

SEMANTICA OPERACIONAL

ENTIDADES CON LAS QUE TRABAJAN LOS PROGRAMAS

1

SEMÁNTICA OPERACIONAL

Permite:

- Describir el significado preciso de un programa
- Verificar el resultado final de la ejecución de un programa.

La semántica operacional es fundamental para diversos aspectos del proceso de **desarrollo de software**, como el **diseño de lenguajes de programación**, la **verificación de programas** y la **comprensión de cómo se ejecutan los programas** en un nivel más bajo.

SEMÁNTICA DE LOS LENGUAJES DE PROGRAMACIÓN

```
p014estructuras.rb
var = 5
if var > 4
  puts "La variable es mayor que 4"
  puts "Puedo tener muchas declaraciones a"
  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"
end

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

Ruby

```
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.
```

```
#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;
}
```

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

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)
```

¿Qué elementos encontramos?

¿Qué ENTIDADES principales hay?

SEMÁNTICA DE LOS LENGUAJES DE PROGRAMACIÓN

| ENTIDADES | ATRIBUTOS |
|------------------------------|-------------------------------------------------------------------------------------------------|
| Variables | Nombre, Tipo, rango de valores, área de memoria, etc. |
| Rutinas/ subprogramas | Nombre, parámetros formales y reales, convención de pasaje de parámetros, área de memoria, etc. |
| Sentencias | acción asociada |

Repositorio DESCRIPTOR:

Lugar donde se almacena la información de estos atributos. Se va completando en compilación, ejecución

CONCEPTO DE LIGADURA (BINDING)

Hay que **asociar** cada **entidad** a sus **atributos**.

Es un **concepto central** en la **definición** de la **semántica** de los lenguajes de programación

*Por ejemplo: **int a;***

***Entidad:** Variable*

Atributos:

***Nombre:** a **Tipo:** *int* entero*

***rango de valores y operaciones:** determinado por el tipo*

CONCEPTO DE LIGADURA (BINDING)

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



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



Estos **atributos** tienen que **establecerse**
(**tener un valor**) antes de poder **usar la**
entidad



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

LIGADURA

Hay diferencias entre los lenguajes de programación en:

- El **número de entidades**
- El **número de atributos** que se les pueden ligar
- El **momento** de la **ligadura** cuando toma el valor (binding time).
- La **estabilidad** de la ligadura:
 - ✓ una vez establecida **se puede modificar? o queda fija?** Por ej. Se puede cambiar el tipo de una variable que se asigno en compilación en tiempo de ejecución?

MOMENTO Y ESTABILIDAD DE LA LIGADURA

○ Ligadura es Estática

1. Se establece antes de la ejecución.
2. No se puede modificar.

El termino estática referencia al **binding time (1)** y a su **estabilidad (2)**.

○ Ligadura es Dinámica

1. Se establece durante la ejecución
 2. Si puede modificarse durante ejecución de acuerdo a alguna **regla específica del lenguaje**.
- Excepción: constantes (el binding es en ejecución/runtime pero no puede ser modificado luego de establecido)

MOMENTO DE LIGADURA

Los **ATRIBUTOS** pueden **ligarse** en el momento:

- Definición del lenguaje
- Implementación del lenguaje
- Compilación/traducción
- Ejecución

Veamos el siguiente ejemplo

ESTÁTICO
DINÁMICO



MOMENTO DE LIGADURA Y ESTABILIDAD

En Definición del lenguaje

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

En Implementación

- Set de valores y su representación numérica
- sus operaciones

En Compilación

- Asignación/redefinición del tipo a las variables

Ejemplo en lenguaje C

int

Define los tipos permitidos, como se escriben, nombran, y los vincula a operaciones algebraicas

Int

- Vincula un tipo de variable a su representación en memoria, y determina el set de valores que están contenidos en el tipo.

int a

- Liga tipo a la variable(atributo)
- cambia el tipo en compilación si está permitido (ej. Caso Pascal)

MOMENTO Y ESTABILIDAD – CONTINUACIÓN

Ejemplo en lenguaje C

○ En Ejecución

- Variables se **enlazan** con sus **valores**
- Variables se **enlazan** con su **lugar de almacenamiento**

int a

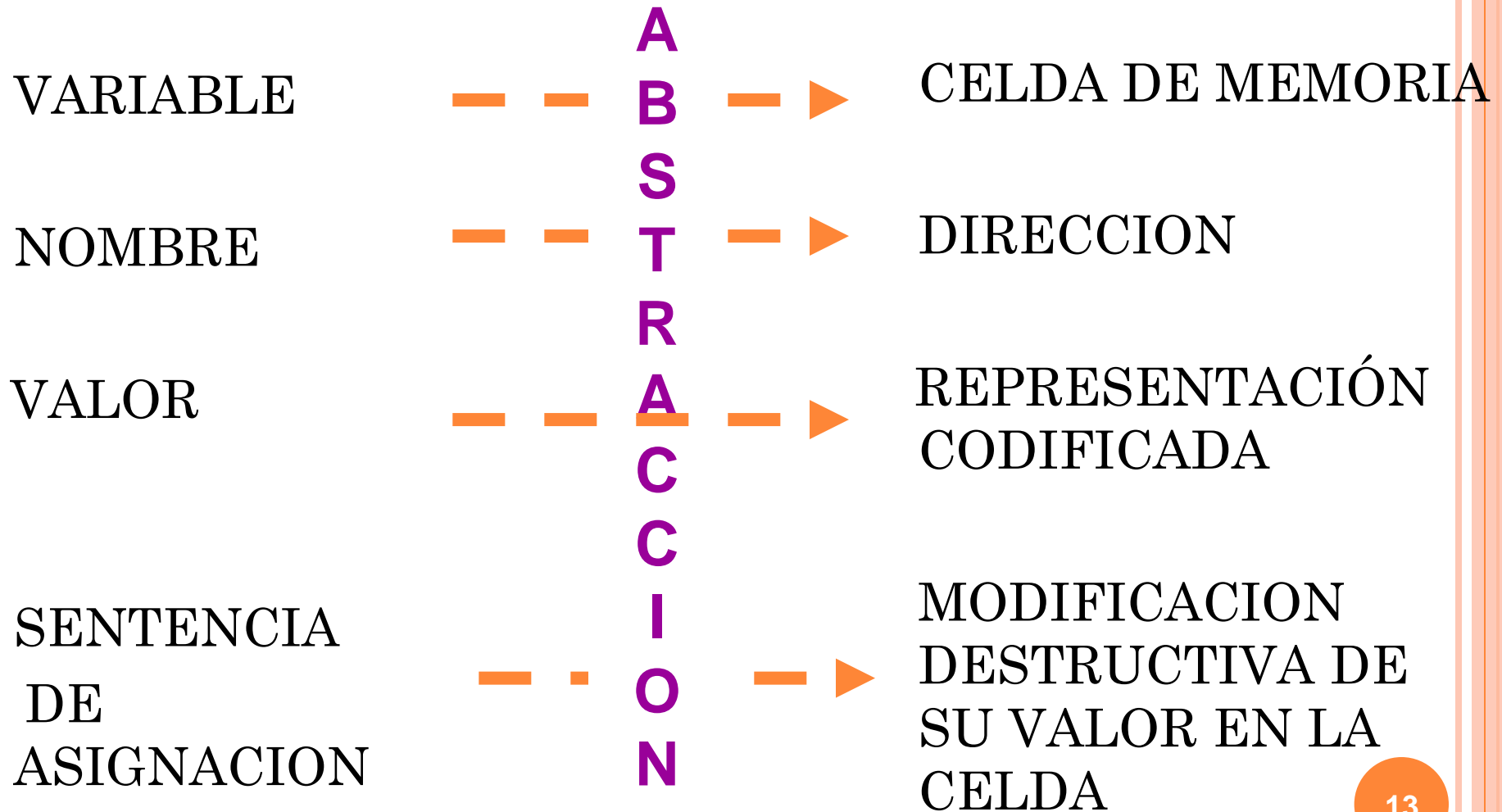
a=10

a=15

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

SEMANTICA OPERACIONAL VARIABLES

VARIABLE



VARIABLES

CONCEPTO

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

¿esto sólo me da alguna información?

¿Es x la misma variable?

¿Qué dispara esa sentencia?

¿Hay algún error? ¿Está permitido?

¿tiene valor asignado y?

¿Qué lenguaje es?

No tenemos información, debo ver el contexto,

Ver donde está y si es visible, etc.

**Necesitamos más información
para decidir: Atributos**

VARIABLES

CONCEPTOS

Atributos de una variable

1. Nombre
2. Alcance
3. Tipo
4. l-valor
5. r-valor

Una Variable es una 5-TUPLA

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 conjunto de 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 que hay que conocer y validar:

- El nombre es introducido por una sentencia de declaración
- Longitud máxima según lenguaje (se define en la etapa de definición del lenguaje)

| | |
|---------------------------------------|---------------------------------------------------------------------------------------|
| Fortran | 6 caracteres |
| C | depende del compilador suele ser de 32 caracteres y se ignora el resto |
| 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 | — Permitido en el nombre |
| Ruby | <ul style="list-style-type: none">• solo letras minúsculas para variables locales• \$ para comenzar nombres de variables globales |

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

Aspectos de diseño del nombre:

- *Sensitivos a mayúsculas*

Sum = sum = SUM ?

- C, C++, Java y Python no es lo mismo escribir un nombre en mayúsculas que en minúsculas, es sensible a mayúsculas
- Pascal no sensible a mayúsculas y minúsculas
- *palabra reservada - palabra clave:*
 - **palabra clave: palabras propias del lenguaje** tienen un significado especial solo en contextos particulares y se pueden utilizar como identificadores en otros contextos
 - **palabra reservada** es aquella **palabra clave** que **no puedo utilizar** para asignar a un **identificador**, depende de cada **lenguaje** (ej. En Java **const** y **goto** no tienen significado, pero no pueden ser usadas como identificadores).

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

**Va a depender de cada tipo de
lenguaje, de las reglas de su diseño e
implementación, de las diferencias que
puedan surgir en un cambio de versión
(actualización) del lenguaje,
para que no nos lleve a errores**

Todos estos temas los vieron en Sintaxis

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

- El alcance de una **variable** es el **rango de instrucciones** en el que **es conocido el nombre de la variable**. (visibilidad)
- Las **instrucciones** del programa pueden **manipular las variables a través de su nombre dentro de su alcance**.
- **Afuera de ese alcance son invisibles**

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

REGLAS PARA LIGAR UNA VARIABLE A SU ALCANCE

Los diferentes **lenguajes** adoptan **diferentes reglas** para **ligar el nombre de una variable** a su **alcance**

- 1. LIGADURA POR ALCANCE ESTÁTICO**
- 2. LIGADURA POR ALCANCE DINÁMICO**

Se usan cuando aparece referenciada en el código una variable que no es local, para saber a donde va a buscarla y saber a quien pertenece

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 de variables** (referencia explícita o implícita) **examinando el texto del programa, *sin necesidad de ejecutarlo***.

Si una unidad necesita referenciar una variable que no fue declarada en su unidad, debe buscarla siguiendo la estructura del programa según donde está contenida la unidad pasando por las estructuras más externas (sino encuentra da error)

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 **donde es conocida** 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 de la variable (con el mismo nombre)** sea encontrada **durante la ejecución**.

Si una unidad necesita referenciar una variable que no fue declarada en su unidad, debe buscarla hacia afuera siguiendo la ejecución y viendo quien llamó a la unidad que la necesita

Usada por algunos lenguajes: **APL**, **Lisp** (original), **Afnix** (llamado *Aleph* hasta el 2003), **TCL** (Tool Command Language), **Perl**, **Snobol4**

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

Veamos algunos ejemplos de
alcance en distintos lenguajes

EJEMPLO DE ALCANCE LENGUAJE C - LIKE

```
int x;
```

```
{
```

```
/*bloque A*/
```

```
int x;
```

```
....
```

```
}
```

```
{
```

```
/*bloque B*/
```

```
int x;
```

```
....
```

```
}
```

```
{
```

```
/*bloque C*/
```

```
x = ...;
```

```
...
```

```
}
```

```
....
```

Ejecución:

Supongamos que el bloque C puede ser llamado por A o por B

○ ligadura con alcance Dinámico

Nos preguntamos ¿quién lo llamó?

1. Si A llama a C: Toma x de A
2. Si B llama a C: Toma x de B

Dependerá del flujo

Es más difícil de seguir

○ ligadura con alcance Estático

Nos Preguntamos ¿Dónde está contenida?

1. en ambos casos da lo mismo x del bloque Principal

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.
```

Pascal like: NO es PASCAL (es estático)

Ejecución:

Alcance estático:

- Al invocar a *tres()*:
- Se invoca a **uno()**
- Se invoca a **dos()** y
 z:= a + 1+ b;
 z de Alcance
- variable **a** es de **Alcance** (**a=4**)
- variable **b** es de **uno()** (**b=20**)
- **Resultado Z=25**

Alcance dinámico:

- Al invocar a *tres()*:
- Se invoca a **uno()**
- Se invoca a **dos()** y
 z:= a + 1+ b;
 z de Alcance
- variable **a** es de **tres()** (**a=20**)
- variable **b** es de **uno()** (**b=20**)
- **Resultado Z=41**

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
```

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

X Y X'

Barras y llaves - Una forma práctica de visualizar dónde están contenidas las variables

El alcance se extiende desde su declaración hacia los bloques anidados a menos que aparezca otra declaración para la variable con = 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;
```

Diagram illustrating static scope resolution for the variable `x`:

- 1) X=1, Y=1**: Initial values in the main program scope.
- 2) X=2**: Value of `x` after the first call to `uno()` from the main program.
- 2) X=20**: Value of `x` after the call to `tres()` from the main program.
- 1) X=10**: Value of `x` after the call to `tres()` from the `dos` procedure.

Result

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

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 begin
17   y:=7;
18   Prueba;
19
20 end Principal;
```

NO da error, 3+y se
resuelve en ejecución en
ADA

X Y

Y'

el alcance es **estático**
pero las **reglas** de ADA
dicen que es **desde**
dónde se declara
hacia abajo! No desde
el inicio del bloque

- por eso toma **Y=7** de Principal para calcular $x=3+y$ da **X=10**
- Luego pone **y=4**

```
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
```

El **alcance es estático**: Por más que alcance1 se llame desde alcance2 la **variable X** tomará su valor **del programa principal**.

1. No es necesario declarar variables antes de usarlas.
2. Las variables se crean automáticamente cuando se les asigna un valor por primera vez.

En Python (no confundir)
Alcance estático con

- Tipado dinámico: El tipo de una variable *se infiere* automáticamente en función del **valor** que se le asigna.
- Fuertemente tipado: no permite operaciones **sobre** tipos distintos, *se deben convertir antes!*

```
C:\Windows\system32\cmd.exe
```

```
hola Juan
```

```
Presione una tecla para continuar . . .
```

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:

- **Menos utilizadas** por los LP
- **Más fáciles de implementar**
- **Poco claras** en cuanto a la programación y **poco eficientes en la implementación. Encontrar una declaración de una variable en el flujo de ejecución puede ser duro. El código se hace más difícil de leer y seguir**, sobre todo en **grandes programas** con cientos de sentencias es complejo.

CONCEPTOS ASOCIADOS CON EL ALCANCE

CLASIFICACIÓN DE VARIABLES POR SU ALCANCE

1. **Global:** Son todas las **referencias a variables creadas en el Programa Principal**.
2. **Local:** Son todas las **referencias a variables creadas dentro de una Unidad** (programa o subprograma).
3. **No Local:** Son todas las **referencias a variables que se utilizan dentro de subprograma pero que no han sido creadas en la unidad**. (son externas al subprograma)

CONCEPTOS ASOCIADOS CON EL ALCANCE - 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; Declaración de x y
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; Declaración de x'
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.
35
```

X Y Global - son de Alcance (el principal)

X No Local – es de dos()
Y Global – es de Alcance (el principal)

X Local - es de Dos()

CONCEPTOS ASOCIADOS CON EL ALCANCE – PYTHON ALCANCE ESTÁTICO

Hay que conocer las reglas de cada lenguaje!!

- ✓ En Python las variables se crean al darles un nombre y asignarles un valor.
- ✓ $x=x+1$ daría error
- ✓ Uso de palabras claves “**global**” y “**nonlocal**”. Sino daría error $x=x+1$

```
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/Viviana/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
>>>
```

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

Se define el tipo de una variable cómo la especificación del:

- 1. conjunto de valores posibles que se pueden asociar a la variable y del**
- 2. conjunto de operaciones permitidas sobre ellas(crear, acceder, modificar).**

✓ **Antes de referenciar una variable hay que ligarle el tipo**

✓ **Una variable de un tipo dado es una Instancia de ese tipo**

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

Ligar el tipo (conjunto de valores posibles y conjunto de operaciones asociadas) **ayuda a:**

- **proteger a las variables de operaciones no permitidas**
 - **Realizar chequeo de tipos**
 - **Verificar el uso correcto de las variables (ej.. Cada lenguaje tiene sus reglas de *combinaciones de tipos*)**
- ✓ **Ayuda a que el compilador o intérprete detecte errores en forma temprana y a dar confiabilidad del código.**

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

REGLAS DE TIPO

Hay que conocer la reglas del lenguaje!!

```
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;
```

ERROR – No soportado

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
```

- ADA es fuertemente tipado (NO se puede mezclar valores de tipo diferente)
- NO aplica reglas de conversión implícita
- NO tiene reglas de compatibilidad de tipo
- SI se pueden aplicar reglas de conversión explícita entre tipos relacionados aplicando funciones de conversión

Ejemplo ADA

X : Float;

Y : Integer := 7;

...

X := Float(Y);

Asigna a X el valor REAL 7.0

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

REGLAS DE TIPO

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 operaciones entre variables de tipo diferente)
- Pero tiene Reglas de Conversión

Ln: 9 Col: 4

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

REGLAS DE TIPO

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.

>>> # Tipos de datos

>>> x = 9

>>> print (type(x), type(x + 0.9))
<class 'int'> <class 'float'> **Imprime**

>>> conv = int(x + 0.9)

>>> print (type(conv))
<class 'int'> **Imprime**

Python permite aplicar reglas de conversión:

- **Implícitas:** SI. Promoción de Tipo En operaciones aritméticas, promueve a un tipo superior
- **Explícitas:** SI. funciones de conversión de tipo explícitas como `int()`, `float()`, `str()`, `list()`, etc..

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

CLASES DE TIPO

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

Dependerá de cada lenguaje

Se verán en más detalle en otras clase

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 asociadas**
 - ✓ **Tipo boolean**
 - ✓ **valores: true, false**
 - ✓ **operaciones: and, or , not**
- Los **valores se ligan en la Implementación a representación de máquina** según la arquitectura

true string 000000.....1

false string 0000.....000

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

○ Tipos Definidos por el Usuario:

- Permiten al programador con la declaración de tipos definir nuevos tipos a partir de los tipos base y constructores predefinidos.
- Esa declaración establece el binding en tiempo de traducción y heredan todas las operaciones de la representación de la estructura de datos base

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

○ Tipos Definidos por el Usuario:

- Permite al programador **crear abstracciones, encapsular la lógica y datos, reutilizar código** y mejorar la **claridad y legibilidad** del código.
- **Fundamentales** para el desarrollo de **programas complejos** y para mantener un **código organizado y mantenible**.

**Hay que saber que hace cada lenguaje,
que definen en su etapa de diseño
y cómo lo hacen en la etapa de implementan**

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

○ Tipos Definidos por el Usuario:

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

int main() {
typedef int TipoVectorEnteros [4];
TipoVectorEnteros numeroDeCoches;

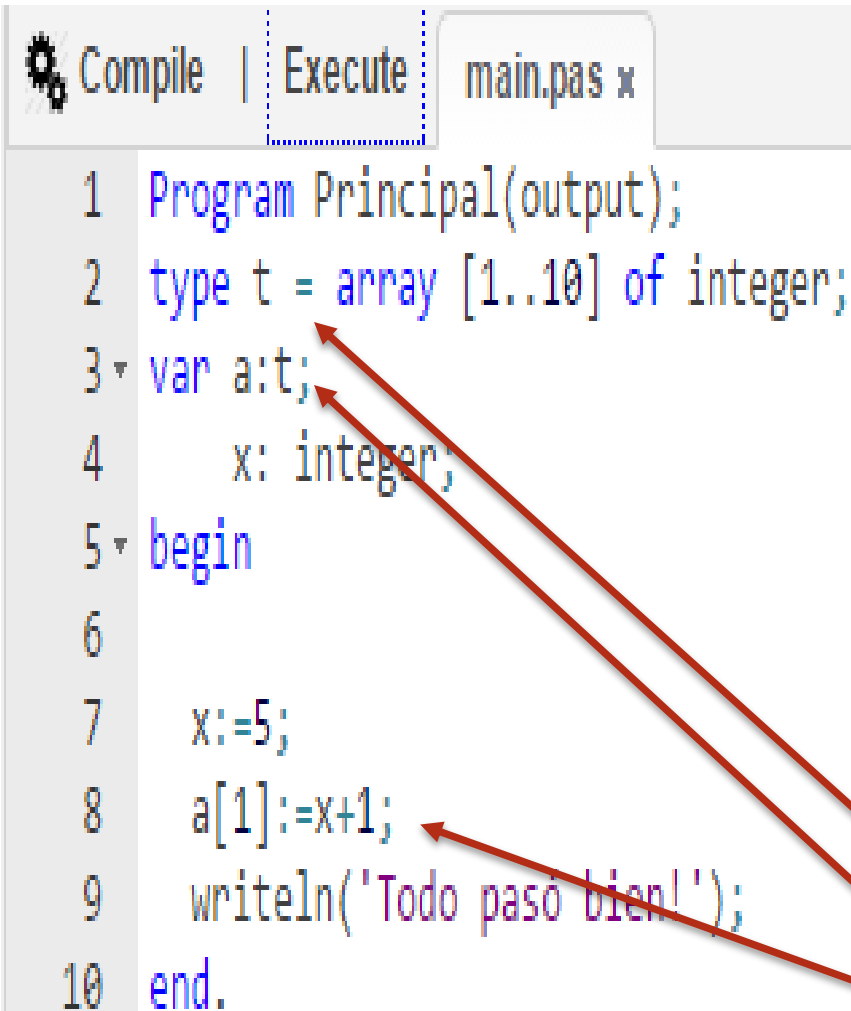
numeroDeCoches[0] = 32;
numeroDeCoches[1]=0;
numeroDeCoches[2]=0;
numeroDeCoches[3]=0;
printf ("El numero de coches en la hora cero fue %d \n", numeroDeCoches[0]);
printf ("El numero de coches en la hora uno fue %d \n", numeroDeCoches[1]);
printf ("El numero de coches en la hora dos fue %d \n", numeroDeCoches[2]);
printf ("El numero de coches en la hora tres fue %d \n", numeroDeCoches[3]);
return 0;
}
```

C

- **Define nuevo tipo**
- **Liga el typename en tiempo de traducción a su implementacion** (que es un vector de 4 elementos enteros consecutivos accesibles por un indice de 0 a 3),
- **heredan** todas las **operaciones** de la representación de la estructura de datos “array” (que le permiten leer, modificar cada element por su indice)

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

Tipos Definidos por el usuario:



```
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 pasó bien!');
10 end.
```

PASCAL:

- **type** *t* la ligadura es 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 sintaxis:

- = para **definir** el tipo
- : para **declarar**
- := para **asignar**

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

• Tipos de Datos Abstractos (TAD):

Abstracto porque quien lo utiliza no necesita conocer los detalles de la **representación interna ni cómo están implementadas las operaciones**. Hay ocultamiento

- No todos los lenguajes soportan la **implementación de un “tipo” definido por el usuario llamado “Tipo de Datos Abstracto”**
- Se debe asignar un **nombre que lo identifique**
- Se realiza la **asociación del nuevo tipo con un set de operaciones que pueden ser usadas sobre sus instancias para manipular los objetos.**
- Las **operaciones son descriptas como un set de rutinas que establece el programador y se especifican en la declaración del nuevo tipo.**

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

• Tipos de Datos Abstractos (TAD):

- Cada TAD define un conjunto de operaciones permitidas (público), pero oculta los detalles de implementación interna (privado).
- No hay ligadura por defecto, el programador debe especificar la representación y las operaciones!
- TAD comunes: *Listas, colas, pilas, arboles, grafos*, etc...

Esto se verá en más detalle en otra Clase

TIPOS ABSTRACTOS (EJEMPLO EN C++)

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

Class

*Estructura de
datos interna
(privada)*

```
class stack_of_char{  
    int size;      máximo tamaño  
    char* top;     puntero al top  
    char* s;       puntero a inicio lista
```

Las operaciones
están ocultas y
protegidas del
acceso no
autorizado.

*Comportamiento
Rutinas
(pública)*

Operaciones que
son accesibles
desde fuera del
TAD

```
public:  
    stack_of_char (int sz) {          rutina construir  
        top = s = new char [size =sz];  
    }  
    ~stack_of_char ( ) {delete [ ] s;}  rutina destruir  
    void push (char c) {*top++ = c;}    insertar nuevo al top  
    char pop ( ) {return *--top;}       extraer un elemento  
    int length ( ) {return top - s;}    retornar tamaño  
};
```

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

◦ Momentos de ligadura de Variable-Tipo

1. Estático (static typing)

- en traducción/compilación

2. Dinámico (dynamic typing)

- en ejecución

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

Momento de ligadura variable-tipo - Estático

- La ligadura entre la variable y su tipo se especifica en la declaración
- Se liga en compilación y NO puede cambiarse en ejecución. De esta forma hay protección
- El chequeo de tipo también será estático.

**Ejemplos: Pascal, C, C++, JAVA, Fortran,
COBOL, Algol, Simula, ADA, etc**

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

Momento de ligadura variable-tipo - Estático

- o Declarar previamente las variables de un tipo hace que sean automáticamente protegidas de una operación ilegal
- o el compilador puede detectar “violaciones de la semántica estática” concernientes a variables y sus tipos.
- o Realizar el “chequeo de tipos estático” antes de la ejecución contribuye a la **detección temprana de errores** y a mejorar la **confiabilidad** del programa.

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

Momento de ligadura variable-tipo - Estático

- La ligadura puede ser realizada en forma:
 1. Explícita
 2. Implícita
 3. Inferida

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

○ Momento Estático – Explícito

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

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

La **ventaja** de las reside en la **claridad** de los programas y en una mayor **confiabilidad**, porque cosas como **errores ortográficos en nombres de variables** pueden **detectarse en tiempo de traducción/compilación**.

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

○ **Momento Estático - Implícito**

- Se **deduce por "reglas propias del lenguaje"**.
- Esto **ocurre sin que el programador tenga que especificar explícitamente el tipo de datos de la variable.**
- **Lo define cada lenguaje**

Ej. Fortran 77:

- ✓ **variables que empiezan con 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

- No se requiere que el TIPO de la variable sea declarado. El tipo se deduce/infiere automáticamente de los tipos de sus componentes.
- Se basa en el contexto del código y en el valor asignado a la variable de una asignación.
- El enlace se realiza en etapa de traducción
- ✓ Si no está definido se infiere por componentes
- ✓ Si no puede inferir da error en compilación
- ✓ Aplica en general a Lenguajes Funcionales.

Ejemplos


- `var sum = 0` se puede **inferir** que **sum** es entero
- En Swift: `var nombreCliente = "Pedro"` **infiere** **string**

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

○ Momento de ligadura variable-tipo - Dinámico

- El tipo se liga a la variable en ejecución y puede modificarse.
- **Cambia** cuando se le **asigna un valor** mediante una **sentencia de asignación (no declaración)**

python

 Copy code

```
x = 10    # x se inicializa como un entero
x = "Hola"  # Ahora x se convierte en una cadena de texto
x = [1, 2, 3]  # x ahora es una lista

print(x)   # Imprimirá la lista [1, 2, 3]
```

No se detectan incorrecciones de tipo en las asignaciones. El tipo de la parte izquierda simplemente cambia al tipo de la derecha

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

○ Momento – Dinámico (Problemas)

- El costo de implementación de la ligadura dinámica es mayor, mucho tiempo de ejecución por:
 1. *comprobación de tipos*
 2. *mantenimiento de tablas de Descriptores* asociado a cada variable en el que se almacena el tipo actual
 3. *cambio en el tamaño de la memoria* asociada a la variable (ya que puede ser de cualquier tipo)
- Chequeo dinámico
- Menor legibilidad y más posibilidad de errores

Los lenguajes INTERPRETADOS en general adoptan ligadura dinámica de tipos

APL, Snobol, Javascript, Python, Ruby, etc

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

L-VALUE de una variable:

- Es la dirección del área de memoria ligada a la variable durante la ejecución.
- Las instrucciones de un programa **acceden** a la variable por su L-Valor.

Las variables se alocan en un área de memoria. Ese área de memoria debe ser ligada a la variable en algún momento. En esa área de memoria se almacenará su valor al cual necesito llegar. Para ello uso el L-Valor

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

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

Es el Periodo de tiempo en que existe la ligadura

El tiempo de vida está muy ligado al L-Valor.

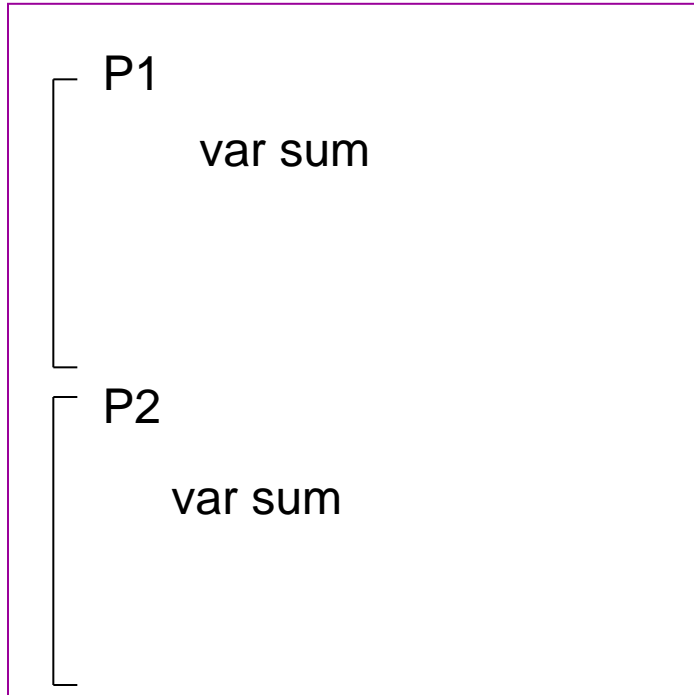
- ✓ Es el tiempo en que está alocada la variable en memoria y en el cual el binding existe.
- ✓ Es desde que se solicita hasta que se libera

- **Alocación de memoria**

**Momento en que se reserva la memoria
para una variable**

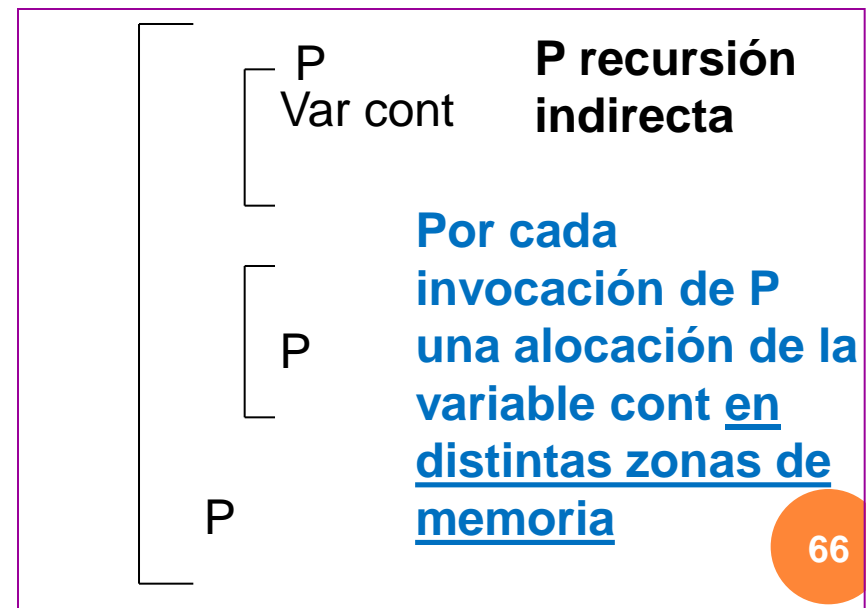
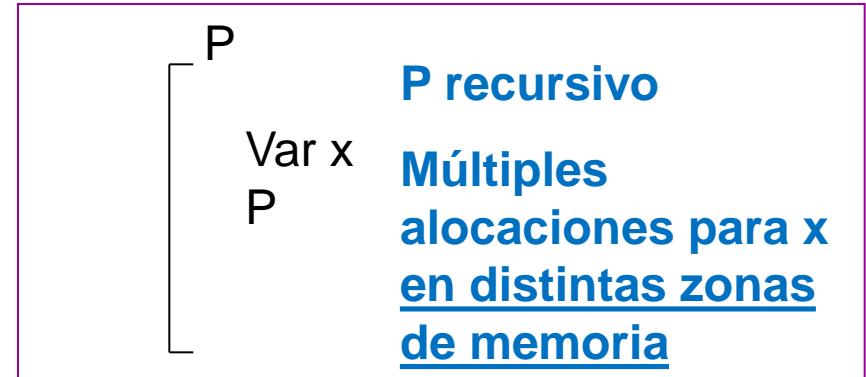
EJEMPLOS DE ALOCACIONES – TIEMPO DE VIDA

recursión



**2 alocaiones diferentes para sum en
distintas zonas de memoria:**

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



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

Tipos de Momentos de Alocación de memoria:

- **Estático**
- **Dinámico**
- **Persistente**

El tipo dependerá del lenguaje

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

Momentos – Alocación de memoria

- **Estático:** 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** (sensible a la historia)
- **Dinámico:** se hace en tiempo de ejecución.
 1. **Automática:** cuando **aparece una declaración de una variable en la ejecución**
 2. **Explícita:** requerida por el programador con una **sentencia de creación, a través de algún constructor** (*por ej.. algún puntero*)

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

Momentos - Alocación

- **Persistente:** Los objetos persistentes que 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/abiertos 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**

Todo esto se verá más adelante en más detalle

EJEMPLOS DE ALOCACIONES

File 1

```
int x;
```

```
static int n;
```

```
int func1()
```

```
{
```

```
    extern int i;
```

```
    float z; ← Variable INTERNA
```

```
    static int m;
```

```
}
```

```
..
```

```
int main()
```

```
{
```

```
    .....
```

```
}
```

File 2

```
extern int x;
```

```
....
```

```
int i;
```

```
extern static int n;
```

```
float func2()
```

```
{
```

```
    int x
```

```
    int z;
```

```
}
```

```
.....
```

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

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 ubicación asociada a la variable (l-valor)**
- La codificación se **interpreta** de acuerdo con el **tipo de la variable**

Ejemplo:

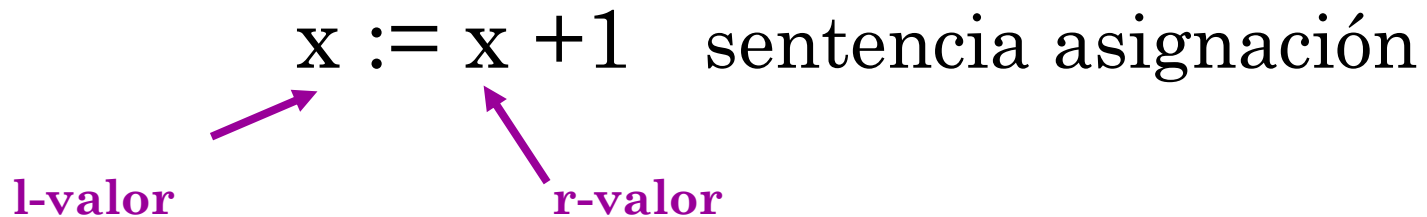
R-Valor: 01110011

que está **almacenado** en una ubicación de memoria

- **interpreta un nro. entero** si la variable es **tipo int**;
- **interpreta una cadena** si la variable es **tipo char**;

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

- Objeto: (l-valor, r-valor) (dirección memoria, valor)



- Se accede a la variable por el l-valor (ubicación)
- Se puede modificar el r-value (valor) (salvo un caso especial)

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

Momentos de ligadura - variable a valor:

Binding Dinámico de una variable a su valor

- el valor (*r-valor*) puede cambiar durante la ejecución con una asignación.
- Constante: el valor (*r-valor*) no puede cambiar si se define como *constante* simbólica definida por el usuario
- $b := a$ (copia el *r-valor* de *a* en el *l-valor* de *b* y cambia el *r-valor* de *b*)
- $a := 17$ (asigna un valor directamente)
- Constante: se congela el valor

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

Momentos de ligadura - variable a valor:

Binding Dinámico: varía según los lenguajes

- Común en lenguajes imperativos (Fortran, C, C++, Pascal, ADA).
- Los lenguajes de programación funcional y lógica pueden vincular un valor mediante el proceso de evaluación, pero una vez establecida la vinculación, no se puede cambiar durante el tiempo de vida de la variable.

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

Constante Simbólica Definida por el Usuario:

- Permitida por algunos lenguajes.
- Su valor se congela

Const pi = 3.1416

*Circunferencia=2*pi*radius*

**El traductor daría error si se quisiera
modificar a pi**

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

Momentos de ligadura - constante a valor:

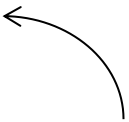
Binding time: varía según los lenguajes

- PASCAL: el valor que proporciona una expresión debe evaluarse en tiempo de compilación (*binding time es compile time*). El compilador puede sustituir legalmente el valor de la constante por su nombre simbólico en el programa.
- C y ADA: permite que el valor se pueda dar como una expresión que involucra otras variables y constantes, en consecuencia, la ligadura sólo puede establecerse en tiempo de ejecución, cuando la variable es creada.

Veamos algunos ejemplos

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

```
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;
```



- NO da ERROR
- El binding del r-valor con la variable es en ejecución.
- Toma x=4 en ejecución.
- ADA permite primero asignar x y luego constante

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

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 }
```

La ligadura de su r-valor con la variable la hace en tiempo 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> EJEMPLO EN PASCAL

main.pas

```
1 { Online Pascal Compiler.
2 | Code, Compile, Run and Debug Pascal program online.
3 | }
4 program Constantes;
5 var
6   i: integer;
7 function prueba(): integer;
8   const x: integer = 9 + i;
9 begin
10   prueba := x;
11 end;
12
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>

Inicialización de una variable

◦ ¿Cuál es el r-valor luego de crearse una variable?

Los lenguajes y sus versiones implementan **diversas** estrategias de inicialización:

1. Inicialización por defecto:

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

2. Inicialización en la declaración:

C `int i = 0, j = 1`

ADA `I, J INTEGER := 0`

opcional

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

Inicialización de una variable

○ ¿Qué pasa si no es inicializada?

3. Estrategia Ignorar el problema:

- Toma como valor inicial lo que hay en memoria (la cadena de bits asociados al área de almacenamiento)
- Puede llevar a errores y requiere chequeos adicionales!

ALGUNAS CONSIDERACIONES ADICIONALES

VARIABLES ANÓNIMAS (SIN NOMBRE) Y REFERENCIAS - PUNTEROS

- Algunos lenguajes permiten que variables sin nombre sean accedidas por el r-valor de otra variable.
- Ese r-valor se denomina *referencia o puntero a la variable*
- La *referencia* puede ser al r-valor de una variable nombrada (access path 0), o al de una variable referenciada con un access path de longitud arbitraria (según cantidad de referencias)

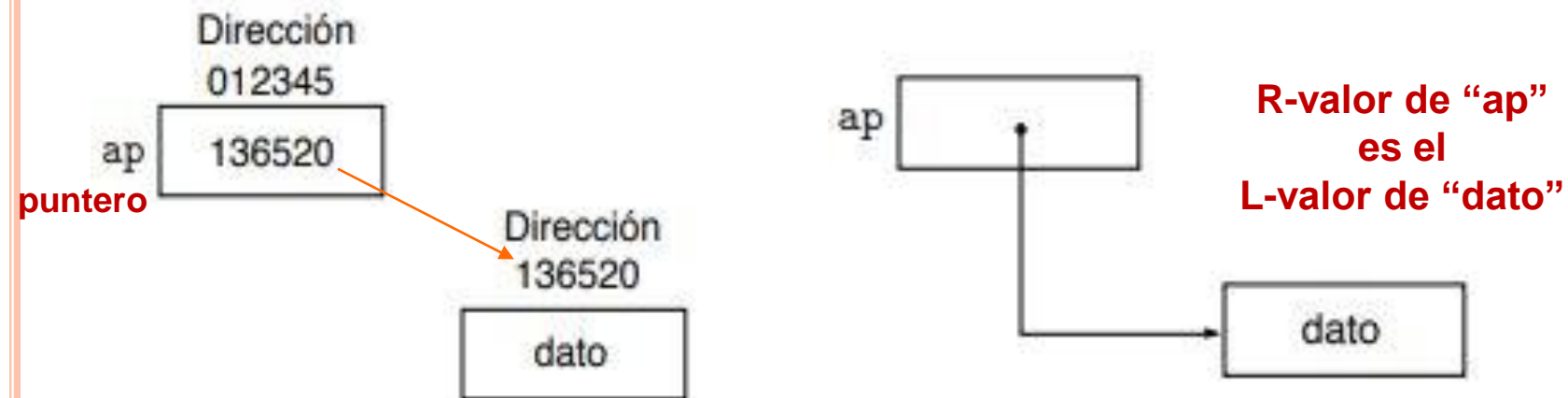
El r-valor de una variable será la referencia al l-valor de otra variable

VARIABLES ANÓNIMAS (SIN NOMBRE) Y REFERENCIAS - PUNTEROS

- PUNTERO:** variable que sirve para señalar la posición de la memoria en la que se encuentra otro dato almacenando como valor, con la dirección de ese dato.

Conviene imaginar gráficamente este mecanismo.

Ejemplo de variable **puntero** **ap**, almacenada en la **dirección 012345**, y la **dirección 136520** celda de memoria que contiene la **variable/dato** a la que apunta.



VARIABLES ANÓNIMAS (SIN NOMBRE) Y REFERENCIAS – PUNTEROS – PASCAL

main.pas

```
1 {  
2  
3  
4  
5  
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.
```

Declaro PI tipo puntero a enteros

¿En qué se diferencian esas 2 variables?

EN SU CONTENIDO: número o dirección

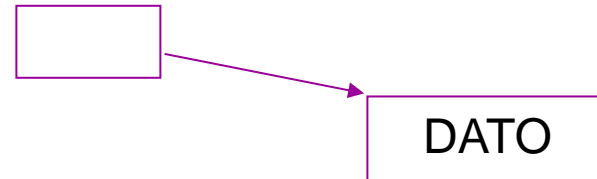
- Con uno accedo al dato directo
- Con otro accedo al dato indirectamente

New le asigna una dirección de memoria en la HEAP a una variable anónima a la cual accedo por puntero

Modifico la dirección de memoria referenciado por punt (punt^) y la pongo en 7

VARIABLES ANÓNIMAS (SIN NOMBRE) Y REFERENCIAS - PASCAL

Ejemplo de Puntero a Puntero



type pi = ^ integer;

Declaro pi puntero a tipo de dato entero

var pxi:pi;

Declaro variable puntero pxi de tipo pi

new (pxi);

Aloca memoria a variable anónima

pxi^:=0;

el valor de la variable referenciada se establece en 0.

Para acceder al objeto sin nombre referenciado por pxi, es necesario usar **un operador de desreferenciación (^)**, que se puede aplicar a una variable puntero para obtener su **r_value**, es decir, el **l_value** del objeto referenciado.

VARIABLES ANÓNIMAS (SIN NOMBRE) Y REFERENCIAS - PASCAL

Ejemplo de Puntero a Puntero



```
type pi = ^ integer;
```

Declaro pi puntero a tipo de dato entero

```
var pxi:pi;
```

Declaro variable puntero pxi de tipo pi

```
new (pxi);
```

Aloca memoria a variable anónima

```
pxi^:=0;
```

el valor de la variable referenciada se establece en 0.

```
type ppi = ^pi;
```

Declaro puntero ppi tipo de dato puntero

```
var ppxi: ppi;
```

Declaro variable puntero ppxi de tipo ppi

```
new(ppxi);
```

Aloca la ubicación de memoria de la variable anónima

```
^ppxi:=pxi;
```

Asigna la dirección de pxi

VARIABLES ANÓNIMAS (SIN NOMBRE) Y REFERENCIAS – PUNTEROS Y ALIAS

Alias:

se da si hay **variables comparten un objeto** en el mismo **entorno de referencia**. Sus **caminos de acceso conducen al mismo objeto**.

Por lo tanto el **objeto compartido modificado por uno se modifica para todos los caminos**

DISTINTOS NOMBRES  **1 ENTIDAD**

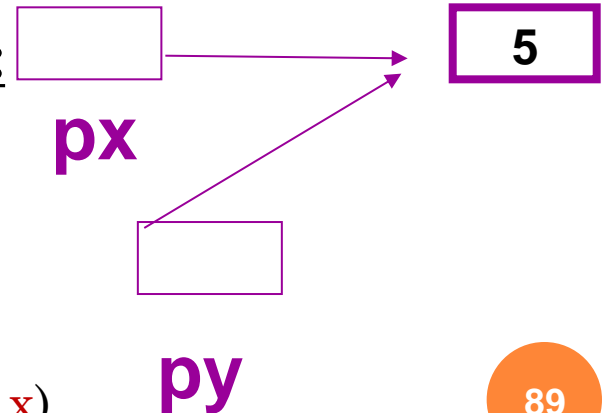
○ Ejemplo: C apuntando a variable:

○ `int x = 5;`

○ `int*px,`

○ `px = &x ;` asiga la dirección de x

○ `py =px ;` asiga el r-valor de px (la dirección de x)



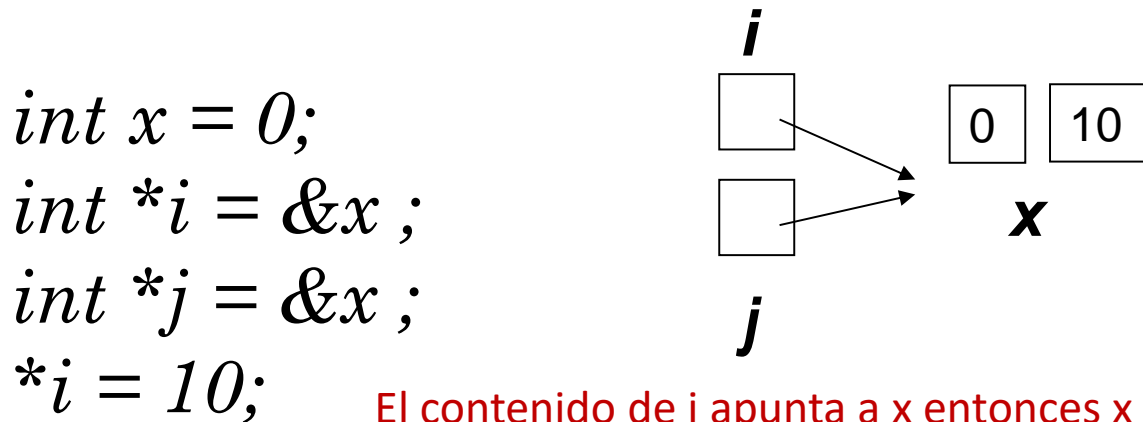
VARIABLES ANÓNIMAS (SIN NOMBRE) Y REFERENCIAS – PUNTEROS Y ALIAS

Ventajas del uso del alias:

- **Compartir objetos** se utiliza para mejorar la **eficiencia**.

Desventajas uso del alias:

- Generan **errores**, el valor de una variable se puede **modificar incluso cuando no se utiliza su nombre**.
- Generan **programas** que son **difíciles de leer y de encontrar error**



El contenido de *i* apunta a *x* entonces *x* se convierte en 10

Efecto colateral: todos quedan en 10
Con modificación indirecta de una variable



CONCEPTO DE SOBRECARGA Y ALIAS

- **Alias**

distintos nombres → 1 entidad

- **Sobrecarga**

1 nombre → **distintas entidades**

CONCEPTO DE SOBRECARGA

Sobrecarga:

Característica de algunos lenguajes de programación que permiten definir **comportamientos personalizados para operadores, métodos, funciones.**

Un mismo nombre que realiza algo distinto según el contexto.

Un nombre está sobrecargado si en un momento referencia más de una entidad

- **Debe estar permitido por el lenguaje.**
- **Debe haber suficiente información para permitir establecer la ligadura unívocamente.** (por ejemplo determinarlo por su tipo)

CONCEPTO DE SOBRECARGA

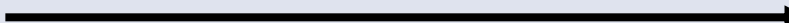
int i,j,k;
float a,b,c;
.....
i = j + k ;
a = b + c;

¿Qué sucede con el operador **+**?

¿Qué hace el lenguaje?

- Suma enteros, flotantes
- Si hay strings los concatena o da error?

Los tipos de las variables, los parámetros en las funciones() son los que permiten que se desambigüe en compilación.

1 nombre  1 entidad
No hay ambigüedad