d",k);
15 }
16
17 int main()
18 {
19     printf("Prueba constantes\n");
20     i= 8;
21     prueba();
22     return 0;
23 }
```

¿Qué pasa con C? La ligadura de su r-valor con la variable la hace en tiempos de ejecución y no da error

Esta expresión es permitida

```
main.c:10:1: warning: data definition has
main.c:10:1: warning: type defaults to 'in
Prueba constantes
9
...Program finished with exit code 0
Press ENTER to exit console.
```

<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

Inicialización de una variable

○ ¿Cuál es el r-valor luego de crearse la variable?

- Estrategia de inicialización:

- Inicialización por defecto:

- Enteros se inicializan en 0
 - Caracteres en blanco
 - Funciones en VOID, etc.

- Inicialización en la declaración:

C `int i =0, j= 1` **ADA** `I,J INTEGER:=0`

62

Opcionales

<NOMBRE, ALCANCE, TIPO, L-VALUE, R-VALUE>

Inicialización de una variable

○ ¿Qué pasa si no es inicializada?

- Los lenguajes y las diferentes versiones lo **implementan de diversas formas**, se suelen **producir errores** y más con cambio de plataformas
- El más común es **Ignorar el problema** tomando lo que haya en memoria. (el string de bits asociados al área de memoria)
- Puede llevar a errores y requiere chequeos adicionales.

VARIABLES ANÓNIMAS (NO NOMBRADAS) Y REFERENCIAS

main.pas

```
1 {  
2  
3      Online Pascal Compiler.  
4      Code, Compile, Run and Debug Pascal program online.  
5      Write your code in this editor and press "Run" button to execute it.  
6  
7 }  
8  
9  
10 program Hello;  
11 type  
12     pi= ^integer;  
13 var  
14     punt: pi;  
15     i: integer;  
16 begin  
17     writeln ('Variables anónimas');  
18     i:= 1;  
19     new(punt);  
20     punt^:= 7;  
21  
22     writeln ('El valor de las variables son:',i, punt^);  
23 end.
```

¿En qué se diferencian
esas dos variables?

El contenido de la
variable referenciada por
el puntero se denota:
punt^

VARIABLES ANÓNIMAS (NO NOMBRADAS) Y REFERENCIAS

- Algunos lenguajes permiten que **variables sin nombre** sean **accedidas** por el **r-valor de otra variable**.
- Ese r-valor se denomina *referencia* o *puntero*
- La *referencia* puede ser el r-valor de una **variable referenciada** llamada *acces path*

VARIABLES ANÓNIMAS (NO NOMBRADAS) Y REFERENCIAS

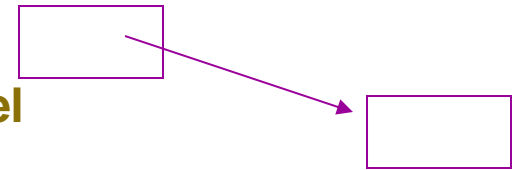
- Algunos lenguajes permiten que el r-valor de una variable sea una referencia al l-valor de otra variable

Puntero a entero

type pi = ^ integer; instancia

var pxi : pi Aloca variable anónima setea el puntero

new (pxi)



type ppi = ^ppi;

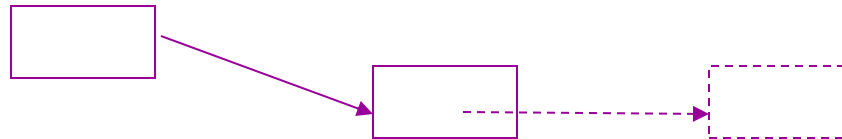
var ppxi: ppi;

....

new(ppxi);

^ppxi: ppi;

Puntero a un puntero



ALIAS

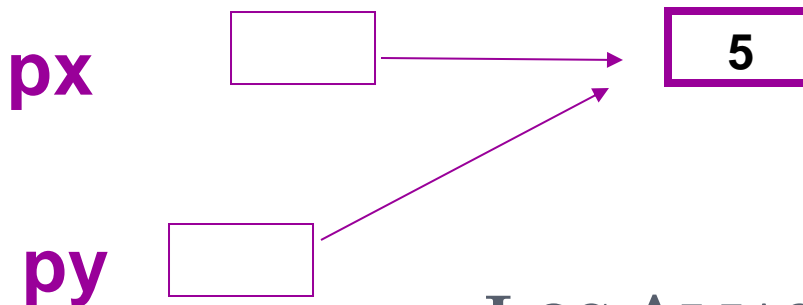
- Dos variables **comparten un objeto**, si sus **caminos de acceso conducen al mismo objeto**.
- Un **objeto compartido modificado vía un camino se modifica para todos los caminos**

- `int x = 5;`

- `int*px,`

- `px = &x ;`

- `py =px`



**LOS ALIAS PUEDEN
TRAER PROBLEMAS. SE
RECOMIENDA NO SE USEN**

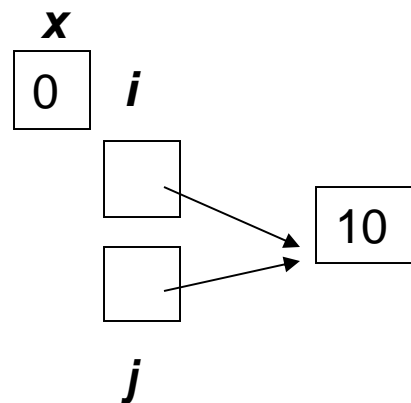
ALIAS

Alias: Dos nombres que denotan la misma entidad (objeto) en el mismo punto de un programa.

distintos nombres \longrightarrow 1 entidad

- Dos variables son **alias** si **comparten el mismo objeto** de dato en el mismo **ambiente de referencia**.
El uso de alias puede llevar a programas de **difícil lectura y a errores**.

```
int x = 0;  
int *i = &x;  
int *j = &x;
```



```
*i = 10;
```

Efecto colateral: modificación de una variable no local

CONCEPTO DE SOBRECARGA Y ALIAS

- **Alias**

distintos nombres \longrightarrow 1 entidad

- **Sobrecarga**

1 nombre \longrightarrow distintas entidades

CONCEPTO DE SOBRECARGA

Sobrecarga - Un **nombre** esta **sobrecargado** si en un momento referencia **más de una entidad**

- Tiene que haber **suficiente información** para permitir establecer la **ligadura unívocamente**.
- Debe estar permitido por el lenguaje

por ejemplo:

- **un operador** que tenga **distintas funciones**
- **funciones con igual nombre** que **hagan cosas distintas**

1 nombre → distintas entidades

CONCEPTO DE SOBRECARGA


```
int i,j,k;  
float a,b,c;  
.....  
i = j + k ;  
a = b + c;
```

¿Qué sucede con el operador "MAS"?..

En un caso suma enteros y en otro flotantes

O hasta concatenar string

Los tipos permiten que se desambigüe en compilación.

1 nombre  1 entidad
No hay ambigüedad